Software Design Patterns - PR1 - Solution - Grade A: 47/50

Análisis y diseño con patrones (Universitat Oberta de Catalunya)

Question 1

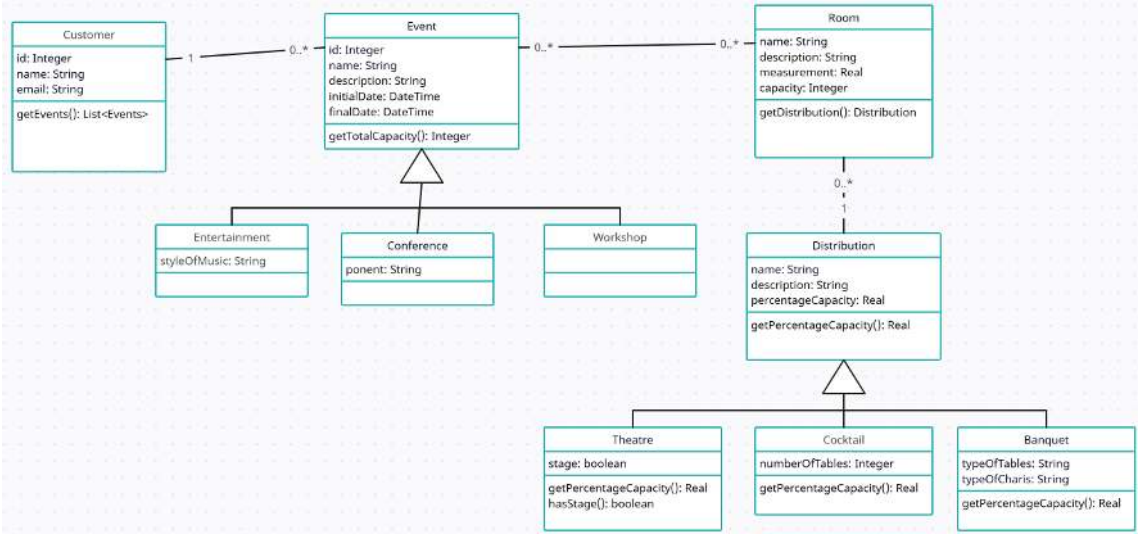
a) It does not satisfy Demeter's law. Demeter's Law says "Don't talk to strangers", so you should only interact with your closest objects. The Event class is directly accessing the methods of the Room class and the Distribution class. What we could call the problematic line is Distribution d = r.getDistribution(); because the Event class is getting the distribution of the room (Distribution) and then doing a verification of the distribution type (Theatre). This check implies direct knowledge of the internal implementation of the Distribution class and its subclasses, which goes against Demeter's Law.

b) This design does not comply with the DRY principle because in the pseudocode we can see these lines of code twice, both in the if and the else:
Real percentCapacity = d.getPercentageCapacity()/100;
capacity = capacity + (roomCapacity * percentCapacity);
Then we affirm that it is not satisfied.

c) It does not satisfy the principle, precisely because of the same two lines discussed in the previous section. It is open to extension but not completely closed to modification, to satisfy the principle we should eliminate this duplicity in the logic.

Question 2

a) Class diagram modified:



b) Pseudocode:

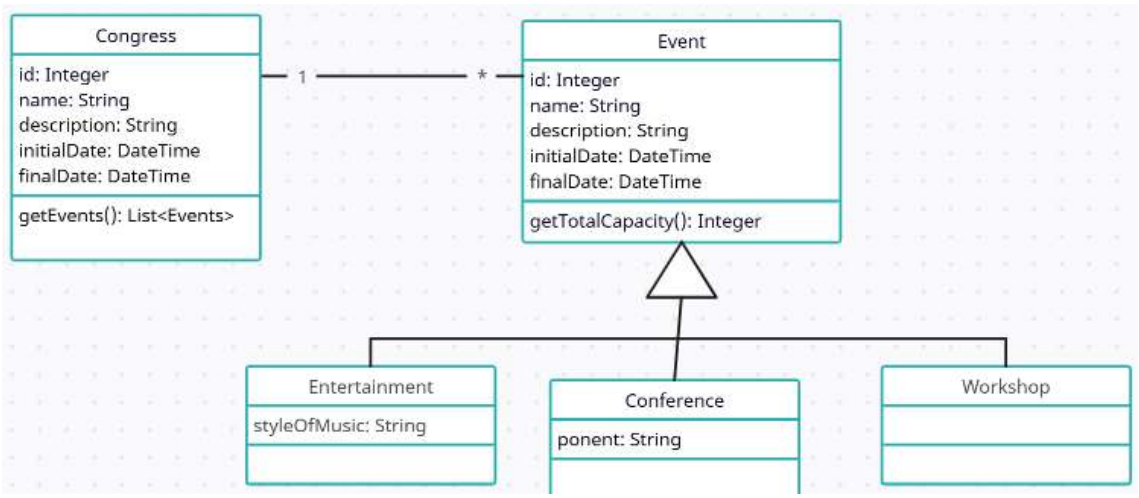
```
public abstract class Event {
        private Integer id;
        private String name;
        private String description;
        private Datetime initialDate;
        private Datetime finalDate;
        private Customer customer;
        private List rooms;
```

```
        public Integer getTotalCapacity() {
            Integer capacity = 0;
            foreach (Room r in rooms) {
                Integer roomCapacity = r.getCapacity();
                Distribution d = r.getDistribution();
                Integer roomTotalCapacity = roomCapacity *
                d.getPercentageCapacity();
                capacity += roomTotalCapacity;
            }
            return capacity;
        }
    }
```

Question 3

a) Composite Object design pattern.
We want to break down complex things into small parts that can be used in the form of objects. In this case, a Congress is composed of several events (e.g. conferences, workshops), and we want to manage them collectively.
The Composite Object design patter fits the hierarchical structure of a congress composed of several sub-events, provides a unified way of accessing both the individual events and the composite structure, and operations can be applied uniformly to both the individual events and the congress itself, simplifying management.

b) Diagram changed:



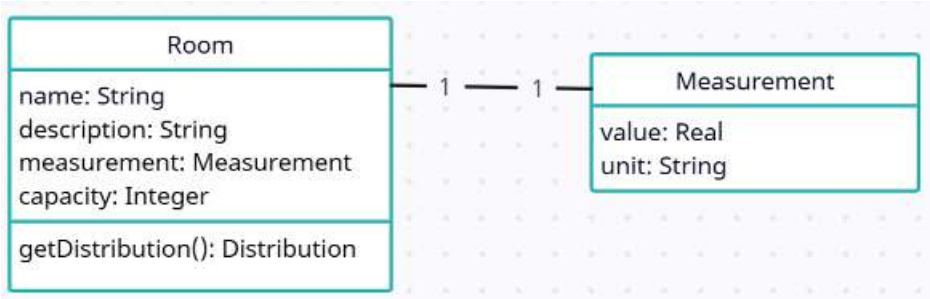
Integrity restrictions:

- The start date of an event must be earlier than its end date.
- The start date of a congress must be earlier than its end date.
- The start date of a congress cannot be later than the start date of an event (belonging to the congress).
- The end date of a congress cannot be earlier than the end date of an event (pertaining to the congress).

Question 4

a) I would apply the Quantity analysis pattern because it is used when a quantity is required to be recorded. The representation of a quantity by a numerical value is unsuitable, so we model the measure as a more complex data type consisting of the fields value and unit.

b) Diagram modification



Integrity restrictions:

- A measurement value must always have an associated unit.
- All integers and reals are positive numbers.
- For this case: The unit of measurement must be "m²" or "ft²".

Question 5

a) MVC

- Model

  EventService:

  getEventDetails(eventId: Integer): Event

  It returns all Event details by its id.

  updateEvent(eventId: Integer): void

  Update Event details changed by the user in the data base.

- View

  Event:

  showEventDetails(event: Event): void

  It shows the details of the Event in the screen for the user.

  printSuccessMessage(): void

  It returns “Successfully registered” message.

- Controller

  EventController

  loadEventDetails(eventId: Integer): void

  It calls the service to obtain the details of the Event and then call the view to show them.

  updateEventDetails(eventId: Integer): void

  It calls the service to update the details of the Event and then call the view to show the success message.

b) Pseudocode:

```
public class EventService {
  public Event getEventDetails(int eventId) {
    // Logic to returns all Event details by its id
```

```
    }

    public void updateEvent(int eventId) {
        // Logic to update Event details changed by the user in the data base
    }
}
public class Event {
    public void showEventDetails(Event event) {
        // Logic to show Event details in the screen to the user
        Console.WriteLine("Details of the event:");
        Console.WriteLine($"ID: { event.Id}");
        Console.WriteLine($"Name: { event.Name}");
        Console.WriteLine($"Description: { event.Description}");
        Console.WriteLine($"Initial Date: { event.InitialDate}");
        Console.WriteLine($"Final Date: { event.FinalDate}");
    }

    public void printSuccessMessage() {
        Console.WriteLine("Successfully registered");
    }
}
public class EventController {
    private EventService eventService;
    private Event event;

    public EventController(EventService eventService, Event event) {
        this.eventService = eventService;
        this.event = event;
    }

    public void loadEventDetails(int eventId) {
        Event eventDetails = eventService.getEventDetails(eventId);
        event.showEventDetails (eventDetails);
    }

    public void updateEventDetails(int eventId) {
        // Update the Event with the changes of the user
        eventService.updateEvent(eventId);

        // Show success message
        event.printSuccessMessage();
    }
}
```

Add a class User to interact with Controller:

```
public class User {
   public void updateEventDetails (EventController eventController, int eventId) {
      // Simulates the interaction with the graphic user interface
      eventController.loadEventDetails(eventId);
      eventController.updateEventDetails(eventId);
   }
}
```

Question 6

Question 7

Exercise 2

a) None of them.
b) All classes in exercise 2 belong to the domain layer.
c) None of them.

Exercice 5

a) None of them.
b) EventService and EventController are in the domain layer.
c) Event class is in the presentation layer.

Exercise 6

a) EventDAO class is in the technical services layer.
b) EventServices is in the domain layer.
c) Event class is in the presentation layer.

General explanation:
Technical services layer: This layer contains the classes that are responsible for accessing and managing data, interacting with the storage layer. That is why in exercise 2 and 5 there is no class in the diagrams on which we are based that is responsible for accessing and managing the data as itself.
Presentation layer: this is the layer where the classes that interact with the user and display information are found. As the explanation says there is an interaction with the user, and this is not the case in exercise 2, but as it is explained in the other exercises it is.
Domain layer: This is the layer that deals more with logic, they do the management. In all the exercises we find classes that are in this layer, since in all the cases information is managed and they have certain logic to carry out what is requested in the exercise.