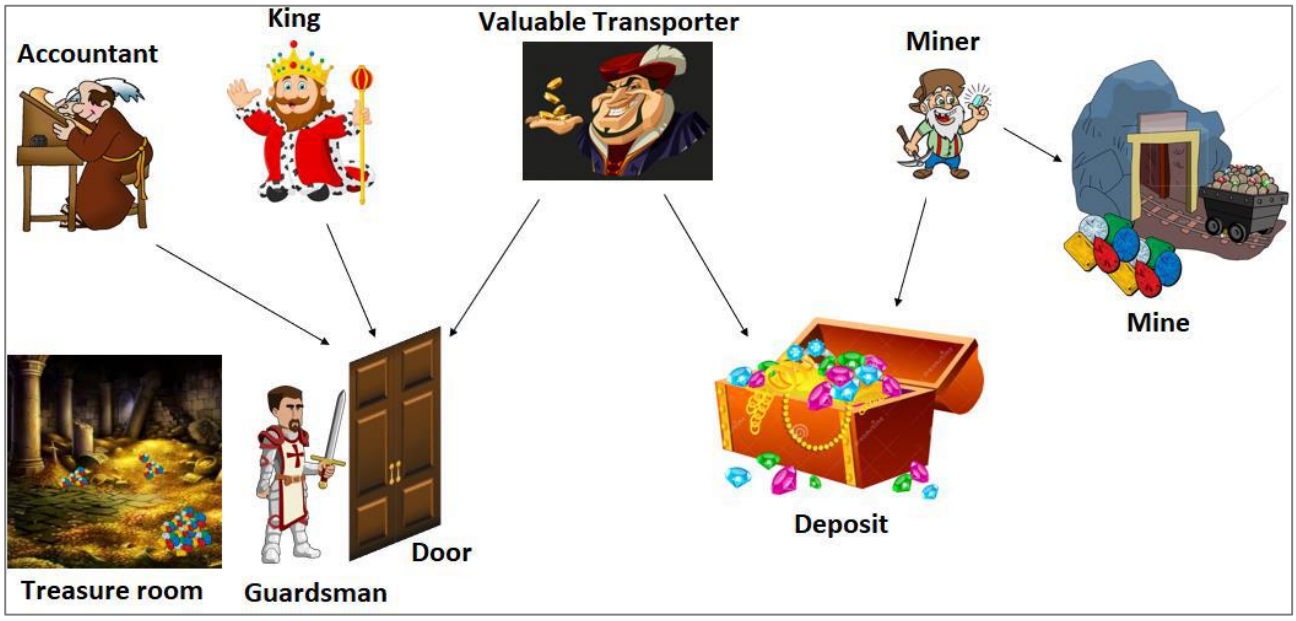
Assignment 4, SDJ2
(Singleton, Multiton, Proxy, Adapter, JUnit, Producer-Consumer, Readers-Writers)

The assignment:

You must design and implement the Kingdom simulation shown. The ultimate goal is to get valuables, so the king can throw parties.



The King has a Treasure room (lower left corner) with the door guarded buy a Guardsman. The valuables in the Treasure room comes from a Mine (upper right corner) with Miners transporting their findings to a Deposit (lower right corner). Occasionally, Valuable Transporters are moving valuables from Deposit to the Treasure room and Accountants are counting all valuables in the Treasure room. When the King feels like partying, he takes valuables from the Treasure room if there are enough for a party.

Requirements

- Threads to simulate the actors: King, Accountant, ValuableTransporter and ValuablesMiner.
- Singleton to log any action, e.g. when an actor waits or perform a job.
- Multiton for different types of Valuables, e.g. Diamond, GoldNugget, Jewel, Ruby, WoodenCoin.
- JUnit testing an ArrayList to be used in the Deposit.
    - Documentation: http://ict-engineering.dk/javadoc/MyArrayList/
    - Jar file: http://ict-engineering.dk/jar/MyArrayList-0.1.jar
- Adapter for the ArrayList in the Deposit.
- Producer-Consumer for the Deposit with the Miners and Valuables Transporters being producers and consumers, respectively.
- Readers-Writers for the Treasure room and Guardsman.
- Proxy between the Treasure room and the Guardsman.

This is quite a long list and therefore you should break it up into parts, finishing one part at the time. One way of working could be:

1. JUnit testing the ArrayList is independent of the rest and could be done at any time. There are some errors, but they may not influence in your use of it.
2. Singleton could be first step. Create a log method with a simple printout.
3. Multiton next step, as the Valuable
4. Adapter for a blocking queue Deposit delegating to the ArrayList given
5. Producer-Consumer creating two Runnable classes Miner and ValuableTransporter and the Deposit as the shared resource. Remember to use the Singleton to print out when a Miner or ValuableTransporter are waiting and working (in class Deposit). *Note: before creating a Treasure room, just let the ValuableTransporter "throw away the valuables" clearing his list.*
    - The Miner could have a while(true) loop, in which he will get a (maybe random) valuable from the Mine, and insert this into the Deposit (the blocking queue). Thus, making the Miner the producer. After, the thread should sleep for a couple of milliseconds, and start over (because of the while(true)).
    - There is quite a distance from the Deposit to the Treasure Room, so a Valuables Transporter does not want to transport only one valuable at a time. Therefore, the behaviour of the Valuables Transporter must be strictly as follows:
        1. Generate a random number, e.g. between 50 and 200.
        2. He will then, a number of times, get the next valuable from the Deposit. This will continue, until he has a list of valuables with a total worth equal to or more than the original target number.
        3. Now clear the List used to contain the valuables (currently we have no place to put them, this will come later)
        4. Sleep for a little while
        5. Start over from step 1
6. Test this part in a main method with a Deposit, a couple of Miners and a couple of ValuableTransporters
7. Readers-Writers and Proxy
    - Create an interface TreasureRoomDoor to be implemented by the Guardsman with methods to acquire and release read and write access.
        - Version 1: Either include methods to add, retrieve and look at valuables
        - Version 2: Or return a proxy object (of interface type) from the acquire methods.
    - Create class TreasureRoom. Use a list as the instance variable, e.g. the ArrayList given
        - Version 1: Implement interface TreasureRoomDoor
        - Version 2: Define a read-only interface with a method to look at valuables (a read method) and implement this interface. Include 'write methods' to add and to retrieve a valuable. Implement the same read-only interface in a TreasureRoomReadProxy class delegating to the Treasure room, see e.g. the last part of the video about the Flight-Plan example. Extra: 1) make it a protection proxy and reject access after release has been called, 2) make a proxy for a write-interface also.
    - Implement class TreasureRoomGuardsman (in version 1 as a proxy)
    - Modify the ValuableTransporter class to add the valuables retrieved from the Deposit, thus ValuableTransporter being a 'writer' (you could use sleep to simulate that it takes time to add valuables to the Treasure room)

- Implement the Accountant as a Runnable class. The Accountant is a “reader” class with a while(true) loop in the run method.
    1. He will acquire read access
    2. Count the total sum of the valuables worth in the TreasureRoom (it may include a sleep to simulate it takes time to count the valuables)
    3. Print the total sum out (using the Singleton class)
    4. Release read access
    5. Sleep for a little while
- Implement the King as a Runnable class being a ‘writer’ wanting to take out valuables from the TreasureRoom in order to throw a party. The behaviour of the King is:
    1. Similar to the ValuableTransporter, the King will generate a random number, e.g. 50-150, to pay for the next party.
    2. He will acquire write access
    3. Retrieve the valuables one at a time. If the target worth cannot be met, he will cancel the party, and put the valuables back (again, it could include a short sleep to simulate it takes time to get the desired valuables)
    4. Release write access
    a. If the target is met, he will hold a party and “throw away” the valuables retrieved.
    5. Sleep for a while
    6. Start over
- Test the full system in a main method with a Deposit, one King and a couple of each of the other actors: Miners, ValuableTransporters and Acountants

Deadline

Monday the 10th of May (week 19) at 22:00

Format

Hand-in as a single zip-file with

- Class diagram(s) (where the different patterns and other subjects are clearly identified)
- Source code for all Java classes
- Related resources if used

Evaluation

Your hand-in will be registered and counts for one of the exam requirements. No feedback will be given.