Asignatura: Programación II – Curso 2023/24 – 2er Semestre

Dpto. LSIIS. Unidad de Programación

Proyecto SmartParking

***Objetivo*:** El objetivo de este proyecto es la familiarización del alumno con la programación orientada a objetos en Java, incluyendo arrays de objetos, las clases contenedoras genéricas de la asignatura, organización en paquetes (modularidad), ejecución de pruebas unitarias, herencia de clases e interfaces y manejo de excepciones.

***Desarrollo del trabajo*:** El proyecto se podrá realizar en grupos de dos alumnos (matriculados en el mismo semestre) o de forma individual. Esto se especificará mediante la anotación Java explicada en el apartado 2. **Este proyecto lo deben hacer todos los alumnos (evaluación continua y sólo examen final)**.

***Código de apoyo*:** Todo este código se suministra en un archivo **SmartParkingAlumnos.zip.**

***Autoevaluación*:** El alumno debe comprobar que su ejercicio no contiene ninguno de los errores explicados en el último apartado de este enunciado.

***Entrega*:** El proyecto se entregará a través de la página web: https://entrega.fi.upm.es/. Una vez realizada la primera entrega en grupo, el grupo no se podrá deshacer, es decir, los dos alumnos del grupo estarán obligados a realizar todas las entregas de este proyecto juntos. La misma norma se aplica cuando un alumno realiza la primera entrega de forma individual, es decir, ese alumno ya no podrá realizar y entregar el proyecto en grupo. El alumno de un grupo que realice la primera entrega del proyecto será el que tenga que hacer el resto de las entregas del mismo, si son necesarias.

***Plazos:* El periodo de entrega finaliza el viernes 3 de mayo a las 10:00 AM**. En el momento de realizar la entrega, el proyecto será sometida a una serie de pruebas que deberá superar para que la entrega sea admitida. El alumno dispondrá de **un número máximo de 10 entregas**. Asimismo, por el hecho de que el proyecto sea admitido, eso no implicará que el proyecto esté aprobado.

***Evaluación*:** Los proyectos entregados serán corregidas por un profesor, que revisará el código con el fin de identificar posibles errores de estilo o problemas de eficiencia. El peso de este proyecto en la nota final de la asignatura es el que indica en la Guía de la Asignatura. **Se mantendrá el mismo enunciado del proyecto para la convocatoria extraordinaria**.

Detección Automática de Copias: Cada proyecto entregado se comparará con el resto de los proyectos entregados en las distintas convocatorias del curso. Esto se realizará utilizando un sofisticado programa de detección de copias.

Consecuencias de haber copiado: Todos los alumnos involucrados en una copia, bien por copiar o por ser copiados, serán sancionados según las normas publicadas en la guía de aprendizaje de la asignatura.

1. Introducción

SmartParking es una aplicación para facilitar el estacionamiento de vehículos en las calles de una localidad que presupone que existe una infraestructura de sensores de presencia en las calles y dispositivos de lectura de códigos QR conectados a Internet (Internet de las cosas). Gracias a esta aplicación, un usuario puede utilizar su móvil para reservar una plaza de aparcamiento por un periodo de tiempo en una zona de una localidad antes de desplazarse a dicha zona. Se van a distinguir dos tipos de reservas:

- **Reserva Anticipada**: el usuario intenta reservar una plaza en una zona de la localidad al menos 24 horas antes del inicio de la reserva. Si no existe ninguna plaza disponible en la zona indicada en el periodo solicitado[1], la solicitud de reserva se incluye en una lista de

1 Supondremos que el máximo tiempo que se puede reservar una plaza de forma anticipada es 2 horas, pero el alumno no tendrá que comprobarlo en su código.

espera asociada a las plazas de esta zona. De esta forma, si se queda libre alguna plaza en una zona, se recorrerá la lista de espera para encontrar solicitudes que puedan ser satisfechas. Si una solicitud en espera puede ser satisfecha, se le notificará al usuario la consecución de su reserva.

- **Reserva Inmediata**: el usuario intenta reservar una plaza en una zona de la localidad en las 24 horas previas al inicio de la reserva. Si no existe ninguna plaza disponible en la zona indicada en el periodo solicitado[2], la aplicación busca la plaza libre más cercana en las zonas limítrofes de la zona indicada dentro de un radio máximo prefijado por el usuario.

Una reserva confirmada puede ser anulada sin penalización con una antelación mínima de 2 horas al periodo de la reserva salvo que se anule una reserva que se aceptó en las 2 horas previas al periodo solicitado tras pasar por la lista de espera, en cuyo caso no se aplicará ninguna penalización. Ahora bien, el alumno no tendrá que implementar el cálculo de las penalizaciones en este ejercicio.

1.1. División en zonas de una localidad

Esta aplicación va a gestionar las reservas que se soliciten para una localidad de NxM zonas, que se dividirá en zonas geográficas cuadradas, en donde cada zona *(i, j)* se encontrará en la fila *i* y la columna *j* de la matriz de zonas asociada a la localidad. Para calcular la distancia entre dos zonas *z* y *z'* con coordenadas *(i, j)* e *(i', j')* respectivamente, se utilizará la distancia de Manhattan[3], que se calcula de la siguiente forma:

$$dist(z, z') = |i - i'| + |j - j'|$$

A continuación, se muestra un ejemplo de cómo podría ser el mapa de zonas de una localidad con dimensiones 5x5 (véase la Figura 1).

	0	**1**	**2**	**3**	**4**
0	24	19	11	18	23
1	20	12	4	10	17
2	5	1	X	3	9
3	13	6	2	8	16
4	21	14	7	15	22

Figura 1 Ejemplo de solicitud de reserva inmediata para zona (2, 2)

2 Supondremos que el máximo tiempo que se puede reservar una plaza de forma inmediata es 1 hora, pero el alumno no tendrá que comprobarlo en su código.

3 La distancia de Manhattan toma su nombre de la geometría cuadriculada del ilustre distrito neoyorquino. Para más información véase https://en.wikipedia.org/wiki/Taxicab_geometry

Si se solicita una reserva inmediata con radio máximo 4 en la zona (2, 2) de la localidad de la Figura 1 y no existe ninguna plaza disponible en esta zona, la aplicación deberá buscar una plaza en las zonas limítrofes en el orden indicado por la numeración de la Figura 1. Si el radio máximo fuera 1 en lugar de 4, la aplicación solo considerará las zonas 1, 2, 3 y 4.

En la Figura 2 se puede observar qué zonas y en qué orden se considerarán si la reserva inmediata con radio máximo 3 se solicita para la zona (0, 0).

	0	**1**	**2**	**3**	**4**
0	X	2	5	9	
1	1	4	8		
2	3	7			
3	6				
4					

Figura 2 Ejemplo de solicitud de reserva inmediata para zona (0, 0) y radio 2

Una vez reservada una plaza para un periodo de tiempo, el usuario podrá ocupar la plaza con su vehículo durante dicho periodo. Para ello, cuando el usuario llegue a la plaza, tendrá que presentar un código QR generado por la app de su móvil para su reserva antes de poder ocupar la plaza. Asimismo, cada plaza dispone de una cámara capaz de leer la matrícula del vehículo de tal forma que la aplicación podrá validar que la reserva se ha realizado para el vehículo que realmente se está estacionando en esta plaza.

Por otro lado, supondremos que, si un vehículo se estaciona en una plaza más allá de su periodo reservado, será retirado inmediatamente por una grúa.

Si el usuario decide llevarse su vehículo antes que expire el periodo reservado, la plaza quedará disponible inmediatamente para otra reserva. Por tanto, el usuario no podrá volver a estacionar su vehículo en la misma plaza, aunque todavía le quede tiempo en su reserva.

1.2. Precios por zona

Adicionalmente, cada zona de la ciudad tiene asociado un precio horario por aparcamiento, que el cliente ha de abonar al efectuar una reserva. Al efectuar una solicitud de reserva inmediata, la aplicación deberá tener en cuenta el precio de las zonas a la hora de buscar un hueco libre. Es decir, entre todas las plazas que se encuentren próximas al usuario, la aplicación intentará reservar en aquella que tenga un precio menor. En resumen, la búsqueda de una plaza se efectúa siguiendo los siguientes criterios en orden de prioridad:

1. Distancia (de Manhattan) a la zona inicial de búsqueda.
2. Precio del aparcamiento en la zona.
3. Sentido antihorario de recorrido como se especifica en las Figuras 1 y 2.

Para completar el ejemplo, en la Figura 3 se muestra una matriz de precios correspondiente a una localidad. En la Figura 4, se muestra el orden de prioridad con el que se selecciona una zona partiendo del punto (0,0) con radio 3 y teniendo en cuenta los criterios anteriores y la matriz de precios de la Figura 3. En particular, el criterio del sentido antihorario se aplica únicamente cuando hay igualdad de precios entre zonas a la misma distancia.

	0	**1**	**2**	**3**	**4**
0	1.0	2.0	1.5	3.0	1.0
1	3.0	1.5	2.0	1.5	1.5
2	3	2.5	2.0	2.0	1.0
3	3.0	1.0	1.0	3.0	3.5
4	1.5	2.0	2.0	2.0	1.0

Figura 3 Ejemplo de matriz de precios de una localidad

	0	**1**	**2**	**3**	**4**
0	X	1	4	9	
1	2	3	6		
2	5	7			
3	8				
4					

Figura 4 Ejemplo de solicitud de reserva inmediata para zona (0, 0) y radio 3, teniendo en cuenta la matriz de precios de la Figura 3.

Como se puede observar en el ejemplo de la Figura 4, entre las zonas con distancia 1 a la zona (0, 0), se reservaría primero una plaza en la zona (0, 1) antes que en la zona (1, 0), ya que aunque el sentido antihorario pasaría primero por la (1, 0), la zona (0, 1) es más barata. Por otro lado, si fuera necesario considerar las zonas a distancia 2 de (0, 0), se intentaría primero reservar en la zona (1, 1) antes que en la zona (0, 2), porque en este caso ambas zonas tendrían el mismo precio y por tanto se tendría en cuenta el sentido antihorario.

2. Consideraciones de entrega

El código debe compilar en la **versión 1.11 de Java**.

Cada fichero debe contener una anotación (justo a continuación de los imports) con los nombres de los alumnos del grupo según la plantilla suministrada en autores.txt:

```
@Programacion2 (
	nombreAutor1 = "nombre",
	apellidoAutor1 = "apellido1 apellido2",
	emailUPMAutor1 = "usr@alumnos.upm.es",
	nombreAutor2 = "",
	apellidoAutor2 = "",
	emailUPMAutor2 = ""
)
```

Se van a distinguir dos tipos de requisitos a la hora de realizar el proyecto, obligatorios y opcionales.

Requisitos obligatorios

- Constructores de todas las clases excepto Reserva Inmediata
- Solicitar reserva anticipada
- Ocupación de una plaza

Requisitos opcionales

- Desocupación de una plaza
- Anulación de reserva
- Revisión de la lista de espera
- Solicitar reserva inmediata

Al entregar el proyecto, el sistema de entrega realizará una serie de pruebas automáticas. Si estas pruebas determinan que al menos las operaciones asociadas a los requisitos obligatorios funcionan correctamente, se aceptará la entrega. En caso contrario, no se admitirá la entrega. Por tanto, antes de realizar la entrega, el alumno debe asegurarse de que su ejercicio se puede compilar y supera todas las pruebas obligatorias (véanse los JUnitTests en el apartado 6).

Si solo funcionan correctamente las operaciones asociadas a los requisitos obligatorios, la máxima nota a la que se podrá aspirar es un 6 sobre 10.

3. Arquitectura de la aplicación

El ejercicio que realizará el alumno se incluirá en un proyecto con los siguientes paquetes:

- *controladores*: incluye la clase ControladorReservas que se encarga, en colaboración con otras clases del programa, de las operaciones principales: reserva de plaza, anulación de reserva, ocupar plaza con vehículo, desocupar plaza y obtención de las reservas atendidas de la lista de espera tras la anulación de una reserva o la desocupación de una plaza. Además, incluye en un subpaquete las excepciones que se pueden lanzar desde la clase ControladorReservas.
- *modelo.gestoresplazas*: incluye las clases que se encargan de gestionar las plazas disponibles para aparcar (huecos) y las listas de espera de solicitudes en las distintas zonas de la localidad. Para ello, llevan un registro de la disponibilidad de las plazas en cada zona a lo largo del tiempo.
- *modelo.gestoresplazas.huecos*: incluye las clases que permiten representar el estado de disponibilidad de cada plaza a lo largo del tiempo y obtener el mejor hueco disponible para una solicitud de reserva en una zona.
- *modelo.reservas*: incluye las clases que permiten representar una reserva, así como un repositorio de todas las reservas realizadas durante la ejecución del programa.
- *modelo.reservas.solicitudesreservas*: incluye las clases que permiten representar la jerarquía de solicitudes de reserva.
- *modelo.vehiculos*: incluya una clase para modelar un vehículo dentro de la aplicación.

En la Figura 5 se detallan las clases de la aplicación. Las clases que se deben definir partiendo de cero son las que aparecen con fondo amarillo. Las clases que se proporcionan parcialmente implementadas, y que el alumno tendrá que ampliar aparecen con fondo azul claro. El resto de las clases, que aparecen con fondo naranja, se proporcionan completas, por lo tanto, no tendrán que ser modificadas por el alumno.

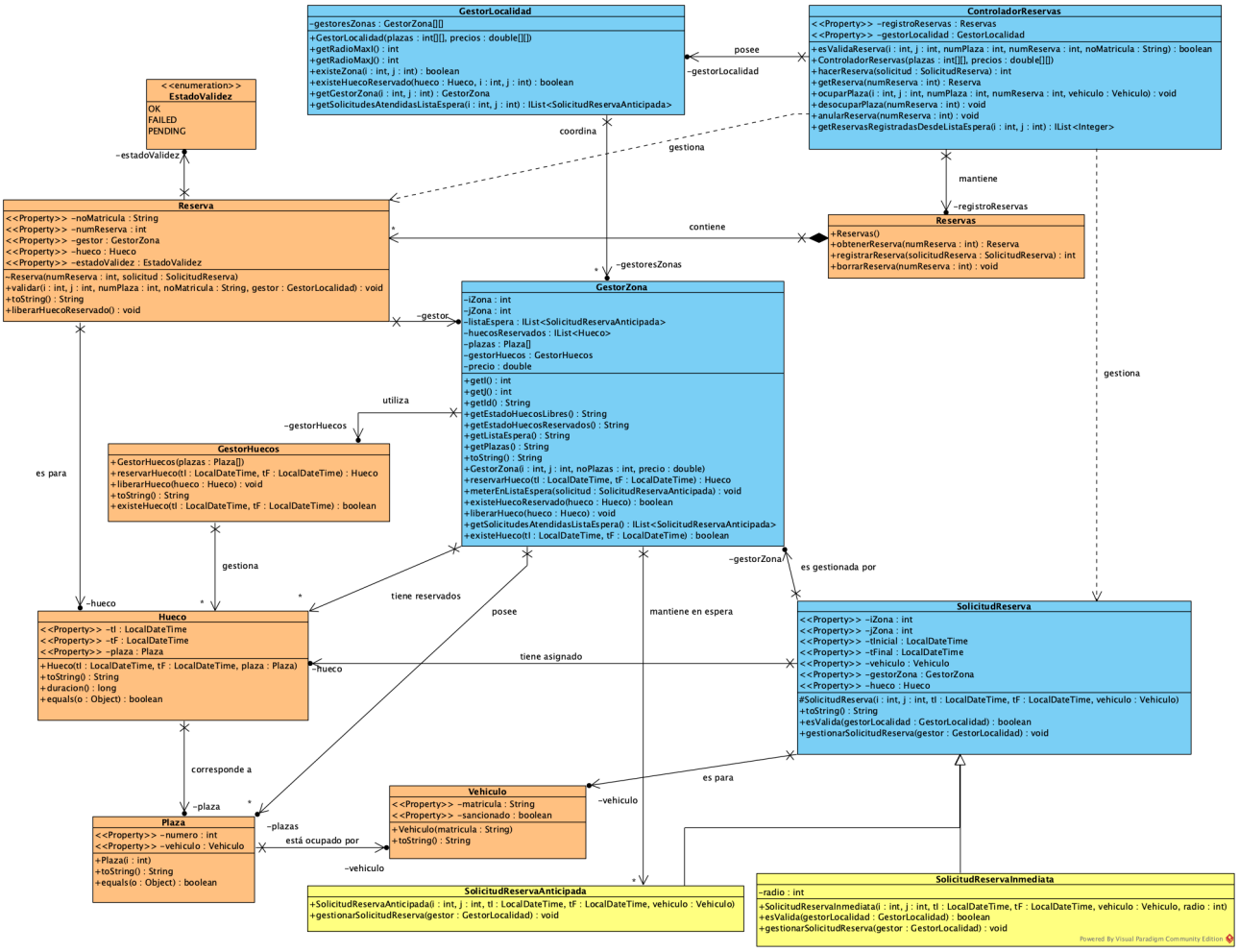

Figura 5 Diagrama de clases del proyecto

NOTA: los atributos marcados como <<Property>> en la Figura 3 son atributos para los cuales existen getters y en algunos casos setters. **En aras de la claridad, estos getters y/o setters se han omitido.**

4. Diseño detallado de las clases

En este apartado se va a explicar el diseño de las clases que el alumno tiene que ampliar o definir partiendo de cero. Se va a omitir la descripción de los métodos que ya vienen implementados en el código de apoyo. El alumno solo tendrá que comprobar las precondiciones de los métodos, y lanzar sus excepciones correspondientes, en las situaciones que se expliquen en este enunciado.

4.1. Paquete controladores

En este paquete se encuentra la clase ControladoresReservas, que tiene que ampliar el alumno. A continuación, se van a explicar los métodos y atributos de esta clase.

Atributos de la clase ControladoresReservas:

registroReservas: contiene una referencia al registro en el que se guardan las reservas realizadas durante la ejecución del programa.

gestorLocalidad: contiene una referencia al gestor de la localidad.

Métodos obligatorios de la clase ControladoresReservas:

ControladoresReservas: constructor de la clase ControladoresReservas que recibe como argumento una matriz de enteros en la que se especifican el número de plazas disponibles en cada zona de la localidad, y la matriz de precios, y se encarga de inicializar los atributos del objeto. Para ello, debe crear el registro de reservas y el gestor de localidad para las matrices dadas.

hacerReserva: gestiona la solicitud de reserva dada. Si la solicitud de reserva dada no es válida, se lanza la excepción *SolicitudReservaInvalida*.

Para gestionar la solicitud, utiliza el método que corresponda de la jerarquía de solicitudes de reservas. Posteriormente, se comprueba si se ha podido asignar un hueco a la solicitud (el hueco es distinto de null), y si es así se registra la reserva utilizando el método correspondiente de la clase Reservas. En caso contrario, se devuelve -1.

getReserva: devuelve la reserva asociada al número dado.

ocuparPlaza: ocupa la plaza asociada a la reserva dada (con numReserva) con el vehículo dado. Si el número de reserva dado no tiene asociada una reserva válida, se lanza la excepción *ReservaInvalida*. Si la plaza ya está ocupada, se lanza la excepción *PlazaOcupada*.

Métodos opcionales de la clase ControladoresReservas:

desocuparPlaza: desocupa la plaza asociada a la reserva dada (con numReserva) asignando un null como vehículo de la plaza. Además, se libera el hueco asociado a la reserva.

anularReserva: anula la reserva (con numReserva) liberando el hueco asociado a la reserva y borrando la reserva del registro de reservas.

getReservasRegistradasDesdeListaEspera: supondremos que este método se ejecutará después de haber desocupado una plaza o anulado una reserva en la zona (i, j) para obtener la lista de reservas (lista de números de reservas) que han podido ser realizadas tras liberar el hueco, si existen. Para ello, primero se obtendrán las solicitudes de reserva que han podido ser atendidas tras liberar el hueco en la zona (i, j) y luego se registrará una reserva para cada solicitud atendida. La lista de reservas obtenida debe estar ordenada en el mismo orden en el que aparecían sus correspondientes solicitudes en la lista de espera.

4.2. Paquete modelo.gestoresplazas

En este paquete se encuentran las clases GestorLocalidad y GestorZona, que tiene que ampliar el alumno. A continuación, se van a explicar los métodos y atributos de estas dos clases.

Atributos de la clase GestorLocalidad:

gestoresZonas: es un vector que contiene los gestores de zona asociados a esta localidad.

Métodos obligatorios de la clase GestorLocalidad:

GestorLocalidad: constructor de la clase GestorLocalidad que recibe como argumento una matriz de enteros en la que se especifican el número de plazas disponibles en cada zona (i, j) de la localidad, junto con los precios de las zonas, y se encarga de inicializar los atributos del

objeto. Para ello, debe crear los gestores de zona para esta localidad con las plazas y precios indicadas para cada uno (posición i, j) en el parámetro de entrada.

existeZona: indica si las coordenadas de una zona dadas están en el rango válido de esta localidad.

existeHuecoReservado: indica si el hueco dado está reservado en la zona (i, j) dada.

getRadioMaxI: devuelve N-1 siendo N el máximo número de filas de la matriz de zonas.

getRadioMaxJ: devuelve M-1 siendo M el máximo número de columnas de la matriz de zonas.

getGestorZona: devuelve el gestor de la zona (i, j) dada.

Métodos opcionales de la clase GestorLocalidad:

getSolicitudesAtendidasListaEspera: devuelve las solicitudes de reserva que pueden ser atendidas en la zona (i, j). La lista de solicitudes obtenida debe estar ordenada en el mismo orden en el que aparecían en la lista de espera.

Atributos de la clase GestorZona:

iZona, jZona: coordenadas de la zona en la localidad.

plazas: vector con las plazas de la zona.

precio: precio correspondiente a las plazas de la zona.

listaEspera: lista de espera de solicitudes anticipadas ordenada por orden de solicitud de más antigua a más reciente.

gestorHuecos: gestor de huecos asociado a esta zona.

huecosReservados: lista de huecos reservados en esta zona ordenada por orden de reserva de más antigua a más reciente.

Métodos obligatorios de la clase GestorZona:

GestorZona: constructor de la clase GestorZona que recibe como argumento las coordenadas de la zona, el número de plazas disponibles, y el precio de la zona, y se encarga de inicializar todos los atributos del objeto. Para ello, debe crear las plazas de esta zona, crear dos listas vacías para los atributos listaEspera y huecosReservados, y crear un gestor de huecos para las plazas de esta zona.

existeHuecoReservado: indica si el hueco dado está reservado en esta zona.

existeHueco: indica si existe un hueco libre en el intervalo temporal dado, utilizando la función correspondiente del gestor de huecos.

reservarHueco: intenta reservar un hueco en el intervalo temporal dado utilizando el gestor de huecos. Si existe, lo añade al final de la lista de huecos reservados y lo devuelve. En caso contrario, devuelve null.

meterEnListaEspera: añade la solicitud dada al final de la lista de espera.

Métodos opcionales de la clase GestorZona:

getSolicitudesAtendidasListaEspera: devuelve las solicitudes de reserva que pueden ser atendidas en esta zona y las elimina de la lista de espera. La lista de solicitudes obtenida debe estar ordenada en el mismo orden en el que aparecían en la lista de espera.

liberarHueco: extrae el hueco dado de la lista de huecos reservados y lo libera en el gestor de huecos.

4.3. Paquete modelo.reservas

En este paquete se encuentran las clases SolicitudReserva, que tiene que ampliar el alumno, y las clases SolicitudReservaAnticipada y SolicitudReservaInmediata, que se tienen que implementar desde cero. Cabe señalar que mientras que las dos primeras se consideran obligatorias, la tercera es optativa. A continuación, se van a explicar los métodos y atributos de estas clases.

Atributos de la clase SolicitudReserva:

iZona, jZona: coordenadas de la zona en la que se solicita la plaza.

tInicial, tFinal: intervalo para el que se solicita la plaza.

vehiculo: vehículo para que el que se solicita la plaza.

gestorZona: gestor asociado a la zona en la que se asigna la plaza. En el caso de la solicitud inmediata, no tiene por qué coincidir con el gestor de la zona (i, j), ya que podría ser otro gestor de zona dentro del radio máximo permitido por la solicitud.

hueco: hueco que se asigna tras completar la reserva. Hasta entonces tiene valor null.

Métodos obligatorios de la clase SolicitudReserva:

esValida: indica si existe la zona (i, j) en el gestor de localidad dado, se cumple tI < tF y el vehículo no está sancionado.

gestionarSolicitudReserva: define las operaciones comunes que se deben realizar tanto para gestionar una solicitud anticipada como una inmediata. Por tanto, debe inicializar el atributo de gestorZona con el gestor de la zona (i, j) e intentar reservar un hueco en dicho gestor.

Métodos obligatorios de la clase SolicitudReservaAnticipada:

SolicitudReservaAnticipada: es el constructor y solo se encarga de pasar todos sus parámetros de entrada al constructor de la clase padre.

gestionarSolicitudReserva: sobrescribe el método heredado extendiendo su definición de manera que se comprueba si después de ejecutarse el método del padre, se ha asignado un hueco a la

solicitud, o por el contrario el atributo sigue valiendo null, en cuyo caso, se añade la solicitud a la lista de espera de la zona (i, j).

Atributos de la clase SolicitudReservaInmediata (opcional):

radio: establece el radio máximo en el que se puede reservar una plaza.

Métodos de la clase SolicitudReservaInmediata (opcional):

SolicitudReservaInmediata: es el constructor y se encarga de pasar los parámetros de entrada al constructor de la clase padre e inicializar el atributo radio.

esValida: sobrescribe el método heredado extendiendo su definición de manera que comprueba también que radio>0 y que existe alguna zona en la localidad en el límite del radio. Es decir, si el radio especificado es tal que no existe ninguna zona con el radio máximo, el método debe devolver *false*.

Por ejemplo, en el caso de la Figura 1, solo se podrían admitir solicitudes con un radio ≤ 4. En cambio, si la solicitud se realiza para la zona (1, 2), entonces se podría admitir un radio ≤ 5, porque al menos las zonas (4, 4) y (4, 0) estarín justo en el límite de esa radio. Con radio=6, ya no habría ninguna zona en la localidad.

gestionarSolicitudReserva: sobrescribe el método heredado extendiendo su definición de manera que se comprueba si después de ejecutarse el método del padre, se ha asignado un hueco a la solicitud, o por el contrario el atributo sigue valiendo null, en cuyo caso, se busca un hueco en un gestor de zona dentro del radio máximo establecido para esta solicitud. Los gestores de zona son considerados en el orden en el que se explica en el apartado 1. Es decir, se considera primero la distancia (de Manhattan) a la plaza inicial, después la matriz de precios, y por último el sentido antihorario de recorrido.

5. Orden recomendado para implementar los métodos

El orden que se recomienda seguir para implementar los métodos es el siguiente:

1 En la clase GestorZona todos los métodos obligatorios excepto meterEnListaEspera(): Se debe comentar provisionalmente el atributo listaEspera, ya que todavía no está definida la clase SolicitudReservaAnticipada. Para probar esta clase se podrá utilizar el JUnitTest TestGestorZonaOblig teniendo en cuenta que todavía no se podrá superar el testMeterListaEspera.
2 En la clase GestorLocalidad todos los métodos obligatorios: Para probar esta clase se podrá utilizar el JUnitTest TestGestorLocalidadOblig.
3 En las clases SolicitudReserva y SolicitudReservaAnticipada todos los métodos y el método meterEnListaEspera() de la clase GestorZona: Se podrán utilizar los JUnitTests TestSolicitudReservaAnticipadaOblig, TestGestorZonaOblig y TestGestorLocalidadOblig.

4 En la clase ControladorReservas se recomienda implementar los métodos obligatorios en el siguiente orden:

4.a Constructor
4.b hacerReserva
4.c getReserva
4.d ocuparPlaza

Para probar estos métodos se debe utilizar el JUnitTest TestControladorReservasOblig.

En este punto, terminaría la implementación de la parte obligatoria del proyecto. En cuanto a la parte opcional, se recomienda implementar los métodos relacionados con las siguientes operaciones en este orden:

1 Desocupar plaza.
2 Anular reserva.
3 Obtener las reservas registradas resultado de las solicitudes atendidas de la lista de espera.
4 Solicitudes inmediatas.

Para las solicitudes inmediatas, se proporcionan métodos de test que comprueban que el recorrido se realiza de forma correcta en varios casos, incluyendo el caso de que todos los precios de las zonas de la localidad sean iguales.

6. Código de apoyo

Los alumnos deben utilizar el código de apoyo que se encuentra en el archivo suministrado **SmartParkingAlumnos.zip**. Este archivo contiene:

- autores.txt: plantilla de anotación para identificar el ejercicio y los alumnos del grupo. Se insertará al comienzo de cada fichero.
- src/test: contiene los JUnitTest. Los ficheros con sufijo “Oblig” contienen los JUnitTests de los requisitos obligatorios, en tanto que los ficheros con sufijo “Opcional” contiene los de los requisitos opcionales.
- doc.zip: contiene la documentación javadoc de las clases con fondo naranja en la Figura 3.
- libSmartParking.jar: contiene los binarios de las clases con fondo naranja en la Figura 3.

Las siguientes carpetas src/X contienen las clases del paquete X correspondiente:

- src/controladores
- src/modelo/gestoresplazas
- src/modelo/reservas/solicitudesreservas

7. Errores a evitar

Algunos errores que provocarán una calificación baja, de acuerdo con el baremo de calificación establecido, son (entre otros):

- Se declaran atributos públicos.
- Se realizan operaciones de entrada/salida en alguna de las clases implementadas por el alumno.

Otros errores que los profesores penalizarán también son los siguientes (entre otros):

- Atributos friendly o protected
- Métodos auxiliares públicos
- Código duplicado
- Código innecesario o inalcanzable
- Llamadas a métodos innecesarias o redundantes
- Documentación deficiente (faltan comentarios significativos)
- Atributos innecesarios (de clase o de instancia)
- Identificadores no significativos
- No sigue el convenio de nombres de Oracle
- Código mal indentado
- Uso innecesario de if para asignar o devolver valores booleanos
- Bucles con ruptura mediante return, break o continue
- Recorridos completos de una estructura (array, arraylist, etc.) cuando basta recorrer una parte

8. Pautas de programación a tener en cuenta

Se deben seguir las pautas explicadas en el Documento de buenas prácticas que se encuentra publicado en el aula virtual de la asignatura (plataforma Moodle).