**Object Oriented Design Principles - Software Design Principles**

Single Responsibility Principle:

**Principle:** A class should have one, and only one, reason to change. This means that a class will do only one job.

Class methods and data will be concerned with one clear purpose. This means [**high cohesion**](https://www.baeldung.com/cs/cohesion-vs-coupling)**,** as well as**robustness, which together reduce errors**. It speaks about more cohesive classes, it means each class should have a focused set of responsibility. It also embraces separation of concern.

**Example:** Suppose there is a requirement to download the file- may be in csv/json/xml format, parse the file and then update the contents into a database or file system.

public class Task

{

public void downloadFile(location)

{

//Download the file

}

public void parseTheFile(file)

{

//Parse the contents of the file- XML/JSON/CSV

}

public void persistTheData(data)

{

//Persist the data to Database or file system.

}

}

It looks good, all in one place easy to understand. But what about the number of times this class has to be updated? What about the reusability of parser code? or download code? Its not good design in terms of reusability of different parts of the code, in terms of cohesiveness.

One way to decompose the Task class is to create different classes for downloading the file- Downloader, for parsing the file- Parser and for persisting to the database or file system.

**Open Close Principle**

Principle : "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be modified without altering its source code.

**What is closed here**?

***A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients****.*

Example:

Suppose you are writing a module to approve personal loans and before doing that you want to validate the personal information, code wise we can depict the situation as:

public class LoanApprover

{

public void approveLoan(PersonalValidator validator)

{

if ( validator.isValid())

{

//Process the loan.

}

}

}

public class PersonalLoanValidator

{

public boolean isValid()

{

//Validation logic

}

}

As you all know the requirements are never the same and now its required to approve vehicle loans, consumer goods loans and what not. So one approach to solve this requirement is to:

public class LoanApprovalHandler

{

public void approvePersonalLoan (PersonalLoanValidator validator)

{

if ( validator.isValid())

{

//Process the loan.

}

}

public void approveVehicleLoan (VehicleLoanValidator validator )

{

if ( validator.isValid())

{

//Process the loan.

}

}

// Method for approving other loans.

}

public class PersonalLoanValidator

{

public boolean isValid()

{

//Validation logic

}

}

public class VehicleLoanValidator

{

public boolean isValid()

{

//Validation logic

}

}

We have edited the existing class to accommodate the new requirements- in the process we ended up changing the name of the existing method and also adding new methods for different types of loan approval. This clearly violates the OCP. Lets try to implement the requirement in a different way

/\*\*

\* Abstract Validator class

\* Extended to add different

\* validators for different loan type

\*/

public abstract class Validator

{

public boolean isValid();

}

/\*\*

\* Personal loan validator

\*/

public class PersonalLoanValidator extends Validator

{

public boolean isValid()

{

//Validation logic.

}

}

/\*

\* Similarly any new type of validation can

\* be accommodated by creating a new subclass

\* of Validator

\*/

Now using the above validators we can write a LoanApprovalHandler to use the Validator abstraction.

public class LoanApprovalHandler

{

public void approveLoan(Validator validator)

{

if ( validator.isValid())

{

//Process the loan.

}

}

}

So to accommodate any type of loan validators we would just have create a subclass of Validator and then pass it to the approveLoan method. That way the class is CLOSED for modification but OPEN for extension.

**Liskov Substitution Principle**

**Principle :** Derived types must be completely substitutable for their base types. It means we must make sure that new derived classes are extending the base

classes without changing the behaviour.

**Example :**

Let us see the violation of this principle.

class Bird {

public void fly(){}

public void eat(){}

}

class Crow extends Bird {}

class Ostrich extends Bird{

fly(){

throw new UnsupportedOperationException();

}

}

public BirdTest{

public static void main(String[] args){

List<Bird> birdList = new ArrayList<Bird>();

birdList.add(new Bird());

birdList.add(new Crow());

birdList.add(new Ostrich());

letTheBirdsFly ( birdList );

}

static void letTheBirdsFly ( List<Bird> birdList ){

for ( Bird b : birdList ) {

b.fly();

}

}

}

As soon as an Ostrich instance is passed, it blows up!!! Here the sub type is not replaceable for the super type. How do we fix such issues ?

By using factoring. Sometimes factoring out the common features into a separate class can help in creating a hierarchy that confirms to LSP.

In the above scenario we can factor out the fly feature into- Flight and NonFlight birds.

class Bird{

public void eat(){}

}

class FlyingBird extends Bird{

public void fly()()

}

class NonFlying extends Bird{}

**Interface Segregation Principle(ISP)**

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

**Example :**

If we create an interface called Worker and a method called lunch break, all workers have to implement it. What if the worker is a robot.

Let us consider another example.

Consider there is one interface Travel

Travel has two methods

travelbyBus()

travelByTrain()

Now there is one class Passenger . Passenger will always travel by bus .

if Passenger class implements Travel interface

Passenger implements Passenger {

travelbyBus(){

}

travelByTrain(){

}

}

It is forced to implement both the methods travelByBus() and travelByTrain() , thought Passenger has nothing to do with travelByTrain().

Now lets read the principal again

Interface Segregation Principle stats that, a client should not implement an interface, if it doesn't use that.

Client Passenger should not be forced to implement travelByTrain() method as it never needs it.

Here comes the Interface designing best practice .

An interface design should be very thoughtful. It should always take in consideration all its possible clients and should always make interface design client friendly. So could that be achieved?

Instead of creating one interface for both methods , two separate interfaces could be designed like :

BusTravel{

travelByBus();

}

TrainTravel{

travelByTrain();

}

Passenger class can implement BusTravel interface and will not be forced to implement anymore method.

**Dependency inversion principle**

**Principle :** High-level modules should not depend on low-level modules. Both should depend on abstractions.

the dependency structure of a well designed object oriented program is “inverted” with respect to the dependency structure that normally results from traditional procedural methods. Note that [Dependency Injection (DI)](http://en.wikipedia.org/wiki/Dependency_injection) is ONE of the possible implementations to achieve the Dependency Inversion Principle (DIP) - [the "D" in SOLID design principles](http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod), so DI and DIP are *not* entirely interchangeable. Bridge design pattern is good example.

**Example :**

Let us consider a bad example

// 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();

}

}