Software Engineering 2

Lecture 1
Clean Code
Functions

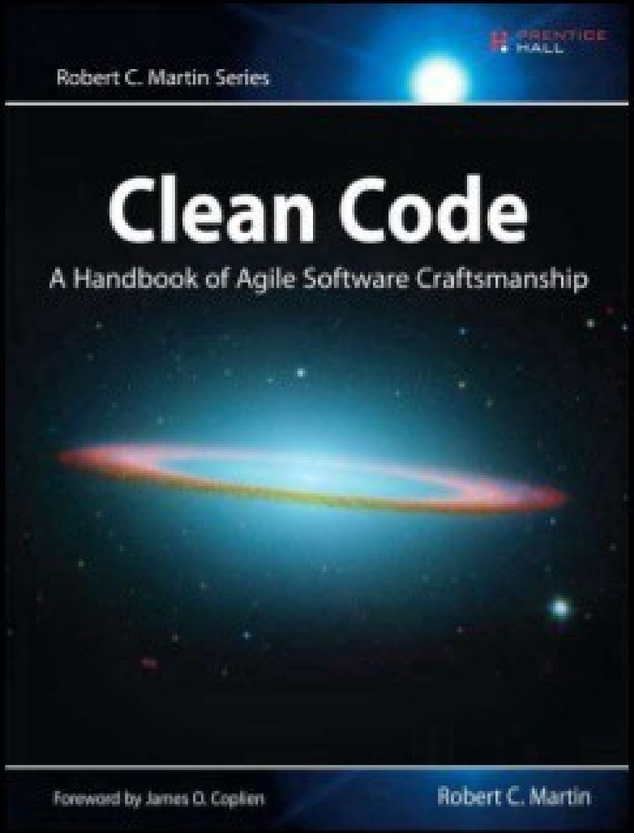
Dr. Ahmed Hesham Mostafa

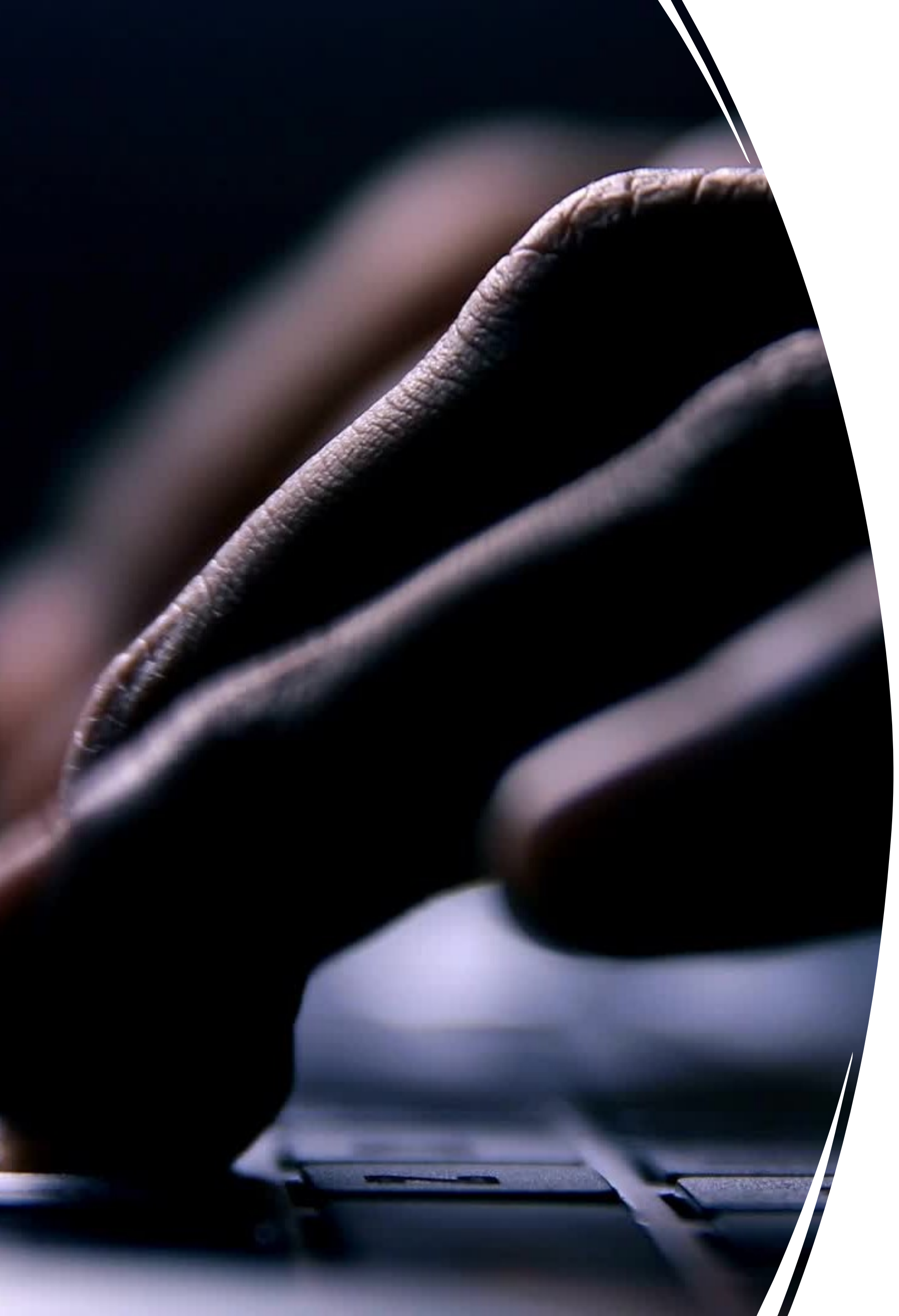
The “Must Read”-Book(s)

by Robert C Martin

A Handbook of Agile Software Craftsmanship

“Even bad code can function. But if code isn’t clean, it can bring a development organization to its knees.”


PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin



Clean Code

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." Martin Fowler

Keep simple, small functions with meaningful names

- The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.
- Remember that your code must be clean enough to be easily readable without spending too much time trying to guess what a function does.
- Smaller functions are easier to read and to understand.

Long Method

- A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- To reduce the length of a method body, use **Extract Method**.
-

Do One Thing

- Functions Should Do One Thing.
- Functions Should Do It Well.
- Functions Should Do It Only.
- Your functions should do only one thing.
- If you follow this rule, it is guaranteed that they will be small.

Do One Thing

- The only thing that function does should be stated in its name.
- Sometimes it is hard to look at the function and see if it is doing multiple things or not
- One good way to check is to try to extract another function with a different name.
- If you can find it, that means it should be a different function.

Example – Long Method Before Refactoring

```
public static void addDataToFile(String path,Person person)  {
        FileInputStream readData = new FileInputStream(path);
        ObjectInputStream readStream = new ObjectInputStream(readData);
        ArrayList people = (ArrayList<Person>) readStream.readObject();
        readStream.close();
        people.add(person);
        FileOutputStream writeData = new FileOutputStream(path);
        ObjectOutputStream writeStream = new ObjectOutputStream(writeData);
        writeStream.writeObject(people);
        writeStream.flush();
        writeStream.close();
        System.out.println(people.toString());
   }
```

Extract methods from long method

```
private static void PrintPersonDataFile(ArrayList people ) {
        System.out.println(people.toString());
    }
    private static void writeDataTofile(String path, ArrayList people) {
        FileOutputStream writeData = new FileOutputStream(path);
        ObjectOutputStream writeStream = new ObjectOutputStream(writeData);
        writeStream.writeObject(people);
        writeStream.flush();
        writeStream.close();
    }
    private static ArrayList readDataFromFile(String path) {
        FileInputStream readData = new FileInputStream(path);
        ObjectInputStream readStream = new ObjectInputStream(readData);
        ArrayList  people = (ArrayList<Person>) readStream.readObject();
        readStream.close();
        return people;
    }
```

Example – Long Method After Refactoring

```
public static void addDataToFile(String path,Person person) {
        ArrayList people = readDataFromFile(path);
        people.add(person);
        writeDataTofile(path, people);
        PrintPersonData(people);
    }
```

Do One Thing

- For example, addDataToFile function has its role to add the person to data file not print or display any data or add the item to Arraylist.
- So, remove PrintPersonData(people) and refactor it

```
public static void addDataToFile(String path,Person person) {
        ArrayList people = readDataFromFile(path, person);
        people.add(person);
        writeDataTofile(path, people);
         PrintPersonData(people);
     }
```

Refactoring addDataToFile() Method

- Move the cod section for adding a person to the people list to the new method addDataToList and also move the printing function out.
- Change the parameter from adding person only to actually responsibility to add a list of people to file

```
public static ArrayList addDataToList(Person person,ArrayList people){
        people.add(person);
        return people;
  }
public static void addDataToFile(String path,ArrayList people){
        writeDataTofile(path, people);
  }
```

Calling all Methods at Main(User Level)

```
public static void main(String[] args) {
        Person p3 = new Person("ahmed", "hesham", 1990);
        String filePath="peopledata.ser";
        ArrayList people = readDataFromFile(filePath);
        addDataToList(p3,people);
        addDataToFile(filePath,people);
        ArrayList afterAddpeople = readDataFromFile(filePath);
        PrintPersonDataFile(afterAddpeople);
    }
```

One Level of Abstraction per Function

- Also called Single Level of Abstraction Principle (SLAP)
- A method is a form of abstraction
- In order to make sure our functions are doing “one thing,” we need to make sure that the statements within our function are all at the same level of abstraction.
- Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail.

Example

```
private static void printNumbersOneToThirtyFive(){

 printNumbersOneToTen();  //high level of abstraction
                                   /*low level of abstraction */
 for(int i=11; i < 21; i++) {
   System.out.println(i);
 }
 System.out.println(21);
 if(22 % 11 == 0) {
   System.out.println(22);
   for(int i=23; i < 31; i++) {
       System.out.println(i);
       if(i == 30) {
           for(int j=31; j < 36; j++) {
           System.out.println(j);}}}}  }
}
```

```
 private static void printNumbersOneToTen() {
   for(int i=1; i < 11; i++) {
     System.out.println(i); }}}
```

One Level of Abstraction per Function

- The details of printing the numbers 1 to 10 have been hidden away behind a method named *printNumbersOneToTen*.
- We should be able to get a good understanding of what a method does from its name alone.
- So If this method does what its name says then that’s all we need to know to continue reading through printNumbersOneToThirtyFive() method

One Level of Abstraction per Function

- after we get passed the invocation of *printNumbersOneToTen* we start to delve into some very detailed steps of how the numbers 11 to 35 are printed.
- To understand that process we have to read 27 lines of code.
- This is where the brain has to do extra work by jumping between the two levels of abstraction.
- So the solution to divide the sections of code into other methods

One Level of Abstraction per Function

- The method has same level of abstraction

```
public static void main(String[] args) {
  printNumbersOneToThirtyFive();
}

private static void printNumbersOneToThirtyFive() {
  printNumbersOneToTen();
  printNumbersElevenToTwenty();
  printNumberTwentyOne();
  printNumbersTwentyTwoToThirtyFive();
}
```

The fewer function arguments, the better

- Function arguments are troublesome and make it harder to understand the function. It is best to build zero-argument functions, then one-argument and two-argument functions.
- Try to avoid functions with 3 arguments, and functions with multiple arguments (3+) should not be used at all.
- For example, writeText(text) function is way easier to understand than writeText(outputStream, text).
- You can always get rid of the outputStream argument by defining it as a class field.

The fewer function arguments, the better

- Arguments are even harder from a testing point of view. Imagine the difficulty of
- writing all the test cases to ensure that all the various combinations of arguments work
- properly. If there are no arguments, this is trivial. If there’s one argument, it’s not too hard.
- With two arguments the problem gets a bit more challenging. With more than two arguments, testing every combination of appropriate values can be daunting.

The fewer function arguments, the better

- When a given function requires too many arguments, some of them might probably be placed in separate classes. For instance:

```
public void sendRequest(String ip, int port,String user, String password,String body);
```

```
public void sendRequest(Request request);
```

Flag Arguments

- Flag arguments are ugly.
- Passing a Boolean into a function is a truly terrible practice.
- It means the function does more than one thing.
- It does one thing if the flag is true and another if the flag is false!
- We should have split the function into two.

Flag Arguments

```
Public void printAddress(Boolean long)
{
If(long)
//code for printing Long address
else
//code for pronting Short address
}
```

```
Public void printshortAddress()
{
//code for printing short address
}
Public void printLongAddress()
{
//code for pronting Long address
}
```

Keep small, Blocks and if-else

- the blocks within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call.
- Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.
- This also implies that functions should not be large enough to hold nested structures.
- Therefore, the indent level of a function should not be greater than one or two.
- This, of course, makes the functions easier to read and understand.

Replace Conditional With Polymorphism

- If you have code that splits a flow or acts differently based on some condition with limited values, with constructs like if or switch statements, then that code naturally violates Open Closed principle because to add a new conditional flow you will have to modify that class.
- Such a conditional check, if not designed correctly, is likely to be duplicated at multiple places.
- This principle suggests to use polymorphism to replace the conditional.

Replace Conditional With Polymorphism

- Idea is to add new conditional flow as a new implementation of well-defined abstraction instead of a new case in if-else ladder or switch statement.
- In many cases, you can not get rid of all the conditionals as somewhere you will have to decide to choose the correct implementation of inheritance hierarchy.
- But idea is to limit such conditionals, probably, to a single place instead spreading it all over the code.
- If can not be completely gotten rid of, the single conditional should be at boundaries of the code integrations.

Replace Conditional With Polymorphism

- Prefer to have the code flow split as early as possible, so that maximum of code would be independent of the conditional.
- General rule for switch statements is that they can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance relationship so that the rest of the system can’t see them

Example
Vehicle Class

```
public class Vehicle {
    private Type type;
    enum Type { BICYCLE,BIKE,CAR,BUS}
    enum ParkingSpotSize {SMALL,MEDIUM,LARGE,XL}
    public ParkingSpotSize spaceRequiredForParking() {
        switch (type) {
            case BICYCLE:
                return ParkingSpotSize.SMALL;
            case BIKE:
                return ParkingSpotSize.MEDIUM;
            case CAR:
                return ParkingSpotSize.LARGE;
            case BUS:
                return ParkingSpotSize.XL; }}
    public BigDecimal parkingCharges() {
        switch (type) {
            case BICYCLE:
                return BigDecimal.valueOf(10.00);
            case BIKE:
                return BigDecimal.valueOf(20.00);
            case CAR:
                return BigDecimal.valueOf(50.00);
            case BUS:
                return BigDecimal.valueOf(100.00);}}}
```

Example - Vehicle Class

- many conditionals (Hard for readability).
- Violate the rule of open close principle
- We must change these places when a new vehicle type is to be added.
- We must change these charges when a new vehicle type is to be added.
- Violate the rule of must do one thing

Solution

- Create interface class Vehicle
- Adding new vehicle type involve adding new implementation.
- Separate each Vehicle type to separated implementation class

```
public interface Vehicle {
    ParkingSpotSize spaceRequiredForParking();
    BigDecimal parkingCharges();
    enum ParkingSpotSize {SMALL,MEDIUM,LARGE,XL
 }
}
```

Solution

- Create implantation class for Car, Bike, Bus, Bicycle and for each new Vehicle type
- For example, the implementation class for car will be

```
public class Car implements Vehicle {
   @Override
   public ParkingSpotSize spaceRequiredForParking()
{
      return ParkingSpotSize.LARGE;    }
   @Override
   public BigDecimal parkingCharges() {
      return BigDecimal.valueOf(50.00);  }
}
```

References

- Replace Conditional With Polymorphism - design-principles (injulkarnilesh.github.io)
- The Single Level of Abstraction Principle (SLAP) - danparkin.com
- Robert C. Martin, “Clean Code: A Handbook of Agile Software Craftsmanship”.
- Refactoring: clean your code

THANKS
SEE U NEXT LECTURE