Software Architecture

COMP 1531

Aarthi Natarajan

The story of two telephone systems

Plain Old Telephone Service (POTS)

Functionality: Call subscriber

Architecture

- Analog telephone service implemented over copper twisted pair wires
- centralized hardware switch

Good qualities

- Reliable
- Works during power outages

Skype

Functionality: Call subscriber

Architecture

- Packet-switched telephone network over internet
- Peer to peer software

Good qualities

- High scalability
- Easy to add new features (e.g., video calling)

A software architect focuses on the essential qualities

Software Architecture

As software systems increase in size and complexity and become distributed

- Design problem extends beyond the algorithms and data structures of computation
- It becomes increasingly vital to specify the overall system structure

Software Architecture as a concept has its origins in the research of Edsger Dijkstra in 1968 and David Parnas in the early 1970s who emphasized that the structure of a software system matters and getting the structure right is critical

Software Architecture

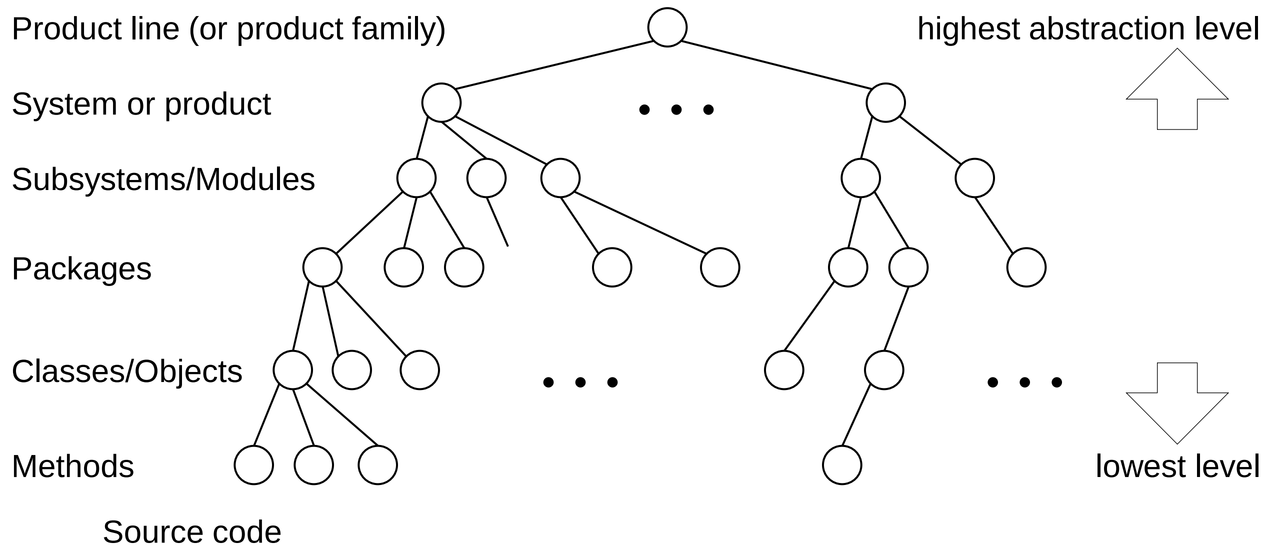
Simply, stated:

- Is the “big picture” or macroscopic organization of the system to be built
- Partition the system in logical sub-systems or parts, then provide a high-level view of the system in terms of these parts and how they relate to form the whole system

Architecture vs Design

- **Architecture** focuses on non-functional requirements ("cross-cutting concerns") and decomposition of functional requirements
- **Design** focuses on implementing the functional requirements
  - Note: The border is not always sharp!

Hierarchical Organization of Software



- Software is not one long list of program statements but it has **structure**
- But first, **why** do we want to decompose systems?

Why do we Decompose Systems (1)?

- Understanding and communication
  - Enables every one to understand how the system works as a whole (various stake-holders, users of the system, developers, architects)
  - Provides an understanding of the macro properties of the system to learn how the system intends to fulfil the key functional and non-functional requirements
  - Pre-determine key system properties (scalability, reliability, performance, usability etc..)
- Tackle complexity by "divide-and-conquer" – allow people to work on individual pieces of the system in isolation which can later be integrated to form the final system

Why do we Decompose Systems (2)?

- To prepare for extension of the system
  - Support flexibility and future evolution by decoupling unrelated parts, so each can evolve separately ("separation of concerns")
  - Subsystems envisioned to be part of a future release can be included in the architecture, even though they are not to be developed immediately.
- Focus on creative parts and avoid "reinventing the wheel"
  - Discover components obtained from past projects
  - Identify components that have high potential reusability

Some Key questions we are faced with:

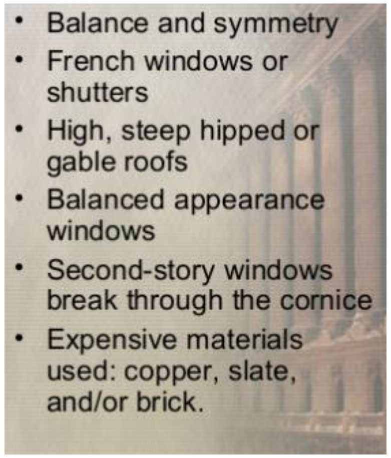
1. How to decompose the system (into parts)?
2. How the parts relate to one another?

← Architectural Style

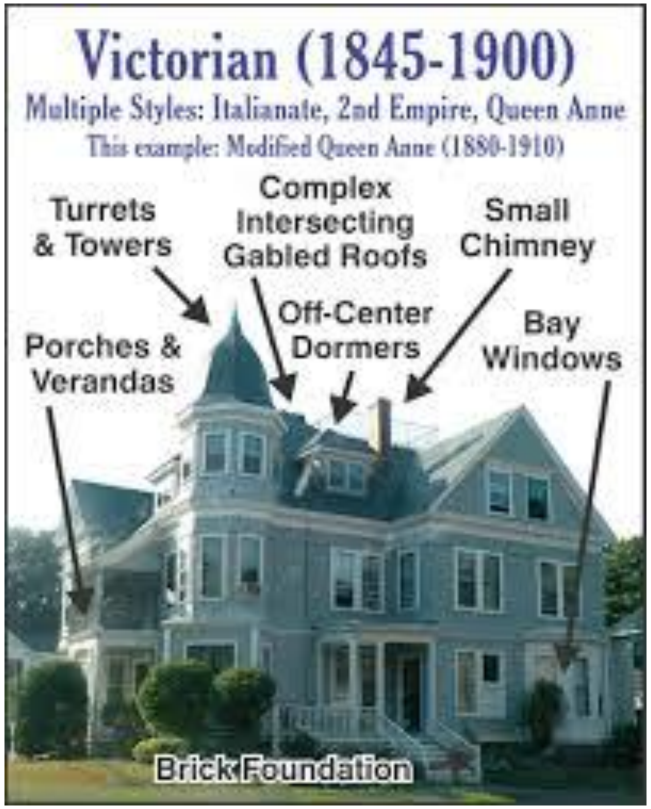
3. How to document the system's software architecture?

← Architectural Views

Architectural Styles



- Balance and symmetry
- French windows or shutters
- High, steep hipped or gable roofs
- Balanced appearance windows
- Second-story windows break through the cornice
- Expensive materials used: copper, slate, and/or brick.


Victorian (1845-1900)
Multiple Styles: Italianate, 2nd Empire, Queen Anne
This example: Modified Queen Anne (1880-1910)
Turrets & Towers
Complex Intersecting Gabled Roofs
Small Chimney
Porches & Verandas
Off-Center Dormers
Bay Windows
Brick Foundation

Software Architectural Styles

Formally introduced by Mary Shaw and David Garlan at Carnegie-Mellon University, mid-90s.

They defined a **software architectural style** as:

"a family of systems in terms of a pattern of structural organisation"

Patterns were described as abstract representations with an aim to:

- Make it easy to communicate among stakeholders
- Document early design decisions
- Allow for the reuse

Basically, an architectural style is defined by:

1. Components
   - A collection of **computational units** or elements that “do the work” (e.g., classes, databases, tools, processes etc.)
2. Connectors
   - Enable communication between different components of the system (e.g., function call, remote procedure call, event broadcast etc.,) and uses a specific **protocol**
3. Constraints
   - Define how the components can be combined to form the system
     - define where data may flow in and out of the components/connectors
     - topological constraints that define the arrangement of the components and connectors

How to Fit Subsystems Together: Some Well-Known Architectural Styles

- Client/Server
  - (2-tiered, n-tiered or multi-tiered)
  - World Wide Web style
    REST (Representational State Transfer)
- Peer-to-Peer
- Pipe-and-Filter
  - UNIX shell script architectural style
- Central Repository (database)
- Layered

Problem Context:

Problem Context:

- How to share data between a client and a service provider distributed geographically across different locations?

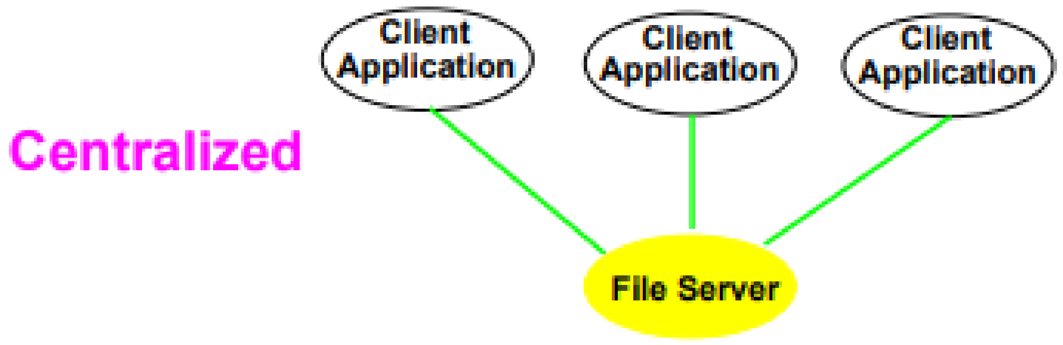
Architectural Style:

- Client-Server Architecture

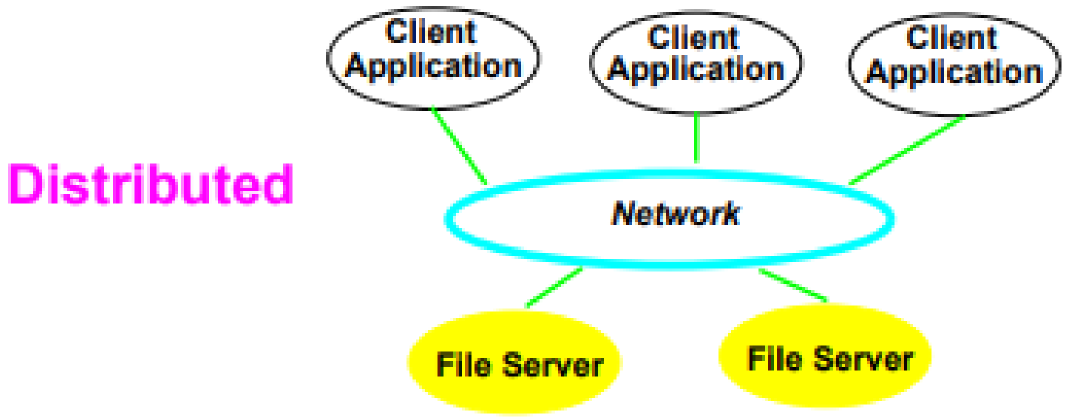
Client-Server Architecture

- Basic architectural style for distributed computing
- Two distinct component types:
  - One component that has the role of a server that provides specific services, waits for and handles connections e.g., database or file server
  - One component that has the role of a client that initiates connections to request services provided by a server
  - Client and Server could be on same machine or different machines
- Connector is based on a **request-response** model
- Examples
  - File Server, Database server, Email Server

File Transfer with FTP: An example of a 2-tier client-server architecture

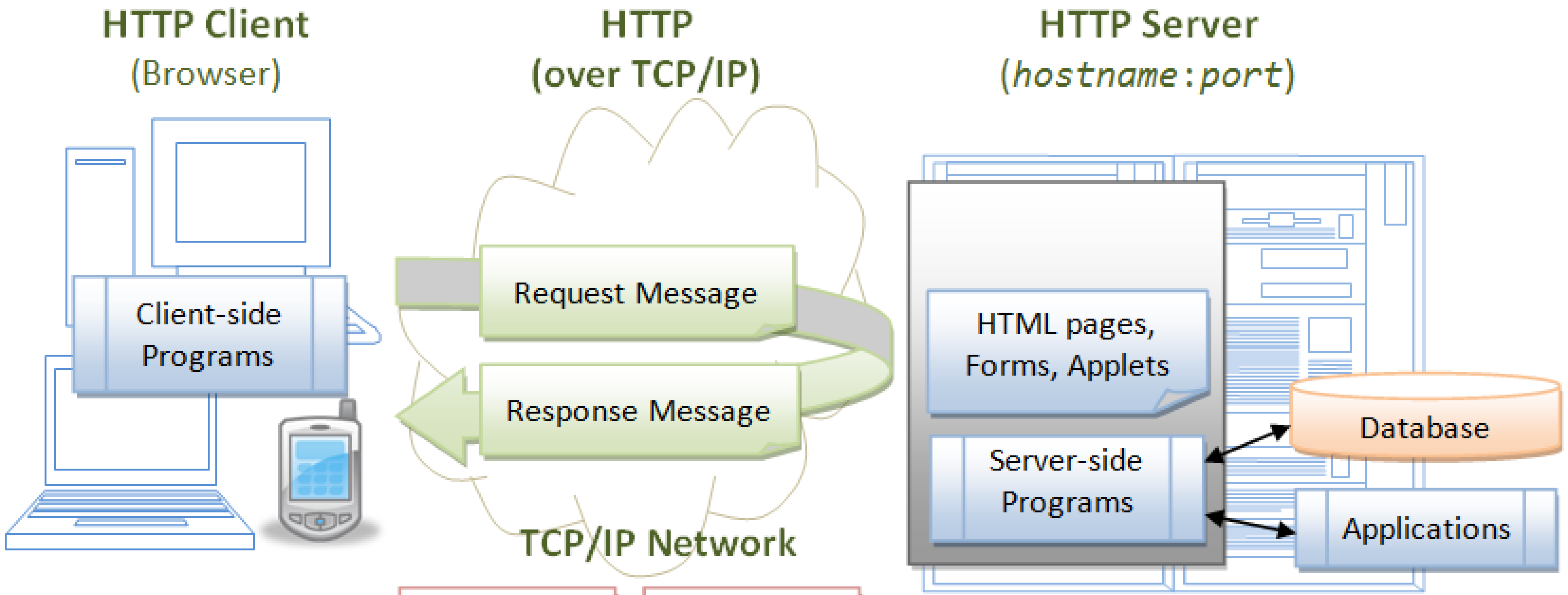


- *The client passes requests to the file server (software) for file records*
- *Clients can reside in the same machine or separate machines (typically PCs)*
- *Requests can be either local or over a network*
- *Indispensable for documents, images, drawings, and other large data objects*



© https://www.utdallas.edu/~chung/SA/2client.pdf

Web Client-Server Architecture



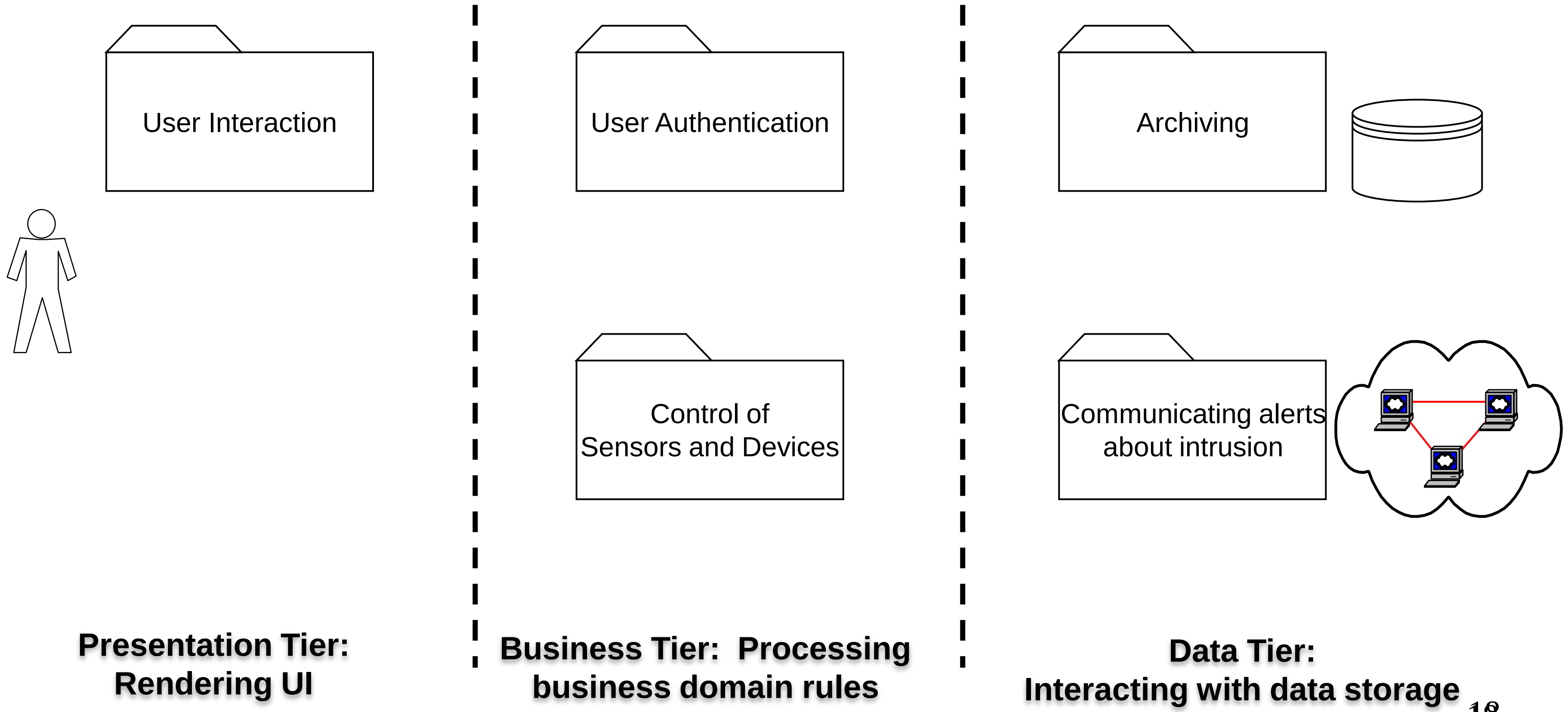
Application	**HTTP**	
Presentation	SSL	
Session		
Transport	**TCP**	Multiplexing (Port), Re-transmission
Network	**IP**	Addressing (IP Address), Routing
Data Link	IEEE 802.11x	
Physical		

https://www.ntu.edu.sg/home/ehchua/programming/howto/Tomcat_HowTo.html

3-Tier / N-Tiered Architecture

- Separates the **deployment** of software components into multiple logical layers, so that each tier can be located on a physically separate computer

e.g., a **3-Tier architecture** is typically decomposed into:


User Interaction
User Authentication
Archiving
Control of Sensors and Devices
Communicating alerts about intrusion
Presentation Tier: Rendering UI
Business Tier: Processing business domain rules
Data Tier: Interacting with data storage

Benefits:

- Makes effective use of network and distribution of data is straightforward
- Roles and responsibilities of system distributed among several independent machines
- Easy to add new servers or upgrade existing servers
- Deployment of modules to different servers enhancing security, scalability and performance

Weakness:

- Single point of failure – when a server is down, client requests cannot be met
- Network congestion, when large number of client requests are made simultaneously to the server
- Complex and expensive infrastructure

Problem Context:

- How do we resolve network congestion and single point of failure that could result in a client-server model?

Architectural Style:

- Peer-to-Peer (P2P)

IF NAPSTER SHUTS DOWN, WE WON'T BE ABLE TO ACCESS MP3 FILES FROM A HARD DRIVE DATABASE AND DOWNLOAD THEM TO BURN A NEW CD! WE'LL HAVE TO GET OUR MUSIC THE OLD-FASHIONED WAY!
YEAH! SHOPLIFTING!
KEVIN SIERS © 2001 THE CHARLOTTE OBSERVER

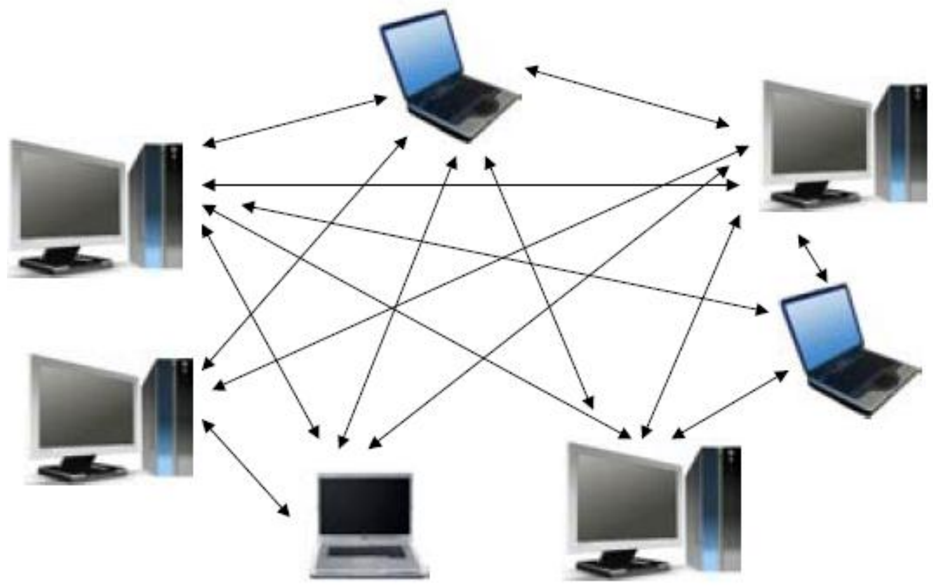
22

Peer-to-Peer (P2P) architecture

- Each peer component can function as both a server and a client
- Information distributed among all peers and any two components can set up a communication channel to exchange information as needed

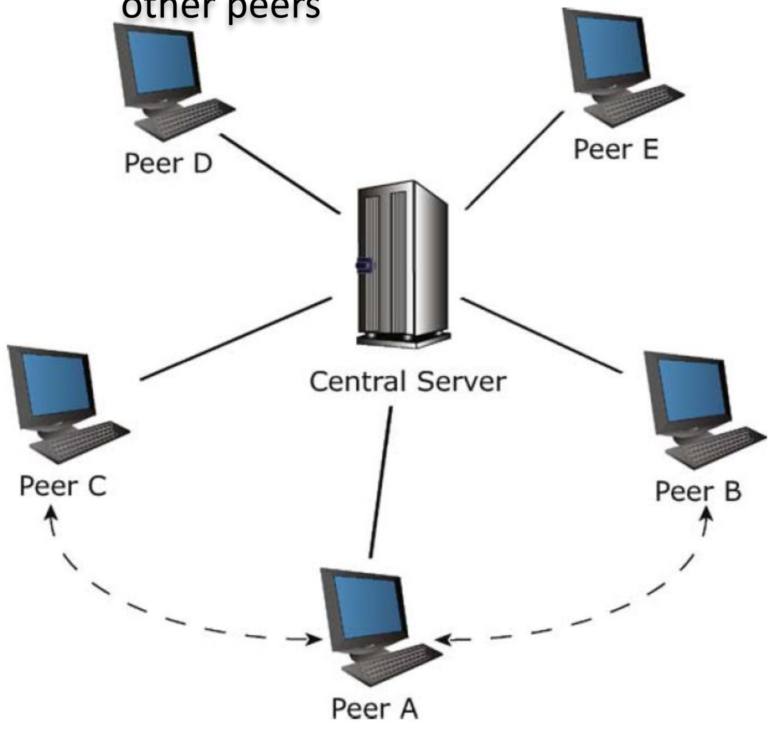
Pure P2P Architecture

(No central Server, No central router)



Hybrid P2P Architecture

Central server keeps information on peers and helps peers to locate other peers

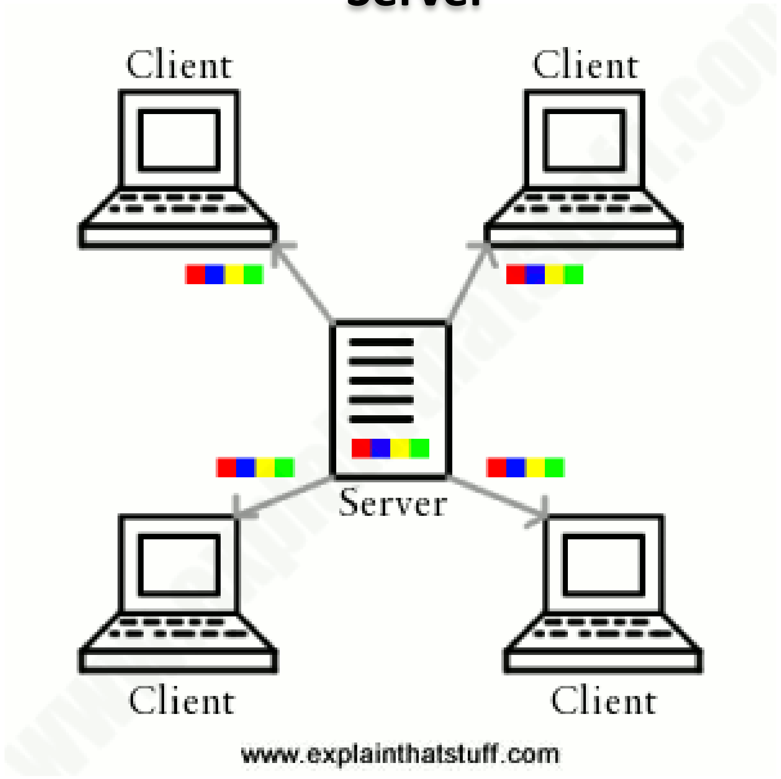


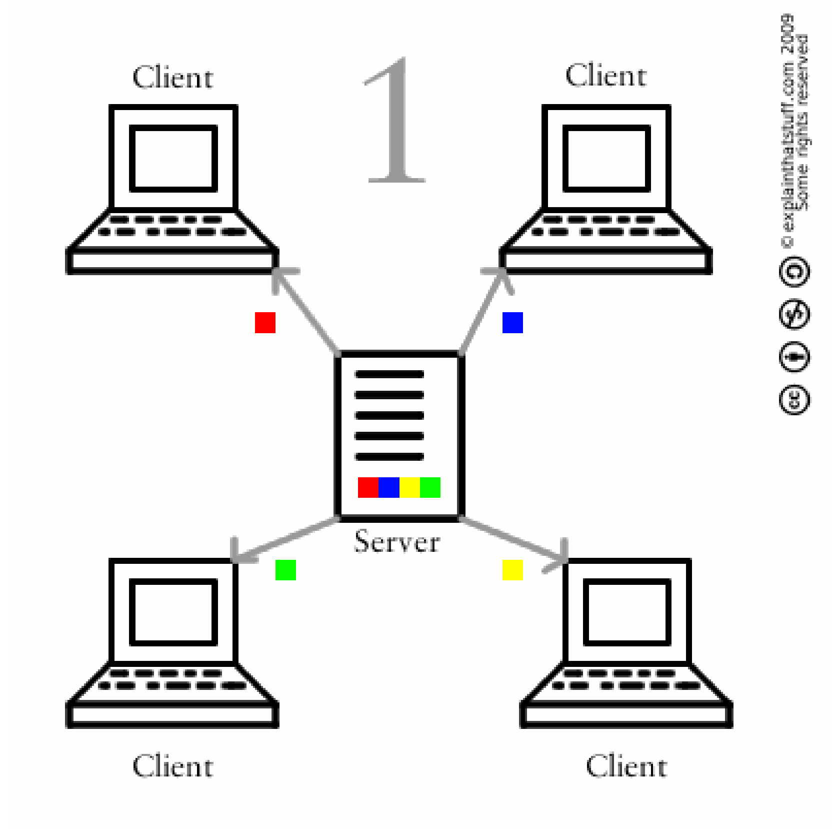
Example: Bit Torrent

Downloading with Client-Server

VS

Peer-to-Peer Torrenting





Other Examples of P2P architecture

- Napster - it was shut down in 2001 since they used a centralized tracking server
- Skype - it used to use proprietary hybrid P2P protocol, now uses client-server model after Microsoft’s acquisition
- Bitcoin - P2P cryptocurrency without a central monetary authority

Benefits:

- Efficiency: More efficient as all clients provide resources
- Scalability: Unlike client-server, capacity of the network increases with number of clients
- Robustness:
  - Data is replicated over peers
  - immune to single point of failure, e.g., if a node failed to download a file, the remaining nodes still have the data needed to complete the download

Weakness:

- Architectural complexity
- Distributed resources are not always available
- More demanding of peers

Problem Context:

- How to design a system that is suitable for processing and transforming data streams?
- Goal – to produce an output stream by suitably transforming the input data stream

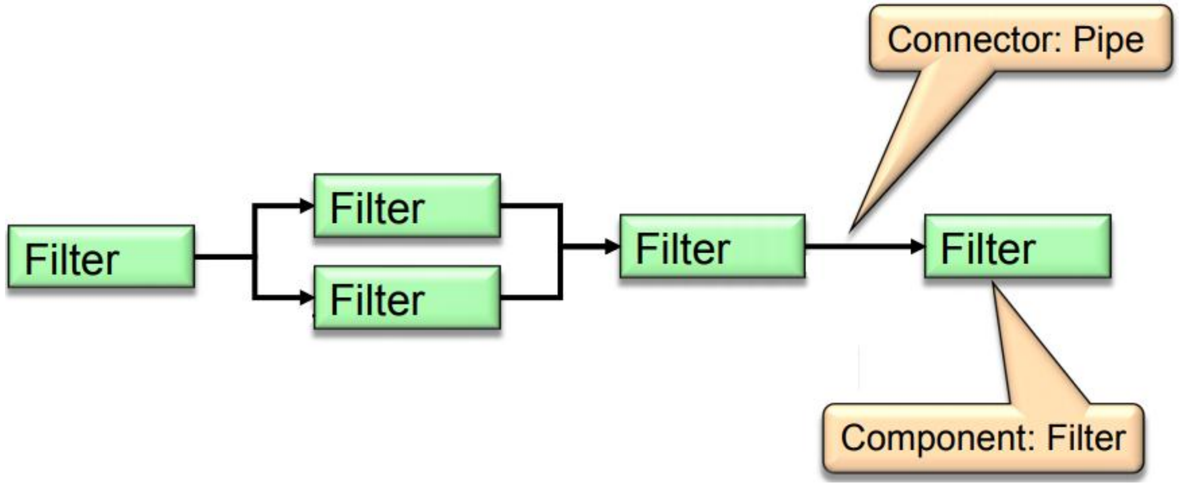
Architectural Style:

- Pipes and Filters

Pipe-and-Filter Architectural Style

- Component: **Filter**
  - Reads input stream (or streams)
  - Each filter encapsulates a data processing step to transform data and produce output stream (or streams)
- Connector: **Pipe** (Serving as a conduit for data)

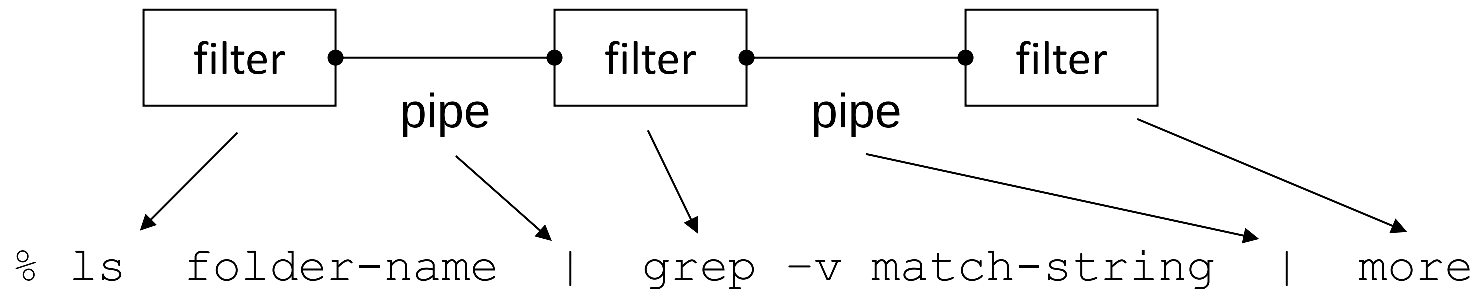
  Data transformed by one filter is sent other filters for further processing using the pipe connector



Pipe-and-Filter Architectural Style

- Data processed incrementally as it arrives
- Output can begin before input fully consumed
- Filters must be independent:
  - Filters do not share state
  - Filters do not know upstream or downstream filters

e.g., UNIX shell commands and pipes, compilers



```
% ls  folder-name  |  grep -v match-string  |  more
```

Benefits:

- Easy to understand the overall input/output behaviour of a system as a simple composition of the behaviours of filters
  - Different data processing steps are decoupled so filters can evolve independently of each other
- Flexible and support reuse - any two filters can be recombined, if they agree on data formats
- Flexible and ease of maintenance - filters can be recombined or easily replaced by new filters
- Support concurrent processing of data streams

Weakness:

- Highly dependent on order of filters - What if an intermediate filter crashes?
- Not appropriate for interactive applications

Problem Context:

- A complex body of knowledge, that needs to be established, persisted and manipulated in several ways

Architectural Style:

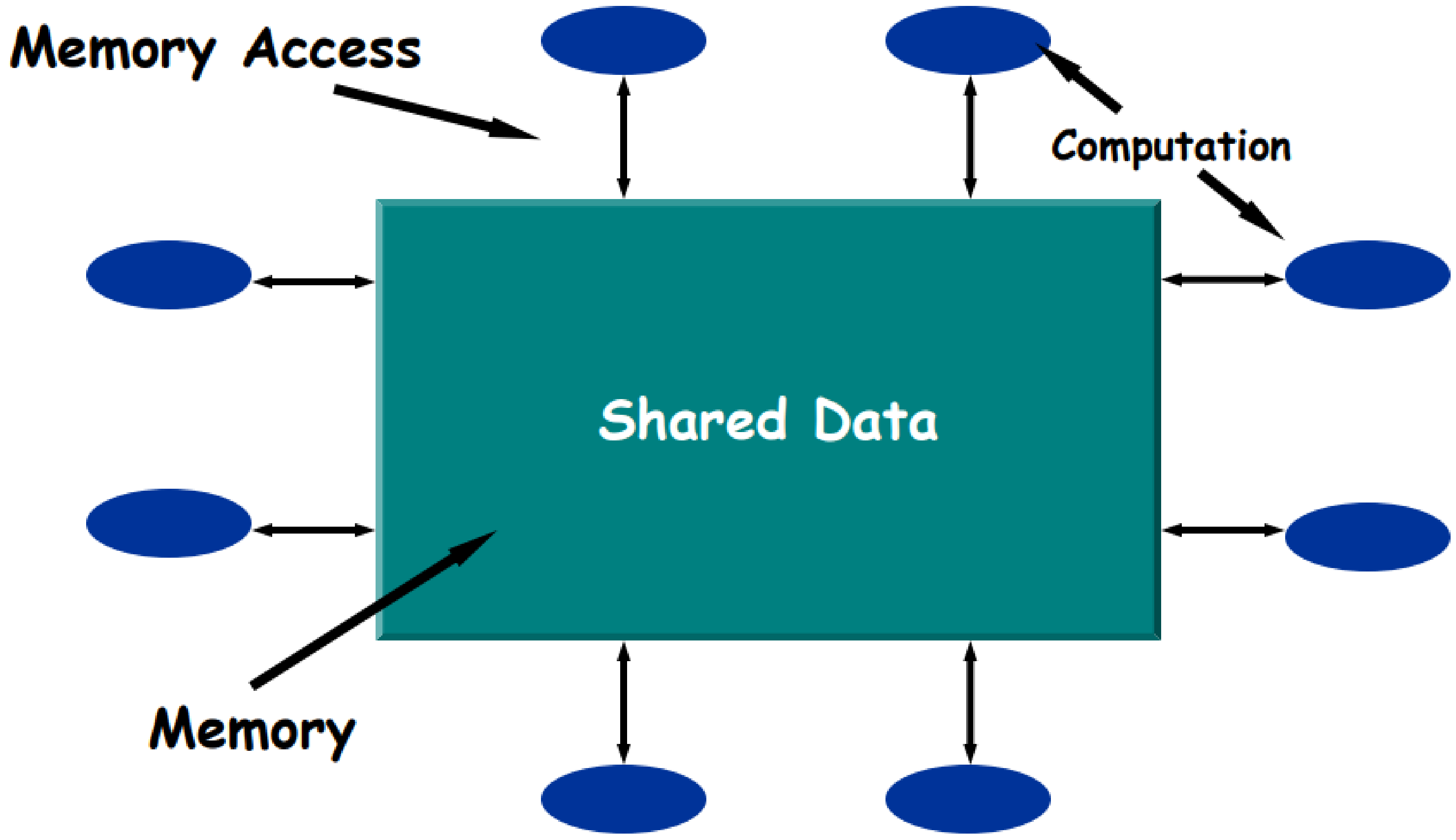
- Repository Style

Central Repository Architectural Style (or Shared Data Style)

- Components : Two distinct types
  - Central data repository - A central, reliable, permanent data structure that represents the state of the system
  - Data accessors – A collection of independent components that operate on the central data; components do not interact directly but only through the central repository
- Connectors:
  - Read/Write mechanism (e.g., procedure calls or direct memory accesses)

- Components access repository as and when they want
- Knowledge sources , do not change the shared data

e.g., graphical editors, IDEs, database applications, document repositories



Benefits:

- Efficient way to share large amounts of data
- Centralised management of the repository
  - Concurrency access and data integrity
  - Security
  - Backup
- Components need not be concerned with how data is produced

Weakness:

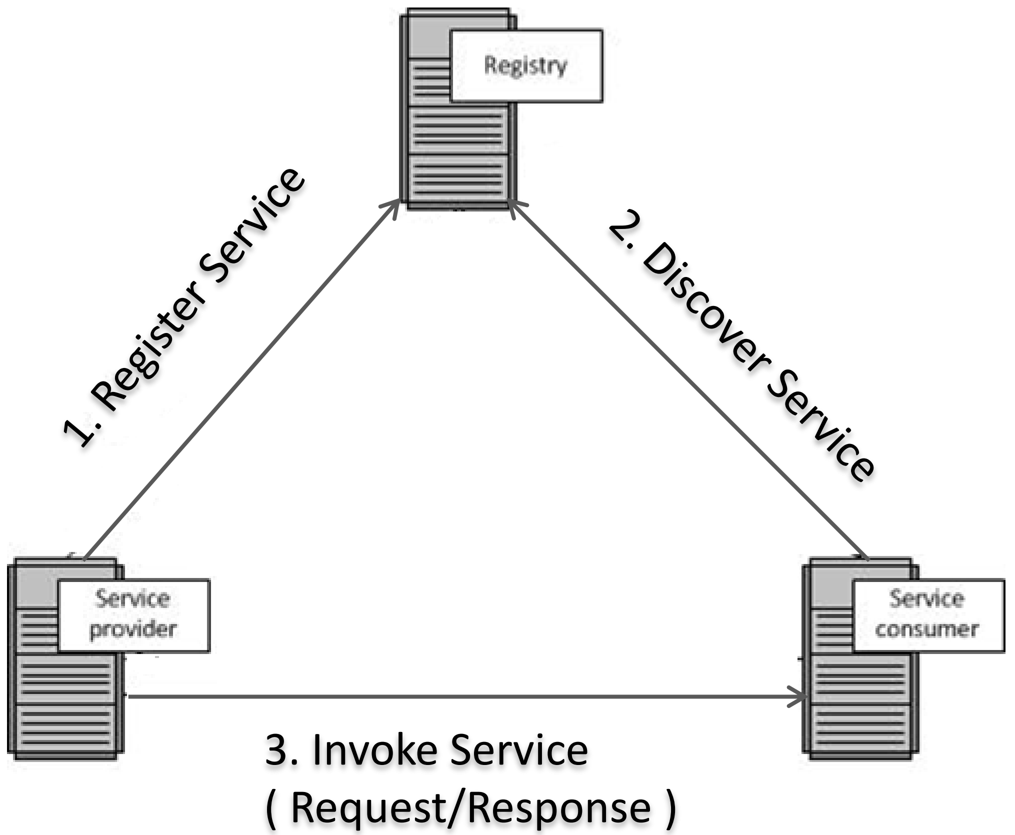
- All independent components **must agree** upon a repository data model a priori.
- Distribution of data can be a problem
- Connectors implement complex infrastructure

Some other styles (1): Publish-subscribe style (Event based style)

- Two distinct component types
  - Publisher: Components that generate or publish events
  - Subscribers: Components register their interest in published events and are invoked when these events occur
- Few examples
  - user interface frameworks
  - traders register interest (subscribe to) particular stock prices and receive updates as prices changes
  - wireless sensor networks

Some other styles (2): SOA (Service Oriented Architecture)

- Software components are created as as autonomous, platform-independent, loosely coupled *services* and is agnostic of the underlying implementation technology
- One type of service - web service
- applications – Business-to-Business (B2B) services

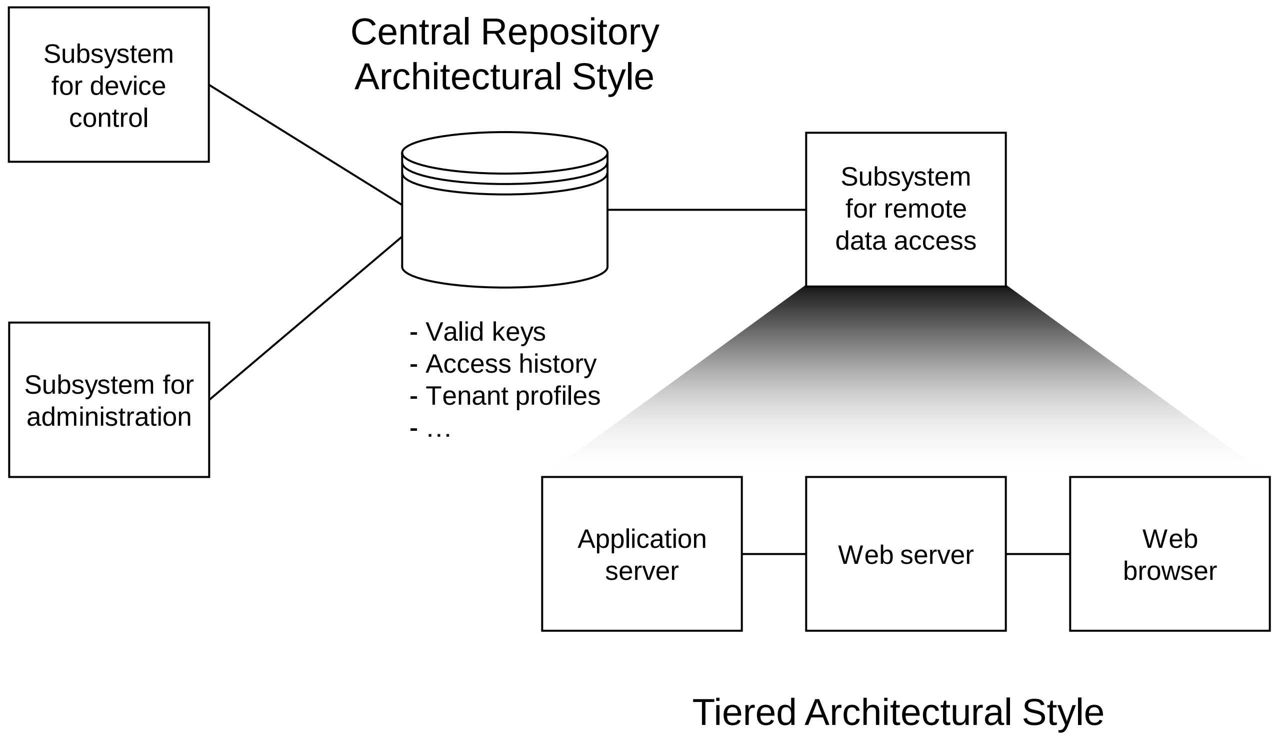


Architectural Decisions often involve Compromise

- Every system has an architecture

  .. But not every architecture suits the application
- The "best" design for a system:
  - Depends on what criteria are used to decide the "goodness" of a design
  - System requirements (functional needs, quality needs (e.g. performance, security)
  - Business priorities (alignment to requirements)
  - Available resources, core competences, target customers, competitors' moves, technology trends, existing investments, backward compatibility, …
- Hard to change architecture later…
  - Does not mean BDUF
  - But, need to think "enough"

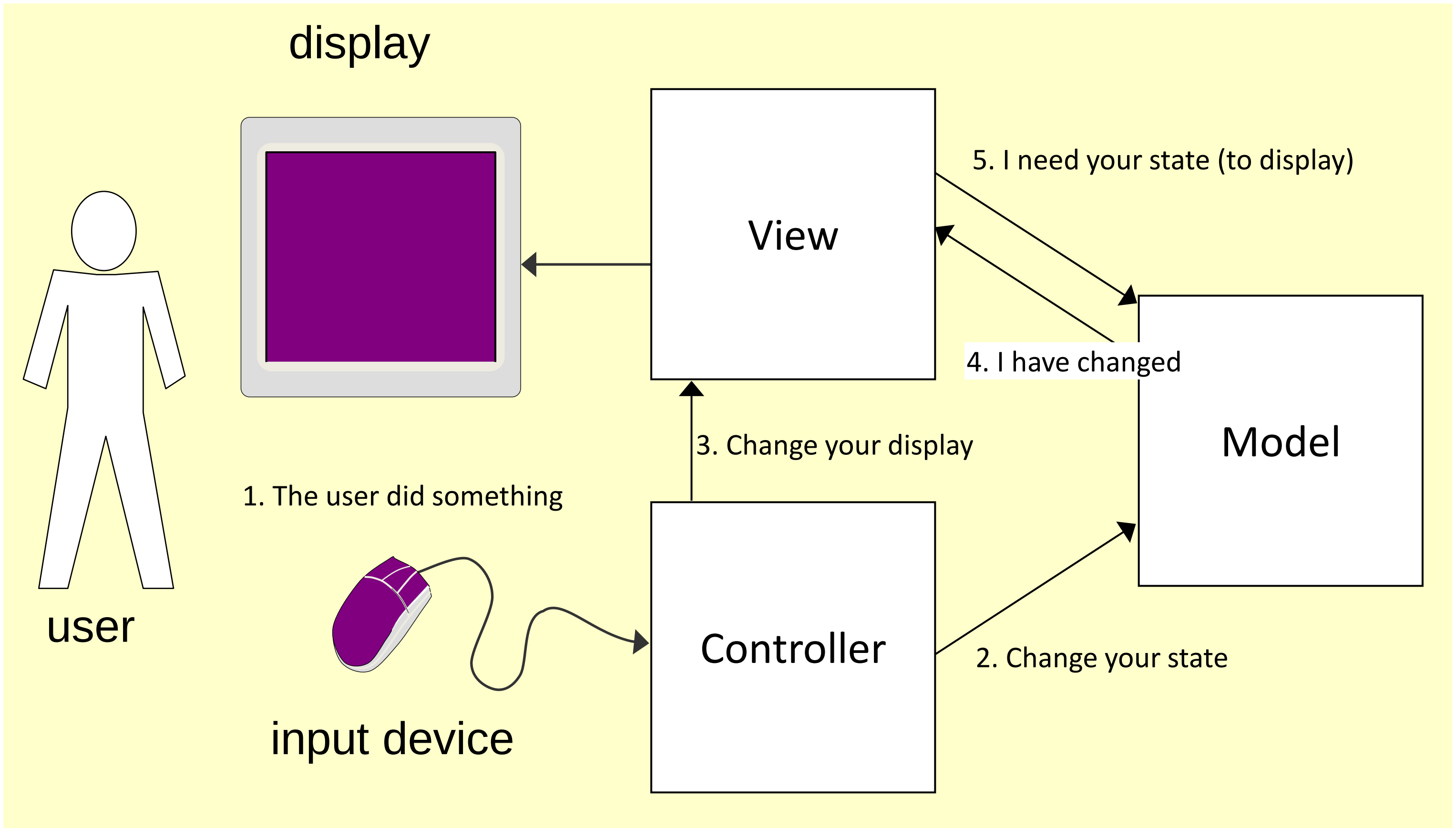
Real System is a Combination of Styles


Central Repository Architectural Style
Subsystem for device control
Subsystem for administration
Subsystem for remote data access
- Valid keys
- Access history
- Tenant profiles
- …
Application server
Web server
Web browser
Tiered Architectural Style

Revisiting MVC – an Application Architectural Pattern

Decouple data access, application logic and user interface into three distinct components

MVC is an application architectural pattern, not a software architectural style



We have seen various architectural styles, but how do we document the system's software architecture?

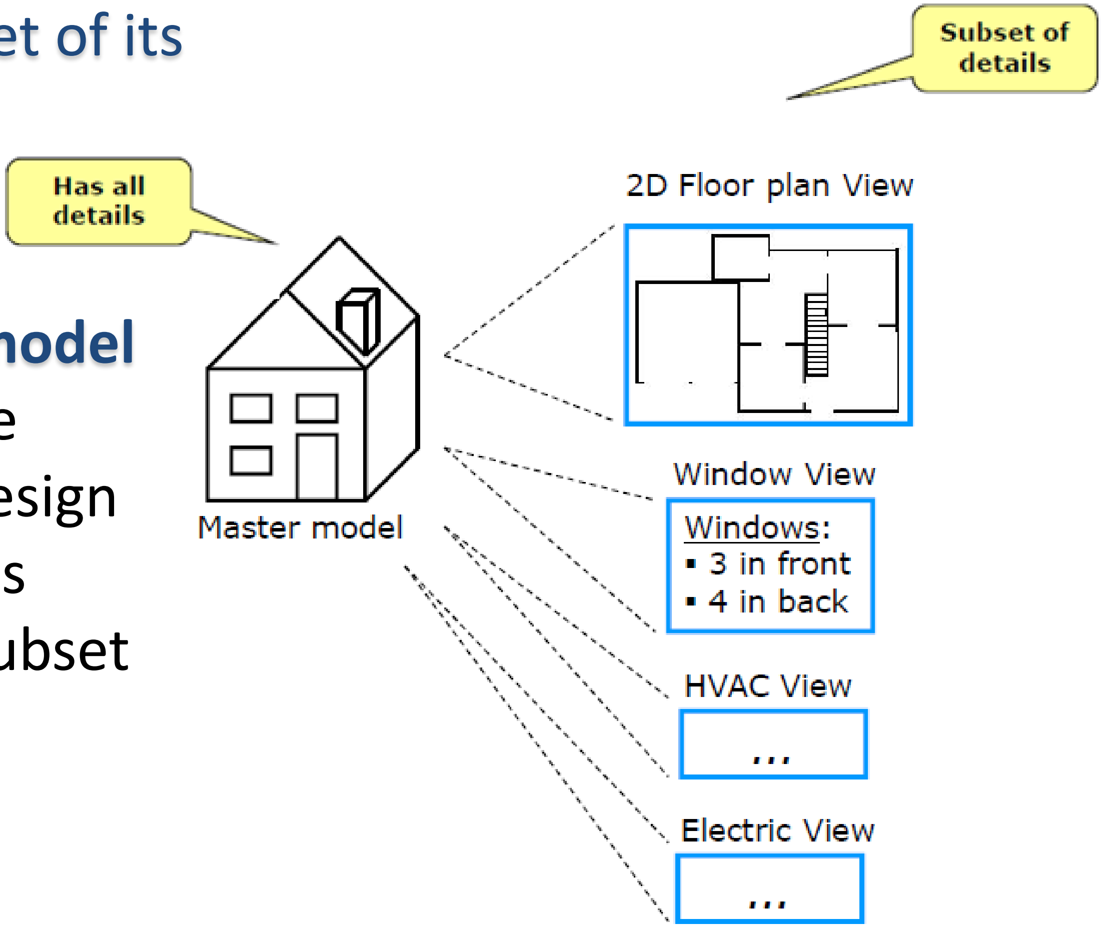
- Architectural Views

What is a view?

Definition: A **view** is a **projection** of a model showing a subset of its details

Projects from the **master model**

- Master model has all the details i.e. the master design
- Views are projects of this master model i.e., the subset of information



Architecture & Design Views (1)

A system's architecture may need to be expressed in terms of several different views e.g.,

- Model view
- Component and Connector view
- Allocation view

Architecture & Design Views: Model View

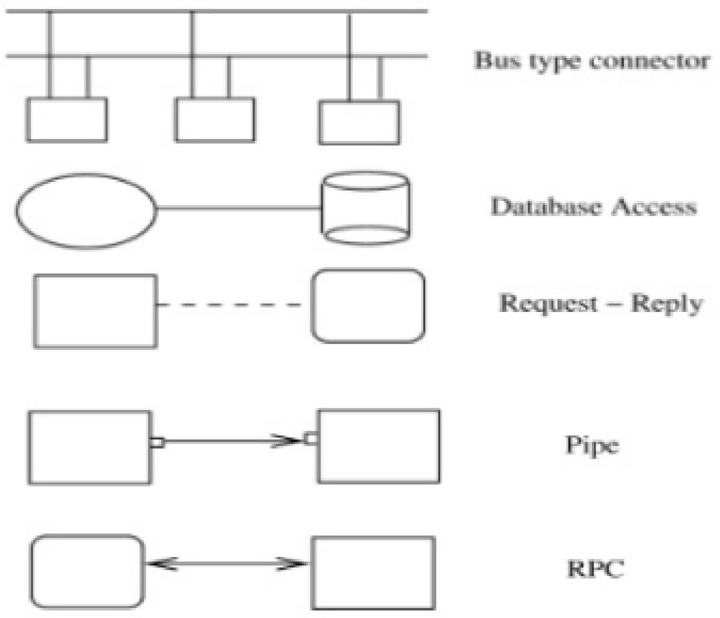
- Model view: Decomposes the functionality into a coherent set of software (code) units
  - Logical break-down into sub-systems often shown using package diagrams
  - Interaction among components at run-time shown using sequence diagrams, activity diagrams
  - Data shared between low-level components and collaborations typically expressed as UML class diagrams

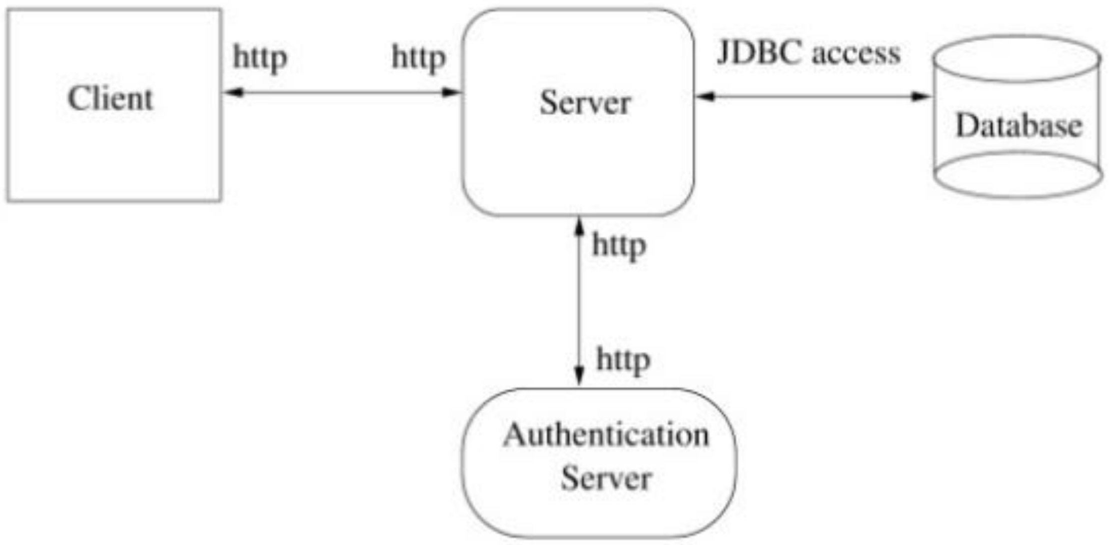
Architecture Views: Component and Connector View

- Describes a runtime structure of the system (components, data stores, connectors (e.g. pipes, sockets that enable interaction between components)
- Box-and-line diagram (informal), UML component diagram (formal)

Component and Connector View: Informal Box-Line diagram

- An informal approach to creating a component and connector view
- Often drawn on a white-board

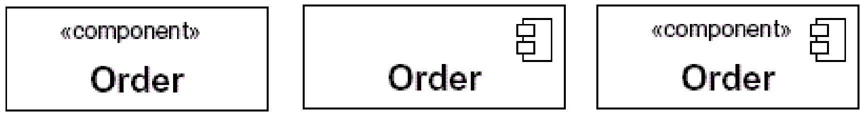




Component & Connector View: UML 2 component diagram

Use a formal notation to visualise the static implementation of a system and understand the wiring of the various components of the system, which includes:

Component: An independent, autonomous unit within a system or sub-system that represents a software module of classes, web service or some software resource, typically implemented as a replaceable module and can be deployed independently



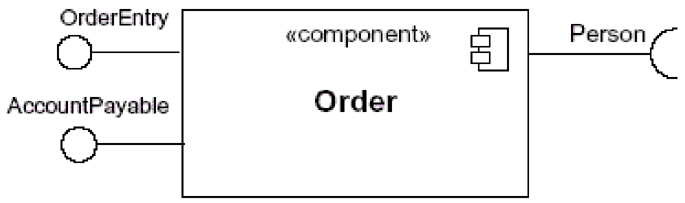
A rectangle with a visual stereotype in the top right corner or a textual stereotype of <<component>> or both

Component Interfaces:

A UML 2 component defines two sets of interfaces:

Provided interfaces : a collection of one or more methods that define the services offered by the component and represent a formal contract with a consumer

Required interfaces : dependency services required by the component in order to perform its function



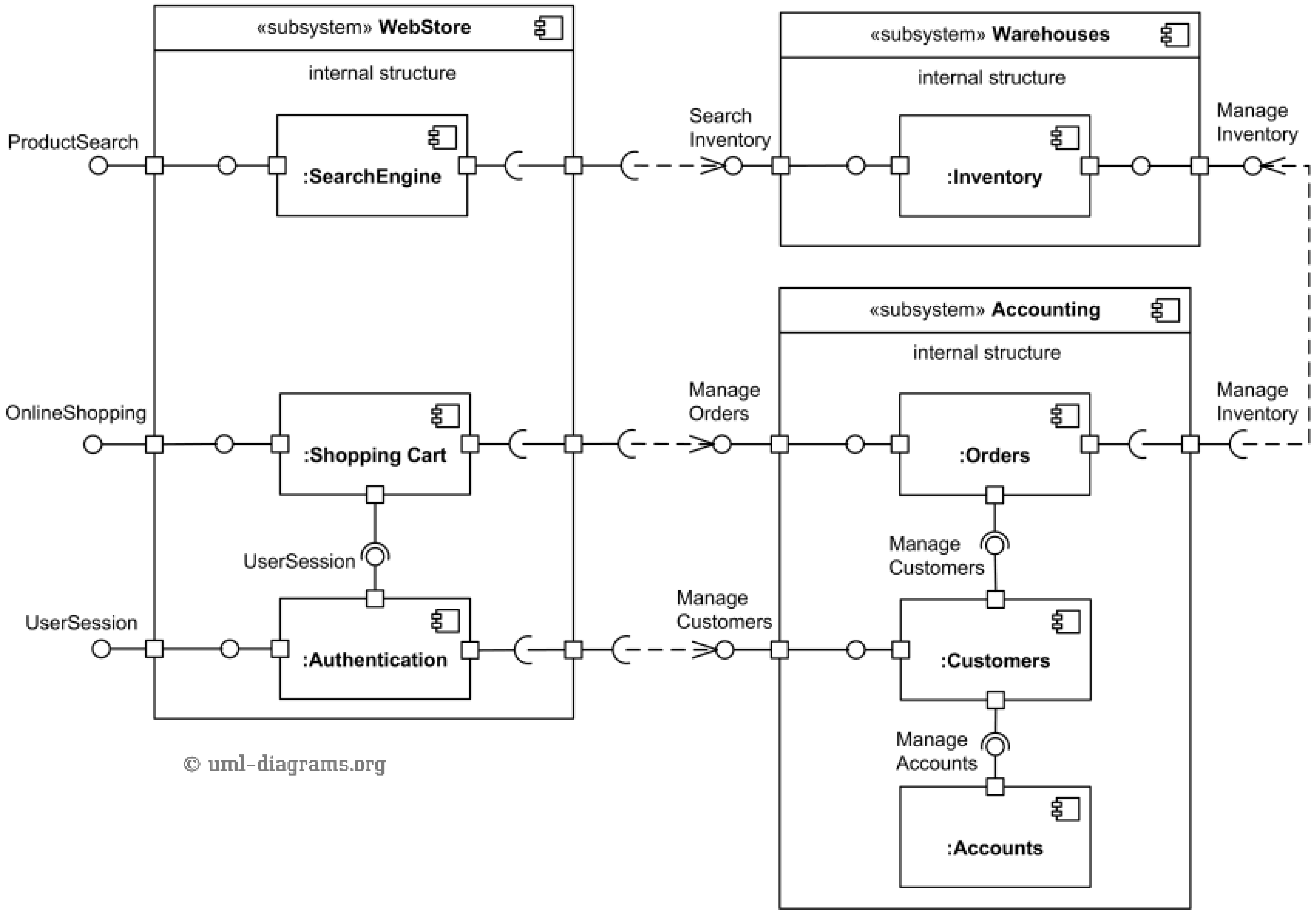
Provided interfaces: OrderEntry, AccountPayable

Lollipop notation (straight line with an attached circle)

Required interfaces: Person

(A straight line with a half circle)

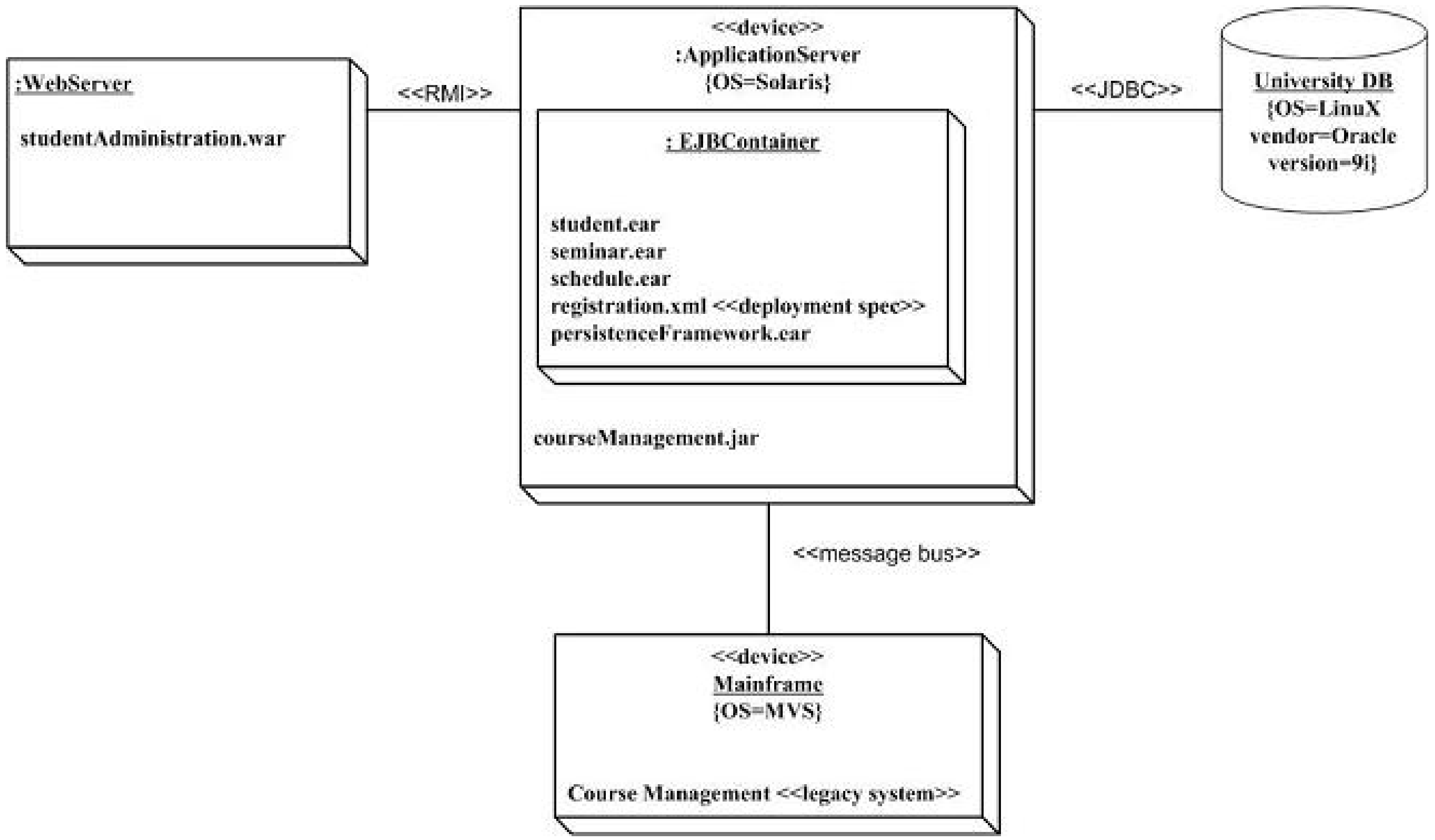
A UML 2 component diagram for an online shopping application



Architecture Views: Allocation View

- Describes how the software units map to the environment (hardware resources, file-systems and people)
- Exposes properties like which process runs on which processor
- Typically expressed as a UML deployment diagram
  - A static view of the run-time configuration of the processing nodes and the components that run on these nodes
  - show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.
  - Ideal for applications deployed to several machines e.g., thin-client network computer which interacts with several internal servers behind your corporate firewall

Example of a UML 2 deployment diagram


:WebServer
studentAdministration.war
<<RMI>>
<<device>>
:ApplicationServer
{OS=Solaris}
: EJBContainer
student.ear
seminar.ear
schedule.ear
registration.xml <<deployment spec>>
persistenceFramework.ear
courseManagement.jar
<<JDBC>>
University DB
{OS=LinuX
vendor=Oracle
version=9i}
<<message bus>>
<<device>>
Mainframe
{OS=MVS}
Course Management <<legacy system>>