GoF Design Patterns -
with examples using Java and UML2

written by:
Benneth Christiansson (Ed.)
Mattias Forss,
Ivar Hagen,
Kent Hansson,
Johan Jonasson,
Mattias Jonasson,
Fredrik Lott,
Sara Olsson, and
Thomas Rosevall

Table of Contents

Foreword

This book is the result of a joint effort of the authors with an equal contribution from all. The idea to the book originated during the participation of a Java Architect training program taught at Logica Sverige AB Karlstad office. During the course the authors identified the lack of a quick-guide book to the basic GoF[1] design patterns. A book that could be used as a bare bone reference as well as a learning companion for understanding design patterns. So we divided the workload and together we created an up-to-date view of the GoF design patterns in a structured and uniform manner. Illustrating the choosen patterns with examples in Java and diagrams using UML2 notation. We have also emphasized benefits and drawbacks for the individual patterns and, where applicable. We also illustrate real world usage situations where the pattern has successfully been implemented.

I personally as editor must express my deepest admiration for the dedication and effort the authors have shown during this process, everyone who ever have written a book knows what I mean. I am really proud to be the editor of this very usable book.

--- Benneth Christiansson, Karlstad, autumn 2008 ---

1 Design Patterns Elements of Reusable Elements by Gamma, Helm, Johnson and Vlissides (1995)

Chapter 1 Creational Patterns

"Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation."[2]

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

2 http://en.wikipedia.org/wiki/Creational_pattern

Factory Pattern

Definition

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

Where to use

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

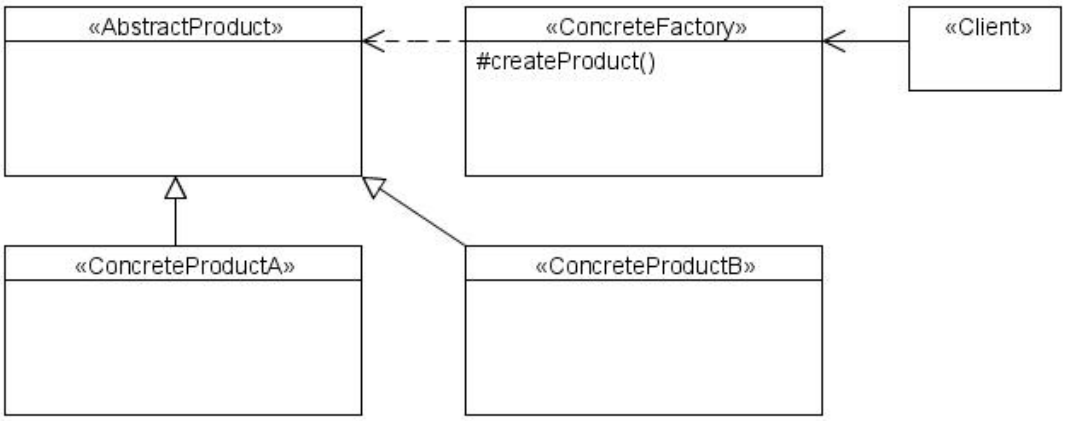
Benefits

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

Drawbacks/consequences

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

Structure



Small example

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

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


public class ProductA extends Product { }


public class ProductB extends Product {
    public void writeName(String name) {
        StringBuilder tempName = new StringBuilder().append(name);
        System.out.println("My reversed name is" +
            tempName.reverse());
    }
}


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

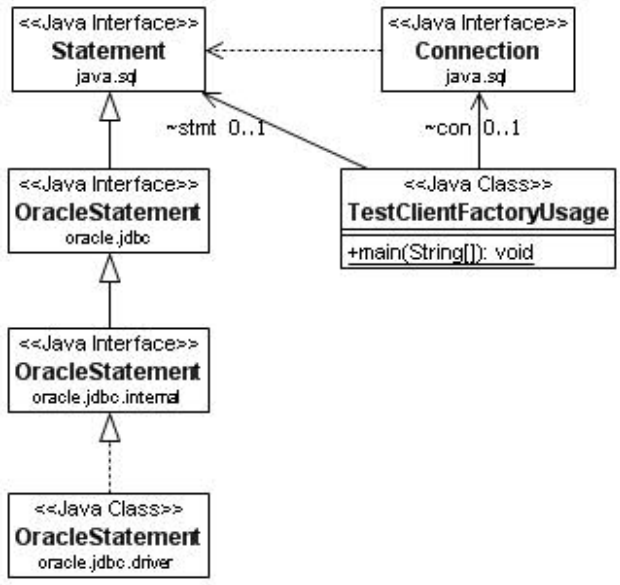
```
public class TestClientFactory {
    public static void main(String[] args) {
        ProductFactory pf = new ProductFactory();
        Product prod;
        prod = pf.createProduct("A");
        prod.writeName("John Doe");
        prod = pf.createProduct("B");
        prod.writeName("John Doe");
    }
}
```

When TestClientFactory is executed the result is:

```
c:> My name is John Doe
c:> My reversed name is eoD nhoJ
```

Usage example

The Connection object in the java package sql is a factory.
Depending on the database driver you use you get the database vendors implementation of the Statement interface. In the following example we actually get an OracleStatement object from the package oracle.jdbc.driver when calling createStatement.



```
import java.sql.*;

public class TestClientFactoryUsage {
    static Connection con;
    static Statement stmt;

    public static void main(String[] args) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("myServer", "user",
                  "password");
            stmt = con.createStatement();
         } catch(Exception e) {}
    }
}
```

Abstract Factory Pattern

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

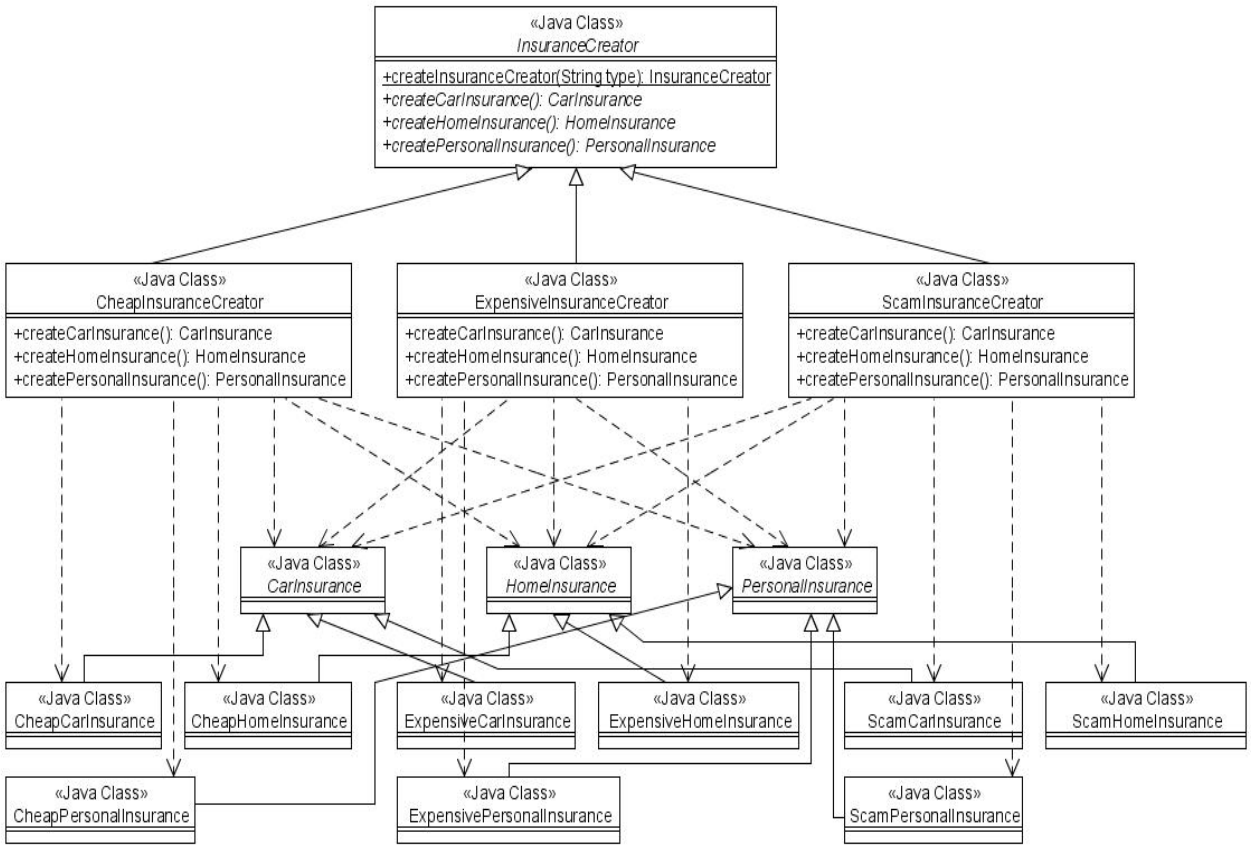
Where to use
The pattern can be used where we need to create sets of objects that share a common theme and where the client only needs to know how to handle the abstract equivalence of these objects, i.e. the implementation is not important for the client. The Abstract Factory is often employed when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

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

Drawbacks/consequences
As with similar design patterns, one of the main drawbacks is the possibility of unnecessary complexity and extra work in the initial writing of the code.

Structure

Below you see the class diagram of the following small example.



Small example

For instance, we could have an abstract class InsuranceCreator that provides interfaces to create a number of products (e.g. createCarInsurance(), createHomeInsurance(), createPersonalInsurance()). Any number of derived concrete classes of the InsuranceCreator class can be created, for example CheapInsuranceCreator, ExpensiveInsuranceCreator or ScamInsuranceCreator, each with a different implementation of createCarInsurance(), createHomeInsurance() and createPersonalInsurance() that would create a corresponding object like CheapCarInsurance, ExpensiveHomeInsurance or ScamPersonalInsurance. Each of these products is derived from a simple abstract class like CarInsurance, HomeInsurance or PersonalInsurance of which the client is aware.

The client code would get an appropriate instantiation of the InsuranceCreator and call its factory methods. Each of the resulting objects would be created from the same InsuranceCreator implementation and would share a common theme (they would all be cheap, expensive or scam objects). The client would need to know how to handle only the abstract CarInsurance, HomeInsurance or PersonalInsurance class, not the specific version that it got from the concrete factory.

Builder Pattern

Definition

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

Where to use

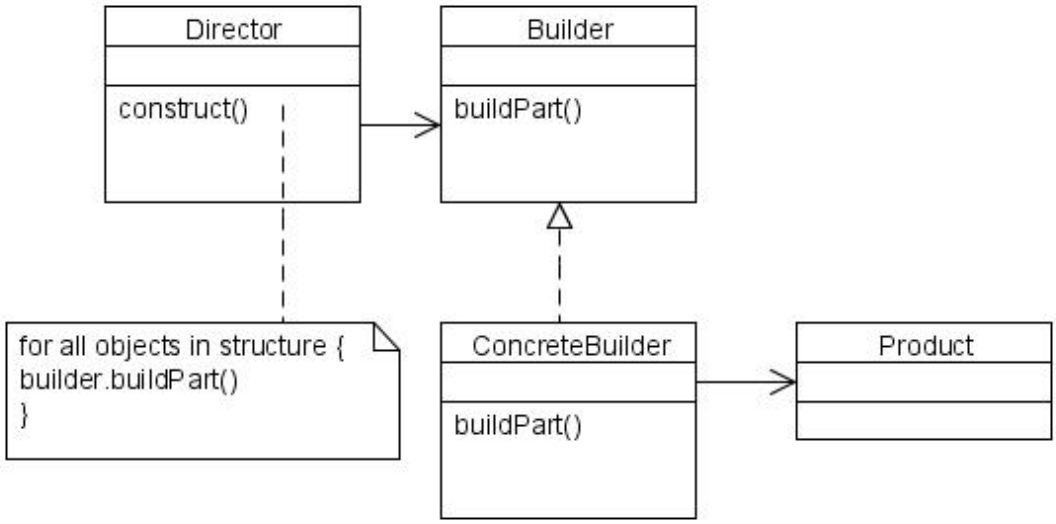
•When the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
•When the construction process must allow different representations for the object that is constructed.
•When you want to insulate clients from the knowledge of the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

Benefits

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

Drawbacks/consequences

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



Structure

In the class diagram above:
•The Builder specifies an abstract interface for creating parts of a Product.
•The ConcreteBuilder constructs and assembles parts of the product by implementing the Builder interface.
•The Director constructs an object using the Builder interface.
•The Product represents the object under construction.

Small example
For instance to build a house, we will take several steps:
1. Build floor
2. Build walls
3. Build roof

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

```
public abstract class HouseBuilder {
      protected House house;
      protected Floor floor;
      protected Walls walls;
      protected Roof roof;
      public abstract House createHouse();
      public abstract Floor createFloor();
      public abstract Walls createWalls();
      public abstract Roof createRoof();
}


public class WoodBuilder extends HouseBuilder {
      public Floor createFloor() {
            floor = new WoodFloor();
            return floor;
      }

      public House createHouse() {
            house = new WoodHouse();
            return house;
      }

      public Roof createRoof() {
            roof = new WoodRoof();
            return roof;
      }

      public Walls createWalls() {
            walls = new WoodWalls();
            return walls;
      }
}


public class BrickBuilder extends HouseBuilder {
      //similar to WoodBuilder
}


public class HouseDirector {
      public House construcHouse(HouseBuilder builder) {
            House house = builder.createHouse();
            System.out.println(house.getRepresentation());
            house.setFloor(builder.createFloor());
            System.out.println(house.getFloor().getRepresentation());
            house.setWalls(builder.createWalls());
            System.out.println(house.getWalls().getRepresentation());
            house.setRoof(builder.createRoof());
            System.out.println(house.getRoof().getRepresentation());
            return house;
      }
}
```

```
public abstract class House {
	protected Floor floor;
	protected Walls walls;
	protected Roof roof;

	public Floor getFloor() {
		return floor;
	}

	public void setFloor(Floor floor) {
		this.floor = floor;
	}

	public Walls getWalls() {
		return walls;
	}

	public void setWalls(Walls walls) {
		this.walls = walls;
	}

	public Roof getRoof() {
		return roof;
	}

	public void setRoof(Roof roof) {
		this.roof = roof;
	}

	public abstract String getRepresentation();
}


public interface Floor {
	public String getRepresentation();
}


public interface Walls {
	public String getRepresentation();
}


public interface Roof {
	public String getRepresentation();
}


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


public class WoodFloor implements Floor {
	public String getRepresentation() {
		return "Finished building wood floor";
	}
}
```

```
public class WoodWalls implements Walls {
   //similar to WoodFloor
}


public class WoodRoof implements Roof {
   //similar to WoodFloor
}


// Similar structure for Brick family
public class BrickHouse extends House …

public class BrickFloor implements Floor …

public class BrickWalls implements Walls …

public class BrickRoof implements Roof …


public class HouseClient {
      public static void main(String[] args)    {
            HouseDirector director = new HouseDirector();
            HouseBuilder woodBuilder = new WoodBuilder();
            BrickBuilder brickBuilder = new BrickBuilder();
            // Build a wooden house
            House woodHouse = director.construcHouse(woodBuilder);
            System.out.println();
            // Build a brick house
            House brickHouse = director.construcHouse(brickBuilder);
      }
}
```

When HouseClient is executed the result is:

```
c:>Building a wood house
c:>Finished building wood floor
c:>Finished building wood walls
c:>Finished building wooden roof
c:>
c:>Building a brick house
c:>Finished building brick floor
c:>Finished building brick walls
c:>Finished building brick roof
```

Usage example
Some examples of using the Builder pattern in knowledge engineering include different generators. Parsers in various compilers are also designed using the Builder pattern.

Prototype Pattern

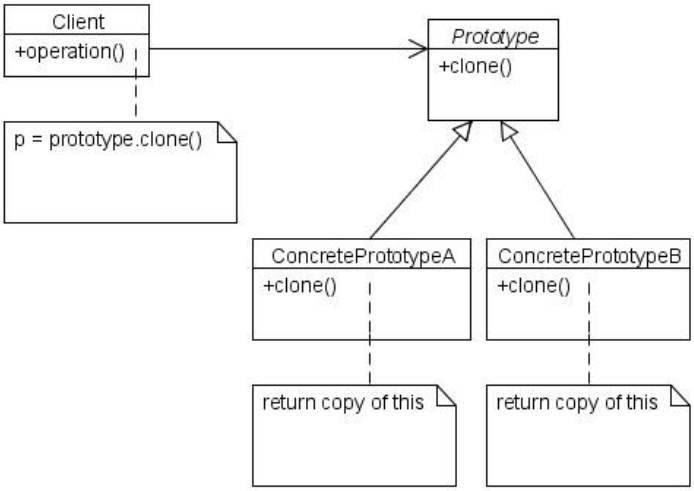
Definition
The Prototype pattern is basically the creation of new instances through cloning existing instances. By creating a prototype, new objects are created by copying this prototype.

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

Benefits
•Speeds up instantiation of large, dynamically loaded classes.
•Reduced subclassing.

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

Structure



In the class-diagram above:
•Prototype declares an interface for cloning itself.
•ConcretePrototype implements an operation for cloning itself.
•Client creates a new object by asking a prototype to clone itself.

You could use a PrototypeManager to keep track on the different types of prototypes. The PrototypeManager maintains a list of clone types and their keys. The client, instead of writing code that invokes the "new" operator on a hard-wired class name, calls the clone() method on the prototype.

Small example

```
import java.util.Hashtable;
public class PrototypeExample {
    Hashtable<String, Product> productMap = new Hashtable<String,
            Product>();

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

      public 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 static void main(String[] args) {
            PrototypeExample pe = new PrototypeExample();
            pe.loadCache();
            Book clonedBook = (Book)pe.getProduct("B1");
            System.out.println("SKU = " + clonedBook.getSKU());
            System.out.println("SKU = " +
                  clonedBook.getDescription());
            System.out.println("SKU = " +
                  clonedBook.getNumberOfPages());
            DVD clonedDVD = (DVD)pe.getProduct("D1");
            System.out.println("SKU = " + clonedDVD.getSKU());
            System.out.println("SKU = " + clonedDVD.getDescription());
            System.out.println("SKU = " + clonedDVD.getDuration());
    }
}
```

```
/** Prototype Class * */
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;
    }
}



/** Concrete Prototypes to clone * */
public class Book extends Product {
    private int numberOfPages;

    public int getNumberOfPages() {
        return numberOfPages;
    }

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

/** Concrete Prototypes to clone * */
public class DVD extends Product {
    private int duration;

    public int getDuration()    {
        return duration;
    }

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

When PrototypeExample is executed the result is:

```
c:>SKU = B1
c:>SKU = Oliver Twist
c:>SKU = 100
c:>SKU = D1
c:>SKU = Superman
c:>SKU = 180
```

Usage example

If you are designing a system for performing bank account transactions, then you would want to make a copy of the Object which holds your account information, perform transactions on it, and then replace the original Object with the modified one. In such cases, you would want to use clone() instead of new.

Singleton Pattern

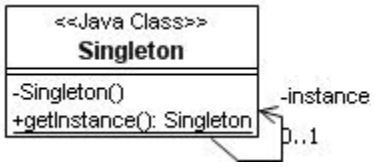
Definition
The Singleton pattern provides the possibility to control the number of instances (mostly one) that are allowed to be made. We also receive a global point of access to it (them).

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

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

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

Structure



Small example

```
package com.logica.singleton;
public class FileLogger {
	private static FileLogger logger;

	// Prevent clients from using the constructor
	private FileLogger() {
	}

	//Control the accessible (allowed) instances
	public static FileLogger getFileLogger() {
		if (logger == null) {
			logger = new FileLogger();
		}
		return logger;
	}

	public synchronized void log(String msg) {
		// Write to log file...
	}
}
```

Chapter 2 Structural Patterns

"In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities."[3]

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

3 http://en.wikipedia.org/wiki/Structural_pattern

Adapter Pattern

Also known as Wrapper.

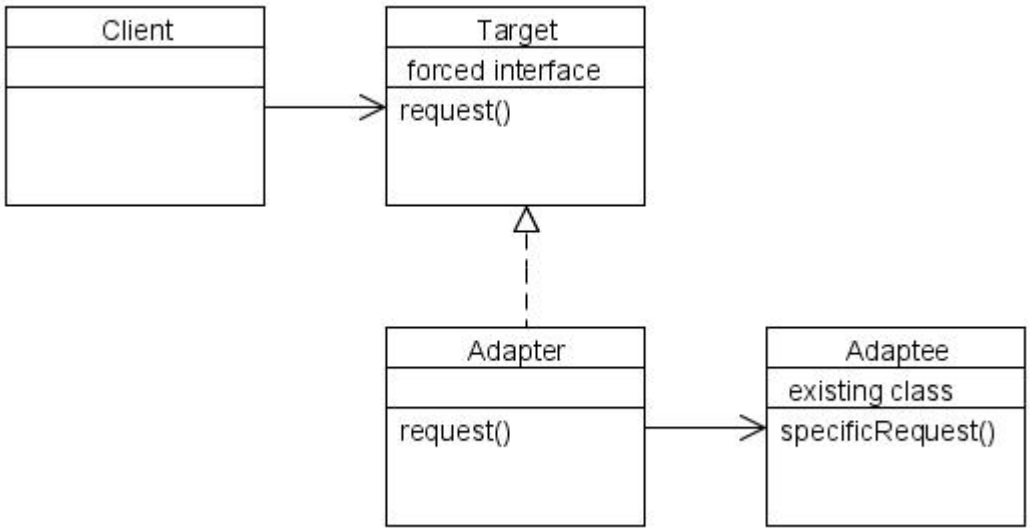
Definition
The Adapter pattern is used to translate the interface of one class into another interface. This means that we can make classes work together that couldn't otherwise because of incompatible interfaces. A class adapter uses multiple inheritance (by extending one class and/or implementing one or more classes) to adapt one interface to another. An object adapter relies on object aggregation.

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

Benefits
•Highly class reusable.
•Introduces only one object

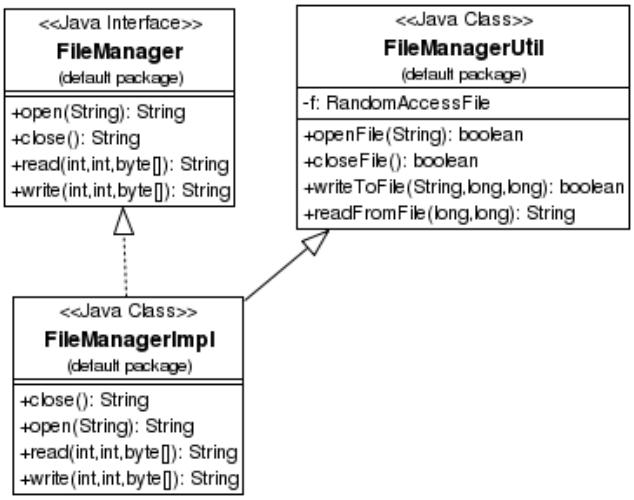
Drawbacks/consequences
When using Java, Target must be an interface.

Structure



In the class-diagram above:
•A Client class expects a certain interface (called the Target interface)
•An available interface doesn't match the Target interface
•An Adapter class bridges the gap between the Target interface and the available interface
•The available interface is called the Adaptee

Small example
Imagine you need to develop a simple file manager to handle text documents. There is an existing resource that already handles this, but by some reason you are forced to a specific interface for your file manager. By using a class adapter we can use the forced interface and still reuse the existing functionality. In the class diagram below the interface FileManager is the target (desired interface). FileManagerUtil is the existing utility class that we would like to adapt to FileManager interface. We do the actual adaptation in the class FileManagerImpl, this class uses the desired interface and the existing functionality through inheritance, i.e. a class adapter.




```
public interface FileManager {
      public String open(String s);
      public String close();
      public String read(int pos, int amount, byte[] data);
      public String write(int pos, int amount, byte[] data);
}


import java.util.*;
import java.io.*;
public class FileManagerUtil {
      private RandomAccessFile f;
      public boolean openFile(String fileName) {
            System.out.println("Opening file: "+fileName);
            boolean success=true;
            return success;
      }

      public boolean closeFile() {
            System.out.println("Closing file");
            boolean success=true;
            return success;
      }

      public boolean writeToFile(String d, long pos, long amount) {

            System.out.print("Writing "+amount+
                  " chars from string: "+d);
            System.out.println(" to pos: "+pos+" in file");
            boolean success=true;
            return success;
      }

      public String readFromFile(long pos, long amount) {
            System.out.print("Reading "+amount+
                        " chars from pos: "+pos+" in file");
            return new String("dynamite");
      }
}
```

```
public class FileManagerImpl extends FileManagerUtil implements
                                                    FileManager {
      public String close() {
            return new Boolean(closeFile()).toString();
      }

      public String open(String s) {
            return new Boolean(openFile(s)).toString();
      }

      public String read(int pos, int amount, byte[] data) {
            return readFromFile(pos, amount);
      }

      public String write(int pos, int amount, byte[] data) {
            boolean tmp= writeToFile(new String(data), pos, amount);
            return String.valueOf(tmp);
      }
}


public class FileManagerClient {
      public static void main(String[] args) {
            FileManager f = null;
            String dummyData = "dynamite";
            f = new FileManagerImpl();
            System.out.println("Using filemanager: "+
                              f.getClass().toString());
            f.open("dummyfile.dat");
            f.write(0, dummyData.length(), dummyData.getBytes());
            String test = f.read(0,dummyData.length(),
                                 dummyData.getBytes());
            System.out.println("Data written and read: "+test);
            f.close();
      }
}
```

When FileManagerClient is executed the result is:

```
c:>Opening file: dummyfile.dat
c:>Writing 8 chars from string: dynamite to pos: 0 in file
c:>Reading 8 chars from pos: 0 in fileData written and read: dynamite
c:>Closing file
```

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

Bridge Pattern

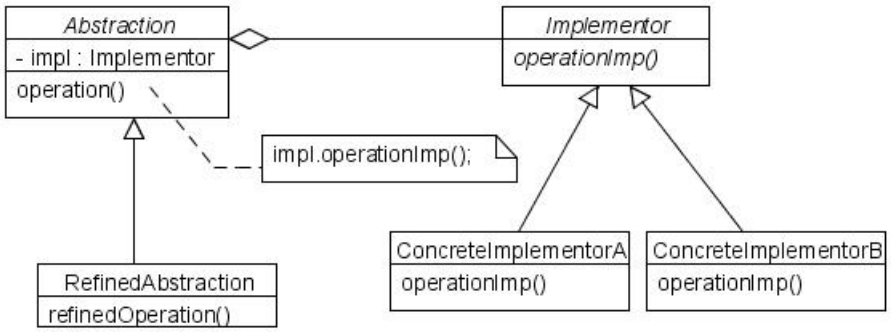
Definition
Decouple an abstraction or interface from its implementation so that the two can vary independently.
Bridge makes a clear-cut between abstraction and implementation.

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

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

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

Structure



In the class-diagram above:
•Abstraction defines the abstract interface and maintains the Implementor reference.
•Refined Abstraction extends the interface defined by Abstraction.
•Implementor defines the interface for implementation classes.
•ConcreteImplementor implements the Implementor interface.

Small example

```
public class BridgeExample {
    public static void main(String[] args)    {
        Shape[] shapes = new Shape[2];
        shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
        shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());
        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}


/** "Abstraction" */
public interface Shape {
    public void draw();
    public void resizeByPercentage(double pct);
}
```

```
/** "Refined Abstraction" */
public class CircleShape implements Shape {
    private double x, y, radius;
    private DrawingAPI drawingAPI;

    public CircleShape(double x, double y, double radius, DrawingAPI
            drawingAPI) {
        this.x = x;
        this.y = y;
        this.radius = radius;
        this.drawingAPI = drawingAPI;
    }

    // Implementation specific
    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }

    // Abstraction specific
    public void resizeByPercentage(double pct) {
        radius *= pct;
    }
}


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


/** "ConcreteImplementor" 1/2 */
public 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 */
public 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);
    }
}
```

When BridgeExample is executed the result is:

c:>API1.circle at 1,000000:2,000000 radius 7,5000000
c:>API2.circle at 5,000000:7,000000 radius 27,5000000

Usage example
Bridge pattern can be found in the AWT package. The AWT separates the general abstraction of GUI components from concrete native implementations of GUI components.

Composite Pattern

Definition

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

Where to use

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

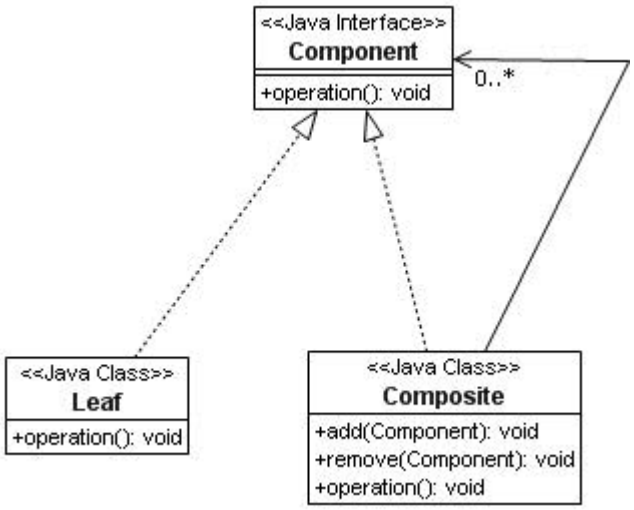
Benefits

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

Drawbacks/consequences

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

Structure



In the class-diagram above, Component:
•is the abstraction for all components, including composite ones,
•declares the interface for objects in the composition,
•implements default behavior for the interface common to all classes, as appropriate,
•declares an interface for accessing and managing its child components, and
•(optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

In the class-diagram above, Leaf:
•represents leaf objects in the composition, and
•implements all Component methods.

In the class-diagram above, Composite:
•represents a composite Component (component having children),
•implements methods to manipulate children, and
•implements all Component methods, generally by delegating them to its children.

Small example

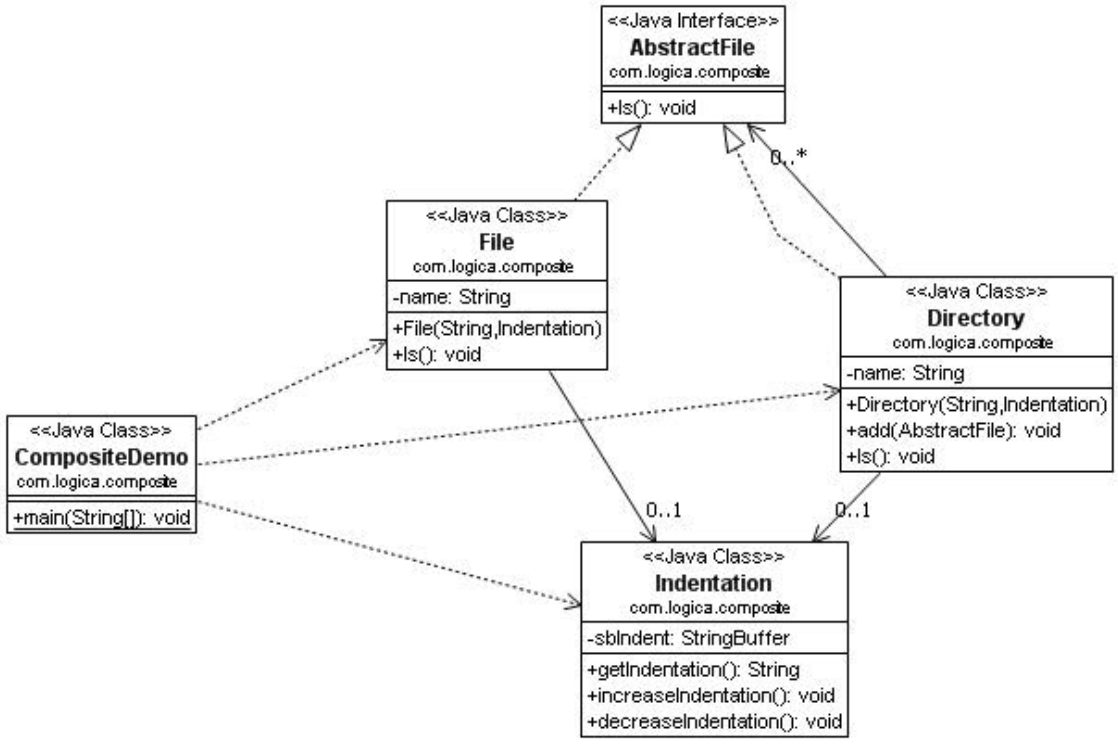
```
package com.logica.composite;
public interface AbstractFile {
       public void ls();
}


package com.logica.composite;
import java.util.ArrayList;
public class Directory implements AbstractFile {
      private String name;
      private ArrayList<AbstractFile> files = new
            ArrayList<AbstractFile>();
      private Indentation indentation;

      public Directory (String name, Indentation indentation) {
            this.name = name;
            this.indentation = indentation;
      }

      public void add(AbstractFile f) {
            files.add(f);
      }

      public void ls() {
            System.out.println(indentation.getIndentation() + name);
            indentation.increaseIndentation();
            for (AbstractFile file : files) {
                  file.ls();
            }
            indentation.decreaseIndentation();
      }
}


package com.logica.composite;
class File implements AbstractFile {
      private String name;
      private Indentation indentation;

      public File(String name, Indentation indentation) {
            this.name = name;
            this.indentation = indentation;
      }

      public void ls() {
            System.out.println(indentation.getIndentation() +
            name);
      }
}
```

```
package com.logica.composite;
public class Indentation {
      private StringBuffer sbIndent = new StringBuffer();

      public String getIndentation() {
            return sbIndent.toString();
      }

      public void increaseIndentation() {
            sbIndent.append("   ");
      }

      public void decreaseIndentation() {
            if (sbIndent.length() >= 3) {
                  sbIndent.setLength(sbIndent.length() - 3);

            }
      }
}


package com.logica.composite;
public class CompositeDemo {
       public static void main(String[] args) {
             Indentation indentation = new Indentation();
             Directory dirOne = new Directory("dir111", indentation);
             Directory dirTwo = new Directory("dir222", indentation);
             Directory dirThree = new Directory("dir333",
                  indentation);
             File a = new File("a", indentation);
             File b = new File("b", indentation);
             File c = new File("c", indentation);
             File d = new File("d", indentation);
             File e = new File("e", indentation);
             dirOne.add(a);
             dirOne.add(dirTwo);
             dirOne.add(b);
             dirTwo.add(c);
             dirTwo.add(d);
             dirTwo.add(dirThree);
             dirThree.add(e);
             dirOne.ls();
       }
}
```

When BridgeExample is executed the result is:

```
c:>dir111
c:>     a
c:>     dir222
c:>             c
c:>             d
c:>             dir333
c:>
c:>                     e
c:>
c:>     b
```

Below is the class-diagram for the small example.



Decorator Pattern

Definition

The Decorator pattern lets you attach additional responsibilities and modify an instance functionality dynamically. Decorators provide a flexible alternative to subclassing for extending functionality, using composition instead of inheritance.

Where to use

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

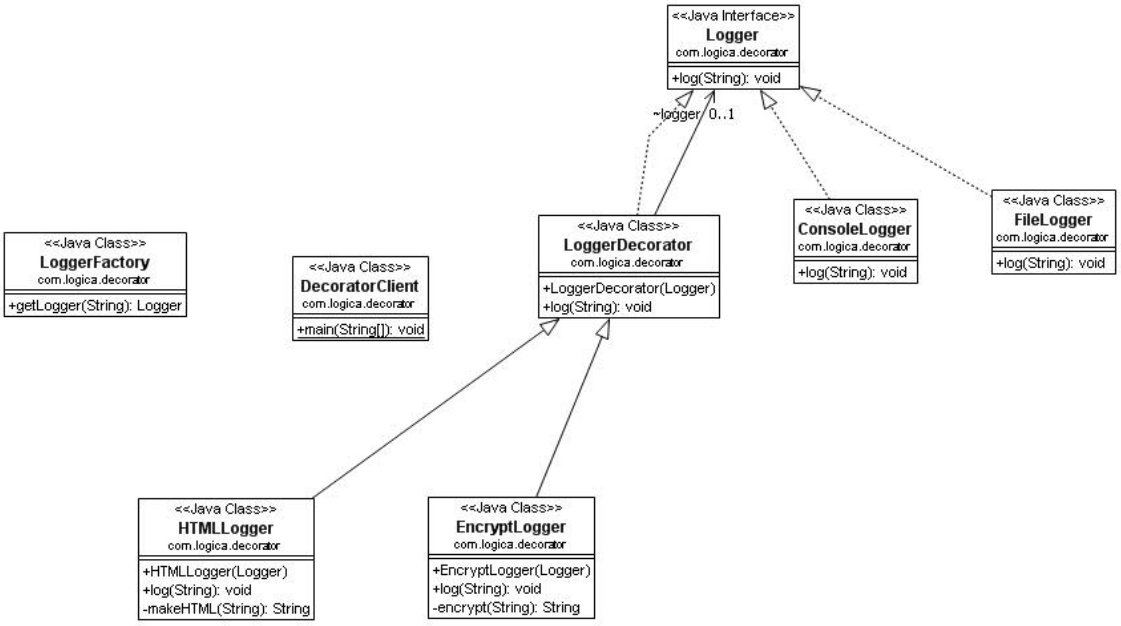
Benefits

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

Drawbacks/consequences

One thing to keep in mind when implementing the Decorator pattern is that you need to keep the component interface simple. You want to avoid making the component interface overly complex, because a complex interface will make it that much harder to get each decorator right.
Another potential drawback of the Decorator pattern is the performance overhead associated with a long chain of decorators.

Structure



Small example

```
package com.logica.decorator;
public interface Logger {
	public void log(String msg);
}


package com.logica.decorator;
public class LoggerDecorator implements Logger {
	Logger logger;

	public LoggerDecorator(Logger logger) {
		super();
		this.logger = logger;
	}

	public void log(String msg) {
		logger.log(msg);
	}
}
```

```
package com.logica.decorator;
public class ConsoleLogger implements Logger {
      public void log(String msg) {
            System.out.println("Detta ska skrivas till consolen! "+
                  msg);
      }
}


package com.logica.decorator;
public class EncryptLogger extends LoggerDecorator {
      public EncryptLogger(Logger logger) {
            super(logger);
      }

      public void log(String msg) {
            msg = encrypt(msg);
            logger.log(msg);
      }

      private String encrypt(String msg) {
            msg = msg.substring(msg.length()-1) + msg.substring(0,
                  msg.length() -1);
            return msg;
      }
}


package com.logica.decorator;
public class HTMLLogger extends LoggerDecorator {
      public HTMLLogger(Logger logger) {
            super(logger);
      }

      public void log(String msg) {
            msg = makeHTML(msg);
            logger.log(msg);
      }

      private String makeHTML(String msg) {
            msg = "<html><body>" + "<b>" + msg + "</b>" +
                  "</body></html>";
            return msg;
      }
}


package com.logica.decorator;
public class LoggerFactory {
      public static final String TYPE_CONSOL_LOGGER = "console";
      public static final String TYPE_FILE_LOGGER = "file";

      public Logger getLogger(String type) {
            if(TYPE_CONSOL_LOGGER.equals(type)) {
                  return new ConsoleLogger();
            } else {
                  return new FileLogger();
            }
      }
}
```

```
package com.logica.decorator;
public class DecoratorClient {
      public static void main(String[] args) {
            LoggerFactory factory = new LoggerFactory();
            Logger logger =
                  factory.getLogger(LoggerFactory.TYPE_FILE_LOGGER);
            HTMLLogger htmlLogger = new HTMLLogger(logger);
            htmlLogger.log("A message to log");
            EncryptLogger encryptLogger = new EncryptLogger(logger);
            encryptLogger.log("A message to log");
      }

}
```

When DecoratorClient is executed the result is:

```
      c:>Detta ska skrivas till en fil! <html><body><b>A message to
      log</b></body></html>
      c:>Detta ska skrivas till en fil! gA message to lo
```

Facade Pattern

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

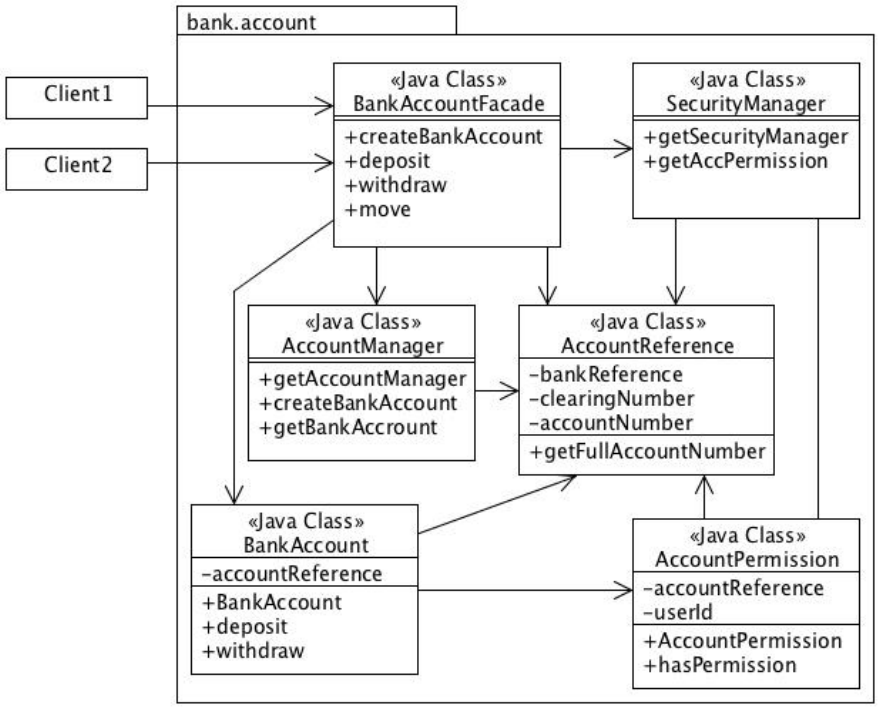
Where to use
The Facade can be used to make a software library easier to use and understand, since the Facade has convenient methods for common tasks. For the same reason, it can make code that uses the library more readable. The pattern can also be used to reduce dependencies of outside code on the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

Benefits
The main benefit with the Facade pattern is that we can combine very complex method calls and code blocks into a single method that performs a complex and recurring task. Besides making code easier to use and understand, it reduces code dependencies between libraries or packages, making programmers more apt to consideration before writing new code that exposes the inner workings of a library or a package. Also, since the Facade makes a weak coupling between the client code and other packages or libraries it allows us vary the internal components since the client does not call them directly.

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

Structure
The class diagram of the small example described in the next section is shown below to illustrate the structure of the Facade design pattern.

Small example



Consider the graphics above, where the Client classes makes calls to the BankAccountFacade class. The Facade class uses the classes AccountManager, SecurityManager, AccountReference and BankAccount to perform the appropriate actions. By using the Facade pattern we decouple the client code from the implementation and security details when performing tasks related to the bank account.

Flyweight Pattern

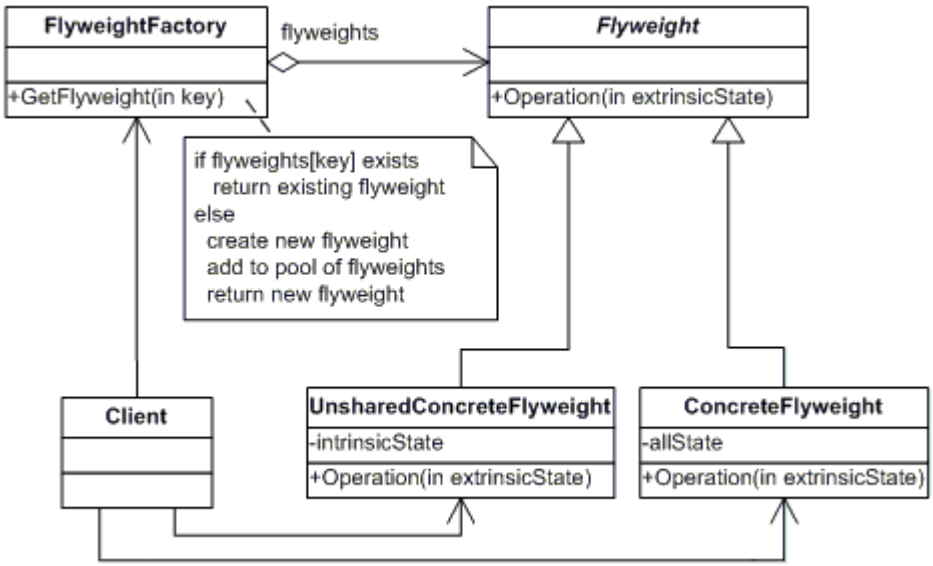
Definition
Th Flyweight pattern provides a mechanism by which you can avoid creating a large number of 'expensive' objects and instead reuse existing instances to represent new ones.

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

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

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

Structure



Small example

Trying to use objects at very low levels of granularity is nice, but the overhead may be prohibitive.

```
public class Gazillion1 {
    private static int num = 0;
    private int row, col;

    public Gazillion1( int maxPerRow ) {
        row = num / maxPerRow;
        col = num % maxPerRow;
        num++;
   }

   void report() {
       System.out.print( " " + row + col );
   }
}


public class NoFlyweight {
    public static final int ROWS = 6, COLS = 10;

    public static void main( String[] args ) {
        Gazillion1[][] matrix = new Gazillion1[ROWS][COLS];
        for (int i=0; i < ROWS; i++)
            for (int j=0; j < COLS; j++)
                matrix[i][j] = new Gazillion1( COLS );
          for (int i=0; i < ROWS; i++) {
              for (int j=0; j < COLS; j++)
                  matrix[i][j].report();
              System.out.println();
          }
     }
}
```

When NoFlyweight is executed we generate an excessive number of objects and the resulting output is:

```
c:> 00 01 02 03 04 05 06 07 08 09
c:>10 11 12 13 14 15 16 17 18 19
c:>20 21 22 23 24 25 26 27 28 29
c:>30 31 32 33 34 35 36 37 38 39
c:>40 41 42 43 44 45 46 47 48 49
c:>50 51 52 53 54 55 56 57 58 59
```

Introducing Flyweight means that we remove the non-shareable state from the class, and making the client supply it when needed. This places more responsibility on the client, but, considerably fewer instances of the Flyweight class are now created. Sharing of these instances is facilitated by introducing a Factory class that maintains a pool of existing Flyweights with a reusable state. In the following example, the "row" state is considered shareable (within each row anyways), and the "col" state has been externalized (it is supplied by the client when report() is called).

```
class Gazillion2 {
    private int row;

    public Gazillion2( int theRow ) {
        row = theRow;
      System.out.println( "Actual ctor: " + row );
    }

    void report( int theCol ) {
        System.out.print( " " + row + theCol );
  }
}


class FlyweightFactory {
    private Gazillion2[] pool;

    public FlyweightFactory( int maxRows ) {
        pool = new Gazillion2[maxRows];
    }

    public Gazillion2 getFlyweight( int theRow ) {
          if (pool[theRow] == null)
              pool[theRow] = new Gazillion2( theRow );
          return pool[theRow];
    }
}
```

```
public class Flyweight {
    public static final int ROWS = 6, COLS = 10;

    public static void main( String[] args ) {
        FlyweightFactory theFactory = new FlyweightFactory( ROWS );
        for (int i=0; i < ROWS; i++) {
            for (int j=0; j < COLS; j++)
                theFactory.getFlyweight( i ).report( j );
            System.out.println();
        }
    }
}
```

When Flyweight is executed we generate, from a client perspective the same (excessive) amount of instances but in reality we only generate one instance per row and reuse that instance several times. This is indicated in the output below with the text “Actual instance no. x”:

```
c:>Actual instance no. 0
c:>00 01 02 03 04 05 06 07 08 09
c:>Actual instance no. 1
c:>10 11 12 13 14 15 16 17 18 19
c:>Actual instance no. 2
c:>20 21 22 23 24 25 26 27 28 29
c:>Actual instance no. 3
c:>30 31 32 33 34 35 36 37 38 39
c:>Actual instance no. 4
c:>40 41 42 43 44 45 46 47 48 49
c:>Actual instance no. 5
c:>50 51 52 53 54 55 56 57 58 59
```

Usage example

The example used in the original GoF book for the flyweight pattern are the data structures for graphical representation of characters in a word processor. It would be nice to have, for each character in a document, a glyph object containing its font outline, font metrics, and other formatting data, but it would amount to hundreds or thousands of bytes for each character. Instead, for every character there might be a reference to a flyweight glyph object shared by every instance of the same character in the document; only the position of each character (in the document and/or the page) would need to be stored externally.

Another typical example of usage is the one of folders. The folder with name

of each of the company employee on it, so, the attributes of class Folder are: “Selected”, “Not Selected” and the third one is “employeeName”. With this methodology, we will have to create 2000 folder class instances for each of the employees. This can be costly, so we can create just two class instances with attributes “selected” and “not selected” and set the employee's name by a method like:

```
setNameOnFolder(String name);
```

This way, the instances of class folder will be shared and you will not have to create multiple instances for each employee.

Proxy Pattern

Definition
A Proxy is a structural pattern that provides a stand-in for another object in order to control access to it.

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

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

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

Structure

Small example

```
package cachedLogging;
public interface ICachedLogging {
      public void logRequest(String logString);
}

package cachedLogging;
public class CachedLogger implements ICachedLogging {
      public void logRequest(String logString) {
            System.out.println("CachedLogger logging to some
                  expensive resource: " + logString + "\n");
      }
}


package cachedLogging;
import java.util.ArrayList;
import java.util.List;
public class CachedLoggingProxy implements ICachedLogging {
      List<String> cachedLogEntries = new ArrayList<String>();
      CachedLogger cachedLogger = new CachedLogger();

      public void logRequest(String logString) {
            addLogRequest(logString);
      }

      private void addLogRequest(String logString) {
            cachedLogEntries.add(logString);
            if(cachedLogEntries.size() >= 4)
                  performLogging();
      }

      private void performLogging() {
            StringBuffer accumulatedLogString = new StringBuffer();
            for (String logString : cachedLogEntries) {
                  accumulatedLogString.append("\n"+logString);
                  System.out.println("CachedLoggingProxy: adding
                        logString \"" +logString + "\" to cached log
                        entries.");
            }
            System.out.println("CachedLoggingProxy: sends accumulated
                        logstring to CachedLogger.");
            cachedLogger.logRequest(accumulatedLogString.toString());
            cachedLogEntries.clear();
            System.out.println("CachedLoggingProxy: cachedLogEntries
                        cleared.");
      }
}
```

```
package client;
import cachedLogging.*;
public class Client {
      public static void main(String[] args) {
            ICachedLogging logger = new CachedLoggingProxy();
            for (int i = 1; i < 5; i++) {
                  logger.logRequest("logString "+i);
            }
      }
}
```

When Client is executed the result is:

c:>CachedLoggingProxy: adding logString "logString 1" to cached log entries.
c:>CachedLoggingProxy: adding logString "logString 2" to cached log entries.
c:>CachedLoggingProxy: adding logString "logString 3" to cached log entries.
c:>CachedLoggingProxy: adding logString "logString 4" to cached log entries.
c:>CachedLoggingProxy: sends accumulated logstring to CachedLogger.
c:>CachedLogger logging to some expensive resource:
c:>logString 1
c:>logString 2
c:>logString 3
c:>logString 4
c:>
c:>CachedLoggingProxy: cachedLogEntries cleared.

Behavioral Design Patterns

"In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication."[4]

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

4 http://en.wikipedia.org/wiki/Behavioral_patterns

Chain-of-responsibility Pattern

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

Where to use

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

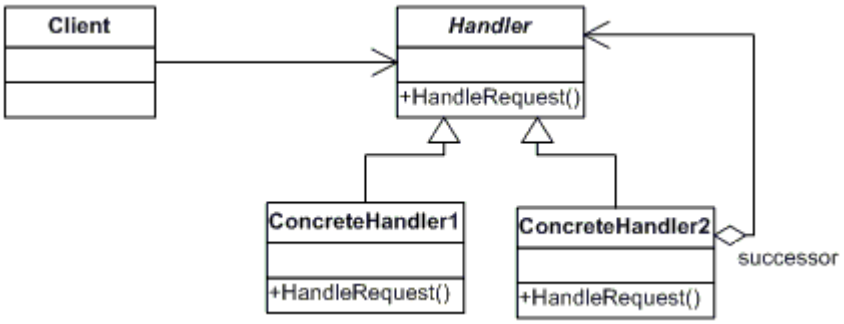
Benefits

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

Drawbacks/consequences

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

Structure



Small example

In the following source code example the ChainDemo is responsible for creating a chain of Handler instances, and pass 3 requests to them. The Handlers will randomly decide if and when any of them will handle the request.

```
public class Handler {
      private static java.util.Random s_rn = new java.util.Random();
      private static int s_next = 1;
      private int m_id = s_next++;
      private Handler m_next;

      public void add(Handler next) {
            if (m_next == null)
                  m_next = next;
            else
                  m_next.add(next);
      }

      public void wrap_around(Handler root) {
            if (m_next == null)
                  m_next = root;
            else
                  m_next.wrap_around(root);
      }

      public void handle(int num) {
            if (s_rn.nextInt(4) != 0) {
                  System.out.print(m_id + "-busy  ");
                  m_next.handle(num);
            }
            else
                  System.out.println(m_id + "-handled-" + num);
      }
}
```

```
public class ChainDemo {

    public static void main(String[] args) {
        Handler chain_root = new Handler();
        chain_root.add(new Handler());
        chain_root.add(new Handler());
        chain_root.add(new Handler());
        chain_root.wrap_around(chain_root);
        for (int i = 1; i <= 3; i++)
          chain_root.handle(i);
    }
}
```

When ChainDemo is executed the result is:

c:>1-busy 2-busy 3-handled-1
c:>1-busy 2-busy 3-busy 4-busy 1-busy 2-busy 3-busy 4-busy 1-busy 2-handled-2
c:>1-handled-3

Usage example

A good example is the creation of an error logging system. By creating different error-handlers, with differentiating responsibilities and connecting them in a chain. We can then pass, any potential errors that we want to log “down the chain” and letting the individual error-handlers handle the error.

Command Pattern

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

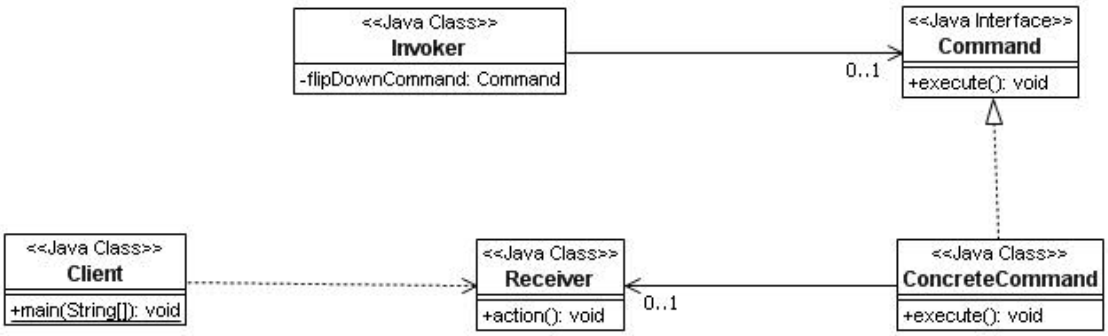
Where to use
• Where you want a action that can be represented in many ways, like drop-down menu, buttons and popup menu.
• To create undo/redo functionality.

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

Drawbacks/consequences
The main disadvantage of the Command pattern seems to be a proliferation of little classes that clutter up the program. However, even in the case where we have separate click events, we usually call little private methods to carry out the actual function. It turns out that these private methods are just about as long as our little classes, so there is frequently little difference

in complexity between building the Command classes and just writing more methods. The main difference is that the Command pattern produces little classes that are much more readable.

Structure



Small example

```
package com.logica.command;
/** Command interface */
public interface Command{
    void execute();
}
```

```
package com.logica.command;
/** The Command for turning on the light */
public class TurnOnLightCommand implements Command{
      private Light theLight;

      public TurnOnLightCommand(Light light){
            this.theLight=light;
      }

      public void execute(){
            theLight.turnOn();
      }
}
```

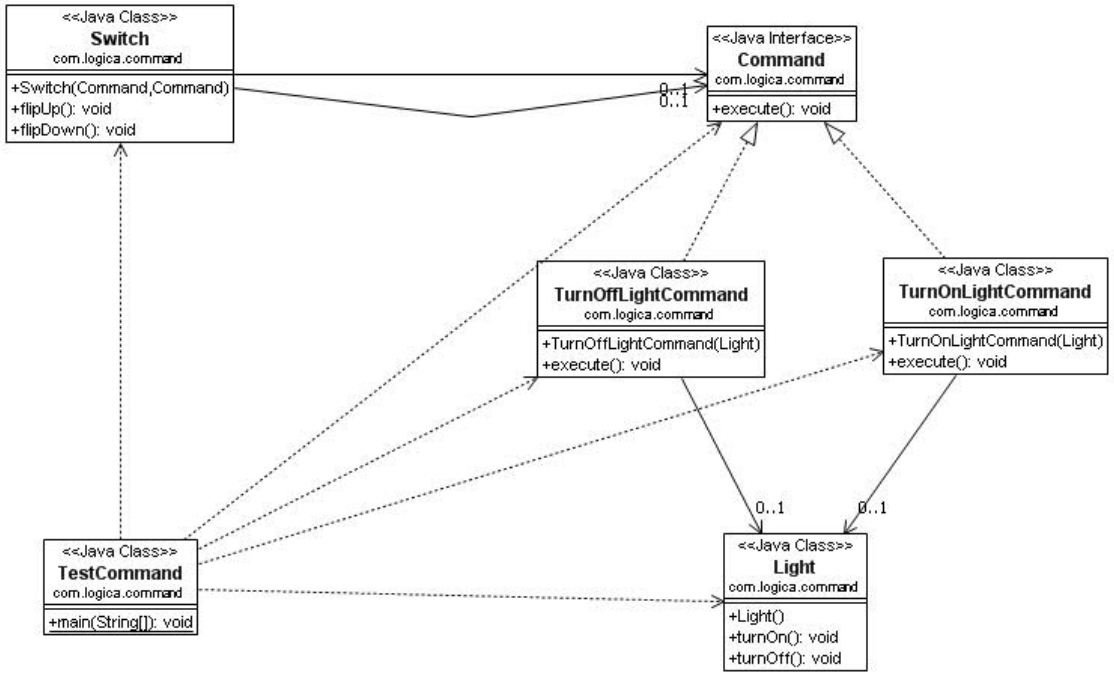
```
package com.logica.command;
/** The Command for turning off the light */
public class TurnOffLightCommand implements Command{
   private Light theLight;

   public TurnOffLightCommand(Light light){
        this.theLight=light;
       }

   public void execute(){
      theLight.turnOff();
   }
}
```

```
package com.logica.command;
/** Receiver class */
public class Light{
      public Light(){  }

      public void turnOn(){
            System.out.println("The light is on");
      }

      public void turnOff(){
            System.out.println("The light is off");
      }
}


package com.logica.command;
/** Invoker class*/
public class Switch {
      private Command flipUpCommand;
      private Command flipDownCommand;

      public Switch(Command flipUpCmd,Command flipDownCmd){
            this.flipUpCommand=flipUpCmd;
            this.flipDownCommand=flipDownCmd;
      }

      public void flipUp(){
            flipUpCommand.execute();
      }

      public void flipDown(){
            flipDownCommand.execute();
      }
}


package com.logica.command;
/** Test class */
public class TestCommand{

      public static void main(String[] args){
            Light l = new Light();
            Command switchUp = new TurnOnLightCommand(l);
            Command switchDown = new TurnOffLightCommand(l);
            Switch s = new Switch(switchUp,switchDown);
            s.flipUp();
            s.flipDown();
      }
}
```

When executing TestCommand the result is:

```
c:>The light is on
c:>The light is off
```

The following class-diagram illustrates the structure for the small example.



Iterator Pattern

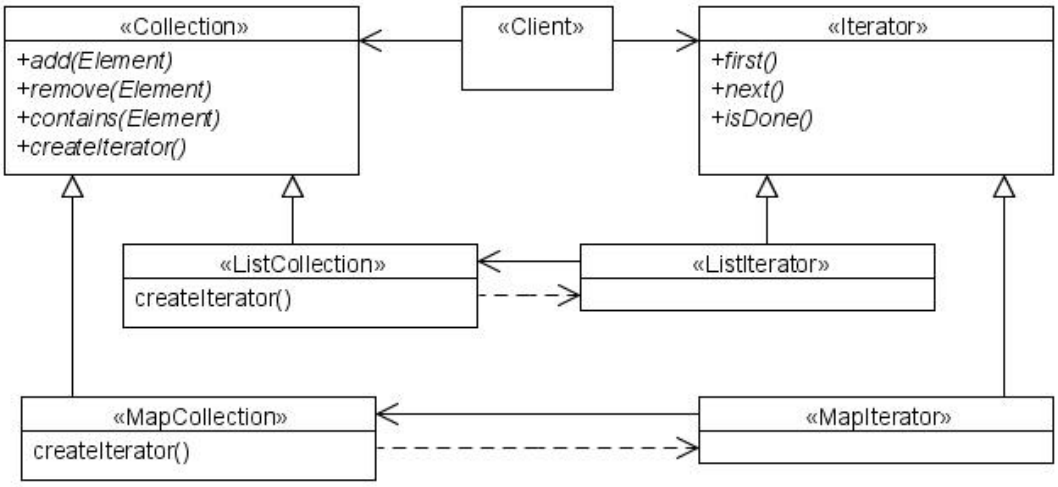
Definition
The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

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

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

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

Structure



Small example

This example shows how you can write your own iterator.

```
import java.util.*;
public class BitSetIterator implements Iterator<Boolean> {
      private final BitSet bitset;
      private int index;

      public BitSetIterator(BitSet bitset) {
            this.bitset = bitset;
      }

      public boolean hasNext() {
            return index < bitset.length();
      }

      public Boolean next() {
            if (index >= bitset.length()) {
                  throw new NoSuchElementException();
            }
            boolean b = bitset.get(index++);
            return new Boolean(b);
      }

      public void remove() {
            throw new UnsupportedOperationException();
      }
}
```

```
public class TestClientBitSet {
	public static void main(String[] args) {
		// create BitSet and set some bits
		BitSet bitset = new BitSet();
		bitset.set(1);
		bitset.set(19);
		bitset.set(20);
		bitset.set(47);
		BitSetIterator iter = new BitSetIterator(bitset);
		while (iter.hasNext()) {
			Boolean b = iter.next();
			String tf = (b.booleanValue() ? "T" : "F");
			System.out.print(tf);
		}
		System.out.println();
	}
}
```

When TestClientBitSet is executed the result is:

c:>FTFFFFFFFFFFFFFFFFFFTTFFFFFFFFFFFFFFFFFFFFFFFFFT

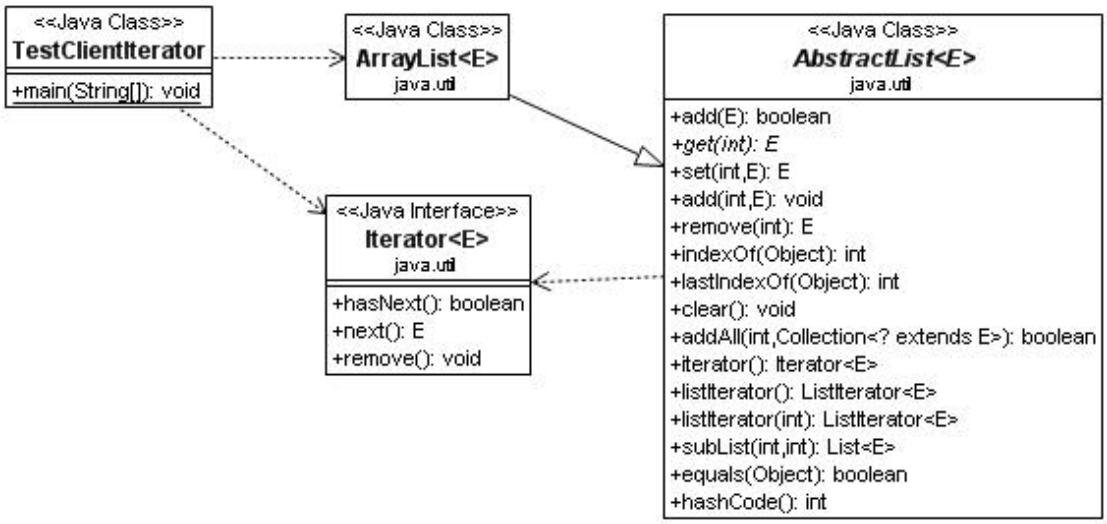
Usage example
This example shows how the Iterator of the ArrayList is used.

```
import java.util.*;

public class TestClientIterator  {

	public static void main(String[] args) {
		ArrayList<Object> al = new ArrayList<Object>();
		al.add(new Integer(42));
		al.add(new String("test"));
		al.add(new Double("-12.34"));
		for(Iterator<Object> iter=al.iterator();
			iter.hasNext();)
			System.out.println( iter.next() );
		// JEE5 syntax
		for(Object o:al)
			System.out.println( o );
	}
}
```

When run the result is:

c:>42
c:>test
c:>-12.34
c:>42
c:>test
c:>-12.34

In the following class-diagram we illustrate the structure from the usage example.



Mediator Pattern

Definition

With the mediator pattern communication between objects is encapsulated with a mediator object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This results in a more cohesive implementation of the logic and decreased coupling between the other objects.

Where to use

The Mediator pattern can be used when a set of objects communicate in well-specified but complex ways and the resulting interdependencies are unstructured and hard to grasp. If it is difficult to reuse an object because it refers to and communicates with many other objects this pattern is a good solution. Communicating objects' identities will be protected when using the Mediator pattern which is good when security is important. Also, if you like to customize some behavior which is spread out between several classes without a lot of subclassing this pattern should be applied. The pattern is used in many modern systems that reflect a send/receive protocol, such as list servers and chat rooms. Another area of use is graphical user interfaces, where the mediator can encapsulate a collective behavior to control and coordinate the interactions of a group of GUI widgets. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator which reduces the number of interconnections.

Benefits
The Mediator pattern has five benefits and drawbacks:

1. Limited subclassing, since a mediator localizes behavior that otherwise would be distributed among several objects. Changing some behavior requires us to subclass only the mediator.

2. Colleagues become decoupled which allows us to vary and reuse colleague and mediator classes independently.

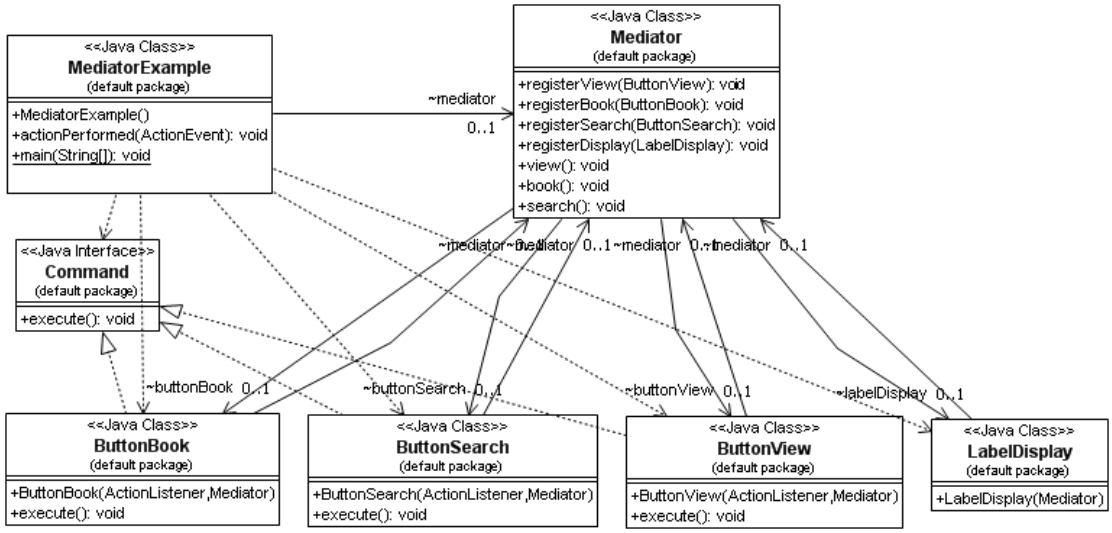
3. A mediator simplifies object protocols since it replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many interactions are easier to understand, maintain, and extend.

4. The mediator abstracts object cooperation. The mediator lets you focus on how objects interact apart from their individual behaviors which can help clarify how objects interact in a system.

5. Centralized control is achieved by trading complexity of interaction for complexity in the mediator. A mediator encapsulates protocols and can become more complex than any individual colleague. This can make the mediator itself a very complex and large piece of code that is hard to maintain.

Drawbacks/consequences
Besides the benefits and drawbacks described above, one important drawback is that the Mediator pattern can have a performance impact on a system. Since all communication must go through the mediator, it can become a bottleneck.

Structure



This is the class diagram of the small example that follows below.

Small example

Consider a dialog where there are a display label and three buttons: view, book and search. When clicking the view button it should be disabled and its two colleague buttons should be enabled. Also, the display label should change to reflect which button was pressed. We create four classes ButtonView, ButtonBook, ButtonSearch and LabelDisplay. Then we let the button classes implement an interface Command which allows us to execute the actions of the buttons from a common interface. All of these GUI classes are unaware of each other; they should only refer to the mediator. We now create the Mediator class which holds references to all the GUI objects to control and coordinate the interactions of these objects. Finally, we create the MediatorExample class to display the components in a frame. The class implements the ActionListener interface to which all the buttons should register. The complete source code of this example is shown below:

```
import java.awt.event.ActionListener;
import javax.swing.JButton;
public class ButtonView extends JButton implements Command {
      Mediator mediator;

      public ButtonView(ActionListener listener, Mediator mediator){
            super("View");
            addActionListener(listener);
            this.mediator = mediator;
            mediator.registerView(this);
      }

      public void execute() {
            mediator.view();
      }
}


import java.awt.event.ActionListener;
import javax.swing.JButton;
public class ButtonSearch extends JButton implements Command {
      Mediator mediator;

      ButtonSearch(ActionListener listener, Mediator mediator{
            super("Search");
            addActionListener(listener);
            this.mediator = mediator;
            mediator.registerSearch(this);
      }

      public void execute() {
            mediator.search();
      }
}


public interface Command {
      public void execute();
}
```

```
public class Mediator {
	ButtonView buttonView;
	ButtonBook buttonBook;
	ButtonSearch buttonSearch;
	LabelDisplay labelDisplay;

	public void registerView(ButtonView buttonView) {
		this.buttonView = buttonView;
	}

	public void registerBook(ButtonBook buttonBook) {
		this.buttonBook = buttonBook;
	}

	public void registerSearch(ButtonSearch buttonSearch) {
		this.buttonSearch = buttonSearch;
	}

	public void registerDisplay(LabelDisplay labelDisplay) {
		this.labelDisplay = labelDisplay;
	}

	public void view() {
		buttonView.setEnabled(false);
		buttonBook.setEnabled(true);
		buttonSearch.setEnabled(true);
		labelDisplay.setText("Viewing...");
	}

	public void book() {
		buttonBook.setEnabled(false);
		buttonView.setEnabled(true);
		buttonSearch.setEnabled(true);
		labelDisplay.setText("Booking...");
	}

	public void search() {
		buttonSearch.setEnabled(false);
		buttonBook.setEnabled(true);
		buttonView.setEnabled(true);
		labelDisplay.setText("Searching...");
	}
}
```

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;
class MediatorExample extends JFrame implements ActionListener {
      Mediator mediator = new Mediator();

      public MediatorExample() {
            JPanel p = new JPanel();
            p.add(new ButtonView(this, mediator));
            p.add(new ButtonBook(this, mediator));
            p.add(new ButtonSearch(this, mediator));
            getContentPane().add(new LabelDisplay(mediator),
                  BorderLayout.NORTH);
            getContentPane().add(p, BorderLayout.SOUTH);
            setTitle("Mediator Example");
            setSize(300, 200);
            addWindowListener(new WindowAdapter() {
                  public void windowClosing(WindowEvent e) {
                        System.exit(0);
                  }
            });
            setVisible(true);
      }

      public void actionPerformed(ActionEvent e) {
            if (e.getSource() instanceof Command) {
                  Command c = (Command)e.getSource();
                  c.execute();
            }
      }

      public static void main(String[] args) {
            new MediatorExample();
      }
}
```

When mediatorExample is executed the result is:

Memento Pattern

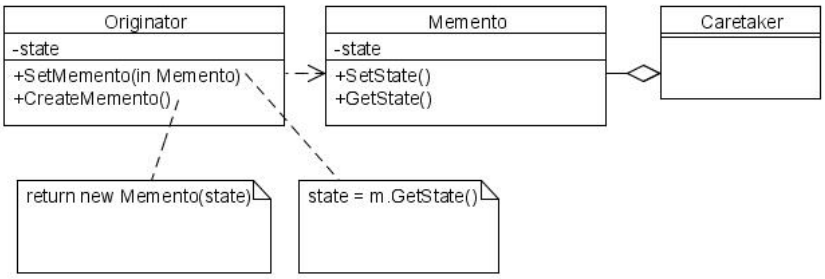
Definition
To record an object internal state without violating encapsulation and reclaim it later without knowledge of the original object. A memento is an object that stores a snapshot of the internal state of another object.

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

Benefits
Ability to restore an object to its previous state.

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

Structure



In the class-diagram above:

•An instance from the Originator class is the object that knows how to save itself. Uses memento to restore itself.

•An instance from the Caretaker class is the object that knows why and when the Originator needs to save and restore itself. Never operates or examines the contents of memento

•An instance of the Memento class is the lock box that is written and read by the Originator, and shepherded by the Caretaker.

The originator is any object that has an internal state, that we want to take a snapshot of. The caretaker is going to do something to the originators state, but wants to be able to later restore the originators state. The caretaker first asks the originator for a memento object, containing the snapshot. Then it performs the sequence of operations it was going to do. To roll back to the state before the operations, it returns the memento object to the originator.

Small example

```
public class Originator {
    private String state;
    /* lots of memory using private data that does not have to be
      saved. Instead we use a small memento object. */

    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }

    public Object saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }

    public void restoreFromMemento(Object m) {
        if (m instanceof Memento) {
            Memento memento = (Memento) m;
            state = memento.getSavedState();
            System.out.println("Originator: State after restoring
                  from Memento: " + state);
        }
    }
}


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

public class Caretaker {
    private List<Object> savedStates = new ArrayList<Object>();

    public void addMemento(Object m) {
        savedStates.add(m);
    }

    public Object getMemento(int index) {
        return savedStates.get(index);
    }
}


public class Memento {
    private String state;

    public Memento(String stateToSave) {
         state = stateToSave;
    }

    public String getSavedState() {
         return state;
    }
}
```

```
public class MementoExample {

    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();
        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State3");
        caretaker.addMemento(originator.saveToMemento());
        originator.set("State4");
        originator.restoreFromMemento(caretaker.getMemento(1));
     }
}
```

When MementoExample is executed the result is:

```
c:>Originator: Setting state to State1
c:>Originator: Setting state to State2
c:>Originator: Saving to Memento.
c:>Originator: Setting state to State3
c:>Originator: Saving to Memento.
c:>Originator: Setting state to State4
c:>Originator: State after restoring from Memento: State3
```

Usage example
Often used in database transactions and undo/redo situations.

Observer Pattern

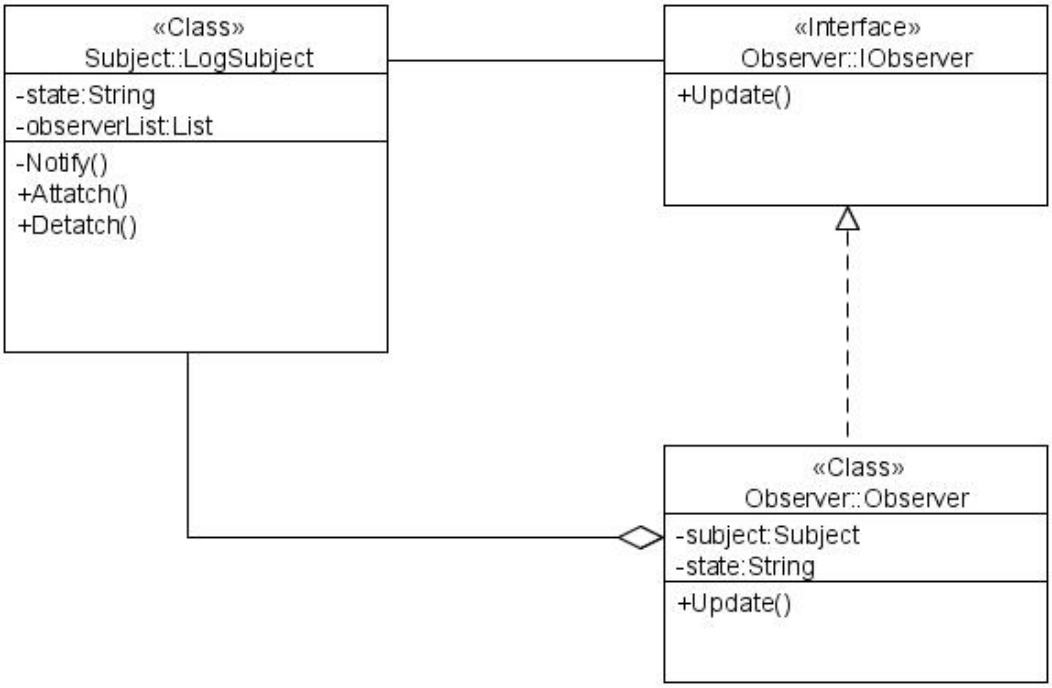
Definition
An observer is a structural pattern that enables publish/subscribe functionality. This is accomplished by an autonomous object, publisher that allows other objects to attach or detach their subscription as they like. The pattern does not impose any limit to the number of observers that can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

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

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

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

Structure



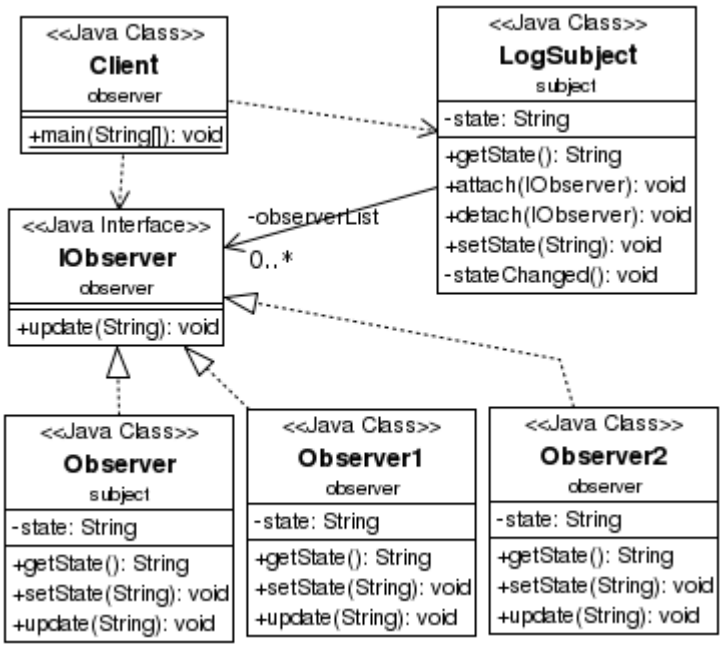
Small example

```
package observer;
public interface IObserver {
	void update(String state);
}


package subject;
import observer.IObserver;
public class Observer implements IObserver {
	private String state;

	public String getState() {
		return state;
	}

	public void setState(String state) {
		this.state = state;
	}

	public void update(String state) {
		setState(state);
		System.out.println("Observer has received update signal
			with new state: " + getState());
	}
}
```

```
package observer;
public class Observer1 implements IObserver {
	private String state;

	public String getState() {
		return state;
	}

	public void setState(String state) {
		this.state = state;
	}

	public void update(String state) {
		setState(state);
		System.out.println("Observer1 has received update signal
			with new state: " + getState());
	}
}


package observer;
public class Observer2 implements IObserver {

	private String state;

	public String getState() {
		return state;
	}

	public void setState(String state) {
		this.state = state;
	}

	public void update(String state) {
		setState(state);
		System.out.println("Observer2 has received update signal
			with new state: " + getState());
	}
}
```

```
package subject;
import java.util.*;
import observer.IObserver;

public class LogSubject {

      private List<IObserver> observerList = new
                                          ArrayList<IObserver>();
      private String state;

      public String getState() {
            return state;
      }

      public void attach(IObserver observer) {
            observerList.add(observer);
      }

      public void detach(IObserver observer) {
            observerList.remove(observer);
      }

      public void setState(String state) {
            this.state = state;
            stateChanged();
      }

      private void stateChanged() {
            for (IObserver item: observerList) {
                  item.update(getState());

            }

      }

}


package observer;
import subject.*;
public class Client {
      public static void main(String[] args) {
            LogSubject subject = new LogSubject();
            IObserver ob = new Observer();
            IObserver ob1 = new Observer1();
            IObserver ob2 = new Observer2();

            subject.attach(ob);
            subject.attach(ob1);
            subject.attach(ob2);
            subject.setState("state1");
            subject.setState("state2");
            subject.detach(ob1);
            subject.setState("state3");
      }
}
```

The following class diagram describes the realtions in the above small example.



When Client is executed the result is:

```
c:>Observer has received update signal with new state: state1
c:>Observer1 has received update signal with new state: state1
c:>Observer2 has received update signal with new state: state1
c:>Observer has received update signal with new state: state2
c:>Observer1 has received update signal with new state: state2
c:>Observer2 has received update signal with new state: state2
c:>Observer has received update signal with new state: state3
c:>Observer2 has received update signal with new state: state3
```

State Pattern

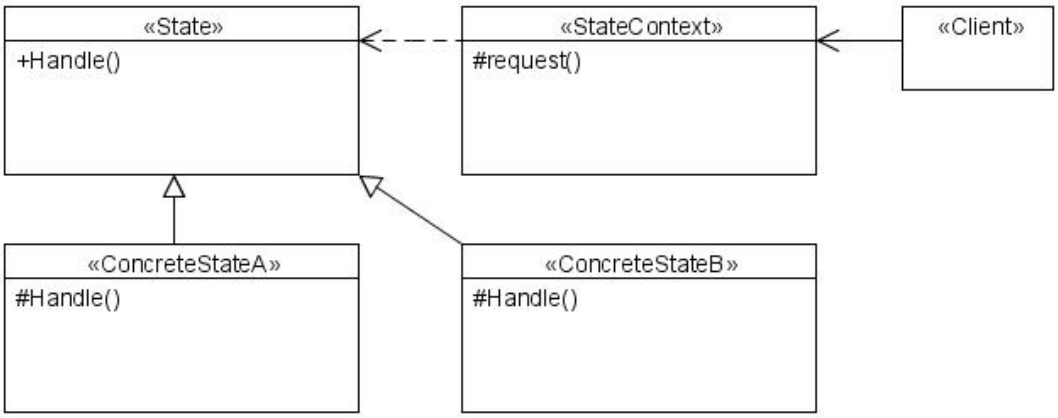
Definition
The State pattern allows an object to alter its behavior when its internal state changes. By using inheritance and letting subclasses represent different states and functionality we can switch during runtime. This is a clean way for an object to partially change its type at runtime.

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

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

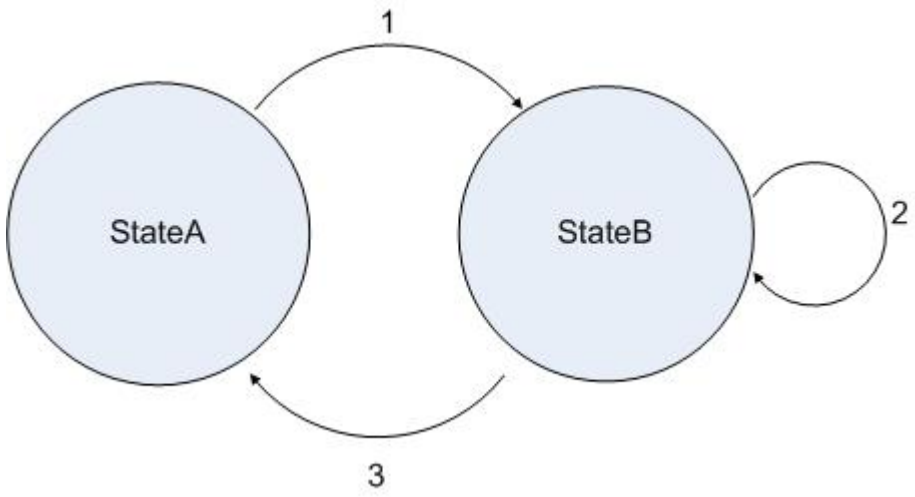
Drawbacks/consequences
•Generates a number of small class objects, but in the process, simplifies and clarifies the program.
•Eliminates the necessity for a set of long, look-alike conditional statements scattered throughout the code.

Structure



Small example

This example has two states. StateA writes name in lower-case and StateB in upper-case. We start in state A and then change to state B where we stay for two calls to writeName before changing back to state A.

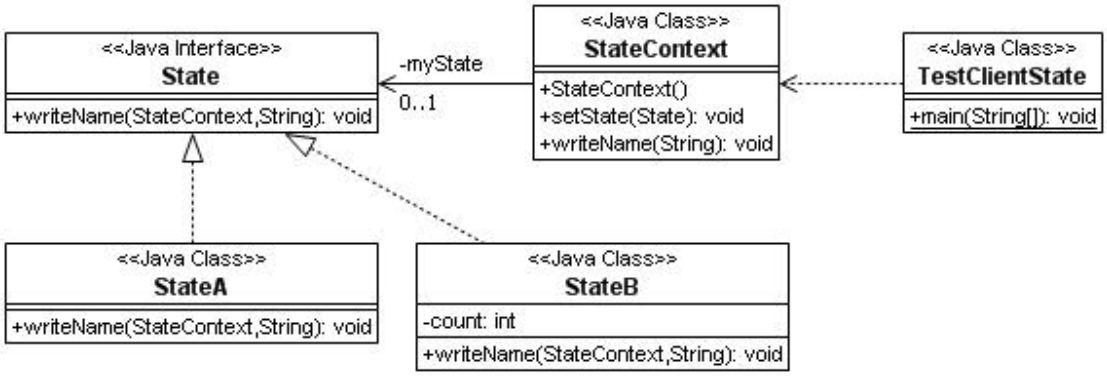


```
public interface State {
    public void writeName(StateContext stateContext, String name);
}


public class StateA implements State {
    public void writeName(StateContext stateContext, String name) {
            System.out.println(name.toLowerCase());
            stateContext.setState(new StateB());
    }
}


public class StateB implements State {
      private int count=0;

      public void writeName(StateContext stateContext, String name){
            System.out.println(name.toUpperCase());
            if(++count>1) {
                  stateContext.setState(new StateA());
            }
      }
}


public class StateContext {
      private State myState;

      public StateContext() {
            setState(new StateA());
      }

      public void setState(State stateName) {
            this.myState = stateName;
      }

      public void writeName(String name) {
            this.myState.writeName(this, name);
      }
}


public class TestClientState {
      public static void main(String[] args) {
            StateContext sc = new StateContext();
            sc.writeName("Monday");
            sc.writeName("Tuesday");
            sc.writeName("Wednesday");
            sc.writeName("Thursday");
            sc.writeName("Saturday");
            sc.writeName("Sunday");
      }
}
```

When TestClientState is executed the result is:

```
c:>monday
c:>TUESDAY
c:>WEDNESDAY
c:>thursday
c:>SATURDAY
c:>SUNDAY
```

Below is a class-diagram illustrating the structure of the small example.



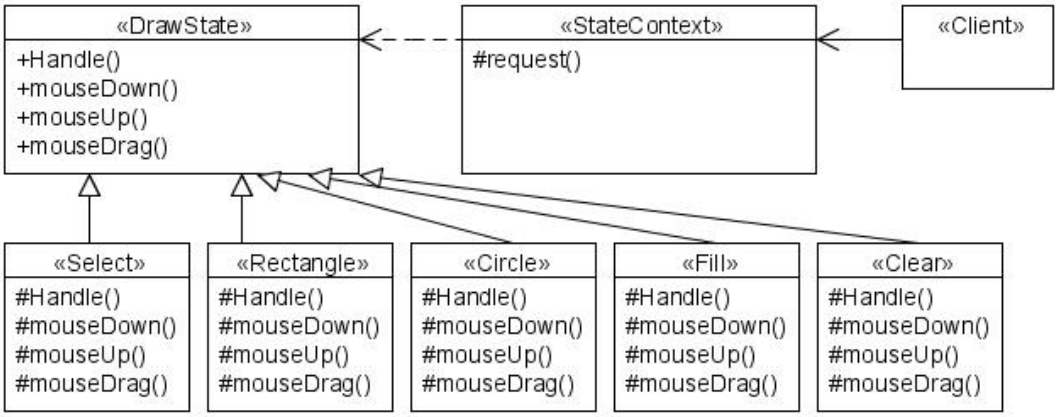
Usage example

Consider a drawing application with different tools like select, rectangle, circle, fill and clear. Each one of the tool behaves different when you click or drag the mouse across the screen. The state of the editor affects the behavior the program should exhibit. There are some mouse activities which needs to be handled.

```
public class DrawState {
      public void mouseDown(int x, int y) {}
      public void mouseUp(int x, int y) {}
      public void mouseDrag(int x, int y) {}
}
```

Each of the states would implement one or more of these methods differently.

```
public class Rectangle extends DrawState {
      //create a new Rectangle where mouse clicks
      public void mouseDown(int x, int y) {
            canvas.addDrawing(new visRectangle(x, y));
      }
}
```



The class-diagram above illustrates the structure of the usage example.

Strategy Pattern

Definition

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

Where to use

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

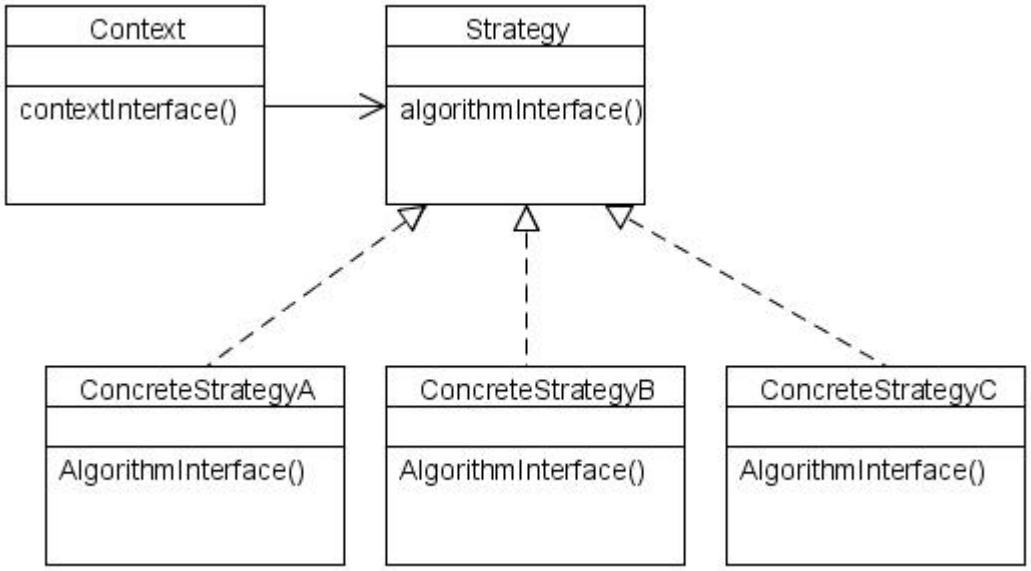
Benefits

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

Drawbacks/consequences

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

Structure

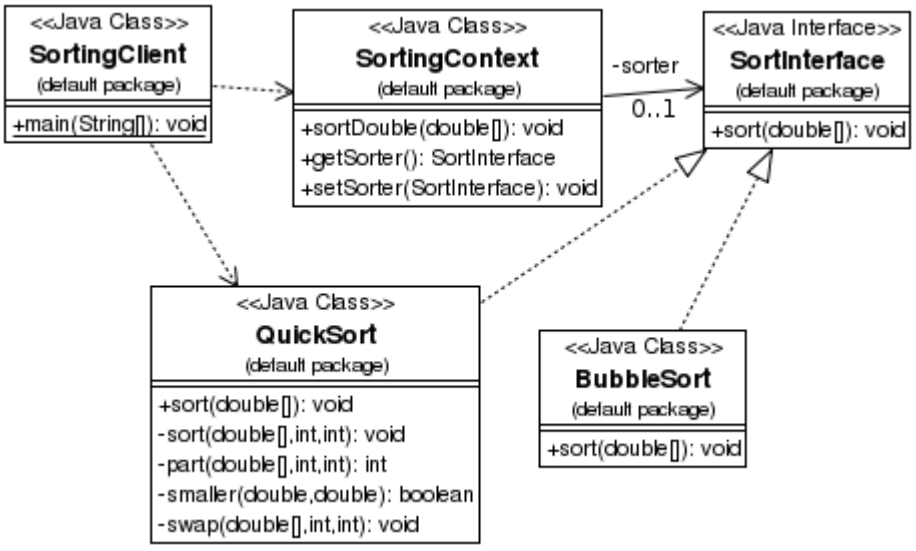


In the class-diagram above:
•The Strategy interface defines the behavior that is common to all the concrete implementations.
•The ConcreteStrategy encapsulates an implementation of a specific algorithm or behavior that is defined through the Strategy interface.
•The Context provides certain services that is defined by the Strategy interface and implemented by different ConcreteStrategy classes depending on behavior.

Small example
This example uses sorting algorithms. Two sorting algorithms (Bubble sort and Quick sort) are implemented and the client can select either of the algorithms. The SortInterface describes what the algorithms must be able to do, sort(). The classes QuickSort and BubbleSort both implements the SortInterface and each have their own algorithm for sorting. SortingContext maintains a reference to a Strategy object and forwards client requests to the strategy. SortingClient set the concrete strategy in the context and invokes the context to run the algorithm.

```
public interface SortInterface {
      public void sort(double[] list);
}

public class QuickSort implements SortInterface {
      public void sort(double[] u) {
            sort(u, 0, u.length - 1);
      }

      private void sort(double[] a, int left, int right) {
            if (right <= left) return;
            int i = part(a, left, right);
            sort(a, left, i-1);
            sort(a, i+1, right);
      }

      private int part(double[] a, int left, int right) {
            int i = left;
            int j = right;
            while (true) {
                  while (a[i]< a[right])
                        i++;
                  while (smaller(a[right], a[--j]))
                        if (j == left) break;
                  if (i >= j) break;
                  swap(a, i, j);
            }
            swap(a, i, right);
            return i;
      }

      private boolean smaller(double x, double y) {
            return (x < y);
      }

      private void swap(double[] a, int i, int j) {
            double swap = a[i]; a[i] = a[j]; a[j] = swap;
      }
}


public class BubbleSort implements SortInterface {
      public void sort(double[] list) {
            //Bubblesort algorithm here
      }
}
```

```
public class SortingContext {
      private SortInterface sorter = null;

      public void sortDouble(double[] list) {
            sorter.sort(list);
      }

      public SortInterface getSorter() {
            return sorter;
      }

      public void setSorter(SortInterface sorter) {
            this.sorter = sorter;
      }
}


public class SortingClient {
      public static void main(String[] args) {
            double[] list = {1,2.4,7.9,3.2,1.2,0.2,10.2,
                        22.5,19.6,14,12,16,17};
            SortingContext context = new SortingContext();
            context.setSorter(new QuickSort());
            context.sortDouble(list);
            for(int i =0; i< list.length; i++) {
                  System.out.println(list[i]);
            }
      }
}
```



When SortingClient is executed the result is:

```
c:>0.2
c:>1.0
c:>1.2
c:>2.4
c:>3.2
c:>7.9
c:>10.2
c:>12.0
c:>14.0
c:>16.0
c:>17.0
c:>19.6
c:>22.5
```

Usage example

The strategy pattern is used in the implementation of the LayoutManager in Java. The LayoutManager can be configured to work with different layout objects, such as FlowLayout, CardLayout, GridLayout, etc. These classes encapsulate the actual algorithms for laying out visual components that the LayoutManager uses to render the interface on the screen. By changing the Layout (algorithm) we can dynamically change how the interface is rendered.