# SOLID principles are 5 concepts

- Single Responsibility Principle
- Open / Closed Principles
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

## Single Responsibility Principle:

- The single responsibility principle says that each of our classes has to be only used for one purpose.
- We need this so that we don't have to change code as often when something changes. It's also hard to understand what the class is doing if it's doing many things.
- Unrelated concepts in the same class also make comprehending the purpose of the code harder.

## Open / Closed Principle:

- The open/closed principle states that a piece of software is open for extension but closed for modification.
- This means that we should be able to add more functionality without changing existing code.

## Liskov Substitution Principle:

This principle states that if we have a parent class and a child class, then we can interchange the parent and child class without getting incorrect results.

This means that the child class must implement everything that's in the parent class. The parent class serves as the base members that child classes extend from.

For example, if we want to implement classes for a bunch of shapes, we can have a parent Shape class, which is extended by all classes by implementing everything in the Shape class.

## Interface Segregation Principle:

The interface segregation principle states that "clients shouldn't be forced to depend on interfaces that they don't use."

This means that we shouldn't impose the implementation of something if it's not needed.

## Dependency Inversion Principle:

This principle states that high-level modules shouldn't depend on low-level modules and they both should depend on abstractions, and abstractions shouldn't depend upon details. Details should depend upon abstractions.

This means that we shouldn't have to know any implementation details of our dependencies. If we do, then we violated this principle.

We need this principle because if we do have to reference the code for the implementation details of a dependency to use it, then when the dependency changes, there's going to be lots of breaking changes to our own code.

As software gets more complex, if we don't follow this principle, then our code will break a lot.

# Scheduling Algorithms

| Algorithm | Characteristics | Advantages | Disadvantages |
|---|---|---|---|
| **1. First Come First Serve (FCFS)** | FCFS is considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue. | • Easy to implement<br>• First come, first serve method | • FCFS suffers from the Convoy effect.<br>• The average waiting time is much higher than the other algorithms.<br>• FCFS is very simple and easy to implement and hence not very efficient. |
| **2. Shortest Job First(SJF)** | Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. | • As SJF reduces the average waiting time thus, it is better than the first come first serve scheduling algorithm.<br>• SJF is generally used for long term scheduling | • One of the demerits SJF has is starvation.<br>• Many times it becomes complicated to predict the length of the upcoming CPU request |
| **4. Priority Scheduling** | Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process. In this | • The average waiting time is less than FCFS<br>• Less complex | One of the most common demerits of the Preemptive priority CPU scheduling algorithm is the Starvation Problem. This is the problem in |

| | | | |
|---|---|---|---|
| | algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first. In the case of any conflict, that is, where there are more than one processor with equal value, then the most important CPU planning algorithm works on the basis of the FCFS (First Come First Serve) algorithm. | | which a process has to wait for a longer amount of time to get scheduled into the CPU. This condition is called the starvation problem. |
| **4. Round robin** | Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique. | ● Round robin seems to be fair as every process gets an equal share of CPU.<br>● The newly created process is added to the end of the ready queue. | |
| **5. Shortest Remaining Time First** | Shortest remaining time first is the preemptive version of the Shortest job first which we have discussed earlier where the processor is allocated to the job closest to completion. In SRTF the process with the smallest amount of time remaining until | ● In SRTF the short processes are handled very fast.<br>● The system also requires very little overhead since it only makes a decision when a process | ● Like the shortest job first, it also has the potential for process starvation.<br>● Long processes may be held off indefinitely if short processes are continually |

|  |  |  |  |
|---|---|---|---|
|  | completion is selected to execute. | completes or a new process is added. | added. |

## SemiStructured DataBase

### What is Semi-Structured Data?

Semi-structured data refers to data that is not captured or formatted in conventional ways. Semi-structured data does not follow the format of a tabular data model or relational databases because it does not have a fixed schema. However, the data is not completely raw or unstructured, and does contain some structural elements such as tags and organizational metadata that make it easier to analyze. The advantages of semi-structured data is that it is more flexible and simpler to scale compared to structured data.

### What are Examples of Semi-Structured Data?

HTML code, graphs and tables, e-mails, XML documents are examples of semi-structured data, which are often found in object-oriented databases.

## Design Patterns

The design pattern is an essential element in object-oriented programming.

It is a software infrastructure made up of a small number of classes that is used to solve a technical problem.

Design pattern has its origin in the fact that developers have noticed recurring and similar design problems. Therefore, it became necessary to conceptualize design problems in such a way that the same answers were reused each time the problem arose.

To conceptualize, it is often essential to use code directly. But in the case of the design pattern, developers translate the conceptualization into an abstract idea of how to solve the problem. Each developer then implements the problem resolution using the language they are familiar with.

Why use a design pattern?

The usefulness of using a design pattern is obvious. The design pattern can accelerate the development process. It provides proven development paradigms, which helps save time without having to reinvent patterns every time a problem arises.

Because the design pattern is created to fix known problems, they can be predicted before they become visible during the implementation process. Again, the design pattern speeds up the development process. Standardization related to the design pattern is also very useful to facilitate code readability.

In short, the design pattern is useful when moving from an analysis model to a development model. It can be used in a concrete way in several cases, in particular for managing the payroll system when changing a salary and keeping the system informed of the changes this implies.

Thanks to the design pattern, it is possible to document the solutions to be adopted based on previous practices and lessons learned. Several software components are used in the implementation of the design pattern. The model therefore speeds up a process that involves several components. Developers use the language they are familiar with in the application of each solution.

There are about 26 Patterns currently discovered:

These 26 can be classified into 3 types:

1. Creational: These patterns are designed for class instantiation. They can be either class-creation patterns or object-creational patterns.

2. Structural: These patterns are designed with regard to a class's structure and composition. The main goal of most of these patterns is to increase the functionality of the class(es) involved, without changing much of its composition.

3. Behavioral: These patterns are designed depending on how one class communicates with others.

## Differences between Architecture and Design Patterns

| **Architecture Pattern** | **Design Pattern** |
|---|---|
| 1. Architecture is the overall structure of software. | 1. Design patterns are concerned with how the components are built.<br>2. Developer chooses different design pattern according to the architecture specification and requirement. |

| | |
|---|---|
| | 3. It's about a particular solution. |
| Architecture comes in the Designing phase. | Design Patterns come in the Building phase. |
| Architectural patterns are like a blueprint. | Design pattern is the actual implementation. |
| Architecture is the base which everything else adheres to. | Design pattern is a way to structure classes to solve common problems. |
| **Architecture** : how components should behave and communicate in the system, set the physical location of components and finally choose the tools in order to create components. | **Design** : while architecture deals more with the wide picture, design should drill down into the details relating to implementing certain components. Designing of components ends up with classes, interfaces, abstract classes and other OO features in order to fulfill the given component tasks. |

## computer architecture

### System design

System design includes all hardware parts of a computer, including data processors, multiprocessors, memory controllers, and direct memory access. It also includes the graphics processing unit (GPU). This part is the physical computer system.

## Instruction set architecture (ISA)

This includes the functions and capabilities of the central processing unit (CPU). It is the embedded programming language and defines what programming it can perform or process. This part is the software that makes the computer run, such as operating systems like Windows on a PC or iOS on an Apple iPhone, and includes data formats and the programmed instruction set.

## Microarchitecture

Microarchitecture is also known as a computer organization and defines the data processing and storage element and how they should be implemented into the ISA. It is the hardware implementation of how an ISA is implemented in a particular processor.

## **RISC vs. CISC Architectures: Which one is better?**

The short answer is that RISC is perceived by many as an improvement over CISC. There is no best architecture since different architectures can simply be better in some scenarios but less ideal in others.  RISC-based machines execute one instruction per clock cycle. CISC machines can have special instructions as well as instructions that take more than one cycle to execute. This means that the same instruction executed on a CISC architecture might take several instructions to execute on a RISC machine. The RISC architecture will need more working (RAM) memory than CISC to hold values as it loads each instruction, acts upon it, then loads the next one.

The CISC architecture can execute one, albeit more complex instruction, that does the same operations, all at once, directly upon memory. Thus, RISC architecture requires more RAM but always executes one instruction per clock cycle for predictable processing, which is good for pipelining. One of the major differences between RISC and CISC is that RISC emphasizes efficiency in cycles per instruction and CISC emphasizes efficiency in instructions per program. A fast processor is dependent upon how much time it takes to execute each clock cycle, how many cycles it takes to execute instructions, and the number of instructions there are in each program. RISC has an emphasis on larger program code sizes (due to a smaller instruction set, so multiple steps done in succession may equate to one step in CISC).

The RISC ISA emphasizes software over hardware. The RISC instruction set requires one to write more efficient software (e.g., compilers or code) with fewer instructions. CISC ISAs use more transistors in the hardware to implement more instructions and more complex instructions as well.

RISC needs more RAM, whereas CISC has an emphasis on smaller code size and uses less RAM overall than RISC. Many microprocessors today hold a mix of RISC- and CISC-like attributes, however, such as a CISC-like ISA that treats instructions as if they are a string of RISC-type instructions.