SOLID principles

- Single Responsibility Principle

```
// What: each method or class should have only on responsibility

// Why: if one method or class needs to be updated, change the code of one functionality will
//      probably affect the code of another functionality.

// how:
// seperate responsibilities in a method to different methods
// each class should also hold single responsibility
Book book = new Book(author, pageNum)
 book.print()
 book.save()

 // two methods above should be extracted to an interface
 // becuase even book cannot be printed or saved, they are still books
 // and something else can also be printed and saved, note just book

 // we should favor the composition over inheritance in OOP
 // therefore we can extract the logic of print and save to
 // another class
 class BookPersistence {
     public void save(Book book) {}
     public String print(Book book) {}
 }

 // so we can compose BookPersistence class with Book class, when logic of save and print changed,
 // we only need to update BookPersistence class
```

- The Open Closed Principle

```
// what: open you extendsion closed for modification. When adding new functionality, we want to min
// the modification to exisiting code

// why: during the development cycle, it will probably cause error if we modify the existed code
//      it is bad since existed code has been unit tested. it's likely to cause refactoring for us

// how : 1. template pattern
//       2. strategy pattern
//       3. other patterns

 // template pattern:
     // define a parent abstract class as template
     // the repeated code should be extracted to a normal method
     // the non-repeated code should be abstrct and force child class to overwrite it
     // example: how to fry chinese food
     // advantage:
         // reuseability
         // scalibility
         // inverse control????
     // disadvantage:
         // too many class


     // strategy pattern:
        //  encapsulate different strategies
        //  client can call strategy and strategies are interchangeable
    //  example: salesman for festival
    // advantage:
        // strategies are interchangeable
        // scalibility: match open close principle
    // disadvantage:
        // will produce many type and object
```

- Liskov substitution principle

```
// what: we usually create class hierachy during development cycle, it would be great
// if new derived class can work as well without overwriting or overloading functionality of classes

// why: To avoid side effect caused by inheritance, we don't want to damage the hierachy system

// how:
    // 1. child class don't overwrite the method implemented by parent class
    // 2. child class can have it's own method
    // 3. when overloading parent class's method, parameter must be less strict than parent class's
     // 4. when implement parent class's abstract method, the output(return type) should be more str
     // pattern?
```

- interface segregation principle

```
// what: seperate method in interface to prevent fat interface
// why: client should not depend on method it does not use. So we can keep
//      a system decoupled and thus easier to refactor, change, and redeploy
// How: create a new interface and put method in it
```

- dependency inversion principle:

```
// what: high level module should not depend on low level module directly. There should be a
//       abstract layer between them.

// why: If low level module is updated, then we need to rewrite the low level module again.

// How: we can use passing by interface. Both high level and low level modules denpend on interface
```