CS335 Software Engineering Process

Lab Week 08

Prototype Design Pattern

In this lab, we are going to study another useful design pattern – the prototype design pattern.

Introduction

In Object Oriented Programming (OOP), you need objects to work with; objects interact with each other to get the job done. But sometimes, creating a heavy object could become costly, and if your application needs too many of that kind of objects (containing almost similar properties), it might create performance issues.

Let us consider a scenario where an application requires access control. The features of the applications can be used by the users according to the access rights provided to them. For example, some users have access to the reports generated by the application, while some don't. Some of them even can modify the reports, while some can only read it. Some users also have administrative rights to add or even remove other users.

Every user object has an access control object, which is used to provide or restrict the controls of the application. This access control object is a bulky, heavy object and its creation is very costly since it requires data to be fetched from some external resources, like databases or some property files etc.

We also cannot share the same access control object with users of the same level, because the rights can be changed at runtime by the administrator and a different user with the same level could have a different access control. One user object should have one access control object.

We can use the Prototype Design Pattern to resolve this problem by creating the access control objects on all levels at once, and then provide a copy of the object

to the user whenever required. In this case, data fetching from the external resources happens only once. Next time, the access control object is created by copying the existing one. The access control object is not created from scratch every time the request is sent; this approach will certainly reduce object creation time.

The Prototype design pattern is used to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when the object creation is a heavy process.

In Java, there are certain ways to copy an object in order to create a new one. One way to achieve this is using the Cloneable interface. Java provides the clone method, which an object inherits from the Object class. You need to implement the Cloneable interface and override this *clone* method according to your needs.

Solution to the Problem

```
public interface Prototype extends Cloneable {
	public AccessControl clone() throws CloneNotSupportedException;
}
```

The above interface extends the Cloneable interface and contains a method clone. This interface is implemented by classes which want to create a prototype object.

```
public class AccessControl implements Prototype{
	private final String controlLevel;
	private String access;
	public AccessControl(String controlLevel,String access){
		this.controlLevel = controlLevel;
		this.access = access;
	}
```

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

// download pattern.zip to see the full source code.
```

The *AccessControl* class implements the Prototype interface and overrides the clone method. The method calls the clone method of the super class and returns the object after down-casting it to the *AccessControl* type. The clone method throws *CloneNotSupportedException* which is caught within the method itself.

The class also contains two properties; the *controlLevel* is used to specific the level of control this object contains. The level depends upon the type of user going to use it, for example, USER, ADMIN, MANAGER etc.

The other property is the access; it contains the access right for the user. Please note that, for simplicity, we have used access as a String type attribute. This could be of type Map which can contain key value pairs of long access rights assigned to the user.

```
public class User {
	private String userName;
	private String level;
	private AccessControl accessControl;

	public User(String userName,String level, AccessControl
accessControl){
		this.userName = userName;
		this.level = level;
```

```
            this.accessControl = accessControl;

        }
        public String getUserName() {
            return userName;
        }
        public void setUserName(String userName) {
            this.userName = userName;
        }
        public String getLevel() {
            return level;
        }
        public void setLevel(String level) {
            this.level = level;
        }

        public AccessControl getAccessControl() {
            return accessControl;
        }
        public void setAccessControl(AccessControl accessControl) {
            this.accessControl = accessControl;

        }
```

The User class has a userName, level and a reference to the AccessControl assigned to it.

We have used an *AccessControlProvider* class that creates and stores the possible *AccessControl* objects in advance. And when the there's a request to an *AccessControl* object, it returns a new object created by copying the stored prototypes.

```
import java.util.HashMap;
import java.util.Map;

public class AccessControlProvider {
        private static Map<String, AccessControl>map = new
HashMap<String, AccessControl>();
        static{
```

```
            System.out.println("Fetching data from external resources and
creating access control objects...");
            map.put("USER", new AccessControl("USER","DO_WORK"));
            map.put("ADMIN", new AccessControl("ADMIN","ADD/REMOVE
USERS"));
            map.put("MANAGER", new
AccessControl("MANAGER","GENERATE/READ REPORTS"));
            map.put("VP", new AccessControl("VP","MODIFY REPORTS"));
        }

        public static AccessControl getAccessControlObject(String
controlLevel){
            AccessControl ac = null;
            ac = map.get(controlLevel);
            if(ac!=null){
                return ac.clone();
            }
            return null;
        }

}
```

The *getAccessControlObject* method fetches a stored prototype object according to the controlLevel passed to it, from the map and returns a newly created cloned object to the client code.

Now, let's test the code.

```
public class TestPrototypePattern {
        public static void main(String[] args) {
            AccessControl userAccessControl =
AccessControlProvider.getAccessControlObject("USER");
            User user = new User("User A", "USER Level",
userAccessControl);
            System.out.println("************************************");
            System.out.println(user);
```

```
                userAccessControl =
AccessControlProvider.getAccessControlObject("USER");
                user = new User("User B", "USER Level", userAccessControl);
                System.out.println("Changing access control of:
"+user.getUserName());
                user.getAccessControl().setAccess("READ REPORTS");
                System.out.println(user);

                System.out.println("************************************");
                AccessControl managerAccessControl =
AccessControlProvider.getAccessControlObject("MANAGER");
                user = new User("User C", "MANAGER Level",
managerAccessControl);

System.out.println(user);

        }

}
```

In the above code, we have created an *AccessControl* object at USER level and assigned it to User A. Then, again another *AccessControl* object to User B, but this time we have changed the access right of User B. And in the end, MANAGER level access control to User C.

The *getAccessControlObject* is used to get the new copy of the *AccessControl* object, and this can be clearly seen when we change the access right for User B, the access right for User A is not changed (just print the User A object again). This confirms that the clone method is working fine, as it returns the new copy of the object not a reference which points to the same object.

When to use the Prototype Design Pattern

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

• When the classes to instantiate are specified at run-time, for example, by dynamic loading; or

• To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

• When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Exercise 1. Download prototype.zip, compile and run the source code.

Exercise 2. Add a SUPERUSER that allows ADD/REMOVE USERS and INSTALL/UNINSTALL APPLICATIONS.

Exercise 3. Add a new feature that **only** allows SUPERUSER to view all other created USERS including USER, ADMIN MANAGER, VP and SUPERUSER. Hint: you need to use a data structure to remember every user created. Note that you should make sure every user's name is unique (case sensitive). In other words, you cannot create a user already exists.

Test your code in the TestPrototypePattern class.