Final Exam (Take Home)

CS 414 Object-Oriented Design, Fall 2012

100 points

Due via RamCT: 11:59 PM MST, Wednesday, December 12, 2012

Honor Pledge

"I have not given, received, or used any unauthorized assistance."

(TYPE YOUR NAME HERE AS A PROXY FOR YOUR SIGNATURE)

Instructions

- Copy and paste the Honor Pledge above into your document and type your name as a proxy for your signature. Note that not signing the honor pledge will not be considered as evidence that a student has committed academic misconduct. Students whose religious tenets prohibit taking oaths will not be expected to sign the pledge. More information on the honor pledge can be found here (`http://tilt.colostate.edu/integrity/honorpledge/howDoCSU.cfm`).
- Include your name on each page of your answers.
- Questions must be answered in numerical order.
- Answers and code must be typed. Diagrams may be hand drawn and scanned as long as they are legible.
- Answers should be succinct, coherent English prose. Make only the main points with necessary support.
- Keep your answers as specific as possible. Avoid generalities.
- The exam must be turned in as a single document (PDF only). Code and diagrams must be included in the same document as part of the question they belong to (not in a different place). Separate code or diagram files will not be accepted.
- This exam is take-home, due at 11:59 PM MST, Wednesday, December 12, 2012. Late submissions are not allowed.
- Do not post comments and questions about the exam to the course discussion board. Send questions directly to the instructor via email (`cs414@cs.colostate.edu`).
- You may use your notes, books, and available articles, but may not consult with other people, except the instructor.
- You must cite your sources properly. Any verbatim quotations must be enclosed in quotation marks, with page numbers indicated. You will receive severe point deductions for using material from the text or other sources without proper citation.
- See the CS department student information guide for guidelines on legitimate and illegitimate consultation.

- Submit your document via RamCT.

Answer all parts of all five (5) questions.

1. (20 points) **Use case diagram, use case description, activity diagram**

   We have provided a high level description of an *Online Medical Appointment Scheduling System*. One overall user goal is to enable a registered patient to schedule a medical appointment. To achieve this goal, a registered patient may need to browse available time slots by medical services or by doctor. Each medical service has a list of available dates and times, and description and duration of service, and the doctors available to perform the service. Each doctor has a name, description of specialty, and list of available times.

   To schedule an appointment, a registered user may log in, browse for services, or browse through doctors, and select one doctor, date and time. Alternatively, the patient may browse first and then log in to select the doctor and time.

   Other uses of the systems include the ability to modify an appointment to change the service and/or the doctor, change the date and time, and cancel the appointment.

   Doctors can view their own appointment schedules. They can also provide their availability information to the system.

   (a) (8 points) Draw a single activity diagram showing the workflow(s) to achieve the **overall goal** of scheduling an appointment. The workflow is likely to include several use cases.

   (b) (4 points) Draw a use case diagram showing five use cases for the above system. There must be at least one use case involving the *Registered Patient* as an actor, and at least one involving the *Doctor* as an actor.

   (c) (8 points) Describe in detail the following two use cases. Mention the pre-condition for each use case. Do not include the steps for logging in as part of the actor-system interactions of the use cases. Be specific about the types of information provided or obtained in each step. Include one main scenario and one alternative scenario for each use case.

      i. Modify Appointment (actor: Registered Patient)

      ii. View Appointment Schedule (actor: Doctor)

2. (35 points) **Refactoring based on State Pattern**

   Consider the class `RentalStateMachine` that implements the states and transitions of a rental object, such as a rental car or a rental video. The state machine has four states: `ForRent`, `Rented`, `ForSale`, and `Sold`. The public methods `rent`, `returnRental`, `setForSale` and `sell` show what happens when corresponding events occur on the rental object. As the implementations of the methods show, each event can either change the current state of the rental, or throw an `IllegalEventException` if that event is not expected in the current state.

```
public class RentalStateMachine {

    public enum RentalState {
        ForRent, Rented, ForSale, Sold;
    }

    private RentalState currentState;

    public void rent() throws IllegalEventException {
        if(currentState == RentalState.ForRent) {
            currentState = RentalState.Rented;
        } else {
            throw new IllegalEventException();
```

```
        }
    }

    public void returnRental() throws IllegalEventException {
        if(currentState == RentalState.Rented) {
            currentState = RentalState.ForRent;
        } else {
            throw new IllegalEventException();
        }
    }

    public void setForSale() throws IllegalEventException {
        if(currentState == RentalState.ForRent) {
            currentState = RentalState.ForSale;
        } else {
            throw new IllegalEventException();
        }
    }

    public void sell() throws IllegalEventException {
        if(currentState == RentalState.ForSale) {
            currentState = RentalState.Sold;
        } else {
            throw new IllegalEventException();
        }
    }
}
```

(a) (10 points) Refactor the `RentalStateMachine` class by using the **State Pattern** to decouple the states from the state machine. The refactored `RentalStateMachine` should not have any conditional statement that checks the present state. Name and describe each refactoring step, and show the final refactored code.

(b) (10 points) Extend the refactored code according to the directions below and show the resulting code:

- Implement code for a new state called `Withdrawn`.
- The rental state machine may go to the `Withdrawn` state if the rental state is not sold. To implement this behavior, add code to implement transitions from each applicable state to `Withdrawn`.
- Once the rental is in a `Withdrawn` state, it will always remain there irrespective of the event. Add code to implement the transitions from `Withdrawn` to itself for all events.

(c) (10 points) Repeat the activities in part (b), but this time with the original code that was provided by us (i.e., the version without the state pattern) and show the resulting code.

(d) (5 points) Provide the number of changes (e.g., lines and methods) that you made for parts (b) and (c). What aspects of the extensions were easier or more difficult to implement after applying the state pattern?

3. (20 points) **Refactoring and Sequence Diagram**

Consider the Java classes `Node`, `Expression`, `Addition`, `Subtraction`, `Negate`, `Token`, and `Evaluator` below.

Java Code:

```
*** File Node.java ***
public abstract class Node {
```

```
}

*** File Expression.java ***
public abstract class Expression extends Node {
    private Node left;
    private Node right;

    public Node getLeft(){
        return left;
    }
    public Node getRight(){
        return right;
    }
}

*** File Addition.java ***
public class Addition extends Expression {
}

*** File Subtraction.java ***
public class Subtraction extends Expression{
}

*** File Negate.java ***
public class Negate extends Expression {
}

*** File Token.java ***
public class Token extends Node {
    public static final int IDENTIFIER = 1;
    public static final int INTEGER_LITERAL = 2;
    private int type;
    private String text;

    public String getText(){
        return text;
    }

    public int getType(){
        return type;
    }
}


*** File Evaluator.java ***
import java.util.*;
public class Evaluator {
    private HashMap<String, Integer> IdentifierToValue;

    public int evaluate(Node node){
        if(node instanceof Token) {
           Token token = (Token)node;
```

```
            int type = token.getType();
            if(type == Token.IDENTIFIER)
                return IdentifierToValue.get(token.getText()).intValue();
            else
                return Integer.parseInt(token.getText());
        } else if (node instanceof Addition) {
            Addition addition = (Addition)node;
            int leftOperand = evaluate(addition.getLeft());
            int rightOperand = evaluate(addition.getRight());
            return leftOperand+rightOperand;
        } else if  (node instanceof Subtraction) {
            Subtraction subtraction = (Subtraction)node;
            int leftOperand = evaluate(subtraction.getLeft());
            int rightOperand = evaluate(subtraction.getRight());
            return leftOperand-rightOperand;
         } else {
            Negate negate = (Negate)node;
            int leftOperand = evaluate(negate.getLeft());
            return (-leftOperand);
         }
    }
}
```

(a) (12 points) Refactor the `Evaluator` class by applying the "*Replace Conditional with Polymorphism*" refactoring from the Fowler text. You may need to modify other classes to complete the task. The final code in your refactored `Evaluator.evaluate()` must not have any conditional statements at all. Do not introduce any new conditional statements anywhere in the code.

i. Show the refactored code.

ii. List every step that you performed to apply the refactoring.

(b) (8 points) Draw a sequence diagram to show the interactions that take place in your refactored code when the `evaluate` operation is called on an instance of `Evaluator` with the argument `a:Addition` shown in Figure 1:

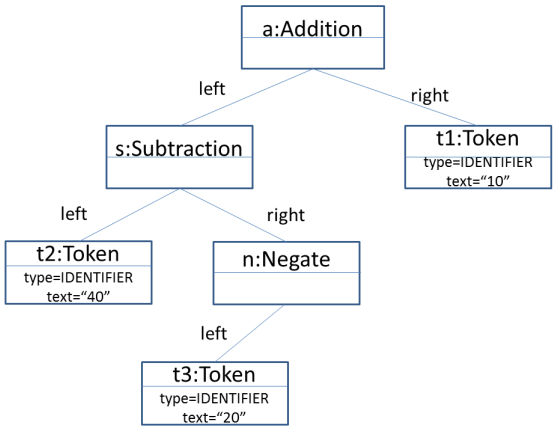
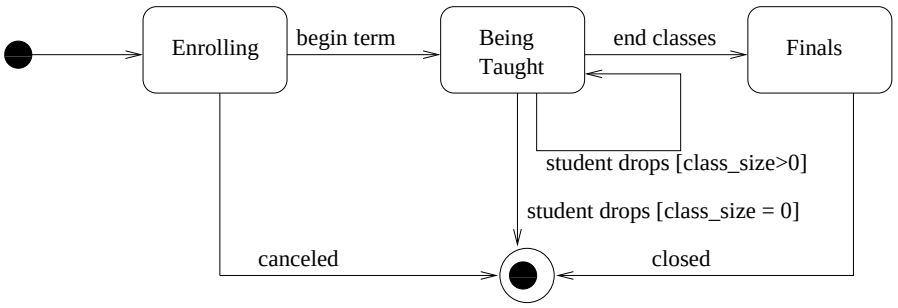


Figure 1: Object Diagram Showing Expression to be Evaluated.

4. (9 points)



The above state diagram shows the lifecycle of a class offered at our university. Your task is to extend the model in the following ways:

- Expand the `Enrolling` state (i.e., create nested states and transitions). During enrollment, a course is scheduled and made available for enrollment. Students can add themselves to the course. Eventually the course may become full, but students may drop again and others may add themselves at that point. Assume that there is a date before teaching begins when the enrollment is closed.
- The current model shows that students can drop any time while the course is being taught. Set up the model to incorporate a drop date, after which no one can drop the course.
- Allow for incomplete grades. That is, some students may get an incomplete grade, for which they must complete the requirements within a fixed period of time after the finals. Thus, the class may not be closed after the finals in case there are students with incomplete grades.

Create one diagram to answer all the parts. You will need to show nested states. Remember to show all the necessary states, transitions, events, and guards.

5. (16 points) Below is a toy program adapted from Larman's text. The program takes the quantity and ID of an item and displays the price. The `SaleFrame` class is a UI layer class, while all the other classes are domain layer classes. Observe that the `SaleFrame` class is directly coupled with domain layer classes and is performing some domain logic inside the `actionPerformed` method. Moreover, the domain class `PriceCalculator` keeps a direct reference to `SaleFrame` and it is directly performing the UI update. These are violations of the principle of Model-View separation.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.HashMap;

public class SaleFrame extends JFrame implements ActionListener {

    private JButton enterItemButton = new JButton("Enter Item");
    private JTextField priceField = new JTextField(30);
    private JTextField itemIDField = new JTextField(10);
    private JTextField quantityField = new JTextField(10);

    public SaleFrame() {
        Container container = getContentPane();
        container.setLayout(new FlowLayout(FlowLayout.CENTER));
        container.add(new JLabel("Enter Item ID(1/2)"));
```

```
        container.add(itemIDField);
        container.add(new JLabel("Enter quantity of the item"));
        container.add(quantityField);
        container.add(enterItemButton);
        container.add(new JLabel("The total price of the item"));
        container.add(priceField);
        priceField.setEditable(false);
        enterItemButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        ProductCatalog productCatalog = ProductCatalog.getInstance();
        int itemID = Integer.parseInt(itemIDField.getText());
        ProductDescription description =
            productCatalog.getProductDescription(itemID);
        PriceCalculator calculator = new PriceCalculator();
        int quantity = Integer.parseInt(quantityField.getText());
        calculator.calculatePrice(quantity, description, this);
    }

    public void displayPrice(int price) {
        priceField.setText("The price for the entered item is "+ price);
    }

    public static void main(String [] args) {
       SaleFrame saleframe = new SaleFrame();
       saleframe.setSize(200, 200);
       saleframe.setVisible(true);
    }
}

public class ProductCatalog {

    private HashMap<Integer, ProductDescription> itemIDtoProductDescriptionMap
        = new HashMap<Integer, ProductDescription>();

    protected ProductCatalog() {
        super();
        itemIDtoProductDescriptionMap.put(1, new ProductDescription(10));
        itemIDtoProductDescriptionMap.put(2, new ProductDescription(15));
    }

    private static ProductCatalog INSTANCE;

    public static ProductCatalog getInstance() {
        if (INSTANCE != null)
            return INSTANCE;
        else
            return new ProductCatalog();
    }

    public ProductDescription getProductDescription(int itemID) {
```

```
        return  itemIDtoProductDescriptionMap.get(itemID);
    }
}

public class ProductDescription {
    private int unitPrice;
    public ProductDescription(int unitPrice) {
        this.unitPrice = unitPrice;
    }

    public int getUnitPrice() {
        return unitPrice;
    }
}

public class PriceCalculator {
    public void calculatePrice(int quantity, ProductDescription description,
        SaleFrame frame) {
        int price = quantity*description.getUnitPrice();
        frame.displayPrice(price);
    }
}
```

Show the resulting code after **each** of the following steps:

(a) (8 points) Introduce a controller between the UI and the domain classes in the program above so that the `SaleFrame` interacts only with the controller to perform the `enterItem`.

(b) (8 points) Then use the Observer pattern to update the UI whenever the item price is calculated.