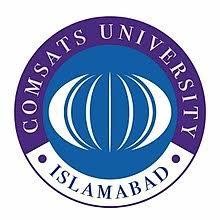
**Assignment # 01**

Prototype and Flyweight Design Patterns



**Instructor: Ma’am Mariyyam Gillani**

**Name: Mohammad Fahim**

**Reg: FA16-BSE-219**

**Prototype Design Pattern**

**Motivation**

Today’s programming is all about costs. Saving is a big issue when it comes to using computer resources, so programmers are doing their best to find ways of improving the performance When we talk about object creation we can find a better way to have new objects: cloning. To this idea one particular design pattern is related: rather than creation it uses cloning. If the cost of creating a new object is large and creation is resource intensive, we clone the object.

The Prototype design pattern is the one in question. It allows an object to create customized objects without knowing their class or any details of how to create them. Up to this point it sounds a lot like the Factory Method pattern, the difference being the fact that for the Factory the palette of prototypical objects never contains more than one object.

**Intent**

* specifying the kind of objects to create using a prototypical instance
* creating new objects by copying this prototype

**Structure**

****

**Components**

1. **Prototype**

Declares an interface for cloning itself.

1. **ConcretePrototype**

Implements an operation for cloning itself.

1. **Client**

Creates a new object by asking a prototype to clone itself.

**Applicability**

Use Prototype Pattern when a system should be independent of how its products are created, composed, and represented, and:

* Classes to be instantiated are specified at run-time
* Avoiding the creation of a factory hierarchy is needed
* It is more convenient to copy an existing instance than to create a new one.

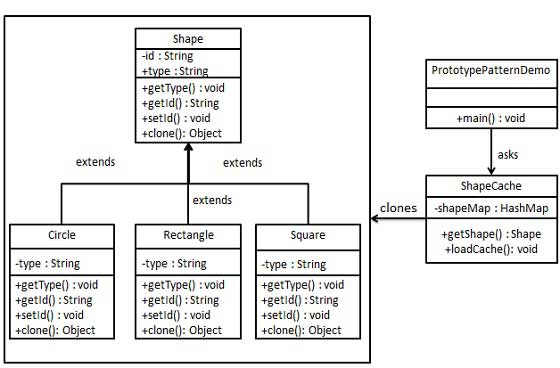
**Consequences**

* Prototype has many of the same consequences that Abstract Factory and Builder have: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification.
* Adding and removing products at run-time. Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client.
* Specifying new objects by varying values. Highly dynamic systems let you define new behavior through object composition—by specifying values for an object's variables, for example—and not by defining new classes. You effectively define new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects.
* Specifying new objects by varying structure. Many applications build objects from parts and subparts. Editors for circuit design, for example, build circuits out of subcircuits. For convenience, such applications often let you instantiate complex, user-defined.
* Reduced subclassing. Factory Method often produces a hierarchy of Creator classes. Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all.
* Configuring an application with classes dynamically. Some run-time environments let you load classes into an application dynamically.
* The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult. Implementing Clone can be difficult when their internals include objects that don't support copying or have circular references.

**Implementation**

We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested.

*PrototypPatternDemo*, our demo class will use *ShapeCache* class to get a *Shape* object.



**Step 1**

Create an abstract class implementing *Clonable* interface.

***Shape.java***

public abstract class Shape implements Cloneable {

private String id;

protected String type;

abstract void draw();

public String getType(){

return type;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public Object clone() {

Object clone = null;

try {

clone = super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return clone;

}

}

**Step 2**

Create concrete classes extending the above class.

***Rectangle.java***

public class Rectangle extends Shape {

public Rectangle(){

type = "Rectangle";

}

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

***Square.java***

public class Square extends Shape {

public Square(){

type = "Square";

}

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

***Circle.java***

public class Circle extends Shape {

public Circle(){

type = "Circle";

}

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

**Step 3**

Create a class to get concrete classes from database and store them in a *Hashtable*.

***ShapeCache.java***

import java.util.Hashtable;

public class ShapeCache {

private static Hashtable<String, Shape> shapeMap = new Hashtable<String, Shape>();

public static Shape getShape(String shapeId) {

Shape cachedShape = shapeMap.get(shapeId);

return (Shape) cachedShape.clone();

}

// for each shape run database query and create shape

// shapeMap.put(shapeKey, shape);

// for example, we are adding three shapes

public static void loadCache() {

Circle circle = new Circle();

circle.setId("1");

shapeMap.put(circle.getId(),circle);

Square square = new Square();

square.setId("2");

shapeMap.put(square.getId(),square);

Rectangle rectangle = new Rectangle();

rectangle.setId("3");

shapeMap.put(rectangle.getId(), rectangle);

}

}

**Step 4**

*PrototypePatternDemo* uses *ShapeCache* class to get clones of shapes stored in a *Hashtable*.

***PrototypePatternDemo.java***

public class PrototypePatternDemo {

public static void main(String[] args) {

ShapeCache.loadCache();

Shape clonedShape = (Shape) ShapeCache.getShape("1");

System.out.println("Shape : " + clonedShape.getType());

Shape clonedShape2 = (Shape) ShapeCache.getShape("2");

System.out.println("Shape : " + clonedShape2.getType());

Shape clonedShape3 = (Shape) ShapeCache.getShape("3");

System.out.println("Shape : " + clonedShape3.getType());

}

}

**Step 5**

Verify the output.

Shape : Circle

Shape : Square

Shape : Rectangle

**Flyweight Design Pattern**

**Motivation**

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

**Intent**

The intent of this pattern is to use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

**Structure**



**Components**

1. **Flyweight**

Declare an interface through which flyweights can receive and act on extrinsic state.

1. **ConcreteFlyweight**

Implements the Flyweight interface and adds storage for intrinsic state, if any.

1. **UnsharedConcreteFlyweight**

Not all flyweight subclasses need to be shared.

1. **Flyweight Factory**

* Creates and manages flyweight objects.
* Ensures that flyweights are shared properly.

1. **Client**

* Maintains a reference to flyweights.
* Computes or stores the extrinsic state of flyweight.

**Applicability**

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

**Consequences**

Storage savings are a function of several factors:

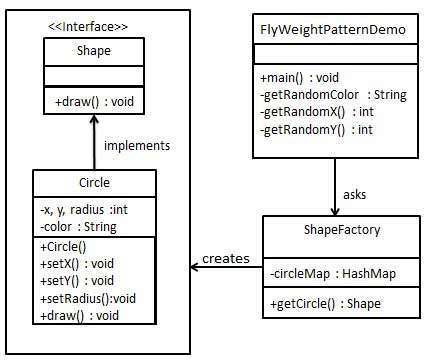
* The reduction in the total number of instances that comes from sharing.
* The amount of intrinsic state per object.
* Whether extrinsic state is computed or stored.
* Flyweight may introduce run-time cost.

**Implementation**

We are going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

*ShapeFactory* has a *HashMap* of *Circle* having key as color of the *Circle* object. Whenever a request comes to create a circle of particular color to *ShapeFactory*, it checks the circle object in its *HashMap*, if object of *Circle* found, that object is returned otherwise a new object is created, stored in hashmap for future use, and returned to client.

*FlyWeightPatternDemo*, our demo class, will use *ShapeFactory* to get a *Shape* object. It will pass information (*red / green / blue/ black / white*) to *ShapeFactory* to get the circle of desired color it needs.



**Step 1**

Create an interface.

***Shape.java***

public interface Shape {

void draw();

}

**Step 2**

Create concrete class implementing the same interface.

***Circle.java***

public class Circle implements Shape {

private String color;

private int x;

private int y;

private int radius;

public Circle(String color){

this.color = color;

}

public void setX(int x) {

this.x = x;

}

public void setY(int y) {

this.y = y;

}

public void setRadius(int radius) {

this.radius = radius;

}

@Override

public void draw() {

System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);

}

}

**Step 3**

Create a factory to generate object of concrete class based on given information.

***ShapeFactory.java***

import java.util.HashMap;

public class ShapeFactory {

// Uncomment the compiler directive line and

// javac \*.java will compile properly.

// @SuppressWarnings("unchecked")

private static final HashMap circleMap = new HashMap();

public static Shape getCircle(String color) {

Circle circle = (Circle)circleMap.get(color);

if(circle == null) {

circle = new Circle(color);

circleMap.put(color, circle);

System.out.println("Creating circle of color : " + color);

}

return circle;

}

}

**Step 4**

Use the factory to get object of concrete class by passing an information such as color.

***FlyweightPatternDemo.java***

public class FlyweightPatternDemo {

private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };

public static void main(String[] args) {

for(int i=0; i < 20; ++i) {

Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());

circle.setX(getRandomX());

circle.setY(getRandomY());

circle.setRadius(100);

circle.draw();

}

}

private static String getRandomColor() {

return colors[(int)(Math.random()\*colors.length)];

}

private static int getRandomX() {

return (int)(Math.random()\*100 );

}

private static int getRandomY() {

return (int)(Math.random()\*100);

}

}

**Step 5**

Verify the output.

Creating circle of color : Black

Circle: Draw() [Color : Black, x : 36, y :71, radius :100

Creating circle of color : Green

Circle: Draw() [Color : Green, x : 27, y :27, radius :100

Creating circle of color : White

Circle: Draw() [Color : White, x : 64, y :10, radius :100

Creating circle of color : Red

Circle: Draw() [Color : Red, x : 15, y :44, radius :100

Circle: Draw() [Color : Green, x : 19, y :10, radius :100

Circle: Draw() [Color : Green, x : 94, y :32, radius :100

Circle: Draw() [Color : White, x : 69, y :98, radius :100

Creating circle of color : Blue

Circle: Draw() [Color : Blue, x : 13, y :4, radius :100

Circle: Draw() [Color : Green, x : 21, y :21, radius :100

Circle: Draw() [Color : Blue, x : 55, y :86, radius :100

Circle: Draw() [Color : White, x : 90, y :70, radius :100

Circle: Draw() [Color : Green, x : 78, y :3, radius :100

Circle: Draw() [Color : Green, x : 64, y :89, radius :100

Circle: Draw() [Color : Blue, x : 3, y :91, radius :100

Circle: Draw() [Color : Blue, x : 62, y :82, radius :100

Circle: Draw() [Color : Green, x : 97, y :61, radius :100

Circle: Draw() [Color : Green, x : 86, y :12, radius :100

Circle: Draw() [Color : Green, x : 38, y :93, radius :100

Circle: Draw() [Color : Red, x : 76, y :82, radius :100

Circle: Draw() [Color : Blue, x : 95, y :82, radius :100

**Iterator Design Pattern:**

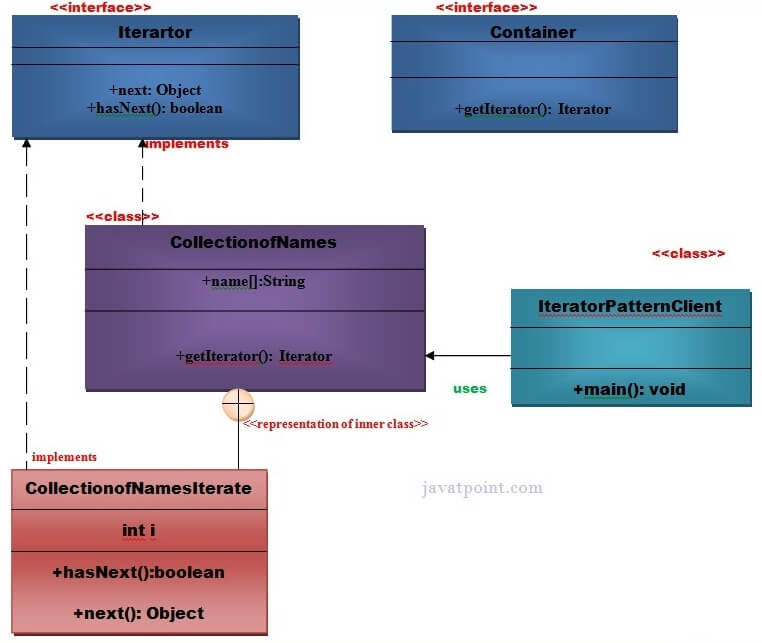
1. **Intent:**

The intent of this design pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. In other words, an iterator allows a client to access a contents of a certain data structure without knowing about the internal representation of the contents.

1. **Applicability:**

The iterator pattern allow us to:

1. Access contents of a collection without exposing its internal structure.
2. Support multiple simultaneous traversals of a collection.
3. Provide a uniform interface for traversing different collection.
4. **Structure:**

****

1. **Components:**

The Design Components in this pattern are:

1. **Iterator**: Defines an interface for accessing the elements and traversing the elements
2. **ConcreteIterator**: Implements the Iterator interface and keeps track of the current position in the traversal of the aggregate
3. **Aggregate**: Defines an interface for creating an Iterator object
4. **ConcreteAggregate**: Implements the Iterator creation interface to return an instance of the proper ConcreteIterator
5. **Consequences:**

There are three major consequences.

1. Iterator Pattern supports variations in the traversal of an aggregate. Every time you want to change the traversal algorithm, you just replace the iterator instance with a different one.
2. Iterators simplify the Aggregate interface.
3. More than one traversal can be pending on an aggregate. An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.
4. **Java Implementation:**

**Step 1:**

Create a **Iterartor** interface.

1. **public** **interface** Iterator {
2. **public** **boolean** hasNext();
3. **public** Object next();
4. }

**Step 2:**

Create a **Container** interface.

1. **public** **interface** Container {
2. **public** Iterator getIterator();
3. }// End of the Iterator interface.

**Step 3:**

Create a **CollectionofNames** class that will implement **Container** interface.

*File: CollectionofNames.java*

1. **public** **class** CollectionofNames **implements** Container {
2. **public** String name[]={"Mohammad Fahim", "Elyas”, "Qazafie", "Ramin", " Zekrullah"};
4. @Override
5. **public** Iterator getIterator() {
6. **return** **new** CollectionofNamesIterate() ;
7. }
8. **private** **class** CollectionofNamesIterate **implements** Iterator{
9. **int** i;
10. @Override
11. **public** **boolean** hasNext() {
12. **if** (i<name.length){
13. **return** **true**;
14. }
15. **return** **false**;
16. }
17. @Override
18. **public** Object next() {
19. **if**(**this**.hasNext()){
20. **return** name[i++];
21. }
22. **return** **null**;
23. }
24. }
25. }
26. }

**Step 4:**

Create a **IteratorPatternDemo** class.

*File: IteratorPatternDemo.java*

1. **public** **class** IteratorPatternDemo {
2. **public** **static** **void** main(String[] args) {
3. CollectionofNames cmpnyRepository = **new** CollectionofNames();
5. **for**(Iterator iter = cmpnyRepository.getIterator(); iter.hasNext();){
6. String name = (String)iter.next();
7. System.out.println("Name : " + name);
8. }
9. }
10. }

#### Output

1. Name : Mohammad Fahim
2. Name : Elyas
3. Name : Qazafie
4. Name : Ramin
5. Name : Zekrullah

**Mediator Design Pattern:**

1. **Intent:**

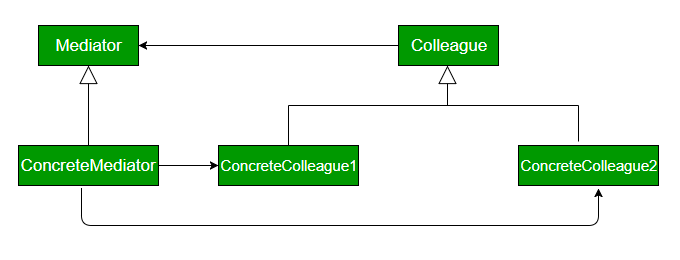
* Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
* Design an intermediary to decouple many peers.
* Promote the many-to-many relationships between interacting peers to "full object status".

1. **Applicability:**

According to Gamma et al, the Mediator pattern should be used when:

* A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
* Reusing an object is difficult because it refers to and communicates with many other objects.
* A behavior that's distributed between several classes should be customizable without a lot of sub classing.

1. **Structure:**

****

1. **Components:**

The Design Components in this pattern are:

* **Mediator:** It defines the interface for communication between colleague objects.
* **ConcreteMediator:** It implements the mediator interface and coordinates communication between colleague objects.
* **Colleague:** It defines the interface for communication with other colleagues
* **ConcreteColleague:** It implements the colleague interface and communicates with other colleagues through its mediator

1. **Consequences:**

* **Comprehension:** The mediator encapsulate the logic of mediation between the colleagues. From this reason it's easier to understand this logic since it is kept in only one class.
* **Decoupled Colleagues:** The colleague classes are totally decoupled. Adding a new colleague class is very easy due to this decoupling level.
* **Simplified object protocols:** The colleague objects need to communicate only with the mediator objects. Practically the mediator pattern reduce the required communication channels (protocols) from many to many to one to many and many to one.
* **Limits Subclassing:** Because the entire communication logic is encapsulated by the mediator class, when this logic need to be extended only the mediator class need to be extended.
* **Complexity:** In practice the mediators tends to become more complex and complex. A good practice is to take care to make the mediator classes responsible only for the communication part. For example when implementing different screens the the screen class should not contain code which is not a part of the screen operations. It should be put in some other classes.

1. **Java Implementation:**

class ATCMediator implements IATCMediator

{

private Flight flight;

private Runway runway;

public boolean land;

public void registerRunway(Runway runway)

{

this.runway = runway;

}

public void registerFlight(Flight flight)

{

this.flight = flight;

}

public boolean isLandingOk()

{

return land;

}

@Override

public void setLandingStatus(boolean status)

{

land = status;

}

}

interface Command

{

void land();

}

interface IATCMediator

{

public void registerRunway(Runway runway);

public void registerFlight(Flight flight);

public boolean isLandingOk();

public void setLandingStatus(boolean status);

}

class Flight implements Command

{

private IATCMediator atcMediator;

public Flight(IATCMediator atcMediator)

{

this.atcMediator = atcMediator;

}

public void land()

{

if (atcMediator.isLandingOk())

{

System.out.println("Successfully Landed.");

atcMediator.setLandingStatus(true);

}

else

System.out.println("Waiting for landing.");

}

public void getReady()

{

System.out.println("Ready for landing.");

}

}

class Runway implements Command

{

private IATCMediator atcMediator;

public Runway(IATCMediator atcMediator)

{

this.atcMediator = atcMediator;

atcMediator.setLandingStatus(true);

}

@Override

public void land()

{

System.out.println("Landing permission granted.");

atcMediator.setLandingStatus(true);

}

}

class MediatorDesignPattern

{

public static void main(String args[])

{

IATCMediator atcMediator = new ATCMediator();

Flight sparrow101 = new Flight(atcMediator);

Runway mainRunway = new Runway(atcMediator);

atcMediator.registerFlight(sparrow101);

atcMediator.registerRunway(mainRunway);

sparrow101.getReady();

mainRunway.land();

sparrow101.land();

}

}