# Software (Systems) Architecture Foundations

Lecture #3
Documenting Architecture

Alar Raabe

# Recap of Last Lecture

A coherent package of pre-made design decisions that provide a set of properties

- Architectural **structures** can embody decisions how the system
  - is to be structured as a set of code or data **units that have to be constructed or procured**
  - is to be structured as a set of **elements that have run-time behavior** – (components) and interactions (connectors)
  - will **relate to non-software structures in its environment**

- **Architecture Style**
  - characterizes **a family or a class** of system architectures that are related by shared structural and semantic properties
  - is defined by
    - a **vocabulary of design elements**
    - **design rules**, or constraints (incl. topology)
    - **semantic interpretation**
    - **analyses** that can be performed on systems built in that style

# Recap of Last Lecture

A coherent package of pre-made design decisions that provide a set of properties

- Usage of Architecture Styles Supports
  - ***Design Reuse*** – *w*ell-understood solutions can be applied to new problems
  - ***Code Reuse*** – shared implementations of invariant aspects of a style
  - ***Understandability of System Organization*** – e.g. meaning of “client-server”
  - ***Interoperability*** – supported by style standardization
  - ***Style-Specific Analysis*** – enabled by the constrained design space
  - ***Visualizations*** – style-specific descriptions matching engineer’s mental models (e.g. stack diagrams for layers)

- Main Architecture Styles can and must be combined
  - to achieve the required properties of interest
  - to match the problem structures (e.g. ways of decomposition) or problem nature
  - by mixing styles, using hierarchical decomposition or conforming architecture elements to multiple styles

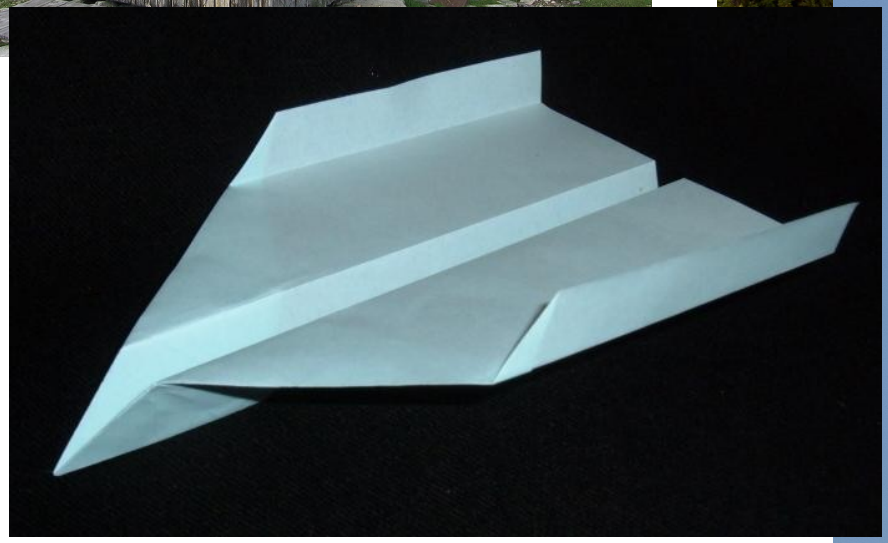
# Content

38. The architect concerns himself with the depth and not the surface, with the fruit and not the flower.

Lao Tsu (by Philippe Kruchten)

- Why to Document Architecture
- CMU SEI – “Views & Beyond” Method
  - Module Views
  - Component-and-Connector Views
  - Allocation Views
  - Advanced techniques
- Some other Architecture Documentation Methods
- Other Architecture Documentation Practices
  - Architecture Description Languages
  - Documenting Architecture in Code
- Conclusions

# Simple & Unimportant vs. Complex & Important
# Built by One vs. Built by Many




Pictures © Wikipedia & Wikimedia Commons

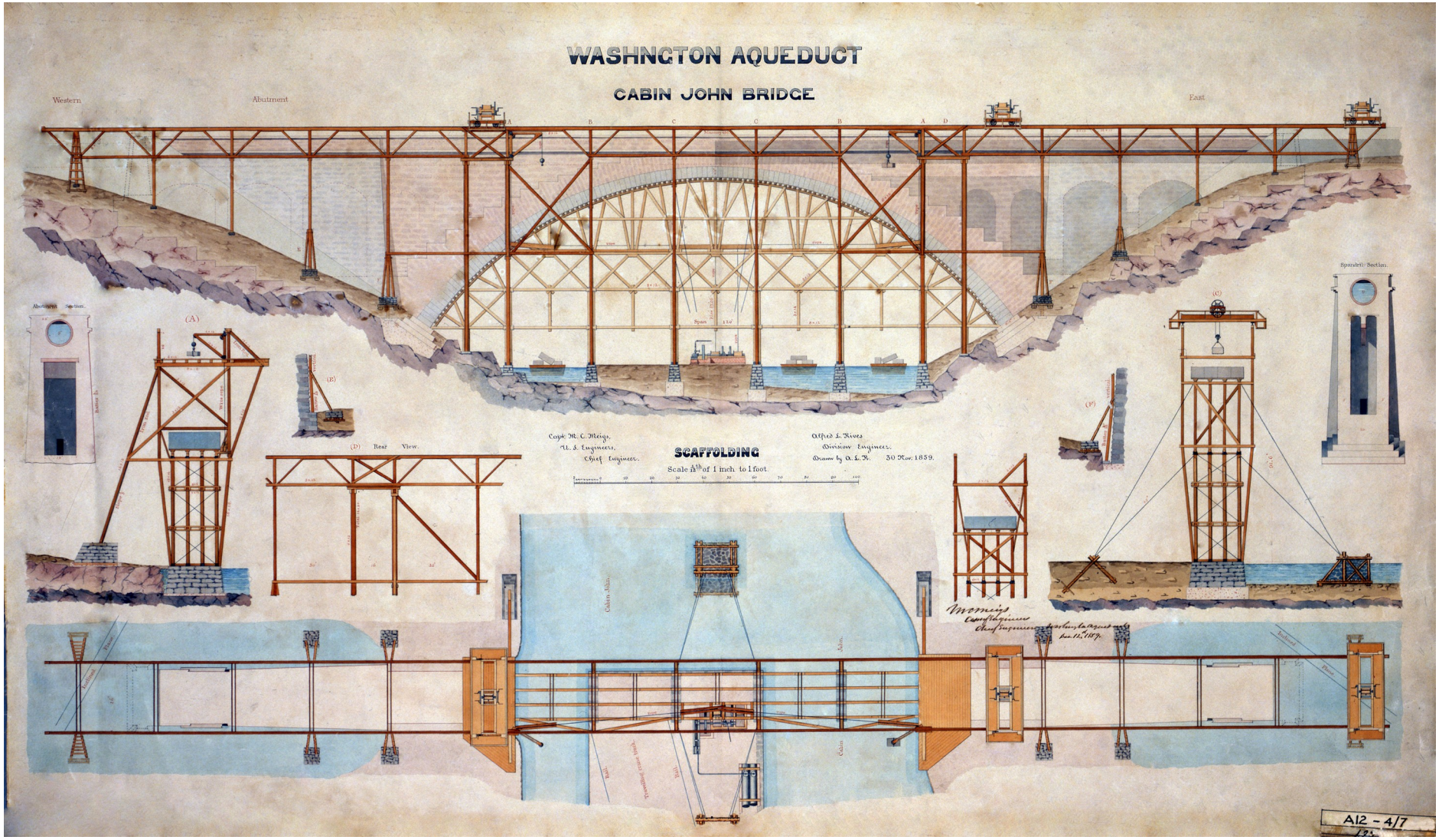
# Many people, same goal →
# need for Common Language

- And the Lord said,
  'Look, they are one people,
  and **they have all one language**; and this is only the beginning of what they will do;
  **nothing that they propose to do will now be impossible for them**.

  – Genesis 11.6

Picture © Wikipedia & Wikimedia Commons

# Complex Object Requires many Views

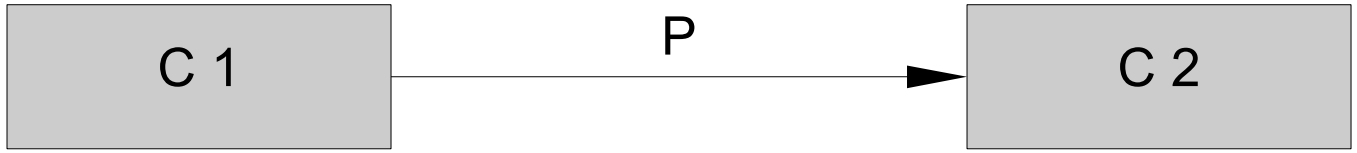



Picture © Wikipedia & Wikimedia Commons

# Boxes and Arrows – What they Mean?

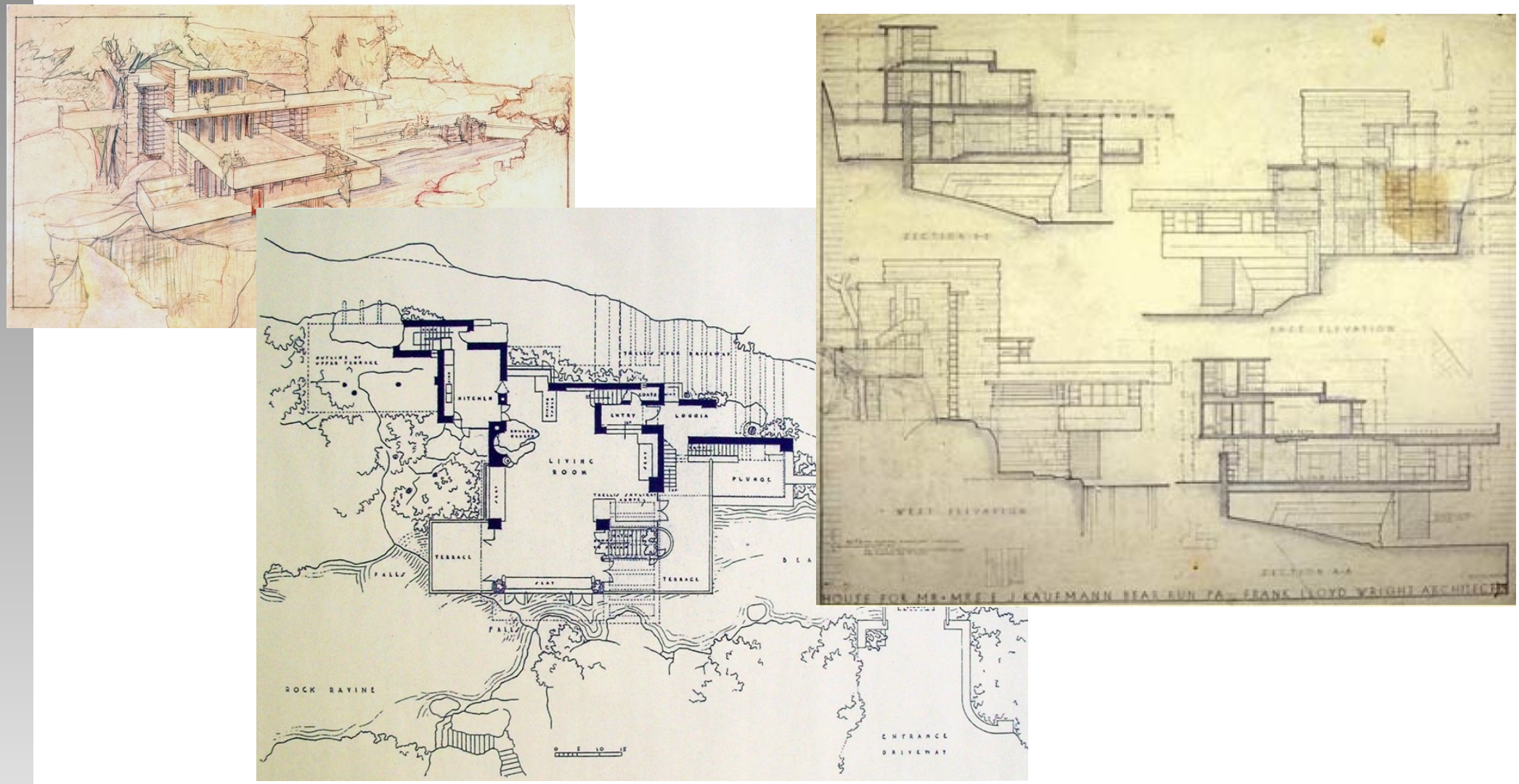
80% of time during maintenance is spent in design-rediscovery

Davidson (2002)




- What does this mean?
  - C1 calls C2
  - Data flows from C1 to C2
  - C1 instantiates C2
  - C1 sends a message to C2
  - C1 is a subtype of C2 (usually C2 would be positioned above C1, but that is not mandatory)
  - C2 is a data repository and C1 is writing data to C2
  - C1 is a repository and C2 is reading data from C1

# Illustration vs. Drawing/Documentation




Picture © The Frank Lloyd Wright Foundation

# Documenting Architecture

Designing an architecture without documenting it, is like winking at a girl in the dark – you know what you´re doing, but nobody else does

E. Woods

- Creating an architecture is not enough – it **has to be communicated** properly to let others use it properly to do their jobs

- Architecture documentation is for
  - **communication** – primary communication vehicle between stakeholders
  - **education** – introducing new people to the system
  - **designing** – provides structure for design decisions
  - **analyzing** – provides information to analyze the system properties (quality attributes)
  - **constructing** – tells what to implement (must contain models to support automated construction)

*Specification* – architecture rendered in a formal language

*Representation* – a model, an abstraction of an architecture

# Purpose of Architecture Documentation for Different Stakeholders

To record and communicate our knowledge and decisions about the software system architecture

- **Business Manager**
  - understanding the ability of selected architecture to meet business goals
- **Customer**
  - assuring that required functionality and quality will be delivered
  - estimating cost and deliveries and following up progress of development
- **Analyst**
  - analyzing satisfaction of quality attribute requirements
- **Architect**
  - making trade-offs between conflicting requirements and design approaches
  - recording design decisions and providing evidence that the architecture satisfies the requirements
- **Designer**
  - understanding the context of their part of the system and its interactions with other parts
- **Developer / Implementer**
  - understanding the constraints on development
- **Tester / Quality Assurer**
  - assuring that implementation has been faithful to the architectural prescription
  - creating test plans and tests
- **Maintainer**
  - understanding how to deploy and operate
  - understanding the effects of change

# Documenting an Architecture

Software can be described by many structures, not just one

D. Parnas

- Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view

- Rules for Sound Documentation
  - Write Documentation from the Reader's Point of View
  - Avoid Unnecessary Repetition
  - Avoid Ambiguity
  - Use a Standard Organization
  - Record Rationale
  - Keep Documentation Current But Not Too Current
  - Review Documentation for Fitness of Purpose

- For systems that change fast
  - Document what is true about all versions of your system
  - Document the ways the architecture is allowed to change
  - Make your system capture its own *architecture-of-the-moment* automatically

# Content

38. The architect concerns himself with the depth and not the surface, with the fruit and not the flower.

Lao Tsu (by Philippe Kruchten)

- Why to Document Architecture
- CMU SEI – “Views & Beyond” Method
  - Module Views
  - Component-and-Connector Views
  - Allocation Views
  - Advanced techniques
- Some other Architecture Documentation Methods
- Other Architecture Documentation Practices
  - Architecture Description Languages
  - Documenting Architecture in Code
- Conclusions

# “Views & Beyond”

CMU SEI Views & Beyond

- Method

1. Finding out what stakeholders need
   (to avoid creating documentation that may serve no one)

2. Providing the information to satisfy those needs by recording design decisions according to a variety of views, plus the beyond-view information

   - how system is structured as a set of implementation units
   - how system structured as a set of elements that have run-time behavior and interactions
   - how software system relates to non-software structures in its environment

3. Checking the resulting documentation to see if it satisfied the needs

4. Packaging the information in a useful form to its stakeholders

# Three Categories of Views – Viewpoints (according to three kinds of structures)

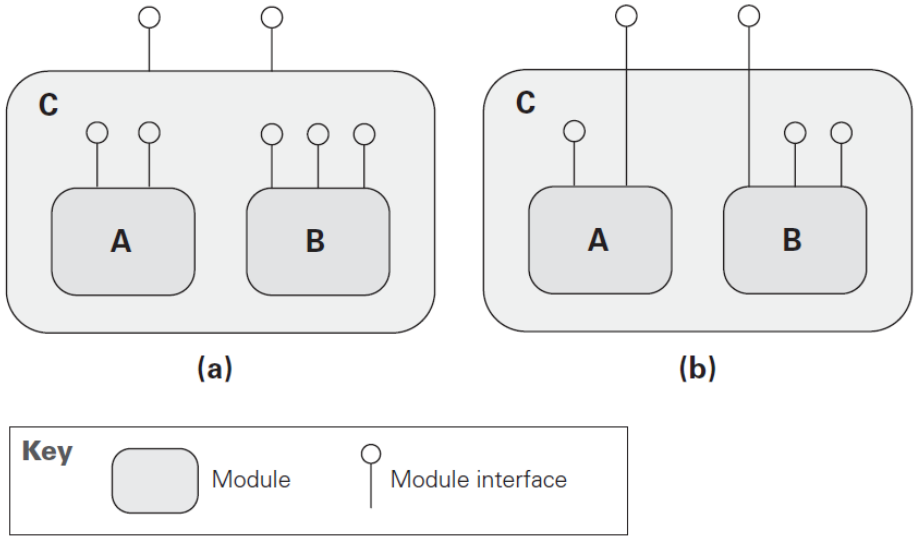
CMU SEI Views & Beyond

- **Module** viewpoint
  - introduces a specific set of module types and specifies rules about how elements of those types can be combined
- **Component-and-connector** viewpoint
  - introduces a specific set of component and connector types and specifies rules about how elements of those types can be combined
  - given that C&C views capture run-time aspects of a system, a C&C style is typically also associated with a computational model that prescribes how data and control flow through systems designed in that style
- **Allocation** viewpoint
  - describes the mapping of software units to elements of an environment in which the software is developed or executes

# The Module Viewpoint Overview

CMU SEI Views & Beyond

- Way to document the modular structures of software system
  - the way in which software is decomposed into manageable units of responsibilities
  - documentation package of any software system must include at least one view in the module viewpoint
- A module is a code unit that implements a set of responsibilities – a principal implementation unit (e.g. a class, a collection of classes, a layer, or any other code unit)
- Modules can be decomposed into modules
- Modules have
  - Properties
    - express important info about module
    - constraints imposed on the module
  - Interfaces (what is available to other modules)
  - Relations to each other (no cycles allowed)




Pictures © CMU SEI & Pearson Education, Inc.

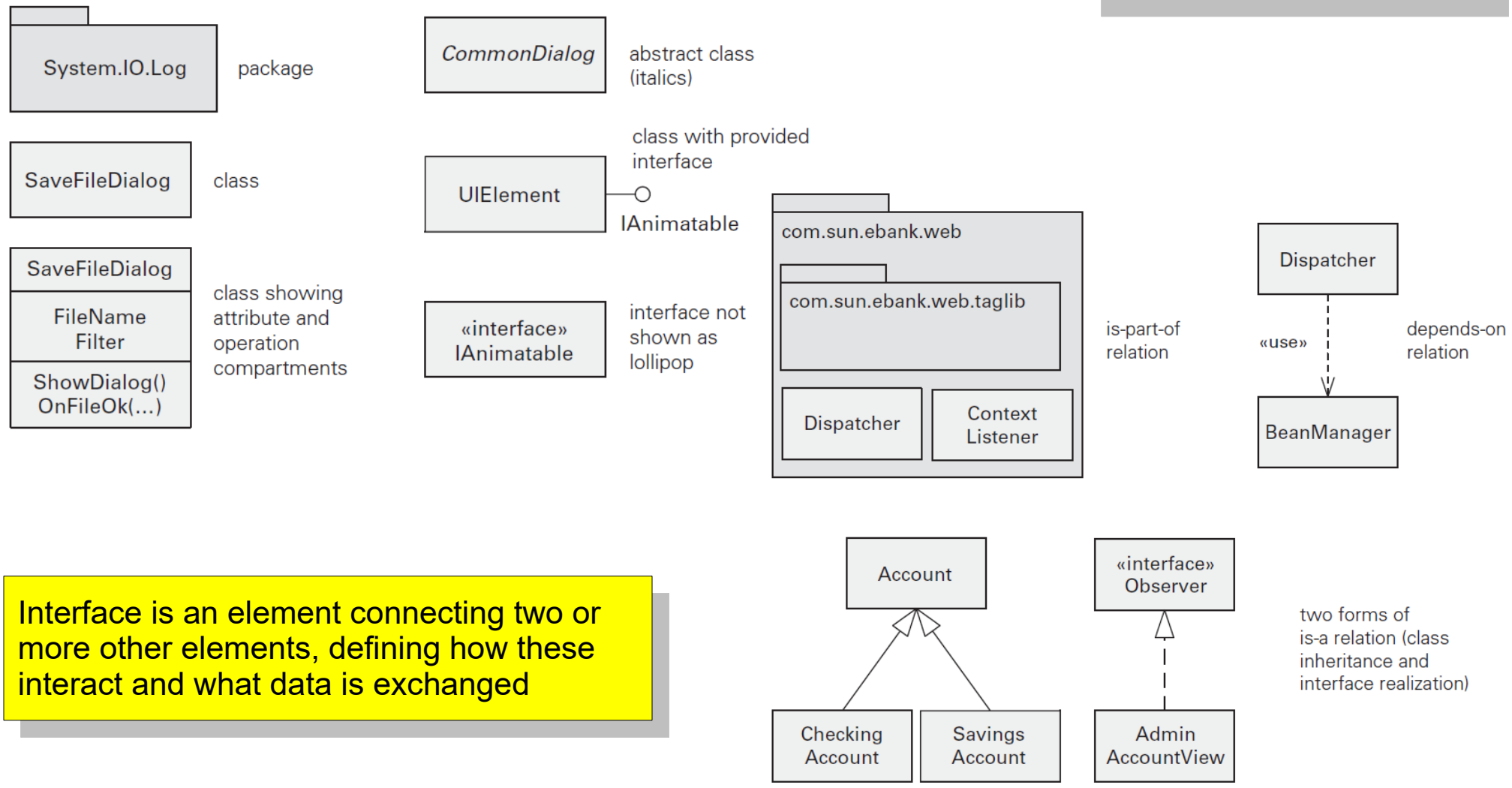
# The Module Viewpoint
# Summary

CMU SEI Views & Beyond

- Elements
  - **Modules** – implementation units of software that provide a coherent set of responsibilities
  - **Interfaces** – boundaries across which two independent entities meet and interact or communicate
- Relations
  - **Is part of** – a relationship between the sub-module (part) and the aggregate module (whole)
  - **Depends on** – a dependency relationship between two modules (specific module styles elaborate what dependency is meant)
  - **Is a** – a generalization/specialization relationship between a more specific module (child) and a more general module (parent)
- Purpose
  - Explains the functionality of the system and the structure of the code base
  - Facilitates impact analysis
  - Supports requirements traceability analysis
  - Provides blueprint for construction of the code, supports planning incremental development, and definition of work assignments, implementation schedules, and budget information
  - Shows the structure of information to be persisted (data model)

# The Module View Notation in UML

CMU SEI Views & Beyond



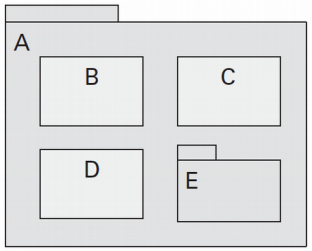

Interface is an element connecting two or more other elements, defining how these interact and what data is exchanged

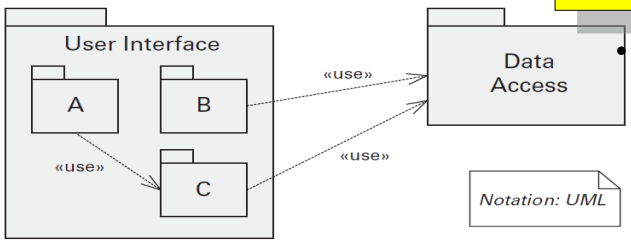
Pictures © CMU SEI & Pearson Education, Inc.

# Module Viewpoint – Different Styles

CMU SEI Views & Beyond

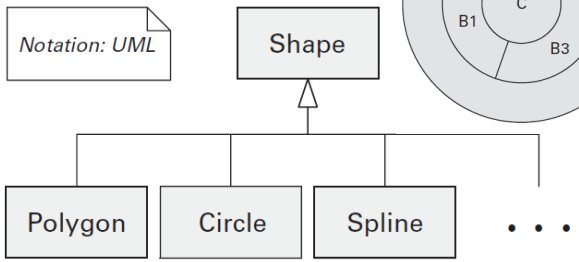
- **Decomposition style** – shows source code structure as *decomposition hierarchy* of modules
  - each module can only have one parent

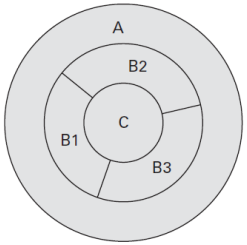


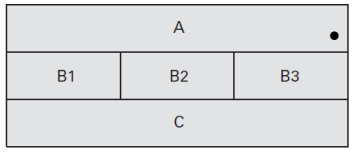




- **Uses style** – shows how modules *depend* on one another (for performing change-impact analysis and supports incremental development)

- **Generalization style** – shows which module is a *generalization* of the other (describes commonality)
  - modules can be abstract (with incomplete implementation)

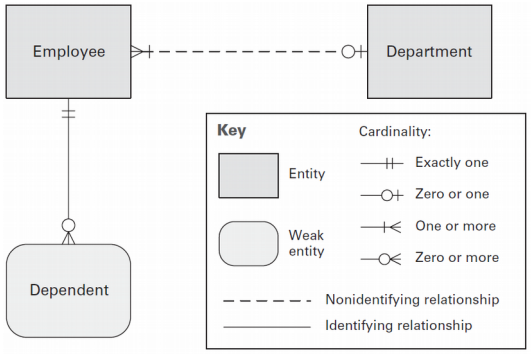



- **Layered style** – divides a system into groups of modules that provide cohesive responsibilities (layers) and relate to each other unidirectionally by the *allowed-to-use* relation (supports portability and modifiability)
  - each module allocated to exactly one layer; at least two layers

- **Aspects style** – shows special modules, called aspects, responsible for crosscutting concerns (supports modifiability)
  - aspects may not crosscut themselves

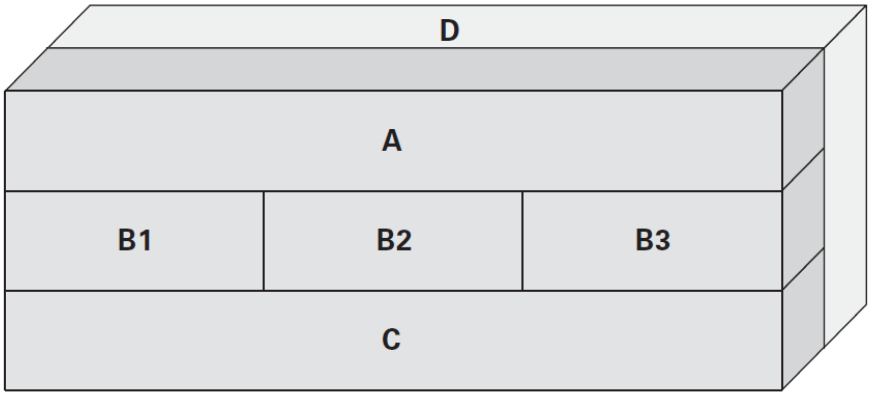



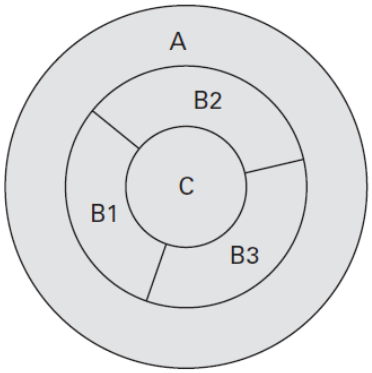

- **Data model style** – describes the structure of the data used in the system in terms of data entities and their relationships
  - avoid functional dependencies

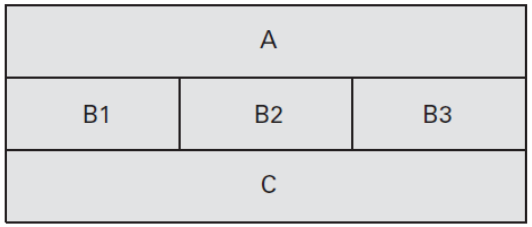
Pictures © CMU SEI & Pearson Education, Inc.

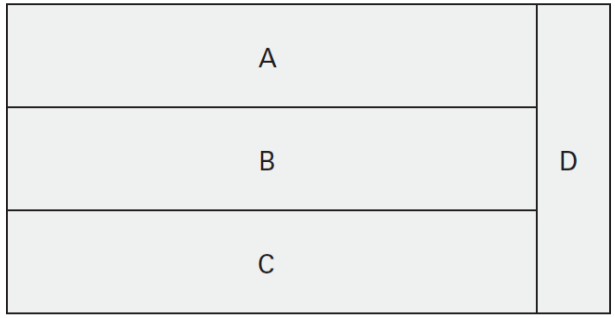
# Module Viewpoint
# Layered Style – Diagram Variants

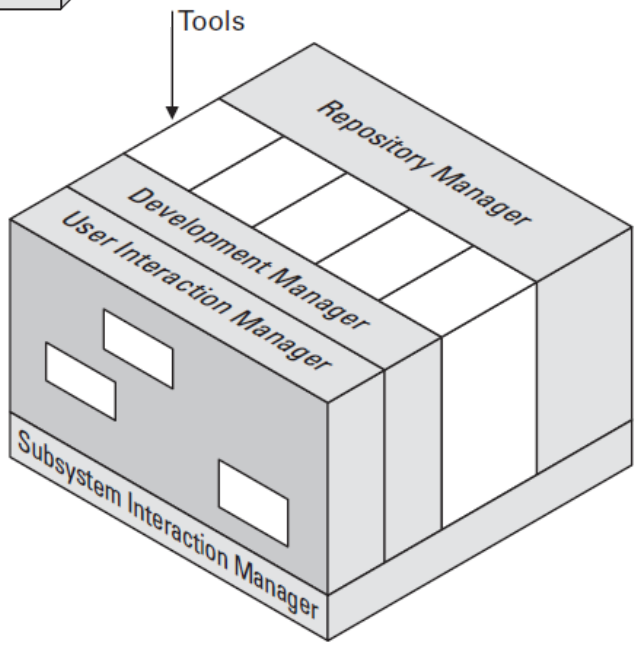
CMU SEI Views & Beyond

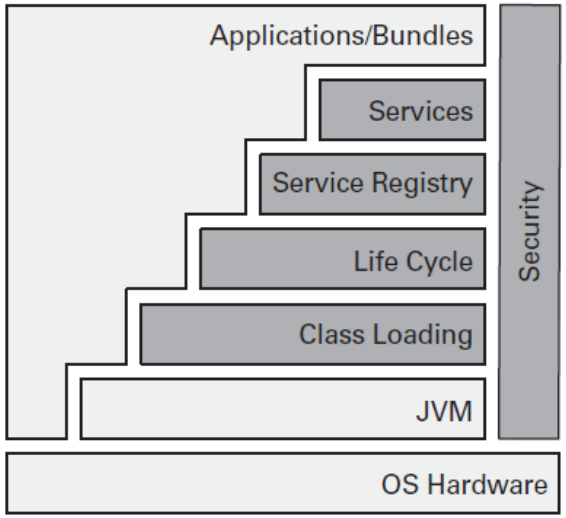


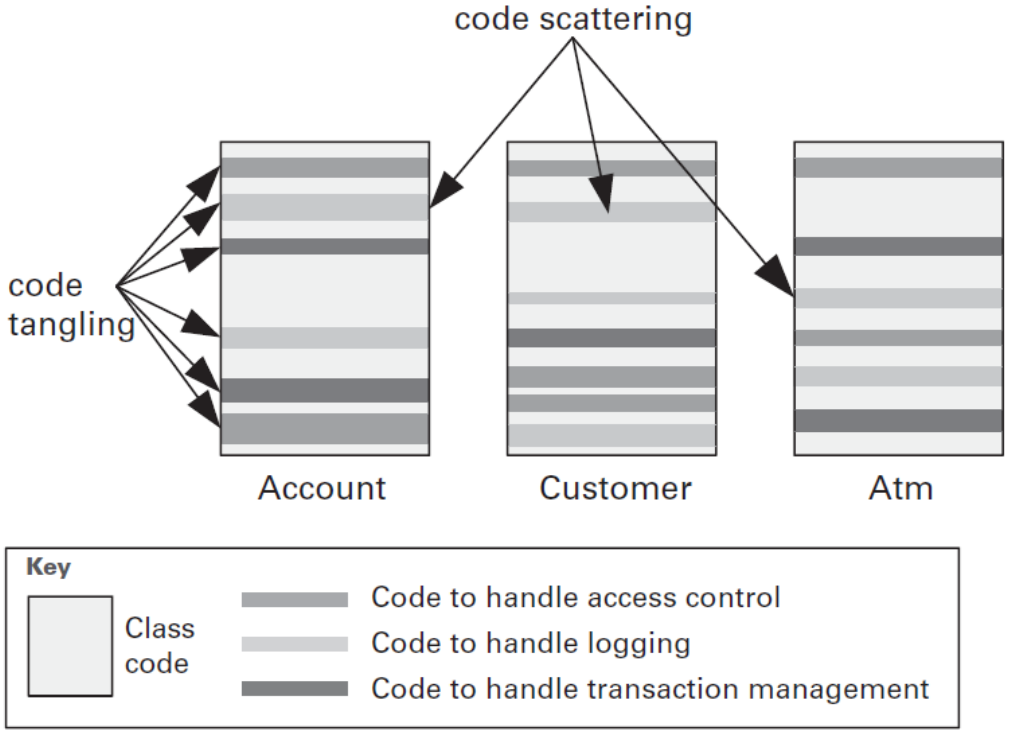



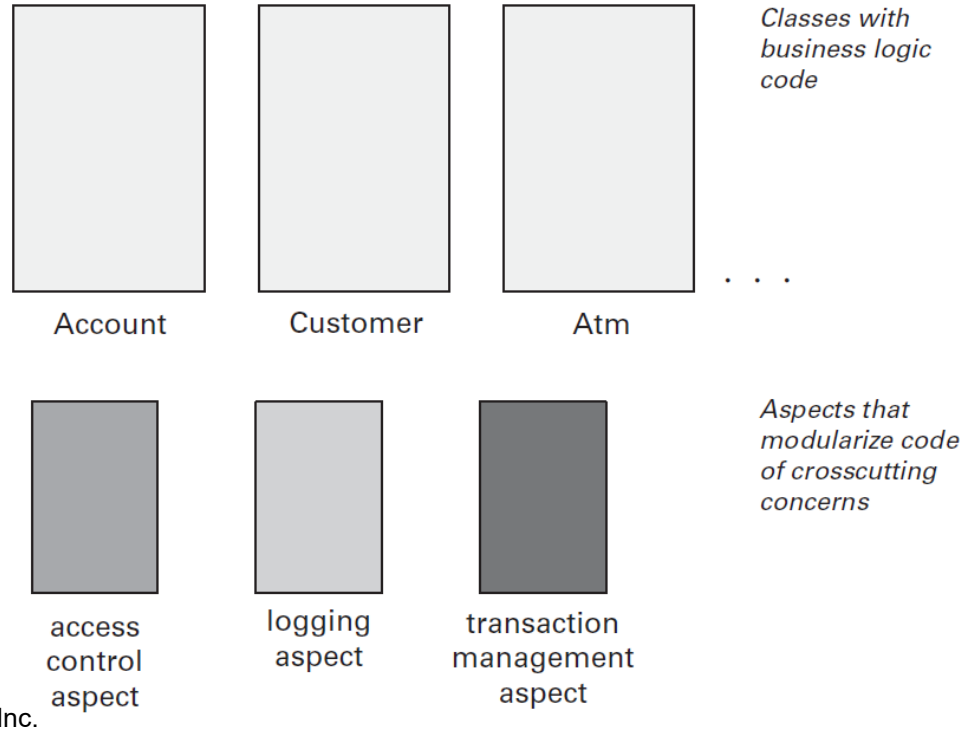

Pictures © CMU SEI & Pearson Education, Inc.

# Module Viewpoint
# Aspect Style

CMU SEI Views & Beyond

- Crosscutting concerns



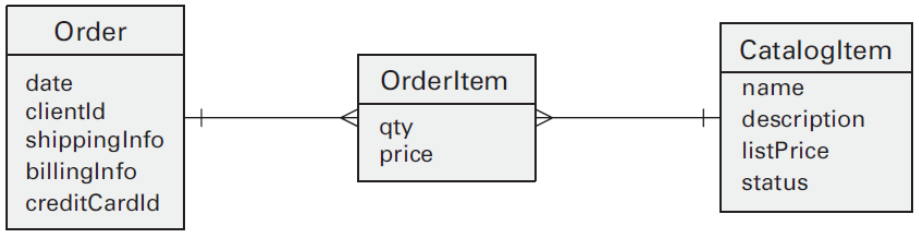



# Module Viewpoint
# Data Model Style

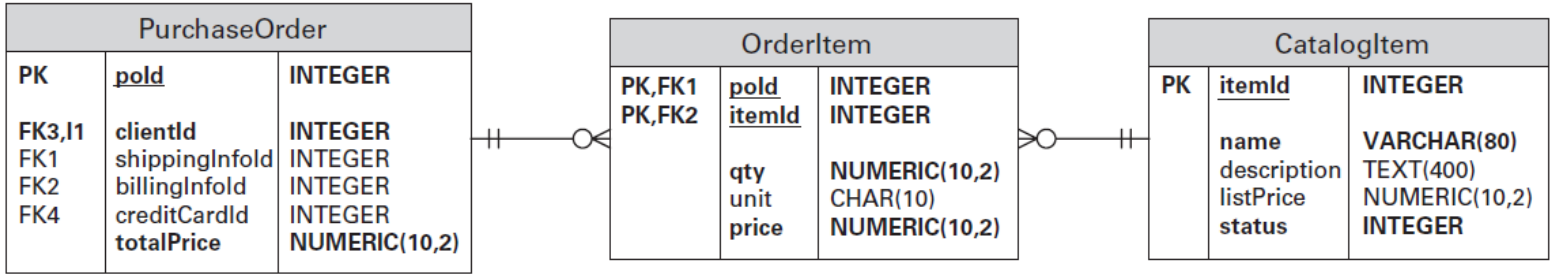
CMU SEI Views & Beyond

- Documentation of different stages of the data model evolution
  - **Conceptual** – the conceptual data model abstracts implementation details and focuses on the entities and their relationships as perceived in the problem domain


  - **Logical** – the logical data model is an evolution of the conceptual data model toward a data management technology (such as relational databases)



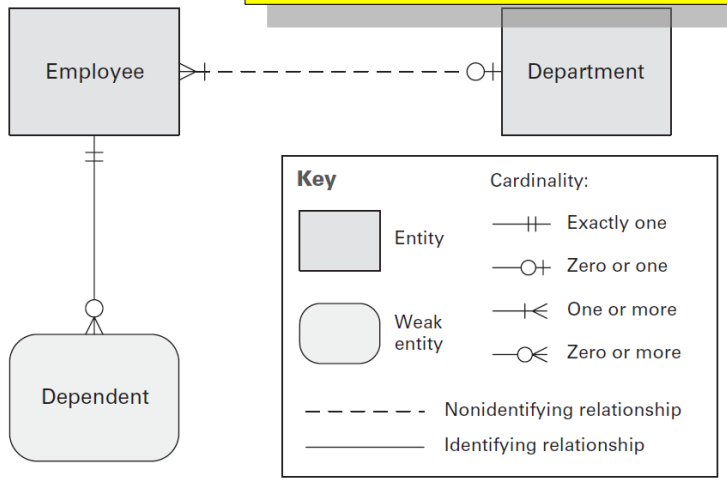

  - **Physical** – the physical data model is concerned with the implementation of the data entities. It incorporates optimizations that may include partitioning or merging entities, duplicating data, and creating identification keys and indexes



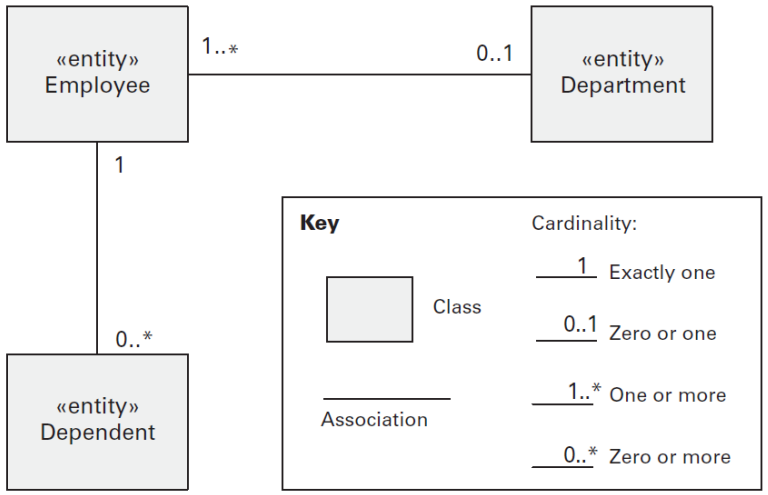
# Module Viewpoint
# Data Model Notations

CMU SEI Views & Beyond

- Entity Relationship Diagram (ERD) Variations
  - Peter Chen's entity-relationship diagram notation (Chen 1976)
  - Crow's foot entity-relationship diagram notation
  - IDEF1X



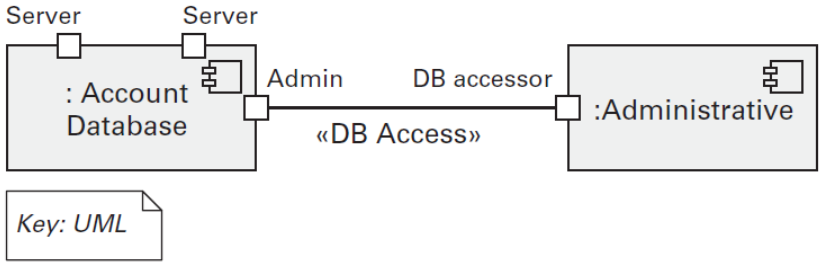

- UML class diagram



# The Component-and-Connector Viewpoint Overview

CMU SEI Views & Beyond

- Component-and-connector views describe
  - structures consisting of elements that have run-time presence (processes, components, data stores, ...)
  - the pathways of interaction (communication links/protocols, data flows, access to shared storage, …)
- Component-and-connector views show **instances, not types**
  - style-specific types are defined in a style guide and application specific types are described in the view documentation
- **Components have** interfaces, called **ports** and **connectors have** interfaces, called **roles** – attachments can be made only between compatible ports and roles
- Components can be attached only to connectors, not other components an vice versa
- Connectors
  - may have more than two roles (need not be binary)
  - cannot appear in isolation, they must be attached to a component
  - can, and often do, represent complex forms of interaction



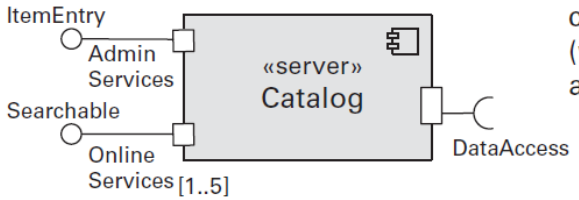
# The Component-and-Connector Viewpoint Summary

CMU SEI Views & Beyond

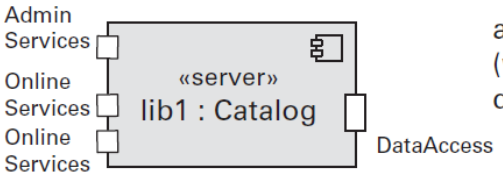
- Elements
  - **Components** – principal processing units and data stores with a set of ports
  - **Connectors** – pathways of interaction between components with a set of roles
- Relations
  - **Attachments** – component ports are associated with connector roles to yield a graph of components and connectors
  - **Interface delegation** – in some situations component ports are associated with one or more ports in an "internal" sub-architecture; similarly for the roles of a connector
- Purpose
  - Shows how the system works
    - system's principal executing components, and their interactions, principal data-stores, changes of structure during execution
  - Guides development and deployment by specifying the structure and behavior of run-time elements
    - used protocols of interactions, which parts of the system are replicated, how does data flow through the system, what parts of the system run in parallel
  - Helps to reason about run-time system quality attributes, such as performance, reliability, and availability

# The Component-and-Connector Viewpoint Notation in UML

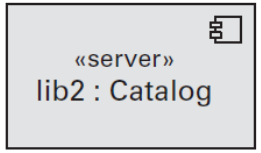
CMU SEI Views & Beyond



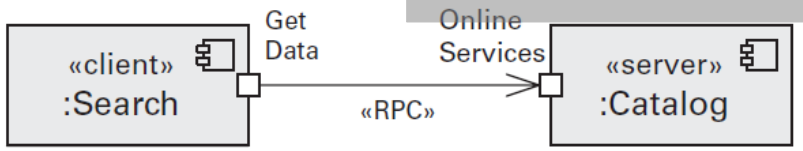

component type (with UML ports and interfaces)



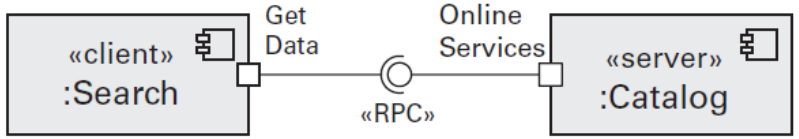

a component instance (with ports explicitly documented)

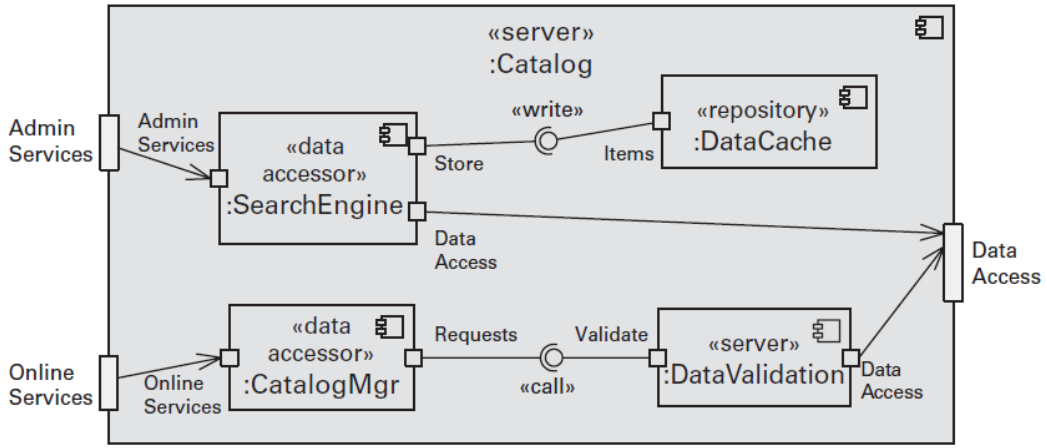



a component instance




adding navigable end to a connector

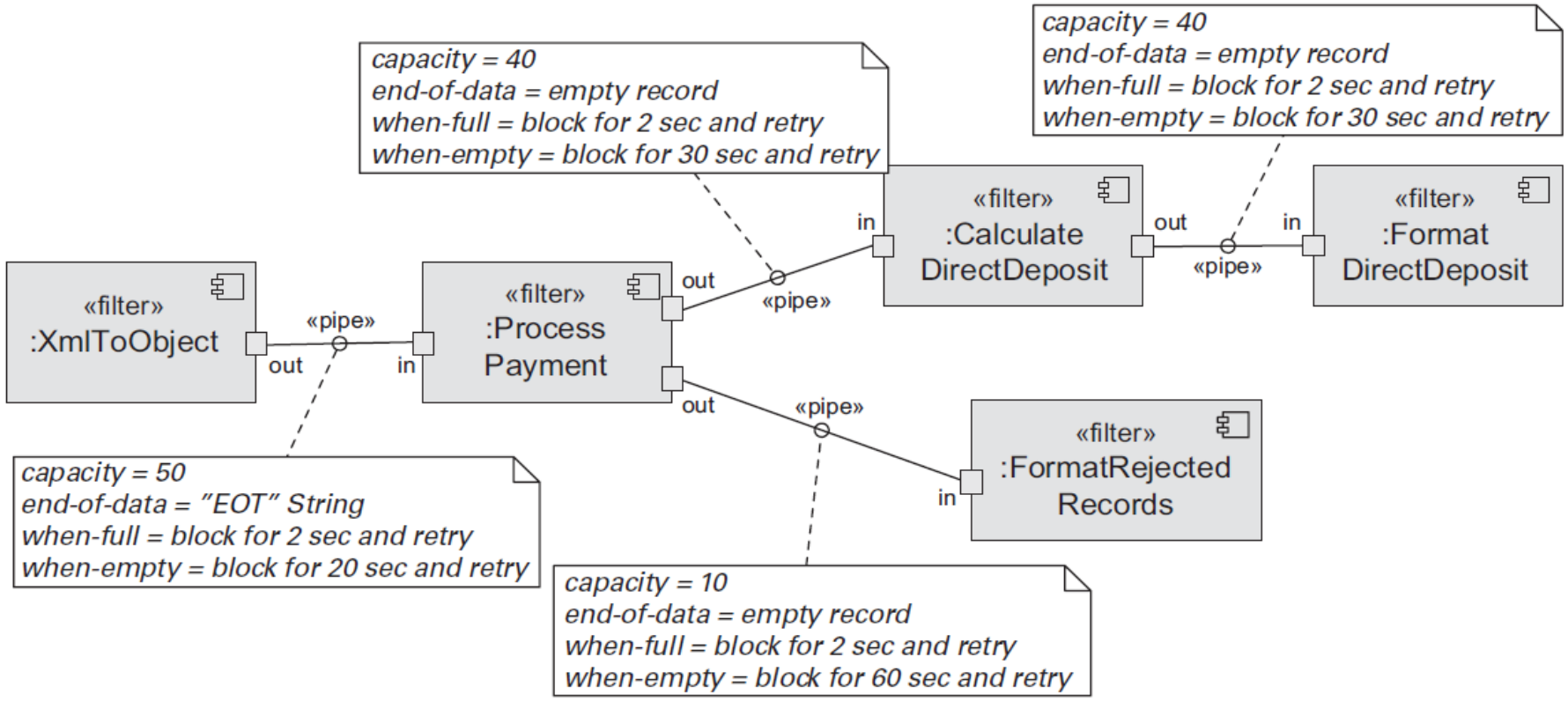



Architecture elements can have both *provided* and *required* interfaces

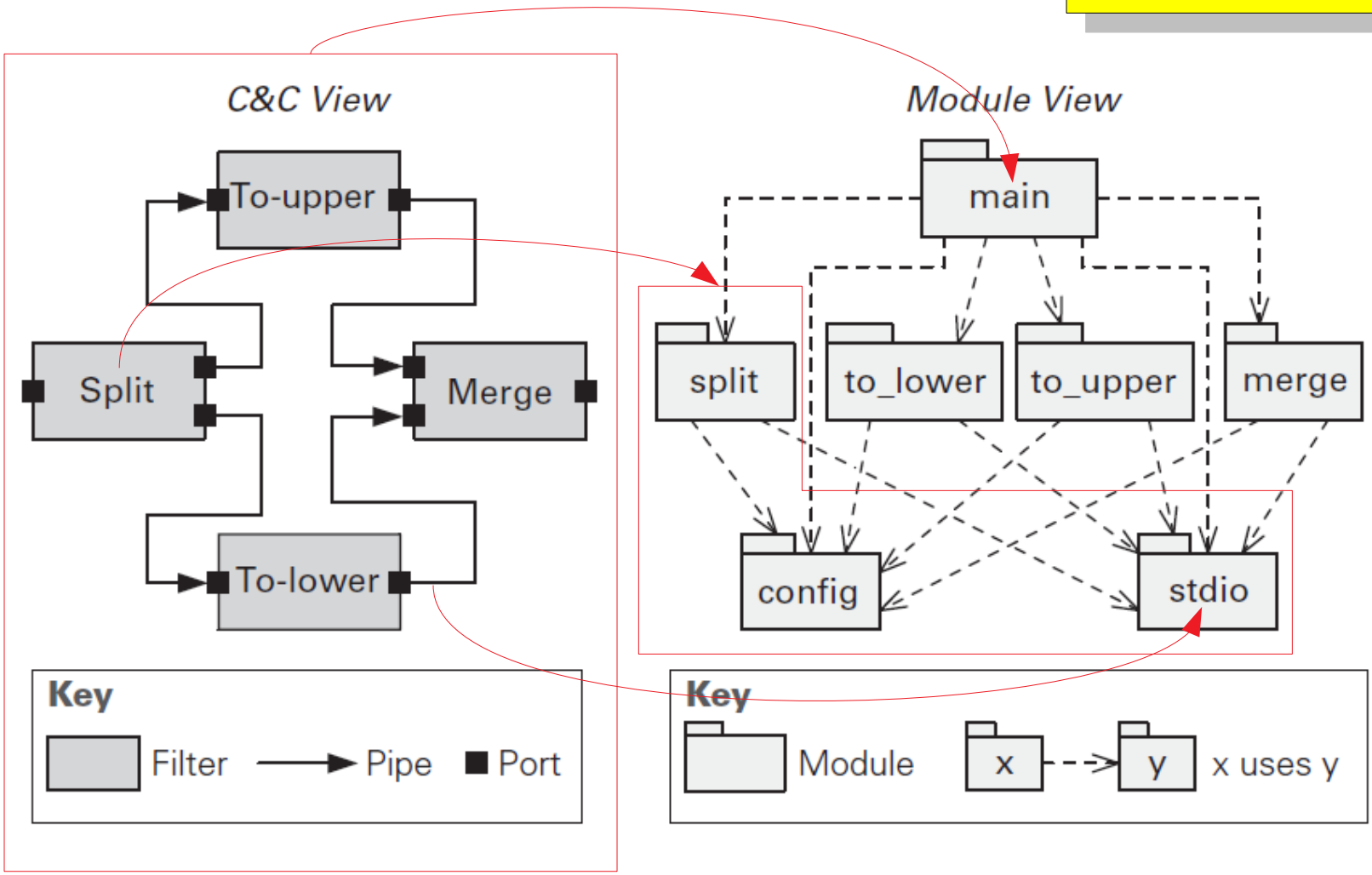
# The Component-and-Connector Viewpoint Notation in UML

CMU SEI Views & Beyond



# The Component-and-Connector Viewpoint
# Connection to other Views






Pictures © CMU SEI & Pearson Education, Inc.

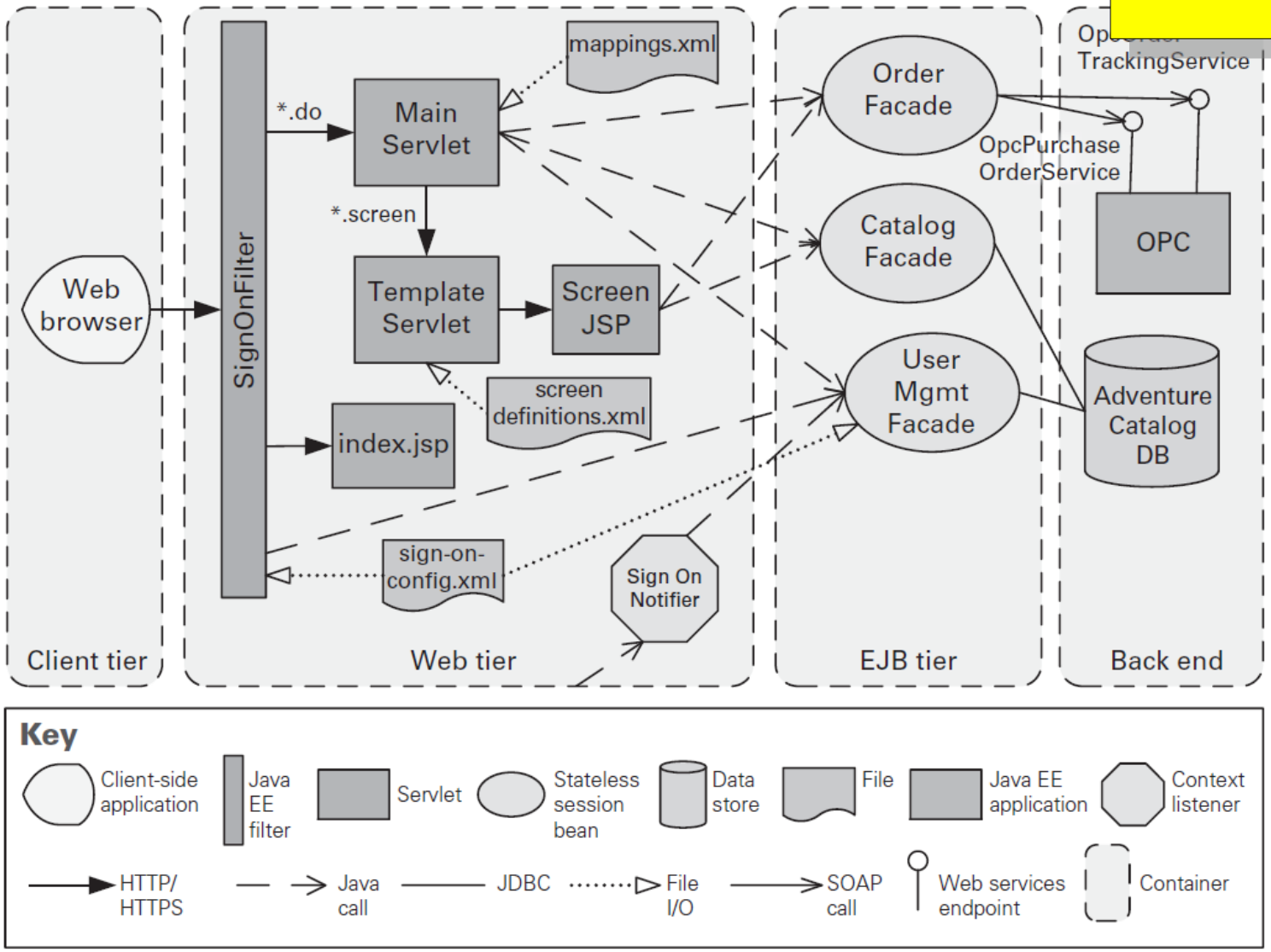
# The Component-and-Connector Viewpoint Checklist

CMU SEI Views & Beyond

- Define component-and-connector element and connector types according to the elements of architectural style (e.g. data flow, call-return, event-based, repository, etc.)
- Always show a component's ports explicitly and always attach a connector to a port of a component, not directly to a component (if it is not clear that it is valid to attach a given port with a given role, provide a justification in the rationale section for the view)
- Make clear which ports are used to connect the system to its external environment
- Data flow and control flow models are best thought of as projections of component-and-connector views, but they are not views because the arrows represent usage of the connectors (which define more completely the components' interactions)
- Show the mapping between components in a component-and-connector view and their respective implementation units in module views (in general, this is many-to-many mapping)
- For components that run as concurrent processes or threads, it's important to document how these processes or threads are scheduled or preempted, and how access to shared resources is synchronized
- Component-and-connector views can be structured in tiers – which are logical groupings of components

# Component-and-Connector View Example

CMU SEI Views & Beyond



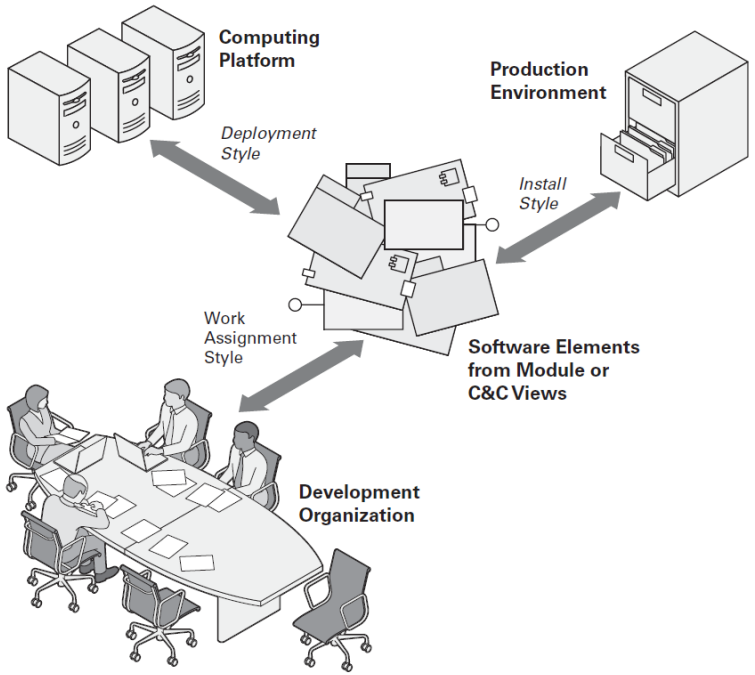
# The Allocation Viewpoint Overview

CMU SEI Views & Beyond

- Describes mapping of software units to elements of environment (e.g. hardware, file systems or development teams)

- Elements
  - **Software element** (with properties required of the environment)
  - **Environmental element** (with properties provided to the software)

- Relations
  - **Allocated-to** – a software element is mapped (allocated to) an environmental element (properties are dependent on the particular style)

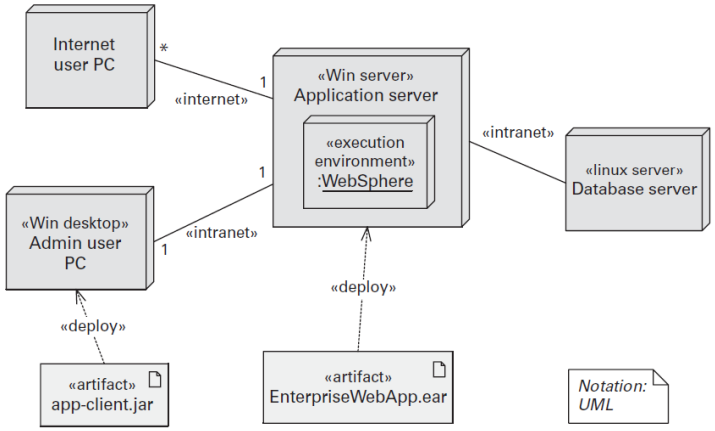



- Purpose
  - Shows the tools and environments in which the software is developed
  - Helps to carry out various tasks (edit, build, package, deploy, configure, ...)
  - Supports analyzing performance, availability, reliability, security and run-time dependencies
  - Helps planning and managing resource allocations, assigning responsibilities
  - Basis for work breakdown structures and for budget and schedule estimates

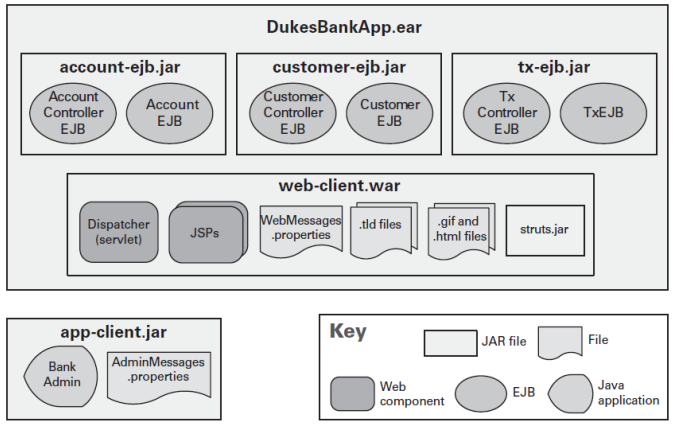
# Allocation Viewpoint Styles

CMU SEI Views & Beyond

- **Deployment style** – maps components and connectors to the hardware elements
  - allocation topology is unrestricted, but the required properties of the software must be satisfied by the provided properties of the hardware

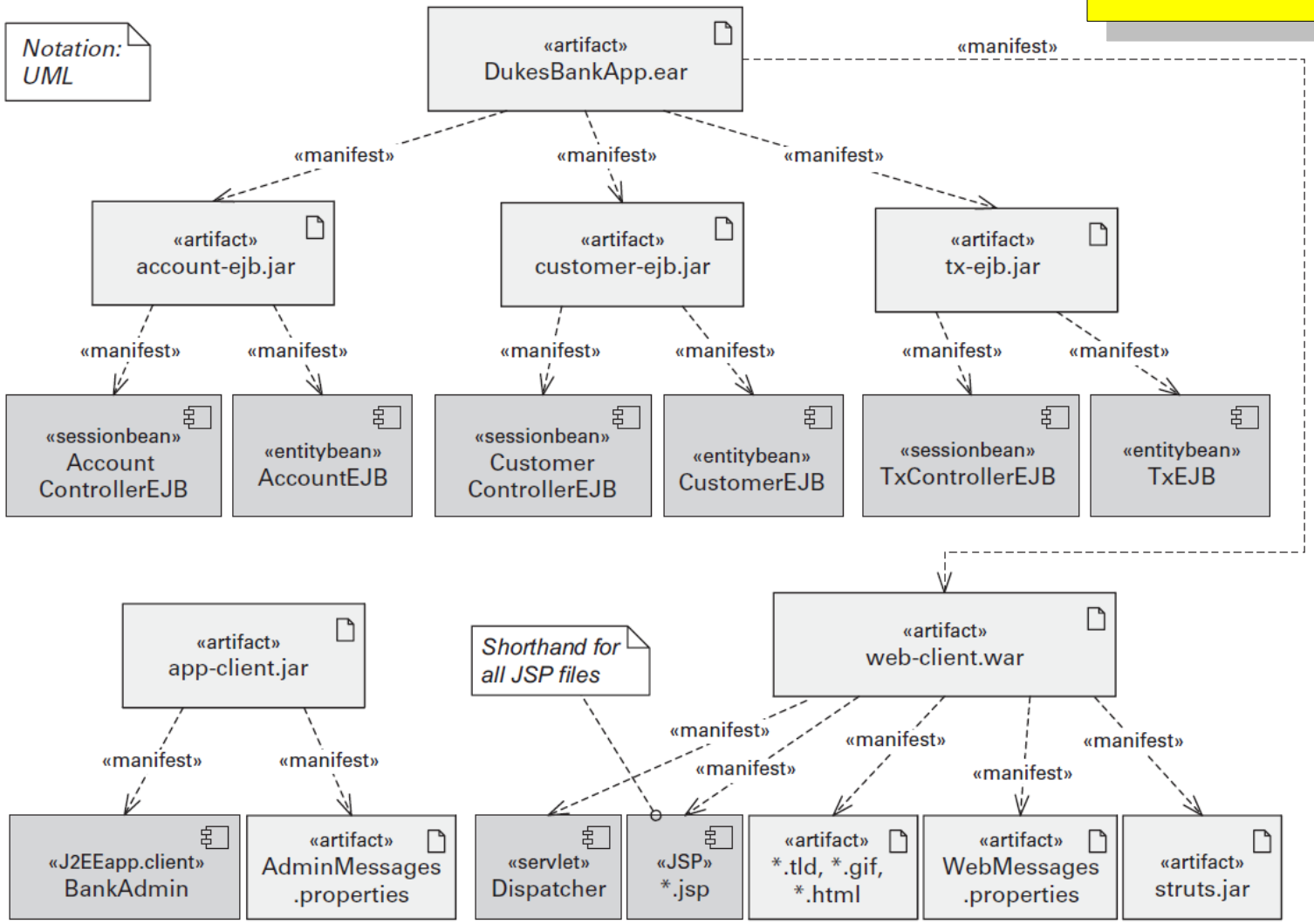



- **Install style** – maps components to a file system in the production environment
  - files and folders are organized in a tree structure, follows an is-contained-in relation



# Allocation Viewpoint
# Install Style Notation in UML

CMU SEI Views & Beyond



# Allocation Viewpoint Styles

CMU SEI Views & Beyond

- **Work assignment style** – maps modules to development teams (describes responsibilities for elements of the work-breakdown structure)
  - usually one module is allocated to one organizational unit
  - ways of work assignment
    - **platform(s)** – reusable core assets vs.applications
    - **competence-centers** – work is allocated depending on the technical or domain expertise
    - **open-source style** – many independent contributors with common integration strategy
    - **process-steps** – work is allocated according to the phases of the software development process)
    - **release-based** – different releases are allocated to different teams

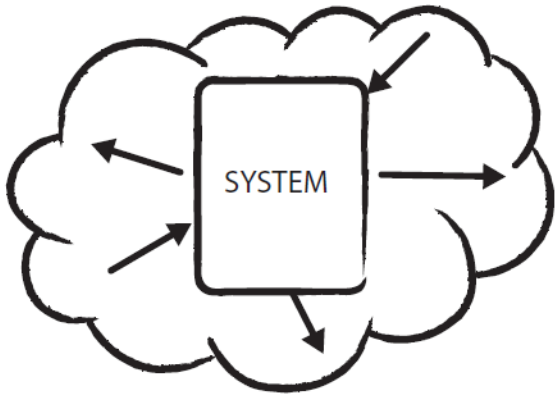
| ECS Element (Module) | | |
|---|---|---|
| **Segment** | **Subsystem** | **Organizational Unit** |
| Science Data Processing Segment (SDPS) | Client | Science team |
| | Interoperability | Prime contractor team 1 |
| | Ingest | Prime contractor team 2 |
| | Data Management | Data team |
| | Data Processing | 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 |
| . . . | . . . | . . . |

- **Implementation style** – (similar to install style) maps modules to a development infrastructure
- **Data stores style** – (similar to deployment style) maps data entities to the hardware of the data servers (e.g. distribution of tables to servers, geographic distribution of the databases, data warehouses and the data stores that feed them)
- **Coordination style** – to align the architecture and the development organization by representing complexity and uncertainty in the communications (e.g. Communication Capacity Matrix (CCM))

# Beyond the Views

CMU SEI Views & Beyond

- Documenting context diagrams
  - A context diagram establishes the boundaries for the information contained in a view (“context” means an environment with which the part of the system interacts)
  - A top-level context diagram
    - is a context diagram in which the scope is the entire system – it defines what is and is not in the system, thus setting limits on the architect’s tasks
    - makes a good first introduction to a system

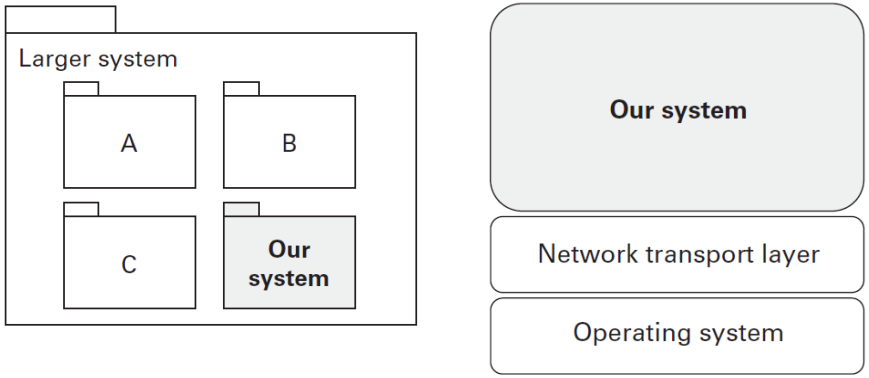


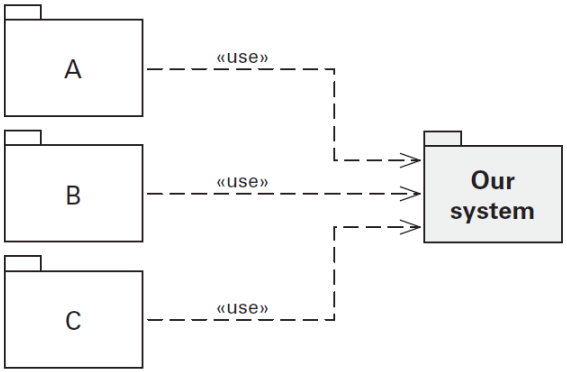

- Documenting variation points
  - Some architectures provide built-in variation points to facilitate building a family of similar but architecturally distinct systems, other architectures are dynamic, in that the systems they describe change their basic structure while they are running
- Documenting architectural decisions
  - Why we made architectural decisions the way we did is just as important as the results of those decisions – how to record the rationale behind your design
- Combining views
  - Prescribing a given set of rigidly partitioned views is naive; there are times and good reasons for combining two or more views into a single combined view

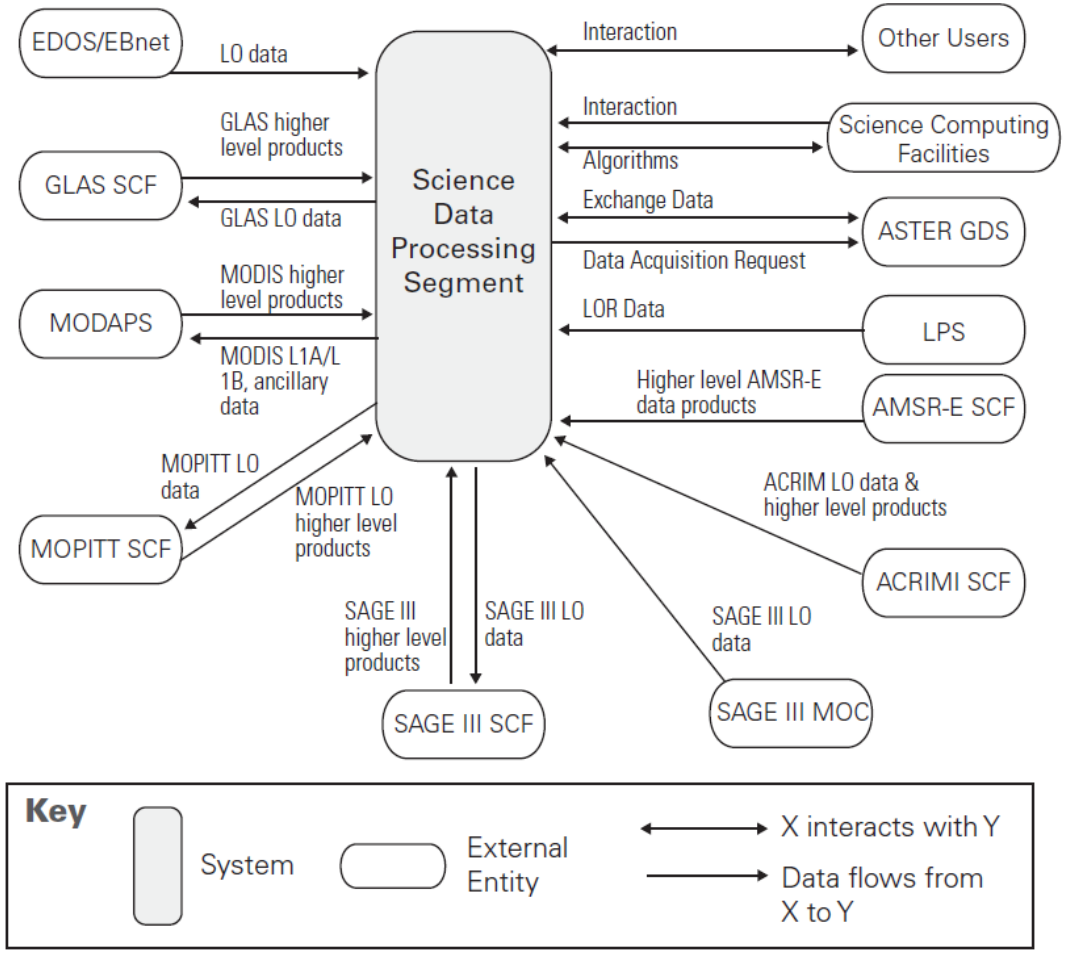
# Documenting Context Diagrams

CMU SEI Views & Beyond

- Describe the context of the system being developed using the vocabulary of the view that you're documenting



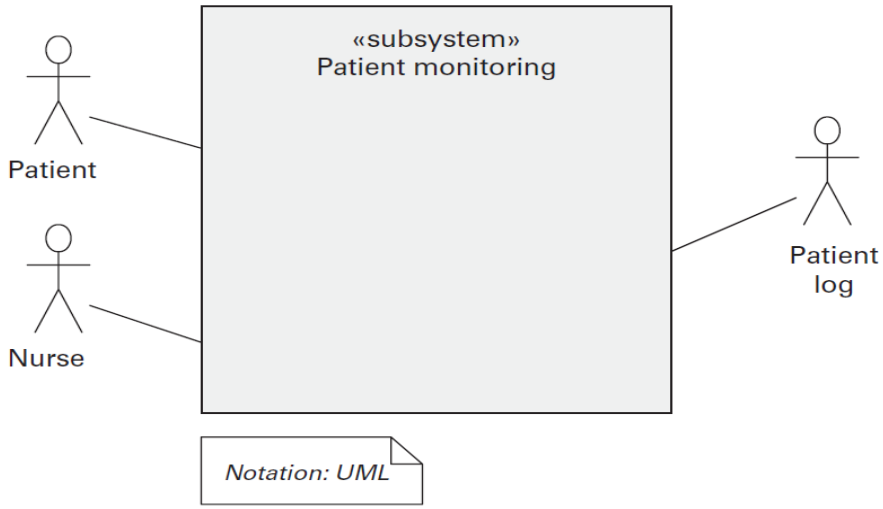



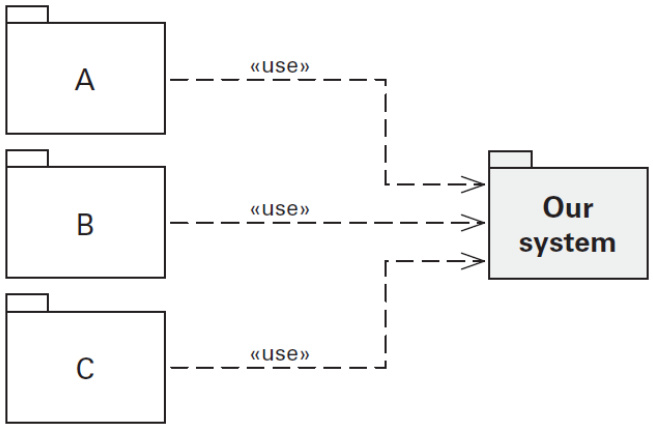
# Documenting Context Diagrams in UML

CMU SEI Views & Beyond

- Combination of Use Case and Class diagrams




- Package diagram

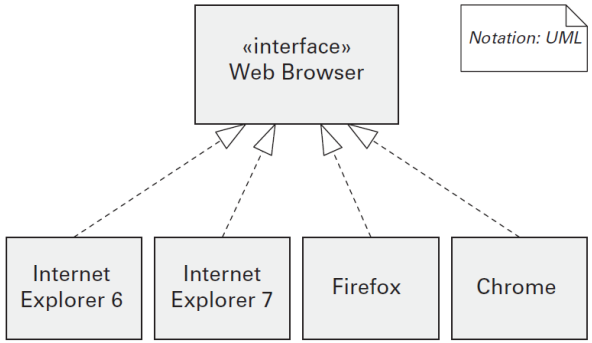


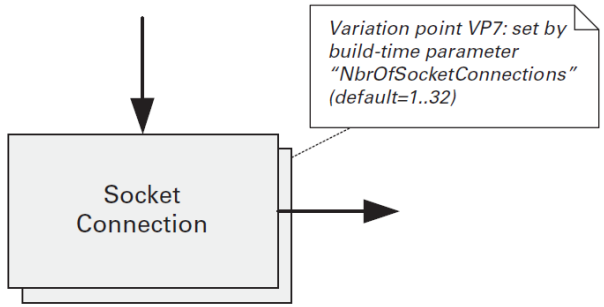
# Documenting Variation Points

CMU SEI Views & Beyond

- Need for variability
  - some set of decisions has not yet been made, but options have been explored
  - the architecture is prepared for envisioned future changes
  - there's need to provides basic functionality that can be extended easily
  - the architecture is for a family/collection of systems and contains explicit places where configurations and extensions to the reference architecture can occur

- Variation points should be documented in two ways
  - their existence should be noted in the appropriate places throughout the view (primary presentation, element catalog, context diagram, and so on) for the view in which they are visible
  - the variation point should be explained in the view's variability guide



# Documenting Variation Points

CMU SEI Views & Beyond

- Description of the variation point
  - What decision has been left open by this variation point should be meaningful to the stakeholders, e.g. choosing different implementations results in different feature behavior)
- Available options and their effects
  - What is the range of choices available to exercise this variation point and what are the effects of each
- Condition of applicability
  - What conditions must be met for a variation point to apply
- The binding time of an option
  - Possible binding times include design time, compile time, link time, or run-time
- How the option is exercised
  - What has to be done to choose an option of the variation point (set a build-time parameter or replace one implementation of a module with another) – this section is the “how-to” guide
- Dependencies among variation point options
  - Does chosen option for one variation point, constrain other choices

# Documenting Architectural Decisions

CMU SEI Views & Beyond

- Document the decision (and explain the reasoning that lies behind), if
  - it **has an important effect** on the system (that will be difficult to undo)
  - the design **team spent significant time and effort evaluating options** before making a decision
  - the **decision is complex or confusing** (seems not to make sense at first)
  - it **seems unusual or unexpected** (because these are very likely to be broken by mistake)

- Essential information about a key architectural decision
  - *Issue* – architectural design issue being addressed by the decision
  - *Decision* – the solution chosen
  - *Assumptions* – based on which the decision is being made (cost, schedule, technology, …)
  - *Alternatives* – alternative solutions/options considered
  - *Argument* – describing why given alternative was selected (e.g implementation cost, total cost of ownership, time to market, availability of development resources)
  - *Implications* – describe the decision's implications
  - *Related things*
    - other decisions, related requirements, affected architecture elements and external artifacts (e.g. budgets, schedules)

The life of a software architect is a long (and sometimes painful) succession of sub-optimal decisions made partly in the dark

P. Kruchten

# Documenting Architectural Decisions Classification

P. Kruchten (2004)

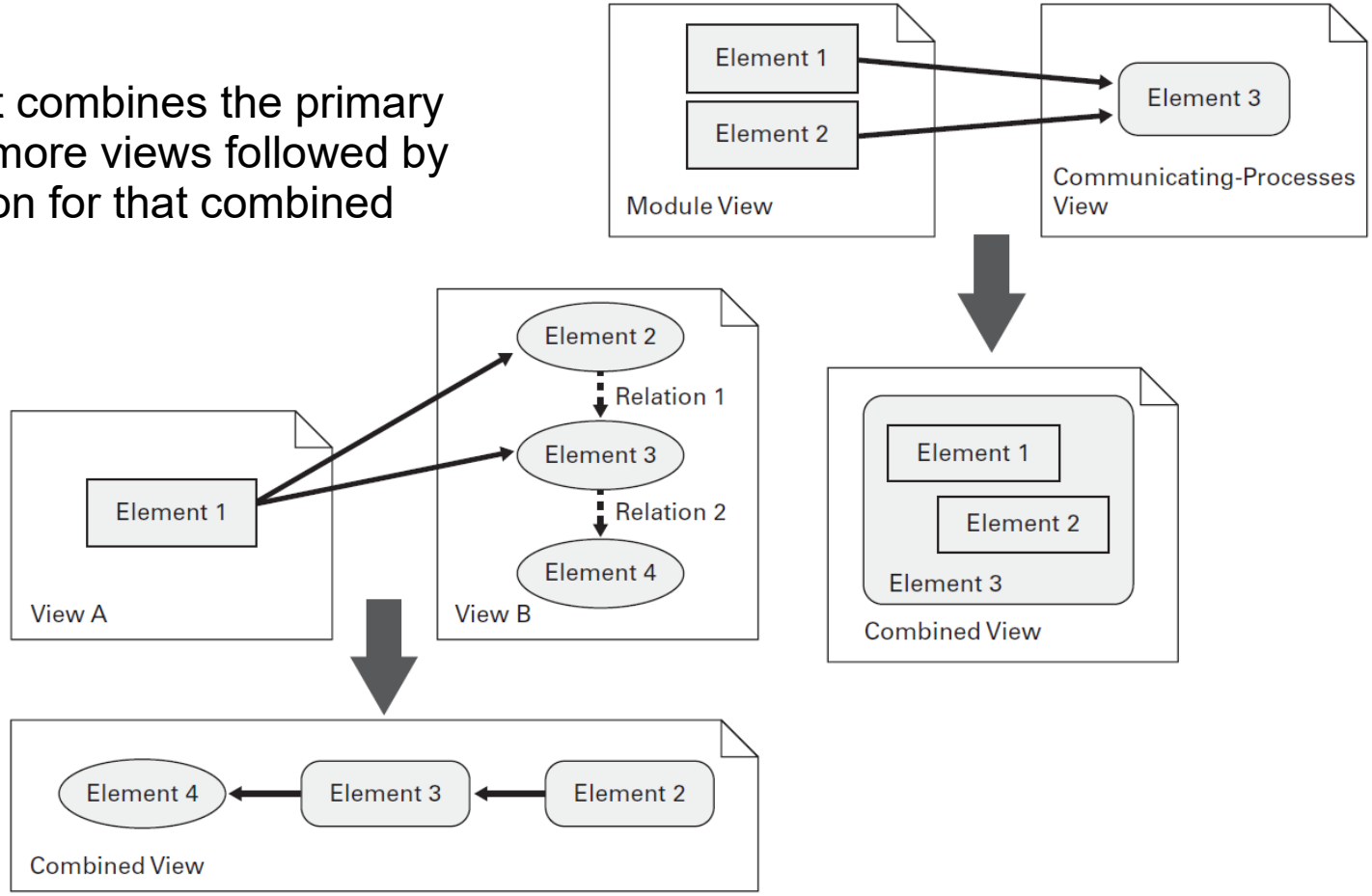
- Kinds of Architectural Design Decisions
  - Existence Decisions (*ontocrises*)
    - Structural and Behavioral decisions
  - Ban or non-existence decisions (*anticrises*)
  - Property Decisions (*diacrises*)
    - Constraints, Design rules, and Guidelines
  - Executive Decisions (*pericrises*)
    - Organizational, Process (methodological), Technology and Tool decisions
- Attributes of Architectural Design Decisions
  - *Epitome* (the Decision itself)
  - Rationale ("why")
  - Scope
  - State (idea, rejected, tentative/challenged, decided, approved)
  - Author, Time-Stamp, History
  - Categories (usability, security, …)
  - Cost, Risk
- Relationships between Architectural Design Decisions
  - Constraints
  - Forbids (Excludes)
  - Enables
  - Subsumes
  - Conflicts with (mutually excluding)
  - Overrides
  - Comprises (is made of, decomposes into)
  - Is bound to (strong)
  - Is an alternative to
  - Is related to (weak)
  - Dependencies
- Relationship with External Artifacts
  - Traces to
  - Does not comply with

# Combining the Views



- A combined view is a view that contains elements and relations that come from two or more other views
- An overlay is a view that combines the primary presentations of two or more views followed by supporting documentation for that combined primary presentation
- A hybrid style is the combination of two or more existing styles

  (element and relation types of the constituent styles can “meld” into new types with new properties)

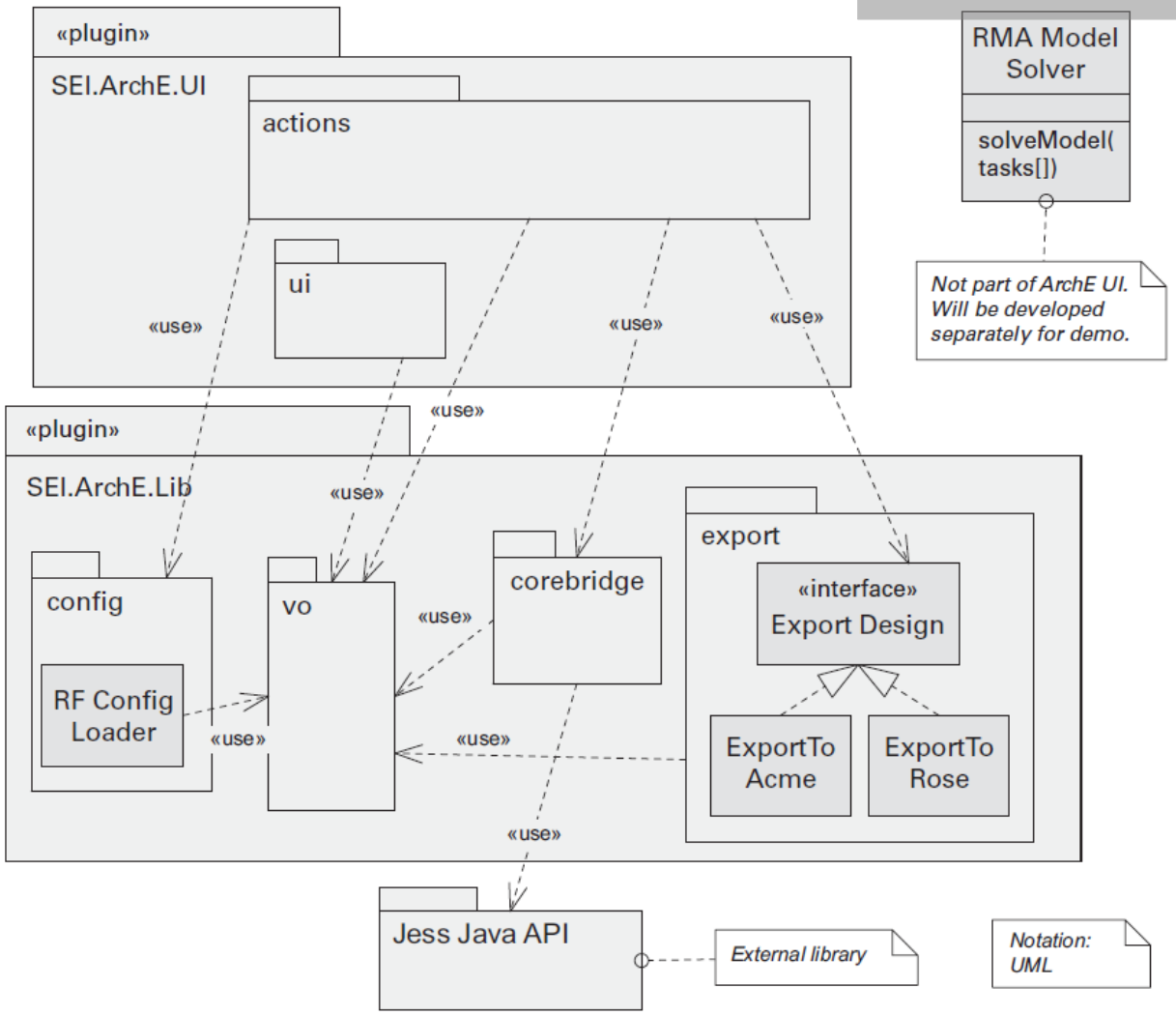


# Combining the Views Example



decomposition-
uses-
generalization

combined view

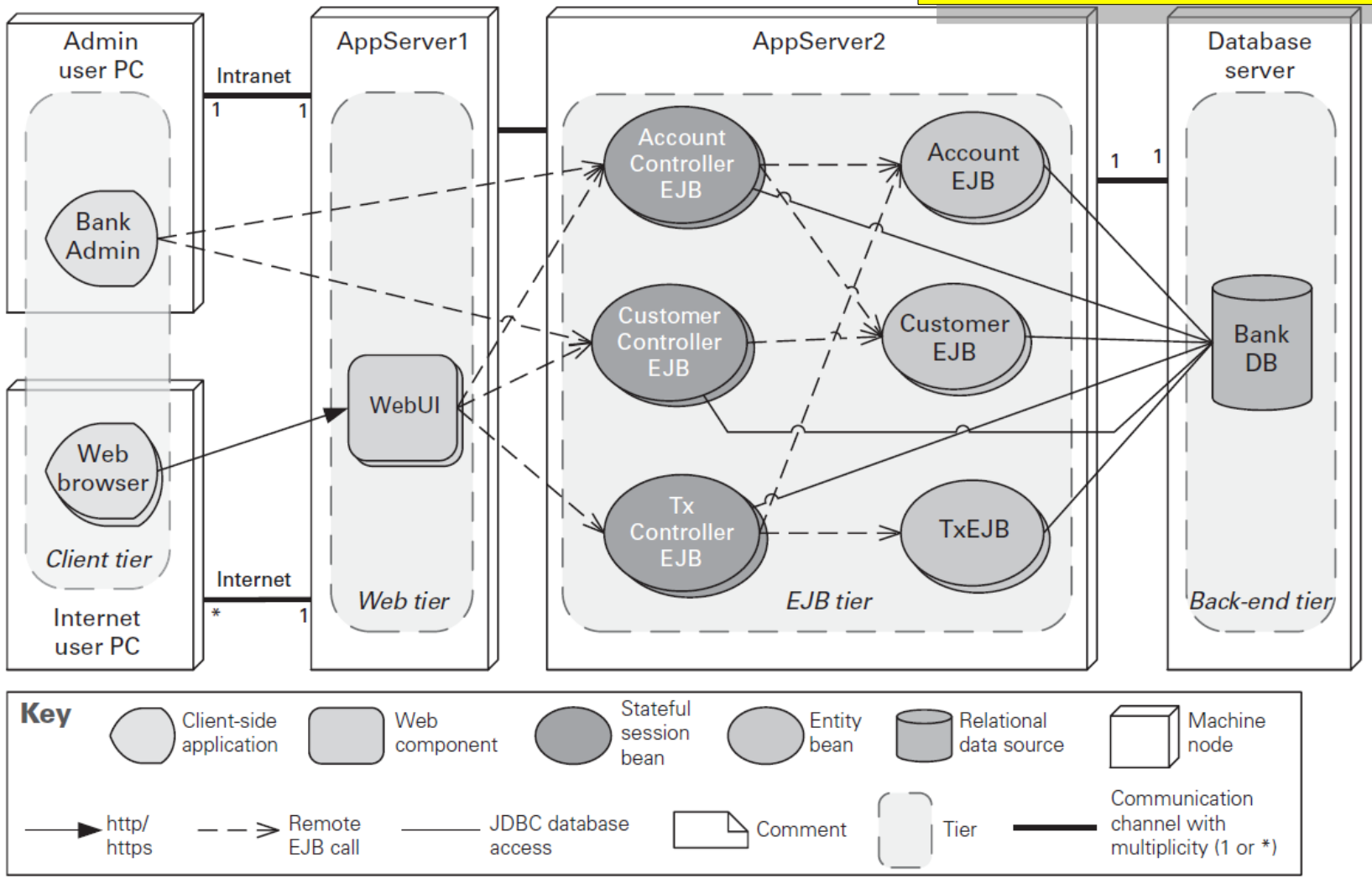


# Combining the Views Example

CMU SEI Views & Beyond

multi-tier client-server deployment

combined view

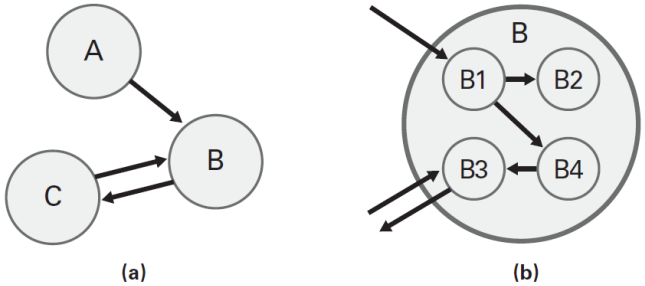


# Descriptive Completeness

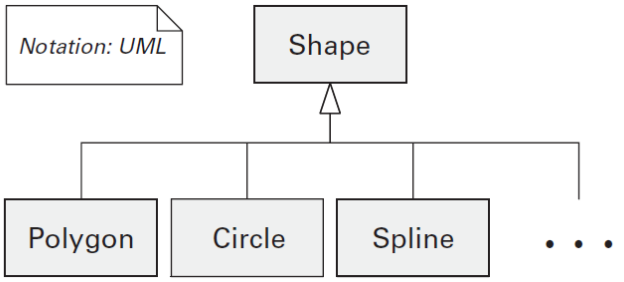


- There may be good reasons to omit some details from architecture descriptions

- Refinement – gradual disclosure of more-detailed information
  - decomposition refinement reveals internal substructure
  - implementation refinement replaces elements with different, more implementation specific elements




- Documentation may or may not show all elements and relations
  - when some elements and relations are suppressed, the view documentation should make it clear to the reader
    - use ellipses ("…") to indicate that there are other elements
    - use a comment box in the diagram to explain that not all elements are being exhibited




  - why omit some elements and relations in a view
    - it's early in the design – all things are not yet known
    - want to focus on the most important parts of the view (reduce clutter in diagrams)

# Advanced Techniques

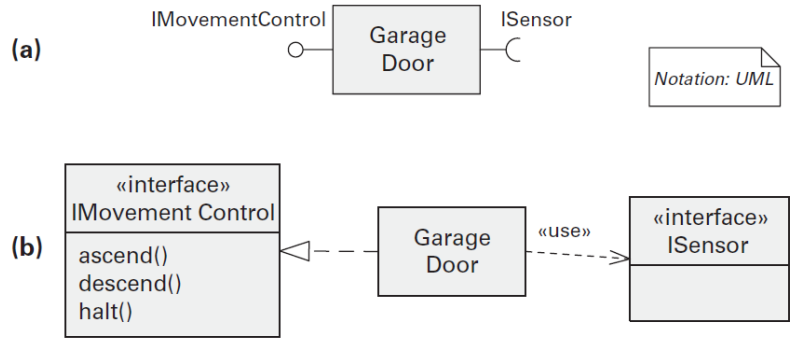
CMU SEI Views & Beyond

- Documenting Software Interfaces
- Documenting Behavior
- Requirements Viewpoint
- Choosing the Views
- Building the Documentation Package
- Architecture Overview Presentation

# Documenting Software Interfaces

CMU SEI Views & Beyond

- An interface is a boundary across which two elements meet and interact or communicate with each other
- An interface document is a specification of what an architect chooses to make publicly known about an element in order for other entities to interact or communicate with it
- A resource of an interface represents a function, method, data stream, global variable, message end point, event trigger, or any addressable facility within that interface




**Interface Documentation**

Section 1. Interface Identity
Section 2. Resources
*For each resource:*
- Syntax
- Semantics
- Error Handling

Section 3. Data Types and Constants
Section 4. Error Handling
Section 5. Variability
Section 6. Quality-Attribute Characteristics
Section 7. Rationale and Design Issues
Section 8. Usage Guide

# Some Principles about Interfaces

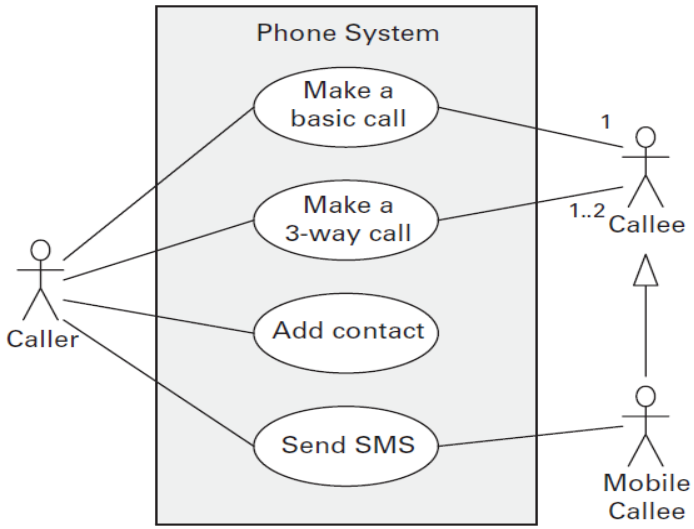
CMU SEI Views & Beyond

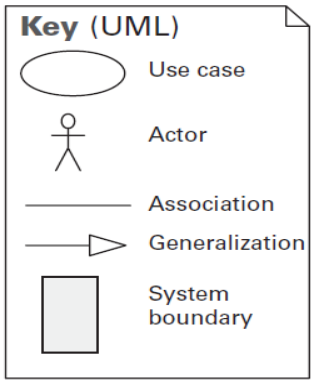
- All elements have interfaces
  - All software elements described in any view interact with their environment – architect decides what to document
- An element's interface is separate from its implementation
- An element can have multiple interfaces
  - Each interface contains a separate collection of resources (functions, data, message end points, event triggers, ...) that have a related logical purpose, or represent a role that the element could fulfill
  - Multiple interfaces provide a separation of concerns – a specific actor might require only a subset of the resources
  - Evolution can be supported by keeping the old interface and adding a new one
- Elements not only provide interfaces but also require interfaces
  - An element interacts with its environment by making use of resources or assuming that its environment behaves in a certain way – without these required resources, the element cannot function correctly
- Multiple actors may interact with an element through its interface at the same time (if interface allows multiple concurrent interactions)
- Interfaces can be extended by generalization
  - Examples of resources often shared by several interfaces include: an initialization operation, a set of exceptions, …
- Sometimes it's useful to distinguish interface types from interface instances in the architecture (if components can provide multiple instances of the same interface)

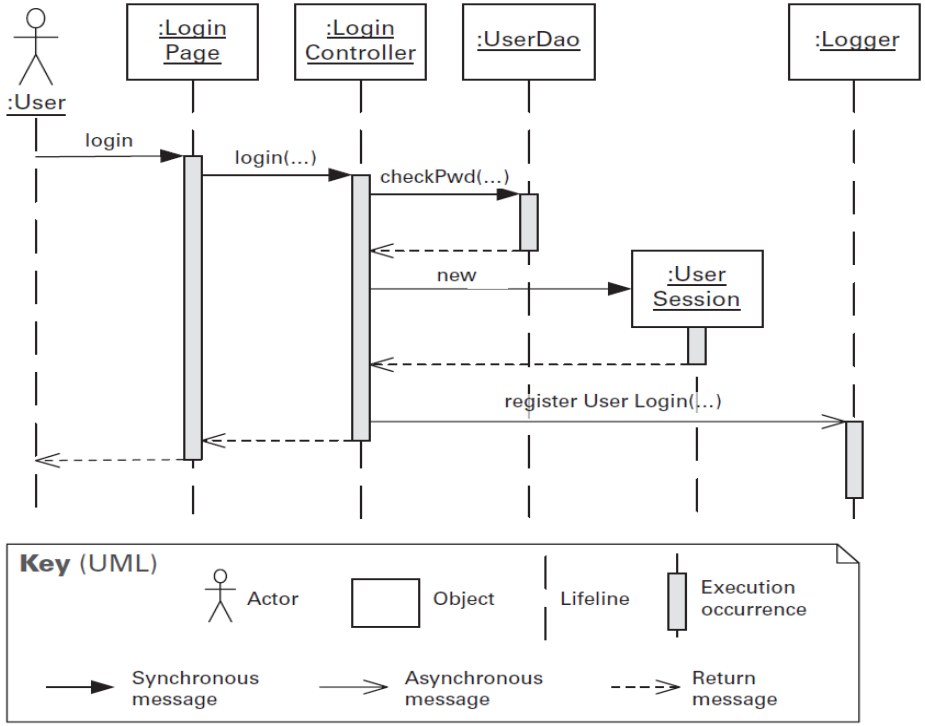
# Documenting Behavior

CMU SEI Views & Beyond

- What to document
  - The ordering of interactions among the elements
  - Opportunities for concurrency
  - Time dependencies of interactions
  - Possible states of the system or parts of the system
  - Usage patterns for different system resources



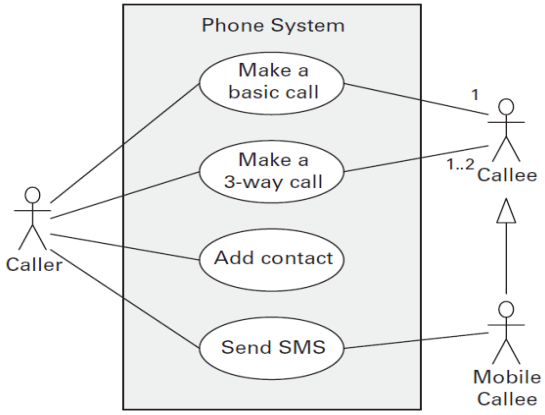



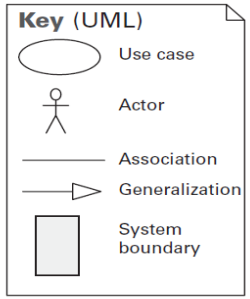



# Documenting Behavior in UML

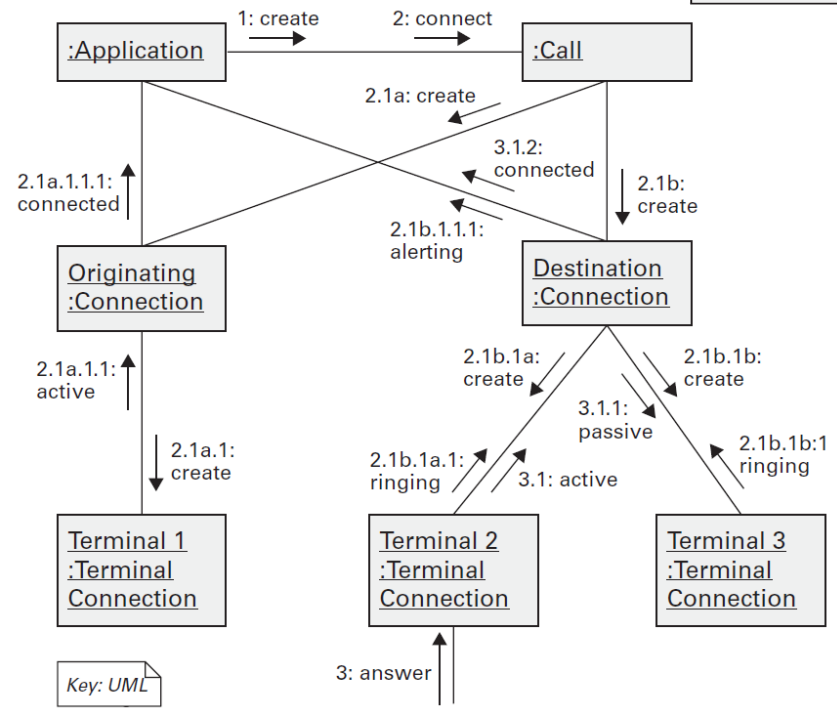
CMU SEI Views & Beyond

## Use Case diagrams

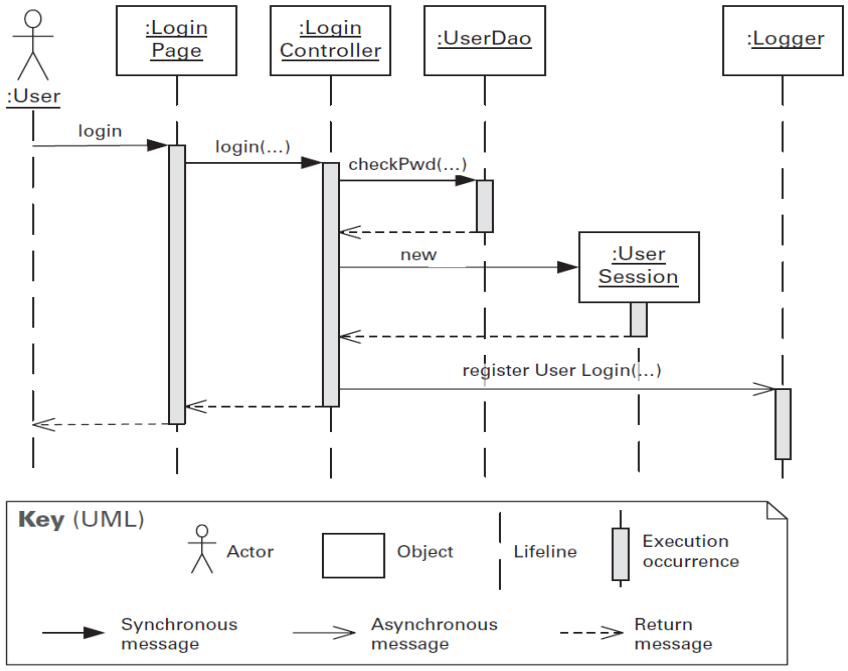



## Communication diagrams




## Sequence diagrams

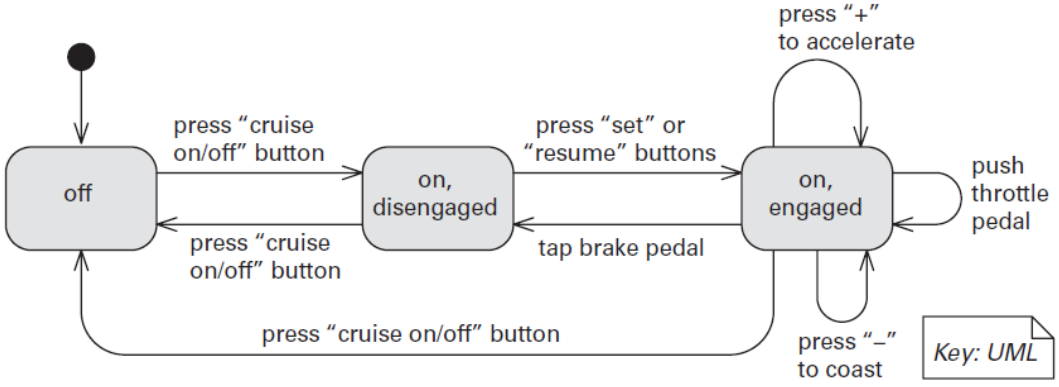


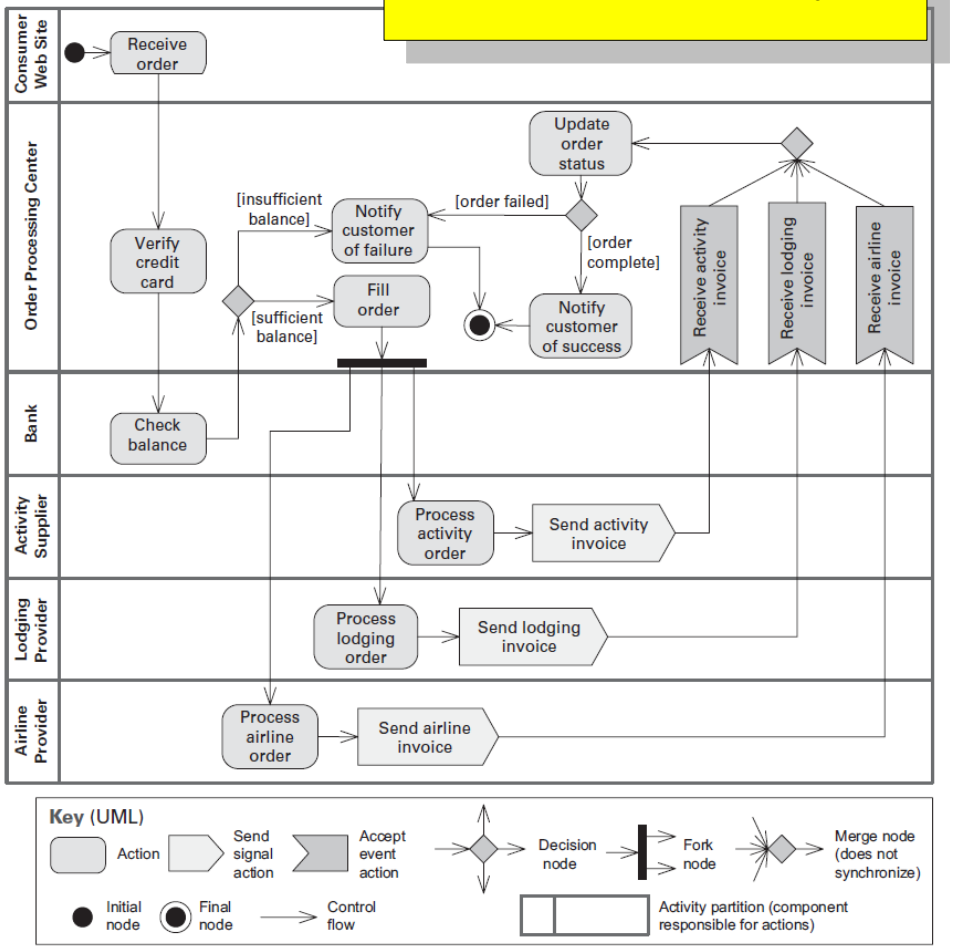
# Documenting Behavior in UML

CMU SEI Views & Beyond

- Activity diagrams

- State Machine diagrams
  - comprehensive model to show the complete behavior of structural element – it is possible to infer all possible paths from initial state to final state



# Documenting Behavior Summary

CMU SEI Views & Beyond

- Documenting behavior adds semantic detail to elements and their interactions that have time-related characteristics
  - behavioral models complement structural models by adding information that reveals ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions


- Use various types of behavior documentation together
  a) begin by documenting an overview of the functional requirements as use case diagrams
  b) then produce use case descriptions to document the events and actions that correspond to performing each use case
  c) next, for each use case produce either a sequence diagram or a communication diagram to define the messages between envisioned architecture elements
  d) finally, produce state-charts to complement the behavior documentation of the elements that have elaborate states and state transitions

- Behavior can be documented in the
  - element catalog of a view
  - interface documentation, as the element's externally visible behavior (used to explain the effects of a particular usage pattern)
  - design background section, which includes results of analyses (as behavior descriptions are often a basis for analysis)

# Documenting Mapping to Requirements

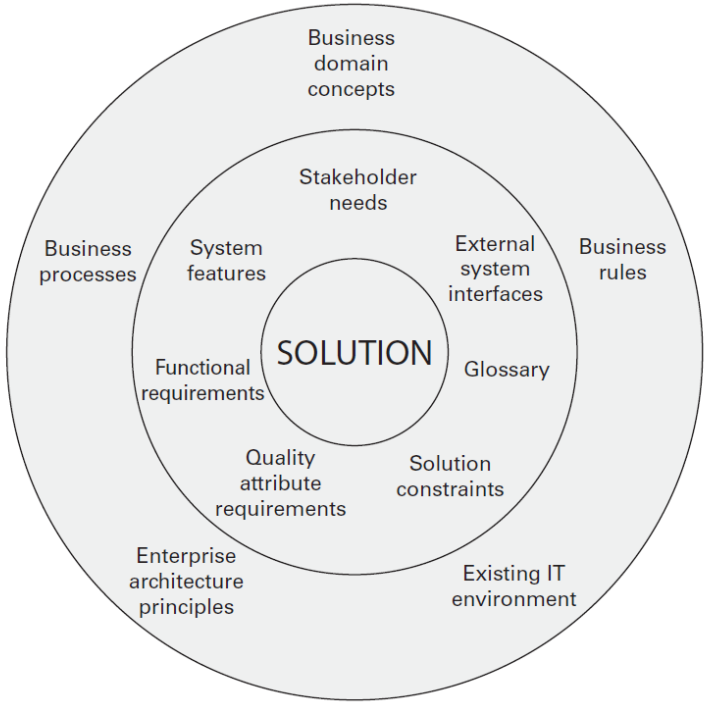
CMU SEI Views & Beyond

- Showing how the architecture satisfies requirements is an important part of the documentation
- This helps to validate the architecture by showing that
  - No requirement was forgotten
  - No requirement was contradicted
  - Every architectural decision is either predicated on at least one requirement
- To facilitate validation, document a mapping between architectural decisions and requirements
  - Put the mapping in a single place in the documentation – a new section in the documentation beyond views (good for informal or fluid requirements or if fine-grained accounting of each requirement is not needed)
  - Distribute the mapping throughout the architecture documentation – add a separate section to each view (good for fine-grained requirements that map to fine-grained architectural decisions)
  - Capture the mapping to requirements in a view of its own – according to “Requirements Viewpoint”

# Requirements Viewpoint

P. Eeles, P. Cripps (2009)

- The requirements view (based on this viewpoint) describes requirements that have shaped the architecture (may include functional requirements, quality attribute requirements, and constraints)
- If requirements are viewed as
  - "structure" in the software's environment, a mapping to requirements could be considered **a kind of allocation style**, and documented as a kind of allocation view
  - a set of concerns that crosscut the architecture elements, a mapping to requirements could be considered **a kind of aspect view** (good for projects with fine-grained requirements that map to multiple architectural decisions or elements)
- The value of a requirements view, is not confined to the identification of the subset of requirements that are deemed to be architecturally significant
- The architecture description as a whole should explicitly define how the architecture addresses each of these requirements



# Choosing the Views

CMU SEI Views & Beyond

- Usability
  - a decomposition view to analyze information presented to the user, and assign responsibility for usability-related operations
  - a component-and-connector view to enable analysis of cancellation possibilities, failure recovery, etc.
- Performance
  - component-and-connector view to support execution tracking (performance modeling)
  - additionally deployment view, behavior documentation
- Modifiability
  - a uses view and a decomposition view to show dependencies (will help with impact analysis)
  - a component-and-connector view is needed to reason about the run-time effects of a proposed change
- Security (generally same information as needed for the performance analysis)
  - a deployment view and context diagrams to see outside connections
  - a component-and-connector view to show data flow and security controls
  - a decomposition view to find where authentication and integrity concerns are handled
- Availability
  - a component-and-connector view to analyze for deadlock, synchronization and data consistency problems, and show how redundancy, fail-over, and other availability mechanisms work
  - a deployment view to show possible points of failure and backups
- Accuracy
  - a component-and-connector view showing flow and transformation of data (helps identify places where computations can degrade accuracy)

# Building the Documentation Package

CMU SEI Views & Beyond

- Document the relevant views, then add documentation that applies to more than one view
- What views you choose depends on
  - who the important stakeholders are
  - what structures are present in the architecture
  - budget, schedule and what skills are available
- Choose at least one view of each of the three different viewpoints
  - combine some views to reduce the number of views to create, keep consistent, and maintain
- Variations
  - document how to use the architecture – the “use cases” for the architecture
  - document the major design approaches taken – a major “motif” or “pattern”
  - make a single element catalog for the whole architecture – because elements appear in more than one view
  - add a section to record open questions
- Document a Mapping to Requirements
  - to validate the architecture by showing that no requirement was forgotten, no requirement was contradicted, and every architectural decision is predicated on at least one requirement

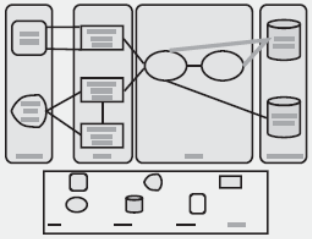
# Documentation Package
# Documenting a View

CMU SEI Views & Beyond

- The primary presentation
  - the summary of most important information about the system
  - includes the primary elements and relations (or part of those)
  - often graphical (a diagram), but might be textual (table or list)
- The element catalog
  - elements in the view and their properties
  - relations (not all the relations are shown) and their properties
  - element interfaces
  - element behavior (if elements have complex interactions)
- A context diagram shows how the system or portion of the system depicted in this view relates to its environment
- A variability guide shows how to exercise any variation points that are a part of the architecture shown in this view
- Rationale explains the reason for the design reflected in the view (provides a convincing argument that it is sound)

**Template for a View**

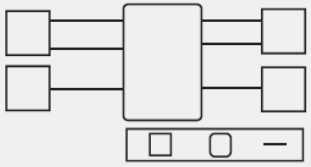
**Section 1. Primary Presentation**



**Section 2. Element Catalog**

Section 2.A. Elements and Their Properties
Section 2.B. Relations and Their Properties
Section 2.C. Element Interfaces
Section 2.D. Element Behavior

**Section 3. Context Diagram**



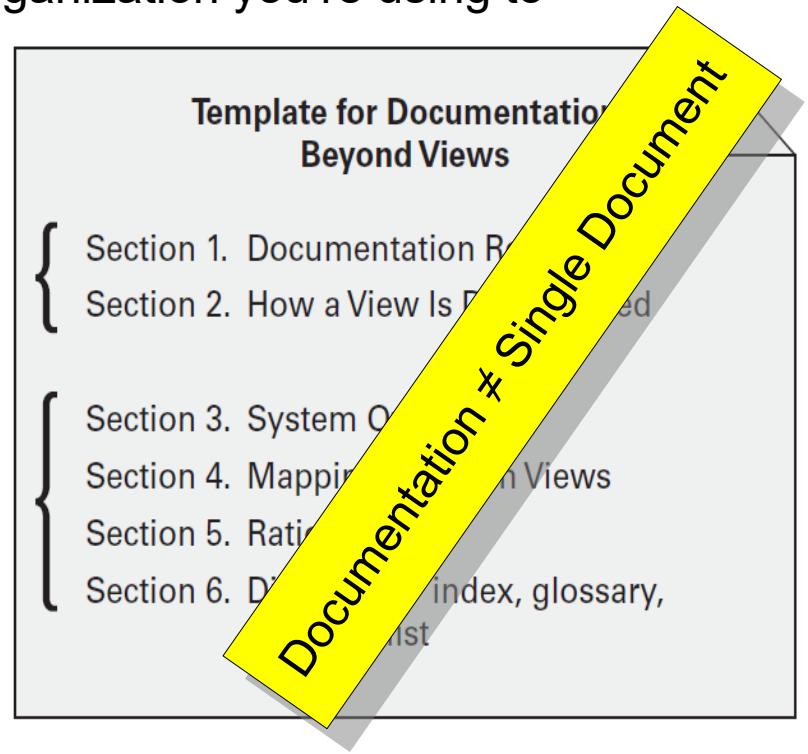
**Section 4. Variability Guide**

**Section 5. Rationale**

# Documentation Package Outside of Views

CMU SEI Views & Beyond

- Documentation Roadmap – what information is in the documentation and where to find it
- How a View Is Documented – explain the standard organization you're using to document views
- System Overview
  - a short description of the system's function, its users, and any important background or constraints
  - provides readers with a consistent mental model of the system and its purpose
- Mapping Between Views
  - to understand the associations between views
- Rationale
  - documents the architectural decisions that apply to more than one view



# Architecture Overview Presentation
# Outline for One-Hour Overview (20-35 slides)

CMU SEI Views & Beyond

- **Problem statement** (2–3 slides) – the problem the system is trying to solve
  - Driving architecture requirements, measurable quantities associated with these, and approaches for meeting these
  - Technical constraints (operating system, hardware, or platform software)
- **Architecture strategy** (2 slides) – the major architecture challenges
  - The architecture approaches, styles, patterns, or mechanisms used (what quality attributes they address and how)
- **System context** (1–2 slides)
  - One or two whole-system context diagrams that clearly show the system boundaries and other systems with which it must interact
- **Architecture views** (12–18 slides)
  - Chosen views (at least one module, one component-and-connector, and one allocation view) – for each, include the top-level (system wide) primary presentation and, if needed, few refined primary presentations (include a notation key)
- **How the architecture works** (3–10 slides)
  - Up to three of the most important use cases (if possible, include the run-time resources consumed for each)
  - Show the architecture's capacity for growth with up to three of the most important change scenarios (describe the change impact)

# Content

38. The architect concerns himself with the depth and not the surface, with the fruit and not the flower.

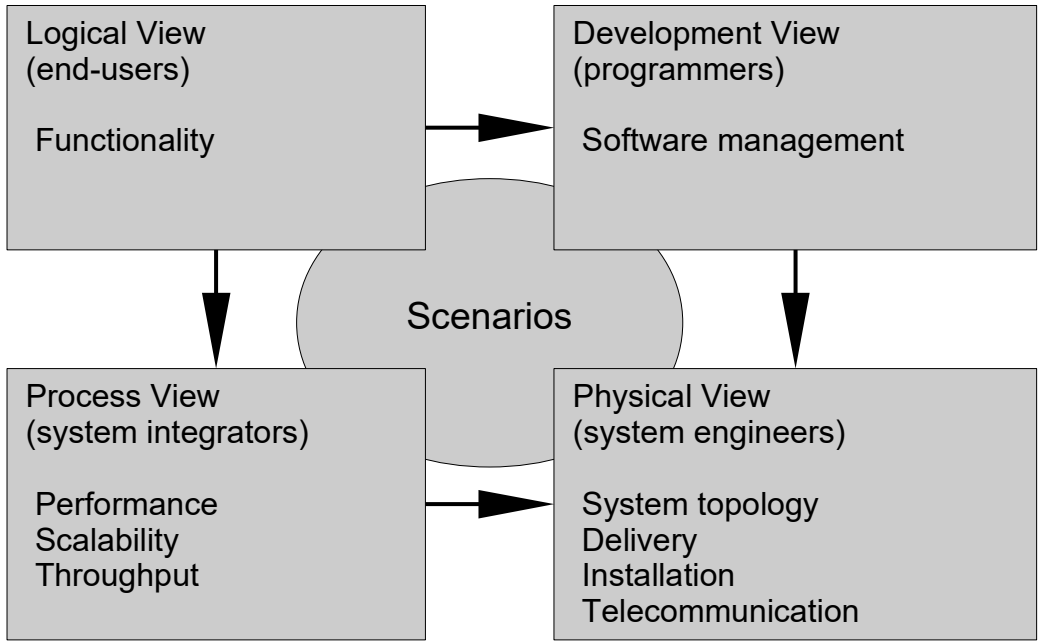
Lao Tsu (by Philippe Kruchten)

- Why to Document Architecture
- CMU SEI – “Views & Beyond” Method
  - Module Views
  - Component-and-Connector Views
  - Allocation Views
  - Advanced techniques
- Some other Architecture Documentation Methods
- Other Architecture Documentation Practices
  - Architecture Description Languages
  - Documenting Architecture in Code
- Conclusions

# Rational Unified Process (RUP) “Five-View Approach” (based on P. Kruchten “4+1 Views”)

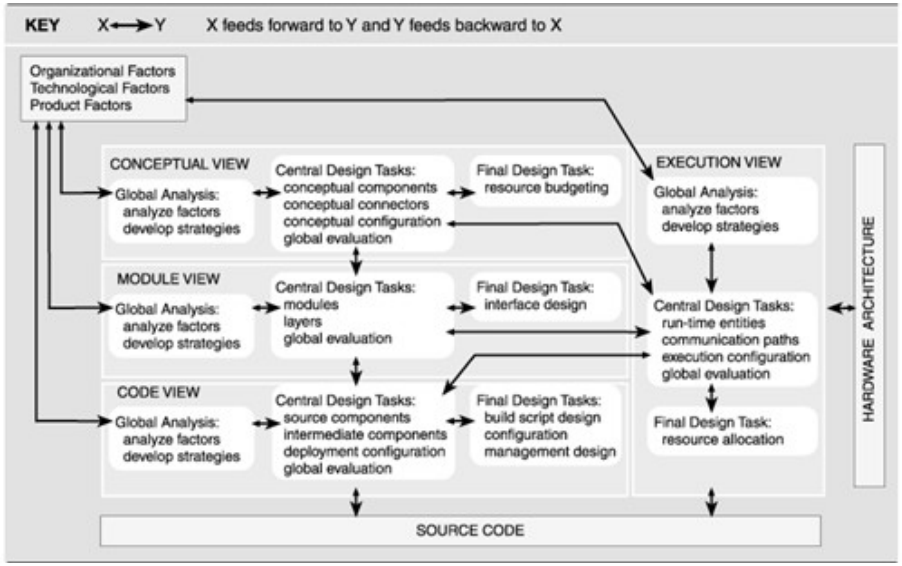
1. **Logical** view captures the functional requirements (what the system should provide in terms of services to its users)
2. **Process** view documents the tasks (processes and threads) involved, takes into account some non-functional requirements, and addresses issues of concurrency and distribution
3. **Development** view focuses on the actual software module organization in the software development environment
4. **Physical** (deployment) view documents the various physical nodes for the most typical platform configurations and takes into account primarily the non-functional requirements of the system

5. **Scenarios** (use case) view documents architecturally significant behavior
   - the system’s intended functions and its environment
   - a contract between the customer and the developers and as a design check on the other views
   - to discover the architectural elements during the architecture design and to validate architecture



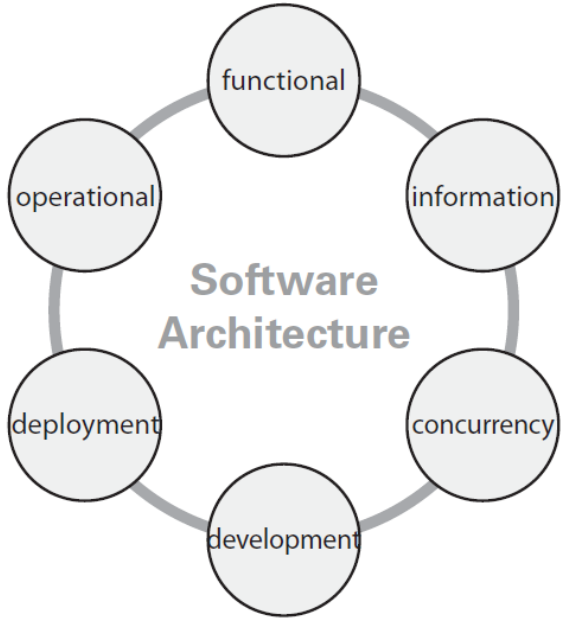
# Siemens “Four Views”

- Conceptual View
  - explains how the system's functionality is mapped to components and connectors
- Module View
  - explains how the components, connectors, ports, and roles are mapped to abstract modules and their interfaces
- Execution View
  - explains how the system's functionality is mapped to run-time platform elements, such as processes and shared libraries
  - platform elements consume platform resources that are assigned to a hardware resource
- Code View
  - explains how the software implementing the system is organized into source and deployment components



# Rozanski and Woods Viewpoint Set

- **Functional view** documents the system's functional elements, their responsibilities, interfaces, and primary interactions – cornerstone of most architecture documents (drives system structures)
- **Information view** documents the way that the architecture stores, manipulates, manages, and distributes information (static data structure and information flow)
- **Concurrency view** describes the concurrency structure of the system and maps functional elements to the parts of the system that can execute concurrently (process and thread structures and the inter-process communication mechanisms)
- **Development view** describes the architecture that supports the software development process




- **Deployment view** describes the environment into which the system will be deployed, including capturing the dependencies the system has on its run-time environment
- **Operational view** describes how the system will be operated, administered, and supported when it is running in its production environment

# Comparison of Viewpoints in different Architecture Description Methods

| CMU SEI | RUP / Kruchten 4+1 | Siemens Four Views | Rozanski & Woods | UML Diagrams |
|---|---|---|---|---|
| Module | Logical<br>Implementation | Module | Development<br>Information | Package<br>Class |
| Component-and-Connector | Process | Conceptual | Functional<br>Concurrence<br>Information (flow) | Component<br>Object |
| Allocation | Deployment | Execution<br>Code | Deployment<br>Operational | Use Case<br>Deployment |
| | | | | |
| Requirements | | | | |
| Behavior | Scenarios | | | Use Case<br>Sequence<br>Communication<br>Activity<br>State Machine |

# Content

38. The architect concerns himself with the depth and not the surface, with the fruit and not the flower.

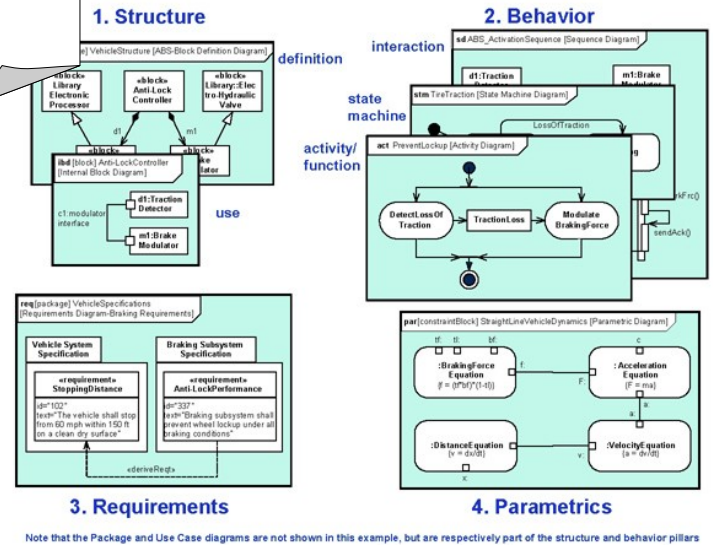
Lao Tsu (by Philippe Kruchten)

- Why to Document Architecture
- CMU SEI – “Views & Beyond” Method
  - Module Views
  - Component-and-Connector Views
  - Allocation Views
  - Advanced techniques
- Some other Architecture Documentation Methods
- Other Architecture Documentation Practices
  - Architecture Description Languages
  - Documenting Architecture in Code
- Conclusions

# Architecture Description Languages

- An approach to **formalize** architecture descriptions and provide standardized representation to support tools, analyzes, simulation and interchange
- Textual and/or graphical syntax and formally defined semantics
- Some ADLs
  - Academic
    - ACME (CMU)
    - C2 (UCI)
    - Wright (CMU)
  - Standards
    - AADL (SAE – Society of Automotive Engineers)
    - SysML (OMG) extension of UML by requirements & parametrics diagrams
    - ArchiMate (OpenGroup) – for enterprise level

```
System simple_cs = {
    Component client = { Port send-request; };
    Component server = { Port receive-request; };
    Connector rpc = { Roles { caller, callee}};
    Attachments {
        client.send-request to rpc.caller;
        server.receive-request to rpc.callee;
    }
}
```




Pictures © OMG

# Documenting Architecture in Code

The source code is the design

J. W. Reeves

- Use naming conventions according to the architectural elements – using vocabulary of relevant architecture style (e.g. components, connectors, layers, ...)
- Organize/package source code into name-spaces and modules
- Use meta-info (annotations and attributes) to map software to the external structures
- Create internal Domain Specific Languages (fluent coding style – method chaining) for expressing architecture structures directly in code (kind of ADL)
- Represent the architectural abstractions (both control and data) directly in code using abstraction mechanisms of programming language
- Represent the domain model including the system environment in the source code
- Document the architectural decisions in source code comments

# Content

38. The architect concerns himself with the depth and not the surface, with the fruit and not the flower.

Lao Tsu (by Philippe Kruchten)

- Why to Document Architecture
- CMU SEI – “Views & Beyond” Method
  - Module Views
  - Component-and-Connector Views
  - Allocation Views
  - Advanced techniques
- Some other Architecture Documentation Methods
- Other Architecture Documentation Practices
  - Architecture Description Languages
  - Documenting Architecture in Code
- Conclusions

# Conclusions

Designing an architecture without documenting it, is like winking at a girl in the dark – you know what you´re doing, but nobody else does

E. Woods

- Creating an architecture is not enough – it **has to be communicated** properly to let others use it properly to do their jobs
- Architecture documentation is for
  - **communication** – primary communication vehicle between stakeholders
  - **education** – introducing new people to the system
  - **designing** – provides structure for design decisions
  - **analyzing** – provides information to analyze the system properties (quality attributes)
  - **constructing** – tells what to implement (must contain models to support automated construction)
- Write documentation
  - from the reader's point of view, for clear purpose and record rationale
  - avoiding unnecessary repetition and ambiguity
  - using a standard organization

# Conclusions

Make your system capture its own current architecture automatically

- Document the **relevant views** (at least one per each major viewpoint), then add documentation that applies to more than one view (combine some views to reduce the number of views to create, keep consistent, and maintain)

- Choose the views depending on
  - who the **important stakeholders** are and what are their **concerns** towards the system
  - what **structures** are present in the architecture
  - budget, schedule and what skills are available

- Additionally to the views
  - document the **major design decisions** taken, how to use the architecture and the ways architecture is allowed to change
  - make a **single element catalog** for the whole architecture – because elements appear in more than one view
  - document a **mapping to requirements**, to show that no requirement was forgotten, nor contradicted
  - add a section to record open questions

38. The architect concerns himself with the depth and not the surface, with the fruit and not the flower.

Lao Tsu (by Philippe Kruchten)

# Thank You!

# Questions

- What's the purpose of software architecture documentation?
- Describe 3 major viewpoints (what they represent and their purpose) in CMU SEI "Views & Beyond" method
- What contains a documentation of specific view in CMU SEI method?
- List the specific view styles of 3 major viewpoints in CMU SEI method and RUP / "4+1" method
- What UML models and diagrams you would use to describe the views in 3 major viewpoints in CMU SEI method?
- In which viewpoint in CMU SEI method are data models used?
- How would you document the cross-cutting concerns?
- What is refinement?
- What is the purpose of Context Diagrams?
- How you would document variability?
- How you would document architectural decisions?
- Name different types of architectural decisions?
- How views can be combined?
- How you would document software interfaces?
- What UML models and diagrams can be used to document behavior?
- How (based on what) you choose the views into the architecture description?
- What is contained in the software architecture documentation package?
- How you would represent architecture in the source code?

# Literature

- https://flylib.com/books/en/2.121.1/
- CMU SEI Library “Views and Beyond”:
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5019
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=9685
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=6497
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5939
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5847
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5471
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=7095
  - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=6997
  - https://wiki.sei.cmu.edu/sad/index.php/The_Adventure_Builder_SAD – Example AD
- https://www.researchgate.net/publication/238381956_A_41_View_Model_of_Software_Architecture
- http://www.iso-architecture.org/ieee-1471/templates/
- http://www.rm-odp.net/
- http://www.lcc.uma.es/~av/download/UML4ODP_IS_V2.pdf
- http://www.cs.cmu.edu/%7Eacme/docs/language_overview.html
- http://www.aadl.info/aadl/currentsite/
- http://sysml.org/
- … Google “documenting software architecture” ...

# Choosing the Views

CMU SEI Views & Beyond

| | Module Views | | | | | C&C Views | Allocation Views | | | | Other Documentation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Decomposition | Uses | Generalization | Layered | Data Model | Various | Deployment | Implementation | Install | Work Assignment | Interface Documentation | Context Diagrams | Mapping Between Views | Variability Guides | Analysis Results | Rationale and Constraints |
| Project managers | s | s | | s | | | d | | | d | | o | | | | s |
| Members of development team | d | d | d | d | d | d | s | s | d | | d | d | d | d | | s |
| Testers and integrators | d | d | d | d | d | s | s | s | s | | d | d | s | d | | s |
| Designers of other systems | | | | | s | | | | | | d | o | | | | |
| Maintainers | d | d | d | d | d | d | s | s | | | d | d | d | d | | d |
| Product-line application builders | d | d | s | o | s | s | s | s | s | | s | d | s | d | | s |
| Customers | | | | | | | o | | | o | | o | | | s | |
| End users | | | | | | s | s | | o | | | | | | s | |
| Analysts | d | d | s | d | d | s | d | | s | | d | d | | s | d | s |
| Infrastructure support personnel | s | s | | | s | s | d | d | o | | | | | s | | |
| New stakeholders | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| Current and future architects | d | d | d | d | d | d | d | s | d | s | d | d | d | d | d | d |

Key: d = detailed information, s = some details, o = overview information, x = anything

# “Marketecture”

Essential Software Architecture

I. Gorton (2006)

- A one page, typically informal depiction of the system’s structure and interactions
- It shows the major components, their relationships and has a few well chosen labels and text boxes that portray the design philosophies embodied in the architecture
  - A marketecture is an excellent vehicle for facilitating discussion by stakeholders during design, build, review, and of course the sales process – it’s easy to understand and explain, and serves as a starting point for deeper analysis

# Quality Attributes in the Documentation

1. Any major design approach (such as an architecture pattern or style) chosen by the architect will have quality attribute properties associated with it
   - client-server → scalability, layering → portability, an information-hiding-based decomposition → modifiability, services → interoperability, …
   - explaining the choice of approach (rationale) includes a discussion about the satisfaction of quality attribute requirements and trade-offs incurred

2. Individual architectural elements that provide a service often have quality attribute bounds assigned to them
   - these quality attribute bounds are defined in the interface documentation for the elements, sometimes in the form of a Quality of Service contract (or simply be recorded as properties that the elements exhibit)

3. Quality attributes often impart a “language” of things that you would look for
   - security involves things like security levels, authenticated users, audit trails, firewalls, and the like
   - performance brings to mind buffer capacities, deadlines, periods, event rates and distributions, clocks and timers, and so on
   - availability conjures up mean time between failure, failover mechanisms, primary and secondary functionality, critical and noncritical processes, and redundant elements

4. Architecture documentation often contains a mapping to requirements that shows how requirements (including quality attribute requirements) are satisfied

5. Every quality attribute requirement will have a constituency of stakeholders who want to know that that quality attribute requirement is going to be satisfied
   - for these stakeholders, the architect should provide a special place in the documentation’s introduction that either provides what the stakeholder is looking for or tells the stakeholder where in the document to find it