In-class midterm practice

Practice problem 1

Turning requirements into an OOD solution

Review problem2 from the **Spring 2019 (https://northeastern.instructure.com/courses/65531/files/7282991?wrap=1)** midterm then answer the following questions:

1. What is an appropriate overall design? i.e. will you need interfaces, abstract classes, concrete classes? What about the question wording gives you clues to the overall design?
2. In which class would you implement each bullet point (parts b and d) and why? What about the question wording gives you clues about where you should implement each bit of logic?
3. How can you break up all the required calculations into helper methods?

If you get done early, start implementing a rough solution.

Practice problem 2

Refactor non-OOD code

Your colleague has written part of an airline ticketing system that will calculate the price of a particular seat for a particular passenger. Unfortunately, their code is functionally correct but it does not adhere to object-oriented design principles and your manager has asked you to refactor it.

You'll need to review the code to see what it does but here is an overview:

- There is an Airplane class that constructs a new plane with a specified number of seats, of which a certain number are business class and a certain number are economy plus class. The remaining seats are all economy. All airplanes have 4 seats per row. Your manager wants to keep this functionality but the code needs to be cleaned up.
- There is a Passenger class, representing the person that wants to buy the seat. Each passenger has a status with the airline (non-member, member, or elite). Elite passengers get a 10% discount on any seat. Your manager is happy with how passengers are represented.
- Each seat should have a class (economy, economy plus, or business). Business class seats are at the front, after that are the economy plus seats, with the economy seats at the back. Different seat classes have different prices, which you can see in the code. This part of the code needs a lot of refactoring.
- The Airplane class has a method, calculateSeatPrice(), that takes a row number, and a seat number, and a particular Passenger. The method returns the seat price based on the seat's

class and the Passenger's status. Although the method is technically correct, it is not object-oriented. Fix this method.

What your manager is looking for:

- Good OOD principles, especially encapsulation and inheritance.
- Appropriate use of interfaces, abstract classes, concrete classes and enums.
- Simple methods – one method, one piece of functionality.
- As few conditional blocks (if..else and switch) as possible. Currently there are 4 separate conditional blocks in the code. See if you can reduce that number to 2 or fewer.

The code you need to refactor is in the lecture-code repo > airlinerbad package.

Practice Problem 3

Design and implement an OOD solution from scratch

You are developing an invoicing system for a company that provides event management services (e.g. setting up parties, performances, graduations). You are expected to write the part of the system that will automatically calculate the number of hours needed to setup and run an event based on its type and scale.

The system calculates the number of hours needed for the following types of event:

- Inside – the event take place in a building.
- Outside – the event will take place outside.

Inside events can be one of the following:

- Performance
- Party

Outside events can be one of the following:

- Sporting events
- Graduation ceremony

The system keeps track of the following information for each event:

- ID – represented as a String
- Location – represented as a String
- Category – represented as an enum. The category can be *private*, *ticketed*, or *open* to the public.
- A Boolean that indicates whether or not this is a recurring event.
- The expected number of attendees.

Additionally, for every outside event we keep track of the following:

- The month, encoded as an integer, in which the event will take place.

For every inside event, we also track:

- Location capacity – the number of people the event location can fit.

The system calculates the number of hours needed to set up and run an event according to the following rules:

- For all events, the company plans for a minimum of 1 hour of setup time for every 100 attendees.
- For all events that are private or ticketed, the company needs an additional half hour for every 50 attendees to setup guest list/ticket verification.
- The event length (not including setup time) is fixed at 3 hours for performances, 5 hours for parties, 4 hours for sporting events, and 6 hours for graduation ceremonies.
- The time taken to tear down and clean up after an event is calculated as double the time taken to set up the event.
- The company is only able to support events with a maximum of 1000 attendees. If an event is created with more than 1000 attendees, throw a TooManyAttendees exception.
- If an inside event is created with more attendees than the location capacity, throw a TooManyAttendees exception.