Project 1

Software Project Day Simulation

October 23, 2016

Benjamin Meyers <bsm9339@rit.edu>
Asma Sattar <aas3799@rit.edu>

Table of Contents

1.0 Project Description

The goal of this project was to simulate a full work day at a software development company. This company has one Manager who oversees three Teams. Each Team has four members, with one member being the Team Lead. In total there are three Team Leads and nine Developers. All of these actors must act as separate threads and complete their daily activities, which can be found here.

1.1 Assumptions

1. We made the assumption that the size of the company is fixed. Therefore there is no need for command line arguments to set the number of employees, teams sizes, or number of managers; there will only ever be one Manager and twelve Employees (three Team Leads, nine Developers).
2. The Manager will take exactly ten minutes to answer every question. While this is not realistic, this saves us from writing some logic.
3. Meetings (regardless of type) will never go over the allotted time. While this is not realistic, this saves us from writing some logic.
4. The Manager and all Employees will never work more than eight hours a day. While this is not realistic, this saves us from writing some logic.

2.0 Design Decisions

The overall design for this simulation is rather simple, since we are required to have the classes outlined in Sections 2.1, 2.2, and 2.3. We added two more classes outlined in Section 2.4 and 2.5.

2.1 Main.java

The **Main** class contains the **main** function and is fairly straightforward. It creates the Manager and all of the Employees. It then assigns those Employees to Teams. We chose to implement two **CountDownLatches**[1] to ensure that meetings did not begin without all essential members present. We chose to use a **Semaphore**[2] to represent the conference room to ensure that it was restricted to a single Team at a time.

2.2 Employee.java

The **Employee** class is by far the most complex. The sensible thing to have done would have been to make **Employee** an interface and then create classes such as **Leader** and **Developer** to implement **Employee**. We decided against doing so because interfaces in Java are not as useful as they seem[3]. Because of this design decision, the Employee class is long and a bit hard to read. We believe that we provided enough documentation to make our implementation clear.

There is one function for handling the logic of answering questions, **askQuestion**. This function determines whether the asker of the question is a Team Lead or a generic Employee, then follows the respective question-asking process outlined in the Project Description. This function is ***synchronized*** to ensure that one Employee cannot ask multiple questions at once (wouldn't it be nice if we could do that in real life?).

[1] java.util.concurrent.CountDownLatch
[2] java.util.concurrent.Semaphore
[3]

2.3 Manager.java

The **Manager** is analogous to real managers in the sense that they are very busy. The Manager handles starting and ending meetings with Employees. This is completed using **CountDownLatches** (as explained in Section 2.1).

The function **answerQuestion** is ***synchronized*** to ensure that the Manager answers questions in the order that they are asked by following the **managerWaitList**. This function makes sure that questions are added to a **Queue**[4] and answered in the order they are received.

2.4 Team.java

For the sake of organization and making the **Employee** class a bit less cluttered, we created the **Team** class. A Team is essentially just an **ArrayList**[5] of **Employees**. The first member in the list is the Team Lead. Functions are provided for retrieving useful information about a **Team**.

2.5 Clock.java

In order to keep track of the current time of day and the amount of time **Employees** spent on different tasks (lunch breaks, meetings, asking questions, etc.) we created the **Clock** class as a sort of interface to the System[6] time. It provides functions for parsing the System time so that **Employees** and the **Manager** do not have to.

[4] java.util.Queue
[5] java.util.ArrayList
[6] java.lang.System

3.0 Results

We wrote a simple bash script (**test.sh**) to run the simulation 100 times and report if there are any failures. This script was executed four times with no failures reported. We were unable to uncover any race conditions (deadlocks, etc.).

In each observed run of the simulation, the behavior of the Employees and the Manager are as expected and meet the criteria outlined in the Project Description.