Práctica 1 - Solución

Análisis y diseño con patrones (Universitat Oberta de Catalunya)

Análisis y Diseño con Patrones

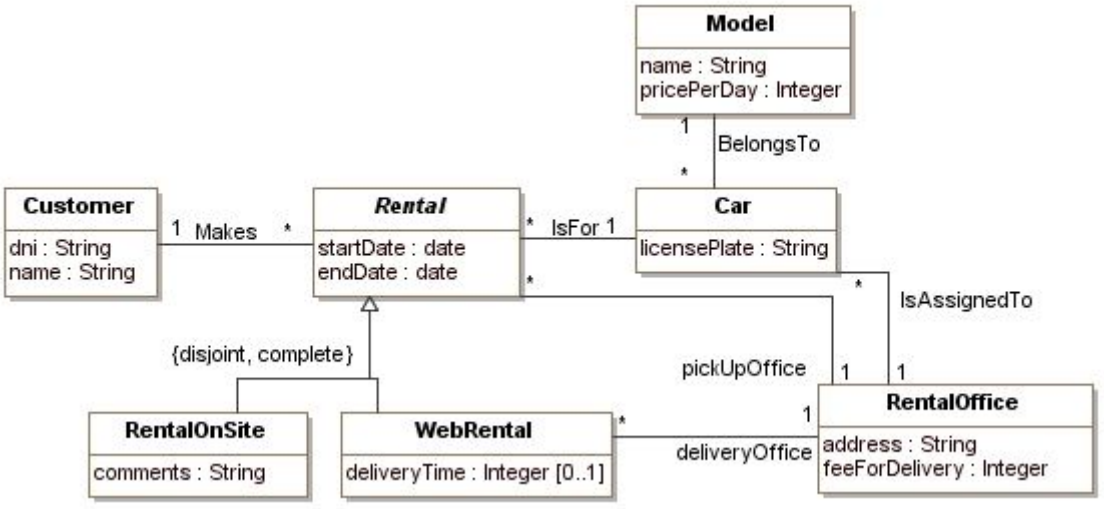
Práctica 1: Principios de diseño, patrones de análisis y arquitectónicos

Una empresa nos ha contratado para desarrollar un software para gestionar los alquileres de coches que hace a sus clientes. Los clientes pueden alquilar coches de diferentes modelos durante un periodo de tiempo. Los coches se tienen que recoger en una oficina de recogida. Los alquileres se pueden hacer directamente en las oficinas de la empresa o bien mediante la aplicación web que la empresa pone a disposición de sus clientes. Para los alquileres que se hacen a través de las oficinas se pueden registrar, si fuera necesario, los comentarios sobre el alquiler. Este tipo de alquileres obliga a los clientes a devolver el coche en la misma oficina donde se ha recogido. Para los alquileres que se hacen vía web se registra la hora de retorno del coche. En estos tipos de alquileres los clientes pueden devolver los coches en una oficina diferente de la donde se ha recogido pagando un cargo por entrega.

Este software es una parte de un sistema más grande. Obviaremos muchos de los elementos que tendría un sistema real. La razón no es otra que hacer una práctica didáctica, y que la dificultad y dedicación prevista sean las deseadas.

Disponemos ya de un análisis previo que podemos utilizar como punto de partida, pero que habrá que corregir y mejorar. A continuación, disponéis del diagrama de clases de esta parte del software:

Diagrama de clases



Claves de las clases:

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

Restricciones de integridad:

- Un cliente no puede tener alquileres solapados.
- La fecha de inicio de un alquiler tiene que ser anterior a la fecha final del alquiler.
- La oficina de recogida de un coche de alquiler tiene que ser la misma que la oficina donde está asignado el coche de alquiler.
- Si la oficina de recogida y de entrega de un alquiler web son diferentes, la hora de entrega del coche de alquiler tiene que ser anterior a las 13 horas.

Para hacer tareas de marketing, nos han pedido diseñar una operación para calcular la suma total de los cargos (*feeForDelivery*) que un cliente ha pagado por los alquileres que ha hecho por la web, donde ha entregado el coche en una oficina diferente de la de recogida. A continuación, disponéis del pseudocódigo que implementa este método:

```
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 total;
  }
}
```

Ejercicio 1 (10%)

Revisamos el pseudocódigo de la operación *getTotalCharges* de la clase *Customer* y nos preocupa que no se cumplan algunos de los principios de diseño. En concreto, se pide:

a) Indica si este diseño satisface o no la Ley de Demeter y razona tu respuesta.
b) Indica si este diseño satisface o no el principio de diseño Abierto-Cerrado y razona tu respuesta.
c) Indica si este diseño satisface o no el principio de diseño Bajo Acoplamiento y razona tu respuesta.

Solución:

a) Esta solución no satisface la Ley de Demeter, puesto que la clase *Customer*, en el método *getTotalCharges*, utiliza la clase *RentalOffice*, que no tiene directamente asociada. Es decir, obtiene instancias de *RentalOffice* que no tiene asociadas, invoca métodos de las mismas y, por lo tanto, habla con desconocidos. En consecuencia, cualquier cambio en la lógica de la clase *RentalOffice* podría afectar al código de la clase *Customer*.
b) Esta solución viola el principio de diseño Abierto-Cerrado, puesto que la clase *Customer* es consciente de que existe una subclase de *Rental* (*WebRental*) e invoca una operación. Si, más adelante, ampliamos el sistema para admitir una nueva subclase de *Rental* y la operación *getTotalCharges* necesitara invocar una operación de esta nueva subclase, tendríamos que modificar el código de *Customer* añadiendo un nueva condición para tratar esta nueva subclase.
c) Esta solución viola el principio de Bajo Acoplamiento. Como consecuencia de la no satisfacción de la Ley de Demeter, el acoplamiento de esta solución es alto. La clase *Customer* tiene conocimiento y, por lo tanto, acoplamiento, con la clase *RentalOffice*.

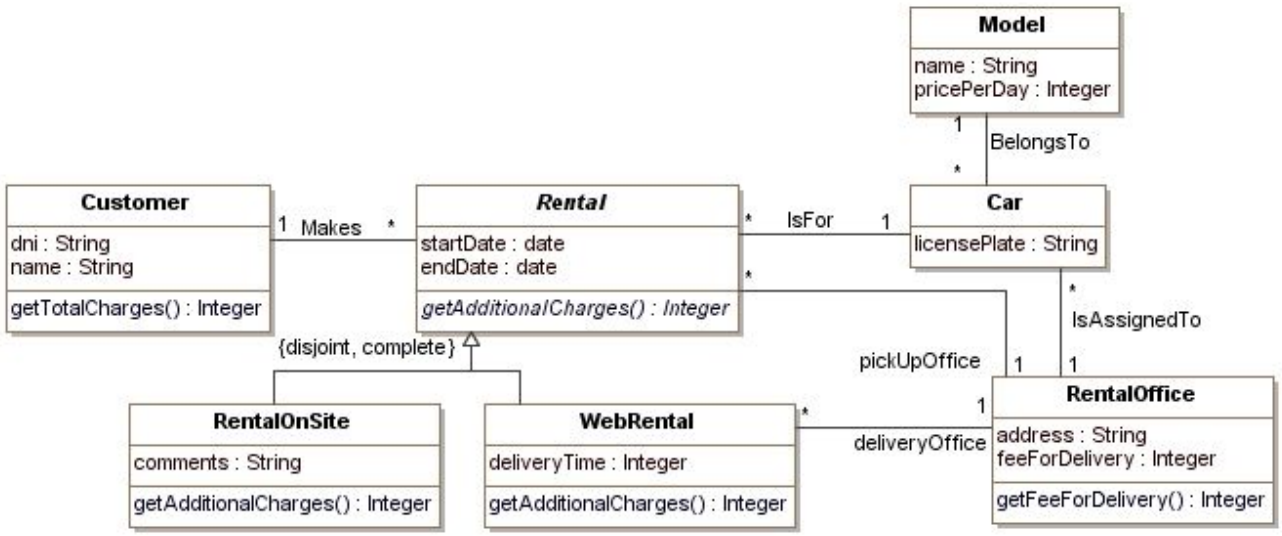
Ejercicio 2 (20%)

Propón una solución alternativa completa (también su pseudocódigo) que corrija los problemas de la operación *getTotalCharges* de *Customer* que has detectado en el ejercicio anterior, y que por lo tanto, cumpla cualquier principio violado. Modifica el diagrama de clases añadiendo todas las operaciones de tu solución.

Solución:

```
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();
            // ahora customer solo habla con rentals
            // ahora se cumple la Ley de Demeter
            // ahora el acoplamiento es menor
            // el uso de esta operación polimórfica hace que se cumpla
            // el principio de Abierto-Cerrado.
    }
    return total;
  }
}


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

      protected abstract Integer getAdditionalCharges();
```

```
	// operación abstracta que será implementada en cada subclase;
	// la decisión de cuáles son los cargos a sumar depende de la subclase
}

public class WebRental inherits Rental
{
	private RentalOffice deliveryOffice;

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

	}
}

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;
	}
}
```



Ejercicio 3 (30%)

Recientemente, la empresa de alquiler de coches se ha dado cuenta de que hay oficinas que tienen asignados muchos coches que no alquilan y, en cambio, hay otras que podrían alquilar más coches de los que tienen asignados. Es por eso que la empresa ha decidido mover los coches de una oficina a otra en función de las necesidades y registrar las asignaciones de los coches durante el tiempo. De este modo los coches podrán estar asignados a diferentes oficinas en periodos de tiempos diferentes.

Nos proponemos buscar una solución que nos permita representar esta información en nuestro sistema. En concreto, se pide:

a) Describe brevemente cómo solucionarías este problema. En caso de que propongas la aplicación de un patrón, explica cuál y razona su conveniencia.

b) Propón una solución detallada en forma de diagrama de clases. Muestra solo las clases que cambien respecto al diagrama del enunciado.

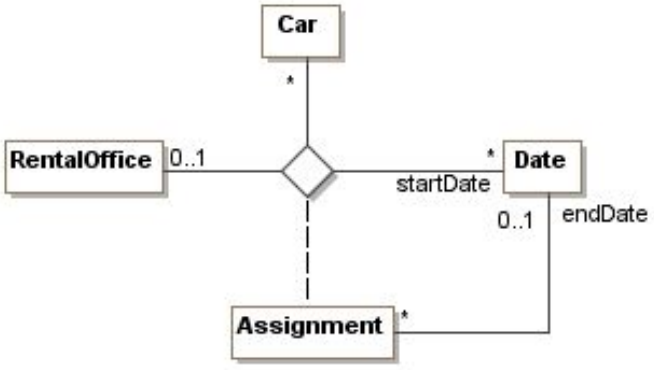
c) Indica cómo sería el pseudocódigo resultante de la operación *ExpensiveCar():String* de la clase *RentalOffice* que devuelve la matrícula del coche que está asignado a la oficina actualmente y que tiene un precio por día mayor. Si hay más de uno, podéis devolver el coche que queráis. Podéis suponer que como mínimo habrá un coche asignado actualmente a la oficina.

Solución:

a) Según el diagrama de clases del enunciado, solo se registraba la asignación actual del coche. Para poder representar que un coche se puede mover de oficinas en diferentes periodos tenemos que utilizar el patrón Asociación Histórica.

Este patrón nos permite transformar la asociación *IsAssignedTo* en un histórico de los valores que ha tomado la asociación a lo largo del tiempo, proporcionando el histórico de la asignación de los coches a las oficinas.

b) El resultado de la aplicación del patrón es:

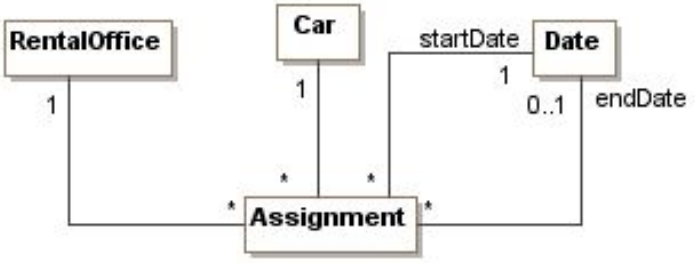


Este diagrama refleja varias situaciones (restricciones) que se pueden dar en la empresa de alquiler de coches:

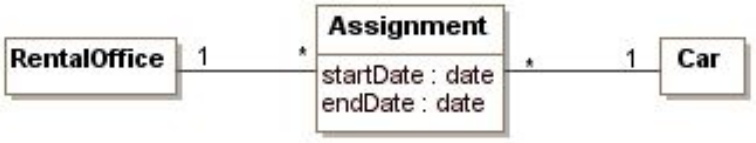
- Dado un coche y una fecha de inicio cualquiera, como mucho puede haber una asignación. Fijaos que para el mismo coche, en cualquier otra fecha de inicio, es posible que no tenga ninguna asignación.
- Dado un coche y una oficina cualquiera puede haber muchas asignaciones. Por ejemplo, un coche puede estar asignado a una oficina, moverse a otra oficina y más tarde volver a la oficina inicial. Por lo tanto, habrá dos asignaciones para el mismo coche y oficina.
- Dada una oficina y una fecha de inicio cualquiera puede haber muchas asignaciones. Por ejemplo, en una fecha determinada pueden llegar muchos coches a la oficina (y por tanto haber diferentes asignaciones).

Los lenguajes de programación orientados a objetos actuales no permiten definir asociaciones ternarias, ni clases asociativas. Para poder trabajar con estos constructores, hay que aplicar una

pequeña transformación al diagrama de clases para obtener otro diagrama de clases que represente la misma semántica, pero con constructores que los lenguajes de programación orientados a objetos permitan representar. Aplicando esta transformación obtendremos:



De hecho, como la clase *Date* la podríamos considerar un tipo de datos, que normalmente pondríamos como atributo, esta solución la podemos transformar en:



c)

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

      public String expensiveCar()
      {
            Integer maxPrice = 0;
            foreach (Assignment a in assignments)
            {
                  if (a.getEndDate() == null) {
                  // es una asignación actual
                        Integer price = a.getCarPricePerDay();
                        if (price > maxPrice) {
                              maxPrice = price;
                              licensePlate = a.getCarLicensePlate();
                        }
                  }
             }

            return licensePlate;
```

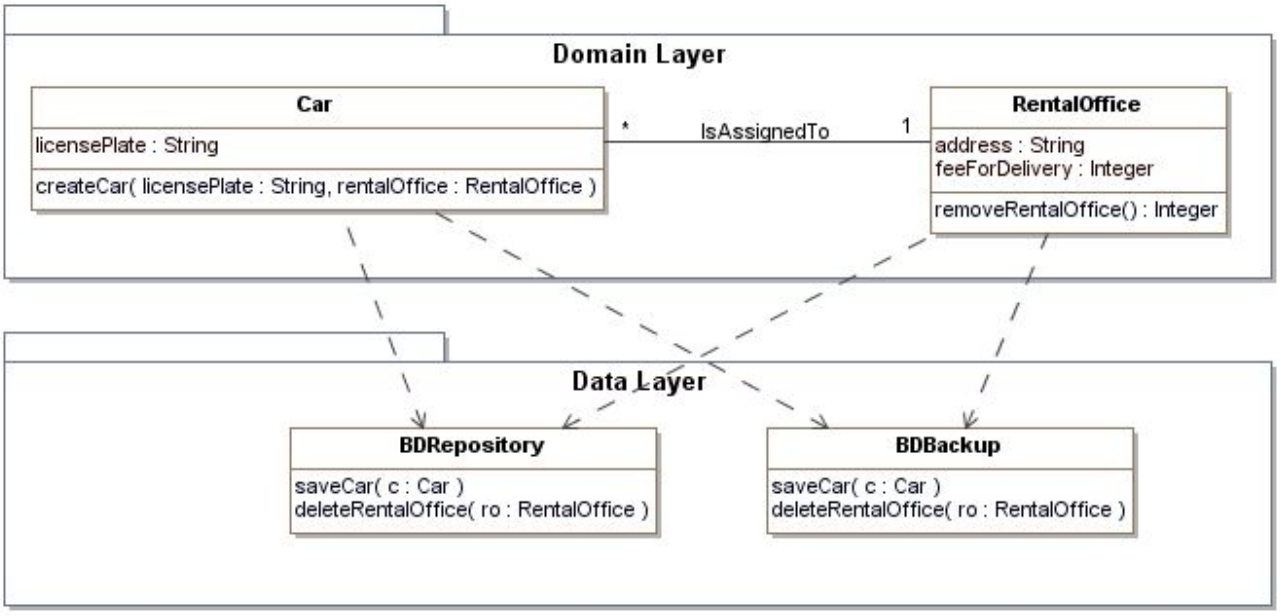
```
	}
}


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();
	}
}
```

Ejercicio 4 (15%)

Empezamos ahora a diseñar la arquitectura del sistema y hemos decidido utilizar una arquitectura de 3 capas: presentación, dominio y servicios técnicos. La empresa de alquiler de coches nos ha pedido tener los datos siempre accesibles. Para satisfacer este requerimiento tendremos una base de datos de backup para que cuando la base de datos principal no esté disponible se redireccionen las invocaciones a la base de datos que hace de backup.

Tenemos una primera versión de la capa de dominio y de la capa de servicios técnicos (en nuestro caso será una capa de datos que permitirá el acceso a la base datos). En la figura siguiente tenéis un fragmento de estas dos capas.



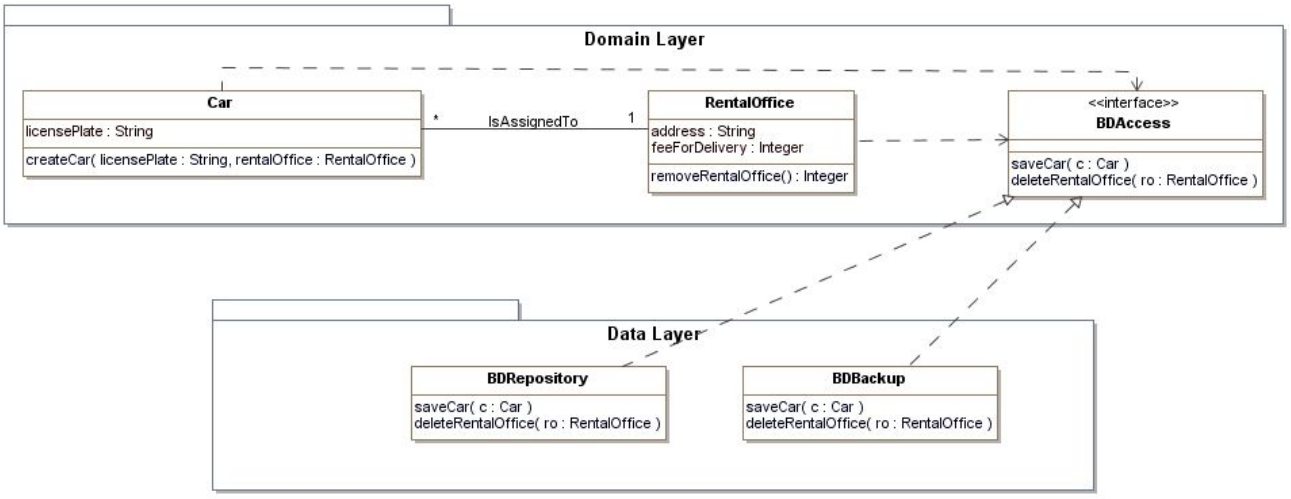
Una vez hemos acabado de hacer un diseño inicial de estas capas, nos preguntamos: ¿cumple esta propuesta todos los principios de diseño que hemos visto? En caso afirmativo, justifica por qué

crees que es así. En caso negativo, indica qué principios crees que no cumple y proporciona el diagrama de clases de las dos capas con los cambios introducidos para cumplirlos.

Solución:

Esta solución incumple claramente algunos de los principios de diseño: el principio de Inversión de Dependencias y el de Bajo Acoplamiento. El principio de Inversión de Dependencias se incumple puesto que las clases *Car* y *RentalOffice* que están al nivel de abstracción de la lógica del dominio, dependen de clases que tienen un nivel de abstracción mucho más bajo, puesto que tienen el nivel de abstracción de la capa de datos. El principio de Bajo Acoplamiento se incumple puesto que cada una de las clases de la capa de dominio están acopladas con las dos clases de la capa de datos. Para satisfacer el principio de Inversión de Dependencias añadimos una interfaz a la capa de dominio *BDAccess* y esta interfaz es implementada por las dos clases de la capa de datos. De este modo, cumpliremos con el principio de Inversión de Dependencias y, a la vez, reduciremos el acoplamiento.

El nuevo diagrama de clases será el siguiente:



Ejercicio 5 (15%)

Queremos definir la capa de presentación de nuestro sistema y para hacerlo aplicamos el patrón de arquitectura Modelo, Vista y Controlador (MVC). Propón un diagrama de clases donde se aplique el MVC para el caso de uso *Baja de un coche*. Este caso de uso permite al usuario indicar la opción de dar de baja un coche. En respuesta, el sistema le mostrará todas las matrículas de los coches que tiene almacenados. El usuario seleccionará la matrícula del coche que quiere dar de baja y el sistema lo dará de baja y le mostrará el mensaje "Car removed successfully". Por cada clase del diagrama de clases, define la signatura de las operaciones correspondientes al tratamiento en la capa de presentación desde que el sistema muestra las diferentes opciones (casos de uso) que ofrece y el usuario selecciona la opción de dar de baja un coche, hasta que el sistema da de baja el coche mostrando el mensaje. Explica qué hace cada operación.

Solución:

Aplicaremos el MVC a la capa de Presentación. La clase *RemoveCarView* representa la vista del patrón, la clase *RemoveCarPresentationController* representa el controlador y la clase *RemoveCarDomainController* representa el controlador de la capa de dominio.

A continuación, se describen las operaciones de cada clase:

Operaciones de la clase **RemoveCarView**:

- RemoveCarView::showOptions() muestra las diferentes opciones correspondientes a los diferentes casos de uso (entre ellos la baja de un coche) a la vista.
- RemoveCarView::setOption(option) registra el caso de uso seleccionado y lo notifica al controlador.
- RemoveCarView::showLicensePlates(licensePlates: Set(String)) muestra la vista con las diferentes etiquetas y en un desplegable las matrículas de los coches que hay almacenados en el sistema.
- RemoveCarView::setLicensePlate(licensePlate: String) registra la matrícula del coche a eliminar y lo notifica al controlador.
- RemoveCarView::showEndMessage() muestra la vista con el mensaje "Car removed succesfully".

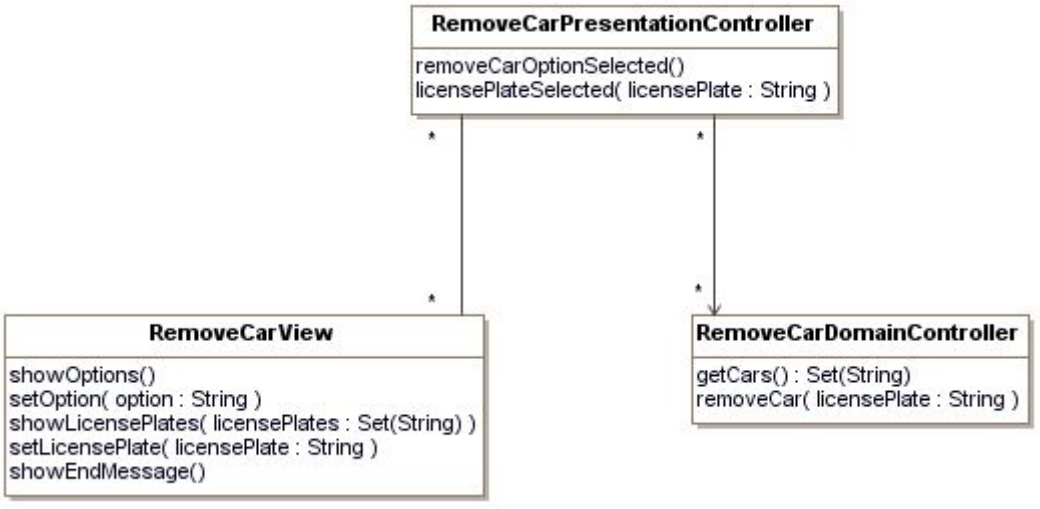
Operaciones de la clase **RemoveCarPresentationController**:

- RemoveCarPresentationController::removeCarOptionSelected() detecta que el usuario ha escogido la opción para dar de baja un coche e invoca a la operación.
- RemoveCarDomainController::getCars(): Set(String). La información recibida es notificada a la vista.
- RemoveCarPresentationController::licensePlateSelected(licensePlate: String) detecta que el usuario ha introducido la matrícula del coche a eliminar e invoca a la operación.

- RemoveCarDomainController::removeCar(licensePlate: String). Se notifica a la vista que puede mostrar el mensaje final.

Operaciones de la clase **RemoveCarDomainController**:

- RemoveCarDomainController::getCars(): Set(String) obtiene las matrículas de todos los coches que hay en el sistema.
- RemoveCarDomainController::removeCar(licensePlate: String) elimina el coche con la matrícula indicada como parámetro.
- RemoveCarView::showEndMessage() muestra la vista con el mensaje “Car removed succesfully”.



Ejercicio 6 (10%)

Propón el diagrama de secuencia de la interacción desde el usuario hasta el controlador de dominio para el caso de uso *Baja de un coche,* que acaba mostrando el mensaje "Car removed succesfully".

Solución: