A02-3. *Module View*

2014

Sungwon Kang

A. Purpose

- Bring the architecture close to implementation
  - In conceptual view, the functional relationship is explicit
  - In module view, the relationship between implementing elements must be explicit
    - How the system uses the underlying software platform
      - e.g. OS services etc.

- The following must be mapped to modules
  - application functionality
  - control functionality
  - adaptation
  - mediation

- Not define a configuration, which is in execution view.

Solution Design	Implementation Preparation	Implementation

Repartitioning

Conceptual View

Module View

Execution View

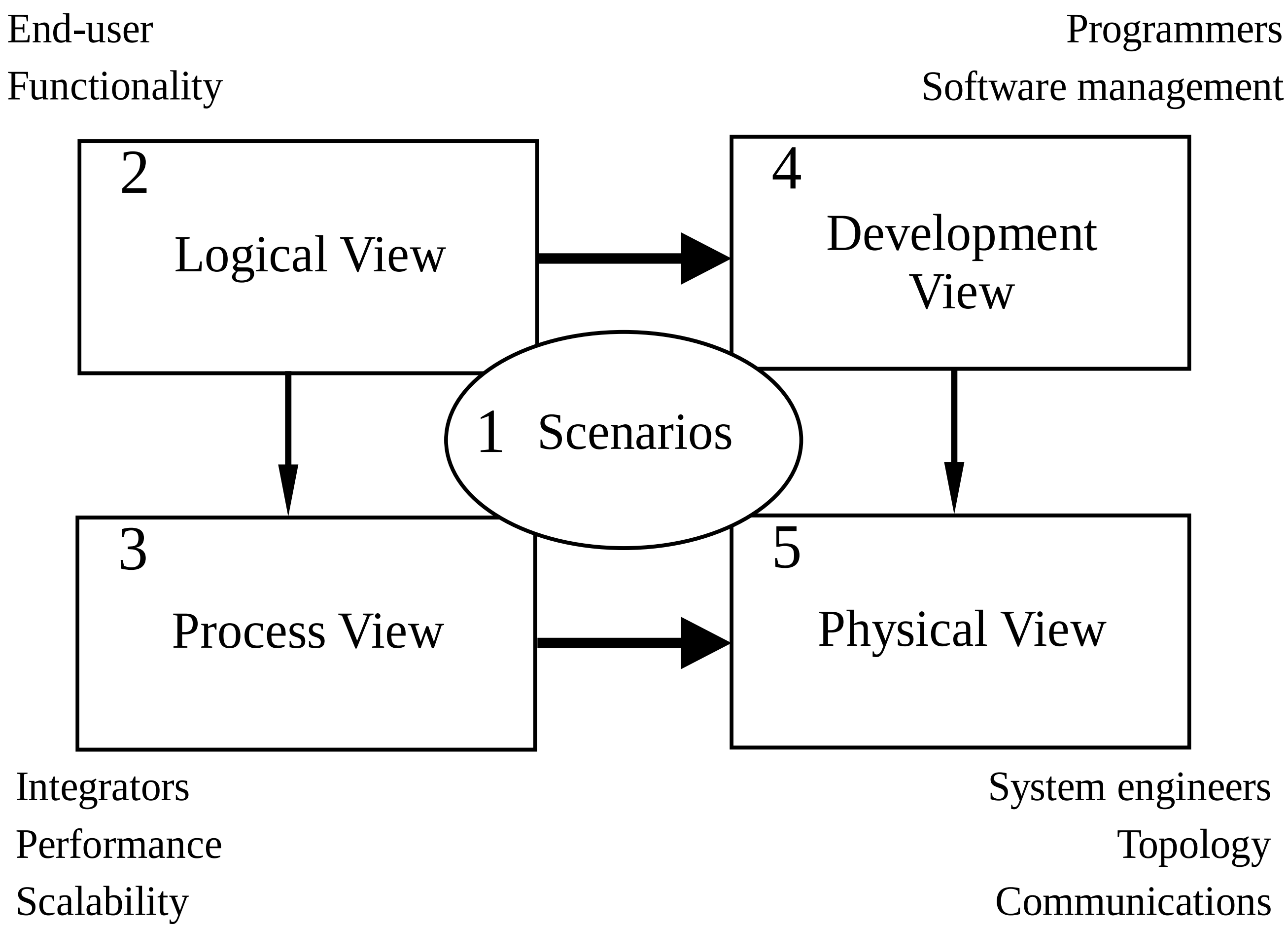
Code View

Components contain functionality,

Connectors contain control and interaction

Modules contain everything

"4+1" View Model ([Kruchten 95]Figure 1)



B. Context

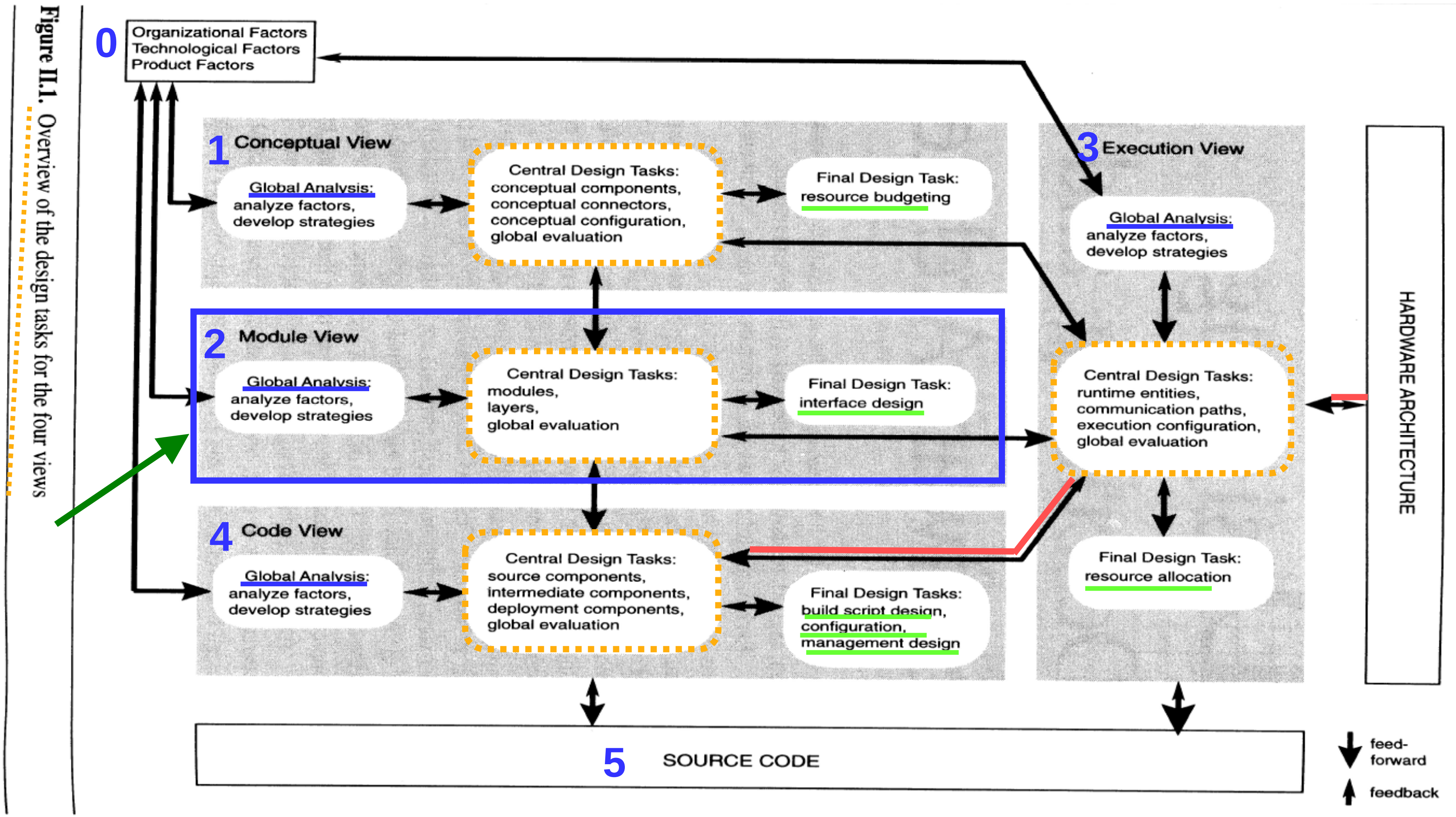


Figure II.1. Overview of the design tasks for the four views
0
Organizational Factors
Technological Factors
Product Factors
1 Conceptual View
Global Analysis: analyze factors, develop strategies
Central Design Tasks: conceptual components, conceptual connectors, conceptual configuration, global evaluation
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

C. Global Analysis

- Similar to the global analysis before
  - **Relevant organizational factors**: staff skills, process and development environment, development budget
  - **Relevant technological factors**: general-purpose hardware, software technology, standards

- Strategies related to
  - Modifiability, portability, reuse
  - Check whether module view supports performance, dependability, failure detection, reporting, recovery

- New factors and strategies may be found.

D. Central Design Tasks (1/2)

1. Map the elements from the conceptual view to subsystems and modules
   – Single conceptual element (component, port , connector, or role) or a set of them –> Module
   - Higher level (i.e. decomposable) conceptual component –> Subsystem

   Note) Subsystem: a conceptual grouping with no corresponding implementation code (= logical grouping of modules)

2. Organize modules into a hierarchy of layers
   - Module can use any of the other modules in the layer
   - When a module usage crosses a layer boundary, the interface required and provided by the modules must also be required and provided by the layer.

   Note) “require”/”provide” relationship becomes “use” or “use-dependency” relationship

Should we call the result of this a layered architecture style?

☛ No. Not only layered architecture applications but also all applications are organized in this way !

D. Central Design Tasks (2/2)

3. Global evaluation
   - Get guidance from
     - Conceptual view
     - Strategies
     - Knowledge and experience
   - Look for feedback
   - Evaluate and adjust

E. Final Design Tasks

- Describe interfaces for each of the modules and layers
  - Detailed interface design based on use-dependencies

- May create new interfaces or split or combine some
  => May feed back to the central design

[1] Global Analysis

Relevant Issues and Strategies

- Issue: Aggressive Schedule
  => S1A: *Reuse existing components*
- Issue: Skill Deficiencies
  => S2B: *Encapsulate multiprocess support facilities*
- Issue: Changes in General-Purpose and Domain-Specific Hardware
  => S3A: *Encapsulate domain-specific hardware*
  => S3B: *Encapsulate general-purpose hardware*
- Issue: Changes in Software Technology
  => S4A: *Use standards*
  => S4B: *Develop product-specific interfaces to external components*
- Issue: Realtime Acquisition Performance
  => S9A: *Separate time-critical from nontime-critical components*
- Issue: Implementation of Diagnostics
  => S11B: *Reduce the effort for error handling*
  => S11C: *Encapsulate diagnostic components*

[2] Central Design Tasks

- Central design tasks:
  - Defining modules
  - Organizing them into layers

Top-down, bottom-up or both

- Approach:

May be repeated

1) Initial layers: use conceptual components
2) Map all the conceptual elements to subsystems and modules
3) Define the use-dependencies between the modules
4) Review in the context
5) Add supporting layers

1) Initial Definition of Layers

- Based on experience, create a working diagram as in Figure 5.4
  ⇒ In the end, we want modules instead of components

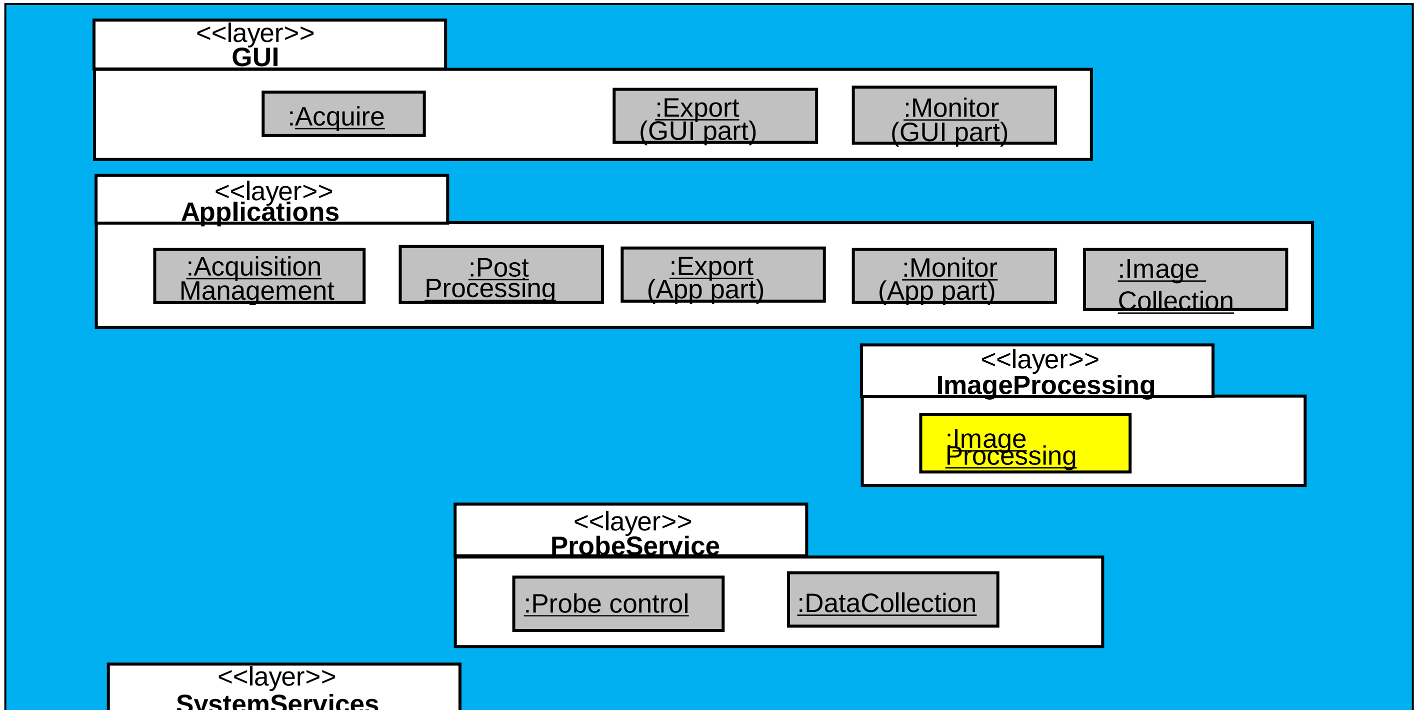


Figure 5.4. Initial creation of layers based on conceptual components.

- Components for the user interface:

  - Acquire, Monitor and Export

    ⇒ Put GUI aspects of these components in the GUI layer

    ⇒ Acquire: mainly a UI functionality

    ⇒ Monitor: GUI + other functionality
    to Applications layer

    ⇒ Export: GUI + image-processing + zoom + image export
    to Applications layer

- GUI defines and manages display and events

  whereas action policies for events are handled by applications

  Example) GUI handles all the user requests, GUI isolates the applications from the low-level interface toolset

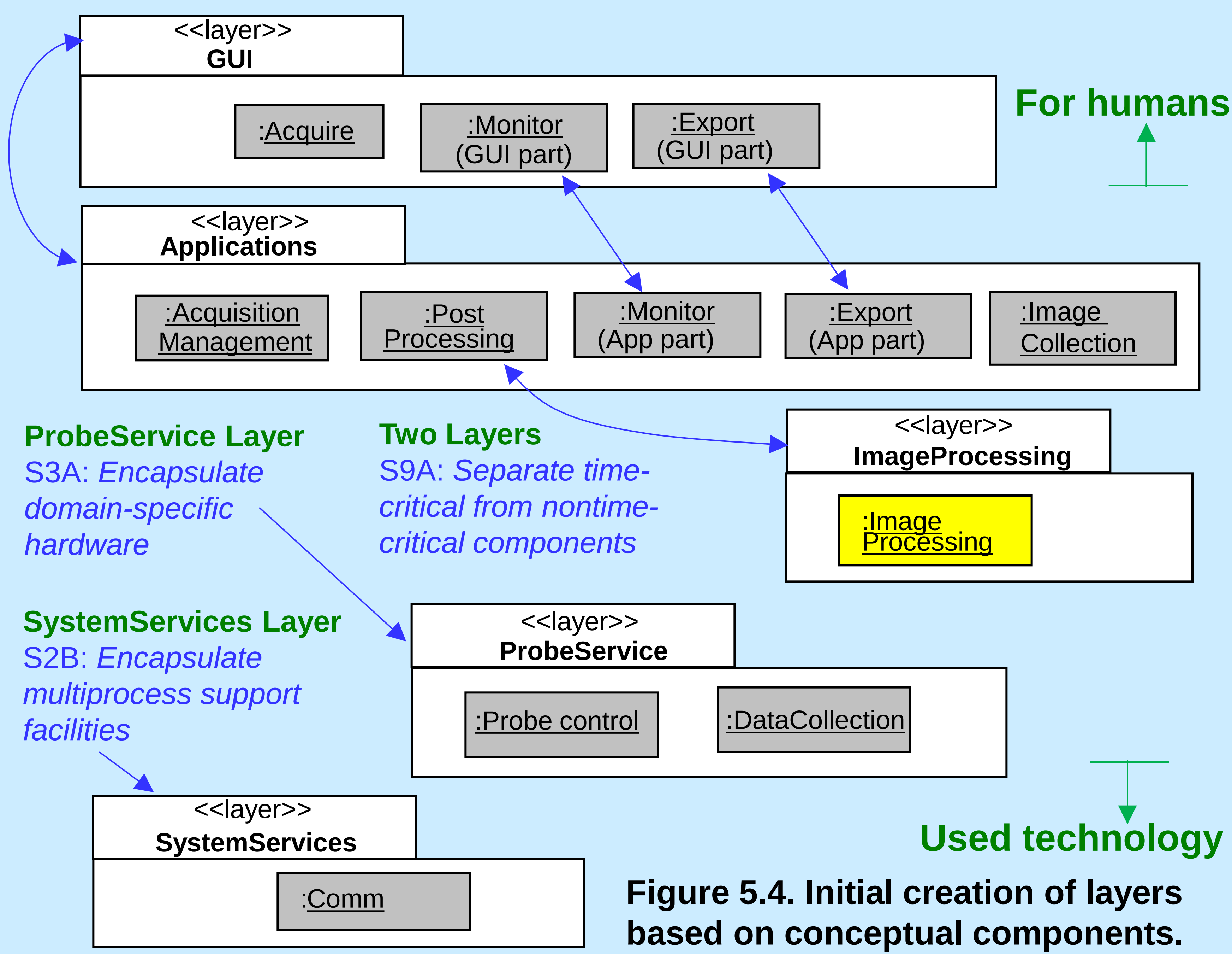
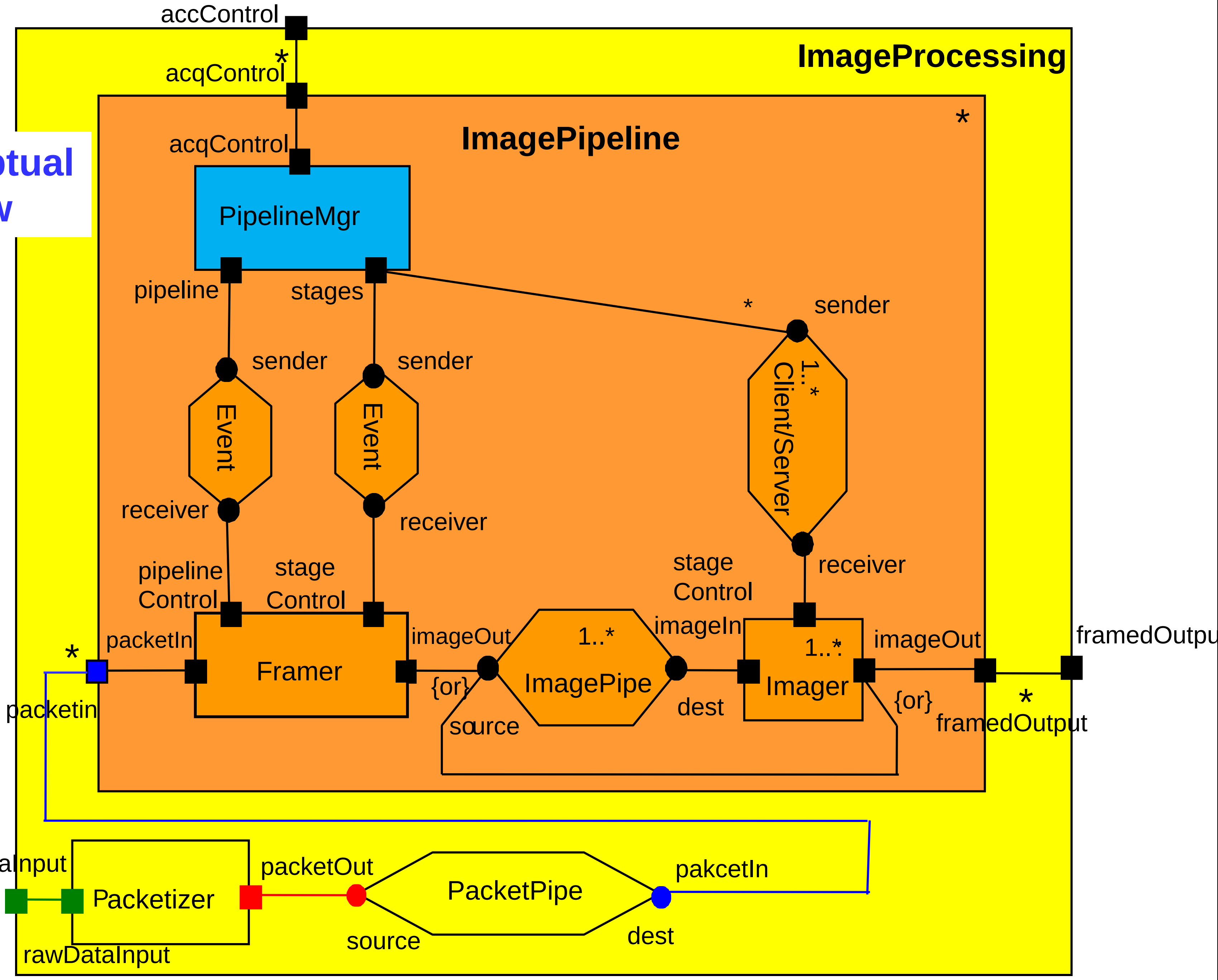


Figure 5.4. Initial creation of layers based on conceptual components.

Conceptual View



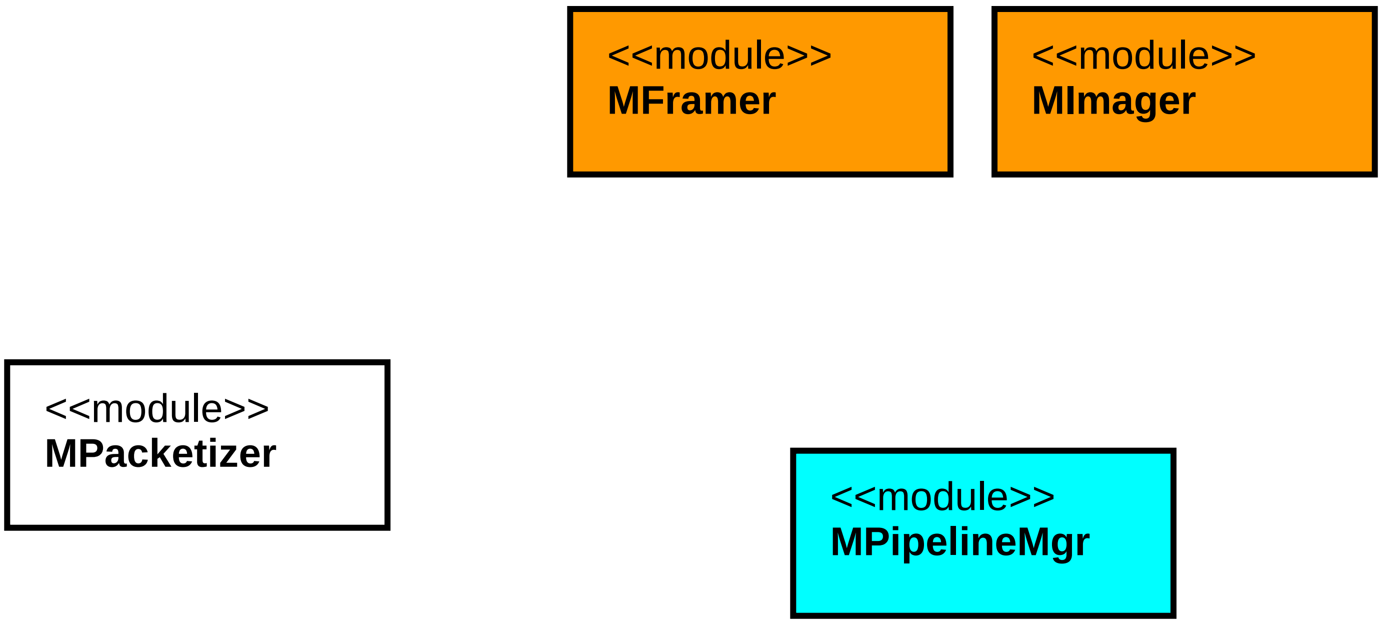
2) Defining Modules for Image Processing

- Begin with the following:

[D21] Assign each component to a single module

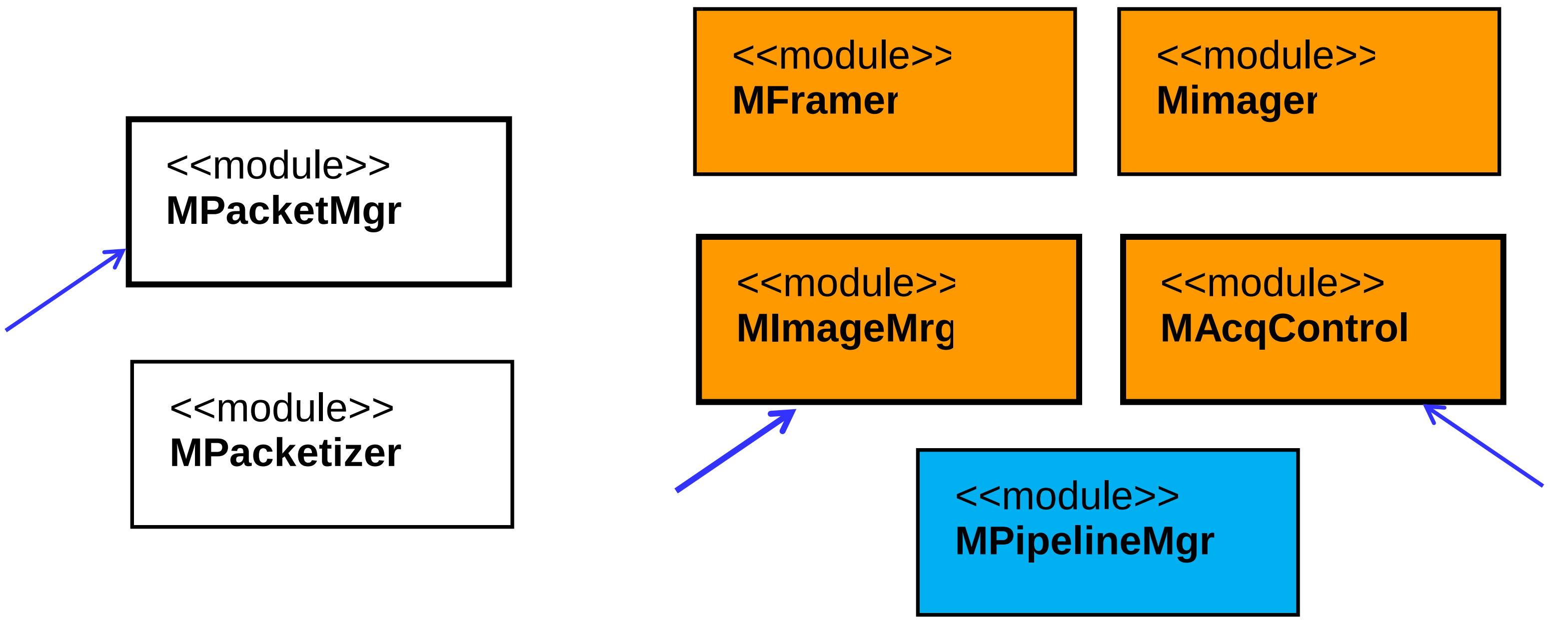
Conceptual Element		Module Element	
Name	Kind	Name	Kind
Packetizer	**Component**	**MPacketizer**	**Module**
PipelineMgr	**Component**	**MPipelineMgr**	**Module**
Framer	**Component**	**MFramer**	**Module**
Imager	**Component**	**MImager**	**Module**

[D21]


<<module>>
MFramer
<<module>>
MImager
<<module>>
MPacketizer
<<module>>
MPipelineMgr

[D22] To *insulate* components *from changes to the software platform*, separate out communication infrastructure into connector-specific/port-specific modules

[D22]



<<module>>
MFramer
<<module>>
Mimager
<<module>>
MPacketMgr
<<module>>
MImageMrg
<<module>>
MAcqControl
<<module>>
MPacketizer
<<module>>
MPipelineMgr

[D23] Map the ImageProcessing and ImagePipeline components to subsystems, and the Packetizer component to a module.

We did this already!

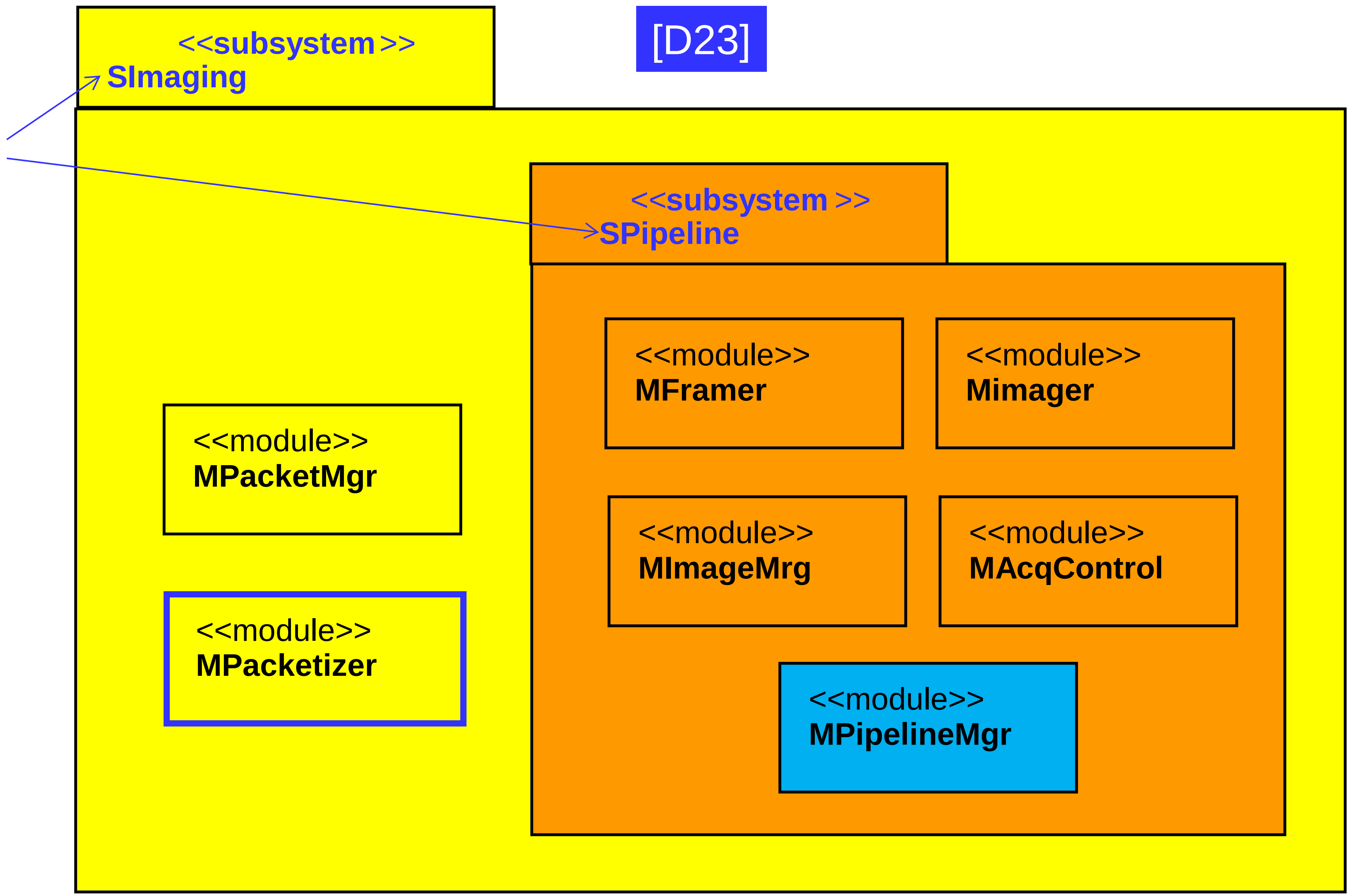


Figure 5.6. Initial containment relationships in Imaging subsystem

1 Experience suggests that because of the volume of the data, we should use a central data buffer for the packetized data rather than give each pipeline its own copy.

Where is this buffer?

⇒ Packetizer and PacketPipe are implemented as the single central buffer manager MPacketizer

- Generally, mapping components and connectors to modules come first and then comes mapping ports and roles to modules

2 packetIn port –> MPacketMgr

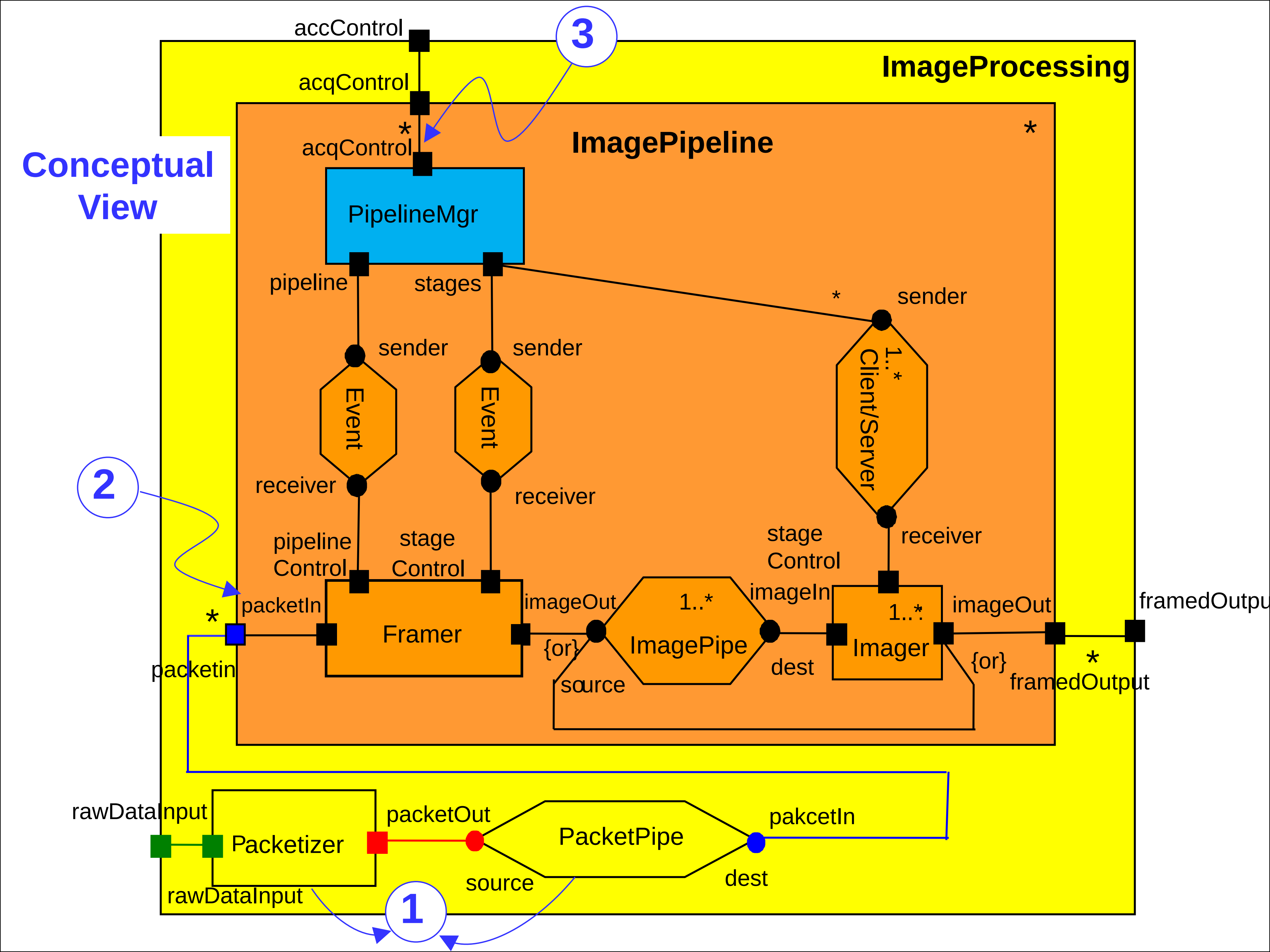
3 acqControl port is the interface used by the acquisition manager for controlling the ImagePipelines

⇒ Want to ensure that only the acquisition manager has access to the MAcqControl module, not to the modules making up the pipeline directly.

☛ Table 5.1

Conceptual Element		Module Element	
Name	Kind	Name	Kind
ImageProcessing	**Component**	**SImaging [D23]**	**Subsystem**
ImagePipeline	**Component**	**SPipeline [D23]**	**Subsystem**
1 **Packetizer** packetOut **PacketPipe** source, dest	**Component** Port **Connector** roles	**MPacketizer**	**Module**
2 packetIn	Port	MPacketMgr	Module
3 acqControl	Port	MAcqControl	Module

Table 5.1 Mapping Conceptual Elements to Module Elements: ImageProcessing



Conceptual View
ImageProcessing
ImagePipeline
accControl
acqControl
acqControl
PipelineMgr
pipeline
stages
sender
sender
sender
Event
Event
Client/Server
1..*
receiver
receiver
receiver
pipeline Control
stage Control
stage Control
packetIn
Framer
imageOut
{or}
source
ImagePipe
1..*
imageIn
dest
Imager
1..*
imageOut
{or}
framedOutput
framedOutput
packetin
rawDataInput
rawDataInput
Packetizer
packetOut
PacketPipe
source
pakcetIn
dest
1
2
3

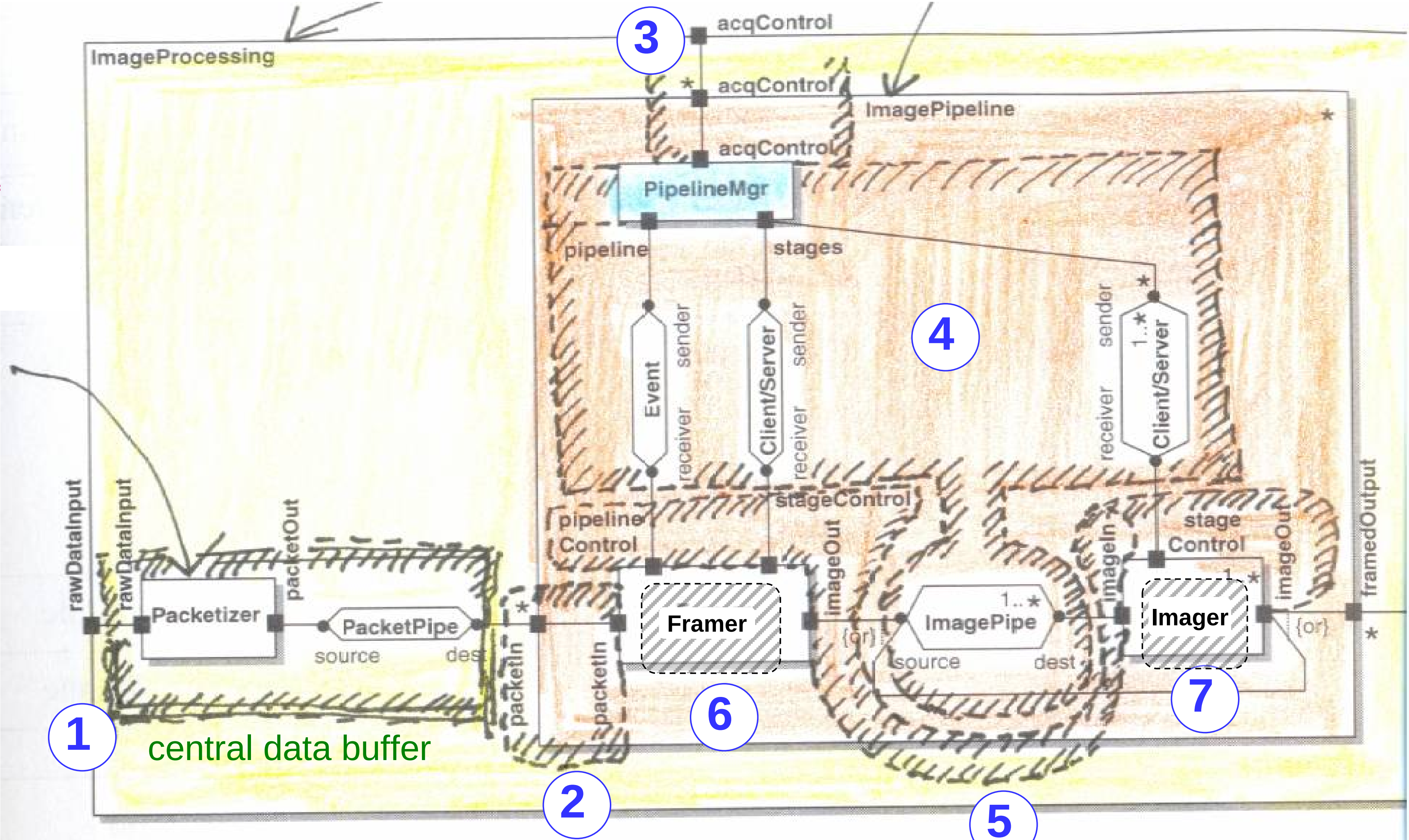


Figure 5.5. Conceptual configuration for ImageProcessing (from the conceptual view)

4 Because of the volume of the data passed between the stages of the pipeline, keep the data in a central buffer with each stage updating it in place.

- Relationship between the pipeline manager and the pipeline stages:
  - Ideally, should encapsulate the high-level protocol specified by the client/server and event connectors as a single module.
  - However, implementations for connectors are not always available and are constrained by the available mechanisms in the software platform.

⇒ Modules for the adjacent port have to implement what is missing

⇒ Decision: use MPipelineMgr (trade-off of an efficient implementation vs. flexibility for future.)

5 At the boundary of the new MPipelineMgr module are pipelineControl, stageControl, imageIn and imageOut ports on the stages

=> Put in MImageMgr module

☛ Table 5.2

	Conceptual Element		Module Element	
	Name	Kind	Name	Kind
4	**PipelineMgr** pipeline, stages **ImagePipe** source, dest **Client/Server** sender, receiver **Event** sender, receiver	**Component** Ports **Connector** roles **Connector** roles **Connector** roles	**MPipelineMgr**	**Module**
5	pipelineControl, stageControl, imageIn, imageOut	Ports	MImageMgr	Module
6	**Framer**	**Component**	**MFramer**	**Module**
7	**Imager**	**Component**	**MImager**	**Module**

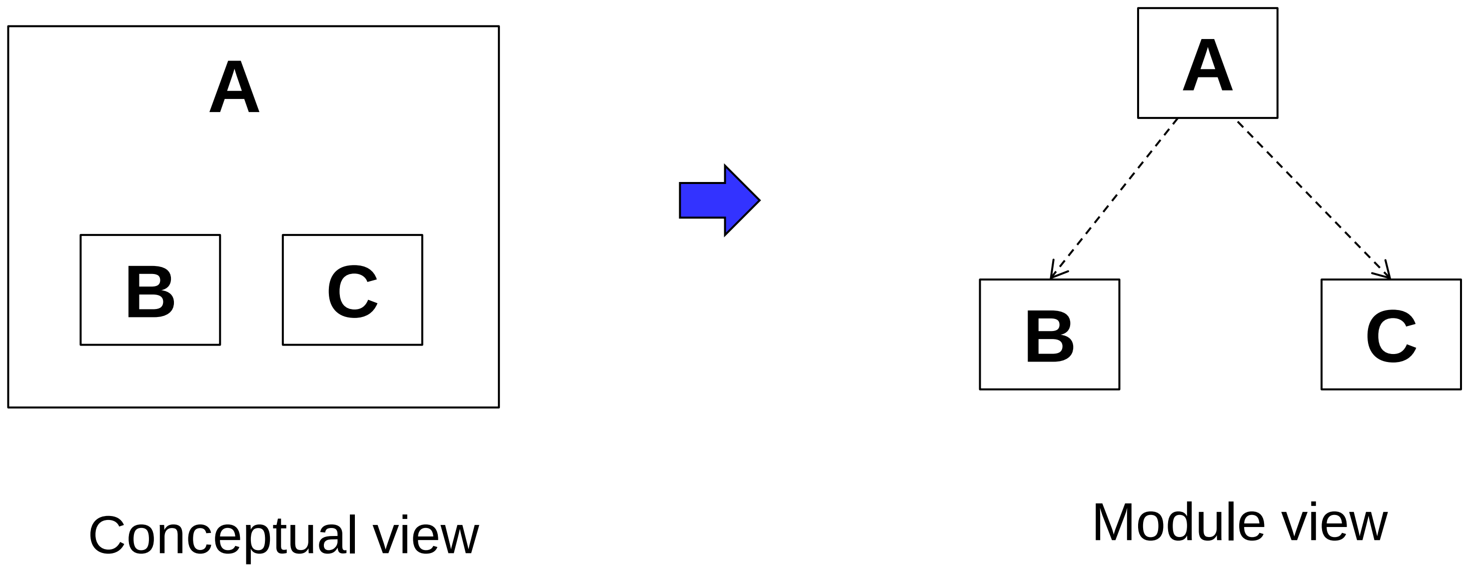
Table 5.2 Mapping Conceptual Elements to Module Elements: ImagePipeline

3) Identify the Dependencies

- Containment-dependency (= Decomposition relationship):

  “*If a conceptual component is decomposed into lower level components, there is a dependency from the corresponding parent module or subsystem to the child module or subsystem.*”

  ⇒ Fig 5.6



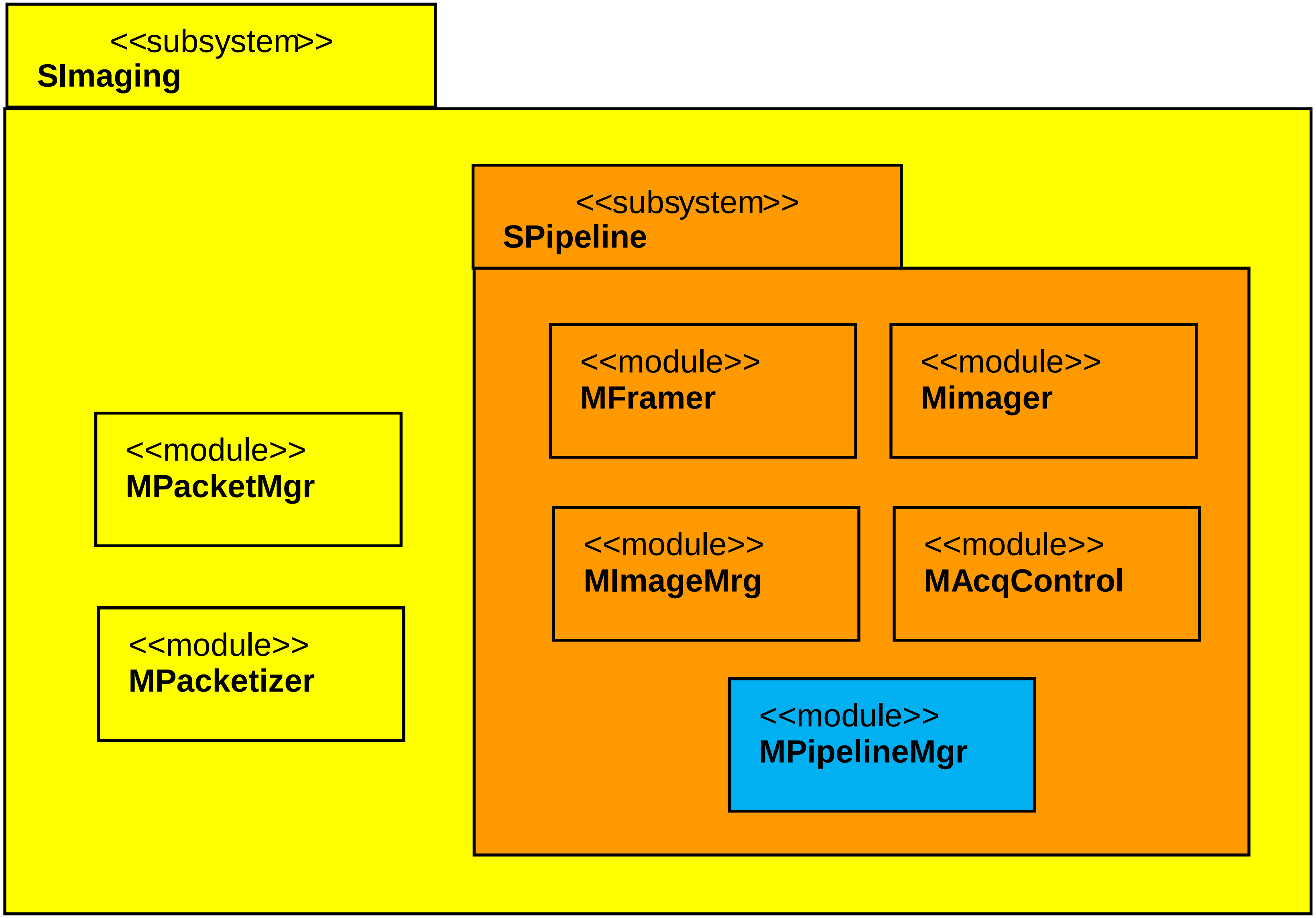
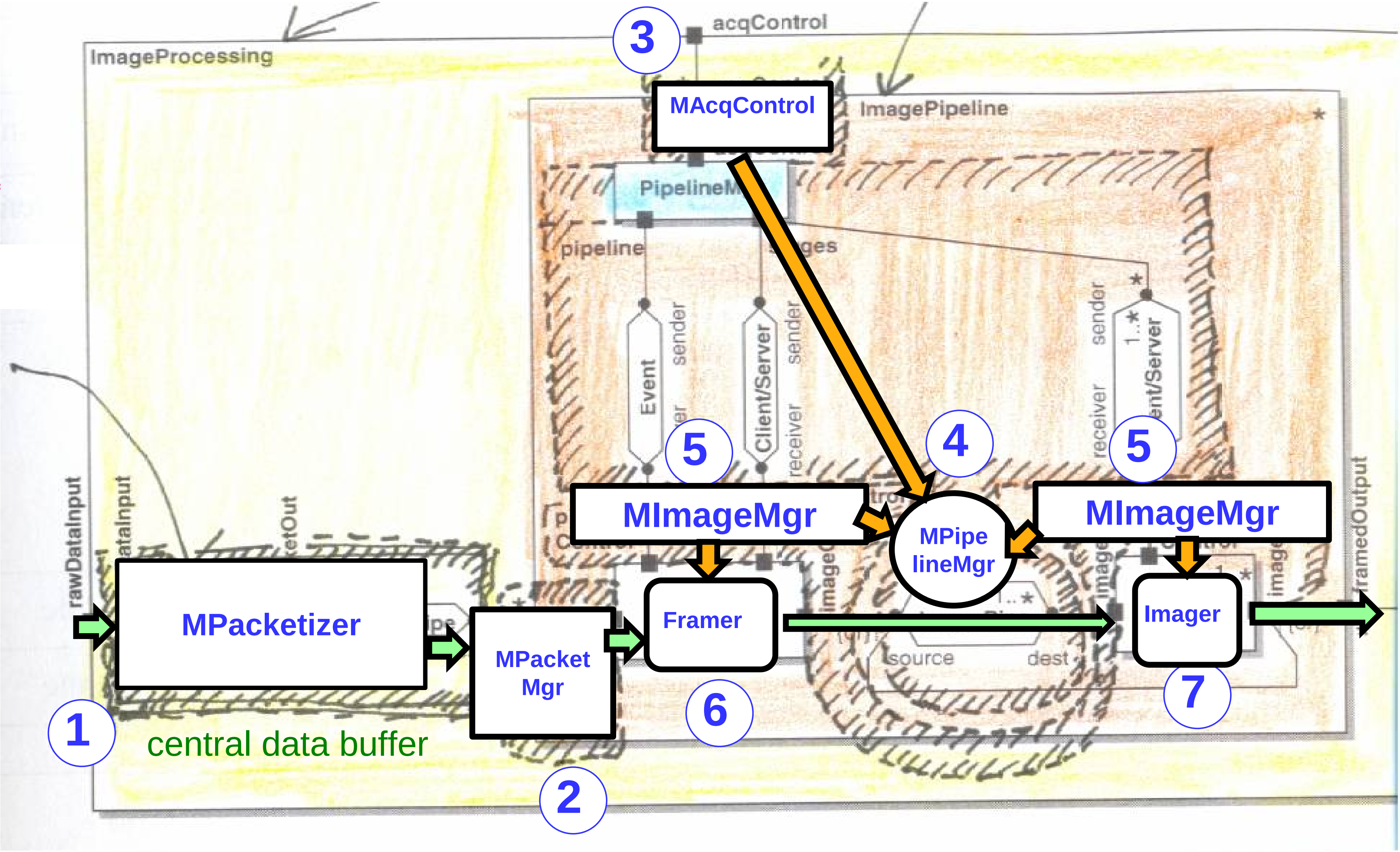


Figure 5.6. Initial containment relationships in Imaging subsystem

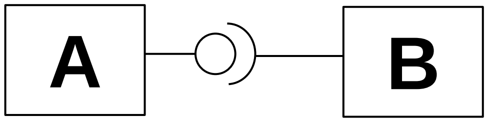
No dependency based on containment of conceptual elements appear in this module dependency view.



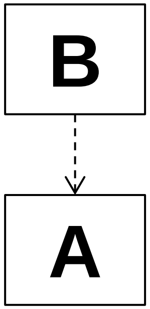
Data flow from left to right maps to the use-dependency between MPacketizer, MPacketMgr and MFramer.
In the process, MImager uses service of MImageMgr, which uses MPipelineMgr.

- Use-dependency:

  "*If a conceptual component provides a service to another component, there is a dependency from the user to the provider of the service.*



Conceptual view



Module view

[D24] Because MAcqControl and MImageMgr act as proxies to request services of MPipelineMgr, they are tightly coupled and implemented together.

[D25] Similarly with MPacketizer and MPacketMgr

- After D24 and D25, SPipeline subsystem is not needed.

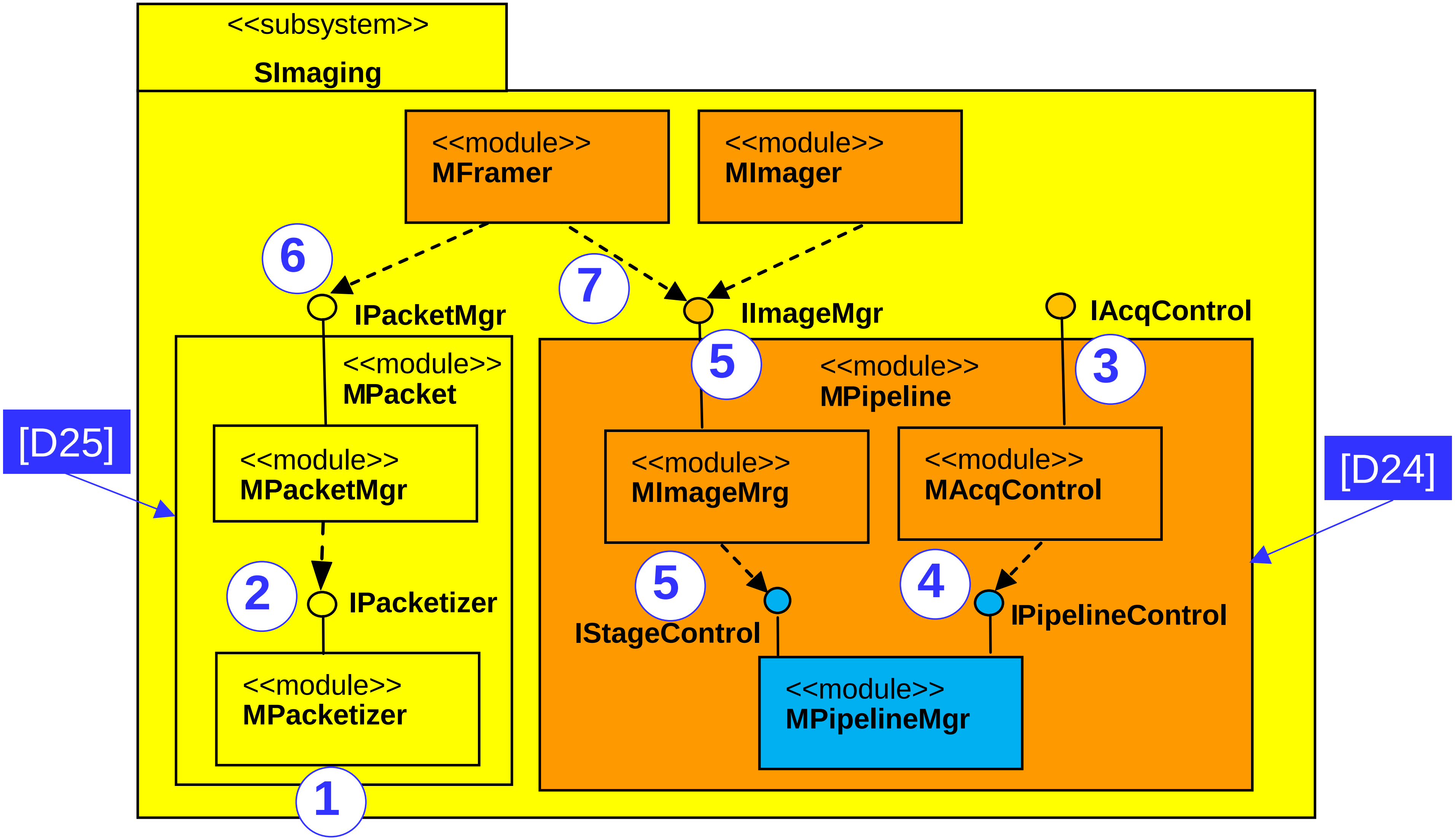
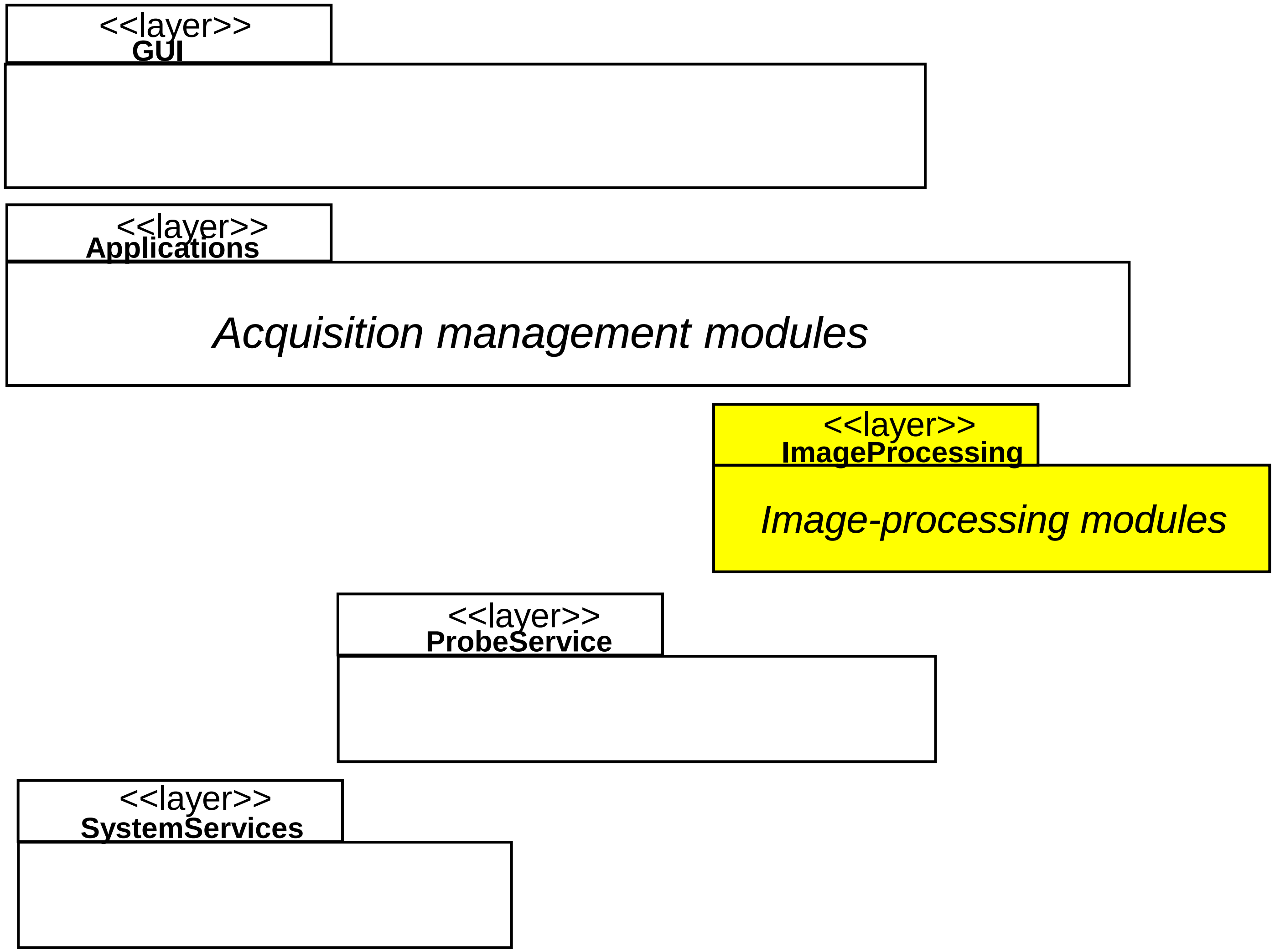


Figure 5.8. Imaging subsystem use-dependencies

Depicts both containment and use dependencies

4) Reviewing in the Context



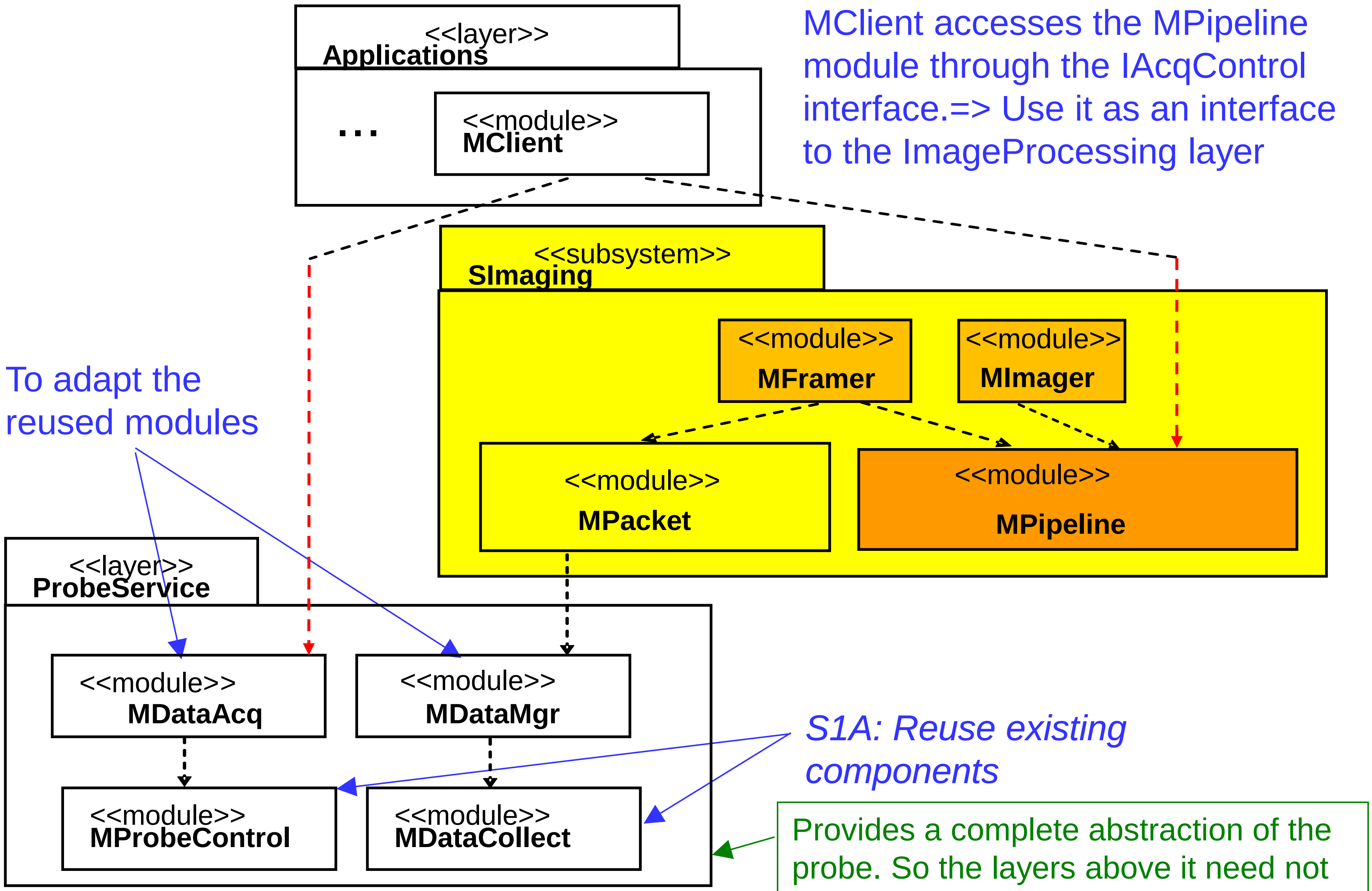


Figure 5.9. Assigning modules to layers

Based on the module dependencies in Figure 5.9.

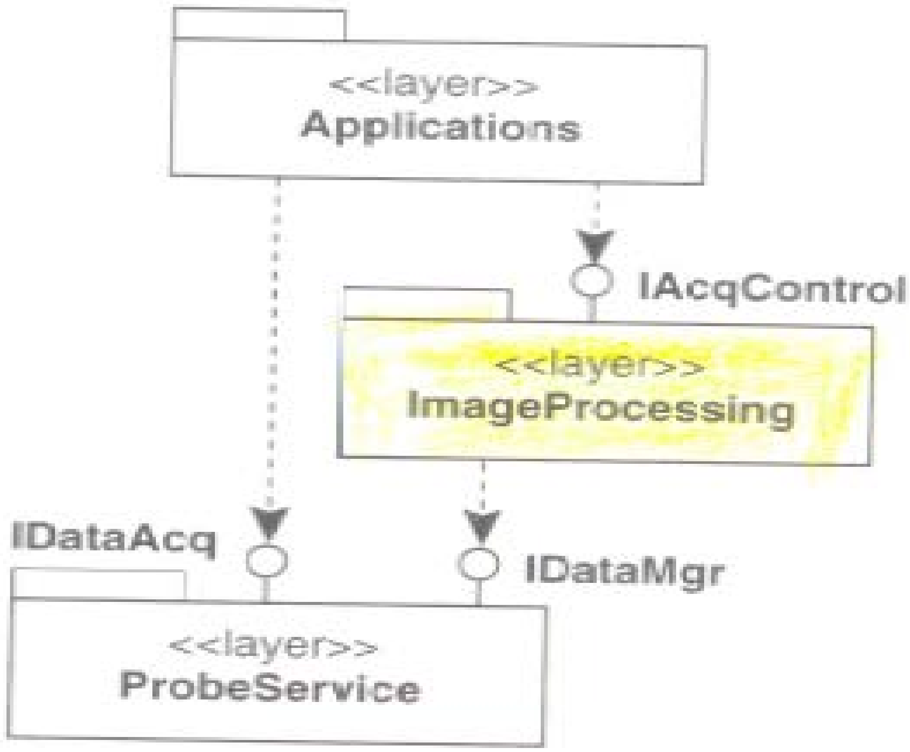


Figure 5.10. Use-dependencies between layers

5) Adding Supporting Layers

A S3B: *Encapsulate general purpose hardware*

=> Encapsulate OS

B Domain-specific support services for storage and communication

– Requirement: ImageCollection component store images for as long as 24 hours

=> Use a commercial database and use this to support recovery (See next slide)

=> implements the PersistentDataPipe connector between the ImageProcessing and PostProcessing components

S4B: *Develop product-specific interfaces to external components*

=> Move image collection modules from Application layer to DatabaseService layer

=> Consists of 3 sublayers

- Database Administration: responsible for installation, configuration, maintenance and database utilities

- Database interface: vendor-independent interface to DBMS

- DBMS: vendor supplied

Technological Factor	Flexibility and Changeability	Impact
T3: Software technology		
T3.3: Database management system (DBMS)		
Use a commercial DBMS.	The DBMS may be upgraded every five years as technology improves.	The impact is transparent, provided it conforms to our DBMS interface standard. Change from a relational to an object-oriented DBMS may have a large impact on all components.
T5: Standards		
T5.5: Standard for DBMS interface		
Open Database Connectivity has been selected as the database access standard.	The standard is stable.	There is a large impact on components interfacing with the DBMS.

Table 5.3. Factors Added During Module View Design

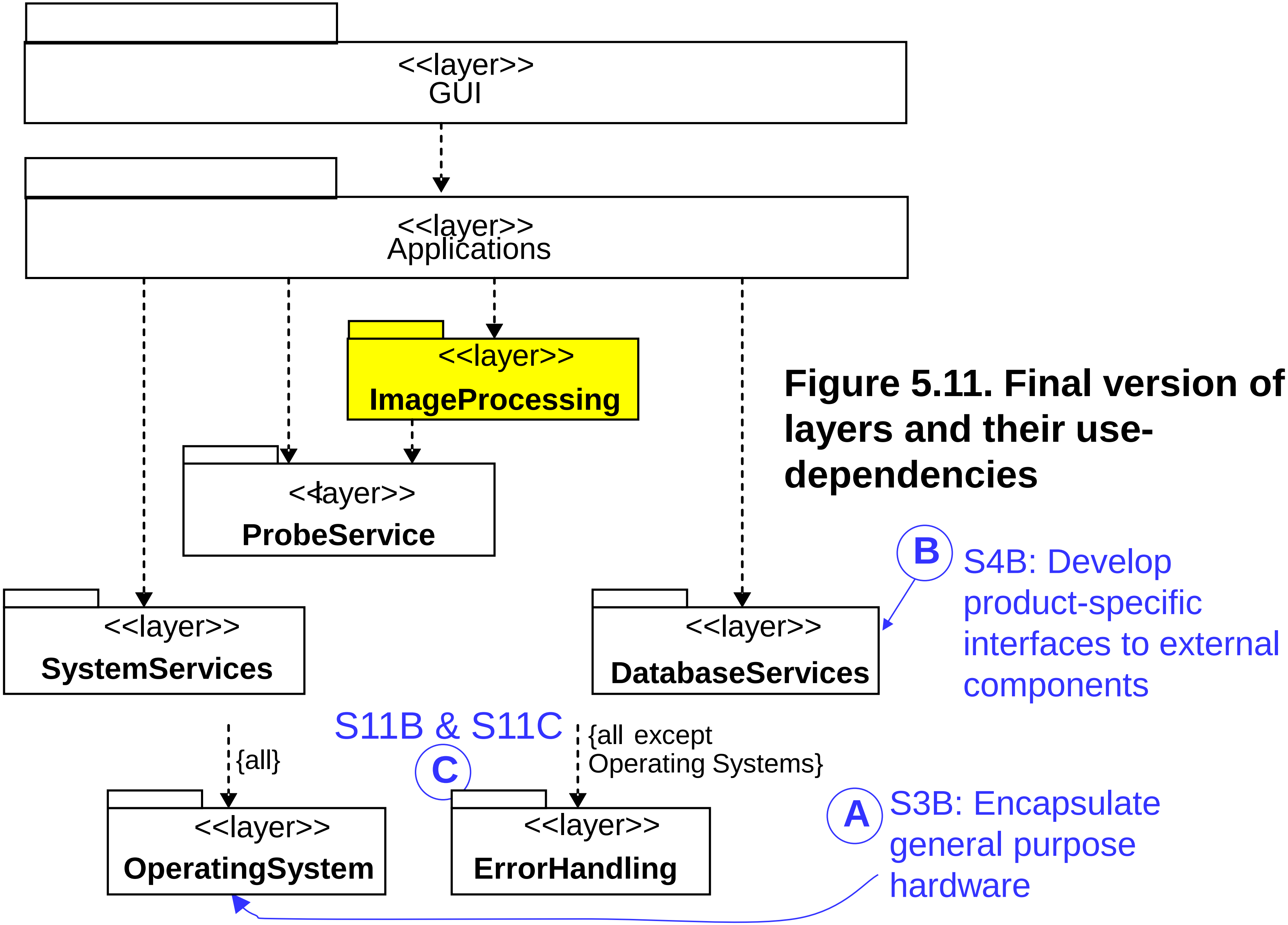


Figure 5.11. Final version of layers and their use-dependencies

C To implement the requirements for error handling and logging

=> introduce modules for a logger and its interface

- logger: receive and store event messages

=> S11C: *Encapsulate diagnostic components*

- interface: any other module that needs the logger can use it

=> S11B: *Reduce effort for error handling*

[3] Final Design Tasks

- Describe interfaces in detail

  –> Use protocol definitions from conceptual view

    - No 1-to-1 correspondence between a protocol definition and an interface definition

      <= The ports or connectors that obey the protocol may be mapped to modules that are split or combined.

- Interface definition:
  - Not all interfaces need be defined

  Example See next slide.

Interface definition for IAcqControl

`Initialize( )`
`        Client initializes interface with ImageProcessing`
`Create(IP_id, Pipeline_config)`
`        Create an image pipeline with the given configuration`
`Delete(IP_id)`
`        Stop execution of the pipeline and tear down the stages`
`Terminate( )`
`        Client terminates interface with ImageProcessing`
`AdjustImage(IP_id, Stage_id, Message)`
`        Client adjusts the processing of the image`
`        (e.g., growth rate, persistence)`

Figure 5.12. Interface definition for IAcqControl

Need much more than this

Questions?