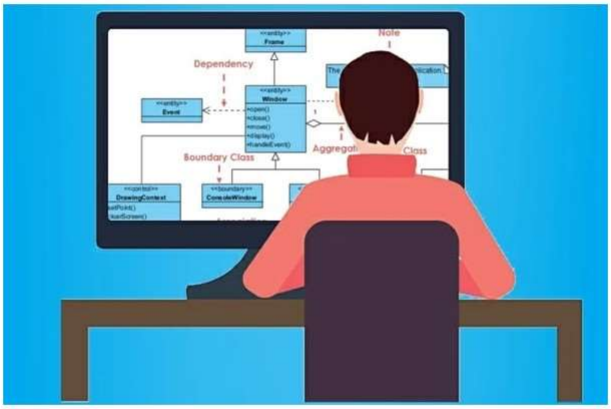
Software Design & Architecture
Spring 2022 - Week-11



مدرس: مہندس ماجد کلیم

جامعہ بحریہ، واقعگاہ کراچی

Engr. Majid Kaleem

WEEKLY AGENDA

TENTATIVE WEEKLY DATES		TENTATIVE TOPICS
1	Mar 7th – Mar 11th	INTRODUCTION TO THE COURSE; DEFINING SOFTWARE ARCHITECTURE & DESIGN CONCEPTS
2	Mar 14th – Mar 18th	DESIGN PRINCIPLES; OBJECT-ORIENTED DESIGN WITH UML
3	Mar 21st – Mar 25th	SYSTEM DESIGN & SOFTWARE ARCHITECTURE; OBJECT DESIGN, MAPPING DESIGN TO CODE
4	Mar 28th – Apr 1st	FUNCTIONAL DESIGN; UI DESIGN; WEB APPLICATIONS DESIGN **ASSIGNMENT & QUIZ #1**
5	Apr 4th – Apr 8th	MOBILE APPLICATION DESIGN; PERSISTENCE LAYER DESIGN
6	Apr 11th – Apr 15th	CREATIONAL DESIGN PATTERNS
7	Apr 18th – Apr 22nd	STRUCTURAL DESIGN PATTERNS **ASSIGNMENT & QUIZ #2**
8	Apr 25th – Apr 29th	BEHAVIORAL DESIGN PATTERNS
← MID TERM EXAMINATIONS →		
9	May 9th – May 13th	INTERACTIVE SYSTEMS WITH MVC ARCHITECTURE; SOFTWARE REUSE
10	May 16th – May 20th	ARCHITECTURAL DESIGN ISSUES; ARCHITECTURE DESCRIPTION LANGUAGES (ADLS)
11	May 23rd – May 27th	ARCHITECTURAL STYLES/PATTERNS & DESIGN QUALITIES
12	May 30th – Jun 3rd	ARCHITECTURAL STYLES/PATTERNS & DESIGN QUALITIES **ASSIGNMENT & QUIZ #3**
13	Jun 6th – Jun 10th	QUALITY TACTICS; ARCHITECTURE DOCUMENTATION
14	Jun 13th – Jun 17th	ARCHITECTURAL EVALUATION TECHNIQUES
15	Jun 20th – Jun 24th	MODEL DRIVEN DEVELOPMENT **ASSIGNMENT (PRESENTATIONS) & QUIZ #4**
16	Jun 27th – Jul 1st	REVISION WEEK
← FINAL TERM EXAMINATIONS →		

WHAT IS AN ARCHITECTURAL STYLE?

- An architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.
- 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

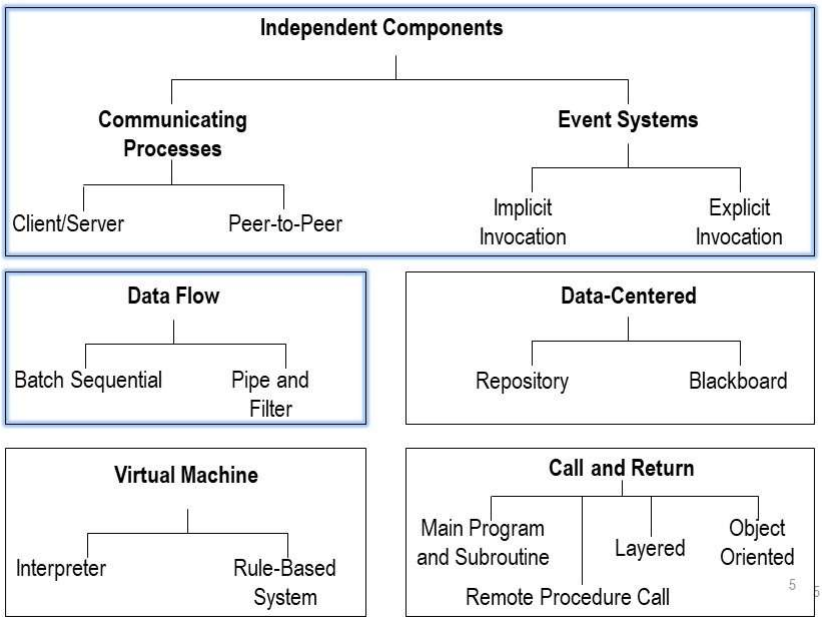
3

WHAT IS AN ARCHITECTURAL STYLE?

- *"Architectural styles define the components and connectors"*
- A software connector is an architectural building block tasked with effecting and regulating interactions among components (Taylor, Medvidovic, Dashofy)

4

VARIOUS ARCHITECTURAL STYLES



5

INDEPENDENT COMPONENTS

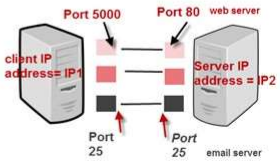
- Independent components architectural style has gained its importance as *it supports addition of components into the system and software reuse*.
- Communicating processes and event based implicit invocation are the sub styles of Independent components architectural style.
- As the name indicates, the components here do not control other components, they just send data to among one another. *In this style all the components should be executable.*
- Active components are known as *processes* and inactive components are known as *modules*.
- The main idea behind this style is that the data can be exchanged through messages, sharing of data is not allowed.
- Each component executes independently and does not control the working of other components.

6

COMMUNICATING PROCESSES

- In communicating processes *all the components are active* and communicate through fixed channels known as communication channels.
- Messages can be sent and received by *ports*.
- Here communication can be synchronous (synchronous means when one component is sending message other component is receiving it at the same time) and asynchronous (asynchronous means component is sending data without knowing whether the other component has received it or not).
- Communication can also be one-to-one(one component to other) or broadcast (one component to all other components).

.



IP Address + Port number = Socket

TCP/IP Ports And Sockets

CLIENT-SERVER ARCHITECTURAL STYLE

- *Client and server architecture is sub type of communicating process*.
- Here a server facilitates data to one or more clients.
- This can work in both synchronous and asynchronous method according to client's request
- **Summary:** Clients send service requests to the server, which performs the required functions and replies as needed with the requested information. Communication is initiated by the clients.
- **Components:** Clients and server.
- **Connector:** Remote procedure calls, network protocols.
- **Data elements:** Parameters and return values as sent by the connectors.
- **Topology:** Two-level, with multiple clients making requests to the server.

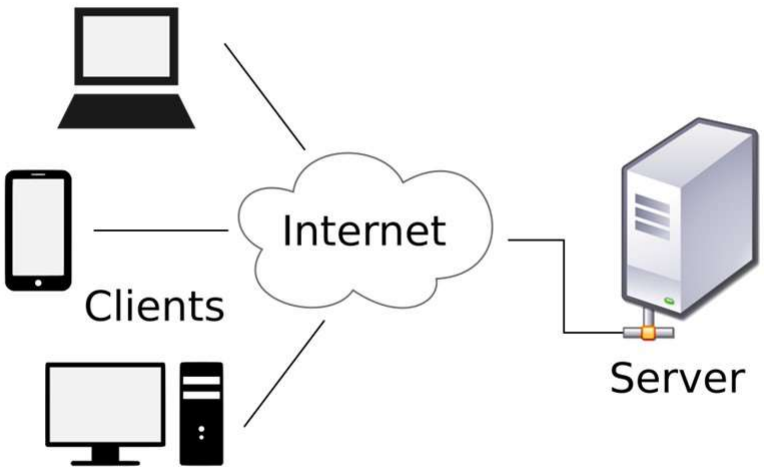
8

CLIENT-SERVER ARCHITECTURAL STYLE

- **Additional constraints imposed:** Client-to-client communication prohibited.
- **Quality yielded:** Centralization of computation and data at the server, with the information made available to remote clients. A single powerful server can service many clients.
- **Typical uses:** Applications where centralization of data is required, or where processing and data storage benefit from a high-capacity machine, and where clients primarily perform simple user interface tasks, such as many business applications.
- **Cautions:** When the network bandwidth is limited and there are large number of client requests.
- **Relation to programming languages or environments:** None

9

CLIENT-SERVER ARCHITECTURAL STYLE



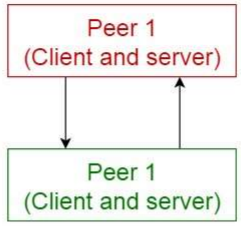
10

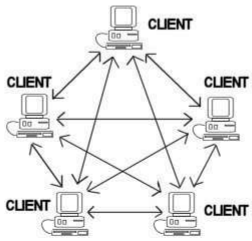
PEER-TO-PEER ARCHITECTURAL STYLE

- State and behavior are distributed among peers which can act as either clients or servers.
- **Peers**: independent **components**, having their own state and control thread.
- **Connectors**: Network protocols, often custom.
- **Data Elements**: Network messages
- **Topology**: Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Supports decentralized computing with flow of control and resources distributed among peers. Highly robust in the face of failure of any given node. Scalable in terms of access to resources and computing power.

11

PEER-TO-PEER ARCHITECTURAL STYLE





12

EVENT SYSTEMS

- Independent components asynchronously emit and receive events communicated over event buses
- Components: Independent, concurrent event generators and/or consumers
- Connectors: Event buses (at least one)
- Data Elements: Events – data sent as a first-class entity over the event bus
- Topology: Components communicate with the event buses, not directly to each other.
- Variants: Component communication with the event bus may either be push or pull based.
- Highly scalable, easy to evolve, effective for highly distributed applications.

13

IMPLICIT INVOCATION ARCHITECTURAL STYLE

- **Implicit invocation** is a term used by some authors for a style of software architecture in which a system is structured around *event handling*, using a form of callback.
- The Main Idea behind implicit invocation is that instead of components invoking procedures of other components, a component announces one or more events.
- The other components if they want to act on a particular event they can register the appropriate methods with that event.
- When that particular event is announced the system invokes all the procedures registered with that event.
- Now here many different procedures can be invoked thus giving a parallel look to it.
- **Examples** of **implicit invocation** systems abound, including virtually all modern operating systems, integrated development environments, and database management systems.

14

IMPLICIT INVOCATION ARCHITECTURAL STYLE

Instead of invoking a procedure directly …

- A **component** can announce (or broadcast) one or more events.
- Other **components** in the system can register an interest in an event by associating a procedure with the event.
- When an event is announced, the broadcasting system (**connector)** itself invokes all of the procedures that have been registered for the event.

15

EXPLICIT INVOCATION ARCHITECTURAL STYLE

- In implicit invocation component does not have control over computations performed in the system.
- Component has no idea what other components will respond to against an event.
- In what order operations take place is also not known to the component and when is the operation done with.
- To overcome this explicit invocation i.e., *Normal procedure call*, is also included in systems.

16

DATA FLOW ARCHITECTURE

- In data flow architecture, the whole software system is seen as a ***series*** of transformations on consecutive pieces or set of input data, where data and operations are independent of each other.
- In this approach, the data enters into the system and then flows through the modules one at a time until they are assigned to some final destination (output or a data store).
- The connections between the components or modules may be implemented as I/O stream, I/O buffers, piped, or other types of connections.

17

DATA FLOW ARCHITECTURE

- The data can be flown in the graph topology with cycles, in a linear structure without cycles, or in a tree type structure.
- The main objective of this approach is to achieve the qualities of reuse and modifiability.
- It is suitable for applications that involve a well-defined series of independent data transformations or computations on orderly defined input and output such as compilers and business data processing applications.

18

BATCH-SEQUENTIAL ARCHITECTURAL STYLE

- Batch sequential is a classical data processing model, in which a data transformation subsystem can initiate its process only after its previous subsystem is completely through −
  - The flow of data carries a batch of data as a whole from one subsystem to another.
  - The communications between the modules are conducted through temporary intermediate files which can be removed by successive subsystems.
  - It is applicable for those applications where data is batched, and each subsystem reads related input files and writes output files.
  - Typical application of this architecture includes business data processing such as banking and utility billing.

19

BATCH-SEQUENTIAL ARCHITECTURAL STYLE

- **Summary:** Separate programs are executed in order, data is passed as an aggregate from one program to the next.
- **Components:** Independent programs.
- **Connector:** The human hand carrying tapes between the programs, aka "sneaker-net".
- **Data elements:** Explicit, aggregate elements passed from one component to the next upon completion of the producing program's execution.
- **Topology:** Linear.
- **Additional constraints imposed:** One program runs at a time, to completion.

20

BATCH-SEQUENTIAL ARCHITECTURAL STYLE

- **Additional constraints imposed:** One program runs at a time, to completion.
- **Quality yielded:** Severable execution; simplicity.
- **Typical uses:** Transaction processing in financial systems.
- **Cautions:** When interaction between the components is required; when concurrency between components is possible or required.
- **Relation to programming languages or environments:** None

21

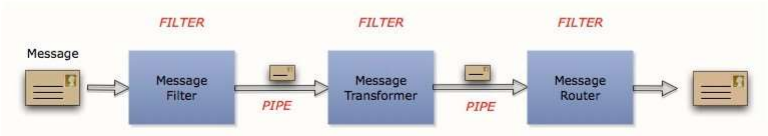
PIPE-AND-FILTER ARCHITECTURAL STYLE

- Pipe and Filter is a simple architectural style that connects a number of components that process a stream of data, each connected to the next component in the processing pipeline via a **Pipe**.
- *The Pipe and Filter architecture is inspired by the Unix technique of connecting the output of an application to the input of another via pipes on the shell.*
- The pipe and filter architecture consists of one or more data sources.
- The data source is connected to data filters via pipes.
- Filters process the data they receive, passing them to other filters in the pipeline.
- The final data is received at a **Data Sink**:

22

PIPE-AND-FILTER ARCHITECTURAL STYLE

- **Pipes** are stateless and they carry binary or character stream which exist between two filters. It can move a data stream from one filter to another. Pipes use a little contextual information and retain no state information between instantiations.
- There are two types of filters − **active filter** and **passive filter**.
- **Active filter**
- Active filter lets connected pipes to **pull data** in and push out the transformed data. It operates with passive pipe, which provides read/write mechanisms for pulling and pushing. This mode is used in UNIX pipe and filter mechanism.
- **Passive filter**
- Passive filter lets connected pipes to **push data** in and pull data out. It operates with active pipe, which pulls data from a filter and pushes data into the next filter. It must provide read/write mechanism.



23

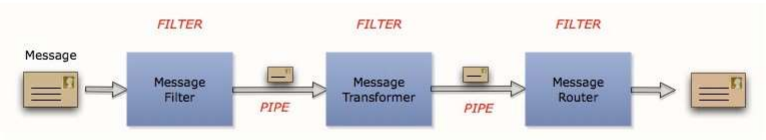
PIPE-AND-FILTER ARCHITECTURAL STYLE

- **Summary:** Separate programs are executed concurrently, data is passed as a stream from one program to the next.
- **Components:** Independent programs, known as filters.
- **Connector:** Explicit routers of data streams; service provided by operating system.
- **Data elements:** Not explicit; must be (linear) data streams. In the typical Unix/Linux/DOS implementation the streams must be text.
- **Topology:** Pipeline, though T fittings are possible.
- **Additional constraints imposed:** -------
- **Quality yielded:** Filters are mutually independent. Simple structure of incoming and outgoing data streams facilitates novel combinations of filters for new, composed applications.

24

PIPE-AND-FILTER ARCHITECTURAL STYLE

- **Typical uses:** Ubiquitous in operating system application programming.
- **Cautions:** When complex data structures must be exchanged between filters; when interactivity between the programs is required.
- **Relation to programming languages or environments:** Prevalent in Unix shells.



25

```
If(anyQuestions)
{
 askNow();
}
else
{
 thankYou();
 submitAttendance();
 endClass();
}
```

REFERENCES

1. ***Software Architecture***, *Perspectives on an Emerging Discipline By Mary Shaw & David Garlan*
2. ***The Art of Software Architecture***, *Design Methods & Techniques By Stephen T. Albin*
3. ***Essential Software Architecture***, *By Ian Gorton*
4. ***Microsoft Application Architecture Guide***, *By Microsoft*
5. ***Design Patterns***, *Elements of Reusable Object-Oriented Software By by Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides*
6. ***Refactoring, Improving the Design of Existing Code***, *By Martin Fowler & Kent Beck*