View Modeling

Lecture 07

BIL428 Software Architectures

Mustafa Sert
Asst. Prof.
msert@baskent.edu.tr

Department of Computer Engineering, Başkent University
Ankara 06810 TURKEY

Agenda..

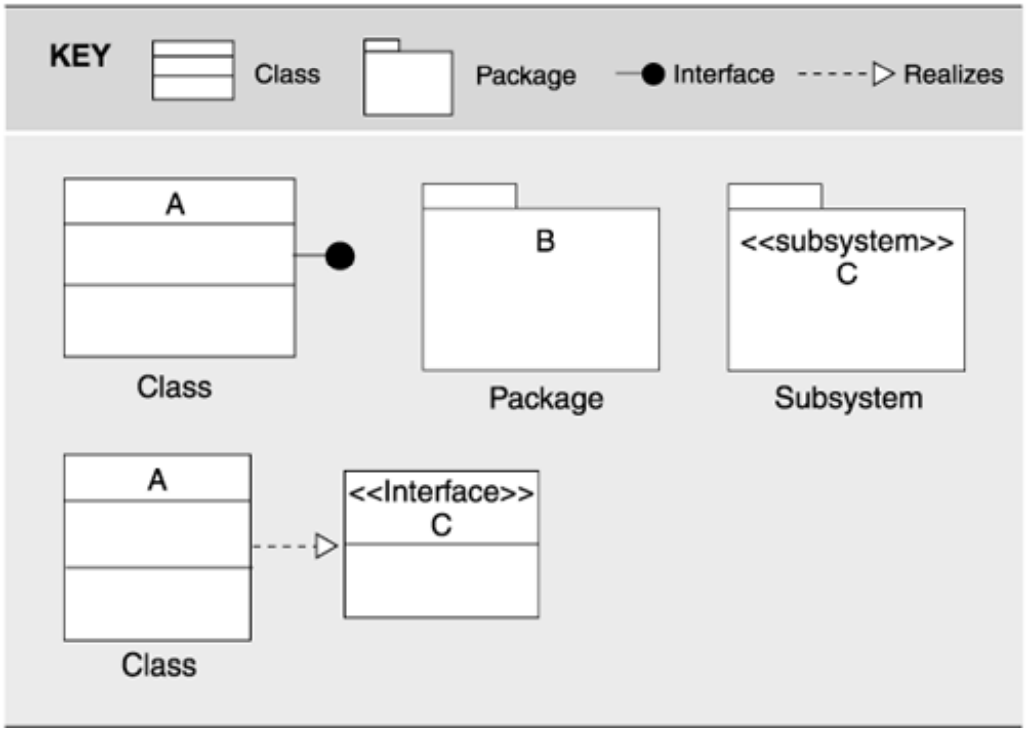
- About Viewtypes
- Module Viewtype
- Component-And-Connector Viewtype
- Allocation Viewtype

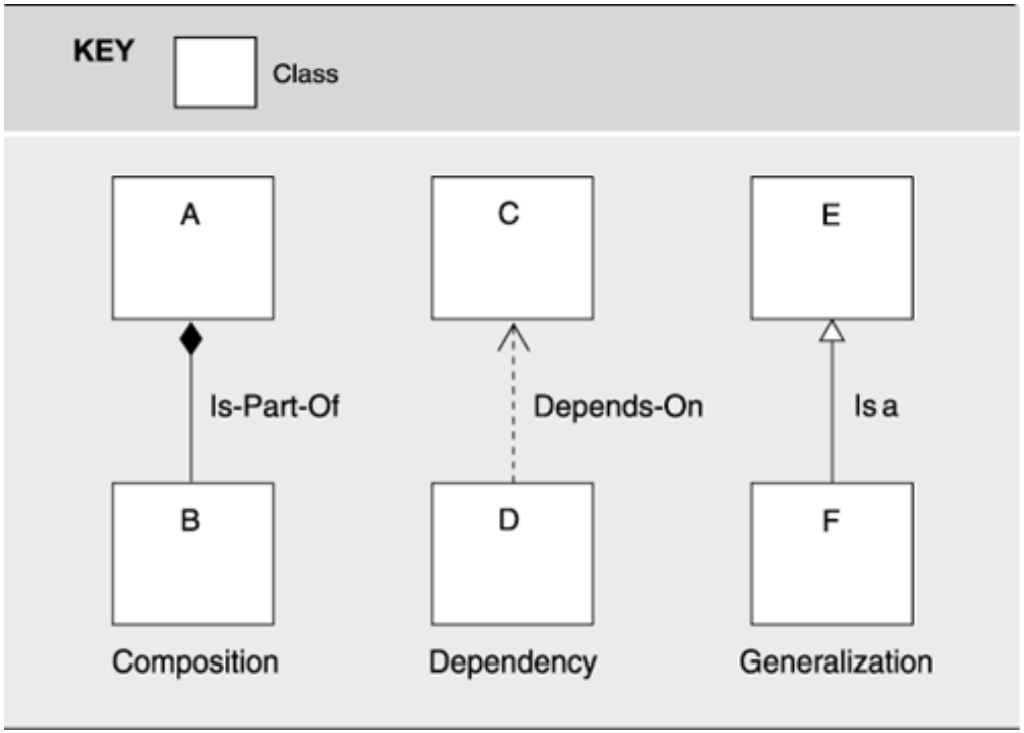
About Viewtypes..

- An architect must consider the system in three ways
  - How is it structured as a set of implementation (code) units (modules) ?
    - **Module Views (Module viewtypes)**
  - How is it structured as a set of elements that have run-time behavior (components) and interactions (connectors)?
    - **Component-and-Connector views (C&C viewtypes)**
  - How does it relate to non-software structures in its environment (i.e., CPUs, file systems, networks, development teams, etc.)?
    - **Allocation views (Allocation viewtypes)**

Module Viewtype

- Document a system's principal units of implementation
- **Elements:** A class, a package, a layer, or any decomposition of the code unit
- **Relations:** is-part-of, depends-on, is-a





Module Viewtype Styles

- A style is a specialization of viewtype
- Predefined styles for Module Viewtypes
  - Decomposition Style
    - Decomposition relation
  - Generalization Style
    - Is-a relation
  - Uses Style
    - Specializes depends-on relation
  - Layered Style
    - Modules are layers
    - Specializes depends-on relation to allowed-to-use
    - Code in higher layers is allowed to use code in lower layers

Decomposition Style

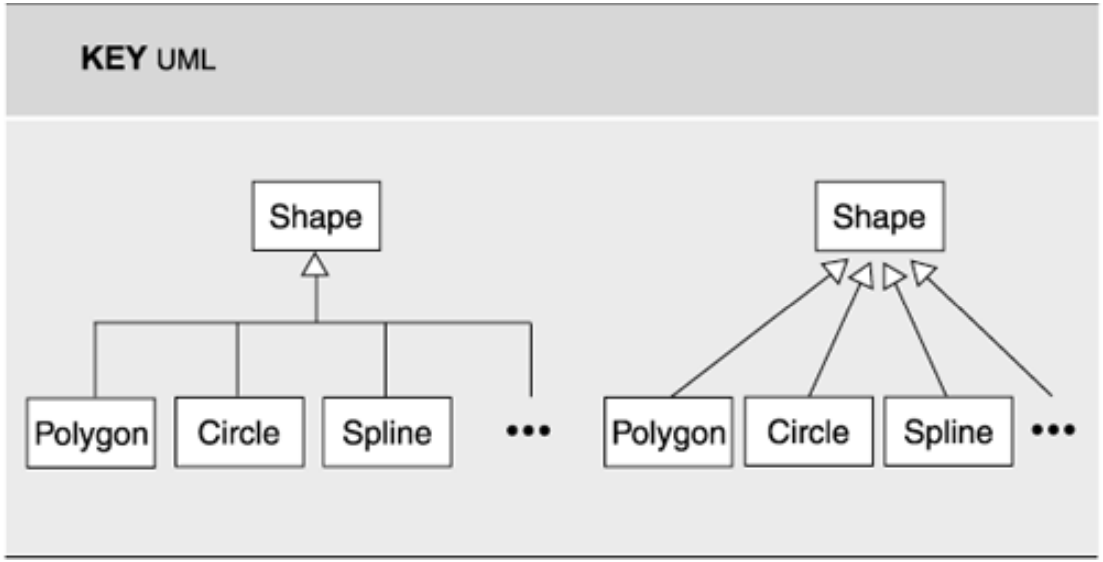
- Almost all architectures begin with module decomposition style. **Divide and conquer**
- Different purposes for decomposition
  - Achievement of quality attributes
    - E.g., modifiability performance, etc.
  - Build-versus-Buy decision
    - Modules may be bought or reused
  - Product line implementations
    - Distinguish btw common modules and variable modules

Uses Style

- Uses relation is a special form of depends-on relation
  - Module A **uses** module B if A **depends on** the presence of a correctly functioning B
  - P1 is said to use P2 if P1's correctness depends on a correct implementation of P2 being present. That is, P1 depends on the result of P2
- Uses relation is NOT a Calls relation
  - A program P1 can **use** P2 without calling it
    - E.g., P2 leaves/updates a shared variable that is used by P1
  - A program P1 can **call** program P2 without using it
    - E.g., P2 is an exception handler that is called (but NOT used) by P1

Generalization Style

- **Is-a** relation in the module viewtype
- The style can be used to support
  - Object-oriented designs (e.g., inheritance)
  - Extension and evolution (e.g., for producing incremental descriptions)
  - Local Change or Variations (e.g., defining commonalities at higher level and to define variations as children of a module)
  - Reuse!

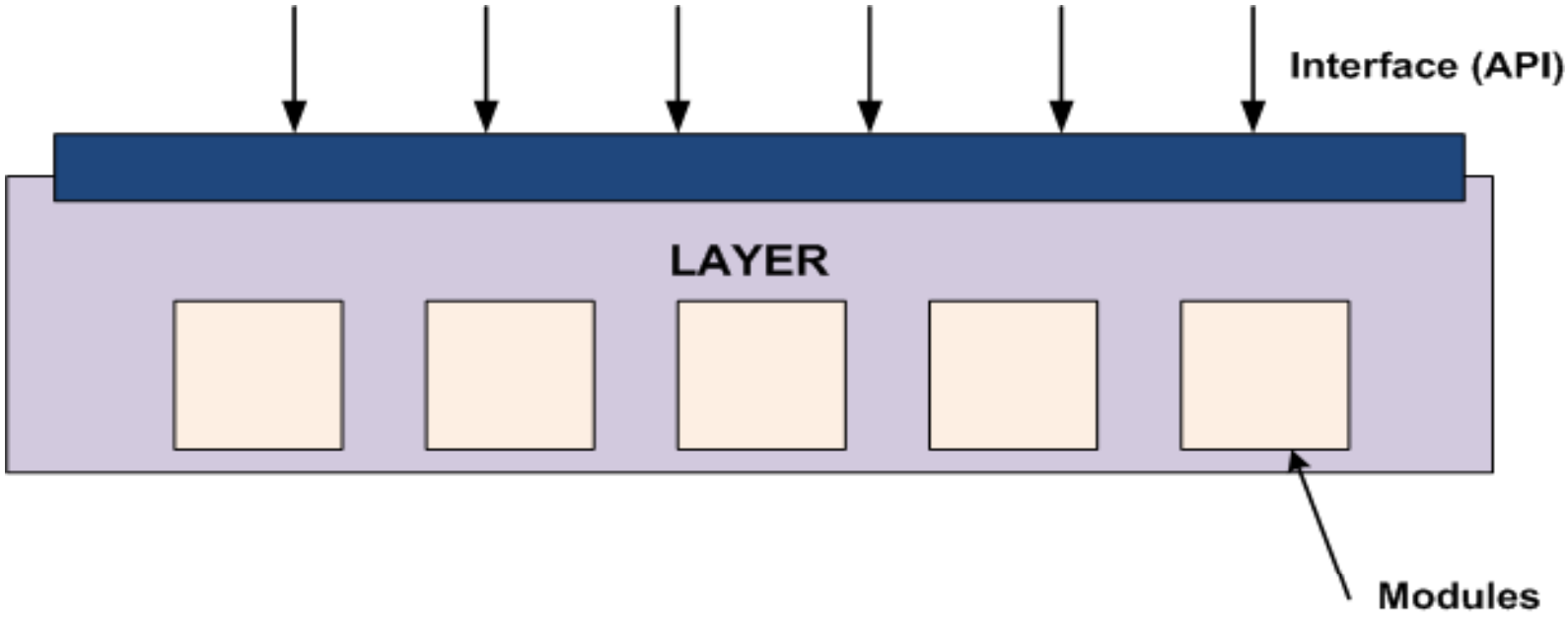


Layers Style

- The layered style, like all module styles, reflects a division of the software into units.
  - In this case, the units are **layers.**
- Each layer represents a grouping of modules that offers a cohesive set of services
- Allowed-to-use relation (specialization of depends-on)
- Topology
  - If layer A is above layer B, then layer B cannot be above layer A
    - Every piece of software is allocated to exactly one layer

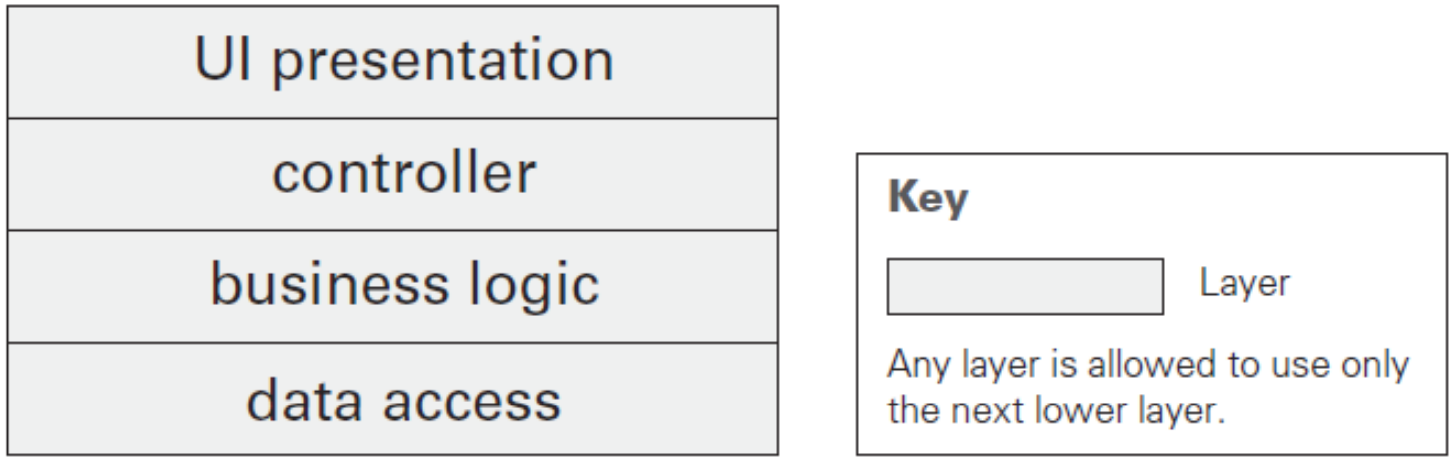
Layers View

- A layer is a collection of software units such as programs, or modules that may be invoked or accessed
- A basic requirement is that the units have an interface by which their services can be accessed



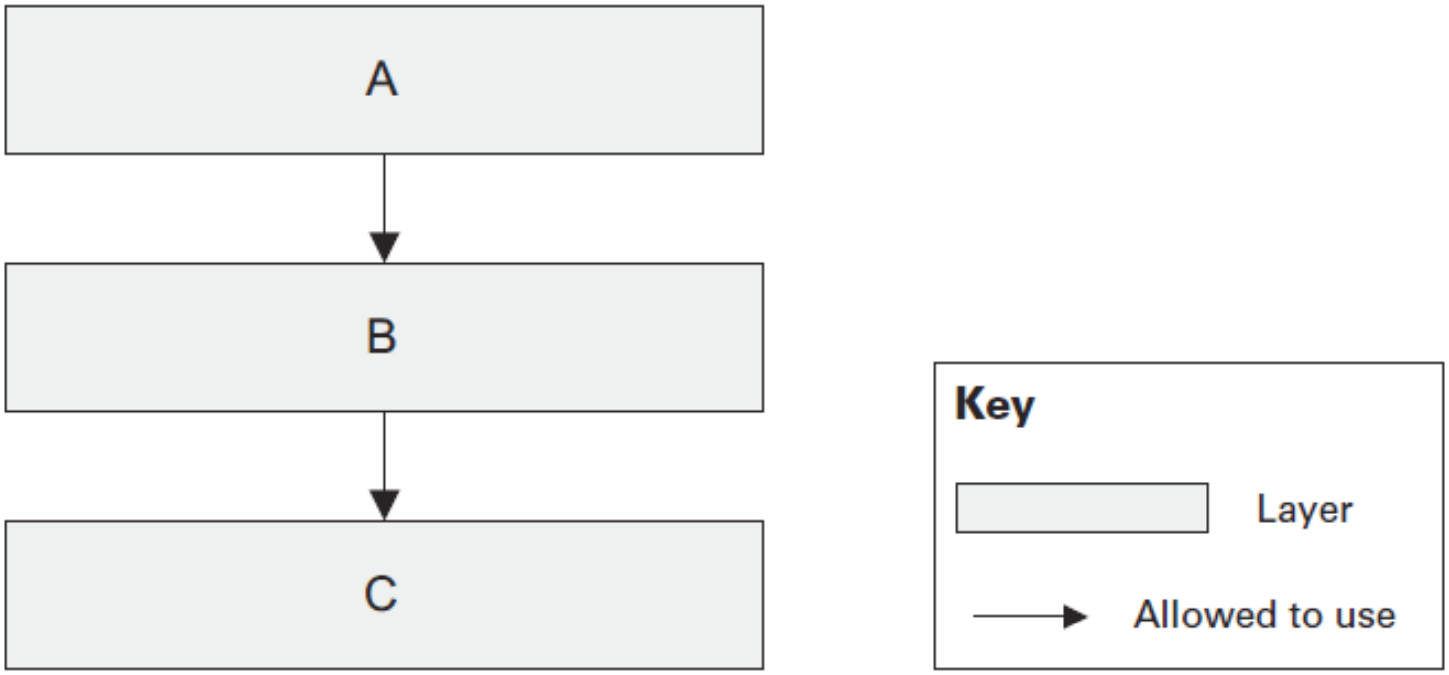
Notations for the Layered Style

- ***Stack***
  - Layers are almost always drawn as a stack of boxes. The *allowed-to-use relation is denoted by geometric adjacency and is read* from the top down



Notations for the Layered Style

- ***Segmented Layers***
  - Sometimes layers are divided into segments denoting a finer-grained aggregation of the modules. Often, this occurs when a preexisting set of units, such as imported modules, share the same *allowed-to-use relation*



Properties of Layering

- **Cohesion**
  - Layer provides a cohesive set of services, that are semantically related
  - In general, P1 and P2 should be in the same layer if they are likely to be ported to a new application together, or together they provide different aspects of the same virtual machine to usage community
- **Interface**
  - Layer provides a set of public interface facilities that may be invoked or accessed by other software
  - In cases where the layer is widely used across many kinds of systems and organizations, its interface may well be a public standard

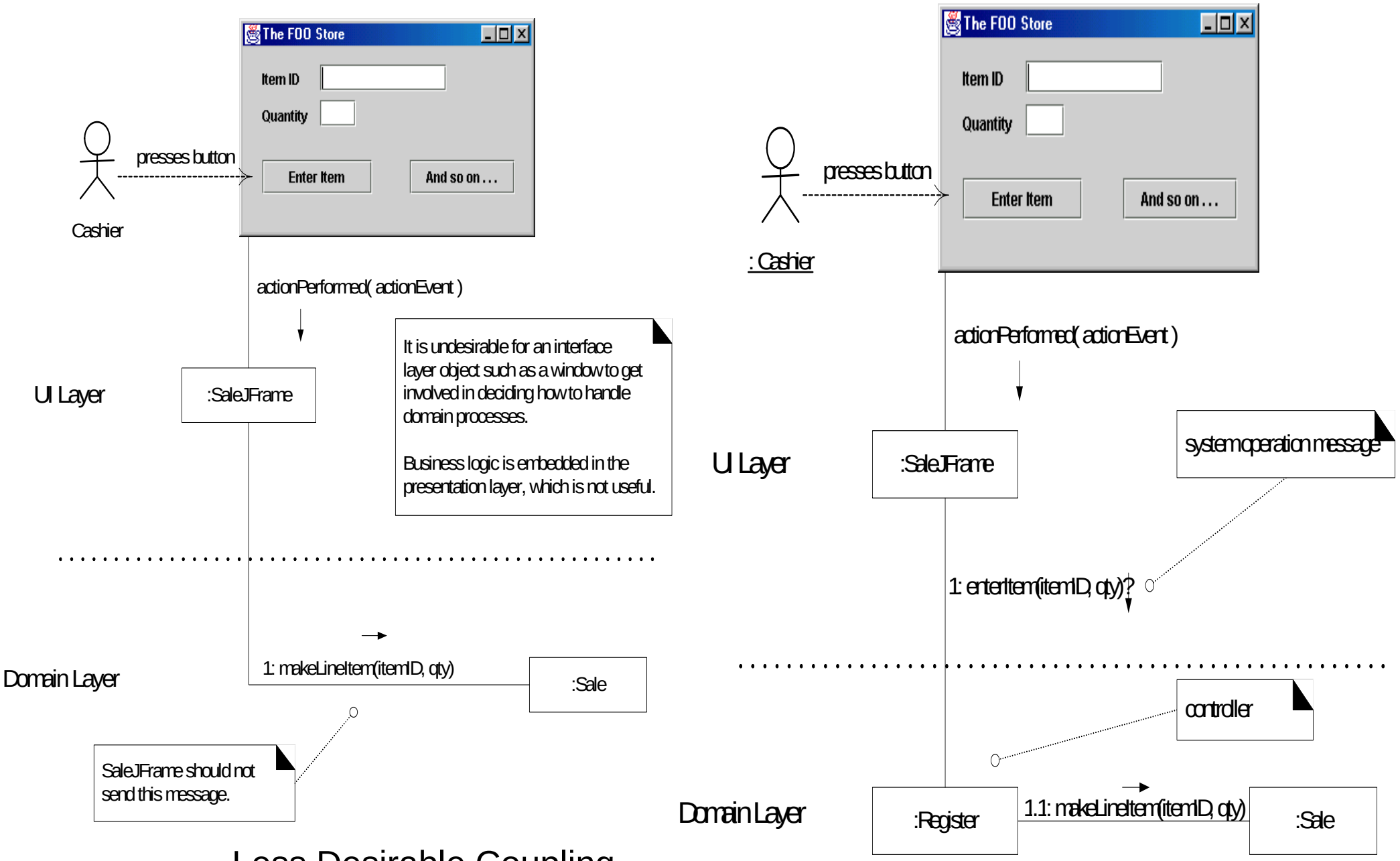
Properties of Layering

- **Coupling**
  - It is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements
  - An element with high coupling depends on many other elements
  - Problems because of a design with high coupling:
    - Changes in related elements force local changes.
    - Harder to understand; need to understand other elements.
    - Harder to reuse because it requires additional presence of other elements.

Properties of Layering – Low Coupling

- Some of the places where coupling occurs (for **class elements**):
  - Attributes: X has an attribute that refers to a Y instance.
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Subclasses: X is a subclass of Y.
  - Types: X implements interface Y.
- There is no specific measurement for coupling, but in general, elements that are generic and simple to reuse have low coupling.
- There will always be some coupling among elements (objects, modules, components), otherwise, there would be no collaboration.

Example: Coupling of UI layer to domain layer



Less Desirable Coupling

Properties of Layering – High Cohesion

- **Cohesion**
  - it is a measure of how strongly related and focused the responsibilities of an element are.
- An element with low cohesion does many unrelated activities or does too much work.
- Problems because of a design with low cohesion:
  - Hard to understand.
  - Hard to reuse.
  - Hard to maintain.
  - Affected by change.

Properties of Layering – High Cohesion

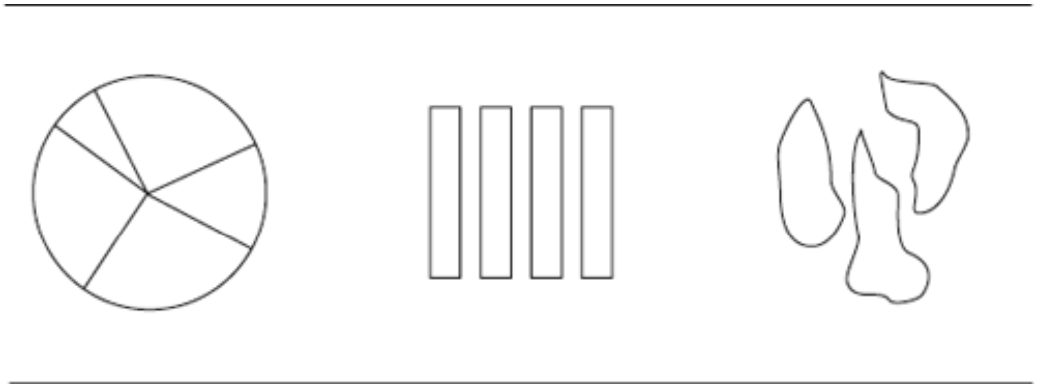
- Scenarios that illustrate varying degrees of functional cohesion

1. **Very low cohesion**: An element responsible for many things in many different areas (e.g., a class responsible for interfacing with a database and remote-procedure-calls.)
2. **Low cohesion:** An element responsible for complex task in a functional area (e.g., a class responsible for interacting with a relational database.)
3. **High cohesion:** An element has moderate responsibility in one functional area and it collaborates with other classes to fulfill a task (e.g., a class responsible for one section of interfacing with a database.)

- **Rule of thumb:** An element with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work. It collaborates and delegates.

Layers View

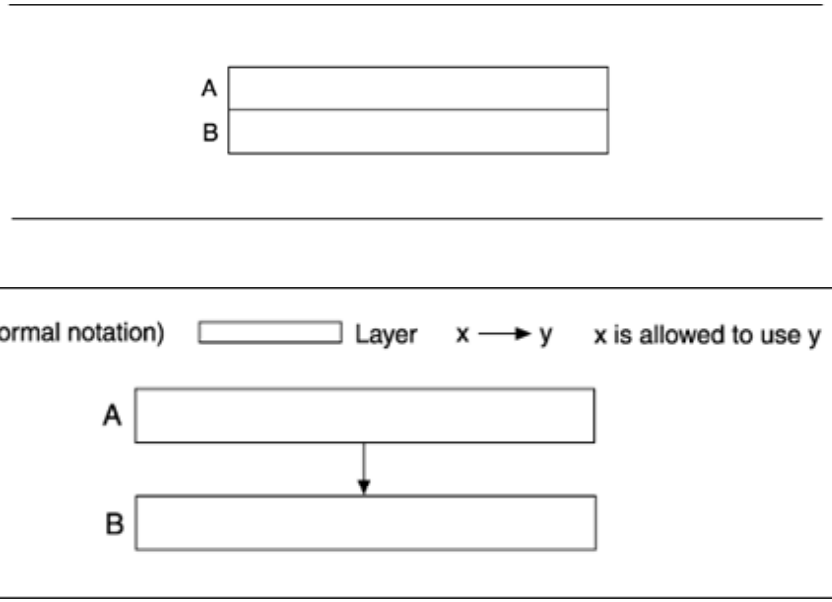
- Layers completely partition the software (classes/packages)
- Each partition constitutes a virtual machine (with a public interface) that provides a cohesive set of services
- Why are the below representations NOT layers?

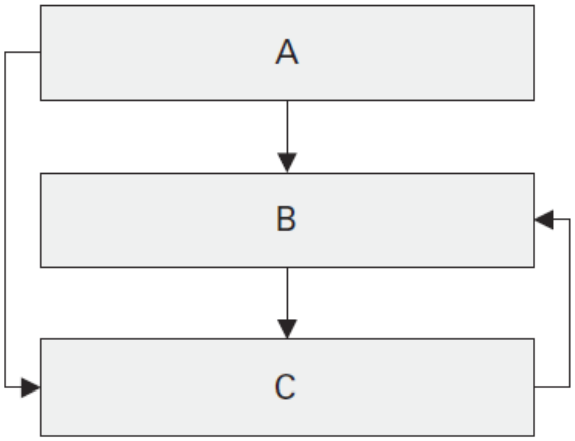


- Layering has one more fundamental property: **The layers are created to interact according to a strict ordering relation.**

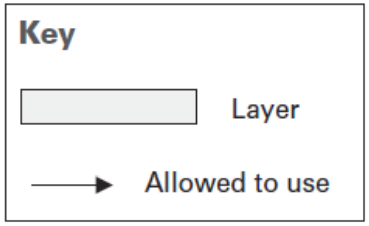
Layers View

- Layers are created to interact with each other according to a strict ordering relation:
  - If (A, B) is in this relation, we say "layer B is beneath layer A", that means either, or both, of the following:
    - "The implementation of layer A is allowed to use any of the public facilities of the virtual machine provided by layer B"
  - Systems with upward usages are not, strictly according to the definition, layered.



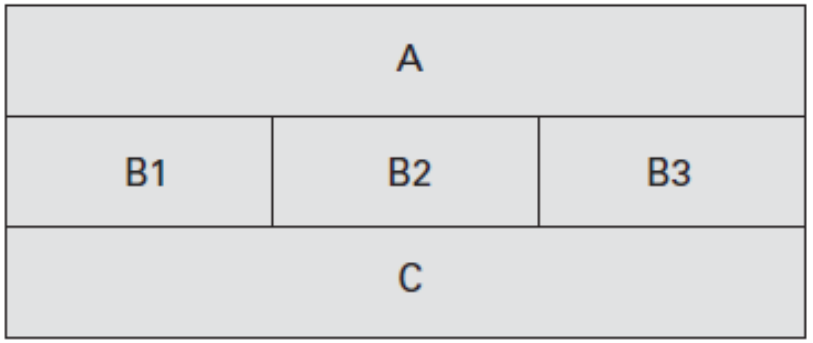


NOT a layered system

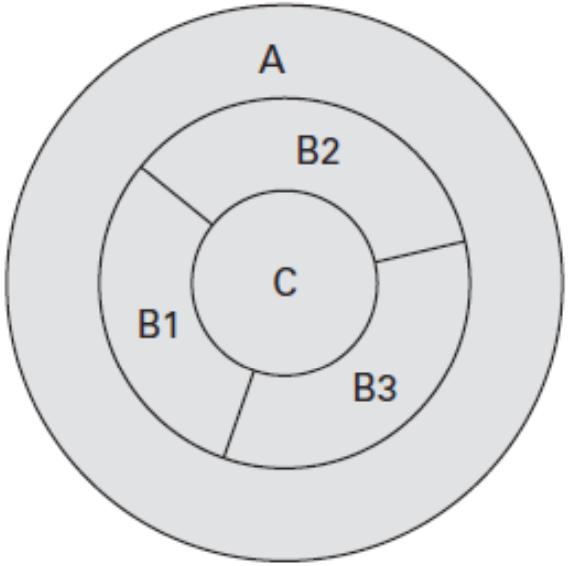


How to Model Layering

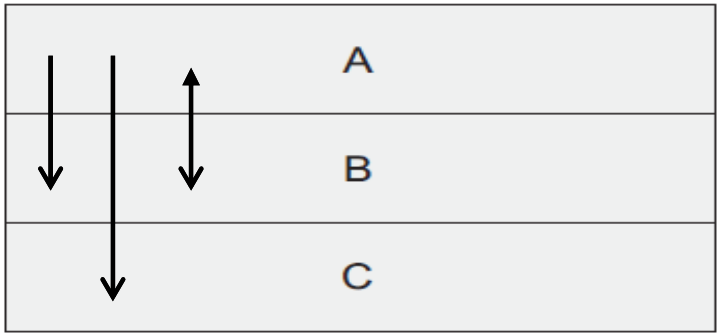
- **Stack**
  - The allowed-to-use relation is denoted by geometric adjacency, or sometimes by an arrow



- **Ring**
  - The innermost ring corresponds to the lower layer
  - The outermost ring corresponds to the highest layer
  - A ring may be subdivided into sectors



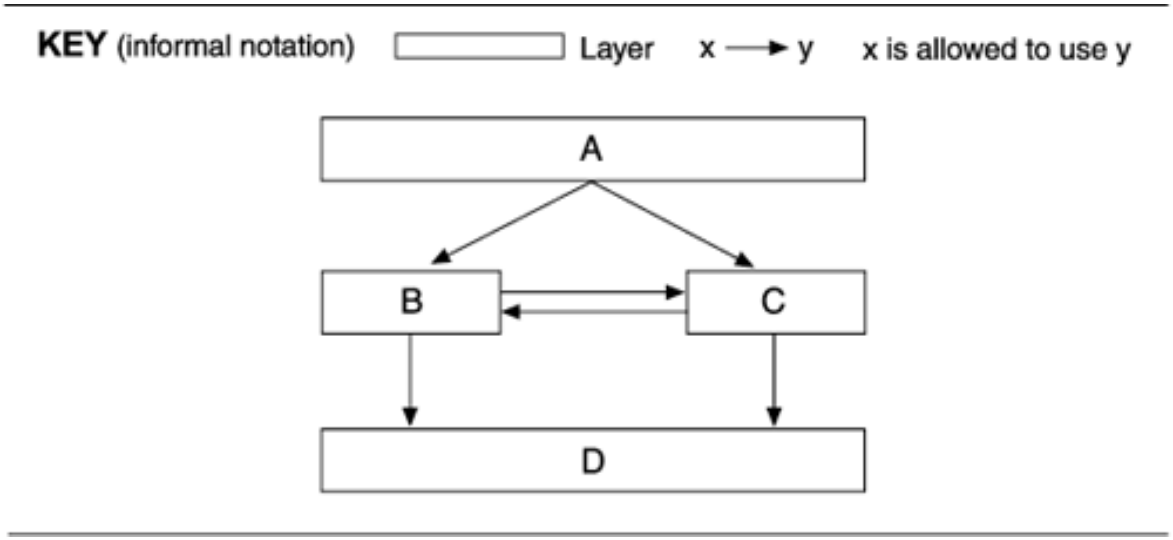
Communication Direction in Layered Styles



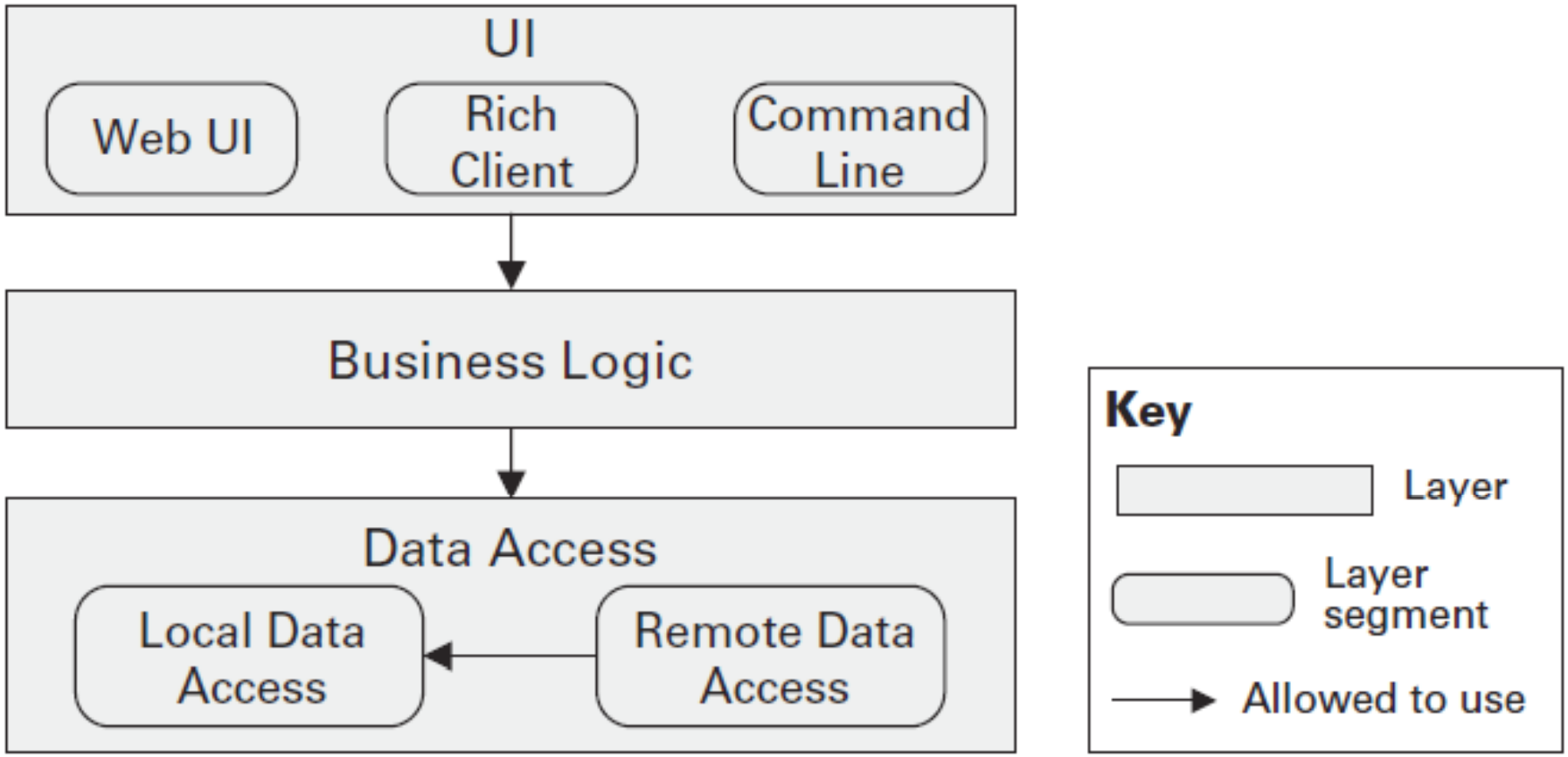
- **Communication**
  - Defining the allows-to-use relation
  - These conventions allow software in a layer to use software in
    - The closest lower layer
    - Any lower layer
    - Any adjacent layer, above or below (not a layer in strict sense)
  - The case of software in a higher layer using modules in a lower layer, BUT not the next lower layer is called **layer bridging**
  - If many of these cases are present, the system is poorly structured; in particular wrt portability and modifiability goals

Layers - Segmented

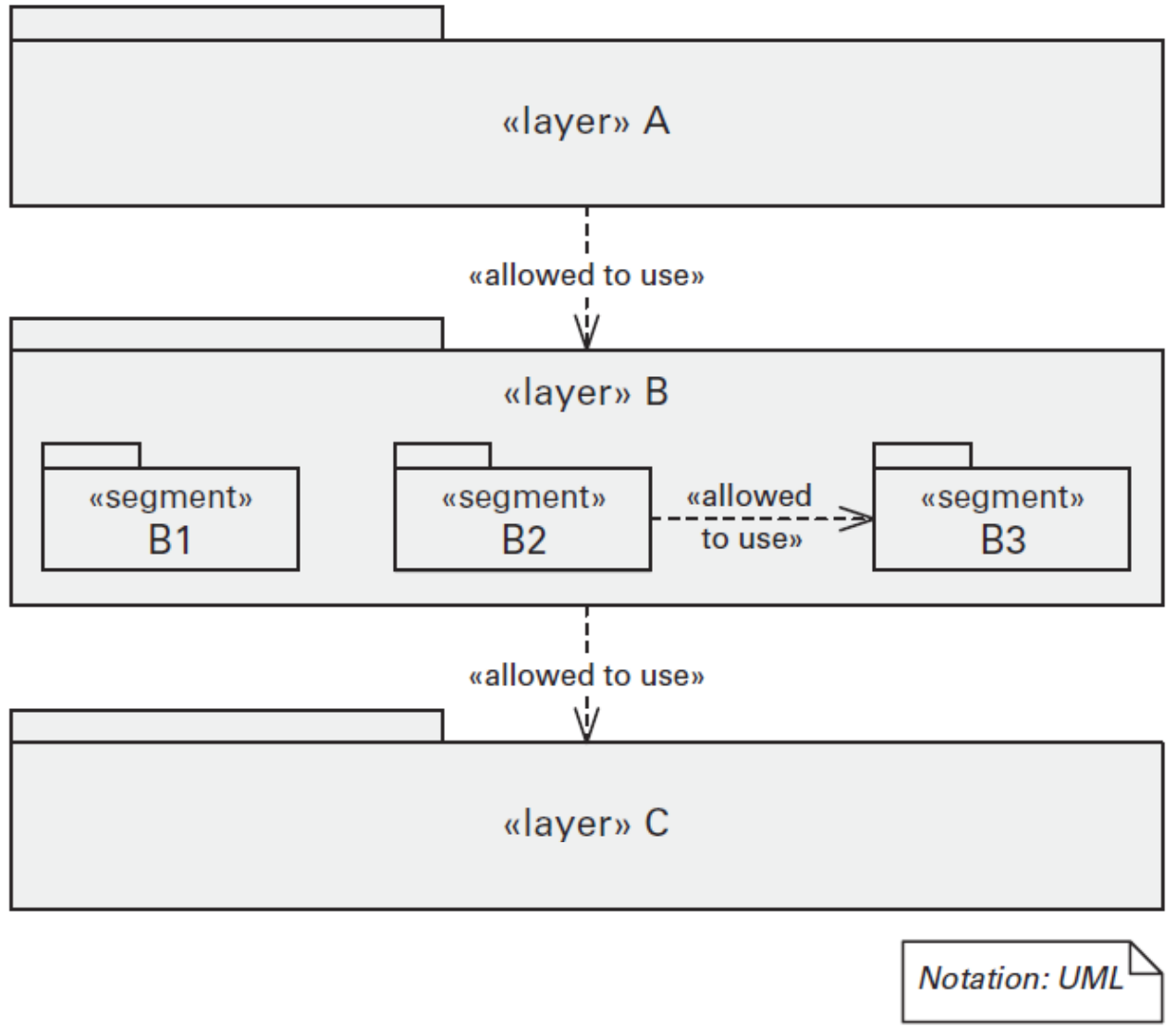
- Layers are divided into segments denoting some finer-grained decomposition of the software
- Example:
  - A is allowed to use B and C
  - B and C are allowed to use each other, and D



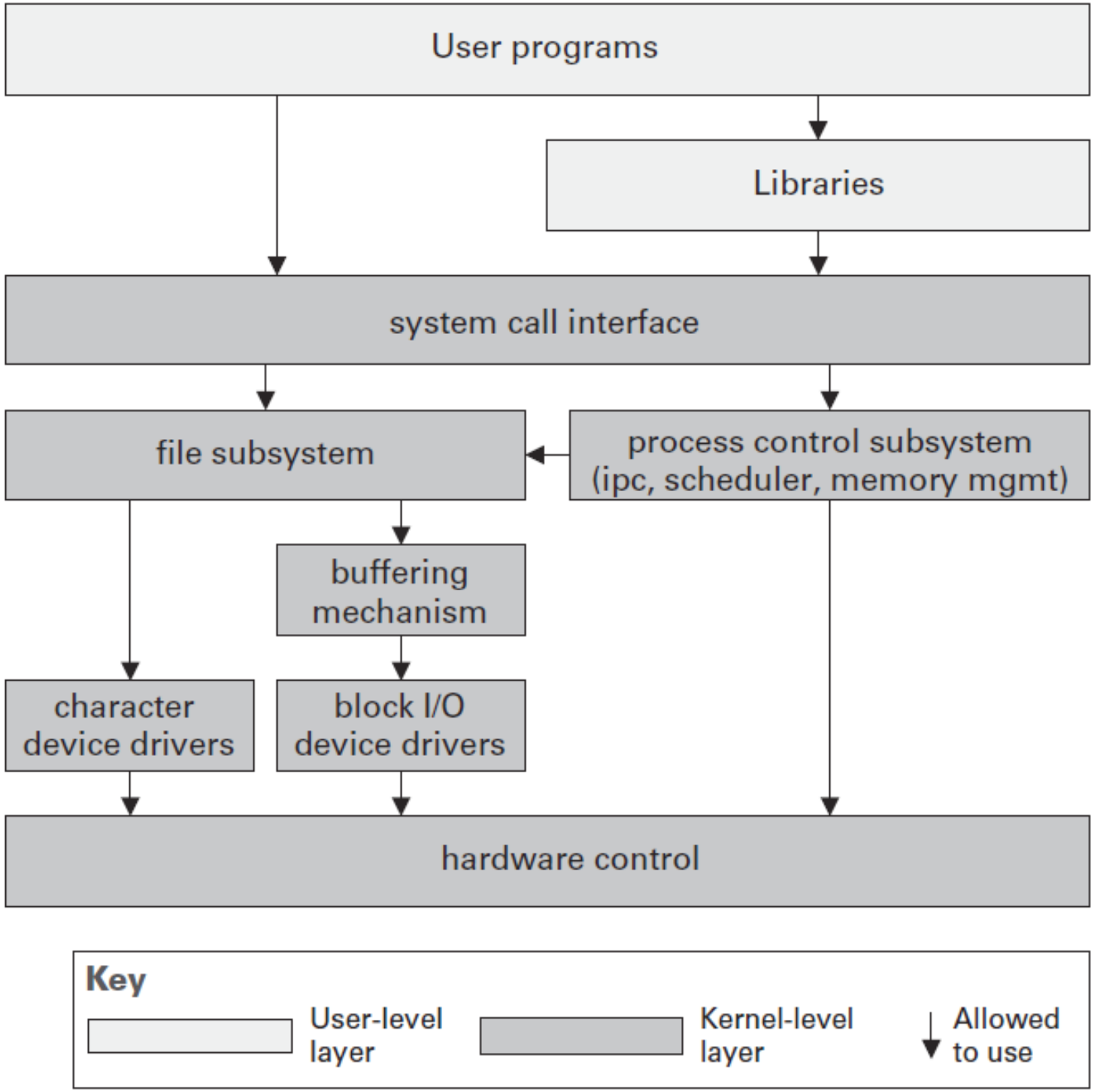
Example – Segmented Layer



Example – Segmented Layer (in UML)



Example – UNIX System OS

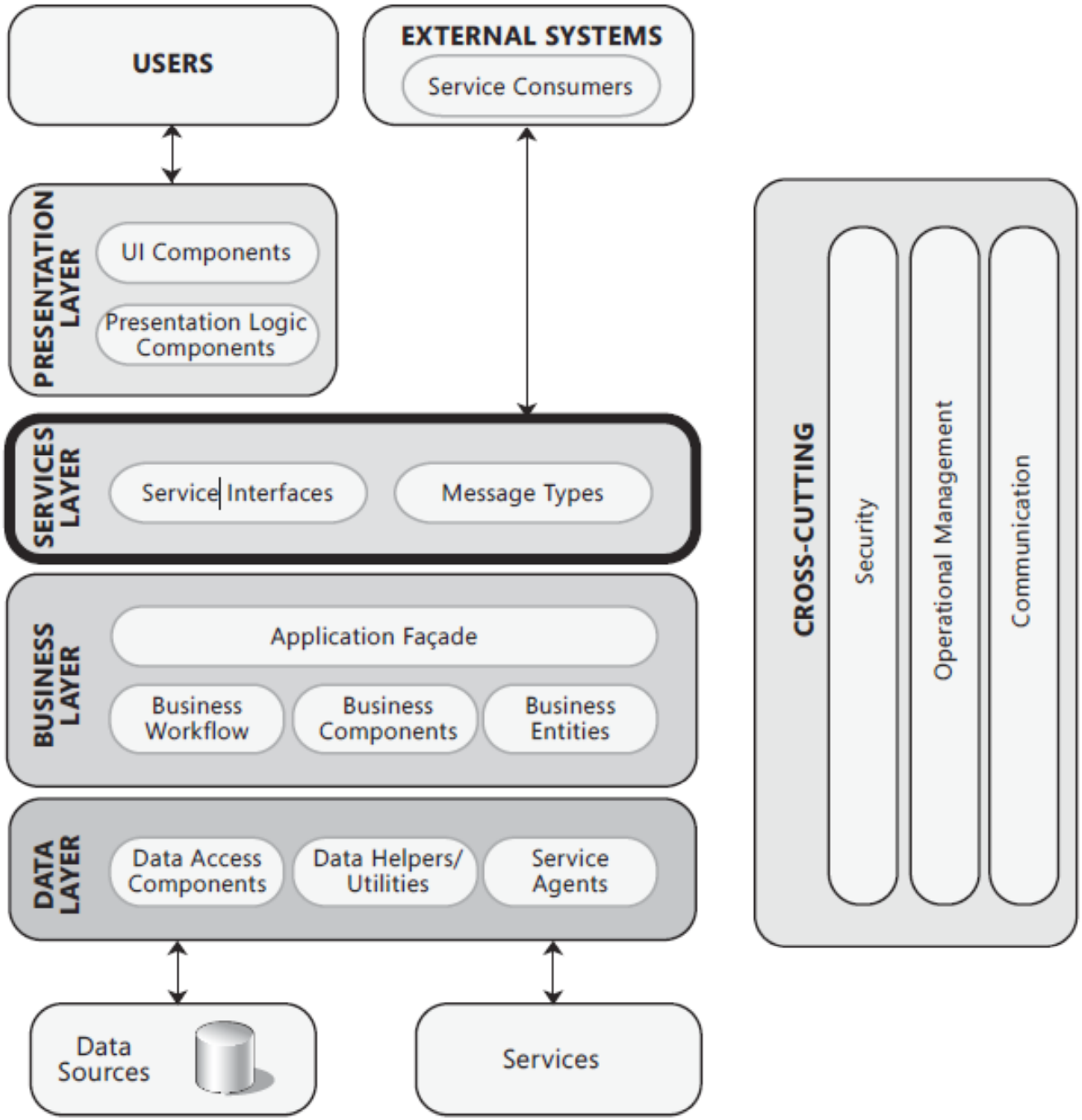


Microsoft's Application Architecture Types

- Key Architectural Styles defined in MS

Architecture style	Description
Client/Server	Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.
Domain Driven Design	An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message Bus	An architecture style that prescribes use of a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other.
N-Tier / 3-Tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Service-Oriented Architecture (SOA)	Refers to applications that expose and consume functionality as a service using contracts and messages.

Example: Service-Enabled Layered Architecture (Microsoft)



Layers vs. Tiers

- Layers are NOT tiers..
- Tiers represent locality of processing (hardware nodes, machines on which the software runs)
- Layers are cohesive set of modules for which a common interface is provided
- Layers might, however, be mapped to tiers

Module Viewtype - Summary

- **Is for..**
  - Construction
    - Module view provides a blueprint for the source code
  - Analysis
    - Modules implement the functional requirements of the system
  - Communication
    - Module view can be used to explain functionality of the system
  - Education of new developers
- **Is NOT for..**
  - Run-time behavior analysis (e.g., performance analysis)

Relation to Other Viewtypes

- Module views are commonly **mapped to** views in the **component-and-connector** viewtype
- Implementation units in module views have a mapping to components that executes at run-time
  - Sometimes quite simple one-to-one
  - Or more complex N-M relation

Viewtypes..

- The module viewtype
- **The component-and-connector viewtype**
- The allocation viewtype

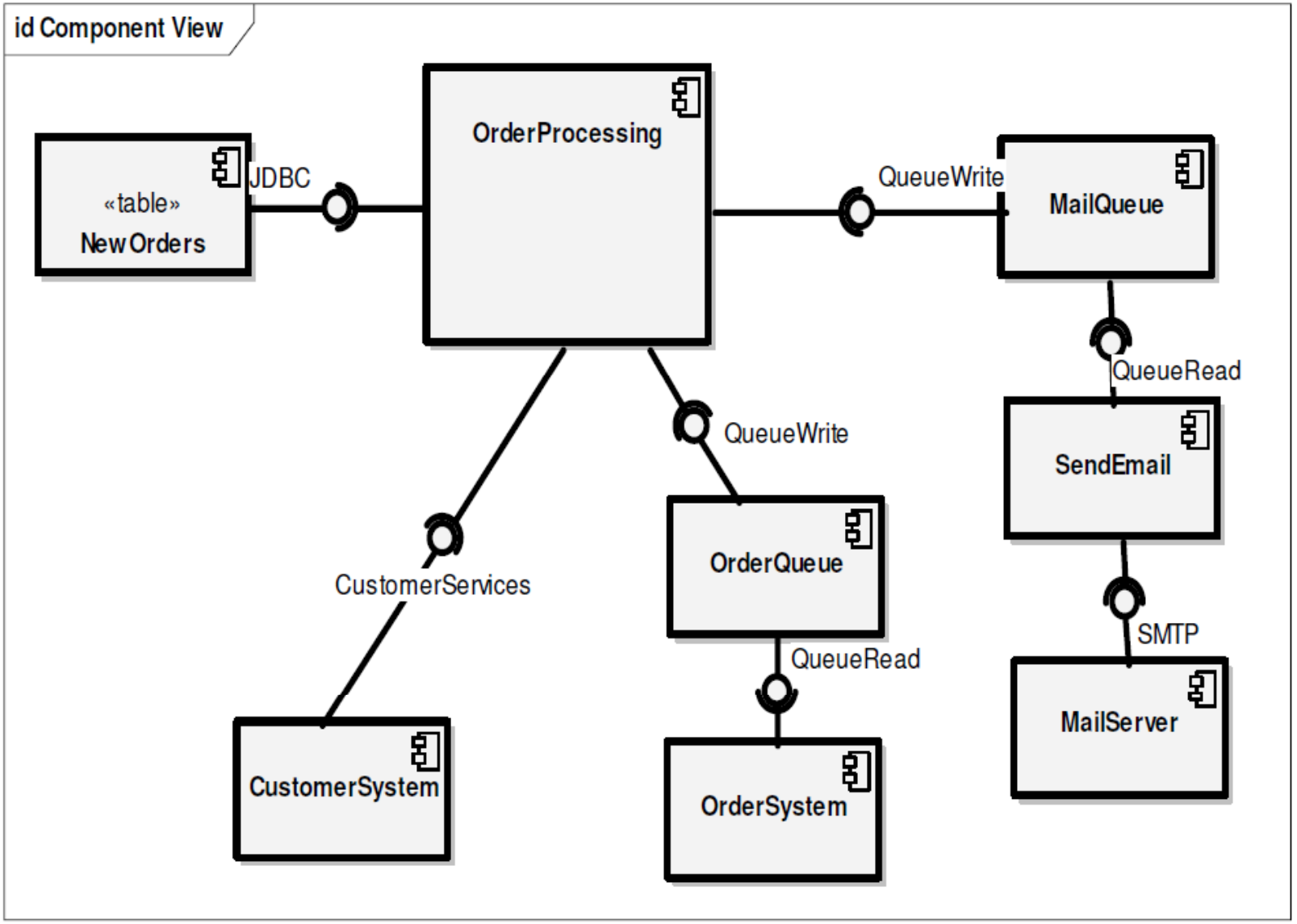
Component-And-Connector (C&C) Viewtype

- Document the system's units of execution
- **Elements:**
  - **Component:** principle processing unit of the executing system (objects, processes, or collections of objects, etc.)
  - **Connector:** interaction mechanism for the components (pipes, repositories, sockets, middleware, etc.)
- **Relations:**
  - **Attachments:** component ports are associated with specific connector roles
  - A component port **p** is attached to a connector role **r**, if the component interacts over the connector, using the interface described by **p**, and conforming to the expectations described by **r**

Component-And-Connector (C&C) Viewtype

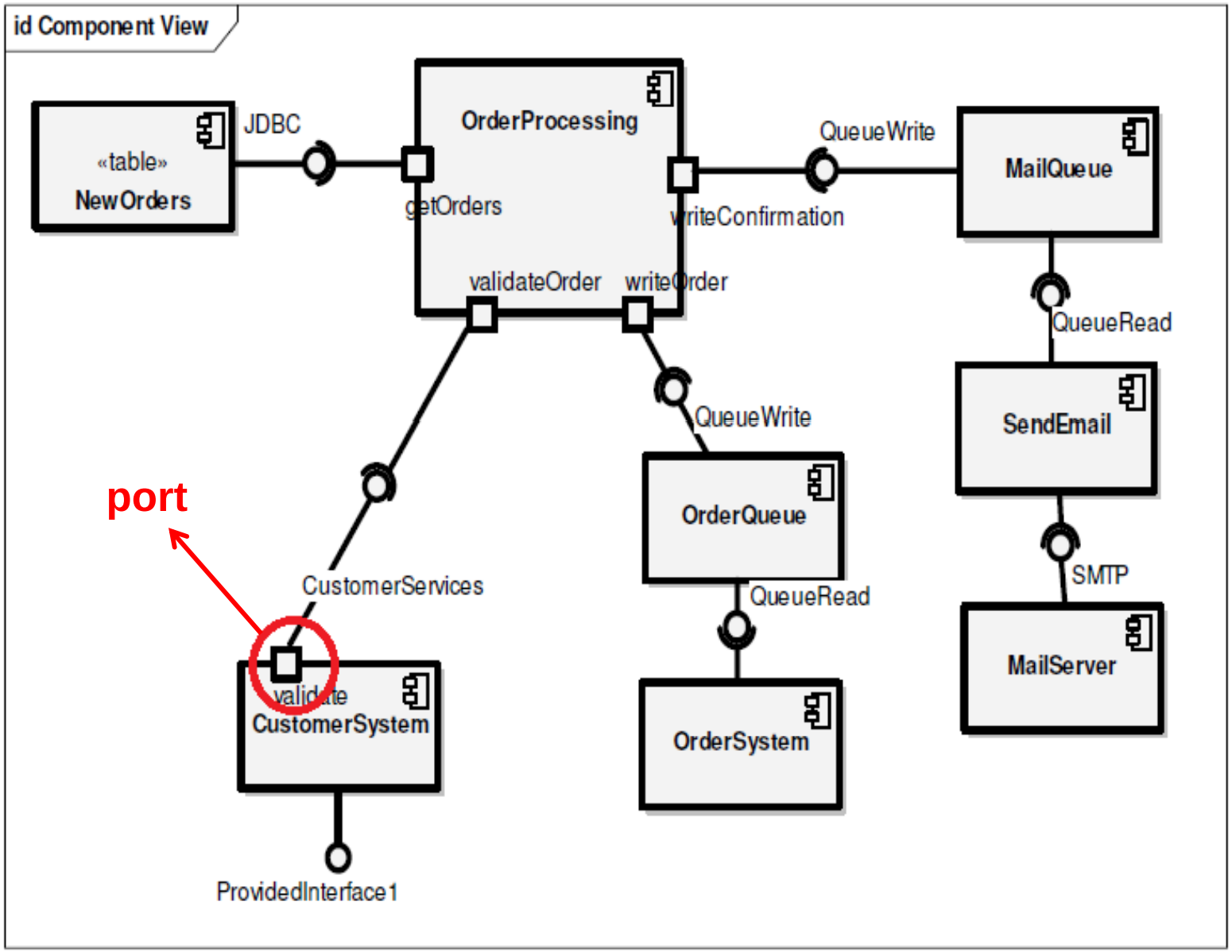
- Components have
  - **Provides interface**

    and
  - **Required interface**



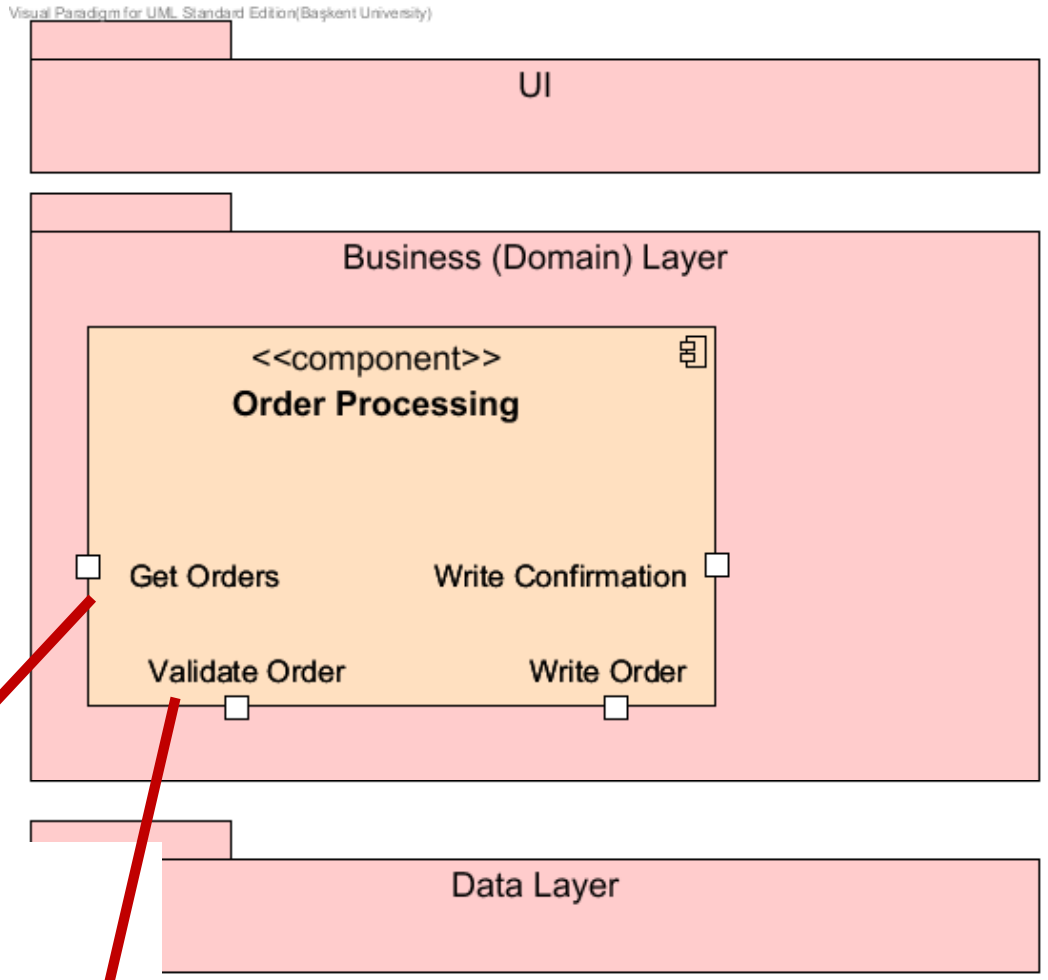
Component-And-Connector (C&C) Viewtype

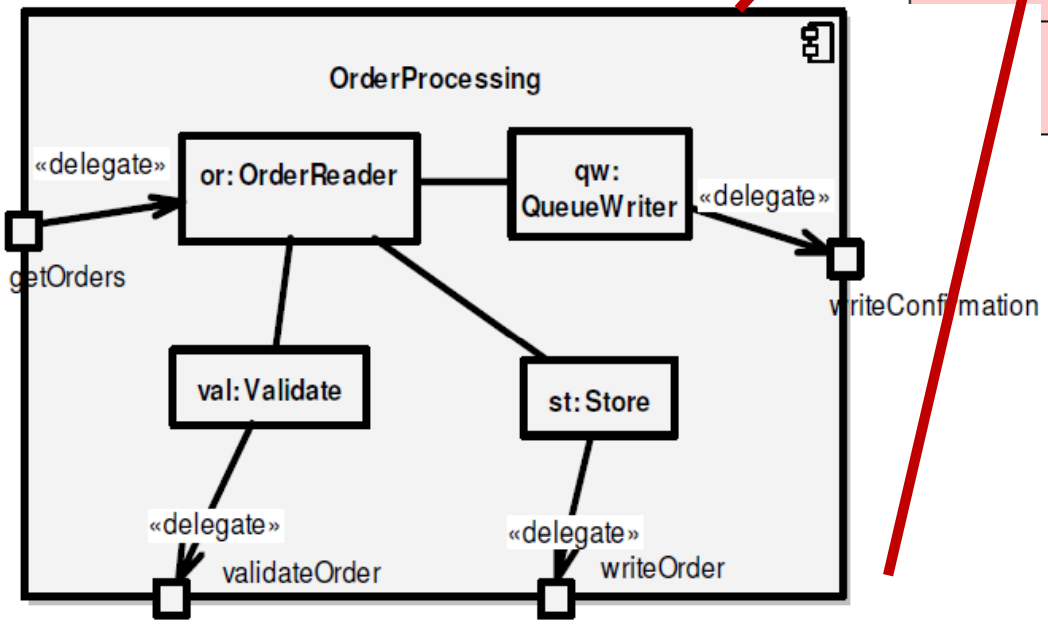
- Components have interfaces called **ports**
  - A port defines a specific point of potential interaction of a component with its environment



Place of Component Diagram

- **Example**
  - System is decomposed into layers
  - Layers consists of subsystems (packages)
  - Subsystem is decomposed into a collection of interacting components, represented in a component diagram (UML)



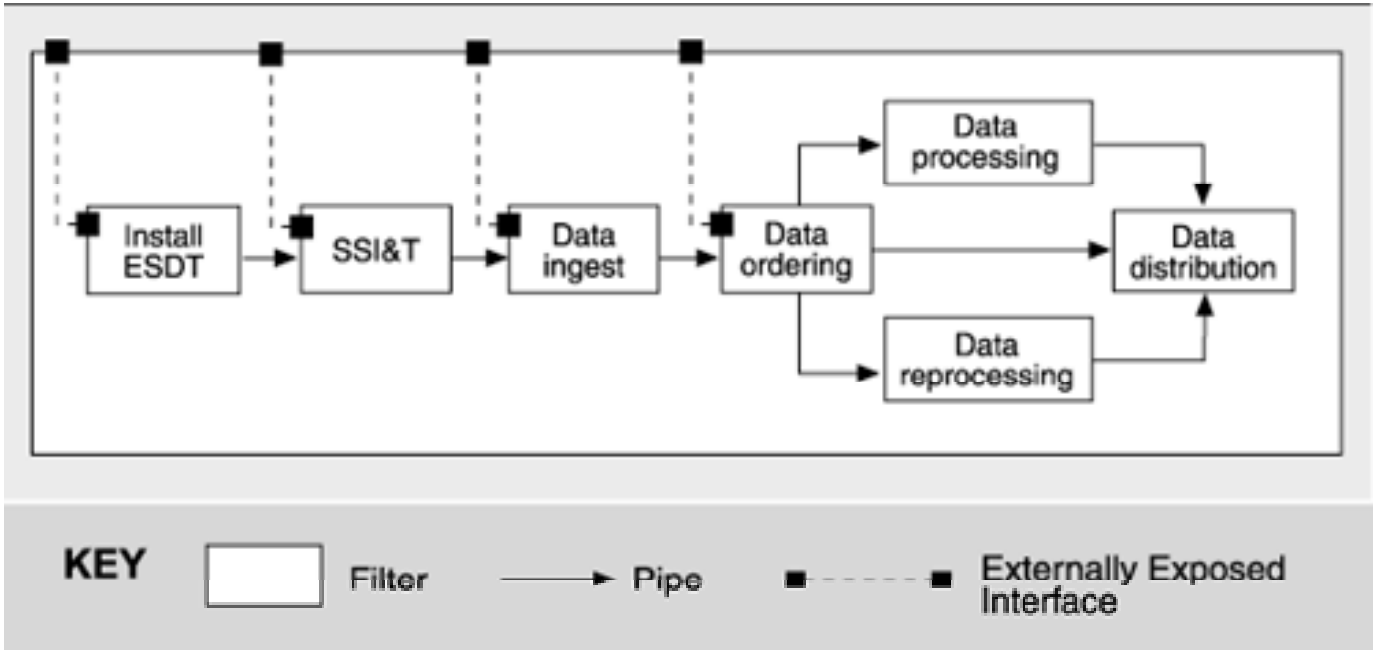


C&C Viewtype Styles

- Pipe-and-Filter
- Shared Data
- Publish Subscribe
- Client-Server
- Peer-to-Peer
- Communicating Processes

C&C Pipe and Filter Style – Example

- **Elements**
  - Component Types: **filters,** data transformers that read streams of data from input ports and write streams of data to output ports
  - Connector Types: **pipe,** convey streams of data from one filter to another
- **Relations:** Attachment relation
- **Topology:** Pipes connect filter output ports to filter input ports



C&C Pipe and Filter Style – Advice

Typical properties to document for pipes include

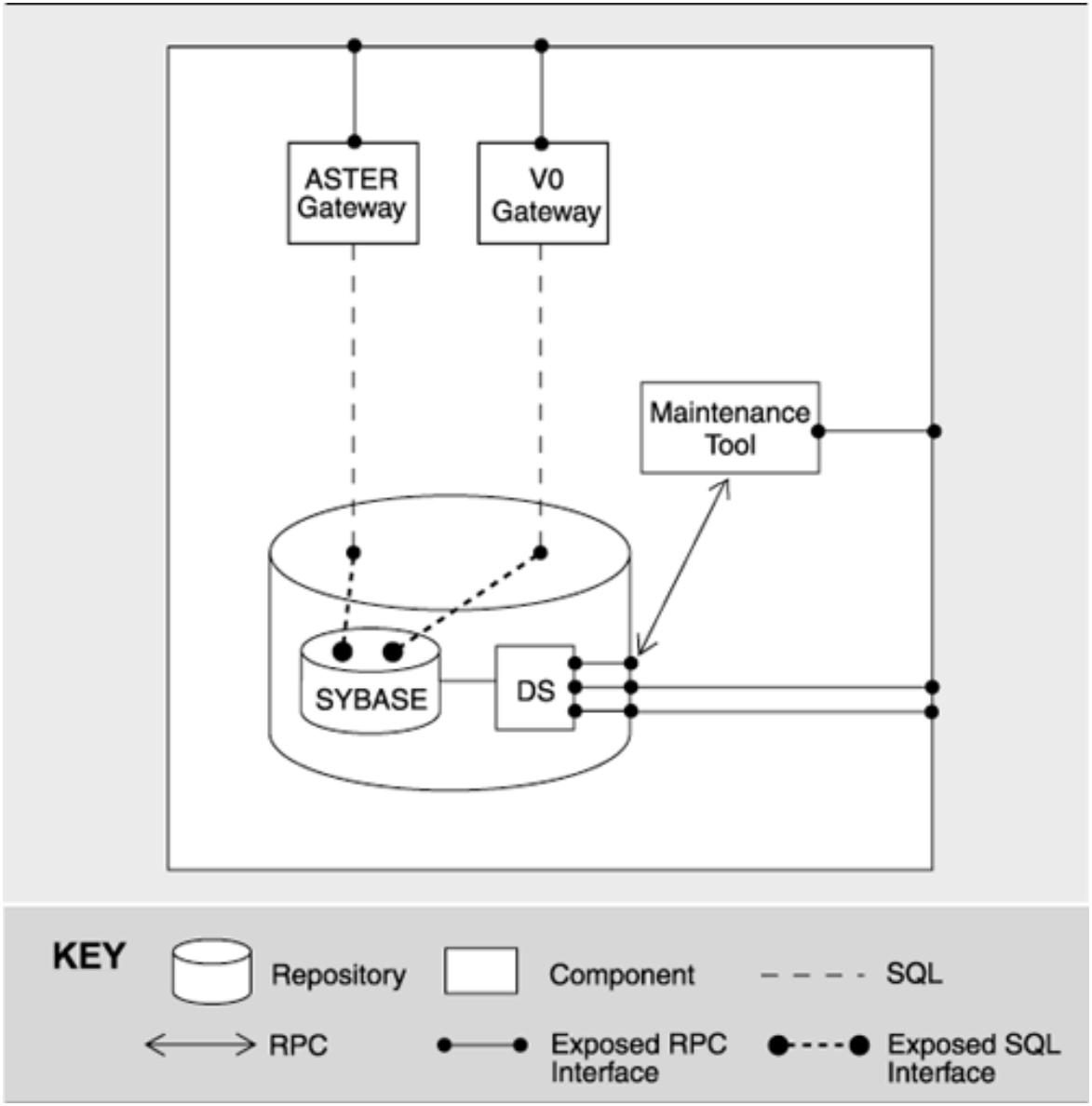
- Pipe capacity (that is, buffer size)
- How end-of-data is signaled
- What form of blocking occurs when writing to a pipe whose buffer is full or reading from a pipe that is empty

Properties of filters can include

- Whether or not each filter is a separate process
- The data stream transformation each performs

C&C Shared Data Style – Example

- **Elements**
  - Component Types: Shared data repositories and data accessors
  - Connector Types: Data reading and writing
- **Relations:** Attachment relation determines which data accessors are connected to which data repositories
- **Topology:** Data accessors are attached to connectors that are attached to data store(s)



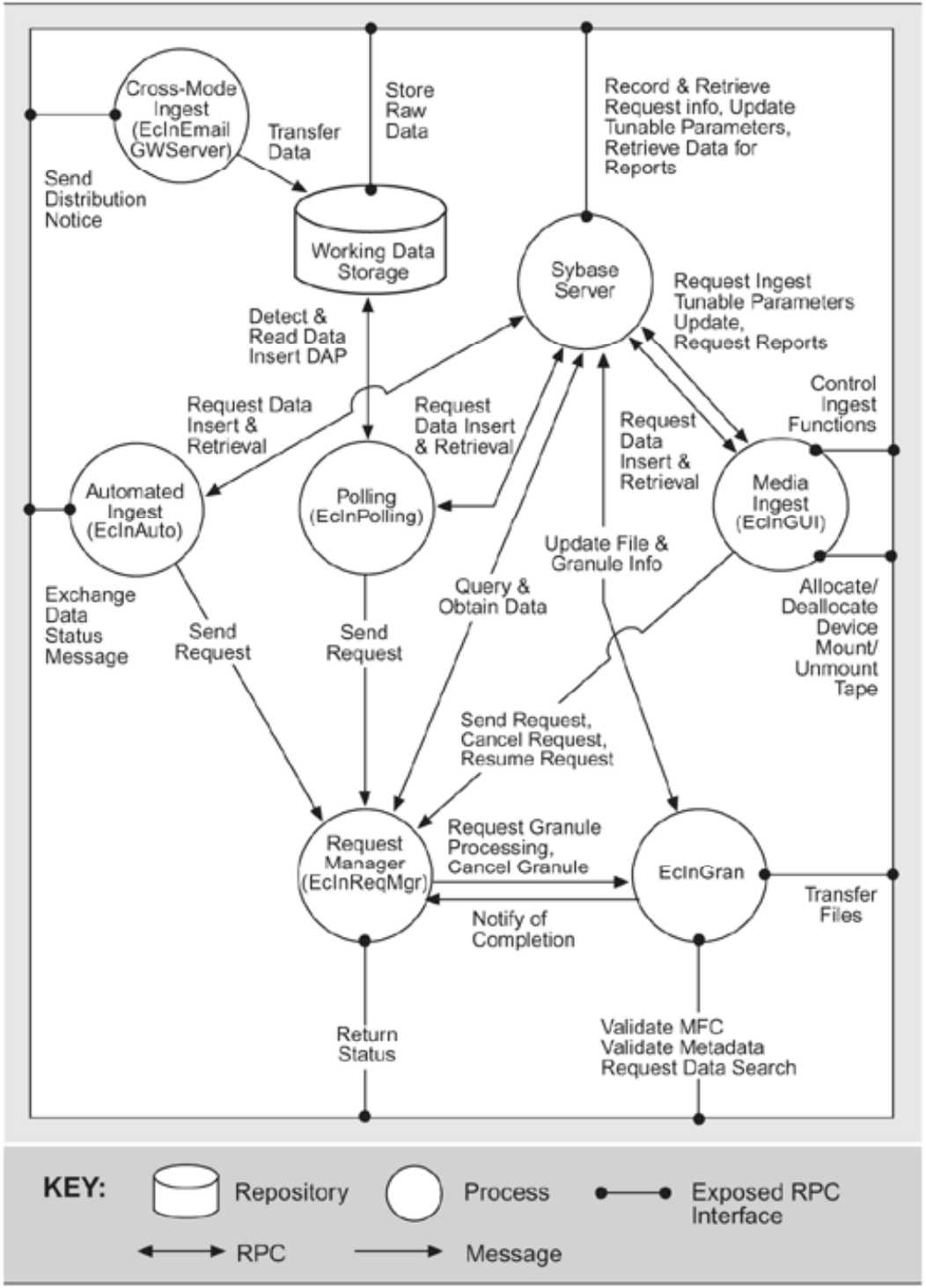
C&C Shared Data Style – Advice

Useful properties to document about data stores include the following:

- Restrictions on the number of simultaneous connections to the data store.
- Whether or not new accessors can be added at runtime.
- Access-control enforcement policies.
- Whether concurrent access to the same data element is permitted, and if so, what kinds of synchronization mechanisms are used.
- Administrative concerns, such as whether one modifies the types of data stored, and if so, who has access, when those changes can be performed, and via what interface.
- Replication of data in a distributed setting.
- Age of data.
- If the repository system supports both query-based and triggered modes of interaction, it is important to clearly document what form of interaction is intended, for example, by using different connector types.

C&C Communicating Processes Style – Example

- **Elements**
  - Component Types: Concurrent units, such as tasks, processes and threads
  - Connector Types: Data exchange, message passing, synchronization, control and other types of communication
- **Relations:** Attachment relation as defined in C&C viewtype
- **Topology:** Arbitrary graphs



What the C&C Viewtype is for?

- Used to **reason about runtime system quality attributes,** such as performance, reliability, and availability.
- NOT appropriate for representing design elements that do not have a runtime presence

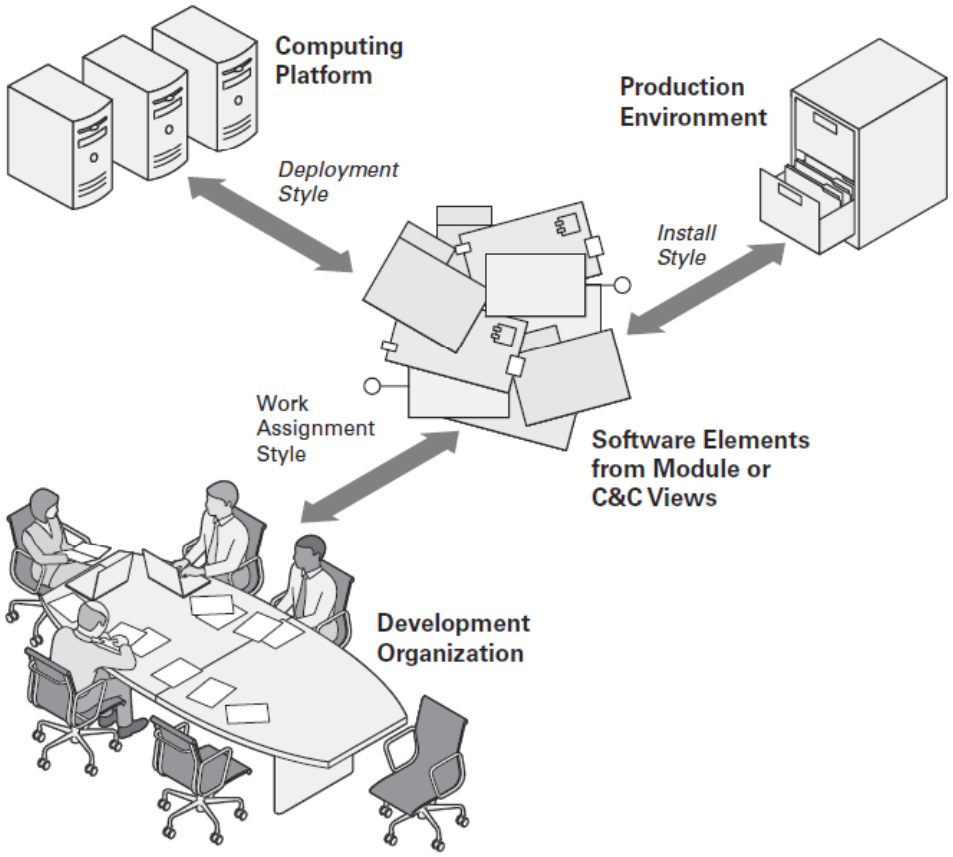
Allocation Viewtype

- The allocation viewtype presents a mapping of the software architecture onto its environment
  - Elements: Software element and environmental element
  - Relations: **Allocated-to**. A software element allocated to an environmental element

- Environment
  - Hardware elements (processors, datastores, etc.)
  - File management
  - Organization team (person)

Allocation Viewtype Styles

- The **deployment style** describes the mapping of the components and connectors onto the **hardware** on which the SW executes
- The **implementation (install) style** describes the mapping of modules onto a **file system** that contains these modules
- The **work assignment style** describes the mapping of modules onto the **people,** groups, or team tasked with the development of the modules

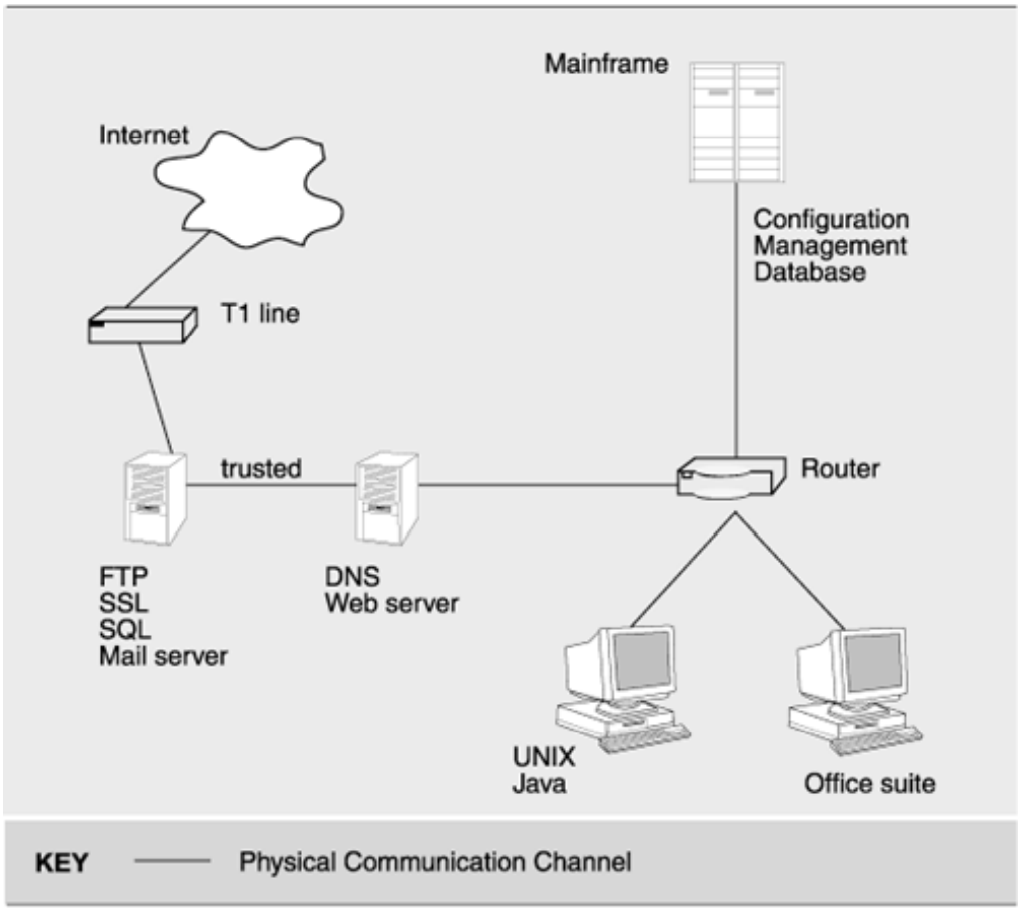


Allocation Viewtype – Deployment View

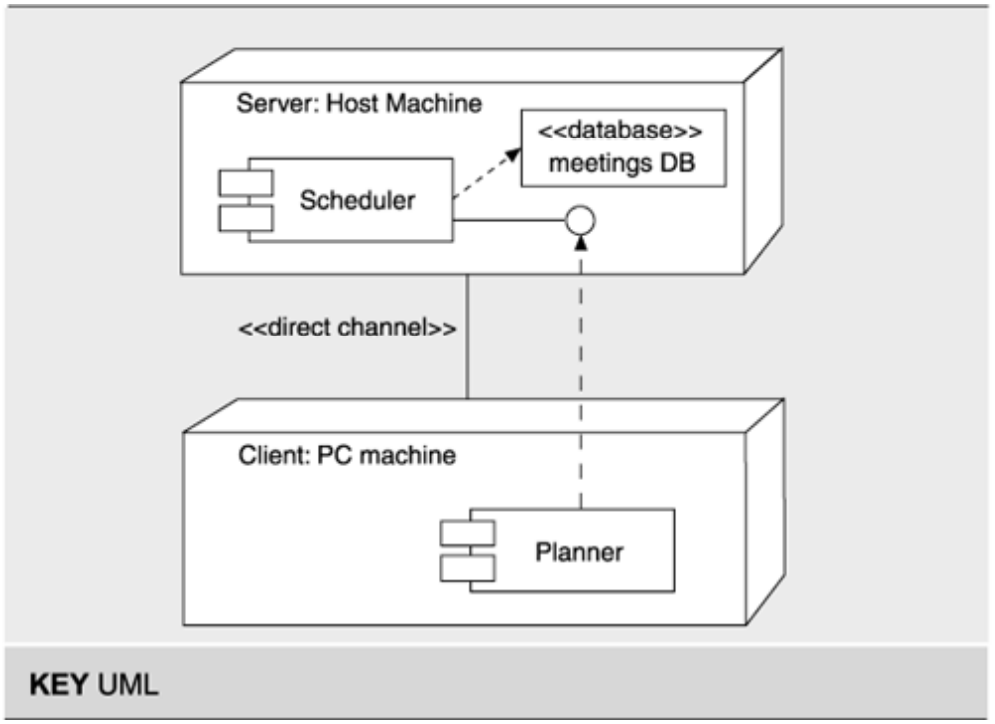
- **Element**
  - SW Element: usually elements from the C&C viewtype
  - Environmental Element: computing hardware – processor, memory, disk, network, and so forth.
- **Relations**
  - Allocated-To: Showing on which physical elements the SW resides
  - Migrates-To: If the allocation is dynamic
- **Topology**
  - Unrestricted
  - Allocated-relation either dynamic or static

Deployment View – Examples

A. Informal



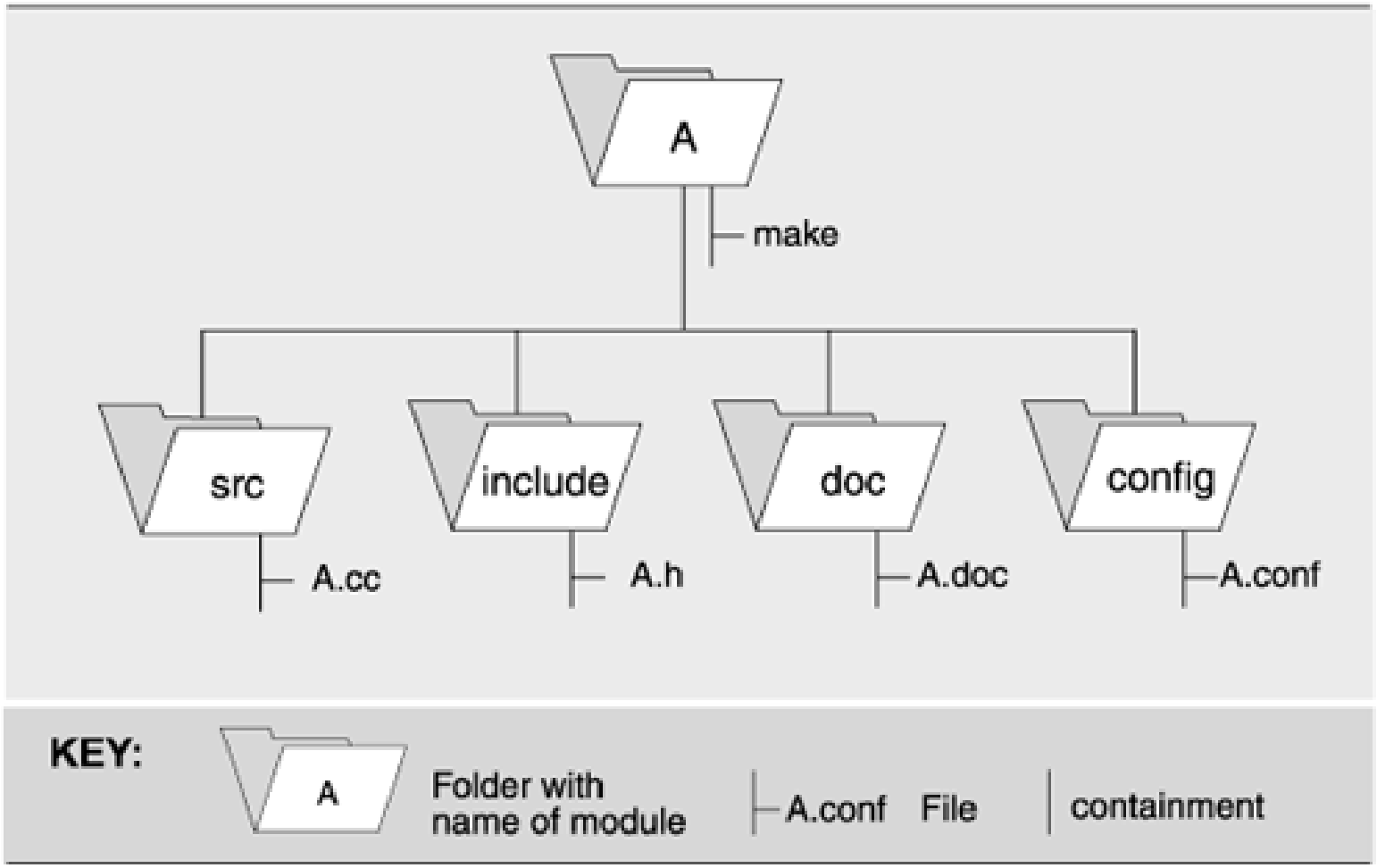
B. In UML



Allocation Viewtype – Implementation View

- **Element**
  - SW Element: A module
  - Environmental Element: A configuration item, such as file or directory
- **Relations**
  - Allocated-To: Describes the allocation of a module to a configuration item
  - Containment: Showing that, one configuration item is contained in another
- **Topology**
  - Usually hierarchical configuration

Implementation View – Example



Allocation Viewtype – Work Assignment View

- **Element**
  - SW Element: A module
  - Environmental Element: An organizational unit, such as a person, a team, a department, and so forth.
- **Relations**
  - Allocated-To: Describes the allocation of a module to a configuration item
- **Topology**
  - In general, unrestricted..
  - In practice, restricted so that, one module is allocated to one organizational unit

Work Assignment View - Example

ECS Element (Module)		
Segment	**Subsystem**	**Organizational Unit**
Science Data Processing Segment (SDPS)	Client	Science team
	Interoperability	Prime contractor team 1
	Ingest	Prime contractor team 2
	Data Management	Data team
	Data Processing	Data team
	Data Server	Data team
	Planning	Orbital vehicle team
Communica-tions and System Management Segment (CSMS)	System Management	Intrastructure team
	Communications	
	Internetworking	
Flight Operations Segment (FOS)	Planning and Scheduling	Orbital vehicle team
	Data Management	Database team
	Command Management	Orbital vehicle team
	Commanding	Orbital vehicle team
	Resource Management	Prime contractor team 3
	Telemetry	Orbital vehicle team
	User Interface	User interface team
	Analysis	Orbital vehicle team

Work Assignment View – Example (in UML)