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.

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.

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.