**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 also known as UML- Architecture

Software architecture is all about how a software system is built at its highest level. It is needed to think big from multiple perspectives with quality and design in mind. The software team is tied to many practical concerns, such as:

* The structure of the development team.
* The needs of the business.
* Development cycle.
* The intent of the structure itself.

Software architecture provides a basic design of a complete software system. It defines the elements included in the system, the functions each element has, and how each element relates to one another. In short, it is a big picture or overall structure of the whole system, how everything works together.

To form architecture, the software architect will take several factors into consideration:

* What will the system are used for?
* Who will be using the system?
* What quality matters to them?
* Where will the system run?

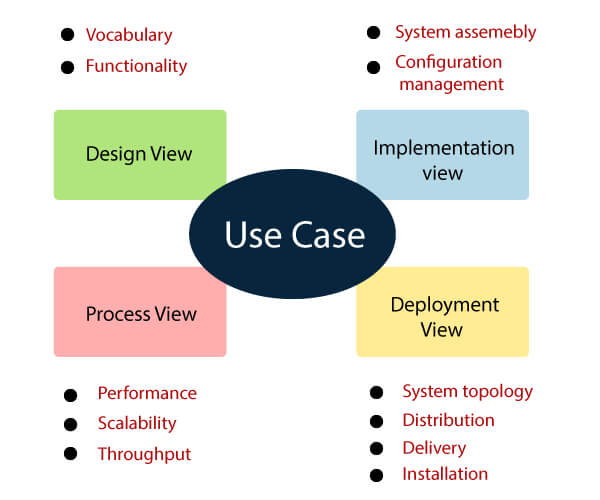
The architect plans the structure of the system to meet the needs like these. It is essential to have proper software architecture, mainly for a large software system. Having a clear design of a complete system as a starting point provides a solid basis for developers to follow.

Each developer will know what needs to be implemented and how things relate to meet the desired needs efficiently. One of the main advantages of software architecture is that it provides high productivity to the software team. The software development becomes more effective as it comes up with an explained structure in place to coordinate work, implement individual features, or ground discussions on potential issues. With a lucid

architecture, it is easier to know where the key responsibilities are residing in the system and where to make changes to add new requirements or simply fixing the failures.

In addition, a clear architecture will help to achieve quality in the software with a well-designed structure using principles like separation of concerns; the system becomes easier to maintain, reuse, and adapt. The software architecture is useful to people such as software developers, the project manager, the client, and the end-user. Each one will have different perspectives to view the system and will bring different agendas to a project. Also, it provides a collection of several views. It can be best understood as a collection of five views:

1. Use case view
2. Design view
3. Implementation view
4. Process view
5. Development view



### Use case view

* It is a view that shows the functionality of the system as perceived by external actors.
* It reveals the requirements of the system.
* With UML, it is easy to capture the static aspects of this view in the use case diagrams, whereas it’s dynamic aspects are captured in interaction diagrams, state chart diagrams, and activity diagrams.

### Design View

* It is a view that shows how the functionality is designed inside the system in terms of static structure and dynamic behavior.
* It captures the vocabulary of the problem space and solution space.
* With UML, it represents the static aspects of this view in class and object diagrams, whereas its dynamic aspects are captured in interaction diagrams, state chart diagrams, and activity diagrams.

### Implementation View

* It is the view that represents the organization of the core components and files.
* It primarily addresses the configuration management of the system’s releases.
* With UML, its static aspects are expressed in component diagrams, and the dynamic aspects are captured in interaction diagrams, state chart diagrams, and activity diagrams.

### Process View

* It is the view that demonstrates the concurrency of the system.
* It incorporates the threads and processes that make concurrent system and synchronized mechanisms.
* It primarily addresses the system's scalability, throughput, and performance.
* Its static and dynamic aspects are expressed the same way as the design view but focus more on the active classes that represent these threads and processes.

### Deployment View

* It is the view that shows the deployment of the system in terms of physical architecture.
* It includes the nodes, which form the system hardware topology where the system will be executed.
* It primarily addresses the distribution, delivery, and installation of the parts that build the physical system.

1. **Structural Models** Illustrates architecture as an ordered collection of program components. In other word 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, or **dynamic models**, which show the organization of the system when it is executing. You create structural models of a system when you are discussing and designing the system architecture.

**UML class** diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. An object class can be thought of as a general definition of one kind of system object. An association is a link between classes that indicates that there is some relationship between these classes. When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

|  |  |
| --- | --- |
| https://cs.ccsu.edu/~stan/classes/CS410/Notes16/images/05-class_diagram.png | https://cs.ccsu.edu/~stan/classes/CS410/Notes16/images/05-class_diagram_consultation.png |

1. **Framework Models** Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction. In other word, a software framework is a concrete or conceptual platform where common code with generic functionality can be selectively specialized or overridden by developers or users. Frameworks take the form of libraries, where a well-defined application program interface (API) is reusable anywhere within the software under development.

**Certain features** make a framework different from other library forms, including the following:

Ÿ **Default Behavior**: Before customization, a framework behaves in a manner specific to the user’s action.

Ÿ **Inversion of Control**: Unlike other libraries, the global flow of control within a framework is employed by the framework rather than the caller.

Ÿ **Extensibility**: A user can extend the framework by selectively replacing default code with user code.

Ÿ **Non-modifiable Framework Code**: A user can extend the framework but not modify the code. The purpose of software framework is to simplify the development environment, allowing developers to dedicate their efforts to the project requirements, rather than dealing with the framework’s mundane, repetitive functions and libraries

1. **Dynamic Modelling** - Specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment. **In** other word, Dynamic Modeling represents the temporal aspects of a system, capturing the control elements through which the behavior of objects can be understood over time. The Dynamic Model describes those aspects of a system concerned with time and the sequencing of operations - events that mark changes, sequences of events, and the organizing of events and states. The Dynamic Model does not consider what the operations do, what they operate on, or how they are implemented.

The Dynamic Model is represented graphically by a set of State Diagrams (called State Charts in UML). Each State Diagram (and its sub-diagrams) models all the possible state and event sequences permitted for one class of objects in response to external and internal events and can be seen as a behavioral template for the class it is modeling. It is necessary to model only those objects with a definite lifecycle or those which exhibit significant behavior. A State Diagram shows the states and the transitions between states through which an object passes during its lifetime, together with its responses to external events.

State Diagrams provide a rich syntax for Dynamic Modeling, including the ability to model:

• **States**—comprise Atomic States, Sequential States, Concurrent States and Submachine States that represent the condition or situation of an object at some point during its lifecycle, during which it satisfies a condition, performs an action or waits for an event.

• **Transitions**—represent a relationship between two states, indicating that an object in the first state will enter the second state and perform specified actions when a specified event occurs.

• **Events**—represent a noteworthy occurrence that may trigger a State transition. Events are modeled through Event Action Blocks.

• **Actions**—correspond to functions from the functional model and specify responses to events. Actions are modeled through Event Action Blocks.

• **Guard Conditions**—a Boolean condition associated with a transition event that determines whether the transition actually occurs when the event fires. Guard Conditions are modeled through Event Action Blocks.

• **Pseudostates**—comprise Initial States, Final States, History States, Junction States, Entry States, Exit States, Fork States and Join States. Pseudostates are used to build up compound transitions.

• **Activities**—describe continuous state behavior and correspond to Operations, Activities or Activities in the Class Model.

**What is the definition of process modeling?**

Focuses on the design of the business or technical process, which must be implemented in the system.

in other word, Process modeling is the graphical representation of business processes or workflows. Like a flow chat, individual steps of the process are drawn out so there is an end-to-end overview of the tasks in the process within the context of the business environment.

Modeling allows visualization of business processes so organizations can better understand their internal business procedures so that they can be managed and made more efficient. This is usually an agile exercise for continuous improvement.

Process modeling is a vital component of process automation, as a process model needs to be created first to define tasks and optimize the workflow before it is automated.

Or a software process model is an abstraction of the software development process. The models specify the stages and order of a process. So, think of this as a representation of the **order of activities** of the process and the **sequence** in which they are performed.

**What are the benefits of using process modeling?**

The act of process modeling provides a visualization of business processes, which allows them to be inspected more easily, so users can understand how the processes work in their current state and how they can be improved. Other benefits from process modeling include:

**Improve efficiency** – process modeling helps to improve the process, helping business workers to be more productive by saving time

**Gain transparency –**modeling provides a clear overview of the process, identifying the start and end point and all the steps in between

**Ensure best practice –**using process models ensures consistency and standardization across the organization

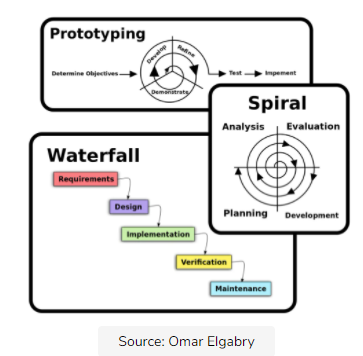
**Create understanding** – by using the common language of process, it makes it easier for users across the organization to communicate with each other

**Business orchestration** – supports the coordination of people, systems and information across the organization to support business strategy

## A model will define the following:

* The tasks to be performed
* The input and output of each task
* The pre and post conditions for each task
* The flow and sequence of each task

**The goal** of a software process model is to provide guidance for controlling and coordinating the tasks to achieve the end product and objectives as effectively as possible.



There are many kinds of process models for meeting different requirements. We refer to these as SDLC models (Software Development Life Cycle models). The most popular and important SDLC models are as follows:

|  |  |
| --- | --- |
| * Waterfall model * V model * Incremental model * RAD model | * Agile model * Iterative model * Prototype model * Spiral model |

## Factors in choosing a software process

Choosing the right software process model for your project can be difficult. If you know your requirements well, it will be easier to select a model that best matches your needs. You need to keep the following factors in mind when selecting your software process model:

## Project requirements

Before you choose a model, take some time to go through the project requirements and clarify them alongside your organization’s or team’s expectations. Will the user need to specify requirements in detail after each iterative session? Will the requirements change during the development process?

## Project size

Consider the size of the project you will be working on. Larger projects mean bigger teams, so you’ll need more extensive and elaborate project management plans.

## Project complexity

Complex projects may not have clear requirements. The requirements may change often, and the cost of delay is high. Ask yourself if the project requires constant monitoring or feedback from the client.

## Cost of delay

Is the project highly time-bound with a huge cost of delay, or are the timelines flexible.

## Customer involvement

Do you need to consult the customers during the process? Does the user need to participate in all phases?

## Familiarity with technology

This involves the developer’s knowledge and experience with the project domain, software tools, language, and methods needed for development.

## Project resources

This involves the amount and availability of funds, staff, and other resources.

## Architecture Style

Every software requires a proper plan and detailed blueprint before stepping into the development. Software architecture is the high level structure used for creating software systems and is actually a step-by-step blueprint of the entire software that is to be built. The purpose of the software and its specific functionalities are defined by the software's architectural style and pattern used.

The **architectural style** is a very specific solution to particular software, which typically focuses on how to organize the code created for the software. It focuses on creating the layers and modules of the software and allowing an appropriate interaction between the various modules for giving the right results upon implementation.

The **architectural pattern** is the description of relationship types and elements along with a set of constraints to implementing a software system. The patterns are usually reusable solutions for common problems or models.

There are various types of architectural styles followed for software creation. In this lesson, we discuss data-centric, object-oriented, and layered architectural patterns.

## The architectural style

* The architectural style is a transformation and it is applied to the design of an entire system.
* The main aim of architectural style is to build a structure for all components of the system.
* Architecture of the system is redefined by using the architectural style.
* An architectural pattern such as architectural style introduces a transformation on the design of architecture.
* The software is constructed for computer based system and it shows one of the architectural styles from many of style.

The architectural styles that are used while designing the software as follows:

1. **Data-centered architecture**

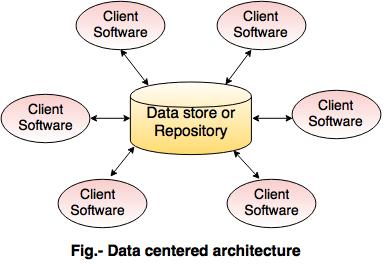
A data-centered architecture has two distinct components: a **central data structure**or data store (central repository) and a **collection of client software.**The data store (for example, a database or a file) represents the current state of the data and the client software performs several operations like add, delete, update, etc., on the data stored in the data store. In some cases, the data store allows the client software to access the data independent of any changes or the actions of other client software.

In this architectural style, new components corresponding to clients can be added and existing components can be modified easily without taking into account other clients. This is because client components operate independently of one another.

A variation of this architectural style is blackboard system in which the data store is transformed into a blackboardthat notifies the client software when the data (of their interest) changes. In addition, the [information](https://ecomputernotes.com/fundamental/information-technology/what-do-you-mean-by-data-and-information) can be transferred among the clients through the blackboardcomponent.

Some advantages of the data-centered architecture are listed below.

* Clients operate independently of one another.
* Data repository is independent of the clients.
* It adds scalability (that is, new clients can be added easily).
* It supports modifiability.
* It achieves data integration in component-based development using blackboard.
* **Th**e data store in the file or database is occupying at the center of the architecture.
* Store data is access continuously by the other components like an update, delete, add, modify from the data store.
* Data-centered architecture helps integrity.
* Pass data between clients using the blackboard mechanism.
* The processes are independently executed by the client components.



**2. Data-flow architecture**

Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations. Each component, known as **filter,**transforms the data and sends this transformed data to other filters for further processing using the connector, known as **pipe**. Each filter works as an independent entity, that is, it is not concerned with the filter which is producing or consuming the data. A

pipe is a unidirectional channel which transports the data received on one end to the other end. It does not change the data in anyway; it merely supplies the data to the filter on the receiver end.

|  |
| --- |
| Data-flow Architecture |

Most of the times, the data-flow architecture degenerates a batch sequential system. In this system, a batch of data is accepted as input andthen a series of sequential filters are applied to transform this data. One commonexample of this architecture is UNIX shell programs. In these programs, UNIXprocesses act as filters and the file system through which UNIX processes interact,act as pipes. Other well-known examples of this architecture are compilers, signal processingsystems, parallel programming, functional programming, and distributedsystems. Some advantages associated with the data-flow architecture are listedbelow.

* It supports reusability.
* It is maintainable and modifiable.
* It supports concurrent execution.
* Some disadvantages associated with the data-flow architecture are listed below.
* It often degenerates to batch sequential system.
* It does not provide enough support for applications requires user interaction.
* It is difficult to synchronize two different but related streams.
* **This** architecture is applied when the input data is converted into a series of manipulative components into output data.
* A pipe and filter pattern is a set of components called as filters.
* Filters are connected through pipes and transfer data from one component to the next component.
* The flow of data degenerates into a single line of transform then it is known as batch sequential.

**3. Call and return architectures:**

This architecture style allows achieving a program structure which is easy to modify.  
**Following are the sub styles exist in this category:  
1. Main program or subprogram architecture**

* The program is divided into smaller pieces hierarchically.
* The main program invokes many of program components in the hierarchy that program components are divided into subprogram.

1. **Remote procedure call architecture**

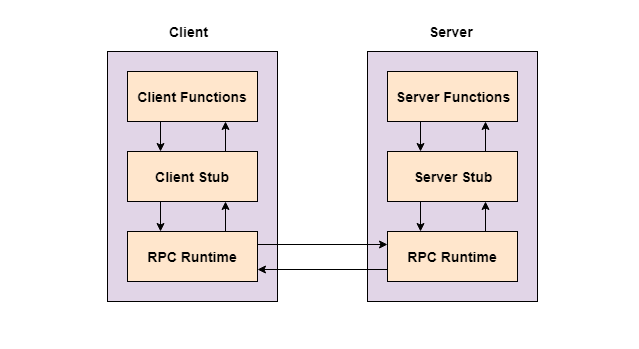
A remote procedure call is an inter-process communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.

A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

The sequences of events in a remote procedure call are given as follows −

* The client stub is called by the client.
* The client stub makes a system call to send the message to the server and puts the parameters in the message.
* The message is sent from the client to the server by the client’s operating system.
* The message is passed to the server stub by the server operating system.
* The parameters are removed from the message by the server stub.
* Then, the server procedure is called by the server stub.

A diagram that demonstrates this is as follows −



**Advantages of Remote Procedure Call**

Some of the advantages of RPC are as follows −

* Remote procedure calls support process oriented and thread oriented models.
* The internal message passing mechanism of RPC is hidden from the user.
* The effort to re-write and re-develop the code is minimum in remote procedure calls.
* Remote procedure calls can be used in distributed environment as well as the local environment.
* Many of the protocol layers are omitted by RPC to improve performance.

**Disadvantages of Remote Procedure Call**

Some of the disadvantages of RPC are as follows −

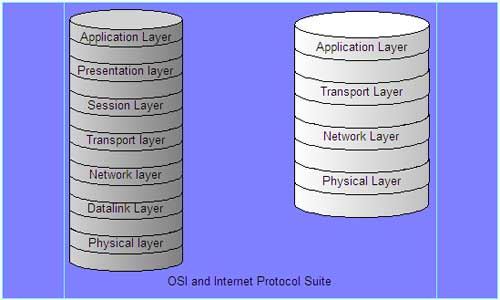
* The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
* There is no flexibility in RPC for hardware architecture. It is only interaction based.
* There is an increase in costs because of remote procedure call.
* The main program or subprogram components are distributed in network of multiple computers.
* The main aim is to increase the performance.

**Object-oriented architectures**

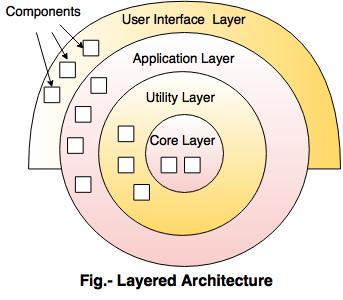
* This architecture is the latest version of call-and-return architecture.
* It consists of the bundling of data and methods.

1. **Layered architectures**

In layered architecture, several layers (components) are defined with each layer performing a well-defined set of operations. These layers are arranged in a hierarchical manner, each one built upon the one below it. Each layer provides a set of services to the layer above it and acts as a client to the layer below it. The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during Interaction. One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system.



* The different layers are defined in the architecture. It consists of outer and inner layer.
* 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.
* In many cases, it is possible that more than one pattern is suitable and the alternate architectural style can be designed and evaluated.



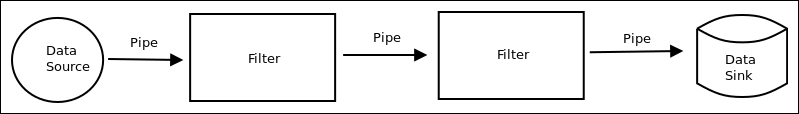
**Pipe and Filter style**

The Pipe and Filter is an architectural design pattern that allows for stream/asynchronous processing. In this pattern, there are many components, which are referred to as filters, and connectors between the filters that are called pipes. Each filter is responsible for applying a function to the given data; this is known as filtering. Filters can work asynchronously. The final output is given to the consumer, known as a sink. In other word

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:



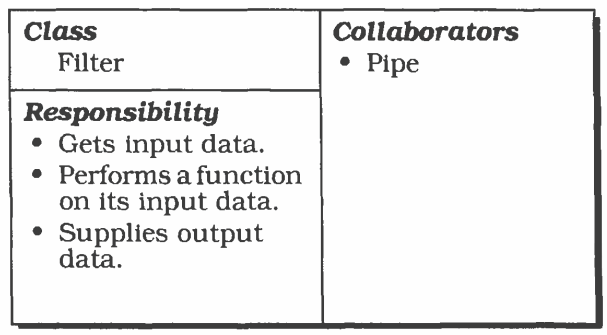
Pipe and Filter Architecture

**Filters**

Filter components are the processing units of the pipeline. A filter enriches, refines or transforms its input data. It enriches data by computing and adding information, refines data by concentrating or extracting information, and transforms data by delivering the data in some other representation. A concrete filter implementation may combine all three basic principles. The activity of a filter can be triggered by several events:

* The subsequent pipeline element pulls output data from the filter.
* The previous pipeline element pushes new input data to the filter.
* The filter is active in a loop, pulling its input from and pushing its output down the pipeline.

The first two cases denote *passive* filters, whereas the last case is an *active* filter. An **active** filter starts processing on its own as a separate program, process or thread. A **passive** filter component is activated by being called either as a function (*pull*) or as a procedure (*push*). See [Control Flow](https://homepages.fhv.at/thjo/lecturenotes/sysarch/pipes-and-filters.html#control-flow) for a more technical discussion of active and passive filters.



Filter responsibilities and collaborators (Figure taken from [[5](https://homepages.fhv.at/thjo/lecturenotes/sysarch/pipes-and-filters.html#ref-buschmann2008pattern)]).

Among the important invariants of the style is the condition that filters must be independent entities and should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specification might restrict what appears on their upstream and downstream filter. Their specification might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes. Furthermore, the correctness of the output of a Pipes and Filters network should not depend on the order in which the filters perform their incremental processing (given fair scheduling).

**Summarising Filters:**

Filters must be independent with no shared state.

* Filters don’t know and must not make assumptions about upstream or downstream filter identity.
* Correctness of output from network must not depend on execution order in which individual filters provided their incremental processing (fair scheduling).
* Emphasise the port concept as they have input ports (for incoming data) and output ports (for outgoing data).
* Ports are connection points of a filter
* The port data are typed, and may follow a protocol: start and end of a (sub) stream.
* Read streams of data on input producing streams of data on output.
* **Ideally**: local incremental transformation to input stream that means, output usually begins before input is consumed.
* The computation is done incrementally and locally.
* **Incrementally**: the portion of data that is available at the input ports is transformed.

**Locally**: no outside influence except through ports.

* **Enrich input data**: e.g. add product details to product ID.
* **Refine input data**: e.g. filter out uninteresting or redundant data.
* **Compose input data**: e.g. streams of words to sentences.
* **Transform input data**: e.g. transform units of values.
* **Analyze data**: e.g. counting, statistics, identification, classification.
* **Active filters**: drive the data flow on the pipes (a pump): pulls autonomously its input from upstream and pushes it downstream.
* **Passive filters**: is driven by the data flow on the pipes: gets pushed or pulled.

**Pipes**

*Pipes* denote the connections between filters, between the data source and the first filter, and between the last filter and the data sink. If two active components are joined, the pipe synchronizes them with a FIFO buffer. If activity is controlled by one of the adjacent filters, the pipe can be implemented by a direct call from the active to the passive component - direct calls make filter recombination harder, however.

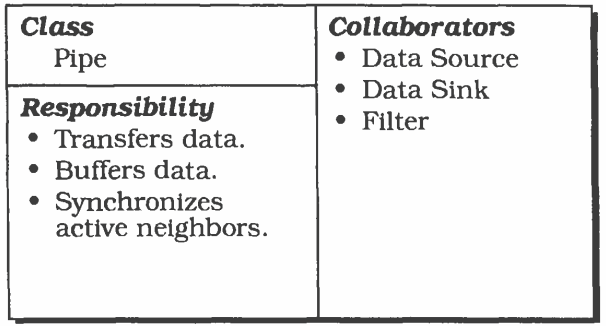


Figure 2.4: Pipe responsibilities and collaborators (Figure taken from [[5](https://homepages.fhv.at/thjo/lecturenotes/sysarch/pipes-and-filters.html#ref-buschmann2008pattern)]).

**Summarizing Pipes:**

* + Conduits for streams, e.g. **first-in-first-out buffer**.
  + Transmit outputs of one filter to the input of another.
  + May synchronize data flow between two filters.
  + A pipe usually is a first class object (can be represented and manipulated in its own right).
  + A pipe transfers data from one filter to the next.
  + A pipe may implement *a (bounded or unbounded) buffer.*
  + *Restrictions on their ends may be possible (for example in*typed pipes*), but: no data transformation!*
  + *A pipe may sit in between*
  + Two filter **objects/functions** in a single **process/thread**.
  + Two threads in a single process: e.g. **a Java Pipe**, stream may contain simultaneous references to shared objects.
  + Two processes on a single host: e.g. **a UNIX Named Pipe**.
  + Two processes in a distributed system: e.g. **a Socket**.

**Agent base architecture:**

## What is an Agent?

An agent can be anything that perceiveits environment through sensors and act upon that environment through actuators. An Agent runs in the cycle of **perceiving**, **thinking**, and **acting**. An agent can be:

* **Human-Agent:** A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
* **Robotic Agent:** A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.
* **Software Agent:** Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

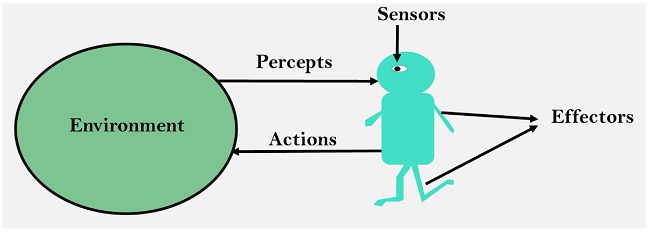
Hence the world around us is full of agents such as thermostat, cellphone, camera, and even we are also agents.

Before moving forward, we should first know about sensors, effectors, and actuators.

**Sensor:** Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.

**Actuators:** Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.

**Effectors:** Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.



## Intelligent Agents:

An intelligent agent is an autonomous entity which act upon an environment using sensors and actuators for achieving goals. An intelligent agent may learn from the environment to achieve their goals. A thermostat is an example of an intelligent agent.

Following are the main four rules for an AI agent:

* **Rule 1:** An AI agent must have the ability to perceive the environment.
* **Rule 2:** The observation must be used to make decisions.
* **Rule 3:** Decision should result in an action.
* **Rule 4:** The action taken by an AI agent must be a rational action.

## Rational Agent:

A rational agent is an agent which has clear preference, models uncertainty, and acts in a way to maximize its performance measure with all possible actions.

A rational agent is said to perform the right things. AI is about creating rational agents to use for game theory and decision theory for various real-world scenarios.

For an AI agent, the rational action is most important because in AI reinforcement learning algorithm, for each best possible action, agent gets the positive reward and for each wrong action, an agent gets a negative reward.

## Structure of an AI Agent

The task of AI is to design an agent program which implements the agent function. The structure of an intelligent agent is a combination of architecture and agent program. It can be viewed as:

**Agent = Architecture + Agent program**

**Following** are the main three terms involved in the structure of an AI agent:

**Architecture:** Architecture is machinery that an AI agent executes on.

**Agent Function:** Agent function is used to map a percept to an action.

**Agent** architecture has been one of the core components in building an agent application. Agent architecture is considered as the functional brain of an agent in making decision and reasoning to solve problem and achieving goals.

Agent-based technology provides a new computing paradigm, where intelligent agents can be used to perform tasks such as sensing, planning, scheduling, reasoning and decision-making. In an agent-based system, software agents with sufficient intelligence and autonomy can either work independently or co-ordinately with other agents to accomplish tasks and missions. In this book, we provide up-to-date practical applications of agent-based technology in various fields, such as electronic commerce, grid computing, and adaptive virtual environment. The selected applications are invaluable for researchers and practitioners to understand the practical usage of agent-based technology, and also to apply agent-based technology innovatively in different areas.

**Micro-service architecture**,

Micro-Services, is a specific method of designing software systems to structure a single application as a collection of loosely coupled services. Applications tend to begin as a monolithic architecture, and over time grow into a set of interconnected micro-services.

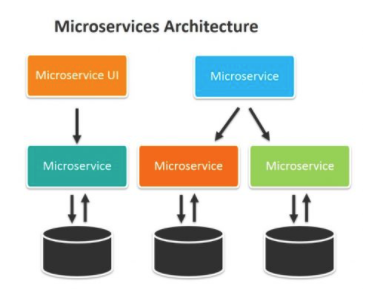
The main idea behind micro-services is that some types of applications are easier to build and maintain when they are broken down into many small pieces that work together. Though the architecture has increased complexity, micro-services still offer many advantages over the monolithic structure.

The concept of micro stems from the existing monolithic infrastructure most companies came up using, especially if the company has been around for a decade or longer. Instead of a monolithic architecture, each component of micro-service architecture has:

* Its own CPU
* Its own runtime environment
* Often, a dedicated team working on it, ensuring each service is distinct from one another

**This architecture means each service can:**

* Run its own unique process
* Communicate autonomously without having to rely on the other micro-services or the application as a whole

This ability to be separated and recombined protects the entire system against decay and better facilitates [agile processes](https://www.bmc.com/blogs/agile-vs-waterfall/), making it appealing to organizations—especially those still utilizing monolithic infrastructures.

## Advantages to Micro-services

Applications built as a set of independent, modular components are easier to test, maintain, and understand. They enable organizations to:

* Increase agility
* Improve workflows
* Decrease the amount of time it takes to improve production

While each independent component increases complexity, the component can also have added monitoring capabilities to combat it.

Here are the most common pros of micro-services, and why so many enterprises already use them.

**Developer independence**

Each micro-service will often be assigned a [single dev team](https://www.bmc.com/blogs/it-teams/) to maintain it. Thus, there is greater developer freedom and independence. Small teams that are working in parallel can iterate faster than larger teams. When a single service takes off in popularity, the smaller team can also scale the services on their own without having to wait for a larger and more complex team.

**Isolation & resilience**

If one of the components should fail, due to issues like outdated technology or inability to further develops the code, developers are able to spin up another component while the rest of the application continues to function independently. This capability gives developers the freedom to develop and deploy services as needed, without having to wait on decisions concerning the entire application.

**Scalability**

Because micro-services are made of much smaller components, they can take up fewer resources and therefore more easily scale to meet increasing demand of that specific component.

As a result of their isolation, micro-services can properly function even during large changes in size and volume, making it ideal for enterprises dealing with a wide range of platforms and devices.

**Autonomously developed**

As opposed to monoliths, individual components are much easier to fit into continuous delivery pipelines and complex deployment scenarios. Only the pinpointed service needs to be modified and redeployed when a change is needed. If a service should fail, the others will continue to function independently.

**Its autonomous nature benefits teams because it:**

* Enables scaling and development
* Doesn’t require much coordination with other teams

Micro-services are a particular advantage when companies become more distributed and workers more remote.

**Relationship to the business**

Micro-service architectures are split along business domain boundaries, organized around capabilities such as logistics, billing, etc. This increases independence and understanding across the organization: different teams are able to utilize a specific product and then own and maintain it for its lifetime.

**Evolutionary**

Any micro-service architecture is highly evolutionary.

Micro-services are an excellent option for situations where developers can’t fully predict what devices will be accessed by the application in the future. They also allow quick and controlled changes to the software without slowing the application as a whole—so you can be more iterative in developing features and new products.

## Representational State Transfer

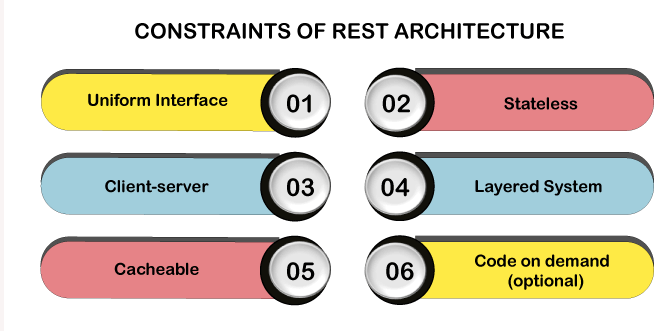
REST, or Representational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. REST-compliant systems, often called Restful systems, are characterized by how they are stateless and separate the concerns of client and server.

In the REST architectural style, the implementation of the client and the implementation of the server can be done independently without each knowing about the other. This means that the code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.

As long as each side knows what format of messages to send to the other, they can be kept modular and separate. Separating the user interface concerns from the data storage concerns; we **improve the flexibility of the interface across platforms** and **improve scalability by simplifying the server components**. Additionally, the separation allows each component the ability to evolve independently.

By using a REST interface, different clients hit the same REST endpoints, perform the same actions, and receive the same responses.

**Representational State Transfer** (REST) is a software architectural style that defines the constraints to create web services. The web service that follows the **REST** architectural style is called **RESTful Web Services**. It differentiates between the computer system and web services. The REST architectural style describes the **six** barriers.



### 1. Uniform Interface

The Uniform Interface defines the interface between **client** and **server**. It simplifies and decomposes the architecture which enables every part to be developed.

The **Uniform Interface** has **four** guiding principles:

1. **Resource-based:** Individual resources are identified using the URI as a resource identifier. The resources themselves are different from the representations returned to the customer. For example, the server cannot send the database but represents some database records expressed to **HTML, XML or JSON** depending on the server request and the implementation details.
2. **Manipulation of resources by representation:** When a client represents a resource associated with metadata, there is information on the server to modify or delete it.
3. **Self-Descriptive Message:** Each message contains enough information to describe how the message is processed. For example, the parser can be specified by the Internet media type (known as the **MIME** type).
4. **As the engine of Hypermedia Application State (HATEOAS):** Customers provide states by query-string parameters, body content, request headers, and requested URIs. The services provide customers with the state by **response codes, response headers** and **body content**. It is called hypermedia (**hyperlink within hypertext**).

* In addition to the above description, HATEOS also means that, where necessary, the object or itself is contained in the linked body (or header) to supply the URI for retrieving the related objects.
* The same interface that any **REST services** provide is fundamental to the design.

### 2. Client-server

A client-server interface separates the client from the server. For Example, the separation of concerns not having an internal relationship with internal storage for each server to improve the portability of customer's data codes. Servers are not connected with the user interface or user status to make the server simpler and scalable. Servers and clients are independently replaced and developed until the interface is changed.

### 3. Stateless

**Stateless** means the state of the service doesn't persist between subsequent requests and response. It means that the request itself contains the state required to handle the request. It can be a query-string parameter, entity, or header as a part of the **URI**. The URI identifies the resource and state (or state change) of that resource in the unit. After the server performs the appropriate state or status piece (s) that matters are sent back to the client through the header, status, and response body.

* Most of us in the industry have been accustomed to programming with a container, which gives us the concept of **"session,"** which maintains the status among multiple HTTP requests. In **REST**, the client may include all information to fulfil the server's request and multiple requests in the state. Statelessness enables greater scalability because the server does not **maintain, update**, or **communicate** any session state. The resource state is the data that defines a resource representation.

Example, the **data stored** in a database. Consider the application state of having data that may vary according to client and request. The resource state is constant for every customer who requests it.

### 4. Layered system

It is directly connected to the end server or by any intermediary whether a client cannot tell. Intermediate servers improve the system **scalability** by enabling **load-balancing** and **providing** a shared cache. Layers can enforce security policies.

### 5. Cacheable

On the World Wide Web, customers can cache responses. Therefore, responses clearly define themselves as unacceptable or prevent customers from **reusing** stale or **inappropriate data** to further requests. **Well-managed** caching eliminates some client-server interactions to improving scalability and performance.

### 6. Code on Demand (optional)

The server temporarily moves or optimizes the functionality of a client by logic that it executes. Examples of compiled components are **Java applets** and **client-side scripts**.

Compliance with the constraints will enable any distributed hypermedia system with desirable contingency properties such as **performance, scalability, variability, visibility, portability,** and **reliability**.

**Reactive systems architecture**

Reactive systems architecture is a computer systems paradigm that takes advantage of the *responsiveness*, *flexibility*and *resiliency*offered in reactive programming so that various components (e.g., software applications, databases and servers) can continue to function and even thrive if one of the components is compromised.

The reactive programming on which reactive systems architecture allows -- at its simplest expression -- data to automatically change when related data changes. So, for example, in the expression *a = b + c*, if either *b*or *c*changes, then the value of *a*will change automatically when a notification of the change to *b*or *c*to is sent. The change represented by *b*or *c*might be a change in availability, while *a*might be the price.

### How reactive systems architecture works

Reactive systems architecture adhere to the tenets of the [Reactive Manifesto](http://www.reactivemanifesto.org/), which characterizes systems created by reactive programming as "responsive," providing rapid and consistent response times; "resilient," meaning that the system stays responsive despite hardware or software failures; "elastic," wherein the system remains responsive in varying workloads; and "message driven," relying on asynchronous message-passing.

Proponents of reactive systems architecture say that the challenge is not in making systems that are simply responsive or flexible or elastic -- it is in making systems that are all three. So, a system must be responsive to change, e.g., to the variables *b*or *c*in the example above. It must also be elastic; accommodating rises and falls in load, e.g., an online shopping site that experiences peak demand times. And it must also be resilient, built to minimize and recover from application and system failure.

Reactive systems architecture is also said to use software architectural patterns that promote self-healing and self-monitoring systems that can repair themselves.