Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 12: Emergence

Simple Design Rule

One approach to have clean code is via emergence design. Based on Beck's opinion, the design is simple if it follows these rules:

- Runs all the tests.
- It has no duplication.
- It is expressive.
- It has minimal classes and methods.

These rules are prioritized in order of importance.

Rule Number 1

- Runs All The Tests.

We start be designing a system that works as expected. However, even if a system theoretically functions well, it's all in vain if we can't easily verify it. A system that can be thoroughly tested and consistently passes those tests is what we consider as testable system. This phrase might seem really clear, but it’s also really important. Remember, a system that can't be effectively tested and verified is a red flag – it's up for debate whether it's even worth developing.

As we increase the number of tests, two important things happen. Firstly, we're getting closer to having testable components. So we’ll have confidence for building better system design grows. Secondly it opens up the opportunity to apply principles like **dependency inversion**, **dependency injection**, and the use of **interfaces** and **abstractions**.

Refactoring

After the system passed the tests, it's time to clean our classes and their codes. This could happen by gradual refactoring. We change the structure of the code; not its logic. After changing the code we run all the tests again. If they're passed our refactoring did not interrupt the business logic. If not, we have to fix the problem or undo our changes.

How do we start the refactoring?

This question could be quite difficult to answer. So wen can do it by applying last three rules of a simple design.

Rule Number 2

- No Duplication

Duplication is the enemy of a well-designed system. Duplication means additional effort, additional risk and unnecessary complexity. It reveals itself in variant forms.

Lines with exact same code are duplication.

Lines with similar codes are duplication.

They could be manipulated, standardized and be converted into a method.

Rule Number 2

```
public static void dirtyCode(String... args) {
    JButton button1 = new JButton("1");
    button1.setBounds(0, 0, 50, 60);
    button1.setBackground(Color.white);
    button1.setBorder((BorderFactory.createLineBorder(Color.white)));

    JButton button2 = new JButton("2");
    button2.setBounds(100, 100, 50, 60);
    button2.setBackground(Color.white);
    button2.setBorder((BorderFactory.createLineBorder(Color.white)));

    JButton button3 = new JButton("3");
    button3.setBounds(200, 200, 50, 60);
    button3.setBackground(Color.white);
    button3.setBorder((BorderFactory.createLineBorder(Color.white)));
}
```

Rule Number 2

```
private static final int BUTTON_WIDTH = 50;
private static final int BUTTON_HEIGHT = 60;
private static final Color BORDER_COLOR = Color.white;
private static final Color BACK_COLOR = Color.white;

public static void cleanCode(String... args) {
   JButton button1 = new JButton("1");
   button1.setBounds(0, 50, BUTTON_WIDTH, BUTTON_HEIGHT);
   button1.setBackground(BACK_COLOR);
   button1.setBorder(BorderFactory.createLineBorder(BORDER_COLOR));

   JButton button2 = new JButton("2");
   button2.setBounds(100, 150, BUTTON_WIDTH, BUTTON_HEIGHT);
   button2.setBackground(BACK_COLOR);
   button2.setBorder((BorderFactory.createLineBorder(BORDER_COLOR)));

   JButton button3 = new JButton("3");
   button3.setBounds(200, 250, BUTTON_WIDTH, BUTTON_HEIGHT);
   button3.setBackground(BACK_COLOR);
   button3.setBorder((BorderFactory.createLineBorder(BORDER_COLOR)));
}
```

Rule Number 2

```
private static final int BUTTON_WIDTH = 50;
private static final int BUTTON_HEIGHT = 60;
private static final Color BORDER_COLOR = Color.white;
private static final Color BACK_COLOR = Color.white;

public static void cleanCode(String... args) {
   JButton button1 = createButton("1", 0, 50);
   JButton button2 = createButton("2", 100, 150);
   JButton button3 = createButton("3", 150, 250);
}

private static JButton createButton(String text, int x, int y) {
   JButton button = new JButton(text);
   button.setBounds(x, y, BUTTON_WIDTH, BUTTON_HEIGHT);
   button.setBackground(BACK_COLOR);
   button.setBorder(BorderFactory.createLineBorder(BORDER_COLOR));
   return button;
}
```

Rule Number 3

- Expressive

The major cost of a software project is attributed to its long-term maintenance. To minimize the risk of encountering issues while applying modifications, we have to understand system's behavior and functionality. As the system becomes more complex, it becomes challenging for the developers to understand it, resulting in a higher possibility of misunderstandings. The solution lies in writing code that clearly reflects the author's intentions. Code that is well-written and easily understandable significantly reduces the time needed for others to understand it. This leads to the reduction of losses and lower maintenance cost.

This could be done by keeping methods and classes small. Or even choosing better names for them.

Rule Number 4

- Minimal Classes And Methods

The objective is to keep the system compact at its highest level. So we need minimal number of not lengthy classes and methods. Just remember that this rule holds the lowest priority compared to the other three. Meaning, although it's important to have minimal classes and methods, but it's more important to have covering tests, duplication deletion and expressive code.