Notes on Assignment 4
Design Guide

In these slides we only discuss design issues of assignment 4, they are complementary to what have been explained in class and to the assignment's statement ...

Simulated Objects

Lets first build the classes that correspond to the **basic objects** we want to simulate ...

SimulatedObject

```
public abstract class SimulatedObject {

    protected String _id;

    public SimulatedObject(String id) { ... }
    public String getId() { ... }

    public String toString() {
        return _id;
    }

    public String generateReport(int time) {
        IniSection is = new IniSection(getReportSectionTag());
        is.setValue("id", _id);
        is.setValue("time", time);
        fillReportDetails(is);
        return is.toString();
    }

    protected abstract void fillReportDetails(IniSection is);
    protected abstract String getReportSectionTag();
    abstract void advance();
}
```

Simulated Object keep the ID

toString should just return the ID

When generating a report, we ask the sub classes for the section tag and also to fill in extra details

All simulated objects have a method 'advance' and two methods for filling reports

Vehicle

```
public class Vehicle extends SimulatedObject {
    ...

    public Vehicle(String id, int maxSpeed, List<Junction> itinerary) { ... }

    public Road getRoad() { ... }
    public int getMaxSpeed() { ... }
    public int getSpeed() { ... }
    public int getLocation() { ... }
    public int getKilometrage() { ... }
    public int getFaultyTime() { ... }
    public boolean atDestination() { ... }
    public List<Junction> getItinerary() { ... }

    void makeFaulty(int faultyTime) { ... }
    void setSpeed(int speed) { ... }
    void advance() { ... }
    void moveToNextRoad() { ... }

    protected String getReportSectionTag() { ... }
    protected void fillReportDetails(IniSection is) { ... }
}
```

You should keep track (as fields) of location, kilometrage, remaining faulty, the road on which it is travelling, etc.

Methods for consulting information

add faultyTime to the remaining faulty time counter. You should put current speed to zero as well

set the current speed. You should never exceed the maximum speed of the vehicle. Speed of a faulty vehicle should always be zero

see next slide for 'advance' and 'moveToNextRoad'

fill in the report when requested by the super class

Vehicle - advance

```
void advance() {
```

1. if the faulty time counter is not zero, decrease it by one and do nothing else

2. if the vehicle is waiting in a junction, and is not faulty, do nothing ...

3. if the vehicle is not faulty, and is not waiting in a junction:
    1. new location = current location + current speed
    2. the new location should not exceed the length of the road
    3. update the kilometrage
    4. if at the end of the road enter the destination junction (most likely you'll need to keep a boolean to indicate that the vehicle is waiting in a junction). You should also put current speed to zero

```
}
```

Vehicle - moveToNextRoad

```
void moveToNextRoad() {
```

When adding a vehicle to the road map, this method should be called so the vehicle enter the first road — see the RoadMap class. It will also be called by Junction to tell the vehicle to move to the next road

1. if the current road is **not null**, exit the current road
2. modify the location to 0
3. calculate the next road
4. move to the next road if there is a next road (you should modify the boolean that indicates if the vehicle is waiting in a junction, and if the vehicle has arrived)

```
}
```

Road

```
public class Road extends SimulatedObject {

    protected Junction _srcJunc;
    protected Junction _destJunc;
    protected List<Vehicle> _vehicles;
    ...

    public Road(String id, int length, int maxSpeed, Junction src, Junction dest) {...}

    public Junction getSource() { ... }
    public Junction getDestination() { ... }
    public int getLegth() { ... }
    public List<Vehicle> getVehicles() { ... }

    void advance() { ... }


    void enter(Vehicle vehicle) { ... }
    void exit(Vehicle vehicle) { ... }

    protected String getReportSectionTag() { ... }
    protected void fillReportDetails(IniSection is) { ... }

    protected int calculateBaseSpeed() { ... }
    protected int reduceSpeedFactor(int obstacles) { ... }
}
```

You should keep track (as fields) of all required information ...

The list of vehicles should be kept sorted by location, descending order

Methods for consulting information

Traverse the list of vehicels, for each one calculate and set its speed as a function of the base-speed and the number of faulty vehicles a head, and ask it to advance. At the end sort the list of vehicles again.

Add/remove the vehicle to/from the list of vehicles. Keep the list sorted!

fill in the report when requested by the super class

Keeping these calculations in a separate methods allow us to get different behaviours simply by overriding them

Junction

```
public class Junction extends SimulatedObject {

    protected List<IncomingRoad> _roads;
    ...

    public Junction(String id) { ... }


    public Road roadTo(Junction junction) { ... }
    public Road roadFrom(Junction junction) { ... }
    public List<IncomingRoad> getRoadsInfo() { ...}

    void addIncomingRoad(Road road) {...}
    void addOutGoingRoad(Road road) {...}

    void enter(Vehicle vehicle) { ... }
    void advance() { ... }
    protected IncomingRoad createIncomingRoadQueue(Road road) { ... }
    protected void switchLights() { ... }

    protected String getReportSectionTag() { ... }
    protected void fillReportDetails(IniSection is) { ... }
}
```

static nested class managing the queue for one road

```
static public class IncomingRoad
        protected Road _road;
        protected List<Vehicle> _queue;
        protected boolean _green;

        protected IncomingRoad(Road road) { ... }

        public Road getRoad() { ... }
        public boolean hasGreenLight() { ... }
        protected void setGreen(boolean green) { ... }
        protected void advanceFirstVehicle() { ... }
        protected void addVehicle(Vehicle v) { ... }
        public String toString() { ... }
}
```

Explained later

List of queues for roads, ordered according to order in which they where added to the list

You should keep track (as fields) of all required information ...

Methods for consulting information

These methods are called by RoadMap to tell the junction to which roads it is connected. Keep this information in a table, later it will be used by Vehicles (via roadTo, roadFrom) to know to which road it should move. When adding an incoming road, you should add a corresponding instance of IncomingRoad to the list _roads, create this instance using the method createIncomingRoadQueue, this way if we override this method we get different behaviours (see next slide)

These methods are explained in the next line

fill in the report when requested by the super class

Junction

```
public class Junction extends SimulatedObject {

    protected List<IncomingRoad> _roads;
    ...

    void enter(Vehicle vehicle) { ... }
    void advance() { ... }

    protected IncomingRoad createIncomingRoadQueue(Road road) { ... }
    protected void switchLights() { ... }

    ...
}
```

Vehicles call this method to enter the corresponding queue. You should look for the corresponding IncomeingRoad instance in the list _roads and call its addVehicle

Advance looks for the road with a green light (in the list _roads), if any, and asks it to advance one vehicle by calling advanceFirstVehicle. Then it calls method swithLights() to switch the traffic lights. Note that, for simplicity, we assume that at most one road might have a green light.

static nested class representing a queue for a road

```
static public class IncomingRoad
        protected Road _road;
        protected List<Vehicle> _queue;
        protected boolean _green;

        protected IncomingRoad(Road road) { ... }

        public Road getRoad() { ... }
        public boolean hasGreenLight() { ... }
        protected void setGreen(boolean green) { ... }
        protected void advanceFirstVehicle() { ... }
        protected void addVehicle(Vehicle v) { ... }
        public String toString() { ... }
}
```

Explained later

Just returns

"new IncommingRoad(road)"

It is important to keep it separated from addRoad so we can override it to get different behaviour (see advanced junctions). The light should be red at the beginning

Looks for the road with green light and turn it to red, then it turn the next one in the list road to green. It is important to keep it separated from method advance so we can override it to get different behaviour (see advanced junctions). (see advanced junctions)

Junction - Incoming Road

```
static public class IncomingRoad {
    protected Road _road;
    protected List<Vehicle> _queue;
    protected boolean _green;

    protected IncomingRoad(Road road) { ... }

    public Road getRoad() { ... }

    public boolean hasGreenLight() { ... }
    protected void setGreen(boolean green) { ... }

    protected void advanceFirstVehicle() { ... }
    protected void addVehicle(Vehicle v) { ... }

    public String toString() { ... }
}
```

a static nested class of Junction representing a queue for a road

The road associated to the queue

The queue, just a list, the first is the one served first. Depending on how we order vehicles in the list they will be served in one way or a mother

has a green or red light

create an empty queue and set the light to red

returns the road

Consult/set the light

tell the first vehicle in _queue, if any, to moveToNextRoad and remove it from the queue

Add vehicle v to the queue, i.e., at the end of the list _queue. Note that if we override this method we can order vehicles in the queue according to different criterions

return a string that represents the queue as require in the junction reports, e.g., (r3,green,[v2,v5,v1])

The Road Map

Now that we have our basic objects, let us build a maps that serves like a "data base" that has all simulated objects in a road system ...

RoadMap

```
public class RoadMap {
    ...
    RoadMap() { ... }

    public Vehicle getVehicle(String id) { ... }
    public Road getRoad(String id) { ... }
    public Junction getJunction(String id) { ... }

    public List<Vehicle> getVehicles() { ... }
    public List<Road> getRoads() { ... }
    public List<Junction> getJunctions() { ... }

    void addJunction(Junction junction) { }
    void addRoad(Road road) { ... }
    void addVehicle(Vehicle vehicle) { ... }

    void clear() { ... }

    public String generateReport(int time) { ... }
}
```

You should have lists for junctions, roads and vehicles. Each ordered in the same order in which its elements were added. You might also want to keep Maps from identifiers to objects of each kind, e.g., Map<String,Vehicle>, in order to make searching for an object that corresponds to an identifier more efficient.

Consult simulated objects by ID

Consult the different lists of simulated objects

Adds a Junction to the road map, you should check that there is no other junction has the same ID

Adds a Road to the road map, you should check that there is no other road has the same ID. You should tell the corresponding junctions to add this road as incoming/outgoing road

Adds a Vehicle to the road map, you should check that there is no other vehicle has the same ID. You should tell the vehicle to moveToNextRoad

Delete all objects from lists/maps

Generate a report by concatenating the reports of all objects. Firsts all junction reports, then all road reports, then all vehicle reports

Events

Now we have simulated object, and class for maintaining a road map. We will add events that do things on the road map and the objects ...

What is an Event?

```
public abstract class Event {
    protected Integer _time;

    public int getScheduledTime() {
        return _time;
    }
```

It has a time on which it is supposed to be executed, it is not important now who will execute it ....

We can consult the time of an event

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

}
```

It has a method execute that receives a road map, and maybe the current time of the simulator (the time is not really needed), and it does something on the map, e.g., adds a junction, adds a road, adds a vehicle, makes a vehicle faulty, etc.

Example: NewJunctionEvent

```
public class NewJunctionEvent extends Event {

    protected String _id;

    public NewJunctionEvent(int time, String id) {
        super(time);
        _id = id;
    }

    @Override
    public void execute(RoadMap map, int time) {
        map.addJunction(new Junction(_id));
    }

    @Override
    public String toString() {
        return "New Junction " + _id;
    }

}
```

This is an event that add a Junction with a specific _id to the road map

In the constructor it receives the time and the Junction identifier ....

In execute it simple adds a new Junction with the corresponding _id to the road map ...

More Events

NewJunctionEvent: adds a new junction

NewRoadEvent: adds a new road

NewVehicleEvent: adds a new vehicle

MakeVehicleFaulty: makes a car faulty for some time units

We will have more **event for advanced objects** ....

Traffic Simulator

Now we have simulated object, and class for maintaining a road map, and events. Let us write the simulator that:

1. maintains a road map
2. execute events when its time to execute them
3. advances the state of the road system;
4. and write reports on the state of the road system

TrafficSimulator

```
public class TrafficSimulator {

    private RoadMap _map;
    private List<Event> _events;
    private int _time;
    private OutputStream _outStream;

    public TrafficSimulator(OutputStream outStream) { ... }

    public void reset() { ... }
    public void setOutputStream(OutputStream outStream) { ... }
    public void addEvent(Event e) { ... }
    public void run(int ticks) { ... }

}
```

It has a road map, which is initial empty

It has a list of events, order by time (ascending) and events with the same time are ordered according to the order in which they were added

Discrete time counter, initial 0

Output stream to which reports are written

Delete events, delete roadmap, put _time to 0

Change the _outStream

Adds an event to the list of events, respecting the order. The time of e MUST be greater than or equal to the simulator's time (throw an exception if not)

```
// execute the simulator for `ticks' time units
limit = _time + ticks - 1;
while (time <= limit) {
  1. execute all events scheduled for this current time (i.e., _time)
  2. call advance() of all roads
  3. call advance() of all junctions
  4. _time++;
  5. print  _map.generateReport(_time) to the _outStream if it is not null
}
```

The workflow so far ...

With all the pieces we have so far, one can use the simulator as follows:

1. Create an instance of TrafficSimulator and set its output stream, e.g., to a file or to the standard output
2. Create events and add them to the simulator
3. Execute the 'run' method of the simulator for some ticks
4. Inspect the output

But ...

1. We don't expect users to write the input, i.e., events, directly in Java, but rather describe them textually, e.g., in text files.
2. Someone should read these textual representations and convert them to event objects so they can be added to the simulator.

Event Builders

Its time to start building a "bridge" between the simulator and the user!

1. The event input files will be in INI format, where each INI section describes an event — see assignment text
2. Let us assume that someone has parsed these files already and constructed the INI sections (using the INI library we provided you)
3. Now we want to write event builders, that take an INI section as input and return a corresponding event if they recognise the event described by the INI section

EventBuilder

```
public abstract class EventBuilder {
    public abstract Event parse(IniSection section);
}
```

Takes an INI section that describes an event as input, and builds a corresponding Event and returns it, and if it cannot, because it does not understand the INI section, it simply returns null (so we can try another builder)

Ex. Event Builder for a junction.

```
[new_junction]
time = 0
id = j2
```

```
class NewJunctionEventBuilder extends EventBuilder {
    public Event parse(IniSection section) {
        if ( !section.getTag().equals("new_junction") )
            return null;

        return new NewJunctionEvent(
                        EventBuilder.parseNonNegInt(section, "time", 0),
                        EventBuilder.validId(section, "id"));
    }
}
```

if the section tag is not the expected one, return 'null', i.e., say I'm not the one who understands this kind of events, otherwise extract info and build and return the event

make sure the section has an 'id' key with valid, and returns it

make sure the section an 'time' key with valid, and returns it

The Controller

Yet another step towards the user

1. We want to build something that can take text files as input, load all events, add them to the simulator, etc
2. It is also able to tell the traffic simulator to run for some ticks, to reset, etc.
3. Later this will be connected to the Main class

Controller

```
public class Controller {
    protected TrafficSimulator _sim;
    EventBuilder[] _eventBuilders = {};
    ...

    public Controller(TrafficSimulator sim) { ... }

    public void setEventBuilders(EventBuilder[] eventBuilders) {
        _eventBuilders = eventBuilders;
    }

    public void reset() { ... }
    public void setOutputStream(OutputStream outStream) { ... }
    public void run(int ticks) { ... }


    public void loadEvents(InputStream inStream) { ... }

}
```

It has a traffic simulator that is received in the constructor

An array of EventBuilder(s) that is provided from outside, this was we can configure the controller to support some set of events or another

These methods can reset the simulator, change its output stream, or run it for 'tick' time units

The inStream is supposed to contain events in INI format. First construct an instance of class Ini from the inStream, i.e., "x = new Ini(inStream)". Then try to parse each ini section in x.getSections() using the _eventBuilders, by calling their parse method. The first one that succeeds will return an event that you should ad to the simulator. You should report errors for unrecognised events.

The main

We want users to run the program using:

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

We already gave you a Main class that parses the command-line, you just have to complete some methods ...

Main

```
public class ExampleMain {
    ...
    private static EventBuilder[] _eventBuilders = {
            new NewJunctionEventBuilder(),
            new NewRoadEventBuilder(), ...
    };

    private static Integer _timeLimit = null;
    private static String _inFile = null;
    private static String _outFile = null;

    private static void parseArgs(String[] args) { ... }
    ...
    private static void startBatchMode() ... { ... }
    ...
}
```

Supported event builders to be passed to controller ...

This method parses the command line arguments and leaves their values in the corresponding fields. It is called from method main. You don't need to understand it now, but in assignment 5 you will need to add more arguments. You need to include the library commons-cli-1.3.1.jar in your Java project

1. Create an InputStream from _inFile
2. Create an OutputStream from _outFile or use System.out if it is null
3. Create a TrafficSimulator
4. Create a Controller passing it the simulator
5. Pass the event builders to the controller
6. Set the output stream via the controller
7. load the event files via the controller
8. run the simulator for _timeLimit ticks time units via the simulator

Advanced Objects

Now that we have the system working, we can start using OO to define new behaviour for Junctions, Roads, and Vehicles ...

Note that for each advanced simulated object, you should create a corresponding event and event builder ...

Lanes and Dirt Roads

These advanced objects are quite easy, they are only different from Road only in the way they calculate the base-speed and the speed-reduction-factor.

Just inherit from class Road and override methods calculateBaseSpeed and reduceSpeedFactor. You should also override the method for filling in the report in order to add the "type" key-value to the report.

Cars and Bikes

These advanced objects are also quite easy, they inherit from Vehicle and override few things …

For Bike you just need to change the behaviour of the makeFaulty method …

For Car you should change 'advance' to make the car faulty if some conditions (see assignment text) hold, and then call the 'advance' method of the super class. Note that you should keep track of the kilometrage when the car was faulty last time (simply store the current kilometrage if the car is faulty)

In both cases you should also override the method for filling in the report in order to add the "type" key-value to the report.

Advanced Junctions

These advanced objects look more complicated 😱, but since we have a "good" design it should not be too much 😌 ...

The new kind of junction require to track extra information for each road, for example the time slice, used time slice, etc.

We define an abstract class called JunctionWithTimeSlice that manages this extra information, it will do this by defining a nested class that inherits from IncomingRoad and has all the extra information/operations needed ...

Then we define advanced junctions RoundRobinJunction and MostCrowdedJunction by inheriting from this new abstract class ...

JunctionWithTimeSlice

```
public abstract class JunctionWithTimeSlice extends Junction {

    static protected class IncominRoadWithTimeSlice extends IncomingRoad {
        int _timeSlice;
        int _usedTimeUnits;
        boolean _fullyUsed;
        boolean _used;

        protected IncominRoadWithTimeSlice(Road road) { ... }

        public int getTimeSlice() { ... }
        protected void setTimeSlice(int timeSlice) { ... }
        public int getUsedTimeUnits() { ... }
        protected void setUsedTimeUnits(int usedTimeUnits) { ... }
        public boolean isFullyUsed() { ... }
        protected void setFullyUsed(boolean fullyUsed) { ... }
        public boolean isUsed() { ... }
        protected void setUsed(boolean used) { ... }

        protected void advanceFirstVehicle() { ... }

        public String toString() { ... }
    }

    public JunctionWithTimeSlice(String id) {
        super(id);
    }

}
```

It adds the extra information in the IncomingRoad, by extending it

Methods for consulting and setting information

Should advance the first vehicle in the queue as before and modify _usdTimeUnits, _fullyUsed, and _used as follow:

1. _usdTimeUnits is incremented by one
2. _fullyUsed is set to false if there are no cars in the queue — this means that green light was not fully used
3. _used is set to true if the queue is not empty — means that the green light has been used at least once
4. advance the first vehicle, if any

return a string that represents the queue as require in the advanced junction reports.

RoundRobinJunction

```
public class RoundRobinJunction extends JunctionWithTimeSlice {
    ...
    public RoundRobinJunction(String id, int minTimeSlice, int maxTimeSlice) {
        super(id);
        ...
    }

    protected IncomingRoad createIncomingRoadQueue(Road road) { ... }

    protected void switchLights() { ... }

    protected void fillReportDetails(IniSection is) {

    }
```

Constructor receives the min and max time slice

This will be called by class Junction to create a queue for a road, now it will create IncomingRoadWithTimSlice instead of IncomingRoad, its light should be red. Set its time slice to the maximum at the beginning — see assignment text

We override the switchLights behaviour of Junction

1. If all lights are red (the first time), then turn the light of the first road to green. Otherwise,
2. If the current road with green light has consumed its time slice, then turn it to red and modify its timeSlice as described in the assignment text (depending if it was used, fully used, etc.) and turn the next one in the list to green.

IMPORTANT: when retrieving an element from the list _roads, it will be of type IncomingRoad, cast to InComingRoadWithTimeSlice in order to access/set the different information ...

Override the method for filling in the report in order to add the "type" key-value to the report.

MostCrowderJunction

```
public class MostCrowdedJunction extends JunctionWithTimeSlice {
    ...
    public MostCrowdedJunction(String id) {
        super(id);
    }

    protected IncomingRoad createIncomingRoadQueue(Road road) { ... }

    protected void switchLights() { ... }

    protected void fillReportDetails(IniSection is) {

    }
}
```

This will be called by class Junction to create a queue for a road, now it will create IncomingRoadWithTimSlice instead of IncomingRoad, its light should be red

Override the method for filling in the report in order to add the "type" key-value to the report.

We override the switchLights behaviour of Junction

1. If all lights are red (the first time), then turn the light of the road with most vehicles in the queue to green, and set its time slice to the max(_queue.size(),1). Otherwise;
2. If the current road with green light has consumed its time slice, then turn it to red and turn the one with most vehicles in the queue (that is different from last one) to green with time slice max(_queue.size(),1)

IMPORTANT: when retrieving an element from the list _roads, it will be of type IncomingRoad, cast to InComingRoadWithTimeSlice in order to access/set the different information ...

Handling Errors

Create a class SimulatorError that extends RunTimeException and use it everywhere to report errors. You MUST handle ALL possible errors.

Packaging

It is recommended that you organise the classes in packages as follows:

- control
- ini
- launcher
- model

Controller and all EventBuilder classes

This is the INI parser we provided you

The MAIN class

The TrafficSimulator class, all simulated objects classes, and all Event classes