Imperial College of Science, Technology and Medicine	University of London
Computer Science (CS) / Software Engineering (SE)	BEng and MEng Examinations Part I
Department of Computing	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations
and is a required part of the degree assessment.
Laboratory work must be handed in for marking by the due date.
Late submissions may not be marked.

Exercise: 14	Working: Individual
Title: Discrete Event Simulation	
Issue date: 16th February 2004	Due date: 23rd February 2004
System: linux	Language: Java

Aims

- To provide practice at defining and using abstract classes
- To build a class that implements a simple interface
- To contruct a simple class library that can be re-used in many different applications
- To introduce the concept of discrete-event simulation
- To experiment with static class variables

The problem

Discrete-event simulation is a way of modelling the evolution of systems over time. It is a very general and powerful technique that is used extensively to study behavioural and performance aspects of real-world systems, e.g. manufacturing systems, computer networks, queueing systems, transportation etc. You will have the opportunity to study these issues later in the degree course.

In this exercise you a required to build a small package of classes that collectively will manage a discrete-event simulation. The classes have been carefully designed to provide a high degree of flexibility (in terms of re-use) whilst at the same time being small and simple. There are *many* ways to build such a simulation library in Java. The one described here has been designed specifically to exercise certain features of the Java language; it has not been engineered to minimise execution time, for example.

Discrete-event Simulation

Discrete-event simulation (DES) is based on the *scheduling* of *events* in time order. Associated with every simulation is a (possibly empty) set of variables, which represent the state of the simulated system, together with other variables that may be required to control the simulation. These are typically implemented as instance variables and/or class variables in some Java class. Invoking an event will typically change one or more of the these variables and may also *schedule* one or more new events to happen at future time(s). At the heart of every DES is a method that traverses a list of events in time order, invoking each event in turn. Note that all times are *virtual*: the simulated passing of time has nothing to do with the passing of real time as the simulation executes. Time is just a Java **double**!

A good analogy is a standard pocket diary: the entries in the diary (lunch at 12-00, meeting at 2-00, opera at 5-00, etc.) are the events and the diary itself is the event list. The diary is "processed" in time order and one event may schedule another, for example in the meeting at 2-00 you may schedule the next meeting, e.g. for the same time the following week. The "state" in this case is essentially the state of "your life"(!) For example, event "'My birthday today" will increase your age by one year and "Hairdresser at 9-00" may very well change the length, and possibly colour, of your hair!

Requirements

An important part of the exercise is to come up with a library of classes that can be used to manage a wide range of discrete-event simulations. The basic requirements are:

- It should be possible to build many different types of event, although each event should share important characteristics, e.g. the ability to be scheduled at a specified time
- Events should be able to take an arbitrary set of parameters, in addition to required parameters, e.g. the time at which the event should happen
- The simulation class library should not depend in any way on the system being modelled, i.e. it should be possible to simulate any system, at least in principle

Submit by Monday 23rd February 2004

What to Do

You're going to build a package called **simulation** which should contain the classes **Simulation** and **Event** as described below:

Class Simulation

This should contain two (private) instance variables: one a double that represents virtual time, in arbitrary time units, and the other the event list. You should provide a method, e.g. **now()**, that returns the value of current time.

To manage the event list you need two additional methods: one that adds an event to the list (**void schedule(Event e)**, say) and one that controls the processing of the event list

in time order (**void simulate()**, say). The latter should repeatedly perform the following actions:

- Remove the next event from the front of the event list
- Update the time to the invocation time of that event
- Invoke the event

A method is invoked by calling the event's **invoke()** method (see below).

The Event List

The event list can be implemented in many ways – in this exercise we recommend you use a predefined class called **TreeSet** that implements ordered sets; the name comes from the fact that it uses trees to implement sets, although this is of no concern to you. **TreeSet** is defined in class **java.util** and can be imported by typing:

```
import java.util.TreeSet ;
```

at the top of the file. The **TreeSet**s class contains two methods that will prove useful when defining the **Simulation** class: **void add(Object x)** that adds an object to a **TreeSet** and **Object first()** that returns the first object in the **TreeSet** according to a specified ordering.

TreeSets can contain any type of object *provided* they can be ordered in a meaningful way. This is enforced by requiring the objects to implement the interface **Comparable**. This is defined in **java.lang** and contains a single method definition **int compareTo(Object x)** that returns -1 if the object is "smaller than" **x**, 0 of it is "equal to" **x** and 1 if it is "larger than" **x**.[1] The ordering on events is based on their time of occurrence: one event is "smaller than" another if it is to be invoked earlier. Here, the objects will be events, so the **Event** class (below) must be defined to **implement Comparable**.

Note that when you remove an object from a **TreeSet** the result is an **Object**, i.e. the most general object type. This means that when you remove an event from the list it too will be identified as being of type **Object** even though you know it's of type **Event**! You therefore have to *cast* the **Object** into an **Event**, for example:

```
Event e = (Event) diary.first() ;
```

where **diary** here is the name of the event list. Compare this with Haskell which supports proper polymorphic data types.[2]

Termination

All that remains is to determine when to terminate the **simulate()** loop above. We need to be able to control termination in a completely general way. We do this by making class **Simulation** an abstract class, with an abstract boolean method **stop()** that must be defined for each simulation program, i.e. for each subclass of **Simulation**. In this way the user of the

[1] Some of Java's library classes are highly questionably designed. The function **int compareTo(Object o)**, for example, is particularly awful: comparison functions should not return integers! Be warned that the Java class libraries do *not* always represent best practice in design.

[2] This *is* being fixed! A version of Java called GJ (Generic Java) essentially allows parameterised types in the same spirit as Haskell.

Simulation class can choose any termination criteria, e.g. based on time (execute for 100 simulated days) or on the simulation state (simulate the manufacturing of 5000 components). Of course, it may be that the event list becomes empty *before* the **stop** method delivers true – you need to check this separately and terminate the **simulate()** loop accordingly. Note also that updating the current time may cause the **stop** method to deliver true, when previously it delivered false. You should check this before invoking each event.

Class Event

The **Event** superclass should be defined to be abstract with an abstract void method **invoke()** that must be defined in each subclass. This is the method called by the **simulate** method above when the event reaches the front of the event list. The time at which the event should be invoked needs to be recorded in an instance variable visible to the simulation package but not elsewhere. You want to ensure that the current simulated time can only be accessed from outside the package by calling the method **now()**.

The **Event** constructor should take two arguments. The first (a **double**) is the *interval* of time that must pass *before* the event is to be invoked, i.e. it is *not* an absolute time. The second argument (a **Simulation**) should be the "current simulation". This needs to be passed to all events so that they have access to both the current time and the event list, in case they need to schedule one or more future events. The **Event** constructor should set the invoke time and current simulation instance variables appropriately and add itself to the event list associated with that simulation.

The **Event** class should implement the interface **Comparable** so that events can be added to the **TreeSet** representing the event list. This means that it must also contain a definition of the **compareTo** method described above. in order to define the ordering on events.

Application Programs

With the **simulation** package complete, you can now build discrete-event simulations by importing its various classes. Every simulation has the following general structure:

- A separate class for each event – each of these will be a subclass of the abstract superclass **Event** but may take additional parameters, for instance the current simulation state.
- A subclass of the **Simulation** class, **mySim** for example, with the **stop()** method suitably defined. To kick-start a simulation you need to schedule one or more initial events and then invoke the **simulate()** method in the superclass.
- An optional set of state/control variables—note that some simulations may not require this. These variables should be instance variables in some class, e.g. **State**; later you will be invited to experiment by making them static class variables instead. An instance of the state class should be created when the simulation is initialised and passed to each event (i.e. each **Event** subclass) as an additional parameter.

You can arrange it so that the **Simulation** subclass contains the 'main' method which sets the simulation running by creating an instance of the simulation class, e.g. **mySim**. Note that you could put the main method in a separate class, but for this exercise you should

include it here. This looks a little odd at first since the main method creates an instance of a class in which it is defined! This is perfectly legal, however, as it is a static method.

Experiments

You should build the following simulation examples in order to test your library. The first has been provided for you in the classes **Print3** and **Print**. These should get you started and can be obtained by typing **exercise 14**. You can constuct an infinity of other simulation models using the class library, however; see the competition below.

Simple Fixed Schedule

Schedule a fixed number of instances of a Print event that prints a message when it is invoked. The Print event should take an integer parameter, in addition to the time parameter, that should be displayed as part of the message. For example, given the following schedule (note that `this` here refers to the "current simulation"):

```
new Print( 1, 7.2, this ) ;
new Print( 2, 11.6, this ) ;
new Print( 3, 2.9, this ) ;
```

and defining the **stop** method so that it always returns **false**, the program should print the following:

```
Event 3 invoked at time 2.9
Event 1 invoked at time 7.2
Event 2 invoked at time 11.6
```

You can experiment with the **stop** method, e.g. making it always deliver false, making it deliver true some time before the last **Print** event is invoked, making it deliver true before the first **Print** event has been invoked etc. Note: no state/control variables are required in this example.

Infinite Ticks

Build a simulation class **Ticks** that initially schedules an instance of a **Tick** event at time 1. The **Tick** event should print the current time and then schedule a new **Tick** event at time t+1 where t is the current time. The program should stop after a specified time – a parameter of the **Ticks** constructor, which should be given as a command line argument. As an example, given a stopping time of 10.0 your program should produce the following sequence of messages:

```
Tick at: 1.0
Tick at: 2.0
Tick at: 3.0
Tick at: 4.0
Tick at: 5.0
Tick at: 6.0
Tick at: 7.0
Tick at: 8.0
Tick at: 9.0
```

Simple Queueing System

Schedule an initial instance of an **Arrival** event. The **Arrival** event should change the state of a queue, as represented by a single integer state variable representing the population of the queue. This queue length should be an instance variable in a separate class, e.g. **State**. The arrival event should also schedule the next arrival event, similar to the **Tick** event above; it should also schedule a **Departure** event if the arriving customer has joined an empty queue and therefore moved straight into service. A **Departure** event decrements the queue population and checks whether there is another customer in the queue. If so, it must schedule *that* customer's **Departure** event accordingly, as it will by now have hit the front of the queue. Each event should print a message, reporting the event type, the time of the event and the current population of the queue.

You need to make assumptions about the service times and inter-arrival times. To make the simulation more interesting you should use a *random number generator* to generate the inter-arrival times. Java has a predefined class **Random** that can be used to do this. To load this type

```
import java.util.Random ;
```

at the top of your simulation class file. The constructor for class **Random** takes a *seed* value which is used to initialise the generator. The seed is of type **long** — a double-precision variant of the **int** type. If two generators are initialised with the same seed they will produce identical sequences of random numbers; this is useful when testing your program. If **gen** is the name of the random number generator object then method **gen.nextDouble()** can be used to generate a random **double** between 0 and 1. This can be easily scaled, but here we will use the numbers directly as inter-arrival times. The obvious place to put the random number generator is in the same (**State**) class that stores the state variable for the queue length.

The service times should be assumed to be fixed at 0.25. In this way the inter-arrival times are twice as large as the service times, on average.

The constructor for the queue simulation should take the stopping time and random number seed as parameters. You should provide these as command line arguments to your main program class **SSQSim**. For example, given a stopping time of 4.0 and a seed of 1987281099, execution of the simulation should produce the following trace:

```
Arrival at: 0.7777800058166073, new population = 1
Arrival at: 0.8532981371697105, new population = 2
Departure at: 1.0277800058166073, new population = 1
Departure at: 1.2777800058166073, new population = 0
Arrival at: 1.717518578826721, new population = 1
Arrival at: 1.939347130653938, new population = 2
Departure at: 1.967518578826721, new population = 1
Arrival at: 1.9765137857679331, new population = 2
Departure at: 2.217518578826721, new population = 1
Departure at: 2.467518578826721, new population = 0
Arrival at: 2.828433719186867, new population = 1
Departure at: 3.078433719186867, new population = 0
Arrival at: 3.157263945340226, new population = 1
```

```
Arrival at: 3.1874794285156836, new population = 2
Departure at: 3.407263945340226, new population = 1
Arrival at: 3.4427516133419704, new population = 2
Arrival at: 3.5976721621551273, new population = 3
Departure at: 3.657263945340226, new population = 2
Departure at: 3.907263945340226, new population = 1
SIMULATION COMPLETE
```

You should be able to reproduce this exactly!

Optional Extra 1 – Unassessed

Notice that if we only build one simulation we only need one set of state/control variables. Rather than have a **State** class with instance variables you can alternatively have one with static class variables. In this way you don't need to create an instance of the class and carry it around the various events - each event can refer to the variables directly through the class name. Try solving the problem both ways - this will reinforce your understanding of static variables.

Optional Extra 2 – Unassessed

In practice people build simulations because they want to measure something, for example efficiency, cost, load, time etc. Try augmenting the simulation above by measuring the average number of jobs in the queue (you need to think carefully about how to do this!), and/or the proportion of time that the server is busy ("utilisation"). Having done this, vary the mean inter-arrival times and service times and explore the effect this has on the various measurements. Can you see any patterns?!

Optional Extra 3 – Unassessed

As a simlpe extension to the single-server queue model, add another variable to the **State** class that keeps track of the total number of service completions (i.e. invocations of the departure event). Now change the program so that it terminates after a specified number of completions rather than a specified time.

Competition

In practice computer simulation is used extensively to understand the behaviour of both real and imaginary systems. Notice that your simulation package is powerful enough to describe any discrete-event system[3] so you can used to build arbitrarily grandiose models. You might like to browse the web for inspirational ideas. This year we're holding a competition for the "most interesting" simulation model with the prize being a bottle of champagne, or book tokens to the same value. Why not have a go?! The competition will be judged by a panel of modelling experts who will take into account the scope of the model and the way you have used the model to understand the behaviour of the system being modelled. Competition

[3] Actually, we haven't included a method for *removing* and event from the event list—this makes some simulations much easier to write. You could add this if you need it.

entries can be submitted by typing submit DES; the deadline is 15/3/04. If you enter the competition you should include all your code in a single file **DESCompetition.java** using *inner classes* to ensure this.

Submission

- Submit your program by copying the files **Ticks.java**, **Tick.java**, **SSQSim.java**, **Arrival.java**, **Departure.java** and **State.java** into the **simulation** subdirectory and then *cd* into the **simulation** subdirectory (which should contain your files **Simulation.java** and **Event.java**) and then type: **submit 14**. Check your mail to ensure that you have submitted your program files correctly.

Assessment

`Simulation.java`	1.00
`Event.java`	1.00
`Ticks.java`	0.25
`Tick.java`	0.25
`SSQSim.java`	0.75
`Arrival.java`	0.75
`Departure.java`	0.75
`State.java`	0.25
`Design, style and readability`	5
`Total`	10