Doctor under pressure (IPO 2021/2022 G12)

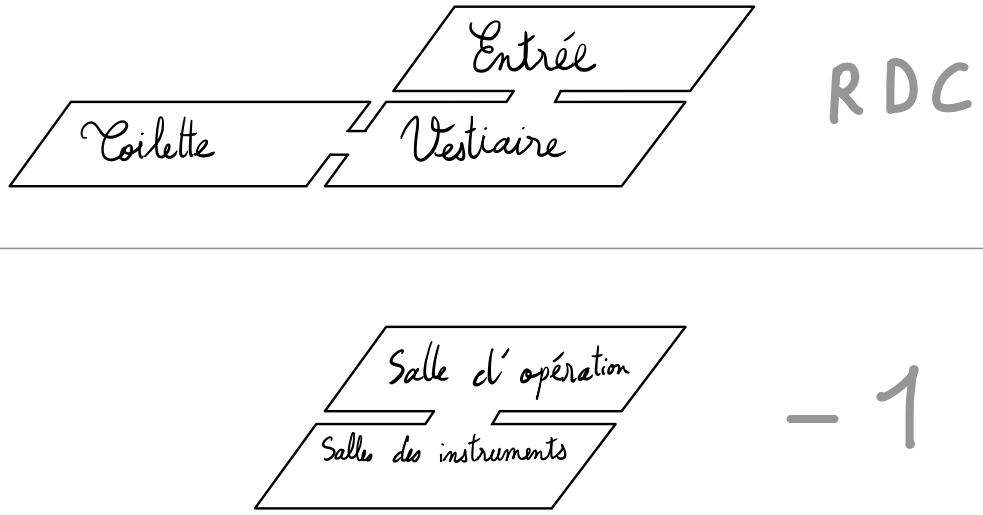
Supervisé par Denis BUREAU

I.A) Auteur : DAUVERGNE Hugo

I.B) Thème : Un médecin procède à une opération médicale

I.C) Résumé du scénario : Un médecin est appelé à opérer en urgence un patient en danger de mort. Mais tout ne se passe pas comme prévu… En effet, il manque des instruments nécessaires à l'opération et le personnel est épuisé.

I.D) Plan du jeu :



I.E) Scénario détaillé : Le chirurgien obstétrique Patrick, est appelé en urgence pour une césarienne. Néanmoins, quand il arrive à l'hôpital, il doit traverser le hall d'entrée où se trouve le secrétaire qui pose toute sorte de question. Après cela il doit se changer dans les vestiaires afin d'être stérile. Puis enfin il rentre dans la salle d'opération ou la sage-femme l'attend.
Mais il manque le bistouri électrique afin de procéder à la découpe.
De plus la péridural n'a pas été injecté correctement.
Patrick ne réussira sont opération que s'il réunit tous les instruments stériles afin de procéder à l'opération.

I.F) Détail des lieux, items, personnages :

Les lieux :

L'entrée : Salle de départ du jeu. Un secrétaire très bavard s'y trouve. Il fera tout pour prendre le plus de temps

Vestiaire : Salle dans laquelle Patrick doit se changer.
Toilette :
Salle d'opération : Endroit où le médecin interagit avec le patient et le personnel de santé
Salle des instruments : Tous les instruments nécessaires à l'opération s'y trouve.

Les personnages :

Patrick : médecin chirurgien
Secrétaire : très bavard
Sage-femme : Accompagne la patiente
Patiente : en panique

Les items :

IDBadge : permet d'accéder au vestiaire
DiathermyKnife : nécessaire à la découpe du ventre
Bistoury : nécessaire à la découpe du ventre

I.G) Situations gagnantes et perdantes :

Gagne : Patrick traverse les salles et réunit tous les instruments nécessaires à l'opération dans un temps imparti.

Perd : temps imparti écoulé

II. Réponses aux exercices

7.4 On recréer une nouvelle version du jeu à partir du TP 3.2. On a détruit les packages v1 et veref pour laisser place au 5 classes du jeu.

7.5 On cherche à éviter la duplication de code. Pour cela on va utiliser la méthode printLocationInfos() dans la classe Game qui va nous afficher les sorties disponibles en évitant la répétition de code, notamment des fonctions goRoom() et printWelcome().

7.6 La méthode getExit() dans Room retourne la salle se trouvant dans la direction entrée en paramètre. Cette méthode permet de découpler les classes en créant des attributs privés.
On va essayer de découpler les classes en introduisant le hashmap.

7.7 On a écrit la méthode getExitString dans la classe Room afin d'éviter la duplication de code dans la méthode printLocationInfos de la classe Game.

7.8 On introduit la HashMap dans la classe Room qui nous permet d'éviter de déclarer tous les attributs publics des différentes sorties. On va réutiliser ce HashMap dans la classe createRooms. Celui-ci est utilisé afin de positionner les sorties pour créer le réseau de salles.
Avant cela, il faut importer la HashMap avec : import java.util.HashMap;
Puis on l'initialise les sorties dans setExits.

7.8.1 Grâce à la HashMap introduite précédemment, nous pouvons inventer différentes directions tel que le déplacement à la vertical. Ainsi, j'ai déplacer ma salle d'opération sous la salle de vestiaire.

7.9 On a introduit une nouvelle encapsulation hashMap en écrivant : import java.util.Set; dans la classe Room. Cela nous a permis de remplacer les sorties par des clées (« Key ») et donc d'éviter la duplication de code.

7.10 La méthode getExitString permet d'afficher les sorties grâce à une boucle for. Cette boucle va afficher les sorties disponibles en fonction de la salle où se trouve le joueur.

-Set<String> key = this.aExits.keySet(); Crée une variable clé de type Set<String>. Il est donc dans une liste d'éléments de type String, les éléments stockés sont les clés des aExits de HashMap.

-for (String exit : keys) { ReturnString += " " + Exit; } pour chaque boucle qui itérera sur la liste clé précédemment stockée, la variable de sortie prend la valeur de la clé suivante du HashMap à chaque nouveau cycle. A chaque tour, la variable locale vReturnString est allongée.

-Return vReturnString.toString(); Renvoie une chaîne montrant toutes les sorties possibles.

7.11 On a créé la méthode getLongDescription() dans Room. Ainsi, la méthode printLocalInfo() de Game est simplifiée.

7.14 On a créé la commande look qui va permettre au joueur de savoir où il se trouve. Pour cela il a fallu ajouter le nouveau mot dans la classe CommandWords puis, il faut l'ajouter dans la méthode processCommand de la classe Game.

7.15 On fait la même chose que pour la création de la commande look, néanmoins, je l'ai appelé treat.

```java
/**
 * Constructor - initialise the command words.
 */
public CommandWords()
{
    this.aValidCommands = new String[5];
    this.aValidCommands[0] = "go";
    this.aValidCommands[1] = "help";
    this.aValidCommands[2] = "quit";
    this.aValidCommands[3] = "look";
    this.aValidCommands[4] = "treat";
} // CommandWords()
/**
 * Permet d'afficher les différentes commandes si le joueur ne sait pas comment fonctionne le jeu
 */
private boolean processCommand(final Command pCommand)
{
    String CommandWord =  pCommand.getCommandWord();
    switch(CommandWord){
        case "go":
            this.goRoom(pCommand);
            break;
        case "help":
            this.printHelp();
            break;
        case "look":
            this.look();
            break;
        case "treat":
            this.treat();
            break;
        default :
            System.out.println("I don't know what you mean...");
            return false;
        case "quit":
            return this.quit(pCommand);
    }
    return false;
}// processCommand
private void treat()
{
    System.out.println("Caesarian in process");
}
```

7.16 On crée la méthode showAll, qui affichera la liste des commandes répertoriées dans aValidCommands. En effet, cela permettra d'éviter des oublies dans l'ajout de la liste des commandes. De plus, pour éviter de créer un lien entre la classe Game et CommandWords, on va l'appeler depuis Game en utilisant la classe Parser.

```java
/**
 * Affiche toutes les commandes valides sur System.out.
 */

public void showAll()
{
    for (String command : validCommands){
        System.out.print (command + " ");
    }
    System.out.println ();
}
```

```java
/**
 * Affiche une liste des commandes valides.
 */
public void showCommands()
{
    aValidCommands.showAll();
}//showCommands
```

7.18 On modifie le nom de la méthode showAll() en getCommandList() pour éviter des problèmes pouvant venir plus tard à cause d'encapsulation. On doit aussi changer les noms des méthodes qui faisaient appelle à la méthode showAll().

```
/**
 * Affiche toutes les commandes valides sur System.out.
 */

public void getCommandList()
{
    for (String command : validCommands){
        System.out.print (command + " ");
    }
    System.out.println ();
}
```

7.18.1 J'ai comparé ma version ZuulBad avec ZuulBetter en intégrant tout ce qu'il me manquait.

7.18.3 J'ai trouvé 5 images pour chacune de mes pièces

7.18.4 Le titre de mon jeu est « Doctor Under Pressure ».

7.18.5 J'ai ajouté une interface graphique à mon jeu qui s'adapte en fonction de la pièce ou le joueu se trouve.

7.18.8 J'ai créé le bouton help en adaptant ma méthode createGUI afin de le faire apparaitre. Je l'ai dimensionné et placé à l'endroit où je veux qu'il soit présent.

7.19.2 J'ai regroupé mes images dans un répertoire « Images » et j'ai modifié l'adresse des images dans le classe createRooms.

7.20. Création d'une classe item décrivant les caractéristiques d'un objet. On va aussi ajouter l'attribut aItem qu'on déclare dans le constructeur. Puis on ajoute l'item dans la méthode createRooms de la classe GameEngine.

7.21 On modifie la méthode getLongDescription() de la classe Room afin qu'elle affiche les items dans la Room.

7.22 Afin de faire apparaitre plusieurs Items dans une Room, on définit une HashMap <String, Item> pour notre attribut aItem. On adapte getLongDescription() et on créer la méthode addItem dans la classe Room.

7.22.2 J'ai intégré 2 objets dans ma Room vInstrument.

7.23 Afin de créer la commande back pour retourner dans la pièce précédente où je me trouvais, j'ajoute à la liste de commande le mot « back », dans la classe CommandWords. Par la suite, j'ai ajouté un attribut aLastRoom de type Room, ce qui me permet de modifier goRoom().

```
/**
 * Try to go to one direction. If there is an exit, enter the new
 * room, otherwise print an error message.
 */
private void goRoom( final Command pCommand )
{
    if ( ! pCommand.hasSecondWord() ) {
        // if there is no second word, we don't know where to go...
        this.aGui.println( "Go where?" );
        return;
    }

    String vDirection = pCommand.getSecondWord();

    // Try to leave current room.
    Room vNextRoom = this.aCurrentRoom.getExit( vDirection );

    if ( vNextRoom == null )
        this.aGui.println( "There is no door!" );
    else {
        this.aLastRoom = this.aCurrentRoom;
        this.aCurrentRoom = vNextRoom;
        this.aGui.println( this.aCurrentRoom.getLongDescription() );
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }
}//goRoom
```

```
/**
* Procédure Back
*/
private void back()
{
    Room vCurrentRoom = this.aCurrentRoom;
    this.aCurrentRoom = this.aLastRoom;
    this.aLastRoom = vCurrentRoom;

    this.aGui.println(this.aCurrentRoom.getLongDescription());
    if(this.aCurrentRoom.getImageName() != null)
     this.aGui.showImage(this.aCurrentRoom.getImageName());
}//back
```

J'ai créé la méthode back :
Aussi, dans interpretCommand(), j'ajoute notre nouveau mot.

7.26 Avant tout, j'ai importé une collection du JDK dans GameEngine : import java.util.Stack ; .J'ai créé l'attribut Stack<Room> aPreviousRoom dans la classe GameEngine puis je l'ai initialisé dans le constructeur. J'ai ajouté dans la méthode goRoom() : this.aLastRoom.push(aCurrentRoom); .
Puis j'ai modifié la méthode back

```
/**
* Procédure Back, permettant de retourner dans la Room précédente.
*/
private void back()
{
   if(this.aLastRoom.empty()) {
       this.aGui.println( "You are in the first room.");
    }
   else{

       this.aPlayer.setCurrentRoom(this.aLastRoom.pop());
       this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription( ) );
      if (this.aPlayer.getCurrentRoom().getImageName( ) != null )
       {
        this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName());
      }

    }
}//back
```

7.28.1 J'ai créé une nouvelle commande test en l'ajoutant dans la classe CommandWords pour que le mot test soit valide. J'ai aussi importé des élément du JDK tel que import.io.File ; import java.io.IOException ; et import java.util.Scanner ;

J'ai par la suite écrit la procédure test dans GameEngine :

```
/**
 * Procédure de la commande test qui actionne les commandes d'un .txt dans le dossier racine du jeu.
 * Il suffit d'écrire : test + (nom du .txt).
 */
private void test(final Command pCommand)
{
    if(!pCommand.hasSecondWord()) {
        aGui.println("Which files do you want to test ?");
    }
    try {
        File vFileTest = new File(pCommand.getSecondWord()+".txt");
        Scanner vScan = new Scanner (vFileTest);
        while (vScan.hasNextLine()) //a la ligne suivante
        {
            interpretCommand(vScan.nextLine());
        }
        vScan.close();
    }
    catch (final java.io.FileNotFoundException pE){
       aGui.println("File not found.");
    }
}//test
```

J'ai ajouté dans la procédure interpretCommand cela :

```
else if (vCommandWord.equals("test")) {
    this.test(vCommand);
}
```

7.28.2 J'ai créé le fichier court.txt avec écrit help dessus. Puis le fichier allroom.txt qui permet de visiter toutes les pièces en partant de la room initiale. Et enfin, le fichier win.txt qui donne la direction pour pouvoir gagner si on l'utilise.

7.29 J'ai créé la classe Player :

```
import java.util.Stack;
/**
 * Charactéristiques du joueur
 *
 * @author Hugo DAUVERGNE
 * @version 20/04/22
 */
public class Player
{
    private String aName;
    private double aWeight;
    private Room aCurrentRoom;
    private Stack<Room> aLastRoom;

    public Player(final String pName, final double pMaxWeight)
    {
        this.aName = pName;
        this.aWeight = pMaxWeight;
        this.aCurrentRoom = null;
        this.aLastRoom = new Stack();
    }

    public String getName(){return this.aName;}
    public double getMaxWeight() {return aWeight;}
    public Room getCurrentRoom(){return this.aCurrentRoom;}
    public Room getLastRoom(){return this.aLastRoom.pop();}

    public void setName(final String pName){this.aName=pName;}
    public void setMaxWeight(final double pWeight) {aWeight= pWeight; }
    public void setCurrentRoom(final Room pCurrentRoom){this.aCurrentRoom = pCurrentRoom;}
    public void setLastRoom(final Room pLastRoom){this.aLastRoom.push(pLastRoom);}

}//Player
```

Par la suite j'ai adapté la classe GameEngine pour qu'elle prenne en compte la création du Player. J'ai créé l'attribut : private Player aPlayer; puis l'ai ajouté à mon constructeur. J'ai remplacé toutes les méthodes faisant appellent à aCurrentRoom par aPlayer.getCurrentRoom. Désormais, GameEngine fournit les paramètres et affiche les résultats des actions de Player.

7.30 Dans la classe CommandWords on a ajouté « take » et « drop » dans le tableau de Strings aValidCommands. Puis, dans la classe GameEngine on a modifié la méthode interpretCommand() pour prendre take et drop en compte. Par la suite, j'ai créé les méthodes addIte et removeItem dans la classe Room, pour pouvoir ajouter un item lorsqu'on le dépose dans une room quelconque ou au contraire le retirer si on le personnage récupère l'item.

```
/**
 * Procédure permettant d'ajouter un item dans une room
 */
public void addItem(final String pNomItem, final Item pItem )
  {
  this.aItem.put(pNomItem, pItem);
  }


/**
 * Procédure removeItem permettant de retirer un item d'une room
 */
public void removeItem(final String pNomItem)
  {
  this.aItem.remove(pNomItem);
  }
```

Il nous faut un attribut qui stocke les items récupérer par le personnage. Pour cela on crée l'attribut aInventary qu'on initialise dans le constructeur. On ajoute toujours dans la classe Player, takeItem, dropItem et getItem :

```
 /**
 * Permet de prendre des objets disponible dans une piece
 * @param pStringItem, pItem
 */
public void takeItem(final String pStringItem, final Item pItem)
{
    this.aList.takeItem(pStringItem, pItem);
}
/**
 * Permet de lacher des objets qui sont dans l'inventaire
 * @param pStringItem
 */
public void dropItem(final String pStringItem)
{
    this.aList.dropItem(pStringItem);
}

// ## Accesseurs ##
/**
 * Renvoi l'item correspondant à la string
 * @param pItem String qui correspond à la description de l'objet
 * @return Item qui correspond à la string
 */
public Item getItem(final String pItem){
    return this.aList.getItem(pItem);
}
```

Enfin, on a ajouté les méthodes take() et drop() dans la classe GameEngine :

```java
/**
 * Procédure take
 */
private void take(final Command pCommand )
{
    if(!pCommand.hasSecondWord()){
        this.aGui.println( "Take what ?");
        return;
    }
    Item vItem = this.aPlayer.getCurrentRoom().getItem(pCommand.getSecondWord()); // création de la variable vItem
    if (vItem == null){
        this.aGui.println("This item is not here. ");
    }
    else{
        this.aPlayer.takeItem(pCommand.getSecondWord(), vItem);
        this.aPlayer.getCurrentRoom().removeItem(pCommand.getSecondWord( )); //retire l'item de la Room
        this.aGui.println("You take the item "+pCommand.getSecondWord( ));
    }
}//take
/**
 * Procédure drop
 */
private void drop(final Command pCommand)
{
    if(!pCommand.hasSecondWord())
    {
        this.aGui.println("Drop what ?");
        return;
    }
    String vItem = pCommand.getSecondWord();
    Item vToDrop = this.aPlayer.getItem(vItem);
    if(vToDrop == null) {
        this.aGui.println("I don't have it !");
    }
    else{
        this.aPlayer.getCurrentRoom().addItem(vItem, vToDrop);
        this.aPlayer.dropItem(vItem);
        this.aGui.println("I have drop it !");
    }//drop
}//drop
```

7.31 J'ai créé la classe ItemList qui reprend des méthodes de la classe Player afin qu'elle puisse permettre d'être utilisé comme un inventaire :

```java
import java.util.HashMap;
import java.util.Set;
/**
 * Classe qui s'occupe de l'inventaire des Rooms et du Player.
 * @author Hugo DAUVERGNE
 * @version 22/04/22
 */
public class ItemList
{
    private HashMap<String, Item> aInventary; //HashMap ≈ « tableau associatif » qui contient un ensemble d'associations clé→valeur,
                                              //ici dans le cas présent, associe toute les chaines de caractère et toutes ces caractéristiques

    // ## Constructeur ##
    /**
     * Constructeur d'objets de classe ItemList
     */
    public ItemList()
    {
        this.aInventary = new HashMap<>();
    }

    // ## Accesseurs ##
    /**
     * Renvoi l'item correspondant à la string
     * @param pItem String qui correspond à la description de l'objet
     * @return Item qui correspond à la string
     */
    public Item getItem(final String pItem){
        return this.aInventary.get(pItem);
    }

    /**
     * Donne le poids total du personnage
```

```java
/**
 * Fonction getItemString de type String
 * @return Renvoi toutes les clées auxquels les items sont liées
 */
public String getItemString(){
    String vReturnString = "";
    Set<String> vKeys = this.aInventary.keySet();
    for(String vItem : vKeys)
    {
        vReturnString += " a "+vItem+"\n";
    }
    return vReturnString;
}
/**
 * Procédure takeItem qui a partir de la HashMap aInventary,
 * associe une chaine de caractère et une caractéristique à takeItem
 */
public void takeItem(final String pStringItem, final Item pItem){
    this.aInventary.put(pStringItem, pItem);
}
/**
 * Procédure dropItem qui a partir de la HashMap aInventary,
 * associe une chaine de caractère à dropItem
 */
public void dropItem(final String pStringItem){
    this.aInventary.remove(pStringItem);
}
```

7.32 J'ai créé la méthode getTotalWeight() dans la classe ItemList :

```java
/**
 * Donne le poids total du personnage
 * @return Weight qui correspond au poids
 */
public double getTotalWeight()
{
    double vWeight = 0;
    Set <String> vKeys = this.aInventary.keySet();
    for(String vItem: vKeys)
    {
        vWeight+=getItem(vItem).getWeight();
    }
    return vWeight;
}
```

Par la suite j'ai ajouté deux attributs dans la classe Player qui me permettent d'indiquer le poids maximum que peut porter le personnage avec aMaxWeight, initialiser à 2. Le second attribut, aWeight représente le poids courant du personnage, et est initialisé à 0.

```java
/**
 * Modifie le poids courant du personnage aWeight
 * pour qu'il prenne valeur du poids total
 */
public double totalWeight()
{
    return this.aWeight=this.aList.getTotalWeight();
}
/**
 * Peut ou pas porter l'item sur lui
 */
public boolean canCarry(final Item vItem)
{
    return totalWeight()+vItem.getWeight()<= this.aMaxWeight;
}
```

Ajout dans la classe GameEngine, méthode take() :

```java
else if(this.aPlayer.canCarry(vItem)==false)
{
    this.aGui.println("WARNING!! It's too heavy");
}
```

7.33 Ajout de la commande « item », qui permet d'afficher l'inventaire, dans InterpretCommand(). Puis écriture de la méthode item dans la classe GameEngine :

```java
/**
 * Affiche l'inventaire
 */
private void item(final Command pCommand)
{
    if(pCommand.hasSecondWord())
    {
        this.aGui.println("Item cannot take parameter.");
        return;
    }
    String Inventory = this.aPlayer.getInventoryString();
    this.aGui.println(Inventory);
}
```

III.Mode d'emploi

Téléchargez le .jar du jeu. Puis ouvrez BlueJ. Lancez le .jar dans le lecteur.
A partir de la classe Game, faites un clic droit et cliquer sur play.
Le jeu se lance, et il ne vous reste plus qu'à profiter.
Bon jeu ☺

IV. Déclaration obligatoire anti-plagiat

Je certifie ne pas avoir recopié de ligne de code sans en comprendre le réel sens. J'ai été aidé dans ce projet par les membres du club Nix : Velaris, Reis TITOUAMANE, Veikoon, Chelinka, Blackjack. L'intervenante Céline Po m'a aussi aidé. Je me suis inspiré des rapports de projet de Blackjack, de Lucas PLANCHAIS et de Eliott Blondin.