

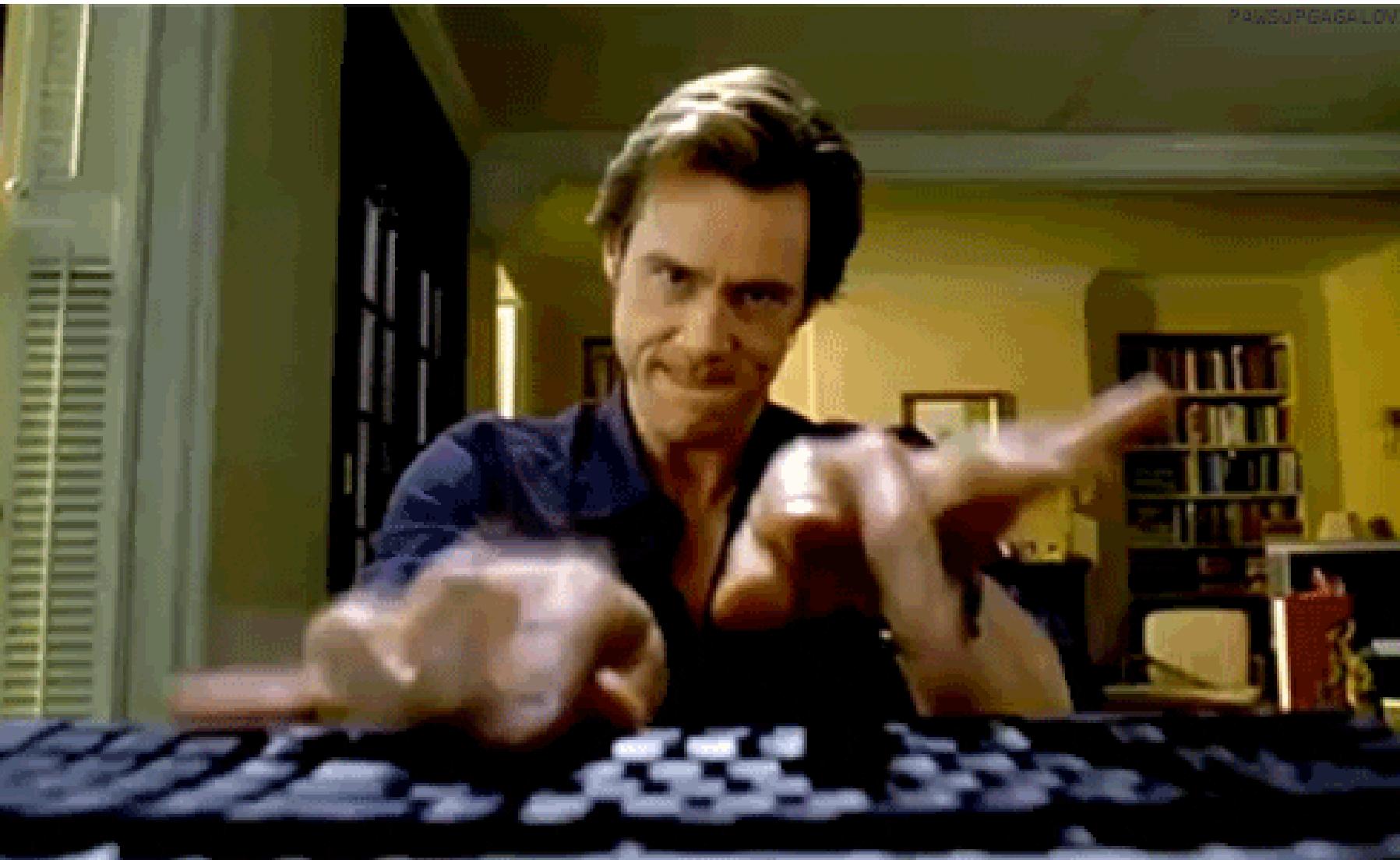
Object Orientation, SOLID and GOF deep dive

Rajeev Gupta
rgupta.mtech@gmail.com
<https://www.linkedin.com/in/rajeevguptajavatrainer>



rgupta.mtech@gmail.com

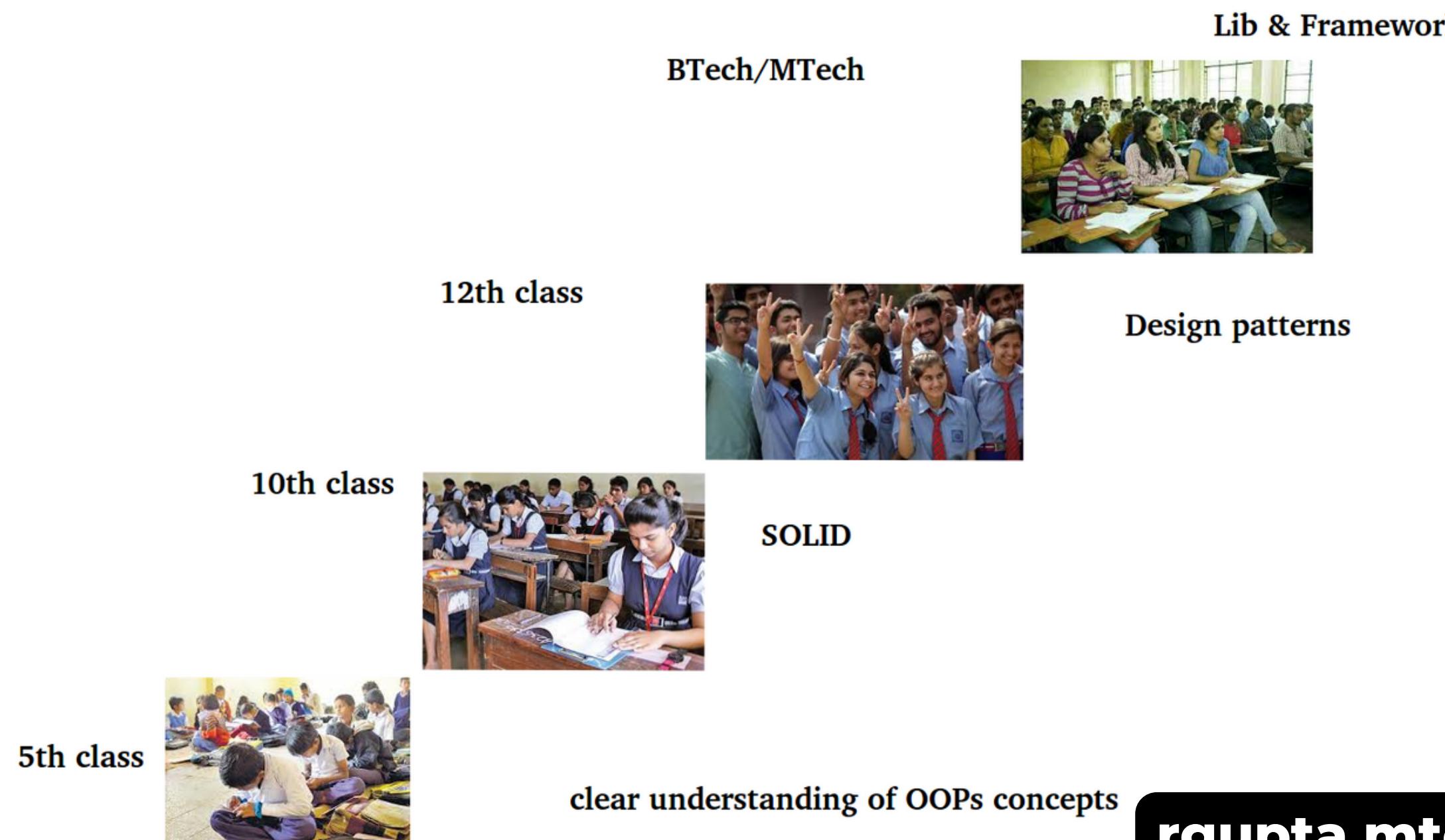
Programmer are Not typist but thinker



rgupta.mtech@gmail.com

Learning OOPs, SOLID, Design Patterns

step by step

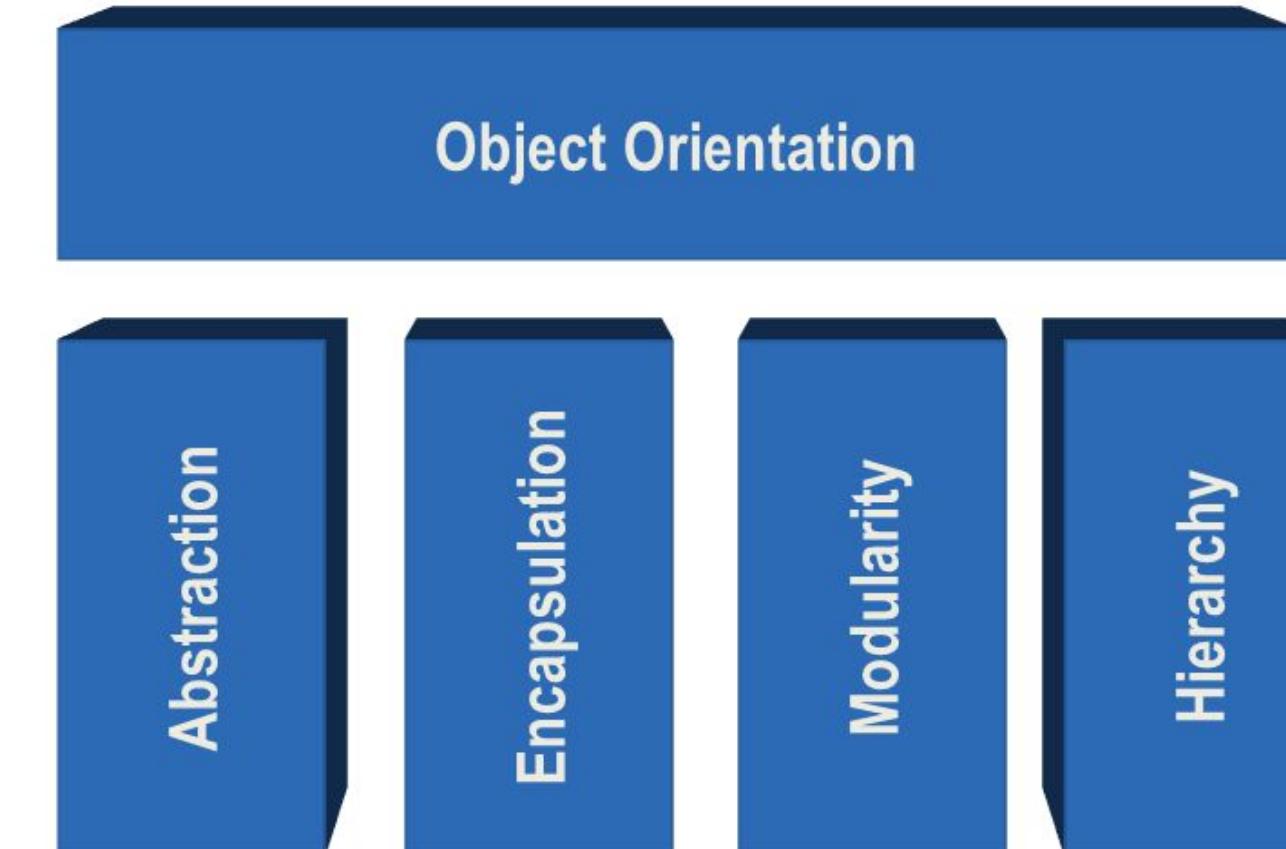


rgupta.mtech@gmail.com

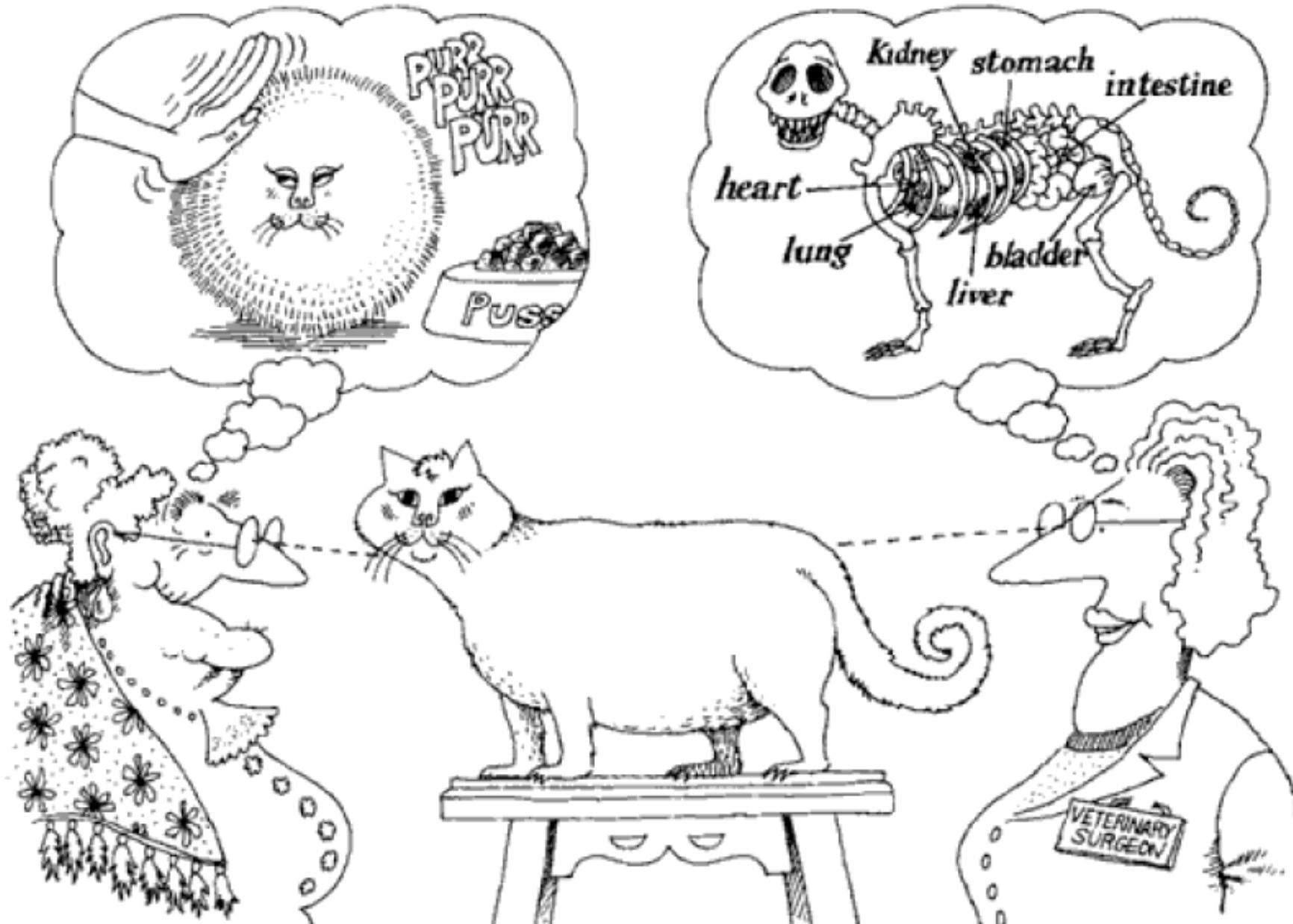
Pillars of object oriented programmin

Abstraction
Encapsulation
Modularity
Hierarchy

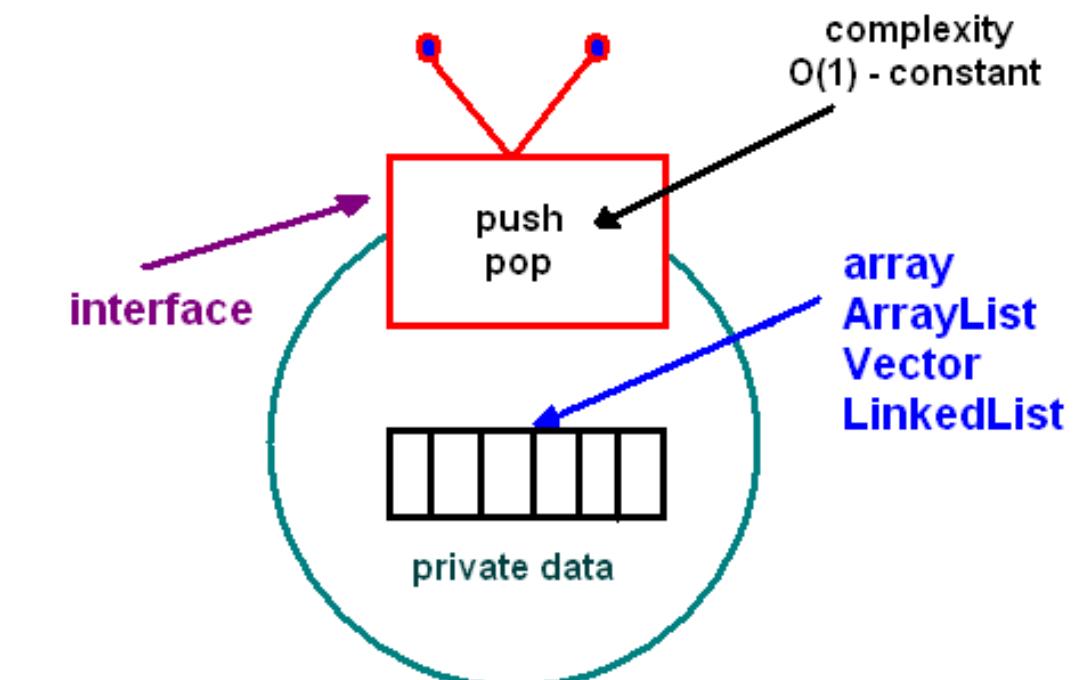
Basic Principles of Object Orientation



Abstraction



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



Encapsulation

How to implement Encapsulation provide private visibility to an instance variable

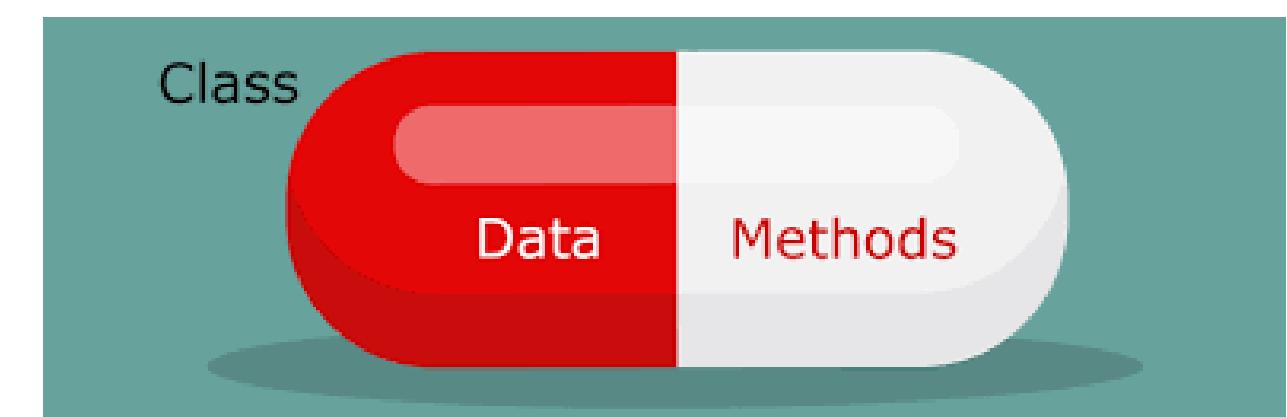
```
class Account{  
    private int account_number;  
    private int account_balance;
```

```
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



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.

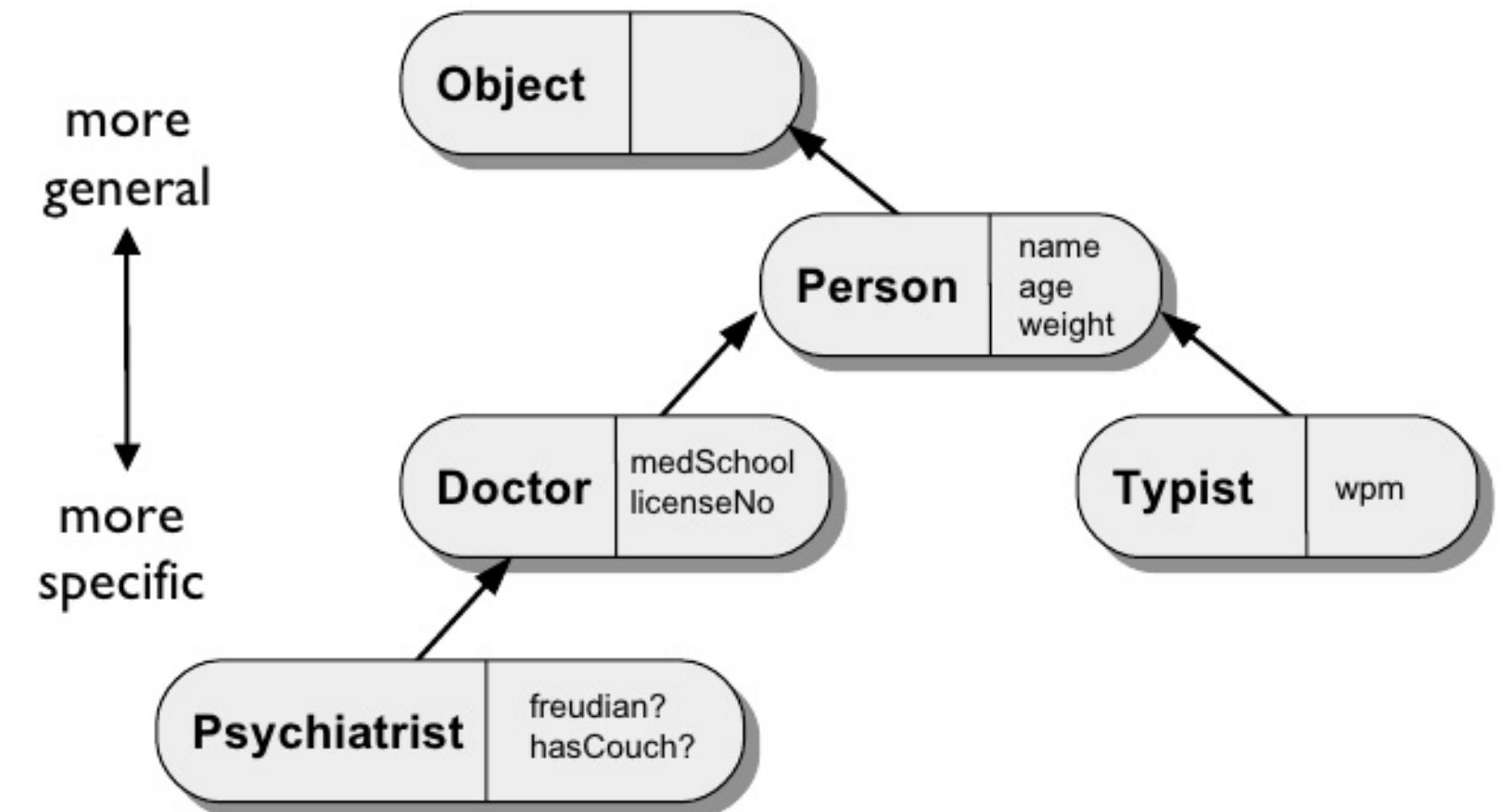


Abstraction vs Encapsulation

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

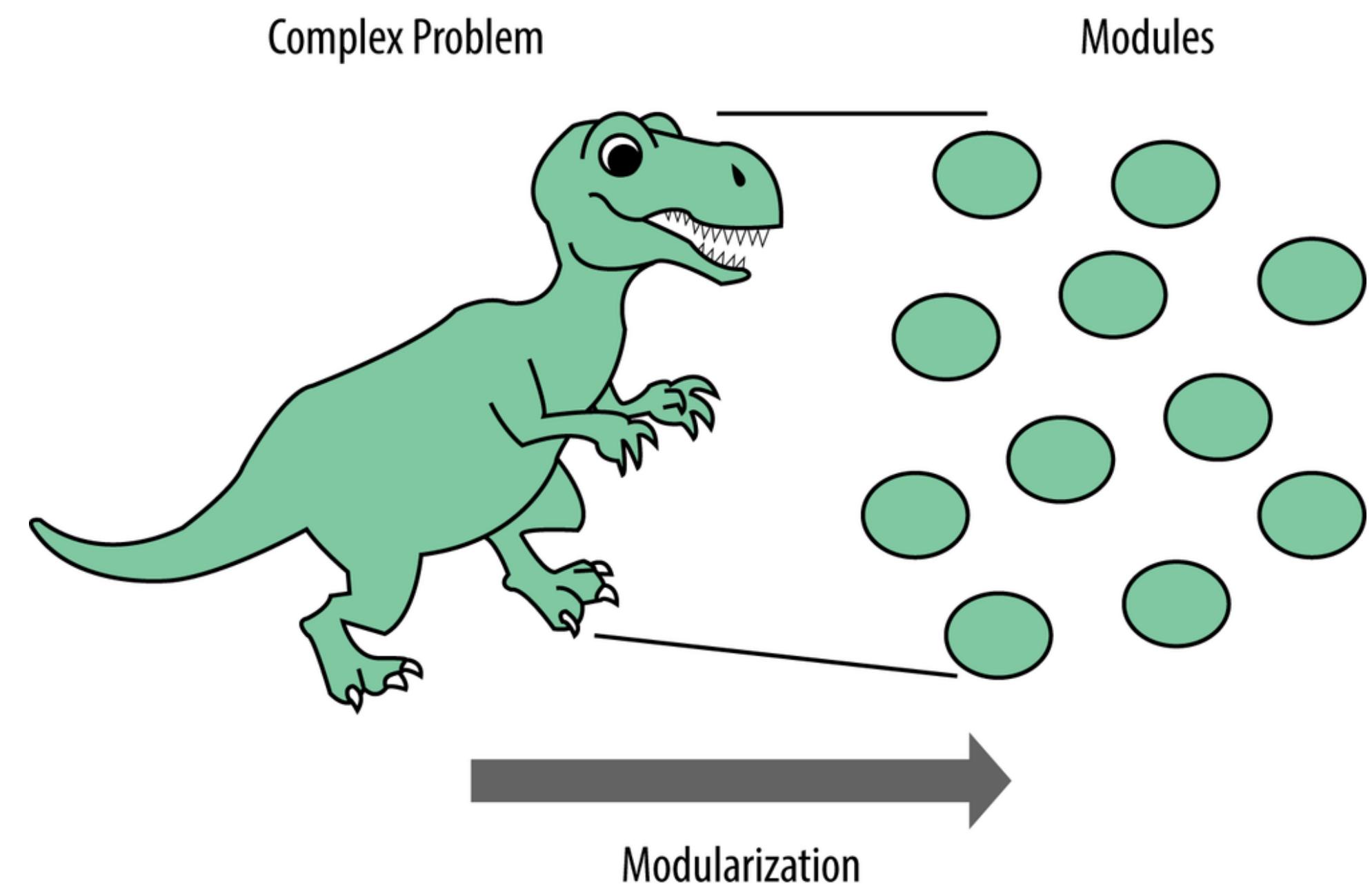
Hierarchy

**Hierarchy give rise
to inheritance and
polymorphism**



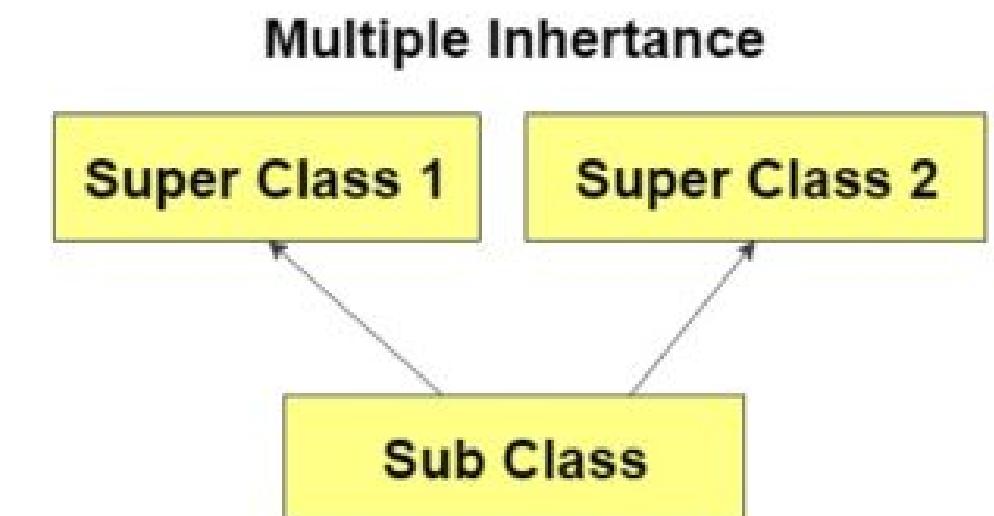
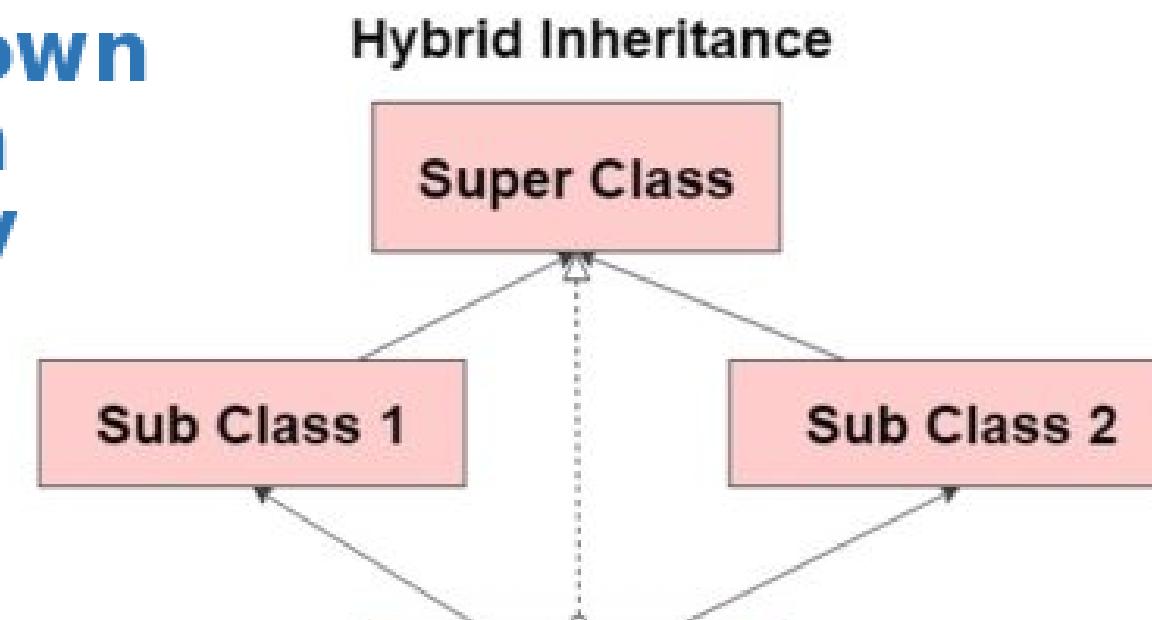
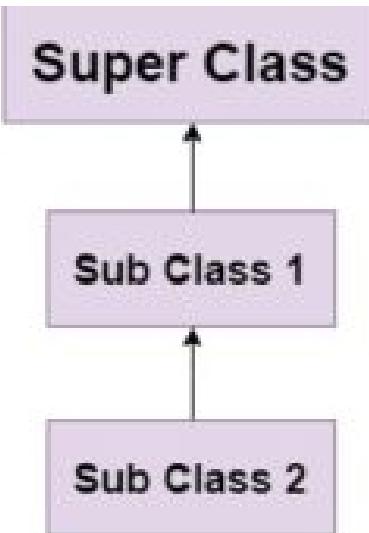
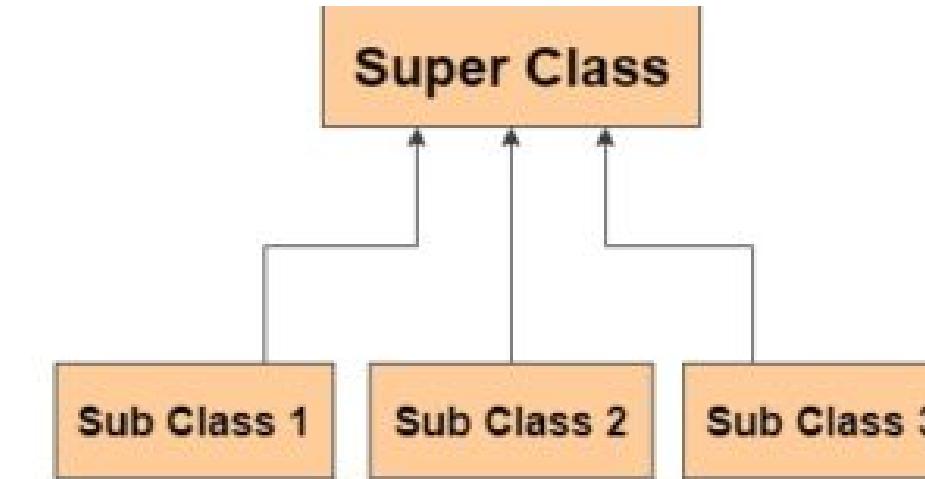
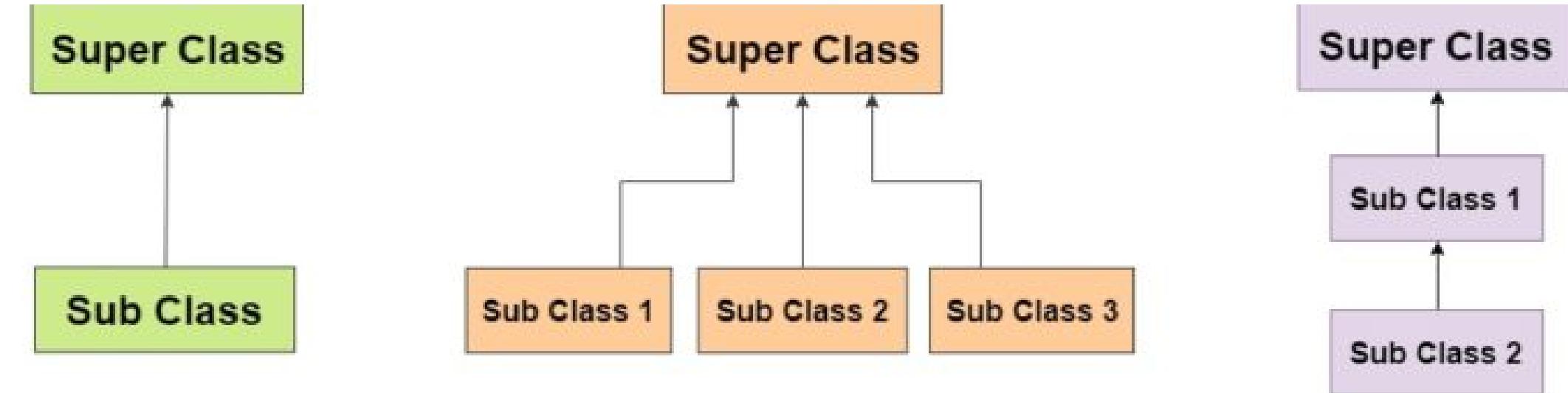
Modularity

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



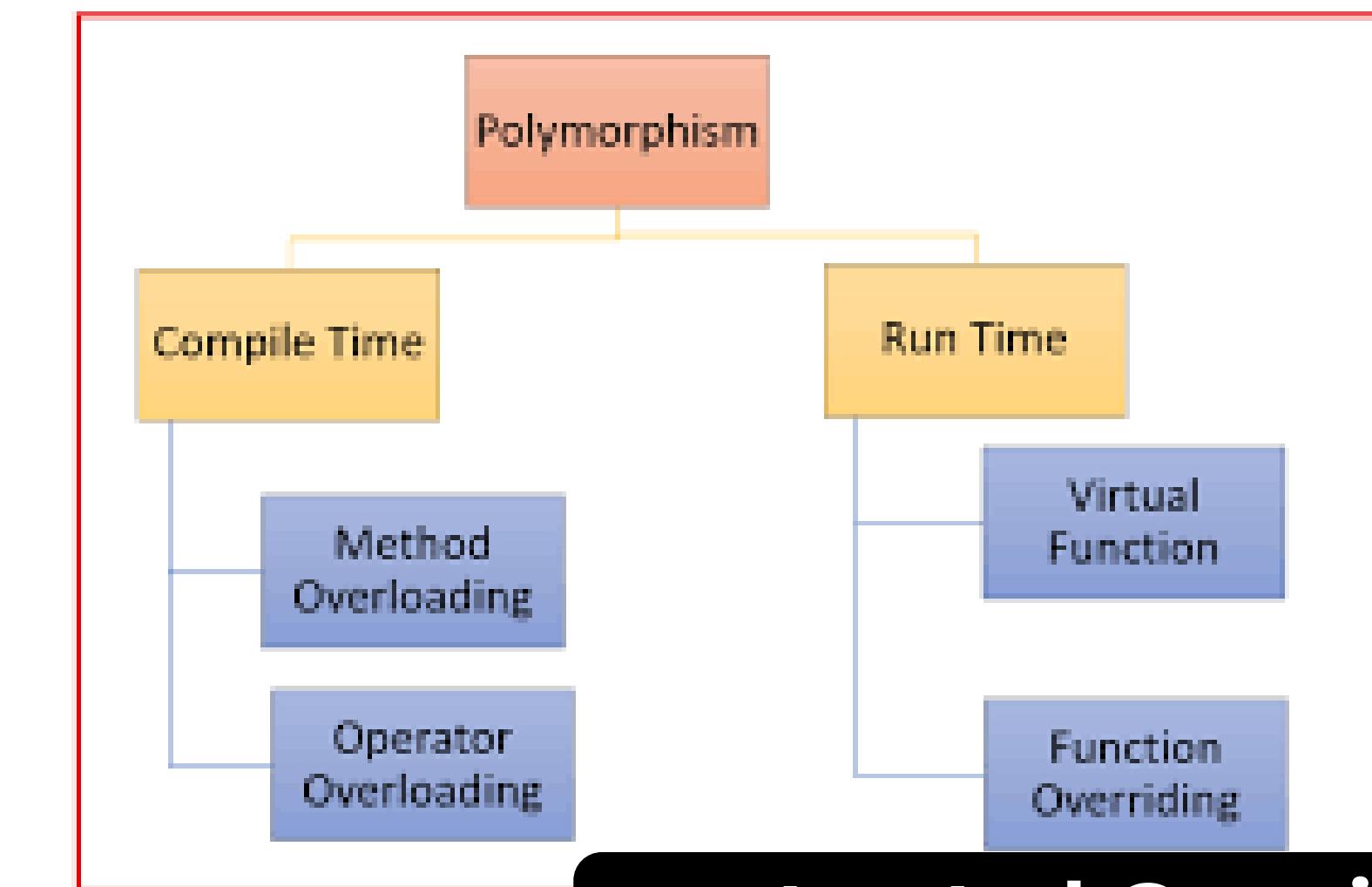
Inheritance

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



Polymorphism

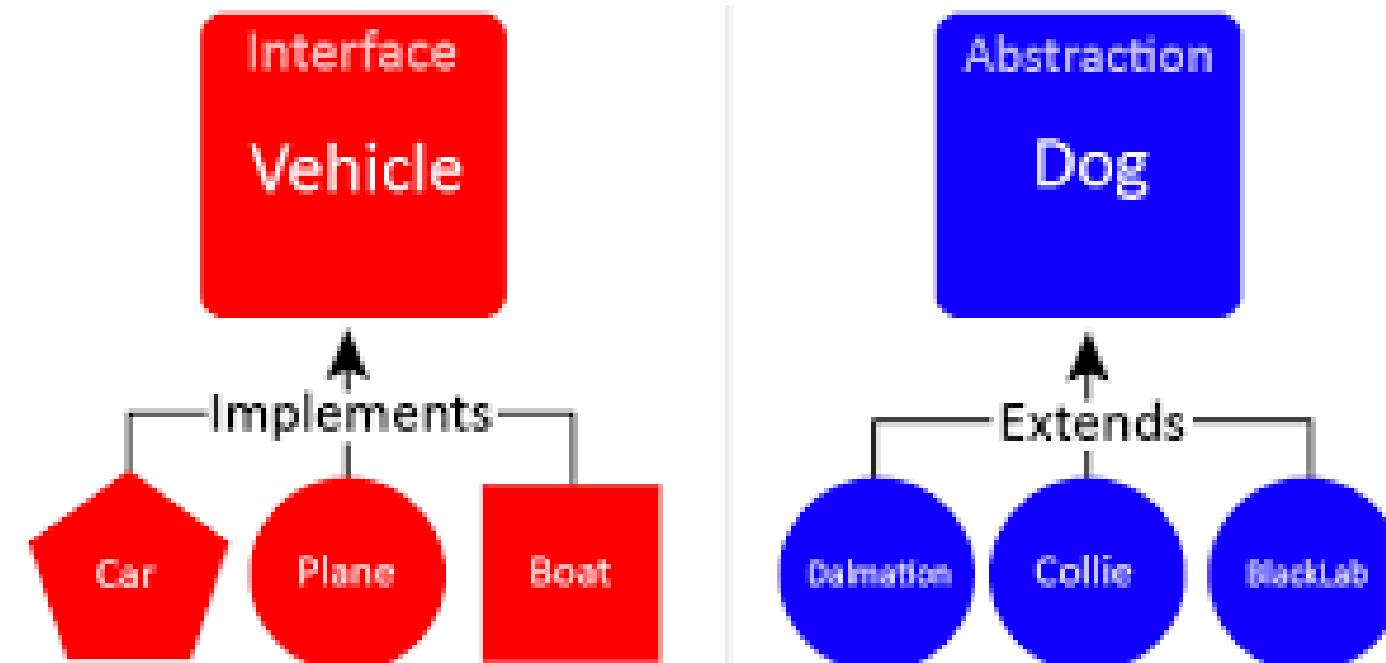
Polymorphism is one of the core concepts of object-oriented programming (OOP) and describes situations in which something occurs in several different forms. In computer science, it describes the concept that you can access objects of different types through the same interface.



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	✓	✗

Interfaces vs. Abstract Classes

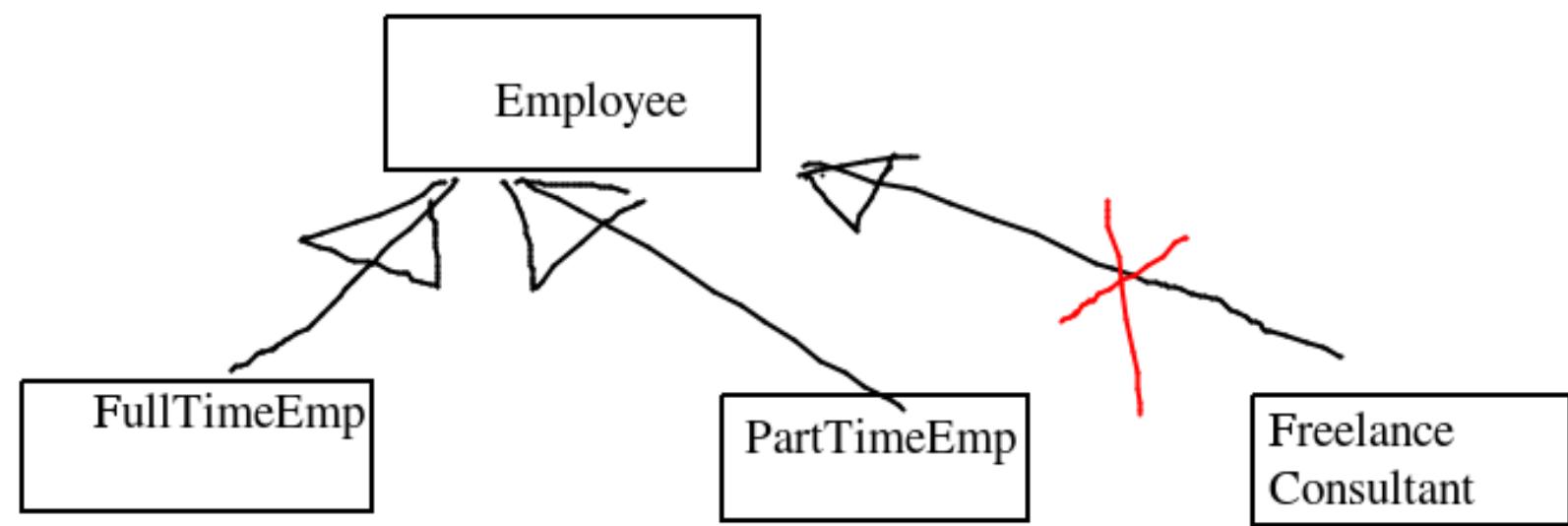


Interface vs Abstract class

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

What an object can do?

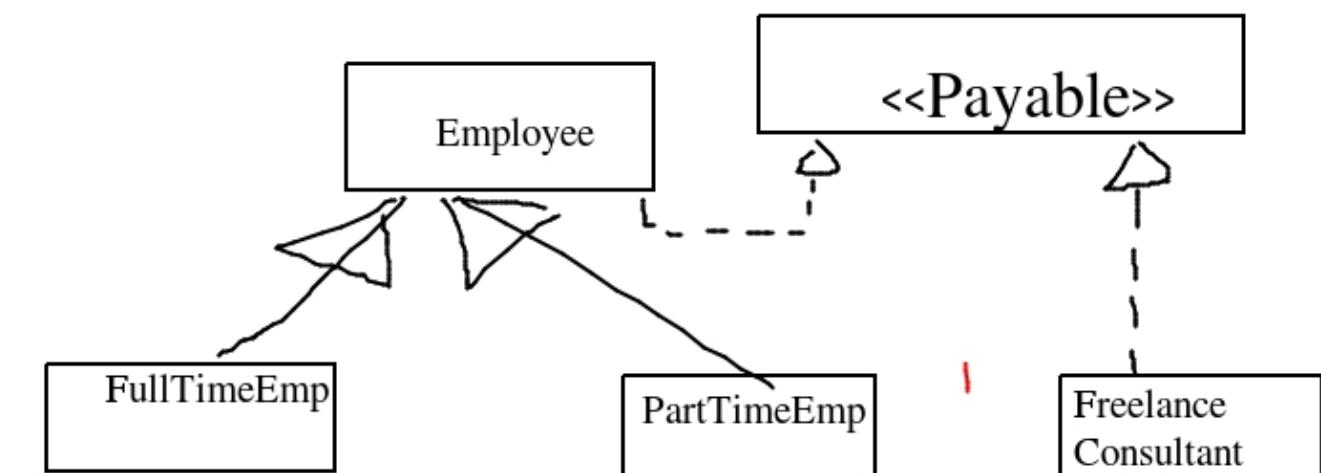
Standardization of a behavior



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

What an object is?

IS-A SavingAccount IS-A Account

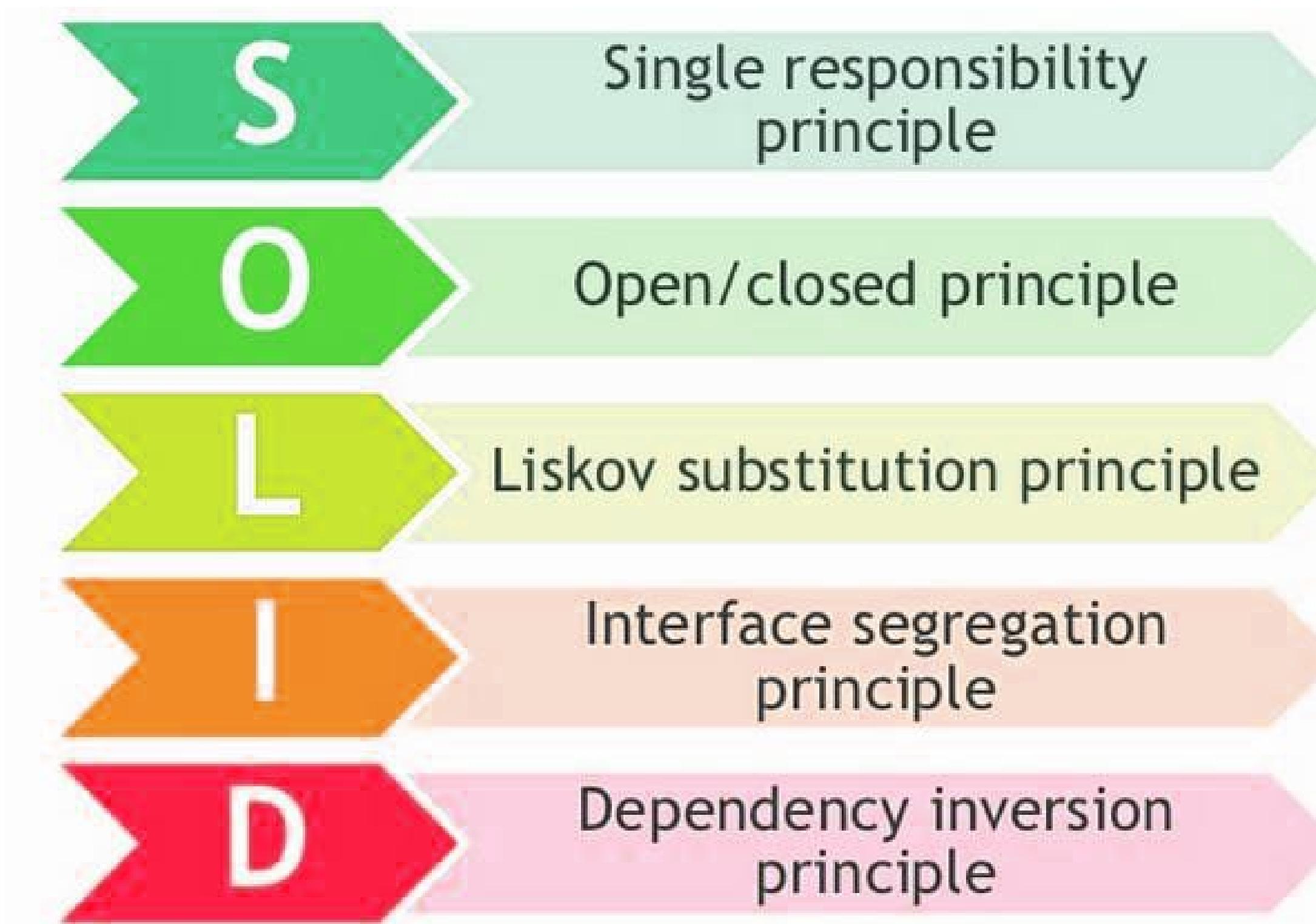


When we should go for interface and when we should go for abstract class?

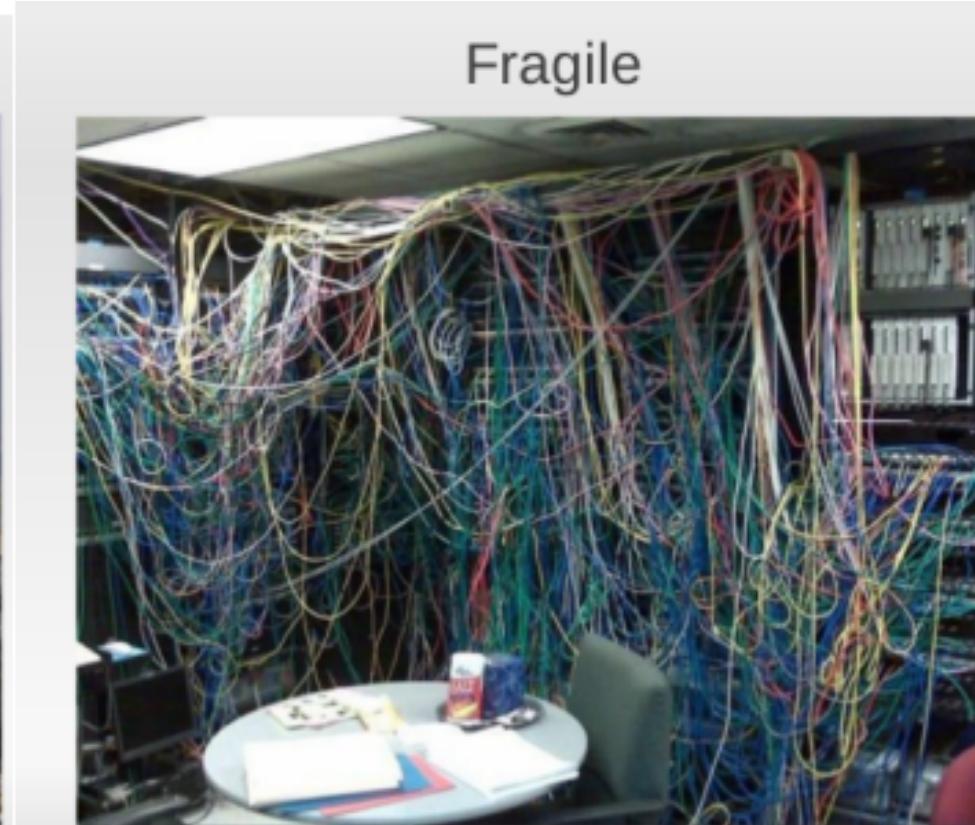
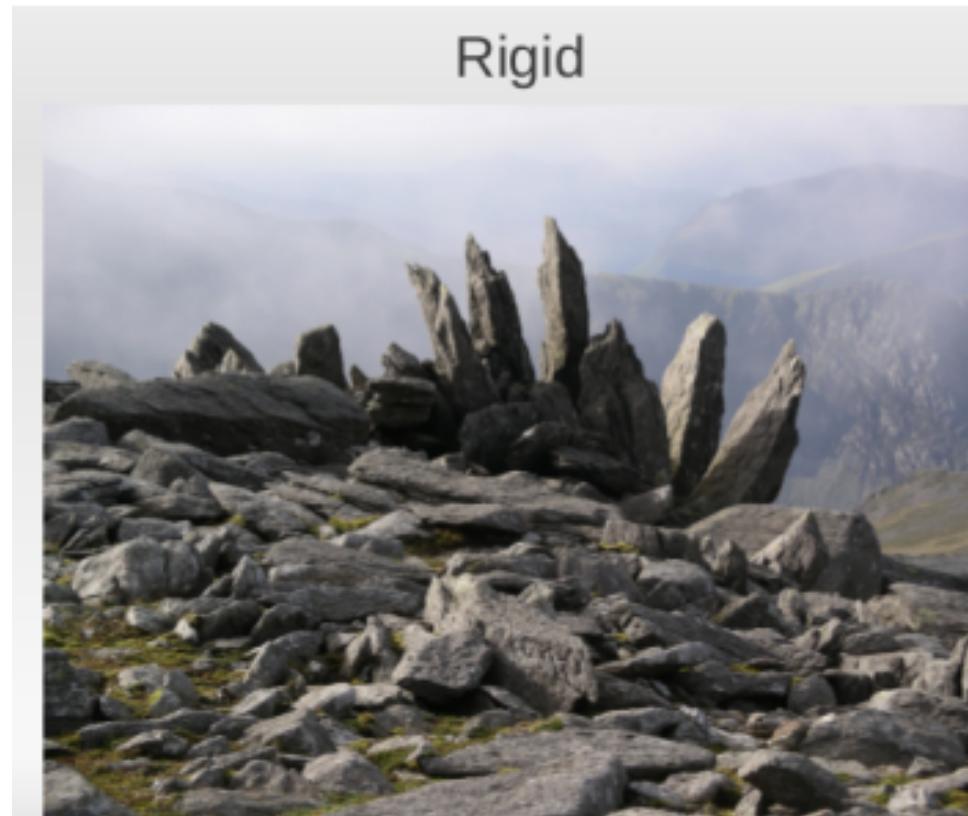
Interface break the hierarchy

rgupta.mtech@gmail.com

SOLID Principles



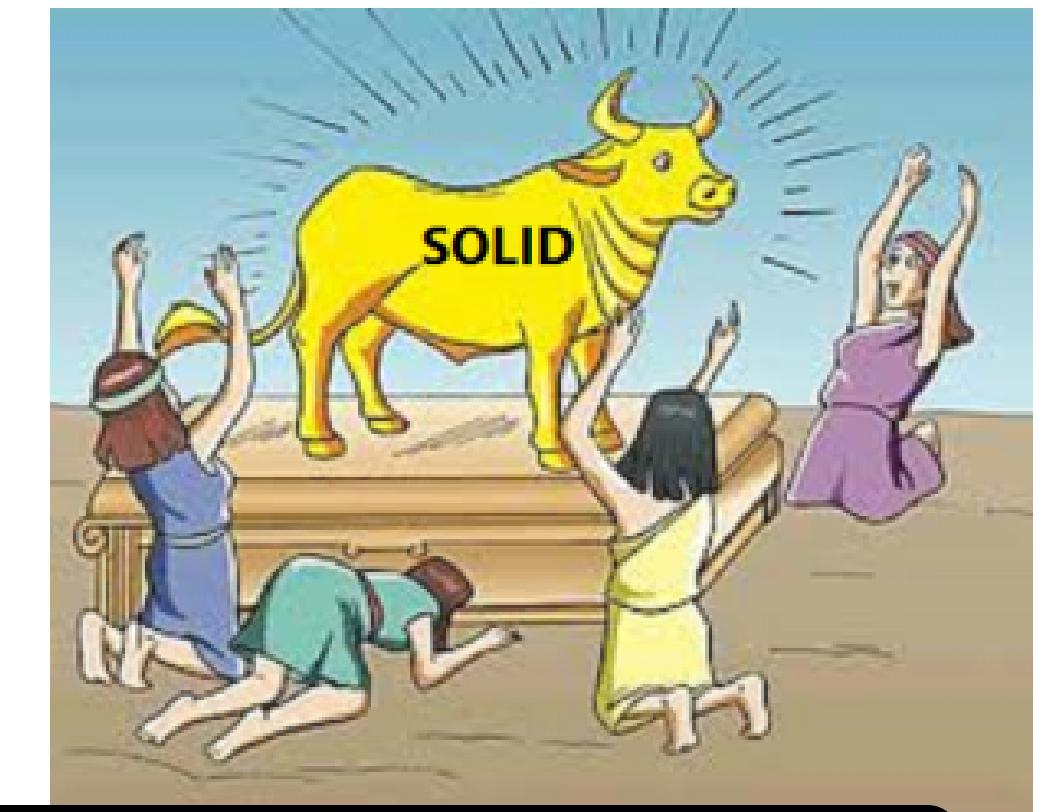
Without Good design our application would be...



rgupta.mtech@gmail.com

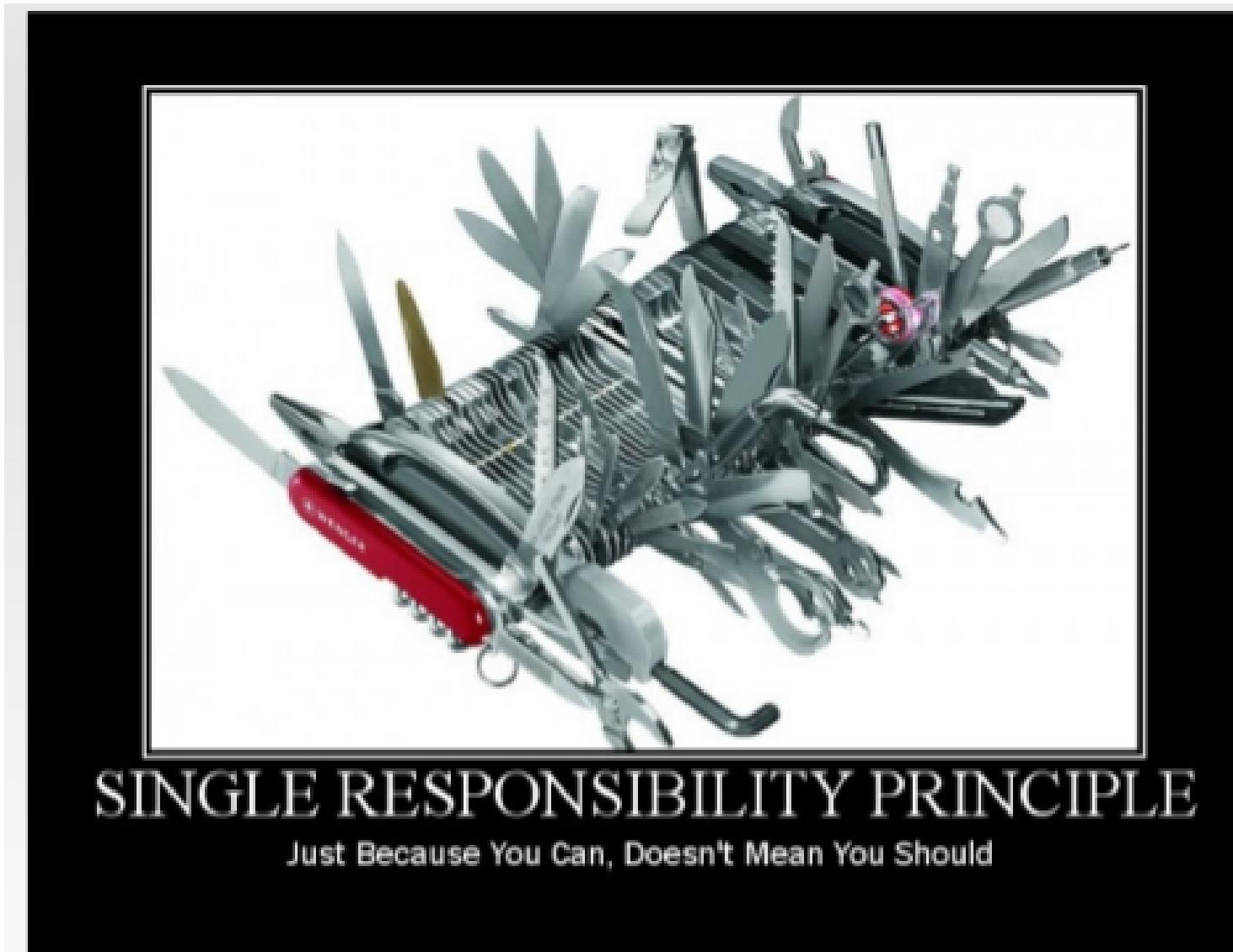
SOLID Principles

- 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**



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"

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 RESPONSIBILITY 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

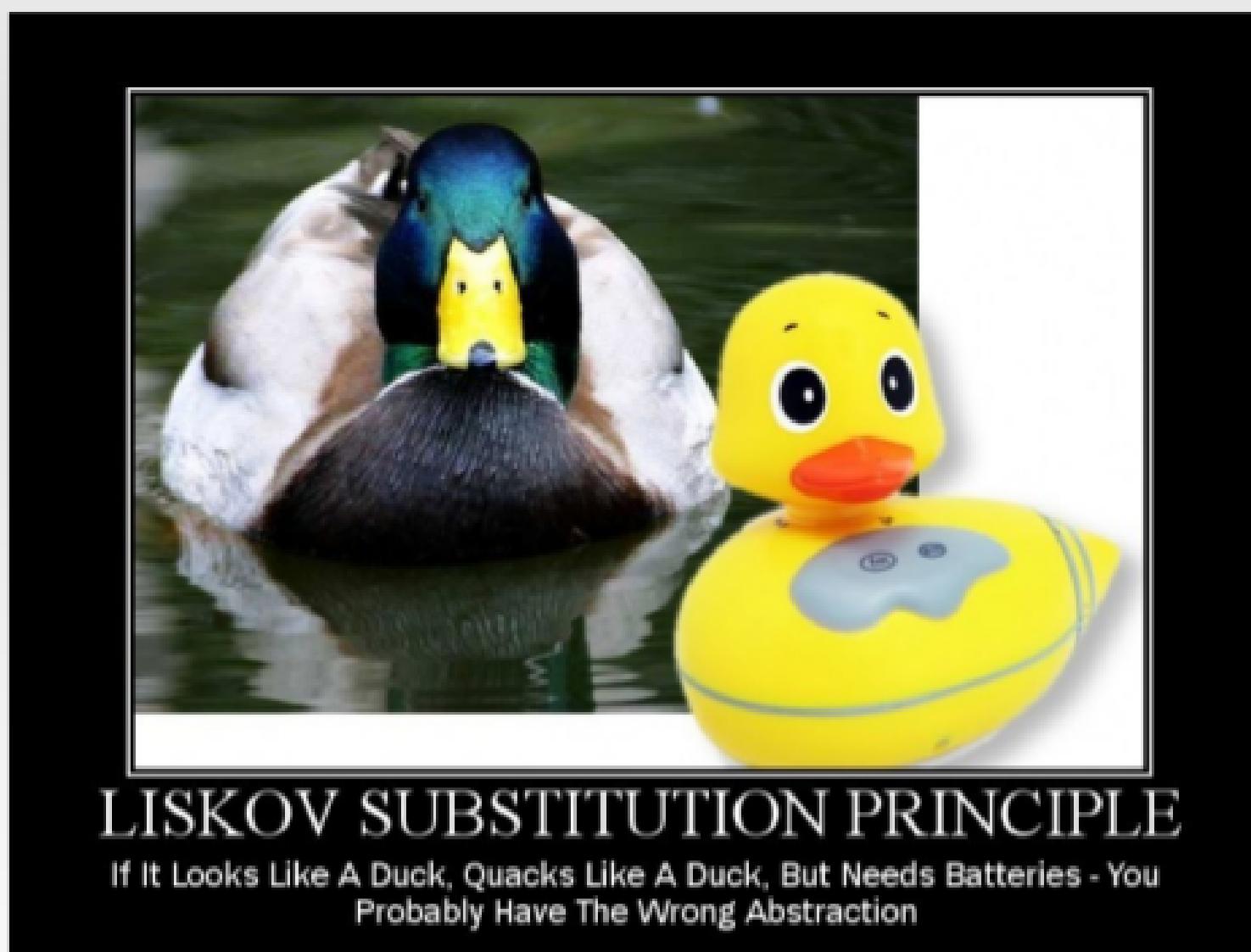


"Modules must be **OPEN** for extension,
CLOSED for modification"

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**”*

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 interfaces they do not use"



How can we pollute the interfaces?
OR

How do we end up creating Fat interfaces?

D – Dependency Inversion Principle

- What is the motivation behind dependency inversion principle?
- **USUALLY THE LOW LEVEL 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"

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

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

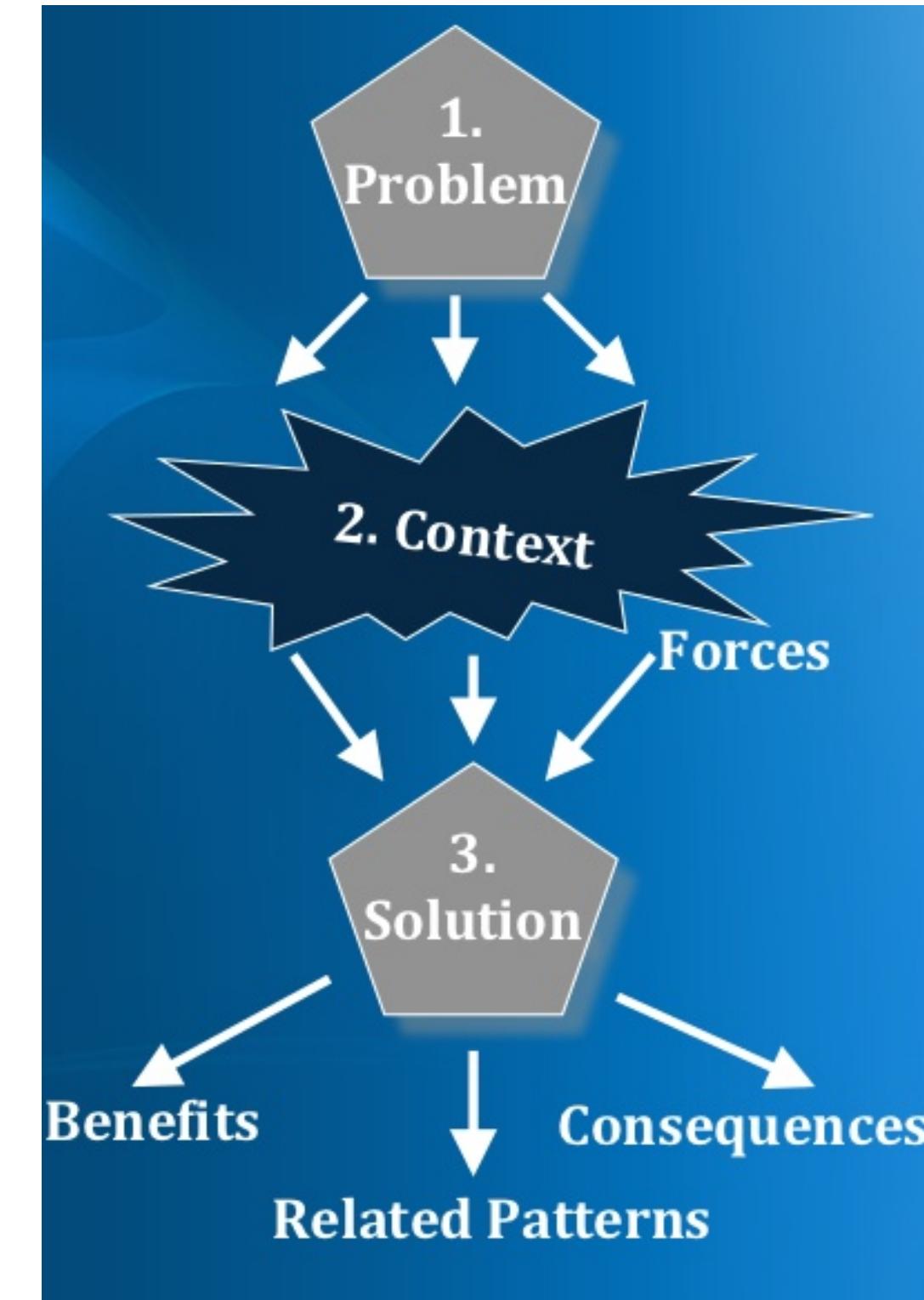
Design Patterns

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.

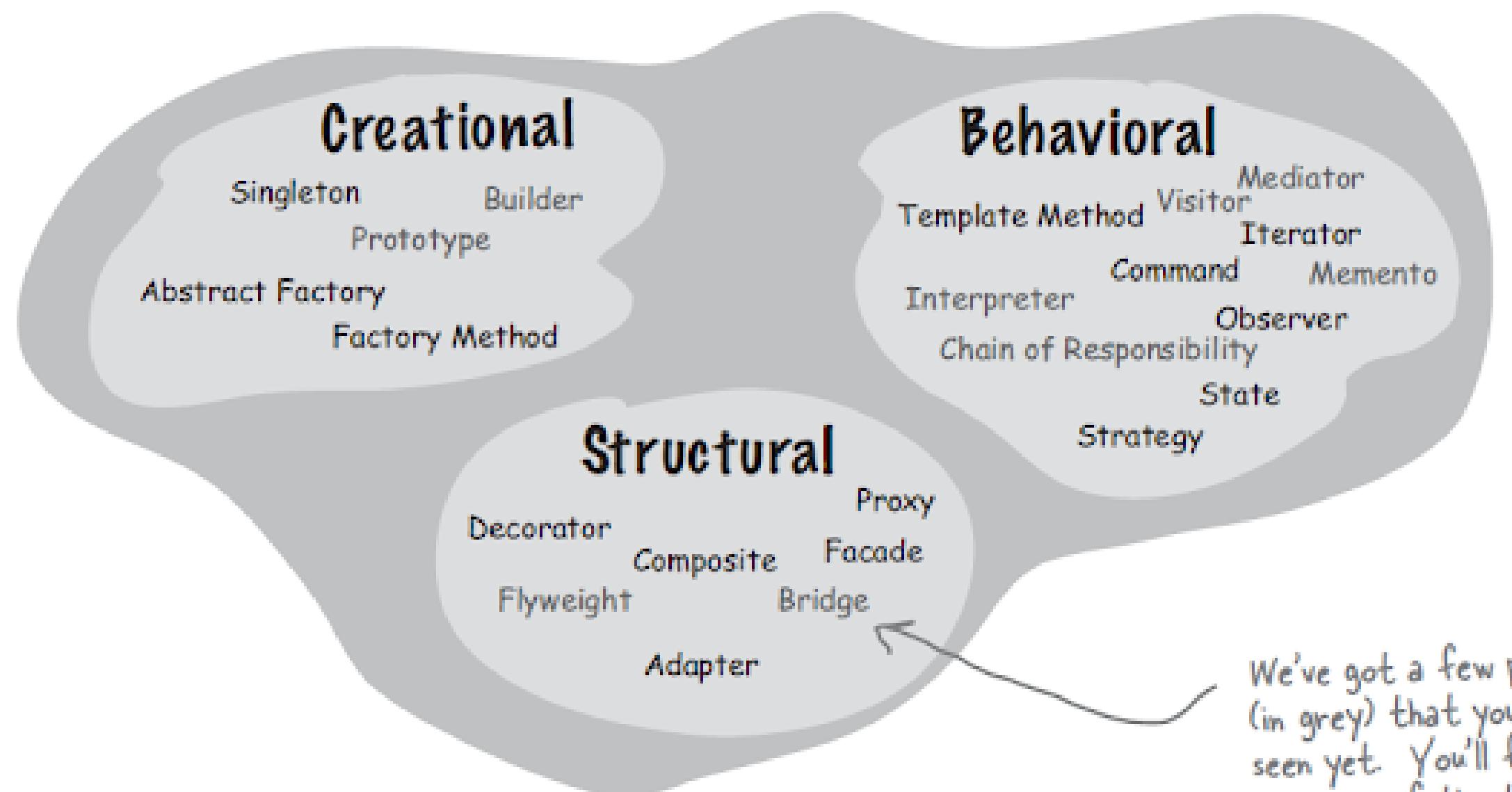
It's not code - I repeat, XCODE. It is like a description on how to tackle these problems and design a solution



Design Patterns

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

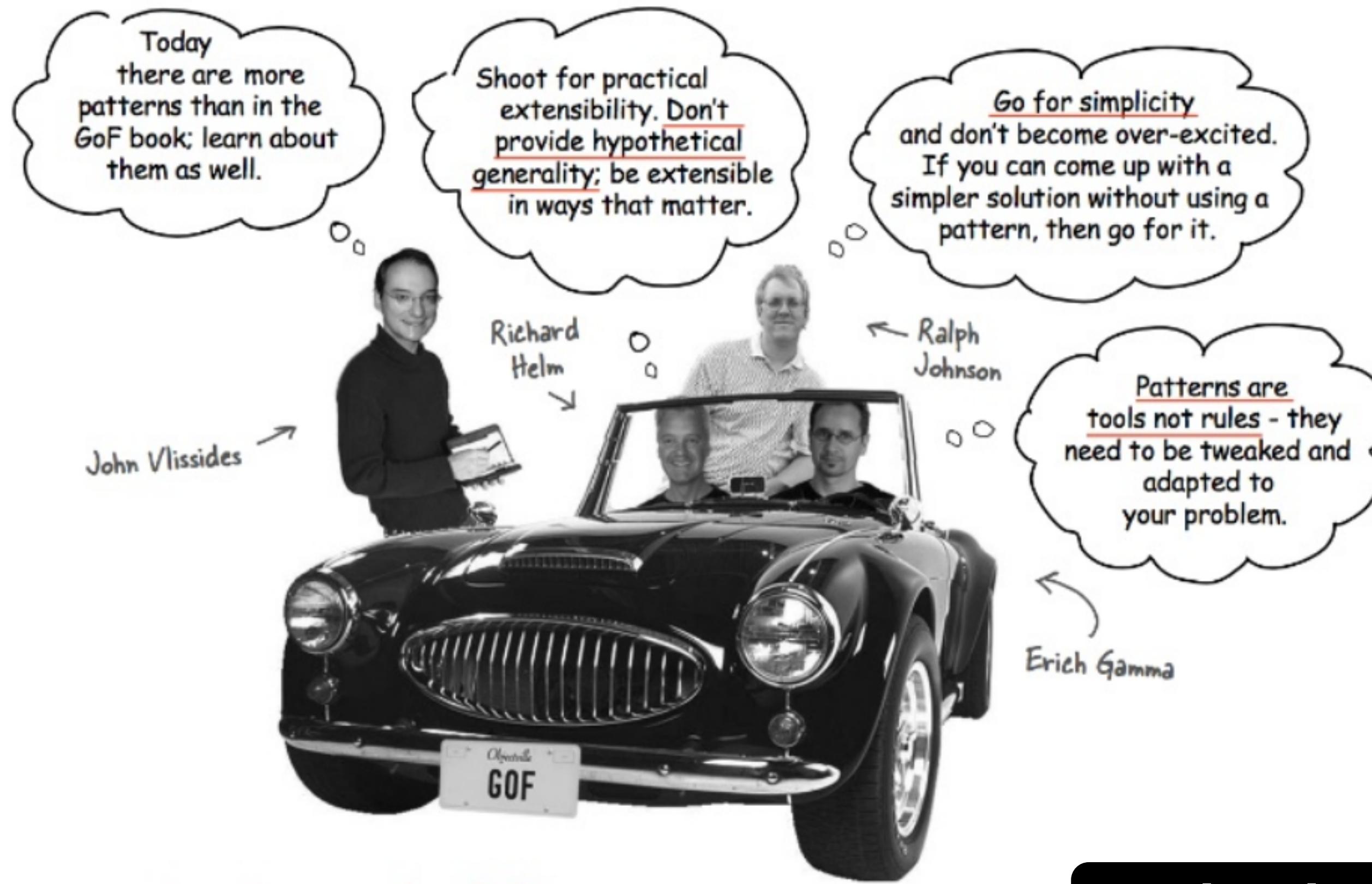
Any pattern that is a Behavioral Pattern is concerned with how classes and objects interact and distribute responsibility.



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



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



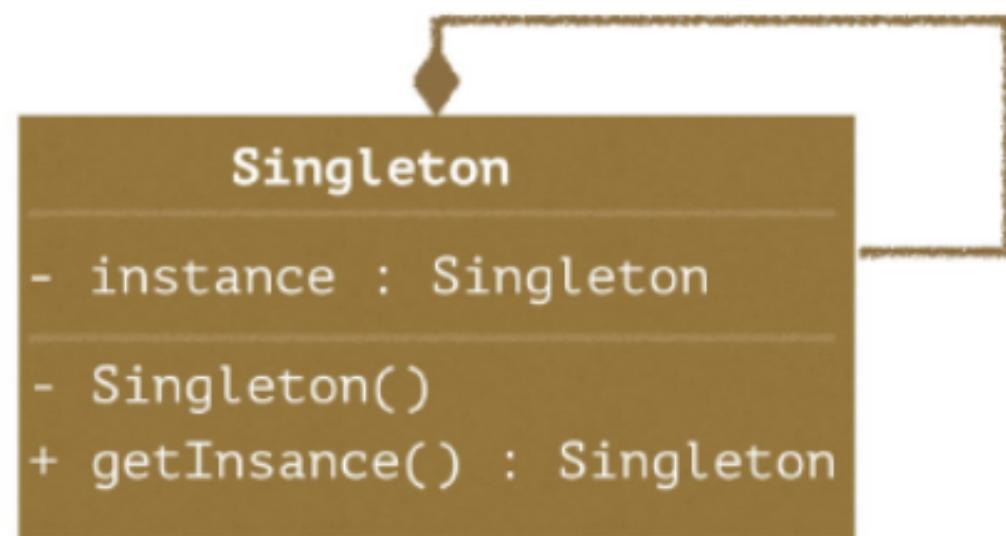
Creationl Design Patterns

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()`



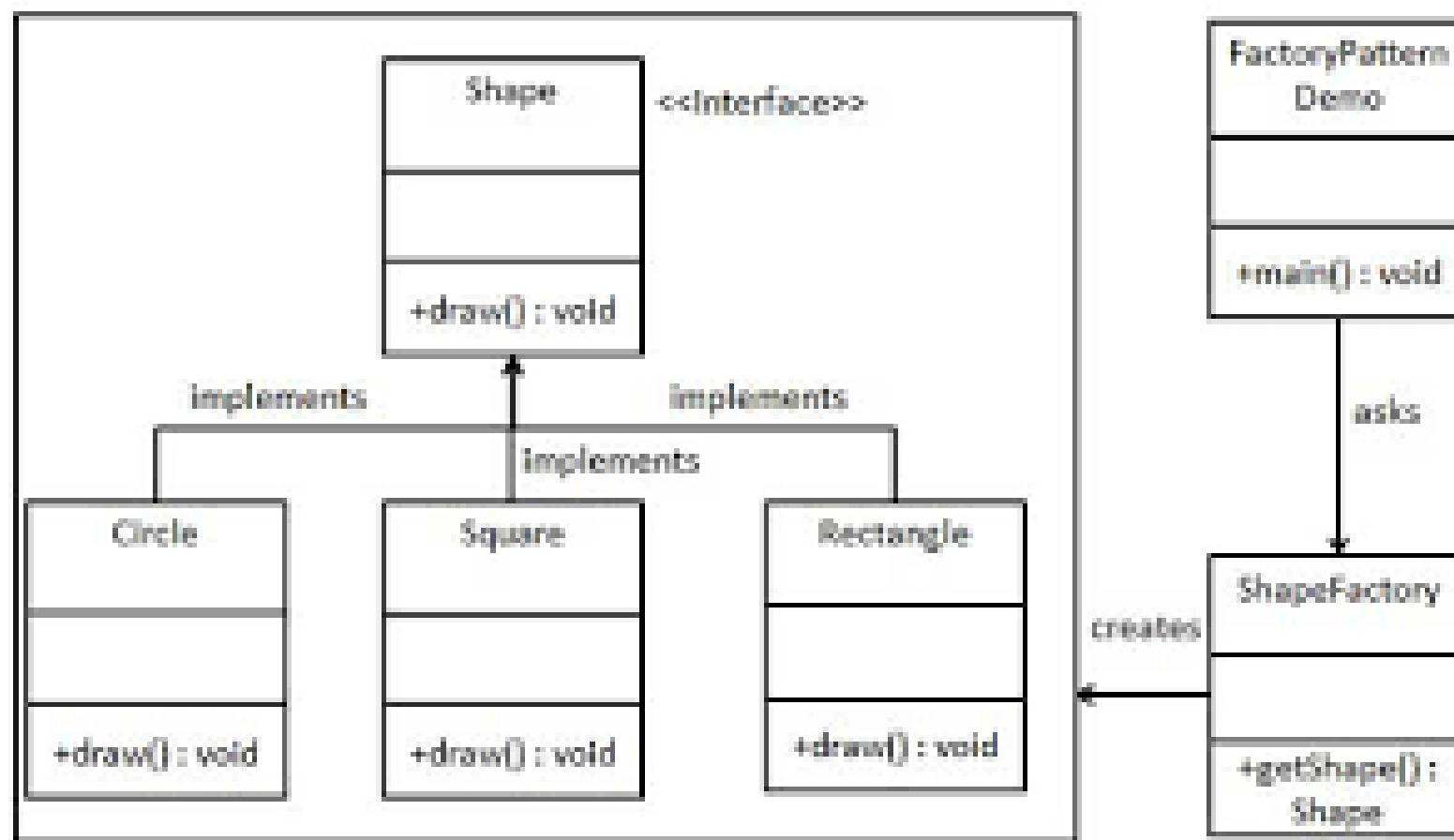
- ❖ Java Runtime class is used to *interact with java runtime environment*.
- ❖ Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc.
- ❖ There is only one instance of `java.lang.Runtime` class available for one java application.
- ❖ The `Runtime.getRuntime()`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

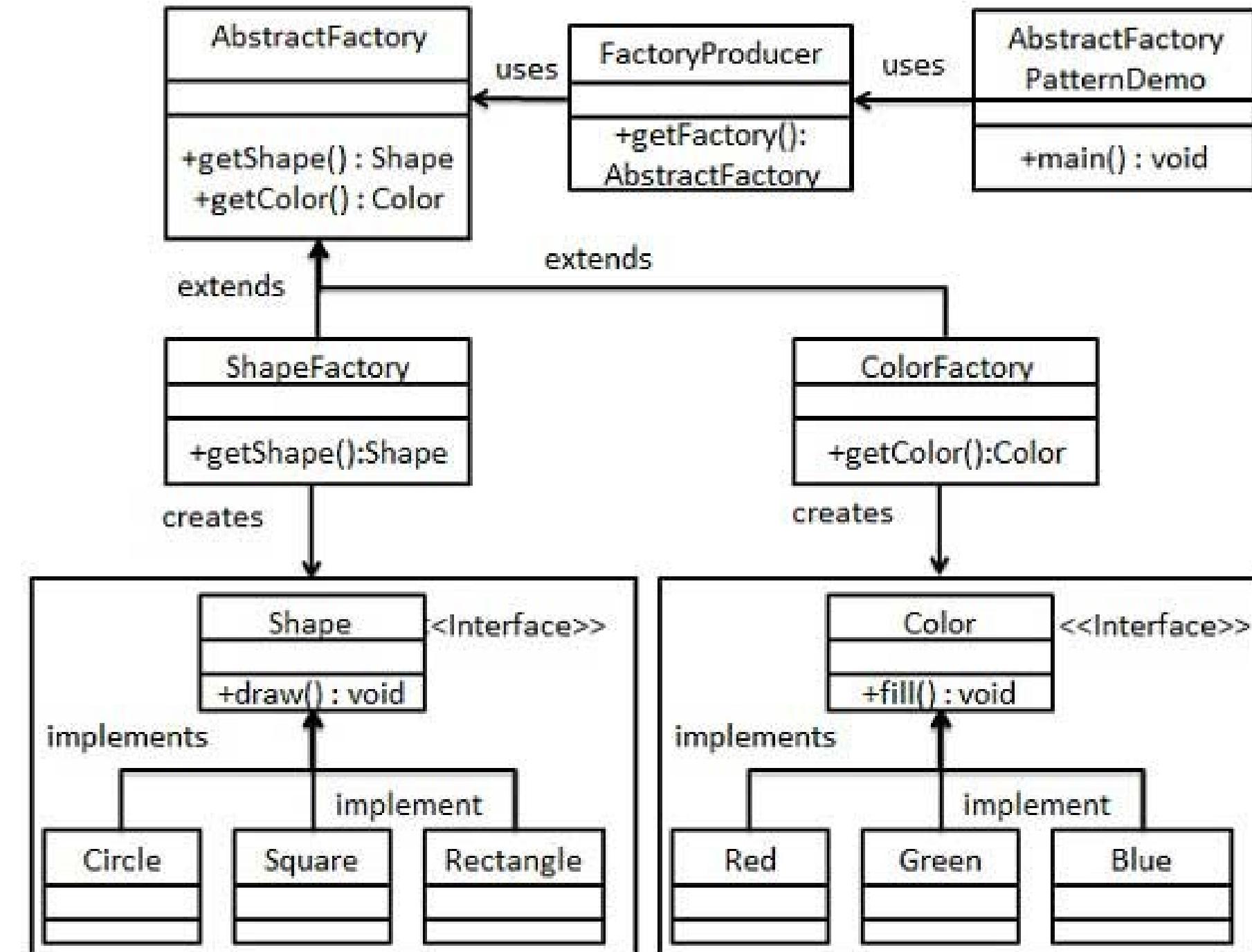
- [java.util.Calendar#getInstance\(\)](#)
- [java.util.ResourceBundle#getBundle\(\)](#)
- [java.text.NumberFormat#getInstance\(\)](#)

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.



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

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)

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

```
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 dateOf  
        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;  
    ...  
}
```

telescoping
constructors

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`)

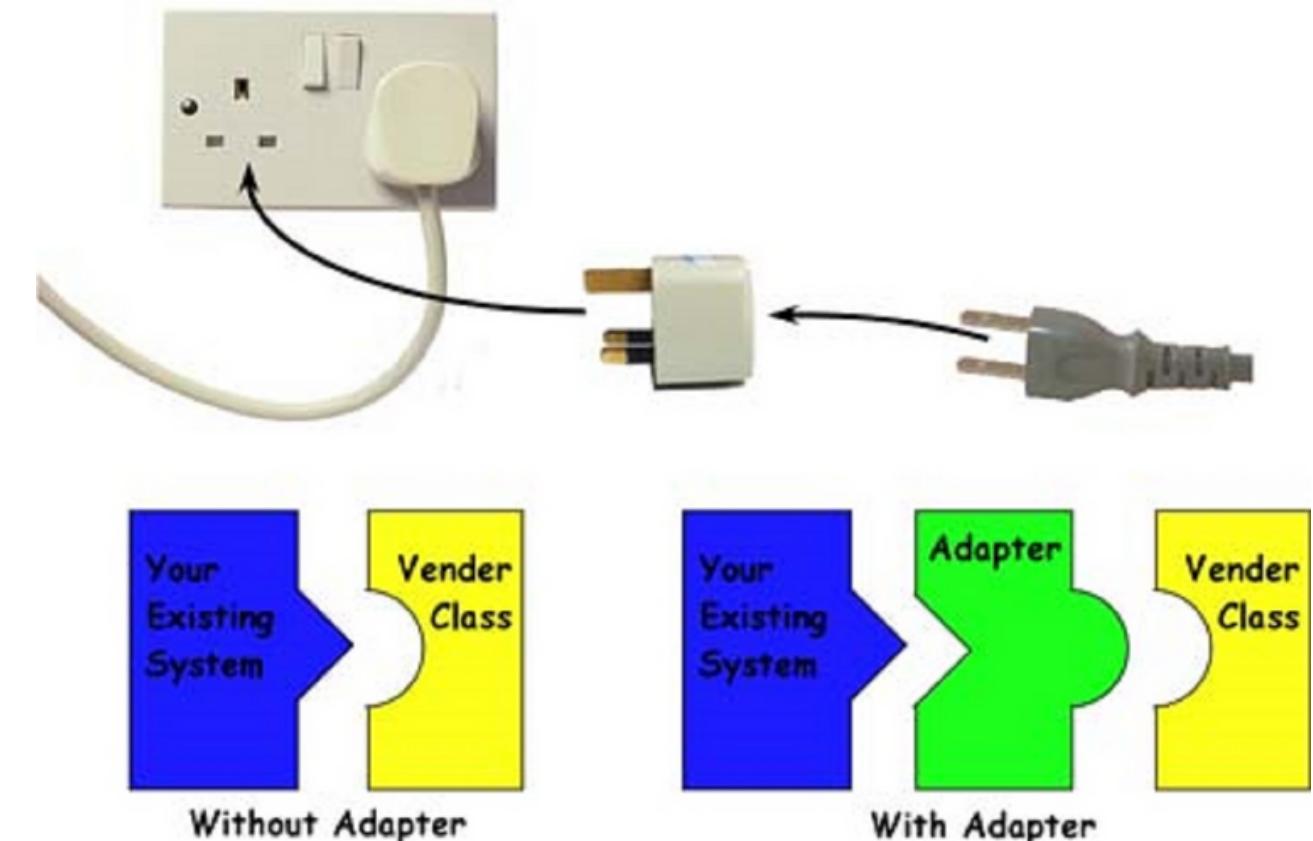


Structural Design Patterns

rgupta.mtech@gmail.com

Adapter Pattern

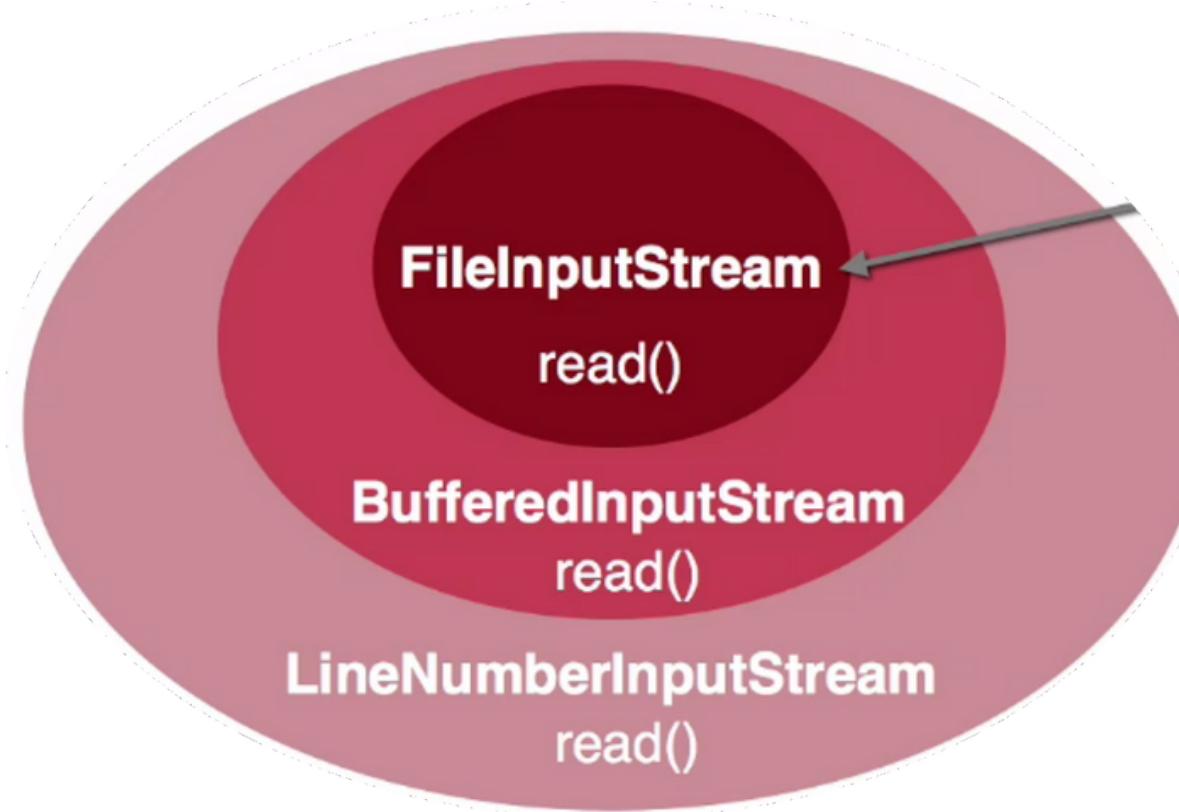
The Adapter pattern is used so that two unrelated interfaces can work together.



- `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`)

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.

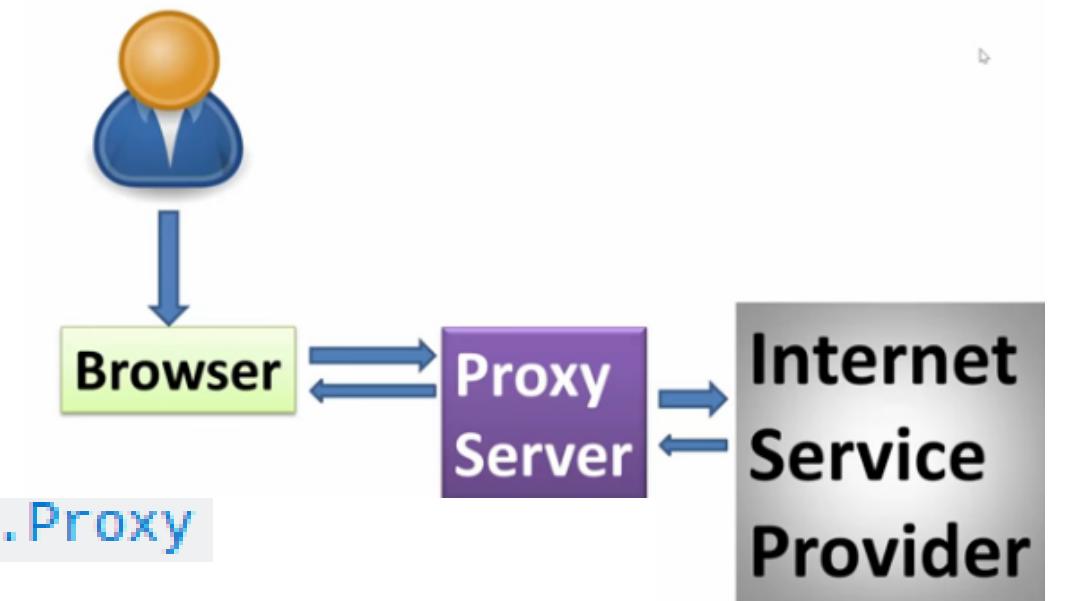
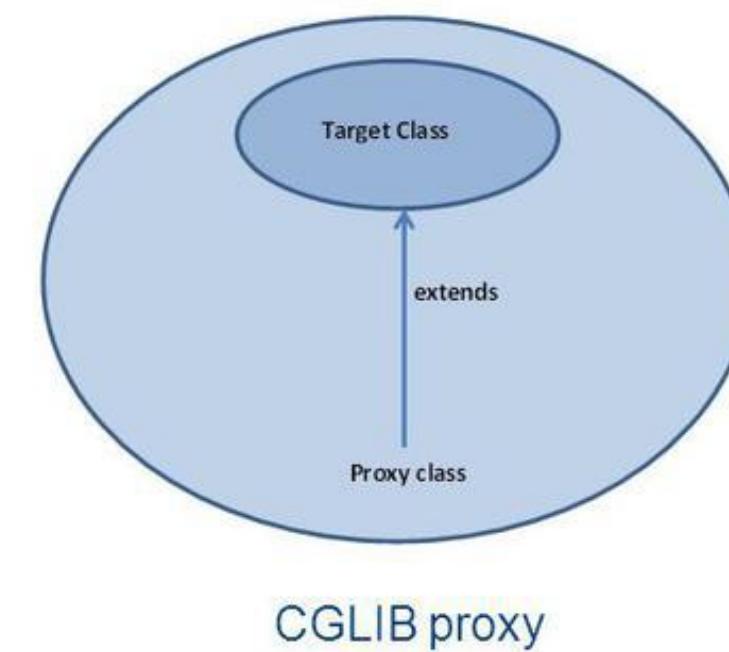
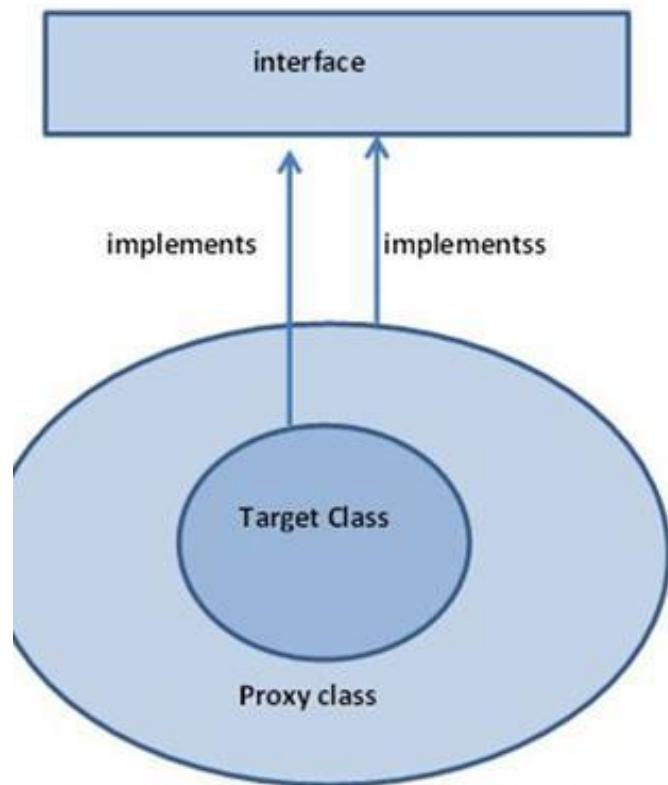
`java.util.Collections`, the `checkedXXX()`, `synchronizedXXX()` and `unmodifiableXXX()` methods.

`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.*](#)

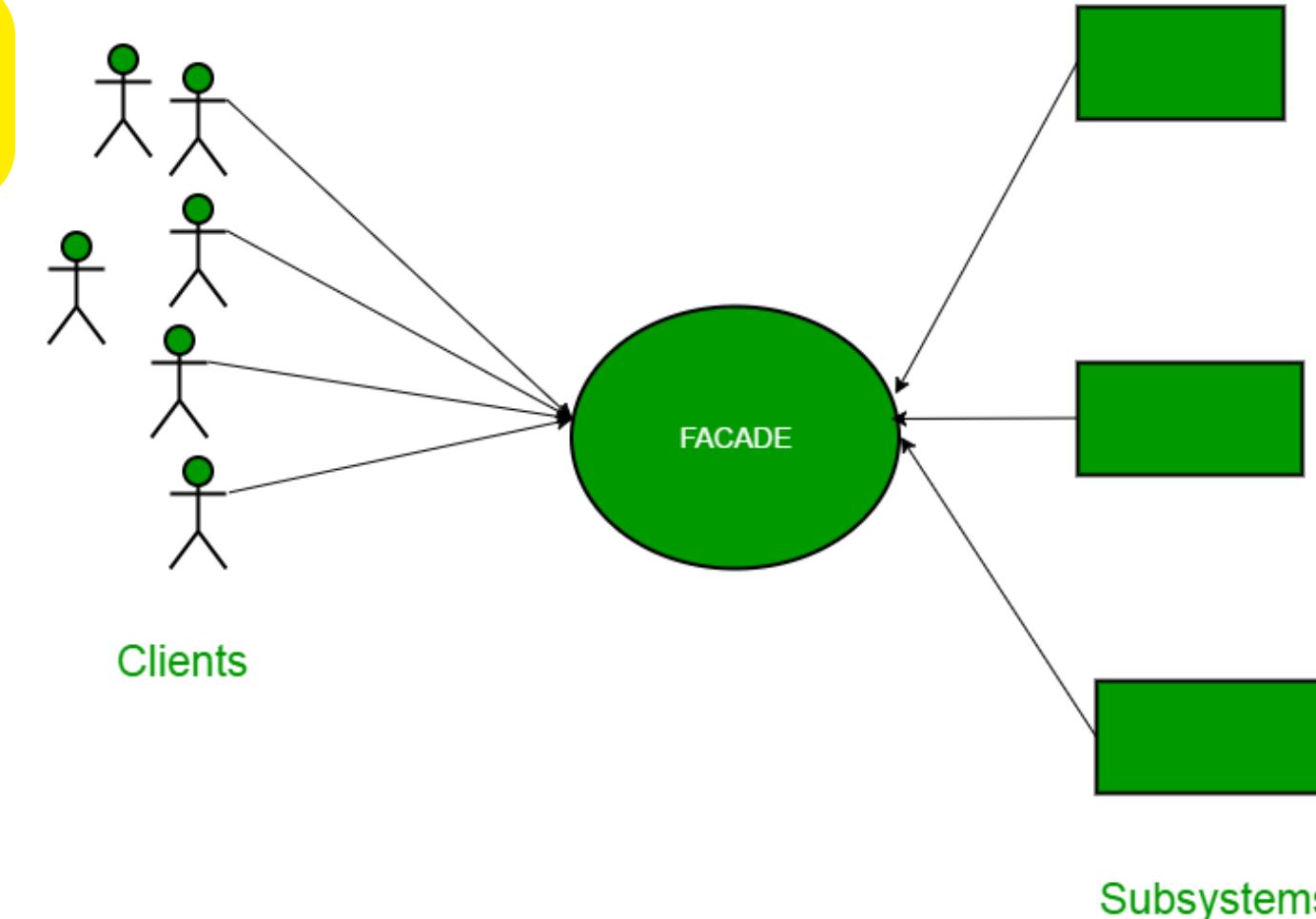
[javax.ejb.EJB](#) (explanation here)

[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.





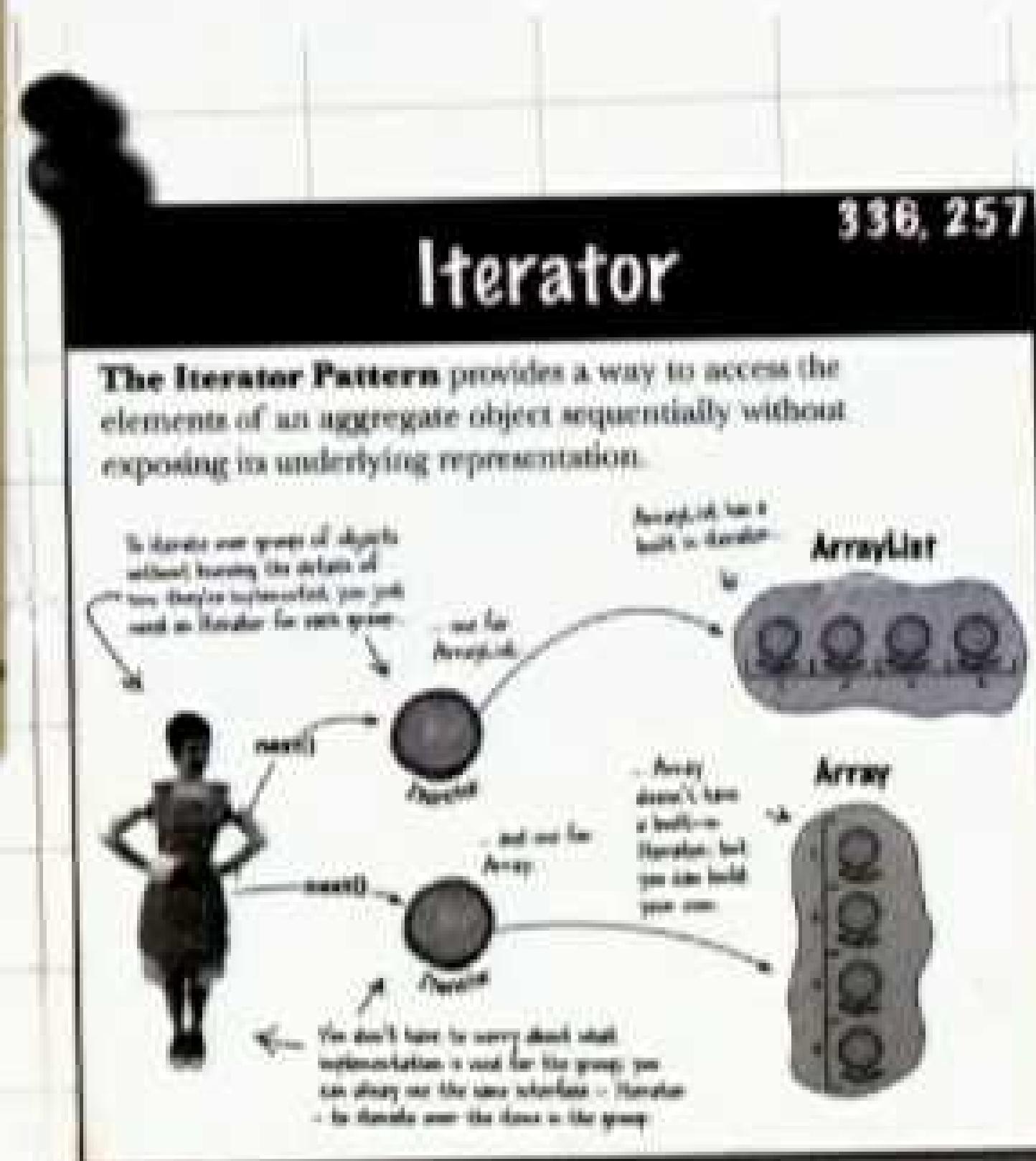
Behavioral Design Patterns

rgupta.mtech@gmail.com

Iterator Pattern

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

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



Strategy pattern /policyPattern

< continue shopping

Your Shopping Bag

Prefer to shop in our boutiques?
Take this list with you!

EMAIL PRINT

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

CHECKOUT

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](#)

Need Help?

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

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

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

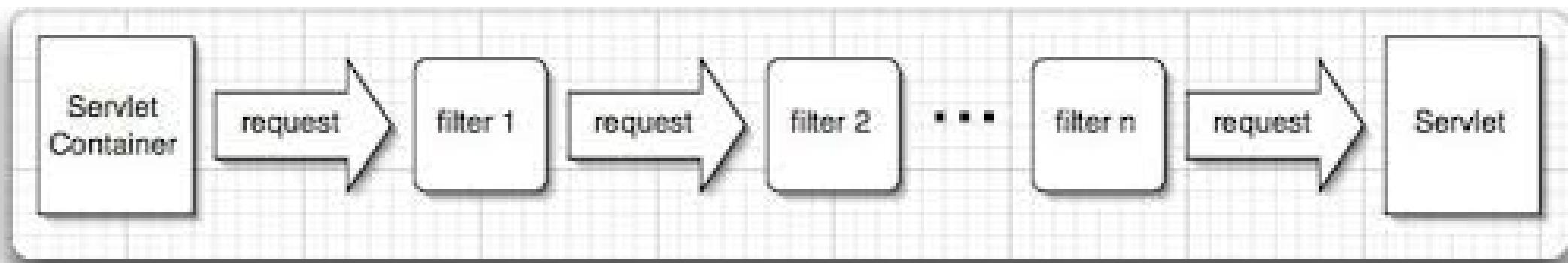
- `java.util.Comparator#compare()`, executed by among others `Collections#sort()`.
- `javax.servlet.http.HttpServlet`, the `service()` and all `doXXX()` methods take `HttpServletRequest` and `HttpServletResponse` and the implementor has to process them (and not to get hold of them as instance variables!).

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

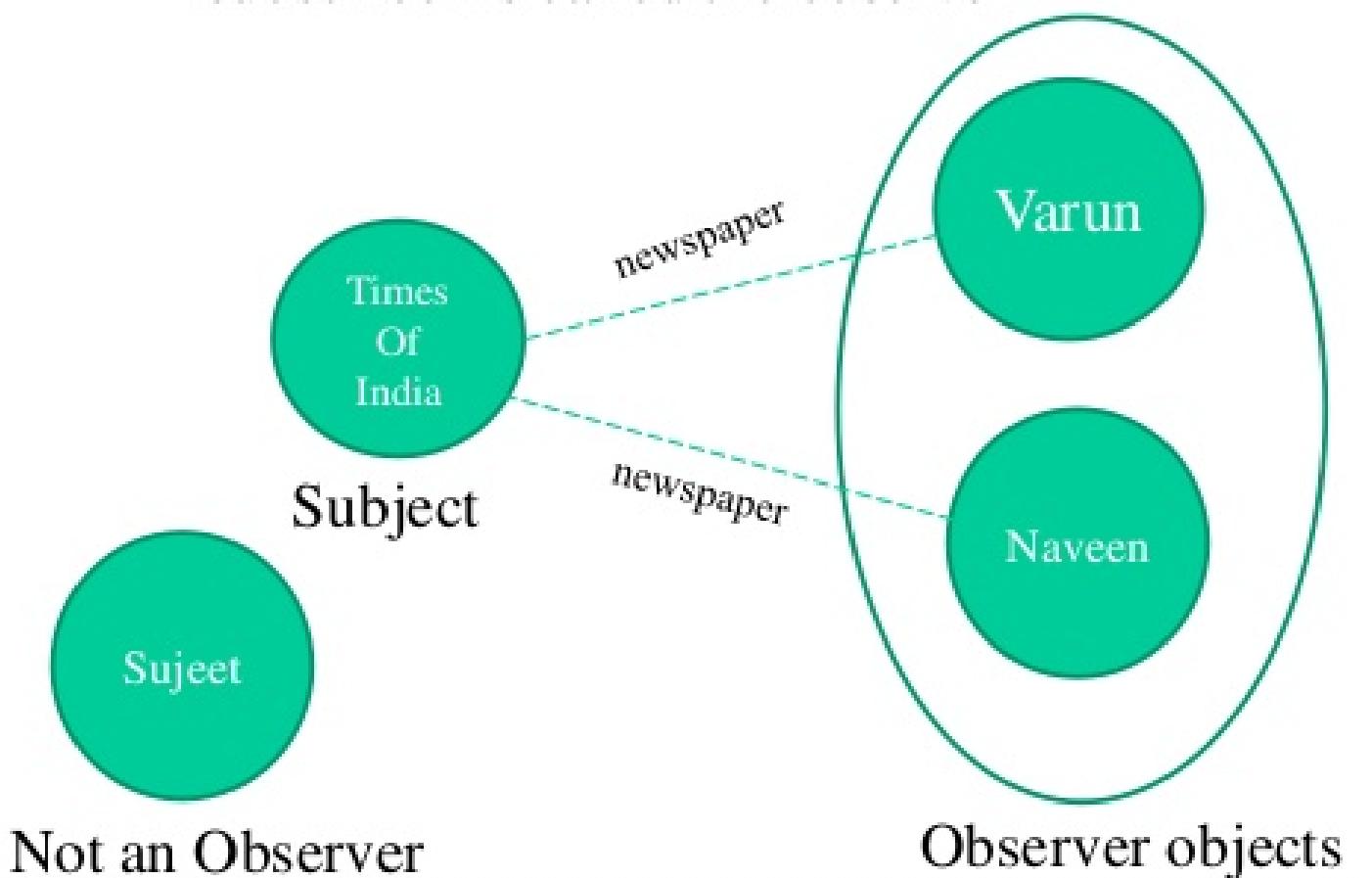


Observer Design Pattern

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.

- `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`

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



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 Siriporn Road, Nonthaburi, Thailand 10140215 Tel: (02) 3234 8878 Fax: (02) 3234 8878
Email: jlpco@jlpco.co.th Website: www.jlpco.com

Customer Feedback Form

Dear Valued Customer,
Thank you for choosing JONNY LEVER PARCEL INTERNATIONAL. We are glad to invite you to tell us about your experience with our services.

Your feedback is extremely valuable to us. Your honest feedback will help us to serve you better and enable us to work on improving our service standards. Thank you.

Customer Name:

Address:

Destination:

Account:

Rating: Bad Poor Fair Good Excellent

1. Service's management and assistance
2. Order's punctuality
3. How would you rate the quality of our product?
4. JLP's professionalism and attitude
5. Completion of job without breakdown
6. Address & delivery of items
7. Was the job item acceptable?
8. How will you rate our overall quality of our packing and moving?
9. How would you like to recommend us to others?

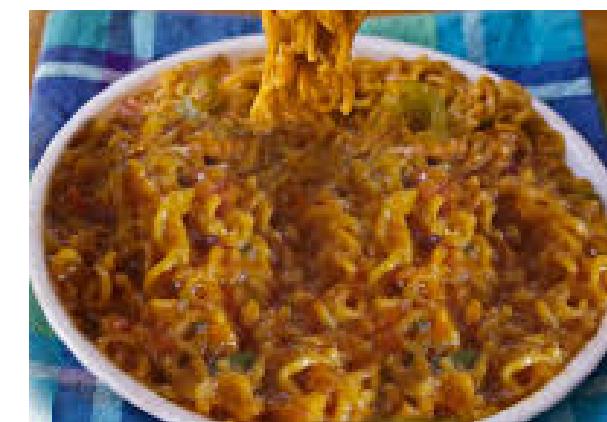
Your comments: _____

Signature: _____ Date: September 03, 2013

spring
JDBC

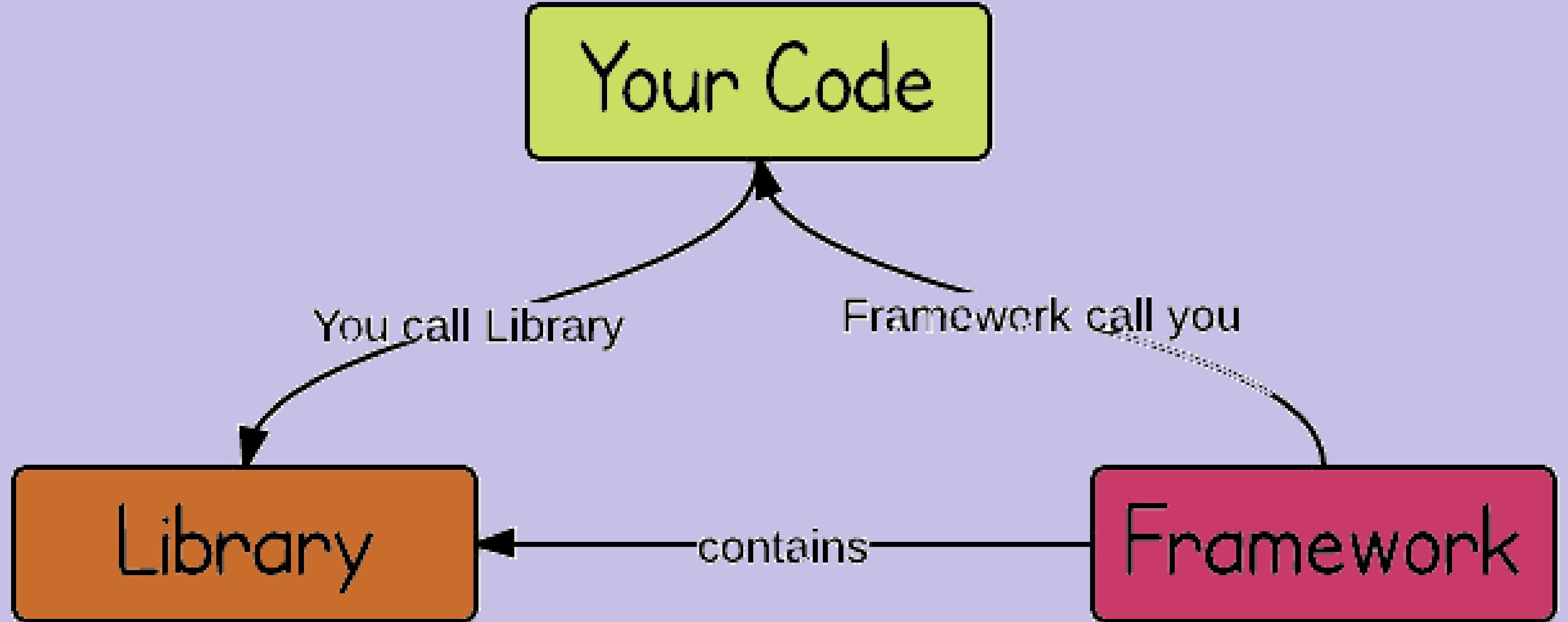
Pattern vs Framework

- 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.**

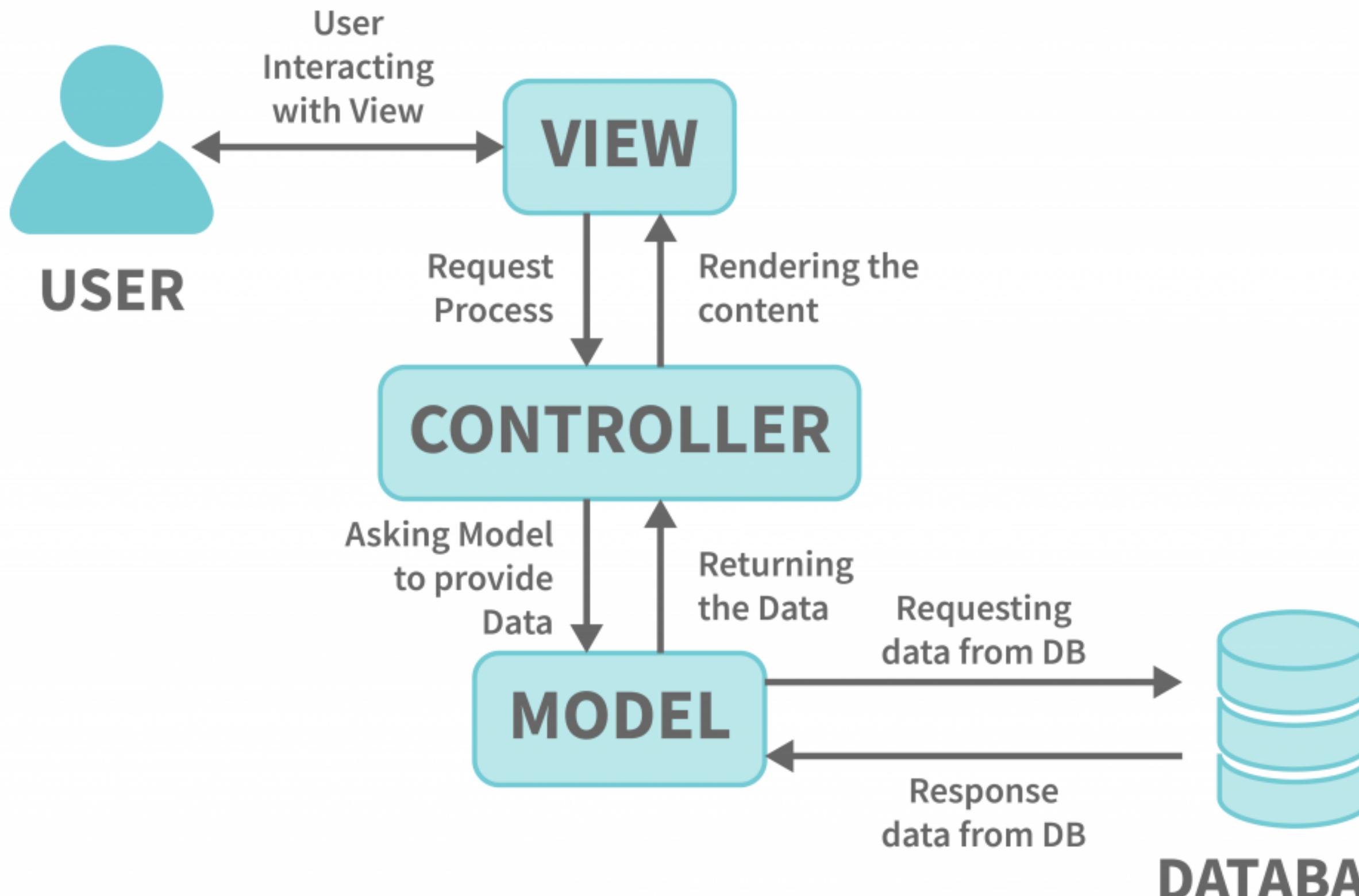


rgupta.mtech@gmail.com

Library vs Framework



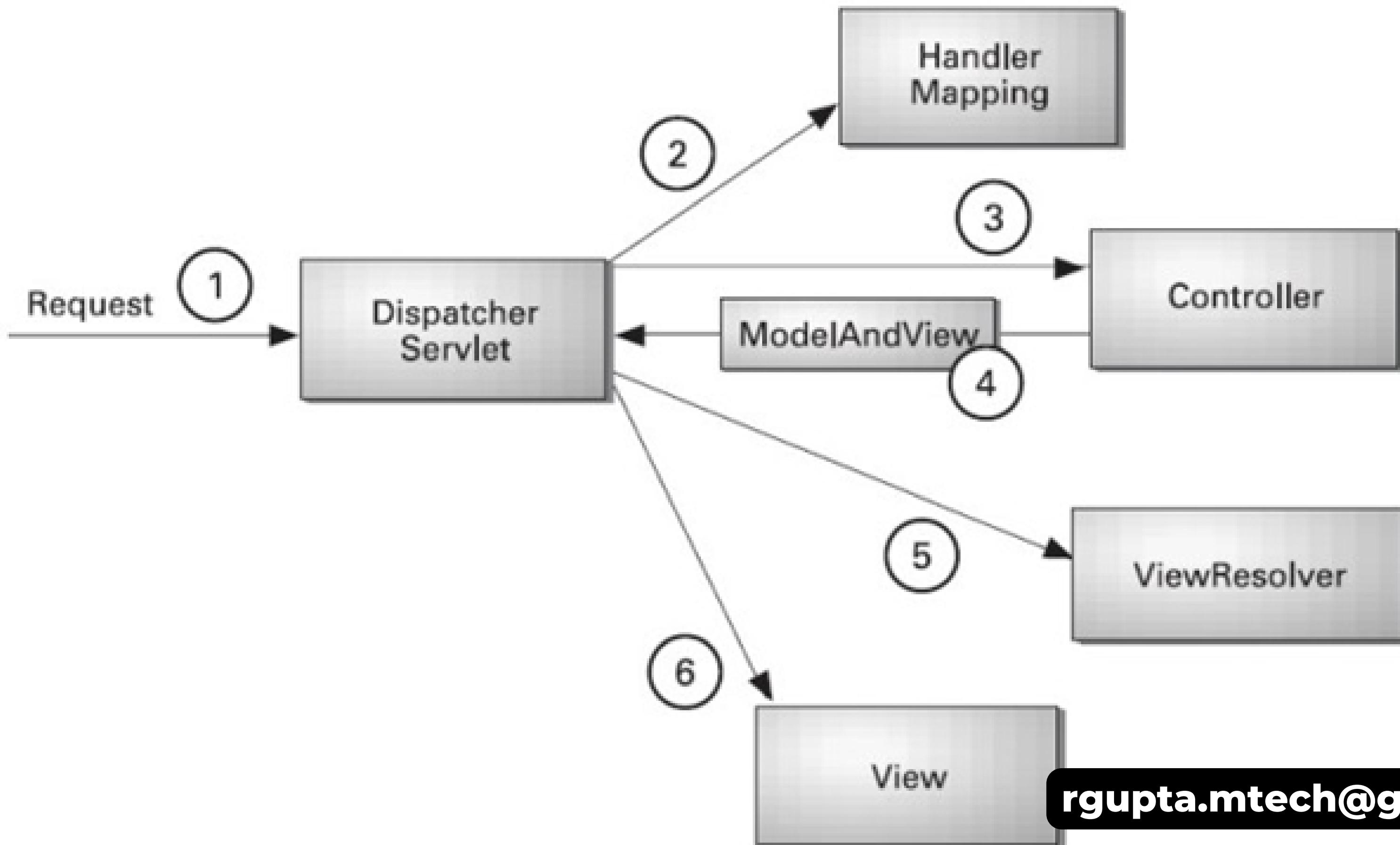
MVC Design Pattern



DATABASE

rgupta.mtech@gmail.com

MVC Design Pattern



rgupta.mtech@gmail.com

MVC Design Pattern

