# Design Pattern

# 1.

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content:

# 1-a. Design Patterns là gì

Tilte: Design Patterns là gì

Map: design-pattern-la-gi

Type: Design-Pattern

Meta Des: Design Patterns là gì

Meta Keys: Design Patterns là gì

Description:

Content:

1. **Design Patterns là gì?**

Design patterns là các giải pháp đã được tối ưu hóa, được tái sử dụng cho các vấn đề lập trình mà chúng ta gặp phải hàng ngày. Nó là một khuôn mẫu đã được suy nghĩ, giải quyết trong tình huống cụ thể rồi.

Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.

Design patterns có thể thực hiện được ở phần lớn các ngôn ngữ lập trình. Ta thường gặp nó nhất trong lập trình OOP.

2. **Tại sao phải sử dụng Design Patterns?**

Design Pattern giúp bạn tái sử dụng mã lệnh và dẽ dàng mở rộng.

Nó là tập hơn những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong software engineering. Vậy khi bạn gặp bất kỳ khó khăn gì, design patterns là kim chỉ nam giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp cho một vấn đề đã được chứng minh.

Design pattern cung cấp giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.

Dùng lại các design pattern giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, dễ dàng nâng cấp, bảo trì về sau.

Giúp cho các lập trình viên có thể hiểu code của người khác 1 cách nhanh chóng (có thể hiểu là tính communicate). Mọi thành viên trong team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà k mất quá nhiều thời gian.

**3. Khi nào nên sử dụng Design pattern?**

Giúp cho chương trình của mình thực sự đơn giản. Việc sử dụng các design pattern sẽ giúp chúng ta giảm được thời gian và công sức suy nghĩ ra các cách giải quyết cho những vấn đề đã có lời giải.

Bạn có thể đọc qua cuốn "Head First Design Patterns" để có cái nhìn tổng quát hơn về design pattern.

**4. Phân loại Design Patterns**

Có 3 nhóm chính sau:

Creational Pattern (nhóm khởi tạo) gồm: Abstract Factory, Factory Method, Singleton, Builder, Prototype. Nó sẽ giúp bạn trong việc khởi tạo đối tượng, như bạn biết để khởi tạo bạn phải sử dụng từ khóa new, nhóm Creational Pattern sẽ sử dụng một số thủ thuật để khởi tạo đối tượng mà bạn sẽ không nhìn thấy từ khóa này.

Structural Pattern (nhóm cấu trúc) gồm: Adapter, Bridge, Composite, Decorator, Facade, Proxy và Flyweight. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

Behavioral Pattern gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng.

**5.Để học Design Patterns cần có gì?**

Bốn đặc tính của OOP: Thừa kế, Đa hình, Trừu tượng, Bao đóng.

Khái niệm interface và abstract. Cái này cực kỳ quan trọng, để hiểu và áp dụng 2 khái niệm này có thể sẽ mất một thời gian, nhưng khi bạn nắm chắc nó bạn sẽ thấy nó thực sự cần thiết.

Bỏ tư duy theo lối cấu trúc, nâng tư duy hoàn toàn OOP.

6. Kết luận

Design patterns là 1 đoạn mô tả, hoặc 1 khuôn mẫu để giải quyết 1 vấn đề nào đó.

Không phải là thiết kế cuối cùng.

Cho phép code được tối ưu hóa, dễ tái sử dụng. Người khác dễ dàng nắm bắt được code của bạn. Dễ nâng cấp sửa chữa.

Giúp cho lập trình viên có thể communicate với nhau.

# 2. Example Design Pattern Factory Pattern - Creational Pattern

Tilte: Example Design Pattern Factory Pattern - Creational Pattern

Map: example-design-pattern-factory-pattern-creational-pattern

Type: Design-Pattern

Meta Des: Example Design Pattern Factory Pattern - Creational Pattern

Meta Keys: Example Design Pattern Factory Pattern - Creational Pattern

Description:

Content:

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Example : Create a app allow user create draw shape . Use factory pattern

1. Shape.java

package com.codewr.example.creation.factory.pattern;

/\*\*

\*

\* @author codewr

\*/

public interface Shape {

void draw();

}

2. Circle.java

package com.codewr.example.creation.factory.pattern;

/\*\*

\*

\* @author codewr

\*/

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Draw a circle !");

}

}

3.Square.java

package com.codewr.example.creation.factory.pattern;

/\*\*

\*

\* @author codewr

\*/

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Draw a square !");

}

}

4. Rectangle.java

package com.codewr.example.creation.factory.pattern;

/\*\*

\*

\* @author codewr

\*/

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Draw a rectangle !");

}

}

5. ShapeFactory

package com.codewr.example.creation.factory.pattern;

/\*\*

\*

\* @author codewr

\*/

public class ShapeFactory {

//use getShape method to get object of type shape

public Shape getShape(String shapeType) {

if (shapeType == null) {

return null;

}

if (shapeType.equalsIgnoreCase("CIRCLE")) {

return new Circle();

} else if (shapeType.equalsIgnoreCase("RECTANGLE")) {

return new Rectangle();

} else if (shapeType.equalsIgnoreCase("SQUARE")) {

return new Square();

}

return null;

}

}

6. App.java

package com.codewr.example.creation.factory.pattern;

/\*\*

\*

\* @author codewr

\*/

public class App {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

//get an object of Circle and call its draw method.

Shape shape1 = shapeFactory.getShape("circle");

//call draw a Circle

shape1.draw();

//get an object of Rectangle and call its draw method.

Shape shape2 = shapeFactory.getShape("rectangle");

//call draw a Rectangle

shape2.draw();

//get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("square");

//call draw a square

shape3.draw();

}

}

7. Run App.java

Draw a circle !

Draw a rectangle !

Draw a square !

# 3. Example Design Pattern Abstract Factory Pattern – Creational Pattern

Tilte: Example Design Pattern Abstract Factory Pattern – Creational Pattern

Map: example-design-pattern-abstract-factory-pattern-creational-pattern

Type: Design-Pattern

Meta Des: Example Design Pattern Abstract Factory Pattern – Creational Pattern

Meta Keys: Example Design Pattern Abstract Factory Pattern – Creational Pattern

Description:

Content:

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Example: Create a app allow user create draw shape and fill color for shape. Use abstract pattern.

1. Shape.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public interface Shape {

void draw();

}

2. Circle.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Draw a circle !");

}

}

3. Rectangle.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Draw a rectangle !");

}

}

4.Square.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Draw a square !");

}

}

5.Color.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public interface Color {

void fill();

}

6.Blue.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class Blue implements Color {

@Override

public void fill() {

System.out.println("fill a blue color!");

}

}

7.Green.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class Green implements Color {

@Override

public void fill() {

System.out.println("fill a green color!");

}

}

8. Red.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class Red implements Color {

@Override

public void fill() {

System.out.println("fill a red color!");

}

}

9. AbstractFactory.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public abstract class AbstractFactory {

abstract Color getColor(String color);

abstract Shape getShape(String shape);

}

10. ShapeFactory.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class ShapeFactory extends AbstractFactory {

@Override

Shape getShape(String shapeType) {

if (shapeType == null) {

return null;

}

if (shapeType.equalsIgnoreCase("CIRCLE")) {

return new Circle();

} else if (shapeType.equalsIgnoreCase("RECTANGLE")) {

return new Rectangle();

} else if (shapeType.equalsIgnoreCase("SQUARE")) {

return new Square();

}

return null;

}

@Override

Color getColor(String color) {

throw new UnsupportedOperationException("Not supported yet.");

}

}

11. ColorFactory.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class ColorFactory extends AbstractFactory {

@Override

Color getColor(String color) {

if (color == null) {

return null;

}

if (color.equalsIgnoreCase("RED")) {

return new Red();

} else if (color.equalsIgnoreCase("GREEN")) {

return new Green();

} else if (color.equalsIgnoreCase("BLUE")) {

return new Blue();

}

return null;

}

@Override

Shape getShape(String shape) {

throw new UnsupportedOperationException("Not supported yet.");

}

}

12. FactoryProducer.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class FactoryProducer {

public static AbstractFactory getFactory(String choice){

if(choice.equalsIgnoreCase("SHAPE")){

return new ShapeFactory();

}else if(choice.equalsIgnoreCase("COLOR")){

return new ColorFactory();

}

return null;

}

}

13.App.java

package com.codewr.example;

/\*\*

\*

\* @author codewr

\*/

public class App {

public static void main(String[] args) {

//get shape factory

AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

//get an object of Shape Circle

Shape shape1 = shapeFactory.getShape("circle");

//call draw method of Shape Circle

shape1.draw();

//get an object of Shape Rectangle

Shape shape2 = shapeFactory.getShape("rectangle");

//call draw method of Shape Rectangle

shape2.draw();

//get an object of Shape Square

Shape shape3 = shapeFactory.getShape("square");

//call draw method of Shape Square

shape3.draw();

//get color factory

AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

//get an object of Color Red

Color color1 = colorFactory.getColor("red");

//call fill method of Red

color1.fill();

//get an object of Color Green

Color color2 = colorFactory.getColor("green");

//call fill method of Green

color2.fill();

//get an object of Color Blue

Color color3 = colorFactory.getColor("blue");

//call fill method of Color Blue

color3.fill();

}

}

14. Run

Draw a circle !

Draw a rectangle !

Draw a square !

fill a red color!

fill a green color!

fill a blue color!

# 4. Example Design Pattern Singleton Pattern – Creational Pattern

Tilte: Example Design Pattern Singleton Pattern – Creational Pattern

Map: example-design-pattern-singleton-pattern-creational-pattern

Type: Design-Pattern

Meta Des: Example Design Pattern Singleton Pattern – Creational Pattern

Meta Keys: Example Design Pattern Singleton Pattern – Creational Pattern

Description:

Content:

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Example: Create a app. Use singleton Pattern.

1. SingleOb.java

package com.codewr.example.creational.singleton.pattern;

/\*\*

\*

\* @author codewr

\*/

public class SingleOb {

//create an object of SingleOb

private static SingleOb instance = new SingleOb();

//make the constructor private so that this class cannot be

//instantiated

private SingleOb() {

}

//Get the only object available

public static SingleOb getInstance() {

return instance;

}

public void showMessage() {

System.out.println("App use Singleton Pattern!");

}

}

2. App.java

package com.codewr.example.creational.singleton.pattern;

/\*\*

\*

\* @author codewr

\*/

public class App {

public static void main(String[] args) {

SingleOb object = SingleOb.getInstance();

//show the message

object.showMessage();

}

}

3. Run

App use Singleton Pattern!

# 5. Example Design Pattern Builder Pattern – Creational Pattern

Tilte: Example Design Pattern Builder Pattern – Creational Pattern

Map: example-design-pattern-builder-pattern-creational-pattern

Type: Design-Pattern

Meta Des: Example Design Pattern Builder Pattern – Creational Pattern

Meta Keys: Example Design Pattern Builder Pattern – Creational Pattern

Description:

Content:

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.

Example: Create app for a fast-food restaurant

1. Packing.java

package com.codewr.example.interfaces;

/\*\*

\*

\* @author codewr

\*/

public interface Packing {

public String pack();

}

2.Item.java

package com.codewr.example.interfaces;

/\*\*

\*

\* @author codewr

\*/

public interface Item {

public String name();

public Packing packing();

public float price();

}

3. Wrapper.java

package com.codewr.example;

import com.codewr.example.interfaces.Packing;

/\*\*

\*

\* @author codewr

\*/

public class Wrapper implements Packing {

@Override

public String pack() {

return "Wrapper";

}

}

4. Bottle.java

package com.codewr.example;

import com.codewr.example.interfaces.Packing;

/\*\*

\*

\* @author codewr

\*/

public class Bottle implements Packing {

@Override

public String pack() {

return "Bottle";

}

}

5.ColdDrink.java

package com.codewr.example.abstracts;

import com.codewr.example.Bottle;

import com.codewr.example.interfaces.Packing;

import com.codewr.example.interfaces.Item;

/\*\*

\*

\* @author codewr

\*/

public abstract class ColdDrink implements Item {

@Override

public Packing packing() {

Packing packing = new Bottle();

return packing;

}

@Override

public abstract float price();

}

6. Burger.java

package com.codewr.example.abstracts;

import com.codewr.example.Wrapper;

import com.codewr.example.interfaces.Packing;

import com.codewr.example.interfaces.Item;

/\*\*

\*

\* @author codewr

\*/

public abstract class Burger implements Item {

@Override

public Packing packing() {

Packing packing = new Wrapper();

return packing;

}

@Override

public abstract float price();

}

7.ChickenBurger.java

package com.codewr.example.abstracts.extds;

import com.codewr.example.abstracts.Burger;

/\*\*

\*

\* @author codewr

\*/

public class ChickenBurger extends Burger {

@Override

public float price() {

return 50.5f;

}

@Override

public String name() {

return "Chicken Burger";

}

}

8. VegBurger.java

package com.codewr.example.abstracts.extds;

import com.codewr.example.abstracts.Burger;

/\*\*

\*

\* @author codewr

\*/

public class VegBurger extends Burger {

@Override

public float price() {

return 25.0f;

}

@Override

public String name() {

return "Veg Burger";

}

}

9.Pepsi.java

package com.codewr.example.abstracts.extds;

import com.codewr.example.abstracts.ColdDrink;

/\*\*

\*

\* @author codewr

\*/

public class Pepsi extends ColdDrink {

@Override

public float price() {

return 35.0f;

}

@Override

public String name() {

return "Pepsi";

}

}

10. Coke.java

package com.codewr.example.abstracts.extds;

import com.codewr.example.abstracts.ColdDrink;

/\*\*

\*

\* @author codewr

\*/

public class Coke extends ColdDrink {

@Override

public float price() {

return 30.0f;

}

@Override

public String name() {

return "Coke";

}

}

11. Meal.java

package com.codewr.example;

import com.codewr.example.interfaces.Item;

import java.util.ArrayList;

import java.util.List;

/\*\*

\*

\* @author codewr

\*/

public class Meal {

private List<Item> items = new ArrayList<Item>();

public void addItem(Item item) {

items.add(item);

}

public float getCost() {

float cost = 0.0f;

for (Item item : items) {

cost += item.price();

}

return cost;

}

public void showItems() {

for (Item item : items) {

System.out.print("Item : " + item.name());

System.out.print(", Packing : " + item.packing().pack());

System.out.println(", Price : " + item.price());

}

}

}

12. MealBuilder.java

package com.codewr.example;

import com.codewr.example.abstracts.extds.Coke;

import com.codewr.example.abstracts.extds.Pepsi;

import com.codewr.example.abstracts.extds.ChickenBurger;

import com.codewr.example.abstracts.extds.VegBurger;

/\*\*

\*

\* @author codewr

\*/

public class MealBuilder {

public Meal prepareVegMeal() {

Meal meal = new Meal();

meal.addItem(new VegBurger());

meal.addItem(new Coke());

return meal;

}

public Meal prepareNonVegMeal() {

Meal meal = new Meal();

meal.addItem(new ChickenBurger());

meal.addItem(new Pepsi());

return meal;

}

}

13.App.java

package com.codewr.example.app;

import com.codewr.example.Meal;

import com.codewr.example.MealBuilder;

/\*\*

\*

\* @author codewr

\*/

public class App {

public static void main(String[] args) {

MealBuilder mealBuilder = new MealBuilder();

Meal vegMeal = mealBuilder.prepareVegMeal();

System.out.println("Vegetable Meal");

vegMeal.showItems();

System.out.println("Total Cost: " + vegMeal.getCost());

Meal nonVegMeal = mealBuilder.prepareNonVegMeal();

System.out.println("Non-Vegetable Meal");

nonVegMeal.showItems();

System.out.println("Total Cost: " + nonVegMeal.getCost());

}

}

14. Run

Vegetable Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Vegetable Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

# 6. Example Design Pattern Prototype pattern – Creational Pattern

Tilte: Example Design Pattern Prototype pattern – Creational Pattern

Map: example-design-pattern-prototype-pattern-creational-pattern

Type: Design-Pattern

Meta Des: Example Design Pattern Prototype pattern – Creational Pattern

Meta Keys: Example Design Pattern Prototype pattern – Creational Pattern

Description:

Content:

Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

Example: Create app loadCache from database 1 time

1. Shape.java

package com.codewr.example.creational.prototype.pattern;

/\*\*

\*

\* @author codewr

\*/

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;

}

}

2.Circle.java

package com.codewr.example.creational.prototype.pattern;

/\*\*

\*

\* @author codewr

\*/

public class Circle extends Shape {

public Circle() {

type = "Circle";

}

@Override

public void draw() {

System.out.println("Draw a circle !");

}

}

3. Rectangle.java

package com.codewr.example.creational.prototype.pattern;

/\*\*

\*

\* @author codewr

\*/

public class Rectangle extends Shape {

public Rectangle() {

type = "Rectangle";

}

@Override

public void draw() {

System.out.println("Draw a rectangle !");

}

}

4.Square.java

package com.codewr.example.creational.prototype.pattern;

/\*\*

\*

\* @author codewr

\*/

public class Square extends Shape {

public Square() {

type = "Square";

}

@Override

public void draw() {

System.out.println("Draw a square !");

}

}

5.ShapeCache.java

package com.codewr.example.creational.prototype.pattern;

import java.util.Hashtable;

/\*\*

\*

\* @author codewr

\*/

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);

}

}

6. App.java

package com.codewr.example.creational.prototype.pattern;

/\*\*

\*

\* @author codewr

\*/

public class App {

public static void main(String[] args) {

ShapeCache.loadCache();

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

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

clonedShape.draw();

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

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

clonedShape2.draw();

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

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

clonedShape3.draw();

}

}

7.Run

Shape : Circle

Draw a circle !

Shape : Square

Draw a square !

Shape : Rectangle

Draw a rectangle !

.8.

# 7. Example Design Pattern Object Pool Pattern – Creational Pattern

Tilte: Example Design Pattern Object Pool Pattern – Creational Pattern

Map: example-design-pattern-object-pool-pattern-creational-pattern

Type: Design-Pattern

Meta Des: Example Design Pattern Object Pool Pattern – Creational Pattern

Meta Keys: Example Design Pattern Object Pool Pattern – Creational Pattern

Description:

Content:

Object pooling can offer a significant performance boost; it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.

Example: Create app have use object pool pattern

1.ObjectPool.java

package com.codewr.examplecreational.object.pool.pattern;

import java.util.Enumeration;

import java.util.Hashtable;

/\*\*

\*

\* @author codewr

\*/

public abstract class ObjectPool<T> {

private long expirationTime;

private Hashtable<T, Long> locked, unlocked;

public ObjectPool() {

expirationTime = 30000; // 30 seconds

locked = new Hashtable<T, Long>();

unlocked = new Hashtable<T, Long>();

}

protected abstract T create();

public abstract boolean validate(T o);

public abstract void expire(T o);

public synchronized T checkOut() {

long now = System.currentTimeMillis();

T t;

if (unlocked.size() > 0) {

Enumeration<T> e = unlocked.keys();

while (e.hasMoreElements()) {

t = e.nextElement();

if ((now - unlocked.get(t)) > expirationTime) {

// object has expired

unlocked.remove(t);

expire(t);

t = null;

} else if (validate(t)) {

unlocked.remove(t);

locked.put(t, now);

return (t);

} else {

// object failed validation

unlocked.remove(t);

expire(t);

t = null;

}

}

}

// no objects available, create a new one

t = create();

locked.put(t, now);

return (t);

}

public synchronized void checkIn(T t) {

locked.remove(t);

unlocked.put(t, System.currentTimeMillis());

}

}

2. JDBCConnectionPool.java

package com.codewr.examplecreational.object.pool.pattern;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

/\*\*

\*

\* @author codewr

\*/

//The three remaining methods are abstract

//and therefore must be implemented by the subclass

public class JDBCConnectionPool extends ObjectPool<Connection> {

private String dsn, usr, pwd;

public JDBCConnectionPool(String driver, String dsn, String usr, String pwd) {

super();

try {

Class.forName(driver).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

this.dsn = dsn;

this.usr = usr;

this.pwd = pwd;

}

@Override

protected Connection create() {

try {

Connection connection = DriverManager.getConnection(dsn, usr, pwd);

System.out.println("create connection success");

return connection;

} catch (SQLException e) {

e.printStackTrace();

return (null);

}

}

@Override

public void expire(Connection o) {

try {

((Connection) o).close();

System.out.println("Connection expire! ");

} catch (SQLException e) {

e.printStackTrace();

}

}

@Override

public boolean validate(Connection o) {

try {

return (!((Connection) o).isClosed());

} catch (SQLException e) {

e.printStackTrace();

return (false);

}

}

}

3.App.java

package com.codewr.examplecreational.object.pool.pattern;

import java.sql.Connection;

/\*\*

\*

\* @author codewr

\*/

public class App {

public static void main(String args[]) {

// Do something...

// ...

// Create the ConnectionPool:

JDBCConnectionPool pool = new JDBCConnectionPool("com.mysql.cj.jdbc.Driver",

"jdbc:mysql://localhost:3306/codewr", "root", "");

// Get a connection:

Connection con = pool.checkOut();

Thread.sleep(10000);

Connection con2 = pool.checkOut();

Thread.sleep(25000);

boolean isConnected = pool.validate(con);

boolean isConnected2= pool.validate(con2);

System.out.println("con isconnect: "+isConnected);

System.out.println("con2 isconnect: "+isConnected2);

// Use the connection

// ...

// Return the connection:

pool.checkIn(con);

}

}

4.Run:

create connection success

create connection success

con isconnect: true

con2 isconnect: true

5.

# 8.

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content:

# Sample

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content:

# Sample

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content:

# Sample

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content:

# Sample

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content:

# Sample

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content:

# Sample

Tilte:

Map:

Type: Design-Pattern

Meta Des:

Meta Keys:

Description:

Content: