Java Design Patterns & Examples

Author: Pawan Modi

Table of Contents

INTRODUCTION 3

STRUCTURAL PATTERNS 4

ADAPTER PATTERN 5
BRIDGE PATTERN 8
COMPOSITE PATTERN 12
DECORATOR PATTERN 16
FACADE PATTERN (FACE OR FRONT APPEARANCE) 20
FLYWEIGHT PATTERN 23
PROXY PATTERN 27
SUMMARY OF STRUCTURAL PATTERNS 30

BEHAVIORAL PATTERNS 31

MEDIATOR PATTERN 32
MEMENTO PATTERN 37
INTERPRETER PATTERN 41
ITERATOR PATTERN 46
CHAIN-OF-RESPONSIBILITY PATTERN 50
COMMAND PATTERN 53
STATE PATTERN 58
STRATEGY PATTERN 60
OBSERVER PATTERN 64
TEMPLATE PATTERN 68
VISITOR PATTERN 70

CREATIONAL PATTERNS 73

ABSTRACT FACTORY PATTERN 74
BUILDER PATTERN 79
FACTORY PATTERN 84
PROTOTYPE PATTERN 87
SINGLETON PATTERN 90
SUMMARY OF CREATIONAL PATTERNS 92

Introduction

lets define what we mean by design and what we mean by pattern. According to me design is blue print (sketch) of something so it can be defined as creation of something in mind. Moving to pattern, we can define it as guideline, or something that repeats.

Design pattern is a widely accepted solution to a recurring design problem. A design pattern describes how to structure classes to meet a given requirement provides a general blueprint to follow when implementing part of a program does not describe how to structure the entire application does not describe specific algorithms focuses on relationships between classes.

Structural Patterns

Structural Patterns describe how objects and classes can be combined to form structures. We distinguish between object patterns and class patterns. The difference is that class patterns describe relationships and structures with the help of inheritance. Object patterns, on other hand, describe how objects can be associated and aggregated to form larger, more complex structures.

The difference between *class* patterns and *object* patterns is that class patterns describe how inheritance can be used to provide more useful program interfaces. Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition, or the inclusion of objects within other objects.

1. Adapter - Match interfaces of different classes.
2. Bridge - Separates an object's abstraction from its implementation.
3. Composite - A tree structure of simple and composite objects.
4. Decorator - Add responsibilities to objects dynamically.
5. Façade - A single class that represents an entire subsystem.
6. Flyweight - A fine-grained instance used for efficient sharing.
7. Proxy - An object representing another object.

Note: To remember structural pattern best is (ABCDFFP)

Adapter Pattern

Many times two classes are incompatible because of their incompatible interfaces. The Adapter pattern is used to translate the interface of one class into another interface.

The classes who can't work together due to incompatible interfaces, we can make these classes working together. A class adapter uses multiple inheritance (by extending one class and/or implementing one or more classes) to adapt one interface to another. An object adapter relies on object aggregation.

Adapter pattern can be used to make one class interface match another to make programming easier. The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program. The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.

There are two ways to do this: by inheritance, and by object composition (aggregation). In the first case, we derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface. The other way is to include the original class inside the new one and create the methods to translate calls within the new class. These two approaches called as class adapters and object adapters.

Example

Consider the below figure "Incompatible interfaces" both of them are collections to hold string values. Both of them have a method which helps us to add string in to the collection. One of the methods is named as "add" and the other as "push". We want to make the stack object compatible with the collection object.

```
package design.patterns.adapter;

public abstract class Stack
{
    public abstract void push(String s);
}
```

```
package design.patterns.adapter;

public abstract class CollectionBase
{
    public abstract void add(String str);
}
```

```
package design.patterns.adapter;

public class ClsCollection extends CollectionBase
{
    @Override
    public void add(String str)
    {
        System.out.println("Added element " + str + " to the list");
    }
}
```

```
package design.patterns.adapter;

public class ClsStack extends Stack
{
    @Override
    public void push(String s)
    {
        System.out.println("Pushed element " + s + " in stack");
    }
}
```

Solution 1:

First let's try to cover class adapter pattern. In class adapter pattern we have inherited the "ClsStack" class in the "ClassAdapter" and made it compatible with the "ClsCollection" class.

```
package design.patterns.adapter;

public class ClassAdapter extends ClsStack
{
    public void add(String str)
    {
        this.push(str);
    }
}
```

Solution 2:

"Object Adapter pattern" shows a broader view of how we can achieve the same. "ObjectAdapter" class wraps on the top of the "ClsStack" class and aggregates the "push" method inside a new "add" method, thus making both the classes compatible.

```
package design.patterns.adapter;

public class ObjectAdapter extends ClsCollection
{
    private ClsStack objStack = new ClsStack();  // Aggregatior

    public void add(String str)
    {
        objStack.push(str);
    }
}
```

Where to use

- When you want to use an existing class and its interface does not match the one you need.
- When you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- When you want to increase transparency of classes. When you want to make a pluggable kit.

Usage example

The Java API uses the Adapter pattern, WindowAdapter, ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter.

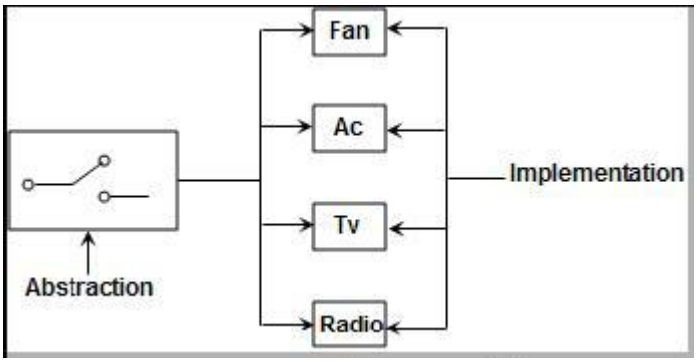
Bridge Pattern

Decouple an abstraction or interface from its implementation so that the two can vary independently. Bridge makes a clear-cut between abstraction and implementation. Bridge pattern helps to decouple abstraction from implementation. With this if the implementation changes it does not affect abstraction and vice versa.

At first sight, the bridge pattern looks much like the Adapter pattern, in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class's interface from its implementation, so that you can vary or replace the implementation without changing the client code.

Example:

Consider the figure "Abstraction and Implementation". The switch is the abstraction and the electronic equipments are the implementations. The switch can be applied to any electronic equipment, so the switch is an abstract thinking while the equipments are implementations.



Let's try to code the same switch and equipment example. First we segregate the implementation and abstraction in to two different classes. Following figures shows we have made an interface "IEquipment" with "start()" and "stop()" methods. We have implemented two equipments one is the refrigerator and the other is the bulb.

```java
package design.patterns.bridge;

public interface IEquipment
{
    void start();
    void stop();
}
```

```java
package design.patterns.bridge;

public class ClsBulb implements IEquipment
{
    @Override
    public void start()
    {
        System.out.println("Started Bulb");
    }

    @Override
    public void stop()
    {
        System.out.println("Stopped Bulb");
    }
}
```

```java
package design.patterns.bridge;

public class ClsRefrigerator implements IEquipment
{
    @Override
    public void start()
    {
        System.out.println("Started Refrigerator");
    }

    @Override
    public void stop()
    {
         System.out.println("Stopped Refrigerator");
    }
}
```

The second part is the abstraction. Switch is the abstraction in our example. It has a “SetEquipment” method which sets the object. The “On” method calls the “Start” method of the equipment and the “off” calls the “stop”.

```java
package design.patterns.bridge;

public interface ISwitch
{
    void on();
    void off();
}
```

```java
package design.patterns.bridge;

public class ClsSwitch implements ISwitch
{
    private IEquipment objEquipment;

    public void setEquipment(IEquipment objEquip)
    {
        this.objEquipment = objEquip;
    }

    @Override
    public void off()
    {
        objEquipment.stop();
    }

    @Override
    public void on()
    {
         objEquipment.start();
    }
}
```

Finally we see the Test code. You can see we have created the implementation objects and the abstraction objects separately. We can use them in an isolated manner.

```java
package design.patterns.bridge;

public class Test
{
    public static void main(String arg[])
    {
        IEquipment objBulb = new ClsBulb();
        IEquipment objRefrigerator = new ClsRefrigerator();
        ClsSwitch objSwitch = new ClsSwitch();

        objSwitch.setEquipment(objBulb);
        objSwitch.on();
        objSwitch.off();

        objSwitch.setEquipment(objRefrigerator);
        objSwitch.on();
        objSwitch.off();
    }
}
```

Where to use

- When you want to separate the abstract structure and its concrete implementation. When you want to share an implementation among multiple objects,
- When you want to reuse existing resources in an 'easy to extend' fashion.
- When you want to hide implementation details from clients. Changes in implementation should have no impact on clients.

Benefits

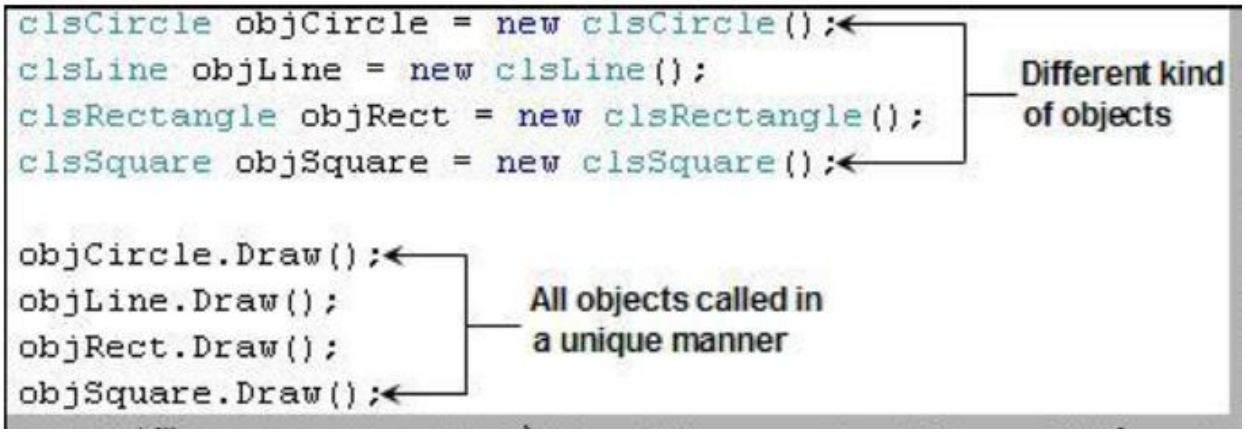
Implementation can be selected or switched at run-time. The abstraction and implementation can be independently extended or composed.

Drawbacks/consequences

Double indirection - In the example, methods are implemented by subclasses. Super class must delegate the message to a subclass which implements the appropriate method. This will have a slight impact on performance.

Composite Pattern

Composite pattern allows treating different objects in a similar fashion. Following figure shows how different objects are called in a uniform manner.



In order to treat objects in a uniformed manner we need to inherit them from a common interface.

Frequently programmers develop systems in which a component may be an individual object or it may represent a collection of objects. The Composite pattern is designed to accommodate both cases. You can use the Composite pattern to build part-whole hierarchies or to construct data representations of trees. In summary, a composite is a collection of objects, any one of which may be either a composite, or just a primitive object. In tree nomenclature, some objects may be nodes with additional branches and some may be leaves.

The Composite pattern helps you to create tree structures of objects without the need to force clients to differentiate between branches and leaves regarding usage. The Composite pattern lets clients treat individual objects and compositions of objects uniformly.

Example:

In order to treat objects in a uniformed manner we need to inherit them from a common interface. Following figures shows the objects inheriting from a common interface.

```java
package design.patterns.composite;

public interface IUserInterface
{
    void draw();
}
```

```java
package design.patterns.composite;

public class ClsCircle implements IUserInterface
{
    @Override
    public void draw()
    {
        System.out.println("Drawn circle");
    }
}
```

```java
package design.patterns.composite;

public class ClsLine implements IUserInterface
{

    @Override
    public void draw()
    {
        System.out.println("Drawn Line");
    }

}
```

```java
package design.patterns.composite;

public class ClsSquare implements IUserInterface
{
    @Override
    public void draw()
    {
        System.out.println("Drawn Square");
    }

}
```

Following figure shows how we added all the different kind of objects in to one collection and then called them in a uniform fashion.

```
package design.patterns.composite;

import java.util.ArrayList;

public class Test
{
    public static void main(String args[])
    {
        List<IUserInterface> list = new ArrayList<IUserInterface>();

        IUserInterface objCircle = new ClsCircle();
        IUserInterface objLine = new ClsLine();
        IUserInterface objSquare = new ClsSquare();

        list.add(objCircle);
        list.add(objLine);
        list.add(objSquare);

        for(IUserInterface obj : list)
        {
            obj.draw();
        }
    }
}
```

Where to use

- When you want to represent a part-whole relationship in a tree structure.
- When you want clients to be able to ignore the differences between compositions of objects and individual objects.
- When the structure can have any level of complexity and is dynamic.

Benefits

- Define class hierarchies consisting of primitive objects and composite objects.
- Makes it easier to add new kind of components.

Drawbacks/consequences

The problem that develops here is an ability to distinguish between nodes and leaves. Nodes have children and can have children added to them, while leaves do not at the moment have children, and in some implementations may be prevented from having children added to them. Some authors have suggested creating a separate interface for nodes and leaves, where a leaf could have the methods.

The Composite pattern makes it easy for you to add new kinds of components to your collection as long as they support a similar programming interface. On the other hand, this has the disadvantage of making your system overly general. You might find it harder to restrict certain classes where this would normally be desirable.

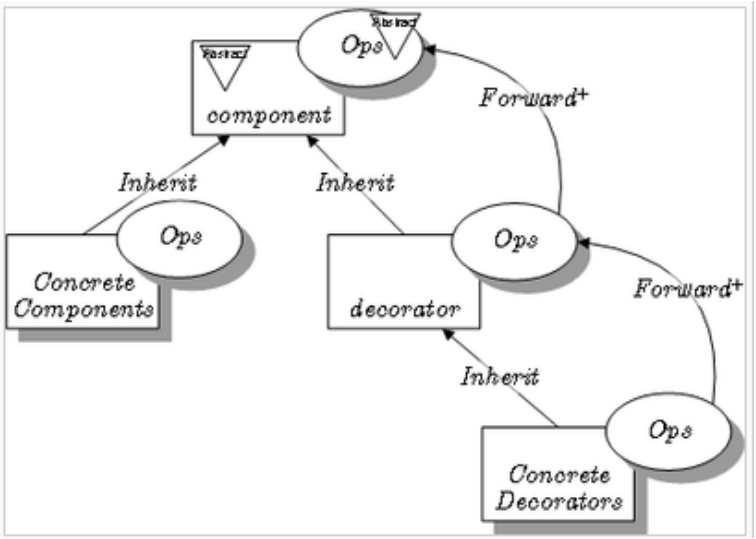
Decorator Pattern

In object-oriented programming, the decorator pattern is a design pattern that allows adding new behavior to an existing class dynamically.

The Decorator pattern lets you attach additional responsibilities and modify instance functionality dynamically.

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a class at runtime. This is achieved by designing a new decorator class that wraps the original class. This wrapping could be achieved by the following sequence of steps:

1. Subclass the original "Component" class into a "Decorator" class (see UML diagram).
2. In the Decorator class, add a Component pointer as a field.
3. Pass a Component to the Decorator constructor to initialize the Component pointer.
4. In the Decorator class, redirect all "Component" methods to the "Component" pointer.
5. In the Decorator class, override any Component method(s) whose behavior needs to be modified.

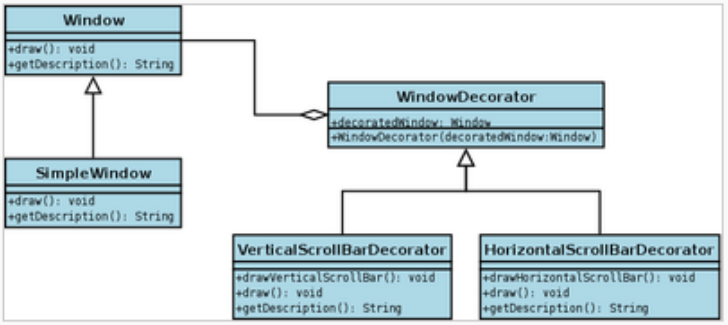


This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding a new functionality to the overridden method(s).

The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time whereas decorating can provide new behavior at runtime.

Example:

As an example, consider a window in a windowing system. To allow scrolling of the window's contents, we may wish to add horizontal or vertical scrollbars to it, as appropriate. Assume windows are represented by instances of the Window class, and assume this class has no functionality for adding scrollbars. We could create a subclass ScrollingWindow that provides them, or we could create a ScrollingWindowDecorator that adds this functionality to existing Window objects. At this point, either solution would be fine.



UML Diagram for the Window Example

Now let's assume we also wish the option to add borders to our windows. Again, our original Window class has no support. The ScrollingWindow subclass now poses a problem, because it has effectively created a new kind of window. If we wish to add border support to all windows, we must create subclasses WindowWithBorder and ScrollingWindowWithBorder. Obviously, this problem gets worse with every new feature to be added. For the decorator solution, we simply create a new BorderedWindowDecorator at runtime, we can decorate existing windows with the ScrollingWindowDecorator or the BorderedWindowDecorator or both, as we see fit.

```
// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription(); // returns a description of the Window
}


// implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow; // the Window being decorated

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
}


// the first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }

    public void draw() {
        drawVerticalScrollBar();
        decoratedWindow.draw();
    }

    private void drawVerticalScrollBar() {
        // draw the vertical scrollbar
    }

    public String getDescription() {
        return decoratedWindow.getDescription() + ", including vertical scrollbars";
    }
}
```

```
// the second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }

    public void draw() {
        drawHorizontalScrollBar();
        decoratedWindow.draw();
    }
```

```
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator(new SimpleWindow()));

        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

Where to use

- When you want to add responsibilities to individual objects dynamically and transparently, without affecting the original object or other objects.
- When you want to add responsibilities to the object that you might want to change in the future.
- When extension by static sub-classing is impractical.

Benefits

- More flexibility than static inheritance.
- Avoids feature-laden classes high up in the hierarchy.
- Simplifies coding because you write a series of classes each targeted at a specific part of the functionality rather than coding all behavior into the object.
- Enhances the object's extensibility because you make changes by coding new classes.

Drawbacks/consequences

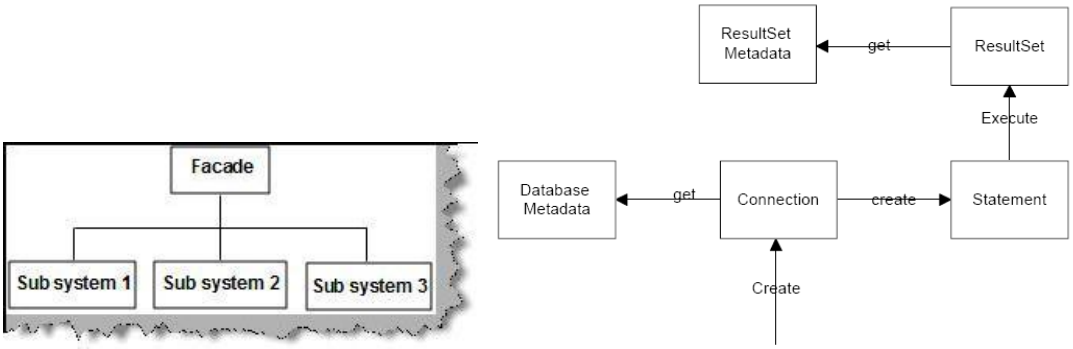
One thing to keep in mind when implementing the Decorator pattern is that you need to keep the component interface simple. You want to avoid making the component interface overly complex, because a complex interface will make it that much harder to get each decorator right.

Another potential drawback of the Decorator pattern is the performance overhead associated with a long chain of decorators.

Facade Pattern (Face or Front Appearance)

Frequently, as your programs evolve and develop, they grow in complexity. These patterns sometimes generate so many classes that it is difficult to understand the program's flow. Furthermore, there may be a number of complicated subsystems, each of which has its own complex interface.

The Façade pattern allows you to simplify this complexity by providing a simplified interface to these subsystems. This simplification may in some cases reduce the flexibility of the underlying classes, but usually provides all the function needed for all.



The database classes in the java.sql package provide an excellent example of a set of quite low level classes that interact in a convoluted manner. Java provides a set of classes that connect to databases using an interface called JDBC. You can connect to any database for which the manufacturer has provided a JDBC connection class.

This design pattern provides an integrated (combined) interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use. A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

Example:

Following figure shows how class "ClsOrder" unifies (combines) / uses "ClsProduct", "ClsPayment" and "ClsInvoice" to implement "PlaceOrder" functionality. Client has to only create instance of ClsOrder & call the method placeOrder() instead of creating instances & calling method of each individual subsystems.

```java
package design.patterns.facade;

public class ClsProduct
{
    public void getDetails()
    {
        System.out.println("Product Information");
    }
}
```

```java
package design.patterns.facade;

public class ClsPayment
{
    public void payOnline()
    {
        System.out.println("Paid Online");
    }
}
```

```java
package design.patterns.facade;

public class ClsInvoice
{
    public void printInvoice()
    {
        System.out.println("Printing Invoice");
    }
}
```

```java
package design.patterns.facade;

public class ClsOrder
{
    public void placeOrder()
    {
        ClsProduct clsProduct = new ClsProduct();
        ClsPayment clsPayment = new ClsPayment();
        ClsInvoice clsInvoice = new ClsInvoice();

        clsProduct.getDetails();
        clsPayment.payOnline();
        clsInvoice.printInvoice();
    }

    public static void main(String args[])
    {
        ClsOrder c = new ClsOrder();
        c.placeOrder();
    }
}
```

Where to use

The Facade can be used to make a software library easier to use and understand, since the Facade has convenient methods for common tasks. It can make code that uses the library more readable. The pattern can also be used to reduce dependencies of outside code on the inner workings of a library, since most code uses the Facade it allows more flexibility when developing the system. A final usage scenario is where we can wrap several poorly-designed APIs with a single well-designed API.

Benefits

The main benefit with the Facade pattern is that we can combine very complex method calls and code blocks into a single method that performs a complex and recurring task. It also reduces code

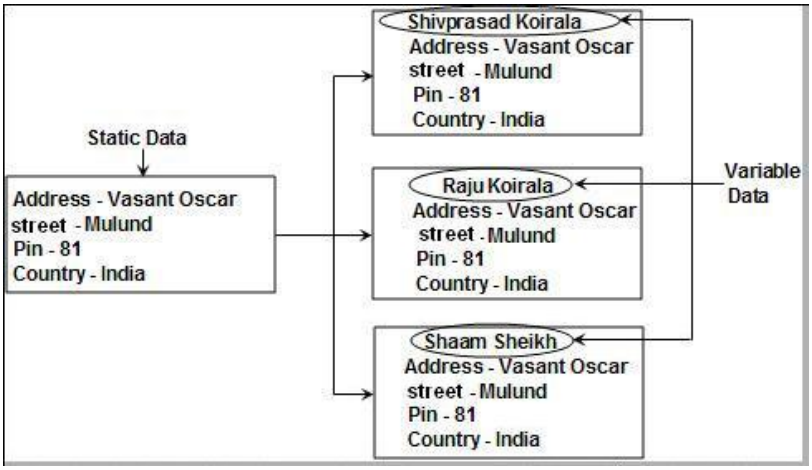
dependencies between libraries or packages, making programmers more apt to consideration before writing new code that exposes the inner workings of a library or a package. Also, since the Façade makes a weak coupling between the client code and other packages or libraries it allows us vary the internal components since the client does not call them directly.

Drawbacks/consequences

One drawback is that we have much less control of what goes on beyond the surface. Also, if some classes require small variations to the implementation of Facade methods, we might end up with a mess.

Flyweight Pattern

Fly weight pattern is useful where we need to create many objects and these entire objects share some kind of common data. Consider following figure. We need to print visiting card for all employees in the organization. We have two parts of data one is the variable data i.e. the employee name and the other is static data i.e. address. We can minimize memory by just keeping one copy of the static data and referencing the same data in all objects of variable data. So we create different copies of variable data, but reference the same copy of static data. With this we can optimally use the memory.



The Flyweight pattern provides a mechanism by which you can avoid creating a large number of 'expensive' objects and instead reuse existing instances to represent new ones.

There are cases in programming where it seems that you need to generate a very large number of small class instances to represent data. Sometimes you can greatly reduce the number of different classes that you need to instantiate if you can recognize that the instances are fundamentally the same except for a few parameters. If you can move those variables outside the class instance and pass them in as part of a method call, the number of separate instances can be greatly reduced.

The Flyweight design pattern provides an approach for handling such classes. It refers to the instance's *intrinsic* data that makes the instance unique, and the *extrinsic* data which is passed in as arguments.

An another example in *Design Patterns*, each character in a font is represented as a single instance of a character class, but the positions where the characters are drawn on the screen are kept as external data so that there needs to be only one instance of each character, rather than one for each appearance of that character.

Example

Below is a sample code demonstrating how flyweight can be implemented practically. We have two classes, “ClsVariableAddress” which has the variable data and second “ClsAddress” which has the static data. To ensure that we have only one instance of “ClsAddress” we have made a wrapper class “ClsStatic” and created a static instance of the “ClsAddress” class. This object is aggregated in the “ClsVariableAddress” class.

```
package design.patterns.flyweight;

public class ClsStaticAddress
{
    public String strHouseNumber = "452";
    public String strAddress = "msr nagar";
    public String strStreet = "6th cross";
    public String strLandMark = "Near Balaji Kalyana Mantupa";
    public String strCity = "Bangalore";
    public String strPin = "560054";
    public String strCountry = "India";
}
```

```
package design.patterns.flyweight;

public class ClsStatic
{
    public static ClsStaticAddress objAddress = new ClsStaticAddress();
}
```

```
package design.patterns.flyweight;

public class ClsVariableAddress
{
    public String strName;

    public ClsVariableAddress(String name)
    {
        strName = name;
    }
    public ClsStaticAddress address()
    {
        return ClsStatic.objAddress;
    }

    public String getStrName()
    {
        return strName;
    }
}
```

Following figure shows we have created two objects of "ClsVariableAddress" class, but internally the static data i.e. the address is referred to only one instance.

```
package design.patterns.flyweight;

public class Test
{
    public static void main(String s[])
    {
        ClsVariableAddress a = new ClsVariableAddress("Tom");
        ClsVariableAddress b = new ClsVariableAddress("Harry");
        ClsVariableAddress c = new ClsVariableAddress("Jack");

        display(a);
        display(b);
        display(c);

    }

    public static void display(ClsVariableAddress a)
    {
        sop(a.getStrName());
        sop(a.address().strHouseNumber);
        sop(a.address().strAddress);
        sop(a.address().strStreet);
        sop(a.address().strCity);
        sop(a.address().strPin);
        sop(a.address().strCountry);
    }

    public static void sop(Object o)
    {
        System.out.println(o);
    }
}
```

Where to use

- When there is a very large number of objects that may not fit in memory.
- When most of an objects state can be stored on disk or calculated at runtime.
- When there are groups of objects that share state.
- When the remaining state can be factored into a much smaller number of objects with shared state.

Benefits

Reduce the number of objects created, decrease memory footprint and increase performance.

Drawbacks/consequences

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

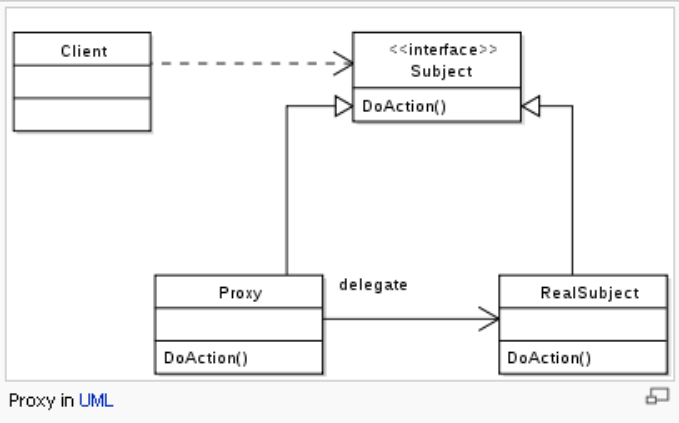
Proxy Pattern

A proxy is a class functioning as an interface to something else. The proxy could interface to the resource that is expensive or impossible to duplicate. This actual object can be a huge image or a data object which is very large and cannot be duplicated. So you can create multiple proxies and point towards the actual object (memory consuming object) and perform operations. This avoids duplication of the heavy object and thus saving memory. Proxies are references which points towards the actual object.

There are several cases where a Proxy can be useful. Few of them as follow.

1. If an object, such as a large image, takes a long time to load.
2. If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
3. If the object has limited access rights, the proxy can validate the access permissions for that user.

In proxy pattern, typically one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory may be deallocated.



Example:

```
import java.util.*;

interface Image {
    public void displayImage();
}

//on System A
class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }

    public void displayImage() {
        System.out.println("Displaying " + filename);
    }
}
```

```
//on System B
class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) {
        this.filename = filename;
    }
    public void displayImage() {
        image = new RealImage(filename);
        image.displayImage();
    }
}

class ProxyExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        Image image2 = new ProxyImage("HiRes_10MB_Photo2");

        image1.displayImage(); // loading necessary
        image2.displayImage(); // loading necessary
    }
}
```

The program output.

```
Loading    HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Loading    HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
```

Where to use

- When the creation of one object is relatively expensive it can be a good idea to replace it with a proxy that can make sure that instantiation of the expensive object is kept to a minimum.
- Proxy pattern implementation allows for login and authority checking before one reaches the actual object that's requested.
- Proxy pattern can provide a local representation for an object in a remote location.

Benefits

Gives the ability to control access to an object, whether it's because of a costly creation process of that object or security issues.

Drawbacks/consequences

Introduces another abstraction level for an object, if some objects accesses the target object directly and another via the proxy there is a chance that they get different behavior this may or may not be the intention of the creator.

Summary of Structural Patterns

- Adapter pattern, used to change the interface of one class to that of another one.
- Bridge pattern, intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. You can then change the interface and the underlying class separately.
- Composite pattern, a collection of objects, any one of which may be either itself a Composite, or just a primitive object.
- Decorator pattern, a class that surrounds a given class, adds new capabilities to it, and passes all the unchanged methods to the underlying class.
- Façade pattern, which groups a complex object hierarchy and provides a new, simpler interface to access those data.
- Flyweight pattern, which provides a way to limit the proliferation of small, similar class instances by moving some of the class data outside the class and passing it in during various execution methods.
- Proxy pattern, which provides a simple place-holder class for a more complex class which is expensive to instantiate.

Behavioral Patterns

Behavioral patterns are patterns that focus on the interactions between cooperating objects. The interactions between cooperating objects should be such that they are communicating while maintaining as loose coupling as possible. The loose coupling is the key to n-tier architectures. The implementation and the client should be loosely coupled in order to avoid hard-coding and dependencies.

Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

1. Mediator - Defines simplified communication between classes.
2. Memento - Capture and restore an object's internal state.
3. Interpreter - A way to include language elements in a program.
4. Iterator - Sequentially access the elements of a collection.
5. Chain of Responsibility - A way of passing a request between chains of objects.
6. Command - Encapsulate a command request as an object.
7. State - Alter an object's behavior when its state changes.
8. Strategy - Encapsulates an algorithm inside a class.
9. Observer - A way of notifying change to a number of classes.
10. Template Method - Defer the exact steps of an algorithm to a subclass.
11. Visitor - Defines a new operation to a class without change.

Note: - To remember behavioral pattern best is 2 MICS On TV (MMIICCSSOTV).

Mediator Pattern

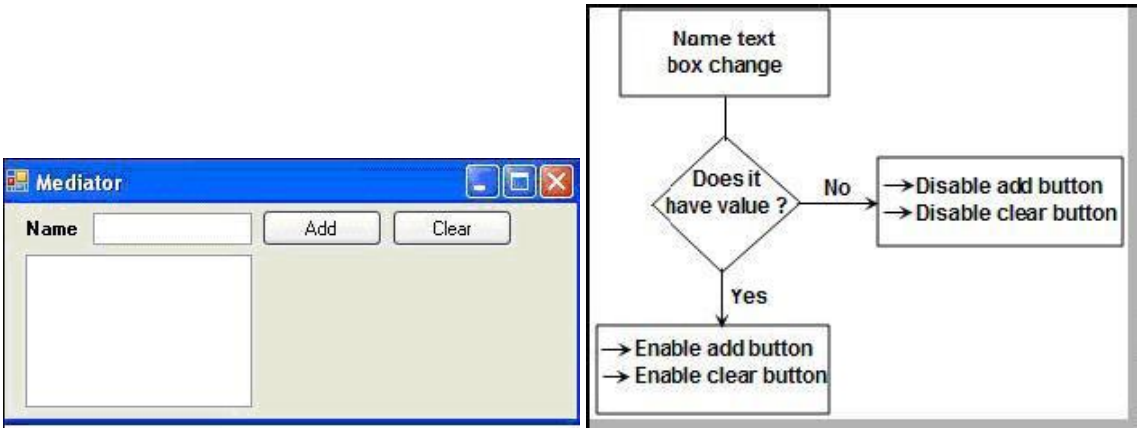
The mediator pattern provides a unified interface to a set of interfaces in a subsystem.

Usually a program is made up of a large number of classes. So the logic and computation is distributed among these classes. Increasing the number of classes in the system, 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 with 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 lowering the coupling.

The Mediator pattern addresses this problem by promoting looser coupling between these classes. Mediators accomplish this by being the only class that has detailed knowledge of the methods of other classes. Classes send inform the mediator when changes occur and the Mediator passes them on to any other classes that need to be informed.

Many times in projects communication between components are complex. Due to this the logic between the components becomes very complex. Mediator pattern helps the objects to communicate in a disassociated manner, which leads to minimizing complexity.

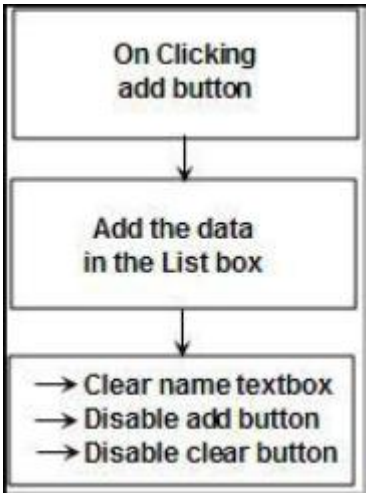


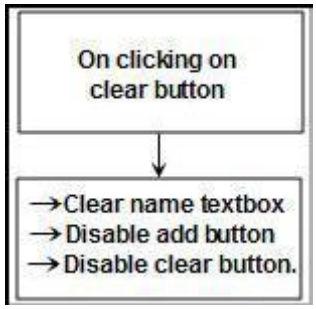
Let's consider the figure which depicts a true scenario of the need of mediator pattern. It's a very user-friendly user interface. It has three typical scenarios.

Scenario 1:- When a user writes in the text box it should enable add and the clear button. In case there is nothing in the text box it should disable add and the clear button.

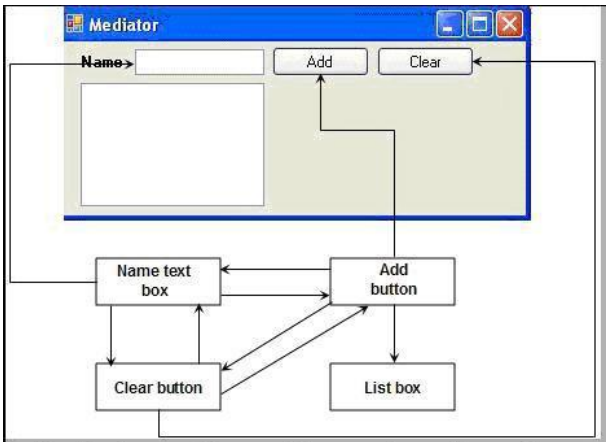
Scenario 2:- When the user clicks on the add button the data should get entered in the list box. Once the data is entered in the list box it should clear the text box and disable add and clear button.

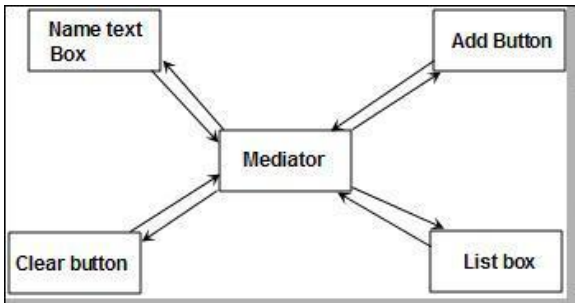
Scenario 3:- If the user clicks the clear button it should clear the name text box and disable add and clear button.





Now looking at the above scenarios for the UI we can conclude how complex the interaction will be in between these UI's. Below figure depicts the logical complexity.

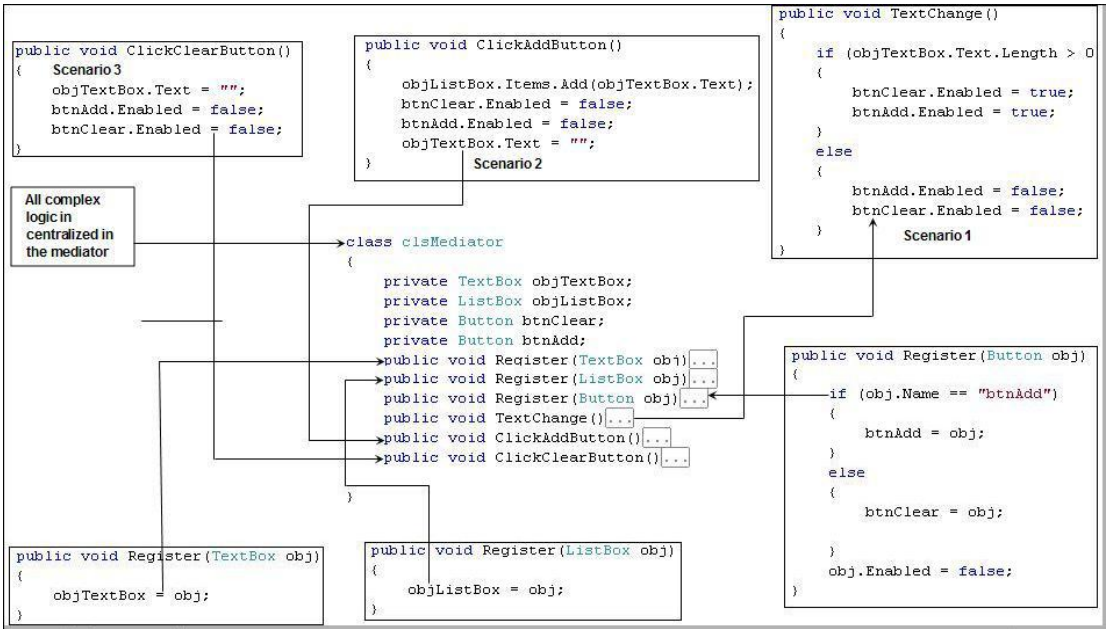




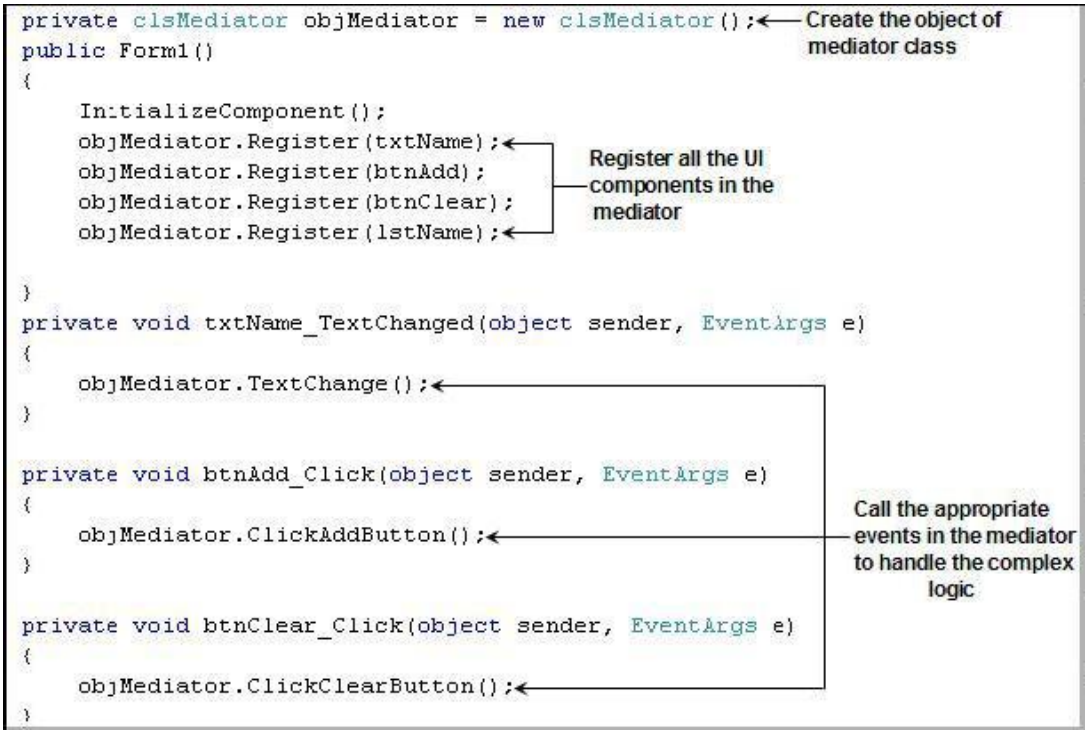
Ok now let give a nice picture as shown above simplifying using mediator. Rather than components communicating directly with each other if they communicate to centralized component like mediator and then mediator takes care of sending those messages to other components, logic will be neat and clean.

Example

Below figure shows the complete code overview of what the mediator class will look like. The first thing the mediator class does is takes the references of the classes which have the complex communication. So here we have exposed three overloaded methods by name “Register”. “Register” method takes the text box object and the button objects. The interaction scenarios are centralized in “ClickAddButton”,”TextChange” and “ClickClearButton” methods. These methods will take care of the enable and disable of UI components according to scenarios.



The client logic is pretty neat and cool now. In the constructor we first register all the components with complex interactions with the mediator. Now for every scenario we just call the mediator methods. In short when there is a text change we call the “TextChange” method of the mediator, when the user clicks add we call the “ClickAddButton” and for clear click we call the “ClickClearButton”.




```
private clsMediator objMediator = new clsMediator();
public Form1()
{
    InitializeComponent();
    objMediator.Register(txtName);
    objMediator.Register(btnAdd);
    objMediator.Register(btnClear);
    objMediator.Register(lstName);

}
private void txtName_TextChanged(object sender, EventArgs e)
{
    objMediator.TextChange();
}

private void btnAdd_Click(object sender, EventArgs e)
{
    objMediator.ClickAddButton();
}

private void btnClear_Click(object sender, EventArgs e)
{
    objMediator.ClickClearButton();
}
```

Where to use

The Mediator pattern can be used when a set of objects communicate in well-specified but complex ways and the resulting interdependencies are unstructured and hard to grasp. If it is difficult to reuse an object because it refers to and communicates with many other objects this pattern is a good solution. Communicating objects' identities will be protected when using the Mediator pattern which is good when security is important. Also, if you like to customize some behavior which is spread out between several classes without a lot of sub-classing this pattern should be applied.

The pattern is used in many modern systems that reflect a send/receive protocol, such as list servers and chat rooms. Another area of use is graphical user interfaces, where the mediator can encapsulate a collective behavior to control and coordinate the interactions of a group of GUI widgets. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator which reduces the number of interconnections.

Benefits

1. Limited sub-classing, since a mediator localizes behavior that otherwise would be distributed among several objects. Changing some behavior requires us to subclass only the mediator.
2. Colleagues become decoupled which allows us to vary and reuse colleague and mediator classes independently.

3. A mediator simplifies object protocols since it replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many interactions are easier to understand, maintain, and extend.
4. The mediator abstracts object cooperation. The mediator lets you focus on how objects interact apart from their individual behaviors which can help clarify how objects interact in a system.
5. Centralized control is achieved by trading complexity of interaction for complexity in the mediator

Drawbacks/consequences

A mediator encapsulates protocols and can become more complex than any individual colleague. This can make the mediator itself a very complex and large piece of code that is hard to maintain.

One important drawback is that the Mediator pattern can have a performance impact on a system. Since all communication must go through the mediator, it can become a bottleneck.

Memento Pattern

Memento pattern record an object internal state without violating encapsulation and reclaim it later without knowledge of the original object. A memento is an object that stores a snapshot of the internal state of another object.

Suppose you would like to save the internal state of an object so you can restore it later. Ideally, it should be possible to save and restore this state without making the object and without violating encapsulation. This is the purpose of the Memento pattern.

Objects frequently expose only some of their internal state using public methods, but you would still like to be able to save the entire state of an object because you might need to restore it later.

If all of the information describing an object is available in public variables, it is not that difficult to save them in some external store. However, making these data public makes the entire system vulnerable to change by external program code, when we usually expect data inside an object to be private and encapsulated from the outside world.

The Memento pattern attempts to solve this problem by having privileged access to the state of the object you want to save. Other objects have only a more restricted access to the object, thus preserving their encapsulation. This pattern defines three roles for objects:

1. **Originator** is the object whose state we want to save.
2. **Memento** is another object that saves the state of the Originator.
3. **Caretaker** manages the timing of the saving of the state, saves the Memento and, if needed, uses the Memento to restore the state of the Originator.

Saving the state of an object without making all of its variables publicly available is tricky. In Java, this privileged access is possible using a little known and infrequently used protection mode. Variables within a Java class can be declared as

1. Private
2. Protected
3. Public, or
4. (Private protected)

Variables with no declaration are treated as private protected. Other classes can access public variables, and derived classes can access protected variables. However, another class in the same module can access protected or private-protected variables. This feature of Java that we can use to build Memento objects. For example, suppose you have classes A and B declared in the same module:

```
public class A {
int x, y:
public Square() {}
x = 5;                              //initialize x
}
//-------------------------
class B {
public B() {
  A a = new A();                    //create instance of A
  System.out.println (a.x);         //has access to variables in A
 }
}
```

Class A contains a private-protected variable *x*. In class B in the same module, we create an instance of A, which automatically initializes *x* to 5. Class B has direct access to the variable *x* in class A and can print it out without compilation or execution error. It is exactly this feature that we will use to create a Memento.

Memento pattern is the way to capture objects internal state without violating encapsulation. Memento pattern helps us to store a snapshot which can be reverted at any moment of time by the object. Consider following figure that shows a customer screen. Let's say if the user starts editing a customer record and he makes some changes. Later he feels that he has done something wrong and he wants to revert back to the original data. This is where memento comes in to play. It will help us store a copy of data and in case the user presses cancel the object restores to its original state.

Example:

Below is the customer class "ClsCustomer" which has the aggregated memento class "ClsCustomerMemento" which will hold the snapshot of the data. The memento class "ClsCustomerMemento" is the exact replica (excluding methods) of the customer class "ClsCustomer". When the customer class "ClsCustomer" gets initialized the memento class also gets initialized. When the customer class data is changed the memento class snapshot is not changed. The "revert" method sets back the memento data to the main class.

```
package design.behavioral.patterns.memento;

public class ClsCustomerMemento
{

    public String strCustomerName = "";
    public String strCustomerCode = "";
    public String strAddress = "";


    public ClsCustomerMemento(String strCustomerName, String strCustomerCode, String strAddress)
    {
        this.strCustomerName = strCustomerName;
        this.strCustomerCode = strCustomerCode;
        this.strAddress = strAddress;
    }
}
package design.behavioral.patterns.memento;

public class ClsCustomer
{
    private String strCustomerName = "";
    private String strCustomerCode = "";
    private String strAddress = "";

    private ClsCustomerMemento objMememto;

    public ClsCustomer()
    {
        strCustomerName = "Pawan Modi";
        strCustomerCode = "8126";
        strAddress = "msr nagar Bangalore";
        objMememto = new ClsCustomerMemento(strCustomerName,strCustomerCode,strAddress);
    }

    public void update(String strCustomerName, String strCustomerCode, String strAddress)
    {
        this.strCustomerName = strCustomerName;
        this.strCustomerCode = strCustomerCode;
        this.strAddress = strAddress;
    }
    public void revert()
    {
        strCustomerName = objMememto.strCustomerName;
        strCustomerCode = objMememto.strCustomerCode;
        strAddress = objMememto.strAddress;
    }

    public void display()
    {
        sop(strCustomerName);
        sop(strCustomerCode);
        sop(strAddress);
    }
    public void sop(Object o)
    {
        System.out.println(o);
    }

}
```

The client code is pretty simple. We create the customer class. In case we have issues we call the cancel method which in turn calls the "revert" method and reverts the changed data back to the memento snapshot data. Following figure shows the same in a pictorial format.

```
package design.behavioral.patterns.memento;

public class Test
{
    public static void main(String arg[])
    {
        ClsCustomer objCustomer = new ClsCustomer();
        display(objCustomer);
        update(objCustomer);
        display(objCustomer);
        cancelUpdate(objCustomer);
    }

    public static void cancelUpdate(ClsCustomer objCustomer)
    {
        objCustomer.revert();
        display(objCustomer);
    }
    public static void update(ClsCustomer objCustomer)
    {
        objCustomer.update("Tom", "2345", "rm nagar Bangalore");
    }

    public static void display(ClsCustomer objCustomer)
    {
        objCustomer.display();
    }
}
```

Where to use

- When letting some info in an object available by another object.
- When you want to create snapshots of a state for an object.
- When you need undo/redo features.

Benefits

Ability to restore an object to its previous state.

Drawbacks/consequences

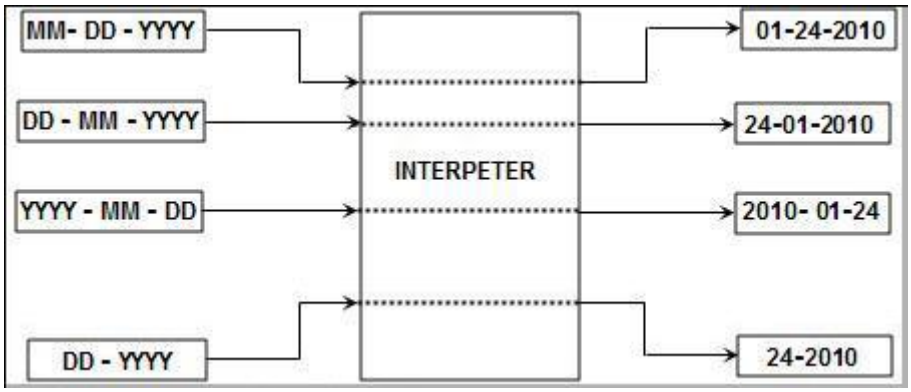
1. Care should be taken if the originator may change other objects or resources
2. Memento pattern operates on a single object.
3. Using memento to store large amounts of data from Originator might be expensive if clients create and return mementos frequently.

Interpreter Pattern

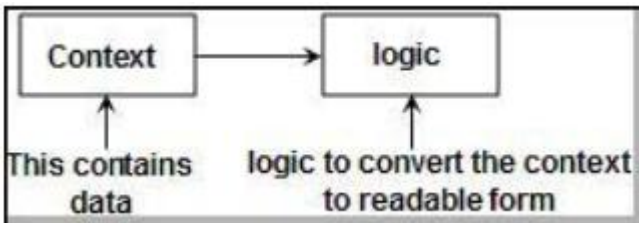
The interpreter pattern specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence.

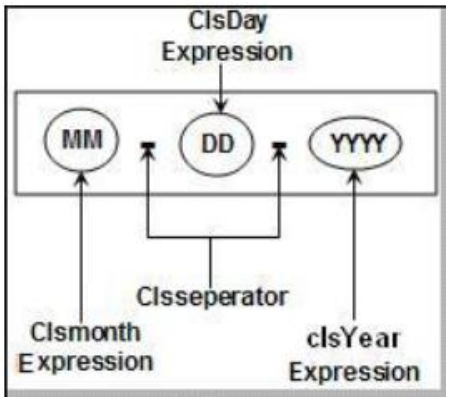
Some programs benefit from having a language to describe operations they can perform. The Interpreter pattern generally describes defining a grammar for that language and using that grammar to interpret statements in that language.

Interpreter pattern allows us to interpret grammar in to code solutions. Grammars are mapped to classes to arrive to a solution. For instance 7 – 2 can be mapped to "ClsMinus" class. In one line interpreter pattern gives us the solution of how to write an interpreter which can read a grammar and execute the same in the code. For instance below is a simple example where we can give the date format grammar and the interpreter will convert the same in to code solutions and give the desired output.



The two different components of interpreter pattern are context & logic. Context contains the data and the logic part contains the logic which will convert the context to readable format.



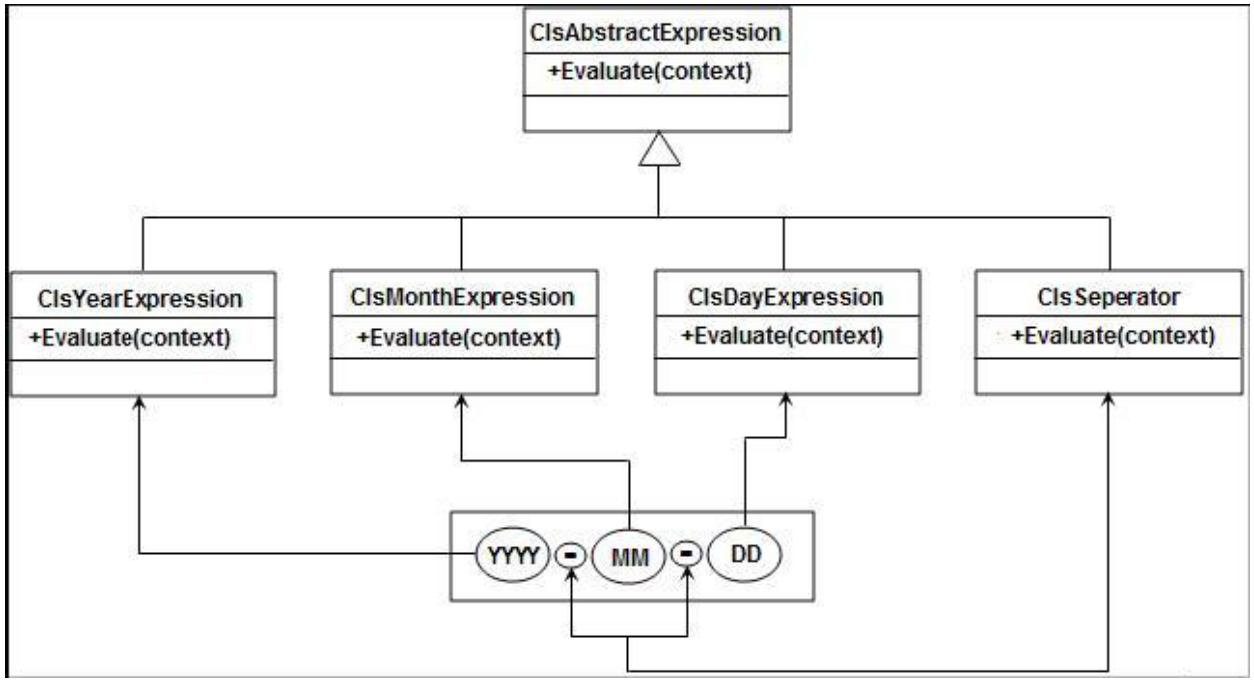


Example

Let's understand what is the grammar in the date format is. To define any grammar we should first break grammar in small logical components.

So we have broken the date format in to four components Month, Day, Year and the separator. For all these four components we will define separate classes which will contain the logic.

As said there are two types of classes one is the expression (logical) classes which contain logic and the other is the context class which contain data. We have defined all the expression parsing in different classes, all these classes inherit from common interface "ClsAbstractExpression" with a method "evaluate". The "evaluate" method takes a context class which has the data; this method parses data according to the expression logic. For instance "ClsYearExpression" replaces the "YYYY" with the year value, "ClsMonthExpression" replaces the "MM" with month and so on.




```
package design.behavioral.patterns.interpreter;

public class ClsContext
{
    public String strExpression = "";

    public ClsContext(String expression)
    {
        strExpression = expression;
    }
}
```

```
package design.behavioral.patterns.interpreter;

public abstract class ClsAbstractExpression
{
    public abstract void evaluate(ClsContext objContext);
}
```

```java
package design.behavioral.patterns.interpreter;

public class ClsDayExpression extends ClsAbstractExpression
{
    @Override
    public void evaluate(ClsContext objContext)
    {
        String strTemp = objContext.strExpression;

        objContext.strExpression = strTemp.replace("dd", "22");
    }
}
```

```java
package design.behavioral.patterns.interpreter;

public class ClsMonthExpression extends ClsAbstractExpression
{

    @Override
    public void evaluate(ClsContext objContext)
    {
        String strTemp = objContext.strExpression;

        objContext.strExpression = strTemp.replace("mm", "06");
    }

}
```

```java
package design.behavioral.patterns.interpreter;

public class ClsYearExpression extends ClsAbstractExpression
{

    @Override
    public void evaluate(ClsContext objContext)
    {
         String strTemp = objContext.strExpression;

         objContext.strExpression = strTemp.replace("yyyy", "2009");
    }

}
```

```java
package design.behavioral.patterns.interpreter;

public class ClsSeperator extends ClsAbstractExpression
{
    @Override
    public void evaluate(ClsContext objContext)
    {
        String strTemp = objContext.strExpression;

        objContext.strExpression = strTemp.replace(" ", "-");
    }
}
```

Now that we have separated expression parsing logic in different classes, let's look at how the client will use the iterator logic. The client first passes the date grammar format to the context class. Depending on the date format we now start adding the expressions in a collection. So if we find a "DD" we add the "ClsDayExpression", if we find "MM" we add "ClsMonthExpression" and so on. Finally we just loop and call the "evaluate" method. Once all the evaluate methods are called we display the output.

```
package design.behavioral.patterns.interpreter;

import java.util.ArrayList;

public class Test
{

    public static void main(String s[])
    {
        //ClsContext objContext = new ClsContext("yyyy mm dd");
        ClsContext objContext = new ClsContext("dd mm yyyy");
        String []a = objContext.strExpression.split(" ");

        List<ClsAbstractExpression> obj = new ArrayList<ClsAbstractExpression>();

        for(String str : a)
        {
            if(str.equals("dd"))
            {
                obj.add(new ClsDayExpression());
            }
            else if (str.equals("mm"))
            {
                obj.add(new ClsMonthExpression());
            }
            else if(str.equals("yyyy"))
            {
                obj.add(new ClsYearExpression());
            }
        }
        obj.add(new ClsSeperator());

        for(ClsAbstractExpression objAbstract : obj)
        {
            objAbstract.evaluate(objContext);
        }
        System.out.println(objContext.strExpression);
    }
}
```

Iterator Pattern

The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The Iterator is one of the simplest and most frequently used of the design patterns. The Iterator pattern allows you to move through a list or collection of data using a standard interface without having to know the details of the internal representations of that data. In addition you can also define special iterators that perform some special processing and return only specified elements of the data collection.

Iterator pattern allows sequential access of elements without exposing the inside code. Let's say you have a collection of records which you want to browse sequentially and also maintain the current place which recordset is browsed, then the answer is iterator pattern. It's the most common and unknowingly used pattern. Whenever you use a "foreach" (It allows us to loop through a collection sequentially) loop you are already using iterator pattern to some extent.

Example:

In following figure, we have the "ClsIterator" class which has collection of customer classes. So we have defined an array list inside the "ClsIterator" class and a "fillObjects" method which loads the array list with data. The customer collection array list is private and customer data can be looked up by using the index of the array list. So we have public function like "getByIndex" (which can look up using a particular index), "prev" (Gets the previous customer in the collection, "next" (Gets the next customer in the collection), "getFirst" (Gets the first customer in the collection) and "getLast" (Gets the last customer in the collection).

```
package design.behavioral.patterns.iterator;

public class ClsCustomer
{
    public String strCustomerName = "";
    public String strCode = "";

    public ClsCustomer(String strName, String code)
    {
        strCustomerName = strName;
        strCode = code;
    }
}
```

```
package design.behavioral.patterns.iterator;

import java.util.ArrayList;

public class ClsIterator
{
    private int intCurrentIndex = 0;
    private List<ClsCustomer> objArray = new ArrayList<ClsCustomer>();

    public void fillObjects()
    {   ClsCustomer obj1 = new ClsCustomer("Pawan", "8126");
        objArray.add(obj1);

        ClsCustomer obj2 = new ClsCustomer("Tom", "3456");
        objArray.add(obj2);

        ClsCustomer obj3 = new ClsCustomer("Jack", "7654");
        objArray.add(obj3);
    }

    public ClsCustomer getByIndex(int intIndex) throws Exception
    {
        if(intIndex > objArray.size() || intIndex == -1)
        {
            throw new Exception("End or Begin");
        }
        intCurrentIndex = intIndex;
        return objArray.get(intIndex);
    }

    public ClsCustomer prev()
    {   int intTemp = intCurrentIndex;
        intTemp--;
        return objArray.get(intTemp);
    }

    public ClsCustomer next()
    {   int intTemp = intCurrentIndex;
        intTemp++;
        return objArray.get(intTemp);
    }

    public ClsCustomer first()
    {       return objArray.get(0); }

    public ClsCustomer last()
    {       return objArray.get(objArray.size()-1);}
}
```

So the client is exposed only these functions. These functions take care of accessing the collection sequentially and also it remembers which index is accessed. Below figures shows how the "objIterator" object which is created from class "ClsIterator" is used to display next, previous, last, first and customer by index.

```
package design.behavioral.patterns.iterator;

public class Test
{
    public static void main(String arg[])
    {
        try
        {
            ClsIterator objIterator = new ClsIterator();
            objIterator.fillObjects();
            display(objIterator.getByIndex(1));
            display(objIterator.first());
            display(objIterator.next());
            display(objIterator.prev());
            display(objIterator.last());
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

    }

    public static void display(ClsCustomer objCustomer)
    {
        sop(objCustomer.strCustomerName + "      " + objCustomer.strCode);
    }
    public static void sop(Object o)
    {
        System.out.println(o);
    }
}
```

Where to use

Use to access the elements of an aggregate object sequentially. Java's collections like ArrayList and HashMap have implemented the iterator pattern.

Benefits

1. The same iterator can be used for different aggregates.
2. Allows you to traverse the aggregate in different ways depending on your needs.
3. It encapsulates the internal structure of how the iteration occurs.
4. Don't need to bloat your class with operations for different traversals.

Drawbacks/consequences

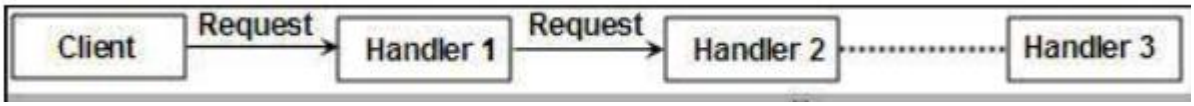
Not thread safe unless it's a robust iterator that allows insertions and deletions. This can be be solved by letting the Iterator use a Memento to capture the state of an iteration.

Chain-of-responsibility Pattern

The Chain-of-responsibility pattern lets more than one object handle a request without mutual knowledge. We avoid coupling between the sender of a request and the possible receivers. We place all receivers in a chain which lets the receiving objects pass the request along to the next receiver in the chain until one receiver handles it, or the end of the chain is reached.

The Chain of Responsibility pattern allows a number of classes to attempt to handle a request, without any of them knowing about the capabilities of the other classes. It provides a loose coupling between these classes; the only common link is the request that is passed between them. The request is passed along until one of the classes can handle it.

Chain of responsibility is used when we have series of processing which will be handled by a series of handler logic. There are situations when a request is handled by series of handlers. So the request is taken up by the first handler, he either can handle part of it or cannot, once done he passes to the next handler down the chain. This goes on until the proper handler takes it up and completes the processing.



Example:

Following figure showing the three process classes which inherit from the same abstract class. One of the important points to be noted is that every process points to the next process which will be called. So in the process class we have aggregated one more process object called as “objProcess”. Object “objProcess” points to next process which should be called after this process is complete.

```
package design.behavioral.patterns.chainofresponsibility;

public abstract class IProcess
{
    protected IProcess objProcess;

    public void setProcess(IProcess process)
    {
        objProcess = process;
    }

    public abstract void runProcess();
}
```

```
package design.behavioral.patterns.chainofresponsibility;

public class ClsProcess1 extends IProcess
{

    @Override
    public void runProcess()
    {
          System.out.println("Run Process 1");
          if(objProcess !=null)
          {
              objProcess.runProcess();
          }
    }

}
```

```
package design.behavioral.patterns.chainofresponsibility;

public class ClsProcess2 extends IProcess
{

    @Override
    public void runProcess()
    {
          System.out.println("Run Process 2");
          if(objProcess !=null)
          {
              objProcess.runProcess();
          }

    }

}
```

```
package design.behavioral.patterns.chainofresponsibility;

public class ClsProcess3 extends IProcess
{

    @Override
    public void runProcess()
    {
          System.out.println("Run Process 3");
          if(objProcess != null)
          {
              objProcess.runProcess();
          }

    }

}
```

It's time to call the classes in the client. So we create all the process objects for process1, process2 and process3. Using the "setProcess" method we define the link list of process objects. You can see we have

set process2 as a link list to process1 and process2 to process3. Once this link list is established we run the process which in turn runs the process according to the defined link list.

```
package design.behavioral.patterns.chainofresponsibility;

public class Test
{
    public static void main(String args[])
    {
        IProcess obj1 = new ClsProcess1();
        IProcess obj2 = new ClsProcess2();
        IProcess obj3 = new ClsProcess3();

        obj1.setProcess(obj2);
        obj2.setProcess(obj3);
        obj1.runProcess();
    }
}
```

Where to use

- When more than one object may handle a request and the handler isn't known.
- When you want to issue a request to one of several objects without specifying the receiver explicitly.
- When the set of objects that can handle a request should be specified dynamically.

Benefits

- It reduces coupling.
- It increases the flexibility of handling a request.

Drawbacks/consequences

Reception isn't guaranteed since a request has no explicit receiver, there's no guarantee it will be handled unless the chain is configured properly.

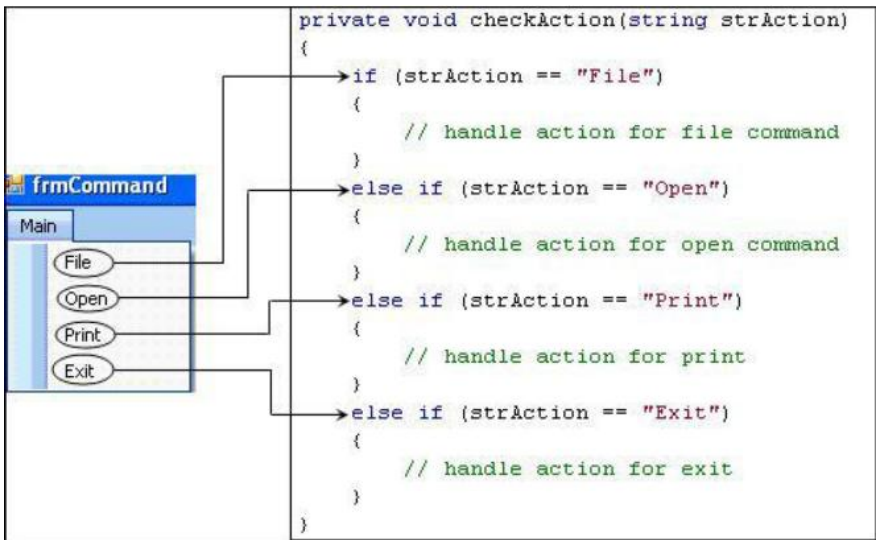
Command Pattern

The Command pattern is used to create objects that represent actions and events in an application. A command object encapsulates an action (event) and contains all information required to understand exactly what has happened. By passing the command object as a parameter we can, anywhere needed extract information about occurred actions and events.

The Chain of Responsibility pattern forwards requests along a chain of classes, but the Command pattern forwards a request only to a specific module. It encloses a request for a specific action inside an object and gives it a known public interface. It lets you give the client the ability to make requests without knowing anything about the actual action that will be performed, and allows you to change that action without affecting the client program in any way.

When you build a Java user interface, you provide menu items, buttons, and checkboxes and so forth to allow the user to tell the program what to do. When a user selects one of these controls, the program receives an *ActionEvent*, which it must trap by subclassing, the *actionPerformed* event.

Command pattern allows a request to exist as an object. Consider the following figure. It has different actions depending on which menu is clicked. So depending on which menu is clicked we have passed a string which will have the action text in the action string. Depending on the action string we will execute the action. The bad thing about the code is it has lot of "if" condition which makes the coding more cryptic.



Command pattern moves the above action in to objects. These objects when executed actually execute the command. As said previously every command is an object. We first prepare individual classes for every action i.e. exit, open, file and print. All the above actions are wrapped in to classes like Exit action is wrapped in "ClsExecuteExit", open action is wrapped in "ClsExecuteOpen", print action is wrapped in "ClsExecutePrint" and so on. All these classes are inherited from a common interface "IExecute".

Example

```
package design.behavioral.patterns.command;

public abstract class IExecute
{
    public String strCommand = "";
    public abstract void execute();
}
```

```
package design.behavioral.patterns.command;

public class ClsExecuteOpen extends IExecute
{
    public ClsExecuteOpen()
    {
        strCommand = "Open";
    }
    @Override
    public void execute()
    {
         System.out.println("Open Command");
    }
}
```

```
package design.behavioral.patterns.command;

public class ClsExecuteFile extends IExecute
{
    public ClsExecuteFile()
    {
        strCommand = "File";
    }
    @Override
    public void execute()
    {
         System.out.println("File Command");

    }
}
```

```
package design.behavioral.patterns.command;

public class ClsExecutePrint extends IExecute
{

    public ClsExecutePrint()
    {
        strCommand = "Print";
    }
    @Override
    public void execute()
    {
         System.out.println("Print Command");
    }

}
```

```
package design.behavioral.patterns.command;

public class ClsExecuteExit extends IExecute
{

    public ClsExecuteExit()
    {
        strCommand = "Exit";
    }
    @Override
    public void execute()
    {
         System.out.println("Exit Command");
    }

}
```

Using all the action classes we can now make the invoker. The main work of invoker is to map the action with the classes which have the action. So we have added all the actions in one collection i.e. the arraylist. We have exposed a method “getCommand” which takes a string and gives back the abstract object “IExecute”. The client code is now neat and clean. The entire “if” conditions are now moved to the “clsInvoker” class.

```
package design.behavioral.patterns.command;

import java.util.ArrayList;

public class ClsInvoker
{
    private List<IExecute> objArrayList = new ArrayList<IExecute>();

    public ClsInvoker()
    {
        objArrayList.add(new ClsExecuteFile());
        objArrayList.add(new ClsExecuteOpen());
        objArrayList.add(new ClsExecutePrint());
        objArrayList.add(new ClsExecuteExit());
    }

    public IExecute getCommand(String strCommand)
    {

        for(IExecute obj : objArrayList)
        {
            if(obj.strCommand.equals(strCommand))
            {
                return obj;
            }
        }

        return null;
    }
}
```

```
package design.behavioral.patterns.command;

public class Test
{
    public static void main(String arg[])
    {
        ClsInvoker objInvoker = new ClsInvoker();
        display(objInvoker.getCommand("Open"));
        display(objInvoker.getCommand("File"));
        display(objInvoker.getCommand("Print"));
        display(objInvoker.getCommand("Exit"));
    }

    public static void display(IExecute obj)
    {
        sop(obj.strCommand);
    }

    public static void sop(Object o)
    {
        System.out.println(o);
    }
}
```

Where to use

- Where you want an action that can be represented in many ways, like dropdown menu, buttons and popup menu.
- To create undo/redo functionality.

Benefits

- A command object is a possible storage for procedure parameters. It can be used while assembling the parameters for a function call and allows the command to be set aside for later use.
- A class is a suitable place to collect code and data related to a specific action or event.
- It allows the reaction to a command to be executed some time after it has occurred.
- Command objects enable data structures containing multiple commands.
- Command objects support undoable operations, provided that the command objects are stored (for example in a linked list).

Drawbacks/consequences

The main disadvantage of the Command pattern seems to be a proliferation of little classes that clutter up the program. However, even in the case where we have separate click events, we usually call little

private methods to carry out the actual function. It turns out that these private methods are just about as long as our little classes, so there is frequently little difference in complexity between building the Command classes and just writing more methods. The main difference is that the Command pattern produces little classes that are much more readable.

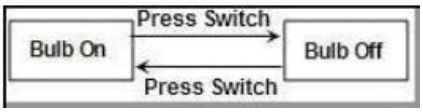
State Pattern

The State pattern allows an object to alter its behavior when it's internal state changes. By using inheritance and letting subclasses represent different states and functionality we can switch during runtime. This is a clean way for an object to partially change its type at runtime.

The State pattern is used when you want to have an enclosing class switch between a number of related contained classes, and pass method calls on to the current contained class. *Design Patterns* suggests that the State pattern switches between internal classes in such a way that the enclosing object appears to change its class.

Many programmers have had the experience of creating a class which performs slightly different computations or displays different information based on the arguments passed into the class. This frequently leads to some sort of *switch* or *if-else* statements inside the class that determine which behavior to carry out. It is this inelegance that the State pattern seeks to replace.

State pattern allows an object to change its behavior depending on the current values of the object. Following figure is an example of a bulb operation. If the state of the bulb is off and you press the switch the bulb will turn on. If the state of bulb is on and you press the switch the bulb will be off. So the behavior changes depending on the state.



Example

Now let's try to implement the same bulb sample. Following figure shows the ClsState class. We have made a class called as "ClsState" which has an enum with two state constants "on" and "off". We have defined a method "pressSwitch" which toggles its state depending on the current state.

In the following figure we have defined a client which consumes the "ClsState" class and calls the "pressSwitch" method. We have displayed the current status using the "getStatus" method. When we click the press switch it toggles to the opposite state of what we have currently.

```java
package design.behavioral.patterns.state;

public class ClsState
{
    enum BulbState
    {
        on,off
    }

    public BulbState currentBulbState = BulbState.off;

    public String getBulbState()
    {
        return currentBulbState.name();
    }

    public void pressSwitch()
    {
        if(currentBulbState.equals(BulbState.off))
        {
            currentBulbState = BulbState.on;
        }
        else if(currentBulbState.equals(BulbState.on))
        {
            currentBulbState = BulbState.off;
        }
    }
}
```

```java
package design.behavioral.patterns.state;

public class Test
{
    public static void main(String args[])
    {
        ClsState objState = new ClsState();
        sop(objState.getBulbState());
        objState.pressSwitch();
        sop(objState.getBulbState());
        objState.pressSwitch();
        sop(objState.getBulbState());
        objState.pressSwitch();
        sop(objState.getBulbState());
        objState.pressSwitch();
        sop(objState.getBulbState());
    }

    public static void sop(Object o)
    {
        System.out.println(o);
    }
}
```

Where to use

- When we need to define a "context" class to present a single interface to the outside world. By defining a State abstract base class.
- When we want to represent different "states" of a state machine as derived classes of the State base class.

Benefits

- Cleaner code when each state is a class instead.
- Use a class to represent a state, not a constant.

Drawbacks/consequences

- Generates a number of small class objects, but in the process, simplifies and clarifies the program.
- Eliminates the necessity for a set of long, look-alike conditional statements scattered throughout the code.

Strategy Pattern

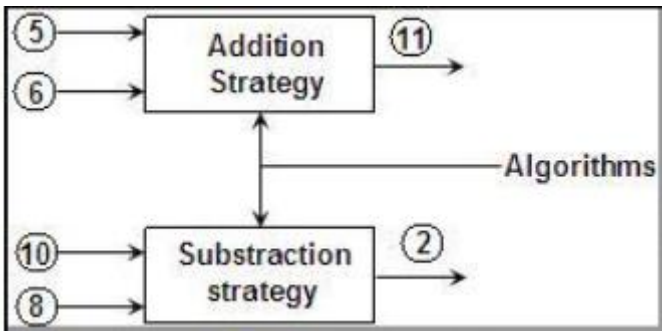
Use strategy when you need to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Related patterns include State, Flyweight, Decorator, and Composite.

The Strategy pattern is much like the State pattern, but a little different in intent (aim). The Strategy pattern consists of a number of related algorithms encapsulated in a driver class called the Context. Your client program can select one of these differing algorithms or in some cases the Context might select the best one for you. The intent (aim), like the State pattern, is to switch easily between algorithms without any monolithic conditional statements.

The difference between State and Strategy is that the user generally chooses which of several strategies to apply and that only one strategy at a time is likely to be instantiated and active within the Context class. But in state pattern it is likely that all of the different States will be active at once and switching may occur frequently between them. In addition, Strategy encapsulates several algorithms that do more or less the same thing, while State pattern encapsulates related classes that each does something somewhat different. Finally, the concept of transition between different states is completely missing in the Strategy pattern.

Strategy pattern are algorithms inside a class which can be interchanged depending on the class used. This pattern is useful when you want to decide on runtime which algorithm to be used.

Let's try to see an example of how strategy pattern works practically. We have strategies like add and subtract. Following figure shows the same in a pictorial format. It takes two numbers and the depending on the strategy it gives out results. So if it's an addition strategy it will add the numbers, if it's a subtraction strategy it will give the subtracted results. These strategies are nothing but algorithms. Strategy pattern are nothing but encapsulation of algorithms inside classes.



Example:

Below figure shows how the “add” is encapsulated in the “ClsAddStatergy” class and “substract” in the “ClsSubstractStatergy” class. Both these classes inherit from “ClsStratergy” defining a “calculate” method for its child classes.

```
package design.behavioral.patterns.strategy;

public abstract class ClsStrategy
{
    public abstract int calculate(int a, int b);
}
```

```
package design.behavioral.patterns.strategy;

public class ClsAddStrategy extends ClsStrategy
{
    public int calculate(int a, int b)
    {
        return a+b;
    }
}
```

```
package design.behavioral.patterns.strategy;

public class ClsSubstractStrategy extends ClsStrategy
{
    @Override
    public int calculate(int a, int b)
    {
        return a-b;
    }
}
```

Now we define a wrapper class called as “ClsMaths” which has a reference to the “ClsStatergy” class. This class has a “setStatergy” method which sets the strategy to be used.

```
package design.behavioral.patterns.strategy;

public class ClsMaths
{
    public int number1 = 0;
    public int number2 = 0;

    private ClsStrategy objStrategy;

    public void setStrategy(ClsStrategy obj)
    {
        objStrategy = obj;
    }

    public int calculate()
    {
        return objStrategy.calculate(number1, number2);
    }
}
```

Below figure shows how the wrapper class is used and the strategy object is set on runtime using the "setStatergy" method.

```
package design.behavioral.patterns.strategy;

public class Test
{
    public static void main(String args[])
    {
        ClsMaths a = new ClsMaths();
        a.number1 = 10;
        a.number2 = 5;
        a.setStrategy(new ClsAddStrategy());
        sop(a.calculate());

        a.setStrategy(new ClsSubstractStrategy());
        sop(a.calculate());
    }

    public static void sop(Object o)
    {
        System.out.println(o);
    }
}
```

Where to use

- When you need to use one of several algorithms dynamically.
- When you want to configure a class with one of many related classes (behaviors).
- When an algorithm uses data that clients shouldn't know about.

Benefits

- Reduces multiple conditional statements in a client.
- Hides complex, algorithmic-specific data from the client.
- Provides an alternative to sub classing.
- Can be used to hide data that an algorithm uses that clients shouldn't know about.

Drawbacks/consequences

- Clients must be aware of different strategies. A client must understand how strategies differ before it can select the appropriate one.
- Increases the number of objects in an application.

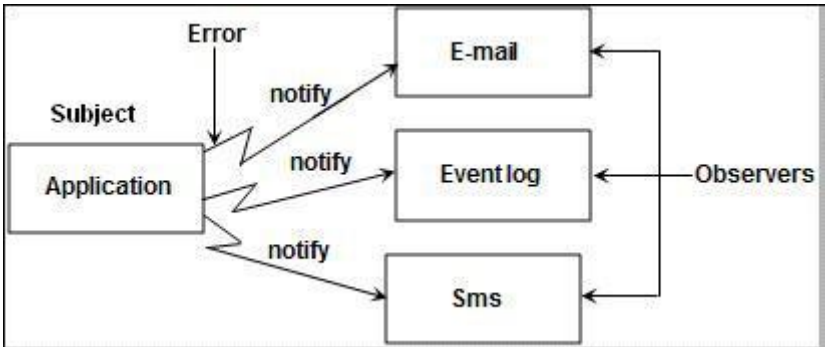
Observer Pattern

An observer is a structural pattern that enables publish/subscribe functionality. This is accomplished by an autonomous object, publisher that allows other objects to attach or detach their subscription as they like. The pattern does not impose any limit to the number of observers that can attach, or subscribe, themselves for notification on future changes in the publisher's state.

A real example of this pattern, we often would like to display data in more than one form at the same time and have all of the displays reflect any changes in that data. For example, you might represent stock price changes both as a graph and as a table or list box. Each time the price changes, we'd expect both representations to change at once without any action on our part.

Note: Observer pattern is very much similar to Reactor design pattern. Reactor design pattern de-multiplexes events (separating single stream into multiple streams) and dispatches them to registered object handlers. The reactor pattern is similar to an observer pattern (aka publisher and subscriber design pattern), but an observer pattern handles only a single source of events (i.e. a single publisher with multiple subscribers) where a reactor pattern handles multiple event sources (i.e. multiple publishers with multiple subscribers). The intent of an observer pattern is to define a one-to-many dependency so that when one object (i.e. the publisher) changes its state, all its dependents (i.e. all its subscribers) are notified and updated correspondingly. Reactor pattern is used by Java NIO (non-blocking or new IO)

Observer pattern helps us to communicate between parent class and its associated or dependent classes. There are two important concepts in observer pattern "Subject" and "Observers". The subject sends notifications while observers receive notifications if they are registered with the subject. Below figure shows how the application (subject) sends notification to all observers (email, event log and SMS). You can map this example to publisher and subscriber model. The publisher is the application and subscribers are email, event log and sms.



Example

Let's try to code the same example. First have a look at the subscribers / notification classes. Following figures shows the same in a pictorial format. So we have a common interface for all subscribers i.e. "INotification" which has a "sendNotification" method. This interface "INotification" is implemented by all concrete notification classes. All concrete notification classes define their own notification methodology. For the current scenario we have just displayed a print saying the particular notification is executed.

```
package design.behavioral.patterns.observer;

public interface INotification
{
    public void sendNotification();
}
```

```
package design.behavioral.patterns.observer;

public class ClsEmailNotification implements INotification
{
    @Override
    public void sendNotification()
    {
        System.out.println("Email notification executed");
    }
}
```

```
package design.behavioral.patterns.observer;

public class ClsEventNotification implements INotification
{
    @Override
    public void sendNotification()
    {
        System.out.println("Event notification executed");
    }
}
```

As said previously there are two sections in an observer pattern one is the observer/subscriber whom we have covered in the previous section and second is the publisher or the subject.

The publisher has a collection (arraylist) which will have all subscribers added who are interested in receiving the notifications. Using "addNotification" and "removeNotification" we can add and remove the subscribers from the arraylist. "sendNotificationAll" method loops through all the subscribers and send the notification.

```
package design.behavioral.patterns.observer;

import java.util.ArrayList;

public class ClsNotifier
{
    private List<INotification> objNotifications = new ArrayList<INotification>();

    public void addNotification(INotification obj)
    {
        objNotifications.add(obj);
    }

    public void removeNotification(INotification obj)
    {
        objNotifications.remove(obj);
    }

    public void sendNotificationAll()
    {
        for(INotification obj : objNotifications)
        {
            obj.sendNotification();
        }
    }
}
```

Now that we have an idea about the publisher and subscriber classes lets code the client and see observer in action. Below is a code for observer client snippet. So first we create the object of the notifier which has collection of subscriber objects. We add all the subscribers who are needed to be notified in the collection. Now if the customer code length is above 10 characters then it will notify all the subscribers about the same.

```
package design.behavioral.patterns.observer;

public class Test
{
    public static void main(String args[])
    {
        String strCustomerCode = "";

        ClsNotifier objNotifier = new ClsNotifier();
        ClsEmailNotification objEmailNotification = new ClsEmailNotification();
        ClsEventNotification objEventNotification = new ClsEventNotification();

        objNotifier.addNotification(objEmailNotification);
        objNotifier.addNotification(objEventNotification);

        strCustomerCode = "Setting the length of customer code more than 10";

        if(strCustomerCode.length() > 10)
        {
        objNotifier.sendNotificationAll();
        }
    }
}
```

Where to use

When an object wants to publish information and many objects will need to receive that information.

Benefits

Makes for a loose coupling between publisher and subscriber as the publisher does not need to know who or how many subscribers there will be.

Drawbacks/consequences

In a complex scenario there may be problems to determining whether the update to the publisher is of relevance to all subscribers or just some of them. Sending an update signal to all subscribers might impose a communication overhead of not needed information.

Template Pattern

Whenever you write a parent class where you leave one or more of the methods to be implemented by derived classes, you are in essence using the Template pattern. The Template pattern formalizes the idea of defining an algorithm in a class, but leaving some of the details to be implemented in subclasses. In other words, if your base class is an abstract class, as often happens in these design patterns, you are using a simple form of the Template pattern.

In template pattern we have an abstract class which acts as a skeleton for its inherited classes. The inherited classes get the shared functionality. The inherited classes take the shared functionality and add enhancements to the existing functionality. In word or power point how we take templates and then prepare our own custom presentation using the base. Template classes works on the same fundamental.

Example

```
package design.behavioral.patterns.template;

public abstract class ClsCustomer
{
    private String customerCode;
    private String customerName;

    public String getCustomerCode()
    {
        return customerCode;
    }
    public void setCustomerCode(String customerCode)
    {
        this.customerCode = customerCode;
    }
    public String getCustomerName()
    {
        return customerName;
    }
    public void setCustomerName(String customerName)
    {
        this.customerName = customerName;
    }
}
```

```
package design.behavioral.patterns.template;

public class ClsCustomerAdd extends ClsCustomer
{
    public void add(String name, String code)
    {
        setCustomerName(name);
        setCustomerCode(code);

        System.out.println(getCustomerName());
        System.out.println(getCustomerCode());
    }
}
```

```
package design.behavioral.patterns.template;

public class ClsCustomerUpdate extends ClsCustomer
{
    public void update(String name)
    {
        setCustomerName(name);
        System.out.println(getCustomerName());
        System.out.println(getCustomerCode());
    }
}
```

Visitor Pattern

The visitor design pattern is a way of separating an algorithm from an object structure upon which it operates. A practical result of this separation provides the ability to add new operations (algorithm) to existing object structures without modifying those structures.

Visitor pattern allows us to change the class structure without changing the actual class. It is a way of separating the algorithm (logic) from the current data structure. Due to this you can add new logic to the current data structure without altering the structure. Second you can alter the structure without touching the logic.

Example

We have a simple Employee object which maintains a record of the employee's name, salary, vacation taken and number of sick days taken. A simple version of this class is:

```
public class Employee
{
   int sickDays, vacDays;
   float Salary;
   String Name;

 public Employee(String name, float salary,
             int vacdays, int sickdays)
 {
  vacDays = vacdays;        sickDays = sickdays;
  Salary = salary;          Name = name;
 }
 public String getName()   { return Name;      }
 public int getSickdays()  { return sickDays;  }
 public int getVacDays()   { return vacDays;   }
 public float getSalary()  { return Salary;    }
 public void accept(Visitor v) { v.visit(this); }
}
```

Now let's suppose that we want to prepare a report of the number of vacation days that all employees have taken so far this year. We could just write some code in the client to sum the results of calls to each Employee's *getVacDays* function, or we could put this function into a Visitor.

```
public abstract class Visitor
{
  public abstract void visit(Employee emp);
}
```

Notice that there is no indication what the Visitor does with each class in either the client classes or the abstract Visitor class. We can in fact write a whole lot of visitors that do different things to the classes in our program. The Visitor we are going to write first just sums the vacation data for all our employees:

```
public class VacationVisitor extends Visitor
{
 protected int total_days;
 public VacationVisitor()  {    total_days = 0;  }
 //-----------------------------
 public void visit(Employee emp)
 {
    total_days += emp.getVacDays();
 }
 //-----------------------------
 public int getTotalDays()
 {
    return total_days;
 }
}
```

Now, all we have to do to compute the total vacation taken is to go through a list of the employees and visit each of them, and then ask the Visitor for the total.

Where to Use

You should consider using a Visitor pattern when you want to perform an operation on the data contained in a number of objects that have different interfaces. Visitors are also valuable if you have to perform a number of unrelated operations on these classes.

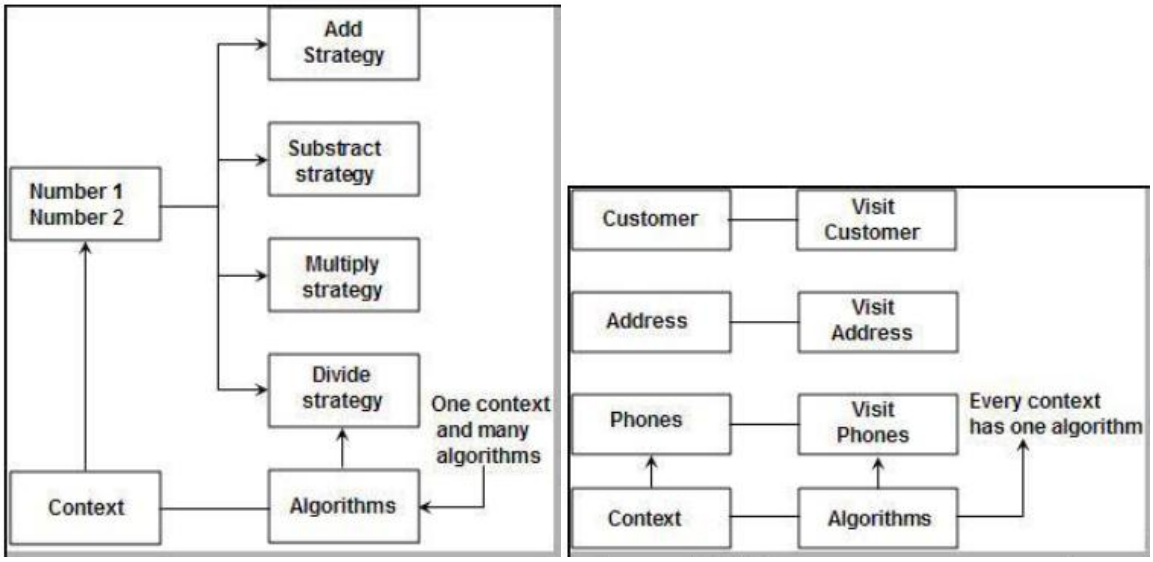
On the other hand visitors are a good choice only when you do not expect many new classes to be added to your program.

```
VacationVisitor vac = new VacationVisitor();
 for (int i = 0; i < employees.length; i++)
 {
  employees[i].accept(vac);
 }
 System.out.println(vac.getTotalDays());
```

Difference between visitor and strategy pattern?

Visitor and strategy look very much similar as they deal with encapsulating complex logic from data. We can say visitor is more general form of strategy. In strategy we have one context (single logical data) on which multiple algorithms operate. In strategy we have a single context and multiple algorithms work on it. In visitor we have multiple contexts and for every context we have an algorithm.

So in short strategy is a special kind of visitor. In strategy we have one data context and multiple algorithms while in visitor for every data context we have one algorithm associated. The basic criteria of choosing strategy or visitor depend on the relationship between context and algorithm. If there is one context and multiple algorithms then we go for strategy. If we have multiple contexts and multiple algorithms then we implement visitor algorithm.



Creational Patterns

Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

All the creational patterns define the best possible way in which an object can be created considering reuse and changeability. These describe the best way to handle instantiation. Hard coding the actual instantiation is a pitfall and should be avoided if reuse and changeability are desired. In such scenarios, we can make use of patterns to give this a more general and flexible approach.

Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.

All of the creational patterns deal with the best way to create instances of objects. This is important because your program should not depend on how objects are created and arranged. In Java, of course, the simplest way to create an instance of an object is by using the **new** operator.

```
Fred = new Fred(); //instance of Fred class
```

1. Abstract Factory - Creates an instance of several families of classes
2. Builder - Separates object construction from its representation
3. Factory Method - Creates an instance of several derived classes
4. Prototype - A fully initialized instance to be copied or cloned
5. Singleton - A class in which only a single instance can exist

Note: - The best way to remember Creational pattern is by remembering ABFPS (Abraham Became First President of States).

Abstract Factory Pattern

The Abstract Factory pattern is a creational pattern which is related to the Factory Method pattern, but it adds another level of abstraction. What this means is that the pattern encapsulates a group of individual concrete factory classes (as opposed to concrete factory methods which are derived in subclasses) which share common interfaces. The client software uses the Abstract Factory which provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern separates the implementation details of a set of objects from its general usage.

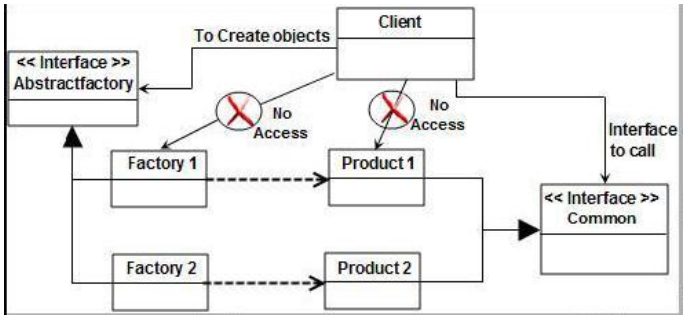
The Abstract Factory pattern is one level of abstraction higher than the factory pattern. You can use this pattern when you want to return one of several related classes of objects, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several factories.

One classic application of the abstract factory is the case where your system needs to support multiple "look-and-feel" user interfaces, such as Windows-9x, Motif or Macintosh. You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects. Then when you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

Abstract factory expands on the basic factory pattern. Abstract factory helps us to unite similar factory pattern classes in to one unified interface. So basically all the common factory patterns now inherit from a common abstract factory class which unifies them in a common class. All other things related to factory pattern remain same.

A factory class helps us to centralize the creation of classes and types. Abstract factory helps us to bring uniformity between related factory patterns which leads more simplified interface for the client.

Let's try to understand the details of how abstract factory patterns are actually implemented. We have the factory pattern classes (factory1 and factory2) tied up to a common abstract factory (AbstractFactory Interface) via inheritance. Factory classes stand on the top of concrete classes which are again derived from common interface. For instance in figure both the concrete classes "product1" and "product2" inherits from one interface i.e. "common". The client who wants to use the concrete class will only interact with the abstract factory and the common interface from which the concrete classes inherit.



Example:

We have scenario where we have UI creational activities for textboxes and buttons through their own centralized factory classes "ClsFactoryButton" and "ClsFactoryText". Both the factories "ClsFactoryButton" and "ClsFactoryText" inherits from the common factory "ClsAbstractFactory". Following figures shows how these classes are arranged and the client code for the same. One of the important points to be noted about the client code is that it does not interact with the concrete classes. For object creation it uses the abstract factory (ClsAbstractFactory) and for calling the concrete class implementation it calls the methods via the interface "InterfaceRender". So the "ClsAbstractFactory" class provides a common interface for both factories "ClsFactoryButton" and "ClsFactoryText".

```
package design.creational.patterns.abstractfactory;

public abstract class ClsAbstractFactory
{
    public static InterfaceRender getUIObject(int intTypeOfObject)
    {
        if(intTypeOfObject == 1)
        {
            return ClsFactoryTextBox.getTextBoxObject();
        }
        else
        {
            return ClsFactoryButton.getButtonObject();
        }
    }
}
```

```
package design.creational.patterns.abstractfactory;

public class ClsFactoryButton extends ClsAbstractFactory
{
    public static InterfaceRender getButtonObject()
    {
        return new ClsButton();
    }
}
```

```
package design.creational.patterns.abstractfactory;

public class ClsFactoryTextBox extends ClsAbstractFactory
{
    public static InterfaceRender getTextBoxObject()
    {
        return new ClsTextBox();
    }
}
```

Following Figures shows the common Interface for concrete classes' how the concrete classes inherits from a common interface 'InterFaceRender' which enforces the method 'render' in all the concrete classes.

```
package design.creational.patterns.abstractfactory;

public interface InterfaceRender
{
    void render();
}
```

```
package design.creational.patterns.abstractfactory;

public class ClsButton implements InterfaceRender
{
    @Override
    public void render()
    {
        System.out.println("Button is rendered");
    }
}
```

```
package design.creational.patterns.abstractfactory;

public class ClsTextBox implements InterfaceRender
{
    @Override
    public void render()
    {
        System.out.println("Text box is rendered");
    }
}
```

The final thing is the client code which uses the interface 'InterfaceRender' and abstract factory 'ClsAbstractFactory' to call and create the objects. One of the important points about the code is that it is completely isolated from the concrete classes. Due to this any changes in concrete classes like adding and removing concrete classes does not need client level changes.

```
package design.creational.patterns.abstractfactory;

import java.io.BufferedInputStream;

public class Test
{
    public static void main(String args[])
    {
        int intObjectType =0;

        InterfaceRender objUiObject;

        System.out.println("Enter the object type you want to render ");
        BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
        try
        {
            intObjectType = Integer.parseInt(bf.readLine());
        } catch (IOException e)
        {
            e.printStackTrace();
        }
        objUiObject = ClsAbstractFactory.getUIObject(intObjectType);
        objUiObject.render();

    }
}
```

Where to use

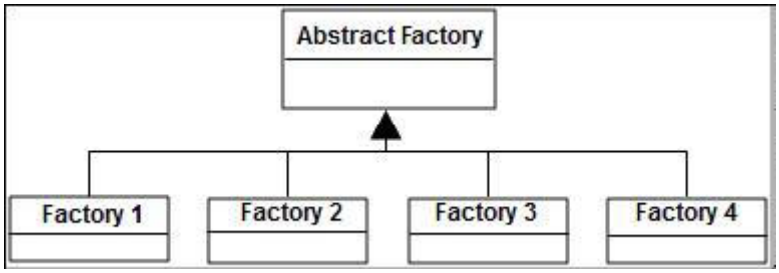
The pattern can be used where we need to create sets of objects that share a common theme and where the client only needs to know how to handle the abstract equivalence of these objects, i.e. the implementation is not important for the client. The Abstract Factory is often employed when there is a need to use different sets of objects and where the objects could be added or changed some time during the lifetime of an application.

Benefits

Use of this pattern makes it possible to interchange concrete classes without changing the code that uses them, even at runtime.

Drawbacks/consequences

One of the main drawbacks is the possibility of unnecessary complexity and extra work in the initial writing of the code.



Builder Pattern

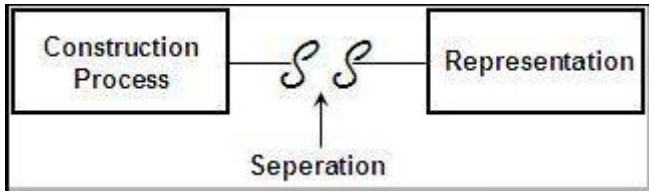
The Builder pattern can be used to ease the construction of a complex object from simple objects. The Builder pattern also separates the construction of a complex object from its representation so that the same construction process can be used to create another composition of objects. Related patterns include Abstract Factory and Composite.

The Factory Pattern returns one of several different subclasses depending on the data passed to in arguments to the creation methods. But suppose we don't want just a computing algorithm, but a whole different user interface depending on the data we need to display. A typical example might be your E-mail address book. You probably have both people and groups of people in your address book, and you would expect the display for the address book to change so that the People screen has places for first and last name, company, E-mail address and phone number.

On the other hand if you were displaying a group address page, you'd like to see the name of the group, its purpose, and a list of members and their E-mail addresses. You click on a person and get one display and on a group and get the other display.

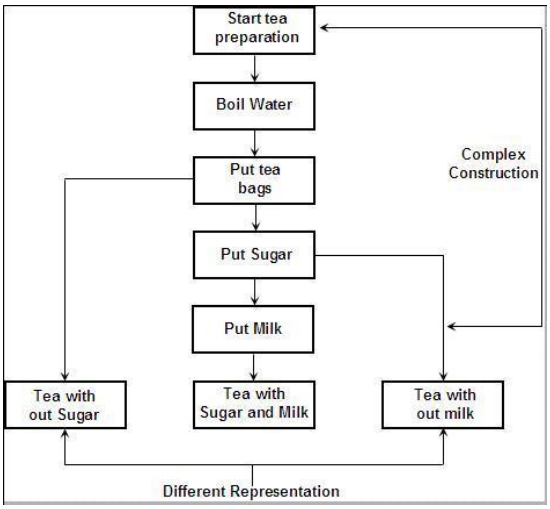
We'd like to see a somewhat different display of that object's properties. This is a little more than just a Factory pattern, because we aren't returning objects which are simple descendents of a base display object, but totally different user interfaces made up of different combinations of display objects.

Builder pattern helps us to separate the construction of a complex object from its representation so that the same construction process can create different representations. Builder pattern is useful when the construction of the object is very complex. The main objective is to separate the construction of objects and their representations. If we are able to separate the construction and representation, we can then get many representations from the same construction.

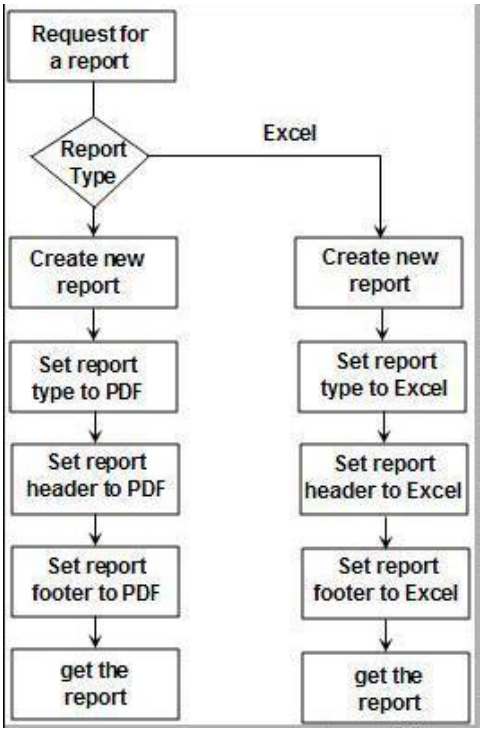


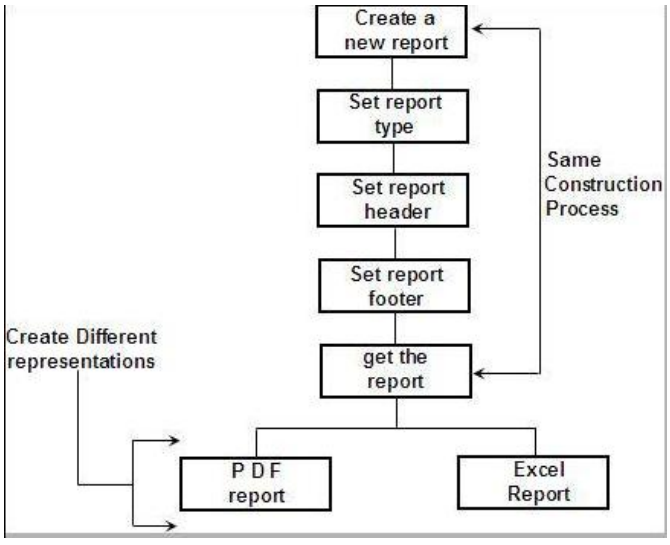
Let's take the example of "Tea preparation" sequence shown below to understand what we mean by construction and representation.

You can see from the figure that the same preparation steps we can get three representation of tea's (i.e. Tea without sugar, tea with sugar / milk and tea without milk). The construction process is same for all types of tea but they result in different representations.



Now let's take a real time example to see how builder can separate the complex creation and its representation. Consider we have application where we need the same report to be displayed in either "PDF" or "EXCEL" format. Following figure shows the series of traditional steps to achieve the same. Depending on report type a new report is created, report type is set, headers and footers of the report are set and finally we get the report for display.





Now let's take a different view of the problem as shown in figure above. The same flow is now analyzed in representations and common construction. The construction process is same for both the types of reports but they result in different representations.

There are three main parts when you want to implement builder patterns.

- **Builder** - Builder is responsible for defining the construction process for individual parts. Builder has those individual processes to initialize and configure the product.
- **Director** - Director takes those individual processes from the builder and defines the sequence to build the product.
- **Product** - Product is the final object which is produced from the builder and director coordination.

Example:

Let's use an abstract class HouseBuilder. Any subclass of HouseBuilder will follow the same steps to build house (that is to say to implement same methods in the subclass). Then we use a HouseDirector class to force these steps. The HouseClient (Test) orders the building of two houses, one wood house and one brick house. Even though the houses are of different types (wood and brick) they are built the same way, the construction process allows different representations for the object that is constructed.

```
package design.creational.patterns.builder;

public abstract class House
{
    public abstract String getRepresentation();
}
```

```
package design.creational.patterns.builder;

public class BrickHouse extends House
{
    @Override
    public String getRepresentation()
    {
        return "Building brick house";
    }
}
```

```
package design.creational.patterns.builder;

public class WoodHouse extends House
{
    @Override
    public String getRepresentation()
    {
        return "Building a wood house";
    }
}
```

```java
package design.creational.patterns.builder;

public abstract class HouseBuilder
{
    protected House house;
    public abstract House createHouse();
}
```

```java
package design.creational.patterns.builder;

public class BrickBuilder extends HouseBuilder
{
    @Override
    public House createHouse()
    {
        house = new BrickHouse();
        return house;
    }
}
```

```java
package design.creational.patterns.builder;

public class WoodBuilder extends HouseBuilder
{
    @Override
    public House createHouse()
    {
        house = new WoodHouse();
        return house;
    }
}
```

```java
package design.creational.patterns.builder;

public class HouseDirector
{
    public House constructHouse(HouseBuilder builder)
    {
        House house = builder.createHouse();
        sop(house.getRepresentation());
        return house;
    }

    public void sop(Object o)
    {
        System.out.println(o);
    }
}
```

```java
package design.creational.patterns.builder;

public class Test
{
    public static void main(String args[])
    {
        HouseDirector director = new HouseDirector();
        HouseBuilder woodBuilder = new WoodBuilder();
        House house = director.constructHouse(woodBuilder);

        HouseBuilder brickBuilder = new BrickBuilder();
        House house2 = director.constructHouse(brickBuilder);
    }
}
```

Where to use

- When the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
- When the construction process must allow different representations for the object that is constructed.
- When you want to insulate clients from the knowledge of the actual creation process and/or resulting product.

Benefits

- The built object is shielded from the details of its construction.
- Code for construction is isolated from code for representation and both are easy to replace without affecting the other.
- Gives you control over the construction process.
- Gives you the possibility to reuse and/or change the process and/or product independently.

Drawbacks/consequences

Need flexibility in creating various complex objects. Need to create complex, aggregate objects.

Factory Pattern

The Factory pattern provides a way to use an instance as an object factory. The factory can return an instance of one of several possible classes (in a subclass hierarchy), depending on the data provided to it.

One type of pattern that we see again and again in OO programs is the Factory pattern or class. A Factory pattern is one that returns an instance of one of several possible classes depending on the data provided to it. Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently and is optimized for different kinds of data.

Below is a code shot of a client who has different types of invoices. These invoices are created depending on the invoice type specified by the client.

```
if (intInvoiceType == 1)
{
   objinv = new clsInvoiceWithHeader();
}
else if (intInvoiceType == 2)
{
    objinv = new clsInvoiceWithOutHeaders();
}
```

There are two issues with the traditional way of creation of objects.

- First we have lots of “new” keyword scattered in the client. In other ways the client is loaded with lot of object creational activities which can make the client logic very complicated.
- Second issue is that the client needs to be aware of all types of invoices. So if we are adding one more invoice class type called as “InvoiceWithFooter” we need to reference the new class in the client and recompile the client also.

Example

This example shows how two different concrete Products are created using the ProductFactory. ProductA uses the superclass writeName method. ProductB implements writeName that modify the name.

```java
package design.creational.patterns.factory;

public abstract class Product
{
    public void writeName(String name)
    {
        System.out.println("my name is " + name);
    }
}
```

```java
package design.creational.patterns.factory;

public class ProductA extends Product
{

}
```

```java
package design.creational.patterns.factory;

public class ProductB extends Product
{
    public void writeName(String name)
    {
        StringBuilder sb = new StringBuilder();
        sb.append(name).append(" ").append("ProductB");
        System.out.println("my name is " + sb.toString());
    }
}
```

```java
package design.creational.patterns.factory;

public class ProductFactory
{
    Product createProduct(String type)
    {
        if(type.equals("B"))
        {
            return new ProductB();
        }
        else
        {
            return new ProductA();
        }
    }
}
```

```java
package design.creational.patterns.factory;

public class Test
{
    public static void main(String args[])
    {
        ProductFactory pf = new ProductFactory();
        Product productA = pf.createProduct("A");
        productA.writeName("product A");

        Product productB = pf.createProduct("B");
        productB.writeName("Product B");
    }
}
```

Where to use

- When a class can't anticipate which kind of class of object it must create.
- You want to localize the knowledge of which class gets created.
- When you have classes that is derived from the same subclasses, or they may in fact be unrelated classes that just share the same interface. Either way, the methods in these class instances are the same and can be used interchangeably.
- When you want to insulate the client from the actual type that is being instantiated.

Benefits

- The client does not need to know every subclass of objects it must create. It only needs one reference to the abstract class/interface and the factory object.
- The factory encapsulates the creation of objects. This can be useful if the creation process is very complex.

Drawbacks/consequences

There is no way to change an implementing class without a recompile.

Prototype Pattern

The Prototype pattern is basically the creation of new instances through cloning existing instances. By creating a prototype, new objects are created by copying this prototype.

Prototype pattern is used when creating an instance of a class is very time-consuming or complex in some way. Then, rather than creating more instances, you make copies of the original instance, modifying them as appropriate.

Prototypes can also be used whenever you need classes that differ only in the type of processing they offer, for example in parsing of strings representing numbers in different radix

It gives us a way to create new objects from the existing instance of the object. In one sentence we clone the existing object with its data. By cloning any changes to the cloned object does not affect the original object value. If you are thinking by just setting objects we can get a clone then you have mistaken it. By setting one object to other object we set the reference of object BYREF. So changing the new object also changed the original object.

Many times we want the new copy object & its changes should not affect the old object. The answer to this is prototype patterns.

Example:

```
package design.creational.patterns.prototype;

public class ClsCustomer implements Cloneable
{
    public String name = "customer1";

    public int code = 1001;

    public Object clone()
    {
        try
        {
            return super.clone();
        } catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
            return null;
        }
    }
}
```

```
package design.creational.patterns.prototype;

public class Test
{
    public static void main(String args[])
    {
        ClsCustomer c1 = new ClsCustomer();
        ClsCustomer c2 = (ClsCustomer) c1.clone();

        sop(c1.name + " " + c2.name);
        sop(c1.code + " " + c2.code);

        c2.name = "Tom";
        c2.code = 9999;

        sop(c1.name + " " + c2.name);
        sop(c1.code + " " + c2.code);
    }

    public static void sop(Object o)
    {
        System.out.println(o);
    }
}
```

Where to use

- When a system needs to be independent of how its objects are created, composed, and represented.
- When adding and removing objects at runtime.
- When specifying new objects by changing an existing objects structure.
- When configuring an application with classes dynamically.
- When trying to keep the number of classes in a system to a minimum.
- When state population is an expensive or exclusive process.

Benefits

- Speeds up instantiation of large, dynamically loaded classes.
- Reduced sub classing.

Drawbacks/consequences

Each subclass of Prototype must implement the clone operation. Could be difficult with existing classes with internal objects with circular references or which does not support copying.

Singleton Pattern

The Singleton pattern provides the possibility to control the number of instances (mostly one) that are allowed to be made. We also receive a global point of access to it (them).

There are many cases in programming where you need to make sure that there can be one and only one instance of a class. For example, your system can have only one window manager or print spooler, or a single point of access to a database engine.

The easiest way to make a class that can have only one instance is to embed a `static` variable inside the class that we set on the first instance and check for each time we enter the constructor. A static variable is one for which there is only one instance, no matter how many instances there are of the class.

There are situations in a project where we want only one instance of the object to be created and shared between the clients. No client can create an instance of the object from outside. There is only one instance of the class which is shared across the clients.

Below are the steps to make a singleton pattern:

1. Define the constructor as private.
2. Define the instances and methods as static.

Example

```
package design.creational.patterns.singleton;

public class FileLogger
{
    private static FileLogger logger;

    private static int counter = 0;

    private FileLogger()
    {

    }

    public static FileLogger getInstance()
    {
        if(logger == null)
        {
            counter++;
            logger = new FileLogger();
        }
        return logger;
    }

    public static int getCounter()
    {
        return counter;
    }
}
```

```
package design.creational.patterns.singleton;

public class Test
{
    public static void main(String s[])
    {
        FileLogger logger1 = FileLogger.getInstance();
        sop(FileLogger.getCounter());
        FileLogger logger2 = FileLogger.getInstance();
        sop(FileLogger.getCounter());
        sop(logger1 == logger2);
    }

    public static void sop(Object o)
    {
        System.out.println(o);
    }
}
```

Where to use

When only one instance or a specific number of instances of a class are allowed. Facade objects are often Singletons because only one Facade object is required.

Benefits

- Controlled access to unique instance.
- Reduced name space.
- Allows refinement of operations and representations.

Drawbacks/consequences

Singleton pattern is also considered an anti-pattern by some people, who feel that it is overused, introducing unnecessary limitations in situations where a sole instance of a class is not actually required.

Summary of Creational Patterns

· The Factory Pattern is used to choose and return an instance of a class from a number of similar classes based on data you provide to the factory.

· The Abstract Factory Pattern is used to return one of several groups of classes. In some cases it actually returns a Factory for that group of classes.

· The Builder Pattern assembles a number of objects to make a new object, based on the data with which it is presented. Frequently, the choice of which way the objects are assembled is achieved using a Factory.

· The Prototype Pattern copies or clones an existing class rather than creating a new instance when creating new instances is more expensive.

· The Singleton Pattern is a pattern that insures there is one and only one instance of an object, and that it is possible to obtain global access to that one instance.