Mohammadmahdi Farrokhy

Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 03: Methods

How should I write clean methods?

Methods play an important role in code; they define a class's behavior. You might have encountered methods

- with more than 100 lines
- managing multiple of tasks
- dealing with excessive arguments
- returning values calculated from mathematical operations.

Such methods are listed as dirty code. However, there are some points to consider that makes methods way too clean.

Rules of clean methods:

- Rule #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. One way to know if the method does more than one job 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 separate 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 #2: Method Is Small As Possible.

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

  We break a method down to smaller ones to reduce complexity and code duplication. This aligns with rule #1.

Rules of clean methods:

- Rule #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. Although we can have method arguments, but it's not recommended to have more than with 3 arguments. One way to decrease the number of arguments is to gather similar ones 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 #4: No Boolean Arguments/Null/Enum Arguments

  Introducing boolean/nullable/enum arguments introduces branching logic, leading to multiple distinct paths depending on argument's value. So it violates rule #1. Because more than one task is being implemented in one method. We should check its value before calling the method and implement different methods for its different values.

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()
```

4) 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 #5: Argument Should Not Be Changed In The Method Body.

  Changing the argument's value makes it unpredictable and difficult for us to trace the code in debugging.

Rules of clean methods:

- Rule #6: No switch-case To Decide What To Do Or How To Do It.

  In many situations we check the value of an object in a switch statement and decide what operation to perform in different cases. This is doing multiple tasks and violates rule #1. Instead, we create an abstract class with abstract methods and some concrete classes that extend it. Each child class is related to one case. Then we create an object of the abstract class and initialize it in cases by concrete classes and call the abstract method on the object.

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

```
// BAD CODE
public Shape createShape(ShapeType shapeType) {
   switch(shapeType)
   {
      case ShapeType.Rectangle:
         return side * side;
      case ShapeType.Circle:
         return 3.14 * radius * radius;
      case ShapeType.Triangle:
         return (base * height) / 2;
   }
}
```

```
// GOOD CODE
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 Shape createShape(ShapeType shapeType) {
   switch (shapeType) {
      case ShapeType.Rectangle:
         return new Rectangle();
      case ShapeType.Circle:
         return new Circle();
      case ShapeType.Triangle:
         return new Triangle();
   }
}

public double calculateShapeAre() {
   Shape shape = createShape(shapeType);
   return shape.area();
}
```

Rules of clean methods:

- Rule #7: Separate Commands From Queries.

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

7) 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 #8: 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 first alternative approach is to return an object with default values. The best one is to define customized exceptions and throw them instead of returning null.

8) Avoid Returning Null.

```
// BAD CODE
public User createNewUser() {
   User newUser;

   if (isPasswordConfirmed)
      newUser = new User();
   else
      newUser = null;

   return newUser;
}
```

```
// GOOD CODE
public User createNewUser() {
   User newUser;

   if (isPasswordConfirmed)
      newUser = new User();
   else
      throw new PasswordNotConfirmedException();

   return newUser;
}
```

Rules of clean methods:

- Rule #9: One Entry – One Exit

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

9) One Entry – One Exit

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

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

   return result;
}
```

Rules of clean methods:

- Rule #10: Blocking & Indentation

The depth of blocks should not pass three levels.

10) Blocking & Indentation

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

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

Rules of clean methods:

- Rule #11: try-catch Extraction

If **try-catch** is required, it should be isolated as the implementation of a method. The method begins with try and ends with catch or finally. In the try scope we call a method that might throw a certain exception.

11) try-catch Extraction

```
// BAD CODE
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
}
```

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

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