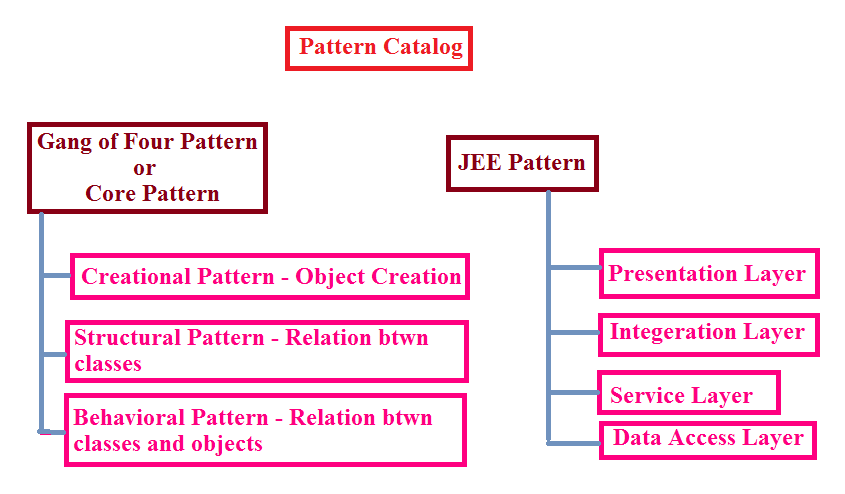
**Design Patterns**

1. While working on enterprise problems we document **best practices**, **common recurring problems** and their **proven solutions**.
2. Design pattern helps identifying recurring problem and providing ready to use solution.

For example – If we want to create only 1 object of the class no matter how many times that class is being invoked we have **Singleton pattern**, if we want to separate concerns of web application we have **MVC pattern**.

1. Advantages of Design Pattern –
2. Capturing complete design pattern experience.
3. Promote reusability
4. Define system structure better
5. Provides a common design language
6. Check if a problem statement can be solved by any of the existing design pattern, if not then create and document your own design pattern which becomes a **candidate design pattern** if same problem occurs multiple times and your design pattern can be used to solve it then it becomes Design Pattern.
7. Design Pattern Category –



GOF Patterns –

1. Creational Patterns –

* Singleton
* Factory
* Abstract Factory
* Builder
* Prototype

1. Structural Patterns –

* Adapter
* Bridge
* Flyweight
* Decorator
* Proxy
* Facade

1. Behavioral Patterns –

* Command
* Interpreter
* Template method
* Observer
* Visitor
* Mediator
* Memento
* Iterator

JEE Patterns –

1. Presentation Layer –

* Intercepting Filter
* Front Controller
* MVC
* Context Object

1. Business Layer –

* Business Delegate
* Transfer Object
* Session Façade
* Service Locator

1. Data Access Layer –

* DAO

1. Integration Layer –

* Service Activator
* Web Service Broker

1. **Singleton Pattern** –

A class must ensure that only **single instance** should be created and same object should be used by all other classes.

Example –

1. Single instance of Datasource class for all DB connectivity across application
2. Single instance of configuration manager or error logger class

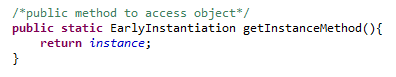
How to create Singleton class –

1. Create a **private constructor**
2. Create a **static instance** of return type same as that of class.
3. Create a public method and return this static instance.

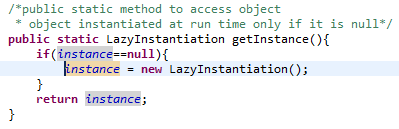
Advantage – It **saves memory** as same instance is used again and again.

Types –

1. Early(Eager) Instantiation – Creating instance at the load time



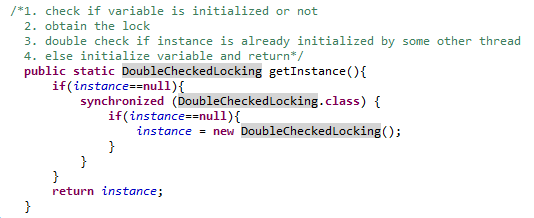
1. Lazy Instantiation – Creating instance when required – Inside static method do a object null check if null then only create instance.



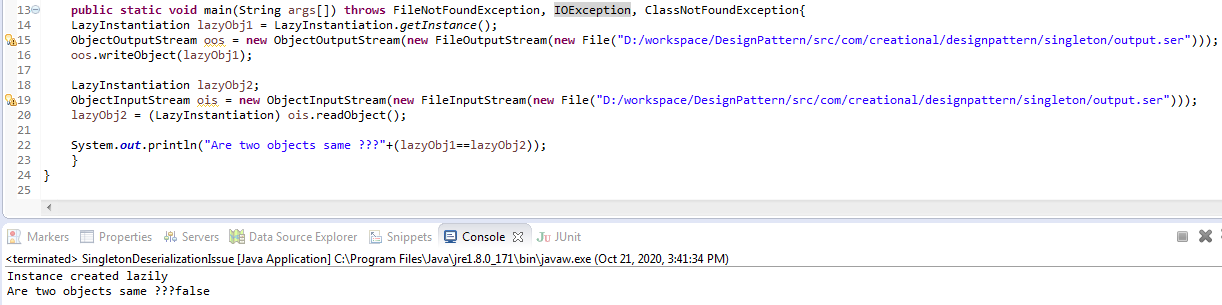
**Lazy Instantiation** is preferred over Eager Instantiation.

**Handling multi threading scenario** – If more than 1 thread try to access static method to get instance it is possible that 2 threads may find instance==null, thereby creating 2 instances of Singleton class. This issue can be resolved via **Double Checked Locking**.

Ref – <https://en.wikipedia.org/wiki/Double-checked_locking#Usage_in_Java>



**Handling Serializable scenario** – When we save singleton object instance to a file and try to read back from the same file, object instance returned is not same.



This problem can be resolved by overriding **readResolve()** method and return instance.



To avoid cloning of the instance –

1. Implement Cloneable interface in Singleton class
2. Override clone() method and throw new CloneNotSupportedException
3. **Factory Pattern** –

* We create object **without exposing the creation logic** to the client.
* Also known as **Virtual Constructor**.
* Example – DriverManager.getConnection(String conString)
* **Problem statement** – Let’s take an example we have travel type of air and train and client handles the logic of object creation for these two types. Tomorrow if a new travel type bus is introduced then client will need to handle implementation for it, which is a bad design practice.
* **Advantages** –

1. Promotes loose coupling.
2. Client need not to bother about handling object creational logic.
3. **Abstract Factory Pattern** –

* Also known as **Kit** or **Factory of Factory** pattern.
* It is one level higher than Factory pattern as it lets a class return a factory of classes.
* Abstract Factory provides interfaces for **creating families of related or dependent objects** without specifying their concrete classes.
* Example – JAXP – Java API for XML Parsing

DocumentBuilderFactory – DocumentBuilder – Document

* **Problem statement** –

We want to build a global car factory, suitable for multiple locations with critical design changes. Application should be smart enough to identify location where it is used so that use appropriate factory without even knowing which car implementation will be used internally. This needs another layer of abstraction to identify correct location and correct car factory.

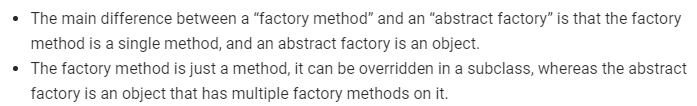
* **Advantages** –

1. Isolates client code with implementation class
2. Eases of exchanging of object families
3. Promotes consistency among objects

* **Disadvantages** –

1. Difficult to support new kind of products

Difference between Factory Pattern and Abstract Factory Pattern –



1. **Builder Pattern** –

* Builds a complex object using **simple objects** and using a **step by step approach**.
* Mostly used when object cannot be created in single step.
* Example – Preparing a meal in McD, constructing a house
* **Problem statement** –

Consider we have to create a cake, need number of items sugar, flour, milk, fruits, and nuts (optional). Creating big parameterized constructor is problematic.

* **Advantages** –

1. Provides separation between construction and representation of object.
2. Better control over construction of object.
3. Support changing internal representation of objects.

* **Disadvantages** –

1. Too many lines of code
2. Code duplication
3. **Prototype Pattern** –

* Creating **duplicate object** while keeping performance in mind.
* **Cloning** an existing object and customizing as per requirement.
* Hash codes will be different of cloned object.

**Problem statement** –

1. This pattern is used when creating object is **expensive** and **resource intensive**.

* **Advantages** –

1. Hides complexity of creating object.
2. Can add/remove objects at runtime
3. Reduces number of sub classing

* **Disadvantages** –

1. Not useful when project/application creates minimum objects
2. **Object Pool Pattern** –

* To reuse the object that is expensive to create.
* Objects in the pool have a lifecycle – **creation**, **validation** and **destroy**
* **Advantages** –

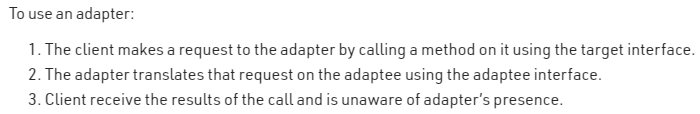
1. Boosts performance of application significantly.
2. Most effective when rate of initializing the class instance is high.
3. Can limit for maximum number of objects that can be created.

**Structural Pattern** **–**

* Identifies how classes and objects can be composed to form large structures.
* Simplifies the structure by identifying the relationships.
* Focus on **how classes inherit from each other** and how they are composed from other classes.

1. **Adapter Pattern –**

* Also called as **Wrapper**.
* Converts the interface of a class into another interface that a client wants.
* Works as a **bridge between two incompatible interfaces**.
* In design, adapters are used when we have a class (**Client**) expecting some type of object and we have an object (**Adaptee**) offering the same features but exposing a different interface.



* Example – Laptop charger adaptor used in various countries across globe.
* **Advantages** –

1. Allow two or more incompatible classes to interact.
2. Allows reusability of existing functionality.

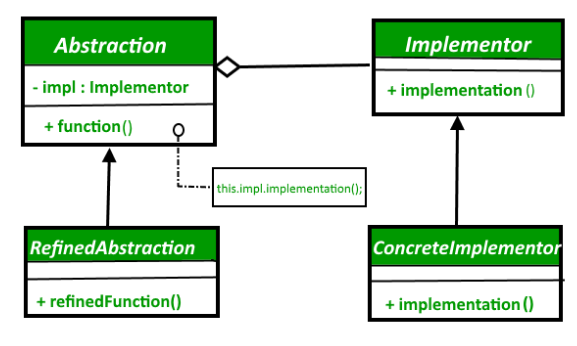
* **Disadvantages** –

1. All requests are forwarded, so there is a slight increase in the overhead.

* **When to use** –

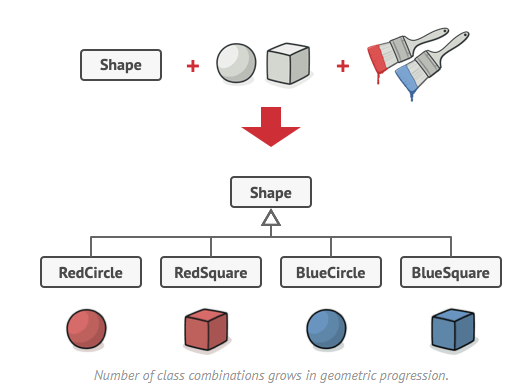
1. When an object needs to utilize existing class with an incompatible interface.
2. **Bridge Pattern –**

* ***Decouple*** an ***abstraction*** from its ***implementation*** so that two can vary ***independently***.
* It follows notion to prefer Composition over Inheritance.
* Another name is **Handle (Interface class)/Body (Implementation class)**.
* It is designed up-front.
* **UML Diagram –**

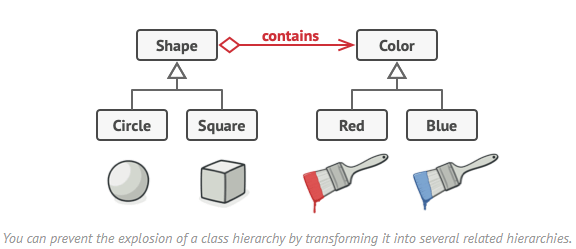


**Problem statement** –

1. In below example if we add a new color or shape or both it will result in increase in number of subclasses exponentially.

****

**Solution –**

****

* **Advantages** –

1. It improves extensibility.
2. It allows the hiding of implementation details from client.
3. We can create platform independent classes and apps.

* **Disadvantages** –

1. You might make the code more complicated by applying the pattern to a highly cohesive class.

* **When to use** –

1. When we have interface hierarchies in both interfaces as well as implementations.
2. You need to map orthogonal class hierarchies. Example – Bus, ProduceBus and AssembleBus if change in Bus then also change in other 2 class, this can be solved using bridge pattern.
3. To switch implementation at runtime.
4. When we want to divide and organize monolithic class that has several variants of some functionality.
5. Platform independent feature.
6. **Flyweight Pattern –**

* To reuse already existing similar kind of objects by storing them and create new object when no matching object is found.
* Flyweight object is divided into 2 pieces – (a) state dependent (extrinsic) part passed when operation is invoked and is stored by client objects. (b )state independent (intrinsic) part shared and stored by Flyweight object.
* Example – Pen is FW object, refill is extrinsic part and pen body is intrinsic part.

Browser images loaded

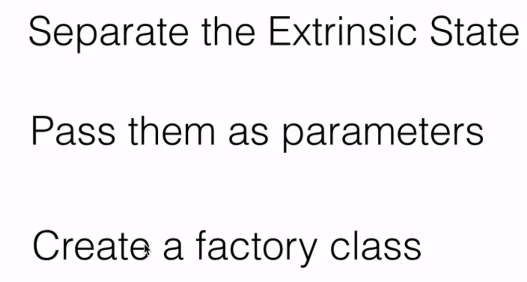
java.lang.String constants stored in string pool.

* **Problem statement** –

Creating multiple objects in an application can be expensive in terms of performance and memory usage.

* **Solution** –

We use **static** hashmap to create and store object in factory class.



* **Advantages** –

1. Reduces number of objects.
2. Reduces amount of memory consumed.

* **When to use** –

1. When we need a large number of similar objects, FW is used to reduce number of objects created and to decrease memory footprint and increase performance.
2. When an application doesn’t depend on identity.
3. When the storage cost is high and we need to control memory consumption by large number of objects.
4. **Decorator Pattern –**

* Allows adding new functionality to an existing object without altering its structure dynamically.
* Decorators have the same super type as the object they decorate.
* We can use multiple decorators to wrap an object.
* Decorator **HAS-A** relationship with component it decorates.
* Decorator pattern is also known as **Wrapper**.
* Decorating an object changes its behavior but not its interface.
* **Examples** –

1. ColorDecorator for Shape
2. PizzaDecorator

* **Advantages** –

1. Solves Open Closed Principle design.
2. Alternative to subclassing, subclassing adds behavior at compile time and changes affect on all instances of original class.
3. Decorator pattern makes it possible to extend functionality of object at runtime.
4. Offers pay as you go approach.

* **Disadvantages** –

1. Hard to remove specific wrapper from wrapper stack.
2. Hard to implement a decorator in such a way that behavior doesn’t depend on order in decorator stack.
3. Can complicate the process of instantiating object.
4. Overuse can be complex.

* **When to use** –

1. To assign extra behavior to object at runtime.
2. When it is not possible to extend behavior using inheritance
3. **Proxy Pattern –**

* Provide a **surrogate** or **placeholder** for another object to control access to it.
* Example – In hibernate, lazy loading scenarios of DAO operations.

Corporate network internet access is guided by network proxy.

AOP framework object creation.

* Different types of proxies –

1. Remote – Remotely located object
2. Virtual – Delay creation of objects until needed, example Hibernate
3. Protection – Provides security over original object.
4. Smart – Additional housekeeping when object is accessed by client.

* **Advantages** –

1. Pattern hides actual object from outside world.
2. Can improve performance as object is created on demand.

* **Disadvantages** –

1. Response from service might get delayed.

* **When to use** –

1. When we want a simplified version of complex or heavy object.
2. When original object is present at different address and we want to represent it locally.
3. When we want to add a layer of security to original object.
4. **Façade Pattern –**

* It hides the complexities of the system and provides an interface to the client from where client can access the system.
* Example – Computer startup it involves work of cpu, memory, hard drive, in java – JDBC Connection
* Every **abstract factory** is a type of façade.
* **Advantages** –

1. It shields clients from complexities of sub system components.
2. Promotes loose coupling between subsystems and its clients.

* **When to use** –

1. When we have a complex system that we want to expose to client in a simplified way.
2. When we want to structure a subsystem into layers.

**Behavioral Patterns –**

1. Concerned with interaction and responsibility of objects.
2. Interaction between objects should be in such a way that they can easily talk to each other and still should be loosely coupled.
3. **Command Pattern –**
4. Also known as **Action** or **Transaction**.
5. Also known as **Producer-Consumer** pattern.
6. “Encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command”
7. Decouple objects that produce commands from their consumers.
8. Example –

Stock – Request

Order – Command

BuyStock/SellStock – Concrete command classes implementing Order interface

Broker – Invoker

* **When to use –**

1. When we need parameterized objects according to an action perform.
2. When we want to queue operations, schedule their executions or execute them remotely.
3. When we want to support rollback, logging or transaction functionality.

* **Advantages –**

1. Single responsibility principle
2. Open/Closed Principle – Can introduce new command without breaking existing client code
3. Can implement redo/undo
4. Can implement deferred execution of operations.

* **Disadvantages** –

1. Code gets huge and confusing with high number of action methods and because of so many associations.
2. **Template Design Pattern –**

* Defines skeleton of an algorithm in an operation and defers some of the steps to client’s subclasses.
* RealTime Example – Constructing house, order of steps to construct an house can be defined in the method of abstract class.
* Method declared in base class should be **final** so that no one can change implementation.
* It is based on **Inheritance**.
* Factory method is specialization of Template method
* **When to use –**

1. When we want to let client extend only particular steps of algorithm but not whole algorithm or structure.
2. When several classes have identical code, common behavior can be moved to abstract class to avoid duplication.

* **Advantages** –

1. Common technique for reusing code.
2. Client only change certain parts of algorithm making him less affected with other changes in algorithm.

* **Disadvantages** –

1. Hard to maintain when we have more steps.
2. **Interpreter Pattern –**

* Defines a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language.
* Pattern is used in **SQL Parsing**, symbol processing engine etc.
* Code example – TerminalExpression, AndExpression, OrExpression, Expression interface

<https://www.tutorialspoint.com/design_pattern/interpreter_pattern.htm>

* **Advantages** –

1. It is easier to change and extend the grammar.
2. Implementing the grammar is straight forward.

* **When to use** –

1. When grammar of the language is not complicated
2. When efficiency is not a priority

* **Disadvantages** –

1. Complex grammars hard to maintain.
2. **Observer Pattern –**

* **One to Many** dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* **Observers** register themselves to a subject to get a notification when there is a change made inside that **Subject(Publisher)**.
* It is also called **Publish – Subscribe** pattern.
* Real world example – Facebook notifications
* Examples from Java- EventListener,HttpSessionBindingListener, HttpSessionAttributeListener
* **Advantages** –

1. Supports broadcast type communications.
2. Follows Open/Closed Principle
3. Can establish relations between objects at runtime

* **When to use** –

1. When changes to state of one object may require changing other objects, and actual set of objects is unknown beforehand or changes dynamically.
2. When some objects in app must observe others but only for limited time.

* **Disadvantages** –

1. Notified at random orders.
2. **Visitor Pattern –**

* Allows for one or more operation to be applied to a set of objects at runtime, decoupling the operations from the object structure.

1. **Mediator Pattern –**

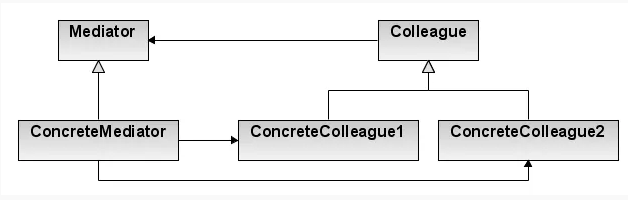
* Define an object that encapsulates how a set of objects interact.
* Provides **mediator class** which normally handles all the communications between different classes and supports easy maintainability of the code by **loose coupling**.
* Reduces communication complexity between multiple objects or classes.
* Real World Example – Traffic control rooms at airport, Chat application

Java Example – execute() method in Executor interface, schedule() method in Timer class.

* **Problem Statement** –

Think of a scenario where we start development with few classes which interacts with each other, as slowly logic increases and new classes are introduced, interaction between classes become complex to track.

* UML Diagram –



* **Advantages** –

1. Decouples number of classes.
2. Simplifies object protocols.
3. Centralizes the control.
4. Single Responsibility Principle.
5. Open/Closed Principle.
6. Reduce coupling between components.
7. Reuse components easily.

* **When to use** –

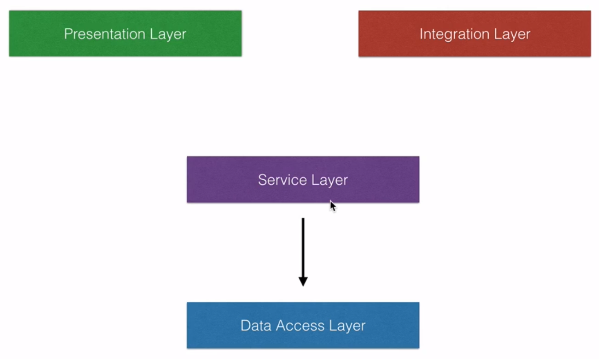
1. When it is hard to change some classes because they are tightly coupled.
2. When we want to reuse behavior of some component in different program/contexts.

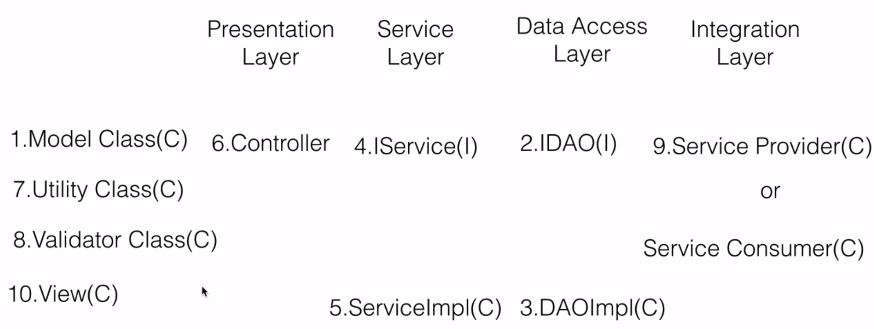
* **Disadvantages** –

1. Over a period of time, Mediator can become God object.
2. **Memento Pattern –**

* It is used to restore state of an object to a previous state without violating encapsulation.
* Promote undo or rollback to full object status.
* Also known as **Snapshot pattern** and **Token**.

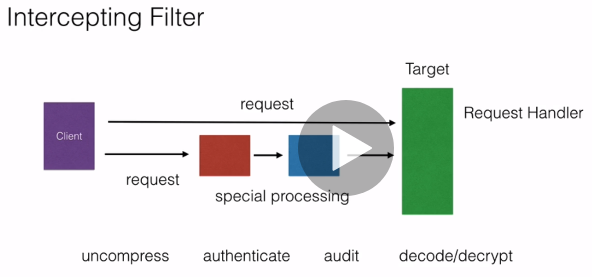
1. **Iterator Pattern –**
2. **JEE Application Layers –**



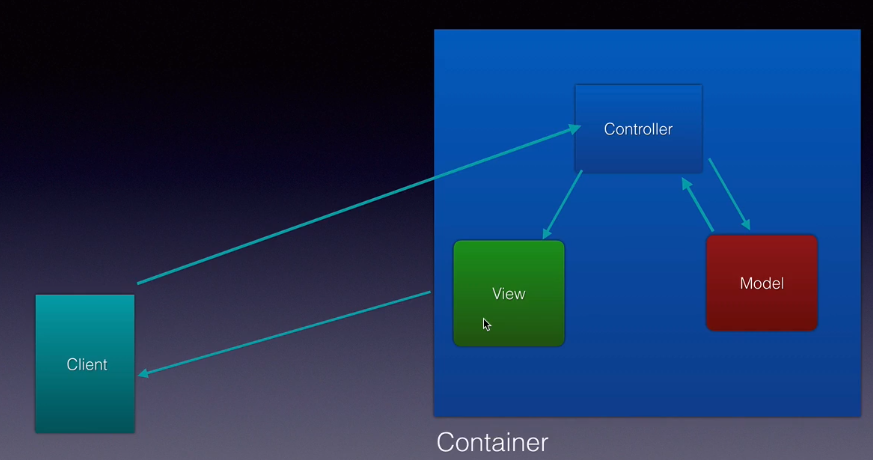


**Why to use layers –**

1. Simplicity
2. Separation of concerns
3. Easy maintenance
4. **Intercepting Filter Pattern –**



1. **Model View Controller (MVC) Pattern –**



1. Easy maintenance
2. Supports Parallel development

Interview Question –

1. <https://www.geeksforgeeks.org/design-parking-lot-using-object-oriented-principles/>
2. <https://howtodoinjava.com/interview-questions/core-java-interview-questions-series-part-1/>