Design Pattern Course Content Details

07 November 2021 15:41

◆ Pre-Requisite : Knowledge on Core Java

◆ Course Content

- What Is a Design Pattern?
- How Design Patterns Solve Design Problems ?
- How to Select a Design Pattern ?
- How to Use a Design Pattern ?
- What are the different Type of Design Patterns ?
- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy

- **Template Method**
- **Visitor**

- **Course Duration: 45 Days**

Design Pattern

08 November 2021 08:57

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples.

The concept of patterns was first described by Christopher Alexander in A Pattern Language: Towns, Buildings, Construction. The book describes a “language” for designing the urban environment. The units of this language are patterns. They may describe how high windows should be, how many levels a building should have, how large green areas in a neighborhood are supposed to be, and so on.

The idea was picked up by four authors: Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. In 1994, they published Design Patterns: Elements of Reusable Object-Oriented Software, in which they applied the concept of design patterns to programming. The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly. Due to its lengthy name, people started to call it “the book by the gang of four” which was soon shortened to simply “the GoF book”.

Since then, dozens of other object-oriented patterns have been discovered. The “pattern approach” became very popular in other programming fields, so lots of other patterns now exist outside of object-oriented design as well.

Why should I learn patterns?

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern. A lot of people do just that. Even in that case, though, you might be implementing some patterns without even knowing it. So why would you spend time learning them?

Design patterns are a toolkit of tried and tested solutions to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.

Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern and its name.

Classification of patterns

Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed. I like the analogy to road construction: you can make an intersection safer by either installing some traffic lights or building an entire multi-level interchange with underground passages for pedestrians.

The most basic and low-level patterns are often called idioms. They usually apply only to a single programming language.

The most universal and high-level patterns are architectural patterns. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

In addition, all patterns can be categorized by their intent, or purpose. Three main groups of patterns are

Creational patterns provide object creation mechanisms that increase flexibility and reuse of existing code.

Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

Behavioral patterns take care of effective communication and the assignment of responsibilities between objects.

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Factory Method Pattern
Abstract Factory Pattern
Singleton Pattern
Prototype Pattern
Builder Pattern

Structural patterns

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

Adapter Pattern
Adapting an interface into another according to client expectation.
Bridge Pattern
Separating abstraction (interface) from implementation.
Composite Pattern
Allowing clients to operate on hierarchy of objects.
Decorator Pattern
Adding functionality to an object dynamically.
Facade Pattern
Providing an interface to a set of interfaces.
Flyweight Pattern
Reusing an object by sharing it.
Proxy Pattern
Representing another object.

Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

Chain of Responsibility Pattern
Command Pattern
Interpreter Pattern
Iterator Pattern
Mediator Pattern
Memento Pattern
Observer Pattern
State Pattern
Strategy Pattern
Template Pattern
Visitor Pattern

Singleton Design Pattern

08 November 2021 18:36

- **Singleton pattern** is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of the class is created
  A singleton class shouldn't have multiple instances in any case and at any cost.
  Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

- **Examples** of Singleton class
  java.lang.Runtime : Java provides a class Runtime in its lang package which is singleton in nature. Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime() method.
  An application cannot instantiate this class so multiple objects can't be created for this class. Hence Runtime is a singleton class.

  In the real-time situation, This Singleton Design Pattern can be used in the below use cases
  - Accessing the Configuration files
  - Creating the Data base connection.
  - Accessing the Loggers

- **Steps to Create the Singleton class**

  1. Whichever the class that you want to have only 1 object throughout the application, that class should have the constructor as Private. In this way no one can create the object from other classes.
  2. Whichever the class that you want to have only 1 object or which you want to have only single object, Make that class as Final so that inheritance cannot be possible and no one can create the object.
  3. Create the global instance of the class like below and make that object as static and private
     static SingletonExample s= new SingletonExample();
  4. Create a method called getInstance() or any other meaning full name and return the above instance using this method
  5. By using this getInstance() method object is returned
  6. Creating the object at class level is called, Early initialization
  7. In Early initialization, even though object is created without calling the getInstance() method. This is the drawback of early initialization hence we go for lazy initialization

8. In Lazy initialization object is create like below

```
static SingletonExample s= null;
public static SingletonExample getInstance() throws Exception {
        if(s==null)
        {
            s= new SingletonExample();
        }
        return s;
    }
```

9. In the above code when the getInstance() is called for the first time, global object is null and hence object will be created, and from the second time onwards, global object is not null and hence no more new object gets created.

- **How to Break the Singleton Design Pattern**

- **Reflection:** Using Reflection, Even though we create the private constructor, this constructor can be called using reflection and which will create the new object.
  To avoid this in the constructor, we should throw exception when the object is not null or assign the global instance with the instance that is already present instead of creating object with new Keyword.

- **De Serialization**: During the de Serialization, readObject method of Object Input Stream will call the readResolve method and which will create the new object.
  To Avoid this override the readResolve method and return the same object instead of creating the new object

- **Cloning:** Using the cloning concept we can create the new object because default implementation of clone method will provide the new object like (return super.clone)
  To avoid this, Override the clone method, and return the object that is created instead of creating the new object or throw the exception.

- **Multi-Threading**: In Multi-Threading, since the getInstance is static method there may be a possibility that 2 or more threads can access the getInstance method at a time and end of creation of multiple objects.
  To avoid this make the getInstance method as Synchronized so that when one thread is accessing the get Instance method, Class lock will acquire and no other thread can access the getInstance method at the same time
  Instead of creating the Synchronized method writing the synchronized block will be better

- **Double Checked locking:**
  Inside the getInstance method, we are writing the null check 2 times, this is referred as Double checked locking.
  When multiple threads are working, when there is an object no need to enter the synchronized loop hence one more condition is added.

Real time uses of Singleton Design Pattern

- Hardware access
- Database connections
- Config files
- Configuration File: This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration file or properties file. It creates a single instance of the configuration file which can be accessed by multiple calls concurrently as it will provide static config data loaded into in-memory objects. The application only reads from the configuration file for the first time and thereafter from second call onwards the client applications read the data from in-memory objects.

When you use Logger, you use Singleton pattern for the Logger class. In case it is not Singleton, every client will have its own Logger object and there will be concurrent access on the Logger instance in Multithreaded environment, and multiple clients will create/write to the Log file concurrently, this leads to data corruption.

Design Pattern

08 November 2021 22:28

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples.

The concept of patterns was first described by Christopher Alexander in A Pattern Language: Towns, Buildings, Construction. The book describes a "language" for designing the urban environment. The units of this language are patterns. They may describe how high windows should be, how many levels a building should have, how large green areas in a neighborhood are supposed to be, and so on.

The idea was picked up by four authors: **Erich Gamma**, **John Vlissides**, **Ralph Johnson**, and **Richard Helm**. In 1994, they published Design Patterns: Elements of Reusable Object-Oriented Software, in which they applied the concept of design patterns to programming. The book featured 23 patterns solving various problems of object-oriented design and became a best-seller very quickly. Due to its lengthy name, people started to call it **"the book by the gang of four" which was soon shortened to simply "the GoF book".**
Since then, dozens of other object-oriented patterns have been discovered. The "pattern approach" became very popular in other programming fields, so lots of other patterns now exist outside of object-oriented design as well.

Why should I learn patterns?

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern. A lot of people do just that. Even in that case, though, you might be implementing some patterns without even knowing it. So why would you spend time learning them?

Design patterns are a toolkit of tried and tested solutions to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.

Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, "Oh, just use a Singleton for that," and everyone will understand the idea behind your

suggestion. No need to explain what a singleton is if you know the pattern and its name.

Classification of patterns

Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed. I like the analogy to road construction: you can make an intersection safer by either installing some traffic lights or building an entire multi-level interchange with underground passages for pedestrians.

The most basic and low-level patterns are often called idioms. They usually apply only to a single programming language.

The most universal and high-level patterns are architectural patterns. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

In addition, all patterns can be categorized by their intent, or purpose. Three main groups of patterns are

Creational patterns provide object creation mechanisms that increase flexibility and reuse of existing code.

Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

Behavioral patterns take care of effective communication and the assignment of responsibilities between objects.

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Factory Design Pattern

13 November 2021 19:53

- The factory design pattern says that define an interface (A java interface or an abstract class) and let the subclasses decide which object to instantiate. The factory method in the interface lets a class defer the instantiation to one or more concrete subclasses

- In Factory pattern, we create objects without exposing the creation logic to the client and the client uses the same common interface to create a new type of object.

- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

- A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate**

- The Factory Method Pattern is also known as **Virtual Constructor.**

Below is the Application in which we use Factory Design Pattern
Whenever we develop an app, we do have different type of notifications, like SMS Notification, Email Notification and Push Notification.
Here Factory will decide which class to be instantiated.

```
public interface Notification {
    void notifyUser();
}
```

```
public class SMSNotification implements Notification {

    @Override
    public void notifyUser()
    {
        // TODO Auto-generated method stub
        System.out.println("Sending an SMS notification");
    }
}

public class EmailNotification implements Notification {

    @Override
    public void notifyUser()
```

```
    {
        // TODO Auto-generated method stub
        System.out.println("Sending an e-mail notification");
    }
}

public class NotificationFactory {
    public Notification createNotification(String channel)
    {
        if (channel == null || channel.isEmpty())
            return null;
        if ("SMS".equals(channel)) {
            return new SMSNotification();
        }
        else if ("EMAIL".equals(channel)) {
            return new EmailNotification();
        }
        else if ("PUSH".equals(channel)) {
            return new PushNotification();
        }
        return null;
    }
}

public class NotificationService {
    public static void main(String[] args)
    {
        NotificationFactory notificationFactory = new
NotificationFactory();
        Notification notification =
notificationFactory.createNotification("SMS");
        notification.notifyUser();
    }
}
```

➢ Real time Uses of Factory Design Pattern:

- ➢ getInstance() method of java.util.Calendar, NumberFormat, and ResourceBundle uses factory method design pattern.

- ➢ Consider an example of using multiple database servers like SQL Server and Oracle. If you are developing an application using SQL Server database as backend, but in future need to change backend database to oracle, you will need to modify all your code, if you haven't written your code following factory design pattern.
- ➢ In factory design pattern you need to do very little work to achieve this. A class implementing factory design pattern takes care for you and lessen your burden. Switching from database server won't bother you at all. You just need to make some small changes in your configuration file. (In Factory class pick the configuration file or data base connection based on the type of database then, in future there is no need to modify the change whenever data base is switched)

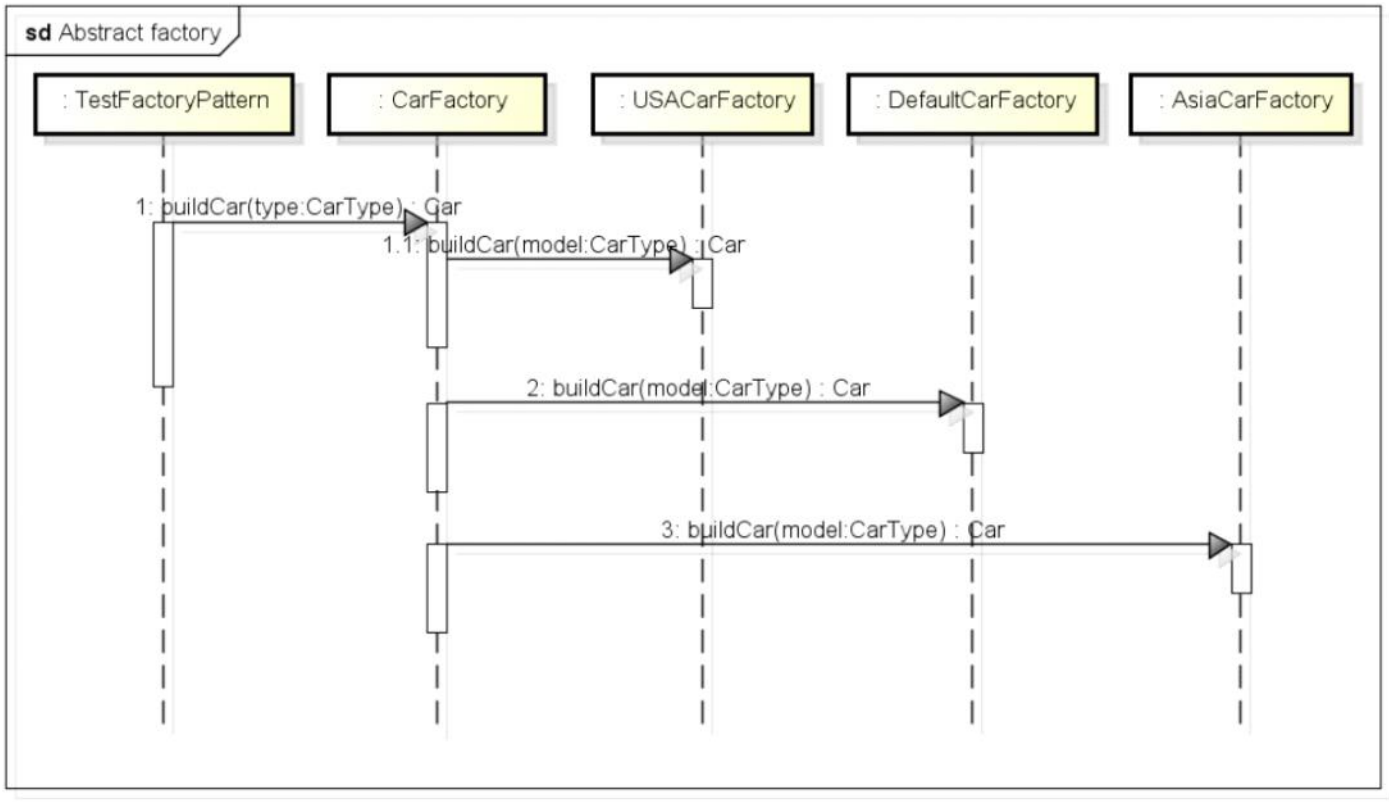
Abstract Factory Design Pattern

15 November 2021 19:41

- The Abstract Factory is known as a **creational** pattern - it's used to construct objects such that they can be decoupled from the implementing system. The definition of Abstract Factory provided in the original Gang of Four book on Design Patterns states

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes

- Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern



- **Implementation of Abstract Factory Design Pattern**

  - Create an Abstract Factory as an Interface or an Abstract Class.
  - Abstract factory defines abstract methods for creating products.
  - Provide a concrete implementation of factory for each set of products.
  - Abstract Factory makes use of factory method pattern. Abstract Factory is an object with multiple factory methods.

- Where Would I Use This Pattern?

  - The pattern is best utilised when your system has to create multiple families of products or you want to provide a library of products without exposing the implementation details

- Here are some examples from core Java libraries:
  - **javax.xml.parsers.DocumentBuilderFactory#newInstance()**
  - **javax.xml.transform.TransformerFactory#newInstance()**
  - **javax.xml.xpath.XPathFactory#newInstance()**
- **Identification:** The pattern is easy to recognize by methods, which return a factory object. Then, the factory is used for creating specific sub-components.

- Let's take an example, Suppose we want to build a global car factory. If it was a factory design pattern, then it was suitable for a single location. But for this pattern, we need multiple locations and some critical design changes.
- We need car factories in each location like IndiaCarFactory, USACarFactory, and DefaultCarFactory. Now, our application should be smart enough to identify the location where it is being used, so we should be able to use the appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling the wrong factory for a particular location.
- Here we need another layer of abstraction that will identify the location and internally use correct car factory implementation without even giving a single hint to the user. This is exactly the problem, which an abstract factory pattern is used to solve.

- **Difference between Abstract Factory and Factory**

  - With the Factory pattern, you produce instances of implementations (Apple, Banana, Cherry, etc.) of a particular interface -- say, IFruit.
  - With the Abstract Factory pattern, you provide a way for anyone to provide their own factory. This allows your warehouse to be either an IFruitFactory or an IJuiceFactory, without requiring your warehouse to know anything about fruits or juices.

- **Example:**

  - Assume that we have one car company, which is providing the services in India, US and other countries
  - When the customer came to website and put a request, (Since the web site is common for all the countries).
  - If the customer who is in INDIA puts a request, I have to redirect the request to Car show room present in INDIA at the same time that show room has also multiple models of the car.
  - So Here based on the GPS location of the customer we can detect the country and forward the request to corresponding country servicing center and that service center will give the details of all the cars that they have.

Strategy Design Pattern

16 November 2021 08:51

- Strategy pattern (also known as the policy pattern) is a behavioural software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use
- The application can switch strategies at run-time.
- Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.

- When to Use Strategy Design Pattern ?

- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime. Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behaviour.

- Here some examples of Strategy in core Java libraries:
  - java.util.Comparator#compare() called from Collections#sort().
  - javax.servlet.http.HttpServlet: service() method, plus all of the doXXX() methods that accept HttpServletRequest and HttpServletResponse objects as arguments.
  - javax.servlet.Filter#doFilter()

In order to explain the Strategy Design Pattern, consider the below example. Example of Bank, which has Privileged customer and Normal customer and interest rates also will be different for each type of customer.
So In future this functionality may be extended for different customers or for example Employee of bank.
If we implement this functionality using if and else conditions then we may spoil the Open closed and single responsibility principle so in order to implement this we go for Strategy, such that at run time it will be decided which algorithm can be implemented.

1. Create the interface and make 1 abstract method
2. Create Privileged customer and Un Privileged customer class and extend the interface and override the abstract method for the interest rate
3. Create a separate class and create the constructor for this class which accepts interface reference mentioned in point 1 as parameter

4. Create 1 method and on this object call the abstract method overridden in the Privileged and Un Privileged classes
5. From the Main class whenever u pass the Privileged Object then Privileged class interest methods gets invoked
6. And if we want to extend this functionality for Employee in future just create New class called Employee and implement the interface written in point1.
7. From Main method pass this Employee reference and call the method.
8. In this approach we are not modifying the code but extending the functionality.

Decorator Design Pattern

17 November 2021 06:39

The decorator pattern is a design pattern that allows behaviour to be added to an individual object, dynamically, without affecting the behaviour of other objects with in the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern Decorator patterns allow a user to add new functionality to an existing object without altering its structure. So, there is no change to the original class

Decorator design patterns create decorator classes, which wrap the original class and supply additional functionality by keeping the class methods' signature unchanged.

Decorator design patterns are most frequently used for applying single responsibility principles since we divide the functionality into classes with unique areas of concern

Decorator design pattern is useful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the amount of choices are more

Decorator pattern is used a lot in Java IO classes, like FileReader, BufferedReader, etc.

Procedure:

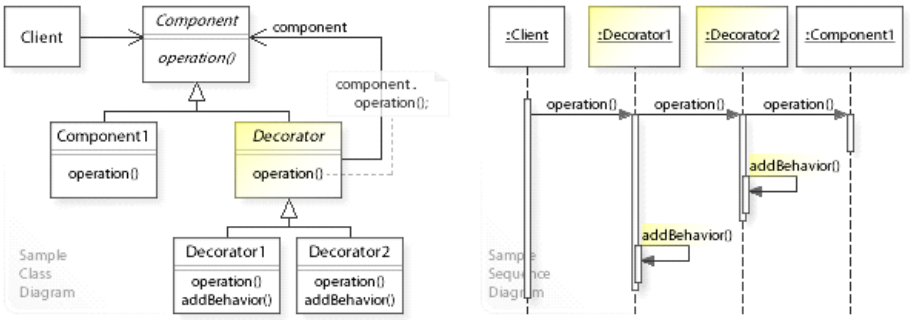
- Create an interface. -> This step corresponds to coffee interface in our example
- Create concrete classes implementing the same interface. ->This step corresponds to Simple Coffee class in our example
- Create an abstract decorator class implementing the above same interface. ->This step corresponds to Coffee Decorator abstract class in our example
- Create a concrete decorator class extending the above abstract decorator class. ->This step corresponds to WithMilk and With Water classes in our example
- Now use the concrete decorator class created above to decorate interface objects. -> This step corresponds to DecorratorDesignPatternExample in our example where main method is present and we are passing simple object and modifying the object behaviour

What problems can it solve?

- Responsibilities should be added to (and removed from) an object dynamically at run-time
  The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at run-time for selected objects.

Alternatives to Decorator

Pattern	Intent
Adapter	Converts one interface to another so that it matches what the client is expecting
Decorator	Dynamically adds responsibility to the interface by wrapping the original code
Facade	Provides a simplified interface

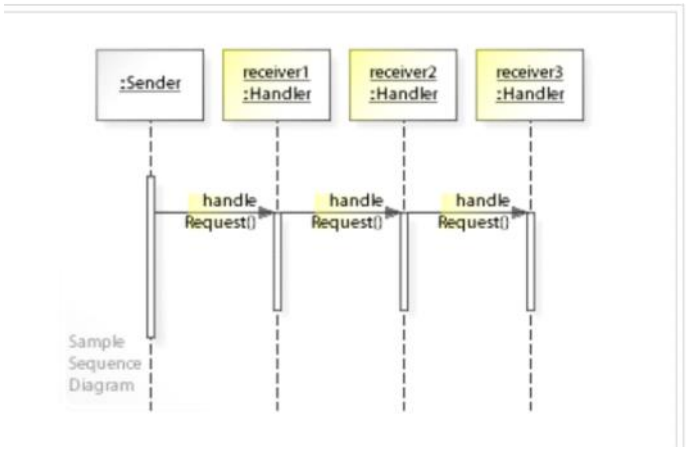


A sample UML class and sequence diagram for the Decorator design pattern.[6]

Chain of Responsibility Design Pattern

18 November 2021 06:39

- Chain of Responsibility as a design pattern consisting of “a source of command objects and a series of processing objects”.



What problems can the Chain of Responsibility design pattern solve?

- Coupling the sender of a request to its receiver should be avoided.
- It should be possible that more than one receiver can handle a request.

- In the Java world, we benefit from Chain of Responsibility every day. **One such classic example is Servlet Filters in Java** that allow multiple filters to process an HTTP request. Though in that case, **each filter invokes the chain instead of the next filter.**

```
public class CustomFilter implements Filter{ public void doFilter( ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException { // process the request // pass the request (i.e. the command) along the filter chain chain.doFilter(request, response); } }
```

- **Chain of Responsibility Pattern Example in JDK**

Let’s see the example of chain of responsibility pattern in JDK and then we will proceed to implement a real life example of this pattern. We know that we can have multiple catch blocks in a try-catch block code. Here every catch block is kind of a processor to process that particular exception.
So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

- Look at the ATM example that we discussed in the class that , ATM dispenses 50,20 and 10 Notes
When we enter the amount to be withdrawn as 230 then first request should be handled by 50 Notes class. So 230/50- >4 -> 50 Notes and after that remaining amount is 30 -> so in order to dispense the remaining 30 the same request to be transferred to another class that is 20 Notes
So here we are following the chain and forwarding the request from one class to other class
Remaining amount 30/20 ->One 20 Note and remaining amount is 10.
To dispense this 10 then we should forward the request to 10 Note section.

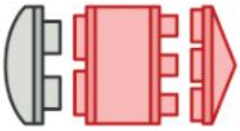
10/10 -> One 10 Notes. Done with the process
Here in this example one request passed to different object and following the chain.

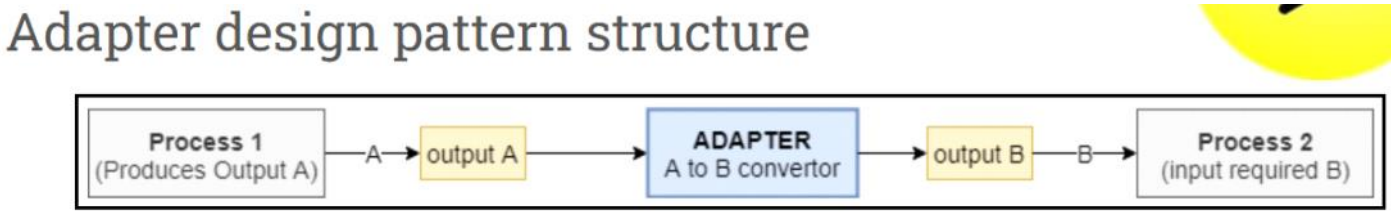
Disadvantages

- And now that we've seen how interesting Chain of Responsibility is, let's keep in mind some drawbacks:
  - Mostly, it can get broken easily:
    - if a processor fails to call the next processor, the command gets dropped
    - if a processor calls the wrong processor, it can lead to a cycle
  - It can create deep stack traces, which can affect performance
  - It can lead to duplicate code across processors, increasing maintenance

Adapter Design Pattern

19 November 2021 08:03





- The **adapter pattern** is a software design pattern that allows the interface of an existing class to be used as another interface
- It is often used to make existing classes work with others without modifying their source code.
- **Simple definition in other words**
- The adapter design pattern is a structural design pattern that allows two unrelated/uncommon interfaces to work together. In other words, the adapter pattern makes two incompatible interfaces compatible without changing their existing code.
- Adapter patterns use a single class (the adapter class) to join functionalities of independent or incompatible interfaces/classes.
- The Adapter pattern is a structural design pattern that acts as a bridge between two interfaces that are incompatible. The term "Adapter" is used to represent an object that enables two mutually incompatible interfaces to communicate and collaborate
- In essence, the Adapter pattern enables classes (that have incompatible interfaces) to work together and their objects communicate if need be.
- For Implementation please look at the example that we discussed in the class of implementing the different payment gateway in Insurance Application from Billdesk to Razopay

Example implementations of Adapter Design Pattern

- Some other examples worth noticing is as below:
  1) java.util.Arrays#asList()

This method accepts multiple strings and return a list of input strings. Though it's very basic usage

- 2) java.io.OutputStreamWriter(OutputStream)

Builder Design Pattern

20 November 2021 23:21

- The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation.
- It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

- Thus, there are two specific problems that we need to solve:
  - Too many constructor arguments.
  - Incorrect object state.

Advantages of Builder Design Pattern

- The parameters to the constructor are reduced and are provided in highly readable method calls.
- Builder design pattern also helps in minimizing the number of parameters in the constructor and thus there is no need to pass in null for optional parameters to the constructor.
- Object is always instantiated in a complete state
- Immutable objects can be built without much complex logic in the object building process.

- All implementations of java.lang.Appendable are in fact good example of use of Builder pattern in java. e.g. java.lang.StringBuilder#append() [Unsynchronized class]

  java.lang.StringBuffer#append() [Synchronized class]

  java.nio.ByteBuffer#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)

- Look how similar these implementations look to what we discussed above.
  StringBuilder builder = new StringBuilder("Temp");

```
String data = builder.append(1)
      .append(true)
      .append("friend")
      .toString();
```

- Separate the construction of a complex object from its representation so that the same construction process can create different representations
- The Builder pattern allows us to write readable, understandable code to set up complex objects

When to use Builder Pattern:

- If you find yourself in a situation where you keep on adding new parameters to a constructor, resulting in code that becomes error-prone and hard to read, perhaps it's a good time to take a step back and consider refactoring your code to use a Builder.

- Look at the User class and User Builder class which we created in the class to implement builder design pattern. User class can have many user attributes, like first name, last name, email, mobile, address
  And in future we can have more. If we create the constructor with these fields if the field is optional then we have to pass null and every time we should be conscious while creating the object, we may pass address in place of email if we mistaken.

- To avoid all these issues instead of calling the constructor, we can create the method and call these methods to create the object. So complex object can be easily developed with this approach
- We created User Builder inner class and added all the attributes using the methods and created the User object

Observer Design Pattern

23 November 2021 08:13

- Observer is a behavioural design pattern. It specifies communication between objects: *observable* and *observers*. **An *observable* is an object which notifies *observers* about the changes in its state**

- The **observer pattern** is a software design pattern in which an object, named the **subject**, maintains a list of its dependents, called **observers**, and notifies them automatically of any state changes, usually by calling one of their methods.

- it is mainly used for implementing distributed event handling systems

- In Simple words, If we take the example of Twitter, If 100 users are following Modi then when any tweet is added by modi then this tweet notification will be sent to 100 users.

- For small scale applications, if we want to implement the event driven mechanism then we should go for Observer Design Pattern.

- Look at the example that we have added, created the class for Insurance company and added LIC and TATA as objects
  Created user class and created some 4 users (karthik, ashok, uday etc)
  Who ever is interested for LIC or taken policy with LIC we added then as subscribers so for this added the method in Insurance company class like (Add subscriber, remove subscriber, notify Subscriber etc.)
  Added one more method in this Insurance class called Tweet so when LIC tweets this method gets invoked and under this we can Notify Subscriber.
  Here When ever LIC or TATA announces something then all the subscribers will be notified.

Benefits:

- It describes the coupling between the objects and the observer.
- It provides the support for broadcast-type communication.

Usage:

- When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
- When the framework we writes and needs to be enhanced in future with new observers with minimal changes

Here are some examples of the pattern in core Java libraries:

- **java.util.Observer**/**java.util.Observable** (rarely used in real world)
- All implementations of **java.util.EventListener** (practically all over Swing components)
- **javax.servlet.http.HttpSessionBindingListener**
- **javax.servlet.http.HttpSessionAttributeListener**
- **javax.faces.event.PhaseListener**

Iterator Design Pattern

23 November 2021 08:28

- The **iterator pattern** is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

- Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.
- Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.
- But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

- **The main idea of the Iterator pattern is to extract the traversal behaviour of a collection into a separate object called an *iterator*.**

- In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.
- Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.
- All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

Prototype Design Pattern

24 November 2021 08:23

- **Prototype is a creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes**

- It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
- The client, instead of writing code that invokes the "new" operator on a hard-coded class name, calls the *clone()* method on the prototype, calls a factory method with a parameter designating the particular concrete derived class desired, or invokes the *clone()* method through some mechanism provided by another design pattern

- Prototype Design Participants
  1) Prototype : This is the prototype of an actual object.
  2) Prototype registry : This is used as a registry service to have all prototypes accessible using simple string parameters.
  3) Client : Client will be responsible for using registry service to access prototype instances.

- **Usage examples:** The Prototype pattern is available in Java out of the box with a Cloneable interface.
- Any class can implement this interface to become cloneable.
  - java.lang.Object#clone() (class should implement the java.lang.Cloneable interface)
- **Identification:** The prototype can be easily recognized by a clone or copy methods, etc.

- **Example**: Look at the example that we created,
  When we look at the eligibility of a person for a specific product we need the User Object which contains the personal information, Income, Asset etc. Assume that we are evaluating the criteria of multiple products, each product requires the user object, to get this user object we need to call the DB and get the object which is costly operation. Instead we can use Prototype design pattern which uses clone method and by following the Deep Clone we can create a new User object with the details instead of calling the DB.

Mediator Design Pattern

27 November 2021 09:01

- **The intent of the Mediator Pattern is to reduce the complexity and dependencies between tightly coupled objects communicating directly with one another**.

- This promotes loose coupling, as a set of components working together no longer have to interact directly. Instead, they only refer to the single mediator object. This way, it is also easier to reuse these objects in other parts of the system.

- In object-oriented programming, programs often consist of many classes. Business logic and computation are distributed among these classes. However, as more classes are added to a program, especially during maintenance and/or refactoring, the problem of communication between these classes may become more complex. This makes the program harder to read and maintain. Furthermore, it can become difficult to change the program, since any change may affect code in several other classes.

- With the **mediator pattern**, communication between objects is encapsulated within a **mediator** object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

- Good example of mediator pattern is a chat application. In a chat application we can have several participants. It's not a good idea to connect each participant to all the others because the number of connections would be really high. The best solution is to have a hub where all participants will connect; this hub is just the mediator class.
- In Java programming, the execute() method inside the java.util.concurrent.Executor interface follows this pattern. The different overloaded versions of various schedule() methods of the java.util.Timer class also can be considered to follow this pattern.

- ## Differences between Observer and Mediator Design Pattern

  - The Observer pattern: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

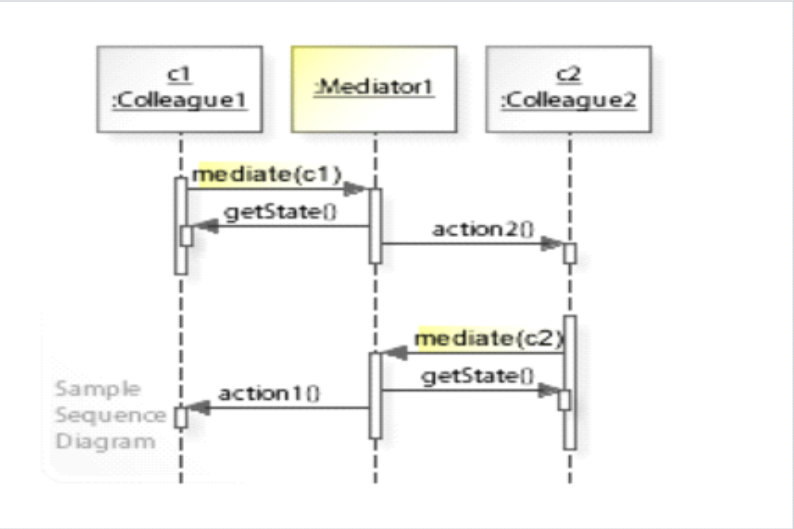
  - The Mediator pattern: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

  - **Mediator is a little more specific, it avoids having classes communicate directly but instead through a mediator**. This helps the Single Responsibility principle by allowing communication to be offloaded to a class that just handles that.

  - For Communication from 1 to 1 without contacting each other we user Mediator

Example:

- The observer pattern: Class A, can have zero or more observers of type O registered with it. When something in A is changed it notifies all of the observers.
- The mediator pattern: You have some number of instances of class X (or maybe even several different types:X, Y & Z), and they wish to communicate with each other (but you don't want each to have explicit references to each other), so you create a mediator class M. Each instance of X has a reference to a shared instance of M, through which it can communicate with the other instances of X (or X, Y and Z).
- Look at the example of Chat application that we developed.



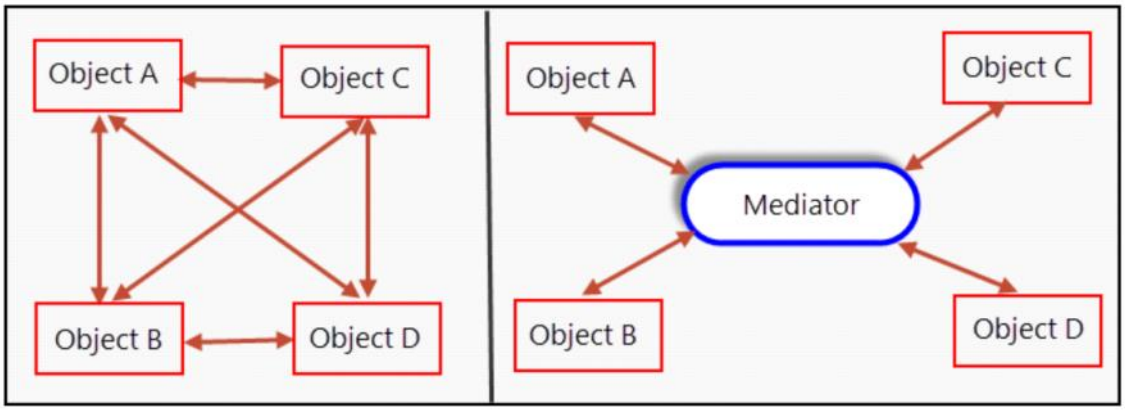
- In the above UML class diagram, the Colleague1 and Colleague2 classes do not refer to (and update) each other directly. Instead, they refer to the common Mediator interface for controlling and coordinating interaction (mediate()), which makes them independent from one another with respect to how the interaction is carried out. The Mediator1 class implements the interaction between Colleague1 and Colleague2.

- In this example, a Mediator1 object mediates (controls and coordinates) the interaction between Colleague1 and Colleague2 objects.

- **Mediator** - defines the interface for communication between *Colleague* objects
- **ConcreteMediator** - implements the Mediator interface and coordinates communication between *Colleague* objects. It is aware of all of the *Colleagues* and their purposes with regards to inter-communication.
- **Colleague** - defines the interface for communication with other *Colleagues* through its *Mediator*
- **ConcreteColleague** - implements the Colleague interface and communicates with other *Colleagues* through its *Mediator*

- A Mediator commonly consists of these four components:
  - Mediator — Defines communication between Colleagues. -
  - Mediator is typically an abstract-based class or Interface

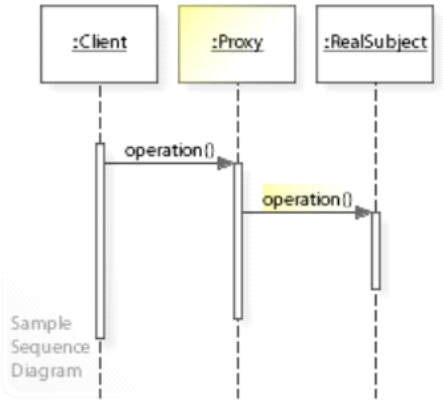
  - Concrete Mediator — Implements communication between Colleagues.
  - Colleague — Communicates only with the Mediator.
  - Concrete Colleague — Receive messages from the Mediator.



Proxy Design Pattern

01 December 2021 08:18

A proxy object hides the original object and control access to it. We can use proxy when we may want to use a class that can perform as an interface to something else.
Proxy is heavily used to implement lazy loading related usecases where we do not want to create full object until it is actually needed.
A proxy can be used to add an additional security layer around the original object as well.



- Subject – is an interface which expose the functionality available to be used by the clients.
- Real Subject – is a class implementing Subject and it is concrete implementation which needs to be hidden behind a proxy.
- Proxy – hides the real object by extending it and clients communicate to real object via this proxy object. Usually frameworks create this proxy object when client request for real object.

Different Proxies

- Remote proxy – represent a remotely lactated object. To talk with remote objects, the client need to do additional work on communication over network. A proxy object does this communication on behalf of original object and client focuses on real talk to do.
- Virtual proxy – delay the creation and initialization of expensive objects until needed, where the objects are created on demand. Hibernate created proxy entities are example of virtual proxies.
- Protection proxy – help to implement security over original object. They may check for access rights before method invocations and allow or deny

access based on the conclusion.

- Smart Proxy – performs additional housekeeping work when an object is accessed by a client. An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.

➢ Difference between Proxy and Decorator Design Pattern

- The primary difference between both patterns are responsibilities they bear. Decorators focus on adding responsibilities, but proxies focus on controlling the access to an object.

- Authentication purpose we can use proxy, call the real object only when we the user tried to access the page who has the proper role mapping.

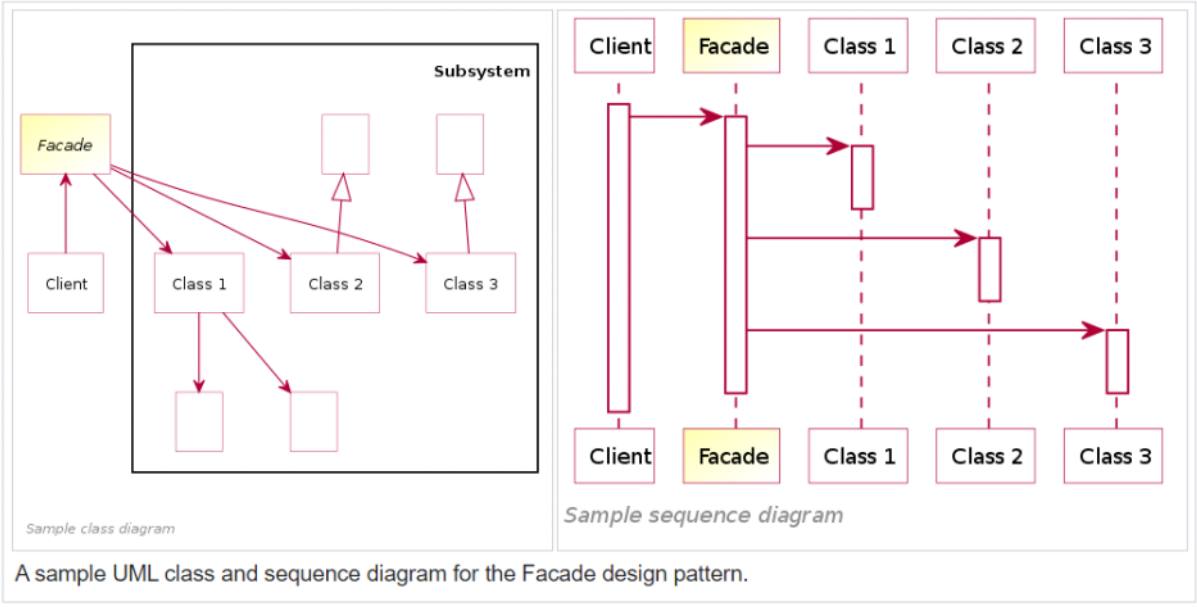
- using the JDK mechanism, by interface. Take a look at java.lang.reflect.Proxy.

Façade Design Pattern

01 December 2021 08:26

- The facade pattern is appropriate when we have a complex system that we want to expose to clients in a simplified way. Its purpose is to hide internal complexity behind a single interface that appears simple from the outside.
- The interface JDBC can be called a facade because, we as users or clients create connection using the “java.sql.Connection” interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.
- we must have connected to a database to fetch some data. We simply call the method dataSource.getConnection() to get the connection but internally a lot of things happen such as loading the driver, creating connection or fetching connection from pool, update stats and then return the connection reference to caller method. It is another example of Facade pattern in the programming world.

- **We can use Factory pattern with Facade to provide better interface to client systems.**



A sample UML class and sequence diagram for the Facade design pattern.

- The **Facade** object is used to provide a front-facing interface by masking a more complex underlying system

- The **Facade** may provide a limited or dedicated set of functionalities. But, the functionalities Facade provides are mainly required by the client application. So, its more caring as per client needs

- A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.
- A facade hides the complexity of subsystem and provides a simple interface to client.
- It decouples the client implementation from the subsystems. We can any of the subsystem without modifying client's code provides client is only interacting through facade.

Bridge Design Pattern

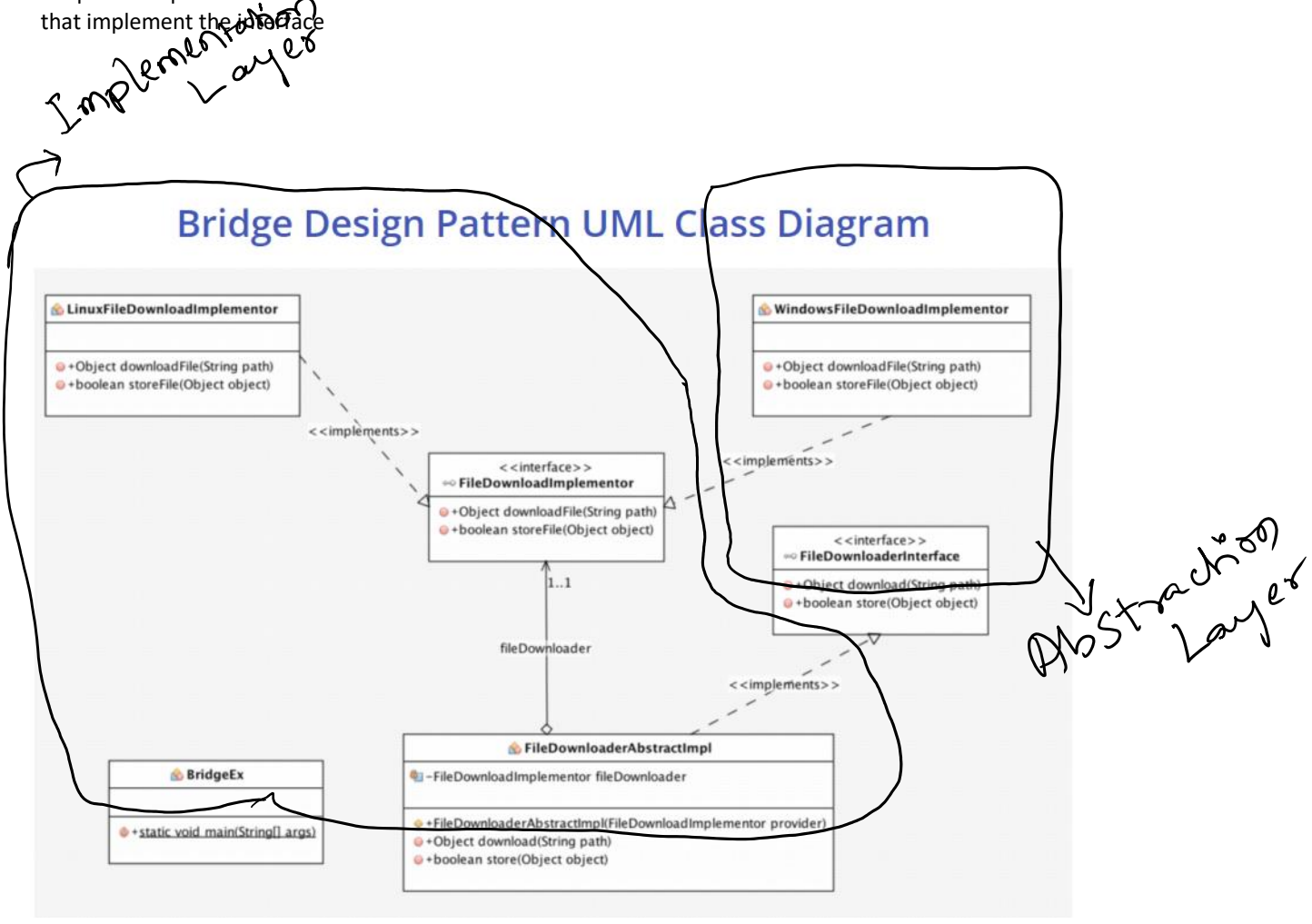
03 December 2021 06:26

- **Difference between Adapter and Bridge is**,
  - Bridge works for new code
  - Adapter works for old code

- Bridge Decouples abstraction and implementation
- Changes in abstraction wont effect the client or implementation layer

- Example for Bridge design pattern is
  JDBC -> which uses the Driver

- Bridge is mainly used for separation of concern in design. We can create an implementation and store it in the interface, which is an abstraction. Where as specific implementation of other features can be done in concrete classes that implement the interface



- Decouple an abstraction from its implementation so that the two can vary independently
- Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
- It is used mainly for implementing platform independence feature

Example: Look at the code that we implemented for File downloader in Linux machine and Windows Machine
Here in this we have interface for implementation layer and interface for abstraction layer, so any changes performed in implementation layer wont effect the abstraction Layer.

Template Design Pattern

06 December 2021 08:15

- The template method is a method in a superclass, usually an abstract superclass, and defines the skeleton of an operation in terms of a number of high-level steps. These steps are themselves implemented by additional *helper methods* in the same class as the *template method*.

- **The intent of the template method is to define the overall structure of the operation, while allowing subclasses to refine, or redefine, certain steps**

- **Template method defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.**

- Let's understand this pattern with an example, suppose we want to provide an algorithm to build a house. The steps need to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that the we can't change the order of execution because we can't build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house

- **To make sure that subclasses don't override the template method, we should make it final.**

- **Since we want some of the methods to be implemented by subclasses, we have to make our base class as abstract class.**

Template Method Design Pattern in JDK

- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.
- All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

- 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

- Most of the times, subclasses calls methods from super class but in template pattern, superclass template method calls methods from subclasses

➢ **Example**:

- Look at the Example of Insurance application, where for each product we wanted to perform some validations like (age calculation, income validation, resident validation)
- All these can be kept as a abstract methods and each product should provide separate implementation for these methods and in the abstract class we can create the build method and we can provide the hierarchy to call these methods, to avoid the overriding of this method in sub classes we should make this method as final so that overriding is not possible.
- If the build method is not final then is can be overridden in the child classes, then when we can the build method child class overridden method gets called then we don't achieve the Template pattern.

Memento Design Pattern

08 December 2021 06:42

- **Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.**

- There are three important classes (Originator, Memento, and Caretaker) involved in the memento design pattern

- The Originator class has two methods. One is CreateMemento and the other one is SetMemento. The SetMemento method accepts the memento object. So, what this Originator class will do is, it will instantiate the Memento object and set the internal state of the originator to memento state. So, basically what the originator will do is, it will take a snapshot of the originator and put it on the memento object. And that memento object will save into the Caretaker for later use.
- Later use means, suppose at a later point of time, originator wants to restore its state then at that time it will take the old state from the caretaker and set it in the originator state so that the originator can restore its old state.

- **Originator:** It creates a memento containing a snapshot of its current internal state and uses the memento to restore its internal state.
- **Memento:** It holds the internal state of an Originator.
- **Caretaker:** It is responsible for keeping the mementos. Like maintaining save points and never operates on or examine the contents of a memento.

- In Real-time applications we need to use the Memento Design Pattern, when
  - The state of an object needs to be saved and restored at a later time.

- **The main purpose of the Memento pattern is to save the internal state of an object and the possibility of restoring it again if necessary, without disturbing encapsulation.**

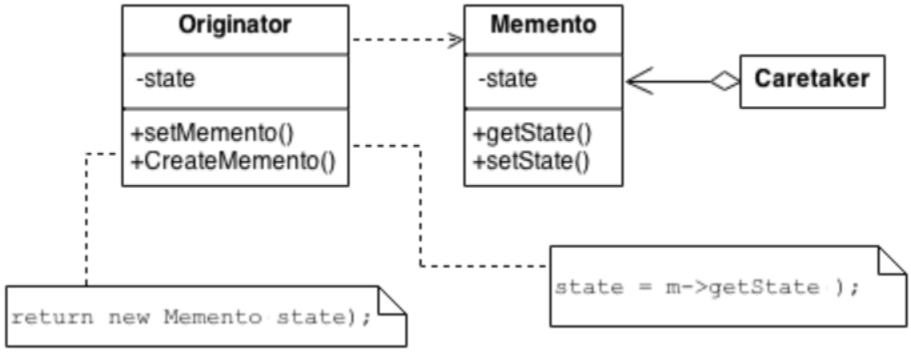
- **Memento design pattern is used to implement rollback feature in an object. In a Memento pattern there are three objects:**

- **Example:**
  - One good use of memento is in online Forms. If we want to show to user a form pre-populated with some data, we keep this copy in memento. Now user can update the form. But at any time when user wants to reset the form, we use memento to make the form in its original pre-populated state. If user wants to just save the form we save

the form and update the memento. Now onwards any new changes to the form can be rolled back to the last saved Memento object.

- Another example is Online complaint form, we can provide the user with two options like save and Undo. When he clicks on save we can save the form in terms of Object (Example: this form has 3 elements, Name, email and Complaint or Feedback text, so create a class with these 3 fields and when clicked on Save store the object in Momento) and later user want to modify the text , email and Name. Once it is modified and user wants to undo then when clicked on Undo, previous Object values like (Name, email id and complaint or feedback text will be read from the Momento) and can be shown. In this way we are restoring or getting the previous state of an object using Memento design pattern.

- Basically Memento Design Pattern works to restore the previous Object without disturbing the encapsulation of the class.
- **With Memento Design Pattern we achieve the CTRL+Z behaviour**



Command Design Pattern

10 December 2021 08:17

- The **command pattern** is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

- Four terms always associated with the command pattern are ***command*, *receiver*, *invoker* and *client*.**

- In the **command design pattern, there's a command object that sits between the sender and the receiver objects**

- The sender object can create a command object. The command object then calls the exposed method in the receiver. And so, the sender object doesn't need to know about the receiver and its exposed methods

- We also have another object known as invoker. **An invoker is an object responsible for invoking the appropriate command object to complete a task**

- Two of the Command pattern examples from JDK are java.lang.Runnable and javax.swing.Action interface. If you look closely you will find that Thread Pool executors are invoker of Runnable commands.

- In one of our projects, we have the following requirement:
  - Create a record in DB.
  - Call a service to update a related record.
  - Call another service to log a ticket.

- To perform this in a transactional manner, each operation is implemented as a command with undo operation. At the end of each step, the command is pushed onto a stack. If the operation fails at some step, then we pop the commands from the stack and call undo operation on each of the command popped out. The undo operation of each step is defined in that command implementation to reverse the earlier command.execute().

- **Example**:
  - The example of Command Design Pattern is, In order to Switch the FAN, we click on SWITCH. We don't know which button corresponds to FAN but we give input to SWITCH BOX, it will decide which electric equipment to be switched ON. That means user will provide the input to the Command Object and command object decides which object to be invoked.

  - We created the application where we created the application with Taking the order, edit the order, delete the order and save the order.

  - When er click on Open Order, request will be sent to Command object and from there request will be transferred to Open Order class

  - Similarly, When Edit action is performed, then Request will be transferred to Command Object and it redirects the request to Edit order. In this way request is given to command and it decides which object to be called.

Fly weight

13 December 2021 08:14

- The main intent of Flyweight design pattern is “Facilitates the reuse of many fine grained objects and makes the use of large numbers of objects more efficient.”. In other words the Flyweight pattern explains how objects can be distributed so that they can be used without restrictive costs in fine granules. In software development each ” flyweight ” object is categorised into two parts: the (extrinsic) state – dependent part and the (intrinsic) state – independent part. So, the intrinsic state of the Flyweight object is stored (shared) whereas extrinsic state is stored or computed by user objects and invoked to the Flyweight object.

- **Why do we care for number of objects in our program?**
  - Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
  - Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

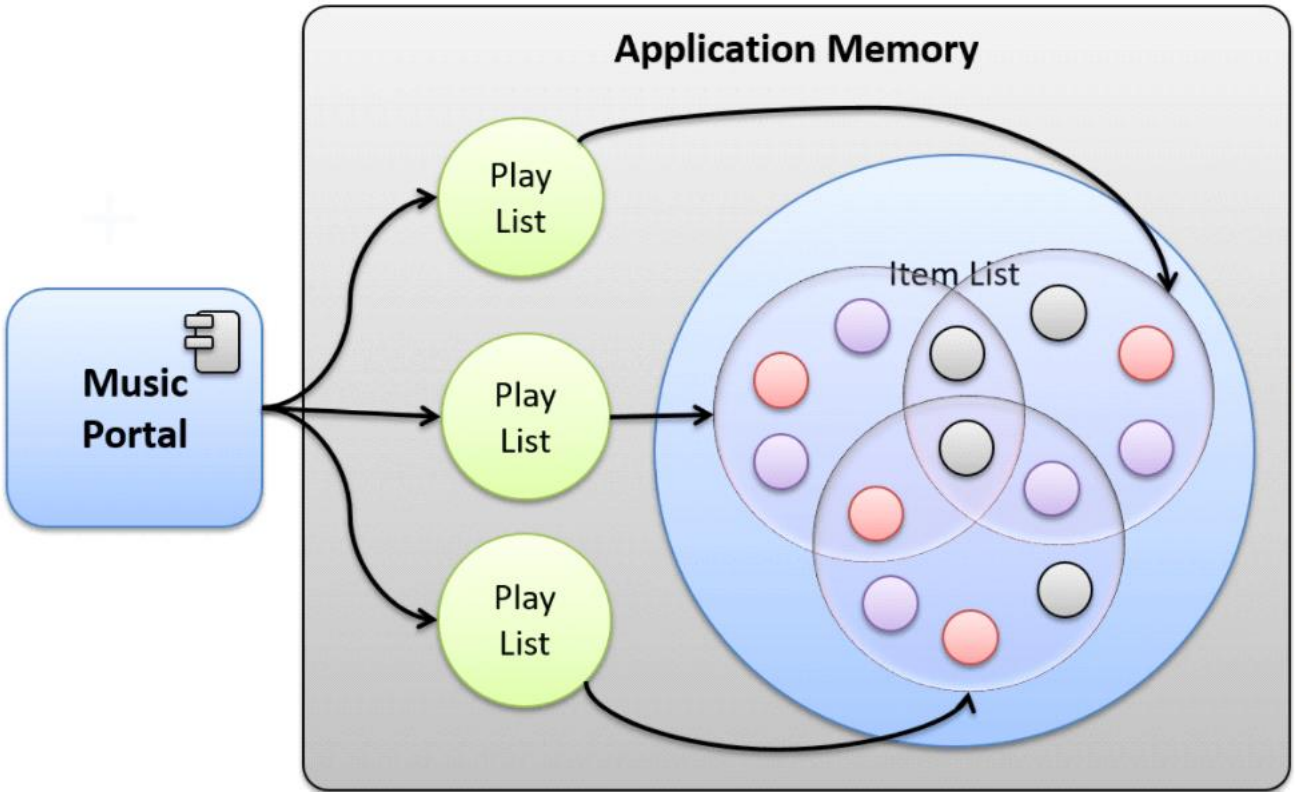
- **In Flyweight pattern we use a HashMap that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.**

- **It's very important that the flyweight objects are immutable: any operation on the state must be performed by the factory.**

- In programming, we can see java.lang.String constants as flyweight objects. All strings are stored in string pool and if we need a string with certain content then runtime return the reference to already existing string constant from the pool – if available.
- In browsers, we can use an image in multiple places in a webpage. Browsers will load the image only one time, and for other times browsers will reuse the image from cache. Now image is same but used in multiple places. It’s URL is intrinsic attribute because it’s fixed and shareable. Images position coordinates, height and width are extrinsic attributes which vary according to place (context) where they have to be rendered.

- **Difference between singleton pattern and flyweight pattern**
  - The singleton pattern helps we maintain only one object in the system. In other words, once the required object is created, we cannot create more. We need to reuse the existing object in all parts of the application.
  - The flyweight pattern is used when we have to create large number of similar objects which are different based on client provided extrinsic attribute

- Real-world example
  - By implementing the Flyweight design pattern, we are going create an application for managing playlists. Some songs must be shared between various playlists, and some songs must have shared sections between them in order to save memory space.
  - Let's take a real world example of ticket booking system. This system issues large number of tickets to passengers per day. Tickets can be of two types "ADULT" and "INFANT". In ticket booking system, ticket objects doesn't have much functionalities except printing itself. Passenger name and fare can vary depending on the ticket type. We can use flyweight design pattern in this scenario as every ticket of a particular type(let's say adult) is almost similar except it's extrinsic state(Name and fare) which can be passed to ticket object on runtime.



- **The goal of the flyweight pattern is to reduce memory usage by sharing as much data as possible**

- The Flyweight object essentially has two different kind of attributes –
  - **Intrinsic** - An **intrinsic** (invariant) state attribute is stored and shared in the **flyweight** object. It is independent of *flyweight's context*. So, as the best practice we should make intrinsic states **immutable**.
  - **Extrinsic** - An **extrinsic** (variant) state attribute does not store and share in the flyweight object because it depends on *flyweight's context* and varies as context change. Generally, we store and maintain the extrinsic state in the Client objects. We need to pass this extrinsic state to the **flyweight** object for object creation and processing.

State Design Pattern

15 December 2021 06:15

- a state allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.

- The **State Design Pattern** allows us to change the behavior of an object based on it's current state. Sometimes we want a class to behave differently in different state, this pattern allows us to abstract out state specific code form context object and model it as a composition of state object. For any state specific behavior, context object delegates control to state object. All state classes must implement State interface that declares methods representing the behaviors of a particular state.

- **When to use state pattern**

  - In any application, when we are dealing with an object which can be in different states during it's life-cycle and how it processes incoming requests (or make state transitions) based on it's present state – we can use the state pattern.
  - If we do not use the state pattern in such case, we will end up having lots of if-else statements which make the code base ugly, unnecessarily complex and hard to maintain. State pattern allows the objects to behave differently based on the current state, and we can define state-specific behaviors within different classes

- **Real world example of state pattern**

  - To make things simple, let's visualize a TV box operated with remote controller. We can change the state of TV by pressing buttons on remote. But the state of TV will change or not, it depends on the current state of the TV. If TV is ON, we can switch it OFF, mute or change aspects and source. But if TV is OFF, nothing will happen when we press remote buttons.
    For a switched OFF TV. only possible next state can be switch ON.
  - State patterns are used to implement state machine implementations in complex applications.
  - Another example can be of Java thread states. A thread can be one of its five states during it's life cycle. It's next state can be determined only after getting it's current state. e.g. we can not start a stopped thread or we cannot a make a thread wait, until it has started running

- **The state pattern is set to solve two main problems**
  - An object should change its behavior when its internal state changes.
  - State-specific behavior should be defined independently. That is, adding new states should not affect the behavior of existing states

- Define separate (state) objects that encapsulate state-specific behavior for each state. That is, define an interface (state) for performing state-specific behavior, and define classes that implement the interface for each state.
- A class delegates state-specific behavior to its current state object instead of implementing state-specific behavior directly.

Composite Design Pattern

15 December 2021 08:51

- The composite pattern describes a group of objects that are treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.
- The Composite design pattern allows us to arrange objects into tree structures to represent part-whole hierarchies. It allows client to treat individual objects as well as collection of objects in similar manner. For client, the interface to interact with individual objects and composites remains the same, client doesn't require different code for handling individual or collection of objects.
- Composite pattern comes under *structural design pattern* as it provide one of the best ways to arrange similar objects in hierarchical order. A composite object may contains a collection object, where each object itself can be composite.

- **Advantages of Composite Pattern**
  - *It provides a hierarchical tree structure to represent composite and individual objects.*
  - *It allows clients treat individual objects and compositions of objects uniformly.*
  - *We can change the structure of tree any time by calling methods of composite objects like addNode, removeNode etc.*

Visitor Design Pattern

17 December 2021 06:20

- the **visitor** design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the open/closed principle.

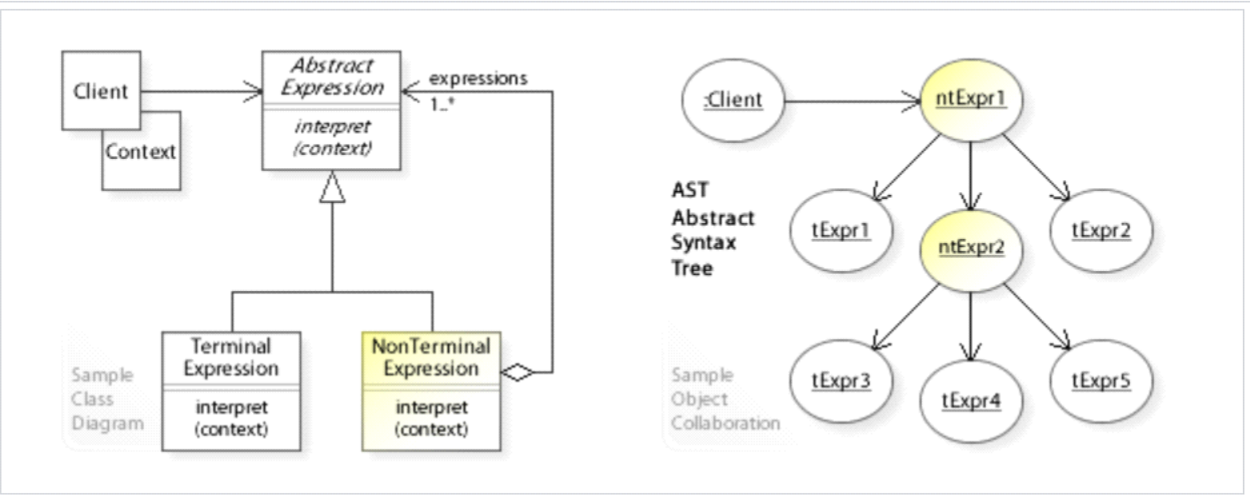
- want a hierarchy of objects to modify their behavior but without modifying their source code
- Above design flexibility allows to add methods to any object hierarchy without modifying the code written for hierarchy

- Example:
- Let’s consider an example of school were many small children's are studying and one day the school management decides for the health check-up of students and the school management approaches a child specialist doctor. After a couple of days doctor visit the school for regular check-up of students.

Interpreter Design Pattern

21 December 2021 06:03

the **interpreter pattern** is a design pattern that specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language



A sample UML class and object diagram for the Interpreter design pattern.[4]

Interpreter Design Pattern in JDK

- java.util.Pattern
- java.text.Normalizer
- java.text.Format

Summary

23 December 2021 08:01

S. No	Design Pattern Name	Example to Remember
1	Factory	Subclass will be created based on client input. Client doesn't know which sub class object and method is called. Example: Application with Oracle and My sql, Based on client input the sub class DB object gets created
2	Decorator	Dynamically add the behavior to the individual object without modifying the other objects of the class. Example: Simple coffee price is 5$, if coffee is made of water for this object alone we can change the price to 10$.
3	Template Design Pattern	We use this to perform some algorithm for every implementation then we go for this. Ex: if the insurance product has some certain checks, in order to perform those checks we can create abstract methods and each product class can implement those, and build method can provide the corresponding object.
4	Prototype	In order to create the object every time, clone can be used to create ethe object from main object Deep cloning to be suggested for this
5	Memento	To get the CTRL Z functionality. To restore the object state we go for this design pattern. Any feedback, complaint form can be developed using this pattern
6	Builder	Instead of creating the object using constructor, we go for this. Advantage is if constructor has multiple elements and if some are optional, then we have to create multiple constructors and creating object becomes difficult so using this, and we will create inner class, and using the build method object of outer class is created String Buffer or String Builder
7	Singleton	In order to have only single instance of an object throughout the application we go for this Runtime is the example
8	Flyweight	In flyweight, we can have only single instance of the object and if needed, we can create the multiple objects for the same class
9	Adapter	In order to provide the compatibility between two different interfaces, we will go for adapter. In the insurance application, if we want to change the payment gateway then instead of changing the objects or methods in the entire application, we create adapter class which converts the existing class object to the new payment gateway object.
10	Chain of Responsibility	JVM follows this design pattern. Object delegation will happened from one class to another. Example is ATM, while withdraw the amount, input request is being transferred from 2000 notes to 500 notes and then to 100
11	Observer	In order to perform event driven architecture we use this pattern. For insurance company whenever the users are subscribed then any notification or information shared by company automatically received by all the subscribers Similar to twitter tweets (if a person keeps the tweet that will be received by all the followers)
12	Abstract Factory	Factory on top of factory. In order to implement some application behavior which provides the accessibility at global level then we use this. Example: Car factory provides the services in US, India and internally that car factory provides different models (here we have 2 factories)
13	Strategy	this enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use. Example: Implemented the functionality of loan rates for privileged customer and non-privileged customer. If we want to implement for employee then just one class addition is enough

Interview Questions

02 December 2021 08:11

- **What is Singleton Design Patten ?**
- **how many ways can you create singleton pattern? (eager and lazy loading)**
- **which classes in JDK uses singleton pattern ?**
- **What is lazy loading and eager loading ?**
- **Why to keep the instance variable as volatile in Singleton design Pattern ?**
- **What is double checked locking ?**
- **How to break the singleton using reflection and how to fix it ?**
- **How to break the deserialization using reflection and how to fix it ?**
- **How to break the Cloning using reflection and how to fix it ?**
- **How to break the multi-Threading using reflection and how to fix it ?**

- **When to use Singleton Design Pattern ? (what are some examples where you can use singleton design pattern ?)**
- **When will you prefer to use a Factory Pattern?**
  (create the object based on the input parameter, client should not know underlying logic of encapsulation framework)
  a class does not know which class of objects it must create. factory pattern can be used where we need to create an object of any one of sub-classes depending on the data provided

- **When to use builder design pattern ?**
- **What are the examples of Builder design pattern classes in Java ?**
- **Which design pattern is used to get a way to access the elements of a collection object in sequential manner?**
- **which design pattern will be helpful to add new functionality to an existing object?**
  Decorator
- **What are the advantages of builder design patterns?**
- **How is the bridge pattern different from the adapter pattern?**
- **Difference between Factory and Abstract Factory Design Pattern?**
- **When to Use Proxy Design Pattern ?**
  Control access to another object
  Lazy initialization
  Implement logging

Facilitate network connection
Count references to an object

- **When to use Decorator pattern?**

  Use Decorator pattern when
  To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects

- **JDK examples which are using Decorator design pattern ?**
- **Difference between strategy and state design pattern in Java?**
- **What is the benefit of using a prototype design pattern over creating an instance using the new keyword?**

- **What is Observer design pattern?**
- **Which design pattern can be used when to decouple abstraction from the implement**
- **How Decorator design pattern is different from Proxy pattern?**
  - Main differences between Decorator and Proxy design pattern are:
    - Decorator provides an enhanced interface after decorating it with additional features. Proxy provides same interface since it is just acting as a proxy to another object.
    - Decorator is a type of Composite pattern with only one component. But each decorator can add additional features. Since it is one component in Decorator, there is no object aggregation.
    - Proxy can also provide performance improvement by lazy loading. There is nothing like this available in Decorator.
    - Decorator follows recursive composition. Proxy is just one object to another object access.
    - Decorator is mostly used for building a variety of objects. Proxy is mainly used for access to another object.

- **What are the examples of Proxy design pattern in JDK?**

  - In JDK there are many places where Proxy design pattern is used. Some of these are as follows:
    - java.lang.reflect.Proxy
    - java.rmi.*
    - javax.inject.Inject
    - javax.ejb.EJB
    - javax.persistence.PersistenceContext

- **What are the examples of Chain of Responsibility design pattern in JDK?**

  - javax.servlet.Filter.doFilter(): In the Filter class, the Container calls the doFilter method when a request/response pair is passed through the

chain. With filter the request reaches to the appropriate resource at the end of the chain. We can pass FilterChain in doFilter() method to allow the Filter to pass on the request and response to the next level in the chain.

➢ How is the bridge pattern different from the adapter pattern?

main difference between the two is that using a bridge pattern separates the essential data from its implementation (CHANGES DONE IN ABSTRACTION LAYER DOESN'T HAVE THE IMPACT IN IMPLEMENTATION LAYER). In contrast, an adapter pattern allows incompatible classes to an interface without changing the source code

➢ Which are all the design pattern can be used to design ATM Machine ?

➢ What is Memento Design pattern ?

This Design pattern is used when we wan restore the object state without disturbing the encapsulation

➢ What is façade and how it is different from singleton ?

➢ When to use Template Design Pattern ?

When the steps to solve an algorithm is fixed and if we can create a template of steps which subclass can implement based on their need, we can use template method design pattern.

In this pattern the template method itself should be final, so that subclass cannot override it and change the steps but if needed they can be made abstract so that the subclass can implement them based on the need.

➢ What are the main uses of Command design pattern?

Command design pattern is a behavioral design pattern. We use it to encapsulate all the information required to trigger an event. Some of the main uses of Command pattern are:

- I. Graphic User Interface (GUI): In GUI and menu items, we use command pattern. By clicking a button we can read the current information of GUI and take an action.
- II. Macro Recording: If each of user action is implemented as a separate Command, we can record all the user actions in a Macro as a series of Commands. We can use this series to implement the “Playback” feature. In this way, Macro can keep on doing same set of actions with each replay.
- III. Multi-step Undo: When each step is recorded as a Command, we can use it to implement Undo feature in which each step can by undo. It is used in text editors like MS-Word.
- IV. Networking: We can also send a complete Command over the network to a remote machine where all the actions encapsulated within a Command are executed.
- V. Progress Bar: We can implement an installation routine as a series of Commands. Each Command provides the estimate time. When we execute the installation routine, with each command we can display the progress bar.
- VI. Wizard: In a wizard flow we can implement steps as Commands. Each step may have complex task that is just implemented within one command.
- VII. Transactions: In a transactional behavior code there are multiple tasks/updates. When all the tasks are done then only transaction is committed. Else we have to rollback

the transaction. In such a scenario each step is implemented as separate Command.

Q. What are the examples of Command design pattern in JDK?

In JDK there are many places where Command design pattern is used. Some of these are as follows:

- All implementations of java.lang.Runnable
- All implementations of javax.swing.Action