The University of Melbourne
Department of Computing and Information Systems
SWEN40004 Modelling Complex Software Systems

Assignment 2a, 2015

Released: 29 April. Deadline: 15 May at 17:00

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 2. This first part, 2a, is worth 10% of your final mark; the second part, 2b, will be worth 15%.

This first part of the assignment deals with programming threads in Java. Your task is to implement (or rather complete) and test a simulator for a baggage handling system.

The system to simulate

The system to be built is a simulator of a baggage handling facility. The simulated part of the system is responsible for moving bags from check-in to the area where the baggage is loaded onto carts, as well as scanning and cleaning bags that are deemed suspicious. (We ignore the fact that bags have to be distributed according to which planes they are destined for.) Figure 1 shows a screen shot of a visualisation of the baggage handling simulator.

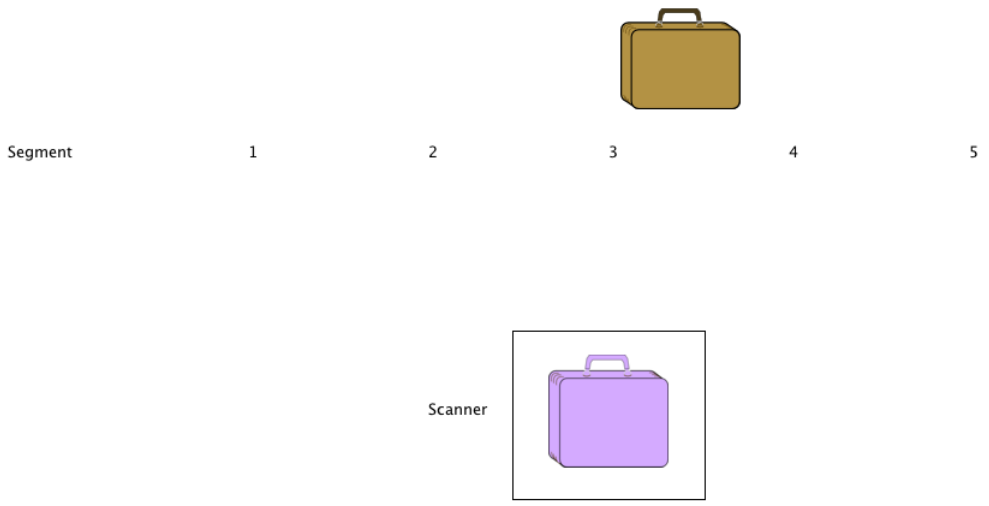

Figure 1: The simulator's visualisation of the belt and the scanner

The belt has five segments. The flow of control in the baggage area is that bags are placed onto the belt at the left end (segment 1). A bag can be clean or suspicious—the latter occurs if the passenger matches some criteria at check-in. If a bag is clean, it travels along the belt (to segment 5) without any problem, and it is taken off the belt there. If a bag is suspicious, it needs to be removed and scanned by a scanner. A *sensor* placed at segment 3 will identify the suspicious bag, and then the bag is moved to the scanner. The scanner can scan only one bag at a time. In our first model we will assume that the scanned bag is returned to segment 3 of the belt, but only when that segment is free. We will assume that all bags are okay for flying—the scanner should mark them all as clean.

To the left of the belt, there is a *producer* that loads bags onto the belt at random times. At the end of the belt, there is a *consumer* that takes the bags off the belt and (outside the simulated system) loads them onto a cart, and later onto a plane. A *robot* handles the transfer of bags between the belt and the scanner.

A partial solution

A partial simulator has been implemented, and the source code can be found on the LMS (in file `sim.zip`). You should study it carefully, compile it and run it. It is flawed, because the sensor, the scanner, and the robot, have not been implemented. Hence suspicious bags make it all the way to the end of the belt without being checked.

Your tasks

Your task is to implement a better simulator. Just like the initial, flawed, simulator, it should produce a trace of the important events (to standard output) and ideally, it should also visualise the working system (just as the flawed simulator does). There is no expectation that the visualisation will be fancy, and it is acceptable to produce only the trace.

In fact, you are asked to provide *two* solutions in this project.

Solution 2a1: First assume that bags have to go back onto the belt's segment 3, when the scanner has finished with them. This is not an ideal solution, and the only reason we want to see it implemented is that Assignment 2b will make use of it.

Solution 2a2: As a second solution, introduce an additional short belt, to take bags from the scanner to the handler (the Consumer) at the end of the main belt. This belt needs just two segments. The Consumer will need to take bags off both belts.

You should implement the new components (sensor, scanner, robot) and whatever other components you may need, and update the provided code as necessary, for the two required solutions. You will need to submit code for both.

There are numerous ways the visualisation can be improved. It might be nice if the bags' identification number was displayed, it might be nice if there was a panel showing the robot holding a bag that it is transferring, and so on, but be warned that this kind of work can be very time consuming. Concentrate on design and correctness of the system first, and only then think about the graphical interface (if at all).

The scaffold code

Part of the project is to make sense of the provided code. The driver of the whole simulation is `Sim.java`. Most other classes should be easy to understand, and their names reflect the categories (bags, belts, and so on) that they represent. `Animation.java` is a view class with tedious code for the visualisation part; this is the least transparent component.

Procedure and assessment

The project should be done by students individually. On the LMS you will find a zip file containing the scaffolding to start from.

The submission deadline is Friday 15 May at 17:00. A late submission will attract a penalty of 1.5 marks for every calendar day it is late. If you have a reason that you require an extension, email Harald *well before the due date* to discuss this. Submit a single zip file via LMS. The file should include two separate folders (or Unix directories), called `2a1` and `2a2`, and a text file called `reflection.txt`. The folder `2a1` should be a complete and self-contained solution for task 2a1, and similarly for `2a2`. Each should include:

- All Java source files needed to create a file called `Sim.class`, such that '`java Sim`' will start the simulator.
- A makefile that will generate `Sim.class` (it may be very simple; perhaps just containing the action "`javac *.java`").

The file `reflection.txt` should contain some 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.

We encourage the use of the LMS's discussion board for discussions about the project. However, all submitted work is to be your own individual work.

This project counts for 10 of the 50 marks allocated to project work in this subject. Marks will be awarded according to the following guidelines:

Criterion	**Description**	**Marks**
Correctness	The code runs and generates output that is consistent with the specification.	4 marks
Design	The code is well designed, potentially extensible, and shows understanding of concurrent programming concepts and principles.	3 marks
Structure & style	The code is well structured and readable.	1 marks
Layout	The code adheres to the code format rules (Appendix A) and in particular is well commented and explained.	1 marks
Reflection	The reflection document demonstrates engagement with the project.	1 marks
Total		10 marks

The 4 marks for correctness are spread evenly over sub-tasks 2a1 and 2a2.

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 should contain no more than 80 characters.

Harald Søndergaard
28 April 2015