Practice midterm

[2 minutes]

Suppose you need to design an alarm system that allows you to add different alarming devices such as a flashing light, an alarming horn, etc. It should be easy to add new alarming devices. Give the name of the pattern that you would use so that it is easy to add new alarming devices.

Observer

[2 minutes]

Suppose you need to design a payment system that can handle different payments. The system should support all popular payment types like visa, mastercard, debit card, PayPal, apple pay, etc. It should be easy to add new payment types. Give the name of the pattern that you would use so that it is easy to add new payment types.

Chain of responsibility

[2 minutes]

Suppose you need to write a windows UI framework, and you need to implement event handling for UI controls like buttons, check boxes, etc. Give the name of the pattern that you would use for event handling

Observer

[2 minutes]

We need to be able to record all actions done by the user so we can replay these actions later. Give the name of the pattern that you would use to implement this.

Command

[25 minutes]

We have to write the software that goes into a fancy ceiling fan. This ceiling fan has 4 speed states, with 2 pull chain cords. A red pull chain cord and a green pull chain cord.
The 4 speed states are: Off, Low, Medium and High
The ceiling fan always starts in the Off state.

If you pull the green cord in the Off state the fan goes to the Low speed
If you pull the green cord in the Low state the fan goes to the Medium speed
If you pull the green cord in the Medium state the fan goes to the High speed
If you pull the green cord in the High state the fan goes to the Off speed

If you pull the red cord in the Off state the fan goes to the High speed
If you pull the red cord in the High state the fan goes to the Medium speed
If you pull the red cord in the Medium state the fan goes to the Low speed
If you pull the red cord in the Low state the fan goes to the Off speed

Here is the code that works correctly:

```
public class Application {

   public static void main(String[] args) {
      CeilingFan fan = new CeilingFan();
      fan.pullgreen();
      fan.pullgreen();
      fan.pullgreen();
      fan.pullgreen();
      fan.pullred();
      fan.pullred();
   }
}

public class CeilingFan {
   int current_state=0;


   public void pullgreen() {
        if (current_state == 0) {
          current_state = 1;
          System.out.println( "low speed" );
        } else if (current_state == 1) {
          current_state = 2;
          System.out.println( "medium speed" );
        } else if (current_state == 2) {
          current_state = 3;
          System.out.println( "high speed" );
        } else {
          current_state = 0;
          System.out.println( "turning off" );
        }
   }
```

```
    public void pullred() {
        if (current_state == 0) {
          current_state = 3;
          System.out.println( "high speed" );
        } else if (current_state == 1) {
          current_state = 0;
          System.out.println( "turning off" );
        } else if (current_state == 2) {
          current_state = 1;
          System.out.println( "low speed" );
        } else {
          current_state = 2;
          System.out.println( "medium speed" );
        }
    }
}
```

We learned that the State pattern can improve our application when we introduce new states. When we implement the state pattern to the given code, write the Java code of the class that is responsible for the functionality of the medium state.

```
public class MediumState extends FanState {

	public MediumState(CeilingFan fan) {
		super(fan);
		System.out.println("medium state");
	}


	@Override
	public void pullgreen() {
		fan.setState(new HighState(fan));
		}


	@Override
	public void pullred() {
		fan.setState(new LowState(fan));
	}
}
```

[15 minutes]

Suppose we have the following calculator class:

class Calculator
Attribute: currentvalue
Methods:
void add(int value)
void subtract(int value)
int getCurtentValue

Every time the value of a Calculator changes, we want to log the calculator value to a logfile. We have the following Logger class:

```
public class Logger{
   public void log(int calculatorvalue) {
      System.out.println(calculatorvalue);
   }
}
```

Now we want to apply the observer pattern to this code so that the Logger becomes an observer of the Calculator. Write the code of the Logger if we use the **pull** version of the observer pattern. If your Logger implements an interface of extends from an abstract class, then also write the Java code of this interface or abstract class.

```
public interface Observer {
     public void update();
}


public class Logger implements Observer{
     private Calculator calculator;

    public Logger(Calculator calculator) {
         this.calculator=calculator;
     }

     public void log(int calculatorvalue) {
        System.out.println(calculatorvalue);
    }

    public void update() {
     log(calculator.getCurrentValue());
    };
}
```

[10 minutes]

Similar as in the previous question we have the following calculator class:

class Calculator
Attribute: currentvalue
Methods:
void add(int value)
void subtract(int value)
int getCurtentValue

Every time the value of a Calculator changes, we want to log the calculator value to a logfile. We have the following Logger class:

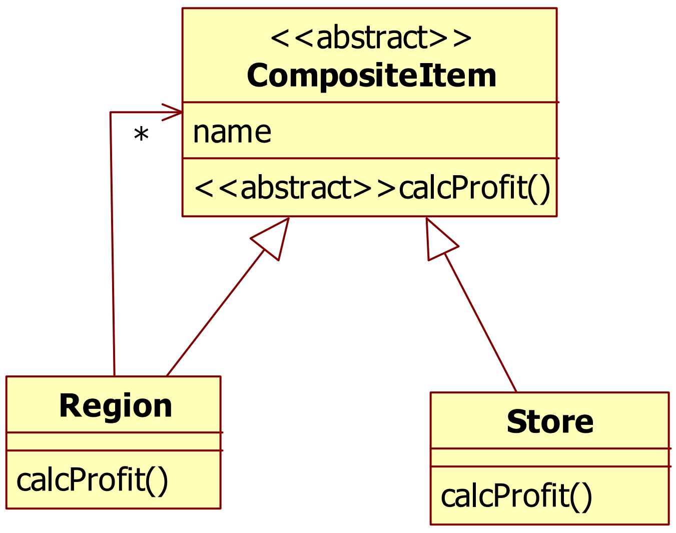
```
public class Logger{
   public void log(int calculatorvalue) {
      System.out.println(calculatorvalue);
   }
}
```

Now we want to apply the observer pattern to this code so that the Logger becomes an observer of the Calculator. Write the code of the Logger if we use the **push** version of the observer pattern. If your Logger implements an interface of extends from an abstract class, then also write the Java code of this interface or abstract class.

```
public interface Observer {
    public void update(int calculatorvalue);
}


public class Logger implements Observer{

    public void log(int calculatorvalue) {
       System.out.println(calculatorvalue);
    }

   public void update(int calculatorvalue) {
    log(calculatorvalue);
   }
}
```

[20 minutes]

Suppose you have to design an application that computes the profit for a retailer. The retailer has stores worldwide. In some big cities this retailer can have multiple stores. The application should allow us to compute the profit of every single store, the profit of all stores in a certain city, the profit of all stores in a certain state, the profit of all stores in a certain region and the profit of all stores in a certain country. A region can be anything, like West Chicago, Iowa and Ohio, East Europe, Europe and Asia, Boston, California, etc. With StarUML draw the class diagram of all domain classes (and/or interfaces) you need to design this. For every domain class show its attributes and methods. If a class is abstract or is an interface, then make this clear in your answer. If you use inheritance or implement an interface, then make this clear in you answer.

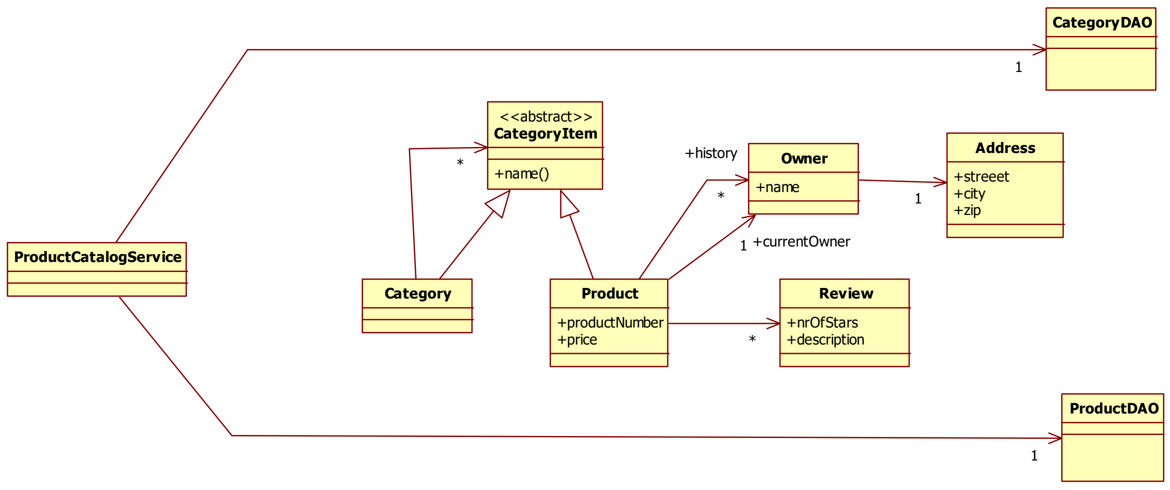


[30 minutes]

Draw the class diagram in StarUML of a product catalog application with the following requirements:

- You can add and delete different product categories (like electronics, books, cars)
- For every category we store its name.
- A product category can have subcategories (and subcategories can also have subcategories, etc.)
- You can add and delete products in/from categories
- For every product we store its productnumber, name and price.
- You can add customer reviews to products
- You can view all reviews for a certain product
- A review consists of a description and a certain integer number between 1 and 5 that shows how much stars a product can get from a customer. 1 star means that the customer was not very happy with the product and 5 stars means the customer was very happy with the product.
- All the products in this product catalog have a certain owner
- Because these products change from owner very quickly, we can see the history of all previous owners of a certain product
- All data is stored in a database.

Your design should implement the appropriate design principles we studied in the course.


CategoryDAO
1
<<abstract>>
CategoryItem
+name()
*
+history
Owner
+name
*
Address
+streeet
+city
+zip
1
1 +currentOwner
ProductCatalogService
Category
Product
+productNumber
+price
Review
+nrOfStars
+description
*
ProductDAO
1