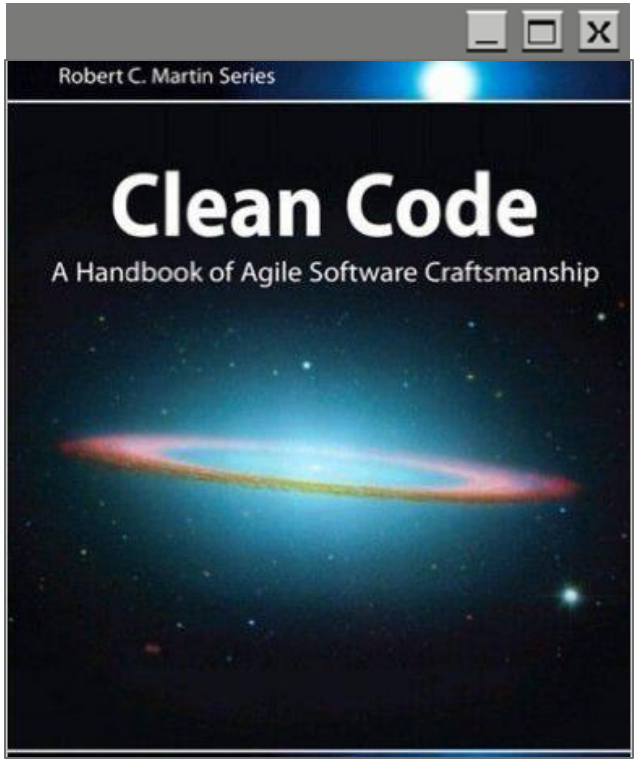
Clean Code

Advanced Programming
Dr. Mojtaba Vahidi Asl
Ramona Noroozi
Fall 1404

Table of contents



To read and learn more

If you'd like to dive deeper into writing better code, *Clean Code* by Robert C. Martin is a great place to start. It explains how to make your code more readable, organized, and easy to maintain, with plenty of examples and tips from real projects.

01 What's Clean Code?

Clean Code always looks like it was written by someone who cares. *-Michael Feathers*

What's Clean Code?

In a nutshell, **clean coding** is a programming practice that involves writing code in a way that makes it easy to read, understand and maintain.

The values of clean code are:

- changeability
- efficiency
- continuous improvement.

02 Why Clean Code?

It's not enough for code to work. *-Robert C. Martin*

Why Clean Code?

In addition to improved comprehensibility and maintainability through clean code principles, clean code offers many other advantages. Clean Code is:

- Elegant
- Efficient
- Direct
- Simple
- Readable

03 Clean Code Principles

Clean Code reads like well-written prose.
-Grady Booch

Clean Code Principles

Clean code principles are guidelines that, when followed, *can* lead to clean, efficient, and maintainable code.

1. KISS: Keep It Simple Stupid
2. DRY: Don't Repeat Yourself
3. SRP: Single Responsibility Principle
4. SLA: Single Level of Abstraction
5. Preferred Readability

Clean Code Principles

KISS

Code should be as simple and clear as possible. This approach avoids unnecessary complexity.

DRY

Duplication must be avoided at all costs. This saves development time and means that errors only have to be fixed once.

SRP

Each unit of code should have exactly *one* clearly defined task. This increases readability and simplifies testing.

Preferred Readability

‘Put yourself in my shoes!’. If there are several ways to do something, choose the one that is easiest for the reader to understand.

SLA

Let's imagine that the driver of a car is replaced by the function ‘DriveTo’. This function now controls the car – including the brakes, accelerator and steering wheel. However, it should not deal with more complex processes such as fuel supply and ignition. These tasks belong to a different level of abstraction.

Single Responsibility Principle

The Single Responsibility Principle (SRP) is a principle in software development that states that each class or module should have only one reason to change.
This makes it less likely to have side effects or dependencies that can make the code harder to work with.
This principle helps create code that is easy to:

- Understand
- Test
- Maintain
- Extend

```
// Example: Withouth SRP

function processOrder(order) {
  // validate order
  if (order.items.length === 0) {
    console.log("Error: Order has no items");
    return;
  }
  // calculate total
  let total = 0;
  order.items.forEach(item => {
    total += item.price * item.quantity;
  });
  // apply discounts
  if (order.customer === "vip") {
    total *= 0.9;
  }
  // save order
  const db = new Database();
  db.connect();
  db.saveOrder(order, total);
}
```

Modularization

It refers to the practice of breaking down large, complex code into smaller, more manageable modules or functions. Using modularization provides several benefits such as:

- Re-usability
- Encapsulation
- Scalability
- Easier to understand, maintain and test

Modularization

```
// Without modularization
function calculatePrice(quantity, price, tax) {
  let subtotal = quantity * price;
  let total = subtotal + (subtotal * tax);
  return total;
}

// With modularization
function calculateSubtotal(quantity, price) {
  return quantity * price;
}

function calculateTotal(subtotal, tax) {
  return subtotal + (subtotal * tax);
}
```

Reusability

Code reusability is a fundamental concept in software engineering that refers to the ability of code to be used multiple times without modification.
Reusing existing code has benefits such as:

- Saving time and effort
- Improving quality and consistency
- Minimizing the risk of introducing bugs and errors

```
// Example 1: No re-usability
function calculateCircleArea(radius) {
  const PI = 3.14;
  return PI * radius * radius;}
function calculateRectangleArea(length, width) {
  return length * width;
}
function calculateTriangleArea(base, height) {
  return (base * height) / 2;
}
const circleArea = calculateCircleArea(5);
const rectangleArea = calculateRectangleArea(4, 6);
const triangleArea = calculateTriangleArea(3, 7);
console.log(circleArea, rectangleArea, triangleArea);
```

```
// Example 2: Implementing re-usability
function calculateArea(shape, ...args) {
  if (shape === 'circle') {
    const [radius] = args;
    const PI = 3.14;
    return PI * radius * radius;
  } else if (shape === 'rectangle') {
    const [length, width] = args;
    return length * width;
  } else if (shape === 'triangle') {
    const [base, height] = args;
    return (base * height) / 2;
  } else {
    throw new Error(`Shape "${shape}" not supported.`);
  }
}
const circleArea = calculateArea('circle', 5);
const rectangleArea = calculateArea('rectangle', 4, 6);
const triangleArea = calculateArea('triangle', 3, 7);
console.log(circleArea, rectangleArea, triangleArea);
```

Format and Syntax

Indentation and spacing:

```
 // bad indentation and spacing
const myFunc=(number1,number2)=>{
const result=number1+number2;
return result;
}

// good indentation and spacing
const myFunc = (number1, number2) => {
    const result = number1 + number2
    return result
}
```

Format and Syntax

Consistent case conventions:

```
// camelCase
const myName = 'John'
// PascalCase
const MyName = 'John'
// snake_case
const my_name = 'John'
```

Meaningful Names

Use intention revealing names:

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Meaningful Names

Use intention revealing names:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Meaningful Names

Use pronounceable names:

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

Meaningful Names

Use pronounceable names:

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Meaningful Names

Use searchable names:

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

Meaningful Names

Use searchable names:

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Class Names

Classes and objects should have noun or noun phrase names like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Avoid words like `Manager`, `Processor`, `Data`, or `Info` in the name of a class. A class name should not be a verb.

Method Names

Methods should have verb or verb phrase names like `postPayment`, `deletePage`, or `save`. Accessors, mutators, and predicates should be named for their value and prefixed with `get`, `set`, and `is` according to the javabean standard.4

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Avoid Mental Mapping

Readers shouldn't have to mentally translate your names into other names they already know. This is a problem with single-letter variable names. Single-letter names for loop counters (such as `i`, `j`, or `k`) are traditional. However, in most other contexts a single-letter name is a poor choice; it's just a place holder that the reader must mentally map to the actual concept.

Functions

Small:
The first rule of functions is that they should be small. In the eighties it was said that a function should be no bigger than a screen-full.

- < 20 lines
- < 150 characters per line

Functions

Reading code from top to bottom, The Stepdown Rule:

We want the code to read like a top-down narrative. We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. This is called *The Stepdown Rule*.

Functions

Function Arguments:

The ideal number of arguments for a function is zero (niladic).
Next comes one (monadic), followed closely by two (dyadic).
Three arguments (triadic) should be avoided where possible.
More than three (polyadic) requires very special justification—
and then shouldn't be used anyway.

Functions

Have no side effects:

Your function promises to do one thing, but it also does other *hidden* things.

```
// do something or answer something, but not both
public boolean set(String attribute, String value);
```

```
setAndCheckIfExists
```

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
    ...
}
```

Comments

Comments do not make up for bad code:

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

Comments

Explain yourself in code:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Good Comments

Legal comments:

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

Good Comments

Informative comments:

It is sometimes useful to provide basic information with a comment. For example, consider this comment that explains the return value of an abstract method:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

Good Comments

Clarification comments:

Sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that's readable.

```
assertTrue(a.compareTo(a) == 0);    // a == a
assertTrue(a.compareTo(b) != 0);    // a != b
```

Good Comments

TODO comments:

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
   return null;
}
```

Bad Comments

Redundant comments:

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
   if(!closed)
   {
      wait(timeoutMillis);
      if(!closed)
         throw new Exception("MockResponseSender could not be closed");
   }
}
```

Bad Comments

Journal comments:

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new package
*               com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
*               class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
*               class is gone (DG);  Changed getPreviousDayOfWeek(),
*               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
*               bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
```

Bad Comments

Noise comments:

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}

/** The day of the month. */
    private int dayOfMonth;
```

04 Practices For Clean Coding

Bad code can be cleaned up, but It's very expensive. *-Robert C. Martin*

Practices For Clean Coding

Boy Scout Rule

Similar to scouts who leave a place cleaner than they found it, developers should always make the code a little better than they found it. This practice helps to implement clean code step by step and without much additional effort.

Root Cause Analysis

Treating symptoms superficially is like painting over cracks in a wall – it looks good, but doesn't last long. In the long run, it even leads to increased complexity and error-prone code. Developers who follow clean code principles should therefore always look for the root cause of the problem and fix it.

Test Driven Development

TDD is a design strategy for code that uses tests to guide the development of the implementation and enable safe refactoring.

```java
public class MakingCustomer{
    int a;
    int b;
    int t;

    MakingCustomer(int one, int two, int three){
        a = one;
        b = two;
        t = three;
    }

    int retrieve(){
        return a;
    }

    int get(){
        return b;
    }

    int setAndCalculate(int one){
        t = one;

        temp = a * b + t;
        result = 0;
        for(int a = 0; a < temp; a++){
            result = temp + result;
        }
        return result;
    }
}
```

```java
public class Calculator {
    public static void main(String[] args) {
        double result = e(10, 20);
        System.out.println("Sum is: " + result);
        boolean result2 = check(8);
        System.out.println(result2);
    }
    public static double e(int a, int b) {
        // sum
        double c;
        c = a + b;
        return c;
    }
    public static boolean check(int a){
        boolean flag = a % 2 ;
        if (flag == true){
            // even
            return true;
        }
        else{
            // odd
            return false;
        }
    }
}
```

Resources

- Clean Code, Robert C. Martin
- https://www.freecodecamp.org/news/how-to-write-clean-code/
- https://www.maibornwolff.de/en/know-how/clean-code/#clean-code-an-overview

Thank you

Do you have any questions?