Principios de diseño

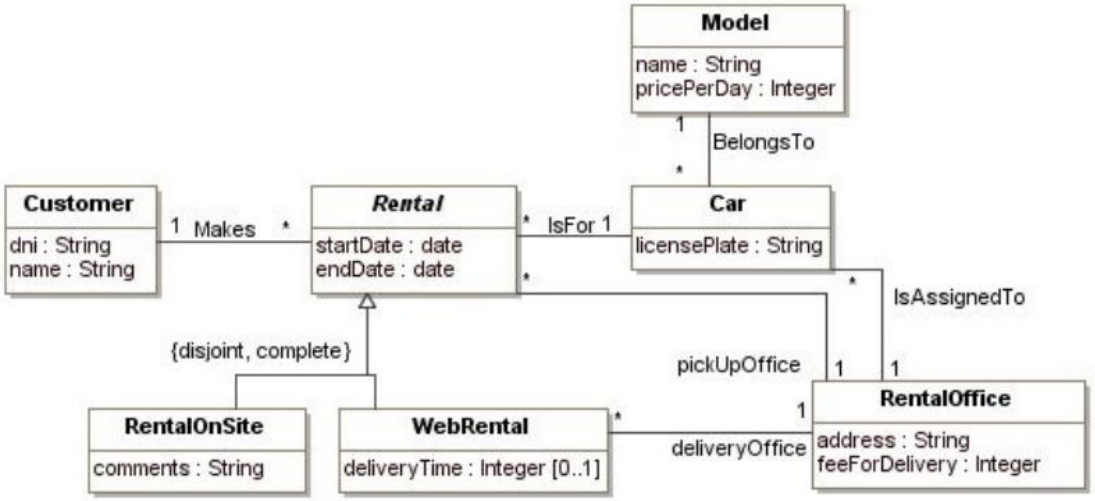
Práctica 2: Alquiler de coches

Una empresa de renting de coches nos ha contratado para desarrollar un software de gestión de alquileres. Los clientes pueden alquilar coches de diferentes modelos durante un periodo de tiempo. Los coches están asociados a una oficina de alquiler. Hay dos maneras de alquilar un coche: una de manera física (en la propia oficina) y otra vía web. Para los alquileres que se hacen en las oficinas se registra si el pago se ha hecho en efectivo o con tarjeta. 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 si el pago se ha hecho con tarjeta o con PayPal. En estos tipos de alquileres, los clientes pueden devolver los coches en una oficina distinta a donde lo han recogido pagando un cargo extra.

Hablando con el cliente nos recalca los siguientes requisitos:

- Los clientes van a tener un id y un nombre
- Los coches van a tener una matrícula además de un modelo. A cada modelo se le aplicará un precio por día distinto.
- Cada oficina tendrá una dirección y un cargo extra.
- Quiere que únicamente los alquileres vía web puedan dejar el coche en una oficina distinta a la de origen. En la que se le aplicará el cargo extra.

A pesar de haber hecho nuestro propio diseño UML, la empresa decide llevar a cabo el siguiente diseño:



A continuación, se muestra la implementación del método de la clase cliente que devuelve el total a pagar por todos los alquileres que ha hecho vía web y en los que ha dejado el coche en una oficina distinta a la de origen.

```
public class Customer {

  private List<Rental> rentals;

  public Integer getTotalAdditionalCharges(){
  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

Revisando el diagrama UML y el código del método getTotalAdditionalCharges() nos preocupa que su diseño sea un poco frágil y esté violando algunos principios de diseño. Identifica que principios está violando este diseño y justifícalo.

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.

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 una nueva condición para tratar esta nueva subclase.

Ejercicio 2

Propón una solución a los errores que has encontrado en el primer ejercicio. Realiza los cambios que consideres oportuno tanto en el UML como en el código. Puedes generar código de otras clases si lo ves necesario.

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

  public Integer getTotalAdditionalCharges(){
  Integer total = 0;
      foreach (Rental r in rentals){
                total += r.getAdditionalCharges();
       }
     }
  return total;
}

public abstract class Rental {

      private Date startDate;
      private Date endDate;
      private Car car;
      private RentalOffice pickUpOffice;

      public abstract Integer getAdditionalCharges();
}

public class WebRental extends Rental {

      private RentalOffice deliveryOffice;

      @Override
      public Integer getAdditionalCharges(){
            if (deliveryOffice != pickUpOffice){
                  return deliveryOffice.getFeeForDelivery();
            }
      }
}
```

```
public class RentalOnSite extends Rental {

	private String commnets;

	@Override
	public Integer getAdditionalCharges(){
		return 0;
	}
}

public class RentalOffice {

	private String address;
	private Integer feeForDelivery;

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