Architectural Design

Slide Set - 10

Organized & Presented By:
Software Engineering Team CSED
TIET, Patiala

Why Architecture

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

Definitions

- The <u>software architecture</u> of a program or computing system is the structure or structures of the system which <u>comprise</u>
  - The software <u>components</u>
  - The externally visible <u>properties</u> of those components
  - The <u>relationships</u> among the components
- <u>Software architectural design</u> represents the <u>structure</u> of the data and program <u>components</u> that are required to build a computer-based system
- An architectural design model is <u>transferable</u>
  - It can be <u>applied</u> to the design of other systems
  - It <u>represents</u> a set of <u>abstractions</u> that enable software engineers to describe architecture in <u>predictable</u> ways

Example -Architectural Design

customer requirements

"four bedrooms, three baths,
lots of glass ..."
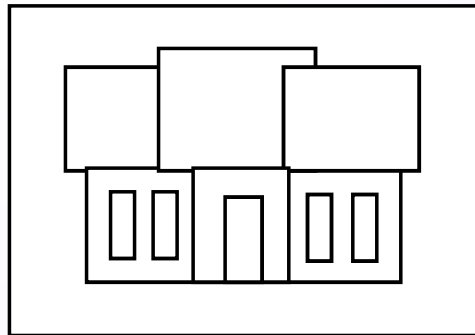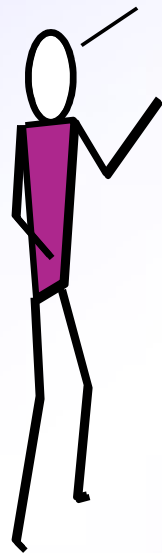
architectural design
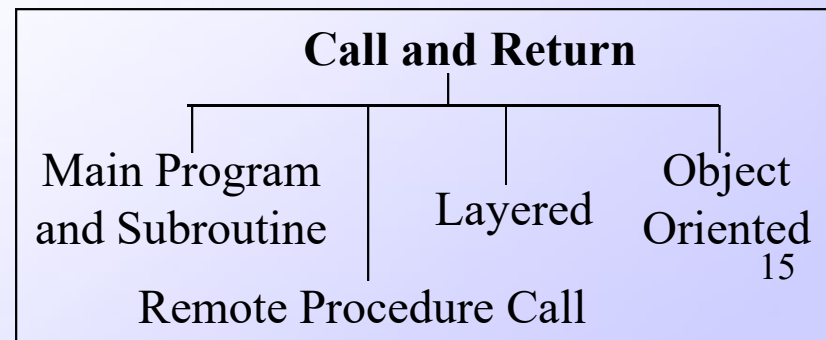
Architectural Design Process
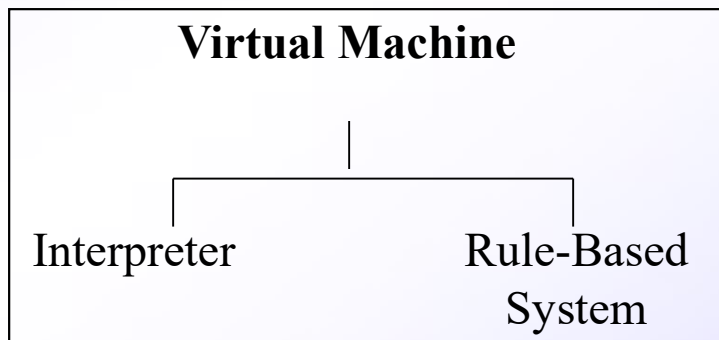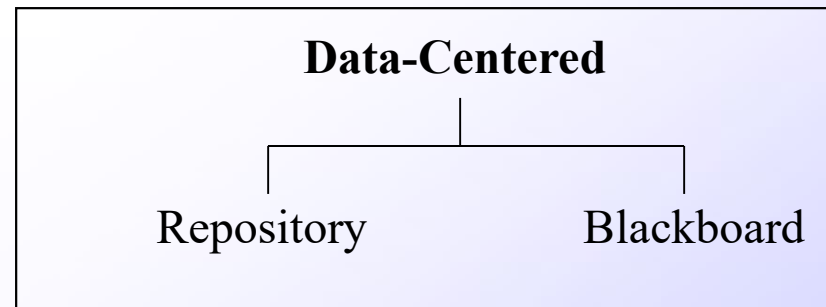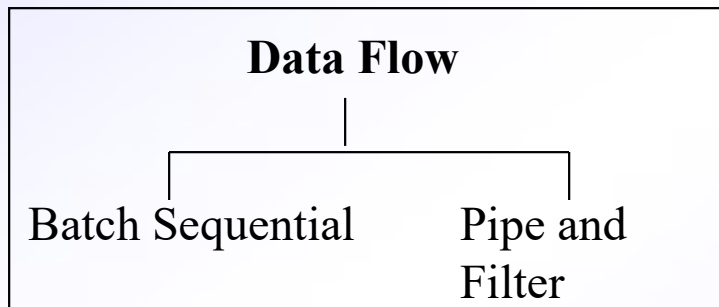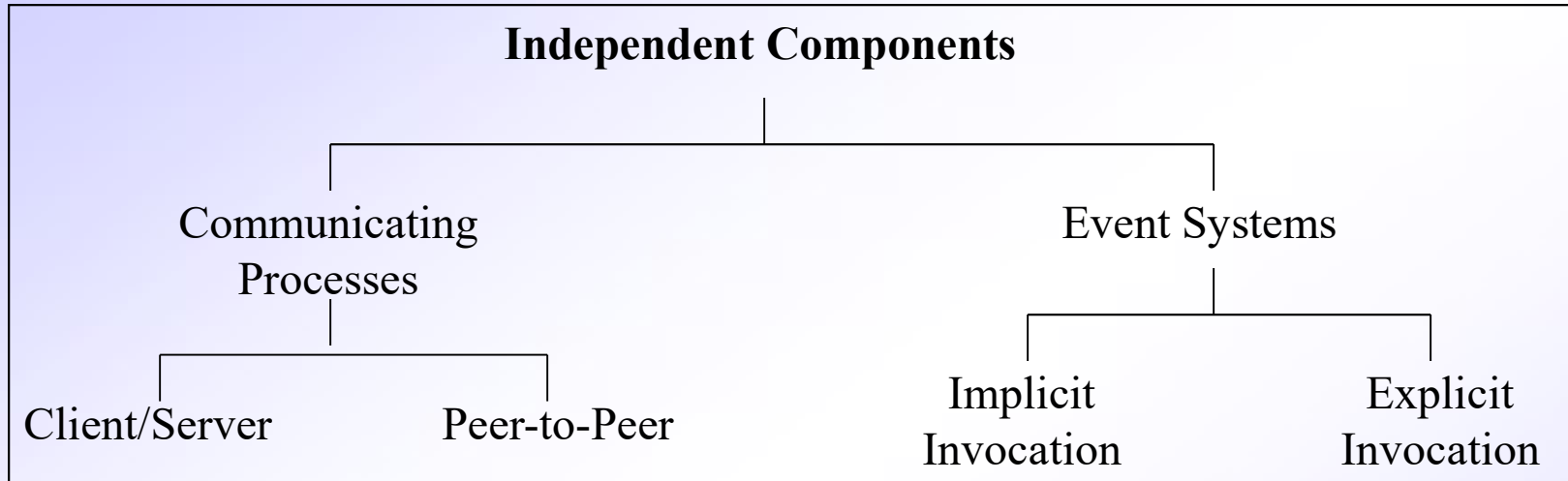
- Basic Steps
  - <u>Creation</u> of the data design
  - <u>Derivation</u> of one or more representations of the <u>architectural structure</u> of the system
  - <u>Analysis</u> of alternative <u>architectural styles</u> to choose the one best suited to customer requirements and quality attributes
  - <u>Elaboration</u> of the architecture based on the selected architectural style
- A <u>database designer</u> creates the data architecture for a system to represent the data components
- A <u>system architect</u> selects an appropriate architectural style derived during system engineering and software requirements analysis
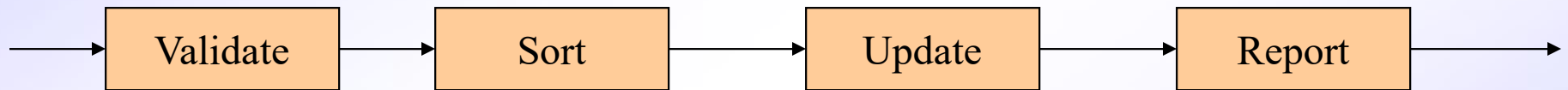
Software Architectural Styles

Software Architectural Style

- The software that is built for computer-based systems exhibit one of many <u>architectural styles</u>
- Each <u>style</u> describes a system category that encompasses
  - A set of <u>component types</u> that perform a function required by the system
  - A set of <u>connectors</u> (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
  - <u>Semantic constraints</u> that define how components can be integrated to form the system
  - <u>A topological layout</u> of the components indicating their runtime interrelationships

14

(Source: Bass, Clements, and Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003)

A Taxonomy of Architectural Styles

Independent Components

Communicating Processes

Client/Server   Peer-to-Peer

Event Systems

Implicit Invocation   Explicit Invocation

Data Flow

Batch Sequential   Pipe and Filter

Data-Centered

Repository   Blackboard

Virtual Machine

Interpreter   Rule-Based System

Call and Return

Main Program and Subroutine   Layered   Object Oriented

Remote Procedure Call

15

Data Flow Style

Data Flow Style

- Has the <u>goal</u> of modifiability
- Characterized by viewing the system as a series of transformations on successive pieces of input data
- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store
- <u>Batch sequential</u> style
  - The processing steps are independent components
  - Each step runs to completion before the next step begins
- <u>Pipe-and-filter</u> style
  - Emphasizes the incremental transformation of data by successive components
  - The filters incrementally transform the data (entering and exiting via streams)
  - The filters use little contextual information and retain no state between instantiations
  - The pipes are stateless and simply exist to move data between filters

(More on next slide)

Data Flow Architectures
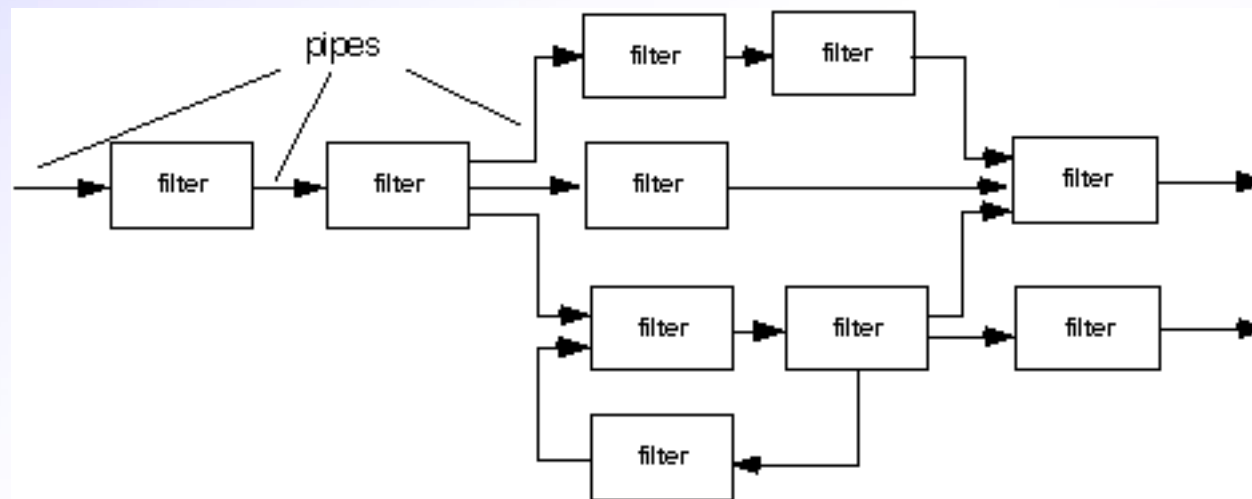
(a) pipes and filters

(b) batch sequential
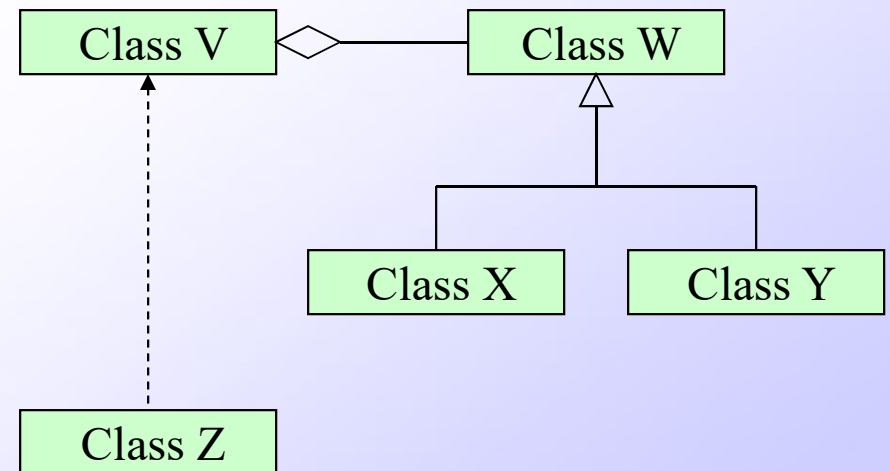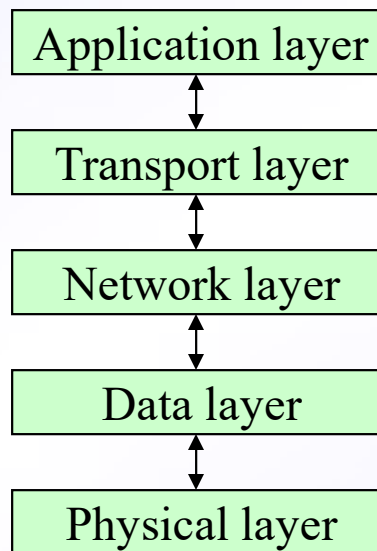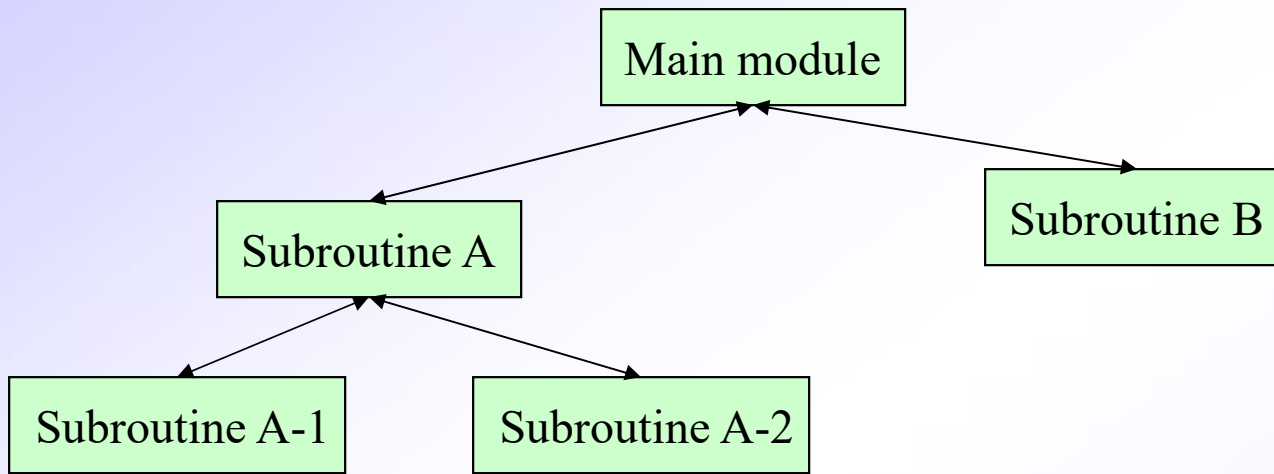
Data Flow Style (continued)

- Advantages
  - Has a <u>simplistic</u> design in the limited ways in which the components interact with the environment
  - Consists of no more and no less than the construction of its parts
  - Simplifies reuse and maintenance
  - Is easily made into a <u>parallel</u> or <u>distributed</u> execution in order to enhance system performance
- Disadvantages
  - Implicitly encourages a <u>batch mentality</u> so interactive applications are difficult to create in this style
  - <u>Ordering</u> of filters can be <u>difficult</u> to maintain so the filters cannot cooperatively interact to solve a problem
  - Exhibits <u>poor performance</u>
    - Filters typically force the least common denominator of data representation (usually ASCII stream)
    - Filter may need unlimited buffers if they cannot start producing output until they receive all of the input
    - Each filter operates as a separate process or procedure call, thus incurring overhead in set-up and take-down time

19

(More on next slide)

Data Flow Style (continued)

- Use this style when it makes sense to view your system as one that produces a well-defined easily identified output
  - The output should be a direct result of <u>sequentially transforming</u> a well-defined easily identified input in a time-independent fashion

Call-and-Return Style

Main module

Subroutine A

Subroutine B

Subroutine A-1

Subroutine A-2

Application layer

Transport layer

Network layer

Data layer

Physical layer

Class V

Class W

Class X

Class Y

Class Z

21

Call-and-Return Style

- Has the <u>goal</u> of modifiability and scalability
- Has been the dominant architecture since the start of software development
- <u>Main program and subroutine</u> style
  - Decomposes a program <u>hierarchically</u> into small pieces (i.e., modules)
  - Typically has a <u>single thread</u> of control that travels through various components in the hierarchy
- <u>Remote procedure call</u> style
  - Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
  - Strives to increase performance by distributing the computations and taking advantage of multiple processors
 - Incurs a finite communication time between subroutine call and response

(More on next slide)

Call and Return Architecture
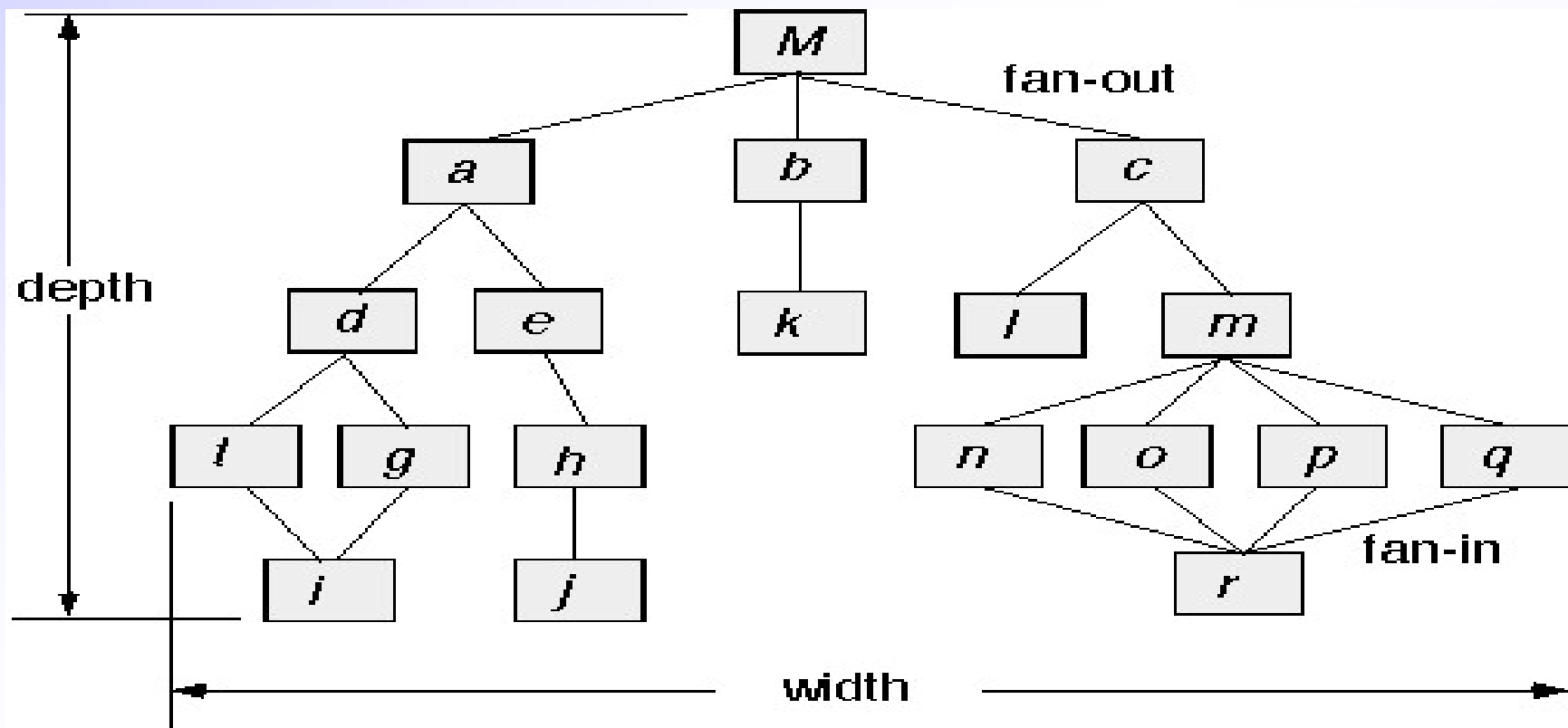
Call-and-Return Style (continued)

- Layered system
    - Assigns components to layers in order to control inter-component interaction
    - Only allows a layer to communicate with its immediate neighbor
    - Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
    - Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
    - Is compromised by layer bridging that skips one or more layers to improve runtime performance
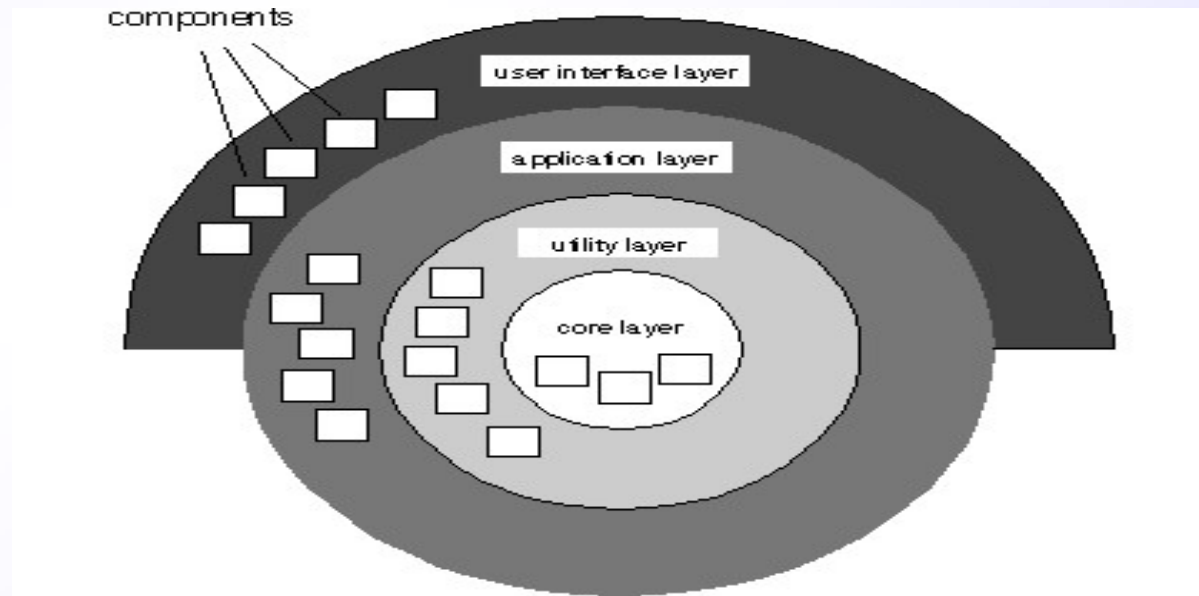
Call and Return Style (Continued)

- Object-oriented or abstract data type system
  - Emphasizes the bundling of data and how to manipulate and access data
  - Keeps the internal data representation hidden and allows access to the object only through provided operations
  - Permits inheritance and polymorphism

Data-Centered Style (continued)

- Has the <u>goal</u> of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an <u>independent</u> thread of control
- The shared data may be a <u>passive</u> repository or an <u>active</u> blackboard
  - A blackboard notifies subscriber clients when changes occur in data of interest
- At its heart is a <u>centralized</u> data store that communicates with a number of clients
- Clients are relatively <u>independent</u> of each other so they can be added, removed, or changed in functionality
- The data store is <u>independent</u> of the clients
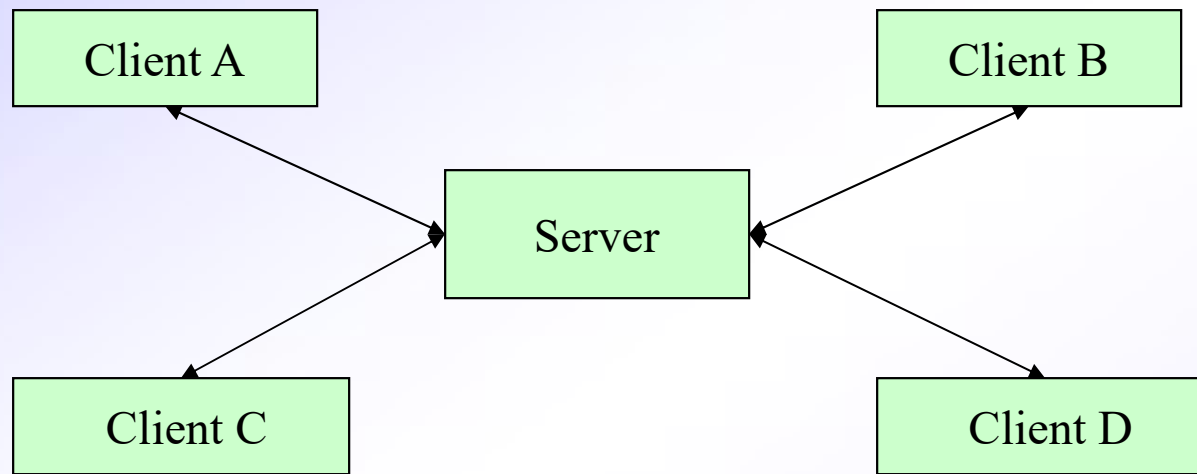
Data Centred Architecture
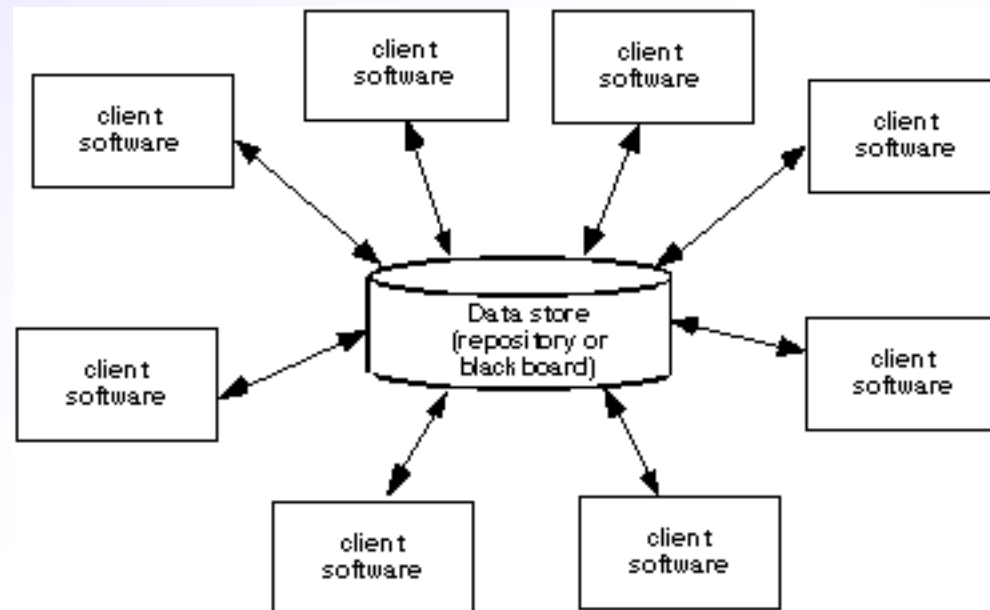
Data-Centered Style (continued)

- Use this style when a <u>central issue</u> is the storage, representation, management, and retrieval of a large amount of related persistent data

- Note that this style becomes <u>client/server</u> if the clients are modeled as independent processes

Independent Component Style
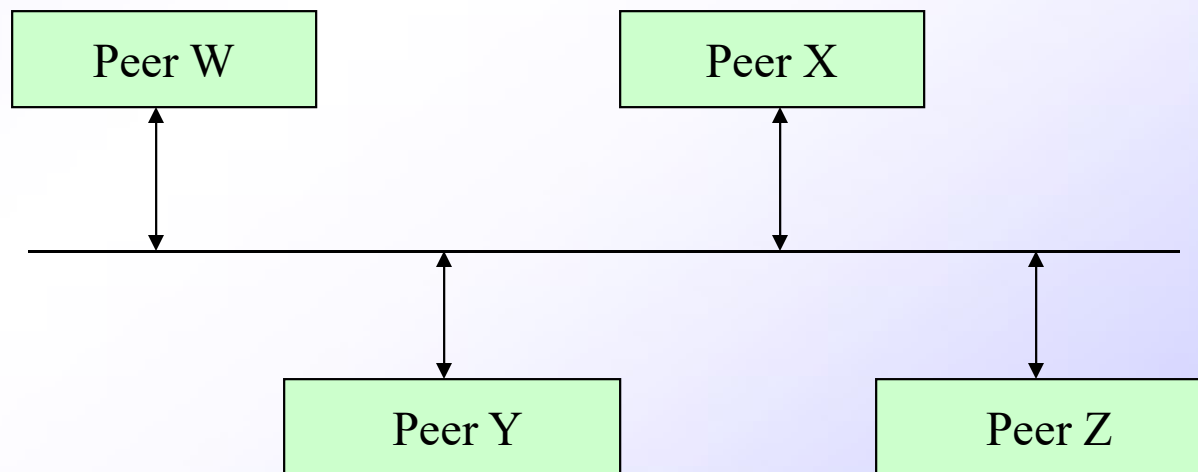
Independent Component Style

- Consists of a number of <u>independent</u> processes that communicate through messages
- Has the <u>goal</u> of modifiability by decoupling various portions of the computation
- Sends data between processes but the processes <u>do not</u> directly control each other
- <u>Event systems</u> style
  - Individual components <u>announce</u> data that they wish to share (<u>publish</u>) with their environment
  - The other components may <u>register</u> an interest in this class of data (subscribe)
  - Makes use of a message component that <u>manages</u> communication among the other components
  - Components <u>publish</u> information by <u>sending</u> it to the message manager
  - When the data appears, the subscriber is invoked and receives the data
  - <u>Decouples</u> component implementation from knowing the names and locations of other components

(More on next slide)

Independent Component Style (continued)

- <u>Communicating processes</u> style
  - These are classic multi-processing systems
  - Well-know subtypes are client/server and peer-to-peer
  - The <u>goal</u> is to achieve scalability
  - A <u>server</u> exists to provide data and/or services to one or more clients
  - The <u>client</u> originates a call to the server which services the request
- Use this style when
  - Your system has a <u>graphical user interface</u>
  - Your system runs on a <u>multiprocessor</u> platform
  - Your system can be structured as a set of <u>loosely coupled</u> components
  - Performance tuning by <u>reallocating </u>work among processes is important
  - <u>Message passing</u> is sufficient as an interaction mechanism among components

Architectural Design Process
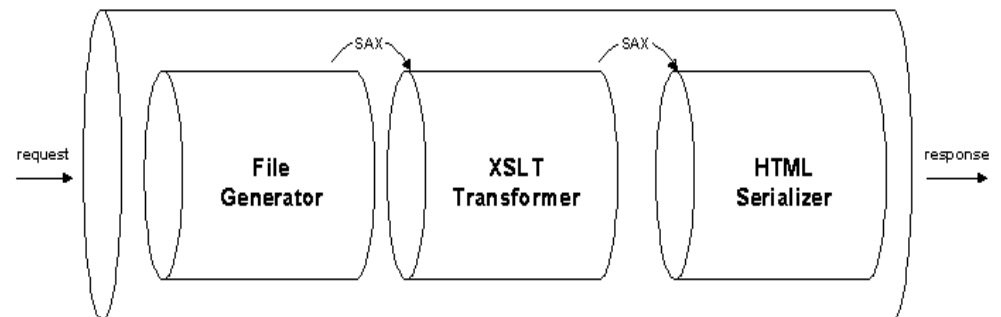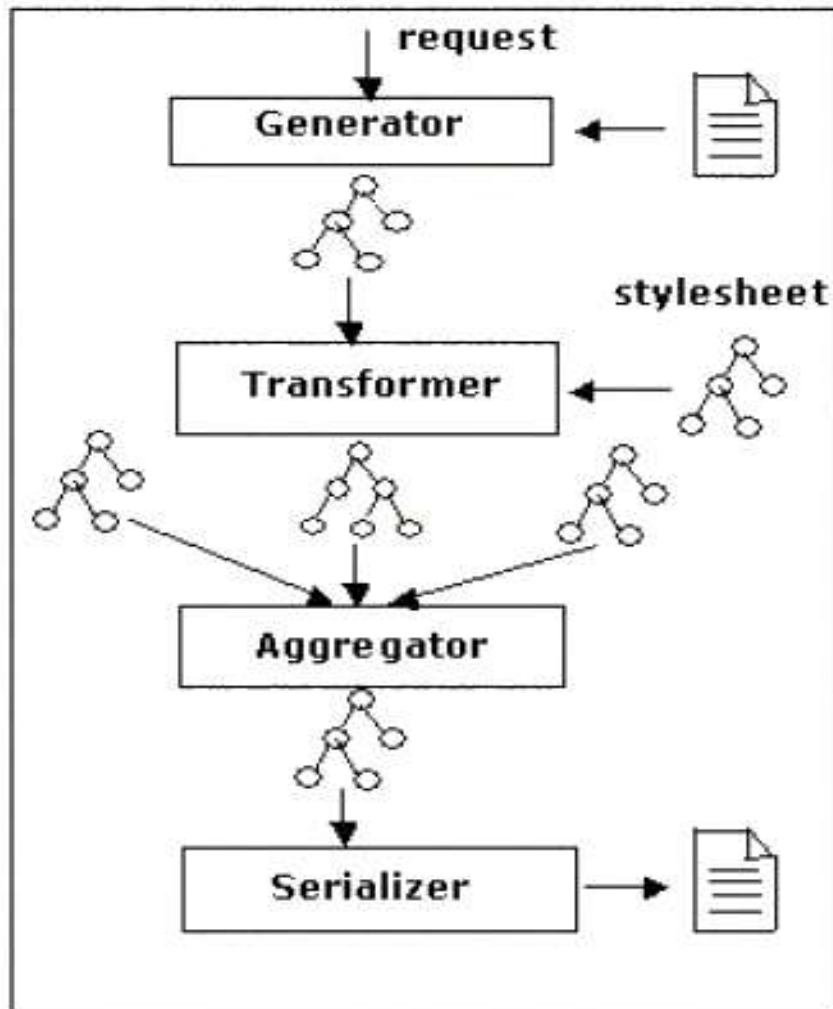
Pipes and filters

The Pipes and Filters architectural pattern [style] provides a structure for systems that process a stream of data.

Each processing step is encapsulated in a filter component.

Data is passed through pipes between adjacent filters.

Recombining filters allows you to build families of related systems.

Example: apache cacoon's pipes & xslt filters

Tiered architecture (layering)

2 - tier architecture (traditional client-server)

A two-way interaction in a client/server environment, in which the user interface is stored in the client and the data are stored in the server. The application logic can be in either the client or the server.

2 Tier
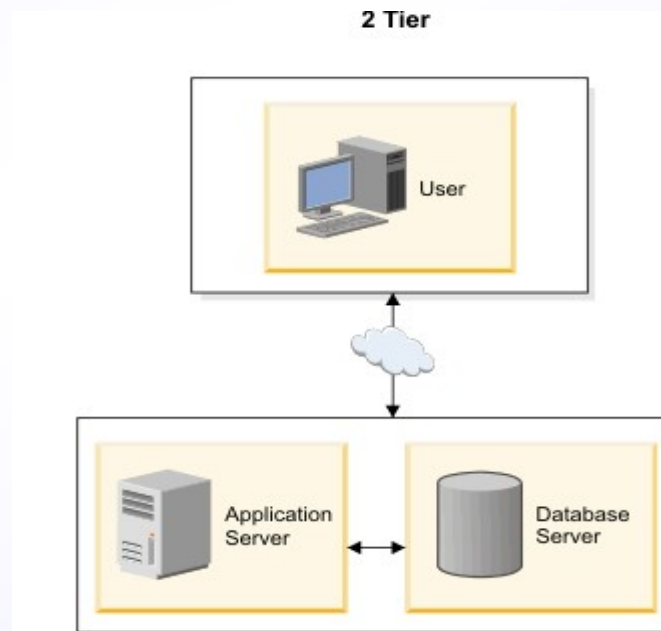
3 tier architecture

Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

> GET SALES TOTAL

> GET SALES TOTAL

4 TOTAL SALES

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

GET LIST OF ALL SALES MADE LAST YEAR

ADD ALL SALES TOGETHER

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

QUERY

SALE 1
SALE 2
SALE 3
SALE 4

Database

Storage
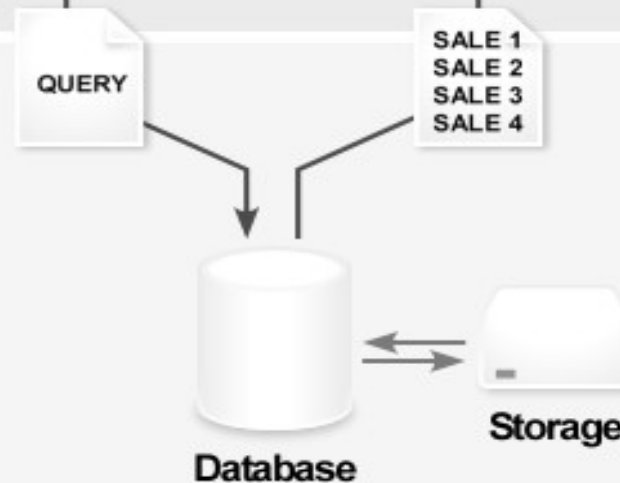
Model-view-controller (1)

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts:

➢ the model, the view, and the controller.

MVC was originally developed to map the traditional input, processing, output roles into the GUI realm:

$$\text{Input} \rightarrow \text{Processing} \rightarrow \text{Output}$$

$$\text{Controller} \rightarrow \text{Model} \rightarrow \text{View}$$

Model-view-controller (2)

The pattern isolates business logic from input and presentation, permitting independent development, testing and maintenance of each.

Model-View-Controller concept. Note: The solid line represents a direct association, the dashed an indirect association via an observer (for example).