The University of Melbourne
School of Computing and Information Systems
SWEN90004 Modelling Complex Software Systems

Assignment 1a, 2017

Released: 2 March, 2018. Deadline: 17:00, 21 March, 2018

Objectives

To provide programming practice in a monitor-oriented concurrent programming language and to get a better understanding of safety and liveness issues.

Background and context

There are two parts to Assignment 1 (a and b), each worth 10% of your final mark. This first part of the assignment deals with programming threads in Java. Your task is to implement a simulator of the transport network in a mountain park.

The system to simulate

Deep in the mountains of Concurrencia lies a cluster of historic villages. The villages are perched high in the mountains and accessible from the outside world by a cable car connecting them to the valley below. Travel between the villages is by means of mountain railway. The cable car terminus and each of the villages are connected to their two neighbouring in a ring topology (see Figure 1).

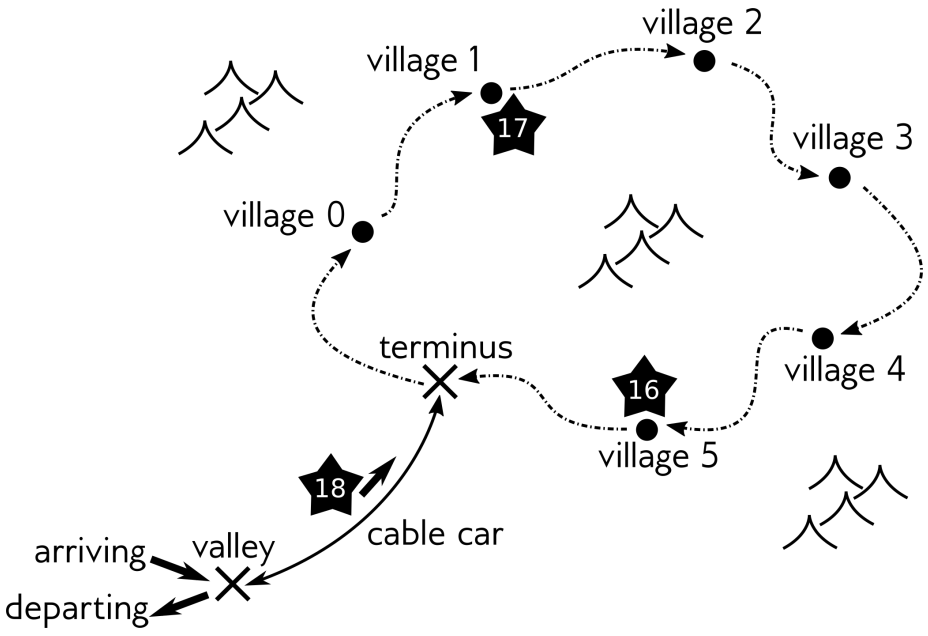

Figure 1: The mountain villages of Concurrencia, showing the cable car connecting them to valley below and the railway lines between them (with arrow indicating the direction in which tour groups travel). Numbered stars indicate tour groups; see text and example trace for further description.

The villages of Concurrencia are a popular destination for tour groups, who ride the cable car up from the valley and spend some time (and money!) visiting each of the villages. In order to preserve the

peaceful mountain atmosphere, the council has ruled that only one tour group is allowed to be present in each village at a time. This rule is strictly enforced. If a village is currently occupied by a tour group, no other group is allowed to enter that village until the first group leaves.

Tour groups arrive in the valley at random times (we are not concerned with how they get there). Each group is then transported into the mountain park via the cable car, which carries one group at a time from the valley (bottom) to the terminus (top). Tour groups then travel on to the first village (0) by mountain railway, after which they visit each successive village (1, 2, ...) in order, before returning to the cable car terminus, from where they are carried by the cable car back down to the valley (at which point we are no longer concerned with them).

Each railway line (ie, between the terminus and the first village, between each pair of villages, and between the last village and the terminus) is served by a single train that shuttles backwards and forwards between the same pair of destinations continuously. To ensure that there is only ever one tour group in a village at a time, all tour groups travel in a clockwise direction. That is, each train will carry a single tour group while travelling clockwise, and then return (empty) to collect the next tour group.

For simplicity, we will treat the cable car (including terminus and valley ends) as a single location that has space for a single group to occupy, and a flag indicating which end it is currently located at. So no tour group may enter the valley or the terminus ends of the cable car if there is another group currently on the cable car.

Figure 1 shows a snapshot of the system at a given point in time. Tour group 18 is currently occupying the cable car, ascending into the mountains. Group 17 is occupying village 1 and group 16 is occupying village 5, where it is waiting for the cable car to become free, so that it can enter it to descend from the mountains and depart. After group 18 leaves the cable car (for village 0), group 16 will enter and descend. If, instead, there was no tour group 16 waiting to descend, but more tour groups arriving in the valley, it would be necessary for the cable car to descend (empty) to collect them. Thus, the cable car operator will occasionally send the empty cable car up or down the mountain (at random intervals).

Your tasks

Your task is to implement a simulator for the mountain transport system. It should be suitable parameterised such that the timing assumptions can be varied and the number of villages can be varied. The simulator should produce a trace of events matching that below. Note that after the event "`[18] enters the cable car to go up`" we have the situation illustrated in Figure 1.

```
  :
[17] enters village 1
[18] enters cable car to go up
[18] leaves the cable car
[16] enters cable car to go down
[16] departs
[17] leaves village 1
[18] enters village 0
[18] leaves village 0
[17] enters village 2
[17] leaves village 2
[18] enters village 1
[18] leaves village 1
[17] enters village 3
[17] leaves village 3
cable car ascends
[18] enters village 2
[18] leaves village 2
[17] enters village 4
[18] enters village 3
[17] leaves village 4
[18] leaves village 3
[17] enters village 5
[17] leaves village 5
[18] enters village 4
[18] leaves village 4
cable car descends
[19] enters cable car to go up
[19] leaves the cable car
[17] enters cable car to go down
[17] departs
cable car ascends
[18] enters village 5
[18] leaves village 5
[19] enters village 0
[19] leaves village 0
[19] enters village 1
[19] leaves village 1
[18] enters cable car to go down
[18] departs
[20] enters cable car to go up
  :
```

A possible design and suggested components

In the Java context, it makes sense to think of each village as a monitor, and similarly to think of the cable car as a monitor. A possible set of active processes would then be:

`Producer:` Generates new tour groups that want to visit the mountain villages; hands them to the cable car, subject to the constraint that the cable car is unoccupied and currently located in the valley. The times between arrivals should vary.

`Consumer:` Removes tour groups who have completed their visit, once they are back in the valley; that is, the cable car is occupied and in the valley. The times between departures should vary.

`Train:` Picks up tour groups from one location (either the cable car or a village) and delivers them to the next location (the next village or the cable car). To collect a group from the cable car it must be occupied and currently located at the terminus. To deliver a group to the cable car, it must be unoccupied and currently located at the terminus.

`Operator:` Inspects the cable car at random intervals and, once it is unoccupied, sends it from the valley to the terminus, or from the terminus to the valley.

You do not need to follow this design sketch. We have made some scaffolding available on the LMS that does assume the outline described above. The components we have provided are:

`Producer.java` and `Consumer.java`: as described above.

`Group.java:` Tour groups can be generated as instances of this class.

`Params.java:` A class which, for convenience, gathers together various system-wide parameters, including time intervals.

`Main.java:` The overall driver of the simulation. Note that this won't compile until you have defined some additional classes.

Timing parameters

The class `Params.java` assumes the following time parameters are of interest:

`arrivalLapse():` The time lapsed between two successive groups arriving (varies).

`departureLapse():` The time lapsed between two successive groups departing (varies).

`operateLapse():` The time lapsed between two successive ascent/descent operations (varies).

`OPERATE_TIME:` The time it takes for the cable car to ascend or descend.

`JOURNEY_TIME:` The time it takes a train to take a group from its source to its destination.

Varying these parameters will give you different system behaviours. You are encouraged to experiment with the settings.

Extension tasks

A number of optional extensions and experiments are suggested, ranging from the simple to the more ambitious, that you may wish to explore once you have completed the main assignment. Note that these are for your interest only; they do not contribute to your grade, though you may discuss them in your reflection text if you wish.

1. A new "express" railway line is added between village $k/4$ and village $3k/4$ (where $k \geq 8$ is the total number of villages). There are now two types of tour groups, those taking the "full" tour, visiting every village, and those taking the "short" tour using the express line.

2. A new "branch" railway line is added from village $k/2$ up to a nearby mountain peak. Some tour groups may elect to add on this side trip to their tour.

3. Rather than sampling the arrival and departure times from a Uniform distribution, a more realistic assumption may be an exponential or Poisson distribution.

4. As an alternative to the trace output, it would be helpful to have a sequence of (textual) snapshots of the whole system, showing the cable car, each village and the current location of tour groups (and any other relevant information) after each change.

Procedure and assessment

- The project should be done by students individually.
- Late submissions will attract a penalty of 1 mark for every *calendar* day it is late. If you have a reason that you require an extension, email Nic *well before the due date* to discuss this.
- You should submit a single zip file via LMS. The zip file should include:

    1. All Java source files needed to create a file called `Main.class`, such that "`javac *.java`" will generate `Main.class`, and "`java Main`" will start the simulator.
    2. a plain text file `reflection.txt` should contain 300–500 words evaluating the success or otherwise of your solution, identifying critical design decisions or problems that arose, and summarising any insights from experimenting with the simulator. Please ensure that this is a *plain text* file; ie, not a `doc`, `docx`, `rtf`, or other file type that requires specific software to read.

  All source files and your text file should contain, near the top of the file, your name and student number.

- We encourage the use of the LMS's discussion board for discussions about the project. **However, all submitted work must be your own individual work.**
- This project counts for 10 of the 40 marks allocated to project work in this subject. Marks will be awarded according to the following guidelines:

Criterion	**Description**	**Marks**
Understanding	The submitted code is evidence of a deep understanding of concurrent programming concepts and principles.	3 marks
Correctness	The code runs and generates output that is consistent with the specification.	2 marks
Design	The code is well designed, potentially extensible, and shows understanding of concurrent programming concepts and principles.	2 marks
Structure & style	The code is well structured, readable, adheres to the code format rules (Appendix A), and in particular is well commented and explained.	2 marks
Reflection	The reflection document demonstrates engagement with the project.	1 marks
Total		10 marks

Nic Geard
22 February 2018

A Code format rules

Your implementation must adhere with the following simple code format rules:

- Every Java class must contain a comment indicating its purpose.
- Every method must contain a comment at the beginning explaining its behaviour. In particular, any assumptions should be clearly stated.
- Constants, class, and instance variables must be documented.
- Variable names must meaningful.
- Significant blocks of code must be commented.

  However, not every statement in a program needs to be commented. Just as you can write too few comments, it is possible to write too many comments.
- Program blocks appearing in if-statements, while-statements, etc., must be indented consistently. They can be indented using tabs or spaces, and can be indented 2, 4, or 8 spaces, as long as it is done consistently.
- Each line must contain no more than 80 characters.