Structural Patterns

Introduction

Structural patterns are concerned with how classes and objects can be arranged to form larger structures.

Structural class patterns use inheritance to compose interfaces or different implementations. For example, multiple inheritance can be used to combine features from two or more classes into a single class. This allows two or more independently developed class libraries to work together.

Structural object patterns specify a way to create new objects to realize new functionality. The flexibility of object composition allows us to change the composition at run-time, which is impossible with static class composition.

There are seven structural GOF patterns. They are:

- Adapter pattern
- Bridge pattern
- Composite pattern
- Decorator pattern
- Façade pattern
- Flyweight pattern
- Proxy pattern

Adapter Pattern

Intent:
To convert the interface of one class into another interface that the client expects. Adapter pattern allows two incompatible classes to communicate with one another.

Also knows as:
Wrapper

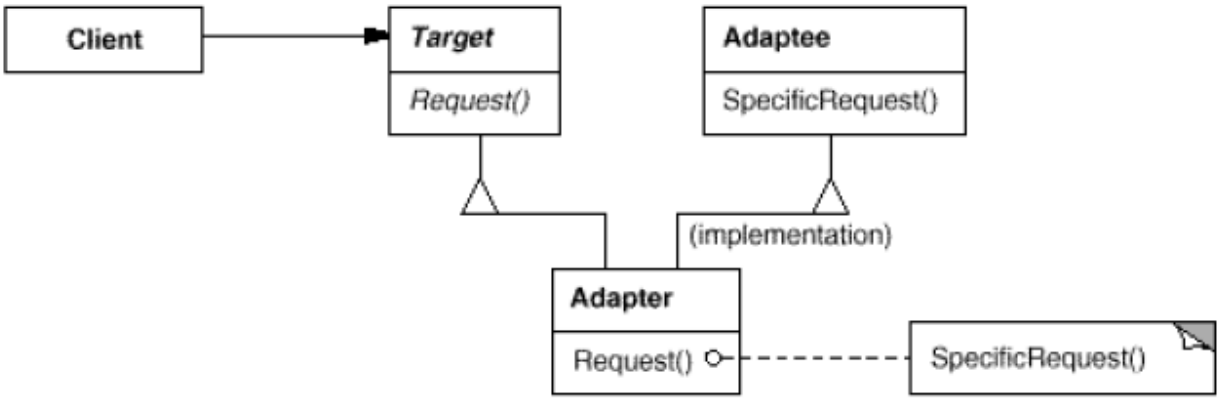
Motivation:
The adapter pattern is adaptation between classes and objects. Like any adapter in the real world it is used to be an interface, a bridge between two objects. In real world we have adapters for power supplies, adapters for camera memory cards, and so on. If you cannot plug in the camera memory card in your laptop, you can use and adapter. You plug the camera memory card in the adapter and the adapter in to laptop slot. That's it, it's really simple.

What about software development? It's the same. Can you imagine a situation when you have some class expecting some type of object and you have an object offering the same features, but exposing a different interface? Of course, you want to use both of them so you don't want to implement again one of them, and you don't want to change existing classes, so why not create an adapter.

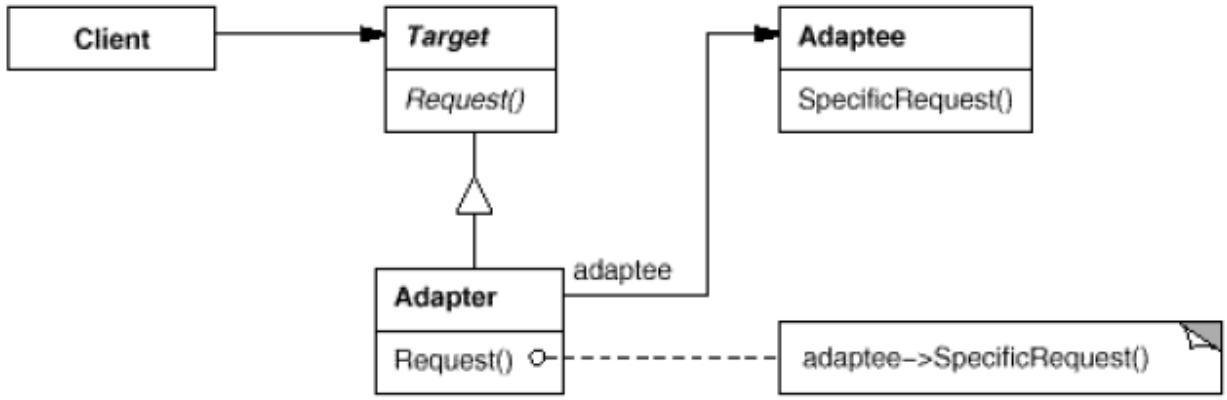
Applicability:
Use adapter pattern when:

- You want to use an existing class, and its interface is not what you needed.
- You want to create a reusable class that cooperates with the incompatible classes.
- You need to use several subclasses (object adapter only) by adapting to their interfaces (by sub classing each subclass) which is impractical. An object adapter can adapt the interface of their parent class.

Structure:
A class adapter uses multiple inheritance to adapt one interface to another. The structure of class adapter is shown below:



An object adapter relies on object composition. The structure of an object adapter is as shown below:



Participants:
Following are the participants in the adapter pattern:

- **Target**: Defines the domain specific interface the client uses.
- **Client**: Collaborates with the objects conforming to the *Target* interface.
- **Adaptee**: Defines an existing interface that needs to be adapted.
- **Adapter**: Adapts the interface of the *Adaptee* to the *Traget* interface.

Collaborations:
Client call operations on an *Adapter* instance. In turn, *Adapter* calls *Adaptee* operations that carry out the request.

Consequences:
Class and object adapters have different trade-offs.

A class adapter:

- Adapts *Adaptee* to *Target* by committing to a concrete *Adapter* class. As a consequence, a class adapter won't work when we want to adapt a class and its subclasses.
- Lets *Adapter* to override some of the behavior of the *Adaptee* since it is a subclass of *Adaptee*.
- Introduces only one object, and no additional pointer indirection is needed to get to the *Adaptee*.

An object adapter:

- Lets a single *Adapter* work with many *Adaptees* i.e the *Adaptee* itself and all of its subclasses. The *Adapter* can also add functionality to all *Adaptees* at once.
- Makes it harder to override *Adaptee* behavior.

Implementation:
Some of the issues to keep in mind while implementing adapter pattern are given below:

1. *Implementing class adapters in C++:* In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptee. Thus Adapter would be a subtype of Target but not of Adaptee.

2. *Pluggable adapters:* Let's look at three ways to implement pluggable adapters for the TreeDisplay widget which can lay out and display a hierarchical structure automatically.

The first step, which is common to all three of the implementations discussed here, is to find a "narrow" interface for Adaptee, that is, the smallest subset of operations that lets us do the adaptation. A narrow interface consisting of only a couple of operations is easier to adapt than an interface with dozens of operations. For TreeDisplay, the adaptee is any hierarchical structure. A minimalist interface might include two operations, one that defines how to present a node in the hierarchical structure graphically, and another that retrieves the node's children.

The narrow interface leads to three implementation approaches:
a. *Using abstract operations*: Define corresponding abstract operations for the narrow Adaptee interface in the TreeDisplay class. Subclasses must implement the abstract operations and adapt the hierarchically structured object.
b. *Using delegate objects:* In this approach, TreeDisplay forwards requests for accessing the hierarchical structure to a delegate object. TreeDisplay can use a different adaptation strategy by substituting a different delegate.
c. *Parameterized adapters:* The usual way to support pluggable adapters in Smalltalk is to parameterize an adapter with one or more blocks. The block construct supports adaptation without subclassing. A block can adapt a request, and the adapter can store a block for each individual request. In our example, this means TreeDisplay stores one block for converting a node into a GraphicNode and another block for accessing a node's children.

Sample code:
Adapter design pattern is one of the structural design pattern and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an Adapter. As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

So first of all we will have two classes – Volt (to measure volts) and Socket (producing constant volts of 120V).

//Volt.java
```
public class Volt {

    private int volts;

    public Volt(int v){
        this.volts=v;
    }

    public int getVolts() {
        return volts;
    }
```

```
    public void setVolts(int volts) {
        this.volts = volts;
    }

}
```

//Socket.java

```
public class Socket {

    public Volt getVolt(){
        return new Volt(120);
    }
}
```

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods.

//SocketAdapter.java

```
public interface SocketAdapter {

    public Volt get120Volt();

    public Volt get12Volt();

    public Volt get3Volt();
}
```

While implementing Adapter pattern, there are two approaches – class adapter and object adapter, however both these approaches produce same result.

1. Class Adapter – This form uses java inheritance and extends the source interface, in our case Socket class.
2. Object Adapter – This form uses Java Composition and adapter contains the source object.

Class Adapter Implementation:

//SocketClassAdapterImpl.java

```
//Using inheritance for adapter pattern
public class SocketClassAdapterImpl extends Socket implements SocketAdapter{

    @Override
    public Volt get120Volt() {
        return getVolt();
    }

    @Override
    public Volt get12Volt() {
        Volt v= getVolt();
        return convertVolt(v,10);
    }
```

```
    @Override
    public Volt get3Volt() {
        Volt v= getVolt();
        return convertVolt(v,40);
    }

    private Volt convertVolt(Volt v, int i) {
        return new Volt(v.getVolts()/i);
    }

}
```

Object Adapter Implementation:

//SocketObjectAdapterImpl.java

```
public class SocketObjectAdapterImpl implements SocketAdapter{

    //Using Composition for adapter pattern
    private Socket sock = new Socket();

    @Override
    public Volt get120Volt() {
        return sock.getVolt();
    }

    @Override
    public Volt get12Volt() {
        Volt v= sock.getVolt();
        return convertVolt(v,10);
    }

    @Override
    public Volt get3Volt() {
        Volt v= sock.getVolt();
        return convertVolt(v,40);
    }

    private Volt convertVolt(Volt v, int i) {
        return new Volt(v.getVolts()/i);
    }
}
```

Here is a test program to consume our adapter implementation:

//AdapterPatternTest.java

```
public class AdapterPatternTest {

    public static void main(String[] args) {

        testClassAdapter();
        testObjectAdapter();
```

```
    }

    private static void testObjectAdapter() {
        SocketAdapter sockAdapter = new SocketObjectAdapterImpl();
        Volt v3 = getVolt(sockAdapter,3);
        Volt v12 = getVolt(sockAdapter,12);
        Volt v120 = getVolt(sockAdapter,120);
        System.out.println("v3 volts using Object Adapter="+v3.getVolts());
        System.out.println("v12 volts using Object Adapter="+v12.getVolts());
        System.out.println("v120 volts using Object
      Adapter="+v120.getVolts());
    }

    private static void testClassAdapter() {
        SocketAdapter sockAdapter = new SocketClassAdapterImpl();
        Volt v3 = getVolt(sockAdapter,3);
        Volt v12 = getVolt(sockAdapter,12);
        Volt v120 = getVolt(sockAdapter,120);
        System.out.println("v3 volts using Class Adapter="+v3.getVolts());
        System.out.println("v12 volts using Class Adapter="+v12.getVolts());
        System.out.println("v120 volts using Class
      Adapter="+v120.getVolts());
    }

    private static Volt getVolt(SocketAdapter sockAdapter, int i) {
        switch (i){
        case 3: return sockAdapter.get3Volt();
        case 12: return sockAdapter.get12Volt();
        case 120: return sockAdapter.get120Volt();
        default: return sockAdapter.get120Volt();
        }
    }
}
```

Known uses:
Following are the examples in Java API where adapter pattern is used:

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)
- javax.xml.bind.annotation.adapters.XmlAdapter #marshal() and #unmarshal()

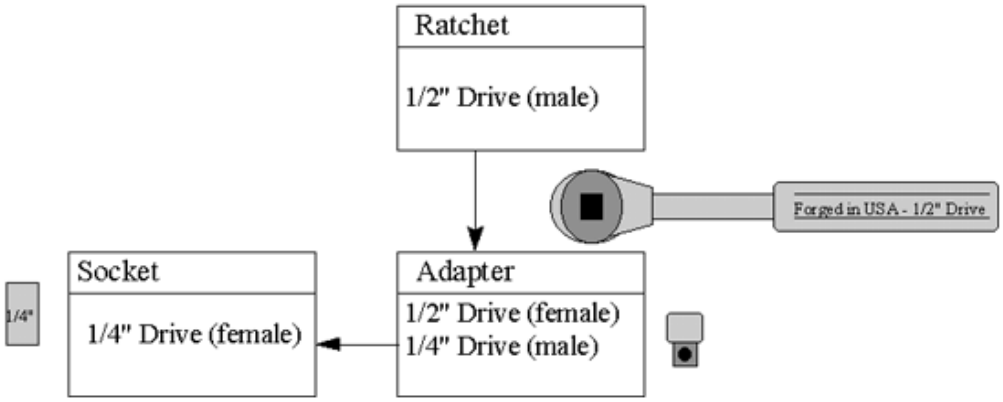
Related patterns:
Bridge has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object.

Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.

Proxy defines a representative or surrogate for another object and does not change its interface.

Non-software example:

The *Adapter* pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the *Adapter*. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



Bridge Pattern

Intent:
To decouple an abstraction from its implementation so that both can be changed independently.

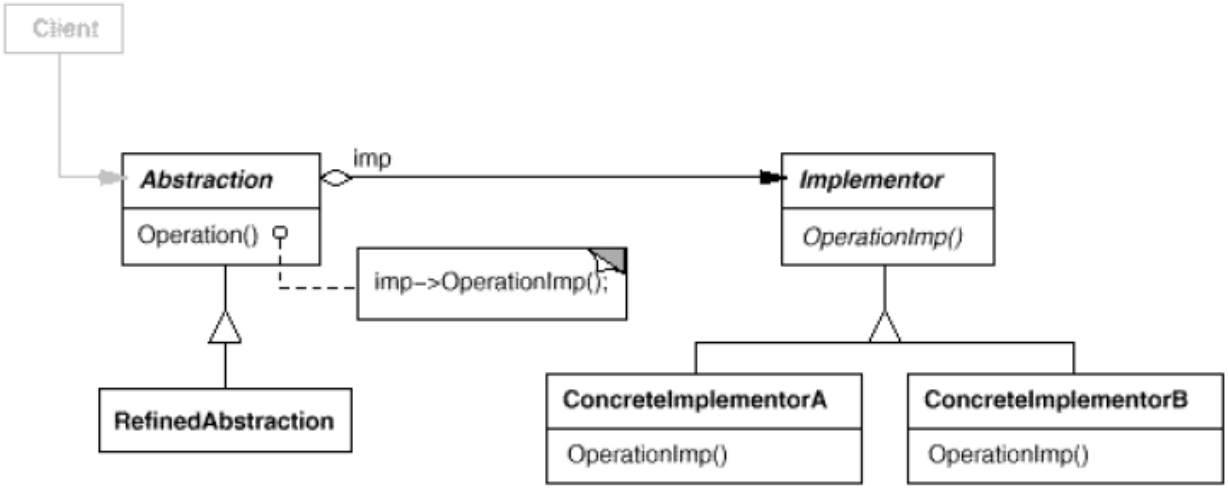
Also knows as:
Handle/Body

Motivation:
Sometimes an abstraction should have different implementations; consider an object that handles persistence of objects over different platforms using either relational databases or file system structures (files and folders). A simple implementation might choose to extend the object itself to implement the functionality for both file system and RDBMS. However this implementation would create a problem; Inheritance binds an implementation to the abstraction and thus it would be difficult to modify, extend, and reuse abstraction and implementation independently.

Applicability:
Use bridge pattern when:

- To avoid permanent binding between abstraction and implementation.
- Both abstractions and implementations should be extensible by creating subclasses.
- Changes in implementation should have no impact on client.
- To share and implementation among multiple objects, and this fact should be hidden from client.

Structure:
The structure of bridge pattern is as shown below:



Participants:
Following are the participants in bridge pattern:

- **Abstraction:** Defines the abstraction interface and maintains a reference to an object of type *Implementor*.

- **RefinedAbstraction:** Extends the interface defined by *Abstraction*.
- **Implementor:** Defines the interface for implementation classes.
- **ConcreteImplmentor:** Implements the *Implementor* interface and defines its concrete implementation.

Collaborations:
Abstraction forwards *client* requests to its *Implementor* object.

Consequences:
The bridge pattern has the following consequences:

1. *Decoupling interface and implementation:* An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.

2. *Improved extensibility*: You can extend the Abstraction and Implementor hierarchies independently.

3. *Hiding implementation details from clients:* You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism.

Implementation:
Following issues should be considered while implementing bridge pattern:

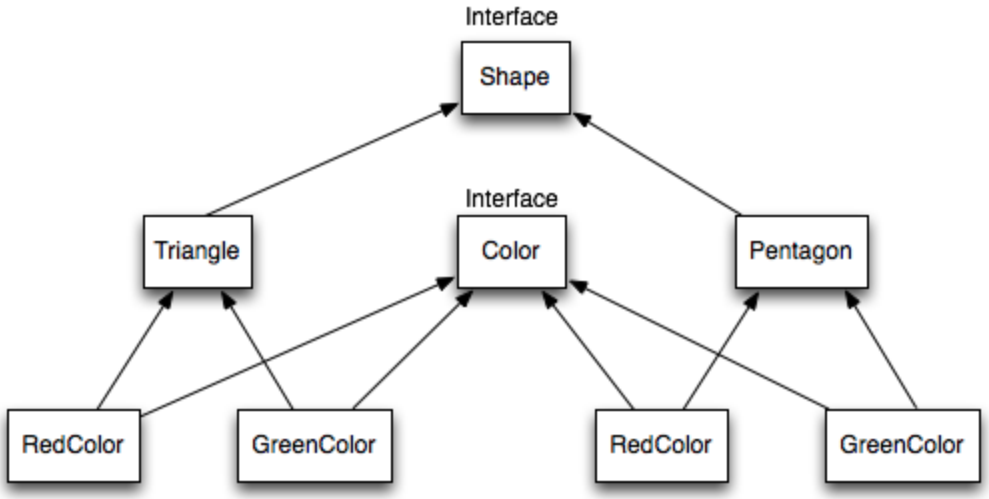
1. Only one Implementor: In situations where there's only one implementation, creating an abstract Implementor class isn't necessary. This is a degenerate case of the Bridge pattern; there's a one-to-one relationship between Abstraction and Implementor. Nevertheless, this separation is still useful when a change in the implementation of a class must not affect its existing clients—that is, they shouldn't have to be recompiled, just relinked.

2. Creating the right Implementor object: How, when, and where do you decide which Implementor class to instantiate when there's more than one? If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor; it can decide between them based on parameters passed to its constructor. If, for example, a collection class supports multiple implementations, the decision can be based on the size of the collection. A linked list implementation can be used for small collections and a hash table for larger ones.

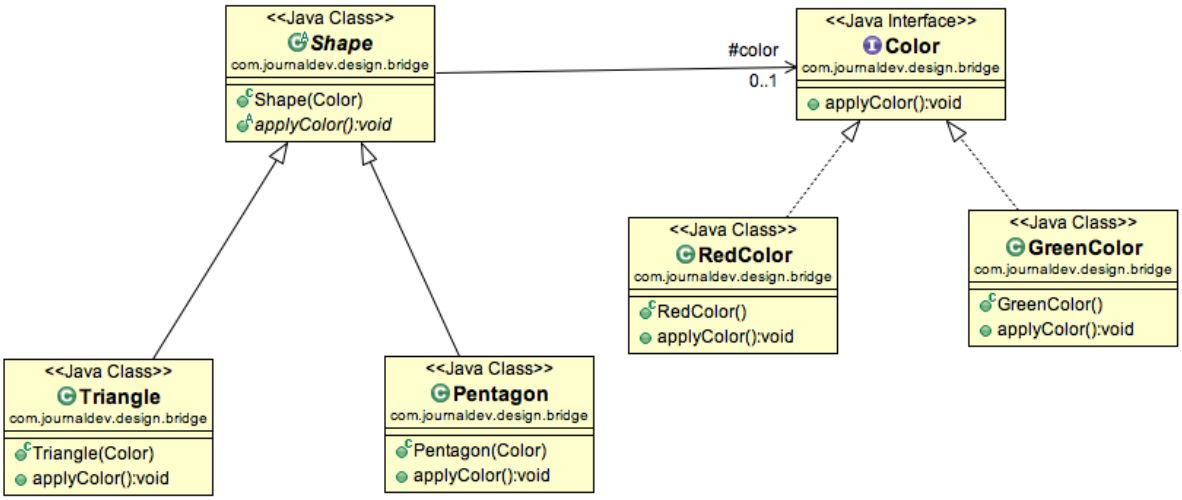
Sample code:
When we have interface hierarchies in both interfaces as well as implementations, then builder design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client programs.

The implementation of bridge design pattern follows the notion to prefer Composition over inheritance.

Let’s say we have an interface hierarchy in both interfaces and implementations like below image.



Now we will use bridge design pattern to decouple the interfaces from implementation and the UML diagram for the classes and interfaces after applying bridge pattern will look like below image.



Notice the bridge between `Shape` and `Color` interfaces and use of composition in implementing the bridge pattern.

Here is the java code for Shape and Color interfaces.

//Color.java

```
public interface Color {

    public void applyColor();
}
```

//Shape.java
```
public abstract class Shape {
    //Composition - implementor
    protected Color color;

    //constructor with implementor as input argument
    public Shape(Color c){
        this.color=c;
    }

    abstract public void applyColor();
}
```

We have Triangle and Pentagon implementation classes as below.

//Triangle.java
```
public class Triangle extends Shape{

    public Triangle(Color c) {
        super(c);
    }

    @Override
    public void applyColor() {
        System.out.print("Triangle filled with color ");
        color.applyColor();
    }

}
```

//Pentagon.java
```
public class Pentagon extends Shape{

    public Pentagon(Color c) {
        super(c);
    }

    @Override
    public void applyColor() {
        System.out.print("Pentagon filled with color ");
        color.applyColor();
    }

}
```

Here are the implementation classes for RedColor and GreenColor.

//RedColor.java
```
public class RedColor implements Color{

    public void applyColor(){
        System.out.println("red.");
    }
```

```
}
```

//GreenColor.java

```
public class GreenColor implements Color{

    public void applyColor(){
        System.out.println("green.");
    }
}
```

Lets test our bridge pattern implementation with a test program.

//BridgePatternTest.java

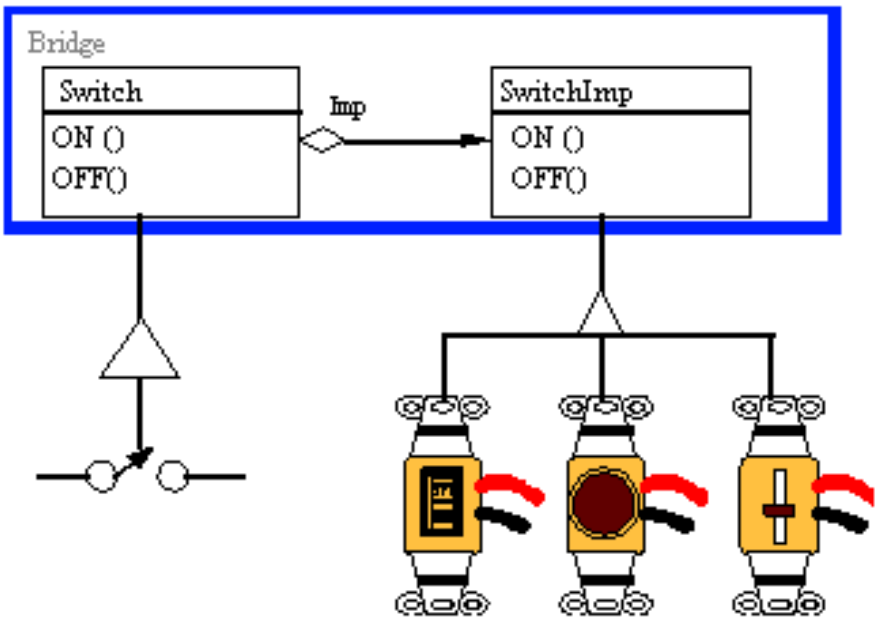
```
public class BridgePatternTest {

    public static void main(String[] args) {
        Shape tri = new Triangle(new RedColor());
        tri.applyColor();

        Shape pent = new Pentagon(new GreenColor());
        pent.applyColor();
    }

}
```

Known uses:
Following are the examples in Java API where bridge pattern is used:
new LinkedHashMap(LinkedHashSet<K>, List<V>) which returns an unmodifiable linked map which doesn't clone the items, but uses them. The java.util.Collections#newSetFromMap() and singletonXXX()

Related patterns:
An Abstract Factory can create and configure a particular Bridge.

The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

Non-software example:
The *Bridge* pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the *Bridge.* The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, a simple two position switch, or a variety of dimmer switches.


Bridge
Switch
ON ()
OFF()
Imp
SwitchImp
ON ()
OFF()

Composite Pattern

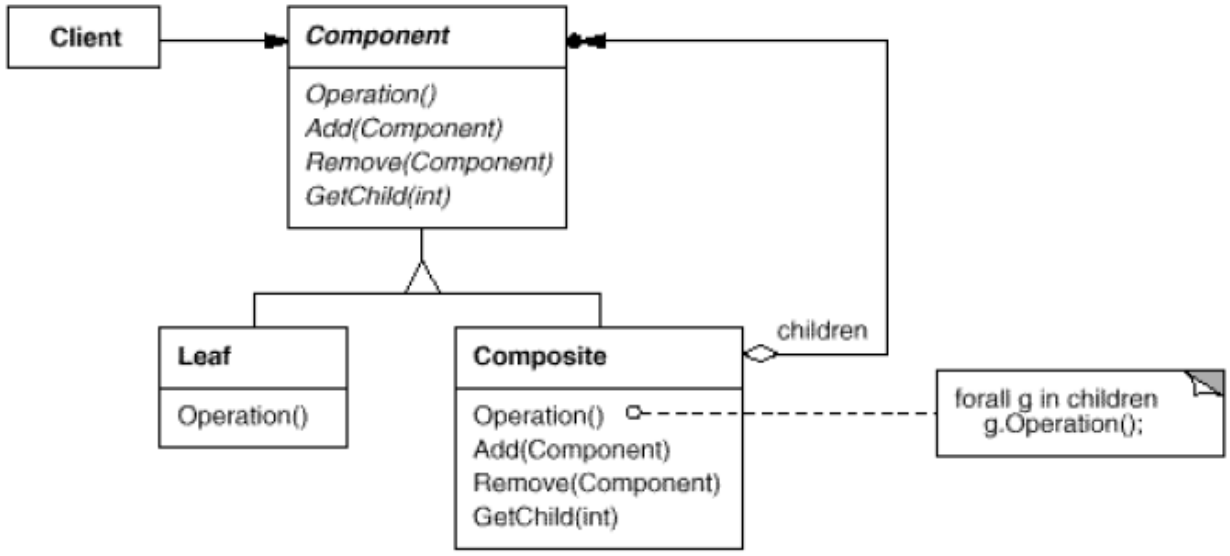
Intent:
To compose objects into tree structures to represent part-whole hierarchies. Composite pattern lets client treat individual objects and compositions of objects uniformly.

Motivation:
There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. Consider for example a program that manipulates a file system. A file system is a tree structure that contains Branches which are Folders as well as Leaf nodes which are Files. Note that a folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object. Note also that since files and folders have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size, it would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface.

Applicability:
Use composite pattern when:

- You-want to represent part-whole hierarchies of objects.
- You want clients to be able to ignore the difference between compositions of objects and individual objects.

Structure:
The structure of composite pattern is as shown below:



Participants:
Following are the participants in composite pattern:

- **Component:** Declares the interface for objects in the composition. Declares an interface for accessing and managing its child components.

- **Leaf:** Represents leaf objects in the composition. A leaf has no children. Defines behavior for primitive objects in the composition.
- **Composite:** Defines behavior for components having children. Stores child components. Implements child-related operations in the *composite* interface.
- **Client:** Manipulates objects in the composition through the *Component* interface.

Collaborations:
Clients use the *Component* class interface to interact with objects in the composite structure.

Consequences:
The composite pattern:

- Defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.

- Makes the client simple. Clients can treat composite structures and individual objects uniformly.

- Makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.

- Can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components.

Implementation:
Following are the issues to consider when implementing composite pattern:

1. *Explicit parent references:* Maintaining references from child components to their parent can simplify the traversal and management of a composite structure. The parent reference simplifies moving up the structure and deleting a component. Parent references also help support the Chain of Responsibility pattern.

2. *Sharing components:* It's often useful to share components, for example, to reduce storage requirements. But when a component can have no more than one parent, sharing components becomes difficult.

3. *Maximizing the Component interface:* One of the goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using. To attain this goal, the Component class should define as many common operations for Composite and Leaf classes as possible. The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them.

4. *Declaring the child management operations:* Although the Composite class implements the Add and Remove operations for managing children, an important issue in the Composite pattern is which classes declare these operations in the Composite class hierarchy.

5. *Should Component implement a list of Components:* You might be tempted to define the set of children as an instance variable in the Component class where the child access and management operations are declared. But putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children.

6. *Child ordering:* Many designs specify an ordering on the children of Composite. In the earlier Graphics example, ordering may reflect front-to-back ordering. If Composites represent parse trees, then compound statements can be instances of a Composite whose children must be ordered to reflect the program.

7. *Caching to improve performance:* If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children. The Composite can cache actual results or just information that lets it short-circuit the traversal or search.

8. *Who should delete components:* In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed. An exception to this rule is when Leaf objects are immutable and thus can be shared.

9. *What's the best data structure for storing components:* Composites may use a variety of data structures to store their children, including linked lists, trees, arrays, and hash tables. The choice of data structure depends (as always) on efficiency.

Sample code:
A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc and when we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

Composite Pattern consists of following objects.

1. Base Component – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an abstract class with some methods common to all the objects.
2. Leaf – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.
3. Composite – It consists of leaf elements and implements the operations in base component.

Here I am applying composite design pattern for the drawing scenario.

Base Component:

Base component defines the common methods for leaf and composites, we can create a class `Shape` with a method `draw(String fillColor)` to draw the shape with given color.

//Shape.java
```
public interface Shape {

        public void draw(String fillColor);
}
```

Leaf Objects:

Leaf implements base component and these are the building block for the composite. We can create multiple leaf objects such as Triangle, Circle etc.

//Triangle.java

```
public class Triangle implements Shape {

    @Override
    public void draw(String fillColor) {
        System.out.println("Drawing Triangle with color "+fillColor);
    }

}
```

//Circle.java

```
public class Circle implements Shape {

    @Override
    public void draw(String fillColor) {
        System.out.println("Drawing Circle with color "+fillColor);
    }

}
```

Composite:

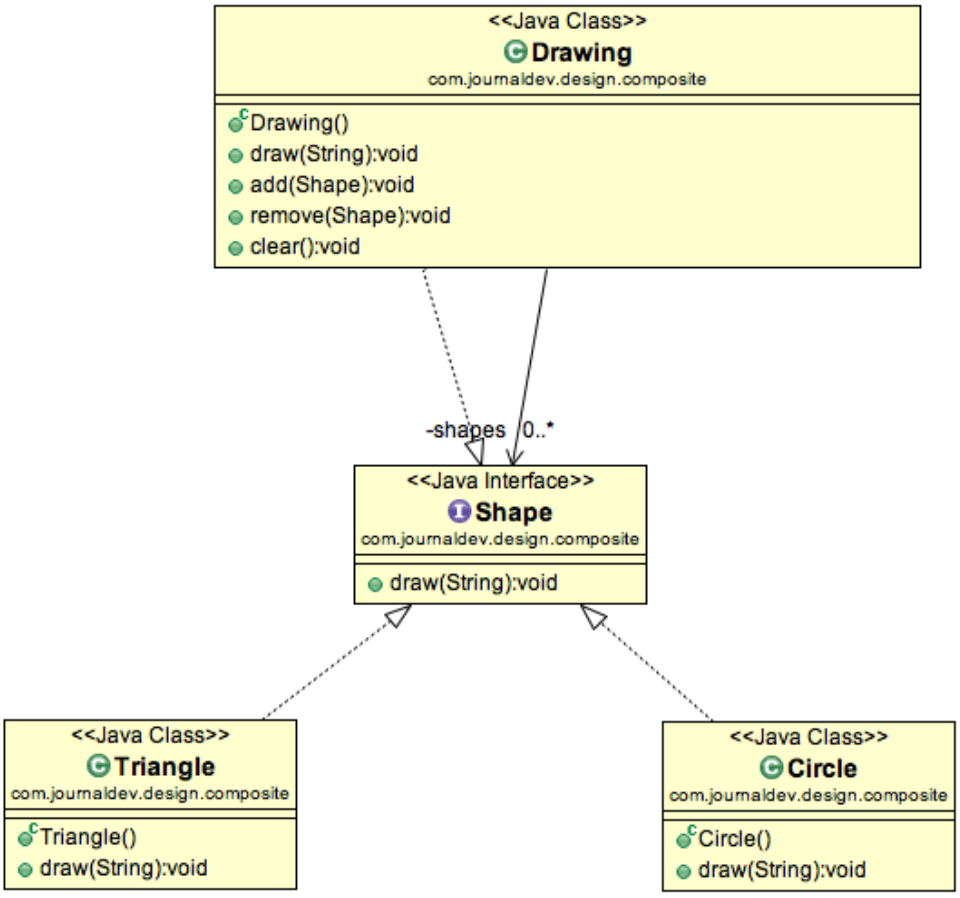
A composite object contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

//Drawing.java

```
import java.util.ArrayList;
import java.util.List;

public class Drawing implements Shape{

    //collection of Shapes
    private List<Shape> shapes = new ArrayList<Shape>();

    @Override
    public void draw(String fillColor) {
        for(Shape sh : shapes)
        {
            sh.draw(fillColor);
        }
    }

    //adding shape to drawing
    public void add(Shape s){
        this.shapes.add(s);
    }

    //removing shape from drawing
```

```
    public void remove(Shape s){
        shapes.remove(s);
    }

    //removing all the shapes
    public void clear(){
        System.out.println("Clearing all the shapes from drawing");
        this.shapes.clear();
    }
}
```

Notice that composite also implements component and behaves similar to leaf except that it can contain group of leaf elements.



Our composite pattern implementation is ready and we can test it with a client program.

//TestCompositePattern.java

```
public class TestCompositePattern {

    public static void main(String[] args) {
        Shape tri = new Triangle();
```

```
        Shape tri1 = new Triangle();
        Shape cir = new Circle();

        Drawing drawing = new Drawing();
        drawing.add(tri1);
        drawing.add(tri1);
        drawing.add(cir);

        drawing.draw("Red");

        drawing.clear();

        drawing.add(tri);
        drawing.add(cir);
        drawing.draw("Green");
    }

}
```

Known uses:
Following are the examples in Java API where composite pattern is used:

- java.awt.Container#add(Component) (practically all over Swing thus)
- javax.faces.component.UIComponent#getChildren() (practically all over JSF UI thus)

Related patterns:
Often the component-parent link is used for a Chain of Responsibility.

Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

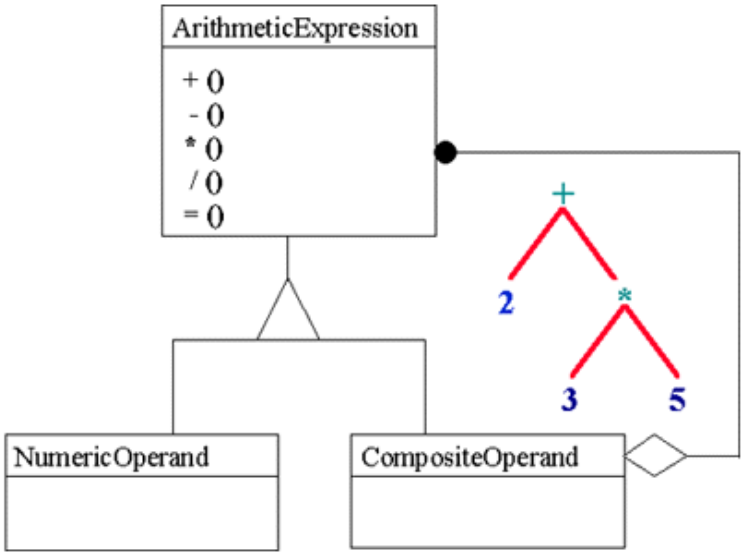
Flyweight lets you share components, but they can no longer refer to their parents.

Iterator can be used to traverse composites.

Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.

Non-software example:

The *Composite* composes objects into tree structures, and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are *Composites.* An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.


ArithmeticExpression
+ 0
- 0
* 0
/ 0
= 0
+
2
*
3
5
NumericOperand
CompositeOperand

Decorator Pattern

Intent:
To attach additional responsibilities to an object dynamically. Decorator provides an alternative to subclassing for extending the functionality.

Also knows as:
Wrapper

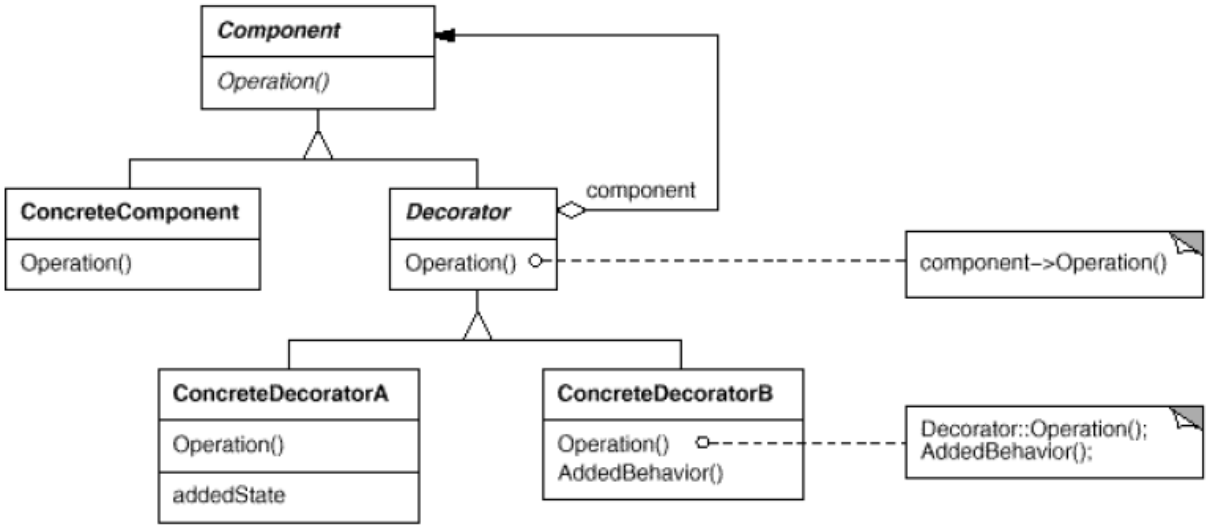
Motivation:
Extending an object's functionality can be done statically (at compile time) by using inheritance however it might be necessary to extend an object's functionality dynamically (at runtime) as an object is used.

Consider the typical example of a graphical window. To extend the functionality of the graphical window for example by adding a frame to the window would require extending the window class to create a FramedWindow class. To create a framed window it is necessary to create an object of the FramedWindow class. However it would be impossible to start with a plain window and extend its functionality at runtime to become a framed window.

Applicability:
Use decorator:

- To add additional responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical.

Structure:
The structure of decorator pattern is as shown below:



Participants:
The participants in the decorator pattern are:

- **Component:** Defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent:** Defines an object to which additional responsibilities can be attached.
- **Decorator:** Maintains a reference to a component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator:** Adds responsibilities to the component.

Collaborations:
Decorator forwards requests to its component object. It may optionally perform additional operations before and after forwarding the request.

Consequences:
The decorator pattern has atleast two key benefits and two liabilities:

1. *More flexibility than static inheritance:* The Decorator pattern providesa more flexible way to add responsibilities to objects than can be had withstatic (multiple) inheritance.

2. *Avoids feature-laden classes high up in the hierarchy:* Decorator offersa pay-as-you-go approach to adding responsibilities. Instead of trying tosupport all foreseeable features in a complex, customizable class, you candefine a simple class and add functionality incrementally with Decoratorobjects.

3. *A decorator and its component aren't identical:* A decorator acts as atransparent enclosure. But from an object identity point of view, adecorated component is not identical to the component itself.

4. *Lots of little objects:* A design that uses Decorator often results in systemscomposed of lots of little objects that all look alike. The objects differonly in the way they are interconnected, not in their class or in the valueof their variables.

Implementation:
Following issues should be considered when applying the decorator pattern:

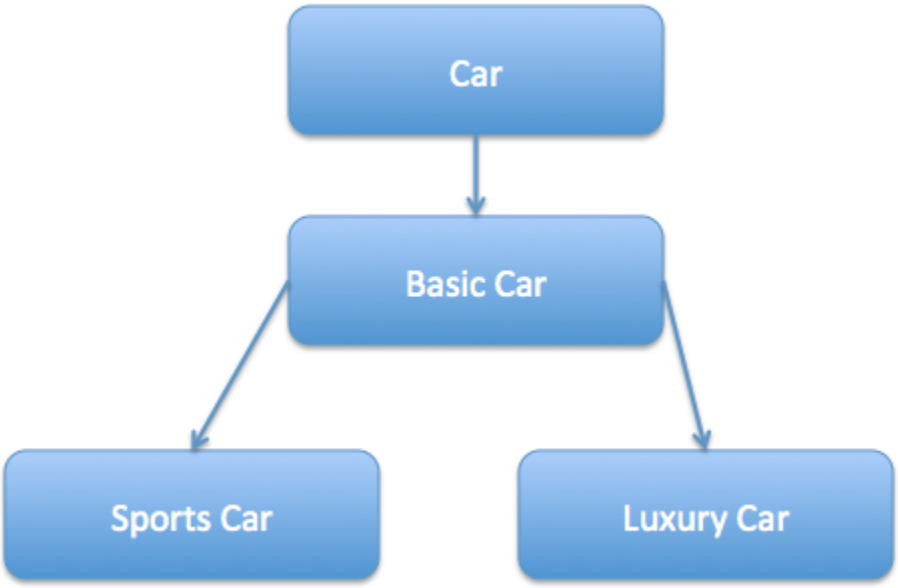
1. *Interface conformance:* A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class.

2. *Omitting the abstract Decorator class:* There's no need to define an abstract Decorator class when you only need to add one responsibility.

3. *Keeping Component classes lightweight:* To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data.

4. *Changing the skin of an object versus changing its guts:* We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy (349) pattern is a good example of a pattern for changing the guts.

Sample code:
Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.



But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now image if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern.

We need to have following types to implement decorator design pattern:

Component Interface – The interface or abstract class defining the methods that will be implemented. In our case Car will be the component interface.

//Car.java
```
public interface Car {

    public void assemble();
}
```

Component Implementation – The basic implementation of the component interface. We can have `BasicCar` class as our component implementation.

//BasicCar.java
```
public class BasicCar implements Car {
```

```
    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }

}
```

Decorator – Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

//CarDecorator.java

```
public class CarDecorator implements Car {

    protected Car car;

    public CarDecorator(Car c){
        this.car=c;
    }

    @Override
    public void assemble() {
        this.car.assemble();
    }

}
```

Concrete Decorators – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as `LuxuryCar` and `SportsCar`.

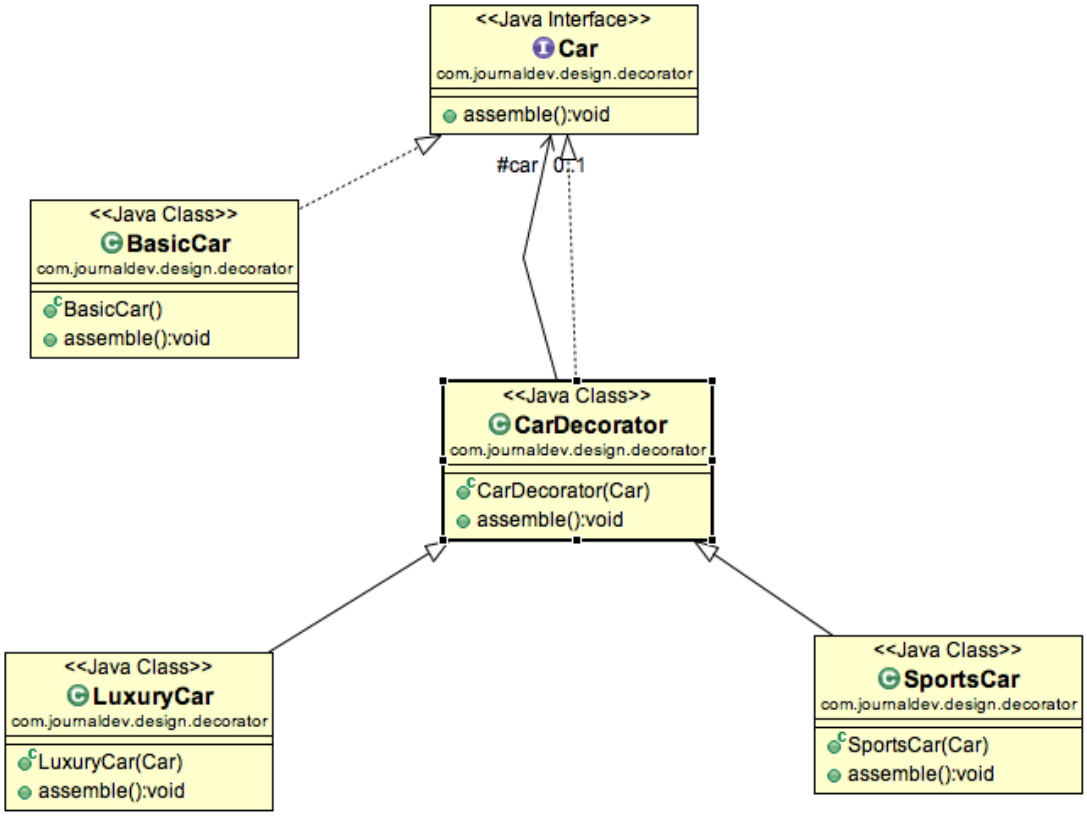
//SportsCar.java

```
public class SportsCar extends CarDecorator {

    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        car.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}
```

//LuxuryCar.java

```
public class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car c) {
        super(c);
    }

    @Override
```

```
    public void assemble(){
        car.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}
```



Here is the client program:

//DecoratorPatternTest.java

```
public class DecoratorPatternTest {

    public static void main(String[] args) {
        Car sportsCar = new SportsCar(new BasicCar());
        sportsCar.assemble();
        System.out.println("\n*****");

        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
        sportsLuxuryCar.assemble();
    }

}
```

Known uses:
Following are the examples in Java API where decorator pattern is used:

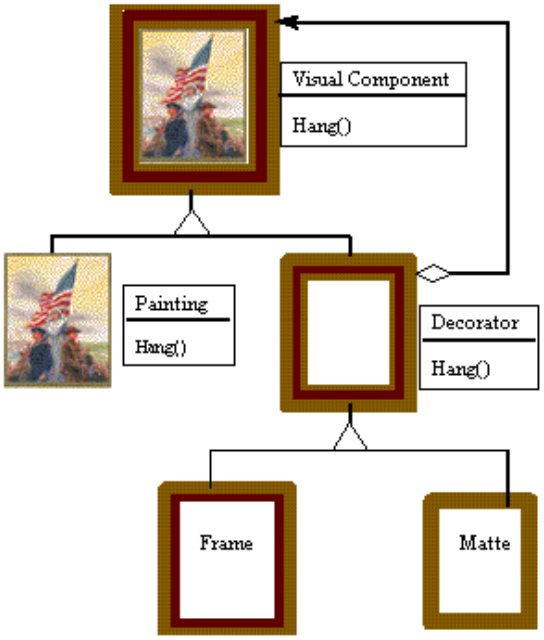
- All subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.
- java.util.Collections, the checkedXXX(), synchronizedXXX() and unmodifiableXXX() methods.
- javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper

Related patterns:
Adapter: A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

Composite: A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.

Strategy: A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

Non-software example:
The *Decorator* attaches additional responsibilities to an object dynamically. Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.



Facade Pattern

Intent:
Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Motivation:
Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

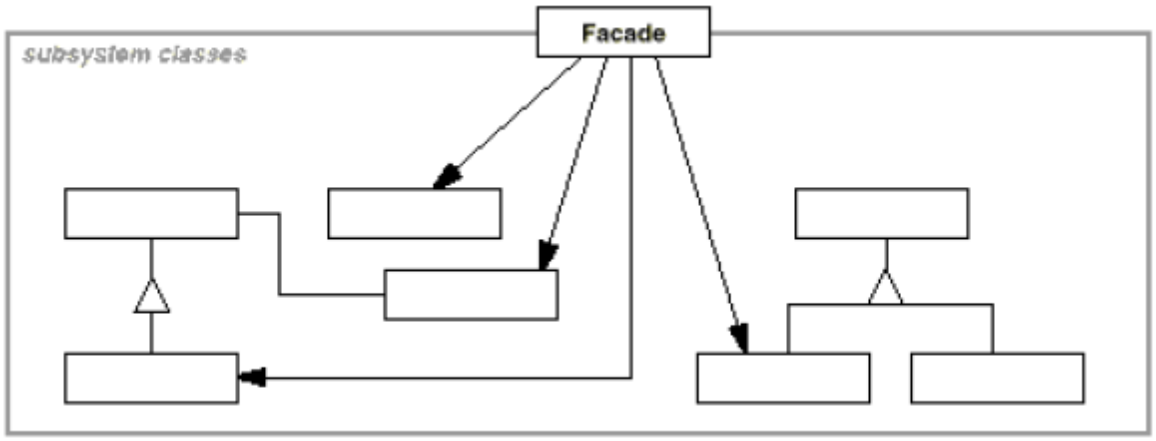
Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler generally don't care about details like parsing and code generation; they merely want to compile some code. For them, the powerful but low-level interfaces in the compiler subsystem only complicate their task.

To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality. The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem.

Applicability:
Use façade pattern :

- To provide a simple interface to a complex system.
- To decouple a subsystem from clients and other subsystems, thereby promoting system independence and portability.
- To define an entry point to each subsystem level. If subsystems are dependent, then the dependencies can be simplified by making them communicate with each other solely through their façade.

Structure:
The structure of façade pattern is shown below:



Participants:
The participants in the façade pattern are:

- **Façade:** Knows which subsystem classes are responsible for a request. Delegates client requests to appropriate subsystem objects.
- **Subsystem classes:** Implement subsystem functionality. Handle work assigned by the *Façade* object. Have no knowledge of the *Façade*.

Collaborations:
Clients communicate with the subsystem by sending requests to the Façade, which forwards them to the appropriate subsystem object.

Consequences:
The façade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.

2. It promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients.

3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation:
Following issues should be considered when implementing façade pattern:

1. *Reducing client-subsystem coupling:* The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.

2. *Public versus private subsystem classes:* A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem.

Sample code:
Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it. So we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.

We can have two helper interfaces, namely `MySqlHelper` and `OracleHelper`.

//MySqlHelper.java
```
import java.sql.Connection;

public class MySqlHelper {

    public static Connection getMySqlDBConnection(){
        //get MySql DB connection using connection parameters
        return null;
    }

    public void generateMySqlPDFReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }

    public void generateMySqlHTMLReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }
}
```

//OracleHelper.java
```
import java.sql.Connection;

public class OracleHelper {

    public static Connection getOracleDBConnection(){
        //get MySql DB connection using connection parameters
        return null;
    }

    public void generateOraclePDFReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }

    public void generateOracleHTMLReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }

}
```

We can create a Facade interface like below. Notice the use of Java Enum for type safety.

//HelperFacade.java
```
import java.sql.Connection;

public class HelperFacade {

    public static void generateReport(DBTypes dbType, ReportTypes reportType,
String tableName){
        Connection con = null;
        switch (dbType){
```

```
        case MYSQL:
            con = MySqlHelper.getMySqlDBConnection();
            MySqlHelper mySqlHelper = new MySqlHelper();
            switch(reportType){
            case HTML:
                mySqlHelper.generateMySqlHTMLReport(tableName, con);
                break;
            case PDF:
                mySqlHelper.generateMySqlPDFReport(tableName, con);
                break;
            }
            break;
        case ORACLE:
            con = OracleHelper.getOracleDBConnection();
            OracleHelper oracleHelper = new OracleHelper();
            switch(reportType){
            case HTML:
                oracleHelper.generateOracleHTMLReport(tableName, con);
                break;
            case PDF:
                oracleHelper.generateOraclePDFReport(tableName, con);
                break;
            }
            break;
        }

    }

    public static enum DBTypes{
        MYSQL,ORACLE;
    }

    public static enum ReportTypes{
        HTML,PDF;
    }
}
```

Here is the client program:

//FacadePatternTest.java

```
import java.sql.Connection;

import com.journaldev.design.facade.HelperFacade;
import com.journaldev.design.facade.MySqlHelper;
import com.journaldev.design.facade.OracleHelper;

public class FacadePatternTest {

    public static void main(String[] args) {
        String tableName="Employee";

        //generating MySql HTML report and Oracle PDF report without using
Facade
        Connection con = MySqlHelper.getMySqlDBConnection();
```

```
        MySqlHelper mySqlHelper = new MySqlHelper();
        mySqlHelper.generateMySqlHTMLReport(tableName, con);

        Connection con1 = OracleHelper.getOracleDBConnection();
        OracleHelper oracleHelper = new OracleHelper();
        oracleHelper.generateOraclePDFReport(tableName, con1);

        //generating MySql HTML report and Oracle PDF report using Facade
        HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL,
HelperFacade.ReportTypes.HTML, tableName);
        HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE,
HelperFacade.ReportTypes.PDF, tableName);
    }

}
```

Known uses:
Following are the examples in Java API where Façade pattern is used:

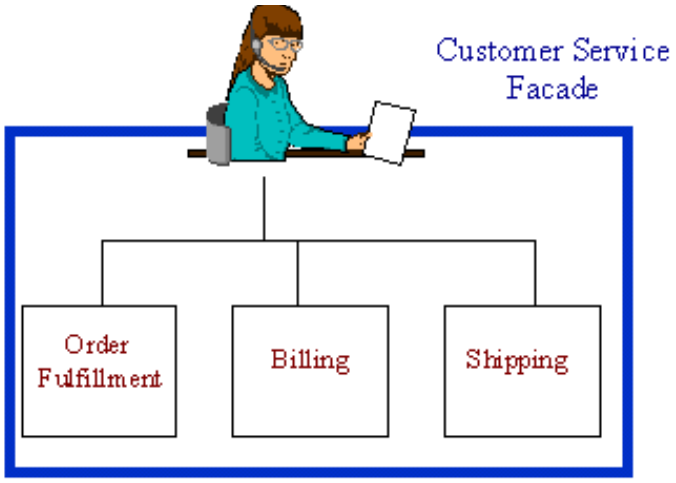
- javax.faces.context.FacesContext, it internally uses among others the abstract/interface types LifeCycle, ViewHandler, NavigationHandler and many more without that the enduser has to worry about it (which are however overrideable by injection).
- javax.faces.context.ExternalContext, which internally uses ServletContext, HttpSession, HttpServletRequest, HttpServletResponse, etc.

Related patterns:
Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.

Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.

Usually only one Facade object is required. Thus Facade objects are often Singletons.

Non-software example:
The *Facade* defines a unified, higher level interface to a subsystem, that makes it easier to use. Consumers encounter a *Facade* when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a *Facade,* providing an interface to the order fulfillment department, the billing department, and the shipping department.


Customer Service
Facade
Order
Fulfillment
Billing
Shipping

Flyweight Pattern

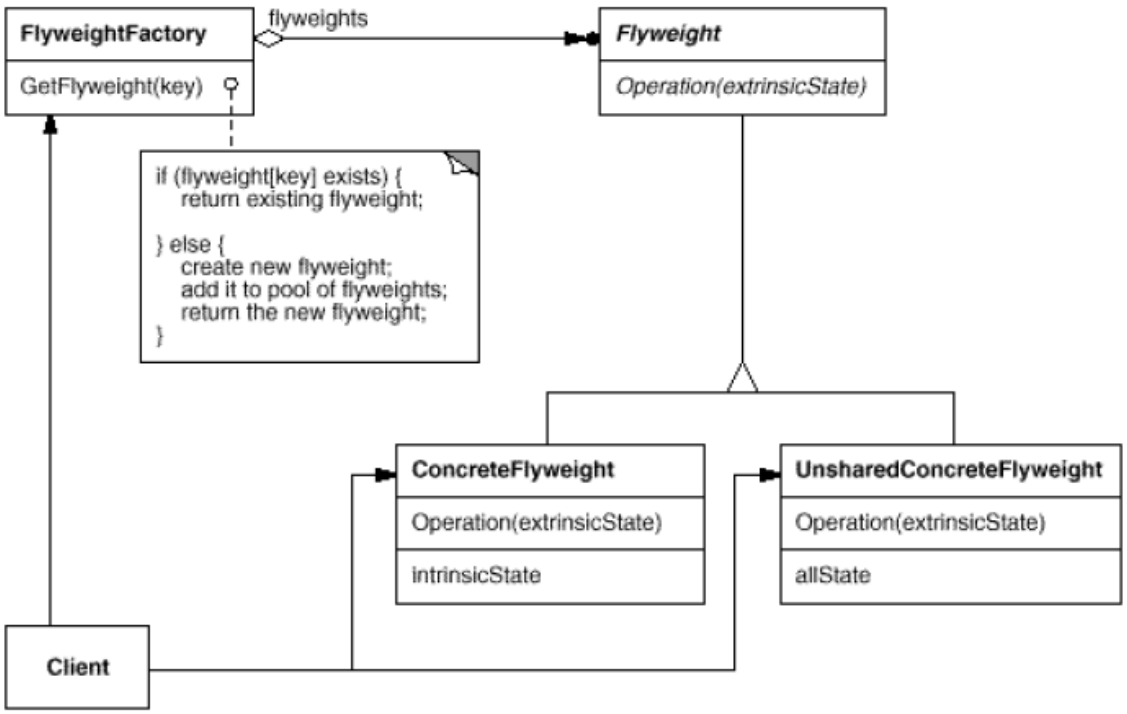
Intent:
To use sharing for supporting large number of fine-grained objects efficiently.

Motivation:
Some programs require a large number of objects that have some shared state among them. Consider for example a game of war, were there is a large number of soldier objects; a soldier object maintain the graphical representation of a soldier, soldier behavior such as motion, and firing weapons, in addition soldier's health and location on the war terrain. Creating a large number of soldier objects is a necessity however it would incur a huge memory cost. Note that although the representation and behavior of a soldier is the same their health and location can vary greatly.

Applicability:
Apply flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend upon object identity.

Structure:
The structure of flyweight pattern is shown below:



Participants:
The participants in the flyweight pattern are:

- **Flyweight:** Declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight:** Implements the *Flyweight*s interface and adds storage for intrinsic state, if any. A *ConcreteFlyweight* object must be sharable.
- **UnsharedConcreteFlyweight:** Not all *Flyweight* subclasses need to be shared. The *Flyweight* interface enables sharing, it doesn't enforce it.
- **FlyweightFactory:** Creates and manages flyweight objects. Ensures that flyweights are shared properly.
- **Client:** Maintains a reference to flyweight(s). Computes or stores the extrinsic state of flyweight(s).

Collaborations:

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic. Intrinsic state is stored in the ConcreteFlyweight object; extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations.

- Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Consequences:
Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared.

Storage savings are a function of several factors:

- The reduction in the total number of instances that comes from sharing
- The amount of intrinsic state per object
- Whether extrinsic state is computed or stored.

Implementation:
Following issues must be considered while implementing flyweight pattern:

1. Removing extrinsic state. The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.

2. Managing shared objects. Because objects are shared, clients shouldn't instantiate them directly. FlyweightFactory lets clients locate a particular flyweight. FlyweightFactory objects often use an associative store to let clients look up flyweights of interest.

Sample code:
For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects. For our example, lets say we need to create a drawing with lines and Ovals. So we will have an interface `Shape` and its concrete implementations as `Line` and `Oval`. Oval class will have intrinsic property to determine whether to fill the Oval with given color or not whereas Line will not have any intrinsic property.

Flyweight Interface and Concrete Classes:

//Shape.java
```
import java.awt.Color;
import java.awt.Graphics;

public interface Shape {

    public void draw(Graphics g, int x, int y, int width, int height,
            Color color);
}
```

//Line.java
```
import java.awt.Color;
import java.awt.Graphics;

public class Line implements Shape {

    public Line(){
        System.out.println("Creating Line object");
        //adding time delay
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void draw(Graphics line, int x1, int y1, int x2, int y2,
            Color color) {
        line.setColor(color);
        line.drawLine(x1, y1, x2, y2);
    }

}
```

//Oval.java
```
import java.awt.Color;
import java.awt.Graphics;

public class Oval implements Shape {

    //intrinsic property
    private boolean fill;
```

```
    public Oval(boolean f){
        this.fill=f;
        System.out.println("Creating Oval object with fill="+f);
        //adding time delay
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void draw(Graphics circle, int x, int y, int width, int height,
            Color color) {
        circle.setColor(color);
        circle.drawOval(x, y, width, height);
        if(fill){
            circle.fillOval(x, y, width, height);
        }
    }

}
```

Notice that delay is intentionally introduced in creating the Object of concrete classes to make the point that flyweight pattern can be used for Objects that takes a lot of time while instantiated.

The flyweight factory will be used by client programs to instantiate the Object, so we need to keep a map of Objects in the factory that should not be accessible by client application. Whenever client program makes a call to get an instance of Object, it should be returned from the HashMap, if not found then create a new Object and put in the Map and then return it. We need to make sure that all the intrinsic properties are considered while creating the Object.

Our flyweight factory class looks like below code.

//ShapeFactory.java

```
import java.util.HashMap;

public class ShapeFactory {

    private static final HashMap<ShapeType,Shape> shapes = new
HashMap<ShapeType,Shape>();

    public static Shape getShape(ShapeType type) {
        Shape shapeImpl = shapes.get(type);

        if (shapeImpl == null) {
            if (type.equals(ShapeType.OVAL_FILL)) {
                shapeImpl = new Oval(true);
            } else if (type.equals(ShapeType.OVAL_NOFILL)) {
                shapeImpl = new Oval(false);
            } else if (type.equals(ShapeType.LINE)) {
                shapeImpl = new Line();
            }
```

```
            shapes.put(type, shapeImpl);
        }
        return shapeImpl;
    }

    public static enum ShapeType{
        OVAL_FILL,OVAL_NOFILL,LINE;
    }
}
```

Below is a sample program that consumes flyweight pattern implementation.

//DrawingClient.java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

import com.journaldev.design.flyweight.ShapeFactory.ShapeType;

public class DrawingClient extends JFrame{

    private static final long serialVersionUID = -1350200437285282550L;
    private final int WIDTH;
    private final int HEIGHT;

    private static final ShapeType shapes[] = { ShapeType.LINE,
ShapeType.OVAL_FILL,ShapeType.OVAL_NOFILL };
    private static final Color colors[] = { Color.RED, Color.GREEN,
Color.YELLOW };

    public DrawingClient(int width, int height){
        this.WIDTH=width;
        this.HEIGHT=height;
        Container contentPane = getContentPane();

        JButton startButton = new JButton("Draw");
        final JPanel panel = new JPanel();

        contentPane.add(panel, BorderLayout.CENTER);
        contentPane.add(startButton, BorderLayout.SOUTH);
        setSize(WIDTH, HEIGHT);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                Graphics g = panel.getGraphics();
```

```
                for (int i = 0; i < 20; ++i) {
                    Shape shape = ShapeFactory.getShape(getRandomShape());
                    shape.draw(g, getRandomX(), getRandomY(),
getRandomWidth(),
                            getRandomHeight(), getRandomColor());
                }
            }
        });
    }

    private ShapeType getRandomShape() {
        return shapes[(int) (Math.random() * shapes.length)];
    }

    private int getRandomX() {
        return (int) (Math.random() * WIDTH);
    }

    private int getRandomY() {
        return (int) (Math.random() * HEIGHT);
    }

    private int getRandomWidth() {
        return (int) (Math.random() * (WIDTH / 10));
    }

    private int getRandomHeight() {
        return (int) (Math.random() * (HEIGHT / 10));
    }

    private Color getRandomColor() {
        return colors[(int) (Math.random() * colors.length)];
    }

    public static void main(String[] args) {
        DrawingClient drawing = new DrawingClient(500,600);
    }
}
```

If you run above client program, you will notice the delay in creating first Line Object and Oval objects with fill as true and false. After that the program executes quickly since its using the shared objects.

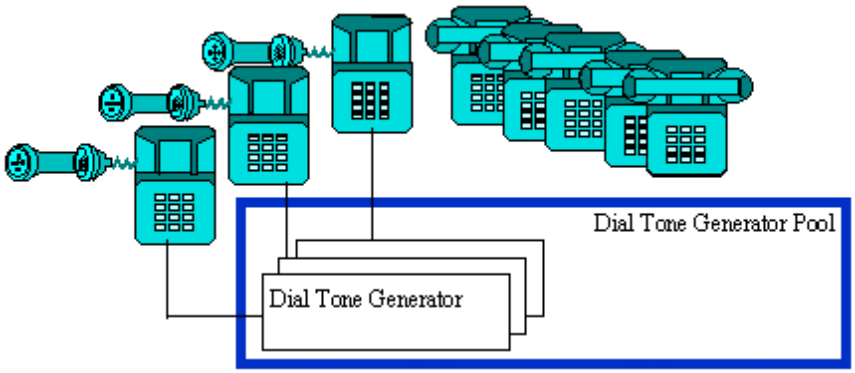
Known uses:
Following are the examples in Java API where flyweight pattern is used:
java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short and Long)

Related patterns:
The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.

It's often best to implement State and Strategy objects as flyweights.

Non-software example:

The *Flyweight* uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a *Flyweight.* There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the hand set to make a call. All that matters to subscribers is that dial tone is provided, digits are received, and the call is completed.



Proxy Pattern

Intent:
To provide a surrogate or placeholder for another object to control access to it.

Also knows as:
Surrogate

Motivation:
Sometimes we need the ability to control the access to an object. For example if we need to use only a few methods of some costly objects we'll initialize those objects when we need them entirely. Until that point we can use some light objects exposing the same interface as the heavy objects. These light objects are called proxies and they will instantiate those heavy objects when they are really need and by then we'll use some light objects instead.

This ability to control the access to an object can be required for a variety of reasons: controlling when a costly object needs to be instantiated and initialized, giving different access rights to an object, as well as providing a sophisticated means of accessing and referencing objects running in other processes, on other machines.

Consider for example an image viewer program. An image viewer program must be able to list and display high resolution photo objects that are in a folder, but how often do someone open a folder and view all the images inside. Sometimes you will be looking for a particular photo, sometimes you will only want to see an image name. The image viewer must be able to list all photo objects, but the photo objects must not be loaded into memory until they are required to be rendered.

Applicability:
Proxy pattern is applicable when:

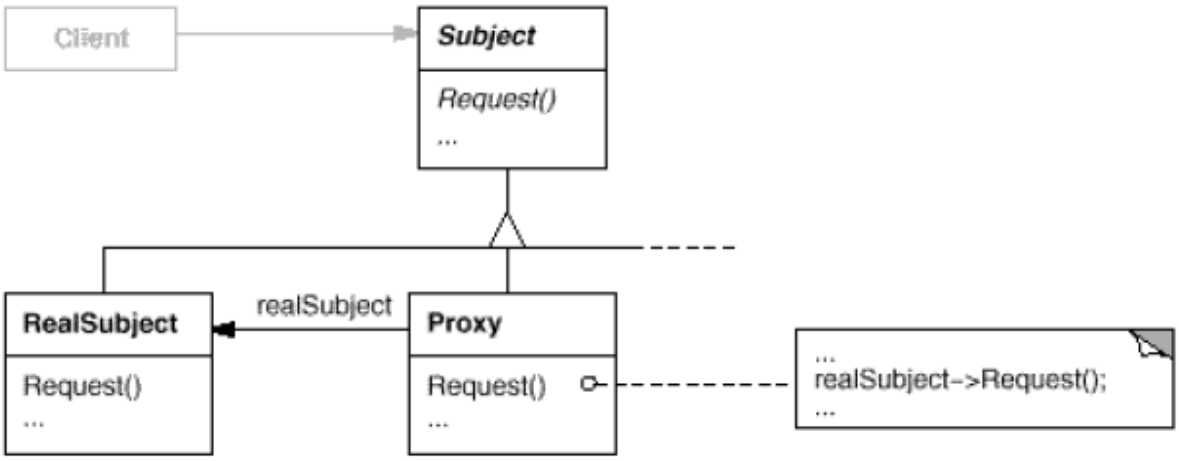
1. A remote proxy provides a local representative for an object in a different address space.

2. A virtual proxy creates expensive objects on demand.

3. A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed.

Structure:
The structure of proxy pattern is as shown below:



Participants:
The participants in proxy pattern are:

- **Proxy:** Maintains a reference that lets the proxy access the real subject. Provides an interface identical to *Subject* so that the *Proxy* can be substituted for the real subject. Controls access to the real subject.
- **Subject:** Defines the common interface for *RealSubject* and *Proxy* so that a *Proxy* can be used anywhere a *RealSubject* is expected.
- **RealSubject:** Defines the real object that the proxy represents.

Collaborations:
Proxy forwards requests to *RealSubject* when appropriate, depending on the kind of proxy.

Consequences:
The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Implementation:
The Proxy pattern can exploit the following language features:

1. *Overloading the member access operator in C++:* C++ supports overloading operator->, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced.

2. *Using doesNotUnderstand in Smalltalk:* Smalltalk provides a hook that you can use to support automatic forwarding of requests. Smalltalk calls doesNotUnderstand: aMessage when a client sends a

message to a receiver that has no corresponding method. The Proxy class can redefine doesNotUnderstand so that the message is forwarded to its subject.

3. *Proxy doesn't always have to know the type of real subject:* If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly. But if Proxies are going to instantiate RealSubjects (such as in a virtual proxy), then they have to know the concrete class.

Sample code:
Let's say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don't want. Here a proxy class can be created to provide controlled access of the program.

Since we code Java in terms of interfaces, here is our interface and its implementation class.

//CommandExecutor.java
```
public interface CommandExecutor {

    public void runCommand(String cmd) throws Exception;
}
```

//CommandExecutorImpl.java
```
import java.io.IOException;

public class CommandExecutorImpl implements CommandExecutor {

    @Override
    public void runCommand(String cmd) throws IOException {
                //some heavy implementation
        Runtime.getRuntime().exec(cmd);
        System.out.println("'" + cmd + "' command executed.");
    }

}
```

Now we want to provide only admin users to have full access of above class, if the user is not admin then only limited commands will be allowed. Here is our very simple proxy class implementation.

//CommandExecutorProxy.java
```
public class CommandExecutorProxy implements CommandExecutor {

    private boolean isAdmin;
    private CommandExecutor executor;

    public CommandExecutorProxy(String user, String pwd){
        if("Pankaj".equals(user) && "J@urnalD$v".equals(pwd)) isAdmin=true;
        executor = new CommandExecutorImpl();
    }

    @Override
```

```
    public void runCommand(String cmd) throws Exception {
        if(isAdmin){
            executor.runCommand(cmd);
        }else{
            if(cmd.trim().startsWith("rm")){
            throw new Exception("rm command is not allowed for non-admin
            users.");
            }else{
                executor.runCommand(cmd);
            }
        }
    }

}
```

Here is the client program:

```
//ProxyPatternTest.java
public class ProxyPatternTest {

    public static void main(String[] args){
        CommandExecutor executor = new CommandExecutorProxy("Pankaj",
"wrong_pwd");
        try {
            executor.runCommand("ls -ltr");
            executor.runCommand(" rm -rf abc.pdf");
        } catch (Exception e) {
            System.out.println("Exception Message::"+e.getMessage());
        }

    }

}
```

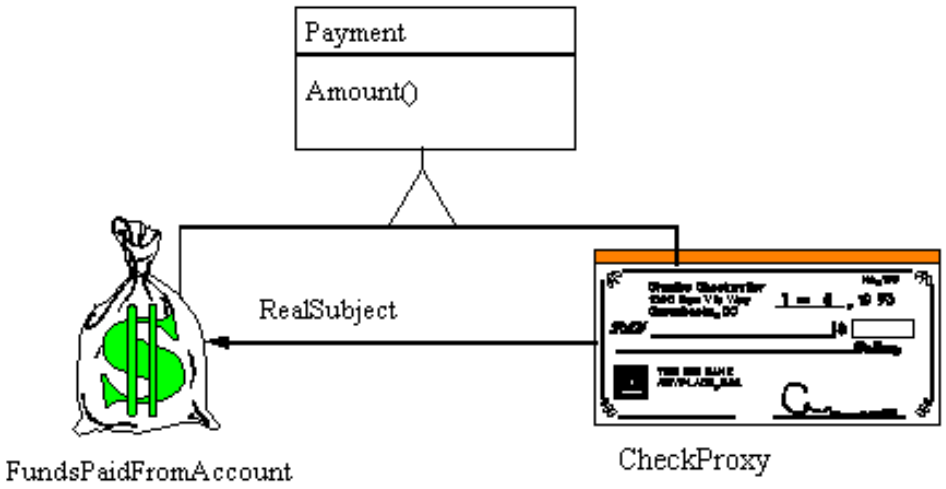
Known uses:
Following are the examples in Java API where proxy pattern is used:

- java.lang.reflect.Proxy
- java.rmi.*, the whole API actually.

Related patterns:
An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.

Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

Non-software example:
The *Proxy* provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.


Payment
Amount()
RealSubject
FundsPaidFromAccount
CheckProxy