# Object Orientation, SOLID and GOF deep dive

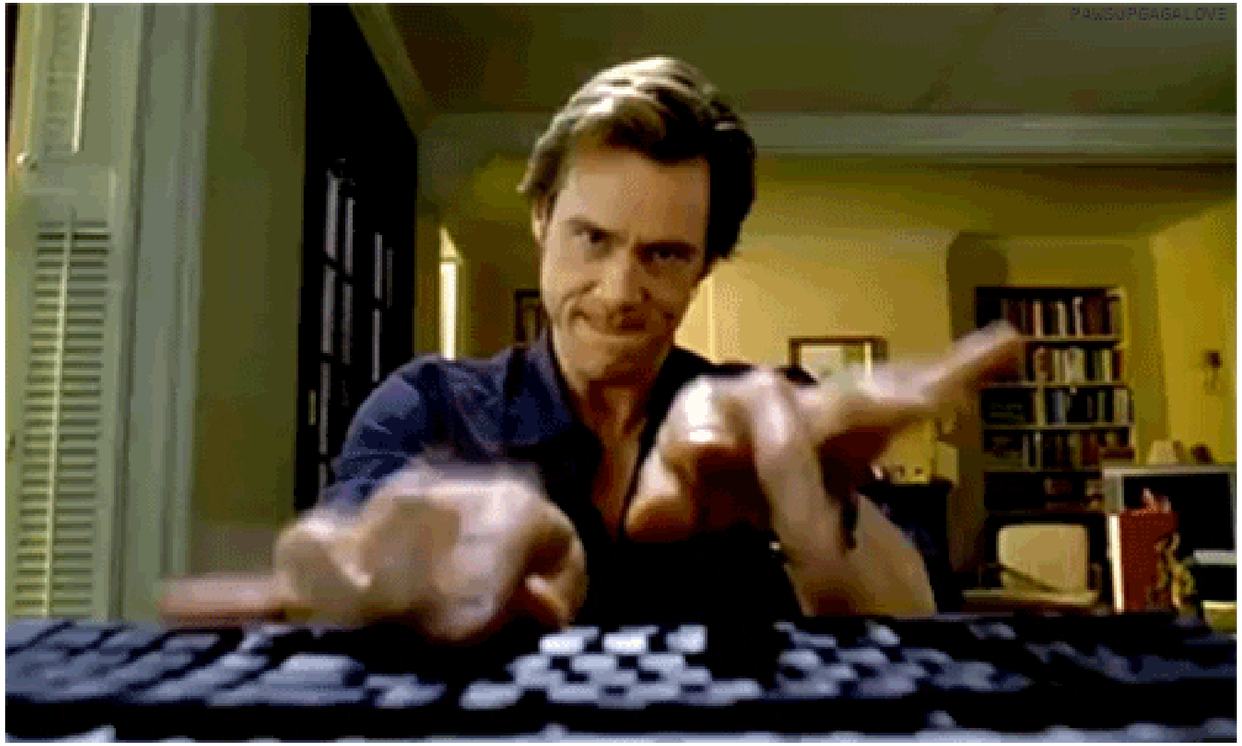
**Rajeev Gupta**

rgupta.mtech@gmail.com

https://www.linkedin.com/in/rajeevguptajavatrainer

rgupta.mtech@gmail.com

# Programmer are Not typist but thinker



rgupta.mtech@gmail.com

# Learning OOPs, SOLID, Design Patterns step by step

MVC

Concurrny

J2EE

EIP

Proven way of doing thing

**GOF design pattern**

Core pattern

(23 pattern
not readymade -> design )

Rx

5th class

clear understanding of OOPs concepts

10th class

**SOLID**

12th class

**Design patterns**

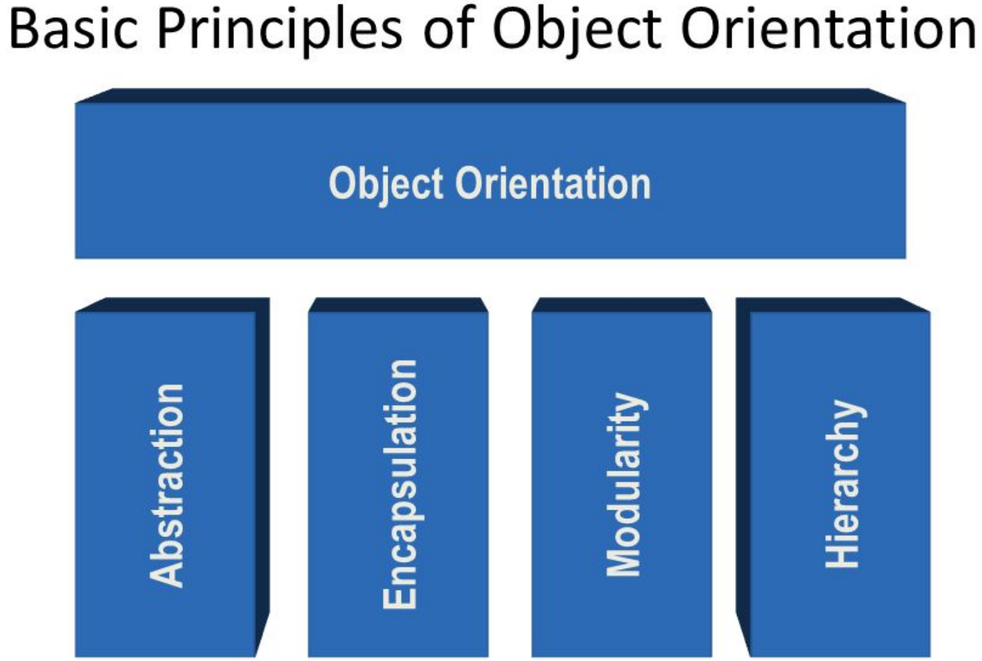
BTech/MTech

**Lib & Framework**

rgupta.mtech@gmail.com

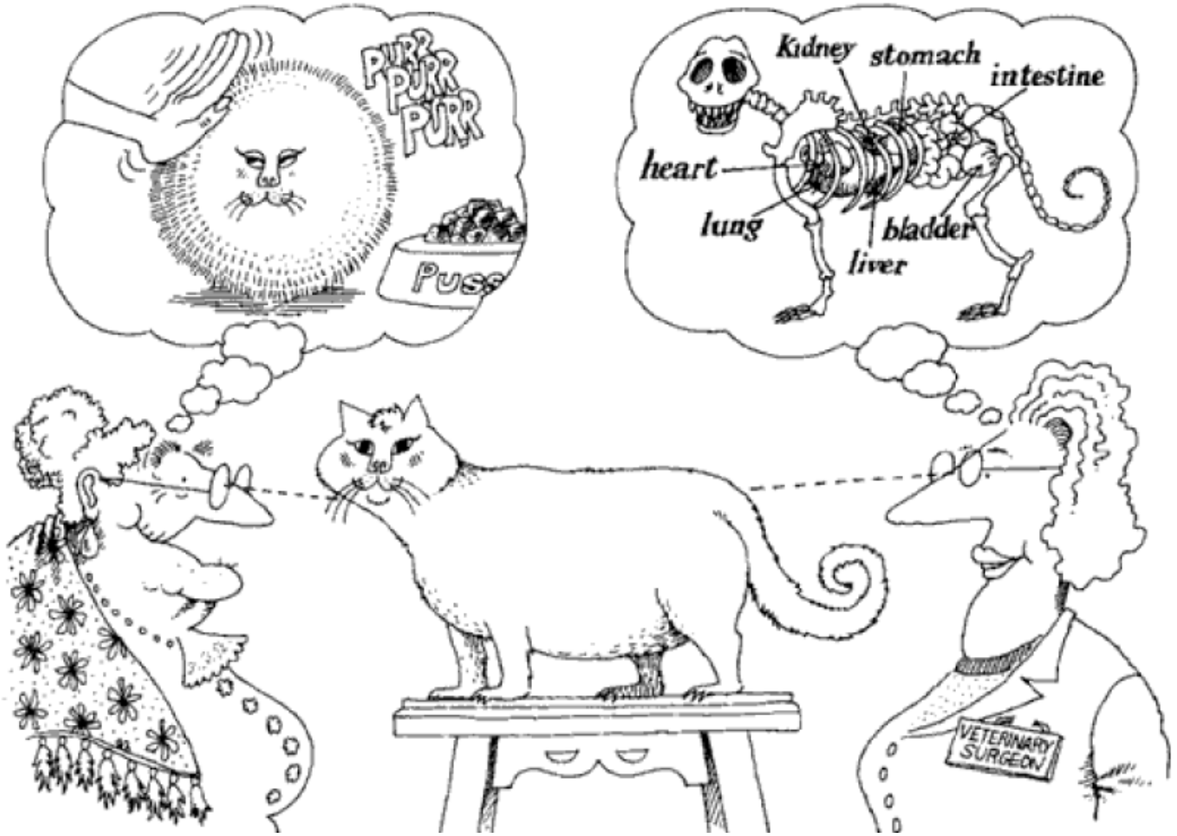
# Pillars of object oriented programmin

Abstraction
Encapsulation
Modularity
Hierarchy

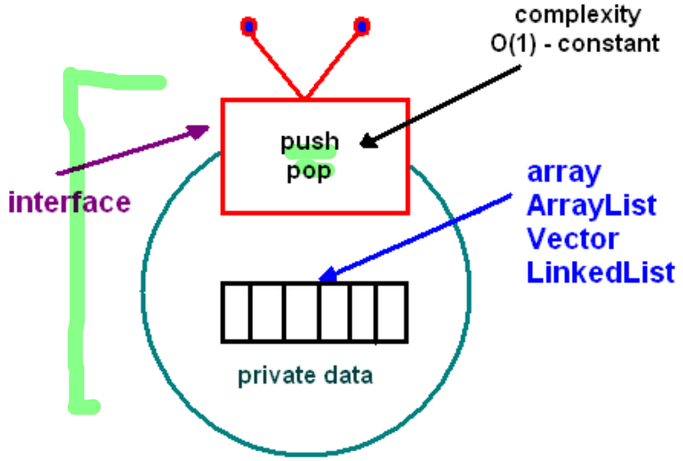



rgupta.mtech@gmail.com

# Abstraction




**Abtraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer**



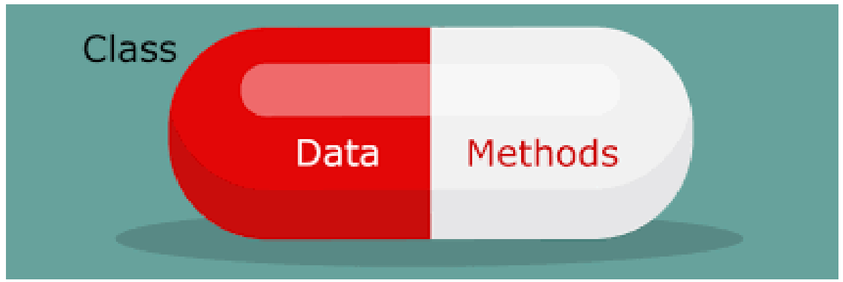

rgupta.mtech@gmail.com

# Encapsulation

**How to implement Encapsulation provide private visibility to an instance variable**

```
class Account{
private int account_number;
private int account_balance;

public void show Data(){
// code to show data
}

public void deposit(int a){
```

**Encapsulation the process of restructuring access to inner implementation details of a class. Internal not exposed to the outside world For example, sealed mobile, if you open it guarantee is violated**




rgupta.mtech@gmail.com

# Abstraction vs Encapsulation

**Abstraction** is the Design principle of separating interface (not java keyword) from implementation so that the client only concern with the interface

Abstraction in Java programming can be achieved using an interface or abstract class

Abstraction and Encapsulation are complimentary concepts

**Encapsulation** the process of restructuring access to inner implementation details of a class.

Internal not exposed to the outside world For example, sealed mobile, if you open it guarantee is violated

How to implement Encapsulation provide private visibility to an instance variable

Ability to refactor/change internal code without breaking other client code


Abstraction and Encapsulation are complementary concepts. Through encapsulation only we are able to enclose the components of the object into a single unit and separate the private and public members. It is through abstraction that only the essential behaviors of the objects are made visible to the outside world.

rgupta.mtech@gmail.com

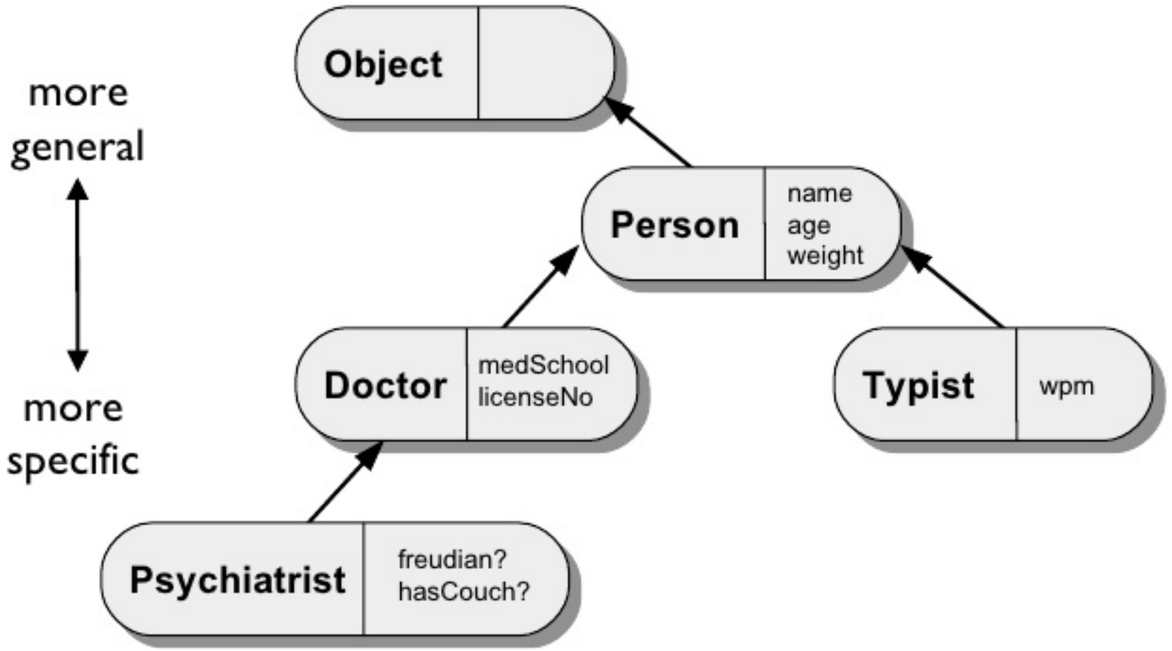
# Abstraction vs Encapsulation

| | |
|---|---|
| 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 |

rgupta.mtech@gmail.com

# Hierarchy

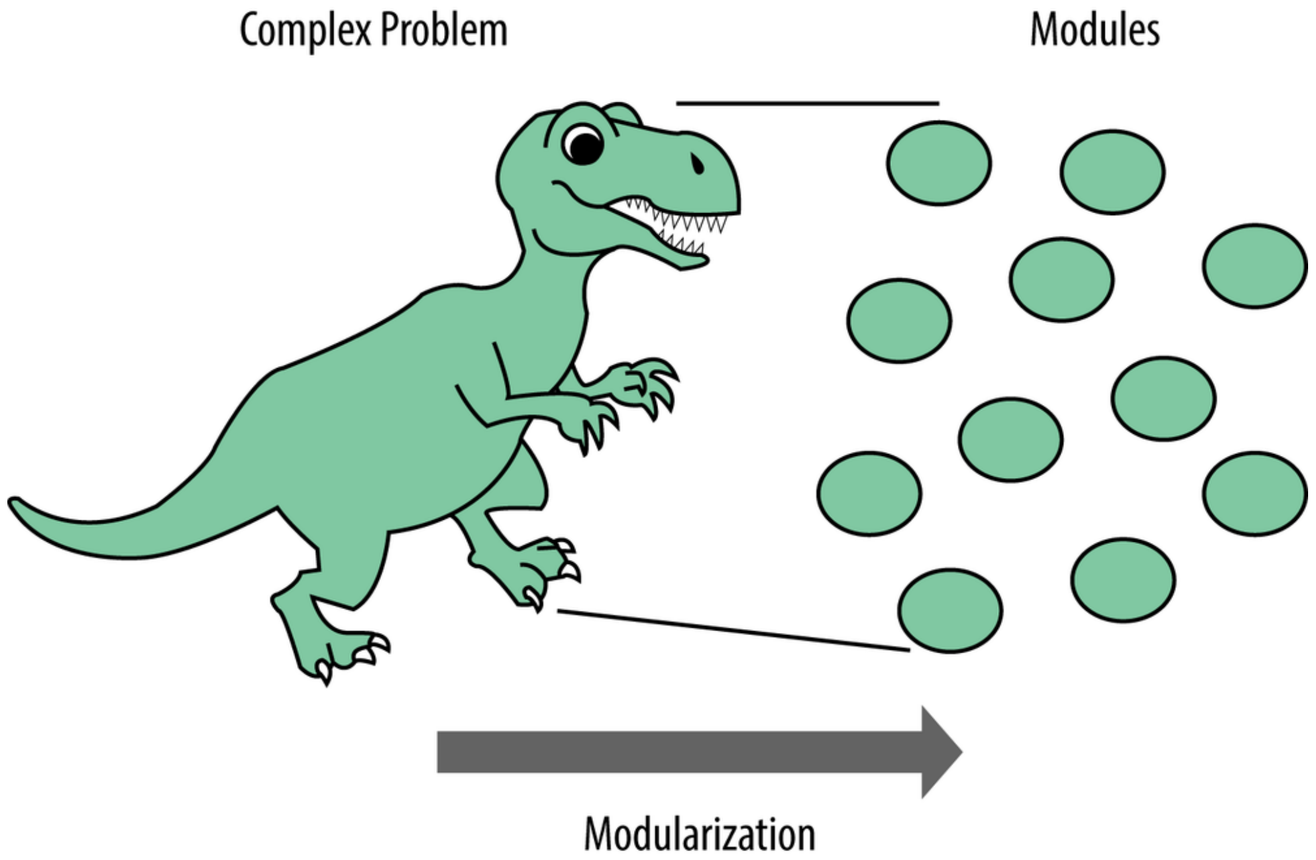
**Hierarchy give rise to inheritance and polymorphism**




rgupta.mtech@gmail.com

# Modularity

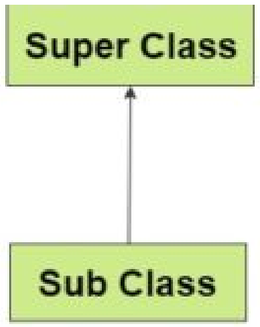
**When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods. An inherited class is defined by using the extends keyword**

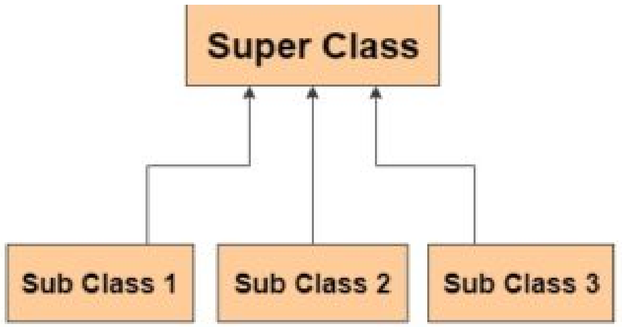


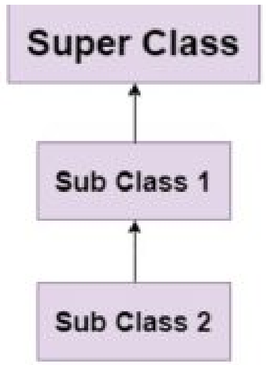

rgupta.mtech@gmail.com

# Inheritance

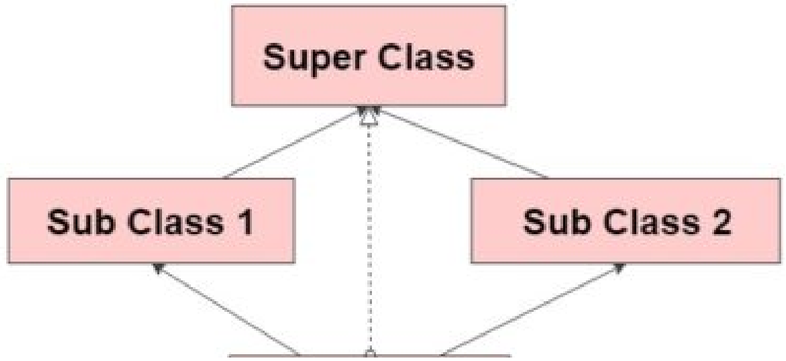
**When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods. An inherited class is defined by using the extends keyword**



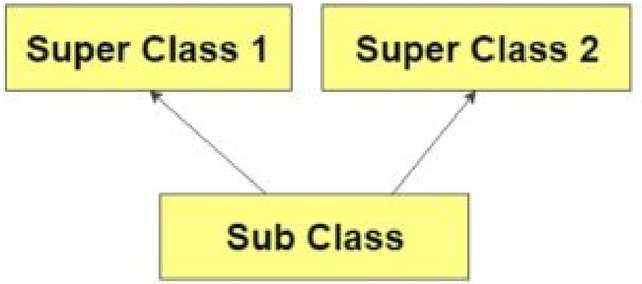




Hybrid Inheritance



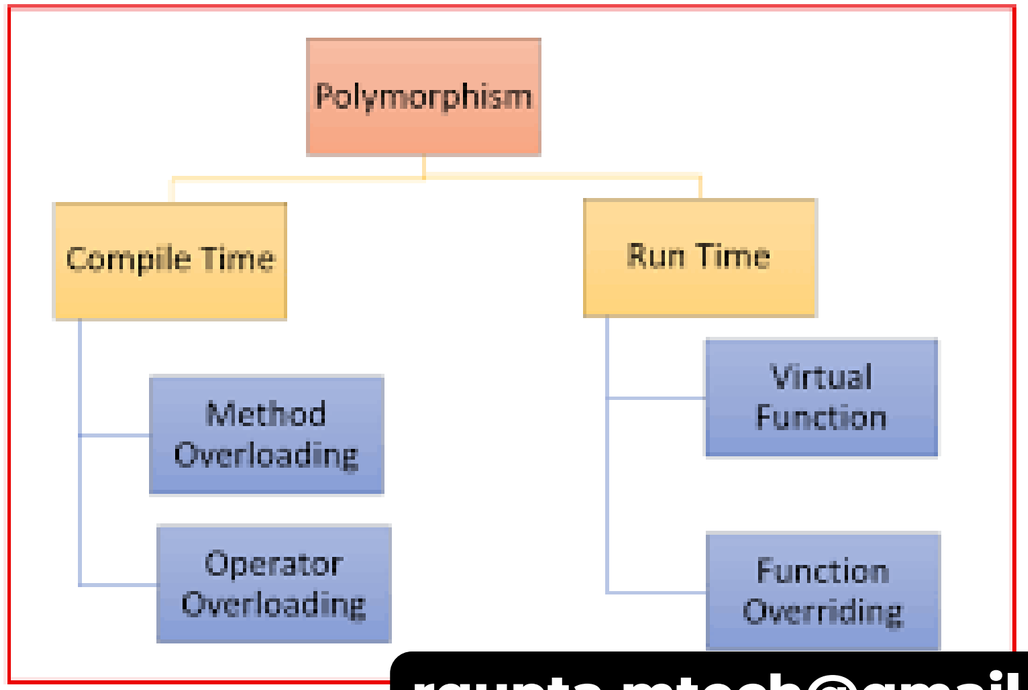

Multiple Inhertance




rgupta.mtech@gmail.com

# Polymorphism

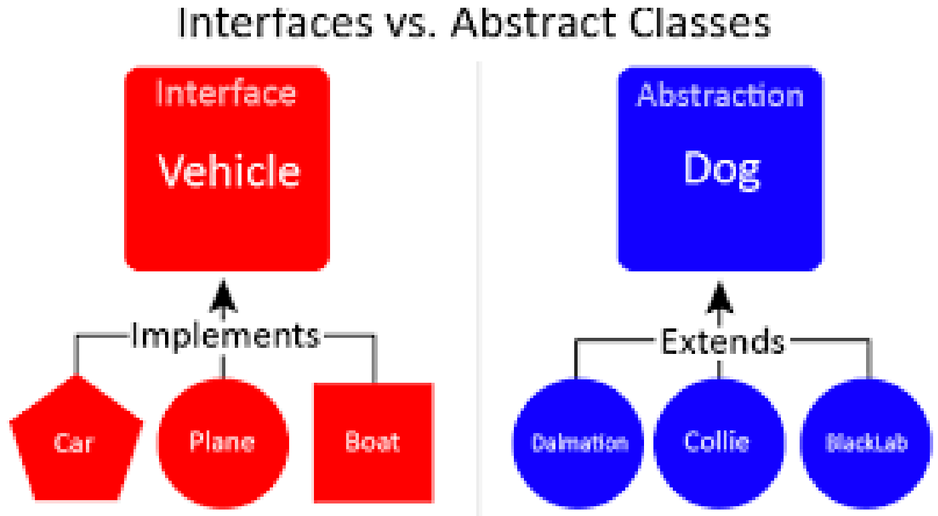
**Polymorphism is one of the core concepts of object-oriented programming (OOP) and describes situations in which something occurs in several different forms. In computer science, it describes the concept that you can access objects of different types through the same interface.**


rgupta.mtech@gmail.com

# Interface vs Abstract class

| | Interface | Abstract Class |
|---|---|---|
| Constructors | ✗ | ✓ |
| Static Fields | ✓ | ✓ |
| Non-static Fields | ✗ | ✓ |
| Final Fields | ✓ | ✓ |
| Non-final Fields | ✗ | ✓ |
| Private Fields & Methods | ✗ | ✓ |
| Protected Fields & Methods | ✗ | ✓ |
| Public Fields & Methods | ✓ | ✓ |
| Abstract methods | ✓ | ✓ |
| Static Methods | ✓ | ✓ |
| Final Methods | ✗ | ✓ |
| Non-final Methods | ✓ | ✓ |
| Default Methods | ✓ | ✗ |



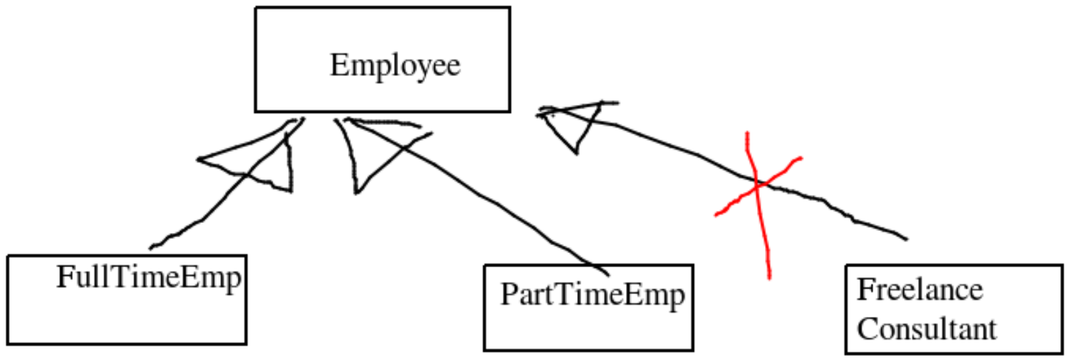

rgupta.mtech@gmail.com

# Interface vs Abstract class

**Interface: Specification of a behavior The interface represents some features of a class**

**What an object can do?**

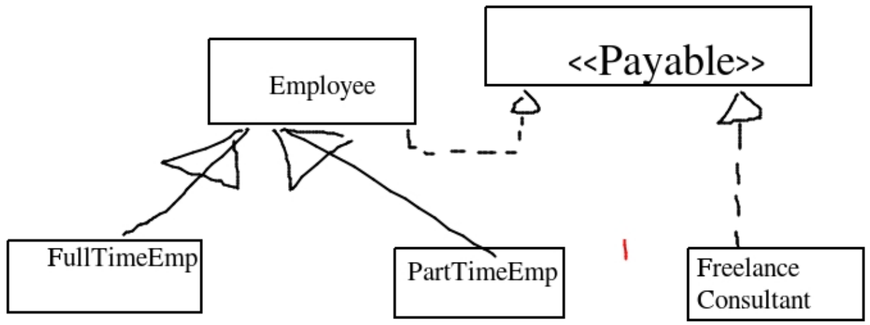
**Standardization of a behavior**




**Abstract class: generalization of a behavior The incomplete class that required further specialization**

**What an object is?**

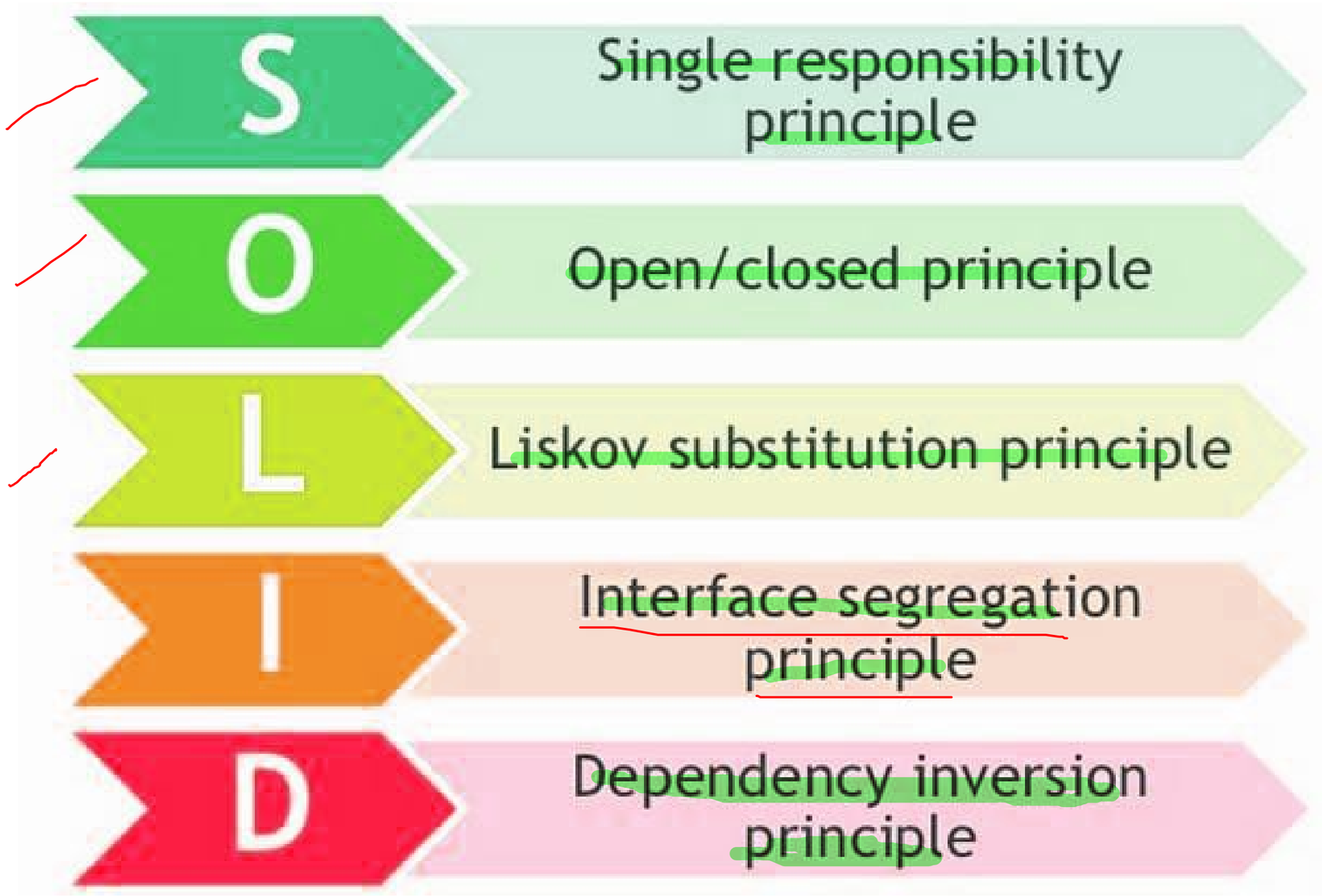
**IS-A SavingAccount IS-A Account**




When we should go for interface and when we should go for abstract class?
Interface break the hierarchy

rgupta.mtech@gmail.com

# SOLID Principles




rgupta.mtech@gmail.com

# Without Good design our application would be...

Rigid

Fragile

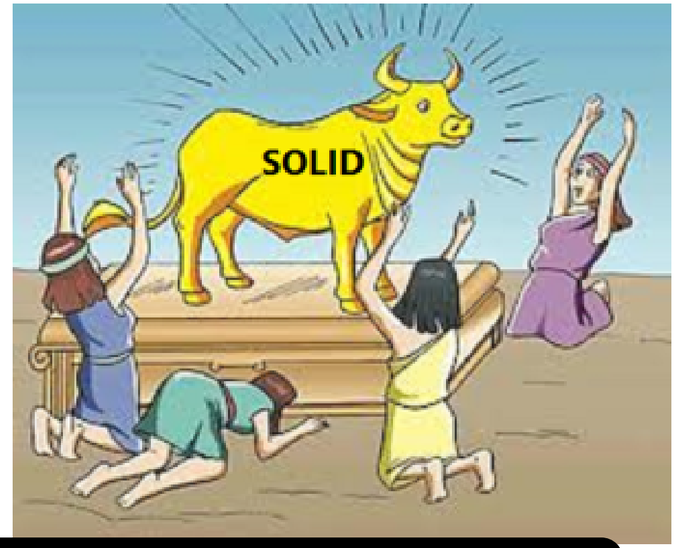
Immobile

Making everyone's life miserable

rgupta.mtech@gmail.com

# SOLID Principles

- In software engineering **SOLID** is an acronym for **5** design principles
- These principles are mainly promoted by **Robert C. Martin** in his book back in **2000**
- **It helps us to write code that is …**
  - **Loosely coupled**
  - **Highly cohesive**
  - **Easily composable**
  - **Reusable**



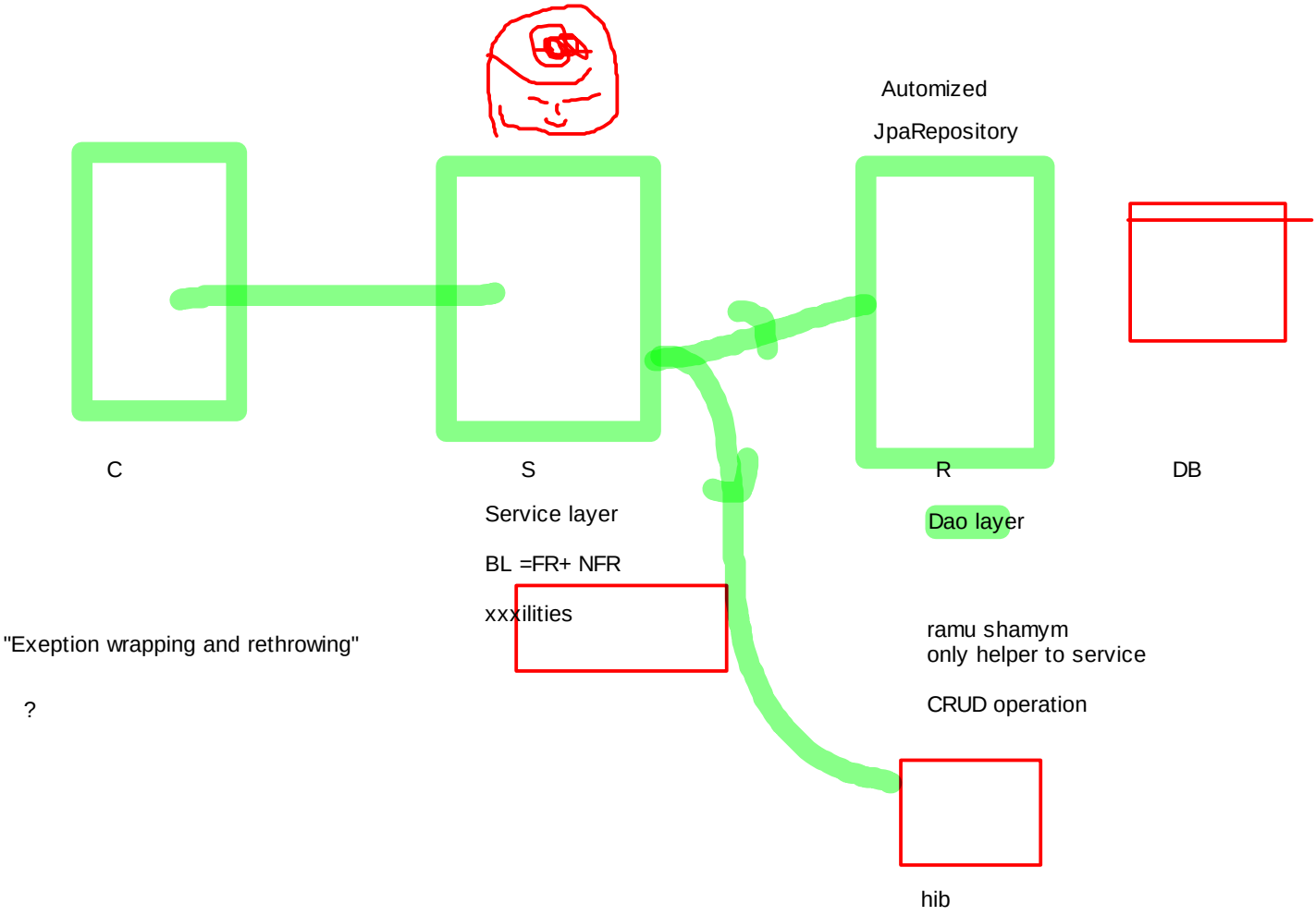

rgupta.mtech@gmail.com

# S - Single Responsibility Principle

- Every single software entity (**class** or **method**) should have only a **single reason** to change
- If a given class (or method) does **multiple operations** then it is advisable to separate into distinct classes (or methods)
- If there are **2** reasons to change a given class then it is a sign of violating the single responsibility principle


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

rgupta.mtech@gmail.com


Automized
JpaRepository
C
S
Service layer
BL =FR+ NFR
xxxilities
R
Dao layer
DB
"Exeption wrapping and rethrowing"
?
ramu shamym
only helper to service
CRUD operation
hib

# O – Open/Closed Principle

- Software entities should be **open for extension** and closed for modification
- We have to design every new module such that if we add a new behavior then we **do not have to change the existing modules**
- CLOSELY RELATED TO SINGLE RESONSIBILITY PRINCIPLE
- a class should not extend an other class explicitly – we should define a **common interface** instead
- we can change the classes at **runtime** due to the common interface

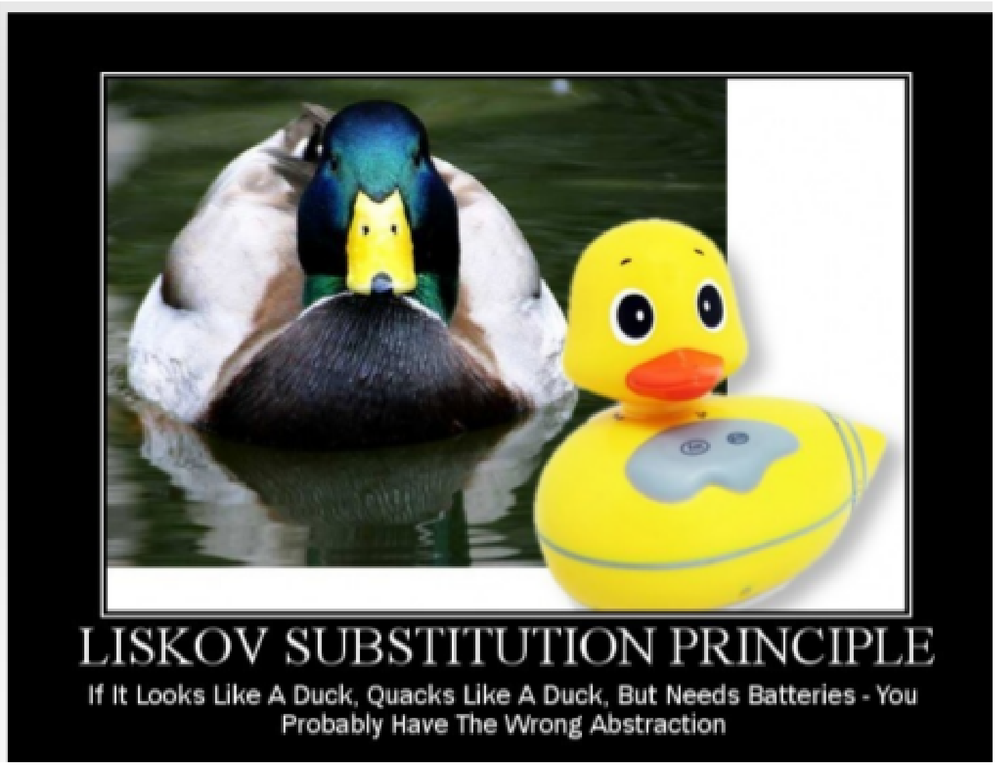

”Modules must be **OPEN** for extension, **CLOSED** for modification”

rgupta.mtech@gmail.com

# L – Liskov Substitution Principle

We extend some classes and creating some **derived classes**

It would be great if the new derived classed would work as well **without replacing the functionality** of the classes otherwise the new classes can produce undesired effects when they are used in **existing program modules**




*"Objects of a superclass shall be replaceable with objects of its subclasses **without breaking** the application"*

**rgupta.mtech@gmail.com**

# I – Interface Segregation Principle

- What is the motivation behind **interface segregation principle**?

WE USE SEVERAL INTERFACES OR ABSTRACT CLASSES
IN ORDER TO ACHIEVE ABSTRACTION

- Sometimes we want to implement that interface but just for the sake of some methods defined in by that interface we can end up with **fat interfaces** – containing more methods than the actual class needs


”Clients should not depend upon the interface they do not use”

How can we pollute the interfaces?
OR
How do we end up creating Fat interfaces?

rgupta.mtech@gmail.com

# D – Dependency Inversion Principle

- What is the motivation behind **dependency inversion principle**?
- USUALLY THE LOW LEVELS MODULES RELY HEAVILY ON HIGH LEVEL MODULES (BOTTOM UP SOFTWARE DEVELOPMENT)
  - When implementing an application usually we start with the **low level** software components then we implement the **high level** modules that rely on these low level modules


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

rgupta.mtech@gmail.com

```
public class Car {
        private V8Engine engine=new V8Engine();
        private CeatTyre tyre=new CeatTyre();

        public Car() {
        }
        public void drive() {
                engine.move();
                tyre.rotate();
        }
}
```

```
public class CeatTyre {
        public void rotate() {
                System.out.println("ceat tyre is rotating");
        }
}
```

```
public class V8Engine{
        public void move() {
                System.out.println("V8 engine is working");
        }
}
```

HLM

LLM

```
public class Car {
        private Engine engine;
        private Tyre tyre;

        public Car(Engine engine, Tyre tyre) {
                this.engine=engine;
                this.tyre=tyre;
        }
        public void drive() {
                engine.move();
                tyre.rotate();
        }
}
```

Engine

V8...

Turbo...

Tyre

Mrf...

Ceat...

```
Tyre tyre=new MrfTyre();
        Engine engine=new TurboEngine();

        Car car =new Car(engine, tyre);
```

You are managing the dep

can u have a freamewrok
that can manage dep for you

YES

Spring!

# Design Patterns

***design patterns*** *are more about how to design your code*

*+ concrete implementations of the design principles*


***design principles*** *(SOLID principles) allow scalable and maintainable software architectures*

rgupta.mtech@gmail.com

# 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

rgupta.mtech@gmail.com

# Design Patterns

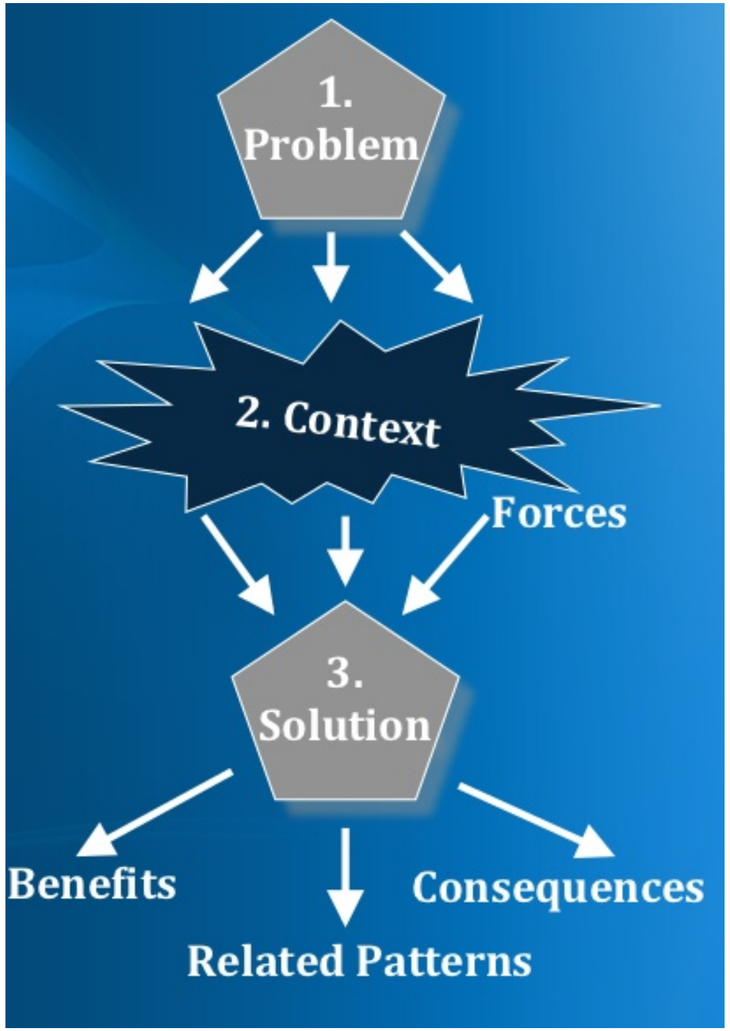
Proven way of doing Things, Gang of 4 design patterns

- Total 23 patterns

- Classification patterns
- Creational
- Structural
- Behavioral

**Design patterns are design level solutions for recurring problems that we software engineers come across often.**

**It's not code - I repeat, ✕CODE. It is like a description on how to tackle these problems and design a solution**

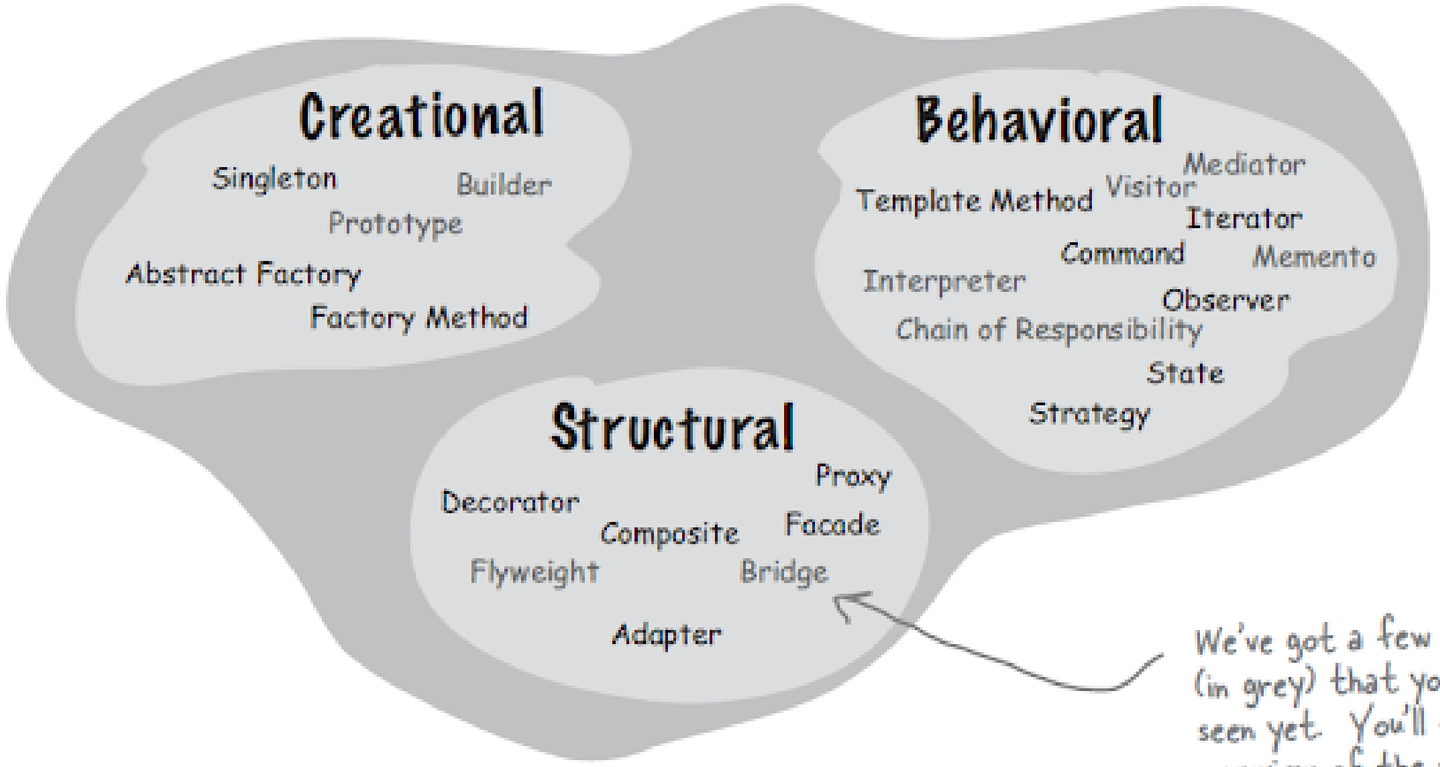



rgupta.mtech@gmail.com

# Design Patterns

**Creational patterns** involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.

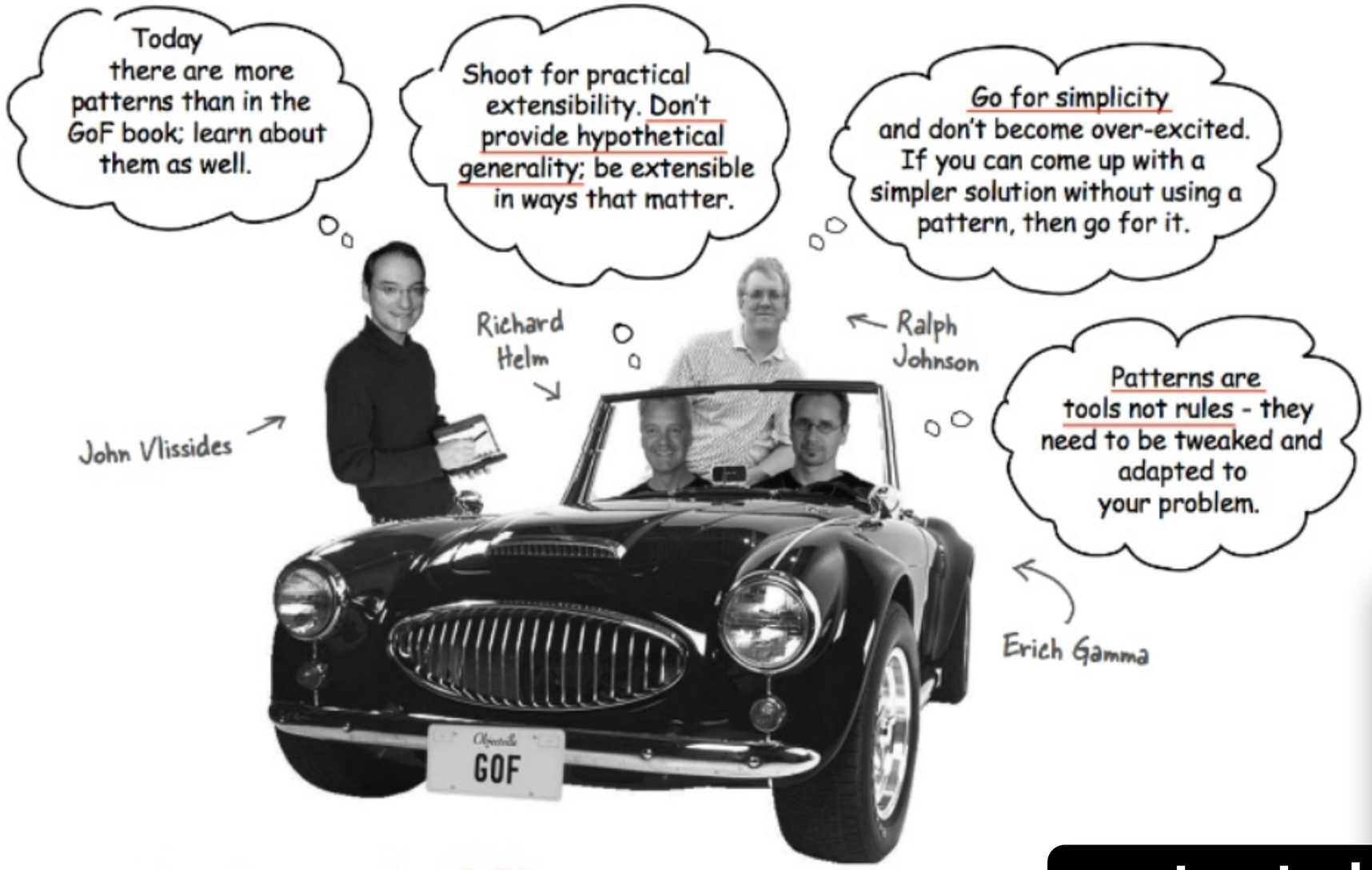



We've got a few patterns (in grey) that you haven't seen yet. You'll find an overview of the these patterns in the appendix.

**Structural patterns** let you compose classes or objects into larger structures.

rgupta.mtech@gmail.com

# Design Patterns





Keep it simple (KISS)

rgupta.mtech@gmail.com

Car car =new Car();

# Design Patterns- Classification

## Structural Patterns

- 1. Decorator
- 2. Proxy
- 3. Bridge
- 4. Composite
- 5. Flyweight
- 6. Adapter
- 7. Facade

## Creational Patterns

- 1. Prototype
- 2. Factory Method
- 3. Singleton
- 4. Abstract Factory
- 5. Builder

## Behavioral Patterns

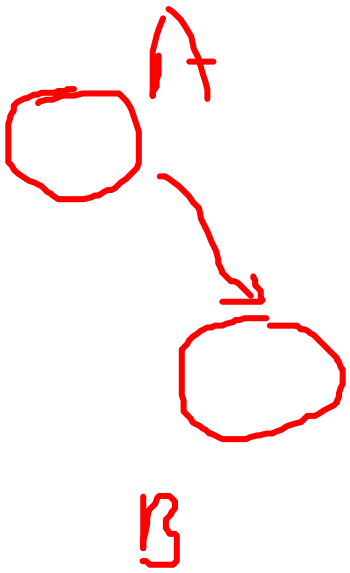
- 1. Strategy
- 2. State
- 3. TemplateMethod
- 4. Chain of Responsibility
- 5. Command
- 6. Iterator
- 7. Mediator
- 8. Observer
- 9. Visitor
- 10. Interpreter
- 11. Memento

rgupta.mtech@gmail.com

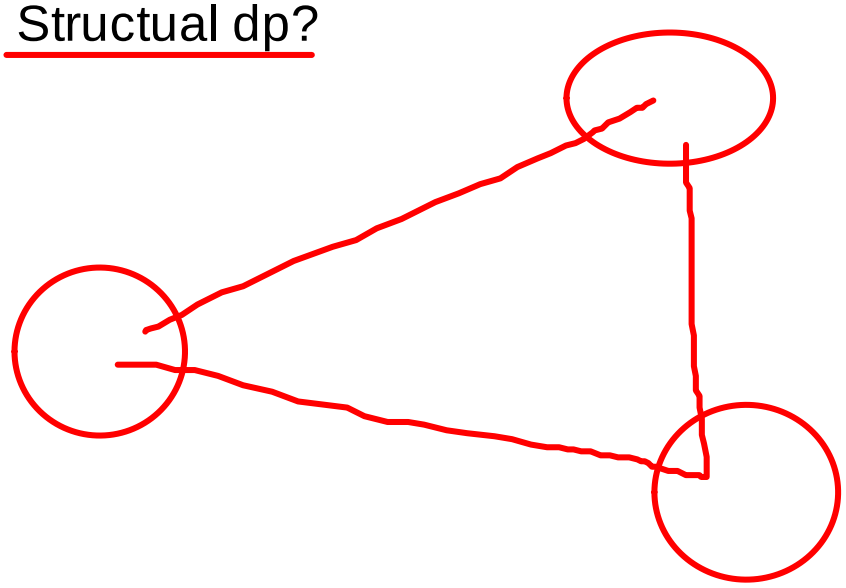
What is creational ?

best way to create the object ....

Car car =new Car();



Structual dp?



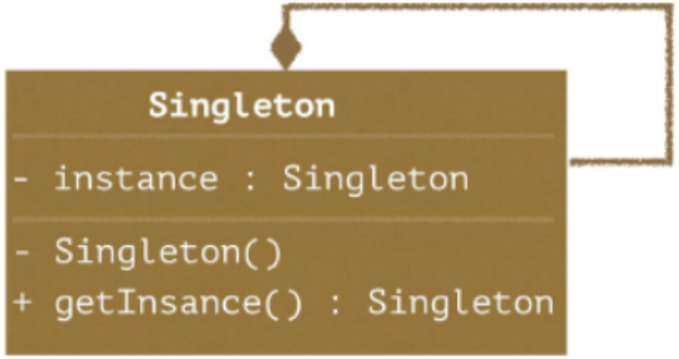
loose coupling and high cohesion


rgupta.mtech@gmail.com

# Singleton Pattern

- **Singleton** pattern is a creational design pattern
- It lets you ensure that a **class has only one instance** while providing a global access point to this instance
- It ensures that a given class has just a single instance
- The singleton pattern provides a **global access point** to that given instance

"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.

rgupta.mtech@gmail.com

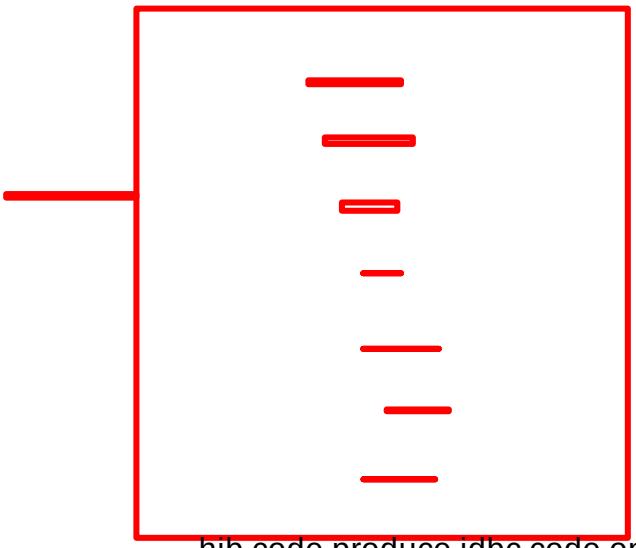
# Singleton Design Consideration

- 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

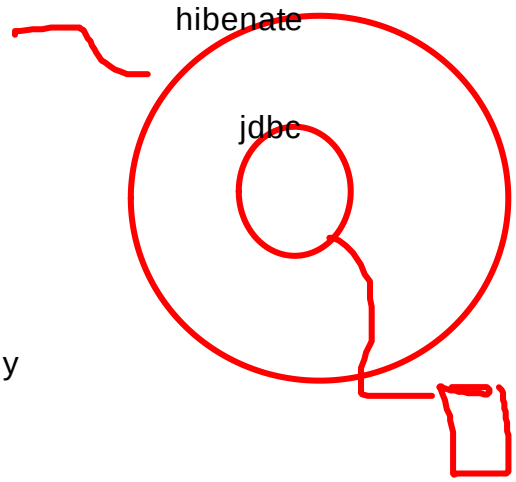

rgupta.mtech@gmail.com

```
@Entity
@Table
class Emp{

    @Id @Gen...
    int id;
    String name;

}

Emp e=....
session.save(e);
```

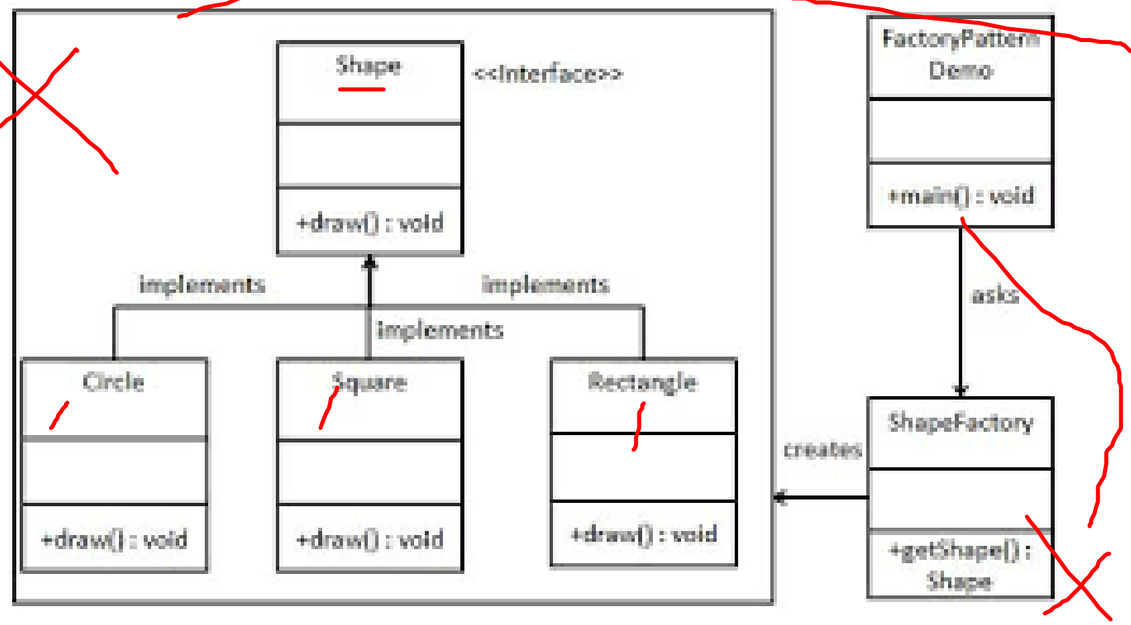
how come?



hib code produce jdbc code on fly



# 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.**

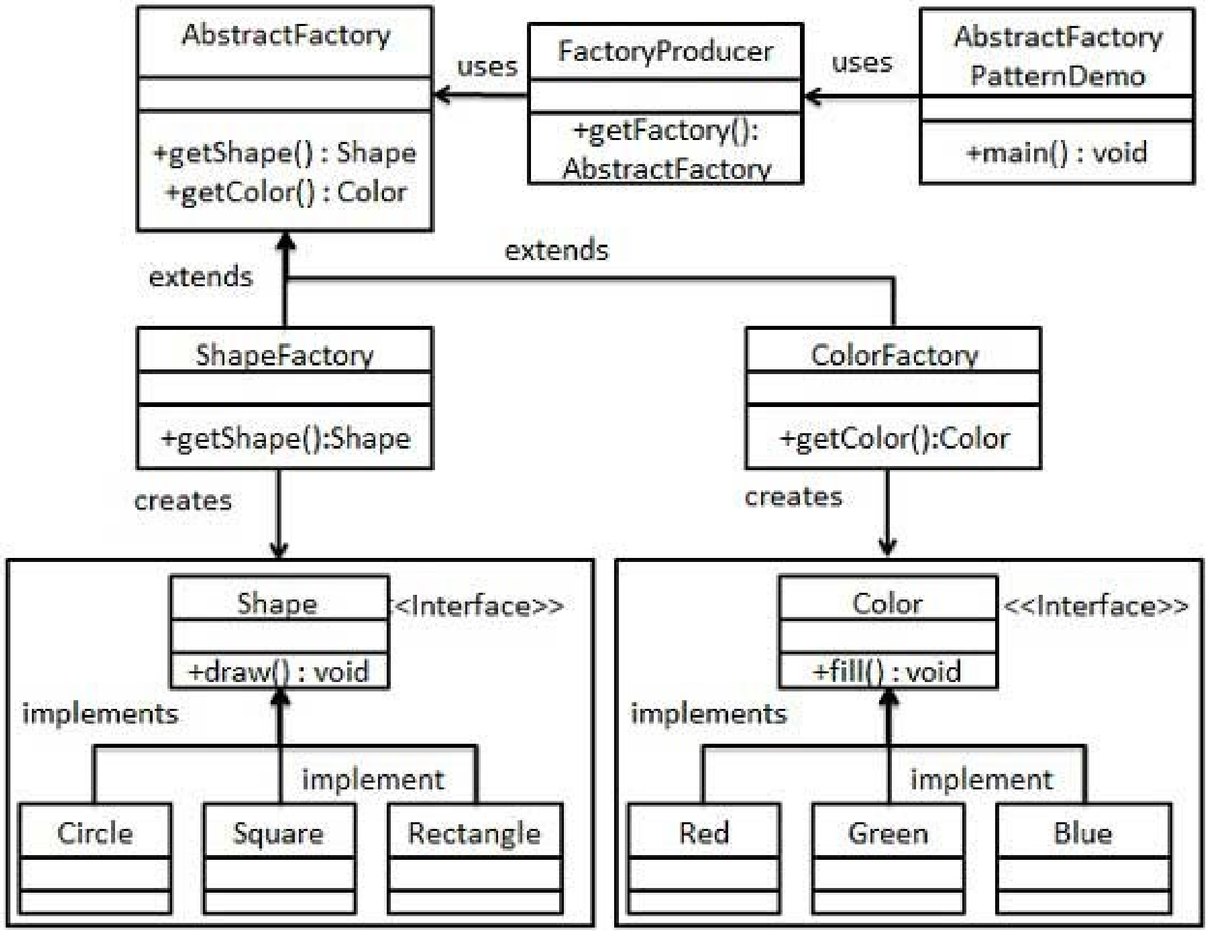
- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`

**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.**

rgupta.mtech@gmail.com

# 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()

rgupta.mtech@gmail.com

# Builder Pattern

what is the **motivation** behind 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)

telescoping constructors

```
public class Person {

    private int age;
    private Gender gender;
    private String dateOfBirth;
    private String firstName;
    private String lastName;
    private String nameOfMother;
    private Address address;
    private PhoneNumber phoneNumber;

    public Person(int age, Gender gender, String dateO
            String lastName, String nameOfMother,
            Address address, PhoneNumber phoneNumber)
        super();
        this.age = age;
        this.gender = gender;
        this.dateOfBirth = dateOfBirth;
        this.firstName = firstName;
        this.lastName = lastName;
        this.nameOfMother = nameOfMother;
```

*LARGE NUMBER OF VARIABLES*

*there may be a **large amount** of parameters in a constructor*

*because there may be several **instance variables** in a given class*

*EASY TO CONFUSE THE PARAMETERS*

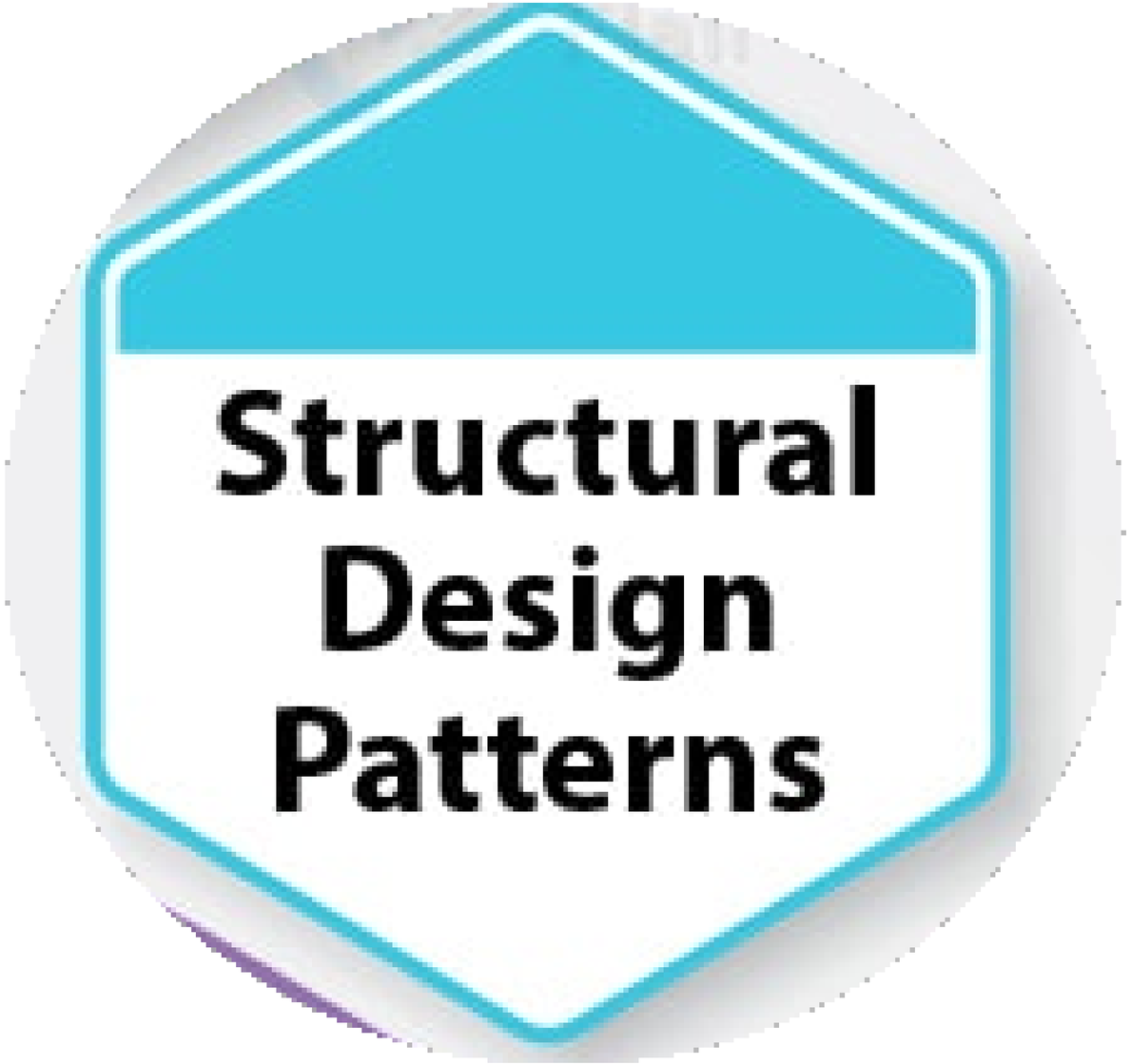
rgupta.mtech@gmail.com

# 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`)

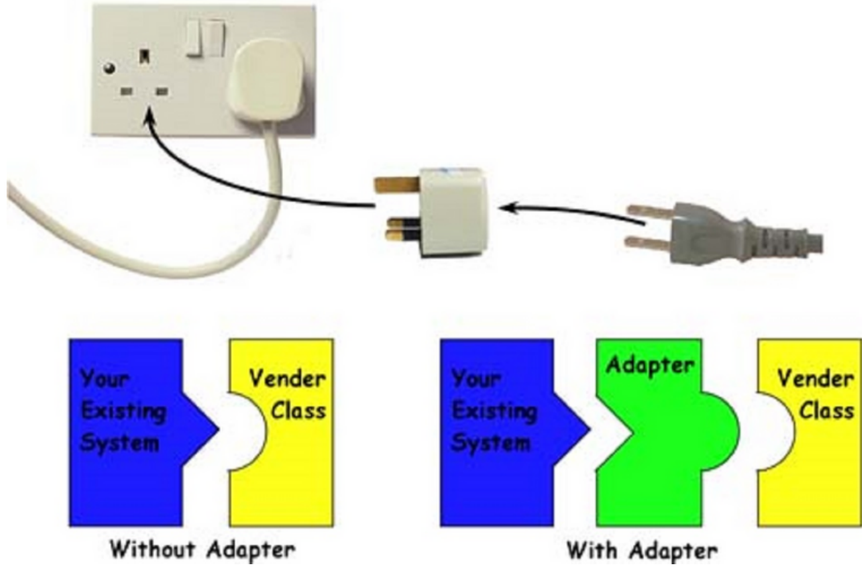
rgupta.mtech@gmail.com




rgupta.mtech@gmail.com

# Adapter Pattern

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

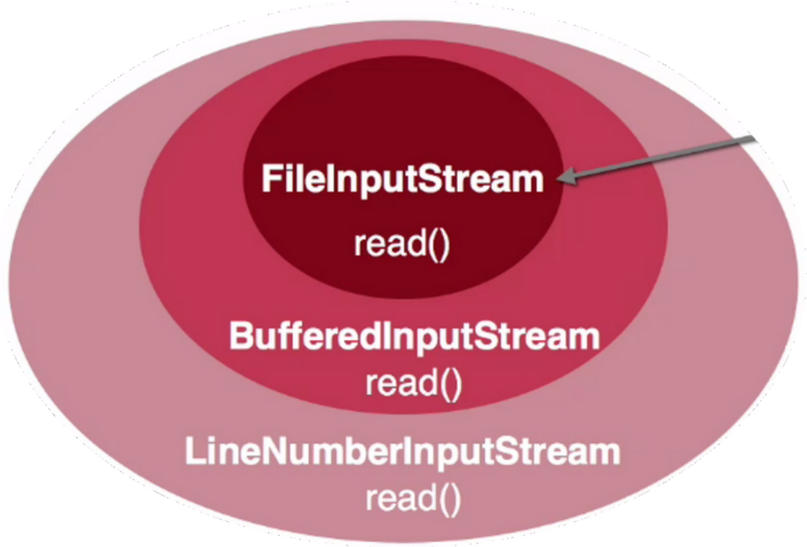



- `java.util.Arrays#asList()`
- `java.util.Collections#list()`
- `java.util.Collections#enumeration()`
- `java.io.InputStreamReader(InputStream)` (returns a `Reader`)
- `java.io.OutputStreamWriter(OutputStream)` (returns a `Writer`)

rgupta.mtech@gmail.com

# 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.

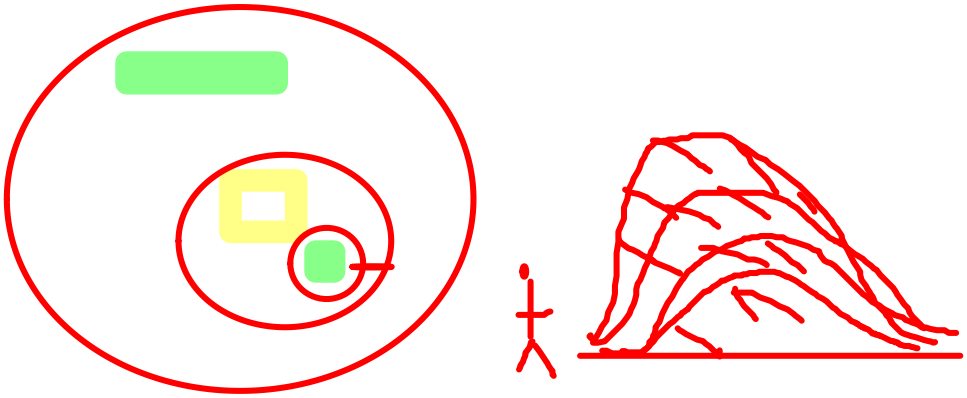
`javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

`javax.swing.JScrollPane`

rgupta.mtech@gmail.com

```
BufferedReader br=
        new BufferedReader(new FileReader(new File("foo.txt")));
```





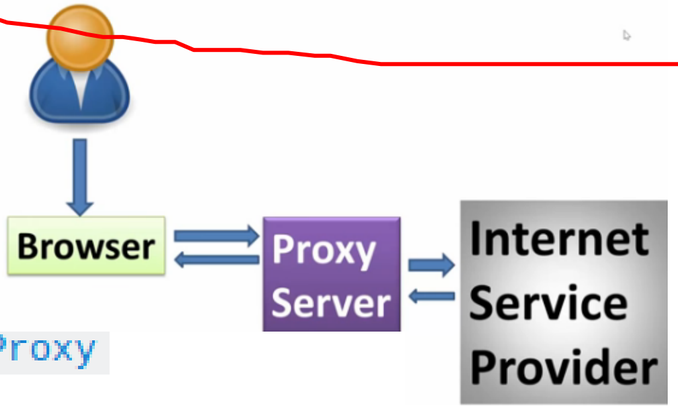
```
Greet greet=new DeepawaliSpecialGreeting(new SpecialGreeting(new Greet()));
```

session.get(...)
session.load(...)

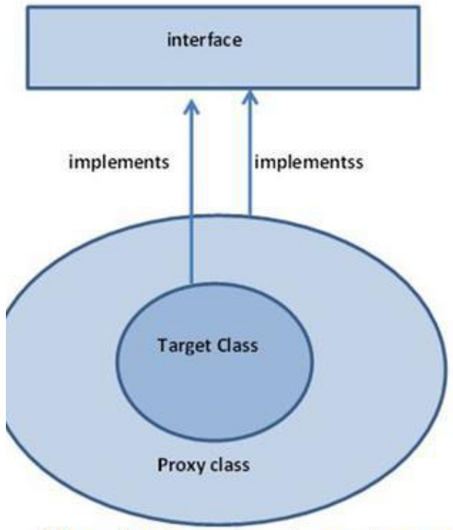
# Proxy design Pattern

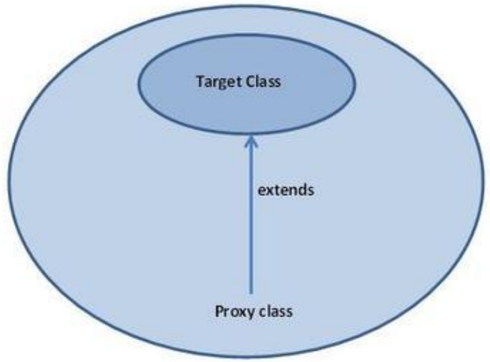
AOP

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.




CGLIB proxy

`java.lang.reflect.Proxy`

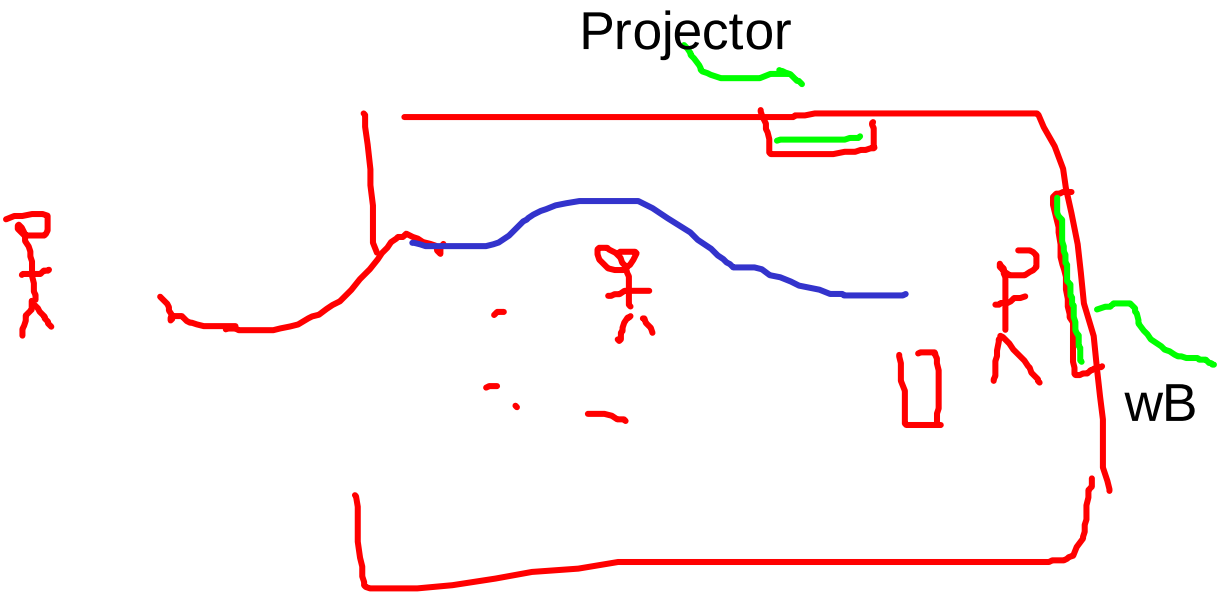
`java.rmi.*`

`javax.ejb.EJB` (explanation here)

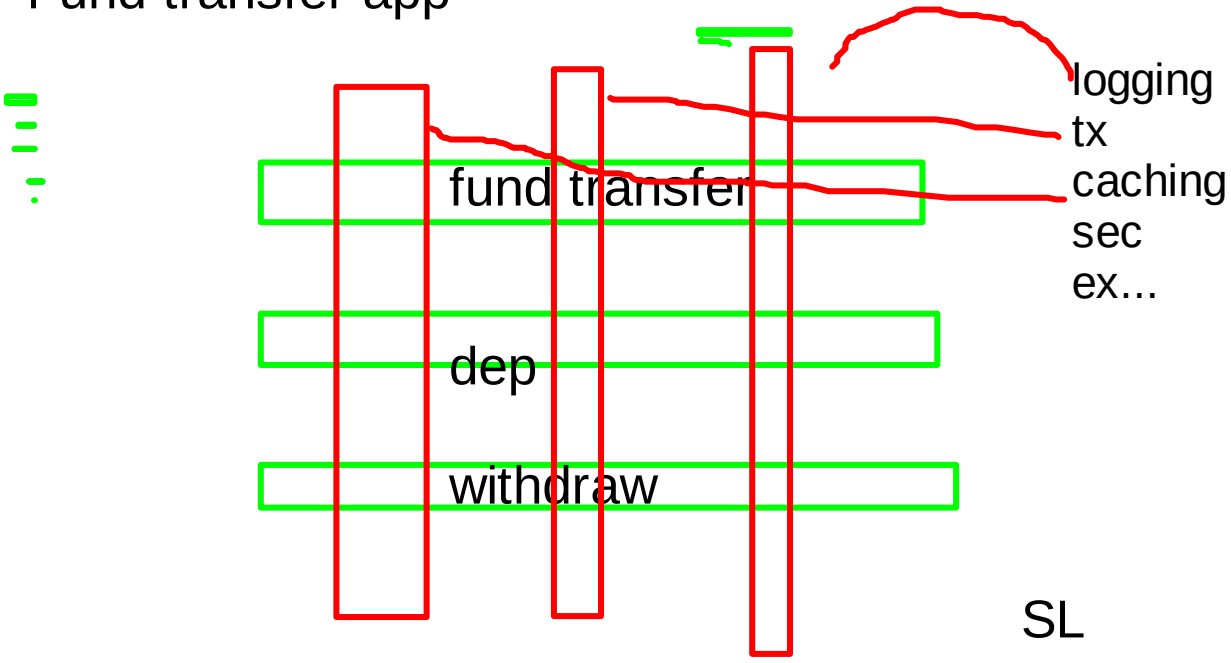
`javax.inject.Inject` (explanation here)

`javax.persistence.PersistenceContext`

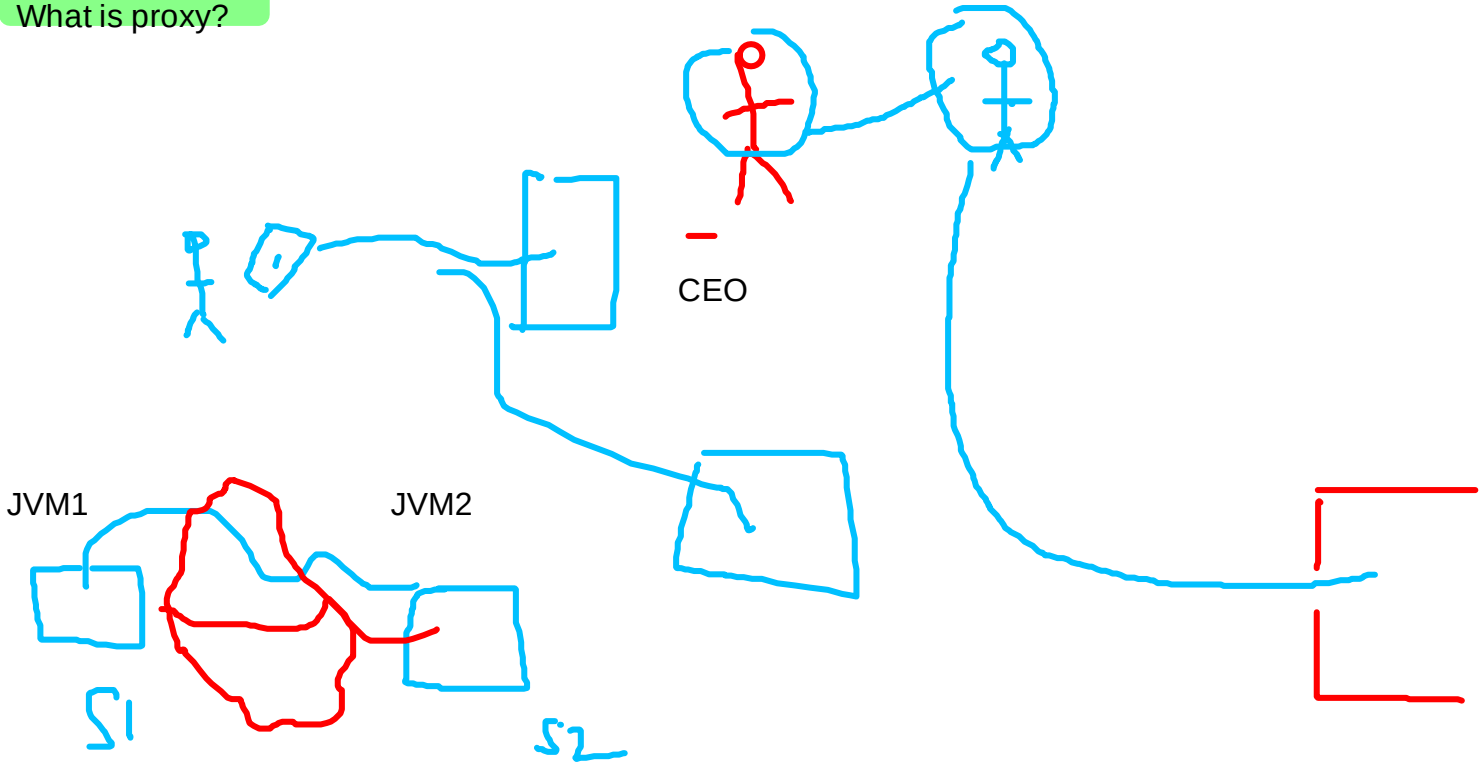
rgupta.mtech@gmail.com



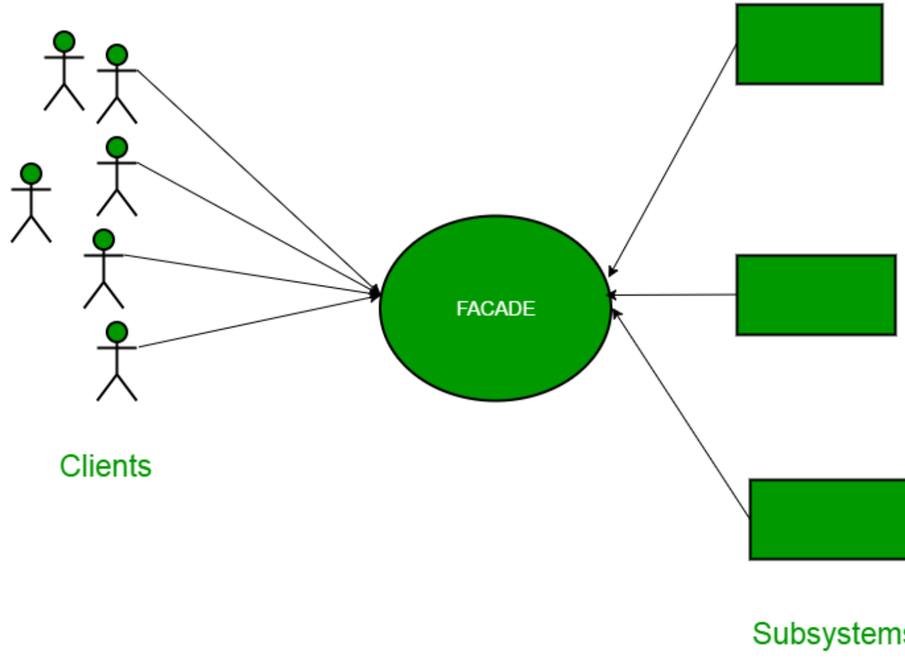

Fund transfer app



What is proxy?



# 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.

rgupta.mtech@gmail.com

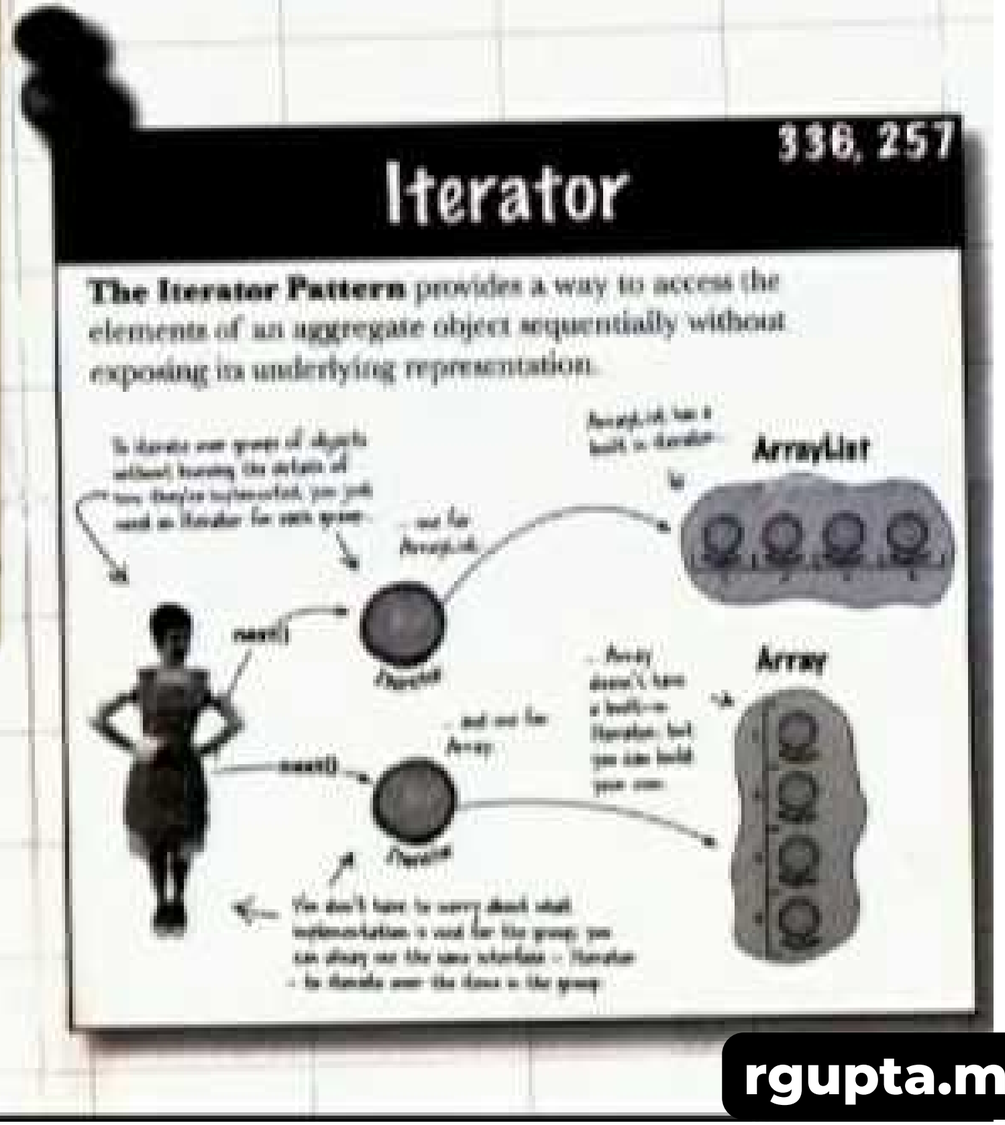

rgupta.mtech@gmail.com

# Iterator Pattern

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

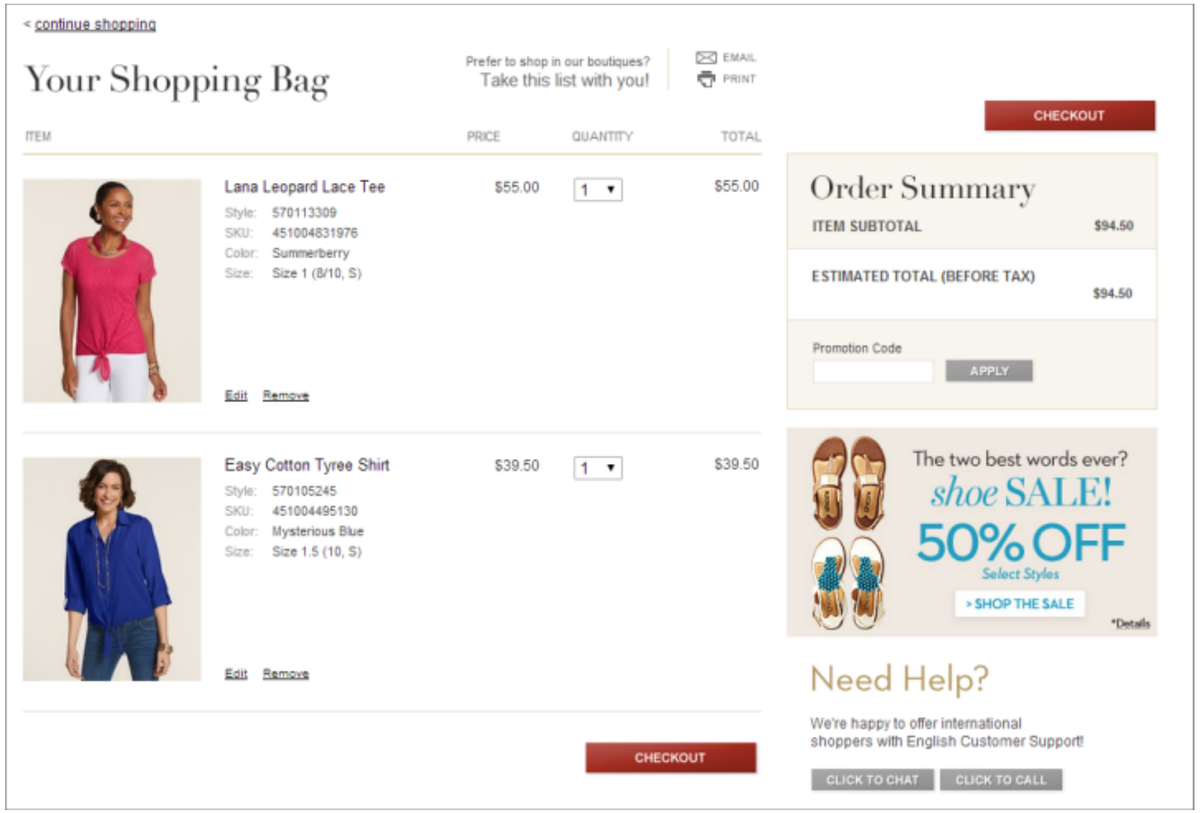
Head First Design Patterns Poster
O'Reilly,
ISBN 0-596-10214-3

Turing.com.br




rgupta.mtech@gmail.com

# Strategy pattern /policyPattern




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

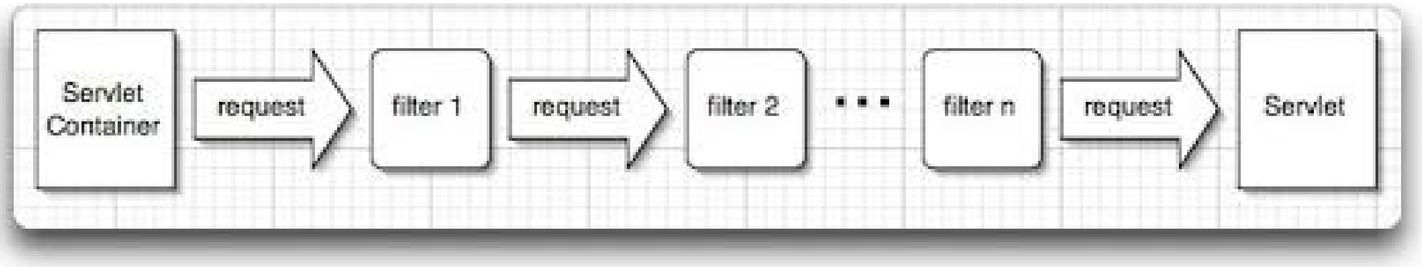
- `java.util.Comparator#compare()`, executed by among others `Collections#sort()`.
- `javax.servlet.http.HttpServlet`, the `service()` and all `doXXX()` methods take `HttpServletRequest` and `HttpServletResponse` and the implementor has to process them (and not to get hold of them as instance variables!).

rgupta.mtech@gmail.com

# Chain of Responsibility Pattern

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.


## Servlet Filter, Spring Security FilterChain

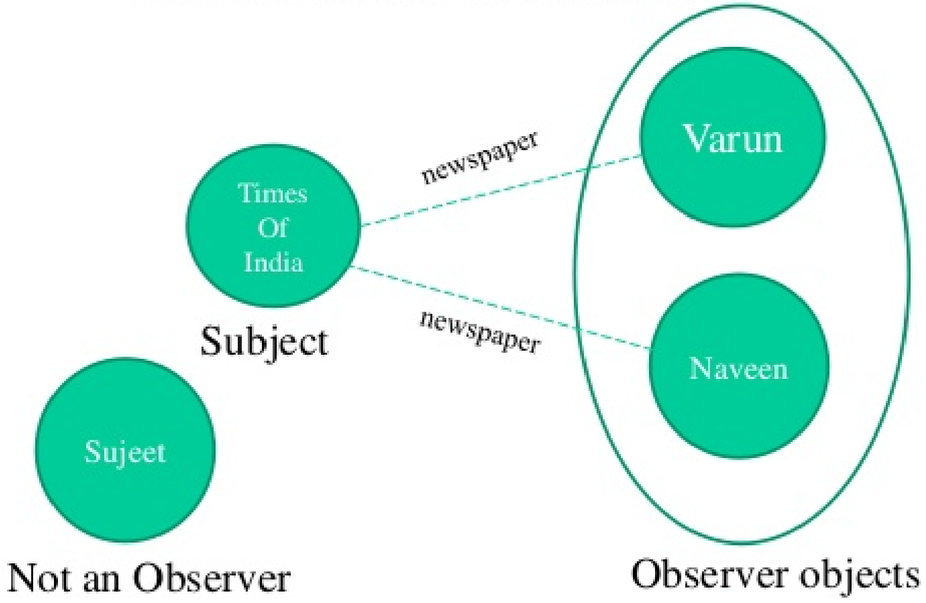



rgupta.mtech@gmail.com

# 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`

Framework --> training
core pattern that is used with that framworks

1/2

rgupta.mtech@gmail.com

Spring jdbc

# 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.

Foundation

Pillars

Walls

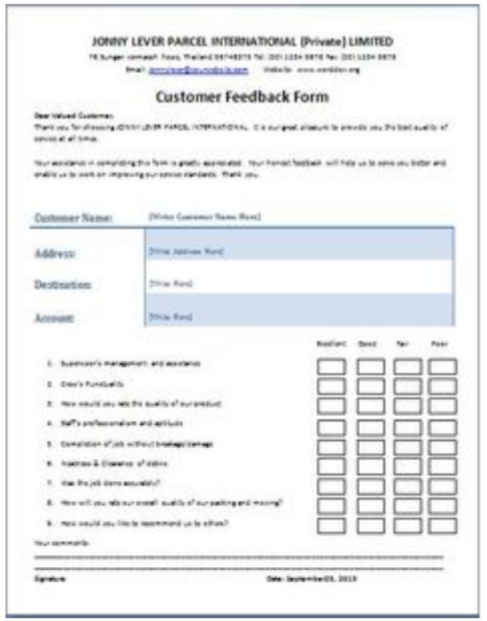
Windows

Template Method [Build a House]

1) Building foundation
2) Building pillars
3) Building walls
4) Building windows

Concrete House

Wooden House



JONNY LEVER PARCEL INTERNATIONAL (Private) LIMITED

Customer Feedback Form

spring JDBC

rgupta.mtech@gmail.com

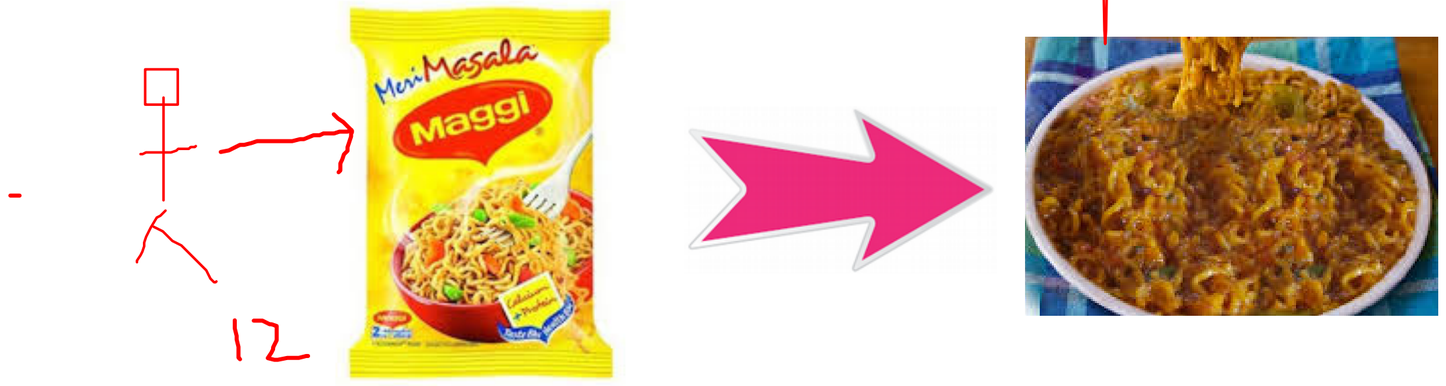
# Pattern vs Framework

- 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.**

Half backed code readymade download

Half backed food

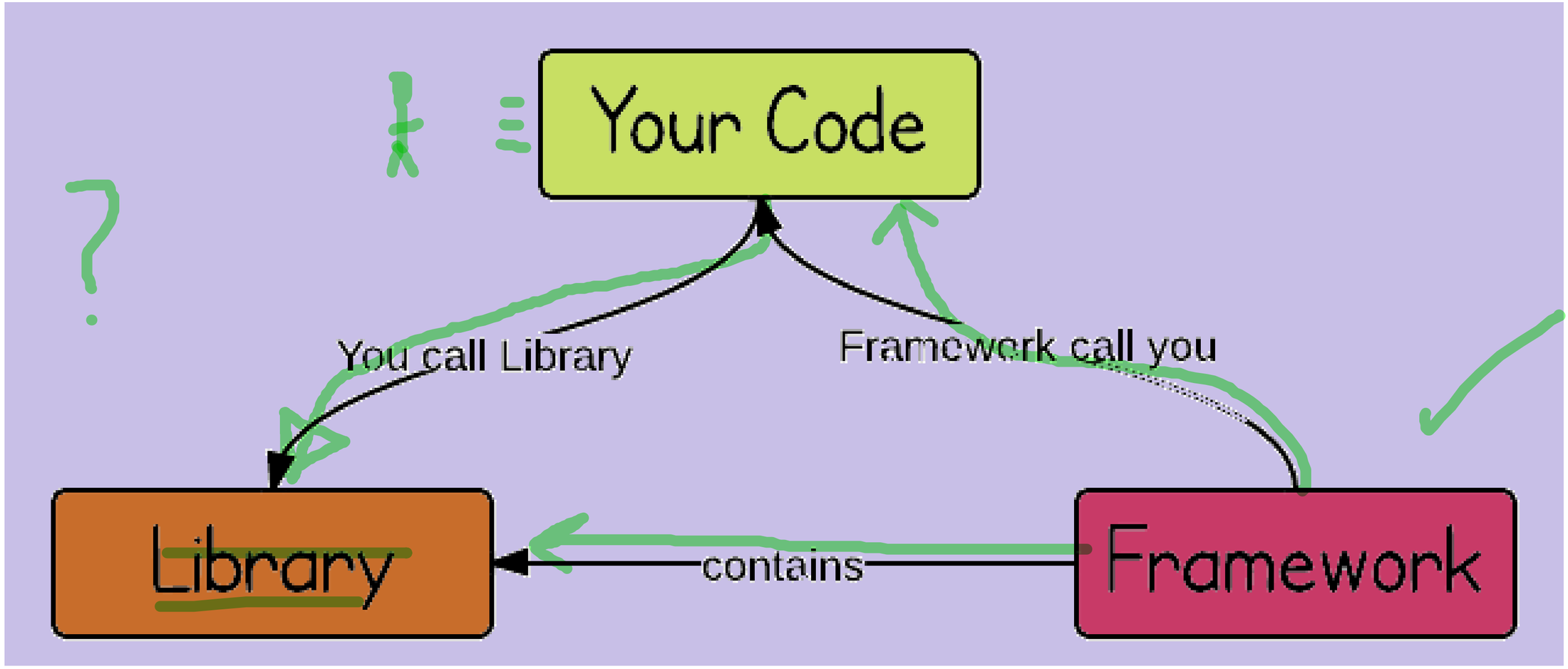



rgupta.mtech@gmail.com

JQuery

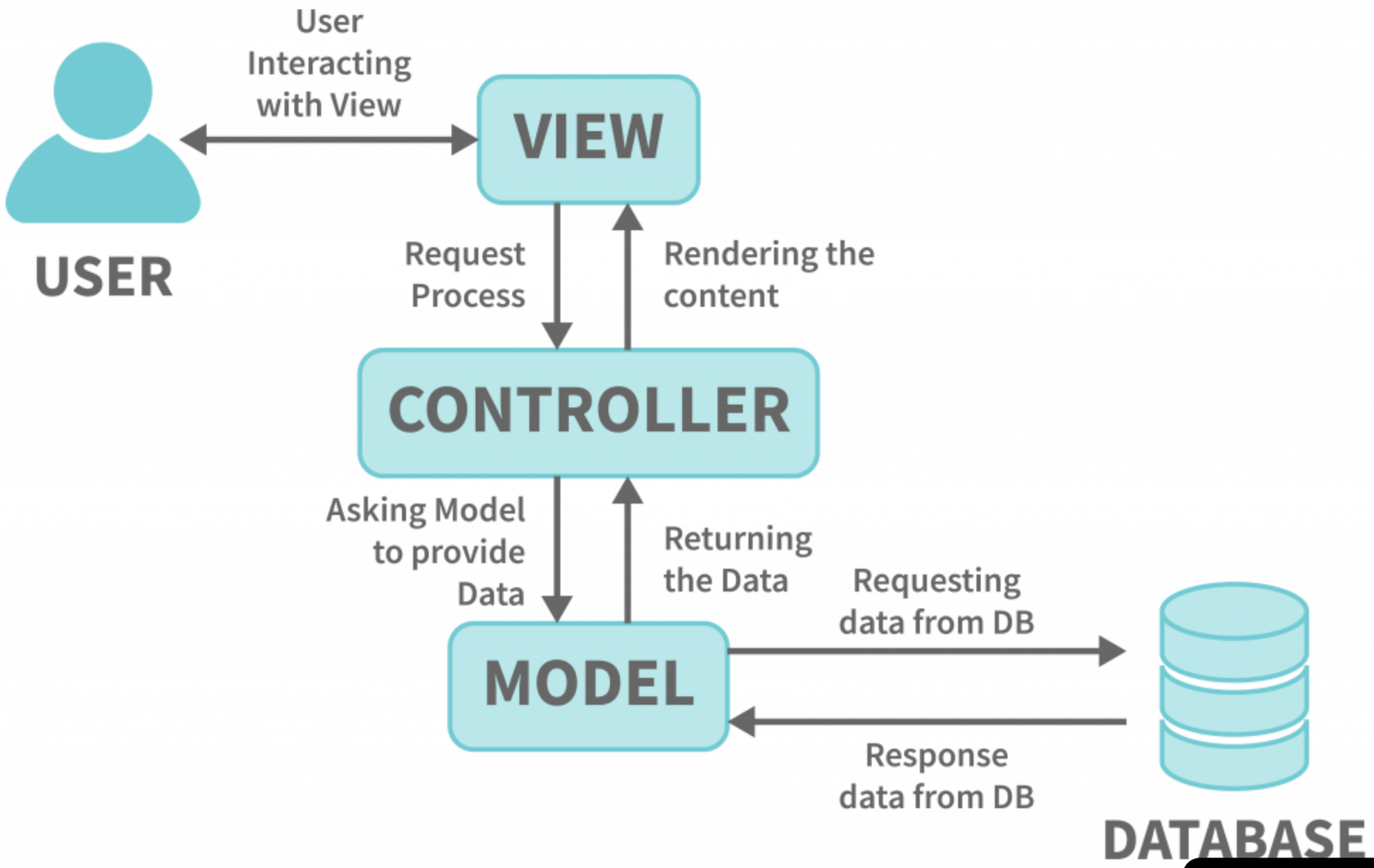
Angular

# Library vs Framework



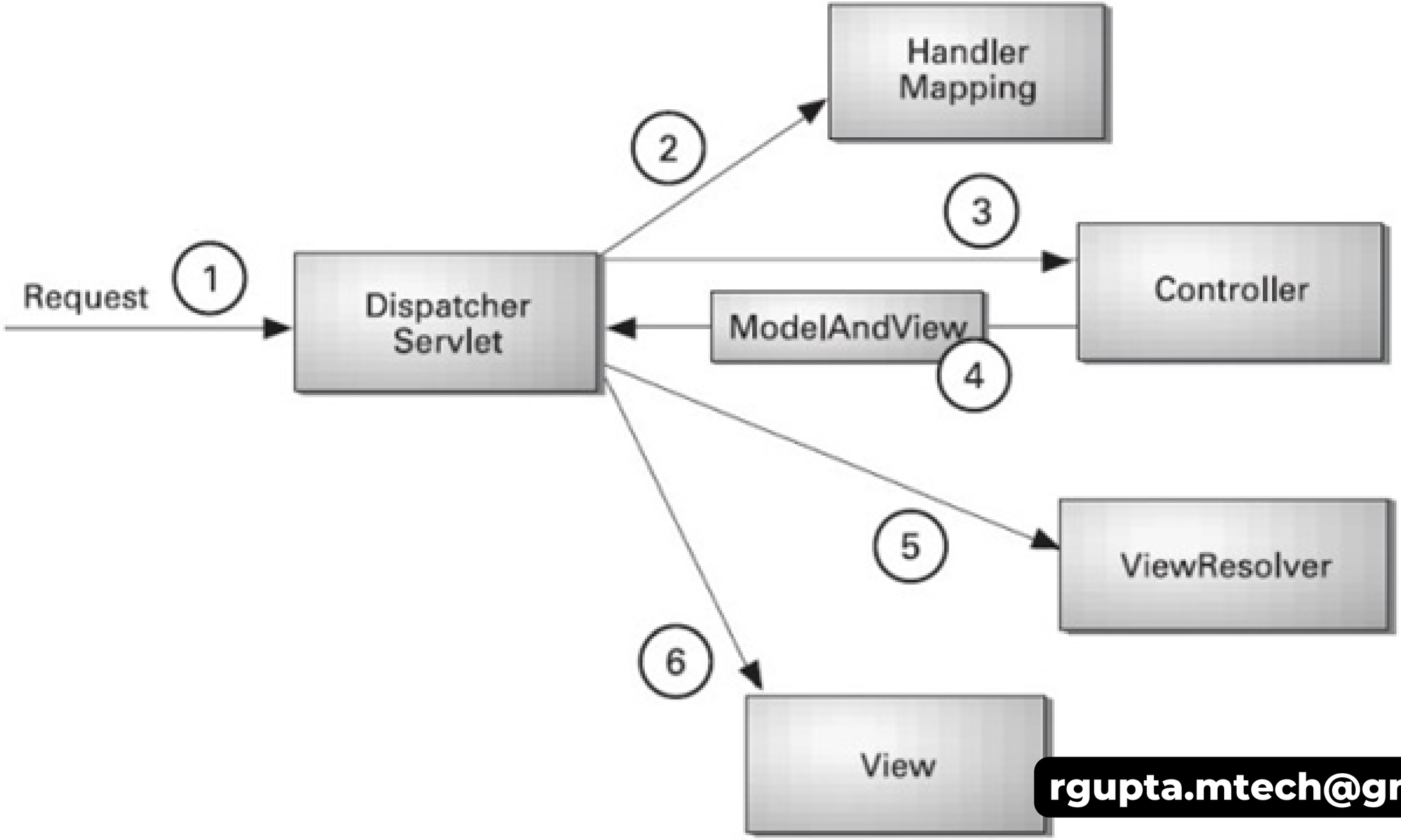

rgupta.mtech@gmail.com

# MVC Design Pattern



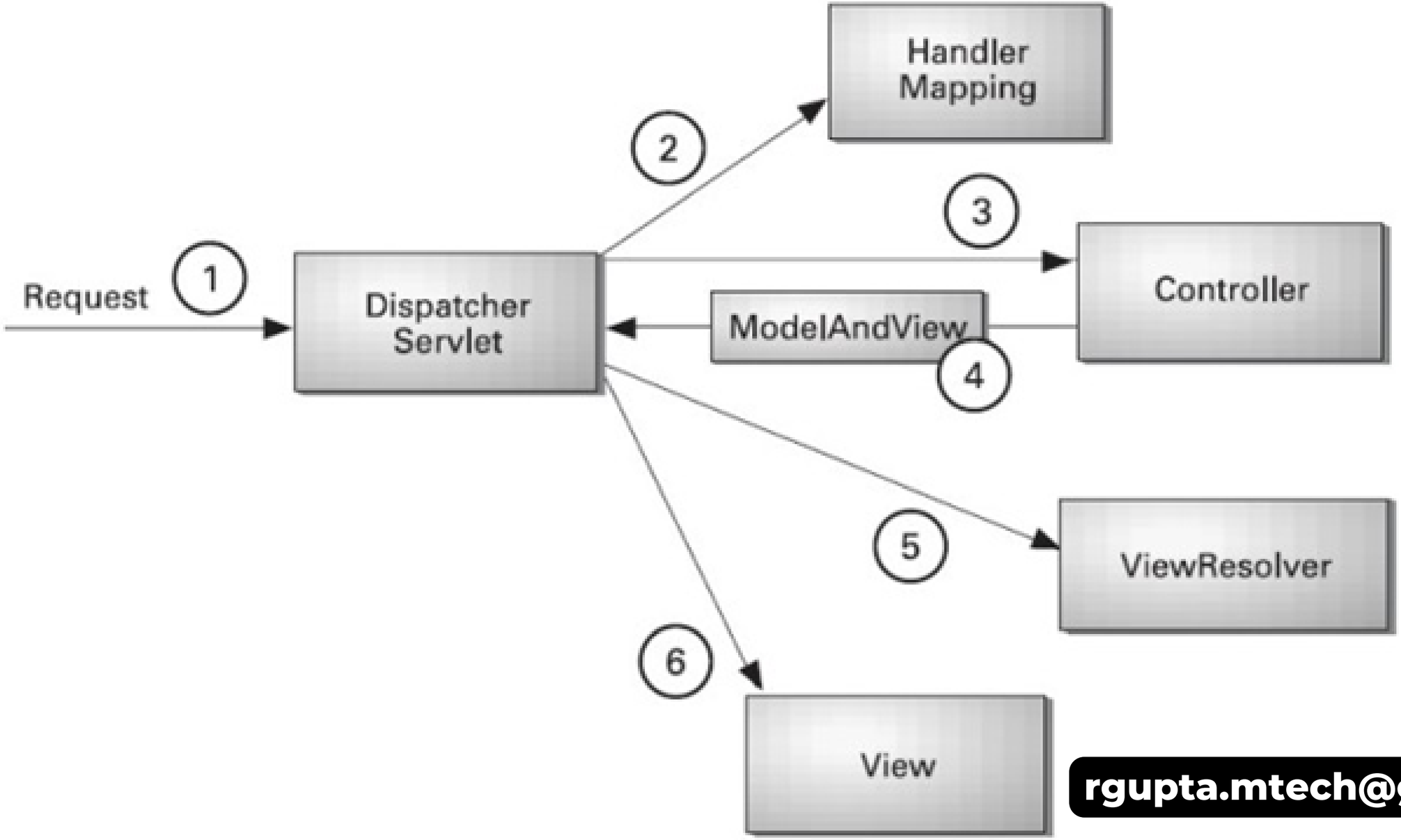

rgupta.mtech@gmail.com

# MVC Design Pattern




rgupta.mtech@gmail.com

# MVC Design Pattern




rgupta.mtech@gmail.com