Modelado y Diseño del Software

Memoria de la Práctica 4

Pablo	Fernández Rueda	pauf@uma.es
Eduardo	García Rivas	eduardogarr@uma.es
Francisco Eloy	González Castillo	eloygonzalez@uma.es
María Paulina	Ordóñez Walkowiak	mpow@uma.es
Javier	Toledo Delgado	javier.toledo.delgado@uma.es
Daniela	Suárez Morales	danielasuarez@uma.es

Índice

Memoria Práctica 4

En el siguiente documento se incluyen vistas de los diagramas desarrollados para la práctica, y todo lo que es necesario para comprender y corregir la solución propuesta de los distintos apartados, incluyendo descripciones textuales, diagramas UML, códigos Java, así como cualquier consideración que es precisa realizar sobre decisiones de diseño del modelo y sobre la interpretación del enunciado de la práctica.

Explicaciones y Consideraciones

Restricciones

Para la implementación del sistema hemos optado por no implementar las restricciones que no son requeridas/necesarias para la realización de los distintos patrones de diseño, se le ha consultado a la profesora previamente en clase.

Un cliente no puede tener alquileres solapados.

La fecha de inicio de un alquiler tiene que ser anterior a la fecha final del alquiler.

La oficina de recogida de un coche de alquiler tiene que ser la misma que la oficina donde está asignado el coche de alquiler.

Si la oficina de recogida y de entrega de un alquiler web son diferentes, la hora de entrega del coche de alquiler tiene que ser anterior a las 13 horas.

Apartados Ejercicios: Patrones de Diseño

Ejercicio 1

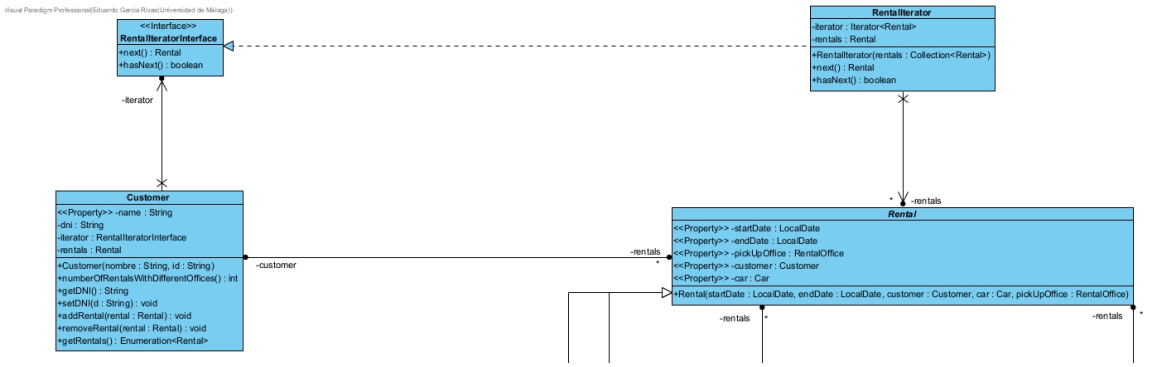
Supongamos ahora que queremos definir una operación numberOfRentalsWithDifferentOffices(): Integer en la clase Customer que devuelve el número de alquileres web que ha hecho un cliente self donde la oficina de recogida y de entrega es diferente. En este momento del diseño del sistema todavía no sabemos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

a) ¿Qué patrón de diseño utilizarías para diseñar esta operación? Justifica tu respuesta.

El patrón de diseño más adecuado para la realización de este ejercicio es el patrón **Iterator**, ya que permite recorrer los alquileres de un cliente sin necesidad de conocer de antemano cuántos existen ni la estructura de datos específica utilizada para almacenarlos.

Este patrón abstrae el acceso a los elementos mediante una interfaz genérica, lo que facilita la implementación con cualquier tipo de colección, como las proporcionadas por la biblioteca estándar de Java. De este modo, se garantiza la flexibilidad y extensibilidad del sistema al desacoplar la lógica de recorrido de la estructura interna de los datos.

b) Muestra la parte del diagrama de clases que se ve modificada como resultado de la aplicación del patrón utilizado.



c) Muestra el código Java de la operación numberOfRentalsWithDifferentOffices():Integer de la clase Customer.

```
public int numberOfRentalsWithDifferentOffices(){
        RentalIterator iterador = new RentalIterator(rentals);
        int count = 0;
        while (iterador.hasNext()) {
            Rental r = iterador.next();

            if(r instanceof WebRental){
                WebRental wr = (WebRental) r;
                if(wr.getPickUpOffice() != wr.getDeliveryOffice()){
                    count++;
                }
```

```
            }
        }
        return count;
    }
```

Como se puede observar, la clase sólo comprueba los alquileres de tipo WebRental, ya que, son los únicos que tienen la capacidad de tener el deliveryOffice distinto que el pickUpOffice. Al contrario que en los RentalOnSite, que tienen el mismo deliveryOffiec y pickUpOffice siempre.

Utilizando la clase RentalIterator se podrá iterar en los alquileres del cliente, además con la variable count se contará los alquileres que tengan distinta oficina de recogida y de entrega.

Comprobación:

```
Customer c1 = new Customer("Pepe", "1");
        Customer c2 = new Customer("Juan", "2");

        RentalOffice r1 = new RentalOffice("Oficina 1",20);
        RentalOffice r2 = new RentalOffice("Oficina 2",30);

        Car car1 = new Car("HOLA", new Model("Corolla", 200), r1);

        Car car2 = new Car("ADIOS", new Model("Yaris", 100), r1);

        Car car3 = new Car("HOLA2", new Model("Corolla", 300), r2);

             // Este WebRental tiene oficina de recogida y entrega
diferentes
        WebRental w1 = new WebRental(LocalDate.of(2024, 12, 5),
                LocalDate.of(2024, 12, 6), c1, car1, r1, null, r2);

        // Este WebRental tiene oficina de recogida y entrega iguales
        WebRental w2 = new WebRental(LocalDate.of(2024, 12, 5),
                LocalDate.of(2024, 12, 6), c1, car2, r1, null, r1);

        RentalOnSite r3 = new RentalOnSite(LocalDate.of(2024, 12, 7),
                LocalDate.of(2024, 12, 6), c1, car3, r2, "algo");

           // Añado un alquiler web al cliente1 que tiene diferente
oficina de recogida y entrega
         // y otro alquiler web al cliente1 que tiene la misma oficina
de recogida y entrega
        c1.addRental(w1);
        c1.addRental(w2);
```

```
        // Añado un alquiler onSite al cliente2
        c2.addRental(r3);

          System.out.println("\nEl numero de alquileres con diferentes
oficinas de recogida del cliente " +
                                    c1.getName()+ " y entrega es: " +
c1.numberOfRentalsWithDifferentOffices());

           System.out.println("El numero de alquileres con diferentes
oficinas de recogida del cliente " +
                                    c2.getName()+ " y entrega es: " +
c2.numberOfRentalsWithDifferentOffices());
```

Salida del programa:

> El número de alquileres con diferentes oficinas de recogida del cliente Pepe y entrega es: 1
> El número de alquileres con diferentes oficinas de recogida del cliente Juan y entrega es: 0

Explicación:
En el siguiente main, se ha añadido al cliente1, Pepe, dos WebRental y al cliente2, Juan, se ha añadido un RentalOnSite. En el cliente 1, a uno de los WebRental, se le ha puesto con distinta oficina de recogida y entrega, y al otro, con misma oficina de recogida y entrega. Al llamar al método numberOfrentalsWithDifferentOffices() en el cliente1, nos devuelve que solo tiene 1, en cambio, al cliente2 nos devuelve 0, ya que, un RentalOnSite por defecto tiene la misma oficina de recogida y entrega.

Ejercicio 2

A menudo, los coches que la empresa de alquiler de coches pone a disposición de sus clientes se tienen que poner fuera de servicio (por reparaciones, para pasar la ITV, etc.). Queremos representar esta situación en nuestro sistema para que cuando los coches estén fuera de servicio no puedan ser alquilados aunque registraremos, si hay, un coche que lo sustituye. Es por eso que nuestro sistema tendrá que proporcionar dos funcionalidades. La funcionalidad (1) poner un coche fuera de servicio (si ya está fuera de servicio o si está en servicio pero es sustituto de algún coche que está fuera de servicio, la funcionalidad no tendrá ningún efecto). 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 (será cualquier coche del mismo modelo del coche que se pone fuera de servicio que esté asignado a la misma oficina y que esté en servicio). La funcionalidad (2) pone un coche que estaba fuera de servicio en servicio.

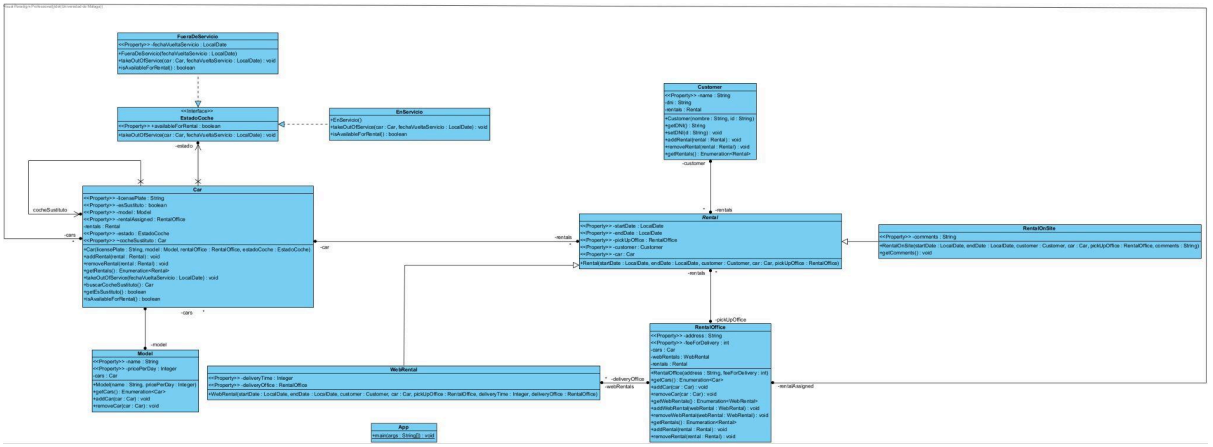
En concreto, nos centraremos en implementar la funcionalidad (1) añadiendo la operación takeOutOfService(backToService:date) de la clase Car. No hace falta implementar la funcionalidad (2).

a) ¿Qué patrón de diseño recomendarías para representar la información descrita (si es que recomiendas alguno)?

El patrón **Estado** es el más adecuado para este ejercicio porque permite gestionar de manera clara los cambios en el comportamiento del coche según su estado (en servicio o fuera de servicio). Entonces, en lugar de manejar los distintos comportamientos del coche dependiendo de su estado mediante condiciones como if-else o switch, se crean clases específicas para representar cada estado del coche.

Esto tiene ciertas ventajas, una de las más importantes es el hecho de que es extendible, pues se podrían añadir más estados para definir comportamientos distintos y no habría que cambiar la lógica ya establecida en los estados ya existentes.

b) Muestra el diagrama de clases resultante de añadir estas funcionalidades.



c) Muestra el código Java de la operación takeOutOfService de la clase Car.

En car:

```
public void takeOutOfService(LocalDate fechaVueltaServicio) {
    estado.takeOutOfService(this, fechaVueltaServicio);
}
```

```
    public Car buscarCocheSustituto(){
         Enumeration<Car> cochesEnLaOficina = rentalAssigned.getCars();
// lista de coches

        while (cochesEnLaOficina.hasMoreElements()) {
            Car coche = cochesEnLaOficina.nextElement();

                // Comprobamos que sea del mismo modelo y que esté en
servicio
                            if  (coche.getModel().equals(this.model)  &&
coche.isAvailableForRental()) {
                coche.setEsSustituto(true);
                  return coche; // Retornamos el primer coche que cumple
las condiciones
            }
        }
            // Si no encontramos un coche que cumpla los criterios,
retornamos null
        return null;
    }
```

En EnServicio:

```
              public    void    takeOutOfService(Car    car,    LocalDate
fechaVueltaServicio) {
           if (!car.getEsSustituto()) { // Un coche sustituto no puede
ponerse fuera de servicio
             car.setEstado(new FueraDeServicio(fechaVueltaServicio)); //
Cambiar estado
              Car cocheSustituto = car.buscarCocheSustituto(); // Buscar
sustituto
                car.setCocheSustituto(cocheSustituto); // Asignar coche
sustituto si se encuentra
        }
    }
```

En FueraDeServicio:

```
              public    void    takeOutOfService(Car    car,    LocalDate
fechaVueltaServicio) {
        // Si el coche ya está fuera de servicio, no se realiza ninguna
acción
         System.out.println("El coche ya está fuera de servicio hasta: "
+ fechaVueltaServicio);
    }
```

Como se puede ver, la función en Car llama a una de las dos funciones de las clases EnServicio y FueraDeServicio, de tal manera que un objeto de tipo Car se va a comportar de una forma u otra dependiendo del estado que tenga asignado.

- Si su estado es EnServicio, comprueba que el coche no es ya un sustituto, y si eso se cumple, va a darlo de baja temporalmente y va a buscar un sustituto viable.
- Si ya está FueraDeServicio no va a hacer nada, aunque informa en la salida del programa, por claridad.

Se ha cambiado ligeramente también la forma de añadir los rentals en RentalOffice para evitar crear un rental ficticio cuando se alquila un coche de sustitución:

```
    public void addRental(Rental rental){
        Car car = rental.getCar();
        if (car.isAvailableForRental()) {
            rentals.add(rental);
        } else {
                    System.out.println("El coche seleccionado no está
disponible actualmente, procedemos a buscar un coche sustituto");
            Car carSustituto = car.getCocheSustituto();
            if (carSustituto == null) {
                     throw new IllegalArgumentException("No hay coches
disponibles");
            }
            rental.setCar(carSustituto);
            rentals.add(rental);
        }
    }
```

Comprobación:

```
public class App {
    public static void main(String[] args) throws Exception {
        // Crear oficinas de alquiler
           RentalOffice pickUpOffice = new RentalOffice("123 Main St",
50);
          RentalOffice deliveryOffice = new RentalOffice("456 Elm St",
30);

        // Crear coches con diferentes estados
        Model carModel = new Model("Toyota Corolla", 100);
            Car car1 = new Car("1234-ABC", carModel, pickUpOffice, new
EnServicio());
            Car car2 = new Car("5678-DEF", carModel, pickUpOffice, new
EnServicio());
```

```
            Car car3 = new Car("1111-XYZ", carModel, pickUpOffice, new
EnServicio());

        // Añadir los coches a la oficina de recogida
        pickUpOffice.addCar(car1);
        pickUpOffice.addCar(car2);
        pickUpOffice.addCar(car3);

         // Mostrar los coches en la oficina de recogida antes de tomar
fuera de servicio
          System.out.println("Coches en la oficina de recogida antes de
tomar fuera de servicio:");
        Enumeration<Car> carsInOffice = pickUpOffice.getCars();
        while (carsInOffice.hasMoreElements()) {
            Car car = carsInOffice.nextElement();
                        System.out.println("Coche en oficina: " +
car.getLicensePlate() + " - Estado: " +
car.getEstado().getClass().getSimpleName());
        }

        // Crear un cliente
        Customer customer = new Customer("John Doe", "234R");
         System.out.println("Cliente creado: " + customer.getName() + "
- ID: " + customer.getDNI());

        // Probar el método takeOutOfService en el coche 1
                    System.out.println("\nPoniendo el coche " +
car1.getLicensePlate() + " fuera de servicio...");
            car1.takeOutOfService(LocalDate.now().plusDays(7));   // El
coche estará fuera de servicio por 7 días

          // Mostrar el estado del coche después de ponerlo fuera de
servicio
         System.out.println("Estado del coche " + car1.getLicensePlate()
+ " después de ponerlo fuera de servicio: " +
car1.getEstado().getClass().getSimpleName());

          // Intentar alquilar el coche 1 después de ponerlo fuera de
servicio (no debe estar disponible)
        try {
                WebRental webRental1 = new WebRental(LocalDate.now(),
LocalDate.now().plusDays(5),   customer,   car1,   pickUpOffice,   15,
deliveryOffice);
```

```
            pickUpOffice.addRental(webRental1); // Añadimos el alquiler
a la oficina
        } catch (IllegalArgumentException e) {
             System.out.println("Error al crear alquiler para coche " +
car1.getLicensePlate() + ": No hay coches disponibles");
        }

        // Comprobar si se ha asignado un coche sustituto
        Car cocheSustituto = car1.getCocheSustituto();
        if (cocheSustituto != null) {
                  System.out.println("Coche sustituto encontrado: " +
cocheSustituto.getLicensePlate());
        } else {
               System.out.println("No se ha encontrado coche sustituto
para el coche " + car1.getLicensePlate());
        }

         // Probar que el coche 2 sea el sustituto de car1 (y que car2
esté disponible para alquiler)
           System.out.println("\nComprobando si el coche 2 es un coche
sustituto de " + car1.getLicensePlate());
        if (car2.getEstado().isAvailableForRental()) {
            System.out.println("Coche 2 es sustituto.");
        } else {
            System.out.println("Coche 2 NO es sustituto.");
        }

        // Ahora, volver a poner el coche 1 en servicio
                       System.out.println("\nPoniendo el coche " +
car1.getLicensePlate() + " de vuelta en servicio...");
        car1.setEstado(new EnServicio());

        // Mostrar el estado después de devolverlo a servicio
         System.out.println("Estado del coche " + car1.getLicensePlate()
+ " después de ponerlo en servicio: " +
car1.getEstado().getClass().getSimpleName());

        // Intentar alquilar el coche 1 ahora que está en servicio
        try {
                  WebRental webRental2 = new WebRental(LocalDate.now(),
LocalDate.now().plusDays(5), customer, car1, pickUpOffice, 10,
deliveryOffice);
```

```
            pickUpOffice.addRental(webRental2); // Añadimos el alquiler
a la oficina
                System.out.println("Alquiler creado con éxito para el
coche: " + car1.getLicensePlate());
        } catch (IllegalArgumentException e) {
             System.out.println("Error al crear alquiler para coche " +
car1.getLicensePlate() + ": " + e.getMessage());
        }

        // Mostrar los alquileres registrados
          System.out.println("\nAlquileres registrados en la oficina de
recogida:");
                            Enumeration<Rental>    rentalsInOffice    =
pickUpOffice.getRentals();
        while (rentalsInOffice.hasMoreElements()) {
            Rental rental = rentalsInOffice.nextElement();
                                System.out.println("Alquiler:    "    +
rental.getCar().getLicensePlate()     +     "     -     Cliente:     "     +
rental.getCustomer().getName());
        }
    }
}
```

Salida del programa:

Coches en la oficina de recogida antes de tomar fuera de servicio:
Coche en oficina: 1234-ABC - Estado: EnServicio
Coche en oficina: 5678-DEF - Estado: EnServicio
Coche en oficina: 1111-XYZ - Estado: EnServicio
Cliente creado: John Doe - ID: 234R

Poniendo el coche 1234-ABC fuera de servicio...
Estado del coche 1234-ABC después de ponerlo fuera de servicio: FueraDeServicio
El coche seleccionado no está disponible actualmente, procedemos a buscar un coche sustituto
Coche sustituto encontrado: 5678-DEF

Comprobando si el coche 2 es un coche sustituto de 1234-ABC
Coche 2 es sustituto.

Poniendo el coche 1234-ABC de vuelta en servicio...
Estado del coche 1234-ABC después de ponerlo en servicio: EnServicio
Alquiler creado con éxito para el coche: 1234-ABC

Alquileres registrados en la oficina de recogida:

Alquiler: 5678-DEF - Cliente: John Doe
Alquiler: 1234-ABC - Cliente: John Doe

Ejercicio 3

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] y 2).

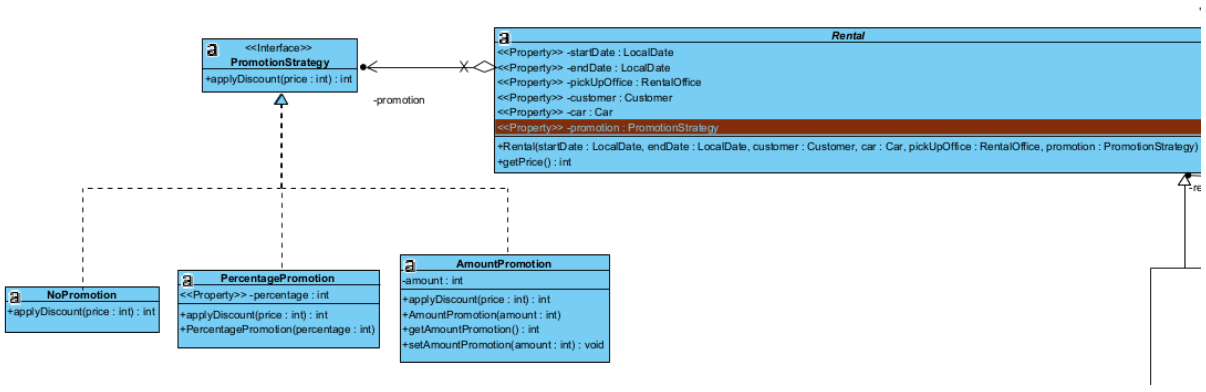
Además, la empresa de alquiler de coches puede añadir al cálculo de precios la posibilidad de hacer promociones que implican descuentos de estos precios. Inicialmente, la empresa ofrecerá dos tipos de promociones: por cantidad y por porcentaje. La promoción por cantidad permitirá decrementar el precio del alquiler en la cantidad indicada en la promoción. La 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). Se pide:

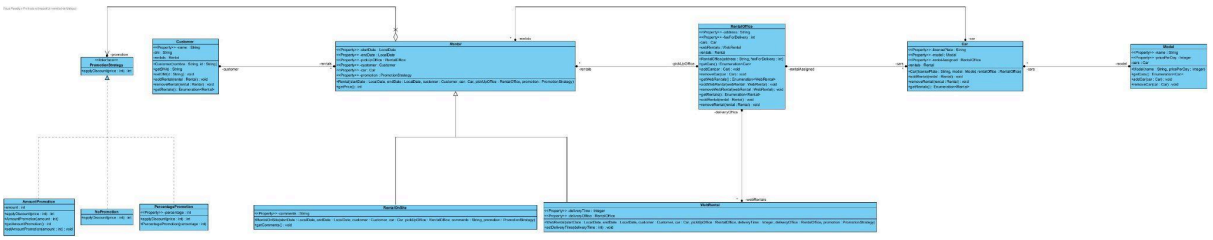
a) ¿Qué patrón de diseño recomendarías para este caso? Justificad vuestra respuesta.

El patrón **Estrategia** es el más adecuado para este ejercicio, porque permite implementar diferentes maneras de calcular los descuentos de los alquileres de forma muy simple. En lugar de usar un bloque switch para evaluar cada tipo de descuento posible en la misma función, se usa una interfaz que define un método para aplicar el descuento. De esta forma, sólo necesitamos asignar la clase que implementa el método, para que de forma externa se calcule.

Este patrón permite separar la lógica de calcular los descuentos de la función principal donde se calcula el precio del alquiler, lo que hace que sea mucho más simple implementar nuevas formas de calcular los descuentos, al solo tener que hacer una nueva clase con un único método.

b) Muestra el diagrama de clases resultante de la aplicación del patrón.





c) Muestra el nuevo código Java de la operación getPrice():Integer.

```
public int getPrice() {
        // Calcular el número de días de alquiler
        long rentalDays = ChronoUnit.DAYS.between(startDate, endDate);
        int basePrice = (int) (car.getModel().getPricePerDay() *
rentalDays);

        // Aplicar promoción
        return promotion.applyDiscount(basePrice);
    }
```

Vemos como el código de la función es únicamente calcular los días totales del alquiler para calcular el precio base, y llamar a promotion.applyDiscount() para calcular el descuento. En este caso, promotion es una clase que implementa la interfaz PromotionStrategy, y se le asigna a esta clase mediante el constructor. También existe otro método setPromotion() que permite que otra parte del sistema cambie el descuento aplicado.

Por ejemplo, setPromotion() se podría usar para establecer el descuento más favorable mientras no se haya realizado el pago.

Comprobación:

```
public static void main(String[] args) throws Exception {
        RentalOffice alquileresMarta = new RentalOffice("calle etsii",
100);
        Model citroen = new Model("Citroen", 100);
        Car c3 = new Car("666", citroen, alquileresMarta);

        Customer paquito = new Customer("Paquito", "123456789J");

        // Fechas de prueba
        LocalDate startDate = LocalDate.of(2024, 12, 1);
        LocalDate endDate = LocalDate.of(2024, 12, 5);

        // Crear un alquiler sin promoción
        Rental alquiler = new RentalOnSite(startDate, endDate, paquito,
c3, alquileresMarta, "Sin comentarios...",
                new NoPromotion());
```

```
        System.out.println("Precio sin promoción: " +
alquiler.getPrice());

        // Aplicar una promoción de cantidad
        alquiler.setPromotion(new AmountPromotion(30)); // Descuento de
30
        System.out.println("Precio con promoción de cantidad: " +
alquiler.getPrice());

        // Aplicar una promoción de porcentaje
        alquiler.setPromotion(new PercentagePromotion(10)); // 10% de
descuento
        System.out.println("Precio con promoción de porcentaje: " +
alquiler.getPrice());

        // Cambiar la promoción a otra más favorable
        alquiler.setPromotion(new AmountPromotion(100)); // Descuento
grande
        System.out.println("Precio con promoción de cantidad más
favorable: " + alquiler.getPrice());
    }
```

Salida del programa:

Precio sin promoción: 400
Precio con promoción de cantidad: 370
Precio con promoción de porcentaje: 360
Precio con promoción de cantidad más favorable: 300