Concurrent and Distributed Systems
Final Homework

Mitre Flavia Antonia

January 12, 2020

Third Year
Group: C.E.N. 3.1
Lecturer: Costin Bădică
Teaching assistant: Cristinel Ungureanu

1 Problem statement

Grand Sorcerer – Save the World

I. Saving the world from the undead!

Halloween has past once more! The Grand Sorcerer and his staff (witches and demons) are preparing for
another turn in saving the world and defeating the undead that every year try to conquer the world.

Grand Sorcerer

He owns multiple covens scatered throughout the world in hidden places (every year he tries again to save the world). He has demons in each coven creating potion ingredients, he has witches awaiting the potion ingredients to be made to create potions and he desperately needs his plan to work. With potions he can defeat all the undeads that will rise.

He found out from us that you are all ready to help him and implement a coven plan by using concurrency in Java (he knows already you are all experts in the field) that will provide the needed help.

Now below you can find some information that we know and found out from the Grand Sorcerer about his plans and things that are needed to be done:

II. Things that The Grand Sorcerer told us

He instructed us to tell you the following things that he has in his mind:
1. There are **multiple** witch Covens around the world:
   - Random number of Covens (between 3 and 20) that create different potion ingredients
   - Each Coven creates an unlimited number of potion ingredients of different types [there are 10 different possible potion ingredients]
   - All covens are in matrix like form (random NxN, where N is between 100-500)

- Each coven contains an array list with demons
- There can be no more than N/2 number of demons in a coven
- Covens every 10 seconds will announce its demons to stop for 1 second, scaring all demons and making them drop their ingredients

2. There are Demons. Demons are really important; they create all potions ingredients. Here's what is known about them:

(Demon at work)

- Demons are spawning randomly in each coven at a random place in the coven (random time between 500-1000 milliseconds)
- When demons are spawned randomly they need to tell to the coven that they are there
- No two demons can be on the same position
- Each demon works independently of any other demon
- Demons create potion ingredients by:

o Moving in one direction (left, right, up, down)
o When they reach a wall of the coven they move in any other direction than the wall's

*Also, for 2 extra turns they will not be able to create any ingredient in this case
*After 10 walls hits demons will be demoted with 100 levels on their social skills (see below what social skills means)

o Once they move they also create an ingredient of a different random type from the list of possible ingredient types
o If a demon is surrounded by other demons and cannot move it stops for a random time (between 10-50 milliseconds)

*When a demon encounters another demon while trying to move, they both increase their social skills rating

o With every 100 levels of social skill ratings, the demons are promoted and thus able to create 1 extra ingredient on every move

*The maximum number of ingredients possible to be created at every turn by a demon is 10

o When the ingredient is created they update their coven to know what ingredient(s) they created
o After the demon creates an ingredient, it needs to rest for a short while (30 milliseconds). They are really fast
o Demons will work like in... forever. As such they don't need to stop.

3. He also has **witches**:

(Awaiting witch, wondering if she receives all ingredients to create potions)

- Witches await to receive ingredients from the coven.
- They can always read from the coven the number of ingredients (and which ingredients), but they won't be allowed access when demons notify the coven about new ingredients
- A maximum of 10 witches can read from a coven at a time (this is also the maximum number of witches the Grand Sorcerer has)
- A random time must pass between two consecutive coven readings

(Witch visiting covens on her broomstick)

- When a witch visits the coven it will get available ingredients, put them in its magic bag and create potions out of them
o There are 20 different possible potions, each requiring between 4-8 ingredients of specific types so that it is created
o Each potion can be created in a random number of time (between 10-30 milliseconds)

o In order for the witches to not get burden with ingredients, find a proper mechanism to read only as many ingredients as possible to create potions, while also considering to not have too many ingredients per coven (see fairness)

o It is left for you to think about potential potions and their ingredient list

- Witches will put all potions through a magic portal and send them to the Grand Sorcerer
- Multiple witches can at the same time put the potions through the magic portal while the Grand Sorcerer reads by itself all potions sent through the magic portal
- Witches are known from the beginning.

4. There are the **undead**:

- There is a fixed number of undeads at the beginning, a random number between 20-50
- Every random time (between 500-1000 milliseconds), they randomly visit covens
- Every visit at a coven, per undead, is translated as follows:

o A random number of demons, between 5-10 is retired, being too scared to work anymore

o Current available ingredients are lost - If there are witches visiting the coven then:

o One witch will fight an undead

o She will request 2-5 potions from the Grand Sorcerer to fight the undead

o If there are available potions, then the undead will run scared

o If not, for every undefeated undead, 10 percent of the ingredients available in the coven will be lost, but no demon will be lost

5. The main Grand Sorcerer's Circle will do the following:

- this Circle contains all covens
- it creates the covens
- it creates the Grand Sorcerer helper that spawns demons and gives them a random coven to work
in (remember that each demon is responsible to register itself to the coven)

III. Things that are necessary

Please note that the (concurrent) control flow in this example is rather subtle. The moving of demons and reporting to the coven is initiated independently by each of the demon, as each of them is running a separate thread of control.

Because of that, different demons may execute the move action and coven reporting concurrently.
Concurrent coven reporting has to also be synchronized.

But calling report also triggers calling the individual report method of all the demons registered with the coven. This again has to be properly taken care of, because both report and move methods are synchronized, which means that they will not be called concurrently for a given object.

IV. Some hints

The Grand Sorcerer knows you can help him. He knows that you have in your tool belt the following concurrent notions that you can help him with:

- Semaphores, Monitors and Locks (he knows that you will fairly use all of them)
- Threads are really important for him. He even said that each of his demons can be a separate thread in your coven plan implementation
- He wants them to communicate: the demons create the potion ingredients and the witches receive them, transform them in potions to give to The Grand Sorcerer to conquer the world
- And The Grand Sorcerer dearly wants the witches to deliver him the potions to his location through a magical portal of TCP/IP (he found out this is good). If he receives all potions then he can know how to conquer the world.

V. Important tasks to be done (as well)

1. You can retire a demon

Your first task is to add still another thread to your program that retires the demons at random times (i.e. with short random delays in between). But the demons should be retired in random order. Retiring a demon means that its run method must terminate. This should be achieved by making the loop terminate normally and NOT by calling the deprecated method to stop a thread.

Further, the demon must be removed from the coven (in a thread-safe way similar to adding a demon to a coven). After this has been done the garbage collector in the run-time system will eventually reclaim the space allocated for the demon object.

To solve the problem you can make use of a semaphore that the demon tries to acquire in order to “get permission to retire”. The semaphore is initially zero and then released a number of times in a thread started in main. Note that it is very useful to try to acquire the semaphore, using its tryAcquire method.

Implement this and test your program. Make sure that you understand how the design makes the retiring order unpredictable. If you want, you can change the behavior of the program further so that retired demons are respawn after some (random) time.

2. You can give a demon time to sleep (because you’ve noticed that

it is really tired)

Now return to the original version of the program (make a new directory and reuse the initial set of classes you implemented).

You must now modify the program to achieve the following behavior: When a demon after one of its moves finds itself in the diagonal area of the world (i.e. where x is very close to y), it will "go to sleep", i.e. stop moving. Note that a demon may jump over the diagonal area in one move; this will not cause it to sleep. When all demons have frozen at the diagonal, they will all wake up and continue moving until they sleep again on the diagonal. This moving/sleeping continues forever.

You should recognize this as a form of barrier synchronization that can be achieved using N + 1 semaphores: one common barrier semaphore, which demon threads release when they reach the synchronization point, and an array of "continue" semaphores, indexed by thread, which threads acquire in order to continue beyond the barrier.

A special barrier synchronization process is also needed, which repeatedly acquire the barrier semaphore N times, followed by releasing all the continue semaphores.

3. Sleeping demons revisited

The package java.util.concurrent includes the class CyclicBarrier, which provides more convenient means to achieve barrier synchronization. Rewrite the program from the previous exercise using this class instead of semaphores.

4. Your own cycle barrier

Now for a harder exercise: implement the body of your own class CyclicBarrier which provides similar features as the Java class of the same name. But we are content with a simpler version with the following spec:

```
public class CyclicBarrier {

    public CyclicBarrier(int parties);

    public void await();

}
```

The parameter parties are the number of threads that need to reach the barrier before they are all allowed to continue. Write the whole class and then use it to solve the sleeping demon problem again.

Hint: We cannot follow the array-of-semaphores approach directly, since that would require await to have a parameter i indicating the index of the semaphore on which to block. Instead, one could try to use a single semaphore on which all processes block and an integer variable counting how many processes have reached the barrier. But this integer variable is shared, so we need to protect updates to it using a second, mutex semaphore. We cannot use the Java synchronized locking instead of the mutex semaphore. Why?

2 Implementation

2.1 Original problem

For solving the problem I've created the classes:

- **RandomNumberGenerator**

  This class contains the method **generateRandomNumber (int MarginLeft, int MarginRight)** that generates a random number in a given range.

- **GrandSorcerersCircle**

  randomNumberGenerator - this variable assures access to the RandomNumberGenerator class

  This class contains the main of the project.

  In the main:

  CovensNumber - contains the coven number that is generated Print the number of covens
  setCovensNumber(CovensNumber) - Store the number of covens// Creates the covens and the matrices of the covens
  covens[i].CovenSize - The size of coven is generated
  covens[i].MaxDemonsNumber = covens[i].CovenSize/2 - Calculate the max number of demons for coven i
  AllCovensMaxDemonsNumber = AllCovensMaxDemonsNumber + covens[i].MaxDemonsNumber - Calculate the total maximum number of demons from all the covens
  covens[i].CovenMatrix[j][k] = 0 - Every cell of the coven matrix is initialised with 0 - indicates that is free
  End of main

  setCovens(covens) - passes the covens created here
  setMaxDemonsNumber(AllCovensMaxDemonsNumber) - saving the max demons number
  setUndeadsNumber(randomNumberGenerator.generateRandomNumber(20,50)) - saving the number of undeads setGrandSorcererHelper(grandSorcererHelper) - save the grand sorcerer helper
  A new thread is created for the grand sorcerer helper.

- **GrandSorcererHelper**

Method **DemonCreator**(int demonIndex, Demon[] demons) creates demons and assigns them to a random coven

For every demon:

Print it's index
Create a new instance
setWallHitsCounter to 0
Save the number of created demons
A new thread is created, for every demon
This thread is also started

Method **WitchesCreator**(int witchIndex, Witch[] witch) creates witches
200 is the maximum number of witches.
A new thread is created and started, for every witch.

Method **UndeadCreator** (int undeadIndex, Undead[] undead) creates undeads grandSorcerersCircle.getUndeadsNumber() is the number of undeads.
A new thread is created and started, for every undead.

Method **retireDemons** (int destroyedDemonsNumber, int UndeadIndex, int CovenIndex)

Print which undead wants demons to retire
Generate a random demon index while the demon generated is already retired or its coven is not corresponding to the attacked coven.
demon.isRetired = true - The demon retires.
A value that is never generated for a demon index, indicating there is no demon (-1).
The position where the retired demon was becomes free.
The current number of demons from the coven decreases.

We will need a lock and a semaphore with one permit

This class implements Runnable (overrides the run method from threads).

In the run method:
A lock is used to lock the witches creating critical section.
Witches are created using the method for creating witches.
A semaphore with one permit is used to help create undeads one at a time
A lock is used to lock the demons creating critical section.

Between creations the thread sleeps.
Demons are created using the method for creating demons.

- **Coven**

Here I created the varialble UndeadsNumber that will store the number of undeads from the coven, four semaphores with different purposes: one for stoping demons synchronisation (1 permit), one for demon spawn synchronisation (1 permit), one for the maximum number of witches that can visit the coven at a time(10 permits), and one for demons creating (100 permits).
I created also an array list of demons(the coven must have a demons list).

The internal class **CovenMatrix**
This class was created to fulfill the need of knowing line and column values for a certain demon.
Methods from this class: void setDemonIndex, setLine, setColumn, getDemonIndex, getLine, getColumn.

In the Coven class: list of Ingredients and their names, list of ingredients amounts.
Method **IncreaseIngredientCounter** increases the number of ingredients of a given kind:
calculates the new number of ingredients, replaces the old number of ingredients of this type(from the given index) with the new number of ingredients.

Method **DecreaseIngredientCounter** decreases the number of ingredients of a given kind:
If the condition is false, the number of ingredients becomes 0 - dropping all the ingredients.
calculates the new number of ingredients, replaces the old number of ingredients of this type(from the given index) with the new number of ingredients.

Method **loseIngredients**:
Variable amountLost - used to calculate the amount of lost potions.
Variable newAmount - the new amount of existing potions.
Calculate the amount lost as the multiplication between the number of potions and the percent of lost potions.
The new amount is calculated by decreasing the old amount of potions

with the amount lost.
IngredientsCount.set(i, newAmount) - storing the new amount of potions.
Print what percent of ingredients has the coven lost.
Method **StopDemons** returns true when the current time can be divided by 10 seconds.

- **Demon**
  Method **decreaseSocialSkillsLevel** decreases the social skills level of a demon by a given value.
  Method **increaseSocialSkillsLevel** increases the social skills level of a demon by a given value.
  Constructor: the demon registers to a coven using it's index
  Method **createIngredient**:
  the number of ingredients produced when a demon moves is given by its social skills level
  usepackage the demon must create at least 1 ingredient regardless its social skills level
  the maximum number of ingredients a demon can create per move is 10
  Print which demon creates which ingredients.
  the coven knows that new ingredients are created

  Method **RandomSpawn**: the demon spawn in a random coven.
  Generating a random line to spawn at.
  Generating a random column to spawn at.
  Checks is the position is free (if there is not already other demon), if it is free, this position becomes occupied.
  If the demon hasn't spawned, it will try again until he succeeds

  Method **isSurrounded** checks if a demon is surronded by other demons.
  If the demon is not at the left wall, if the left position is occupied by a demon, we count it.
  if the left position can't be occupied by a demon, it is blocked, so the demon
  can't move there and we count it.
  If the demon is not at the right wall, if the right position is occupied by a demon, we count it.
  If the right position can't be occupied by a demon, it is blocked, so the demon can't move.
  If the demon is not at the upper wall and if the upper position is occupied by a demon, we count it
  If the upper position can't be occupied by a demon, it is blocked, so the demon can't move there and we count it.
  If the demon is not at the down wall and if the upper position is occupied by a demon, we count it
  If the down position can't be occupied by a demon, it is blocked, so the

demon can't move there and we count it.
If 4 demons are surrounding the demon, return true, else false.

Method **RandomMove** makes a demon move random in the coven.
avoid walls: move up - the demon shall go to the upper line or move down - the demon shall go to the next column or move down - the demon shall go to the lower line or move left - the demon shall go to the previous column.
If the position is free: the last position becomes free, this position becomes occupied, a value that is never generated for a demon index, indicating there is no demon.
if the demon does not collide a wall and it is not restricted from producing ingredients, it creates a number of ingredients.
If the demon collides a wall, for the next 2 moves he can't create ingredients.
If a demon is blocked from creating ingredients from the last move, this should decrease
If the demon hasn't moved he met another demon.
If the position where the demon wants to move is occupied, then its social skills level increases
If the position where the demon wants to move is occupied, then the other's demon social skills level increases.
Returns a boolean value telling if the demon has or hasn't moved.

Method **CheckWallCollision** checks if a demon collides a wall.
The number of wall hits is increased for this demon.
If a demon hits a wall 10 times, this demon will be demoted with 100 levels on their social skills.
If the demon collides the wall the function returns true.
If the demon does not collide the wall the function returns false.

Method **dropIngredients** makes demons drop all their ingredients of a given type.

This class implemments runnable(overrides the run method from threads).
In the run method:
The coven is announced that a new demon will be spawned, so the demon spawn semaphore is acquired
he random spawn function is called
The new spawned demon is added to the covens's demon list (he joins the coven)

The semaphore for demon spawning is released

While the demon is not retired:
the demons creating ingredients should not be interrupted by witches taking ingredients,
if a demon is surrounded by other demons it sleeps a random time between 10 and 50 milliseconds
if the demon has moved, it has created one or more ingredients, so it sleeps for 30 milliseconds
The demons finished making ingredients, so witches can take them.
if the coven tells the demons to stop the semaphore for demons stop is acquired.
Demons drop all their ingredients.
Demons sleep for 1 second.
The semaphore for demons stop is released.
End of while.

- **Witch**
  Method **readIngredients** is used to read ingredients from a coven.
  Method **selectRandomCoven** is used to select a random coven to read ingredients from.
  Method **fightUndead** is used to fight undeads. If the grand sorcerer can provide the number of necessary potions return true, else return false (has been or hasn't been defeated).

  This class overrides the run method from threads.
  In the run method:
  Select a random coven to read ingredients from.
  If a witch reads, she should acquire the semaphore for reading ingredients from the selected coven.
  Reading ingredients and creating potions must be synchronised.
  The grand sorcerer should not read potions while witches are creating them
  read ingredients from this coven.
  A time must pass until the next reading.
  There are 20 potions that can be created.
  If there are undeads attacking the coven a witch will fight.
  The witch has finished reading ingredients and creating potions so the lock is unlocked.
  Release the semaphore for reading from this coven.
  End of the run method.

- **Undead**

  Method **RandomVisit** makes an undead go to a random visit in a coven.
  If the undead isn't fighting a witch and there are more than 10 demons at the moment in a coven retire a random number of demons (between 5 and 10), also lose all the ingredients from the coven.
  If the undead was fighting a witch, if it isn't defeated, the coven loses 10 percent of the ingredients.
  If it was defeated he runs scared.
  Decrease the undeads number from the coven.
  This class implements runnable.
  In the run method:
  Acquire a semaphore because if more undeads are trying to make demons retire from the same coven, all the demons created end up being retired in a short time.
  Increase the undeads number from the random generated coven to visit.
  Visit the coven.
  Release the semaphore for visits.
  Sleep for some time until another visit.

- **GrandSorcerer**
  This class represents the grand sorcerer that reads and gets potions to save the world.
  Method **readPotions** is used to read potions and their amounts.
  Method **givePotions** is used to give potions to witches when they fight undeads. If the number of asked potions is found, it returns true, else, false.

- **MagicPortal**
  This class helps witches give the grand sorcerer potions and viceversa.
  ethod **increasePotionsCount** increases the potions count for a potion with a given amount.
  The new number of potions is stored here.
  The value from the list is modified to the new value.

- **Potion**
  This class is used for defining potions: their names, receipes and necessary amount of ingredients.
  Method **createPotion** is used to create potions.
  Returns true if there are enough ingredients to create the potion.

Task I: retiring demons

To solve this task I've created the class **DemonRetirer**
This class implements Runnable.
In the run method it tries to acquire a demonRetirer semaphore.
The demon retiring must be synchronised
Sleep for some time.
Get a random coven index from the existing ones.
Get a random demon from the coven's demon list.
Save the demon.
This demon becomes retired, so it will end it's execution soon.
The retiring ended so the semaphore is released.
This method is called in the main of the program.
The demon retirer thread is started there.

Task II: demons sleeping in the diagonal area

For solving this task I've created the class "DiagonalArea"
This class get the covens from the grand sorcerer'r circle and implements runnable.
In the run method:
For every coven:
If the diagonal is full of sleeping demons try to acquire the semaphore, if no demon is moving it will succeed
Diagonal area is full.
The demons will all wake up, so the asleep counter resets.
For every demon from the coven if the demon is asleep, the demon wakes up.
The semaphore is released.
This run method is called in the main of the program.
The diagonal area thread is started there.
In the demon's run method the same semaphore as above is acquired after a move is made, and released before.

Task III: demons sleeping in the diagonal area with cyclic barrier

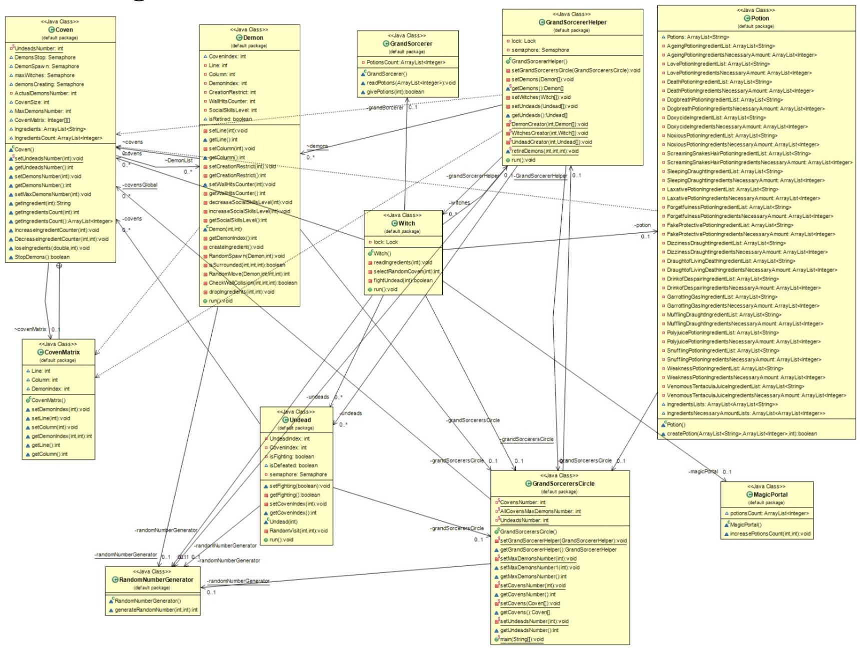
For solving this task I've created the class "DiagonalArea" This class get the covens from the grand sorcerer'r circle and implements runnable.
In the run method:
For every coven:
If the diagonal area is full the demons will all wake up, so the asleep counter resets.
For every demon from the coven if the demon is asleep, the demon wakes

up.
This run method is called in the main of the program.
The diagonal area thread is started there.
In the demon's run method the cyclic barrier for sleeping demons awaits for the diagonal to be full of demons. When it is, they all wake up.
The broken barrier exception is mentioned in the try-catch statement.

Task IV: your own cyclic barrier

For solving this task I've created the class **CyclicBarrier**.
This class contains a number of parties, a number of parties received, a semaphore with the number of permits equal with the number of parties, a constructor that has an int value as a parameter(parties number) and the method **await**.
In this method the number of parties received is increasing at every call of the function.
The semaphore is acquired.
If the number of parties received is equal to the initial number of parties, all the semaphores are released.
End of await method.
The demon run function stays the same as in task III.

UML Diagram


<<Java Class>>
Coven
(default package)
UndeadsNumber: int
DemonsStop: Semaphore
DemonSpawn: Semaphore
maxWitches: Semaphore
demonsCreating: Semaphore
ActualDemonsNumber: int
CovenSize: int
MaxDemonsNumber: int
CovenMatrix: Integer[][]
Ingredients: ArrayList<String>
IngredientsCount: ArrayList<Integer>
Coven()
setUndeadsNumber(int):void
getUndeadsNumber():int
setDemonsNumber(int):void
getDemonsNumber():int
setMaxDemonsNumber(int):void
getIngredient(int):String
getIngredientsCount(int):int
getIngredientsCount():ArrayList<Integer>
IncreaseIngredientCounter(int):void
DecreaseIngredientCounter(int,int):void
loseIngredients(double,int):void
StopDemons():boolean
<<Java Class>>
Demon
(default package)
CovenIndex: int
Line: int
Column: int
DemonIndex: int
CreationRestrict: int
WallHitsCounter: int
SocialSkillsLevel: int
isRetired: boolean
setLine(int):void
getLine():int
setColumn(int):void
getColumn():int
setCreationRestrict(int):void
getCreationRestrict():int
setWallHitsCounter(int):void
getWallHitsCounter():int
decreaseSocialSkillsLevel(int):void
increaseSocialSkillsLevel(int):void
getSocialSkillsLevel():int
Demon(int,int)
getDemonIndex():int
createIngredient():void
RandomSpawn(Demon,int):void
isSurrounded(int,int,int):boolean
RandomMove(Demon,int,int,int):int
CheckWallCollision(int,int,int):boolean
dropIngredients(int,int):void
run():void
<<Java Class>>
GrandSorcerer
(default package)
PotionsCount: ArrayList<Integer>
GrandSorcerer()
readPotions(ArrayList<Integer>):void
givePotions(int):boolean
<<Java Class>>
GrandSorcererHelper
(default package)
lock: Lock
semaphore: Semaphore
GrandSorcererHelper()
setGrandSorcerersCircle(GrandSorcerersCircle):void
setDemons(Demon[]):void
getDemons():Demon[]
setWitches(Witch[]):void
setUndeads(Undead[]):void
getUndeads():Undead[]
DemonCreator(int,Demon[]):void
WitchesCreator(int,Witch[]):void
UndeadCreator(int,Undead[]):void
retireDemons(int,int,int):void
run():void
<<Java Class>>
Potion
(default package)
Potions: ArrayList<String>
AgeingPotionIngredientList: ArrayList<String>
AgeingPotionIngredientsNecessaryAmount: ArrayList<Integer>
LovePotionIngredientList: ArrayList<String>
LovePotionIngredientsNecessaryAmount: ArrayList<Integer>
DeathPotionIngredientList: ArrayList<String>
DeathPotionIngredientsNecessaryAmount: ArrayList<Integer>
DogbreathPotionIngredientList: ArrayList<String>
DogbreathPotionIngredientsNecessaryAmount: ArrayList<Integer>
DoxycideIngredientList: ArrayList<String>
DoxycideIngredientsNecessaryAmount: ArrayList<Integer>
NoxiousPotionIngredientList: ArrayList<String>
NoxiousPotionIngredientsNecessaryAmount: ArrayList<Integer>
ScreamingSnakesHairPotionIngredientList: ArrayList<String>
ScreamingSnakesHairPotionIngredientsNecessaryAmount: ArrayList<Integer>
SleepingDraughtIngredientList: ArrayList<String>
SleepingDraughtIngredientsNecessaryAmount: ArrayList<Integer>
LaxativePotionIngredientList: ArrayList<String>
LaxativePotionIngredientsNecessaryAmount: ArrayList<Integer>
ForgetfulnessPotionIngredientList: ArrayList<String>
ForgetfulnessPotionIngredientsNecessaryAmount: ArrayList<Integer>
FakeProtectivePotionIngredientList: ArrayList<String>
FakeProtectivePotionIngredientsNecessaryAmount: ArrayList<Integer>
DizzinessDraughtIngredientList: ArrayList<String>
DizzinessDraughtIngredientsNecessaryAmount: ArrayList<Integer>
DraughtofLivingDeathIngredientList: ArrayList<String>
DraughtofLivingDeathIngredientsNecessaryAmount: ArrayList<Integer>
DrinkofDespairIngredientList: ArrayList<String>
DrinkofDespairIngredientsNecessaryAmount: ArrayList<Integer>
GarrottingGasIngredientList: ArrayList<String>
GarrottingGasIngredientsNecessaryAmount: ArrayList<Integer>
MufflingDraughtIngredientList: ArrayList<String>
MufflingDraughtIngredientsNecessaryAmount: ArrayList<Integer>
PolyjuicePotionIngredientList: ArrayList<String>
PolyjuicePotionIngredientsNecessaryAmount: ArrayList<Integer>
SnufflingPotionIngredientList: ArrayList<String>
SnufflingPotionIngredientsNecessaryAmount: ArrayList<Integer>
WeaknessPotionIngredientList: ArrayList<String>
WeaknessPotionIngredientsNecessaryAmount: ArrayList<Integer>
VenomousTentaculaJuiceIngredientList: ArrayList<String>
VenomousTentaculaJuiceIngredientsNecessaryAmount: ArrayList<Integer>
IngredientsLists: ArrayList<ArrayList<String>>
IngredientsNecessaryAmountLists: ArrayList<ArrayList<Integer>>
Potion()
createPotion(ArrayList<String>,ArrayList<Integer>,int):boolean
<<Java Class>>
Witch
(default package)
lock: Lock
Witch()
readIngredients(int):void
selectRandomCoven(int):int
fightUndead(int):boolean
run():void
<<Java Class>>
CovenMatrix
(default package)
Line: int
Column: int
DemonIndex: int
CovenMatrix()
setDemonIndex(int):void
setLine(int):void
setColumn(int):void
getDemonIndex(int,int):int
getLine():int
getColumn():int
<<Java Class>>
Undead
(default package)
UndeadIndex: int
CovenIndex: int
isFighting: boolean
isDefeated: boolean
semaphore: Semaphore
setFighting(boolean):void
getFighting():boolean
setCovenIndex(int):void
getCovenIndex():int
Undead(int)
RandomVisit(int,int):void
run():void
<<Java Class>>
GrandSorcerersCircle
(default package)
CovensNumber: int
AllCovensMaxDemonsNumber: int
UndeadsNumber: int
GrandSorcerersCircle()
setGrandSorcererHelper(GrandSorcererHelper):void
getGrandSorcererHelper():GrandSorcererHelper
setMaxDemonsNumber(int):void
setMaxDemonsNumber1(int):void
getMaxDemonsNumber():int
setCovensNumber(int):void
getCovensNumber():int
setCovens(Coven[]):void
getCovens():Coven[]
setUndeadsNumber(int):void
getUndeadsNumber():int
main(String[]):void
<<Java Class>>
MagicPortal
(default package)
potionsCount: ArrayList<Integer>
MagicPortal()
increasePotionsCount(int,int):void
<<Java Class>>
RandomNumberGenerator
(default package)
RandomNumberGenerator()
generateRandomNumber(int,int):int
~covens
~covens
0..*
~DemonList
0..*
-covensGlobal
0..*
-covens
0..*
~demons
0..*
~covenMatrix
0..1
-grandSorcerer
0..1
-grandSorcererHelper
0..1
-GrandSorcererHelper
0..1
-witches
0..*
-potion
0..1
-undeads
0..*
-undeads
0..*
-grandSorcerersCircle
-grandSorcerersCircle
-grandSorcerersCircle
0..1
0..1
-grandSorcerersCircle
0..1
-grandSorcerersCircle
0..1
-magicPortal
0..1
-randomNumberGenerator
-randomNumberGenerator
0..1
-randomNumberGenerator
-randomNumberGenerator
0..1
-randomNumberGenerator
0..1

3 Conclusions

This final homework made me learn a lot of things and tested my skills and knowledge in Object Oriented Programming, multithreading and synchronisation mechanisms.// I've used many types of synchronisation mechanisms: semaphores with different number of permits (ex: 10 permits semaphore for witches reading from a certain coven), locks that lock some critical sections, and even cyclic barriers to help demons stop in the diagonal area.
It also tested my imagination: creating the potions and recipes, the ingredients etc.