Department Of Computer Science

Lecture 05

Software Architectural designs

Objectives

- **What is Software Architecture?**
- **Role of the Software Architecture**
  - Assessment and evaluation
  - Configuration management
  - Dynamic software architectures
- **Architecture Design Process**
  - Functionality-Based Design
  - Assessment of the Quality Attributes
  - Architecture Transformation
- **What is Architectural Styles?**
  - Basic Properties of Styles
  - Benefits of Styles

Objectives

- **Architectural Styles** (Name, Figure, Advantages, Disadvantages, Example, when to use?)
  - Pipes and Filters
  - 2-Tiered (Client/Server)
  - N-Tiered
  - Layered
  - Blackboard
  - Model View Controller (MVC)
  - Repository Architecture
  - Object Oriented Style
  - Heterogeneous Styles
- **Difference Architectural Models and Styles**
- **Difference Architectural Style vs Architectural Pattern with example**
- **Analysis Vs Design**

What is Software Requirement?

- It may **range from a high-level abstract statement of a service** or of a system constraint to a detailed mathematical functional specification.
- **This is inevitable as requirements may serve a dual function**
  - **May be the basis for a bid for a contract -** therefore must be open to interpretation;
  - **May be the basis for the contract itself -** therefore must be defined in detail;
  - **Both these statements may be called requirements.**

What is Software Architecture?

- "**The software architecture of a program** or computing system is the structure or structures of the system, which comprise software components [and connectors], the externally visible properties of those components [and connectors] and the relationships among them."
- **Software architecture defines** the components that make up the system, but on the other hand, the properties and functionality of the components define the overall system functionality and behavior.

Role of the Software Architecture

❖The main uses of a software architecture are:

1. Assessment and evaluation
2. Configuration management
3. Dynamic software architectures

1- Assessment and Evaluation

- Stakeholder-Based Assessment is concerned with determining whether the trade-offs between requirements in the software architecture match the actual stakeholder priorities of these requirements.

- Expert-based Assessment a team of experienced architects and designers

- Quality-Attribute Oriented Assessment aims at providing a quantitative prediction of one quality attribute (e.g. maintainability, performance, reliability or security)

  - QDR (Quality-Driven Re-engineering) Framework

2- Configuration Management

- The software architecture is frequently used as a means to manage the configuration of the product.

3- Dynamic Software Architectures

- The software architecture should reorganize itself in response to the dynamic change of the systems quality requirements.

- Maintained even during run-time.

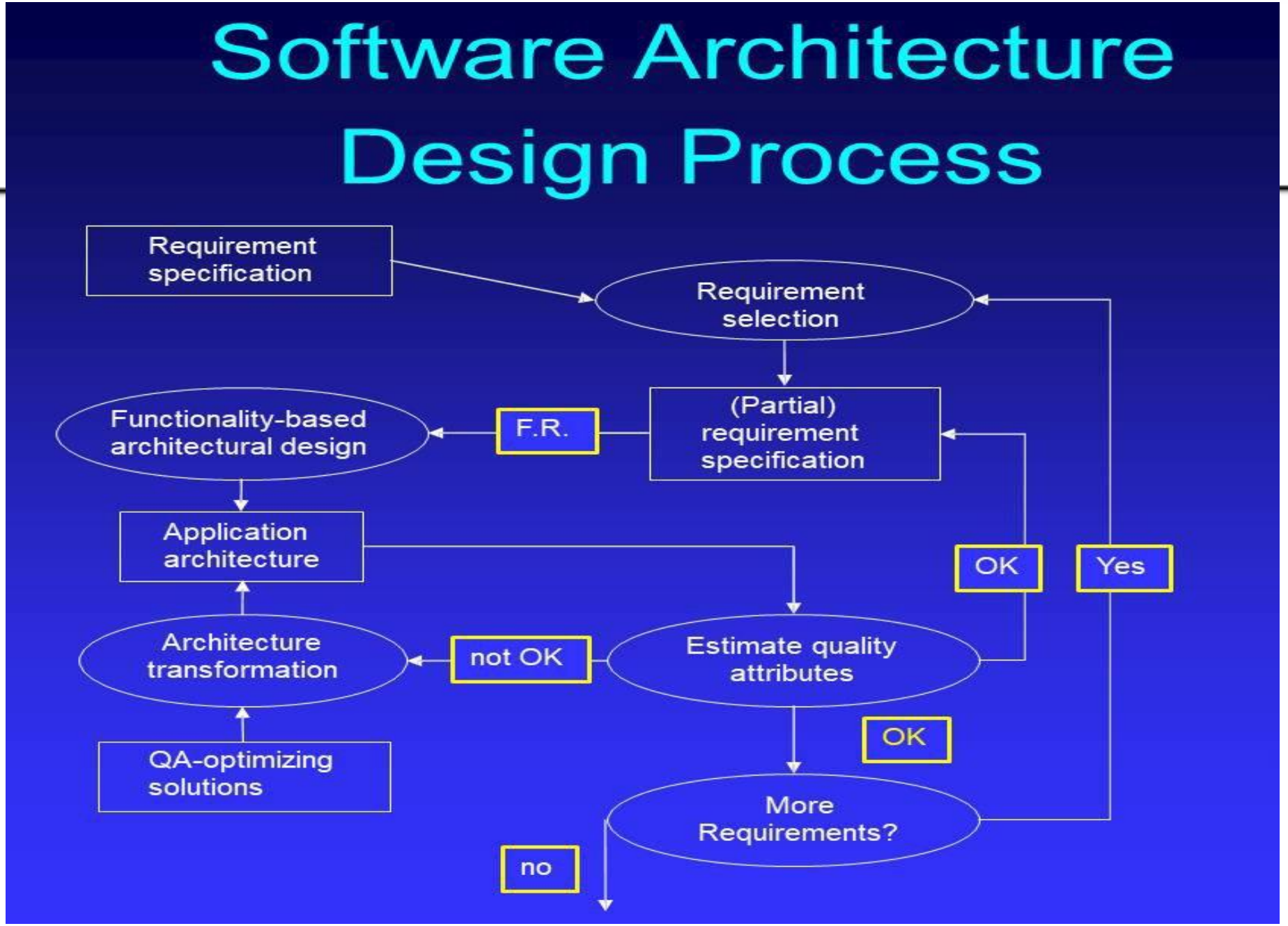
- **Requirements to Architecture**

Architecture Design Process

- Can be seen as a function that:
  - Takes a requirement specification as input.
  - Generates an architectural design as output.
  - Is not an automated process, necessitating great effort and creativity from the involved software architects.

- Is comprised of three steps:
  - Functionality-Based Design
  - Assessment of the Quality Attributes
  - Architecture Transformation

Software Architecture Design Process



1- Functionality-Based Design

- **The design process starts** with functionality-based design and consists of **four steps**:
  - Defining the boundaries and context of the system.
  - Identification of archetypes.
  - Decomposition of the system into its main components.
  - The first validation of the architecture by describing a
    - number of system instances.

2- Assessment of the Quality Attributes

❖ The second phase is the assessment of the quality attributes in which:

➢ **Each quality attribute** is given *an estimate:*

- scenario-based assessment, simulation, static analysis, metrics-based models and expert-based assessment.

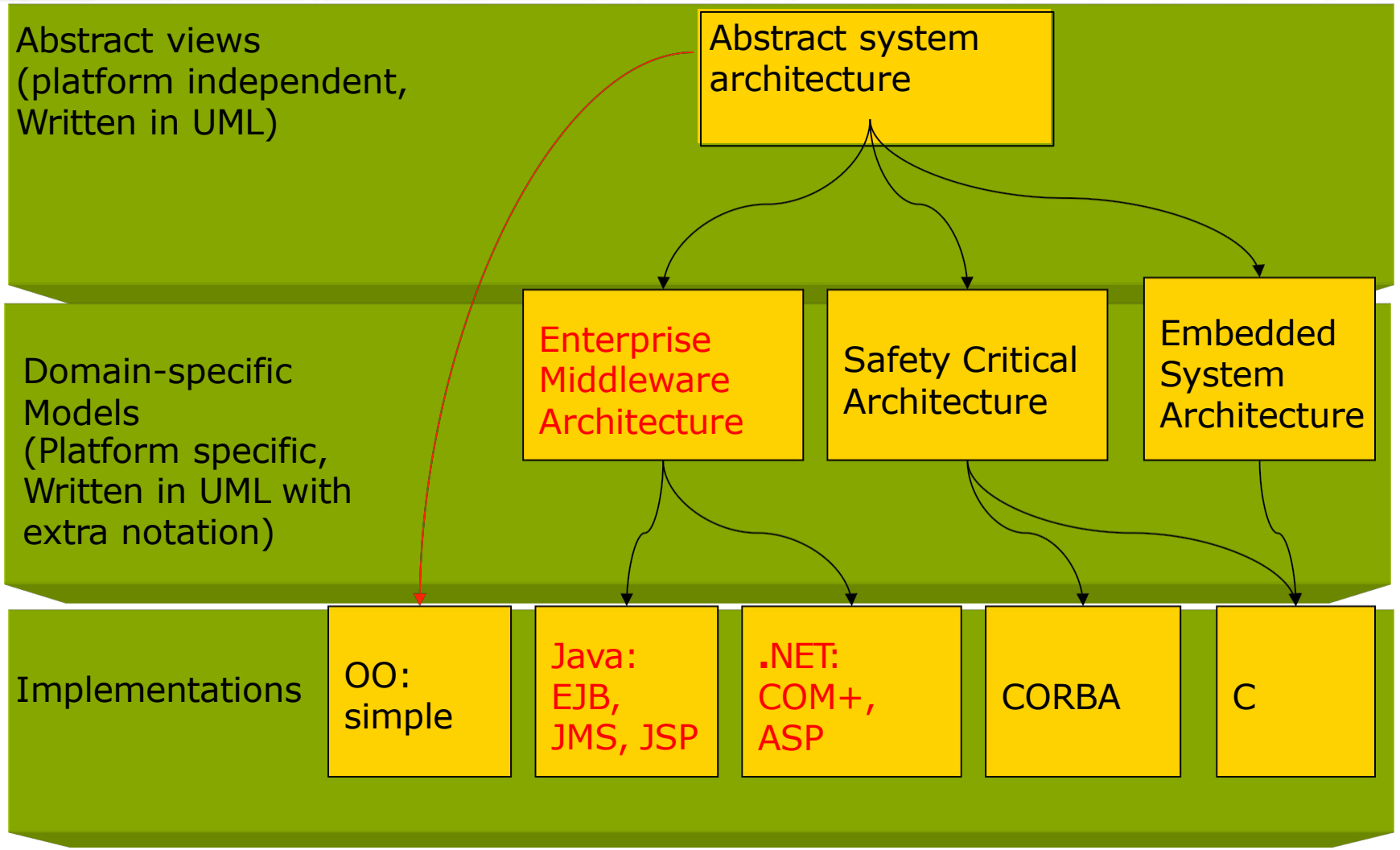
▪ **If all estimated quality attributes** are as good or better than required, the architectural design process is finished

▪ **If not the third phase of software architecture design** is entered: architecture transformation

3- Architecture Transformation

- **Is concerned with selecting design solutions** to improve the quality attributes while preserving the domain functionality:
  - **The design is again evaluated** and the same process is repeated if necessary.
  - **The transformations** (i.e. quality attribute optimizing solutions) generally improve one or some quality attributes while they affect others negatively.

From Abstractions to Implementations



- **Architecture Details**

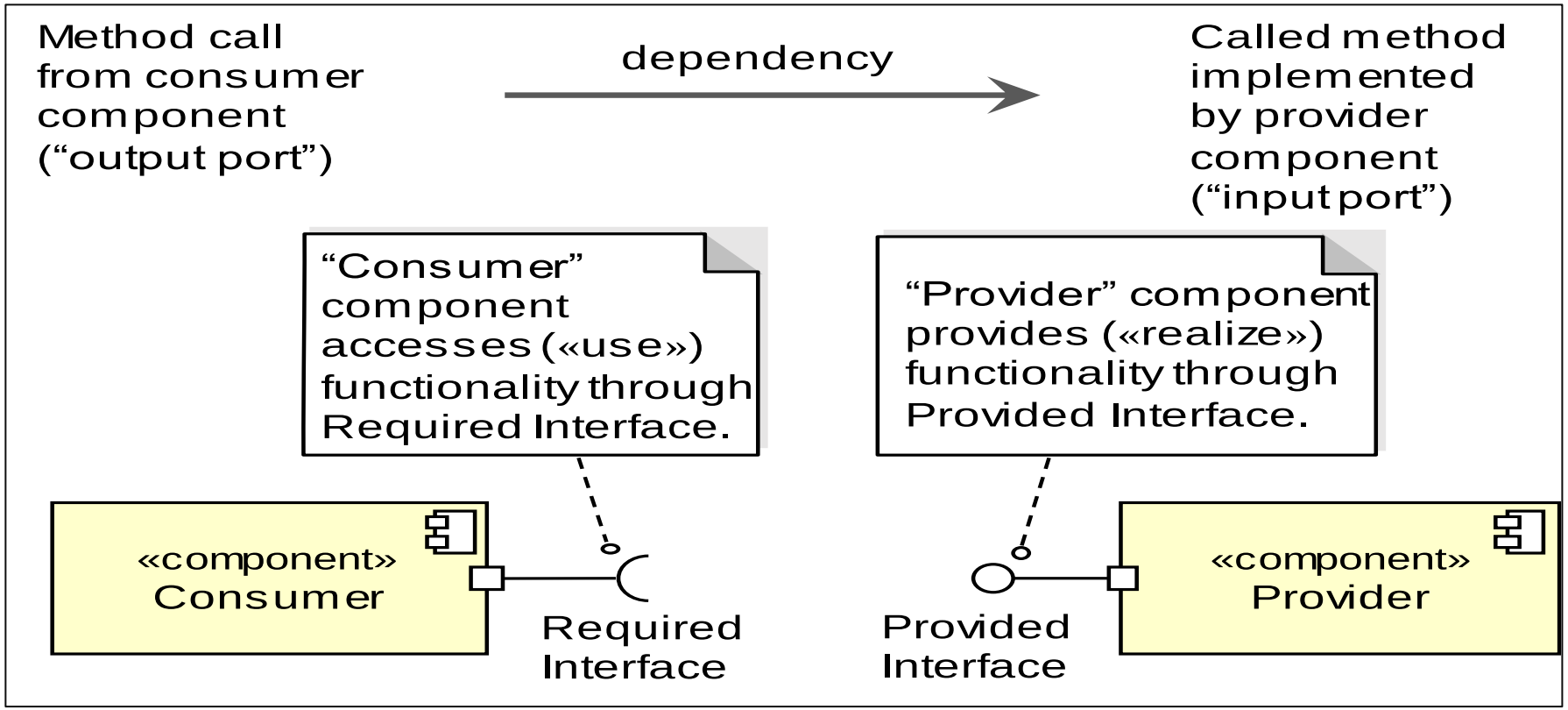
Components and Connectors

Components

- A *component* is a building block that is …
  - A unit of **computation** or a **data store**, with an **interface** specifying the services it provides
  - A unit of **deployment**
  - A unit of **reuse**

UML Notation for Software Components

A component has its behavior defined in terms of **provided interfaces** and **required interfaces** (potentially exposed via ports)



The Difference Between Components and Objects

- **Lifecycle**
  - Objects are created and destroyed constantly
  - Components are created and destroyed infrequently
- **Purpose of use**
  - Graphics toolkit (component) vs. graphics widget (object)
  - Data store (component) vs. data structure (object)
- **Type system**
  - Objects are instances of a class, with classes arranged in hierarchies according to inheritance relationships
  - Components may have their own type system (may be trivial), often very few components of the same type
- **Size**
  - Objects tend to be small
  - Components can be small (one object) or large (a library of objects or a complete application)

Connectors

- **A *connector* is a building block that enables interaction among components**
  - Shared variables
  - Procedure calls (local or remote)
  - Messages and message buses
  - Events
  - Pipes
  - Client/server middleware
- **Connectors may be *implicit* or *explicit***
  - **Implicit:** procedure calls
  - **Explicit:** First-class message buses

The Difference Between Components and Connectors

- **Task Performed**
  - Components focus on computational tasks
  - Connectors focus on communication tasks
- **Application Semantics**
  - Components generally implement most of the application semantics
  - Connectors do not (they may change the form of the message, but do not generally change its meaning)
- **"Awareness"**
  - Components (should be) unaware of who is using them and for what purpose
  - Connectors are more aware of components connected to them so they can better facilitate communication

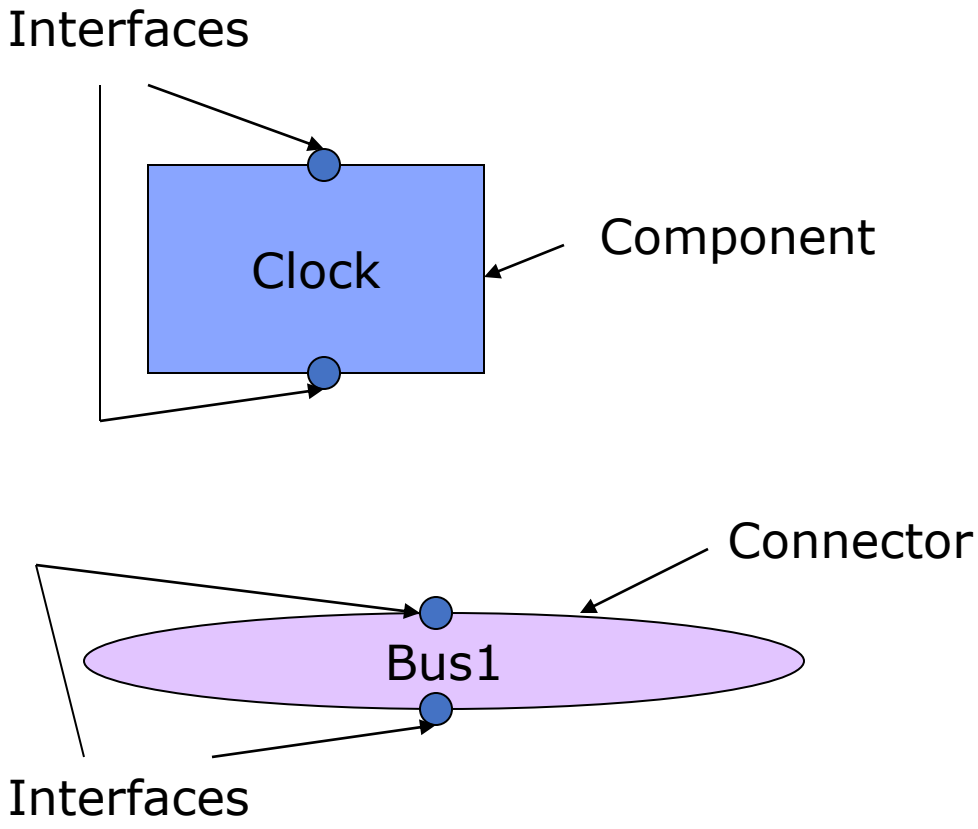
Interfaces

- **An *interface* is the external** "connection point" on a component or connector that describes how other components/connectors interact with it
- **Provided *and* required interfaces are important**
- Spectrum of interface specification
  - Loosely specified (events go in, events go out)
  - API style (list of functions)
  - Very highly specified (event protocols across the interface in CSP)
- Interfaces are *the* key to component interoperability (or lack thereof)

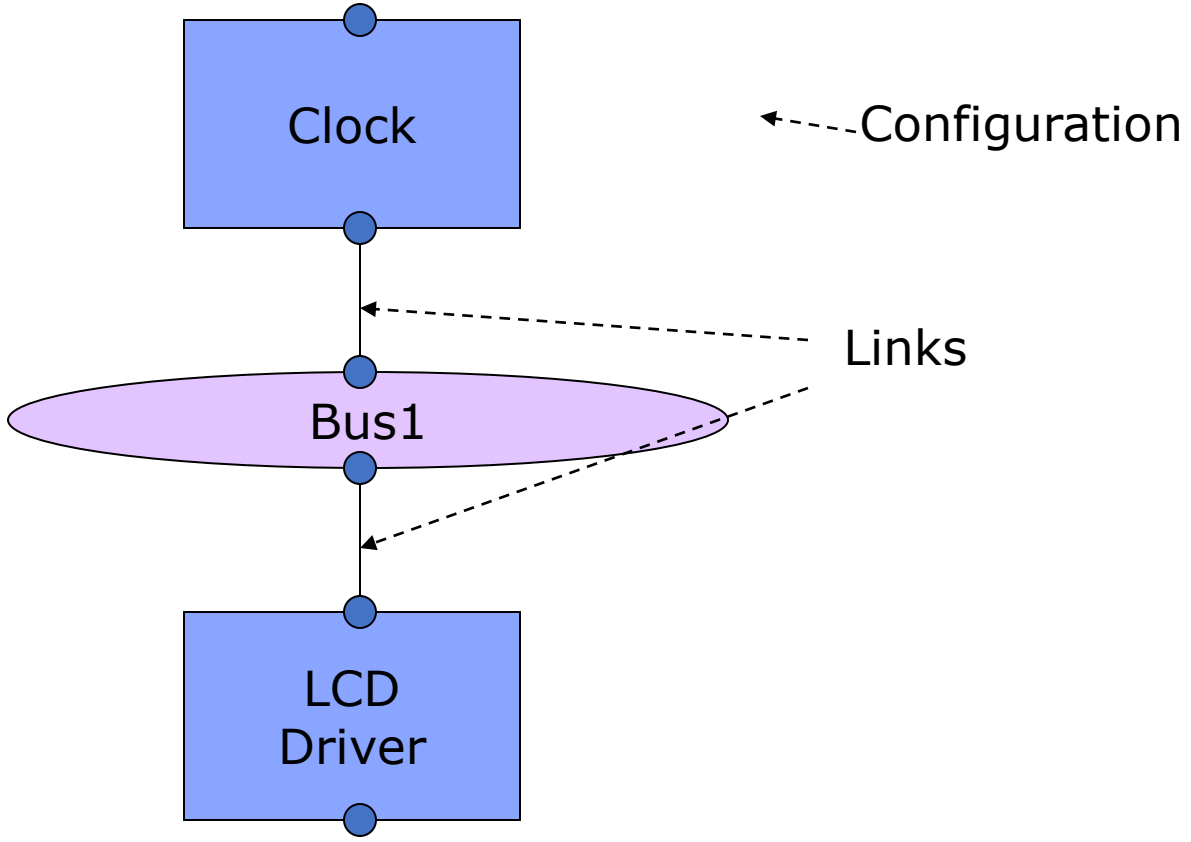
Configurations

- **A *configuration* is …**
  - The overall structure of a software architecture
  - The topological arrangement of components and connectors
    - Implies the existence of links among components/connectors
  - A framework for checking for compatibility between interfaces, communication protocols, semantics, …
- "If links had semantics, they'd be connectors."
- Usually constructed according to an *architectural style*

Graphically…



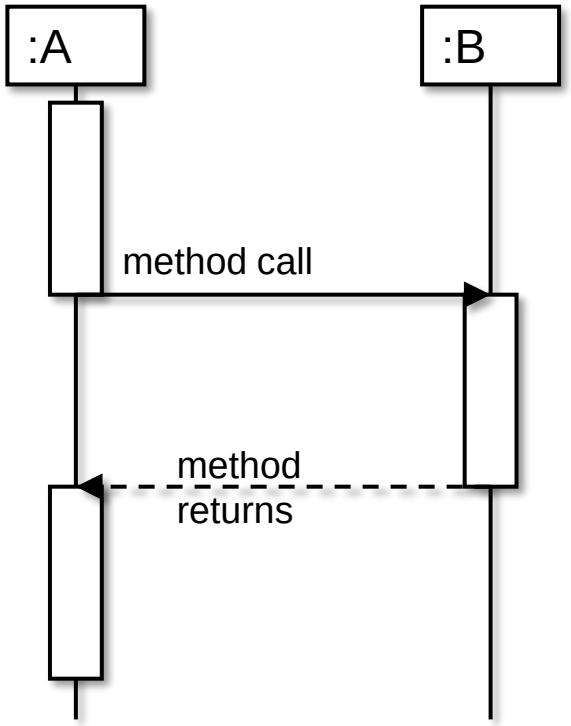
Graphically (cont).



UML2 Connectors in Detail

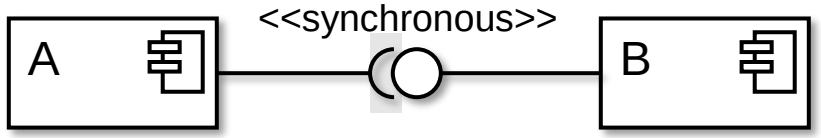
- An architectural abstraction of usage between components
  - **Many possible implementations**
  - Our simple Java implementation: a reference to one component class B within another A
    - Object implementing A communicates with the object for B by calling methods on B's interface
    - **Synchronous communication:** A must wait for method calls to finish before continuing

Synchronous Communication

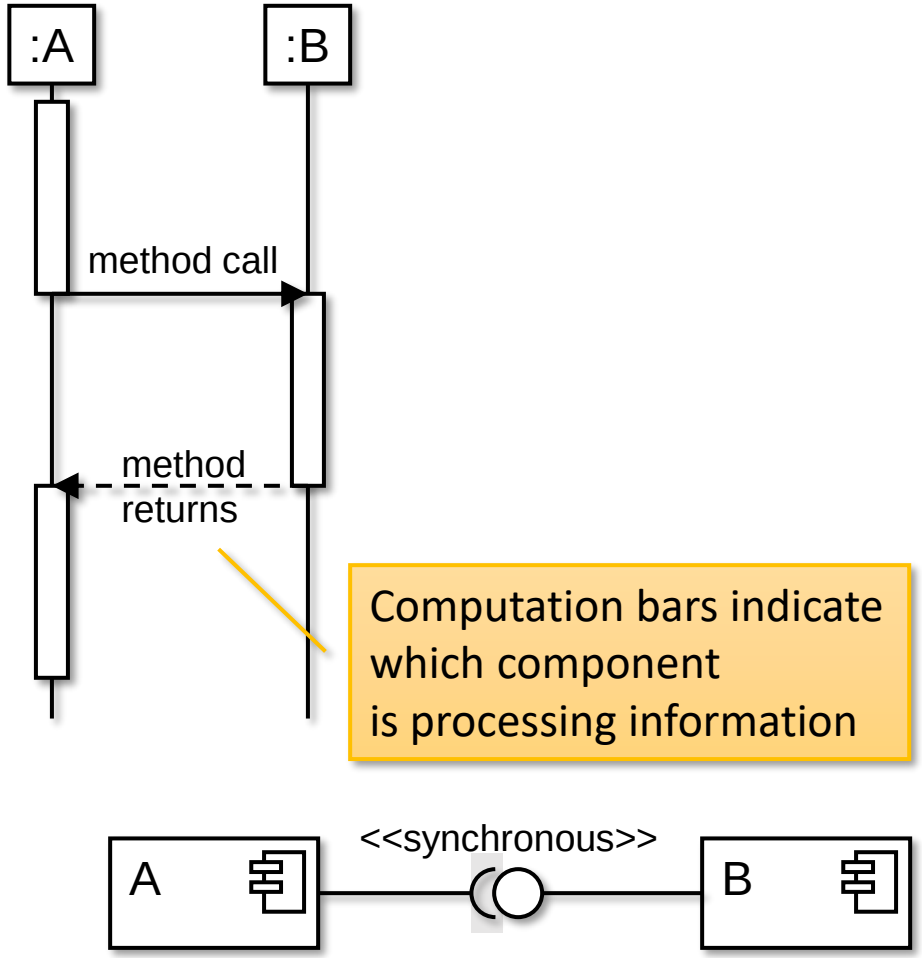


UML Sequence Diagram: shows object lifelines & messages

- UML2 Component Diagrams cannot show this directly
  - All connectors are considered equal by default
  - Can define stereotypes to adjust connector semantics
  - We will use <<synchronous>> stereotype

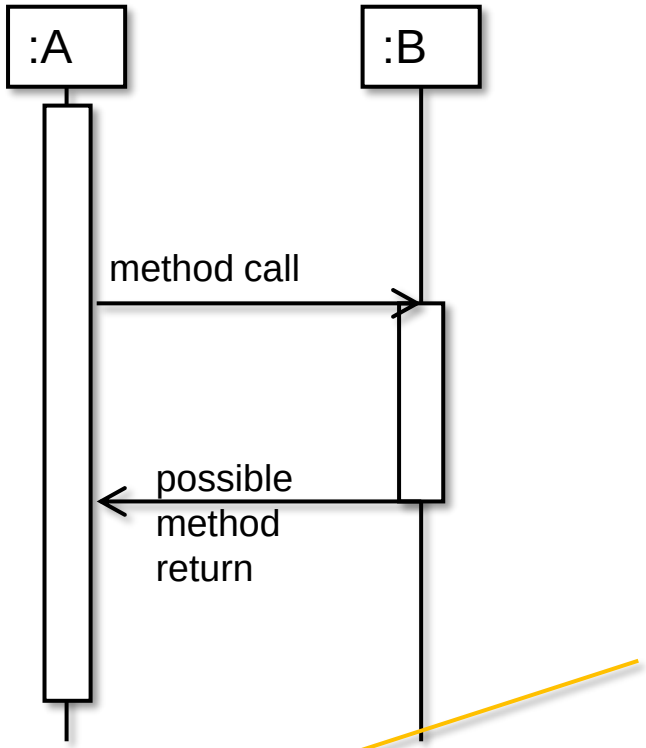


Synchronous Communication



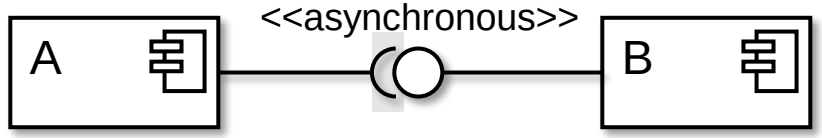
- UML2 Component Diagrams cannot show this directly
  - All connectors are considered equal by default
  - Can define stereotypes to adjust connector semantics
  - We will use <<synchronous>> stereotype

Asynchronous Communication

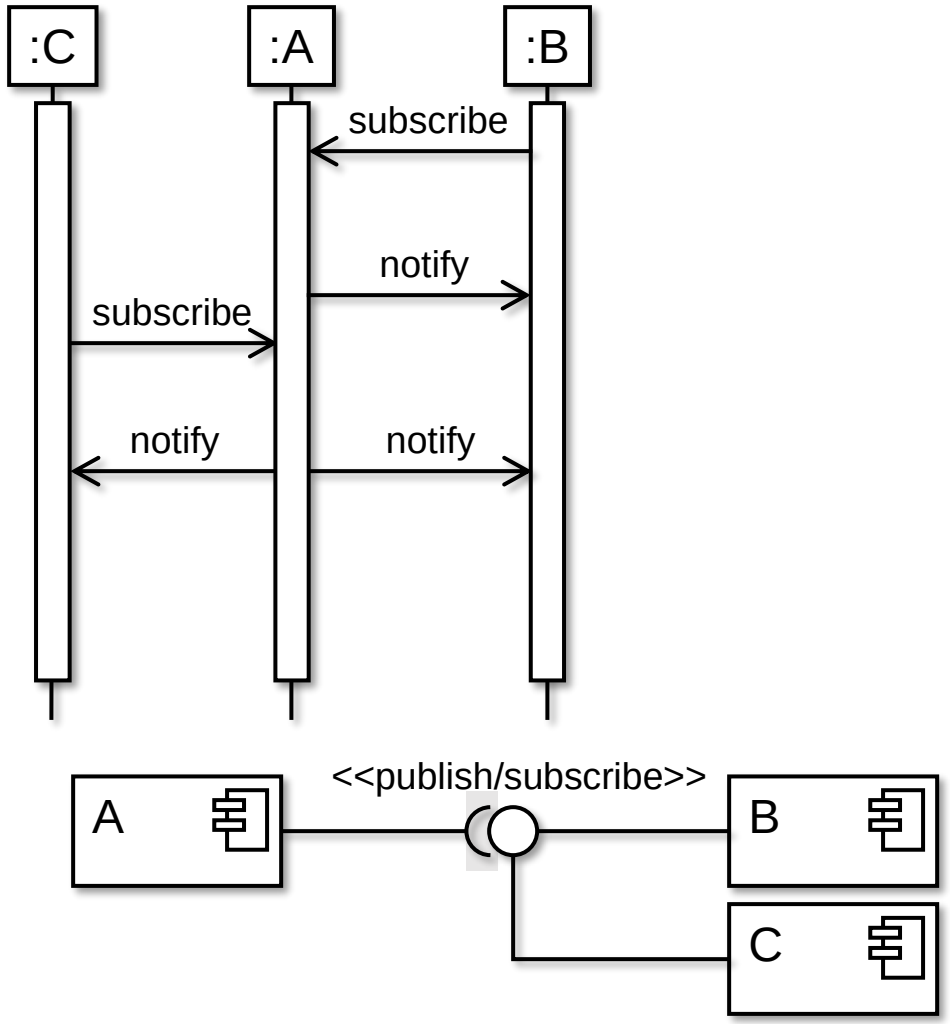


- A can continue computation after sending call to B
  - No need to wait
  - Good for decoupling components
  - Good for shielding against communication delays
- Will use <<asynchronous>> stereotype

Notice the different arrow heads
This is how UML2 indicates asynchronous calls.



Publish/Subscribe Connector



- Components subscribe to events from publisher
  - Publisher does not know subscribers
  - Sends event to all subscribers
- We will use `<<publish/subscribe>>` stereotype

What does architecture buy you?

- On its face, nothing!
- How can we use architecture to improve the qualities of our software systems?
- **Answer:** Architectural Styles

- **Architecture Design**

Architecture Design

Architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.

The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components

Block Diagram - Architecture Design

System architectures are often modeled using simple block diagrams.

Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows.

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand.

Architecture Design

The architecture of a software system may be based on a particular architectural pattern or style.

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

An architectural style is a set of principles. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems.

- **Architecture Style**

Architectural Styles: Definitions

- "Architectural styles are *recurring patterns*".
- "Established, *shared understanding of common design* forms is a mark of a mature engineering field".
- "Architectural style is an abstraction of *recurring composition and interaction characteristics* of a set of architectures".
- "Styles are *key design idioms* that enable exploitation of suitable structural and evolution patterns and facilitate component, connector, and process reuse".

Architectural Styles

- **An *architectural style* is …**
  - A set of *constraints* you put on your development to elicit desirable properties from your software architecture.
  - These constraints may be:
    - Topological
    - Behavioral
    - Communication-oriented
    - etc. etc.
  - Working within an architectural style makes development *harder*
  - BUT architectural styles help you get beneficial system properties that would be *really* hard to get otherwise

Basic Properties of Styles

- A vocabulary of design elements
  - Component and Connector type:
  - (pipes, filters, clients, servers)
- A set of configuration rules
  - Constraints that determine allowed compositions of
    - elements

    (a component may be connected to at most two other components)
- A semantic interpretation
  - Compositions of design elements have well-defined meanings
  - Explanation of what components are supposed to do in general
  - Explanation of how connections are meant to work

Architecture Styles – Constituent Parts

1. **Components**
   - Processing elements that "do the work"
2. **Connectors**
   - Enable communication among components
     - Broadcast Bus, Middleware-enabled, implicit (events), explicit (procedure calls, ORBs, explicit communications bus) …
3. **Interfaces**
   - Connection points on components and connectors
     - define where data may flow in and out of the components/connectors
4. **Configurations**
   - Arrangements of components and connectors that form an architecture

Benefits of Styles

- Design reuse
  - well-understood solutions applied to new problems
- Code reuse
  - shared implementations of invariant aspects of a style
- Understandability of system organization
  - a phrase such as “client-server” conveys a lot of information
- Interoperability
  - supported by style standardization
  - components can be independently developed with a particular style in mind
- Visualizations
  - style-specific depictions matching engineers’ mental models

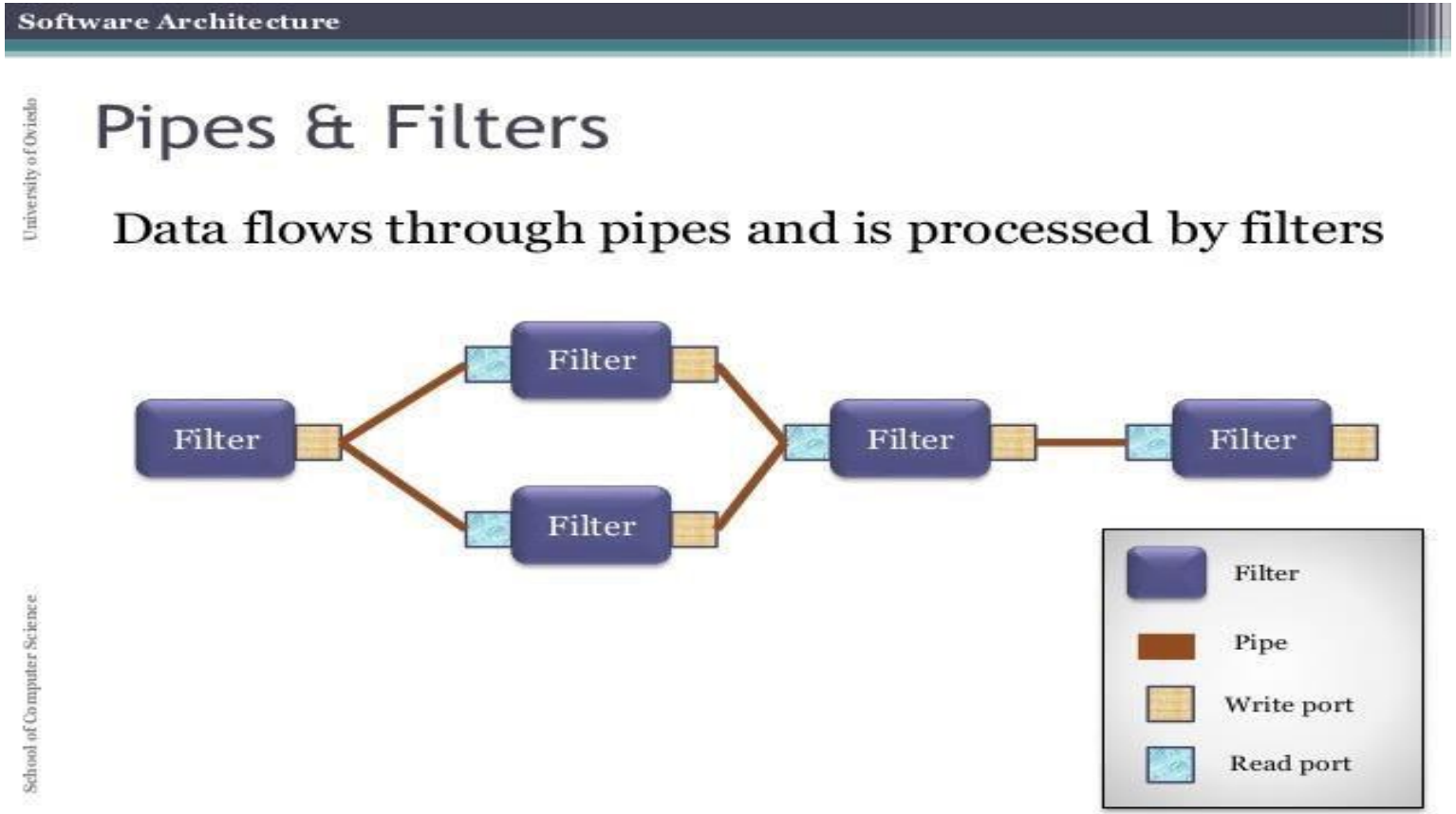
Some Architectural Styles

- Learning the following domain/platform-independent styles:
  - Pipes and Filters
  - 2-Tiered (Client/Server)
  - 3- Tier
  - N-Tiered
  - Layered
  - Blackboard
  - Model View Controller (MVC)
  - Object Oriented Style
  - Heterogeneous Styles
- Pipes and filters were easy
- The rest require a deeper understanding of connections

Style 1: Pipes and Filters

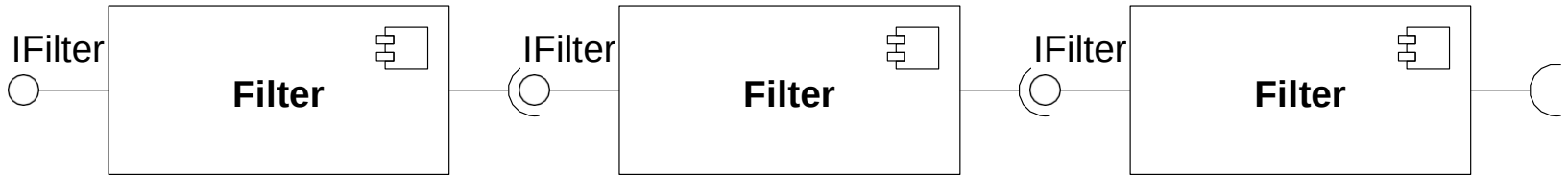
➢The pipe-and-filter architectural style has only one component and one connector type.

➢The component type is the filter, and the connector type is the pipe.



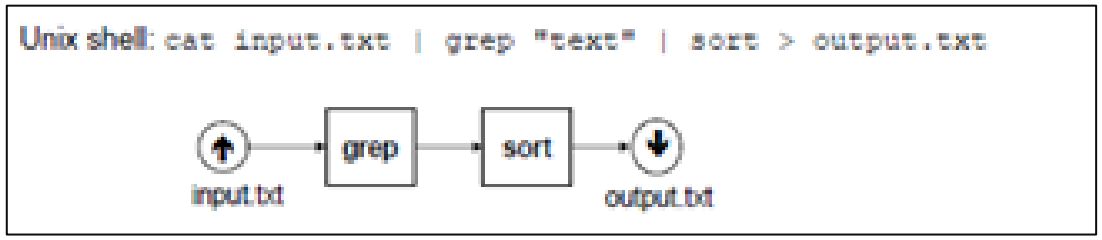
Style 1: Pipes and Filters

- **Components are filters**
  - transform input messages into output messages
  - All components of a pipe and filter architecture use the **same interface**
  - Input through provided interface
  - Output through required interface
  - Usually – one provided (input) interface and one required (output) interface
- **Connectors are pipes**
  - simple connections between filters' provided and required interfaces
- **Style invariants**
  - filters are independent
  - a filter has no knowledge of up- and down-stream filters
- **Each filter exposes a very simple interface:** it receives messages on the **inbound pipe**, processes the message, and publishes the results to the **outbound pipe**.



Style 1: Pipes and Filters

- Advantages
  - Easy understanding of the system's behavior as the composition of filters
  - Filter addition, replacement, and reuse
- Disadvantages
  - Batch organization of processing
  - Interactive applications
  - Lowest common denominator on data transmission
- Commonly seen in
  - UNIX shells
  - Compilers
  - Signal Processing



- But also useful style for *Enterprise Systems*
- When Used: Whenever different data sets need to be manipulated in different ways, you should consider using the pipe and filter architecture.

Style 1: PF Example

- In many enterprise integration scenarios, a single event triggers a sequence of processing steps, each performing a specific function.
- Consider *an order processing system* -- new orders arrive from a client component in the form of a message.
  - Requirement 1: the message is encrypted to prevent eavesdroppers from spying on a customer's order.
  - Requirement 2: the messages contain authentication information in the form of a digital certificate to ensure that orders are placed only by trusted customers.

Style 1: PF Example

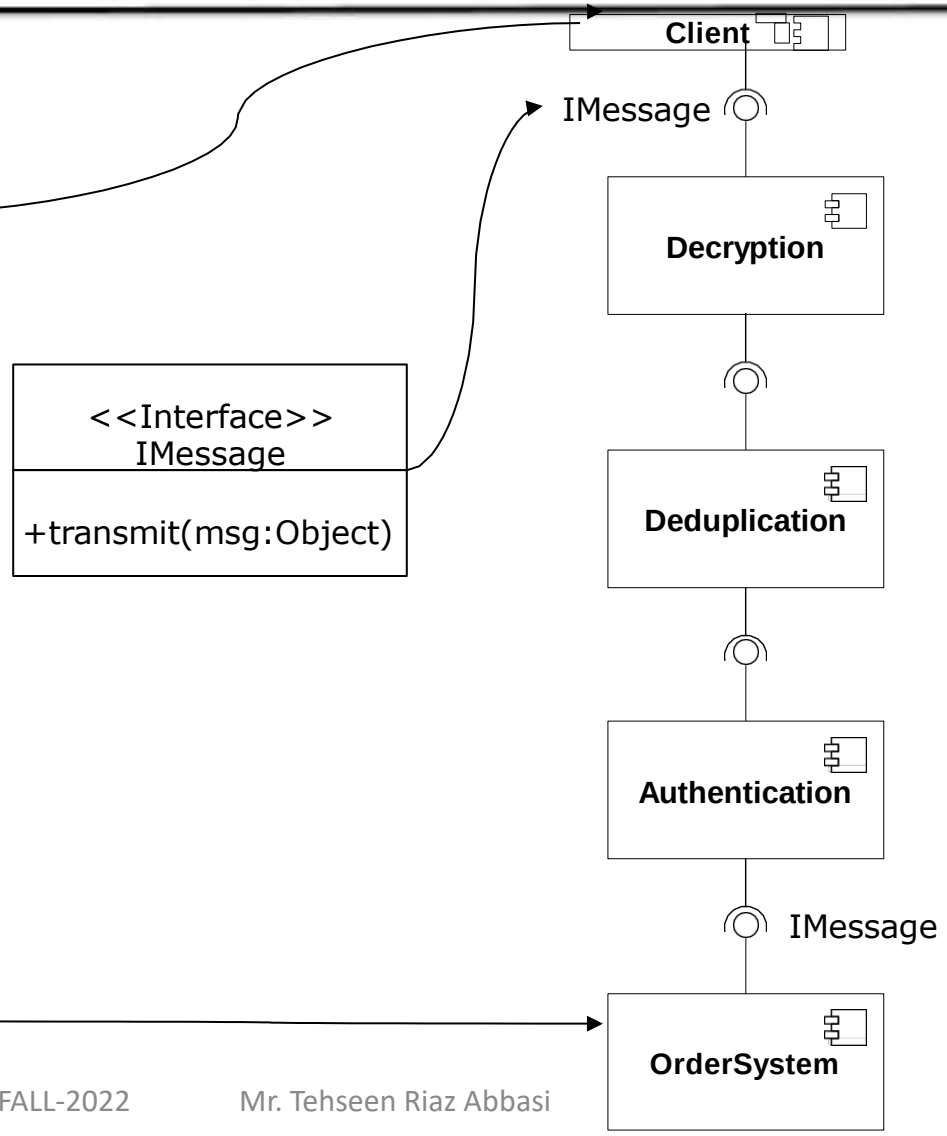
- Consider an order processing system -- new orders arrive from a client component in the form of a message.
  - Requirement 3: duplicate messages could be sent from external parties (remember all the warnings on the popular shopping sites to click the 'Order Now' button only once?).
    - To avoid duplicate shipments and unhappy customers, we need to eliminate duplicate messages before subsequent order processing steps are initiated.
- To meet these requirements, we need *to transform a stream of possibly duplicated, encrypted messages containing extra authentication data into a stream of unique, simple plain-text order messages without the extraneous data fields.*

Style 1: PF Example

External client component – produces an order message

Messages are input/ output using the same simple interface – transmit(msg:Object) is the method that transmits the message msg, typed as an Object

Order processing system



Style 1: PF Example

- Pipes and filters lets us perform complex processing on a message while *maintaining*

  flexibility

- Because all components use the *same external*

  composed into different solutions by connecting the components to different pipes

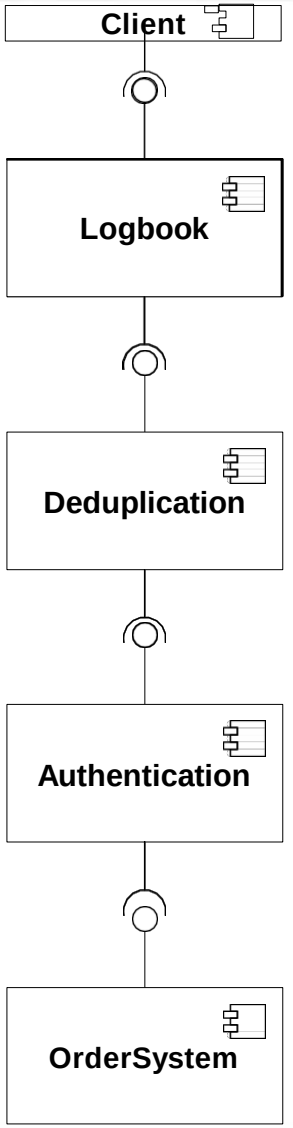
- We can *add* ***new filters***, ***omit existing ones*** or ***rearrange them*** into a new sequence -- all without having to change the filters themselves.

Orders are no

encrypted – but we want a log of each message

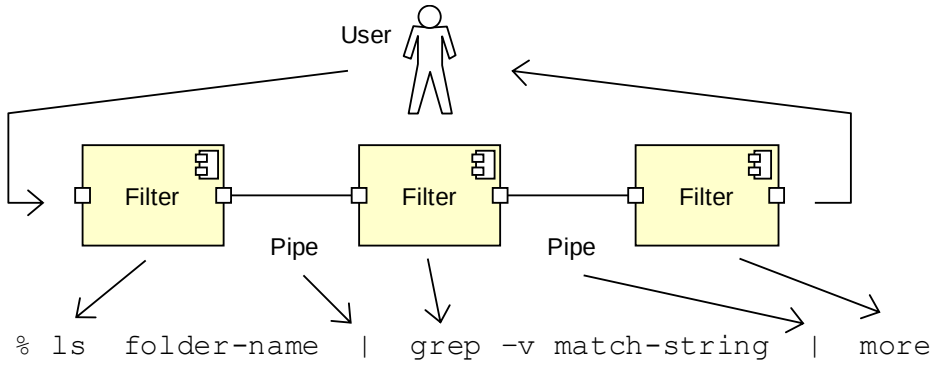
Better to authenticate

deduplicating

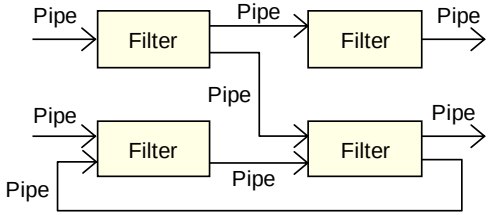


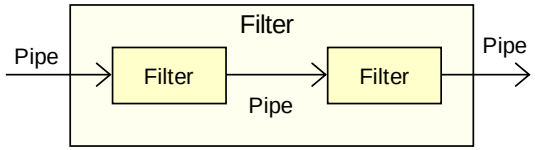
Architecture Style: Pipe-and-Filter (Summary)

- Components: **Filters** transform input into output
- Connectors: **Pipe** data streams
- **Example:** UNIX shell commands



- More complex configurations:





Software Architecture:

One-Tier, Two-Tier, Three Tier, N Tier

A “tier” can also be referred to as a “layer”.

https://www.softwaretestingmaterial.com/software-architecture/

https://dev.to/desi109/architectural-styles-by-examples-387b

https://www.guru99.com/n-tier-architecture-system-concepts-tips.html

Architectural Styles

- Software Architecture consists of One Tier, Two Tier, Three Tier, and N-Tier architectures.
- A “tier” can also be referred to as a “layer”.

- **Three layers are involved in the application namely**
  - 1- Layer
  - 2- Business Layer
  - 3- Data Layer.

Architectural Styles

#1. Presentation Layer

It is also known as the Client layer.

The topmost layer of an application.

This is the layer we see when we use the software.

By using this layer, we can access the web pages.

The main function of this layer is to communicate with the Application layer.

This layer passes the information which is given by the user in terms of keyboard actions, mouse clicks to the Application Layer.

For example, the login page of Gmail where an end-user could see text boxes and buttons to enter user id, password, and to click on sign-in.

In simple words, it is to view the application.

Architectural Styles

#2. Application Layer

It is also known as Business Logic Layer which is also known as the logical layer.

As per the Gmail login page example, once the user clicks on the login button, the Application layer interacts with the Database layer and sends required information to the Presentation layer.

It controls an application's functionality by performing detailed processing.

This layer acts as a mediator between the Presentation and the Database layer. Complete business logic will be written in this layer.

In simple words, it is to perform operations on the application.

Architectural Styles

#3. Data Layer

The data is stored in this layer.

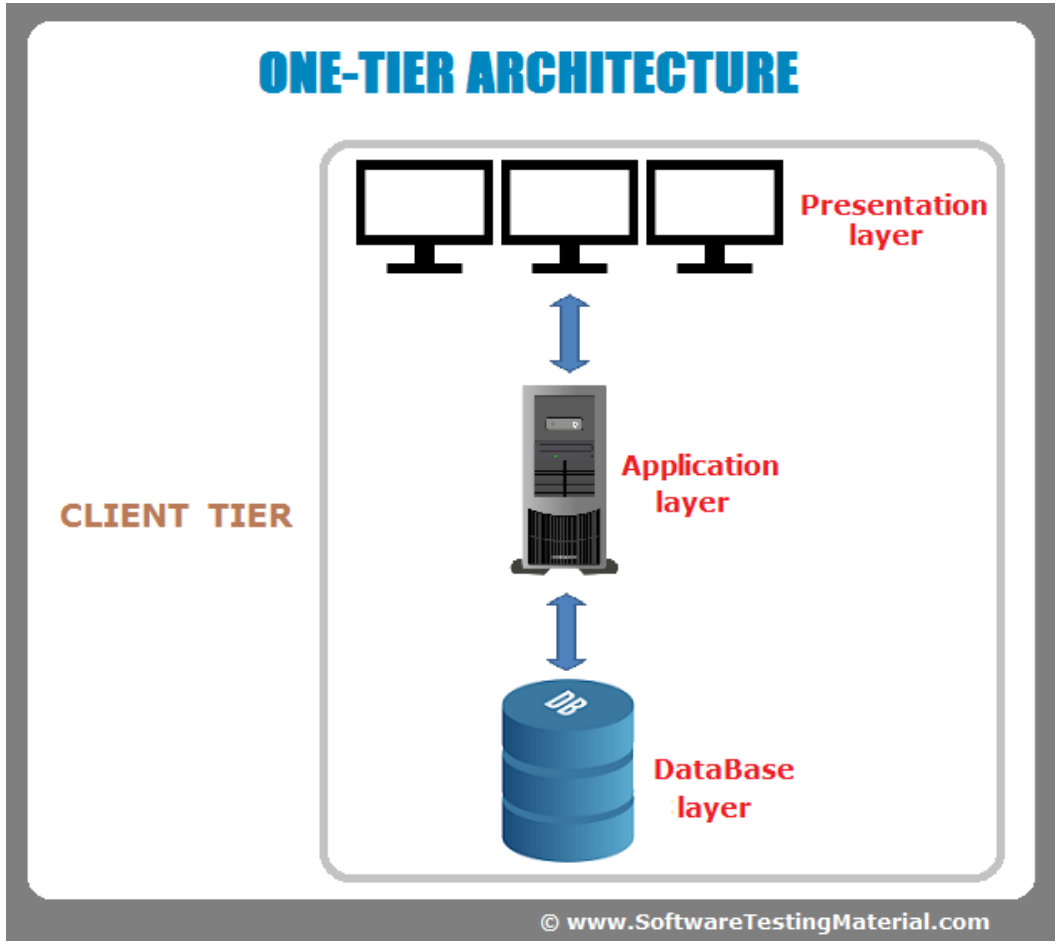
The application layer communicates with the Database layer to retrieve the data.

It contains methods that connect the database and performs required action e.g.: insert, update, delete, etc.

Style 1 - One Tier Architecture:

#1. One Tier Architecture:

- **One Tier application is a** Standalone application
- **One-tier architecture** has all the layers such as Presentation, Business, Data Access layers in a single software package.
- **Applications that handle** all the three tiers such as MP3 player, MS Office come under the one-tier application. The data is stored in the local system or a shared drive..



Style 2 - Two Tier Architecture:

#2. Two-Tier Architecture:

It is also known as a client-server application. Two Tier application is a Client-Server application. The Two-tier architecture is divided into two parts:

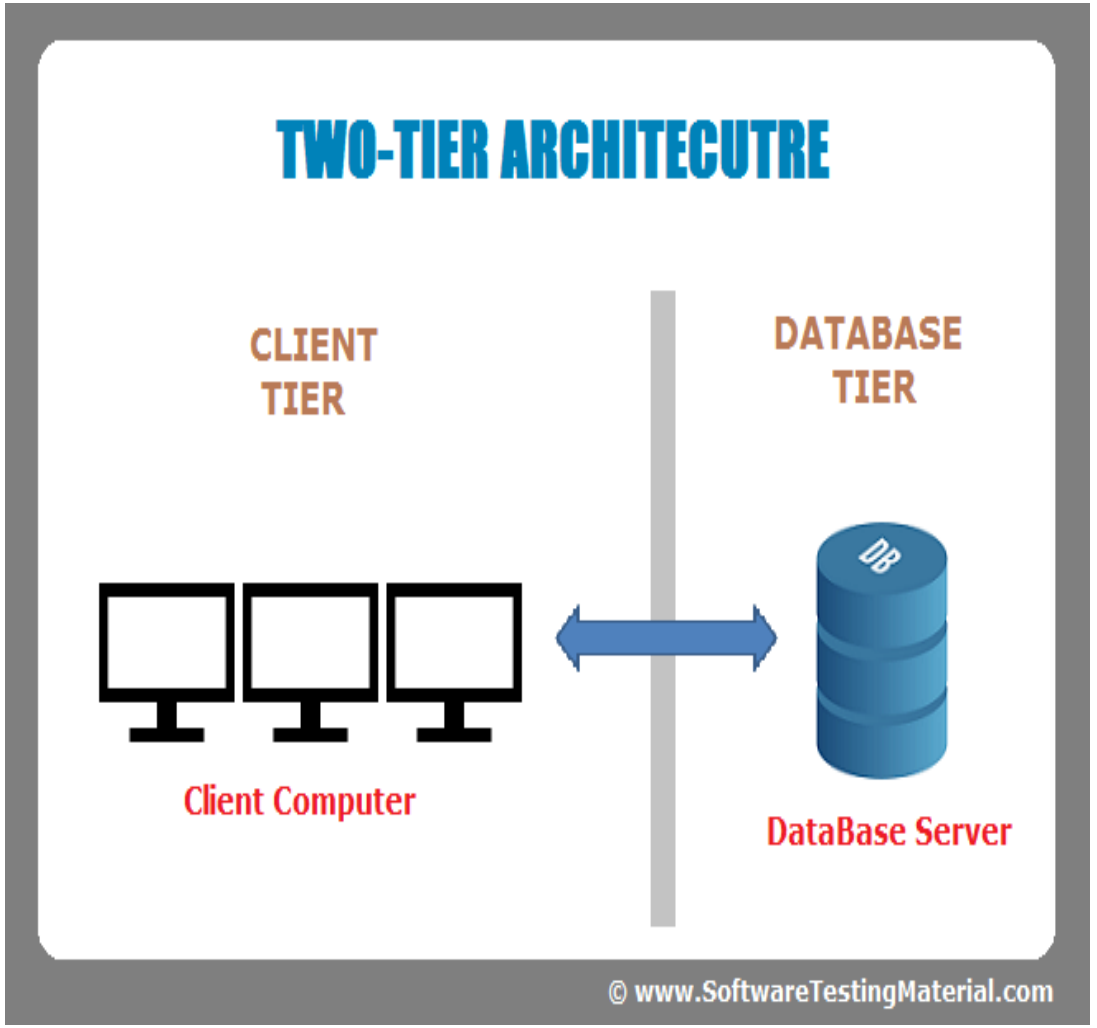
1. Client Application (Client Tier)

2. Database (Data Tier)

The client system handles both Presentation and Application layers and the Server system handles the Database layer.

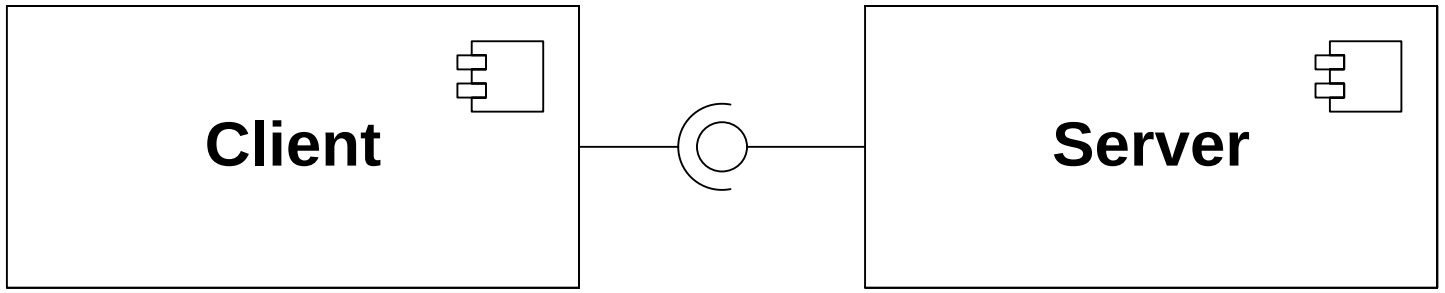
The communication takes place between the Client and the Server.

The client system sends the request to the server system and the Server system processes the request and sends back the data to the Client System



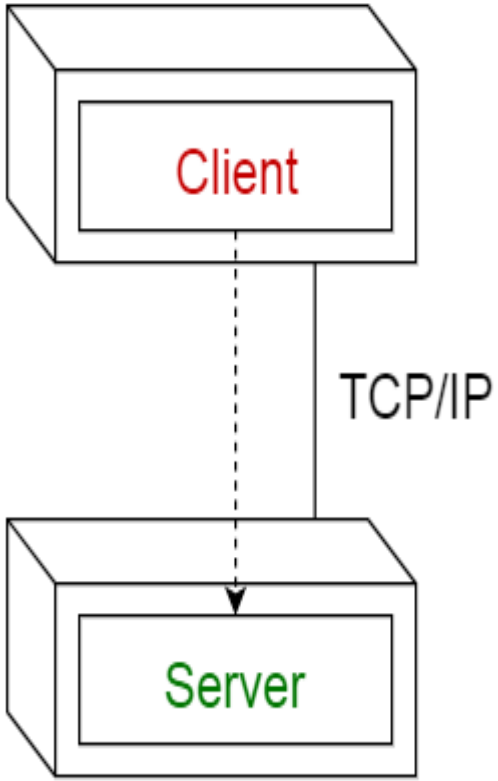
Style 2: 2-Tiered (Client/Server)

- The client-server pattern has two major entities. They are a server and multiple clients.
- One of most common styles in distributed systems
- **Components are clients or servers**
  - Clients request services from servers
  - Clients and servers may be composite
- **Constraints**
  - Server provides services to the client, but does not require services from client
  - Clients require services of the server
  - A larger number of clients than servers

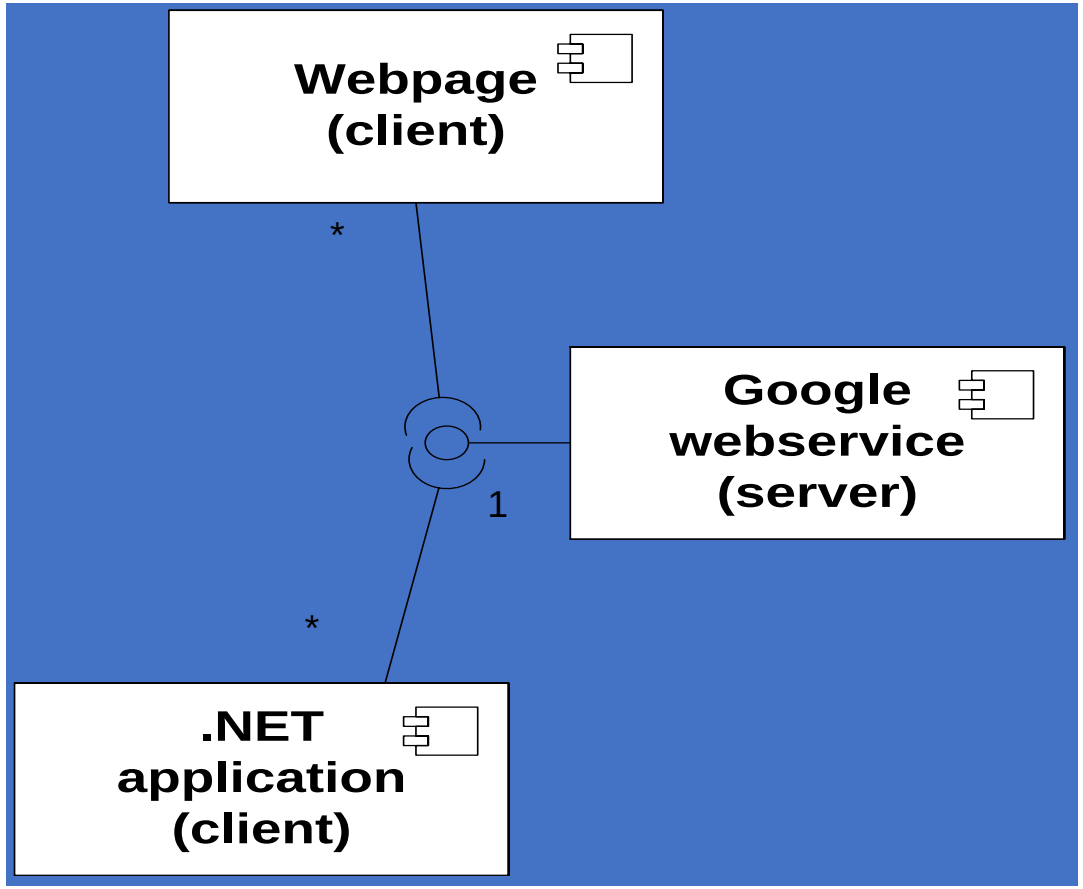


Style 2: Client/Server: Example

- **Google offers search webservice**
- A range of clients use the webservice
  - Webpage servlet clients use it as part of the pages
  - .NET based b-2-c application uses webservice to search for best buys of products
- **Different clients**
  - Very different purposes
  - But the same server functionality
- **Keep web searching and client functionality separate to**
  - Maximize possible uses of the server
  - Enable better maintenance

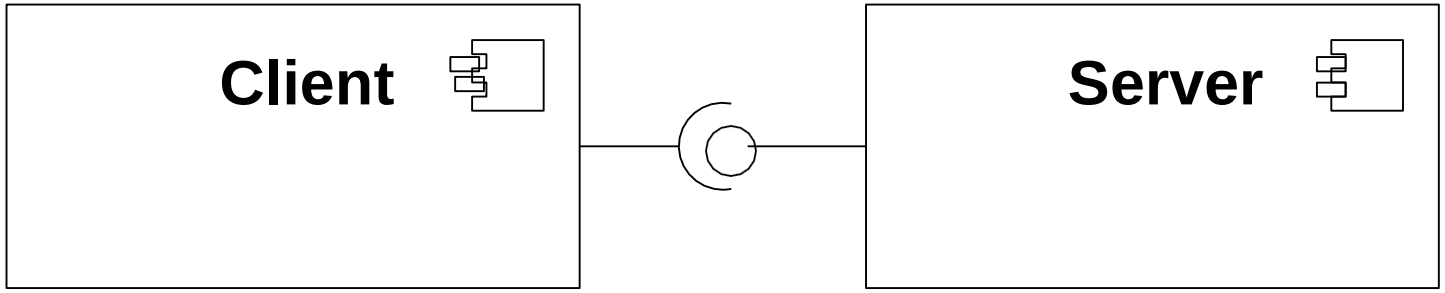


Style 2: Client/Server: Example



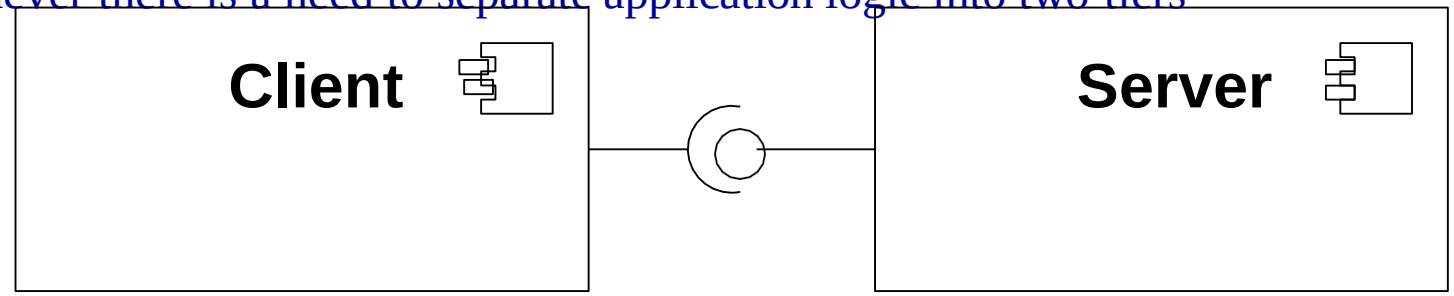
Style 2: 2-Tiered (Client/Server)

- One of the most common styles in distributed systems
- *Components are clients and servers*
  - Application logic is divided between clients and servers – two tiers
  - E.g., client could handle presentation logic, while the server handles business rules and the database
  - Clients and servers may be composite
- *Constraints*
  - *Server provides services to the client, but doesn't require*
  - *Clients require services of the server*
  - A larger number of clients than servers
- *Semantics* (meaning and implementation)
  - Servers do not know the number or identities of clients
  - Clients know server's identity
  - Clients and servers are often distributed
- Useful whenever there is a need to separate application logic into two tiers



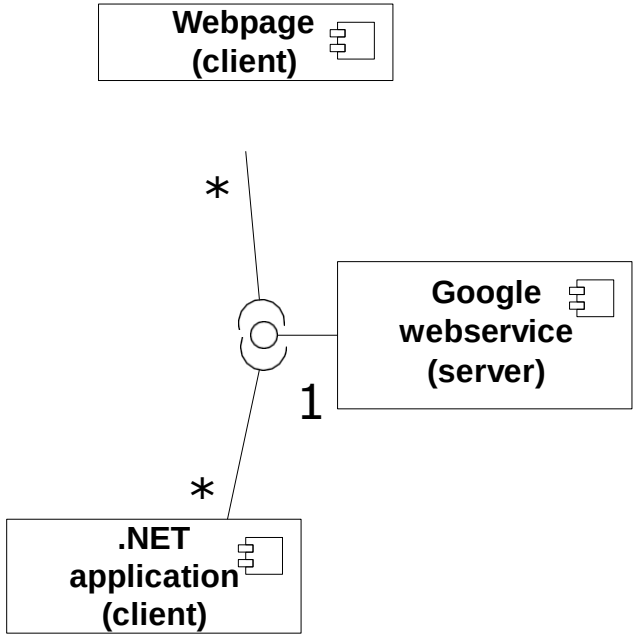
Style 2: 2-Tiered (Client/Server)

- **A "tier" can also be referred to as a "layer".**
- One of the most common styles in distributed systems
- *Components are clients and servers*
  - Application logic is divided between clients and servers – two tiers
  - E.g., client could handle presentation logic, while the server handles business rules and the database
  - Clients and servers may be composite
- *Constraints*
  - *Server provides services to the client, but doesn't require*
  - *Clients require services of the server*
  - A larger number of clients than servers
- *Semantics* (meaning and implementation)
  - Servers do not know the number or identities of clients
  - Clients know server's identity
  - Clients and servers are often distributed
- Useful whenever there is a need to separate application logic into two tiers



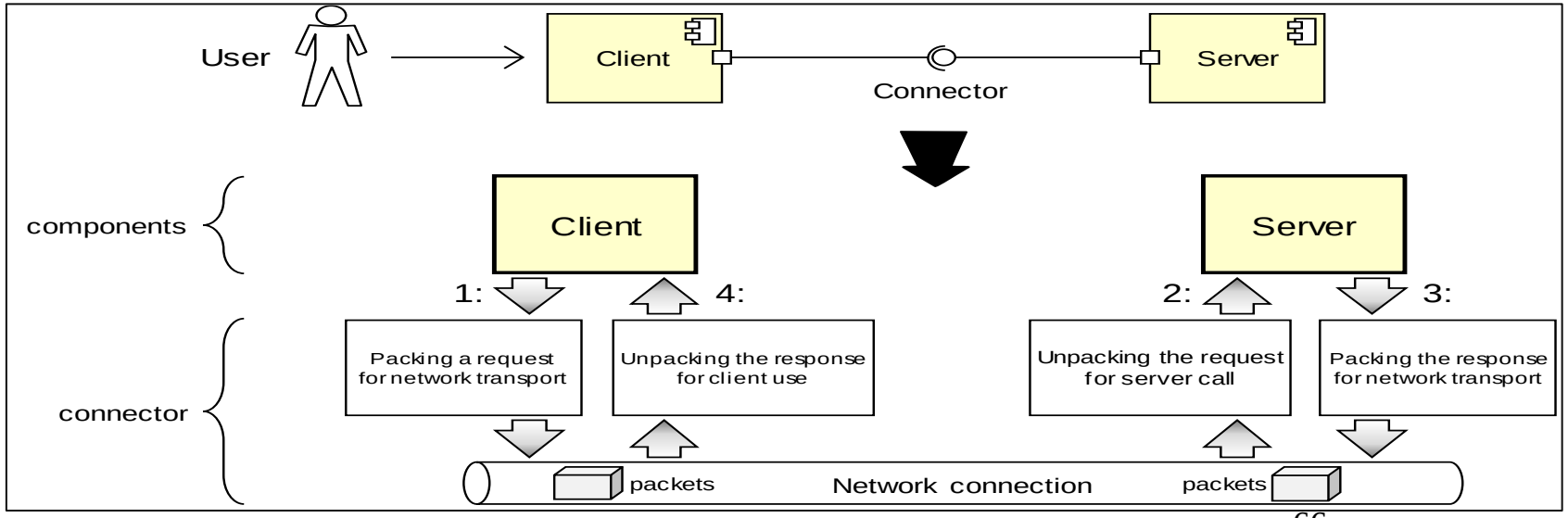
Style 2: 2 Tiered (Client/Server)- Example

- **Google offers its search engine as a webservice**
- A range of clients use SOAP to use the provided interface of the webservice
  - A webpage servlet client uses the webservice as part of its webpages
  - A .NET based b-2-c middleman application uses the webservice to search for best buys of products
- Different clients have very different purposes – but the webservice server has the same functionality
- *It makes sense to keep web searching and client functionality separate – to both maximize on possible uses of the server and also to enable better maintenance*
- Multiplicities of usage are written on the sides of connected interfaces
  - In this example, the webservice can have many clients, so we put * on the clients' required interfaces



Architecture Style: Style 2: 2 Tiered (Client/Server) (Summary)

- A **client** is a triggering process; a **server** is a reactive process. Clients make requests that trigger reactions from servers.
- A **server** component, offering a set of services, listens for requests upon those services. A server waits for requests to be made and then reacts to them.
- A **client** component, desiring that a service be performed, sends a request at times of its choosing to the server via a connector.
- The server either rejects or performs the request and sends a response back to the client



Style 3: N-tiered

N-tier architecture

N-tier architecture - also called or multi-tier architecture - refers to any application architecture with more than one tier. But applications with more than three layers are rare, because additional layers offer few benefits and can make the application slower, harder to manage and more expensive to run.

As a result, n-tier architecture and multi-tier architecture are usually synonyms for three-tier architecture.

- An **N-Tier Application** program is one that is distributed among three or more separate computers in a distributed network.
- **In N-tier, "N" refers to a number of tiers or layers are being used** like – 2-tier, 3-tier or 4-tier, etc. **It is also called "Multi-Tier Architecture".**
- The most common form of n-tier is the 3-tier Application, and it is classified into three categories.

Style 3: N-tiered

- **Similar to client / server**
- **Servers can also be clients of other servers**
  - Application logic is divided between the chain of components e.g., client could handle presentation logic, while one server handles business rules, connected to another server that handles the database
  - Clients and servers may be composite
- **Constraints**
  - Servers provide services to their clients, but do not require services from them
  - Clients require services of the server

Style 3: N-tiered

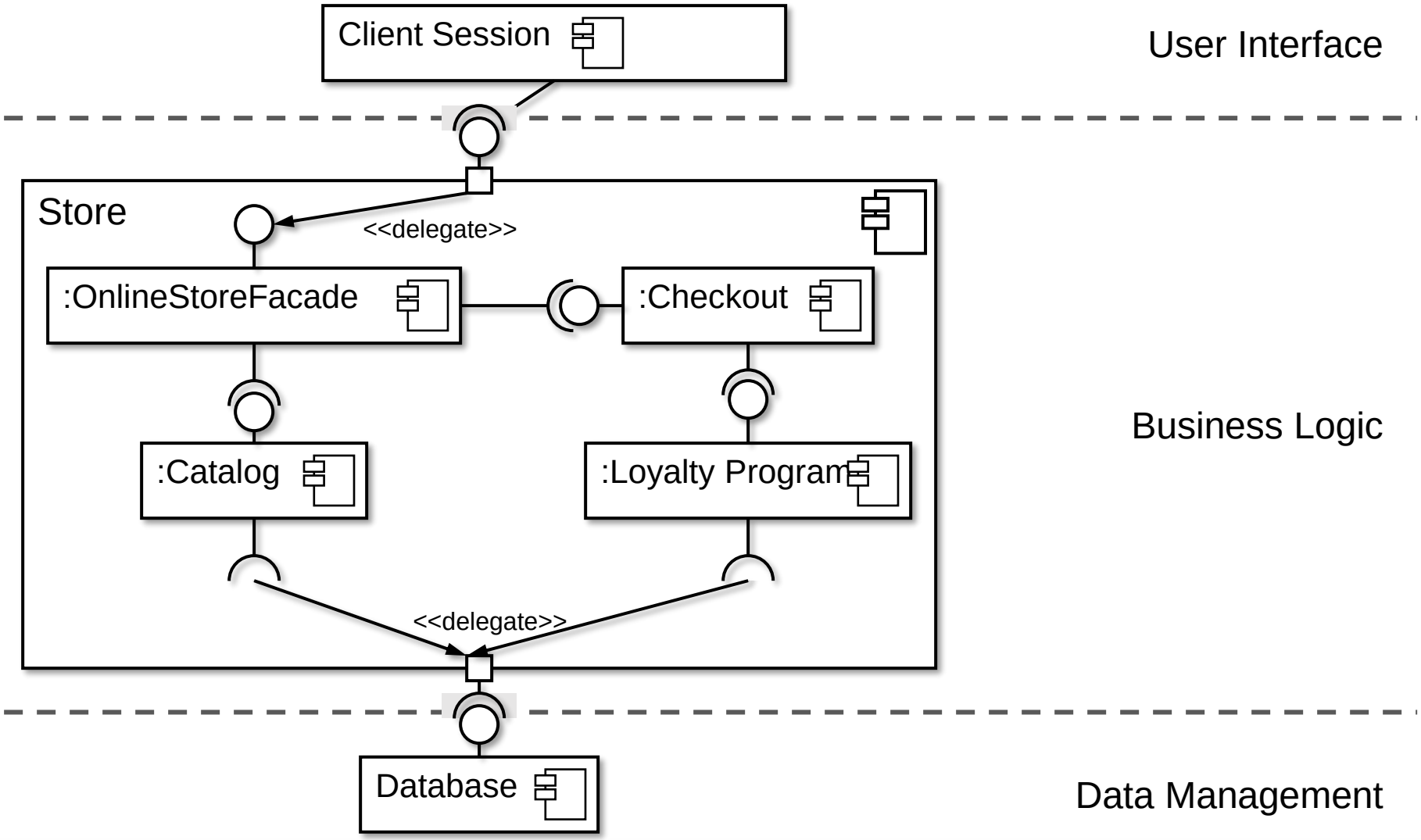
- **Semantics**
  - Servers do not know the number or identities of their clients
  - Clients know server's identity
  - Components are usually distributed (RPC used to implement connections)
- **Useful to separate application logic into software packages**

Style 3: N-Tiered

- **Advantages**
  - Separation of concerns, *divide and conquer approach* to solving large scale software problem
  - Maintenance

- **Disadvantages**
  - Performance (especially if distributed components)

Style 3: N-Tiered Example-01
A Typical 3-tiered Architecture



N-Tier: - Three Tier Architecture (Summary)

#3. Three-Tier Architecture:

Three Tier application in a Web Based application

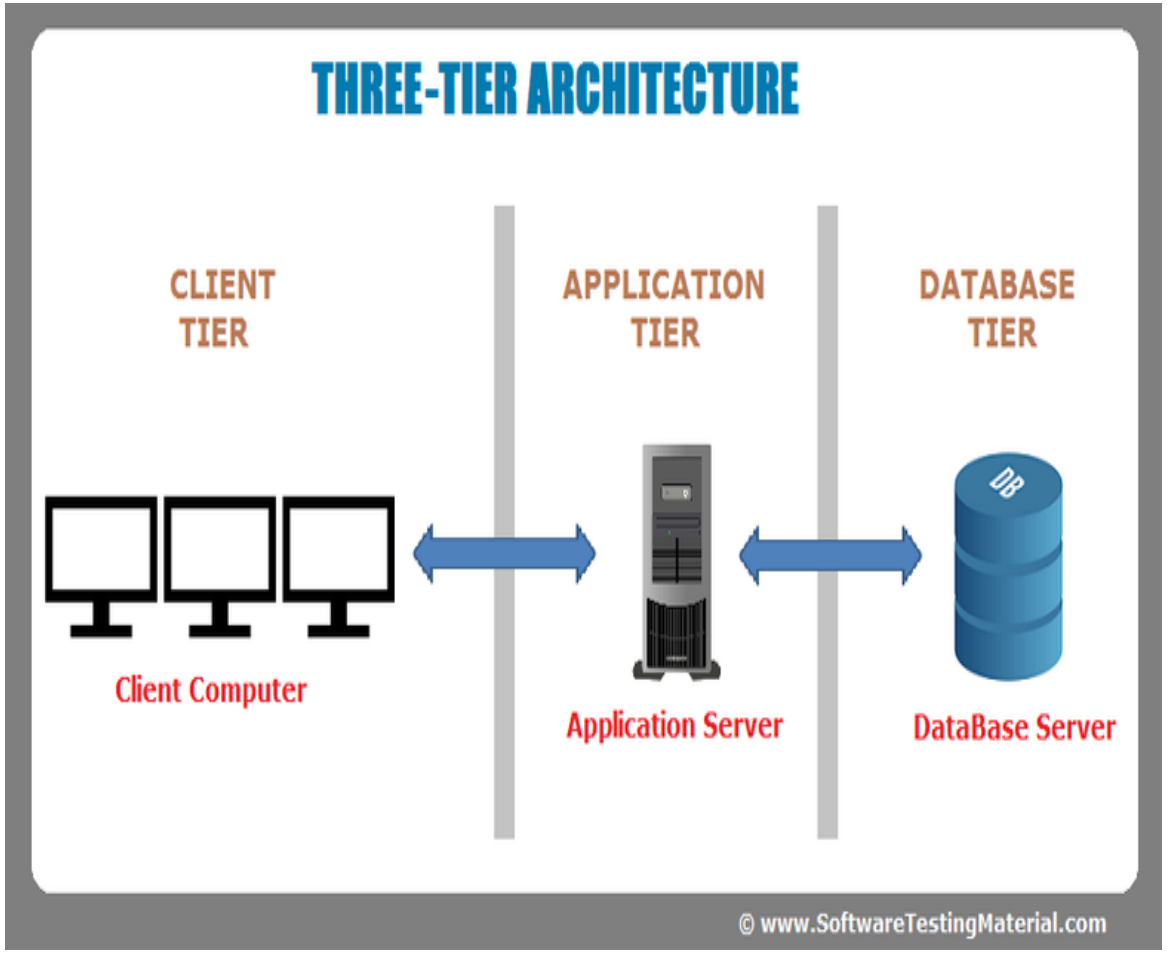
Three-Tier Architecture:

Three Tier application AKA Web Based application

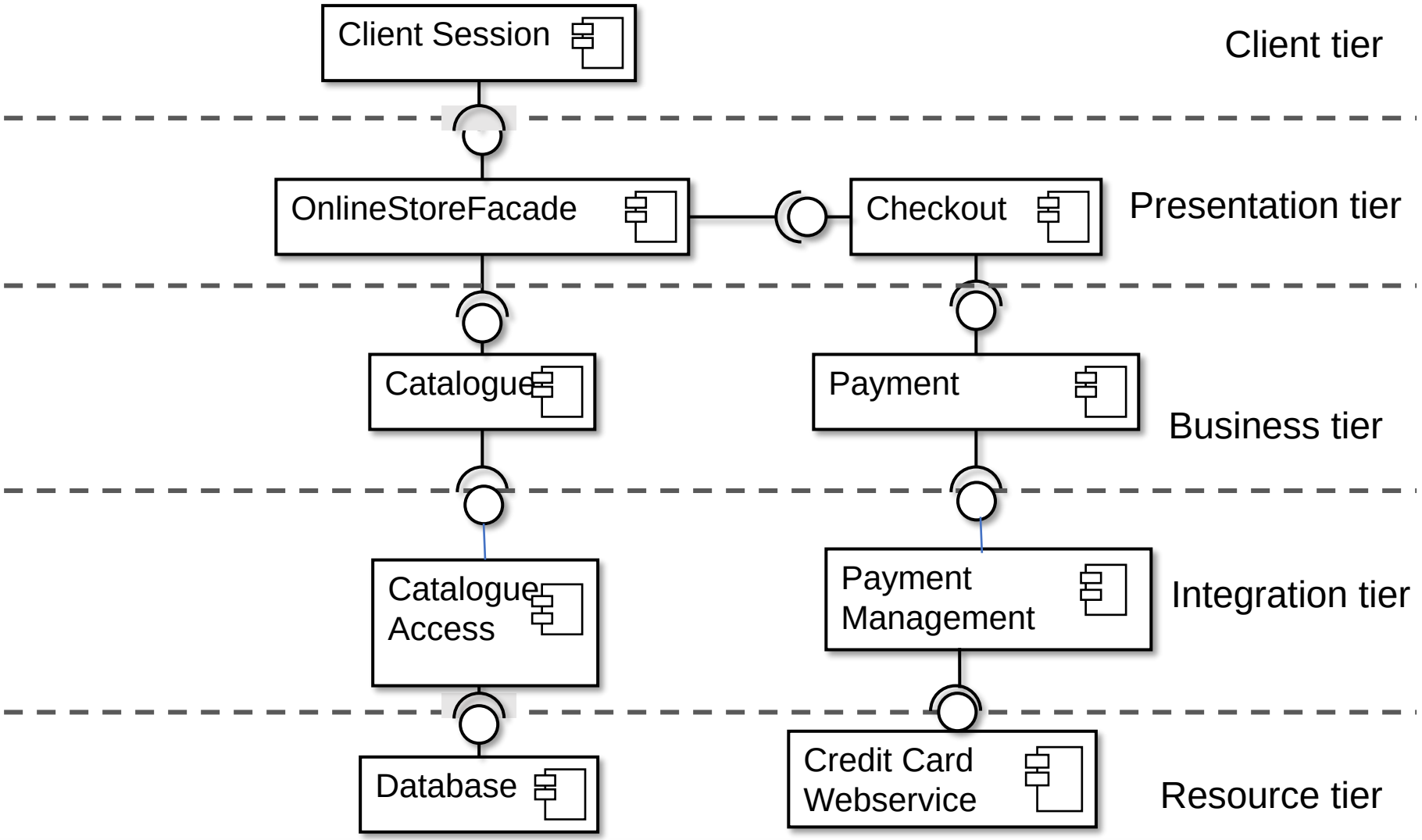
The Three-tier architecture is divided into three parts:

1. Presentation layer (Client Tier)
2. Application layer (Business Tier)
2. Database layer (Data Tier)

The client system handles the Presentation layer, the Application server handles the Application layer, and the Server system handles the Database layer.



Style 3: N-Tiered Example-02
EIS 5-tier architecture



Style 4: Layered Architectures

As the name suggests, components(code) in this pattern are separated into layers of subtasks and they are arranged one above another.

Each layer has unique tasks to do, and all the layers are independent of one another. Since each layer is independent, one can modify the code inside a layer without affecting others.

It is the most commonly used pattern for designing most of the software. **This layer is also known as 'N-tier architecture'. Basically, this pattern has 4 layers.**

1.Presentation layer (The user interface layer where we see and enter data into an application.)

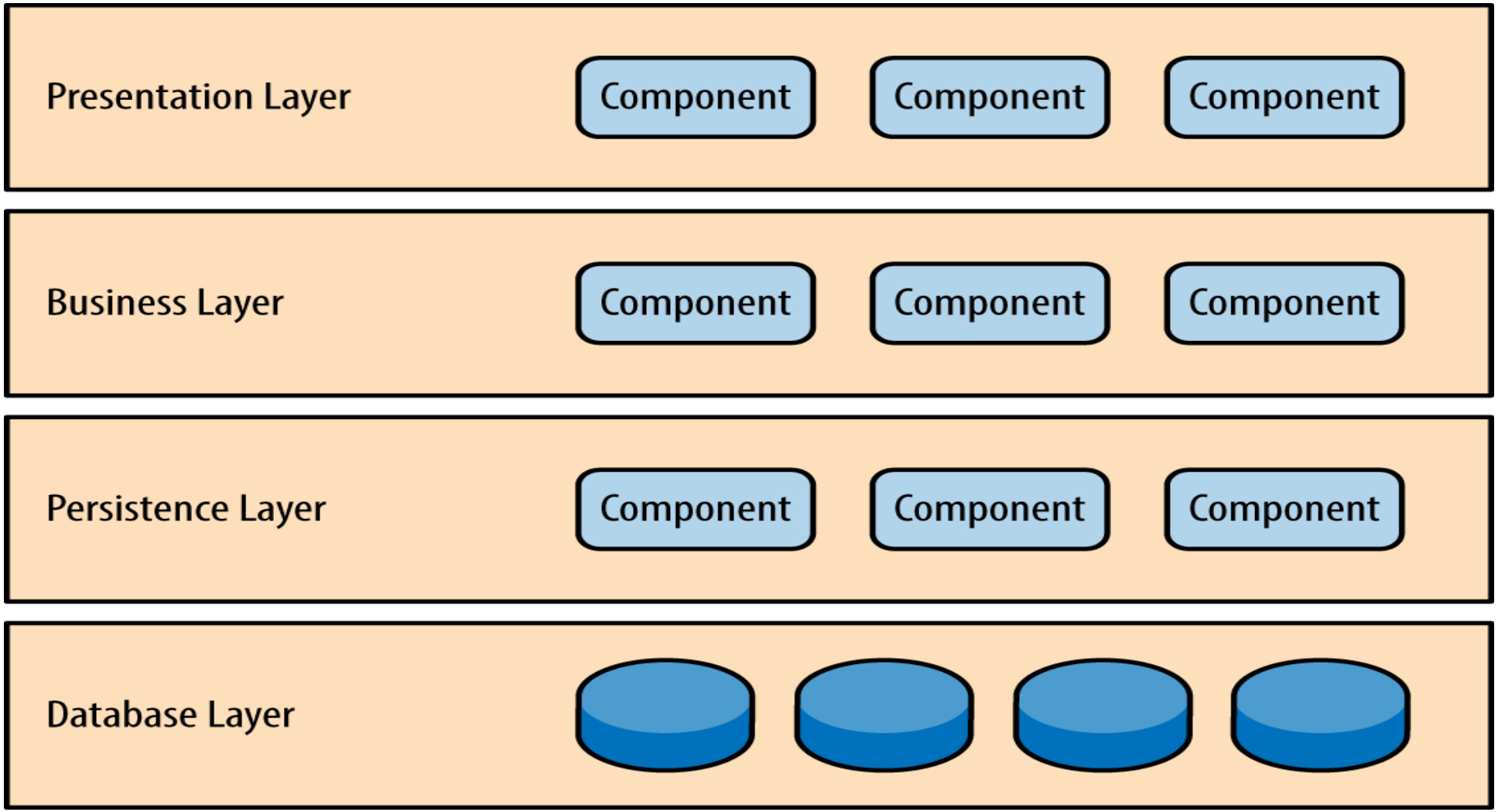
2.Business layer (this layer is responsible for executing business logic as per the request.)

3.Application layer (this layer acts as a medium for communication between the 'presentation layer' and 'data layer'.

4.Data layer (this layer has a database for managing data.)

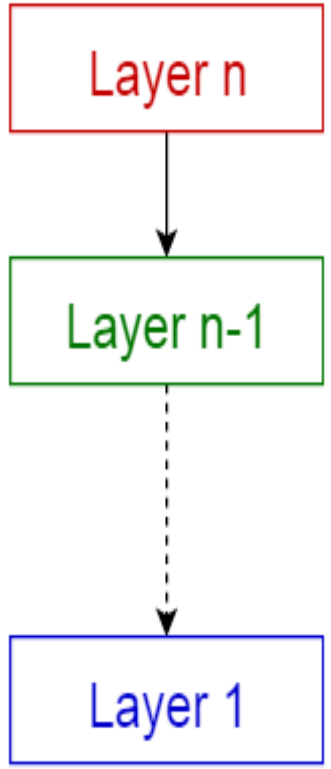
Ideal for: E-commerce web applications development like Amazon.

Style 4: Layered Architectures



Style 4: Layered Architectures

- **Hierarchical system organization**
  - "multi-level client-server"
  - Components are classified by layers
  - each layer exports an interface to be used by above layers
- **Constraints**
  - Each layer acts as a 1-1 relation
  - Each layer communicates only with its neighbors below it
- *Connectors are protocols of layer interaction*
- **Examples**
  - Operating Systems
  - Network Communications

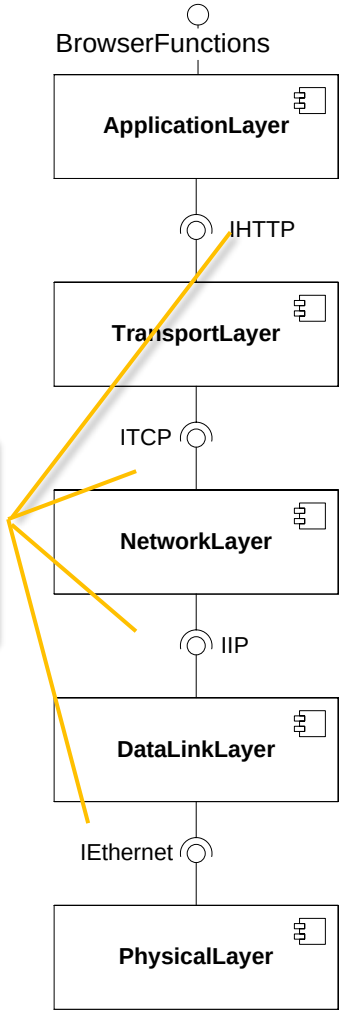


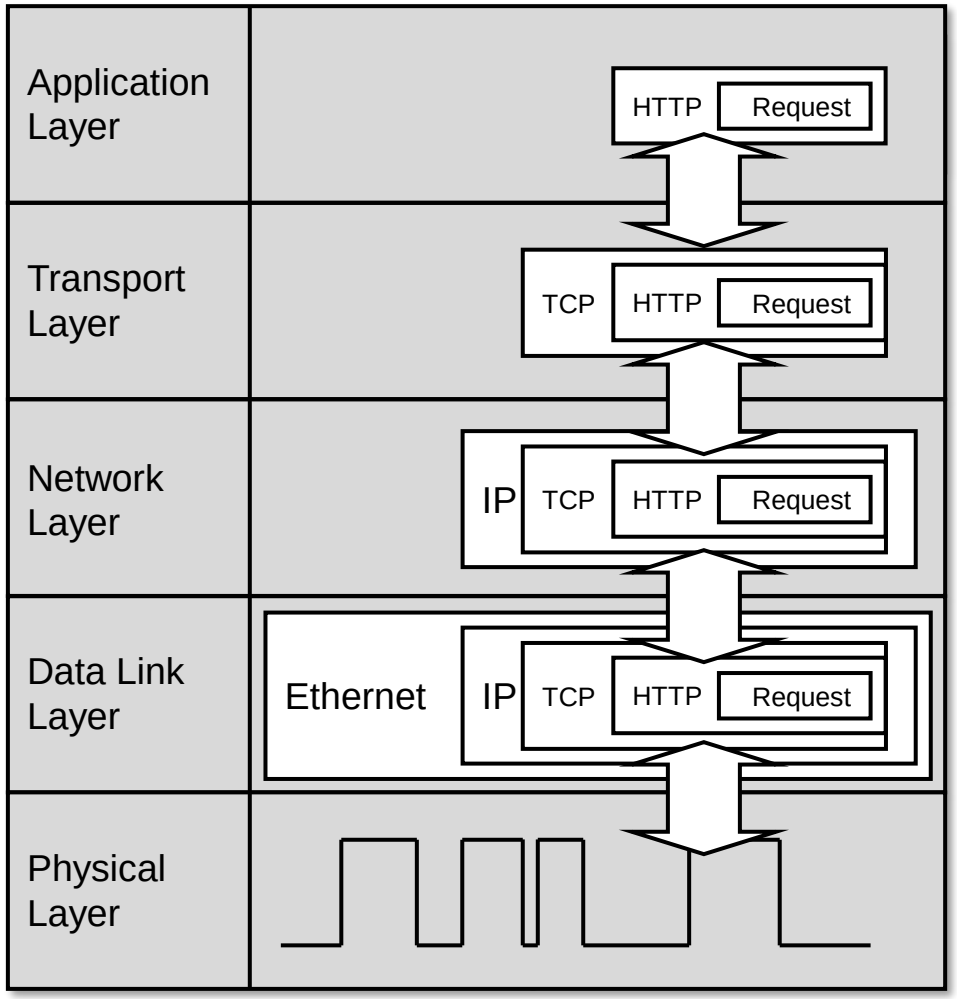
Style 4: Layered Architectures

- **Advantages**
  - Increased abstraction
  - Maintainability
    - Each layer's role is defined
  - Reuse
  - Portability
    - Replacing layers to work within a different context
- **Disadvantages**
  - Not universally applicable
    - Some components may be spread around different layers
    - Layer bridging may be required – a layer might need to communicate directly with a non-neighbour
  - Performance
  - Determining the correct abstraction level

Example: HTTP – Protocol stack in UML 2

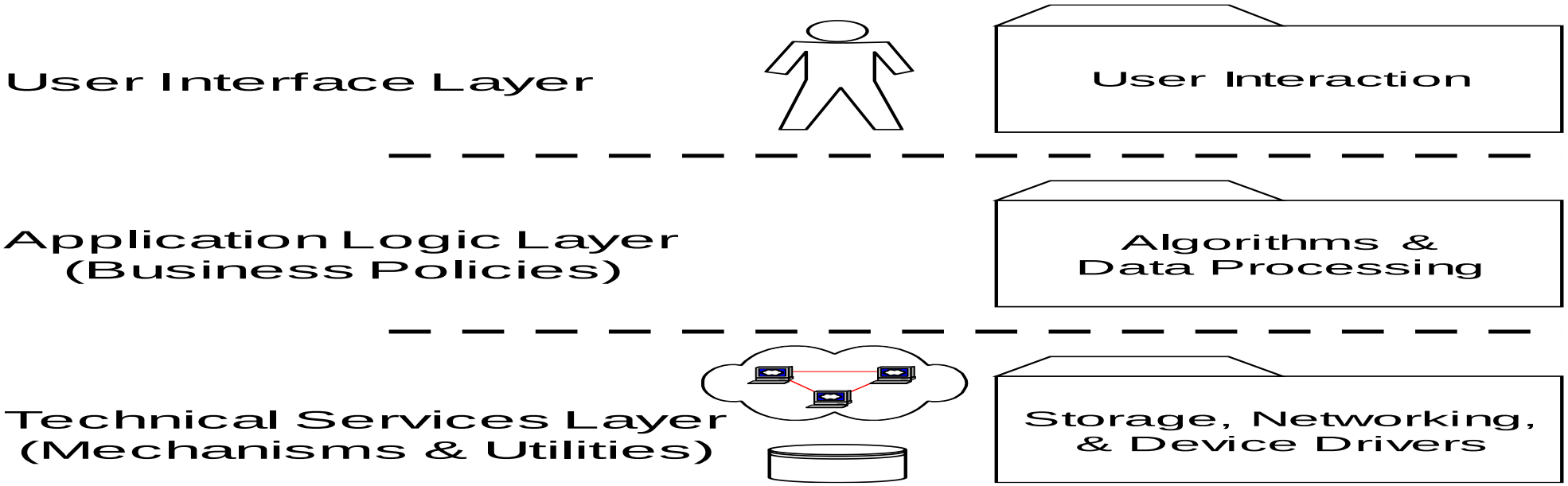
Each layer communicates to the next via an interface, taking information according to a certain network protocol





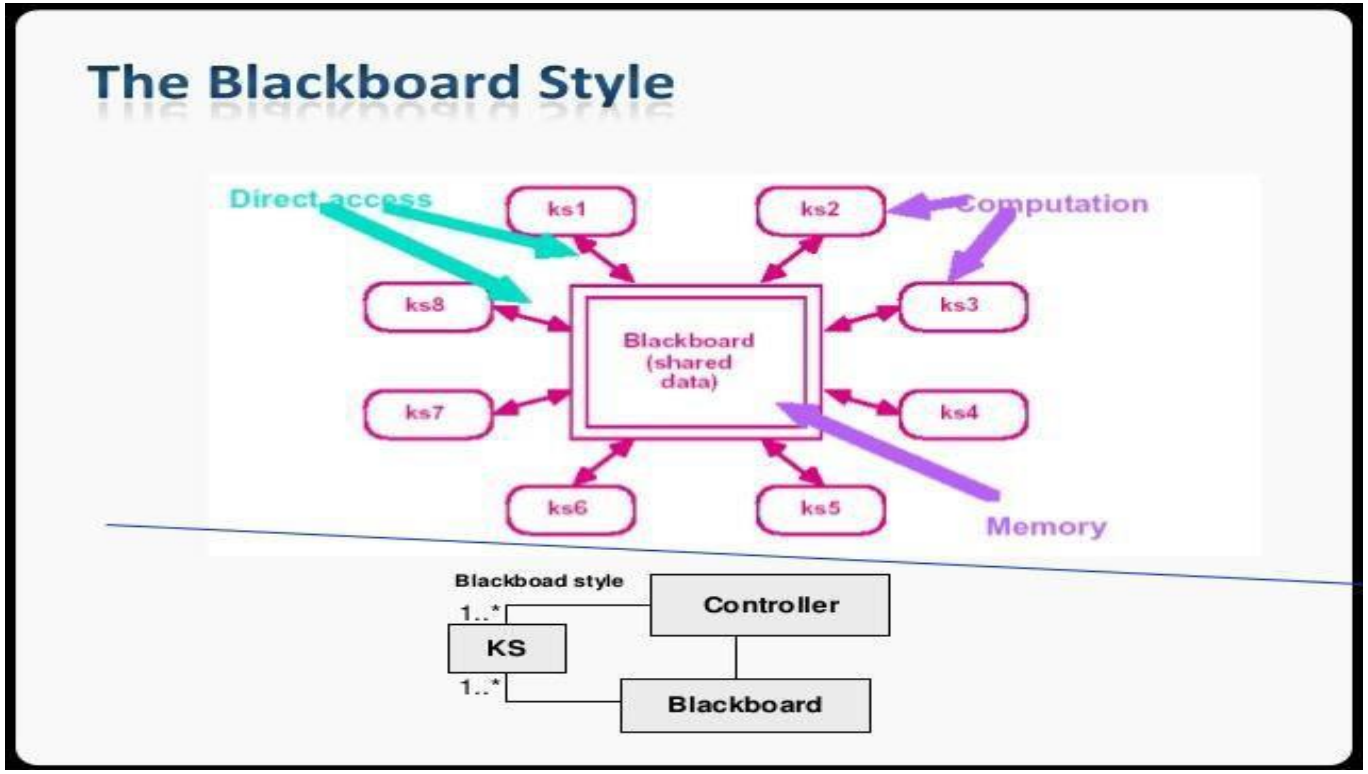
Architecture Style: Layered (Summary)

- A layered system is organized hierarchically, each layer providing services to the layer above it and using services of the layer below it
- Layered systems reduce coupling across multiple layers by hiding the inner layers from all except the adjacent outer layer, thus improving evolvability and reusability



Style 5: Blackboard

- The component types of the blackboard style are repositories and data processing components that use the repository.
- The data processing elements interact with the repository by scanning the repository for required inputs, and by writing to the repository when appropriate.

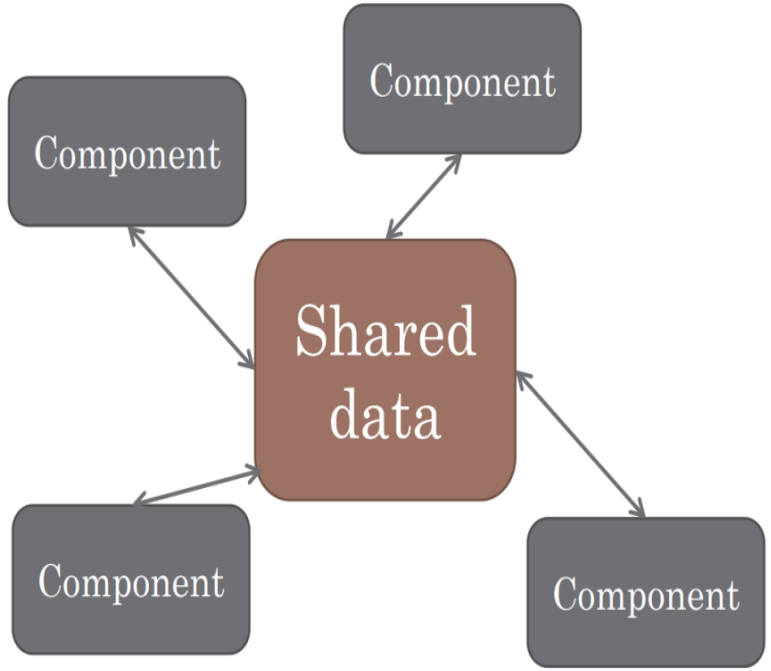


Style 5: Blackboard (Repository)

- Two kinds of components
  - Central data structure — blackboard or *repository*
  - *Client components* operate on and receive information from the blackboard
- System control is driven by the blackboard state
- Constraints
  - One blackboard
  - 1..n clients

Style 5: Blackboard (Repository)

- Two kinds of components
  - Blackboard
    - Central data structure (sometimes called repository)
  - Clients
    - Operate on and receive information from the blackboard
- System control driven by blackboard state
- Constraints
  - 1 blackboard
  - 1..n clients



Style 5: Blackboard

- **Semantics**
  - **Clients**
    - Encapsulate individual pieces of application functionality
    - Interact only through blackboard
    - Make changes to the blackboard that lead incrementally to a desired outcome
  - **Blackboard**
    - Active component
    - State change triggers selection of clients to execute
  - **Connectors**
    - Publish/Subscribe for clients to be informed of blackboard changes
    - Separate (a)synchronous connectors for changing the blackboard state

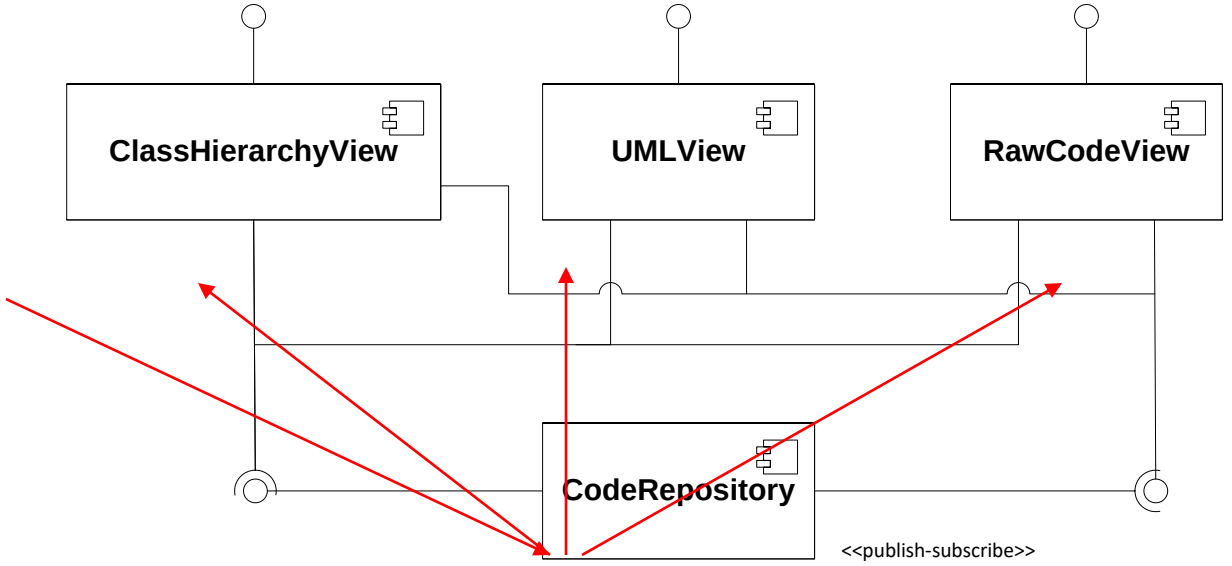
Style 5: Blackboard Examples

- **AI systems**
- Integrated software developments environments (IDEs)
  - E.g., Visual Studio, Borland Development Environment, Eclipse
- Compiler architecture
- **In e-commerce:**
  - Stateless components driven by stateful data components

Style 5: Blackboard 1st Example

- **Software development environment**
  - Stateful repository maintains software-project code
  - Three stateless components present views on the code
    - Change in one changes the code in the repository
    - This results in parallel changes to the other views

Changing class hierarchy (e.g., making one class inherit from another) results in change to code in repository and consequent updates to UML visual class structure and code view

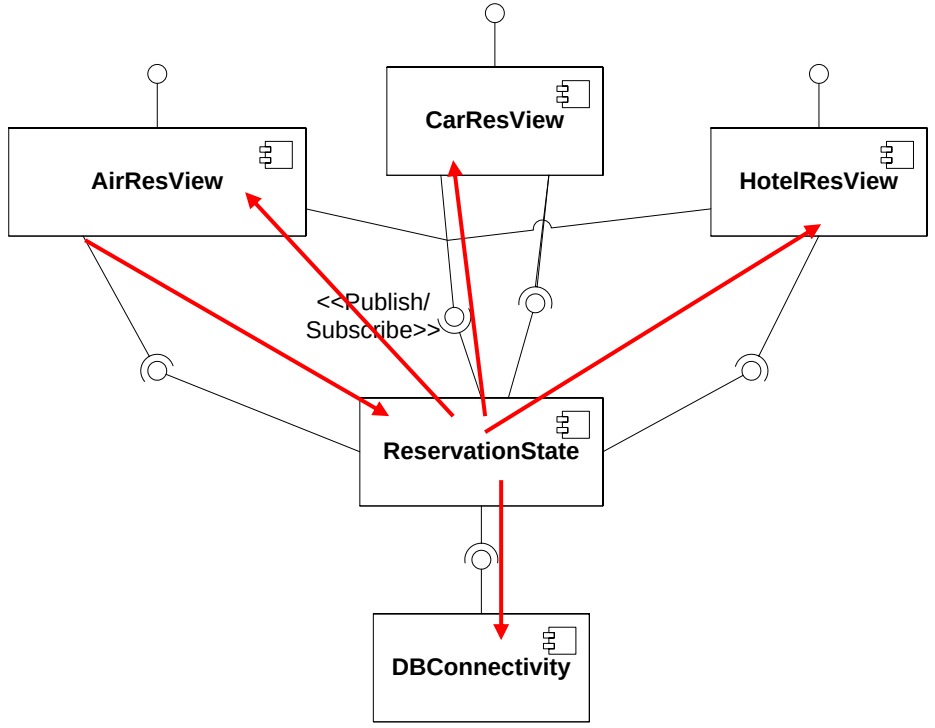


Style 5: Blackboard 2nd Example

- **Holiday reservation system**
  - All reservations ultimately stored in database
  - Three components for different aspects of holiday reservation logic
  - **Each component could connect to the database separately**
    - Would tax performance
    - Especially as a change in one aspect (like date of air reservation) affects another aspect (like valid dates to book a hotel)
  - **Solution: use an intermediate single blackboard component**
    - Connects to database
    - Deals with all data of a user's reservation
  - **Three reservation logic components can change the blackboard**
  - Components can also subscribe to be notified of important changes to the blackboard data

Style 5: Blackboard 2nd Example

1. Dates of aeroplane ticket changed – Reservation State updated
2. Reservation State sends a notification if car reservation or hotel reservation dates do not lie within aeroplane dates
3. Only valid data is sent to database.



Style 5: Blackboard

- **Advantages**
  - Clients are relatively independent of each other
  - Data store is independent of clients
    - Scalable (new clients can be added easily)
    - Modifiable, maintainable
  - Data integration can be achieved through the blackboard

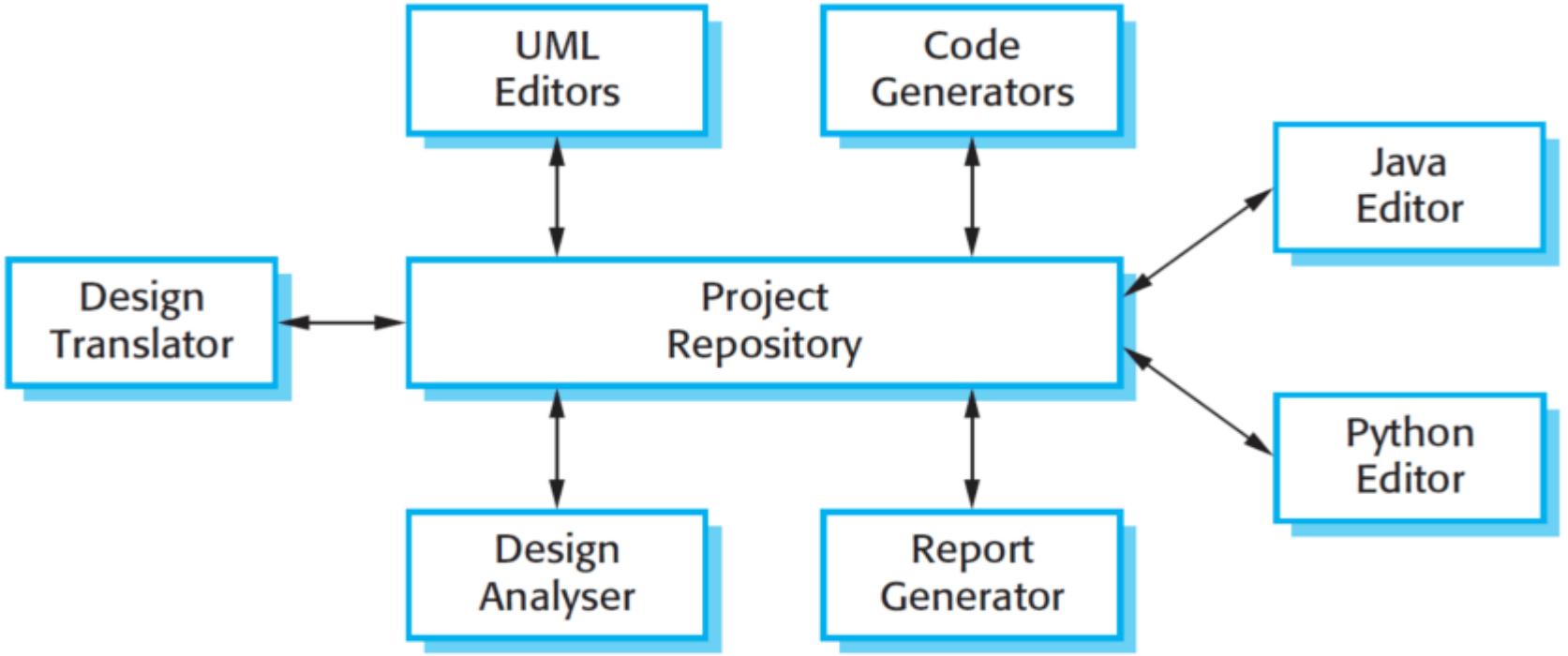
Style 6: Repository Architecture

- **Sub-systems must exchange data**. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

Style 6: The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Style 6: A repository architecture for an IDE



A repository architecture for an IDE

Style 6: A repository architecture for an IDE

Advantages

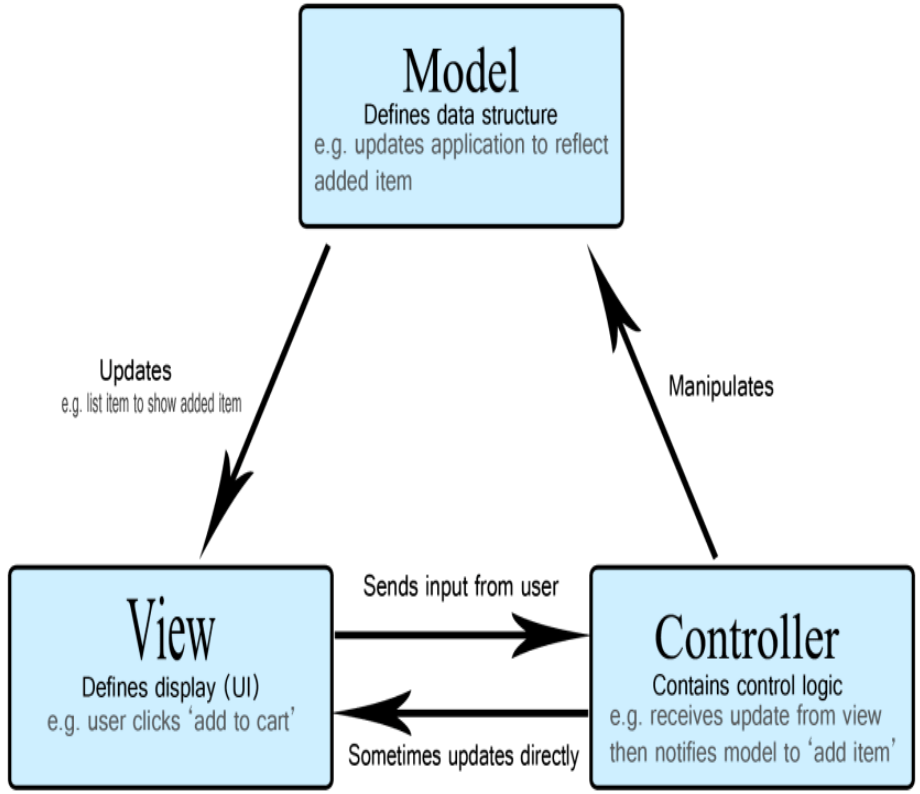
- Scalability - new components can be added
- Concurrency - all components can work in parallel
- Reuse - components are not direct communication with each other
- Centralized data management
- Better conditions for security, archiving, etc.
- The components are independent of the manufacturer data

Disadvantages

- High dependence between the data structure of storage and components
- Storage - single point of failure (single point of failure)
- Changes in data structure strongly affect customers
- Multiple sync issues components

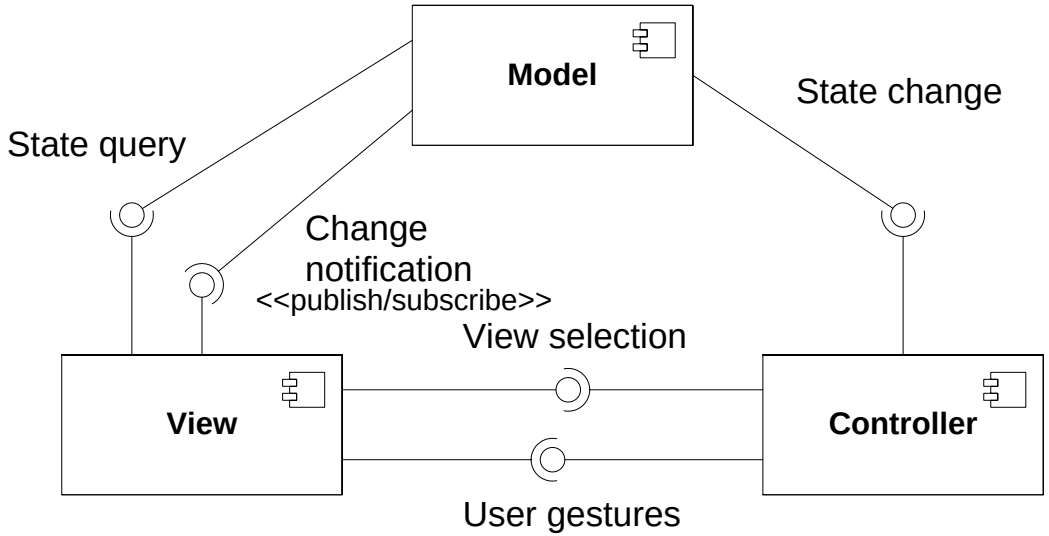
Style 7: Model-View-Controller

- Specifically for supporting multiple views and modifiers for shared data
- Three different kinds of components
  - **View**
    - Responsible for displaying information to the user
  - **Controller**
    - Responsible for managing interaction with user
  - **Model**
    - Responsible for maintaining domain knowledge
    - Do not depend on any view or controller
    - Propagate changes in state to views via publish/subscribe



Style 7: Model-View-Controller

- **Semantics:**
  - Model
    - encapsulates application state and functionality
    - Notifies views of changes
  - View
    - Requests updates from models
    - Maps user gestures to controller actions
    - Allows controller to select view
  - Controller
    - Maps user actions to model updates
    - Maps user actions to model updates (often implemented as publish/subscribe)
    - Selects view for response



Generalises Observer OO pattern and Blackboard architecture: Controller manages change instead of the model (blackboard)

Style 7: Model-View-Controller

- Three different kinds of components
  - *Model* components are responsible for maintaining domain knowledge
  - *View* components are responsible for displaying information to the user
  - *Controller* components responsible for managing the sequence of interactions with the user.
- The model subsystems are designed such that they do not depend on any view or controller subsystems.
- Changes in their states are propagated to the view subsystems via a subscribe/notify protocol.

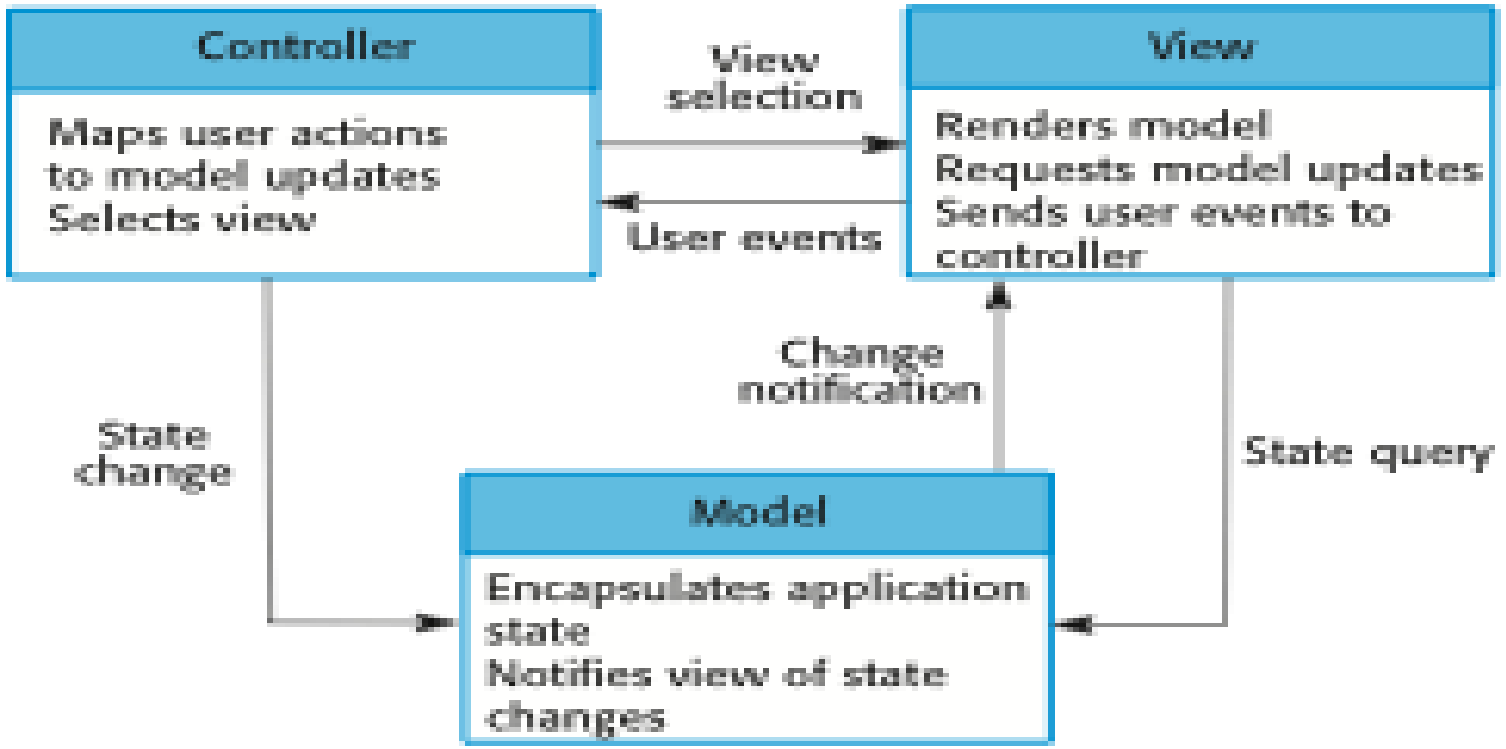
• Style 7: Model-View-Controller

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.
Example	Figure (next slide) shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

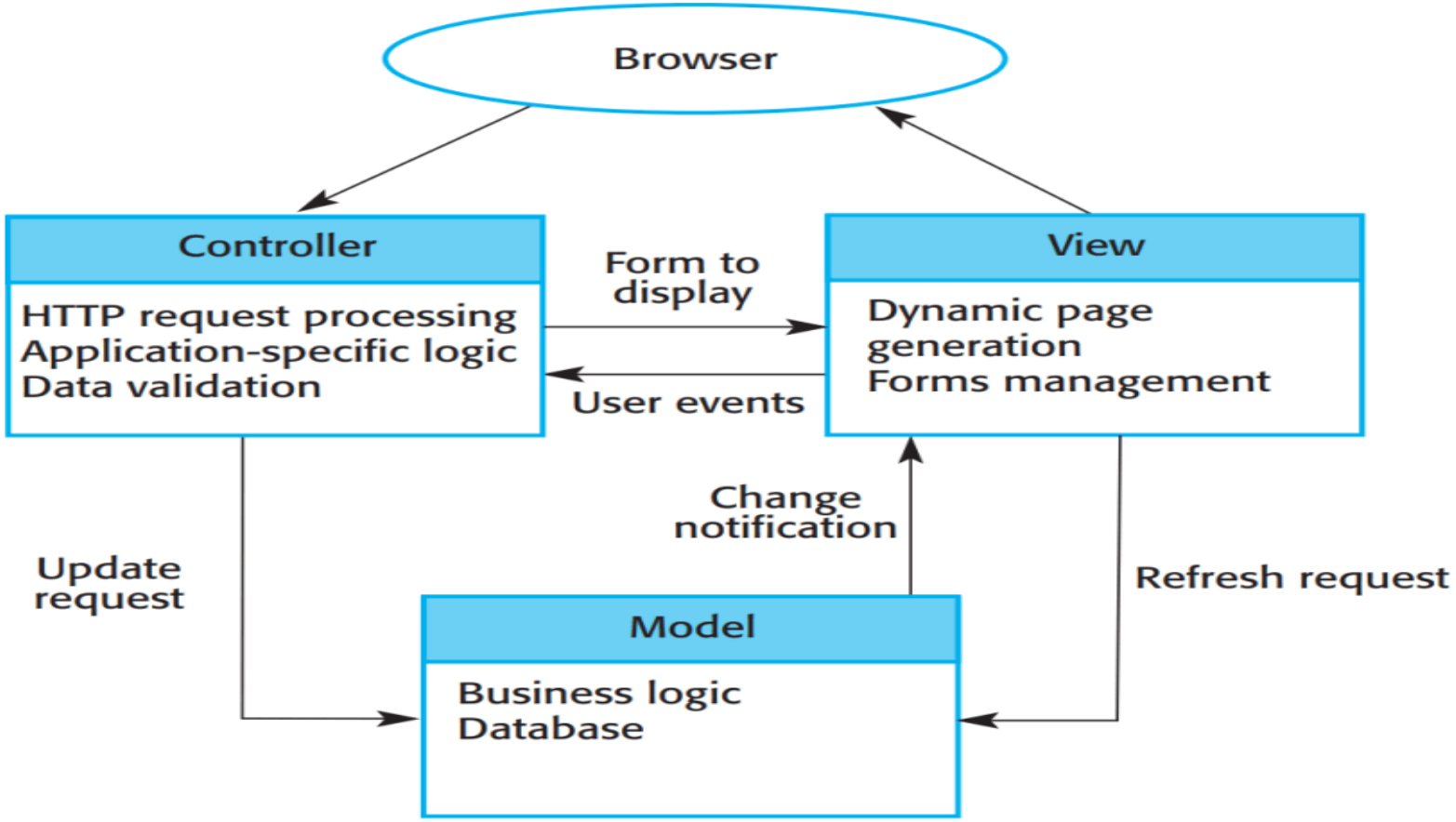
• Style 7: Model-View-Controller

- **Advantages**
- Great flexibility
- Easy to maintain and implement future improvements
- Clear separation between presentation logic and business logic
- Allows data to change independently their performance and vice versa
- Multiple views for one model
- **Disadvantages**
- Even if the data model and interactions are simple, this style can introduce complexity and require a lot of code
- Not suitable for small applications
- Performance issue with frequent updates in model
- Software developers using MVC must be skilled in many technologies

The organization of the Model-View-Controller

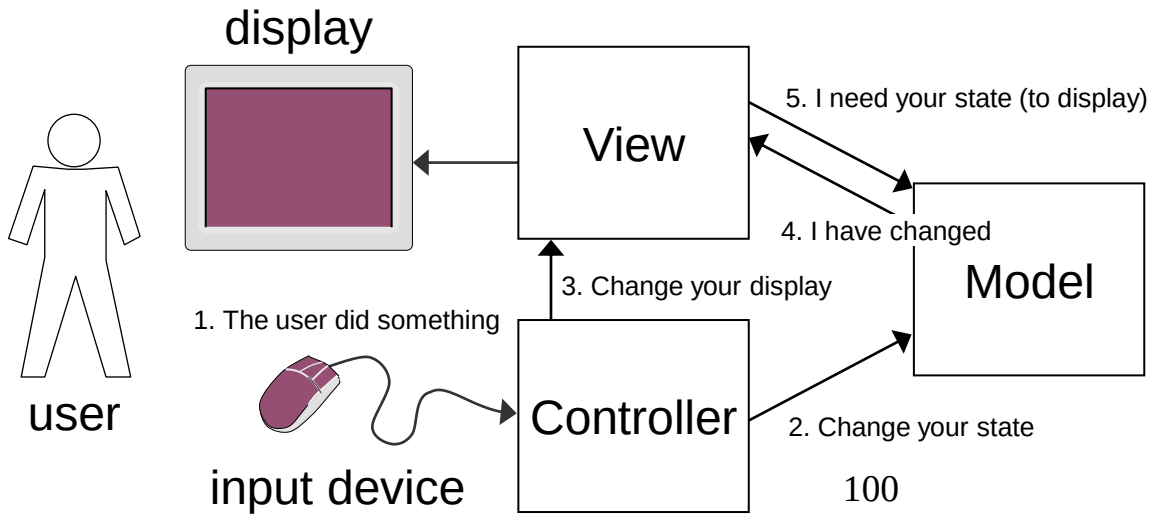


Web application architecture using the MVC pattern



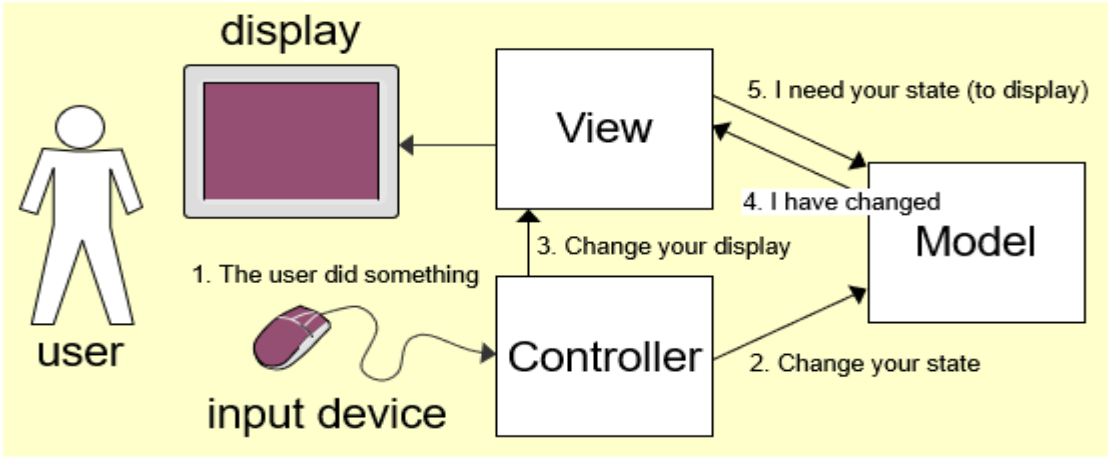
Style 7: Model-View-Controller(Summary)

- **Model**: holds all the data, state and application logic. Oblivious to the View and Controller. Provides API to retrieve state and send notifications of state changes to "observer"
- **View**: gives user a presentation of the Model.
  Gets data directly from the Model
- **Controller**: Takes user input and figures out what it means to the Model



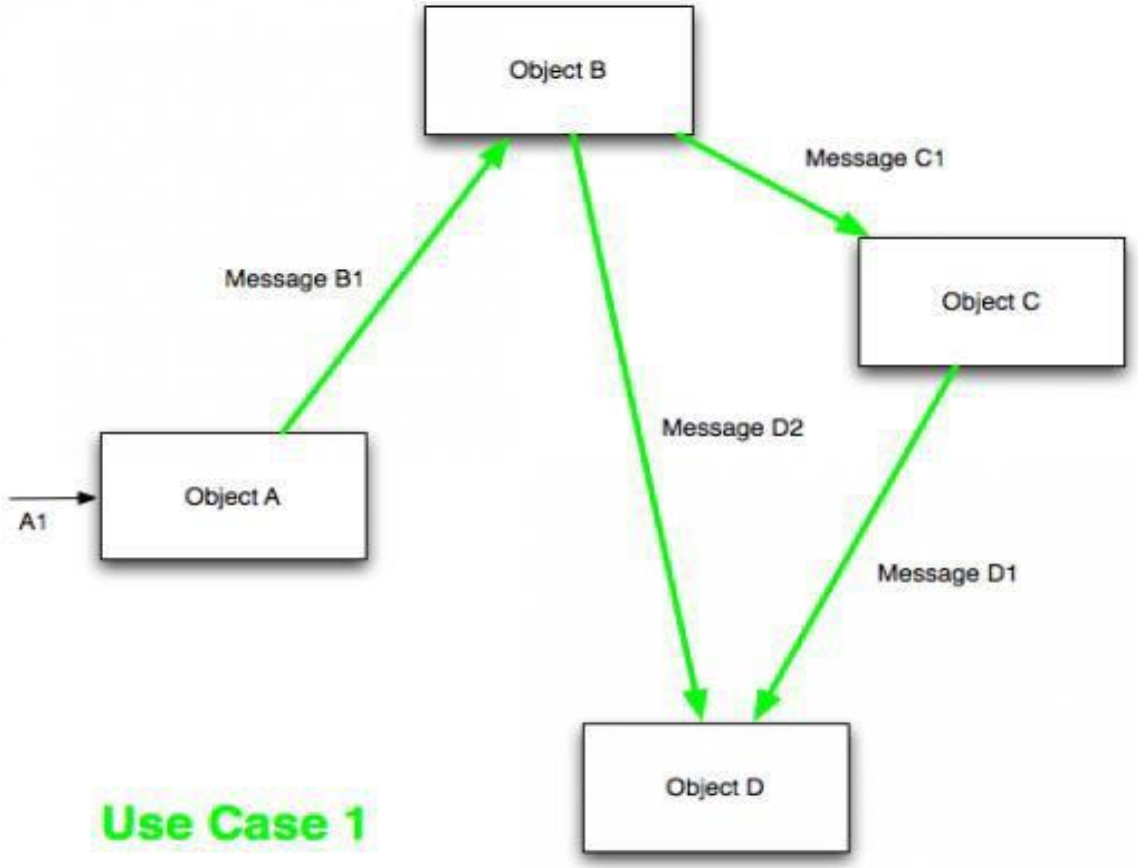
Style 7: Model-View-Controller(Summary)

- **Model**: holds all the data, state and application logic. Oblivious to the View and Controller. Provides API to retrieve state and send notifications of state changes to "observer"
- **View**: gives user a presentation of the Model.
Gets data directly from the Model
- **Controller**: Takes user input and figures out what it means to the Model



Style 8: Object Oriented

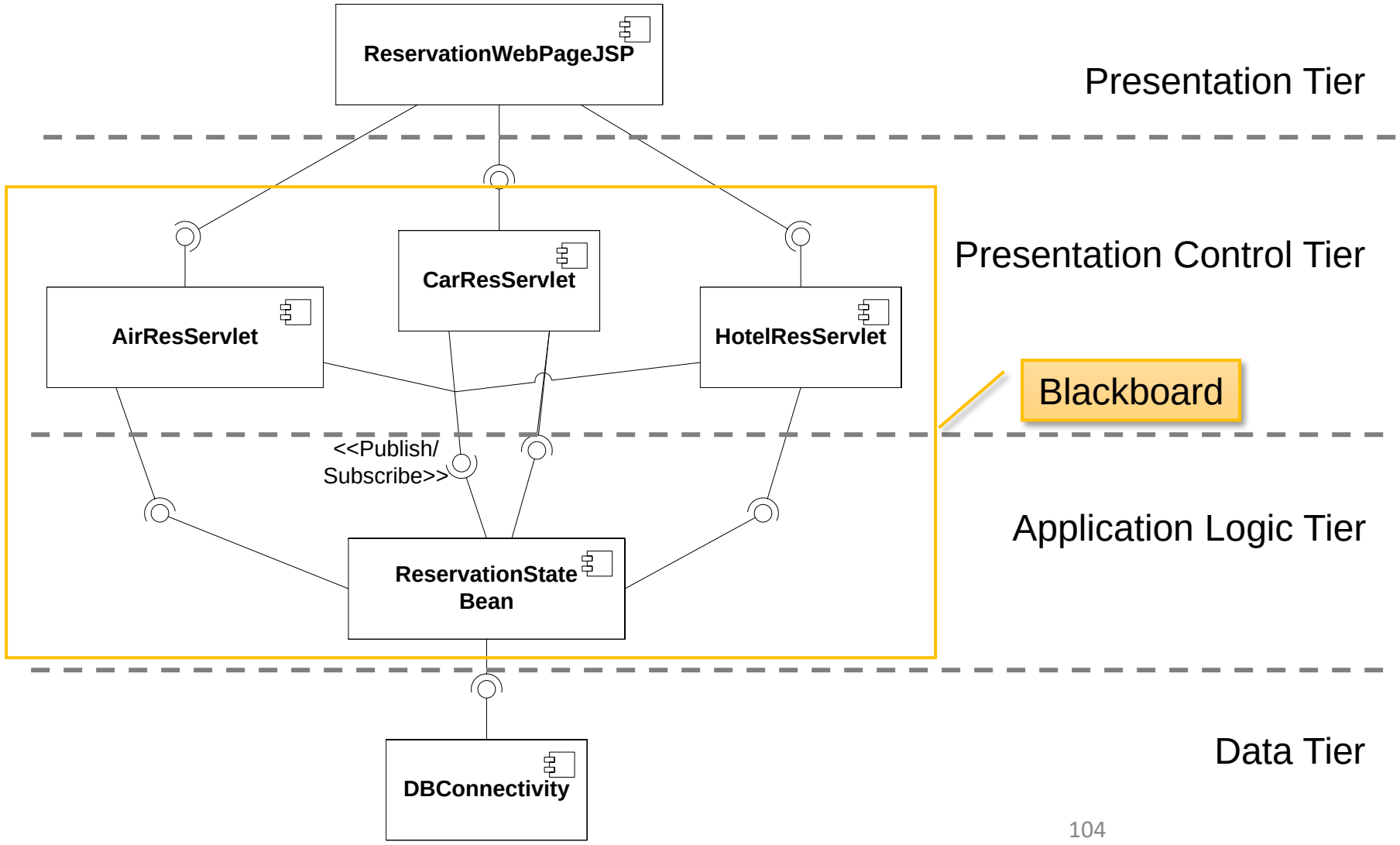
➢ The component type of the object-oriented style is the object and the connector type is the synchronous message passing, or procedure call.



Heterogeneous Styles

- **Combine different architectural styles**
  - Large architectures involve a combination of functionalities
  - Often entails a combination of styles in the best solution
- **Challenge:**
  - Ensure advantages of individual style choices are not negated by their combination
- **Good practice:** Apply styles compositionally
  - Decompose problem into subsystems
  - Apply one style per subsystem (e.g., in a composite component)
  - Helps avoid style benefits being in conflict
- **Safe practice:** Mix styles addressing orthogonal issues
  - Example:
    - N-tiered style is used to structure an overall architecture
    - Blackboard style is used for a specific kind of problem solving
    - Styles can be combined

Example: 4-tiered Blackboard



Contrast: IDE vs. Architecture-based DE

Code-based IDE Tools:

- Text editor
- GUI Builder
- Compiler
- Debugger
- Static Analysis Tools
- Project Management
- Source file configuration management

Architecture-based Tools:

- Visual Editor
- Composition assistants
- Code generator
- Instantiation & Evolution Management
- Various analysis tools
- Style-specific constraint management
- Component-based configuration management

Architectural Models and Styles

- Architectural models are achieved in UML2 by:
  - *Component-and-connector diagrams*, to define component relationships
  - *Interface descriptions*, consisting of:
    - method signatures (method names, input and output types)
    - a semantics that tells us how methods work
- Architectural styles are defined in UML2 through:
  - Explaining the kinds of *components* that are used in a style
  - A *semantics* explaining how components and connections should behave
  - *Constraints* telling us how the components are to be assembled

Architectural Style vs Architectural Pattern

- In general, both terms are used interchangeably. For example, MSDN says that Architectural Styles and Architectural Patterns are the same things.

- However, there is a difference. The key difference is the **scope**.
- An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context.
- An Architectural Pattern is a way of solving a recurring architectural problem. MVC, for instance, solves the problem of separating the UI from the model. Sensor-Controller-Actuator, is a pattern that will help you with the problem of actuating in face of several input senses.
- An Architectural Style, on the other hand, is just a name given to a recurrent architectural design. Contrary to a pattern, it doesn't exist to "solve" a problem.
- Pipe & filter doesn't solve any specific problem, it's just a way of organizing your code. Client/server, Main program & subroutine and Abstract Data Types / OO, the same.
- An Architectural Style is the application design at the highest level of abstraction.

- An Architectural Pattern is a way to implement an Architectural Style;

Architectural Style vs Architectural Pattern

- Architectural styles tell us, in very broad strokes, how to organise our code. It's the highest level of granularity and it specifies layers, high-level modules of the application and how those modules and layers interact with each other, the relations between them. Examples of Architectural Styles:
  - Component-based
  - Monolithic application
  - Layered
  - Pipes and filters
  - Event-driven
  - Publish-subscribe
  - Plug-ins
  - Client-server
  - Service-oriented

- Architectural Patterns solve the problems related to the Architectural style. For example that classes will we have and how will they interact, in order to implement a system with a specific set of layers that high-level modules will have in our Service-Oriented Architecture and our Client-ser| er Architecture Architectural Patterns have an extensive impact on the code base, most often impacting the whole application horizontally (ie. how to structure the code inside a layer) or vertically (ie. how a request is processed from the outer layers into the inner layers and back). Examples of Architectural Patterns:
  - Three-tier
  - Microkernel
  - Model-View-Controller
  - Model-View-ViewModel

Architecture-Driven Component Development

- The goal for the embodiment phase of design is to either build or *select components and connectors that possess the quality attributes identified during the architecting phase of development.*

- Three types of components:
  1. Custom built components
  2. Reusable components
  3. Commercial components

1- Custom Components

- Demands both time and money.
- Are most likely to pay off in cases of software that are:
  - Very unusual
  - Safety critical
  - Highly secure
- The component assembly will possess the quality attributes it was designed around.

Pre-Existing Components

- There are two main classes of pre-existing components:
  - Reusable components
  - Commercial components
- Is a fundamentally different problem than custom design.
  - The requirements to use specific components and component frameworks drive the architecture.

2- Reusable Components

- Can exist on a wide scale of reusable-ness within any organization.
- Custom designed components can be adapted
  - *In most cases it will be necessary to create adaptors, often referred to as glue code.*
- Are developed with reuse in mind.
  - These may be in the form of parameterized components that can readily be used in a variety of settings and with a variety of property values and interdependencies explicitly stated.

3- Commercial Components

- Introduce a large degree of uncertainty.
- Tend to be
  - Complex
  - Idiosyncratic
  - Unstable

Component-Driven Architecture Development

- Constraints due to the use of pre-existing components:
  - *Design freedom is limited to component selection*.
  - Sufficient information about how a component will behave is not generally provided.
  - Component properties must be verified.
  - The framework into which components are to be plugged influences the architecture and the process by which the system is designed.
  - *Such components can not be optimized*.

- It is expected that *more reliable systems will be produced*, with greater speed and at lower expense due to the restrictions on design freedom.


Thank
you