Tecnología de la Programación

Implementación de la Práctica 4

(Basado en la práctica de Samir Genaim)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Introducción

- ✓ Las prácticas del segundo cuatrimestre permiten construir un simulador de tráfico que modela vehículos, carreteras y cruces.
- ✓ Esta práctica utiliza entrada/salida estándar (ficheros y/o consola) en su arquitectura. En las siguientes modificaremos la arquitectura añadiendo una interfaz gráfica.
- ✓ Nuevos objetivos: colecciones y genéricos.
- ✓ La práctica se divide en dos partes. No comenzar la segunda sin haber finalizado y probado la primera.
  - En la primera parte se implementan todos los componentes de la arquitectura con objetos simulados básicos en su modelo.
  - En la segunda parte se incorporan objetos avanzados al modelo utilizando herencia de objetos simulados básicos.

2. Análisis del Simulador de Tráfico

- ✓ El simulador contiene una colección de **objetos simulados** (vehículos y carreteras conectadas con cruces), una colección de **eventos** a ejecutar y un **contador de tiempo** que se incrementa en cada paso de simulación.
- ✓ Un paso de simulación consiste en realizar las siguientes operaciones:
  - Procesar los eventos (que pueden añadir y/o alterar los estados de los objetos simulados).
  - Avanzar el estado actual de los objetos simulados. Por ejemplo, los vehículos avanzaran en su carretera mientras esta finalice en un cruce con un semáforo en verde. Si el semáforo está en rojo, ningún vehículo en esa carretera podrá avanzar.
  - Mostrar el estado actual de los objetos simulados.

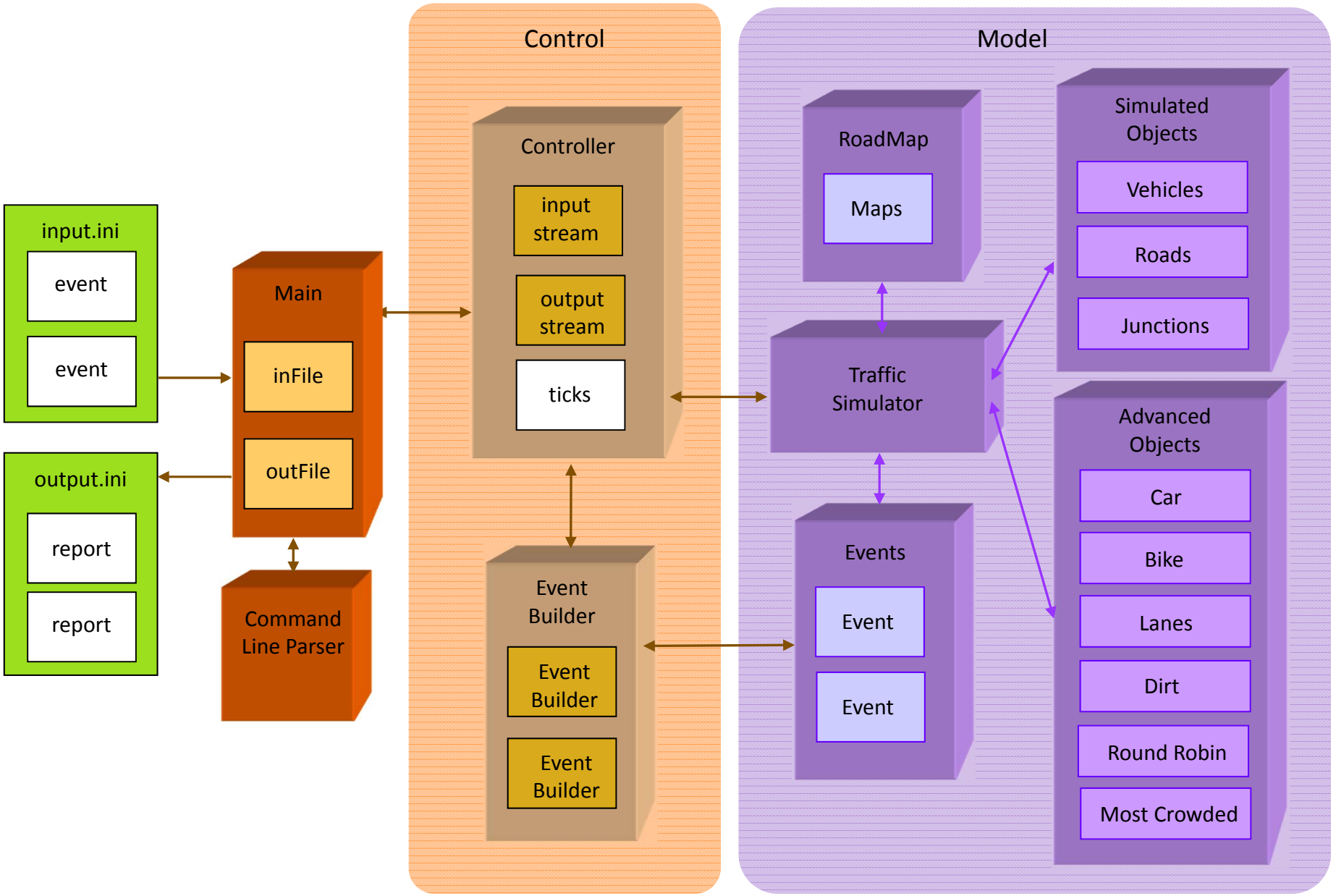
- ✓ Los eventos se leen de un fichero de texto antes de que la simulación comience. La simulación se ejecuta un número determinado de pasos (suministrados por el usuario). Al final de cada paso de simulación, el estado de todos los objetos simulados se puede mostrar, bien en fichero de texto, bien en la consola (según especifique también el usuario final).
- ✓ Los objetos simulados básicos son:
  - **Vehículos**. Viajan a través de carreteras y se pueden averiar ocasionalmente. Cada vehículo tiene un itinerario que consiste en todos los cruces que tiene que atravesar.
  - **Carreteras de dirección única**. Cada carretera tiene un cruce de origen y un cruce de destino.
  - **Cruces**. Estos conectan carreteras y organizan el tráfico a través de semáforos. Cada cruce tiene asociada una colección de carreteras entrantes. Desde un cruce solo se puede llegar directamente a otro cruce a través de una única carretera.

- ✓ Los objetos avanzados (de la segunda parte de la práctica) son tipos de vehículos, carreteras y cruces. En concreto, la práctica debe implementar:
  - **Coches**. Los coches son un tipo de vehículos que se averían aleatoriamente con cierta probabilidad.
  - **Bicis**. También son un tipo de vehículos. Se averían menos.
  - **Autopistas**. Son carreteras de varios carriles.
  - **Caminos**. Son carreteras por las que resulta mas difícil conducir
  - **Cruces circulares**. La duración de semáforos en verde es mayor.
  - **Cruces congestionados**. La siguiente carretera entrante en poner su semáforo en verde es aquella que tiene mas vehículos en su cola.

3. Diseño de Alto Nivel (Arquitectura)

- ✓ Para la resolución de la práctica se ha diseñado una arquitectura de alto nivel que incluye dos capas: **Model** y **Control**, además de otros componentes como **Main**.
- ✓ La capa Model contiene los componentes *TrafficSimulator*, *SimulatedObjets*, *AdvancedObjects*, *RoadMap* y *Events*.
- ✓ *TrafficSimulator* realiza los pasos de simulación procesando los eventos (*Events*) y avanzando los objetos simulados (*SimulatedObjects* y *AdavancedObjects*) con colecciones (*Maps*) que se mantienen en una estructura *RoadMap*.
- ✓ La capa Control incluye el controlador (*Controller*) que guarda información sobre los flujos de entrada y de salida (*intput stream* y *output stream*) e invoca a *TrafficSimulator* una vez establecido el número de pasos de simulación (*ticks*).

Arquitectura: Diseño de Alto Nivel



- ✓ La capa Control construye los eventos que tiene que ejecutar el simulador utilizando el componente *EventBuilder*. Este incluye un constructor por cada uno de los tipos de eventos que el simulador es capaz de procesar.
- ✓ El componente *Main* recoge los argumentos de ejecución desde la línea de comandos (el fichero de entrada *input.ini* y posible fichero de salida *output.ini*) y los parsea gracias a *CommandLineParser* (sumunistrado para la práctica).
- ✓ Cuando arranca la ejecución, *Main* parsea los argumentos e invoca a *Controller* para construir los objetos de eventos incluidos en el fichero *input.ini* de entrada.
- ✓ A cada paso de simulación de *TrafficSimulator* se generan reports de los objetos simulados. Estos informes son incluidos en el fichero de salida *output.ini* o en la consola.

4. Diseño Detallado de Clases (UML)

- ✓ El diseño detallado de la solución contiene la representación UML de todas y cada una de las clases necesarias para la realización de los componentes de alto nivel que hemos visto.
- ✓ Estas clases se pueden organizar en los siguientes paquetes caracterizando cada una de las capas de la arquitectura:

```
es.ucm.fdi.model;
es.ucm.fdi.control;
es.ucm.fdi.launcher; // para la clase Main
```

- ✓ A continuación mostramos el diseño UML de las clases suministrado como material de la práctica (ver al final de la presentación). Cada una de las clases contiene métodos necesarios para su implementación.

5. Modelo: Objetos Simulados Básicos

- ✓ Cada objeto simulado tiene un identificador único.
- ✓ La clase *SimulatedObject* tiene este atributo y todas las clases de objetos simulados derivan de ella.
- ✓ La clase ***Vehicle*** necesita atributos como los siguientes:

```
protected int maxSpeed; // máxima velocidad
protected int currSpeed; // velocidad actual
private Road road; // carretera por la que viaja
private int location; // localización en la carretera (desde 0)
protected List<Junction> itinerary; // lista de cruces
protected int kilometrage; // distancia recorrida por el vehículo
protected int faultyTime; // tiempo que resta en estado averiado
protected boolean atJunction; // true si ha entrado en cola de cruce
protected boolean arrived; // true cuando el coche llega a su destino
```

- ✓ Cuando un vehículo está averiado no puede circular y los vehículos detrás de él en una carretera van mas despacio.

- ✓ Un vehículo averiado se repara después de un determinado número de pasos de simulación.
- ✓ Fíjate en el diseño UML para ver los métodos de esta clase. Algunos de ellos son:

```
void advance(int time)  // con esta operación el vehículo avanza. Primero comprueba si esta averiado o no. Si esta averiado decrementa faultyTime en 1 y no se mueve. Si no está averiado (faultyTime es 0) entonces avanza su posicioón según su velocidad actual. La nueva localización es la localización anterior mas la velocidad actual. Si la nueva localización es igual o mayor a la longitud de la carretera entonces el vehículo entrara en la cola del correspondiente cruce. Los vehículos que esperan en la cola de un cruce no pueden avanzar y permanecen en esta cola hasta que el cruce determine que el vehículo debe moverse a la siguiente carretera, invocando el método moveToNextRoad().

void moveToNextRoad() // solicita al vehículo que se mueva a la siguiente carretera. Para ello el vehículo sale de su carretera actual y entra en la siguiente que esta en su itinerario, en la localización 0. Observa que la primera vez que se ejecuta este método no hay ninguna carretera saliente puesto que no ha entrado todavía en ninguna carretera. Además si el vehículo sale de la última carretera de su itinerario no entra en ninguna y arrived toma el valor true.
```

```
void makeFaulty(int n)  // pone el estado del vehículo en modo avería
durante n pasos. Si el vehículo ya está averiado acumula n al valor
del contador de tiempo de avería (faultyTime).

void setSpeed(int speed) // pone la velocidad actual del vehículo a
la que se indica por parámetro. Este método es utilizado por las
carreteras para fijar la velocidad de los vehículos. Si la velocidad
que se desea poner excede la velocidad máxima del vehículo se pone
igual a dicha velocidad máxima.

protected void fillReportDetails(IniSection is) // rellena los
valores de una sección ini con los atributos del vehículo para
mostrar un informe de su estado en uno de los pasos de simulación.
```

- ✓ En *SimulatedObject* debe implementarse un método que genere un String como informe (report) completo del estado del vehículo. Por ejemplo:

```
[vehicle_report]
id = v1
time = 5
speed = 20
kilometrage = 60
faulty = 0
location = (r1,30)
```

- ✓ En este informe *id* es el identificador del vehículo (v1), *time* es el tiempo de simulación en que se ha generado el informe, *speed* es la velocidad actual del vehículo, *kilometrage* es la distancia total recorrida por el vehiculo, *faulty* es 0 si el vehiculo no esta averiado o el tiempo que resta para ser reparado, y *location* es la localización del vehículo y está compuesta por el identificador de la carretera (r1) y su localización en ella (30).
- ✓ Recuerda que la velocidad de un vehículo es 0 si el vehiculo esta averiado, si esta esperando en la cola de un cruce o ya ha llegado a su destino. Y tal valor (0) deberás asignar a su atributo *currSpeed* en los métodos correspondientes.
- ✓ En la segunda parte de la práctica se implementan dos tipos de vehículos que heredan de *Vehicle*: *Car* y *Bike*.

- La clase ***Road*** debe incluir atributos como los siguientes:

```
protected int length; // longitud de la carretera
protected int maxSpeed; // limite máximo de velocidad
protected Junction srcJunc; // cruce origen de la carretera
protected Junction destJunc; // cruce destino de la carretera
… vehicles; // lista de vehículos que circulan por la carretera
ordenados por su localización (la localización 0 es el último
vehículo)
```

- Múltiples vehículos pueden estar en la mima localización de una carretera, en cuyo caso hay que preservar el orden de llegada a la lista para que el primero que llega sea el primero en abandonar la carretera.
- Algunos de los métodos diseñados en esta clase son:

```
void enter(Vehicle vehicle) // añade un vehículo a la lista de
vehículos de la carretera. Recuerda ordenar los vehículos después de
sumar a la lista. Este método lo invocan los vehículos cuando
necesitan entrar en la carretera.
```

```
public void exit(Vehicle vehicle) // Elimina un vehículo de la
carretera. Este método también lo invocan los vehículos cuando tienen
que abandonar una carretera.

void advance(int currTime) // este método es invocado por el
simulador para dar un paso en  el estado de la carretera y, en
particular, para decir a cada coche de esa carretera que avance.

protected void fillReportDetails(IniSection is) // rellena los
valores de una sección ini con los atributos de la carretera para
mostrar un informe de su estado en uno de los pasos de simulación.
```

- ✓ En *advance()* implementa las siguientes operaciones:
  - Calcular su velocidad base (*baseSpeed*) utilizando la fórmula *mın(m, m/max(n,1) +1),* donde *m* es la velocidad máxima de la carretera y *n* es el número de vehículos en la carretera. La división utilizada es la división entera.
  - Para cada vehículo de la lista de vehículos de la carretera:

    a) Poner su velocidad a *baseSpeed/reductionFactor*, donde *reductionFactor* es una función del número de vehículos averiados (considerados obstáculos) que se encuentran delante de este vehículo antes de que ningún vehículo haya avanzado.

El factor de reducción es 1 por defecto si no hay obstáculos y 2 en otro caso.

b) Se pide al vehículo que avance invocando a su método *advance()*.

- ✓ El formato del report debe ser como en el siguiente ejemplo:

```
[road_report]
id = r3
time = 4
state = (v2,80),(v3,67)
```

- ✓ donde *id* es el identificador de la carretera (r3), *time* es el paso de simulación en el cual se ha generado el informe y *state* es una lista con los identificadores y las posiciones de los vehículos que en ese momento están en la carretera, en el mismo orden en el que están en su lista de vehículos.

- ✓ Cuando todos los vehículos han avanzado, si varios están en la misma localización, sus órdenes relativos en la lista de vehículos debe ser el del orden de llegada. Recuerda que la lista tiene que estar ordenada por las localizaciones de los vehículos, siendo la localización 0 la que ocupa la última posición.
- ✓ La clase ***Junction*** permite administrar las carreteras que entran en los cruces utilizando semáforos. Estos pueden estar en verde, permitiendo a los vehículos entrar al cruce y salir por la carretera correspondiente de acuerdo a su itinerario, o pueden estar rojos, impidiendo que los vehículos circulen. En cada paso, habrá solo un semáforo en verde.
- ✓ *La calse Juntion* utiliza la clase *IncomingRoad* con los siguientes atributos y métodos:

```
protected Road road; // carretera
protected List<Vehicle> queue; // cola de vehiculos
protected boolean green; // true si su semaforo esta verde
protected void advanceFirstVehicle() // avanza primero de la cola
protected void addVehicle(Vehicle v) // suma vehiculo a la cola
```

- La clase *Junction* puede incluir:

```
protected List<IncomingRoad> roads; // lista de carreteras entrantes
protected Map<Junction, Road> incomingRoads; // mapa de carreteras
entrantes indicando cual es su cruce origen
void enter(Vehicle vehicle)  // método invocado por los vehículos
para entrar en la cola de una carretera entrante al cruce cuando
llegan al cruce. Los vehículos en la cola del cruce están ordenados
según su orden de llegada (el primero que llega al cruce es el
primero que sale).
void advance(int time) // ver a continuación
protected void fillReportDetails(IniSection is) // rellena los
valores de una sección ini con los atributos del cruce para mostrar
un informe de su estado en uno de los pasos de simulación.
```

- ✓ El método *advance()* debe realizar las siguientes operaciones:
  - Preguntar al primer vehículo de la cola (y solo al primer vehículo en caso de que haya) asociada a la carretera entrante con el semáforo en verde, que avance a su siguiente carretera de acuerdo con su itinerario. Si el avance es posible, el vehículo se elimina de la cola.
  - Actualizar el semáforo. Por defecto, un semáforo estará en verde solamente durante un paso. Transcurrido el paso se pondrá en rojo, y se pasará cíclicamente a la siguiente carretera entrante para poner su semáforo en verde, preservando el orden en el cual las carreteras se añadieron al cruce.
- ✓ Al comienzo de la simulación, antes de que se ejecute ninguna llamada a *advance()*, todos los semáforos estarán en rojo. La primera carretera entrante que pondrá su semáforo en verde será la primera que se añadió al cruce.

- Un ejemplo de formato de texto para los reports de los cruces es el siguiente:

```
[junction_report]
id = j2
time = 5
queues = (r1,red,[]),(r2,green,[v3,v2,v5]),(r3,red,[v1,v4])
```

- donde *id* es el identificador del cruce (j2), *time* es el paso de simulación en el que se ha generado el informe y *queues* es la lista con el estado de todas las carreteras entrantes al cruce en el mismo orden en el que fueron añadidas. Cada carretera entrante se representa con su identificador, el estado de su semáforo asociado (green o red) y la lista de vehículos en la cola.

6. Modelo: Eventos y Ficheros INI

- ✓ Los eventos permiten la interacción con el simulador, añadiendo vehículos, carreteras y cruces, y provocando que algunos coches se averíen. Cada evento tiene un tiempo asociado, en el cual debe ejecutarse. En su representación textual, la clave *time* será opcional. Si no aparece significa que dicho tiempo es 0, y que el evento debe ser ejecutado al inicio de la simulación. En cada paso *t*, el simulador ejecuta todos los eventos cuyo tiempo asociado es *t*, en el orden en que fueron añadidos a la cola de eventos.
- ✓ Para ejecutar los eventos, el simulador invocará a la operación *execute()* de los eventos.
- ✓ Implementa una superclase ***Event*** de la cual hereden todas las clases correspondientes a cada tipo de evento.

- La clase *Event* solo necesita un atributo:

```
protected Integer time; // tiempo de ejecución del evento
```

- El diseño UML contiene métodos para esta clase. Es importante implementar un método abstracto para las subclases de *Event*:

```
public abstract void execute(RoadMap map, int time);
```

- Las subclases incluidas en el diseño UML son: ***NewVehicleEvent***, ***MakeVehicleFaultyEvent***, ***NewRoadEvent***, ***NewJunctionEvent***. Cada una de ellas implementa, al menos, el método *execute()*.
- En la segunda parte de la práctica se añaden mas subclases. Una por cada uno de los eventos de los objetos avanzados (ver mas adelante).

- Los **ficheros** que contienen los eventos utilizan el formato **INI**. Este formato también es utilizado para generar los report por parte de los objetos de simulación.
- En un formato INI puede haber múltiples secciones, cada una de ellas con una etiqueta asociada. Dentro de cada sección puede haber varias líneas, cada línea con una clave textual y un valor asociado.
- Junto con el enunciado de la práctica se proporciona código que puede parsear y generar ficheros y secciones en formato INI. Este paquete permite además comparar dos ficheros o secciones para comprobar su equivalencia. Debes usar esta funcionalidad para asegurarte que la salida de tu práctica es equivalente a la salida esperada que también proporcionamos para muestras de eventos de entrada.

- ✓ Para las secciones en formato INI adoptamos los siguientes convenios:
  - Los identificadores de los objetos de la simulación deben ser strings compuestos únicamente de letras, números y subrayados ("_").
  - Los identificadores de las listas deben contener únicamente identificadores válidos separados por comas, sin espacios en blanco.
- ✓ Tienes que comprobar, antes de ejecutar un evento, que su identificador es válido.
- ✓ Finalmente ten en cuenta que el parser que suministramos tiene además una funcionalidad extra (no presente en el formato INI estándar) que permite ignorar secciones. Para ello basta añadir el símbolo "!" delante de la etiqueta de la sección.

- Por ejemplo, el parser ignorará la sección:

```
[!make_vehicle_faulty]
time = 2
vehicles = v1
duration = 3
```

- Esta funcionalidad es muy útil para experimentar con el código. En lugar de eliminar la sección entera, se puede simplemente comentar.
- A continuación detallamos el formato INI de los distintos objetos de la simulación.
- **New Juntion Event** añade un nuevo cruce a la simulación con los siguientes parámetros:

```
[new_junction]
time = <NONEG-INTEGER>
id = <JUNC-ID>
```

- **New Road Event** añade una nueva carretera al simulador con los siguientes parámetros:

```
[new_road]
time = <NONEG-INTEGER>
id = <ROAD-ID>
src = <JUNC-ID>
dest = <JUNC-ID>
max_speed = <POSITIVE-INTEGER>
length = <POSITIVE-INTEGER>
```

- ✓ **New Vehicle Event** añade un nuevo vehículo a la simulación. El itinerario es una lista de identificadores de cruces y debe contener al menos dos.

```
[new_vehicle]
time = <NONEG-INTEGER>
id = <VEHICLE-ID>
max_speed = <POSITIVE-INTEGER>
itinerary = <JUNC-ID>,<JUNC-ID>(,<JUNC-ID>)
```

- ✓ **Faulty Vehicle Event** provoca que todos los vehículos que aparecen en la lista *vehicles* de identificadores pase a estar en estado averiado durante el tiempo especificado en el campo *duration*.

```
[make_vehicle_faulty]
time = <NONEG-INTEGER>
vehicles = <VEHICLE-ID>(,<VEHICLE-ID>)*
duration = <POSITIVE-INTEGER>
```

7. Modelo: Simulador de Trafico y RoadMap

- ✓ La clase ***TrafficSimulator*** necesita al menos los siguientes atributos:

```
private RoadMap map; // estructura que almacena los objetos simulados
… events; // lista de eventos que hay que ejecutar
private int time; // contador de pasos del simulador
private OutputStream outStream; // flujo de salida utilizado
```

- ✓ El diseño UML del simulador incluye los siguientes métodos:

```
public void addEvent(Event e) // añade un evento a la lista de
eventos. Los eventos tienen que estar ordenados por su atributo
tiempo y, en caso de tiempos iguales, por su orden de llegada. Al
añadir el evento debe comprobarse que su tiempo de ejecución asociado
debe ser mayor o igual que el contador de tiempo del simulador.
public void run(int ticks) // Recibe como parámetros el número de
pasos (ticks), y la salida donde escribir los informes (del tipo
OutputStream). Esta puede ser System.out para escribir en la salida
estándar, o new FileOutputStream(file) para escribir en un fichero.
```

- El pseudo-código del método *run()* es el siguiente:

```
limiteTiempo = this.contadorTiempo + pasosSimulacion - 1;
while (this.contadorTiempo <= limiteTiempo) {
    // 1. ejecutar los eventos correspondientes a ese tiempo
    // 2. invocar al método avanzar de las carreteras
    // 3. invocar al método avanzar de los cruces
    // 4. this.contadorTiempo++;
    // 5. escribir un informe en OutputStream en caso de que no sea
          null
}
```

- Para las primeras operaciones pueden ser útiles los siguientes métodos:

```
private void executeEvents() // ejecuta los eventos de la lista de un
tiempo determinado llamando a su método execute(). Por ejemplo, el
método execute() de NewVehicleEvent suma un vehículo a map.
private void updateRoards() // llama al método advance() de las
carreteras de map
private void updateJunctions() // llama al método advance() de los
cruces de map
```

```
private void printReports() // invoca al metodo generateReport() de
cada uno de los objetos contenidos en la estructura map.
```

- ✓ El informe completo se genera simplemente llamando al método *generateReport()* de cada uno de los objetos simulados en el siguiente orden: primero se muestra la información de los cruces, después de las carreteras y finalmente de los vehículos. En cada caso, el informe del objeto simulado se generará en el orden en el cual fueron añadidos al simulador.
- ✓ La clase ***RoadMap*** mantiene colecciones de objetos simulados para el simulador. Estas colecciones se van actualizando según se ejecutan los eventos.
- ✓ A continuación se muestra parte del diseño UML de esta clase y se dan pautas para su implementación final.

```
private List<Vehicle> vehicles; // lista de vehículos
… roads; // lista de carreteras
… junctions; // lista de cruces
// mapeado de vehículos
// mapeado de carreteras
// mapeado de cruces
void addVehicle(Vehicle vehicle) // añade un vehículo a la lista y al
mapeado de vehículos
void addRoad(Road road) // añade una carretera a la lista y al
mapeado de carreteras
void addJunction(Junction junction) // añade un cruce a la lista y al
mapeado de cruces
void clear() // limpia todas las colecciones
public String generateReport(int time) // llama a generateReport() de
cada objeto simulado
```

8. Control: Controlador y EventBuilder

- La clase ***Controller*** controla la ejecución del simulador. Estos son algunos de los atributos y métodos considerados en su diseño UML:

```
protected TrafficSimulator sim; // simulador que usa
protected OutputStream outputStream; // flujo de salida
protected InputStream inputStream; // flijo de entrada
protected int ticks; // pasos de la simulacion dados por el usuario
EventBuilder[] eventBuilders = {}; // array de constructores de eventos
public void run() // llama a loadEvents() y al método run() del simulador
public void loadEvents(InputStream inStream) // crea un objeto Ini(inStream) y llama al método parse() de los EventBuilder de su array para parsear todos los eventos. Cada evento parseado lo suma al simulador
private Event parseEvent(IniSection sec) // devuelve el objeto Event que corresponde a una seccion INI
```

- ***EventBuilder*** es una clase abstracta que contiene al menos los siguientes atributos:

```
protected String tag;
protected String[] keys;
protected String[] defaultValues;
public abstract Event parse(IniSection section) // método abstracto
// métodos de id valido, parseo de datos enteros, …
```

- Implementar una subclase de *EventBuilder* por cada uno de los tipos de eventos vistos anteriormente. En las constructoras de estas subclases inicializar *tag* y *keys* a los valores correspondientes. En la parte segunda de la práctica es necesario ampliar esta jerarquía de clases con nuevos constructores.

9. Otros: Main y SortedArrayList

- ✓ Con la práctica suministramos una clase ***Main*** que utiliza la librería *commons-cli-1.3.1.jar* para simplificar el parseo de las opciones de la línea de comandos.
- ✓ La clase *Main* debe procesar las siguientes líneas de comandos:

```
usage: es.ucm.fdi.launcher.Main [-h] [-i <arg>] [-o <arg>] [-t <arg>]
 -h,--help Muestra la ayuda.
 -i,--input <arg> Fichero de entrada de eventos.
 -o,--output <arg> Fichero de salida, donde se escriben los informes.
 -t,--ticks <arg> Pasos que ejecuta el simulador en su bucle
principal (el valor por defecto es 10).
```

- ✓ Por ejemplo, podemos ejecutar:

```
java Main -i eventsfile.ini -o output.ini -t 100
java Main -i eventsfile.ini -t 100
```

- ✓ La clase ***SortedArrayList*** implementa una lista ordenada que sirve para mantener ordenada la lista de eventos del simulador.
- ✓ Esta clase debe extender *ArrayList* utilizando genéricos.
- ✓ Veremos un ejemplo de utilización en clase junto con la redefinición de los métodos.
- ✓ Definir un nuevo paquete de miscelaneos para esta clase:

```
package es.ucm.fdi.misc;
```

10. Modelo: Objetos Avanzados

- ✓ Los eventos asociados a los objetos avanzados son similares a los eventos de los que heredan, extendidos con una nueva clave *type* que identifica la variante del evento que debe crearse. Por otro lado, dado que algunos objetos avanzados contienen información adicional, sus eventos asociados podrán tener nuevos campos.
- ✓ De forma similar, los informes asociados a los objetos avanzados son idénticos a sus correspondientes objetos básicos (aquellos de los que heredan), pero deben incluir información sobre su atributo *type* y posiblemente información adicional específica para cada variante particular.

- La clase ***Car*** puede incluir:

```
protected int last_faulty_km;
protected int resistance;
protected int maxFaultDuration;
protected double probability;
protected Random random;
void advance(int time)
```

- Antes de llamar a super.*advance()* deben comprobar las siguientes condiciones:
  - El coche no está averiado.
  - El coche ha viajado al menos *resistance* kilómetros desde la última vez que estuvo averiado. Ten en cuenta que esta distancia no es el total de la distancia recorrida y por lo tanto debes mantener almacenada esta información.
  - Se genera un número aleatorio real *x* entre 0 (incluido) y 1 (excluido) en *random* y se comprueba si este número es menor que la probabilidad de avería (*probability*).

- ✓ Si todas las condiciones son ciertas, entonces el coche pone su contador de avería (*faultyTime*) a un número entero aleatorio entre 1 y *maxFaultDuration* (ambos inclusive) y entonces ejecuta la operación estándar de avanzar (*advance()*). Observa que *makeFaulty()* debe funcionar exactamente igual que en la primera parte.
- ✓ Los coches se añaden a la simulación vía el evento:

```
[new_vehicle]
time = <NONEG-INTEGER>
id = <VEHICLE-ID>
itinerary = <JUNC-ID>,<JUNC-ID>(,<JUNC-ID>)
max_speed = <POSITIVE-INTEGER>
type = car
resistance = <POSITIVE-INTEGER>
fault_probability = <NONEG-DOUBLE>
max_fault_duration = <POSITIVE-INTEGER>
seed = <POSITIVE-LONG>
```

- La entrada *type* = *car* es sólo válida para esta clase de vehículos y debe ser incluida en su informe. El campo *fault_probability* es un número real entre 0 y 1, ambos inclusive. Finalmente, la clave *seed*, que explicaremos más tarde, es opcional y su valor por defecto es *System.currentTimeMillis()*.
- Puesto que esta clase de vehículos tiene un comportamiento aleatorio, es difícil comprobar si la salida producida por el programa, sobre los ejemplos suministrados, encaja con los resultados esperados (que también se adjuntarán con el enunciado). Para solucionar este problema vamos a utilizar un generador de números aleatorios con una semilla de la siguiente forma:

- Esta clase de vehículos recibirá una semilla, *seed* (un número de tipo long), como parámetro en su constructora, y almacenará *new Random(seed)* en un atributo, por ejemplo, *random*.
- Usaremos *random.nextDouble()* para generar un número aleatorio x entre 0 y 1.
- Usaremos *random.nextInt(maxFaultDuration) + 1* para generar un en- tero entre 1 y *maxFaultDuration*.

✓ Cuando testeemos nuestro programa, utilizaremos la misma semilla que la utilizada para generar la salida de los ejemplos suministrados. De esta manera ambos ficheros deben coincidir.

- ✓ La clase ***Bike*** no necesita añadir atributos a los de su superclase.
- ✓ El método *super.makeFaulty()* no se aplicará a las bicicletas que viajen a una velocidad menor o igual que la mitad de su velocidad máxima.
- ✓ Las bicicletas se añaden a la simulación vía el siguiente evento:

```
[new_vehicle]
time = <NONEG-INTEGER>
id = <VEHICLE-ID>
itinerary = <JUNC-ID>,<JUNC-ID>(,<JUNC-ID>)
max_speed = <POSITIVE-INTEGER>
type = bike
```

- ✓ La entrada *type = bike* es sólo válida para esta clase de vehículos y por lo tanto debe ser incluida en su informe.

- La clase ***LanesRoad*** implementa autopistas con varios carriles y soportan mayor tráfico que las carreteras convencionales.

```
protected int numLanes; // numero de carriles
```

- Los vehículos podrán circular a mayor velocidad incluso aunque haya varios vehículos averiados.
- Su factor de reducción *reductionFactor* es 1 siempre que haya mas carriles que vehículos averiados (i.e., *numLanes* > *numObstacles*) y 2 en otro caso.
- La velocidad base (*baseSpeed*) de una autopista también considera el número de carriles y se define como $min(m, m * l / max(n,1) + 1)$, donde m es la velocidad máxima, n es el número de vehículos en la carretera y l es el número de carriles. La división es entera.
- Las autopistas se añaden usando el siguiente evento:

```
[new_road]
time = <NONEG-INTEGER>
id = <ROAD-ID>
src = <JUNC-ID>
dest = <JUNC-ID>
max_speed = <POSITIVE-INTEGER>
length = <POSITIVE-INTEGER>
type = lanes
lanes = <POSITIVE-INTEGER>
```

- ✓ donde *type = lanes* es una entrada válida sólo para este tipo de carreteras y por tanto debe ser incluida en los informes.
- ✓ La clase ***DirtRoad*** implementa caminos por los que es mas difícil conducir si hay vehículos averiados.
- ✓ Su factor de reducción (*reductionFactor*) será 1 más el número de coches averiados. Su velocidad base (*baseSpeed*) se define como la velocidad máxima de la carretera.
- ✓ Los caminos se añaden utilizando el siguiente evento:

```
[new_road]
time = <NONEG-INTEGER>
id = <ROAD-ID>
src = <JUNC-ID>
dest = <JUNC-ID>
max_speed = <POSITIVE-INTEGER>
length = <POSITIVE-INTEGER>
type = dirt
```

- ✓ donde la entrada *type = dirt* es una entrada válida sólo para esta clase de objetos y por lo tanto debe incluirse en sus informes.
- ✓ La clase ***RoundRobinJunction*** implementa cruces circulares y añade los siguientes atributos a su superclase:

```
protected int maxTimeSlice;
protected int minTimeSlice;
```

- En este tipo de cruce, los semáforos están en verde durante períodos de tiempo mas largos llamados *time slice*.
- En lugar de ir cambiando el semáforo en cada paso de la simulación como se hace en los cruces convencionales, en este tipo de cruces el semáforo estará en verde durante un intervalo de tiempo [*minTimeSlice*, *maxTimeSlice*].
- Estos cruces se representan en modo textual como:

```
[new_junction]
time = <NONEG-INTEGER>
id = <JUNC-ID>
type = rr
max_time_slice = <POSITIVE-INTEGER>
min_time_slice = <POSITIVE-INTEGER>
```

- ✓ Al comienzo de la simulación el intervalo de tiempo (*timeSlice*) de cada carretera entrante se pone a *maxTimeSlice* y su correspondiente *usedTimeUnits* se pone a 0.
- ✓ Ten en cuenta que al comienzo de la simulación, antes de ejecutar ninguna llamada a avanza, todos los semáforos están en rojo.
- ✓ Cuando se le pide al cruce que actualice sus semáforos, primero comprueba si el cruce que tiene el semáforo en verde ha consumido su intervalo (i.e., *usedTimeUnits* es igual a *timeSlice*).
- ✓ Si fuera así, entonces el cruce hace lo siguiente:

- Pone el semáforo verde a rojo.
- Actualiza el *timeSlice* de la correspondiente carretera entrante como sigue:
  - Si el intervalo de tiempo ha sido completamente usado (en cada paso de la simulación ha pasado un vehículo de esa carretera entrante), entonces *timeSlice* se pone a *mın(timeSlice + 1, maxTimeSlice).*
  - Si el intervalo de tiempo no se ha usado nada (ningún vehículo ha pasado mientras el semáforo estaba en verde), entonces *timeSlice* se pone a max(timeSlice− 1, minTimeSlice).
  - En otro caso no modificados *timeSlice*.
- Pone *usedTimeUnits* a 0.
- Pone en verde el semáforo de la siguiente carretera entrante, en el orden en que fueron añadidas. Observa que cuando todos los semáforos están en rojo, la siguiente carretera entrante es la primera que se añadió al cruce.

- ✓ Los vehículos de la cola se ordenan respetando el orden de llegada (el primero que entra es el primero que sale).
- ✓ Los informes para estos cruces contienen la misma información que para los cruces convencionales, pero ahora, para cada semáforo verde, mostramos también las unidades de tiempo restantes. Por ejemplo, para:

```
[junction_report]
id = j2
time = 5
type = rr
queues = (r1,red,[]),(r2,green:3,[v3,v2,v5]),(r3,red,[v1,v4])
```

- ✓ se indica que la carretera *r2* tiene el semáforo en verde y todavía le quedan tres pasos más para cambiar a rojo.

- ✓ La clase ***MostCrowdedJunction*** implementa los cruces congestionados.
- ✓ Cuando un cruce está congestionado, la siguiente carretera entrante en poner su semáforo en verde es aquella que tiene más vehículos en su cola. En caso de que haya varias carreteras entrantes con la misma longitud en sus colas, entonces se tiene en cuenta el orden en que fueron añadidas a la simulación.
- ✓ Al inicio de la simulación *timeSlice* y *usedTimeUnits* son 0. Cuando se pide un a cruce congestionado que actualice su semáforo en verde, entonces primero comprueba si el cruce que actualmente tiene el semáforo en verde ha consumido su intervalo de tiempo (i.e., *usedTimeUnits* es igual a *timeSlice*). Si este fuera el caso, entonces el cruce se comporta como sigue:

- ✓ Al inicio de la simulación *timeSlice* y *usedTimeUnits* son 0. Cuando se pide un a cruce congestionado que actualice su semáforo en verde, entonces primero comprueba si el cruce que actualmente tiene el semáforo en verde ha consumido su intervalo de tiempo (i.e., *usedTimeUnits* es igual a *timeSlice*). Si este fuera el caso, entonces el cruce se comporta como sigue:
  - Poner el semáforo verde a rojo.
  - Se busca la carretera entrante (distinta a la actual) con más vehículos en la cola respetando el orden de llegada. Se pone su semáforo a verde y se le asigna un *timeSlice* igual a *max(n/2 , 1)*, donde *n* es el número de vehículos en la cola de la carretera, y *usedTImeUnits* se pone a 0. La división es entera.
- ✓ Los vehículos de la cola de las carreteras entrantes a estos cruces se ordenan de acuerdo al orden de llegada.

- ✓ Además al inicio de la simulación, antes de ejecutar ninguna llamada a avanza, todos los semáforos están en rojo.
- ✓ Los eventos que definen los cruces congestionados tienen el siguiente formato:

```
[new_junction]
time = <NONEG-INTEGER>
id = <JUNC-ID>
type = mc
```

- ✓ La entrada *type = mc* es única para esta clase de objetos y por tanto debe aparecer en sus informes.

11. Manejo de Errores

- ✓ El manejo de errores, tales como:
  - añadir objetos de la simulación con el mismo identificador,
  - referenciar objetos simulados que no existen,
  - utilizar valores inválidos de parámetros, etc.,

  deben ser controlados usando tratamiento de excepciones.
- ✓ El uso de flujos de entrada y salida implica el uso de *IOException*.
- ✓ La clase *SimulatorError* incluida en el disenio UML extiende *RuntimeException* y sirve para gestionar los errores del simulador de trafico.

12. Material de la Practica

- ✓ Para la realización de la práctica tenéis el siguiente material:
  - Especificaciones del Simulador (Enunciado)
  - Diseño de Alto Nivel (Arquitectura)
  - Diseño Detallado de Clases (UML)
  - Librería *commons-cli-1.3.1.jar* (incluir directorio lib/ con esta librería en el proyecto de Eclipse).
  - Código:
    - es.ucm.fdi.ini.* (para parsear los eventos de ficheros INI)
    - es.ucm.fdi.launcher.ExampleMain.java
  - Ejemplos de ejecución (cada ejemplo de entrada tiene su correspondiente fichero de salida, y esto es lo que tenéis que obtener con la ejecución de vuestro código)
    - Ficheros input.ini
    - Ficheros output.ini