7. Software Architecture

COMP1531 • KC Notes

7.1 Software Architecture

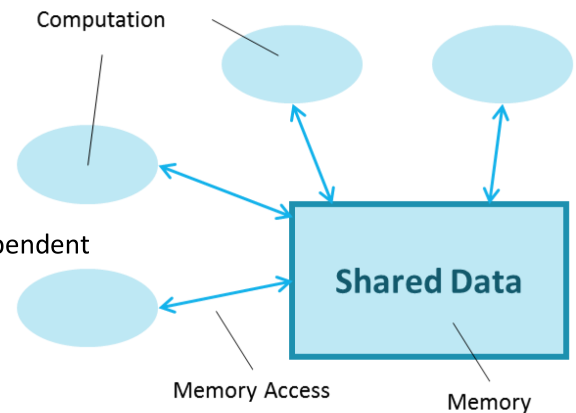
- **<u>Software Architecture</u>**: the **structure of the system**, which comprise of **software elements, the externally visible properties** of those elements and the **relationships** among them
    - "Big picture" and organisation of the system-to-be
    - System is **partitioned into logical sub-systems or parts**
    - Provides a **high-level view** of the system
- Decomposing a system is useful
    - **Tackle complexity** – "**divide and conquer**"
    - Reduce duplicated functionality – reuse parts
    - Focus on creative parts
    - Support future changes by decoupling unrelated parts – "**separation of concerns**"
    - Useful for communication between stakeholders
    - Understanding of **macro properties in the system** and how the system fulfils requirements
- **<u>Architecture</u>**: focus on **non-functional requirements** and **decomposition of functional requirements**
    - Design focuses on **implementing functional requirements**
    - Software architecture involves a **set of high-level decisions** that determine **structure**
    - Should be determined early in the system – hard to modify later

7.2 Software Architectural Styles

- **<u>Software Architectural Style</u>**: a family of systems, a **pattern of structural organisation**
    - Patterns described as abstract representations
    - Real system is a **combination of multiple styles**
- A style is defined by:
    1. **Components**: **elements** that do the work
        - e.g. classes, databases, processes
    2. **Connectors**: **communication protocol** between the different components
        - e.g. function call, remote procedure call, event broadcasts
    3. **Constraints**: how the **components can be combined**
        - Where data flows in and out of components/connectors
        - Topological constraints that define arrangement of components/connectors
- **Compromise**: which design is "best" based on all considerations

7.3 Central Repository Architectural Style (Shared Data Style)

- **Central Repository Architectural Style**: has permanent 'shared data' module
- **Problem context**: a complex body of knowledge that needs to be persisted and manipulated in several ways
    1. Components:
        - **Data repository** – a central reliable permanent data structure that represents the state of the system
        - **Data accessors** – collection of independent elements that operate on that central data
    2. Connectors:
        - **Read/Write mechanism** (procedure calls, direct memory access)
    3. Examples:
        - Graphical editors
        - Database applications
        - AI knowledge bases



- Benefits:
    - Efficient way to share large amount of data
    - Concurrent access and data integrity, benefits security and backup
- Weaknesses:
    - All independent components must agree on a **repository data model** a priori
    - Connectors implement complex infrastructure
    - Distribution of data can be a problem
- Specialisations:
    - **Blackboard architecture**: accessor component changes data on the repository, all components are notified
        - Requires **Active Data Repository**
    - **Passive data repository**: components access repository when they want

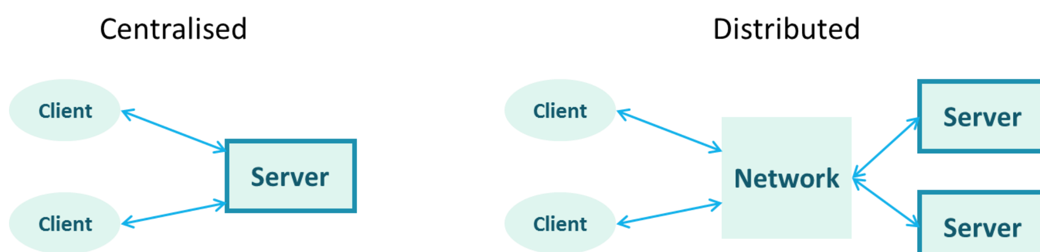
7.4 Pipe-and-Filter Architecture Style

- **Problem context**: design that is suitable for processing and transforming data streams
  1. Components: **Filters** transform input into output
  2. Connectors: **Pipe** data streams
  3. Examples:
     - UNIX shell commands, compilers

- Benefits
  - Easy to understand **input and output**
  - **Decouples data processing steps**, each step can evolve independently
  - **Reuse**: any two filters can be recombined, given the same data formats
  - Flexible and easily maintained
  - Supports **concurrent processing** of data streams
- Weakness
  - **One filter breaking** causes whole system to break

7.5 Client-server architecture

- **Problem context**: share data between client and a service provider across different locations
  1. Components:
     - **Server**: provides specific services, e.g. database, file server
     - **Client**: requests these services
  2. Connector:
     - **Request-response model**
  3. Examples:
     - File server, Database server, Email server



- Web client-server architecture: client requests (GET/POST) and receives a response from HTTP server
- 3-Tier/N-Tiered architecture: **deployment of components separated into multiple layers**
    - E.g. **presentation** (rendering, UI)**, business** (business domain rules)**, data** (storage)
- Benefits:
    - **Modularisation and separation into multiple tiers**
        - Easy to distribute roles and responsibilities of a system
        - Easier maintenance and reuse of modules
- Weakness:
    - Complex and expensive infrastructure

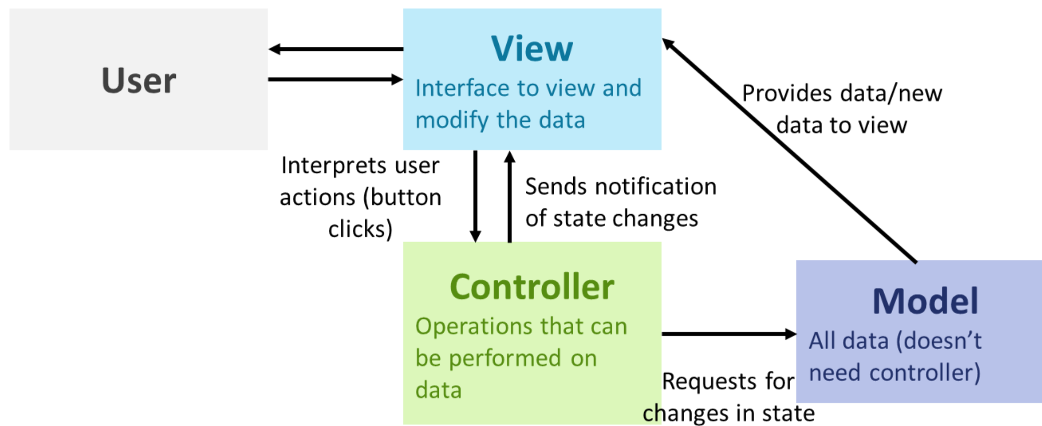
7.6 Peer-to-Peer (P2P) architecture

- **Problem context**: Resolve **network congestion and single point of failure**
    1. Components: **each peer component can be both server and client**
    2. Connector: request-response model
- Examples:
    - Bit Torrent, Napster, Skype
- Benefits:
    - More efficient as all clients provide resources
    - Capacity of network **increases with number of clients**
    - More robust – no single point of failure
- Weakness:
    - Architectural complexity
    - Distributed resources may not be always available

7.7 Other Styles

- **Publish-subscribe style**:
    - Components:
        - Generate or publish events
        - Register their interest in events
    - Specialisation of blackboard architecture, without central repository
    - Examples: User interface frameworks, trader subscribing to stock price updates, wireless networks
- **SOA (Service Oriented Architecture)**:
    - Components:
        - Autonomous, platform-independent loosely coupled services
        - Agnostic (does not know of) underlying implementation technology
    - Examples: web services, B2B services

7.8 Application Architecture: **Model-View-Controller**



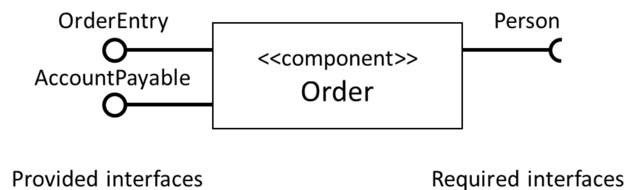
- **Model**:
  - Holds all data, state and application logic
  - **Responds to instructions** from **controller** to change of state
  - Responds to **requests for information** from **view** about its state
  - **Sends notifications** of state changes to the view
- **View**
  - Gets data from Model and **displays information**
- **Controller**
  - Takes user input and **informs view or model** to change

- Benefits:
  - Accommodates change
  - Supports multiple views of the same data on different platforms
  - Enhances testability
- Weaknesses:
  - Complexity
  - Cost of frequent update – an active model that undergoes frequent changes can flood the view with update requests

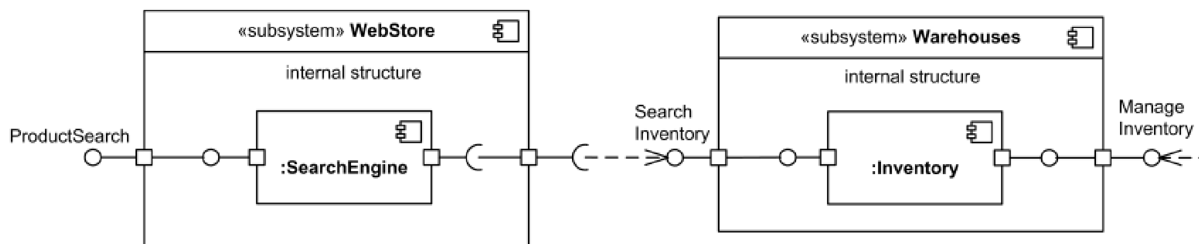
7.9 UML 2 **component diagram**

- Diagram is a **formal notation to visualise static implementation of a system**
- Components: rectangle with visual stereotype on top right corner

- Component **interfaces**:
  - **Provided interfaces**: one or more **methods that define the services** offered by a component
  - **Required interfaces**: dependency services, e.g. **services required for the component to function**



- **Example: Online shopping**
  - **WebStore** contains three components, **SearchEngine** has a **provided interface** ProductSearch and requires **SearchInventory** from **Inventory** to operate. Dotted line symbolises a **dependency**



Source uml-diagrams.org

7.10 UML 2 **deployment diagram**

- A **static view of run-time configuration** of the processing nodes and the components that run on these nodes
  - Shows hardware, software installed on hardware, and middleware