What is clean code ?

Clean code is not a set of strict rules. It is a set of principles for writing code that is easy to understand and modify. In this case, “understandable” means that the code can be immediately understood by any experienced developer..

13 Clean Code 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.

2. Write Everything Twice (WET)

This is an opposite principle of DRY. It suggests 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 (OSP)

As the name suggests, this principle states that software entities should be open for extension, but closed for modification. As a result, when the business requirements change then the entity can be extended, but not modified.

5. Liskov Substitution Principle (LSP)

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.

6. Interface Segregation Principle (ISP)

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.

7. Dependency Inversion Principle (DIP)

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.

8. Keep It Simple Stupid (KISS)

It states that most systems should be kept as simple as possible (but not simpler, as Einstein would have said). Unnecessary complexity should be avoided. The question to ask when you're writing code is "can this be written in a simpler way?"

9. You Aren't Gonna Need It (YAGNI)

A developer should not add functionality unless deemed necessary. YAGNI is part of the Extreme Programming (XP) methodology, which wants to improve software quality and increase responsiveness to customer requirements. YAGNI should be used in conjunction with continuous refactoring, unit testing, and integration.

10. Failing Fast

To fail fast means to have a process of starting work on a project, immediately gathering feedback, and then determining whether to continue working on that task or take a different approach—that is, adapt. If a project is not working, it is best to determine that early on in the process rather than waiting until too much money and time has been spent.

11. The Law of Demeter (LoD)

The Law of Demeter or principle of least knowledge says a module should not know about the internals of the objects it manipulates.
An object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.

The advantage of following the Law of Demeter is that the resulting software tends to be more maintainable and adaptable. Since objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers.

12. Command Query Separation (CQS)

The pattern itself is pretty simple. It all comes down to two principles.
Every method should either be a **command** that performs an action, or a **query** that returns data to the caller, **but never both**.
Methods should return a value only if they create no side effects.

13. Composition over Inheritance

Composition is favored over inheritance by many developers, because inheritance forces you to build a taxonomy of objects early on in a project, making your code inflexible for changes later on.

When to use SOLID principles?

The SOLID design principles help us to **create more maintainable, understandable, and flexible software.**
The goal of the SOLID principles is to reduce dependencies so that we can change one area of software without impacting others. Additionally, they're intended to make designs easier to understand, maintain, and extend. Ultimately, using these design principles makes it easier for software engineers to avoid issues and to build adaptive, effective, and agile software.

While the principles come with many benefits, following the principles generally leads to writing longer and more complex code. This means that it can extend the design process and make development a little more difficult. However, this extra time and effort are well worth it because it makes software so much easier to maintain, test, and extend.

SOLID stands for:

- S – Single Responsibility Principle
- O – Open-Closed Principle
- L – Liskov Substitution Principle
- I – Interface Segregation
- D – Dependency Inversion

Every single one of them is explained above.