Design Patterns

Creational Patterns

Structural Patterns

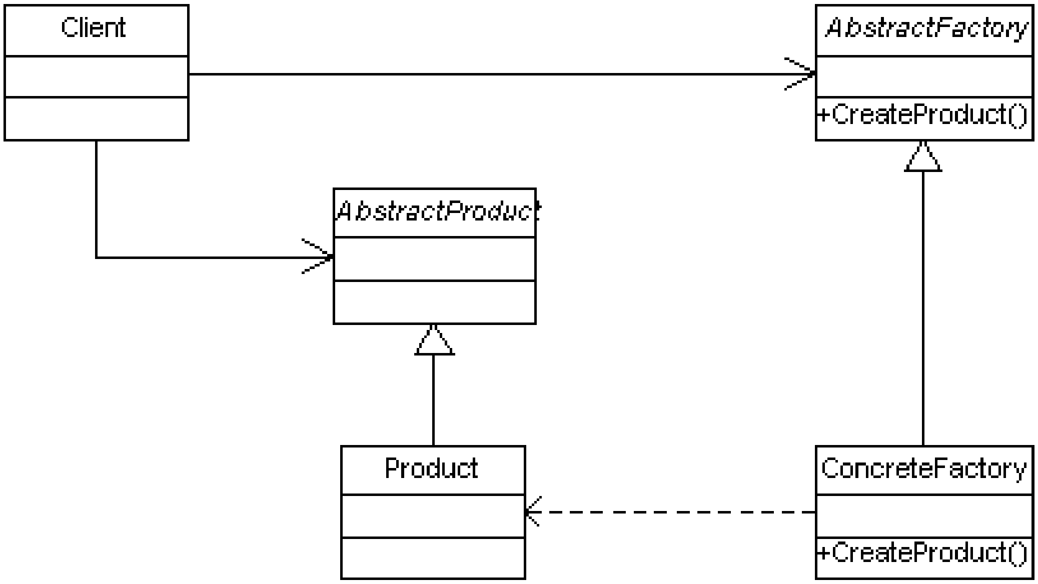
Behavioral Patterns

Creational Patterns

- How to make an object.
- Isolating how objects are created, composed and represented from the rest of the system
-

C1. Factory Method

- Defines an interface for creating an oject but lets subclasses decide which class to instanciate
- *When to use it?*
    - A class can't anticipate the class of objects it must create. A class wants its subclasses to specify the objects it creates.



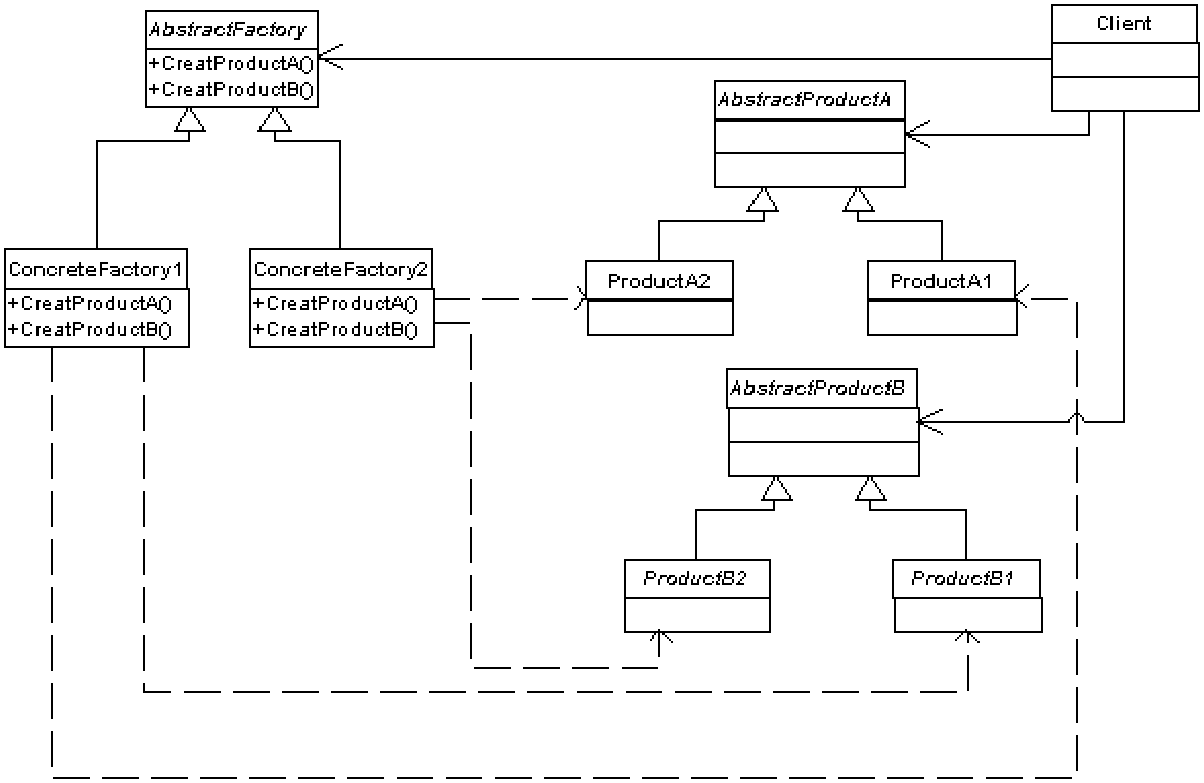
C2. Abstract Factory

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes
- *When to use it:*

  - The system should be independent of how its products are created, composed and represented

- A family of related product objects is designed to be used together and we need to enforce this constraint

- We need to provide a class library of products and we want to reveal just their interfaces not their implementations



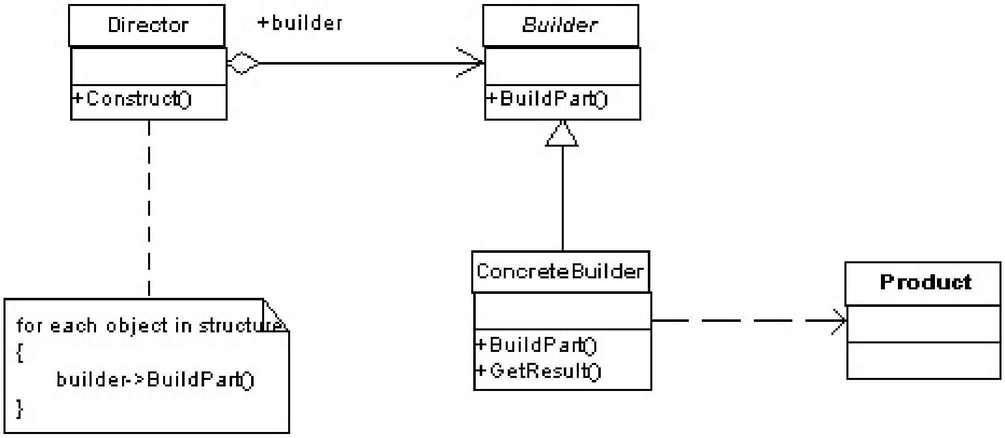
```
// Factories
        public abstract class FinancialToolsFactory {
                public abstract TaxProcessor createTaxProcessor();
                public abstract ShipFeeProcessor createShipFeeProcessor();
        }
        public class CanadaFinancialToolsFactory extends FinancialToolsFactory {
                public TaxProcessor createTaxProcessor() {
                        return new CanadaTaxProcessor();
                }
                public ShipFeeProcessor createShipFeeProcessor() {
                        return new CanadaShipFeeProcessor();
                        }
        }
        public class EuropeFinancialToolsFactory extends FinancialToolsFactory {
                public TaxProcessor createTaxProcessor() {
                        return new EuropeTaxProcessor();
                }
                public ShipFeeProcessor createShipFeeProcessor() {
                        return new EuropeShipFeeProcessor();
                }
```

```
}


// Products
public abstract class ShipFeeProcessor {
        abstract void calculateShipFee(Order order);
}
public abstract class TaxProcessor {
        abstract void calculateTaxes(Order order);
}
public class EuropeShipFeeProcessor extends ShipFeeProcessor {
        public void calculateShipFee(Order order) {
        // insert here Europe specific ship fee calculation
        }
}
public class CanadaShipFeeProcessor extends ShipFeeProcessor {
        public void calculateShipFee(Order order) {
        // insert here Canada specific ship fee calculation
        }
}
public class EuropeTaxProcessor extends TaxProcessor {
        public void calculateTaxes(Order order) {
                // insert here Europe specific taxt calculation
        }
}
public class CanadaTaxProcessor extends TaxProcessor {
        public void calculateTaxes(Order order) {
                // insert here Canada specific taxt calculation
        }
}

// Client
public class OrderProcessor {
        private TaxProcessor taxProcessor;
        private ShipFeeProcessor shipFeeProcessor;

        public OrderProcessor(FinancialToolsFactory factory) {
                taxProcessor = factory.createTaxProcessor();
                shipFeeProcessor = factory.createShipFeeProcessor();
        }
        public void processOrder (Order order)   {
                // ....
                taxProcessor.calculateTaxes(order);
                shipFeeProcessor.calculateShipFee(order);
                // ....
        }
}

// Integration with the overall application
public class Application {
        public static void main(String[] args) {
                // .....
                String countryCode = "EU";
                Customer customer = new Customer();
                Order order = new Order();
                OrderProcessor orderProcessor = null;
```

```
            FinancialToolsFactory factory = null;

            if (countryCode == "EU") {
                factory = new EuropeFinancialToolsFactory();
            } else if (countryCode == "CA") {
                factory = new CanadaFinancialToolsFactory();
            }
            orderProcessor = new OrderProcessor(factory);
            orderProcessor.processOrder(order);
        }
    }
```

C3. Builder

- Separates object construction from its representation, always creates the same type of object.
- *When to use it:*
    - The algorithm for creating a complex object should be independent of the parts that make up the object and who they are assembled
    - The construction process must allow different representations for the constructed object



```
// Builders
    public abstract class PromoKitBuilder {
        protected PromoKit promoKit = new PromoKit();
        public abstract void buildVideoPart();
        public abstract void buildGarmentPart();
        public abstract void buildBookPart();
        public abstract PromoKit getPromoKit();
    }

    public class MenPromoKitBuilder extends PromoKitBuilder {
        public void buildVideoPart() {
            // add videos to PromoKit based on men-specific preferences
        }
    public void buildGarmentPart() {
        // add men garments to PromoKit
    }
    public void buildBookPart() {
        // add books to PromoKit based on men-specific preferences
    }
    public PromoKit getPromoKit() {
        return promoKit;
    }
}

public class WomenPromoKitBuilder extends PromoKitBuilder {
    public void buildVideoPart() {
        // add videos to PromoKit based on women-specific preferences
    }
    public void buildGarmentPart() {
        // add women garments to PromoKit
    }
    public void buildBookPart() {
        // add books to PromoKit based on women-specific preferences
    }
    public PromoKit getPromoKit() {
        return promoKit;
    }
}

// Director
public class PromoKitDirector {
    public PromoKit createPromoKit(PromoKitBuilder builder) {
        builder.buildVideoPart();
        builder.buildGarmentPart();
        builder.buildBookPart();
        return builder.getPromoKit();
    }
}

// Integration with overal application
public class Application {
    public static void main(String[] args) {
        String gendre = "M";
        PromoKitDirector director = new PromoKitDirector();
        PromoKitBuilder promoKitBuilder = null;

        if (gendre.equals("M")) {
            promoKitBuilder = new MenPromoKitBuilder();
        } else if (gendre.equals("F")) {
            promoKitBuilder = new WomenPromoKitBuilder();
        } else {
            // ....
        }
        PromoKit result = director.createPromoKit(promoKitBuilder);
    }
}
```

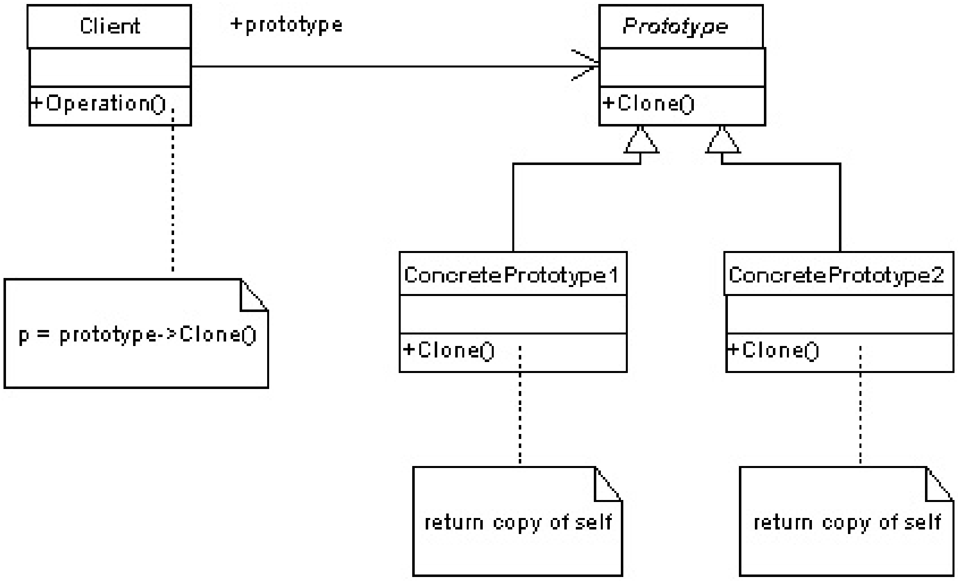
C4. Prototype

- A fully initialized instance used for copying or cloning.
- *When to use it?*

  - the classes to instanciate are specified at runtime

  - we have to avoid building a class hierarchy of factories that parallels the class hierarchy of products.

  - instances of a class have only a few different combinations of state



```
public abstract class Product implements Cloneable {
                    private String SKU;
                    private String description;

                    public Object clone() {
                              Object clone = null;
                              try {
                                        clone = super.clone();
                              } catch (CloneNotSupportedException e) {
                                        e.printStackTrace();
                              }
                              return clone;
                    }
                    public String getDescription() {
                              return description;
                    }
                    public String getSKU() {
                              return SKU;
                    }
```

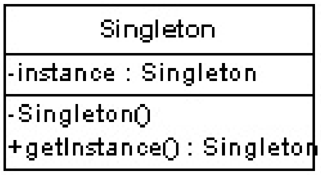
```
        public void setDescription(String string) {
                description = string;
        }
        public void setSKU(String string) {
                SKU = string;
        }
}

public class Book extends Product {
        private int numberOfPages;

        public int getNumberOfPages() {
                return numberOfPages;
        }
        public void setNumberOfPages(int i) {
                numberOfPages = i;
        }
}

public class DVD extends Product {
        private int duration;

        public int getDuration() {
                return duration;
        }
        public void setDuration(int i) {
                duration = i;
        }
}

import java.util.*;
public class ProductCache {
        private static Hashtable productMap = new Hashtable();

        public static Product getProduct(String productCode) {
                Product cachedProduct = (Product) productMap.get(productCode);
                return (Product) cachedProduct.clone();
        }

        public static void loadCache() {
                // for each product run expensive query and instantiate product
                // productMap.put(productKey, product);
                // for exemplification, we add only two products
                Book b1 = new Book();
                b1.setDescription("Oliver Twist");
                b1.setSKU("B1");
                b1.setNumberOfPages(100);
                productMap.put(b1.getSKU(), b1);
                DVD d1 = new DVD();
                d1.setDescription("Superman");
                d1.setSKU("D1");
                d1.setDuration(180);
                productMap.put(d1.getSKU(), d1);
        }
}
```

```
public class Application {
        public static void main(String[] args) {
                ProductCache.loadCache();

                Book clonedBook = (Book) ProductCache.getProduct("B1");
                System.out.println("SKU = " + clonedBook.getSKU());
                System.out.println("SKU = " + clonedBook.getDescription());
                System.out.println("SKU = " + clonedBook.getNumberOfPages());

                DVD clonedDVD = (DVD) ProductCache.getProduct("D1");
                System.out.println("SKU = " + clonedDVD.getSKU());
                System.out.println("SKU = " + clonedDVD.getDescription());
                System.out.println("SKU = " + clonedDVD.getDuration());
        }
}
```

C5. Singleton

- A class with only a single instance with global access points.
- *When to use it?*

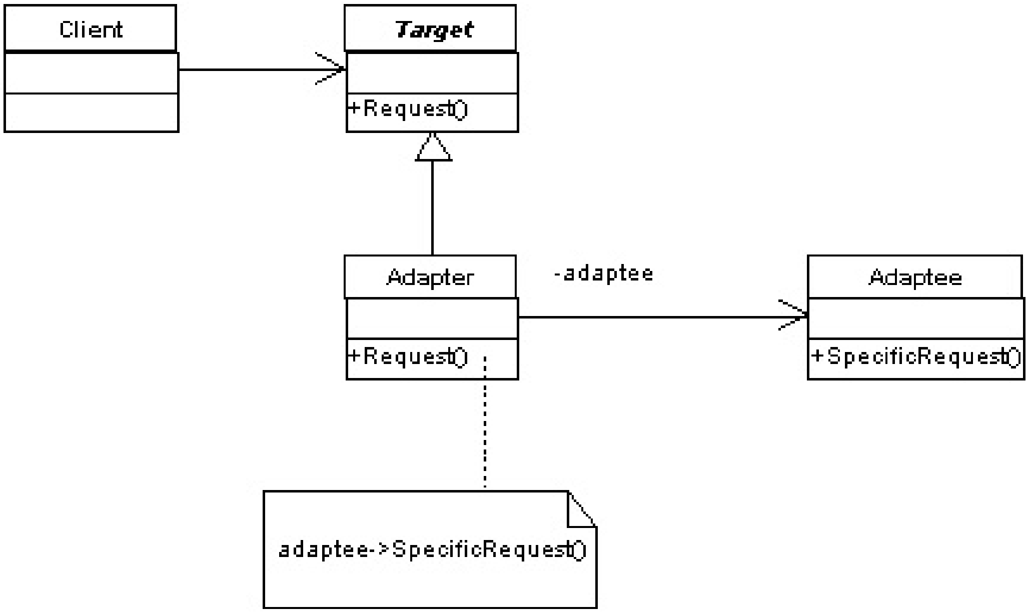
  - It must be exactly one instance of a class and it must be accessible to clients from a well-known access point.



Structural Patterns - The building blocks of objects

S1. Adapter

- Defines an intermediary between two classes, converting the interface of one class so it can be used with the other. This enables classes with incompatible interface to work together.
- *When to use it?*
  - When we need to allow one or more incompatible objects to communicate and interact
  - When we need to improve reusability of older functionality

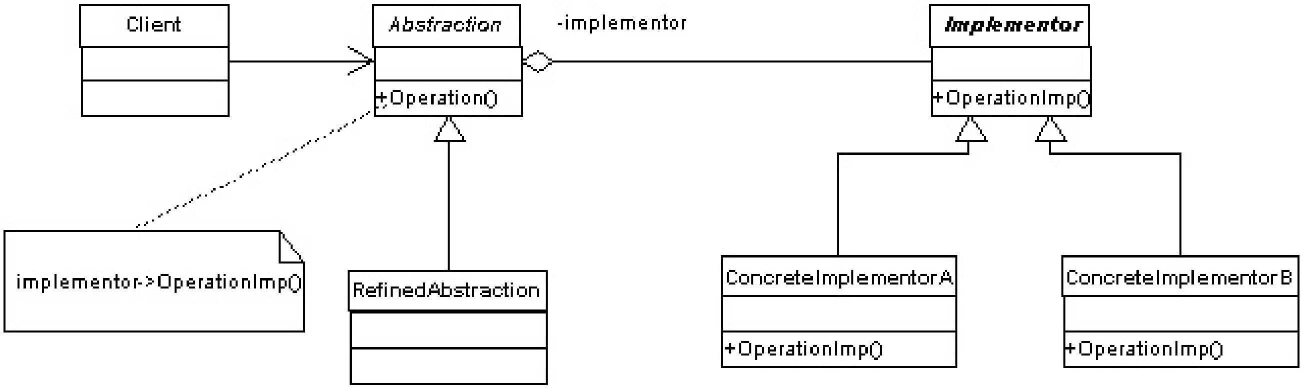


```
class LegacyLine{
    public void draw(int x1, int y1, int x2, int y2){
        System.out.println("line from (" + x1 + ',' + y1 + ") to (" + x2 + ',' + y2 + ')');
    }
}

class LegacyRectangle{
    public void draw(int x, int y, int w, int h) {
        System.out.println("rectangle at (" + x + ',' + y + ") with width " + w
          + " and height " + h);
    }
}

interface Shape{
  void draw(int x1, int y1, int x2, int y2);
}

class Line implements Shape{
    private LegacyLine adaptee = new LegacyLine();
    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw(x1, y1, x2, y2);
    }
}

class Rectangle implements Shape{
    private LegacyRectangle adaptee = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2){
        adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 - x1),
          Math.abs(y2 - y1));
    }
}
```

```
public class AdapterDemo{
    public static void main(String[] args) {
        Shape[] shapes =
        {
            new Line(), new Rectangle()
        };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
          shapes[i].draw(x1, y1, x2, y2);
    }
}
```

S2. Bridge

- Separates an object's interface from its implementation so the two can vary independently
- *When to use it?*
  - When we need to avoid a permanent binding between an abstraction and its implementation
  - When we want to have both abstractions and their implementations extensible using subclasses

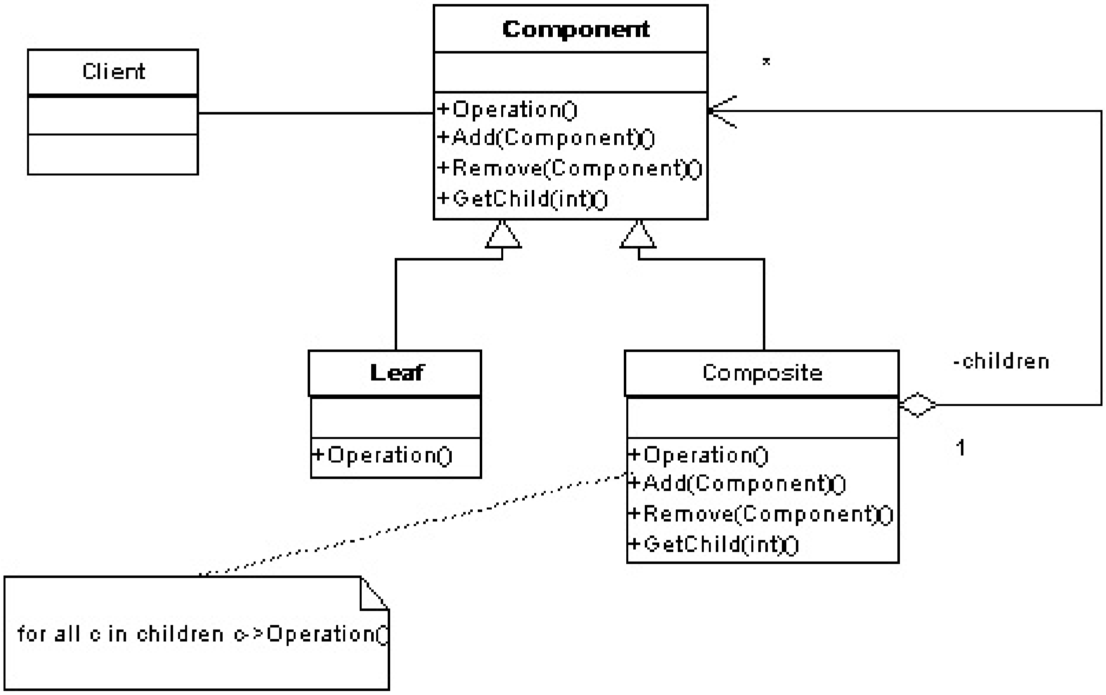


```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor"  1/2 */
class DrawingAPI1 implements DrawingAPI {
   public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
   }
}
```

```
/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
  public void drawCircle(double x, double y, double radius) {
     System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
  }
}

/** "Abstraction" */
abstract class Shape {
  protected DrawingAPI drawingAPI;

  protected Shape(DrawingAPI drawingAPI){
    this.drawingAPI = drawingAPI;
  }

  public abstract void draw();                            // low-level
  public abstract void resizeByPercentage(double pct);    // high-level
}

/** "Refined Abstraction" */
class CircleShape extends Shape {
  private double x, y, radius;
  public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
    super(drawingAPI);
    this.x = x;  this.y = y;  this.radius = radius;
  }

  // low-level i.e. Implementation specific
  public void draw() {
     drawingAPI.drawCircle(x, y, radius);
  }
  // high-level i.e. Abstraction specific
  public void resizeByPercentage(double pct) {
     radius *= pct;
  }
}

/** "Client" */
class BridgePattern {
  public static void main(String[] args) {
     Shape[] shapes = new Shape[] {
        new CircleShape(1, 2, 3, new DrawingAPI1()),
        new CircleShape(5, 7, 11, new DrawingAPI2()),
     };

     for (Shape shape : shapes) {
        shape.resizeByPercentage(2.5);
        shape.draw();
     }
  }
}
```

S3. Composite

- A structure of simple and composite objects which makes the total object more than just the sum of its parts.
- Allows clients to treat individual objects and compositions of objects uniformly
- *When to use it?*
    - When we want to represent part-while hierarchies of objects
    - When we want clients to be able to ignore the difference between compositions of objects and individual objects

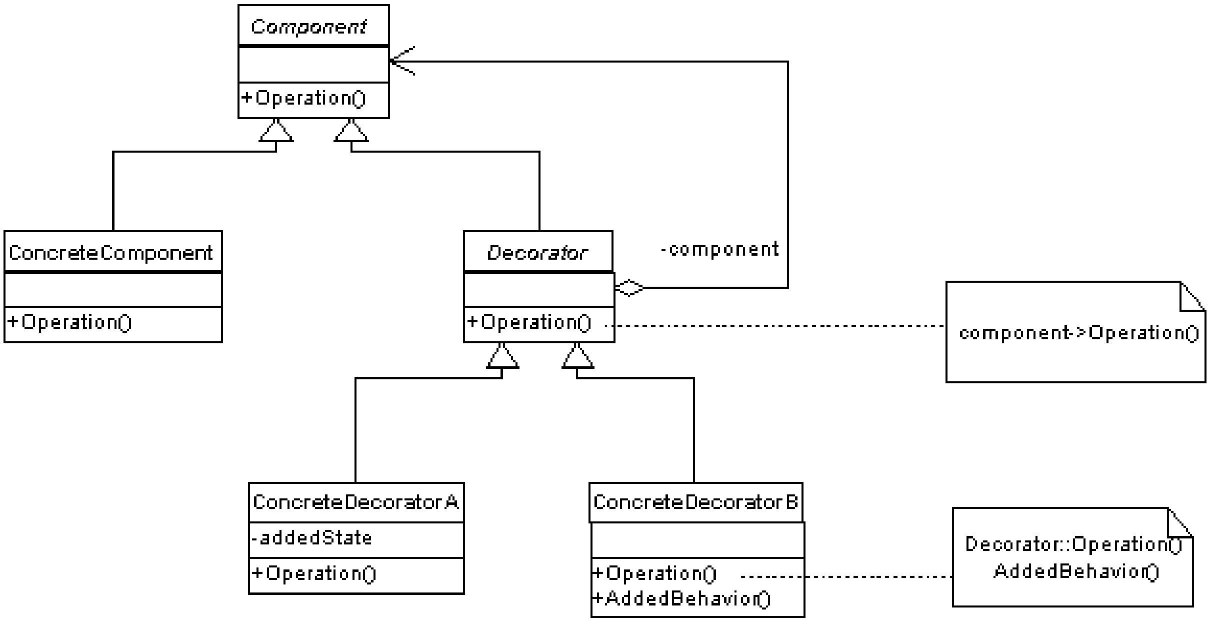


```
interface Component { void traverse(); }      // 1. "lowest common denominator"
class Primitive implements Component {        // 2. "Isa" relationship
   private int value;
   public Primitive( int val ) { value = val; }
   public void traverse()      { System.out.print( value + "  " ); }
}

abstract class Composite implements Component {      // 2. "Isa" relationship
   private Component[] children = new Component[9];  // 3. Couple to interface
   private int         total    = 0;
   private int         value;
   public Composite( int val )    { value = val; }
   public void add( Component c ) { children[total++] = c; } // 3. Couple to
   public void traverse() {                                  //    interface
      System.out.print( value + "  " );
```

```
    for (int i=0; i < total; i++)
       children[i].traverse();          // 4. Delegation and polymorphism
} }

class Row extends Composite {               // Two different kinds of "con-
  public Row( int val ) { super( val ); }   // tainer" classes.  Most of the
  public void traverse() {                  // "meat" is in the Composite
    System.out.print( "Row" );              // base class.
    super.traverse();
} }

class Column extends Composite {
  public Column( int val ) { super( val ); }
  public void traverse() {
    System.out.print( "Col" );
    super.traverse();
} }

public class CompositeDemo {
  public static void main( String[] args ) {
    Composite first  = new Row( 1 );          // Row1
    Composite second = new Column( 2 );       //   |
    Composite third  = new Column( 3 );       //   +-- Col2
    Composite fourth = new Row( 4 );          //   |     |
    Composite fifth  = new Row( 5 );          //   |     +-- 7
    first.add( second );                      //   +-- Col3
    first.add( third  );                      //   |     |
    third.add( fourth );                      //   |     +-- Row4
    third.add( fifth  );                      //   |     |     |
    first.add(  new Primitive( 6 ) );         //   |     |     +-- 9
    second.add( new Primitive( 7 ) );         //   |     +-- Row5
    third.add(  new Primitive( 8 ) );         //   |     |     |
    fourth.add( new Primitive( 9 ) );         //   |     |     +-- 10
    fifth.add(  new Primitive(10 ) );         //   |     +-- 8
    first.traverse();                         //   +-- 6
} }
```

S4. Decorator

- Dynamically add responsibilities to an object
- Provides a flexible alternatives to subclassing for extending functionality
- *When to use it?*
    - When we want to add responsibilities to individual objects dynamically and transparently – without affecting other objects
    - When extension by subclassing is impractical



```
// The Coffee Interface defines the functionality of Coffee implemented by decorator
public interface Coffee {
   public double getCost(); // returns the cost of the coffee
   public String getIngredients(); // returns the ingredients of the coffee
}

// implementation of a simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee {
   public double getCost() {
      return 1;
   }

   public String getIngredients() {
      return "Coffee";
   }
}
```

The following classes contain the decorators for all Coffee classes, including the decorator classes themselves..

```
// abstract decorator class - note that it implements Coffee interface
abstract public class CoffeeDecorator implements Coffee {
   protected final Coffee decoratedCoffee;
   protected String ingredientSeparator = ", ";

   public CoffeeDecorator(Coffee decoratedCoffee) {
      this.decoratedCoffee = decoratedCoffee;
   }

   public double getCost() { // implementing methods of the interface
      return decoratedCoffee.getCost();
   }

   public String getIngredients() {
      return decoratedCoffee.getIngredients();
```

```
    }
}

// Decorator Milk that mixes milk with coffee
// note it extends CoffeeDecorator
public class Milk extends CoffeeDecorator {
    public Milk(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() { // overriding methods defined in the abstract superclass
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Milk";
    }
}

// Decorator Whip that mixes whip with coffee
// note it extends CoffeeDecorator
public class Whip extends CoffeeDecorator {
    public Whip(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() {
        return super.getCost() + 0.7;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Whip";
    }
}

// Decorator Sprinkles that mixes sprinkles with coffee
// note it extends CoffeeDecorator
public class Sprinkles extends CoffeeDecorator {
    public Sprinkles(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() {
        return super.getCost() + 0.2;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Sprinkles";
    }
}
```

Here's a test program that creates a Coffee instance which is fully decorated (i.e., with milk, whip, sprinkles), and calculate cost of coffee and prints its ingredients:

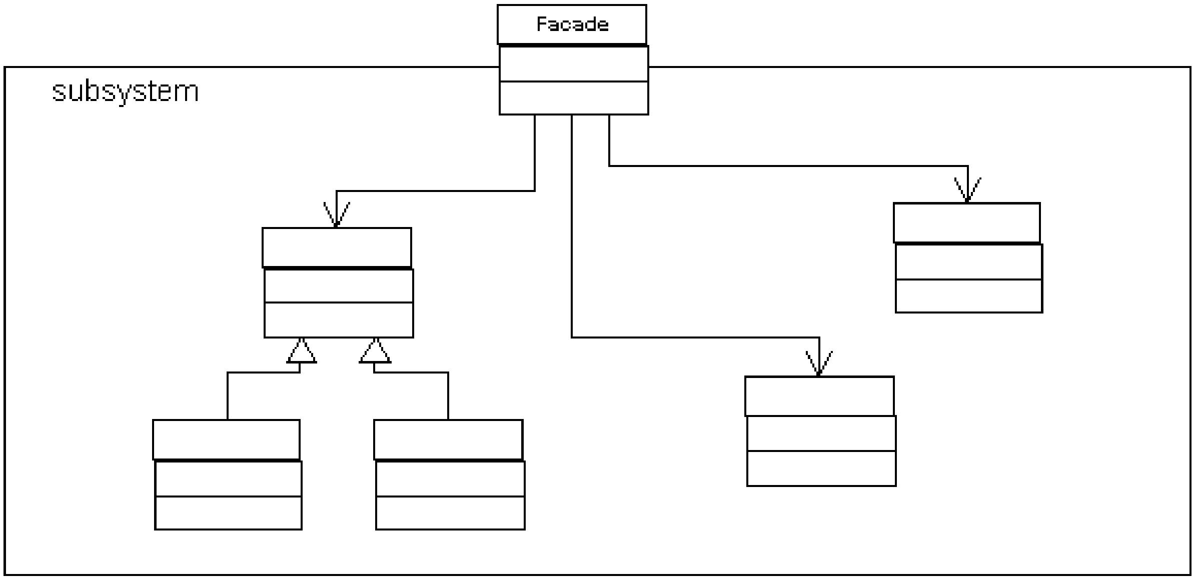
```
public class Main
{
    public static void main(String[] args)
    {
        Coffee c = new SimpleCoffee();
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

        c = new Milk(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

        c = new Sprinkles(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

        c = new Whip(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

        // Note that you can also stack more than one decorator of the same type
        c = new Sprinkles(c);
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
    }
}
```

The output of this program is given below:

```
Cost: 1.0; Ingredients: Coffee
 Cost: 1.5; Ingredients: Coffee, Milk
 Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
 Cost: 2.4; Ingredients: Coffee, Milk, Sprinkles, Whip
 Cost: 2.6; Ingredients: Coffee, Milk, Sprinkles, Whip, Sprinkles
```

S5. Façade

- A single class that hides the complexity an entire subsystem.
- Defines a higher level interface that makes the subsystem easier to use, because we only have to deal with one interface to communicate with the subsystem
- *When to use it?*
    - We want to provide a simple interface to a complex subsystem
    - There are many dependencies between clients and the implementation classes of abstraction

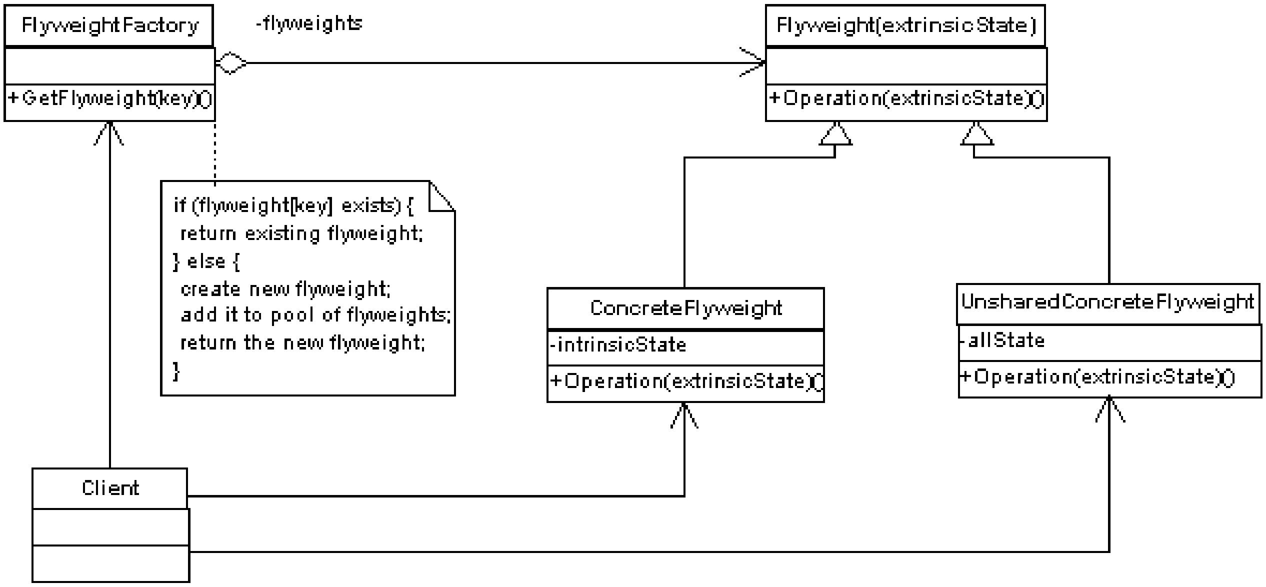


This is an abstract example of how a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive).

```
/* Complex parts */

class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

/* Facade */

class Computer {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public Computer() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void startComputer() {
        cpu.freeze();
        memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
        cpu.jump(BOOT_ADDRESS);
        cpu.execute();
    }
}

/* Client */

class You {
    public static void main(String[] args) {
        Computer facade = new Computer();
        facade.startComputer();
    }
}
```

S6. Flyweight

- A fine-grained instance used for efficient sharing of information contained elsewhere.
- *When to use it?*
    - When an application uses a large number of objects.
    - When most of the objects state can be made extrinsic
    - When many groups of objects may be replaced by relatively few shared objects, once extrinsic state is removed.
    - When storage cost is high because of the high number of objects

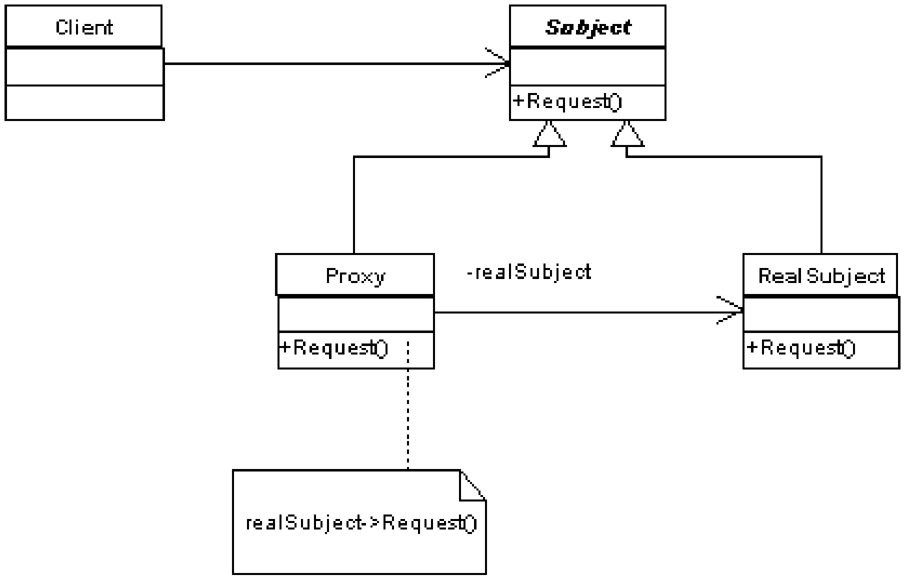

FlyweightFactory
+GetFlyweight(key)()
-flyweights
Flyweight(extrinsicState)
+Operation(extrinsicState)()
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
-intrinsicState
+Operation(extrinsicState)()
UnsharedConcreteFlyweight
-allState
+Operation(extrinsicState)()
Client

```
/**
 * Flyweight Interface
 *
 */
public interface Soldier {

        /**
         * Move Soldier From Old Location to New Location
         * Note that soldier location is extrinsic
         *    to the SoldierFlyweight Implementation
         * @param previousLocationX
         * @param previousLocationY
         * @param newLocationX
         * @param newLocationY
         */
        public void moveSoldier(int previousLocationX,
                int previousLocationY , int newLocationX ,int newLocationY);
}

public class SoldierImp implements Soldier {

        /**
         * Intrinsic State maintained by flyweight implementation
         * Solider Shape ( graphical represetation)
         * how to display the soldier is up to the flyweight implementation
         */
        private Object soldierGraphicalRepresentation;

        /**
         * Note that this method accepts soldier location
         * Soldier Location is Extrinsic and no reference to previous location
         * or new location is maintained inside the flyweight implementation
         */
        public void moveSoldier(int previousLocationX, int previousLocationY,
                        int newLocationX, int newLocationY) {

                // delete soldier representation from previous location
                // then render soldier representation in new location
        }
}

/**
 * Flyweight Factory
 */
public class SoldierFactory {

        /**
         * Pool for one soldier only
         * if there are more soldier types
         * this can be an array or list or better a HashMap
         *
         */
        private static Soldier SOLDIER;

        /**
         * getFlyweight
```

```
	 * @return
	 */
	public static Soldier getSoldier(){

		// this is a singleton
		// if there is no soldier
		if(SOLDIER==null){

			// create the soldier
			SOLDIER = new SoldierImp();
		}

		// return the only soldier reference
		return SOLDIER;
	}
}

/**
 * This is the "Heavyweight" soldier object
 * which is the client of the flyweight soldier
 *  this object provides all soldier services and is used in the game
 */
public class SoldierClient {

	/**
	 * Reference to the flyweight
	 */
	private Soldier soldier = SoldierFactory.getSoldier();

	/**
	 * this state is maintained by the client
	 */
	private int currentLocationX = 0;

	/**
	 * this state is maintained by the client
	 */
	private int currentLocationY=0;


	public void moveSoldier(int newLocationX, int newLocationY){

		// here the actual rendering is handled by the flyweight object
		soldier.moveSoldier(currentLocationX,
			currentLocationY, newLocationX, newLocationY);

		// this object is responsible for maintaining the state
		// that is extrinsic to the flyweight
		currentLocationX = newLocationX;

		currentLocationY = newLocationY;
	}
}

/**
 * Driver : War Game
 */
public class WarGame {

        public static void main(String[] args) {
                // start war

                // draw war terrain

                // create 5 soldiers:
                SoldierClient warSoldiers [] ={
                                new SoldierClient(),
                                new SoldierClient(),
                                new SoldierClient(),
                                new SoldierClient(),
                                new SoldierClient()
                };

                // move each soldier to his location
                // take user input to move each soldier
                warSoldiers[0].moveSoldier(17, 2112);

                //      take user input to move each soldier
                warSoldiers[1].moveSoldier(137, 112);

                // note that there is only one SoldierImp ( flyweight Imp)
                // for all the 5 soldiers
                // Soldier Client size is small due to the small state it maintains
                // SoliderImp size might be large or might be small
                // however we saved memory costs of creating 5 Soldier representations
        }
}
```

S7. Proxy

- A placeholder for another object to control access to it
- *When to use it?*
    - Whenever there is a need for more sophisticated reference to an object than a simple pointer



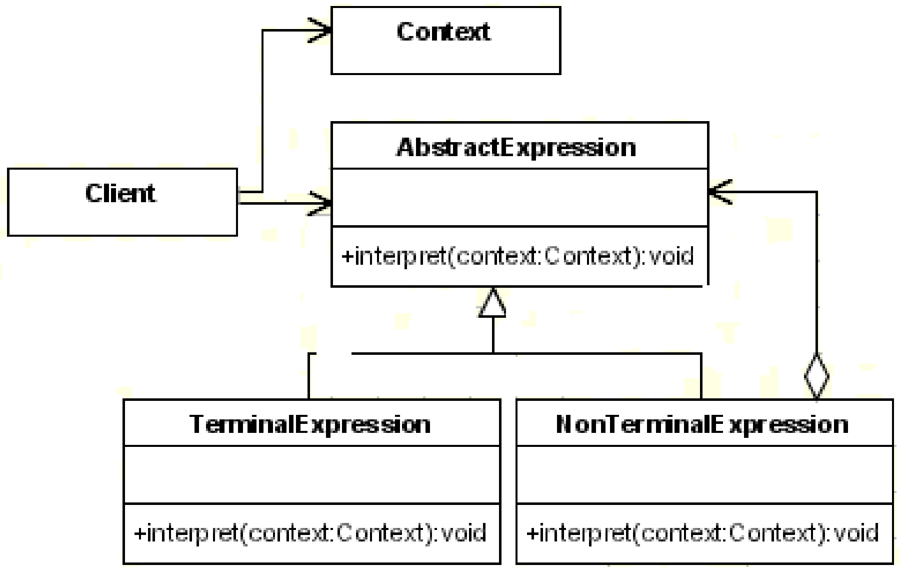
```
/**
 * Subject Interface
 */
public interface Image {

        public void showImage();

}

/**
 * Proxy
 */
public class ImageProxy implements Image {

        /**
         * Private Proxy data
         */
        private String imageFilePath;

        /**
         * Reference to RealSubject
         */
        private Image proxifiedImage;


        public ImageProxy(String imageFilePath) {
                this.imageFilePath= imageFilePath;
        }

        @Override
        public void showImage() {

                // create the Image Object only when the image is required to be shown

                proxifiedImage = new HighResolutionImage(imageFilePath);
        // now call showImage on realSubject
        proxifiedImage.showImage();

    }

}

/**
 * RealSubject
 */
public class HighResolutionImage implements Image {

    public HighResolutionImage(String imageFilePath) {

        loadImage(imageFilePath);
    }

    private void loadImage(String imageFilePath) {

        // load Image from disk into memory
        // this is heavy and costly operation
    }

    @Override
    public void showImage() {

        // Actual Image rendering logic

    }

}

/**
 * Image Viewer program
 */
public class ImageViewer {


    public static void main(String[] args) {

    // assuming that the user selects a folder that has 3 images
    //create the 3 images
    Image highResolutionImage1 = new ImageProxy("sample/veryHighResPhoto1.jpeg");
    Image highResolutionImage2 = new ImageProxy("sample/veryHighResPhoto2.jpeg");
    Image highResolutionImage3 = new ImageProxy("sample/veryHighResPhoto3.jpeg");

    // assume that the user clicks on Image one item in a list
    // this would cause the program to call showImage() for that image only
    // note that in this case only image one was loaded into memory
    highResolutionImage1.showImage();

    // consider using the high resolution image object directly
    Image highResolutionImageNoProxy1 = new
HighResolutionImage("sample/veryHighResPhoto1.jpeg");
```

```
		Image highResolutionImageNoProxy2 = new
HighResolutionImage("sample/veryHighResPhoto2.jpeg");
		Image highResolutionImageBoProxy3 = new
HighResolutionImage("sample/veryHighResPhoto3.jpeg");


		// assume that the user selects image two item from images list
		highResolutionImageNoProxy2.showImage();

		// note that in this case all images have been loaded into memory
		// and not all have been actually displayed
		// this is a waste of memory resources

		}

}
```

Behavioral Patterns - the way objects play and work

B1. Interpreter

- A way to include language elements in a program to match the grammer of the intended language.
- *When to use it?*
    - The grammar of the language is simple


Context
AbstractExpression
+interpret(context:Context):void
Client
TerminalExpression
+interpret(context:Context):void
NonTerminalExpression
+interpret(context:Context):void

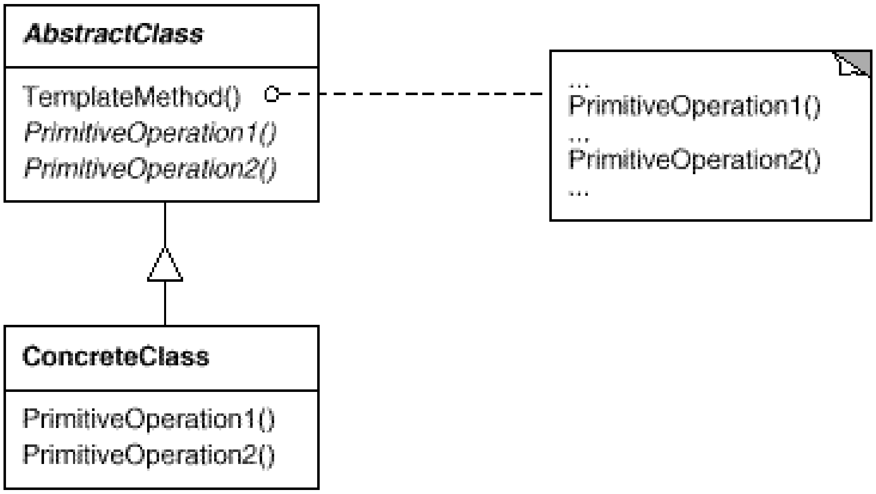
```
public abstract class Expression {
		abstract public boolean interpret(String str);
}
```

```
public class TerminalExpression extends Expression {

    private String literal = null;

    public TerminalExpression(String str) {
        literal = str;
    }

    public boolean interpret(String str) {
        StringTokenizer st = new StringTokenizer(str);
        while (st.hasMoreTokens()) {
            String test = st.nextToken();
            if (test.equals(literal)) {
                return true;
            }
        }
        return false;
    }

}

public class OrExpression extends Expression{
    private Expression expression1 = null;
    private Expression expression2 = null;

    public OrExpression(Expression expression1, Expression expression2) {
        this.expression1 = expression1;
        this.expression2 = expression2;
    }

    public boolean interpret(String str) {
        return expression1.interpret(str) || expression2.interpret(str);
    }
}

public class AndExpression extends Expression{

    private Expression expression1 = null;
    private Expression expression2 = null;

    public AndExpression(Expression expression1, Expression expression2) {
             this.expression1 = expression1;
             this.expression2 = expression2;
    }

    public boolean interpret(String str) {
        return expression1.interpret(str) && expression2.interpret(str);
    }
}

public class Main {

             /**
              * this method builds the interpreter tree
              * It defines the rule "Owen and (John or (Henry or Mary))"
```

```
          * @return
          */
    static Expression buildInterpreterTree()
    {
        // Literal
        Expression terminal1 = new TerminalExpression("John");
        Expression terminal2 = new TerminalExpression("Henry");
        Expression terminal3 = new TerminalExpression("Mary");
        Expression terminal4 = new TerminalExpression("Owen");

        // Henry or Mary
        Expression alternation1 = new OrExpression(terminal2, terminal3);

        // John or (Henry or Mary)
        Expression alternation2 = new OrExpression(terminal1, alternation1);

        // Owen and (John or (Henry or Mary))
        return new AndExpression(terminal4, alternation2);
    }


          /**
          * main method - build the interpreter
          *  and then interpret a specific sequence
          * @param args
          */
          public static void main(String[] args) {

        String context = "Mary Owen";

        Expression define = buildInterpreterTree();

        System.out.println(context + " is " + define.interpret(context));
          }
}
```

B2. Template Method

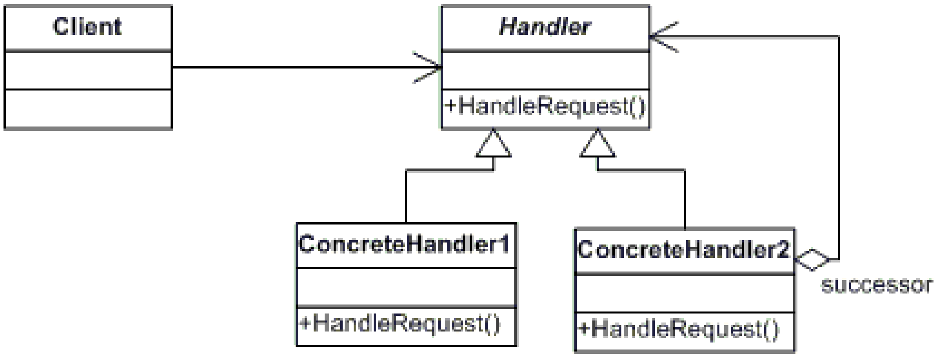
- Create the skeleton of an algorithm in a method, then defer the exact steps to a subclass. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- *When to use it?*
    - To implement the invariant parts of an algorithm once and leave it up ti subclasses to implement the behavior that can vary



```
public abstract class WorkSchedule {
   public final void threeHourOperaton(){
      startTask();
      hourOneTask();
      hourTwoTask();
      hourThreeTask();
   }
   protected abstract void startTask();
   protected abstract void hourOneTask();
   protected abstract void hourTwoTask();
   protected abstract void hourThreeTask();

}

public class CleaningOperation extends WorkSchedule{
   public  void startTask(){
      System.out.println("Cleaning opertion started");
   }
   public void hourOneTask(){
      System.out.println("Cleaning step 1");
   }
   public  void hourTwoTask(){
      System.out.println("Cleaning step 2");
   }
   public  void hourThreeTask(){
      System.out.println("Cleaning step 2");
   }
}

public class Maintenance extends WorkSchedule{
   public  void startTask(){
        System.out.println("Maintenance task started");
    }
    public void hourOneTask(){
        System.out.println("Maintenance step 1");
    }
    public  void hourTwoTask(){
        System.out.println("Maintenance step 2");
    }
    public  void hourThreeTask(){
        System.out.println("Maintenance step 3");
    }
}
public class WorkScheduler {

    public static void main(String[] args){
        WorkSchedule cleaningWork = new CleaningOperation();
        WorkSchedule maintenance = new Maintenance();
        cleaningWork.threeHourOperaton();
        maintenance.threeHourOperaton();
    }

}
```

B3. Chain of Responsibility

- A way of passing a request between a chain of objects to find the object that can handle the request. In this way avoids coupling the sender o request to its receiver by giving more than one object a chance to handle the request
- Reduces coupling
- Adds flexibility in assigning responsibilities to objects
- *When to use it?*
    - More than one object can handle a request and the handler isn't known
    - The set of objects that can handle a request should be specified dynamically

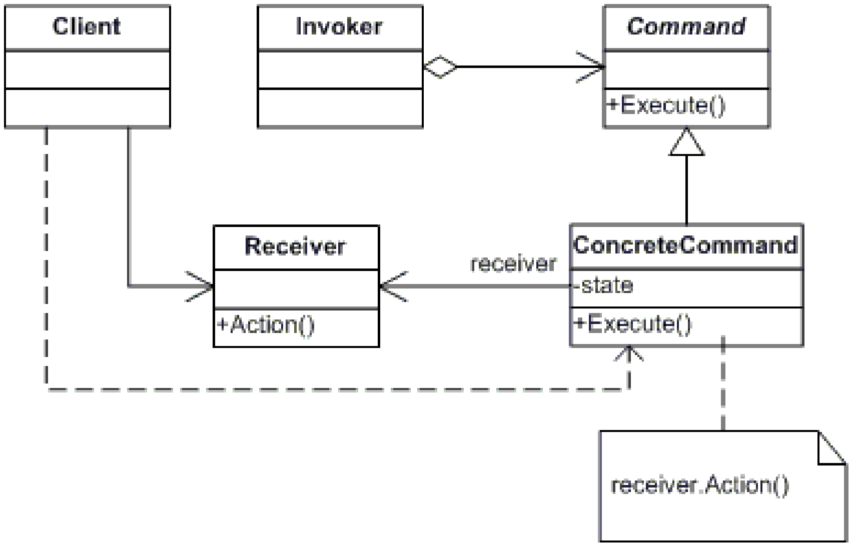


```
public class Request {
    private int m_value;
    private String m_description;

    public Request(String description, int value){
        m_description = description;
        m_value = value;
    }

    public int getValue(){
        return m_value;
    }

    public String getDescription(){
        return m_description;
    }
}

public abstract class Handler{
    protected Handler m_successor;
    public void setSuccessor(Handler successor){
        m_successor = successor;
    }

    public abstract void handleRequest(Request request);
}

public class ConcreteHandlerOne extends Handler{
    public void handleRequest(Request request){
        if (request.getValue() < 0){
             //if request is eligible handle it
            System.out.println("Negative values are handled by ConcreteHandlerOne:");
            System.out.println("\tConcreteHandlerOne.HandleRequest : " +
request.getDescription()+ request.getValue());
        }
        else{
            super.handleRequest(request);
        }
    }
 }

public class ConcreteHandlerThree extends Handler{
    public void handleRequest(Request request){
        if (request.getValue() >= 0){
             //if request is eligible handle it
            System.out.println("Zero values are handled by ConcreteHandlerThree:");
            System.out.println("\tConcreteHandlerThree.HandleRequest : " +
request.getDescription() + request.getValue());
        }
        else{
            super.handleRequest(request);
        }
    }
}
```

```
public class ConcreteHandlerTwo extends Handler]{
        public void handleRequest(Request request){
                if (request.getValue() > 0)
                {       //if request is eligible handle it
                        System.out.println("Positive values are handled by ConcreteHandlerTwo:");
                        System.out.println("\tConcreteHandlerTwo.HandleRequest : " +
request.getDescription() + request.getValue());
                }
     else{
                        super.handleRequest(request);
                }
        }
}

public class Main {
        public static void main(String[] args) {
                // Setup Chain of Responsibility
                Handler h1 = new ConcreteHandlerOne();
                Handler h2 = new ConcreteHandlerTwo();
                Handler h3 = new ConcreteHandlerThree();
                h1.setSuccessor(h2);
                h2.setSuccessor(h3);

                // Send requests to the chain
                h1.handleRequest(new Request("Negative Value ", -1));
                h1.handleRequest(new Request("Negative Value ",  0));
                h1.handleRequest(new Request("Negative Value ",  1));
                h1.handleRequest(new Request("Negative Value ",  2));
                h1.handleRequest(new Request("Negative Value ", -5));
        }
}
```

B4. Command

- Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
- *When to use it?*
    - When it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
    - You specify queue and execute requests at different times
    - You must support undo, logging or transactions

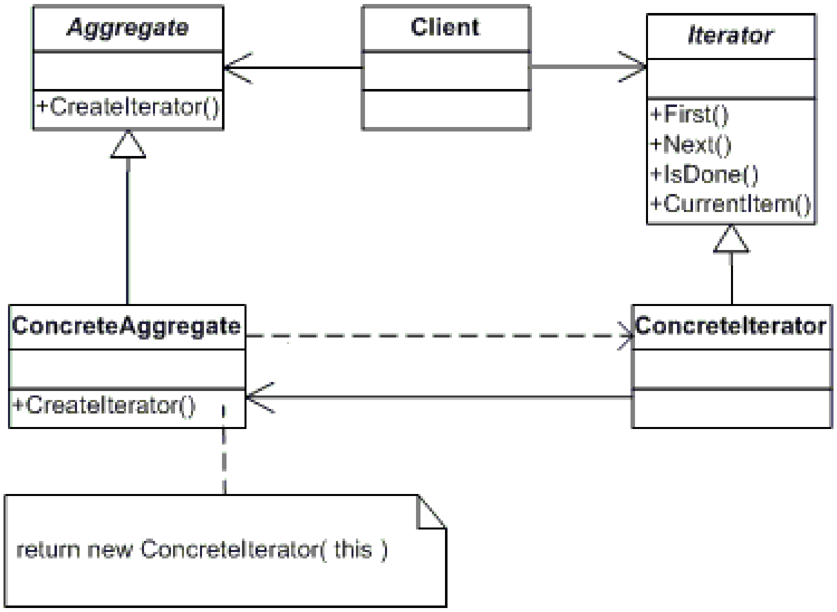


```
public interface Order {
   public abstract void execute ( );
}
// Receiver class.
class StockTrade {
   public void buy() {
      System.out.println("You want to buy stocks");
   }
   public void sell() {
      System.out.println("You want to sell stocks ");
   }
}

// Invoker.
class Agent {
   private m_ordersQueue = new ArrayList();

   public Agent() {
   }

   void placeOrder(Order order) {
      ordersQueue.addLast(order);
      order.execute(ordersQueue.getFirstAndRemove());
   }
}

//ConcreteCommand Class.
class BuyStockOrder implements Order {
   private StockTrade stock;
   public BuyStockOrder ( StockTrade st) {
      stock = st;
   }
   public void execute( ) {
      stock . buy( );
```

```
    }
}

//ConcreteCommand Class.
class SellStockOrder implements Order {
    private StockTrade stock;
    public SellStockOrder ( StockTrade st) {
        stock = st;
    }
    public void execute( ) {
        stock . sell( );
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        StockTrade stock = new StockTrade();
        BuyStockOrder bsc = new BuyStockOrder (stock);
        SellStockOrder ssc = new SellStockOrder (stock);
        Agent agent = new Agent();

        agent.placeOrder(bsc); // Buy Shares
        agent.placeOrder(ssc); // Sell Shares
    }
}
```

B5. Iterator

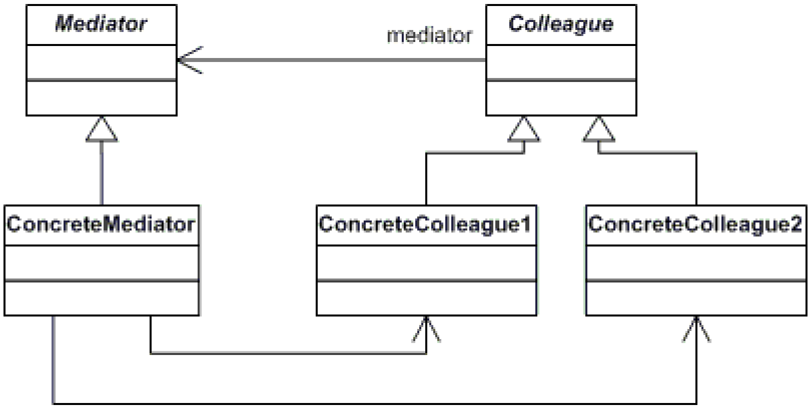
- Sequentially access the elements of a collection without knowing the inner workings of the collection.
- *When to use it?*
    - We want to access a collection object’s contents without exposing its internal representation
    - Provide a uniform interface for traversing different structures in a collection.



```
interface IIterator{
        public boolean hasNext();
        public Object next();
}

interface IContainer{
        public IIterator createIterator();
}

class BooksCollection implements IContainer{
        private String m_titles[] = {"Design Patterns","1","2","3","4"};

        public IIterator createIterator(){
                BookIterator result = new BookIterator();
                return result;
        }


        private class BookIterator implements IIterator{
                private int m_position;

                public boolean hasNext(){
                        if (m_position < m_titles.length)
                                return true;
                        else
                                return false;
                }
                public Object next(){
                        if (this.hasNext())
                                return m_titles[m_position++];
                        else
                                return null;
                }
        }}
```

B6. Mediator

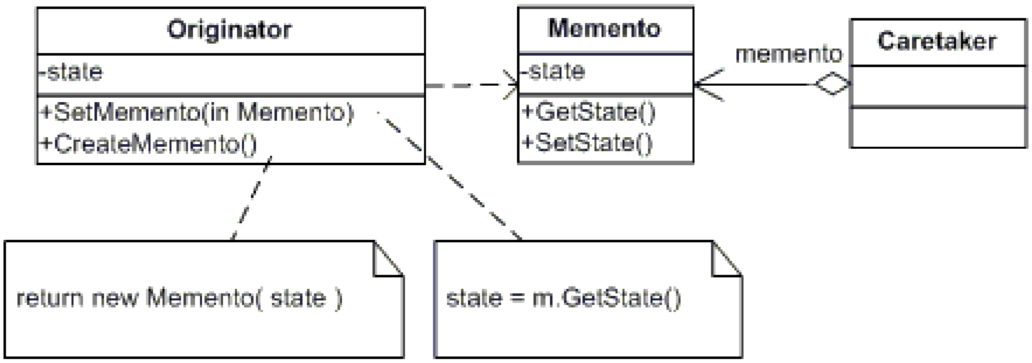
- Defines an object that encapsulates how a set of objects interacts. In a complex object structure where they are many connections between objects, every object must has knowledge of the other. We can avoid this by defining a separate mediator object that is responsible for controlling and coordinating the interaction of a group of objects (like a router for a network hosts).
- *When to use it?*
  - A set of objects of objects communicates in well-defined but complex ways.
  - Reusing an object is difficult because it refers to and communicates with many objects.



```
// 1. The "intermediary"
class Mediator {
  // 4. The Mediator arbitrates
  private boolean slotFull = false;
  private int number;
  public synchronized void storeMessage( int num ) {
    // no room for another message
    while (slotFull == true) {
      try {
        wait();
      }
      catch (InterruptedException e ) { }
    }
    slotFull = true;
    number = num;
    notifyAll();
  }
  public synchronized int retrieveMessage() {
    // no message to retrieve
    while (slotFull == false)
      try {
        wait();
      }
      catch (InterruptedException e ) { }
    slotFull = false;
    notifyAll();
    return number;
  }
}

class Producer extends Thread {
  // 2. Producers are coupled only to the Mediator
  private Mediator med;
  private int    id;
  private static int num = 1;
  public Producer( Mediator m ) {
    med = m;
    id = num++;
  }
  public void run() {
    int num;
    while (true) {
      med.storeMessage( num = (int)(Math.random()*100) );
      System.out.print( "p" + id + "-" + num + "  " );
    }
  }
}

class Consumer extends Thread {
  // 3. Consumers are coupled only to the Mediator
  private Mediator med;
  private int    id;
  private static int num = 1;
  public Consumer( Mediator m ) {
    med = m;
    id = num++;
  }
  public void run() {
    while (true) {
      System.out.print("c" + id + "-" + med.retrieveMessage() + "  ");
    }
  }
}

class MediatorDemo {
  public static void main( String[] args ) {
    Mediator mb = new Mediator();
    new Producer( mb ).start();
    new Producer( mb ).start();
    new Consumer( mb ).start();
    new Consumer( mb ).start();
    new Consumer( mb ).start();
    new Consumer( mb ).start();
  }
}
```

B7. Memento

- Capture an object's internal state to be able to restore it later – violates encapsulation
- Is an object that stores a snapshot of an internal state of another object
- *When to use it?*
    - A snapshot of an object’s state must be saved so it can be restored to that state later.
    - A direct interface to obtain the state would expose implementation details and break te object encapsulation.

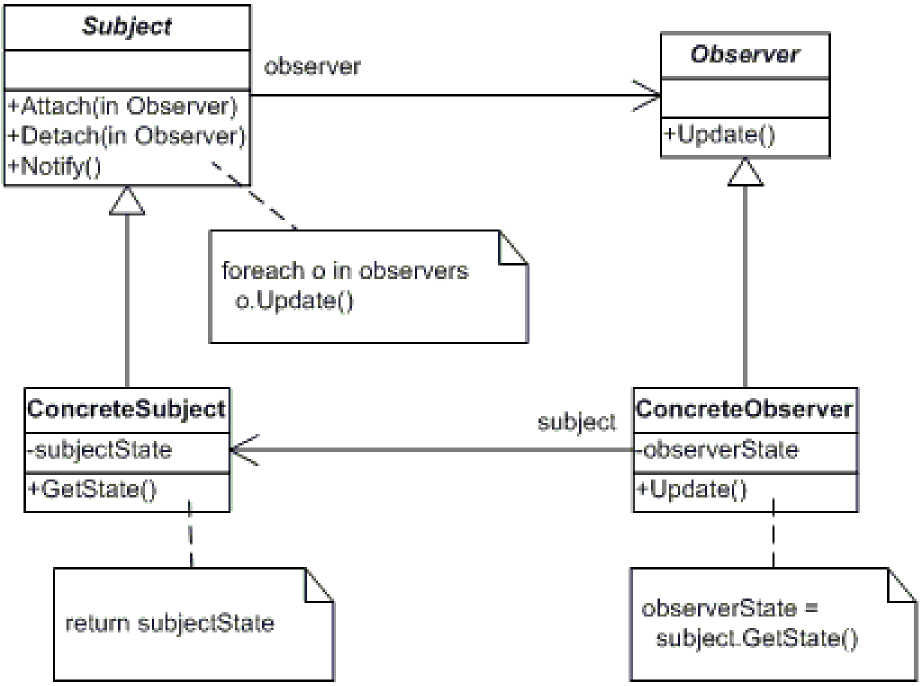


```
/**
 *  Memento interface to CalculatorOperator (Caretaker)
 */
public interface PreviousCalculationToCareTaker {
        // no operations permitted for the caretaker
}
/**
 * Memento Interface to Originator
 *
 * This interface allows the originator to restore its state
 */
public interface PreviousCalculationToOriginator {

        public int getFirstNumber();
        public int getSecondNumber();
}
/**
 * Memento Object Implementation
 *
 * Note that this object implements both interfaces to Originator and CareTaker
 */
public class PreviousCalculationImp implements PreviousCalculationToCareTaker,
                PreviousCalculationToOriginator {

        private int firstNumber;
        private int secondNumber;

        public PreviousCalculationImp(int firstNumber, int secondNumber) {
```

```
        this.firstNumber =  firstNumber;
        this.secondNumber = secondNumber;
    }

    @Override
    public int getFirstNumber() {

        return firstNumber;
    }

    @Override
    public int getSecondNumber() {

        return secondNumber;
    }
}

/**
 * Originator Interface
 */
public interface Calculator {

    // Create Memento
    public PreviousCalculationToCareTaker backupLastCalculation();

    // setMemento
    public void restorePreviousCalculation(PreviousCalculationToCareTaker memento);

    // Actual Services Provided by the originator
    public int getCalculationResult();
    public void setFirstNumber(int firstNumber);
    public void setSecondNumber(int secondNumber);
}

/**
 * Originator Implementation
 */
public class CalculatorImp implements Calculator {

    private int firstNumber;
    private int secondNumber;

    @Override
    public PreviousCalculationToCareTaker backupLastCalculation() {

        // create a memento object used for restoring two numbers
        return new PreviousCalculationImp(firstNumber,secondNumber);
    }

    @Override
    public int getCalculationResult() {

        // result is adding two numbers
        return firstNumber + secondNumber;
    }
    @Override
    public void restorePreviousCalculation(PreviousCalculationToCareTaker memento) {

        this.firstNumber = ((PreviousCalculationToOriginator)memento).getFirstNumber();
        this.secondNumber = ((PreviousCalculationToOriginator)memento).getSecondNumber();
    }

    @Override
    public void setFirstNumber(int firstNumber) {

        this.firstNumber =  firstNumber;
    }

    @Override
    public void setSecondNumber(int secondNumber) {

        this.secondNumber = secondNumber;
    }
}
/**
 * CareTaker object
 */
public class CalculatorDriver {

    public static void main(String[] args) {

        // program starts
        Calculator calculator = new CalculatorImp();

        // assume user enters two numbers
        calculator.setFirstNumber(10);
        calculator.setSecondNumber(100);

        // find result
        System.out.println(calculator.getCalculationResult());

        // Store result of this calculation in case of error
        PreviousCalculationToCareTaker memento = calculator.backupLastCalculation();

        // user enters a number
        calculator.setFirstNumber(17);

        // user enters a wrong second number and calculates result
        calculator.setSecondNumber(-290);

        // calculate result
        System.out.println(calculator.getCalculationResult());

        // user hits CTRL + Z to undo last operation and see last result
        calculator.restorePreviousCalculation(memento);

        // result restored
        System.out.println(calculator.getCalculationResult());
    }
}
```

B8. Observer

- A way of notifying change to a number of classes to ensure consistency between the classes.
- *When to use it?*
  - A need to support the broadcast communication
  - A change to one object requires changing others and we don't know how many objects need to be changed
  - An object should be able to notify other objects without making assumptions about who these objects are.

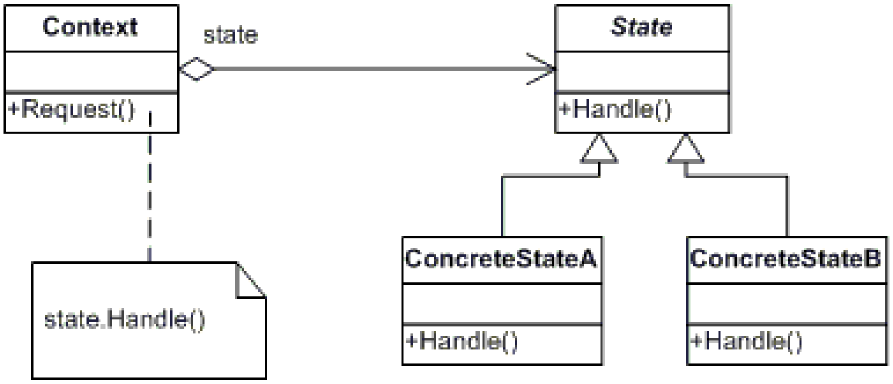


```
abstract class Observer {
protected Subject subj;
  public abstract void update();
}

class HexObserver extends Observer {
  public HexObserver( Subject s ) {
    subj = s;
    subj.attach( this );
  }

  public void update() {
  System.out.print( " " + Integer.toHexString( subj.getState() ) );
  }
} // Observers "pull" information
```

```
class OctObserver extends Observer {
  public OctObserver( Subject s ) {
    subj = s;
    subj.attach( this );
  }
  public void update() {
    System.out.print( " " + Integer.toOctalString( subj.getState() ) );
  }
} // Observers "pull" information

class BinObserver extends Observer {
  public BinObserver( Subject s ) {
    subj = s;
    subj.attach( this ); } // Observers register themselves
    public void update() {
    System.out.print( " " + Integer.toBinaryString( subj.getState() ) );
  }
}

class Subject {
  private Observer[] observers = new Observer[9];
  private int totalObs = 0;
  private int state;
  public void attach( Observer o ) {
    observers[totalObs++] = o;
  }

  public int getState() {
    return state;
  }

  public void setState( int in ) {
    state = in;
    notify();
  }

  private void notify() {
    for (int i=0; i < totalObs; i++) {
      observers[i].update();
    }
  }
}

public class ObserverDemo {
  public static void main( String[] args ) {
    Subject sub = new Subject();
    // Client configures the number and type of Observers
    new HexObserver( sub );
    new OctObserver( sub );
    new BinObserver( sub );
    Scanner scan = new Scanner();
    while (true) {
      System.out.print( "\nEnter a number: " );
      sub.setState( scan.nextInt() );
    }}}
```

B9. State

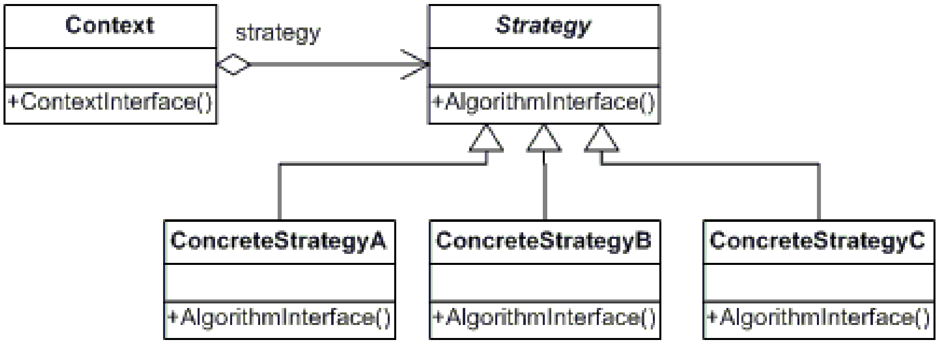
- Alter an object's behavior when its state changes
- *When to use it?*
    - An object’s behavior depends on its state and it must change its behavior at runtime depending on that state.
    - Operations have large, multipart conditional statements that depend on the object’s state.



```
class Chain {
   private State current;
   public Chain()                  { current = new Off(); }
   public void setState( State s ) { current = s; }
   public void pull()              { current.pull( this ); }
}

abstract class State {
   public void pull( Chain wrapper ) {
      wrapper.setState( new Off() );
      System.out.println( "   turning off" );
}  }

class Off extends State {
   public void pull( Chain wrapper ) {
      wrapper.setState( new Low() );
      System.out.println( "   low speed" );
}  }

class Low extends State {
   public void pull( Chain wrapper ) {
      wrapper.setState( new Medium() );
      System.out.println( "   medium speed" );
}  }

class Medium extends State {
   public void pull( Chain wrapper ) {
      wrapper.setState( new High() );
      System.out.println( "   high speed" );
}  }

class High extends State { }

public class StateDisc {
   public static void main( String[] args ) throws IOException {
      InputStreamReader is = new InputStreamReader( System.in );
      int ch;
      Chain chain = new Chain();
      while (true) {
         System.out.print( "Press 'Enter'" );
         ch = is.read();    ch = is.read();
         chain.pull();
}  }  }
```

B10. Strategy

- Defines a group of classes that represent a set of possible behaviors. The functionality differs depending on the strategy (algorithm) chosen.
- *When to use it?*
  - Many related classes differ only in their behavior
  - We need different variants of an algorithm.



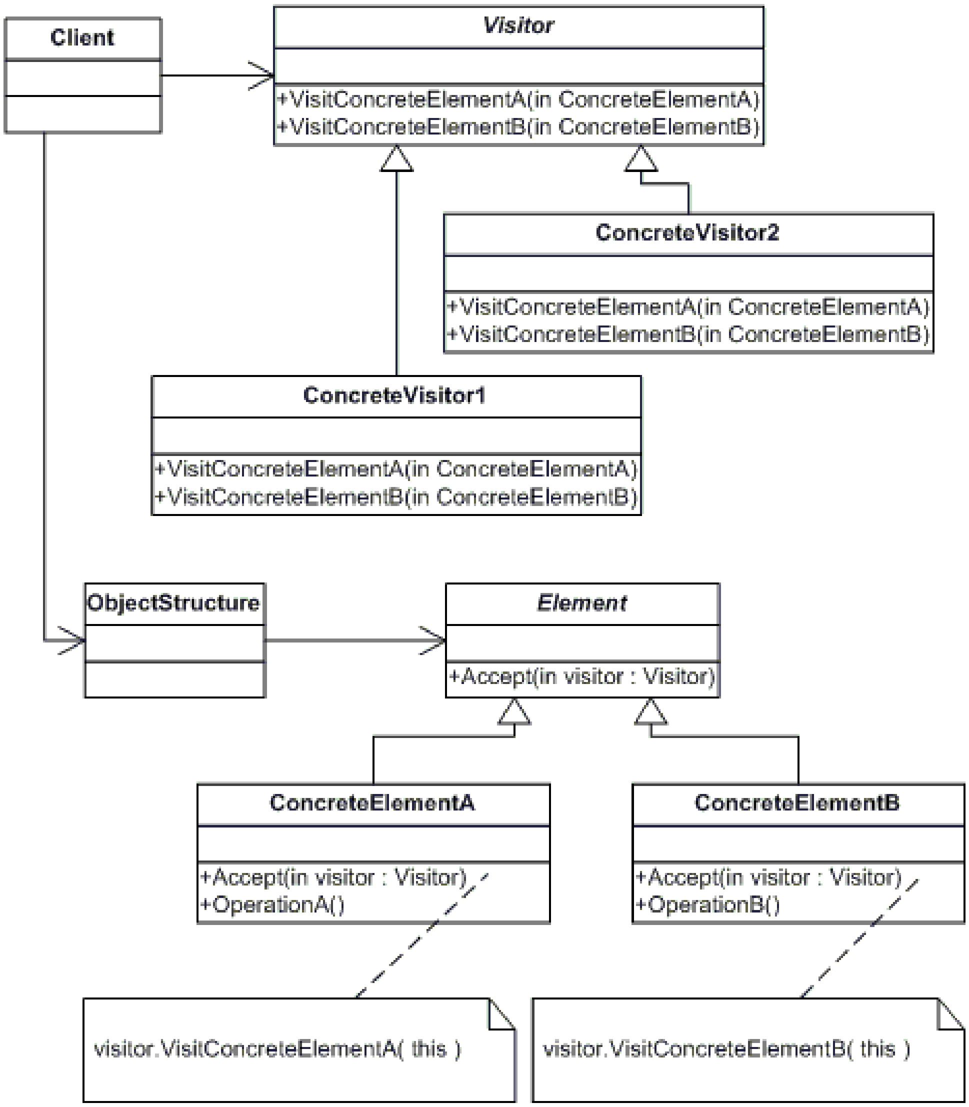
```
public interface IBehaviour {
          public int moveCommand();
}

public class AgressiveBehaviour implements IBehaviour{
          public int moveCommand(){
                    System.out.println("\tAgressive Behaviour: if find another robot attack it");
                    return 1;
          }
}

public class DefensiveBehaviour implements IBehaviour{
          public int moveCommand(){
                    System.out.println("\tDefensive Behaviour: if find another robot run from it");
```

```
                return -1;
        }
}

public class NormalBehaviour implements IBehaviour{
        public int moveCommand(){
                System.out.println("\tNormal Behaviour: if find another robot ignore it");
                return 0;
        }
}

public class Robot {
        IBehaviour behaviour;
        String name;

        public Robot(String name){
                this.name = name;
        }

        public void setBehaviour(IBehaviour behaviour){
                this.behaviour = behaviour;
        }

        public IBehaviour getBehaviour(){
                return behaviour;
        }

        public void move(){
                System.out.println(this.name + ": Based on current position" +
                                         "the behaviour object decide the next move:");
                int command = behaviour.moveCommand();
                // ... send the command to mechanisms
                System.out.println("\tThe result returned by behaviour object " +
                                        "is sent to the movement mechanisms " +
                                        " for the robot '"  + this.name + "'");
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
}

public class Main {

        public static void main(String[] args) {

                Robot r1 = new Robot("Big Robot");
                Robot r2 = new Robot("George v.2.1");
                Robot r3 = new Robot("R2");

                r1.setBehaviour(new AgressiveBehaviour());
                r2.setBehaviour(new DefensiveBehaviour());
```

```
		r3.setBehaviour(new NormalBehaviour());

		r1.move();
		r2.move();
		r3.move();

		System.out.println("\r\nNew behaviours: " +
				"\r\n\t'Big Robot' gets really scared" +
				"\r\n\t, 'George v.2.1' becomes really mad because" +
				"it's always attacked by other robots" +
				"\r\n\t and R2 keeps its calm\r\n");

		r1.setBehaviour(new DefensiveBehaviour());
		r2.setBehaviour(new AgressiveBehaviour());

		r1.move();
		r2.move();
		r3.move();
	}
}
```

B11. Visitor

- Adds a new operation to a class without changing the class.
- *When to use it?*
  - When we need to make adding new operations easy
  - When the classes defining the object structure rarely change, but we often want to define new operations over the structure.


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

```
interface Element {
  // 1. accept(Visitor) interface
  public void accept( Visitor v ); // first dispatch
}

class This implements Element {
  // 1. accept(Visitor) implementation
  public void   accept( Visitor v ) {
    v.visit( this );
  }
  public String thiss() {
    return "This";
  }
}

class That implements Element {
  public void   accept( Visitor v ) {
    v.visit( this );
```

```
   }
   public String that() {
     return "That";
   }
}

class TheOther implements Element {
   public void   accept( Visitor v ) {
     v.visit( this );
   }
   public String theOther() {
     return "TheOther";
   }
}

// 2. Create a "visitor" base class with a visit() method for every "element" type
interface Visitor {
   public void visit( This e ); // second dispatch
   public void visit( That e );
   public void visit( TheOther e );
}

// 3. Create a "visitor" derived class for each "operation" to perform on "elements"
class UpVisitor implements Visitor {
   public void visit( This e ) {
      System.out.println( "do Up on " + e.thiss() );
   }
   public void visit( That e ) {
      System.out.println( "do Up on " + e.that() );
   }
   public void visit( TheOther e ) {
      System.out.println( "do Up on " + e.theOther() );
   }
}

class DownVisitor implements Visitor {
   public void visit( This e ) {
      System.out.println( "do Down on " + e.thiss() );
   }
   public void visit( That e ) {
      System.out.println( "do Down on " + e.that() );
   }
   public void visit( TheOther e ) {
      System.out.println( "do Down on " + e.theOther() );
   }
}

class VisitorDemo {
   public static Element[] list = { new This(), new That(), new TheOther() };

   // 4. Client creates "visitor" objects and passes each to accept() calls
   public static void main( String[] args ) {
      UpVisitor    up   = new UpVisitor();
      DownVisitor  down = new DownVisitor();
      for (int i=0; i < list.length; i++) {
         list[i].accept( up );
```

```
    }
    for (int i=0; i < list.length; i++) {
      list[i].accept( down );
    }
  }
}
```