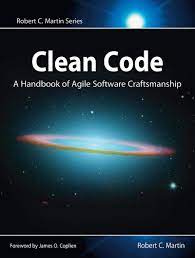
## 5.1

## The concept of clean code can be traced back to Robert Cecil Martin and his book ”Clean Code: A Handbook of Agile Software Craftsmanship”



Code is clean if it can be understood easily – by everyone on the team. Clean code can be read and enhanced by a developer other than its original author. With understandability comes readability, changeability, extensibility and maintainability

## We try to prevent code from smells so we prevent :

1. Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
2. Fragility. The software breaks in many places due to a single change.
3. Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
4. Needless Complexity.
5. Needless Repetition.
6. Opacity. The code is hard to understand.

# **13 Clean Code Principles :**

DRY, WET, SRP, OCP, LSP, ISP, DIP, KISS, YAGNI, Fail Fast, LoD, CQS, Composition over Inheritance

# 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.

# 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.

# 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.

# 4. Open/closed Principle (OCP)

A module or function should be open for extension but closed for modification. For example, instead of modifying an existing class to add new functionality, create a new class that extends the original class and add the new functionality there.

# 5. Liskov Substitution Principle (LSP)

Objects of a superclass should be able to be replaced with objects of a subclass without altering the correctness of the program. For example, a subclass should be able to replace its parent class without breaking the program

# 6. Interface Segregation Principle (ISP)

A client should not be forced to implement interfaces it doesn’t use. For example, instead of having a monolithic interface with many methods, split it up into smaller, more specific interfaces.

# 7. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions. For example, instead of having a high-level module depend on a specific implementation of a low-level module, have it depend on an abstraction of the low-level module.

# 8. Keep It Simple, Stupid (KISS)

This principle suggests that code should be as simple as possible, and should avoid unnecessary complexity. For example, instead of using a complex algorithm to solve a problem, use a simpler one that gets the job done.

# 9. You Aren’t Gonna Need It (YAGNI)

This principle suggests that code should not be written until it is actually needed, as it can add unnecessary complexity and make the code harder to maintain. For example, instead of adding a feature that may be needed in the future, focus on the features that are needed now

# 10. Fail Fast

This principle suggests that code should fail as early as possible, so that issues can be identified and resolved quickly. For example, instead of waiting until the end of a function to check for errors, check for errors as soon as possible.

# 11. Law of Demeter (LoD)

This principle suggests that an object should only communicate with its immediate neighbors and should not reach into the internal state of other objects. For example, instead of accessing the internal state of an object, use a method to get the information you need.

# 12. Command Query Separation (CQS)

It is a principle that suggests that methods should either be command methods that change the state of an object, or query methods that return information about an object, but not both. For example, instead of having a method that both changes the state of an object and returns a value, have separate methods for changing the state and returning the value.

# 13. Composition over Inheritance

It suggests that code should favor composition over inheritance, as composition allows for greater flexibility and easier maintenance. For example, instead of inheriting properties and methods from a parent class, compose objects with the properties and methods they need

## 5.2

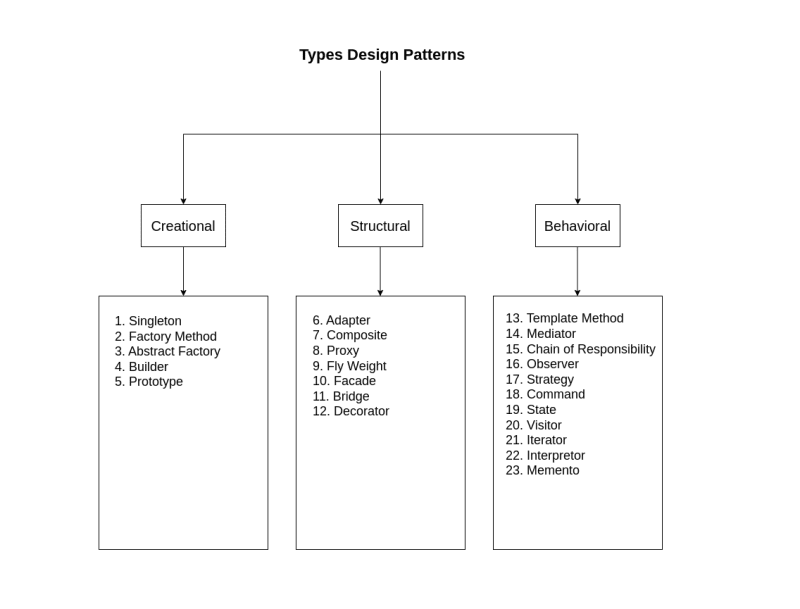
## Design patterns are a systematic approach for addressing recurring problems that programmers face. They describe a universal standard to solve those problems so that we don’t have to rediscover the solution from scratch.

## Elements of a Design Pattern

1. **Name** - is a handle used to describe the design issue. It is necessary because it gives a common ground of understanding among programmers.
2. **Problem** - is used to describe when the pattern can be used.
3. **Solution** - is a template that describes elements and their relationships without providing implementation detail.
4. **Consequence** - is used to describe the trade-offs of the pattern. Examples: time and space trade-offs, flexibility, extensibility, etc.

## Types of design patterns :

1. **Creational** - They are concerned with object-creation techniques.
2. **Structural** - They are concerned with the combination of objects and classes in a flexible and effective way.
3. **Behavioral** - They are concerned with the interaction between classes and objects and their responsibility



## The Catalog of Design Patterns or Gang of Four (GOF)

This website illustrate desighn patterns Extensively, with visual images and examples : <https://refactoring.guru/design-patterns/catalog>

1. **Abstract Factory** provides an interface for creating families of related or dependent objects without specifying their concrete classes.
2. **Adapte**r converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
3. **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
4. **Builder** separates the construction of a complex object from its representation so that the same construction process can create different representations.
5. **Chain of Responsibility** avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
6. **Command** encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
7. **Composite** composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
8. **Decorator** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
9. **Facade** provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
10. **Factory Method** defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
11. **Flyweight** uses sharing to support large numbers of fine-grained objects efficiently.
12. **Interpreter**,given a language, defines a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

1. **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
2. **Mediator** defines an object that encapsulates how a set of objects interact.
3. **Memento**, without violating encapsulation, captures and externalizes an object's internal state so that the object can be restored to this state later.
4. **Observer** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
5. **Prototype** specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
6. **Proxy** provides a surrogate or placeholder for another object to control access to it.
7. **Singleton** ensures a class only has one instance, and provide a global point of access to it.
8. **State** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
9. **Strategy** defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
10. **Template Method** defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
11. **Visitor** represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**What types of problems Design pattern solves :**

## Not knowing what object implementations you’ll need ahead of time

* **Factory Method** – if an object needs to be instantiated in one go
* **Abstract Factory** – if multiple object needs to be instantiated in one go
* **Builder** – if an object needs to be built step-by-step

## Making several exact copies of a complex object

* **Prototype**

## Using many instances of an object while keeping code running smoothly

* **Object pool** – facilitates the reuse of pre-instantiated objects
* **Flyweight** – allows you to have a very large number of similar objects without much performance penalty, but makes the code complicated
* **Prototype** – can be combined with Object Pool to make initial creation of the objects easier

## Using the same single instance of an object throughout the application

* **Singleton**

## Third party components aren’t directly compatible with your code

* **Adapter**

## Adding new functionality to existing objects that cannot be modified

* **Decorator**

## Accessing complex back-end logic from the presentation layer

* **Facade** – simplifies the access interface to the complex logic
* **Proxy** – caches the result of expensive operations

## User interface and business logic are developed separately

* **Bridge** – suitable when front-end and back-end can be designed together up-front\*
* **Facade** – suitable when back-end is hosted by a third party or cannot be designed alongside the user interface up-front
* **Proxy** – prevents services outage during back-end redeployment

## Building a complex object hierarchy

* **Composite**

## Implementing complex conditional logic

* **Strategy** – facilitates a conditional one-off action
* **Factory Method** – facilitates a conditional creation of a long-lived object
* **Abstract Factory** – facilitates a conditional creation of multiple long-lived objects

## Multiple object instances of different types need to be able to communicate with each other

* **Mediator** – easier to implement when communication logic between different objects doesn’t expect to be changed
* **Observer** – easier to implement when communication between different objects is expected to change at runtime or during configuration

## Multiple stages of processing are needed

* **Chain of Responsibility** – suitable in scenarios where processing steps are pre-defined and a one-off logical flow is executed
* **Builder** – suitable in scenarios where the order of processing stages can be arbitrary and a reusable object is being built

## The system is controlled by complex combinations of inputs

* **Command**

## Ability to undo an action that has been applied

* **Memento** – allows you to store the exact snapshots of the state
* **Command** – allows you to revert by performing an opposite action

## Ability to traverse a collection without knowing its underlying structure

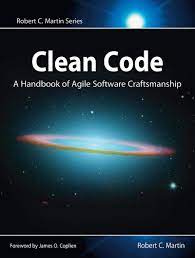
* **Iterator**

## Creating a family of related algorithms

* **Template Method** – easy to implement, but might violate Liskov substitution principle
* **Visitor** – allows you to separate an object from its behavior and add many differential types of behavior to the objects
* **State** – allows you to change the behavior of an entire object in one go by changing the mode (state) that the object is in
* **Strategy** – suitable when algorithms are selected by conditional logic

## 5.3

## The concept of clean code can be traced back to Robert Cecil Martin and his book ”Clean Code: A Handbook of Agile Software Craftsmanship”



When the developer builds software following a bad design, the code can become inflexible and more brittle. Small changes in the software can result in bugs. For these reasons, we should follow SOLID Principles.

A book by **Robert Cecil Martin known as uncle Bob**

**SOLID principles :**

* Single Responsibility Principle:  
   each of our classes has to be only used for one purpose so , we don’t have to change code as often when something changes. And make comprehending the purpose of the code easier
* Open / Closed Principles:  
  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 the class has the base members that child classes extend from
* Interface Segregation Principle ;  
  clients shouldn’t be forced to depend on interfaces that they don’t use ,t his 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.  
  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.

**When to use SOLID principles :**

**1 for Single Responsibility Principle:**   
Let’s take a look at a proper example. You’ll always be tempted to put similar classes together – but unfortunately, this goes against the Single-responsibility principle as The ValidatePerson class does two jobs – it validates the person’s name and age and then displays some information.

The way to avoid this problem is to separate code that supports different actions and jobs so that each class only performs one job and has one reason to change. As The Display() method goes against the SRP because the goal is that a class should have only one job and do one thing.

The ValidatePerson class does two jobs – it validates the person’s name and age and then displays some information. Then it will enhanced to do one job as a DisplayPerson class:







**2 for Open / Closed Principles:** it is used when need for **class, module or function extension and developing** as entity should be extendable without modifying the entity itself. )modifying a code without editing it )

For example : the array of iceCreamFlavours, which contains a list of possible flavors , this means for you to add new flavors, you would need to directly edit the iceCreamFlavors array.  
o fix this, you would need an extra class or entity to handle addition, so you no longer need to modify the code directly to make any extension.

.





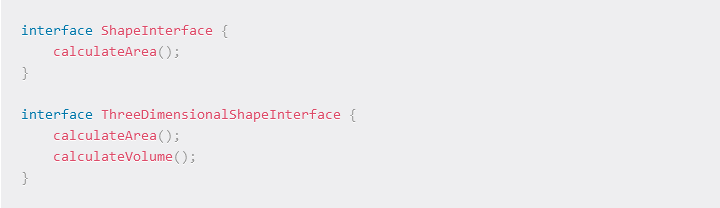
## 3 for Liskov Substitution Principle: The principle defines that in an inheritance, objects of a superclass (or parent class) should be substitutable with objects of its subclasses (or child class) without breaking the application or causing any error. then we use it among inherited classes and make sure to satisfy it as shown





## This way, you can set the color and get the color using either the super or subclasses (on another words parent classes contain only the common attributes and methods among all Childs classes

4 for Interface Segregation Principle: it is used when dealing with interfaces **as this principle is about keeping things separated**When any class implements this interface, all the methods must be defined even if you won't use them or if they don’t apply to that class even of class does not use all interface methods .as shown the interface ShapeInterface that calculate area and volume is called by regtangle that has no volume





You can now implement the specific interface that works with each class:

## 5 for Dependency Inversion Principle : This principle is targeted towards loosely coupling software modules as classes should depend upon interfaces or abstract classes instead of concrete classes and functions. This makes your classes open to extension, following the open-closed principle.

for example. When building a store, you would want your store to make use of a payment gateway like stripe or any other preferred payment method. You might write your code tightly coupled to that API without thinking of the future.

But then what if you discover another payment gateway that offers far better service, let’s say PayPal? Then it becomes a struggle to switch from Stripe to Paypal, which should not be an issue in programming and software design.

## 5.4

## Design pattern vs Architecture pattern

## Architectural pattern is higher level of abstraction of software design and Design pattern provides solution to specific module level problems.

# Architecture

An **architectural pattern** is a general, reusable solution to a commonly occurring problem in **software architecture** within a given context. … The **architectural patterns** address various issues in **software** engineering, such as computer hardware performance limitations, high availability and minimization of a business risk

Architecture is the overall structure of software.

1. Architecture is the structure of the software system in its entirety.
2. Developer chooses different design pattern according to the architecture specification and requirement.
3. It’s define the granularity of the component.

# **Design pattern**:

1. Design patterns are concerned with how the components are built.

2. It’s about particular solution.

# **Differences** :

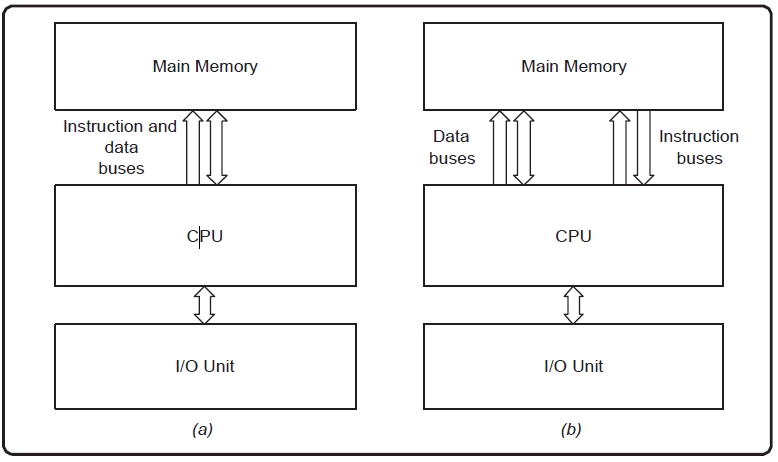
1. Architecture comes in Designing phase and Design Patterns comes in Building phase.
2. Architectural pattern is like a blue print and design pattern is actual implementation.
3. Architecture is base which everything else adhere to and design pattern is a way to structure classes to solve common problems.
4. All **Architecture is design pattern** but all **design pattern can not be architecture.** Like MVC can come under both. But singleton design pattern can not be an architecture pattern. MVC, MVVM all come under both.
5. **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 end up with classes, interfaces, abstract classes and other OO feature in order to fulfil the given component tasks.
6. Architectural pattern focuses more on the abstract view of idea while Design pattern focuses on the implementation view of idea.
7. One Architectural pattern can be implemented by using many design patterns. There is one to many relationship between architectural pattern and design pattern.
8. Architectural pattern is not App Architecture. Architectural pattern provides guidelines and rules to make application more maintainable, loosely coupled and extensible at project/solution level. Design patterns also do at some extend but more at module or component level.

## As a summary we can say that Architectural pattern is higher level of abstraction of software design and Design pattern provides solution to specific module level problems.

## 5.5

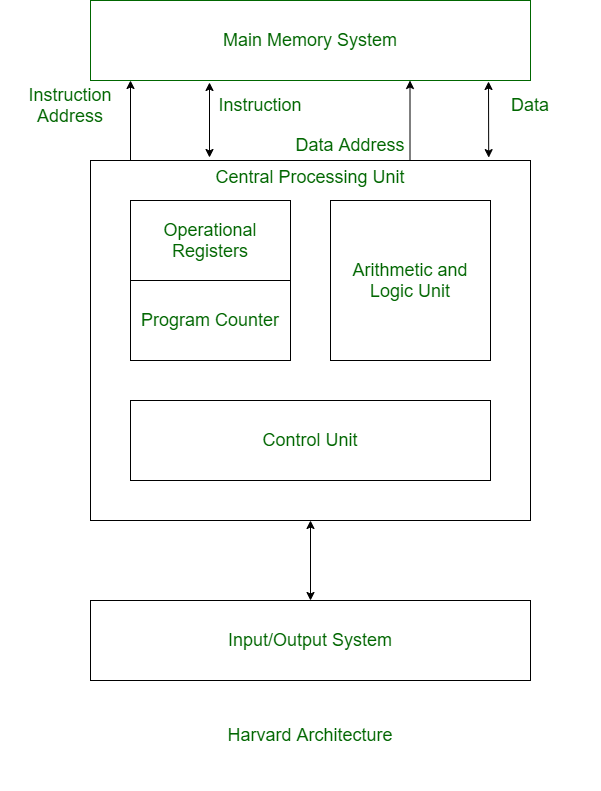
## Computer architecture:

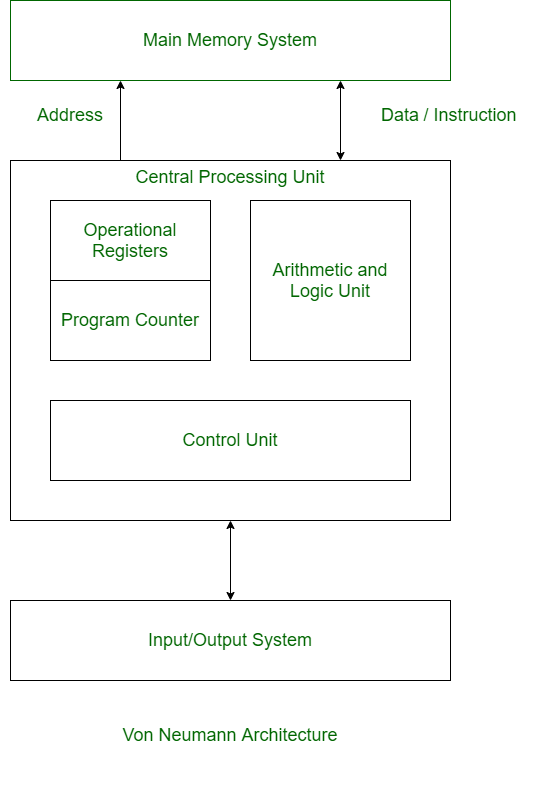
## Computer architecture is defined as the end-to-end structure of a computer system that determines how its components interact with each other in helping execute the machine’s purpose (i.e., processing data). all computers are founded on a set of principles describing how hardware and software connect to make them function.



Von Neumann Architecture (a) and Harvard Architecture (b)

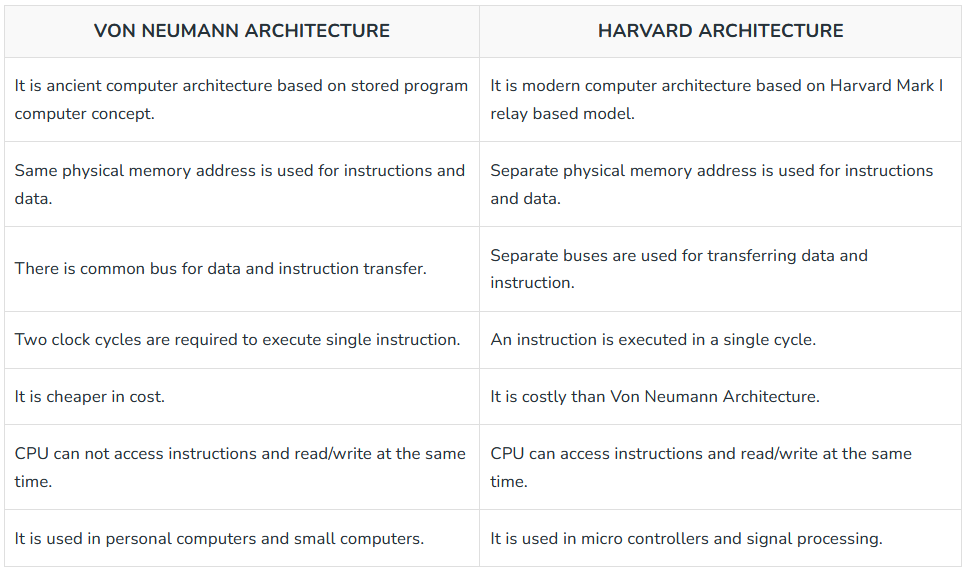
## [Von Neumann Architecture:](https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/)  Von Neumann Architecture is a digital computer architecture whose design is based on the concept of stored program computers where program data and instruction data are stored in the same memory. This architecture was designed by the famous mathematician and physicist John Von Neumann in 1945.





## [Harvard Architecture:](https://www.geeksforgeeks.org/harvard-architecture/)  Harvard Architecture is the digital computer architecture whose design is based on the concept where there are separate storage and separate buses (signal path) for instruction and data. It was basically developed to overcome the bottleneck of Von Neumann Architecture.

**Difference between Von Neumann and Harvard Architecture :**



## 5.6

## RISC vs CISC

## Complex instruction set computer (CISC) and reduced instruction set computer (RISC) are the two predominant approaches to the architecture that influence how computer processors function.

**CISC** processors have one processing unit, auxiliary memory, and a tiny register set containing hundreds of unique commands. These processors execute a task with a single instruction, making a programmer’s work simpler since fewer lines of code are required to complete the operation. This method utilizes less memory but may need more time to execute instructions.

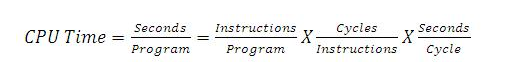
A reassessment led to the creation of high-performance computers based on the **RISC** architecture. The hardware is designed to be as basic and swift as possible, and sophisticated instructions can be executed with simpler ones.

**Reduced Instruction Set Architecture (RISC) –**   
The main idea behind this is to make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, a store command will store the data.

**Complex Instruction Set Architecture (CISC) –**   
The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it’s complex.

Both approaches try to increase the CPU performance

* **RISC:** Reduce the cycles per instruction at the cost of the number of instructions per program.
* **CISC:** The CISC approach attempts to minimize the number of instructions per program but at the cost of an increase in the number of cycles per instruction.



**Characteristic of RISC –**

1. Simpler instruction, hence simple instruction decoding.
2. Instruction comes undersize of one word.
3. Instruction takes a single clock cycle to get executed.
4. More general-purpose registers.
5. Simple Addressing Modes.
6. Fewer Data types.
7. A pipeline can be achieved.

**Characteristic of CISC –**

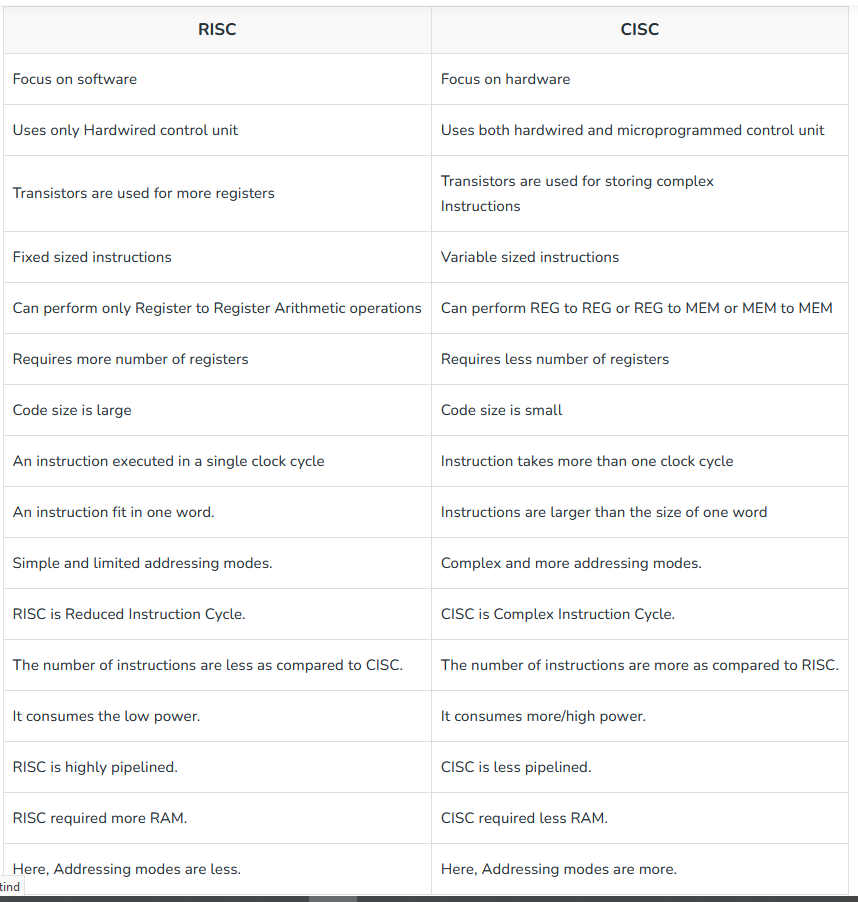
1. Complex instruction, hence complex instruction decoding.
2. Instructions are larger than one-word size.
3. Instruction may take more than a single clock cycle to get executed.
4. Less number of general-purpose registers as operations get performed in memory itself.
5. Complex Addressing Modes.
6. More Data types.

**Example –** Suppose we have to add two 8-bit numbers:

* **CISC approach:** There will be a single command or instruction for this like ADD which will perform the task.
* **RISC approach:** Here programmer will write the first load command to load data in registers then it will use a suitable operator and then it will store the result in the desired location.

So, add operation is divided into parts i.e. load, operate, store due to which RISC programs are longer and require more memory to get stored but require fewer transistors due to less complex command.

**Difference :**



## 5.8

A scheduling algorithm is used to estimate the CPU time required to allocate to the processes and threads. The prime goal of any [CPU scheduling algorithm](https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/) is to keep the CPU as busy as possible for improving CPU utilization.

## Scheduling Algorithms

**1.** [**First Come First Serve(FCFS)**](https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/): As the name implies that the jobs are executed on a first come first serve basis. It is a simple algorithm based on FIFO that’s first in first out. The process that comes first in the ready queue can access the CPU first. If the arrival time is less, then the process will get the CPU soon. It can suffer from the convoy effect if the burst time of the first process is the longest among all the jobs.

**2.** [**Shortest Job First (SJF)**](https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/): Also known as the shortest job first or shortest job next is a non-preemptive type algorithm that is easy to implement in batch systems and is best in minimizing the waiting time. It follows the strategies where the process that is having the lowest execution time is chosen for the next execution.

**3.** [**Longest Job First Scheduling(LJF)**](https://www.geeksforgeeks.org/longest-job-first-ljf-cpu-scheduling-algorithm/): The longest job first (LJF) is the non-preemptive version. This algorithm is also based upon the burst time of the processes. The processes are put into the ready queue according to their burst times and the process with the largest burst time is processed first.

**4.** [**Longest Remaining Time First Scheduling (LRTF)**](https://www.geeksforgeeks.org/longest-remaining-time-first-lrtf-cpu-scheduling-algorithm/): The preemptive version of LJF is LRTF. Here the process with the maximum remaining CPU time will be considered first and then processed. And after some time interval, it will check if another process having more Burst Time arrived up to that time or not. If any other process has more remaining burst time, so the running process will get pre-empted by that process.

**5.** [**Shortest Remaining Time First(SRTF)**](https://www.geeksforgeeks.org/introduction-of-shortest-remaining-time-first-srtf-algorithm/): This algorithm is based on SJF and this is the preemptive version of SJF. In this scheduling algorithm, the process with the smallest remaining burst time is executed first and it may be preempted with a new job that has arrived with a shorter execution time.

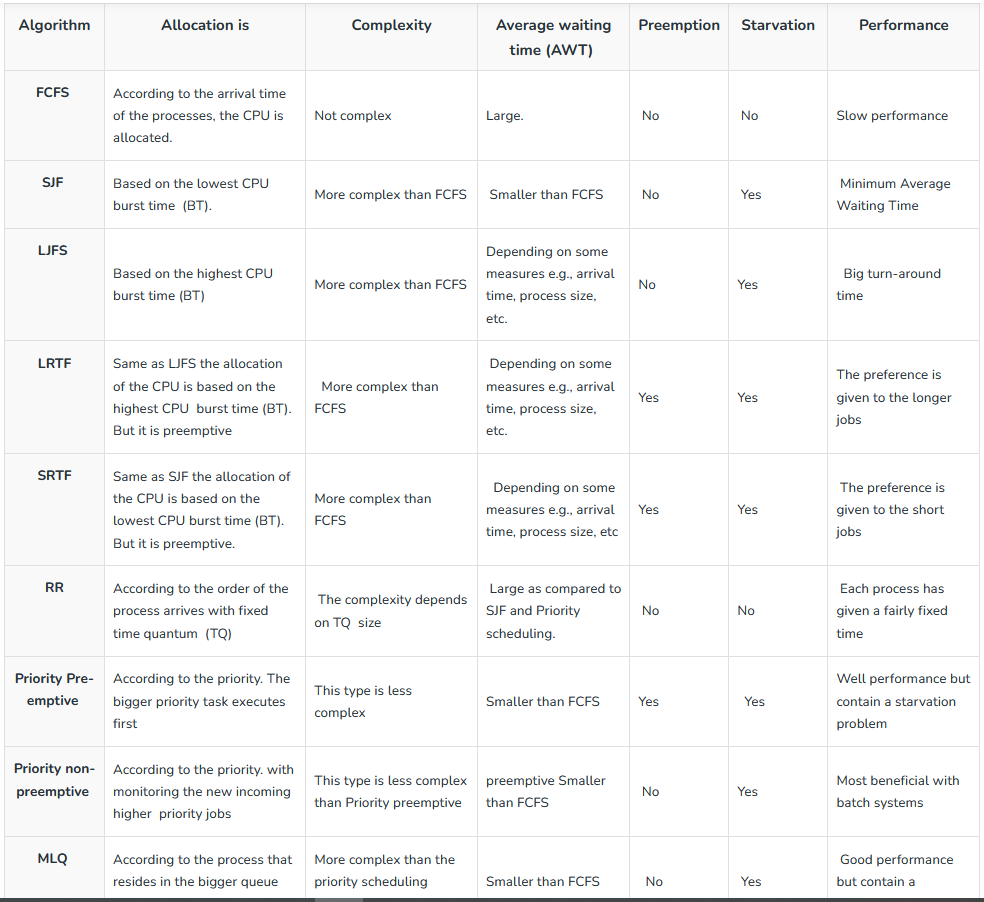
**6.** [**Round Robin(RR)**](https://www.geeksforgeeks.org/program-round-robin-scheduling-set-1/): It is a preemptive scheduling algorithm in which each process is given a fixed time called quantum to execute. At this time one process is allowed to execute for a quantum and then preempts and then another process is executed. In this way, there is context switching between processes to save states of these preempted processes.

**7.** [**Priority Scheduling**](https://www.geeksforgeeks.org/program-for-priority-cpu-scheduling-set-1/): It is a non-preemptive algorithm that works in batch systems and, each process is assigned with a priority and the process with the highest priority is executed first. This can lead to starvation for other processes.

**8.** [**Multiple Levels Queues Scheduling**](https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/): In this scheduling, multiple queues have their scheduling Algorithms and are maintained with the processes that possess the same characteristics. For this, priorities are assigned to each queue for the jobs to be executed.

**9.** [**Multilevel-Feedback-Queue Scheduler**](https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/): It defines several queues and the scheduling algorithms for each queue. This algorithm is used to determine when to upgrade a process when to demote a process, and also determine the queue in which a process will enter and when that process needs service.

***Note:***The SJF scheduling algorithm is hypothetical and un-implementable, as it is impossible to determine the burst time of any process without running it. SJF is a benchmarking algorithm as it provides minimum waiting time than any other scheduling algorithm



### **Usage of Scheduling Algorithms in Different Situations**

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

#### **Situation 1:**

The incoming processes are short and there is no need for the processes to execute in a specific order.

In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

#### **Situation 2:**

The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.

Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.

#### **Situation 3:**

The processes are a mix of user based and kernel based processes.

Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.

For example, the scheduler itself is a kernel based process, it should run first so that it can schedule other processes.

**5.9**

**Fragmentation**

Fragmentation, in the context of a hard disk, is a condition in which the contents of a single file are stored in different locations on the disk rather than in a contiguous space. This results in inefficient use of storage space as well as occasional performance degradation. Users frequently create, modify, delete and save files. Back-end operating systems (OS) continuously store these files on hard drives, which inevitably creates scattered files. When fragmentation occurs, the OS needs to consolidate stored files to enhance processing efficiency.

**Fragmentation happens when** the contents of a single file are stored in a non-contiguous space, occurs three forms:

* Internal fragmentation: Unusable allocated space that reduces performance. Files are stored in clusters, which are mini-allocated hard drive storage areas. Each file is automatically written at the beginning of a cluster, which creates a gap potential between the first and last file bytes, that is, slack space. Internal fragmentation also occurs when additional specific bytes are allocated to each file for alignment.
* External fragmentation: Unusable allocated storage space. Applications divide and allocate available space as drive data are read and written. Leftover space becomes fragmented due to allocating algorithm weakness, which renders available storage unusable. External fragmentation also occurs when a large number of files are created, modified and deleted, that is, deleted files are divided into smaller stored chunks.
* Data fragmentation: Occurs when large memory files are divided into smaller pieces and/or an OS tries to allocate large files to an external fragmented storage. For example, when a user creates new files and performs different operations (e.g., renaming, modifying and deletion), extremely small spaces are expected to hold new data files. However, if these new files require more overhead, the OS must seek storage that is greater than average.

## Causes of Fragmentation

User processes are loaded and unloaded from the main memory, and processes are kept in memory blocks in the main memory. Many spaces remain after process loading and swapping that another process cannot load due to their size. Main memory is available, but its space is insufficient to load another process because of the dynamical allocation of main memory processes.

## Types of Fragmentation

There are mainly two types of fragmentation in the operating system. These are as follows:

**1-Internal Fragmentation :** When a process is allocated to a memory block, and if the process is smaller than the amount of memory requested, a free space is created in the given memory block. Due to this, the free space of the memory block is unused, which causes **internal** fragmentation.

**2-External Fragmentation :**External fragmentation happens when a dynamic memory allocation method allocates some memory but leaves a small amount of memory unusable. The quantity of available memory is substantially reduced if there is too much external fragmentation. There is enough memory space to complete a request, but it is not contiguous. It's known as **external** fragmentation.

**5.10**

# **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](https://www.teradata.com/Glossary/What-is-Structured-Data).

Semi-structured data is becoming increasingly common as organizations collect and process more data from a variety of sources, including social media, IoT devices, and other unstructured sources. While semi-structured data can be more challenging to work with than strictly structured data, it offers greater flexibility and adaptability, making it a valuable tool for data analysis and management.

**Semi-structured data** is data that does not conform to a data model but has some structure. It lacks a fixed or rigid schema. It is the data that does not reside in a rational database but that have some organizational properties that make it easier to analyze. With some processes, we can store them in the relational database.

# **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.

**Characteristics of semi-structured Data:**

* Data does not conform to a data model but has some structure.
* Data can not be stored in the form of rows and columns as in Databases
* Semi-structured data contains tags and elements (Metadata) which is used to group data and describe how the data is stored
* Similar entities are grouped together and organized in a hierarchy
* Entities in the same group may or may not have the same attributes or properties
* Does not contain sufficient metadata which makes automation and management of data difficult
* Size and type of the same attributes in a group may differ
* Due to lack of a well-defined structure, it can not used by computer programs easily

## Refrences :

Clean code :

<https://medium.com/@caodanju/13-clean-code-principles-explained-with-examples-dd5be127ed20>

<https://sayansingha.medium.com/the-s-o-l-i-d-principles-clean-code-9fcca658dab5>

<https://www.ionos.com/digitalguide/websites/web-development/clean-code-principles-advantages-and-examples/>

desighn patterns :

<https://scientificprogrammer.net/2022/04/13/summary-of-the-problems-design-patterns-are-intended-to-solve/>

<http://www.cs.unb.ca/~wdu/cs4015w02/ch1.htm>

<https://refactoring.guru/>

<https://dev.to/documatic/from-problems-to-solutions-understanding-design-patterns-3b7i>

solid principles :

<https://www.freecodecamp.org/news/solid-principles-for-programming-and-software-design/>

<https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29>

<https://sayansingha.medium.com/the-s-o-l-i-d-principles-clean-code-9fcca658dab5>

architecture vs design patterns :

<https://singhdivesh.medium.com/according-to-wikipedia-b1afa6de08c>

<https://www.linkedin.com/pulse/architectural-pattern-vs-design-praveen-kumar-kushwaha>

<https://stackoverflow.com/questions/4243187/whats-the-difference-between-design-patterns-and-architectural-patterns>

computer archticture:

<https://www.spiceworks.com/tech/tech-general/articles/what-is-computer-architecture/>

<https://www.geeksforgeeks.org/difference-between-von-neumann-and-harvard-architecture/>

risk vs cisc

<https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/>

<https://www.spiceworks.com/tech/tech-general/articles/risc-vs-cisc/>

<https://www.geeksforgeeks.org/computer-organization-risc-and-cisc/>

scheduling algorithms.

<https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm>

<https://www.geeksforgeeks.org/comparison-of-different-cpu-scheduling-algorithms-in-os/>

<https://www.studytonight.com/operating-system/comparision-scheduling-algorithms>

fragmentation

<https://www.javatpoint.com/fragmentation-in-operating-system>

<https://www.techopedia.com/definition/14383/fragmentation-hard-disks>

semi structural data:

<https://www.geeksforgeeks.org/what-is-semi-structured-data/>

<https://www.teradata.com/Glossary/What-is-Semi-Structured-Data>