Software Architecture

COMP 1531, 17s2

Aarthi Natarajan

Week 7

As software systems increase in size and complexity and become distributed

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

Software Architecture

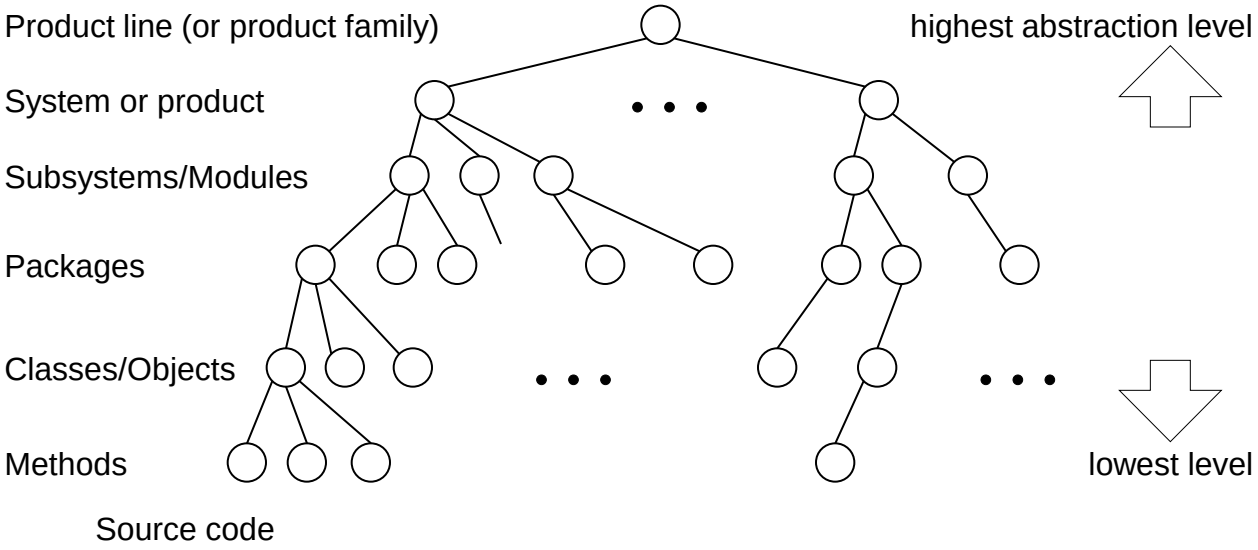
- As a concept has its origins in the research of Edsger Dijkstra in 1968 and David Parnas in the early 1970s.
- These scientists emphasized that the structure of a software system matters and getting the structure right is critical

Software Architecture

"The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them"

- 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

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)?

- Tackle complexity by "divide-and-conquer"
- See if some parts already exist & can be reused
- Focus on creative parts and avoid "reinventing the wheel"
- Support flexibility and future evolution by decoupling unrelated parts, so each can evolve separately ("separation of concerns")

Why do we Decompose Systems (2)?

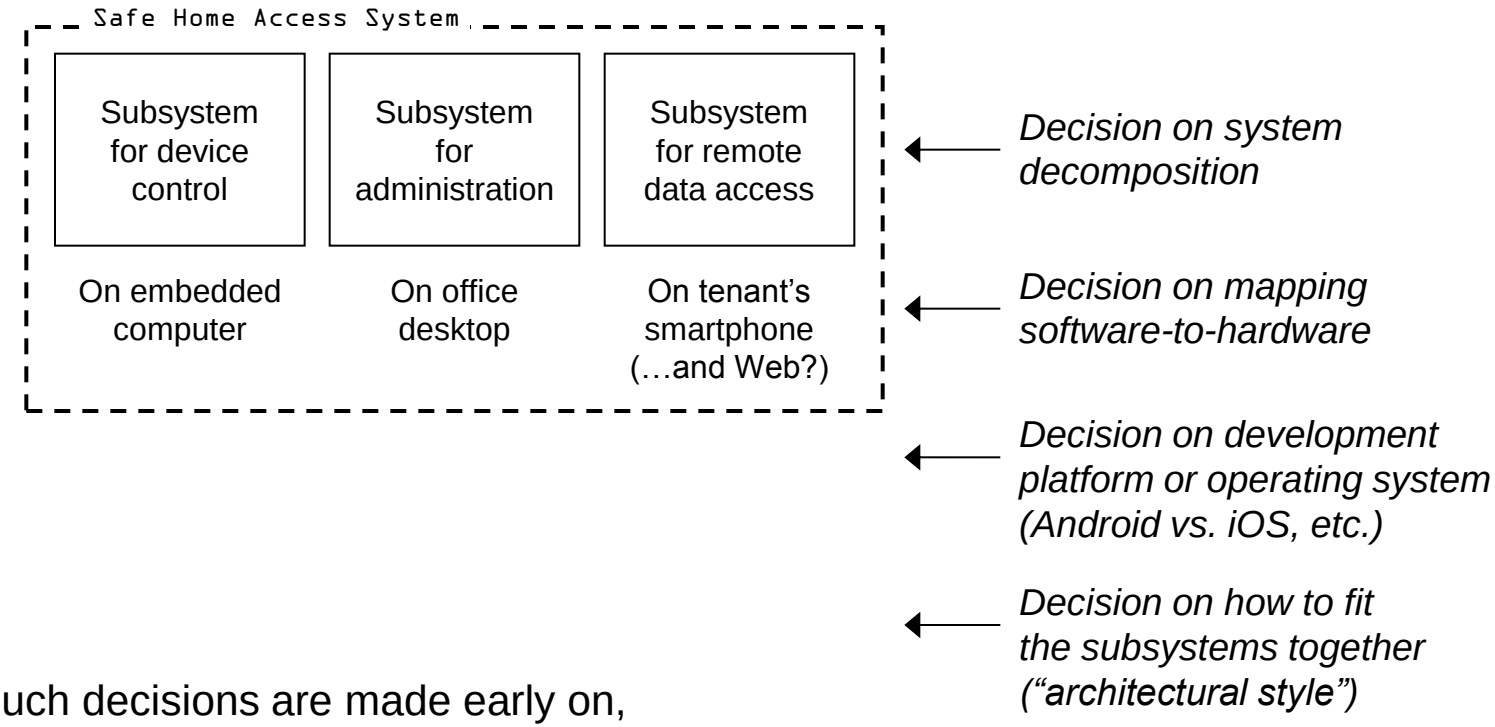
- Highly desirable if key system properties are pre-determined
  - Security, Reliability, Performance
  - Reusability, Extensibility, Maintainability
  - Coupling, Cohesiveness
  - Usability, Compatibility
  - Cost
- Understanding and communication
  - communicate stakeholders, end-users, clients, developers, architects.
  - Provide an understanding of the macro properties of the system and how the system intends to fulfil the key functional and non-functional requirements

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!

Example Architectural Decisions

Example decisions:



Such decisions are made early on,

perhaps while discussing the requirements with the customer

to decide which hardware devices will be used for user interaction and device control

Formally,

Software Architecture =

a set of high-level **decisions** that determine the structure of the solution
(parts of system-to-be and their relationships)

- Principal decisions made throughout the development *and* evolution of a software system
- made early and affect large parts of the system (“design philosophy”) — such decisions are hard to modify later

Architectural Styles (Building)

- 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.

Some Key questions we are faced with:

1. How to decompose the system (into parts)?
2. How the parts relate to one another?
3. How to document the system's software architecture?

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
   - Processing elements that “do the work” (e.g., classes, databases, tools, processes etc.)
2. Connectors
   - Enable communication among different components (e.g., function call, remote procedure call, event broadcasts 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

Architectural Decisions often involve Compromise

- The "best" design for a component considered in isolation may not be chosen when components considered together or within a broader context
  - Depends on what criteria are used to decide the "goodness" of a design
  - E.g., car components may be "best" for racing cars or "best" for luxury cars, but will not be best together
- Additional considerations include business priorities, available resources, core competences, target customers, competitors' moves, technology trends, existing investments, backward compatibility, …

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)
- Pipe-and-Filter
  - UNIX shell script architectural style
- Central Repository (database)
- Layered
- Peer-to-Peer
- Model-View-Controller

Development platform (e.g., Web vs. mobile app, etc.) may dictate the architectural style or vice versa…

Problem Context:

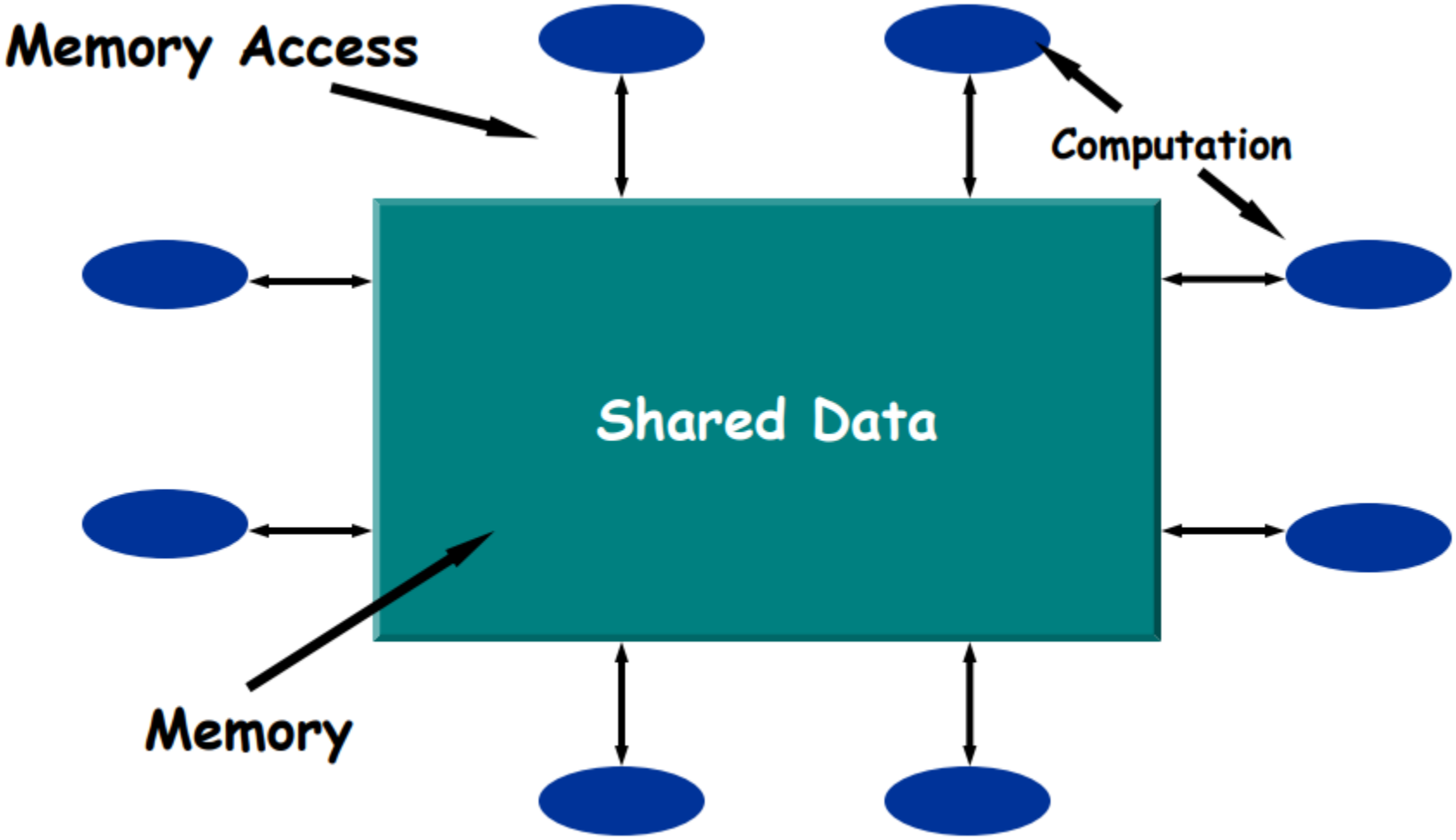
- A complex body of knowledge, that needs to be persisted and manipulated in several ways

Architectural Style:

- Repository Style

Central Repository Architectural Style (aka Shared Data Style)

- Components :
  - Data repository - A central, reliable, permanent data structure that represents the state of the system
  - Data accessors – A collection of independent computational elements that operate on the central data
- Connectors:
  - Read/Write mechanism (e.g., procedure calls or direct memory accesses), requiring sophisticated infrastructure
- Examples:
  - Graphical editors, database applications, AI Knowledge Bases



Central Repository Specializations

- **Blackboard Architecture**:
  - An accessor component changes data on the repository, all other components are notified
  - Require an Active Data Repository that notifies all components about arrival of new data or changes
  - Often, notification implemented as database triggers
- **Passive Data Repository**:
  - Components access repository as and when they want

Benefits:

- Efficient way to share large amounts of data
- Centralised management of the repository
  - Concurrency access and data integrity
  - Security
  - Backup

Weakness:

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

Problem Context:

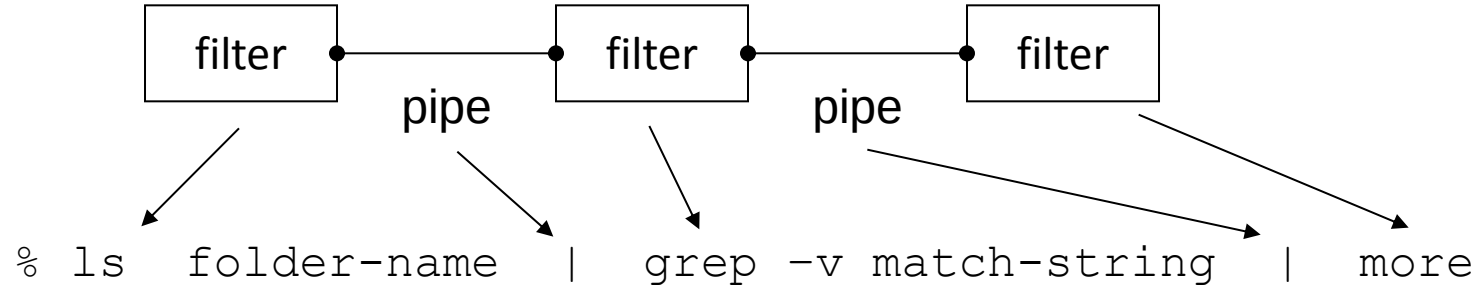
- How to provide a design that is suitable for processing data streams?
- How to transform input data streams stepwise into output data streams?

Architectural Style:

- Pipes and Filters

Pipe-and-Filter Architectural Style

- Components: **Filters** transform input into output
- Connectors: **Pipe** data streams
- Examples: UNIX shell commands, Compilers



Benefits:

- Easy to understand the overall input/output behaviour of a system as a simple composition of the behaviours of filters
- Decouples different data processing steps so that they can evolve independently of one another
- Support Reuse:
  - Any two filters can be recombined, if they agree on data formats
- Flexible and easily maintained as filters can be recombined or easily replaced by new filters
- Support concurrent processing of data streams

Weakness:

- What if an intermediate filter crashes?

Problem Context:

Problem Context:

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

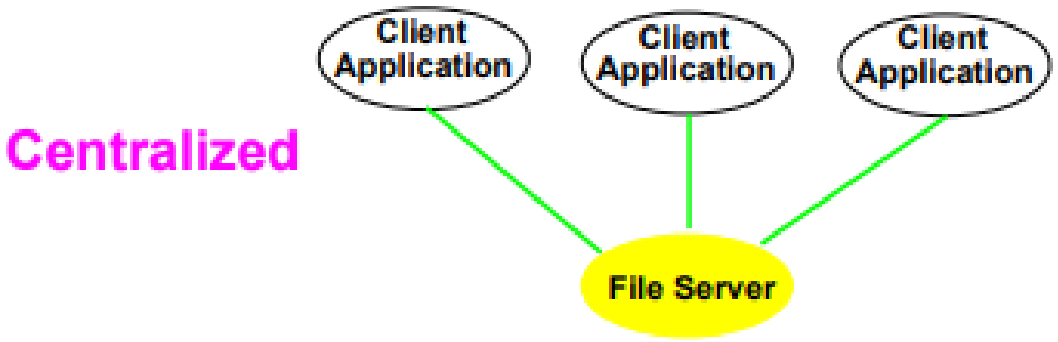
Architectural Style:

- Client-Server Architecture

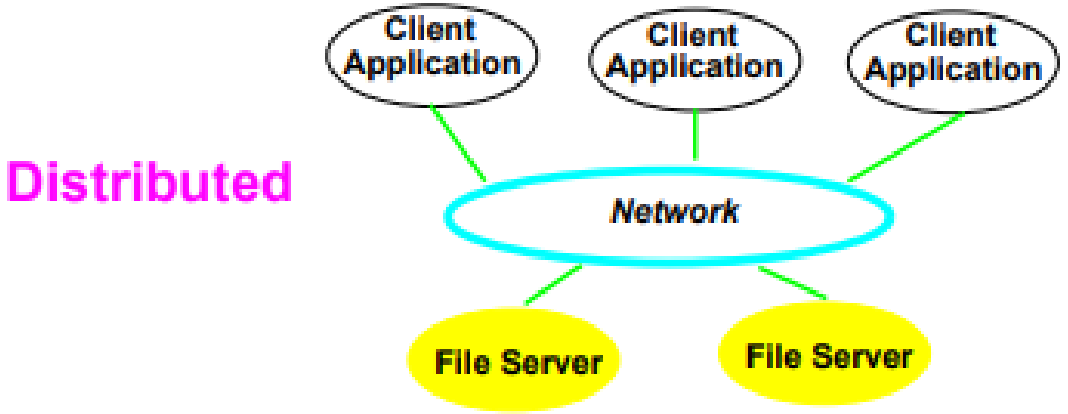
Client-Server Architecture

- Basic architectural style for distributed computing
- Two distinct component types:
  - A server that provides specific services e.g., database or file server
  - A client component that requests these services
  - 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

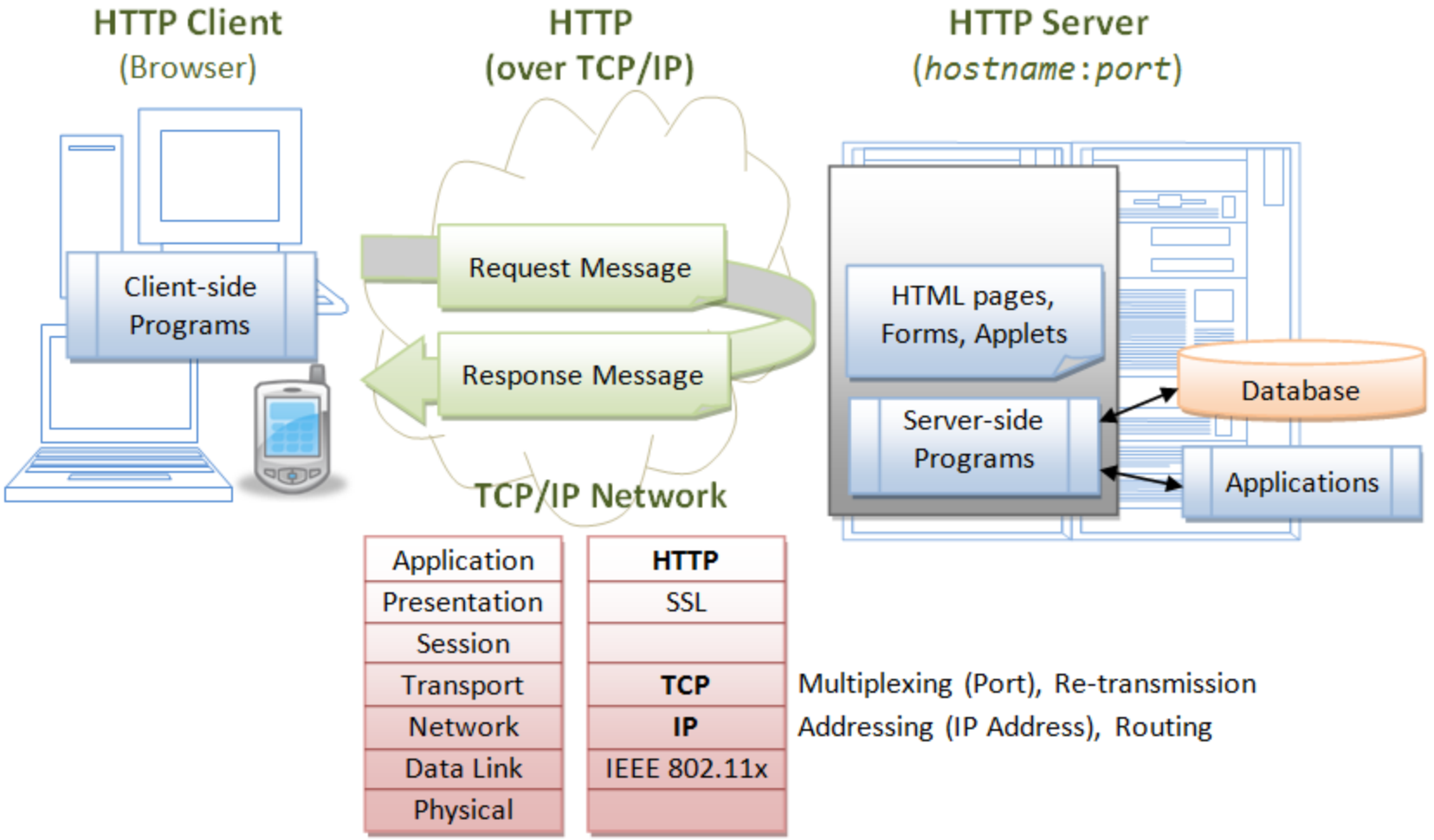
A 2-tier client-server architecture



- *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*



Web Client-Server Architecture

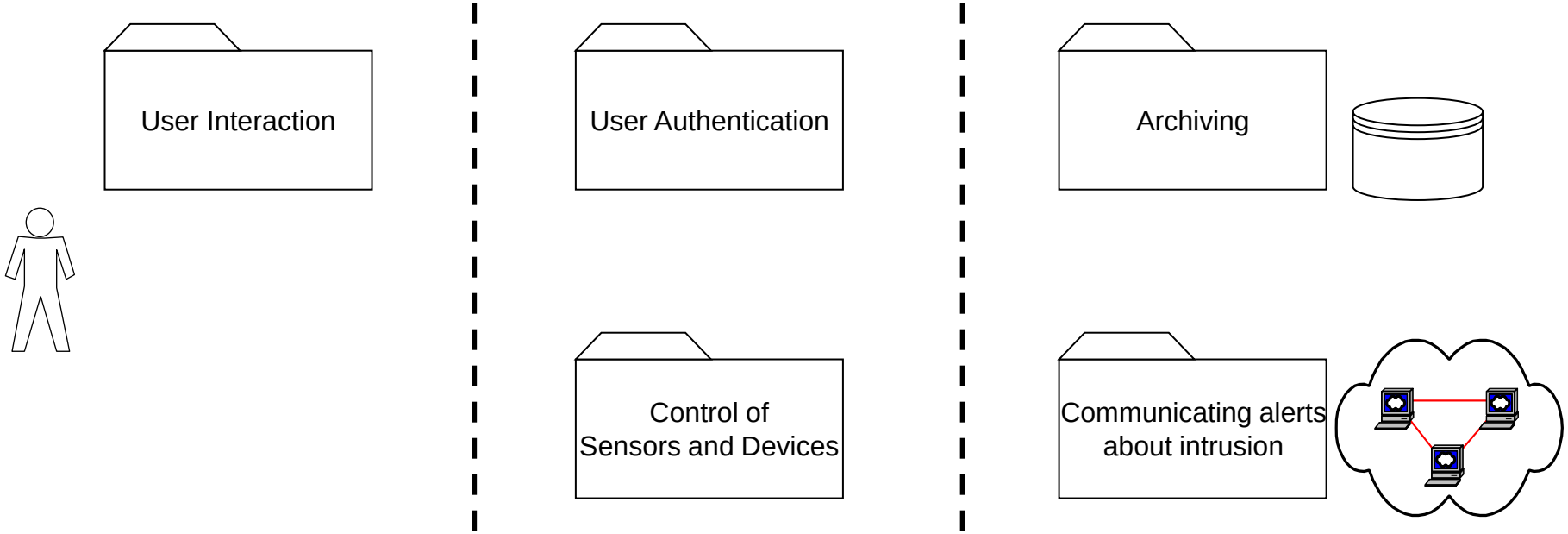


https://www.ntu.edu.sg/home/ehchua/programming/howto/Tomcat_HowTo.html

3-Tier / N-Tiered Architecture

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

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



Client-Server 3-Tier Architecture

Presentation tier

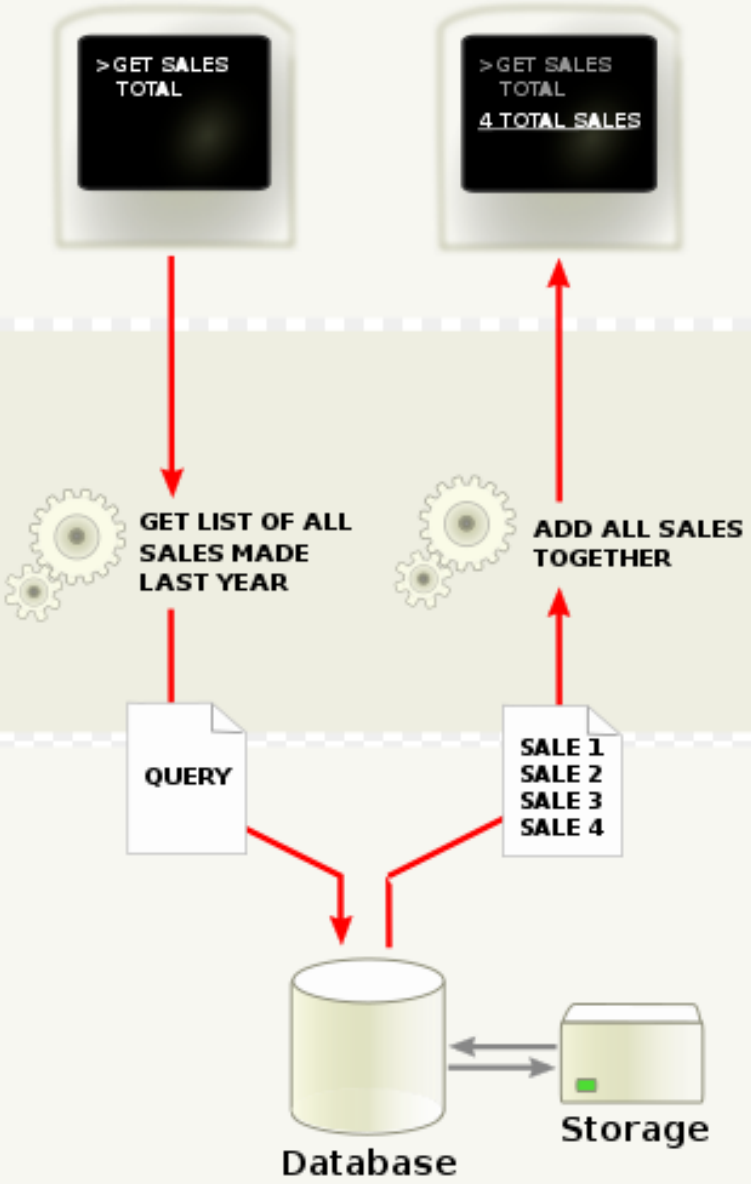
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

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.



Benefits:

Modularisation and separation into multiple tiers provides for:

- Distribution of roles and responsibilities of a system to be distributed among several independent machines
- Easier **maintenance** and **reuse** as each is built for a discrete purpose
- Deployment of modules to different servers enhancing security, scalability and performance

Weakness:

- 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!
KEINSTERS ©2001 THE CHARLOTTE OBSERVER

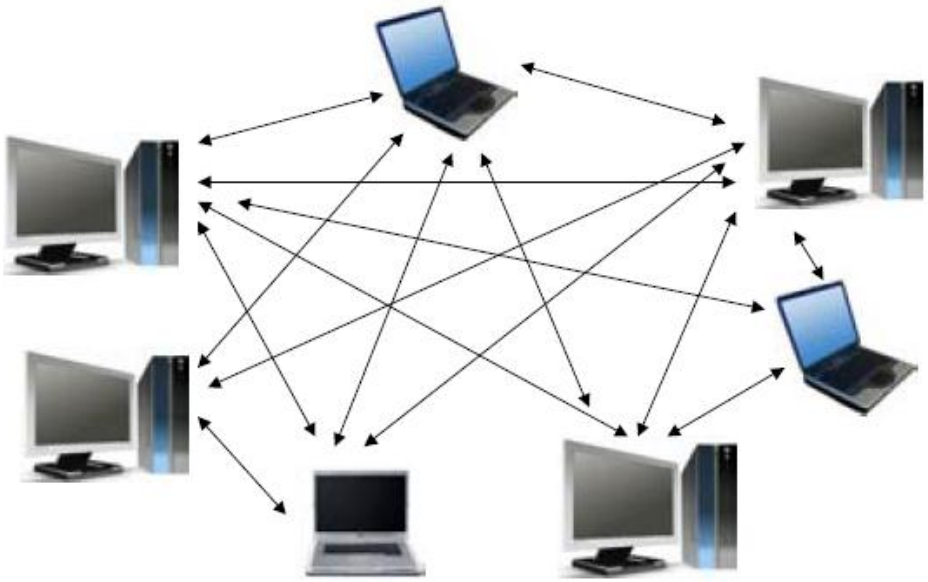
22

Peer-to-Peer (P2P) architecture

- Each peer component can function as both a server and a client
- Information distributed among all peers
- A peer may need to communicate with other peers to locate information

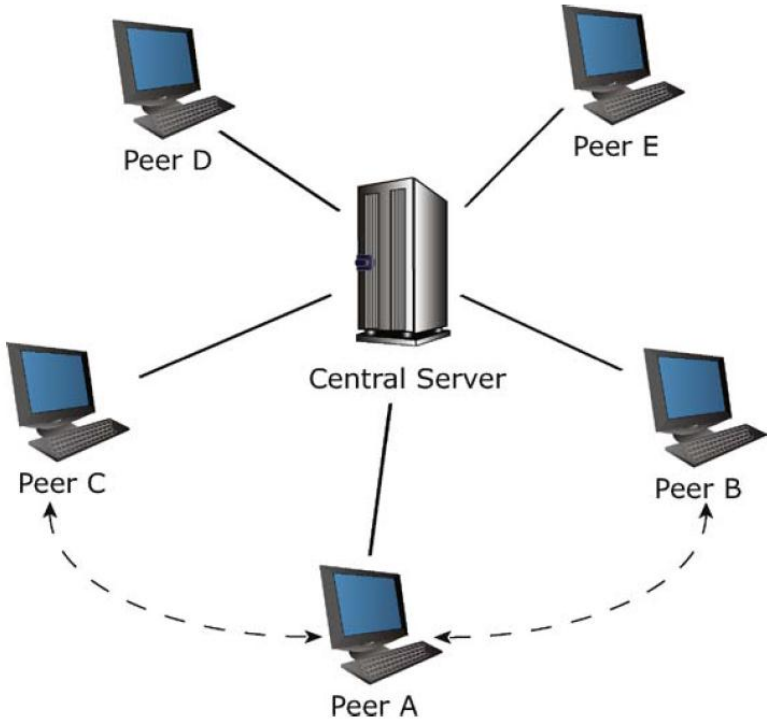
Pure P2P Architecture

(No central Server)



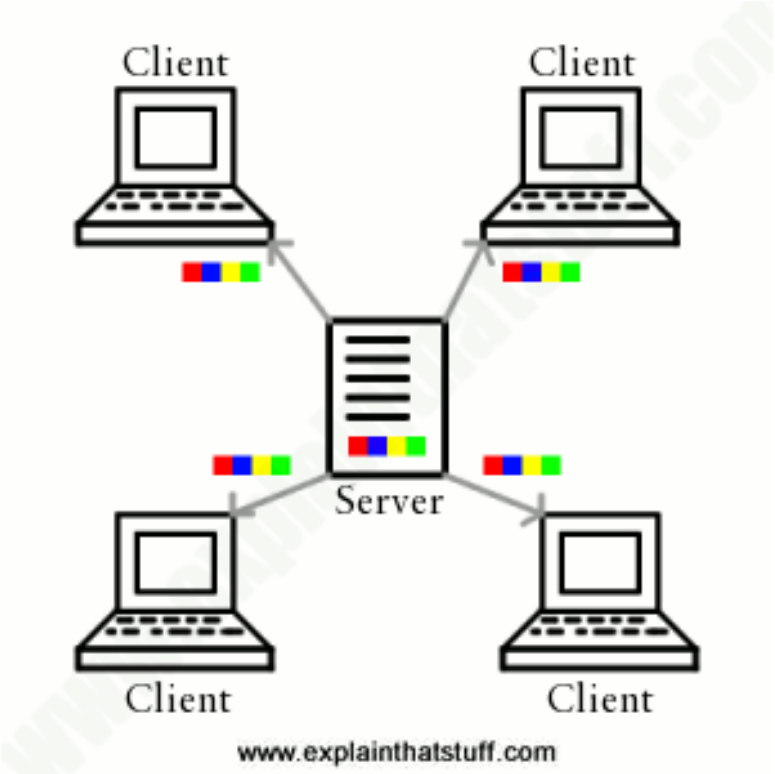
Hybrid P2P Architecture

Central server helps peers to locate other peers

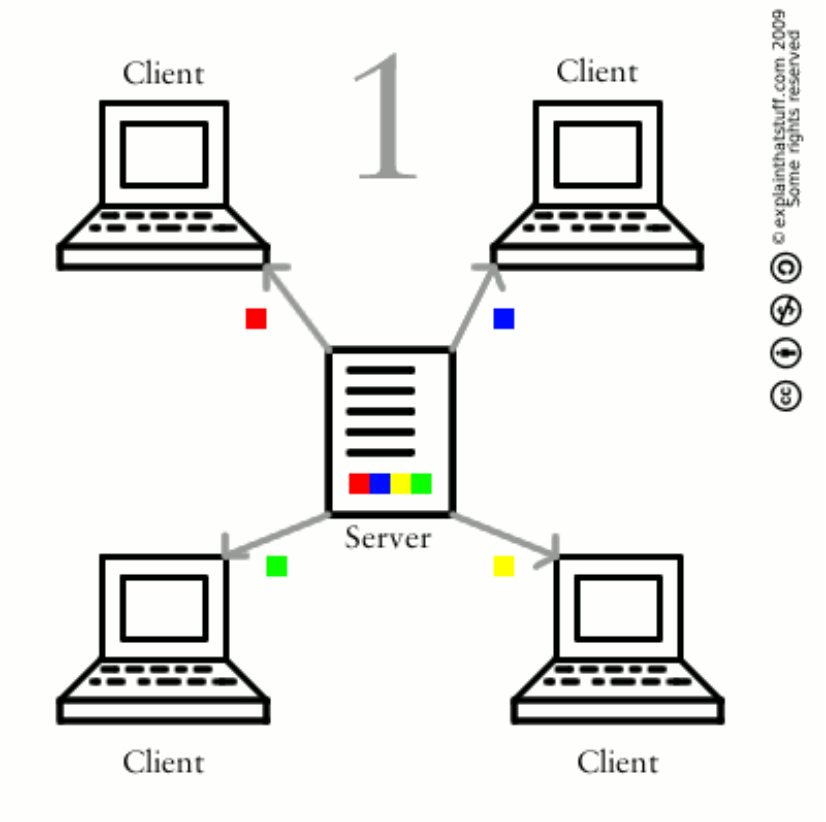


Example (1): Bit Torrent

Client-Server



Peer-to-Peer Bit Torrent



Example (2):

Peer-to-Peer Internet Telephony Network

"Hello Mum!"

www.skype.com/

Benefits:

- More efficient as all clients provide resources
- Unlike client-server, capacity of the network increases with number of clients
- More robust as 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

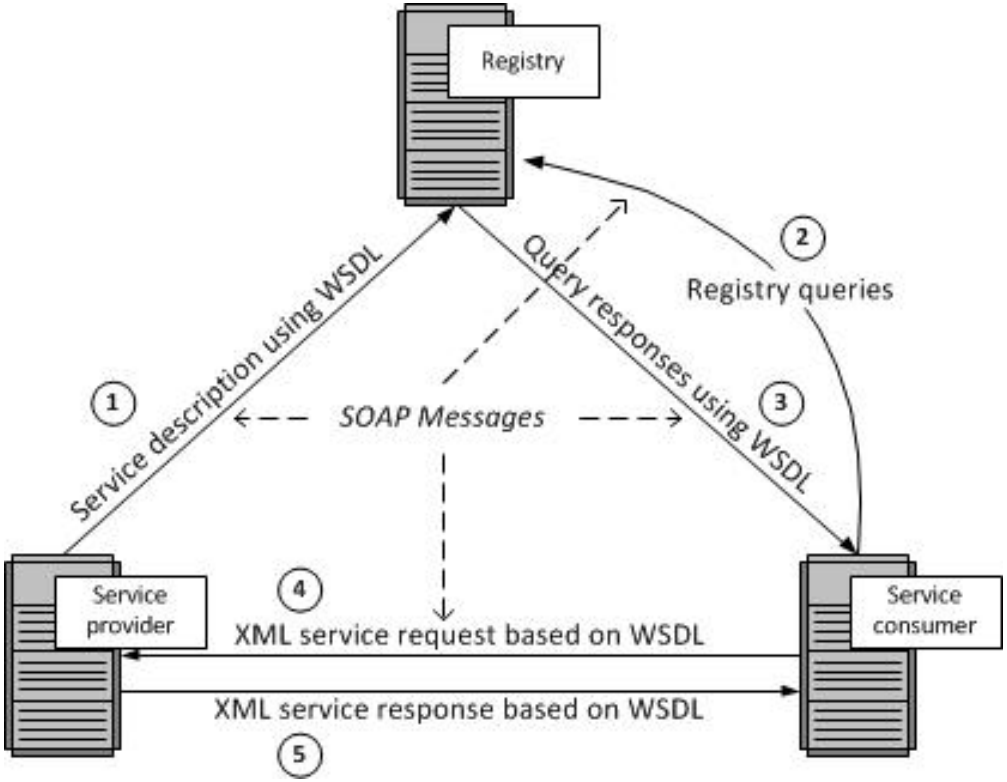
Some other styles:

Publish-subscribe style

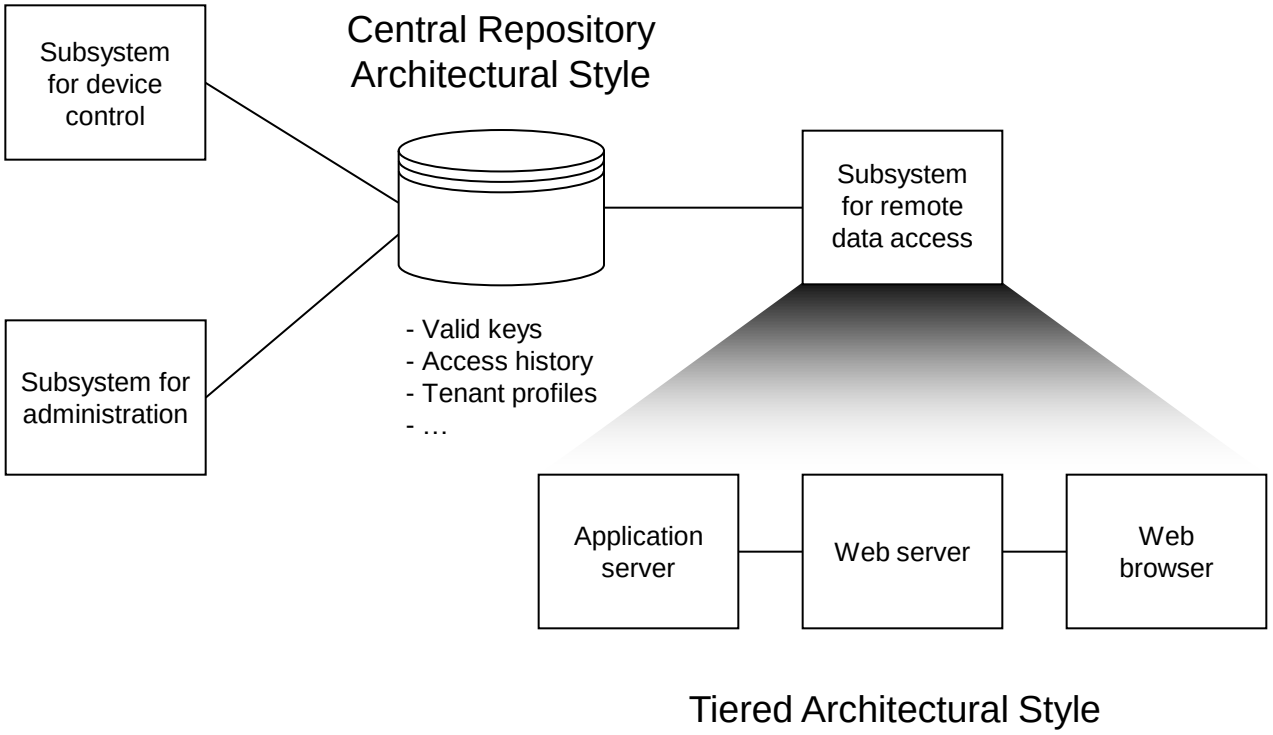
- Two distinct component types
  - Some components generate or publish events
  - Some components register their interest in these events and are invoked when these events occur
- Can be seen as a specialisation of blackboard architecture without the central repository
- Few examples
  - User interface frameworks
  - traders register interest (subscribe to) particular stock prices and receive updates as prices changes
  - wireless sensor networks

Some other styles: SOA (Service Oriented Architecture)

- an architectural approach that advocates the creation of software components as autonomous, platform-independent, loosely coupled *services* and is agnostic of the underlying implementation technology
- *web services* are the preferred realisation of SOA
- applications – B2B services

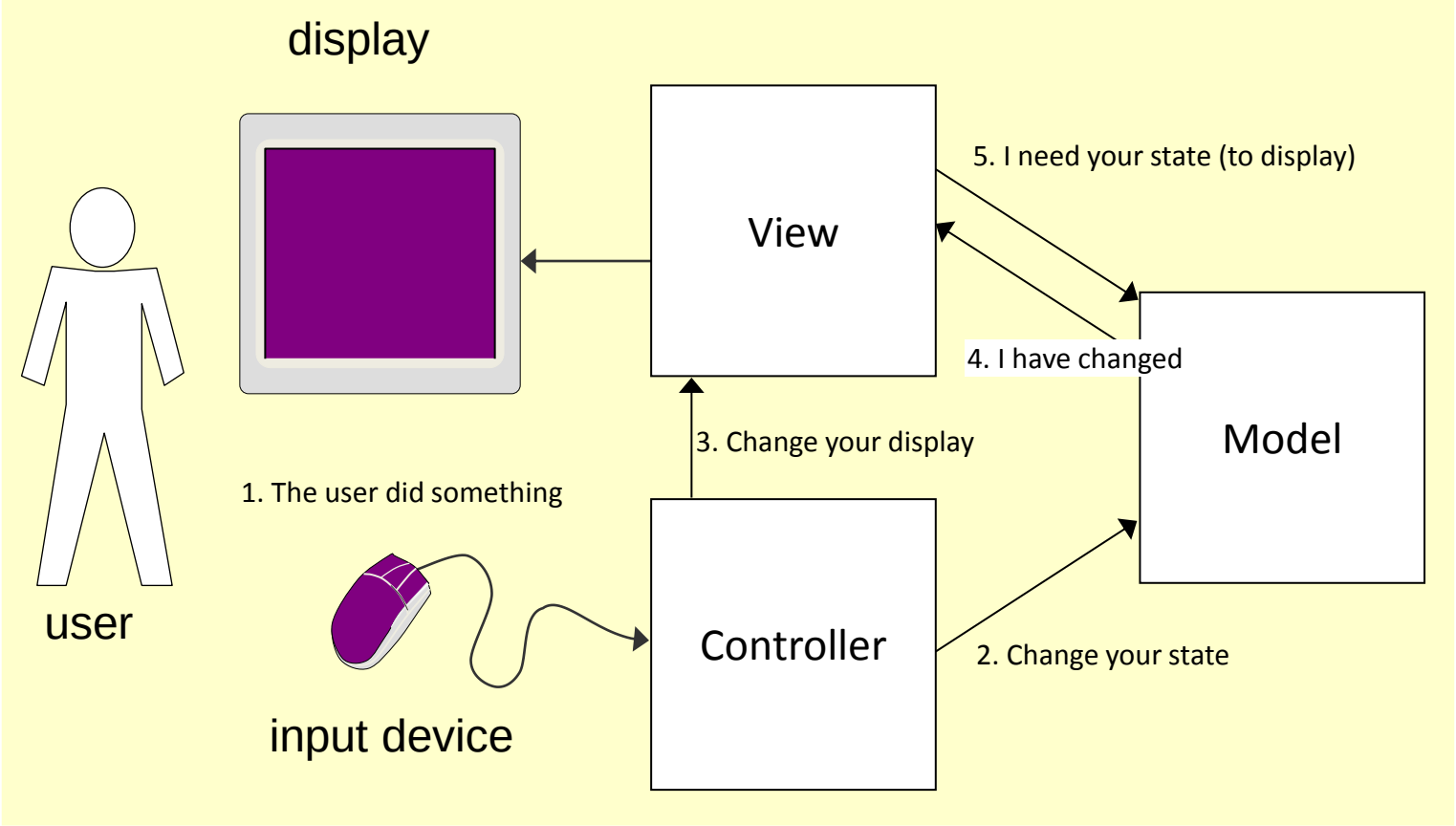


Real System is a Combination of Styles



Application Architecture: Model-View-Controller (MVC)

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



Model

- Holds all the data, state and application logic
- Responds to instructions to change of state (from the controller
- Responds to requests for information about its state (usually from the view),
- Sends notifications of state changes to “observer” (view)

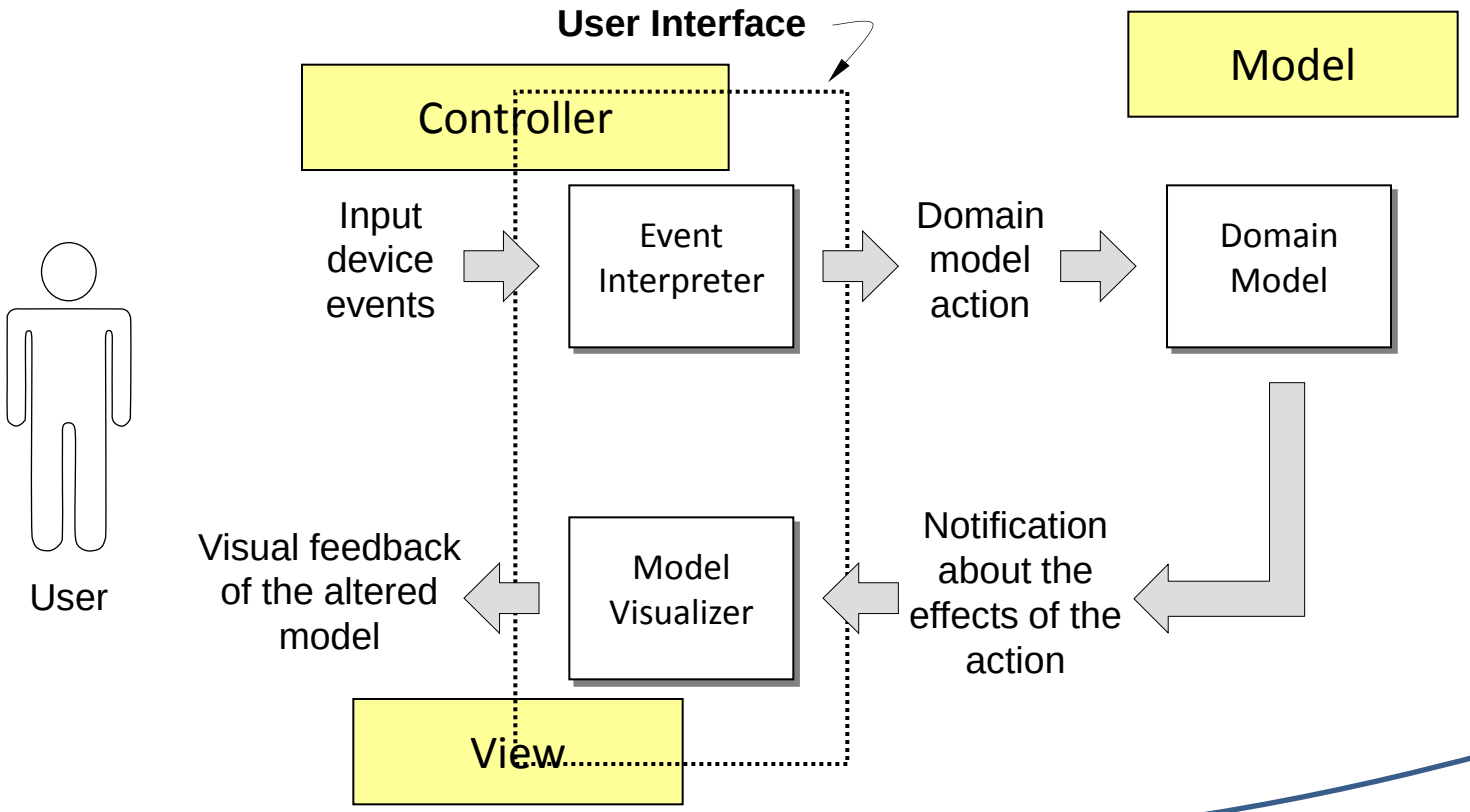
View

- Gets data directly from the Model and manages display of information

Controller

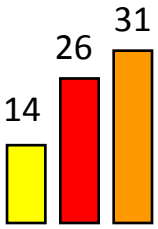
- Takes user input and informs the view or model to change as appropriate

Model-View-Controller

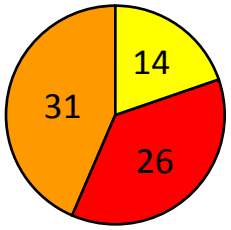


Model: array of numbers [14, 26, 31]

➔ Different Views for the same Model:



versus



Benefits:

- Accommodates change
- Supports multiple views of the same data on different platforms at the same time
- Enhances testability

Weakness:

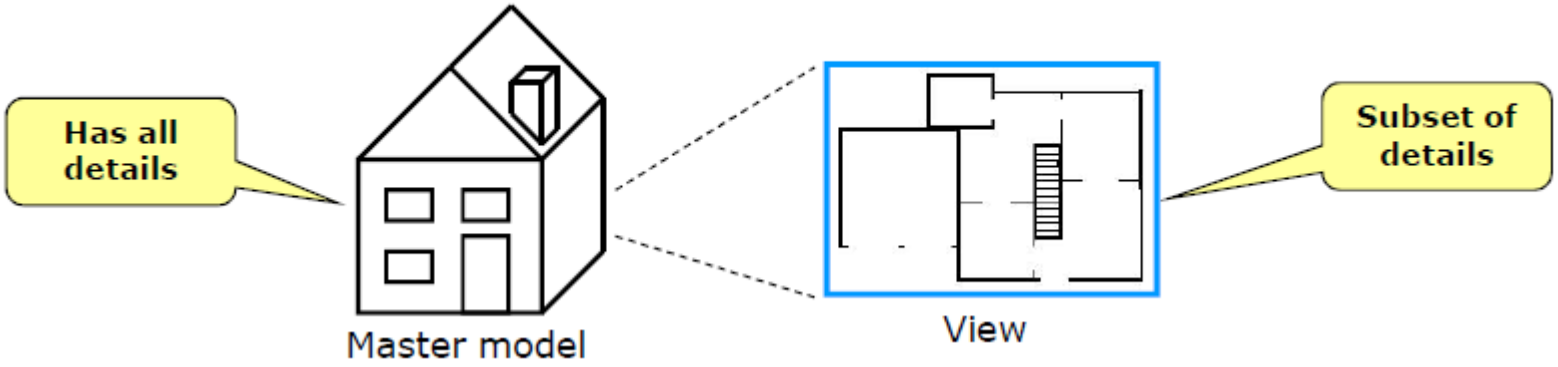
- Complexity
- Cost of frequent update – an active model that undergoes frequent changes could flood the views with update requests

Views:

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

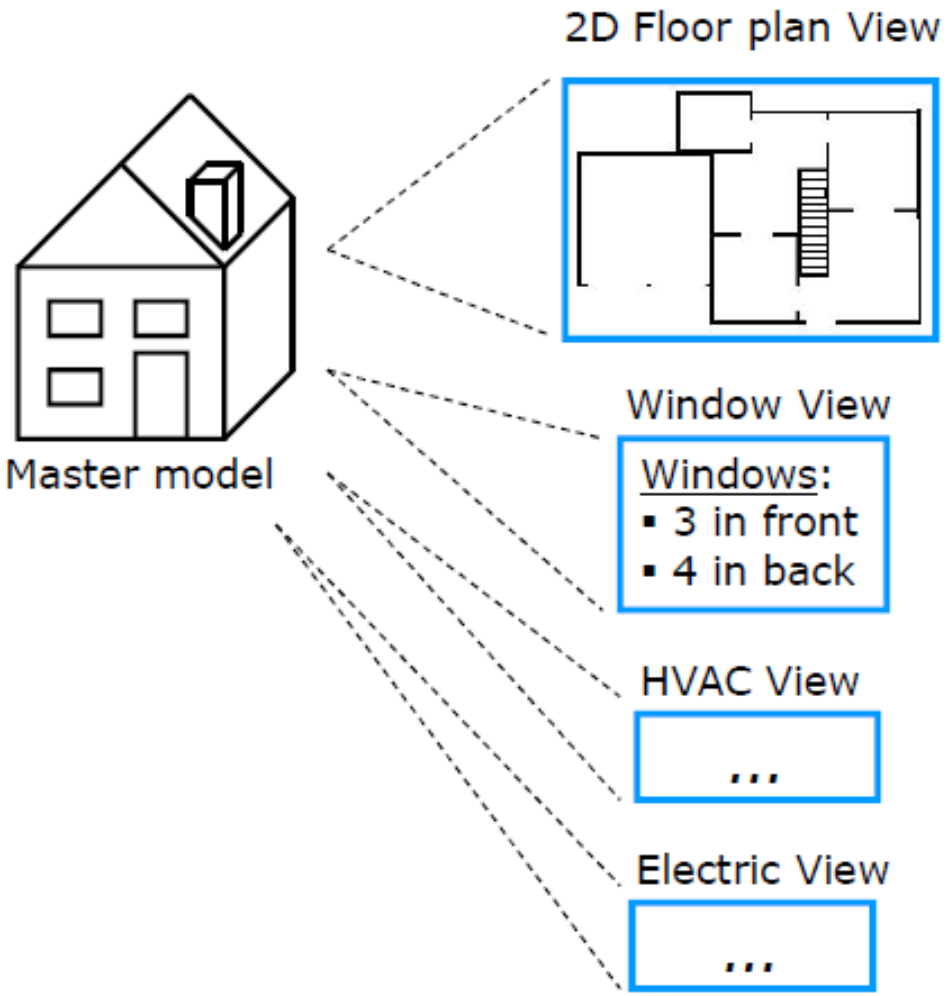
Projects from the **master model**

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



Example House Views:

- 2D view of floor plan
- Window layouts
- Electrical wiring circuits
- Landscaping
- Zoning for ducted a/c
- Plumbing
- Networking



Architecture Views:

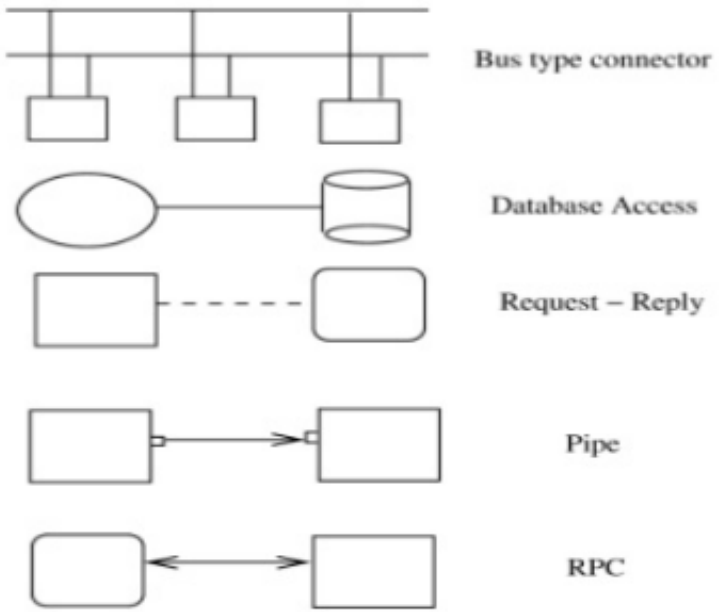
- Model view
  - Decomposes the functionality into a coherent set of software (code) units
- Component and Connector view
  - Describes a runtime structure of the system (components, data stores, connectors (e.g. pipes, sockets that enable interaction between components)
- Allocation view
  - Describes how the software units map to the environment (hardware resources, file-systems and people)
  - Exposes properties like which process runs on which processor

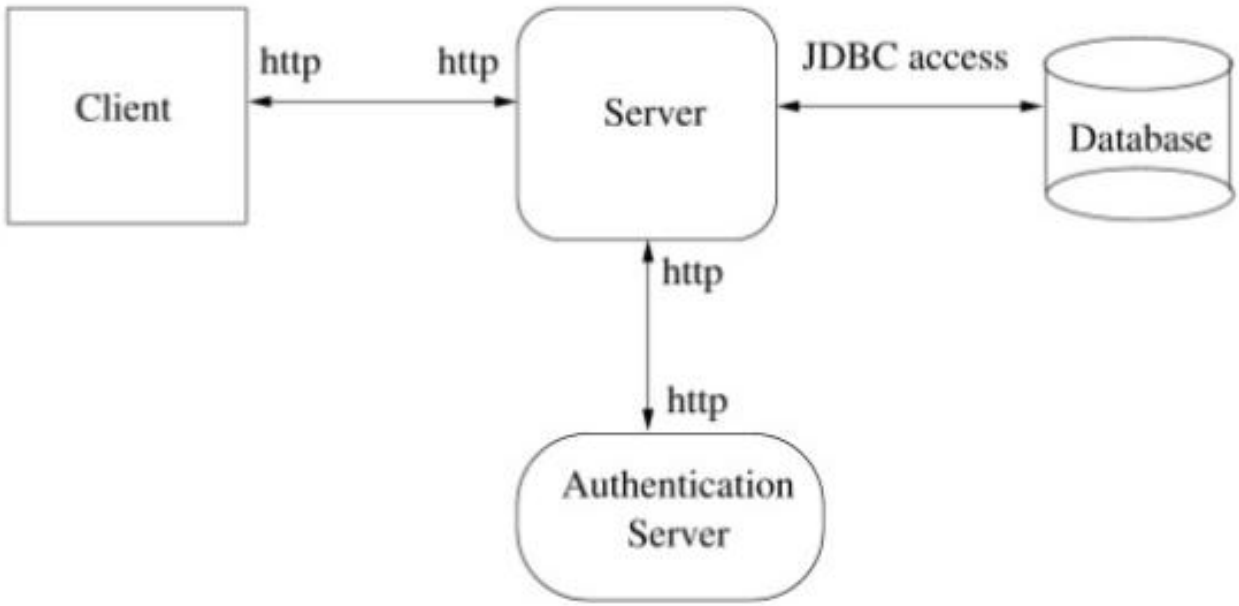
Documenting software architectures

- Model view
  - UML class diagrams and package diagrams
- Component and Connector view
  - box-and-line diagram (informal), UML component diagram (formal)
- Allocation view
  - UML deployment diagram

Box-Line diagram

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





UML 2 component diagram

- A formal notation to visualise the static implementation of a system and understand the wiring of the various components that make up the system
- A useful communication tool for various groups:
  - architects – it is a natural format to model their solution
  - stakeholders - provides an early understanding of the overall system being built
  - developers - provides a macroscopic view of the system being built, helping them to create an implementation roadmap, make decisions about task assignments, necessary skills required.

What is a component?

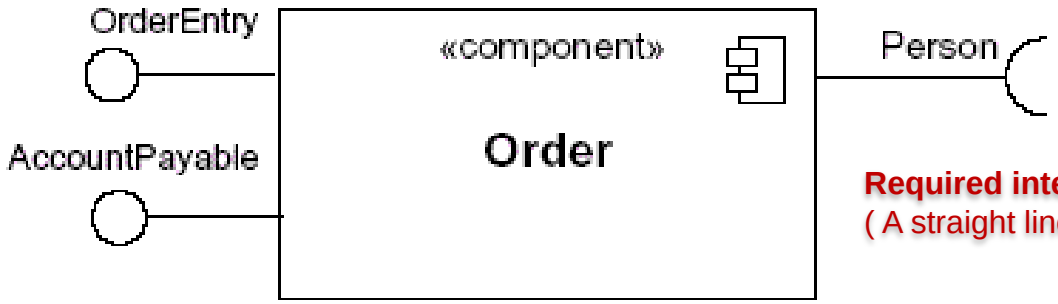
- An independent, autonomous unit within a system or sub-system that represents a software module of classes, web service or some software resource
- Typically implemented as a replaceable module and can be deployed independently
- A UML 2 component is depicted as:

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:

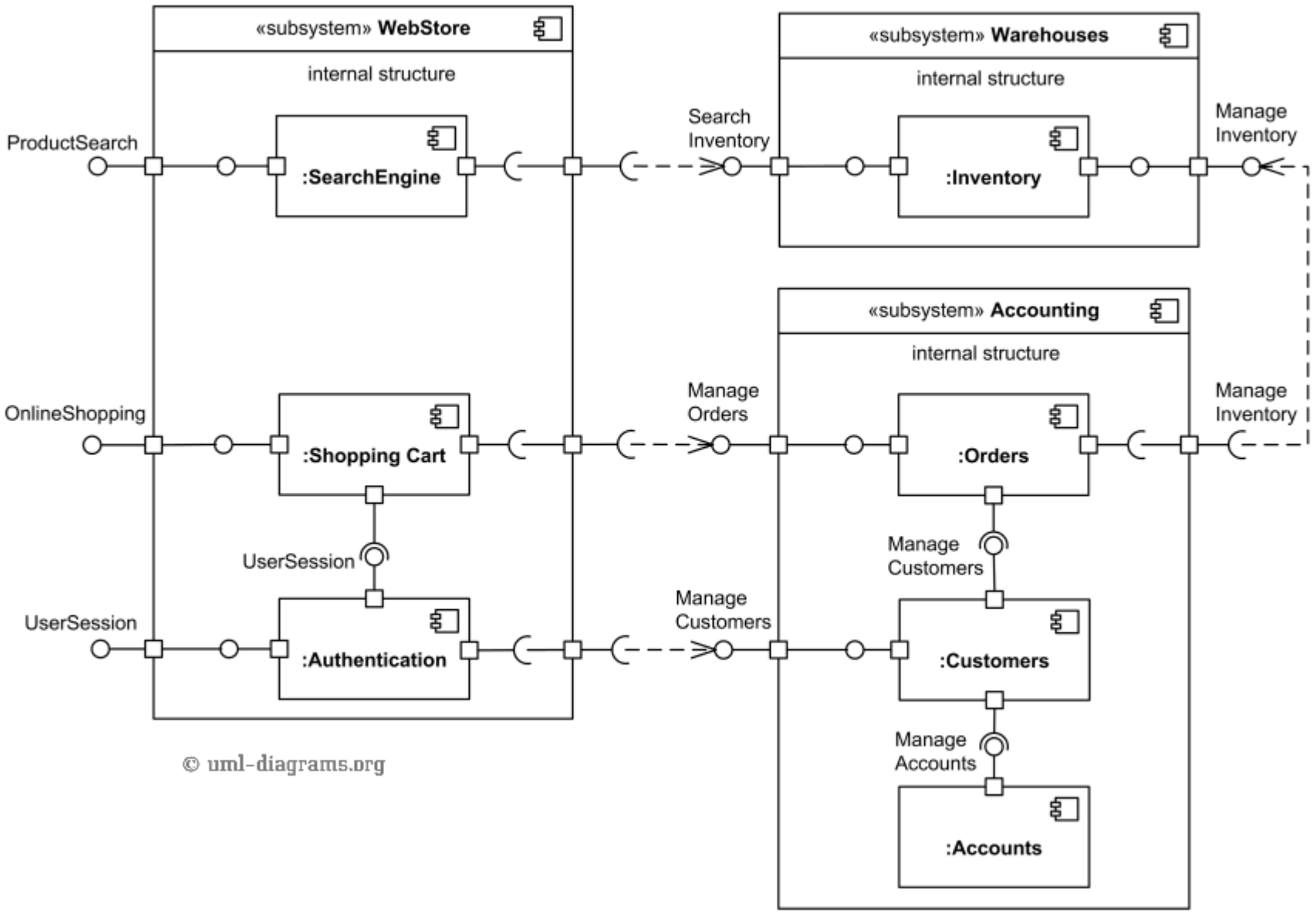
- Provided interfaces : a collection of one or more methods that define the services offered by the component and represent a formal contract with a consumer
- Required interfaces : dependency services i.e. services required by the component in order to perform its function



Required interfaces: Person
(A straight line with a half circle)

Provided interfaces: OrderEntry, AccountPayable
Lollipop notation (straight line with an attached circle)

A UML 2 component diagram for an online shopping application



UML 2 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