Task 5 Instant course

# **Clean Code 13 Principles:**

# 1. Don’t Repeat Yourself (DRY)

This principle suggests that code should not have unnecessary duplication. Instead, it should be organized in a way that avoids redundancy and makes it easy to maintain. For example, instead of writing the same calculation in multiple places in the code, create a function that performs the calculation and call that function from the different places where the calculation is needed.

// bad example  
let total = 0;  
for (let i = 0; i < prices.length; i++) {  
 total += prices[i];  
}  
console.log(total);  
for (let i = 0; i < prices.length; i++) {  
 total += prices[i];  
}  
console.log(total);  
// good example  
function calculateTotal(prices) {  
 let total = 0;  
 for (let i = 0; i < prices.length; i++) {  
 total += prices[i];  
 }  
 return total;  
}  
console.log(calculateTotal(prices));  
console.log(calculateTotal(prices));

# 2. Write Everything Twice (WET)

This is an opposite principle of DRY. It suggest that if you find yourself copy-pasting code multiple times, anticipating the identical code forking in different directions later on, having WET code may make that future change easier.

class Dog {  
 name = "Dog";  
 move() {  
 console.log("Dog is moving");  
 // future implementation  
 // console.log("Dog is trotting");  
 }  
}  
  
class Cat {  
 name = "Cat";  
 move() {  
 console.log("Cat is moving");  
 // future implementation  
 // console.log("Cat is sneaking");  
 }  
}  
// with DRY, we would have a Animal with move method  
// however, dog and cat could be moving in different ways,   
// so we want to keep them seperate following WET

# 3. Single Responsibility Principle (SRP)

Each module or function should have only one reason to change. For example, instead of having a function that handles multiple tasks, split it up into multiple functions, each with a single responsibility.

// bad example  
function processData(data) {  
 // validate data  
 // save data to database  
 // send data to another system  
 // log data  
}  
  
// good example  
function validateData(data) {  
 //…

# 2- What types of problems Design pattern solves?

Design patterns are reusable solutions to common problems that software developers encounter while designing and implementing software systems. They provide a structured way to solve specific design problems and help create more maintainable, flexible, and efficient software. Here are some types of problems that design patterns can help solve:

1. **Creational Patterns:**
   * **Problem**: How to create objects in a way that's flexible, extensible, and follows best practices.
   * **Solution**: Creational patterns provide mechanisms for object creation, such as abstracting the process, controlling instantiation, and ensuring single instance creation.
   * Examples: Singleton, Factory Method, Abstract Factory, Builder, Prototype.
2. **Structural Patterns:**
   * **Problem**: How to define the relationships between classes and objects to form larger structures.
   * **Solution**: Structural patterns help compose classes and objects to create flexible and efficient structures.
   * Examples: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
3. **Behavioral Patterns:**
   * **Problem**: How to define the communication and interactions between objects to achieve specific behaviors.
   * **Solution**: Behavioral patterns address communication and collaboration between objects, promoting flexibility and reusability.
   * Examples: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.
4. **Architectural Patterns:**
   * **Problem**: How to structure the entire software system to ensure modularity, maintainability, and scalability.
   * **Solution**: Architectural patterns provide high-level structures for organizing the components of a system.
   * Examples: Model-View-Controller (MVC), Model-View-ViewModel (MVVM), Layered Architecture, Microservices, Event-Driven Architecture.
5. **Concurrency Patterns:**
   * **Problem**: How to handle concurrency, synchronization, and parallelism in a multi-threaded or distributed system.
   * **Solution**: Concurrency patterns address issues related to managing and coordinating concurrent execution.
   * Examples: Thread Pool, Producer-Consumer, Reader-Writer Lock, Monitor Object, Active Object.
6. **Presentation Patterns:**
   * **Problem**: How to handle user interfaces, presentation logic,

Top of Form

# 3- when to use solid principle?

SOLID is a popular set of design principles that are used in object-oriented software development. SOLID is an acronym that stands for five key design principles: single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle. All five are commonly used by software engineers and provide some important benefits for developers.

4- Design pattern vs Architecture pattern

Architecture deals with abstract elements like strategy, purpose, and structures. Design on the other hand inclines more toward concrete implementation. Similarly, architecture curates schematic representation for the construction of structures. A designer rather focuses on enriching the overall experience of the structures and creating them to be more lively.

Only when both of these niches merge into one and the professionals handling these work in alignment with each other. The overall value of the construction will scale up this way. A flair for design, an eye for details, and an artistic understanding are what will help bring any project to life.

There is also a significant difference between architects and designers compared to that civil engineers. While the former deals with the visualization of the construction, the latter deals with the design-to-completion procedures. They bring the design to a concrete standing.

#### Some of the examples of architecture include:

* Building architecture
* Enterprise architecture
* Data architecture
* Business architecture
* Animation architecture
* Technical architecture
* Brand architecture

#### Some of the examples of design include:

* Interior design
* Visual design
* Technology design
* User Interface design
* Web design
* Experience design
* Product design

5- Two basic computer architecture Which one is better

**John von Neumann** coined and developed this architecture. The computer we are using nowadays is based on the von Neumann architecture. It has some concepts. It is also known as Princeton architecture. It renders a unique design for the electronic digital systems having the following components:

* A Central Processing Unit (CPU) with arithmetic and logic unit (ALU) and processors with attached registers.
* A memory that can store data and instructions.
* External mass storage or secondary storage.
* A Control Unit (CU) with the ability to hold instructions in the program counter (PC) or instruction register (IR).
* Input and output mechanisms and peripherals.

The von Neumann design thus constitutes the foundation of modern computing. The Harvard architecture, a similar model, had committed data addresses and buses for reading and writing to memory. It wins because von Neumann's architecture was easier to execute in real hardware.

### Harvard Architecture

Harvard Architecture consists of code and data laid in distinct memory sections. It requires a separate memory block for data and instruction. It has solely contained data storage within the Central Processing Unit (CPU). A single collection of clock cycles is needed. Data accessibility in one memory is done by a single memory location in the case of Harvard architecture.

6 - RISC vs CISC

|  |  |
| --- | --- |
| **RISC** | **CISC** |
| Emphasis on software | Emphasis on hardware |
| Small number of fixed length instructions | Large number of instructions |
| Simple, standardised instructions | Complex, variable-length instructions |
| Single clock cycle instructions | Instructions can take several clock cycles |
| Heavy use of RAM | More efficient use of RAM |
| Low cycles per second with large code sizes | Small code sizes with high cycles per second |

# 7- ARM vs AVR

**AVR** is an abbreviation for **Alf and Vegard’s RISC processor, also Advanced Virtual RISC.** It is named in the honor of its developers, Alf-Egil Bogen and Vegard Wollan. AVR is a RISC (Reduced Instruction Set Computer) based microcontroller architecture. It was first produced by Atmel Corporation in the year of 1997.

The AT90S8515 was the first microcontroller developed based on the AVR microcontroller architecture. AVR microcontrollers have simple instruction sets, making them fast and efficient. The major advantages of AVR microcontrollers include low power consumption, low cost, and high performance. We can use assemble language as well as high-level languages like C, C++, etc. to program these microcontrollers for a specific function.

AVR microcontrollers are widely used in several different applications like robotics, home and office appliances, industrial automation systems, automobiles, etc.

**ARM** is the abbreviation for **Advanced RISC Machine. ARM microcontroller** is a 32-bit architecture microcontroller that was developed by Acorn Computers in 1983.

ARM is basically a family of Reduced Instruction Set Computing (RISC) architecture-based microprocessors. ARM microcontrollers consist of ARM processors, RAM, ROM, and I/O peripherals. ARM microcontrollers are used in a wide range of applications due to their low power consumption, low cost, and high performance.

One of the important features of ARM microcontrollers is that they are highly customizable depending on requirements of the applications. Therefore, it is highly versatile microcontroller architecture.

We can use assembly language as well as high level programming languages such as C, C++ to program the ARM microcontrollers. ARM microcontrollers are highly scalable; hence they can be used in several applications, from simple embedded systems to high-end computing systems.

After getting an overview of AVR and ARM individually, let us now discuss their important differences.

# 8- Difference between scheduling algorithms and when to use each of them

Scheduling algorithms are essential components in operating systems that manage the allocation of resources, such as CPU time, among multiple processes or tasks. The choice of scheduling algorithm can significantly impact system performance, fairness, and responsiveness. Different algorithms are designed to achieve various goals, so the choice of which one to use depends on the specific requirements and characteristics of the system and its workload. Here's an overview of some common scheduling algorithms and when to use each of them:

1. First-Come, First-Served (FCFS) Scheduling:
   * Description: Processes are executed in the order they arrive. The first process to arrive is the first to be executed.
   * Use When: Suitable for batch processing or situations where fairness is important. Not suitable for interactive systems as it may lead to poor response times.
2. Shortest Job Next (SJN) Scheduling:
   * Description: The process with the smallest execution time is selected next.
   * Use When: It's difficult to predict job lengths, but you want to minimize average waiting time. May cause starvation for longer jobs.
3. Round Robin (RR) Scheduling:
   * Description: Each process is assigned a fixed time slice (quantum) to run. If it doesn't complete in the time slice, it's moved to the back of the queue.
   * Use When: Suitable for interactive systems, prevents any process from monopolizing the CPU for too long. Fairness is a concern.
4. Priority Scheduling:
   * Description: Each process is assigned a priority, and the process with the highest priority is executed first.
   * Use When: When certain processes are more important or time-sensitive than others. Care must be taken to avoid starvation of lower-priority processes.
5. Multilevel Queue Scheduling:
   * Description: Processes are divided into multiple queues with different priorities, and each queue can use a different scheduling algorithm.
   * Use When: Suitable for systems with a mix of CPU-bound and I/O-bound tasks or varying levels of importance.
6. Multilevel Feedback Queue Scheduling:
   * Description: Similar to multilevel queue, but processes can move between queues based on their behavior. A process that uses too much CPU time can be moved to a lower-priority queue.
   * Use When: Useful for systems with varying workloads and where you want to adapt to changing behavior.
7. Lottery Scheduling:
   * Description: Each process is assigned a number of lottery tickets. A random lottery ticket is drawn, and the process with that ticket is selected for execution.
   * Use When: Suitable when you want to allocate resources in a probabilistic manner, providing a sense of fairness and randomness.
8. Guaranteed Scheduling (Rate Monotonic Scheduling, Earliest Deadline First, etc.):
   * Description: Real-time scheduling algorithms that ensure tasks meet their deadlines by assigning priorities based on deadlines or periodicity.
   * Use When: Critical for real-time systems where meeting deadlines is crucial, such as in industrial control systems or embedded applications.

The choice of scheduling algorithm depends on factors such as the nature of the tasks, system goals (throughput, response time, fairness), and real-time requirements. It's important to analyze the specific characteristics of the system and workload to make an informed decision.

Top of Form

# 9-What is Fragmentation when it occurs?

# Fragmentation refers to the phenomenon where files or data become scattered or divided into various pieces, or "fragments," across a storage medium, such as a hard disk drive or solid-state drive. This can occur over time as files are created, modified, and deleted on the storage medium.

# There are two main types of fragmentation:

# **File Fragmentation:** This occurs when a single file is divided into multiple fragments that are stored in non-contiguous sectors on the storage medium. This happens because the operating system may allocate space for a file in chunks as the file grows or is modified. As a result, reading or writing such fragmented files can lead to slower performance, as the drive's read/write head has to move around to different locations on the disk to access all the fragments of the file.

# **Free Space Fragmentation:** This type of fragmentation happens when free space on a storage medium is scattered into smaller non-contiguous chunks. When files are created or extended, they may be allocated to available free space, and if the free space is not contiguous, it can lead to inefficient use of the available storage capacity. This can contribute to slower write speeds and reduced overall storage performance.

# Fragmentation can impact the efficiency and performance of a storage device. When fragmentation becomes severe, it can lead to increased seek times, longer file access times, and reduced data transfer rates. To address fragmentation-related performance issues, operating systems often include tools like defragmentation or optimization utilities. These utilities can help rearrange and consolidate fragmented files and free space, improving overall storage performance.

# It's worth noting that with the rise of solid-state drives (SSDs), fragmentation has become less of a concern due to the nature of how data is accessed on SSDs, which doesn't involve mechanical read/write heads like traditional hard disk drives. However, some level of file system fragmentation can still occur on SSDs and can impact performance, albeit to a lesser extent than on traditional HDDs.

# 10-Semi structured database:

# A semi-structured database is a type of database that does not require a rigid, predefined schema like traditional relational databases. In a semi-structured database, data can be stored in a flexible format that allows for variations in structure and data types. This makes it suitable for handling data that doesn't fit neatly into rows and columns, as is the case with structured databases.

# The key features of a semi-structured database include:

# Flexible Schema: Unlike traditional relational databases, where data must adhere to a fixed schema, a semi-structured database allows data to be stored with varying structures. Each piece of data can have different attributes, and new attributes can be added without affecting the existing data.

# Hierarchical Structure: Semi-structured databases often use hierarchical structures like XML (eXtensible Markup Language) or JSON (JavaScript Object Notation) to represent data. These formats allow for nested and complex data structures.

# NoSQL Databases: Many semi-structured databases fall under the category of NoSQL (Not Only SQL) databases. NoSQL databases are designed to handle large amounts of unstructured or semi-structured data efficiently.

# Schema-on-Read: In contrast to traditional relational databases where data is validated and structured at the time of insertion (schema-on-write), semi-structured databases validate and structure data at the time of retrieval (schema-on-read). This approach allows for more flexible querying and analysis.

# Agile Development: Semi-structured databases are well-suited for scenarios where the data evolves rapidly or where the exact structure of the data is not known in advance. This is common in applications like social media, content management systems, e-commerce, and IoT (Internet of Things) devices.

# Examples of semi-structured databases include:

# MongoDB: A popular NoSQL database that uses a flexible document-based model (BSON) to store data. It allows for nested and dynamic schemas.

# Cassandra: A distributed NoSQL database that can handle large amounts of semi-structured data. It uses a wide-column store model.

# Couchbase: Another NoSQL database that can store JSON documents and provides features like automatic sharding and replication.

# Amazon DynamoDB: A managed NoSQL database service provided by Amazon Web Services (AWS) that offers flexible data models and automatic scaling.

# Semi-structured databases are particularly useful when dealing with data that doesn't fit neatly into tables and rows, when the schema might change frequently, or when agility and scalability are important factors.