

Designing Software Architecture



*Please read Chapter
13 & 14*

Dr. Abdelkarim Erradi

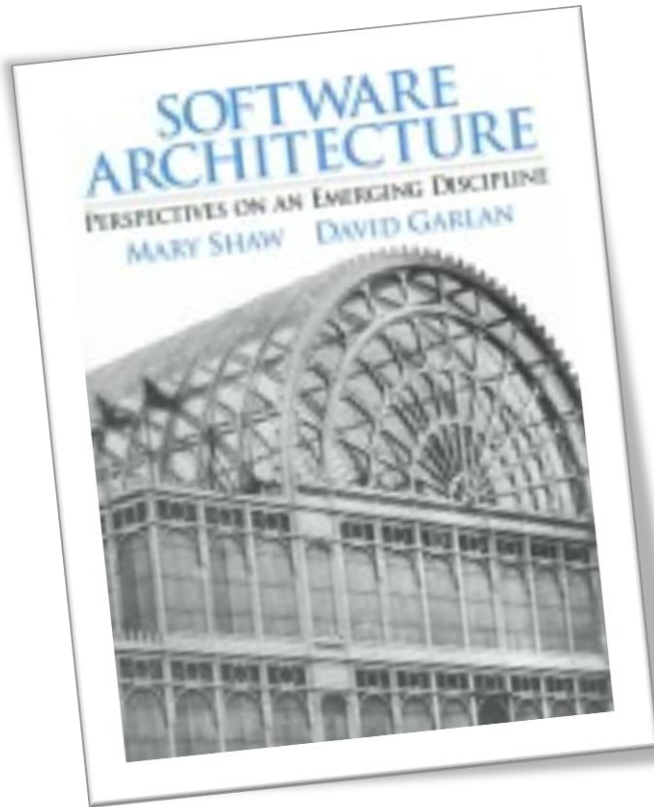
Computer & Engineering Science Dept.

QU

Outline

- What is Software Architecture
- Connectors = how we can integrate components
- Essential Software Quality Attributes
- Documenting Software Architecture

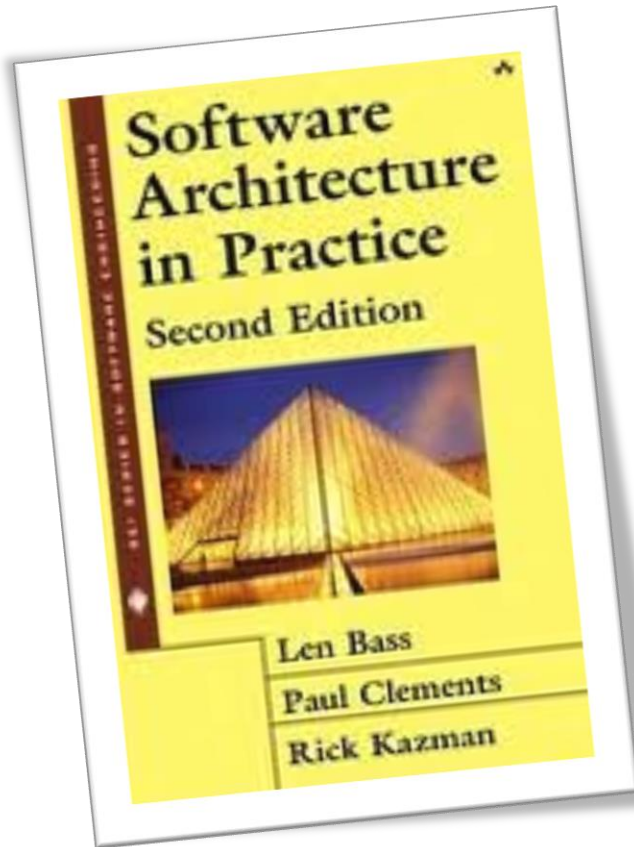
Understanding architecture (1)



“The architecture of a software system defines that system in terms of computational components and interactions among those components.”

Shaw & Garlan,
*Software Architecture: Perspectives on
an Emerging Discipline, 1996*

Understanding architecture (2)



“The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”

Bass, Clements & Kazman,
Software Architecture in Practice, 2003

Software architecture

- Software architecture (SA) describes **how software is decomposed and organized into components.**
 - Architecture = **overall structure** of the system
- **Software architecture = Components + Connectors**
(e.g., method invocation is example of connectors)
- Architecture supports stakeholder communication and early evaluation of design

Software Architecture Elements

- A **component** is a *physical, replaceable* part of a system containing an implementation which conforms to a set of *interfaces* and *realizes* these.
- A **connector** is an mechanism that mediates communication, coordination, or cooperation among components.

Software Architecture Process

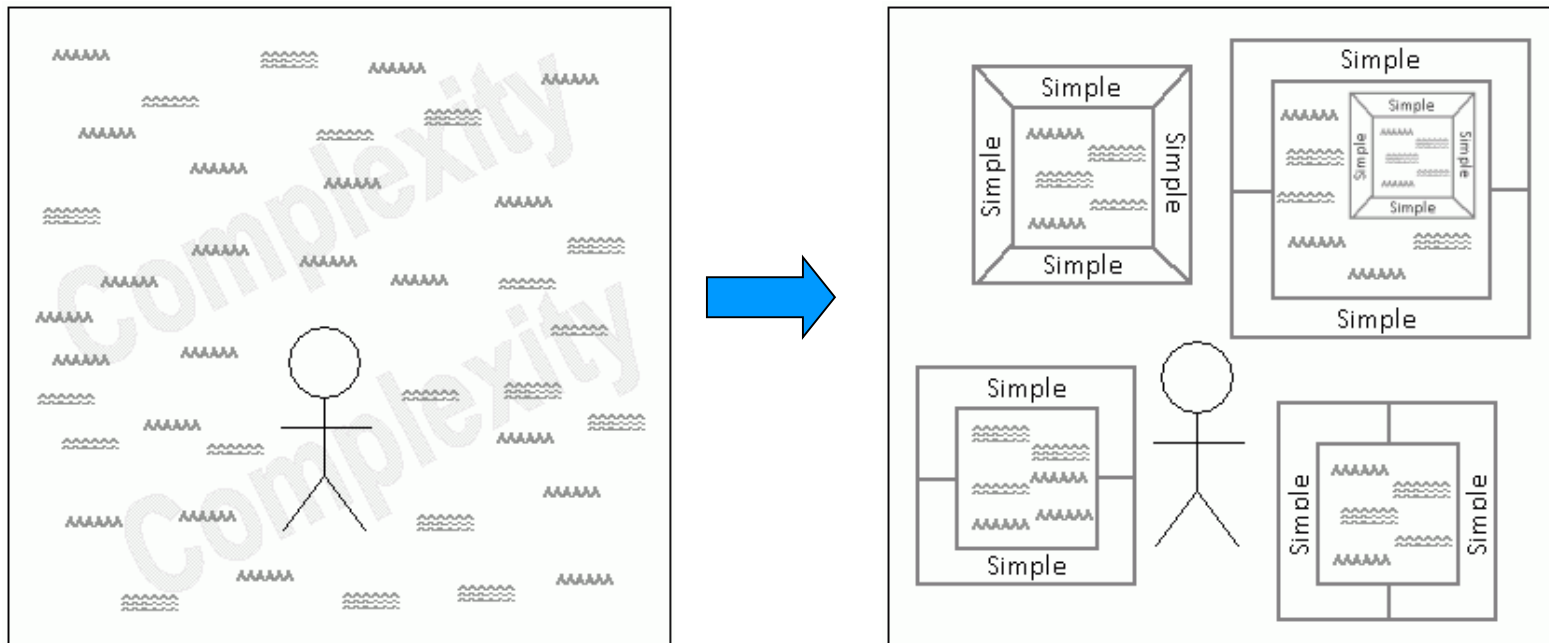
- Designing a *Software architecture* include:
 - Divide the software into **components**
 - Design the **interfaces** of components: what each component can do
 - Decide the **connectors**: how components will **interact** within the system and with other systems: how components communicate
 - Composition of these structural and behavioral elements into larger subsystems
 - Applying an architectural style that guides this organization
 - Choose among the various **architectural patterns** to help us to decompose our system more effectively to meet Non Functional Requirements (NFRs)
 - Defines the rationale behind the components and the structure

Fundamental Principals

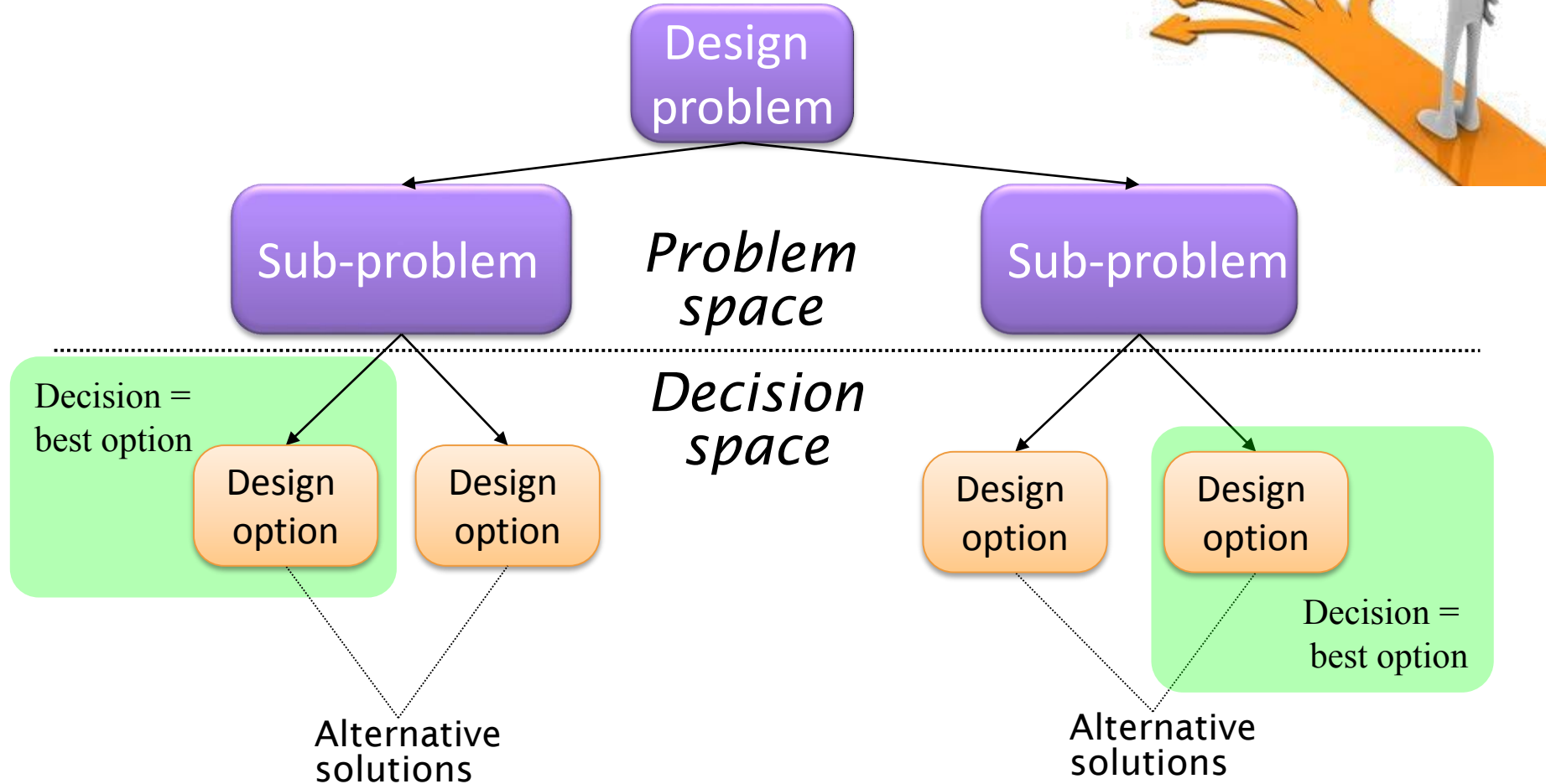
- Goal of software architecture = satisfy the Non-Functional Requirements (NFR)
 - Architectural choices determines the non-Functional Properties (NFPs) of a software such as the system's overall efficiency, reusability and maintainability.
- Basic fundamental principles guiding the design:
 - **Separation of concerns**: separation of components (computation) from connectors (communication)
 - **Abstraction**: hide the component's complexity behind simple interface
 - **Modularity**: divide the system into components
 - **High cohesion** = cohesive responsibility principle: the component's functions should be functionally related.
 - **Low coupling** = reduce dependencies between components

Design Techniques for Dealing with Software Complexity

- Modularity – subdivide the solution into smaller easier to manage components.
- Information Hiding – hide details and complexity behind simple interfaces
- Hierarchical Organization – larger components may be composed of smaller components.



Taking decisions

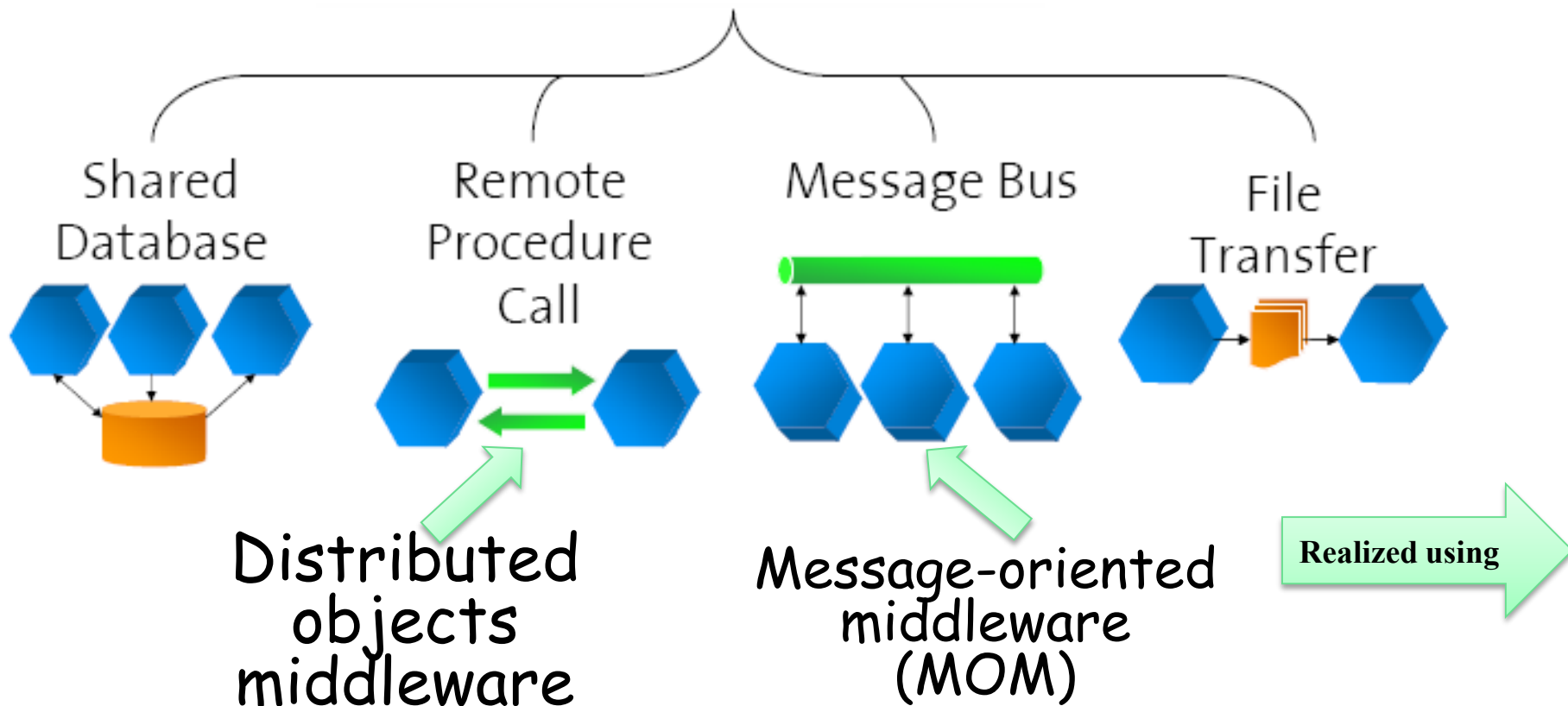


Choose architectural decisions based on their impact on NFRs:

- Build a set of alternate design approaches
- Analyze the cost, quality, and feasibility of the alternatives
- Choose the best solution that maximizes the achievement of NFRs

**Connectors = how we can integrate
components**

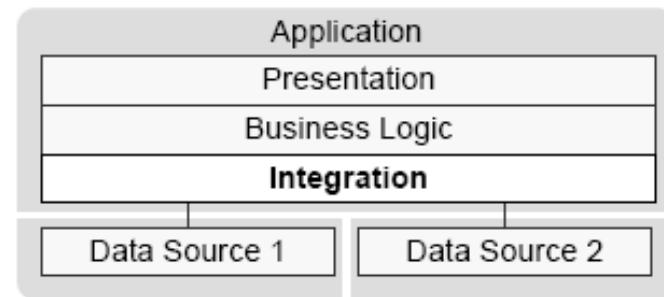
Connectors = Application Integration Styles



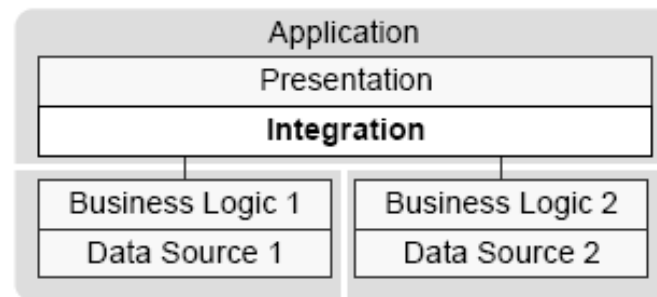
Latest fashion to integrate components is using **Web services** (i.e., systems talk via exchange standard XML messages)

Integration Styles (classified by layer)

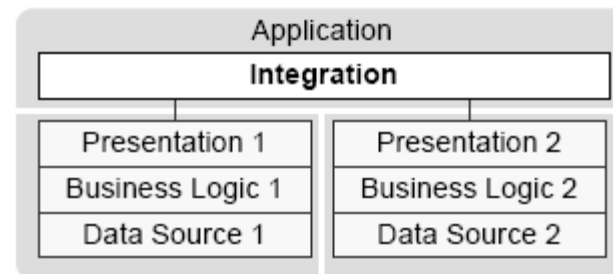
- **Data Integration**
 - Shared/synchronized databases
 - File transfer
- **Business logic / process Integration**
 - Reuse application APIs
 - Remote Method Invocation (RMI), message queues, workflow
- **UI Integration**
 - Portlets and Mashups



(a) Integration of different data sources



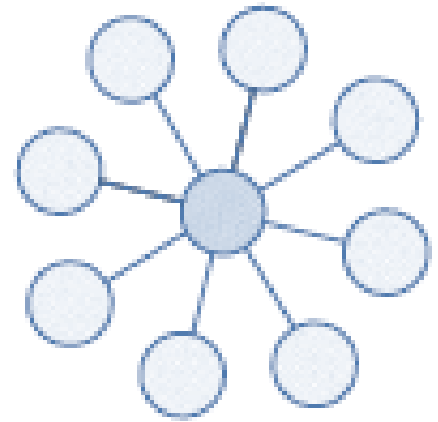
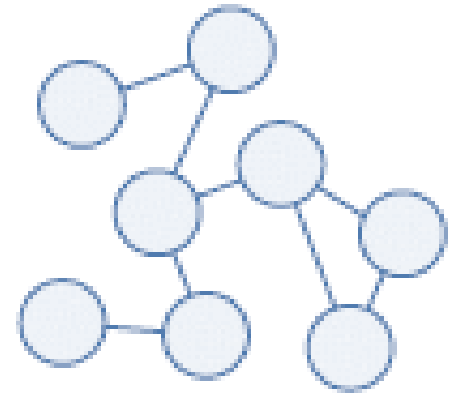
(b) Integration of distributed business logic elements



(c) Integration of two autonomous applications

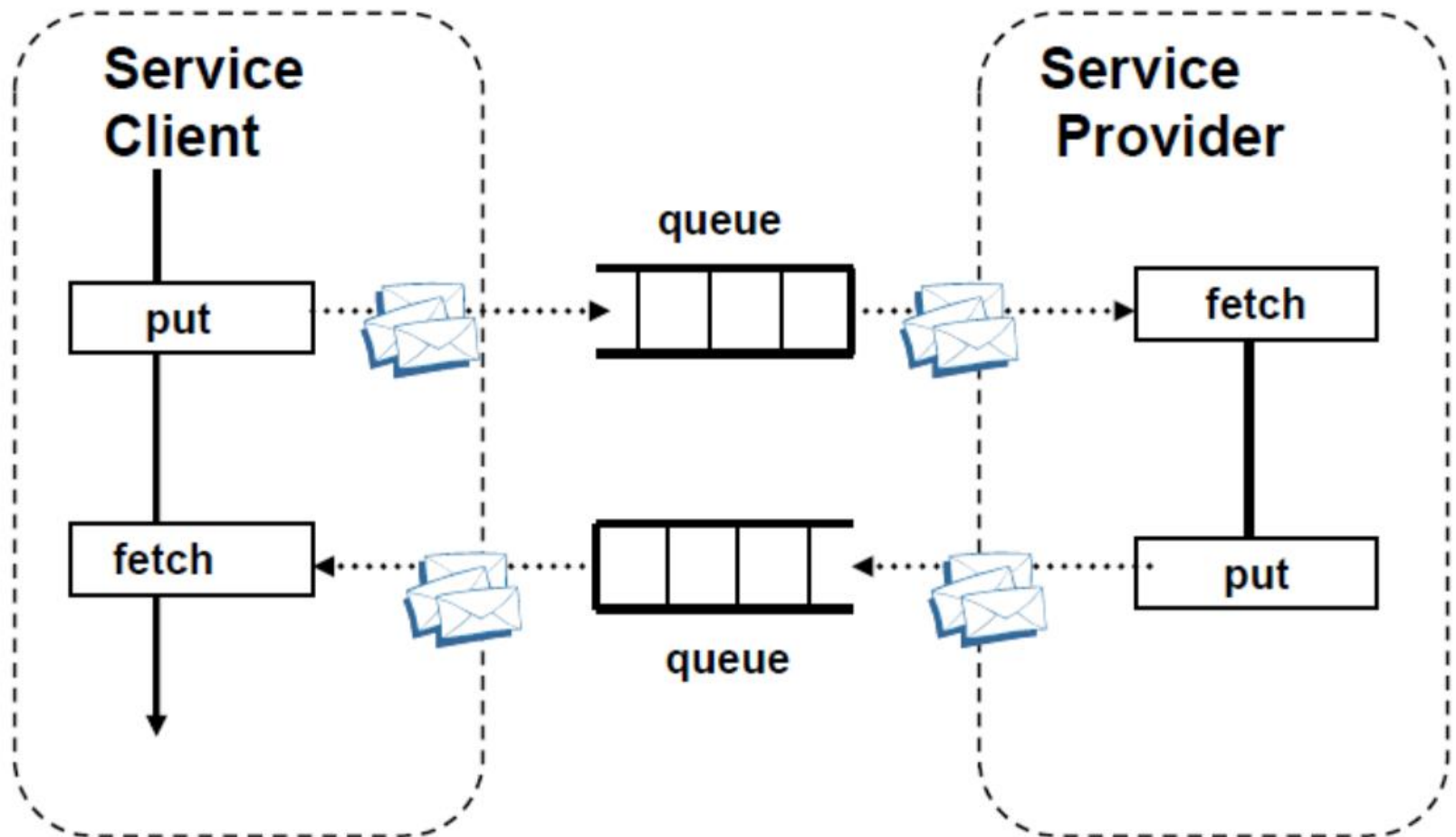
Integration Styles (classified by topology)

- Point-to-point integration
 - Remote Method Invocation (RMI)
 - File transfer
 - Shared database
- Hub-and-spoke integration
 - Message brokers
 - Object brokers (such as Java EE)
 - Workflow systems




For more details see design patterns slides

MOM-based Connector



Aimed at achieving **decoupling** and **reliability**

 = Messages

Essential Software Quality Attributes



Software Quality Attributes

- Performance
- Availability
- Usability
- Security

End user's
view

- Maintainability
- Portability
- Reusability
- Testability

Developer's
view

- Time to market
- Cost and benefits
- Projected life time
- Targeted market
- Integration with legacy systems

Business view

What are Quality Attributes

- Often know as –ilities
 - Reliability
 - Availability
 - Scalability
 - Performance (!)
 - Usability
 - Portability
- Part of a system's NFRs
 - “how” well the system achieves its functional requirements
- Architects are often told: “My application must be fast/secure/scale”
 - Far too imprecise to be any use at all
- Quality attributes (QAs) **must be made precise/measurable/testable** for a given system design

Examples - Quality Attribute Requirements

Quality Attribute	Architecture Requirement
Performance	Getting courses for a particular semester and a particular program should take less than 15 seconds
Security	Communication with login and payment pages must be encrypted using 64 bit certificates.
Efficiency	The server component must run on a low end server with 1GB memory.
Usability	A new user can register a course in 5min without any prior training.
Availability	The system must run 24x7x365, with overall availability of 0.99.
Reliability	For the Payment service, no message loss is allowed, and all message delivery outcomes must be known with 30 seconds
Scalability	Banner must be able to handle a peak load of 1000 concurrent users during the enrollment period.
Modifiability	Add an android mobile interface to the application in 3 weeks with 5 developers.

Example Constraints

- Some architecture requirements are constraints:
- Constraints impose restrictions on the architecture and are (almost always) non-negotiable.
- They limit the range of design choices an architect can make.

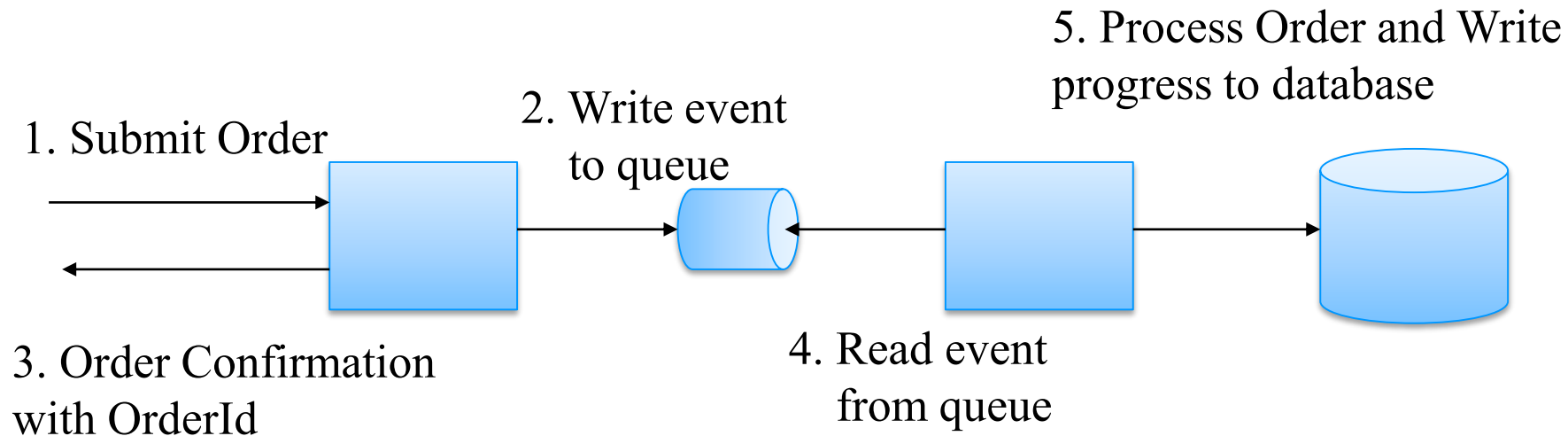
Constraint	Architecture Requirement
Business	The technology must run as a plug-in for MS BizTalk, as we want to sell this to Microsoft.
Development	The system must be written in Java so that we can use existing development staff.
Schedule	The first version of this product must be delivered within six months.
Business	We want to work closely with and get more development funding from MegaHugeTech Corp, so we need to use their technology in our application.

Performance

- Performance Metrics:
 - **Throughput** = Amount of work performed in unit time
 - Transactions per second
 - Messages per minute
 - E.g., System must process 1000 payment transactions per second
 - **Response time** = Measure of the latency an application exhibits in processing a request
 - Usually measured in (milli)seconds
 - Often an important metric for users
 - E.g., Search catalog within 4 seconds for 95% of requests, and all within 10 seconds
 - **Deadline** that must be met: ‘something must be completed before some specified time’
 - Payroll system must complete by 2am so that electronic transfers can be sent to bank

Performance Example - Stock Trading System

- **Response time:**
 - Users should be able to submit orders within 15 seconds
- **Solution :**
 - Decouple user Order Creation from Order Processing using a queue



Scalability

- How well the application will work **when the size of the problem increases.**
 - 2 common scalability issues in IT systems:
 - Increased number of concurrent requests
 - Increased size of processed data
 - e.g., Banner should be capable of handling a peak load of 1000 concurrent course registration requests from students during the 3 week enrollment period.
- => Solution: The Web server can be replicated on a tree machines cluster to handle the increased request load.

Modifiability

- Modifiability measures how easy it **may** be to change an application to cater for new requirements.
 - Must estimate cost/effort
- **Minimizing dependencies increases modifiability**
 - Changes isolated to single components likely to be less expensive than those that cause ripple effects across the architecture.
 - Hide implementation behind well defined interfaces
- Modifiability Scenarios:
 - Add a Mobile Interface to the Banking Application within 5 weeks with 3 developers.
 - If the speech recognition software vendor goes out of business then we should be able to replace this component within 4 weeks with 2 developers.

Availability

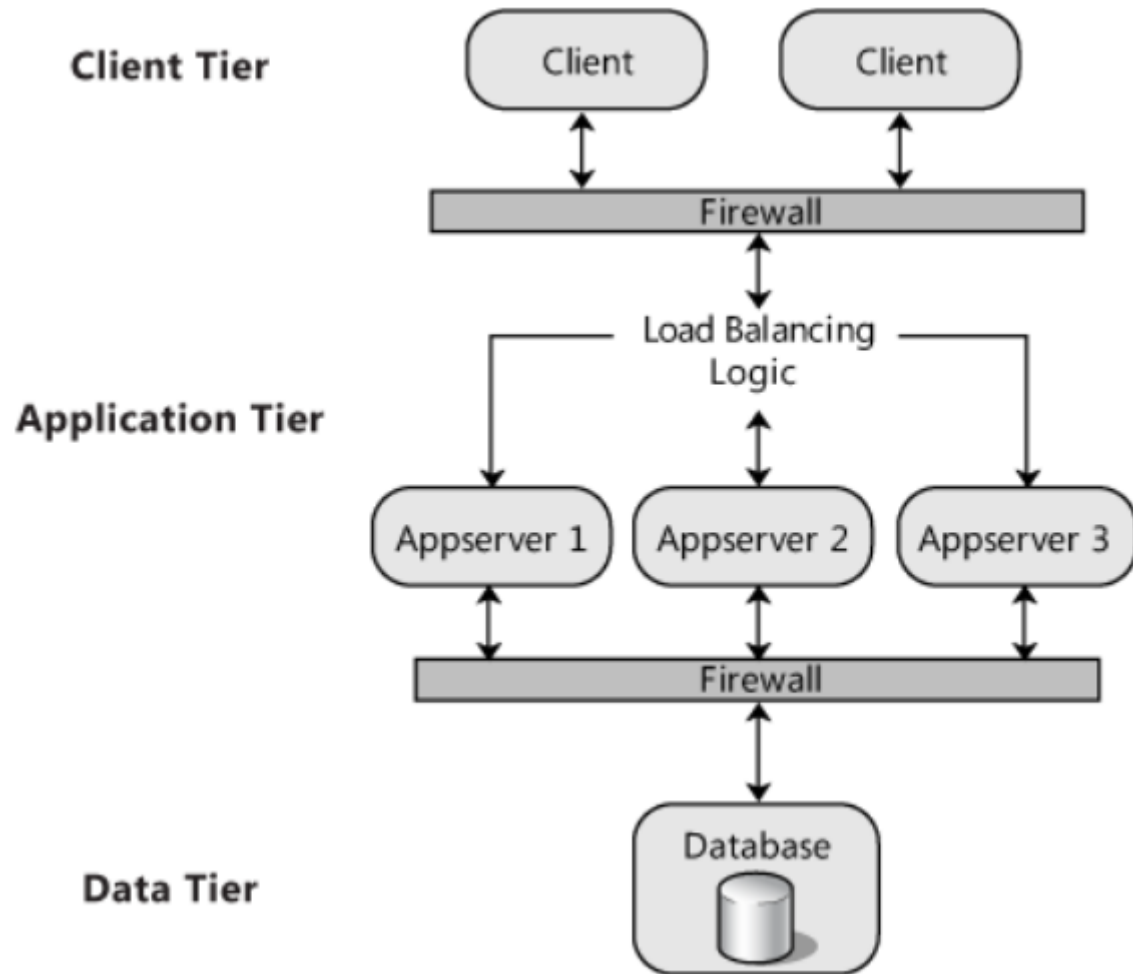
- Measured by the proportion of the required time the application is useable.
 - Key requirement for most IT applications
- Related to an application's reliability
 - Unreliable applications suffer poor availability
- Example scenarios:
 - 100% available during business hours
 - No more than 2 hours scheduled downtime per week
 - 24x7x52 (100% availability)

Availability

- Period of loss of availability determined by:
 - Time to detect failure
 - Time to correct failure
 - Time to restart application
- Key strategy for high availability is:
 - **Eliminate single points of failure using Load-Balancing and failover**

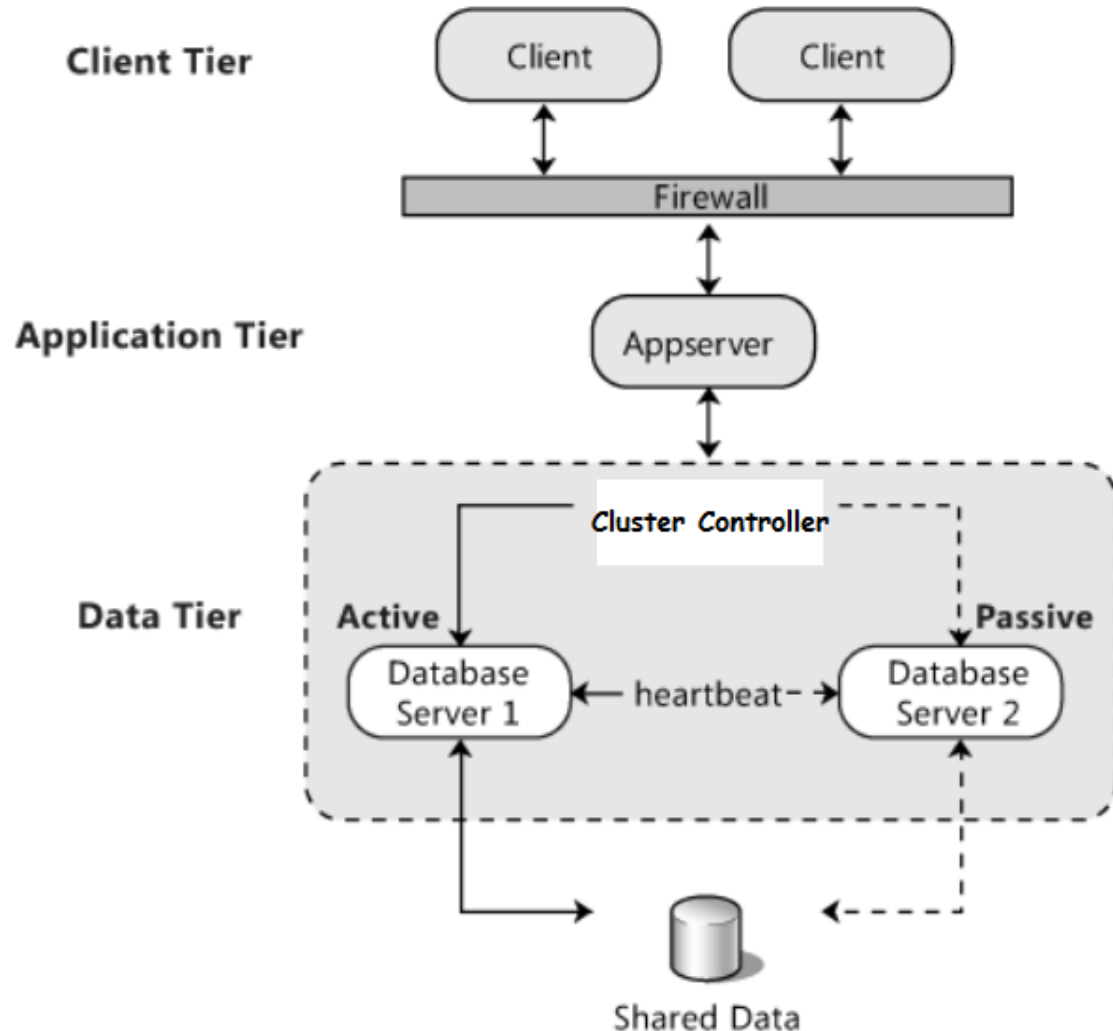
Load-balanced Cluster

- Distribute client requests across multiple servers
- You can install your application components onto multiple servers that are configured to share the workload



Failover Cluster

- Install your application or service on multiple servers that are configured to take over for one another when a failure occurs.



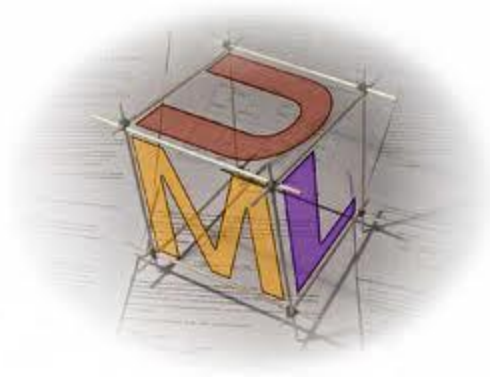
Other Quality Attributes

- Usability
 - How easy it is to use the system and to learn how to use it
- Portability
 - Can an application be easily executed on a different software/hardware platform to the one it has been developed for?
- Testability
 - How easy or difficult is an application to test?
- Supportability
 - How easy an application is to support once it is deployed?

Design Trade-offs

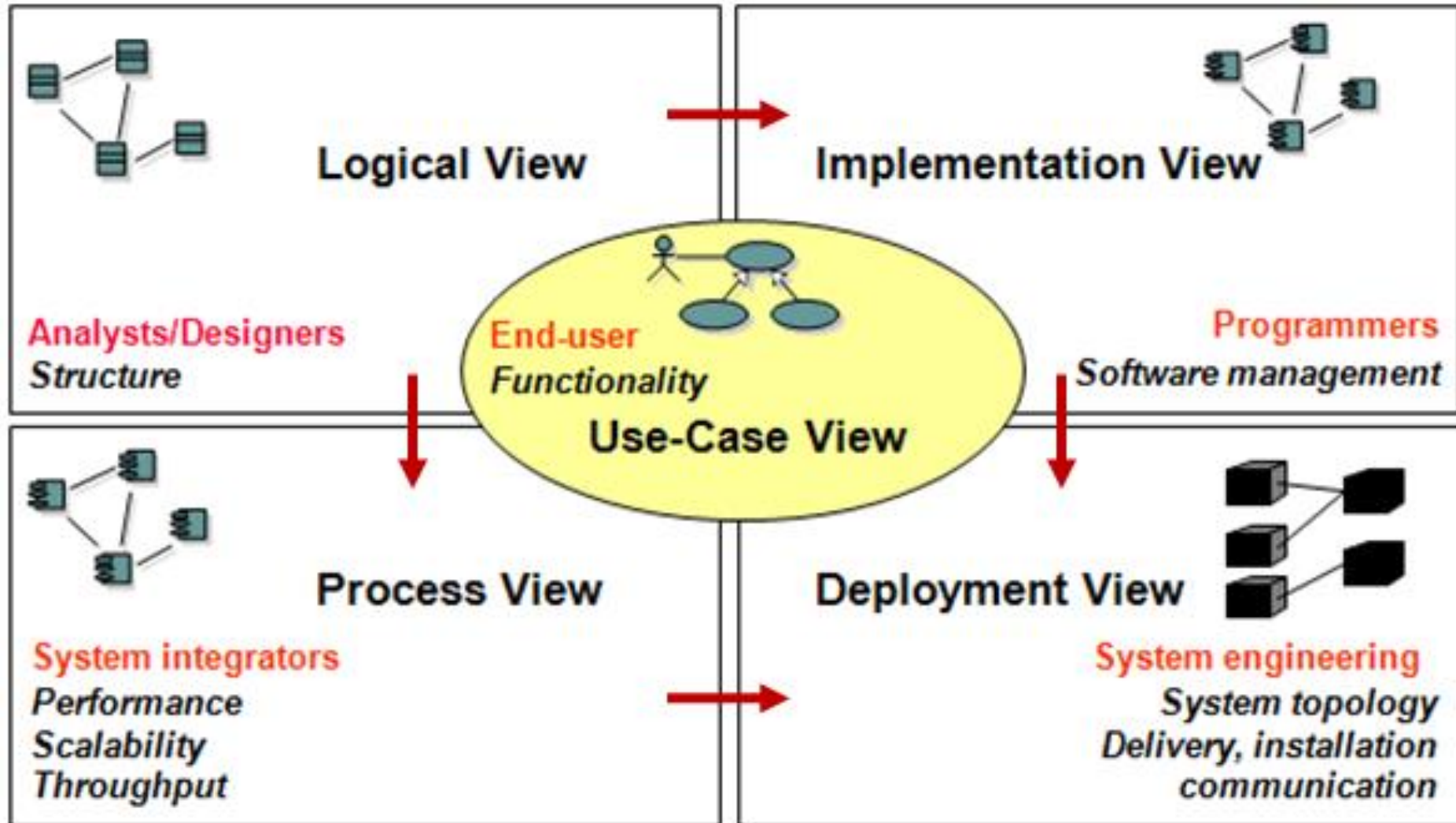
- Quality attributes (QAs) affect each other
 - highly available application may trade-off lower performance for greater availability
 - high performance application may be tied to a given platform, and hence not be easily portable
 - highly secure system may be difficult to integrate
- Architects must design solutions that make the right balance between competing NFRs
 - Not possible to fully satisfy all competing requirements
 - **Understand competing requirements and trade-offs**
 - **Architect must decide which QAs are important for a given application**
 - **Must find the right balance between competing NFRs.**
 - This is the difficult bit!

Documenting Software Architecture



Kruchten's 4+1 view model (1 of 3)

Allow documenting the architecture from different viewpoints



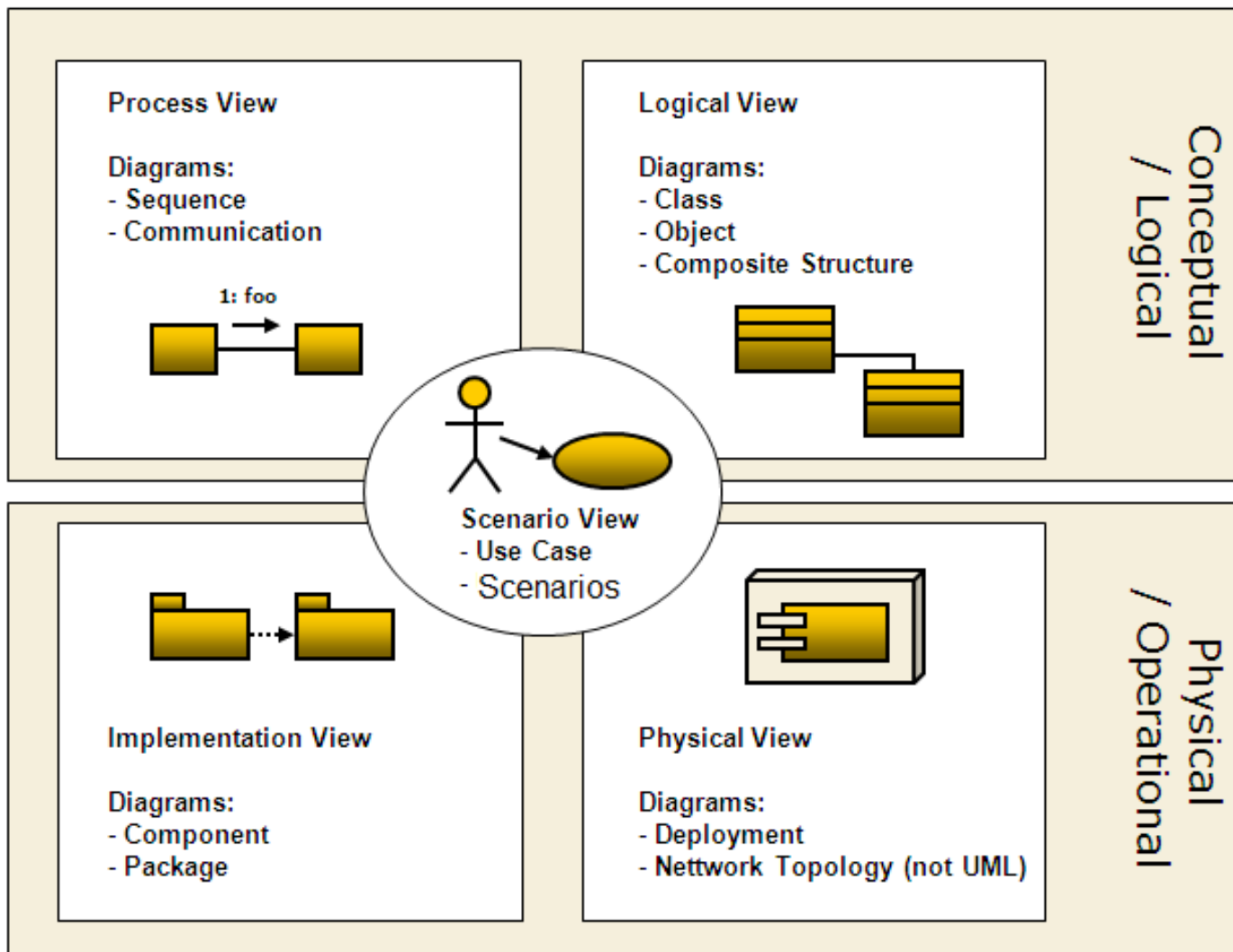
Kruchten's 4+1 view model (2 of 3)

- **Logical view**
 - Describes the structures of the software that solve the functional requirements
 - Shows the key classes and relationships
 - Relevant UML diagrams: Class Diagram, Object Diagram
- **Process view**
 - Deals with the dynamic aspects of the system and describes important concurrency and synchronization issues.
 - Shows the system processes and how they communicate. It explores 'what needs to happen' in a system.
 - Relevant UML diagrams: Activity Diagram, Sequence Diagram
- **Implementation view**
 - Focuses on the module organization of the software.
 - Shows software decomposition into units of implementation (i.e., components): shows how classes are organized into packages, and outlines the dependencies between packages.
 - Relevant UML diagrams: Package Diagram and Component Diagram
- **Deployment view**
 - Describes physical infrastructure: the machines that run the software and how components, objects, and processes are deployed onto those machines and their configurations at run-time
 - Relevant UML diagrams: Deployment Diagram

Kruchten's 4+1 view model (3 of 3)

- **The scenarios**
 - Consists of a subset of important scenarios (e.g., use cases) that the architecture is meant to satisfy
 - The chosen scenarios are those that are the most important to solve because they are either:
 - the most frequently executed or
 - they pose some technical risk or unknown that must be proven out by the architecture
- Plays **two critical roles**:
 - acts as a driver to help designers discover architectural elements
 - validates and illustrates the architecture design: the other four views are centered on the set of scenarios that are chosen for the creation of the architecture

4+1 Views with appropriate diagrams



Process View

- Sequence diagrams
- Communication diagrams
- Activity diagrams

Logical View

- Class diagrams
- Object Diagrams

Implementation View

- Component Diagram
- Package diagram

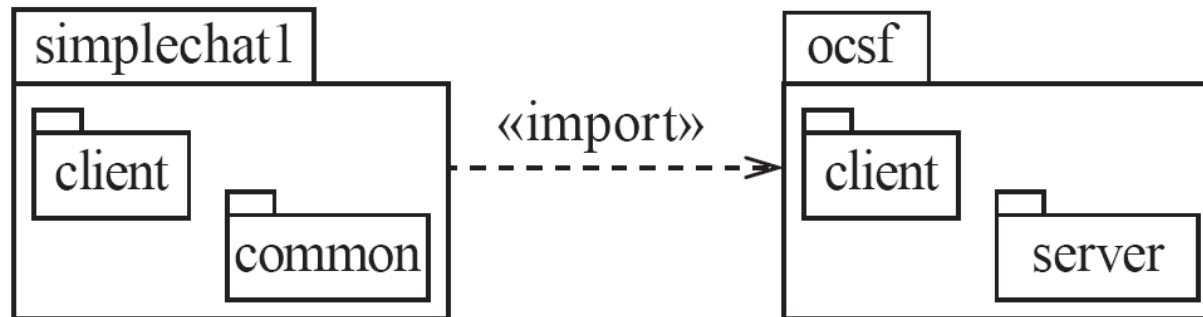
Physical View

- Deployment diagram
- Network topology (not UML)

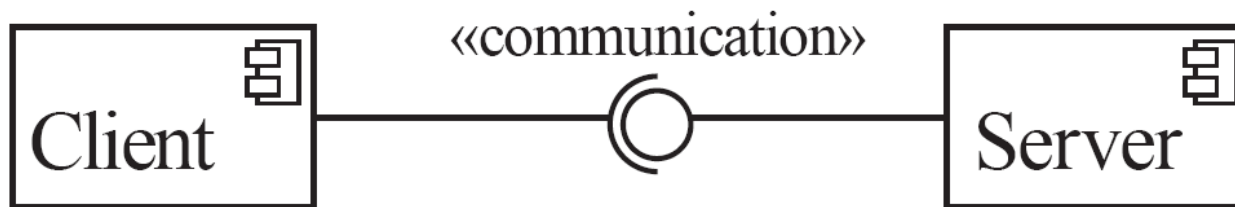
Describing an architecture using UML

- All UML diagrams can be useful to describe aspects of the architectural model
- Tree UML diagrams are particularly suitable for architecture modelling:
 - Package diagrams
 - Component diagrams
 - Deployment diagrams

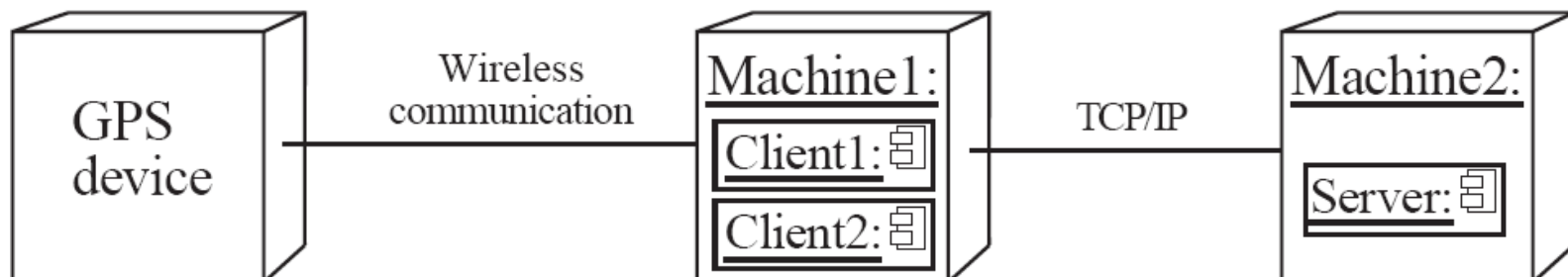
Package diagrams



Component diagrams

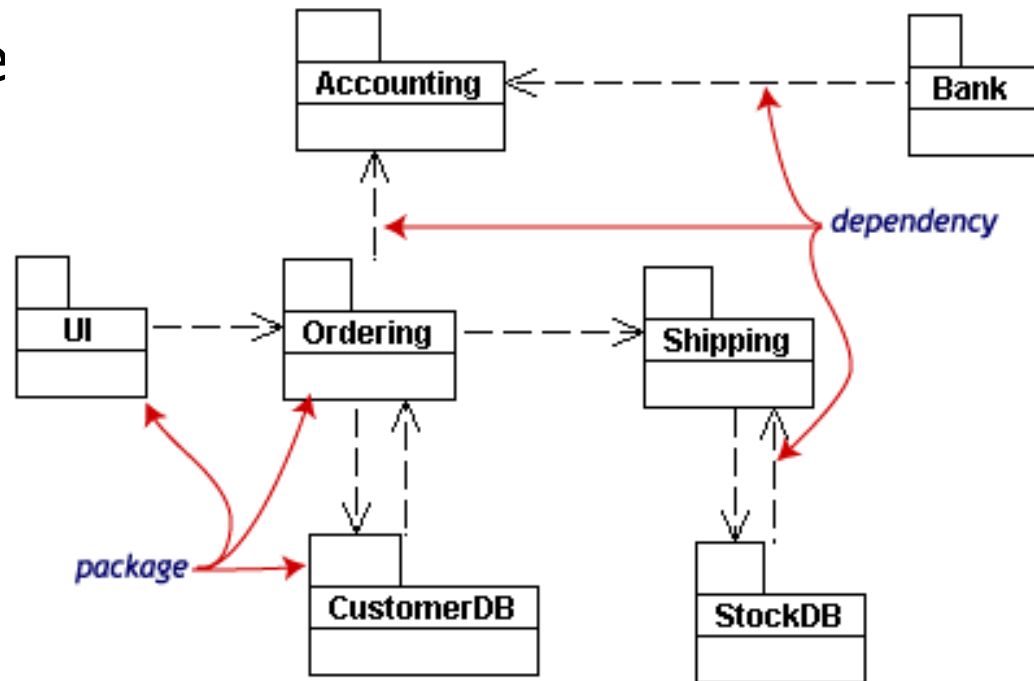


Deployment diagrams

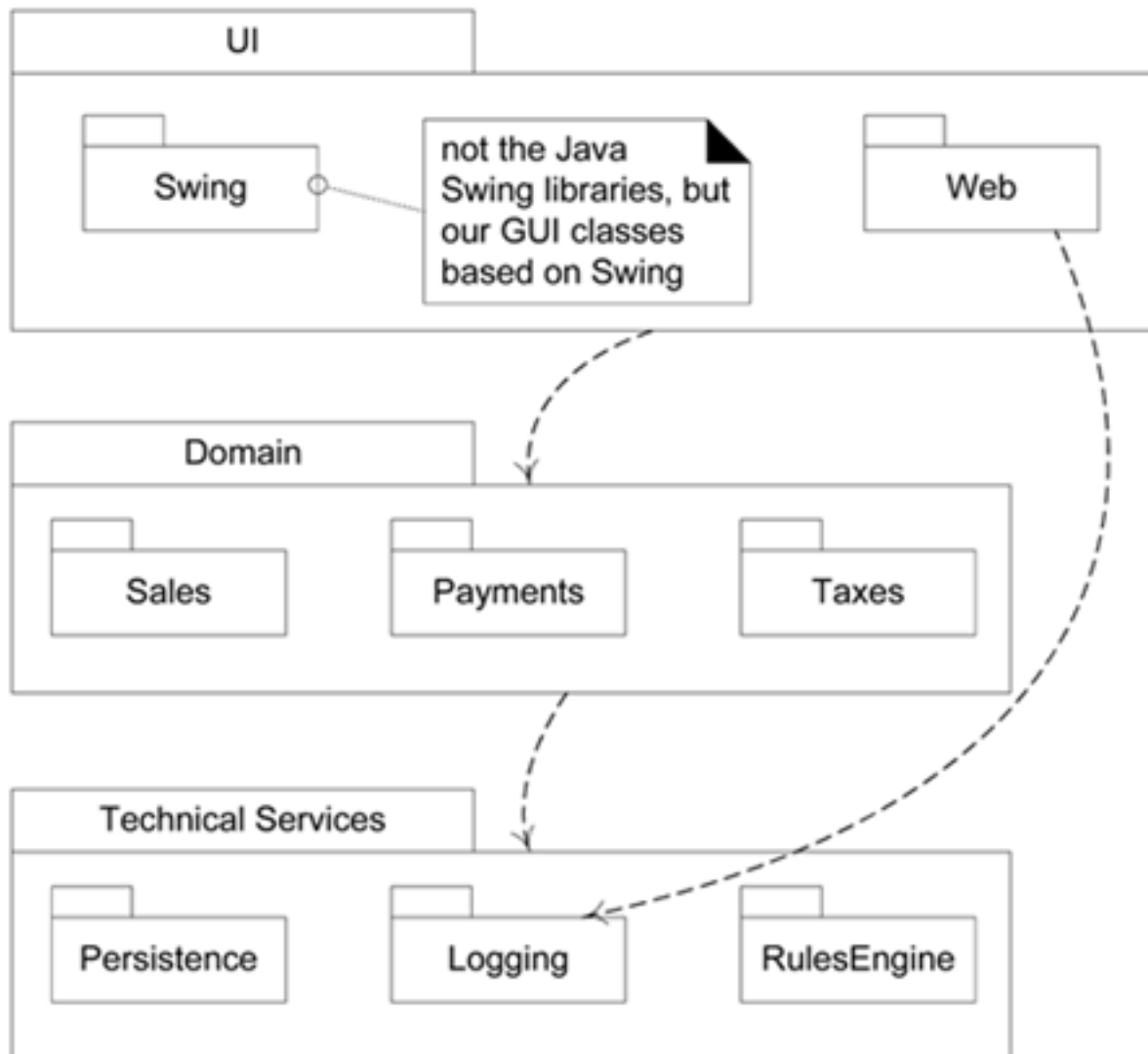


Package Diagram

- Packages are used to group related classes under a single namespace
 - A large project can easily require hundreds of classes, so packages become an efficient organizing tool.
- Package diagram shows the decomposition of the system into **packages** and their **dependencies**



Example - Logical architecture using a UML package diagram



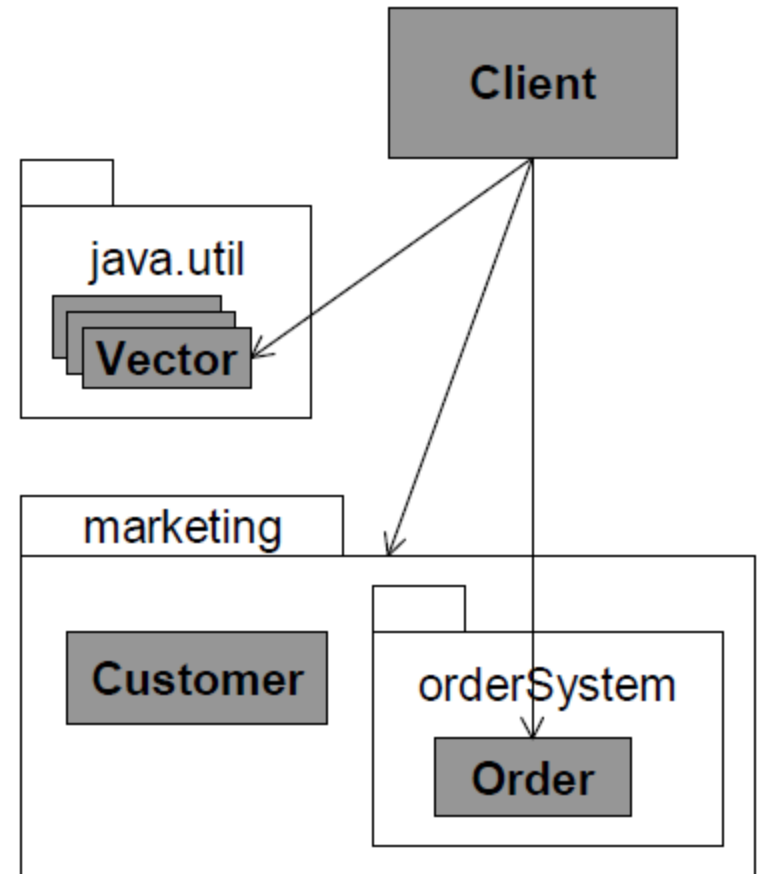
Example: Packages in Java

Users (clients) import packages:

```
import marketing.*;  
import marketing.orderSystem.Order;  
import java.util.Vector;  
class Client {...}
```

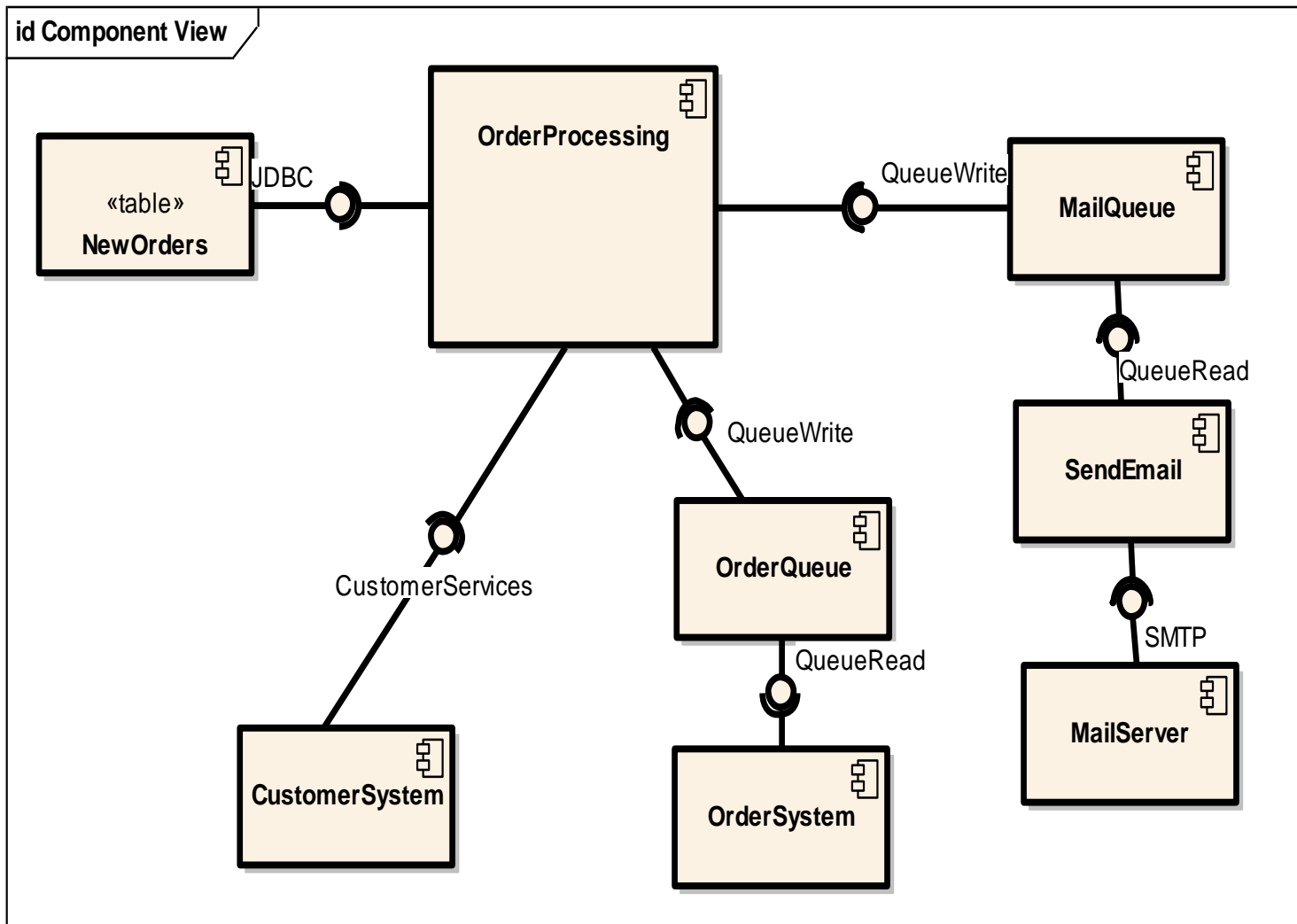
Package Declaration in Java:

```
package marketing;  
public class Customer { ... }  
  
package marketing.orderSystem;  
public class Order { ... }
```

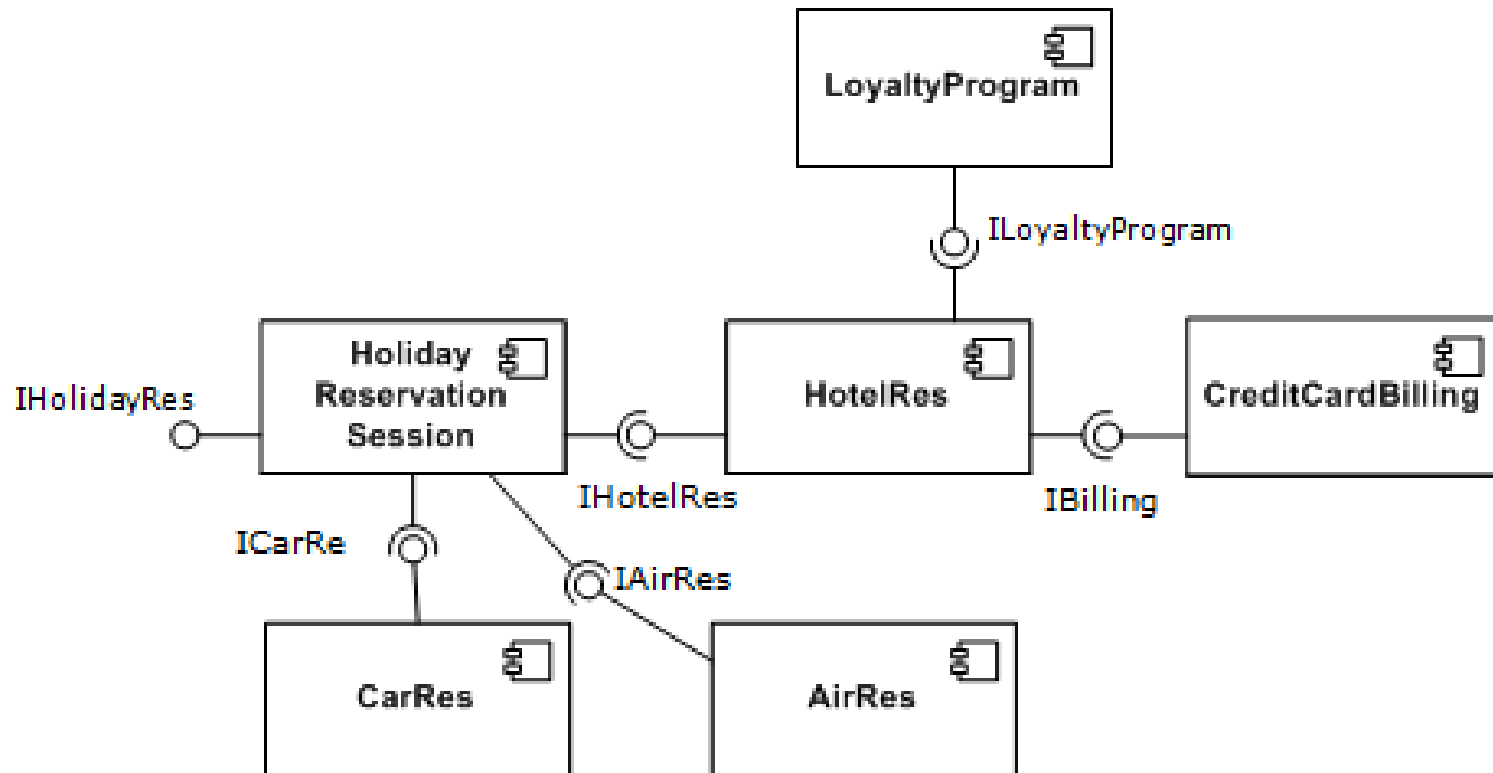


What Is a Component Diagram?

- A diagram that shows the **inter-relationships** and **dependencies among** components

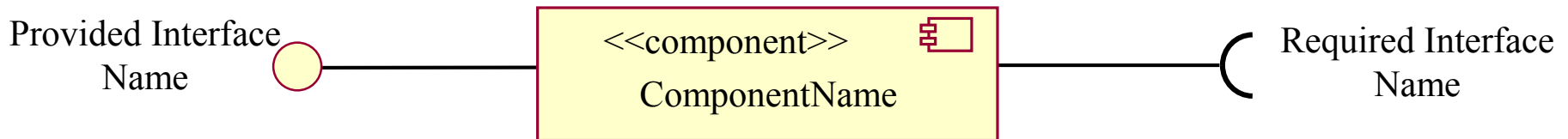


Example Component Diagram



What Is a Component?

- A **component** is a *physical, replaceable* part of a system containing an implementation which conforms to a set of *interfaces* and *realizes* these.
 - It provides the **realization of a set of interfaces**
 - It specifies the dependency to interfaces it requires

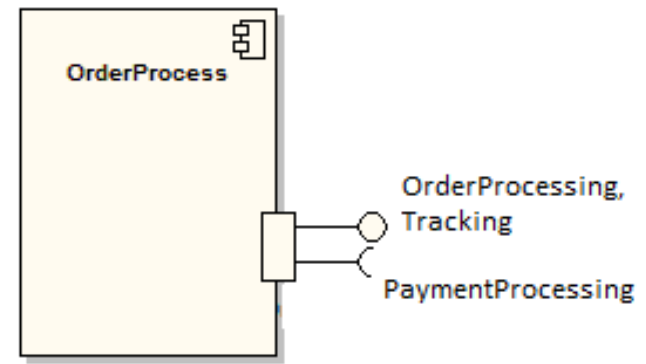


- A component has a clear role and interfaces, allowing you to replace it with a different component that has equivalent functionality.
- Communication between components uses well defined interfaces



Attributes of a Component

- Cohesive responsibility
 - Do not overload components by adding unrelated or mixed functionality
- Visible interface
 - one component provides services that another component requires
- Hidden internal implementation
 - usually a component is implemented by one or more classes
- Loosely coupled
 - Reduce interactions and dependencies to simplify reuse and maintenance
- Autonomous - can be considered to be **stand-alone service providers**
 - Component = unit that you can implement, test, distribute and install separately



=> Components should be reusable, replaceable, extensible, encapsulated, independent, and not context specific.

Packages vs. Components

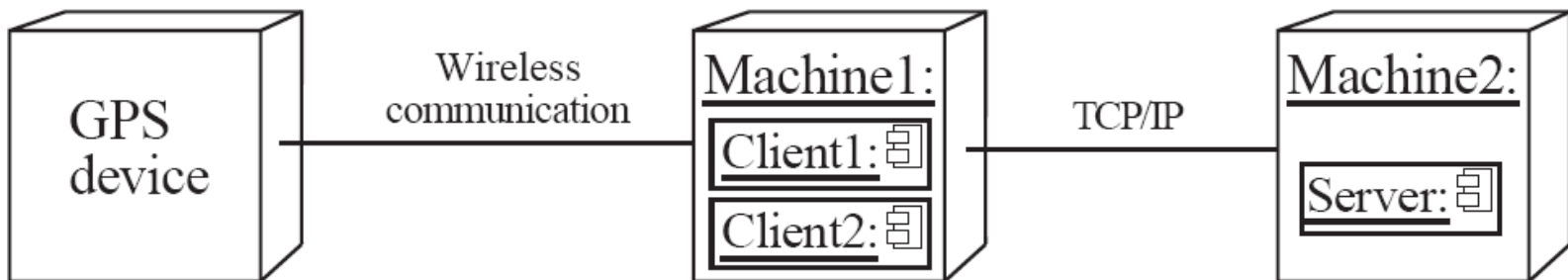
- Packages are logical abstractions
- Components represent physical things – **executable entity** (e.g., jar or dll that can be implemented, tested and deployed separately).

Interface

- A component defines its behavior in terms of provided and required interfaces
- An interface
 - Is the definition of a collection of one or more operations
 - Provides only the operations but not the implementation
 - Implementation is normally provided by a group of classes

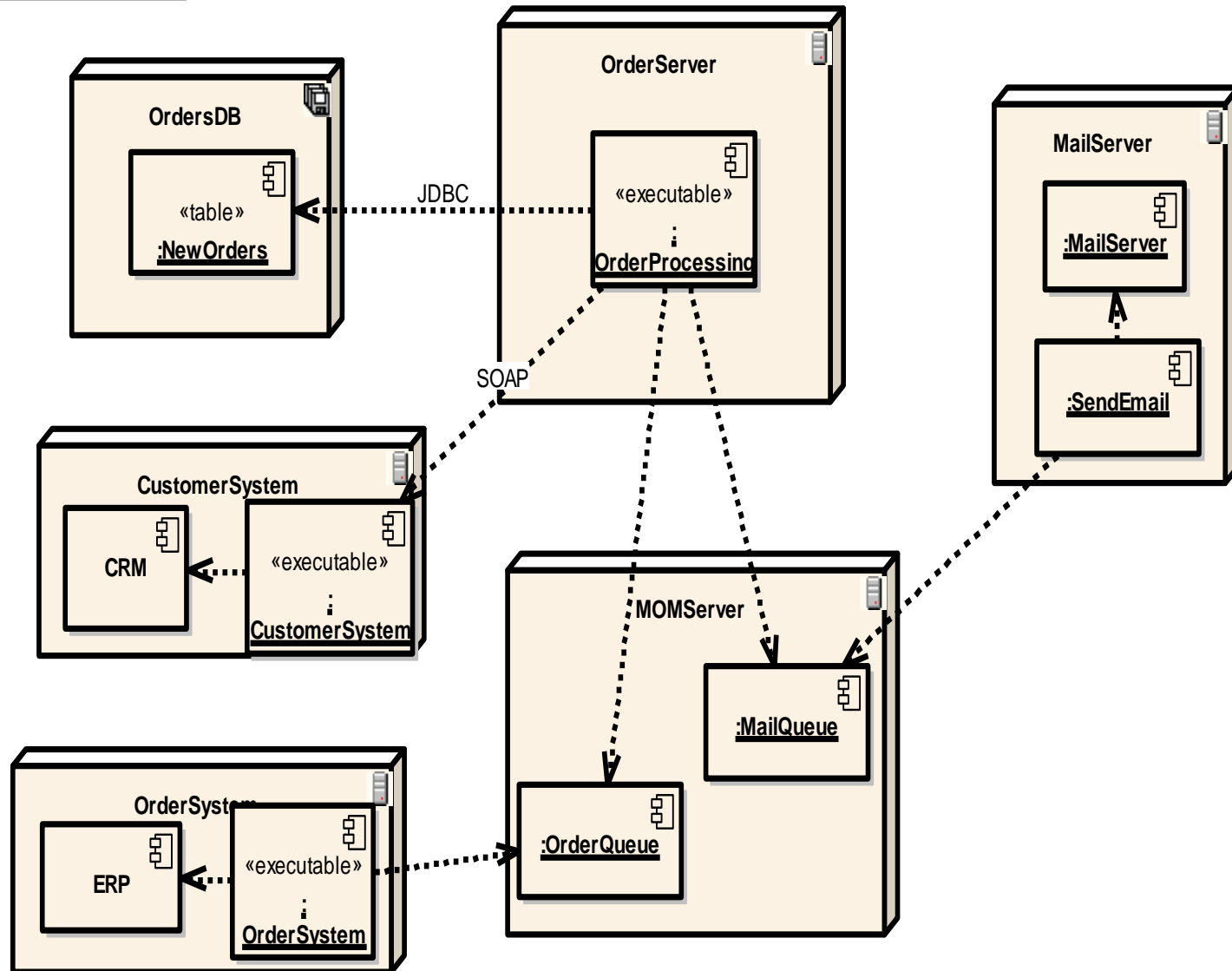
What Is a Deployment Diagram?

- Deployment diagram show the physical configurations of software and hardware components:
 - **Processing nodes:** computational resource with processing capability
 - **Communication links** between these nodes: physical medium and communication protocols
 - **Deployed components** that reside on each node



Example Deployment Diagram

dd Deployment View



Deployment diagram for Real Estate Application

