Práctica 1

El controlador

Controller: clase encargada de leer del fichero de entrada las estructuras JSON, transformarlas en eventos y cargar dichos eventos en el simulador.

```
private TrafficSimulator _sim;
private Factory<Event> _eventsFactory;


// eventsFactory y sim se crean en "Main"
public Controller(TrafficSimulator sim, Factory<Event> eventsFactory) ...

public void reset() {
      _sim.reset();
 }
}
```

2

El controlador

Controller: clase encargada de leer del fichero de entrada las estructuras JSON, transformarlas en eventos y cargar dichos eventos en el simulador.

```
private TrafficSimulator _sim;
private Factory<Event> _eventsFactory;

// eventsFactory y sim se crean en "Main"
public Controller(TrafficSimulator sim, Factory<Event> eventsFactory) ...

public void loadEvents(InputStream in) {
        JSONObject jsonInupt = new JSONObject(new JSONTokener(in));



}
```

Este método primero convierte la entrada JSON en un objeto JSONObject utilizando:

3

El controlador

Controller: clase encargada de leer del fichero de entrada las estructuras JSON, transformarlas en eventos y cargar dichos eventos en el simulador.

```
private TrafficSimulator _sim;
private Factory<Event> _eventsFactory;

// eventsFactory y sim se crean en "Main"
public Controller(TrafficSimulator sim, Factory<Event> eventsFactory) ...

public void loadEvents(InputStream in) {
        JSONObject   jo = new JSONObject(new JSONTokener(in));

        JSONArray events = jo.getJSONArray("events");

        }
```

Extrae lita de evento de jo,

4

El controlador

Controller: clase encargada de leer del fichero de entrada las estructuras JSON, transformarlas en eventos y cargar dichos eventos en el simulador.

```
private TrafficSimulator _sim;
private Factory<Event> _eventsFactory;

// eventsFactory y sim se crean en "Main"
public Controller(TrafficSimulator sim, Factory<Event> eventsFactory) ...

public void loadEvents(InputStream in) {
        JSONObject jo = new JSONObject(new JSONTokener(in));
        JSONArray events = jo.getJSONArray("events");

        for (int i = 0; i < events.length(); i++) {
            _sim.addEvent(_eventsFactory.createInstance(events.getJSONObject(i)));
        }
}
```

crea el evento correspondiente utilizando la *factoría de eventos*

5

El controlador

Controller: clase encargada de leer del fichero de entrada las estructuras JSON, transformarlas en eventos y cargar dichos eventos en el simulador.

```
public void run(int n, OutputStream out) {
  if (out == null) {
    out = new OutputStream() {
      @Override
        public void write(int b) throws IOException {}
      };
    }
  PrintStream p = new PrintStream(out);
  p.println("{");
  p.println("  \"states\": [");

  // escribir los n-1 primeros pasos (para que no salga coma
  // en el último paso
  // _sim.advance(); p.print(_sim.report()); p.println(",");

  // escribir el último paso

  p.println("]");
  p.println("}");
}
```

{"states": [*s*1,...,*sn*] }

6

La clase Main

Es la encargada de procesar los argumentos de entrada, cargar los eventos usando el controlador, y mandar al controlador que ejecute la simulación.

- java Main –h

usage: Main [-h] -i <arg> [-o <arg>] [-t <arg>]
-h,--help Print this message
-i,--input <arg> Events input file
-o,--output <arg> Output file, where reports are written.
-t,--ticks <arg> Ticks to the simulator's main loop (default value is 10).

java Main -i eventsfile.json o output.json -t 100
java Main 100 -i eventsfile.json -t 100

7

La clase Main

Es la encargada de procesar los argumentos de entrada, cargar los eventos usando el controlador, y mandar al controlador que ejecute la simulación.

```
private final static Integer _timeLimitDefaultValue = 10;
private static Integer _timeLimit = null;          // número de pasos
private static String _inFile = null;                    // fichero del que se leen eventos
private static String _outFile = null;             // fichero de salida
private static Factory<Event> _eventsFactory = null;      // factoría de eventos
```

8

La clase Main

Es la encargada de procesar los argumentos de entrada, cargar los eventos usando el controlador, y mandar al controlador que ejecute la simulación.

```
private final static Integer _timeLimitDefaultValue = 10;
private static Integer _timeLimit = null;          // número de pasos
private static String _inFile = null;                    // fichero del que se leen eventos
private static String _outFile = null;             // fichero de salida
private static Factory<Event> _eventsFactory = null;      // factoría de eventos
```

```
public static void main(String[] args) {
  try {
      start(args);
  } catch (Exception e) {
       e.printStackTrace();
  }
}
```

```
private static void start(String[] args) throws IOException {
     InitFactories();       // inicia las factorías
     parseArgs(args);    // procesa los argumentos de entrada
     StartBatchMode();  // inicia la simulación con E/S estándar
}
```


La clase Main

Es la encargada de procesar los argumentos de entrada, cargar los eventos usando el controlador, y mandar al controlador que ejecute la simulación.

```
private static void initFactories() {
   // se crean las estrategias de cambio de semáforo
   ArrayList<Builder<LightSwitchingStrategy>> lsbs = new ArrayList<>();
   lsbs.add(new RoundRobinStrategyBuilder());
   lsbs.add(new MostCrowdedStrategyBuilder());
   Factory<LightSwitchingStrategy> lssFactory = new BuilderBasedFactory<>(lsbs);
```

La clase Main

Es la encargada de procesar los argumentos de entrada, cargar los eventos usando el controlador, y mandar al controlador que ejecute la simulación.

```
private static void initFactories() {
   // se crean las estrategias de cambio de semáforo
   ArrayList<Builder<LightSwitchingStrategy>> lsbs = new ArrayList<>();
   lsbs.add(new RoundRobinStrategyBuilder());
   lsbs.add(new MostCrowdedStrategyBuilder());
   Factory<LightSwitchingStrategy> lssFactory = new BuilderBasedFactory<>(lsbs);

   // se crean las estrategias de extracción de la cola
   ArrayList<Builder<DequeuingStrategy>> dqbs = new ArrayList<>();
   dqbs.add(new MoveFirstStrategyBuilder());
   dqbs.add(new MoveAllStrategyBuilder());
   Factory<DequeuingStrategy> dqsFactory = new BuilderBasedFactory<>(dqbs);
```

11

La clase Main

Es la encargada de procesar los argumentos de entrada, cargar los eventos usando el controlador, y mandar al controlador que ejecute la simulación.

```
private static void initFactories() {
   // se crean las estrategias de cambio de semáforo
   ….

   // se crean las estrategias de extracción de la cola
   …

   // se crea la lista de builders
   List<Builder<Event>> eventBuilders = new ArrayList<>();

   eventBuilders.add(new NewJunctionEventBuilder(lssFactory, dqsFactory));
   eventBuilders.add(new NewCityRoadEventBuilder());
   ...
  _eventsFactory = new BuilderBasedFactory<>(eventBuilders);
}
```

12

La clase Main

Es la encargada de procesar los argumentos de entrada, cargar los eventos usando el controlador, y mandar al controlador que ejecute la simulación.

```
private static void startBatchMode() throws IOException {

    InputStream in = new FileInputStream(new File(_inFile));
    OutputStream out = _outFile == null ?
            System.out : new FileOutputStream(new   File(_outFile));

    TrafficSimulator sim = new TrafficSimulator();
    Controller ctrl = new Controller(sim, _eventsFactory);

    ctrl.loadEvents(in);
    ctrl.run(_timeLimit, out);
    in.close();
    System.out.println("Done!");
}
```

13