Técnicas de Programación Avanzada

TECNICATURA UNIVERSITARIA EN DESARROLLO DE SOFTWARE

academia CIMNE IBER

UPATECO UNIVERSIDAD PROVINCIAL

Ministerio de Economía y Servicios Públicos
Gobierno de Salta

Tarea 4: Interfaces, clases abstractas y genéricos

Sistema de Hotel

Este proyecto tiene la finalidad de gestionar las reservas en un hotel. En general, el sistema nos debe permitir realizar reservas de habitaciones, gestionar su disponibilidad, así como el cálculo de los costos de las reservas, que pueden incluir comodidades y puntos acumulados para el cliente que la realiza.

El sistema contará con la siguiente arquitectura:

Interfaces

Habitación<T>

La interfaz deberá ser genérica que acepte parámetros limitados a subtipos de Comodidad (aplicación de genéricos).

- Obtener el tipo de habitación, `getTipo(): String`
- Precio base, `getPrecio(): double`
- Comodidades, `getComodidades(): List<T>`
- Verificar su disponibilidad, `estaDisponible(fechaInicio: LocalDate, fechaFin: LocalDate): boolean`
- Calcular precio total según fechas, `calcularPrecioTotal(fechaInicio: LocalDate, fechaFin: LocalDate): double`

Reserva

- Obtener el cliente, `getCliente(): Cliente`
- Obtener la habitación, `getHabitacion(): Habitacion`
- Obtenerlas fechas, `getFechaInicio(): LocalDate, getFechaFin(): LocalDate`
- Calcular el costo total, `calcularCosto(): double`
- Cancelar la reserva, `cancelarReserva(): void`
- Modificar la fecha, `modificarFechas(fechaInicio: LocalDate, fechaFin: LocalDate): void`

Cliente

- Obtener el nombre del cliente, `getNombre(): String`
- Obtener puntos del cliente, `getPuntos(): Integer`
- Acumular puntos en su cuenta, `acumularPuntos(reserva: Reserva): void`

Clases abstractas

HabitacionGeneral

Se definirá la clase abstracta `HabitacionGeneral` considerando los subtipos de Comodidad, implementando la interface Habitacion.

```
abstract class HabitacionGeneral<T extends Comodidad> implements Habitacion<T> {
  protected String tipo;
  protected double precio;
  protected List<T> comodidades;
  public HabitacionGeneral(String tipo, double precio, List<T> comodidades){
    this.tipo = tipo;
    this.precio = precio;
```

```
    this.comodidades = comodidades;
  }
  // getters y setters
  // Métodos abstractos necesarios
```

Comodidad

Se definirá la clase abstracta Comodidad que cuente con la información básica de una de la siguiente forma.

```
abstract class Comodidad {
  protected String nombre;
  protected double precio;
  // getters y setters
  public abstract double calcularCosto(LocalDate fechaInicio, LocalDate fechaFin);
}
```

Clases finales

GestorDisponibilidad

Se encargará de manejar las operaciones en cuanto a disponibilidad de habitaciones, en esta ocasión solo necesitamos contarReservas, se brinda la implementación a continuación.

```
class GestorDisponibilidad {
  private List<HabitacionGeneral<? extends Comodidad>> habitaciones;
  private List<Reserva> reservas;

  // Constructor que recibe los objetos de los atributos

  public int contarReservas(HabitacionGeneral<?> hab, LocalDate fechaInicio, LocalDate fechaFin) {
    int cantReservas = 0;
    for (Reserva reserva : reservas) {
      if (reserva.getHabitacion().equals(hab) &&
       (reserva.getFechaInicio().isBefore(fechaFin) && reserva.getFechaFin().isAfter(fechaInicio))) {
           cantReservas++;
      }
    }
    return cantReservas;
  }
```

GestorReservas

Se encarga de las operaciones relacionadas con las reservas y su costo, esta cuenta con la siguiente estructura de clase.

```
class GestorReservas {
  private List<Reserva> reservas;

  // Constructor que instancia la lista de reservas vacía

  void realizarReserva(Cliente cliente, Habitacion hab, fechaInicio y fechaFin LocalDate);
  void cancelarReserva(Reserva reserva);
  void modificarReserva(Reserva reserva, fechaInicio y fechaFin LocalDate);
  double calcCostoReserva(Reserva reserva);
}
```

Nota: Los métodos cancelarReserva y modificarReserva trabajarán con los métodos implementados en Reserva que la interface nos solicitó.

HabitacionSimple

class HabitacionSimple extends HabitacionGeneral<ComodidadBasica>

HabitacionDoble

class HabitacionDoble extends HabitacionGeneral<ComodidadDoble>

HabitacionSuite

class HabitacionSuite extends HabitacionGeneral<ComodidadSuite>

Según la implementación cada método se comportará de una manera diferente.

Habitación	Método estaDisponible	Método calcularPrecioTotal
HabitacionSimple	habitación estará disponible si hay menos de 4 reservas dentro del rango de fechas	Se calcula el precio total sumando el precio de la habitación y el costo de las comodidades
HabitacionDoble	La habitación estará disponible si no hay reservas para dichas fechas, y se quedarán al menos 2 noches	Si se queda más de 3 noches, se aplica un 10% de descuento sobre el precio total
HabitacionSuite	La habitación estará disponible si hay menos de 5 reservas dentro del rango, y se quedará al menos 3 noches	Si la estancia es mayor a 5 noches, se aplica un descuento del 15% sobre el precio total.

Nota: Para poder utilizar el gestor de disponibilidad en cada una de estas clases crearemos una instancia global en App como sigue.

```
public class App {
  private static GestorDisponibilidad gestorDisp;
  public static void main(String[] args){
    List<Habitacion> habitaciones = inicializmos la lista
    List<Reserva> reservas = inicializamos la lista
    gestorDisp = new GestorDisponibilidad(habitaciones, reservas);
    // resto del código en método main principal
  }
  public static GestorDisponibilidad getGestor(){
    return gestorDisp;
  }
}
```

Ejemplo de uso en HabitacionSimple

```
public boolean estaDisponible(LocalDate fechaInicio, LocalDate fechaFin){
  int cantReservas = App.getGestor().contarReservas(this, fechaInicio, fechaFin);
  return cantReservas < 10;
}
```

Clase Cliente

Recordemos la info de la interface de Cliente.

- Obtener el nombre del cliente, `getNombre(): String`
- Obtener puntos del cliente, `getPuntos(): Integer`
- Acumular puntos en su cuenta, `acumularPuntos(reserva: Reserva): void`

La clase tendrá los atributos nombre y puntos los cuales almacena el objeto.
El método `acumularPuntos` será llamado luego de realizar una reserva y aumentará en una proporción 1 a 100 (1 punto cada 100$ de costo). Es decir:

```
puntos = puntos + reserva.CalcularCosto() / 100;
```

Clase ComodidadBasica

Recordemos lo que tenemos en la clase abstracta Comodidad.

```
abstract class Comodidad {
  protected String nombre;
  protected double precio;
  // getters y setters
  public abstract double calcularCosto(LocalDate fechaInicio, LocalDate fechaFin);
}
```

El costo se calculará como la cantidad de días por el precio de dicha comodidad. Es la única operación pendiente a implementar.

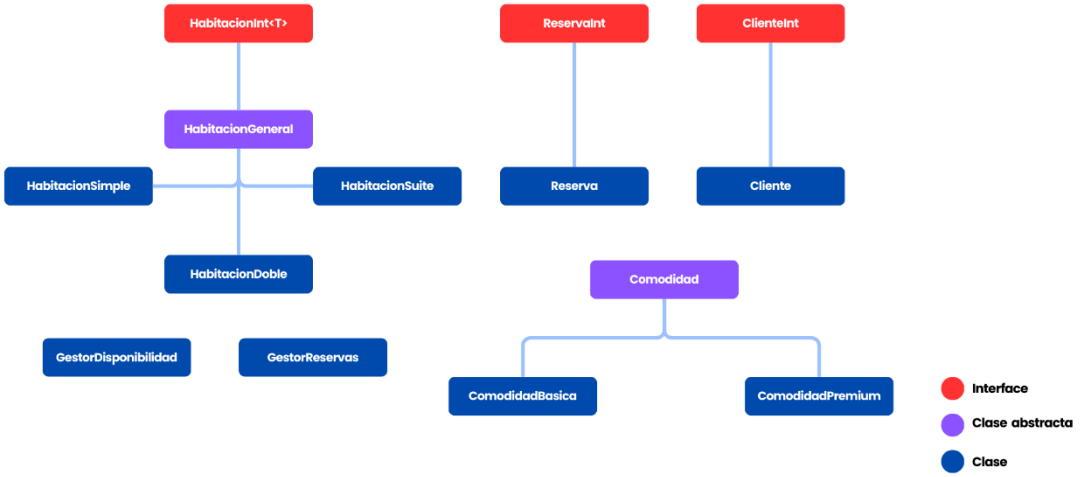
Clase ComodidadPremium

Usando la misma clase abstracta, sumaremos el atributo `especialidad`, una cadena que nos podría decir el material del mueble, o la velocidad de WiFi disponible.

El costo se calculará similar a ComodidadBasica, solo que tendrá un 25% de recargo final una vez calculado el costo.

Estructura del Proyecto

Con esta implementación, tendremos un esquema similar al siguiente:



Funcionamiento

En la clase App que gestiona el proyecto, realizará las siguientes acciones:

- Instanciar una lista de habitaciones
- Instanciar una lista de reservas
- Instanciar el gestor de disponibilidad
- Instanciar el gestor de reservas
- Crear 3 instancias de habitaciones, una de cada clase hija
- Crear 6 instancias de comodidades de una u otra subclase definida
- Crear 2 instancias de clientes
- Realizar 5 reservas para cada cliente, cada una en un tipo de habitación a gusto
- Calcular el costo de las habitaciones reservadas y mostrar
- Calcular los puntos acumulados para cada cliente y mostrarlos.