

Design Patterns DAO, Prototype

B.Tech. (IT), Sem-6,
Applied Design Patterns and Application Frameworks (ADPAF)

Dharmsinh Desai University
Prof. H B Prajapati

1

The Data Access Object (DAO) Design Pattern

- Data Persistence
 - In real-life projects, you will encounter situations in which you want to make your **data persist**.
- Diverse data files
 - You might use flat files (i.e., data files on your native OS), XML files, OODBMS, RDBMS, etc. In such situations, you can use the DAO design pattern.
- Abstract persistence mechanism
 - This design pattern **abstracts** the details of the underlying **persistence mechanism** and
 - offers you an **easy-to-use interface** for **implementing the persistence** feature in your application.

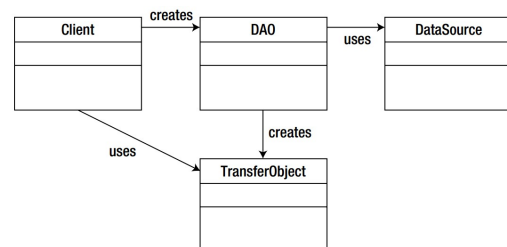
2

The Data Access Object (DAO) Design Pattern

- Loose coupling
 - The DAO pattern **hides** the **implementation details** of the **data source** from its clients, thereby introducing **loose coupling** between your core **business logic** and your **persistence mechanism**.
- Change persistence mechanism
 - This loose coupling enables you to **migrate** from one type of **persistence mechanism** to another without any big-bang change.

3

The Data Access Object (DAO) Design Pattern



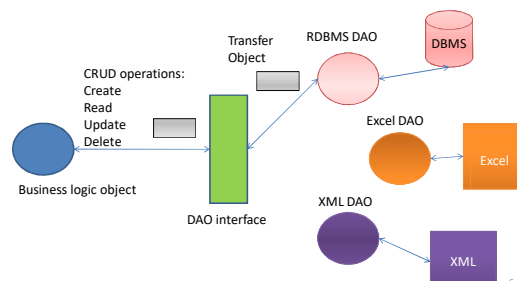
4

The Data Access Object (DAO) Design Pattern

- DAO provides an **abstraction** for the **DataSource**
 - It **hides** the specific **implementation level details**, and
 - Provides a **single interface** for all **different types of data sources**.
- A **Client** is a **user** of DAO pattern that uses DataSource through DAO
- **TransferObject** is a **data transfer object** used as a medium to transfer the core objects.
- Apart from the above-mentioned participants (Client, DAO, TransferObject, and DataSource), there could be one more participant for this pattern—**DAOFactory**.
- You may have multiple DAO objects, corresponding to all the different types of objects you want to store. You may define a factory (using the factory design pattern), which can have **one method for each DAO object**.

5

The Data Access Object (DAO) Design Pattern



6

Example

- We have a Circle class that we want to store in a persistent data store.

```
// Circle.java
public class Circle {
    private int xPos, yPos;
    private int radius;
    public Circle(int x, int y, int r) {
        xPos = x;
        yPos = y;
        radius = r;
    }
    public String toString() {
        return "center = (" + xPos + ", " + yPos + ") and radius = " + radius;
    }
}
```

7

Example

```
public CircleTransfer getCircleTransferObject() {
    CircleTransfer circleTransfer = new CircleTransfer();
    circleTransfer.setRadius(radius);
    circleTransfer.setXPos(xPos);
    circleTransfer.setYPos(yPos);
    return circleTransfer;
}
// other members
}
```

8

Example

```
// CircleDAO.java
public interface CircleDAO {
    public void insertCircle(CircleTransfer circle);
    public CircleTransfer findCircle(int id);
    public void deleteCircle(int id);
}
```

9

Example

```
// RDBMSCircleDAO.java
public class RDBMSCircleDAO implements CircleDAO {
    @Override
    public void insertCircle(CircleTransfer circle) {
        // insertCircle implementation
        System.out.println("insertCircle implementation");
    }
}
```

10

Example

```
@Override
public CircleTransfer findCircle(int id) {
    // findCircle implementation
    return null;
}
@Override
public void deleteCircle(int id) {
    // deleteCircle implementation
}
}
```

11

Example

```
// DAOFactory.java
public class DAOFactory {
    public static CircleDAO getCircleDAO(String sourceType) {
        // This is a simple example, so we have listed only "RDBMS" as the
        // only source type
        // In a real-world application, you can provide more source types
        switch(sourceType){
            case "RDBMS":
                return new RDBMSCircleDAO();
        }
        return null;
    }
}
```

12

Example

```
// CircleTransfer.java
import java.io.Serializable;
public class CircleTransfer implements Serializable {
    private int xPos;
    private int yPos;
    private int radius;
    public int getxPos() {
        return xPos;
    }
    public void setxPos(int xPos) {
        this.xPos = xPos;
    }
}
```

13

Example

```
public int getyPos() {
    return yPos;
}
public void setyPos(int yPos) {
    this.yPos = yPos;
}
public int getRadius() {
    return radius;
}
public void setRadius(int radius) {
    this.radius = radius;
}
}
```

14

Example

```
// Test.java
public class Test {
    public static void main(String[] args) {
        Circle circle = new Circle(10, 10, 20);
        System.out.println(circle);
        CircleTransfer circleTransfer = circle.getCircleTransferObject();
        CircleDAO circleDAO = DAOFactory.getCircleDAO("RDBMS");
        circleDAO.insertCircle(circleTransfer);
    }
}
```

15

Example

- The **Circle** class belongs to your **core business logic**
- The Circle class contains a method **getCircleTransferObject()** that returns the **CircleTransferObject** with the required data.
- You define the **CircleDAO interface** with three methods commonly used with data sources.
- The **RDBMSCircleDAO implements CircleDAO** with a concrete implementation to access the RDBMS data source.
- The **CircleTransfer object** plays a **data carrier role** between the main() method (which is acting as a Client) and DAO implementation (i.e., the RDBMSCircleDAO class).

16

Example

- There might be many DAO objects:
 - CircleDAO,
 - RectangleDAO,
 - SquareDAO, etc.
- You may define getter methods to get corresponding DAO object in a single DAO factory.
- In each method, you may return an appropriate DAO object based on the provided type, as you do in this example with the RDBMS type.

17

Example

- Benefits of DAO design pattern:
 - The DAO **hides the implementation details** of the actual **data source** from the core business logic.
 - The pattern **separates the persistence mechanism** from **rest of the application code** and puts it together in one class specific to one data source.
 - It is quite **easy to extend** support for other data sources using this pattern. For instance, if you want to provide support for the XML-based repository.
 - This can be achieved by defining a new class (say XMLDAO). This new class will implement your CircleDAO interface, such that you do not need to change the way you access the data source. The only thing that needs to be changed is the parameter you pass to DAOFactory to create a DAO.

18

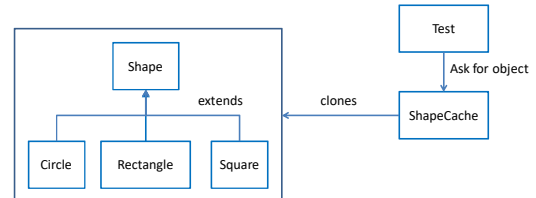
The Prototype design pattern

- It is creational design pattern.
- It can provide best way to create duplicate object while keeping performance in mind.
- This pattern is useful in scenarios where creating objects directly is costly.
 - E.g., an object is created after performing a costly database operation.
 - Objects can be cached, and cached object can be returned when needed.

19

Example: The Prototype design pattern

- We have three types of objects (Square, Circle, and Rectangle)
- We create objects and keep them in cache.



20

Example: The Prototype design pattern

```

package designpattern.prototype;
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;
    }
}

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

21

Example: The Prototype design pattern

```

package designpattern.prototype;
public class Circle extends Shape {
    public Circle(){
        type = "Circle";
        System.out.println("Circle object created");
    }
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
  
```

22

Example: The Prototype design pattern

```

package designpattern.prototype;
public class Rectangle extends Shape {
    public Rectangle(){
        type = "Rectangle";
        System.out.println("Rectangle object created");
    }
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
  
```

23

Example: The Prototype design pattern

```

package designpattern.prototype;
public class Square extends Shape {
    public Square(){
        type = "Square";
        System.out.println("Square object created");
    }
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
  
```

24

Example: The Prototype design pattern

```
package designpattern.prototype;

import java.util.HashMap;

public class ShapeCache {
    private static HashMap<String, Shape> shapeMap = new HashMap<>();
    public static Shape getShape(String shapeId) {
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }
}
```

25

Example: The Prototype design pattern

```
//Keep objects already created.
//Create operation is costly operation.
public static void loadCache() {
    System.out.println("Starting objects creation and caching ");
    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);
    System.out.println("Objects creation and caching completed");
}
```

26






Example: The Prototype design pattern

```
package designpattern.prototype;

public class Test {
    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());
    }
}
```

27

Running the Example: The Prototype design pattern

Notifications	Output - DesignPattern (run)	Search
	run:	
	Starting objects creation and caching	
	Circle object created	
	Square object created	
	Rectangle object created	
	Objects creation and caching completed	
	Shape : Circle	
	Shape : Square	
	Shape : Rectangle	

28

References

- Oracle Certified Professional Java SE 7 Programmer Exams 1Z0-804 and 1Z0-804 (A Comprehension OCPJP 7 Certification Guide), by S G Ganesh and Tushar Sharma, publisher Apress
- Java Design Patterns, problem solving approaches, tutorials point, www.tutorialspoint.com

29