[L9P3]

Gems of wisdom: software development principles

Principles

This handout covers some general Software engineering 'principles'.

Separation of concerns

In most introductory programming courses, students learn about the idea of *modularity*, which is to split a program into smaller, more manageable modules. This is actually the first step towards a more general principle known as *separation of concerns*. The term separation of concerns was coined by Edsger W. Dijkstra. A "concern" can be taken as a *single feature* or a single functionality. This principle basically states that a program should be separated into modules, each tackling distinct concerns. This reduces functional overlaps and also limits the ripple effect when changes are introduced to a specific part of the system.

This principle can be applied at the class level, as well as on higher levels. For example, the n-tier architecture utilizes this principle. Each layer in the architecture has a well-defined functionality that has no functional overlap with each other. Clearly, a correct application of this principle should lead to higher cohesion and lower coupling.

Law of Demeter

Another famous principle goes by the name *Law of Demeter* (two other names: *Principle of Least Knowledge*; *Don't talk to strangers*). This principle aims to lower coupling by restricting the interaction between objects in the following ways:

- An object should have limited knowledge of another object.
- An object should only interact with objects that are closely related to it.

In other words, *'only talk to your immediate friends and don't talk to strangers'*. More concretely, a method m of an object O should invoke only the methods of the following kinds of objects:

- The object O itself
- Objects passed as parameters of m
- Objects created/instantiated in m
- Objects from the direct association of O

For example, the following code fragment violates LoD due to the reason: while b is a 'friend' of foo (because it receives it as a parameter), g is a 'friend of a friend' (which should be considered a 'stranger'), and g.doSomething() is analogous to 'talking to a stranger'.

```
void foo(Bar b){
       Goo g = b.getGoo();
       g.doSomething();
}
```

Limiting the interaction to a closely related group of classes aims to reduce coupling. In the example able, foo is already coupled to Bar. Upholding LoD avoids foo being coupled to Goo.

An analogy for LoD can be drawn from Facebook. If Facebook followed LoD, you would not be allowed to see posts of friends of friends, unless they are your friends as well. If Jake is your friend and Adam is Jake's friend, you should not be allowed to see Adam's posts unless Adam is a friend of yours as well.

SOLID principles

Five OOP principles covered in other handouts are popularly known as SOLID principles:

- ***Single Responsibility Principle(SRP)*** - Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. In other words, a class should be highly cohesive. This was proposed by Robert C. Martin. This is a more concrete application of the Separation of Concerns Principle.
- ***Open-Closed Principle (OCP)*** – A software module should be open for extension but should be closed for modification. This was proposed by Bertrand Meyer.
- ***Liskov Substitution Principle*** – if a program module is using a super class, then the reference to the super class can be replaced with a sub class without affecting the functionality of the program module. This was proposed by Barbara Liskov.
- ***Interface Segregation Principle*** – No client should be forced to depend on methods it does not use.
- ***Dependency Inversion Principle (DIP)***. High-level modules should not depend on low-level modules. Both should depend on abstractions.

Other principles

Some other principles:

- ***Brooks' law*** – Adding people to a late project will make it later. This is because the additional communication overhead will outweigh the benefit of adding extra manpower. This principle was proposed by Fred Brooks (author of *The Mythical Man-Month*)
- ***The later you find a bug, the more costly it is to fix***. Covered in a previous handout.
- ***Good, cheap, fast – select any two***. For example, good software developed within a very short time will not come cheap.

As you can see, different principles have varying degrees of formality – rules, opinions, rules of thumb, observations, and axioms. Hence, their applicability is wider with correspondingly greater overlap as compared to patterns.

Worked examples

[Q1]

Explain the Law of Demeter using code examples. You are to make up your own code examples. Take Minesweeper as the basis for your code examples.

[A1]

Let us take the Logic class as an example. Assume that it has the following operation.

setMinefield(Minefiled mf):void

Consider the following that can happen inside this operation.

mf.init(); //this does not violate LoD since LoD allows calling operations of parameters received.

mf.getCell(1,3).clear(); //this violates LoD since Logic is handling Cell objects deep inside Minefield. Instead, it should be mf.clearCellAt(1,3);

timer.start(); //this does not violate LoD since timer appears to be an internal component (i.e. a variable) of Logic itself.

Cell c = new Cell(); c.init(); // this does not violate LoD since c was created inside the operation.

[Q2]

Do these principles apply to your project? If yes, briefly explain how.

i. Brooks' Law.

ii. Good, cheap, fast –select any two.

iii. The later you find a bug, the more costly it is to fix.

[A2]

One of the important learning points here is that it is easy to dismiss a principle as 'not applicable'. However, a second look often reveals a way we can relate them to situations that look 'unrelated' at the beginning.

i. Brooks' Law:
Yes. Adding a new student to a project team can result in a slow-down of the project for a short period. This is because the new member needs time to learn the project and existing members will have to spend time helping the new guy get up to speed. If the project is already behind schedule and near a deadline, this could delay the delivery even further.

ii. Good, cheap, fast –select any two:
Yes. While our project does not involve payments, there is still a hidden price to pay. For example, assume we need the product to be good but finish the project within one week. During that week, the time invested in the project will be very heavy and your other modules will suffer as a result (i.e. you will pay a heavy price in terms of falling behind in other modules). If there is no such cost, everyone can do the project in the last week and still have a good project, right?

iii. The later you find a bug, the more costly it is to fix:
Yes. Imagine you found a bug just before the submission deadline. Not only you have to find the bug and fix it, it could mean redoing the video, modifying the user guide, re-uploading files, not forgetting the additional stress incurred on all other members and the possibility of penalties for overrunning a deadline. The matter could have cost much less if you were able to find the bug and have it fixed before the code was integrated.

[Q3]

Find out, and explain in your own words (in 2-3 sentences), what is meant by the 'second system effect' (attributed to Prof Fred Brooks). Do you think it is applicable to your project?

[A3]

The *second system effect* refers to the tendency to follow a relatively small, elegant, and successful system with an over-engineered, feature-laden, unwieldy successor (source: Wikipedia).

We should keep the above in mind when we do V0.2. If we are not careful, our relative success in V0.1 could mislead us to create an unwieldy feature-laden product at V0.2. Taking this notion further, your success in the CS2103 project can even mislead you in coming up with a failed product in your next project module.

--- End of document ---