Java Discrete Event Simulation

COMP40009 – Computing Practical 1

30th January – 3rd February 2023

Aims

- To introduce the concept of *discrete-event simulation*.
- To construct a simple class library that can be re-used in many different applications.
- To recall the different visibility modifiers Java offers, and to use them appropriately in class, method and field definitions.
- To gain familiarity with defining and using *interfaces*.
- To practice defining and using *type parameters* on classes.
- To further practice using classes from the standard Java Collections libraries.

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.

This exercise asks you to build a small package of classes that collectively will manage a discrete-event simulation. You are provided with one simple program that uses your library, and are asked to implement a further two.

This specification will outline a *suggested* design, which has 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. You are, as always, free (and encouraged) to experiment with other designs, though please try to keep the names of the programs in your submissions the same as specified for auto-testing purposes.

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 ending of the simulation. 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.

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!

Simulations

Once you have created a library for managing simulations, you are asked to use it to create three different types of simulation.

Simple Fixed Schedule (Print3)

You are provided with a program for implementing a simple fixed schedule. The events in this simulation simply print out an assigned number at a given virtual time. The simulation is stateless and in theory can run forever, however only a finite number of events will be scheduled at the start, so termination is guaranteed. The sample program schedules three such events and then simulates them running.

Ticks

A slightly more complicated simulation is one where the events can themselves cause further events to happen. The second simulation you are asked to model is a finitely ticking clock, where each tick event prints out the current time, *and* causes a further tick to happen one second later. This simulation also has a time limit, and will stop running when that amount of time has elapsed.

Single Queueing System

Many real-life processes can be modelled by a queue like system (or a combination of multiple queues). As a simple example, consider a shop with a single till. One event in such a system would be a customer arriving in the queue for the till. Another event would be a customer departing the head of the queue for the till once the till is free. However it takes time for the customer at the till to pay, which is called the *service time.* Many customers can join the queue, and we call the time between customers joining the *inter-arrival* time.

You are asked to create a simulation modelling such a system. The current length of the queue would be modelled as a variable of the simulation. An arrival event will increase the size of the queue and print out that an arrival has happened. In addition, an arrival event will schedule the next arrival event according to the inter-arrival time. If the arrival occurs in an empty queue, then that arrival's subsequent departure is also scheduled according to the service time. Departure events decrement the size of the queue, and print out that a departure has happened. In addition, if the queue is not empty then they must schedule the next departure, according to the service time.

For the single queueing system simulation, you should assume the inter-arrival times are to be uniformly distributed between 0 (inclusive) and 1 (exclusive), and the service time fixed at 0.25. The system is initialized with a single arrival, and runs for a configurable amount of time.

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.

What to Do

Setting up the skeleton project

Clone your exercise repository (remembering to replace *login* with your login) with:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab2223_spring/javadiscreteeventsimulation_login.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[1]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

```
7z x skel.7z -p2CA14D31
```

(*n.b., 2CA14D31 is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named “skel”, because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.

Part I - Suggested Library Classes and Design

This section and the two following outline some *suggested* library classes and design ideas you may wish to follow. You are free to add additional classes and packages under the `src` directory as you see fit.

However, please ensure that your three programs reside under the `src` directory in the classes `print3.Print3`, `ticks.Ticks` and `ssq.SingleServerQueue` and that they can understand the program arguments specified.

This suggested design is deliberately vague on the *visibility* of methods, fields, and classes. It is up to you to choose the correct modifiers (`public`, `protected`, none, or `private`).

[1] `https://www.7-zip.org/download.html`

Storing the Diary of Events

The diary of events to happen can be implemented in many ways. In this exercise we recommend you use the class `java.util.PriorityQueue`, which maintains a queue of objects.[2] Items are added to the queue using the `add` method, and the queue places them into their *natural ordering*. Items can then be removed in this order using the `poll` method.

The natural ordering is specified by the items implementing the interface `Comparable`.[3] `Comparable` requires the definition of the method `compareTo`. For a given call `x.compareTo(y)`, this method should return < 0 if `x` is "smaller than" `y`, 0 if they are "equal" , and > 0 if `x` is "larger than" `y`. You may find the static helper method `Double.compare(double d1, double d2)` helpful when implementing the `compareTo` method.

It is good practice to access Java Collections through their interfaces as opposed to their concrete implementations (in case you wish to change concrete implementation later). For `PriorityQueue` this would be the interface `java.util.Queue`.[4]. For example, if you wished to create a queue of elements of type `ScheduledEvent`, you would write:

```
Queue<ScheduledEvent> diary = new PriorityQueue<>();
```

Predictable Random numbers

The class `java.util.Random` should be used to generate the random inter-arrival times in the Single Queueing System program.[5]. The constructor of `Random` can be given a `long` *seed* value which makes the sequence of random numbers it returns *reproducible*.

Suggested Simulation Library Design

The simulation library should reside within a package `simulation`.

- interface `Event`. Instances of this interface will represent the events within a particular discrete-event simulation. This interface should feature a single method:
  - `void invoke(Simulation simulation)`, which is called when the event happens. This should use its `simulation` argument to access the simulation's variables and to schedule further events. In addition, implementations of this method will perform any other action as required by the simulation (such as printing to the terminal).
- class `ScheduledEvent`. This class stores an `Event` and the fixed time it has been scheduled at. It should implement `Comparable<ScheduledEvent>`, where the comparison method `compareTo` compares the times the two `ScheduledEvent`s have been scheduled at.
- class `Simulation`. This class stores the current virtual time (which should be represented as a `double`), and the diary of `ScheduledEvent`s. It then features *at least* the following methods:
  - `boolean stop()`, which subclasses of `Simulation` should override to return `true` if the stopping condition of their simulation has been reached.
  - `void schedule(Event e, double offset)`, which will schedule the event `e` to run after `offset` virtual time has elapsed from the current virtual time.

[2] http://docs.oracle.com/javase/7/docs/api/index.html?java/util/PriorityQueue.html
[3] http://docs.oracle.com/javase/7/docs/api/index.html?java/lang/Comparable.html
[4] http://docs.oracle.com/javase/7/docs/api/index.html?java/util/Queue.html
[5] http://docs.oracle.com/javase/7/docs/api/index.html?java/util/Random.html

– `void simulate()`, which runs the simulation. A simulation finishes when the diary of events becomes empty, or if the `stop()` method returns true. Running one step of the simulation involves getting the next event from the queue, updating the virtual time to that event's scheduled time and then (assuming we shouldn't `stop`), invoking the event.

The Applications

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

- A class that extends `Simulation`. This will have the state/control variables needed for the simulation as fields. It will also implement the `stop` method from `Simulation`.
- One or more classes that implement `Event`. These instances will then need to implement the `invoke` method as appropriate for the simulation.
- A program, (i.e. a `public static void main(String[] args)` within a `public` class), which has the appropriate name given below. You can arrange it so that the `Simulation` subclass contains the `main` method. This program should create an instance of the simulation, schedule any initial events and then start the simulation running using `simulate`.

package `print3`

Within this package you'll find the example Print3 simulation. If your design does not line up the interface it expects, feel free to alter the contents of this package. When the program `print3.Print3` is run it should output:

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

package `ticks`

You should create a program `ticks.Ticks` which models the Ticks simulation described earlier. The program should assume a single command line argument which is how long (in virtual time) to run the simulation for. If run for 10 seconds this program should output:

```
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
```

package `ssq`

Finally you should create a program `ssq.SingleServerQueue` which models the Single Server Queue simulation as described earlier. The program should accept two arguments, the first being a seed for the random number generator (which should be converted to a `long`) and the second an amount of time to execute for (a `double`).

Hint: you may want a method for increasing and decreasing the queue length. This will be helpful in part 2.

If this program is invoked with a seed value of 1987281099, and time of 4.0, this program should output:

```
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 will find it necessary in your events to use a cast from `Simulation` to `SingleServerQueue`. Why is this usually a bad idea?

Testing

This time, instead of JUnit tests, a small script (`runSimulations.sh`) is provided in your skeleton project, as is transcript of the expected output in `runSimulations_part1.txt`. This corresponds to the example outputs given above. Running the script should produce this output exactly. You can invoke this script in a terminal with:

```
% ./runSimulations.sh
```

If you wish to see if your output is identical to the sample transcript, you can *pipe* its output to the `diff` tool:

```
% ./runSimulations.sh | diff - runSimulations_part1.txt
```

Once you have finished part 1, you should commit your work with a message clearly identifying that this is the end of part 1 before starting part 2.

Part II - Suggested Simulation Library Design

In this part, you should modify your library so that the core classes are all parameterised by a type, `S`. This type represents the state of the simulation visible to the events. This should mean that casting is no longer necessary!

- `Event` should become `Event<S>`.
  - `void invoke(Simulation simulation)` should now be `void invoke(S simulation)`
- `ScheduledEvent` should now be `ScheduledEvent<S>`
- `Simulation` will become `Simulation<S>`. You will also need to add the following method (which you should use when calling invoke):
  - `S getState()`, which subclasses should override to return the value that the events of the simulation need when invoked. In general the subclasses will return themselves.

The Applications

Once the **simulation** package is modified, you should modify the earlier discrete-event simulations:

- The classes which extend `Simulation` should specify themselves as `Simulation`'s type parameter. Implement the `getState` method from `Simulation` (this method's implementation will be `return this;`).
- The classes which implement `Event` will specify the name of the subclass of `Simulation` as the type parameter to `Event`. Now the `invoke` method takes an argument of type `S`, you should no longer require casting.

Mean Queue Length

In practice people build simulations because they want to measure something, for example efficiency, cost, load, time etc. You should now augment the simple server queue simulation by measuring the average length (population below) of the queue. The simplest way of doing this is to modify the methods of your `SingleServerQueue` which change the queue length to also record the data needed to calculate the mean queue length. You will then need to implement a `getMeanQueueLength` method.

If this program is invoked with a seed value of 1987281099, and time of 4.0, this program should output:

```
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 - the mean queue length was 0.8825706991365345
```

Having done this, an interesting exercise would be to vary the mean inter-arrival times and service times and explore the effect this has on the various measurements. Can you see any patterns?!

Submission

As usual, use `git add`, `git commit` and `git push` to add and send your created **.java** files to the lab server.

Then, log into the gitlab server `https://gitlab.doc.ic.ac.uk`, and click through to your `javadiscreteeventsimulation_<login>` repository. If you view the `Commits` tab, you will see a list of the different versions of your work that you have pushed. Select the right version of your code and ensure it is working before submitting to CATe.

Suggested Extension

As an extension to the single-server queue model, add another variable that keeps track of the total number of service completions (i.e. invocations of the departure event). Create a new program that terminates after a specified number of completions, rather than a specified time.

Assessment

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```