Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 10: Classes

Class Organization

Each language has its own standard structure for the classes. In Java (and a few other languages), we write a class in this order:

- Public static final variables
- Private static variables
- Private field(properties)
- Public methods
- Private methods which are called inside the upper public method

Encapsulation

Tests dominate the code. If a test inside the package needs to call a method or get a filed, that method or field should be declared as **protected** or **default** to be accessible from inside the package. Yet, decreasing the encapsulation should be the last approach for accessing the fields. We do our best to keep the fields **private**.

Minimal Classes

The first rule of class is that it should be small. The second rule of class is that it should be smaller. Just like the methods, being minimal is the primary principle for designing the classes.

Class Name

A class's name should clearly reflect its responsibilities and purpose. The chosen name can even serve as a prior indicator of its potential size. If we couldn't find a proper short name for the class, it is most likely going to be large. An ambiguous class name informs us that this class has multiple responsibilities. For instance, a class containing **Processor**, **Manager** or **Super**.

Furthermore, challenge yourself to write a concise 25-word description of the class, omitting if, and, or but. This ensures a clear and focused class definition.

Single Responsibility Principle (SRP)

A class should have one and only one reason to change or responsibility. This is one of most important principles of object oriented design and S.O.L.I.D principles.

Designing a software product that just works and a clean software are two different dos. Unfortunately, we mostly focus on performance of our code, and pay no attention to its organization. The fact is, that we seek systems that are made of too many small classes; not a few large classes. Every small class encapsulates a single responsibility and has only one reason for changing. It cooperates with different parts of the system.

Class Cohesion

Classes should have the minimal number of instance variables. Every method manipulates a number of variables. Generally, the greater amount of this number implies more cohesion between this method and its class.

Though achieving the highest cohesion is challenging, it's beneficial to pursue greater cohesion. High cohesion leads to a more seamless class structure, where methods and variables harmonize to form a complete logical unit.

Begin by breaking methods into smaller components. Some of these smaller methods share mutual logic and pertain to a single concept. Through refactoring, these methods can be moved to a separate class, potentially yielding numerous smaller classes in place of a few large ones.

Tips Around Refactoring

Using descriptive names, breaking methods down to smaller ones, moving methods into multiple classes, calling more methods and declaring more classes and using spaces and formatting techniques will probably make the code longer. But it is totally worth it. Because now we have a clean and organized software, easy to modify and understand.

Organization For Modification

Every change in software triggers a chain reaction, influencing subsequent changes. Since change is inevitable in software development, it's essential to structure the system in a way that minimizes its impact. An organized structure facilitates the addition of new features and problem-solving with minimal effort.

In an ideal system, adding a new feature is expanding the existing system, not modifying the current codebase. This approach streamlines development and helps maintain software stability.

Isolation From Changes

As needs evolve, code must adapt. Minimizing tight couplings between classes aligns with the **Dependency Inversion Principle (DIP)**. This principle advocates that classes should rely on abstractions rather than concrete implementations. This practice enhances flexibility, making it easier to accommodate changes in the system while maintaining stability and reusability.

Summary

- There is conceptual continuity between variables, methods and fields in a clean class. If a method is too large, it should be divided into multiple small methods, each one responsible for one task. Some of these small methods interact with some objects from a specific class. They should be moved to the class.
- It is very usual to add new features or fix bugs in a software. The legacy code has to change. Within it, lies a hidden risk. The change on one module would interrupt the functionality of others. To avoid this issue, we use interfaces or abstract classes, and place the new codes inside a new class that inherits or implements the parent class or interface.