Réalisation d'application groupe 11

LUKAKU KATANU TRESOR 14500958

MBOMBO MOKONDA CHRIST 13409354

JEUX D ‘AVENTURE

LA MAISON HANTE

ITERATION I

Synopsis :

Bonjour est Bienvenu dans la maison Hanté, la demeure des esprits perdus, là ou la pitié, l'amour, n'existe pas. Vous êtes courageux, en quête d'une expérience Spirituelle et d'aventure, guidée par l'esprit de découvrir les choses cachés.

Vous êtes jeunes, brave, vif(ve) et vous voulez mettre vos compétences et votre spiritualité à rue d'épreuve, c'est ce pourquoi vous vous êtes lancé le défi d'aller seul dans ce guide là ou aucun Humain n'a survécu "La maison Hanté".

Vous témérité est votre a tout mais aussi votre défaut :
Dans votre empressement de partir seul(e) à l'aventure, vous avez oublié de vous renseigner sur la Zone.

Vous ne connaissez donc rien de l'endroit dans lequel vous vous trouverez.

La nuit commence à tomber, une pluie diluvienne fait son apparition, de bouillard vous empêche de voir à plus d'un mettre de vous et la ville la plus proche se trouve à 10 000 pieds de là

Vous ne pouvez plus faire demi-tour et décidez donc d’entrer dans l'enceinte de la maison.

Juste avant d'entrer, devant la porche une voix soudain sortie de nulle part vous explique que les lieux sont habités par des esprit surnaturel, et de démons déchut et qu’ils s’agissent donc bien d’une maison hantée où on ne peut ni parler, ni respirer. Beaucoup d'aventurier s'y sont précipité et s'y précipite en espérant devenir le premier aventurier à entrer dans la salle de l'enfer et y ressortir.

Selon la légende lors de la création du monde y a 25 milliard d'année, L’antéchrist avait construit cette maison pour y vivre avec ses anges et donc dans cette maison l'une de pièce serait la demeure de l'Antéchrist.

Vous décidez d'y entrer et vous demandez sur quoi allez tomber en traversant le lieu.

Essayez de Survivre à votre propre Destin.

Contenu

5 Personnages :

- Personnage Principal (Vie, ArmeP, ArmeS, Mana, Armure(jambes, gants, …), Trousse)

- Personnage Secondaire (Vie, ArmeP, ArmeS, Mana, Armure(jambes, gants, …), Trousse)

- LA VOIX (Trousse, code)

- Gardien(Argent)

- Magicien(Trousse)

Bestiaires :

Liste de tous les monstres

3 Coffres :

- coffre0 → Chaque début de partie (Zone 0)

- coffre1 → Finir le jeu (Zone 13)

- coffre2 → Atteindre un certain niveau (Zone 12)

2 Digicodes :

- Digicode0 : mène à la salle 13 (Finir le Royal Rumble de la salle 12)

- Digicode1 : relie des couloirs (Acheter au près du vendeur de la salle 11)

Rôles

Personnage principal :

C'est le joueur que l'on incarne. Il évolue au cours du jeu et peu apprendre Jusqu'à deux sortes de magie parmi les quatre qu'on lui proposera (Glace, Feu, Foudre, Soin, Psy).
A chaque nouvelle partie, il garde l'expérience accumulé. Dans les précédentes. Il aura sur lui un Talisman qui lui permettra d'oublier ses magies, cependant, il ne pourra l'utiliser que deux fois. Il Devra en trouver d'autre ou s'en acheter une s'il le peut.

Personnage Secondaire :
Prisonnier des monstres de la zone 3, il est délivré par le Personnage Principal. Ils deviendront alors alliés dans cette aventure. Personnages Secondaire détient les mîmes Aptitude que Personnage Principal. L'exception des magies : il n'en poss.de qu'une. Sa trousse est Aussi plus petite que celle de Personnage principal.

LA VOIX:
Aucune information n'est connue à propos de ce personnage mystique. On le trouve dans La zone 0 et la zone 11.

Magicien :
Prisonnier des monstres de la zone 5, il apprendra l'art de la magie. Personnage Principal et Personnage Secondaire dans la zone 0. Une fois que ceux-ci auront appris la magie, il Disparaitra et réapparaîtra lors du combat final contre Octopussy pour prêter main forte. Ses Disciple. Il mourra. La fin du combat.

Gardien :
Il garde l'entrée de la zone 3 et ne l'ouvrira qu'avec une coquette somme d'argent. Après
L'ouverture on le retrouvera dans certaine pièce du jeu. Nous donner des conseils.
Lorsque Personnage Principal atteindra son dernier niveau et vaincra le dernier boss de la zone 13,
Le gardien lui demandera s'il est prêt. Affronter le véritable maitre des lieux. Si Personnage Principal accepte, le gardien se transformera en un monstre incroyable, Octopussy, la légende.

Détails des 13 ZONES

Zone 0 :

Départ de partie, Vendeur (1,0), gardien (0), coffre0. Zone d'arrivée. Le personnage principal peut effectuer plusieurs actions avant d'avancer vers les Zones de combat.

Zone 3 :

Abrite un Boss et le Personnage Secondaire. Une fois battu, le Personnage Secondaire suivra le Personnage Principal dans ses aventures.

Zone 5 :

Abrite un Boss et le magicien. Une fois le magicien délivré., il sera possible de dialoguer et Apprendre la magie directement à partir de la zone 0.

Zone 8 :

Abrite un Boss.

Zone 11 :

Check point permettant de faire le plein de potion, armure, etc. Il sera possible d'acheter le code Permettant de déverrouiller le Digicode1.

Zone 10, 12 :

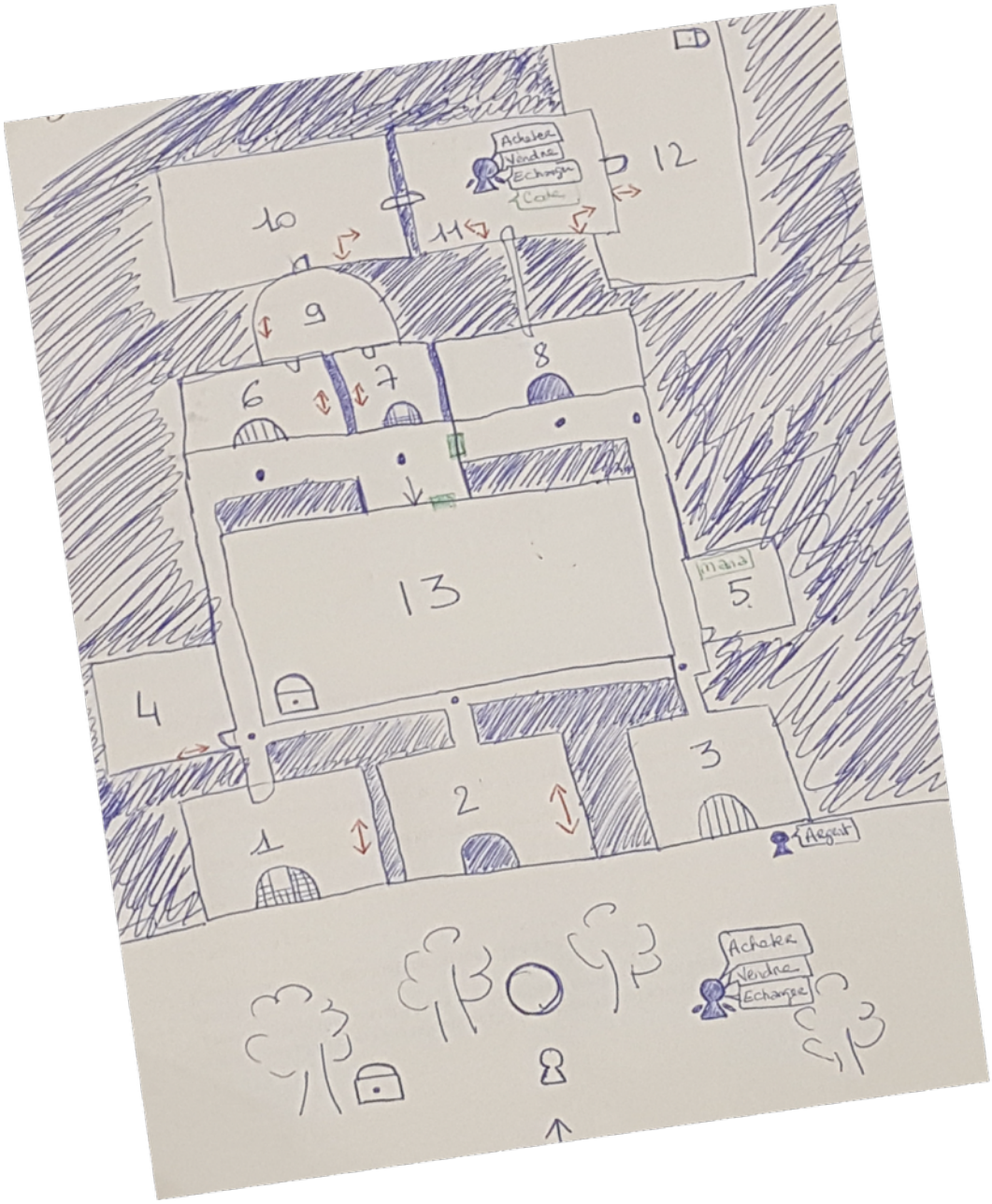
Royal Rumble. Attention une fois les combats début., il ne sera pas possible de quitter la zone sans Devoir recommencer le Royal Rumble depuis le début. La zone 12 est beaucoup plus difficile que la 10 et une fois terminé., elle permettra de débloquer le code délivrant l'accès La salle 13. La Zone 12 ne sera accessible qu'à partir d'un certain niveau et le coffre qui s'y trouve ne pourra être ouvert Qu'en la possession d'un artefact que l'on pourra uniquement avoir en battant le Boss de la Zone 13.

Zone 1, 2, 4, 6, 7, 9 :

Abrite des monstres. A chaque fin de jeu, le nombre de monstre et leurs niveaux augmentent. Il faudra donc parfaitement maitriser son staff.

Zone 13 : Bonne de fin de partie

PLAN


Acheter
Vendre
Echanger
Café
12
10
11
9
8
7
6
13
5
4
3
2
1
Accueil
Acheter
Vendre
Echanger
8

QUESTION 7.18.7

ActionListener est l'interface qui écoute et attend la réception d'un action event.
La classe qui s'intéresse au traitement d'un action event implémente cette interface et l'objet crée avec cette classe est enregistré avec un composant à l'aide de la méthode **addActionListener** de ce composant.
Lorsque l'action event se produit, la méthode **actionPerformed** de cet objet est appelée, qui est chargée d'exécuter les instructions souhaitées.

QUESTION 7.19 -(mvc)

Tout d'abord le MVC(Model-View-Controller) , est un motif d'architecture logicielle destiné aux interface graphiques lancé en 1978 et tres populaire pour les application web.
le MVC est composé de ces 3 éléments:

- -Le modèle
- -La vue
- -Le contrôleur

ce motif est utilisé par de nombreux frameworks pour les applications web tels que Ruby , Symfony, Apache Tapestry.
En d'autre terme le MVC permet de séparé la gestion de donnée , l'interface graphique et l'interprétation des commandes.

QUESTION 7.21

les room sont responsable des items,
Les informations contenues dans un item présent dans une room devrait être gérées par ce dernier.
La description des items sont envoyé par les rooms, idem pour sa propre description(room), tous cela sont affichés dans les GameEgine.goRoom().

Lorsque l'on se déplace avec goRomm(), on affiche les informations sur la nouvelle room (exits). Les informations sur l'item sont ainsi traitées comme une prolongation des informations sur la room.

QUESTION 7.24 et 7.25

Nous aurons un retour "back what ?" si on tape back et autre chose (y compris back).

QUESTION 7.28.1

Dans GameEngine,java :

```
/**
* Given a command, process (that is: execute) the command.
* If this command ends the game, true is returned, otherwise false is
* returned.
* @param commandLine The commande to be processed.
*/

public void interpretCommand(String commandLine)
{
gui.println(commandLine);
Command command = parser.getCommand(commandLine);
if(command.isUnknown()) {
gui.println("I don't know what you mean...");
return;
}


String commandWord = command.getCommandWord();
if (commandWord.equals("help"))
printHelp();
else if (commandWord.equals("go"))
goRoom(command);
else if (commandWord.equals("look"))
look();
else if(commandWord.equals("eat"))
eat();
else if(commandWord.equals("back")) {
if(command.hasSecondWord())
gui.println("Back what ?");
else
back();
}

else if(commandWord.equals("test")) {
if(!command.hasSecondWord())
```

```
gui.println("test what ?");
else
test(command);
}

else if (commandWord.equals("quit")) {
if(command.hasSecondWord())
gui.println("Quit what?");
else
endGame();
}

}
```

Ajout d'un else if pour tester si la commande égale « test ». On appelle alors la fonction test :

```
/**
* Read a file and execute the commands.
* @param command the command whith the file name in it.
*/

private void test(Command command) {
String f = command.getSecondWord();
FileReader ff;
BufferedReader br;
try {
ff = new FileReader(f);
br = new BufferedReader(ff);
String line;
while ((line = br.readLine()) != null) {
interpretCommand(line);
}
br.close();
}
catch(FileNotFoundException e) {
gui.println("File not found");
}
catch(IOException e) {
```

```
e.printStackTrace();
}

}
```

On lit le fichier ligne par ligne et on envoie chaque ligne à interpretCommand.

QUESTION 7.29, 7.30, 7.31 :

Voici la classe Player rajoutée au jeu.

```
import java.util.HashMap;

/**
* This class is part of the "World of Zuul" application.
* "World of Zuul" is a very simple, text based adventure game.
*
* This class creates all rooms, creates the parser and starts
* the game. It also evaluates and executes the commands that
* the parser returns.
*
* @author Michael Kolling and David J. Barnes
* @version 1.0 (Jan 2003)
*/

public class Player {
private Room currentRoom;
private HashMap<Item,String> items;
public Player() {
currentRoom = null;
items = new HashMap<Item, String>();
}

/**
* @return the current room
*/

public Room getCurrentRoom() {
return currentRoom;
```

```
}

/**
* Set the new current room.
* @param room the new current room.
*/


public void setCurrentRoom(Room room) {
this.currentRoom = room;
}


/**
* @return the item list.
*/
public HashMap<Item,String> getItemsList() {
return items;
}


/**
*take an item
* @param item the item to take.
*/

public void takeItem(Item item) {
items.put(item,"none");
}

/**
* drop an item.
*/

public void dropItem(Item item) {
items.remove(item);
}

/**
* Print the items of the player.
```

```
*/

public String getItemsInfo() {
if(items.isEmpty())
return "";
StringBuilder returnString = new StringBuilder( "Player's Items :\n" );
for(Item it : items.keySet())
returnString.append(" "+ it.getDescription()+"\n");
return returnString.toString();
}
}
```

Nous avons la currentRoom, une liste d'items, et des méthodes utiles comme
dropItem, takeItem ou encore getItemsInfo pour afficher les items que le player a
récupéré.
Voici les commandes que nous avons rajoutés dans GameEngine.java :

```
/**
* Print all the items of the player.
*/

private void printItemsList() {
String it = player.getItemsInfo();
gui.println(it);
}

/**
*
* @param command the command with item's name in it
*/
private void take(Command command) {

String i = command.getSecondWord();
HashMap<Item,String> items = player.getCurrentRoom().getItemsList();
for(Item it : items.keySet()) {
if(i.equals(it.getName())) {//Si l'item qu'on veut prendre est bien dans la pièce
player.takeItem(it);
```

```
player.getCurrentRoom().removeItem(it);
printItemsList();
return;
}

}
gui.println("No item with this name");
}

/**
* Drop the item of the player
*/
private void drop(Command command) {
String i = command.getSecondWord();
HashMap<Item, String> tmp = player.getItemsList();
for(Item it : tmp.keySet()) {
if(i.equals(it.getName())) {
player.getCurrentRoom().addItem(it);
player.dropItem(it);
return;
}

}
gui.println("No item with this name");
}
```

Lorsque l'on prend un item, il faut le rajouter à la liste d'items du player mais aussi
l'enlever de la room.
Pour respecter au mieux le principe de responsability-driven design, c'est à la classe
GamEngine de faire la liaison entre les classes Item, Room et Player.
Voici la méthode interpretCommand actualisée :

```
/**
* Given a command, process (that is: execute) the command.
* If this command ends the game, true is returned, otherwise false is
* returned.
* @param commandLine The command to be processed.
*/
```

```
public void interpretCommand(String commandLine)
{
gui.println(commandLine);
Command command = parser.getCommand(commandLine);
if(command.isUnknown()) {
gui.println("I don't know what you mean...");
return;
}

String commandWord = command.getCommandWord();
if (commandWord.equals("help"))
printHelp();
else if (commandWord.equals("go"))
goRoom(command);
else if (commandWord.equals("look"))
look();
else if(commandWord.equals("eat"))
eat();
else if(commandWord.equals("back")) {
if(command.hasSecondWord())
gui.println("Back what ?");
else
back();
}

else if(commandWord.equals("test")) {
if(!command.hasSecondWord())
gui.println("test what ?");
else
test(command);
}

else if(commandWord.equals("take")) {
if(!command.hasSecondWord())
gui.println("take what ?");
else
take(command);
}

else if(commandWord.equals("drop")) {
if(!command.hasSecondWord())
```

```
gui.println("drop what ?");
else
drop(command);
}

else if (commandWord.equals("quit")) {
if(command.hasSecondWord())
gui.println("Quit what?");
else
endGame();
}
}
```

QUESTION 7.31.1 :

Voici la nouvelle classe ItemList avec des méthodes utiles :

```
import java.util.HashMap;
import java.util.Set;
public class ItemList {
private HashMap<Item,String> items;

/**
* Constructor.
*/

public ItemList() {
items = new HashMap<Item, String>();
}

/**
* Get all the items.
* @return the items.
*/

public HashMap<Item,String> getItemList() {
return items;
}

/**
* Push an item into the HashMap.
```

```
* @param item the item to push.
*/

public void pushItem(Item item) {
items.put(item,"none");
}

/**
* delete an item
* @param item the item to remove
*/

public void removeItem(Item item) {
items.remove(item);
}

/**
* @return true if the HashMap is empty, false eitherway.
*/

public boolean emptyDumpty() {
return items.isEmpty();
}
}
```

QUESTION 7.32 :

Ajout dans Player,java de deux attributs.
PoidsMax, et currentPoid le poid actuel des items du player.

```
private int poidsMax;
private int currentPoid;
```

Au moment de prendre un item, on regarde si la somme du poid de cet item et du
poid actuel du Player est inférieur ou égal au poid max.
Si c'est le cas on l'ajoute à la liste d'items du player

```
/**
*take an item
* @param item the item to take.
*/

public boolean takeItem(Item item) {
```

```
int tmp = currentPoid + item.getWeight();
if(tmp <= poidsMax) {
items.put(item,"none");
currentPoid += item.getWeight();
return true;
} else
return false;
}
```

La méthode renvoie true si elle peut prendre l'item, false sinon.
Elle est appelée par la méthode take dans GameEngine,java.

```
/**
*
* @param command the command with item's name in it
*/


private void take(Command command) {
String i = command.getSecondWord();
HashMap<Item,String> items = player.getCurrentRoom().getItemsList();
for(Item it : items.keySet()) {
if(i.equals(it.getName())) {//Si l'item qu'on veut prendre est bien dans la pièce
boolean tmp = player.takeItem(it);
if(tmp == true)
player.getCurrentRoom().removeItem(it);
else
gui.println("Too heavy");
return;
}
}



gui.println("No item with this name");
}
```

Si on peut prendre l'item, donc que takeItem a renvoyé true, alors on enlève l'item de
la room.

QUESTION 7.33 :

Voilà la partie rajoutée dans interpretCommands :

```
else if(commandWord.equals("items")) {
if(command.hasSecondWord())
gui.println("print what ?");
else
printItemsList();
}
```

Et voilà la fonction printItemsList() :

```
/**
* Print all the items of the player.
*/

private void printItemsList() {
String it = player.getItemsInfo();
gui.println(it);
}
```

Ainsi que getItemsInfo() :

```
/**
* return the items of the player.
*/

public String getItemsInfo() {
if(items.isEmpty())
return "";
StringBuilder returnString = new StringBuilder( "Player's Items :\n" );
for(Item it : items.keySet())
returnString.append(" "+ it.getDescription()+"\n");
return returnString.toString();
}
```

7.34 :

Le magic_cookie est un item comme un autre avec un poid de 0.

On peut le prendre, et une fois pris on peut le manger avec la commande eat qui ne
fait plus qu'afficher « you have eaten ».

```
/**
* Look if the player have a magic_cookie.
* Eat it.
*/

private void eat() {
String i = "magic_cookie";
HashMap<Item, String> tmp = player.getItemsList();
for(Item it : tmp.keySet()) {
if(i.equals(it.getName())) {
player.increaseMaxWeight();
player.dropItem(it);
return;
}
}

gui.println("No magic_cookie found.");
}
```

Elle récupère la HashMap d'Items du Player, parcours tous les Items.
Si l'un d'eux a pour nom « magic_cookie » alors elle appelle la méthode increaseMaxWeight puis drop l'item.

```
/**
* Increase the max weight by 50;
*/

public void increaseMaxWeight() {
poidsMax += 50;
}
```

QUESTION 7.34.1 :

ideal_path.txt :

```
go north
go north
go north
go east
```

go up
allRoom,txt :
go north
go north
go west
back
go north
go north
go south
go east
go north
look
go north
go north
go east
go east
back
go south
go south
go south
go south
go south
back
back
take chocolate_rain
look
go east
take clef
take LightBringer
items
back
go north
take magic_cookie
go south
go east
take clef
eat
eat
look
take clef
back

go north
go west
go up

QUESTION 7.35, 7.35.1 :

Voilà la nouvelle classe CommandWord dans CommandWord.java :

```
/**
* Representations for all the valid command words for the game.
*
* @author Michael Kolling and David J. Barnes
* @version 2006.03.30
*/

public enum CommandWord
{
// A value for each command word, plus one for unrecognised
// commands.
GO, QUIT, HELP, LOOK, BACK, EAT, TAKE, DROP, ITEMS, TEST,
UNKNOWN;
}
```

Voici la nouvelle méthode interpretCommand :

```
/**
* Given a command, process (that is: execute) the command.
* If this command ends the game, true is returned, otherwise false is
* returned.
* @param commandLine The command to be processed.
*/



public void interpretCommand(String commandLine)
{
Command command = parser.getCommand(commandLine);
//gui.println(commandLine);
CommandWord commandWord = command.getCommandWord();
if(commandWord == CommandWord.UNKNOWN) {
gui.println("I don't know what you mean...");
```

```
return;
}

//String commandWord = command.getCommandWord();
if (commandWord == CommandWord.HELP)
printHelp();
else if (commandWord == CommandWord.GO)
goRoom(command);
else if (commandWord == CommandWord.LOOK)
look();
else if(commandWord == CommandWord.EAT)
eat();
else if(commandWord == CommandWord.BACK) {
if(command.hasSecondWord())
gui.println("Back what ?");
else
back();
}

else if(commandWord == CommandWord.TEST) {
if(!command.hasSecondWord())
gui.println("test what ?");
else
test(command);
}

else if(commandWord == CommandWord.TAKE) {
if(!command.hasSecondWord())
gui.println("take what ?");
else
take(command);
}


else if(commandWord == CommandWord.DROP) {
if(!command.hasSecondWord())
gui.println("drop what ?");
else
drop(command);
}
```

```
else if(commandWord == CommandWord.ITEMS) {
if(command.hasSecondWord())
gui.println("print what ?");
else
printItemsList();
}

else if (commandWord == CommandWord.QUIT) {
if(command.hasSecondWord())
gui.println("Quit what?");
else
endGame();
}
}
```

QUESTION 7.35.2 :

Voici la nouvelle fonction interpretCommand :

```
/**
* Given a command, process (that is: execute) the command.
* If this command ends the game, true is returned, otherwise false is
* returned.
* @param commandLine The command to be processed.
*/

public void interpretCommand(String commandLine)
{
Command command = parser.getCommand(commandLine);
//gui.println(commandLine);
CommandWord commandWord = command.getCommandWord();
switch(commandWord) {
case UNKNOWN:
gui.println("I don't know what you mean...");
return;
case HELP:
printHelp();
break;
case GO:
goRoom(command);
break;
```

```
case LOOK:
look();
break;
case EAT:
eat();
break;
case BACK:
if(command.hasSecondWord())
gui.println("Back what ?");
else
back();
break;
case TEST:
if(!command.hasSecondWord())
gui.println("test what ?");
else
test(command);
break;
case TAKE:
if(!command.hasSecondWord())
gui.println("take what ?");
else
take(command);
break;
case DROP:
if(!command.hasSecondWord())
gui.println("drop what ?");
else
drop(command);
break;
case ITEMS:
if(command.hasSecondWord())
gui.println("print what ?");
else
printItemsList();
break;
case QUIT:
if(command.hasSecondWord())
gui.println("quit what ?");
else
endGame();
```

```
}
}
```

QUESTION 7.36 :

Il n'y avait rien à faire la commande look existait déjà :

```
/**
* Print informations about the room.
*/
private void look()
{
gui.println(player.getCurrentRoom().getLongDescription());
}
```

Voici la méthode getLongDescription :

```
/**
* Return a long description of this room, of the form:
* You are in the kitchen.
* Exits: north west
* @return A description of the room, including exits.
*/

public String getLongDescription()
{
return "You are in the " + description + ".\n" + getExitString() + "\n" +
getItems();
}
```

getItems et getExitString :

```
/**
* Return the list of items,
* @return A list of the items.
*/
public String getItems() {
if(items.isEmpty())
return "";
StringBuilder returnString = new StringBuilder( "Items:\n" );
for(Item it : items.keySet()) {
returnString.append(it.getDescription()+"\n");
}
```

```
return returnString.toString();
}

/**
* Return a description of the room's exits,
* for example, "Exits: north west".
* @return A description of the available exits.
*/

public String getExitString() {
StringBuilder returnString = new StringBuilder( "Exits:" );
for(String exit : exits.keySet())
returnString.append(" "+ exit);
return returnString.toString();
}
```

QUESTION 7.37 :

Modification du constructeur de CommandWords :
Ajout d'un d à la fin de chaque commande pour tester.
Nous n'avons pas seulement fait sur go et quit.

```
/**
* Constructor - initialise the command words.
*/

public CommandWords()
{
validCommands = new HashMap<String, CommandWord>();
validCommands.put("god", CommandWord.GO);
validCommands.put("helpd", CommandWord.HELP);
validCommands.put("quitd", CommandWord.QUIT);
validCommands.put("lookd", CommandWord.LOOK);
validCommands.put("eatd", CommandWord.EAT);
validCommands.put("backd", CommandWord.BACK);
validCommands.put("testd", CommandWord.TEST);
validCommands.put("taked", CommandWord.TAKE);
validCommands.put("dropd", CommandWord.DROP);
validCommands.put("itemsd", CommandWord.ITEMS);
}
```

Nous n'avons pas eu à modifier autre chose.
Ce n'est cependant pas suffisant pour traduire le jeu car il reste les direction qui sont
en anglais (east,north…).

QUESTION 7.38 :

Le message nous dit d'entrer help pour de l'aide et pas la nouvelle commande.

QUESTION 7.39 :

Voilà l'enum :

```
public enum Position
{
TOP,MIDDLE,BOTTOM
}
```

Mais nous ne savons pas quoi en faire
.

QUESTION 7.40 :

Nous avons rajoutés LOOK à CommandWord,java :

```
GO("go"), QUIT("quit"), HELP("help"), LOOK("look"), UNKNOWN("?");
```

Ensuite il a fallu rajouter look à processCommand et adapter la méthode

```
look(System.out.println au lieu de gui.println
else if (commandWord == CommandWord.LOOK) {
look();
}
/**


* Print informations about the room.
*/
private void look()
{
System.out.println(currentRoom.getLongDescription());
```

```
}
```

Nous avons également apportés les modifications discutées dans la section 7.13.2 à notre propre projet :

```
/**
* Representations for all the valid command words for the game
* along with a string in a particular language.
*
* @author Michael Kolling and David J. Barnes
* @version 2006.03.30
*/`

public enum CommandWord
{
// A value for each command word along with its
// corresponding user interface string.
GO("go"), TAKE("take"), TEST("test"), DROP("drop"), LOOK("look"),
EAT("eat"), ITEMS("items"),
BACK("back"), QUIT("quit"), HELP("help"), UNKNOWN("?");
// The command string.
private String commandString;

/**
* Initialise with the corresponding command word.
* @param commandWord The command string.
*/
CommandWord(String commandString)
{
this.commandString = commandString;
}
/**
* @return The command word as a string.
*/


public String toString()
{
return commandString;
}
}
```

Voilà le nouveau CommandWord.java.
Ainsi le constructeur de CommandWords.java a pu être modifier de la sorte :

```
/**
* Constructor - initialise the command words.
*/

public CommandWords()
{
validCommands = new HashMap<String, CommandWord>();
for(CommandWord command : CommandWord.values()) {
if(command != CommandWord.UNKNOWN)
validCommands.put(command.toString(), command);
}
}
```

QUESTION 7.41 :

Oui la bonne commande est affichée au début du jeu grâce à

printWelcome :
System.out.println("Type '" + CommandWord.HELP + "' if you need help.");

Nous avons donc rajouté cela à notre projet.

ITERATION 4

QUESTION 7.42 :

Pour cette question nous comptons le nombre de mouvements.

```
private int stepsMax;
private int currentSteps;
```

Voilà les deux attributs ajoutés à la classe GameEngine.
On fixe un nombre de mouvements et on garde un compteur de mouvements qui
commence à 0.
Dans la méthode interpretCommand, on rajoute ça à la fin :

```
currentSteps++;
if(currentSteps == stepsMax)
endGame();
```

QUESTION 7.43, 7.45 :

Les deux questions ont été faites d'un coup.
Nous avons une classe Door que l'on peut verrouiller. Il faut un item avec un nom
particulier (le nom est stocké comme attribut de la porte) pour déverrouiller une
Door.
Pour la porte qui ne peut être franchie, vu que les portes sont contenues dans les
Rooms en fonction de leurs directions, il suffit de placer une porte dans une room, et
de ne pas en mettre dans l'autre sens :
Room1 -> Door Room2
Ici, la Door a été mise dans Room1 vers Room2, mais pas dans Room2 vers Room1.
Voici la classe Door :

```
/**
```

```
* Class Door - a door in an adventure game.
*
* This class is part of the "World of Zuul" application.
* "World of Zuul" is a very simple, text based adventure game.
*
* A "Door" represents something in the scenery of the game.
*
* @author LUKAKU KATANU TRESOR
* @version 2019.03.6
*/

public class Door {
boolean locked;
String toUnlock;//the item's name to unlock.
/**
* set the door's lock.
* @param b the boolean if locked.
*/

public Door(String toUnlock) {
this.locked = true;
this.toUnlock = toUnlock;
}

/**
* verify if the door is locked.
* @return locked.
*/
public boolean ifLocked() {
return locked;
}

/**
* get the key.
* @return toUnlock the key's name.
*/
public String getKey() {
return toUnlock;
}
/**
* set the door's lock
```

```
* @param b the lock's setting.
*/

public void setLock(boolean b) {
locked = b;
}
}
```

Les doors sont stockées dans les rooms avec une HashMap.
La Door est sa direction cardinale.
Voilà les deux méthodes ajoutées à la classe Room :

```
/**
* Define the doors of this room. Every direction can have a door
* @param direction The direction of the door.
* @param neighbor The door in the given direction.
*/

public void setDoors(String direction, Door door)
{
doors.put(direction,door);
}

/**
* get the door in the given direction
* @param direction the direction.
*/

public Door getDoor(String direction) {
return doors.get(direction);
}
```

QUESTION 7.44 :

Le téléporteur est une nouvelle classe qui hérite de la classe Item :

```
/**
* Class Beamer - an item in an adventure game.
*
* This class is part of the "World of Zuul" application.
*  "World of Zuul" is a very simple, text based adventure game.
```

```
*
* A "Beamer" represents something in the scenery of the game.
*
* @author LUKAKU KATANU TRESOR
* @version 2019.03.6
*/
public class Beamer extends Item {
private boolean loaded;
private Room loadedRoom;

/**
* Create a beamer described "description".
* @param description The beamer's description.
* @param poid The weight of the beamer.
*/

public Beamer(String description, int poid)
{
super(description,poid);
this.loaded = false;
this.loadedRoom = null;
}

/**
* load the beamer.
*/
public void load(Room room) {
loaded = true;
loadedRoom = room;
}

/**
* fire : teleport to the loadedRoom.
*/

public Room fire() {
loaded = false;
Room tmp = loadedRoom;
loadedRoom = null;
return tmp;
}
```

```
/**

* verify if the beamer is loaded
* @return loaded the boolean.
*/
public boolean ifLoaded() {
return loaded;
}
}
```

Voici les méthodes rajoutées dans GameEngine.java pour gérer le téléporteur :

```
/**
* load the beamer if the player have it.
*/

public void load() {
String b = "beamer";
HashMap<Item, String> tmp = player.getItemsList();
for(Item it : tmp.keySet()) {
if(b.equals(it.getName())) {
Beamer tmp2 = new Beamer(it.getName(),it.getWeight());
tmp2.load(player.getCurrentRoom());
player.dropItem(it);
player.takeItem(tmp2);
return;
}
}

gui.println("You don't have a beamer");
}

/**
* fire the beamer if the player have it
* and the beamer is loaded.
*/
public void fire() {
String b = "beamer";
HashMap<Item, String> tmp = player.getItemsList();
for(Item it : tmp.keySet()) {
```

```
if(b.equals(it.getName())) {
Beamer tmp2 = (Beamer)it;
if(tmp2.ifLoaded()) {
Room r = tmp2.fire();
setPlayer(r);
printLocationInfo();
if(player.getCurrentRoom().getImageName() != null)
gui.showImage(player.getCurrentRoom().getImageName());
player.dropItem(it);
player.takeItem(tmp2);
}
else
gui.println("Load it if you want to fire it.");
return;
}
}

gui.println("You don't have a beamer");
}
```

Les commandes load et fire ont été rajoutées également.

QUESTION 7.46 :

La classe TransporterRoom hérite de la classe Room.
Elle téléporte le player dans l'une des rooms qui lui sont adjacente.

```
import java.util.Scanner;
import java.util.HashMap;
import java.util.Set;
import java.util.Iterator;
import java.util.ArrayList;
import java.lang.Math;

/**
* Class TransporterRoom - a room in an adventure game.
*
* This class is part of the "World of Zuul" application.
* "World of Zuul" is a very simple, text based adventure game.
*
```

```
* A "TransporterRoom" represents one location in the scenery of the game. It
is
* connected to other rooms via exits. The exits are labelled north,
* east, south, west. For each direction, the room stores a reference
* to the neighboring room, or null if there is no exit in that direction.
*
* @author Michael Kolling and David J. Barnes
* @version 2006.03.30
*/

public class TransporterRoom extends Room {
/**
* Create a room described "description". Initially, it has
* no exits. "description" is something like "a kitchen" or
* "an open court yard".
* @param description The room's description.
* @param image The image of the room.
*/

public TransporterRoom(String description, String image)
{
super(description,image);
}

/**
* Surcharge of the constructor.
* @param description The room's description.
* @param image The image of the room.
* @param
*
*/

public TransporterRoom(String description, String image, Item item)
{
super(description,image,item);
}
public Room teleport() {
Room nextRoom = null;
int nb = 0;
String direction="";
while(nextRoom == null) {
```

```
nb = (int) (Math.random()*7);
switch(nb) {
case 0:
direction = "north";
break;
case 1:
direction = "south";
break;
case 2:
direction = "east";
break;
case 3:
direction = "west";
break;
case 4:
direction = "up";
break;
case 5:
direction = "down";
break;
}
nextRoom = super.getExit(direction);
}
return nextRoom;
}
}
```

QUESTION 7.46.2

Pour la Door, la classe de base fait office de trap door et de locked door donc pas
besoin d'héritage.
Pour le beamer, nous avons déjà utiliser l'héritage pour en faire un extends de Item.

Cas d'utilisation ET Description textuelle

Titre : Gestion de stock d'un magasin

Résumé : Ce cas d'utilisation permet la consultation de la Gestion de stock d'un magasin
Acteurs : détaillant, système de gestion
Préconditions : que le système d'information marche déjà

Description nominale :

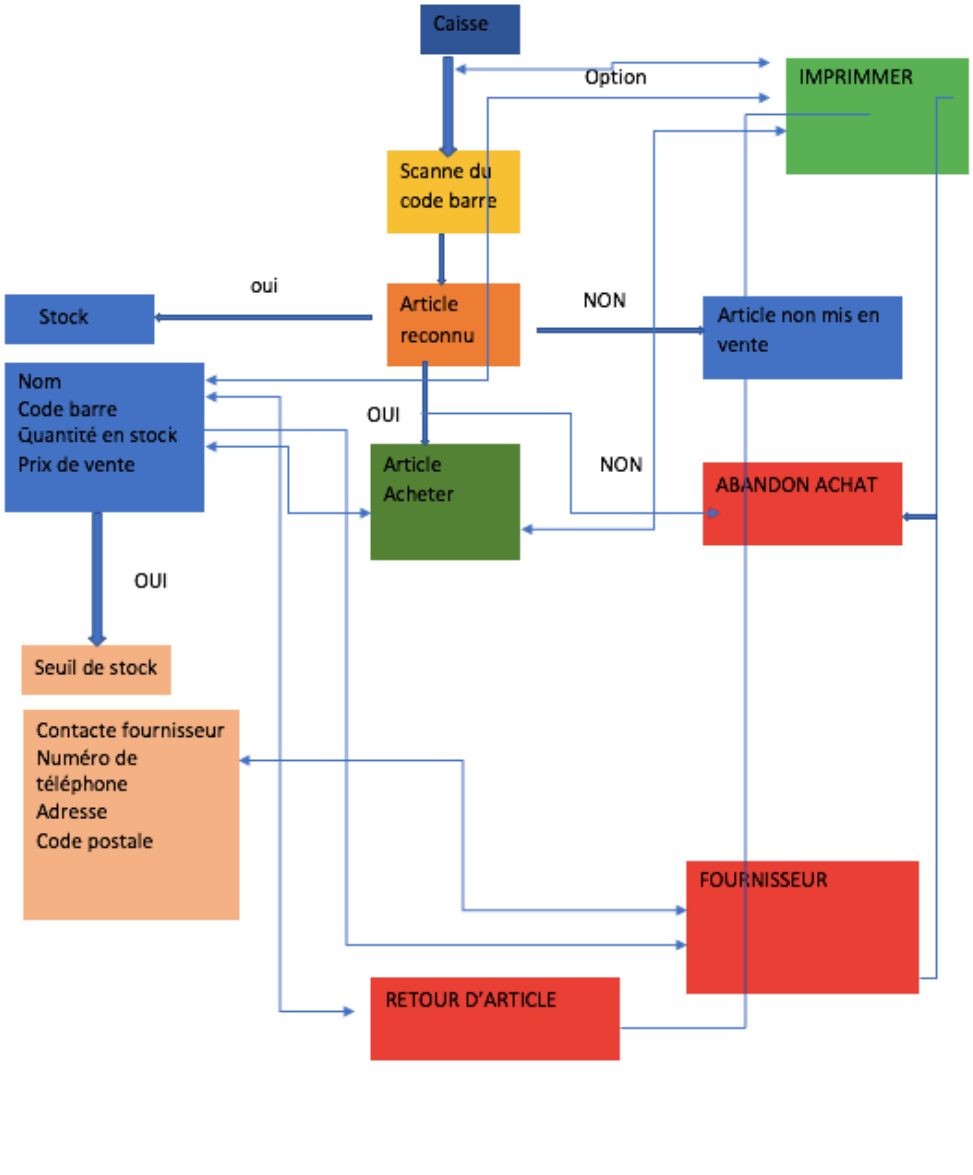
1.Le gestionnaire accède à la plateforme de gestion

2.Le gestionnaire peut afficher l'inventaire. Il peut ensuite :

- imprimer l'inventaire
- effacer un article
- éditer un article

3.Le gestionnaire peut ajouter un article dans l'inventaire

=> donc ajouter un fournisseur


Caisse
Option
IMPRIMMER
Scanne du code barre
oui
Stock
Article reconnu
NON
Article non mis en vente
Nom
Code barre
Quantité en stock
Prix de vente
OUI
Article Acheter
NON
ABANDON ACHAT
OUI
Seuil de stock
Contacte fournisseur
Numéro de téléphone
Adresse
Code postale
FOURNISSEUR
RETOUR D'ARTICLE

Cas d'utilisation 1 : Acheter un article

I.Acteurs

Client, Caissier

II.Pré-conditions

Aucune

III.Scénario nominal

1.Le client entre dans le magasin
2.Le client choisit l'article qu'il veut acheter
3.Le client paie
4.Le client sort du magasin

IV. Scénario alternatif

1b.Le client sort du magasin finalement
2b.Le client change d'article
3b.Le paiement est refusé

Cas d'utilisation 2 : Mise à jour du stock

I.Acteurs

Détaillant, système

II.Pré-conditions

Les articles doivent déjà être scannés, la caisse déjà connecté , et la liste des
Transactions déjà effectuées

III.Scénario nominal

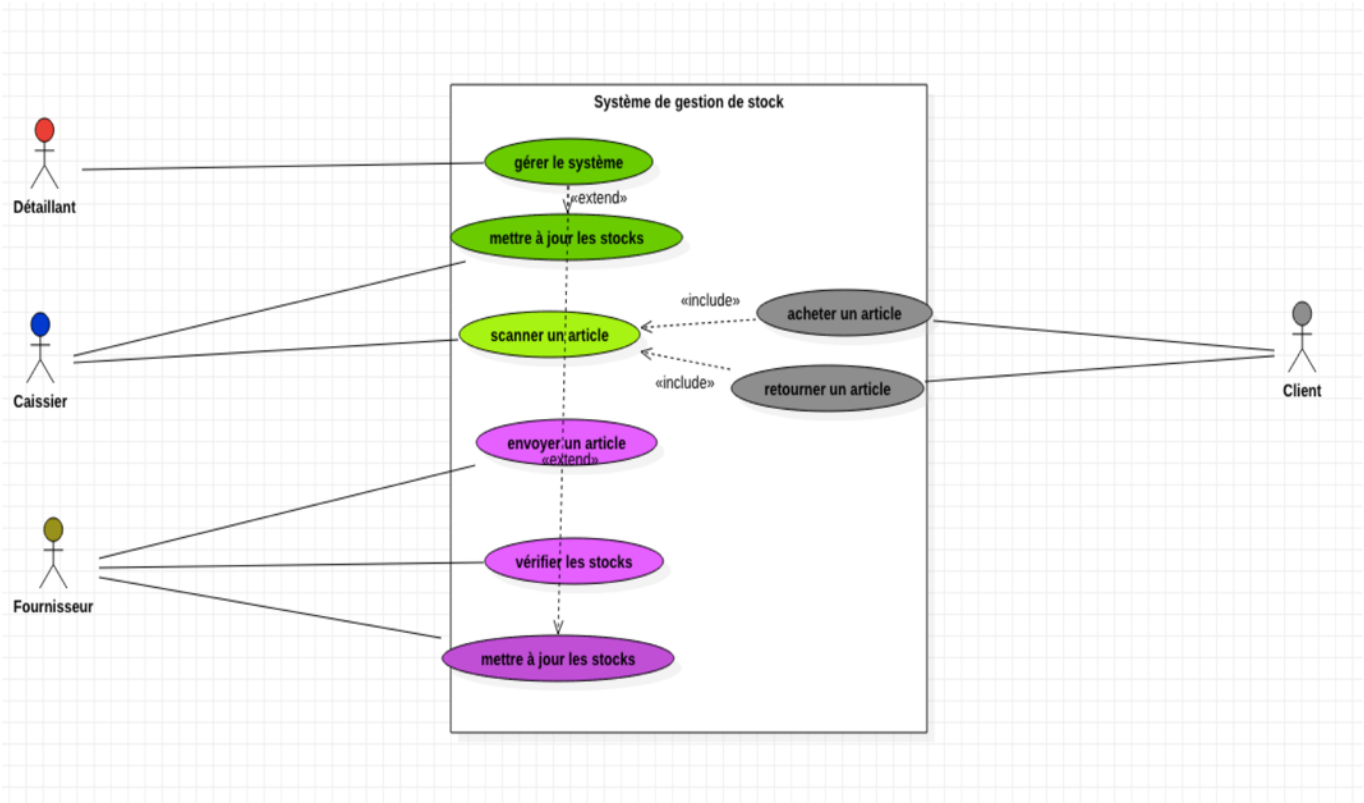
1.Le caissier se connecte à la caisse
2.Le caissier appuie sur le bouton total
3.Le caissier ferme la caisse
4.Les informations sont envoyé à la base de données
5.Le système met à jour le stock

IV.Scénario alternatif

5b.Le système incrémente le stock
5c.Le système décrémente le stock

Étape 3

1)
Les diagrammes de compositions indiqués concernent les cas d'utilisations suivant :
Acheter un article ,le client scanne un article



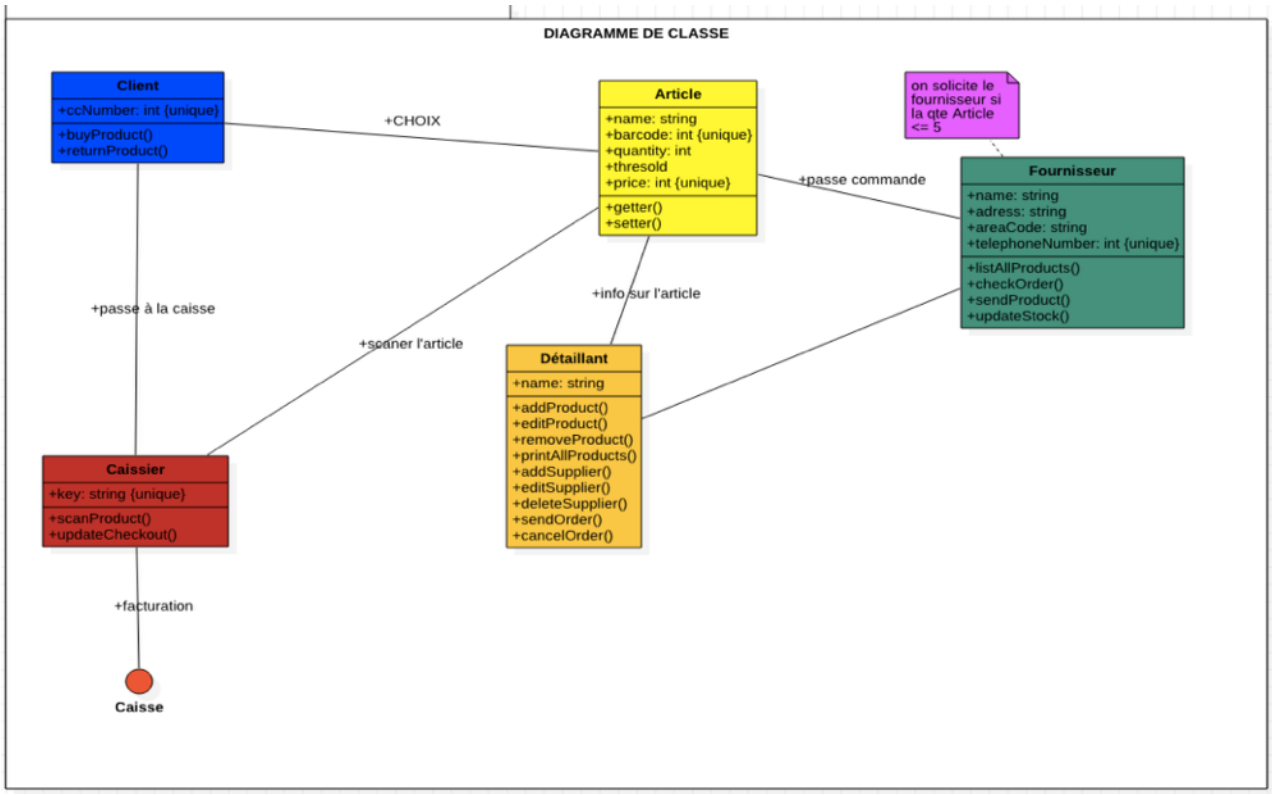
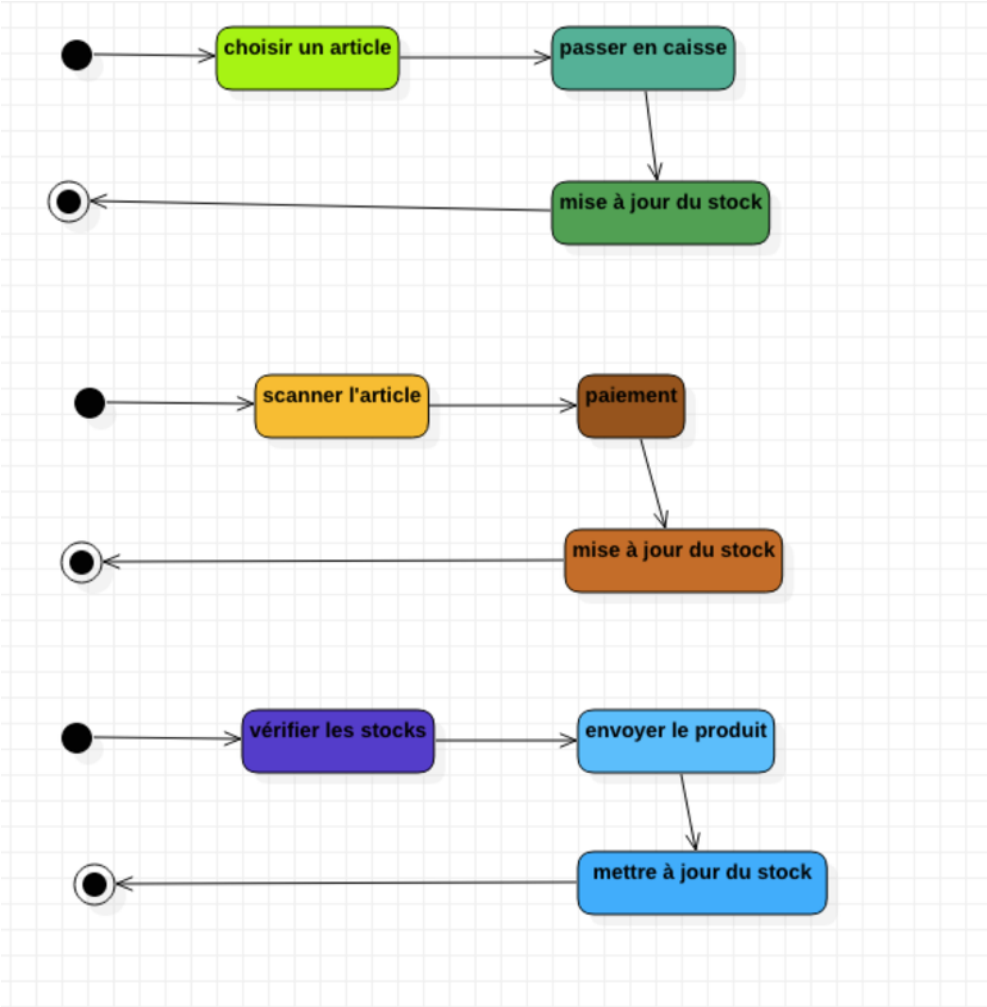
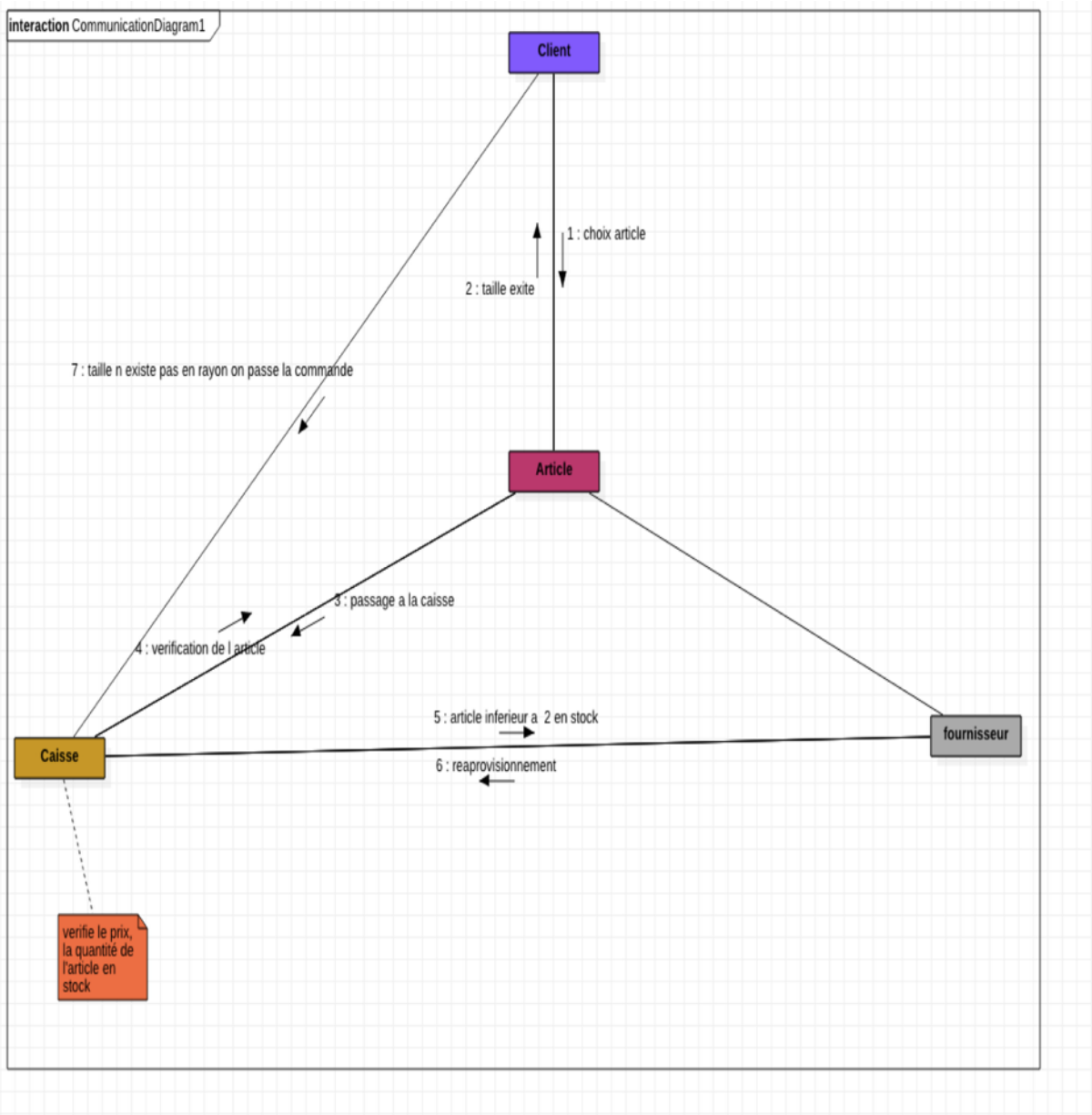
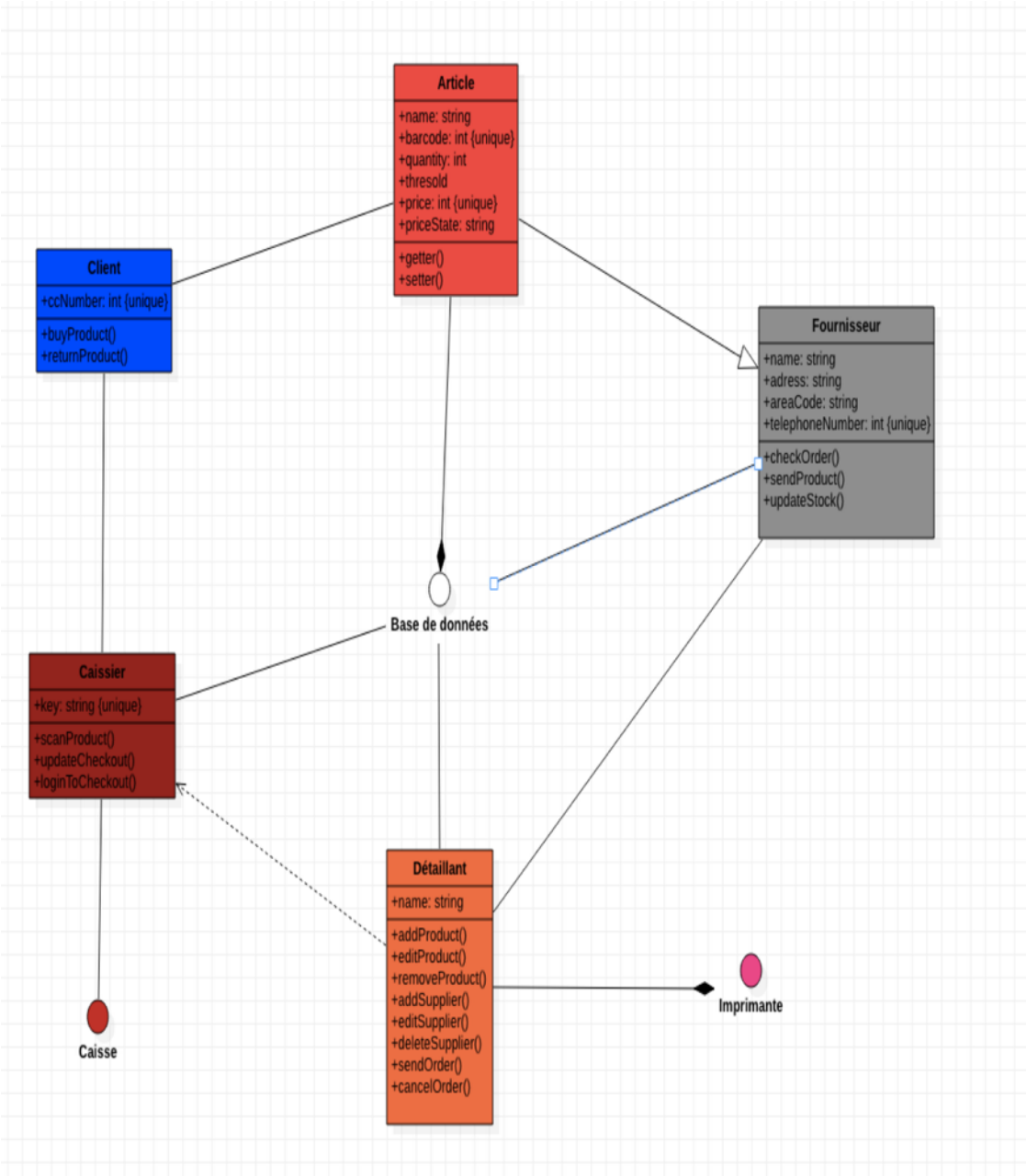


DIAGRAMME DE CLASSE
Client
+ccNumber: int {unique}
+buyProduct()
+returnProduct()
+CHOIX
Article
+name: string
+barcode: int {unique}
+quantity: int
+thresold
+price: int {unique}
+getter()
+setter()
on solicite le fournisseur si la qte Article <= 5
+passe commande
Fournisseur
+name: string
+adress: string
+areaCode: string
+telephoneNumber: int {unique}
+listAllProducts()
+checkOrder()
+sendProduct()
+updateStock()
+passe à la caisse
+info sur l'article
+scanner l'article
Détaillant
+name: string
+addProduct()
+editProduct()
+removeProduct()
+printAllProducts()
+addSupplier()
+editSupplier()
+deleteSupplier()
+sendOrder()
+cancelOrder()
Caissier
+key: string {unique}
+scanProduct()
+updateCheckout()
+facturation
Caisse


choisir un article
passer en caisse
mise à jour du stock
scanner l'article
paiement
mise à jour du stock
vérifier les stocks
envoyer le produit
mettre à jour du stock


interaction CommunicationDiagram1
Client
1 : choix article
2 : taille exite
7 : taille n existe pas en rayon on passe la commande
Article
3 : passage a la caisse
4 : verification de l article
5 : article inferieur a 2 en stock
fournisseur
Caisse
6 : reaprovisionnement
verifie le prix, la quantité de l'article en stock

Classe Final

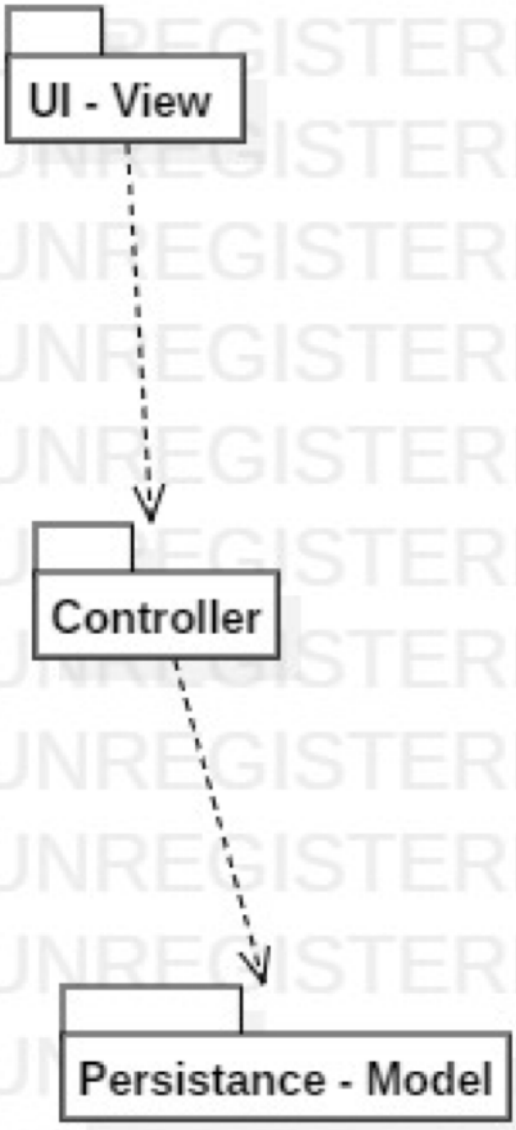


Phase de conception et Préliminaire

Étape 1 : Conception générale

1)

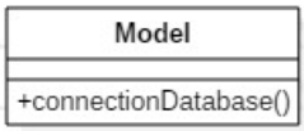
On détermine 3 paquetages principales , dont les dépendance sont résumé par le schéma ci-dessous



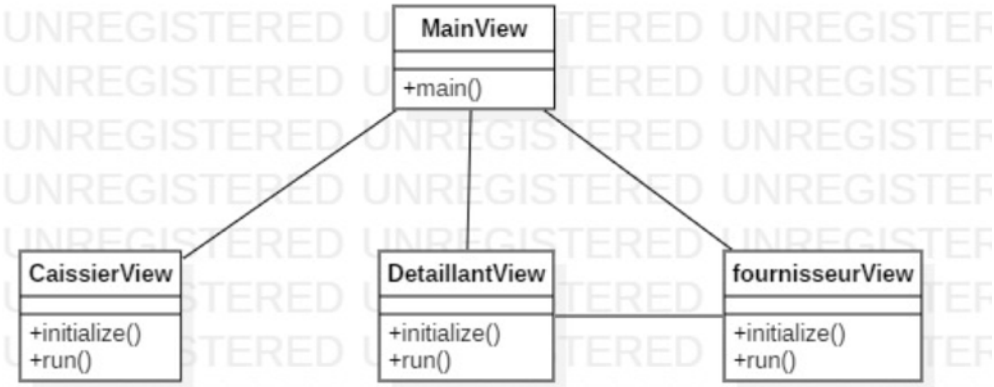
2)

Voici les diagrammes de classes de chaque paquetage

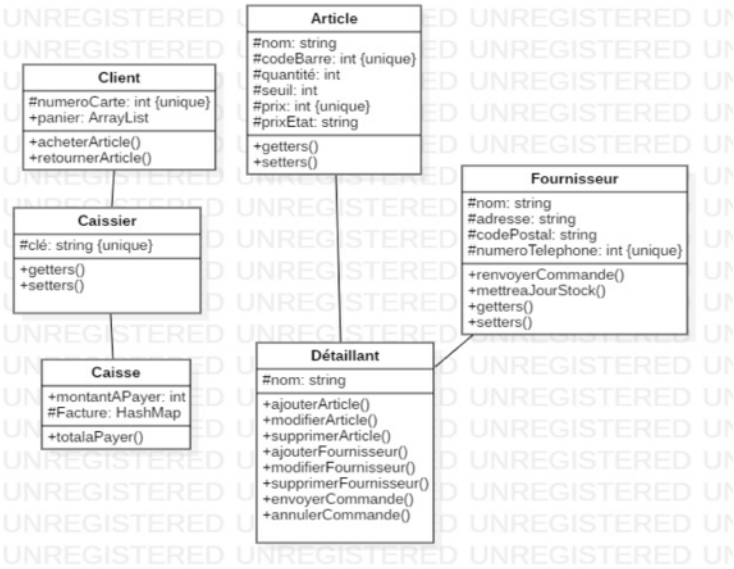
Model



View



Controller



Cas d'utilisation

Cas d'utilisation 1 : Acheter un article

I.Acteurs

Client,Caissier

II.Pré-conditions

Aucune

III.Scénario nominal

1.Le client entre dans le magasin
2.Le client choisit l'article qu'il veut acheter
3.Le client paie
4.Le client sort du magasin

IV.Scénario alternatif

1b.Le client sort du magasin finalement
2b.Le client change d'article
3b.Le paiement est refusé

Cas d'utilisation 2 : Mise à jour du stock

I.Acteurs

Détaillant , système

II.Pré-conditions

Les articles doivent être scannées , la caisse déjà connecté© , et la liste des transactions déjà effectué

III.Scénario nominal

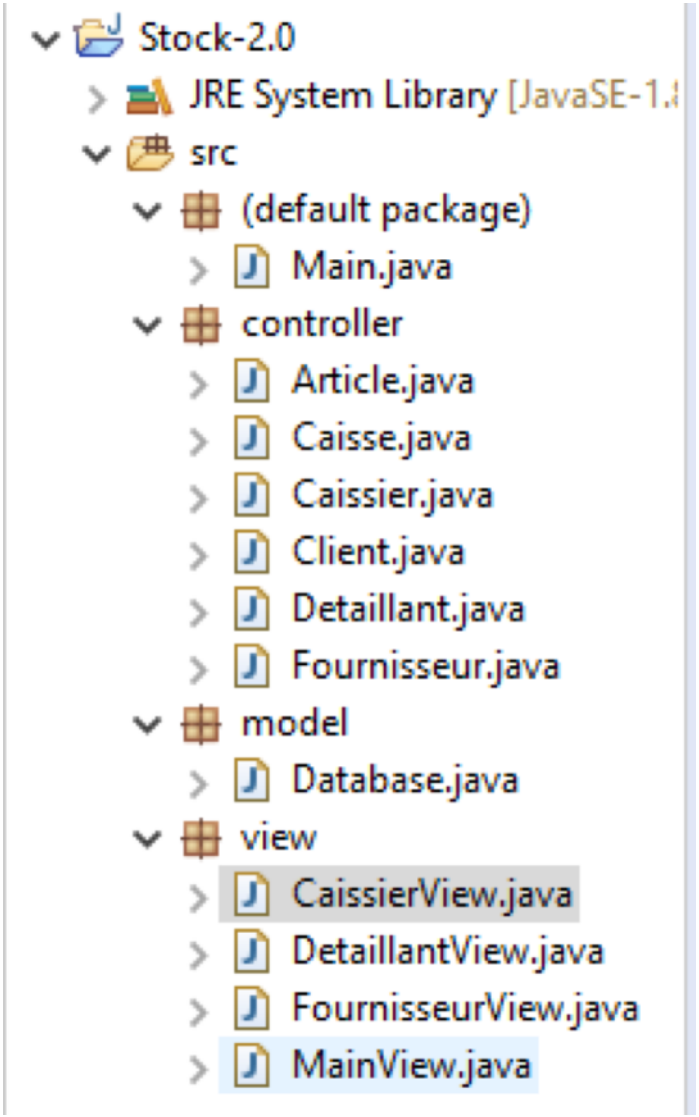
1.Le caissier se connecte Ã la caisse
2.Le caissier appuie sur le bouton total
3.Le caissier ferme la caisse
4.Les informations sont envoyés la base de données
5.Le système met à jour le stock

IV.Scénario alternatif

5b.Le système incémente le stock
5c.Le système décrémente le stock

2)

Voici l'arborescence du code et les différents packages



L'IHM est construit grace à Windowbuilder . Voici un exemple d'utilisation (ici la classe CaissierView,java)