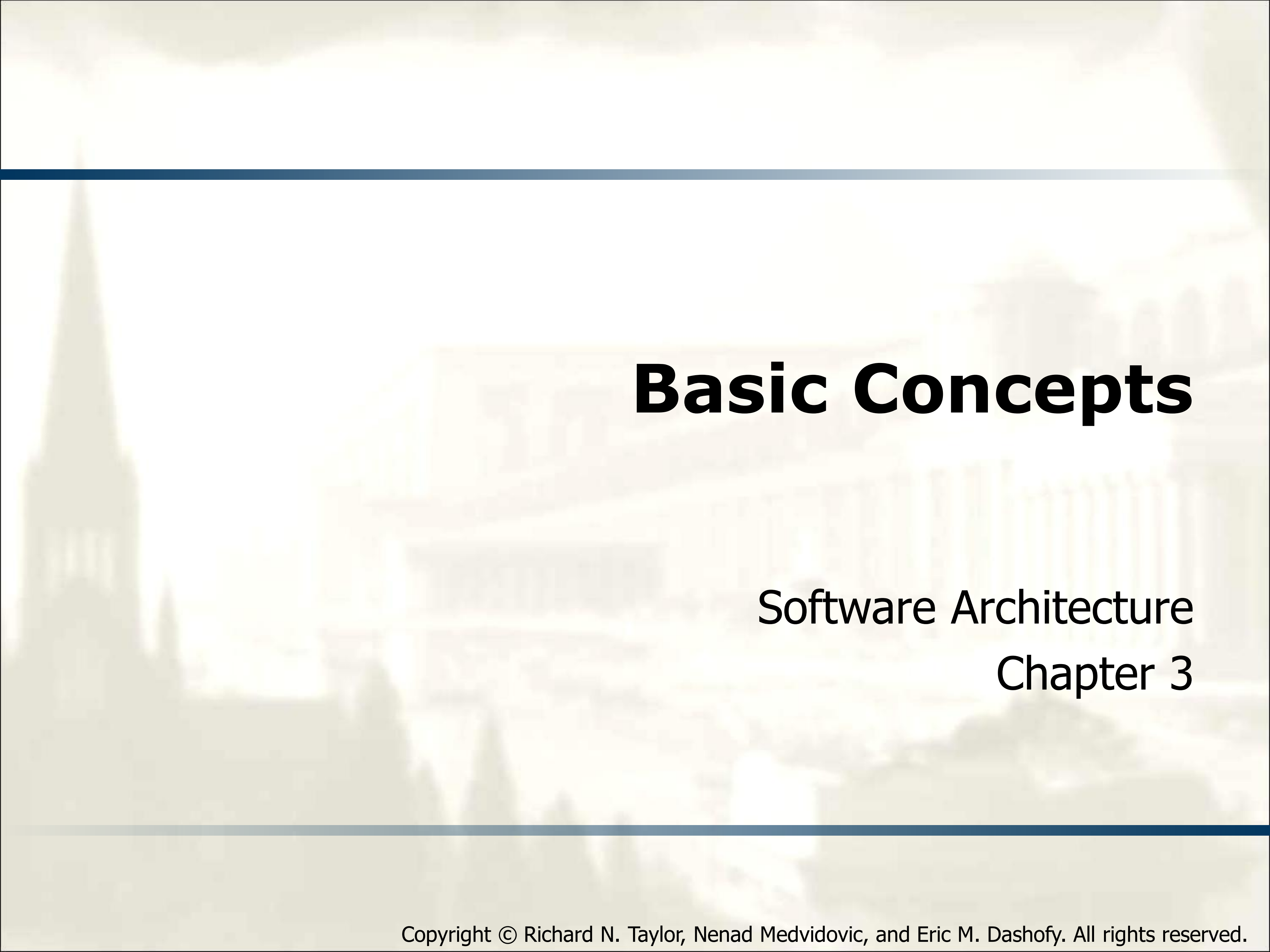


Basic Concepts

Software Architecture

Chapter 3

What is Software Architecture?

- **Definition**
  - A software system's architecture is the set of *principal design decisions* about the system
- Software architecture is the blueprint for a software system's construction and evolution
- Design decisions encompass every facet of the system under development
  - Structure
  - Behavior
  - Interaction
  - Non-functional properties
  - Implementation

What is Software Architecture? (cont'd)

- Structure
  - "The architectural elements should be organized and composed exactly like this …"
- Behavior
  - "Data processing, storage and visualization will be performed in strict sequence."
- Interaction
  - "Communication among all system elements will occur only using event notifications."
- Nonfunctional properties
  - "The system's dependability will be ensured by replicated processing modules"
- Implementation
  - "UI will be built with Java Swing."

What is “Principal”?

- “Principal” implies a degree of importance and topicality that grants a design decision “architectural status” (i.e., makes it an *architectural design decision*)
  - It implies that not all design decisions are architectural
  - That is, they do not necessarily impact a system’s architecture
- How one defines “principal” will depend on what the stakeholders define as the system goals

Other Definitions of Software Architecture

- Perry and Wolf
  - Software Architecture = { Elements, Form, Rationale }

    what *how* *why*
- Shaw and Garlan
  - Software architecture [is a level of design that] involves
    - the description of elements from which systems are built
    - interactions among those elements
    - patterns that guide their composition, and
    - constraints on these patterns
- Kruchten
  - Software architecture deals with the design and implementation of the high-level structure of software
  - Architecture deals with abstraction, decomposition, composition, style, and *aesthetics*

Temporal Aspect

- Design decisions are and unmade over a system’s lifetime

  → Architecture has a temporal aspect
- At any given point in time the system has only one architecture
- A system’s architecture will change over time

Prescriptive vs. Descriptive Architecture

- At any time t, during the process of engineering a software system, a system's *prescriptive architecture* captures the set of design decisions ***P*** made prior to the system's construction, that reflect the intent
  - It is the *as-conceived* or *as-intended* architecture
  - It need not necessarily exist in a tangible form
    - It may be entirely in the architect's mind
    - Alternatively, it may have been captured in a more concrete or formal notation (e.g., using an ADL as we will see later on) or in some form of documentation

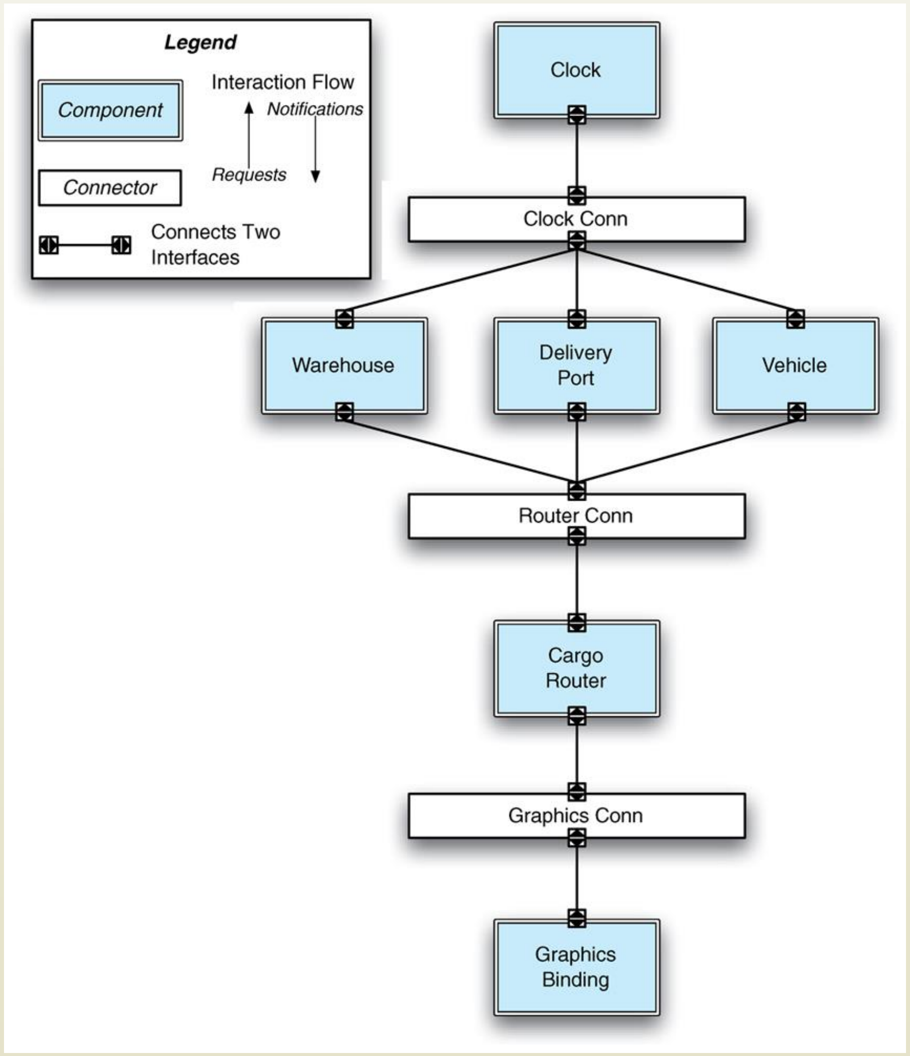
Prescriptive vs. Descriptive Architecture (cont’d)

- In order to realize and refine the prescriptive architecture, we select a set of artifacts ***A***, that embody a set of principal design decisions ***D***
- A system’s *descriptive architecture* describes how the system has been built
  - It is the *as-implemented* or *as-realized* architecture
  - Effectively, it is the embodiment of ***D*** by ***A***
- At the beginning of a system’s inception, the set ***P*** will have some initial architectural design decisions, but the sets ***D*** and ***A*** may be empty

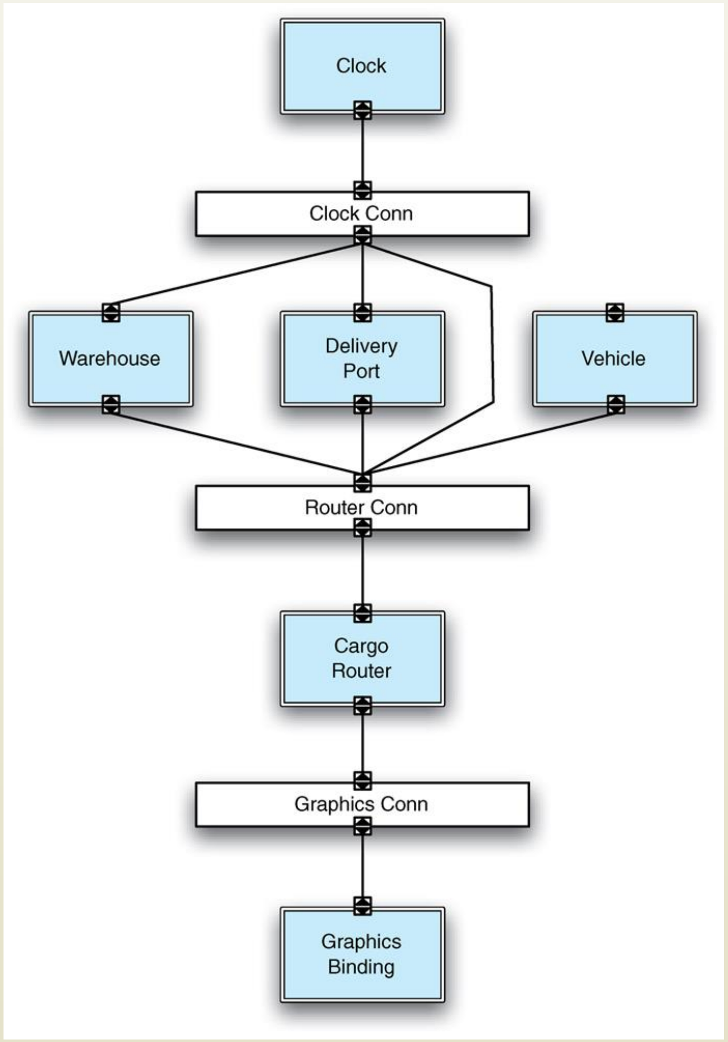
An Example of a Prescriptive vs. Descriptive Architecture

- An application controls the routing of cargo from a set of incoming `Delivery Ports` to a set of `Warehouses` via a set of `Vehicles`
- The `Cargo Router` component tries to optimize the use of vehicles and deliver the cargo to the warehouses
- The `Clock` component provides the time and helps the ports, vehicles and warehouses to synchronize
- The `Graphics Binding` component provides the GUI for human operators to assess the system's state
- The above components interact via three connectors
  - `Clock Conn`, `Router Conn`, `Graphics Conn`

As-Designed Architecture



As-Implemented Architecture



Differences Between the Two Architectures

- The `Vehicle` component is not connected to the `Clock Conn` connector
- Also, the `Router Conn` and the `Clock Conn` connectors are connected
  - This allows `Clock` to interact directly with `Cargo Router` via the two connectors
- Which architecture is correct?
- Are the two architectures consistent with one another?
- What criteria are used to establish the consistency between the two architectures?
- On what information is the answer to the preceding questions based?

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

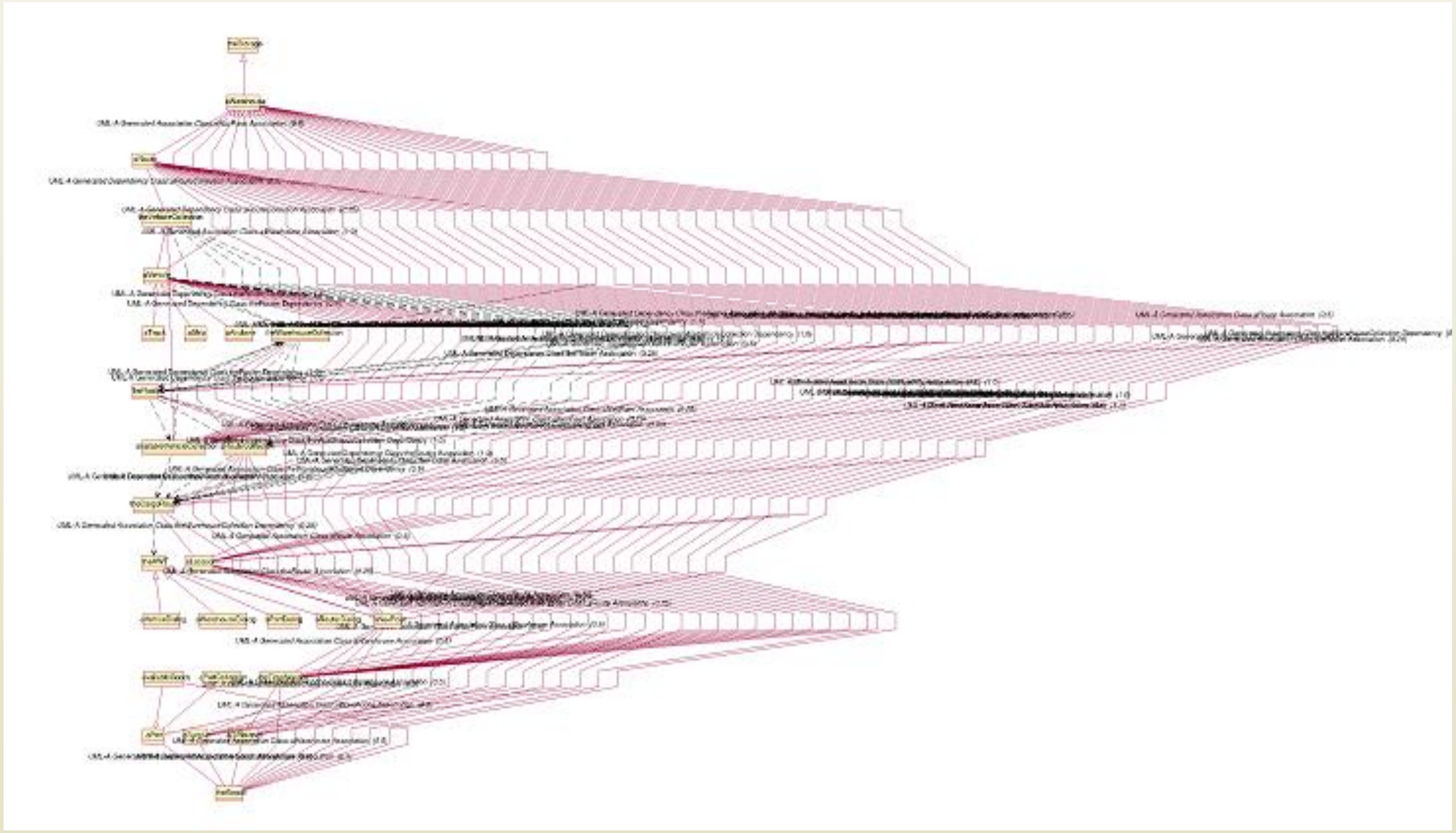
Architectural Degradation

- 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

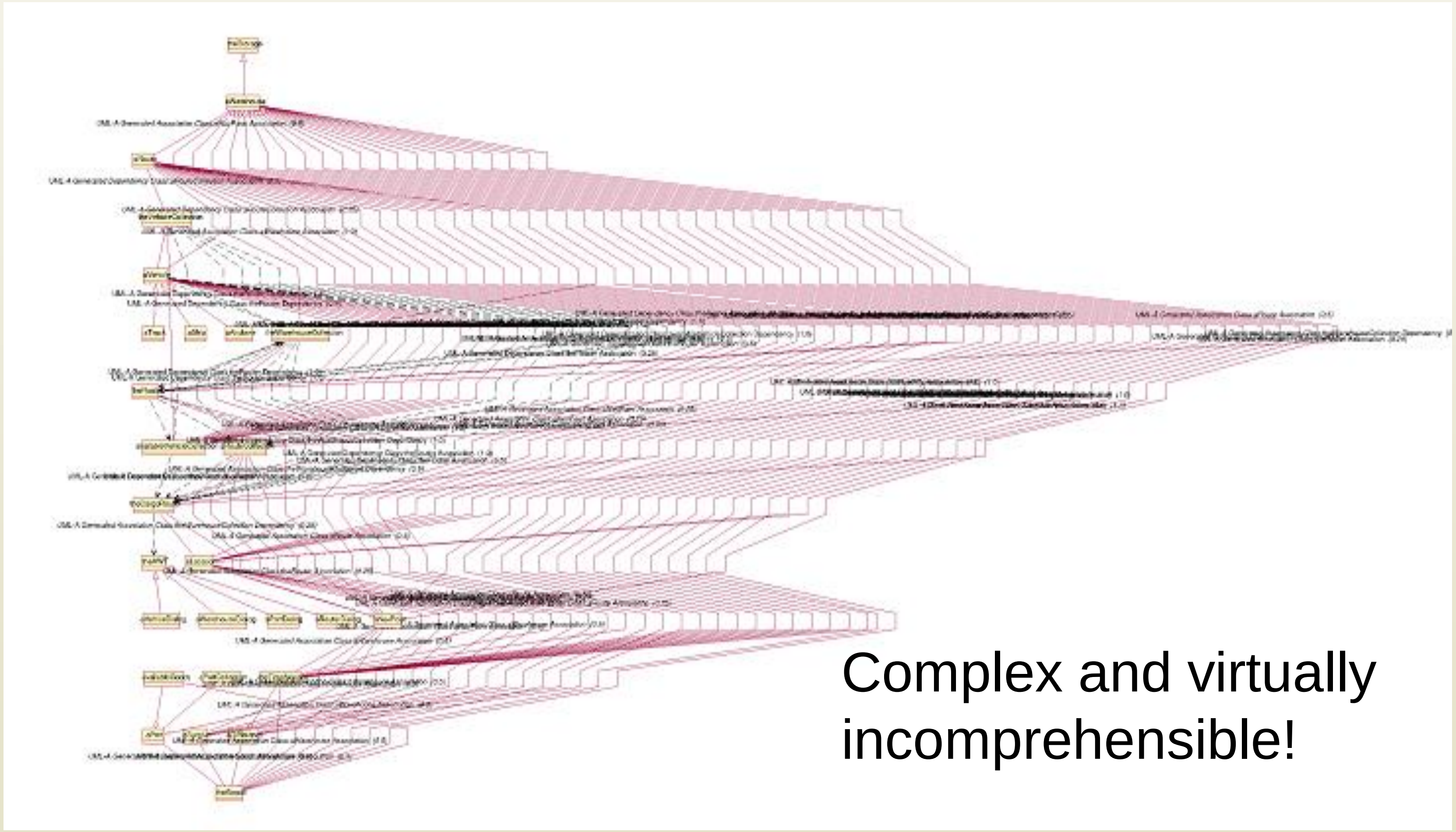
Architectural Recovery

- 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 artifacts can be source code, executable files, Java class files, etc.
- The process of architectural recovery extracts a system’s descriptive architecture
- If complemented with a statement of the architect’s original intent, in principle the system’s prescriptive architecture can be recovered
- Recovery is a very complex and time-consuming process

Implementation-Level View of an Application



Implementation-Level View of an Application



Complex and virtually incomprehensible!

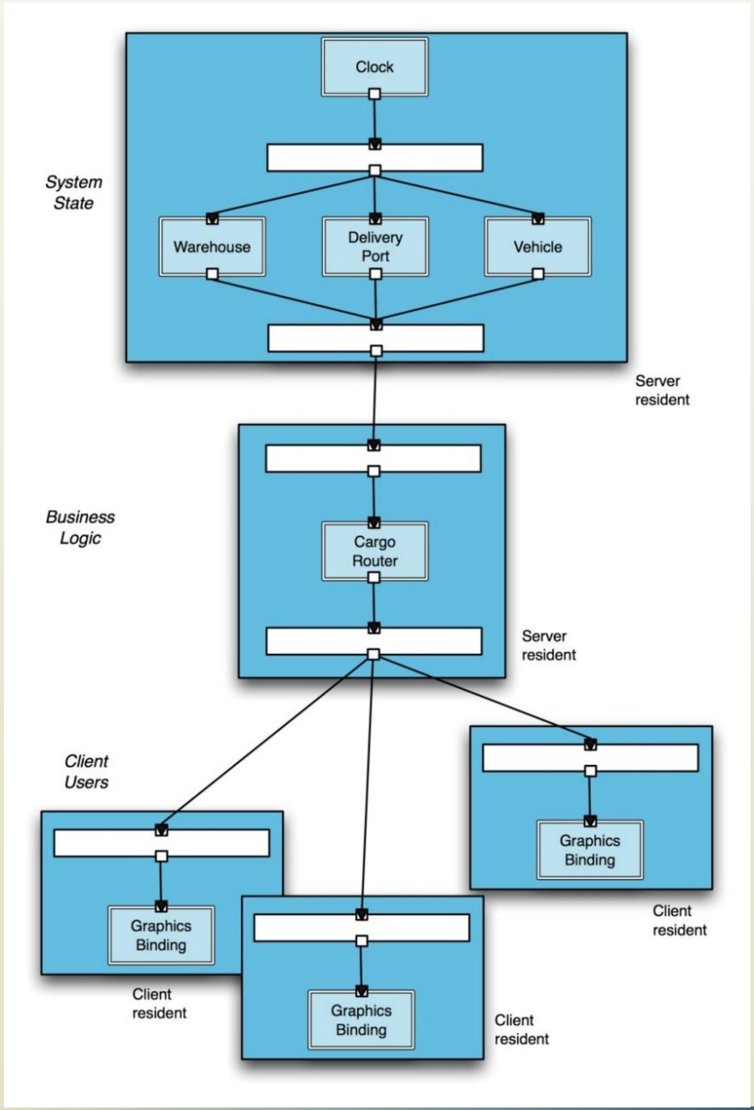
Architectural Perspectives

- An *architectural perspective* is a nonempty set of types of architectural design decisions
- Its purpose is to direct attention to a subset of decisions, say, for purposes of analysis
- The figures on the cargo routing application provide the *structural* perspective for this application
  - It says nothing about, for instance, its behavior, interaction, rationale, and so on
- Another example of perspective is *deployment*

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



Software Architecture’s Elements

- A software system’s architecture typically is not (and should not be) a uniform monolith
- A software system’s architecture should be a composition and interplay of different elements
    - Processing
    - Data, also referred as information or state
    - Interaction

Components

- Elements that encapsulate processing and data in a system’s architecture are referred to as *software components*
- **Definition**
  - A *software component* is an architectural entity that
    - Encapsulates a subset of the system’s functionality and/or data
    - Restricts access to that subset via an explicitly defined interface
    - Has explicitly defined dependencies on its required execution context
- Components typically provide application-specific services

Execution Context that a Component Assumes

- This is a critical factor that makes components usable and reusable across applications
- The extent of the context captured by a component can include:
  - The component's *required* interface, that is the interface to services provided by other components on which this component's operations depend upon
  - The availability of specific resources (e.g., data file) on which this component relies
  - The required system software (e.g., run time environment, OS, middleware, etc.)
  - The hardware configuration needed to execute the component

Connectors

- In complex systems *interaction* may become more important and challenging than the functionality of the individual components
- **Definition**
  - A *software connector* is an architectural building block tasked with effecting and regulating interactions among components
- In many software systems connectors are usually simple procedure calls or shared data accesses
  - Much more sophisticated and complex connectors are possible!
- Connectors typically provide application-independent interaction facilities

Examples of Connectors

- Procedure call connectors, directly implemented in programming languages, enabling synchronous exchange of data and control between two components
- Shared memory connectors, allowing multiple components to interact for reading or writing
- Message passing connectors
- Streaming connectors
- Distribution connectors, such as RPC
- Wrapper/adaptor connectors, for establishing communication between and integration of different types of components (e.g., legacy systems)

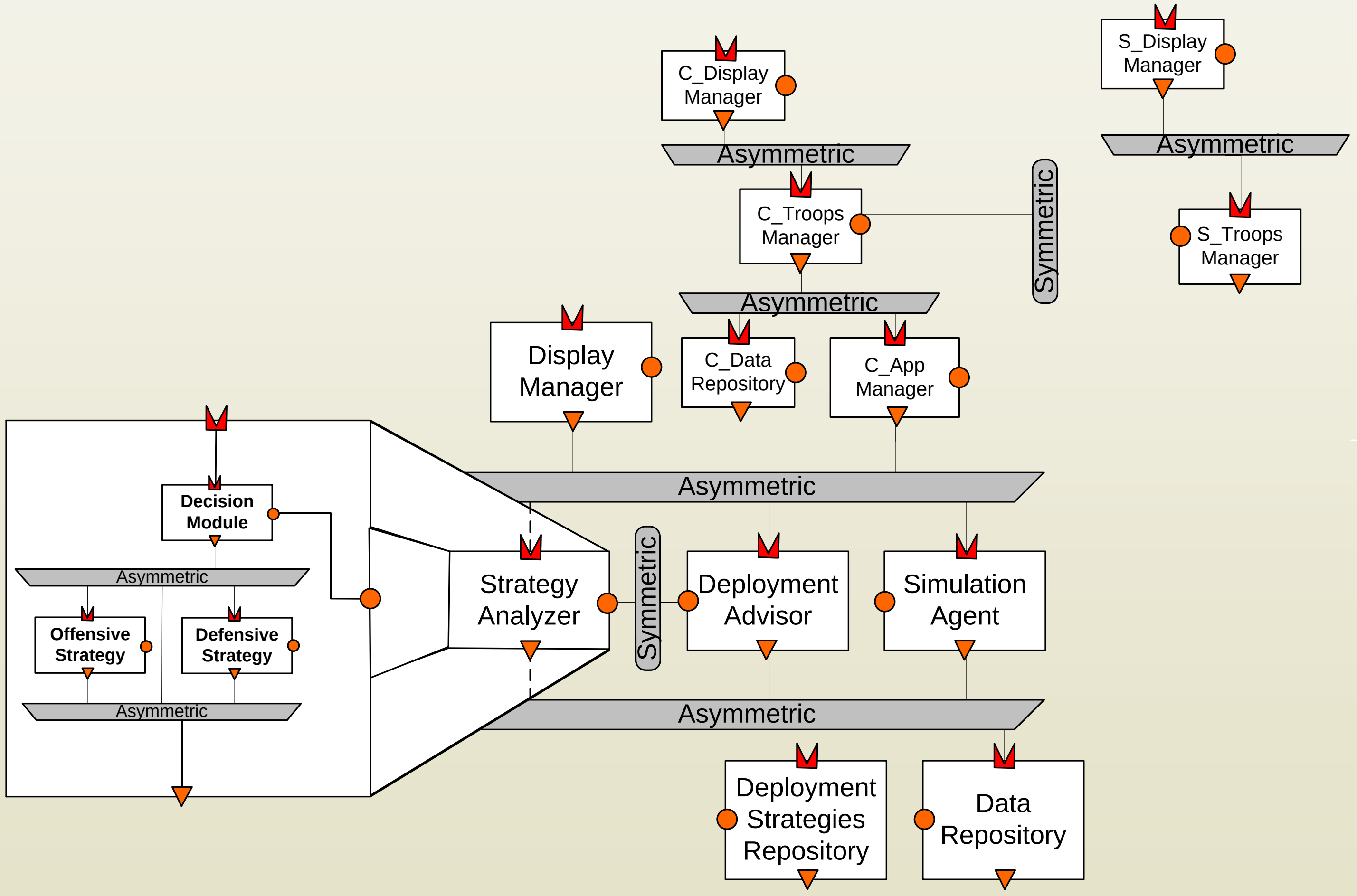
Components vs Connectors

- Typically, components are often targeted at the processing and data capture needs of a particular application or classes of applications, i.e., they are *application-specific*
    - E.g., `Vehicle` and `Warehouse`
    - Or a Web server
- Connectors are *application independent*
    - They are built without a specific application or category of applications in mind and they can be used in all kinds of applications repeatedly

Configurations

- Components and connectors are composed in a specific way in a given system's architecture to accomplish that system's objective
- **Definition**
  - An *architectural configuration*, or topology, is a set of specific associations between the components and connectors of a software system's architecture

An Example Configuration



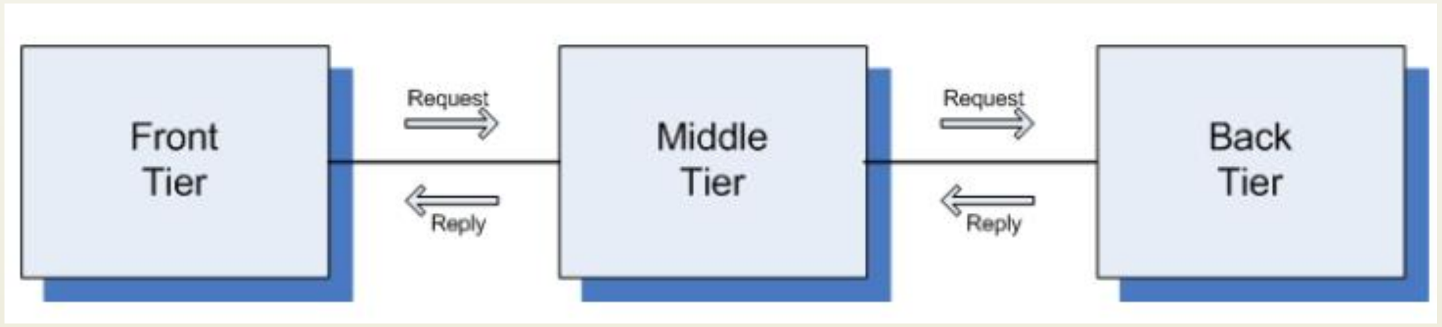
Architectural Styles

- Certain design choices regularly result in solutions with superior properties
  - Compared to other possible alternatives, solutions such as this are more elegant, effective, efficient, dependable, evolvable, scalable, and so on
- **Definition**
  - An *architectural style* is a named collection of architectural design decisions that
    - Are applicable in a given development context
    - Constrain architectural design decisions that are specific to a particular system within that context
    - Elicit beneficial qualities in each resulting system

Architectural Patterns

- **Definition**
  - An *architectural pattern* is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears
- A widely used pattern in modern distributed systems is the *three-tiered system* pattern
  - Science
  - Banking
  - E-commerce
  - Reservation systems

Three-Tiered Pattern



- Front Tier
  - Contains the user interface functionality to access the system's services
- Middle Tier
  - Contains the application's major functionality
- Back Tier
  - Contains the application's data access and storage functionality

Differences Between Styles and Patterns

- **Scope**
  - An architectural style applies to a development *context*
    - "Highly distributed systems" or "GUI-intensive"
  - An architectural pattern applies to a specific design problem
    - "The system's state must be presented in multiple ways."
  - Architectural styles are *strategic* while architectural patterns are *tactical* design tools

Differences Between Styles and Patterns (cont'd)

- **Abstraction**
  - By themselves, architectural styles are too abstract to yield a concrete system design and require human interpretation towards this purpose
  - Architectural patterns are parameterized architectural fragments that can be thought of as concrete pieces of a design
- **Relationship**
  - A system designed according to the rules of a single style may involve the use of multiple patterns
  - Conversely, a single pattern could be applied to systems designed according to guidelines of multiple styles

Architectural Models

- Architecture *Model*
  - An artifact documenting some or all of the architectural design decisions about a system
- Architectural *Modeling*
  - The reification and documentation of those design decisions
- Architectural Modeling *Notation*
  - A language or means of capturing design notations

Architectural Processes

- Architectural design
- Architecture modeling and visualization
- Architecture-driven system analysis
- Architecture-driven system implementation
- Architecture-driven system deployment, runtime redeployment, and mobility
- Architecture-based design for non-functional properties, including security and trust
- Architectural adaptation

Stakeholders in a System’s Architecture

- Architects
- Developers
- Testers
- Managers
- Customers
- Users
- Vendors

Summary

- Any mature engineering field must be accompanied by a shared, precise understanding of its basic concepts, commonly used models, and processes
- The notions that underlie the field of software architecture are itself the concept of architecture, coupled with the notions of components and connectors, configurations, architectural styles and patterns, models and processes, and stakeholders
- In the remainder of the semester, we will analyze in depth these concepts