Universidad de Málaga

ETS Informática

SISTEMA DE ALQUILER DE COCHES

Artina Aradnasab
Soraya Bennai Sadqi
Luis Alberto Castaño Quero
María Victoria Huesca Peláez
Marcos Luque Montiel
Francisco Ramírez Cañadas
Juan José Serrano España

Grupo 3-5

ÍNDICE

INTRODUCCIÓN

En esta práctica nos hemos focalizado en diseñar un sistema para gestionar una empresa de alquiler de coches, permitiendo resolver tareas como alquileres, controlar la disponibilidad de los vehículos y aplicar promociones. Cada ejercicio lo resolvemos encontrando los patrones de diseño que mejor se adaptan a lo que se pide.

Para resolver estas cuestiones, usamos patrones de diseño como Iterador, Estado y Estrategia, siguiendo lo que vimos en el Tema 6, donde se explican dichos patrones como formas de solucionar problemas en el desarrollo de software.

Aquí presentamos las soluciones a los ejercicios de la práctica, explicando por qué escogimos cada patrón, cómo modificamos el diagrama de clases y cómo se implementa el código.

EJERCICIO 1

Apartado A

Para diseñar la operación numberOfRentalsWithDifferentOffices() hemos decidido usar el patrón de diseño **Iterator** ya que en este momento del diseño del sistema todavía no sabemos qué estructura de datos utilizaremos para almacenar los alquileres que ha hecho/hace un cliente.

El patrón **Iterator** es útil en este caso, porque permite recorrer cualquier tipo de colección de manera uniforme. Así, si más adelante cambiamos la estructura donde guardamos los alquileres, mientras el iterador siga funcionando, no hará falta modificar el resto del código.

Lo hemos implementado de la siguiente forma:

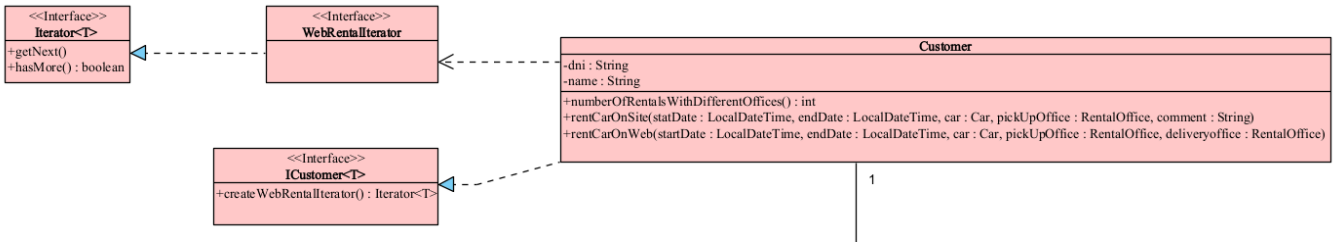
- Interfaz **ICustomer<T>**: En el patrón Iterator, la clase Customer juega el rol de agregado, el cual define una interfaz para crear un iterador (createWebRentalIterator), asegurando que cualquier clase que la implemente proporcione un método para crear un iterador sobre una colección de objetos de tipo T.

- Clase **Customer**: En el patrón Iterator, la clase Customer juega el rol de agregado concreto, el cual implementa la interfaz del agregado (ICustomer) y devuelve una instancia del iterador concreto. En particular, la clase Customer implementa el método createWebRentalIterator, que devuelve una instancia de WebRentalIterator. Este iterador permite recorrer la lista de alquileres web (webRentals) del cliente.

- Interfaz **Iterator<T>**: La interfaz Iterator define los métodos necesarios para recorrer una colección de objetos.
- Clase **WebRentalIterator**: El iterador WebRentalIterator implementa la interfaz Iterator y permite recorrer la lista de alquileres web. Su función es proporcionar una forma de recorrer la colección de alquileres web (WebRental) de un cliente sin exponer la representación subyacente de la colección.

Apartado B

El diagrama de clases tras aplicar el patrón Iterator quedaría de la siguiente forma:



El diagrama muestra cómo se usa el patrón Iterator para recorrer los alquileres de un cliente de forma fácil y ordenada. La interfaz Iterator<T> define las funciones básicas para moverse por la lista, y WebRentalIterator se encarga de manejar específicamente los alquileres web. La clase Customer puede crear este iterador gracias a la interfaz ICustomer<T>, y usa el método numberOfRentalsWithDifferentOffices() para contar cuántos alquileres tienen oficinas de recogida y entrega diferentes.

Apartado C

Operación numberOfRentalsWithDifferentOffices() de la clase Customer:

```
/**
     * Devuelve el número de alquileres web de un cliente en los que
difieren la oficina de entrega y devolución del coche
     */
    public Integer numberOfRentalsWithDifferentOffices() {
        int count = 0;
        Iterator<WebRental> iterator = createWebRentalIterator();
        while (iterator.hasNext()) {
            WebRental rental = iterator.next();
            if
(!rental.getpickUpOffice().equals(rental.getDeliveryOffice())) {
                count++;
            }
        }
        return count;
    }
```

Este código cuenta cuántos alquileres web de un cliente tienen oficinas de recogida y entrega diferentes. Primero se crea un iterador (WebRentalIterator) para recorrer todos los alquileres web del cliente. Luego, con un bucle while, se va revisando cada alquiler usando hasNext() para ver si hay más elementos y next() para obtener el siguiente. Si la oficina donde se recogió el coche no es la misma donde se entregó, se suma 1 al contador (count). Al final, devuelve ese número total. Básicamente, revisa cada alquiler y va contando los que cumplen la condición.

RESTO DEL CÓDIGO:

Clase Car

```
import java.time.*;
import java.util.*;

public class Car {
    private String licensePlate;

    private Model model;
    private RentalOffice assignedOffice;
    private List<Rental> rentals;

    public Car(String licensePlate, Model model, RentalOffice
assignedOffice) {
        assert(licensePlate != null && model != null && assignedOffice !=
null);
```

```
        this.licensePlate = licensePlate;
        this.model = model;
        model.getCars().add(this);
        this.assignedOffice = assignedOffice;
        this.rentals = new LinkedList<Rental>();
        this.assignedOffice.getCars().add(this);
        System.out.println("El coche con matrícula " + licensePlate + " se
ha creado correctamente");
    }

//-------------------- GETTERS -------------------------

    private String getLicensePlate() {
        return licensePlate;
    }

    protected List<Rental> getRental(){
        return rentals;
    }

    private Model getModel() {
        return model;
    }

    protected RentalOffice getAssignedOffice() { //es protegido porque
accedemos a él en otra clase
        return assignedOffice;
    }

//-------------------- SETTERS -------------------------

    private void setLicensePlate(String licensePlate) {
        assert(licensePlate != null);
        this.licensePlate = licensePlate;
    }

    private void setModel(Model model) {
        assert(model != null);
        this.model = model;
    }

    private void setAssignedOffice(RentalOffice assignedOffice) {
        assert(assignedOffice != null);
```

```
        this.assignedOffice = assignedOffice;
    }

    private void setRentals(List<Rental> rentals){
        assert(rentals != null);
        this.rentals = rentals;
    }
}
```

La clase Car representa un coche dentro del sistema de alquiler, gestionando su información básica y sus relaciones con otros elementos como el modelo, la oficina asignada y los alquileres. Tiene atributos clave: licensePlate (la matrícula), model (el modelo del coche), assignedOffice (la oficina donde está asignado) y rentals (una lista de los alquileres realizados con este coche). El constructor inicializa estos atributos y asegura que no sean nulos usando un assert. Además, al crear un coche, automáticamente lo añade a la lista de coches de su modelo y a la lista de coches de la oficina asignada, asegurando que las relaciones estén actualizadas.

En la clase hay getters y setters para acceder y modificar estos atributos. Muchos métodos son privados o protegidos para restringir su uso desde otras clases, como el acceso a la oficina asignada, que es protegido para que solo las clases relacionadas puedan usarlo.

Clase Customer

```
import java.util.*;
import java.time.*;

public class Customer implements ICustomer<WebRental> {
    private String dni;
    private String name;

    private List<RentalOnSite> rentalsOnSite;
    private List<WebRental> webRentals;

    public Customer (String dni, String name) {
        this.dni = dni;
        this.name = name;
        this.rentalsOnSite = new LinkedList<>();
        this.webRentals = new LinkedList<>();
            System.out.println("El cliente " + name + " se ha registrado
correctamente");
```

```
    }

//-------------------- GETTERS --------------------------

    private String getDni(){
        return this.dni;
    }

    protected String getName(){
        return this.name;
    }

    public List<RentalOnSite> getRentalsOnSite() {
        return this.rentalsOnSite;
    }

    protected List<WebRental> getWebRentals() {
        return this.webRentals;
    }

//-------------------- SETTERS --------------------------

    private void setName(String name) {
        assert(name != null);
        this.name = name;
    }

    private void setDni(String dni) {
        assert(dni != null);
        this.dni = dni;
    }

    private void setRentalsOnSite (List<RentalOnSite> rentalsOnSite) {
        assert(rentalsOnSite != null);
        this.rentalsOnSite = rentalsOnSite;
    }

    private void setWebRentals (List<WebRental> webRentals) {
        assert(webRentals != null);
        this.webRentals = webRentals;
    }

//-------------------- OTHER METHODS --------------------------
```

```
    /**
     * Permite alquilar un coche en una oficina de alquiler
     */
        public void rentCarOnSite(LocalDateTime startDate, LocalDateTime
endDate, Car car,RentalOffice pickUpOffice, String comment){
            assert(startDate != null && endDate != null && car != null &&
pickUpOffice != null && comment != null);
            RentalOnSite rental = new RentalOnSite(startDate, endDate, car,
this, pickUpOffice, comment);
        rentalsOnSite.add(rental);
        car.getRental().add(rental);
        pickUpOffice.getRental().add(rental);
          System.out.println(this.name + " ha añadido un nuevo alquiler en
oficina. Total alquileres de este cliente: " + (rentalsOnSite.size() +
webRentals.size()));
    }

    /**
     * Permite alquilar un coche a través de la web
     */
    public void rentCarOnWeb(LocalDateTime startDate, LocalDateTime endDate,
Car car,RentalOffice pickUpOffice, RentalOffice deliveryoffice){
            assert(startDate != null && endDate != null && car != null &&
pickUpOffice != null && deliveryoffice != null);
            WebRental rental = new WebRental(startDate, endDate, car, this,
pickUpOffice, deliveryoffice);
        webRentals.add(rental);
        car.getRental().add(rental);
        pickUpOffice.getRental().add(rental);
        deliveryoffice.getWebRentals().add(rental);
          System.out.println(this.name + " ha añadido un nuevo alquiler por
web. Total alquileres de este cliente: " + (rentalsOnSite.size() +
webRentals.size()));
    }

    /**
        * Devuelve el número de alquileres web de un cliente en los que
difieren la oficina de entrega y devolución del coche
     */
    public Integer numberOfRentalsWithDifferentOffices() {
        int count = 0;
        Iterator<WebRental> iterator = createWebRentalIterator();
```

```
        while (iterator.hasNext()) {
            WebRental rental = iterator.next();
                                                                    if
(!rental.getpickUpOffice().equals(rental.getDeliveryOffice())) {
                count++;
            }
        }
        return count;
    }

    @Override
    public Iterator<WebRental> createWebRentalIterator() {
        return new WebRentalIterator(webRentals);
    }
}
```

La clase Customer representa a un cliente en el sistema de alquiler de coches. Guarda datos básicos como el dni y el nombre (name), además de dos listas para los alquileres: los realizados en oficina (rentalsOnSite) y los hechos por la web (webRentals). Cuando se crea un cliente, inicializa sus datos y las listas vacías, mostrando un mensaje de confirmación.

Tiene métodos para alquilar coches, ya sea en oficina (rentCarOnSite) o a través de la web (rentCarOnWeb). Estos métodos crean un nuevo alquiler, lo añaden a las listas del cliente, del coche y de las oficinas correspondientes, y muestran un mensaje con el total de alquileres que tiene el cliente.

También incluye el método numberOfRentalsWithDifferentOffices(), que usa un iterador para recorrer los alquileres web y contar cuántos tienen oficinas de recogida y entrega diferentes. Este iterador se crea con el método createWebRentalIterator().

Interfaz ICustomer

```
public interface ICustomer<T> {
   Iterator<T> createWebRentalIterator();
}
```

Esta interfaz asegura que cualquier clase que la implemente tenga un método llamado createWebRentalIterator(). Este método permite crear un iterador que recorrerá una lista de elementos genéricos, como los alquileres web. Básicamente, obliga a las clases a organizar mejor cómo manejan sus datos.

Interfaz Iterator<T>

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

Representa un iterador que ayuda a recorrer listas o colecciones. Tiene dos métodos básicos:

- hasNext(): Verifica si hay más elementos por recorrer.
- next(): Devuelve el siguiente elemento de la colección.

Sirve para hacer recorridos ordenados sin preocuparte por cómo están guardados los datos.

Clase Model

```
import java.util.*;

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

    private List<Car> cars;

    public Model (String name, Integer pricePerDay) {
        assert(name != null && pricePerDay != null);
        this.name = name;
        this.pricePerDay = pricePerDay;
        this.cars = new LinkedList<>();
        System.out.println("El modelo de coche" + name + " se ha creado
correctamente");
    }

//-------------------- GETTERS -------------------------

    private String getName(){
        return this.name;
    }

    private Integer getpricePerDay(){
        return this.pricePerDay;
    }

    protected List<Car> getCars(){
```

```
        return this.cars;
    }

//-------------------- SETTERS --------------------------

     private void setName(String name) {
        assert(name != null);
        this.name = name;
    }

    private void setpricePerDay(Integer pricePerDay){
        assert(pricePerDay != null);
        this.pricePerDay = pricePerDay;
    }

    private void setCars(List<Car> cars){
        assert(cars != null);
        this.cars = cars;
    }
}
```

La clase Model representa un modelo de coche en el sistema y organiza toda la información relacionada con él. Incluye atributos como el nombre del modelo (name), el precio base por día (pricePerDay) y una lista de coches (cars) que pertenecen a este modelo. Su constructor asegura que estos datos se inicialicen correctamente, usando assert para validar que no sean nulos, y automáticamente crea una lista vacía para los coches. Tiene métodos getters para acceder a sus datos y setters privados para proteger la modificación de los mismos, asegurando que cualquier cambio sea válido.

Clase Rental

```
import java.time.*;
import java.util.ArrayList;
import java.util.List;

public class Rental {
    private LocalDateTime startDate;
    private LocalDateTime endDate;

    private Car car;
    private Customer customer;
    private RentalOffice pickUpOffice;
```

```
    public Rental(LocalDateTime sDate, LocalDateTime eDate, Car c, Customer
customer, RentalOffice pickUpOffice) {
        assert(sDate != null && eDate != null && car != null && customer !=
null && pickUpOffice != null);
        assert(noAlquileresSolapados(customer, sDate) &&
sDate.isBefore(eDate) && noOficinasDistintas(car, pickUpOffice)); // Se
comprueban las restricciones de integridad 1, 2 y 3
        this.car = c;
        this.startDate = sDate;
        this.endDate = eDate;
        this.customer = customer;
        this.pickUpOffice = pickUpOffice;
    }

//-------------------- GETTERS --------------------------

    private LocalDateTime getStartDate() {
        return this.startDate;
    }

    private LocalDateTime getEndDate() {
        return this.endDate;
    }

    private Car getCar() {
        return this.car;
    }

    private Customer getCustomer() {
        return this.customer;
    }

    protected RentalOffice getpickUpOffice() {
        return this.pickUpOffice;
    }

//-------------------- SETTERS --------------------------

    private void setStartDate(LocalDateTime startDate) {
        assert(startDate != null);
        this.startDate = startDate;
    }
```

```
    private void setEndDate(LocalDateTime endDate) {
        assert(endDate != null);
        this.endDate = endDate;
    }

    private void setCar(Car car) {
        assert(car != null);
        this.car = car;
    }

    private void setCustomer(Customer customer) {
        assert(customer != null);
        this.customer = customer;
    }

    private void setpickUpOffice(RentalOffice pickUpOffice) {
        assert(pickUpOffice != null);
        this.pickUpOffice = pickUpOffice;
    }

//-------------------- OTHER METHODS --------------------------

    /**
     * Comprueba que un cliente no tenga alquileres solapados
     */
    private boolean noAlquileresSolapados(Customer customer, LocalDateTime
startDate) {
        boolean sol = true;
        List<Rental> allRentals = new
ArrayList<>(customer.getRentalsOnSite());
        allRentals.addAll(customer.getWebRentals());
        for(Rental rental: allRentals) {
            if(rental.getEndDate().isAfter(startDate)){
                sol = false;
            }
        }
        return sol;
    }

    /**
     * Comprueba que la oficina asignada al coche es la misma que la de
pickup
     */
```

```
    private boolean noOficinasDistintas(Car car, RentalOffice pickUpOffice)
{
        assert(car != null && pickUpOffice != null);
        return car.getAssignedOffice().equals(pickUpOffice);
    }
}
```

La clase Rental es la base para representar un alquiler en el sistema. Tiene datos como las fechas de inicio y fin (startDate y endDate), el coche alquilado (car), el cliente (customer) y la oficina de recogida (pickUpOffice). Cuando se crea un alquiler, el constructor se asegura de que todo sea válido, como que no haya solapamientos con otros alquileres del cliente o que la oficina de recogida coincida con la oficina asignada al coche. Además, incluye métodos para acceder y modificar estos datos, aunque los setters son privados. También tiene funciones auxiliares que verifican las reglas del sistema:

- noAlquileresSolapados: Asegura que un cliente no tenga alquileres con fechas que se solapen.
- noOficinasDistintas: Verifica que la oficina de recogida coincida con la oficina asignada al coche.

Clase RentalOffice

```
import java.util.*;

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

    private List<Car> cars;
    private List<Rental> rentals;
    private List<WebRental> webRentals;

    public RentalOffice(String address, Integer feeForDelivery) {
        assert(address != null && feeForDelivery != null);
        this.address = address;
        this.feeForDelivery = feeForDelivery;
        this.rentals = new LinkedList<Rental>();
        this.cars = new LinkedList<Car>();
        this.webRentals = new LinkedList<WebRental>();
        System.out.println("La oficina situada en " + address + " se ha
creado correctamente");
    }

//------------------- GETTERS -------------------------
```

```
    private String getAddress() {
        return address;
    }

    private Integer getFeeForDelivery(){
        return feeForDelivery;
    }

    protected List<Rental> getRental(){
        return rentals;
    }

    protected List<Car> getCars(){
        return cars;
    }

    protected List<WebRental> getWebRentals(){
        return webRentals;
    }

//-------------------- SETTERS --------------------------

    private void setAddress(String address) {
        assert(address != null);
        this.address = address;
    }

    private void setfeeForDelivery(Integer feeForDelivery) {
        assert(feeForDelivery != null);
        this.feeForDelivery = feeForDelivery;
    }

    private void setRentals(List<Rental> rentals){
        assert(rentals != null);
        this.rentals = rentals;
    }

    private void setCars(List<Car> cars){
        assert(cars != null);
        this.cars = cars;
    }
```

```
    private void setWebRentals(List<WebRental> webRentals){
        assert(webRentals != null);
        this.webRentals = webRentals;
    }
}
```

La clase RentalOffice representa una oficina de alquiler de coches y organiza todos los datos relacionados con ella. Incluye atributos como la dirección (address), la tarifa por entregas en oficinas diferentes (feeForDelivery), y tres listas para gestionar coches disponibles (cars), alquileres realizados en oficina (rentals), y alquileres realizados por la web (webRentals). El constructor valida que todos los datos sean válidos y garantiza que las listas se inicialicen correctamente, además de imprimir un mensaje confirmando la creación de la oficina.

Clase RentalOnSite

```
import java.time.*;

public class RentalOnSite extends Rental {
    private String comments;

    public RentalOnSite(LocalDateTime startDate, LocalDateTime endDate, Car
car, Customer customer, RentalOffice pickUpOffice, String comments){
        super(startDate, endDate, car, customer, pickUpOffice);
        assert(comments != null);
        this.comments = comments;
    }

//-------------------- GETTERS --------------------------

    private String getComments() {
        return this.comments;
    }

//-------------------- SETTERS --------------------------

    private void setComments(String comments) {
        assert(comments != null);
        this.comments = comments;
    }
}
```

RentalOnSite es una versión específica de Rental para los alquileres que se hacen directamente en la oficina. Además de los datos básicos de un alquiler, incluye un campo para comentarios (comments)

donde se pueden registrar notas adicionales sobre el alquiler. Su constructor valida que los comentarios no sean nulos, y los métodos permiten acceder o cambiarlos si es necesario.

Clase WebRental

```
import java.time.*;

public class WebRental extends Rental{
    private Integer deliveryTime;

    private RentalOffice deliveryOffice;

    public WebRental(LocalDateTime startDate, LocalDateTime endDate, Car
car, Customer customer, RentalOffice pickUpOffice, RentalOffice
deliveryOffice) {
        super(startDate, endDate, car, customer, pickUpOffice);
        assert(deliveryOffice != null);
        assert(comprobarHoraOficinasDiferentes(pickUpOffice, deliveryOffice,
endDate));
        this.deliveryTime = 0;
        this.deliveryOffice = deliveryOffice;
    }

 //-------------------- GETTERS -------------------------

    private Integer getDeliveryTime(){
        return deliveryTime;
    }

    protected RentalOffice getDeliveryOffice(){
        return deliveryOffice;
    }

//-------------------- SETTERS -------------------------

    private void setDeliveryTime(Integer deliveryTime){
        assert(deliveryTime != null);
        this.deliveryTime = deliveryTime;
    }

    private void setRentalOffice(RentalOffice deliveryOffice){
        assert(deliveryOffice != null);
```

```
        this.deliveryOffice = deliveryOffice;
    }

//-------------------- OTHER METHODS -------------------------

    /**
     * Devuelve true si, cuando las oficinas de recogida/entrega son
diferentes, la hora es anterior a las 13h
     */
    private boolean comprobarHoraOficinasDiferentes(RentalOffice
pickupOffice, RentalOffice deliveryOffice, LocalDateTime endDate){
        boolean sol = true;
        if(pickupOffice != deliveryOffice){
            if(endDate.getHour() > 13) {
                sol = false;
            }
        }
        return sol;
    }
}
```

La clase WebRental extiende a Rental para gestionar los alquileres realizados a través de la web. Tiene algunos atributos extra, como la hora de entrega (deliveryTime) y la oficina donde se devuelve el coche (deliveryOffice). El constructor se asegura de que, si las oficinas de recogida y entrega son diferentes, la hora de devolución sea antes de las 13:00, validándolo con un método específico. También incluye funciones para acceder y modificar estos datos.

Clase WebRentalIterator

```
import java.util.List;
import java.util.NoSuchElementException;

public class WebRentalIterator implements Iterator<WebRental> {

    private List<WebRental> webRentals;
    private int position;

    public WebRentalIterator(List<WebRental> webRentals) {
        assert(webRentals != null);
        this.webRentals = webRentals;
        this.position = 0;
    }
```

```
    @Override
    public boolean hasNext() {
        return position < webRentals.size();
    }

    @Override
    public WebRental next() {
        if (!hasNext()) throw new NoSuchElementException();
        return webRentals.get(position++);
    }
}
```

La clase WebRentalIterator implementa un iterador que facilita recorrer listas de alquileres web (WebRental). Tiene una lista de alquileres (webRentals) y un índice (position) que indica dónde estás en el recorrido. Con el método hasNext() puedes verificar si quedan elementos por recorrer, y con next() obtienes el siguiente alquiler y avanzas en la lista. El constructor valida que la lista no sea nula y empieza el recorrido desde el principio.

Clase Main para probar el código

```
import java.time.*;

public class Main {
    public static void main(String[] args) {
        Customer Customer1 = new Customer("77515974W", "Pepe");

        LocalDateTime fecha1 = LocalDateTime.of(LocalDate.of(2020,12,20),
LocalTime.of(12,53));
        LocalDateTime fecha2 = LocalDateTime.of(LocalDate.of(2021,2,12),
LocalTime.of(8,35));
        LocalDateTime fecha3 = LocalDateTime.of(LocalDate.of(2021,4,21),
LocalTime.of(10,11));
        LocalDateTime fecha4 = LocalDateTime.of(LocalDate.of(2021,7,18),
LocalTime.of(22,37));
        LocalDateTime fecha5 = LocalDateTime.of(LocalDate.of(2021,9,27),
LocalTime.of(18,47));
        LocalDateTime fecha6 = LocalDateTime.of(LocalDate.of(2021,12,8),
LocalTime.of(10,28));


        Model ferrari = new Model("Ferrari", 50);

        RentalOffice oficina1 = new RentalOffice("Av. Plutarco, 75", 20);
```

```
        RentalOffice oficina2 = new RentalOffice("Calle Larios, 3", 25);

        Car coche1 = new Car("2587MDN", ferrari, oficina1);
        Car coche2 = new Car("7856BPM", ferrari, oficina1);

        Customer1.rentCarOnWeb(fecha1, fecha2, coche1, oficina1, oficina2);
        Customer1.rentCarOnWeb(fecha3, fecha4, coche2, oficina1, oficina2);
        Customer1.rentCarOnWeb(fecha5, fecha6, coche2, oficina1, oficina1);

        System.out.println("Número de alquileres con diferentes oficinas de
" + Customer1.getName() + ": " +
Customer1.numberOfRentalsWithDifferentOffices());
    }
}
```

La clase Main sirve para probar el funcionamiento del sistema de alquileres de coches. En ella se crean instancias de las clases principales, como un cliente (Customer1), fechas específicas para los alquileres, un modelo de coche (Ferrari), dos oficinas de alquiler (oficina1 y oficina2), y dos coches asociados al modelo (coche1 y coche2). Luego, se realizan tres alquileres por web para el cliente: los dos primeros tienen oficinas de recogida y entrega diferentes (oficina1 y oficina2), mientras que el tercero utiliza la misma oficina para ambas operaciones (oficina1). Finalmente, el sistema imprime el número total de alquileres donde las oficinas de recogida y entrega son diferentes, utilizando el método numberOfRentalsWithDifferentOffices().

Salida por terminal

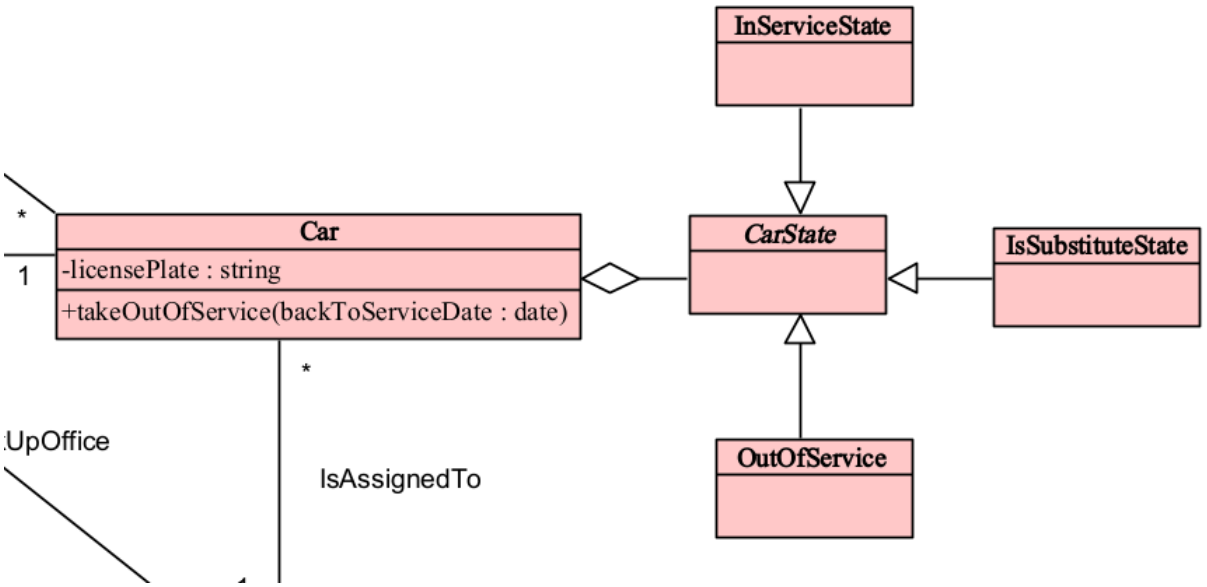
```
El cliente Pepe se ha registrado correctamente
El modelo de cocheFerrari se ha creado correctamente
La oficina situada en Av. Plutarco, 75 se ha creado correctamente
La oficina situada en Calle Larios, 3 se ha creado correctamente
El coche con matrícula 2587MDN se ha creado correctamente
El coche con matrícula 7856BPM se ha creado correctamente
Pepe ha añadido un nuevo alquiler por web. Total alquileres de este cliente: 1
Pepe ha añadido un nuevo alquiler por web. Total alquileres de este cliente: 2
Pepe ha añadido un nuevo alquiler por web. Total alquileres de este cliente: 3
Número de alquileres con diferentes oficinas de Pepe: 2
```

EJERCICIO 2

Apartado A

El patrón Estado (State) es el más adecuado para este ejercicio porque permite gestionar de manera organizada los cambios en el estado de un coche (en servicio o fuera de servicio). Evitando el uso de muchas condiciones para poder verificar el estado en el que está, por lo que hace que el código sea más claro. Además, este patrón cumple con el principio de abierto/cerrado, permitiendo añadir nuevos estados (como "en mantenimiento") sin modificar el diseño existente. También promueve la separación de responsabilidades, ya que las acciones específicas de cada estado se delegan a las clases correspondientes, manteniendo el código limpio y fácil de extender.

Apartado B



Este diagrama del ejercicio 2, apartado B, muestra cómo se usa el patrón Estado (State) para gestionar los diferentes estados de un coche en el sistema. La clase Car tiene un atributo state que representa su estado actual, y este estado puede ser InServiceState (en servicio), OutOfService (fuera de servicio) o IsSubstituteState (coche sustituto). Este diseño permite que Car delegue las acciones específicas a cada estado, haciendo que el código sea fácil de extender si se añaden nuevos estados en el futuro.

Apartado C

Código de la operación takeOutOfService() de la clase Car:

```
/**
        Esta funcionalidad pondrá el coche fuera de
        servicio y registrará la fecha hasta la cual estará fuera de
servicio y, si hay, buscará y registrará también un coche
        sustituto
     */
    public void takeOutOfService(LocalDateTime backToServiceDate) {
        assert(backToServiceDate != null);
        if (this.getState() instanceof OutOfServiceState || this.getState()
instanceof IsSubstituteState) {
            System.out.println("El coche ya está fuera de servicio o es un
coche sustituto");
        } else {
            this.setState(new OutOfServiceState(this, backToServiceDate));
            List<Car> cars = this.assignedOffice.getCars();
            Car substitute = null;
            for(Car car : cars){
                if(car.getModel().equals(this.getModel()) && car.getState()
instanceof InServiceState && car != this){
                    substitute = car;
                    substitute.setState(new IsSubstituteState(substitute));
                    break;
                }
            }

            System.out.println("El coche se ha puesto fuera de servicio
correctamente, fecha de vuelta al servicio: " + backToServiceDate);

            if(substitute != null) {
                System.out.println("El coche sustituto es: " +
substitute.getLicensePlate());
            } else {
                System.out.println("No hay coches sustitutos disponibles");
            }
        }
    }
```

El método takeOutOfService() de la clase Car pone un coche fuera de servicio y, si es posible, asigna un coche sustituto. Primero, comprueba que la fecha de retorno (backToServiceDate) no sea nula.

Después, verifica si el coche ya está fuera de servicio o es un coche sustituto. Si es así, muestra un mensaje diciendo que no se puede realizar la acción. Si el coche está en servicio, cambia su estado a OutOfServiceState y busca un sustituto dentro de la oficina. Para ser sustituto, el coche debe ser del mismo modelo, estar en servicio (InServiceState) y no ser el mismo coche que se está poniendo fuera de servicio. Si encuentra un sustituto, le cambia el estado a IsSubstituteState y lo asigna; si no, informa que no hay coches disponibles. Por último, confirma que el coche está fuera de servicio y, si hay un sustituto, muestra su matrícula.

RESTO DEL CÓDIGO:
Las siguientes clases sus códigos no han sido modificados en comparación con el que había anteriormente: Car, Customer, Model, WebRental, RentalOnSite, RentalOffice y Rental.

Clase CarState

```
package src;
import java.time.*;
public abstract class CarState {
    Car car;
    public CarState(Car car) {
        assert(car != null);
        this.car = null;
    }
}
```

La clase CarState es una clase abstracta que sirve como base para representar los diferentes estados de un coche en el sistema, como InServiceState, OutOfServiceState o IsSubstituteState. Su único atributo es el coche (car) asociado al estado, que se valida en el constructor para garantizar que no sea nulo. Esta clase no tiene métodos definidos aquí, está pensada para que las clases específicas de estado la extiendan y añadan el comportamiento específico de cada estado.

Clase InServiceState

```
package src;
import java.time.*;

class InServiceState extends CarState {
    public InServiceState(Car car) {
        super(car);
    }

    public String toString() {
        return "In Service";
    }
```

```
}
```

La clase InServiceState representa el estado "en servicio" de un coche en el sistema, es decir, disponible para alquilar. Hereda de CarState y usa su constructor para asociarlo a un coche. Su único método sobrescrito es toString(), que devuelve "In Service" como descripción del estado.

Clase IsSubstituteState

```
package src;
public class IsSubstituteState extends CarState {
    public IsSubstituteState(Car car) {
        super(car);
    }

    public String toString() {
        return "Substitute";
    }
}
```

La clase IsSubstituteState define el estado "sustituto" para un coche, indicando que el coche está sirviendo como reemplazo de otro que está fuera de servicio. También hereda de CarState y usa su constructor para asociarse al coche correspondiente. Sobrescribe el método toString() para devolver "Substitute" como descripción del estado.

Clase OutOfServiceState

```
package src;
import java.time.*;


class OutOfServiceState extends CarState{
    private LocalDateTime backToServiceDate;

    public OutOfServiceState(Car car, LocalDateTime backToServiceDate) {
        super(car);
        assert(backToServiceDate != null);
        this.backToServiceDate = backToServiceDate;
    }

    public String toString() {
        return "Out of service until " + this.backToServiceDate;
    }
```

```
}
```

La clase OutOfServiceState representa el estado "fuera de servicio" de un coche. Hereda de CarState y utiliza su constructor para asociar el estado con un coche específico. Además, añade un atributo backToServiceDate, que almacena la fecha en la que el coche volverá a estar disponible. En el constructor, valida que esta fecha no sea nula antes de asignarla. También sobrescribe el método toString() para devolver una descripción del estado junto con la fecha de regreso al servicio.

Clase Main_ej2

```
package src;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class Main_ej2 {
    public static void main(String[] args) {
        LocalDateTime fecha1 = LocalDateTime.of(LocalDate.of(2021,12,8),
LocalTime.of(10,28));



        Model ferrari = new Model("Ferrari", 50);

        RentalOffice oficina1 = new RentalOffice("Av. Plutarco, 75", 20);

        Car coche1 = new Car("2587MDN", ferrari, oficina1);
        Car coche2 = new Car("7856BPM", ferrari, oficina1);
        Car coche3 = new Car("1234ABC", ferrari, oficina1);

        System.out.println("\nEl estado del coche 1 antes de la operación
es: " + coche1.getState().toString());
        System.out.println("El estado del coche 2 antes de la operación es:
" + coche2.getState().toString());
        System.out.println("El estado del coche 3 antes de la operación es:
" + coche3.getState().toString() + "\n");

        System.out.println("CASO 1: El coche 1 se pone fuera de servicio y
hay un coche sustituto disponible");
        coche1.takeOutOfService(fecha1);
        System.out.println("El estado del coche 1 tras la operación es: " +
coche1.getState().toString());
```

```
        System.out.println("El estado del coche 2 tras la operación es: " +
coche2.getState().toString());
        System.out.println("El estado del coche 3 tras la operación es: " +
coche3.getState().toString() + "\n");

        System.out.println("CASO 2: Se intenta poner fuera de servicio un
coche que es sustituto");
        coche2.takeOutOfService(fecha1);
        System.out.println("El estado del coche 1 tras la operación es: " +
coche1.getState().toString());
        System.out.println("El estado del coche 2 tras la operación es: " +
coche2.getState().toString());
        System.out.println("El estado del coche 3 tras la operación es: " +
coche3.getState().toString() + "\n");

        System.out.println("CASO 3: Se pone fuera de servicio un coche que
ya está fuera de servicio");
        coche1.takeOutOfService(fecha1);
        System.out.println("El estado del coche 1 tras la operación es: " +
coche1.getState().toString());
        System.out.println("El estado del coche 2 tras la operación es: " +
coche2.getState().toString());
        System.out.println("El estado del coche 3 tras la operación es: " +
coche3.getState().toString() + "\n");

        System.out.println("CASO 4: Se pone fuera de servicio un coche pero
no hay coche sustituto disponible");
        coche3.takeOutOfService(fecha1);
        System.out.println("El estado del coche 1 tras la operación es: " +
coche1.getState().toString());
        System.out.println("El estado del coche 2 tras la operación es: " +
coche2.getState().toString());
        System.out.println("El estado del coche 3 tras la operación es: " +
coche3.getState().toString() + "\n");

    }
}
```

La clase Main_ej2 prueba el funcionamiento del sistema para gestionar los estados de los coches usando el patrón Estado (State). Se crean un modelo de coche (Ferrari), una oficina de alquiler (oficina1), y tres coches (coche1, coche2 y coche3) asociados a esa oficina. Luego se simulan cuatro situaciones para probar la operación takeOutOfService() y ver cómo cambian los estados de los coches.

1. Caso 1: Se pone fuera de servicio coche1. Como hay un sustituto disponible (coche2), este cambia su estado a "sustituto".
2. Caso 2: Se intenta poner fuera de servicio un coche que ya es sustituto (coche2). El sistema no lo permite y muestra un mensaje explicativo.
3. Caso 3: Se intenta poner fuera de servicio un coche que ya está fuera de servicio (coche1). Nuevamente, el sistema informa que no es posible.
4. Caso 4: Se pone fuera de servicio coche3, pero no hay coches sustitutos disponibles, por lo que solo coche3 cambia su estado a "fuera de servicio".

Salida por terminal

```
El modelo de coche Ferrari se ha creado correctamente
La oficina situada en Av. Plutarco, 75 se ha creado correctamente
El coche con matrícula 2587MDN se ha creado correctamente
El coche con matrícula 7856BPM se ha creado correctamente
El coche con matrícula 1234ABC se ha creado correctamente

El estado del coche 1 antes de la operación es: In Service
El estado del coche 2 antes de la operación es: In Service
El estado del coche 3 antes de la operación es: In Service

CASO 1: El coche 1 se pone fuera de servicio y hay un coche sustituto disponible
El coche se ha puesto fuera de servicio correctamente, fecha de vuelta al servicio: 2021-12-08T10:28
El coche sustituto es: 7856BPM
El estado del coche 1 tras la operación es: Out of service until 2021-12-08T10:28
El estado del coche 2 tras la operación es: Substitute
El estado del coche 3 tras la operación es: In Service

CASO 2: Se intenta poner fuera de servicio un coche que es sustituto
El coche ya está fuera de servicio o es un coche sustituto
El estado del coche 1 tras la operación es: Out of service until 2021-12-08T10:28
El estado del coche 2 tras la operación es: Substitute
El estado del coche 3 tras la operación es: In Service

CASO 3: Se pone fuera de servicio un coche que ya está fuera de servicio
El coche ya está fuera de servicio o es un coche sustituto
El estado del coche 1 tras la operación es: Out of service until 2021-12-08T10:28
El estado del coche 2 tras la operación es: Substitute
El estado del coche 3 tras la operación es: In Service
```

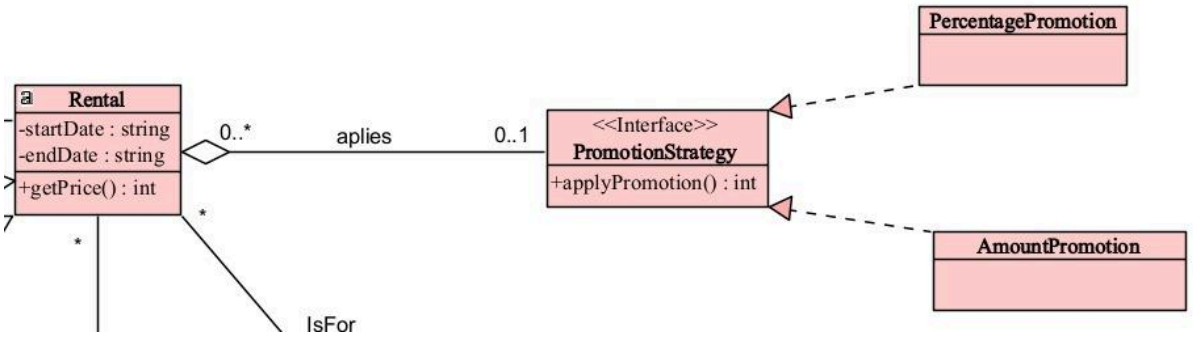
```
CASO 4: Se pone fuera de servicio un coche pero no hay coche sustituto disponible
El coche se ha puesto fuera de servicio correctamente, fecha de vuelta al servicio: 2021-12-08T10:28
No hay coches sustitutos disponibles
El estado del coche 1 tras la operación es: Out of service until 2021-12-08T10:28
El estado del coche 2 tras la operación es: Substitute
El estado del coche 3 tras la operación es: Out of service until 2021-12-08T10:28
```

EJERCICIO 3

Apartado A

La utilidad del Patrón Estrategia (Strategy) es que permite separar y organizar las diferentes formas de calcular el precio de un alquiler dependiendo de la promoción aplicada. En lugar de tener condicionales complejas dentro de la clase Rental, para calcular el precio según la promoción, cada tipo de promoción (como descuento fijo o porcentual) se maneja en su propia clase. Si se necesita añadir o cambiar una promoción, solo hay que crear una nueva estrategia sin afectar la lógica de Rental. Además, este patrón permite que cada alquiler pueda tener una promoción personalizada o no tener ninguna, adaptándose dinámicamente a las necesidades del sistema.

Apartado B



El diagrama muestra cómo se aplica el Patrón Estrategia (Strategy) para gestionar las promociones en los alquileres. La clase Rental tiene un método getPrice() que calcula el precio usando una estrategia de promoción (PromotionStrategy). Esta estrategia está definida en una interfaz con el método applyPromotion(), que es implementado por clases concretas como PercentagePromotion (descuento en porcentaje) y AmountPromotion (descuento fijo). Cada alquiler puede tener una promoción específica o no tener ninguna (0..1). Esto hace que el sistema sea muy flexible y fácil de modificar, ya que puedes cambiar o añadir promociones sin tocar la lógica de Rental.

Apartado C

Las siguientes clases sus códigos no han sido modificados en comparación con los ejercicios anteriores: Car, Model y RentalOffice.

Código de la operación getPrice() de la clase Rental:

```
    /**
     * Proporciona el precio del alquiler del coche
     */
    public double getPrice(){
        int days = (int) Duration.between(startDate, endDate).toDays();
        double basePrice = car.getModel().getpricePerDay() * days;
        if (promotion != null) {
            return promotion.applyPromotion(basePrice);
        }
        return basePrice;
    }
```

El método getPrice() de la clase Rental calcula el precio total de un alquiler teniendo en cuenta los días de duración y, opcionalmente, una promoción aplicada al precio base. Primero calcula cuántos días hay entre las fechas de inicio (startDate) y fin (endDate) del alquiler. Luego, multiplica esa cantidad de días por el precio diario del modelo del coche (car.getModel().getpricePerDay()) para sacar el precio base. Si hay una promoción asociada (almacenada en promotion), aplica el descuento llamando a promotion.applyPromotion(basePrice). Si no hay ninguna promoción, simplemente devuelve el precio base.

Clase AmountPromotion

```
package Ejercicio3.ApartadoC;
public class AmountPromotion implements IPromotionStrategy {
    private int amount;

    public AmountPromotion(int amount) {
        assert(amount > 0);
        this.amount = amount;
    }

    @Override
    public double applyPromotion(double basePrice) {
        return basePrice - amount;
    }
```

```
}
```

La clase AmountPromotion aplica un descuento fijo al precio del alquiler y forma parte del patrón Estrategia (Strategy). Implementa la interfaz IPromotionStrategy, que define el método applyPromotion(). Tiene un atributo amount, que representa cuánto se descuenta, y el constructor valida que este valor sea positivo. Cuando se llama a applyPromotion(double basePrice), la clase simplemente resta el valor del descuento (amount) al precio base y devuelve el resultado.

Clase IPromotionStrategy

```
package Ejercicio3.ApartadoC;

public interface IPromotionStrategy {
    double applyPromotion(double basePrice);
}
```

La interfaz IPromotionStrategy define la estructura que deben seguir todas las promociones dentro del sistema. Contiene un único método, applyPromotion(double basePrice), que toma el precio base de un alquiler y devuelve el precio modificado según la promoción aplicada. Esta interfaz permite que diferentes tipos de promociones (como descuentos fijos o porcentuales) se implementen de manera consistente y sean intercambiables.

Clase PercentagePromotion

```
package Ejercicio3.ApartadoC;
public class PercentagePromotion implements IPromotionStrategy {
    private double percentage;

    public PercentagePromotion(int percentage) {
        assert(percentage > 0);
        this.percentage = percentage;
    }

    @Override
    public double applyPromotion(double basePrice) {
        return basePrice - (basePrice * (percentage / 100));
    }

}
```

La clase PercentagePromotion aplica un descuento basado en un porcentaje al precio de un alquiler. Tiene un atributo percentage, que guarda el porcentaje del descuento, y en el constructor se asegura

que este sea un valor positivo. El método applyPromotion(double basePrice) calcula el precio final restando el porcentaje indicado al precio base (basePrice * (percentage / 100)).

Clase Rental

```
package Ejercicio3.ApartadoC;

import java.time.*;
import java.util.ArrayList;
import java.util.List;
import java.time.Duration;

public class Rental {
    private LocalDateTime startDate;
    private LocalDateTime endDate;

    private Car car;
    private Customer customer;
    private RentalOffice pickUpOffice;
    private IPromotionStrategy promotion;

    public Rental(LocalDateTime sDate, LocalDateTime eDate, Car c, Customer
customer, RentalOffice pickUpOffice, IPromotionStrategy promotion) {
        // no hace falta comprobar que promotion no sea null ya que podría
no aplicarse ninguna promoción
        assert(sDate != null && eDate != null && car != null && customer !=
null && pickUpOffice != null);
        assert(noAlquileresSolapados(customer, sDate) &&
sDate.isBefore(eDate) && noOficinasDistintas(car, pickUpOffice)); // Se
comprueban las restricciones de integridad 1, 2 y 3
        this.car = c;
        this.startDate = sDate;
        this.endDate = eDate;
        this.customer = customer;
        this.pickUpOffice = pickUpOffice;
        this.promotion = promotion;
    }

//-------------------- GETTERS -------------------------

    private LocalDateTime getStartDate() {
        return this.startDate;
    }
```

```
    private LocalDateTime getEndDate() {
        return this.endDate;
    }

    private Car getCar() {
        return this.car;
    }

    private Customer getCustomer() {
        return this.customer;
    }

    protected RentalOffice getpickUpOffice() {
        return this.pickUpOffice;
    }

    private IPromotionStrategy getPromotion() {
        return this.promotion;
    }

//-------------------- SETTERS --------------------------

    private void setStartDate(LocalDateTime startDate) {
        assert(startDate != null);
        this.startDate = startDate;
    }

    private void setEndDate(LocalDateTime endDate) {
        assert(endDate != null);
        this.endDate = endDate;
    }

    private void setCar(Car car) {
        assert(car != null);
        this.car = car;
    }

    private void setCustomer(Customer customer) {
        assert(customer != null);
        this.customer = customer;
    }
```

```
    private void setpickUpOffice(RentalOffice pickUpOffice) {
        assert(pickUpOffice != null);
        this.pickUpOffice = pickUpOffice;
    }

    protected void setPromotion(IPromotionStrategy promotion) {
        this.promotion = promotion;
    }

//-------------------- OTHER METHODS --------------------------

    /**
     * Comprueba que un cliente no tenga alquileres solapados
     */
    private boolean noAlquileresSolapados(Customer customer, LocalDateTime
startDate) { // Comprueba que un cliente no tenga alquileres solapados
        boolean sol = true;
        List<Rental> allRentals = new
ArrayList<>(customer.getRentalsOnSite());
        allRentals.addAll(customer.getWebRentals());
        for(Rental rental: allRentals) {
            if(rental.getEndDate().isAfter(startDate)){
                sol = false;
            }
        }
        return sol;
    }

    /**
     * Comprueba que la oficina asignada al coche es la misma que la de
pickup
     */
    private boolean noOficinasDistintas(Car car, RentalOffice pickUpOffice)
{ // Comprueba que la oficina asignada al coche es la misma que la de pickup
        assert(car != null && pickUpOffice != null);
        return car.getAssignedOffice().equals(pickUpOffice);
    }

    /**
     * Proporciona el precio del alquiler del coche
     */
    public double getPrice(){
        int days = (int) Duration.between(startDate, endDate).toDays();
```

```
        double basePrice = car.getModel().getpricePerDay() * days;
        if (promotion != null) {
            return promotion.applyPromotion(basePrice);
        }
        return basePrice;
    }
}
```

La clase Rental se encarga de representar un alquiler en el sistema y manejar toda la información relacionada con él, como las fechas de inicio y fin (startDate y endDate), el coche que se alquila (car), el cliente (customer), la oficina de recogida (pickUpOffice) y, si aplica, una promoción (promotion) para calcular el precio con descuento.

El constructor valida varias cosas antes de crear el alquiler: que las fechas y los datos no sean nulos, que el cliente no tenga otros alquileres solapados (con el método noAlquileresSolapados()), que la fecha de inicio sea anterior a la de fin, y que la oficina de recogida coincida con la asignada al coche (usando noOficinasDistintas()).

El método más importante es getPrice(), que calcula el precio total del alquiler. Primero multiplica los días del alquiler (calculados con Duration.between) por el precio diario del modelo del coche. Luego, si hay una promoción, aplica el descuento llamando al método applyPromotion() de la estrategia asociada. Si no hay promoción, simplemente devuelve el precio base.

Clase RentalOnSite

```
package Ejercicio3.ApartadoC;
import java.time.*;

public class RentalOnSite extends Rental {
    private String comments;

    public RentalOnSite(LocalDateTime startDate, LocalDateTime endDate, Car
car, Customer customer, RentalOffice pickUpOffice,IPromotionStrategy
promotion, String comments){
        super(startDate, endDate, car, customer, pickUpOffice, promotion);
        assert(comments != null);
        this.comments = comments;
    }

//-------------------- GETTERS -------------------------
```

```
    private String getComments() {
        return this.comments;
    }

//------------------- SETTERS -------------------------

    private void setComments(String comments) {
        assert(comments != null);
        this.comments = comments;
    }

}
```

La clase RentalOnSite es una extensión de Rental pensada para los alquileres que se hacen directamente en una oficina. Además de los atributos heredados, incluye un campo extra llamado comments, donde se pueden agregar notas específicas sobre el alquiler. El constructor se asegura de que estos comentarios no sean nulos antes de asignarlos.

Clase WebRental

```
package Ejercicio3.ApartadoC;
import java.time.*;

public class WebRental extends Rental{
    private Integer deliveryTime;   // 0..1 (cuando sea 0 Integer será null)

    private RentalOffice deliveryOffice;

    public WebRental(LocalDateTime startDate, LocalDateTime endDate, Car
car, Customer customer, RentalOffice pickUpOffice, IPromotionStrategy
promotion, RentalOffice deliveryOffice) {
        super(startDate, endDate, car, customer, pickUpOffice, promotion);
        assert(deliveryOffice != null &&
comprobarHoraOficinasDiferentes(pickUpOffice, deliveryOffice, endDate));
        this.deliveryTime = 0;
        this.deliveryOffice = deliveryOffice;
    }

//------------------- GETTERS -------------------------

    private Integer getDeliveryTime(){
```

```
        return deliveryTime;
    }

    protected RentalOffice getDeliveryOffice(){
        return deliveryOffice;
    }

//-------------------- SETTERS -------------------------

    private void setDeliveryTime(Integer deliveryTime){
        assert(deliveryTime != null);
        this.deliveryTime = deliveryTime;
    }

    private void setRentalOffice(RentalOffice deliveryOffice){
        assert(deliveryOffice != null);
        this.deliveryOffice = deliveryOffice;
    }

//-------------------- OTHER METHODS -------------------------

    /**
     * Devuelve true si, cuando las oficinas de recogida/entrega son
diferentes, la hora es anterior a las 13h
     */
    private boolean comprobarHoraOficinasDiferentes(RentalOffice
pickupOffice, RentalOffice deliveryOffice, LocalDateTime endDate){
        boolean sol = true;
        if(pickupOffice != deliveryOffice){
            if(endDate.getHour() > 13) {
                sol = false;
            }
        }
        return sol;
    }
}
```

La clase WebRental extiende Rental para gestionar los alquileres hechos por la web. Agrega dos atributos extra: deliveryTime, que guarda la hora de devolución del coche (puede ser null si no está definida), y deliveryOffice, que indica la oficina donde se devolverá el coche. El constructor valida que la oficina de devolución no sea nula y que, si es diferente de la de recogida, la devolución sea antes de las 13:00. Esto se verifica con el método comprobarHoraOficinasDiferentes().

El método adicional asegura que se cumplan las reglas para devoluciones entre oficinas.

Clase Main

```
package Ejercicio3.ApartadoC;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Duration;

public class Main {
    public static void main(String[] args) {
        PercentagePromotion percentagePromotion = new
PercentagePromotion(10);
        AmountPromotion amountPromotion = new AmountPromotion(5);

        Customer Customer1 = new Customer("77515974M", "Francisco");
        Customer Customer2 = new Customer("77515974F", "Marcos");
        Customer Customer3 = new Customer("77515974M", "Soraya");
        Customer Customer4 = new Customer("77515974V", "Victoria");

        LocalDateTime fecha1 = LocalDateTime.of(LocalDate.of(2020,12,20),
LocalTime.of(8,00));
        LocalDateTime fecha2 = LocalDateTime.of(LocalDate.of(2020,12,22),
LocalTime.of(8,00));



        Model ferrari = new Model("Ferrari", 100.00);

        RentalOffice oficina1 = new RentalOffice("Av. Plutarco, 75", 20);
        RentalOffice oficina2 = new RentalOffice("Calle Larios, 3", 25);

        Car coche1 = new Car("2587MDN", ferrari, oficina1);
        Car coche2 = new Car("7856BPM", ferrari, oficina1);
        Car coche3 = new Car("7856BPZ", ferrari, oficina1);
        Car coche4 = new Car("7856BPU", ferrari, oficina1);

        System.out.println("\nEl precio del modelo ferrari por día es: " +
ferrari.getpricePerDay() + "\n");

        // CASO 1: alquiler de coche con descuento porcentaje
        Customer1.rentCarOnSite(fecha1, fecha2, coche1, oficina1,
percentagePromotion, "Alquiler de coche con descuento por porcentaje");
```

```
        double price1 =
Customer1.getRentalsOnSite().get(Customer1.getRentalsOnSite().size()-1).getP
rice();
        System.out.println("El precio del alquiler con descuento por
porcentaje es: " + price1 + "\n");


        // CASO 2: alquiler de coche con descuento cantidad
        Customer2.rentCarOnSite(fecha1, fecha2, coche2, oficina1,
amountPromotion, "Alquiler de coche con descuento por cantidad");
        double price2 =
Customer2.getRentalsOnSite().get(Customer2.getRentalsOnSite().size()-1).getP
rice();
        System.out.println("El precio del alquiler con descuento por
cantidad es: " + price2 + "\n");


        // CASO 3: alquiler de coche sin descuento
        Customer3.rentCarOnSite(fecha1, fecha2, coche3, oficina1, null,
"Alquiler de coche sin descuento");
        double price3 =
Customer3.getRentalsOnSite().get(Customer3.getRentalsOnSite().size()-1).getP
rice();
        System.out.println("El precio del alquiler sin descuento es: " +
price3 + "\n");


        //CASO 4: aplicar primero una estrategia de descuento y después
cambiarla
        Customer4.rentCarOnSite(fecha1, fecha2, coche4, oficina2,
amountPromotion, "Alquiler de coche con descuento por cantidad");
        double price4 =
Customer4.getRentalsOnSite().get(Customer4.getRentalsOnSite().size()-1).getP
rice();
        System.out.println("El precio del alquiler con descuento por
cantidad es: " + price4);

Customer4.getRentalsOnSite().get(Customer4.getRentalsOnSite().size()-1).setP
romotion(percentagePromotion);
        double price4changed =
Customer4.getRentalsOnSite().get(Customer4.getRentalsOnSite().size()-1).getP
rice();
```

```
        System.out.println("El precio del alquiler con descuento por
porcentaje es: " + price4changed);
    }
}
```

La clase Main se usa para probar cómo funcionan los alquileres y las promociones con el Patrón Estrategia (Strategy). Aquí se crean clientes (Customer), coches (Car), oficinas (RentalOffice), y promociones (PercentagePromotion y AmountPromotion). También se define un modelo de coche (ferrari) con un precio diario que sirve como base para calcular el costo de los alquileres.

Se prueban cuatro casos:

1. Caso 1: Alquiler con descuento porcentual
   Customer1 alquila un coche con un descuento del 10% usando la estrategia PercentagePromotion. Se calcula el precio con el método getPrice() y se imprime el resultado.

2. Caso 2: Alquiler con descuento fijo
   Customer2 alquila un coche con un descuento fijo de 5 unidades, utilizando la estrategia AmountPromotion. El precio calculado también se imprime.

3. Caso 3: Alquiler sin descuento
   Customer3 alquila un coche sin aplicar ninguna promoción (promotion es null). El precio base del alquiler se imprime directamente.

4. Caso 4: Cambiar la promoción después de aplicar una
   Customer4 alquila un coche con un descuento fijo primero (AmountPromotion). Luego, cambia la promoción a un descuento porcentual (PercentagePromotion) usando el método setPromotion(). Se imprimen los precios antes y después del cambio para comparar.

Salida por terminal

```
El cliente Francisco se ha registrado correctamente
El cliente Marcos se ha registrado correctamente
El cliente Soraya se ha registrado correctamente
El cliente Victoria se ha registrado correctamente
El modelo de coche Ferrari se ha creado correctamente
La oficina situada en Av. Plutarco, 75 se ha creado correctamente
La oficina situada en Calle Larios, 3 se ha creado correctamente
El coche con matrícula 2587MDN se ha creado correctamente
El coche con matrícula 7856BPM se ha creado correctamente
El coche con matrícula 7856BPZ se ha creado correctamente
El coche con matrícula 7856BPU se ha creado correctamente

El precio del modelo ferrari por día es: 100.0

Francisco ha añadido un nuevo alquiler en oficina. Total alquileres de este cliente: 1
El precio del alquiler con descuento por porcentaje es: 180.0

Marcos ha añadido un nuevo alquiler en oficina. Total alquileres de este cliente: 1
El precio del alquiler con descuento por cantidad es: 195.0

Soraya ha añadido un nuevo alquiler en oficina. Total alquileres de este cliente: 1
El precio del alquiler sin descuento es: 200.0

Victoria ha añadido un nuevo alquiler en oficina. Total alquileres de este cliente: 1
El precio del alquiler con descuento por cantidad es: 195.0
El precio del alquiler con descuento por porcentaje es: 180.0
```