Deep Dive Design pattern

Spring and Java EE

“Too busy chopping wood to sharpen the axe”

Lib & Framework

BTech/MTech

12th class

Design patterns

10th class

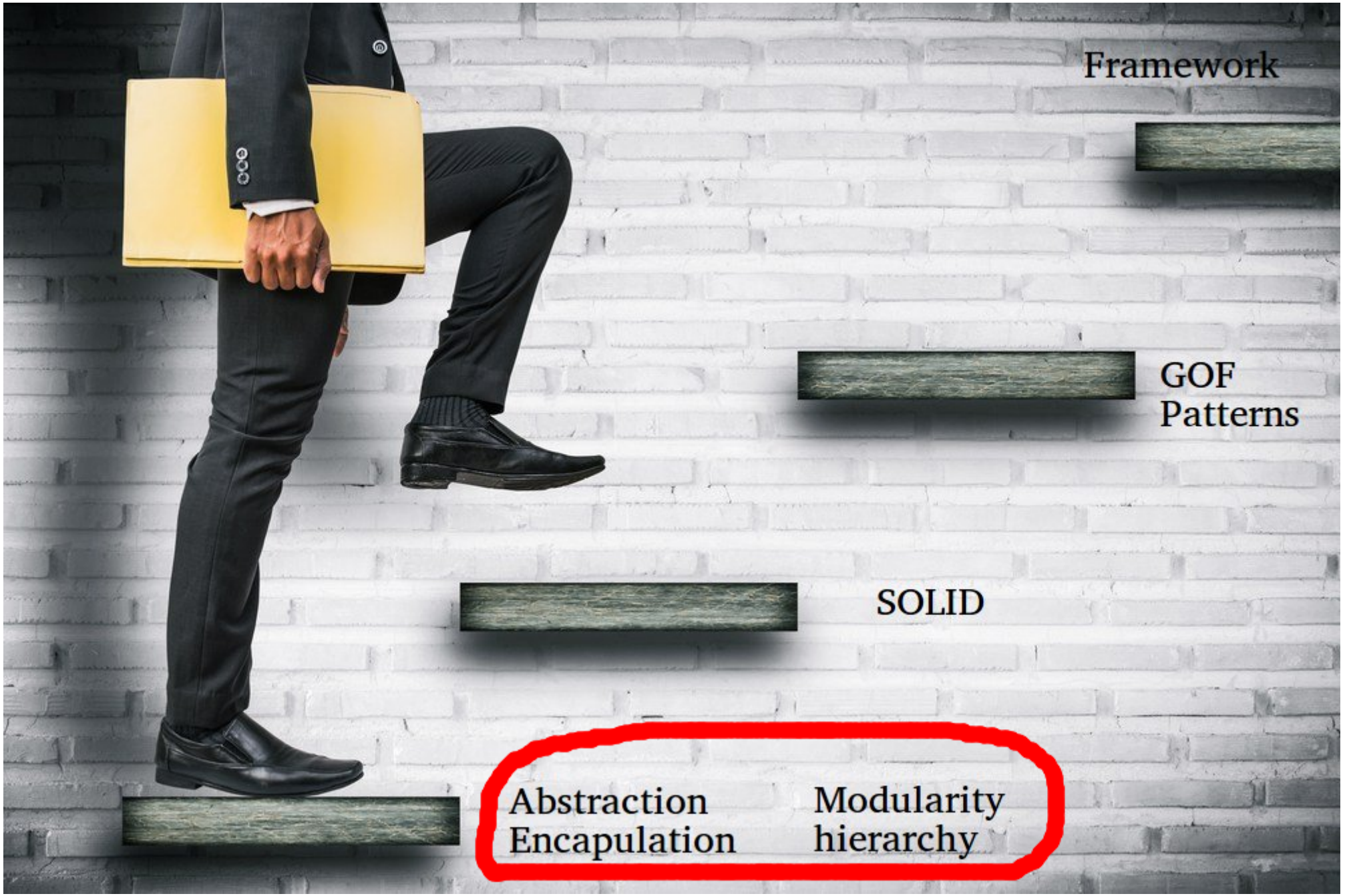
SOLID

5th class

clear understanding of OOPs concepts

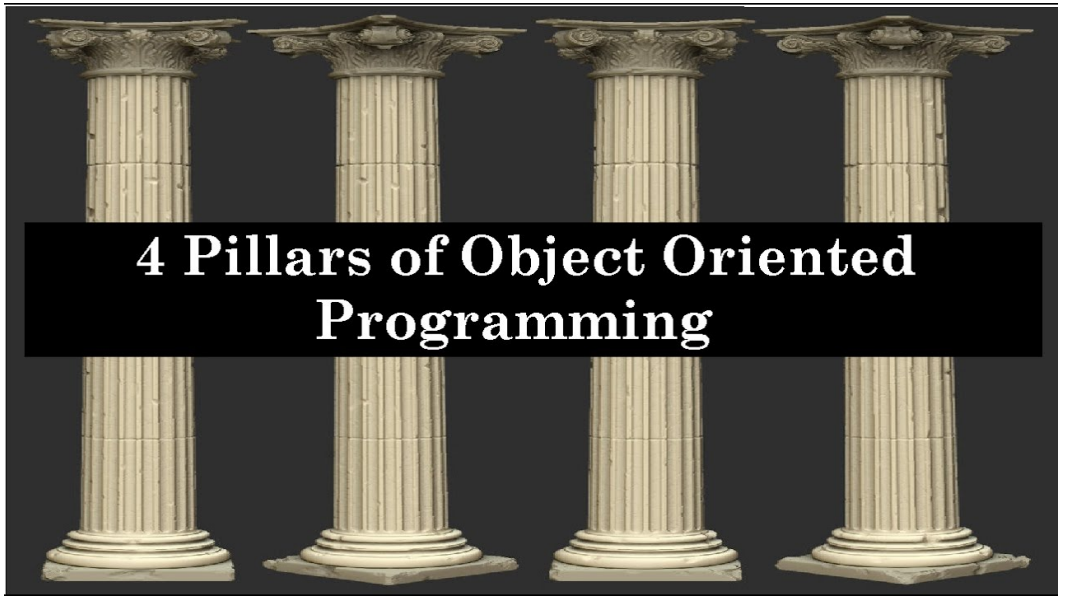
When watching experts perform it's easy to forget how much effort they've put into reaching high levels of achievement


Framework
GOF
Patterns
SOLID
Abstraction
Encapulation
Modularity
hierarchy


Framework
GOF
Patterns
SOLID
Abstraction
Encapulation
Modularity
hierarchy

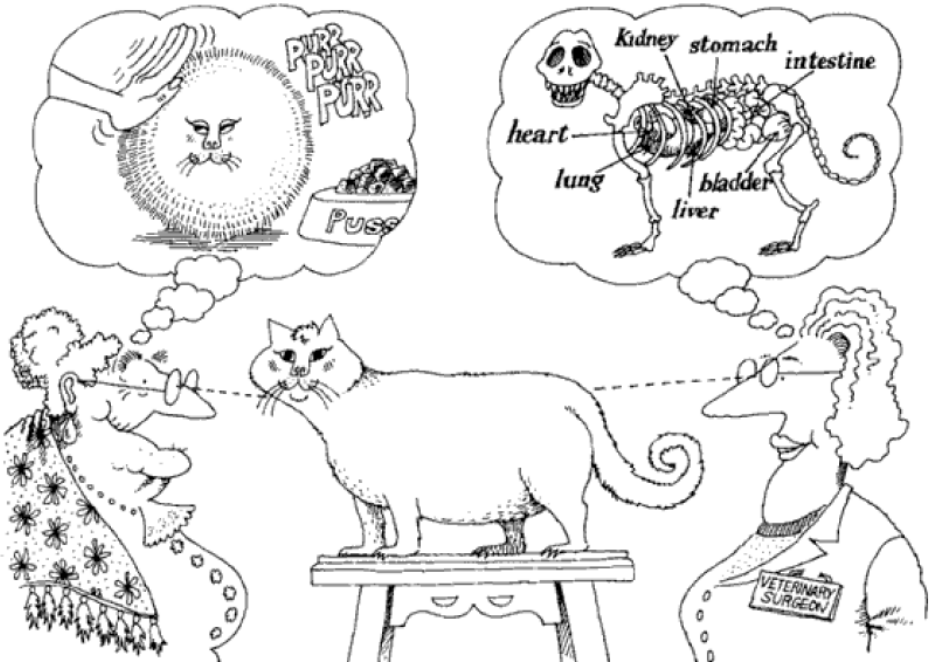
Pillors of OO

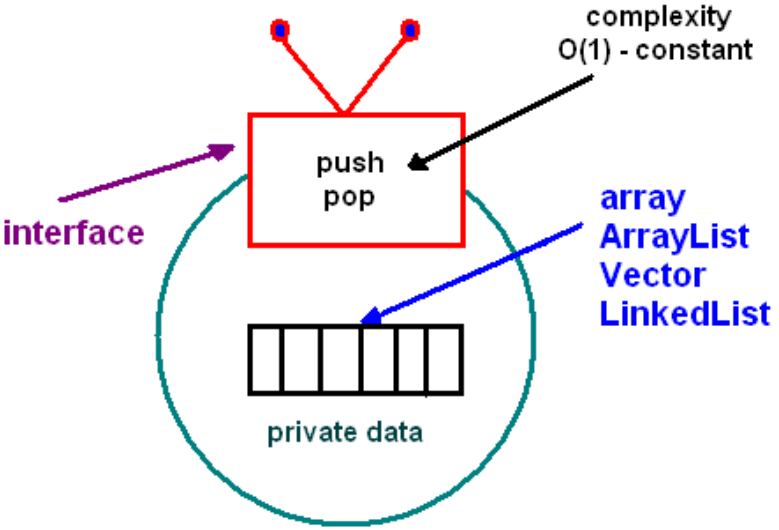
- Abstraction
- Encapsulation
- Modularity
- Hierarchy



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 !!

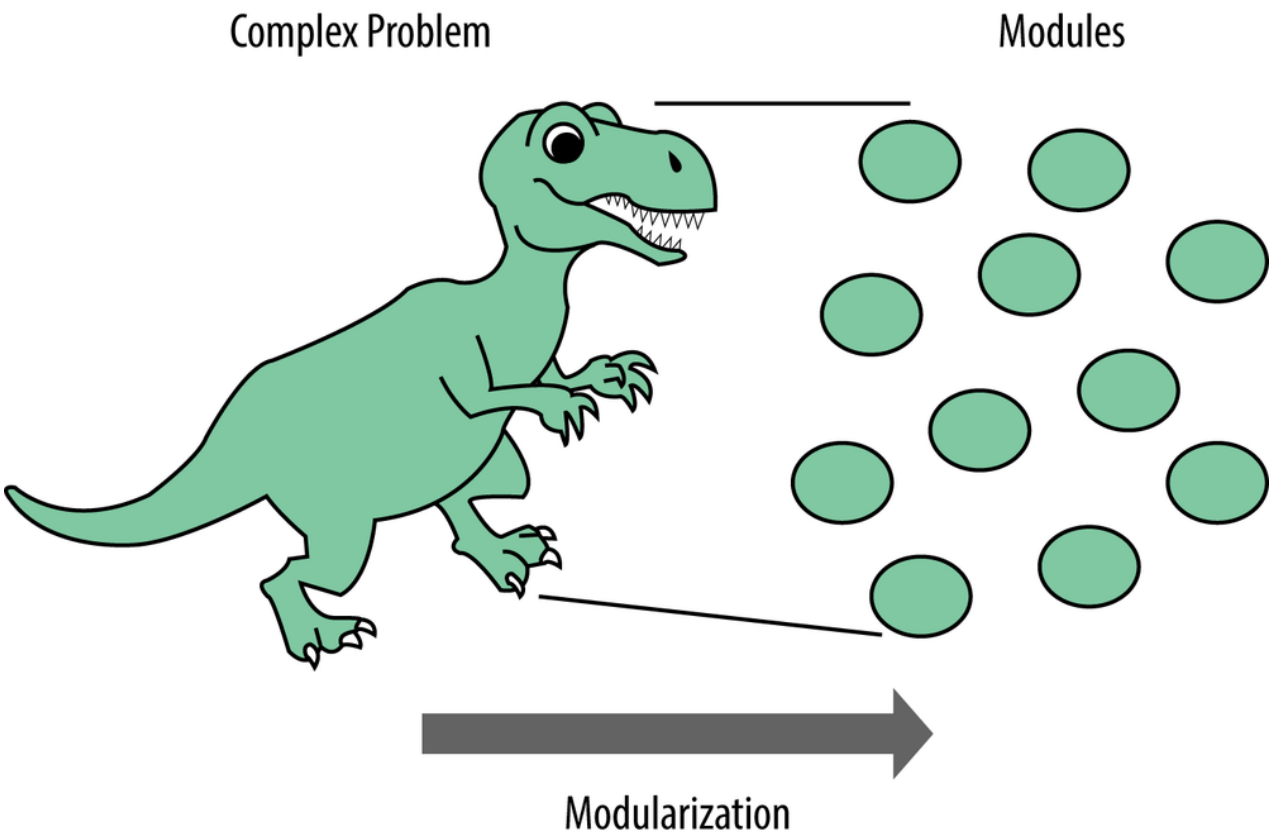
Encapsulation

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

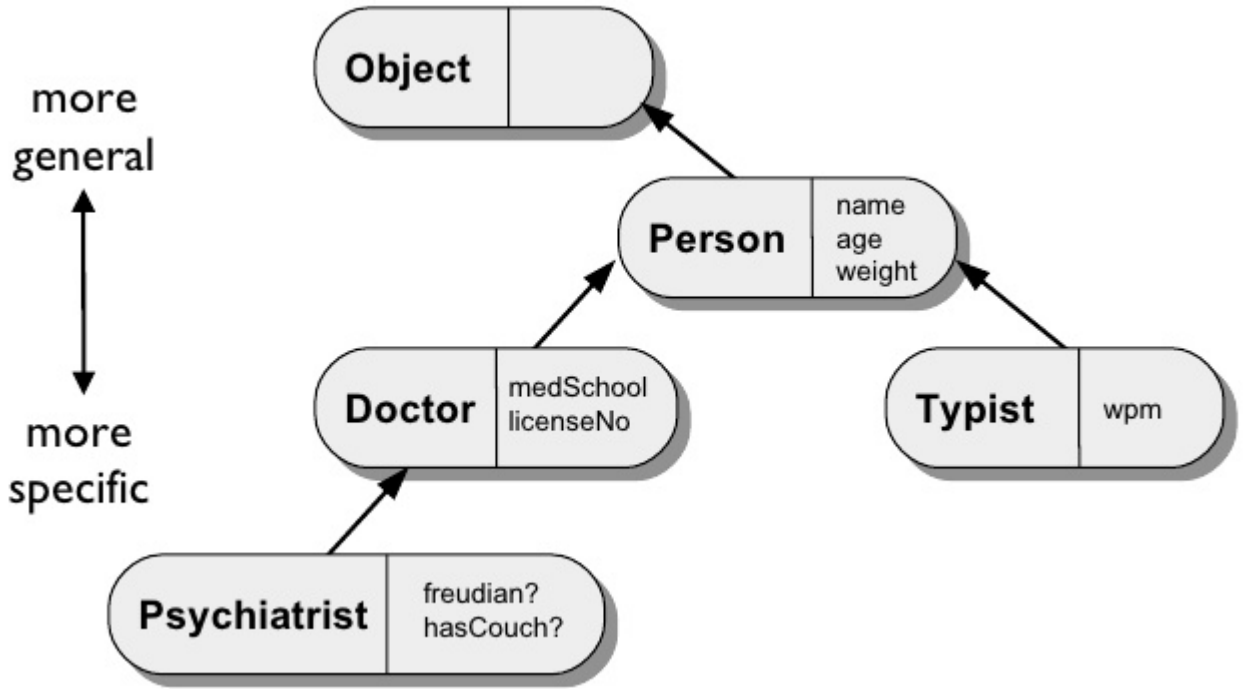
Abstraction vs Abstraction

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



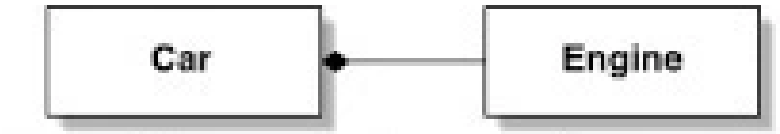
Hierarchy


more
general
more
specific
Object
Person
name
age
weight
Doctor
medSchool
licenseNo
Typist
wpm
Psychiatrist
freudian?
hasCouch?

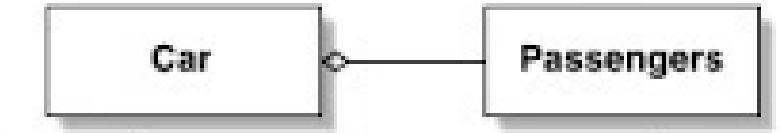
UML 101

UML symbols

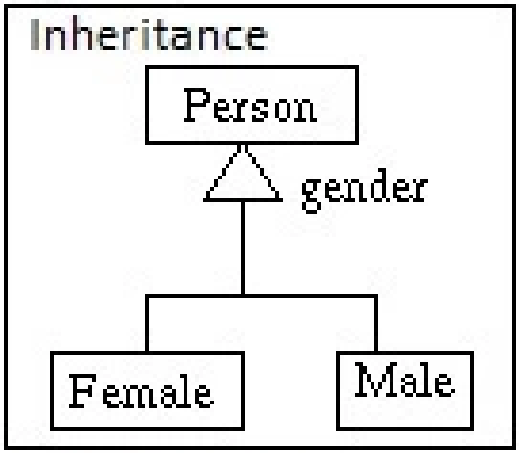
Association	Symbol
Composition	
Aggregation	
Inheritance	
Implementation	

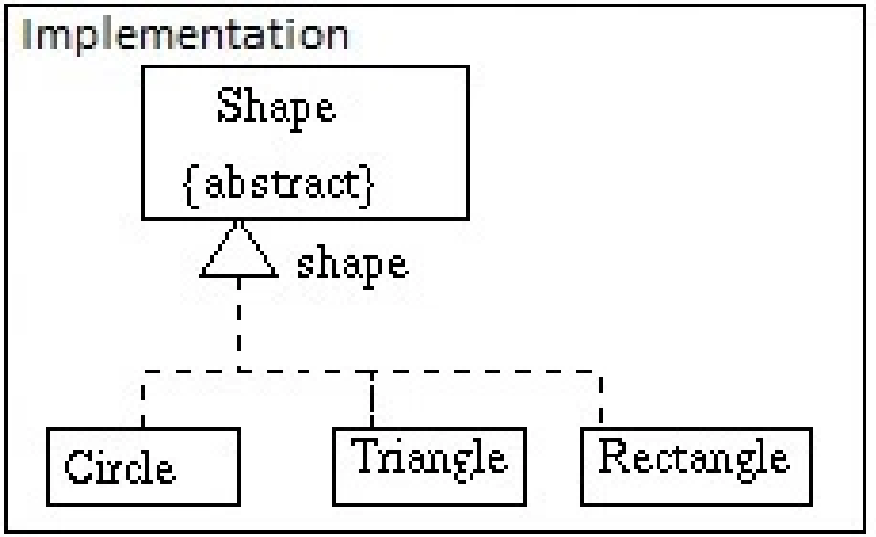


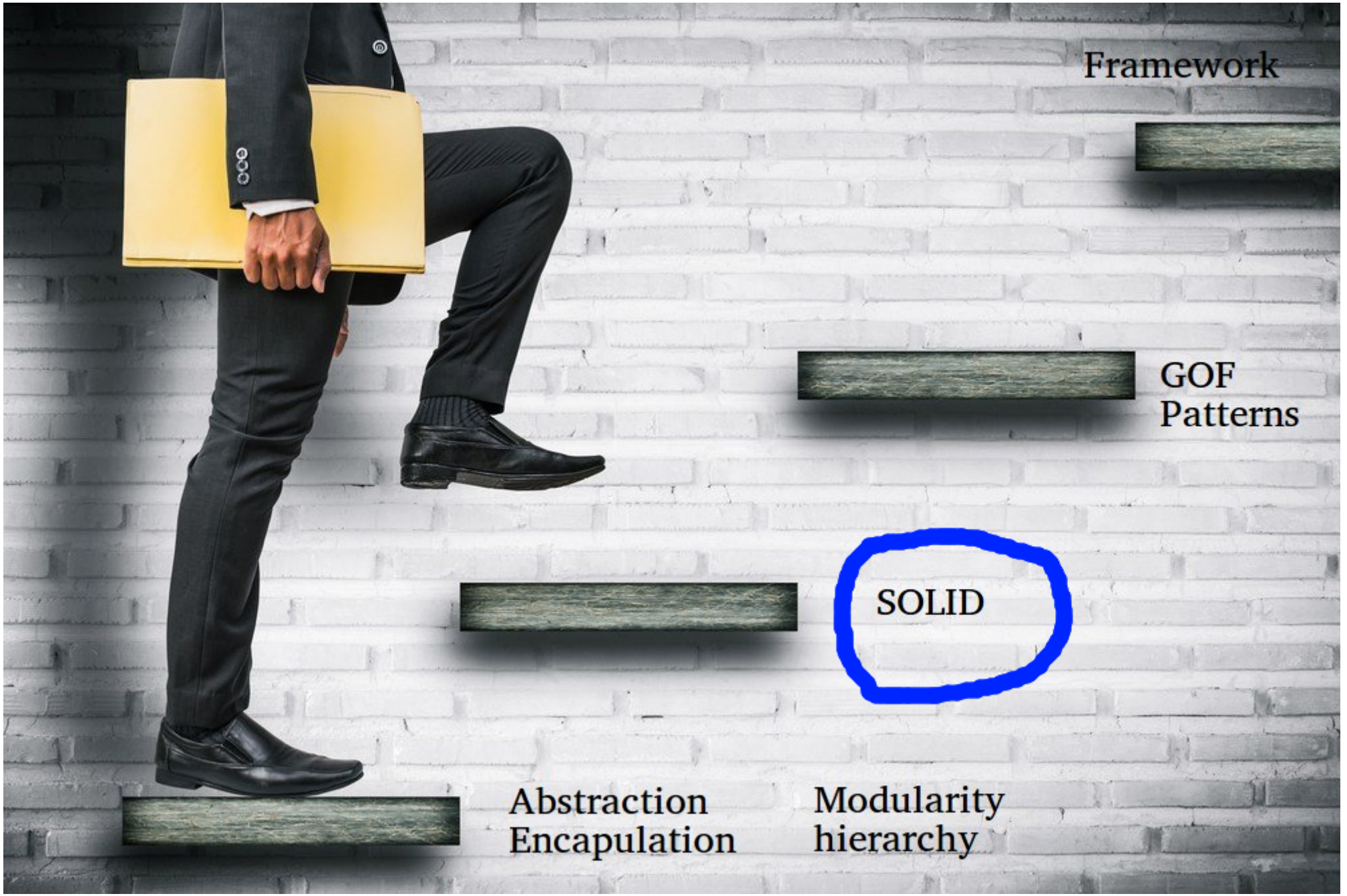
Composition: every car has an engine.



Aggregation: cars may have passengers, they come a






Framework
GOF
Patterns
SOLID
Abstraction
Encapulation
Modularity
hierarchy

We write code that is ...

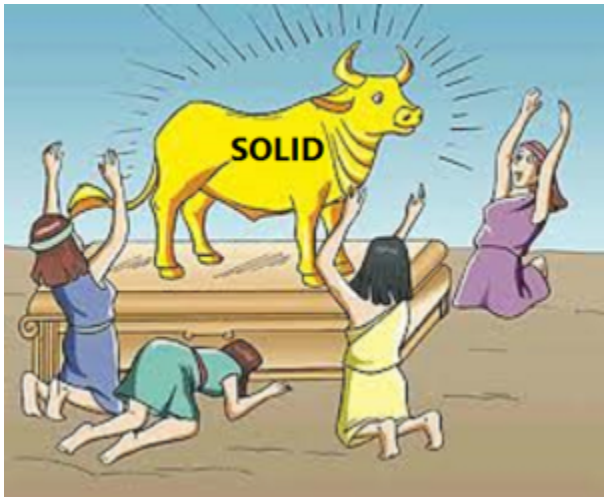
Rigid

Fragile

Immobile

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, exsiting principles brought together

SOLID

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

Single Responsibility Principle

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

Open Closed Principle

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

Liskov Substitution Principle

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

Interface Seggregation Principle

”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

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

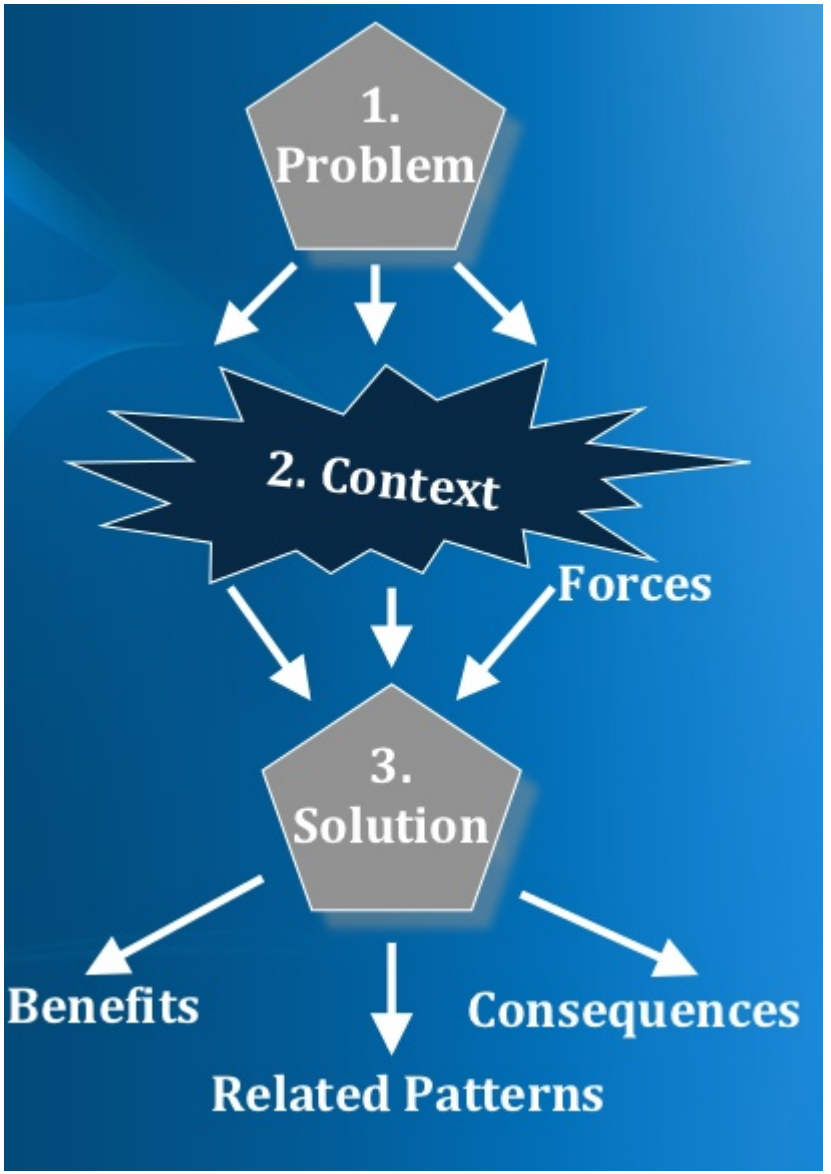
Top 10 Object Oriented Design Principles

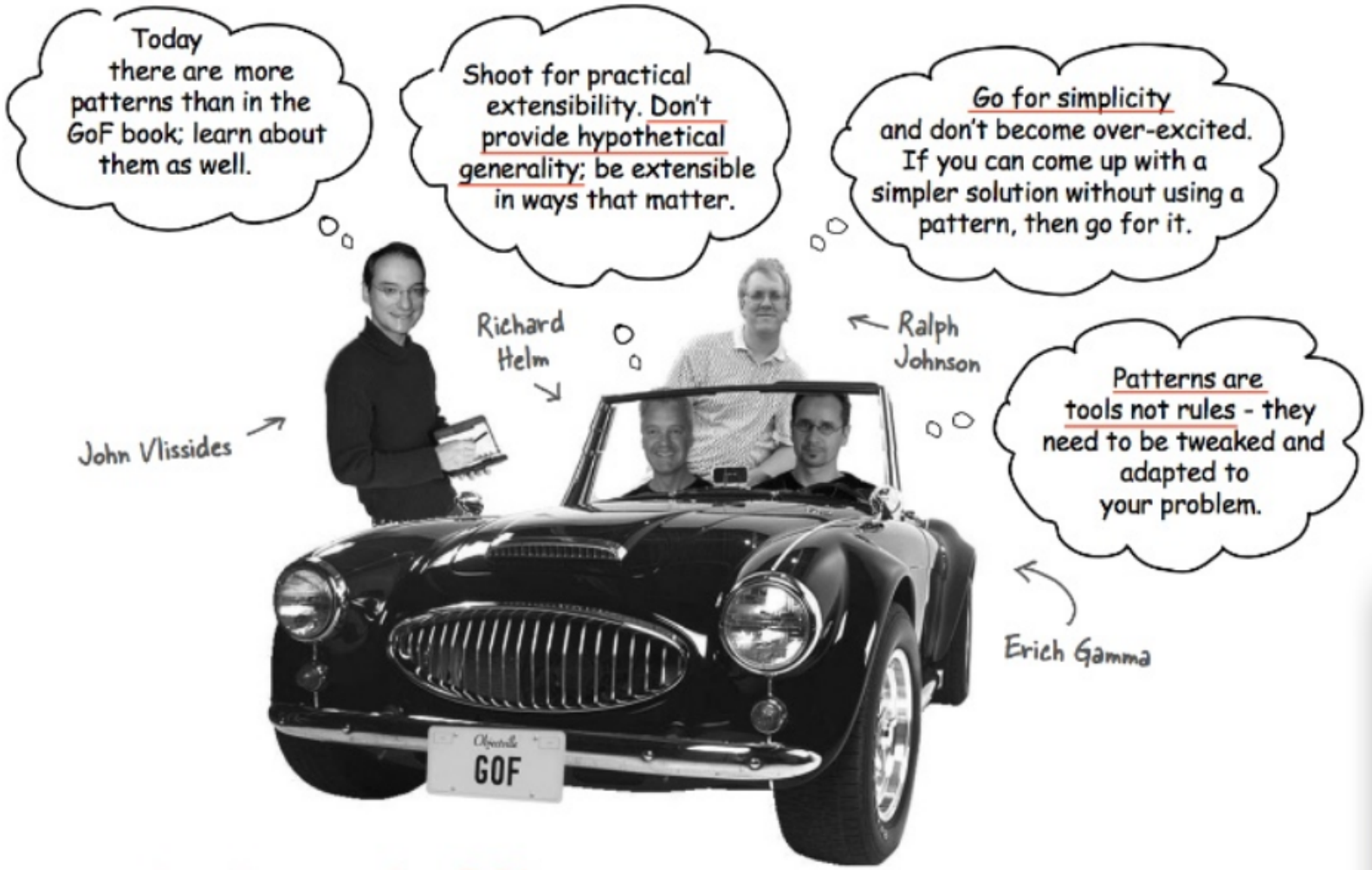
1. DRY (Don't repeat yourself) - avoids duplication in code.
2. Encapsulate what changes - hides implementation detail, helps in maintenance
3. Open Closed design principle - open for extension, closed for modification
4. SRP (Single Responsibility Principle) - one class should do one thing and do it 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
Encapulation
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
• 1. Decorator • 2. Proxy • 3. Bridge • 4. Composite • 5. Flyweight • 6. Adapter • 7. Facade	• 1. Prototype • 2. Factory Method • 3. Singleton • 4. Abstract Factory • 5. Builder	• 1. Strategy • 2. State • 3. TemplateMethod • 4. Chain of Responsibility • 5. Command • 6. Iterator • 7. Mediator • 8. Observer • 9. Visitor • 10. Interpreter • 11. Memento

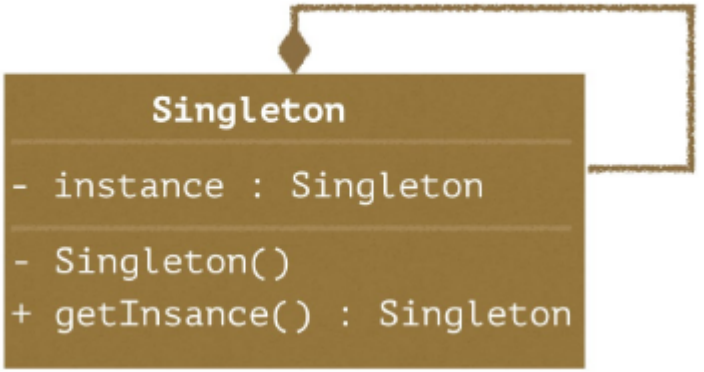
Patterns classification

- **Creational patterns?**
  - What is the **best way to create object** as per requirment
- **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."



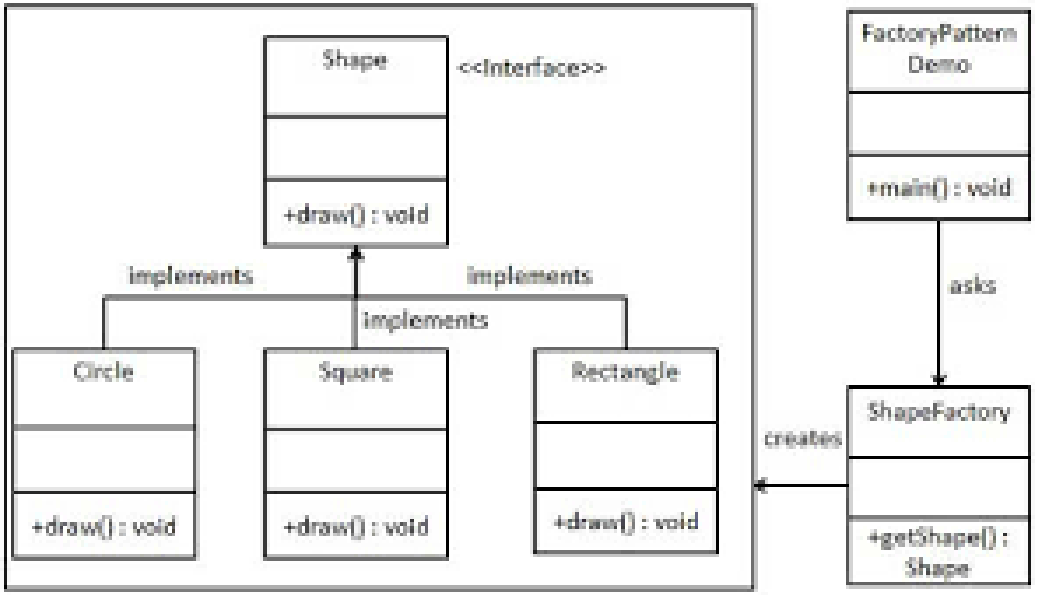
- `java.lang.Runtime#getRuntime()`
- `java.awt.Desktop#getDesktop()`
- `java.lang.System#getSecurityManager()`

❖ **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



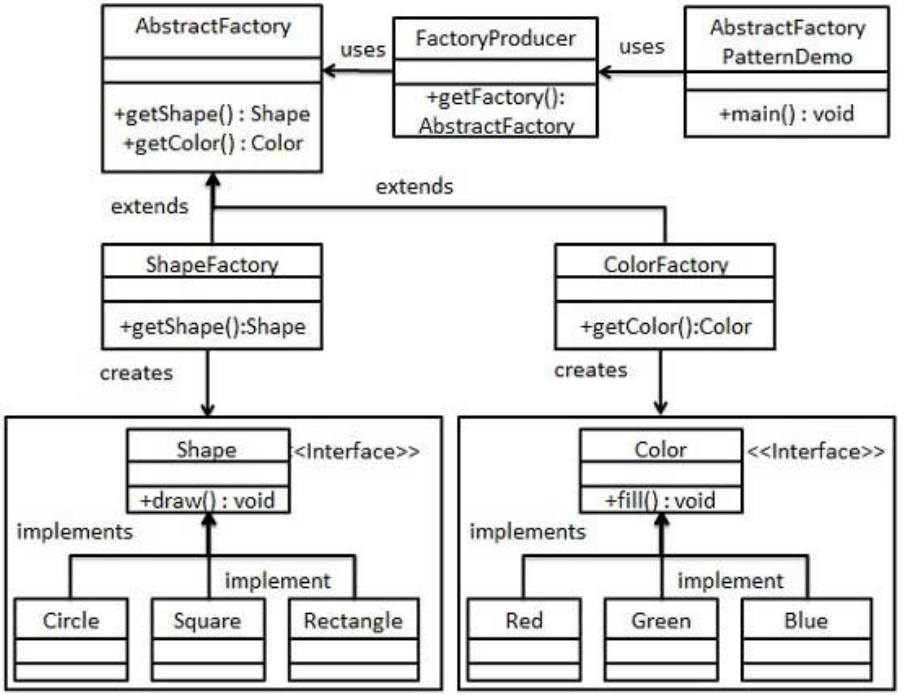
Factory(Simplified version of Factory Method) - **Creates objects without exposing the instantiation logic to the client** and **Refers to the newly created object through a common interface.**

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

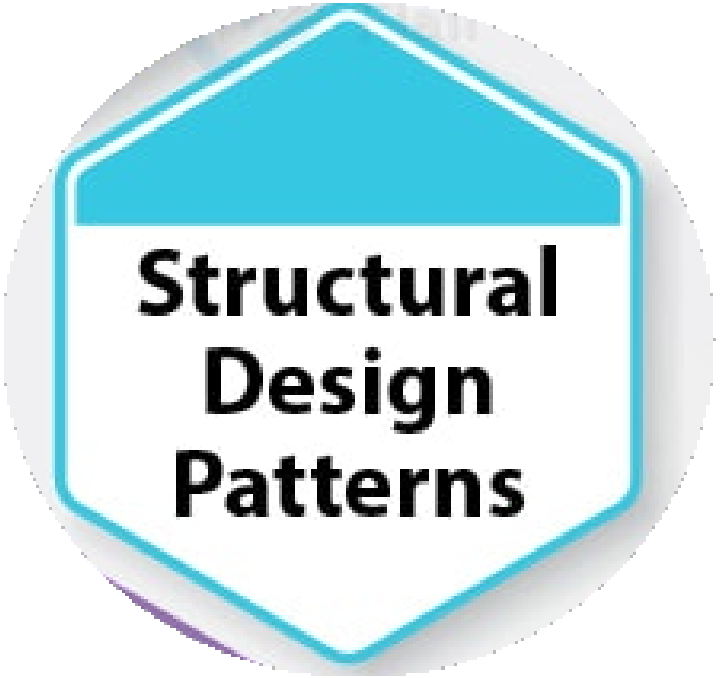
The Builder pattern can be used to ease the construction of a complex object from simple objects.

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

Prototype Pattern

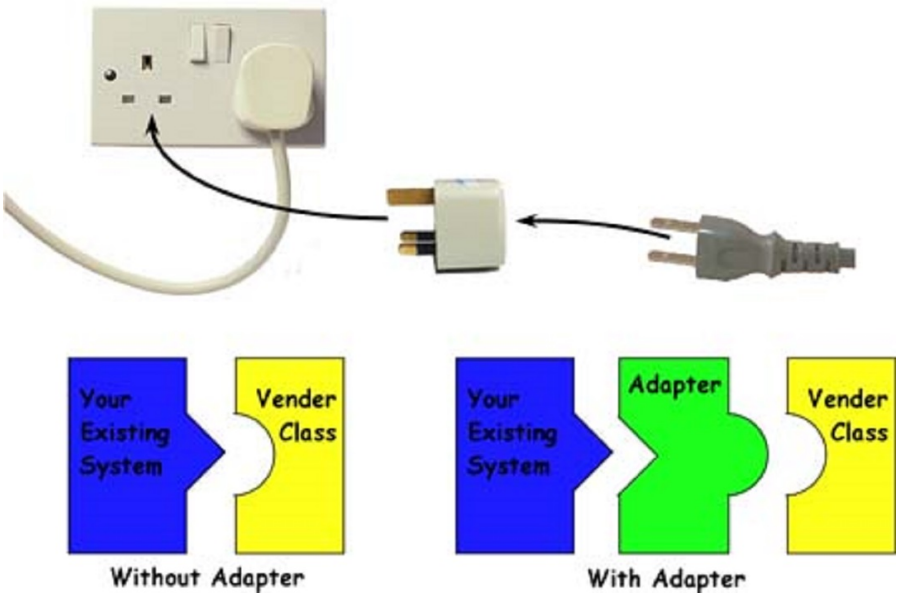
Cloning of an object to avoid creation. If the cost of creating a new object is large and creation is resource intensive, we clone the object.

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


Structural
Design
Patterns

Adapter Pattern

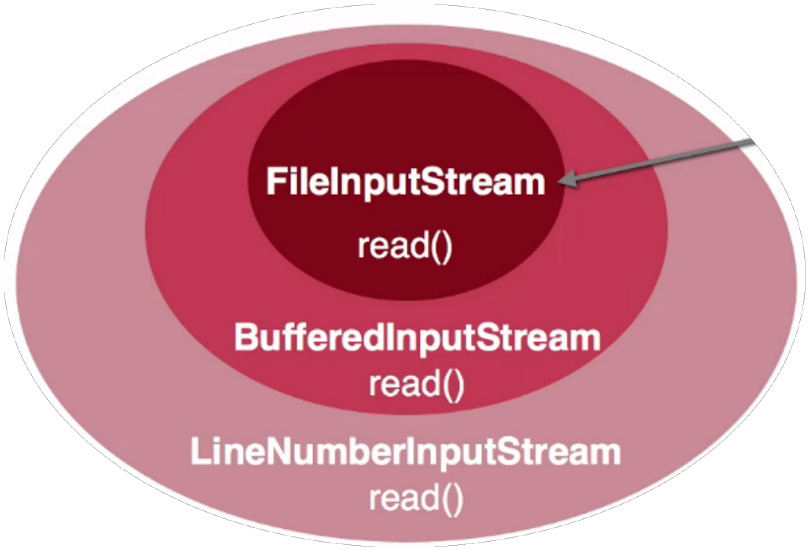
The Adapter pattern is used so that two unrelated interfaces can work together.



- `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.

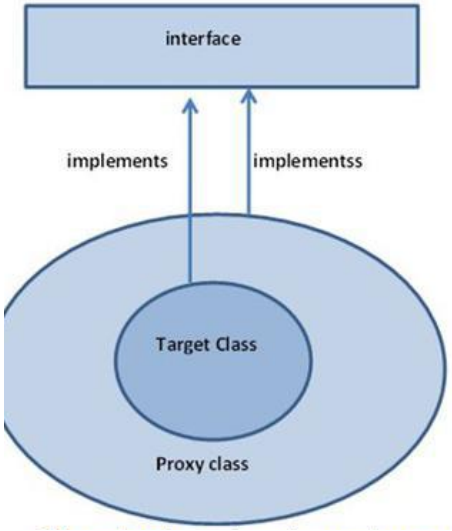
`javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

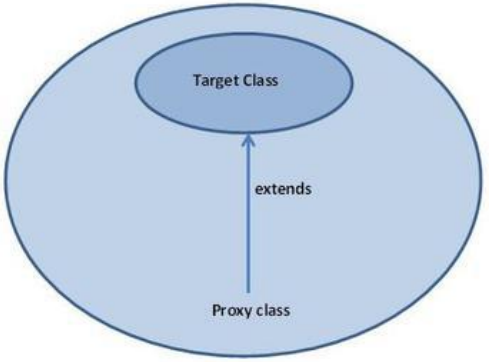
`javax.swing.JScrollPane`

Proxy design pattern

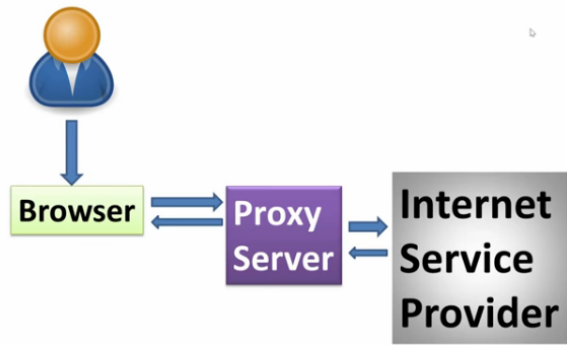
Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.

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





CGLIB proxy



`java.lang.reflect.Proxy`

`java.rmi.*`

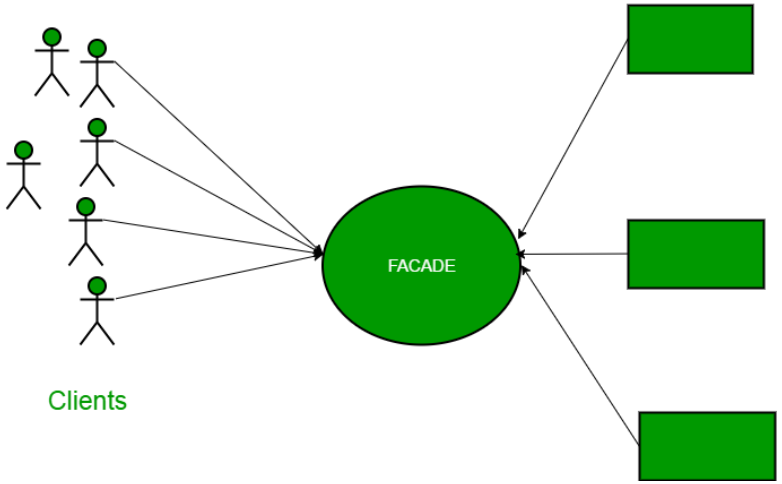
`javax.ejb.EJB` (explanation here)

`javax.inject.Inject` (explanation here)

`javax.persistence.PersistenceContext`

Facade pattern

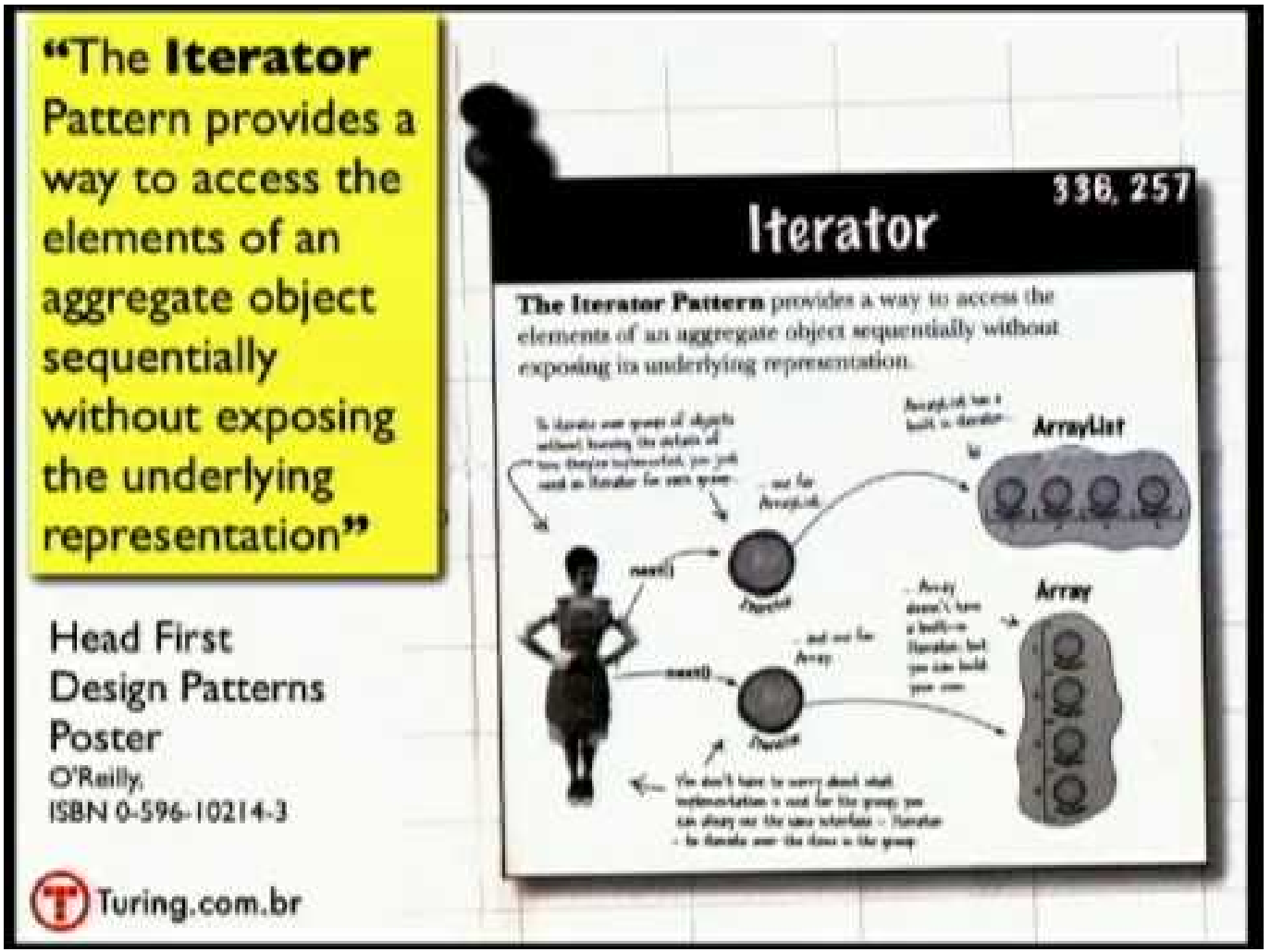
The **facade pattern** (also spelled as **façade**) is a software-**design pattern** commonly used with object-oriented programming. The name is an analogy to an architectural **façade**. A **facade** is an object that provides a simplified interface to a larger body of code, such as a class library.



- `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.

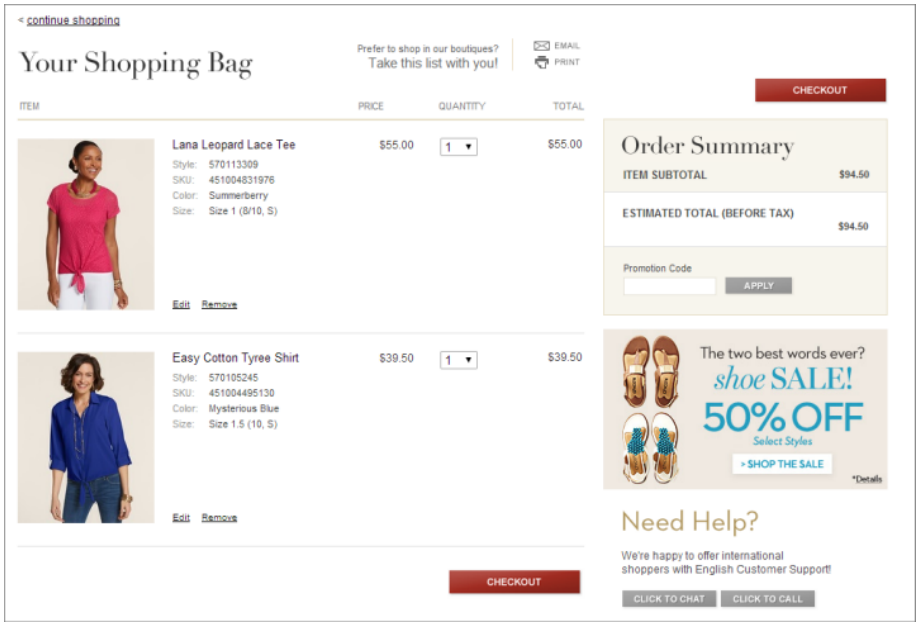

Behavioral
Design
Patterns

Iterator design pattern



Strategy pattern /policy pattern

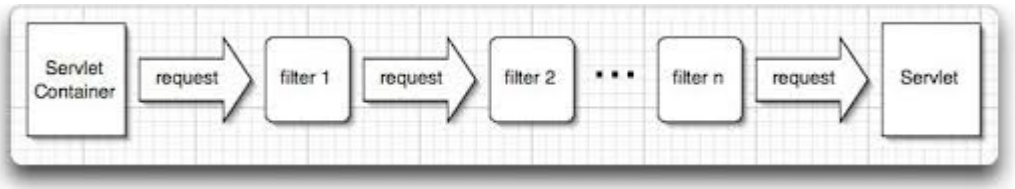
Strategy - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



- `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

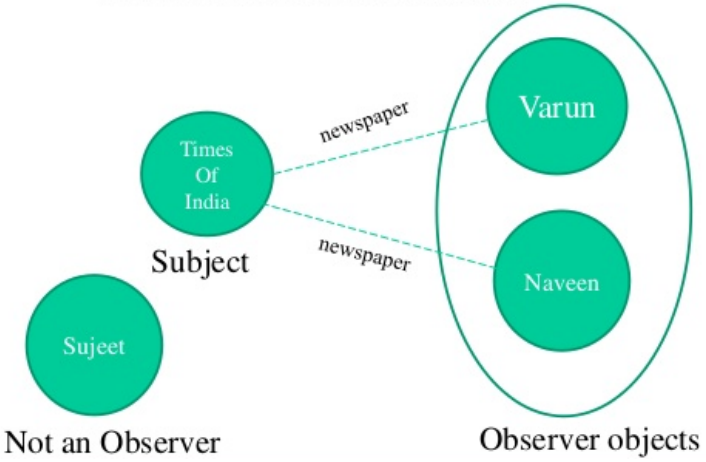
Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them.



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

Customer Feedback Form

Customer Name:

Address:

Destination:

Account:


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

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