Asignatura: Programación II – Curso 2018/19 – 2er Semestre

Dpto. LSIIS. Unidad de Programación

Proyecto NowMeal

***Objetivo*:** El objetivo de este proyecto es la familiarización del alumno con la programación orientada a objetos en Java, incluyendo manejo de arrays de objetos, manejo de las clases genéricas list.ArrayList<E> y queues.NaiveQueue<E> de la asignatura, organización en paquetes (modularidad), ejecución de pruebas unitarias y herencia de clases.

***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 6. 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 **NowMealAlumnos.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: http://triqui2.fi.upm.es/entrega/. 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 día 30 de abril 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

NowMeal es una aplicación para la venta de comida a domicilio. Gracias a esta aplicación, los clientes pueden pedir comida a domicilio por medio de un dispositivo móvil. Esta aplicación ofrece a los clientes una lista de restaurantes, los cuales, a su vez, ofrecen distintos tipos de comidas. Al pedir la comida, el cliente debe indicar el restaurante y los platos de comida que desea recibir de ese restaurante. Cada vez que llega un pedido, NowMeal le debe asignar un transporte. Este transporte debe ir primero a recoger la comida al restaurante y luego debe llevar esta comida al cliente.

Existen dos tipos de transportes, motos y furgonetas. A su vez, esta aplicación va a distinguir entre dos tipos de furgonetas, propias y subcontratadas. Las furgonetas propias son propiedad de la propia empresa y básicamente se diferencian de las subcontratadas en la manera de calcular el coste que supone para la empresa que la furgoneta vaya de una posición a otra de una localidad. En el apartado 3.3. Paquete gestionpedidos.transportes se explicarán con más detalle las características de cada tipo de transporte.

Si la cantidad de comida que hay que transportar para atender un pedido está por debajo de un umbral preestablecido, se asignará una moto al pedido, y en caso contrario, se le asignará una furgoneta. De entre las motos/furgonetas que trabajan para NowMeal y que no están ya asignadas a un pedido, NowMeal debe elegir aquella moto/furgoneta que minimice el coste del pedido para la empresa.

El coste de un pedido se calcula sumando el importe de los platos de comida a los costes del transporte. A su vez, los costes del transporte se calculan sumando el coste que supone para la empresa que el transporte vaya desde su posición actual hasta el restaurante y luego desde el restaurante hasta el domicilio del cliente.

Una vez que el transporte ha entregado el pedido al cliente, se le notifica a la aplicación NowMeal. En ese momento, este transporte pasa a estar disponible de nuevo.

Los objetos moto, furgoneta, cliente, plato de comida y restaurante poseen un atributo *codigo* que los identifica. No puede haber dos objetos, del mismo tipo o diferente, con el mismo código.

Para poder calcular las distancias entre distintas posiciones de una localidad, las posiciones de las motos, furgonetas, clientes y restaurantes se encuentran representadas en un mapa. Este mapa está dividido en cuatro localidades o zonas de reparto contiguas para las cuales los pedidos se van a gestionar de forma independiente. Es decir, un cliente solo podrá realizar pedidos a restaurantes de su misma zona y todos los pedidos de una zona solo podrán ser atendidos por transportes que se encuentren en esa misma zona.

Cada zona de reparto tiene asignado un entero y está ubicada en el mapa de la siguiente manera (véase la figura 1):

0	2
1	3

Figura 1 Mapa de zonas

Las coordenadas X e Y de cada zona se encuentran en los siguientes intervalos cerrados:

- Zona 0: X ∈ [0, maxCoordX/2] Y ∈ [0, maxCoordY/2]
- Zona 1: X ∈ [0, maxCoordX/2] Y ∈ [maxCoordY/2 + 1, maxCoordY]
- Zona 2: X ∈ [maxCoordX/2 + 1, maxCoordX] Y ∈ [0, maxCoordY/2]
- Zona 3: X ∈ [maxCoordX/2 + 1, maxCoordX] Y ∈ [maxCoordY/2 + 1, maxCoordY]

A continuación, se muestra un ejemplo de cómo podría ser el submapa de la zona de reparto 0 para un mapa de dimensiones 10x10 (véase la figura 2).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	C1									
1								C3		
2						F1				
3										
4				C2					M1	
5						R2				
6										
7		R1				F2	F3	M2		
8										
9										C4

Figura 2 Ejemplo de submapa para la zona de reparto 0

En este submapa se representan las posiciones de 4 clientes (C1, C2, C3 y C4), 2 restaurantes (R1 y R2), 3 furgonetas (F1, F2 y F3) y 2 motos (M1 y M2). Por ejemplo, C2 se encuentra en la posición (3, 4).

2. Arquitectura de la aplicación

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

- *gestionpedidos*: incluye las clases que se encargan de la asignación de transportes a los pedidos.

- *gestionpedidos.mapa*: incluye las clases que representan el mapa en el que se ubican los clientes, los restaurantes y los transportes.
- *gestionpedidos.pedido*: incluye las clases que describen la información asociada a un pedido, es decir, el pedido propiamente dicho, el cliente que solicita el pedido, los platos de comida que pide y el restaurante al que pide estos platos.
- *gestionpedidos.transportes*: incluye las clases que definen los distintos tipos de transportes.
- *simulador:* incluye la clase Simulador, que permite mostrar en la consola el resultado de ejecutar las operaciones de asignación de un transporte a un pedido y de notificación de entrega de un pedido a un cliente. Para ello, cada vez que se ejecuta una de estas operaciones, se invocará a la operación correspondiente de la clase GestionPedido, y luego se mostrará en la pantalla el submapa de la zona de reparto en la que se ha realizado el pedido, así como los transportes disponibles y los pedidos pendientes de asignación en esa misma zona de reparto. Si se asigna un transporte a un pedido, el simulador mostrará un submapa en el que el transporte asignado al pedido aparecerá en la posición del restaurante para el que se solicitó el pedido. Posteriormente, cuando el transporte entregue el pedido al cliente, el simulador mostrará un submapa en el que este transporte aparecerá en la posición del cliente.
- *test.simulaciones*: incluye dos programas de prueba, que invocan las operaciones del simulador para recrear una secuencia de solicitudes de pedidos y notificaciones de entregas. El programa *PruebaSoloMotos* recrea un escenario de prueba en el que solo hay disponibles motos, mientras que el programa *PruebaFurgosMotos* recrea un escenario de prueba con motos y furgonetas de ambos tipos.

En las figuras 3 y 4 se muestran todas las clases que forman parte 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 anaranjado**. **El resto de las clases, que aparecen con fondo azul claro, se proporcionan completas, por lo tanto, no tendrán que ser modificadas por el alumno**.

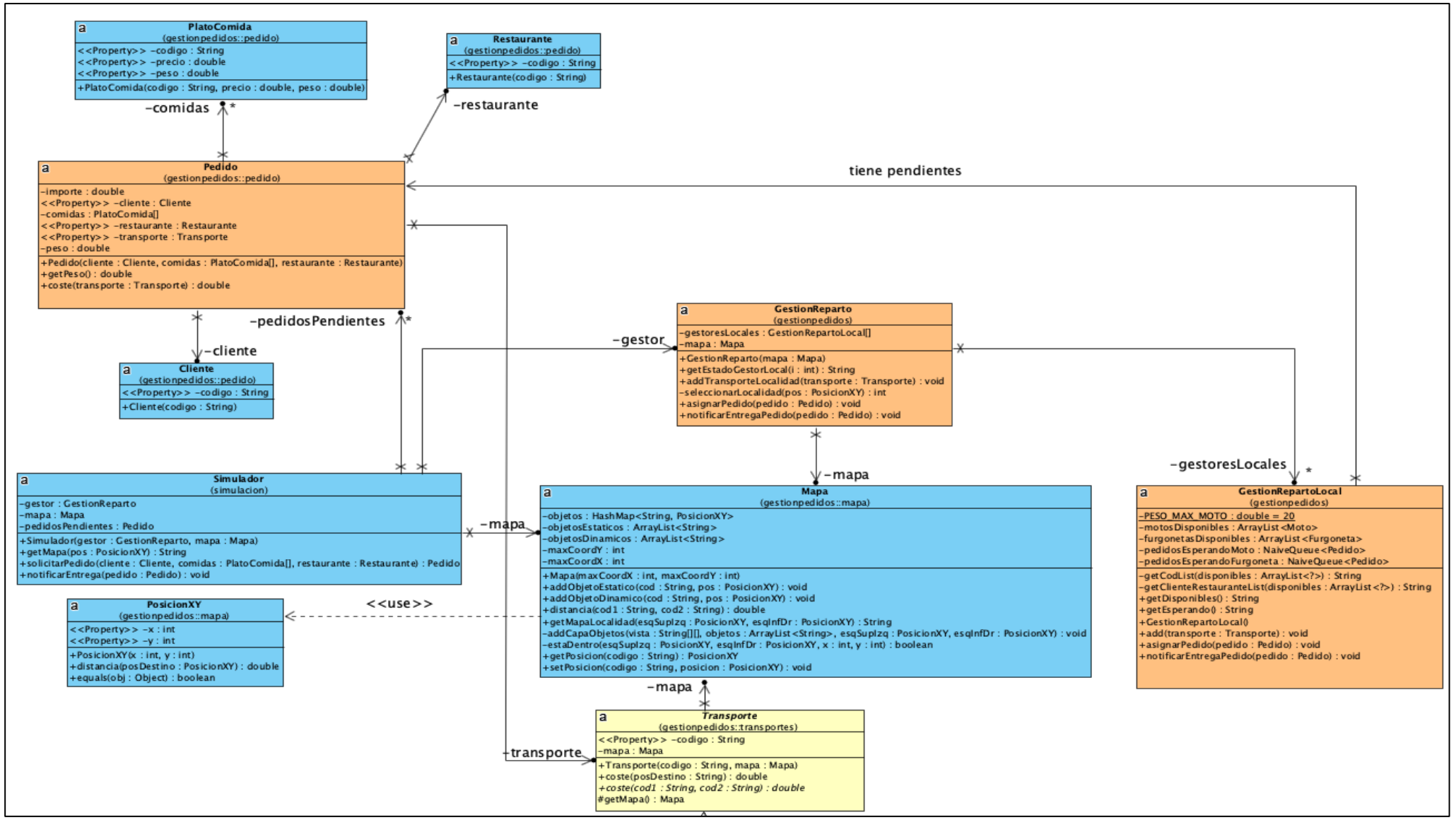

Figura 3. Diagrama de clases UML (1/2)

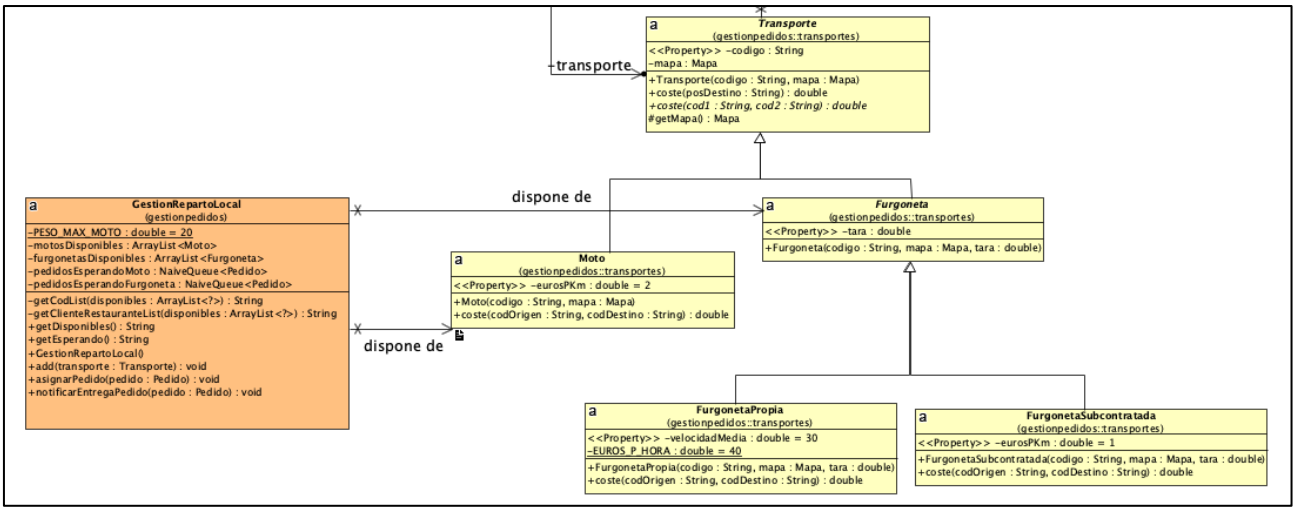

Figura 4. Diagrama de clases UML (2/2)

NOTA: los atributos marcados como <<Property>> 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.

3. 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.

3.1. Paquete gestionpedidos

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

Atributos de la clase GestionReparto:

gestoresLocales: es un vector (array) que contiene los gestores de reparto local para las 4 zonas consideradas.

mapa: contiene una referencia al mapa de las cuatro zonas.

Métodos de la clase GestionReparto:

GestionReparto: constructor de la clase GestionReparto que recibe como argumento el mapa, y se encarga de inicializar los atributos del objeto. Para ello, debe crear los 4 gestores de reparto locales.

addTransporteLocalidad: añade el transporte dado al gestor de reparto local que le corresponde por su ubicación en el mapa.

asignarPedido: asigna el pedido dado al gestor de reparto local que le corresponde por su ubicación en el mapa.

notificarEntregaPedido: notifica la entrega del pedido al gestor de reparto local que le corresponde por su ubicación en el mapa.

Atributos de la clase GestionRepartoLocal:

motosDisponibles, furgonetasDisponibles: listas de motos y furgonetas disponibles para transportar pedidos.

pedidosEsperandoMoto, pedidosEsperandoFurgoneta: colas de pedidos pendientes de ser asignados a una moto o a una furgoneta. Las colas están ordenadas por orden de llegada, por tanto, el primer pedido de cada cola es el que lleva más tiempo esperando.

Métodos de la clase GestionRepartoLocal:

add: añade el transporte dado al final de la lista de motos disponibles o a la lista de furgonetas disponibles dependiendo de si el transporte dado es una moto o una furgoneta.

asignarPedido: asigna el pedido dado a un transporte disponible, si existe. Para ello, se distinguen dos casos:

1. Si el peso del pedido es menor o igual que el peso máximo que puede transportar una moto, se le asigna la moto de la lista de disponibles que minimice el coste del pedido. Si no existe ninguna moto disponible, se guarda el pedido al final de la lista de pedidos esperando una moto.
2. En caso contrario, hay que utilizar una furgoneta en lugar de una moto. Por tanto, se le asigna la furgoneta de la lista de disponibles que minimice el coste del pedido. Supondremos que ningún pedido sobrepasa la capacidad máxima de una furgoneta. Si no existe ninguna furgoneta disponible, se guarda el pedido al final de la lista de pedidos esperando una furgoneta.

notificarEntregaPedido: notifica la entrega del pedido dado. Eso implica que el transporte asignado al pedido dado pasa a estar disponible. Por tanto, si hay pedidos esperando por una moto y se acaba de quedar libre una, se asigna al pedido que lleve más tiempo esperando. Si el transporte que se acaba de quedar libre es una furgoneta, se actúa de forma análoga. Si no hay pedidos esperando por un transporte del tipo que se acaba de quedar libre, dicho transporte se guarda al final de la lista de transportes disponibles del mismo tipo.

3.2. Paquete gestionpedidos.pedido

En este paquete se encuentran las clases Pedido, Cliente, PlatoComida y Restaurante, de las cuales, el alumno solo tiene que ampliar la clase Pedido.

Atributos de la clase Pedido:

cliente: es el cliente que ha realizado el pedido.

comidas: es un vector (array) de objetos de tipo PlatoComida que incluye los platos que ha pedido el cliente.

restaurante: es el restaurante que prepara los platos de comida del pedido.

importe: es el precio total de todos los platos de comida del pedido.

transporte: es el transporte asignado al pedido. Hasta que se le asigne un transporte, tendrá valor null.

peso: es el peso total de todos los platos de comida del pedido.

Métodos de la clase Pedido:

Pedido: constructor de la clase Pedido. Inicializa los atributos del objeto con los argumentos recibidos como entrada. Calcula el importe y el peso del pedido y se los asigna a sus respectivos atributos.

getPeso: devuelve el peso total de los platos de comida.

coste: devuelve el coste del pedido. Dicho coste se calcula sumando el importe, el coste que supone para el transporte dado ir desde su ubicación actual hasta el restaurante y el coste que supone para el transporte dado ir desde el restaurante hasta el domicilio del cliente.

3.3. Paquete gestionpedidos.transportes

En este paquete se encuentran las clases que definen la jerarquía de transportes: Transporte, Furgoneta, Moto, FurgonetaPropia y FurgonetaSubcontratada. El alumno tendrá que implementarlas partiendo de cero siguiendo las especificaciones de este enunciado.

Atributos de la clase Transporte:

codigo: es el código del transporte.

mapa: es el mapa en el que se encuentra ubicado el transporte.

Métodos de la clase Transporte:

Transporte: constructor de la clase Transporte. Inicializa los atributos del objeto con los argumentos recibidos como entrada.

coste(codPosDestino): devuelve el coste en euros que supone para el transporte ir desde su ubicación actual hasta la ubicación del objeto con código codPosDestino.

coste(codPosOrigen, codPosDestino): devuelve el coste en euros que supone para el transporte ir desde la ubicación del objeto con código codPosOrigen hasta la ubicación del objeto con código codPosDestino. Se declara como método abstracto que luego se sobrescribe en las clases Moto, FurgonetaPropia y FurgonetaSubcontratada.

Atributos de la clase Moto:

eurosPKm: es la tarifa que se ha negociado para esta moto expresada en euros/Km. Su valor por defecto es 2.

Métodos de la clase Moto:

Moto: constructor de la clase. Inicializa los atributos del objeto con los argumentos recibidos como entrada.

coste(codPosOrigen, codPosDestino): devuelve el coste en euros que supone para el transporte ir desde la ubicación del objeto con código codPosOrigen hasta la ubicación del objeto con código codPosDestino. Este coste se calcula con la fórmula:

*distancia(codPosOrigen, codPosDestino)*eurosPKm.*

Atributos de la clase Furgoneta:

tara: es la tara de la furgoneta expresada en kg.

Métodos de la clase Furgoneta:

Furgoneta: constructor de la clase. Inicializa los atributos del objeto con los argumentos recibidos como entrada.

Atributos de la clase FurgonetaPropia:

velocidadMedia: es la velocidad media registrada hasta la fecha para esta furgoneta expresada en Km/h.

EUROS_P_HORA: es una constante que representa los euros por hora que cobra una furgoneta.

Métodos de la clase FurgonetaPropia:

FurgonetaPropia: constructor de la clase. Inicializa los atributos del objeto con los argumentos recibidos como entrada.

coste(codPosOrigen, codPosDestino): devuelve el coste en euros que supone para el transporte ir desde la ubicación del objeto con código codPosOrigen hasta la ubicación del objeto con código codPosDestino. Si la tara de la furgoneta **es menor que 500 Kg**, el coste se calcula así *distancia(codPosOrigen, codPosDestino)*EUROS_P_HORA/velocidadMedia*. **En caso contrari**o, el coste se obtiene con la fórmula:

*distancia(codPosOrigen, codPosDestino)*EUROS_P_HORA/velocidadMedia*1.10.*

Atributos de la clase FurgonetaSubcontratada:

eurosPKm: es la tarifa que se ha negociado para esta furgoneta expresada en euros/Km. Su valor por defecto es 1.

Métodos de la clase FurgonetaSubcontratada:

FurgonetaSubcontratada: constructor de la clase. Inicializa los atributos del objeto con los argumentos recibidos como entrada.

coste(codPosOrigen, codPosDestino): devuelve el coste en euros que supone para el transporte ir desde la ubicación del objeto con código codPosOrigen hasta la ubicación del objeto con código codPosDestino. Si la tara de la furgoneta es menor que 1000 Kg, el coste se calcula así *distancia(codPosOrigen, codPosDestino)*eurosPKm*. En caso contrario, el coste se obtiene con la fórmula:

*distancia(codPosOrigen, codPosDestino)*eurosPKm*1.10.*

4. Código de apoyo

Los alumnos deben utilizar el código de apoyo que se encuentra en el archivo suministrado **NowMealAlumnos.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/simulaciones: contiene los ficheros PruebaSoloMotos.java y PruebaFurgosMotos.java.
- src/test: contiene los ficheros con los JUnitTest:
  - AtributosDeclaradosTest.java,
  - PedidoJUnitTest.java,
  - ObligGestionRepartoJUnitTest.java,
  - ObligGestionRepartoLocalJUnitTest.java y
  - MotoJUnitTest.java,
- doc: contiene la documentación javadoc de los JUnitTest, tanto de las pruebas obligatorias como de las pruebas opcionales.

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

- src/gestionpedidos
- src/gestionpedidos/mapa
- src/gestionpedidos/pedido
- src/gestionpedidos/transportes
- src/simulador

6. Consideraciones de entrega

El código debe compilar en la versión 1.8 del J2SE de Oracle. Se entregarán los 8 ficheros creados o modificados por el alumno: Pedido.java, GestionReparto.java, GestionRepartoLocal.java, Transporte.java, Moto.java, Furgoneta.java, FurgonetaPropia.java y FurgonetaSubcontratada.java. **Las clases incluidas en estos ficheros deben contener obligatoriamente todos los atributos y las cabeceras de todos los métodos descritos en el apartado 3 y deben pertenecer a los paquetes especificados en dicho apartado**. Si no se desea entregar algún método, este método deberá tener un cuerpo que asegure al menos la correcta compilación del programa principal PruebaFurgosMotos.java. Eso se puede conseguir simplemente utilizando cuerpos vacíos o cuerpos con una única sentencia *return*.

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 = ""
    )
```

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

Requisitos obligatorios

- Todos los elementos se encuentran en la zona o submapa 0.
- En el paquete transporte solo deben estar implementados los métodos de la clase Transporte y la clase Moto.
- Solo se van a considerar pedidos que puedan ser transportados por motos, por tanto los pedidos solo se van a asignar a motos y solo se van a notificar entregas realizadas por motos.

Los JUnitsTests proporcionados dentro del zip solo comprueban los requisitos obligatorios, pero en la carpeta doc del zip se puede encontrar también la documentación de los JUnitTests correspondientes a las pruebas opcionales.

Requisitos opcionales

- Los elementos se pueden encontrar en cualquier zona o submapa.
- En el paquete transporte deben estar implementados los métodos de todas las clases especificadas en este enunciado.
- Se van a considerar pedidos que puedan ser transportados por motos o furgonetas.

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
- Getter o setter no previsto (accede a un atributo al que no debería acceder)
- 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).