Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 03: Methods

How should I write clean methods?

Methods play an important role in code as they define a class's behavior. You might have encountered methods spanning more than 100 lines, managing a multitude of tasks, dealing with excessive arguments, and returning values calculated from mathematical operations. Such methods, unfortunately, fall under the category of dirty code. However, simple rules exist that can simplify debugging and enhance the flexibility of code alteration.

Rules of clean methods:

- Rule Number 1: One Method – One Task

  Each method should implement one and only one task. While the implementation itself may involve calling multiple methods that execute a solitary job.

  If a method takes the responsibility of more than one task, it does the task either wrong or dirty.

  How could I know if the method does more than one job? One way to answer this question is to name the method as it reflects the sequence of events within it. If the method name contains **or**, **and** or **then** it does multiple tasks. Each proposition between these words is to be implemented as a method.

1) One Method – One Task

```
public void getUserInputAndSaveInDatabase(){
        // Code To Get User Input
        // Code To Save In Database
    }
```

```
public void SaveInDatabase(){
    SaveUserInputInDataBase(getUserInput());
}

public UserInput getUserInput(){
    // Code To Get And Return User Input
}

public void SaveUserInputInDataBase(UserInput userInput){
    // Code To Save User Input In Database
}
```

Rules of clean methods:

- Rule Number 2: Method Is Small As Possible.

  There is a saying about this rule: ***Extract till you drop.***

  We break a method down to smaller methods to reduce complexity and duplication.
  This would align with rule 1.

Rules of clean methods:

- Rule Number 3: The Fewer The Arguments, The Cleaner The Method

  Having too many arguments for a method confuses the reader and makes it more difficult to understand the method's behavior. It also complicates writing coverable tests.

  Best kind of method has no arguments. But in case of need we can increase the number of them. Although it is not recommended to use methods with 3 arguments and more.

  How could I decrease the number of arguments? We could maybe gather similar arguments in a single object.

3) The Less The Arguments, The Cleaner The Method

```
private Circle createCircle(double x, double y, double radius)
```

```
private Circle createCircle(Point center, double radius)
```

Rules of clean methods:

- Rule Number 4: No Boolean Arguments

  Introducing boolean arguments introduces branching logic, leading to two distinct paths depending on the argument's value. So it violates rule number 1. Because two tasks are being implemented in one method.

  Instead, we check its value before calling the method. And implement two methods. One to be called if the boolean value is **true** and one to be called if it is **false**.

4) No Boolean Arguments

```
private void mainMethod(){
    someMethod(someCondition);
}

private void someMethod(boolean condition){
    if(condition)
        doThis();
    else
        doThat();
}
```

```
private void mainMethod(){
    if (someCondition)
        doSomething();
    else
        doSomethingElse();
}

private void doSomething()
private void doSomethingElse()
```

Rules of clean methods:

- Rule Number 5: No Null Arguments

  Introducing a nullable argument introduces branching logic, leading to two distinct paths depending on the argument's value. So it violates rule number 1. Because two tasks are being implemented in one method.

  Instead, we check its value before calling the method. And implement two methods. One to be called if the variable value is **null** and one to be called if it is not.

5) No Null Arguments

```
private void mainMethod(){
    someMethod(someObject);
}

private void someMethod(Object someObject){
    if(someObject == null)
        doThis();
    else
        doThat();
}
```

```
private void mainMethod(){
    if (someObject == null)
        doSomething();
    else
        doSomethingElse();
}

private void doSomething()
private void doSomethingElse()
```

Rules of clean methods:

- Rule Number 6: Argument Should Not Be Changed In The Method Body.

  Changing the value of the argument makes it unpredictable for us to use the argument in another place in the code. So we declare the arguments as **final** so we are sure we do not accidentally change its value.

Rules of clean methods:

- Rule Number 7: No switch-case To Decide What To Do.

  In many situations we check the value of an object in a **switch** statement and decide what operation to perform in different **case**s. This is doing multiple tasks and violation of rule number 1.

  Instead we create a parent class and some child classes that inherit the parent. Each child class is related to each case. Then we create an object of parent class and initialize it in cases. All child classes have the same method only with different implementations. After initializing the object of parent class we call this method on the object. So the operation will be done specific for every case.

7) No switch-case To Decide What To Do.

```
switch(shapeType){
    case "Rectangle":
        area = side * side;
    case "Circle":
        area = 3.14 * radius * radius;
    case "Triangle":
        area = (base * height) / 2;
}
```

```
public interface Shape {
    public double area();
}
```

```
public class Rectangle implements Shape {
    private double side;

    @Override
    public double area() {
        return side * side;
    }
}
```

```
public class Circle implements Shape {
    private double radius;

    @Override
    public double area() {
        return radius * radius * 3.14;
    }
}
```

```
public class Triangle implements Shape {
    private double base;
    private double height;

    @Override
    public double area() {
        return base * height / 2;
    }
}
```

```
public double calculateShapeArea(){
    Shape shape = createShape(someShape);
    return shape.area();
}

public Shape createShape(Shape shape) {
    switch (shape.Type) {
        case "Rectangle":
            return new Rectangle();
        case "Circle":
            return new Circle();
        case "Triangle":
            return new Triangle();
    }
}
```

Rules of clean methods:

- Rule Number 8: Separate Commands From Queries.

  Imagine a method that sends a request or execute a command. But at the same time it executes query and returns data. It is both **setter** and **getter**.

8) Separate Commands From Queries.

```
public boolean saveInDatabase(Data data){
    // Code To Save Data In Database
    // Code To Check If The Data Was Successfully Saved In Database And Return The Result
}
```

```
public void saveInDatabase(Data data){
    // Code To Save Data In Database
}

public boolean isDataSaved(Data data){
    // Code To Check If The Data Was Successfully Saved In Database And Return The Result
}
```

Rules of clean methods:

- Rule Number 9: Avoid Returning Null.

Sometimes we return null if an operation couldn't be done or a variable has unexpected value. It increases the risk of **NullPointerException** which is a difficult exception to handle.

The primary alternative approach is to return an empty object with default values.

The best alternative approach is to define customized exceptions and throw them instead of returning **null**.

9) Avoid Returning Null.

```
public Password getPasswordFromUser() {
    Password result;
    if (isPasswordConfirmed())
        result = password;
    else
        result = null;

    return result;
}
```

```
public Password getPasswordFromUser() {
    Password result;
    if (isPasswordConfirmed())
        result = password;
    else
        throw new PasswordNotConfirmedException();

    return result;
}
```

Rules of clean methods:

- Rule Number 10: One Entry – One Exit

Using **return** or using **break** statements in the middle of method makes it dirty. The best practice is to have only one **return** in the method.

10) One Entry – One Exit

```
public Value getSomeValue() {
    if (condition)
        return value1;
    else
        return value2;
}
```

```
public Value getSomeValue() {
    Value result;
    if (condition)
        result = value1;
    else
        result = value2;

    return result;
}
```

Rules of clean methods:

- Rule Number 11: Blocking & Indentation

Each **for**, **while**, **if** and ... should have one command; which can be a method calling. The depth of indentation should not pass three levels. Meaning two nested decision and loop statements and one command.

11) Blocking & Indentation

```
public void doSomething() {
    if (condition) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                // Commands
            }
        }
    }
}
```

```
public void doSomething() {
    if (condition) {
        for (int i = 0; i < n; i++) {
            executeCommands();
        }
    }
}
```

Rules of clean methods:

- Rule Number 12: try-catch Extraction

If **try-catch** is needed to be used it should be the implementation of a method. The **try-catch** has to be isolated. The method begins with **try** and ends with **catch** or **finally**. In the **try** scope we call a method that throws a certain exception.

12) try-catch Extraction

```
private String openDocumentButtonPushed() {
    // Codes That Open And Read The Document
    try {
        // Code With Possibility Of An Exception
    } catch (SomeException exception) {
        showMessageBox("An error occurred while opening the document");
    }

    // Code To Process Data Read From The Document
}
```

```
private void openDocumentButtonPushed(){
    try{
        openDocument();
    } catch(NullPointerException | IOException exception){
        showMessageBox("An error occurred while opening the file");
    }
}

public void openDocument() throws IOException, NullPointerException {
    // Implementation
}
```