Rational Environment Training—
Large-System Development

Document Control Number: 1032
Rev. 0, September 1986
Rev. 1.0, March 1987

This document subject to change without notice.

Rational
1501 Salado Drive
Mountain View, California 94043

Rational Environment Training—
Large-System Development

Slides

Contents

Concepts of Subsystems

Construction and Modification of Subsystems

Additional Topics

RATIONAL

Seminar Outline

Concepts of Subsystems

- ● Introduction
  - Key Concepts
  - Subsystem Structure
  - Traversal
  - Execution

Construction and Modification of Subsystems

Additional Topics

Seminar Objectives

- Introduce concepts and mechanisms of Rational Subsystems
- Provide experience in modifying, testing, and releasing systems using subsystem facilities

Seminar Materials

- *Rational Environment Training — Large-System Development*
  - — Seminar slides
- *Rational Environment Project Management*
  - — Introduction
  - — Packages Activity, History, View, and Check
- *Rational Environment Reference Summary*
- *Rational Environment Basic Operations*

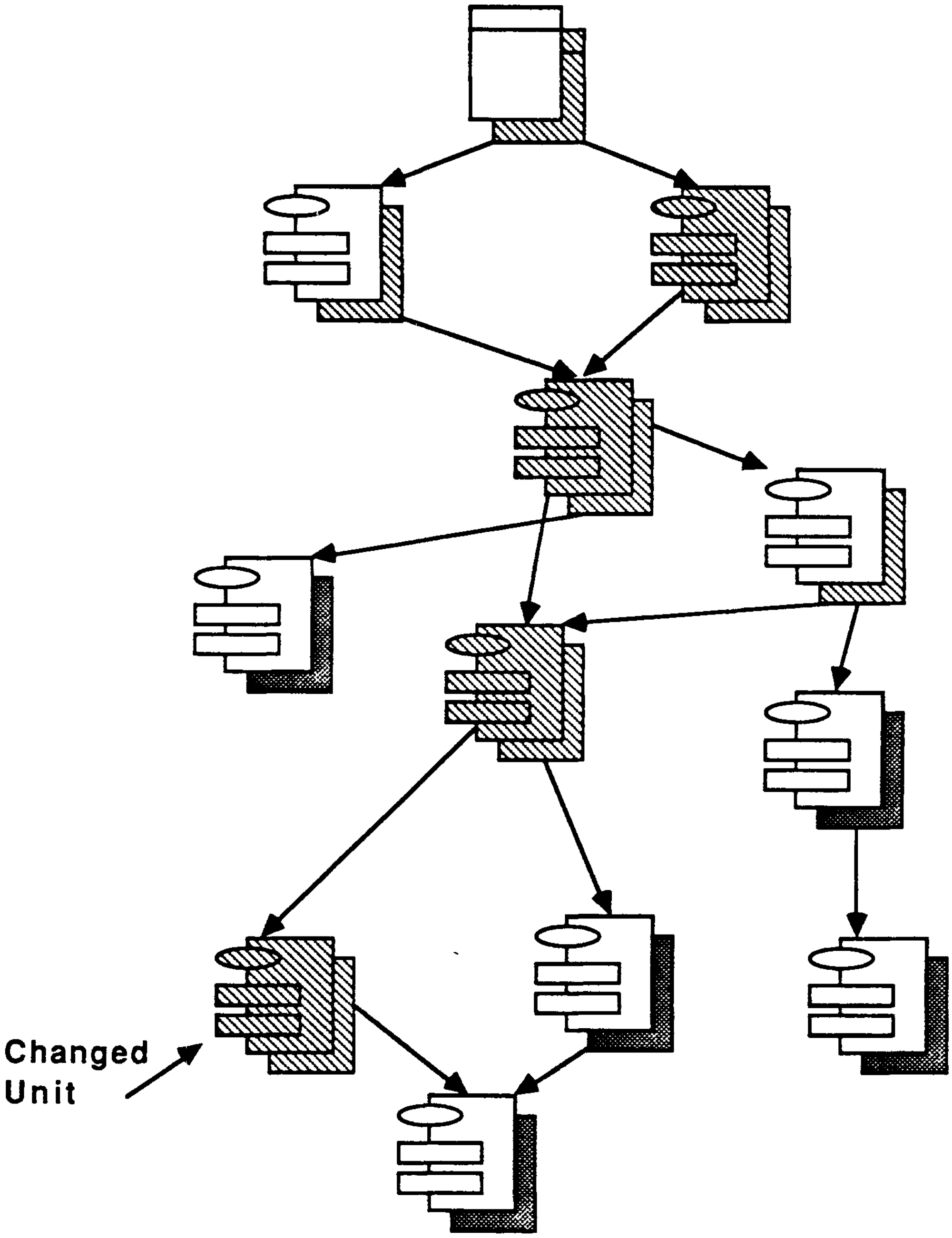
Motivations for Rational Subsystems

- Subsystems originated with Rational's experience in developing the Environment
- Existing development facilities were discovered to be inadequate in
    - Reducing the time and costs of making and testing changes to a large Ada system
    - Managing the complexity of the project and design
    - Supporting parallel team development and testing
    - Supporting multihost and multisite development

Changes to Ada Systems

- Recompilation is required to verify the correctness of changes
- Recompilation may not be limited to the changed unit
- Recompilation time can become a major factor in development delays

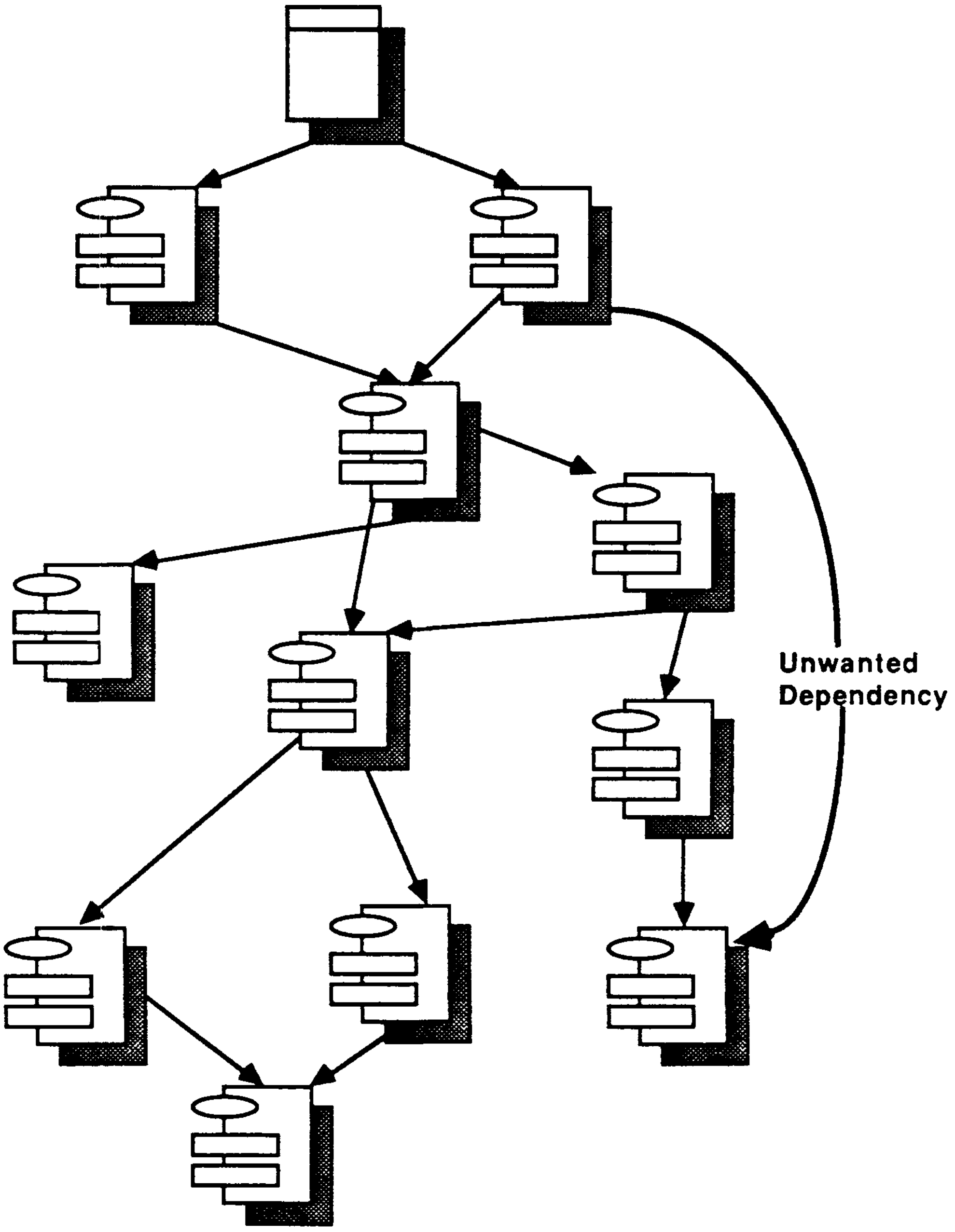
Changes to Ada Systems, cont.



Design Degradation

- Dependencies in a system reflect part of the overall design
- Unwanted dependencies are easily added by any developer inserting a *with* clause
- Conventional library facilities have no safeguards for the integrity of the design

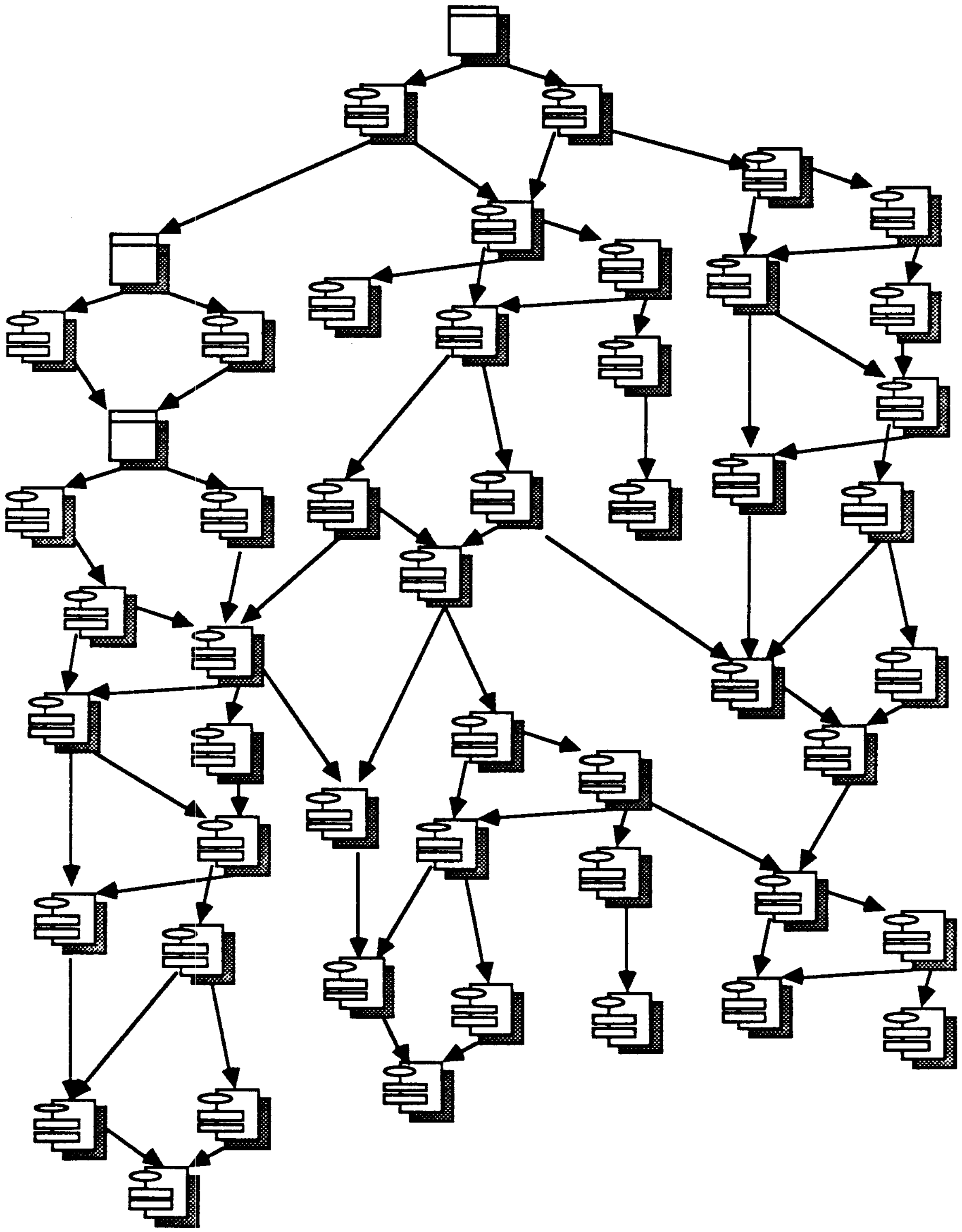
Design Degradation, cont.



Large Ada Systems

- Difficult to understand the application by the picture
- Difficult to reason about the dependencies
- Recompilation can take hours or days
- Difficult to allow individuals to develop in parallel because of Ada's strong dependencies
- Difficult to partition for individual developers to implement

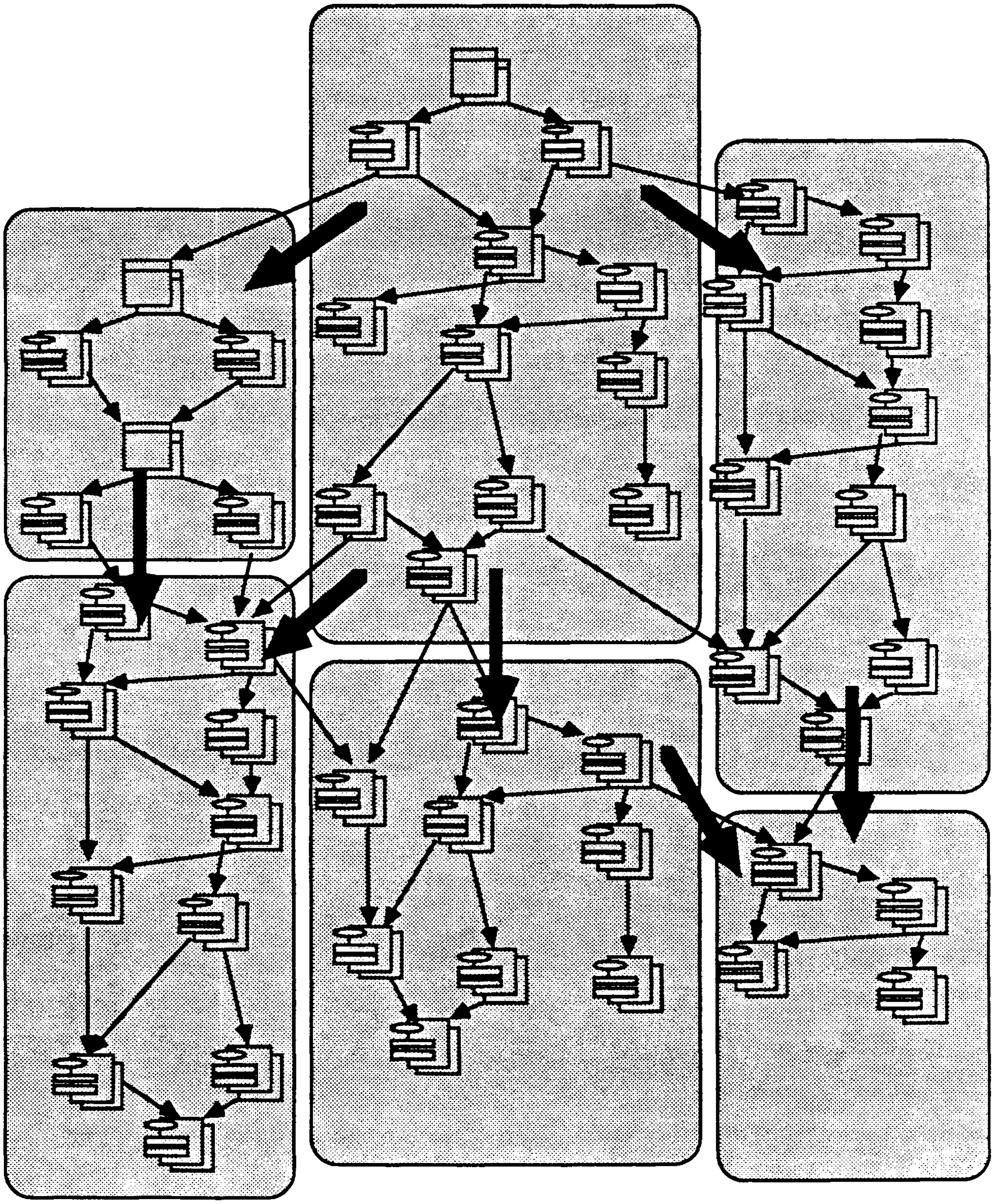
Large Ada Systems, cont.



Partition of Large Systems

- Subsystem partitioning improves understanding of the application
- Dependencies can be defined at the subsystem level
- Dependencies between subsystems can be enforced through tool enforcement
- Each subsystem can be assigned to an individual developer or implementation team
- Subsystems provide the opportunity to firewall compilation through use of closed private parts

Partition of Large Systems, cont.



Rational Subsystems

- Provide designers and project managers with a powerful decomposition and structuring mechanism
- Provide enforcement of design decisions
- Reduce time to make and test changes by minimizing recompilation requirements
- Facilitate multihost, multisite development
- Allow parallel development and testing

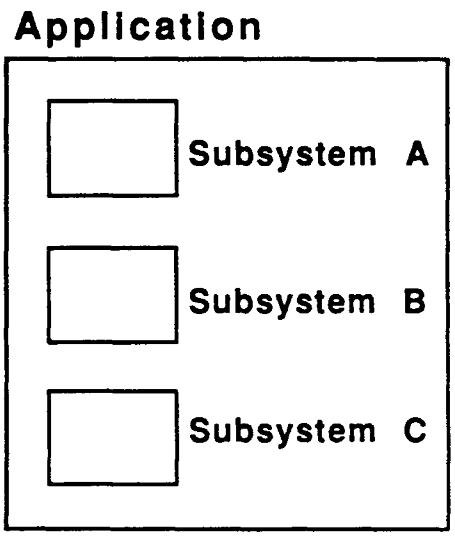
Seminar Outline

Concepts of Subsystems
- Introduction
- ● Key Concepts
- Subsystem Structure
- Traversal
- Execution

Construction and Modification of Subsystems

Additional Topics

System Structure

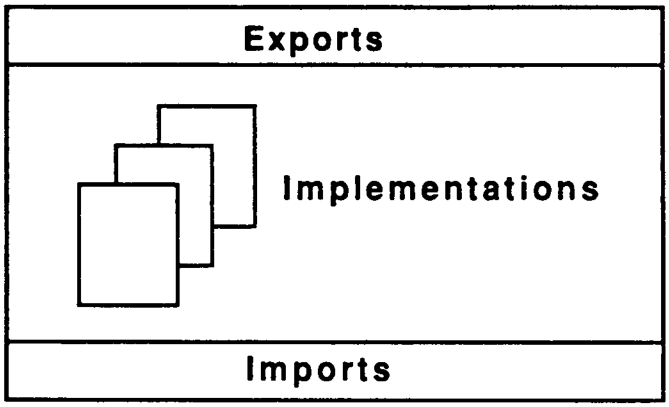


- An application is an entire software system that can be composed of subsystems
- An application can contain any number of subsystems

Subsystems

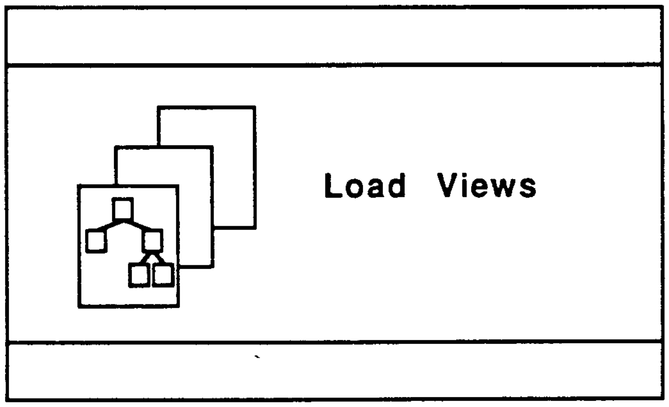
- Each subsystem should be a complete logical component of the application
- Each subsystem contains
  - A series of implementations
  - Resources needed in this subsystem (imports)
  - Resources made available to other subsystems (exports)
  - Historic logs about creation and modification of the subsystem
  - Optional test information or documentation

Subsystems, cont.



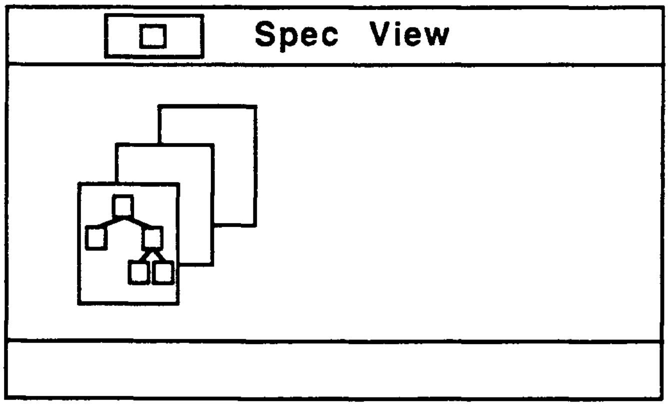
Implementations

- Are called *load views*
- Define a logical part of the entire application
  - — Contain the Ada units
  - — Are analogous to Ada package bodies
  - — Are a specific instance of a subsystem
- Are generally derived from a previous load view



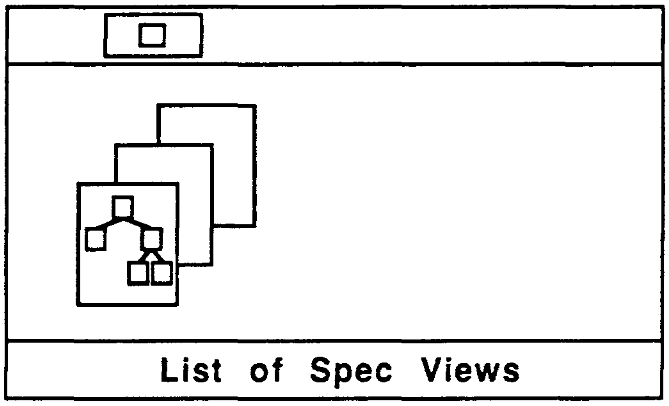
Exports

- Are called *spec views*
- Define the interface between subsystems
  - — Contain a subset of units in a subsystem that are visible for other subsystems to import
  - — Are analogous to Ada package specifications
  - — Are used in compiling other subsystems
- Have one or more corresponding load views



Imports

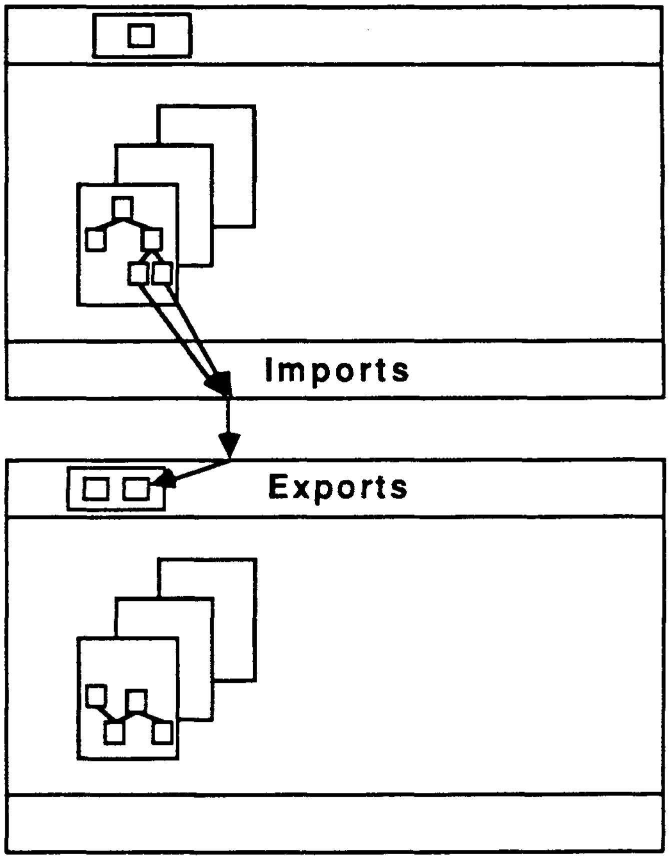
- Define the set of spec views that this subsystem can use
- Are analogous to adding *with* clauses to an Ada package



Dependencies between Subsystems

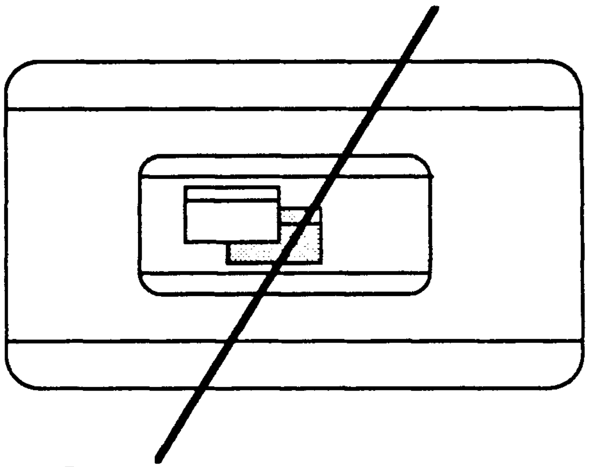
- Spec views (exports) define the units of a subsystem that are available for importing into another subsystem
- Imports define the set of spec views that a subsystem can use
- Ada units in a load view can reference units in the spec view of another subsystem if they have been imported

Dependencies between Subsystems, cont.

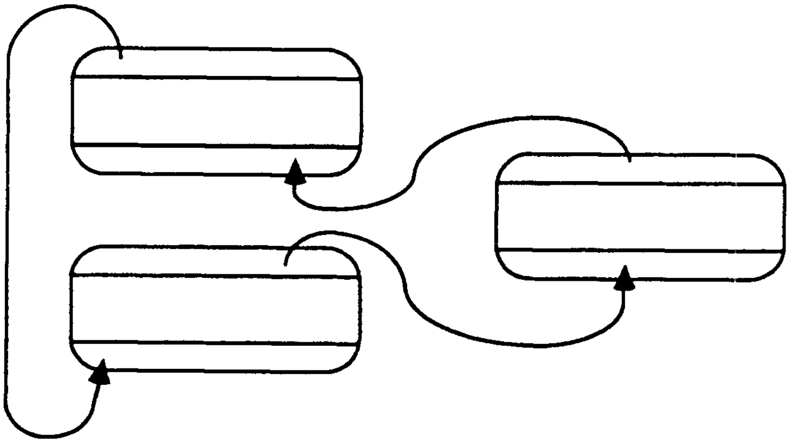


Dependency Restrictions

- Subsystems cannot be nested



- Circular dependencıes between subsystems cannot exist



Dependency Restrictions, cont.

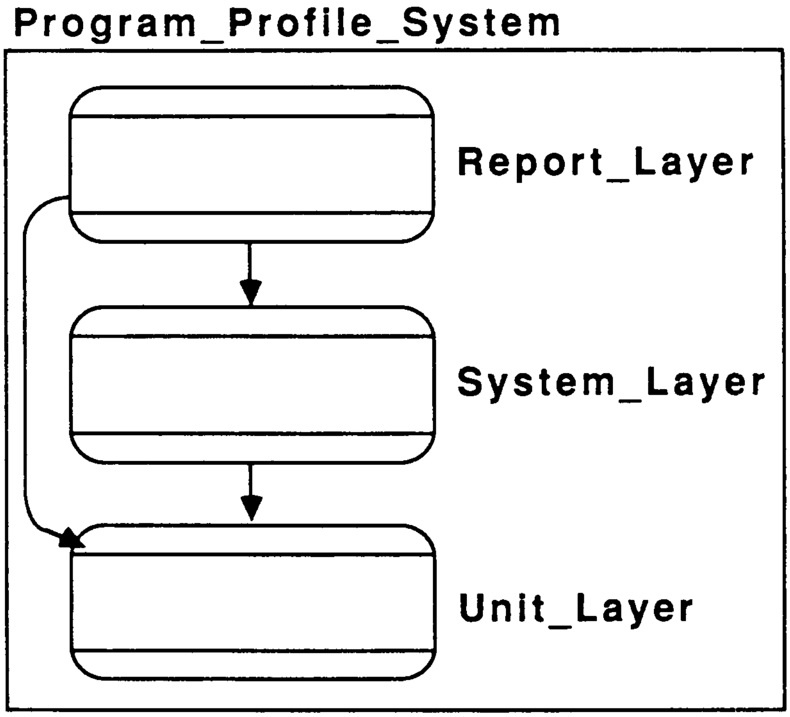
- Library-level generic instantiations cannot be exported

  - Example:

```
with List_Generic;
package Complex_List is new List_Generic (Float);
```

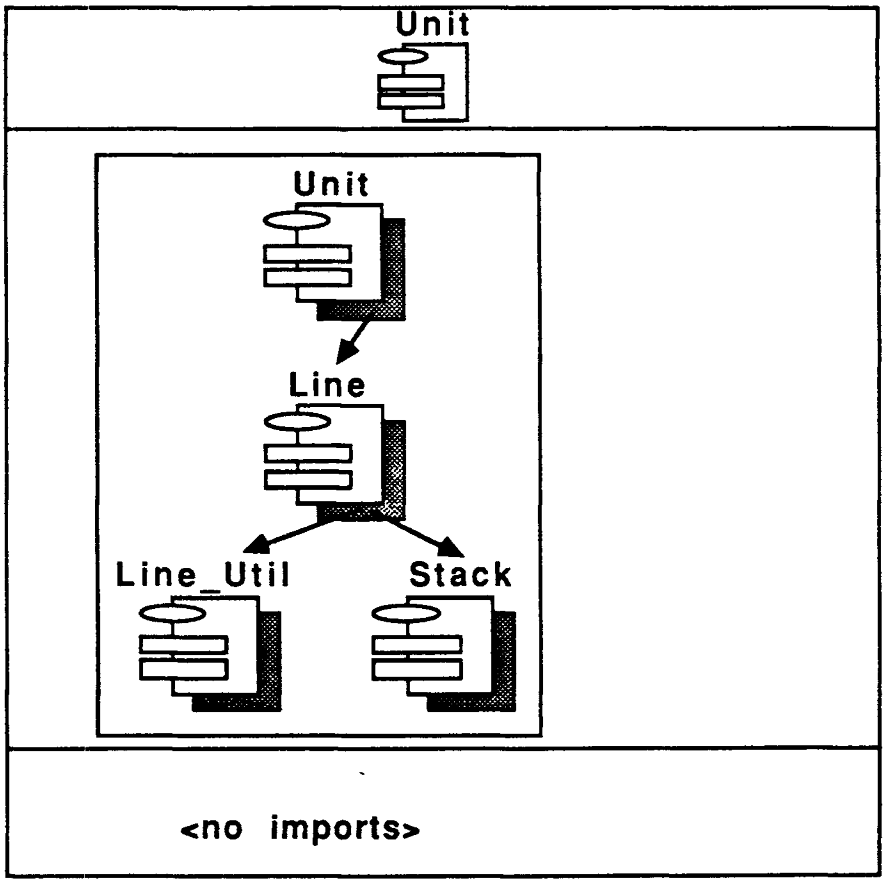
Sample Application

- Application counts various kinds of lines in an Ada program
- Application consists of three subsystems



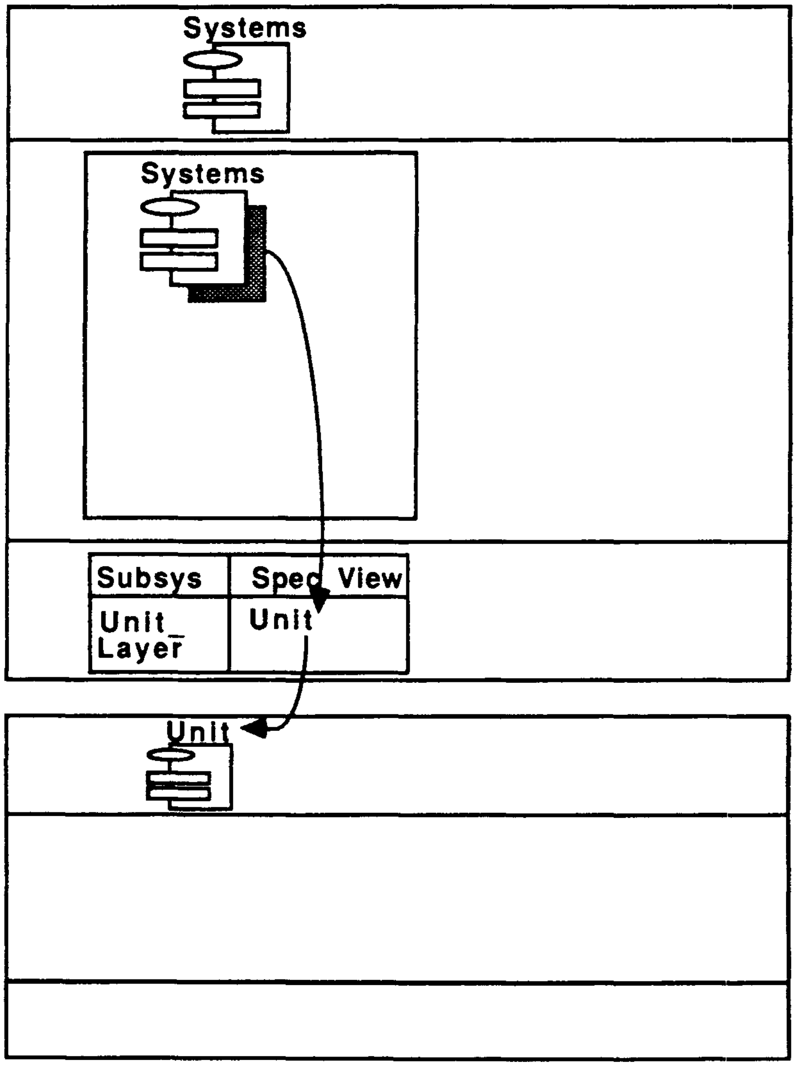
Sample Application, cont.

- `Unit_Layer` contains the Ada units that analyze each line and collect the statistics for an Ada unit



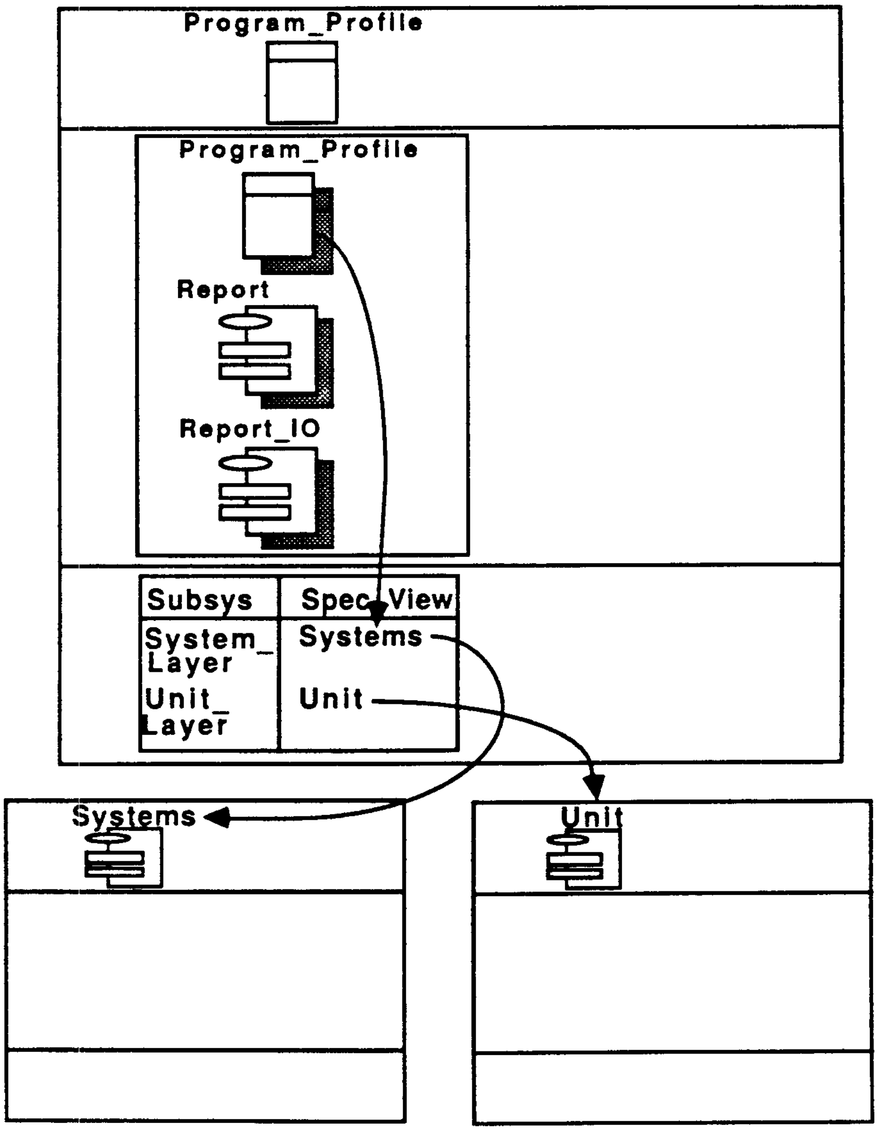
Sample Application, cont.

- `System_Layer` contains the Ada units that determine the set of Ada units to be analyzed and collects statistics for all units



Sample Application, cont.

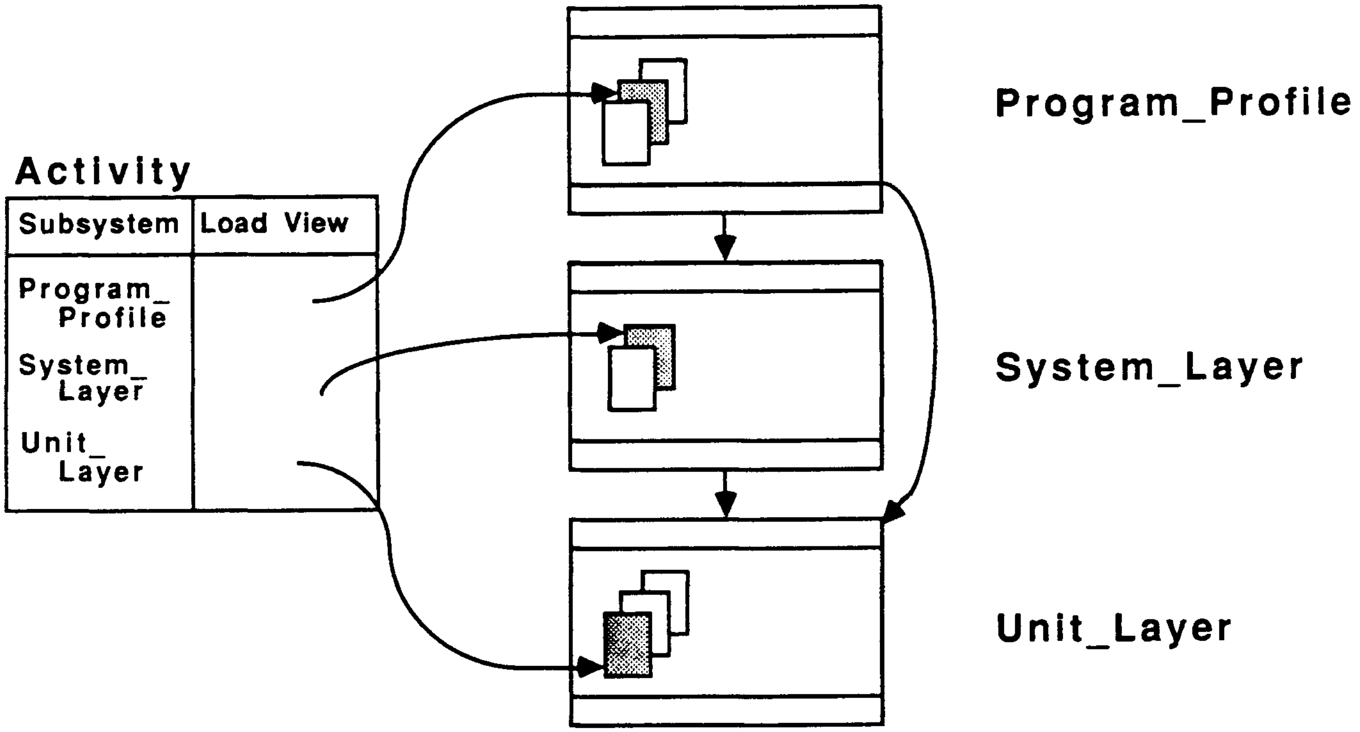
- `Report_Layer` contains the Ada units that format the output and provide the user interface and the main driver



Subsys	Spec_View
System_Layer	Systems
Unit_Layer	Unit

Execution of an Application

- Requires specifying the load view to use for each subsystem via *activities*
- Executes as does any other program once an activity is specified



Activity

Subsystem	Load View
Program_ Profile	
System_ Layer	
Unit_ Layer	

Seminar Outline

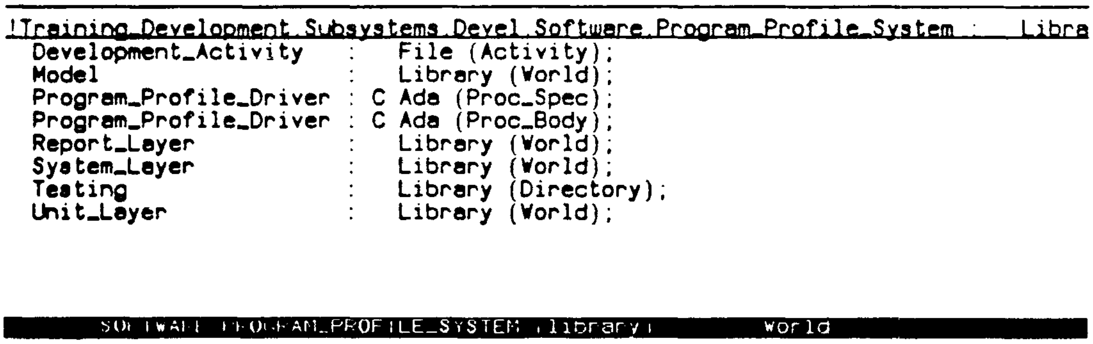
Concepts of Subsystems
Introduction
Key Concepts
● Subsystem Structure
Traversal
Execution

Construction and Modification of Subsystems

Additional Topics

Application Notation

- Each application is typically a world containing
  - A world for each subsystem
  - Optional project-level libraries for documentation
  - An optional main driver
  - Optional activities
- Example



```
!Training_Development.Subsystems.Devel.Software.Program_Profile_System :    Libra
  Development_Activity    :   File (Activity);
  Model                   :   Library (World);
  Program_Profile_Driver  : C Ada (Proc_Spec);
  Program_Profile_Driver  : C Ada (Proc_Body);
  Report_Layer            :   Library (World);
  System_Layer            :   Library (World);
  Testing                 :   Library (Directory);
  Unit_Layer              :   Library (World);


        SOFTWARE PROGRAM_PROFILE_SYSTEM (library)        World
```

Subsystem Notation

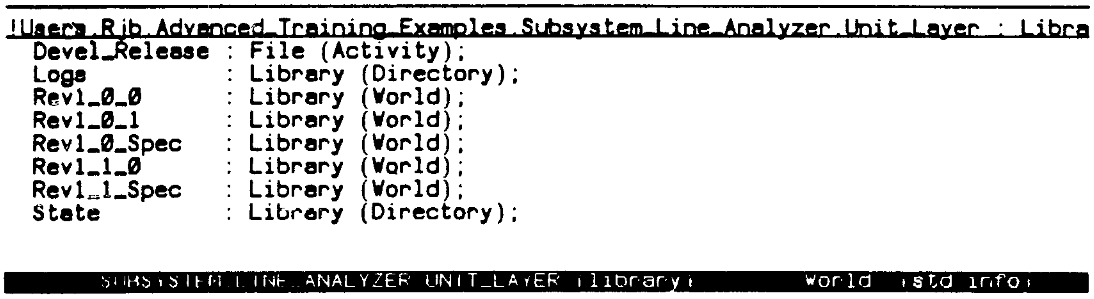
- Each subsystem is a world containing
  - A Logs directory for logs from each command that manipulates the subsystem
  - A State directory for information about the genealogy of each view in the subsystem
  - A world for each spec or load view
  - Optional libraries for tests or documentation
  - Optional activities
- The Environment creates Logs and State directories and spec and load view worlds
- The user supplies test or documentation libraries and activities

View Names

- Load view names are created from two parts: a base name and a set of release numbers
- Example with load view name of `Rev1_0_2`
  - `Rev1` is the base name
  - `_0_2` is the pair of release numbers
- Spec view names are created from load view names by removing the last `_N` and appending `_Spec`
- Example
  - Load view name is `Rev1_0_2`
  - Corresponding spec view name is `Rev1_0_Spec`

Subsystem Notation Examples

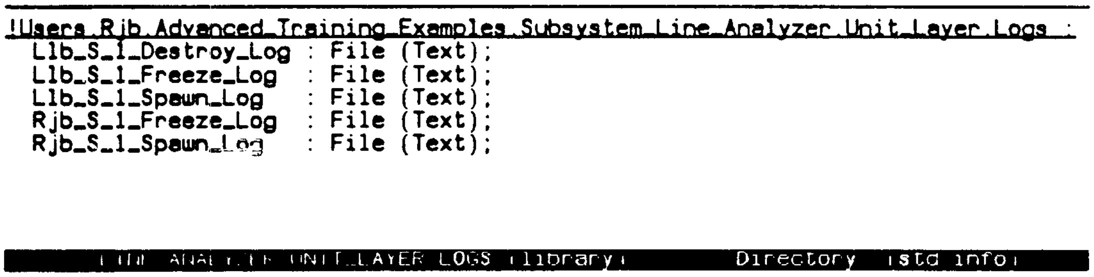
- Subsystem library



```
!Users.Rjb.Advanced_Training_Examples.Subsystem_Line_Analyzer.Unit_Layer : Libra
  Devel_Release  : File (Activity);
  Logs           : Library (Directory);
  Rev1_0_0       : Library (World);
  Rev1_0_1       : Library (World);
  Rev1_0_Spec    : Library (World);
  Rev1_1_0       : Library (World);
  Rev1_1_Spec    : Library (World);
  State          : Library (Directory);

        SUBSYSTEM.LINE_ANALYZER.UNIT_LAYER (library)        World  (std info)
```

- Logs directory in a subsystem listed by user and session

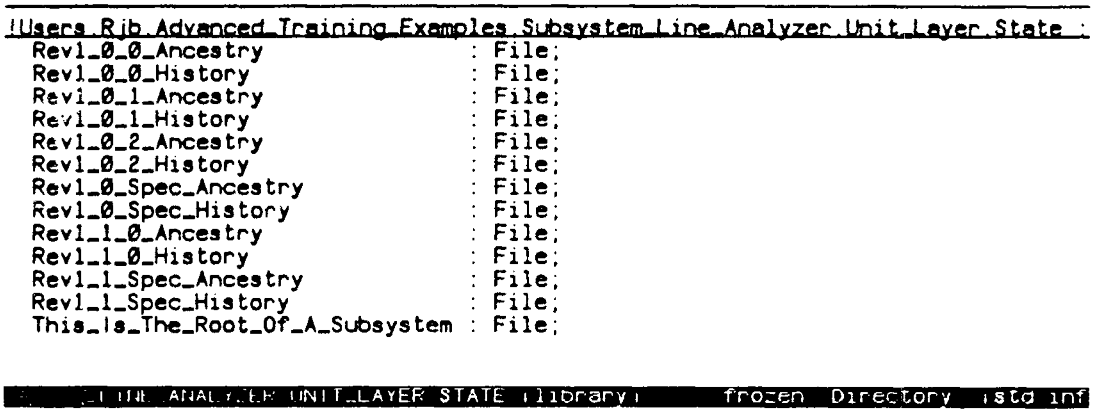


```
!Users.Rjb.Advanced_Training_Examples.Subsystem_Line_Analyzer.Unit_Layer.Logs :
  Llb_S_1_Destroy_Log : File (Text);
  Llb_S_1_Freeze_Log  : File (Text);
  Llb_S_1_Spawn_Log   : File (Text);
  Rjb_S_1_Freeze_Log  : File (Text);
  Rjb_S_1_Spawn_Log   : File (Text);

        LINE_ANALYZER.UNIT_LAYER.LOGS (library)        Directory  (std info)
```

Subsystem Notation Examples, cont.

- `State` directory in a subsystem listed by view



```
!Users.Rjb.Advanced_Training_Examples.Subsystem_Line_Analyzer.Unit_Layer.State :
  Rev1_0_0_Ancestry                 : File;
  Rev1_0_0_History                  : File;
  Rev1_0_1_Ancestry                 : File;
  Rev1_0_1_History                  : File;
  Rev1_0_2_Ancestry                 : File;
  Rev1_0_2_History                  : File;
  Rev1_0_Spec_Ancestry              : File;
  Rev1_0_Spec_History               : File;
  Rev1_1_0_Ancestry                 : File;
  Rev1_1_0_History                  : File;
  Rev1_1_Spec_Ancestry              : File;
  Rev1_1_Spec_History               : File;
  This_Is_The_Root_Of_A_Subsystem   : File;

      _LINE_ANALYZER UNIT_LAYER STATE (library)      frozen  Directory  (std inf
```

View Notation

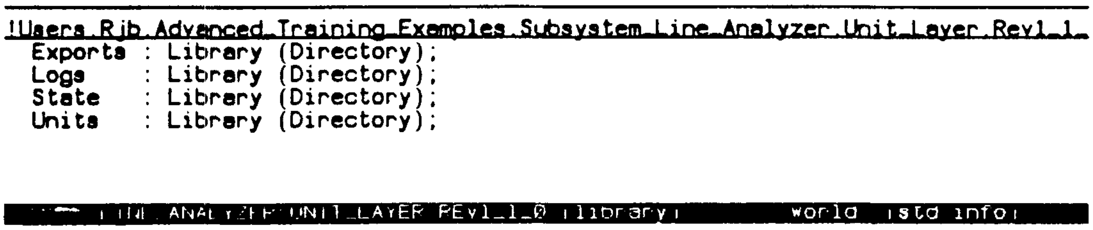
- Each spec or load view is a world containing
  - An `Exports` directory for specifying indirect files defining subsets of the units in spec views to be imported into other views
  - A `Logs` directory for logs from each command that manipulates the view
  - A `State` directory for files and activities used by subsystem commands for this particular view
  - A `Units` directory for the actual Ada code for the view
  - Optional libraries for tests or documentation

View Notation, cont.

- The Environment creates `Exports`, `Logs`, `State`, and `Units` directories
- The user supplies Ada units to be placed in the `Units` directory and optional test or documentation libraries

Examples of View Notation

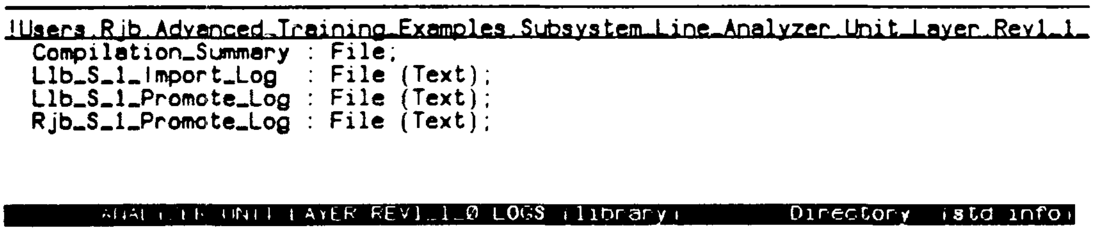
- View library



```
!Users.Rjb.Advanced_Training_Examples.Subsystem_Line_Analyzer.Unit_Layer.Rev1_1_
  Exports : Library (Directory);
  Logs    : Library (Directory);
  State   : Library (Directory);
  Units   : Library (Directory);

     LINE_ANALYZER_UNIT_LAYER.REV1_1_0 (library)         World  (std info)
```

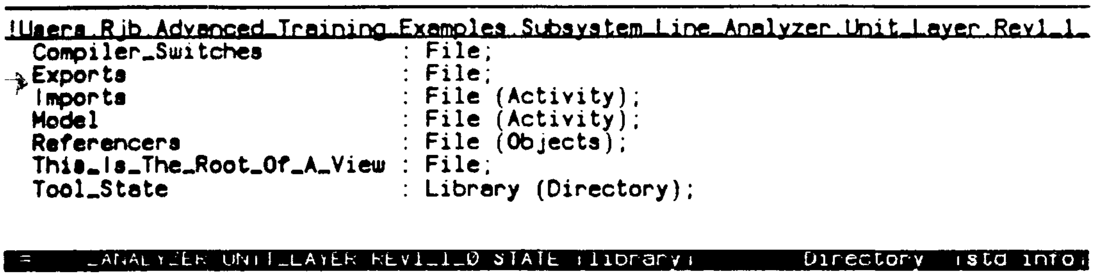
- **Logs** directory in a view



```
!Users.Rjb.Advanced_Training_Examples.Subsystem_Line_Analyzer.Unit_Layer.Rev1_1_
  Compilation_Summary : File;
  Lib_S_1_Import_Log  : File (Text);
  Lib_S_1_Promote_Log : File (Text);
  Rjb_S_1_Promote_Log : File (Text);

      ANALYZER_UNIT_LAYER.REV1_1_0.LOGS (library)        Directory  (std info)
```

- **State** directory in a view

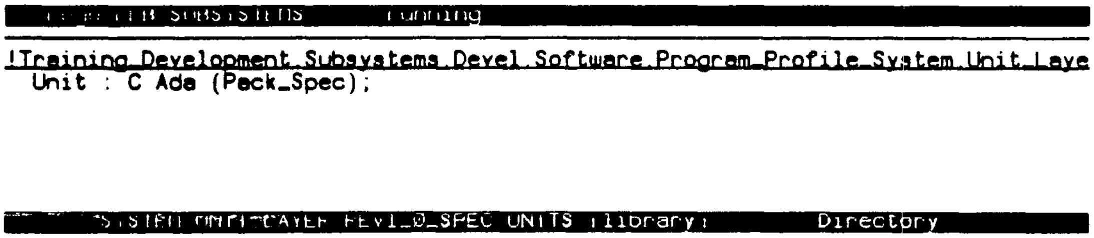


```
!Users.Rjb.Advanced_Training_Examples.Subsystem_Line_Analyzer.Unit_Layer.Rev1_1_
  Compiler_Switches          : File;
  Exports                    : File;
  Imports                    : File (Activity);
  Model                      : File (Activity);
  Referencers                : File (Objects);
  This_Is_The_Root_Of_A_View : File;
  Tool_State                 : Library (Directory);

 =    _ANALYZER_UNIT_LAYER.REV1_1_0.STATE (library)        Directory  (std info)
```

Examples of View Notation, cont.

- **Units** directory in a spec view



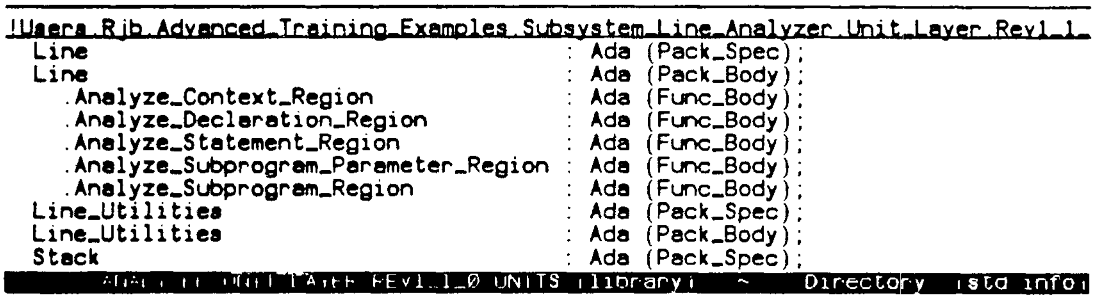
```
[illegible] SUBSYSTEMS          running

!Training.Development.Subsystems.Devel.Software.Program_Profile.System.Unit_Laye
  Unit : C Ada (Pack_Spec);




[illegible] UNIT_LAYER REV1_0_SPEC UNITS (library)          Directory
```

- **Units** directory in a load view



```
!Users.Rjb.Advanced_Training.Examples.Subsystem.Line_Analyzer.Unit_Layer.Rev1_1_
  Line                                       : Ada (Pack_Spec);
  Line                                       : Ada (Pack_Body);
     .Analyze_Context_Region                 : Ada (Func_Body);
     .Analyze_Declaration_Region             : Ada (Func_Body);
     .Analyze_Statement_Region               : Ada (Func_Body);
     .Analyze_Subprogram_Parameter_Region    : Ada (Func_Body);
     .Analyze_Subprogram_Region              : Ada (Func_Body);
  Line_Utilities                             : Ada (Pack_Spec);
  Line_Utilities                             : Ada (Pack_Body);
  Stack                                      : Ada (Pack_Spec);
[illegible] UNIT_LAYER REV1_1_0 UNITS (library)  ~   Directory  (std info)
```

Seminar Outline

Concepts of Subsystems

Introduction

Key Concepts

Subsystem Structure

● Traversal

Execution

Construction and Modification of Subsystems

Additional Topics

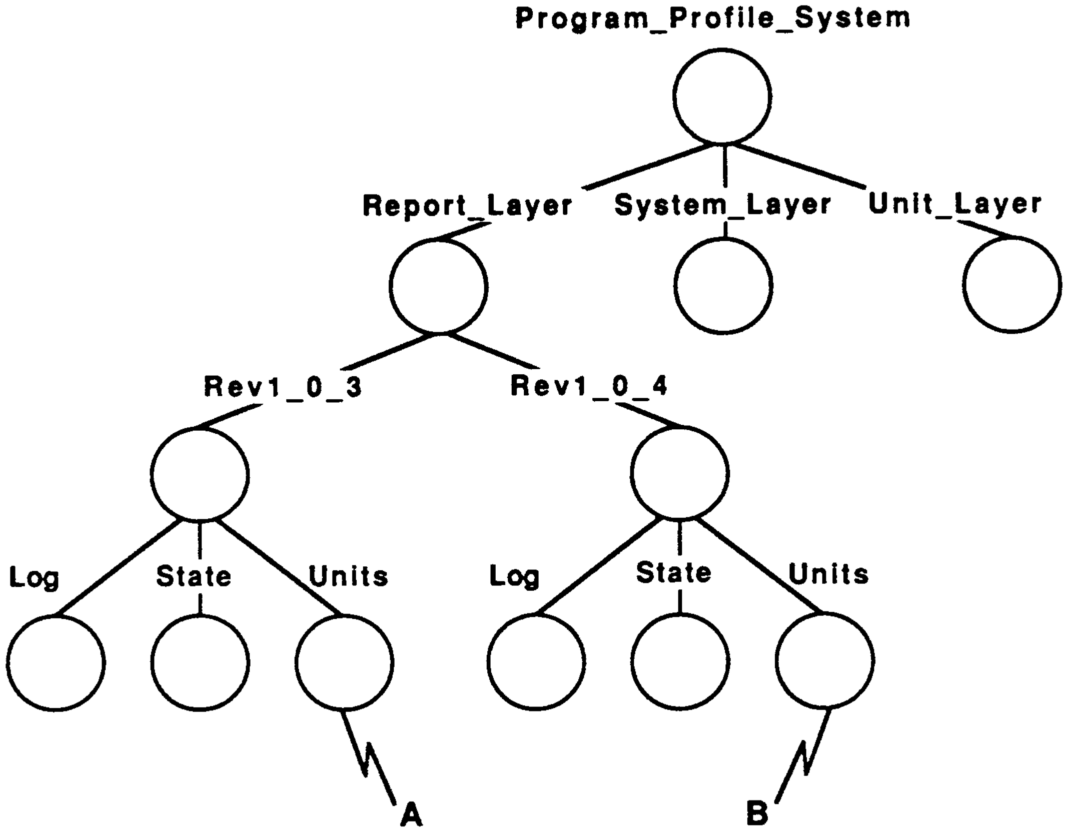
Mechanisms for Traversal

- Using basic traversal keys
  - Move up one level in the structure: [Enclosing Object]
  - Move down one level in the structure: [Definition]
  - Prefix to display a unit in the same window: [Window] - [Demote]
- Using library naming in the `Definition` command
  - Move up to the nearest enclosing view or subsystem world to resolve the name: double dollar sign (`$$`)
  - Move up one level in the structure to resolve the name: caret (^)
  - Move to the unit in the library specified in the session searchlist: backslash (\)

Mechanisms for Traversal, cont.

- Using [Definition] on spec views
  - — Move to a corresponding load view from the current activity: [Definition]
- Using subsystem-tool-supplied commands
  - — Move to a specified view in the same subsystem: `View.Goto_View`
  - — Move to a load view from a corresponding spec view: `View.Goto_View`
  - — Move to the display of the log for a specified command: `View.Find_Log`
- Using keybindings
  - — Login procedure with key rebindings to go to common locations
  - — Macros to go to common locations

Examples of Library Naming



- Move from point A to point B
  - — `Definition ("$$^Rev1_0_4.U@");`
  - — `Definition ("Rev1_0_4.Units");`
    assuming session searchlist contains
    `... Program_Profile_System.Report_Layer`

Seminar Outline

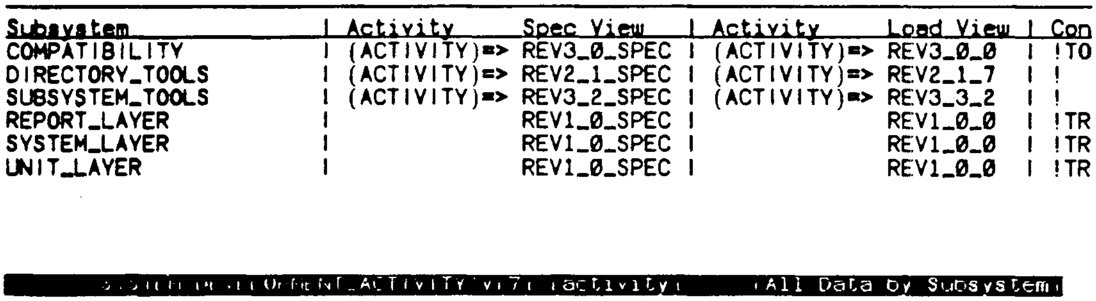
Concepts of Subsystems
- Introduction
- Key Concepts
- Subsystem Structure
- Traversal
- ● Execution

Construction and Modification of Subsystems

Additional Topics

Activities

- Specify various views that make up a set of subsystems that are to be linked and executed
- Consist of entries of specific spec and load views for each subsystem
- Example



```
Subsystem                 | Activity       Spec View    | Activity       Load View  | Con
COMPATIBILITY             | (ACTIVITY)=> REV3_0_SPEC | (ACTIVITY)=> REV3_0_0   | !TO
DIRECTORY_TOOLS           | (ACTIVITY)=> REV2_1_SPEC | (ACTIVITY)=> REV2_1_7   | !
SUBSYSTEM_TOOLS           | (ACTIVITY)=> REV3_2_SPEC | (ACTIVITY)=> REV3_3_2   | !
REPORT_LAYER              |              REV1_0_SPEC |              REV1_0_0   | !TR
SYSTEM_LAYER              |              REV1_0_SPEC |              REV1_0_0   | !TR
UNIT_LAYER                |              REV1_0_SPEC |              REV1_0_0   | !TR
```

```
[illegible] CURRENT_ACTIVITY'V(7) (activity)        (All Data by Subsystem)
```

Current Activity

- Users can have multiple activities related to an application
- Current activity is the activity to be used by the Environment when executing a program
- Current activity by default upon logging in is `!Release.Current.Activity`
- User can change current activity to be any activity
- Key commands
  - — Set the specified activity to be the current activity for the session: `Activity.Set_Default`
  - — Display the activity name associated with the current job or session: `Activity.Current`

Modification of Activities

- Activities are a type of object (file)
- Activity files must be saved to make changes permanent
- Commands
  - Edit the current activity: `Activity.Edit`
  - Edit the selected activity: Edit
  - Add a new entry to an activity: Object - I
  -
  - Delete a selected entry in an activity: Object - D
  - Save changes to an activity: Enter

Exercise: Managing Activities

Execute the Program Profile program with two different implementations of the same subsystem.

1. Run the `Program_Profile_Driver` program located in the `Activities_Exercise` library in your home world with the following subsystem configuration:

```
Unit_Layer.Rev1_0_0
System_Layer.Rev1_0_0
Report_Layer.Rev1_0_0
```

— Edit the `Current_Release` activity in the `Activities_Exercise` world. Use the activity entry for the `Unit_Layer` as a sample to add the entries for the other two layers of the system. Save the activity file after all changes are made. Make this the default activity.

Exercise: Managing Activities, cont.

- — Test the application by trying a few units in the `Test_Data` subdirectory in the `Activities_Exercise` world.

2. Run the `Program_Profile_Driver` program again with the same configuration for the `Unit_Layer` and `Systems_Layer`, but with `Report_Layer.Rev1_0_1`.

   This requires changing only the `Current_Release` activity before rerunning any tests. Note that the output is now in a tabular form.

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems

- Subsystem Construction
  Basic Modification Concepts
  Changes to Load Views
  Changes to Nonexported Specs
  Changes to Exported Specs
  Changes to Dependencies

Additional Topics

Early Design Methodology

- Prototype the design in a single world on the Rational Environment
  - As the initial structure stabilizes, partition it into logical components (that is, what you think will become subsystems)
  - Put each logical component into a sublibrary
- Suggested partitioning criteria:
  - A subsystem should be a complete, logical component of the system
  - A subsystem should have a well-defined, narrow interface

Early Design Methodology, cont.

— Package interfaces should export private types and avoid reexporting declarations from other subsystem interfaces

— A subsystem eventually should contain a manageable amount of code (5K-25K lines)

— A subsystem should have 1-3 developers working on it

Transition to Subsystems

- When should a preliminary design be moved into subsystems?
  - The set of units forming subsystem interfaces is defined and stable
  - The interdependence (linkage) between subsystems is defined and stable
  - Environment resources to be used in the system are defined and stable
- First steps:
  - Identify all components in each subsystem
  - Identify all exports and imports of each subsystem

Transition to Subsystems, cont.

- — Identify any external resources required from the Environment and any required compilation switches
- — Check for possible cycles across the current subsystem partitioning

Models

- External resource requirements are defined for each application with a model
- A project model is built as part of the design process
- Models can contain other project-specific tailoring of naming conventions and compilation switches
- Models can be built anywhere
  - Greater flexibility results if models are kept with application project library
  - Standard Environment models are kept in world `!Model`

Utilization of Resources

- From outside the view
  - — Links are managed by explicit importing or defined in the *model*
  - — A view's links should never be changed manually with link commands
- From other subsystems
  - — Dependencies are created between the current load view and other spec views
  - — Dependencies never exist between load views

Method for Building Systems from Bottom Up

- Basic model
  - Build a project model
  - Build a load view and spawn a spec view for the lowest subsystem first
  - Establish imports and exports
  - Add each subsystem on top of existing subsystems
  - Build all subsystems and their dependencies in a single pass

Method for Building Systems from Bottom Up, cont.

- Basic method for each subsystem, starting with the lowest
  - Create an empty subsystem and load view: `View.Initial`
  - Import any necessary subsystems: `View.Import`
  - Add Ada units to the units directory of the load view (partial skeletons acceptable for bodies)
  - Edit the file `Exports` in the view's `State` directory to reflect the set of units to be exported
  - Create a spec view: `View.Export`
  - Make each view consistent with `View.Make`

Exercise: Building an Application in Subsystems

Build a subsystem structure for the Program Profile program using the project library called `Projects` in your home world. Build each subsystem in the Subsystem.Application directory. A stable design currently is in the subworld `Design`.

1. Create each subsystem inside the `Projects` world. Use the model in `Projects.Design`.
2. Copy the actual Ada units from the corresponding directories in `Projects.Design`.
3. Set up all the necessary imports and exports, referring to the diagram in the Sample Application section of Key Concepts (page 28).
4. Make each view consistent with a View.Make command.

Exercise: Building an Application in Subsystems, cont.

5. Build an activity specifying the initial load view for each subsystem and verify that the execution matches the system defined in `Design`. Add new subsystem entries in the existing current release activity inside the Subsystem.Application directory.

Method for Building Systems from Top Down

- Basic model
  - Build a project model
  - Build a view for the top subsystem first
  - Add subsystems under existing subsystems
  - Build all subsystems and their dependencies in two passes

Method for Building Systems from Top Down, cont.

- Basic method builds all subsystems without interdependencies, starting with the top subsystem
  - — Create an empty subsystem and load view: `View.Initial`
  - — Add all specifications for the subsystem and promote to installed
  - — Edit the file `Exports` in the view's `State` directory to reflect the set of units to be exported
  - — Create a spec view: `View.Export`
  - — Repeat for all remaining subsystems

Method for Building Systems from Top Down, cont.

- Basic method builds all interdependencies between subsystems, starting again with the top subsystem
  - Import any necessary subsystems:
    `View.Import`
  - Repeat for all remaining subsystems

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems
Subsystem Construction
● Basic Modification Concepts
Changes to Load Views
Changes to Nonexported Specs
Changes to Exported Specs
Changes to Dependencies

Additional Topics

Ongoing Development Activities

- Create new load views from the current release
- Implement changes to fix bugs and/or add functionality
- Test
  - Unit
  - Integration or layer level
  - System level
- Release the modified and tested subsystem for other project members to use

Creation of New Load Views

- Is called *spawning*: `View.Spawn`
  - — Creates a load view from an existing view
  - — Copies the contents of the existing view including links into the new view
  - — Promotes all units in the `Units` directory to the installed state
  - — Disconnects as soon as copy begins
  - — Adds a command log to the subsystem `Logs` directory

Implementation of Changes

- Editing operations on Ada units still apply
- Incremental operations on Ada units still apply
- Compiling views: `View.Make`
  - Is conceptually the same as compiling worlds
  - Automatically maintains a history log in the view's `Logs` directory
  - Operates only on units in the current view

Design Visibility Management

- From outside the view
  - — Links are managed by explicit importing or defined in the *model*
  - — A view's links should never be explicitly changed
- From other subsystems
  - — Dependencies are created between the current load/spec view and other spec views
  - — Dependencies never exist between two load views

Testing

- Can use test scaffolds, which can be built within the view
- Does not require additional copying or recompilation
- Does not interfere with users of the released system
- Basic method
  - Ensure that units in all load views that make up the test are coded
  - Modify a local development activity to include the new load view
  - Verify that the local activity is the default activity
  - Execute the subprogram in a Command window

Release of Views

- Means freezing a view and making it available for use by other members of the application development team
- Basic method
  - Freeze the released load view:
    `View.Freeze`
  - Note that the `Logs` directory in the view is updated with the results of the `Freeze` command
  - Modify the current release activity in the subsystem to include the new load view

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems
- Subsystem Construction
- Basic Modification Concepts
- ● Changes to Load Views
- Changes to Nonexported Specs
- Changes to Exported Specs
- Changes to Dependencies

Additional Topics

Modification of Load Views

- Is similar to changes in nonsubsystem libraries
- Basic method
  - Create a new load view to make changes: `View.Spawn`
  - Implement the changes using incremental changes
  - Make the view consistent: `View.Make`
  - Check the log if there are errors: `View.Find_Log ("make");`
  - Test the modified units and the subsystem as a whole
  - Release the changed subsystem for others to use

Exercise: Changing Load Views

Complete the implementation of the body of package `Line` in the `Program_Profile_System` library in your home world. Package `Line` is in the `Unit_Layer` subsystem.

1. Spawn a new load view to make the changes in the subsystem containing package `Line`. Use the most current release of the load view, `Rev 1_0_0`.

2. Make the following changes:

   — Incrementally add (edit the statement prompt) the following statement to `Line.Has_Semi_Colon`:

   ```
   return Lu.Is_Semi_Colon (Su.Strip (The_Line));
   ```

Exercise: Changing Load Views, cont.

- Incrementally add the following statement to `Line.Is_Assignment`:

  ```
  return Lu.Is_Assignment (Su.Strip (The_Line));
  ```

- Incrementally add the following statement to `Line.Is_Loop`:

  ```
  return Lu.Is_End_Loop (Su.Strip (The_Line));
  ```

3. Use `View.Make` to make the view consistent.
4. Update the activity local to the `Unit_Layer` subsystem and make it the default.
5. Verify your changes by rerunning the program using `Test_Driver1` in the `Testing` subdirectory in `Program_Profile_System`.
6. Release the new view by freezing the view and updating the current release activity.

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems
Subsystem Construction
Basic Modification Concepts
Changes to Load Views
• Changes to Nonexported Specs
Changes to Exported Specs
Changes to Dependencies

Additional Topics

Modification of Nonexported Specs

- Nonexported specs refer to package specifications in a load view that are not included in the corresponding spec view
- Changes to nonexported specs do not change the spec view
- The basic method is the same as when changing bodies in load views
  - — Spawn a new load view in which to make changes
  - — Change nonexported specs (and bodies)
  - — Make the load view consistent
  - — Test changes
  - — Release the new load view

Exercise: Making Changes to Nonexported Specs

Incrementally add a function that checks for the occurrence of an *if* statement to the spec and body of package `Line` in the `Program_Profile_System` library. Package `Line` is in the `Unit_Layer` subsystem.

1. Spawn a new load view in the subsystem containing package `Line`.
2. Incrementally add the following function to the spec and body of `Line`.

```
function Is_If (The_Line : String) return Boolean is

begin
   return Lu.Is_End_If (Su.Strip (The_Line));
end Is_If;
```

Exercise: Making Changes to Nonexported Specs, cont.

3. Incrementally modify the following *if* statement in the body of `Unit` to read as follows:

   ```
   if Line.Is_Assignment (Unit_Line) then
        The_Statistics.Assignments :=
           The_Statistics.Assignments + 1;

   elsif Line.Is_Loop (Unit_Line) then
           The_Statistics.Loops :=
              The_Statistics.Loops + 1;

   elsif Line.Is_If (Unit_Line) then
           The_Statistics.Ifs :=
              The_Statistics.Ifs + 1;

   end if;
   ```

4. Use `View.Make` to make the view consistent.

5. Verify your changes using the local development activity.

6. Release the new view.

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems
- Subsystem Construction
- Basic Modification Concepts
- Changes to Load Views
- Changes to Nonexported Specs
- ● Changes to Exported Specs
- Changes to Dependencies

Additional Topics

Classes of Changes to Exported Specs

- Three classes of changes can be made
  - Changes to closed private parts of specs
  - Other upward-compatible changes to specs
  - Incompatible changes to specs
- Each class of change uses a different method for making the change
  - Closed private part changes are similar to changes to load views
  - Other compatible changes are similar to changes to load views but require recoding of dependent views
  - Incompatible changes can require significant reconstruction of the system

Private Parts of Exported Specs

- Closed private parts
  - Environment provides support for conceptual separation of private part from visible package declarations
  - Changes to private parts behave as if a change was made to the package body in the load view
  - By default, private parts are "closed" in subsystems

Private Parts of Exported Specs, cont.

- Pragma `Private_Eyes_Only`
  - Is used in package spec in view
  - Identifies units in context clauses that are needed only for the private part
  - Makes specified units unnecessary in a spec view
  - Applies to all context clauses following the pragma
- Example

```
with Time_Utilities;
pragma Private_Eyes_Only;
with Time_List;
package Event_Log is
    type Log is private;
    function Make return Log;
    function Start_Time (L : Log)
            return Time_Utilities.Time;
    ...
private
    type Log is new Time_List.List;
end Event_Log;
```

Modification of Private Parts

- Changes to closed private parts do not require recompilation of external dependents
- Basic method
  - — Spawn a new load view
  - — Change the private part of the exported specification in the *load* view
  - — Make the load view consistent
  - — Test changes
  - — Release the new load view

Exercise: Changing a Private Part

Modify the data structure representation for type `Statistics` in package `Systems` of the `Program_Profile_System`. This change requires modification to the load view only. Package `Systems` is in the `System_Layer` subsystem.

1. Spawn a new load view in the subsystem containing package `Systems`. Set the parameter `Goal` to `Compilation.Source`.

2. Make the following modifications:

   — Add a context clause for `Unbounded_String` to the spec of package `Systems`.

   — Change the private part of the spec of package `Systems` to appear as in the private part provided on the following page. The changed or new lines are indicated by _***.

Exercise: Changing a Private Part, cont.

```
private

  package Name_String is new Unbounded_String; -- ***

  type Statistics is
    record
      Name         : Name_String.Variable_String; -- ***
      Units        : Natural := 0;
      Lines        : Natural := 0;
      Statements   : Natural := 0;
      Declarations : Natural := 0;
      Withs        : Natural := 0;
      Assignments  : Natural := 0;
      Comments     : Natural := 0;
    end record;

  type Unit_Iterator is access Object_Naming.Iterator;

end Systems;
```

— Replace all statements in `Systems.Set_Name` with the following:

```
The_Statistics.Name := Name_String.Value (The_Name);
```

— Replace the statement in `Systems.Name_Of_System` with the following:

```
return Name_String.Image (The_Units.Name);
```

Exercise: Changing a Private Part, cont.

3. Make the view consistent.
4. Verify your changes. (The execution should be identical.)
5. Release the new view.

Other Compatible Spec Changes

- Three other kinds of changes are considered upward compatible
  - Adding context clauses
  - Adding package renaming declarations anywhere in the package
  - Adding new declarations at the end of the package

Other Compatible Spec Changes, cont.

- Upward-compatible changes require consistent changes to spec and load views within a subsystem and recoding of dependent views in other subsystems
  - — Making compatible additions requires demoting the spec view to the installed state
  - — Subsystem tools demote dependent views in other subsystems to the installed state
  - — Incremental operations are used in both spec and load views
  - — Dependent views can be recoded with `View.Make`

Method for Making Upward-Compatible Changes

- Spawn a new load view
- Demote the affected units in the spec view to installed
- Incrementally make changes to the affected spec view
- Make a consistent set of changes to the new load view and any previous views as necessary
- Recode all load views that were uncoded
- Test changes
- Release the new load view

Exercise: Making Upward-Compatible Changes

Add a new function to calculate the average statements per unit to package Systems in the Program_Profile_System following these steps. Package Systems is in the System_Layer subsystem.

1. Spawn a new load view in the subsystem containing package Systems.

2. Modify the new load view to include the new function.

   — Incrementally add the subprogram declaration defined below to the specification of package Systems. Note that the subprogram spec must be placed at the end of the declarations in the spec of package Systems.

   ```
   function Average_Statements_Per_Unit
               (The_Units : Statistics) return Float;
   ```

Exercise: Making Upward-Compatible Changes, cont.

— Incrementally add the subprogram body defined below to the body of package `Systems`.

```
function Average_Statements_Per_Unit
        (The_Units : Statistics) return Float is
begin

    if The_Units.Units /= 0 then
        return Float (The_Units.Statements) /
             Float (The_Units.Units);
    else
        raise Bad_Data;
    end if;

end Average_Statements_Per_Unit;
```

3. Make the load view consistent.

4. Modify the spec view corresponding to the new load view to include the new function.

   — Unfreeze the spec view and all externally dependent load views with the command `Library.Unfreeze`.

Exercise: Making Upward-Compatible Changes, cont.

- — Demote the exported unit `Systems` with the `View.Demote` command, setting the `Goal` parameter to `Compilation.Installed` and the `Limit` parameter to `Compilation.All_Worlds`.
- — Incrementally add the subprogram declaration provided to the specification of package `Systems`.

```
function Average_Statements_Per_Unit
            (The_Units : Statistics) return Float;
```

- — Make the spec view consistent.

5. Make any dependent load views in the `Report_Layer` consistent and refreeze them.
6. Verify your changes by adding the new functionality to the body of package `Report` in the `Report_Layer`.

Exercise: Making Upward-Compatible Changes, cont.

- Spawn a new load view for the `Report_Layer`.
- Add the following statement to `Report.Display (The_System : ...)`:

  ```
  Rio.Put (Systems.Average_Statements_Per_Unit
                (The_System));
  ```

  which should follow the statement `Rio.Put ("| Average Statements ....`
- Make the new view consistent and execute.

7. Release the new views in the `System_Layer` and `Report_Layer` subsystems.

Incompatible Spec Changes

- Are considered to be design changes
- Require a new spec view
- Require new load view(s) and possibly new spec view(s) in other subsystems that import the new spec view
- Imply that each user of that subsystem must be recompiled

Method for Making Incompatible Changes

- Create a new load view in the subsystem requiring the change
  - — Spawn a new load view
  - — Make all changes and make consistent
- Create the new spec view from the modified load view: `View.Export`
  - — Copies only the exported units from the load view
  - — Hides the private parts of exported packages

Method for Making Incompatible Changes, cont.

- Create a new load view to use the changed import in each subsystem that imports the changed subsystem
    - — In the `State` directory of each new load view, create a copy of the imports activity called `Activity_For_Spawn`. The `State` will need to be unfrozen in order to do this.

      ```
      activity.create ("activity_for_spawn",
         "imports")
      ```

    - — Edit the new activity, changing the modified views to their new release
    - — Spawn a new load view as before:

      ```
      View.Spawn
      ```

    - — Make any necessary changes in the new load view to utilize the newly imported view

Method for Making Incompatible Changes, cont.

- Repeat the entire process as necessary for other subsystems if any changes cause another spec view to be created

Exercise: Making Incompatible Changes

Add a new type `Status_Code` to the specification of package `Unit` in the `Program_Profile_System`. Package `Unit` is in the `Unit_Layer` subsystem.

1. Add an enumeration type to the declarative part of the package specification of `Unit` in the load view of `Unit_Layer`.

```
type Status_Code is (Normal,
                     Illegal_Unit_Name,
                     Inaccessible_Unit,
                     Data_Error,
                     Unknown);
```

2. Change the declaration of the function `Analyze` to a procedure with the following definition:

```
procedure Analyze (The_Unit : Unit_Name;
                   The_Statistics : in out Statistics;
                   The_Status : out Status_Code);
```

Exercise: Making Incompatible Changes, cont.

3. Modify the corresponding body of `Analyze`.

   — Change the function to a procedure and change the parameter profile.

   — Delete the declaration of `The_Statistics` (this is now declared as a parameter).

   — Replace the return statement with the following:

   ```
   The_Status := Normal;
   ```

4. Spawn a new spec view of `Unit_Layer`.

5. Rebuild the `Systems_Layer` by spawning a new load view that imports the new spec view of `Unit_Layer`. Since the spec view also depends on the `Unit_Layer`, a new view importing the new `Unit_Layer` spec view must also be created.

Exercise: Making Incompatible Changes, cont.

6. Rebuild the `Report_Layer` by spawning a new load view that imports the new spec view of `Unit_Layer` and `System_Layer`. Note that, since the spec of `Program_Profile` does not change, a new spec view is not required.

7. Modify the main program unit `Program_Profile` in the new load view to include the new parameter profile for `Analyze`.

   — Add the following declaration:

   ```
   Unit_Status : Unit.Status_Code;
   ```

   — Also change the call to `Analyze` to the following:

   ```
   Unit.Analyze
        (Systems.Value (Units_Iterator),
         Unit_Statistics,
         Unit_Status);
   ```

8. Make the view consistent

9. Verify your changes.

10. Release the new views.

Compatibility of Changes Revisited

- Following methods in this module should maintain compatibility between the spec view and the corresponding load view(s)
- The Environment does not prevent changes that would make spec views incompatible with load views
  - Any change can be made to a load view without changing the corresponding spec view
  - Incompatibilities will produce nondeterministic errors when the system is run
- Compatibility can be checked with procedures from package `!Tools.Compatibility.Revn.Units.Check`
  - Compares a spec view and a load view
  - Compares two load views

Compatibility of Changes Revisited, cont.

— Compares all spec view and load view pairs in an activity

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems
- Subsystem Construction
- Basic Modification Concepts
- Changes to Load Views
- Changes to Nonexported Specs
- Changes to Exported Specs
- ● Changes to Dependencies

Additional Topics

Changes to Imports

- Imports are governed by the subsystem tools
  - Changes to imports are considered to be design changes
  - Additional history is maintained in the subsystem
  - Once defined for a view, imports can be modified or deleted only by creating a new view
- Four types of changes to imports can be done
  - Importing a subsystem not previously imported into the view
  - Importing a spec view that has new units in it
  - Importing a different view of a subsystem than the view currently imported

Changes to Imports, cont.

— Removing a subsystem from the set of imported subsystems

Importation of a New Subsystem

- Imports can be added to an existing subsystem: `View.Import`
  - Adds the necessary links and other information to the current view
  - Immediately allows units in the current view to utilize the new subsystem
- Imports from subsystems that have newly exported units can also be updated without creating a new view

Removal of a View or Importation of a Different View

- Once a view is imported, removing or changing an imported view requires a new load view
- Basic method
  - Create a copy of the imports activity called `Activity_For_Spawn` in the state directory of the current load view:

    ```
    activity.create ("activity_for_spawn",
    "imports")
    ```

  - Edit the new activity, deleting or changing the view(s) imported
  - Spawn a new load view as before:

    ```
    View.Spawn
    ```

  - Make any necessary changes in the new load view to utilize the newly imported view

Removal of a View or Importation of a Different View, cont.

— Repeat the entire process as necessary for other subsystems if any changes cause another spec view to be created

Changes to Exports

- Changing the set of exported specs
  - Requires a new spec view
  - Is similar to making incompatible changes to exported specs
- Exported specs are selected from the set of units in a load view via an `Exports` file in the view's `State` directory
- Basic method
  - Change all necessary units in the current load view
  - Edit the file `Exports` in the `State` directory of the current load view
  - Change the file to represent the new set of exported units
  - Spawn a new spec view: `View.Export`

Changes to Project Models

- Changing the project-specific model
  - Is considered to be a design change
  - Should be done by the system designer
  - Changes the links, switches, or other objects in the model
- Changes to the model are incorporated into all views created after the model is changed

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems

Additional Topics

- Helpful Hints

Test and Release Alternatives
Change Tracking

Workspace Management

- A view can be destroyed: `View.Destroy`
  - Unfreezes and deletes entire view
  - Records a log of the deletion
- Information about a view can be displayed: `View.Information`
  - Displays imports and exports
  - Displays model, dependencies, and units
  - Displays switches, creation time, and ancestry

Typical Errors

- Activity does not specify necessary views for execution: `Error in subsystem_spec look-through for VIEW`
- Units in view specified in activity are not coded
- Context for command is incorrect: `View name not resolved`

Management of Design Changes

- Many kinds of design changes to exported units are costly
  - Incompatible changes require reconstruction of subsystems that import the changed subsystem
  - Dependencies can cause much of the entire system to be reconstructed
- Design changes should be collected and integrated
  - Integrate changes when a subsystem is reconstructed to consolidate change impact
  - Should schedule such reconstructions at regular intervals

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems

Additional Topics

Helpful Hints

- Test and Release Alternatives

Change Tracking

Test and Release

- Locations for building tests are
  - In `Units` directory of view
  - In a user-created directory in a view
  - In a separate subsystem
- Recombinant testing requires simply changing the test activity
- Releases are managed with activities
  - Release a view with a current release activity in the subsystem
  - Release a system with a system release activity in the project library

Activities

- Can be created in three forms: differentials, exact copy, value copy
  - — Differential activities contain pointers to other activities
  - — Exact-copy activities contain exactly what was in the source activity
  - — Value-copy activities contain the dereferenced values of the source activity
- Can be managed in several ways:
  - — Use a value activity for each development or release activity

Activities, cont.

- — Use a current release activity for each subsystem and a differential activity that points to the current release activity in each subsystem
- — Make a value activity from a differential activity for each release

Creation of Activities

- Activities typically are created from Environment default
  - The default activity, `!Releases.Current-.Activity`, is supplied to the `Source` parameter of the `Activity.Create` command
  - Automatic access is provided to latest versions of Environment-supplied tools released via subsystems
- Empty activities are created with the default parameter value, `Activity.Nil`, in the `Activity.Create` command

Alternative View Management

- The view management used in this course is based on always creating and working in a new load view after every release
  - Tools handle all naming of new views
  - The location of work is always changing
- The alternative is to have a fixed development view and to spawn released views from it
  - Utilizes one view for all development (for example, `Devel`)
  - Requires spawning explicitly named released views: `View.Spawn_Named`

Alternative View Management, cont.

- Command parameters allow the tailoring of the command
  - The Level parameter specifies which revision level should be incremented (default is lowest)
    Level 0 creates `Rev1_0_1` from `Rev1_0_0`
    Level 1 creates `Rev1_1_0` from `Rev1_0_0`
  - Other parameters specify the goal state of promotion, view to copy, imports for new view, and whether to run as a foreground or background job

Seminar Outline

Concepts of Subsystems

Construction and Modification of Subsystems

Additional Topics
Helpful Hints
Test and Release Alternatives
● Change Tracking

Histories

- Allow annotation of changes in a view
  - Annotate changes to a particular unit: `History.Indicate_Change`
  - Annotate changes to all changed units in a view: `History.Change`
- Track the units that have not been annotated since changed: `History.Show_Undocumented_Changes`

Histories, cont.

- Display history, both user annotations and compilation summaries, of changed units in a view
  - — Display list of changed units: `History.Show_Changed_Units`
  - — Display user annotations for changed units: `History.Show_Change_History`
  - — Display compilation summaries for a view: `History.Show_Compilation_History`