SOLID Principles

1. Single Responsibility Principle (SRP)
   1. A class should have one primary job. One reason to change.
   2. Separation of concerns.
   3. Example:
      1. You have a class called journal. You have a list of entries in this journal.
      2. Adding and removing an entry is this class’s concern
      3. but when you are trying to get a url, take all these entries from this url or save these entries to a file and so on.
      4. This is considered as a new concern since it is about persistence. So you need to create another class for JournalPersistence.
2. Open Closed Principle (OCP)
   1. Open for extension but closed for modification.
   2. Problem. I have a product with a color and a size. Boss wants to filter the products by color. You write a class for filtering logic and add a filterByColor function. But later the boss wants to filter by size so you copy and paste the code with minimal change. So you modified your class which we don't want. How can we solve this problem?
   3. Open closed principle with Specification pattern.
      1. Code example written.

interface Specification<T>{

boolean isSatisfied(T item);

}

static class ColorSpecification implements Specification<Product>{

Color color;

ColorSpecification(Color color){

this.color = color;

}

@Override

public boolean isSatisfied(Product item) {

return item.color == this.color;

}

}

public static void main(String[] args) {

Product product = new Product(Color.*GREEN*, Size.*SMALL*);

Product product2 = new Product(Color.*BLUE*, Size.*MEDIUM*);

Product product3 = new Product(Color.*RED*, Size.*LARGE*);

List<Product> products = List.*of*(product, product2, product3);

BetterFilter betterFilter = new BetterFilter();

betterFilter.filter(products, new ColorSpecification(Color.*BLUE*)).forEach(System.*out*::println);

//*TODO with this way you can add multiple filter functionalities by simply adding a new class without touching other classes*

}



1. Liskov substitution Principle (LSP)
   1. You should be able to substitute a subclass with a base class.
   2. Example given with rectangle and square classes. For a method that takes a rectangle, when we give a square class which extends the Rectangle class, it can give unexpected results.
   3. To solve the problem we used Factory pattern which will be seen in detail in later classes.
   4. Example code written.
2. Interface Segregation Principle (LSP)
   1. It is a recommendation on how to split interfaces into smaller interfaces.
   2. YAGNI = You Aint Going to Need It.
   3. The point is to put minimum code to interfaces so the person who is going to implement the interface won't implement a method that they won't use.
   4. Example
      1. You have an interface that has following methods:
         1. Print
         2. Scan
         3. Fax
      2. When you need all of them you can simply implement this interface but when for example you have a machine that can only print you don't need other methods to be implemented.
      3. So the best way to go about this problem is to divide all methods into separate interfaces and they can be implemented if needed.
3. Dependency Inversion Principle
   1. Two main points:
      1. High level modules should not depend on low level modules. Both should depend on abstractions. (Abstraction is either abstract class or interface)
      2. Abstractions should not depend on details. Details should depend on abstractions.
   2. We will mostly focus on the first one.
   3. Low level module → example: has storage
   4. High level module → has some operations not related to storage and so on.