A02-5. *Code View*

2014

Sungwon Kang

A. Purpose

- To facilitate the construction, integration, installation and testing
  - Supports:
    - Daily concurrent development tasks
    - Constructing and releasing parts of the system to different teams for installation, integration and testing
    - Ease of maintenance or change
    - Enforcement of architectural design decision
    - CM
- Describe how the implementation is organized
  - Source components implement individual elements in the module view
  - Deployment components instantiated to runtime entities in the execution view

    E.g.) executables, libraries, configuration files
  - Describe design decisions related to CM, multiple releases and testing
- Language-dependent view

B. Context

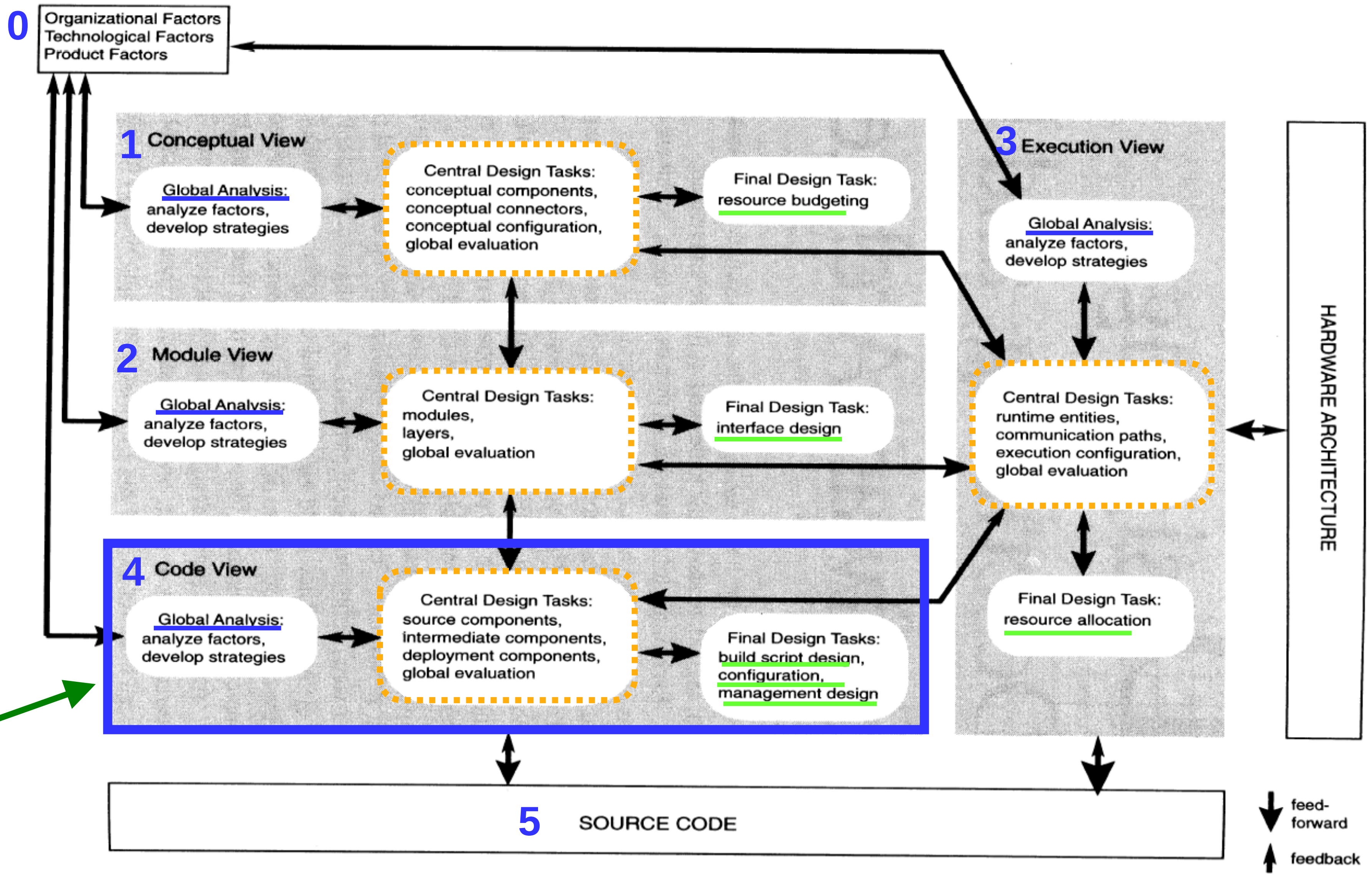

0
Organizational Factors
Technological Factors
Product Factors
1 Conceptual View
Global Analysis: analyze factors, develop strategies
Central Design Tasks: conceptual components, conceptual connectors, conceptual configuration, global evaluation
Final Design Task: resource budgeting
2 Module View
Global Analysis: analyze factors, develop strategies
Central Design Tasks: modules, layers, global evaluation
Final Design Task: interface design
3 Execution View
Global Analysis: analyze factors, develop strategies
Central Design Tasks: runtime entities, communication paths, execution configuration, global evaluation
Final Design Task: resource allocation
4 Code View
Global Analysis: analyze factors, develop strategies
Central Design Tasks: source components, intermediate components, deployment components, global evaluation
Final Design Tasks: build script design, configuration, management design
HARDWARE ARCHITECTURE
5 SOURCE CODE
feed-forward
feedback

Figure II.1. Overview of the design tasks for the four views

C. Global Analysis (1/2)

1. Review Global Analyses
   - From the module and execution views.
   - Review factors and strategies for additional requirements or constraints on the design of this view

2. Analyze Factors
   - Development platform
   - Development environment:
     - List capabilities of tools
     - Pay attention to CM
     - Think feasibility of cross-platform development
   - Development process
     - Identify process and testing requirements

C. Global Analysis (2/2)

2. Analyze Factors (Cont.)
   - Development schedule
     - Staged release, internal release, multiple concurrent releases by developers and testers
     - Whether to develop multiple versions of components
   - Target platform and environment
     - Analyze the support for installing the system on the target platform

3. Develop Strategies

   - To address the factors

D. Central Design Tasks

1. Design source components
   - Identify source components and map elements and dependencies from the module view to the source components and dependencies
   - Organize them using storage structures (e.g. directories, files)

2. Design Intermediate components
   - Identify intermediate components (e.g. binary components, libraries) and their dependencies on source components and each other
   - Organize them using storage structures (e.g. directories, files)

3. Design deployment components
   - Identify and map runtime entities and dependencies in the execution view to deployment components and their dependencies
   - Organize the deployment components

Components in Code View – Source (1/2)

- Source components
  - Language-specific interfaces
    - *.h files (C), *.H files (C++), Package specification (Ada)
  - Language-specific modules
    - *.c files (C), *.CPP files (C++), Package implementation (Ada)
  - Components that generate them

- Relations between source components
  - "import" relation:

    E.g. In C and C++, a source file and a header file
  - "generate" relation:

    E.g. Preprocessing generates expanded sources component

Components in Code View – Source (2/2)

- How to organize source components
  - Directory hierarchy, database, shared repository
  - For simple systems:
    - Similarity of functionality (e.g. GUI components)
    - Developer responsibility
  - For large systems, should also consider:
    - Ease of accessing
    - Related components such as test drivers and test data
    - Release process
    - Edit-compile-link cycle time

Components in Code View - Intermediate

- Intermediate component
  - Specific to implementation language and development environment
  - E.g. binary components, static libraries
    - C++: .CPP file => .obj file
    - Ada: binary files => program libraries

- How to organize intermediate components
  - Increase sharing to reduce edit-compile-link time

    E.g. static library can be used by components in the higher layer

Components in Code View - Deployment

- Deployment component
  - Deployed at runtime to instantiate the runtime entities

  E.g. executables, dynamic libraries, configuration description (describes processes and/or resources, such as storage for data and shared memory)
  - Link dependency is important:

$$\begin{tikzcd}
\text{dynamic libraries} \arrow[dr, dashed] \arrow[dd, dashed, leftrightarrow] & \\
& \{\text{binary components, static libraries}\} \\
\text{Executables} \arrow[ur, dashed] &
\end{tikzcd}$$

- How to organize deployment components
  - For a simple system, a few packages containing

    executable files + configuration description
  - For a large system, may need
    - Separate file system for data and shared memory areas
    - Organize separately each executable and its associates components such as required resources and test data

E. Final Design Tasks

1. Define build procedure
   - Design the procedure for building and installing the intermediate and deployment components
     - Based on components, dependencies, the release process

2. Define Configuration Management process
   - Determine design decisions related to management of versions and releases of components

[1] Global Analysis

A. Review Global Analysis

- Identified strategies relevant to code view
  - **Issue6**: Addition and Removal of Features
    - *S6A: Separate components and modules along dimensions of concern*
    - *S6B: Encapsulate features into separate components*

    => Need to design the components in the code view so that they can be linked to different executables without modifying them
  - **Issue9**: Realtime Acquisition Performance
    - *S9C: Use flexible allocation of modules to processes*

    => Modules assigned to processes may change as the system is tuned for performance
  - **Issue1**: Aggressive Schedule
    - *S1C: Make it easy to add or remove features*

    => Many features make use of the same modules or the same interfaces

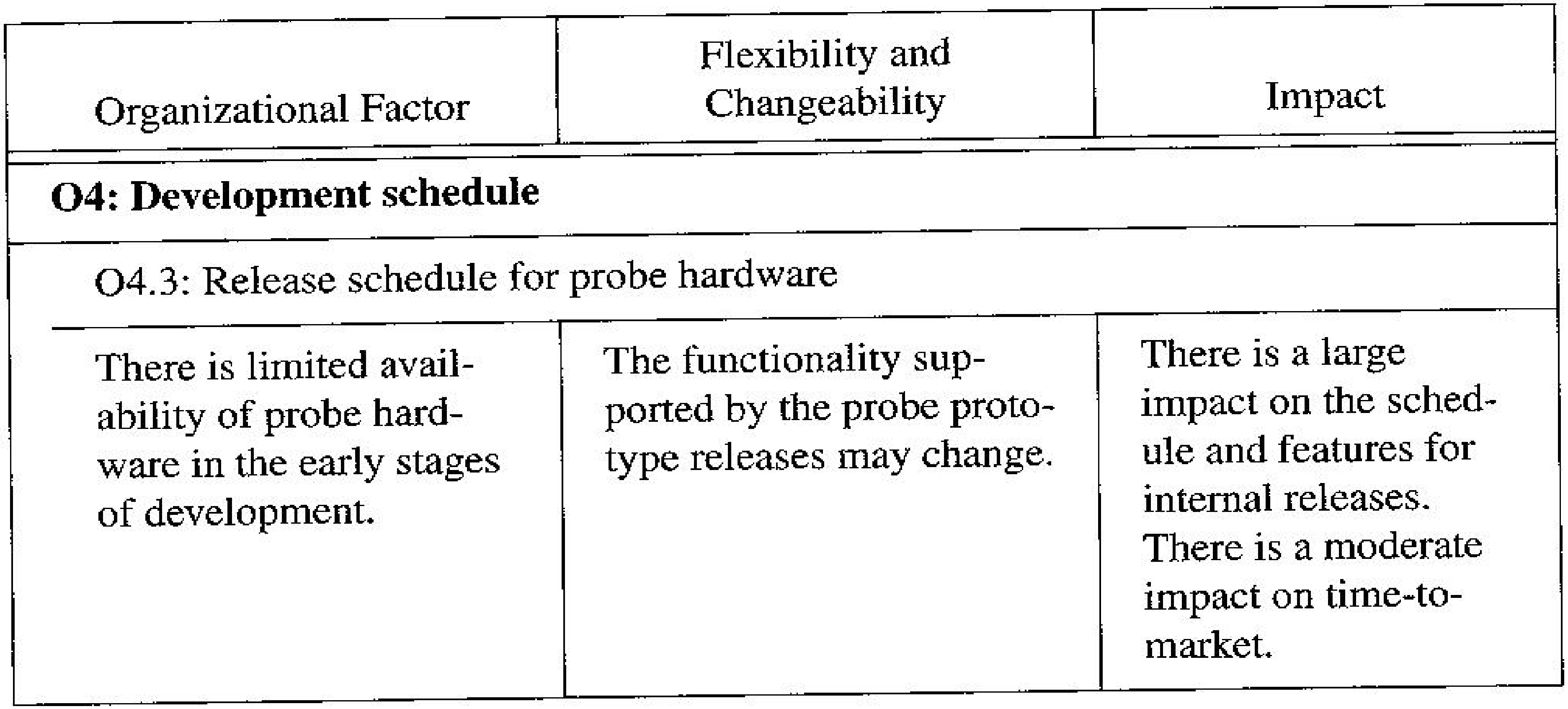
    => New strategy: Separately organize the deployment components and the source components that are linked to them

B. Analyze Factors

- New factors added: Tables 7.1 and 7.2

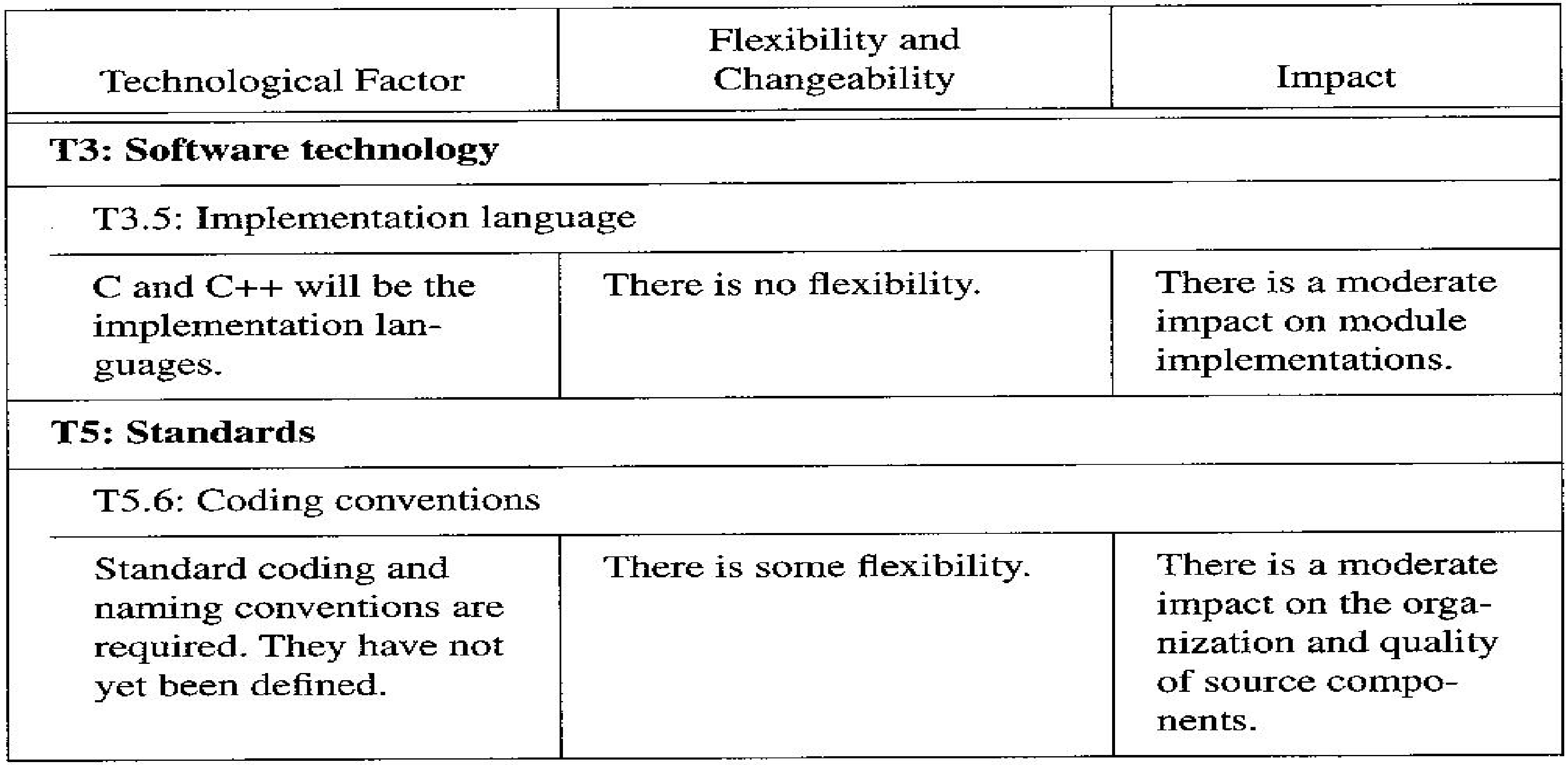
Organizational Factor	Flexibility and Changeability	Impact
O3: Process and development environment		
O3.1: Development platform		
Use UNIX and real-time UNIX.	Components that use real-time features can only be developed on the real-time UNIX platform.	There is a moderate impact on the productivity for development of real-time features.
O3.2: Configuration management		
Facilitates a variety of roles and views of the code architecture.	There is a lot of flexibility.	Facilitates concurrent development and flexible releases of software components.
O3.3: Release process		
Three internal releases are required. Concurrent development, integration, and testing are also envisioned.	There is no flexibility.	The configuration management process and the code architecture view must be designed to support concurrent development and testing.
O3.4: Testing process		
Three levels of testing are required: module, integration, and system.	There is no flexibility.	There is a moderate impact on the configuration management process.

Table 7.1. Organizational Factors Added During Code Architecture View Design



Organizational Factor	Flexibility and Changeability	Impact
O4: Development schedule		
O4.3: Release schedule for probe hardware		
There is limited availability of probe hardware in the early stages of development.	The functionality supported by the probe prototype releases may change.	There is a large impact on the schedule and features for internal releases. There is a moderate impact on time-to-market.

Table 7.1. Organizational Factors Added During Code Architecture View Design *(continued)*



Technological Factor	Flexibility and Changeability	Impact
T3: Software technology		
T3.5: Implementation language		
C and C++ will be the implementation languages.	There is no flexibility.	There is a moderate impact on module implementations.
T5: Standards		
T5.6: Coding conventions		
Standard coding and naming conventions are required. They have not yet been defined.	There is some flexibility.	There is a moderate impact on the organization and quality of source components.

Table 7.2. Technological Factors Added During Code Architecture View Design

C. Develop Strategies

- New Issues:
  - **Issue**: **Architectural Integrity**
    - Architectural decisions and principles must be preserved and enforced in the code architecture. Violation should be detected.
  - **Issue**: **Concurrent Development Tasks**
    - Developers may be working concurrently on different component versions for multiple internal releases
  - **Issue**: **Limited Availability of Probe Prototypes**
    - It will adversely affect the developers' ability to test many components
  - **Issue**: **Multiple Development and Target Platforms**
    - The use of multiple development and target platforms must be considered and facilitated.

- Strategies for the new Issues are developed

12. Architectural Integrity

Architectural design decisions related to layering, encapsulation, abstraction, or separation of concerns can be violated in code. This eventually leads to degradation of the architecture. It is the architect's responsibility to ensure architectural integrity in code.

Influencing Factors

O3.2: The configuration management tool can be used to define restricted views of the code architecture view for predefined roles. This approach reduces degradation of the architecture.

T3.5: It is possible, though undesirable, to circumvent the module encapsulation and abstraction decisions through techniques such as inclusion of private header files and use of untyped conversions. Similarly, it is easy to create disallowed module dependencies. The decomposition of header files also has a significant impact on the time to execute the compile-link-test cycle.

T5.6: Standard coding conventions and techniques will be used. There is some flexibility in selecting, combining, or adding conventions.

Solution

Making architectural decisions explicit in the code can go a long way toward detecting violations. There is tension between the goal to make editing more convenient by reducing the number of header files and between reducing compile-time dependencies by separating header files into public and private ones.

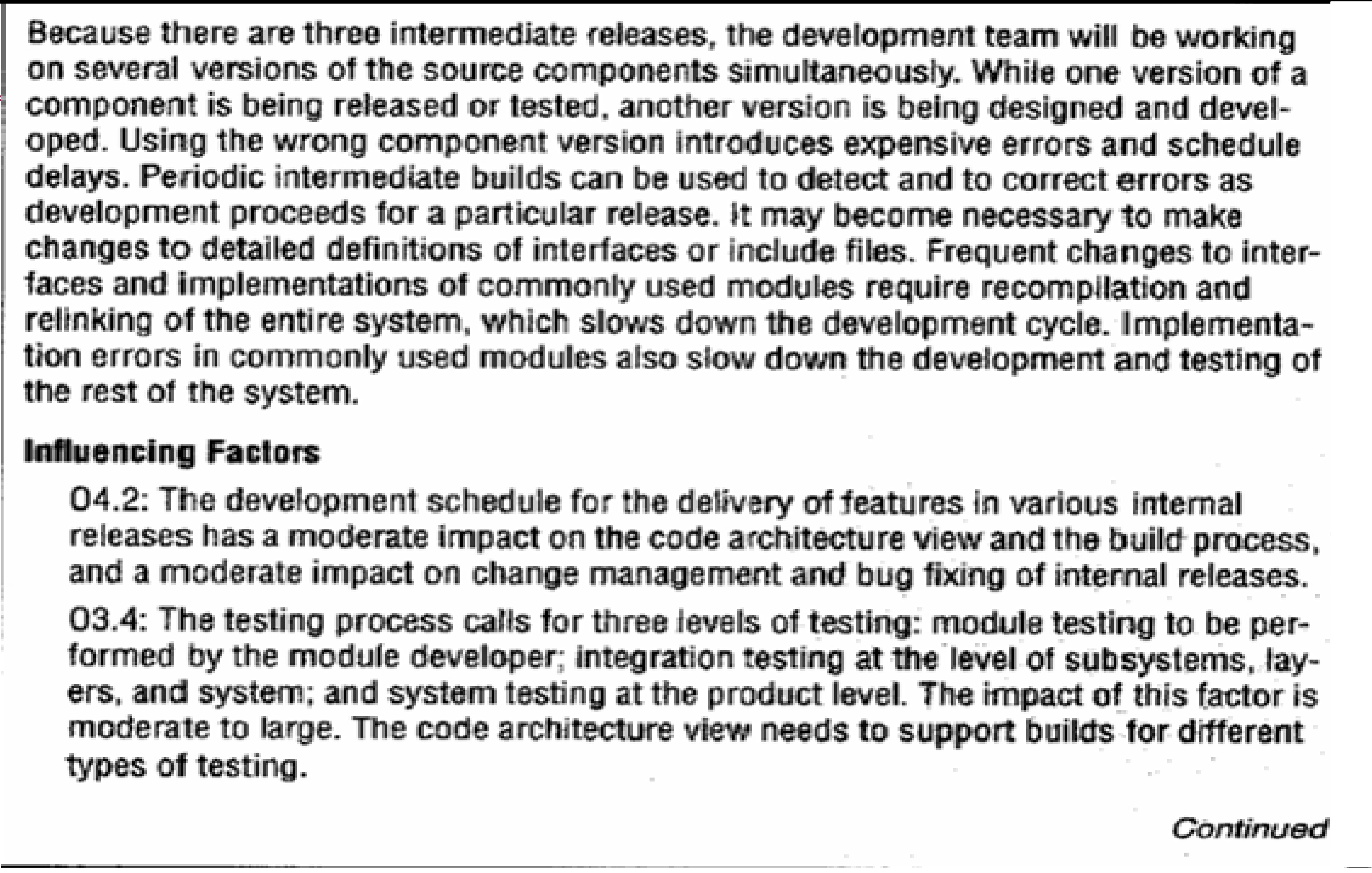
Strategy: ***Preserve module view hierarchies.*** S12A

Reflect module and layer hierarchies explicitly in the code architecture view by hierarchically mapping each layer and module into a separate package. This preserves previous decisions related to component separation, improves traceability between module and code architecture views, and thus makes violation of architectural decisions easier to detect. This also makes it easier to assign responsibility for particular modules and layers to a single person. Note that this strategy does not mean that the code architecture view should be made identical to the module architecture view.

Strategy: ***Separate organization of public interface components.*** S12B

Organize public interface components in separate directories. In particular, create a public interface package for each layer and subsystem. This makes it easier to restrict the use of private header files through build scripts and search paths, thus enforcing certain architectural decisions such as encapsulation and abstraction, and detecting their violation.

13. Concurrent Development Tasks



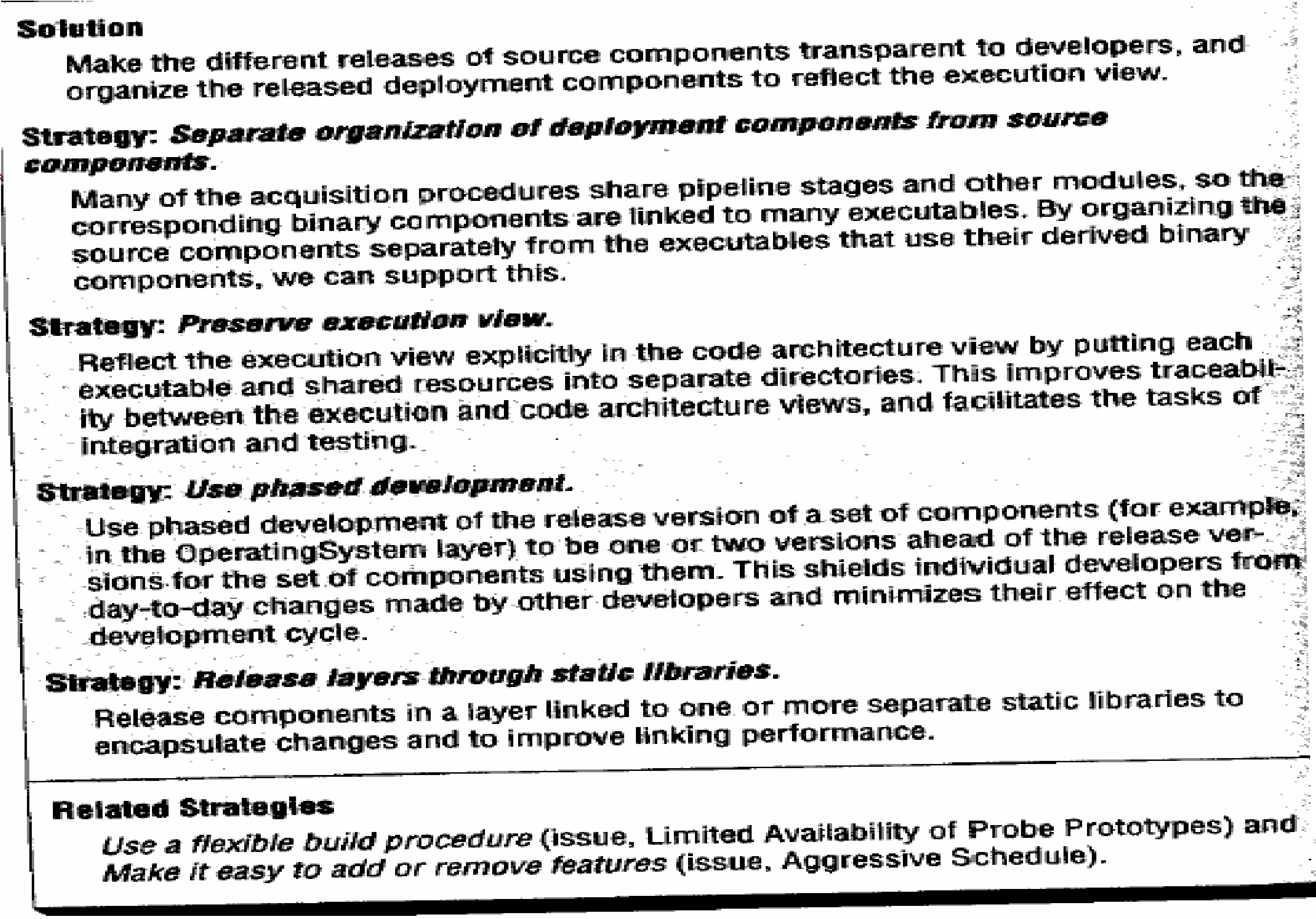
Because there are three intermediate releases, the development team will be working on several versions of the source components simultaneously. While one version of a component is being released or tested, another version is being designed and developed. Using the wrong component version introduces expensive errors and schedule delays. Periodic intermediate builds can be used to detect and to correct errors as development proceeds for a particular release. It may become necessary to make changes to detailed definitions of interfaces or include files. Frequent changes to interfaces and implementations of commonly used modules require recompilation and relinking of the entire system, which slows down the development cycle. Implementation errors in commonly used modules also slow down the development and testing of the rest of the system.

Influencing Factors

O4.2: The development schedule for the delivery of features in various internal releases has a moderate impact on the code architecture view and the build process, and a moderate impact on change management and bug fixing of internal releases.

O3.4: The testing process calls for three levels of testing: module testing to be performed by the module developer; integration testing at the level of subsystems, layers, and system; and system testing at the product level. The impact of this factor is moderate to large. The code architecture view needs to support builds for different types of testing.

Continued



Solution

Make the different releases of source components transparent to developers, and organize the released deployment components to reflect the execution view.

Strategy: *Separate organization of deployment components from source components.*

Many of the acquisition procedures share pipeline stages and other modules, so the corresponding binary components are linked to many executables. By organizing the source components separately from the executables that use their derived binary components, we can support this.

Strategy: *Preserve execution view.*

Reflect the execution view explicitly in the code architecture view by putting each executable and shared resources into separate directories. This improves traceability between the execution and code architecture views, and facilitates the tasks of integration and testing.

Strategy: *Use phased development.*

Use phased development of the release version of a set of components (for example, in the OperatingSystem layer) to be one or two versions ahead of the release versions for the set of components using them. This shields individual developers from day-to-day changes made by other developers and minimizes their effect on the development cycle.

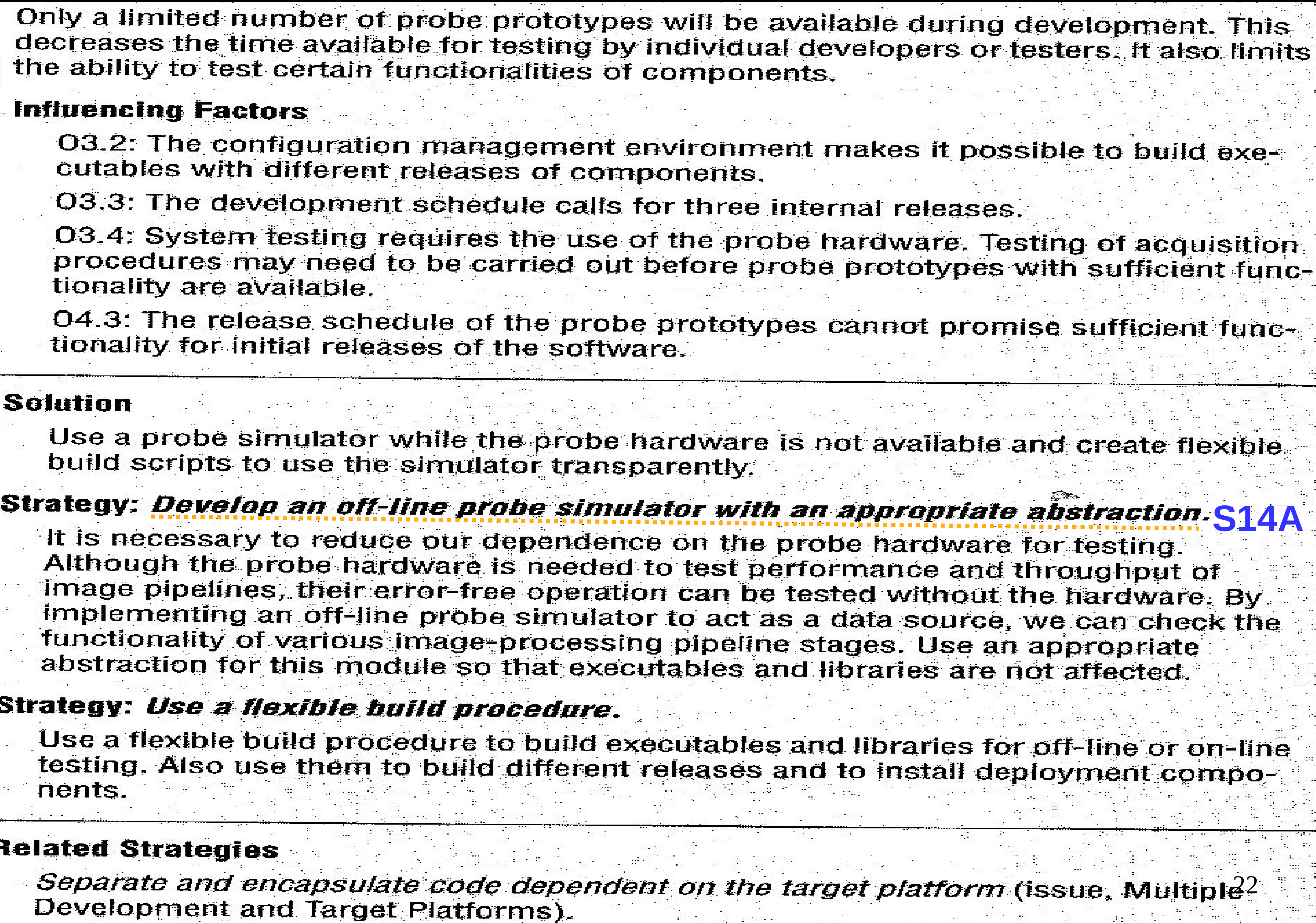
Strategy: *Release layers through static libraries.*

Release components in a layer linked to one or more separate static libraries to encapsulate changes and to improve linking performance.

Related Strategies

Use a flexible build procedure (issue, Limited Availability of Probe Prototypes) and *Make it easy to add or remove features* (issue, Aggressive Schedule).

14. Limited Availability of Probe Prototypes



Only a limited number of probe prototypes will be available during development. This decreases the time available for testing by individual developers or testers. It also limits the ability to test certain functionalities of components.

Influencing Factors

O3.2: The configuration management environment makes it possible to build executables with different releases of components.

O3.3: The development schedule calls for three internal releases.

O3.4: System testing requires the use of the probe hardware. Testing of acquisition procedures may need to be carried out before probe prototypes with sufficient functionality are available.

O4.3: The release schedule of the probe prototypes cannot promise sufficient functionality for initial releases of the software.

Solution

Use a probe simulator while the probe hardware is not available and create flexible build scripts to use the simulator transparently.

Strategy: ***Develop an off-line probe simulator with an appropriate abstraction.*** S14A

It is necessary to reduce our dependence on the probe hardware for testing. Although the probe hardware is needed to test performance and throughput of image pipelines, their error-free operation can be tested without the hardware. By implementing an off-line probe simulator to act as a data source, we can check the functionality of various image-processing pipeline stages. Use an appropriate abstraction for this module so that executables and libraries are not affected.

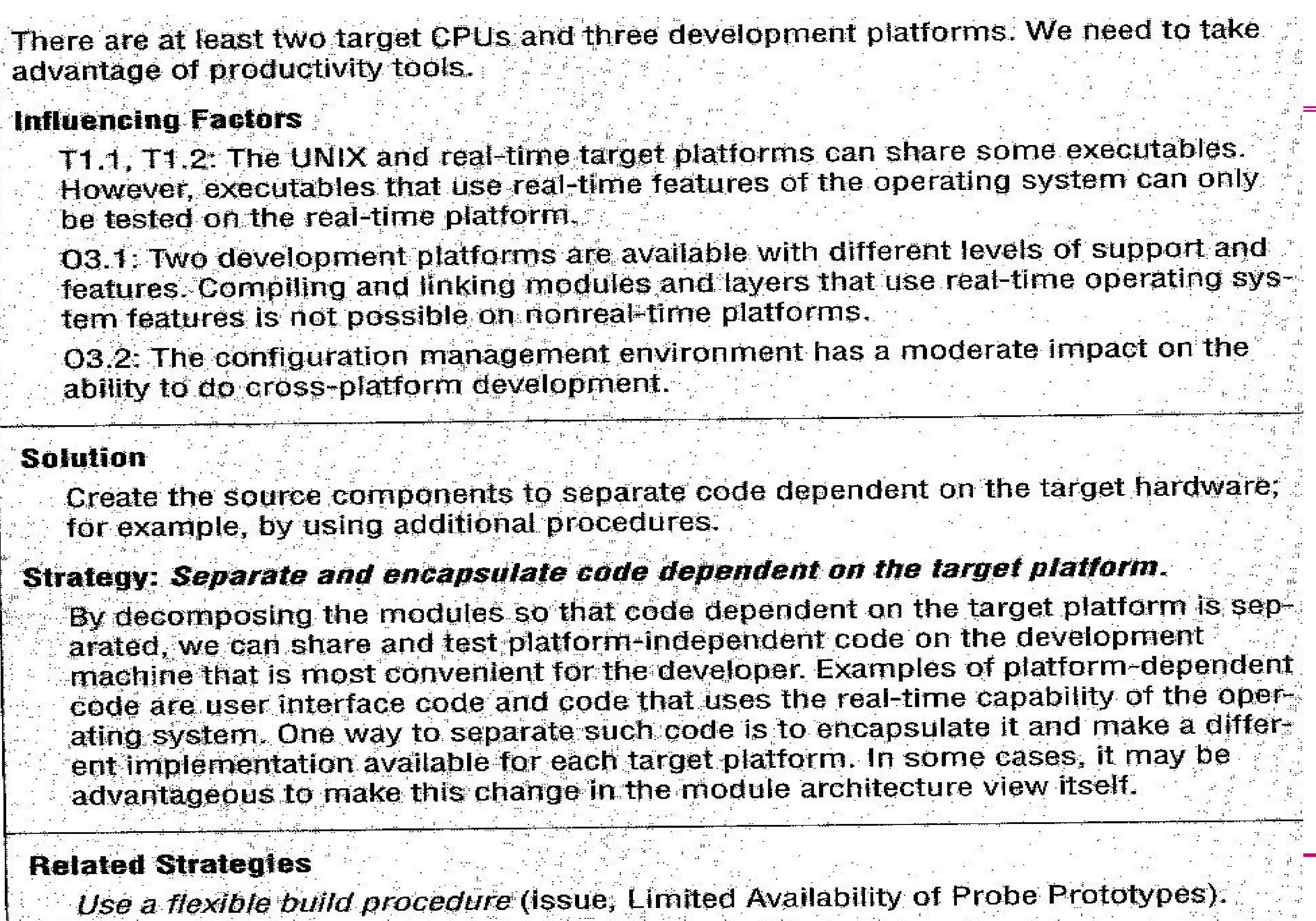
Strategy: ***Use a flexible build procedure.***

Use a flexible build procedure to build executables and libraries for off-line or on-line testing. Also use them to build different releases and to install deployment components.

Related Strategies

Separate and encapsulate code dependent on the target platform (issue, Multiple Development and Target Platforms).

15. Multiple Development and Target Platforms



There are at least two target CPUs and three development platforms. We need to take advantage of productivity tools.

Influencing Factors

T1.1, T1.2: The UNIX and real-time target platforms can share some executables. However, executables that use real-time features of the operating system can only be tested on the real-time platform.

O3.1: Two development platforms are available with different levels of support and features. Compiling and linking modules and layers that use real-time operating system features is not possible on nonreal-time platforms.

O3.2: The configuration management environment has a moderate impact on the ability to do cross-platform development.

Solution

Create the source components to separate code dependent on the target hardware; for example, by using additional procedures.

Strategy: ***Separate and encapsulate code dependent on the target platform.***

By decomposing the modules so that code dependent on the target platform is separated, we can share and test platform-independent code on the development machine that is most convenient for the developer. Examples of platform-dependent code are user interface code and code that uses the real-time capability of the operating system. One way to separate such code is to encapsulate it and make a different implementation available for each target platform. In some cases, it may be advantageous to make this change in the module architecture view itself.

Related Strategies

Use a flexible build procedure (issue, Limited Availability of Probe Prototypes).

[2] Central Design Tasks:

Source Components, Intermediate Components, and Deployment Components

D. Central Design Tasks

1. Design source components
2. Design Intermediate components
3. Design deployment components

Source Components

- **1st step:** Start with layers defined in the module view
  - ***S12A**: Preserve module view hierarchies*

    (<= Architectural Integrity Issue)

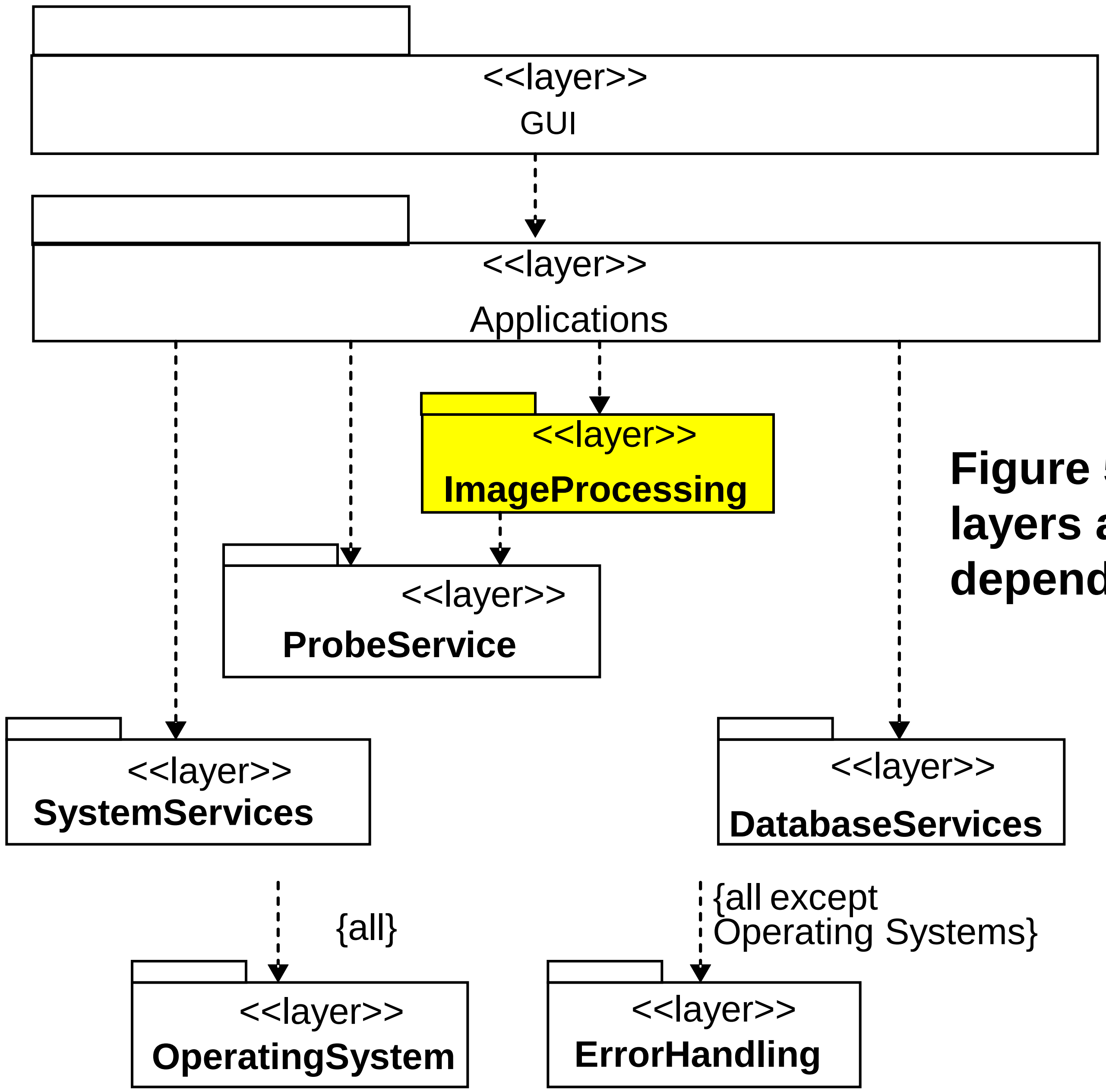


Figure 5.11. Final version of layers and their use-dependencies

Source Components

- **1st Step:** Start with layers defined in the module view (Cont.)
  - ***S12B****: Separate organization of public interface components*

    (<= Architectural Integrity Issue)

  => Create 'include' code group

Figure 7.3 Code groups for organizing source components in IS2000's layers

Created to copy and organize the **public interface components** that are commonly used by all of the layers

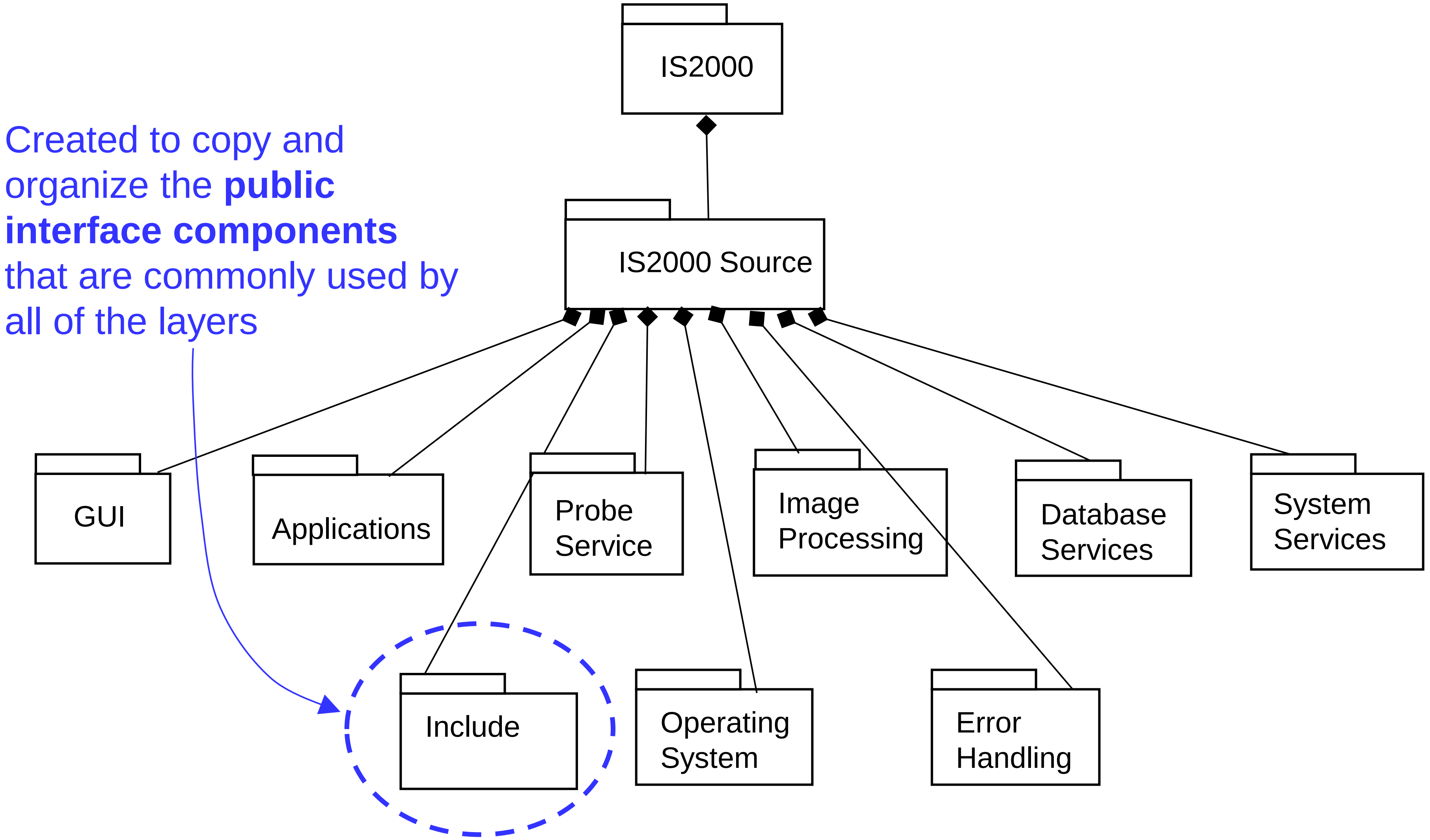
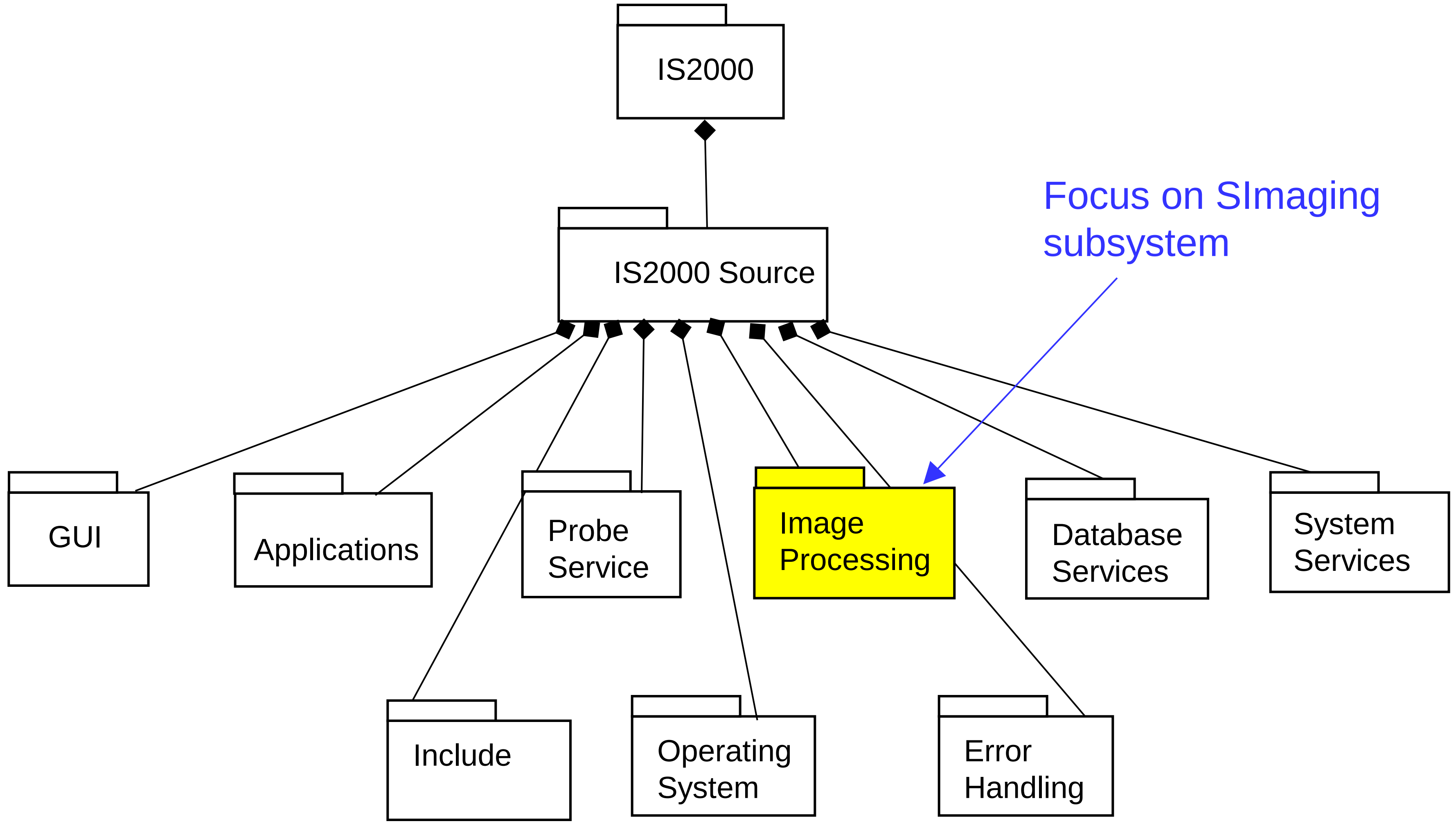


Figure 7.3 Code groups for organizing source components in IS2000's layers



Source Components

- **2nd Step:** Map **(1) modules** and (2) interfaces in the subsystem (in the module view) to source components
  - ***S12B**: Separate organization of public interface components*

○ notation in Figures 7.4 and 7.5

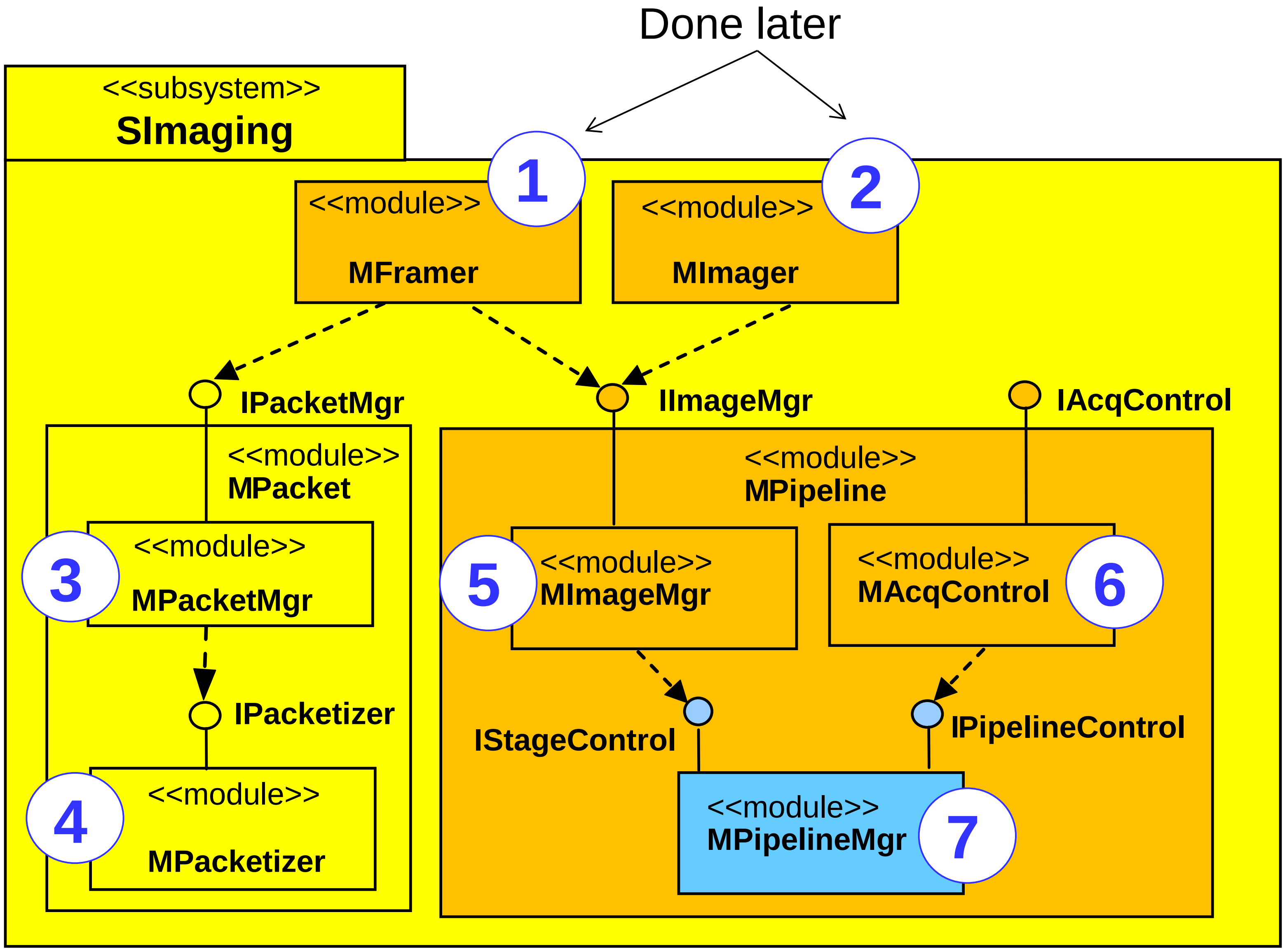


Figure 7.4. Imaging subsystem use-dependencies (from the module view) (Cf. Figure 5.8)

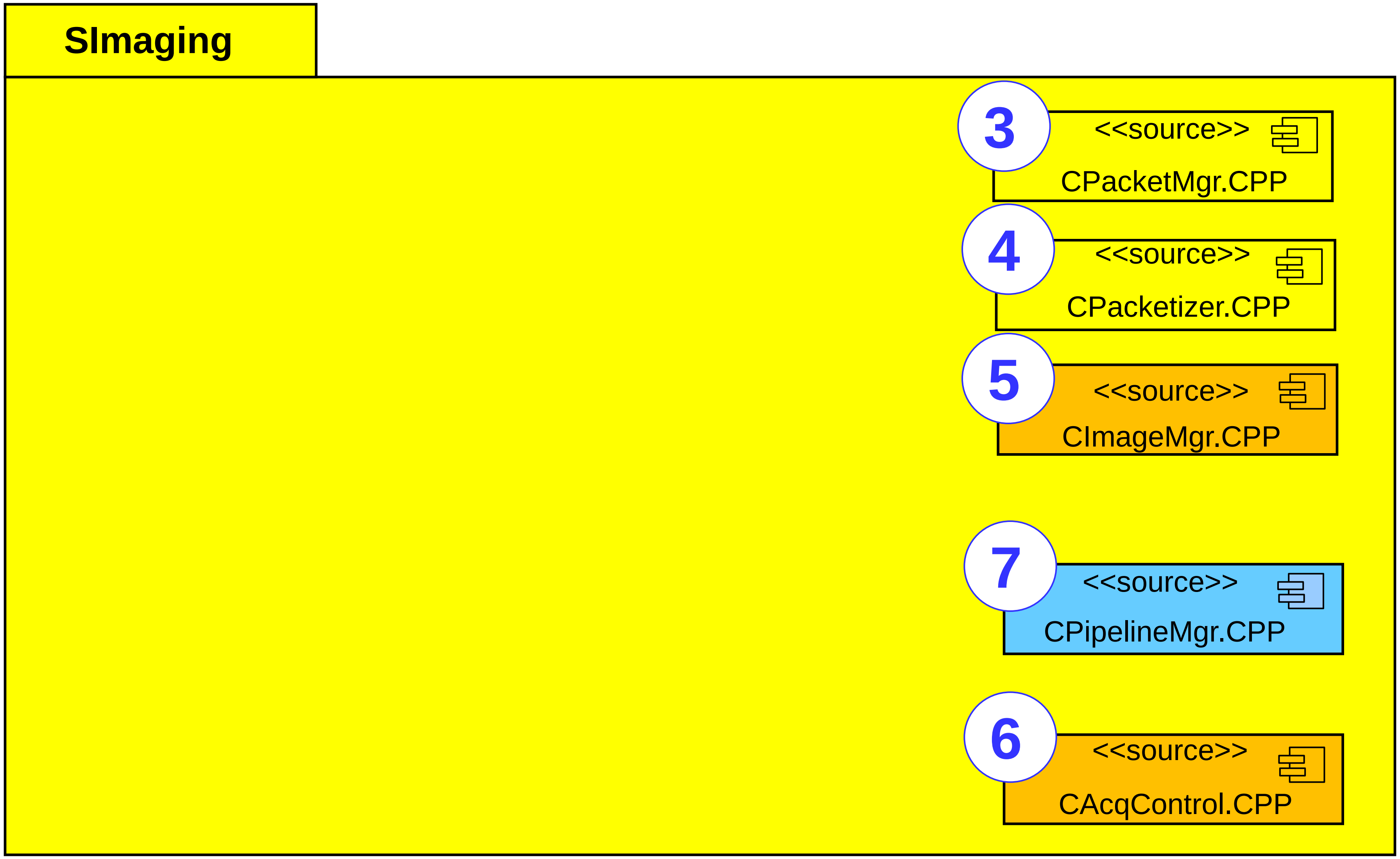


Figure 7.5. Source components and dependencies for the image-processing system

Source Components

- **2nd Step**: Map (1) modules and **(2) interfaces** in the subsystem (in the module view) to source components
  - ***S12B**: Separate organization of public interface components*

    □ notation in Figures 7.4 and 7.5

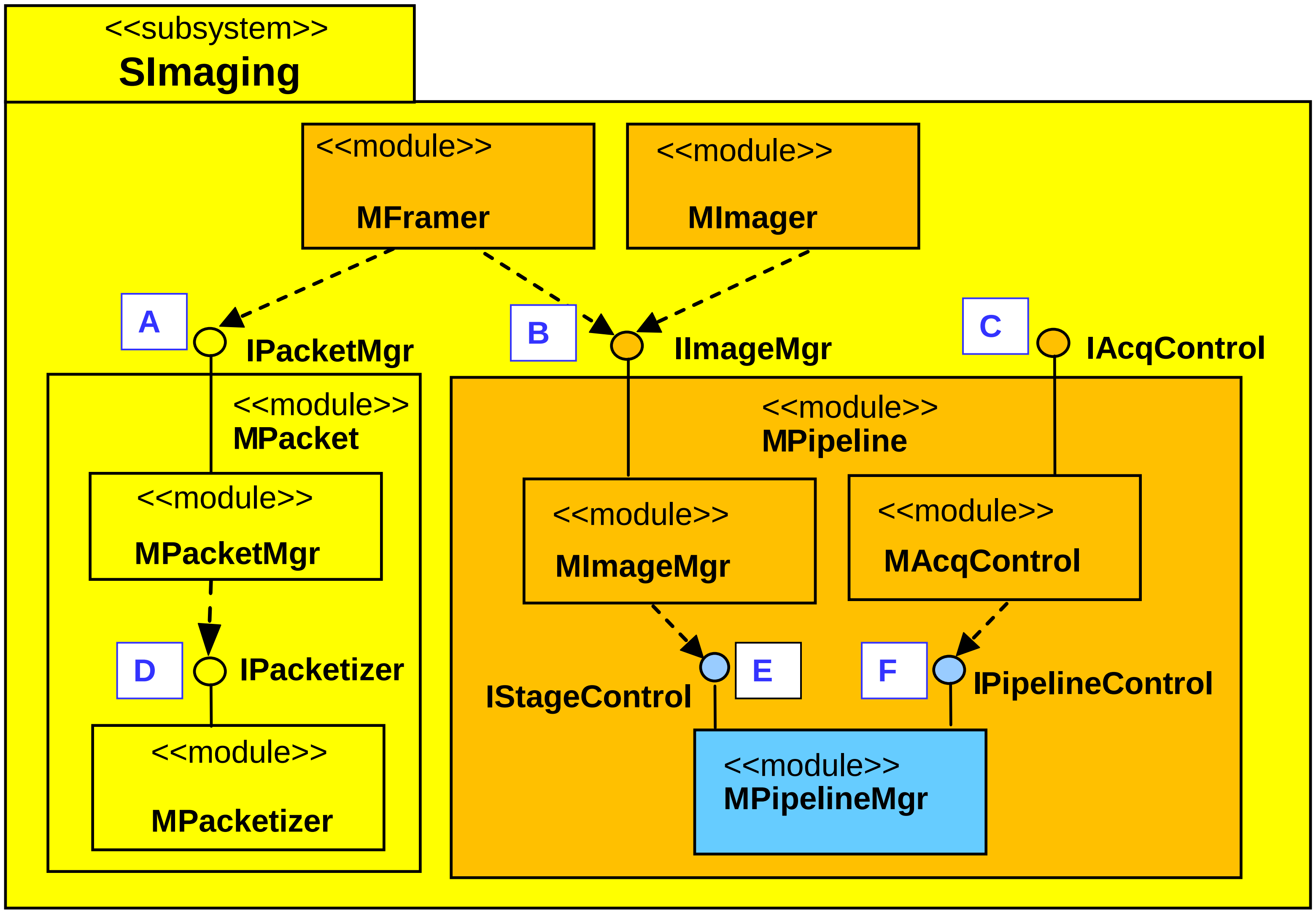


Figure 7.4. Imaging subsystem use-dependencies (from the module view) **(Cf. Figure 5.8)**

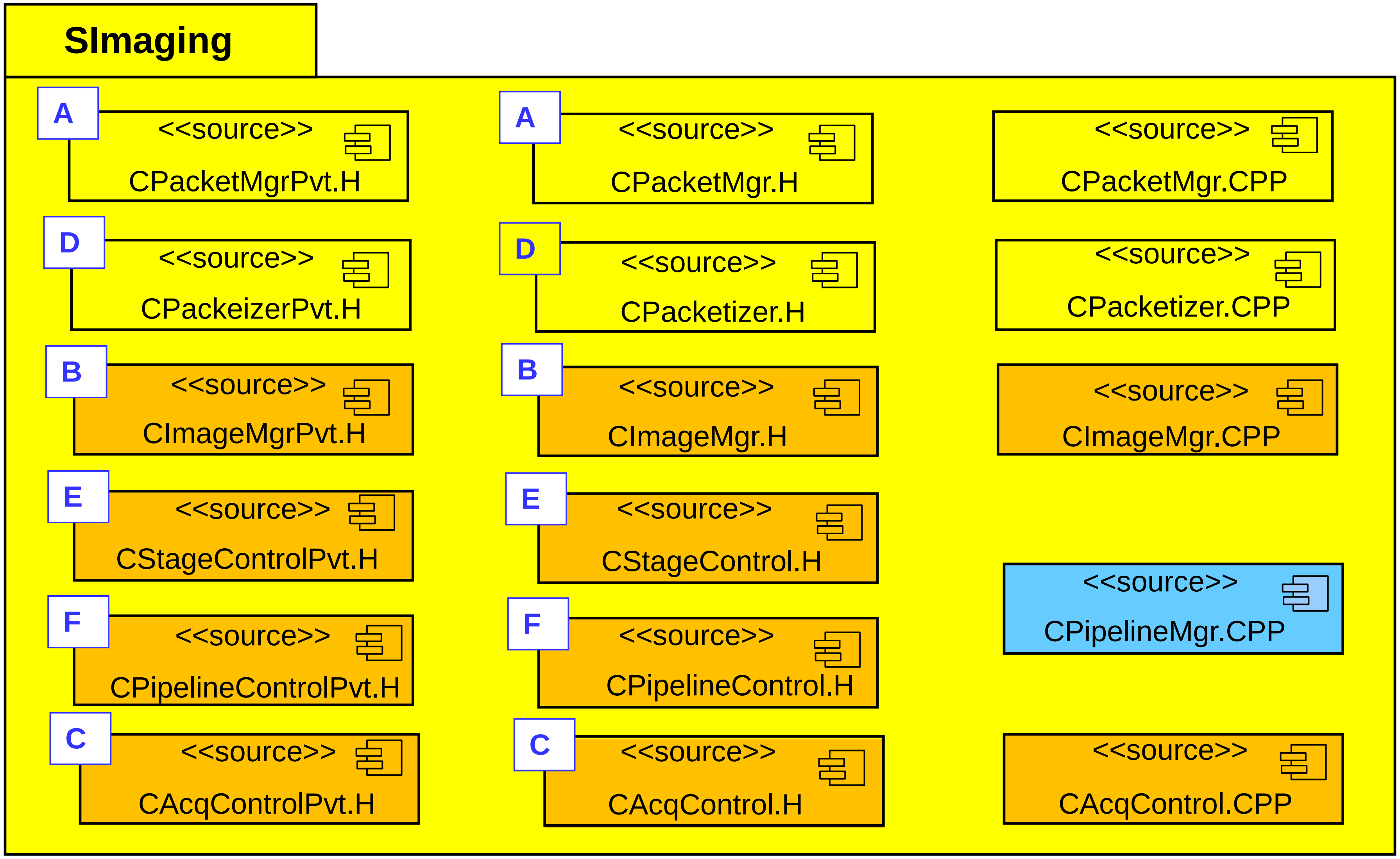


Figure 7.5. Source components and dependencies for the image-processing system

Source Components

- **3rd Step:** Map the dependencies among the modules

◇ x : required interface

◯ y : provided interface

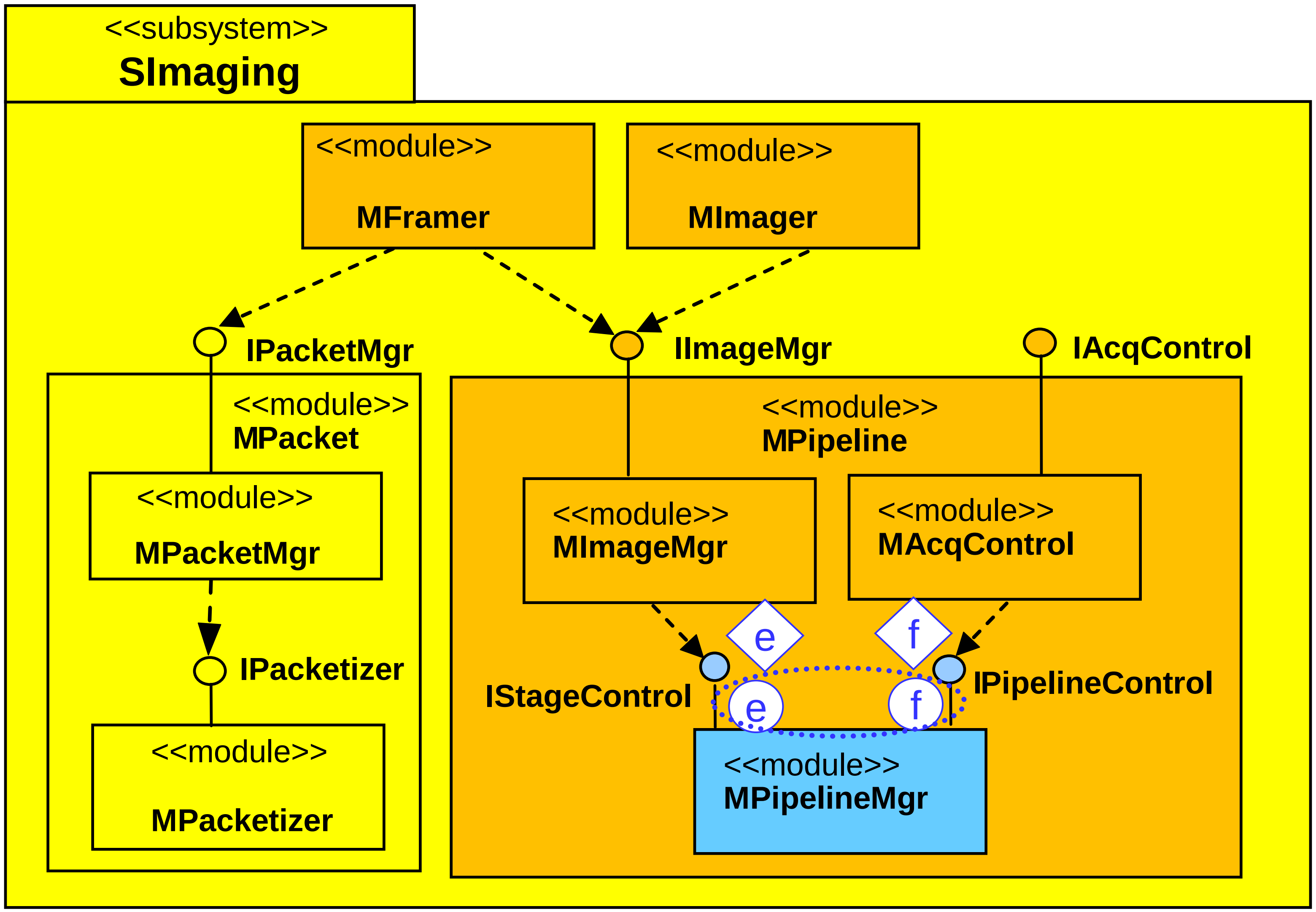


Figure 7.4. Imaging subsystem use-dependencies (from the module view) (Cf. Figure 5.8)

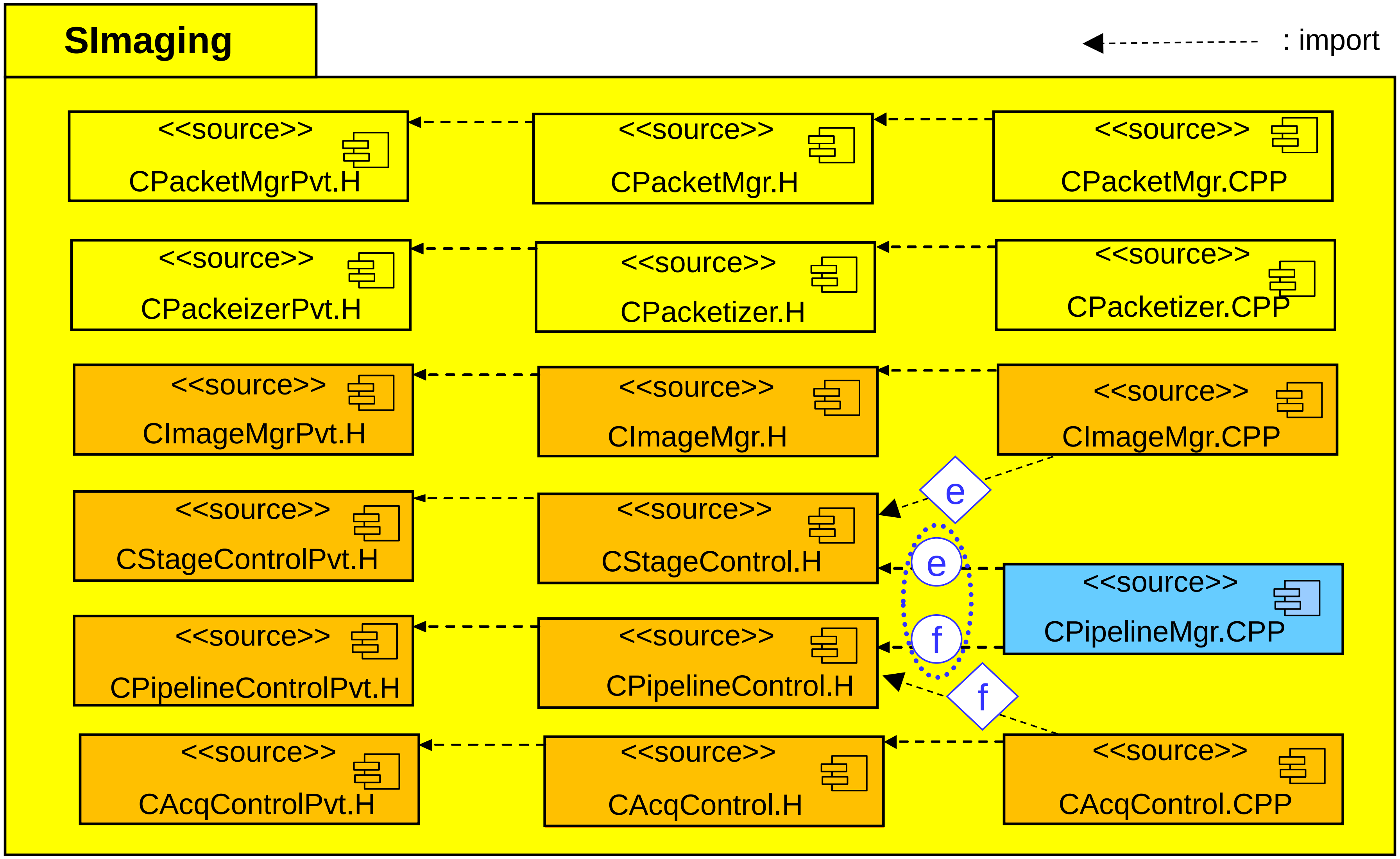


Figure 7.5. Source components and dependencies for the image-processing system

Source Components

- **4th Step:** Look at MFramer and MImager in Figure 7.4
  - There are several instances (2D, 3D, ...) of these modules, depending on various acquisition procedures
  - See Figure 7.6

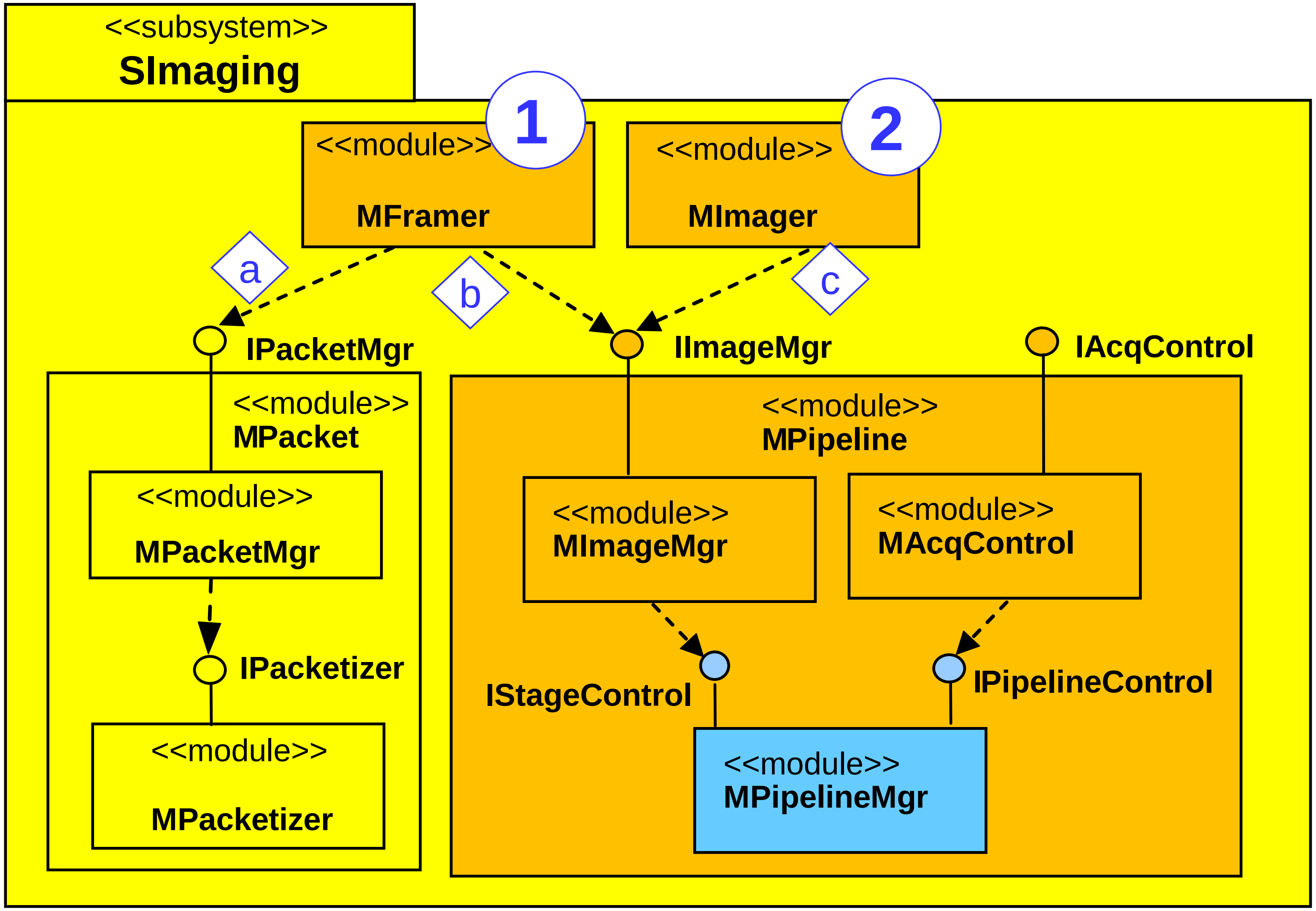
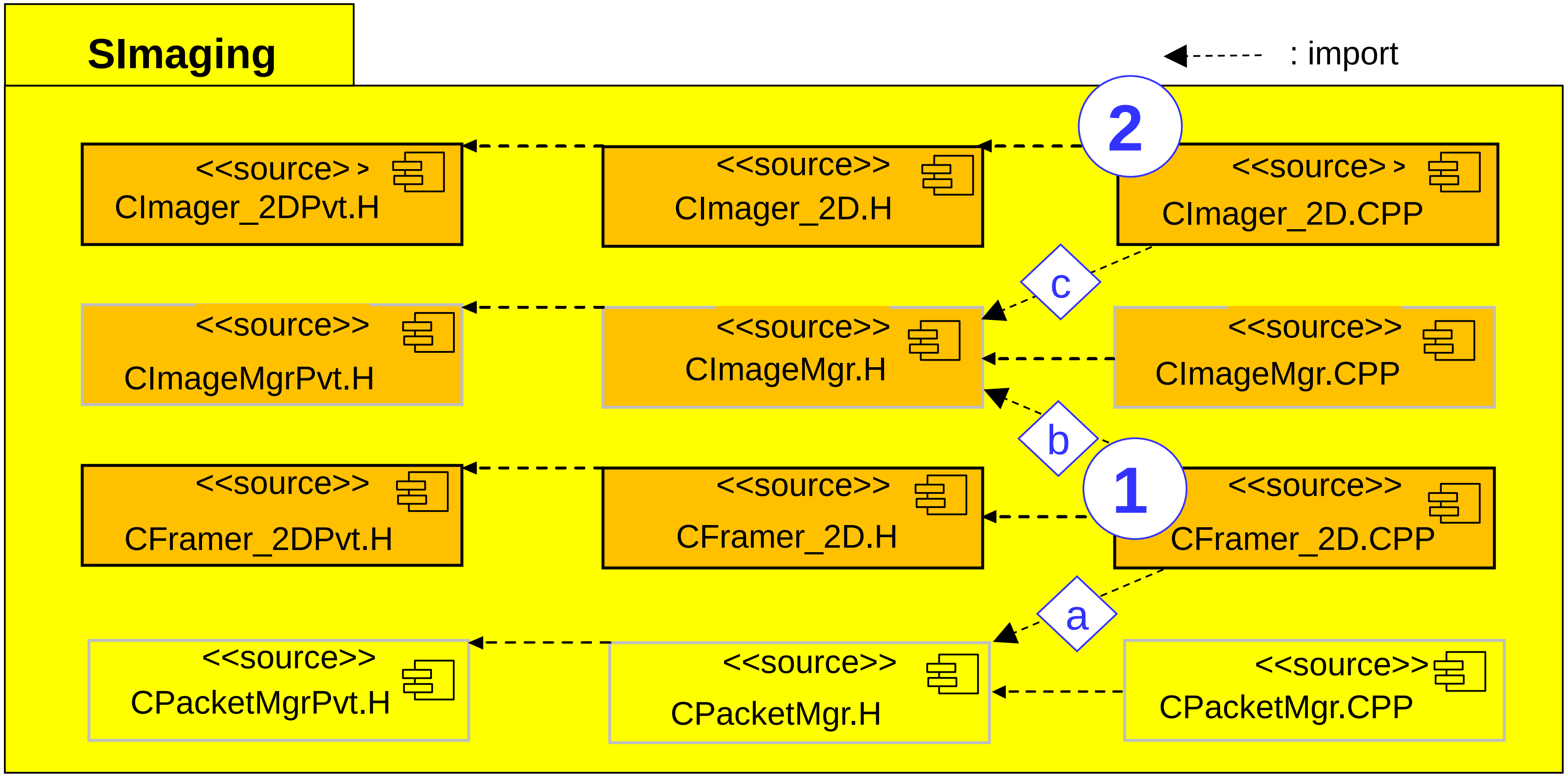


Figure 7.4. Imaging subsystem use-dependencies (from the module view) (Cf. Figure 5.8)

Figure 7.6 Mapping MFramer and MImager to source components

- Two specific **instances of MFramer and MImager** – named MFramer_2D and MImager_2D – can be mapped to corresponding source components as follows:



SImaging
◀------ : import
<<source>> CImager_2DPvt.H
<<source>> CImager_2D.H
<<source>> CImager_2D.CPP
2
c
<<source>> CImageMgrPvt.H
<<source>> CImageMgr.H
<<source>> CImageMgr.CPP
b
1
<<source>> CFramer_2DPvt.H
<<source>> CFramer_2D.H
<<source>> CFramer_2D.CPP
a
<<source>> CPacketMgrPvt.H
<<source>> CPacketMgr.H
<<source>> CPacketMgr.CPP

Source Components

- **5th Step:** Source components corresponding to each module may be further decomposed
  - Organize them in separate packages
  - Use the strategy: ***S12B****:Preserve module view hierarchies*

Figure 7.3 Code groups for organizing source components in IS2000's layers (Shown Earlier)

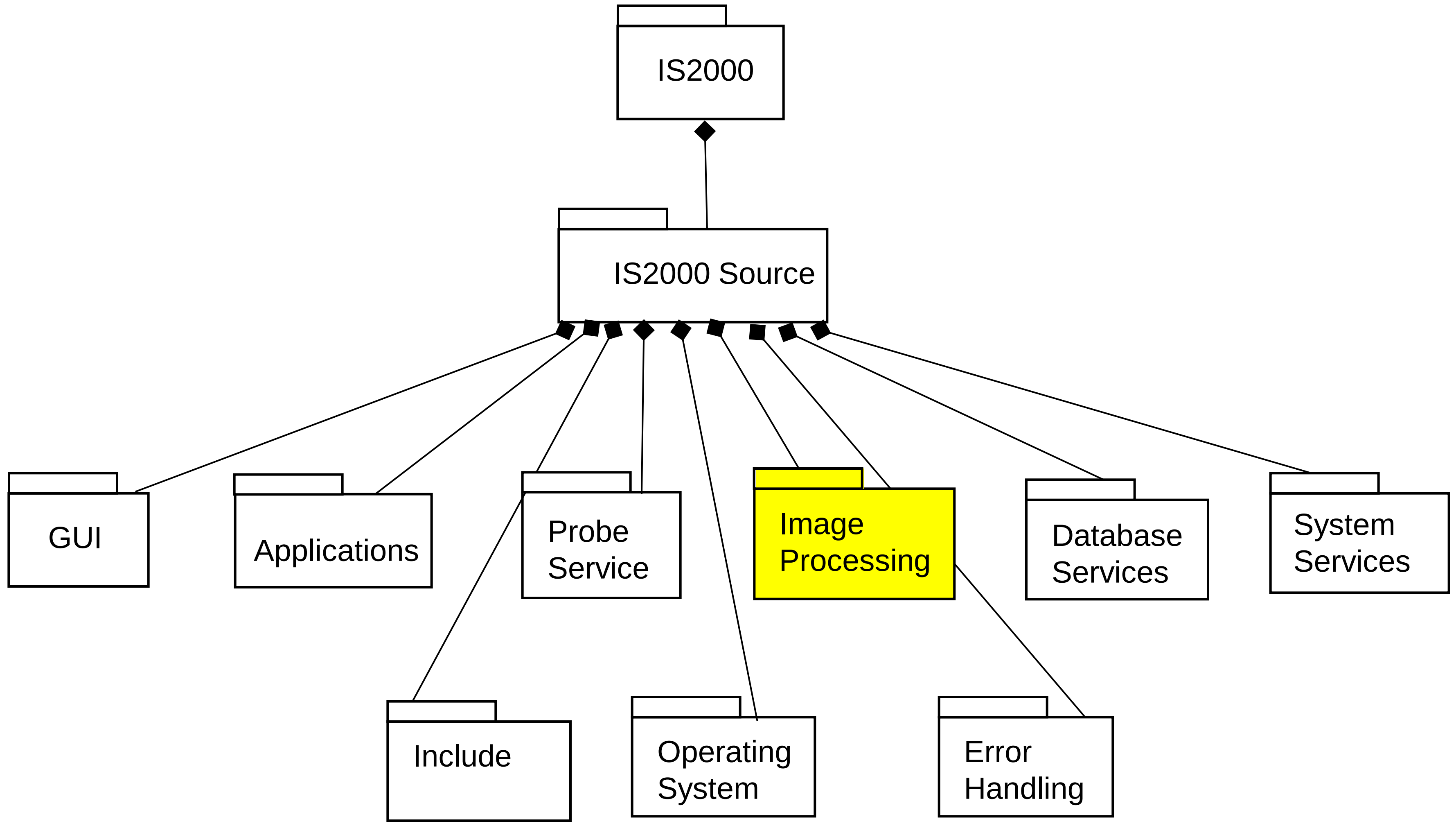
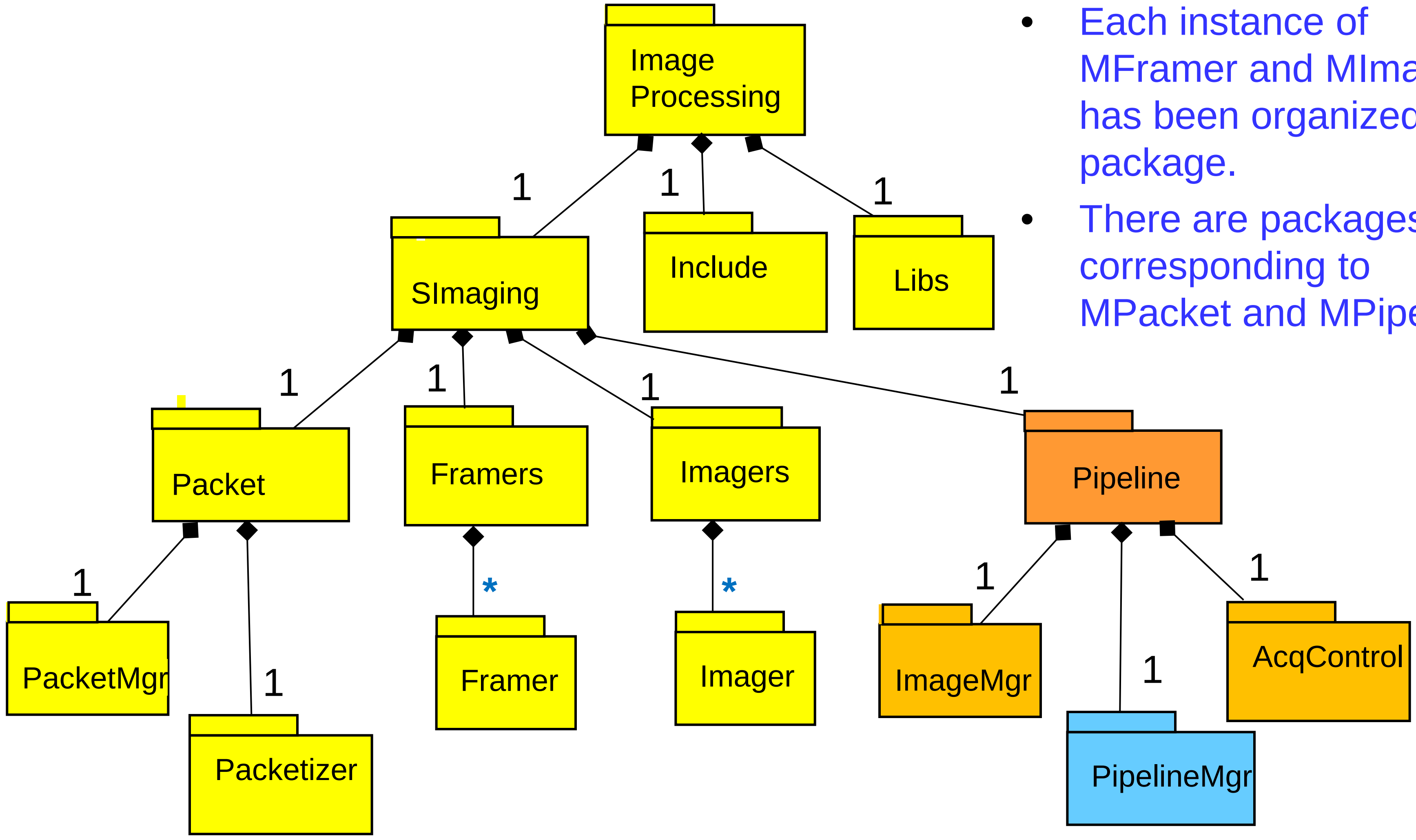


Figure 7.7 Organizing source components for the SImaging subsystem

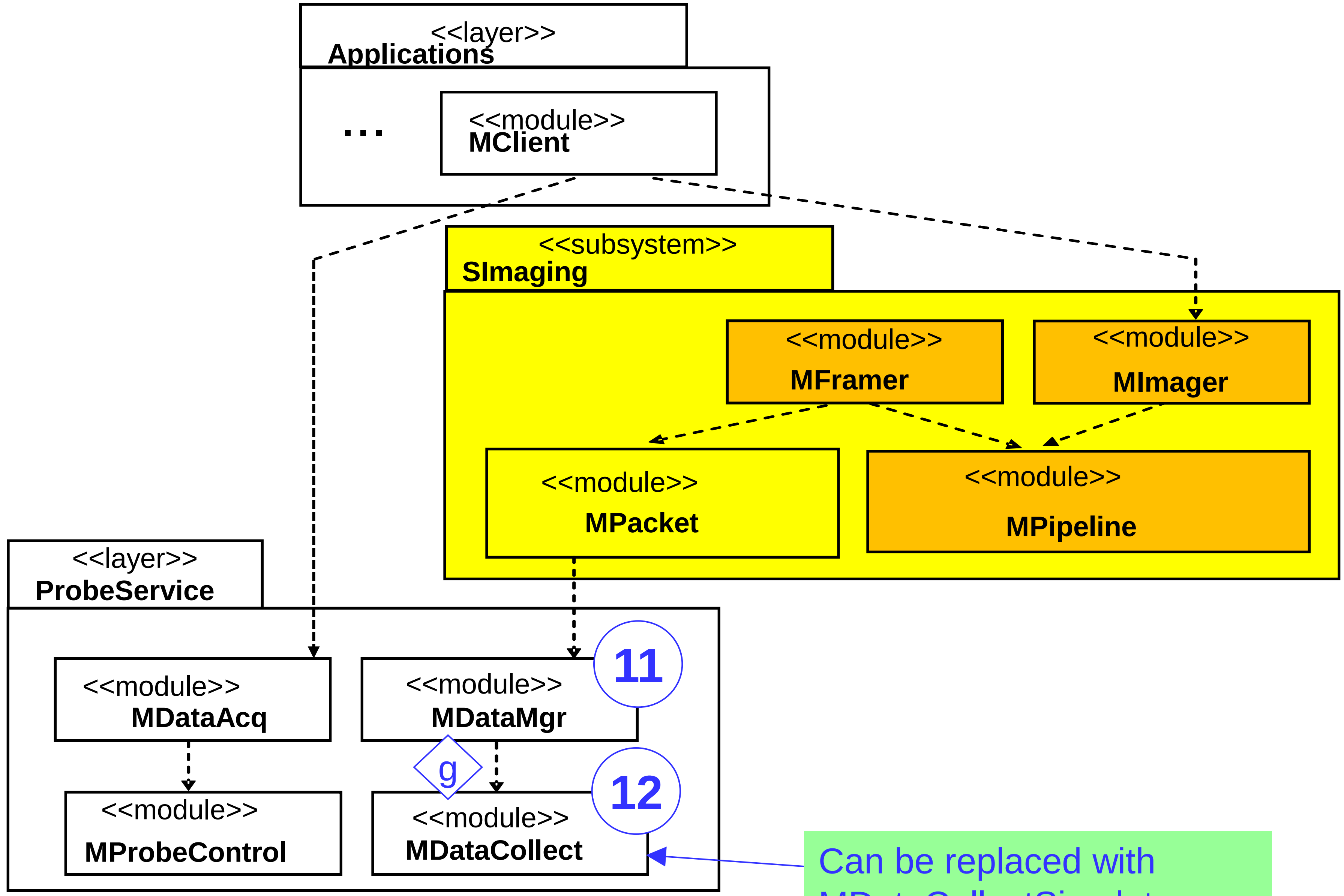


- Each instance of MFramer and MImager has been organized in its package.
- There are packages corresponding to MPacket and MPipeline.

Source Components

- **6th Step:**
  - Look at ProbeService layer
  - Because of the strategy: ***S14A**: Develop an off-line probe simulator with an appropriate abstraction*

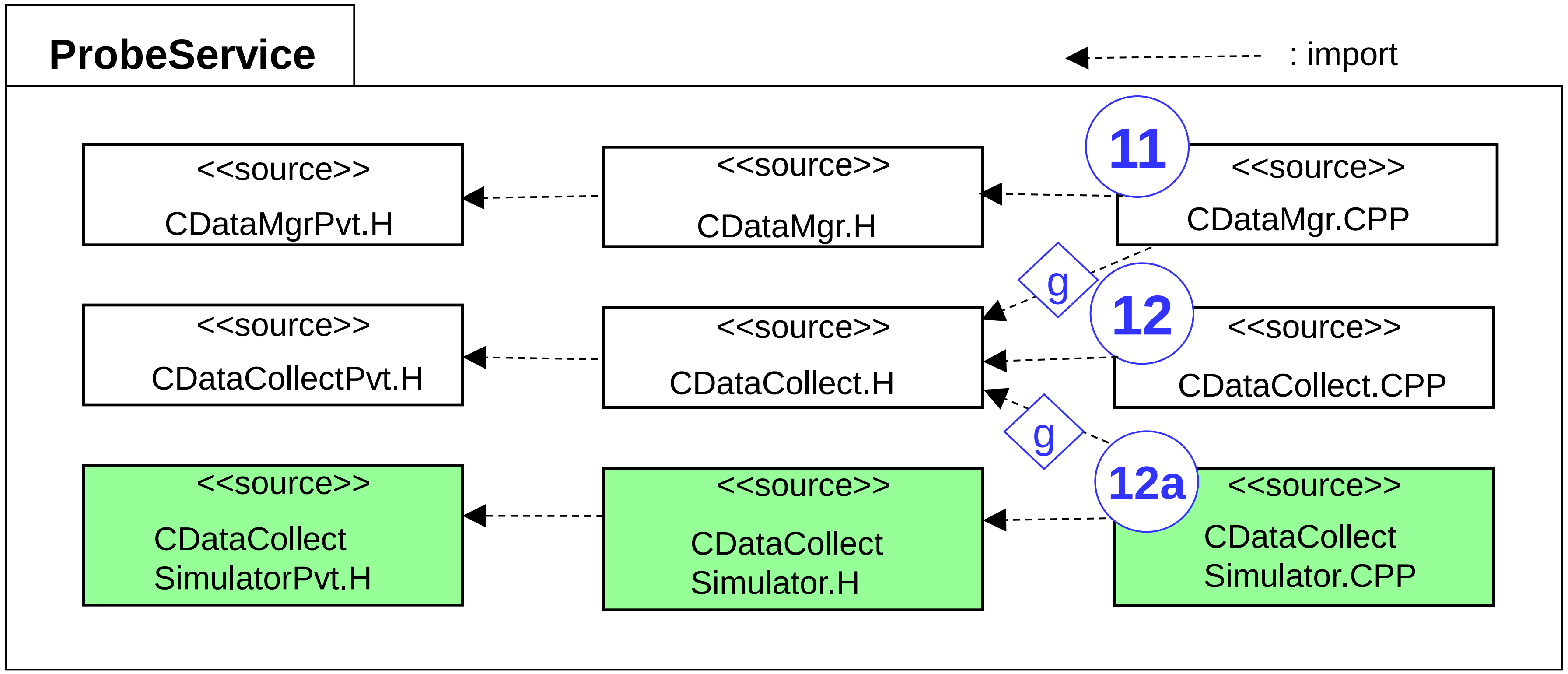
    => MDataCollect module can be replaced with MDataCollectSimulator module



Can be replaced with MDataCollectSimulator module

Figure 5.9. Assigning modules to layers

Figure 7.8 ProbeService source components and their dependencies


ProbeService
: import
11
12
12a
g
g
<<source>> CDataMgrPvt.H
<<source>> CDataMgr.H
<<source>> CDataMgr.CPP
<<source>> CDataCollectPvt.H
<<source>> CDataCollect.H
<<source>> CDataCollect.CPP
<<source>> CDataCollect SimulatorPvt.H
<<source>> CDataCollect Simulator.H
<<source>> CDataCollect Simulator.CPP

Intermediate Components (1/3)

- Strategies to apply:
  - *S14A: Develop an off-line probe simulator with an appropriate abstraction*
  - *S15A: Separate and encapsulate code dependent on the target platform*
  - *S13C: Use phased development*
  - *S13D: Release layers through static libraries*

  ⇒ To allow developers to work independently and concurrently with each other on the target and development platforms of their choice

  ⇒ To improve development cycle time for individual developers

Revision of the source code view considering

- team development

- release plan

- target platform variation

Intermediate Components (2/3)

- Main source components for IS2000 are C++ code components

  => For each .cpp component, there is a .obj binary component
- Compiler, by default, creates .obj files in the same package

  ⇒ Instead, let's organize .obj files in a separate package for each target platform

  => Allows a developer to change the target platform to transform the intermediate components into a format suitable for the particular target platform
- *Use S13D: Release layers through static libraries*

  => One static library for each library
- *Use S14A: Develop an off-line probe simulator*

  ⇒ Two libraries:

  - ProbeService.lib: provides service of the real probe hardware
  - ProbeSimulatorService.lib: provides service of the probe simulator

Intermediate Components (3/3)

- MFramer and MImager are internal to SImaging subsystem.
  - So the library for the subsystem need not contain intermediate components.

- Five separate libraries for the ImageProcessing layer:
  - For the five circled modules in Figure 7.4,
    put them in a subpackage called libs in each layer package

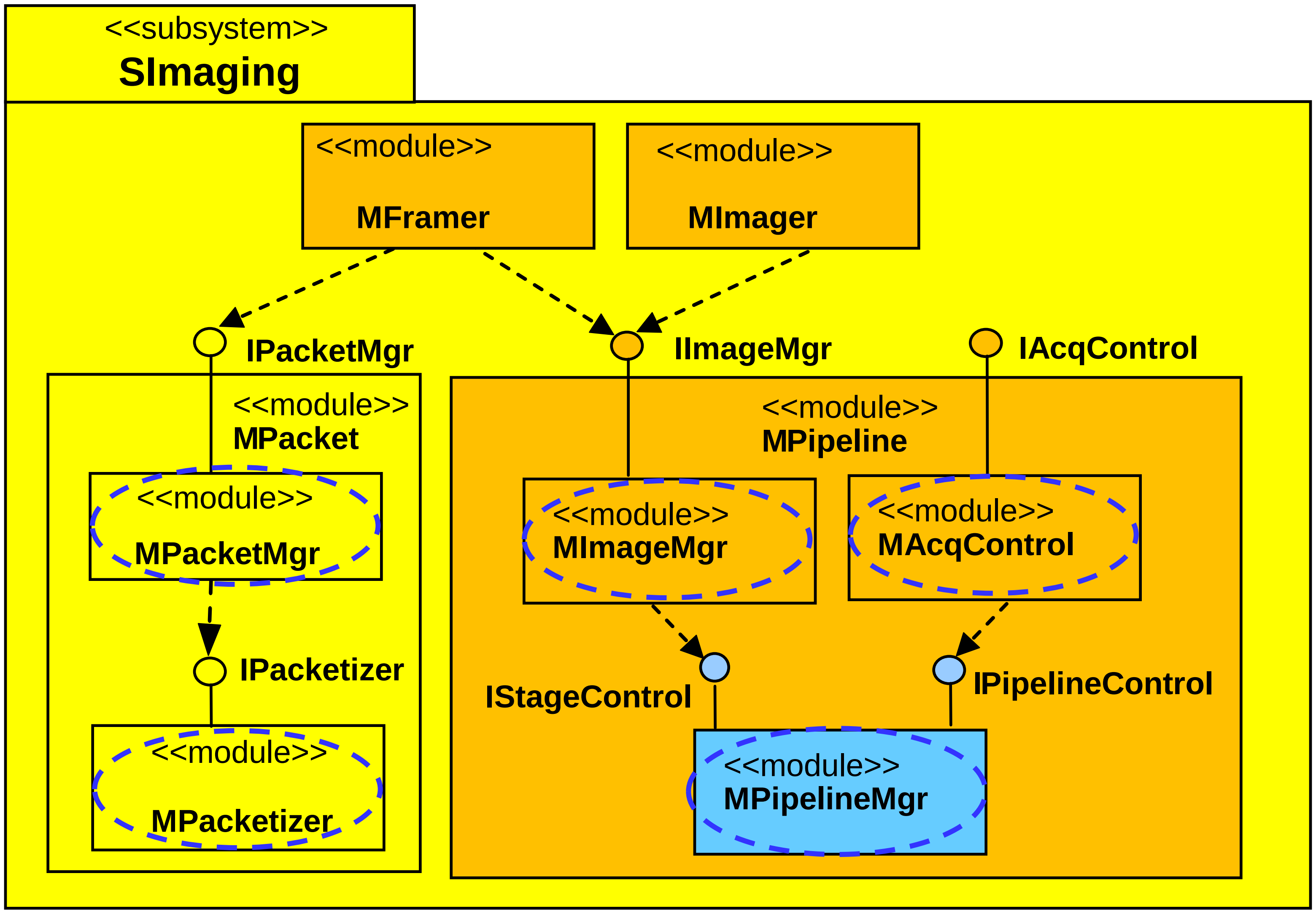


Figure 7.4. Imaging subsystem use-dependencies (from the module view) (Cf. Figure 5.8)

Deployment Components

- Start by mapping executables, configurations and resources
- Focus on the processes in Figure 7.9. For each process, identify an executable in the code architecture view.
- The dependencies of executable to binaries and static libraries can be identified based on the modules assigned to each runtime entity
- Each executable needs to link with intermediate components.
- Figure 7.10
  - Describe deployment components
  - Notes
    1. One package for each executable + configuration description
    2. Executable for different EFramer and EImager processes organized hierarchically
    3. Configurations for each pipeline is mapped to configuration files in Configuration package

Figure 7.9 Execution Configuration of the imaging subsystem

(ksw: Runtime Entities Deployment View)

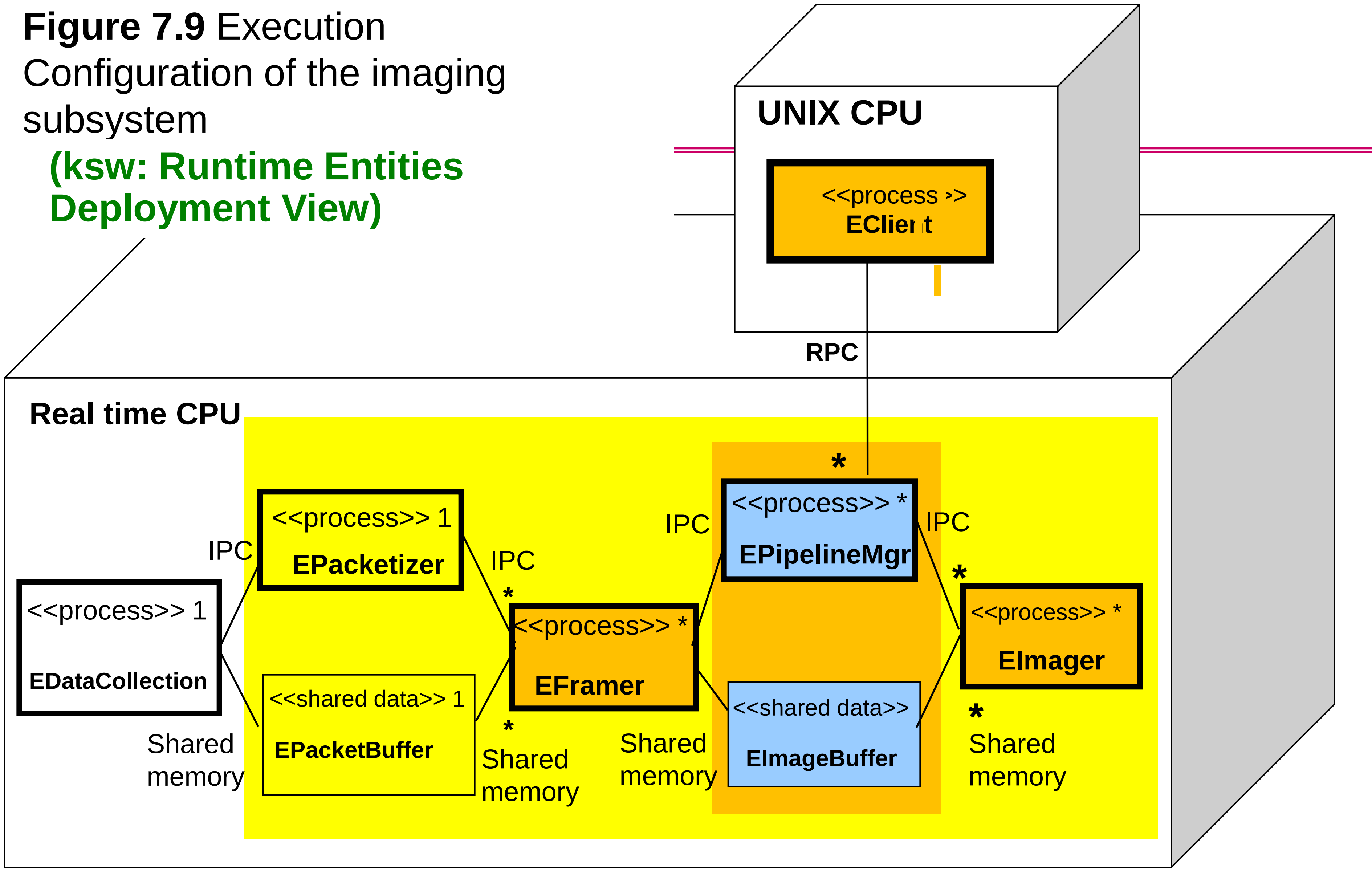
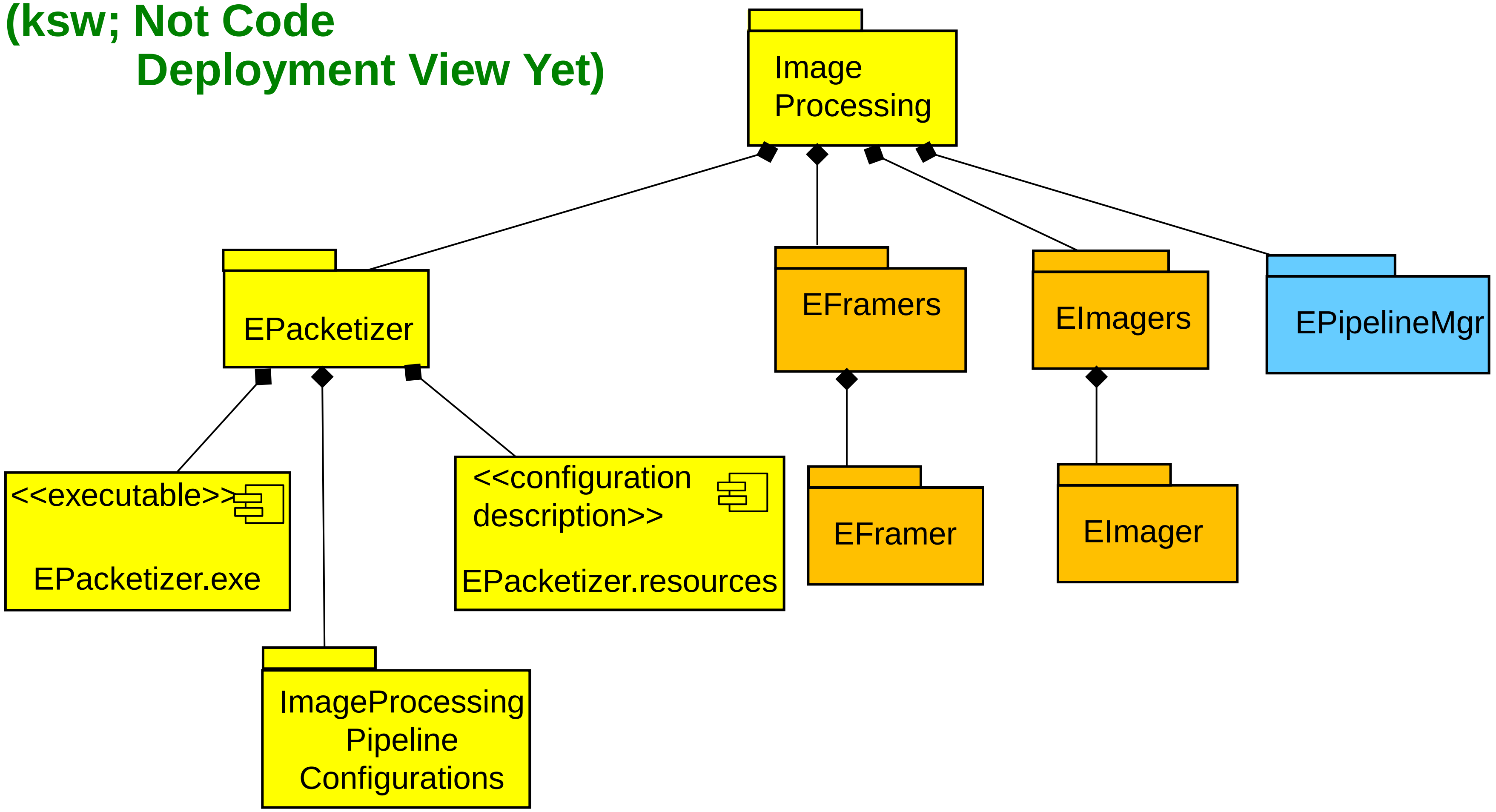
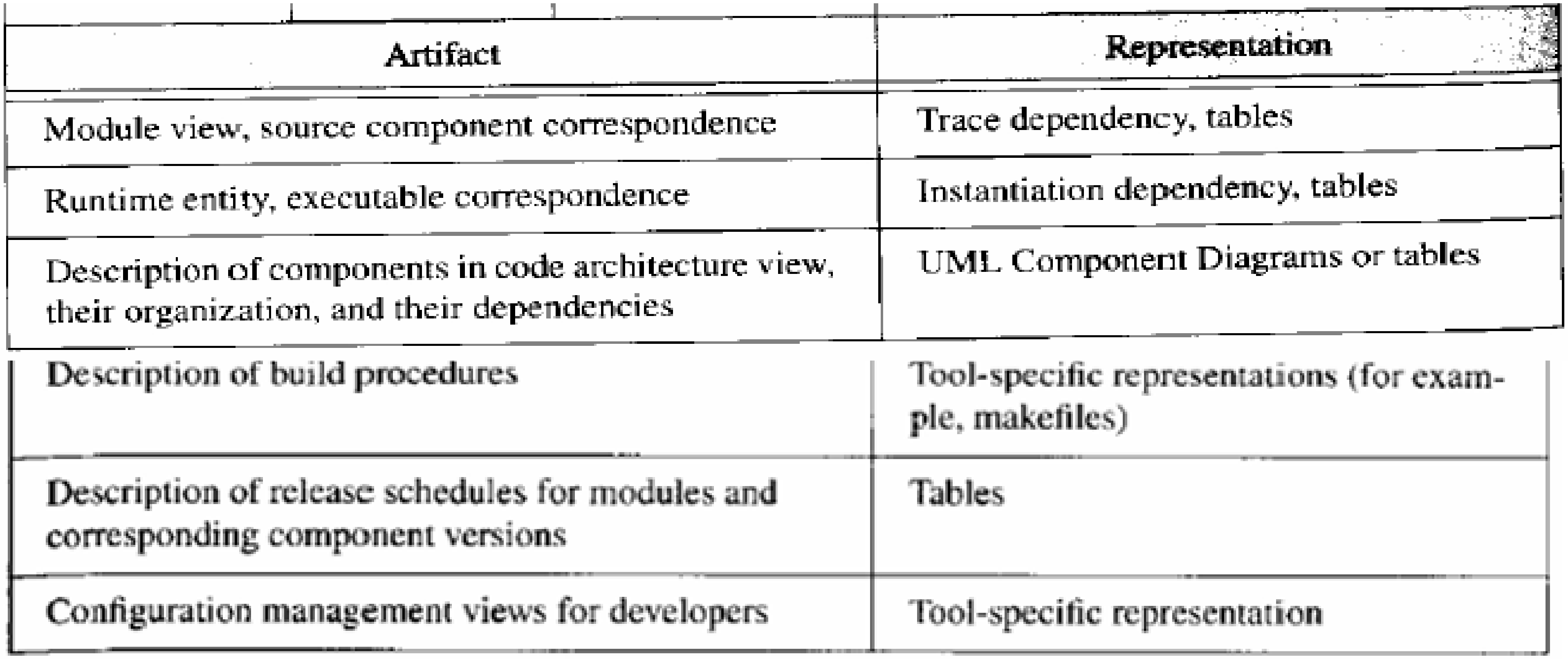


Figure 7.10 Organizing of executables for image processing

(ksw; Not Code Deployment View Yet)



Description of Additional Artifacts



Artifact	Representation
Module view, source component correspondence	Trace dependency, tables
Runtime entity, executable correspondence	Instantiation dependency, tables
Description of components in code architecture view, their organization, and their dependencies	UML Component Diagrams or tables
Description of build procedures	Tool-specific representations (for exam-ple, makefiles)
Description of release schedules for modules and corresponding component versions	Tables
Configuration management views for developers	Tool-specific representation

[3] Final Design Tasks

A. Build Procedure

- Builds parts or the entire system in an efficient manner

- Includes
  - The order in which derived objects are built or rebuilt
  - How the dependencies are expressed in build scripts (e.g. makefiles)
  - Installation procedures to install and organize deployed components

B. Configuration Management

- Design how versions and releases of components are managed
  - Divide the intermediate components into categories based on source components
    - developed by the developer
    - developed by other team members
    - developed by other teams
- For IS2000 project, teams were organized based on layers and subsystems.
- *Using S13C: Use phased development*, decide that each team will release its components first internally for other team members

Questions?