**DESIGN PATTERNS**

1)design patterns is a proven solution to a commonly occurring problem.design pattern promotes software reuse.reuse of software component potentially low us production cost and safe time by eliminating redesign.

2)a design pattern is not a finished design that can be transform directly into source code.

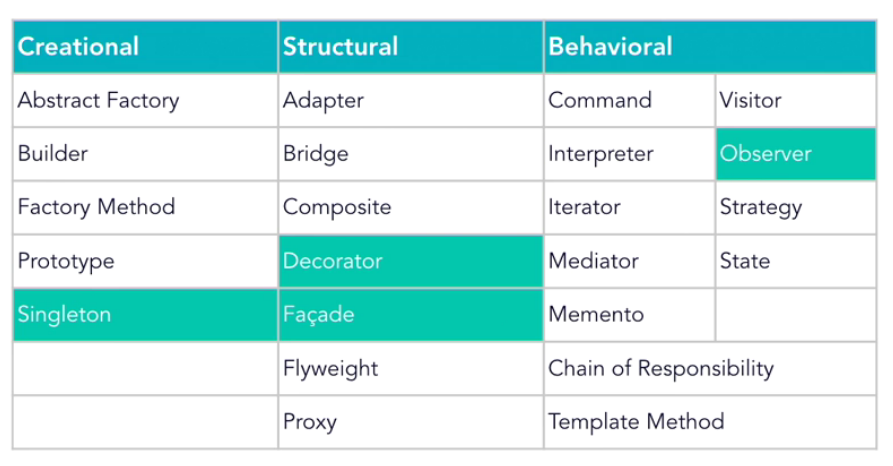
3)a design pattern can have many different implementation and be implemented in many different programming languages.

4)few commonly use design patterns are

->singleton design pattern.

->Data Access Object(DAO) design pattern.

->Model View Controller(MVC) design pattern.



## Java Singleton

The class should be instantiated only once per [classloader](http://www.javabeat.net/classloader-java/). Here is the basic steps for implementing the Java Singleton class.

* Constructor of that class has to be made as private to avoid instantiation from external classes.
* Declare a static variable to store the instance for that class.
* Declare a method that returns the instance of that class.

e.g.-connection pool

2)there are many ways where we can implement singleton design pattern.few are

1. Eager Initialization
2. Static Block Initialization
3. Lazy Initialization
4. Thread Safe Singleton

**Eager Initialization**

* + the singleton instance is created at the time of loading class loading.
  + But, the main drawback of this approach is that the instance will be created even though it is not used by application.
  + This approach doesn’t provide any option for [exception handling](http://www.javabeat.net/java-exception-handling/).

|  |
| --- |
| public class EagerInitialization { |

|  |  |
| --- | --- |
| 2 | private static final EagerInitialization instance = new EagerInitialization(); |

|  |  |
| --- | --- |
| 3 | private EagerInitialization(){} |

|  |  |
| --- | --- |
| 4 | public static EagerInitialization getInstance(){ |

|  |  |
| --- | --- |
| 5 | return instance; |

|  |  |
| --- | --- |
| 6 | } |

|  |  |
| --- | --- |
| 7 | } |

### Static Block Initialization

* + This also falls under the eager initialization.
  + the only difference is that instance creation is completed inside the [static initializer](http://www.javabeat.net/java-static-initializer-instance-initializer-constructor/) block.
  + This approach doesn’t provide any option for [exception handling](http://www.javabeat.net/java-exception-handling/).

|  |  |
| --- | --- |
| 1 | public class StaticBlockInitialization { |
| 2 | private static StaticBlockInitialization singletonInstance; | |

|  |  |  |
| --- | --- | --- |
| 3 | private StaticBlockInitialization(){} | |
| 4 | static{ |

|  |  |
| --- | --- |
| 5 | try{ |
| 6 | singletonInstance = new StaticBlockInitialization(); | |

|  |  |
| --- | --- |
| 7 | }catch(Exception e){ |
| 8 | throw new RuntimeException("Exception occured while creating the singleton instance"); | |

|  |  |  |
| --- | --- | --- |
| 9 | } | |
| 10 | | } | |

|  |  |  |
| --- | --- | --- |
| 11 | public static StaticBlockInitialization getInstance(){ | |
| 12 | return singletonInstance; |

|  |  |  |
| --- | --- | --- |
| 13 | } | |
| Lazy Initialization  * + we are creating the instance only after loading the class and first invocation of the instance.   + This will avoid pre-initializing the instance where you don’t need the instance it is requested by the service.   + This approach works fine when this class is executed in the single threaded environment, but this will not work fine in the multi-threaded environment when multiple threads are trying to get the new instance simultaneously .  |  |  | | --- | --- | |  |  |  |  |  | | --- | --- | | 11 | class Singleton { | | 12 | private Singleton() {} | |  |  |  |  | | --- | --- | --- | | 13 | private static Singleton singleton; | | |  |  |  |  |  |  | | --- | --- | --- | | 15 | public static Singleton getInstance() { | | | 16 | if (singleton == null) { |  |  |  |  | | --- | --- | --- | | 17 | singleton = new Singleton(); | | | 18 | } |  |  |  |  | | --- | --- | --- | | 19 | return singleton; | | | 20 | } |  |  |  | | --- | --- | |  | } | | Thread Safe Singleton  * + This is similar to the above approach except that the method used for creating the singleton instance is [synchronized](http://www.javabeat.net/threads-synchronization/).   This approach is the most safest and best solution to implement the singleton instance, but it has the performance issues when multiple threads to trying to access the instance simultaneously.   |  |  | | --- | --- | | 1 | public class ThreadSafeInstance { | | 2 | private static ThreadSafeInstance singletonInstance; | |  |  |  | | --- | --- | | 3 | private ThreadSafeInstance(){} | | 4 | public static synchronized ThreadSafeInstance getInstance(){ | |  |  |  | | --- | --- | | 5 | if(singletonInstance == null){ | | 6 | singletonInstance = new ThreadSafeInstance(); | |  |  |  | | --- | --- | | 7 | } | | 8 | return singletonInstance; | |  |  |  |  |  | | --- | --- | --- | --- | | 9 | } | | | | 10 | | } | |  | |  | |
| 14 | } |

**Data Access Object(DAO) design pattern**

Benefit-maintainance,reusability

1)Data Access Object(DAO) design pattern will abstract and encapsulate all access to the DB data.

2)DAO completely hides the data handling implementation details from its client.because the interface expose by the DAO to clients doesn’t change when the underlying data handling logic changes.

3)this patterns allows the application to use different DB without affecting its client or business component.

4)the DAO manages the connection(by using connection pool) with the DB to obtain and store data.

## Design Patterns

* Design patterns is used for reusability that leads to more robust and highly maintainable code.

## Creational Design Patterns

solution to instantiate a object in the best possible way for specific situations.

Singleton Design Pattern

* Allow to create only one object within the class loader.
* For that we have to do below steps.

Create one reference variable as private and static.

Create a constructor as private.

Create a factory method as public static and synchronized.

* By using class name if we call any method and return the same class object then that method is called factory method.

Factory Design Pattern

* Based on input parameter we are calling the implementation class. That is written in factory class so that client can’t know that which class we have called.



# Abstract Factory Design Pattern

* + In factory design pattern, Suppose we have to add new implementation class then every time we have to change the factory class and put the condition and no of condition will be huge. So at the result very hard to maintain the code and very tightly coupled of factory and implementation classes.
  + Resolving the above problem we have to use abstract factory design pattern.

# 

# Builder Design Pattern

* + Suppose I am creating the object using constructor then I have to set all the parameters and I have to remember the sequence of the parameter. If someone doesn’t want all the parameter(mandatory and optional) then problem will come through constructor.
  + Resolving above problem we can use builder design pattern.

# Prototype Pattern

Prototype pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object.

## Structural Design Patterns

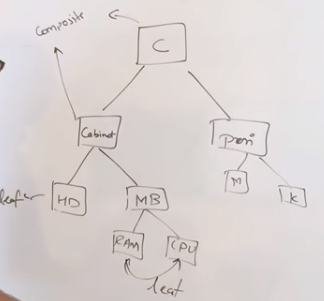
* + Suppose we have two objects and one object want to use another object functionality then we have to use structural design pattern. This is used to compose one object into another.

# Adapter Pattern

* + Suppose we have two interfaces or two classes or one interface and one class and they can’t communicate to each other. So we can create one adapter class and in that we can do that.
  + Ex: I have to write an assignment so for that we need pen. But I don’t have pen implementation but I have pilot pen implementation. So for that I will write one adapter class then I will take pilot pen object and provide pen object.

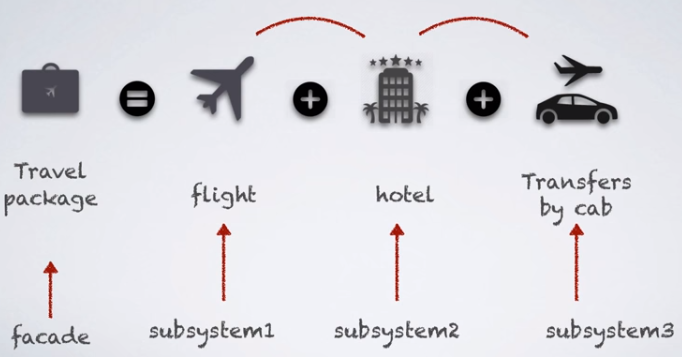
# Composite Pattern

* + Used to create structure of the object.
  + If we have to create an object of the class that contains multiple objects for that we have to use composite.



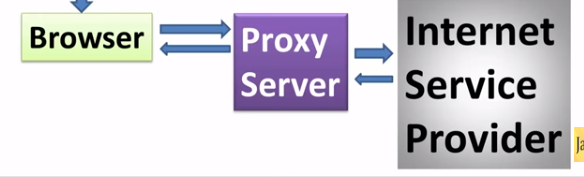
# Facade Pattern

* + In this pattern client will interact with one facade interface and that interface will handle all the internal subsystem.



# Proxy Pattern

* + It is used to authorize something or add some functionality on behalf of the original object.
  + Ex: preprocessing ,postprocessing,logging,filtering, caching or security mechanism.



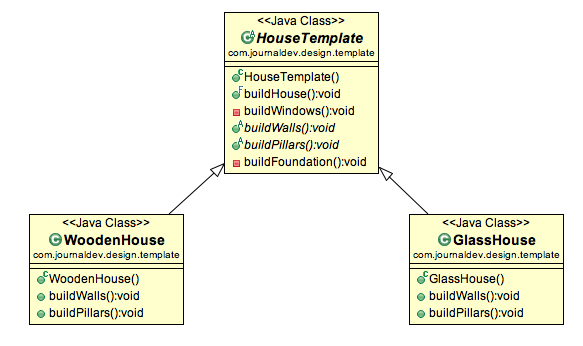
## Behavioral Design Patterns

* + Used to communicate between two objects.

solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

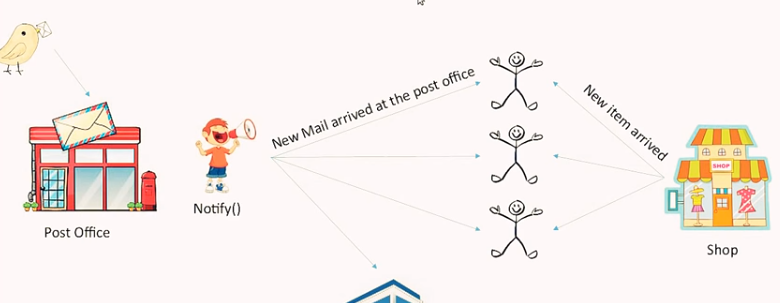
### Template Method Pattern

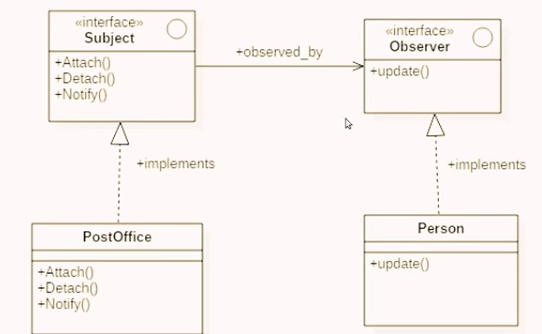
Template method should consists of certain steps whose order is fixed and for some of the methods, implementation differs from base class to subclass. Template method should be final.



# Obeserver Pattern

* + It is used to make the relationship between shop and customer. If any new item has been arrived then no need to go to shop and check but shop will notify to each customer.





# Strategic design Pattern

* + Based on strategic we have to implement the logic. EX: Sorting

Design principles

* + Single Responsibility Principle(SRP) : Only one reason to change. It means that class have one task to do.
  + Open Closed Principle(OCP) : Open for extension but closed for modification.
  + Liskov Substitution Principle(LSP) : The reference of base class can be replaced with a derived class without affecting the functionality of the program module.
  + Interface Segregation Principle(ISP) : client should not force to implement interfaces they don’t use.
  + Dependency Inversion Principle(DIP) : class should depend only on abstractions but not on concretions.