Analysis and Design with Patterns Design

Principles, analysis and architectural patterns. Solutions.

Solution to Exercise 1

Solution to Exercise 1.1

a) This solution does not satisfy the Law of Demeter, since the Customer class, in the getTotalCost method, uses the Item, MultipleDayMove and Company classes that are not directly associated with it. That is, it obtains instances of Item, MultipleDayMove, and Company that are not associated with it, invokes their methods, and therefore talks to *strangers*. Consequently, any changes to this logic of the Item, MultipleDayMove, or Company class could affect the code in the Customer class.

b) This solution violates the Open-Close design principle, since the Customer class is aware that a subclass (MultipleDayMove) of Move exists and calls an operation of it. If, later on, we extend the system to support a new subclass of Move and the getTotalCost operation would need to call an operation of this new subclass, we should modify the Customer code by adding a new condition to handle this new subclass.

c) This solution violates the principle of Low Coupling. As a consequence of the non-satisfaction of the Law of Demeter, the coupling of this solution is high. The Customer class has knowledge of, and therefore coupling to, among others, the Item, MultipleDayMove, and Company classes.

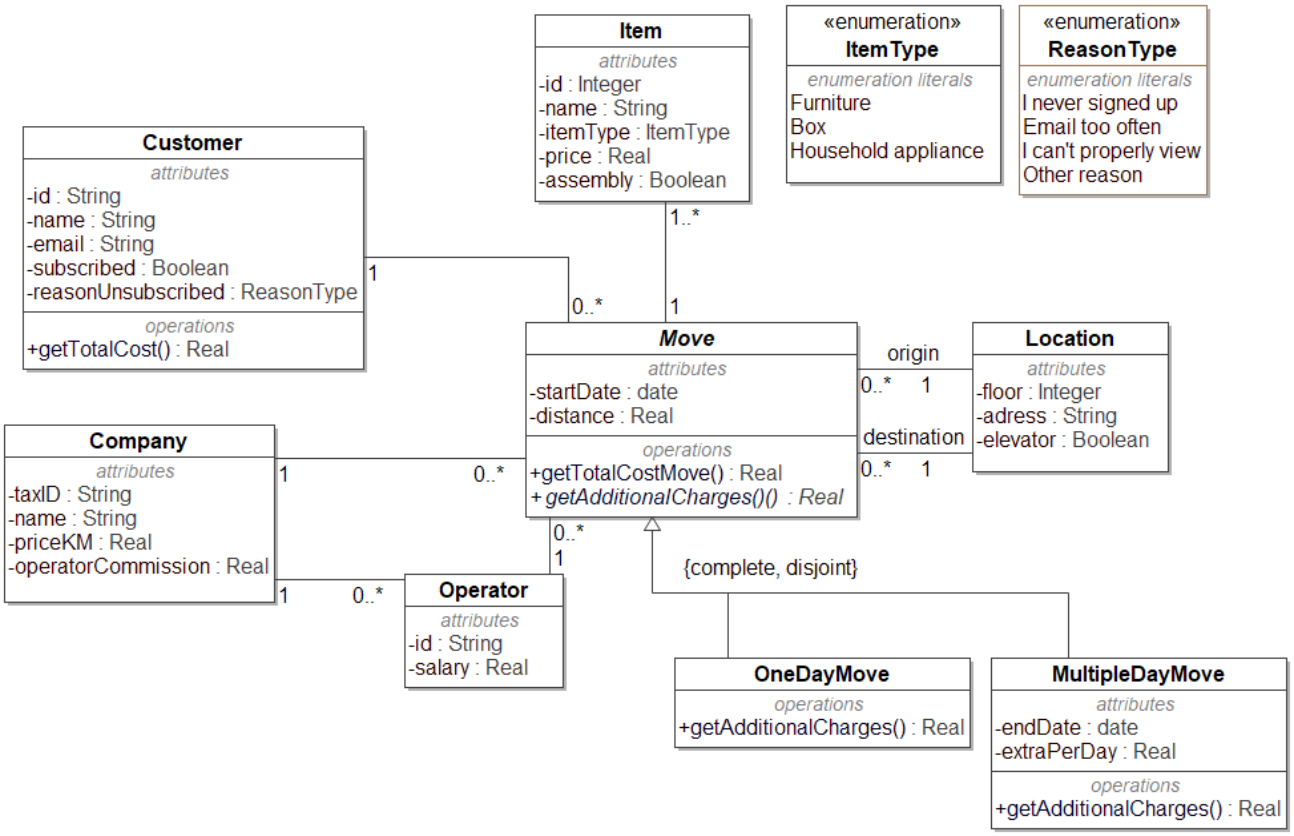
Solution to Exercise 1.2

```
public class Customer
{
  private String dni;
  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)
    {
            total += m.getTotalCostMove();
            // now Customer only talks with Move
            // now the Law of Demeter is fulfilled
            // now the coupling is lower
            // the use of this polymorphic operation makes
            // the Open-Closed principle be fulfilled.
    }
    returnfull;
  }

}


public abstract class Move
{
      private Date startDate;
      private Real distance;
      private Location origin;
      private Location destination;
      private List<Item> items;
      private Operator operator;
      private Company company;

      public Real getTotalCostMove()
      {
            Total Actual = 0;
            foreach (Item i in items)
            {
                  total += i.getPrice();
            }
            total += distance * company.getPriceKM();
            total += getAdditionalCharges();
            returnfull;
      }

      protected abstract Real getAdditionalCharges();
      // abstract operation that will be implemented in each subclass;
      // the decision of which charges to add is up to the subclass
```

```
}


public class MultipleDayMove inherits Move
{
      private Date endDate;
      private Real extraPerDay;

      public override Real getAdditionalCharges()
      {
            Integer extraDays = endDate - startDate;
            return extraDays * extraPerDay;

      }
}


public class DayMove inherits Move
{
      public override Real getAdditionalCharges()
      {
            return 0;
      }
}


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;
        }

}
```



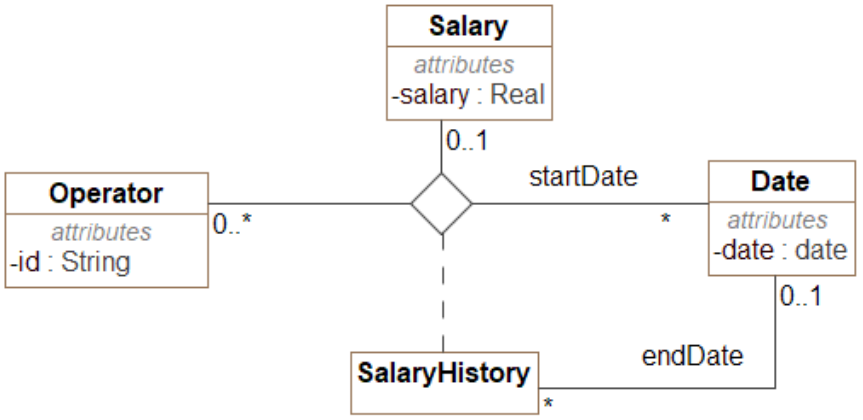
Solution to Exercise 1.3

d) According to the class diagram of the statement, only the current salary of an operator was recorded. In order to represent that an operator can have different salaries over time, we have to use the Historical Association pattern.

This pattern allows us to transform the salary attribute into a history of the values that the attribute has taken over time, providing the history of the workers' salaries.

e) First we pass the Salary attribute to a class and associate it with the Operator class:

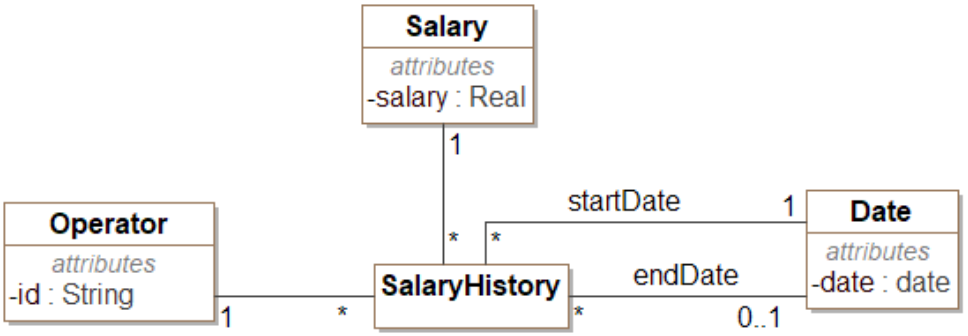
Once the attribute has been passed to a class, we can apply the historical association pattern. The result of the application of the pattern is:



Integrity constraints:

- The end date must be greater than the start date.
- An operator cannot have overlapping wages.

Current object-oriented programming languages do not allow the definition of ternary associations or associative classes. In order to work with these constructors, a small transformation must be applied to the class diagram to obtain another class diagram that represents the same semantics but with constructors that object-oriented programming languages enable. Applying this transformation we would obtain:



Integrity constraints:

- SalaryHistory key: idOperator + startDate
- Salary is an optional attribute.

In fact, as Date (date) could be considered as a data type, and we would normally put the data types as attributes, we can transform this solution into:

f)

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


      public String higherSalaryOperator()
      {
         Real maxSalary = 0;
         foreach (Operator or in operators)
            {
```

```
            Real monthExtra = o.getNumMovesCurrentMonth() * operatorCommission;
            Real salary = o.getSalary() + monthExtra;
            if (salary > maxSalary) {
                maxSalary = salary;
                id = o.getID();
            }
        }
        returnid;
    }
}


public class Operator
{
    public String id;
    public Real salary;
    public List<Move> moves;

    public String getID()
    {
        return id;
    }

    public Real getSalary()
    {
        return Salary;
    }

    public Real getNumMovesCurrentMonth()
    {
        Real movesMonth = 0;
        foreach (Move m in moves)
        {
            Date d = m.getStartDate();
            Date today = now();
            if (d.getYear() == today.getYear() and d.getMonth() ==
today.getMonth()) { //check that it is the same year and month
                movesMonth += 1;
            }
        }
        return movesMonth;
    }
}

public abstract class Move
{
```

```
    protected Date startDate;
    protected Real distance;
    protected Location origin;
    protected Location destination;
    protected List<Item> items;
    protected Operator operator;
    protected Company company;


    public String getStartDate()
    {
          return startDate;
    }

}
```

Solution to exercise 1.4

We will apply the MVC in the Presentation layer. The **UnsubscribeEmailView** represents the pattern view, the **UnsubscribeEmailPresentationController** represents the controller, and the **UnsubscribeEmailDomainController** represents the domain layer controller.

The operations of each class are described below:

Operations of the class **UnsubscribeEmailView**:

- UnsubscribeEmailView::showOptions() displays the different options corresponding to the different use cases (including email unsubscribe) on view.
- UnsubscribeEmailView::setOption(option) registers the selected use case and notifies the controller. Only if you want the view to record the selected use case.
- UnsubscribeEmailView::showReasons(reasons: Set (ReasonType)) displays the view with a dropdown of possible reasons for unsubscribing that are stored in the system.
- UnsubscribeEmailView::setReason(email:String, reason: ReasonType) logs the email and the reason for unsubscribing and notifies the controller.
- UnsubscribeEmailView::showEndMessage() shows the view with the message "Unsubscribed successfully".

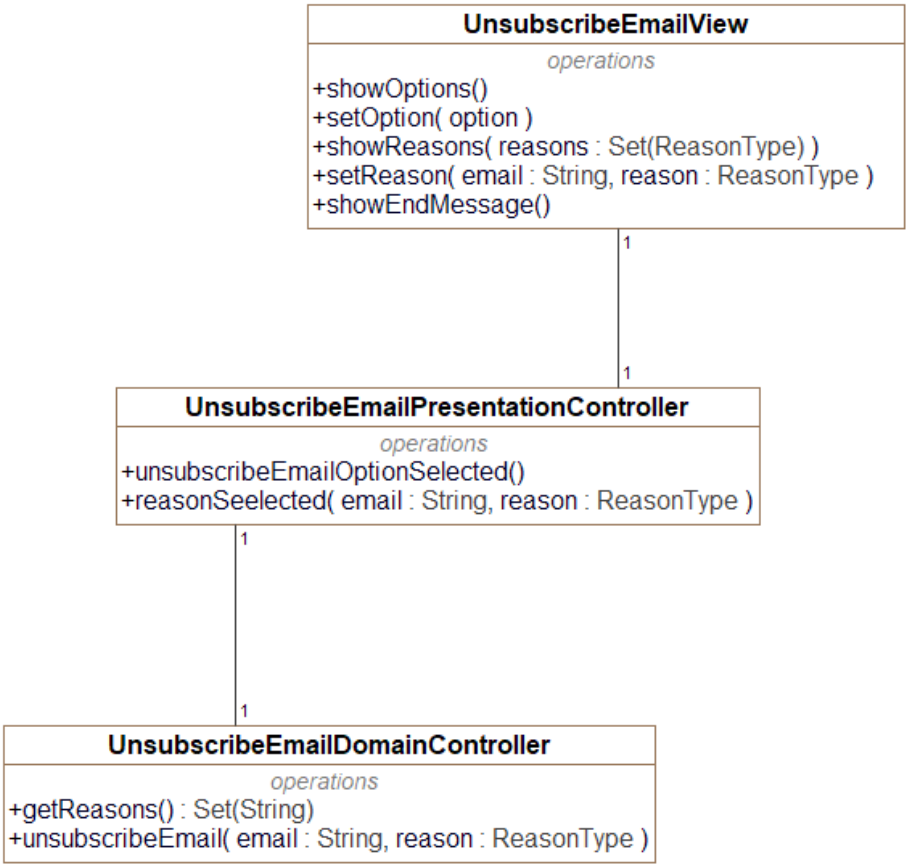
Operations of the class **UnsubscribeEmailPresentationController**:

- UnsubscribeEmailPresentationController::unsubscribeEmailOptionSelected() detects that the user has chosen the option to unsubscribe the email and invokes the operation.

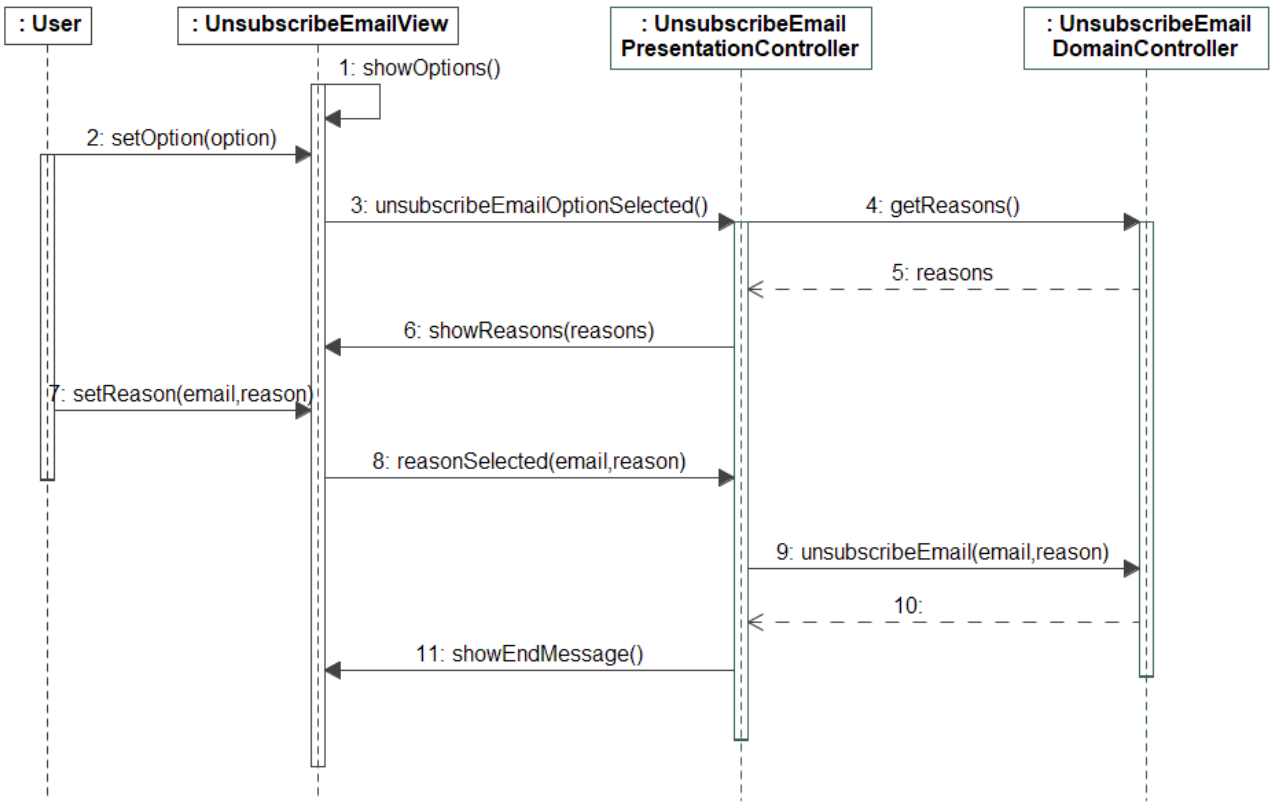
- UnsubscribeEmailPresentationController::reasonSelected(email: String, reason: ReasonType) detects that the user has marked the reason for unsubscribing and invokes the operation.

Operations of the class **UnsubscribeEmailDomainController** class**:**

- UnsubscribeEmailDomainController::getReasons(): Set(ReasonType) Gets the reasons for unsubscribing from the email in the system.
- UnsubscribeEmailDomainController::unsubscribeEmail(email: String, reason: ReasonType) unsubscribes the email and logs the reason.

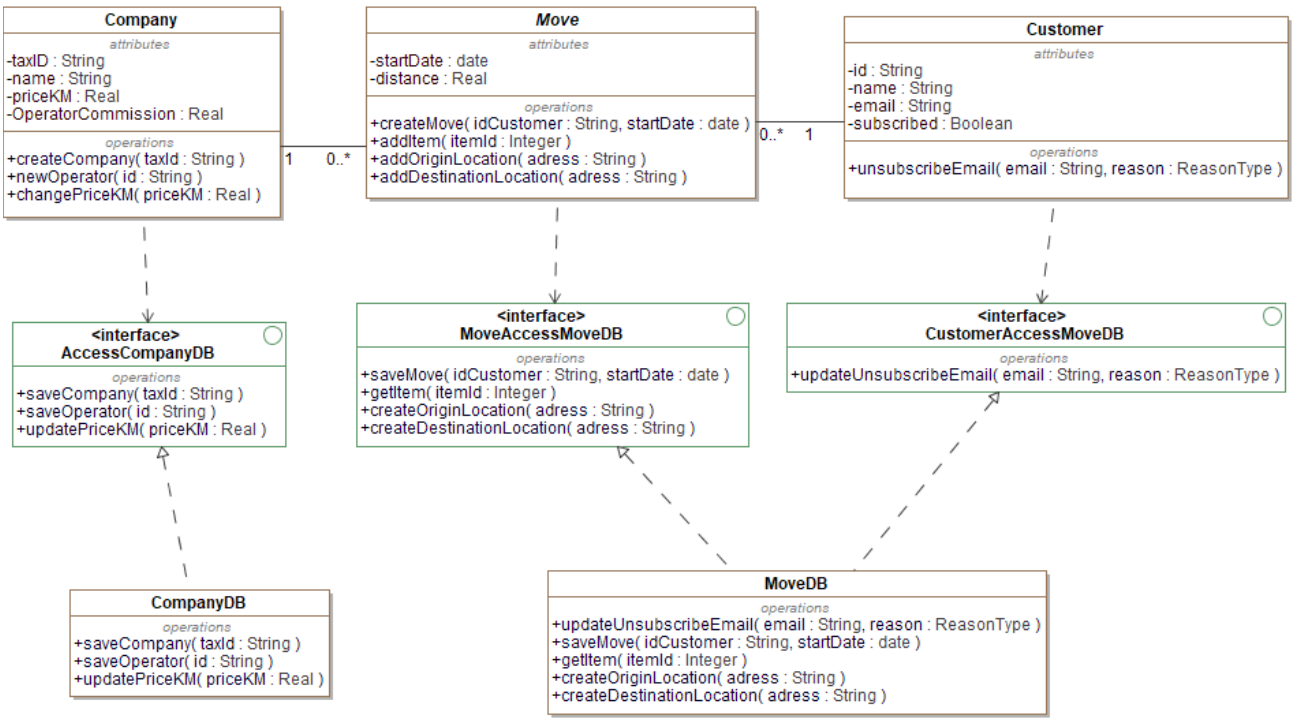


Solution to exercise 1.5



Solution to exercise 1.6

The design principle that is violated is interface segregation. This principle tells us that clients should not depend on operations that they do not use. In this case, the CompanyDB client objects depend on operations that it does not use, such as all those that go to the MoveDB database, and the MoveDB object depends on the CompanyDB operations, which it does not use either. The static analysis diagram that solves the violation of this principle is:



Solution to exercise 1.7

a)

The classes in the domain layer are where the business logic resides, where they work on the problem that our system solves.

This naturally includes classes, instances of which represent real-world objects, documents, or values, such as a mover, customer, location, object, carrier, and company:

- Item, Move, OneDayMove, MultipleDayMove, Location, Customer, Operator and Company and the types of the attributes to the first diagram

It also includes the classes that provide domain level operations for use by the presentation layer, which we can call domain controllers:

- UnsubscribeEmailDomainController

Finally, for interface segregation, we have added classes that represent the set of domain objects known to the system and that will eventually be implemented by the technical services layer to offer persistence:

- AccessCompanyDB and AccessMoveDB (from the solution of exercise 6)

b)

The technical services layer has the level of abstraction of the infrastructure on which our system runs (files, ports, etc.) or the level of abstraction of external services such as databases. In our case, therefore, it includes those classes that implement access to a relational database to retrieve persistent data:

- *CompanyDB, MoveDB*

Solution to Exercise 2

Solution to Exercise 2.1

a) This solution does not satisfy the Law of Demeter, since the *Customer*, in the *getTotalCharges()* operation, uses the *RentalOffice*, which is not directly associated with it. In other words, it obtains instances of *RentalOffice* that are not associated with it, invokes their methods and, therefore, talks to strangers. Consequently, any change in the logic of the *RentalOffice* could affect the code in the *Customer*.

b) This solution violates the Open-Closed design principle, since the *Customer* is aware that a subclass of *Rental* (*WebRental*) exists and invokes an operation. If, later on, we decide to extend the system to support a new subclass of *Rental* and the *getTotalCharges* code by *Customer* adding a new condition to handle this new subclass, we would have to modify the code of Customer to add a new restriction to deal with this new subclass.

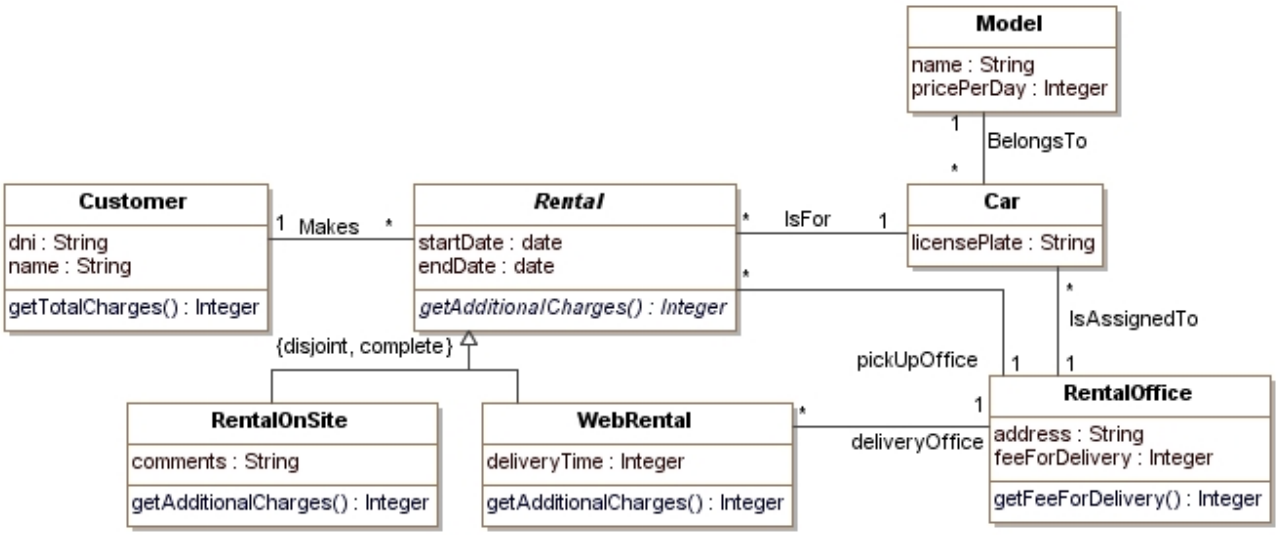
c) This solution violates the principle of Low Coupling. As a consequence of the non-satisfaction of the Law of Demeter, the coupling of this solution is high. Class *Customer* has knowledge of, and therefore coupling to, the *RentalOffice*.

Solution to exercise 2.2

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

  public Integer getTotalCharges()
  {
    Integer total = 0;
```

```
    foreach (Rental r in rentals)
    {
            total += r.getAdditionalCharges();
            // now customer only talks with rentals
            // now the Law of Demeter is fulfilled
            // now the coupling is less
            // the use of this polymorphic operation makes
            // the Open-Closed principle be fulfilled.
    }
    returnfull;
  }
}


public abstract class Rental
{
      protected Date startDate;
      protected Date endDate;
      protected Car car;
      protected RentalOffice pickUpOffice;

      protected abstract Integer getAdditionalCharges();
      // abstract operation that will be implemented in each subclass;
      // the decision of which charges to add depends on the subclass
}

public class WebRental inherits Rental
{
      private RentalOffice deliveryOffice;

      public override Integer getAdditionalCharges()
      {
            if (this.deliveryOffice != this.pickUpOffice)
                  return deliveryOffice.getFeeForDelivery();
            else
                  return 0;

      }
}

public class RentalOnSite inherits Rental
{
      private String comments;

      public override Integer getAdditionalCharges()
      {
```

```
            return 0;
      }
}

public class RentalOffice
{
      private String address;
      private Integer feeForDelivery;

      public float getFeeForDelivery()
      {
            return feeForDelivery;
      }
}
```

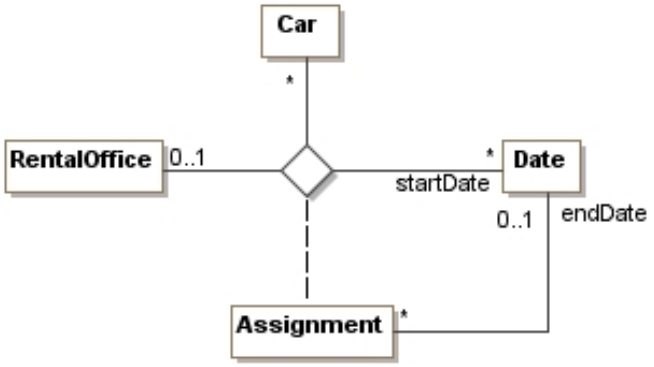


Solution to exercise 2.3

a) According to the class diagram of the statement, only the current assignment of the car was registered. In order to represent that a car can be moved from offices in different periods we have to use the Historical Association pattern.

   This pattern allows us to transform the *IsAssignedTo* association into a history of the values that the association has taken over time, providing the history of the assignment of cars to offices.

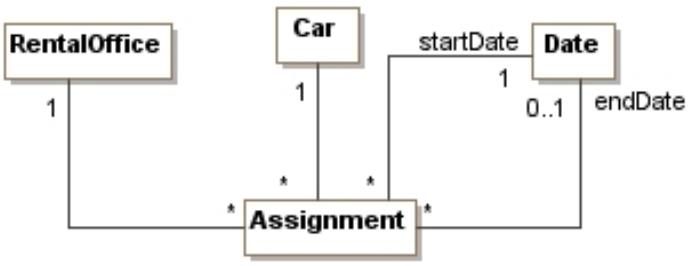
b) The result of applying the pattern is:



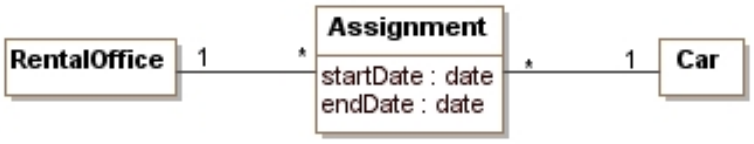
This diagram reflects several situations (restrictions) that can occur in the car rental company:

- Given a car and any start date, there can be at most one assignment. Note that for the same car, on any other start date, you may not have any assignments.
- Given a car and an office there can be many assignments. For example, a car may be assigned to one office, move to another office, and later return to the initial office. Therefore, there will be two assignments for the same car and office.
- Given an office and any starting date, there can be many assignments. For example, on a certain date many cars can arrive at the office (and therefore have different assignments).

Current object-oriented programming languages do not allow the definition of ternary associations or associative classes. In order to work with these constructors, a small transformation must be applied to the class diagram to obtain another class diagram that represents the same semantics, but with the constructors that object-oriented programming languages enable. After applying this transformation, we will obtain:



In fact, since *Date* a data type, which we would normally put as an attribute, we can transform this solution into:



c)

```
public class RentalOffice
{
	public String address;
	public Integer feeForDelivery;
	public List<Assignment> assignments;

	public String expensiveCar()
	{
		Integer maxPrice = 0;
		foreach (Assignment to in assignments)
		{
			if (a.getEndDate() == null) {
			// is a current assignment
				Integer price = a.getCarPricePerDay();
				if (price > maxPrice) {
					maxPrice = price;
					licensePlate = a.getCarLicensePlate();
				}
			}
		 }

		return licensePlate;
	}
}


public class Assignment
{
	public Date startDate;
	public Date endDate;
	public Car car;

	public Date getEndDate()
	{
		return endDate;
	}

	public Integer getCarPricePerDay()
	{
		return car.getModelPricePerDay();
```

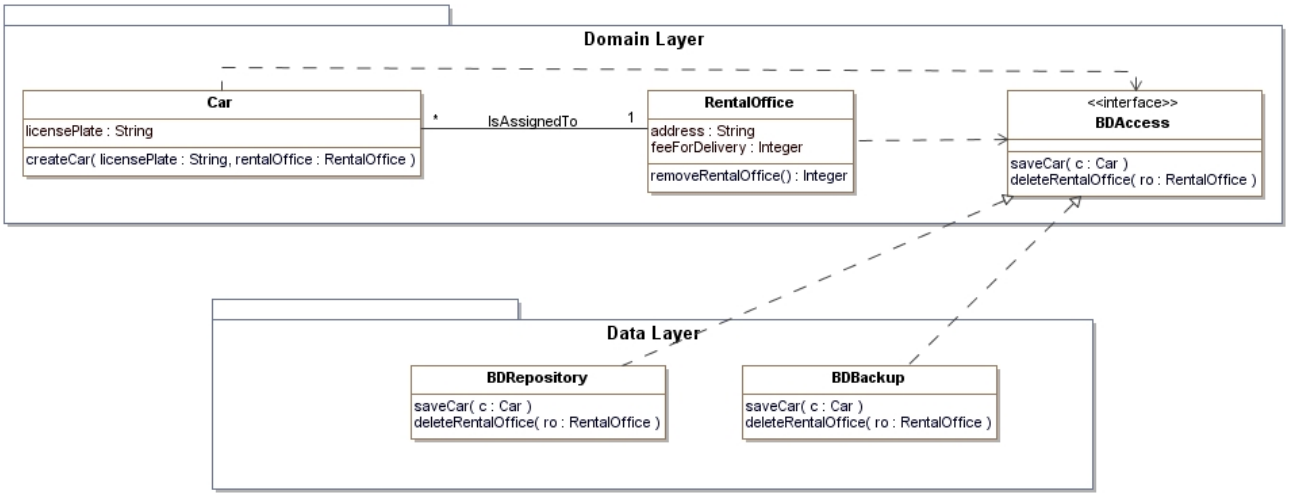
```
	}

	public String getCarLicensePlate()
	{
		return car.getLicensePlate();
	}
}


public class Car
{
	public String licensePlate;
	public Model model;

	public String getLicensePlate()
	{
		return licensePlate;
	}

	public Integer getModelPricePerDay()
	{
		return model.getPricePerDay();
	}

}

public class Model
{
	public String name;
	public Integer pricePerDay;

	public String getPricePerDay()
	{
		return pricePerDay();
	}
}
```

Solution to exercise 2.4

This solution clearly violates some of the design principles: the Dependency Inversion principle and the Low Coupling principle. The principle of Dependency Inversion is not fulfilled since the *Car* and *RentalOffice* which are at the level of abstraction of the domain layer, depend on classes that have a much lower level of abstraction, since they have the level of abstraction of the data layer. The principle of loose coupling is broken since each of the classes in the domain layer is coupled with

the two classes in the data layer. To satisfy the Dependency Inversion principle we add an interface to the *BDAccess* and this interface is implemented by the two data layer classes. In this way, we will comply with the Dependency Inversion principle and, at the same time, we will reduce the coupling.

The new class diagram will be as follows:



Solution to exercise 2.5

We will apply the MVC to the Presentation layer. The *RemoveCarView* represents the pattern view, the *RemoveCarPresentationController* represents the controller, and the *RemoveCarDomainController* represents the domain layer controller.

The operations of each class are described below:

Operations of the **RemoveCarView** class**:**

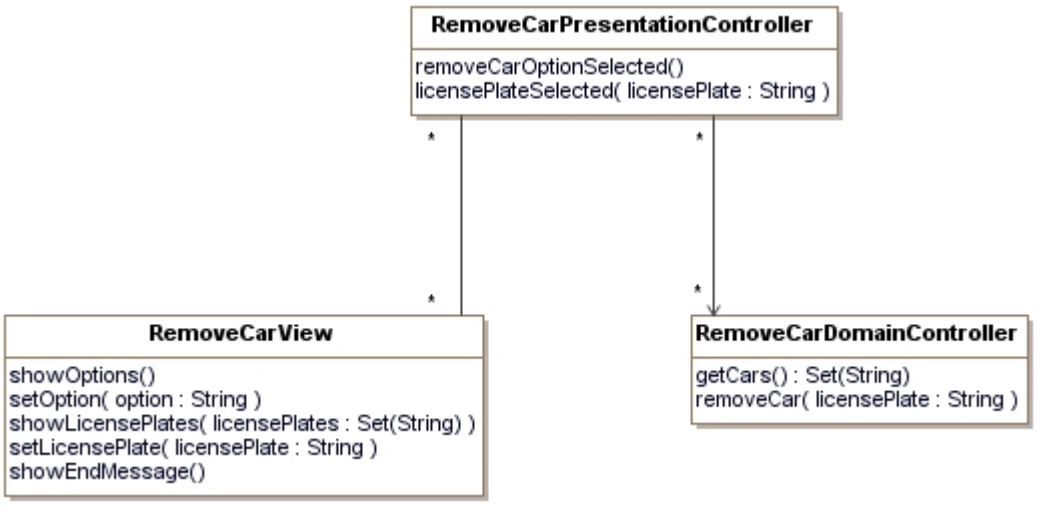
- RemoveCarView::showOptions() displays the different options corresponding to the different use cases (including unloading a car) in view.
- RemoveCarView::setOption(option) registers the selected use case and reports it to the controller. Only if one wants the view to record the selected use case.
- RemoveCarView::showLicensePlates(licensePlates: Set(String)) shows the view with the different labels and the license plates of the cars stored in the system in a dropdown.
- RemoveCarView::setLicensePlate(licensePlate: String) records the license plate of the car to be removed and notifies the controller. Only if one wants to register the license plate.
- RemoveCarView::showEndMessage() shows the view with the message “Car removed successfully”.

Operations of the **RemoveCarPresentationController** class**:**

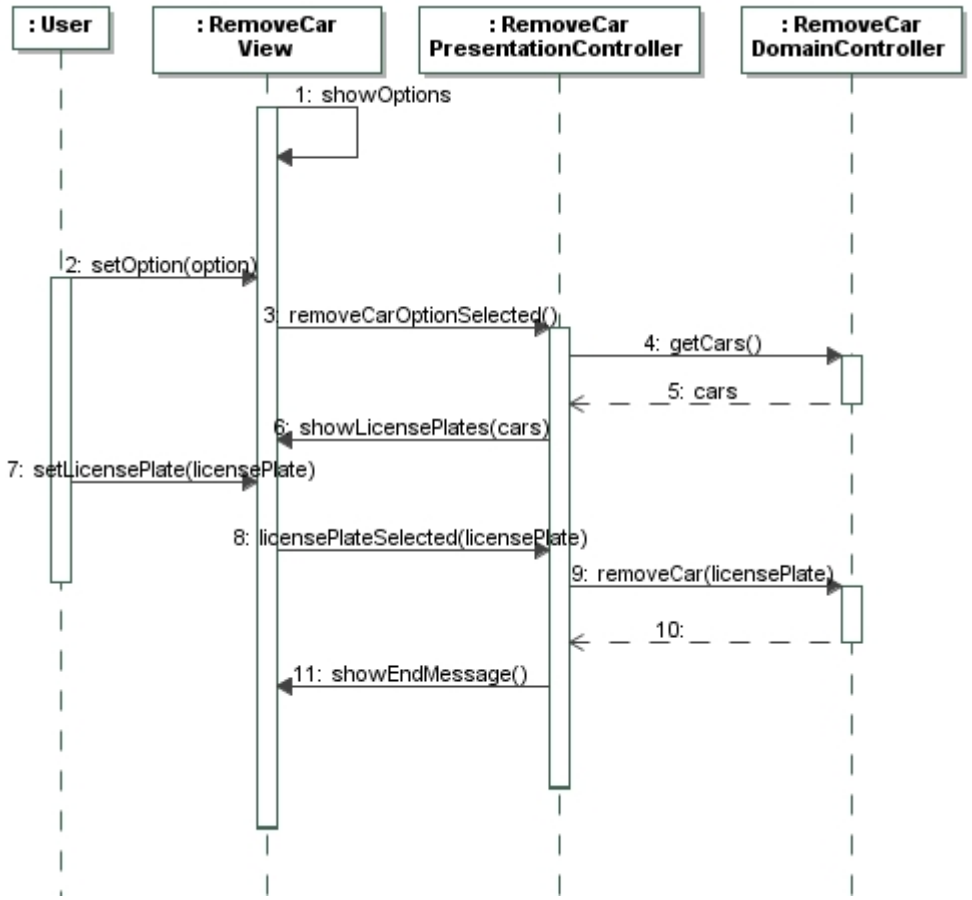
- RemoveCarPresentationController::removeCarOptionSelected() detects that the user has chosen the option to remove a car and calls the operation.
- RemoveCarPresentationController::licensePlateSelected(licensePlate: String) detects that the user has entered the license plate of the car to be removed and calls the operation.

Operations of the **RemoveCarDomainController** class**:**

- RemoveCarDomainController::getCars(): Set(String) Gets the license plates of all cars in the system.
- RemoveCarDomainController::removeCar(licensePlate: String) removes the car with the specified license plate as a parameter.



Solution to exercise 2.6



Solution to Exercise 3

Solution to exercise 3.1

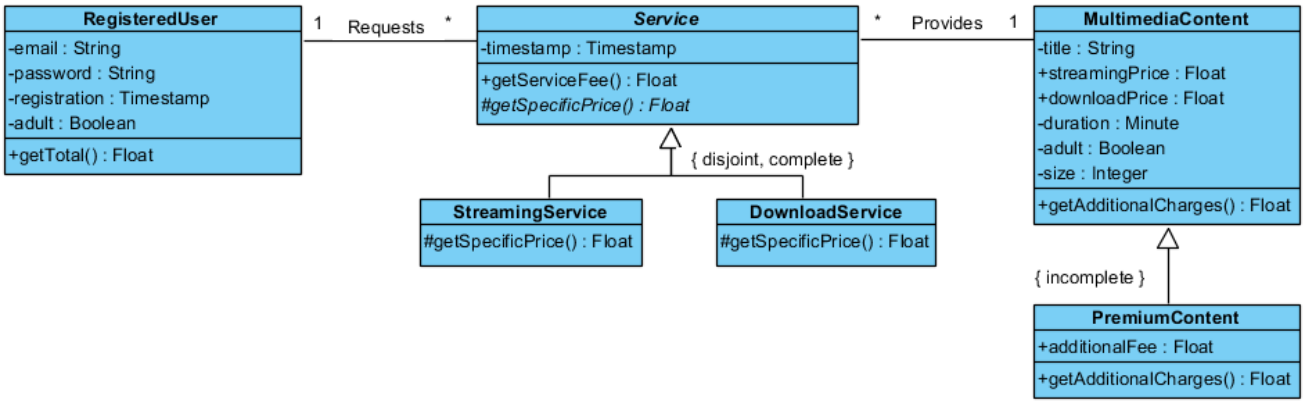
a) No, this solution does not satisfy the Law of Demeter, since the RegisteredUser class, in the getTotal() method, uses the MultimediaContent class, which is not directly associated with it. That is, it obtains instances of MultimediaContent that are not associated with it, invokes methods on them, and therefore talks to strangers. Consequently, any change in this logic of multimedia content would affect the RegisteredUser code.

b) No, this solution violates the Open-Closed design principle, since the RegisteredUser class is aware of Service subclasses and calls one method or another depending on which subclass the object belongs to each iteration. If we later extend the system to support a new Service subclass, we should modify the RegisteredUser code to get the corresponding price inside a new else-if.

Solution to exercise 3.2

```
public class RegisteredUser
{
      private List<Service> services;

      public float getTotal()
      {
            float total = 0.0;
            foreach (Service s in services) total += s.getServiceFee();
            // esta clase ya sólo "habla" con Service,
            // quien sí se podrá comunicar con MultimediaContent
            // al estar asociada con ella
            return total;
      }
}

public abstract class Service
{
      protected MultimediaContent content;

      // operación plantilla
      public float getServiceFee()
      {
            return this.getSpecificPrice() + content.getAdditionalCharges();
      }

      protected abstract float getSpecificPrice();
      // operación abstracta que será implementada en cada subclase;
      // la decisión de qué precio aplicar depende de la subclase
}

public class StreamingService inherits Service
{
      protected override float getSpecificPrice()
      {
            return content.streamingPrice;
```

```
        }
}

public class DownloadService inherits Service
{
        protected override float getSpecificPrice()
        {
                return content.downloadPrice;
        }
}

public class MultimediaContent
{
        public float getAdditionalCharges()
        {
                return 0.0;
                // operación polimórfica;
                // por defecto, un contenido no tiene cargo adicional
        }
}

public class PremiumContent inherits MultimediaContent
{
        private float additionalFee;
        public overrides float getAdditionalCharges()
        {
                return additionalFee;
                // sólo los premiums tienen cargo adicional, así que
                // sobreescribimos la operación de la superclase para así informarlo
        }
}
```



Solution to exercise 3.3

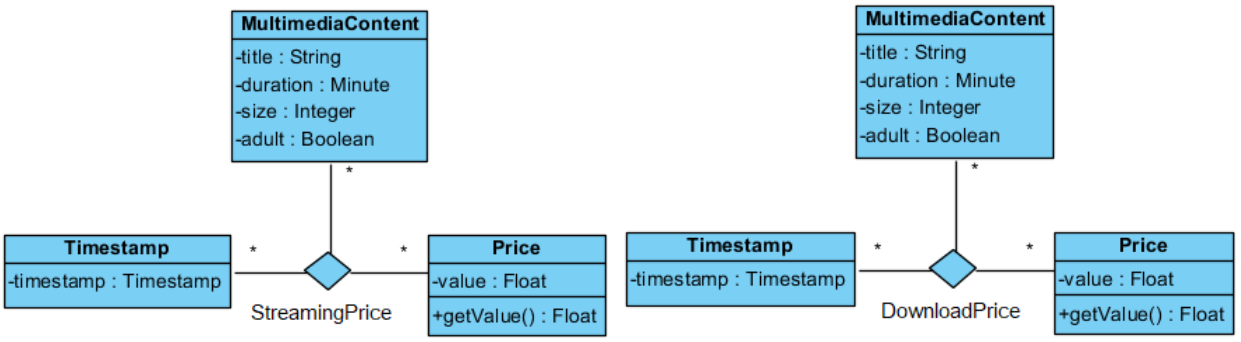
a)

Right now the price of a multimedia content is an attribute of it and, therefore, we only know the current price of each content. Although it is an attribute and not an association, the pattern that allows us to solve this problem is the Historical Association pattern.

This pattern allows us to transform an association (or an attribute, as the case may be) into a history of the values that the association or attribute has taken over time. This is exactly the problem we want to solve.

b)

Since what we have is an attribute, we need to apply the variation where we first extract the attribute as an association. The end result is:



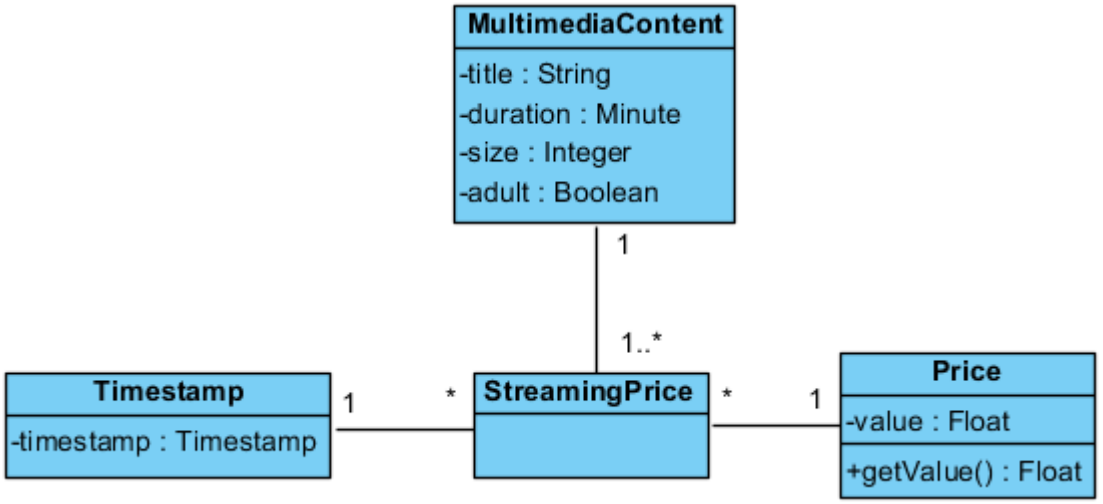
This diagram reflects a restriction that we did not have before: given a specific content and date-time (instant or timestamp), we can only have one price (for each type of service). In other words, content can only have one price change at a given moment.

In this solution we assume that the price of a content is the current price until an association with a later Timestamp appears, at which point that price becomes historical and is no longer the current price. Therefore:

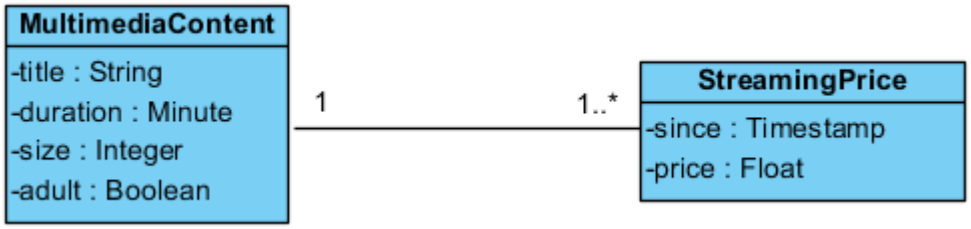
- We are already reflecting that a multimedia content cannot have two prices for the same service at the same time, since we always assume that if there are two associations, each one represents a price that ceases to be valid when the next one begins.

- We are assuming that a content can NOT have periods of time in which it does not have any price for each service.

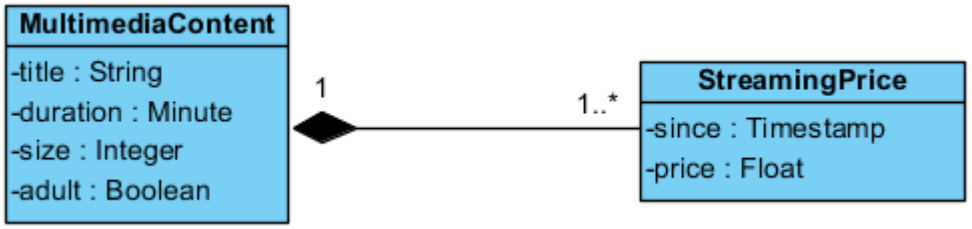
On the other hand, with current object-oriented languages it is not usually allowed to use ternary associations. It can be done by modeling the association history as having an associative class and then doing a little transformation. For example, with *StreamingPrice*:



In fact, as Timestamp and Price we could consider them data types, which we would normally put as attributes, we can transform this solution into:



Finally, we can see that both StreamingPrice and DownloadPrice are components of a multimedia content, since they belong to one and only one multimedia content, it makes no sense for us to change the content and if we destroy a content we will have to destroy its price history. Therefore, we can model it as:



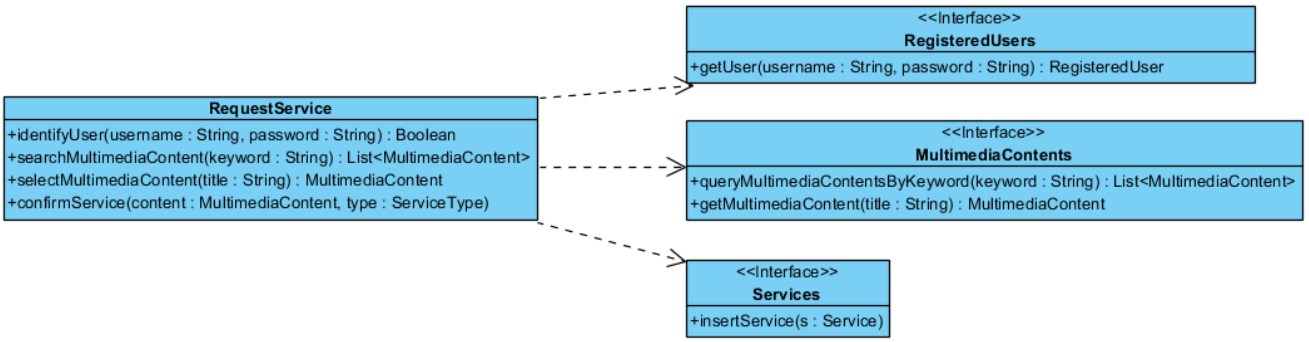
c)

```
public class MultimediaContent
{
	private List<StreamingPrice> streamingHistory;

	public float getCurrentStreamingPrice()
	{
		if (streamingHistory.isEmpty()) return null;
		return streamingHistory.last().price;
	}
}

public class StreamingPrice
{
	public Timestamp since;
	public float price;
}
```

Solution to exercise 3.4

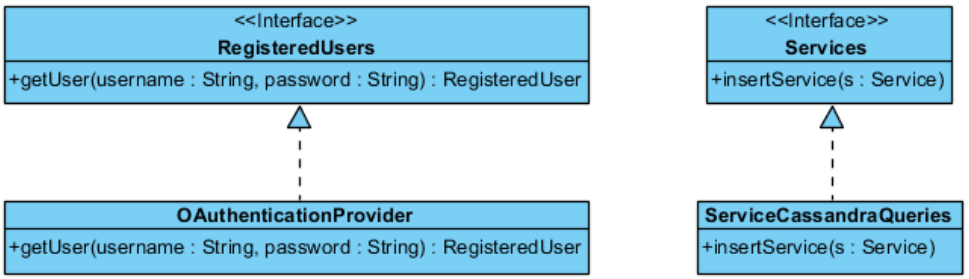
It can be questioned whether this solution satisfies some of the design principles. But a principle that clearly fails is the principle of Dependency Inversion, since RequestService, which is at the domain logic abstraction level, depends on classes that have a much lower abstraction level, since they have the abstraction level of the infrastructure; in fact, that of a particular non-relational database vendor and that of a popular authentication method.

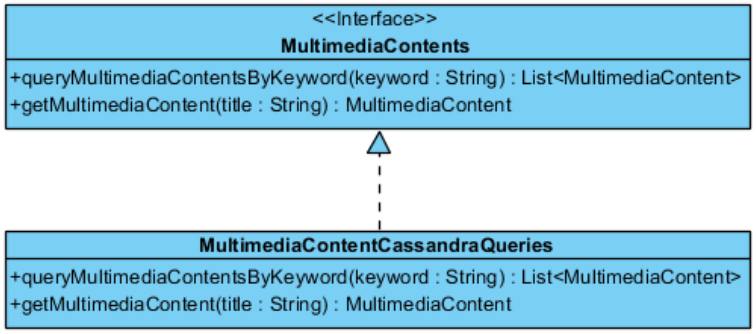
At the class diagram level, the solution is to make the RequestService depend on abstractions and the details also depend on these abstractions. One way to achieve this is to create an abstraction for each overly concrete class:



In this case we have created 3 interfaces that represent, at the abstraction level of the domain, the set of users, multimedia content and requested services that we have in the system. Now RequestService does not depend on details but on another abstraction that is at the same level of abstraction.

Then, the details, which we already had, come to depend on the abstractions. A valid form of dependency is that of interface implementation:





Solution to exercise 3.5

In the solution to the previous exercise, we have already modeled that *RequestService* only uses (invokes) methods of abstractions, interfaces belonging to the domain class, such as *MultimediaContents*, *RegisteredUsers*, and *Services*.

At some point, however, we need to get an instance of this type. If RequestService instantiated the instances it needs itself, it would be coupled back to the concrete classes of the technical services layer:

```
MultimediaContents contents = new MultimediaContentCassandra();
```

So we want to choose the implementation of *MultimediaContents* (and so on) to use without coupling *RequestService* to that implementation. This is exactly the problem that the Dependency Injection architecture pattern solves.

The solution is to associate an instance of MultimediaContents, an instance of RegisteredUsers, and an instance of Services to the RequestService. Going down to the code level, RequestService will have an attribute for each of these instances. So, instead of the RequestService itself initializing these attributes, we will inject these dependencies from outside. We propose to use the variant in which they are injected through the constructor:

```
class RequestService
{
      private MultimediaContents contents;
      private RegisteredUsers users;
      private Services services;

      public RequestService (
            MultimediaContents cs,
            RegisteredUsers us,
            Services ss)
      {
            this.contents = cs;
            this.users = us;
            this.services = ss;
      }

      public void identifyUser (string email)
      {
            RegisteredUser ru = users.getRegisteredUser(email);
```

```
            ...
        }
}
```

We have shown the resulting layout and how one of the abstractions would be used from one of the class operations.