**OO Design principles (SOLID overview)- What are they and why we need them**

We write or modify code but offenly neglect some important guidelines which results in a complete bad design. It takes just a few minutes to go through these principles and every time we write code – **its a good design**.

**Object Oriented design principles** are the guidelines that helps us to avoid having a bad design. Robert Martin gave these guidelines in his book “Agile software develpment, design patterns and practices”. Martin says, there are mainly five reasons for having a bad design in a software-

1. **Rigidity** – It is hard to change because every change affects too many other parts of the system.

2. **Fragility** – When you make a change, unexpected parts of the system break.

3. **Immobility** – It is hard to reuse in another application because it cannot be disentangled from the current application.

4. **Viscosity** **of Design**- For any software requirement or a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (i.e.they are hacks.) When the design preserving methods are harder to employ than the hacks, then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing.

**5. Viscosity of Environment**- Viscosity of environment comes about when the development environment is slow and inefficient. For example, if compile times are very long, engineers will be tempted to make changes that don’t force large recompiles, even though those changes are not optiimal from a design point of view.

Directly or indirectly, the root cause of all the above reasons is the **improper dependencies among the modules of the software**. So, if we properly  manage the dependencies between the modules -we can get over to all of the above causes. These principles popularly known as **SOLID** would help us in a proper management of all such dependencies – which if we just remember while coding could help us avoid having a bad design.

[**1. Single Responsibility Principle (SRP)**](http://www.gontu.org/solid-single-responsibility-principle/)

there should never be more than one reason for a class to change.

[**2. Open Close principle (OCP)**](http://www.gontu.org/solid-open-close-principle-ocp/)

software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

[**3. Liskov’s Substitution Principle (LSP)**](http://www.gontu.org/liskovs-substitution-principle-lsp/)

functions that use references to base classes must be able to use objects of derived classes without knowing it.

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

many client specific interfaces are better than one general purpose interface.

**5. Dependency inversion Principle (DIP)**

one should “Depend upon Abstractions. Do not depend upon concretions.

## SOLID- Single Responsibility Principle (SRP)

The Principle states that

**there should not be more than a single reason for a class to change.**

Which means whenever you are modifying a class you should not have more than one reason to modify it. If you have two or more reasons for a class to modify then you are violating the principle. The principle may be stated in a different way, that your class should perform only one responsibility. So, whenever you are making any modifications to it you would always have a single reason to do so. It’s the simplest of all SOLID principles but difficult one to implement.

When I say a class should not have more than a single responsibility, **it does not mean that a class should have a single function in it**. A class may perform a single responsibility with one or more functions in it. **Its very important to understand what does ‘reason for change’ or ‘single responsibility’ means. Following examples would explain all about SRP.**

### Explanation:

Suppose, we have to write a class that makes a call to the specified mobile number. Before the class would make a call, the mobile number should get validated. So, if the mobile number is valid, its only then the class should make a call otherwise not. One way to implement this is,

public class MakeCallToMobile

{

public void callMobile(MobileNo mobileNo)

{

if(validateMobileNumber(mobileNo))

{

//code to make a call on the mobile number

}

}

public boolean validateMobileNumber(MobileNo mobileNo)

{

//Verify if the mobile no is valid.

}

}

If we look at the above approach, everything is fine. The **MakeCallToMobile** class is exactly serving the purpose i.e  **callMobile(MobileNo mobile)** would make a call only if its other function **validateMobileNumber(MobileNo mobileNo)** would return true.

**But, the class is a clear violation of SRP**: The class is performing two responsibilities

1. Making a call to the specified mobile number.

2. Validating the mobile number.

**Problems because of such a design:** Because of ever changing requirements in real world, if you need to make a change in the way mobile number is getting validated, the only place where you can modify code is **MakeCallToMobile** class. If we observe the class it has got one more action to do, which is making a call; which is no way related to this change. But, if you make such a change , you would need to again rebuild and retest the  call mobile functionality. Similarly, if you need to make a change to the way class makes a call to the mobile number, you need to again rebuild and retest  the mobile number validation.  So, making a small change in one responsibility would force us to rebuild and retest the other responsibility also.

**You can overcome this problem by following SRP**. You create two classes each performing a single responsibility as shown below.

public class MakeCallToMobile

{

public void callMobile(MobileNo mobileNo)

{

if(ValidationeService.validateMobileNumber(mobileNo))

{

//code to make a call on the mobile number

}

}

}

public class ValidationeService

{

public static boolean validateMobileNumber(MobileNo mobileNo)

{

//Verify if the mobile no is valid.

}

}

In the above design, since both responsibilities reside in different classes, you are relieved of making a change in one without affecting other in any way. Clearly in the above design, you would have only one reason to make a change in any of the classes. For ***MakeCallToMobile class, reason to change would be ‘changing the way it makes a call to mobile number’*** and for ***ValidationService class –  reason to change would be ‘changing the way it performs validation of a mobile number’***.

**Conclusion:** Using **SRP**,

You would achieve **better code readability, reduce testing and maintenance efforts** and consequently strive for **faster development life cycles.**

## SOLID- Open Closed Principle (OCP)

The principle states that,

**software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.**

The principle says that for every new functionality to be added, you should limit yourself for editing the existing modules (classes), rather you should always write a new class for every new functionality by extending the existing classes. This is one of the simplest SOLID principles, easy to understand and implement with great design benefits.

In simple terms, the principle talks about the usage of **Abstract classes** to achieve this. The direct benefits you get by following the principle are-

1. Since, you are not editing the existing classes and for any new functionality to be added, you would add a new class- you do not need to test the already written class. If you would have edited the already written class, you would have to test the new functionality and already existed old functionality in the class. So, following this, **you would reduce huge testing efforts.**

2. **Your code would be more fragile** i.e you can easily accommodate new functionality(ies) in your project.

**Its very important to understand about how you can achieve the principle through the usage of Abstract classes. Following example would explain all about OCP.**

### ****Explanation:****

Let us go through the below example to understand the principle and the underlying benefits: Suppose for a certain requirement you would need to write a class for calculating the fare that a person is charged with. So, you write the following two classes.

public class Bus {

public double CostPerMile;

public double MilesTravelled;

public int NoOfSeatsBooked;

}

public class FareCalculator {

public double Fare(Bus bus) {

double fare = 0;

fare = bus.CostPerMile \* bus.MilesTravelled \* bus.NoOfSeatsBooked;

return fare;

}

}

Everything is fine in the above code. **FareCalculator** class would return the computed fare for Bus properly.

**But the above design is a clear violation of OCP**: Let’s say, the new requirement says that the  **FareCalculator** class should include fare calculation for a new vehicle – Truck also. And, you know that the calculation would differ in both the cases. In case of Bus, the calculation depends on the number of seats booked and in case of Truck, it depends on the Weight of Goods that truck would be carrying.

So, you would modify **FareCalculator** class as follows:

public class FareCalculator {

public double BusFare(Bus bus) {

double fare = 0;

fare = bus.CostPerMile \* bus.MilesTravelled \* bus.NoOfSeatsBooked;

return fare;

}

public double TruckFare(Truck truck) {

double fare = 0;

fare = truck.CostPerMile \* truck.MilesTravelled \* truck.GoodsWeight;

return fare;

}

}

public class Bus {

public double CostPerMile;

public double MilesTravelled;

public int NoOfSeatsBooked;

}

public class Truck {

public double CostPerMile;

public double MilesTravelled;

public int GoodsWeight;

}

Definitely your Code would work fine and delivers the results accurately.

**But, there are numerous design problems with the above modification**:

If the requirement is to compute fare for one more vehicle say- Car etc, you would be making a change in the existing FareCalculator class to implement the new requirement. So, every time you need to make sure that the new function added in the class has not broken any older functinality present already in the class.

So, in the above design, its very clear that the class is **not really closed for modifcation and also not open for extension**. In the real world, the class may not be having just a few lines of code and in actual would be having hundreds or thousands lines of code. So, anything that you are modifying in an existing  class in order to implement a new functionality lead to enormous development and testing efforts. For every new functionality to be added, you need to make sure that nothing has been broken with respect to the functinality already present in the class.

**You can overcome the said problems by following OCP:**

You create a Abstract class **Vehicle** as follows

public abstract class Vehicle

{

public double Fare();

}

And, then write **Bus** and **Truck** class extending the **Vehicle** class and implementing the Fare function as follows:

public class Bus extends Vehicle {

public double CostPerMile;

public double MilesTravelled;

public int NoOfSeatsBooked;

public double Fare() {

return CostPerMile \* MilesTravelled \* NoOfSeatsBooked;

}

}

public class Truck extends Vehicle {

public double CostPerMile;

public double MilesTravelled;

public int GoodsWeight;

public double Fare() {

return CostPerMile \* MilesTravelled \* GoodsWeight;

}

}

Finally, you write **FareCalculator** as follows:

public class FareCalculator {

public double Fare(Vehicle vehicle) {

double fare = 0;

fare = vehicle.Fare();

return fare;

}

}

In the above design, the **FareCalculator** and all other classes are absolutely closed for modifications. With the usage of Abstract class **Vehicle,** now, even if the requirement is to compute fare for a new vehicle, you can easily do it by writing a new class and extending the Vehicle class in it.  By doing so, you would not really need to make modifications in any of the existing classes and you will always extend the existing classes to implement any new functionality.

**Conclusion**, By following **OCP**, you would make your code **more** **fragile** and adds value towards the **faster development life cycles**.

## SOLID- Liskov’s Substitution Principle (LSP)

The principle states that

**functions that use references to base classes must be able to use objects of derived classes without knowing it.**

In simple terms, the principle says that if your code is calling a method residing in a base class than your code must be able to call the same method  if you would replace reference of the base class with a reference of any of its derived classes i.e at any given place in your code, you should be able to replace reference of a base class with a reference of any of its derived classes without  affecting the execution of the program. The principle seems difficult to understand from its definition but is the easiest one among all. The following example would talk all about LSP.

**Benefits of following LSP:**

1. Your class hierarchies would be easy to understand. Any new person in the project would quickly and easily understand it.

2. It would confirm to the perfect class hierarchies which inturns confirm to ease of code maintenance.

3. Preventing subtyping related bugs.

### ****Explanation:****

Let us go through the example to understand LSP in detail: suppose you have written a **Vehicle** class as follows which returns cost of the vehicle and the petrol consumption by the vehicle per Mile.

class Vehicle {

public void vehicleCost()

{

//code to get you on road cost of the vehicle

}

public void petrolConsumptionPerMile()

{

//code to get you Petrol consumption per Mile

// by vehicle

}

}

Now, you would extend the **Vehicle** class in the **HONDACivic** and **CHEVROLETCruze** classes as follows,

class HONDACivic extends Vehicle { }

class CHEVROLETCruze extends Vehicle

{

petrolConsumptionPerMile()

{

throw new UnsupportedOperationException();

}

dieselConsumtionPerMile()

{

//code to get you Diesel consumption per Mile

// by the Chevrolet Cruze car

}

}

Everything is fine, your code would absolutely work when you test the classes with below class:

public VehicleTest

{

public static void main(String[] args)

{

List vehicleList = new ArrayList();

vehicleList.add(new HONDACivic());

vehicleList.add(new CHEVROLETCruze());

getFuelConsumption ( vehicleList );

}

static void getFuelConsumption ( List vehicleList )

{

for ( int i=0; i < vehicleList.size(); i++ )

{

Vehicle v = vehicleList.get(i);

v.petrolConsumptionPerMile();

}

}

}

In the above main function; you are framing a list objects and adding **HONDACivic** and **CHEVROLETCruze** object references and calling the method petrolConsumptionPerMile. Definitely the code would not break because you have taken utmost care while writing the classes. Now, you would deal with two categories of Vehicle – PetrolVehicle and DieselVehicle than dealing with a Vehicle class.

**But, the program is a clear violation to LSP:**

You know that **HONDACivic** is a petrol car and **CHEVROLETCruze** is a diesel car… So, when you make a call to PetrolConsumptionPerMile of a base class using HONDACivic reference, you get the petrol consumption but for CHEVROLETCruze you would get unsupported exception as this does not operate on petrol but diesel. It’s the clear violation of LSP as by replacing **Vehicle** class reference to its derived class CHEVROLETCruze class, program is calling petrolConsumptionPerMile method present in CHEVROLETCruze class and not the one present in the Vehicle class. What could be the consequences of such a class hierarchy:

1. Logically, both derived classes can extend **Vehicle** class, but if there is a functionality lying in a base class which a derived class should not support then definitely it’s not the correct class hierarchy and, in such a case it does make a sense to redefine the hierarchy.

2. In this present case, CHEVROLETCruze does not support petrolConsumptionPerMile method of the base class and it has written the same method in it (which says unsuppported exception). But, imagine a case when the derived class does not contain the overloaded method which it does not support…. your program may break unexpectedly or you get undesirable results (preventing subtyping related bugs)

So, the direct consequence of not following LSP is: the resulting classes would not confirm to appropriate class hierarchies which in some cases may result in break down of the program execution too if not handled properly.

**How to overcome the above problems by following LSP:** You further re-factor your class hierarchy as follows,

**Vehicle** class will be extended by **PetrolVehicle** Class and **DieselVehicle** class as follows:

class Vehicle

{

public void VehicleCost(){

//code to get you on road cost of the Vehicle

}

}

class PetrolVehicle extends Vehicle

{

public void petrolConsumptionPerMile() {

//Code to get you Petrol consumption per mile by Petrol Vehicle

}

}

class DieselVehicle extends Vehicle

{

public void dieselConsumptionPerMile() {

//Code to get you Diesel consumption per mile by Diesel Vehicle

}

}

And, then the **HONDACivic** and **CHEVROLETCruze** classes should extend the **PetrolVehicle** and **DieselVehicle** classes respectively. By doing this, you confirm to the perfect class hierarchy structure and pave way to good programming skills.

**Conclusion**, By following **LSP**, you make your class hierarchies more accurate and pave the way for a robust program structure.

**You 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.**

# 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.

# Dependency Inversion Principle

**Motivation**

In an application we have low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and 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 the 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 read characters from keyboard and write 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 the low level classes. In this case if we want to change the design to direct the output to a new FileWriter class we have to change the Copy class. (Let's assume that it is a very complex class, with a lot of logic and realy hard to test).

In order to avoid such problems we can introduce an abstraction layer between the high level classes and low level classes. Since the high level modules contains the complex logic they should not depend on the low level modules and that the new abstraction layer should not be created based on low level modules. The low level modules are 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 Worker. We need to add a new module to our application because in the company there are some new specialized workers employed. We created a new class SuperWorker for this.

Let's assume that the Manager class is a complex one containing a 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 some time and effort).
* some present functionality from the manager class might be affected.
* the unit testing should be redone.

All those problems will take a lot of time to solve. Now it would be very simple if the application was designed following the Dependency Inversion Principle. That means that 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.

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

|  |
| --- |
| // Dependency Inversion Principle - Bad example class Worker {  public void work() {  // ....working  } }  class Manager {  Worker m\_worker;   public void setWorker(Worker w) {  m\_worker=w;  }   public void manage() {  m\_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:

* Manager class should not be changed.
* minimized risk to affect old funtionallity present in Manager class.
* no need to redone 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 m\_worker;   public void setWorker(IWorker w) {  m\_worker=w;  }   public void manage() {  m\_worker.work();  } } |

**Conclusion**

When this principle is applied it means that the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In that case the creation 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 and a more complex code, but more flexible. This principle can not be applied 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.