**Basics of Object-oriented programming**

Abstraction

Encapsulation

Polymorphism

Inheritance

abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

to handle complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

**Rule of three** is a [code refactoring](https://en.wikipedia.org/wiki/Code_refactoring) [rule of thumb](https://en.wikipedia.org/wiki/Rule_of_thumb) to decide when a replicated piece of code should be replaced by a new procedure. It states that the code can be copied once, but that when the same code is used three times, it should be extracted into a new procedure. The rule was introduced by [Martin Fowler](https://en.wikipedia.org/wiki/Martin_Fowler) in *Refactoring*[[1]](https://en.wikipedia.org/wiki/Rule_of_three_(computer_programming)" \l "cite_note-1) and attributed to Don Roberts.

**Principles Of Object Oriented Design**

**A suite of eleven principles, conceived by people such as RobertCecilMartin, BertrandMeyer, BarbaraLiskov, etc. and compiled by RobertCecilMartin.**

**There are six papers that describe all of these principles. They can be found in the resources section of http://www.objectmentor.com/. The first four papers are named for the first four principles, the final papers cover the remaining two principles: "Granularity" and "Stability".**

**There are five principles of class design (aka SOLID):**

**S.O.L.I.D** is an acronym for the **first five object-oriented design (OOD) principles** by Robert C. Martin, popularly known as [Uncle Bob](https://en.wikipedia.org/wiki/Robert_Cecil_Martin).

S.O.L.I.D stands for:

S - Single-responsiblity principle

O - Open-closed principle

L - Liskov substitution principle

I - Interface segregation principle

D - Dependency Inversion Principle

**(SRP) The SingleResponsibilityPrinciple**

**(OCP) The OpenClosedPrinciple**

**(LSP) The LiskovSubstitutionPrinciple**

**(ISP) The InterfaceSegregationPrinciple**

**(DIP) The DependencyInversionPrinciple**

**There are three principles of package cohesion**

**(REP) The ReuseReleaseEquivalencePrinciple**

**(CRP) The CommonReusePrinciple**

**(CCP) The CommonClosurePrinciple**

**There are three principles of package coupling**

**(ADP) The AcyclicDependenciesPrinciple**

**(SDP) The StableDependenciesPrinciple**

**(SAP) The StableAbstractionsPrinciple**

**(SRP) SingleResponsibilityPrinciple**

**What?**

**A class should have one, and only one, reason to change.**

This principle can apply for method as well.

**Why?**

How?

If there are more than one reason to change the class or method then create separate class or method for each reason.

**(OCP) OpenClosedPrinciple**

**What?**

Objects or entities should be open for extension, but closed for modification.

**Why?**

We should restrict modification of software entity otherwise it has to test again and again.

**How?**

OCP can be used with the help of Template pattern or Strategy pattern.

**(LSP) LiskovSubstitutionPrinciple**

**What?**

Every subclass/derived class should be substitutable for their base/parent class.

If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S)

If a code violates LSP then it'll also violate the OCP as well.

**Why?**

**How?**

We can solve this principle using Template design pattern.

Interface segregation principle can also be used to solve this principle.

**(ISP) Interface Segregation Principle**

**What?**

Make fine grained interfaces that are client specific.

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

**Why?**

**How?**

**(DIP) Dependency Inversion Principle**

**What?**

Depend on abstractions, not on concretions.

If we violate the dependency injection principle then we violate open closed principle as well and vice versa.

**Why?**

Low level modules first then implement high level modules is not very flexible.

Whenever there is change in low level modules we'll have to rewrite the high level module.

HLM should not be directly depends on LLM.

HLM interact with LLM using Abstract Layer (interface).

LLM depends on abstract layer  and similarly HLM also depends on abstract layer. They don't know about each other and don't depend to each other.

**How?**

Avoid the instantiation of a given class directly.

Dog dog = new Dog()

Animal dog = new Dog()

Instantiating a class using new keyword inside  another class, makes classes tightly coupled to each other.

Modules should depend on abstraction.

Obj1 refers obj2 - direct Dependency

Obj1 refers interface and obj2 implements interface. So they don't know about each other.

DIP can be implemented using several ways.

1. Factory design pattern, template , strategy pattern.

2. Service locator pattern

To use inversion of control

3. Using dependency Inversion

**2. There are three principles of package cohesion**

**(REP) ReuseReleaseEquivalencePrinciple**

**(CCP) CommonClosurePrinciple**

**(CRP) CommonReusePrinciple**

**(REP) ReuseReleaseEquivalencePrinciple**

**What?**

**The granule of reuse is the granule of release.**

**Reusability is one of the most oft claimed goals of OOD. But what is reuse? Is it reuse if I snatch a bunch of code from one program and textually insert it into another? It is reuse if I steal a module from someone else and link it into my own libraries? I don’t think so.**

**The above are examples of code copying; and it comes with a serious disadvantage: you own the code you copy! If it doesn’t work in your environment, you have to change it.**

**If there are bugs in the code, you have to fix them. If the original author finds some bugs in**

**the code and fixes them, you have to find this out, and you have to figure out how to make**

**the changes in your own copy. Eventually the code you copied diverges so much from the**

**original that it can hardly be recognized. The code is yours. While code copying can make**

**it easier to do some initial development; it does not help very much with the most expen-**

**sive phase of the software lifecycle, maintenance.**

**I prefer to define reuse as follows. I reuse code if, and only if, I never need to look at**

**the source code (other than the public portions of header files). I need only link with static**

**libraries or include dynamic libraries. Whenever these libraries are fixed or enhanced, I**

**receive a new version which I can then integrate into my system when opportunity allows.**

**That is, I expect the code I am reusing to be treated like a product. It is not maintained**

**by me. It is not distributed by me. I am the customer, and the author, or some other entity,**

**is responsible for maintaining it.**

**When the libraries that I am reusing are changed by the author, I need to be notified.**

**Moreover, I may decide to use the old version of the library for a time. Such a decision**

**will be based upon whether the changes made are important to me, and when I can fit the**

**integration into my schedule. Therefore, I will need the author to make regular releases of**

**the library. I will also need the author to be able to identify these releases with release**

**numbers or names of some sort.**

**Thus, I can reuse nothing that is not also released. Moreover, when I reuse something**

**in a released library, I am in effect a client of the entire library. Whether the changes affect**

**me or not, I will have to integrate with each new version of the library when it comes out,**

**so that I can take advantage of later enhancements and fixes.**

**And so, the REP states that the granule of reuse can be no smaller than the granule of**

**release. Anything that we reuse must also be released. Clearly, packages are a candidate**

**for a releasable entity. It might be possible to release and track classes, but there are so**

**many classes in a typical application that this would almost certainly overwhelm the**

**release tracking system. We need some larger scale entity to act as the granule of release;**

**and the package seems to fit this need rather well.**

**(CRP) The CommonReusePrinciple**

What?

**Classes that are used together are packaged together.**

SOLID principle

These principle will be implemented using design patterns.

S

Single Responsibility Principal:

A class or method should have only one reason to change.

If there are more than one reason to change the class or method then create separate class or method for each reason.

It's favourable to use composition over inheritance.

O - Open Closed Principal

Application should be able to accept frequent changes.

New functionality can be added with minimum changes.

Software entity should be open for extension but closed for modification.

Restrict modification of software entity otherwise it's to test again and again.

OCP can be used with the help of Template pattern or Strategy pattern.

Strategy pattern: keep existing code unchanged as much as it is possible.

SortType - enum

Abstract class: Sorter -

SortType

Void sort()

MergeSort extends Sorter -

sort() - implementation

MergeSort()

super.SortType=MERGE

SorterManager-

sort(Sorter)-

If Sorter.SortType == MERGE

Sorter.sort()

Main

SorterManager sm = new SorterManager();

sm.sort(new MergeSort());

---------

Strategy pattern

Sorter- interface

sort()

MergeSort implements Sorter -

Sort()

SorterManager-

Sort(Sorter)-

Sorter.sort()

L - Liskov substitution principle

Child class should never break the parent class Type definition.

 if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S)

If a code violates LSP then it'll also violate the OCP as well.

Derived classes must be substitutable for the base class.

Car IS A Vehicle

Vehicle - drive ()

In the method argument where we are using Car if we change it to Vehicle then that method should work.

We can solve this principle using Template design pattern.

Interface segregation principle can also be used to solve this principle.

Interface

Vehicle

speed ()

addFuel()

Car implements Vehicle

Speed

addFuel

ElectricCar implements Vehicle

speed

addFuel ()

Throw Exception

CarManager

Main()

Vehicle car = new ElectricCar ();

car.addFuel();//this will throw Exception

//Either We should not implement the method which we are not going to use or throw runtime exception.

LSP

Interface NormalVehicle

speed

addFuel

Interface ElectricVehicle

speed

chargeBattery

Car implements NormalVehicle

speed

addFuel

ElectricCar implements ElectricVehicle

speed

chargeBattery

CarManager

main

ElectricVehicle ev = new ElectricCar()

ev.chargeBattery();

--------------\_----------------------+-

I

Interface Segregation Principle

ISP

We can have abstraction of the system using interface.

D - Dependency Inversion Principle

Low level modules first then implement high level modules is not very flexible.

Whenever there is change in low level modules we'll have to rewrite the high level module.

HLM should not be directly depends on LLM.

HLM interact with LLM using Abstract Layer (interface).

LLM depends on abstract layer  and similarly HLM also depends on abstract layer. They don't know about each other and don't depend to each other.

Avoid the instantiation of a given class directly.

Dog dog = new Dog()

Animal dog = new Dog()

Instantiating a class using new keyword inside  another class, makes classes tightly coupled to each other.

Modules should depend on abstraction.

Obj1 refers obj2 - direct Dependency

Obj1 refers interface and obj2 implements interface. So they don't know about each other.

DIP can be implemented using several ways.

1. Factory design pattern, template , strategy pattern.

2. Service locator pattern

To use inversion of control

3. Using dependency Inversion

If we violate the dependency injection principle then we violate open closed principle as well and vice versa.

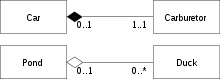
References:

<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

<https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>

<http://wiki.c2.com/?PrinciplesOfObjectOrientedDesign> - **Principles Of Object Oriented Design**

<https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view> **- SRP**

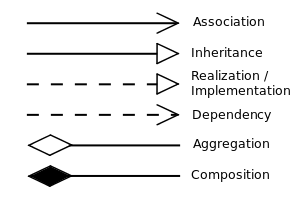


Two class diagrams. The diagram on top shows **Composition** between two classes: A Car has exactly one Carburetor, and a Carburetor has at most one Car (Carburetors may exist as separate parts, detached from a specific car).

The diagram on bottom shows **Aggregation** between two classes: A Pond has zero or more Ducks, and a Duck has at most one Pond (at a time).



Class diagram showing **dependency** between "Car" class and "Wheel" class (An even clearer example would be "Car depends on Wheel", because Car already aggregates (and not just uses) Wheel)



The UML representation of a composition relationship shows composition as a filled diamond shape on the containing class end of the lines that connect contained class(es) to the containing class.

Differences between Composition and Aggregation

**Composition relationship**

1. When attempting to represent real-world whole-part relationships, e.g. an engine is a part of a car.

2. When the container is destroyed, the contents are also destroyed, e.g. a university and its departments.

**Aggregation relationship**

1. When representing a software or database relationship, e.g. car model engine ENG01 is part of a car model CM01, as the engine, ENG01, may be also part of a different car model.

2. When the container is destroyed, the contents are usually not destroyed, e.g. a professor has students, when the professor dies the students don't die along with him or her.