Clean Code

Gathered by Danial Salari

Pre-read:

I tried to simplify and summarize the Clean Code book by Robert C. Martin with the help of **Chatgpt**. This PDF dives deeply with all the **important details**, **examples**, and the **core principles** that make the book so influential.
You may or may not know many of these things so don't go hard on this.

Introduction: The Importance of Clean Code

Key Idea:

Clean code is code that is easy to read, understand, and maintain. Writing clean code is not about writing code that just works, but code that is elegant, simple, easy to debug, and easy to extend over time.

Martin argues that clean code is an art: it's not just technical; it's about creating a codebase that is pleasant to work with. Code that looks like garbage today will be garbage tomorrow.

Example:

Consider the long-term maintenance of a messy system. What may seem like "working code" at first might become impossible to extend and debug in the future. Clean code avoids these problems by keeping things manageable and understandable.

1. Meaningful Names

Key Idea:

Names are incredibly important. Good names improve readability and understanding of code. Naming conventions guide developers in understanding the purpose of variables, methods, classes, and functions at a glance.

Principles:

• Descriptive names: Use names that describe the intent of the code.

• Avoid abbreviations: Abbreviations can lead to confusion.

• Use pronounceable names: If you can't pronounce it, you probably can't discuss it with colleagues.

• Don't use single-letter names, except for loop counters or variables in tight scopes (like i in a for-loop).

Examples:

• Bad Example:

```
int t; // What does 't' represent? It's unclear.
```

• Good Example:

```
int totalAmount; // Immediately clear and easy to understand.
```

Real-World Application:

Imagine a class that stores data for customer orders. If you name a method processOrder() instead of something vague like doSomething(), it instantly tells any developer what that function is supposed to accomplish.

2. Functions: Keep Them Small and Focused

Key Idea:

A function should do one thing, and do it well. Functions should be short and focus on a single responsibility. The shorter a function is, the easier it is to read, understand, and maintain.

Principles:

• Small functions: A function should ideally fit on a single screen.

• Descriptive function names: A function name should describe what it does.

• Avoid side effects: Functions should not modify global states or variables.

Examples:

• Bad Example:

```
void processOrder(Order order) {
    // checks payment
    // ships the product
    // generates invoice
    // sends confirmation email
}
```

• Good Example:

```
void checkPayment(Order order) { ... }
void shipProduct(Order order) { ... }
void generateInvoice(Order order) { ... }
void sendConfirmationEmail(Order order) { ... }
```

Each function is doing one thing, which makes it easier to test and maintain

Real-World Application:

In a system that processes payment, instead of putting everything in processPayment(), we split it up into functions like verifyTransaction(), chargeCustomer(), and updatePaymentStatus(). This makes each part easy to understand, test, and modify in the future.

3. Comments: Use Them Sparingly

Key Idea:

Code should be self-explanatory. Good code doesn't need excessive comments to explain what it's doing. When you do need comments, make sure they explain why something is done, not how.

Principles:

• Avoid redundant comments: If the code is simple and clear, don't add comments just for the sake of it.

• Use comments to explain the "why": Comments should be used to clarify decisions that aren't immediately obvious from the code itself.

Examples:

- Bad Example:

```
int x = 5; // Set x to 5
```

- Good Example:

```
// Adjust price based on promotional discount for VIP customers
int price = calculateDiscount(order);
```

Real-World Application:

In complex logic or special cases, you'll often need a comment to explain why you're doing something unusual, but don't comment on the obvious, like the calculation of totalPrice = itemPrice + tax, which is self-explanatory.

4. Formatting: Make It Readable

Key Idea:

Proper formatting is critical for code readability. Clean code should follow a consistent style throughout the codebase to ensure that the structure of your code is easy to navigate.

Principles:

• Consistent indentation: Use consistent spaces or tabs throughout the codebase.

• Limit line length: Keep lines within a reasonable width (80–100 characters) to avoid horizontal scrolling and ensure readability.

• Separation of concerns: Separate logical sections of code with blank lines.

Examples:

• Bad Example:

```
if (x>0) {y++;} else {y--; return;}
```

• Good Example:

```
if (x > 0) {
    y++;
} else {
    y--;
    return;
}
```

Real-World Application:

Following a consistent style guide (e.g., Google Java Style or Airbnb's JavaScript Style Guide) helps ensure that everyone on the team writes in a way that's easy to read and understand. This makes collaboration smoother.

5. Objects and Data Structures: Keep It Encapsulated

Key Idea:

When designing objects, encapsulation is key. The internal details of an object should be hidden, and external code should interact with well-defined public methods.

Principles:

• Encapsulate data: Keep fields private and provide getter/setter methods to access or modify them.

• Use data structures effectively: Use the right data structures for the job (e.g., List, Set, Map).

Examples:

• Bad Example:

```
class Person {
    public String name;
    public int age;
}
```

• Good Example:

```
class Person {
    private String name;
    private int age;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Real-World Application:

Encapsulation helps protect the integrity of an object by restricting direct access to its fields and ensuring that data is modified in controlled ways.

6. Error Handling: Be Explicit

Key Idea:

Error handling should be clear, predictable, and explicit. Avoid using return codes or silent failures—use exceptions to signal errors clearly.

Principles:

• Use exceptions for exceptional cases.

• Don’t ignore exceptions: Always handle them appropriately, either by logging or throwing them.

• Keep error handling simple: Don’t use complex logic in exception-handling blocks.

Examples:

• Bad Example:

```
if (order.process() == -1) {
    // Handle failure silently
}
```

• Good Example:

```
try {
    order.process();
} catch (OrderProcessingException e) {
    // Handle specific error
    log.error("Error processing order: " + e.getMessage());
}
```

Real-World Application:

When you interact with external APIs or perform database operations, explicit error handling ensures that errors are managed in a structured way. For instance, if a payment fails, exceptions should be thrown and handled accordingly.

7. Classes: Organize with Purpose

Key Idea:

Classes should be small, focused, and have single responsibility. A class that has more than one reason to change violates the Single Responsibility Principle (SRP).

Principles:

• Small, cohesive classes: Each class should do one thing.

• Use clear class names to reflect the responsibility.

Examples:

• Bad Example:

```
class UserManager {
    void createUser(User user) { /* create user logic */ }
    void sendWelcomeEmail(User user) { /* send email logic */ }
}
```

• Good Example:

```
class UserRepository {
    void createUser(User user) { /* create user logic */ }
}

class EmailService {
    void sendWelcomeEmail(User user) { /* send email logic */ }
}
```

Real-World Application:

By splitting responsibilities, we make the code easier to test, extend, and maintain. For instance, we can update the email service or change the user storage strategy without impacting the other part.

8. Tests: Write Clean Tests

Key Idea:

Testing is crucial. Clean tests should be simple, focused, and easy to maintain. Each test should check only one thing.

Principles:

• Test only one thing at a time.

• Use meaningful test names.

• Keep tests independent of one another.

Examples:

• Bad Example:

```
@Test
void testOrderProcessing() {
    Order order = new Order();
    order.addItem(new Item("Apple", 1));
    order.addItem(new Item("Banana", 2));
    order.processPayment();
    order.generateInvoice();
    assertEquals(50, order.getTotalPrice());
}
```

- Good Example:

```
@Test
void testAddItem() {
    Order order = new Order();
    order.addItem(new Item("Apple", 1));
    assertEquals(1, order.getItemsCount());
}

@Test
void testProcessPayment() {
    Order order = new Order();
    order.addItem(new Item("Apple", 1));
    order.processPayment();
    assertTrue(order.isPaid());
}
```

Real-World Application:

Writing small and isolated tests allows for easier debugging and guarantees that the software behaves as expected.

9. Avoiding Long Methods

Key Idea:

Long methods are usually a sign that you're doing too much in one place. They're hard to understand and debug. Break methods up into smaller, focused functions that each do one thing.

Principles:

• Keep methods short: If a method is too long, it should be split up.

• Each method should have one reason to change: If a method is doing multiple things, it's harder to maintain.

Examples:

- Bad Example:

```
void processOrder(Order order) {
    // process payment
    // update inventory
    // generate invoice
    // send confirmation email
}
```

- Good Example:

```
void processOrder(Order order) {
    processPayment(order);
    updateInventory(order);
    generateInvoice(order);
    sendConfirmationEmail(order);
}

void processPayment(Order order) { ... }
void updateInventory(Order order) { ... }
void generateInvoice(Order order) { ... }
void sendConfirmationEmail(Order order) { ... }
```

By splitting the logic into smaller methods, each with a single responsibility, we make the code more maintainable, testable, and understandable.

10. Conclusion: Striving for Clean Code

In the conclusion, Martin emphasizes that writing clean code is a **continuous process**—it's a **habit**. You have to invest in writing clean, understandable code and make constant improvements over time. Clean code isn't just about reducing complexity for the sake of it, but also about enabling the code to be easily tested, maintained, and extended in the future.

Key Takeaways:

1. Code readability and maintainability are the highest priorities.

2. Follow principles like Single Responsibility, Open/Closed, and Don’t Repeat Yourself (DRY).

3. Use tools and methods like polymorphism, encapsulation, small functions, and clean names to achieve clarity.

4. Test early and often to ensure robustness.

5. Keep refactoring to improve code quality over time.