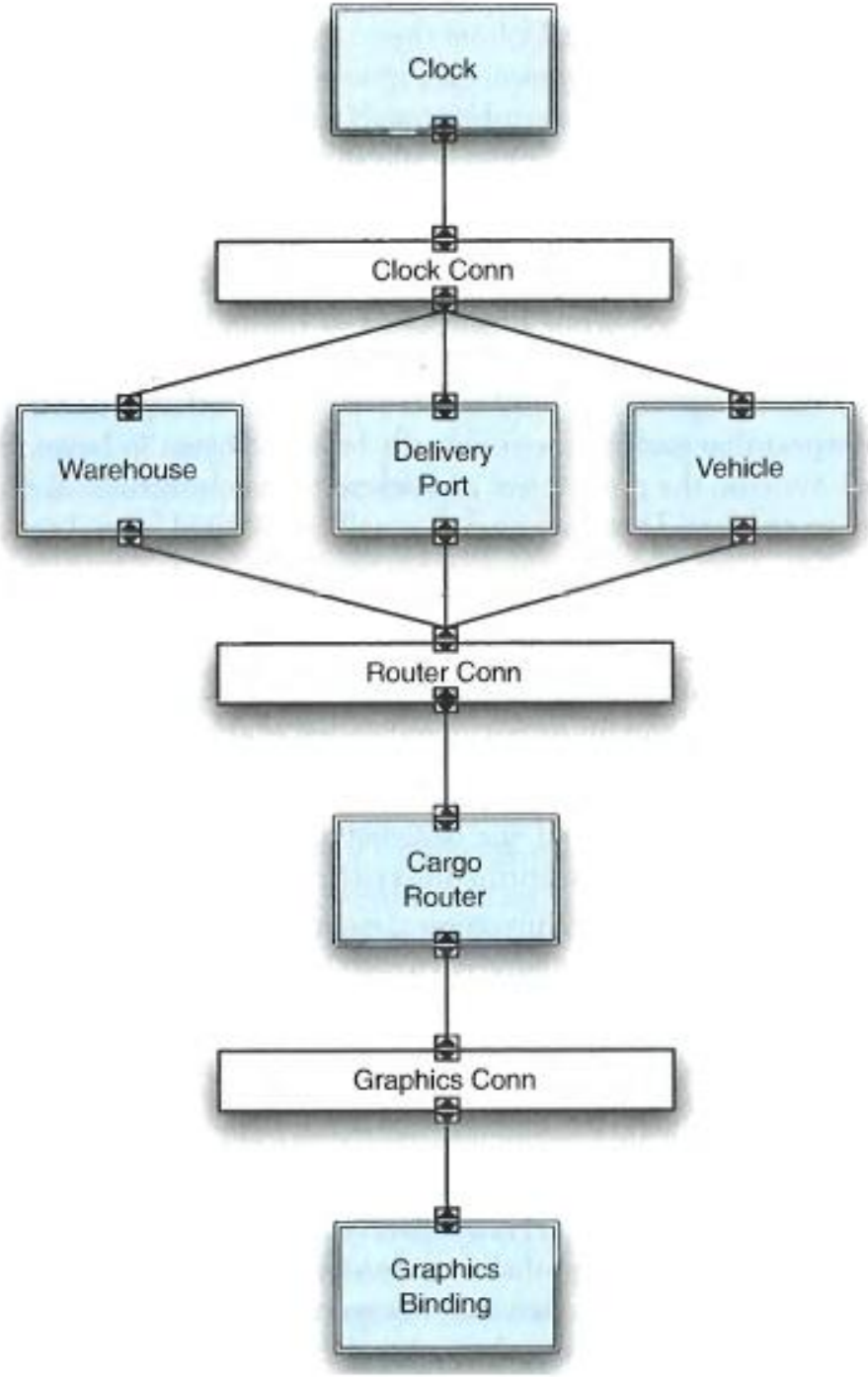
Basic Concepts

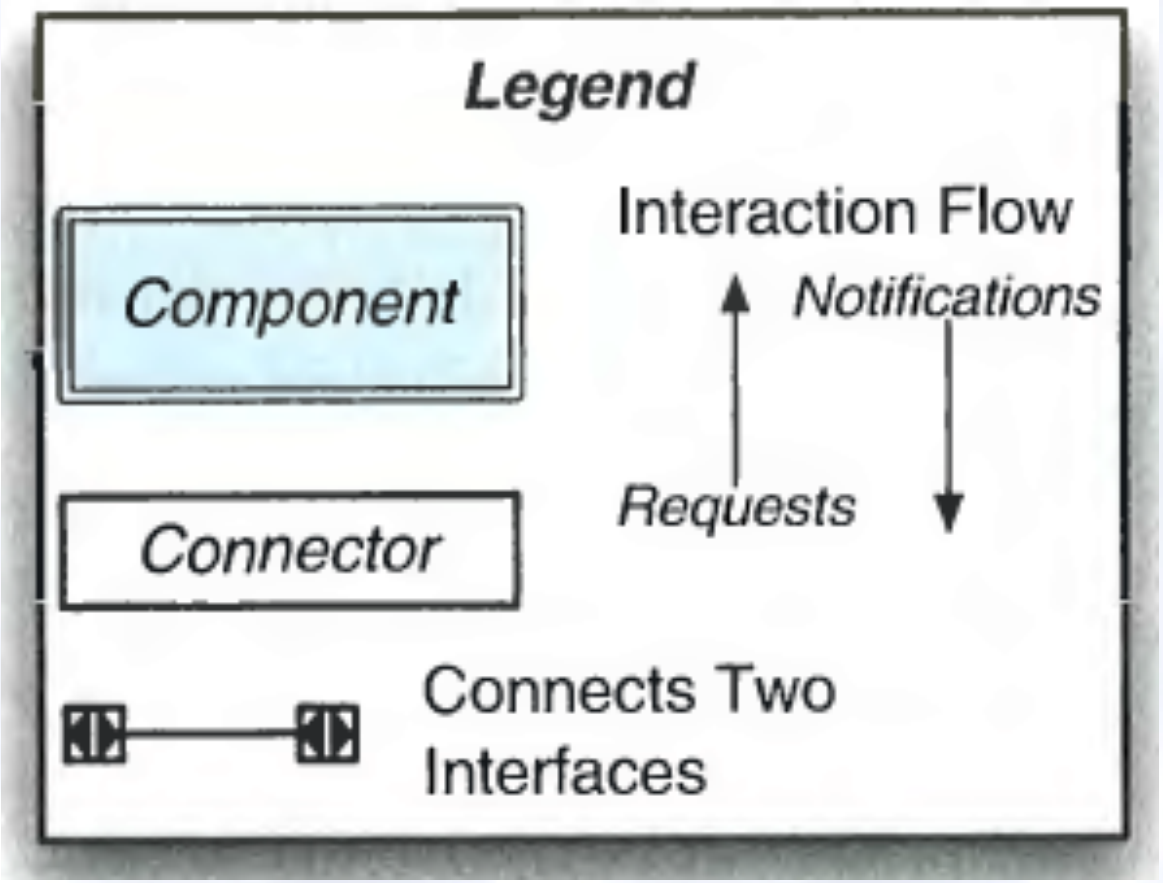
Software Architecture

Prescriptive vs. Descriptive Architecture

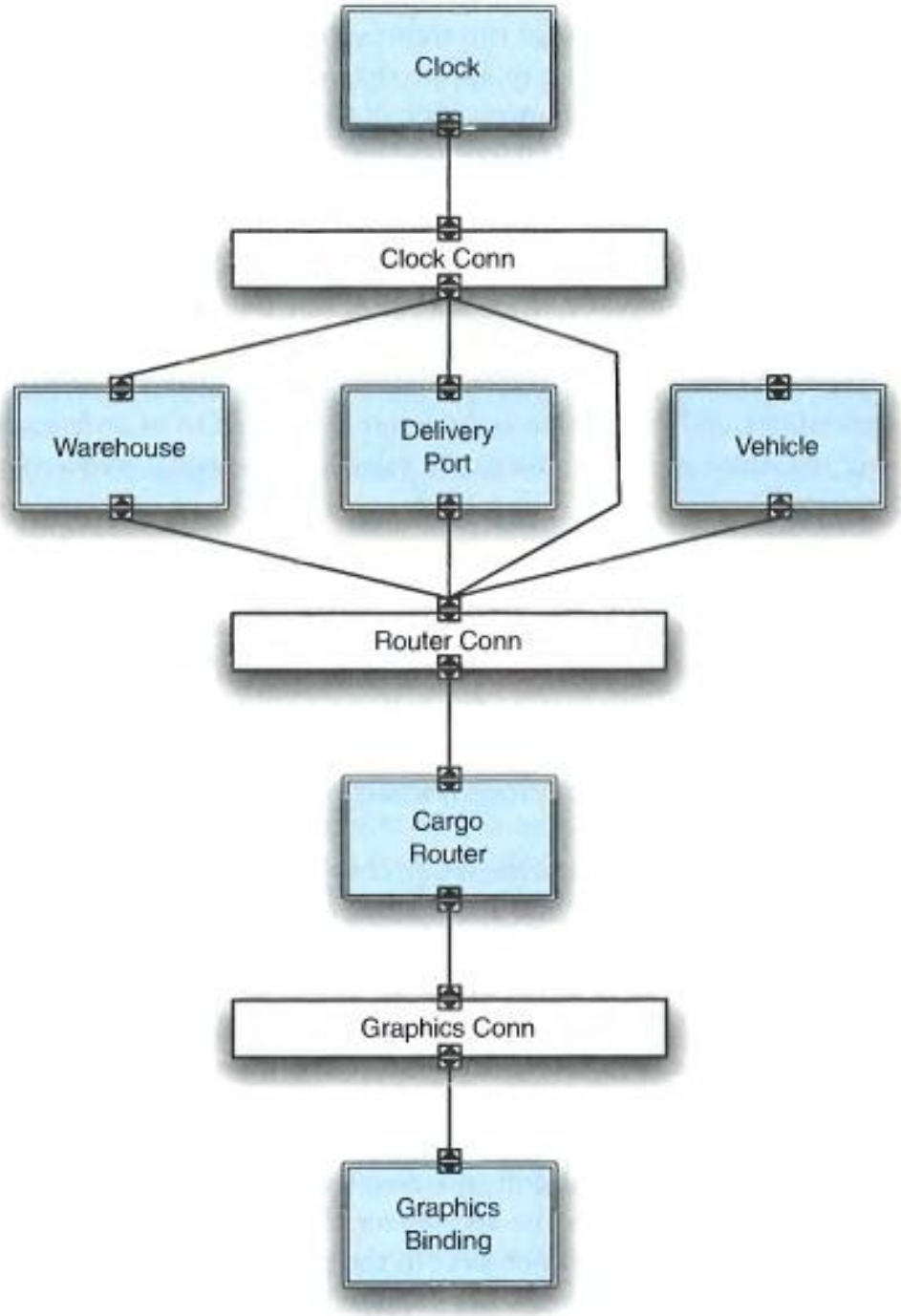
- A system's *prescriptive architecture* captures the design decisions made prior to the system's construction
  - It is the *as-conceived* or *as-intended* architecture
- A system's *descriptive architecture* describes how the system has been built
  - It is the *as-implemented* or *as-realized* architecture

Prescriptive architecture




Legend
Component
Connector
Interaction Flow
Notifications
Requests
Connects Two
Interfaces

Descriptive architecture

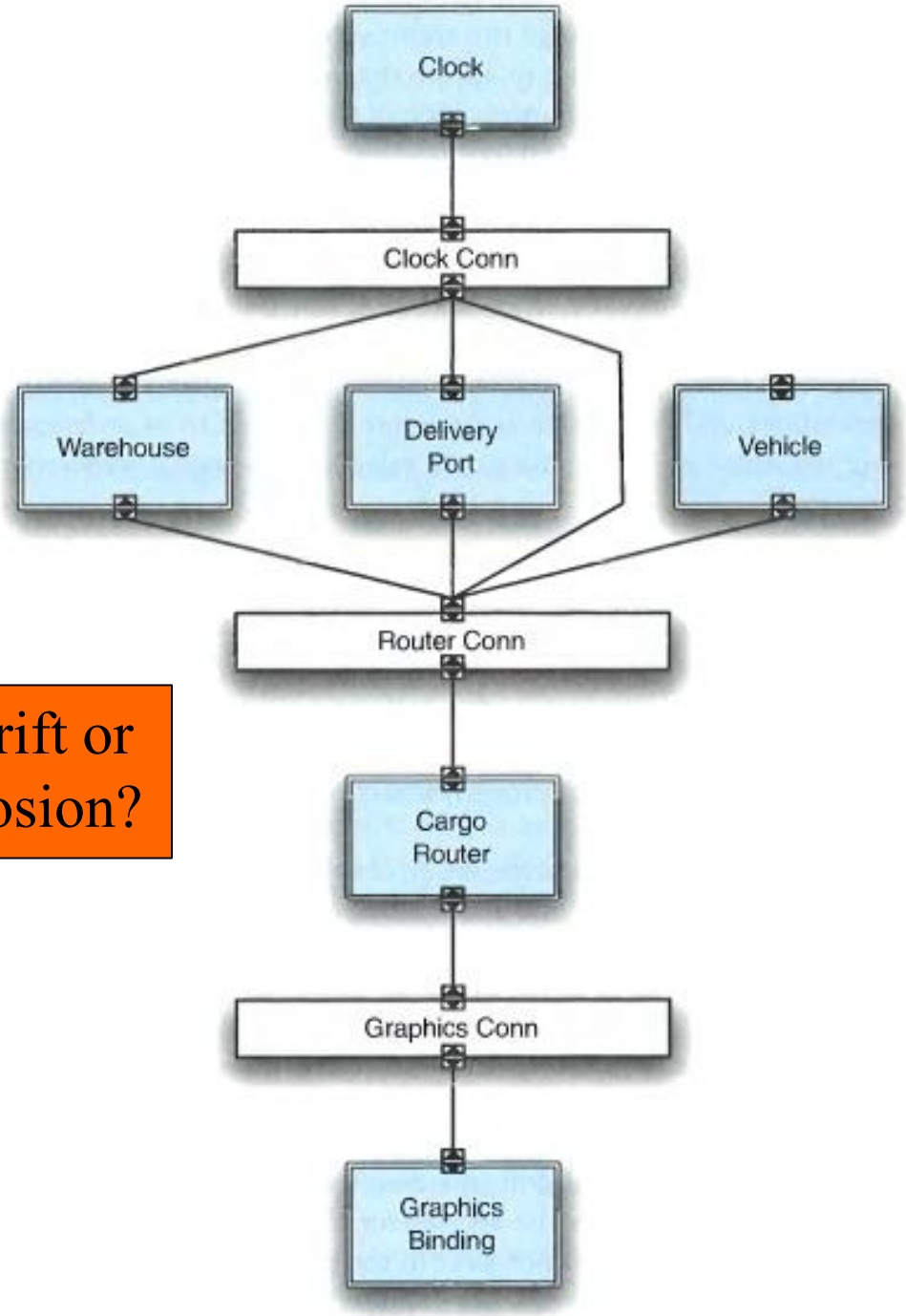


Architectural Evolution

- When a system evolves, ideally its prescriptive architecture is modified first
- In practice, the system – and thus its descriptive architecture – is often directly modified
- This happens because of
  - Developer sloppiness
  - Perception of short deadlines which prevent thinking through and documenting
  - Lack of documented prescriptive architecture
  - Need or desire for code optimizations
  - Inadequate techniques or tool support

Architectural Degradation

- Two related concepts
  - Architectural drift
  - Architectural erosion
- *Architectural drift* is introduction of principal design decisions into a system's descriptive architecture that
  - are not included in, encompassed by, or implied by the prescriptive architecture
  - but which do not violate any of the prescriptive architecture's design decisions
- *Architectural erosion* is the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture



Architectural drift or architectural erosion?

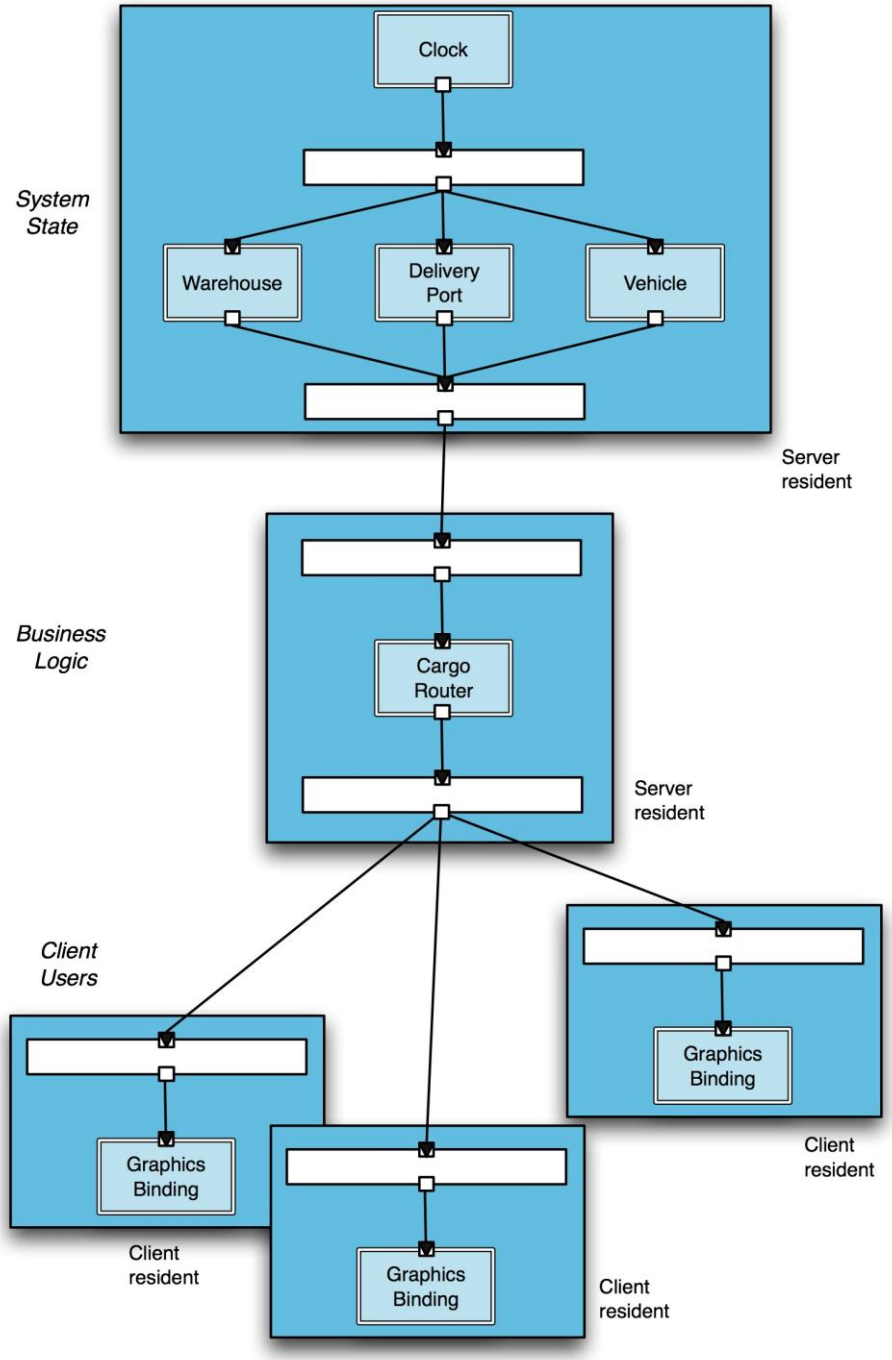
Architectural Recovery

- If architectural degradation is allowed to occur, one will be forced to *recover* the system's architecture sooner or later
- *Architectural recovery* is the process of determining a software system's architecture from its implementation-level artifacts
- Implementation-level artifacts can be
  - Source code
  - Executable files
  - Java .class files

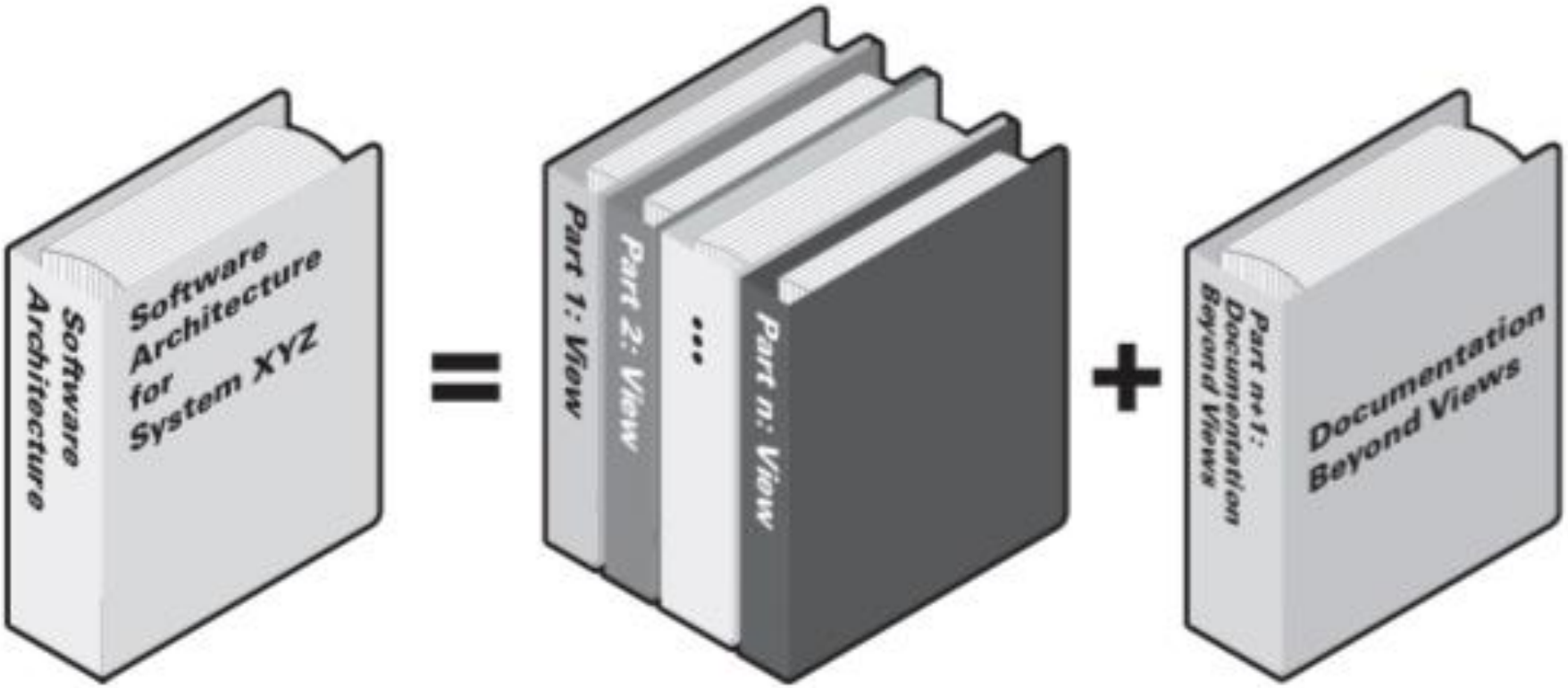
Deployment

- A software system cannot fulfill its purpose until it is *deployed*
  - Executable modules are physically placed on the hardware devices on which they are supposed to run
- The deployment view of an architecture can be critical in assessing whether the system will be able to satisfy its requirements
- Possible assessment dimensions
  - Available memory
  - Power consumption
  - Required network bandwidth

A System's Deployment Architectural Perspective



Architectural Views



Architectural Views (Cont'd)

- What are the relevant views?
- Different views expose different quality attributes to different degrees.
- Quality attributes that are of most concern to you and the other stakeholders in the system's development will affect the choice of what views to document.
- Examples?
  - Layered view?
  - Deployment view?

Architectural Views (Cont'd)

- Different views support different goals.
- Different views will highlight different system elements and/or relations.
- Can a single view represent an architecture?
- Now, let us recall one definition of software architecture:

The software architecture of the system is the **set of structures** needed to **reason about the system**, which comprises software elements, relations among them, and properties of both.

- Structures vs. views?

Structures vs. Views?

- A view is a representation of a coherent set of architectural elements. It consists of a representation of a set of elements and the relations among them.
- A structure is the set of elements itself.
- A view is a representation of a structure(s)

One System, Many Views

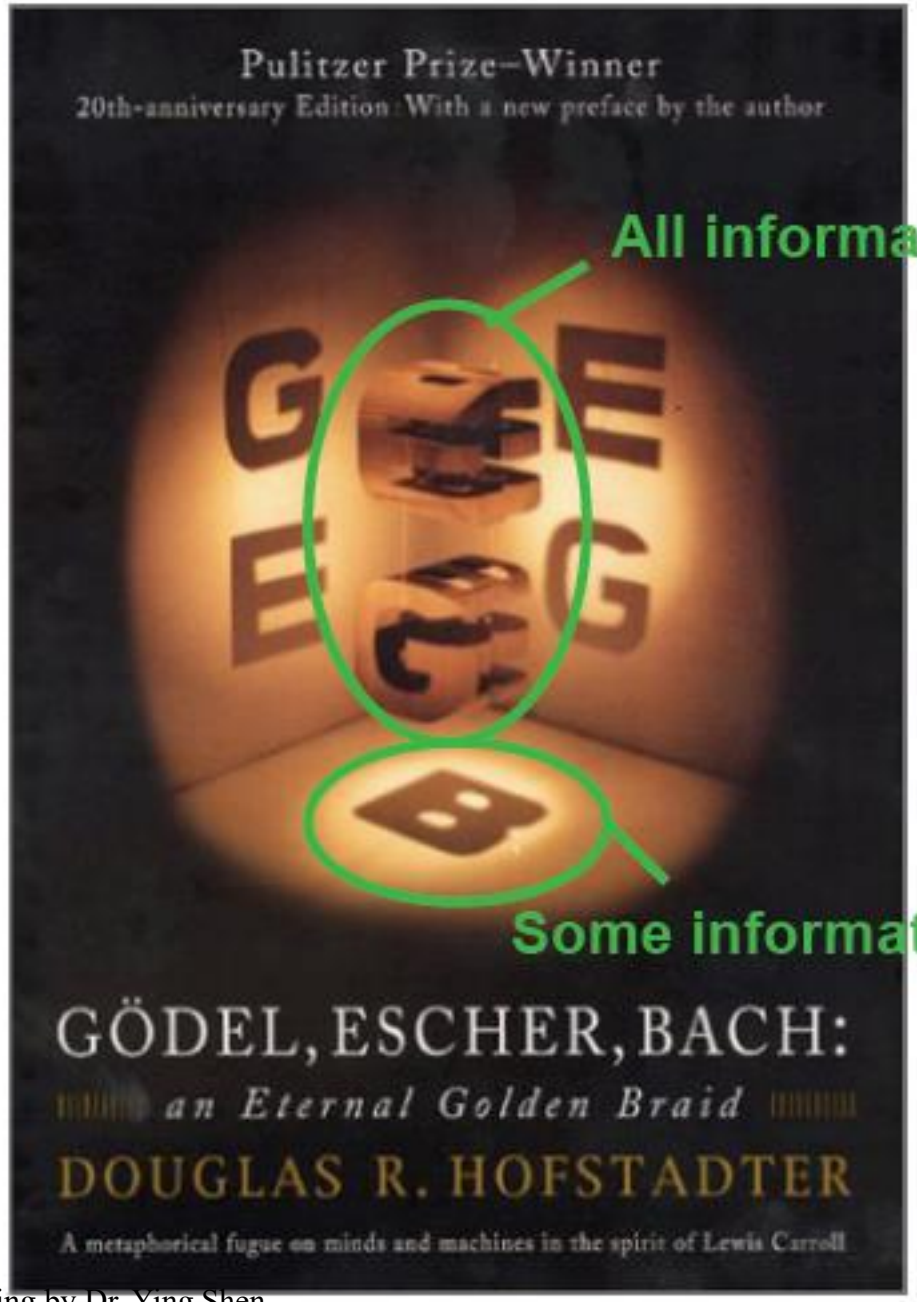


Image taken from a previous Software Architecture course offering by Dr. Ying Shen
http://sse.tongji.edu.cn/yingshen

Kinds of Structures

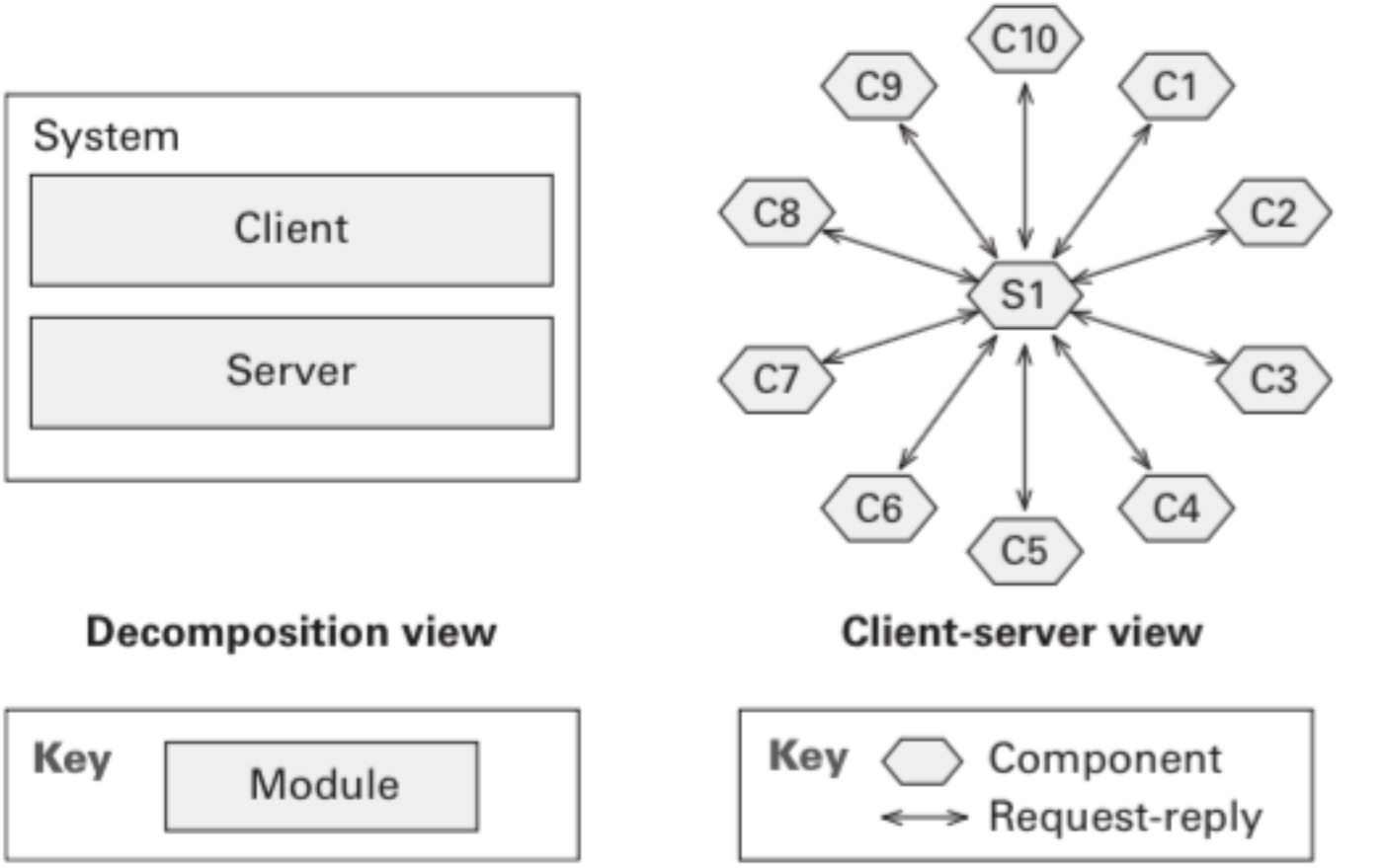
- Architectural structures can be divided into three major categories depending on the broad nature of the elements they show.
- Those categories correspond to three broad kinds of decisions that architectural design involves:
  - Module structures
  - Component-and-connector structures
  - Allocation structures
- Module vs. Component?

Module vs. Component?

- Static vs. dynamic?
- Unit of implementation vs. runtime entity?
- Source code, XML files, config files, GNF files for parsers…Where do they fit?
- An executable binary…Where does it fit?
- Who cares?! Every module is turned into exactly one component at runtime, so this difference is not that important…
  - No!

Module vs. Component?

- A single module may turn into many components.
- Many modules may turn into a single component.



Kinds of Structures

- Those categories correspond to three broad kinds of decisions that architectural design involves:
  - Module structures
  - Component-and-connector structures
  - Allocation structures

Module Structures

- Questions that module structures help us answer…
  - What is the primary functional responsibility assigned to each module?
  - What other software elements is a module allowed to use?
  - What other software does the module use/depend on?
  - What modules are related to other modules by generalization or specialization relationships?
- How can such structures be helpful?
  - Modifiability?

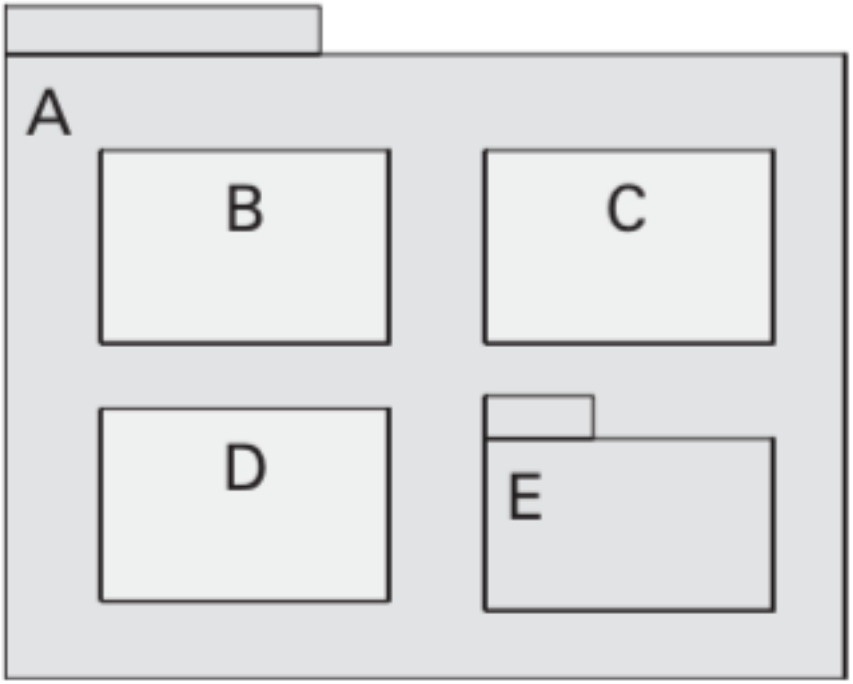
Module Structures

- Some useful module structures:
  - Decomposition structure
  - Uses structure
  - Generalization structure

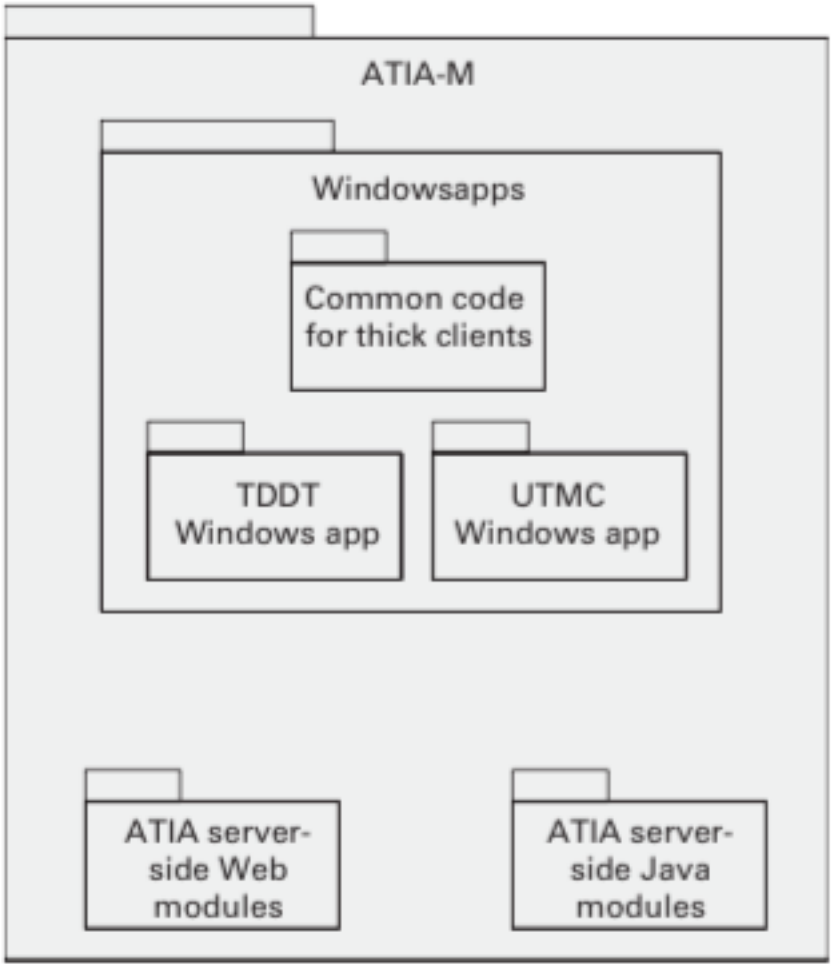
Module Structures – Decomposition Structure

- The decomposition style is used for decomposing a system into units of implementation.
- A decomposition view describes the organization of the code as modules and sub-modules and shows how system responsibilities are partitioned across them.
- Relation: is-a-part of
- Elements: Module (subsystem, package, class,..etc)
- Constraints:
  - No loops are allowed in the *decomposition* graph.
  - A module can have only one parent.

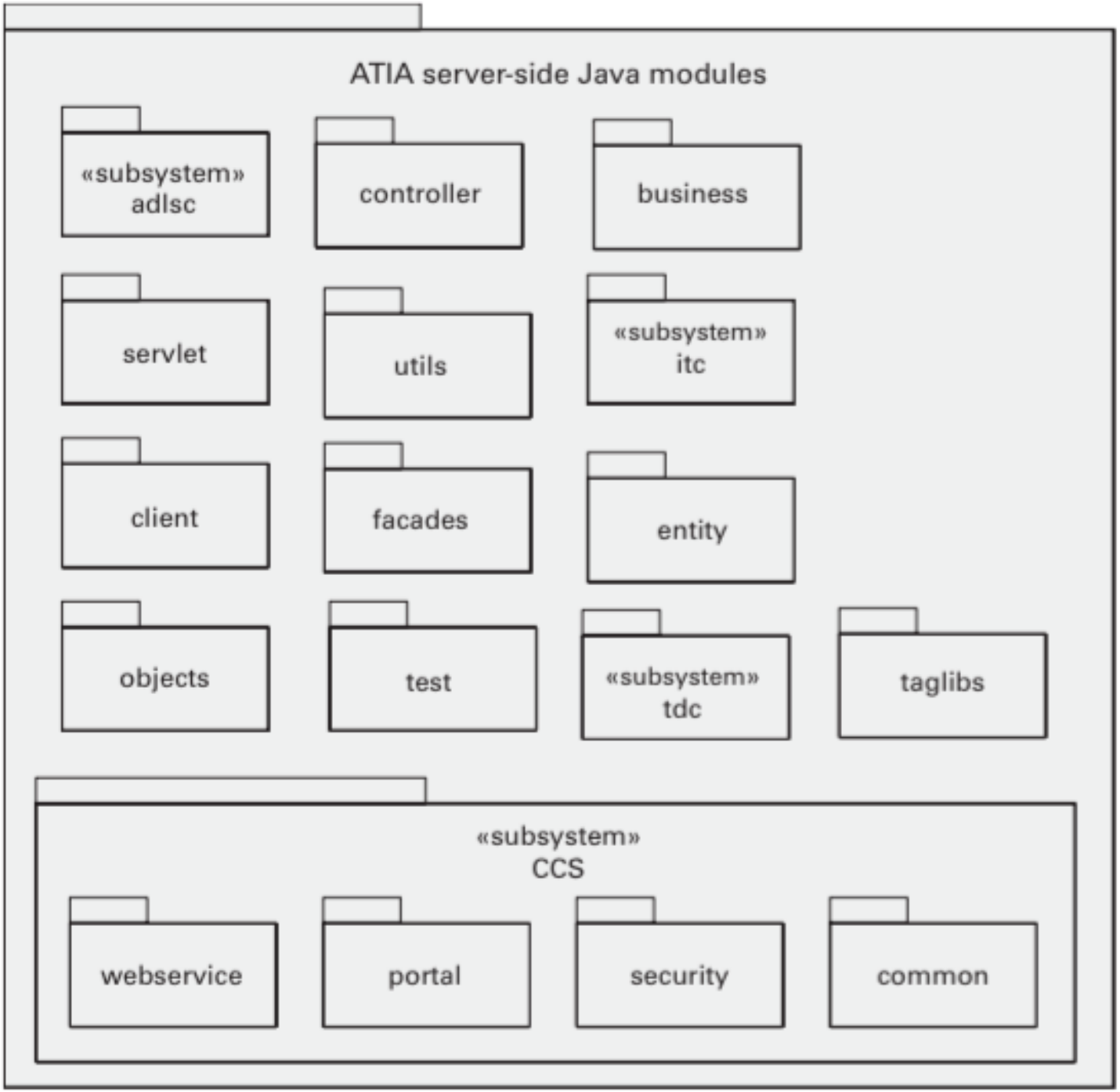
Module Structures – Decomposition structure



Module Structures – Decomposition structure The ATIA-M System

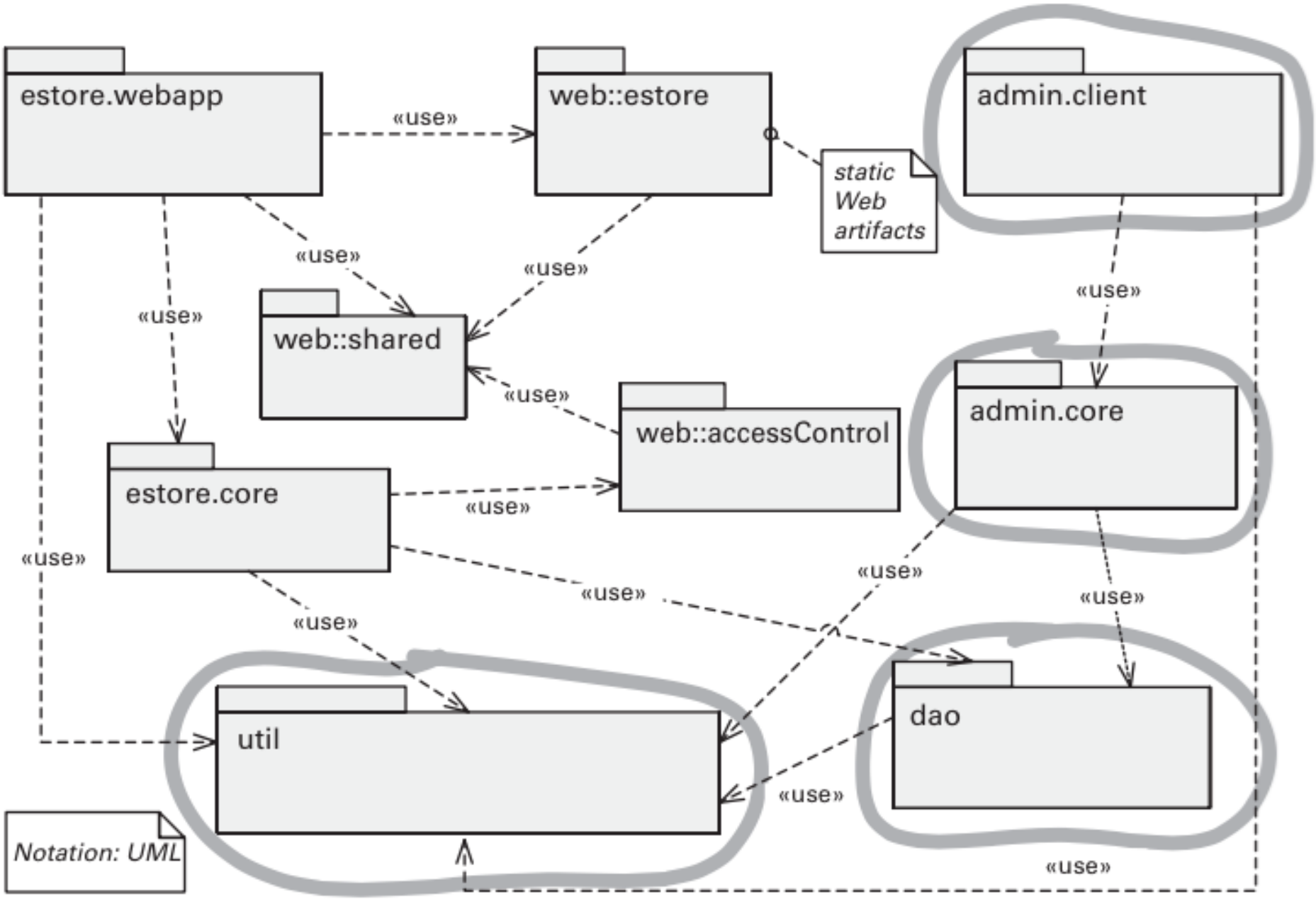


Moc ure
The

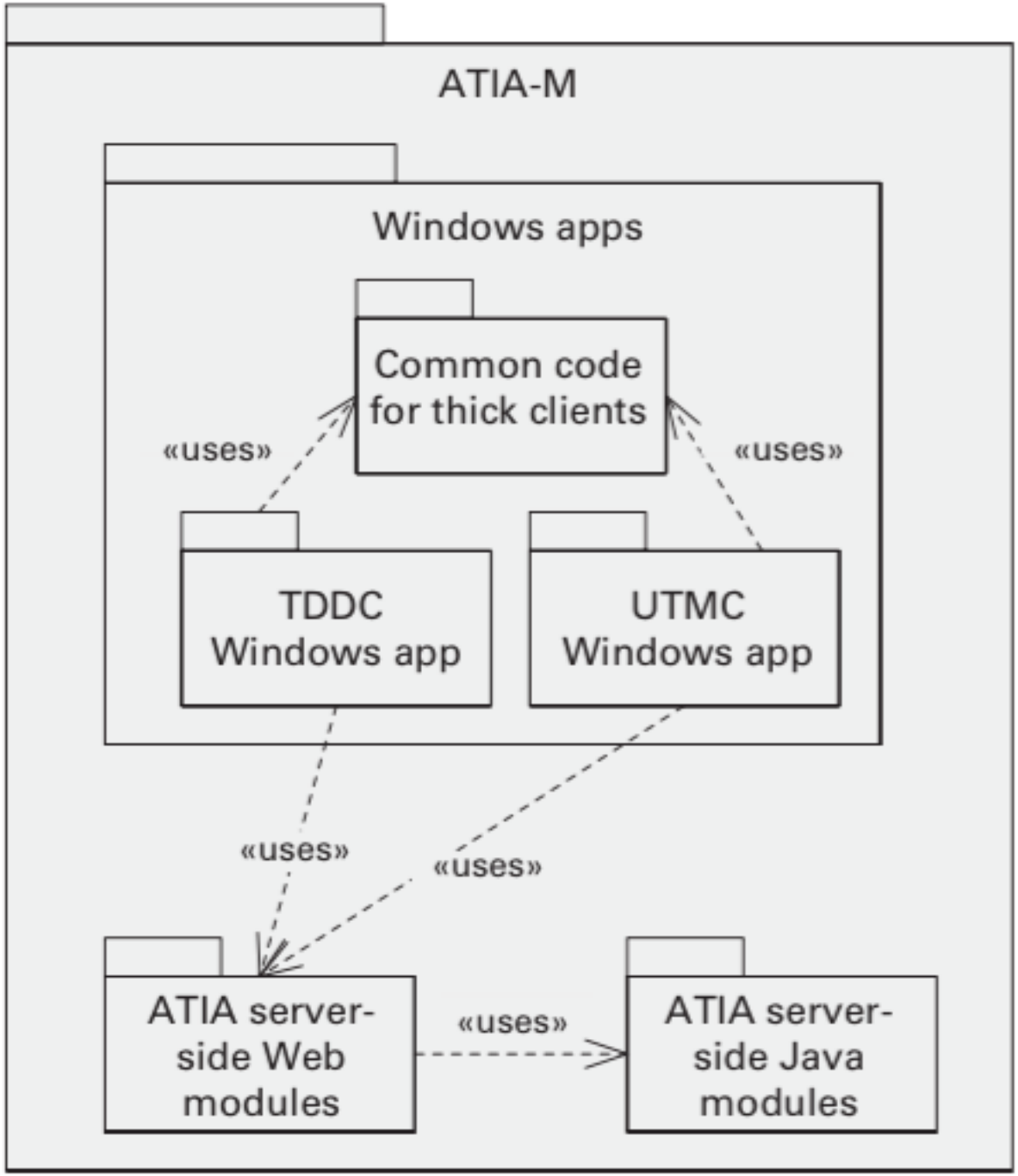

ATIA server-side Java modules
«subsystem»
adlsc
controller
business
servlet
utils
«subsystem»
itc
client
facades
entity
objects
test
«subsystem»
tdc
taglibs
«subsystem»
CCS
webservice
portal
security
common

Module Structures – Uses Structure

- A module ***uses*** another module if its correctness depends on the correctness of the other.
- The uses view makes explicit which modules use which other modules to achieve their responsibilities.
- Relation: uses relation (depends on)
- Elements: Module
- What's it for: Planning incremental development
- Constraints:
  - Using loops?
- Uses vs. calls?


estore.webapp
web::estore
admin.client
«use»
static
Web
artifacts
web::shared
web::accessControl
admin.core
estore.core
util
dao
Notation: UML

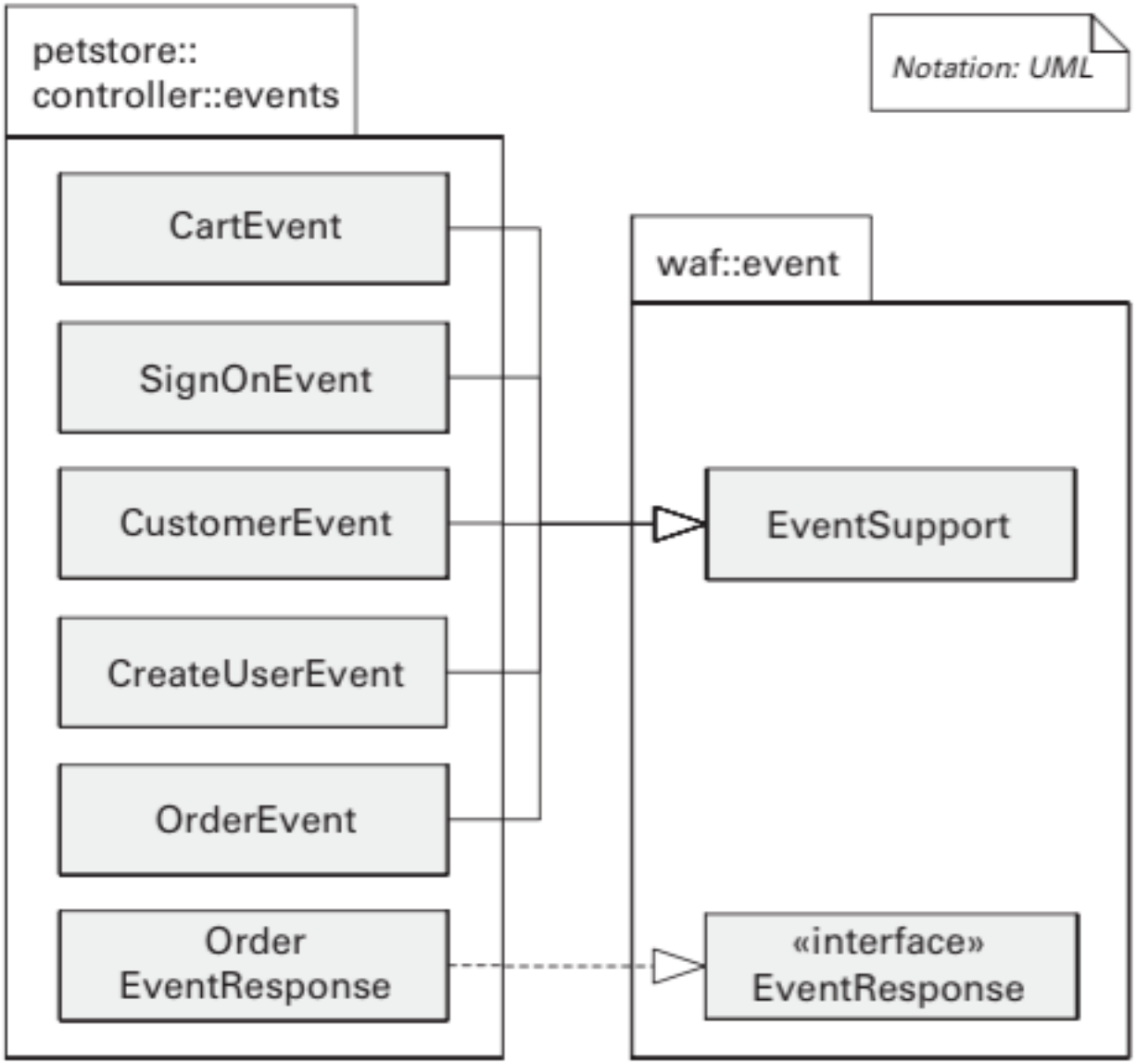
Modu


ATIA-M
Windows apps
Common code for thick clients
«uses»
«uses»
TDDC Windows app
UTMC Windows app
«uses»
«uses»
ATIA server-side Web modules
«uses»
ATIA server-side Java modules

Module Structures – Generalization Structure

- The generalization style employs the *is-a* relation to support extension and evolution of architectures and individual elements.
- Modules in this style are defined in such a way that they capture commonalities and variations.
- Relation: is-a relationship
- Elements: Classes
- What's it for: Supporting reuse.
- Constraints: Cycles in the *generalization* relation are not allowed; that is, a child module cannot be a generalization of one or more of its ancestor modules in a view.

Module Structures – Generalization Structure



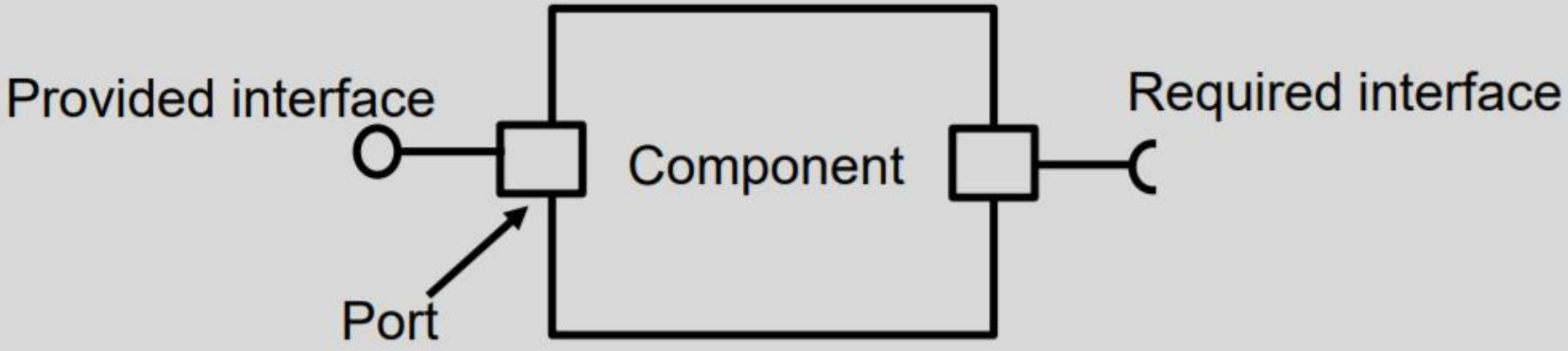
Component-and-Connector (C&C) Structures

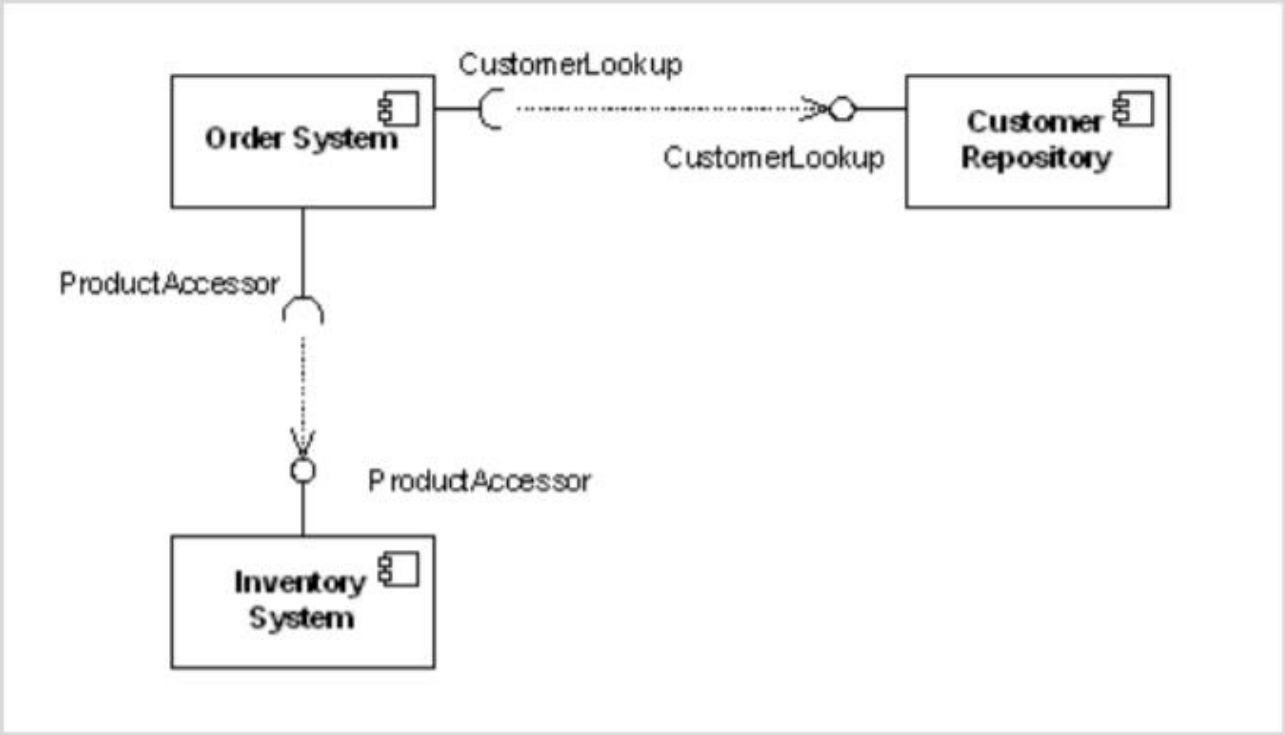
- Questions that C&C structures help us answer…
  - What are the major executing components and how do they interact?
  - What are the major shared data stores?
  - Which parts of the system are replicated?
  - How does data progress through the system?
  - What parts of the system can run in parallel?
- How can such structures be helpful?
  - Performance?
  - Security?
  - Availability?

(C&C) Structures - Concurrency

- Allows the architect to determine opportunities for parallelism and the locations where resource contention may occur
- The units are Components and connectors are their communication mechanisms.

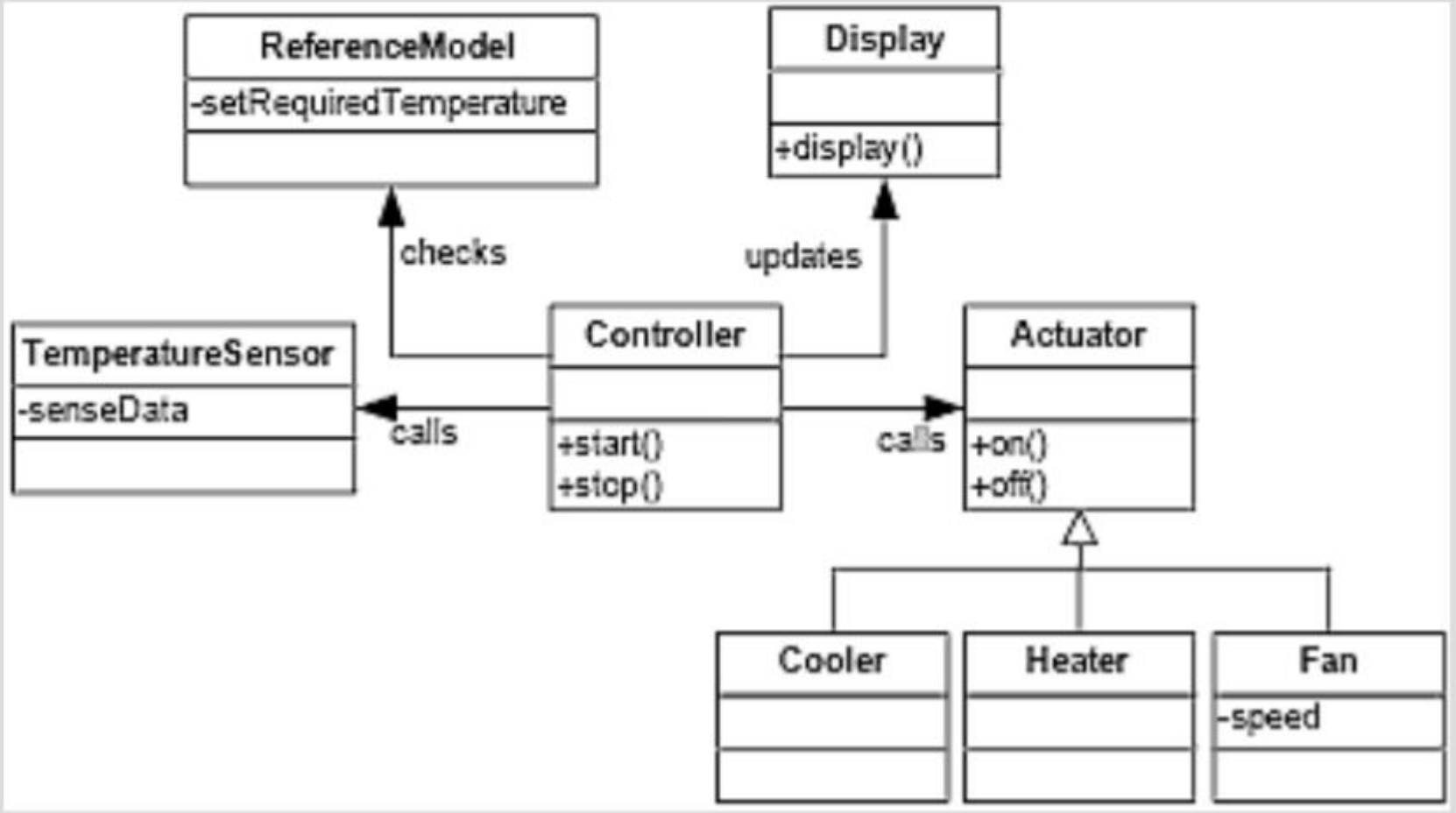
(C&C) Structures – UML Notation



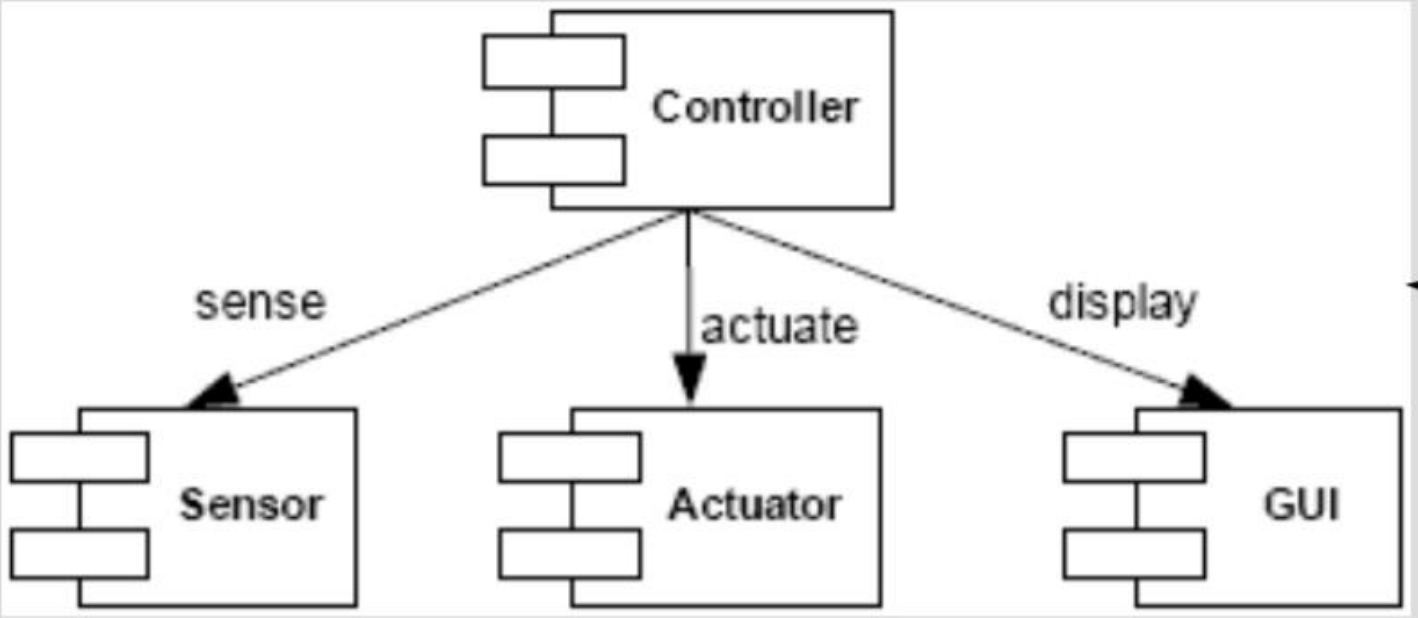


Module View Example

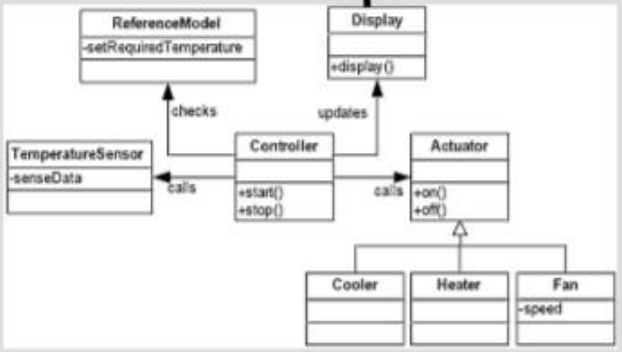
Climate control system in vehicles



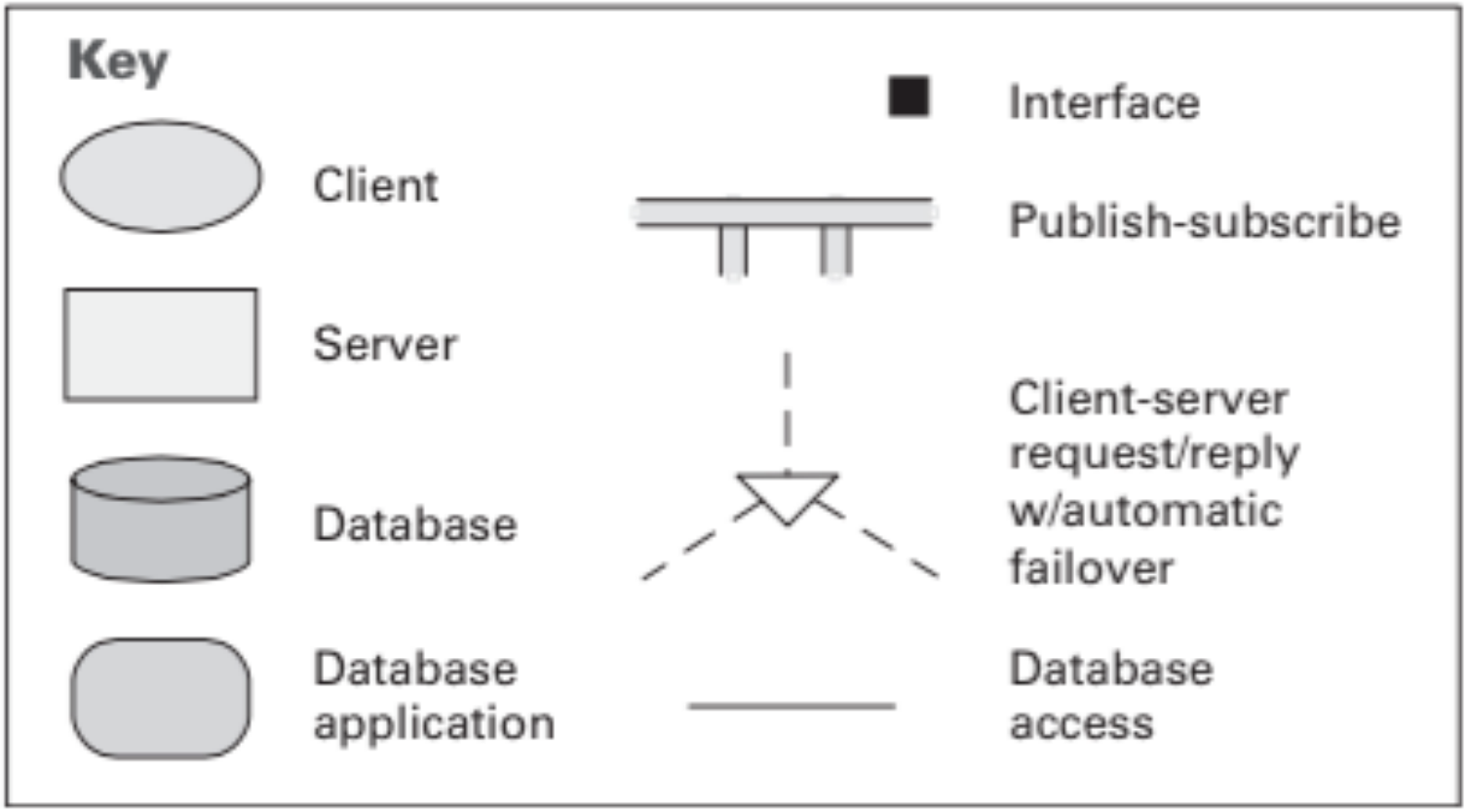
Component-and-Connector View Example



(Can show simplified relationships)



(C&C) views-


Key
Client
Server
Database
Database application
Interface
Publish-subscribe
Client-server request/reply w/automatic failover
Database access

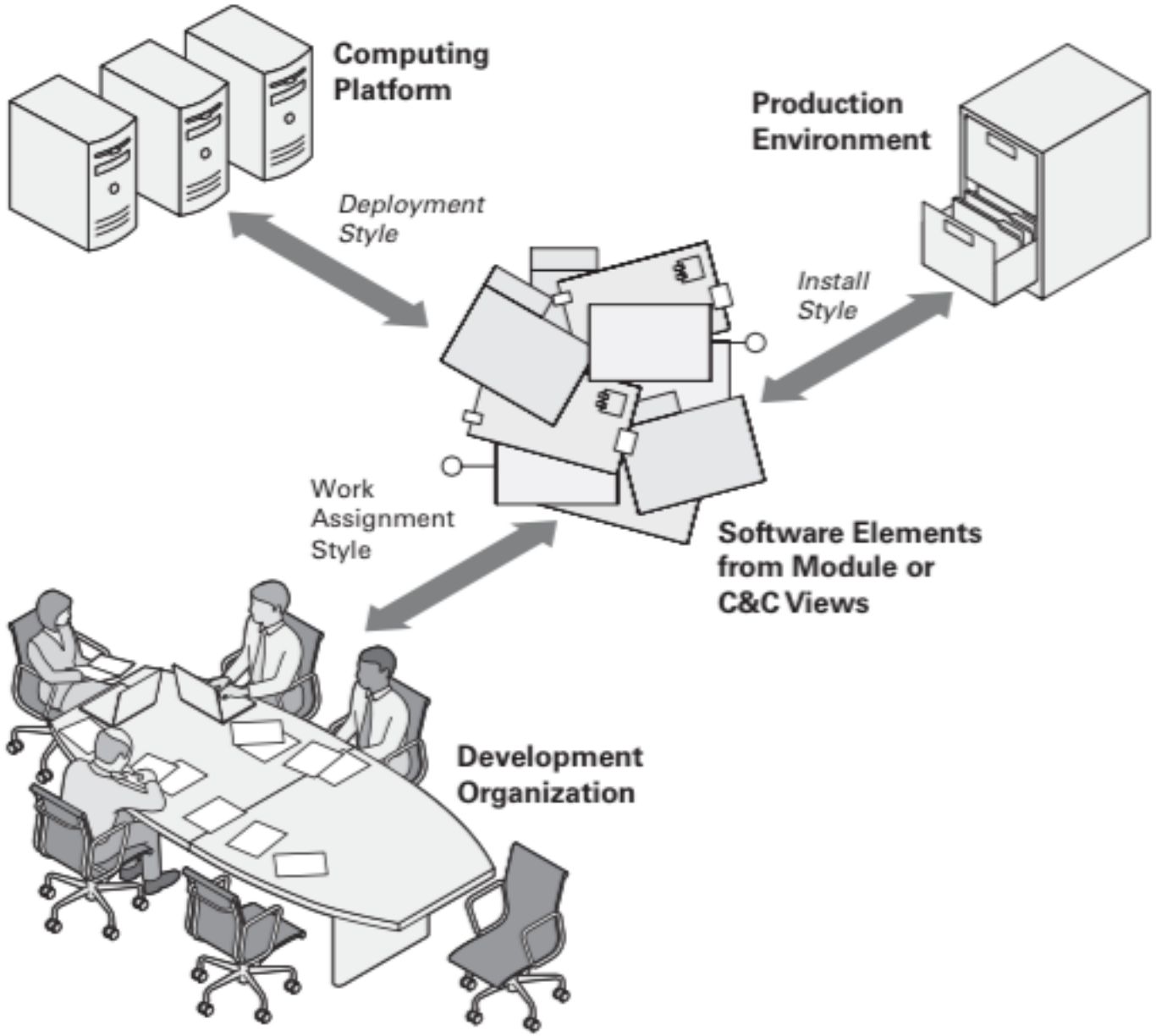
Allocation Structures

- Embody decisions as to how the system will related to non software structures in its environment (such as CPUs, file systems, networks).
- These structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed.
- Questions allocation structures help answer…
  - What processor does each software element execute on?
  - In what directories is each element stored during development/testing/system building?

Allocation Structures

- Allocation styles describe the mapping between the software architecture and its environment.
- Elements: *Software element* and *environmental element*.
- A software element has properties that are *required* of the environment. An environmental element has properties that are *provided* to the software.
- Relation: *Allocated-to*. A software element is mapped (allocated to) an environmental element.

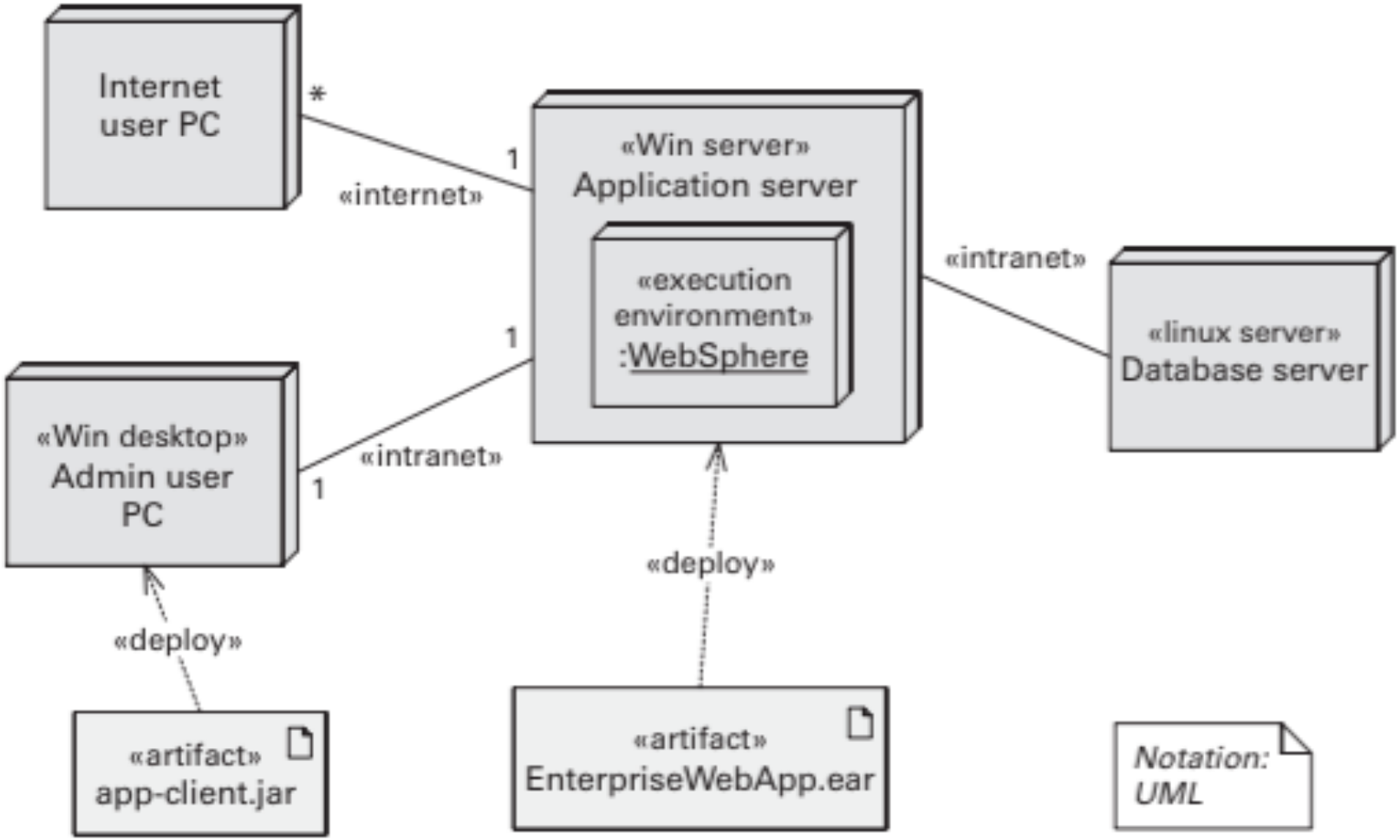
Allocation Structures



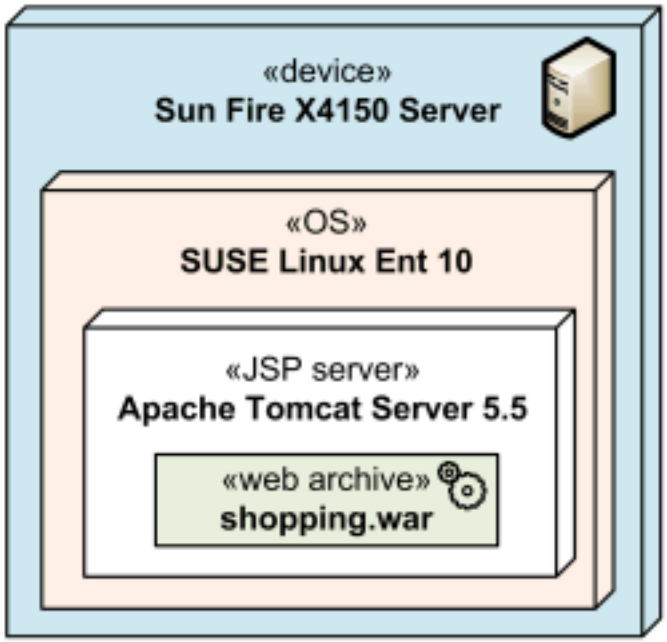
Allocation Views

- *Allocation views* present a mapping between software elements (from either a module view or a component-and-connector [C&C] view) and non-software elements in the software's environment.
- Three common allocation styles:
  - Deployment
  - Install
  - Work assignment

Allocation Structures – Deployment view



Allocation Structures – UML Deployment Diagram



https://www.uml-diagrams.org/deployment-diagrams.html

Allocation Structures – Deployment View

- The deployment style describes the mapping of components and connectors in the software architecture to the hardware of the computing platform.
- Elements:
  - Software elements: from a C&C view
  - Environmental elements
- Usage?
  - Performance?
  - Availability?

Allocation Structures – Deployment View

- Can such view help in memory capacity analysis?
  - Properties relevant to physical units.
  - Properties relevant to software elements.

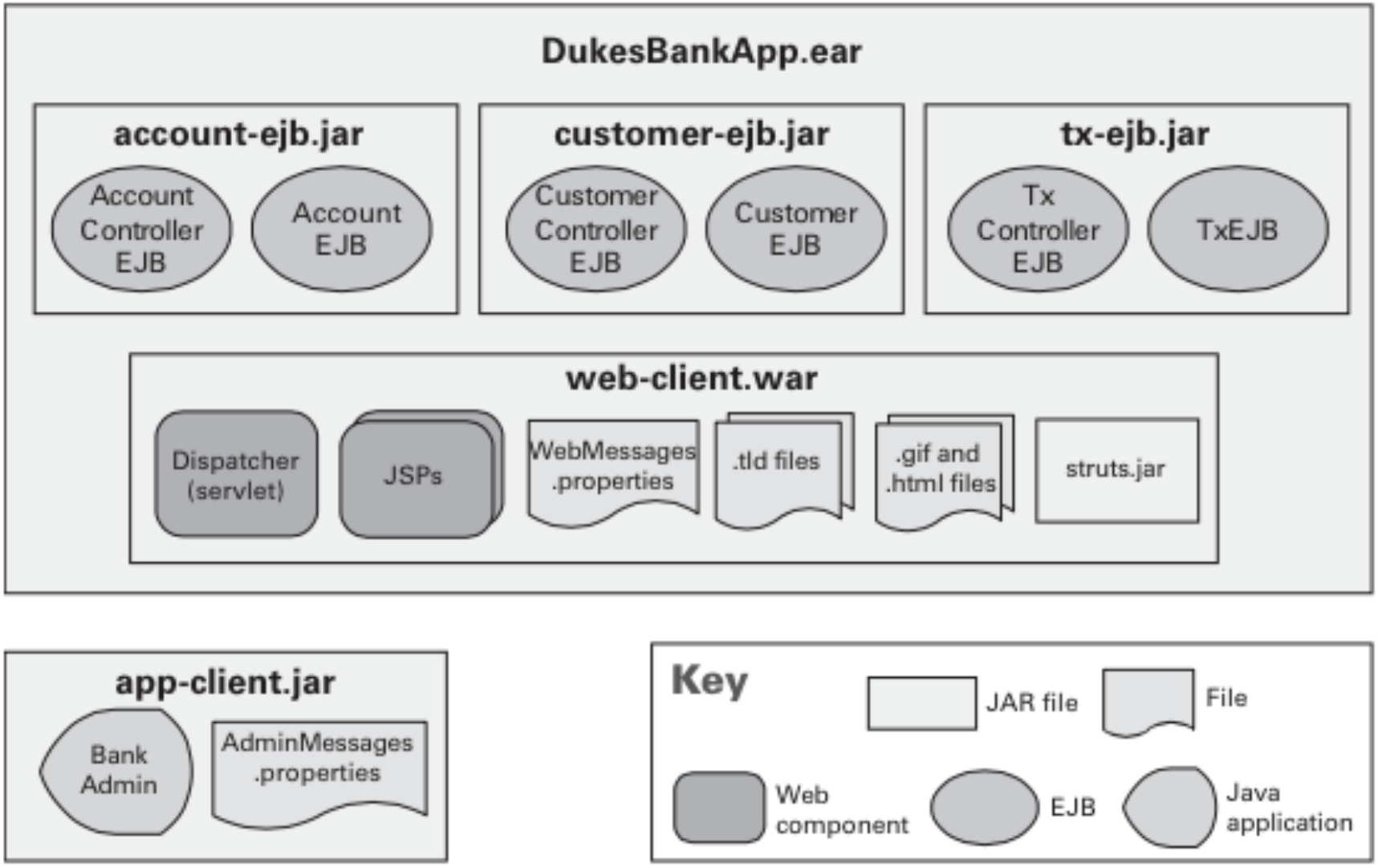
Allocation Structures – Install View

- The install style allocates components of a C&C style to a file management system in the production environment.
- Once a software system is implemented, the resulting files have to be packaged to be installed on the target production platform (such as a desktop computer or a server machine running an application server).
- These files include libraries, executable files, data files, log files, configuration and version control files, license files, help files, deployment descriptors, scripts, and static content (for example, HTML files and images).

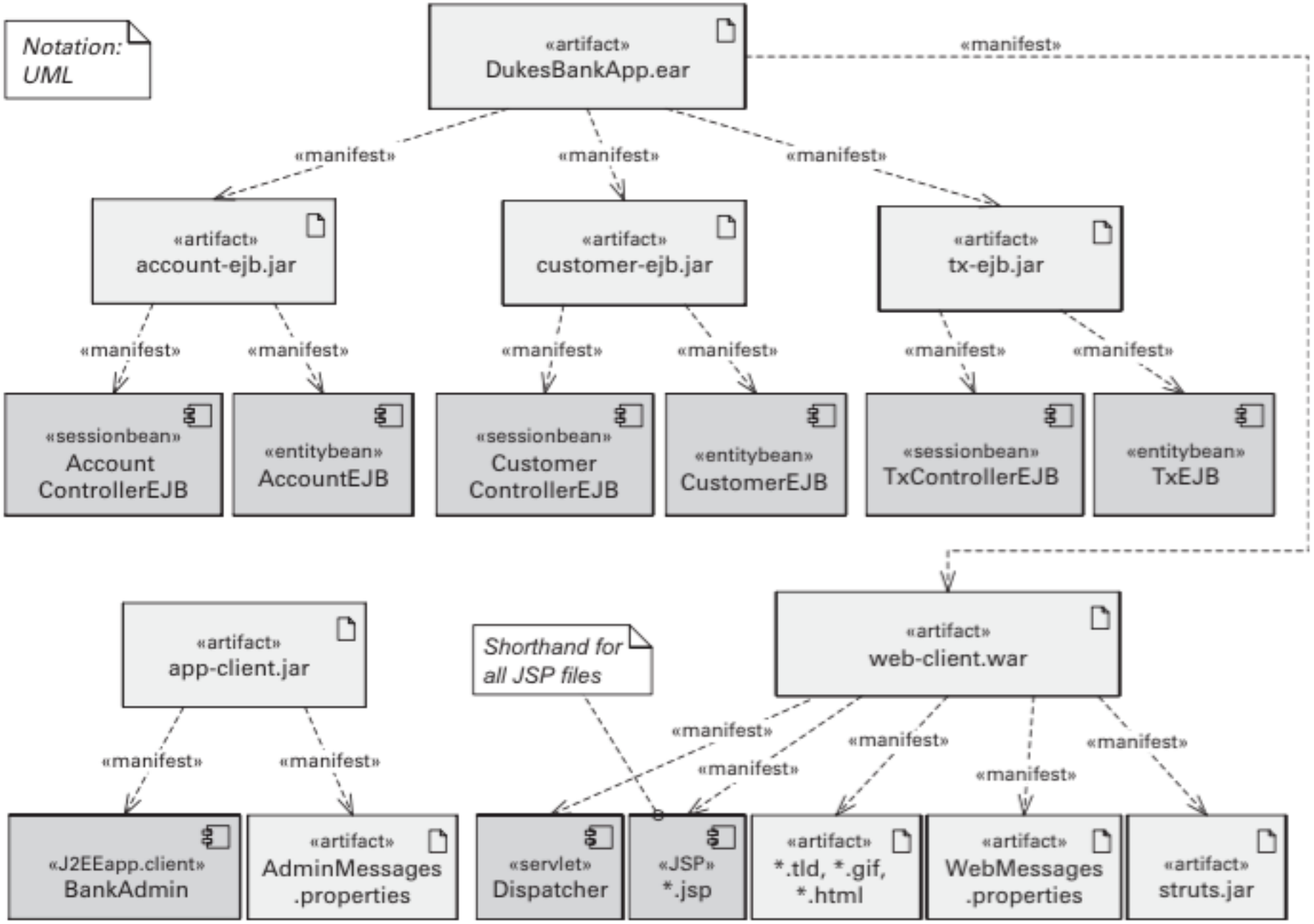
Allocation Structures – Install View

- Any notation for an install view must show components, the files and folders, and the mapping between them.
- The tree structure organization of the files and folders should also be shown.

Allocation Structures – Install View



Allocation Structures – Install View



Allocation Structures – Work Assignment View

- The work assignment style allocates modules of a module style to the groups and individuals who are responsible for the realization of a system.
- This style defines the responsibility for implementing and integrating the modules to the appropriate development teams.
- Elements:
  - Software elements (modules or components?)
  - Environmental elements

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
Flight Operations Segment (FOS)	Planning and Scheduling	Orbital vehicle team
	Data Management	Database team
	User Interface	User interface team
. . .	. . .	. . .

References

- Prologue: “P3. Architecture Views” from: “Documenting Software Architectures” textbook.
- Chapters 2 and 3 from “Documenting Software Architectures” textbook.
- Chapter 5 from “Documenting Software Architectures” textbook
- Epilogue E.2 from “Documenting Software Architectures” textbook
- Taylor, Richard N., Nenad Medvidovic, and Eric M. Dashofy. "Software Architecture: Foundations, Theory, and Practice." . Wiley, 2010.