Software Design Patterns

Principles, Analysis and Architectural Patterns

Exercise 1

We are hired to develop a software to connect clients with moving companies. In order to request a move, a customer must enter their ID, name, surname, email, and they will automatically subscribe to receive offers. If the user wishes, they can unsubscribe at any time by requesting it. In this case, the reason why you want to unsubscribe will be recorded.

Clients can request a single-day or multi-day move, depending on their needs. The move starts from a place of origin to a destination place, which must have the following characteristics: address, floor and if they have an elevator. Customers must mark all the types of objects (items) they want to transport and if they need assembly or not. The move will be carried out by a single operator of the contracted company. Each company registers its taxId number, name, price per km and commission that the operator will charge.

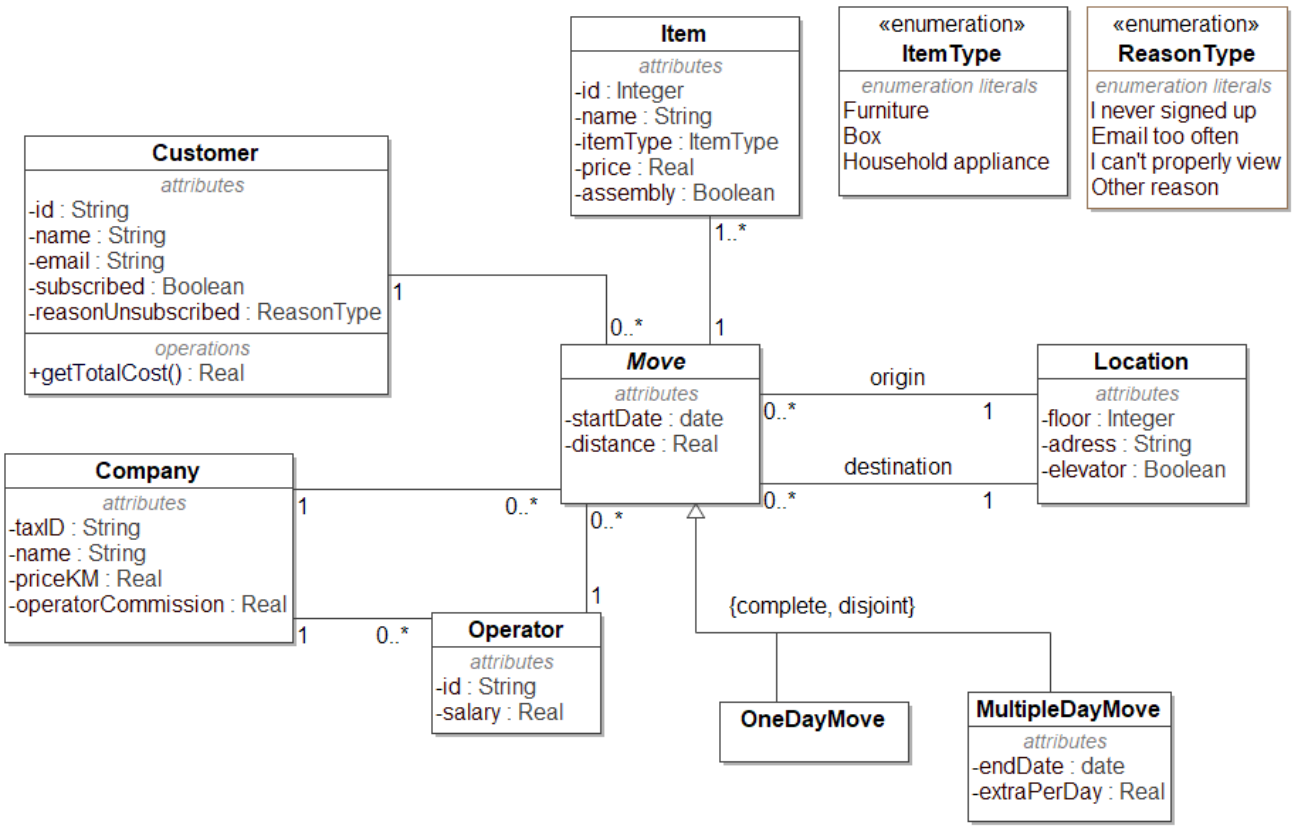
The price of the move will depend on the distance between the origin and the destination (given by the price per km that the company establishes), the price of transporting each object (which is given by the price attribute of the object and which is already defined by the system) and the extra charge per day if the move takes several days.

For example, if a client requests a two-day move from Barcelona to Sant Cugat - a distance of 17.5 km - and wants to transport 10 pieces of furniture, 2 electrical appliances and 50 boxes, the price of this move will be calculated as the sum of: 17.5 kilometers times the price per km of the company, the price of each object and one day for the extra per day (the first day is not charged).

This software is a part of a larger system. We will leave out many elements that a real system would contain.

We already have a previous analysis diagram that we can use as a starting point, but that we will have to correct and improve. Below is the class diagram for this part of the software.

Class Diagram:



Keys:

- *Customer: id*
- *Operator: id*
- *Location: address*
- *Company: taxID*
- *Item: id*
- *Move: id(Customer) + originAddress + startDate*

Integrity constraints:

- The start date of a move must be earlier than the ending date of MultipleDayMove.
- The source address cannot be the same as the destination address.
- An operator cannot have overlapping moves.

- All integers and reals are positive numbers.
- The company that makes the move is the same in which the participating operator works.

For marketing purposes, we have been asked to design an operation to calculate the total amount of the move price (getTotalCost()) that a customer has. Here is pseudocode for a possible implementation of this method:

```
public class Customer
{
  private String id;
  private String name;
  private String email;
  private Boolean subscribed;
  private ReasonType reasonUnsubscribed;
  private List<Move> moves;

  public Real getTotalCost()
  {
    Real total = 0;
    foreach (Move m in moves)
    {
      List<Item> items = m.getItems();
      foreach (Item i in items)
      {
            total += i.getPrice();
      }
      Company c = m.getCompany();
      total += m.getDistance() * c.getPriceKM();

      if(typeof(m)==MultipleDayMove) {
            Integer extraDays = m.getEndDate() - m.getStartDate();
            total += extraDays * m.getExtraPerDay();
      }
    }
    return whole;
  }
}


public class Item
{
      private Integer id;
      private String name;
      private ItemType itemType;
      private Real price;
```

```
	private Boolean assembly

	public Real getPrice()
	{
		return price;
	}

}


public class Company
{
	private String taxID;
	private String name;
	private Real priceKM;
	private Real operatorCommission;
	private List<Operator> operators;
	private List<Move> moves;

	public Real getPriceKM()
	{
		return priceKM;
	}

}
```

Exercise 1.1

We reviewed the pseudocode for the Customer's class getTotalCost() operation and are concerned that some of the design principles may not be met. Specifically, you need to:

a) Indicate whether or not this design satisfies the Law of Demeter and justify your answer.
b) Indicate whether or not this design satisfies the Open-Closed design principle and give reasons for your answer.
c) Indicate whether or not this design satisfies the Low Coupling design principle and give reasons for your answer.

Exercise 1.2

Propose a complete alternative solution (also in pseudocode of the same style) that corrects the problems with the Customer's getTotalCost() operation that you have detected in the previous

exercise and that, therefore, complies with the violated principles identified in exercise 1.1. Modify the class diagram by adding all the operations of your solution.

Exercise 1.3

Recently, moving companies have realized that they have to change the salary of the workers more than they would like depending on the season. That is why it has been decided to record the salaries of the operators over time. In this way the operators will be able to see the record of their salary in different periods of time.

They propose that we find a solution that allows us to represent this information in our system. Specifically, we need to:

a) Briefly describe how you would solve this problem. If you propose the application of a pattern, explain why and justify it.
b) Propose a detailed solution in the form of a static analysis diagram. Shows only the classes that change with respect to the statement diagram and the integrity constraints that appear.
c) Indicate the resulting pseudocode of the higherSalaryOperator(): String operation in the Company class. This operation returns the id of the operator with the highest salary in the current month. The monthly salary of an operator is calculated as his salary plus the commission (operatorCommission) for each move in which he has participated during the month.

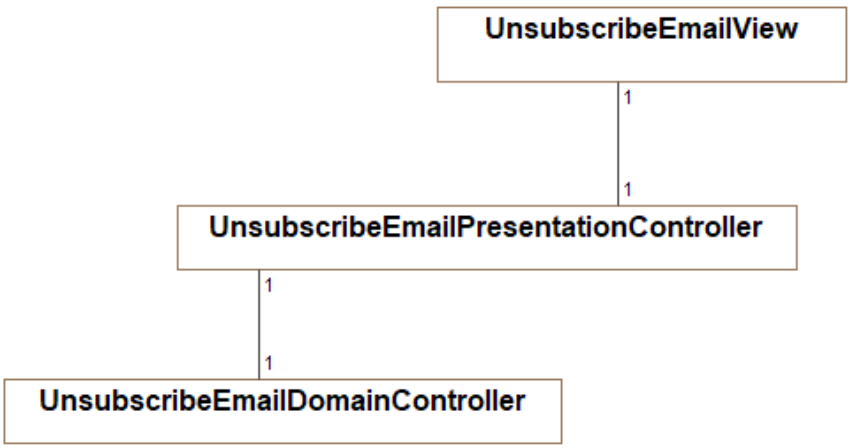
   Note: moves of more than one day are counted in the month in which they start. For example, a move that covers from April 30 to May 2 is counted as made in April.
   Nota2: It is not necessary to consider the solution or assumptions made in points a) and b) to solve point c).

Exercise 1.4

We want to define the presentation layer of our system and for this we apply the Model, View and Controller (MVC) architecture pattern. We propose an initial (not finished) static analysis diagram where the MVC pattern is applied for the Email Unsubscribe use case. This use case allows the user to indicate the option to unsubscribe from their email so as not to receive advertising from the

company. In response, the system will show you all possible reasons for unsubscribing. The user will select the reason why they want to unsubscribe and the system will unsubscribe and display the message "Unsubscribed successfully".
The diagram has all the necessary classes. You need to explain what each class represents (in the MVC pattern) and define the signature of the operations from when the user sees a control panel with all the actions that they can do, in which one of them is the email unsubscription option, until that the system unsubscribes the email showing the message "Unsubscribed successfully".

For each operation, you must specify the class they belong to, the parameters, what it returns, and explain what it does.

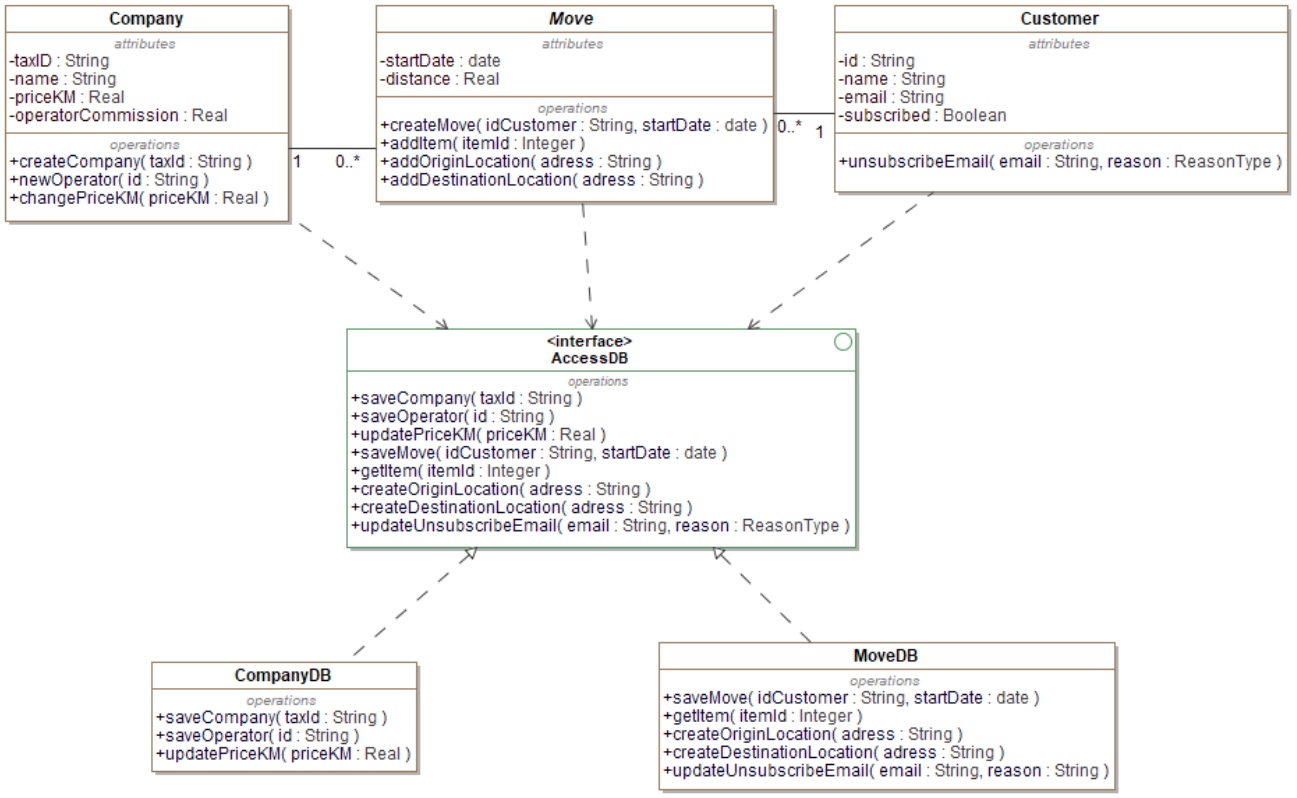


Exercise 1.5

Propose the sequence diagram for the interaction from the user to the domain controller for the use case Mail unsubscription, which ends up showing the message "Unsubscribed successfully".

Exercise 1.6

We continue designing the architecture of the system and we have decided to use an architecture with 3 layers: presentation, domain and technical services. We find that the company has two databases that they do not want to merge. One is for the companies and operators and the other one for customers and removals. We have made a first design of how the classes we already have could communicate with the databases:



What design principle is it violating? Justify your answer. Propose a static analysis diagram that solves the violation of the principle.

Exercise 1.7

To finish the system design architecture, we have already decided to use an architecture with 3 layers: presentation, domain and technical services. We will use the following three classic layers of many information systems:

1. Presentation
2. Domain
3. Technical services

We want to reflect on to which layer belong each of the classes that have been presented before (and on the solution you have developed so far):

- Classes in the first diagram: Item, Move, OneDayMove, MultipleDayMove, Location, Customer, Operator, and Company
- Classes from the second diagram (Exercise 4): UnsubscribeEmailView, UnusbscribeEmailPresentationController, UnsubscribeEmailDomainController.

- Classes from the third diagram (Exercise 6): AccessDB (interface), CompanyDB, MoveDB
- Possible classes and/or interfaces introduced/changed in your solution from Exercise 6.

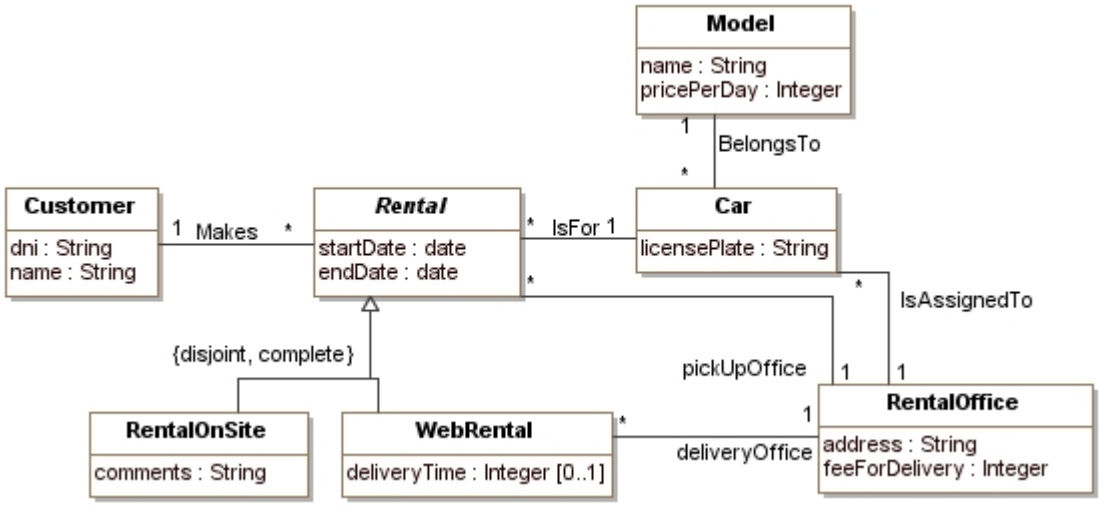
a) Indicate which classes (and interfaces, if any) belong to the domain layer. Justify your response.
b) Indicates which classes (and interfaces, if any) belong to the technical services layer. Justify your response.

Exercise 2

A company has hired us to develop a software to manage car rentals. Customers can rent cars of different models for a period of time. The cars have to be collected at a pick up office. Rentals can be made directly at the company's offices or through the web application that the company makes available to its customers. For rentals made through the offices, comments on the rental can be recorded, if necessary. This type of rental requires customers to return the car to the same office where it was picked up. For rentals made via the website, the return time is recorded. In these types of rentals, customers can return the cars at a different office from the one where they were picked up, paying a delivery charge.

We already have a previous analysis that we can use as a starting point, but that will have to be corrected and improved. Next, you have the class diagram for this part of the software:

Class diagram Class



Keys:

- *Customer: dni*
- *Car: licensePlate*

- *RentalOffice: address*
- *Model: name*
- *Rental: dni (Customer) + startDate*

Integrity restrictions:

- A client cannot have overlapping rentals.
- The start date of a rental must be before the end date of the rental.
- The pick-up office for a rental car has to be the same as the office where the rental car is assigned.
- If the collection and delivery office of an online rental are different, the time of delivery of the rental car must be before 1 p.m.

To carry out marketing tasks, we have been asked to design an operation to calculate the total sum of the charges (*feeForDelivery*) that a client has paid for the rentals he has made through the web, when he has delivered the car in a different office from the one of pickup. In the following, you have the pseudocode that implements this method:

```
public class Customer
{
  private String dni;
  private String name;
  private List<Rental> rentals;

  public Integer getTotalCharges()
  {
    Integer total = 0;
    foreach (Rental r in rentals)
    {
      if(typeof(r)==WebRental) {
            RentalOffice pickUpOffice = r.getpickUpOffice();
            RentalOffice deliveryOffice = (WebRental)r.getDeliveryOffice();
            if (deliveryOffice != pickUpOffice)
                  total += deliveryOffice.getFeeForDelivery();
      }
    }
    return full;
  }
}
```

Exercise 2.1

We reviewed the pseudocode for the *getTotalCharges()* operation and are concerned that some of the design principles are not being followed. Specifically, you need to:

a) Indicate whether or not this design satisfies the Law of Demeter and justify your answer.
b) Indicate whether or not this design satisfies the Open-Closed design principle and give reasons for your answer.
c) Indicate whether or not this design satisfies the Low Coupling design principle and give reasons for your answer.

Exercise 2.2

Propose a complete workaround (also its pseudocode) that corrects the detected problems with the Customer *getTotalCharges()* operation in the previous exercise, and therefore satisfies any violated principles. Modify the class diagram by adding all the operations you need for your solution.

Exercise 2.3

Recently, the car rental company has realized that there are offices that have assigned many cars that they do not rent and, on the other hand, there are others that could rent more cars than they have assigned. That is why the company has decided to move cars from one office to another based on need and record car assignments over time. In this way the cars can be assigned to different offices in different time periods.

We intend to find a solution that allows us to represent this information in our system. Specifically, you need to:

a. Briefly describe how you would solve this problem. If you propose the application of a pattern, explain which one and justify its convenience.

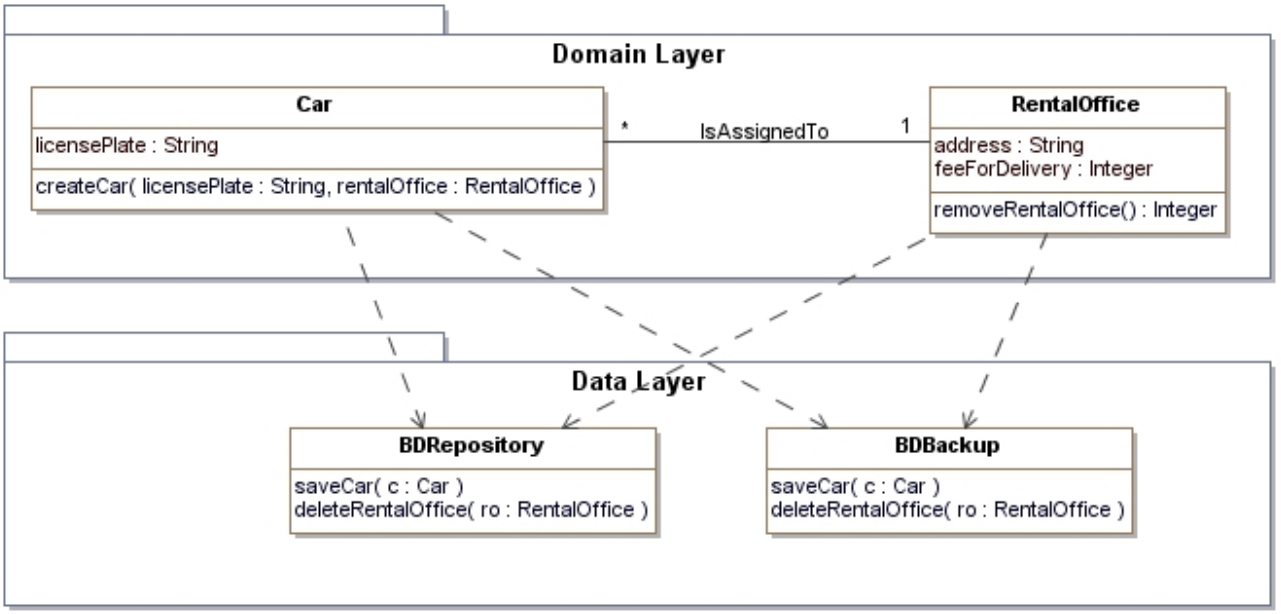
b. Propose a detailed solution in the form of a class diagram. Show only the classes that change with respect to the statement diagram.

c. Show the resulting pseudocode from the *ExpensiveCar():String* operation in class *RentalOffice* which returns the license plate of the car that is currently assigned to the office and that has a higher price per day. If there is more than one, you can return the car you want. You can assume that there will be at least one car currently assigned to the office.

Exercise 2.4

We now start to design the architecture of the system and we have decided to use a 3-layer architecture: presentation, domain and technical services. The car rental company has asked us to always have the data accessible. To satisfy this requirement, we will have a backup database so that when the main database is not available, the invocations are redirected to the database that acts as a backup.

We have a first version of the domain layer and the technical services layer (in our case it will be a data layer that will allow access to the database). In the following figure you have a fragment of these two layers.



Once we have finished making an initial design of these layers, we ask ourselves: does this proposal meet all the design principles that we have seen? If so, justify why you think so. If not, indicate which principles you think it does not comply with and provide the class diagram of the two layers with the changes introduced to comply with them.

Exercise 2.5

We want to define the presentation layer of our system and to do so we apply the Model, View and Controller (MVC) architecture pattern. Propose a class diagram where the MVC is applied for the use case *Remove a car*. This use case allows the user to indicate the option to remove a car. In

response, the system will show all the license plates of the cars it has stored. The user will select the license plate of the car they want to remove, the system will remove it and show the message “Car removed successfully”. For each class in the class diagram, define the signature of the operations used in the presentation layer from the moment the system offers the different options (use cases) and until the user selects the option to remove a car. Explain what each operation does.

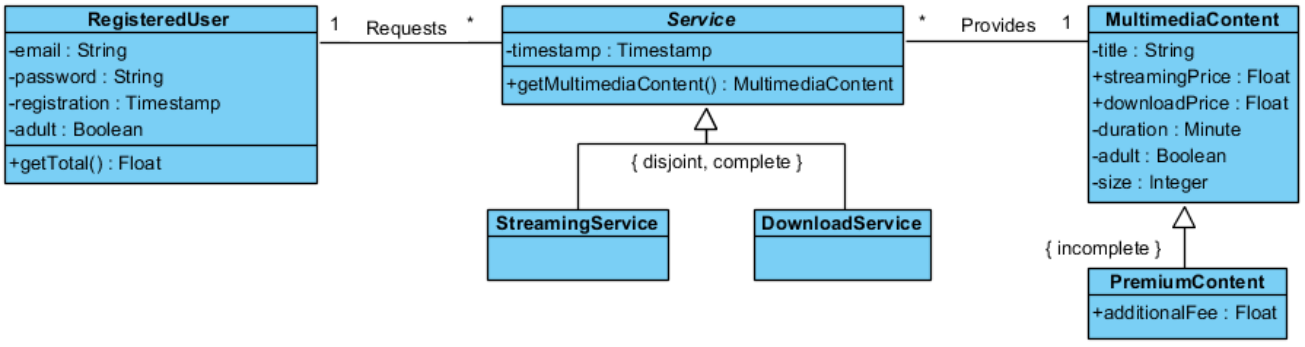
Exercise 2.6

Propose the sequence diagram for the interaction from the user to the domain controller for the use case *Remove a car.* It ends up displaying the message "Car removed successfully".

Exercise 3

We have been hired to develop software to manage the online movies in streaming. The software will be part of a larger system, but it will only have to manage information about the movies offered, the subscribers of the platform and the reproductions and downloads they make on it.

We already have a previous analysis that we can use as a starting point, but that will have to be corrected and improved:



Class keys:

- *RegisteredUser*: *email*
- *Service*: *timestamp*
- *MultimediaContent: title*

Integrity restrictions:

- For any service requested by a registered user, the The date and time of the service must be after the date and time of the user's registration on the platform.
- Adult media content may not be offered to non-adult registered users.

As we can see in the starting diagram, users have an operation that returns the total amount paid for all the services they have requested. The price of a service is calculated as follows:

- For all users who want to watch streaming media content, the streaming price of that media content is applied.
- For all users who want to download multimedia content from the platform, the download price of this multimedia content applies.
- If the content is premium, in any of the above cases the additional charge specified in the *additionalFee* is added.

Suppose the implementation of this method looks like the following pseudocode:

```
public class RegisteredUser

{
  private List<Service> services;

  public float getTotal()
  {
    float total = 0.0;
    foreach (Service s in services)
    {
      MultimediaContent mc = s.getMultimediaContent();
      if(typeof(s)==StreamingService) total += mc.streamingPrice;
      else if (typeof(s)==DownloadService) total += mc.downloadPrice;
      if (typeof(mc)==PremiumContent) total += mc.additionalFee;
    }
    returnfull;
  }
}
```

Exercise 3.1

We review the pseudocode of the operation *getTotal()* in class *RegisteredUser* and we are concerned that its design is fragile since we do not see clearly whether it contemplates possible future scenarios and their impact:

- we change the data type of amounts or the form to calculate an additional charge to our premium content; and/or
- we extend the system to support another type of service (such as physical purchase).

a) Indicate whether or not this design satisfies the Law of Demeter and justify your answer.
b) Indicate whether or not this design satisfies the Open-Closed design principle and give reasons for your answer.

Exercise 3.2

Propose an alternative solution (also in pseudocode of the same style) that corrects the problems with the *getTotal* operation of *RegisteredUser* class that you have detected in the previous exercise and therefore satisfies any violated principles. Make any changes you deem necessary to any of the model classes in the exercise.

Exercise 3.3

As for the prices, we realize that if a content changes its price (for example, streaming), we need the ongoing services (i.e., those provided before the price change) not to change their prices.

We intend to find a solution that allows us to know the price of each content at the time the service request was made as well as at any time in the past.

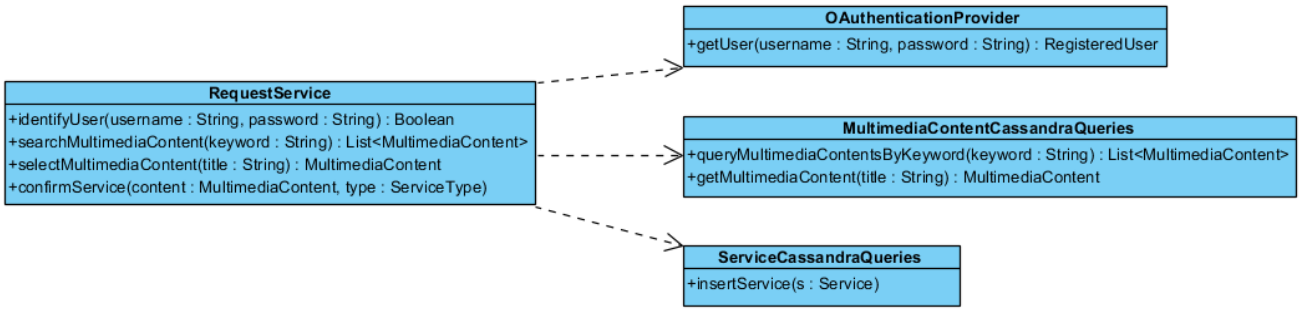
Therefore, instead of having a simple attribute for each price (streaming and download), we would like to know the prices of each multimedia content at any time in the past and suppose that for this we have a new *Price* to represent prices and that it offers a *getValue* to get the value of type float.

a) Briefly describe how you would solve this problem. If you propose the application of a pattern, explain which one and justify its convenience.
b) Propose a detailed solution in the form of a class diagram. Show only the classes that change from the statement diagram.
c) Indicate what the resulting pseudocode would be like for a multimedia content to return the last active streaming price.

Exercise 3.4

We now start to design the software architecture of the system and we have decided to use an architecture with 3 layers: presentation, domain and technical services.

During the design of the first elements, we find ourselves at a point where we have a *RequestService* from the domain layer that allows us to respond, at the logical abstraction level, to the use case for which a registered user authenticates, searches through the available multimedia content, selects one of your choice and requests to stream or download it.



As the different interactions of the use case are carried out, the *RequestService*:

- *identifyUser*: Invoked by a security subsystem that informs which user is using the system. The *RequestService* can fetch user data using *OAuthenticationProvider*. If it is indeed fetched from *OAuthenticationProvider*, the use case can continue.
- *searchMultimediaContent*: Obtains a list of multimedia content available on the platform that meets the search criteria in the title. The *RequestService* can fetch the media content data using *MultimediaContentCassandraQueries*.
- *selectMultimediaContent*: Obtains a multimedia content from the title selection made by the user on the list obtained in the previous operation. Getting the concrete object is done using *MultimediaContentCassandraQueries*.
- *confirmService*: Creates the service from the chosen media content and the desired type of service (streaming or download), and registers it in the database using *ServiceCassandraQueries*. Also, it obtains the binaries of the multimedia content in the optimal format for the player of our platform, if you have selected streaming, or obtains the

downloadable file, if you have selected download; but we do not represent the latter for the sake of simplifying the model.

The *OAuthenticationProvider*, *MultimediaContentCassandraQueries* and *ServiceCassandraQueries*, as you can tell, are from the technical services layer.
Do you think this class meets all the design principles we've seen? If so, justify why you think this is so. If not, indicate which principles you think it does not comply with and indicate what changes you would introduce in the class diagram to comply with them.

Exercise 3.5

In the previous exercise we raised a design issue.

Let's look at a snippet of the *RequestService*:

```
class RequestService
{
	...
	public MultimediaContent selectMultimediaContent (string title)
	{
		// oops! We do not know how to get the
		// MultimediaContentsCassandra instance that we need
		MultimediaContentsCassandra queries = ???;
		MultimediaContent mc = queries.getMultimediaContent(title);
		return mc;
	}
	...
}
```

Looking at the code snippet above, we don't know where to get an instance of *MultimediaContentCassandra* from. We could convert all operations of this class to class-scoped (static) operations and use them directly from *RequestService,* but we are worried that later we may want to support the ElastiCache database as well (and be able to decide on each installation whether to use one or the other database); in such a case we would have *RequestService* coupled

to the *MultimediaContentCassandra* and it would be complicated to be able to decide the database at the time of software configuration.

Propose a solution that allows the decoupling of *RequestService* from Cassandra-specific classes. This solution should be in line with the solution you have proposed for the previous exercise. The solution must also explain how *RequestService* obtains the instance of the class that it will use to consult a multimedia content based on its name, thus resolving the doubt that arose in the pseudocode of the previous exercise.