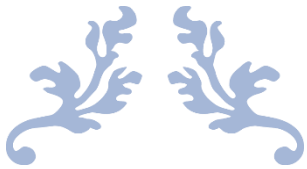


SURVIVING HISTORY

COLNOT Clémence

26 OCTOBRE 2021

ESIEE

Table des matières

I. PROJET

A. Auteur

COLNOT Clémence

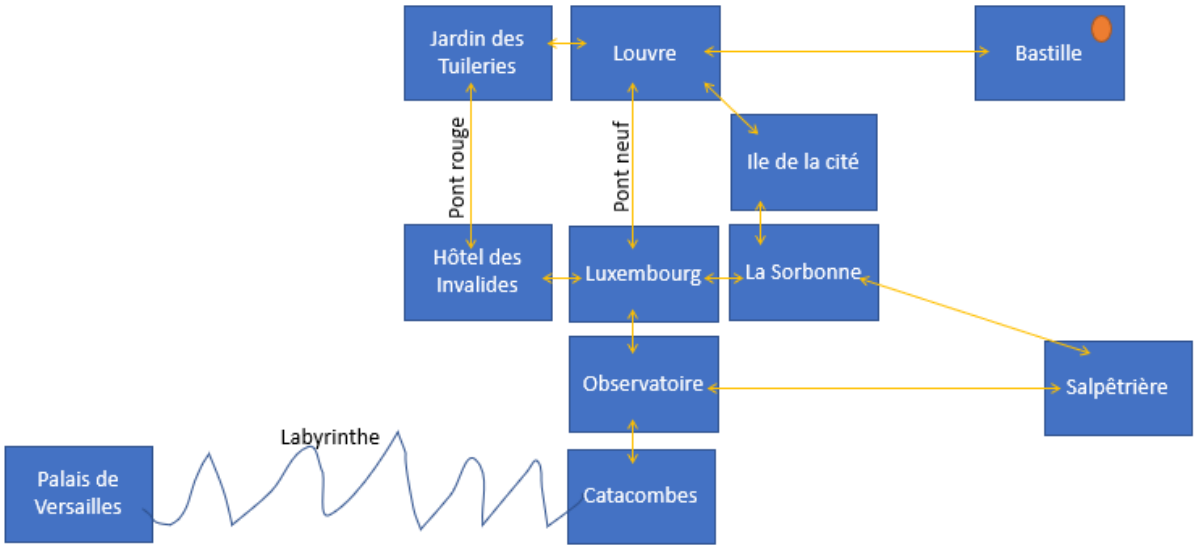
B. Thème

Après un saut dans le temps, une jeune fille doit recréer ses conditions de voyage dans le temps pour retourner dans son époque.

C. Résumé du scénario

Vous être une jeune fille du nom de Lexie, vivant à Paris au XXIème siècle. C'était une journée d'école tout à fait classique, cours de maths, et d'histoire mais ce que vous attendiez le plus était votre cours de robotique. Votre sac rempli de vos outils préférés, vous allez enfin pouvoir travailler sur votre projet, une machine pour voir le passé. À la suite d'une manipulation peu fructueuse entrainant une explosion de la machine, vous vous retrouvez dans un lieu totalement inconnu. Les personnes sont habillées d'une drôle de manière et vous êtes emprisonnée ! Vous êtes maintenant dans une prison de la Bastille au XVIIème siècle. Une seule pensée vous habite retourner chez vous. Alors vous vient l'idée de reproduire l'explosion qui vous a fait arriver ici. Parviendrez-vous à créer cette explosion sans éveiller les soupçons et perturber l'histoire ?

D. Plan



E. Scénario détaillé

F. Détails des lieux, items, personnages

Les objets sont :

Dans la cellule de la bastille : une clé pour sortir de la cellule.
Dans la Bastille : un feu.
Dans la Louvre : une note avec le chemin vers le château de Versailles par les catacombes.
Dans Notre-Dame : une bougie.
Dans la chambre du roi : la recette de la poudre à canon.
Dans l'observatoire : des livres de jardinage, géologie, et un dictionnaire.
Dans le jardin des Tuileries : du salpêtre.
Dans les catacombes : du sulfure.
Dans l'Hôtel des Invalides : du charbon.
Dans la Salpêtrière : le cookie magique.

G. Situations gagnantes et perdantes

Situation gagnante : Le joueur trouve les objets qui lui permettront de recréer l'explosion dans le cachot de la Bastille.

H. Enigmes, mini-jeux, combats

I. Commentaires

II. REPONSES AUX EXERCICES

7.4_ Les noms des salles ont été adaptées sur le plan dessiné plus haut.

7.5_ La méthode *printLocationInfo* permet d'afficher l'ensemble des sorties, ainsi que la description de la salle courante.

7.6_ La méthode *getExit* permet de retourner les sorties possibles dans la salle courante, sans pour autant l'afficher.

7.7_ La méthode *getExitString*, permettra d'afficher les sorties existantes au joueur en fonction des retours donnés par la fonction *getExit.* Cette fonction pourra donc être appelée par la méthode *printLocationInfo*, afin d'éviter la duplication de code.

7.8_ L'utilisation de HashMap permet de stocker les sorties d'une manière plus efficace. Cette modification a entrainé le changement de la méthode *setExits*.

```
public class Room
{
    private String aDescription;
    private HashMap <String, Room> aExits;
```

```
public void setExits(final String pName, final Room pNextRoom){
    switch (pName){
        case "North":
            this.aExits.put("North", pNextRoom);
            break;
        case "South":
            this.aExits.put("South", pNextRoom);
            break;
        case "West":
            this.aExits.put("East", pNextRoom);
            break;
        case "East":
            this.aExits.put("West", pNextRoom);
            break;
        case "Up":
            this.aExits.put("Up", pNextRoom);
            break;
        case "Down":
            this.aExits.put("Down", pNextRoom);
            break;
        default :
    }

}// setExits
```

7.8.1_ Le déplacement vertical ajouté se passe dans le Palais de Versailles. Ceci a permis d'ajouter un accès aux catacombes, et à la galerie des glaces.

7.9 _ La méthode *keySet* retourne l'ensemble des valeurs que peut prendre les clés de la HashMap. Ici les valeurs « north », « south », « east » et « west ».

7.10_ La méthode *getExitString* fait une concaténation des chaînes de caractères stockées dans chaque case de la HashMap existante, après la chaîne de caractère « Sorties : »

```
public String getExitsString(){
    String vExits = "Sorties :";
    Set<String> keys = this.aExits.keySet();
    for(String vExit : keys){
        vExits += " " + vExit;
    }
    return vExits;
}// getExitsString
```

7.11_ La méthode *getLongDescription* permet d'afficher à la fois la description de la salle courante et l'ensemble des sorties qui la constitue.

```
public String getLongDescription(){
    return "Vous êtes "+ this.getDescription() + " "+ this.getExitsString()+".\n";
}// getLongDescription
```

7.14_ La méthode *look* permet d'utiliser la méthode *getLongDescription* au travers d'une commande émanant directement du joueur.

7.15_ Ajout des fonctionnalités *eat*.

```
public void eat(){
    System.out.println("You've eaten. You're full of energy now.");
}// eat
```

7.16_ La méthode *showAll* de la classe CommandWords permet d'afficher au joueur l'ensemble des commandes existantes. Alors que la méthode *showCommands*, elle permet de faire un lien avec la classe *Parser* pour l'affichage.

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

```
public void showCommands(){
    this.aValidCommands.showAll();
}// showCommands
```

```
private void printHelp(){
    System.out.println ("You are lost. You are alone. You're locked in the Bastille, in the XVII century.");
    System.out.println("Your command words are :");
    this.aParser.showCommands();
}// printHelp
```

7.18_ La méthode *getCommandList* remplacera la méthode *showAll*. En stockant dans un String l'ensemble des valeurs des commandes. Pour ensuit les afficher dans la méthode *showCommands* de la classe Parser.

```
public String getCommandList(){
    String vList = "";
    for(String command : sValidCommands){
        vList += command + " ";
    }
    return vList;
}
```

```
public void showCommands(){
    String vList = this.aValidCommands.getCommandList();
    System.out.println(vList);
}// showCommands
```

7.18.1_ Après comparaison, peu de choses ont changés dans la version actuelle du code. Quelques petites modifications, notamment des retraits de variables non utilisées.

7.18.2_ Les *StringBuilder* permettent de faciliter la formation de longues chaînes de caractères. L'optimisation apporté par cette classe est observable dans les méthodes suivantes, respectivement de la classe *Room* et *CommandWords* :

```java
/**
 * Get Exits String
 * @return available exits
 */
public String getExitsString(){
    StringBuilder vExits = new StringBuilder( "Exits:" );
    Set<String> keys = this.aExits.keySet();
    for(String vExit : keys){
        vExits.append(" " + vExit);
    }
    return vExits.toString();
}// getExitsString
```

```java
/**
 * Print all the valid commands
 */
public String getCommandList(){
    StringBuilder vList = new StringBuilder();
    for(int i = 0; i < sValidCommands.length; i++) {
        vList.append( sValidCommands[i] + "   " );
    }
    return vList.toString();
}
```

7.18.3_ Les liens des premières images trouvées pour les salles :

Palais de Versailles : https://www.chateaux-forts-de-france.fr/conseils-pratiques-chateau-de-versailles/

Jardin de Versailles : https://www.nationalgeographic.fr/thematique/sujet/histoire/architecture/chateaux

Catacombes : https://parissecret.com/les-catacombes-de-paris-une-balade-underground-et-mysterieuse/

Sorbonne : https://www.futura-sciences.com/sciences/questions-reponses/sciences-education-sous-ancien-regime-10580/

Observatoire : https://voyageforum.com/guides/observatoire_de_paris/

Louvre : https://stringfixer.com/fr/Louvre_Palace

Tuileries : https://www.pca-stream.com/fr/articles/allen-weiss-un-espace-hybride-inaugurant-la-modernite-126

Bastille : https://www.geo.fr/histoire/14-juillet-1789-la-vraie-histoire-de-la-prise-de-la-bastille-195527

Cachot bastille : http://www.verlatradition.fr/archives/2015/07/22/32387477.html

Luxembourg : https://artsandculture.google.com/exhibit/le-jardin-du-luxembourg-du-jardin-des-chartreux-au-jardin-du-s%C3%A9nat/wRSJ-ZJb

Ile de la cité : http://paris-atlas-historique.fr/30.html

Chambre de la Reine : https://www.chateauversailles.fr/decouvrir/domaine/chateau/grand-appartement-reine

Chambre du Roi : http://www.versailles3d.com/fr/au-cours-des-siecles/xxe/1980.html

Galerie des glaces : https://www.francetvinfo.fr/culture/patrimoine/versailles-les-secrets-de-l-eblouissante-galerie-des-glaces_3357961.html

Souterrain Versailles : https://subterranologie.com/les-caves-du-roi

Hôtel des Invalides : https://www.defense.gouv.fr/web-documentaire/site-info-familles-terre/page22.html

Couloir Versailles : https://fr.dreamstime.com/photos-images/couloir-de-versailles.html

Notre Dame : https://www.filiere-3e.fr/2019/04/18/notre-dame-de-paris-en-lumiere/

Salpêtrière : https://www.wikiwand.com/fr/H%C3%B4pital_de_la_Salp%C3%AAtri%C3%A8re

[En cours]

7.18.4_ Mon jeu se nomme « Surviving History ».

7.18.5_ Les objets *Room* sont maintenant stockés dans une *HashMap*, contenant en clé le nom de la salle et en valeur la *Room* associée.

```
private HashMap <String, Room> aRooms;
```

Ainsi après avoir initialisé les différentes salles dans la procédure *createRooms*, on stocke les différentes salles sans la HashMap comme suivant :

```
//Stockage dans la HashMap
this.aRooms.put("vBastille", vBastille);
this.aRooms.put("vCelluleBastille", vCelluleBastille);
this.aRooms.put("vLouvre", vLouvre);
this.aRooms.put("vJardinTuileries",vJardinTuileries);
this.aRooms.put("vIleCite", vIleCite);
this.aRooms.put("vSorbonne", vSorbonne);
this.aRooms.put("vLuxembourg", vLuxembourg);
this.aRooms.put("vHotelInvalides", vHotelInvalides);
this.aRooms.put("vSalpetriere", vSalpetriere);
this.aRooms.put("vObservatoire", vObservatoire);
this.aRooms.put("vCatacombes", vCatacombes);
this.aRooms.put("vPalaisVersailles", vPalaisVersailles);
this.aRooms.put("vGallerieGlasses", vGallerieGlasses);
this.aRooms.put("vCouloirChambre", vCouloirChambre);
this.aRooms.put("vChambreReine", vChambreReine);
this.aRooms.put("vChambreRoi", vChambreRoi);
this.aRooms.put("vSouterrainVersailles", vSouterrainVersailles);
this.aRooms.put("vJardinVersailles", vJardinVersailles);
```

7.18.6_ Afin de rajouter l'interface graphique, il nous faut modifier la classe *Room* pour ajouter l'image associée à la pièce et son getter

```
/**
 * Constructeur naturel
 * @param Description du lieu et Image associée
 */
public Room(final String pDescription, final String pImage){
    this.aDescription = pDescription;
    this.aExits = new HashMap <String, Room>();
    this.aImageName = pImage;

}// Room
```

```
/**
 * @return le nom de l'image
 */
public String getNameImage(){
    return aImageName;
}// getNameImage
```

Cet ajout de l'image entraîne donc le changement de la méthode *createRoom* dans laquelle il va falloir ajouter le nom de l'image.

Pour ce qui est de la classe *Parser*, on intègre une nouvelle version qui n'utilisera plus les *Scanner* mais les *StringTokenizer*. Ce qui permet de simplifier le code d'un attribut non utile.

La classe *Game* est la classe qui subit le plus de modifications. Elle va être divisée en trois différentes classes : *Game, UserInterface* et *GameEngine.*

La classe *Game* ne contiendra que l'instanciation du jeu, et la méthode *quit.*

La classe *GameEngine* reprend la majorité de la classe *Game* de la version précédente en permettant de faire le lien avec la future interface graphique, par l'ajout de l'attribut aGui. Ainsi tous les affichages par *System.out* sont remplacés par des affichages *this.aGui.print*.

```
public class GameEngine
{
    private Parser        aParser;
    private Room          aCurrentRoom;
    private UserInterface aGui;
```

```
/**
 * Set the value of the User Interface
 */
public void setGUI(final UserInterface pUserInterface){
    this.aGui = pUserInterface;
    this.printWelcome();
}// setGUI
```

```
/**
 * Print Location infos
 */
private void printLocationInfos(){
    this.aGui.println(this.aCurrentRoom.getLongDescription());
    this.aGui.println(this.aCurrentRoom.getNameImage());
}
```

Pour ce qui est de la classe *UserInterface*, cette classe permettra toute l'interaction avec l'utilisateur notamment l'interface graphique.

```
public class UserInterface implements ActionListener
{
    private GameEngine aEngine;
    private JFrame     aMyFrame;
    private JTextField aEntryField;
    private JTextArea  aLog;
    private JLabel     aImage;

    /**
     * Constructeur d'objets de classe UserInterface
     * @param  GameEngine pGameEngine
     */
    public UserInterface(final GameEngine pGameEngine)
    {
        this.aEngine = pGameEngine;
        this.createGUI();
    }
```

Le constructeur de la classe *UserInterface* appelle la méthode *createGUI* qui initialise l'interphase graphique du jeu.

```
private void createGUI()
{
    this.aMyFrame = new JFrame( "Zork" ); // change the title
    this.aEntryField = new JTextField( 34 );

    this.aLog = new JTextArea();
    this.aLog.setEditable( false );
    JScrollPane vListScroller = new JScrollPane( this.aLog );
    vListScroller.setPreferredSize( new Dimension(200, 200) );
    vListScroller.setMinimumSize( new Dimension(100,100) );

    JPanel vPanel = new JPanel();
    this.aImage = new JLabel();

    vPanel.setLayout( new BorderLayout() ); // ==> only five places
    vPanel.add( this.aImage, BorderLayout.NORTH );
    vPanel.add( vListScroller, BorderLayout.CENTER );
    vPanel.add( this.aEntryField, BorderLayout.SOUTH );

    this.aMyFrame.getContentPane().add( vPanel, BorderLayout.CENTER );

    // add some event listeners to some components
    this.aEntryField.addActionListener( this );

    // to end program when window is closed
    this.aMyFrame.addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent e) { System.exit(0); }
    } );

    this.aMyFrame.pack();
    this.aMyFrame.setVisible( true );
    this.aEntryField.requestFocus();
} // createGUI()
```

Cette classe possède notamment le méthodes *showImage, print, println* et *enable,* qui permettent d'afficher ou non des éléments d'images ou de texte dans l'interface.

7.18.8_ L'insertion du bouton se fait dans la classe *UserInterface* par l'incorporation de la librairie *javax.swing.JButton,* et l'ajout d'un attribut à la classe de type JButton. Ce bouton est initialisé dans la méthode *createGUI*

```
this.aButton = new JButton("OK");
```

Il est positionné à droite de la fenêtre graphique par la commande :

```
vPanel.add(this.aButton, BorderLayout.EAST);
```

Et effectue une action suite au cliquage, grâce à la méthode *actionPerformed* appelée dans la méthode *addActionListener*.

7.19.2_ Après ajout des images dans le répertoire Images, il faut ajouter le chemin vers les images.

```
Room vBastille = new Room("in the court yard of the Bastille.","Images/bastille.jpg");
Room vCelluleBastille = new Room("locked a cell of the Bastille.","Images/cachot.jpg");
```

7.20_ Afin d'avoir des objets dans les pièces, on ajoute au projet une classe *Item* ainsi qu'un attribut *Item* dans la classe *Room*.

```
public class Item
{
    private String aDescription;
    private int aPoids;

    /**
     * Constructeur d'objet de classe Item
     */
    public Item(final String pDescription, final int pPoids)
    {
        this.aDescription = pDescription;
        this.aPoids = pPoids;
    }// Item

    /**
     * Getter item description
     * @return String aDescription
     */
    public String getDescription(){
        return this.aDescription;
    }// getDescription

    /**
     * Getter item weight
     * @return int aPoids
     */
    public int getWeight(){
        return this.aPoids;
    }// getWeight
```

```
public class Room
{
    private String aDescription;
    private HashMap <String, Room> aExits;
    private String aImageName;
    private Item aItem;
```

L'instanciation des *Item* se fait alors dans la méthode *createRooms* de la classe *GameEngine*.

```
//Placement Item dans les lieux
vCelluleBastille.setItem(new Item("Key", 3));
vLouvre.setItem(new Item("Note", 10));
vNotreDame.setItem(new Item("Candle stick", 1));
vObservatoire.setItem(new Item("Key", 1));
vChambreRoi.setItem(new Item("Powder Instructions", 10));
vJardinTuileries.setItem(new Item("Salpetre", 1));
vCatacombes.setItem(new Item("Soufre", 1));
vHotelInvalides.setItem(new Item("Coal", 1));
```

7.21_ Les informations de l'objet seront accessibles par la fonction *getStringDescription* retournant un *String* contenant les données complètes de l'objet, soit sa description et son poids. Cette méthode sera appelée dans la procédure *printLocationInfos* de la classe *GameEngine* pour un affichage au joueur.

```
/**
 * Get String item description and weight
 * @return String Description and weight
 */
public String getStringDescription(){
    return "Item Description : "+ this.aDescription +"\n"+"Weight Item: "+ this.aPoids;
}// getStringDescription
```

```
/**
 * Print Location infos
 */
private void printLocationInfos(){
    this.aGui.println(this.aCurrentRoom.getLongDescription());
    aGui.println(this.aCurrentRoom.getItem().getStringDescription());
    this.aGui.println(this.aCurrentRoom.getImageName());
}
```

7.21.1_ La méthode *look* prend maintenant en compte le recherche d'information sur un objet précis dans la pièce.

7.22_ Pour mettre plusieurs objets dans une même pièce on transforme l'attribut de type *Item* de la classe *Room* en une *HashMap* contenant des *Item.* Cette modification entraine des changements des méthodes *getItem* et *setItem* de la même classe.

```
public class Room
{
    private String aDescription;
    private HashMap <String, Room> aExits;
    private String aImageName;
    private HashMap <String, Item> aItems;

    /**
     * Constructeur naturel
     * @param Description du lieu et Image associée
     */
    public Room(final String pDescription, final String pImage){
        this.aDescription = pDescription;
        this.aExits = new HashMap <String, Room>();
        this.aItems = new HashMap <String, Item>();
        this.aImageName = pImage;

    }// Room
```

```java
/**
 * Get Item
 * @return Item
 */
public Item getItem(final String pName){
    return this.aItems.get(pName);
}// getItem
```

```java
/**
 * Set Item
 * @param Item
 */
public void setItem(final String pName, final Item pItem){
    this.aItems.put(pName, pItem);
}// setItem
```

De plus il faut maintenant afficher l'ensemble des objets présents dans la pièce. Ce qui amène à la création de la méthode *getItemsNames.* On fait de même pour les descriptions des objets avec la méthode *getItemsDescription.*

```java
public String getItemsNames(){
    StringBuilder vItemsNames = new StringBuilder("Items:\n");
    for(Item vI: this.aItems.values()){
        vItemsNames.append(vI.getName()).append("\n");
    }
    return vItemsNames.toString();
}
```

7.22.1_ Les objets sont ajoutés par la méthode *setItem* qui est appelée dans la méthode de la classe *GameEngine*, *createRooms*.

7.23_ La méthode *back* doit pouvoir ramener le joueur à la salle précédente. Il faut donc dans son trajet stocker la salle d'où il arrive. Pour ce faire, on ajoute un attribut *aPrevRoom* à la casse *GameEngine*. On ajoute donc l'accesseur et le modificateur associé.

```java
public Room getPreviousRoom(){
    return this.aPrevRoom;
}// getPreviousRoom
```

```java
public void setPreviousRoom(final Room pPrevRoom){
    this.aPrevRoom = pPrevRoom;
}// setCurrentRoom
```

La méthode *goRoom* est donc modifiée pour conserver la pièce précédente.

```
private void goRoom(final Command pCommand){
    if (!pCommand.hasSecondWord()) {
        this.aGui.println("Go where?");
        return;
    }

    String vDirection = pCommand.getSecondWord();
    Room vNextRoom = this.aCurrentRoom.getExit(vDirection);
    if (this.aCurrentRoom.getExit(vDirection) == null){
        this.aGui.println("There is no door !");

    }else {
        vNextRoom = this.aCurrentRoom.getExit(vDirection);
        this.setPreviousRoom(this.aCurrentRoom);
        this.setCurrentRoom(vNextRoom);
        this.printLocationInfos();
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }

}// goRoom
```

Ainsi la méthode *back* est créée :

```
public void back(){
    if (this.aPrevRoom.getShortDescription().equals("")){
        this.aGui.println("There are no previous room.");
    }
    else {
        this.setCurrentRoom(this.aPrevRoom);
        this.printLocationInfos();
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }
}
```

7.26_ La méthode *back* de la classe *GameEngine* permet de revenir à la salle précédente, mais pour pouvoir revenir en arrière de plusieurs salles, il faut donc en amont stocké l'ordre par lequel on passe par les salles. Le moyen de stockage le plus approprié est celui de la *Stack*. On va donc empiler l'une après l'autre les salles dans un nouvel attribut *aPrevRoom* de la classe *GameEngine.*

On ajoute alors les méthodes :

```
public void setPreviousRoom(final Room pPrevRoom){
    this.aPrevRoom.push(pPrevRoom);
}// setCurrentRoom
```

```
public Room getPreviousRoom(){
    return this.aPrevRoom.pop();
}// getPreviousRoom
```

Il faut donc ajouter le stockage des pièces dans la méthode *goRoom*. Et modifier la fonction *back* qui prend maintenant en paramètre une commande pour vérifier qu'il n'y a pas de second mot.

```
public void back(final Command pCommand){
    if (pCommand.hasSecondWord()){
        aGui.println("Didn't you mean back?");
        return;
    }
    if (this.aPrevRoom.empty()){
        this.aGui.println("There are no previous room.");
    }
    else {
        this.setCurrentRoom(this.aPrevRoom.pop());
        this.printLocationInfos();
        if ( this.aCurrentRoom.getImageName() != null )
            this.aGui.showImage( this.aCurrentRoom.getImageName() );
    }
}
```

7.26.1_ Générer deux javadocs

```
javadoc -d userdoc -author -version *.java
javadoc -d progdoc -author -version -private -linksource *.java
```

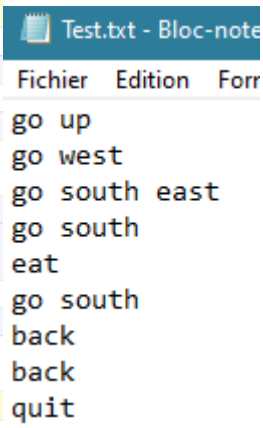
Les javadocs sont bien crées. // PBM UBUNTU GENERATION JAVADOC (installation)

7.28.1_ On crée la commande *test* afin de pouvoir lancer une série d'actions automatisées, écrites dans un fichier « Test.txt ». Mais pour lancer cette série de commande il faut en amont créer une méthode qui permet de lire le fichier Test.txt. Pour lire dans le fichier il faut donc importer les classes *Scanner*, *File.*

C'est donc naturellement que la méthode *readTestFile()* de la classe *GameEngine* est codé comme ce qui suit :

```
public void readTestFile(final String pFileName){
    File vFile = new File(pFileName);
    try {
        Scanner vScan = new Scanner(vFile);
        while(vScan.hasNextLine()){
            String vCommand = vScan.nextLine();
            interpretCommand(vCommand);
        }
    }
    catch(FileNotFoundException vException){
        this.aGui.println("File not found.");
    }

}// readTestFile
```



7.28.2_ A présent que nous avons une fonction permettant de lire une suite de commandes, nous pouvons utiliser cette manière afin de tester la résolution du jeu avec le fichier test Resolution.txt et l'ensemble des possibilités du jeu avec All.txt. Pour choisir le fichier lu par le jeu il faut donc lancer la commande *test* suivie du nom du fichier désiré.

```
case "test":
    if ( vCommand.hasSecondWord() )
        this.readTestFile("Commande/"+ vCommand.getSecondWord());

    else
        this.aGui.println("Which file?");
    return;
```

7.29_ La classe *Player* va permettre de stocker tous les attributs relatifs au joueur, tels que la salle courante et la liste des salles précédentes. On crée donc une classe Player dans laquelle on déplace les attributs *aCurrentRoom* et *aPrevRoom* initialement dans la classe *GameEngine.*

```
public class Player
{
    private Room            aCurrentRoom;
    private Stack<Room>     aPrevRoom;

public Player(final Room pCurrent)
{
    this.aPrevRoom = new Stack<Room>();
    this.aCurrentRoom = pCurrent;
}
```

Ces modifications entrainent des ajustements dans la classe *GameEngine.* Dans la méthode *createRooms*

```
// Position courante
this.aPlayer = new Player(vCelluleBastille);
```

On déplace avec eux les getters et setters associés. De plus, on ajoute une méthode permettant de récupérer la liste des pièces par lesquelles le joueur est passé :

```
/**
 * Getter All Previous positions
 * @return Stack aPrevRoom
 */
public Stack<Room> getPreviousRooms(){
    return this.aPrevRoom;
}// getPreviousRoom
```

7.30 & 7.31_ On veut permettre à l'utilisateur de prendre et relâcher des items. On implémente donc les méthodes *take* et *drop* dans la classe *GameEngine*.

```
public void take(final String pCommand){
    if (this.aPlayer.getCurrentRoom().getItems().containsKey(pCommand) ){
        Item vItem = this.aPlayer.getCurrentRoom().getItem(pCommand);
        this.aPlayer.take(vItem,this.aPlayer.getCurrentRoom());
        this.aGui.print("You took ");
        this.aGui.println(vItem.getName());
    }
    else{
        this.aGui.println("This item is not in the room.");
    }
}// take
```

```
public void drop(final String pCommand){
    if (this.aPlayer.getItems().containsKey(pCommand)){
        Item vItem = this.aPlayer.getItem(pCommand);
        this.aPlayer.drop(vItem, this.aPlayer.getCurrentRoom());
        this.aGui.print("You droped ");
        this.aGui.println(vItem.getName());
    }
    else
        this.aGui.println("You don't have his item.");
}// drop
```

Et dans la méthode *Player* afin de les stocker dans le nouvel attribut *aItems* de type *HashMap*.

```
public void take(final Item pItem, final Room pRoom){
    if (this.getCurrentRoom().getItems().containsKey(pItem)){
        if (!this.aItems.containsValue(pItem)){
            this.addItem(pItem);
            pRoom.getItems().remove(pItem);

        }
    }
}// take
```

```
public void drop(final Item pItem, final Room pRoom){
    if (!this.getCurrentRoom().getItems().containsKey(pItem)){
        if (this.aItems.containsValue(pItem)){
            this.removeItem(pItem);
            pRoom.getItems().put(pItem.getName(), pItem);
        }
    }
}// drop
```

Ces dernières utilisent alors les méthodes suivantes utilisant les méthodes préexistantes de la classe *HashMap.*

```
public void addItem(final Item pItem){
    this.aItems.put(pItem.getName(),pItem);
}// addItem
```

```
public void removeItem(final Item pItem){
    this.aItems.remove(pItem.getName(),pItem);
}// removeItem
```

```
public Item getItem(final String pItem){
    return this.aItems.get(pItem);
}// getItem
```

7.31.1_ On peut à présent remplacer la *HashMap* par un objet de la classe *ItemList.* Cette nouvelle classe possède alors elle-même la précédente *HashMap* ainsi que toutes les méthodes liées à la prise, et le dépôt d'item dans une pièce. Après les modifications on constate que l'on cherche souvent à vérifier la présence d'item dans un objet *ItemList*. On ajoute donc dans la classe du même nom une méthode permettant de savoir si un item est présent ou non.

```
public boolean itemIsInList(final Item pItem){
    if (this.aItems.containsValue(pItem))
        return true;
    return false;
}// itemIsInList
```

```
public boolean nameIsInList(final String pItem){
    if (this.aItems.containsKey(pItem))
        return true;
    return false;
}// nameIsInList
```

La classe *ItemList* étant aussi utilisée dans la classe *Room* on doit donc déplacer les méthodes qui retournent les descriptions et les noms de la totalité des items présent dans une pièce.

Cela apporte notamment des modifications dans la méthode *createRoom* de la classe *GameEngine* :

```
vCelluleBastille.setItemList(new Item("key", "Big rusted key",3));

vBastille.setItemList(new Item("fire", "This is a fire pit", 10));
```

7.32_ Un joueur ne peut maintenant que transporter des objets s'ils ne sont pas trop lourd. Il faut alors ajouter une condition de poids maximal supporter par le joueur, créer par un attribut, initialisé à zéro à la création de l'objet *Player*. Ce nouvel attribut augmentera à l'appel de la méthode *take* si l'objet n'est pas trop lourd et diminuera à l'appel de *drop.*

```
private int                     aWeight;
```

Il faut modifier les méthodes *take* et *drop* pour qu'elles prennent en compte ce nouveau paramètre.

```
public void take(final Item pItem, final Room pRoom){
    if (pRoom.getItemList().itemIsInList(pItem)){
        if (!this.aItems.itemIsInList(pItem)){
            if ((pItem.getWeight() + this.aWeight) < 8){
                this.aWeight += pItem.getWeight();
                this.aItems.addItem(pItem);
                pRoom.getItemList().removeItem(pItem);
            }
        }
    }
}// take
```

```
public void drop(final Item pItem, final Room pRoom){
    if (!pRoom.getItemList().itemIsInList(pItem)){
        if (this.aItems.itemIsInList(pItem)){
            this.aWeight -= pItem.getWeight();
            this.aItems.removeItem(pItem);
            pRoom.getItemList().addItem(pItem);
        }
    }
}// drop
```

7.33_ On veut que l'utilisateur puisse consulter son inventaire.

```
public void printInventory(){
    this.aGui.println(this.aPlayer.getItemList().getItemsNames());

}// inventory
```

Cette méthode emploie une méthode préexistante dans la classe *ItemList* :

```
public String getItemsNames(){
    if (!this.getItemList().isEmpty()){
        StringBuilder vItemsNames = new StringBuilder("   Items:  ");
        for(Item vI: this.getItemList().values()){
            vItemsNames.append(vI.getName()+"["+vI.getWeight()+"]").append("    ");
        }
        vItemsNames.append("\n");
        return vItemsNames.toString();
    }
    else
        return "  There are no item left.";

}// getItemsNames
```

7.34_ On ajoute l'item cookie par l'intermédiaire de la méthode *createRoom* de la classe *GameEngine.*

```
vSalpetriere.setItemList(new Item("cookie", "Magic cookie",2));
```

```
public void eat(final String pCommand){
    switch (pCommand){
        case "cookie":
            if (this.aPlayer.getItemList().getList().containsKey(pCommand)){
                this.aGui.println("You've eaten. You've gained strength.");
                this.aPlayer.getItemList().getList().remove(pCommand);
                this.aPlayer.setWeightMax(2*this.aPlayer.getWeightMax());
            }
            else
                this.aGui.println("You don't have this item on you.");
            return;

        default:
            this.aGui.println("You shouldn't eat this...");
            return;
    }

}// eat
```

Cette méthode vérifie que l'item mangé soit bien le cookie, modifie le poids maximal pouvant être porté par le *Player* et retire le cookie de la liste d'objets portés.

7.34.1_ Les fichiers de tests (Test.txt ; Resolution.txt) ont été mis à jour.

J'ai ajouté une méthode *light* dans la classe *GameEngine,* afin de vérifier si les conditions de fin de jeu gagnant sont réunies et terminer le jeu :

```
public void light(){
    if (this.aPlayer.getCurrentRoom().getShortDescription().equals("locked a cell of the Bastille.") ){
        if (this.aPlayer.getItemList().nameIsInList("salpetre") &&
        this.aPlayer.getItemList().nameIsInList("coal") &&
        this.aPlayer.getItemList().nameIsInList("sulfur") &&
        this.aPlayer.getItemList().nameIsInList("fire") &&
        this.aPlayer.getItemList().nameIsInList("candle") ){
            this.aGui.println("You create the gun powder, and light it whith the lighted candle.");
            this.aGui.println("Well done! The explosion worked ! You are back in your time zone!");
            this.endGame();

        }
    }
}// light
```

7.34.2_ La javadoc a été générée.

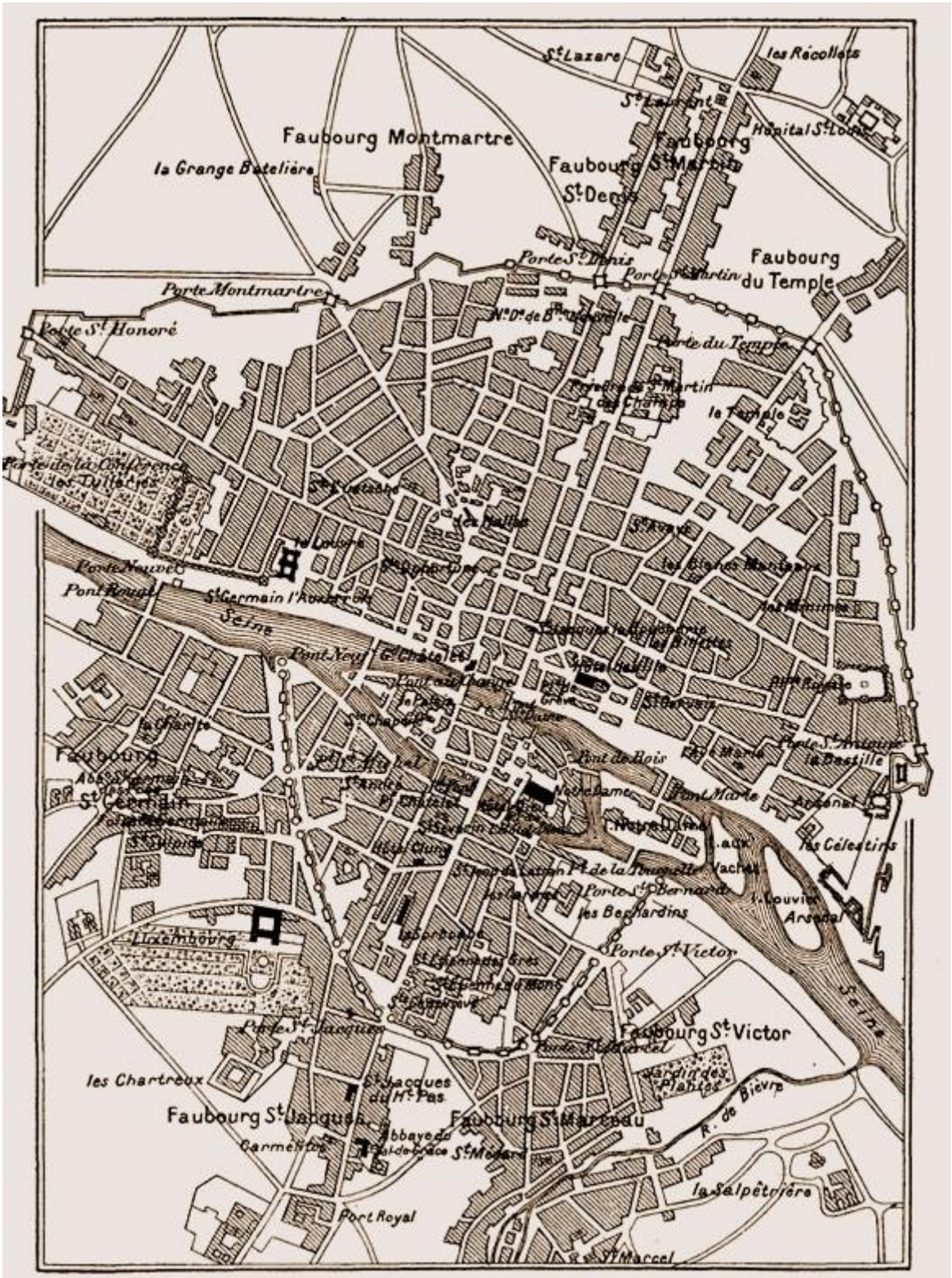
III. Mode d'emploi (installation ou demarrage du jeu)

IV. Declaration obligatoire anti-plagiat

Je déclare avoir produit l'ensemble des lignes de code sans recopie d'une source autre que les fichiers donnés dans le cadre du projet.

V. SOURCES

http://fernandbournon.free.fr/paris/livre-2-chapitre-05.php



Plan de Paris à l'avènement de Louis XIV. Echelle de 1/30 000.