Object oriented programming – supervision 3

James Wood

January 24, 2015

a

The singleton pattern would be used. The singleton pattern ensures that only one instance of the class is created whilst the program is running. The `Configuration` class would have a static member of type `Configuration`, used whenever an instance of that type is needed. It will be modified each time it is used. The member will be private, and be accessed by a method that mutates it and then returns a reference to it.

b

Behavioural patterns are patterns used to specify how objects interact, including specification of run time details. Creational patterns are used to control the building of new objects. Compositional patterns describe how objects of different types can be combined to form objects of new types.

The *Bridge* and *Strategy* patterns both allow an abstract task to be specified along with various implementations. However, the *Bridge* pattern is designed to give an interface to many different existing classes, whereas the *Strategy* pattern is generally set up before specifying implementations. What implementation the *Bridge* pattern uses is based on the type of value the *Bridge* instance is wrapping, but what implementation the *Strategy* chooses is based on an arbitrary run time decision process.

c

The *Adapter* pattern provides a way of accessing the members of multiple different classes with a common interface. This is a useful way to abstract away the differences between multiple different implementations of a data structure. The *Faade* pattern provides a way to give a collection of classes a unified interface, in which members of all of the classes can be accessed through a single instance of the *Faade*.

d

1. Yes.
2.
```
// ‘shapes‘ is the list of Shape objects
for (Shape s : shapes)
    s.draw();
```
3.
```
class Group extends Shape {
    private List<Shape> mShapes;

    public Group(List<Shape> shapes) {
        mShapes = shapes;
    }

    @Override
    public void draw() {
        for (Shape s : mShapes)
            s.draw();
    }
}
```

4. The *Decorator* is used as follows:

```
class Frame extends Shape {
    private Shape mShape;

    public Frame(Shape shape) {
        mShape = shape;
    }

    @Override
    public void draw() {
        mShape.draw();

        // draw frame around shape
        // ...
    }
}
```

e

The *Observer* pattern is the primary pattern used in solving this problem.

```
package code.s3;

import java.util.LinkedList;

public class User {
    private LinkedList<User> mFollowers = new LinkedList<>();
    private String mName;

    public User(String name) {
        mName = name;
    }

    public String getName() { return mName; }

    public void tweet(String msg) {
        System.out.println(mName + "␣tweeted:␣" + msg);
        for (User u : mFollowers)
            u.notify(msg);
    }

    public void follow(User user) {
        System.out.println(mName + "␣followed␣" + user.getName());
        user.registerFollower(this);
    }

    public void registerFollower(User user) {
        mFollowers.add(user);
    }

    public void notify(String msg) {
        System.out.println(mName + "␣received:␣" + msg);
    }
}
```

Test:

```
package code.s3;

public class S3e {
    public static void main(String[] args) {
        User u0 = new User("u0");
        User u1 = new User("u1");

        u0.tweet("u0's␣first␣tweet");
        u1.follow(u0);
        u0.tweet("u0's␣second␣tweet");
        u1.tweet("u1's␣first␣tweet");
        u0.follow(u1);
        u1.tweet("u1's␣second␣tweet");
    }
}
```

Output:

```
u0 tweeted: u0's first tweet
u1 followed u0
u0 tweeted: u0's second tweet
u1 received: u0's second tweet
u1 tweeted: u1's first tweet
u0 followed u1
u1 tweeted: u1's second tweet
u0 received: u1's second tweet
```