THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Architectural Design

Slide Set - 10

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

Chapter 10

Architectural Design

- Introduction
- Data design
- Software architectural styles
- Architectural design process

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

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

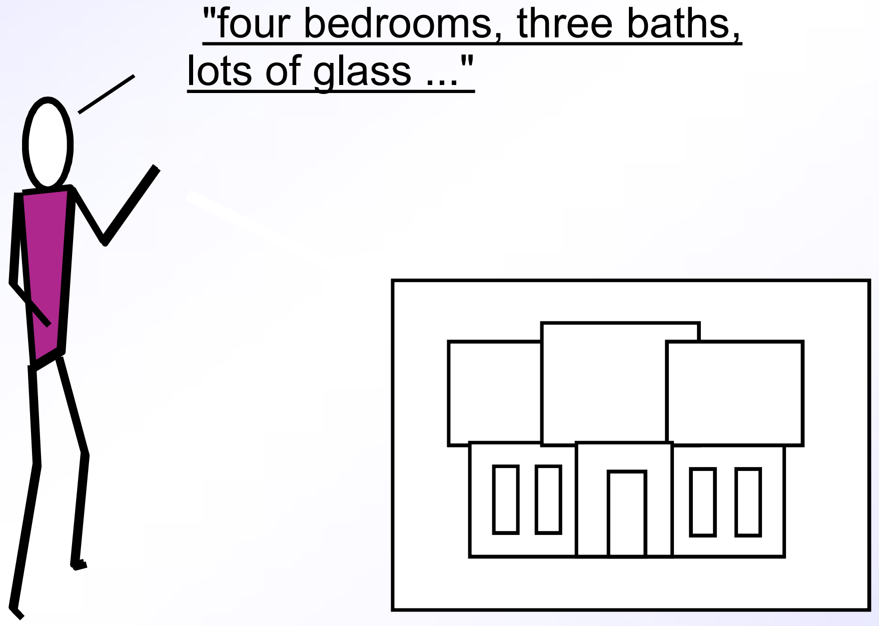
(3) reduce the risks associated with the construction of the software.

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



architectural design

Architectural Design Process

- Basic Steps
  - Creation of the data design
  - Derivation of one or more representations of the architectural structure of the system
  - Analysis of alternative architectural styles to choose the one best suited to customer requirements and quality attributes
  - Elaboration of the architecture based on the selected architectural style
- A database designer creates the data architecture for a system to represent the data components
- A system architect selects an appropriate architectural style derived during system engineering and software requirements analysis

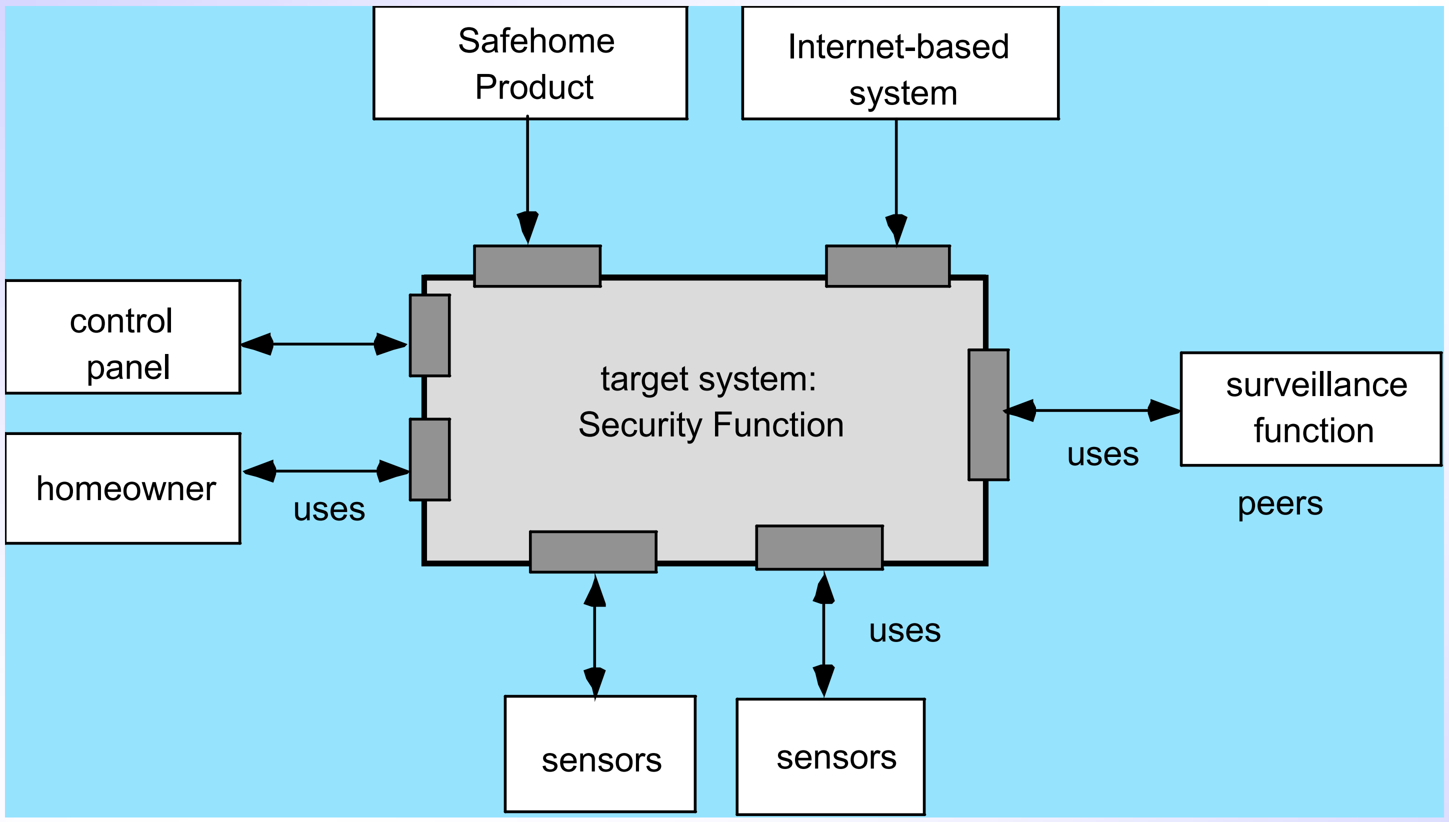
Importance of Software Architecture

- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system
- The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity
- The software architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

Emphasis on Software Components

- A software architecture enables a software engineer to
  - Analyze the effectiveness of the design in meeting its stated requirements
  - Consider architectural alternatives at a stage when making design changes is still relatively easy
  - Reduce the risks associated with the construction of the software
- Focus is placed on the software component
  - A program module
  - An object-oriented class
  - A database
  - Middleware

Architectural Context



Data Design

Purpose of Data Design

- Data design <u>translates</u> data objects defined as part of the analysis model into
  - Data structures at the software component level
  - A possible database architecture at the application level
- It <u>focuses</u> on the representation of data structures that are directly accessed by one or more software components
- The challenge is to <u>store and retrieve</u> the data in such way that useful information can be extracted from the data environment
- "Data quality is the <u>difference</u> between a data warehouse and a data garbage dump"

Data Design Principles

- The systematic analysis principles that are applied to function and behavior should also be applied to data
- All data structures and the operations to be performed on each one should be identified
- A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it
- Low-level data design decisions should be deferred until late in the design process
- The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure
- A library of useful data structures and the operations that may be applied to them should be developed
- A software programming language should support the specification and realization of abstract data types

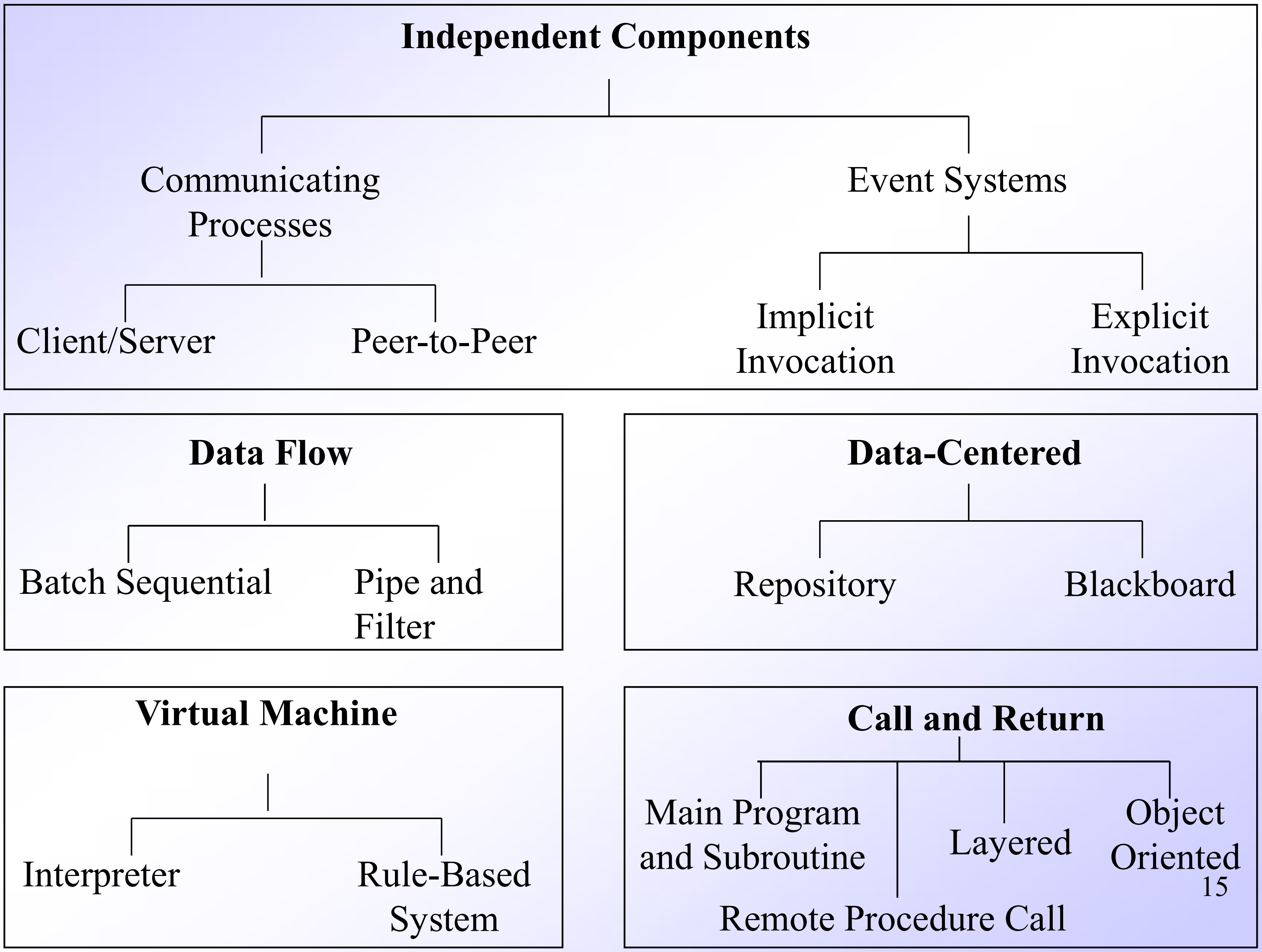
Software Architectural Styles

Software Architectural Style

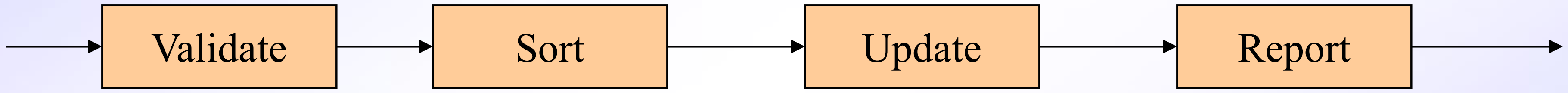
- The software that is built for computer-based systems exhibit one of many architectural styles
- Each style describes a system category that encompasses
  - A set of component types that perform a function required by the system
  - A set of connectors (subroutine call, remote procedure call, data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

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

A Taxonomy of Architectural Styles



Data Flow Style

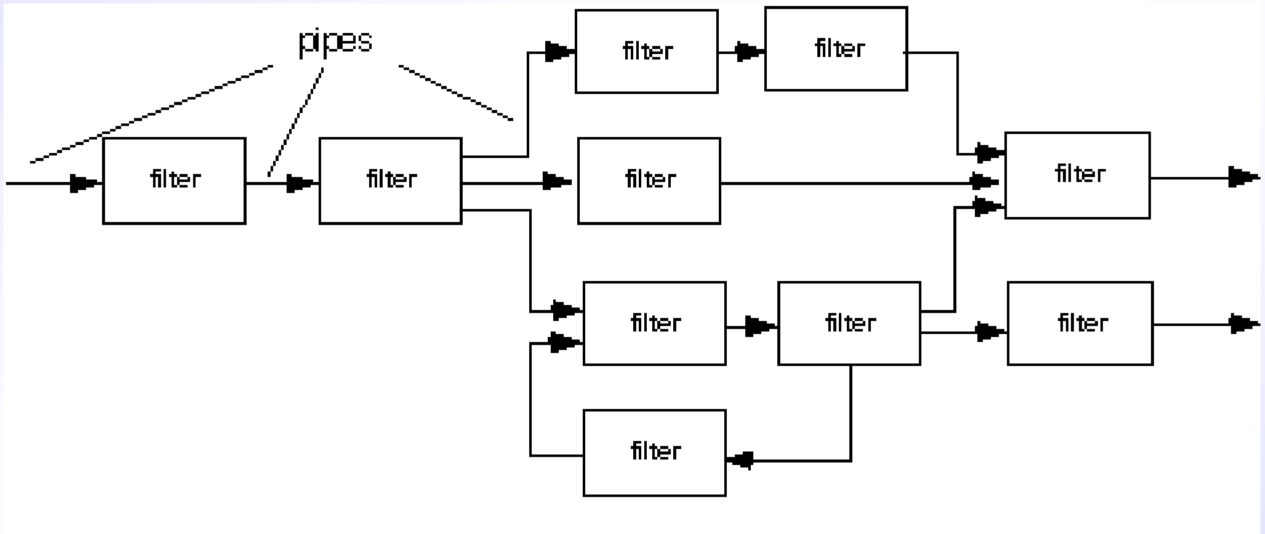


Data Flow Style

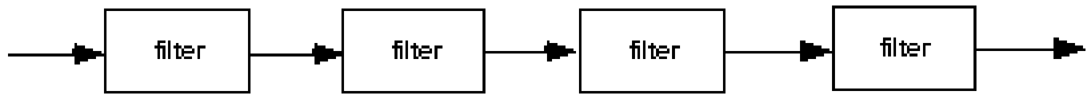
- Has the goal of modifiability
- Characterized by viewing the system as a series of transformations on successive pieces of input data
- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store
- Batch sequential style
  - The processing steps are independent components
  - Each step runs to completion before the next step begins
- Pipe-and-filter 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)

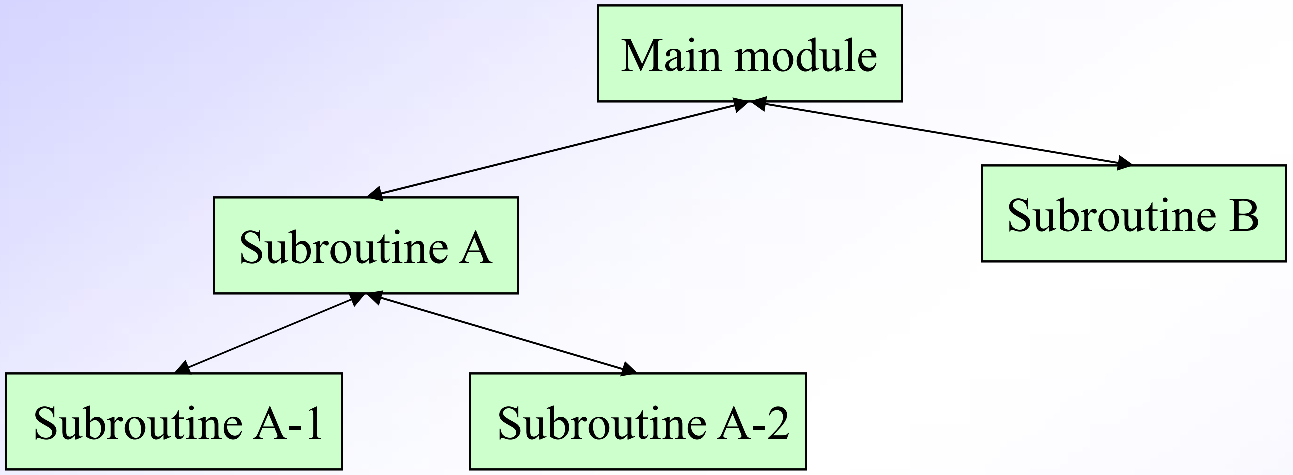
- Advantages
  - Has a <u>simplistic</u> design in the limited ways in which the components interact with the environment
  - Consists of no more and no less than the construction of its parts
  - Simplifies reuse and maintenance
  - Is easily made into a <u>parallel</u> or <u>distributed</u> execution 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

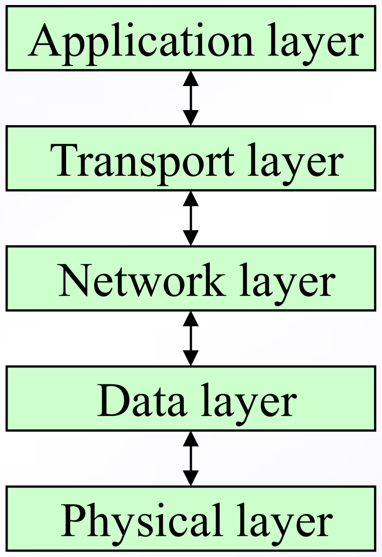
(More on next slide)

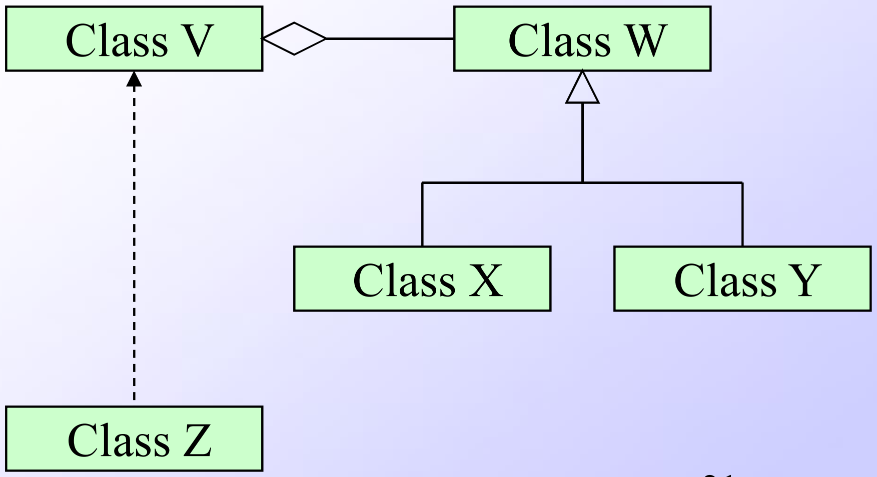
Data Flow Style (continued)

- 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 sequentially transforming a well-defined easily identified input in a time-independent fashion

Call-and-Return Style





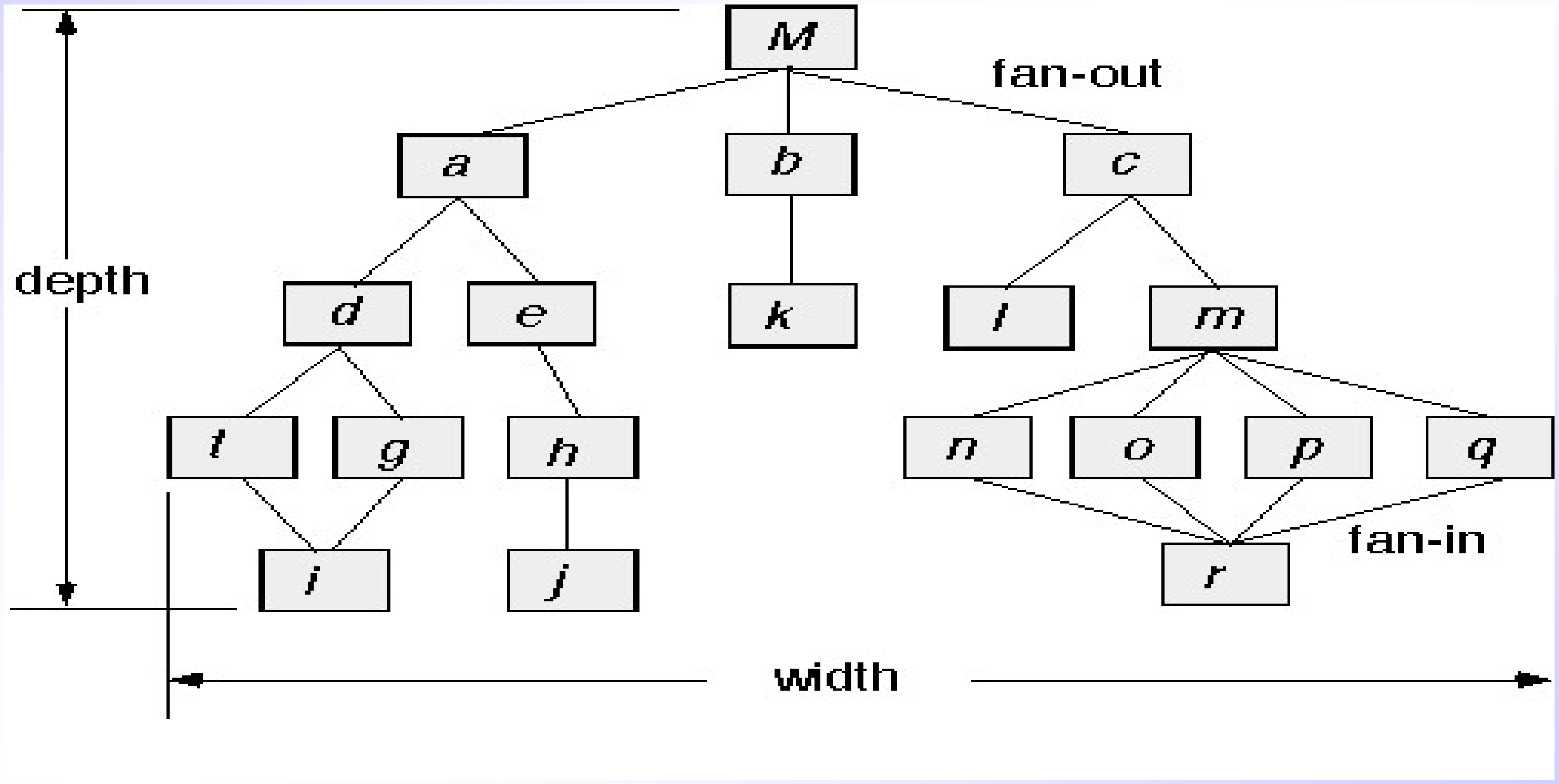


Call-and-Return Style

- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
  - Decomposes a program hierarchically into small pieces (i.e., modules)
  - Typically has a single thread of control that travels through various components in the hierarchy
- Remote procedure call 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

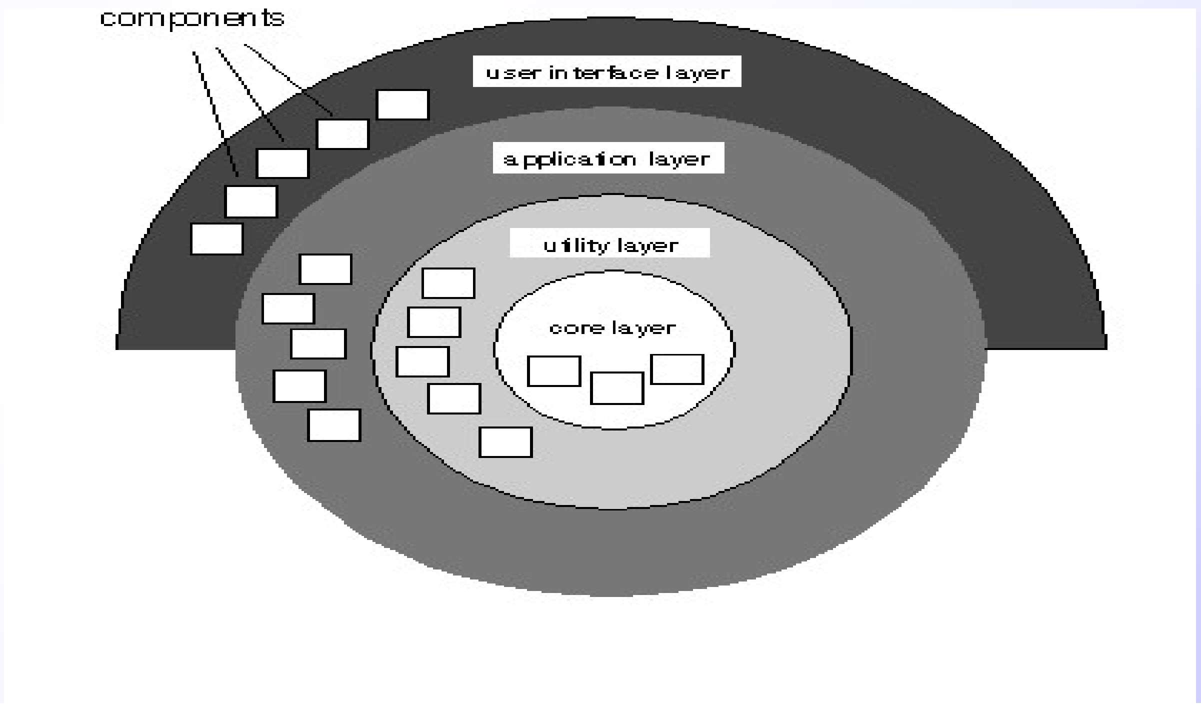
(More on next slide)

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)

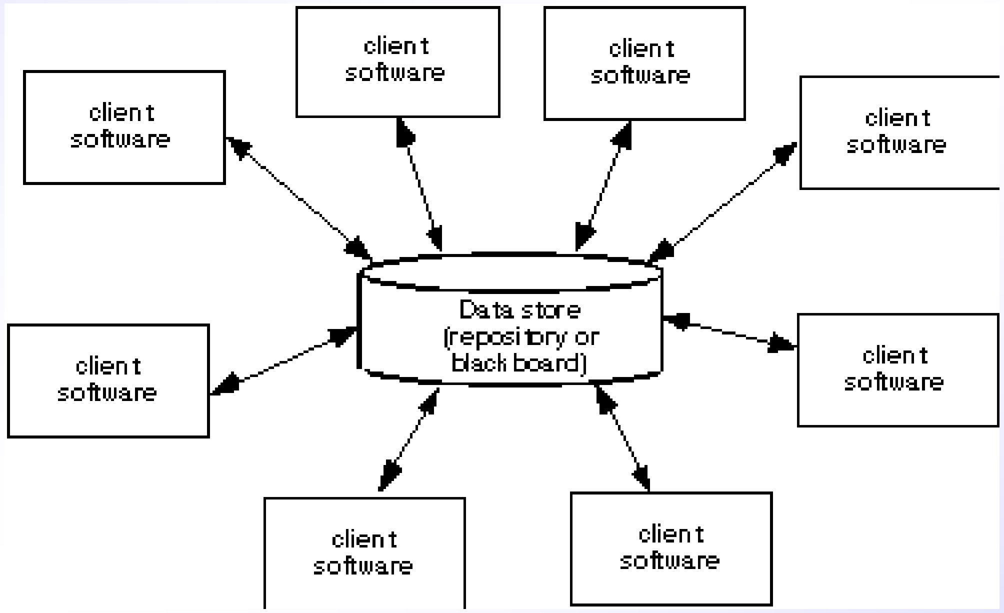
- Object-oriented or abstract data type system
  - Emphasizes the bundling of data and how to manipulate and access data
  - Keeps the internal data representation hidden and allows access to the object only through provided operations
  - Permits inheritance and polymorphism

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

(More on next slide)

Data Centred Architecture

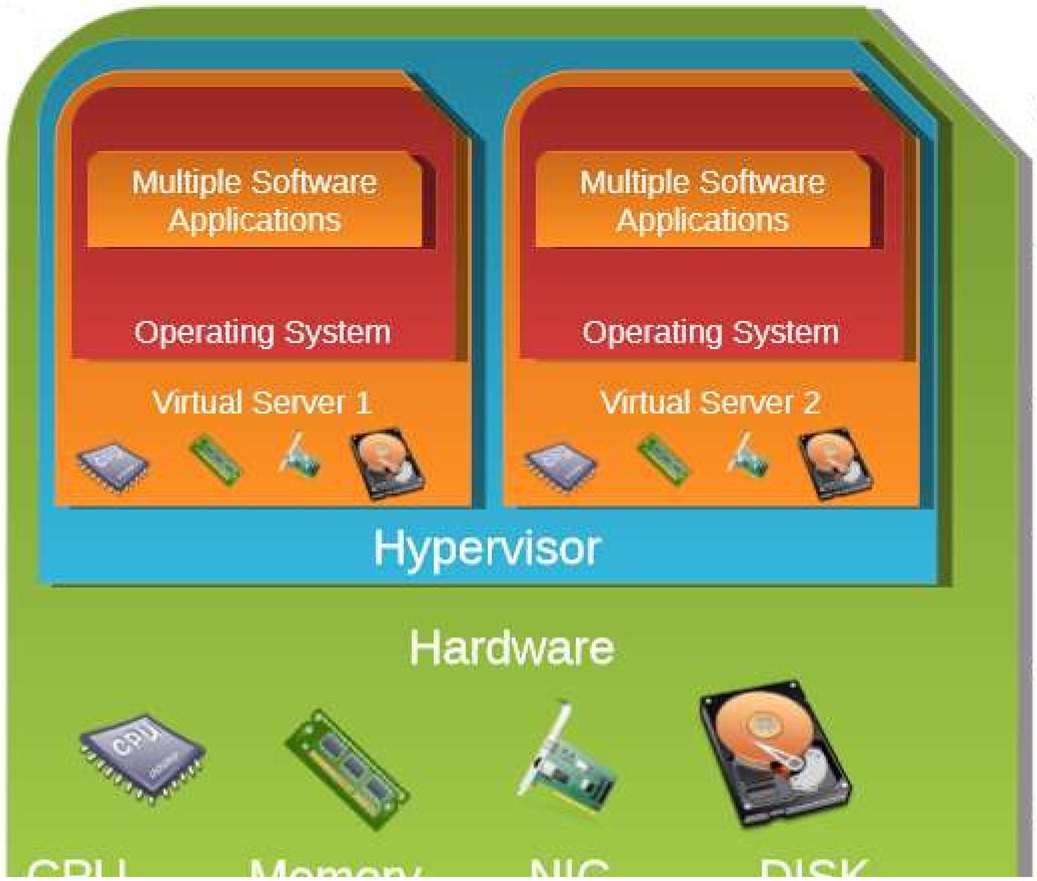


Data-Centered Style (continued)

- Use this style when a central issue is the storage, representation, management, and retrieval of a large amount of related persistent data
- Note that this style becomes client/server if the clients are modeled as independent processes

Virtual Machine Style

SERVER WITH VIRTUALIZATION

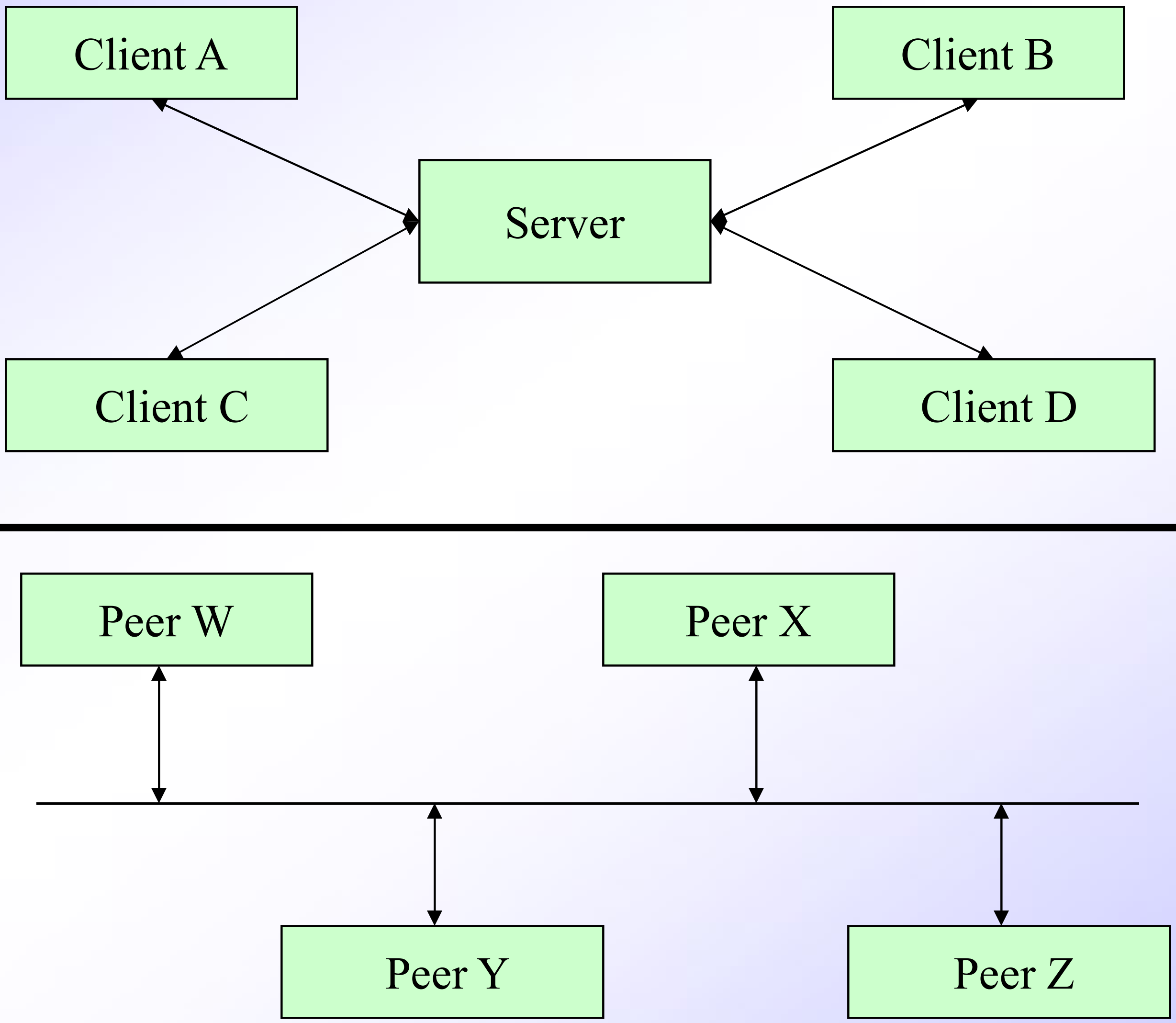


- Can run multiple OS simultaneously.
- Each OS can have different hardware configuration.
- Efficient utilization of hardware resources.
- Each virtual machine is independent.
- Save electricity, initial cost to buy servers, space etc.
- Easy to manage and monitor virtual machines centrally

Virtual Machine Style

- Has the goal of portability
- Software systems in this style simulate some functionality that is not native to the hardware and/or software on which it is implemented
  - Can simulate and test hardware platforms that have not yet been built
  - Can simulate "disaster modes" as in flight simulators or safety-critical systems that would be too complex, costly, or dangerous to test with the real system
- Examples include interpreters, rule-based systems, and command language processors
- Interpreters
  - Add flexibility through the ability to interrupt and query the program and introduce modifications at runtime
  - Incur a performance cost because of the additional computation involved in execution
- Use this style when you have developed a program or some form of computation but have no make of machine to directly run it on

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)

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

Architectural Design Process

Pipes and filters

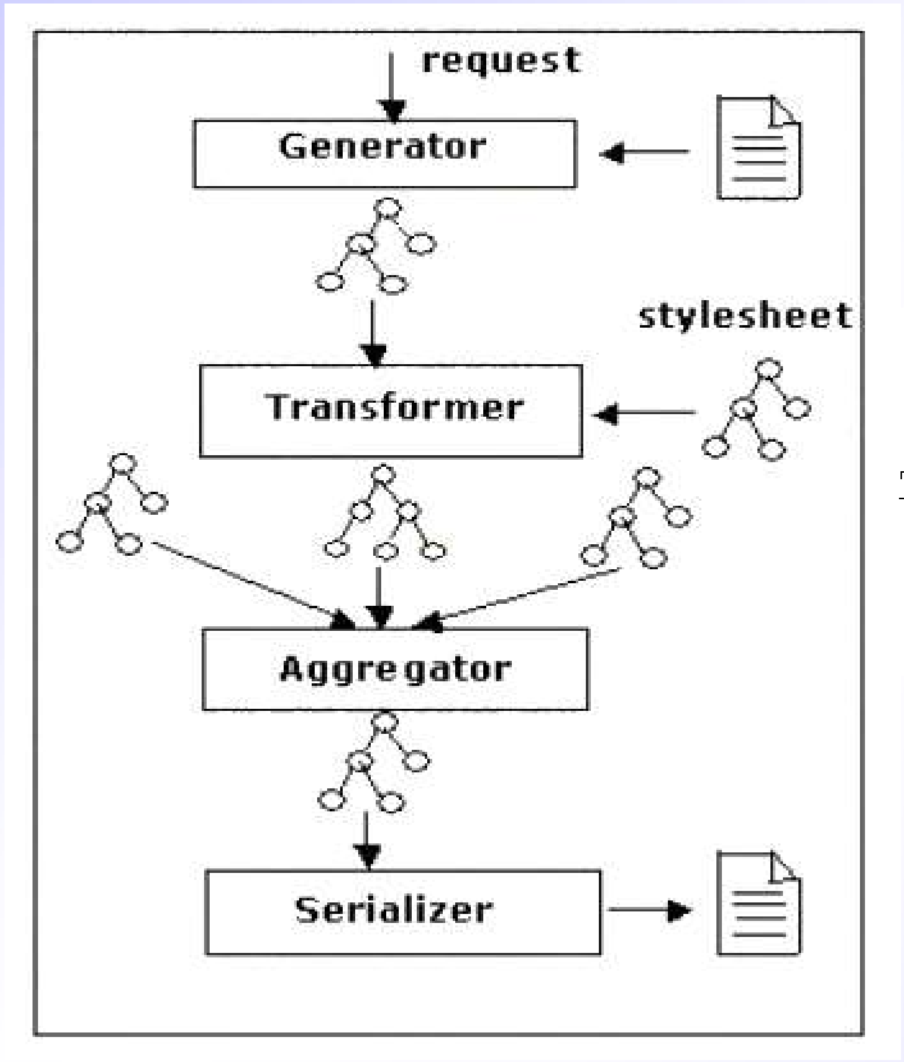
The Pipes and Filters architectural pattern [style] provides a structure for systems that process a stream of data.

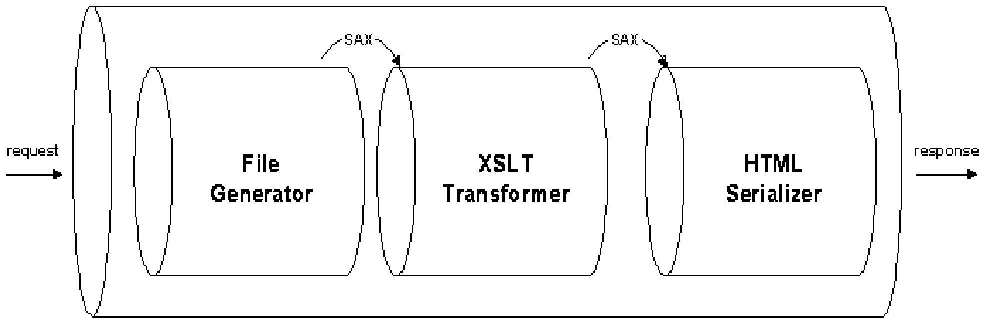
Each processing step is encapsulated in a filter component.

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

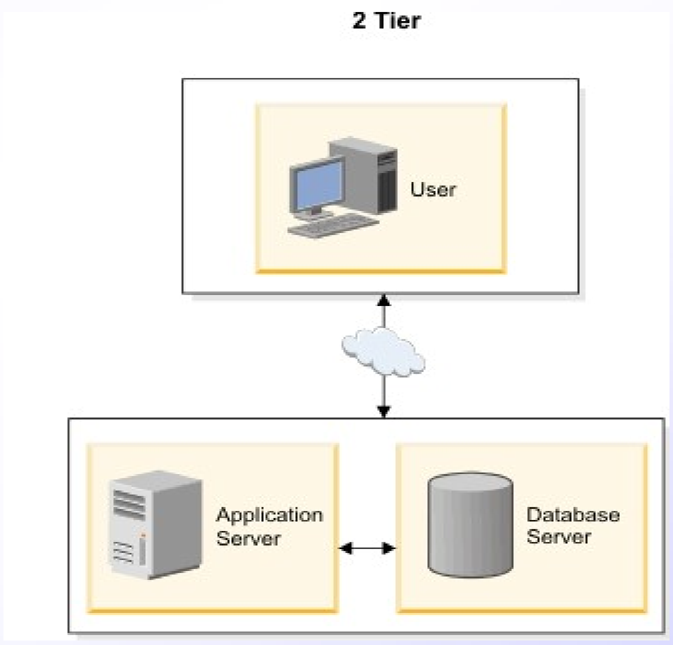




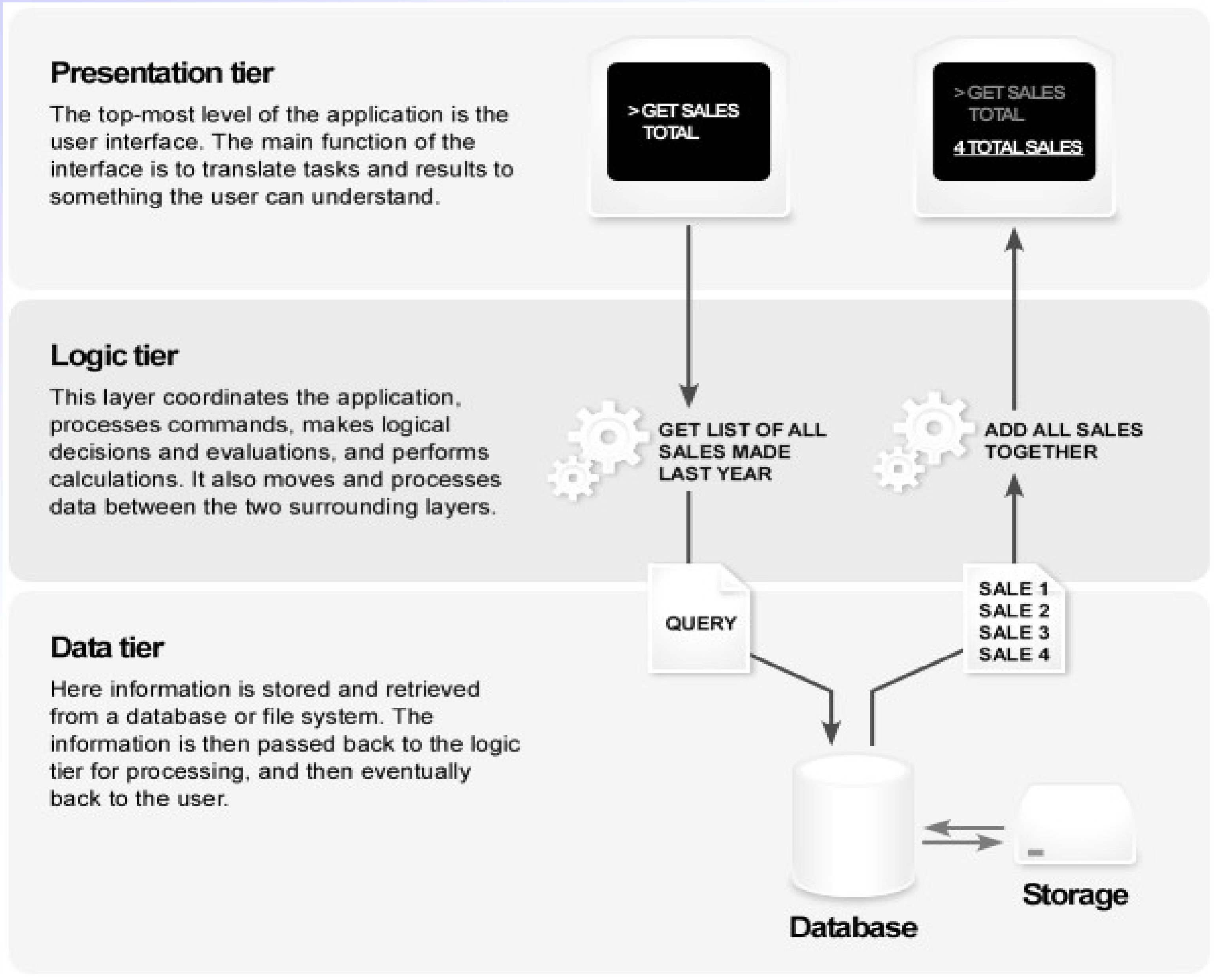
Tiered architecture (layering)

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.



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.

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.

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.

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.

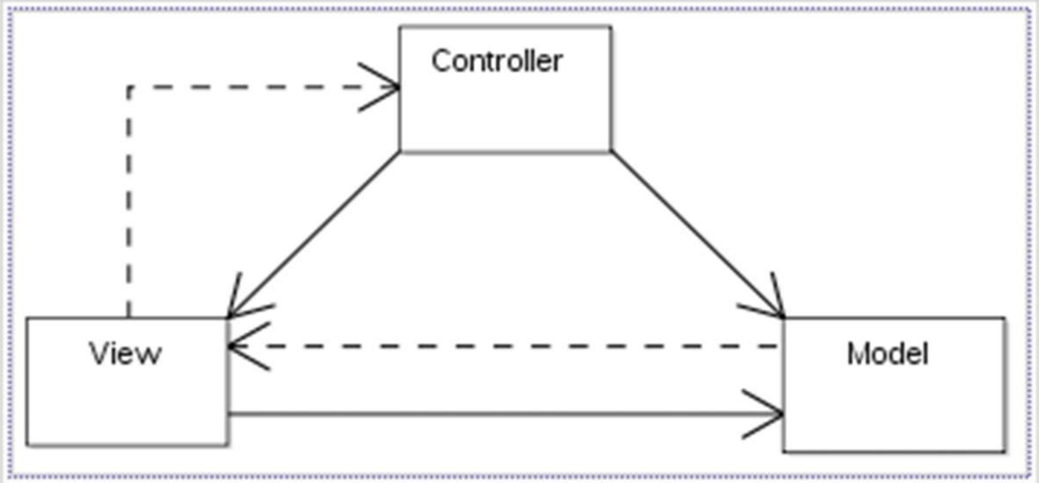
MVC was originally developed to map the traditional input, processing, output roles into the GUI realm:

Input → Processing → Output

Controller → Model → 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).