Creating and Analyzing Software Architecture

Dr. Igor Ivkovic

iivkovic@uwaterloo.ca

[with material from “Software Architecture: Foundations, Theory, and Practice”, by Taylor, Medvidovic, and Dashofy, published by Wiley; and from “Object-Oriented Software Engineering”, by Bruegge and Dutoit, published by Prentice Hall]

Objectives

- Model the architecture using architectural views
- Express architectural viewpoints using metamodels
- Use quality attributes to drive architectural design

Architectural Views and Viewpoints /1

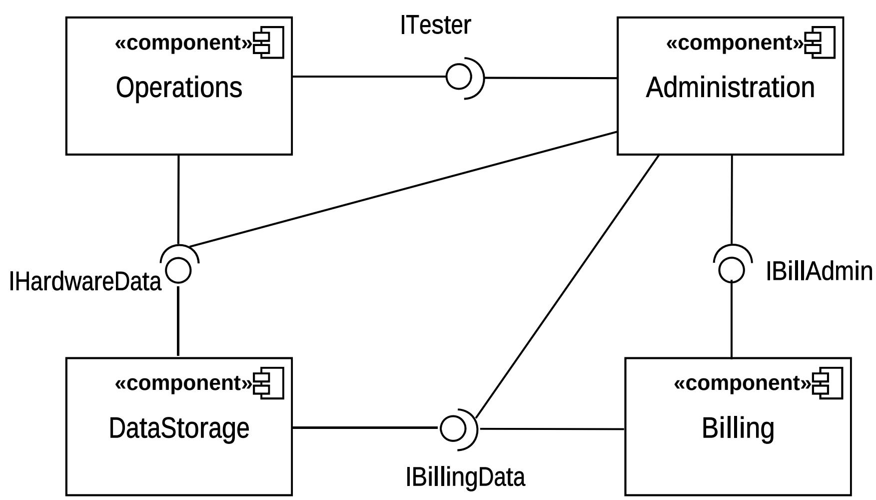
- **It is typically not feasible to capture everything we want to model in a single architectural model**
  - The model would be too big, complex, and confusing
- Instead, we create several coordinated models, each capturing a subset of the design decisions
  - Generally, the subset is organized around a particular concern or other selection criteria

Architectural Views and Viewpoints /2

- **The subset-model is called an “architectural view” and the concern an “architectural viewpoint”**
  - For instance, deployment viewpoint is concerned with how software systems are deployed on hardware and networking nodes
  - **Instances of the deployment viewpoint are called views**
  - See Kruchten’s “4+1 View Model of Software Architecture” paper for additional viewpoints

Commonly-Used Viewpoints /1

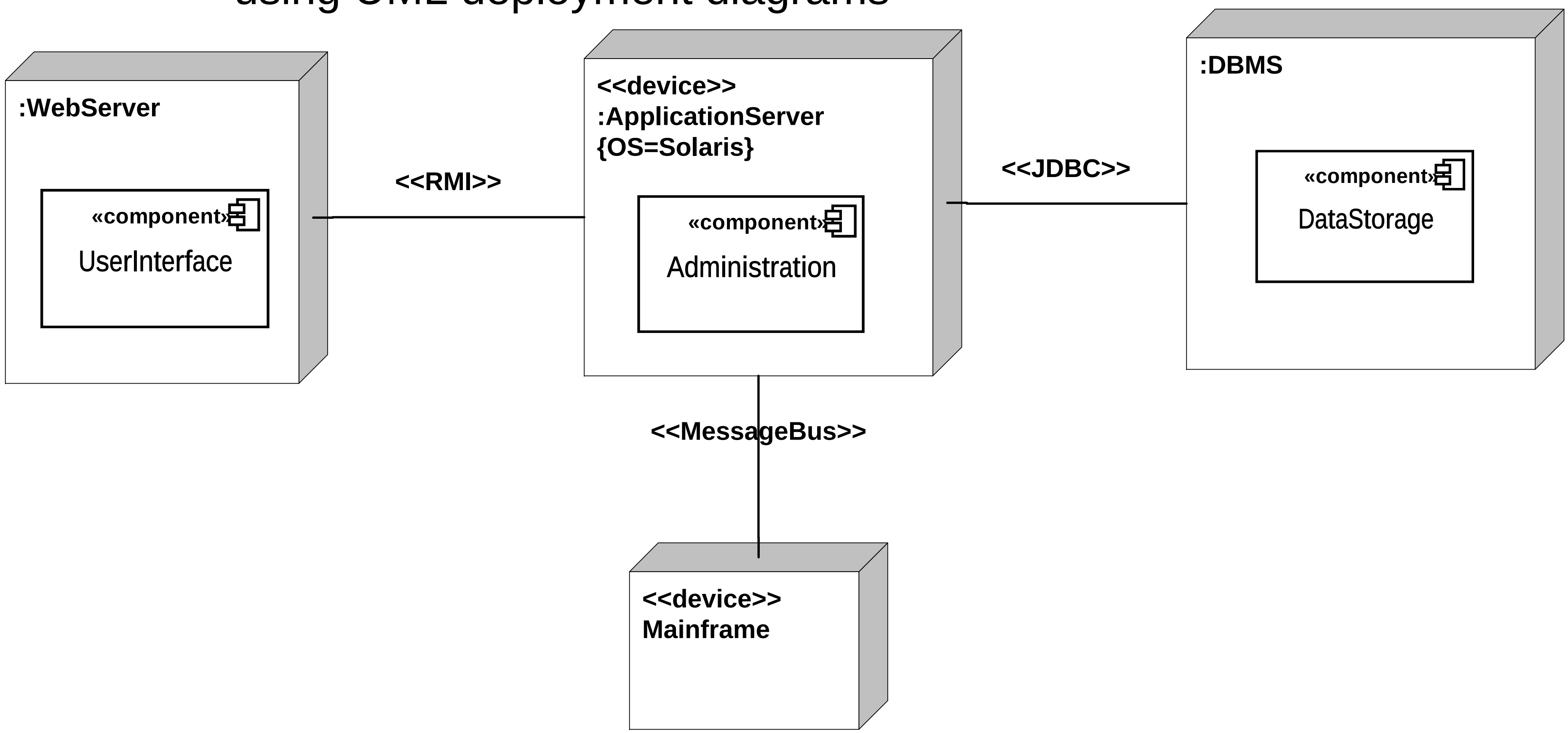
- **Logical Viewpoints:**
  - Capture the logical entities in a system and how they are interconnected
  - Use UML component diagram to show interfaces and decomposition between components



- **Physical Viewpoints:**
  - Capture the physical (often hardware) entities in a system and how they are interconnected

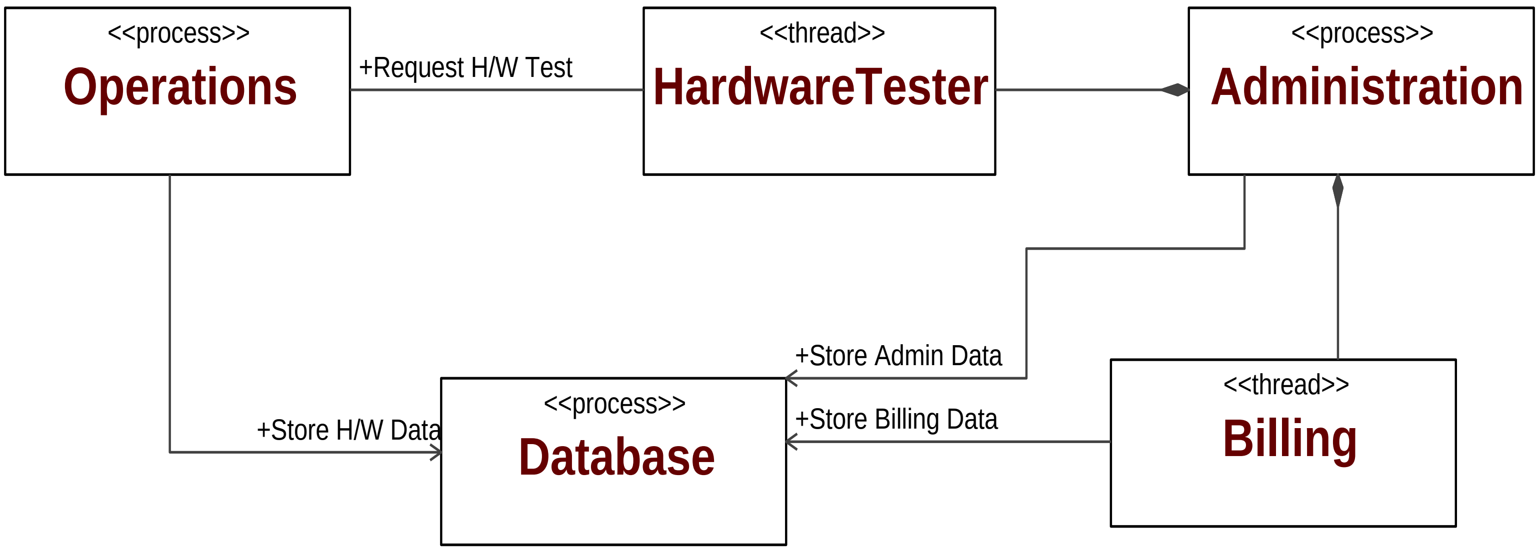
Commonly-Used Viewpoints /2

- **Deployment Viewpoints:**
  - Capture how logical entities are mapped onto physical entities
  - Can combine the deployment and physical viewpoints into one using UML deployment diagrams



Commonly-Used Viewpoints /3

- **Concurrency/Process Viewpoints:**
  - Capture how concurrency and threading will be managed in a system

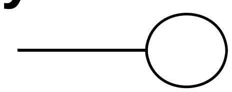
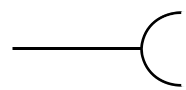


- **Behavioral Viewpoints:**
  - Capture the expected behavior of (parts of) a system
  - System scenarios using UML collaboration diagrams

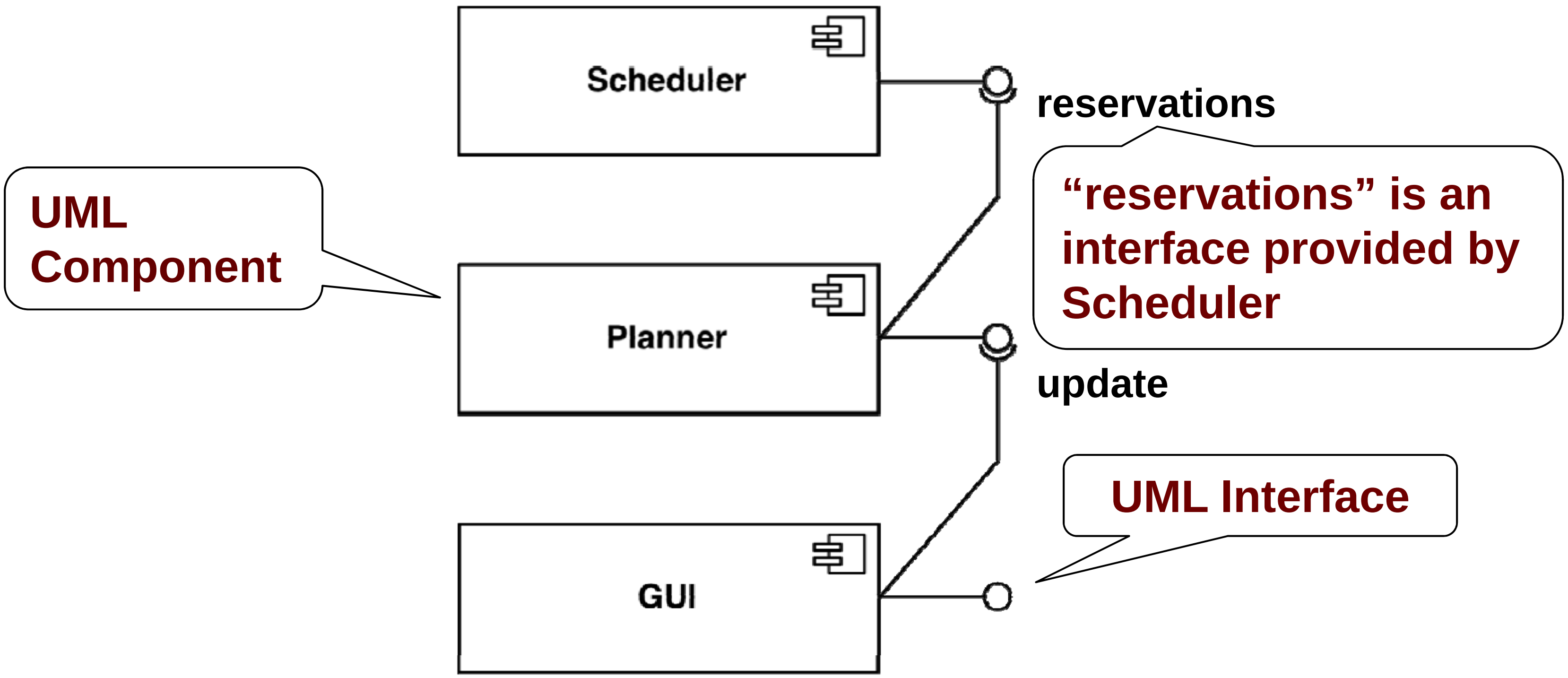
UML Component Diagram

- **Illustrate dependencies between architectural components at design time, compilation time and runtime**
  - Used to model the system in terms of components and dependencies among the components
  - Also called “software wiring diagrams“
  - They show how the software components are wired together in the overall application
- The dependencies (edges in the graph) are shown as dashed lines with arrows from the client component to the supplier component:
  - The lines are often also called connectors
  - The types of dependencies are implementation-language specific
- Components can also be
  - Source code, linkable libraries, executables

UML Interfaces

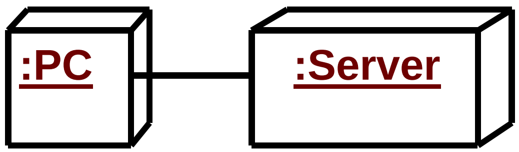
- **A UML interface describes a group of operations used or created by UML components**
- There are two types of interfaces: provided and required interfaces
  - A provided interface (**implemented by the component**) is modeled using the lollipop notation 
  - A required interface (**accessed by the component**) is modeled using the socket notation 
- A port specifies a distinct interaction point between the component and its environment
  - Ports are depicted as small squares on the sides of classifiers (in some tools, ports are mandatory)

UML Component Diagram Example



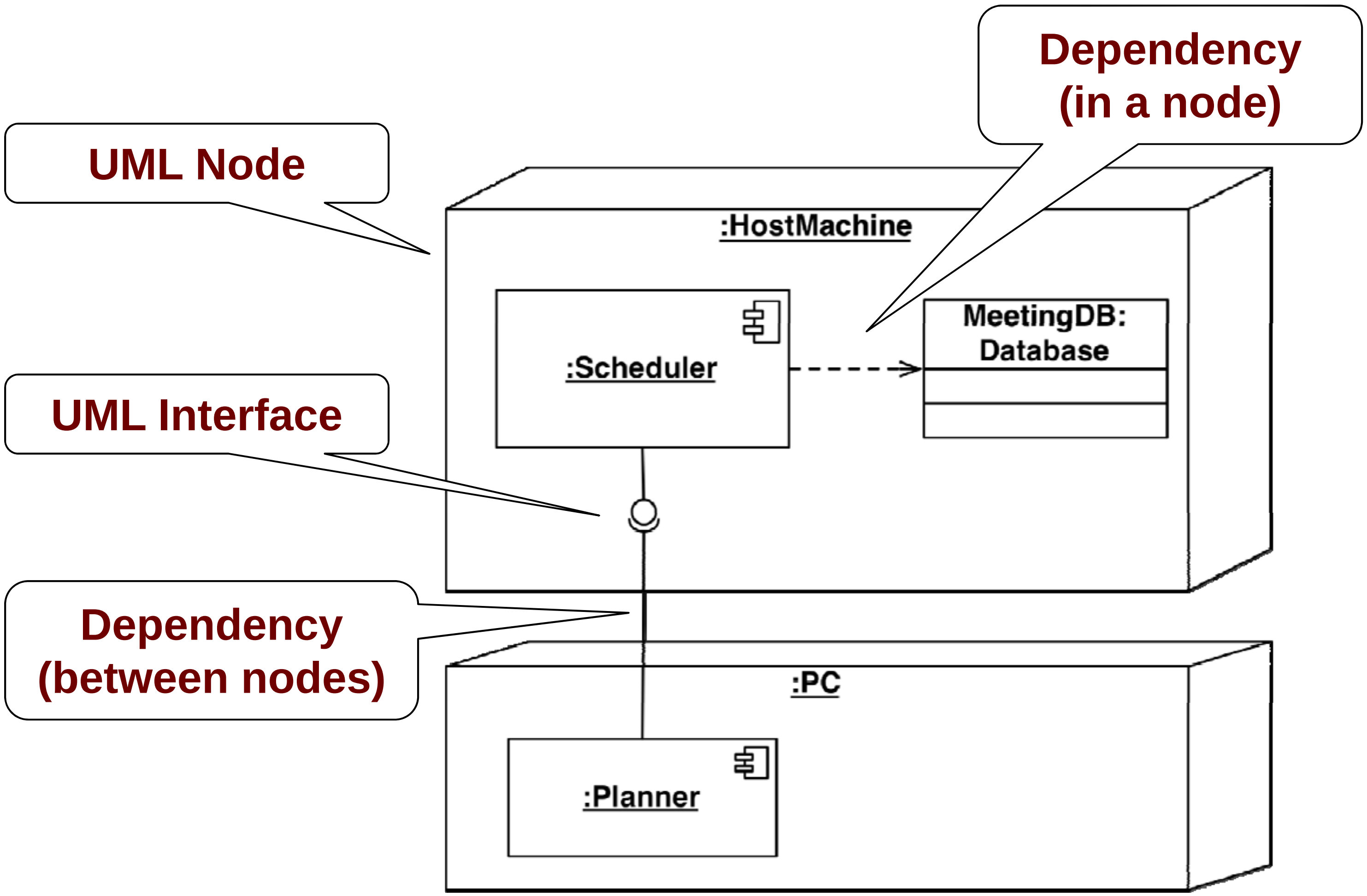
UML Deployment Diagrams

- **UML deployment diagrams are used for showing physical and deployment architectural viewpoints**
- UML deployment diagrams are also used during
  - Subsystem decomposition
  - Concurrency
  - Hardware/Software Mapping

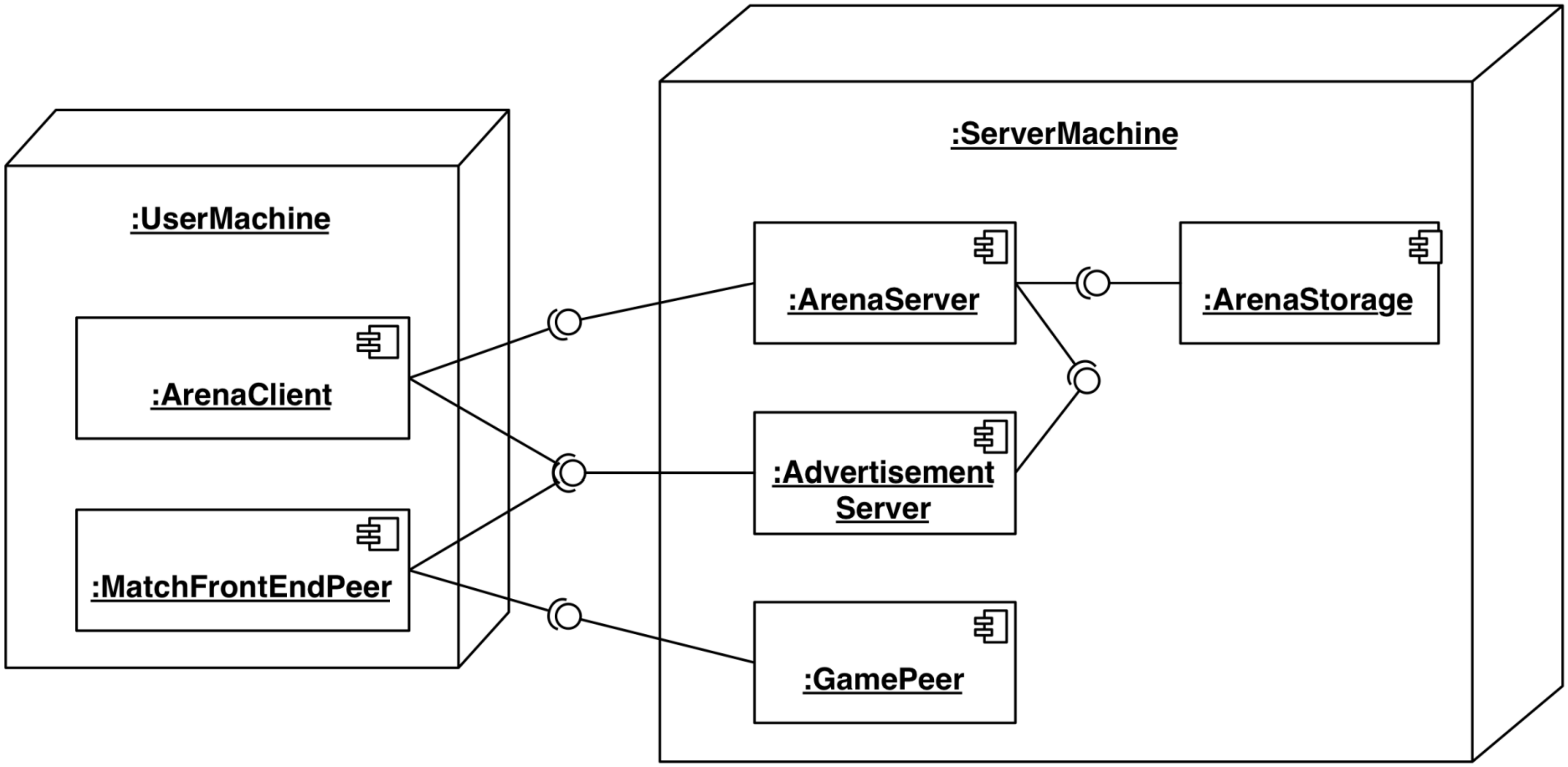


- **A deployment diagram is a graph of nodes and connections (“communication associations”)**
  - Nodes are shown as 3D boxes
  - Connections between nodes are shown as solid lines
  - Nodes may contain components
  - Components are connected through deployment connectors
  - Components may contain objects (indicating that the object is part of the component)

UML Deployment Diagram Example



ARENA Deployment Diagram



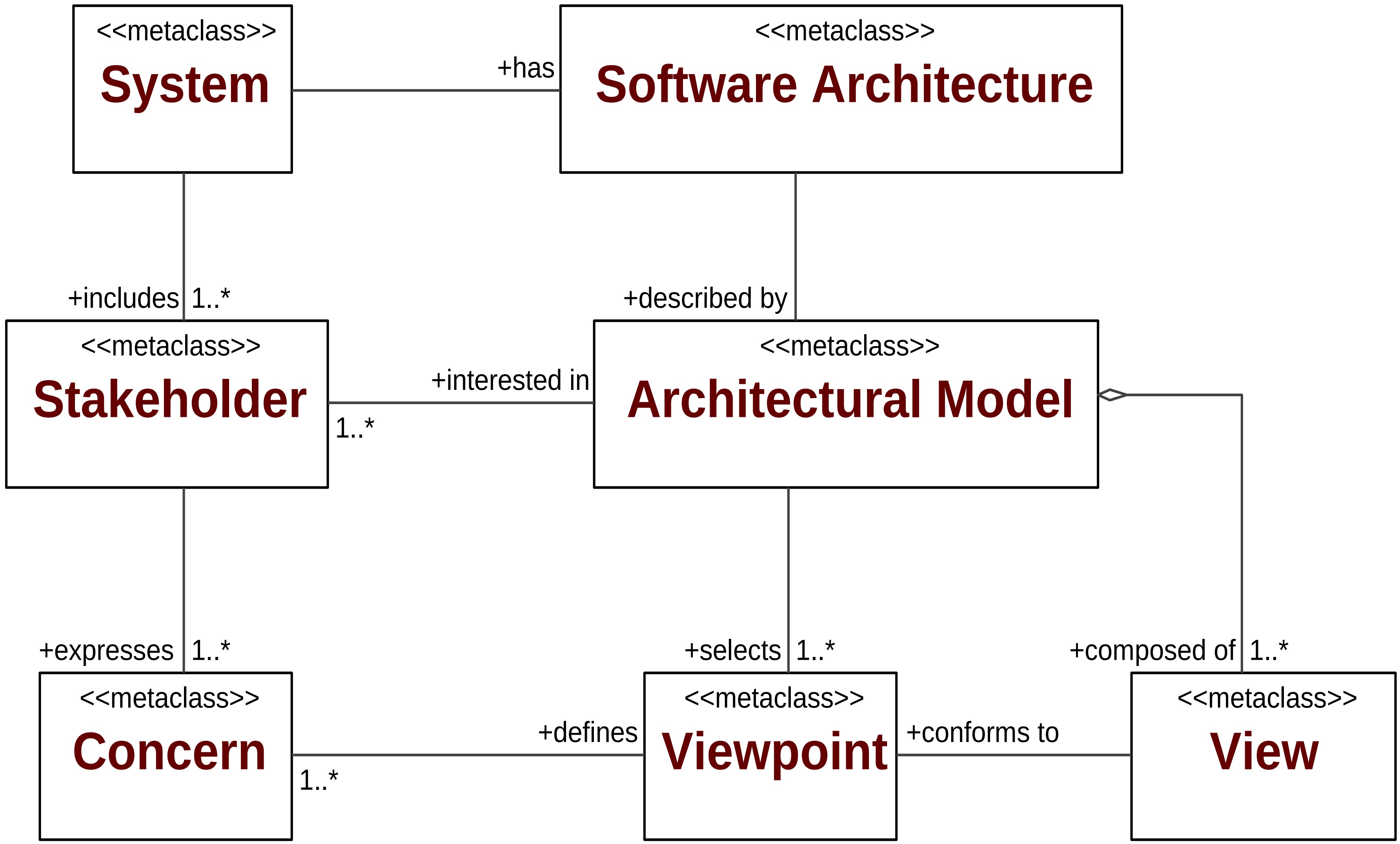
Consistency Among Views

- **Views can contain overlapping and related design decisions**
  - There is the possibility that the views can thus become inconsistent with one another
- Views are consistent if the design decisions they contain are compatible
  - **Views are inconsistent if two views assert design decisions that cannot simultaneously be true**
- Inconsistency is usually but not always indicative of problems
  - Temporary inconsistencies are a natural part of exploratory design
  - Inconsistencies cannot always be fixed

Integrating Multiple Architectural Views

- Diagrams of different types cover different, complementary facets of the system
  - Overlapping aspects require an integration mechanism needed for compatibility among diagrams
- **Metamodel**
  - Model defining and relating conceptual abstractions in terms of which other models are defined
  - Defines the structure of the modeling language, provides meta-language for defining it, summarizes main features
- Typical approach:
  - Define a common metamodel interrelating all conceptual abstractions used in each type of diagram
  - **Enforce inter-diagram consistency rules based on the integrated metamodel**

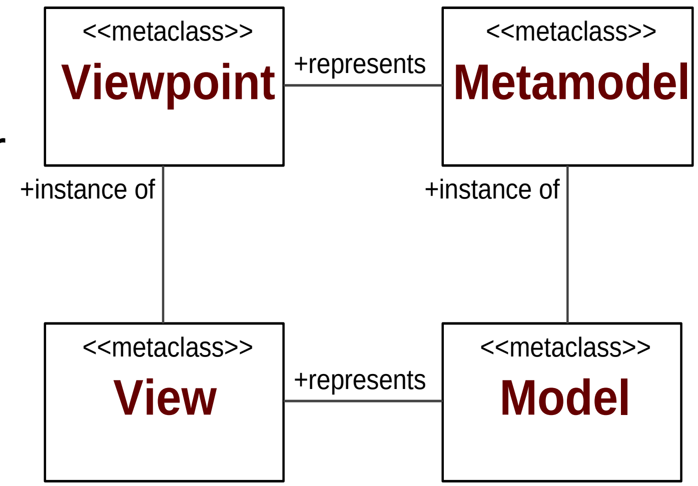
Model-Driven Software Architecture /1



Based on IEEE Std 1471-2000:
IEEE Recommended Practice for Architectural Description of Software-Intensive Systems

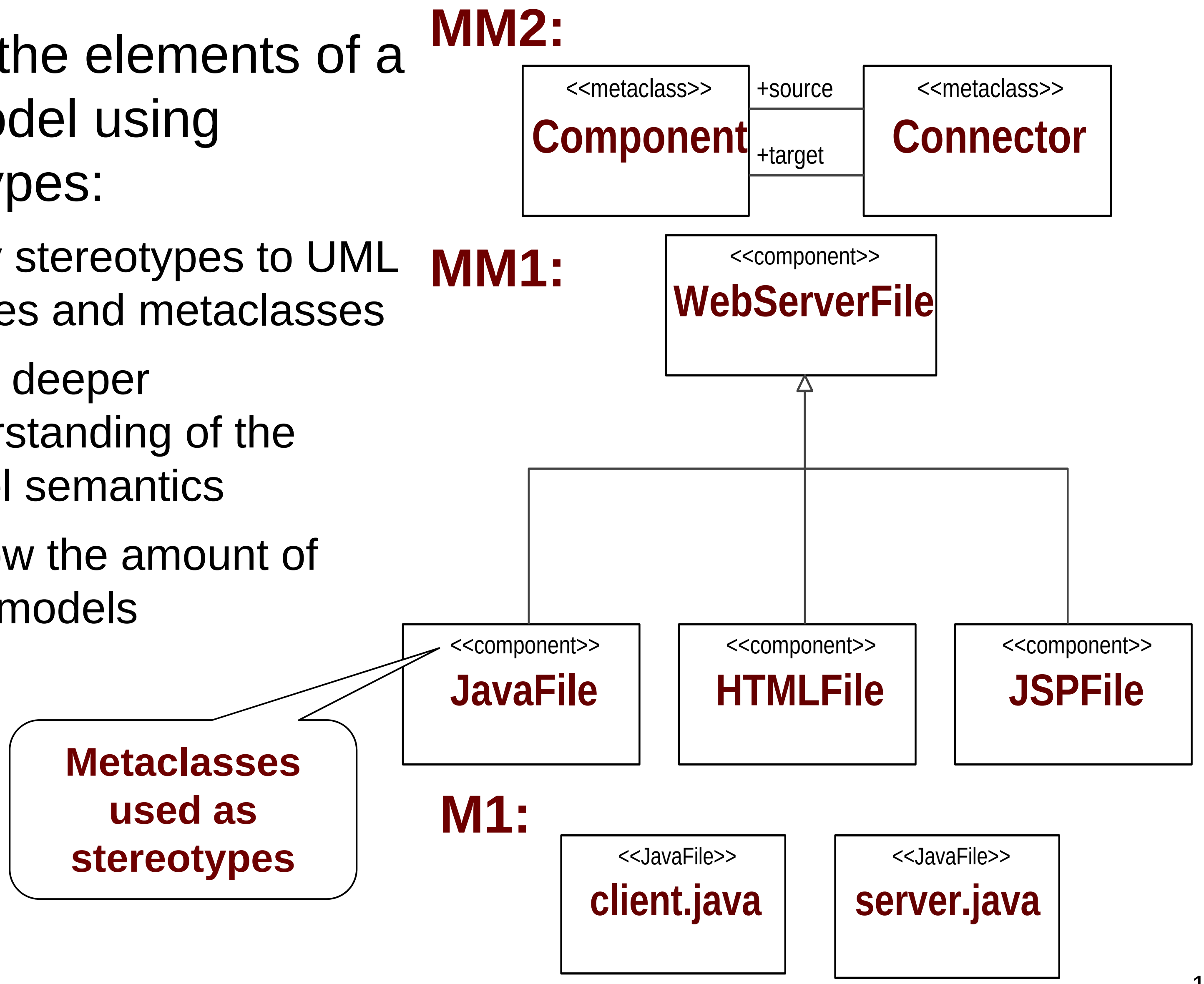
Model-Driven Software Architecture /2

- **Model-driven software architecture creation:**
  1. Identify relevant system stakeholders
  2. For each stakeholder, determine their key concerns (e.g., logical structure, performance, concurrency)
  3. Express their concerns as viewpoints
     - Model the viewpoint elements (terminology) as metaclasses, and each viewpoint itself as a metamodel
  4. Create views that conform to the matching viewpoints
     - Represent each view as a model that conforms to the corresponding viewpoint/metamodel



Model-Driven Software Architecture /3

- Project the elements of a metamodel using stereotypes:
  - Apply stereotypes to UML classes and metaclasses
  - Get a deeper understanding of the model semantics
  - Narrow the amount of valid models



Types of Requirements

- **Functional requirements**
  - Describe the interactions between the system and its environment, independent from the implementation
    - Example: “An operator must be able to issue a new ticket“
- **Nonfunctional requirements**
  - Aspects not directly related to functional behavior
    - Example: “The response time must be less than 1 second”
- **Constraints**
  - Imposed by the client or the environment
    - Example: “The system must be implemented on Windows“
  - Also referred to as “pseudo requirements”

Functional vs. Nonfunctional Requirements

- Functional Requirements
  - Describe user tasks that the system needs to support
  - Phrased as actions
    - "Advertise a new league"
    - "Schedule tournament"
    - "Notify an interest group"
- Nonfunctional Requirements
  - Describe properties of the system or the domain
  - Phrased as constraints or negative assertions
    - "All user inputs should be acknowledged within 1 second"
    - "A system crash should not result in data loss".

Examples of Nonfunctional Requirements

- Usability Requirement
  - “Passengers must be able to buy a ticket for travel without prior registration”
- Performance Requirement
  - “The system must support 10 parallel transactions”
- Supportability Requirement
  - “The operator must be able to add new travel routes without modifications to the existing system.”

Quality Attributes in Architectural Design

- **Quality attributes can be used to guide software architecture design:**
  - Architectural styles discussed so far offer specific advantages and disadvantages with respect to various quality attributes
  - For instance, pipes and filters does not support interactive software and hence (broadly speaking) has limited adaptability
  - In addition, there are general architectural heuristics that apply to specific quality-driven design goals and can be used as drivers of architectural design
- Specific design goals include (see Chapter 12 of Taylor et al textbook for details of these and other goals):
  - Reducing complexity
  - Improving scalability and heterogeneity
  - Improving dependability and fault tolerance

Goal: Reduce Complexity /1

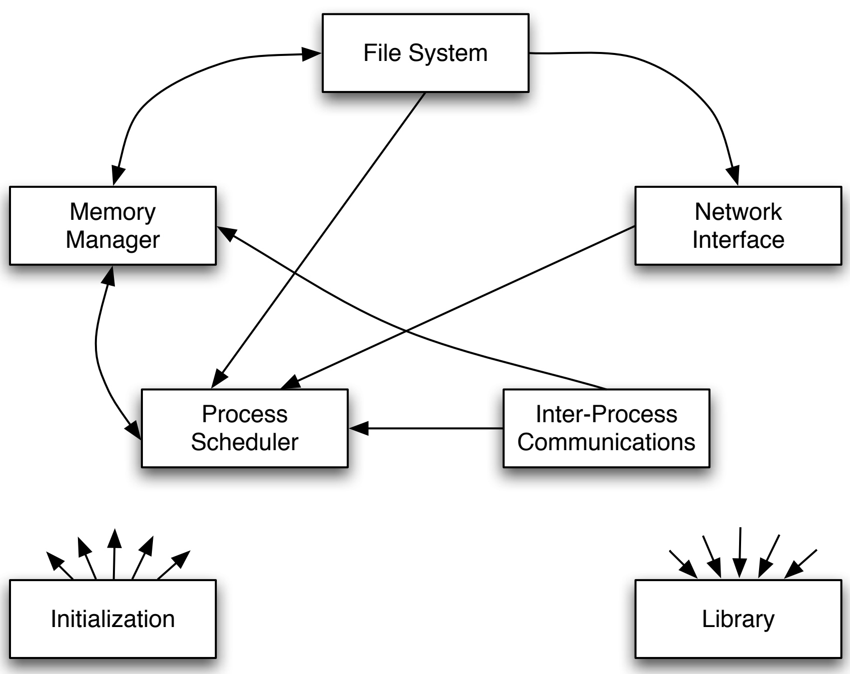
- IEEE Definition:
  - Complexity is the degree to which a software system or one of its components has a design or implementation that is difficult to understand and verify
- **Complexity can also be viewed as:**
  - A property that is directly proportional to the size of the system, number of its constituent elements, their internal structure, and the number of their interdependencies

Goal: Reduce Complexity /2

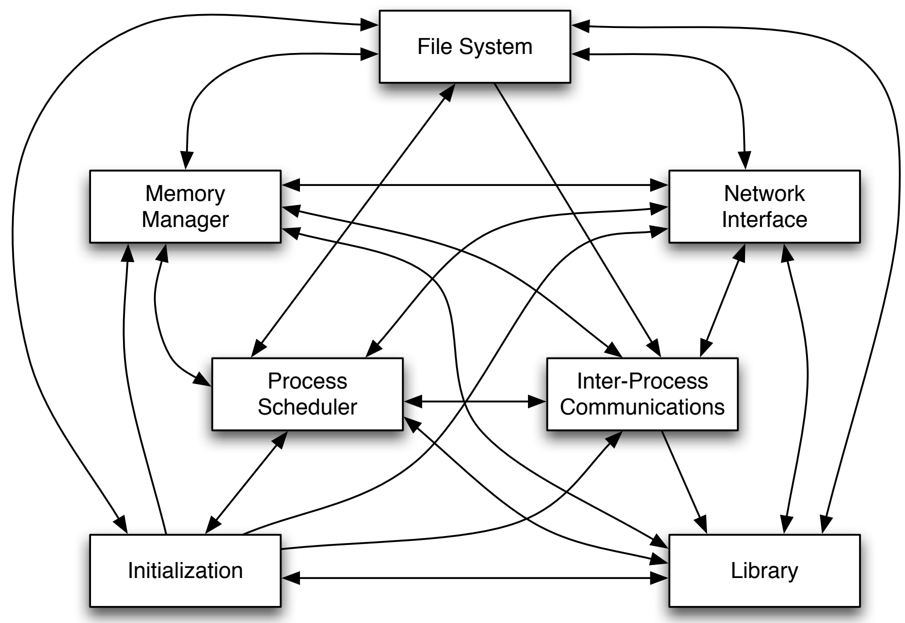
- **Software components and complexity:**
  - Separate concerns into different components
  - Keep only the functionality inside components
    - Move the interactions into connectors
  - Insulate processing components from changes in data format (use abstract data types, or specialized data components)
- **Software connectors and complexity:**
  - Keep only interaction facilities inside connectors
  - Separate interaction concerns into different connectors
    - E.g., split communication (streams) from facilitation (delivery, routing)
  - Support connector composition
  - Restrict interactions facilitated by each connector
- **Architectural configurations and complexity:**
  - Eliminate unnecessary dependencies
  - Use hierarchical decomposition

Goal: Reduce Complexity /3

- **Linux OS Kernel**
  - **Conceptual Architecture:**



- **Linux OS Kernel**
  - **Concrete Architecture:**



Goal: Scalability and Heterogeneity /1

- **Scalability:**
  - The capability of a software system to be adapted to meet new requirements of size and scope
- **Adaptability:**
  - Ability to satisfy new requirements and adjust to new operating conditions during its lifetime
- **Heterogeneity:**
  - Ability to consist of multiple disparate constituents or function in multiple disparate computing environments
- **Portability:**
  - Ability to execute on multiple platforms with minimal modifications and without significant degradation in functional or non-functional characteristics

Goal: Scalability and Heterogeneity /2

- **Software components and scalability:**
  - Define each component to have a simple and understandable interface
  - Distribute the data sources and replicate data when necessary
- **Software connectors and scalability:**
  - Give each connector a clearly defined responsibility
  - Choose the simplest connector suited for the task
- **Architectural configurations and scalability:**
  - Place the data sources close to the data consumers
  - Make use of parallel processing capabilities

Goal: Dependability /1

- **Dependability** is a collection of system properties that allows one to rely on a system functioning as required
  - **Reliability** is the probability that a system will perform its intended functionality under specified design limits, without failure, over a given time period
  - **Availability** is the probability that a system is operational at a particular time
  - **Robustness** is a system's ability to respond adequately to unanticipated runtime conditions
  - **Fault-tolerant** is a system's ability to respond gracefully to failures at runtime

Goal: Dependability /2

- **Software components and dependability:**
  - Provide reflection capabilities in components to check status of a component at runtime
  - Provide suitable exception handling mechanisms and avoid single points of failure
- **Software connectors and dependability:**
  - Employ connectors that strictly control component dependencies
- **Architectural configurations and dependability:**
  - Provide redundancy of critical functionality and data
  - Implement non-intrusive system health monitoring

Food for Thought

- **Read:**
  - Chapter 6 from “Software Architecture: Foundations, Theory, and Practice”
    - Read sections 6.1, 6.2, and 6.3 on architecture modeling
  - Chapter 12 from “Software Architecture: Foundations, Theory, and Practice”
    - Read detailed explanations of design heuristics discussed in the lecture notes