Rational Environment Training

Basic Product Training Series: Subsystems and Configuration Management

Instructor's Notes

Copyright © 1988 by Rational

Document Control Number: 8030A-01

Rev. 1.0, September 1988 (Software Rev. D_10_20_0)

This document is subject to change without notice.

Note the Reader's Comments forms at the end of this book, which request the user's evaluation to assist Rational in preparing future documentation.

Rational and R1000 are registered trademarks and Rational Environment and Rational Subsystems are trademarks of Rational.

Rational
3320 Scott Boulevard
Santa Clara, California 95054-3197

Contents

Course Outline

Subsystems and CMVC

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

Notes on Course Outline

Transition: This is an introduction to project management. A more complete investigation is available in the course on Project Development Methods.

This course introduces the following concepts:

Basic project development methodology
Subsystems and spec/load views
Importing and exporting
Reservations (`Check_In` and `Check_Out`)
Parallel development subpaths
Propagating changes across subpaths (`Accept_Changes`)

Special Instructions:

Motivations

- Rational's solution originates from our experience in:
  - — Developing the Environment
  - — Consulting with customer applications
- Existing development facilities were discovered to be inadequate in:
  - — Reducing the time and costs of making and testing changes to large Ada systems
  - — Managing the complexity of a project and its design
  - — Supporting parallel team development and testing
  - — Supporting multihost and multisite development

Notes on Motivations

Transition:

Special Instructions:

Key Points:

-
  - Rational has developed more than a million lines of code with a team of 15–30 developers.
  - Philips, for example.
- As Rational was developing the Environment, the development team ran into several unforeseen issues with the development of large Ada systems. From this experience, Rational has designed and implemented a development methodology and tools to address and minimize some of these difficulties.
  -
  -
  -
  - Many projects are so large that they require the use of more than one machine. Also, subcontracting situations will require this. The ability to partition projects across multiple hosts with a high degree of transparency can be a distinct advantage.

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
- Continuous changes by multiple developers are difficult to test independently
- Historical record of changes is essential to project management

Notes on Changes to Ada Systems

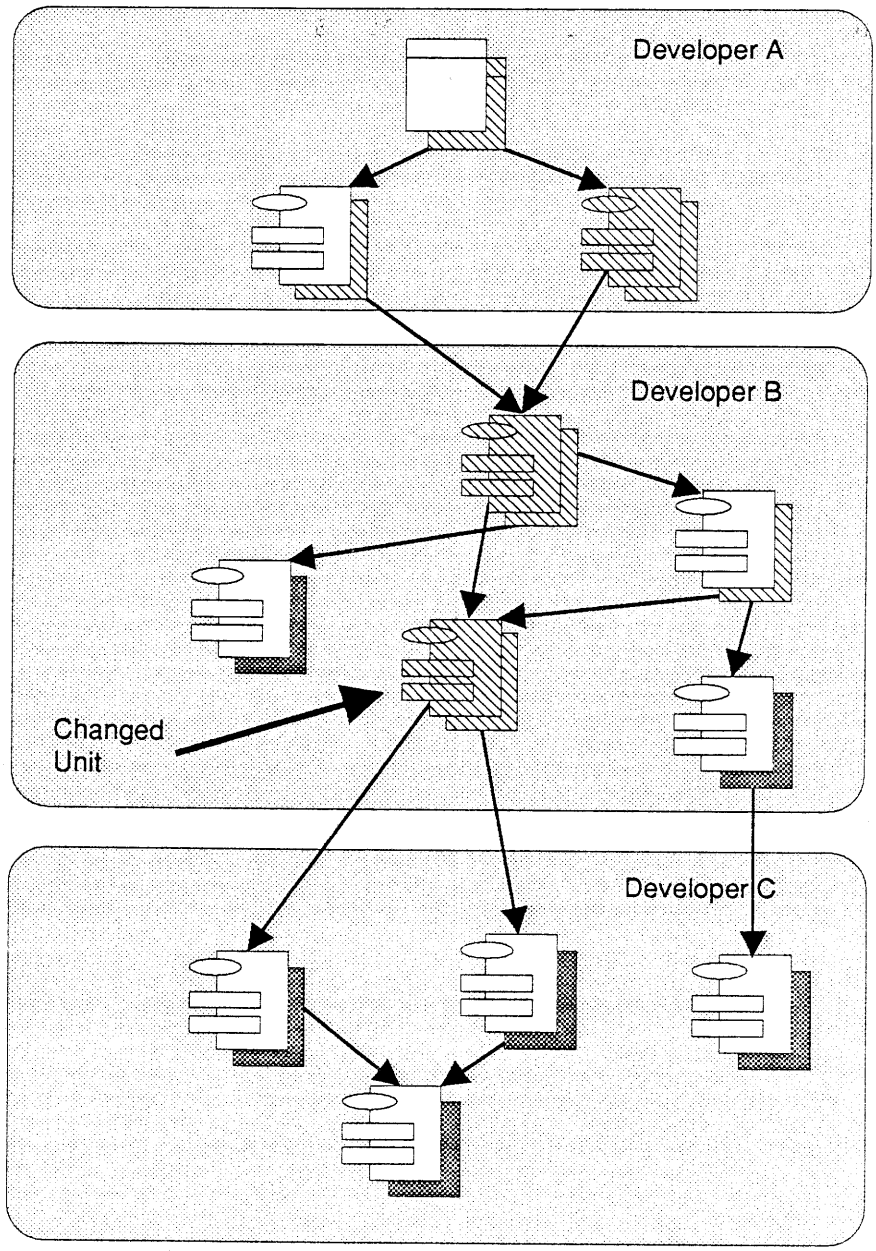
Transition: Let's look at a few of these problems. First, let's look at the organization of Ada systems and its impact on the development progress of a project. The first diagram illustrates the dependency closure and recompilation requirements for a sample change.

Special Instructions: Use the diagrams to illustrate the bullets. Units shaded with slashes represent the units that would need to be recompiled. Boxes with two rectangles represent Ada packages. The shaded box represents the body. The box at the top of the diagram is the main driver. Arrows indicate dependencies between units. The arrow is drawn from the dependent unit to the specification it *with*s.

Key Points:

- A side effect of requiring that dependencies be checked is that recompilation is required to verify that some change doesn't invalidate the correctness of the program. The time it takes to perform this recompilation can cause major delays.
  - –
  - – This we found from firsthand experience in developing the Environment.
- It is very difficult to tell which changes resulted in specific execution effects in the program when releases are constantly changing. It would be preferable to have stable releases from all other developers so that individuals see the effects of changes to their code only.
-

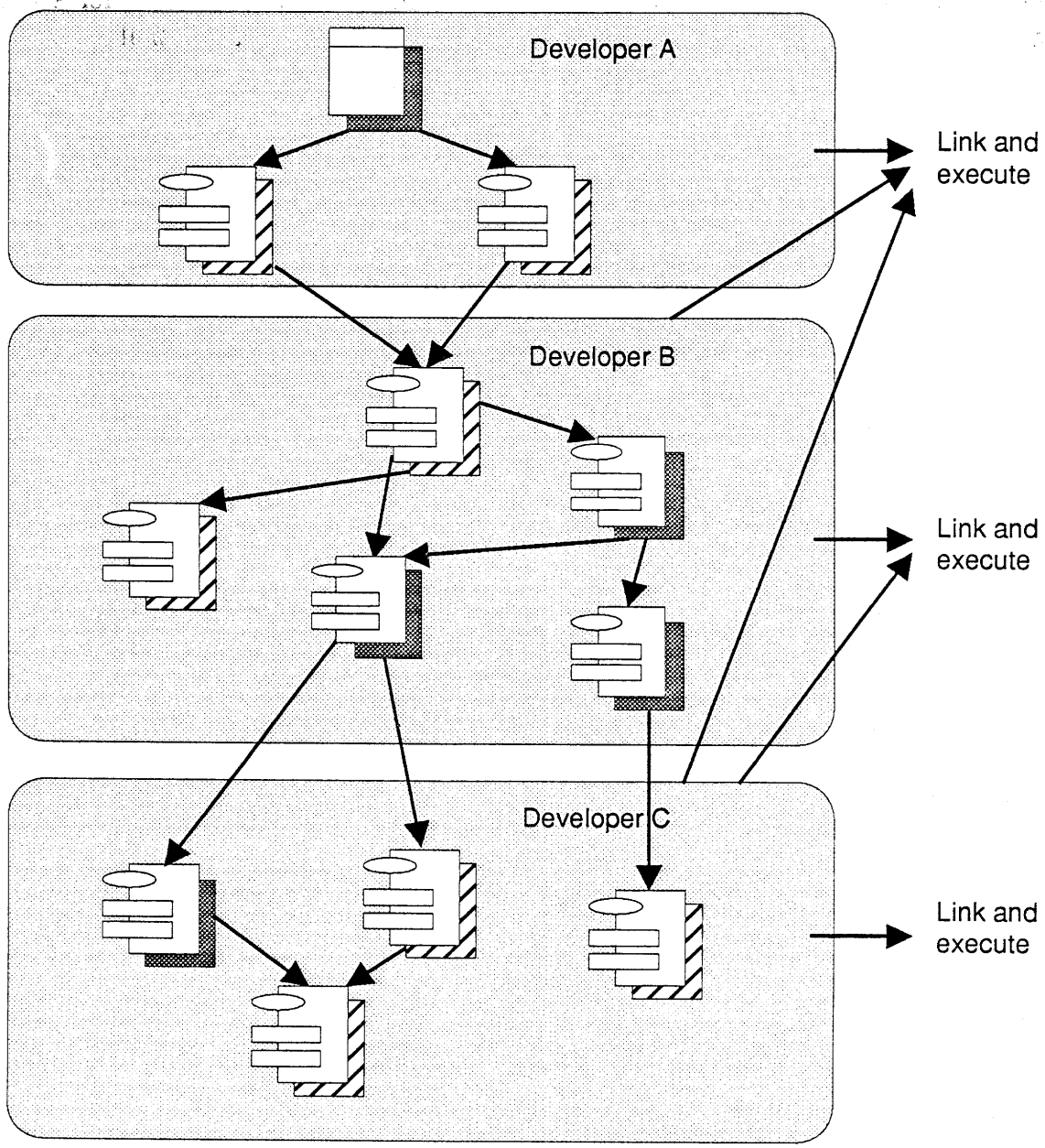
Changes to Ada Systems (cont.)



Notes on Changes to Ada Systems (cont.)

Special Instructions: Dependent units must be recompiled if changes to Ada units are made. Emphasize transitive closure of dependencies. Even with several libraries, developer B can impact developer A's work, requiring notification and serialization of work. In larger systems, with 15, 30, 50, or more developers, this can become a major issue.

Changes to Ada Systems (cont.)



Notes on Changes to Ada Systems (cont.)

Special Instructions: Even with just changes to bodies, it can be difficult if not impossible to see the cause and effect of changes. With everyone making changes at the same time, an effect in the program could have been caused by several factors. It would be preferable to have stable, unchanging code for all units except those that an individual developer controls.

Design Degradation

- Dependencies in a system reflect part of the overall design
- Unwanted dependencies are easily added by any developer inserting a *with* clause
- With simple library facilities, no safeguards are available to preserve the integrity of the design

Notes on Design Degradation

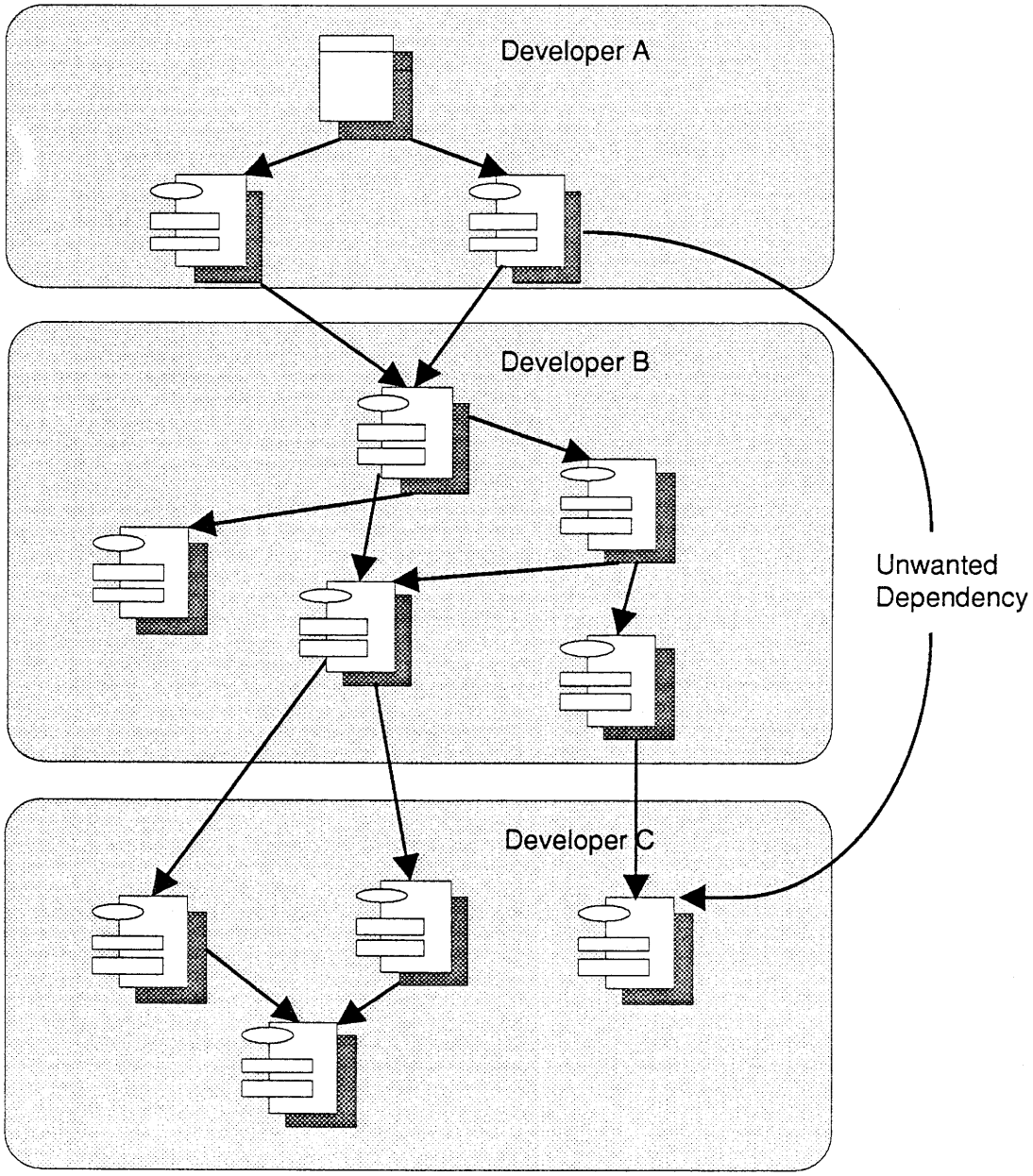
Transition: Another issue facing large development efforts is maintaining design integrity through the actual development and maintenance phases.

Special Instructions: Use the diagram to illustrate how designs can become corrupted over time.

Key Points: It's easy to corrupt designs with Ada only. Generally, it is undesirable for just anyone to change the design.

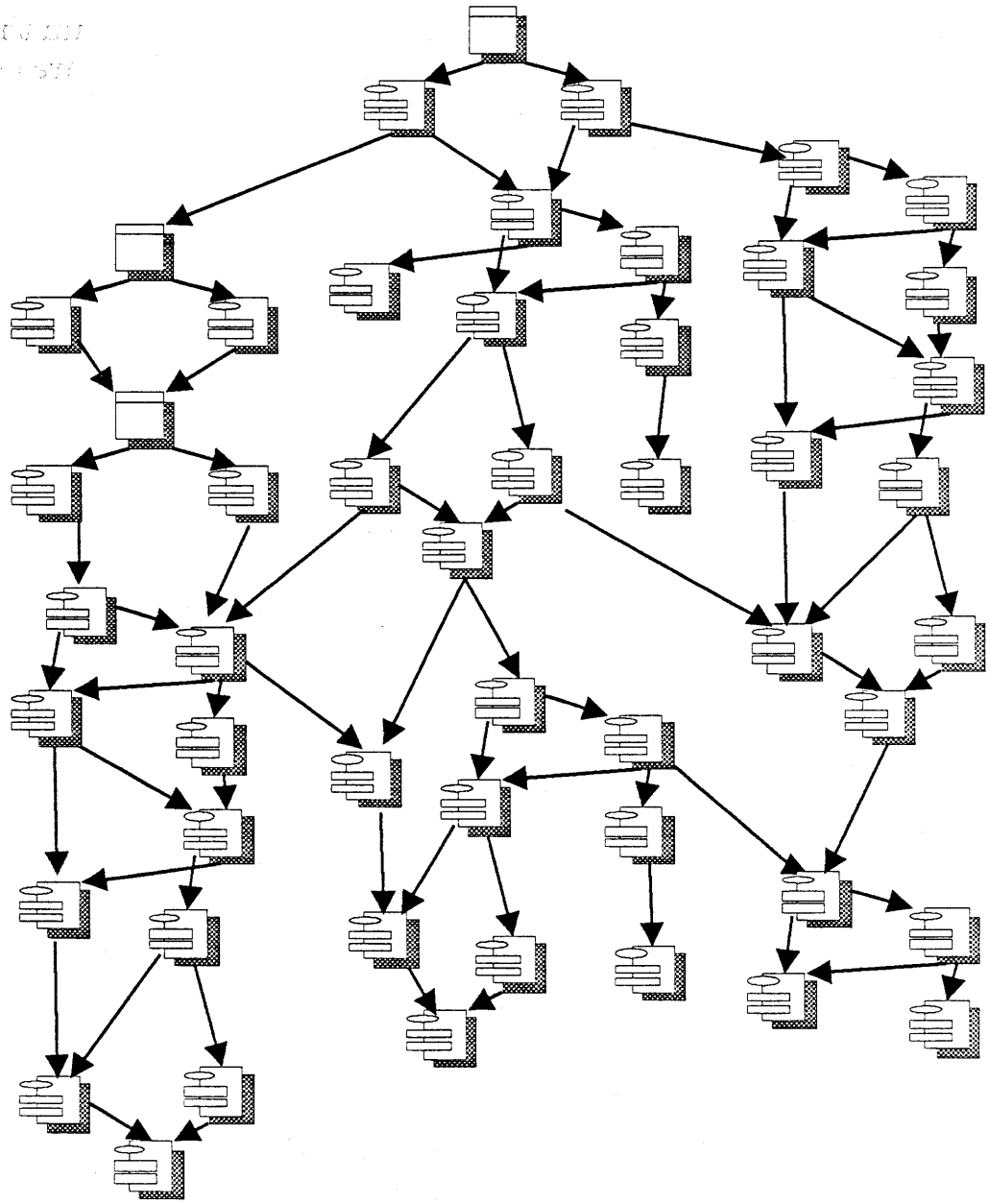
-
- Design changes of this nature are hard to capture and track. They may have a different functionality than originally intended.
-

Design Degradation (cont.)



Notes on Design Degradation (cont.)

Large Ada Systems



Notes on Large Ada Systems

Transition: So far we've looked at two of the issues: recompilation requirements and design integrity. We've also been looking at small Ada systems. Let's look now at these same issues and some new ones with a larger Ada system.

Special Instructions: Use the diagram to illustrate and explain the key points. The diagram is placed first for effect. Even though there are only 50 packages, the complexity grows rapidly.

Key Points:

Large Ada Systems (cont.)

- Difficult to understand the application by the picture
- Difficult to reason about the dependencies
- Required recompilation can take hours or days
- Difficult to allow individuals to develop in parallel because of Ada's strong dependencies
- Difficult to partition so that individual developers can implement and test in parallel

Notes on Large Ada Systems (cont.)

- It's hard to understand a system with this many units. Even knowing what individual packages do doesn't help in understanding the big picture.
- The amount of recompilation that would be required for different kinds of changes is hard to determine. Ada packages are too fine a level of granularity to provide an effective focus for project management.
-
- They are bound to others by the dependencies of their code on code owned by other people.
-

Subsystem Partitioning

- Subsystem partitioning improves understanding of the application
- Dependencies can be defined at the subsystem level
- Dependencies between subsystems can be enforced by tools
- Each subsystem can be assigned to an individual developer or implementation team
- Subsystems provide the opportunity to contain recompilation

Notes on Subsystem Partitioning

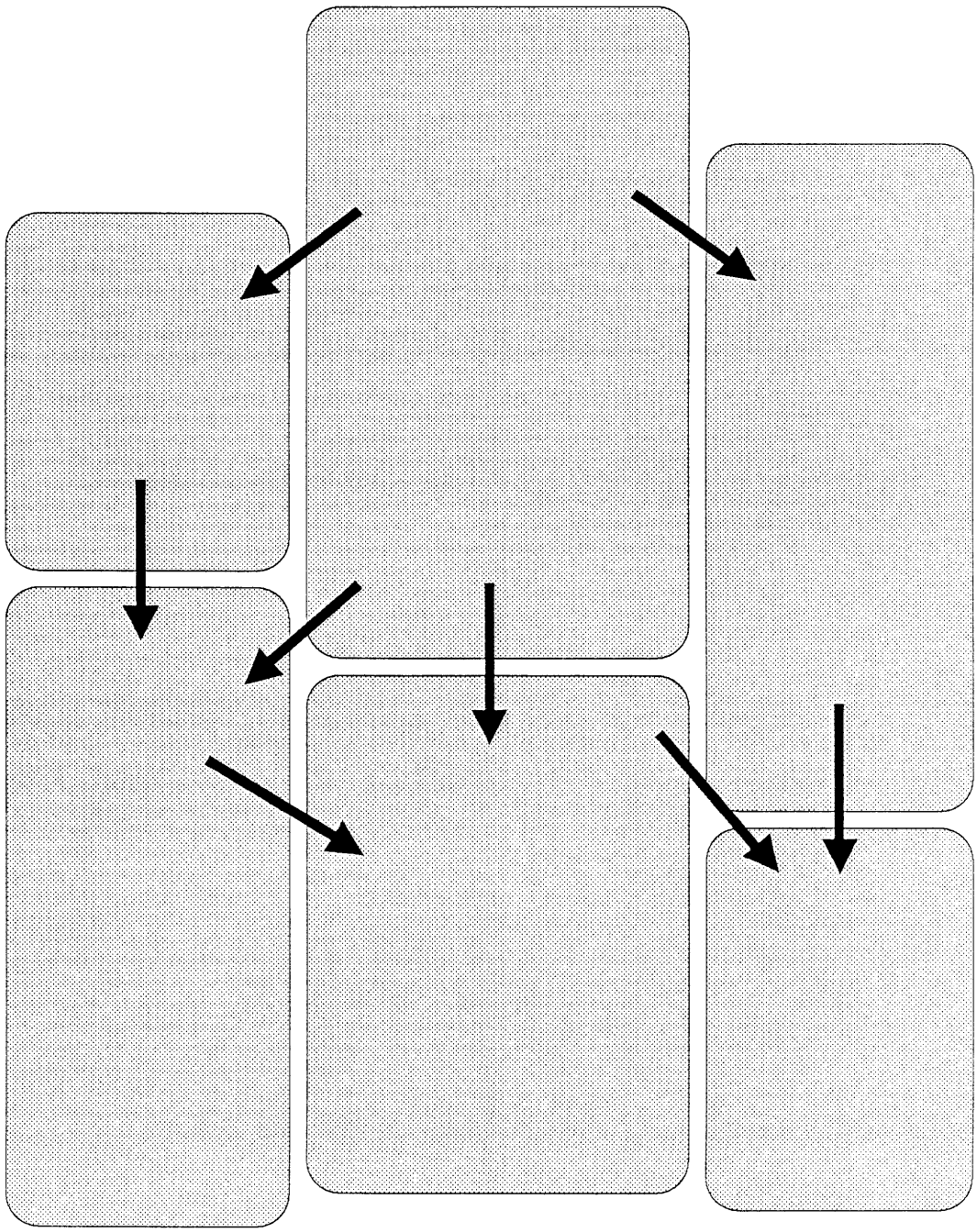
Transition: But if we look at the same system partitioned into several subsystems ...

Special Instructions: Use the diagram to illustrate and explain the points.

Key Points: Subsystem-level partitioning can improve the comprehension and management of large systems.

-
- This reduces the number of dependencies at the macro or subsystem level.
- These design decisions can then be enforced.
-
- Mechanisms to do this will not be covered in this course. This is covered in the course on Project Development Methods. This is similar to the ability to perform separate compilation but at the subsystem level. It not only saves time but also increases independence.

Subsystem Partitioning (cont.)

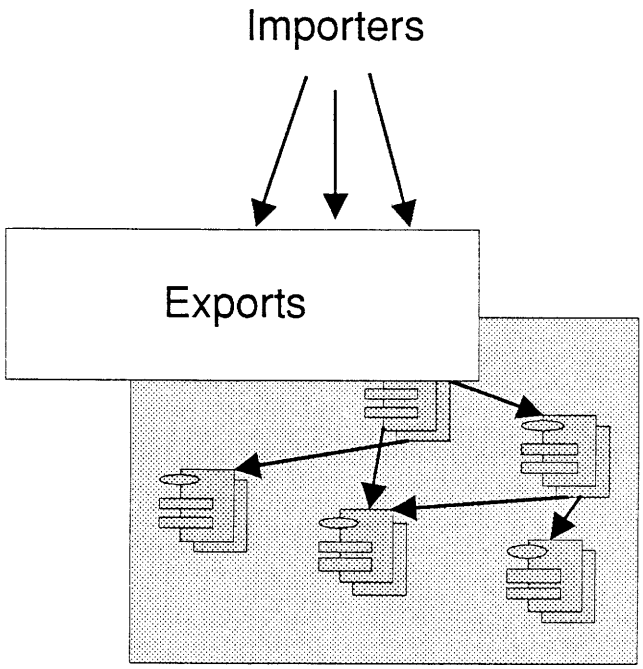


Notes on Subsystem Partitioning (cont.)

Special Instructions: Use this overlay to partition the previous slide into subsystems. Talk about the ability to reason about subsystems abstractly as whole functional units such as user interfaces or databases. Also, show how dependencies can be formed between subsystems.

Subsystems as Higher-Level Packages

- Subsystems can be thought of as meta-packages
  - — Exports form the visible part (specification) of the subsystem
  - — Imports are analogous to *with* clauses
  - — Subsystem implementation is analogous to package bodies



Notes on Subsystems as Higher-Level Packages

Transition:

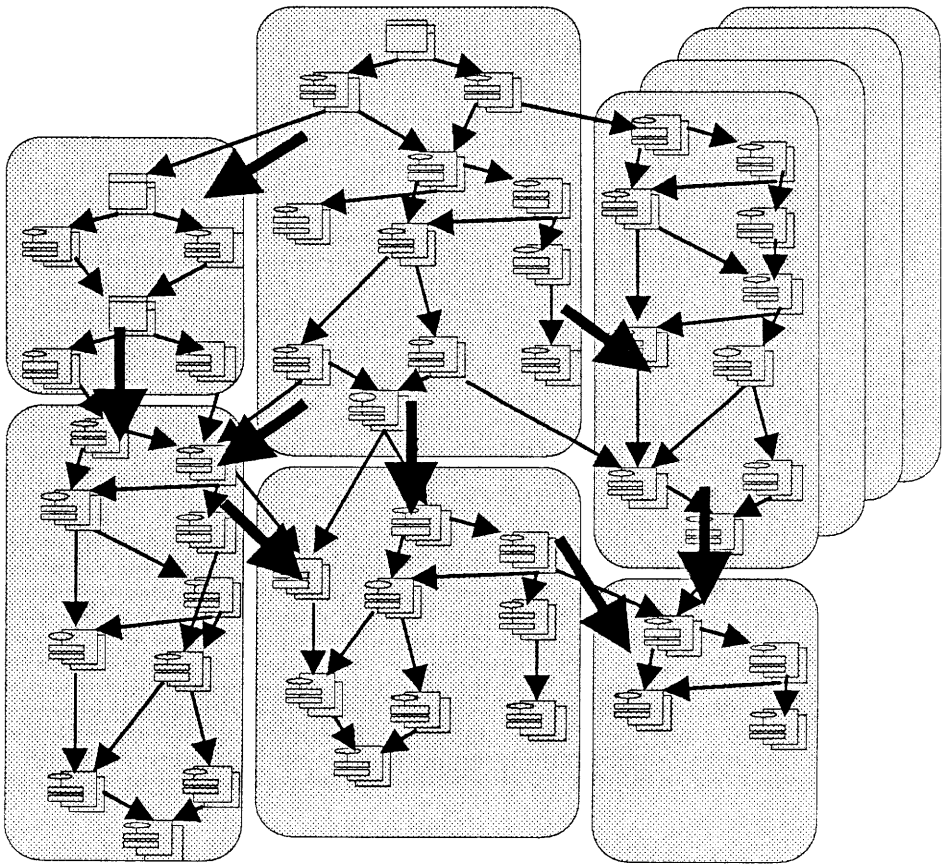
Special Instructions:

Key Points:

-
  -
  -
  -

Subsystem Releases

- Multiple releases define different implementations of a subsystem
  - — Each release is composed of a different configuration of Ada units
  - — Configurations are stored in *compiled* (coded) form



Notes on Subsystem Releases

Transition:

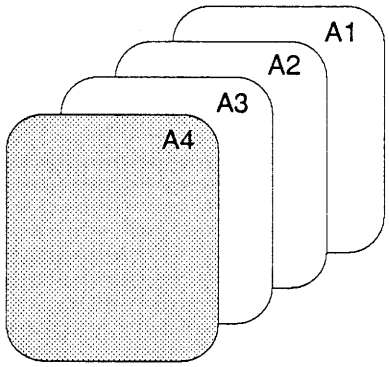
Special Instructions:

Key Points:

-
  -
  - In contrast to other CM systems that store configurations as source.

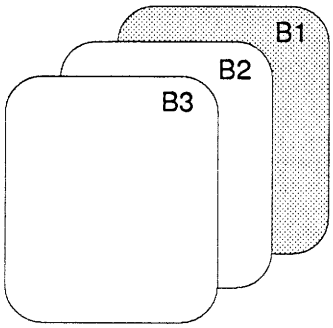
System Testing

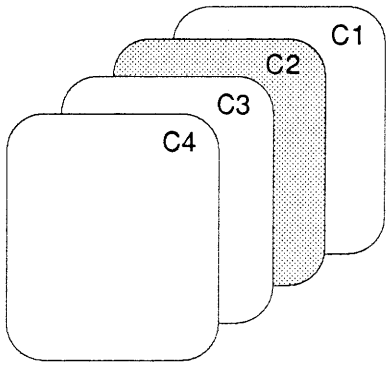
- System-build process is simpler and faster
  - — Execution table specifies system-level combinations
  - — No compilation is necessary
  - — Multiple tables specify multiple test combinations



Execution Table

Subsystem	Release
A	4
B	1
C	2





Notes on System Testing

Transition:

Special Instructions:

Key Points:

-
  -
  - Changing the system build is a matter of editing the execution table (later to be called an *activity*).
  - Each member of the programming team can independently test a separate combination of subsystems.

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
- Support parallel development and testing

Notes on Rational Subsystems

Transition: Rational Subsystems and CMVC are intended to address these issues.

Special Instructions:

Key Points: This slide summarizes the purpose of subsystems.

-
-
- Firewalling can localize recompilation when changes are made.
- We won't talk about how this is done in this course, but subsystem-level partitioning allows the design team to allocate subsystems to machines for development. The subsystem serves as the focus for configuration management and making releases from machine to machine for testing.
- By decoupling subsystem development from other development, programmers working on a subsystem can develop and test independently from other subsystems. There are also mechanisms to support parallelism inside a subsystem, which we will talk about later.

Rational CMVC

- Configuration management and version control
  - — Synchronizes changes to shared code
  - — Collects change history
  - — Supports parallel development paths within a subsystem
  - — Supports propagation of changes across development paths

Notes on Rational CMVC

Transition: This slide summarizes the purpose of CMVC. We will talk more about CMVC in later sections.

Special Instructions:

Key Points:

-
  -
  -
  -
  -

Course Outline

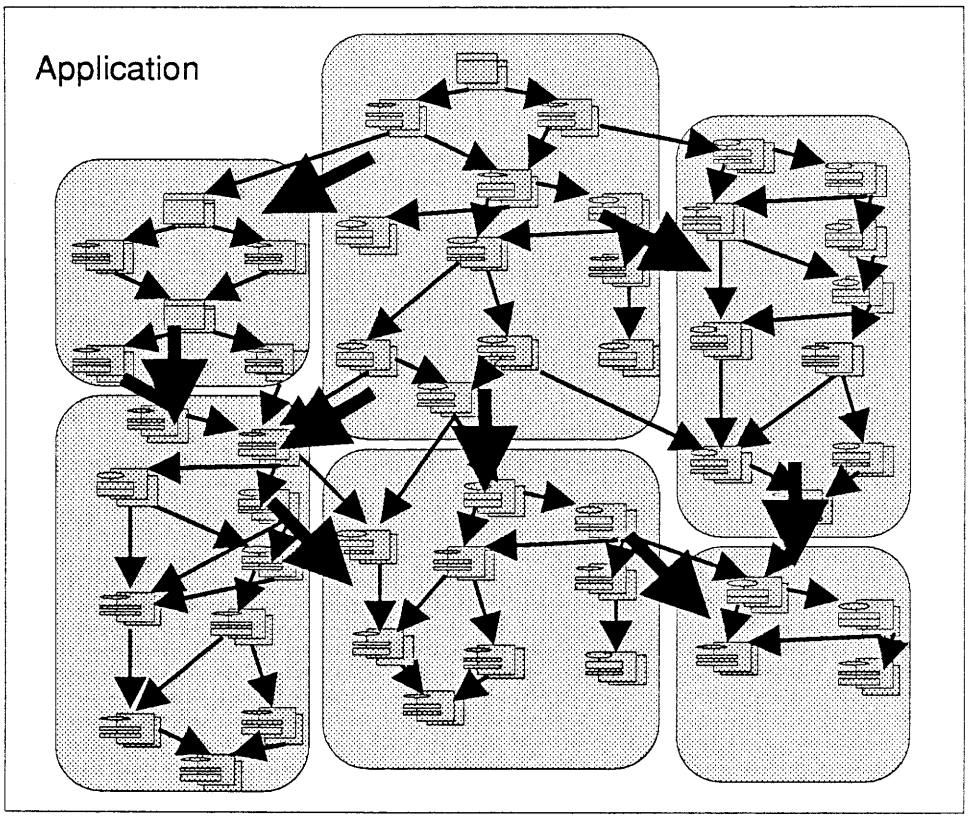
Subsystems and CMVC
Issues of Project Management
- Project Structuring with Subsystems
Subsystem Construction
Basic Development Methodology
Source Reservation with CMVC
Parallel Development with Subpaths

Notes on Course Outline

Transition:

Special Instructions:

Project Structure



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

Notes on Project Structure

Transition:

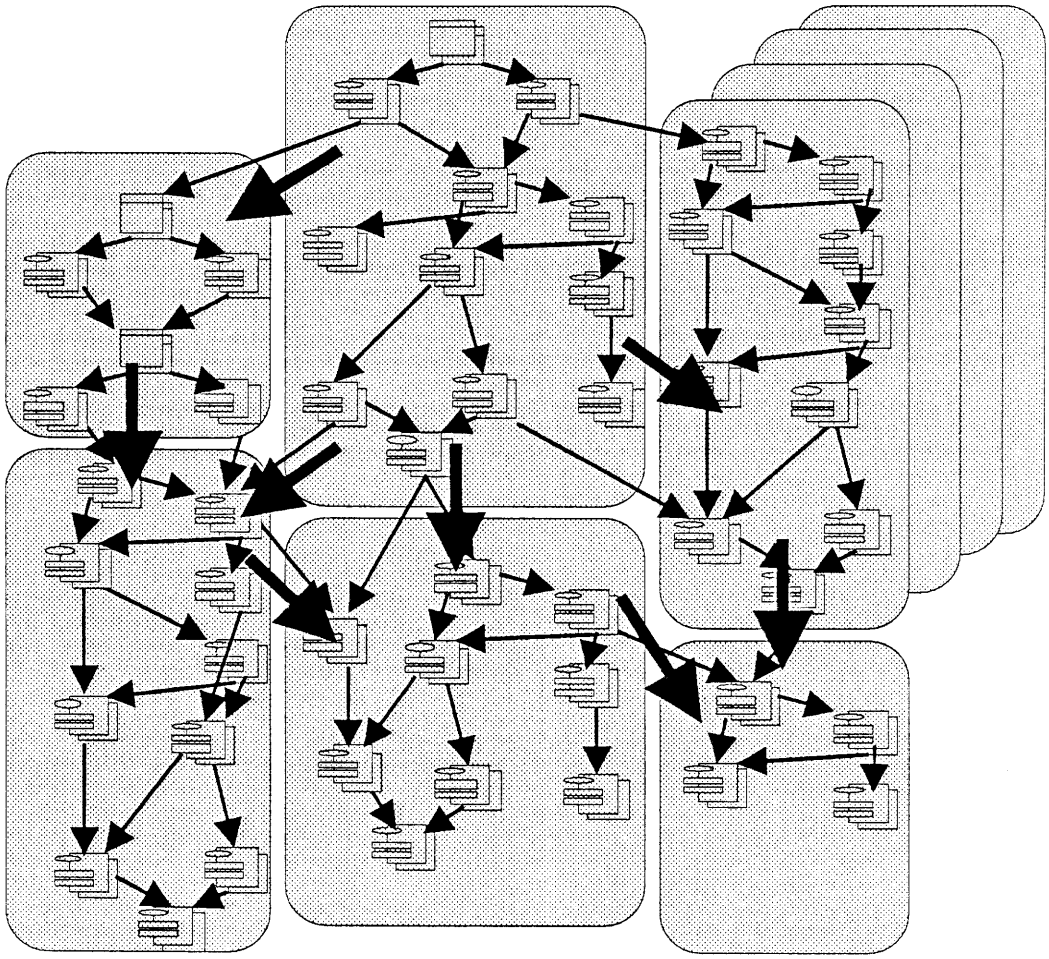
Special Instructions:

Key Points:

- These approximate the size and function of a CSCI in 2167 language.
-

Subsystem Releases

- Each subsystem release contains a different compiled configuration of Ada units
  - — Subsystem configurations are called *views*



Notes on Subsystem Releases

Transition:

Special Instructions:

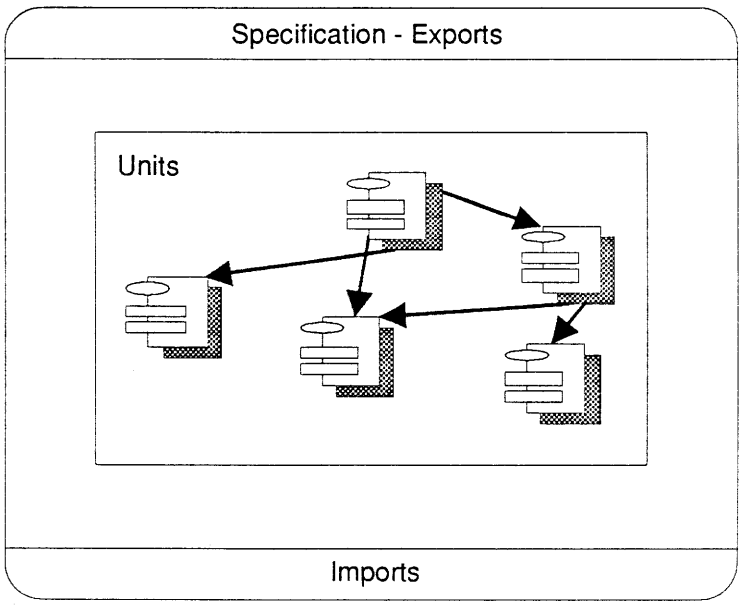
Key Points:

-
  -

Views

- Define a single instance or implementation of a subsystem
  - — Contain Ada units in a subdirectory called **`Units`**
  - — Define exports as a subset of the Ada unit specifications
  - — Define imports required from other subsystems in order to implement the view

A View:



Notes on Views

Transition:

Special Instructions:

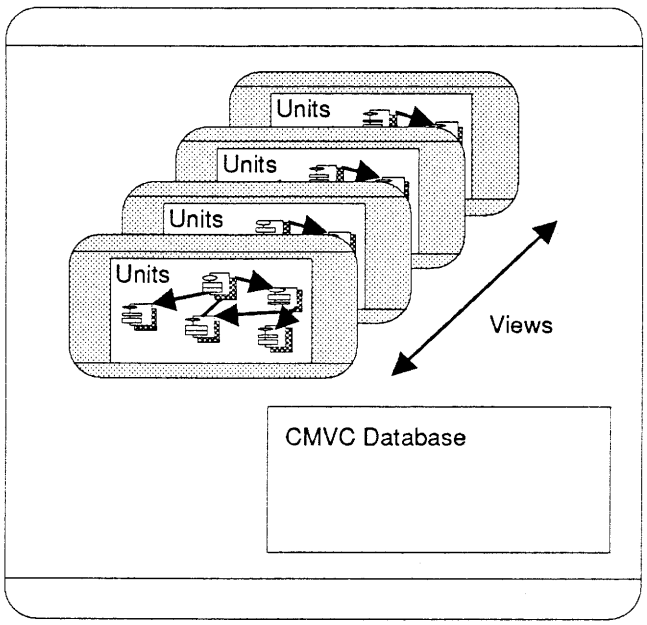
Key Points:

- The set of views that makes up the system is executed. Imports and exports are not executed.
  -
  -
  -

Subsystems

- Define a series of implementations (views)
- Collect development history and information on project management in a CMVC database
- Can include documentation, test cases, and other management information

A Subsystem:



Notes on Subsystems

Transition:

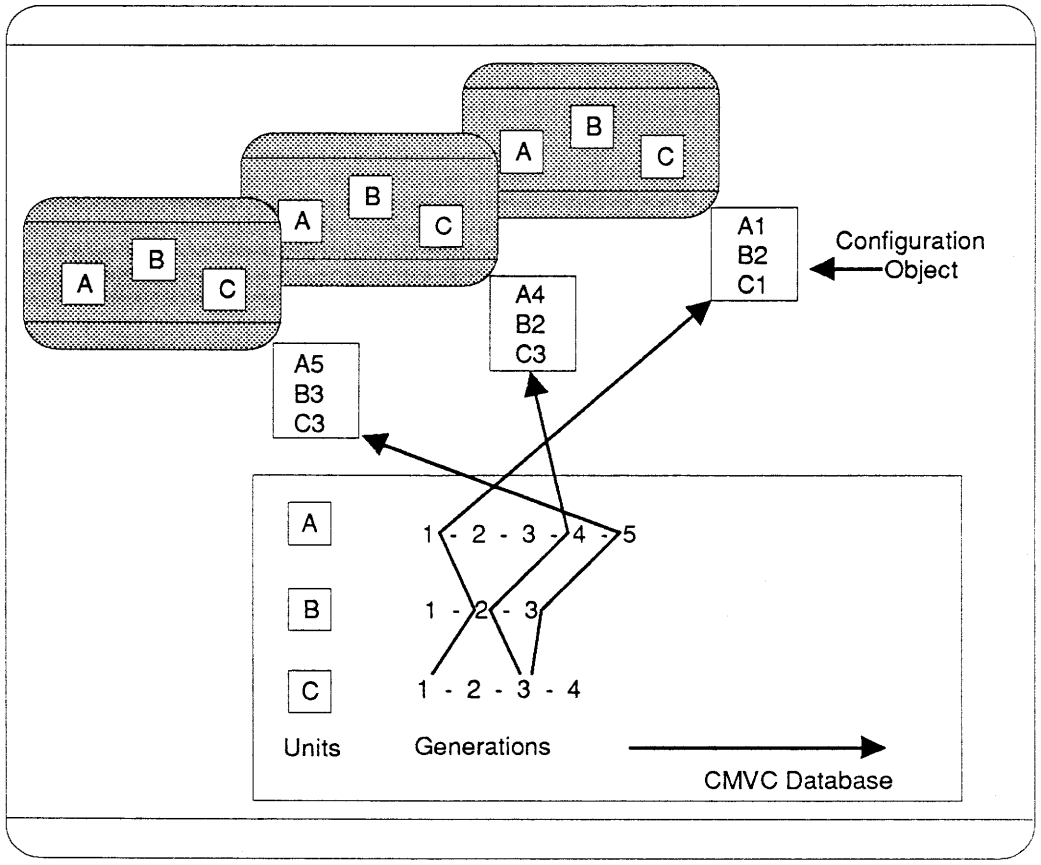
Special Instructions: Use the diagram to explain the major parts of a subsystem: exports, imports, implementation.

Key Points:

-
-
- Subsystems are used to manage all pieces of an application, not just the code.

CMVC Database

- The database stores a series of *generations* of objects
- A *configuration* is a combination of generations from the database
- A *view* is a real instance of a configuration



Notes on CMVC Database

Transition:

Special Instructions:

Key Points:

- A generation is different from a version in the directory system. This will be discussed later.
-
-

Dependencies between Subsystems

- At any one time, a single view defines a subsystem
- The view's exports define the set of Ada units available to other subsystems
- The view's imports define the set of available views from other subsystems
- Ada units in a view can reference exported units in other subsystem views if they have been imported

Notes on Dependencies between Subsystems

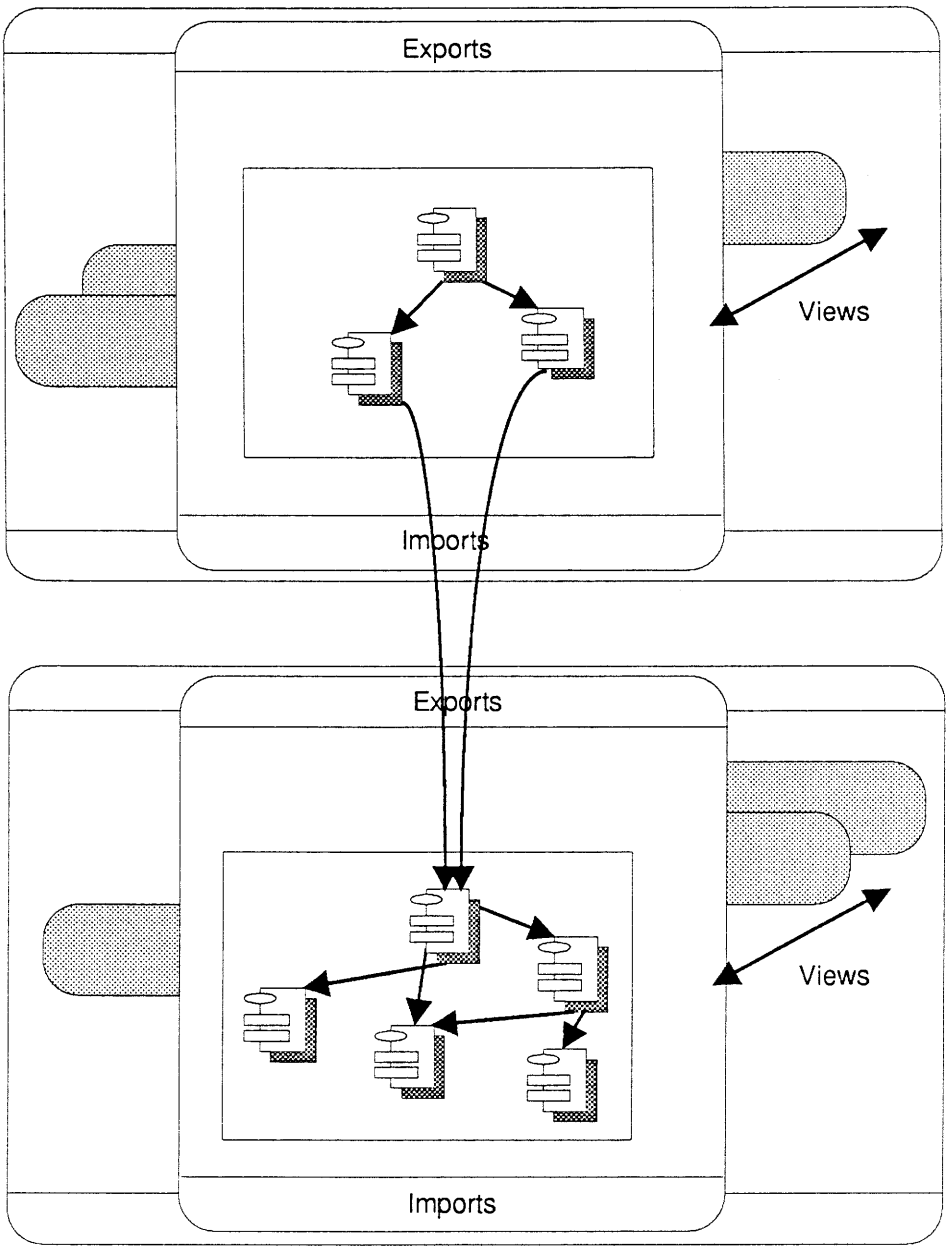
Transition:

Special Instructions: Use the diagram to explain the bullets.

Key Points:

-
-
-
-

Dependencies between Subsystems (cont.)



Notes on Dependencies between Subsystems (cont.)

Two-Part Views

- Advantages of two-part Ada packages
  - — Visibility control
  - — Separate compilation
- Spec/load views
  - — Define a single instance of a subsystem
  - — Support recompilation containment
  - — Spec views provide a separate specification of a subsystem similar to Ada package specifications
  - — Load views correspond to the *body* of the subsystem
  - — Spec views can have multiple load views

Notes on Two-Part Views

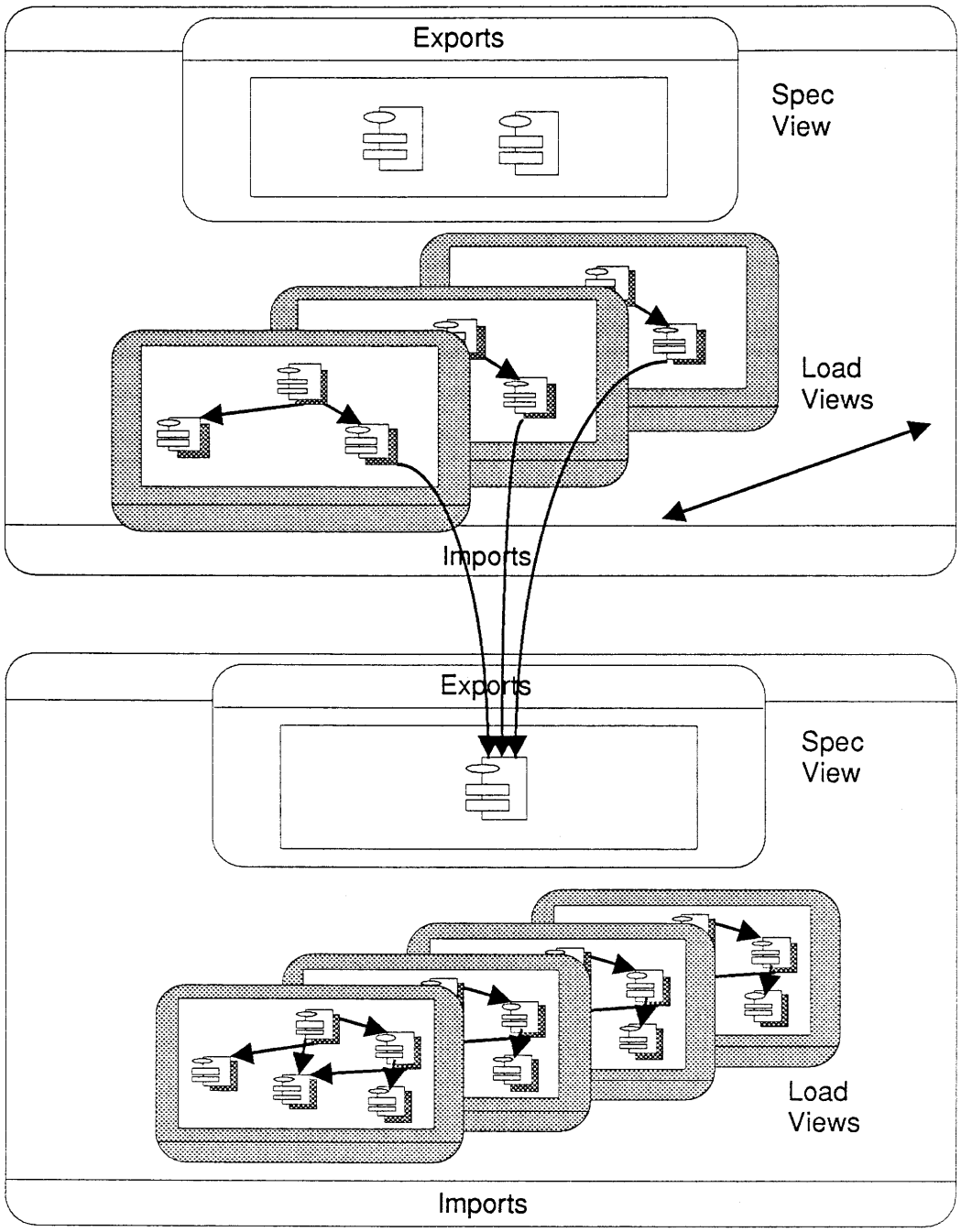
Transition:

Special Instructions:

Key Points: There are specific advantages in making two-part views in the same way that Ada packages have two parts.

-
  -
  -
-
  - We will not discuss spec and load views in detail in this course. They will be covered in great detail in the course on Project Development Methods.
  -
  - Together spec and load views provide the same compilation separation that Ada specs and bodies provide. Thus, compilation firewalls can be built by allowing a wide class of changes to load views (bodies) without requiring recompilation of clients that depend on the spec view.
  -
  - Load views offer the possibility of multiple bodies for the same spec. This allows for recombinant testing with different load views.

Two-Part Views (cont.)



Notes on Two-Part Views (cont.)

Subsystem Libraries

- Application library

```
!Application : World;
  Current_Release  : Activity;
  Model            : World;
  Report_Subsystem : Subsystem;
  System_Subsystem : Subsystem;
  Testing          : Directory;

= !APPLICATION (library)   @    World
```

- Subsystem library

```
!Application.System_Subsystem : Subsystem;
  Configurations : Directory;
  Rev1_0_1       : Load_View;
  Rev1_0_Spec    : Spec_View;
  Rev1_Working   : Load_View;
  State          : Directory;

= !APPLICATION.SYSTEM_SUBSYSTEM (library)   @    World
```

Notes on Subsystem Libraries

Transition: Now let's look at some screen images of subsystems and views. This may help you to better understand what subsystems actually are in the Environment.

Special Instructions:

Key Points: Use the screen images to point out various structures within subsystems and views. Stress that subsystems are composed of simple libraries and use the same compilation system as the basic Environment. Subsystems are merely a formalized library structure and a set of tools that manipulate that structure.

- Make sure to point out the two subsystems. We will talk about activities later.
- Note the three views; one spec view, one released load view, and one working view. Also mention the State directory and the Configurations directory. Don't explain too much other than that this is the place where the subsystem tools store information about the subsystem.

Subsystem Libraries (cont.)

- Spec-view **Units** directory

```
!Application.System_Subsystem.Rev1_0_Spec.Units : Directory;
  Unit : Pack_Spec;

= ...SYSTEM_SUBSYSTEM.REV1_0_SPEC.UNITS (library)      Directory
```

- Load-view **Units** directory

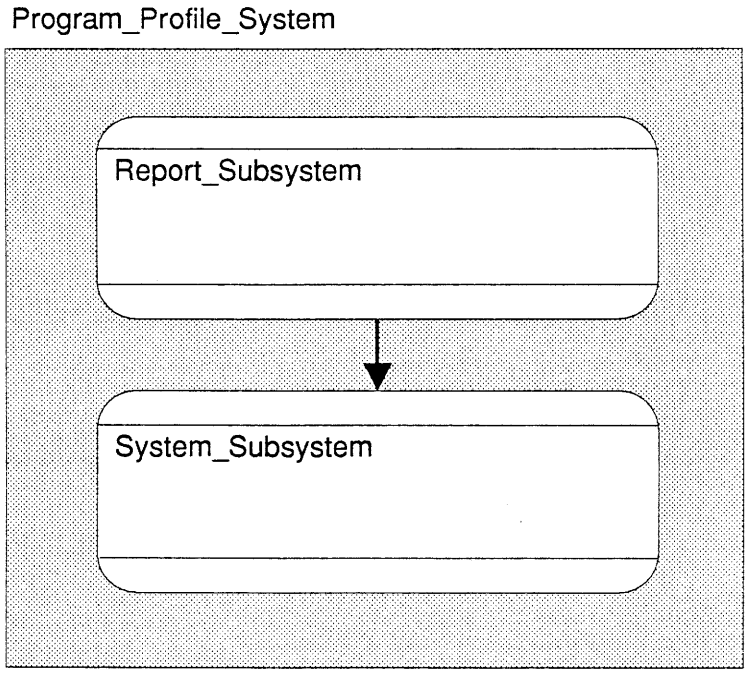
```
!Application.System_Subsystem.Rev1_Working.Units : Directory;
  Line : Pack_Spec;
  Line : Pack_Body;
  Unit : Pack_Spec;
  Unit : Pack_Body;

= ....SYSTEM_SUBSYSTEM.REV1_WORKING.UNITS (library)        Directory
```

Notes on Subsystem Libraries (cont.)

- Note that the spec view contains no bodies.
- The load view contains bodies and other nonexported packages.

Sample Application

- The `Program_Profile` application counts various kinds of lines in an Ada program
- The application consists of two subsystems



Notes on Sample Application

Transition:

Special Instructions:

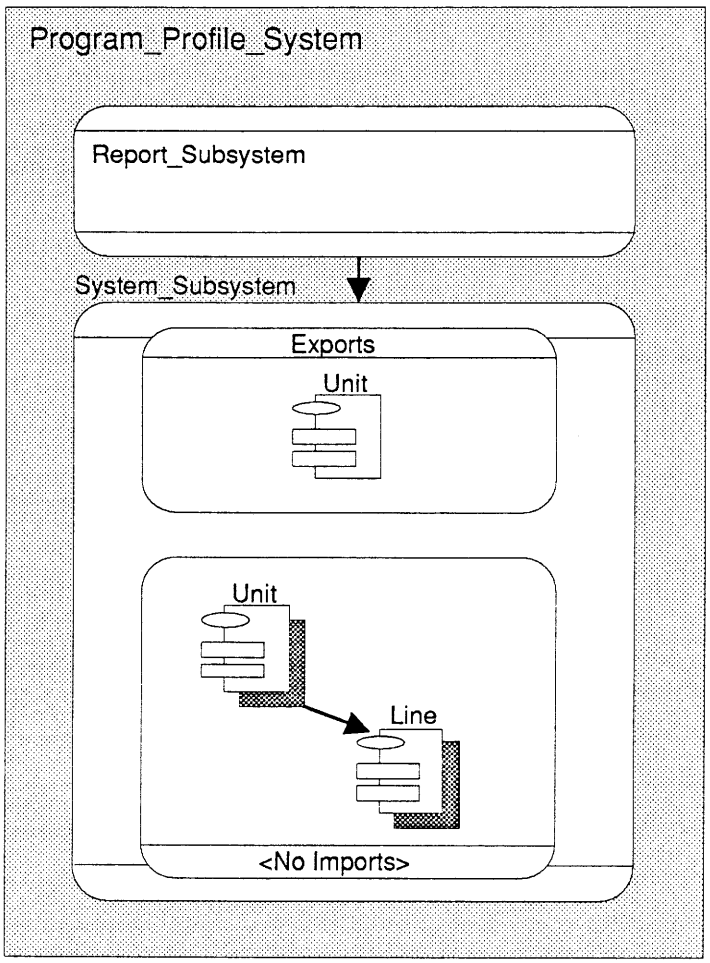
Key Points: Introduce the `Program_Profile` program—what it does and its initial subsystem structure.

•

•

Sample Application (cont.)

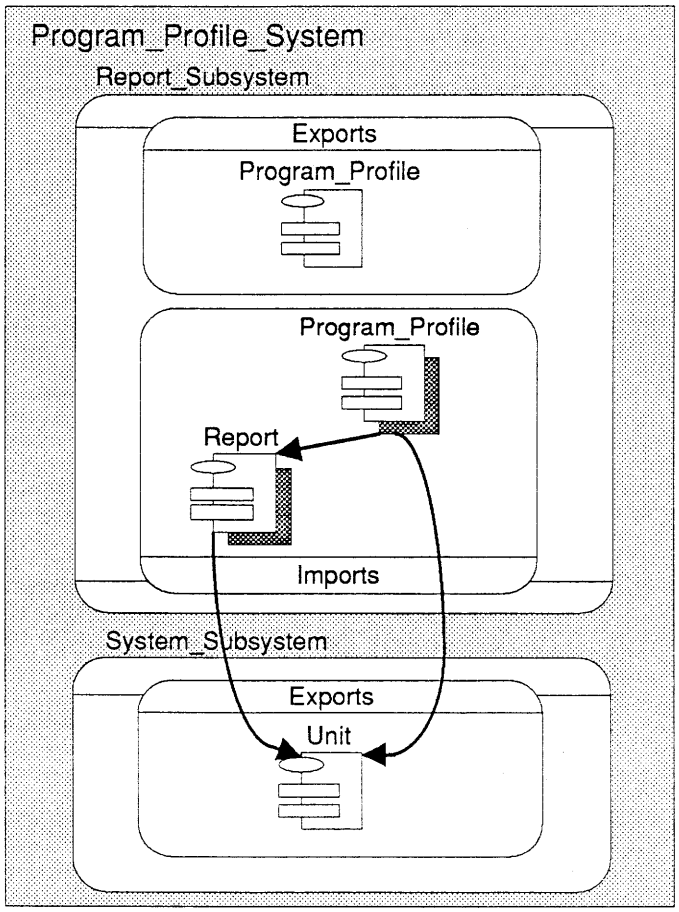
- **`System_Subsystem`** contains the Ada units that analyze each line and collect the statistics for an Ada system



Notes on Sample Application (cont.)

Sample Application (cont.)

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



Notes on Sample Application (cont.)

Terminology Summary

- *Application*: An entire software system partitioned into subsystems
- *Subsystem*: A large partition of an application
- *View*: A single instance or implementation of a subsystem
- *CMVC database*: Subsystem-level object that collects configuration management information on units in views
- *Configuration*: A set of generations of units in the subsystem
  - — A view contains a specific configuration
  - — Configurations are formally kept in configuration objects

Notes on Terminology Summary

Transition: This is a recap of the terminology we have defined to this point in the course.

Special Instructions:

Key Points:

-
-
-
-
-
  -
  -

Terminology Summary (cont.)

- *Generation*: One in a series of object *versions* stored in the CMVC database
  - — A new generation is created whenever an object is checked in
- *Exports*: The units that are made visible outside the subsystem to other client subsystems
  - — A spec view is a realization of a subsystem's exports
- *Imports*: A list of spec views referenced by a given subsystem
- *Layer*: Sometimes used to define a group of subsystems that form some higher-level partition of the application

Notes on Terminology Summary (cont.)

-
  -
-
  -
-
-

Course Outline

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

Notes on Course Outline

Transition:

Special Instructions:

Subsystem Partitioning

- Suggested partitioning criteria
  - — A subsystem should be a complete, logical component of the system
  - — A subsystem should have well-defined, narrow interfaces
  - — The total number of subsystems should be a manageable number
  - — Subsystem interface packages should export private types
  - — Subsystem interface packages should avoid reexporting declarations from other subsystem interfaces
  - — A subsystem eventually should contain a manageable amount of code (5K–25K lines)

Notes on Subsystem Partitioning

Transition: These are some guidelines for designing a system into what will eventually be developed with Rational Subsystems.

Special Instructions:

Key Points:

-
  - It should have one consistent functional objective.
  - This translates into a small number of package interfaces. Subsystem interfaces should define private types whenever possible to take advantage of closed private parts.
  - Otherwise the complexity starts to overtake you again.
  - This will provide the ability to firewall compilation and not let lower-level changes propagate through to upper-level subsystems.
  - This can limit recompilation to the local subsystem.
  - This is a suggested order of magnitude.

Subsystem Partitioning (cont.)

- A subsystem should have 1–5 developers working on it
- It is better if subsystems are separately testable

Notes on Subsystem Partitioning (cont.)

- Again, a suggestion. More is possible.
-

Method for Building Systems from Bottom Up

- Basic method
  - — Build an initial load view for the lowest subsystem first
  - — Copy any available Ada units into the `Units` directory of the load view
  - — Import lower-level spec views as necessary
  - — Compile the Ada units in the load view
  - — Establish exports and build a spec view
  - — Add each new subsystem on top of existing subsystems
  - — Build all subsystems and their dependencies in a single pass

Notes on Method for Building Systems from Bottom Up

Transition:

Special Instructions:

Key Points:

-
  -
  -
  - ONLY SPEC VIEWS SHOULD/CAN BE IMPORTED.
  - Use [Code (This World)].
  - This will define the Ada units that can be imported into other subsystems.
  -
  -

Subsystem Construction

- Build a subsystem and construct an initial load view: `Cmvc.Initial`

  Important parameters:

  — `Subsystem`: Specifies name of the subsystem

  — `Subsystem_Type`: Specifies whether to build a spec/load view or a combined view; this course will use the value `Cmvc.Spec_Load`

Notes on Subsystem Construction

Transition:

Special Instructions:

Key Points:

-
  -
  -

Spec-View Construction

- First define exports
  - — List of units to be exported is in a file called *View_Name*`.State.Exports`
  - — File must be edited to list only those units to be exported in the spec view
- Then create a spec view: `Cmvc.Make_Spec_View`

  Important parameters:
  - — `From_Path`: Specifies name of the load view from which to build the spec view
  - — `Spec_View_Prefix`: Specifies the prefix name for the new spec view

Notes on Spec-View Construction

Transition:

Special Instructions:

Key Points:

-
  - All views have a subdirectory called `State`. The `Exports` file is in this directory.
  - This is very similar to using an indirect file.
-
  - This is the same view whose `Exports` file has been edited.
  - This is simply Rev1 of Rev1_0_Spec, for example.

Importing

- Import a view from another subsystem:
  `Cmvc.Import`

  Important parameters:

  — **`View_To_Import`**: Specifies name of the view to import

  — **`Into_View`**: Specifies importing view; if the cursor is anywhere inside a view or any of its subobjects, that view will import the new view

Notes on Importing

Transition:

Special Instructions:

Key Points:

- IMPORTANT: Views import other views. They do not import subsystems. Only one view can be imported from each subsystem at any one time.
  -
  -

Subsystem Destruction

- Destroy a view within a subsystem:
  `Cmvc.Destroy_View`
  - — `Compilation.Destroy` or `Library.Destroy` will not work
- Destroy a subsystem: `Cmvc.Destroy_Subsystem`
  - — All views must be destroyed first

Notes on Subsystem Destruction

Transition: We all make mistakes, occasionally.

Special Instructions:

Key Points:

-
  - This is because one of the state units must be checked out first. `Compilation.Destroy` does not know how to do this.
-
  -

Exercise: Building an Application in Subsystems

Build a subsystem structure for the `Program_Profile` application using the project world called `Subsystem_Application` in your home library. Partition units into two subsystems as shown in the previous slides.

1. Create the lower (`System_Subsystem`) subsystem inside the `Subsystem_Application` world. Create a load view for the initial view.

2. Copy the appropriate Ada units for the lower subsystem from the `Program_Profile_System` in your home library into the load view.

3. Compile the view.

4. Edit the exports file and create a spec view for this subsystem.

Notes on Exercise: Building an Application in Subsystems

Special Instructions: Go over each step in class before beginning. This exercise is intended to be be very free-form. Note that the reference to the model in `Cmvc.Initial` *must be fully qualified*—that is: `!Users.Advanced_N.Subsystem_Application.Model`.

Refer to the previous slides showing the sample application for a picture of the final subsystem partition. Note also that the instructions apply once to each subsystem.

Exercise: Building an Application in Subsystems (cont.)

5. Create the upper (`Report_Subsystem`) subsystem inside the `Subsystem_Application` world. Create a load view for the initial view.
6. Copy the appropriate Ada units for the upper subsystem from the `Program_Profile_System` in your home library into the load view.
7. Set up the necessary import to the lower subsystem.
8. Compile the view.
9. Create a spec view for the upper subsystem.
10. Decide what to do with `Test_Input` units.

Notes on Exercise: Building an Application in Subsystems (cont.)

Course Outline

Subsystems and CMVC
- Issues of Project Management
- Project Structuring with Subsystems
- Subsystem Construction

- Basic Development Methodology
- Source Reservation with CMVC
- Parallel Development with Subpaths

Transition:

Special Instructions:

Types of Views

- Released views
  - — Are frozen views that cannot be modified
  - — Are used for execution testing by other higher-level subsystems
  - — Sequences of released views are called *paths*
- Examples of paths
  - — Multiple-target development; each target is a separate *path*
  - — Development of major product versions
- Working views
  - — Are unfrozen views with ongoing changes
  - — Are often allocated to each subsystem developer working within a subsystem

Notes on Types of Views

Transition:

Special Instructions:

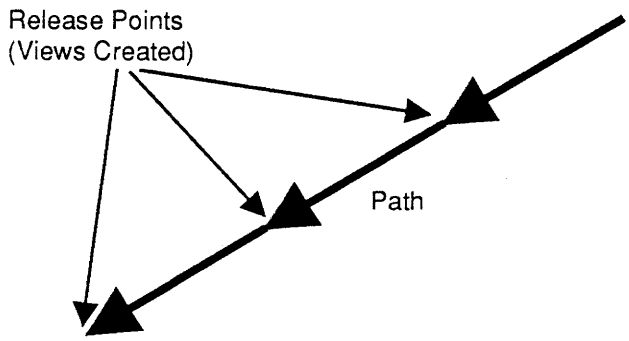
Key Points:

-
  -
  -
  -
-
  - The first would be developing the same code to run on multiple targets with some target-dependent code.
  - The second example is the Gamma-Delta-Epsilon model.
- Releases are made from a tested working view.
  -
  - Doing this supports parallelism within a subsystem. This is described in the last section of this course.

Releasing

- Changes are made to units in working views
- Releases are made when the current working view is tested and stable
  - — Each subsystem can be released independently
- Release a new load view: **`Cmvc.Release`**

  Important parameter:
  - — **`From_Working_View`**: Specifies name of view to release from
  - — Use all other defaults



Notes on Releasing

Transition:

Special Instructions:

Key Points:

-
-
  - This allows each development team to match their release schedule to their needs. This is the essence of parallelism across subsystems.
- This is the same as with the basic method.
  - In our scenario, this would be the integration directory.
  -

Execution

- System combinations of views for execution are specified with an activity object
- 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          |    Spec View     |    Load View  | Context

REPORT_SUBSYSTEM   |    REV1_0_SPEC   |    REV1_0_1   | !APPLICATION
SYSTEM_SUBSYSTEM   |    REV1_0_SPEC   |    REV1_0_1   | !APPLICATION

= ... SOLUTION.CURRENT_RELEASE'V(1)  (activity)      (All Data by Subsystem)
```

Notes on Execution

Transition:

Special Instructions:

Key Points:

- Previously called an *execution table*.
-
  -
  -
- Explain what is in each column.

Current Activity

- Users can have multiple activities with different view combinations
- Current activity is the activity to be used by the Environment when executing a program
- 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`

- System default activity is in
  `!Machine.Release.Current.Activity`

  — Contains entries for Rational interfaces delivered as subsystems

Notes on Current Activity

Transition:

Special Instructions:

Key Points:

-
-
-
  -
  -
-
  - Students may need to reset their activity to this in order to execute certain commands.

Modification of Activities

- Activities are a subclass of file objects
- Activity files must be saved to make changes permanent
- Commands
  - Create an activity: **`Activity.Create`**
  - Edit the current activity: **`Activity.Edit`**
  - Add a new entry to an activity: [Object] - [I]
  - Delete a selected entry in an activity: [Object] - [D]
  - Edit the selected subsystem entry: [Edit]
  - Save changes to an activity: [Enter]

Notes on Modification of Activities

Transition:

Special Instructions:

Key Points:

-
-
- Note the similarity to other Environment functionality.
  -
  -
  -
  -
  - A Command window will be created with the `Activity.Change` command.
  -

Exercise: Managing Activities

Execute the `Program_Profile` application with two different implementations of the same subsystem.

1. Create a frozen release for each subsystem.
2. Create an activity within `Subsystem_Applications` referencing the newly released views.
3. Make the activity the default activity and execute `Program_Profile` in the spec view of `Report_Subsystem`.
4. Modify the body of `Report` in the working view of the `Report_Subsystem` to use `x` characters for the boundaries instead of - characters.
5. Test this change by editing the activity to reference the working view.
6. Create a new release of the `Report_Subsystem` when testing is complete.

Notes on Exercise: Managing Activities

Work Orders

- Are a tool of project management used for defining units of work
  - — Problem reports to be fixed
  - — Feature additions
  - — Follow-up documentation and testing
- Have three states
  - — Defined: Work has been specified
  - — Open: Work is ongoing
  - — Closed: Work is completed

Notes on Work Orders

Transition:

Special Instructions:

-
  -
  -
  - When the development work order is completed, new work orders for documentation updates and testing can be generated.
-
  -
  -
  -

Work Orders (cont.)

- Set a default work order to record all development: `Work_Order.Set_Default`
- View a work order: `Work_Order.Edit` or [Definition]
  - — Expand an elided entry: [Object] - [!]
  - — Visit the full list of history entries: [Definition]

Notes on Work Orders (cont.)

-
- Use [Definition] on a work-order object in a directory.
  - Some entries are not completely expanded.
  -

Work-Order Lists

- Are collections of like work orders
- Can be used to organize
  - — Problem class
  - — Single user's work
  - — Subsystem *to do* list
- View a work-order list: `Work_Order.Edit_List` or [Definition]

Notes on Work-Order Lists

Transition:

Special Instructions:

Key Points:

-
-
  -
  -
  -
-

Course Outline

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

Transition: With multiple people working in a subsystem, we need to provide some way to synchronize their activities.

Special Instructions:

Controlled Objects

- Are managed by the CMVC system
- Have *reservation tokens* requiring `Check_Out` and `Check_In`
- Have history collected
  - — Checkout and checkin times
  - — Who checked out the unit
  - — Other information
  - — Have generations recorded as deltas
- Control objects in a view: `Cmvc.Make_Controlled`
  - — Use `What_Object` parameter to define which objects should be controlled (wildcards can be used)

Notes on Controlled Objects

Transition:

Special Instructions:

Key Points:

-
- `Check_Out` reserves the right to modify a controlled object.
-
  -
  -
  - Examples include when it will be checked in and any comments that the user may make.
  - A new generation is created on `Check_Out`.
-
  -

Modification of Controlled Objects

- Get access to edit by taking the reservation token: `Cmvc.Check_Out`

  Important parameters:

  — `What_Object`: Specifies which objects to check out

  — `Comments`: Specifies user comments to be saved in CMVC database as history

  — `Expected_Check_In_Time`: Specifies time of expected checkin

- Checkout

  — Automatically pulls latest generation from the CMVC database

  — Demotes unit to overwrite with new generation (`Allow_Demotion` parameter must be `True`)

  — Creates next generation

Notes on Modification of Controlled Objects

Transition:

Special Instructions:

Key Points:

-
  -
  -
  - This will accept any legal time image.
-
  - This happens with multiple subpaths, discussed in the next section.
  - It is important to distinguish between versions and generations.
  -

Modification of Controlled Objects (cont.)

- Return reservation token: `Cmvc.Check_In`

  Important parameters:

  — `What_Object`: Specifies which objects to check in

  — `Comments`: Specifies user comments to be saved in CMVC database as history

Notes on Modification of Controlled Objects (cont.)

-
  -
  -

Exercise: Modifying Controlled Objects

Make controlled all units in the initial views of each subsystem that you created and experiment with `Cmvc.Check_In` and `Cmvc.Check_Out`.

1. Use `Cmvc.Make_Controlled` to control each unit in each working view of each subsystem.
2. Visit the body of the `Line` package. Try to incrementally add an Ada comment to the unit.
3. Check out the unit, providing some comment as to why you are checking out the unit.
4. Incrementally add the Ada comment to the unit.
5. Check the unit back in, providing additional comments about your changes.

Notes on Exercise: Modifying Controlled Objects

Special Instructions: Comments will be added so as not to change the execution of the program.

Course Outline

Subsystems and CMVC
- Issues of Project Management
- Project Structuring with Subsystems
- Subsystem Construction
- Basic Development Methodology
- Source Reservation with CMVC

- Parallel Development with Subpaths

Transition: Now we need to discuss a methodology that allows us to support parallel development within a subsystem.

Special Instructions:

Parallel Development Methodology

- Parallel development within subsystems is supported through use of subpaths
- Each developer is given a separate subpath (working view)
  - — Each subpath is a separate copy of the Ada units in the subsystem
  - — All objects are joined to enforce synchronization across working views
- An integration subpath can used to collect and test latest unit generations
  - — Releases are made from the integration working view

Notes on Parallel Development Methodology

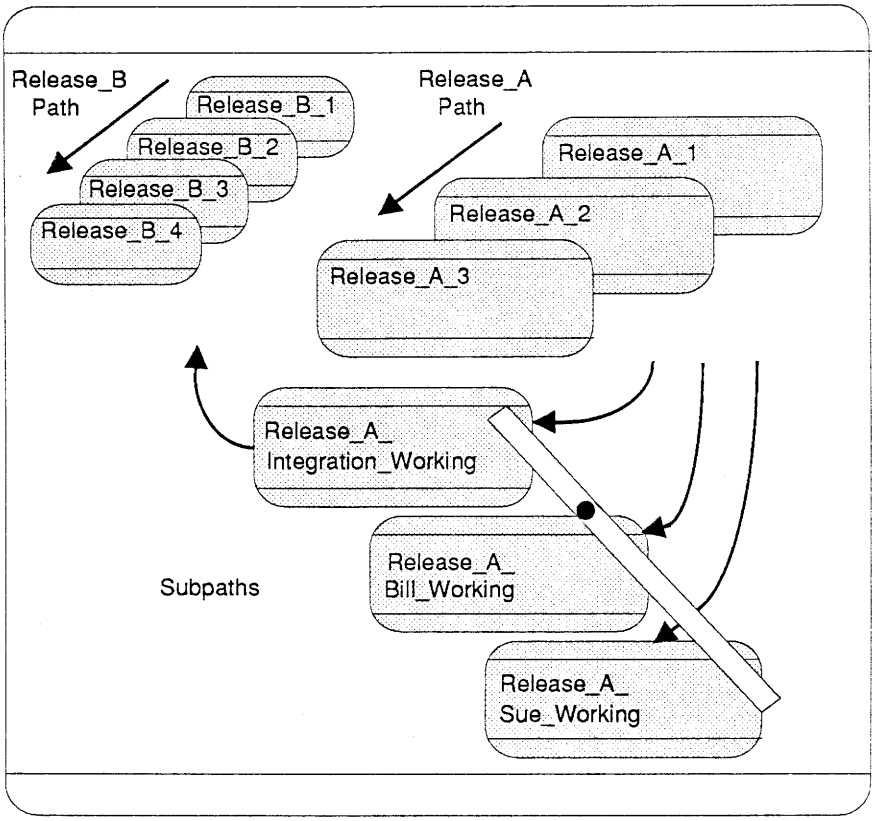
Transition:

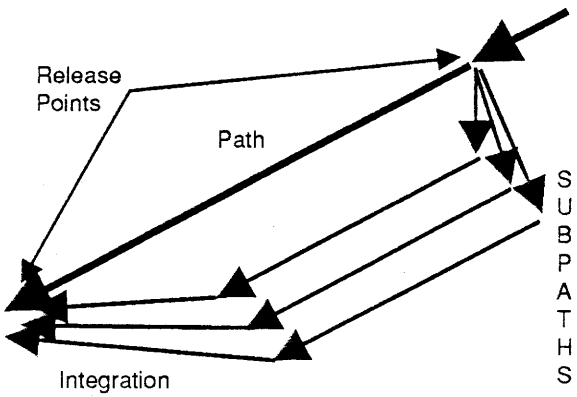
Special Instructions:

Key Points:

-
-
  - Remember that subsystems and views can be thought of as the same at some level of abstraction.
  -
- This is actually another subpath joined with the other subpaths. Releases into the path are made from the integration directory.
  -

Parallel Development Methodology (cont.)





Notes on Parallel Development Methodology (cont.)

Creation of Subpaths

- Select an initial working view
- Create additional working views and join units in the new view with units in the source view:
  `Cmvc.Make_Subpath`

  Important parameters:

  — **`From_Path`**: Specifies the initial view from which to create the new subpath

  — **`New_Subpath_Extension`**: Specifies the name of the subpath; for example:

  `Rev1_Ext_Working`

Notes on Creation of Subpaths

Transition:

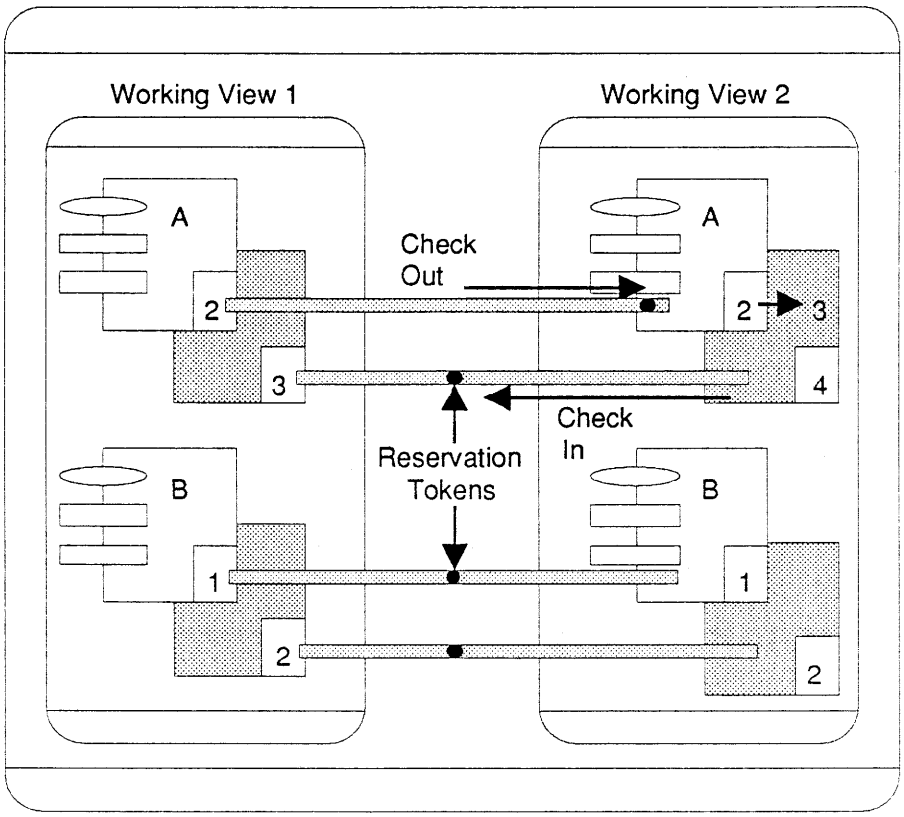
Special Instructions:

Key Points:

-
-
  -
  -

Joined Objects

- Joined objects share a single reservation token
  - — Ada units and text objects in multiple working views can be *joined*
  - — Joined objects have synchronized access (only one working view can modify a joined object at any one time)



Notes on Joined Objects

Transition:

Special Instructions:

Key Points:

-
  -
  - Only one view may check out the object.

Change Propagation

- Each working view will change subsystem objects over time
  - — Changes are recorded in CMVC database on `Check_In`
  - — Other subpaths may want to propagate changes into their working view
  - — Integrator will want to merge all changes for test
- All generations are available for restoration from the database

Notes on Change Propagation

Transition:

Special Instructions:

Key Points:

-
  -
  -
  -
-

Change Propagation (cont.)

- Overwrite existing generation with new generation from the CMVC database:
  `Cmvc.Accept_Changes`

  Important parameters:

  — **`Destination`**: Specifies a set of new objects, all in one working view; the name of a view also can be used

  — **`Source`**: Specifies which generation to accept; use the default (**`"Latest"`**) to get the most recent generation

  — **`Allow_Demotion`**: Specifies that demotion of updated unit be allowed if necessary

- Returning to a previous generation can be performed with **`Cmvc.Revert`**

Notes on Change Propagation (cont.)

-
  - Specifying the view will accept changes for all objects in the view. This would be especially useful for the integration directory.
  -
  - Note that this demotion may cause other units to be demoted.
-

Releasing

- Releases should be made when the integration view is stable and tested
  - — Frozen view is created for use by other subsystems
  - — New view can now be executed with an activity
- Release a new view: **`Cmvc.Release`**

  Important parameter:
  - — **`From_Working_View`**: Specifies name of view to release from

Notes on Releasing

Transition:

Special Instructions:

Key Points:

-
  -
  -
- This is the same as with the basic method.
  - In our scenario, this would be the integration directory.

History

- Commands
  - `Cmvc.Show_History`: Shows `Check_Out` and `Check_In` comments and actual changes made
  - `Cmvc.Show_All_Checked_Out`: Lists all checked-out objects and who has reserved them
  - `Cmvc.Show_Out_Of_Date_Objects`: Shows all objects that are not at the most recent generation

Notes on History

Transition:

Special Instructions:

Key Points:

-
  -
  -
  - This will list all objects in a working view that are not the most recent generation. As others make changes in other working views, objects will become out of date.

Exercise: Creating Parallel Development Paths

Create additional subpaths within the subsystem **`System_Subsystem`** and experiment with **`Check_In`**, **`Check_Out`**, **`Accept_Changes`**, and **`Release`**.

1. If you did not complete the previous exercise making all units controlled, do so now.
2. Create three new working subpaths with **`Cmvc.Make_Subpath`**. Name them **`Sue`**, **`Bill`**, and **`Integration`**.
3. Begin in Sue's working view and make some change that does not affect the execution of the program (a comment, perhaps).
4. Move to Bill's view and try to check out the same unit.
5. Use **`Show_All_Checked_Out`** to see who has checked out the unit.

Notes on Exercise: Creating Parallel Development Paths

Special Instructions: You may want to organize groups of two, with each person playing the role of a single developer in the subsystem.

Exercise: Creating Parallel Development Paths (cont.)

6. Return to Sue's working view and check the unit back in. Provide good commentary.
7. Return to Bill's view and accept the changes for that unit. Verify that the changes have propagated.
8. Make some other change to a unit, providing good commentary.
9. Move to the `Integration` working view and accept all changes.
10. Release the integration view with `Cmvc.Release`.
11. Modify the default activity to reference the new release and verify that the program still executes the same.

Notes on Exercise: Creating Parallel Development Paths (cont.)

Additional Terminology

- *Model*: A template world defining links, access control, and other baseline attributes for subsystem views
- *Release*: A frozen, nonworking view generated by the CMVC tools
- *Working view*: A view in which ongoing development occurs
- *Path*: A series of released views
- *Activity*: A table listing subsystems and specific spec and load views within those subsystems
  - — Activities specify which combination of released load views will execute together

Notes on Additional Terminology

Transition:

Special Instructions:

Key Points:

-
-
-
-
-
  -

Additional Terminology (cont.)

- *Controlled object*: An object for which generation history is collected in the CMVC database
  - — Controlled objects must be checked out before editing
- *Reservation token*: A logical token that all controlled objects reserve during checkout and return on checkin
  - — Two people cannot reserve the reservation token at the same time
- *Joined objects*: Objects that share a single reservation token
  - — Objects are usually *joined* across multiple working views to ensure synchronization

Notes on Additional Terminology (cont.)

-
  -
-
  -
-
  -

Additional Terminology (cont.)

- *Subpaths*: Multiple working views within the same subsystem
  - — Each developer within a subsystem is given a separate subpath
  - — Synchronization is accomplished with joined objects
  - — Change propagation is possible
  - — The next release in a path is generated from the integration of changes in all subpaths

Notes on Additional Terminology (cont.)

•

- –
- –
- –
- –

Notes

Notes

Notes

Notes