Clean code

***Principles of clean code :***

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.

***Another principles :***

1. **Readability**: Clean code is easy to read, which means that anyone - including your future self - can understand it quickly. This reduces the time required to grasp the code's functionality, leading to faster development and debugging.
2. **Maintainability**: Code is read more often than it is written. When you write clean code, it becomes easier to maintain and extend the application over time. This is crucial in the software development lifecycle, where projects often evolve and grow.
3. **Collaboration**: Clean code encourages collaboration. When your code is clean and well-organized, other team members can work on it effectively. This makes it easier to divide tasks and work on different parts of the codebase simultaneously.
4. **Bug Reduction**: Clean code reduces the likelihood of introducing bugs. Code that is difficult to understand is more prone to errors during modifications or enhancements.
5. **Efficiency**: Clean code is efficient code. It typically runs faster and uses fewer resources because it avoids unnecessary operations and complexity.
6. **Meaningful Variable and Function Names**: Use descriptive names for variables, functions, classes, and other identifiers. A well-chosen name can convey the purpose of the entity, making the code more understandable. Avoid single-letter variable names or cryptic abbreviations.
7. **Comments and Documentation**: Use comments sparingly, and when you do, make them meaningful. Code should be self-explanatory whenever possible. Documentation, such as inline comments and README files, helps other developers understand your code's purpose and usage. Document complex algorithms, non-trivial decisions, and public APIs.
8. **Consistent Formatting and Indentation**: Adhere to a consistent coding style and indentation. This makes the codebase look clean and organized. Most programming languages have community-accepted coding standards (e.g., PEP 8 for Python, eslint for JavaScript) that you should follow. Consistency also applies to naming conventions, spacing, and code structure.
9. **Use Meaningful Whitespace**: Properly format your code with spaces and line breaks. This enhances readability. Use whitespace to separate logical sections of your code. Well-formatted code is easier to scan, reducing the cognitive load on readers.
10. **Error Handling**: Handle errors gracefully. Use appropriate try-catch blocks or error-handling mechanisms in your code. This prevents unexpected crashes and provides valuable information for debugging. Don't suppress errors or simply log them without a proper response.
11. **Testing**: Write unit tests to verify your code's correctness. Test-driven development (TDD) can help you write cleaner code by forcing you to consider edge cases and expected behavior upfront. Well-tested code is more reliable and easier to refactor.
12. **Refactoring**: Refactor your code regularly. As requirements change and your understanding of the problem domain deepens, adjust your code accordingly. Refactoring helps maintain clean code as the project evolves. Don't be afraid to revisit and improve existing code when necessary.
13. **Version Control**: Use version control systems like Git to track changes to your code. This allows you to collaborate effectively with team members, revert to previous versions if necessary, and maintain a clean history of your project's development. Git provides tools for code review, branching, and merging, facilitating collaboration and code cleanliness.