Software Design Principles

Software design principles represent a set of guidelines that helps us to avoid having a bad design. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices". According to Robert Martin there are 3 important characteristics of a bad design that should be avoided:

* Rigidity - It is hard to change because every change affects too many other parts of the system.
* Fragility - When you make a change, unexpected parts of the system break.
* Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

There are 5 desing principle which we always needs to remember when we program for applications.

1. Open Close Principle(OCP)

2. Dependency Inversion Principle (DIP)

3. Interface Segration Principle (ISP)

4. Single Responsibility Principle (SRP)

5. Lisko Substitution Principle (LSP)

**1.Open Close Principle**

"Software entities like classes, modules and functions should be **open for extension but closed for modifications**."

**Motivation**

A clever application design and the code writing part should take care of the frequent changes that are done during the development and the maintaining phase of an application. Usually, many changes are involved when a new functionality is added to an application. Those changes in the existing code should be minimized, since it's assumed that the existing code is already unit tested and changes in already written code might affect the existing functionality in an unwanted manner.

The **Open Close Principle** states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

Example

Bellow is an example which violates the Open Close Principle. It implements a graphic editor which handles the drawing of different shapes. It's obviously that it does not follow the Open Close Principle since the GraphicEditor class has to be modified for every new shape class that has to be added. There are several disadvantages:

for each new shape added the unit testing of the GraphicEditor should be redone.

when a new type of shape is added the time for adding it will be high since the developer who add it should understand the logic of the GraphicEditor.

adding a new shape might affect the existing functionality in an undesired way, even if the new shape works perfectly

In order to have more dramatic effect, just imagine that the Graphic Editor is a big class, with a lot of functionality inside, written and changed by many developers, while the shape might be a class implemented only by one developer. In this case it would be great improvement to allow the adding of a new shape without changing the GraphicEditor class.

Open Close Principle(OCP) - bad



// Open-Close Principle - Bad example

class GraphicEditor {

public void drawShape(Shape s) {

if (s.m\_type==1)

drawRectangle(s);

else if (s.m\_type==2)

drawCircle(s);

}

public void drawCircle(Circle r) {....}

public void drawRectangle(Rectangle r) {....}

}

class Shape {

int m\_type;

}

class Rectangle extends Shape {

Rectangle() {

super.m\_type=1;

}

}

class Circle extends Shape {

Circle() {

super.m\_type=2;

}

}

Bellow is a example which supports the Open Close Principle. In the new design we use abstract draw() method in GraphicEditor for drawing objects, while moving the implementation in the concrete shape objects. Using the Open Close Principle the problems from the previous design are avoided, because GraphicEditor is not changed when a new shape class is added:

* no unit testing required.
* no need to understand the sourcecode from GraphicEditor.

since the drawing code is moved to the concrete shape classes, it's a reduced risk to affect old functionallity when new functionallity is added.



Open Close Principle(OCP) - good

// Open-Close Principle - Good example

class GraphicEditor {

public void drawShape(Shape s) {

s.draw();

}

}

class Shape {

abstract void draw();

}

class Rectangle extends Shape {

public void draw() {

// draw the rectangle

}

}

**Conclusion**

Like every principle OCP is only a principle. Making a flexible design involves additional time and effort spent for it and it introduce new level of abstraction increasing the complexity of the code. So this principle should be applied in those area which are most likely to be changed.

There are many design patterns that help us to extend code without changing it. For instance the **Decorator pattern** help us to follow Open Close principle. Also the **Factory Method** or the **Observer pattern** might be used to design an application easy to change with minimum changes in the existing code.

**2. Dependency Inversion Principle**

**Motivation**

When we design software applications we can consider the low level classes the classes which implement basic and primary operations(disk access, network protocols,...) and high level classes the classes which encapsulate complex logic(business flows, ...). The last ones rely on the low level classes. A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes. Since high level classes are defined in terms of others this seems the logical way to do it. But this is not a flexible design. What happens if we need to replace a low level class?

Let's take the classical example of a copy module which reads characters from the keyboard and writes them to the printer device. The high level class containing the logic is the Copy class. The low level classes are KeyboardReader and PrinterWriter.

In a bad design the high level class uses directly and depends heavily on the low level classes. In such a case if we want to change the design to direct the output to a new FileWriter class we have to make changes in the Copy class. (Let's assume that it is a very complex class, with a lot of logic and really hard to test).

In order to avoid such problems we can introduce an abstraction layer between high level classes and low level classes. Since the high level modules contain the complex logic they should not depend on the low level modules so the new abstraction layer should not be created based on low level modules. Low level modules are to be created based on the abstraction layer.

According to this principle the way of designing a class structure is to start from high level modules to the low level modules:

**High Level Classes --> Abstraction Layer --> Low Level Classes**

**Intent**

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.

Example

Below is an example which violates the Dependency Inversion Principle. We have the manager class which is a high level class, and the low level class called Worker. We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers. We created a new class SuperWorker for this.

Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:

we have to change the Manager class (remember it is a complex one and this will involve time and effort to make the changes).

some of the current functionality from the manager class might be affected.

the unit testing should be redone.

All those problems could take a lot of time to be solved and they might induce new errors in the old functionlity. The situation would be different if the application had been designed following the Dependency Inversion Principle. It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface. When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it. No additional changes in the existing classes.

// Dependency Inversion Principle - Bad example

class Worker {

public void work() {

// ....working

}

}

class Manager {

Worker worker;

public void setWorker(Worker w) {

worker = w;

}

public void manage() {

worker.work();

}

}

class SuperWorker {

public void work() {

//.... working much more

}

}

Below is the code which supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved(considering there is no change in the high level logic):

Manager class doesn't require changes when adding SuperWorkers.

Minimized risk to affect old functionality present in Manager class since we don't change it.

No need to redo the unit testing for Manager class.

// Dependency Inversion Principle - Good example

interface IWorker {

public void work();

}

class Worker implements IWorker{

public void work() {

// ....working

}

}

class SuperWorker implements IWorker{

public void work() {

//.... working much more

}

}

class Manager {

IWorker worker;

public void setWorker(IWorker w) {

worker = w;

}

public void manage() {

worker.work();

}

}

**Conclusion**

When this principle is applied it means the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In this case instantiation of new low level objects inside the high level classes(if necessary) can not be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype.

The Template Design Pattern is an example where the DIP principle is applied.

Of course, using this principle implies an increased effort, will result in more classes and interfaces to maintain, in a few words in more complex code, but more flexible. This principle should not be applied blindly for every class or every module. **If we have a class functionality that is more likely to remain unchanged in the future there is not need to apply this principle.**

**3.Interface Segregation Principle (ISP)**

**Motivation**

When we design an application we should take care how we are going to make abstract a module which contains several submodules. Considering the module implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application adding another module that contains only some of the submodules of the original system, we are forced to implement the full interface and to write some dummy methods. Such an interface is named fat interface or polluted interface. Having an interface pollution is not a good solution and might induce inappropriate behavior in the system.

The Interface Segregation Principle states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

**Intent**

Clients should not be forced to depend upon interfaces that they don't use.

**Example**

Below is an example which violates the Interface Segregation Principle. We have a Manager class which represent the person which manages the workers. And we have 2 types of workers some average and some very efficient workers. Both types of workers works and they need a daily launch break to eat. But now some robots came in the company they work as well , but they don't eat so they don't need a launch break. One on side the new Robot class need to implement the IWorker interface because robots works. On the other side, the don't have to implement it because they don't eat.

This is why in this case the IWorker is considered a polluted interface.

If we keep the present design, the new Robot class is forced to implement the eat method. We can write a dummy class which does nothing(let's say a launch break of 1 second daily), and can have undesired effects in the application(For example the reports seen by managers will report more lunches taken than the number of people).

According to the Interface Segregation Principle, a flexible design will not have polluted interfaces. In our case the IWorker interface should be split in 2 different interfaces.

// interface segregation principle - bad example

interface IWorker {

public void work();

public void eat();

}

class Worker implements IWorker{

public void work() {

// ....working

}

public void eat() {

// ...... eating in launch break

}

}

class SuperWorker implements IWorker{

public void work() {

//.... working much more

}

public void eat() {

//.... eating in launch break

}

}

class Manager {

IWorker worker;

public void setWorker(IWorker w) {

worker=w;

}

public void manage() {

worker.work();

}

}

Following it's the code supporting the Interface Segregation Principle. By splitting the IWorker interface in 2 different interfaces the new Robot class is no longer forced to implement the eat method. Also if we need another functionality for the robot like recharging we create another interface IRechargeble with a method recharge.

// interface segregation principle - good example

interface IWorker extends Feedable, Workable {

}

interface IWorkable {

public void work();

}

interface IFeedable{

public void eat();

}

class Worker implements IWorkable, IFeedable{

public void work() {

// ....working

}

public void eat() {

//.... eating in launch break

}

}

class Robot implements IWorkable{

public void work() {

// ....working

}

}

class SuperWorker implements IWorkable, IFeedable{

public void work() {

//.... working much more

}

public void eat() {

//.... eating in launch break

}

}

class Manager {

Workable worker;

public void setWorker(Workable w) {

worker=w;

}

public void manage() {

worker.work();

}

}

**Conclusion**

If the design is already done **fat interfaces can be segregated** using the **Adapter pattern**.

Like every principle Interface Segregation Principle is one principle which require additional time and effort spent to apply it during the design time and increase the complexity of code. But it produce a flexible design. If we are going to apply it more than is necessary it will result a code containing a lot of interfaces with single methods, so applying should be done based on experience and common sense in identifying the areas where extension of code are more likely to happens in the future.

**4. Single Responsibility Principle**

**Motivation**

In this context, a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and if in the future we need to make one change we are going to make it in the class which handles it. When we need to make a change in a class having more responsibilities the change might affect the other functionality related to the other responsibility of the class.

The Single Responsibility Principle is a simple and intuitive principle, but in practice it is sometimes hard to get it right.

**Intent**

A class should have only one reason to change.

**Example**

Let's assume we need an object to keep an email message. We are going to use the IEmail interface from the below sample. At the first sight everything looks just fine. At a closer look we can see that our IEmail interface and Email class have 2 responsibilities (reasons to change). One would be the use of the class in some email protocols such as pop3 or imap. If other protocols must be supported the objects should be serialized in another manner and code should be added to support new protocols. Another one would be for the Content field. Even if content is a string maybe we want in the future to support HTML or other formats.

If we keep only one class, each change for a responsibility might affect the other one:

Adding a new protocol will create the need to add code for parsing and serializing the content for each type of field.

Adding a new content type (like html) make us to add code for each protocol implemented.

// single responsibility principle - bad example

interface IEmail {

public void setSender(String sender);

public void setReceiver(String receiver);

public void setContent(String content);

}

class Email implements IEmail {

public void setSender(String sender) {// set sender; }

public void setReceiver(String receiver) {// set receiver; }

public void setContent(String content) {// set content; }

}

We can create a new interface and class called IContent and Content to split the responsibilities. Having only one responsibility for each class give us a more flexible design:

adding a new protocol causes changes only in the Email class.

adding a new type of content supported causes changes only in Content class.

// single responsibility principle - good example

interface IEmail {

public void setSender(String sender);

public void setReceiver(String receiver);

public void setContent(IContent content);

}

interface Content {

public String getAsString(); // used for serialization

}

class Email implements IEmail {

public void setSender(String sender) {// set sender; }

public void setReceiver(String receiver) {// set receiver; }

public void setContent(IContent content) {// set content; }

}

**Conclusion**

The Single Responsibility Principle represents a good way of identifying classes during the design phase of an application and it reminds you to think of all the ways a class can evolve. A good separation of responsibilities is done only when the full picture of how the application should work is well understand.

**5. Liskov's Substitution Principle(LSP)**

Motivation

All the time we design a program module and we create some class hierarchies. Then we extend some classes creating some derived classes.

We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

Intent

Derived types must be completely substitutable for their base types.

Example

Below is the classic example for which the Likov's Substitution Principle is violated. In the example 2 classes are used: Rectangle and Square. Let's assume that the Rectangle object is used somewhere in the application. We extend the application and add the Square class. The square class is returned by a factory pattern, based on some conditions and we don't know the exact what type of object will be returned. But we know it's a Rectangle. We get the rectangle object, set the width to 5 and height to 10 and get the area. For a rectangle with width 5 and height 10 the area should be 50. Instead the result will be 100

// Violation of Likov's Substitution Principle

class Rectangle

{

protected int m\_width;

protected int m\_height;

public void setWidth(int width){

m\_width = width;

}

public void setHeight(int height){

m\_height = height;

}

public int getWidth(){

return m\_width;

}

public int getHeight(){

return m\_height;

}

public int getArea(){

return m\_width \* m\_height;

}

}

class Square extends Rectangle

{

public void setWidth(int width){

m\_width = width;

m\_height = width;

}

public void setHeight(int height){

m\_width = height;

m\_height = height;

}

}

class LspTest

{

private static Rectangle getNewRectangle()

{

// it can be an object returned by some factory ...

return new Square();

}

public static void main (String args[])

{

Rectangle r = LspTest.getNewRectangle();

r.setWidth(5);

r.setHeight(10);

// user knows that r it's a rectangle.

// It assumes that he's able to set the width and height as for the base class

System.out.println(r.getArea());

// now he's surprised to see that the area is 100 instead of 50.

}

}

**Conclusion**

This principle is just an **extension of the Open Close Principle** and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.