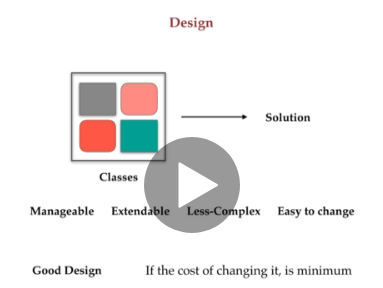
**Design Principles | Art and Craft OOP’s**

**Introduction:**

**Design principles for applying OOPS in java:** First lets discussed about the **Design** what it is and why do we need a good design and how do we know that a design is good or bad?

**Design** is an **art** more than the science, of **how the classes** are to be created and use to deliver a solution. These classes should be **Manageable**, **Extendable, Less Complex and Easy to Change** without any negative impact.



So, we consider a design as **good design if the cost of changing it is minimum**.

And a design is as good design if **it’s less complex. OR simpler.**

And a design is as good design if **it’s easy to test.**

**And there are many more things which makes design as good design that we need to know so that we can design a good design.**

However, it’s not possible most of the time to get a perfect design at first time.

It comes with practice and knowledge that how can we make it less complex and easy to change. So, there are few design principles presents in OOPS to help us to create a clean and modular design which would be very easy to test deeper and maintain in the future.

We will start with the

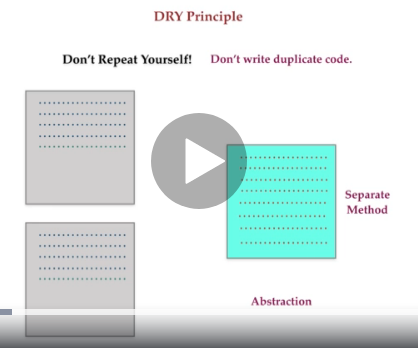
* **DRY(Don’t Repeat Yourself Principle)**
* **KISS(Keep It Simple and Stupid Principle)**
* **YAGNI (You Aren’t Gonna Need It Principle)**
* **SLAP(Single Level of Abstraction Principle)**

After these we will go through the major important concept of design which is **SOLID** acronym below:

* **SRP(Single Responsibility Principle)**
* **OCP(Open-Closed Principle)**
* **LSP(Liskov Substitution Principle)**
* **ISP(Interface Segregation Principle)**
* **DIP(Dependency Inversion Principle)**

**Dry(Don’t Repeat Yourself):-**  This Dry principle says “Don’t Repeat yourself” – That simply means don’t write duplicate code.

Example: Like if you have block of code present in more than one places then put it in separate method and then use it from there. Simply we should use abstraction to abstract common things at one place.

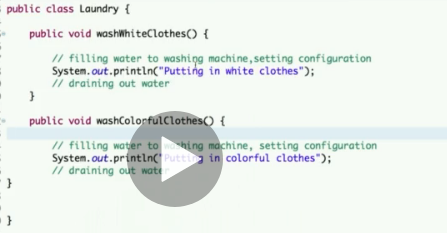


So, we are copying and pasting block of code, we are in violation of  **DRY** principleand that violation may be consider as **WET ( Wasting Everyone Time!)** Code duplication resulting duplicating efforts. If one logic is implemented in more than one places and some ways it gets changed then it will be very difficult to manage that code and we have to make changes all the place where we have written that code.

So, to avoid such situation simple keep it DRY. Remove the duplicate code and duplicating code is very bad idea.



Let’s take a code example:

****

So, as per above screenshot you can see we have two method present inside the Laundry class. If you notice in both method we have common task which are being replicated / duplicated like Filling water to washing machine, setting configuration and draining out the water. So, even these two-method created for performing two different tasks one for **washing white clothes** and another for **washing colorful clothes** but there are common tasks which is present in both like I mentioned above filling and draining of water. So, as per **DRY** principle says we should not keep repeat the same task.

So, we would create another method name **performCommonTasks()** and keep the duplicate task / code in that and then would call this method **[ performCommonTasks() ]** from the **washWhiteClothes()** and **washColorfulClothes()** method.



And there could be a condition which you want to perform but the common objective is to put the common tasks in same place and remove the duplicity as well.

**So, if you have duplicate set of code then create a separate method & put it inside that and if you are using a value more than one time then make it public final concept.**

So, as per Dry principle states : Every piece of knowledge in a system should has single and unambiguous authoritative representation.

* **KISS(Keep It Simple and Stupid Principle):**  This principle says, “**keep it simple and stupid”**. It states that we always should keep the code simple and clear so that it will be easily understood by anyone. So, we should keep the program simple, but how? 🡪 By taking care of few things like **Keep your method small and simple, it should have single type of task and** if we have lot of condition we should break that into smaller method and also if we find long method or lengthy code then we must divide that into multiple method. We can also use **Refactor** option in editor for that. This will make out program clearer and will fail less. Simple code will make you **focused, Easier to understand, fails less, can be easily changed.** Whereas the complexity in code can make you difficult to understand.

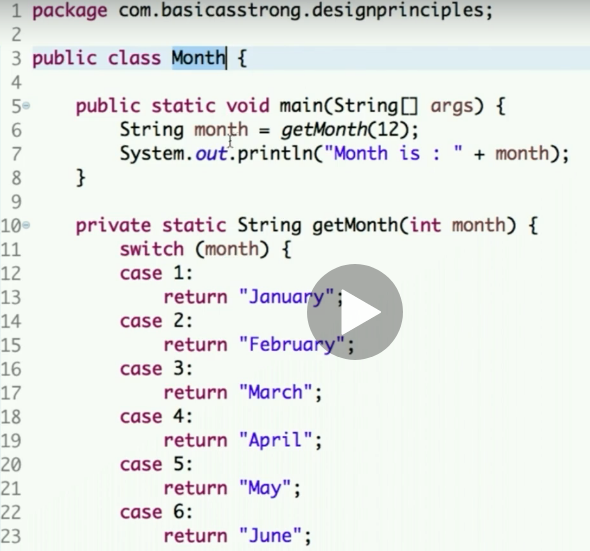
**So how can we differentiate between the complex code and simple code?**

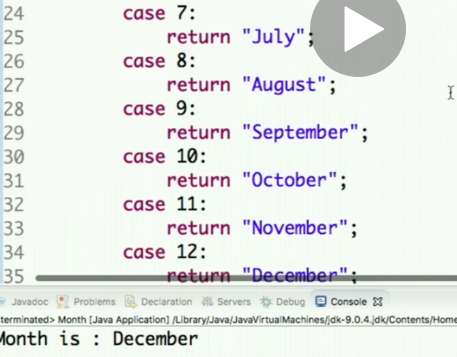
a simple code is **Readable, Maintainable and self-descriptive** so anyone can frame the story of code in his head just by seeing. **And** note it carefully that making code simple is not always taking or using the solution which are familiar to user or other developer.

For example: a for loop is familiar every programmer but it’s not necessarily simple for every problem. May be there are many more solutions that easier or simpler than using a for loop.

So, keep in mind simpler always does not mean familiar.

Example:





So, above as per the screenshot we have one class **Month** and we pass **getMonth(12);** while creatingstring object month in. and we pass 12 as an integer value to the method and while creating string object we get respective month in string like December and then we print that month.

And you see we have method **getMonth(int month)** which returns the months in string and here we are using the switch case and when we run this program we get the output **December as we passed 12 in the argument**

**So, Is this the right way to code?**

If you see the above code this code is full of unnecessary statements for just a simple task.

So, this is the clear violation of KISS principle.

We have one more method given below **getMethodName**(int month);



If you see both are serving the same purpose but you can see the code if the month is not in range between 1 to 12 then it throws the IlligalArgumentException.

And in **ELSE** part we have taken the string array of months and then we are returning the respective month that we want. So, both the method is doing the same job.

In first method there are unnecessary statements there are 12 line of case code BUT in second method there is LESS line of code.

* **YAGNI (You Aren’t Gonna Need It Principle)** This principle is sparrow of object-oriented programming design which is YAGNI principle. (You are not gonna need it.)
* **This principle says: Don’t implement it until you need it.**  Sometime as a programmer you may think of your future perspective and this is not bad at all but it makes things complex and a way complicated you may end-up changing and removing the code as they may not fit in the requirement of your project in future. So, YAGNI principle says for future requirement or approaches **DO NOT WRITE ANY IRRELEVANT OR ADDITIONAL THINGS IN YOUR CODE**” may be you can think of this and that but in most of the cases what happen is you are not gonna need it.

**And if you do so, then you may need to bear the COST OF DELAY, COST TO CARRY, and it will affect the existing implementation. And then you may also need to bear the cost of repairment.**

**So, the question is when should we implement something?**

**Well it’s depends on programming requirements. If the solution is relevant to problem then only you may implement that and also it depend on cost of implementation.**

**If the cost if lower, then you may implement and if the cost is high then you should not implement it.**

**BUT recommended is to not implement and postpone it. It will save your time and move the project efficiently.**

* **SLAP(Single Level of Abstraction Principle):-** So, this principle is about writing function, or we can say methods we know what function what they are for, creating a function is about writing a block of code and that should be re-usable. Re-usable unit should not be big, So, function should not be lengthy to write bigger and longer method the code will look like long scripts messy! And lengthy methods are
* hard to read,
* hard to test,
* hard to remember,
* hard to re-use,
* leads to duplication,
* contains many reasons to change,
* can’t be optimized,
* Not developer friendly

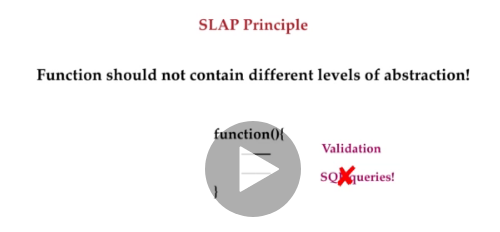
**Low Cohesion, High Coupling and hard to debug and many more reasons of not making methods lengthy.**

**So, how can we judge the method is lengthy? 🡪 So this SLAP comes** m

**We should not care how long method is ! BUT we must know what the level of abstraction level of function is. 🡪 A Function should not contain different levels in abstraction.**

**Example:- A function written for validation only perform validation operations BUT should not perform any SQL query operation.**

**A function should perform one single task and should not perform many tasks.**

****

**So, whatever the method OR block of code we write if should contain same level of abstraction.**

**We do so, so that a function should not have more than one reason to change.**

* **ONLY ONE REASON TO CHANGE**
* **Easy to read**
* **Easy to Test 🡪 As you only test for one thing and developer friendly.**

Let’s take an example: We have class **SlapPrinciple** and we will create a method (**getDetailsAndMails()**)which will take out the details from database and will mail the details to the respective user for which the details are.

package OOPS;

public class SlapPrinciple {

public void getDetailsAndMails() {

// Code to fetch the details from the database.

System.out.println("Fetching the details from the database");

// Code to email the user

System.out.println("Mailing to the user");

}

public static void main(String[] args) {

}

}

So, this is the method and it has functionality of getting / fetching the details from the database and it has the code for sending the email to the user.

So, clearly this is violating the principle as it is having the different abstraction level in above block of code.

Getting details from the database is a **lower level abstraction** as compare to email functionality.

So, if we have **any different level of abstraction** in a single method then we should create a new method for that particular code.

**So, lets create a method :**

package OOPS;

public class SlapPrinciple {

public void getDetailsAndMails() {

// Code to fetch the details from the database.

System.out.println("Fetching the details from the database");

String details = "Details";

emailTheUser(details);

}

public void emailTheUser(String details) {

// Code to email the user

System.out.println("Mailing to the user");

}

public static void main(String[] args) {

}

}

So, now we have separated method having single level of abstraction each.

Remember – If the method is mix of different level of abstraction then divide the code in multiple function by implementing this principle our methods will be simpler and shorter and easy to read.

* **Delegation Principle| Why it is important):-**  Imagine a world without delegation. If manager is only alone working after having a number of employees, so it’s not possible.

In BIG organization a particular task is assigned to manager and then manager delegates the responsibility to the employee. So, it’s not possible to run such organization and big code base without delegation. IN code base too let the specialize the class or method do job is the life will not remain simpler anymore.

**So, Delegation principle says :- Handling over the responsibility for a particular task to another class or method. It’s a technique where object expresses a certain behavior offside but in reality it delegates the responsibility for implementing that behavior to associated object.**

****

**Delegation is used to reduce the coupling of method often to their classes. Component are identical but realize that the situation can change in future. So, if you need to use a functionality in another class but you don’t want to change the functionality then use the delegation instead of inheritance.**

****

**Let’s see in an example:**  Below code example:

package DesignPrinciple;

public interface TravelBooking {

// create one abstract method

public void bookTicket();

}

// Next step is to create a class which will train ticket and then will create one more class which will live ticket.

class TrainTicket implements TravelBooking {

// Now will override the methods

@Override

public void bookTicket() {

System.out.println("Train ticket booked!");

}

}

// Now for the flight we will implements the same

class FlightTicket implements TravelBooking {

// Now will override the methods

@Override

public void bookTicket() {

System.out.println("Flight ticket booked!");

}

}

// Now the code is ready, but we have not implemented delegation principle yet.

// Now let's delegates this responsibility of booking the train ticket to

// TicketBooking by agent by delegation.

// So that in future if the booking logics need to be changed then it has to be

// changed in class only.

// We will create a class called TicketbookingByAgent which will also implement

// TravelBooking because you also want to delegate

// the responsibility of booking the tickets to this agent.

class TicketBookingByAgent implements TravelBooking {

TravelBooking t;

public TicketBookingByAgent(TravelBooking t) {

this.t = t;

}

@Override

public void bookTicket() {

t.bookTicket();

}

}

**Driver Class:**

package DesignPrinciple;

public class DelegationDemonstration {

public static void main(String[] args) {

TicketBookingByAgent agent = new TicketBookingByAgent(new FlightTicket());

agent.bookTicket();

}

}

**So, here the booking of TravelBooking is delegated to TicketBookingByAgent() and using that TravelBooking class has become separate and when the logic of booking needs to be changed then it has to be change only in this class. So, this is the benefit of implementing the giving an implementation to this interface.**

**SOLID | SRP |(Single Responsibility Principle):**  Single Responsibility Principle says – A class should have one, and only one reason to change. That is a class should have single responsibility.

For example: Every java built-in class which we used is written for a specified reason. Like **Date** class – We can use Date class for **Date and Time related features** only.

So, when you start programming and creating classes focus on the primary responsibility of the class.

Below we have example how a class written having multiple responsibilities. Suppose we need a scheduling application with **database persistent** and **notification sending feature.**

Code Example:

package DesignPrinciple;

class TaskScheduling {

public String compute() {

return "Computing data based on user input";

}

public String save() {

return "Save the object to database";

}

public String sendNotification() {

return "Send the notification to the user";

}

}

class ScheduleApplication {

TaskScheduling ts;

public ScheduleApplication(TaskScheduling task) {

this.ts = task;

}

public void schedule() {

System.out.println(ts.compute());

System.out.println(ts.save());

System.out.println(ts.sendNotification());

}

}

public class TaskExe {

public static void main(String[] args) {

ScheduleApplication sa = new ScheduleApplication(new TaskScheduling());

sa.schedule();

}

}

**So,**  as per above program, we are computing data based on user input, saving the object to the database and sending notification to the user.

**BUT you can see the “Single Responsibility Principle” is violated. Why this single class having three responsibilities? i.e. compute(), Save() and sendNotification().**

How can we prevent this? – We can create a three-separate class

Compute(+computingData()), Save(+saveData()) and Notification(+sendNotification()).

So, let’s create three different classes for three different class.

Now below is optimize code and does not violate the principle of SRP. Now three different classes perform single responsibility, Compute class for all computing Save class for saving data into database and Notification class for sending notification. So, this is all how w can refactor a good code.

package DesignPrinciple;

class Compute {

public String compute() {

return "Computing data based on user input";

}

}

class Save {

public String save() {

return "Save the object to database";

}

}

class Notification {

public String sendNotification() {

return "Send the notification to the user";

}

}

class ScheduleApplication {

Compute c;

Save s;

Notification n;

public ScheduleApplication(Compute c, Save s, Notification n) {

this.c = c;

this.s = s;

this.n = n;

}

public void schedule() {

System.out.println(c.compute());

System.out.println(s.save());

System.out.println(n.sendNotification());

}

}

public class TaskExe {

public static void main(String[] args) {

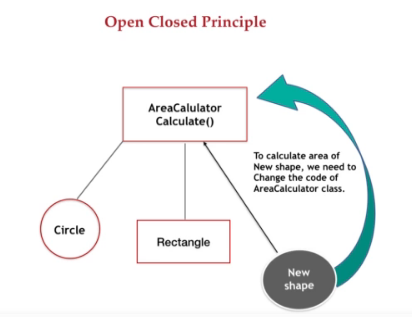
ScheduleApplication sa = new ScheduleApplication(new Compute(), new Save(), new Notification());

sa.schedule();

}

}

**SOLID | Open-Closed Principle:** Open-Closed principle states that “Software entities (classes, modules, functions, etc.) should be **open** for **extension** BUT **closed** for **modification**.



If we try to pass a new shape and if we do some changes there then it violates the principle of open-closed. Why because we are changing core logics of calculator class. Which is nothing but the modification of that class. So, we have to understand that if we are changing the stable or core logics of any class then we need to test all the functionalities in the application which are using that class because all other existing component may get affected with that.

So, let’s understand this violation by code:

package DesignPrinciple;

class Rectangle {

double height;

double width;

public Rectangle(double h, double w) {

this.height = h;

this.width = w;

}

}

class Circle {

double radius;

public Circle(double r) {

this.radius = r;

}

}

class Trangle {

double base;

double height;

public Trangle(double base, double height) {

this.base = base;

this.height = height;

}

}

class AreaCalculator {

Object shape;

public AreaCalculator(Object shape) {

this.shape = shape;

}

public void calculate() {

if (shape instanceof Rectangle) {

Rectangle r = (Rectangle) shape;

System.out.println(r.height \* r.width);

}

if (shape instanceof Circle) {

Circle c = (Circle) shape;

System.out.println(3.14 \* c.radius \* c.radius);

}

if (shape instanceof Trangle) {

Trangle t = (Trangle) shape;

System.out.println(0.5 \* t.base \* t.height);

}

}

}

public class Area {

public static void main(String[] args) {

AreaCalculator cal = new AreaCalculator(new Rectangle(20, 10));

cal.calculate();

}

}

So, does this class violate the principle of open-closed? And the answer is **YES,** because this class contain core logic and we have to touch it this **AreaCalculator** class core logic for any new shape. Here we are adding in core when any new shape comes.

And this is makes this **AreaCalculator** class more dependent on the other classes.

So, above code works fine but since we passed / added new shape “Trangle”, but it’s not the proper way of doing it and violates the open-closed principle.

So, how can we refactor this core code logic and implement it in correct way so, it does not violate the principle? Let’s see this below :- Abstraction and Polymorphism are the key to refactor the code the calculation logic of the shape should remain the new object that we are supplying.

So, we will declare the shape class **Abstract class and whenever the new shape is added it will come with its own logic of area calculation.**

Let’s see how the above written code is refactored below: -

package DesignPrinciple;

abstract class Shape {

public abstract void calculateArea();

}

class Rectangle extends Shape {

double height;

double width;

public Rectangle(double h, double w) {

this.height = h;

this.width = w;

}

@Override

public void calculateArea() {

System.out.println(height \* width);

}

}

class Circle extends Shape {

double radius;

public Circle(double r) {

this.radius = r;

}

@Override

public void calculateArea() {

System.out.println(3.14 \* radius \* radius);

}

}

class Triangle extends Shape {

double base;

double height;

public Triangle(double base, double height) {

this.base = base;

this.height = height;

}

@Override

public void calculateArea() {

System.out.println(0.5 \* base \* height);

}

}

class AreaCalculator {

public void calculate(Shape shape) {

shape.calculateArea();

}

}

public class Area {

public static void main(String[] args) {

AreaCalculator cal = new AreaCalculator();

cal.calculate(new Rectangle(20, 10));

cal.calculate(new Triangle(10, 5));

cal.calculate(new Circle(10));

}

}

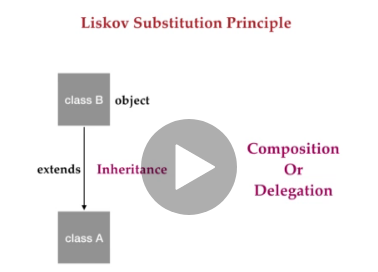
So, just by moving the logic of calculation to the respective owner now we are not required to change the logic of the core class. So, now AreaCalculator class is open for extension and can be consider as closed for any modification.

**SOLID | Liskov Substitution Principle:** This principle prescribes substitutability of a class by its sub-class.

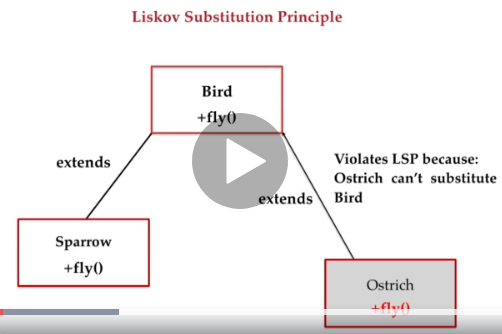
Broadly put, a class can be replaced by its sub-class in all practical usage scenarios.

A derived class / sub-class should be substitutable their base and parent class.

So, inheritance should be used inly for substitutability. Like if we have class B and B object can be anywhere as an object of class A then we should used inheritance, BUT it cannot be used as an object of A then we should use **Composition OR Delegation**



Example:



Code:

package DesignPrinciple;

class Bird {

// Base functionality related to bird.

public void fly() {

System.out.println("I am a flying bird!");

}

}

class Sparrow extends Bird {

}

class Ostrich extends Bird {

}

public class LSPExe {

public static void main(String[] args) {

Ostrich bird = new Ostrich();

bird.fly(); /\* Ostrich cannot fly \*/

}

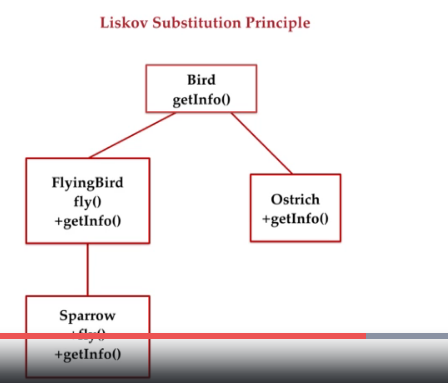
}

As per above code Ostrich is also extending all the features of Bird class BUT

Ostrich cannot fly!

Ostrich is also a bird, so it has to extends the Bird class to have all the functionalities, BUT Ostrich actually does not have flying functionality. So, it’s always a bad practice to create a parent with functionality which all its child cannot have.

Let’s refactor the above code: -



package DesignPrinciple;

class Bird {

// Base functionality related to bird.

public void getInfo() {

System.out.println("I am a Bird!");

}

}

class FlyingBird extends Bird {

public void fly() {

System.out.println("I am a flying bird!");

}

}

class Sparrow extends Bird {

}

class Ostrich extends Bird {

}

public class LSPExe {

public static void main(String[] args) {

Ostrich bird = new Ostrich();

bird.getInfo();

}

}

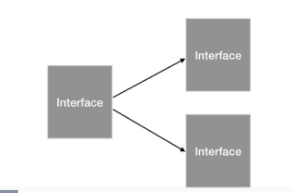
Now the Sparrow class does not extend the Bird class directly because it’s already a bird, but it will extend the FlyingBird class directly because its fly where as Ostrich can extend a bird class as it’s bird but cannot extends FlyingBird because it cannot fly.

So, remember inheritance is used only when the derived one substitute the base class perfectly. Even if you see in core java JDK implementation we cannot decrease visibility of any method in derived class.

For example :- we cannot make method private or protected in derived class if the method is public in the base class. This is the constraint that compiler force to satisfy **Liskov Substitution principle.**

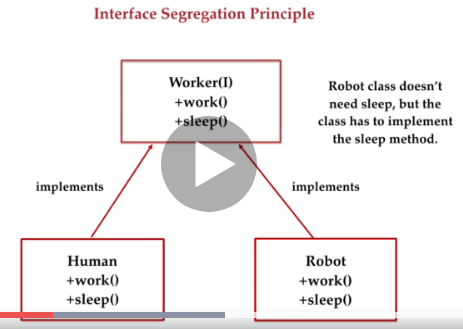
**SOLID | Interface Segregation Principle(ISP):-**  This principle prescribes a **Client should not be forced to implement an interface that does not use.**

This means we should break out the interface in many smaller one so the better satisfy the exact needs of our client. Keep interfaces cohesive and less bulky.



Basically, this is single responsibility principle but at interface level.

Let’s see this in an example:



**Worker(interface)** having two method +work() and +sleep(). **Human** is implementing **worker** interface and he can work and sleep. But as **Robot** is also a worker and if robot will implement **worker** interface then it has to implement the sleep() method along with the work() method. Since this **Robot class** has no need of sleep but it is bound to implement the sleep method. Hence this violates the principle of interface segregation.

Code example:- Create a main class ISP with main method.

Create an interface **Worker** and two method **work() and sleep()** inside it.

Create a class **Human** which implements **Worker.** Also implement the unimplemented method which should be override.

Create a **Robot** class which also implement the **Worker** interface.

Example:

package DesignPrinciple;

interface Worker {

public void work();

public void sleep();

}

class Human implements Worker {

@Override

public void work() {

// TODO Auto-generated method stub

}

@Override

public void sleep() {

// TODO Auto-generated method stub

}

}

class Robot implements Worker {

@Override

public void work() {

// TODO Auto-generated method stub

}

@Override

public void sleep() {

// This sleep() method does not required to be implement as it violate the

// principle of interface segregation.

}

}

public class ISP {

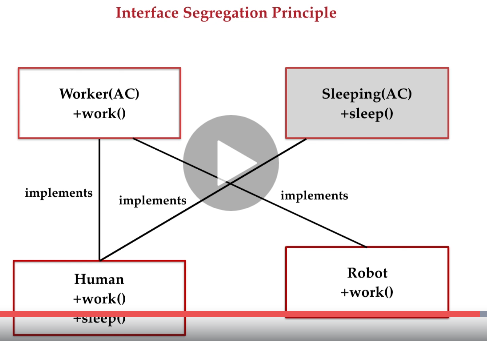
public static void main(String[] args) {

}

}

Here **Robot** class does not need to implement the sleep() method. As it violates the principle of interface segregation.

Let’s Refactor above code.



As you can see the Human can work and sleep so, it can implement both interface(Worker + Sleep) while Robot can only work so it’s implementing only one interface (Worker).

So, below we are segregating the interface so that it can satisfy the exact needs.

To sum-up – this principle enforces the extendibility, Re-usability.

Example:

package DesignPrinciple;

interface Worker {

public void work();

}

interface Sleep {

public void sleep();

}

class Human implements Worker, Sleep {

@Override

public void work() {

// TODO Auto-generated method stub

}

@Override

public void sleep() {

// TODO Auto-generated method stub

}

}

class Robot implements Worker {

@Override

public void work() {

// TODO Auto-generated method stub

}

}

public class ISP {

public static void main(String[] args) {

}

}

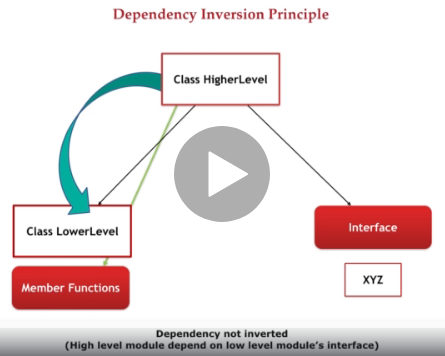
**SOLID | Dependency Inversion Principle :-**  This principle states “**Inverting the design where low level modules defining the interface that higher-level modules depend on.**”

So, we get; “**higher level modules that defines interfaces that lower level modules implement.**”

As per bob martin says: **High level module should not depend on low level modules. Both should depend on abstractions. Abstraction should not depend on details. Details should depend on abstractions.** **Founder of SOLID principle – Bob Martin**

So, high level module should depend on abstractions like **interfaces** and **abstractions** classes instead of concrete classes.

So, while developing new software on application what we do we first always start developing lower level modules such as disk access, database connection etc. after the development and testing of lower level module THEN we develop the higher-level modules that is highly depend on developed lower level modules. So, this is the basic way what we have been following.



BUT right way of doing things let’s see in code example.

Let’s first create a **Logger** class as lower level module and then let’s create a **loggerInformation** method in it. 🡪 This method will log the information in it.

Now let’s create a class **LoggingToFile** as higher-level modules. In this class let’s declare a logger variable of type logger class and initialize it.

Now let’s create a **logging()** method in it. So, by using this method we are calling the logger class object and will call the log information method on it and pass the log information.

Now let’s create an object of **LoggingToFile** class in the main method means Higher module class and pass the Logger class object as parameter in it that is of lower module.

Then call the **logging()** method of **LoggingToFile** class with **LoggingToFile** class object **fileLogger**.

Example:

package DesignPrinciple;

// Let's First create a lower level module.

class Logger {

public void loggerInformation(String logInfo) {

System.out.println(logInfo);

}

}

// Now let's create a higher level module.

class LoggingToFile {

private Logger logger;

public LoggingToFile(Logger log) {

this.logger = log;

}

public void logging() {

logger.loggerInformation("Printing the logs to the file...");

}

}

public class DIP {

public static void main(String[] args) {

LoggingToFile fileLogger = new LoggingToFile(new Logger());

fileLogger.logging();

}

}

So, here we are having two class Logger and LoggingToFile

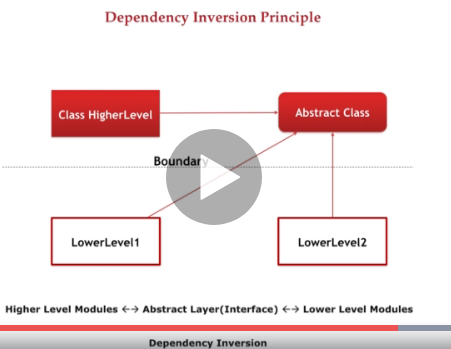
Logger class has method logInformation() and

LoggingToFile class has constructor and inside it constructor it’s initializing the logger object then inside the logging() method calling the logInformation() method by using Logger class object.

So, when we create the object of LoggingToFile and invoke the logging() method with it , then it’s internally calling logInformation() method of Logger class also it’s creating logger object when we are creating LoggingToFile object so by doing this LoggingToFile class is dependent on Logger class and any future change in Logger class is going to affect LoggingToFile class, so this should not happen.

So, the thing is Higher level module should not depend on lower level module directly.

We have to use abstraction by using abstraction class or interface and both higher and lower level module should depend on this interface or this abstract class instead of depending on each other directly.



There should be a boundary between higher level and lower level module that you can see in abstract layer.

So, Let’s now refactor this above code :-

Let’s create one interface to create an abstract layer between the higher and lower level module.

And then just declare the abstract method - “**public void loggerInformation(String logInfo);**”

Example:

Interface ILogger {

public void loggerInformation(String logInfo);

}

Now make Logger class to implement the ILogger interface.

Example: “class Logger implements ILogger”

Now the Logger class is directly not linked to LoggingToFile class.

Now the ILogger instance will be declare in LoggingToFile

Example: Private ILogger logger;

Public LoggingToFile(ILogger log) {

this.logger=log;

}

Code Example:

package DesignPrinciple;

interface ILogger {

public void loggerInformation(String logInfo);

}

// Let's First create a lower level module.

class Logger implements ILogger {

public void loggerInformation(String logInfo) {

System.out.println(logInfo);

}

}

// Now let's create a higher-level module.

class LoggingToFile {

private ILogger logger;

public LoggingToFile(Logger log) {

this.logger = log;

}

public void logging() {

logger.loggerInformation("Printing the logs to the file...");

}

}

public class DIP {

public static void main(String[] args) {

LoggingToFile fileLogger = new LoggingToFile(new Logger());

fileLogger.logging();

}

}

Now the output would be the same but this time LoggingToFile class is not directly depend on Logger class. And utilize the ILogger interface.

Meaning you can switch IN and OUT implementation of ILogger at runtime, object instantiation etc.

Now what if you don’t want to log into console but to the database. So, then we can easily add one higher module such as log to DB without any change in LoggingToFile class.

Example:

class LoggingToDB {

private ILogger logger;

public LoggingToDB(Logger log) {

this.logger = log;

}

public void logging() {

logger.loggerInformation("Saving the logs to the database...");

}

}

So, this is how we are creating one more module without depending on Logger class via ILogger interface.

Example:

package DesignPrinciple;

interface ILogger {

public void loggerInformation(String logInfo);

}

// Let's First create a lower level module.

class Logger implements ILogger {

public void loggerInformation(String logInfo) {

System.out.println(logInfo);

}

}

// Now let's create a higher-level module.

class LoggingToFile {

private ILogger logger;

public LoggingToFile(Logger log) {

this.logger = log;

}

public void logging() {

logger.loggerInformation("Printing the logs to the file...");

}

}

class LoggingToDB {

private ILogger logger;

public LoggingToDB(Logger log) {

this.logger = log;

}

public void logging() {

logger.loggerInformation("Saving the logs to the database...");

}

}

public class DIP {

public static void main(String[] args) {

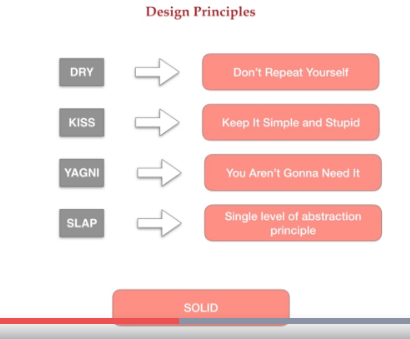
LoggingToFile fileLogger = new LoggingToFile(new Logger());

fileLogger.logging();

}

}

**SO, HERE IS ALL OF THE OOPS DESIGN PRINCIPLE.**



Start practicing these design principles in real life coding because it takes lot of time to practice them and realize the difference between the badly written code and nicely written code. So, gradually when you practice them then definitely you would become a better OOPS programmer.

And your program / classes will become **Manageable, Extendable, Less-Complex and Easy to Change.**

**Remember: Design Principles and Design Patterns are artistic angle of programming.**

**It takes lot of time and practice to become a good technical artist!**