Universidad de Málaga

ETSI Informática

Patrones de diseño aplicados a un sistema de Alquiler de coches

Modelado y Diseño del Software (2024–25)

Daniil Gumeniuk
Angel Bayon Pazos
Diego Sicre Cortizo
Pablo Ortega Serapio
Angel Nicolás Escaño López
Francisco Javier Jordá Garay
Janine Bernadeth Olegario Laguit

Grupo 1.1

Diciembre 2024

Índice

1 Introducción **4**
- 1.1 Diagrama de Diseño . . . 4
- 1.2 Cambios respecto a la implementación original . . . 4
  - 1.2.1 Model . . . 4
  - 1.2.2 Car . . . 5
  - 1.2.3 Customer . . . 6
  - 1.2.4 Rental . . . 7
  - 1.2.5 RentalOnSite . . . 8
  - 1.2.6 WebRental . . . 8
  - 1.2.7 RentalOffice . . . 9

2 Ejercicio 1 **11**
- 2.1 Patrón de Diseño utilizado . . . 11
- 2.2 Efectos sobre el Diagrama de Diseño . . . 12
- 2.3 Implementación de *numberOfRentalsWithDifferentOffices():Integer* . . . 13
- 2.4 Ejemplo de ejecución . . . 14

3 Ejercicio 2 **16**
- 3.1 Patrón de Diseño utilizado . . . 16
- 3.2 Efectos sobre el Diagrama de Diseño . . . 17
- 3.3 Implementación de *takeOutOfService : date* . . . 20
- 3.4 Ejemplo de ejecución . . . 21

4 Ejercicio 3 **25**
- 4.1 Patrón de Diseño utilizado . . . 25
- 4.2 Efectos sobre el Diagrama de Diseño . . . 26
- 4.3 Implementación de *getPrice() : Integer* . . . 27
- 4.4 Ejemplo de ejecución . . . 29

Índice de figuras

1 Diagrama de Diseño del esqueleto 4
2 Diagrama de Diseño modificado para Ejercicio1 12
3 Diagrama de Diseño modificado para Ejercicio2 17
4 Diagrama de Diseño del apartado 3 26

Resumen

Esta práctica tiene como objetivo el diseño e implementación de un sistema de gestión de alquileres de coches para una empresa. El proyecto se centra en desarrollar las funcionalidades necesarias para gestionar el proceso de alquiler, las restricciones de negocio y las operaciones específicas requeridas asi como aplicar *Patrones de Diseño* presentados en el Tema 6.

Estructuramos el trabajo de tal forma que para cada uno de los apartados presentamos en esta memoria, se desarrolla y documenta un modelo UML hecho en *Visual Paradigm* detallado ajustándose a las necesidades de cada sección. También se justifican las posibles contradicciones o lagunas encontradas en el enunciado. Además, se explican las decisiones de diseño tomadas para la implementación del código en lenguaje *Java* respetando las restricciones impuestas por el enunciado.

Todas las implementaciones parten del mismo esqueleto que se presenta en la Introducción de la memoria, detallando en cada apartado las clases que se ven afectadas al implementar el *Patrón de Diseño* escogido (en caso que fuera necesario) y una explicación de como ese patrón enriquece el funcionamiento del sistema original.

Los metodos protected del sistema asumen que el usuario accedera al main fuera del paquete del código fuente.

1. Introducción

1.1. Diagrama de Diseño

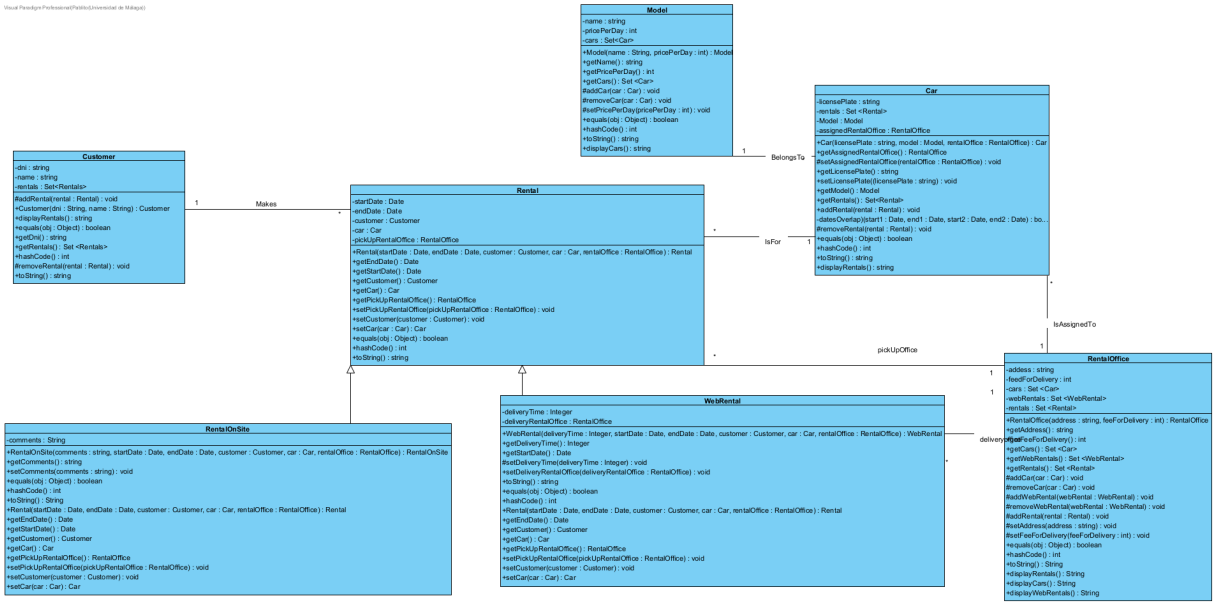

Figura 1: Diagrama de Diseño del esqueleto

1.2. Cambios respecto a la implementación original

1.2.1. Model

Atributos

- **name: String**: Nombre del modelo. Este valor es inmutable.
- **pricePerDay: int**: Precio por día asociado al modelo.
- **cars: Set<Car>**: Conjunto de coches asociados al modelo de Coche. Este conjunto es inmutable.

Operaciones

- **Model(name : String, pricePerDay : int) : Model**: Constructor que inicializa un modelo con su nombre y precio por día. Garantiza que el nombre no sea `null` y que el precio por día sea positivo.
- **getName() : String**: Devuelve el nombre del modelo.
- **getPricePerDay() : int**: Devuelve el precio por día del modelo.
- **getCars() : Set<Car>**: Devuelve el conjunto de coches asociados al modelo.
- **addCar(car : Car) : void**: Añade un coche al conjunto de coches del modelo.

- **removeCar(car : Car) : void**: Elimina un coche del conjunto de coches del modelo.
- **setPricePerDay(pricePerDay : int) : void**: Actualiza el precio por día del modelo.
- **equals(obj : Object) : boolean**: Compara dos modelos para determinar si son iguales basándose en su nombre.
- **hashCode() : int**: Genera un código hash basado en el nombre del modelo.
- **toString() : String**: Devuelve un string con el nombre del modelo, su precio por día y el conjunto de coches.
- **displayCars() : String**: Devuelve los detalles de todos los coches del modelo.

1.2.2. Car

Atributos

- **licensePlate: String**: Matrícula del coche.
- **rentals: Set<Rental>**: Conjunto de alquileres asociados al coche. Este conjunto es inmutable.
- **model: Model**: Modelo del coche. Este valor es inmutable.
- **assignedRentalOffice: RentalOffice**: Oficina de alquiler asignada al coche.

Operaciones

- **Car(licensePlate : String, model : Model, rentalOffice : RentalOffice) : Car**: Constructor que inicializa un coche con su matrícula, modelo y oficina de alquiler asignada.
- **getAssignedRentalOffice() : RentalOffice**: Devuelve la oficina de alquiler asignada al coche.
- **setAssignedRentalOffice(rentalOffice : RentalOffice) : void**: Actualiza la oficina de alquiler asignada al coche.
- **getLicensePlate() : String**: Devuelve la matrícula del coche.
- **setLicensePlate(licensePlate : String) : void**: Actualiza la matrícula del coche.
- **getModel() : Model**: Devuelve el modelo del coche.
- **getRentals() : Set<Rental>**: Devuelve el conjunto de alquileres del al coche.
- **addRental(rental : Rental) : void**: Añade un alquiler al conjunto de alquileres del coche.
- **removeRental(rental : Rental) : void**: Elimina un alquiler del conjunto de alquileres del coche.

- **equals(obj : Object) : boolean**: Compara dos coches para determinar si son iguales basándose en su matrícula.
- **hashCode() : int**: Genera un código hash basado en la matrícula del coche.
- **toString() : String**: Devuelve un string del coche mostrando su matrícula.
- **displayRentals() : String**: Devuelve los detalles de todos los alquileres del coche.

Validaciones y Consideraciones

- La relación entre un coche y su oficina de alquiler es bidireccional. Si se actualiza la oficina, también se actualiza en la oficina anterior y la nueva.
- Se verifica que las fechas de un nuevo alquiler no se solapen con las de los alquileres existentes, garantizando la coherencia.
- El conjunto `rentals` se expone como una colección inmutable para evitar modificaciones desde el exterior.

1.2.3. Customer

Atributos

- **dni: String**: Es el DNI del cliente, un identificador único. No se permite modificarlo después de inicializarlo.
- **name: String**: Es el nombre del cliente. No se permite modificarlo después de inicializarlo.
- **rentals: Set<Rental>**: Conjunto de todos los alquileres de cliente.

Operaciones

- **Customer(dni : String, name : String) : Customer**: Constructor que inicializa un cliente con un DNI y un nombre.
- **getDni() : String**: Devuelve el DNI del cliente.
- **getName() : String**: Devuelve el nombre del cliente.
- **getRentals() : Set<Rental>**: Devuelve el conjunto de alquileres asociados al cliente.
- **addRental(rental : Rental) : void**: Agrega un alquiler al conjunto de alquileres del cliente.
- **removeRental(rental : Rental) : void**: Elimina un alquiler del conjunto de alquileres del cliente.
- **equals(obj : Object) : boolean**: Compara dos clientes por DNI para determinar si son iguales.

- **hashCode() : int**: Genera un código hash basado en el DNI del cliente.
- **toString() : String**: Devuelve un string con el dni, nombre y lista de alquileres de un cliente.
- **displayRentals() : String**: Devuelve los detalles de todos los alquileres del cliente.

1.2.4. Rental

Atributos

- **startDate: Date**: Fecha del inicio del alquiler. Este valor es inmutable.
- **endDate: Date**: Fecha de la finalización del alquiler. Este valor es inmutable.
- **customer: Customer**: Cliente asociado al alquiler.
- **car: Car**: Coche asociado al alquiler.
- **pickUpRentalOffice: RentalOffice**: Oficina de recogida del coche.

Operaciones

- **Rental(startDate : Date, endDate : Date, customer : Customer, car : Car, rentalOffice : RentalOffice) : Rental**: Constructor que inicializa un alquiler con las fechas de inicio y fin, cliente, coche y oficina de recogida.
- **getStartDate() : Date**: Devuelve la fecha de inicio del alquiler.
- **getEndDate() : Date**: Devuelve la fecha de finalización del alquiler.
- **getCustomer() : Customer**: Devuelve el cliente asociado al alquiler.
- **getCar() : Car**: Devuelve el coche asociado al alquiler.
- **getPickUpRentalOffice() : RentalOffice**: Devuelve la oficina de recogida del coche.
- **setPickUpRentalOffice(rentalOffice : RentalOffice) : void**: Actualiza la oficina de recogida del coche.
- **setCustomer(customer : Customer) : void**: Actualiza el cliente asociado al alquiler.
- **setCar(car : Car) : void**: Actualiza el coche asociado al alquiler.
- **equals(o : Object) : boolean**: Compara dos alquileres para ver si son iguales mediante las fechas de inicio y fin, el cliente, y el coche.
- **hashCode() : int**: Genera un código hash basado en las fechas de inicio, fin, el cliente y el coche.
- **toString() : String**: Devuelve un string con la clase de alquiler, su fecha de inicio y su fecha de fin.

1.2.5. RentalOnSite

Atributos

- **comments: String**: Comentarios sobre el alquiler.

Operaciones

- **RentalOnSite(comments : String, startDate : Date, endDate : Date, customer : Customer, car : Car, rentalOffice : RentalOffice) : RentalOnSite**: Constructor que inicializa un alquiler presencial con comentarios opcionales, las fechas de inicio y fin, cliente, coche, y oficina de recogida.
- **RentalOnSite(startDate : Date, endDate : Date, customer : Customer, car : Car, rentalOffice : RentalOffice) : RentalOnSite**: Constructor alternativo que inicializa un alquiler presencial pero sin comentarios.
- **getComments() : String**: Devuelve los comentarios asociados al alquiler.
- **setComments(comments : String) : void**: Actualiza los comentarios asociados al alquiler.
- **equals(o : Object) : boolean**: Compara dos alquileres para determinar si son iguales, mediante los atributos heredados y los comentarios.
- **hashCode() : int**: Genera un código hash basado en los atributos heredados y los comentarios.
- **toString() : String**: Devuelve un string con los atributos heredados y los comentarios.

1.2.6. WebRental

Atributos

- **deliveryTime: Integer**: Tiempo de entrega del Choche en la Oficina correspondiente.
- **deliveryRentalOffice: RentalOffice**: Oficina de entrega del alquiler.

Operaciones

- **WebRental(deliveryTime : Integer, startDate : Date, endDate : Date, customer : Customer, car : Car, rentalOffice : RentalOffice) : WebRental**: Constructor que inicializa un alquiler web con tiempo de entrega, fechas de inicio y fin, cliente, coche, y oficina de entrega. La oficina de recogida es la misma que la del coche al crear el alquiler.
- **getDeliveryTime() : Integer**: Devuelve el tiempo de entrega asociado al alquiler.
- **setDeliveryTime(deliveryTime : Integer) : void**: Actualiza el tiempo de entrega.

- **setDeliveryRentalOffice(deliveryRentalOffice : RentalOffice) : void**: Actualiza la oficina de entrega asociada al alquiler.
- **equals(o : Object) : boolean**: Compara dos alquileres para determinar si son iguales, mediante atributos heredados, el tiempo de entrega y la oficina de entrega.
- **hashCode() : int**: Genera un código hash basado en los atributos heredados, el tiempo de entrega y la oficina de entrega.
- **toString() : String**: Devuelve un string con los atributos heredados y el tiempo de entrega.

1.2.7. RentalOffice

Atributos

- **address: String**: Dirección de la oficina de alquiler.
- **feeForDelivery: Integer**: Tarifa de entrega de la oficina de alquiler
- **cars: Set<Car>**: Conjunto de coches disponibles en la oficina de alquiler.
- **webRentals: Set<WebRental>**: Conjunto de alquileres web asociados a la oficina.
- **rentals: Set<Rental>**: Conjunto de alquileres asociados a la oficina.

Operaciones

- **RentalOffice(address : String, feeForDelivery : Integer) : RentalOffice**: Constructor que inicializa una oficina de alquiler con una dirección y una tarifa de entrega.
- **getAddress() : String**: Devuelve la dirección de la oficina de alquiler.
- **getFeeForDelivery() : Integer**: Devuelve la tarifa de entrega de la oficina.
- **getCars() : Set<Car>**: Devuelve el conjunto de coches disponibles en la oficina.
- **getWebRentals() : Set<WebRental>**: Devuelve el conjunto los alquileres web de la oficina.
- **getRentals() : Set<Rental>**: Devuelve un conjunto de los alquileres de la oficina.
- **addCar(car : Car) : void**: Añade un coche a la oficina.
- **removeCar(car : Car) : void**: Elimina un coche de la oficina.
- **addWebRental(webRental : WebRental) : void**: Añade un alquiler web a la oficina.
- **removeWebRental(webRental : WebRental) : void**: Elimina un alquiler web de la oficina.

- **addRental(rental : Rental) : void**: Añade un alquiler a la oficina.
- **removeRental(rental : Rental) : void**: Elimina un alquiler de la oficina.
- **setAddress(address : String) : void**: Actualiza la dirección de la oficina
- **setFeeForDelivery(feeForDelivery : Integer) : void**: Actualiza la tarifa de entrega de la oficina
- **equals(o : Object) : boolean**: Compara dos oficinas para ver si son iguales, utilizando la dirección.
- **hashCode() : int**: Genera un código hash basado en la dirección de la oficina.
- **toString() : String**: Devuelve un string mostrando la oficina, la dirección, la tarifa de entrega, los coches, los alquileres y los alquileres web.
- **displayCars() : String**: Devuelve un string de todos los coches disponibles en la oficina.
- **displayWebRentals() : String**: Devuelve un string de todos los alquileres web de la oficina.
- **displayRentals() : String**: Devuelve un string de todos los alquileres de la oficina.

2. Ejercicio 1

Instrucción

Supongamos ahora que queremos definir una operación
`numberOfRentalsWithDifferentOffices() : Integer` en la clase `Customer` que devuelve el número de alquileres web que ha hecho un cliente self donde la oficina de recogida y de entrega es diferente. En este momento del diseño del sistema todavía no sabemos qué estructura de datos utilizaremos para guardar los alquileres que ha hecho/hace un cliente.

2.1. Patrón de Diseño utilizado

El objetivo de incorporar la nueva operación es introducir al sistema una forma de recuperar información específica acerca de alquileres web. Como todos los alquileres son realizados por clientes, necesitaríamos verificar cada uno y encontrar todos aquellos que cumplan que su alquiler además de ser mediante la página, cumplan que la oficina de recogida entrega sea diferente.

La descripción inicial del sistema no detalla que esta operación sea la única que hace falta o es la primera que se implementa referente a los alquileres. Es posible que en un futuro se necesite información de clientes que hayan hecho más de un alquiler o se necesite recuperar información de cuantas veces un coche ha sido alquilado.

Pensando en todos los casos que se puedan solicitar, es ineficiente simplemente hacer un bucle que recorra toda la colección y filtrar por la condición necesaria si el sistema puede escalar. Es por ello que el patrón `Iterador` es el más adecuado para implementar esta solución.

Este patrón de diseño nos permite acceder secuencialmente a los elementos de una colección, separando la lógica de iteración del *cliente*[1] y centralizándola en el propio iterador, sin mencionar que permite escalar con facilidad el sistema. Otra ventaja de usar esta patrón es que no expone la estructura interna de la colección y permite ser usada por varios *clientes* simultáneamente.

Consideraciones

Como estamos utilizando un `HashSet` para almacenar los alquileres, somos consientes que esta colección ya implementa el método `iterator()` de la interfaz `Iterable`. No sería correcto usar este iterator por las mismas razones recién comentadas; el iterador nativo de `HashSet` simplemente recorre los elementos uno por uno, sin ningún tipo de lógica adicional. Como queremos filtrar `Rentals` para recuperar los que cumplan una condición específica, tiene sentido implementar un iterador personalizado con este patrón que encapsule esa lógica en particular.

[1] **¿Quiénes son los *clientes*?** Este término no se refiere ni a las clases del sistema ni a los usuarios del programa, sino a las partes del código que utilizan el iterador para recorrer una colección. En este caso, el *cliente* del iterador es el método `numberOfRentalsWithDifferentOffices()`.

2.2. Efectos sobre el Diagrama de Diseño

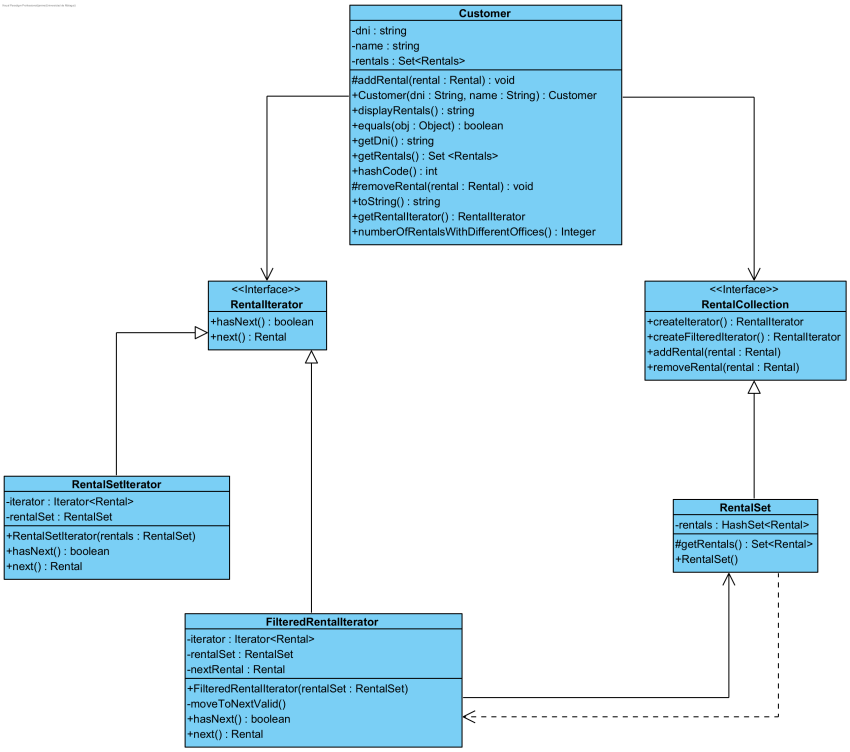

Figura 2: Diagrama de Diseño modificado para Ejercicio1

Hemos tenido que modificar el diseño original para poder implementar este patrón de diseño. Siguiendo la estructura presentada en los apuntes del Tema 6 en la página 108, tenemos:

- `Interfaz` **RentalIterator:** Define las operaciones necesarias para recorrer la colección de elementos.
- `Interfaz` **RentalCollection:** Implementa las operaciones necesarias para crear una colección iterable. Es una `IterableCollection`.
- `ConcreteCollection` **RentalSet:** Almacena los elementos y devuelve un iterador para recorrerlos además de poder gestionarlos y recuperarlos.
- `ConcreteIterator` **RentalSetIterator:** Contiene la lógica específica para moverse de uno en uno a través de la colección.
- `ConcreteIterator` **FilteredRentalIterator:** Contiene la lógica que recorre solo los elementos de tipo `WebRental` cumpla con la condición de tener una oficina de recogida y entrega distintas.

Estructurar el diseño de esta manera, se alinea con el enunciado ya que estamos abstrayendo el tipo de colección usada para guardar los alquileres que ha hecho/hace un cliente.

2.3. Implementación de *numberOfRentalsWithDifferentOffices():Integer*

```
public Integer numberOfRentalsWithDifferentOffices() {
    int count = 0;
    RentalIterator iterator = this.getRentals().
       createFilteredIterator();
    while (iterator.hasNext()) {
        Rental rental = iterator.next();
        count++;
    }
    return count;
}
```

Sigue el patrón de diseño implementado ya que se ha modificado la clase para que este *cliente* solo reciba el resultado del método sin exponer la lógica de la iteración. A continuación mostramos las operaciones que ocurren en segundo plano para devolver el resultado esperado del método.

1. *Cliente* (`Customer`) inicia el cálculo llamando al nuevo método.
2. `RentalSet` crea el `FilteredRentalIterator` en respuesta.
3. Este iterador, contiene la lógica filtrado para los elementos en la colección. Como se ve a continuación.

```
// Clase FilteredRentalIterator
            [...]
    private void moveToNextValid() {
        nextRental = null;
        while (iterator.hasNext()) {
            Rental rental = iterator.next();
            if (rental instanceof WebRental) {
                WebRental webRental = (WebRental) rental;
                if (!webRental.getPickUpRentalOffice().equals
                   (webRental.getDeliveryRentalOffice())) {
                    nextRental = rental;
                    break;
                }
            }
        }
    }
            [...]
```

4. *Cliente* utiliza el iterador para contar los elementos válidos.
5. El cliente recibe el número total de alquileres que cumplen el filtro.

2.4. Ejemplo de ejecución

Código del tester

```
// Crear cliente
Customer customer2 = new Customer("87654321B", "Bob");
// Crear modelos de coches
Model modelA = new Model("Model A", 50);
Model modelB = new Model("Model B", 70);
// Crear oficinas de alquiler
RentalOffice office1 = new RentalOffice("Office 1", 20);
RentalOffice office2 = new RentalOffice("Office 2", 25);
// Crear coches
Car car1 = new Car("ABC-123", modelA, office1);
Car car2 = new Car("XYZ-789", modelB, office2);
// Crear fechas
Date startDate1 = new GregorianCalendar(2024, Calendar.
   JANUARY, 1).getTime();
Date endDate1 = new GregorianCalendar(2024, Calendar.JANUARY,
    10).getTime();
Date startDate2 = new GregorianCalendar(2024, Calendar.
   JANUARY, 5).getTime();
Date endDate2 = new GregorianCalendar(2024, Calendar.JANUARY,
    15).getTime();

// Oficinas diferentes
WebRental webRental1 = new WebRental(5, startDate1, endDate1,
    customer2, car1, office1);
webRental1.setDeliveryRentalOffice(office2);

// Oficinas iguales
WebRental webRental2 = new WebRental(2, startDate1, endDate1,
    customer2, car2, office2);
webRental2.setDeliveryRentalOffice(office2);

// Usar nueva operacion implementada
int rentalsWithDifferentOffices = customer2.
   numberOfRentalsWithDifferentOffices();

System.out.println("Alquileres con oficinas diferentes: " +
   rentalsWithDifferentOffices);

if (rentalsWithDifferentOffices == 1) {
    System.out.println("Correcto: El metodo devuelve el
       numero esperado.");
} else {
    System.out.println("ERROR: El metodo devuelve un
       resultado incorrecto.");
}
```

Output

```
[WebRental: Mon Jan 01 00:00:00 CET 2024 Wed Jan 10 00:00:00
   CET 2024
; 2, WebRental: Mon Jan 01 00:00:00 CET 2024 Wed Jan 10
   00:00:00 CET 2024
; 5]

Alquileres con oficinas diferentes: 1
Correcto: El metodo devuelve el numero esperado.
```

Los `toString()` usados para mostrar el Output fueron:

```java
// en Rental
@Override
public String toString() {
    return this.getClass().getName() + ": " + startDate + " "
        + endDate+"\n";
}

// en WebRental, que llama al de Rental mediante super
@Override
public String toString() {
    return super.toString() + " ; " + deliveryTime.toString()
       ;
}
```

3. Ejercicio 2

Instrucción

En este apartado se desea implementar la siguiente operación:

`takeOutOfService(backToService : date)`: pone un coche que está en servicio fuera de servicio, registra la fecha hasta la que el coche permanecerá fuera de servicio y busca un coche para sustituir al que se pone fuera de servicio. El sustito debe cumplir una serie de restricciones:

- Debe estar asignado a la misma oficina que el coche sustituido.
- Debe ser del mismo modelo.
- Finalmente, debe estar en servicio.

Además, se menciona la funcionalidad que pone un coche fuera de servicio en servicio. Sin embargo, no se pide su implementación

3.1. Patrón de Diseño utilizado

Para poner un coche fuera de servicio hay que considerar varios factores.

Por un lado, el coche que está fuera de servicio no puede ser puesto fuera de servicio de nuevo.

Por otro lado, si un coche está en servicio, pero es sustituto del otro coche, tampoco puede ponerse fuera de servicio.

Un coche que está en servicio, puede ser alquilado, mientras que el coche que está fuera de servicio o que es el sustituto del otro coche, tampoco puede ser alquilado.

Como se puede apreciar, el comportamiento del coche varía en función de su estado. Por eso, hemos considerado usar el `Patrón Estados`dado que nos permite definir el comportamient del coche de manera flexible sin cambiar la estructura interna de la clase.

En nuestro caso, un coche puede estar en uno de los dos posibles estados:

- `En servicio`: el coche en este estado está disponible para ser alquilado.
- `Fuera de servicio`: el coche en este estado está fuera de servicio por varias razones (reparación, ITV, etc.).

3.2. Efectos sobre el Diagrama de Diseño

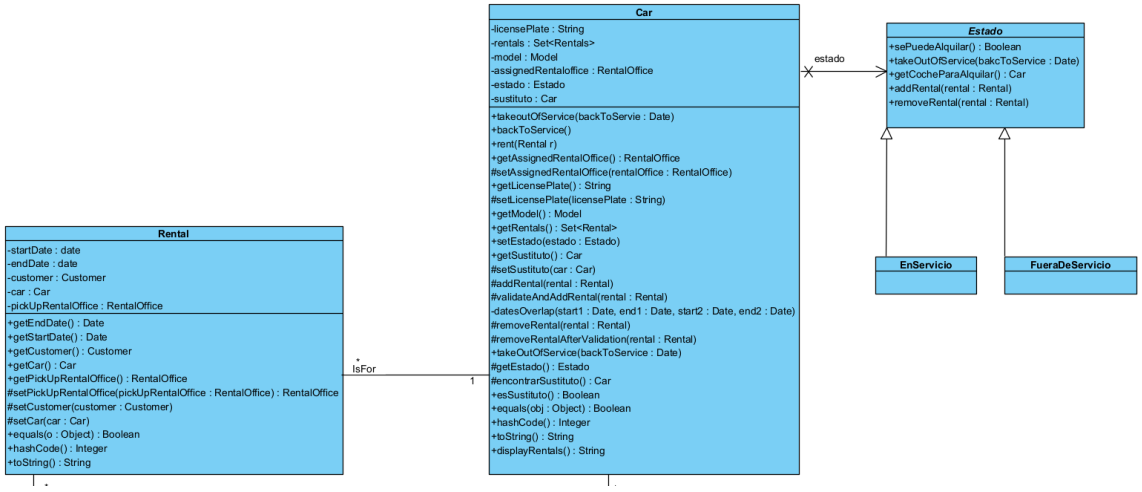

Figura 3: Diagrama de Diseño modificado para Ejercicio2

Con el fin de implementar el patrón escogido hemos tenido que insertar una clase abstracta `Estado` que representa el estado de coche. Además, hemos tenido que insertar dos clases que heredan de la clase mencionada:

- `EnServicio`: la instancia de la clase Car con estado de este tipo está en servicio. Este tipo permite poner un coche fuera de servicio y alquilarlo
- `FueraDeServicio`: la instancia de la clase Car con estado de este tipo está fuera de servicio. No permite alquilar un coche, pero permite alquilar el sustituto asociado, si es que existe.

Métodos adicionales de la calse `Car`:

- `protected void validateAndAddRental(Rental rental)`: es un método de la calse `Car` que antes de asignar un alquiler comprueba si no se solapa on ningún otro que ya está asignado al coche. Para realizar esta comprobación se usa un método privado `datesOverlap`

```
protected void validateAndAddRental(Rental rental) { //
    Metodo que solo llama el estado, por lo que estamos
    delegandole la decision
  assert rental != null : "Rental no puede ser null";
  assert !esSustituto() : "El coche esta siendo usado
     como sustituto";
  for (Rental existingRental : rentals) {
      assert !datesOverlap(existingRental.getStartDate
         (), existingRental.getEndDate(),
          rental.getStartDate(), rental.getEndDate()) :
              "El alquiler se solapa con otro ya
             existente.";
```

```
        }
        rentals.add(rental);
    }
```

- `protected Car encontrarSustituto()`: una vez que el coche se pone fuera de servicio, se usa este método para encontrar sustituto en función de los p usa en la redefinición del método `takeOutOfService(Date backToService)` en la subclase `EnServicio`, esta parte se trata más adelante.

```
    protected Car encontrarSustituto(){
      return assignedRentalOffice.getCars().stream()
              .filter(car -> car.getModel().equals(this.
                 getModel()) && car.getEstado().
                 sePuedeAlquilar() && !car.esSustituto())
              .findFirst().orElse(null);

    }
```

- `public boolean esSustituto()`: comprueba si el coche actual es sustituto o no. Esta información es necesaria a la hora de asignar un alquiler, dado que si un coche se usa como sustituto no puede ser alquilado como un coche normal.

```
    public boolean esSustituto(){
      Iterator<Car> iter = this.getModel().getCars().
         iterator();
      while(iter.hasNext()){
          Car curr = iter.next();
          if(curr.getSustituto() != null && curr.
             getSustituto().equals(this))
           return true;
      }
      return false;
    }
```

Métodos de la clase `Estado`:

- `public abstract boolean sePuedeAlquilar()`: indica si se puede alquilar un coche o si no se puede alquilar un coche. Su implementación en la clase `EnServicio` devuelve `verdadero`, mientras que la de la clase `FueraDeServicio` devuelve `falso`. Cabe destacar su implementación en la clase `EnServicio`.

```
    public boolean sePuedeAlquilar(){
      return !context.esSustituto();
    }
```

  Como se puede observar, usamos el método `esSustituto()`. Esta implementación, la consideramos correcta dado, que la única condición para que un coche en servicio no se pueda alquilar es que sea un sustituto.

```
    public boolean sePuedeAlquilar(){  return false;}
```

En la clase `FueraDeServicio` únicamente se devuelve `false` porque el hecho de que un coche ya esté fuera de servicio es suficiente para que no se pueda alquilar.

- `public Car getCocheParaAlquilar()`: es un método que devuelve la instancia del coche o el sustituto en función del estado del coche. Al igual que los otros métodos, este tiene dos verisones: una para la clase `EnServicio`, otra para la clase `FueraDeServicio`. Empezamos con la primera:

```
public Car getCocheParaAlquilar() {
    return context; // Devuelve el coche original
        porque esta en servicio
 }
```

Si un coche está en servicio entonces se devuelve la instancia actual.

```
public Car getCocheParaAlquilar() {
if (context.getSustituto() != null) {
    return context.getSustituto(); // Devuelve el
        sustituto si existe
}
throw new IllegalStateException("El coche esta fuera
    de servicio y no tiene sustituto");
// Devuelve el coche original porque esta en servicio
}
```

Sin embargo, si un coche está fuera de servicio entonces se comprueba si tiene sustituto. En el caso afirmativo, se devuelve el sustituto. En el caso contrario, se lanza una excepción con el mensaje correspondiente.

- `public boolean addRental(Rental rental)`: añade alquiler al propio coche si este está en servicio o al sustituto, si está fuera de servicio y tiene uno.

Si un coche **está en servicio**, entonces añadimos el nuevo alquiler verificando que es no se solapa con ninguno de los otros asociados al coche.

```
public void addRental(Rental rental) {
    context.validateAndAddRental(rental);
}
// Metodo de la clase Car
protected void validateAndAddRental(Rental rental) { //
    Metodo que solo llama el estado, por lo que estamos
    delegandole la decision
    assert rental != null : "Rental no puede ser null";
    assert !esSustituto() : "El coche esta siendo usado
        como sustituto";
    for (Rental existingRental : rentals) {
        assert !datesOverlap(existingRental.getStartDate
            (), existingRental.getEndDate(),
             rental.getStartDate(), rental.getEndDate()) :
                 "El alquiler se solapa con otro ya
                    existente.";
    }
```

```
        rentals.add(rental);
    }
```

Si estuviera **fuera de servicio**, entonces añadimos el nuevo alquiler al sustituto, si existe, o lanzamos una excepción.

```
     public void addRental(Rental rental) {
    if (this.context.getSustituto() != null) {
        context.getSustituto().addRental(rental);
    } else {
        throw new IllegalStateException("El coche esta
           fuera de servicio y no tiene sustituto");
    }
}
```

- `public boolean removeRental(Rental rental)`: sigue la misma lógica que el método `addRental(Rental rental)`, pero en vez de añadir el alquiler, este se elimina.

  Si el coche está en servicio entonces se elimina el alquiler.

```
      public void removeRental(Rental rental) {
   context.removeRentalAfterValidation(rental);
}
```

  Sin embargo, si el coche está fuera de servicio entonces, o bien se elimina el alquiler del sustituto o se lanza un error si no existe.

```
    public void removeRental(Rental rental) {
     if (this.context.getSustituto() != null){
        context.getSustituto().removeRental(rental);
     }else{
        throw new IllegalStateException("El coche esta
           fuera de servicio y no tiene sustituto");
     }
}
```

3.3. Implementación de *takeOutOfService : date*

Hemos decidido delegar el método `takeOutOfService(Date backToService)` a las subclases de la clase Estado dado que el resultado de la operación depende del estado actual del coche

```
//Implementacion en la clase Car:
    public void takeOutOfService(Date backToService){
    /* assert estado instanceof  EnServicio : "El coche
       ya esta fuera de servicio";
    // Registramos la cuando el coche vuelva en servicio
       */
    assert backToService != null;
```

```
            estado.takeOutOfService(backToService);
        }
        // Implementacion en la clase EnServicio:
         public void takeOutOfService(Date backToService) {
          assert !context.esSustituto(): "El coche es el
             sustituto del otro";                          // "El
             coche es sustituto del otro coche";
          context.setEstado(new FueraDeServicio(context,
             backToService));       // Ponemos un coche fuera de
             servicio
          context.setSustituto(context.encontrarSustituto());
                    // Buscamos un coche sustituto

       }
         // Implementacion en la clase FueraDeServicio:
       public void takeOutOfService(Date backToService) {
           throw new IllegalStateException("El coche ya se
              encuentra fuera de Servicio");
        }
```

En la clase `Car` lo único que se hace es delegar la operación de poner el coche fuera del servicio al estado actual de la instancia. Si el coche está en servicio, lo primero que tenemos que hacer es comprobar que no es sustituto de ningún otro coche. Si lo es, no se puede poner fuera de servicio. Si el coche no es sustituto, se pone fuera de servicio y se busca un sustituto.Si el coche ya está fuera de servicio, entonces se lanza una excepción del tipo `IllegalStateException ()`, indicando que no se puede poner fuera de servicio un coche que ya está fuera de servicio

3.4. Ejemplo de ejecución

```
    import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.*;

public class TestA2 {
    public static void main(String[] args) throws ParseException
       {
        // Crear modelos de coches
        Model modelA = new Model("Model A", 50);

        // Crear oficinas de alquiler
        RentalOffice office1 = new RentalOffice("Office 1", 20);

        // Crear coches
        Car car1 = new Car("ABC-123", modelA, office1);
        Car car2 = new Car("DGJ-324", modelA, office1);
        Car car3 = new Car("XYZ-789", modelA, office1);
```

```
// Ponemos car1 fuera de Servicio:
 SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
    HH:mm:ss");
 Date fecha1 = sdf.parse("2025-01-01 12:00:00");


 //Ponemos car1 fuera de servicio otra vez. No deberia
    permitirlo

 car1.takeOutOfService(fecha1); // El coche estara fuera
    de servicio hasta 1 de enero del 2025 y se le asocia el
     car2

 //Ponemos el car1 fuera de servicio otra vez. Esta vez no
     lo permite
 //car1.takeOutOfService(fecha1);

 //Ponemos el car2 fuera de servicio. No deberia
    permitirlo, dado que car2 es sustituto de car1

 //car2.takeOutOfService(fecha1); // El coche estara fuera
     de servicio hasta 1 de enero del 2025

 //Ponemos car3 fuera de servicio: No deberia de tener un
    sustituto

 car3.takeOutOfService(sdf.parse("2025-01-12 12:00:00"));

 //Intentamos alquilar un coche que esta fuera de servicio
    , pero que tiene un sustituto

 Customer c1 = new Customer("12345H", "Fulanito");

 Date startDate = sdf.parse("2024-01-14 12:00:00");
 Date endDate = sdf.parse("2024-01-21 12:00:00");

 Rental rent1 = new RentalOnSite(startDate, endDate, c1,
    car1, office1); // No funciona bien

 Date startDate2 = sdf.parse("2024-01-14 12:30:00");
 Date endDate2 = sdf.parse("2024-01-21 12:30:00");

 //Ahora vamos a intentar alquilar el coche 2 que esta
    siendo sustituto del coche 1

 //Descomentar esta linea para comprobar el alquiler de un
     coche sustituto
```

```
        //Rental rent2 = new RentalOnSite(startDate2, endDate2,
            c1, car2, office1);



        // Intentamos alquilar un coche que esta fuera de
            servicio y que not tiene sustituto:

        Date startDate3 = sdf.parse("2024-01-14 12:30:00");
        Date endDate3 = sdf.parse("2024-01-21 12:30:00");
        //Falla por intentar alquilar un coche fuera de servicio
        //Rental rent3 = new RentalOnSite(startDate3, endDate3,
            c1, car3, office1);

        System.out.println(rent1.toString());



    }
}
```

Output

Caso 1:

Fulanito DGJ-324 ; null

Caso 2:

```
Exception in thread "main" java.lang.AssertionError: La fecha de
    inicio no es valida
        at Rental.<init>(Rental.java:20)
        at RentalOnSite.<init>(RentalOnSite.java:12)
        at TestA2.main(TestA2.java:54)
```

Vemos en este caso un error debido a que estamos intentando solapar fechas.

Caso 3:

```
Exception in thread "main" java.lang.IllegalStateException: El
    coche esta fuera de servicio y no tiene sustituto
        at FueraDeServicio.getCocheParaAlquilar(FueraDeServicio.
            java:25)
        at Rental.<init>(Rental.java:23)
        at RentalOnSite.<init>(RentalOnSite.java:12)
        at TestA2.main(TestA2.java:64)
```

Vemos que aqui tenemos un error debido a que el coche que estamos intentando asociar se encuentra fuera de servicio.

Caso 4:

```
Exception in thread "main" java.lang.AssertionError: El coche ya
   esta siendo usado como sustituto.
  at Rental.<init>(Rental.java:21)
  at RentalOnSite.<init>(RentalOnSite.java:12)
  at TestA2.main(TestA2.java:54)
```

Vemos que falla porque se esta intentando alquilar un coche sustituto.

4. Ejercicio 3

Instrucción

Cuando los clientes de la empresa de alquiler de coches alquilan un coche, el sistema que estamos construyendo tiene que proporcionarles el precio del alquiler. Este precio lo calcula la operación `getPrice() : Integer` de la siguiente forma: El precio base será el precio del modelo del vehículo por día.

```
(pricePerDay) * [endDate - startDate]
```

Además, la empresa de alquiler de coches puede añadir al cálculo de precios la posibilidad de hacer promociones que implican descuentos de estos precios. Inicialmente, la empresa ofrecerá dos tipos de promociones: por cantidad y por porcentaje.

- **Promoción por cantidad:** permitirá decrementar el precio del alquiler en la cantidad indicada en la promoción.
- **Promoción por porcentaje:** decrementará el precio del alquiler en el porcentaje indicado en la promoción. Las promociones se asignan a los alquileres en el momento de su creación.

Evidentemente, es posible que a algunos alquileres no se les aplique ninguna promoción. Las promociones que se asignan a los alquileres son determinadas por una política de la empresa que no impacta al diseño de nuestra operación (impactará a la operación que crea los alquileres).

Eso sí, la empresa quiere que mientras no se haga el pago del alquiler, si aparecen nuevas promociones, se apliquen a los alquileres siempre y cuando sean más favorables (no nos tenemos que preocupar tampoco de estos cambios, son gestionados por otras operaciones).

4.1. Patrón de Diseño utilizado

Para poder realizar este apartado optamos varias casuísticas de diferentes patrones de diseño validos. Por un lado optamos por el patron visitante, dicho patron complicaría demasiado la implementación de la operación debido al gran numero de instancias presentes de dicho patron para un unico método,esto hace que el uso de este patron sea algo ineficiente. Tras ese planteamiento llegamos a la conclusion de utilizar el patron **Strategy** para poder implementar la operación correctamente. Este patron se basa en la utilización de una interfaz que actúa como padre de todos los objetos que implementen una operación similar, en este caso tenemos dos tipos de promociones, **Promoción por porcentaje** y **Promoción por cantidad** ambas mantienen una lógica similar por lo que el uso del patron **Strategy** es el adecuado.

4.2. Efectos sobre el Diagrama de Diseño

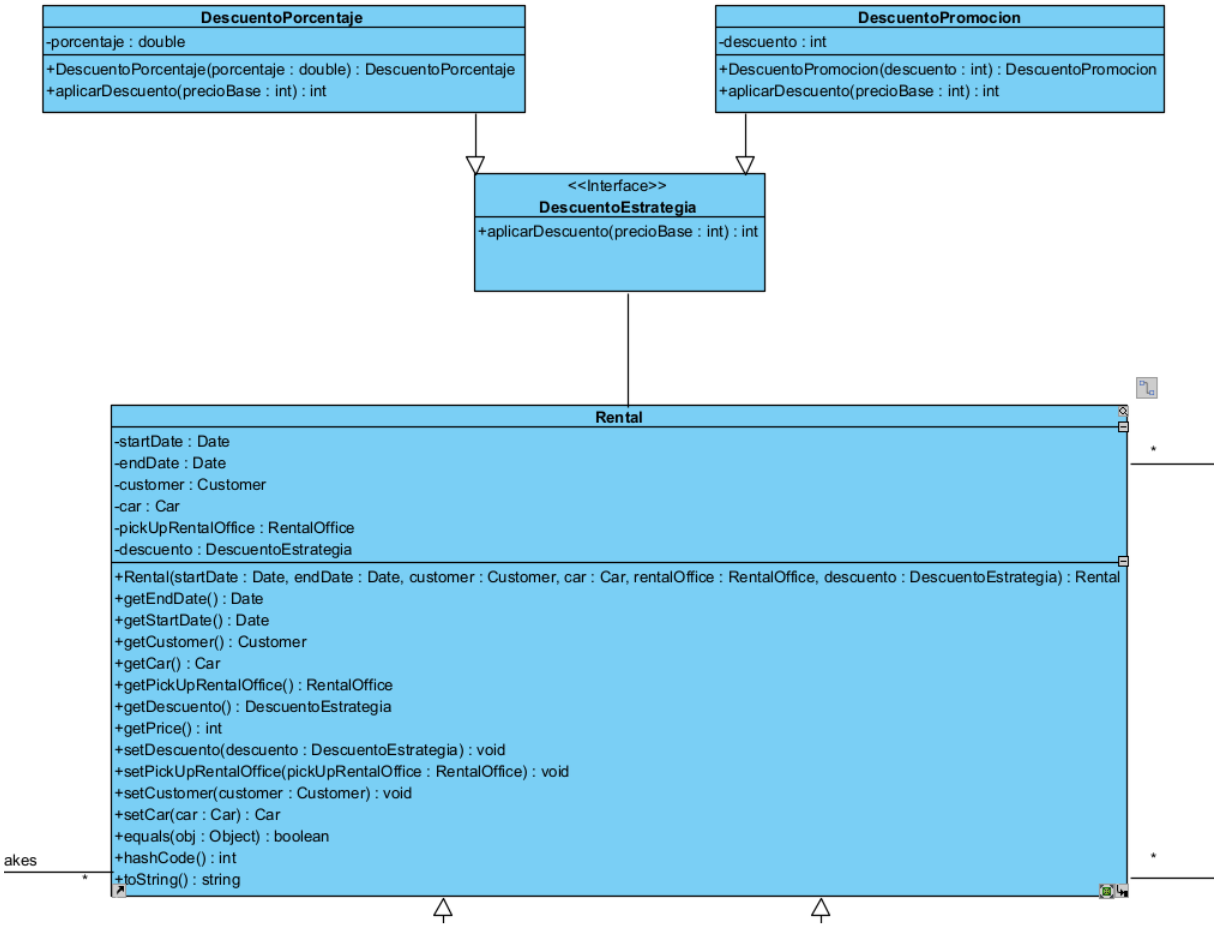

Figura 4: Diagrama de Diseño del apartado 3

Como podemos observar, el patron **Strategy** mantiene una interfaz padre de la cual heredan todas las instancias de objetos que vayan a realizar dicha operación. Como bien hemos dicho antes nos basamos en un sistema con dos operaciones diferentes para obtener el precio final, por lo cual necesitaremos crear dos clases una para cada tipo de promoción. Estas heredaran de la clase **Strategy** para asi obligar al sistema a implementar dichas operaciones en cada una de las clases. Un requisito del sistema es que dichas promociones se apliquen al instanciar un objeto **Rental**, es por eso que dichos tipos de promociones vendrán instanciados en el constructor de dicha clase pudiendo ser nulo en caso de que no se le aplique ningún tipo de promoción.

4.3. Implementación de *getPrice() : Integer*

```
//INTERFAZ GENERAL
public interface DescuentoEstrategia {
//Metodo para aplicar un descuento
int aplicarDescuento(int precioBase);
}
//TIPO DE DESCUENTO 1 POR PORCENTAJE
public class DescuentoPorcentaje implements
   DescuentoEstrategia{
    private final double porcentaje;

    public DescuentoPorcentaje(double porcentaje) {
        assert porcentaje >= 0 && porcentaje <= 100 : "
           Porcentaje invalido";
        this.porcentaje = porcentaje;
    }

    @Override
    public int aplicarDescuento(int precioBase) {
        double res = precioBase * (1-porcentaje/100);
        int sol = (int)res;
        return sol;
    }

    @Override
    public String toString() {
        return "[" + porcentaje + "]";
    }
}
//TIPO DE DESCUENTO 2 POR DESCUENTO
public class DescuentoPromocion implements
   DescuentoEstrategia{
    private final int descuento;

    public DescuentoPromocion(int descuento) {
        assert descuento >= 0 : "Descuento no puede ser
           negativo";
        this.descuento = descuento;
    }

    @Override
    public int aplicarDescuento(int precioBase) {
        if(descuento >= precioBase) return 0;
        return precioBase - descuento;
    }

    @Override
    public String toString() {
```

```
            return "[" + descuento + "]";
        }
    }

    //NUEVO METODO PARA OBTENER PRECIO CON DESCUENTO SI EXISTE
    public int getPrice(){
        LocalDate localStartDate = startDate.toInstant().atZone(
           ZoneId.systemDefault()).toLocalDate();
        LocalDate localEndDate = endDate.toInstant().atZone(
           ZoneId.systemDefault()).toLocalDate();
        double diasRental = ChronoUnit.DAYS.between(
           localStartDate, localEndDate);
        int precioBase = (int) (diasRental * car.getModel().
           getPricePerDay());
        if(this.descuento != null) return descuento.
           aplicarDescuento(precioBase);
        return precioBase;
    }
```

En la primera sección podemos encontrar las nuevas instancias que debemos generar para poder usar el patron **Strategy** dichas clases implementan el método de la interfaz de manera similar pero cambiando ligeramente la forma en la que calculan los precios a dependiendo de sus propios parámetros privados (porcentaje o cantidad).

Para implementar el método getPrice hemos decidido generar a partir de los parámetro **startDate** y **endDate** el precio base que tendría el alquiler antes de aplicar cualquier tipo de promoción. Como bien hemos explicado antes en caso de que no exista promoción alguna se instanciará como nulo el atributo en el constructor. Es por eso que antes de calcular el nuevo precio comprobamos si el **Rental** contiene un tipo de promoción disponible, en caso contrario devolveremos el precio base inicial.

4.4. Ejemplo de ejecución

Código del tester

```
//Comprobar que los precios han cambiado en base a la
   promocion dada en el sistema
System.out.println("Precio tras aplicar un descuento del " +
   rental1.getDescuento().toString() + " del tipo " + rental1.
   getDescuento().getClass().getName() + ": " + rental1.
   getPrice() + "\n");
System.out.println("Precio tras aplicar un descuento del " +
   rental2.getDescuento().toString() + " del tipo " + rental2.
   getDescuento().getClass().getName() + ": " + rental2.
   getPrice() + "\n");
```

Output

```
Precio tras aplicar un descuento del [50.0] del tipo
   DescuentoPorcentaje: 225

Precio tras aplicar un descuento del [100] del tipo
   DescuentoPromocion: 400
```

Los `toString()` usados para mostrar el Output fueron:

```
// en DescuentoPromocion
@Override
public String toString() {
    return "[" + descuento + "]";
}

// en DescuentoPorcentaje
@Override
public String toString() {
    return "[" + porcentaje + "]";
}
```