THE UNIVERSITY OF SYDNEY

SOFT2201/COMP9201 Tutorial 10

Prototype & Memento

Prototype Pattern

Prototype is a pattern that is easy to explain in concept, but very easy to implement poorly (as is Memento). In order to meet the benefits the lecture describes for this pattern it is necessary to have a deep understanding of how memory, scope, and variable storage work in the programming language you are implementing it in (if you had trouble in the early weeks of this unit a review of this material would be useful).

Consider the following code:

```
public class ComplexObject {
    private PrimaryStrategy primaryStrategy;
    private SecondaryStrategy secondaryStrategy;

    // The strategies are created by calling a network
    // socket that takes on average
    // 1.7 seconds to respond
    public ComplexObject(primaryStrategy, secondaryStrategy) {
        this.primaryStrategy = primaryStrategy;
        this.secondaryStrategy = secondaryStrategy;
    }

    // A lot of operations, accessors, and mutators
}
```

The quick, easy, and usually very wrong cloning method to implement looks like this:

```
public ComplexObject copy() {
    return new ComplexObject(primaryStrategy, secondaryStrategy);
}
```

Hopefully you recognise this will copy the same strategy instances as the source object. This is fine (and preferable) if they are able to be shareable (recognising the difference between Shared Aggregation and Composition here is important) - but will be a bug if not.

Question 1: Simple Object Copying

As a class, write the cloning method that assumes SecondaryStrategies can be shared, but PrimaryStrategies can not. Identify what methods you need to assume exist in the strategies in order to achieve this. Also consider an alternative implementation whose method signature looks like this:

```
public ComplexObject (ComplexObject source);
```

Question 2: Implementing Prototype

Individually now, assume the following:

- There are 3 versions of PrimaryStrategy and 2 of SecondaryStrategy
- All combinations are useful and will be repeated - e.g. a ComplexObject with PrimaryStrategy 1 and SecondaryStrategy 2
- You want client code to be able to use these combinations as if they were subclasses of ComplexObject

Write an appropriate ComplexObjectRegistry class that provides this functionality (use the pretend FakeNetwork stub class). Each network call for a specific version of a strategy should be made at most once over the life of the application. See if you can work out a way to have this all happen at the beginning of the application, AND a way to delay each call until the precise time it is required - but only ever being called once. You will also need to write the strategy classes - remember SecondaryStrategies can be shared while PrimaryStrategies cannot.

```
public class FakeNetwork {
    public static PrimaryStrategy getPrimaryStrategy(int version) {
        try {
            // This is simulating a slow network
            // connection that is serialising our object,
            // or sending us a database response
            // telling us what to create, etc.
            Thread.sleep(1700);
        } catch (InterruptedException ignored) {}
        // This could be split by subclass as well as
        // by instance variable
        return new PrimaryStrategy(version);
    }

    public static SecondaryStrategy getSecondaryStrategy(int version) {
        try {
            Thread.sleep(1700);
        } catch (InterruptedException ignored) {}
        return new SecondaryStrategy(version);
    }
}
```

Memento

Memento is somewhat similar to Prototype in its internal structure, but solves a very different problem. Rather than storing different versions of classes, Memento stores different versions of the internals of an object. In our previous example, if we were swapping Strategies, or changing strategy states, Memento would store deep copies of those strategies (unless they were immutable, in which case they could be shallow copies) in an encapsulated way, ready to be swapped back into our ComplexObject to revert it to some previous state.

Question 3: Implementing Memento Step One: Context

The Memento exercise is in 3 stages. In this question you will need to write a class (and its dependencies) that is going to be stored using Memento. This class should provide the following features:

- It is a Student record
- It stores the units of study a student has completed, which semester, and the grades achieved
- It stores the current units of study a student is in this semester
- It stores the planned units of study a student wants to take
- It stores various Student data: SID, name, DOB
- Units of study have a subject name and unit code
- Units of study details may not change
- Grades are between 0.0 and 100.0
- The grade a student has in a given unit starts at 0, and can change only during the semester it is being taken in
- Semesters start with 1 at the start of the application and count upwards
- The Student records are stored in a Class object that manages the application

Try to use as many immutable elements as you can.

Question 4: Implementing Memento Step Two: The Bad Way

The student has invented a time machine, and can now reset their units of study to a previous state. They must also be able to return back to the last future state they left.

Don't use the Memento pattern to achieve this just yet, add Class.revertStudent(Student student, int numberOfSemesters) and Class.returnStudent(Student student) methods that handle this manually.

Question 5: Implementing Memento Step Three: Memento

Now replace the contents of these methods to use a StudentMemento class and a StudentMemento-Store (eg Caretaker) class.

Take a moment (pun intended) to reflect on the changes this has made to the code - if you have implemented Memento correctly you should be seeing a massive improvement in the proper encapsulation between classes and methods. In the 'bad way' Class knows far too much about the internals of Students, and manipulates them in ways that lead to very high coupling. Using a Memento object you can instead abstract this to a more appropriate class.