CMPS411

Spring 2018

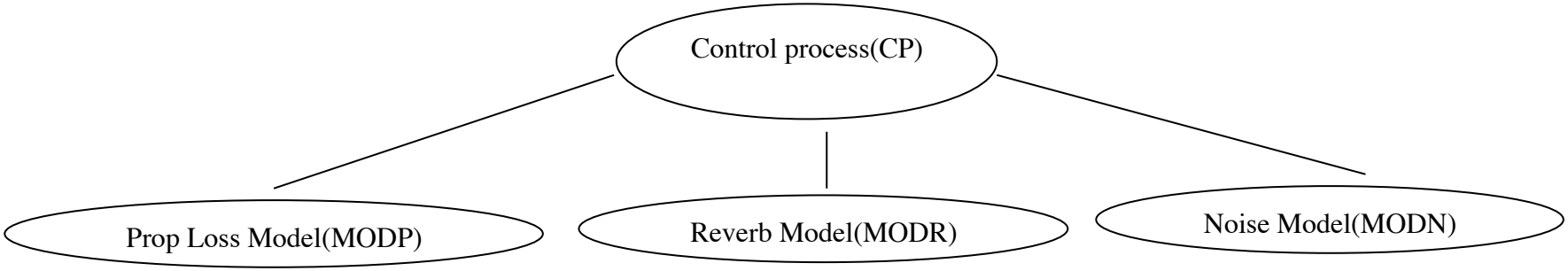
Lecture 11

Software Architectural Styles

Architectural Styles or Architectural Patterns

- Architectural styles or patterns, architectural patterns, and architectural idiom are interchangeable
- An architectural style in software is analogous to an architectural style in buildings
  - Gothic or Greek Revival or Queen Anne
- An architectural style is determined by the following:
  - **A set of component types** that perform some functions in runtime
  - **A topological layout of these components** indicating their runtime interrelationships
  - **A set of semantic constraints or rules**
  - **A set of interaction mechanisms using connectors** that mediate communication, coordination, or cooperation among components
- Thus, a style is not an architecture any more than the term Gothic determines exactly what a building looks like
- A style defines a class of architectures –it is an abstraction for a set of architectures that meet it.

What is Architecture and What is not



- The system consists of four components (elements)
- Lower level three elements might have more in common with each other than the top component (element) CP
- All elements apparently have some sort of relationship with each other.

Is this an architecture?

- What is the nature of the elements? What is the significant of their separation?
- What are the responsibilities of the elements? What is it they do?
- What is the significance of the connections? What are the mechanisms for the communications?
- What is the significance of the layout? Why CP is on a separate level?

Software Architecture: Formal definition

Software architecture is the:

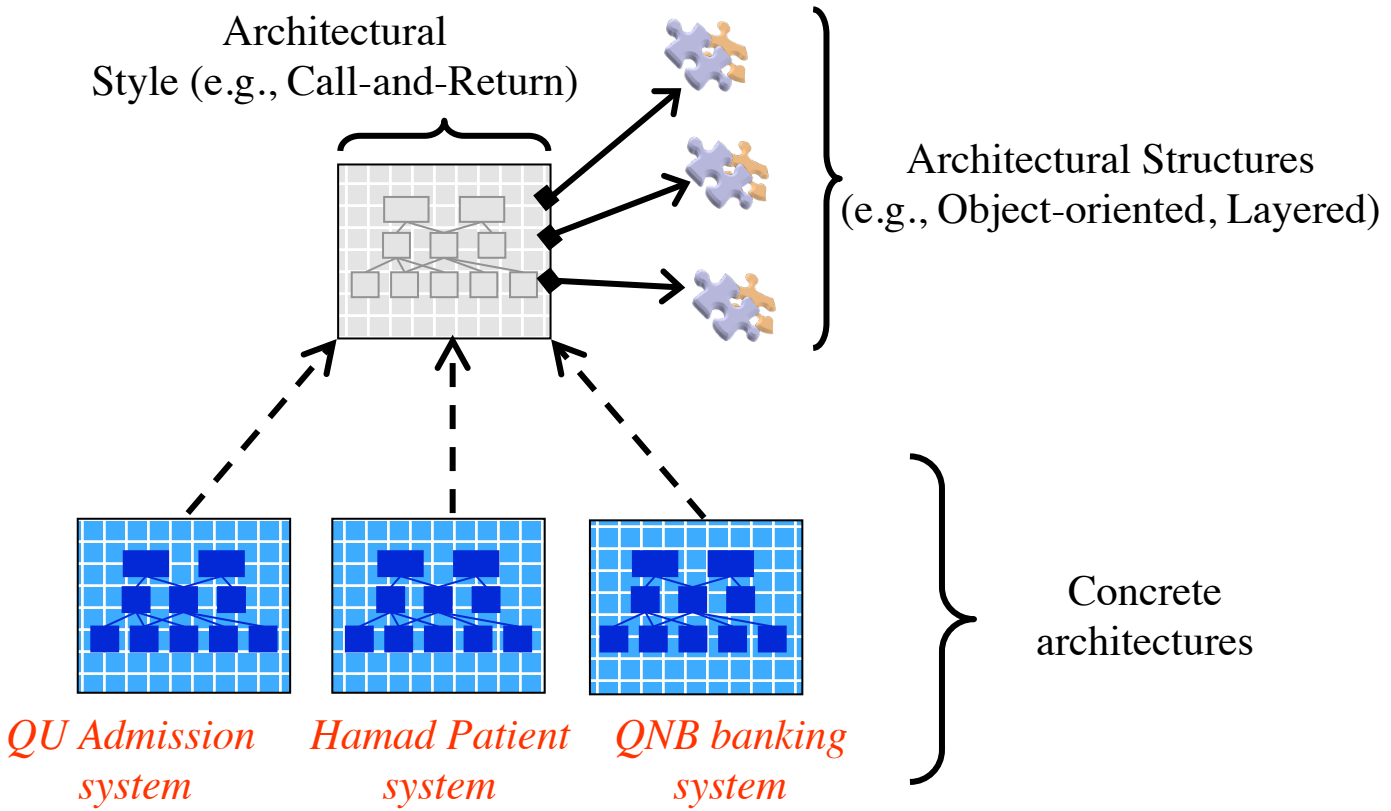
- Organization of a software system
- Selection of the structural elements (components) and their interfaces for composition together with their behaviour
- Composition (using connectors) of these structural and behavioural elements

[Booch et al. 1999]

- Fundamental organization of a system embodied by its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

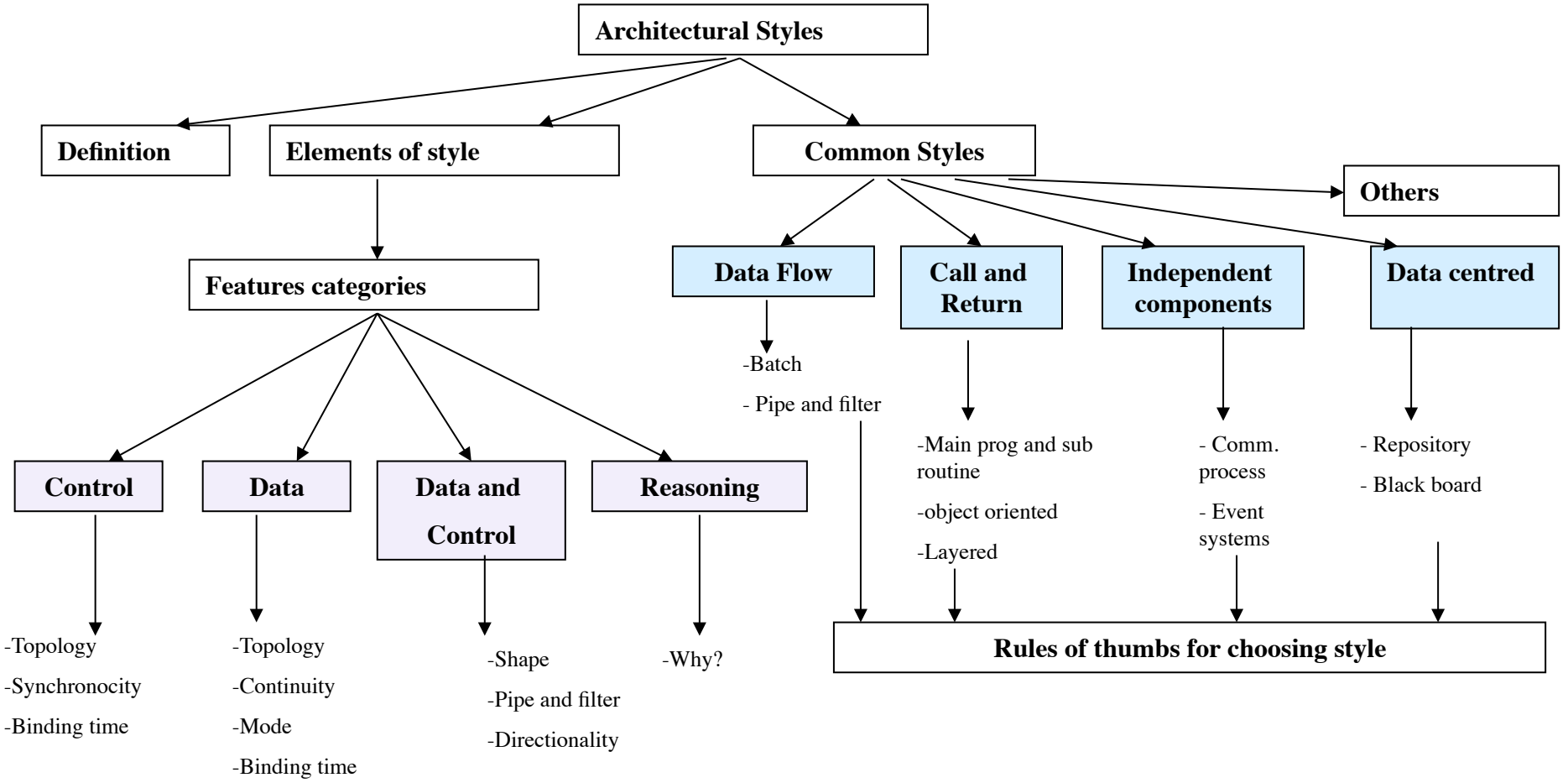
[IEEE Standard P1471]

Architectural Styles-Concrete Architecture-Architectural Structures



- Benefits of architectural styles and patterns
  - Allow for reuse
  - Provide a common architectural vocabulary
  - Quality achievement.

Types and Elements of Architectural Styles

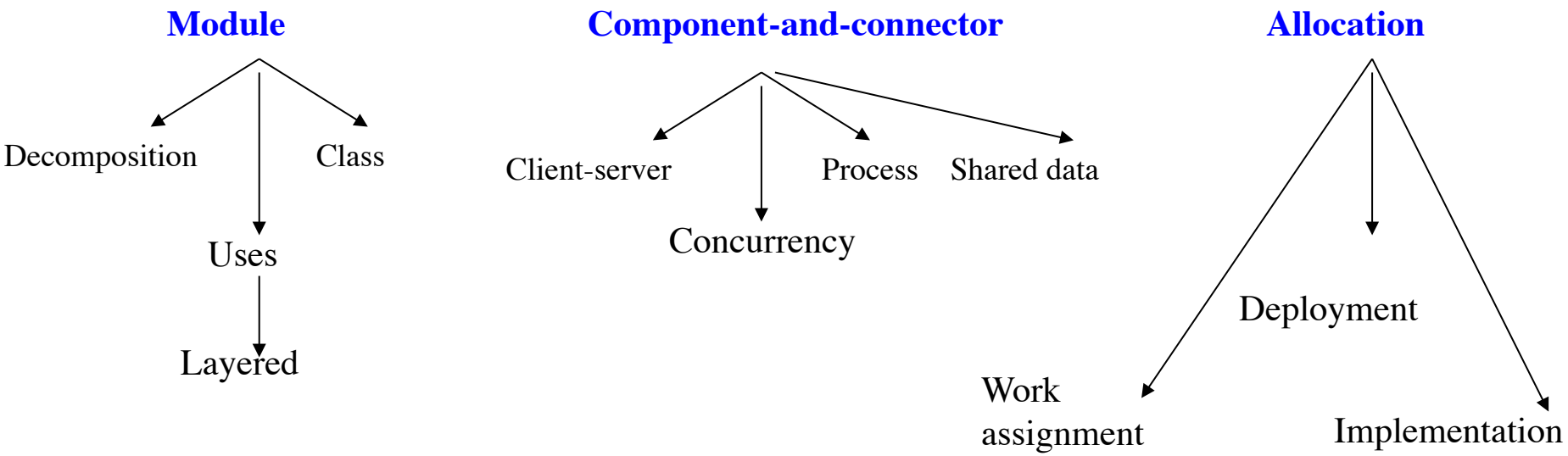


Architectural Structures

- **Module structures**: elements (components) are modules, which are units of implementation.
  - Modules are assigned specific functional responsibility.
- **Component-and-connector structures**:
  - runtime components (units of computation), and
  - connectors (communication vehicles among components).
- **Allocation structures**: Show the relationship between the software elements and the elements in one or more external
- **These three structures correspond to the three broad types of decision that architectural design involves:**
  - How is the system to be structured as a set of code units (**module structures**)
  - How is the system to be structured as a set of elements that have runtime behaviour (**components structures**) and interactions(**connectors structures**)
  - How is the system to relate to non-software structures in its environment (**allocation structures**)

Architectural Styles and Structures: The relationship

- Software architecture is about **software structures**
- Architectural styles can be expressed by using different software structures



Architectural Elements

- A system designer primarily focuses on the **components** and interactions among them using **connectors:**
- **Components**: Examples: 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**: 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

Detail of Feature Categories of Styles

- **Control Issue**: How **control** is shared, allocated, and transferred among components
  - describe how control passes among components
  - how components work together temporarily
    - Topology
    - Synchronocity
    - Binding time
- **Data Issue**: How **data** is communicated through the system
  - describe how data move around a system
    - Topology
    - Continuity
    - Mode
    - Binding time
- **Data and Control Issue**: How **data and control** interact
  - describe the relationship between certain control and data issues
    - Shape
    - Directionality
  - The **kinds of components and connectors** that are used in the style
- **Reasoning issue**: The type of **reasoning** the style permits.

Control Issues: Topology

- What geometric form does the control flow for the system take?
  - a **pipeline style** often has a linear (non-branching) or at least an acyclic control topology
  - a **main-program-and-subroutine** style features a hierarchical (tree-shaped) topology
  - **some server** systems have star (hub-and-spoke) topologies
  - An object-based or event-based style may have arbitrary topology.
- Within each topology it may show the direction in which control flows
- The topology may be static or dynamic depending on the state of component.
- It can be deterministic or not.

Control Issue: Synchronocity

- How dependent are the components' actions upon each others' control states?
  - In **synchronous systems**, components synchronise regularly, and
  - **In asynchronous systems,** components are largely unpredictable in their interaction or synchronous once in a while.

Control Issue: Binding Time

- When is the identity of a partner in a transfer-of-control operation established?
- Some control transfers are predetermined at program
  - Write-time: hard-coded binding between components in the source code
  - Compile-time: binding between components takes place during the compile time, or
  - Invocation/binding-time: binding between components is established only at run time of the system. Components are bound dynamically while the system is running.

Data Issues: Topology and Continuity

- **Topology**
  - Describes the geometric shape of the system's data-flow graph
- **Continuity**
  - Describes how continuous is the flow of data throughout the system?
    - a continuous-flow system has fresh data is available at all times
    - a sporadic-flow system has new data generated at discrete times
    - data transfer may be high volume in data-intensive systems
    - data transfer may be low volume in compute-intensive systems

Data Issues: Mode and Binding Time

- **Mode**
  - describes how data is made available throughout the system
  - data may be passed form component to component in an object style
  - data may be shared in a shared memory
  - data can be **copy-out-copy-in** mode if components tend to modify it and reinsert it into the public store
  - data can be broadcast or multicast to specific recipients in some styles
- **Binding time**
  - what is the identity of the partners in a transfer-of-control operation established?

Control/Data Interaction Issues

- **Shape**
  - are the control-flow and data-flow topologies substantially isomorphic, same form, or composition as another ?
- **Directionality**
  - Does control flow in the same direction as data, or the opposite direction in an architecture?
  - In a data-flow systems and pipe-filter system, control and data pass together from component to component
  - In a client-server style, control tends to flow into the servers and data flow into the clients.

Reasoning

Software designers should have justified reasoning to the following issues:

- Why do we use a particular style?
- What is the rational?
- What quality properties are achieved in a particular style?
- How do we achieve specific quality using a specific style?

Some Common Architectural Styles

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

- **Call-and-return:**
  - **main program and subordinate**
  - **object-oriented systems**
  - **Layered approach**
- **Independent components:**
  - **implicit invocation**
  - **communicating processes**
  - explicit Invocation
- **Data flow :**
  - **pipes and filters**
  - batch sequential
- **Virtual machines:**
  - Interpreters
  - Rule-based systems
- **Data-centric systems:**
  - **shared data storage**
  - blackboard
  - **Repository**
- **Other style**
  - **JSD (object-based)**

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

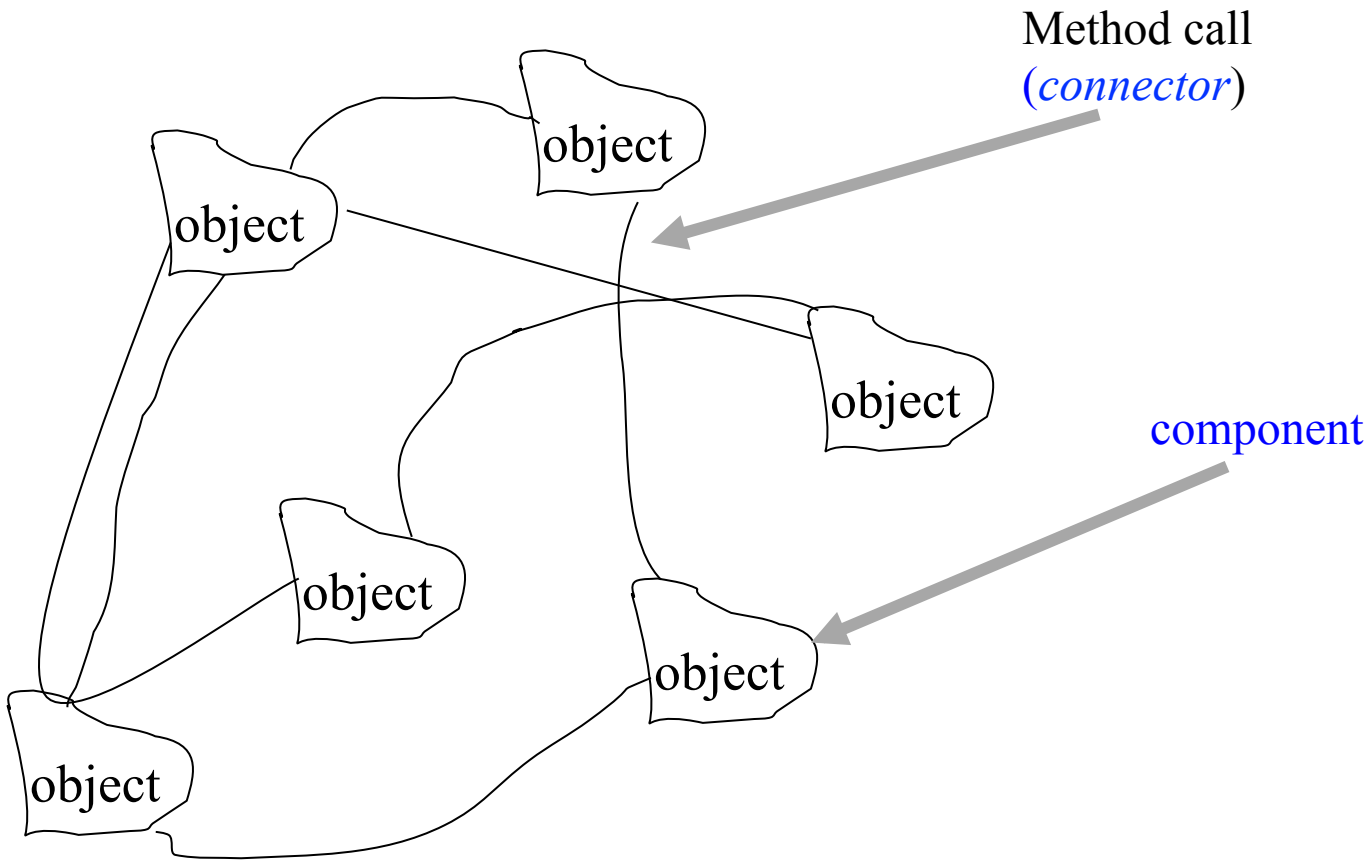
Call and Return Style

- 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:
  - **Object oriented or object-based**
  - **Main program and sub routine**
  - **Layered**

Object Oriented: Call and Return Style

- Based on data abstraction and O-O 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 creation*
- Usually, the topology of O-O is not hierarchical.

Call and Return Style: Non-hierarchical Topology in Object-Oriented Style



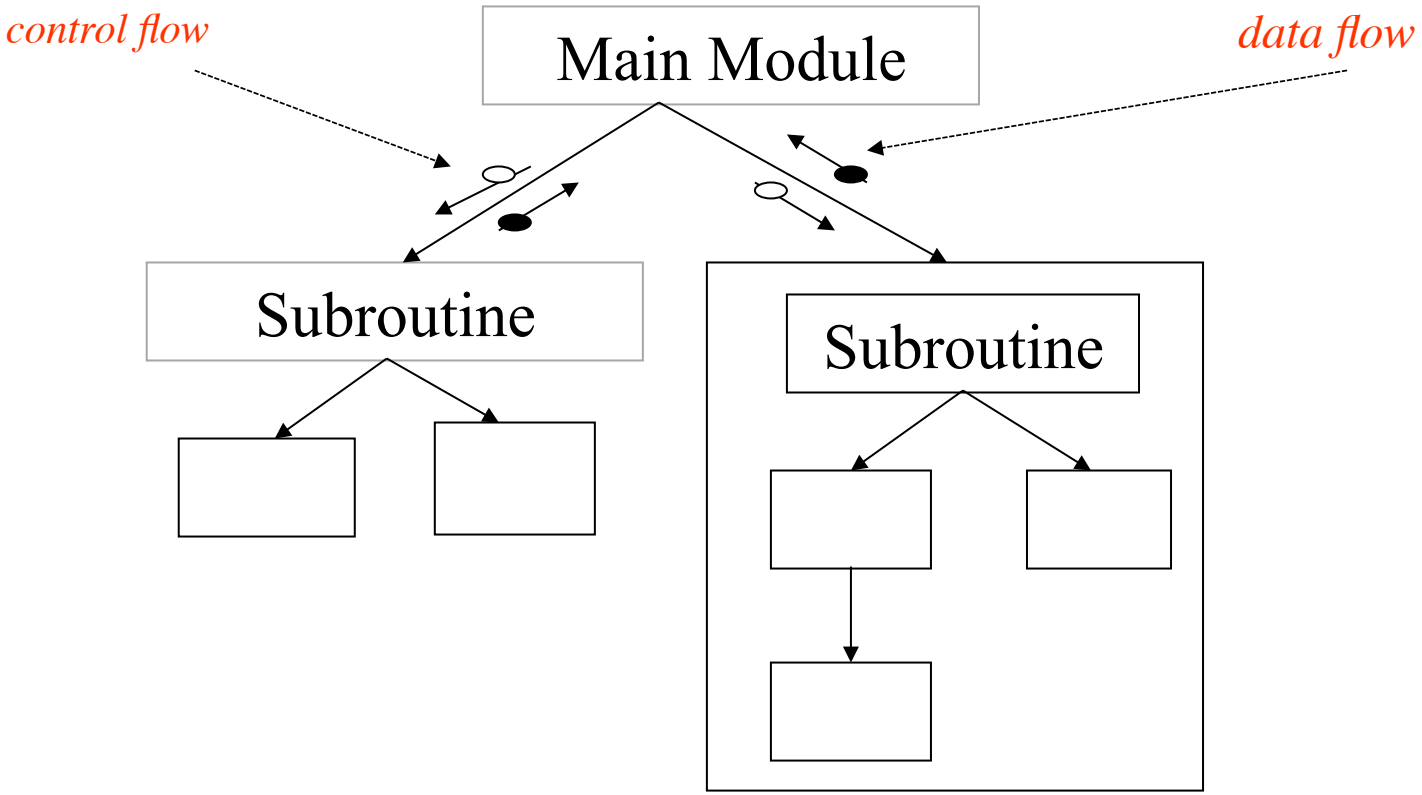
Object Oriented Properties: Call and Return Style

- 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.

Properties of Main Program and Subroutines: Call and Return Style

- 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



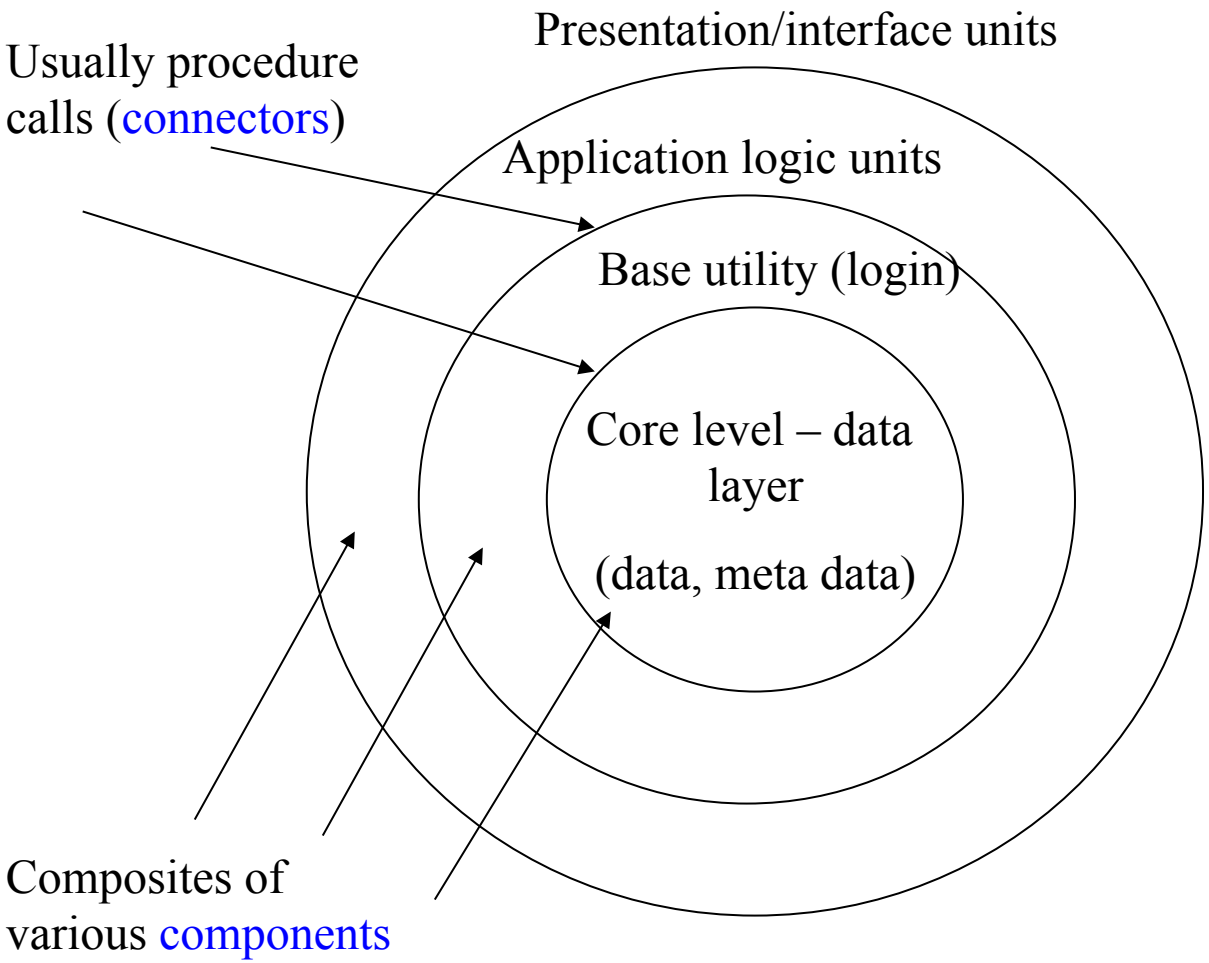
Layered Style: Call and Return Style

- 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

Layered Properties: Call and Return Style

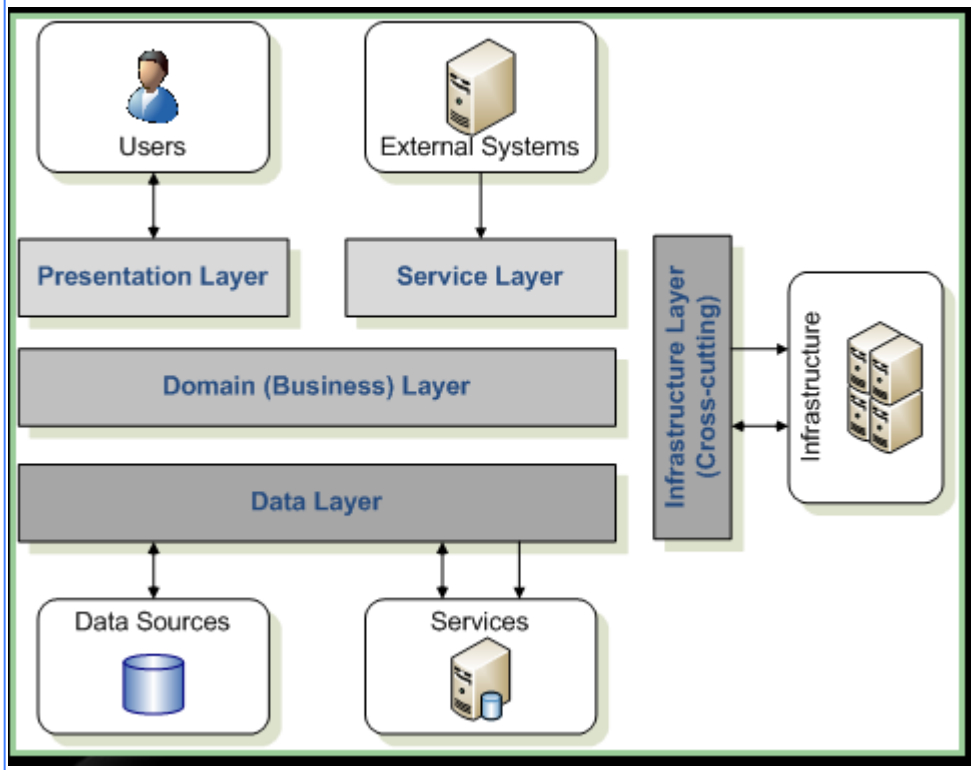
- They support designs based on increasing levels of abstraction
- This abstraction allows designer to partition a complex problem into a sequence of incremental steps
- Each layer interacts with at most the layers below and above
- Changes to one layer affect at most two other layers
- Supports security, modifiability, reusability, high cohesion, low coupling, availability, efficient, scalability, and portability
  - How?
- 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 is gives inflexibility

Layered System: Call and Return Style



Layered Architecture

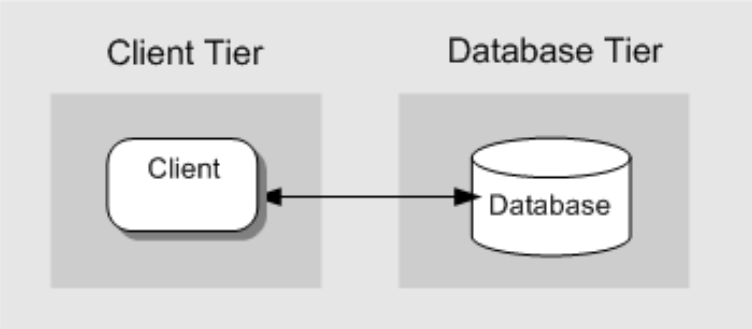
- The high level design solution is decomposed into Layers with a unique role per layer:
  - Structurally, each layer provides a *related set* of services
  - Dynamically, each layer may only use the layers below or above it
- Cross-Cutting Concerns
  - Isolate domain logic from infrastructure concerns 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)

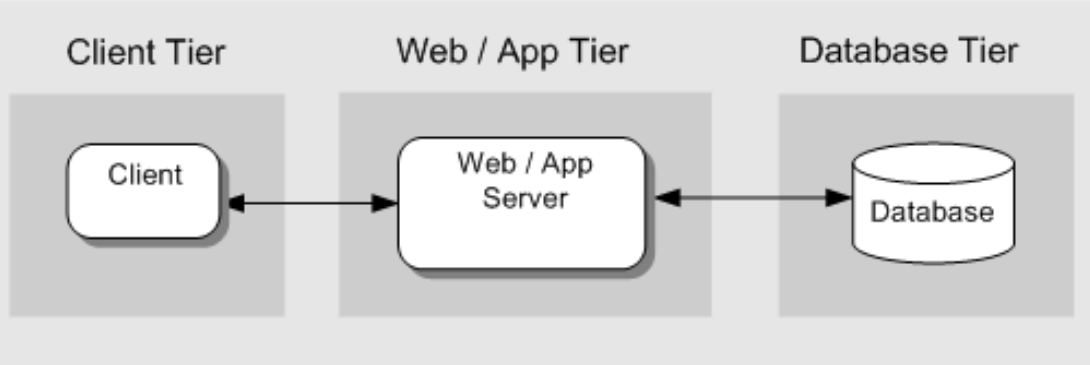


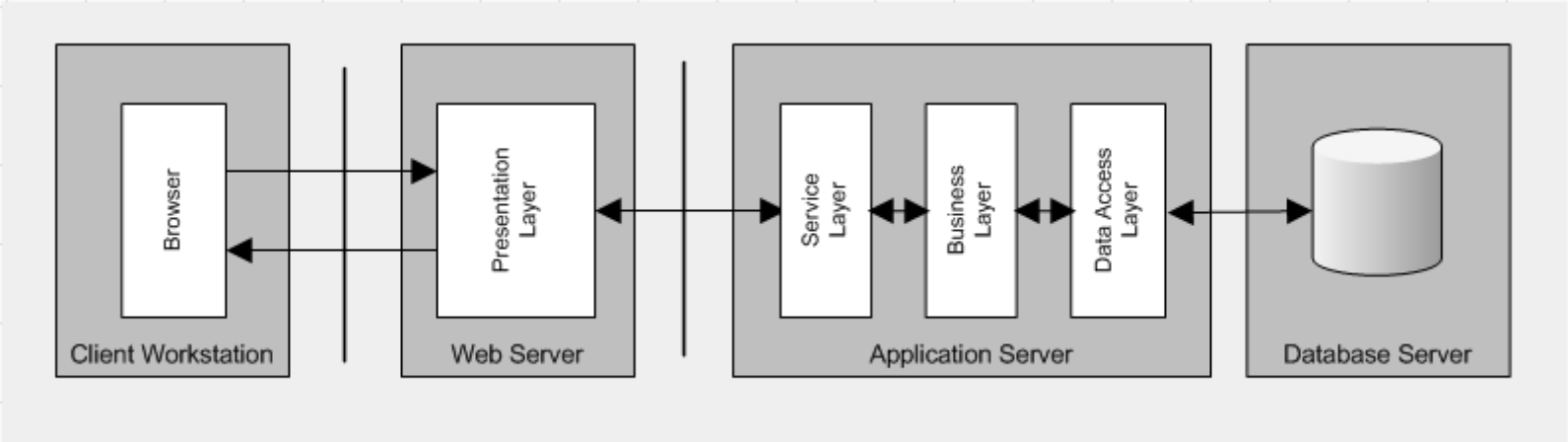
Deployment Patterns: Tiers (2-Tier, 3-Tier, N-Tier)

Layered Architecture provides flexible deployment:

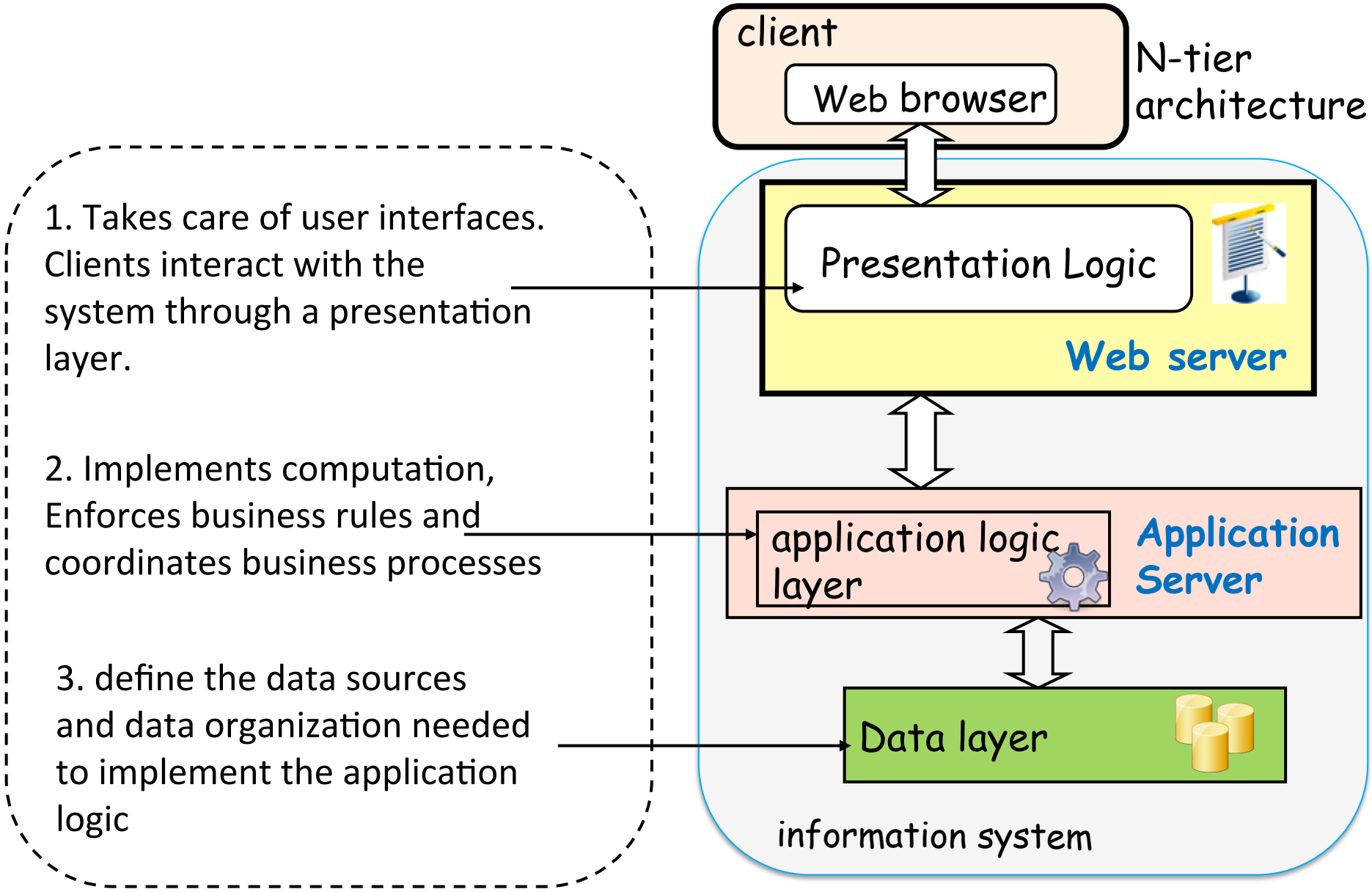
- There are no restrictions on how a multi-layer application is deployed.
- All layers could run on the same machine, or each tier may be deployed on its own machine.



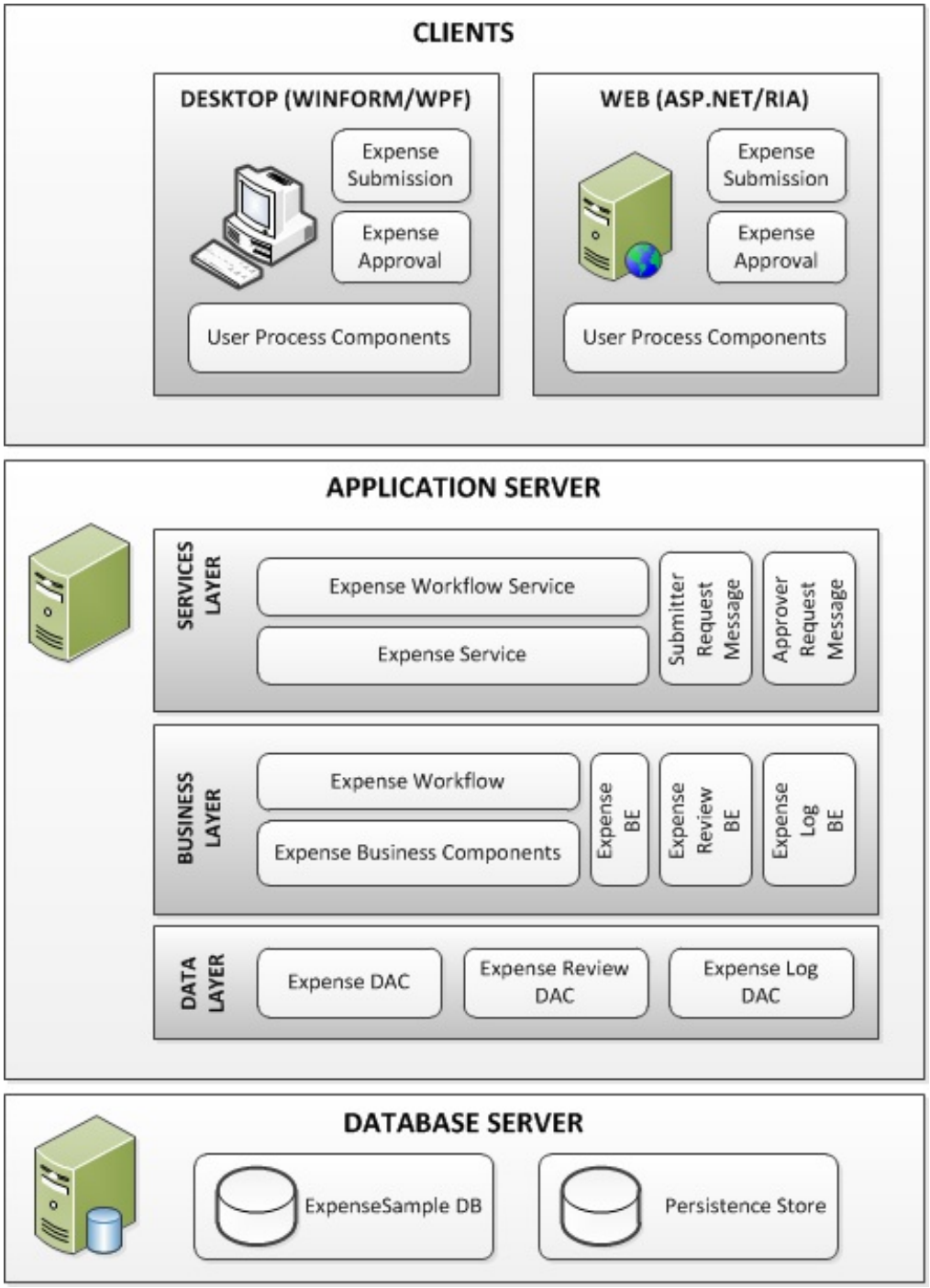




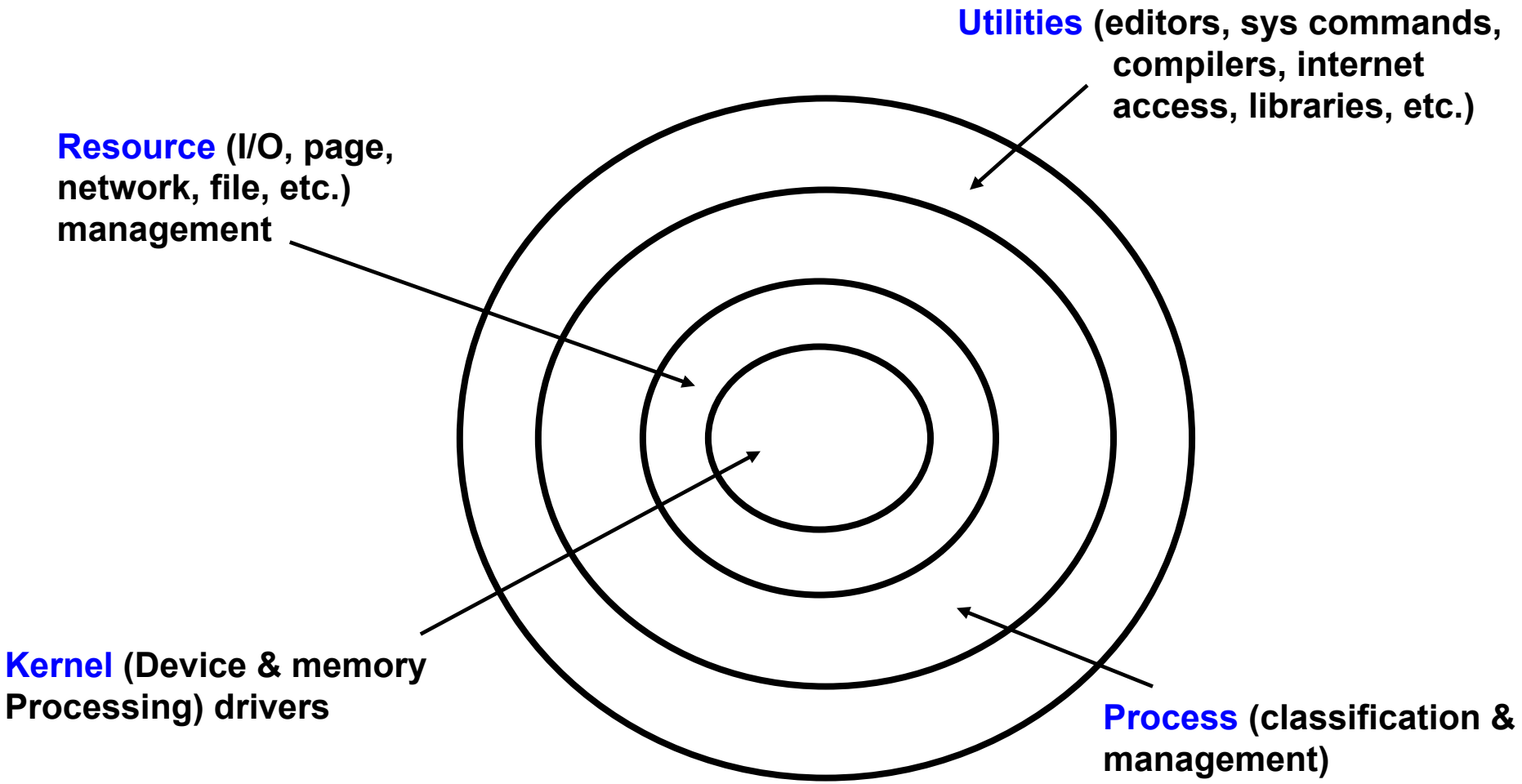
Typical Layered Architecture



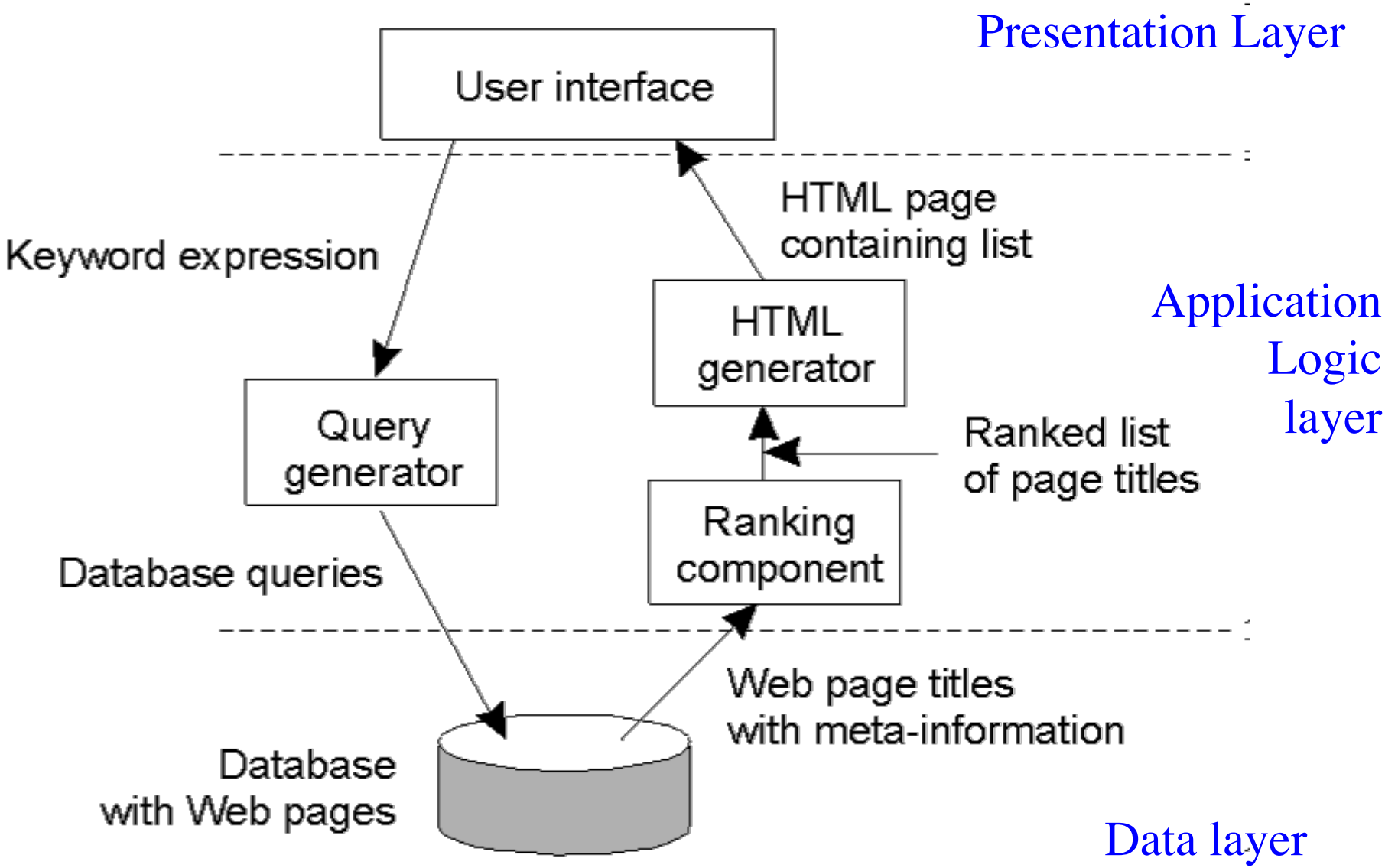
Layered Architecture: Example



Example - Layered Architecture for Operating System



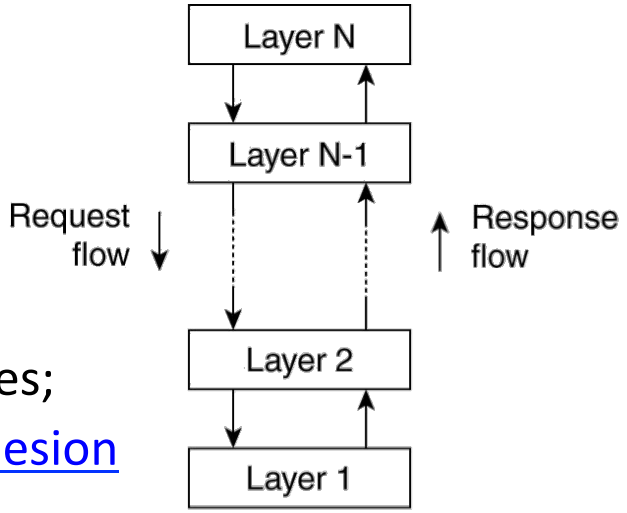
Example - Internet search engine



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



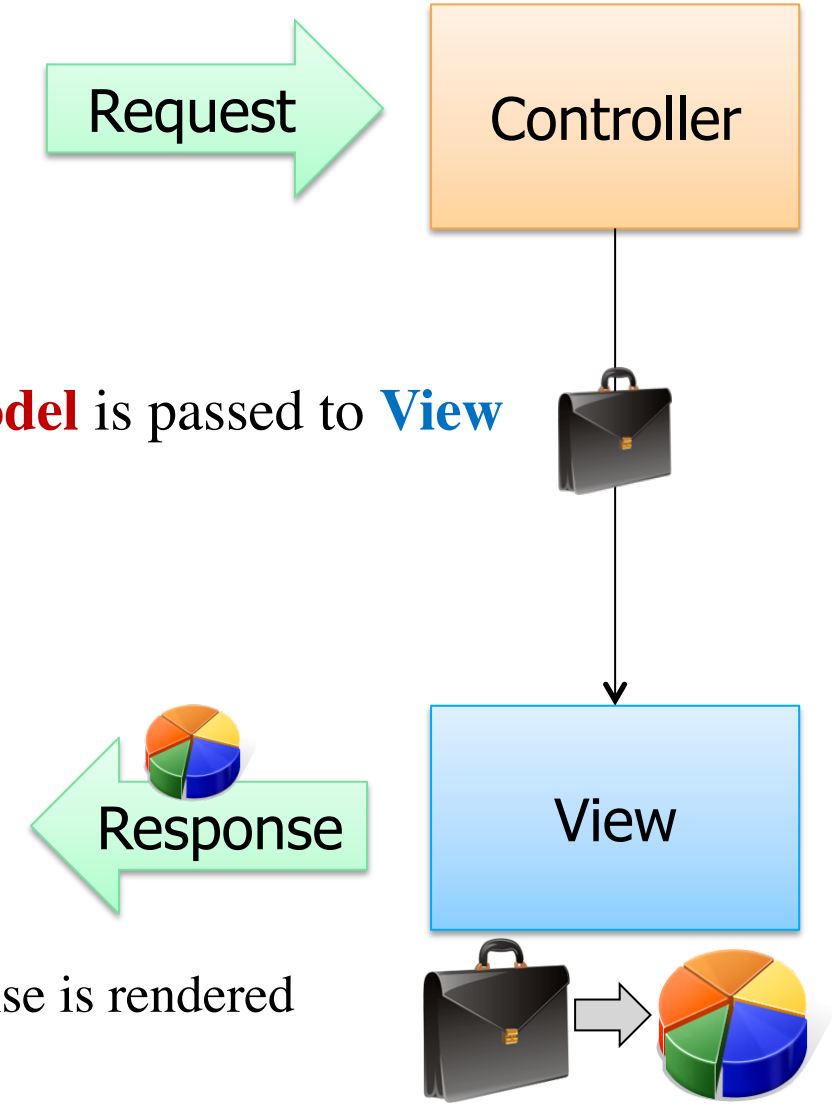
- Advantages:
  - Each layer is selected to be a set of related services; thus the architecture provides high degree of cohesion within the layer.
  - Each layer hides complexity from other layers
  - Layers may use only lower layers hence reducing coupling.
  - Each layer, being cohesive and is coupled 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-Control Architecture (MVC)

Model

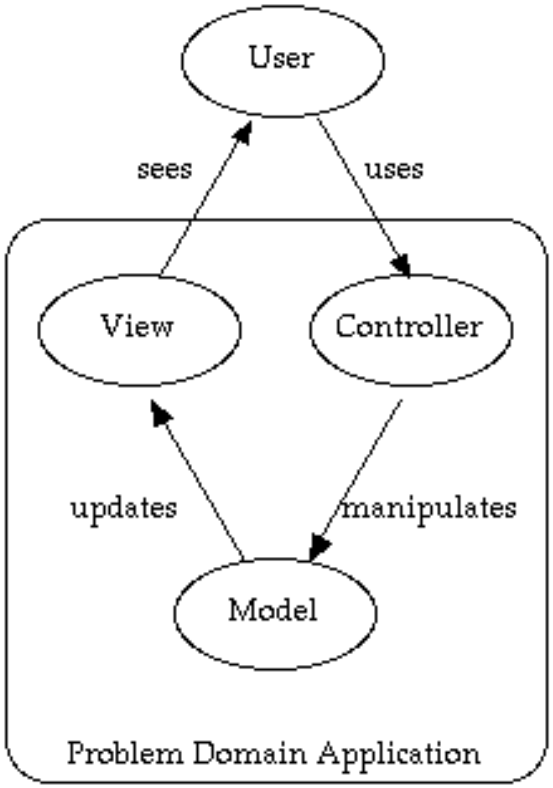
- Holds the application data and implements the application logic
- Know how to carry out specific tasks such as processing a new subscription

View

- The View provides a visual representation of the model.

Controller

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

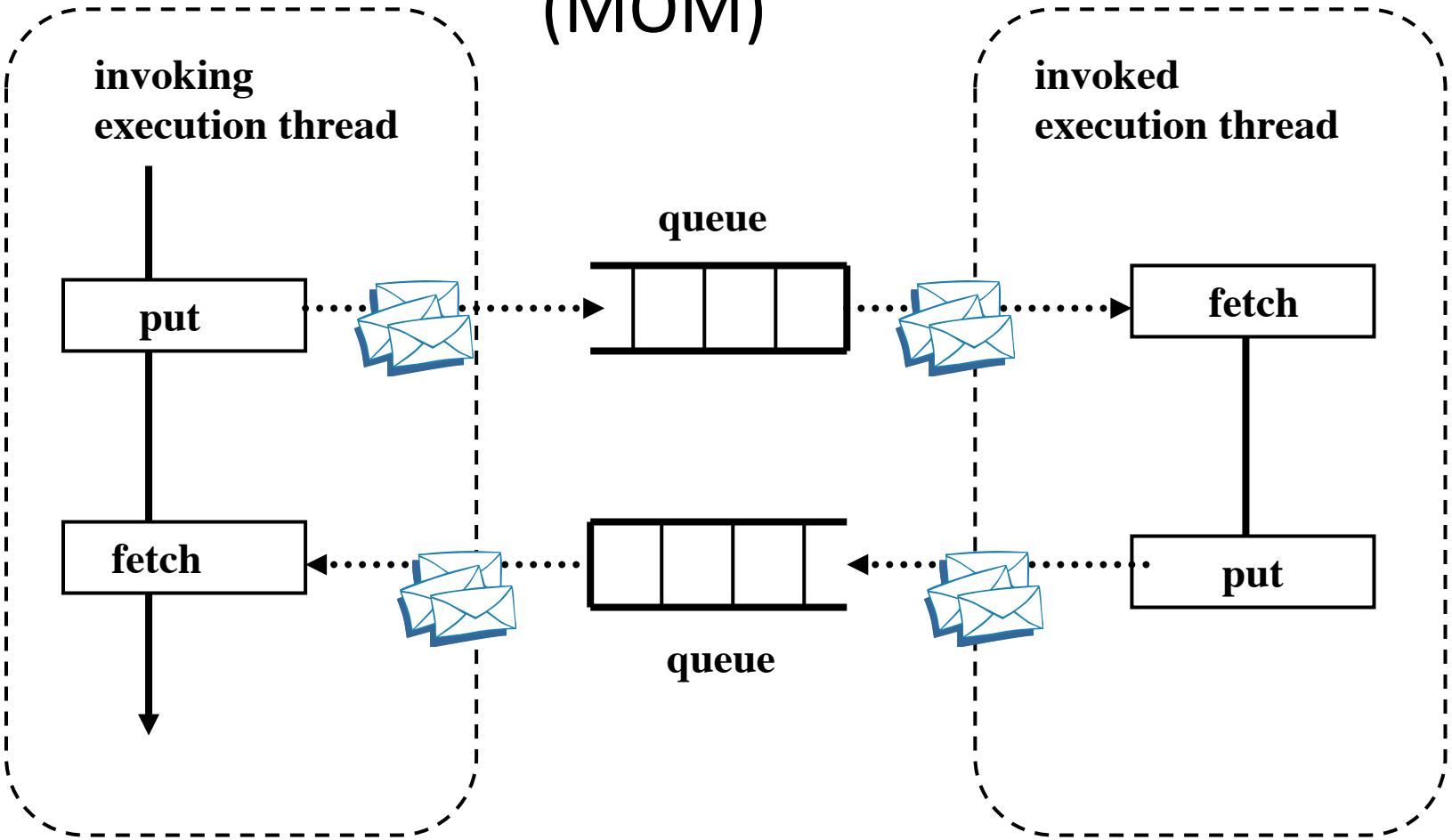


Advantages of MVC

- **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.

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

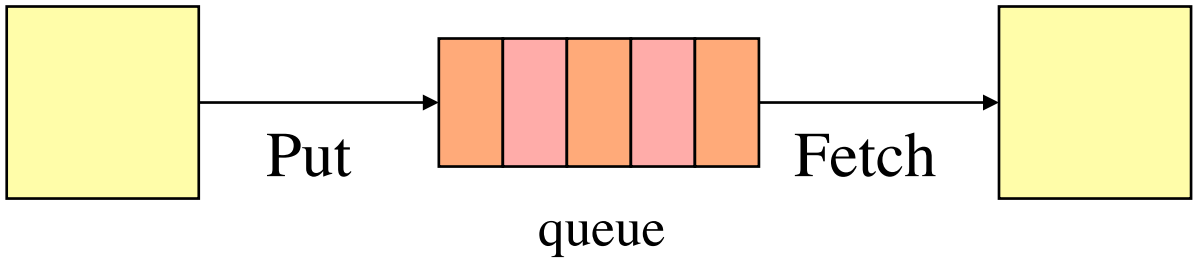


Aimed at achieving *decoupling and reliability*

= Messages

Implicit Invocation Style: Independent Component 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

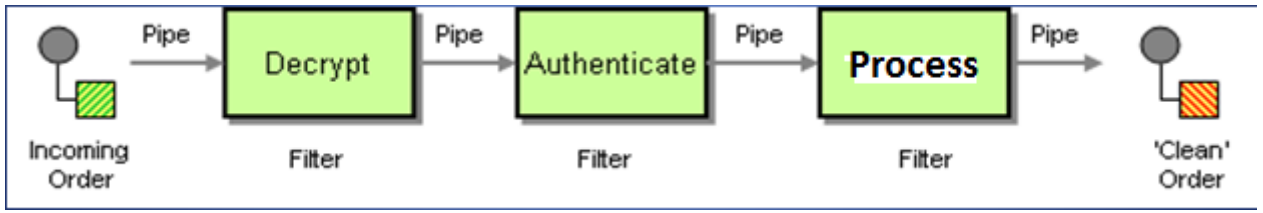
- **Advantages**
  - **Lower coupling between components**: the message senders and the message processors are separate
    - Easier system evolution: e.g., a component can be easily replaced by another one
    - Any sender or processor malfunction will not affect the other senders and message processors
  - 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.

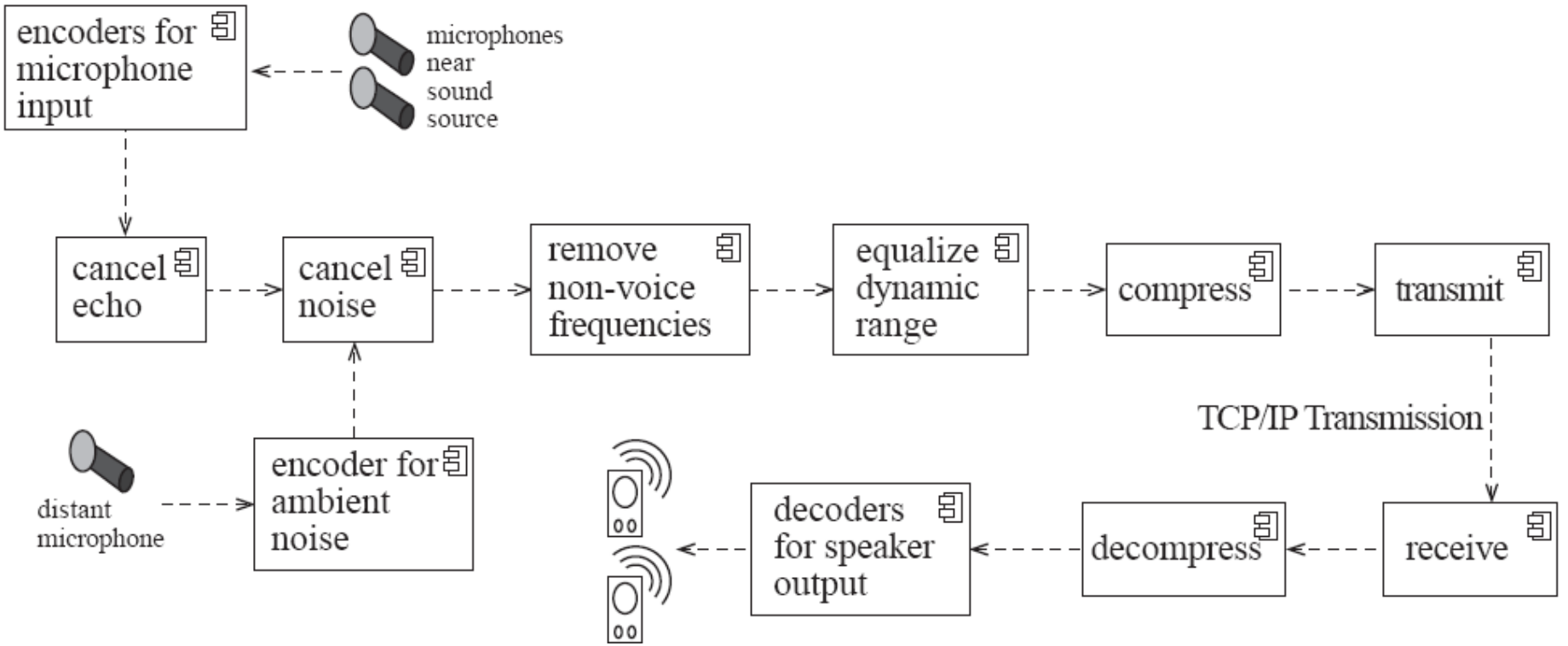
Pipe and Filter : Data Flow Style

- **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 or conduit 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

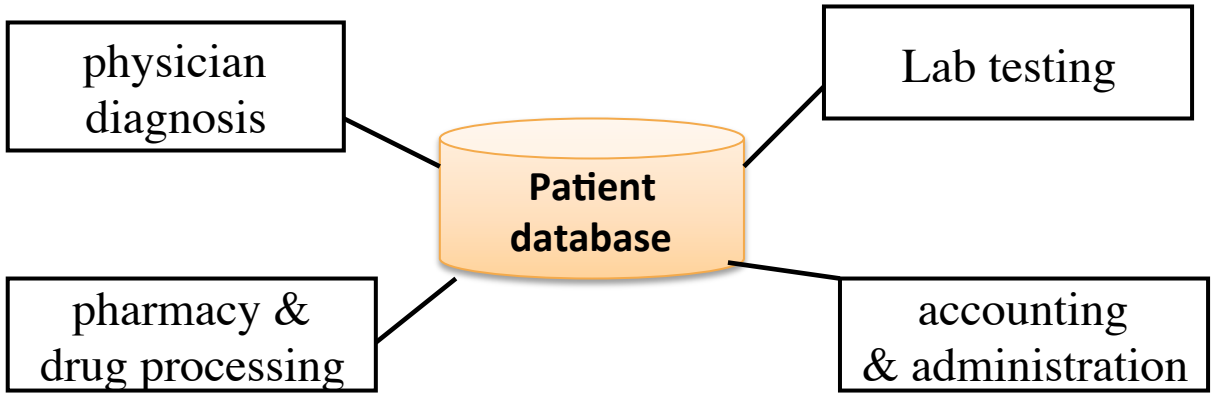


Advantages and Disadvantages of Pipe-Filter

- **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.

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

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.

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

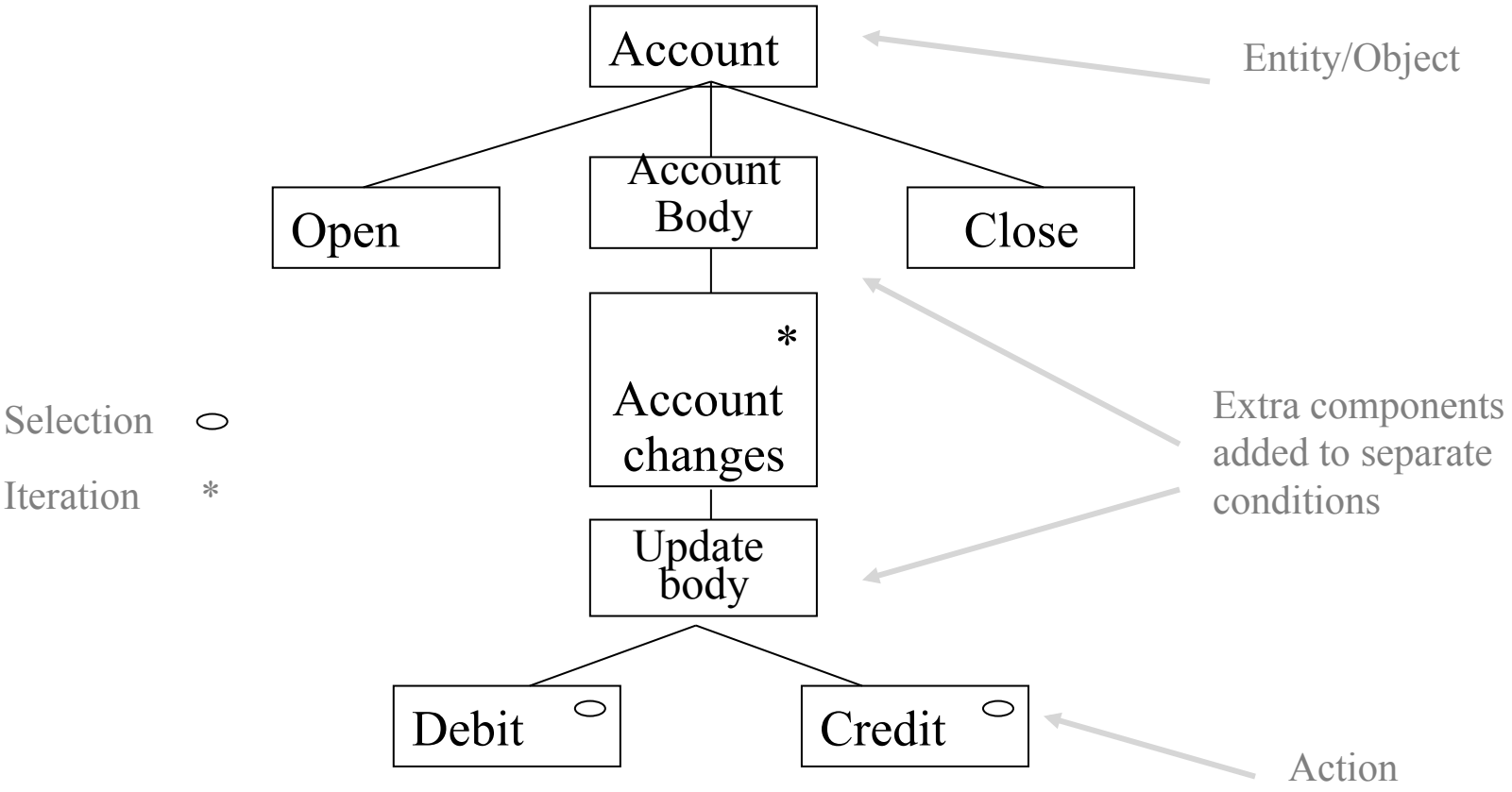
- **Advantages:**
  - Higher component cohesion and low coupling: the **coupling is restricted to the shared data**
  - Single data-store makes the maintenance of data in terms of back-up recovery and security easier to manage
- **Disadvantages:**
  - **High coupling to shared data:** any data format 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.)

Other Style: Object-based Structured Model

- Process structure diagrams (PSD) are tree hierarchy diagrams
  - Trees are composed of leaves called elementary components at the bottom of each branch, and
  - Group components higher up the branches which meet at the top of the tree
- PSD is based on
  - **Entities (objects)**
    - Static
    - Dynamic
    - Active, effect or respond to changes in the real world
  - **Actions (events)**
    - How and when things happen
    - Only actions can be leaves of the PSD

Process Structure Diagram (JSP)

- Classical JSP is the first object-based design paradigm proposed by Michael Jackson.
- The following JSP
- This JSP has one object/Entity: Account
- It has three operations: Open(), (either 'Debit()" or "Credit()", not both), and Close().



Ref.: https://en.wikipedia.org/wiki/Jackson_structured_programming

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