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

# Assignment 1a, 2017

Released: 6 March. Deadline: 17:00, 24 March

## 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. This first part, 1a, is worth 10% of your final mark; the second part, 1b, 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 bicycle quality control system.

## The system to simulate

The system to be built is a simulator of a quality control station in a bicycle manufacturing factory. The purpose of the quality control station is to check that bicycles are not *defective* before they are packaged and shipped to distributors and retailers (Figure 1). The simulated part of the system is responsible for moving bicycles through the quality control station on a conveyor belt. Prior to arrival, bicycles have previously been subjected to an initial check, which has *tagged* some bikes for a more detailed quality control inspection. Tagged bicycles are removed from conveyor belt and subjected to a more thorough inspection. After inspection, tags are removed from bicycles found to be non-defective. After leaving ths quality control station, bicycles which are not tagged will be packed and shipped, while those that are tagged will be recycled.

We will assume that the initial check is too conservative. It correctly identifies (and tags) all defective bicycles, but also incorrectly tags some non-defective bicycles. The more thorough inspection in the quality control station is intended to catch these errors, and we assume that it does so perfectly; that is, it always removes the tags from non-defective bicycles and always leaves the tags on defective bicycles.

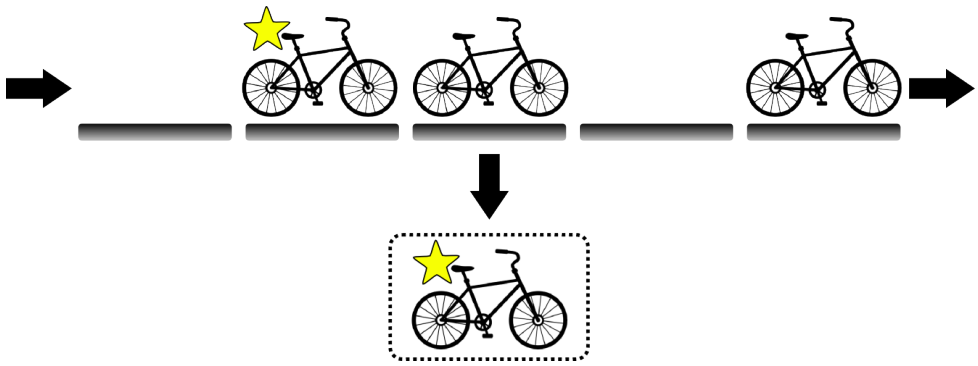

Figure 1: The quality control station to be simulated. Prior to arriving, bicycles have undergone a quick check, which has tagged some (stars) for more thorough inspection (dotted box). After leaving the quality control station, bikes will be taken off the conveyor belt for packing and shipping, unless they remain tagged as defective, in which case they will be taken off the conveyor belt and recycled.

The conveyor belt has five segments, each of which can hold a single bicycle. The flow of control in the quality control station is that bicycles are placed onto the belt at the left end (segment 1). Some bicycles are tagged for quality control inspection. If a bicycle is not tagged, it travels along the belt (to segment 5) and is taken off the belt there. If a bag is tagged, it needs to be removed and inspected by a quality control *inspector*. A *sensor* placed at segment 3 will identify a tagged bicycle, which is then moved to the inspector by a *robot* arm. The inspector can inspect only one bicycle at a time. After the quality control inspection, the inspector will remove the tags from bicycles found to be non-defective, and leave tags on those found to be defective.

In our first version of the model we will assume that all inspected bicycles (non-defective or defective) are returned to segment 3 of the belt by the robot, but only when that segment is free.

To the left of the belt, there is a *producer* that loads bicycles (some of which are tagged) onto the belt at random times. At the end of the belt, there is a *consumer* that takes the bicycles off the belt and (outside of the simulated system) packages and ships those that are non-defective, and recycles those that are defective.

## A partial solution

A partial simulator has been implemented, and the source code can be found on the LMS (in file `scaffold.zip`). You should study it carefully, compile it and run it. It is flawed, because the sensor, the inspector, and the robot, have not been implemented. Hence tagged bicycles make it all the way to the end of the belt without being inspected. This would mean that some non-defective bikes will be recycled unnecessarily.

## 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), such as arrival, movement, inspection and departure of bicycles. You should provide *two* solutions in this project.

**Solution 1:** First assume that all bicycles have to go back onto the belt's segment 3 when the inspector has finished with them. This is not an ideal solution, but we want to see it implemented because Assignment 1b will make use of it.

**Solution 2:** As a second solution, introduce an additional short belt, to take all bicycles (non-defective or defective) from the inspector to the handler (the Consumer) at the end of the main belt. This belt needs just two segments. The handler will need to take bicycles off *both* belts.

You should implement the new components (sensor, inspector, robot) and whatever other components you may need, and update the provided code as necessary, for the two required solutions. You will need to decide which of these components should be implemented as separate processes (ie, by extending BicycleHandlingThread) **You must submit code for both solutions.**

### 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 (bicycles, belts, and so on) that they represent. Suggested parameters and exception classes have been provided. Some of these are used in the scaffold code; others will be useful when implementing the required solutions.

## 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 24 March 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 Nic *well before the due date* to discuss this. Submit a single zip file via LMS. The zip file should include two separate folders (or directories), called `sol1` and `sol2`, and a text file called `reflection.txt`. The folder `sol1` should be a complete and self-contained solution for task Solution 1, and similarly for `sol2`. 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 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 Solutions 1 and 2.

# 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.