# **A** **Developer**

1. Aka mediocre developer
2. They are Good at their job
3. Produce functional code
4. May not prioritize on
   * code readability,
   * maintainability,
   * adherence to best practices

# **Best Developer additionally focus on:**

### **Elevating Code Quality:**

The presentation aims to guide developers towards writing code that is not only functional but also of high quality.

### Improving Readability and Maintainability:

We'll explore techniques to make code more readable, understandable, and easier to maintain over time.

### **Reducing Technical Debt\*:**

By adhering to clean code principles, we can minimize technical debt and enhance the long-term viability of our projects. \*

**Technical dept:** Cost of additional work and challenges that arise when shortcuts or less-than-optimal solutions are used in the development process)

### **Enhancing Collaboration and Productivity:**

Clean code fosters effective collaboration among team members and boosts productivity in development and maintenance phases.

### **Significance of clean code in software development:**

### **Readability and Understandability:**

Clean code is easily readable and comprehensible, making it accessible to all team members. This facilitates collaboration, reduces the learning curve for new developers, and ensures that the code's logic and intent are clear.

### **Maintainability and Extensibility:**

Clean code is designed for ease of maintenance and extensibility. It allows for seamless updates, bug fixes, and the addition of new features. This reduces the risk of introducing errors during modifications and lowers the long-term cost of ownership.

### **Debugging and Troubleshooting:**

Clean code minimizes the time and effort required for debugging and troubleshooting. Clear and well-organized code makes it easier to identify and rectify errors, leading to more efficient development cycles.

### **Reduced Technical Debt:**

Writing clean code helps in managing and minimizing technical debt. By avoiding shortcuts and implementing best practices, developers can prevent the accumulation of unnecessary complexity and maintenance overhead over time.

# **Current State of Code Quality**

***Take some time to review the code:***

<https://github.com/santoshkar/CleanCodeDemo/blob/main/src/main/java/com/cleancode/demo/DispatchInformation.java>

**branch: main**

### Can you please take a look at the code and understand what it's supposed to do?

(The HTTP Session is getting the sales and purchase data as request and sending the information to respective departments)

### What challenges did you face (due to code quality issues)?

### **Modularity and Separation of Concerns:**

The class appears to be **performing multiple tasks** (getting department information, processing employees, sending information), which might violate the Single Responsibility Principle. Consider breaking down the class into smaller, more focused classes or methods.

### **Error Handling:**

Error handling is minimal. **It throws an exception** if the department list is empty, but it might be more informative to include a message or logging to provide context about why the exception was thrown.

### **Comments and Documentation:**

There are **commented-out code blocks and commented-out lines of code**. This can make the code less readable and cluttered. If those sections are not needed, they should be removed. If they are necessary, consider providing descriptive comments explaining their purpose.

### **Input Validation:**

There is **no input validation for the HttpServletRequest** req parameter. It might be a good idea to add some validation checks to ensure that req is not null and that it contains the expected data.

### **Null Checks:**

There are some null checks in the code, but they could be more extensive. For instance, it might be a good idea to check if slSrvc and pSrvc are null before calling their methods.

### **Exception Handling:**

**Exception handling could be improved**. Consider adding meaningful error messages to the exceptions that are thrown.

### **Logging:**

### Question:- how many of you always add logs when you code?

**It might be beneficial to include logging statements** to provide more insight into the execution flow and to aid in debugging.

### **Variable Naming:**

Some variable names could be more descriptive. For example, map1 and map2 don't provide much information about their purpose.

### **Code Formatting and Style:**

The code formatting could be improved for better readability. Consistent indentation and spacing can make the code easier to understand.

### **Testing:**

There are no tests provided with the code. Writing unit tests can help ensure that the code behaves as expected and can catch any regressions.

# **Bad Code vs Clean Code:**

Show the cleaned code (Branch: ***cleancode***)

1. Declared fields as private (***Encapsulation***)
2. Renamed Field names:
   1. **pSrvc** -> purchaseService
   2. **slSrvc** -> salesService
3. Removed commented code
4. Renamed method from 'send' to 'sendInformation'
5. Removed System.out statements
6. Renamed variable name from
   1. map1 -> salesEmp
   2. map2 -> purchaseEmp
   3. lst -> deptList
7. Reduced nested if-else/loop
8. Improved loop syntax
9. Created a new method: buildEmployeeMap()

# **The Benefits of Clean Code**

### **Enhanced Readability:**

Clean code is easy to read and understand, even for developers who did not write it. This leads to quicker comprehension and reduces the likelihood of misinterpretation, making it more efficient for teams to work together.

### **Improved Maintainability:**

Clean code is designed with future changes in mind. It's easier to modify and extend, which reduces the time and effort required for maintenance tasks. This leads to lower long-term costs and a more sustainable codebase.

### **Streamlined Collaboration:**

Clean code promotes effective collaboration among team members. With clear, well-structured code, developers can more easily work together on different parts of the system, ensuring smoother integration and faster development cycles.

### **Reduced Debugging Time:**

Clean code leads to fewer bugs and errors. When issues do arise, they are easier to locate and fix due to the clear and organized nature of the code. This results in shorter debugging and troubleshooting times.

### **Faster Onboarding for New Developers:**

Clean code is more accessible to new team members. They can quickly grasp the codebase, allowing them to become productive contributors sooner. This accelerates the onboarding process.

### **Better Code Reviews:**

Clean code facilitates more effective code reviews. Reviewers can focus on higher-level design decisions and potential optimizations, rather than getting bogged down by confusing or poorly written code.

### **Reduced Technical Debt:**

Writing clean code helps prevent the accumulation of technical debt. By avoiding shortcuts and implementing best practices, developers can keep the codebase manageable and prevent it from becoming unnecessarily complex over time.

### **Higher Quality Software:**

Clean code leads to higher-quality software. It is less error-prone, easier to test, and more robust. This ultimately results in a more reliable and stable product for end users.

# **Guidelines for writing clean code:**

### **Meaningful Names:**

Choose descriptive and meaningful names for variables, functions, classes, and modules. Names should convey the purpose and intent of the entity they represent.

### **Single Responsibility Principle (SRP):**

Each function, class, or module should have a single, well-defined purpose. This promotes modular and maintainable code.

### **Don't Repeat Yourself (DRY):**

Avoid duplication of code. Reuse common functionality through functions, classes, or modules.

### **Keep Functions/Methods Small:**

Functions should ideally do one thing and do it well. They should be concise and focused on a specific task.

### Avoid Deep Nesting:

Excessive levels of indentation can make code hard to read and understand. Aim for a maximum of two or three levels of indentation.

### **Comments and Documentation:**

Write clear and concise comments that explain the purpose and logic of your code. Avoid redundant or misleading comments.

### **Formatting and Indentation:**

Follow a consistent code style. Use indentation to clearly show the structure of your code.

### **Use Descriptive Function/Method Signatures:**

Method and function signatures should clearly convey their purpose and the type of data they operate on.

### **Avoid Magic Numbers and Strings:**

Replace numeric or string constants with meaningful names. This makes the code more readable and maintainable.

### **Error Handling:**

Implement appropriate error handling to gracefully handle exceptions and errors.

### **Separation of Concerns:**

Different parts of your code should have distinct responsibilities. For example, separate user interface code from business logic.

### **Consistent Naming Conventions:**

Follow consistent naming conventions for variables, functions, classes, and other entities. This helps maintain uniformity across your codebase.

### **Use Version Control:**

Utilize version control systems like Git to track changes and collaborate with other developers.

### **Unit Testing:**

Write automated tests to verify the correctness of your code. This helps catch bugs early and ensures that changes don't introduce new issues.

### **Refactor Regularly:**

Keep your codebase clean by refactoring it regularly. This involves restructuring code to improve its readability, maintainability, and performance.

### **Avoid Complex and Nested Conditionals:**

Break down complex conditional statements into simpler, more manageable parts.

### **Limit Function/Method Arguments:**

Avoid functions with a large number of parameters. If a function requires many arguments, it may be an indication that it's doing too much.

### **Use Design Patterns Wisely:**

Apply design patterns where appropriate, but avoid over-engineering. Use patterns to solve specific problems, not as a one-size-fits-all solution.

### **Optimize for Readability:**

Prioritize code readability over clever optimizations. Clear, readable code is easier to maintain and debug.

### **Continuous Learning and Improvement:**

Stay updated with best practices, new technologies, and coding standards. Continuously seek ways to improve your coding skills.

# **SOLID principles**

**(Refer the program where created examples of each principle)**

### **What benefits?**

1. **Maintainability**
2. **Flexibility and Adaptability**
3. **Testability**
4. **Reusability**
5. **Scalability**
6. **Reduced Dependencies**

### S - Single Responsibility Principle (SRP)

### O - Open-Closed Principle (OCP)

### L - Liskov Substitution Principle (LSP)

### I - Interface Segregation Principle (ISP)

### D - Dependency Inversion Principle (DIP)

1. **S - Single Responsibility Principle (SRP)**:

This principle states that

* A class should have only one reason to change,
* meaning it should have only one responsibility or job.

**public** **class** Employee {

**private** String name;

**private** String department;

**private** **double** salary;

// Getters and setters for name, department, and salary

**public** **void** saveToDatabase(Employee employee) {...}

**public** **void** generatePaySlip(Employee employee) {...}

**public** **void** sendEmailNotification(Employee employee) {...}

**public** **void** printReport(Employee employee) {...}

}

In this example, the Employee class is responsible for multiple tasks:

1. Saving employee data to the database.
2. Generating pay slips.
3. Sending email notifications.
4. Printing reports.

This violates the Single Responsibility Principle because the class has multiple reasons to change. For example, if the database schema changes, the saveToDatabase method would need to be modified. If the pay slip format changes, the generatePaySlip method would need to be updated, and so on.

1. **O - Open-Closed Principle (OCP)**:

This principle emphasizes that a class should be open for extension but closed for modification. It means that you should be able to add new functionality to a class without altering its existing code. This is typically achieved through inheritance, interfaces, and abstraction.

**public** **class** DiscountCalculator {

**public** **double** calculateDiscount(**double** totalPrice, String discountType) {

**if** (discountType.equals("percentage")) {

**return** totalPrice \* 0.2; // 20% discount

} **else** **if** (discountType.equals("fixed")) {

**return** 50; // $50 fixed discount

} **else** {

**throw** **new** IllegalArgumentException("Invalid discount type");

}

}

}

method '*calculateDiscount'* that calculates a discount based on the '***discountType'***. However**, if we want to add a new type of discount (e.g., a buy-one-get-one-free discount)**, we would need to modify the existing code, which violates the OCP.

1. **Liskov Substitution Principle (LSP):**

The Liskov Substitution Principle (LSP) states that

* Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.
* In other words, any code that works with a base class should also work with derived classes without knowing it.

Here's an example in Java that violates the Liskov Substitution Principle:

**class** Rectangle {

**protected** **int** width;

**protected** **int** height;

**public** **void** setWidth(**int** width) {

**this**.width = width;

}

**public** **void** setHeight(**int** height) {

**this**.height = height;

}

**public** **int** getArea() {

**return** width \* height;

}

}

**class** Square **extends** Rectangle {

@Override

**public** **void** setWidth(**int** width) {

**super**.setWidth(width);

**super**.setHeight(width);

}

@Override

**public** **void** setHeight(**int** height) {

**super**.setHeight(height);

**super**.setWidth(height);

}

}

In this example, we have a `**Rectangle**` class and a `**Square**` class that extends `**Rectangle**`. The `**Square**` class attempts to override the `setWidth` and `setHeight` methods to ensure that both the width and height are always equal.

However, this violates the Liskov Substitution Principle because a `**Square**` is not a true substitute for a `**Rectangle**` in terms of behavior. Specifically, if we use a `**Square**` in a context where a `**Rectangle**` is expected (e.g., resizing the width and height independently), it will lead to unexpected behavior.

1. **Interface Segregation Principle (ISP):** The Interface Segregation Principle (ISP) states that

* clients should not be forced to depend on interfaces they do not use.
* In other words, it's better to have smaller, more specific interfaces tailored to the needs of the clients, rather than one large interface that includes methods irrelevant to certain clients.

Here's an example in Java that violates the ISP

**interface** Worker {

**void** work();

**void** eat();

**void** sleep();

}

**class** Manager **implements** Worker {

@Override

**public** **void** work() {}

@Override

**public** **void** eat() {}

@Override

**public** **void** sleep() {}

}

**class** Developer **implements** Worker {

@Override

**public** **void** work() {}

@Override

**public** **void** eat() {}

@Override

**public** **void** sleep() {}

}

In this example, we have an **`Worker`** interface with `**work**`, `**eat**`, and `**sleep**` methods. Both `Manager` and `**Developer**` classes implement this interface.

However, this violates the Interface Segregation Principle because not all clients (e.g., managers and developers) need all the methods defined in the `**Worker**` interface. For instance, a `**Manager**` might not need to implement the work method, as their responsibilities may not involve the same type of `**work**` as a `**Developer**`.

1. **D - Dependency Inversion Principle (DIP):** The Dependency Inversion Principle (DIP) states that

* High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (interfaces or abstract classes).
* Additionally, abstractions should not depend on details; details should depend on abstractions.

Here's an example in Java that violates the DIP:

**class** Database {

**public** **void** saveData(String data) {}

}

**class** Application {

**private** Database database;

**public** Application(Database database) {

**this**.database = database;

}

**public** **void** processData(String data) {

// Process data

database.saveData(data);

}

}

In this example, the `Application` class has a direct dependency on the `**Database**` class. This violates the Dependency Inversion Principle because high-level module (`**Application**`) is directly dependent on a low-level module (`**Database**`).

To adhere to the DIP, we should introduce an abstraction (interface or abstract class) between the high-level and low-level modules.

# **Test-Driven Development (TDD)**

* A software development approach in which tests are written before the actual code that needs to be implemented.
* It follows a cyclical process of writing a test, writing code to pass that test, and then refactoring the code to improve it while ensuring it still passes all tests.
* This cycle is commonly referred to as the "Red-Green-Refactor" cycle.

1. Red:

* Write a test that defines a function or behavior, but the test initially fails because the function or behavior has not yet been implemented.

1. Green:

* Write the minimum amount of code necessary to pass the test. The focus is on making the test pass, not on writing perfect or complete code.

1. Refactor:

* Clean up the code while ensuring it still passes all tests. This step involves improving the design, removing duplication, and making the code more maintainable.

This cycle is repeated for each new piece of functionality that needs to be added or modified in the codebase. The goal is to have a comprehensive suite of tests that cover all aspects of the code's functionality.

TDD is known for its benefits in:

1. improving code quality,
2. catching bugs early,
3. providing executable documentation, and
4. promoting modular and maintainable code.
5. It also gains more confident approach to making changes or refactoring existing code.