[Handout for L9P2]
Gems of wisdom: software development principles

Principles

This handout covers some general Software engineering 'principles'. Note that the work we've done so far in the module implicitly followed some of those principles without mentioning them by name. The objective here is to let you know them by name and also to introduce you to other principles that you can apply in future work.

Separation of concerns

In most introductory programming courses, students learn about the idea of *modularity*, which is to split program into smaller and manageable modules. This is actually the first step towards a more general principle known as *separation of concerns*. 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 which have distinct concerns. This reduces the overlapping of functionality and also limits the ripple effect when changes are introduced to a specific part of the system.

Object oriented languages provide language features to achieve this principle. By providing *encapsulation* features like class (in Java or C++), OOP allows binding of data with its associated operation. Access and modification of data are then limited to the provided operations. With *information hiding* aspect of encapsulation, data are further isolated from the "outside world". Utilizing these features, a class can have a well defined role.

This principle can also be applied on a higher conceptual level than individual classes. For example, the layered architecture utilized the same principle. Each layer in the architecture has a well defined functionality that has no overlapping with each other. As we have seen, a correct application of this principle should lead to higher cohesion and lower coupling.

Open-closed principle

Although we can strive to isolate the functionalities of a software system into modules, there is no way we can remove interaction between modules. When modules interact with each other, coupling naturally increases. Consequently, it is harder to localize any changes to the software system. Bertrand Meyer proposed a guiding principle in 1988 to alleviate this problem. That principle, known as *open-closed principle*, states that "A module [component] should be *open* for extension but *closed* for modification". *Extension* here means expanding the functionalities of a module, while *modification* refers to changes in existing functionalities. A module should be available (open) to extension, which enhance its capability. However, at the same time, it is desirable to freeze (close) the module, so that users of the module will not be affected by future changes.

In object-oriented programming, these two seemingly opposite requirements can be achieved in various ways. Generally, we should try to separate the *specification (interface)* of a module from its *implementation*. Specification of a module is the set of operations and their promised behaviors. User of a module should rely only on the specification of the module, without knowing or caring for the actual implementation. With the separation of specification and implementation, as long as an implementation respects the stated specification, it can be extended or modified without affecting its clients.

In Java, this can be achieved by specifying an interface. Users of the interface are protected from changes of the actual classes that *implement* the interface. For C++, this can be achieved by separating the class prototype (specification) from the actual implementation code.

Consider the minesweeper example, we need to record the current high score for a player. In C++, the component MSLogic can supply the following method prototype:

```
class MSLogic
{
.....
    void AddHighScore( string username, int score );
};
```

The specified methods can be implemented in many ways, for example:

```
void MSLogic::AddHighScore(string username, int score )
{
        // HighScore is kept in an array internally
        //place new score at the end of array
        _highScoreArray[ _highScoreIndex ].username = username;
        _highScoreArray[ _highScoreIndex ].score = score;

        //increase the high score array index
        _highScoreIndex++;
}
```

The same method can be implemented differently:

```
void MSLogic::AddHighScore(string username, int score )
{
        // HighScore is kept in tailed linked list internally

        //place high score at the end of list
        newScore = new ScoreNode;
        newScore->username = username;
        newScore->score = score;
        newScore->next = NULL;

        //update the pointers
        tail->next = newScore;
        tail = newScore;

}
```

The important observation here is that the different implementations will not affect the **user** of the method as the method signature remains consistent.

Law of Demeter

Another famous solution to the problem of coupling is provided by *Law of Demeter* (also known as the *Principle of Least Knowledge*) first proposed at Northeastern University in 1987 by Ian Holland. 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.

An interesting analogy to human interaction can be stated as *"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, if O receives an object N by calling a method in M, O should not call methods of N since a 'friend of a friend' is considered a 'stranger'. By limiting the interaction to closely related group of classes, impact of modification introduced to a class can be more contained than otherwise.

Other principles

Some other principles:

- *Liscov substitutability principle* – Where an instance of a super-type is expected, any subtype is also acceptable.
- *Brooks law* – Adding people to a late project will make it later. This is because of the additional communication overhead will outweigh the benefit of extra manpower.
- *The later you find a bug, the more costly it is to fix.*
- *Good, cheap, fast – select any two*. E.g. If you want good software within a very short time, it will not be cheap.

As you can see, Principles have varying degree of formality – rules, opinions, rules of thumb, observations, and axioms – and their applicability is wider and overlapping 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. Let us assume it has the following operation.

setMinefield(Minefiled mf):void

Consider following things 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 vilolate 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]

(a) 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.

(b) 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?

[A2]

(a) 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' to begin with.

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 caused to all four members and possibility of penalties for overrunning a deadline. The matter could have cost much less if you found the bug and fixed it even before the code was integrated.

(b)

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.

---End of Document---