System Programming, Fall 2011

Assignment 3

Due date: 29/12/2011, 23:59

Contact:

Rotem Golan, rotemgol@cs.bgu.ac.il
Carmel Bregman, carmelbr@cs.bgu.ac.il

1. Assignment's Objectives

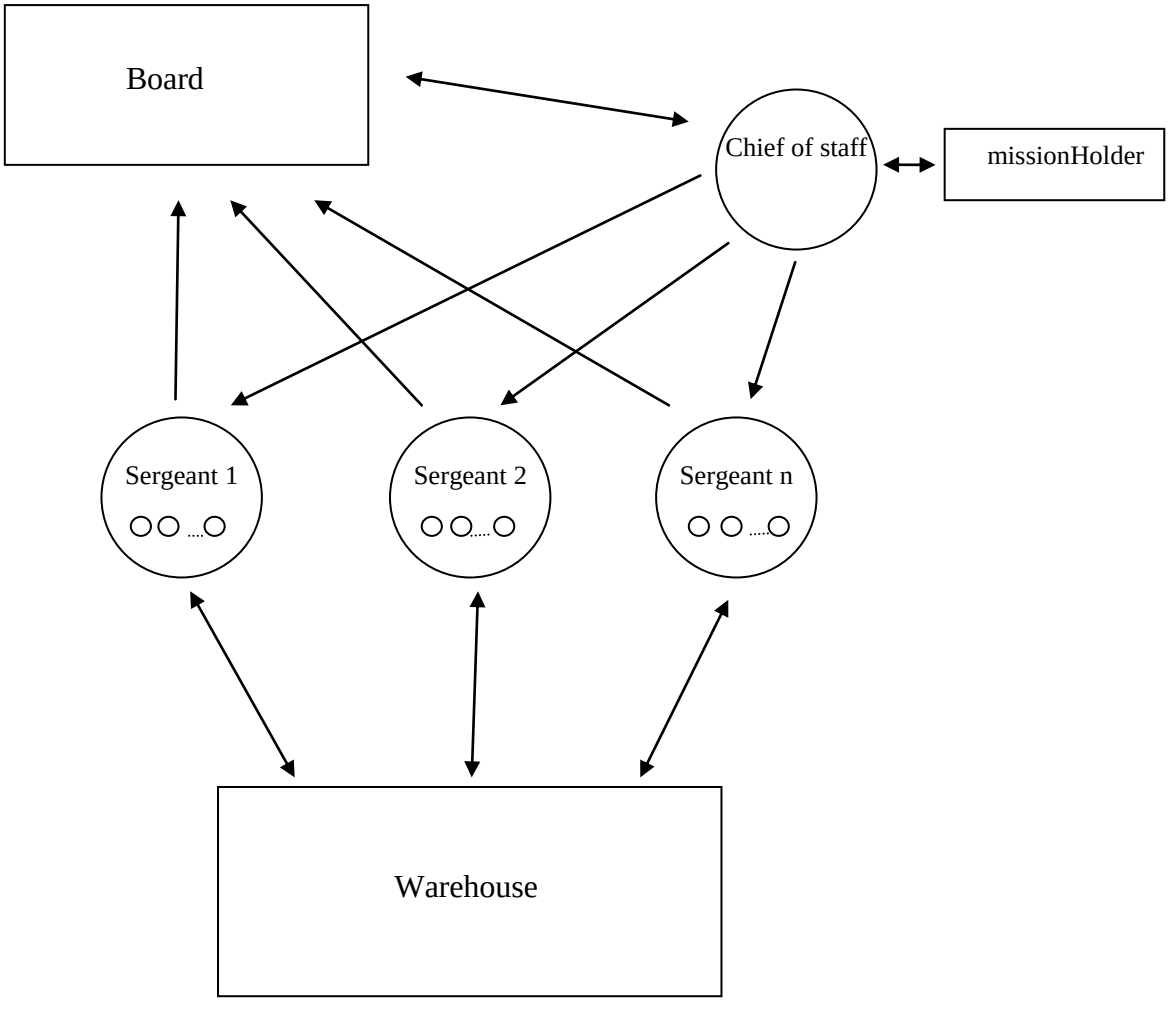
The purpose of this assignment is to gain experience both with the Java concurrent programming model and with the concepts of safety and liveness. Careful design before the actual implementation of the task is essential to make the programming process less painful and more efficient. Having said that, if you feel stuck and are not sure how to proceed with the design, start writing some code, or even better, start writing some tests. This may help sharpen your thought and understanding of the problem and lead you to the right direction.

Another objective of this assignment is to let you practice Test-Driven Development. You are required to use Test Driven Development for some parts of the code, and advised to use it for the other parts. We want to encourage you to write your tests before the actual code, and so for the required tests you should submit your interface and testing code by 15/12/2011, 23:59. For more info on the TDD requirement (see Section 9).

2. General Description

In this assignment you will simulate a full scale army attack on an enemy country. In general, simulators are used to understand the effects changes in the simulation parameters have on the entire system. In this assignment, our simulator will help us understand the relationship between its parameters and the running time of the simulation. Our goal is to get all missions done as fast as possible since we want the war to end as soon as possible.
At first, the simulator will receive missions that are necessary for the attack. These missions will be distributed to different sergeants by the chief of staff who is simulated by a single thread. Each sergeant is simulated by a pool of threads. Sergeants have at least one skill, and each mission requires a single skill to execute it. In order for the sergeants to execute their missions, they will have to get the proper tools and weapons from a shared army warehouse. In war, every mistake is critical so write your code carefully, people's lives depends on it.

The following Figure provides a graphical overview of the system:



Main Actors

2.1. **Mission**: is a passive object. Each mission has the following attributes: the name of the mission, a single sergeant's skill required to execute it, the number of hours required to complete it (timeLeft), a list of items and their amounts, and a list of pre required missions. A mission can't be executed unless all its pre required missions are completed. At program's startup, the missions are read from a configuration file and then posted on the board.

2.2. **Board:** is a passive object. The board contains information on all the missions. At first, it is updated according to the input file, and afterwards, it is used by the sergeants to notify the chief of staff when completing a mission. The chief of staff copies missions which can be executed from the board to his missionHolder.

2.3. **ChiefOfStaff:** Is an active object. The chief of staff scans the board for missions which can be executed, meaning that their pre required missions list is empty, and adds them to his missionHolder. Then he distributes these missions to different sergeants according to their skills and priorityOrder. Notice that this distribution is done in an asynchronous way, meaning that the chief of staff shouldn't wait for the completion of a mission before distributing another.

2.4. **Sergeant**: is a pool of threads. Each sergeant has the following attributes: the name of the sergeant, the number of threads, the maximum number of missions he can handle (maxMissions), a list of the sergeant's skills (its size is at least one), a top bound on the number of hours he can work on a mission without stopping (workHours), and the priority order he prefers to execute his missions. This priority order is used by the chief of staff only (see chiefOfStaff 2.3). For example, let's assume that a certain mission requires 10 hours for its completion. A sergeant that has workHours=3 should execute this task exactly 4 times (3*4=12). The last time he takes the mission, there is only 1 hour left for the completion of the mission; therefore the sergeant will only work 1 hour instead of 3. Since it is only a simulation, sergeants do not really do anything when they execute a mission – they update the mission's time parameter accordingly and perform a Thread:sleep(t) (or equivalent) for the desired amount of time t. When a mission is completed, the board should be updated accordingly.

2.5. **missionHolder:** is a passive object. There is only one instance of this object which is used by the chief of staff. This object is a list of missions. At each life cycle of the chief of staff, he should scan the board to see whether there are missions which can be executed, and if so, he will copy them into the missionHolder. After the scan is complete, the chief of staff will distribute these missions to the different sergeants according to their skills and priorityOrder. For example, let's look at a sergeant with 2 skills (sniper and commando) and a priorityOrder of *shortestMission*. The chief of staff should give this sergeant a mission which requires a commando or sniper skill and has the shortest time for execution. The parameter priorityOrder in the sergeant's state can be one of the following: *shortestMission* – mission with the shortest time for completion, *longestMission* - mission with the longest time for completion, *minItems* - mission with the smallest amount of items required, *maxItems* - mission with the largest amount of items required. We count each item similarly, for example, a mission which requires 1 helicopter and 1 tank has the same number of items as a mission which requires 2 helicopters

2.6. **Warehouse:** is a passive object. The warehouse holds different items and their amounts. It is up to you to decide how a sergeant's acquires these items, and how the collection of items is represented and managed. The only requirements we impose are that: (a) a sergeant cannot acquire an item that is already reserved by another sergeant and (b) every item can belong to only one sergeant at a time.

2.7. **Observer:** is an active object which listens to the console and accepts interactive commands entered by the user. Commands can be any of the following.

2.7.1. *completeMissions*: prints a list of all complete missions. For each mission, it should print its name, the name of the sergeant who executed it. This should be printed in a readable way of your choice.

2.7.2. *incompleteMissions*: prints a list of all incomplete missions. For each mission, it should print its name, the number of hours left to be executed, and the names of pre requisite missions that weren't completed yet. This should be printed in a readable way of your choice.

2.7.3. *sergeants*: prints a list of all sergeants. For each sergeant, it should print his name, the names of all missions that have been given to him by the chief of staff, and the time required for their completion. This should be printed in a readable way of your choice.

2.7.4. *warehouse*: prints a list of all items. For each item, it should print its name, its initial amount and its current amount. If initialAmount > currentAmount then for each sergeant who acquired this item, it should print his name and the amount he acquired. This should be printed in a readable way of your choice.

2.7.5. *addMission* [name] [skill] [time], [item1Name] [item1Amount] … [itemNName] [itemNAmount], [preRequisiteMission1] … [preRequisiteMissionM]: constructs a new mission instance and adds it to the board.

Example: *"**addMission eliminatePrimeMisinster sniper 12, sniperRifle 20 camouflageSuit 20 AK47 20, eliminateChiefofStaff**"*

2.7.6. *addSergeant* [name] [numOfThreads] [maxNumOfMissions], [skill1] … [skillN], [workHours] [priorityOrder]: constructs a new sergeant instance and starts his life.

Example*:"**addSergeant rambo 3 10, sniper commando, 10 shortestMission**"*

2.7.7. *addItem* [name] [amount]: constructs a new item instance and adds it to the warehouse.

Example*:"**addItem sniperRifle 30***"

2.7.8. *stop*: The observer should shut down the system gracefully by waiting for all the threads to end their current execution of a mission, print to the console "System has been terminated", and kill itself.

3. Simulation Time

Simulation time is related to real world time in a linear relation. Every one hour of simulation time equals one second in real world time.

4. Simulation Life Cycle

Your solution must implement the following scenario:

- The class with the main function will be called SimStarter. The final program will be packaged in a .jar file called "sim.jar". The simulation program will be started by executing: "java -jar sim.jar [name of missions file].txt [name of sergeants file].txt [name of warehouse file].txt [name of log file].txt ", for example: "**java -jar sim.jar missions.txt sergeants.txt warehouse.txt log.txt**
- The main method of the simulation loads the missions file **missions.txt**, the sergeants file **sergeants.txt**, and the warehouse file **warehouse.txt**.
- The missions file includes the number of missions, and for each mission: its name, its required skill, its size, a list of its pre required missions, and a list of items and amounts need for the mission.
- The sergeants file includes the number of sergeants, and for each sergeant: his name, number of threads, maximum amount of missions, a list of his skills, his workHours parameter, and his priority order.
- The warehouse file includes the number of items, and for each item: its name, and its amount.

  You do not need to check the configuration files correctness or anticipate inconsistencies.
- The main program creates all objects and missions, adds all missions to the board, and then start the lives of the active objects.
- During the life of the sergeants and the chief of staff, they write a trace of their operations in the log file log.txt using the Logger tool (see Section 7)**.** When all the missions in the missions file are completed, a notification "*All missions are done*" is written to the console and all threads should terminate gracefully except the observer.
- The user interacts with the system through the console.
- The program terminates gracefully after the user issues the stop command to the Observer.

5. Active Actors Life

In this section we describe in more details how the sergeant and observer behave.

Chief of staff. The life cycle of the chief of staff is as follow:

1. Checking the board for missions that can be executed and copying them into his missionHolder.
2. For each sergeant, he should check whether the sergeant can handle more missions, meaning, checking his maxMissions parameter. If that's the case, the chief of staff should add the appropriate amount of missions to this sergeant according to his priorityOrder and skills, and remove these missions from the missionHolder.
3. In case there are no missions in the board which can be added to the missionHolder and the chief of staff can't distribute any missions to his sergeants, (happens when the missionHolder is empty or all sergeants are fully busy) the chief of staff should wait for further updates.

Sergeant. The sergeant is a pool of threads which executes the missions:

1. When executing a mission, the sergeant should acquire the required items from the warehouse, sleep for the appropriate time, update the number of hours left for the completion of the mission, and return the used items to the army warehouse.
2. If the mission is not yet completed, it should be added to the same sergeant again.
3. If the mission is completed, the board's status should be updated for its completion and the sergeant should wake up the chief of staff.

- Implementing the sergeants using the Guava mechanism will give you a 10 bonus points increase to your grade (see section 12.1)

Observer. The Observer life is simple. He is created in the system initialization stage, and from then on he behaves like a "read-evaluate-print loop":

1. Display a prompt on the console (System.out).
2. Wait for input from the user.
3. Parse the input (that is, split the input into separate arguments and verify that the command is legal).
4. Perform the desired action.
5. Print the result.

He then goes back to displaying a prompt and so on. In case of invalid input from the user, the Observer prints an error message and goes back to displaying a prompt and waiting for the next command. When the Observer receives the stop command, he shuts down the system gracefully by waiting for all the threads to end their current execution of a mission, writing to the console "System has been terminated", and killing himself.

6. Files

The system is initialized according to the input files: missions.txt, sergeants.txt, and warehouse.txt. You should use the Properties class provided by Java programming language to read these input files.

Missions file example:

```
numberOfMissions= 5

m0Name= eliminatePrimeMisinster
m0skill= sniper
m0Time= 12
m0PreRequisites= eliminateChiefofStaff
m0Items= sniperRifle, 20, camouflageSuit, 20, AK47, 20

m1Name= eliminateChiefofStaff
m1skill= commando
m1Time= 48
m1PreRequisites= destroyAirForces, destroyNavy, destroyInfantry
m1Items= sniperRifle, 15, camouflageSuit, 15, AK47, 15

m2Name= destroyAirForces
m2Skill= pilot
m2Time= 96
m2PreRequisites= destroyNavy
m2Items= F15, 20, bomber, 40,

m3Name= destroyNavy
m3Skill= sailor
m3Time= 82
m3PreRequisites=
m3Items= combatShip, 30, divingSuit, 40

m4Name= destroyInfantry
m4Skill= commando
m4Time= 120
m4PreRequisites= destroyAirForces, destroyNavy
m4Items= TNT, 150, grenade, 40, rocketLuncher, 50, AK47, 300
```

Sergeants file example:

```
numberOfSergeants= 4

s0Name= rambo
s0NumOfThreads= 3
s0MaxMissions=10
s0Skills= sniper, commando
s0workHours= 10
s0PriorityOrder= shortestMission

s1Name= jake
s1NumOfThreads= 2
s1MaxMissions=7
s1Skills= pilot, commando
s1workHours= 15
s1PriorityOrder= longestMission

s2Name= sallie
s2NumOfThreads= 2
s2MaxMissions=4
s2Skills= sailor
s2workHours= 15
s2PriorityOrder= maxItems

s3Name= john
s3NumOfThreads= 1
s3MaxMissions=5
s3Skills= pilot, commando
s3workHours= 15
s3PriorityOrder= minItems
```

Warehouse file example:

```
numberofItems= 10

item0Name= sniperRifle
item0Amount= 30

item1Name= camouflageSuit
item1Amount= 30

item2Name= AK47
item2Amount= 500

item3Name= F15
item3Amount= 30

item4Name= bomber
item4Amount= 50

item5Name= combatShip
item5Amount= 50

item6Name= divingSuit
item6Amount= 30

item7Name= TNT
item7Amount= 200

item8Name= grenade
item8Amount= 50

item9Name= rocketLuncher
item9Amount= 80
```

7. Logging

In this assignment, you are required to use the logging mechanism of the Java environment (see Section 2 of the pre-assignment).

8. Issues

8.1. Deadlocks are possible. You should identify possible deadlock scenarios and prevent them from happening (an example solution can be to use a policy which involves resource ordering together with cancellation). Be prepared to explain to the grader what possible deadlocks you have identified, and how you have handled them.

8.2. Waiting: In some cases, a sergeant should wait for a new mission to appear on the board. You should avoid "busy waiting" and try to handle this issue efficiently (hint: the board can be smarter than a simple queue). Be prepared to explain to graders how you solved the wait issues.

8.3. Liveness: locking/synchronization are required for your program to work properly. Make sure not to lock too much so that you don't damage the liveness of the program. For example, when a sergeant acquires an item, he can lock all the available items until he is done, or lock just some of them. The second option is much better.

8.4. Completion: as in every multi-threaded design, special attention must be given to the clean shutdown of the system. When the Observer receives the stop command, he shuts down the system gracefully by waiting for all the threads to end their current execution of a mission, writing to the console "System has been terminated", and killing himself.

9. Unit Tests & interfaces

We expect you to write tests and an interface for the MissionHolder class. Notice that the missionHolder list can have different sorting according to its sergeant's priority order. The priority order can be one of the following: *shortestMission, longestMission*, *minItems*, *maxItems*.

Notice that this part of the assignment should be submitted until 15/12/2011, 23:59.

10. Documentation and Coding Style

Your code should be written in good style. In particular, it should pass the automatic checks of the PMD and CheckStyle Eclipse-plugins, with the rule sets supplied by the SPL team (see Section 5 of the pre-assignment). Avoid arrays (you can avoid them completely, if you need them, explain why you use them). Your code is also expected to be fully documented. All classes and public methods must include meaningful Javadoc comments.

You can learn more about Javadoc here:
http://java.sun.com/j2se/javadoc/writingdoccomments/index.html.

11. Submission Instructions

- Submission will be in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than a pair.
- Your code will be tested on a Linux machine (note that even though Java is cross-platform, threads scheduling differs slightly between Linux and Win32 – so test your code on Linux before submitting).
- You should submit one .tar.gz file with all your code. The file should be named "assignment3.tar.gz".
- In addition to the code, you should supply an ant build file which will compile your code and produce a .jar file. For additional information about ant, see http://www.cs.bgu.ac.il/~spl121/Ant.
- In order to run your program, the grader will issue the following stream of commands on a Linux console:

```
mkdir test
cp assignment3.tar.gz test/
cd test/
tar -zxf assignment3.tar.gz
ant
java -jar sim.jar missions.txt sergeants.txt warehouse.txt log.txt
```

Please make sure the file you submit will indeed run after issuing these commands as is. Your work will not be graded if the file you submit doesn't compile and run smoothly using this exact procedure

12. Grading

The assignment is graded in a frontal check, i.e. you will have to present your solution to the grader (in person), who will check your program on few scenarios and ask you questions about your tests and design. The highest score for this assignment is 110.
Note that we require you to use a .tar.gz file. Don't submit .rar, .zip, .bz, or anything else which is not a .tar.gz file.

12.1. Positive Grade

- 20 points of the grade will be for unit tests.
- 10 bonus points will be given to students who use the Guava mechanism for the way sergeants perform their tasks. Meaning, to use the callable interface instead of the runnable interface, to use submit instead of execute, and to use the chain function of the Futures class.
  See links:
  http://codingjunkie.net/google-guava-futures/
  http://guava-libraries.googlecode.com/svn/trunk/javadoc/com/google/common/util/concurrent/Futures.html
  http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html
  http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Callable.html

- 90 points of the grade will be based on the correctness of your solution: does it follow the requirements? does it shut down properly? is the work being completed? is there a single resource allocated at the same time to 2 sergeants (this is bad..)? does it get stuck? can it handle a big load, etc.
  In particular, special attention will be given to avoiding busy-waiting. Expect to lose many points if you implement waiting as busy-waiting and not with the wait/notify mechanism or an equivalent synchronization mechanism.

12.2. Negative Grade

We grade you based on the correctness of your code and your tests, and on following the functional requirements of the assignment. However, we also require you to follow non-functional guidelines, which are mandatory.
Following these requirements will not add to your grade, but not following them will harm your grade.
The following is a list of penalties which are applied AFTER your grade is computed:

- Checkstyle errors: -1 points for each error.
- PMD errors: -1 points for each error.
- Code does not compile or does not follow submission instructions: -10 points.

12.3. Partial Solutions

You can receive partial credit even if you don't complete the entire assignment. However, we need to be able to check whether your solution is correct. This means that if you don't implement parts of the program which are necessary for proving it works (e.g. the logger and the observer), we will not be able to grade you and give you partial credit.
Keep this in mind when submitting a partial solution.

Work Plan

A recommended work plan is as follows (whatever you do, remember to test your code often!):

1. Print the assignment description and read it thoroughly. Read it again. Highlight relevant parts. Understand every aspect of the simulation: the roles of every actor, what data structures each actor will maintain, the multiple wait issue, and the deadlock.
2. Go over the pre-assignment: learn to work with the Properties class. Learn to work with the Logger.
3. Write the interface and the tests for the missionHolder class. Don't forget to submit this part by 15/12/2011, 23:59.
4. Then start implementing the mechanism to acquire items.
5. Write tests and design the interfaces of the board and the other passive objects. Think of concurrency issues: what will be shared between several threads? what needs to be protected?
   Once you designed (and have tests for) and interface, start implementing it.
6. Write some of the Observer.
7. Continue with the Observer and the other active objects.

13. Frequently Asked Question

There are some answers for questions usually asked by the SPL students while working on Assignment 3.

1. Q: Why should I use Properties class? Can I write my own text scanning utility?
A: Properties class is a Java standard tool for parsing input/configuration files. You should use it for the same reason you use other standard Java class. It's convenient and it does not have any bugs (one class less to debug).

2. Q: Why should I use Logger class? Why not write System.out.println or write to a file?
A: The Logger utility is designed to let a Java program produce messages of interest to system administrators, field engineers, and software developers (and NOT to end users – the log of a program is not of interest to end users). Especially in production situations, where things can't be run in a debugger, or if doing so masks the problem that is occurring (because it is timing related as in threads, for example), program logs are frequently the greatest (and sometimes the only) source of information about a running program. Using the logger allows you to manage log efficiently, i.e., directing different types of messages to different output devices.

3. Q: I've read tutorial and examples for Logger/Properties. I still don't get it. Please, explain these topics to me.
A: Part of the assignment is to learn new topics independently. Also, you can come to the office hours of TA's and lecturers.

4. Q: I have some compile-time/run-time error I can't solve. Please, help me.
A: Come to my office hours. Surely, I would be able to understand the nature of the problem, but not always solve it on the spot. It's you who is writing the program.

5. Q: My design of the program is such and such. Is it OK?
A: This is your program. You should be guided by OOP design principles. Usually, if the design is not sound, you will know it, as the testing of your classes will become very unpleasant / inconvenient / irritating.

Other questions and answers will appear in the assignment 3 Forum. Please follow the Forum for update.