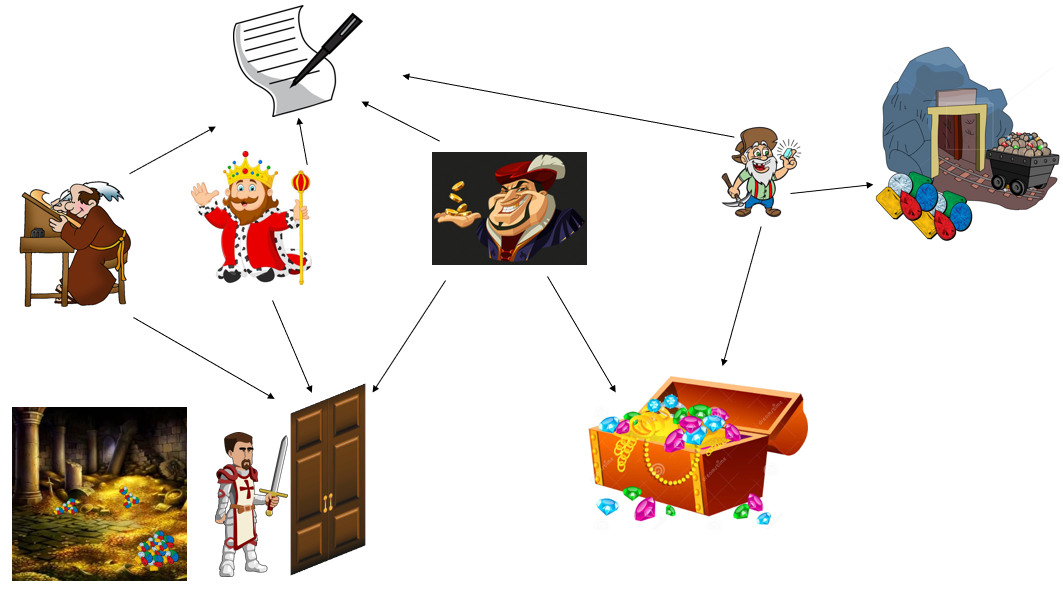
Assignment 4

## (Adapter, proxy, strategy, producer and consumer, readers and writer, unit testing, singleton, multiton)

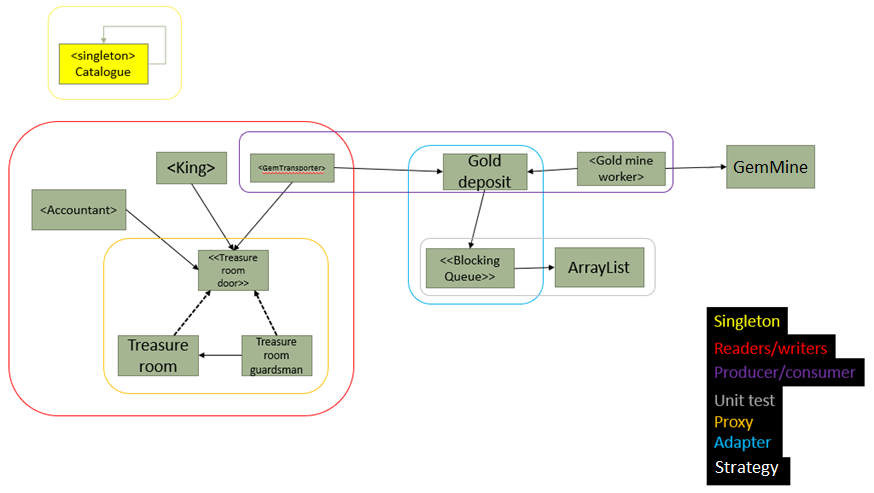
In this assignment you’re going to simulate a kingdom. The ultimate goal is to get Gems, so the king can throw parties.

There are many pieces, which must fit together, so I suggest you do them in the below described order.

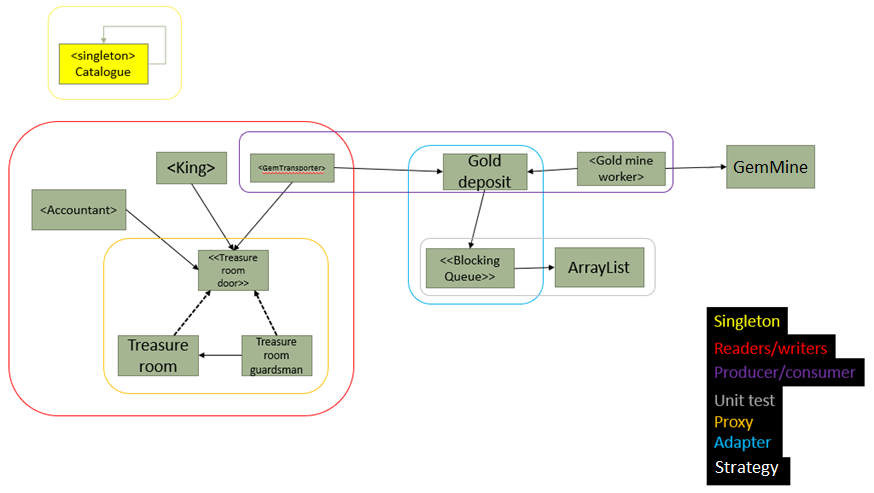
First, you shall be blessed with this wonderful schematic overview:



The following is a very rough class diagram:

****

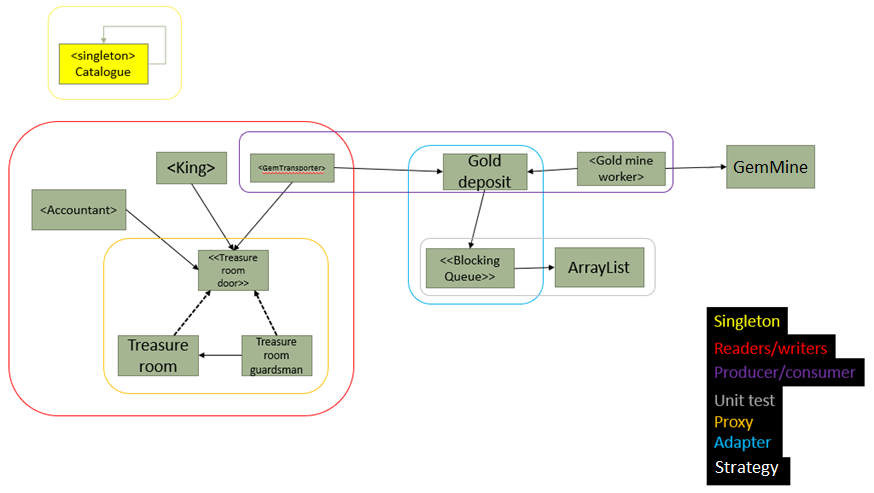
Part 1

****First, you create the Catalogue, this is to be used by all other classes, which wishes to output anything to the console.

This class is a logging functionality, to keep track of the income and outcome of the kingdom, and what actions are taking in the kingdom. Use the Singleton pattern here.

Use this logger-class to log out to the console, what happens in your program. E.g. when and how much a GemTransporter adds to the TreasureRoom, when the King holds a party or cancels it, when the Accountants have calculated the wealth in the TreasureRoom. You may also include logs when one of them are waiting to enter the TreasureRoom.

Part 2

****

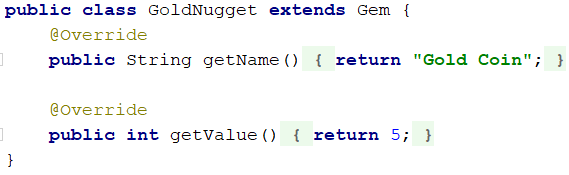
**Gem Mine**

Create the GemMine. This should be a Multiton (kind of).   
The objects stored in the Multiton in this assignment are of the type Gem.

**(Note that this is not exactly multiton, since the (list of) objects is accessed through the GemMine but the objects are of type Gem. In proper Multiton, the class holds a list of instances of itself.  
You could also forgo the GemMine and make Gem the Multiton instead, making it hold a list of Gem instances.   
I recommen you follow the first suggestion, even though it isn’t true Multiton design pattern.)**

The Gem is either an interface or an abstract class, that is up to you. It should represent valuables.

An example of a Gem sub-class could be:



The GemMine is responsible for creating and caching Gem objects, as specified by the multiton pattern. Create a couple of different types, e.g.: Diamond, GoldNugget, Jewel, Ruby, WoodenCoin, Cow etc…

There should only be one of each type.  
Let the key in the multiton be the name of the type (like “Diamond”), mapping to a specific realization of the Gem interface.

**Gem Deposit**

This is a blocking queue, as you have previously worked with in the producer/consumer problem.   
You are being **provided an ArrayList**, implemented by Steffen Vissing Andersen.   
The idea is that you use this ArrayList for your own implementation of the blocking queue.  
You must use the adapter pattern to convert the ArrayList into something that acts as a blocking queue. The class files are attached in the 7z file (in the ArrayList folder, remember to check that the ArrayList you use originates in the utility.collection package)  
Alternatively, you can download a jar file here, which must be imported as a library into your project.

Documentation: <http://ict-engineering.dk/javadoc/MyArrayList/>   
Jar file: <http://ict-engineering.dk/jar/MyArrayList-0.1.jar>

**Gem Mine Worker**

This is a Runnable class, i.e. it implements the Runnable interface, so that we can run the behaviour of the GemMineWorker in a separate thread.

The GemMineWorker must have a while(true) loop, in which he will get a (maybe random) Gem from the GemMine, and insert this Gem into the Gem Deposit (our blocking queue). Thus, making the GemMineWorker the producer.

The Gem Mine Worker should have different strategies for mining (mine fast, mine slow, only mine specific Gems, etc).  
Concrete strategies could take more or less time, and produce different qualities or quantities of Gems.

**Gem Transporter**

This is also a Runnable class, it is the Consumer. There is quite a distance from the Gem Deposit to the Treasure Room, so the Gem Transporter does not want to transport only one Gem at a time. Therefore, the behaviour of the Gem Transporter must be ***strictly*** as follows:

1. *Generate a random target number, e.g. between 50 and 200.*
2. *Then, continuously, get the next Gem from the GemDeposit.   
   Continue to do so, until the total worth of all Gems is equal to or more than the original target number.*
3. *Then, for now, just clear the (currently we have no place to put them, this will come later!)*
4. *Sleep for a little while*
5. *Go back to step 1*

**Finishing Part 2**

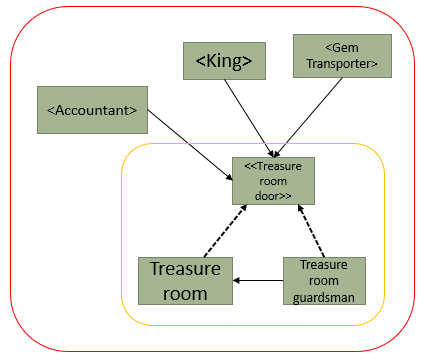
In relevant places, you should have the Catalogue print out what is happening, so you can follow along.

In a main method, instantiate a GemDeposit, start a couple of GemMineWorkers (in separate threads, with

a reference to the GemDeposit), and start a couple of GemTransporters too.

Run the program and inspect the output.

Part 3



You are being given the class for the **TreasureRoom**, and the interface **TreasureRoomDoor**. Notice the methods in the TreasureRoom, everyone can just gain access as they please. Swap out the comments in the methods with relevant calls to the Catalogue.

**TreasureRoomGuardsman**

We want to limit access to the treasure room, so we add a TreasureRoomGuardsman.   
Use the proxy pattern as indicated in the sketch above. We also want to implement the readers/writers problem here in the TreasureRoomGuardsman, so that he can control access to the TreasureRoom.

You may use any approach to solve the problem.   
Either one of the examples from the slides or your own, as long as it follows the general rules.

At the top of the class, write a small comment about your strategy.

**GemTransporter**

Now, go back to the GemTransporter, and update step 3!   
Instead of throwing out the Gems, the GemTransporter should now have a reference to the TreasureRoomDoor, and use it to insert the gems, one at a time, into the TreasureRoom.   
Remember to acquire write, execute the action, and then release write again.   
The GemTransporter is now a “writer”. Make sure the List in GemTransporter is cleared when done.

**Accountant**

The accountant is a “reader” class. It implements Runnable, so it can be run in a separate thread.   
The accountant will have a while(true) loop in the run method.

1. First, acquire read access
2. Count the total worth of all Gems in the TreasureRoom   
   (use sleep to simulate it takes time to count the Gems)
3. Print out the total worth (using the Catalogue class)
4. Release read access
5. Sleep for a little while

You should be able to expand upon your main method to now include the TreasureRoom and the

Accountant, and run the program to inspect the console output.

**King**

Finally, the king! He is a Runnable class, and a Writer.   
He wants to take out Gems from the TreasureRoom in order to throw a party.   
The behaviour of the King is:

1. First, he will determine the **cost** of the party. It should be a random number (could be between 50-150)
2. Then, he will acquire write access
3. He retrieves the Gems **one at a time** and inspects their value, adding the total value so far together

* If the **cost** cannot be met after inspecting all Gems, he will cancel the party, and put the Gems back. (again, this should include a short sleep to simulate it takes time to get the desired Gems)

1. After inspecting all Gems, he releases write access

* If the target is met, he will hold a party. And “throw away” the Gems retrieved (remove from list).

1. After a party he needs to sleep! Sleep for a while
2. Start over from step 1

Again, include relevant printouts. Update your main method to include a King. Run your program.

**Unit testing**

The ArrayList included in the source files given to you contains a few errors. You must unit test this class, use the approach shown in the session (ZOMB(IES)). Add a short comment to each test method, stating what you are testing, how, and which approach you are using (it’s not strictly necessary that you find the errors, the ArrayList still “works”. In any event, don’t correct them!).

You can find the documentation here:

<http://ict-engineering.dk/javadoc/MyArrayList/>

<http://ict-engineering.dk/javadoc/MyRecipeReader/>

Some of you may have already tested this ArrayList as part of the exercises.

**Class diagram**

You must make a detailed class diagram in Astah.

End

# Deadline

**Deadline: Monday 8th of May 23:59 on Itslearning.**

# Format

You are allowed to work in groups, but **you must each** hand in a class diagram (Astah) and the source files for all relevant Java classes, in a single zip-file.

Hand-in as a single zip-file with

* Class diagram(s) (where the different patterns and other subjects are clearly identified, put notes in Astah)
* 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**.