Mohammadmahdi Farrokhy

Clean Code:

A Handbook of Agile Software Craftsmanship

Chapter 08: Class Organization

Class Organization

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

1) Public static final variables
2) Public static variables
3) Public variables
4) Private static variables
5) Private fields
6) Public methods
7) Private methods which are called by public methods

Encapsulation

Tests dominate the code. If a test inside the package needs to call a method or access a field, that method or field should be declared as protected to be accessible all over the package. Yet, decreasing the encapsulation level should be the last thing to consider.

Minimal Classes

The first rule for classes is that they should be small as possible. Just like the methods, being minimal is the primary principle for designing the classes.

Class Name

- Name of a class 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** or **Manager**.

Single Responsibility Principle (SRP)

A class should have one and only one reason to change.

This is one of the most important principles of object oriented design and S.O.L.I.D. One way to comply with this principle is to write too many small classes, instead of a few large classes. Every small class isolates a single responsibility.

- Signs of violating SRP
  - Too many instance variables in a class
  - Too many public methods with implicit interface
  - Methods using different instances- Private methods that do extra work.

- Solution

Use collaborator class that handles minor tasks that are done in the main class.

Class Cohesion

There should be a conceptual relation between fields and methods of a class with its purpose. There are some rules to achieve this:

- Classes should have the minimum number of instance variables (fields).
- Break large methods into smaller ones.
- Move the methods that share a mutual logic to another class.

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 design a structure for the system that minimizes the change. In an ideal system, adding a new feature means expanding the existing system, not modifying the current codebase. This approach streamlines development and helps maintain software stability. To isolate the changes, we should use ***Dependency Inversion Principle (DIP)*** from ***S.O.L.I.D***.

Dependency Inversion Principle (DIP)

Dependency should be defined on abstraction, not concretion. We should not be aware of implementation details, and this lack of awareness can help us in replacement stage. Client class does not determine the required type, it only determines the required rule and system will provide the specific type.

- Signs of violating DIP
  - High level class depends on low level class.
  - **Vendor Lock-In:** A package is developed using a framework and can not be used with other technologies.

- Solution

Adapter design pattern.

Summary

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