**DESIGN PATTERNS**

1. Creational Design Patterns

Creational design patterns deals with creation of object. There might be need of creating object using other objects.

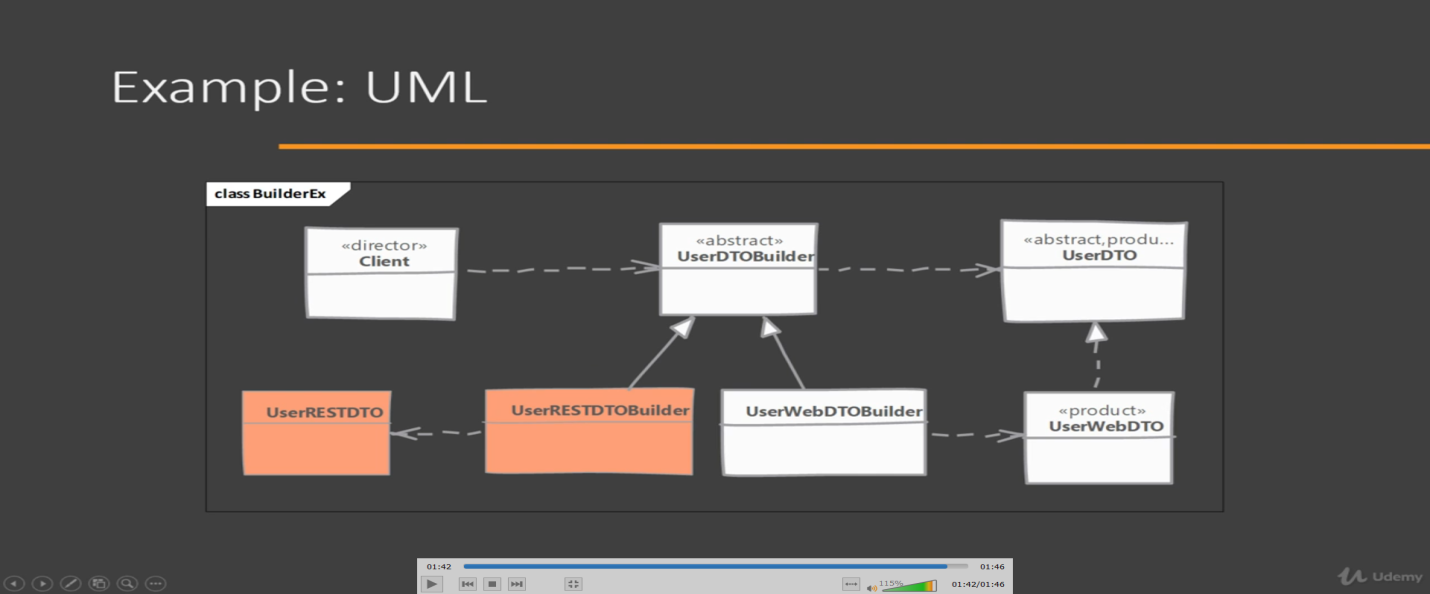
Various Creation design patterns are as follows:

* Builder
* Simple Factory
* Factory Method
* Prototype
* Singleton
* Abstract Factory
* Object Pool

1. Builder

When we have a complex process to construct an object involving multiple steps, then builder design pattern can help us.

Builder pattern should provide a method to construct part of object and a method to provide/get final constructed object.



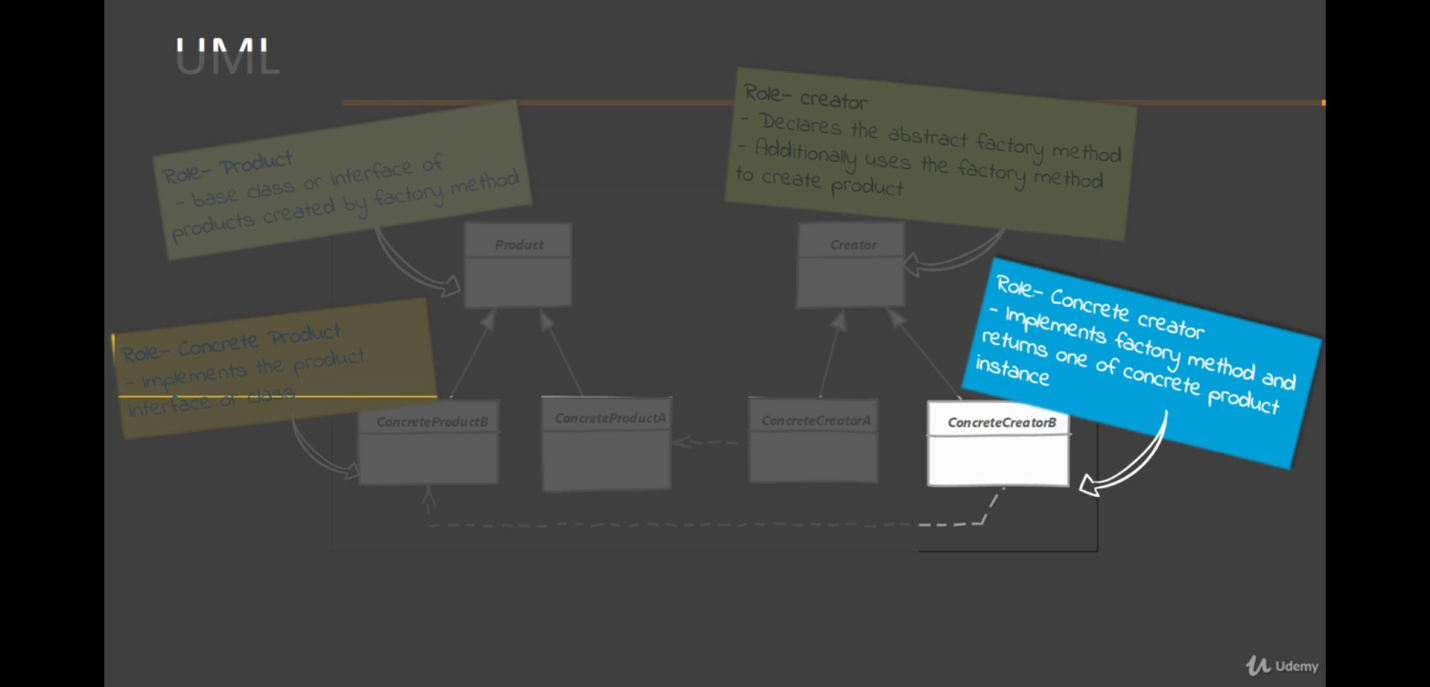
1. Simple Factory

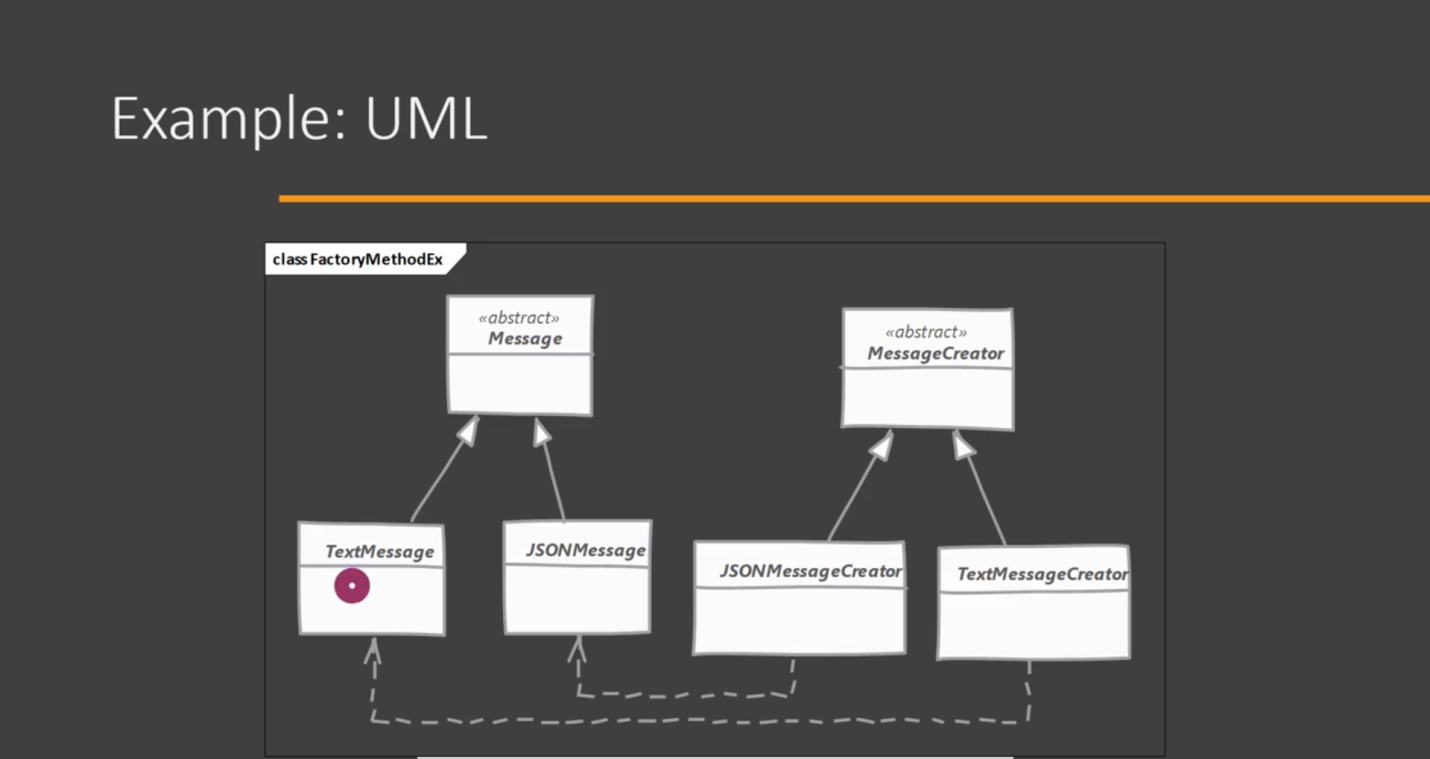
In Simple factory, we will create a class and have a static method which accepts 1 argument and based on argument value, it will create object of intended class. We are creating static method because we don’t need state tracking.

Pitfall: The criteria used by simple factory to decide which object to instantiate can get more complex over time. If you find yourself in such situation then use factory method design pattern.

1. Factory Method

When we want to move the object creation logic to separate class.We use this pattern when we do not know in advance which class we may need to instantiate beforehand & also to allow new classes to be added to system and handle their creation without affecting client code. We let subclasses decide which object to instantiate by overriding the factory method.





Pitfalls: More complex to implement. More classes to involved and need unit testing. You have to start with Factory Method design pattern from beginning. It’s not easy to re factor existing code into factory method pattern. Sometime this pattern forces you to subclass just to create appropriate instance.

1. Prototype Design Pattern

Prototype Design pattern can be used where cost to create an object is high. We create one object and then create copy from existing object.

1. Abstract Factory Method

Abstract Factory provides interfaces for creating families of related or dependent objects without specifying their concrete classes.

1. Singleton Design Pattern

In this article, we will see that what are various concepts which can break singleton property of a class and how to avoid them. There are mainly 3 concepts which can break singleton property of a class. Let’s discuss them one by one.

1. **Reflection:** [Reflection](https://www.geeksforgeeks.org/reflection-in-java/) can be caused to destroy singleton property of singleton class, as shown in following example:

|  |
| --- |
| // Java code to explain effect of Reflection  // on Singleton property    import java.lang.reflect.Constructor;    // Singleton class  class Singleton  {      // public instance initialized when loading the class      public static Singleton instance = new Singleton();        private Singleton()      {          // private constructor      }  }    public class GFG  {        public static void main(String[] args)      {          Singleton instance1 = Singleton.instance;          Singleton instance2 = null;          try          {              Constructor[] constructors =                      Singleton.class.getDeclaredConstructors();              for (Constructor constructor : constructors)              {                  // Below code will destroy the singleton pattern                  constructor.setAccessible(true);                  instance2 = (Singleton) constructor.newInstance();                  break;              }          }            catch (Exception e)          {              e.printStackTrace();          }        System.out.println("instance1.hashCode():- "                                        + instance1.hashCode());      System.out.println("instance2.hashCode():- "                                        + instance2.hashCode());      }  } |

1. Output:-
2. instance1.hashCode():- 366712642
3. instance2.hashCode():- 1829164700
4. After running this class, you will see that hashCodes are different that means, 2 objects of same class are created and singleton pattern has been destroyed.
5. **Overcome reflection issue:** To overcome issue raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

|  |
| --- |
| //Java program for Enum type singleton  public enum Singleton  {    INSTANCE;  } |

1. As enums don’t have any constructor so it is not possible for Reflection to utilize it. Enums have their by-default constructor, we can’t invoke them by ourself.**JVM handles the creation and invocation of enum constructors internally.** As enums don’t give their constructor definition to the program, it is not possible for us to access them by Reflection also. Hence, reflection can’t break singleton property in case of enums.
2. **Serialization:-** [Serialization](https://www.geeksforgeeks.org/serialization-in-java/) can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of byte stream and save in a file or send over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

|  |
| --- |
| // Java code to explain effect of  // Serilization on singleton classes  import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.ObjectInput;  import java.io.ObjectInputStream;  import java.io.ObjectOutput;  import java.io.ObjectOutputStream;  import java.io.Serializable;    class Singleton implements Serializable  {      // public instance initialized when loading the class      public static Singleton instance = new Singleton();        private Singleton()      {          // private constructor      }  }      public class GFG  {        public static void main(String[] args)      {          try          {              Singleton instance1 = Singleton.instance;              ObjectOutput out                  = new ObjectOutputStream(new FileOutputStream("file.text"));              out.writeObject(instance1);              out.close();                // deserailize from file to object              ObjectInput in                  = new ObjectInputStream(new FileInputStream("file.text"));                Singleton instance2 = (Singleton) in.readObject();              in.close();                System.out.println("instance1 hashCode:- "                                                   + instance1.hashCode());              System.out.println("instance2 hashCode:- "                                                   + instance2.hashCode());          }            catch (Exception e)          {              e.printStackTrace();          }      }  } |

1. Output:-
2. instance1 hashCode:- 1550089733
3. instance2 hashCode:- 865113938
4. As you can see, hashCode of both instances is different, hence there are 2 objects of a singleton class. Thus, the class is no more singleton.
5. **Overcome serialization issue:-** To overcome this issue, we have to implement method readResolve() method.

|  |
| --- |
| // Java code to remove the effect of  // Serialization on singleton classes  import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.ObjectInput;  import java.io.ObjectInputStream;  import java.io.ObjectOutput;  import java.io.ObjectOutputStream;  import java.io.Serializable;    class Singleton implements Serializable  {      // public instance initialized when loading the class      public static Singleton instance = new Singleton();        private Singleton()      {          // private constructor      }        // implement readResolve method      protected Object readResolve()      {          return instance;      }  }    public class GFG  {        public static void main(String[] args)      {          try          {              Singleton instance1 = Singleton.instance;              ObjectOutput out                  = new ObjectOutputStream(new FileOutputStream("file.text"));              out.writeObject(instance1);              out.close();                // deserailize from file to object              ObjectInput in                  = new ObjectInputStream(new FileInputStream("file.text"));              Singleton instance2 = (Singleton) in.readObject();              in.close();                System.out.println("instance1 hashCode:- "                                             + instance1.hashCode());              System.out.println("instance2 hashCode:- "                                             + instance2.hashCode());          }            catch (Exception e)          {              e.printStackTrace();          }      }  } |

1. Output:-
2. instance1 hashCode:- 1550089733
3. instance2 hashCode:- 1550089733
4. Above both hashcodes are same hence no other instance is created.
5. **Cloning:** [Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is a concept to create duplicate objects. Using clone we can create copy of object. Suppose, we ceate clone of a singleton object, then it wil create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

|  |
| --- |
| // JAVA code to explain cloning  // issue with singleton  class SuperClass implements Cloneable  {    int i = 10;      @Override    protected Object clone() throws CloneNotSupportedException    {      return super.clone();    }  }    // Singleton class  class Singleton extends SuperClass  {    // public instance initialized when loading the class    public static Singleton instance = new Singleton();      private Singleton()    {      // private constructor    }  }    public class GFG  {    public static void main(String[] args) throws CloneNotSupportedException    {      Singleton instance1 = Singleton.instance;      Singleton instance2 = (Singleton) instance1.clone();      System.out.println("instance1 hashCode:- "                             + instance1.hashCode());      System.out.println("instance2 hashCode:- "                             + instance2.hashCode());    }  } |

1. Output :-
2. instance1 hashCode:- 366712642
3. instance2 hashCode:- 1829164700
4. Two different hashCode means there are 2 different objects of singleton class.
5. **Overcome Cloning issue:-** To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now whenever user will try to create clone of singleton object, it will throw exception and hence our class remains singleton.

|  |
| --- |
| // JAVA code to explain overcome  // cloning issue with singleton  class SuperClass implements Cloneable  {    int i = 10;      @Override    protected Object clone() throws CloneNotSupportedException    {      return super.clone();    }  }    // Singleton class  class Singleton extends SuperClass  {    // public instance initialized when loading the class    public static Singleton instance = new Singleton();      private Singleton()    {      // private constructor    }      @Override    protected Object clone() throws CloneNotSupportedException    {      throw new CloneNotSupportedException();    }  }    public class GFG  {    public static void main(String[] args) throws CloneNotSupportedException    {      Singleton instance1 = Singleton.instance;      Singleton instance2 = (Singleton) instance1.clone();      System.out.println("instance1 hashCode:- "                           + instance1.hashCode());      System.out.println("instance2 hashCode:- "                           + instance2.hashCode());    }  } |

1. Output:-
2. Exception in thread "main" java.lang.CloneNotSupportedException
3. at GFG.Singleton.clone(GFG.java:29)
4. at GFG.GFG.main(GFG.java:38)
5. Now we have stopped user to create clone of singleton class. If you don;t want to throw exception you can also return the same instance from clone method.

|  |
| --- |
| // JAVA code to explain overcome  // cloning issue with singleton  class SuperClass implements Cloneable  {    int i = 10;      @Override    protected Object clone() throws CloneNotSupportedException    {      return super.clone();    }  }    // Singleton class  class Singleton extends SuperClass  {    // public instance initialized when loading the class    public static Singleton instance = new Singleton();      private Singleton()    {      // private constructor    }      @Override    protected Object clone() throws CloneNotSupportedException    {      return instance;    }  }    public class GFG  {    public static void main(String[] args) throws CloneNotSupportedException    {      Singleton instance1 = Singleton.instance;      Singleton instance2 = (Singleton) instance1.clone();      System.out.println("instance1 hashCode:- "                             + instance1.hashCode());      System.out.println("instance2 hashCode:- "                             + instance2.hashCode());    }  } |

1. Output:-
2. instance1 hashCode:- 366712642
3. instance2 hashCode:- 366712642
4. Now, as hashcode of both the instances is same that means they represent a single instance.
5. Structural Design Pattern

Structural Design Pattern deals with how classes and objects are arranged or composed.

* Adapter
* Bridge
* Decorator
* Composite
* Façade
* Flyweight
* Proxy

1. Adaptor Design Pattern

This pattern is easy to understand as the real world is full of adapters.   For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

To use an adapter:

The client makes a request to the adapter by calling a method on it using the target interface.

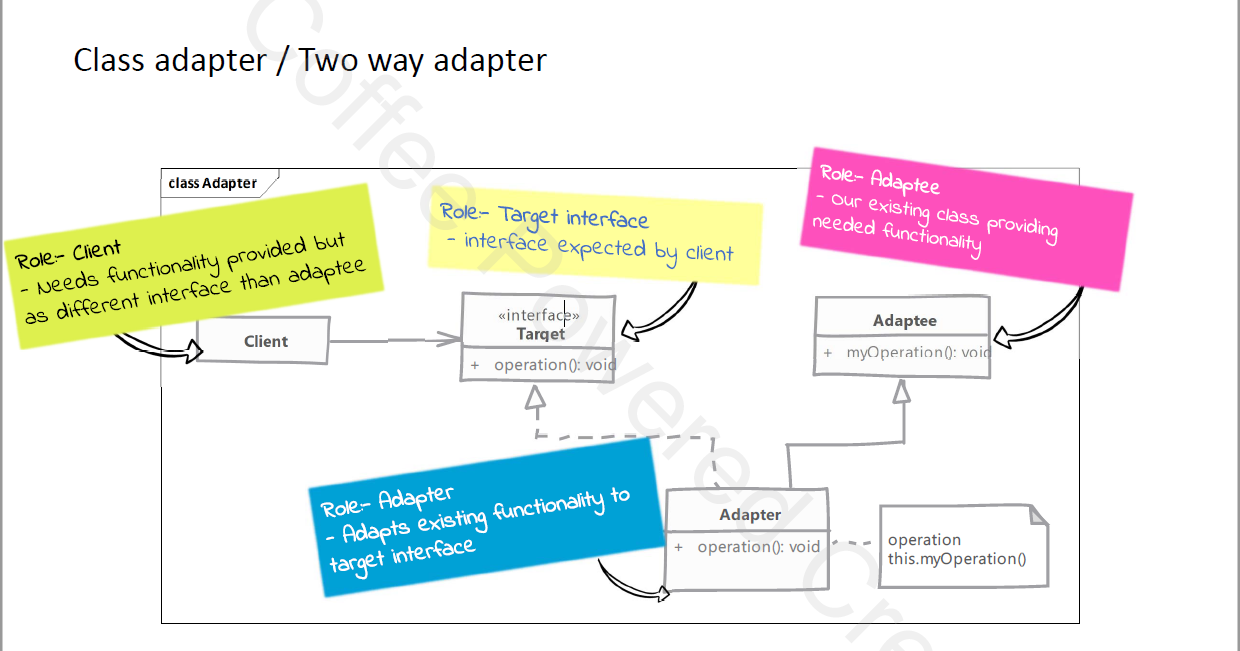
The adapter translates that request on the adaptee using the adaptee interface.

Client receive the results of the call and is unaware of adapter’s presence.

We have an existing object which provides the functionality that client needs. But Client code can’t use this object because it expects an object with different interface.

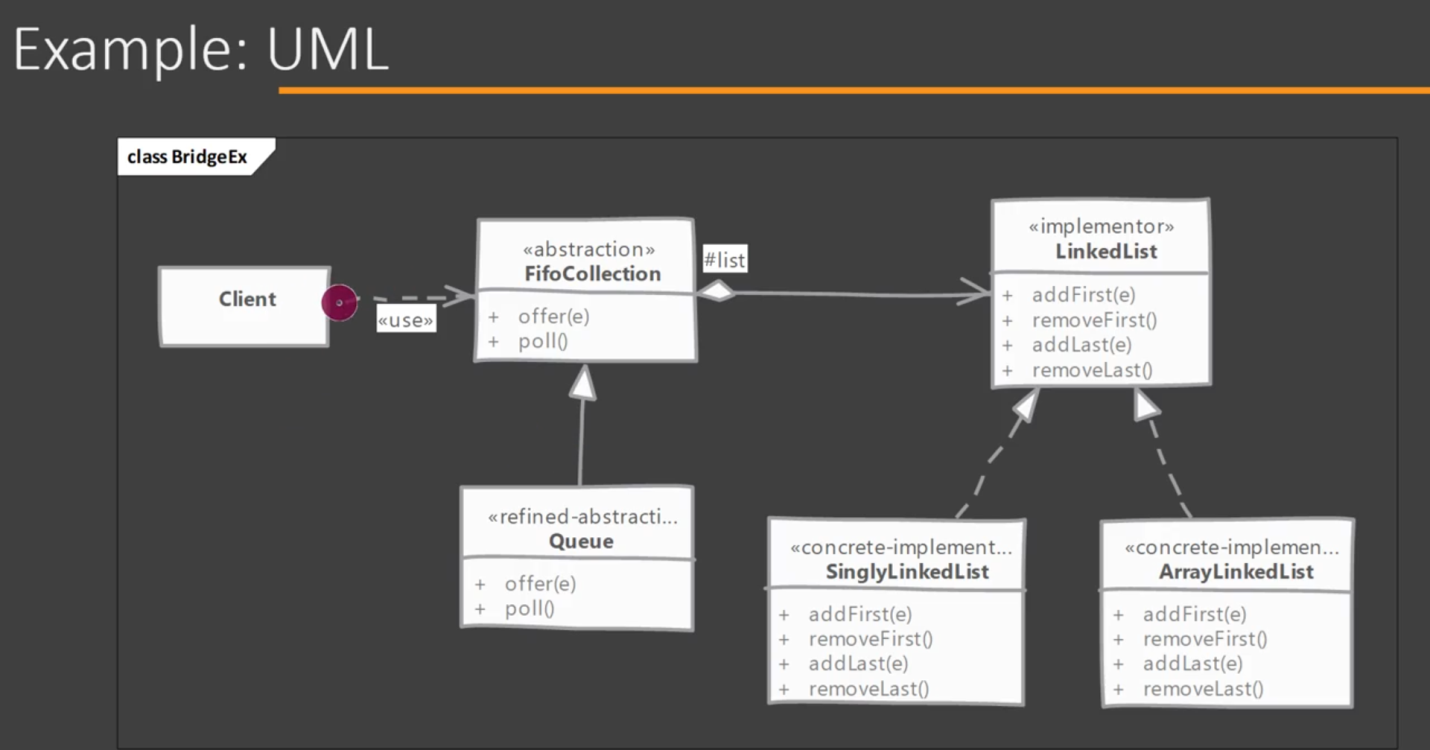
Using Adaptor design pattern we make this existing object work with client by adapting the object to client’s expected interface.

This pattern is also called as wrapper as it “wraps” existing object.



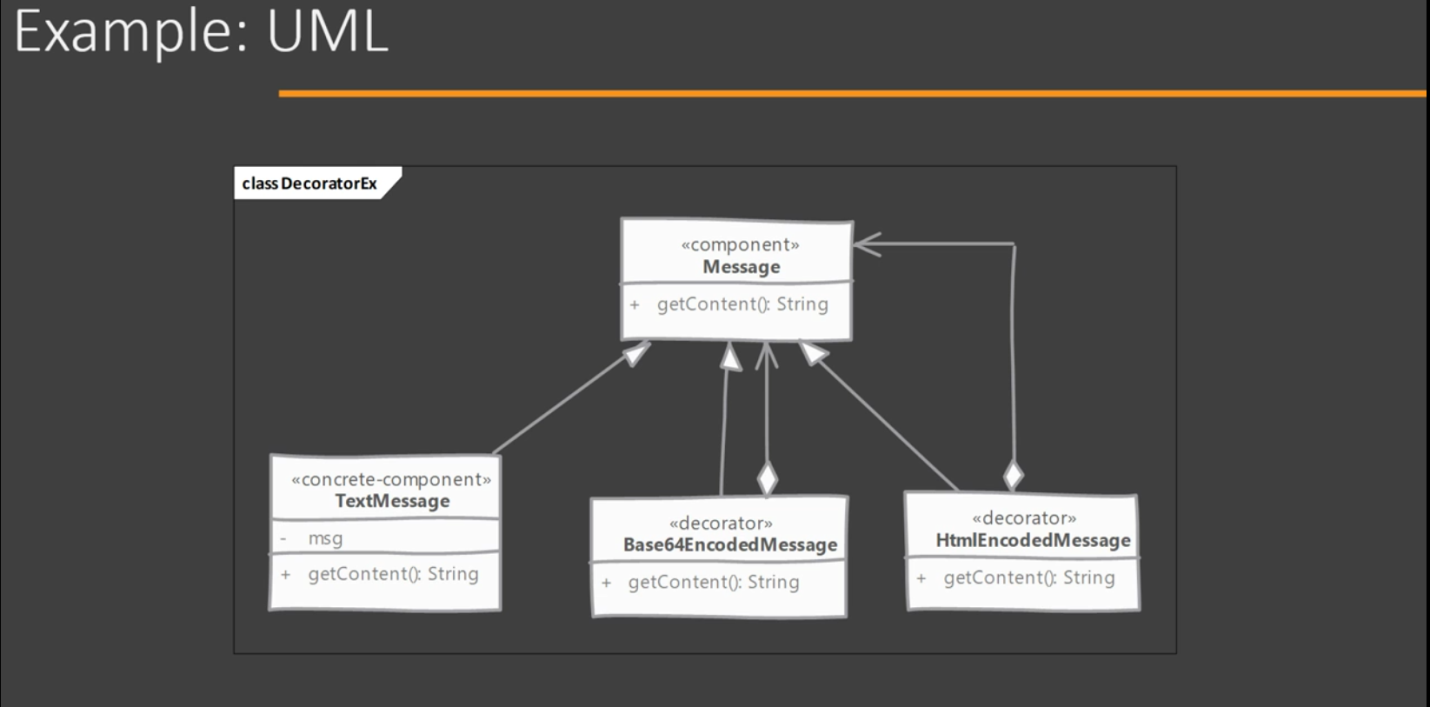
1. Bridge Design Pattern

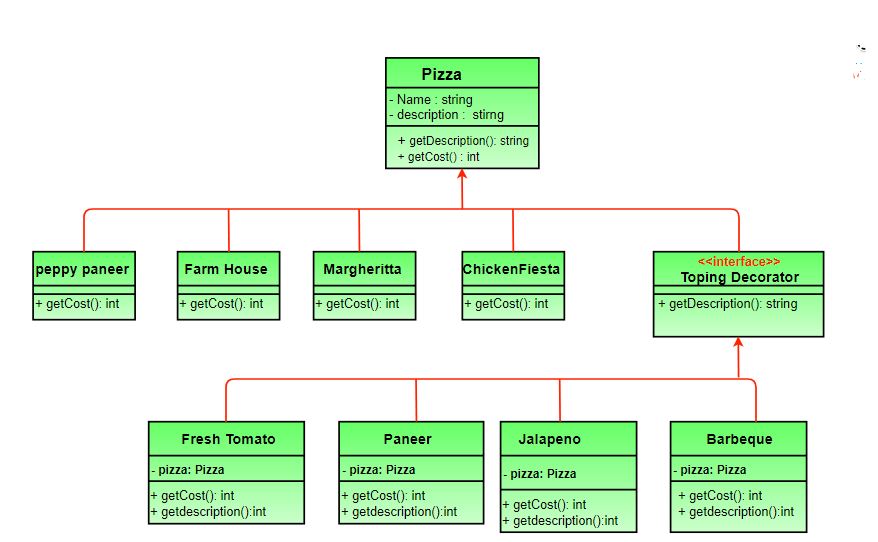
Our implementation and abstraction are generally coupled to each other in normal inheritance. Using bridge pattern we can decouple them so they can both change without affecting each other. We achieve this feat by creating two separate inheritance hierarchy, one for implementation and another for abstraction. We use composition to bridge these two hierarchies.



1. Decorator Design Pattern

When we want to enhance behaviour of our existing object dynamically as and when required then we can use decorator design patter. Decorator wraps an object within itself and provide same interface as the wrapped object. So the client of original object doesn’t need to change. A decorator provides alternative to sub classing for extending functionality of existing classes.





1. Composite Design Pattern

We have a part whole relationship or hierarchy of objects and we want to be able to treat all objects in this hierarchy uniformly. This is NOT simple composition from object oriented programming but a further enhancement to that principal. Think of composite pattern when dealing with tree structure of objects.

1. Facade Design Pattern

Facade is a part of Gang of Four design pattern and it is categorized under Structural design patterns. Before we dig into the details of it, let us discuss some examples which will be solved by this particular Pattern.

So, As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

**More examples**

In Java, 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.

Another good example can be the startup of a computer. When a computer starts up, it involves the work of cpu, memory, hard drive, etc. To make it easy to use for users, we can add a facade which wrap the complexity of the task, and provide one simple interface instead.  
Same goes for the **Facade Design Pattern**. It hides the complexities of the system and provides an interface to the client from where the client can access the system.

The facade pattern is appropriate when you have a **complex system**that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

1. Flyweight Design Pattern

Flyweight pattern is one of the [structural design patterns](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/) as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 105). One important feature of flyweight objects is that they are immutable. This means that they cannot be modified once they have been constructed.

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.](https://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html)

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](https://www.geeksforgeeks.org/hashmap-treemap-java/) 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.

Intrinsic and Extrinsic States

To understand Intrinsic and Extrinsic state, let us consider an example.

Suppose in a text editor when we enter a character, an object of Character class is created, the attributes of the Character class are {name, font, size}. We do not need to create an object every time client enters a character since letter ‘B’ is no different from another ‘B’ . If client again types a ‘B’ we simply return the object which we have already created before. Now all these are intrinsic states (name, font, size), since they can be shared among the different objects as they are similar to each other.

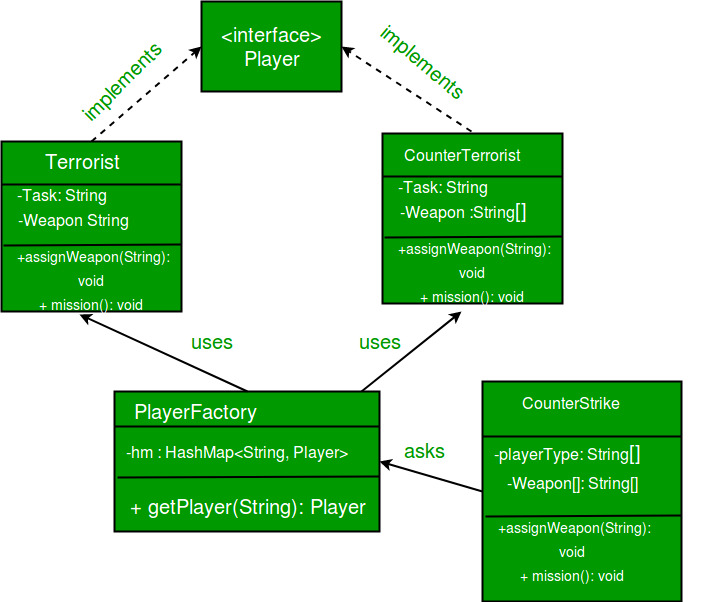
Now we add to more attributes to the Character class, they are row and column. They specify the position of a character in the document. Now these attributes will not be similar even for same characters, since no two characters will have the same position in a document, these states are termed as extrinsic states, and they can’t be shared among objects.

Implementation : We implement the creation of Terrorists and Counter Terrorists In the game of [Counter Strike](https://en.wikipedia.org/wiki/Counter-Strike). So we have 2 classes one for **T**errorist(**T**) and other for **C**ounter **T**errorist(**CT**). Whenever a player asks for a weapon we assign him the asked weapon. In the mission, terrorist’s task is to plant a bomb while the counter terrorists have to diffuse the bomb.

Why to use Flyweight Design Pattern in this example? Here we use the Fly Weight design pattern, since here we need to reduce the object count for players. Now we have n number of players playing CS 1.6, if we do not follow the Fly Weight Design Pattern then we will have to create n number of objects, one for each player. But now we will only have to create 2 objects one for terrorists and other for counter terrorists, we will reuse then again and again whenever required.

Intrinsic State : Here ‘task’ is an intrinsic state for both types of players, since this is always same for T’s/CT’s. We can have some other states like their color or any other properties which are similar for all the Terrorists/Counter Terrorists in their respective Terrorists/Counter Terrorists class.

Extrinsic State : Weapon is an extrinsic state since each player can carry any weapon of his/her choice. Weapon need to be passed as a parameter by the client itself.



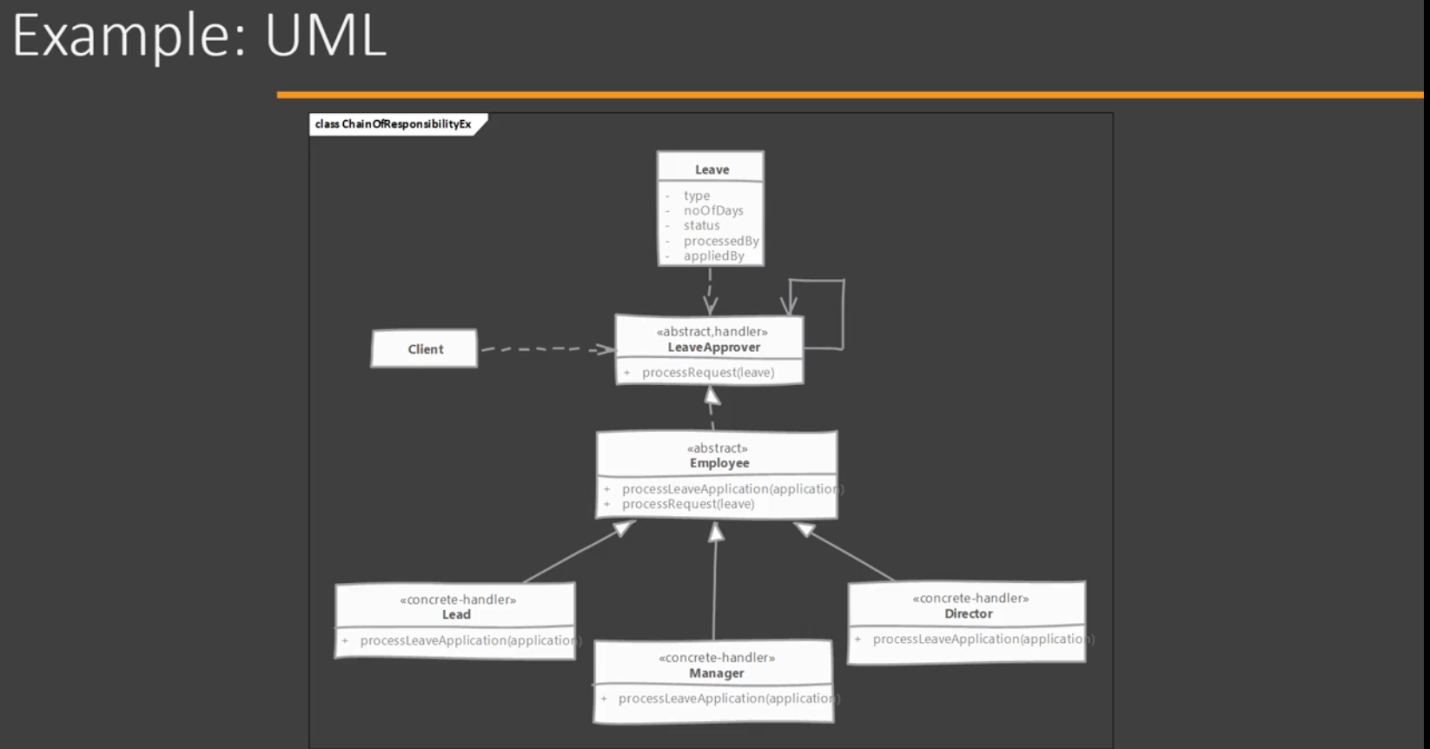
1. Behavioral Design Pattern

Behavioral design pattern describe how classes and objects interact and communicate with each other.

* Chain of Responsibility
* Command
* Interpreter
* Mediator
* Iterator
* Memento
* Observer
* State
* Strategy
* Template Method
* Visitor
* Null Object

1. Chain of Responsibility

We need to avoid coupling the code which sends request to the code which handles that request. Typically the code which wants some request handled calls the exact method on an exact object to process it, thus the tight coupling. Chain of Responsibility solves this problem by giving more than one object, chance to process the request. We create object which are chained together by one object knowing reference of object which is next in chain. We give request to first object in chain, if it can’t handle that it simply passes the request down the chain.



Chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

This pattern is recommended when multiple objects can handle a request and the handler doesn’t have to be a specific object. Also, the handler is determined at runtime. Please note that a request not handled at all by any handler is a valid use case.

https://media.geeksforgeeks.org/wp-content/uploads/desigmpatternuml1.png

Handler : This can be an interface which will primarily recieve the request and dispatches the request to chain of handlers. It has reference of only first handler in the chain and does not know anything about rest of the handlers.

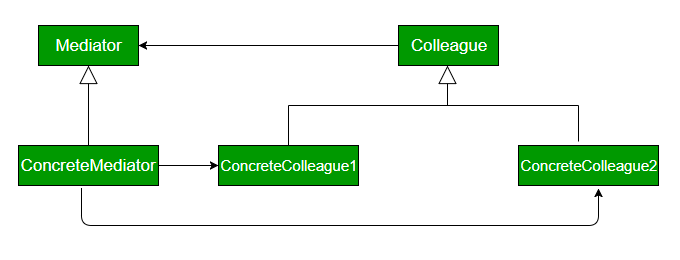
Concrete handlers : These are actual handlers of the request chained in some sequential order.

Client : Originator of request and this will access the handler to handle it.

1. Mediator Design Pattern

Mediator design pattern is one of the important and widely used behavioral design pattern. Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on providing a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it’s own logic to provide way of communication.



Mediator: It defines the interface for communication between colleague objects.

ConcreteMediator: It implements the mediator interface and coordinates communication between colleague objects.

Colleague: It defines the interface for communication with other colleagues

ConcreteColleague : It implements the colleague interface and communicates with other colleagues through its mediator

1. Iterator Design Pattern

Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation.  
Suppose we are building an application that requires us to maintain a list of notifications. Eventually, some part of your code will require to iterate over all notifications. If we implemented your collection of notifications as array you would iterate over them as:

// If a simple array is used to store notifications

for (int i = 0; i < notificationList.length; i++)

Notification notification = notificationList[i]);

// If ArrayList is Java is used, then we would iterate

// over them as:

for (int i = 0; i < notificationList.size(); i++)

Notification notification = (Notification)notificationList.get(i);

And if it were some other collection like set, tree etc. way of iterating would change slightly. Now, what if we build an iterator that provides a generic way of iterating over a collection independent of its type.

// Create an iterator

Iterator iterator = notificationList.createIterator();

// It wouldn’t matter if list is Array or ArrayList or

// anything else.

while (iterator.hasNext())

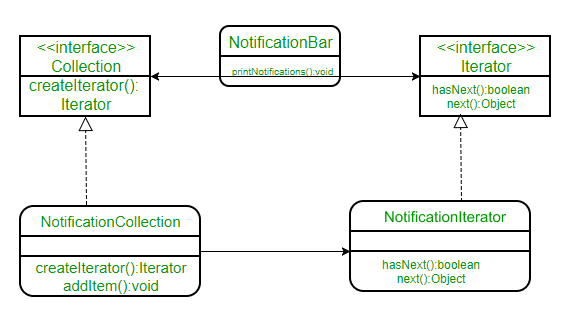
{

Notification notification = iterator.next());

}

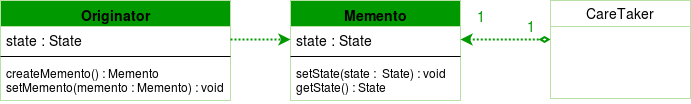
Iterator pattern lets us do just that. Formally it is defined as below:  
**The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying representation.**

Here we have a common interface Aggregate for client as it decouples it from the implementation of your collection of objects. The ConcreteAggregate implements createIterator() that returns iterator for its collection. Each ConcreteAggregate’s responsibility is to instantiate a ConcreteIterator that can iterate over its collection of objects. The iterator interface provides a set of methods for traversing or modifying the collection that is in addition to next()/hasNext() it can also provide functions for search, remove etc.  
Let’s understand this through an example. Suppose we are creating a notification bar in our application that displays all the notifications which are held in a notification collection. NotificationCollection provides an iterator to iterate over its elements without exposing how it has implemented the collection (array in this case) to the Client (NotificationBar).



1. Memento Design Pattern

Memento pattern is a behavioral design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.



Design components

originator : the object for which the state is to be saved. It creates the memento and uses it in future to undo.

memento : the object that is going to maintain the state of originator. Its just a POJO.

caretaker : the object that keeps track of multiple memento. Like maintaining savepoints.

A **Caretaker** would like to perform an operation on the **Originator** while having the possibility to rollback. The caretaker calls the **createMemento()** method on the originator asking the originator to pass it a memento object. At this point the originator creates a memento object saving its internal state and passes the memento to the caretaker. The caretaker maintains the memento object and performs the operation. In case of the need to undo the operation, the caretaker calls the **setMemento()** method on the originator passing the maintained memento object. The originator would accept the memento, using it to restore its previous state.