CMPS310
Fall 2021

Lecture 12

Software Architectural Styles

Architectural Elements

- A system designer primarily focuses on the **components** and interactions among them using **connectors**
- **Components**: objects, databases, filters, ADTs
  - Different levels of software design require
    - different kinds of components (*object, class, function, procedure*)
    - different *ways of composing components* using connectors
    - different *design issues* and different kinds of *reasoning*
- **Connectors**: play a fundamental part in distinguishing one architecture form another and it mediates interactions of
    - components
    - procedure calls
    - message passing
    - method call
    - pipe
    - shared memory
    - event broadcast

Some Common Architectural Styles

Styles or patterns are categorised into related groups in an inheritance hierarchy

1. **Call-and-return:**
    - **1.1. main program and subordinate**
    - **1.2. object-oriented systems**
    - **1.3. Layered approach**
2. **Independent components:**
    - **2.1. implicit invocation**
    - **2.2. communicating processes**
    - 2.3. explicit Invocation
3. **Data flow:**
    - **3.1. pipes and filters**
    - 3.2. batch sequential
4. **Virtual machines:**
    - 4.1. Interpreters
    - 4.2. Rule-based systems
5. **Data-centric systems:**
    - **5.1. shared data storage**
    - **5.2. blackboard**
    - **5.3. Repository**
6. **Other style**
    - 6.1. JSD (object-based)

Note: We will address only the red color styles in this course

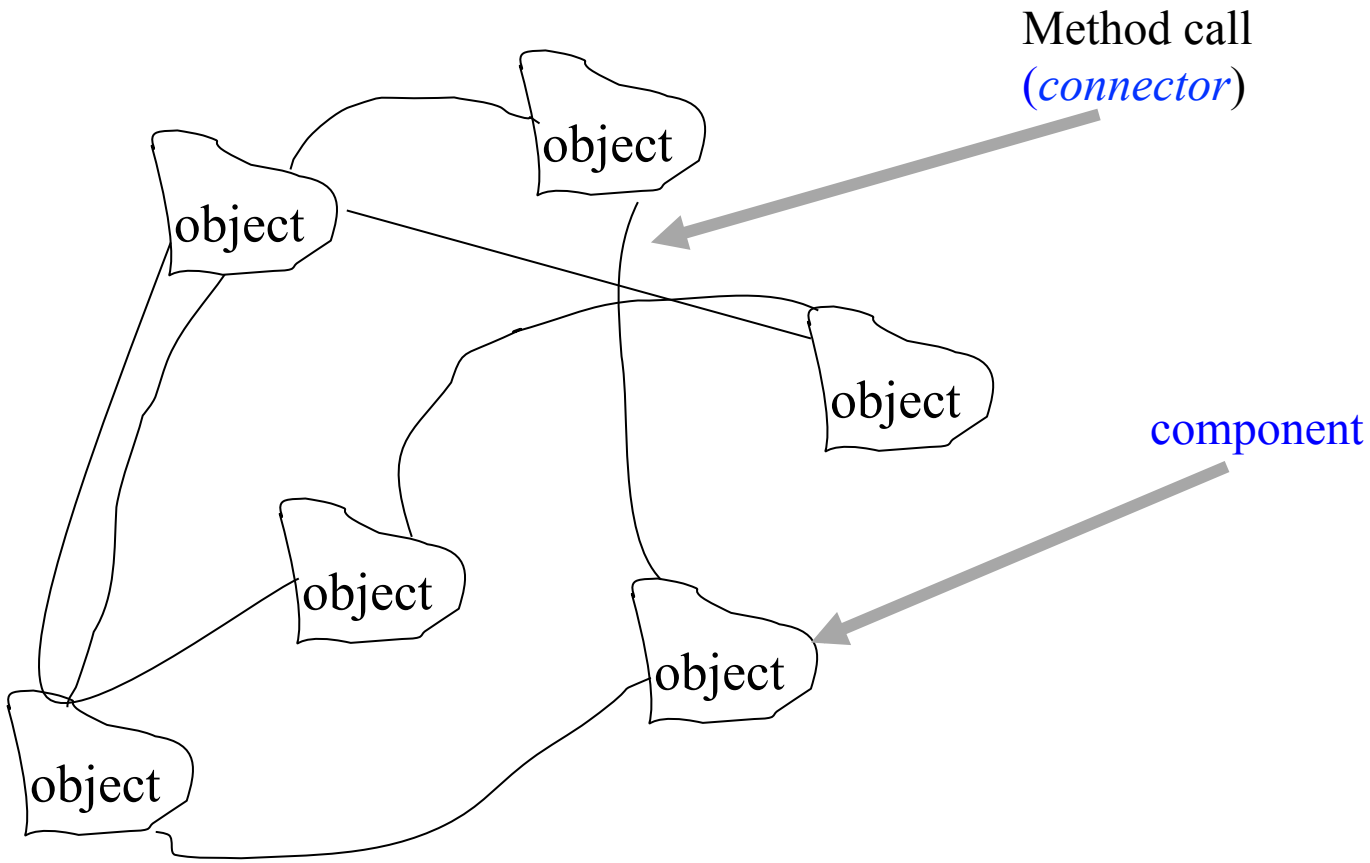
1. Call and Return Style

- This style has the goal of achieving the qualities of modifiability, reusability, and scalability
- It has been the dominant architectural style in large software systems for the past 40 years or so
- It has three main variations
  - **1.1 Object oriented or object-based**
  - **1.2 Main program and sub routine**
  - **1.3 Layered**

1.1. Object Oriented

- Based on data abstraction and OO structure
- The components of this style are objects or instances of classes
- Objects interact through method invocations/message passing (connector)
- Some systems allow objects to be concurrent tasks; others allow objects to have multiple interfaces
- Determine actual operation to call at run time
  - *Dynamic object binding*
- Usually, the topology of O-O is not hierarchical

Non-hierarchical Topology in Object-Oriented Style



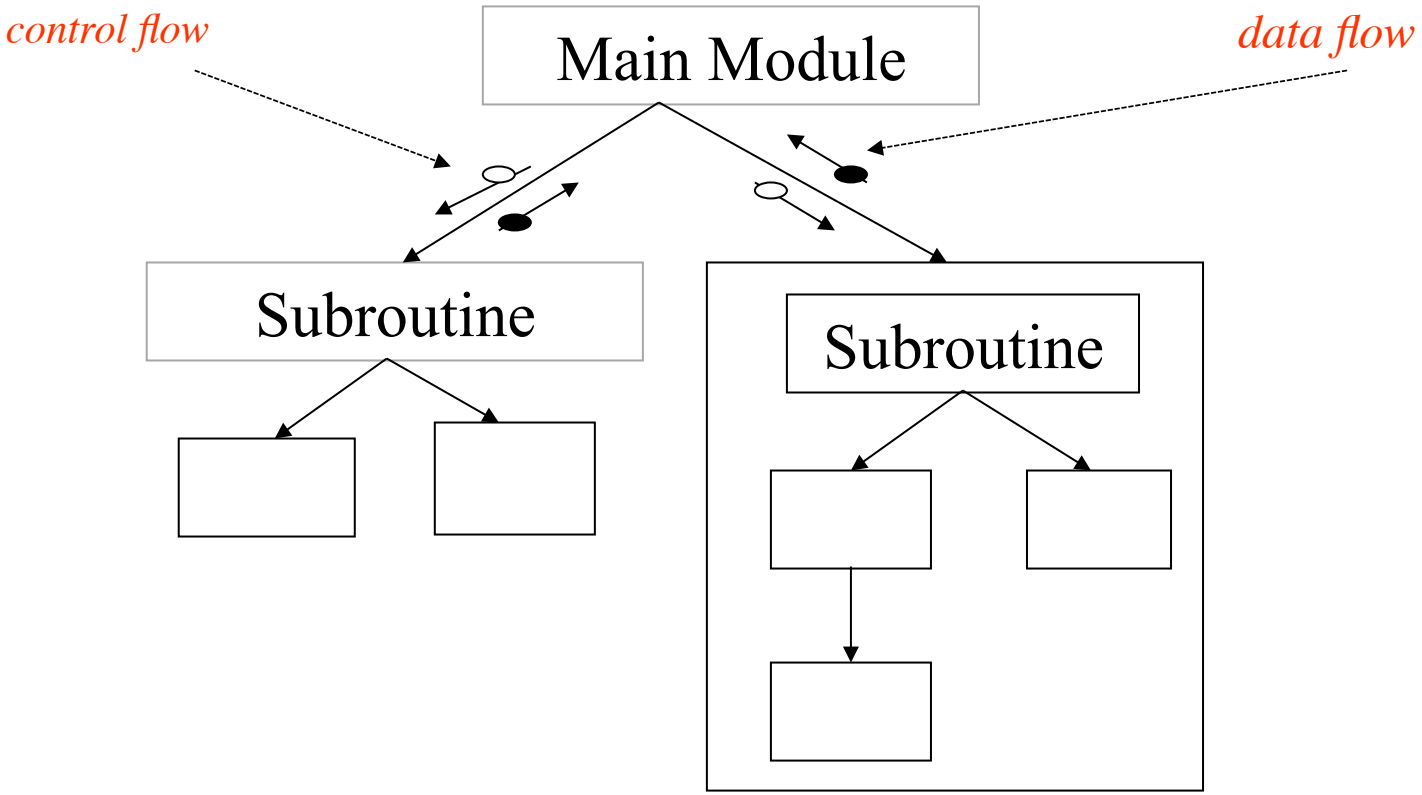
1.1. Object Oriented Properties

- It is possible to change the implementation of object without affecting the clients using ADT and encapsulation
- Supports
  - Modularity
  - Modifiability
  - Reusability
- Designers can decompose problems into collections of interacting objects
- Promotes reuse and modifiability because it supports separation of concerns
- Access to the object is allowed only through provided methods
- The disadvantage is that one object must know the identity of the other object (reference) to communicate

1.2. Properties of Main Program and Subroutines

- Hierarchical decomposition
  - Based on definition and use relationship
- Single thread of control
  - Supported directly by programming languages
  - Each component in the hierarchy gets this control from its parent and passes it along to its children
- Subsystem structure implicit
  - Subroutines typically aggregated into modules
- Hierarchical reasoning
  - Correctness of a subroutine depends on the correctness of the subroutines it calls
- Increase performance
  - By distributing the computations and taking advantage of multiple processors

Structured Charts: Main program and Subroutines



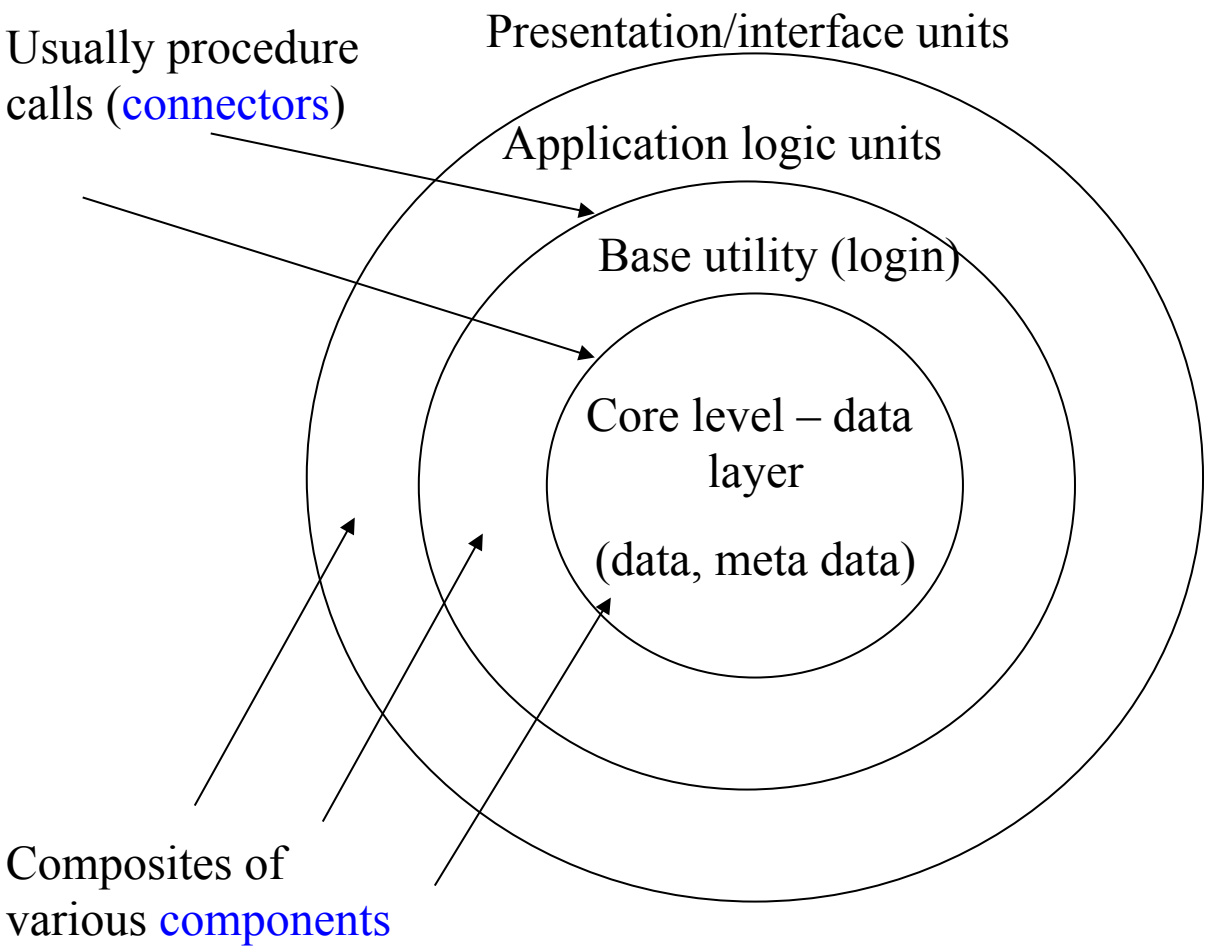
1.3. Layered Style

- A layered system is organised hierarchically
- Each layer only provides service to the layer above it and servicing as a client to the layer below
- Inner layers may be hidden from all except the adjacent outer layer, except for certain selected functions
- Connectors are defined by the protocols that determine how the layers will interact
- Topological constraints include limiting interactions to adjacent layers
- The lowest layer provides some core functionality, such as hardware, or an operating system kernel
- Each successive layer is built on its predecessor, hiding the lower layer
- Example, layered communication protocols, OSI ISO model, X Window System protocols

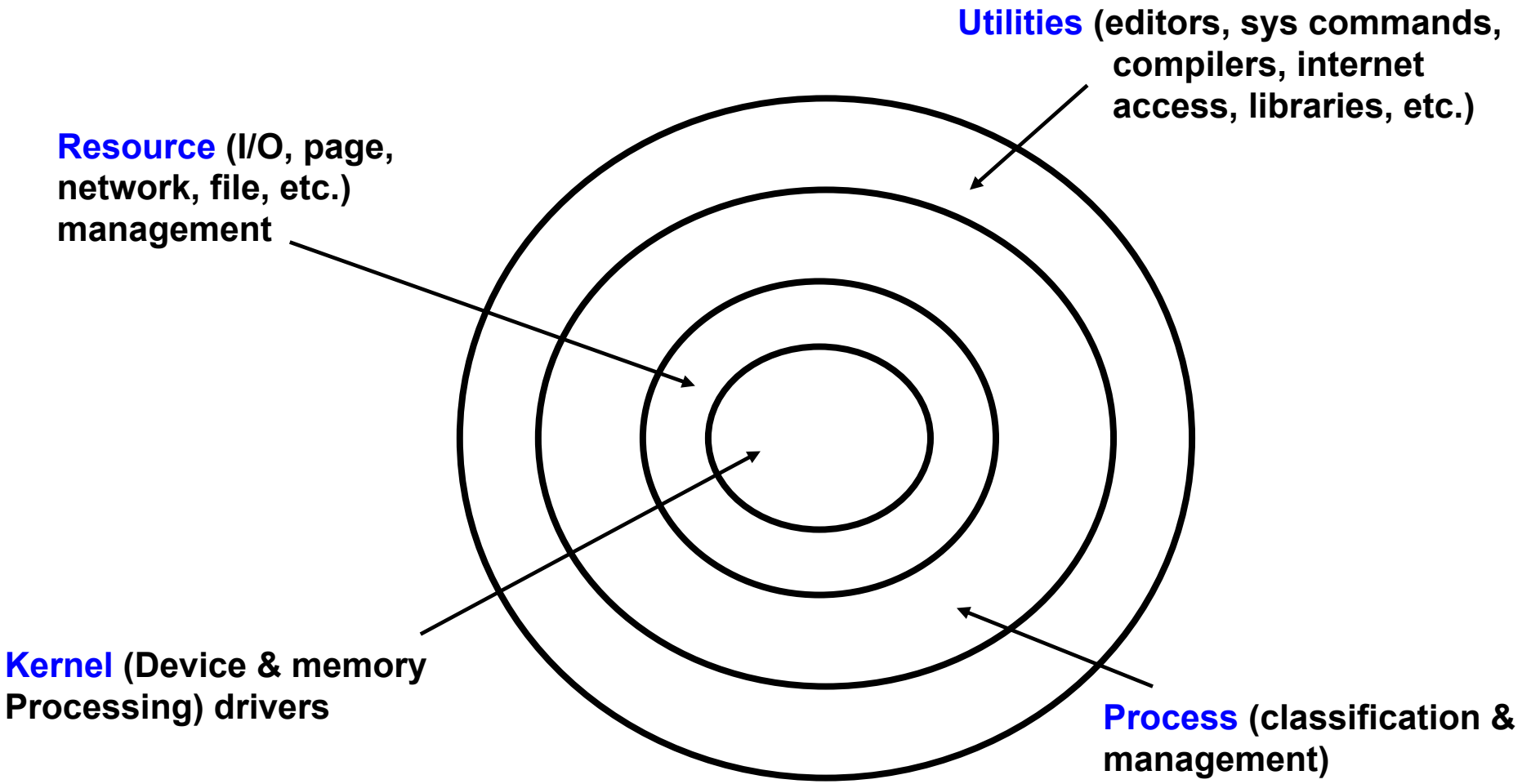
1.3. Layered Properties

- They support designs based on increasing levels of abstraction
- This abstraction allows designer to partition a complex problem into a sequence of incremental steps
- Each layer interacts with at most the layers below and above
- Changes to one layer affect at most two other layers
- Supports security, modifiability, reusability, high cohesion, low coupling, availability, efficient, scalability, and portability
- One major disadvantage is that not all systems are easily structured in a layered fashion
- Closer coupling between logically high-level functions and their lower-level implementations gives inflexibility

1.3. Layered System

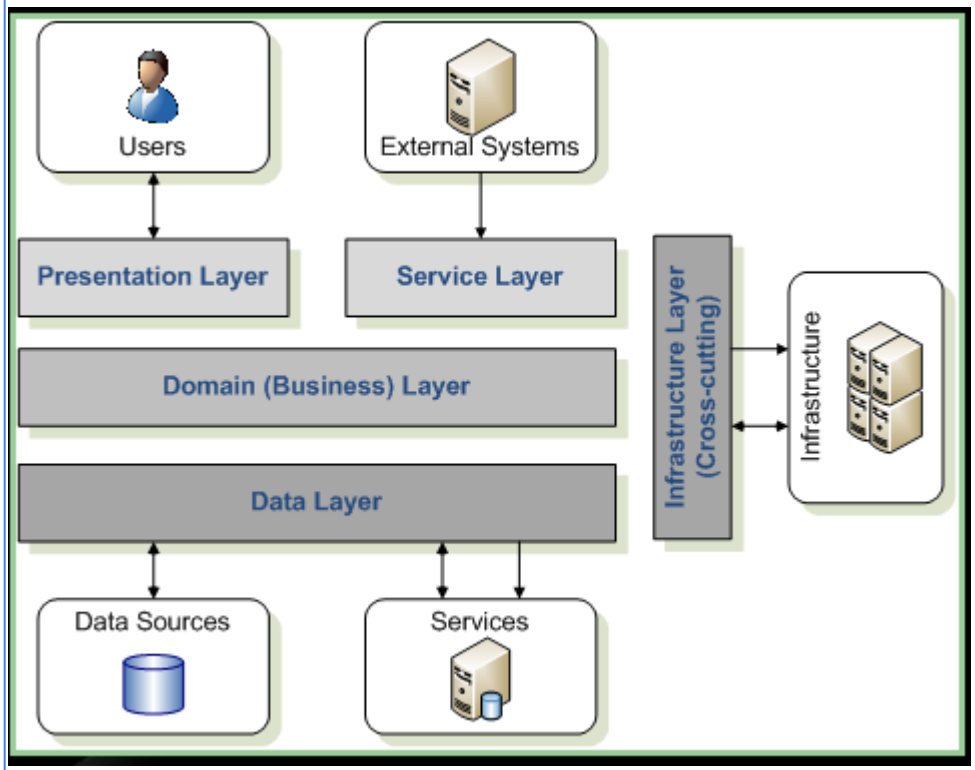


Example - Layered Architecture for Operating System



Layered Architecture

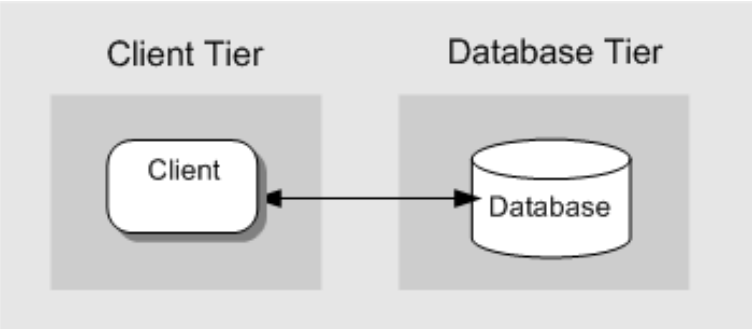
- The high level design solution is decomposed into Layers with a unique role per layer
  - Structurally, each layer provides a *related set* of services
  - Dynamically, each layer may only use the layers below or above it
- **Cross-Cutting Concerns**
  - Isolate domain logic from infrastructure concerns such as Authentication, Authorization, Logging (security)
- Business logic can be used by multiple presentations as well as the service layer
- Internal structure can be modified without disrupting other layers if the interface remains same. (modifiability)
- A layer can be replaced by another new layer without affecting the system (portability)
- A layer can be used in another architecture as long as the interface remains consistent with the new system (reusability)

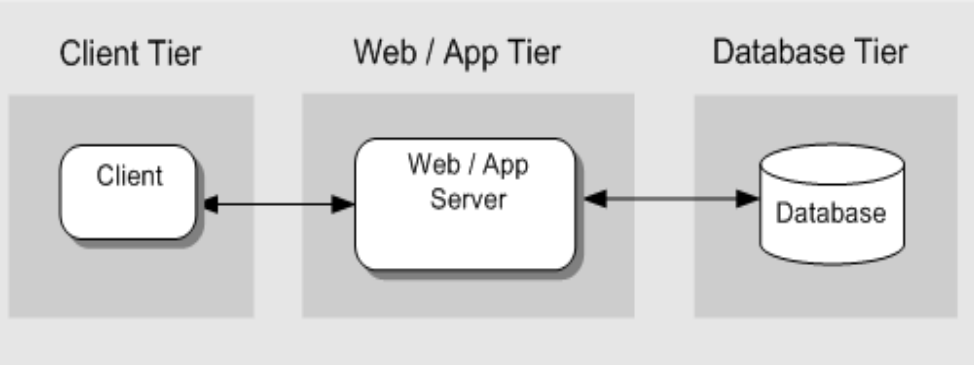


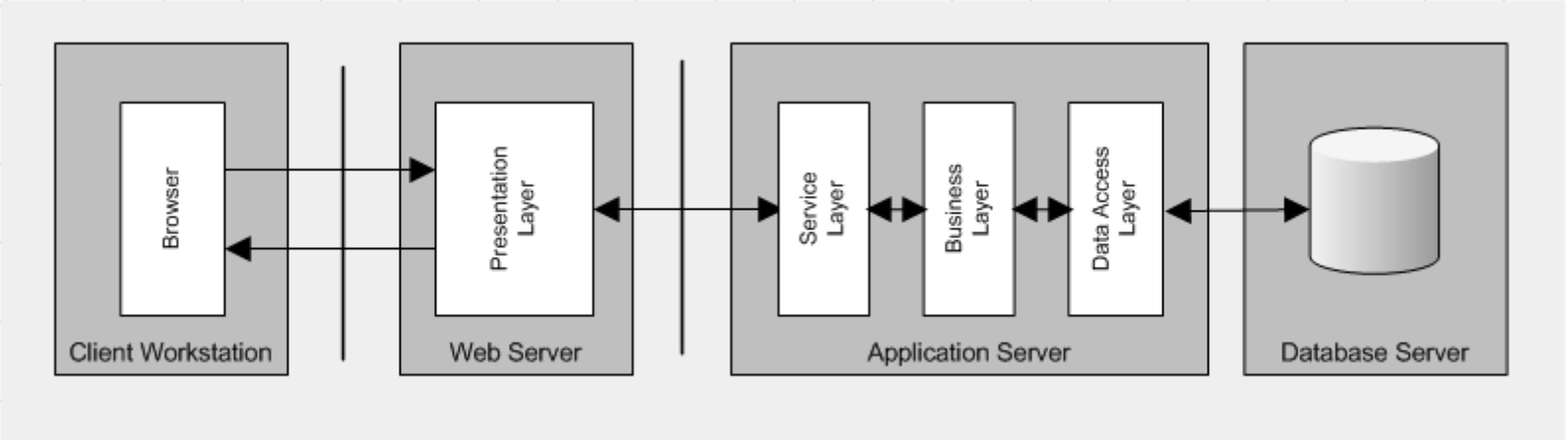
Deployment Patterns: Tiers (2-Tier, 3-Tier, N-Tier)

Layered Architecture provides flexible deployment

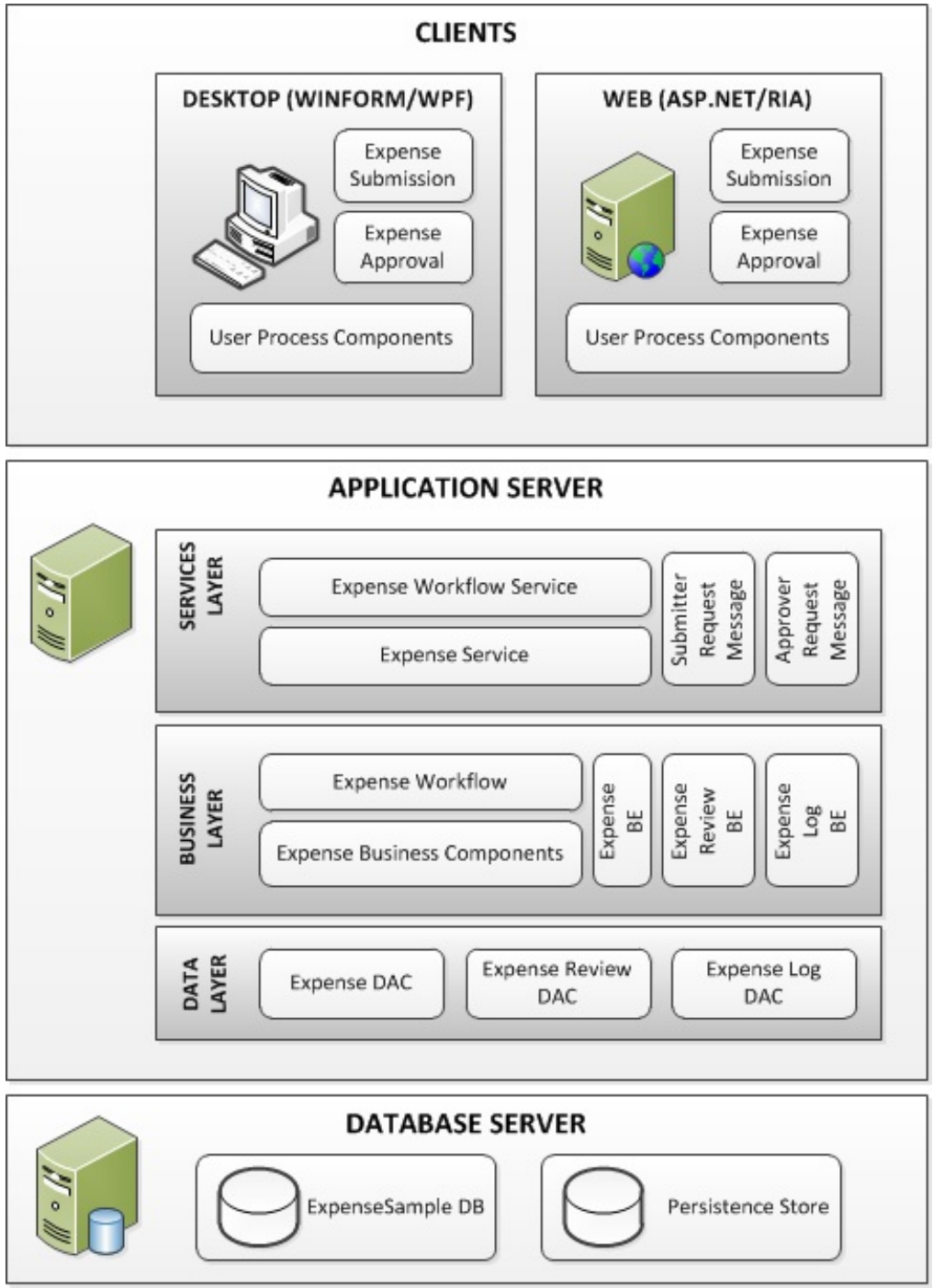
- There are no restrictions on how a multi-layer application is deployed
- All layers could run on the same machine or each tier may be deployed on its own machine



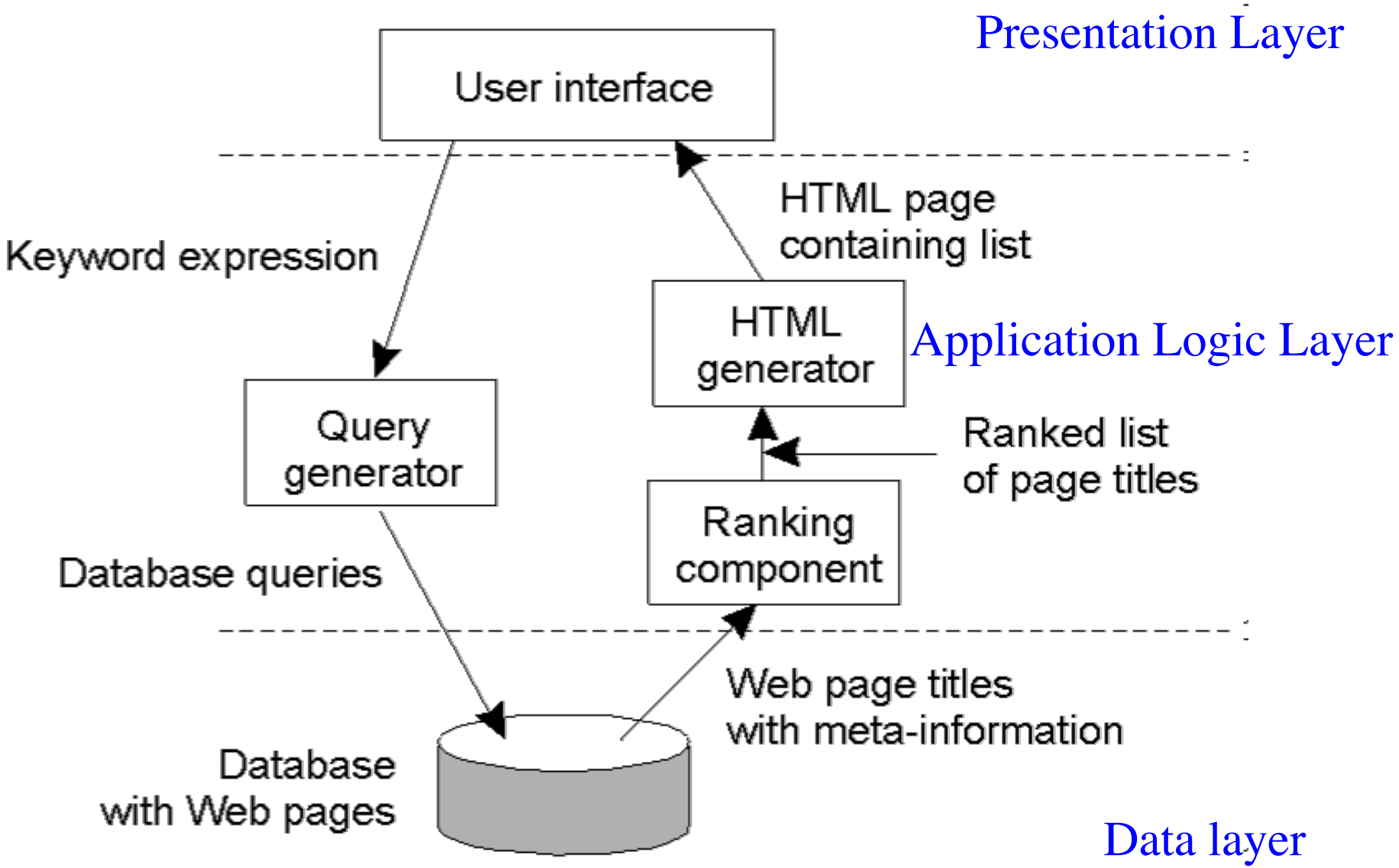




Layered Architecture Example



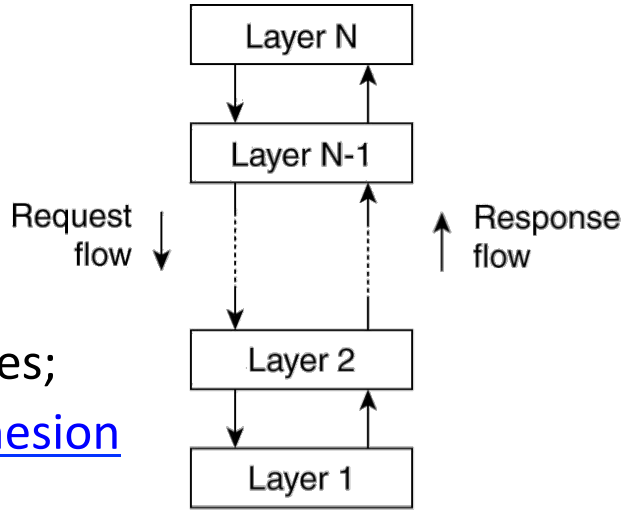
Example - Internet search engine



Rules of Thumb for Choosing an Architectural Style: Call-and-Return Style

- Call-and-Return: The order of computation is fixed, and components can make no useful progress while awaiting the results of requests to other components
  - **Main Program Subroutine**
    - Modifiability with respect to the production of data and how it is consumed is important
  - **Object-Oriented**
    - Overall modifiability is a driving quality requirement
    - Data types whose representation is likely to change
    - Modules whose development time and testing time could benefit from exploiting the commonalities through inheritance
  - **Layered**
    - The tasks of the system can be divided between those specific to application and those generic to many applications but specific to the underlying computing platform
    - Portability across computing platforms is important
    - Can use an already-developed computing infrastructure layer (e.g., O/S, network management package)

Advantages and Disadvantages of Layered Architecture



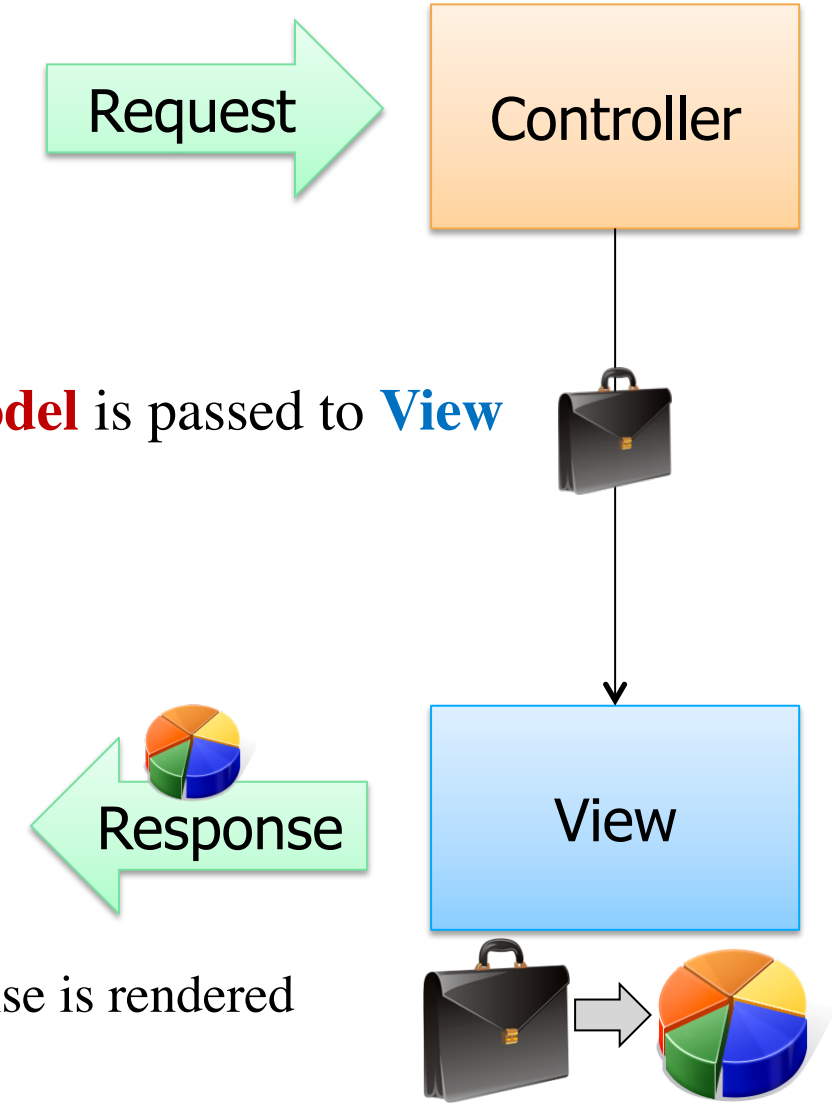
- Advantages:
  - Each layer is selected to be a set of related services; thus the architecture provides high degree of cohesion within the layer
  - Each layer hides complexity from other layers
  - Layers may use only lower layers hence reducing coupling
  - Each layer, being cohesive and is coupled only to lower layers, makes it easier for reuse and easier to be replaced (modifiability)
  - *Flexible deployment:* all layers could run on the same machine, or each tier may be deployed on its own machine (portability)
- Disadvantages:
  - Layered Style **may** cause performance problem depending on the number of layers

Layered Architecture – Quality Attribute Analysis

Quality Attribute	Issues
Availability	• Servers in **each tier can be replicated**, so that if one fails, others remain available • This means a client request is, without its knowledge, redirected to a live replica server that can satisfy the request • Overall the application will provide a lower quality of service until the failed server is restored
Modifiability	• **Separation of concerns enhances modifiability**, as the presentation, application, and data layers are encapsulated • Each can have its internal logic modified in many cases without changes rippling into other tiers
Efficient	• This architecture has **proven high performance** • Key issues to consider are the speed of connections between tiers and the amount of data that is transferred • As always with distributed systems, it makes sense to minimize the calls needed between tiers to fulfill each request
Scalability	• As servers in **each tier can be replicated,** the architecture scales well • In practice, the data management tier often becomes a bottleneck on the capacity of a system

The MVC pattern is intended to allow each part to be changed independently of the others

Model-View-Controller (MVC)



Controller

- Incoming request directed to **Controller**
- A controller accepts input from the user and instructs the model to perform actions based on that input
- e.g. the controller adds an item to the user's shopping cart
- Model is then passed to the View

View

- View transforms Model into appropriate output format

Model-View-Controller Architecture (MVC)

Model

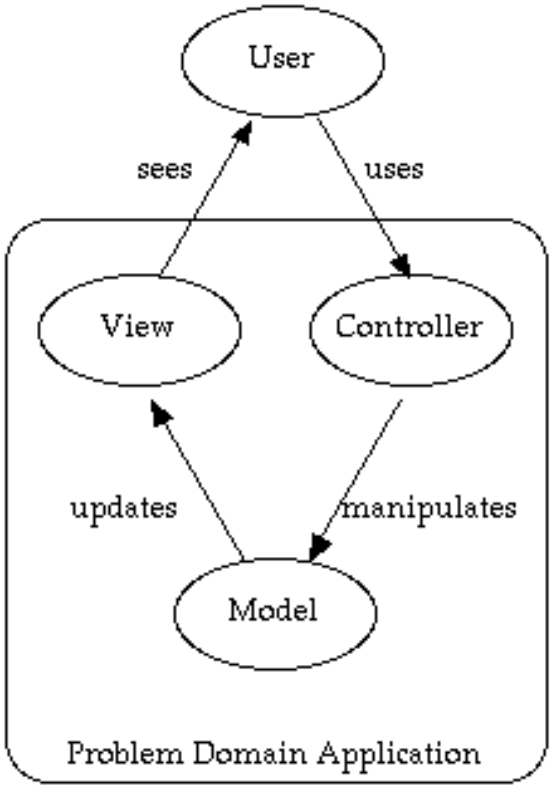
- Holds the application data and implements the application logic
- Know how to carry out specific tasks such as processing a new subscription

View

- View provides a visual representation of the model

Controller

- Handles the user input (mouse movement, clicks, keystrokes, etc.)
- Process data and communicate with the Model to save state (e.g., delete row, insert row)
- Coordination logic is placed in the controllers

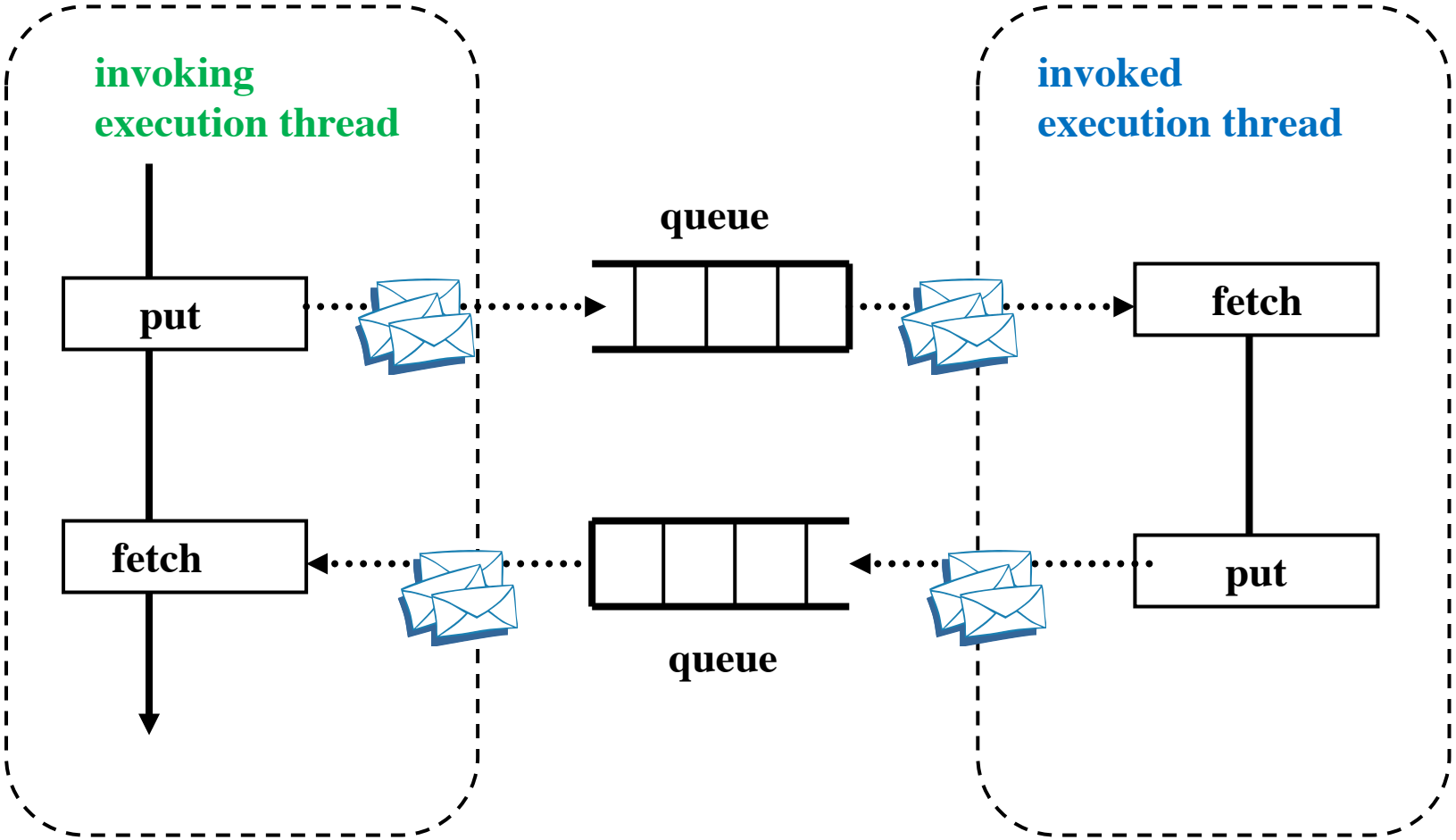


Advantages of MVC

- **Separation of concerns**
  - Views, controller, and model are separate components. This allows modification and change in each component without significantly disturbing the other.
    - Computation is not intermixed with Presentation
    - Consequently, code is cleaner and easier to understand and change
- **Flexibility**
  - The view component, which often needs changes and updates to keep the users continued interests, is separate
    - The UI can be completely changed without touching the model in any way
- **Reusability**
  - The same model can used by different views (e.g., Web view and mobile view)
- **Disadvantages:**
  - Heavily dependent on a framework and tools that support the MVC architecture (e.g., ASP.Net MVC, Ruby on Rails)

MVC is widely used and recommended particularly for interactive web-applications.

2. Independent Component Style: Message Passing - Message Oriented Middleware (MOM)

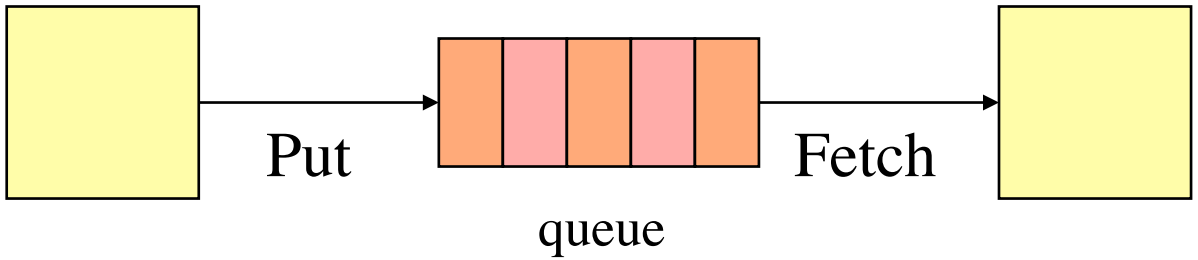


Aimed at achieving *decoupling and reliability*

= Messages

2.1. Implicit Invocation Style

- Usually facilitated by a Message Oriented Middleware (MOM)
  - **Put** (queue, message) – Write message onto queue
  - **Fetch** (queue, message) – Read message from queue
- Sender places a message in a queue instead of method invocation
  - Listeners read message from queue and process it



MOM Advantages and Disadvantages

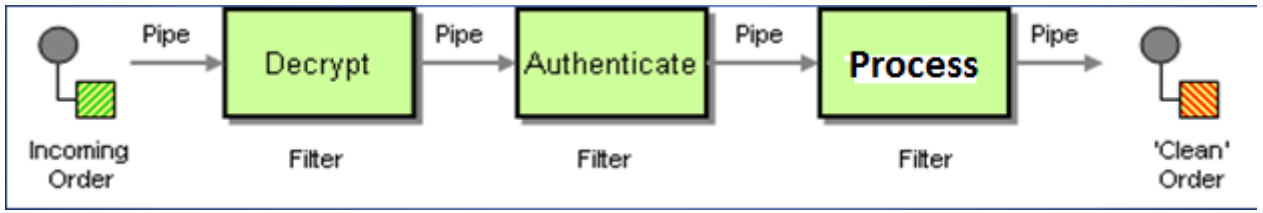
- **Advantages**
  - **Lower coupling between components**: the message senders and the message processors are separate
    - Easier system evolution: e.g., a component can be easily replaced by another one
    - Any sender or processor malfunction will not affect the other senders and message processors
  - Higher component reuse
- **Disadvantages**
  - MOM malfunction will bring the whole system down
  - MOM can be a single point of failure
  - Lower system understandability
    - No knowledge of what components will respond to event
    - No knowledge of order of responses

Messaging – Quality Attribute Analysis

Quality Attribute	Issues
Availability	• Physical queues with the same logical name can be replicated across different messaging server instances. • When one fails, clients can send messages to replica queues.
Modifiability	• Messaging is inherently loosely coupled, and this promotes high modifiability as clients and servers are **not directly bound through an interface**. • Changes to the format of messages sent by clients may cause changes to the server implementations => dependency on message formats
Performance	• Message queuing technology can deliver thousands of messages per second.
Scalability	• Queues can be hosted on the communicating endpoints, or be replicated across clusters of messaging servers hosted on multiple server machines. • This makes messaging a highly scalable solution.

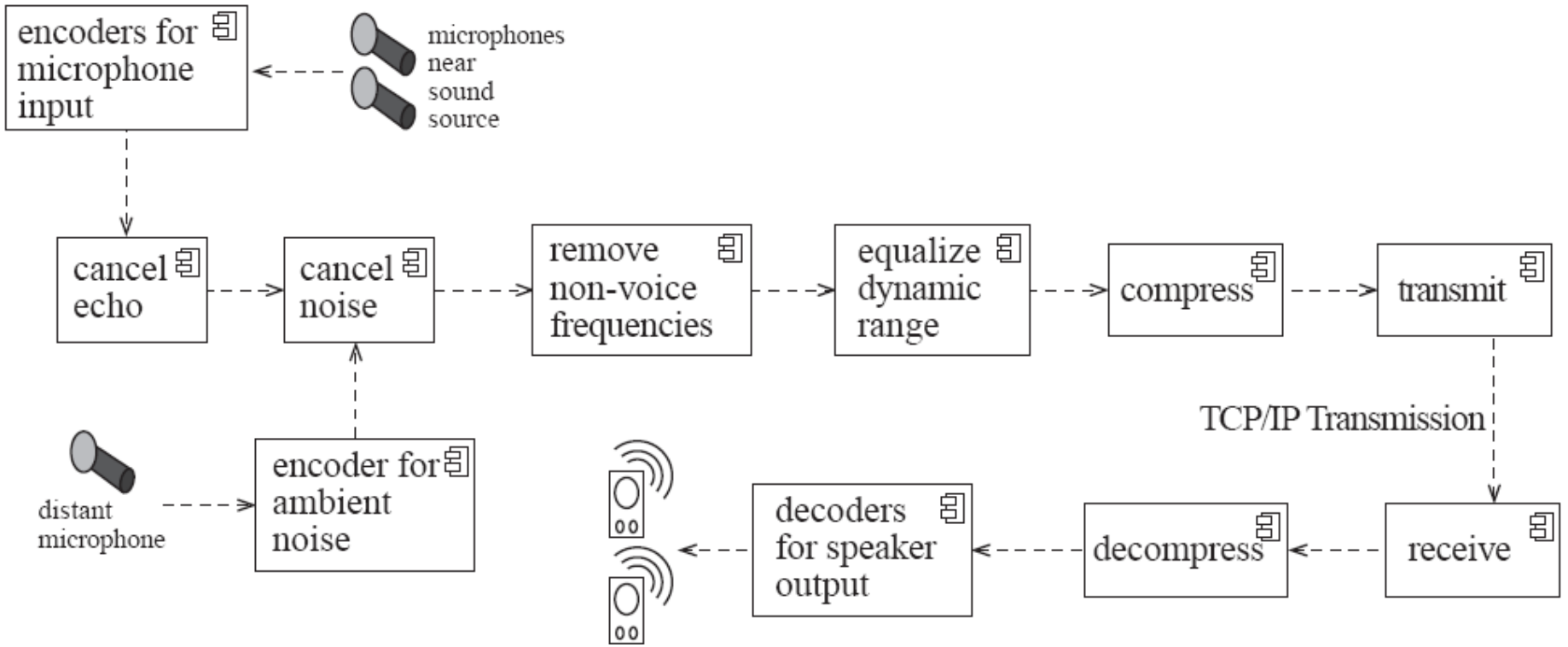
3.1. Pipe and Filter : Data Flow Style

- **The software is decomposed into filters and pipes**
  - Filter (component) is a service that transforms a stream of input data into a stream of output data
  - Pipe (connector) is a mechanism through which the data flows from one filter to another
  - Allows developer to divide larger processing tasks into smaller, independent tasks
- Components are filters and Connectors are pipes
- **Examples:** UNIX shell, Signal processing



Problems that require batch file processing seem to fit this e.g., payroll and compilers

Example of a Pipe-and-Filter: Data Flow Style

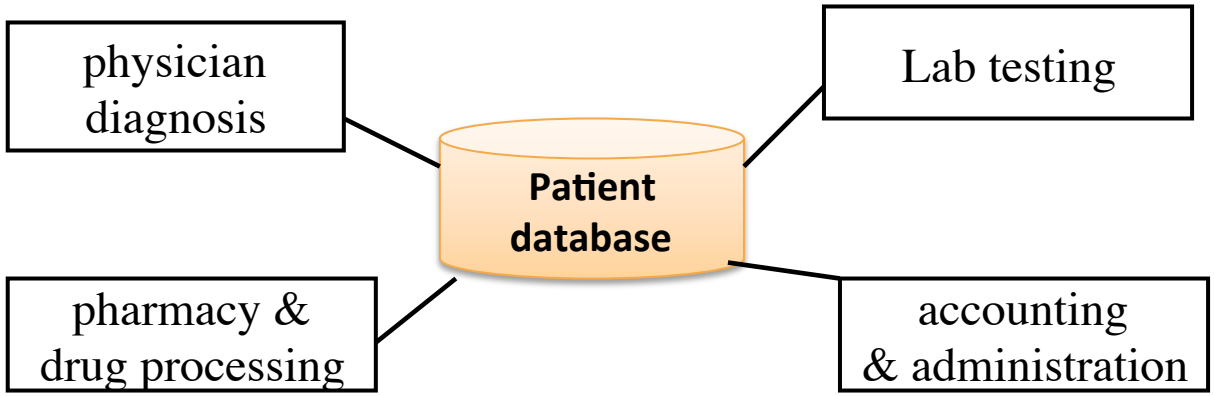


Advantages and Disadvantages of Pipe-Filter

- **Advantages:**
  - Filters are self containing processing service that performs a specific function thus it is fairly cohesive
  - Easier filter addition, replacement, and reuse
  - Filters communicate (pass data most of the time) through pipes only, thus it is constrained in coupling
- **Disadvantages:**
  - Filters processes and sends streams of data over pipes is a solution that fits well with heavy batch processing, but may not do well with any kind of user-interaction.

5.1. Shared Data Storage: Data-Centric Style

- The high level design solution is based on a *shared data storage* which acts as the "central command" with two variations:
  - ***Blackboard style***: the shared data storage **alerts** the participating parties whenever there is a data-store change
  - ***Repository style***: the participating parties **check** the data-store for changes



Problems that fit this style such as *patient processing*, *tax processing system*, *inventory control system*; etc. have the following properties:

1. All the functionalities work off a single data-store
2. Any change to the data-store may affect all or some of the functions
3. All the functionalities need the information from the data-store

Very Common in Business where data is central

5.2. Blackboard: Data Centric Style

- A blackboard sends notification to subscribers when data of interest changes, and is this active
- It is sometimes refereed as active repository
- Many systems, especially those built from pre-existing components, are achieving data integration through the use of blackboard mechanisms
- Data store is independent of the clients, thus, this style is scalable; new clients can easily be added
- It is also modifiable with respect to changing the functionality of any particular client because other clients will not be affected.

5.3. Repository : Data Centric Style

- In a repository style there are two quite distinct kinds of components:
  - A central data base represents the current state, and
  - Independent systems operate on the central data base
    - Global flight reservation system
- Classical database
  - Central repository has schemas designed for specific application
  - Independent operators
    - Operations on database implemented independently, one operation per transaction type
    - Interact with database by queries and updates
      - Global shipping traffic positioning system

Advantages and Disadvantages of Data Centric Style

- **Advantages**
  - Higher component cohesion and low coupling: the **coupling is restricted to the shared data**
  - Single data-store makes the maintenance of data in terms of back-up recovery and security easier to manage
- **Disadvantages**
  - **High coupling to shared data:** any data format change in the shared data requires agreement and, potentially, changes in all or some the functional areas
  - **Data store can become a single point of failure**: if the data-store fails, all parties are affected and possibly all functions have to stop (may need to have redundant database for this architecture style; also, should have good back up- and recovery procedures)

References

- R. Pressman: Software Engineering: A practitioner's approach
- M. Shaw and D. Garlan: Software Architecture: Perspectives on an emerging discipline
- L. Bass and P. Clements: Software Architecture in Practice
- M. Jackson: Principles of Program Design.
- JSP: https://en.wikipedia.org/wiki/Jackson_structured_programming