**Open Close Principle**

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.

Intent

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

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

}

}

**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() {

}

}

class Manager {

Worker worker;

public void setWorker(Worker w) {

worker = w;

}

public void manage() {

worker.work();

}

}

class SuperWorker {

public void work() {

}

}

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();

}

}

**Object Composition vs. Inheritance**

Object composition and inheritance are two techniques for reusing functionality in object-oriented systems [7]. Class inheritance, or subclassing, allows a subclass' implementation to be defined in terms of the parent class' implementation. This type of reuse is often called white-box reuse. This term refers to the fact that with inheritance, the parent class implementation is often visible to the subclasses.

Object composition is a different method of reusing functionality. Objects are composed to achieve more complex functionality. This approach requires that the objects have well-defined interfaces since the internals of the objects are unknown. Because objects are treated only as "black boxes," this type of reuse is often called black-box reuse.

Each of these two methods have advantages and disadvantages. The advantage of class inheritance is that it is done statically at compile-time and is easy to use. The disadvantage of class inheritance is that the subclass becomes dependent on the parent class implementation. This makes it harder to reuse the subclass, especially if part of the inherited implementation is no longer desirable. According to [16], "inheritance breaks encapsulation." One way around this problem is to only inherit from abstract classes. Another problem with class inheritance is that the implementation inherited from a parent class cannot be changed at run-time.

In object composition, functionality is acquired dynamically at run-time by objects collecting references to other objects. The advantage of this approach is that implementations can be replaced at run-time. This is possible because objects are accessed only through their interfaces, so one object can be replaced with another just as long as they have the same type. In addition, since each object is defined in terms of object interfaces, there are less implementation dependencies [7]. The disadvantage of object composition is that the behavior of the system may be harder to understand just by looking at the source code. A system using object composition may be very dynamic in nature so it may require running the system to get a deeper understanding of how the different objects cooperate.

In general, object composition should be favored over inheritance. It promotes smaller, more focused classes and smaller inheritance hierarchies. Most designers overuse inheritance, resulting in large inheritance hierarchies that can become hard to deal with. A design based on object composition will have less classes, but more objects. The behavior of such a system depends more on the interrelationships between objects rather then being defined in a particular class [7].

However, inheritance is still necessary. You cannot always get all the necessary functionality by assembling existing components. This is because "the set of components is never quite rich enough in practice" [7]. Inheritance can be used to create new components that can be composed with old components. As a result, the two methods work together.

Due to the flexibility and power of object composition, most design patterns emphasize object composition over inheritance whenever it is possible. Many times, a design pattern shows a clever way of solving a common problem through the use of object composition rather then a standard, less flexible, inheritance based solution.