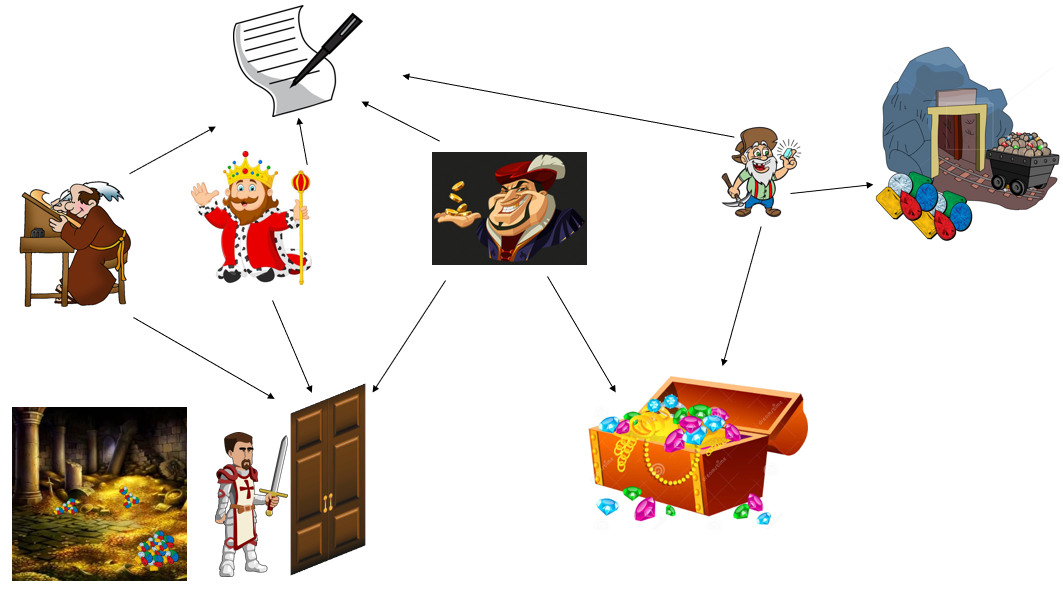
Assignment 4

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

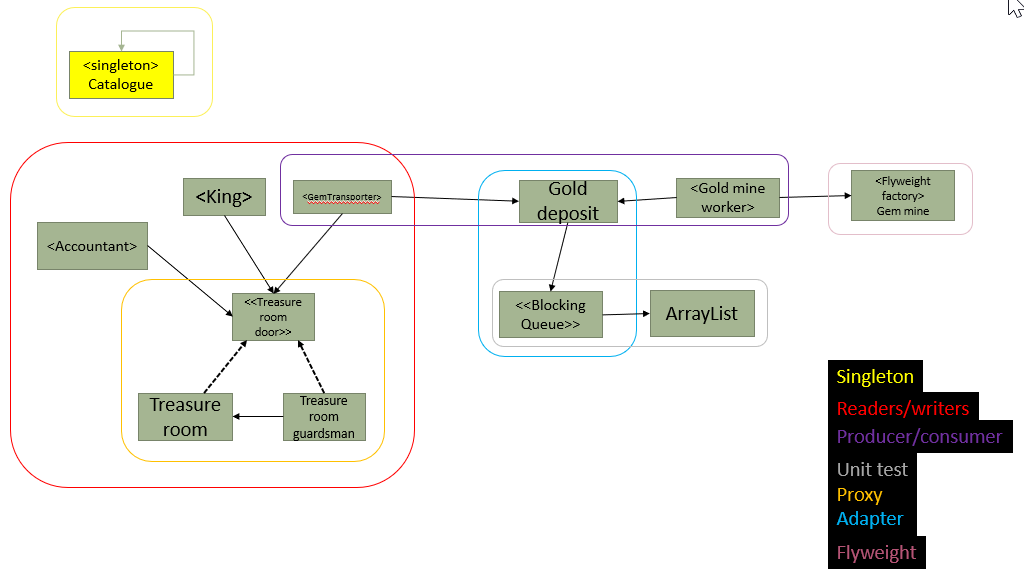
In this assignment you’re going to simulate a kingdom. The ultimate goal is to get Gems, so the king can throw parties.

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:



I have attempted to put colored boxes around the various topics.

Part 1

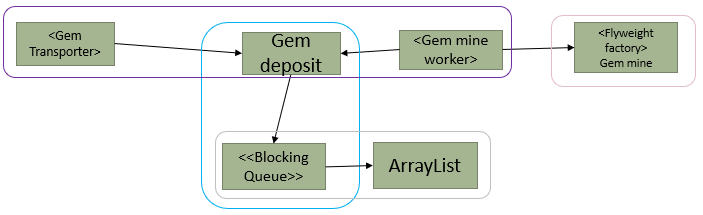
First, you create the Catalog, this is to be used by all other classes, which wishes to output anything to the console.

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

We will do the right-hand part next:

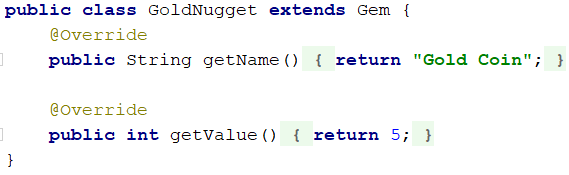


**Gem Mine**

Create the GemMine. This is the factory class in the flyweight pattern. The flyweight objects in this assignment are of the type Valuable.

This Valuable is either an interface or an abstract class, that is up to you.

An example of a Valuable sub-class could be:



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

**Gem Deposit**

This is a blocking, as is used in the producer/consumer problem. You are being **provided an ArrayList**, implemented by Steffen Vissing Andersen. You must use the adapter pattern to convert it into a blocking queue. The class files are attached in the 7z file, or you can download the jar file here, which must be included into your project, similar to what you do with JavaFX

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. After, the thread should sleep for a couple of milliseconds, and start over (because of the while(true) ).

**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 I 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 number, e.g. between 50 and 200.
2. He will then, a number of times, get the next Gem from the GemDeposit. This will continue, until he has a list of Gems with a total worth equal to or more than the original target number.
3. Now clear the List used to contain the Gems (currently we have no place to put them, this will come later)
4. Sleep for a little while
5. Start over from step 1

**Finishing Part 2**

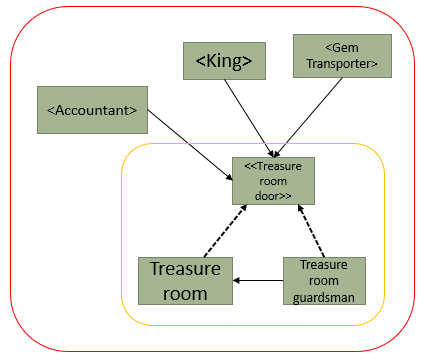
In relevant places you should get the Catalogue and have it print out what is happening, so you can follow along.

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 input.

Part 3

We will continue with the left-hand part, now adding the king, treasure room, and accountant.



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 here the proxy pattern as indicated in the sketch above. We 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, modify the GemTransporter, the step 3. Instead of throwing out the Gems, the GemTransporter should 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, release write again. The GemTransporter is now a “writer”. Clear the List in GemTransporter.

**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. He will acquire read access
2. Count the total sum of the Gems worth in the TreasureRoom (it may include a sleep to simulate it takes time to count the Gems)
3. Print the total sum out (Catalogue)
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. Similar to the GemTransporter, 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 Gems **one at a time**,

* if the target worth cannot be met, he will cancel the party, and put the Gems back. (again, it could include a short sleep to simulate it takes time to get the desired Gems)

1. Release write access

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

1. Sleep for a while
2. Start over

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

**Unit testing**

The ArrayList given to you contains a few errors. You must unit test this class, use the theory from the session. 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. 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

You can work on the assignment in the following SDJ2 sessions, – and of cause outside of class.

Monday the 4th and Friday the 7th of May.

**Deadline: Monday the 11th of May.**

# 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**.