UNIT-II

Course Contents:

Unit 2. Software architecture models: structural models, framework models, dynamic models, process models. Architectures styles: dataflow architecture, pipes and filters architecture, call-and return architecture, data-centered architecture, layered architecture, agent based architecture, Micro-services architecture, Reactive Architecture, Representational state transfer architecture etc.

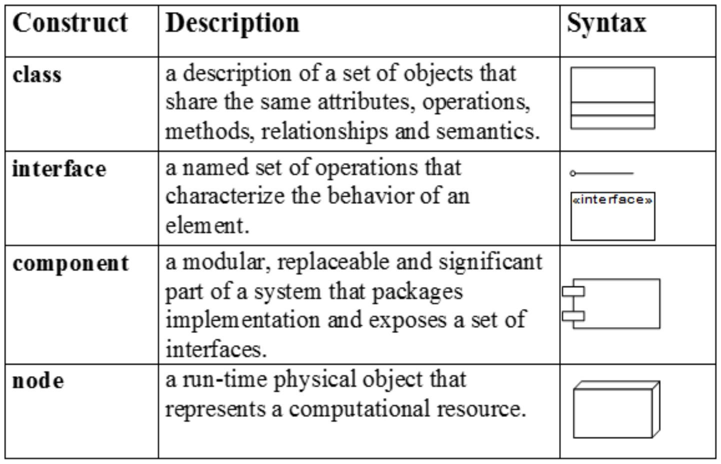
Software Architecture Models:

Structural Models: Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design. Structural models of a system required during the discussing and designing the system architecture. Architectural design is a particularly important topic in software engineering and UML component, package, and deployment diagrams may all be used when presenting architectural models. Structural model represents the framework for the system and this framework is the place where all other components exist.

Structural modelling captures the static features of a system. They consist of the following –

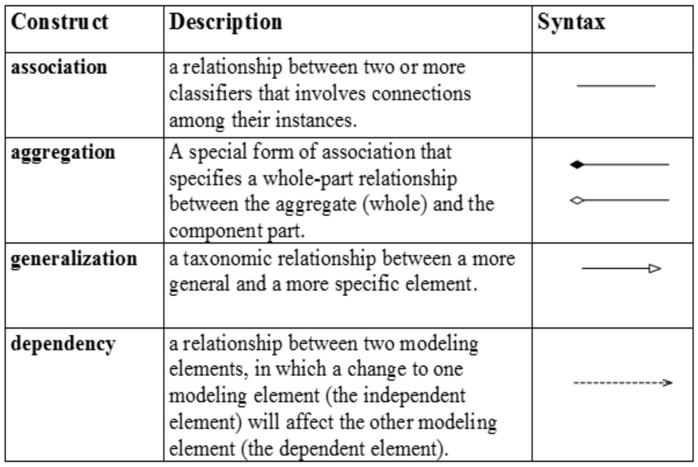
- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural Modelling: Core Elements-



Construct	Description	Syntax
class	a description of a set of objects that share the same attributes, operations, methods, relationships and semantics.	
interface	a named set of operations that characterize the behavior of an element.	«interface»
component	a modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces.	
node	a run-time physical object that represents a computational resource.	

Figure 2.1 (a): Structural Modelling Core Elements



Construct	Description	Syntax
association	a relationship between two or more classifiers that involves connections among their instances.	——
aggregation	A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.	◆—— ◇——
generalization	a taxonomic relationship between a more general and a more specific element.	——▷
dependency	a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).	- - - - ->

Figure 2.1 (b): Structural Modelling Core Relationships

Framework Models: An architecture framework is an encapsulation of a minimum set of practices and requirements for artifacts that describe a system's architecture. Models are representations of how objects fit in system and behave as part of the system. An architecture framework captures the "conventions, principles and practices for the description of architectures, established within a specific domain of application and/or community of stakeholders". A framework is usually implemented in terms of one or more viewpoints. Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

Dynamic Models: The dynamic model is used to express and model the behavior of the system over time. It includes support for activity diagrams, state diagrams, sequence diagrams and extensions including business process modelling. After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modeling.

Dynamic Modeling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world.”

The process of dynamic modeling can be visualized in the following steps −

- Identify states of each object.
- Identify events and analyze the applicability of actions.
- Construct a dynamic model diagram, comprising of state transition diagrams.
- Express each state in terms of object attributes.
- Validate the state–transition diagrams drawn.

Dynamic model is represented graphically with the help of state diagrams. It is also known as state modelling. State model consist of multiple state diagrams, one for each class with temporal behavior that is important to an application. State diagram relates with events and states. Events represents external functional activity and states represents values objects.

Events: An event is something that happen at a point in particular time such as a person press button or train 12345 departs from Indore. Event conveys information from one object to another.

The events are of three types: Signal event, Change event, and Time event.

Signal Event: A signal event is a particular occurrence in time. A signal is an explicit one-way transmission of information from one object to another such as sending or receiving signal. When an object send signal to another object it waits for acknowledgement, but acknowledgement signal is the separate signal under the control of second object, which may or may not choose to send it.

Change Event: It is caused by the satisfaction of a boolean expression. The intent of the change event is that the expression is tested continually whenever the expression changes from false to true.

Example: when (battery power < lower limit)

When (room temperature < heating/cooling point)

Time event: It is caused by occurrence of an absolute or the elapse of time interval. The UML notation for absolute time is the keyword when followed by a parenthesized expression involving time and for the time interval is keyword after followed by a parenthesized expression that evaluates time duration.

Example: when (Date = mar 2, 2005)

after (50 seconds)

State: A state is an abstraction of attribute values and links of an object. Values and links are combined into a state according to their entire behaviour. The response of object according to input event is called state. A state corresponds to the interval between two events received by an object. The state of the event depends on the past event. So basically, state represents intervals of time. The UML notation for the state is a round box containing an optional state name list, list the name in boldface, center the name near the top of the box, capitalize the first letter.

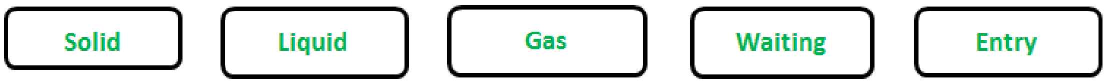
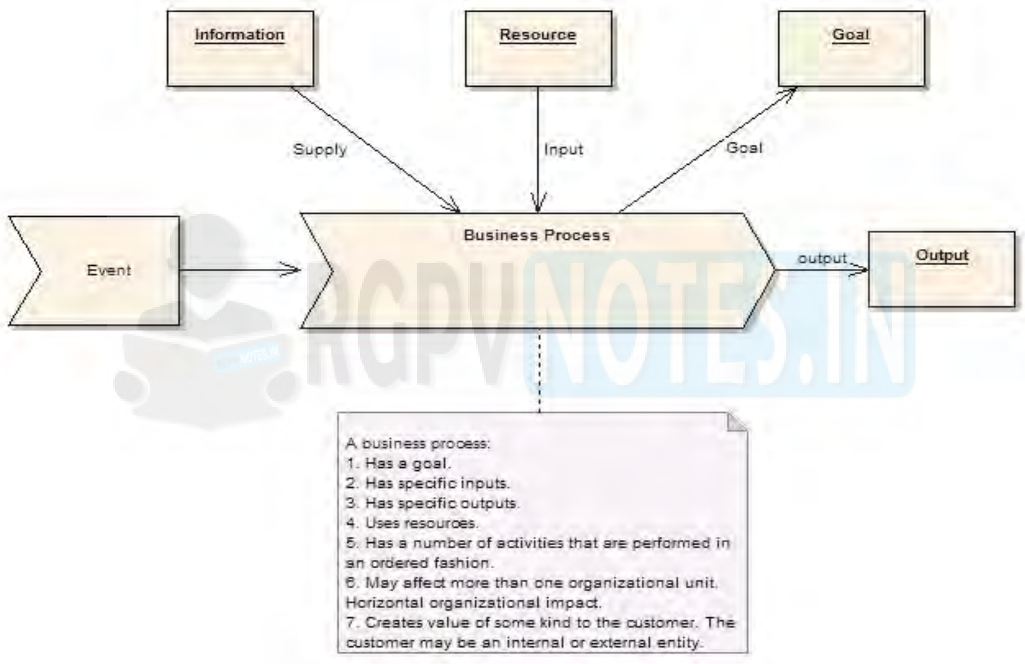


Figure 2.2: Example & representation of State

The following are the important points needs to be remembered about state.

1. Ignore attributes that do not affect the behaviour of object.
2. The objects in the class have finite number of possible states. Each object can be in one state at a time.
3. All events are ignored in a state, except those for which behaviour is explicitly prescribed.
4. Both events and states depend upon level of abstraction.

Process Models: A process model is a UML extension of an activity diagram used to model a business process - this diagram shows what goal the process has, the inputs, outputs, events and information that are involved in the process.



A business process:
1. Has a goal.
2. Has specific inputs.
3. Has specific outputs.
4. Uses resources.
5. Has a number of activities that are performed in an ordered fashion.
6. May affect more than one organizational unit. Horizontal organizational impact.
7. Creates value of some kind to the customer. The customer may be an internal or external entity.

Figure 2.3: Process Model

Architectures Styles: Architectural styles tell, how to organise our code. It's the highest level of granularity and it specifies layers, high-level modules of the application and how those modules and layers interact with each other, the relations between them. An Architectural Style can be implemented in various ways, with a specific technical environment, specific policies, frameworks or practices, or can say - The architectural styles that are used while designing the software.

Dataflow 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).

- Data Flow Architecture is transformed input data by a series of computational or manipulative components into output data. The data can be flow in the graph topology with cycles or in a linear structure without cycles.

- It is a part of Von-Neumann model of computation which consists of a single program counter, sequential execution and control flow which determines fetch, execution, commit order.
- Its main objective is to achieve the qualities of reuse and modifiability. And 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.

There are three types of execution sequences between modules–

1. Batch sequential
2. Pipe and filter or non-sequential pipeline mode
3. Process control

1. Batch Sequential:

- Batch sequential compilation was regarded as a sequential process in 1970. It is a classical data processing model.
- In Batch sequential, separate programs are executed in order and the data is passed as an aggregate from one program to the next.
- It provides simpler divisions on subsystems and each subsystem can be an independent program working on input data and produces output data.
- The main disadvantage of batch sequential architecture is that, it does not provide concurrency and interactive interface. It provides high latency and low throughput.

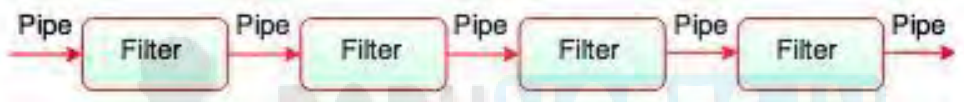


Figure 2.4: Flow of Batch Sequential Architecture

The above diagram shows the flow of batch sequential architecture. It provides simpler divisions on subsystems and each subsystem can be an independent program working on input data and produces output data.

Advantages

- Provides simpler divisions on subsystems.
- Each subsystem can be an independent program working on input data and producing output data.

Disadvantages

- Provides high latency and low throughput.
- Does not provide concurrency and interactive interface.
- External control is required for implementation.

2. Pipe and Filter Architecture:

This approach lays emphasis on the incremental transformation of data by successive component. In this approach, the flow of data is driven by data and the whole system is decomposed into components of data source, filters, pipes, and data sinks. The connections between modules are data stream which is first-in/first-out buffer that can be stream of bytes, characters, or any other type of such kind. The main feature of this architecture is its concurrent and incremented execution.

Pipe represents-

- Pipe is a connector which passes the data from one filter to the next.
- Pipe is a directional stream of data implemented by a data buffer to store all data, until the next filter has time to process it.
- It transfers the data from one data source to one data sink.
- Pipes are the stateless data stream.

Filter represents-

- A filter is a component and an independent entity or independent data stream transformer or stream transducers.
- It transforms and refines input data or input data stream, processes it, and writes the transformed data stream over a pipe for the next filter to process.
- It works in an incremental mode, in which it starts working as soon as data arrives through connected pipe.
- It has interfaces from which a set of inputs can flow in and a set of outputs can flow out.

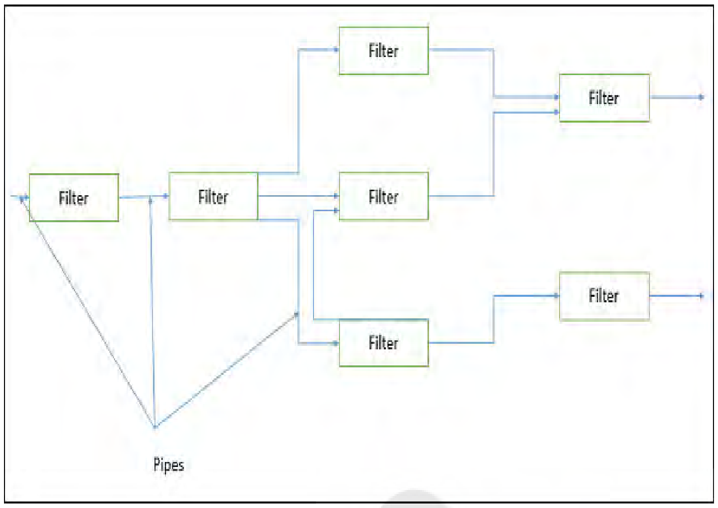


Figure 2.5: Pipes and Filter

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.

All filters are the processes that run at the same time, it means that they can run as different threads, coroutines or be located on different machines entirely. Each pipe is connected to a filter and has its own role in the function of the filter. The filters are robust where pipes can be added and removed at runtime. Filter reads the data from its input pipes and performs its function on this data and places the result on all output pipes. If there is insufficient data in the input pipes, the filter simply waits.

Advantages:

- Provides concurrency and high throughput for excessive data processing.
- Provides reusability and simplifies system maintenance.
- Provides modifiability and low coupling between filters.
- Provides flexibility by supporting both sequential and parallel execution.

Disadvantages:

- Not suitable for dynamic interactions.
- Overhead of data transformation between filters.
- Does not provide a way for filters to cooperatively interact to solve a problem.
- Difficult to configure this architecture dynamically.

3. Process Control: Process Control Architecture is a type of Data Flow Architecture, where data is neither batch sequential nor pipe stream. In process control architecture, the flow of data comes from a set of variables which controls the execution of process.

This architecture decomposes the entire system into subsystems or modules and connects them. Types of Subsystems- A process control architecture would have a processing unit for changing the process control variables and a controller unit for calculating the amount of changes.

A controller unit must have the following elements −

- **Controlled Variable** – Controlled Variable provides values for the underlying system and should be measured by sensors. For example, speed in cruise control system.
- **Input Variable** – Measures an input to the process. For example, temperature of return air in temperature control system
- **Manipulated Variable** – Manipulated Variable value is adjusted or changed by the controller.
- **Process Definition** – It includes mechanisms for manipulating some process variables.
- **Sensor** – Obtains values of process variables pertinent to control and can be used as a feedback reference to recalculate manipulated variables.
- **Set Point** – It is the desired value for a controlled variable.
- **Control Algorithm** – It is used for deciding how to manipulate process variables.

Application Areas:

Process control architecture is suitable in the following domains –

- Embedded system software design, where the system is manipulated by process control variable data.
- Applications, which aim is to maintain specified properties of the outputs of the process at given reference values.
- Applicable for car-cruise control and building temperature control systems.
- Real-time system software to control automobile anti-lock brakes, nuclear power plants, etc.

Call-and Return Architecture: A call and return architecture enables software designers to achieve a program structure, which can be easily modified. This style consists of the following two substyles.

1. Main program/subprogram architecture: In this, function is decomposed into a control hierarchy where the main program invokes a number of program components, which in turn may invoke other components.
2. Remote procedure call architecture: In this, components of the main or subprogram architecture are distributed over a network across multiple computers.

Call and Return (Functional):

- Routines correspond to units of the task to be performed.
- Combined through control structures.
- Routines known through interfaces (argument list)

Advantages:	Disadvantages:
• Architecture based on well-identified parts of the task. • Change implementation of routine without affecting clients. • Reuse of individual operations.	• Must know which exact routine to change. • Hides role of data structure. • Bad support for extendibility.

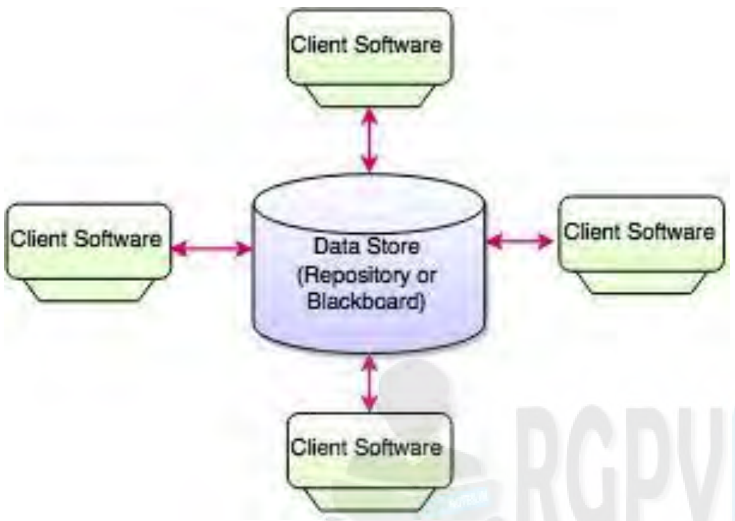
Call and Return (Object-Oriented):

- A class describes a type of resource and all accesses to it (encapsulation).
- Representation hidden from client classes.

Advantages:	Disadvantages:
• Change implementation without affecting clients. • Can break problems into interacting agents. • Can distribute across multiple machines or networks.	• Objects must know their interaction partners; when partner changes, clients must change. • Side effects: if A uses B and C uses B, then C's effects on B can be unexpected to A.

Data-Centered Architecture: Data Centered Architecture is also known as Database Centric Architecture. It is a layered process which provides architectural guidelines in data center development. This architecture is the physical and logical layout of the resources and equipment within a data center facility.

- In data-centred architecture, the data is centralized and accessed frequently by other components, which modify data.
- It consists of different components that communicate through shared data repositories.
- The components access a shared data structure and are relatively independent, in that, they interact only through the data store.
- This architecture specifies how these devices will be interconnected and how physical and logical security workflows are arranged.



The most well-known examples of the data-centered architecture is a database architecture, in which the common database schema is created with data definition protocol – for example, a set of related tables with fields and data types in an RDBMS.

Another example of data-centered architectures is the web architecture which has a common data schema (i.e. meta-structure of the Web) and follows hypermedia data model and processes communicate through the use of shared web-based data services.

Figure 2.6: Data-Centered Architecture

The above figure shows the architecture of Data-centred Architecture. In this architecture, the data is centralized and accessed frequently by other components which modify the data. The main purpose of data centred architecture is to achieve integrity of data.

There are two types of Components: Central Data and Data Accessor

Central Data:

- Central data provides permanent data storage.
- Central data represents the current state.

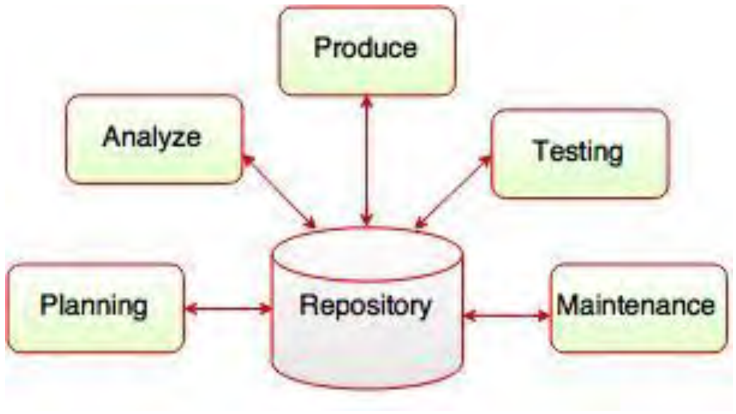
Data Accessor:

- Data accessor is a collection of independent components.
- It operates on the central data store, performs computations and displays the results.
- Communication can be done between the data accessors is only through the data store.

There are two categories which differentiates the architecture flow of control: Repository Architecture Style and Blackboard Architecture Style.

Repository Architecture Style:

- Repository architecture is a collection of independent components which operate on central data structure. It includes central data structure.
- Information System, Programming Environments, Graphical Editors, AI Knowledge Bases, Reverse Engineering System are the examples of Repository Architecture Style.



Repository architecture style is very important for data integration introduced in a variety of applications including software development, CAD etc.

In this architecture, the data store is passive, and the software clients or components of the data store are active which controls the logic flow and checks the data store for changes.

Figure 2.7: Repository Architecture style

Advantages:

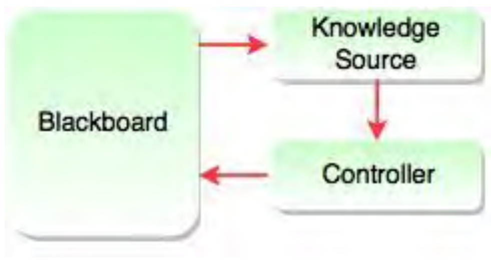
- Repository Architecture Style provides data integrity, backup and restore features.
- It reduces overhead of transient data between software components.
- It has an efficient way to store large amount of data.
- It has a centralized management which consists of backup, security and concurrency control.

Disadvantages:

- In repository architecture style, evolution of data is difficult and expensive.
- It has high dependency between data structure of data store and its software components or clients.

Blackboard Architecture Style:

- Blackboard architecture style is an artificial intelligence approach which handles complex problem, where the solution is the sum of its parts.
- Blackboard architecture style has a blackboard component which acts as a central data repository.
- It is used in location-locomotion, data interpretation and environmental changes for solving the problem.
- It is an approach to processing agent communication centrally.



There are major three components:

1. Knowledge Source
2. Blackboard
3. Control Shell

Figure 2.8: Blackboard Architecture Style

1. Knowledge Source:

- Knowledge source is also known as Listeners or Subscribers. They are distinct and independent units.
- Knowledge source solves the problem and aggregate partial results.

2. Blackboard:

- Blackboard is a shared repository of problems, solutions, suggestions and contributed information.
- It can be thought of as a dynamic library of an information to the current problem which have been published by other knowledge sources.

3. Control Shell:

- In Control shell, it controls the flow of problem-solving activity in the system.

- It manages the task and checks the work state.

Advantages:

- Blackboard architecture style provides concurrency which allows knowledge sources to work in parallel.
- This architecture supports experimentation for hypotheses and reusability of knowledge source components.
- It allows blackboard applications to adapt to changing requirements.
- It allows the new knowledge sources which can be developed and applied to the system without affecting on the existing system.

Disadvantages:

- Blackboard architecture style has the provision of tight dependency between the blackboard and knowledge source.
- It has difficulty in deciding for reasoning termination.
- It has an issue in synchronization of multiple agents.

Layered Architecture:

A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.

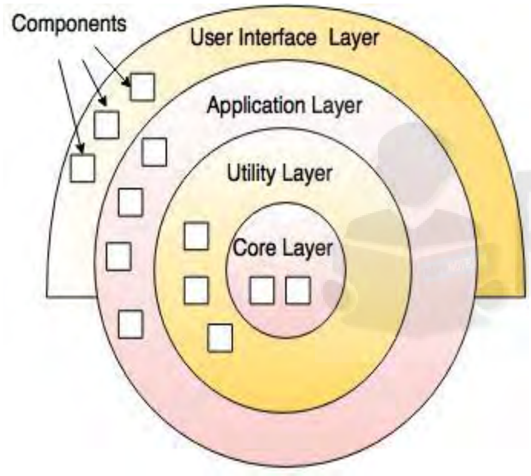


Figure 2.9: Layered architecture

- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS).
- Intermediate layers to utility services and application software functions.
- The components of outer layer manage the user interface operations.
- Components execute the operating system interfacing at the inner layer. The inner layers are application layer, utility layer and the core layer.

Agent Based Architecture: Now a days an increasing number of software projects are revised, restructured, and reconstructed in terms of software agents. Agent based software development a new way of analysis and synthesis of software system. Here software agents are new experimental embodiment of computer program.

- An agent in computer science refers to a software or other computational entities which has intelligence characteristics and can decide, and act based on its intelligence and other information taken from its environment. An agent usually acts on behalf of computer user.
- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

Application of Agent based System:

- An agent become a part of distributed system, as a processing node.
- Agents for distributed sensing, and information retrieval and management.
- Agents for e-commerce, Agents for human-computer interfaces.
- Agents for virtual environments, Agents for social simulation.
- Agents for industrial systems management.

Micro-services Architecture: "Microservices" - yet another new term on the crowded streets of software architecture. The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms.

- The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.
- A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability.
- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.

Benefits:

- Agility. Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process.
- Small, focused teams. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- Small code base. In a monolithic application, there is a tendency over time for code dependencies to become tangled, adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.
- Fault isolation. If an individual microservice becomes unavailable, it won't disrupt the entire application.
- Scalability. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application.
- Data isolation. It is much easier to perform schema updates, because only a single microservice is affected.

Challenges: The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- Complexity: A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- Development and testing: Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies.
- Network congestion and latency: The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem.
- Data integrity: With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- Versioning: Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so design carefully.

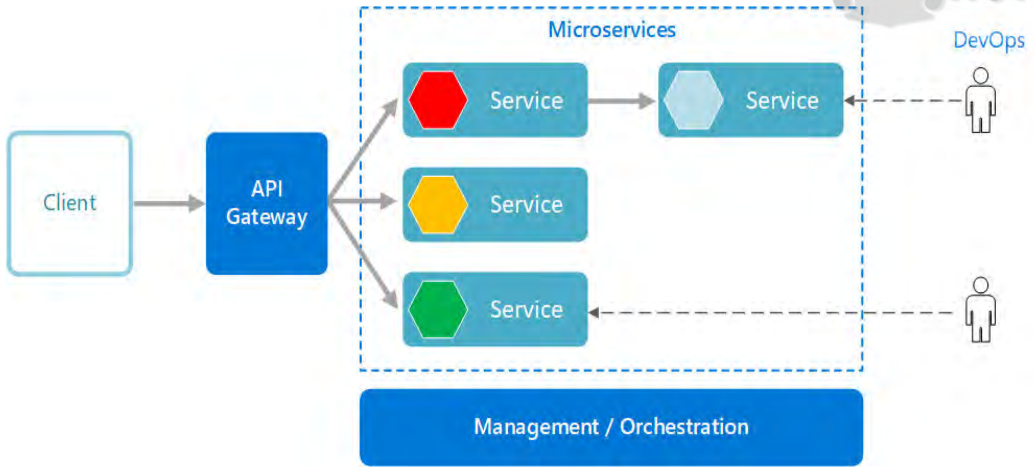


Figure 2.10: Micro-services architecture

Reactive Architecture: Reactive programming is an asynchronous programming paradigm, concerned with streams of information and the propagation of changes. Reactive Architecture is nothing more than the combination of reactive programming and software architectures. Also known as reactive systems, the goal is to make the system responsive, resilient, elastic, and message driven.

A Reactive system is an architectural style that allows multiple individual applications to merge as a single unit, reacting to its surroundings while aware of each other, and enable automatic scale up and down, load balancing, responsiveness under failure, and more.

Reactive Architecture can elastically scale in the face of varying incoming traffic. Scaling usually serves one of two purposes: either we need to scale out (by adding more machines) and up (by adding beefier machines), or we need to scale down, reducing the number of resources occupied by our application.

Reactive Architecture Benefits:

- Be responsive to interactions with its users.
- Handle failure and remain available during outages.
- Strive under varying load conditions.
- Be able to send, receive, and route messages in varying network conditions.
- Systems built as Reactive Systems are more flexible, loosely-coupled and scalable.

Representational State Transfer Architecture: Representational State Transfer (REST) is a style of architecture based on a set of principles that describe how networked resources are defined and addressed. These principles were first described in 2000 by Roy Fielding as part of his doctoral dissertation. REST is an alternative to SOAP and JavaScript Object Notation (JSON).

It is important to note that REST is a style of software architecture as opposed to a set of standards. As a result, such applications or architectures are sometimes referred to as *RESTful* or *REST-style* applications or architectures. Interaction in REST based systems happen through Internet's Hypertext Transfer Protocol (HTTP).

An application or architecture considered RESTful or REST-style is characterized by:

- State and functionality are divided into distributed resources.
- Every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet).
- The protocol is client/server, stateless, layered, and supports caching.

A Restful system consists of a:

- client who requests for the resources.
- server who has the resources.

Architectural Constraints of RESTful API: There are six architectural constraints which makes any web service are listed below:

1. Client-server
2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand (optional)

Client Server: Separation of concerns is the principle behind the client-server constraints.
Stateless: Statelessness means communication must be stateless in nature as in the client stateless server style, i.e. each request from client to server must contain all of the information necessary to understand the request and cannot take advantage of any stored context on the server.
Cacheable: In order to improve network efficiency, cache constraints are added to the REST style.
Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
Uniform interface: The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components.
Layered system: The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
Code on demand: The final addition to our constraint set for REST comes from the code-on-demand style.
REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility.
Steps Creating a RESTful service:

1. Define the domain and data.
2. Organize the data into groups.
3. Create URI to resource mapping.
4. Define the representations to the client (XML, HTML, CSS, …).
5. Link data across resources (connectedness or hypermedia).
6. Create use cases to map events/usage.
7. Plan for things going wrong.