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

Assignment 1a, 2020

Released: Tuesday 10 March, 2020
Deadline: 23:59, Thursday 26 March, 2020

Objectives

The objectives of this assignment are: to convert a description of a system into a simulation model of that system; to implement that simulation in a shared memory concurrent programming language; to use the implemented simulation to explore the behaviour of a complex system; to gain a better understanding of safety and liveness issues in concurrent systems.

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 concurrent simulation of King Arthur and the Knights of the Round Table.

King Arthur was a legendary medieval leader in Britain, who ruled over a court of heroic knights. King Arthur and his knights would gather in the Great Hall and meet around a large Round Table. During these meetings, the knights would report back on their adventures, and King Arthur would allocate them new quests to complete.

The system to simulate

King Arthur and the knights interact according to a set of strict rules. Knights enter the Great Hall, discuss their adventures with one another, and sit at the Round Table. At some point, King Arthur enters the Great Hall. While King Arthur is in the Great Hall, knights may not enter or leave. Once all the knights who are present in the Great Hall are also seated at the Round Table, King Arthur starts a meeting. During a meeting, knights report back on and release their completed quest (if they have one) and acquire new quests. Once a knight has acquired a new quest, they may stand up from the Round Table, and spend more time discussing their adventures with one another. Once all knights have stood up from the Round Table, King Arthur ends the meeting. At some point after the meeting ends, King Arthur leaves the Great Hall, and the knights may leave the Great Hall to complete their quests. After completing their quests, the knights return to the Great Hall and the process begins anew.

Your tasks

Your first task is to implement a simulator for the system described above. It should be suitably parameterised such that the timing assumptions can be varied, and the number of knights can be varied. Use your simulator to explore the behaviour of the system, and identify any potential issues that may prevent the smooth operation of King Arthur and his court.

You may assume that:

- Knights may only have one quest (either completed or uncompleted) at a time.
- Knights who are on a quest will not return to the Great Hall until theat quest is complete.
- Having entered the Great Hall, a knight will not depart until they have acquired a quest.
- Not all knights need be present in the Great Hall before a meeting can start.
- A pair of agendas, for new and completed quests, is used to handle quests during a meeting.
- Quests may only be acquired and released during a meeting.

The simulator should produce a trace of events along the lines of that shown below. Prior to this trace, Knight 1 has acquired Quest 1 in a previous meeting. Note that, after King Arthur entered the Great Hall, the meeting could not begin until Knight 1 was seated. The meeting ended once no knights were left seated, but none of the knights could leave until King Arthur left the Great Hall.

```
              :
Knight 2 enters Great Hall.
Knight 3 enters Great Hall.
Knight 2 sits at the Round Table.
Knight 3 sits at the Round Table.
Knight 1 exits from Great Hall.
Knight 1 sets of to complete Quest 1!
Knight 1 completes Quest 1!
Knight 1 enters Great Hall.
King Arthur enters the Great Hall.
Knight 1 sits at the Round Table.
Meeting begins!
Knight 3 acquires Quest 2.
Knight 1 releases Quest 1.
Quest 3 added to New Agenda.
Knight 3 stands from the Round Table.
Knight 2 acquires Quest 3.
Quest 1 removed from Complete Agenda.
Knight 2 stands from the Round Table.
Quest 4 added to New Agenda.
Knight 1 acquires Quest 4.
Knight 1 stands from the Round Table.
Meeting ends!
King Arthur exits the Great Hall.
Quest 5 added to New Agenda.
Knight 1 exits from Great Hall.
Knight 1 sets of to complete Quest 4!
Knight 2 exits from Great Hall.
Knight 2 sets of to complete Quest 3!
              :
```

Your second task is to write a reflection of approximately 500 words that evaluates the success or otherwise of your implementation, identifies critical design decisions or problems, and summarises any insights you have gained from experimenting with the simulator, in particular any issues you identifed with the performance of the system, and potential solutions.

A possible design and suggested components

In the context of Java, it makes sense to think of the Great Hall and agendas (for new and completed quests) as monitors. A possible set of active processes could then be:

`Producer:` Generates new quests and adds them to the new quest agenda.

`Consumer:` Removes completed quests from the completed quest agenda.

`Knight:` As described above, enters and leaves the Great Hall, sits and stands, acquires new and releases completed quests, and completes quests.

`King Arthur:` Periodically enters and leaves the Great Hall, and starts and ends meetings with his knights.

We have made some scaffold code available on LMS that follows this outline described above. The components we have provided are:

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

`Quest.java`: Quests can be generated as instances of this class.

`Params.java`: A class which, for convenience, gathers together various system-wide parameters, including the number of knights and a number of timing parameters.

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

Note: You should complete this task using the Java concurrency constructs covered in class (ie, monitors, etc); you should not use the high level concurrency objects from the `concurrency` library.

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. A single directory named `src` containing all Java source files needed to create a file called `Main.class`, such that running "`javac *.java`" will generate `Main.class`, and running "`java Main`" will then start the simulator. If you use an IDE to implement your code, please ensure that it can be compiled and executed independently of the IDE (ie, by compiling and executing from a command line).
    2. A plain text file named `reflection.txt` containing your reflection. 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 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, following appropriate principles of abstraction, encapsulation, etc.	2 marks
Structure & style	The code is well structured, readable, adheres to the code format rules (Appendix A), and is well commented and explained.	2 marks
Reflection	The reflection document demonstrates engagement with the project.	1 marks
Total		10 marks

Nic Geard
10 March 2020

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.