



Deep Dive Design pattern

Rajeev Gupta
rgupta.mtech@gmail.com
Java Trainer & consultant
@rajeev_gupta76





Rajeev Gupta

Freelancer Corporate Java JEE/ Spring Trainer
New Delhi Area, India

Add profile section ▾

More...

- freelance**
- Institution of Electronics and Telecommunication...**
- See contact info**
- See connections (500+)**

1. Expert trainer for Java 8, GOF Design patterns, OOAD, JEE 7, Spring 5, Hibernate 5, Spring boot, microservice, netflix oss, Spring cloud, angularjs, Spring MVC, Spring Data, Spring Security, EJB 3, JPA 2, JMS 2, Struts 1/2, Web service

2. Helping technology organizations by training their fresh and senior engineers in key technologies and processes.

3. Taught graduate and post graduate academic courses to students of professional degrees.

I am open for advance java training /consultancy/ content development/ guest lectures/online training for corporate / institutions on freelance basis

Contact :

=====

rauuta.mtech@amail.com

Clients:

=====

Gemalto, Noida
Cyient Ltd, Noida
Fidelity Investment Ltd
Blackrock, Gurgaon
Mahindra comviva
Steria
Bank Of America
incedo gurgaon
MakeMyTrip
Capgemini India
HCL Technologies
CenturyLink
Deloitte consulting
Nucleus Software
Ericsson Gurgaon
Avaya gurgaon
Kronos Noida
NEC Technologies
A.T. Kearney
ust global
TCS
North Shore Technologies Noida

IBM
Sapient
Accenture
Incedo
Genpact
Indian Air Force
Indian railways
Vidya Knowledge Park



“Too busy chopping wood to sharpen the axe”



5th class



10th class



12th class



SOLID

clear understanding of OOPs concepts

BTech/MTech



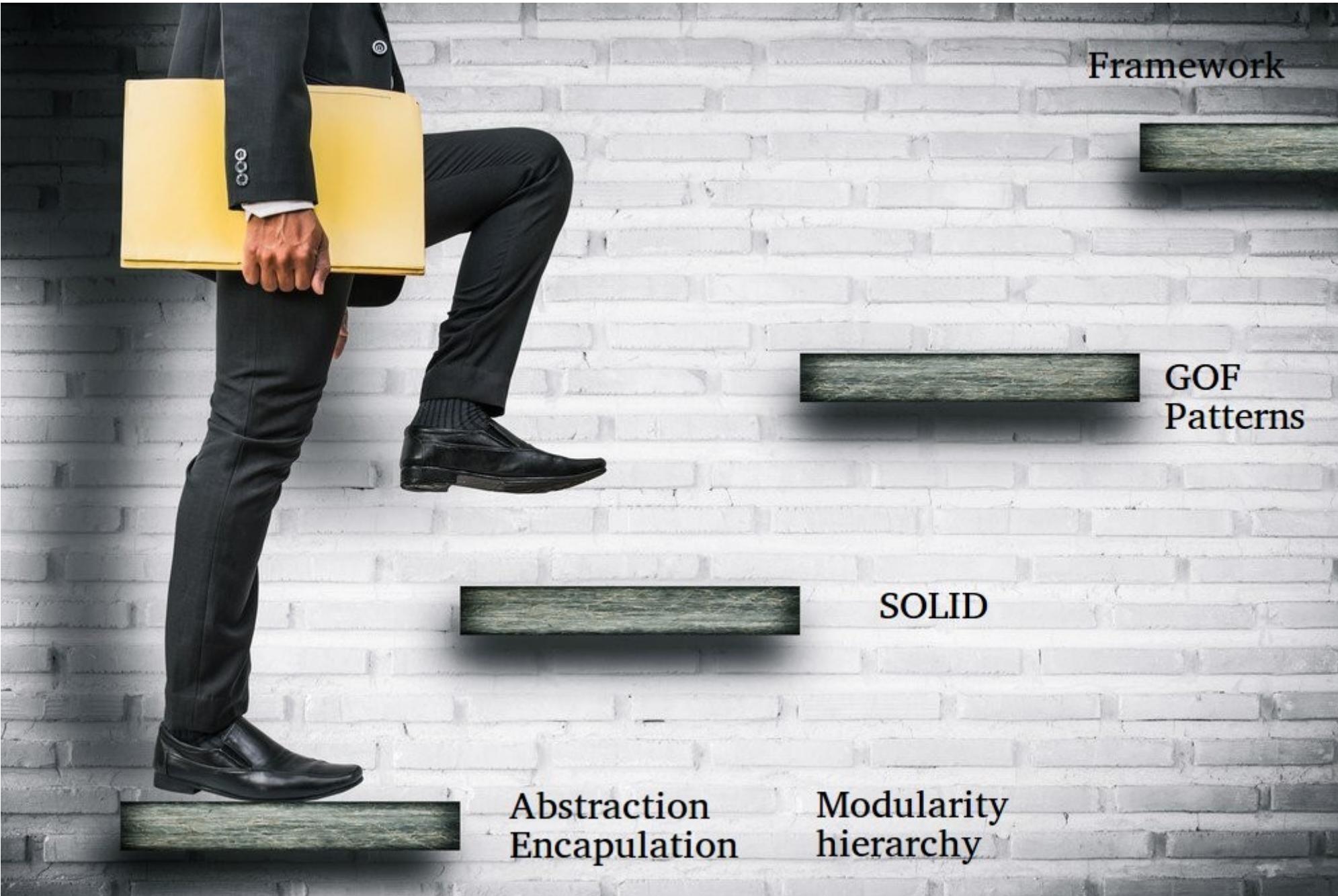
Lib & Framework

Design patterns

PRACTICE
MAKES
PERFECT



When watching experts perform
it's easy to forget how much effort
they've put into reaching high
levels of achievement



Framework



GOF
Patterns



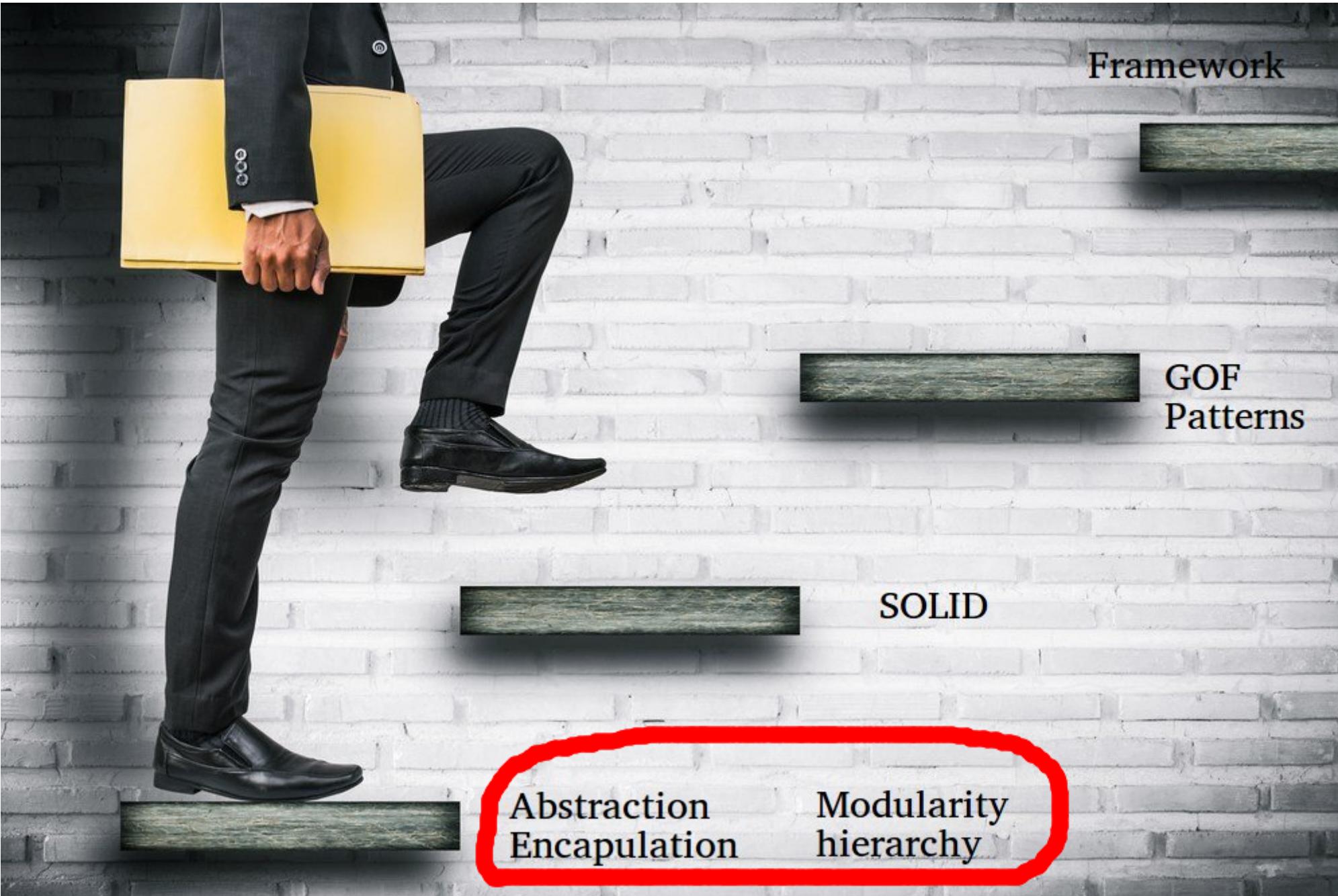
SOLID



Abstraction
Encapsulation



Modularity
hierarchy

A photograph of a man in a dark suit and black leather shoes walking up a set of grey brick stairs. He is carrying a large, light-colored briefcase in his right hand. The background consists of a white brick wall.

Framework



GOF
Patterns

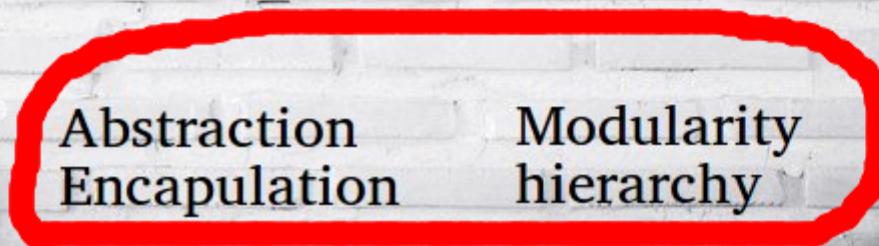


SOLID



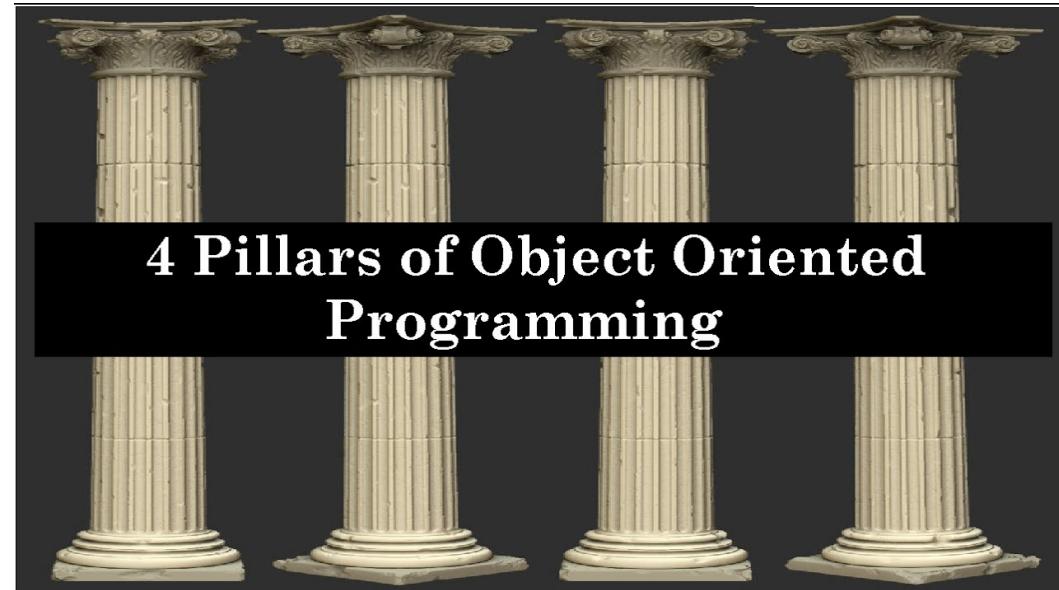
Abstraction
Encapsulation

Modularity
hierarchy



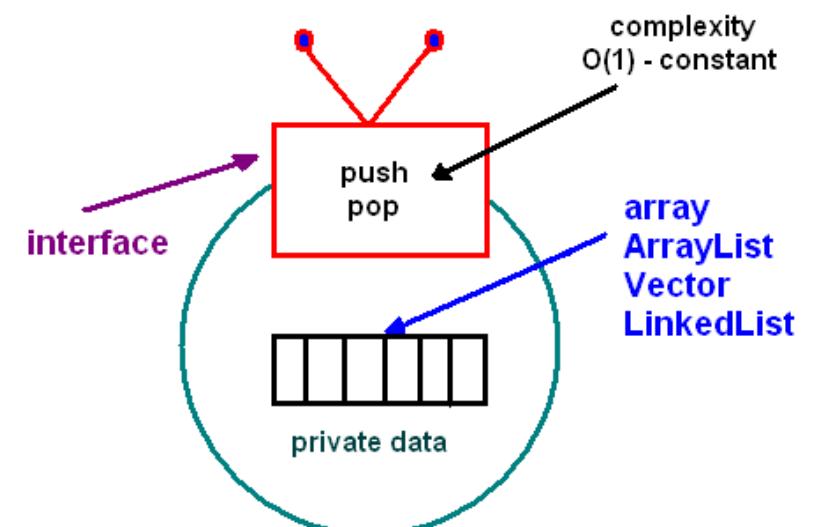
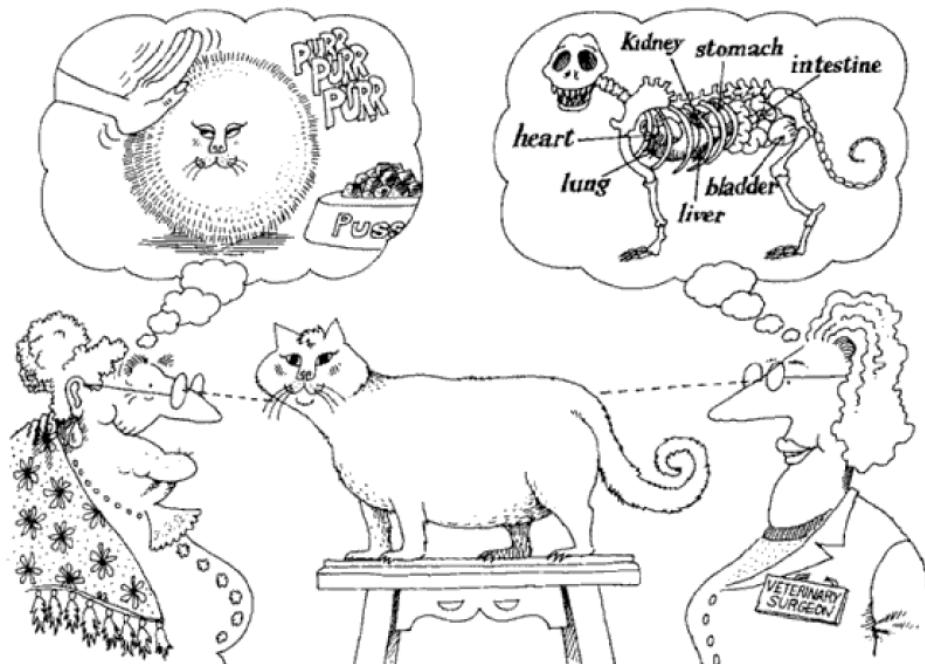
Pillars of OO

- Abstraction
- Encapsulation
- Modularity
- Hierarchy



Abstraction

Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.



!! जाकी रही भावना
जैसी ..
प्रभु मूरत देखी तिन
तैसी !!

Encapsulation

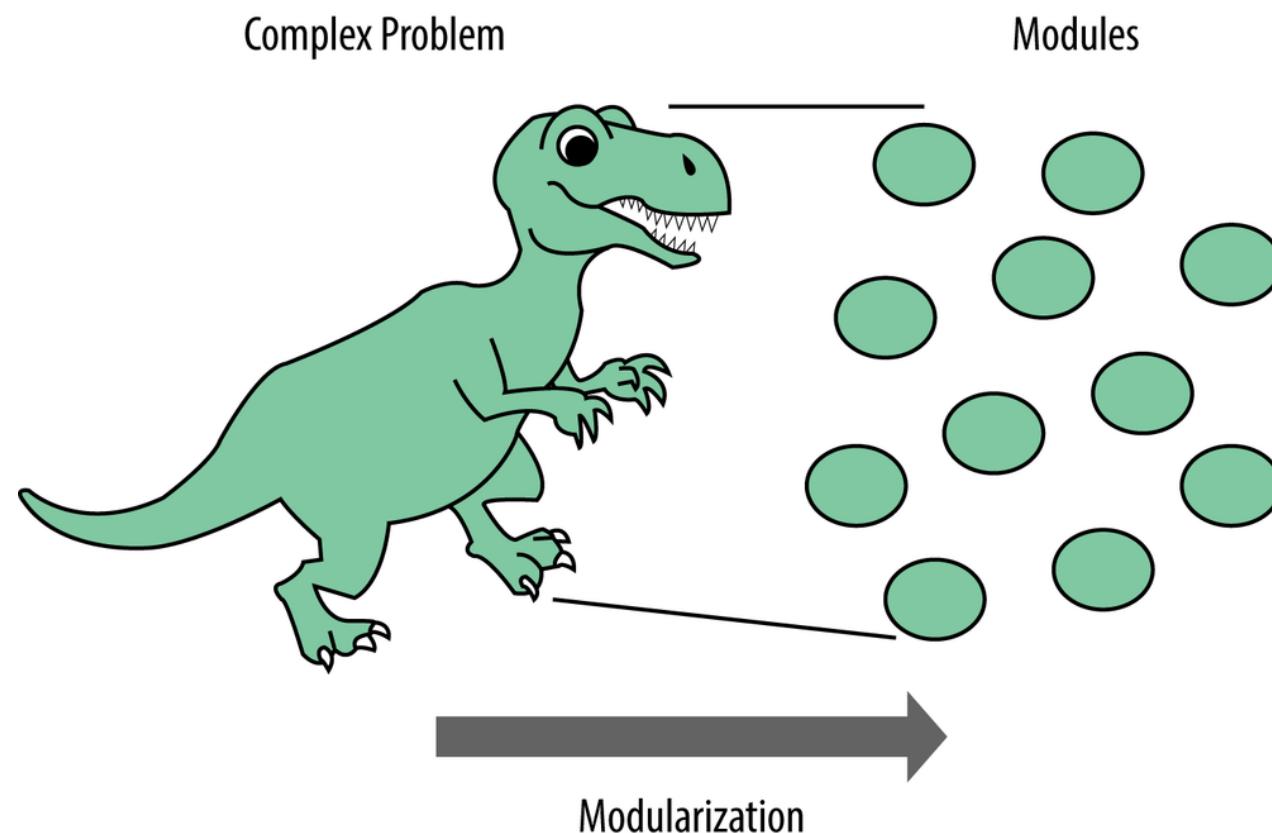
- Changing data in organized way, by using data hiding and applying business constraints
- Encapsulation= data hiding + constraints



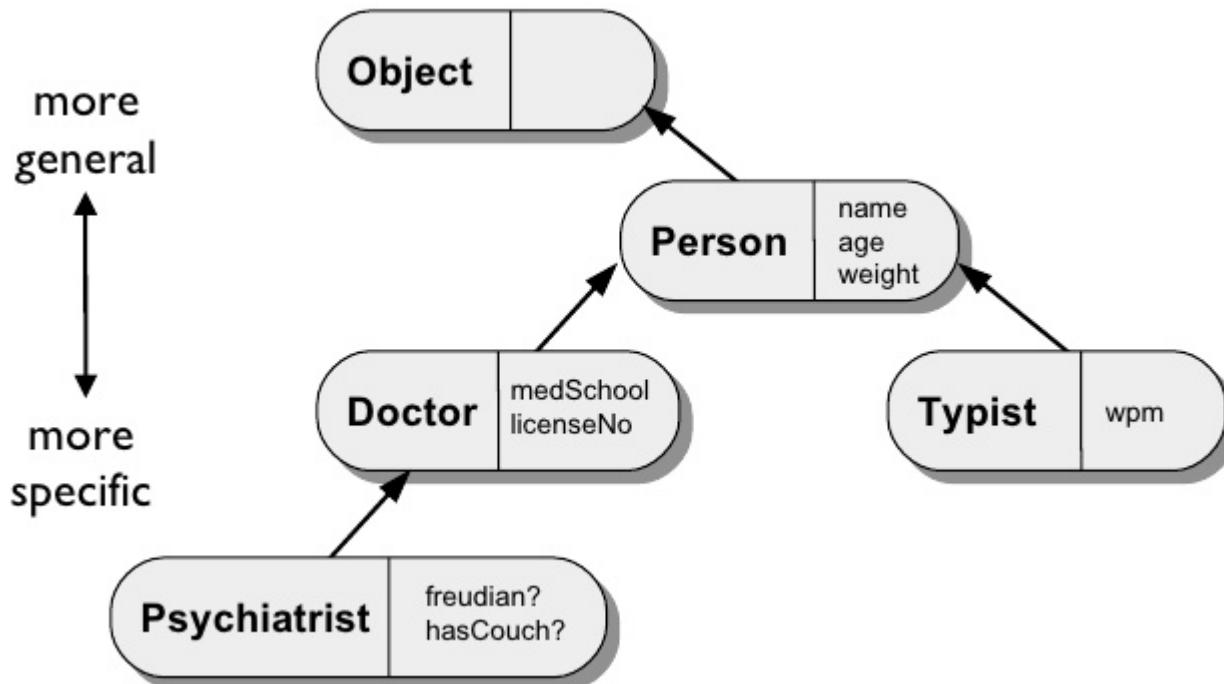
Abstraction vs Abstraction

Abstraction	Encapsulation
Abstraction is a general concept formed by extracting common features from specific examples or The act of withdrawing or removing something unnecessary .	Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse .
You can use abstraction using Interface and Abstract Class	You can implement encapsulation using Access Modifiers (Public, Protected & Private)
Abstraction solves the problem in Design Level	Encapsulation solves the problem in Implementation Level
For simplicity, abstraction means hiding implementation using Abstract class and Interface	For simplicity, encapsulation means hiding data using getters and setters

Modularity



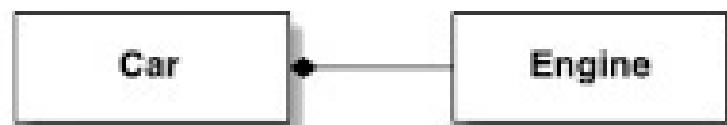
Hierarchy



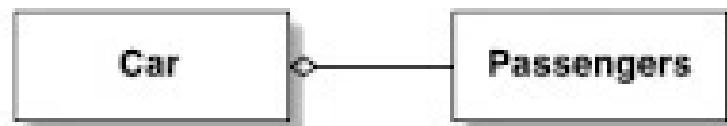
UML 101

UML symbols

Association	Symbol
Composition	◆—————
Aggregation	◇—————
Inheritance	————→
Implementation	-----→

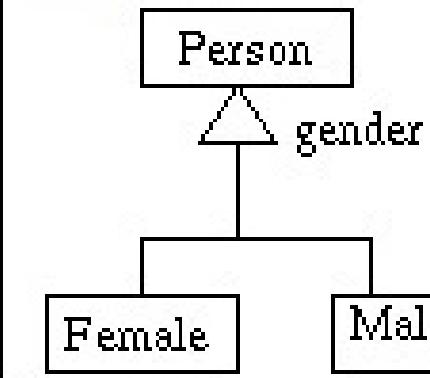


Composition: every car has an engine.

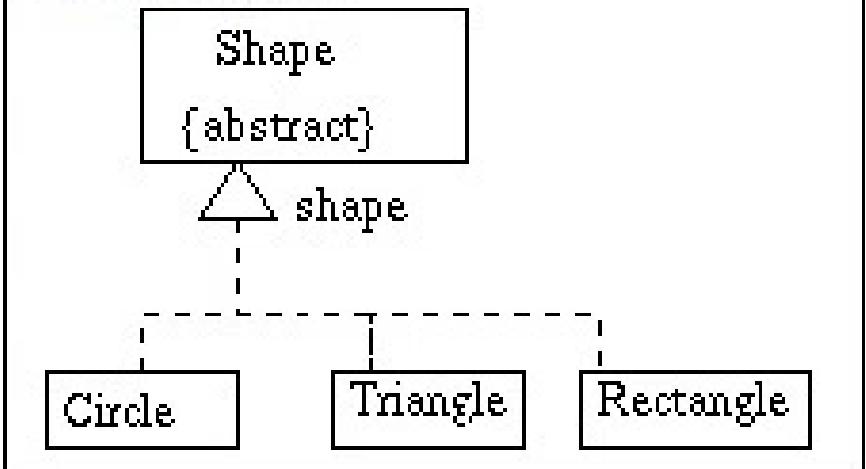


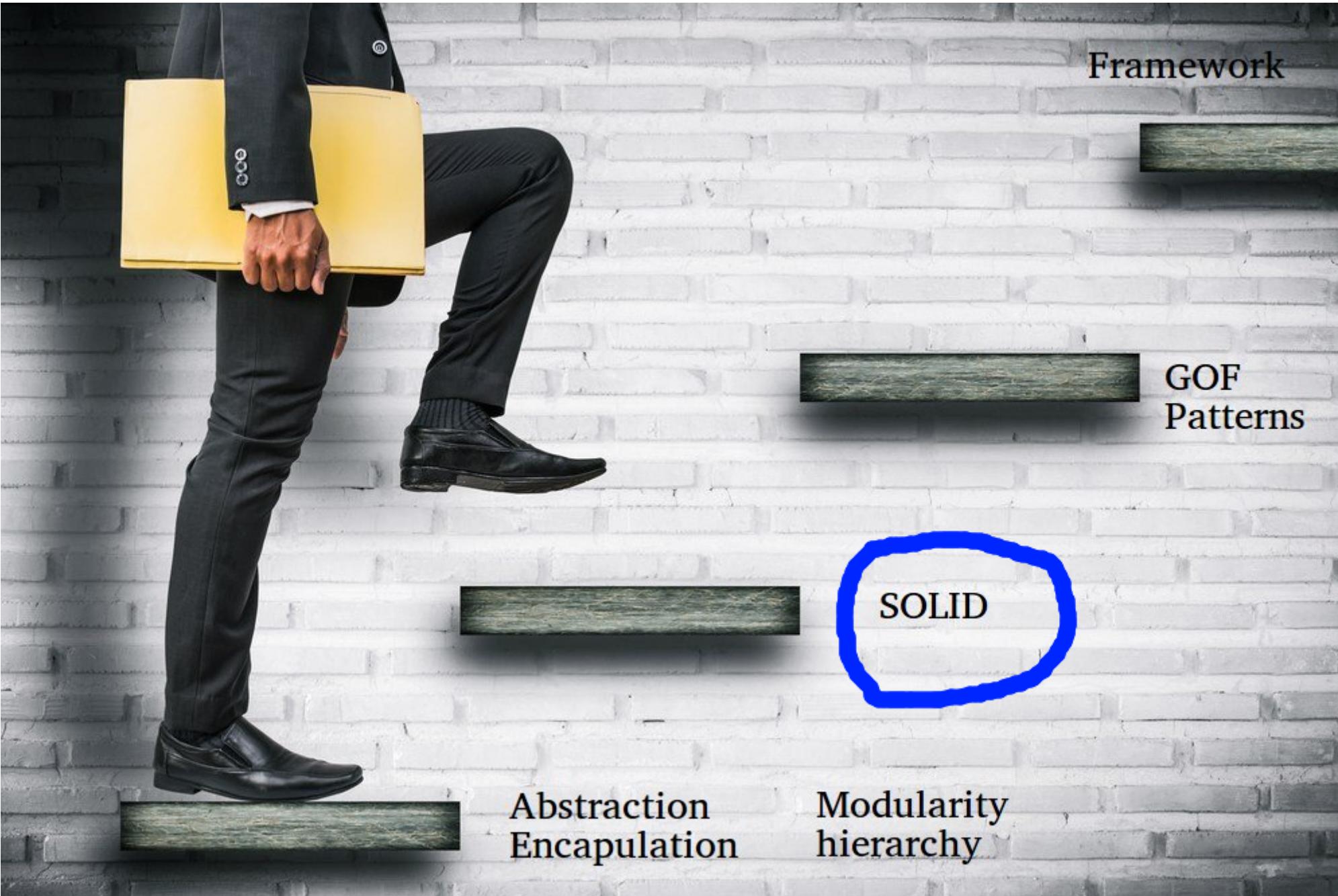
Aggregation: cars may have passengers, they come a

Inheritance



Implementation



A photograph of a man in a dark suit and black leather shoes walking on a grey brick wall. He is carrying a large, bright yellow folder or briefcase in his right hand. The background consists of a white brick wall with two small framed pictures hanging on it.

Framework

GOF
Patterns

SOLID

Abstraction
Encapsulation

Modularity
hierarchy

We write code that is ...

Rigid



Immobile



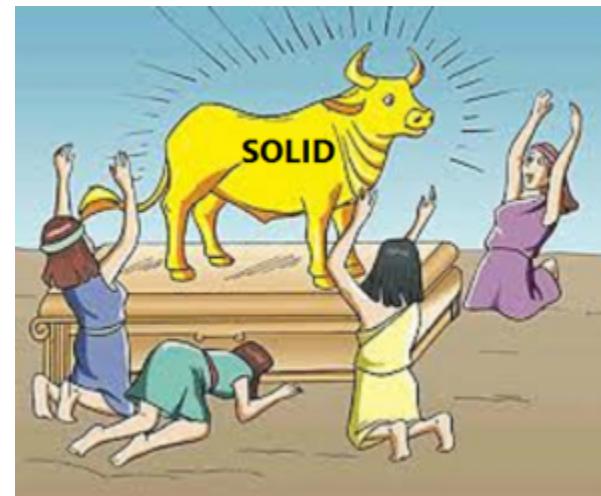
Fragile



Making everyone's life miserable



So WHY SOLID?



It helps us to write code that is ...

- Loosely coupled
- Highly cohesive
- Easily composable
- Reusable

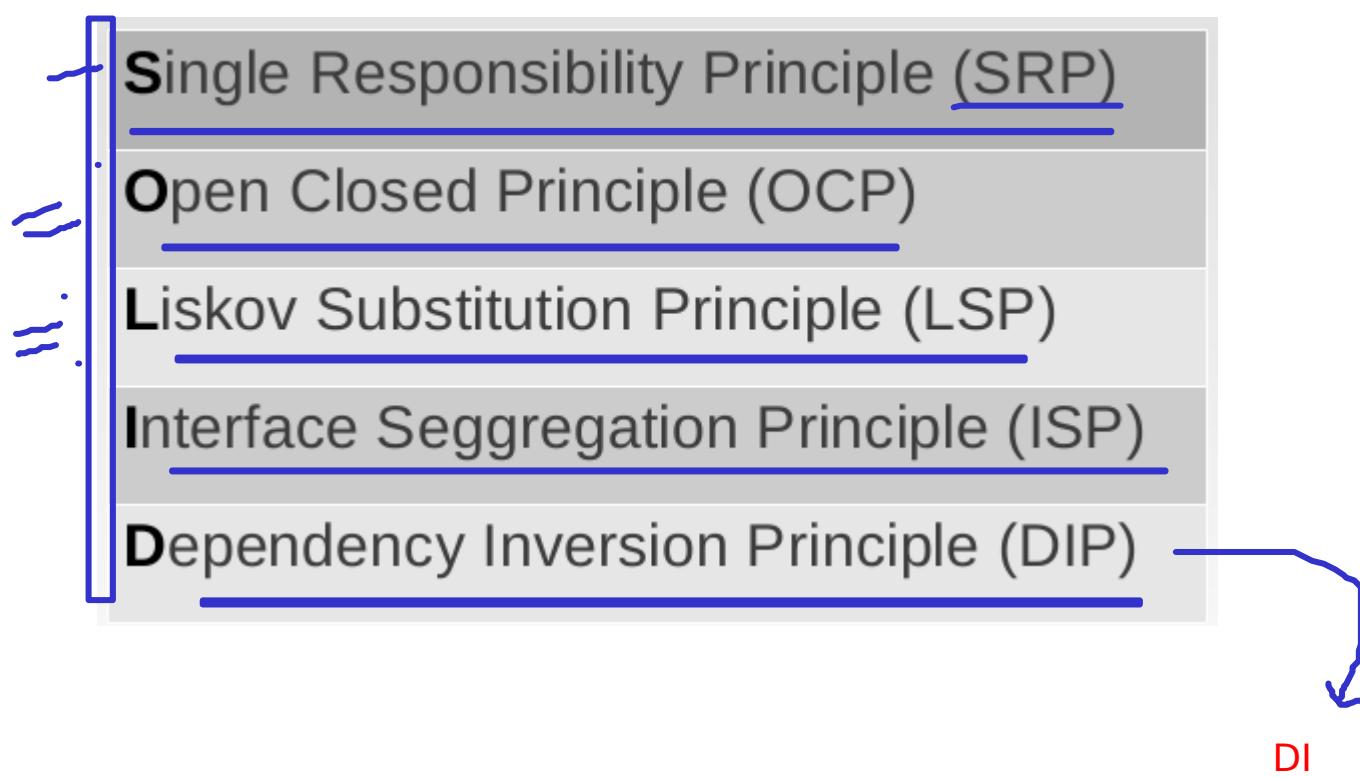
SOLID

Coined by Robert C
Martin

Not new, existing
principles brought
together



SOLID



Single Responsibility Principle



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

"There should be **NEVER** be more than
ONE reason for a class to change"

SOLID

SRP: sw module must have only one reasons to change... ie
sw should not do diff type of operation

God class: is a bad design

```
class Connection{  
    Connection getConnection(){  
    }  
    JmsConnection getConnection(){  
    }  
    HibernateFactory getHibFactory(){  
    }
```

```
class JdbcConnection{  
    Connection getConnection(){  
    }  
}
```

```
class JmsConnection{  
    JmsConnection getConnection(){  
    }  
}
```

```
HibernateFactory getHibFactory(){  
}
```

Open Closed Principle



"Modules must be **OPEN** for extension,
CLOSED for modification"

OCP: A Sw module should be open for extension but close for modification

So that we can achieve loose coupling

if u use polymorphism correctly this is already done

```
class Circle{  
    public void printCircleArea(){  
    }  
}
```

```
class Square{  
    public void printSquareArea(){  
    }  
}
```

```
class Triangle{  
    public void printTriangleArea(){  
    }  
}
```

class AreaProcessor{

```
    public void processArea(Object o){  
        if(o instanceof Triangle){  
            ...  
        }  
    }  
}
```

breaking the pri...
OCP

Liskov Substitution Principle

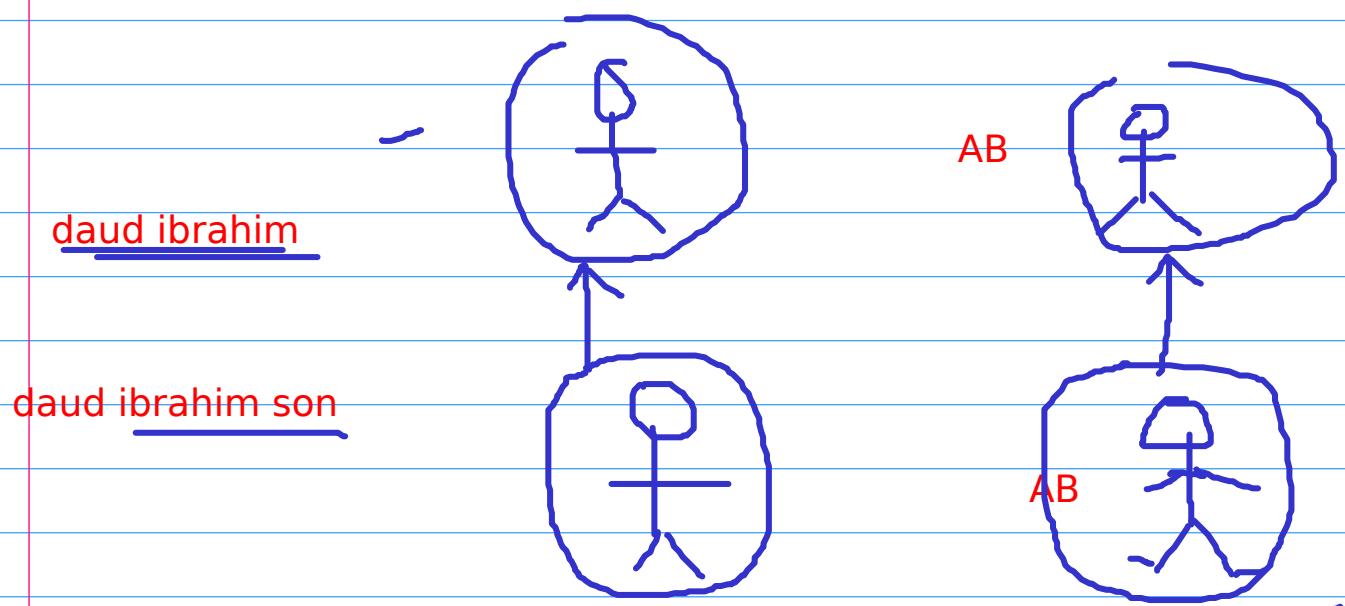


"Base classes instances must be replaceable by the sub class instances without any change in the application"

LSP: liskov Sub pri...

guide us when to go for composition and when to for inheritance

=> if all the methods of base class make sense for derived class
then think of inheritance otherwise think of composition



daud ibrahim son

```
class Bird{  
    void eat(){}
    void fly(){}
    void swim(){}
}
```

4-6pm
test?

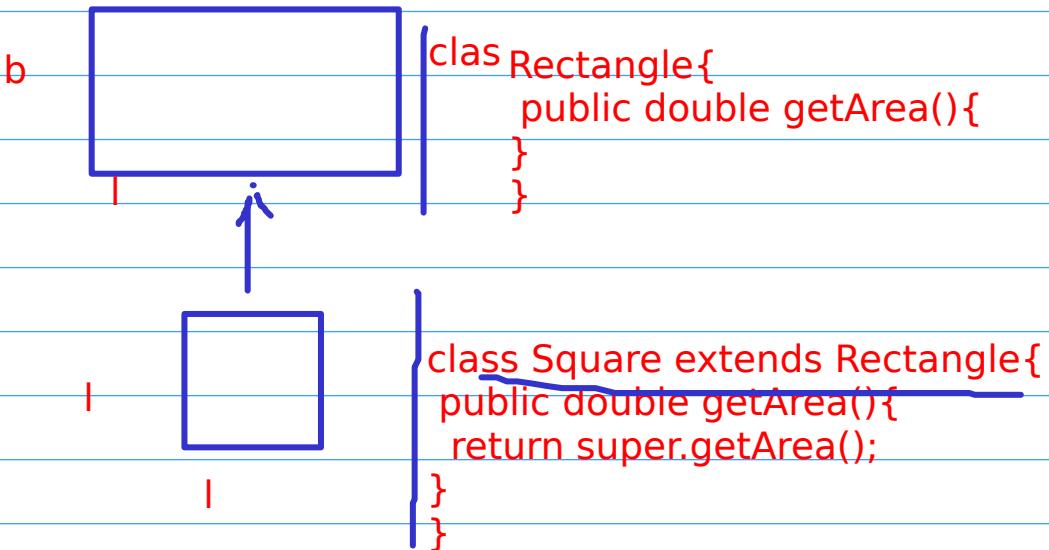
```
class Eagle extends Bird{  
}
```

As one method ie swim() dont
make sense for eagle it
is violation of LSP

some classical example of LSP

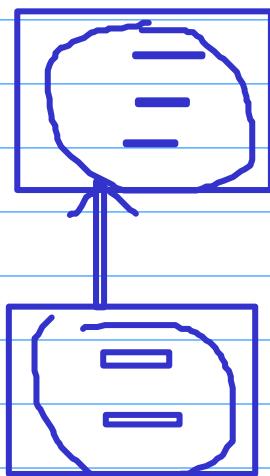
"Square rectangle problem" ==> refer it!

Mathmatically : sq is a special case of rectangle



Why it is not a good code:

As all the functionality of Rectangle dont make sence for Square thats why it is not a valid case of inheritance

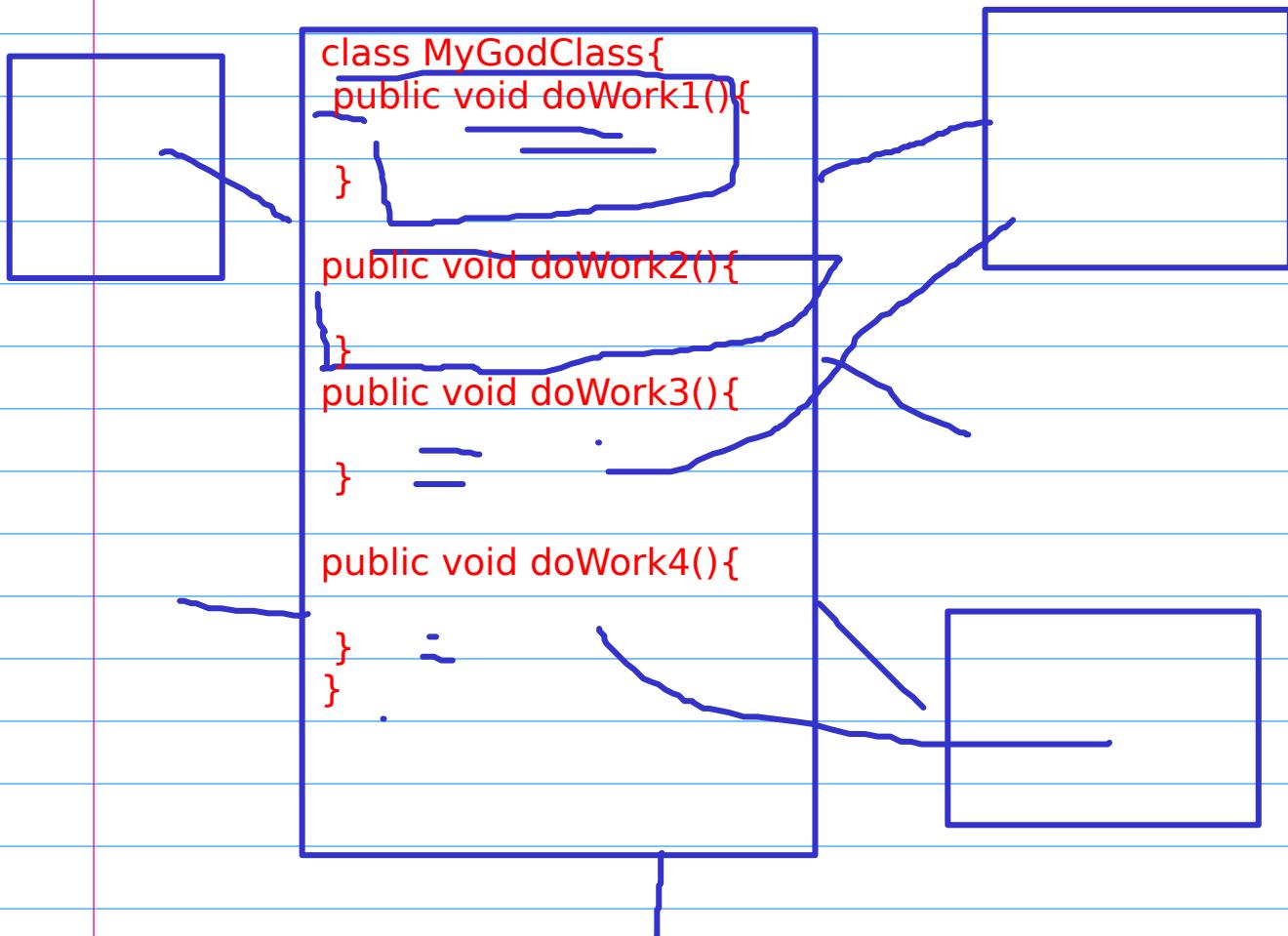


Interface Segregation prin:

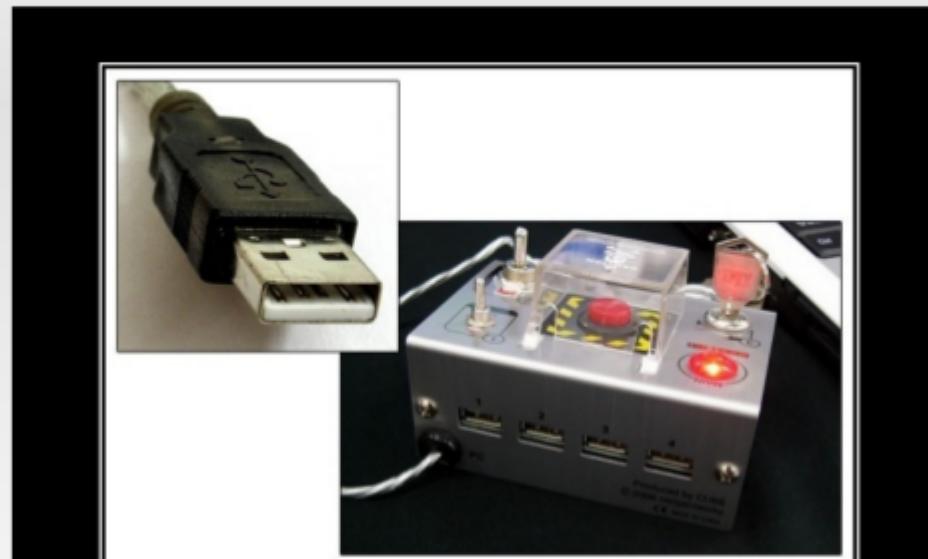
1. interface must be specific

```
interface Foo{  
    public void doWork1();  
    public void doWork2();  
    public void doWork3();  
    public void doWork4();  
}  
  
    }
```

2. God class project



Interface Segregation Principle



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

"Clients should not depend upon the interfaces they do not use"

How can we pollute the interfaces?
OR

How do we end up creating Fat interfaces?

Dependency Inversion Principle



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

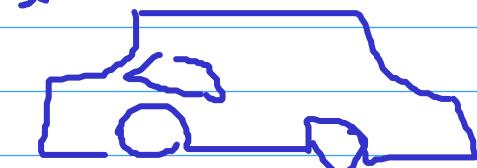
"High level modules should not depend on the low level details modules, instead both should depend on abstractions"

DIP

Dependency

Inversion

Pri...



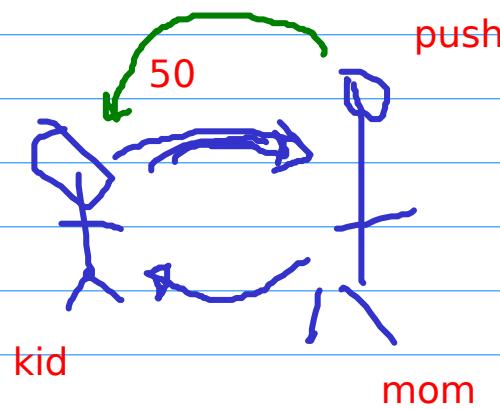
car depend on engine

Tyre

interface Engine{
}



pull: kid need to pull money form mon



Right now i am doing it manually
but it can be done by an external framework: spring freamework

Hollywood:

Car

Engine



Top 10 Object Oriented Design Principles

1. DRY (Don't repeat yourself) - avoids duplication in code.
2. Encapsulate what changes - hides implementation detail, helps in maintenance
3. Open Closed design principle - open for extension, closed for modification
4. SRP (Single Responsibility Principle) - one class should do one thing and do it well
5. DIP (Dependency Inversion Principle) - don't ask, let framework give to you
6. Favor Composition over Inheritance - Code reuse without cost of inflexibility
7. LSP (Liskov Substitution Principle) - Sub type must be substitutable for super type
8. ISP (Interface Segregation Principle) - Avoid monolithic interface, reduce pain on client side
9. Programming for Interface - Helps in maintenance, improves flexibility
10. Delegation principle - Don't do all things by yourself, delegate it



Framework



GOF
Patterns



SOLID

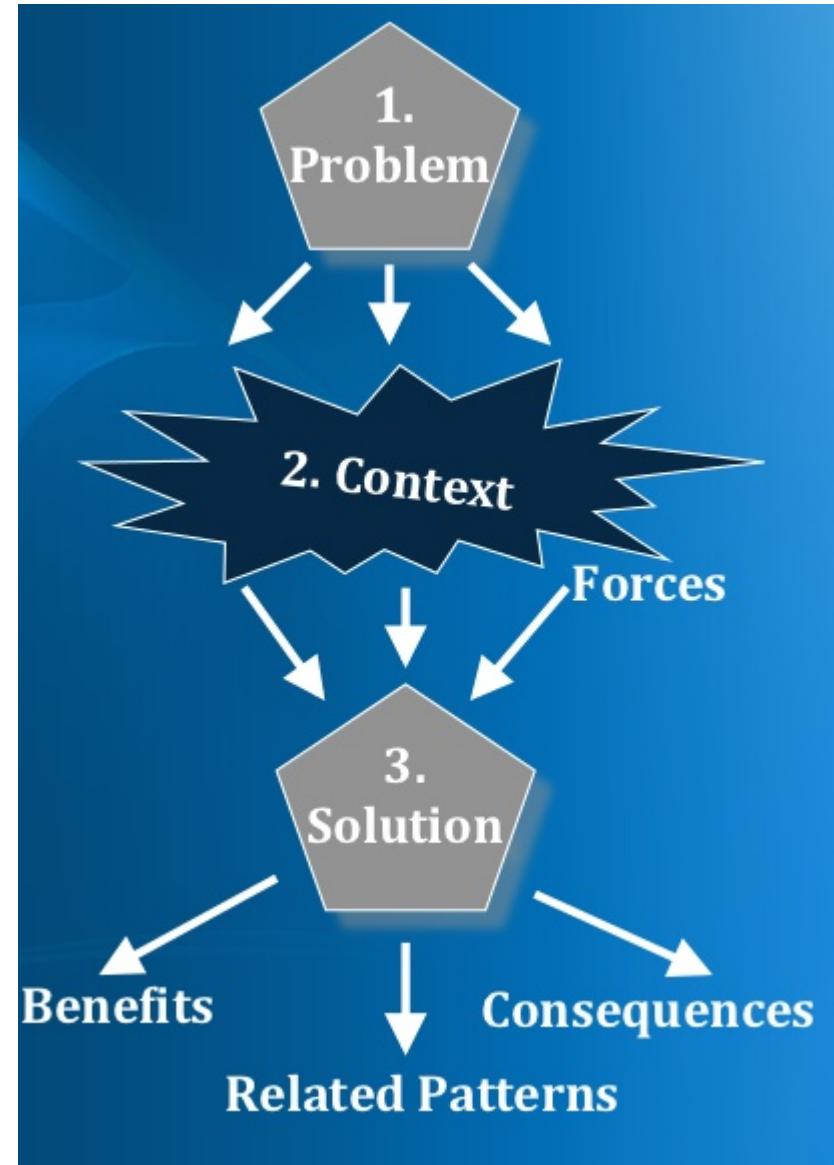
Abstraction
Encapsulation

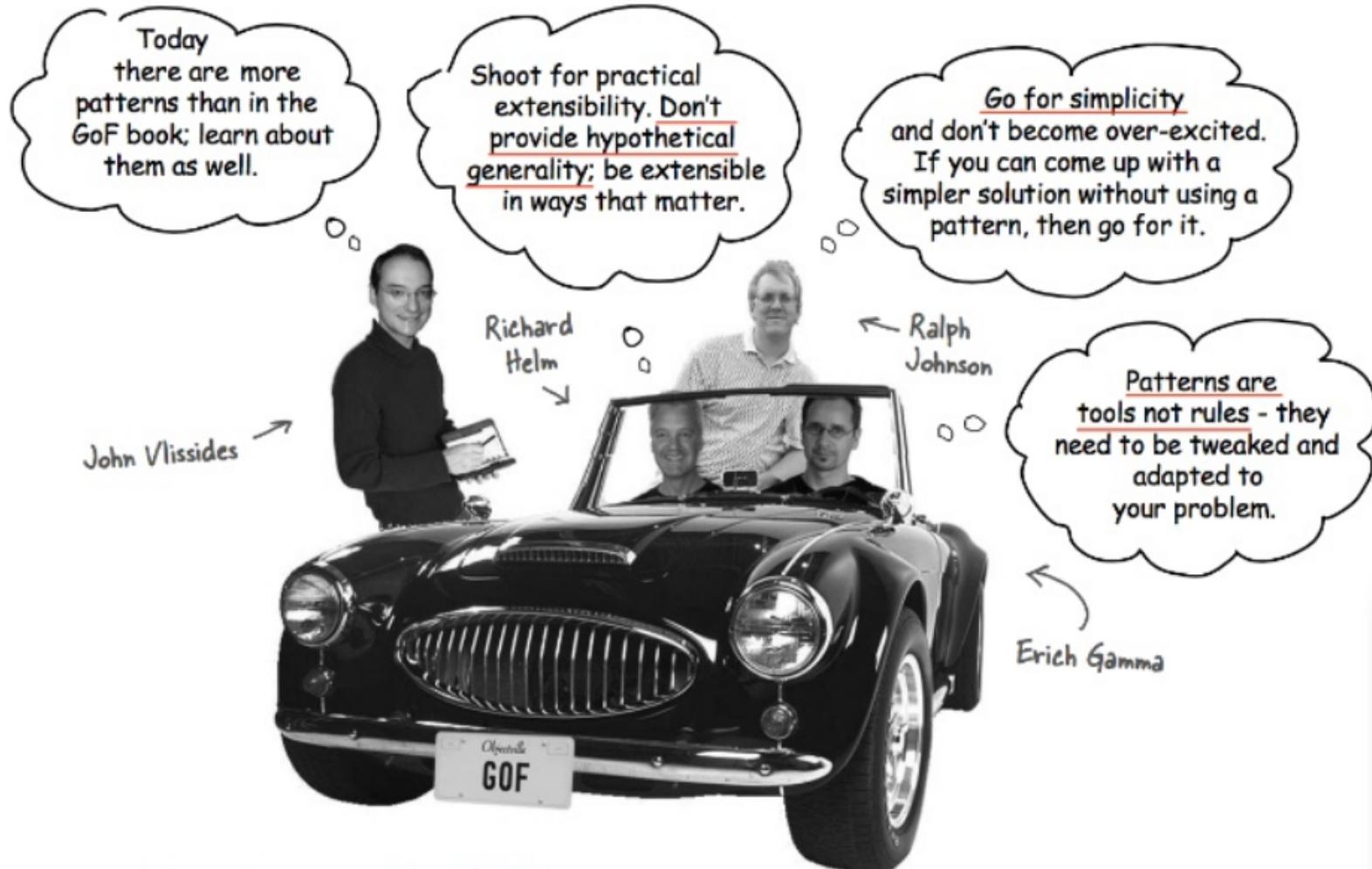
Modularity
hierarchy



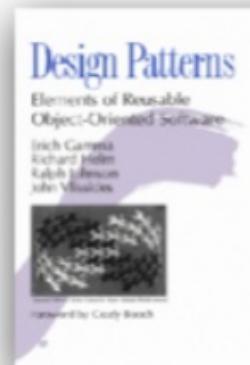
Design pattern

- Proven way of doing things
- **Gang of 4 design patterns ???**
- **total 23 patterns**
- **Classification patterns**
 - 1. **Creational**
 - 2. **Structural**
 - 3. **Behavioral**





Keep it simple (KISS)



DESIGN PATTERNS – CLASSIFICATION

Structural Patterns	Creational Patterns	Behavioral Patterns
<ul style="list-style-type: none">• 1. Decorator• 2. Proxy• 3. Bridge• 4. Composite• 5. Flyweight• 6. Adapter• 7. Facade	<ul style="list-style-type: none">• 1. Prototype• 2. Factory Method• 3. Singleton• 4. Abstract Factory• 5. Builder	<ul style="list-style-type: none">• 1. Strategy• 2. State• 3. TemplateMethod• 4. Chain of Responsibility• 5. Command• 6. Iterator• 7. Mediator• 8. Observer• 9. Visitor• 10. Interpreter• 11. Memento

Patterns classification

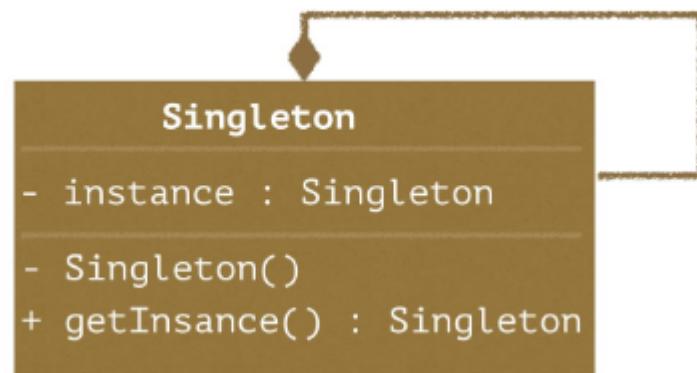
- **Creational patterns?**
 - What is the **best way to create object** as per requirement
- **Structural patterns?**
 - Structural Patterns describe **how objects and classes can be combined to form larger structures**
- **Behavioral Patterns?**
 - Behavioral patterns are those which are concerned with **interactions between the objects** (talking to each other still loosely coupled)



**Creational
Design
Patterns**

Singleton Design Pattern

“Ensure that a class has only one instance and provide a global point of access to it.”



- `java.lang.Runtime#getRuntime()`
- `java.awt.Desktop#getDesktop()`
- `java.lang.System#getSecurityManager()`

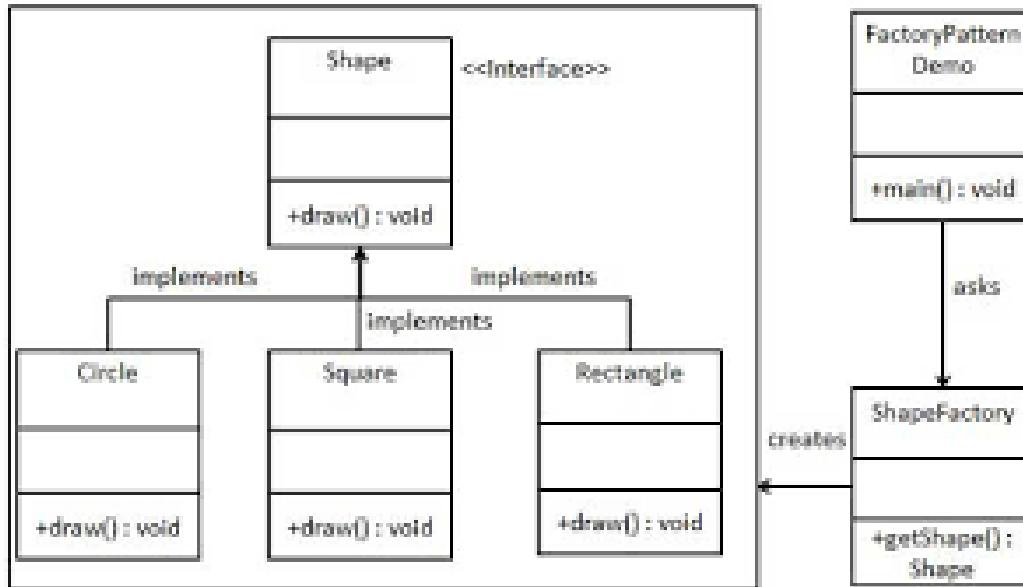
- ❖ Java Runtime class is used to *interact with java runtime environment*.
- ❖ Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc.
- ❖ There is only one instance of `java.lang.Runtime` class is available for one java application.
- ❖ The `Runtime.getRuntime()` method returns the singleton instance of `Runtime` class.

Singleton Design Consideration

- # Eager initialization
- # Static block initialization
- # Lazy Initialization
- # Thread Safe Singleton
- # Serialization issue
- # Cloning issue
- # Using Reflection to destroy Singleton Pattern.
- # Enum Singleton
- # Best programming practices



Factory design pattern



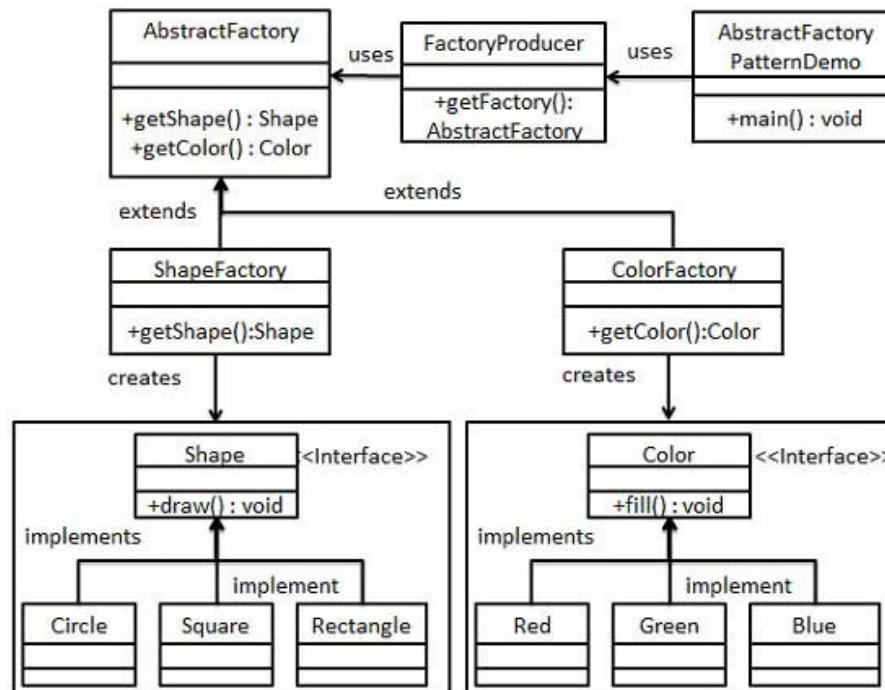
Factory(Simplified version of Factory Method) - Creates objects without exposing the instantiation logic to the client and Refers to the newly created object through a common interface.

Factory Method - Defines an interface for creating objects, but let subclasses to decide which class to instantiate and Refers to the newly created object through a common interface.

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`

Abstract Factory

Java: Abstract Factory. **Abstract Factory** is a **creational design pattern**, which solves the problem of creating entire product families without specifying their concrete classes. **Abstract Factory** defines an interface for creating all distinct products, but leaves the actual product creation to concrete **factory** classes.



- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

Builder Pattern

The Builder pattern can be used to ease the construction of a complex object from simple objects.



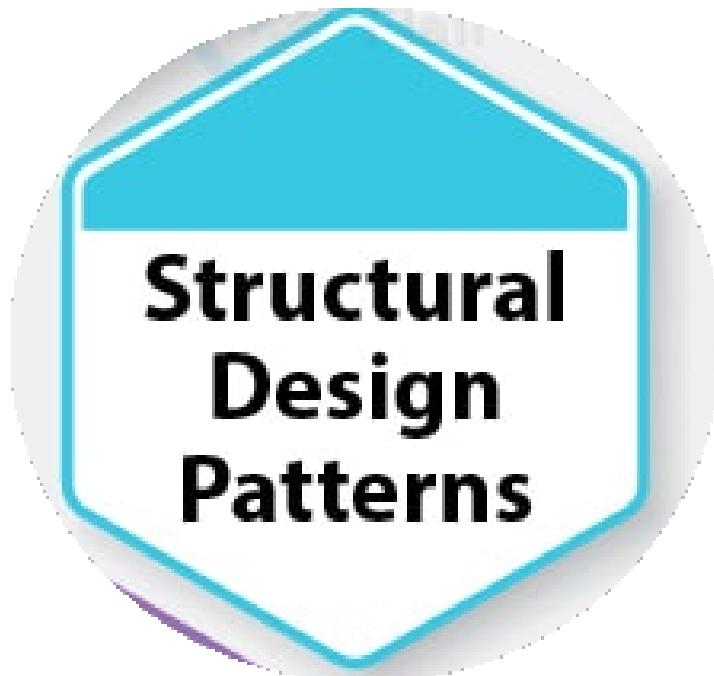
`java.lang.StringBuilder#append()` (unsynchronized)
`java.lang.StringBuffer#append()` (synchronized)

Prototype Pattern

Cloning of an object to avoid creation. If the cost of creating a new object is large and creation is resource intensive, we clone the object.



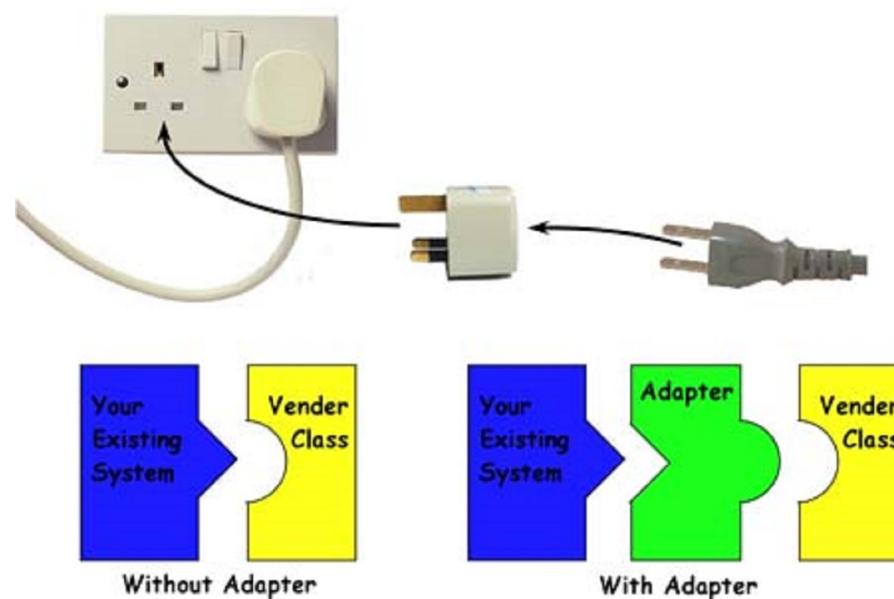
- `java.lang.Object#clone()` (the class has to implement `java.lang.Cloneable`)



**Structural
Design
Patterns**

Adapter Pattern

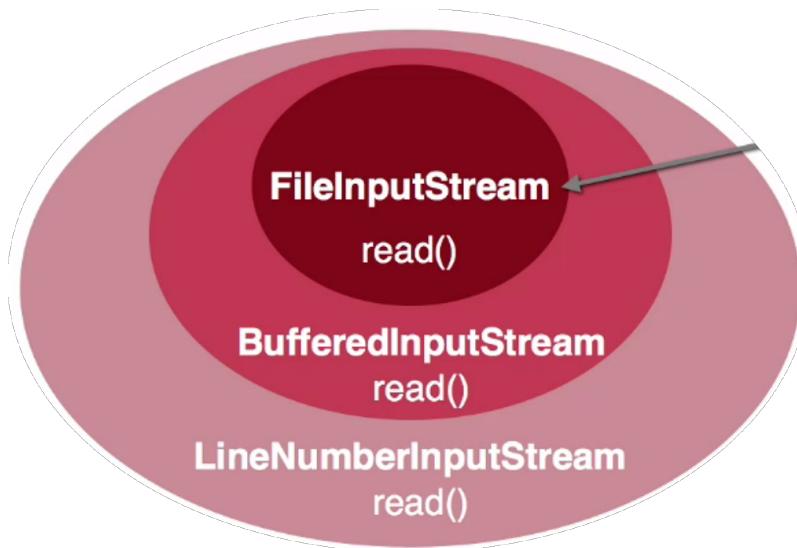
The Adapter pattern is used so that two unrelated interfaces can work together.



- `java.util.Arrays#asList()`
- `java.util.Collections#list()`
- `java.util.Collections#enumeration()`
- `java.io.InputStreamReader(InputStream)` (returns a `Reader`)
- `java.io.OutputStreamWriter(OutputStream)` (returns a `Writer`)

Decorator design pattern

Series of wrapper class that define functionality, In the Decorator pattern, a decorator object is wrapped around the original object.



Adding behaviour statically or dynamically
Extending functionality without effecting the behaviour of other objects.
Adhering to Open for extension, closed for modification.

All subclasses of `java.io.InputStream`, `OutputStream`, `Reader` and `Writer` have a constructor taking an instance of same type.

`java.util.Collections`, the `checkedXXX()`, `synchronizedXXX()` and `unmodifiableXXX()` methods.

`javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

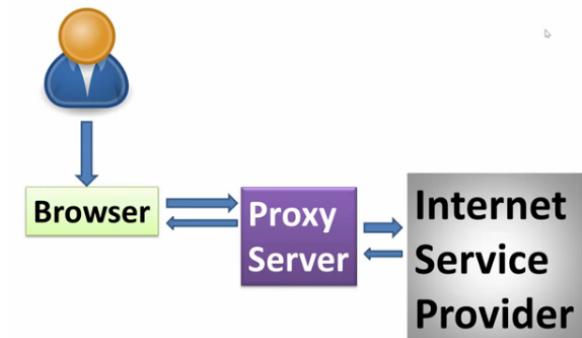
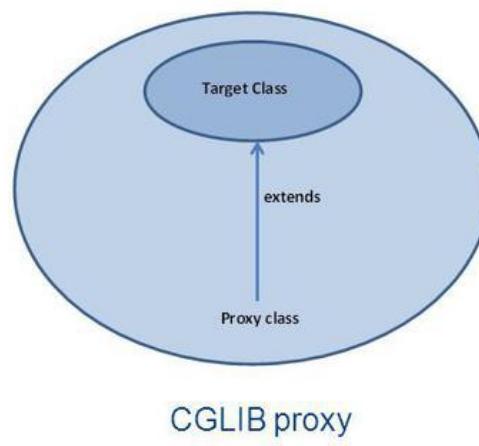
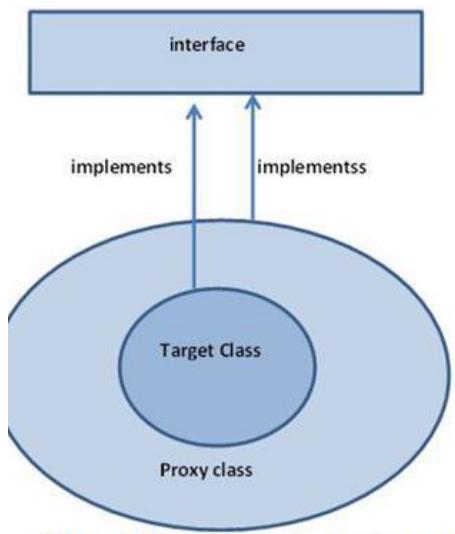
`javax.swing.JScrollPane`

Proxy design pattern

Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.



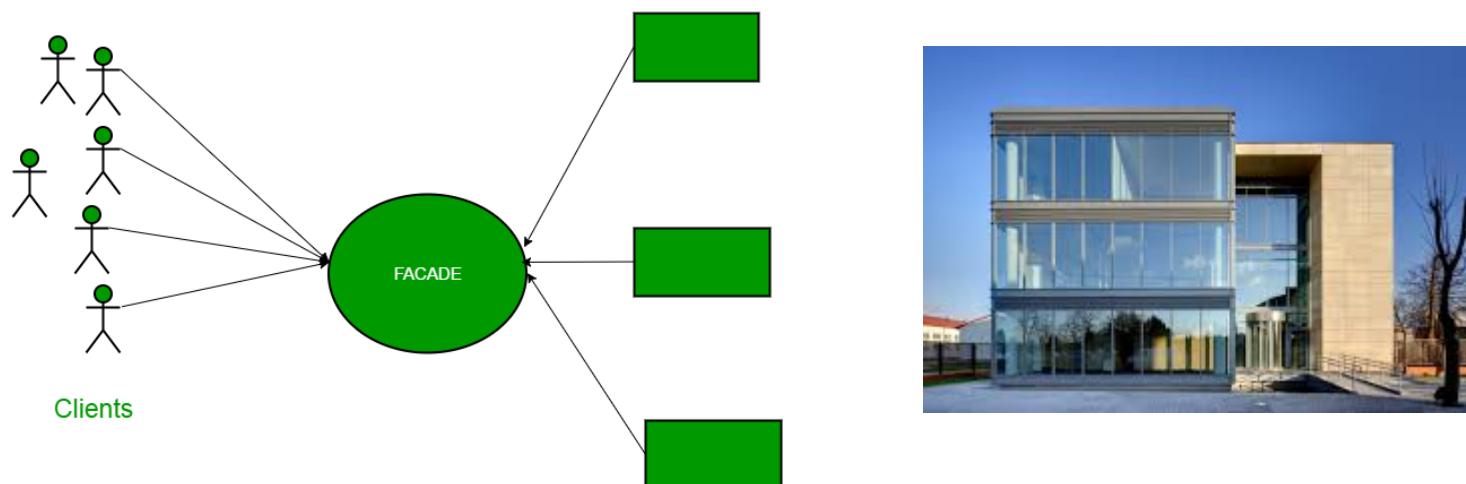
- In Spring Framework AOP is implemented by creating proxy object for your service.



`java.lang.reflect.Proxy`
`java.rmi.*`
`javax.ejb.EJB` (explanation here)
`javax.inject.Inject` (explanation here)
`javax.persistence.PersistenceContext`

Facade pattern

The **facade pattern** (also spelled as **façade**) is a **software-design pattern** commonly used with object-oriented programming. The name is an analogy to an architectural **façade**. A **facade** is an object that provides a simplified interface to a larger body of code, such as a class library.



- `javax.faces.context.FacesContext`, it internally uses among others the abstract/interface types `LifeCycle`, `ViewHandler`, `NavigationHandler` and many more without that the enduser has to worry about it (which are however overrideable by injection).
- `javax.faces.context.ExternalContext`, which internally uses `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse`, etc.



Behavioral Design Patterns

Iterator design pattern

"The Iterator

Pattern provides a way to access the elements of an aggregate object sequentially without exposing the underlying representation"

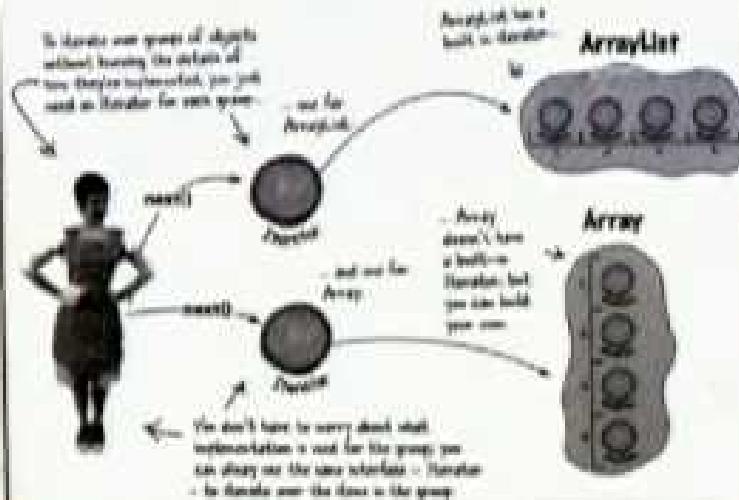
Head First
Design Patterns
Poster
O'Reilly,
ISBN 0-596-10214-3



336, 257

Iterator

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Strategy pattern /policy pattern

Strategy - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The screenshot shows a shopping cart interface with two items: a pink shirt and a blue blouse. The right side features an 'Order Summary' section with a subtotal of \$94.50 and a promotional offer for a 50% discount on shoes. The bottom right corner includes a 'Need Help?' section with customer support links.

ITEM	PRICE	QUANTITY	TOTAL
Lana Leopard Lace Tee Style: 570113309 SKU: 451004831976 Color: Summerberry Size: Size 1 (8/10, S)	\$55.00	1	\$55.00
Easy Cotton Tyree Shirt Style: 570105245 SKU: 451004495130 Color: Mysterious Blue Size: Size 1.5 (10, S)	\$39.50	1	\$39.50

Order Summary

ITEM SUBTOTAL	\$94.50
ESTIMATED TOTAL (BEFORE TAX) \$94.50	

Promotion Code **APPLY**

The two best words ever?
shoe SALE!
50% OFF
Select Styles [> SHOP THE SALE](#) *Details

Need Help?

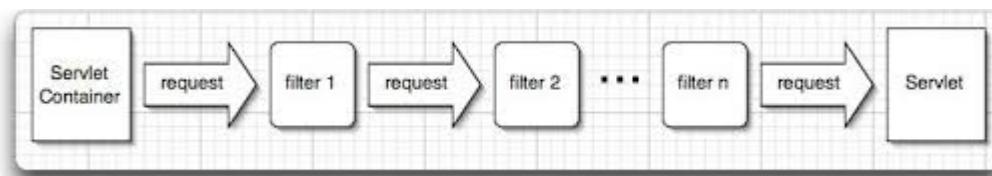
We're happy to offer international shoppers with English Customer Support!

[CLICK TO CHAT](#) [CLICK TO CALL](#)

- `java.util.Comparator#compare()`, executed by among others `Collections#sort()`.
- `javax.servlet.http.HttpServlet`, the `service()` and all `doXXX()` methods take `HttpServletRequest` and `HttpServletResponse` and the implementor has to process them (and not to get hold of them as instance variables!).

Chain of Responsibility

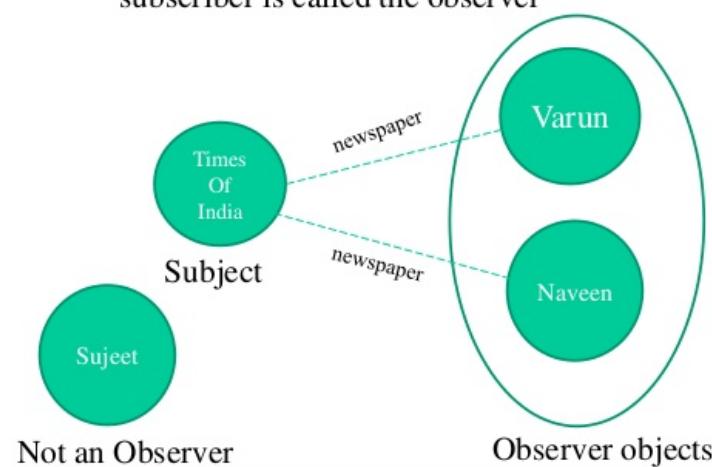
Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them.



Observer Design pattern

Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change.

- Publisher + Subscribers = observer pattern
- In observer pattern publisher is called the subject and subscriber is called the observer



- `java.util.Observer / java.util.Observable` (rarely used in real world though)
- All implementations of `java.util.EventListener` (practically all over Swing thus)
- `javax.servlet.http.HttpSessionBindingListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.faces.event.PhaseListener`

Template Design Pattern

Template Method - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses / Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.

JONNY LEVER PARCEL INTERNATIONAL (Private) LIMITED
78, Bungalow Complex, Main Road, Sector 84, Panchkula, Haryana - 164102, India
Phone: +91 172 5101000 | Fax: +91 172 5101010
[Email: info@jlpinternational.com](mailto:info@jlpinternational.com) | Website: www.jlpinternational.com

Customer Feedback Form

Your valuable feedback is very important to us. Your feedback will help us to improve better and faster.

Customer Name: [Visitor Customer Name Here]

Address: [Visitor Address Here]

Destination: [Visitor Here]

Amount: [Visitor Here]

Feedback: [Visitor Feedback Here]

Excellent Good Fair Poor

1. Supplier's management and assistance
2. Order's Punctuality
3. How would you rate the quality of our product?
4. Staff's performance and attitude
5. Delivery of order without damage/defect
6. Address & Distance of delivery
7. How will you rate our overall quality of our packing and moving?
8. How would you like to recommend us to others?

Your comments: _____

Signature: _____ Date: September 03, 2013



spring
JDBC

Pattern vs. Framework

- Pattern is a set of guidelines on how to architect the application
- When we implement a pattern we need to have some classes and libraries
- **Thus, pattern is the way you can architect your application.**
- Framework helps us to follow a particular pattern when we are building a web application
- These prebuilt classes and libraries are provided by the MVC framework.
- **Framework provides foundation classes and libraries.**



Library vs Framework

