

IPO 2022-2023

RAPPORT PROJET ZUUL



Shi : Kiyowara Fumiaki's Adventure



ESIEE
PARIS

Rapport de :
HAKIM Justine
Groupe 6

I - Informations générales sur le jeu

I.A) Auteur

HAKIM Justine

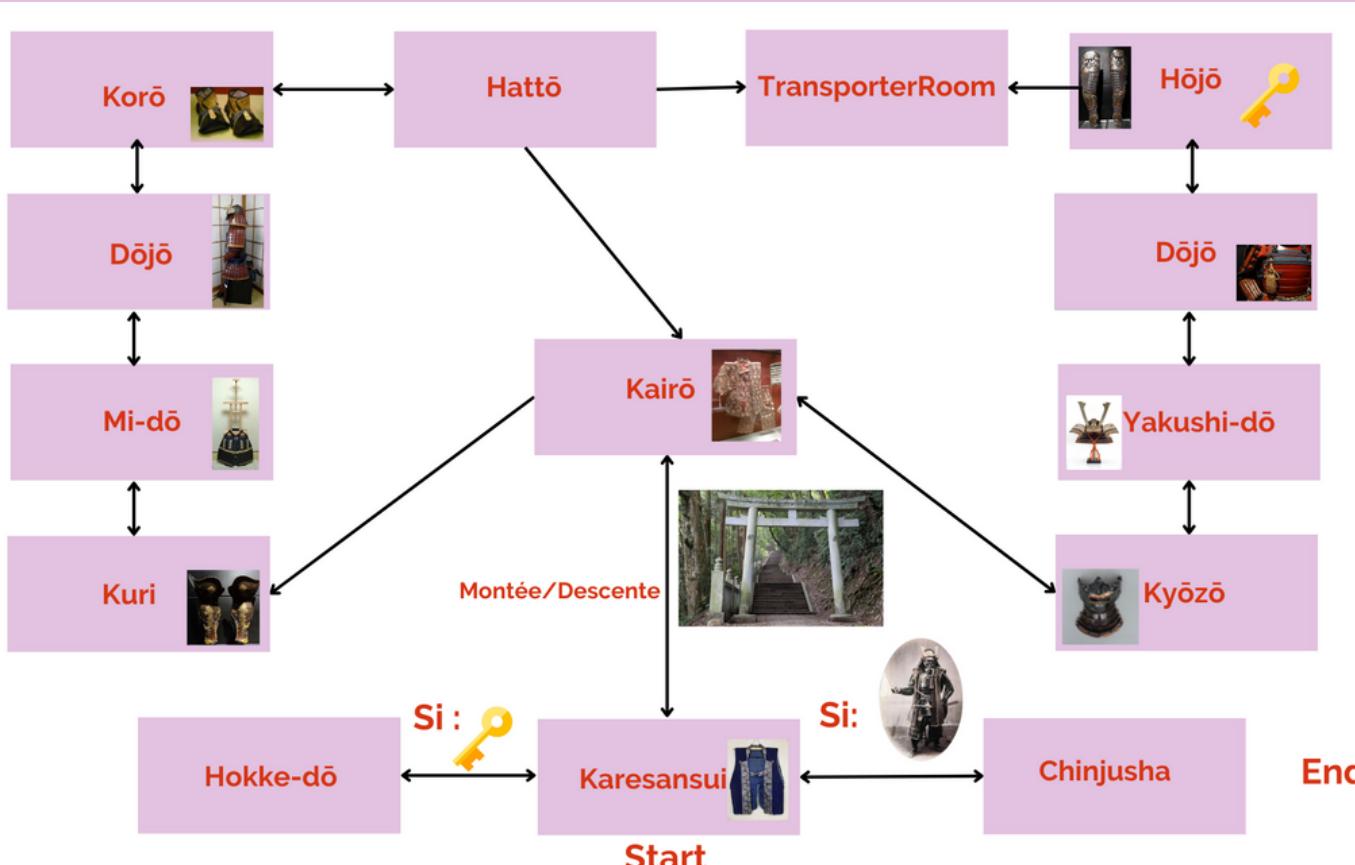
I.B) Thème

En 1278, dans le temple de Shizue, un samouraï tente de récupérer le katana légendaire de sa famille.

I.C) Résumé du scénario complet

Le personnage incarné par le joueur est un samouraï de sexe masculin, âgé de 27ans et du nom de Kiyowara Fumiaki (Nom, prénom). Le katana se transmettant de génération en génération de notre samouraï ayant été volé par le maître du temple, le redoutable Shizue, doit être à tout prix remis en main de ses propriétaires. Cet ennemi est placé dans le sanctuaire du temple, protégeant cet objet mythique tant recherché par notre héros. Pour l'atteindre, il va falloir s'armer de patience, de force et de résistance. En effet, notre héros commencera son périple dans le jardin du temple et va devoir parcourir tous les lieux de celui-ci afin de constituer une armure complète de samouraï. Il devra pour s'acquérir de tous les éléments pour la composer, résoudre des énigmes ou récupérer des objets qui lui permettront d'avancer dans sa quête. Après avoir obtenu une armure de samouraï complète, avec tous les objets collectés, il obtiendra une carte justifiant du titre de samouraï qui lui permettra de pénétrer dans le sanctuaire. Une fois dans ce lieu, si il possède le titre de samouraï et a médité alors Shizue lui rendra shi, son katana.

I.D) Plan complet



I.E) Scénario complet

Kiyowara Fumiaki, un jeune samouraï, doit récupérer Shi, le katana légendaire, se transmettant de génération en génération dans sa famille. Ainsi, il doit rejoindre le maître du temple, Shizue, qui détient cet objet précieux. Pour cela, Fumiaki débute sa quête dans le lieu Karesansui : un jardin de rocaille, souvent présent dans les temples. Il va pouvoir se renseigner auprès de Yuri sur les objets présents dans les différentes pièces et prendre le jinbaori, qui est un manteau sans manches, à l'origine porté sous l'armure puis par-dessus. Il arbore le blason de la famille. L'accès aux salles à l'est et à l'ouest étant pour le moment inaccessible, il emprunte les escaliers et monte jusqu'à Kairō : un long passage couvert semblable à un portique reliant deux bâtiments où il va pouvoir récupérer le hitatare et le hakama qui se portaient traditionnellement sous l'armure. Cet ensemble est confectionné dans une belle étoffe en soie décorée de motifs. Une fois dans cette pièce, il va pouvoir choisir d'explorer l'un des deux bâtiments et pouvoir ramasser dans chacune d'elles, un nouvel élément de son armure. Pour ce faire, dans certaines salles il va devoir résoudre une énigme ou encore échanger des objets à des personnages afin d'obtenir les composants de l'armure nécessaires. Notre personnage pour partir du temple avec le katana doit posséder la carte justifiant de son titre de samouraï ainsi qu'avoir médité dans la pièce fermée à clé et pour cela il doit aller chercher les dix objets qui composent son armure se trouvant dans les différentes pièces du temple. Une fois cette carte dans son inventaire, il va pouvoir accéder au sanctuaire où se trouve son ennemi Shizue ainsi que le katana qu'il recherche, Shi. Pour pouvoir méditer, il doit récupérer la clé dans la salle Hōjō en répondant correctement à une énigme. Une fois tous ces objets récupérés notre joueur peut rejoindre le maître du temple et récupérer le katana. Si le samouraï à réussi à récupérer tous les objets et à méditer avant le combat alors il pourra quitter le temple avec son katana et il aura gagné. Dans le cas contraire, Shizue gardera Shi.

I.F) Détails des lieux, Items et Personnages

- Karesansui: un jardin de rocaille. Il s'agit de la pièce dans laquelle apparaît le joueur. Personnages : Yuri, un moine qui donne des informations. Haruki qui donne le titre de samouraï si le joueur a tous les objets dans son inventaire. Item : Le jinbaori, un manteau sans manches, porté sous l'armure.
- Chinjusha : un petit sanctuaire. Si le joueur possède la carte justifiant de son titre de samouraï il peut accéder à cette salle. Personnage : Shizue son ennemi.

- Kyōzō : dépôt de livres traitant de l'histoire du temple.

Item : Mengu, masque en métal qui assurent la protection complète du visage et de la gorge du samouraï, le livre Torii, temples et sanctuaires japonais. Personnage : Dan, qui est un homme rempli de savoir sur l'histoire du temple et est en charge de la protéger. Il lui donne l'objet s'il ramène un livre qu'il a perdu dans une autre salle.

- Yakushi-dō : un bâtiment dans lequel est vénérée une statue de Yakushi Nyorai. Item : livre perdu par Dan. + Kabuto, un casque composé de diverses plaques en métal forgé, rivetées entre elles auxquelles s'ajoute une série de panneaux souples protégeant le cou. Personnage : Yakushi qu'il doit vénérer pour obtenir l'objet.
- Dōjō : Historiquement le dojo était la salle du temple religieux. Ces salles sont maintenant utilisées pour l'enseignement des arts martiaux. Le dojo est un lieu où l'on progresse. Cette progression est obligatoirement supervisée et contrôlée par un maître. Personnage : Maître Kai donne l'objet. Item : Do, La cuirasse composée de petites lamelles en cuir, trouées, recouvertes de laque et liées entre elles par un laçage de soie. La cuirasse est en deux parties et s'attache dans le dos. + la sauce soja perdu par Sanji. +50 Yen
- Hōjō : les quartiers d'habitation du prêtre responsable du temple. Personnage : prêtre, qui posera une énigme. Item : Si l'énigme est correctement répondu on obtient la clé pour ouvrir Hokke-dō. + Kote, objet dans la pièce à ramasser Les manches faites de tissu et de métal. Elles protègent les bras.
- Hokke-dō : Une salle dont la disposition permet la marche autour d'une statue pour la méditation. Le but de la marche est de se concentrer et de chercher la vérité ultime. Accès si le joueur possède la clé. Personnage : Sogen, lui parler donne un objet qui prouve que le joueur a médité. Item : Shojin Ryori
- Kairō : un long passage couvert semblable à un portique reliant deux bâtiments. Item : Le hitatare et le hakama qui se portent sous l'armure + Shukkō ki le beamer. Cet ensemble est confectionné dans une belle étoffe en soie décorée de motifs. Accès par une montée depuis le jardin Karesansui.
- Hattō : un bâtiment où l'abbé donne des conférences sur les écritures du bouddhisme. Personnage : Abbé
- Korō : tour dans laquelle se trouve un tambour qui marque le passage du temps. Item : Kutsu, chaussures qui sont faites de cuir. Personnage : Katakuri donne une énigme sur le temps afin de récupérer l'item.
- Dōjō : Historiquement le dojo était la salle du temple religieux. Ces grandes salles ont aussi été utilisées par la suite pour l'enseignement des arts martiaux. Le dojo est un lieu où l'on progresse. Cette progression est obligatoirement supervisée et contrôlée par un maître. Personnage : Maître Zoro donne l'objet. Item : Sode, les épaulières. Elles s'attachent avec des cordons à la cuirasse et sont confectionnées comme elle. + 50Yen

- Mi-dō : un terme générique honorifique pour un bâtiment dans lequel est vénérée une statue sacrée. Personnage : Statue sacrée à vénérer pour récupérer l'objet. Item : Kusazuri et Haidate; les jupes et sous-jupes qui protègent le bas du corps.
- Kuri : Un bâtiment abritant les cuisines du temple. Item donné par Sanji si on ramène un item : Suneate. Faites de tissu et de métal, ce sont les jambières qui protègent le bas de la jambe, le genoux, et s'attachent à l'arrière. + des sobas + du saké (donnent plus de force pour porter)

I.G) Situations gagnantes et perdantes

- Situation gagnante : Rejoindre Shizue en ayant le titre de samouraï et médité.
- Situation perdante : Ne pas avoir médité, Ne pas avoir tous les objets, Faire trop de déplacements, Répondre plus de dix mauvaises réponse lors de l'énigme.

I.H) Eventuellement énigmes, mini-jeux, combats, etc.

- Enigme sur le temps par Katakuri : Cette chose, toutes choses dévore. Oiseaux bêtes arbres fleurs. Il ronge le fer et mord l'acier. Il réduit les cailloux en poussière.
- Il tue les rois et détruit les villes. Qui est-ce ?
- Réponse : Le temps.
- Enigme pour la clé : Tu mesures ma vie en heures et je te sers en expirant. Je suis rapide quand je suis mince et lente quand je suis grosse. Le vent est mon ennemi. Réponse : Une bougie.
- Ramener la sauce soja.
- Ramener le livre perdu.

I.I) Commentaires (ce qui manque, reste à faire, ...)

J'ai réussi à quasiment implémenté tout ce qui était prévu dans le scénario mis à part les combats.

II) Réponses aux exercices

Exercice 7.5 :

La duplication de code est un indicateur de mauvaise conception. En effet, lorsque l'on doit modifier une version, on doit aussi appliquer cette modification à l'autre version. Ainsi, on créer une méthode printLocationInfo() dans la classe Game qui nous permettra d'éviter la duplication de l'affichage des sorties de la pièce courante. Grâce à cette nouvelle méthode, on va pouvoir l'utiliser dans la méthode goRoom() et printWelcome().

```

25 /**
26 * Affiche les sorties possibles de la pièce courante
27 */
28 private void printLocationInfo()
29 {
30     System.out.println("Vous êtes actuellement dans " +
31         this.aCurrentRoom.getDescription());
32     if(this.aCurrentRoom.aWestExit != null)
33     {
34         System.out.print("ouest ");
35     }
36     if(this.aCurrentRoom.aEastExit != null)
37     {
38         System.out.print("est ");
39     }
40     if(this.aCurrentRoom.aNorthExit != null)
41     {
42         System.out.print("nord ");
43     }
44     if(this.aCurrentRoom.aSouthExit != null)
45     {
46         System.out.print("sud ");
47     }
48     System.out.println();
49 }

```

Exercice 7.6 :

Cet exercice nous propose d'utiliser un nouveau concept, l'encapsulation afin de réduire le couplage dans les classes et de conduire à une meilleure conception. Pour ce faire, on crée un accesseur getExit() dans la classe Room qui va par exemple si le joueur rentre "nord" l'envoyer dans la pièce qui se situe à la sortie nord.

```

1 /**
2 * Accesseurs d'une des sorties
3 */
4 public Room getExit(String pDirection)
5 {
6     if(pDirection.equals("nord"))
7     {
8         return aNorthExit;
9     }
10    if(pDirection.equals("est"))
11    {
12        return aEastExit;
13    }
14    if(pDirection.equals("sud"))
15    {
16        return aSouthExit;
17    }
18    if(pDirection.equals("ouest"))
19    {
20        return aWestExit;
21    }
22    return null;
23 }
24

```

L'écriture de cette fonction permet de supprimer le switch que nous avions auparavant dans la méthode goRoom() de la classe Game. Grâce à la méthode getExit() de la classe Room qui nous permet d'envoyer le joueur dans la pièce concernée. On remplace donc ce code simplement par :

```

51 Room vNextRoom =
    this.aCurrentRoom.getExit(pDirectionSouhaite.getSecondWord().toLowerCase());

```

toLowerCase permet à la commande de s'exécuter même si l'utilisateur n'écrit pas en majuscules.

Cependant, ce changement enlève la possibilité de dire au joueur que la direction souhaitée est indisponible. Nous devons donc corriger ce problème. J'ai rajouté un attribut dans la classe Room pour résoudre le problème :

```
53 public static final Room UNKNOWN_ROOM = new Room("Pas de pièce");
```

La création de cet attribut m'a permis de l'utiliser dans l'accesseur getExit() que l'on vient de modifier afin de vérifier si la direction est différentes de toutes celles disponible alors on return cet attribut qui indiquera au joueur qu'il n'y a pas de pièce. Par anticipation de l'exercice 7.8, j'ai également vérifié si les directions entrées par l'utilisateur étaient aussi différentes de montée et descente. On obtient donc ce code :

```
55 if(!pDirection.equals("nord") && !pDirection.equals("sud") &&
     !pDirection.equals("est") && !pDirection.equals("ouest") &&
     !pDirection.equals("montée") && !pDirection.equals("descente"))
56     {
57         return UNKNOWN_ROOM ;
58     }
```

De plus, on modifie la classe Game en y ajoutant dans la procédure goRoom() le code suivant :

```
60 if(vNextRoom == Room.UNKNOWN_ROOM)
61     {
62         System.out.println("Direction Inconnue");
63         return;//Arrêter prématurément
64     }
```

Exercice 7.7 :

Dans la classe Room, on ajoute une nouvelle méthode getExitString() qui doit retourner une chaîne de caractère celle-ci contient toutes les sorties :

```
66 public String getExitString()
67     {
68     String vSorties = "";
69     if(aWestExit != null)
70     {
71         vSorties += " ouest";
72     }
73     if(aEastExit != null) {
74         vSorties += " est";
75     }
76     if(aNorthExit != null) {
77         vSorties += " nord";
78     }
79     if(aSouthExit != null) {
80         vSorties += " sud";
81     }
82     return vSorties;
83 }
```

A présent, on peut modifier la méthode printLocationInfo() en y ajoutant la nouvelle méthode que l'on vient de créer puisque la classe Game n'a plus accès aux attributs de la classe Room. En effet, la classe Room produit les informations et ne fait que les retourner car elle n'a pas d'objet sur lequel les appliquer. Tandis que la classe Game affiche ses informations parce qu'elles sont appliquées sur l'objet courant.

```

85 /**
86  * Affiche les sorties possibles de la pièce courante
87 */
88 private void printLocationInfo()
89 {
90     System.out.println("Vous êtes actuellement dans " +
91         this.aCurrentRoom.getDescription());
92     System.out.print("Les sorties : ");
93     System.out.print(this.aCurrentRoom.getExitString());
94     System.out.println();
95 }

```

Exercice 7.8 :

Dans la classe Room, notre objectif est d'avoir plusieurs directions notamment en ajoutant des directions verticales tels que montée et descente. Pour ce faire, on a besoin d'importer la classe HashMap dans la classe Room. Cette nouvelle classe permet de stocker des variables et nous permet d'avoir un seul attribut au lieu d'un attribut par direction. Cette modification facilitera l'ajout futur de directions supplémentaires telles que up et down. Après avoir ajouté un nouvel attribut HashMap, on doit donc modifier le constructeur naturel pour initialiser une nouvelle Hashmap. On doit ensuite modifier la méthode getExit () car celle-ci ne peut plus accéder aux attributs. Enfin, il faut changer la méthode setExits () en la remplaçant par une méthode setExit () qui définira chaque sortie une par une car on sait qu'une Hashmap est extensible et nous n'avons donc pas besoin de mettre les sorties non utilisées en null :

```

97 import java.util.HashMap;
98
99 /**
100 * Classe Room - un lieu du jeu d'aventure Zuul.
101 *
102 * @author HAKIM Justine
103 */
104 public class Room
105 {
106     //Attributs de la classe
107     private String aDescription;
108     private HashMap<String, Room>aExits;//HashMap ("direction", pièce dans cette
109     direction)
110     public static final Room UNKNOWN_ROOM = new Room("Pas de pièce");
111
112     /**
113      * Constructeur naturel
114      * Crée une pièce décrite par la chaîne "description"
115      * Au départ, il n'existe aucune sortie
116      * "description" est une chaîne
117      * @param pDescription Description de la pièce
118     */
119     public Room(final String pDescription ){
120         this.aDescription = pDescription;
121         aExits = new HashMap<String, Room>(); //Créer un nouveau HashMap vide
122     }//Room()
123
124     /**
125      * Renvoie la description de la pièce actuelle
126      * (telle que définie par le constructeur).
127     */
128     public String getDescription(){
129         return this.aDescription;
130     }//getDescription()

```

```

131     /**
132      * Définit une sortie de cette pièce dans la direction indiquée.
133      * @param pDirection Indique la direction de la sortie.
134      * @param pVoisine Indique la pièce dans la direction donnée.
135      */
136     public void setExit(final String pDirection, final Room pVoisine)
137     {
138         aExits.put(pDirection, pVoisine);
139     }//setExit()
140     /**
141      * Renvoie la pièce atteinte si nous nous déplaçons dans la direction
142      * "direction"
143      * S'il n'y a pas de pièces dans cette direction, alors on renvoie null.
144      * Accesseurs d'une des sorties
145      * @param pDirection Direction dont on souhaite connaître la sortie
146      * @return la sortie indiquée
147      */
148     public Room getExit(String pDirection)
149     {
150         if(!pDirection.equals("nord") && !pDirection.equals("sud") &&
151             !pDirection.equals("est") && !pDirection.equals("ouest") &&
152             !pDirection.equals("montée") && !pDirection.equals("descente"))
153         {
154             return UNKNOWN_ROOM ;
155         }
156         return aExits.get(pDirection);
157     }//getExit()

```

Par conséquent, ce changement implique une modification dans la classe Game. En effet, il faut redéfinir les sorties de chaque pièce et s'il y a pas de sortie dans une direction alors on ne note rien. Ainsi on pourra ajouter des sorties vers le haut et vers le bas. Voici un exemple de la définition des différentes sorties de la première pièce :

```

156     //Sorties du jardin
157     vKaresansui.setExit("Nord",vKairo);
158     vKaresansui.setExit("Ouest", vHokke_do);
159     vKaresansui.setExit("Est", vChinjusha);

```

Exercice 7.8.1 :

J'ai donc ajouté un déplacement vertical entre deux pièces.

```

161 vKaresansui.setExit("montée",vKairo);
162 vKairo.setExit("descente",vKaresansui);

```

Exercice 7.9 / 7.10

Dans la classe Room, l'objectif est de retourner les sorties sur un objet. Pour y parvenir, nous avons besoin d'importer dans la classe Room, deux autres classes, la classe Set et la classe Iterator. Cette modification nous offre la possibilité d'utiliser la conception dirigée par responsabilité. Il s'agit d'un processus qui manipule les données stockées dans la classe Room et s'occupe de la communication des informations. Dans HashMap, keySet retourne un ensemble des clés de cette HashMap. Donc, à l'aide de la classe Set et le classe HashMap, on peut stocker toutes nos sorties possibles dans keySet. Lorsqu'on utilise cette fonctionnalité on doit importer la classe Set comme ceci :

```

164 import java.util.Set;
165 import java.util.Iterator;

```

Depuis que nous avons remplacé les quatre attributs par une HashMap la fonction getExitString () dans la classe Room ne peut plus fonctionner. En effet, nous avons besoin d'afficher la totalité des clés de HashMap puisqu'elles sont des chaînes de caractères qui représentent les sorties disponibles :

```
167 /**
168      * Renvoie une description des sorties disponibles
169      *
170      * Par exemple : "Les sorties : Nord Sud "
171      * @return Une chaîne de caractères indiquant une description des sorties
172      * disponibles.
173      */
174     public String getExitString()
175     {
176         String vSorties = "Sorties :";
177         Set<String>vKeys = this.aExits.keySet();
178         for(String vExit : vKeys) {
179             vSorties += " " + vExit;
180         } //for
181     } //getExitString()
```

- 1 On pose une variable String vSorties qui contiendra le texte « Sorties : » qui sera affiché lorsque le jeu voudra afficher les sorties.
- 2 On pose une variable vKeys de type Set qui sera associée à l'ensemble des sorties possibles contenue dans l'expression this.aExits.keySet(). C'est une collection.
- 3 On utilise une boucle FOR EACH avec une condition telle que pour chaque variable de type String vExit de la collection vKeys, elle va ajouter à la String vSorties la variable vExit. Dit autrement, pour chaque tour de boucle, on ajoute à la variable qui stock « Sorties : » une sortie mémoriser dans l'ensemble des clés.
- 4 A chaque fin de boucle, on retourne la String vSorties qui va afficher les sorties.

Exercice 7.10.1 :

La Javadoc a été complétée et générée.

Exercice 7.11 :

L'objectif de cet exercice est de réduire de nouveau le couplage entre la classe Game et la classe Room. De plus, nous devons modifier la classe Room car elle ne respecte pas la conception dirigée par responsabilité. En effet, la classe Room doit créer la description de la pièce. Afin de respecter tous ces points, il suffit de créer une méthode getLongDescription() dans la classe Room, qui va nous permettre de créer la description d'une pièce et de donner les sorties.

```
183 /**
184      * Renvoie une description détaillée de cette pièce sous la forme :
185      * Vous êtes dans (pièce).
186      * Sorties : nord sud.
187      *
188      * @return Une description de la pièce, ainsi que les sorties
189      */
190     public String getLongDescription()
191     {
192         return " Vous êtes dans "+ this.aDescription + "\n" + getExitString();
193     }
194 }
```

Une fois cette modification faîte, il nous suffit de changer également la méthode printLocationInfo() de la classe Game en appelant la méthode que l'on vient de créer.

```
195 /**
196      * Affiche les sorties possibles de la pièce courante
197      */
198     private void printLocationInfo()
199     {
200         System.out.println(this.aCurrentRoom.getLongDescription());
201     }//printLocationInfo()
```

Exercice 7.14 :

L'objectif de cet exercice est de créer une méthode look() afin de pouvoir savoir ce qu'il se trouve dans la pièce actuelle. On a donc besoin d'utiliser une nouvelle méthode de programmation : la localisation des informations. Celle-ci nous permet de pouvoir changer une classe tout en ayant peu d'impact sur les autres. Tout d'abord, on rajoute la commande « regarder » dans le tableau de commande aValidCommands de la classe CommandWords. On peut donc enlever le contenu du constructeur par défaut puisque l'on déclare un attribut statique non modifiable au début de la classe CommandWords tel que :

```
205 //Tableau constant qui contiendra les commandes valides
206     private static final String[] aValidCommands =
207     {"aller", "aide", "quitter", "regarder"};
```

Dans un second temps, on rajoute dans la classe Game une procédure look() qui nous permettra de regarder ce qu'il y a dans la pièce actuelle.

```
239 /**
240      * Procédure qui affiche ce qu'il y a dans la pièce, la description et les
241      * sorties
242     private void look()
243     {
244         System.out.println(this.aCurrentRoom.getLongDescription());
245     }//look()
```

Enfin, on ajoute à la méthode processCommand() de la classe Game la possibilité que le joueur entre la commande « regarder » pour que le programme puisse exécuter la méthode look().

```
257 /**
258      * Appelle la méthode souhaitée par le joueur
259      *
260      * @param pCommand Commande écrite par le joueur
261      * @return Vrai si le joueur veut arrêter le jeu, Faux s'il veut le continuer
262      */
263     private boolean processCommand(final Command pCommand)
264     {
265         if (pCommand.isUnknown()){
266             System.out.println("La demande ne peut pas être prise en compte...");
267             return false;
268         }//if
269         switch(pCommand.getCommandWord()){
270             case "aller":
271                 this.goRoom(pCommand);
272                 return false;
273             case "quitter":
274                 this.quitter(pCommand);
275                 return quitter(pCommand);
276             case "aide":
277                 this.printAide();
278                 return false;
279             case "regarder":
280                 this.look();
281                 return false;
282             default:
283                 System.out.println("Erreur du programmeur : commande non reconnue
284                 !");
285                 return false;
286         }//switch
287     }//processCommand()
```

Exercice7.15 :

A présent, on souhaite ajouter une autre nouvelle commande « manger ». Il suffit alors de procéder exactement de la même manière que précédemment avec la commande « regarder ». Il faut dans un premier temps ajouter la commande « manger » dans le tableau de commande aValidCommands de la classe CommandWords.

```
205 //Tableau constant qui contiendra les commandes valides
206     private static final String[] aValidCommands =
207         {"aller","aide","quitter","regarder","manger"};
```

Dans un second temps, il faut rajouter dans la classe Game une procédure eat() qui affiche un message lorsque le joueur a mangé.

```
248     /**
249      * Procédure qui affiche un message une fois que le joueur a mangé quelque
250      * chose
251      */
252     private void eat()
253     {
254         System.out.println("Vous avez mangé et vous êtes rassasié");
255     }//eat()
```

Enfin, on ajoute à la méthode processCommand() de la classe Game la possibilité que le joueur entre la commande « manger », pour que le programme puisse exécuter la méthode eat().

```
257 /**
258     * Appelle la méthode souhaitée par le joueur
259     *
260     * @param pCommand Commande écrite par le joueur
261     * @return Vrai si le joueur veut arrêter le jeu, Faux s'il veut le continuer
262     */
263     private boolean processCommand(final Command pCommand)
264     {
265         if (pCommand.isUnknown()){
266             System.out.println("La demande ne peut pas être prise en compte...");
267             return false;
268         }//if
269         switch(pCommand.getCommandWord()){
270             case "aller":
271                 this.goRoom(pCommand);
272                 return false;
273             case "quitter":
274                 this.quitter(pCommand);
275                 return quitter(pCommand);
276             case "aide":
277                 this.printAide();
278                 return false;
279             case "regarder":
280                 this.look();
281                 return false;
282             case "manger" :
283                 this.eat();
284                 return false;
285             default:
286                 System.out.println("Erreur du programmeur : commande non reconnue
287 !");
288                 return false;
289         };//switch
290     }//processCommand()
```

Exercice 7.16 :

L'objectif de cet exercice est de créer une méthode showAll() dans la classe CommandWords afin de permettre au joueur de voir toutes les commandes valides. Cependant, la classe CommandWords et la classe Game n'ont pas de couplage entre eux, et ne doivent pas en avoir. Ainsi, pour éviter d'en créer un, on va avoir besoin d'utiliser la classe Parser, qui elle possède un couplage avec la classe Game et un également avec la classe CommandWords. Donc, dans la classe Parser, il faut que l'on crée une méthode showCommands() qui va afficher au joueur toutes les commandes valides en faisant appel à la méthode showAll() de la classe CommandWords. Tout d'abord, on crée showAll() qui va afficher chaque commande disponible :

```
208 /**
209      * Affiche toutes les commandes valides sur System.out
210     */
211    public void showAll()
212    {
213        for(String vCommand : aValidCommands) {
214            System.out.print(vCommand + " ");
215        }//for
216        System.out.println();
217    }//showAll()
218
```

Dans un second temps, on ajoute la méthode showCommands() :

```
293 /**
294      * Affiche une liste des commandes valides.
295     */
296    public void showCommands()
297    {
298        this.aValidCommands.showAll();
299    }//showCommands()
300
```

Enfin, il faut que l'on ajoute dans la méthode printAide() de la classe Game l'appel vers la méthode showCommands() de la classe Parser comme ci dessous :

```
220 /**
221      * Procédure qui affiche les commandes d'aides possibles
222     */
223    private void printAide()
224    {
225        System.out.println("Vous êtes perdu. Vous êtes seul.\n Vous vous baladez dans
226        le temple de Shizue. \n\n Les commandes disponibles sont :\n");
227        aParser.showCommands();
228    }//printAide()
```

Exercice 7.18:

L'objectif de cet exercice est de respecter la règle du couplage implicite entre les classes CommandWords, Parser et Game. On doit ainsi, modifier la méthode showAll que l'on va désormais nommer getCommandList(). Cette méthode va également changer de type et devenir un String. On a besoin de créer une nouvelle variable vResult dans getCommandList() qui permettra de contenir une nouvelle variable, qui possède toutes les commandes disponibles. Il faut qu'elle nous retourne la liste des commandes disponibles à la place de l'afficher comme auparavant. Donc, elle retournera une chaîne de caractère.

```
23      /**
24      * Affiche toutes les commandes valides.
25      * @return Les commandes valides, sinon rien.
26      */
27     public String getCommandList()
28     {
29         String vResult = " ";
30         for(String vCommand : aValidCommands) {
31             vResult += vCommand + " ";
32         } //for
33         return vResult;
34     } //getCommandList()
```

Après avoir terminé cette modification, la méthode showCommands() de la classe Parser doit également être modifiée. En effet, il faut simplement retourner la chaîne de caractère contenue dans getCommandList() afin que showCommand() nous retourne une chaîne de caractère à la place de l'afficher comme auparavant.

```
36      /**
37      * Affiche une liste des commandes valides.
38      * @return La liste des commandes valides.
39      */
40     public String showCommands()
41     {
42         return this.aValidCommands.getCommandList();
43     } //showCommands()
```

Pour terminer, on a besoin également de modifier la procédure printAide() dans la classe Game qui à la place d'utiliser la méthode showCommands de la classe Parser qui permettait d'afficher la liste des commandes disponibles, cette procédure va afficher la liste de commandes.

```
14      /**
15      * Procédure qui affiche les commandes d'aides possibles
16      */
17     private void printAide()
18     {
19         System.out.println("Vous êtes perdu. Vous êtes seul.\n Vous vous baladez dans
20         le temple de Shizue. \n\n Les commandes disponibles sont :\n");
21         System.out.println(aParser.showCommands());
22     } //printAide()
```

Exercice 7.18.1 et Exercice 7.18.2:

Le code de mon projet a été comparé à celui du fichier zuul-better. J'ai du ainsi apporté quelques modifications. Seule la classe Command n'a pas subit de modification. Dans la classe Parser, la méthode showCommands() a été renommé par getCommandString() puisque en relisant, j'ai compris qu'une méthode dont le nom commence par get ne doit rien afficher, et le nom d'une méthode qui retourne quelque chose sans l'afficher ne doit pas commencer par show.

C'est pourquoi, dans la classe Game, j'ai du modifié la procédure printAide qui faisait appel à la méthode showCommands(). J'ai donc remplacé cette instruction par la méthode getCommandString().

Dans la classe CommandWords, j'ai intégré la notion de StringBuilder dans la méthode getCommandList() et j'ai donc du transformé la boucle for each en boucle for.

```
345     /**
346      * Affiche toutes les commandes valides.
347      * @return Les commandes valides, sinon rien.
348      */
349     public String getCommandList()
350     {
351         StringBuilder vCommands = new StringBuilder();
352         for(int i = 0; i < this.aValidCommands.length; i++) {
353             vCommands.append( this.aValidCommands[i] + " " );
354         }//for
355         return vCommands.toString();
356     }//getCommandList()
357 
```

Pour finir, dans la classe Room, j'ai renommé ma méthode getDescription() en getShortDescription(). J'ai également rendu la méthode getExitString() privée tout en y rajoutant la notion de StringBuilder.

```
358     /**
359      * Renvoie une description des sorties disponibles
360      *
361      * Par exemple : "Les sorties : Nord Sud "
362      * @return Une chaîne de caractères indiquant une description des sorties
363      * disponibles.
364     */
365     private String getExitString()
366     {
367         StringBuilder returnString = new StringBuilder("Les sorties sont:");
368         for (String vS : this.aExits.keySet())
369             returnString.append( " " + vS );
370         return returnString.toString();
371     }//getExitString()
```

En conclusion, cet exercice m'a permis de relire entièrement mes classes de manière minutieuse et j'ai pu remarqué qu'il manquait parfois les this, et j'ai également pu modifier ce qui a été décrit précédemment.

Exercice 7.18.3:

Voici les liens de mes images :

- Kuri :

<https://www.alamyimages.fr/l-interieur-de-la-vieille-maison-traditionnelle-japonaise-dans-la-ville-historique-de-tsumago-post-image274204776.html?imageid=CC77738D-0DCA-4DFB-8931-88642FC972DA&p=1&searchId=d8ce8a8304e5d4ed9f1c720ab6395c3b&searchtype=o>

- Karesansui :

<https://www.pinterest.fr/pin/42362052723891858/>

- Hokke-do :

<https://explore.nara.com/?introduce=hokke-do-sangatsu-do-hall>

- Chinjusha :

https://www.japan-experience.com/sites/default/files/styles/scale_740/public/legacy/japan_experience/1505400395393.jpg?itok=6DhdF5fD

- Kairō :

[https://images.partir.com/XCWffVa7_shKypLPtHhiW03qzFo=/750x/filters:sharpen\(0.3,0.3,true\)/incontournables/japon/japon-nara-4.jpg](https://images.partir.com/XCWffVa7_shKypLPtHhiW03qzFo=/750x/filters:sharpen(0.3,0.3,true)/incontournables/japon/japon-nara-4.jpg)

- Kyōzō :

<https://www.alamyimages.fr/photo-image-rokkaku-kyozo-sutra-hexagonale-a-referentiel-danjo-garan-temple-en-zone-koyasan-a-wakayama-japon-wakayama-japon-le-29-octobre-temple-danjo-garan-171166589.html?imageid=9DC3336F-EFDA-405C-97FE-99E28889BEC5&p=454509&pn=1&searchId=b5d443ed88311cb4768ea4bbe386c18d&searchtype=o>

- Yakushi do :

<https://www.alamyimages.fr/photo-image-la-statue-de-la-triade-de-bouddha-mytreya-de-la-période-hakuho-645-710-et-frenétique-par-houonrin-et-daimyoso-bosatsu-dans-le-kodo-hall-temple-yakushiji-nara-82071126.html?imageid=C43AoD16-240E-4CA1-AEF6-7E475565E779&p=29251&pn=1&searchId=4afedf979414ee2743016d10955683dc&searchtype=o>

- Hojo :

https://commons.wikimedia.org/wiki/File:Ryoanji_Temple_-_Kuri_Main_Building_Interior.jpg

- Dojo :

<https://www.pinterest.fr/pin/326440673002055155/>

- Hatto :

<https://www.alamyimages.fr/photo-image-kencho-ji-kamakura-japon-116953200.html?imageid=A1C4D151-458E-4CC1-B0E4-2F905A8C8BDF&p=148099&pn=1&searchId=c5682f297ece3639bc7b8b9bd82c2264&searchtype=o>

- Koro :

<https://www.alamyimages.fr/photo-image-yashamon-porte-par-un-tour-du-tambour-a-taiyuinbyo-le-mausolée-de-shogun-tokugawa-iemitsu-nikko-japon-17-novembre-2015-yashamon-gate-avec-un-tambour-171872351.html?imageid=EC925345-7239-4C6E-A72E-1A15F9DF31F8&p=454509&pn=1&searchId=5e837203ae3e275640c2f7bab781328e&searchtype=o>

- Mi do :

<https://www.epuzzle.info/fr/puzzle/jouer/architecture/2077-statue-dor%C3%A9e-de-bouddha-assis-vietnam#10x6>

- Dojo 2 :

<https://www.pinterest.fr/pin/100134791707523547/>

Exercice 7.18.3:

Le titre de mon jeu est Shi : Kiyowara Fumiaki's Adventure et il a été implémenté dans le jeu et dans le rapport.

Exercice 7.18.5 :

L'objectif de cet exercice est de créer un tableau afin de pouvoir contenir toutes les pièces Room. Pour ce faire, dans la classe Game, on utilise la notion de HashMap. Tout d'abord, on doit importer la classe HashMap à notre classe Game de la manière suivante : 372 **import java.util.HashMap;**

Après avoir fait cette étape, on doit déclarer un attribut dans notre classe Game de la manière suivante : 374 **private HashMap<String, Room> aRooms;**

Enfin, on crée la HashMap dans le constructeur de notre classe Game de la manière suivante :

```
376     /**
377      * Constructeur par défaut
378      */
379     public Game()
380     {
381         this.createRooms();
382         this.aParser = new Parser();
383         this.aRooms = new HashMap<String, Room>();
384     }//Game()
...=
```

On va à présent créer une méthode getRoom() qui nous retournera la pièce par rapport à la description donnée. Cette méthode nous retourne la valeur qui a été associée à cette clé, ou null s'il n'y en a pas. En effet, si on donne la clé vKaresansui on aura alors sa description.

```
386     /**
387      * Récupère une pièce par rapport à une description donnée.
388      *
389      * @param pDescription Description de la pièce
390      * @return Une pièce par rapport à la description donnée ou null s'il n'y a
391      * aucune qui ne correspond.
392      */
393     public Room getRoom(final String pDescription)
394     {
395         return this.aRooms.get(pDescription);
396     }//getRoom()
```

On modifie alors la procédure createRooms(), en ajoutant put qui nous permet d'associer un objet à une clé. En effet, la description de vKaresansui sera associée à vKaresansui.

```
397     //Stockage des pièces
398     this.aRooms.put("Karesansui",vKaresansui);
399     this.aRooms.put("Chinjusha",vChinjusha);
400     this.aRooms.put("Kyozo",vKyozo);
401     this.aRooms.put("Yakushi_do",vYakushi_do);
402     this.aRooms.put("Dojo_1",vDojo_1);
403     this.aRooms.put("Hojo",vHojo);
404     this.aRooms.put("Hokke_do",vHokke_do);
405     this.aRooms.put("Kairo",vKairo);
406     this.aRooms.put("Hatto",vHatto);
407     this.aRooms.put("Koro",vKoro);
408     this.aRooms.put("Dojo_2",vDojo_2);
409     this.aRooms.put("Mi_do",vMi_do);
410     this.aRooms.put("Kuri",vKuri);
```

Exercice 7.18.6:

- a) Les classes Command et CommandWords ne changent pas.
b) Dans la classe Room, un attribut supplémentaire qui sera le nom de l'image de la pièce est nécessaire.
- 412 private String aImageName; //Nom de l'image
- On doit donc modifier le constructeur de la classe pour mettre une image.

```
414        /**
415        * Constructeur naturel
416        * Crée une pièce décrite par la chaîne "description"
417        * Au départ, il n'existe aucune sortie
418        * "description" est une chaîne
419        * @param pDescription Description de la pièce
420        * @param pImageName Nom de l'image de la pièce
421        */
422        public Room(final String pDescription, final String pImageName ){
423            this.aDescription = pDescription;
424            this.aImageName = pImageName;
425            this.aExits = new HashMap<String, Room>(); //Créer un nouveau HashMap vide
426        } //Room()
```

Pour terminer les modifications de la classe Room, on crée donc la méthode supplémentaire demandé que l'on appelle getImageName() et qui retournera le nom de l'image.

```
428        /**
429        * Retourne une chaîne de caractères décrivant le nom de l'image de la pièce
430        * @return String qui décrit le nom de l'image de la pièce
431        */
432        public String getImageName()
433        {
434            return this.aImageName;
435        } //getImageName()
```

- c) Dans la classe Parser, on doit tout d'abord retirer la classe Scanner car la classe StringTokenizer permettra de la remplacer. On a donc besoin d'importer la classe StringTokenizer de la manière suivante :

```
437        import java.util.StringTokenizer;
```

En ce qui concerne les attributs, il n'en reste donc plus qu'un seul :

```
439        //Attributs
440        private CommandWords aValidCommands; // Commande valide
```

Ce qui implique donc la modification du constructeur :

```
442        /**
443        * Constructeur par défaut qui crée les 2 objets prévus pour les attributs
444        */
445        public Parser()
446        {
447            this.aValidCommands = new CommandWords();
448        } // Parser()
```

Pour terminer les modifications, la méthode getCommand(), on doit rajouter un paramètre qui est une chaîne de caractères et qui sera ainsi la ligne rentrée par le joueur. La méthode vérifie donc ensuite si le joueur a écrit deux mots ou un seul.

e) Dans la classe GameEngine, on va donc récupérer la majeur partie des lignes de code de la classe Game. Néanmoins, les System.out.print sont remplacés par this.aGui.print. On doit ajouter deux nouveaux attributs dans la classe :

```
//Attributs
private Parser aParser;// Commande de l'utilisateur
private Room aCurrentRoom;
private UserInterface aGui;//Interface visuelle
private HashMap<String, Room> aRooms;
```

On initialise les attributs donc dans le constructeur :

```
520     /**
521      * Constructeur par défaut pour les objets de la classe GameEngine
522      */
523     public GameEngine()
524     {
525         this.aParser = new Parser();
526         this.aRooms = new HashMap<String, Room>();
527         this.createRooms();
528     }//GameEngine()
```

On va devoir créer la procédure setGUI qui permettra de modifier la valeur de l'interface graphique.

```
530     /**
531      * Procédure qui permet de modifier la valeur de l'interface et utilise
532      * printWelcome
533      *
534      * @param pUserInterface Interface graphique
535      */
536     public void setGUI( final UserInterface pUserInterface )
537     {
538         this.aGui = pUserInterface;
539         this.printWelcome();
540     }//setGUI()
```

La méthode printWelcome() va conserver les mêmes lignes de code sauf qu'elle va montrer l'image de la pièce également.

```
542     /**
543      * Procédure qui permet d'afficher les messages de bienvenue au début du jeu
544      */
545     private void printWelcome()
546     {
547         this.aGui.print( "\n" );
548         this.aGui.println( "Bienvenue dans Shi : Kiyowara Fumiaki's Adventure !" );
549         this.aGui.println( "Shi : Kiyowara Fumiaki's Adventure est un nouveau et
550           incroyable jeu d'aventure." );
551         this.aGui.println( "Essayer 'aide' si vous avez besoin d'aide." );
552         this.aGui.print( "\n" );
553         this.printLocationInfo();
554     }//printWelcome()
```

Dans la méthode createRooms(), les pièces créées ont donc des descriptions et une image. Voici quelques exemples :

```
562         //Création des différents lieux du temple
563         Room vKaresansui = new Room("un jardin de rocallle.",
564             "./Images/Karesansui.png");
564         Room vChinjusha = new Room("un petit sanctuaire.",
565             "./Images/Chinjusha.png");
565         Room vKyozo = new Room("un dépôt de livres traitant de l'histoire du
566             temple.", "./Images/Kyozo.png");
```

```

450     /**
451      * Récupère la commande et renvoie une commande interprétable par les
452      * fonctions
453      * @param pInputLine String qui permet de lire la commande
454      * @return La prochaine commande du joueur.
455     */
456    public Command getCommand(final String pInputLine)
457    {
458        String vWord1;
459        String vWord2;
460
461        StringTokenizer tokenizer = new StringTokenizer( pInputLine );
462
463        if( tokenizer.hasMoreTokens() )
464            vWord1 = tokenizer.nextToken(); //Prend le premier mot
465        else
466            vWord1 = null;
467
468        if ( tokenizer.hasMoreTokens() )
469            vWord2 = tokenizer.nextToken(); // get second word
470        else
471            vWord2 = null;
472
473        // note: we just ignore the rest of the input line.
474
475        // Now check whether this word is known. If so, create a command
476        // with it. If not, create a "null" command (for unknown command).
477
478        if ( this.aValidCommands.isCommand( vWord1 ) )
479            return new Command( vWord1, vWord2 );
480        else
481            return new Command( null, vWord2 );
482    } // getCommand()

```

d) Dans la classe Game, il faut que l'on déplace la quasi totalité des lignes de code dans une nouvelle classe que l'on créera que l'on nommera GameEngine. En effet, les seuls choses que l'on gardera dans la classe Game sont les attributs et le constructeur même s'ils se verront modifiées tous les deux. Les attributs sont à présent un objet UserInterface et un GameEngine, puisque l'on va créer des objets UserInterface et GameEngine par la suite.

```

483   public class Game
484   {
485       //Attributs
486       private UserInterface aGui;
487       private GameEngine aEngine;
488
489       /**
490          * Constructeur par défaut
491          * Crée le jeu et initialiser sa carte interne. Crée l'interface et s'y
492          * connecter.
493       */
494       public Game()
495       {
496           this.aEngine = new GameEngine();
497           this.aGui = new UserInterface( this.aEngine );
498           this.aEngine.setGUI( this.aGui );
499       } //Game()
500   } //Game()

```

Les méthodes eat(), look(), getRoom() et goRoom vont conserver les mêmes lignes de code (il y a simplement la transformation du System.out.print qui est faite). De plus, la méthode printLocation() on rajoute la ligne de code suivant afin d'afficher l'image :

```
722     if( this.aCurrentRoom.getImageName() != null )
723         this.aGui.showImage( this.aCurrentRoom.getImageName());
```

Nous avons également changer le nom de la méthode processCommand() en interpretCommand() car le nom de processCommand() est déjà utilisé dans une autre classe qui sera donc la classe UserInterface.

```
648     /**
649      * Appelle la méthode souhaitée par le joueur
650      * Transforme et analyse la commande de l'utilisateur
651      *
652      * @param pCommandLine Commande écrite par le joueur
653      * @return Vrai si le joueur veut arrêter le jeu, Faux s'il veut le continuer
654      */
655     public void interpretCommand( final String pCommandLine )
656     {
657         this.aGui.println( "> " + pCommandLine );//Affiche les caractères tapés au
658         clavier
659         Command vCommand = this.aParser.getCommand( pCommandLine );//Converti en
660         commande
661
662         if ( vCommand.isUnknown() ) {
663             this.aGui.println( "La demande ne peut pas être prise en compte..." );
664             return;
665         }//if
666
667         String vCommandWord = vCommand.getCommandWord();
668         switch(vCommandWord){
669             case "aller":
670                 this.goRoom(vCommand);
671                 break;
672             case "quitter":
673                 this.quitter(vCommand);
674                 break;
675             case "aide":
676                 this.printAide();
677                 break;
678             case "regarder":
679                 this.look();
680                 break;
681             case "manger" :
682                 this.eat();
683             break;
684         } //switch
685     } //interpretCommand()
```

Voici la méthode printAide() après les changements :

```
685     /**
686      * Procédure qui affiche les commandes d'aides possibles
687      */
688     private void printAide()
689     {
690         this.aGui.println("Vous êtes perdu. Vous êtes seul.\nVous vous baladez dans le
691         temple de Shizue. \n\n Les commandes disponibles sont :\n");
692         this.aGui.println(this.aParser.getCommandString());
693     } //printAide()
```

La méthode quitter() a été modifiée en implémentant la méthode endGame() que l'on a créée.

```
743     /**
744      * Méthode qui permet de quitter le jeu
745      *
746      * @param pCommand Commande rentrée par le joueur
747      * @return Vrai si le joueur écrit "quitter", Faux s'il y a un second mot
748      * rentré par le joueur
749     */
750     private void quitter(final Command pCommand)
751     {
752         if (pCommand.hasSecondWord()){
753             this.aGui.println("Quitter ?");
754         }else {
755             this.endGame();
756         }
757     }
758 }
```

Pour finir les modifications dans la classe GameEngine, nous avons donc créé la méthode endGame() qui affiche un message et termine le jeu.

```
734     /**
735      * Termine le jeu
736      */
737     private void endGame()
738     {
739         this.aGui.println( "Merci d'avoir joué. Au revoir !" );
740         this.aGui.enable( false );
741     }
742 }
```

Par la suite, nous allons donc créer une nouvelle classe que l'on appellera UserInterface. Nous allons commencer par importer toutes les classes nécessaires à la compilation. Les classes commençant par J sont des composants pour des interfaces graphiques. Ainsi, cette classe aura pour objectif de s'occuper de l'interface graphique de notre jeu. Tout d'abord, voici les importations pour la classe UserInterface dont nous avons besoin :

```
786 import javax.swing.JFrame;
787 import javax.swing.JTextField;
788 import javax.swing.JTextArea;
789 import javax.swing.JLabel;
790 import javax.swing.ImageIcon;
791 import javax.swing.JScrollPane;
792 import javax.swing.JPanel;
793
794 import java.awt.Dimension;
795 import java.awt.BorderLayout;
796
797 import java.awt.event.ActionListener;
798 import java.awt.event.ActionEvent;
799 import java.awt.event.WindowAdapter;
800 import java.awt.event.WindowEvent;
801
802 import java.net.URL;
```

Voici la signature, les attributs ainsi que le constructeur de cette classe :

```
810 public class UserInterface implements ActionListener
811 {
812     // Variables d'instance
813     private GameEngine aEngine;
814     private JFrame aMyFrame;
815     private JTextField aEntryField;
816     private JTextArea aLog;
817     private JLabel aImage;
818
819     /**
820      * Constructeur d'objets de classe UserInterface
821      * Construire une interface utilisateur.
822      * En tant que paramètre, un moteur de jeu (un objet qui traite et exécute les
823      * commandes du jeu) est nécessaire.
824      *
825      * @param gameEngine L'objet GameEngine implémentant la logique du jeu.
826      */
827     public UserInterface(final GameEngine pGameEngine)
828     {
829         // initialisation des variables d'instance
830         this.aEngine = pGameEngine;
831         this.createGUI();
832     } // userInterface()
```

La procédure print() permet d'afficher du texte dans une zone de texte :

```
833     /**
834      * Imprimez un texte dans la zone de texte.
835      *
836      * @param pText Un texte saisi
837      */
838     public void print( final String pText )
839     {
840         this.aLog.append( pText );
841         this.aLog.setCaretPosition( this.aLog.getDocument().getLength() );
842     } // print()
```

De même, la procédure println() permet également d'afficher du texte dans la zone de texte mais suivi d'un saut de ligne :

```
844     /**
845      * Imprimez du texte dans la zone de texte, suivi d'un saut de ligne.
846      *
847      * @param pText Un texte saisi
848      */
849     public void println( final String pText )
850     {
851         this.print( pText + "\n" );
852     } // println()
```

La procédure showImage() qui prend en paramètre le nom de l'image de la pièce permet d'afficher un fichier image dans l'interface :

```
854     /**
855      * Afficher un fichier image dans l'interface.
856      *
857      * @param pImageName Le nom de l'image
858      */
859     public void showImage( final String pImageName )
860     {
861         String vImagePath = "" + pImageName; // to change the directory
862         URL vImageURL = this.getClass().getClassLoader().getResource( vImagePath
863         );
864         if ( vImageURL == null )
865             System.out.println( "Image not found : " + vImagePath );
866         else {
867             ImageIcon vIcon = new ImageIcon( vImageURL );
868             this.aImage.setIcon( vIcon );
869             this.aMyFrame.pack();
870         }
871     } // showImage()
```

La procédure enable() permet d'activer ou de désactiver la saisie dans le champ de saisie.

```
872     /**
873      * Permet d'activer ou de désactiver la saisie dans le champ de saisie.
874      *
875      * @param pOnOff
876      */
877     public void enable( final boolean pOnOff )
878     {
879         this.aEntryField.setEditable( pOnOff ); // enable/disable
880         if ( pOnOff ) { // enable
881             this.aEntryField.getCaret().setBlinkRate( 500 ); // cursor blink
882             this.aEntryField.addActionListener( this ); // reacts to entry
883         }
884         else { // disable
885             this.aEntryField.getCaret().setBlinkRate( 0 ); // cursor won't blink
886             this.aEntryField.removeActionListener( this ); // won't react to entry
887         }
888     } // enable()
```

La procédure createGui() permet de mettre en place l'interface graphique. Nous avons complété la première ligne de la méthode avec le nom de notre jeu.

```
890     /**
891      * Mise en place de l'interface utilisateur graphique.
892      */
893     private void createGUI()
894     {
895         this.aMyFrame = new JFrame( "Shi : Kiyowara Fumiaki's Adventure" );
896         this.aEntryField = new JTextField( 34 );
897
898         this.aLog = new JTextArea();
899         this.aLog.setEditable( false );
900         JScrollPane vListScroller = new JScrollPane( this.aLog );
901         vListScroller.setPreferredSize( new Dimension(200, 200) );
902         vListScroller.setMinimumSize( new Dimension(100,100) );
903
904         this.aImage = new JLabel();
905
906         JPanel vPanel = new JPanel();
907         vPanel.setLayout( new BorderLayout() ); // ==> only five places
908         vPanel.add( this.aImage, BorderLayout.NORTH );
909         vPanel.add( vListScroller, BorderLayout.CENTER );
910         vPanel.add( this.aEntryField, BorderLayout.SOUTH );
911
912         this.aMyFrame.getContentPane().add( vPanel, BorderLayout.CENTER );
913
914         // add some event listeners to some components
915         this.aEntryField.addActionListener( this );
916
917         // to end program when window is closed
918         this.aMyFrame.addWindowListener(
919             new WindowAdapter() { // anonymous class
920                 @Override public void windowClosing(final WindowEvent pE)
921                 {
922                     System.exit(0);
923                 }
924             } );
925
926         this.aMyFrame.pack();
927         this.aMyFrame.setVisible( true );
928         this.aEntryField.requestFocus();
929     } // createGUI()
```

La procédure actionPerformed() est une interface ActionListener pour le champ de saisie :

```
931  /**
932   * Interface ActionListener pour le champ de saisie.
933   *
934   * @param pE
935   */
936  @Override public void actionPerformed( final ActionEvent pE )
937  {
938      // no need to check the type of action at the moment
939      // because there is only one possible action (text input) :
940      this.processCommand(); // never suppress this line
941  } // actionPerformed()
942
```

La procédure processCommand() permet de lire une commande lorsqu'elle a été introduite dans le champ de saisie et fait le nécessaire pour la traiter :

```
943  /**
944   * Une commande a été introduite dans le champ de saisie.
945   * Lit la commande et fait le nécessaire pour la traiter.
946   */
947  private void processCommand()
948  {
949      String vInput = this.aEntryField.getText();
950      this.aEntryField.setText( "" );
951
952      this.aEngine.interpretCommand( vInput );
953  } // processCommand()
```

Exercice 7.18.8:

L'objectif de cet exercice est de créer un bouton dans la classe UserInterface. Pour réaliser cette action, il faut importer la classe JButton :

```
954 import javax.swing.JButton;
```

Par la suite, nous avons besoin de créer un attribut aButton de type JButton. Or, je souhaiterai créer plusieurs boutons, donc je vais avoir besoin de créer un attribut pour chaque bouton représentant une commande.

```
956 private JButton aButtonNord;
957 private JButton aButtonMontee;
958 private JButton aButtonDescente;
959 private JButton aButtonSud;
960 private JButton aButtonEst;
961 private JButton aButtonOuest;
962 private JButton aButtonAide;
963 private JButton aButtonQuitter;
```

En surfant sur Internet, j'ai découvert que l'on peut définir le placement des boutons en utilisant la classe GridLayout : JPanel - Comment utiliser un GridLayout en java (codeurjava.com), on peut également changer la couleur du bouton en important la classe Color (BUINO Erwan m'a donné cette information), on réalise donc les importations nécessaires :

```
965 import java.awt.GridLayout;
966 import java.awt.Color;
```

Afin de pouvoir placer nos boutons on va devoir créer un nouveau panneau à droite, c'est-à-dire à l'endroit où seront stockés les boutons :

```
968 private JPanel aPanelEast;
```

Cependant, ayant définit un nouvel attribut, on va avoir besoin de rajouter dans le constructeur son initialisation. On doit donc rajouter dans createGUI() l'initialisation de l'attribut aPanelEast. (La commande setLayout() nous permet d'organiser les boutons dans une configuration ici de 4 x 4) :

```
970 this.aPanelEast = new JPanel();
971     this.aPanelEast.setLayout(new GridLayout(4,4));
```

Dans la procédure createGUI(), il faut créer un nouveau bouton et définir sa position. Voici les étapes suivis pour réaliser chaque bouton avec l'exemple du bouton quitter. Tout d'abord, chaque bouton doit être initialiser de la manière suivante (La string renseigné définit le nom du bouton):

```
974     this.aButtonQuitter = new JButton("Quitter");
```

Dans un second temps, on doit rajouter au bouton un ActionListener pour que lorsque l'on clic sur le bouton, celui-ci réagisse :

```
975     this.aButtonQuitter.addActionListener(this);
```

Le bouton doit à présent être ajouter au panneau :

```
977     this.aPanelEast.add(this.aButtonQuitter);
```

Une fois que le bouton est inclus dans le panneau, on doit créer à la fenêtre le panneau de boutons. J'ai décidé de placer à l'est le panneau car j'ai trouvé cela plus esthétiques :

```
979     vPanel.add(this.aPanelEast, BorderLayout.EAST );
```

Afin que le clic du bouton provoque la commande nécessaire voulue, il faut que dans la méthode actionPerformed(), on indique que l'action se fait que si on appuie sur le bouton de la manière suivante :

```

/**
 * Interface Actionlistener pour le champ de saisie.
 *
 * @param pE Action réalisée
 */
@Override public void actionPerformed( final ActionEvent pE )
{
    if(pE.getSource() == this.aButtonQuitter)
    {
        this.aEngine.interpretCommand("quitter");
    }
    else if (pE.getSource() == this.aButtonNord)
    {
        this.aEngine.interpretCommand("aller nord");
    }
    else if (pE.getSource() == this.aButtonMontee)
    {
        this.aEngine.interpretCommand("aller montée");
    }
    else if (pE.getSource() == this.aButtonDescente)
    {
        this.aEngine.interpretCommand("aller descente");
    }
    else if (pE.getSource() == this.aButtonSud)
    {
        this.aEngine.interpretCommand("aller sud");
    }
    else if (pE.getSource() == this.aButtonEst)
    {
        this.aEngine.interpretCommand("aller est");
    }
    else if (pE.getSource() == this.aButtonOuest)
    {
        this.aEngine.interpretCommand("aller ouest");
    }
    else if (pE.getSource() == this.aButtonAide)
    {
        this.aEngine.interpretCommand("aide");
    }
    else {
        this.processCommand(); // never suppress this line
    }
} // actionPerformed()

```

Après avoir importé la classe Color, on peut donc ajouter des couleurs à nos boutons. Nous allons donc choisir une couleur pour le bouton quitter, par sens logique on choisira la couleur rouge, ainsi toutes les commandes de direction seront vertes et le bouton aide sera de couleur jaune. Donc, pour ajouter des couleurs, on change la couleur du fond. Pour ce faire on créer des variables vCouleur et vCouleur2 afin de créer des couleurs vertes et rouges très clairs. On utilise donc le code RGB de la couleur que l'on ajoute de la manière suivante :

```
this.aButtonQuitter.setBackground(vCouleur);
```

```

Color vCouleur = new Color(255,102,102);
Color vCouleur2 = new Color(102,255,102);

```

Pour conclure, on remarque que lorsque le joueur va appuyer sur le bouton quitter, le message d'au revoir s'affiche et le champ de saisie se désactive mais les boutons des autres commandes sont toujours cliquables. Pour remédier à cela, en surfant sur internet, j'ai trouvé une commande qui permet de fermer la JFrame du jeu afin que le joueur le quitte réellement. Dans la méthode endGame(), j'ai donc rajouter le code suivant afin que si le joueur entre la commande quitter ou clique sur le bouton quitter, une fenêtre de confirmation s'ouvre et un message d'au revoir dans une autre fenêtre s'affiche :

```
/**  
 * Termine le jeu  
 */  
private void endGame()  
{  
    int exit = JOptionPane.showConfirmDialog(null,"Êtes-vous sûr de vouloir  
quitter? :/","Quitter?",JOptionPane.YES_NO_OPTION);  
    if (exit == JOptionPane.YES_OPTION)  
    {  
        JOptionPane.showInternalMessageDialog(null,"Merci d'avoir joué au  
jeu, au revoir !", "Message d'au revoir", JOptionPane.INFORMATION_MESSAGE);  
        System.exit(0);  
    } else {  
        this.aGui.enable(true);  
    }  
}//endGame()
```

Les explications sont trouvées sur les sites suivants

<https://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html>

<https://openclassrooms.com/forum/sujet/java-annuler-la-fermeture-de-la-fenetre-45851>

Show an internal information dialog with the message, 'information':

```
JOptionPane.showInternalMessageDialog(frame, "information",  
    "information", JOptionPane.INFORMATION_MESSAGE);
```

Et si tu veux qu'une fenêtre apparaisse te demandant si tu veux vraiment quitter :

```
addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent e) {  
        int exit = JOptionPane.showConfirmDialog(null,"Êtes-vous sûr de vouloir  
quitter?","Quitter?",JOptionPane.YES_NO_OPTION);  
        if (exit == JOptionPane.YES_OPTION)  
            System.exit(0);  
    }  
});
```

(le code doit être dans un JFrame, ou un Frame)

Exercice 7.19.2 :

L'objectif de cet exercice est de mettre les images de notre jeu dans un répertoire que l'on va appeler Images et placer à la racine du projet. Pour que les images s'affichent, il faut donc changer le chemin d'accès et mettre le nom du répertoire dans lequel se trouve les images. On doit donc modifier dans la procédure createRooms() dans la classe GameEngine les déclarations des pièces et modifier le nom de l'image en rajoutant « .Images/ » :

```
Room vKaresansui = new Room("un jardin de rocallle.",  
    "./Images/Karesansui.png");
```

Exercice 7.20 :

L'objectif de cet exercice est de rajouter des items dans notre jeu. Afin de ne pas surcharger les autres classes, une classe Item a été créée afin de pouvoir mettre des Items dans des pièces et que lorsque le joueur rentre dans la pièce, il puisse voir la description de l'item et le poids de celui-ci. Chaque Item possèdera une description et un poids. Tout d'abord, la classe item va donc contenir tous les attributs d'un Item; son nom, sa description et son poids. On ajoute alors trois attributs :

```
// variables d'instance
private String aName;
private String aDescription;
private double aWeight;
```

On va ainsi devoir initialiser ces attributs dans le constructeur de la classe :

```
/**
 * Constructeur naturel d'objets de classe Item
 *
 * @param pName Nom de l'item
 * @param pDescription Description de l'item
 * @param pWeight Poids de l'item
 */
public Item(final String pName, final String pDescription, final double pWeight)
{
    // initialisation des variables d'instance
    this.aName = pName;
    this.aDescription = pDescription;
    this.aWeight = pWeight;
} // Item()
```

On ajoute ensuite trois accesseurs de la manière suivante :

```
/**
 * Accesseur du nom de l'item
 *
 * @return Nom de l'item
 */
public String getName()
{
    return this.aName;
} // getName()

/**
 * Accesseur de la description de l'item
 *
 * @return Description de l'item
 */
public String getDescription()
{
    return this.aDescription;
} // getDescription()

/**
 * Accesseur du poids de l'item
 *
 * @return Poids de l'item
 */
public double getWeight()
{
    return this.aWeight;
} // getWeight()
} // Item()
```

A présent, nous devons rajouter un attribut aItem dans la classe Room :

```
private Item aItem; //Item
```

Par la suite, on initialise cet attribut dans une procédure, un modificateur setItem().

```
/**  
 * Procédure qui fixe l'item présent dans la pièce  
 *  
 * @param pItem Item à ajouter dans la pièce  
 */  
public void setItem(final Item pItem)  
{  
    this.aItem = pItem;  
}//setItem()
```

On ajoute alors un accesseur si l'on a besoin de récupérer l'Item que l'on appelle getItem().

```
/**  
 * Accesseur de l'item  
 *  
 * @return Item de la pièce  
 */  
public Item getItem()  
{  
    return this.aItem;  
}//getItem()
```

On ajoute un nouvel accesseur getItemString() qui accède à la chaîne de caractères de l'Item et permet également de rendre compte s'il n'existe pas d'Item. Lorsqu'un joueur entre dans une salle, des informations sur l'objet présent dans cette salle doivent être affichées, ou bien "Pas d'objet ici". C'est pourquoi, getItemString() décrira le(s) objet(s) d'une Room, tout comme getExitString() décrit les sorties d'une Room.

```
/**  
 * Accesseur de la chaîne de caractères de l'item.  
 * Décrit les objets d'une Room  
 *  
 * @return Chaîne de caractères de l'item  
 */  
public String getItemString()  
{  
    if(this.aItem == null) {  
        return "Il n'y aucun objets dans cette pièce";  
    }  
    return "Il y a un " + this.aItem.getName() + ". " + "Il s'agit d' " +  
this.aItem.getDescription() + ". " + "Cet objet pèse" + this.aItem.getWeight() + "  
g.";  
}//getItemString()
```

On va maintenant créer un item. Pour ce faire, on procède exactement de la même manière que lorsque l'on a créé des pièces. Tout d'abord, on va créer ces items dans la procédure `createRooms()` de la classe `GameEngine`. Puis on va donc se servir de la procédure `setItem()` que nous avons créées qui prend en paramètre un Item. Elle nous permet donc d'associer l'objet à la pièce voulue. Voici donc un exemple, j'ai créée l'objet "Jinbaori" de poids 300 g que je place dans la première pièce, Karesansui.

```
//Déclaration des Items
Item vJinbaori = new Item ("Jinbaori","un manteau sans manches, porté sous
l'armure",300);

//Association des Items avec la pièce
vKaresansui.setItem(vJinbaori);
```

Afin que le joueur sache s'il existe un objet ou non dans la pièce, il faut que l'information fasse partie de la description de la pièce. On va donc faire un appel de la méthode `getItemString()` dans la méthode `getLongDescription()`. A présent, lorsque le joueur rentre dans une pièce, il connaît le nom de celle-ci, ses sorties et sait s'il y a un item dans la pièce.

```
 /**
 * Renvoie une description détaillée de cette pièce sous la forme :
 * Vous êtes dans (pièce).
 * Sorties : nord sud.
 *
 * @return Une description de la pièce, ainsi que les sorties
 */
public String getLongDescription()
{
    return "Vous êtes dans "+ this.aDescription + ".\n" + getExitString() + ".\n"
+ getItemString();
}//getLongDescription
```

Exercice 7.21 :

Afin de produire toutes les informations concernant un objet présent dans une Room, on utilise la classe Item. La String qui décrit l'objet est créée elle en revanche par la classe Room et elle est affichée par la classe GameEngine. Le changement dans la méthode `getLongDescription()` que l'on a réalisé à l'exercice précédent permet à l'affichage de se faire.

Exercice 7.22 :

On peut à présent ajouter un item dans chaque pièce. L'objectif de cet exercice est de pouvoir rajouter plusieurs items par pièce et pour se faire il faut utiliser une collection. Cette collection se caractérise par une valeur qui sera le nom de l'item associée à une clé qui sera l'item en lui même.

Ainsi, de la même manière que lorsque l'on a crée l'attribut aExits pour définir nos sorties, on va de nouveau faire appel à la notion de HashMap qui permet de contenir les items. On commence donc par ajouter un attribut aItems à la classe Room puisqu'il s'agit de la classe qui contient les items.

```
private HashMap <String, Item> aItems; //HashMap ("Nom de l'objet", objet)
```

On doit alors initialiser dans le constructeur de la classe Room l'attribut :

```
/**  
 * Constructeur naturel  
 * Crée une pièce décrite par la chaîne "description"  
 * Au départ, il n'existe aucune sortie  
 * "description" est une chaîne  
 * @param pDescription Description de la pièce  
 * @param pImageName Nom de l'image de la pièce  
 */  
public Room(final String pDescription, final String pImageName ){  
    this.aDescription = pDescription;  
    this.aImageName = pImageName;  
    this.aExits = new HashMap<String, Room>(); //Créer un nouveau HashMap vide  
    this.aItems = new HashMap<String, Item>(); //Créer un nouveau HashMap vide  
} //Room()
```

J'ai décidé ensuite de changer le nom de la méthode setItem() par addItem() et de la modifier afin de pouvoir ajouter un item dans une pièce grâce à la méthode put, donner par la collection importée.

```
/**  
 * Ajoute un item à la HashMap  
 *  
 * @param pName Clef de la HashMap (nom de l'item)  
 * @param pItem Item à ajouter dans la HashMap  
 */  
public void addItem(final String pName, final Item pItem)  
{  
    this.aItems.put(pName, pItem);  
} //addItem()
```

Bien évidemment l'accesseur getItem() sera changé également puisque l'on peut maintenant utiliser la méthode get du HashMap.

```
/**  
 * Accesseur de l'item  
 *  
 * @param pItem Clef de l'item à récupérer  
 * @return L'item lié à la clef  
 */  
public Item getItem(final String pItem)  
{  
    return this.aItems.get(pItem);  
} //getItem()
```

Il nous reste à modifier la méthode getItemString() dans laquelle, on va conserver le if mais utiliser la méthode du HashMap isEmpty() pour vérifier que lorsque dans la pièce la HashMap est vide c'est à dire qu'elle ne contient pas d'item alors on retourne qu'il n'y aucun item dans la pièce. On crée ensuite une boucle for each avec la notion de StringBuilder dans laquelle la méthode values prend chaque Item de la HashMap et la méthode append rajoute une valeur dans la chaîne de caractères qui sera retournée.

```

/**
 * Accesseur de la chaîne de caractères de l'item.
 * Envoie une chaîne de caractères de tous les items de la pièce
 *
 * @return Chaîne de caractères avec les items
 */
public String getItemString()
{
    if(this.aItems.isEmpty()) {
        return "Il n'y aucun objets dans cette pièce";
    }//if

    StringBuilder returnString = new StringBuilder ("Les items sont : ");
    for(Item vI : aItems.values())
    {
        returnString.append("\n").append(vI.getItemDescription()).append("\n");
    }
    return returnString.toString();
}//getItemString

```

Exercice 7.22.1 :

Dans l'exercice précédent l'objectif était de pouvoir rajouter plusieurs items par pièce. Pour se faire, nous avons utilisé la notion de HashMap car elle nous permet d'accéder facilement à un item sans avoir besoin de connaître un indice. Ainsi, une HashMap était ici la collection idéal car elle permet d'ajouter et de récupérer facilement des éléments afin de ne pas avoir à changer la taille comme pour les tableaux par exemple lorsque l'on voudra rajouter un item.

Exercice 7.22.2 :

Grâce aux méthodes modifiées et l'utilisation de la HashMap, on va pouvoir à présent ajouter nos items à nos pièces en respectant le scénario. Pour l'instant j'ai placé uniquement les items "ramassable" car je souhaite que les autres soient donné par un personnage. Comme demandé dans la consigne, un Item est placé dans la pièce initiale du jeu et au moins une pièce comporte plusieurs Items. Dans la classe GameEngine, on va donc utiliser la procédure addlItem() que l'on vient de modifier afin d'ajouter les items sur la pièce.

```

//Création des Items
Item vJinbaori = new Item ("Jinbaori","C'est un manteau court traditionnel japonais porté par les samouraïs pendant la période Edo (1603-1868). Il est fabriqué à partir de tissus épais et imperméables et est souvent orné de motifs colorés. Le jinbaori était souvent porté par-dessus le hitatare et le hakama pour fournir une couche supplémentaire de protection contre les éléments lors des batailles.",600);
Item vLivreTorii = new Item("Livre Torii, temples et sanctuaires japonais", "Entre bande dessinée et carnet de voyage, ce livre plein d'humour et très documenté vous dit tout sur les sanctuaires shintoïstes, les temples bouddhistes et sur la place du sacré dans la vie des Japonais.", 500);
Item vLivreShinto = new Item("Livre Shinto", "C'est un livre sur la religion shintoïste, qui est la religion traditionnelle du Japon.",500);
Item vLivrePJap = new Item("Livre Promenades Japonaises","C'est un livre sur les promenades dans les endroits historiques et touristiques du Japon.",500);
Item vSauce = new Item ("Sauce Soja","C'est une sauce salée et légèrement sucrée d'origine japonaise, faite à partir de soja, de blé, d'eau et de sel.",200);
Item vSobas = new Item ("Sobas","Ce sont des nouilles fines et grises, généralement faites à partir de farine de sarrasin, consommées chaudes ou froides au Japon.",100);
Item vSake = new Item ("Saké","C'est une boisson alcoolisée japonaise traditionnelle, fabriquée à partir de riz fermenté.",700);
Item vKote = new Item ("Kote","Ce sont des gants de protection utilisés dans les arts martiaux japonais",350);
Item vHetaH = new Item ("Hitatare et hakama","Il s'agit d'un costume traditionnel japonais porté par les hommes. Le hitatare est une veste ample portée sur un pantalon et le hakama est une jupe-culotte ample portée par-dessus.",3000);

```

```

//Placement des Items dans la pièce
vKaresansui.addItem("Jinbaori",vJinbaori);
vKyozo.addItem("Livre Torii, temples et sanctuaires japonais",vLivreTorii);
vKyozo.addItem("Livre Shinto",vLivreShinto);
vYakushi_do.addItem("Livre Promenades Japonaises",vLivrePJap);
vDojo_1.addItem("Sauce Soja",vSauce);
vKuri.addItem("Sobas",vSobas);
vKuri.addItem("Saké",vSake);
vHojo.addItem("Kote",vKote);
vKairo.addItem("Hitatare et hakama",vHeth);

```

Exercice 7.23 :

L'objectif de cet exercice est de créer une commande afin que le joueur puisse revenir dans la pièce précédente sans devoir écrire de direction. En effet, il serait pratique d'avoir une commande nous permettant de pouvoir retourner une pièce en arrière. Pour ce faire, dans la classe GameEngine on va tout d'abord créer un nouvel attribut aPreviousRoom qui est un attribut de type Room et représente la pièce précédente puisqu'il s'agit de la classe qui s'occupe des déplacements.

```
private Room aPreviousRoom;
```

Dans un deuxième temps, on va créer une procédure back() qui prendra en paramètre une commande. La pièce actuelle sera stockée dans une variable et si la pièce précédente n'existe pas alors on affichera «Il n'y a pas de pièces précédentes.». S'il y a une pièce précédente, la pièce actuelle deviendra donc la pièce précédente et inversement puisque la variable contient la pièce courante. Enfin on appelle la méthode printLocationInfo() qui va afficher la description, les sorties et l'image de la pièce.

```

    /**
     * Permet de revenir en arrière, dans la pièce précédente
     *
     * @param pCommand Commande de l'utilisateur
     */
    private void back(final Command pCommand)
    {
        //Vérifie si la commande entrée a bien un second mot
        if(pCommand.hasSecondWord()){
            this.aGui.println("Je ne comprends pas cette commande, utilisez un seul mot.");
            return; // Arrêt prématuré
        }//if
        Room vNextRoom = this.aPreviousRoom;
        //Vérifie s'il y a une pièce précédente
        if(vNextRoom == null)
        {
            this.aGui.println("Il n'y a pas de pièces précédentes.");
            return;//Arrêt prématuré
        }//if

        this.aPreviousRoom = this.aCurrentRoom;
        this.aCurrentRoom = vNextRoom;// Change la pièce actuelle par la pièce suivante
        this.printLocationInfo();//Affiche les informations de la pièce actuelle
    }//back()

```

On doit à présent s'intéresser à la modification de la procédure goRoom() puisque la commande back() implique un déplacement. En effet, on associait la pièce actuelle à la prochaine pièce et il faut maintenant associer l'ancienne pièce à la pièce actuelle, afin qu'on puisse retourner en arrière.

```
//Indique au joueur s'il n'y a pas de sorties
if (vNextRoom == null)
{
    this.aGui.println("Il n'y a pas de portes!");
} else
{
    this.aPreviousRoom = this.aCurrentRoom;
    this.aCurrentRoom = vNextRoom;//Change la pièce actuelle par la pièce
    suivante
    printLocationInfo(); //Affiche les informations sur la pièce
}
}//goRoom()
```

On doit également adapté la procédure interpretCommand() en rajoutant l'appel à la commande back.

```
case "reculer" :
    this.back(vCommand);
    break;
}//switch
}//interpretCommand()
```

Pour terminer les modifications, il faut que l'on rajoute la commande au tableau des commandes valides dans la classe CommandWords pour que cela fonctionne.

```
private static final String[] aValidCommands =
{"aller", "aide", "quitter", "regarder", "manger", "reculer"};
```

Exercice 7.26 :

Dans l'exercice précédent, on remarque que lorsque l'on se déplace trois fois et que l'on tape successivement la commande « back », on ne retourne pas dans la première pièce. En effet, on alterne entre la dernière pièce et celle juste avant car le code nous permet de retourner dans la pièce d'avant selon la pièce actuelle. Pour régler le problème, on va donc créer une pile qui est un type de liste qui va nous permettre dans notre cas de récupérer le dernier élément de la pile avec une unique commande. On va donc créer un Stack des pièces. Tout d'abord, dans la classe GameEngine, on doit importer la classe Stack de la manière suivante :

```
import java.util.Stack;
```

On va donc remplacer notre ancien attribut de type Room par un attribut aPreviousRoom de type Stack.

```
private Stack <Room> aPreviousRoom;
```

Dans le constructeur de la classe GameEngine, il faut initialiser l'attribut.

```
this.aPreviousRoom = new Stack <Room>();
```

On va donc devoir modifier la procédure goRoom(), en utilisant la méthode push() de la classe Stack qui permet d'ajouter un élément à la pile et le retourner. On va ajouter l'attribut de la pièce courante à la pile des anciennes pièces.

```
if (vNextRoom == null)
{
    this.aGui.println("Il n'y a pas de portes!");
} else
{
    this.aPreviousRoom.push(this.aCurrentRoom);
    this.aCurrentRoom = vNextRoom;//Change la pièce actuelle par la pièce
    suivante
    printLocationInfo(); //Affiche les informations sur la pièce
}
```

Pour finir, on va modifier la commande back en utilisant deux autres méthodes de Stack. La première est un booléen qui va nous permettre de vérifier si la pile de pièces est vide : empty(). La deuxième est la méthode pop() qui permet de récupérer un élément en haut de la pile.

```
/**
 * Permet de revenir en arrière, dans la pièce précédente
 *
 * @param pCommand Commande de l'utilisateur
 */
private void back(final Command pCommand)
{
    //Vérifie si la commande entrée a bien un second mot
    if(pCommand.hasSecondWord()){
        this.aGui.println("Je ne comprends pas cette commande, utilisez un seul
mot.");
        return; // Arrêt prématuré
    }//if
    //Vérifie s'il y a une pièce précédente
    if(this.aPreviousRoom.isEmpty())
    {
        this.aGui.println("Il n'y a pas de pièces précédentes.");
        return; //Arrêt prématuré
    }//if

    this.aCurrentRoom = this.aPreviousRoom.pop();// Change la pièce actuelle par
    la pièce suivante
    this.printLocationInfo(); //Affiche les informations de la pièce actuelle
}//back()
```

Exercice 7.26.1 :

Les deux java docs demandés ont été créés.

Exercice 7.28.1 :

Suite à l'exercice 7.28, l'objectif est maintenant d'ajouter une nouvelle commande appelée « test ». On cherche à créer une commande qui nous permet de tester si les commandes principales sont fonctionnelles. Cette commande effectuera les commandes écrites dans un fichier de texte. Cette commande possèdera un second mot qui sera le nom d'un fichier texte qui contiendra des commandes de notre jeu qu'on va devoir lire. Pour ce faire, on doit tout d'abord importer dans la classe GameEngine les classes Scanner, File et FileNotFoundException.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
```

Puisqu'il s'agit d'une nouvelle commande, on ajoute bien évidemment la String « test » au tableau de CommandWords.

```
private static final String[] aValidCommands = {"aller", "aide", "quitter", "regarder", "manger", "reculer", "test"};
```

On ajoute ainsi dans la méthode interpretCommand() de la classe Game Engine l'appel à la méthode test().

```
case "test":  
    this.test(vCommand);  
    break;
```

On va à présent créer cette méthode test() qui prendra en paramètre une commande. Tout d'abord, on va s'assurer qu'il y ait bien un second mot car si ce n'est pas le cas on affiche un message d'erreur. De plus, s'il manque l'extension de fichier on l'ajoute à la fin de la commande pour pouvoir récupérer le bon fichier texte. Si le nom du fichier n'existe pas ou ne correspond cela va créer une erreur. C'est pourquoi, on va utiliser la notion de Try/Catch. Premièrement, dans le bloc TRY on effectue les instructions dans le cas où le fichier existe. On va donc solliciter la classe Scanner. Comme on utilise un fichier de texte, il faut déclarer une variable de type Scanner qui « représentera » ce fichier au sein du programme. On va donc instaurer une boucle while afin que tant que le fichier ait une prochaine ligne alors on exécute les instructions qui vont permettre de lire chaque ligne et de traiter les commandes contenues dans la ligne grâce à la méthode interpretCommand(). Deuxièmement, un bloc Try ne va pas sans son bloc CATCH. Ce dernier va nous permettre de traiter l'erreur. En effet, si le fichier n'existe pas ou que son nom est invalide, le bloc catch nous permet de traiter cette erreur en passant en paramètre un FileNotFoundException puis on affichera un message d'erreur.

```
/**  
 * Méthode de test  
 *  
 * @param pCommand Commande de l'utilisateur  
 */  
private void test(final Command pCommand)  
{  
    if(!pCommand.hasSecondWord()) {  
        this.aGui.println("Que voulez-vous tester?");  
        return;  
    }  
    String vFile = pCommand.getSecondWord();  
    if(!vFile.contains(".txt")){  
        vFile += ".txt";  
    }  
    try {  
        Scanner vScan = new Scanner(new File (vFile));  
        this.aGui.println("Test " + vFile + "...");  
        while(vScan.hasNextLine()){  
            this.interpretCommand(vScan.nextLine());  
        }  
    }catch(final FileNotFoundException pE){  
        this.aGui.println("Désolé, le fichier " + vFile + " n'a pas été trouvé.  
Réessayer");  
    }  
} //test()
```

Exercice 7.28.2 :

J'ai créé 3 fichiers de commandes : un court fichier pour essayer la commande de test avec au moins 3 commandes différentes que l'on appelle court.txt, le parcours idéal pour gagner que l'on appelle gagner.txt et enfin un fichier long avec l'exploration de toutes les possibilités du jeu que l'on appelle long.txt.

Exercice 7.29 :

On peut constater que la classe GameEngine est de plus en plus chargée et il faut donc la décharger en déplaçant certaines méthodes dans une autre classe pour rendre une cohérence d'utilisation à celle-ci et permettre un debug bien plus simple en cas de problème. En effet, cette classe permet de gérer la création de pièces, d'objets, mais aussi les déplacements du joueur. Ainsi, lorsque dans les exercices suivants on va rajouter de nouvelles fonctionnalités et options pour le joueur, la classe GameEngine sera surcharger davantage. Donc, on va penser à une nouvelle conception, c'est-à-dire la création d'une nouvelle classe : Player. Toutes les informations qui concernent le joueur (pièce courante, pièces précédente ...) seront stockées dans celle-ci. Dans cette dernière se trouve donc la pièce courante, un attribut de type GameEngine, les anciennes pièces fréquentées et de nouveaux attributs tels que son pseudo et son poids. Tout d'abord, on crée la classe Player dans laquelle on importe la classe Stack.

```
1 import java.util.Stack;
```

On rajoute les attributs que l'on a présenté :

```
3 public class Player
4 {
5     private String aPseudo;
6     private Room aCurrentRoom; //Pièce actuelle
7     private Stack aPreviousRoom; //Pièce précédente
8     private double aWeight;
9     private GameEngine aEngine;
```

On initialise les attributs dans le constructeur :

```
10 /**
11 Constructeur naturel d'objets de classe Player
12
13 @param pPseudo Pseudo du joueur
14 @param pCurrentRoom Pièce de départ du joueur */
15 public Player(final String pPseudo, final Room pCurrentRoom, final GameEngine pEngine)
16 {
17     this.aPseudo = pPseudo;
18     this.aWeight = 0;
19     this.aCurrentRoom = pCurrentRoom;
20     this.aPreviousRoom = new Stack();
21     this.aEngine = pEngine;
22 }
```

On crée les accesseurs du pseudo, de la pièce actuelle et de la pièce précédente.

```
22 /**
23 Accesseur du Pseudo
24
25 @return Pseudo du joueur */
26 public String getPseudo()
27 {
28     return this.aPseudo;
29 } //getPseudo()
30 /**
31 Accesseur de la pièce actuelle du joueur
32
33 @return Pièce actuelle du joueur */
34 public Room getCurrentRoom()
35 {
36     return this.aCurrentRoom;
37 } //getCurrentRoom()
38 /**
39 Accesseur du Stack de pièces précédentes
40
41 @return Stack des pièces précédentes */
42 public Stack getPreviousRoom()
43 {
44     return this.aPreviousRoom;
45 } //getPreviousRoom()
```

On déplace la procédure goRoom() entièrement dans la classe Player. Pour ce faire, on va utiliser l'attribut de type GameEngine et on crée un accesseur getGUI() dans la classe GameEngine afin de pouvoir y accéder depuis la classe Player et que la méthode soit entièrement déplaçable dans la classe.

```
57 /**
58 Procédure permettant de se déplacer de pièces en pièces
59
60 @param pDirectionSouhaite Direction dans laquelle le joueur souhaite se rendre */
61 public void goRoom(final Command pDirectionSouhaite) {
62     if(!pDirectionSouhaite.hasSecondWord())
63     {
64         this.aEngine.getGUI().println("Où aller");
65     }
66     if(this.aCurrentRoom == Room.UNKNOWN_ROOM)
67     {
68         this.aEngine.getGUI().println("Direction Inconnue");
69     }
70     if (this.aCurrentRoom.getExit(vDirection) == null)
71     {
72         this.aEngine.getGUI().println("Il n'y a pas de portes!");
73     }
74     Room vNextRoom = this.aCurrentRoom.getExit(vDirection);
75     this.aPreviousRoom.push(this.aCurrentRoom);
76     this.aCurrentRoom = vNextRoom;
77 } //goRoom()
```

On déplace également entièrement les méthodes back(), look() et eat() :

```

80 /**
81 Permet de revenir en arrière, dans la pièce précédente
82
83 @param pCommand Commande de l'utilisateur */
84 public void back(final Command pCommand)
85 { //Vérifie si la commande entrée a bien un second mot
86 if(pCommand.hasSecondWord()){ this.aEngine.getGUI().println("Je ne comprends pas
     cette commande, utilisez un seul mot.");
87 return; // Arrêt prématuré
88 }//if
89 //Vérifie s'il y a une pièce précédente
90 if(this.aPreviousRoom.isEmpty())
91 {
92 this.aEngine.getGUI().println("Il n'y a pas de pièces précédentes.");
93 return;//Arrêt prématuré
94 }//if
95 if(this.aPreviousRoom.empty())
96 {
97 this.aEngine.getGUI().println("Il n'y a pas de pièces précédentes.");
98 } else {
99 this.aCurrentRoom = this.aPreviousRoom.pop(); //Change la pièce actuelle par la
     pièce suivante
100 }
101 }//back()
102 /**
103 Procédure qui affiche ce qu'il y a dans la pièce, la description et les sorties */
104 public void look()
105 { this.aEngine.getGUI().println(this.aCurrentRoom.getLongDescription());
106 }//look()
107 /**
108 Procédure qui affiche un message une fois que le joueur a mangé quelque chose */
109 public void eat()
110 {
111 this.aEngine.getGUI().println("Vous avez mangé et vous êtes rassasié");
112 }//eat()

```

Puisque certaines commandes ont été déplacées dans la classe Player, on va utiliser l'attribut créer afin de modifier la méthode interpretCommand(). Prenons pour exemple, la commande aller :

```

        case "aller":
            this.aPlayer.goRoom(vCommand);
            this.printLocationInfo();
            break;

```

Dans la classe GameEngine on a donc besoin de créer un attribut de type Player.

```

115 //Attributs
116     private Parser aParser;// Commande de l'utilisateur
117     private UserInterface aGui;//Interface visuelle
118     private HashMap<String, Room> aRooms;
119     private Player aPlayer;
120     private String aName;

```

On modifie la méthode printLocationInfo() de la même manière :

```

    /**
 * Affiche les sorties possibles de la pièce courante
 */
private void printLocationInfo()
{
    this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription());
    if( this.aPlayer.getCurrentRoom().getImageName() != null)
        this.aGui.showImage( this.aPlayer.getCurrentRoom().getImageName());
}//printLocationInfo()

```

Dans la classe GameEngine, on a donc l'accesseur getGUI() suivant :

```

    /**
     * Accesseur de l'interface
     *
     * @return L'interface
     */
    public UserInterface getGUI()
    {
        return this.aGui;
    }//getGUI()

```

On initialise dans le constructeur le Player de la manière suivante :

```

    /**
     * Constructeur par défaut pour les objets de la classe GameEngine
     */
    public GameEngine(final String pPseudo)
    {
        this.aName = pPseudo;
        this.aParser = new Parser();
        this.aRooms = new HashMap<String, Room>();
        this.createRooms();
        this.aPlayer = new Player(aName, this.aRooms.get("Karesansui"), this);
    }//GameEngine()

```

On change la procédure printWelcome() afin qu'elle affiche "Bonjour" + Pseudo du joueur :

```

    /**
     * Procédure qui permet d'afficher les messages de bienvenue au début du jeu
     */
    private void printWelcome()
    {
        this.aGui.print( "\n" );
        this.aGui.println("Bonjour " + this.aPlayer.getPseudo() + "!");
        this.aGui.println( "Bienvenue dans Shi : Kiyowara Fumiaki's Adventure !" );
        this.aGui.println( "Shi : Kiyowara Fumiaki's Adventure est un nouveau et incroyable" );
        this.aGui.println( "Essayer 'aide' si vous avez besoin d'aide." );
        this.aGui.print( "\n" );
        this.printLocationInfo();
    }//printWelcome()

```

Pour finir les changements de cet exercice, dans la classe Game, on crée un attribut de type GameEngine et dans le constructeur on rajoute la possibilité au lancement du jeu au joueur de rentrer un Pseudo.

```

import javax.swing.JOptionPane;
public class Game
{
    //Attributs
    private UserInterface aGui;
    private GameEngine aEngine;

    /**
     * Constructeur par défaut
     * Créer le jeu et initialiser sa carte interne. Crée l'interface et s'y connecter
     */
    public Game()
    {
        //Demande un Pseudo à l'utilisateur
        String vPseudo = "";
        while(vPseudo.length() == 0) {
            vPseudo = JOptionPane.showInputDialog("Quel est votre pseudo ?");
        }

        this.aEngine = new GameEngine(vPseudo);
        this.aGui = new UserInterface( this.aEngine );
        this.aEngine.setGUI( this.aGui );
    }//Game()
}//Game()

```

Exercice 7.30 :

L'objectif de cet exercice est de permettre au joueur de pouvoir ramasser des objets de la pièce. On va donc ajouter deux nouvelles commandes : take() et drop(). Ces deux méthodes sont de la responsabilité de la classe Player puisqu'elle concerne l'appartenance d'objets du joueur. Tout d'abord, on doit ajouter les deux nouvelles commandes dans la liste des commandes valides dans la classe CommandWords :

```
174     private static final String[] aValidCommands =  
175         {"aller", "aide", "quitter", "regarder", "manger", "reculer",  
176         "test", "prendre", "jeter"};
```

On va avoir besoin de créer un attribut de type HashMap et pour ce faire en premier, il faut importer la classe HashMap dans la classe Player. On crée donc l'attribut de type HashMap qui contiendra la totalité des objets :

```
175     private HashMap<String, Item> aInventory;
```

On initialise cet attribut dans le constructeur de la classe Player :

```
176     this.aInventory = new HashMap <String, Item>();
```

On rajoute un accesseur qui permettra de récupérer un objet grâce à son nom :

```
177     /**  
178      * Accesseur permettant de récupérer un item par son nom  
179      *  
180      * @return Item  
181      */  
182     public Item getItem(final String pItem)  
183     {  
184         return this.aInventory.get(pItem);  
185     }//getItem()
```

On rajoute un second accesseur qui permet de retourner le contenu de l'inventaire. Pour la construction de celui-ci, on prend exemple sur la méthode getExiString() et l'utilisation du StringBuilder faites :

```
187     /**  
188      * Accesseur permettant de renvoyer le contenu de l'inventaire  
189      *  
190      * @return Contenu de l'inventaire  
191      */  
192     public String getInventoryString()  
193     {  
194         if(this.aInventory.isEmpty())  
195         {  
196             return "Ton inventaire est vide, il ne contient aucun objets...";  
197         }//if  
198  
199         StringBuilder returnString = new StringBuilder ("Les objets dans votre  
200         inventaire sont les suivants :");  
201         for(String vName : this.aInventory.keySet()){  
202             returnString.append("\n").append(vName);  
203         }//for()  
204         return returnString.toString();  
205     }//getInventoryString()
```

Pour finir cet exercice, il ne nous reste plus qu'à créer les méthodes take() et drop(). La méthode take() ajoute le Jinbaori présent dans la pièce dans la HashMap de l'inventaire et il supprime évidemment le Jinbaori de la pièce actuelle afin que l'objet ne puisse pas être présent en double. A l'inverse la méthode drop() ajoute un item à la pièce actuelle et le supprime de la HashMap de l'inventaire. Pour l'instant, le code n'est pas complet, il le sera à l'exercice suivant. Nous avons simplement implémenté, une vérification pour savoir si un second mot est présent (le nom de l'objet à prendre ou à lâcher). On vérifie également si le second mot est bien "Jinbaori" puis on prend ou jette le Jinbaori et enfin on affiche un message indiquant au joueur que l'action a été effectuée correctement et on affiche le contenu de l'inventaire après ce changement.

```

140     /**
141      * Procédure qui permet au joueur de ramasser un objet et de le mettre dans
142      * son inventaire
143      */
144     public void take(final Command pCommand)
145     {
146         if(!pCommand.hasSecondWord()) {
147             this.aEngine.getGUI().println("Que voulez-vous prendre ?");
148         } else if(!pCommand.getSecondWord().equals("Jinbaori")) {
149             this.aEngine.getGUI().println("Vous souhaitez mettre le Jinbaori dans
150             votre inventaire ?");
151         } else {
152             this.aInventory.put("Jinbaori",
153                 this.aCurrentRoom.getItem("Jinbaori"));
154             this.aEngine.getGUI().println(" " +
155                 getItem("Jinbaori").getItemDescription());
156             this.aEngine.getGUI().println(getInventoryString());
157         }
158     }//take()

```

On a donc eu besoin pour la méthode drop de rajouter un accesseur :

```

220     /**
221      * Accesseur permettant de récupérer la HashMap des objets présent dans le sac
222      *
223      * @return Inventaire
224      */
225     public HashMap <String, Item> getItemsInventory()
226     {
227         return aInventory;
228     }//getItemsInventory()

```

```

157     /**
158      * Procédure qui permet au joueur de lâcher un des objets de son inventaire
159      */
160     public void drop(final Command pCommand)
161     {
162         if(!pCommand.hasSecondWord()) {
163             this.aEngine.getGUI().println("Que voulez-vous jeter ?");
164         } else if(!pCommand.getSecondWord().equals("Jinbaori")) {
165             this.aEngine.getGUI().println("Vous souhaitez jeter le Jinbaori de
166             votre inventaire ?");
167         } else {
168             this.aEngine.getGUI().println("Jinbaori n'est à présent plus dans
169             votre inventaire.");
170             this.aInventory.remove("Jinbaori");
171             this.aEngine.getGUI().println(getInventoryString());
172         }
173     }//drop()

```

Afin que les deux commandes fonctionnent correctement, il faut les ajouter dans le switch de la méthode interpretCommand() dans la classe GameEngine :

```
205     case "prendre" :
206         this.aPlayer.take(vCommand);
207         break;
208     case "jeter" :
209         this.aPlayer.drop(vCommand);
210         break;
```

Exercice 7.31 :

L'objectif de cet exercice est donc de pouvoir transporter autant d'objets que l'on souhaite. Pour ce faire, on modifie bien évidemment les méthodes take() et drop() dans la classe Player. On vérifie premièrement s'il y a un second mot, et si l'il existe, on le récupère et on vérifie que l'objet existe dans la pièce actuelle dans laquelle se situe le joueur. Dans le cas contraire, l'objet que l'on demande de prendre est donc nul alors on informe le joueur que l'objet demandé n'existe pas. Pour le reste de la méthode take(), on la modifiée légèrement, elle prend désormais un paramètre en compte, qui est une chaîne de caractères et on affiche le nom de l'objet.

```
1      /**
2       * Procédure qui permet au joueur de ramasser un objet et de le mettre dans
3       * son inventaire
4       *
5       * @param pCommand Une commande entrée par l'utilisateur
6       */
7      public void take(final Command pCommand)
8      {
9          if(!pCommand.hasSecondWord()) {
10              this.aEngine.getGUI().println("Que voulez-vous prendre ?");
11              return;
12          }
13          String vPprise = pCommand.getSecondWord();
14          if(this.aCurrentRoom.getRoomItems().getItem(vPprise) == null)
15          {
16              this.aEngine.getGUI().println("Désolé... Cette objet n'existe
17              pas...");
18              return;
19          }
20          Item vItem = this.aCurrentRoom.getRoomItems().getItem(vPprise);
21          this.aEngine.getGUI().println(" " + vPprise + " est à présent dans votre
22          inventaire.");
23          this.aItems.addItem(vPprise, vItem);
24          this.aCurrentRoom.removeItem(vItem.getName());
25      }
26  } //take()
```

Pour ce qui est de la méthode drop(), on procède exactement de la même manière, à l'exception que on regarde si l'objet existe dans le sac, et non dans la pièce actuelle où le joueur se situe.

```

24     /**
25      * Procédure qui permet au joueur de lâcher un des objets de son inventaire
26      *
27      * @param pCommand Une commande entrée par l'utilisateur
28      */
29     public void drop(final Command pCommand)
30     {
31         if(!pCommand.hasSecondWord())
32         {
33             this.aEngine.getGUI().println("Que voulez-vous jeter ?");
34             return;
35         }
36         String vPoubelle = pCommand.getSecondWord();
37         Item vItem = this.aItems.getItem(vPoubelle);
38         if(vItem == null)
39         {
40             this.aEngine.getGUI().println("Désolé... Cette objet n'est pas dans
41             votre inventaire...");
42             return;
43         }
44         this.aItems.removeItem(vPoubelle);
45         this.aCurrentRoom.addItem(new Item(vItem.getName(),
46             vItem.getDescription(),vItem.getWeight()));
47         this.aEngine.getGUI().println("Vous n'avez plus " + vPoubelle + " dans
48             votre inventaire.");
49     } //drop()

```

La suite du code sera expliqué dans l'exercice suivant avec la création de la classe `ItemList`.

Exercice 7.31.1 :

On remarque après tous ces changements que dans la classe `Room` ainsi que dans la classe `Player`, on utilise une `HashMap` qui contient chez toutes les deux un tableau d'objets et les mêmes méthodes. Pour éviter la duplication de code, on crée une nouvelle classe `ItemList` qui contiendra ce tableau et ces méthodes et on pourra ainsi alléger les classes `Room` et `Player`. On crée donc la classe `ItemList` dans laquelle, on importe la classe `HashMap` au préalable afin de pouvoir créer un attribut de type `HashMap` `aItems` qui stocke les `Items` et on l'initialise dans le constructeur.

```

65     import java.util.HashMap;
66 public class ItemList
67 {
68     // variables d'instance
69     private HashMap <String, Item> aItems;//HashMap ("Nom de l'objet", objet)
70
71     /**
72      * Constructeur d'objets de classe ItemList
73      */
74     public ItemList()
75     {
76         // initialisation des variables d'instance
77         this.aItems = new HashMap <String, Item>() ;
78     } //ItemList()

```

On crée les accesseurs nécessaires `getItem()` qui permet de récupérer un objet à partir de sa clé et `getItemsInventory()` qui permet de récupérer la liste entière sous forme de `HashMap` :

```

80     /**
81      * Accesseur de l'item
82      *
83      * @param pItem Clef de l'item à récupérer
84      * @return L'item lié à la clef
85      */
86     public Item getItem(final String pItem)
87     {
88         return this.aItems.get(pItem);
89     } //getItem()

```

```

91     /**
92      * Accesseur permettant de récupérer la HashMap des objets présent dans le sac
93      *
94      * @return Inventaire
95      */
96     public HashMap <String, Item> getItemsInventory()
97     {
98         return aItems;
99     }//getItemsInventory()

```

On y ajoute les méthodes addItem() et removeItem() :

```

101    /**
102     * Ajoute un item à la HashMap
103     *
104     * @param pName Clef de la HashMap (nom de l'item)
105     * @param pItem Item à ajouter dans la HashMap
106     */
107    public void addItem(final String pName, final Item pItem)
108    {
109        this.aItems.put(pName, pItem);
110    }//addItem()
111
112    /**
113     * Supprime un Item de notre HashMap
114     *
115     * @param pName Nom de l'Item
116     */
117    public void removeItem(final String pName)
118    {
119        this.aItems.remove(pName);
120    }//removeItem()

```

Pour finir la création de cette classe, on y ajoute un nouvel accesseur getItemString() qui renvoie une chaîne de caractères des objets présents dans la pièce ou dans l'inventaire:

```

122    /**
123     * Accesseur de la chaîne de caractères de l'item.
124     * Envoie une chaîne de caractères de tous les items de la pièce
125     *
126     * @return Chaîne de caractères avec les items
127     */
128    public String getItemString()
129    {
130        StringBuilder returnString = new StringBuilder ();
131        if(this.aItems.isEmpty())
132        {
133            returnString.append("Désolé...Il n'y a pas d'objets...");
134        } else {
135            for(String vItemName : this.aItems.keySet()){
136                returnString.append(" " + vItemName);
137            } //for()
138        } //else if
139        return returnString.toString();
140    }//getItemString()

```

Dans la classe Player, nous avons donc besoin de créer un attribut de type ItemList que l'on initialise dans le constructeur :

```

334        private ItemList aItems;
335        this.aItems = new ItemList ();
336

```

On crée l'accesseur nécessaire à cet attribut getItems() :

```

337    /**
338     * Accesseur des items
339     *
340     * @return La liste des items
341     */
342    public ItemList getItems()
343    {
344        return this.aItems;
345    }//getItems()

```

On utilise donc cet attribut dans les méthodes take() et drop() afin de pouvoir accéder aux méthodes addItem(), removeItem() et getItem().

Dans la classe Room, on crée également un attribut de type ItemList que l'on initialise dans le constructeur :

```
347     private ItemList aRoomItems;  
348     this.aRoomItems = new ItemList();
```

On crée de nouveau l'accesseur nécessaire à cet attribut getRoomItems() afin de pouvoir l'utiliser dans les méthodes take() ainsi que look():

```
349     /**  
350      * Accesseur de l'item de la pièce  
351      *  
352      * @return L'item de la pièce  
353      */  
354     public ItemList getRoomItems()  
355     {  
356         return this.aRoomItems;  
357     }//getRoomItems()
```

On modifie la méthode getLongDescription() en y ajoutant la liste des items de la pièce actuelle :

```
359     /**  
360      * Renvoie une description détaillée de cette pièce sous la forme :  
361      * Vous êtes dans (pièce).  
362      * Sorties : nord sud.  
363      *  
364      * @return Une description de la pièce, ainsi que les sorties  
365      */  
366     public String getLongDescription()  
367     {  
368         return "Vous êtes dans " + this.aDescription + "\n" + this.getExitString()  
            + "\n" + "Les objets dans cette pièce sont :" + this.aRoomItems.getItemString();  
369     }//getLongDescription()
```

Exercice 7.32 :

L'objectif de cet exercice est, à présent que l'on peut récupérer des objets, de faire attention à leur poids. Le joueur possède un poids maximal qu'il peut porter à ne pas dépasser. Donc nous avons besoin de créer une restriction qui va indiquer si le joueur peut porter un objet. Pour ce faire, dans la classe Player, on ajoute un nouvel attribut aWeight qui représentera le poids actuel que le joueur porte. Il est donc initialisé dans le constructeur à 0 car au début du jeu il est à 0. On ajoute également un attribut aWeightMax qui représentera le poids maximal que le joueur peut porter, il sera fixé dans le constructeur à 5kg.

```
371     private double aWeight;  
372     private double aWeightMax;  
373     this.aWeight = 0;  
374     this.aWeightMax = 5000.0;
```

Après avoir ajouté les accesseurs et un modificateur nécessaire, on crée une fonction booléenne canTake() afin de vérifier si le poids de l'item permet au joueur de le porter ou non :

```

376      /**
377      * Accesseur du poids maximal que le joueur peut porter
378      *
379      * @return Le poids Maximal que le joueur peut porter
380      */
381     public double getWeightMax()
382     {
383         return this.aWeightMax;
384     }//getWeightMax()

385
386     /**
387     * Modificateur du poids maximal que le joueur peut porter.
388     *
389     * @param pWeightMax Le nouveau poids maximal que le joueur peut porter
390     */
391     public void setWeightMax(final double pWeightMax)
392     {
393         this.aWeightMax = pWeightMax;
394     }//setWeightMax()

395
396     /**
397     * Accesseur du poids total des objets portés par le joueur
398     *
399     * @return Le poids total des objets portés par le joueur
400     */
401     public double getWeight()
402     {
403         return this.aWeight;
404     }//getWeight()

432     /**
433     * Procédure qui permet de savoir si le joueur peut prendre ou pas l'objet
434     *
435     * @return true si la somme du poids passé en paramètre et du poids actuel
436     * porté par le joueur ne dépasse pas le poids maximal que le joueur peut porter
437     */
438     public boolean canTake(final double pWeight)
439     {
440         return (this.aWeight + pWeight <= this.aWeightMax);
441     }//canTake()

```

Pour finaliser les modifications, on doit donc modifier les méthodes take() et drop() de tel sorte que le poids actuel du joueur se mette à jour à chaque fois qu'il prend ou qu'il dépose un item. On doit rajouter une condition dans la méthode take() afin de vérifier que la restriction soit respectée :

```

274     /**
275      * Procédure qui permet au joueur de ramasser un objet et de le mettre dans
276      * son inventaire
277      * @param pCommand Une commande entrée par l'utilisateur
278      */
279     public void take(final Command pCommand)
280     {
281         if(!pCommand.hasSecondWord())
282         {
283             this.aEngine.getGUI().println("Que voulez-vous prendre ?");
284             return;
285         }
286         String vPrise = pCommand.getSecondWord();
287         Item vItem = this.aCurrentRoom.getRoomItems().getItem(vPrise);
288         if(this.aCurrentRoom.getRoomItems().getItem(vPrise) == null)
289         {
290             this.aEngine.getGUI().println("Désolé... Cette objet n'existe
291 pas...");
292             return;
293         }
294         else if(canTake(vItem.getWeight())==false){
295             this.aEngine.getGUI().println("Vous portez déjà une charge lourde.
296             Vous ne pouvez pas prendre cet objet.");
297             return;
298         }
299         this.aEngine.getGUI().println(" " + vPrise + " est à présent dans votre
300         inventaire.");
301         this.aItems.addItem(vPrise, vItem);
302         this.aCurrentRoom.getRoomItems().removeItem(vItem.getName(), vItem);
303         this.aWeight += vItem.getWeight();
304     }//take()

```

```

301     /**
302      * Procédure qui permet au joueur de lâcher un des objets de son inventaire
303      *
304      * @param pCommand Une commande entrée par l'utilisateur
305      */
306     public void drop(final Command pCommand)
307     {
308         if(!pCommand.hasSecondWord())
309             this.aEngine.getGUI().println("Que voulez-vous jeter ?");
310             return;
311         }//if
312         String vPoubelle = pCommand.getSecondWord();
313         Item vItem = this.aItems.getItem(vPoubelle);
314         if(vItem == null)
315             this.aEngine.getGUI().println("Désolé... Cette objet n'est pas dans
316             votre inventaire...");
317             return;
318         }//if
319         this.aItems.removeItem(vPoubelle, vItem);
320         this.aCurrentRoom.getRoomItems().addItem(vItem.getName(), vItem);
321         this.aEngine.getGUI().println("Vous n'avez plus " + vPoubelle + " dans
322             votre inventaire.");
323     }//drop()

```

Exercice 7.33 :

L'objectif de cet exercice est de créer une nouvelle commande qui s'appellera « inventory » et qui va permettre au joueur de pouvoir afficher les items que portent le joueur et le poids total qu'il transporte. Pour ce faire, on va utiliser l'accesseur créé dans la classe ItemList que l'on avait nommé getItemString(). A présent, on peut créer la méthode inventory() dans la classe Player dans laquelle on va créer une variable qui représente la liste d'item que possède le joueur grâce à l'appel à getItemString(). On teste ensuite la valeur de cette variable et on affiche un message selon l'état de la liste (vide ou pas) :

```

406     /**
407      * Commande qui permet de montrer l'inventaire du joueur
408      */
409     public void inventory()
410     {
411         String vInventory = this.aItems.getItemString();
412         if(vInventory.equals("Les objets dans votre inventaire sont :")){
413             this.aEngine.getGUI().println("Votre inventaire est vide.");
414         } else {
415             this.aEngine.getGUI().println("Dans vote inventaire vous avez : " +
416             vInventory + ". Votre poids total est de " + this.aWeight);
417         }
418     }//inventory()

```

Pour finir les modifications, on va ajouter la commande à la liste des commandes valides dans la classe CommandWords() et on ajoute à la méthode interpretCommand(), la nouvelle commande :

```

418
419         private static final String[] aValidCommands =
420             {"aller","aide","quitter","regarder","manger","reculer",
421             "test","prendre","jeter","inventaire"};
422             case "inventaire":
423                 this.aPlayer.inventory();
424                 break;

```

Exercice 7.34 :

L'objectif de cet exercice est de donner la possibilité au joueur de pouvoir manger un objet. En particulier, on cherche à créer la possibilité au joueur de manger un objet qui lui augmentera sa capacité à porter un plus grand nombre d'items. Dans mon scénario, j'ai décidé que cet objet soit le saké. Il faut donc modifier la méthode eat() en conséquence afin de pouvoir manger les items comestibles. On crée la méthode eat() en reprenant une structure similaire que pour drop() et take(), ainsi on ajoute une commande tapée en paramètre. Dans la méthode, lorsque l'item tapé est le saké alors, on le retire de la pièce et on le récupère pour le joueur. Grâce à l'accessseur et au modificateur du poids, on peut donc ajouter à la capacité maximal 1500 grammes. On donne également la possibilité au joueur de manger les sobas et la sauce soja.

```
424          /**
425           * Procédure qui affiche un message une fois que le joueur a mangé quelque chose
426           */
427      public void eat(final Command pCommand)
428      {
429          if(!pCommand.hasSecondWord()){
430              this.aEngine.getGUI().println("Que voulez-vous manger ?");
431              return;
432          }
433          String vItem = pCommand.getSecondWord();
434          Item vMange = this.aItems.getItem(vItem);
435          if(vMange == null) {
436              this.aEngine.getGUI().println("Aucun objet que vous portez peut se manger...");
437          } else if(vItem.equals("Sauce.Soya") || vItem.equals("Sobas")){
438              this.aEngine.getGUI().println("Vous avez mangé " + vItem + ".");
439              this.aItems.removeItem(vItem, vMange);
440              this.aWeight -= vMange.getWeight();
441          } else if (vItem.equals("Saké")){
442              this.aItems.removeItem(vItem, vMange);
443              this.setWeightMax(getWeightMax() + 1500);
444              this.aWeight -= vMange.getWeight();
445              this.aEngine.getGUI().println("Vous avez bu une bouteille de Saké. Elle vous redonne la force de porter de nouveaux objets. Votre poids maximum est à présent de : " + getWeightMax() + " !");
446          } else {
447              this.aEngine.getGUI().println("Aucun objet que vous portez peut se manger... ou alors vous n'avez plus faim ! ");
448          }
449      }//eat()
```

Exercice 7.34.1 :

Les fichiers de test ont été modifiés et mis à jour.

Exercice 7.34.2 :

Les deux java docs ont été complétées et régénérées.

Exercice 7.42 :

L'objectif de cet exercice est d'ajouter à présent un temps limite à notre jeu afin que lorsque celui-ci soit dépassé, le jeu se termine. Pour cet exercice nous n'avons pas besoin d'utiliser le temps réel. On se contentera donc d'utiliser le nombre de commandes entrées/activées par le joueur afin de le limiter.

Ces modifications seront à apporter dans la classe Player puisque le comptage des déplacements concerne le joueur. On ajoute donc un nouvel attribut de type entier aMaxDeplacements dans la classe Player que l'on initialisera à 100 dans le constructeur de celle-ci.

```
451     private int aMaxDeplacements;//Nombre de déplacements maximum avant de perdre  
452         this.aMaxDeplacements = 100;
```

On ajoute un accesseur à cet attribut :

```
453     /**  
454      * Accesseur du nombre de déplacements restants  
455      *  
456      * @return Entier représentant le nombre de déplacements restants avant que le  
jeu ne se termine  
457      */  
458     public int getMaxDeplacements(){  
459         return this.aMaxDeplacements;  
460     }//getMaxDeplacements()
```

Dans les méthodes goRoom() et back(), on doit donc ajouter une ligne de code permettant de décrémenter les déplacements au compteur de déplacements une fois que l'action a été réalisée.

On doit à présent modifiée la classe GameEngine en créant une procédure qui nous permettra de savoir quand le temps est dépassé. De plus, lorsque le compteur de déplacements est à 0 alors il arrête le jeu :

```
500     /**  
501      * Procédure qui vérifie si le compteur de déplacements n'a pas atteint 0  
502      *  
503      * Si le compteur de déplacements est à 0 alors le jeu s'arrête  
504      */  
505     public void timerEnd() {  
506         if(this.aPlayer.getMaxDeplacements() == 0){  
507             JOptionPane.showInternalMessageDialog(null,this.aPlayer.getPseudo() +  
" a fait trop de déplacements et a perdu.", "Message d'au revoir",  
JOptionPane.INFORMATION_MESSAGE);  
508             System.exit(0);  
509         } //if()  
510     }//timerEnd()
```

On doit ainsi appeler cette procédure à la fin des méthodes goRoom() et back() :

```
462         this.aMaxDeplacements -= 1;  
463         this.aEngine.timerEnd();
```

Le fait d'appeler la fonction à cet endroit permet de ne pas avoir à faire un dernier déplacement avant que le jeu ne soit considéré comme perdu.

Afin que le joueur puisse connaître l'avancée du compteur de déplacements, on rajoute dans la méthode printWelcome() :

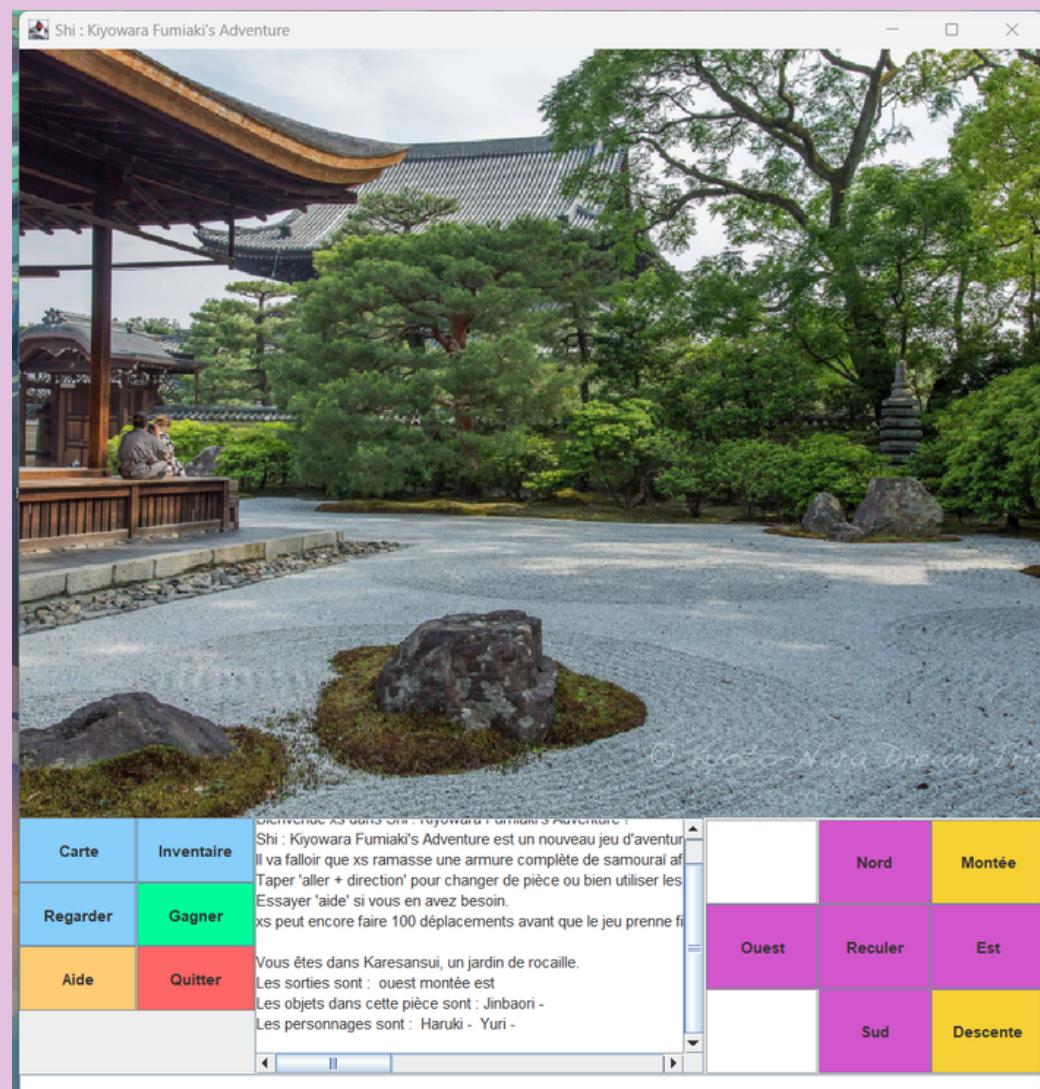
```
465     /**
466      * Procédure qui permet d'afficher les messages de bienvenue au début du jeu
467      */
468  private void printWelcome()
469  {
470      this.aGui.print( "\n" );
471      this.aGui.println( "Bienvenue " + this.aPlayer.getPseudo() + " dans Shi :
Kiyowara Fumiaki's Adventure !" );
472      this.aGui.println( "Shi : Kiyowara Fumiaki's Adventure est un nouveau jeu
d'aventure dans lequel " + this.aPlayer.getPseudo() + " va devoir récupérer Shi,
le katana légendaire de sa descendance." );
473      this.aGui.println( "Il va falloir que " + this.aPlayer.getPseudo() + " ramasse
une armure complète de samouraï afin de remporter le combat contre celui qui a
volé son katana." );
474      this.aGui.println( "Taper 'aller + direction' pour changer de pièce ou bien
utiliser les flèches directionnelles." );
475      this.aGui.println( "Essayer 'aide' si vous en avez besoin." );
476      this.aGui.println( this.aPlayer.getPseudo() + " peut encore faire "+
this.aPlayer.getMaxDeplacements() + " déplacements avant que le jeu prenne fin." );
477      this.aGui.print( "\n" );
478      this.printLocationInfo();
479  }//printWelcome()
```

Ainsi, on ajoute également dans la procédure printAide() l'avancée du compteur de déplacements afin que le joueur puisse connaître en temps réel le nombre de déplacements restants :

```
481  /**
482   * Procédure qui affiche les commandes d'aides possibles
483   */
484  private void printAide()
485  {
486      this.aGui.println( "\n" + this.aPlayer.getPseudo() + " se balade dans le temple
de Shizue. \n\n Les commandes disponibles sont :\n" );
487      this.aGui.println( this.aParser.getCommandString() );
488      this.aGui.println( "Taper 'aller + direction' pour changer de pièce." );
489      this.aGui.println( "Taper 'quitter' si " + this.aPlayer.getPseudo() + " veut
arrêter son aventure." );
490      this.aGui.println( "Taper 'regarder' si " + this.aPlayer.getPseudo() + " veut
regarder ce qu'il se passe dans les environs..." );
491      this.aGui.println( "Taper 'regarder' + le nom de l'objet dans l'inventaire de "
+ this.aPlayer.getPseudo() + " pour obtenir sa description." );
492      this.aGui.println( "Taper 'ingérer + nom de l'objet' si " +
this.aPlayer.getPseudo() + " veut manger." );
493      this.aGui.println( "Taper 'reculer' si " + this.aPlayer.getPseudo() + " veut
retourner dans la pièce précédente." );
494      this.aGui.println( "Taper 'prendre + nom de l'objet' si " +
this.aPlayer.getPseudo() + " veut ramasser un objet de la pièce." );
495      this.aGui.println( "Taper 'jeter + nom de l'objet' si " +
this.aPlayer.getPseudo() + " veut jeter un objet de son inventaire." );
496      this.aGui.println( "Taper 'inventaire' si " + this.aPlayer.getPseudo() + " veut
voir le contenu de celui-ci." );
497      this.aGui.println( this.aPlayer.getPseudo() + " peut encore faire "+
this.aPlayer.getMaxDeplacements() + " déplacements avant que le jeu prenne fin." );
498  }//printAide()
```

Exercice 7.42.2 :

J'ai profité de cet exercice pour rajouter des boutons pour les commandes que nous avons créées depuis l'exercice où l'on a crée les boutons. J'ai décidé cependant de ne pas rajouter de boutons pour les commandes comme manger qui nécessitent un second mot. Les boutons ont été réorganisés proprement en séparant les boutons d'informations des boutons directionnels et les couleurs ont été modifiées :



Pour arriver à une telle disposition dans le Panel Est, j'ai rajouter deux boutons sans texte et non cliquables, afin de garder un lay-out en forme de croix :

```
*  
512     //Création du bouton Vide 2  
513         this.aButtonVide2 = new JButton("");  
514         this.aButtonVide2.setBackground(Color.white);  
515         this.aPanelEast.add(this.aButtonVide2);  
516         this.aButtonVide2.setEnabled(false);
```

J'ai également donné la possibilité au joueur de se déplacer à l'aide des flèches directionnelles. Pour ce faire, j'ai du importer les classes suivantes qui me seront utiles dans le code :

```
519     import java.util.Collections;
520 import javax.swing.KeyStroke;
521 import java.awt.KeyboardFocusManager;
522 import java.awt.event.KeyEvent;
523 import java.awt.event.KeyListener;
```

On doit changer également la signature de la classe pour implémenter une nouvelle classe :

```
525 public class UserInterface implements ActionListener, KeyListener
526 {
```

On doit donc ensuite modifier le constructeur :

```
527 /**
528      * Constructeur d'objets de classe UserInterface
529      * Construire une interface utilisateur.
530      * En tant que paramètre, un moteur de jeu (un objet qui traite et exécute les
531      * commandes du jeu) est nécessaire.
532      *
533      * @param pGameEngine L'objet GameEngine implementant la logique du jeu.
534      */
535 public UserInterface(final GameEngine pGameEngine)
536 {
537     // initialisation des variables d'instance
538     this.aEngine = pGameEngine;
539     this.createGUI();
540     this.aMyFrame.setFocusable(true);
541     this.aMyFrame.requestFocusInWindow();
542     this.aMyFrame.addKeyListener(this);

543     this.aMyFrame.setFocusTraversalKeys(KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS,
544     Collections.singleton(KeyStroke.getKeyStroke(KeyEvent.VK_TAB, 0)));
545 }
```

Les trois dernières instructions affectent le focus et le contrôle du clavier à la fenêtre principale aMyFrame de l'interface utilisateur. La méthode setFocusable(true) permet à la fenêtre de recevoir les événements de clavier. requestFocusInWindow() demande à la fenêtre de prendre le focus, c'est-à-dire d'être l'élément actif dans la fenêtre. Enfin, addKeyListener(this) ajoute un écouteur d'événements de clavier à la fenêtre. La dernière instruction permet de spécifier que lorsqu'un utilisateur appuie sur la touche Tabulation, le focus doit passer au prochain élément du composant de l'interface utilisateur, grâce à la méthode setFocusTraversalKeys().

On doit à présent créer une méthode keyPressed() qui permettra de réagir lorsqu'une touche sur le clavier est pressée. La méthode prend en paramètre un objet de type KeyEvent, qui représente l'événement de frappe de touche. La méthode commence par extraire le code de touche en utilisant la méthode getKeyCode() de l'objet KeyEvent. Ensuite, un bloc switch est utilisé pour traiter les différentes touches en fonction de leur code.

```
545     /**
546      * Réagit lorsqu'une touche du clavier est pressée.
547      * Si la touche pressée est une touche fléchée, une touche de page ou de
548      * suppression, la méthode interprète la commande correspondante en invoquant la
549      * méthode interpretCommand() de l'objet aEngine.
550      */
551     @Override
552     public void keyPressed(KeyEvent e) {
553         int keyCode = e.getKeyCode();
554         switch(keyCode) {
555             case KeyEvent.VK_UP :
556                 this.aEngine.interpretCommand("aller nord");
557                 break;
558             case KeyEvent.VK_DOWN:
559                 this.aEngine.interpretCommand("aller sud");
560                 break;
561             case KeyEvent.VK_LEFT:
562                 this.aEngine.interpretCommand("aller est");
563                 break;
564             case KeyEvent.VK_RIGHT:
565                 this.aEngine.interpretCommand("aller ouest");
566                 break;
567             case KeyEvent.VK_PAGE_UP :
568                 this.aEngine.interpretCommand("aller montée");
569                 break;
570             case KeyEvent.VK_PAGE_DOWN :
571                 this.aEngine.interpretCommand("aller descente");
572                 break;
573             case KeyEvent.VK_BACK_SPACE :
574                 this.aEngine.interpretCommand("reculer");
575                 break;
576             default:
577         }
578     } //keyPressed()
```

Pour que cela fonctionne correctement, on doit implémenter deux méthodes nécessaires car elles font partie de l'interface KeyListener. En effet, si on les retire, on obtient une erreur de compilation car il manque ces deux méthodes requises par l'interface. La première est la méthode keyReleased, qui est appelée lorsque la touche d'un clavier est relâchée. La deuxième est la méthode keyTyped, qui est appelée lorsque l'utilisateur tape une touche. Même si ces deux méthodes ne font rien, elles sont toujours nécessaires car l'interface exige qu'elles soient implémentées.

```
580  /**
581   * Ne fait rien en réponse à la libération d'une touche de clavier.
582   *
583   * @param e L'événement de libération de touche de clavier.
584   */
585  @Override
586  public void keyReleased(KeyEvent e) {
587      // Do nothing
588  } //keyReleased()
589
590  /**
591   * Cette méthode ne fait rien car elle est déclenchée lorsqu'une touche est
592   * tapée et ne nécessite aucune action spécifique.
593   *
594   * @param e l'événement KeyEvent déclenché par la frappe de touche.
595   */
596  @Override
597  public void keyTyped(KeyEvent e) {
598      // Do nothing
599  } //keyTyped()
```

Exercice 7.43 :

L'objectif de cet exercice est d'implémenter une Trap Door. Il s'agit d'une porte où le joueur ne la traversera qu'une seule fois. Il sera donc impossible de retourner en arrière une fois la porte franchie. On doit donc enlever la possibilité au joueur de réemprunter la porte. Pour ce faire, il suffit de supprimer la sortie dans le sens inverse. Cependant, la pièce reste accessible avec la commande reculer car celle-ci utilise uniquement une pile de pièces sans conditions. Pour résoudre ce problème, il va falloir premièrement ajouter dans la classe Room une fonction booléenne que l'on appellera isExit() et qui permettra de vérifier si la pièce vers laquelle on souhaite se rendre dispose bien de cette pièce en sortie. On utilise donc la méthode containsValue() fournie par la classe HashMap qui nous permet de savoir si la HashMap contient bien un élément.

```
600  /**
601   * Fonction booléenne qui vérifie si la pièce est accessible
602   *
603   * @param pRoom Pièce dont on vérifie l'accessibilité
604   * @return Vrai si la pièce est une sortie possible, Faux dans le cas
605   * contraire
606   */
607  public boolean isExit (final Room pRoom) {
608      return pRoom.aExits.containsValue(this);
609  } //isExit()
```

Pour définir la Trap Door, j'ai décidé dans mon jeu de choisir la pièce Kuri. Pour que le joueur ne puisse pas retourner en arrière, on supprime la sortie allant à l'est et ainsi le joueur ne peut plus accéder à la pièce précédente.

Dans la classe Player, avant de modifier la méthode back(), on doit ajouter une fonction booléenne qui va permettre de savoir si le joueur a emprunté une TrapDoor ou non. Cette méthode utilisera isExit() pour savoir s'il y a bien une sortie. On l'appelle passTrapDoor(). On utilise également la méthode peek() qui permet de retourner l'élément en haut de la pile.

```
610     /**
611      * Fonction booléenne qui permet de savoir si le joueur a emprunté une Trap
612      * Door
613      * @return Vrai si la dernière pièce ajoutée à la pile des anciennes pièces
614      * est une sortie de la pièce courante, Faux dans le cas contraire
615      */
616     public boolean passTrapDoor(){
617         if(!this.aPreviousRoom.peek().isExit(this.aCurrentRoom)){
618             return true;
619         }
620         return false;
621     }//passTrapDoor()
```

Pour finir ces modifications, il nous reste plus qu'à changer la méthode back() pour renvoyer un booléen plutôt que prendre en paramètre une commande. De plus, la méthode vérifie maintenant si le joueur a emprunté une trapdoor avant de revenir en arrière, et si tel est le cas, elle renvoie false et affiche un message. Si le joueur n'a pas emprunté une trapdoor, la pièce précédente est définie comme la nouvelle pièce actuelle et la méthode renvoie true.

```
622     /**
623      * Permet de revenir en arrière, dans la pièce précédente
624      *
625      * @param pCommand Commande de l'utilisateur
626      */
627     public boolean back()
628     {
629         if(this.aPreviousRoom.isEmpty())
630         {
631             this.aEngine.getGUI().println("Il n'y a pas de pièces précédentes.");
632             return false;//Arrêt prématuré
633         }//if
634         if(passTrapDoor()){
635             this.aEngine.getGUI().println(this.aPseudo + " ne peut pas revenir en
636             arrière après avoir emprunté une Trap Door.");
637             return false;
638         }
639         this.aCurrentRoom = this.aPreviousRoom.pop();//Change la pièce actuelle
640         par la pièce suivante
641         this.aMaxDeplacements -= 1;
642         this.aEngine.timerEnd();
643         return true;
644     }//back()
```

Exercice 7.43.1 :

Les deux javadocs ont été régénérées.

Exercice 7.44 :

L'objectif de cet exercice est d'ajouter une nouvelle sorte d'Item, un Beamer. Ce beamer sera un téléporteur qui pourra être ramassé dans une première pièce, puis être chargé dans une deuxième pièce et enfin être déchargé dans une troisième pièce. Le déchargement du Beamer a pour but de ramener le joueur dans la pièce où le beamer a été chargé. Beamer étant une sorte d'item, on a besoin de créer une nouvelle classe Beamer qui hérite de la classe Item.

Dans cette nouvelle classe, on ajoute deux attributs : alsCharged de type booléen qui indique vrai si le Beamer est chargé et faux dans le cas contraire et aChargedRoom de type Room qui correspond à la pièce dans laquelle le Beamer est chargé.

```
645     public class Beamer extends Item
646     // variables d'instance
647     private Room aChargedRoom; //Pièce dans laquelle le beamer a été chargé
648     private boolean aIsCharged;
```

On initialise les attributs dans le constructeur de la classe. Beamer étant héritée d'Item on peut donc récupérer les lignes de code du constructeur d'Item via l'utilisation de super(paramètres) :

```
651     /**
652      * Constructeur naturel d'objets de classe Beamer
653      *
654      * @param pName Nom du Beamer
655      * @param pDescription Description du Beamer
656      * @param pWeight Poids du Beamer
657      */
658     public Beamer(final String pName, final String pDescription, final int
       pWeight)
659     {
660         // initialisation des variables d'instance
661         super(pName, pDescription, pWeight);
662         this.aChargedRoom = null;
663         this.aIsCharged = false;
664     }//Beamer()
```

On rajoute un accesseur getChargedRoom() permettant de récupérer la pièce chargée dans le beamer,

```
666     /**
667      * Accesseur de la dernière pièce où le Beamer a été chargé.
668      *
669      * @return La dernière pièce où le Beamer a été chargé.
670      */
671     public Room getChargedRoom(){
672         return this.aChargedRoom;
673     }//getChargedRoom()
```

On rajoute une fonction booléenne qui vérifie si le Beamer est chargé.

```
675     /**
676      * Fonction booléenne qui vérifie si le beamer est chargé
677      *
678      * @return Vrai si une pièce est stockée dans le beamer, Faux dans le cas
       contraire
679      */
680     public boolean isCharged(){
681         return this.aChargedRoom != null;
682     }//isCharged()
```

On crée ensuite 2 méthodes : charged() et discharged(). La première méthode permet de charger le Beamer dans la pièce spécifiée tandis que la deuxième permet de décharger le Beamer.

```
684     /**
685      * Charge le Beamer dans la pièce spécifiée en stockant la pièce dans laquelle
686      * il est chargé.
687      *
688      * @param pChargedRoom La pièce dans laquelle le Beamer est chargé
689      */
690     public void charged(final Room pChargedRoom) {
691         this.aIsCharged = true;
692         this.aChargedRoom = pChargedRoom;
693     }//charged()
694
695     /**
696      * Décharge le Beamer, le désactivant pour une prochaine utilisation.
697      */
698     public void discharged(){
699         this.aIsCharged = false;
700     }//discharged()
```

On n'oublie pas d'ajouter les deux commandes en tant que commandes valides ainsi que le cas des commandes dans interpretCommand() dans la classe GameEngine.

```
701         private static final String[] aValidCommands =
702             {"aller", "aide", "quitter", "regarder", "ingérer", "reculer",
703              "test", "prendre", "jeter", "inventaire", "charger", "décharger"};
```

```
774     case "charger" :
775         this.aPlayer.charge(vCommand);
776         break;
777     case "décharger" :
778         this.aPlayer.fire(vCommand);
779         break;
780
```

On va maintenant s'intéresser à la création des deux méthodes charge() et fire() qui vont permettre au joueur de charger et de décharger respectivement le Beamer. Ces méthodes concernent le joueur et sont donc placées dans la classe Player. On utilise le même raisonnement que dans les méthodes take() et drop().

La première partie de la méthode vérifie si l'utilisateur a entré un deuxième mot dans sa commande. Ensuite, la méthode vérifie si l'objet entré est un objet Beamer. Si l'objet est un Beamer, la méthode vérifie si le joueur possède l'objet dans son inventaire. Ensuite, la méthode vérifie si le Beamer est déjà chargé. Enfin, si le Beamer n'est pas déjà chargé et que le joueur possède l'objet, la méthode appelle la méthode "charged" du Beamer pour le charger et affiche un message indiquant que le Beamer a été chargé.

```

704 /**
705      * Commande qui permet de charger le Beamer
706      *
707      * @param pCommand Une commande entrée par l'utilisateur
708      */
709     public void charge(final Command pCommand) {
710         this.aEngine.timerEnd();
711         if(!pCommand.hasSecondWord()) {
712             this.aEngine.getGUI().println("Que voulez-vous charger ?");
713             return;
714         }//if
715
716         String vSecondWord = pCommand.getSecondWord();
717         if(!vSecondWord.equals("Shukkō.ki")){
718             this.aEngine.getGUI().println("Désolé... Cet objet ne peut pas être
719             chargé...");
720             return;
721         }//if
722
723         Item vItem = this.aItems.getItem(vSecondWord);
724         Beamer vBeamer =(Beamer)vItem;
725         if(vItem == null){
726             this.aEngine.getGUI().println(this.aPseudo + " ne possède pas le
727             beamer dans son inventaire.");
728             return;
729         }//if
730         if(vBeamer.isCharged()){
731             this.aEngine.getGUI().println(this.aPseudo + " a déjà chargé le beamer
732             !");
733             return;
734         }//charge()

```

```

736 /**
737      * Commande qui décharge le beamer
738      *
739      * @param pCommand Une commande entrée par l'utilisateur
740      */
741     public void fire (final Command pCommand){
742         if(!pCommand.hasSecondWord()){
743             this.aEngine.getGUI().println("Que voulez-vous décharger ?");
744             return;
745         }//if
746
747         String vSecondWord = pCommand.getSecondWord();
748         if(!vSecondWord.equals("Shukkō.ki")){
749             this.aEngine.getGUI().println("Désolé... Cet objet ne peut pas être
déchargé...");
750             return;
751         }
752
753         Item vItem = this.aItems.getItem(vSecondWord);
754         Beamer vBeamer = (Beamer)vItem;
755         if(vItem == null){
756             this.aEngine.getGUI().println(this.aPseudo + " ne possède pas le
beamer dans son inventaire.");
757             return;
758         }//if
759         if(!vBeamer.isCharged()){
760             this.aEngine.getGUI().println(this.aPseudo + " doit d'abord charger le
beamer...");
761             return;
762         }//if
763         vBeamer.discharged();
764         this.aEngine.getGUI().println(this.aPseudo + " a déchargé le beamer !");
765         this.aPreviousRoom.push(this.aCurrentRoom); //Ajoute la pièce actuelle aux
anciennes pièces
766         this.aCurrentRoom = vBeamer.getChargedRoom(); //Remplace la pièce actuelle
par celle où le beamer a été chargé
767         this.aItems.removeItem(vBeamer.getName(), vItem); //Retire le beamer de
l'inventaire
768         this.aWeight -= vBeamer.getWeight(); //Retire le poids du beamer du poids
de l'inventaire
769         this.aMaxDeplacements --; //Retire un déplacement
770         this.aEngine.printLocationInfo();
771         this.aEngine.timerEnd();
772     }//fire()

```

Si l'utilisateur entre une commande sans second mot, le programme renvoie un message demandant ce qu'il faut décharger. Ensuite, le programme vérifie si le deuxième mot correspond au nom du Beamer (Shukkô.ki). Ensuite, le programme vérifie si l'utilisateur possède bien le Beamer dans son inventaire. Ensuite, le programme vérifie si le Beamer est chargé. Si ce n'est pas le cas, le programme renvoie un message demandant à l'utilisateur de charger le Beamer. Si le Beamer est chargé, le programme décharge le Beamer et stocke la pièce actuelle dans la liste des pièces précédentes. Ensuite, la pièce actuelle est remplacée par la pièce où le Beamer a été chargé. Le Beamer est également retiré de l'inventaire de l'utilisateur et le poids du Beamer est soustrait du poids de l'inventaire. Enfin, le nombre maximal de déplacements est décrémenté de 1 et les informations sur l'emplacement sont affichées.

Pour finir, on va créer un objet de type Beamer et l'ajouter à une pièce comme on l'a fait pour les items.

```
781             Beamer vBeamer = new Beamer("Shukkô.ki", "Il s'agit d'un  
782                 objet qui permet de se téléporter dans une pièce précédemment chargée.",300);  
783         vKairo.getRoomItems().addItem("Shukkô.ki", vBeamer);
```

Exercice 7.45 :

L'objectif de cet exercice est de créer une porte qui se ferme et s'ouvre uniquement en la présence d'un objet particulier dans l'inventaire. Une clé doit être prise par le joueur avant de pouvoir ouvrir ou fermer une porte. Elle fonctionne des 2 côtés de la porte. Plusieurs clés différentes doivent pouvoir être gérées, et s'il y a plusieurs portes fermées à clé, il faut une clé différente pour ouvrir chacune d'entre-elles. Tout d'abord, on crée donc une classe Door. La classe Door est une classe qui permet de créer des portes fermées et ouvrables avec une clé. On crée trois attributs. Le premier attribut aState représente l'état de la porte (verrouillée ou déverrouillée). Elle est de type boolean, où la valeur true signifie que la porte est verrouillée et la valeur false signifie qu'elle est déverrouillée. Le deuxième attribut est aKey qui représente la clé associée à la porte. Elle est de type "Item". On initialise dans le constructeur de la classe l'état de la porte (variable "aState") à false, ce qui signifie que la porte est déverrouillée.)

```

232 /**
233 La classe Door permet de créer des portes fermées et ouvrables avec une clé.
234 La classe Door représente une porte dans le jeu.
235 Une porte peut être verrouillée ou déverrouillée. Elle peut être associée à une
236 clé.
237 La classe contient des méthodes pour récupérer la clé associée à la porte,
238 verrouiller la porte avec une clé donnée, et vérifier si la porte est verrouillée.
239
240 */
241 public class Door
242 {
243     // variables d'instance
244     private boolean aState;// État de la porte (verrouillée ou déverrouillée)
245     private Item aKey; // Clé associée à la porte
246     private Player aPlayer;
247
248 /**
249 Constructeur par défaut de la classe Door.
250 Initialise l'état de la porte à false, ce qui signifie qu'elle est déverrouillée.
251 */
252 public Door(){
253     this.aState = false;
254 } //Door()

```

On crée l'accesseur getKey() pour l'attribut aKey. Cette méthode renvoie la clé associée à la porte. Elle retourne l'objet de type "Item" représentant la clé.

```

256 /**
257
258 Renvoie la clé associée à cette porte.
259 @return La clé associée à cette porte.
260 */
261 public Item getKey() {
262     return this.aKey;
263 } //getKey()

```

On ajoute une méthode lock() qui permet de verrouiller la porte avec une clé donnée. Elle prend en paramètre un objet "Item" qui représente la clé utilisée pour verrouiller la porte. Lorsque cette méthode est appelée, elle met à jour l'état de la porte à true (verrouillée) et associe la clé donnée à la porte.

```

265 /**
266
267 Verrouille la porte avec la clé donnée.
268 @param pKey la clé utilisée pour verrouiller la porte
269 */
270 public void lock(final Item pKey) {
271     this.aState = true;
272     this.aKey = pKey;
273 } // lock()

```

La dernière méthode de la classe Door est isLocked() qui vérifie si la porte est verrouillée. Elle renvoie un booléen qui indique si la porte est verrouillée (true) ou non (false), en se basant sur la valeur de la variable "aState".

```

275 /**
276 Vérifie si la porte est verrouillée.
277 @return un booléen indiquant si la porte est verrouillée (true) ou non (false)
278 */
279 public boolean isLocked() {
280     return this.aState;
281 } // isLocked()

```

Pour que l'implémentation d'une porte fermée fonctionne, on a besoin de modifier la méthode goRoom() dans la classe Player afin de vérifier si la prochaine pièce est fermée et si elle l'est, est ce que le joueur possède la clé. La partie modifiée de la méthode goRoom() commence par la condition if (vNextDoor != null && vNextDoor.isLocked()) qui vérifie si la porte suivante existe et si elle est verrouillée. Dans le cas d'une porte verrouillée, on vérifie si le joueur possède la clé nécessaire pour ouvrir la porte, en utilisant la méthode hasItem() qui vérifie si le joueur a l'objet correspondant à la clé dans son inventaire. Si le joueur possède la clé, le joueur peut accéder à la prochaine pièce dans la direction spécifiée, la pièce actuelle est ajoutée à la pile et la pièce courante est mise à jour avec la pièce suivante. On n'oublie pas de réduire de 1 le nombre maximal de déplacements. Dans le cas où le joueur ne possède pas la clé, un message indiquant que le joueur n'a pas la clé nécessaire est affiché. Il reste le cas où la porte n'est pas verrouillée, on doit effectuer les mêmes étapes que celles décrites précédemment pour se déplacer vers la pièce suivante.

```

202 /**
203 Procédure permettant de se déplacer de pièces en pièces dans la direction
spécifiée.
204
205 @param pDirectionSouhaite Direction dans laquelle le joueur souhaite se rendre
206 */
207 public void goRoom(final Command pDirectionSouhaite){
208     if(!pDirectionSouhaite.hasSecondWord()){
209         this.aEngine.getGUI().println("Où voulez-vous aller ?") ;
210         return;//Arrête la fonction prématurément car l'on n'a pas besoin de
changer de pièce
211     }//if
212     String vDirection = pDirectionSouhaite.getSecondWord();
213     Door vNextDoor = this.aCurrentRoom.getDoor(vDirection);
214     if(vNextDoor == null && !(this.aCurrentRoom instanceof TransporterRoom)) {
215         this.aEngine.getGUI().println("Il n'y a pas de porte !");
216         return;
217     if (vNextDoor != null && vNextDoor.isLocked()) {
218         if (this.aItems.hasItem(vNextDoor.getKey().getName())) {
219             Room vNextRoom = this.aCurrentRoom.getExit(vDirection);
220             this.aPreviousRoom.push(this.aCurrentRoom);
221             this.aCurrentRoom = vNextRoom;
222             this.aMaxDeplacements -= 1;
223             this.aEngine.timerEnd();} else {
224                 this.aEngine.getGUI().println(this.aPseudo + " n'a pas de quoi ouvrir la
porte dans son inventaire...");} } else {
225         Room vNextRoom = this.aCurrentRoom.getExit(vDirection);
226         this.aPreviousRoom.push(this.aCurrentRoom);
227         this.aCurrentRoom = vNextRoom;
228         this.aMaxDeplacements -= 1;
229         this.aEngine.timerEnd();}
230 } //goRoom()

```

Des modifications dans la classe Room sont ainsi également nécessaire. On ajoute un attribut aDoors de type HashMap<String, Door>. aDoors est une HashMap, où la clé est de type String et représente la direction de la porte, et la valeur est de type Door et représente l'objet Door associé à cette direction.

```
784 private HashMap<String, Door> aDoors; // Stocke les portes de la pièce  
785 this.aDoors = new HashMap<String, Door>();
```

En effet, on doit modifier la méthode setExit() qui est utilisée pour définir une sortie de la pièce dans la direction spécifiée. A présent, cette méthode crée également une porte dans la direction spécifiée en utilisant la classe Door et la stocke dans un autre HashMap appelé aDoors, où la clé est la direction et la valeur est l'objet Door associé à la porte.

```
787 /**  
788 Définit une sortie de cette pièce dans la direction indiquée.  
789 La sortie est représentée par la pièce voisine dans la direction donnée.  
790 Une porte est également créée dans cette direction.  
791  
792 @param pDirection Indique la direction de la sortie.  
793 @param pVoisine Indique la pièce dans la direction donnée.  
794 */  
795 public void setExit(final String pDirection, final Room pVoisine){  
796     this.aExits.put(pDirection, pVoisine);  
797     this.aDoors.put(pDirection, new Door());  
798 } //setExit()
```

On ajoute pour finir deux méthodes. La première étant lockDoor() qui permet de verrouiller une porte dans la direction donnée avec un objet spécifié, en utilisant la HashMap aDoors pour obtenir l'objet Door associé à la porte dans la direction donnée. Elle appelle la méthode lock() pour verrouiller la porte avec l'objet spécifié.

```
800 /*  
801  
802 Verrouille une porte dans la direction donnée avec un certain objet.  
803  
804 @param pDirection La direction de la porte à verrouiller.  
805 @param pItem L'objet utilisé pour verrouiller la porte.  
806 */  
807 public void lockDoor(final String pDirection, final Item pItem) {  
808     this.aDoors.get(pDirection).lock(pItem);  
809 } //lockDoor()
```

La deuxième méthode est getDoor() qui retourne l'objet Door associé à la porte dans la direction spécifiée. Elle utilise également la HashMap aDoors.

```
811 /*  
812 Retourne l'objet Door associé à la porte dans la direction donnée.  
813  
814 @param pDirection La direction de la porte.  
815 @return L'objet Door associé à la porte.  
816 */  
817 public Door getDoor(final String pDirection) {  
818     return this.aDoors.get(pDirection);  
819 } //getDoor()
```

Pour finir, pour instancier les pièces qui possèderont une porte verrouillée, on doit modifier la classe GameEngine. On ajoute le même attribut que dans la classe Room et on l'initialise de la même manière.

```
784 private HashMap<String, Door> aDoors; // Stocke les portes de la pièce  
785 this.aDoors = new HashMap<String, Door>();
```

Dans la méthode createRooms(), j'ai décidé de créer une nouvelle pièce qui sera la porte avant la pièce verrouillée. Donc j'ai crée une nouvelle Room comme précédemment :

```
826 Room vGate = new Room("Porte","la porte devant Hokke Do .","./Images/Gate.png");
```

J'ai donc modifié les sorties pour ajouter la pièce :

```
827 //Sorties du Gate  
828         vGate.setExit("ouest", vHokke_do);  
829         vGate.setExit("est", vKaresansui);  
830 vKaresansui.setExit("ouest", vGate);  
831 //Sortie de la salle de Méditation  
832         vHokke_do.setExit("est",vGate);
```

Prenons pour exemple la première ligne de code ; 'on procèdera de la même manière pour le reste)

```
821 vGate.lockDoor("ouest", vClé);  
822         vHokke_do.lockDoor("est", vClé);  
823  
824         vKaresansui.lockDoor("est", vBadge);  
825         vChinjusha.lockDoor("ouest", vBadge);
```

La variable vGate est la nouvelle pièce de type Room créée. On appelle la méthode lockDoor() sur l'objet vGate pour verrouiller la porte dans la direction "ouest" à l'aide de la clé vClé.

Exercice 7.45.1 :

Les fichiers test ont été mis à jour.

Exercice 7.45.2 :

J'ai regénéré les 2 javadoc (comme au 7.26.1).

Exercice 7.46 :

L'objectif de cet exercice est de créer une pièce TransporterRoom qui téléporte le joueur dans une autre pièce aléatoire en sortant de cette pièce. Pour ce faire, il faut créer tout d'abord une nouvelle classe RoomRandomizer qui importe les classes suivantes :

```
891 import java.util.ArrayList;  
892 import java.util.Random;  
893 import java.util.List;
```

Dans cette nouvelle classe, on crée deux attributs. Le premier attribut aRooms est de type HashMap qui permet de stocker la liste de pièce. Le deuxième attribut est aRandom de type aléatoire.

On initialise dans le constructeur ces deux attributs en créant un nouveau Random et une ArrayList de pièces.

```

894 /**
895
896 La classe RoomRandomizer crée une pièce aléatoire.
897 Cette classe permet de sélectionner une pièce aléatoire à partir d'une liste de
pièces.
898 *
899 @author HAKIM Justine
900 @version 21/05/2023
901 */
902 public class RoomRandomizer {
903     private List<Room> aRooms;
904     private Random aRandom;
905     private String aForcedRoom;
906
907 /**
908 Constructeur de la classe RoomRandomizer.
909 Il crée un objet RoomRandomizer avec une liste vide de pièces.
910 */
911 public RoomRandomizer() {
912     this.aRandom = new Random();
913     this.aRooms = new ArrayList<Room>();
914 }//RoomRandomizer()

```

On ajoute une procédure addRoom() qui permet d'ajouter des pièces dans la liste des pièces.

```

916 /**
917 Ajoute une pièce à la liste de pièces du RoomRandomizer.
918 @param pRoom La pièce à ajouter.
919 */
920 public void addRoom(final Room pRoom){
921     this.aRooms.add(pRoom);
922 }//addRoom()

```

On ajoute également la fonction findRandomRoom() qui tire une pièce aléatoire dans la liste des pièces vers lesquelles on peut être téléportées à partir de son index dans la liste.

```

924 /**
925 Recherche une pièce aléatoire.
926 @return Une pièce aléatoire sélectionnée dans la liste de pièces.
927 */
928 public Room findRandomRoom()
941     int vNb = this.aRandom.nextInt(this.aRooms.size());
942     return this.aRooms.get(vNb);
931 }
944 }//findRandomRoom()

```

On a également besoin de créer une seconde classe TransporterRoom qui sera une sorte de pièce, donc la classe a la signature suivante :

```

834 /**
835 La classe TransporterRoom représente une pièce de type "Transporteur".
836 Une pièce de type Transporteur permet de se téléporter aléatoirement vers une
autre pièce.
837 Elle hérite de la classe Room.
838 *
839 @author HAKIM Justine
840 @version 21/05/2023
841 */
842 public class TransporterRoom extends Room

```

Dans cette nouvelle classe, on ajoute un attribut de type RoomRandomizer qui sera initialisé dans le constructeur de la classe. Dans celui-ci, il faut au préalable utiliser super(paramètres) pour initialiser tous les attributs car la classe est une sorte de Room.

```
344     // variables d'instance
345     private RoomRandomizer aRoomRandomizer;
346
347     /**
348 Constructeur d'objets de classe TransporterRoom
349 Il crée un objet TransporterRoom avec un nom, une description et le nom de l'image
350 associée.
351 * @param pName Le nom de la pièce.
352 * @param pDescription La description de la pièce.
353 * @param pImageName Le nom de l'image associée à la pièce.
354 */
354 public TransporterRoom(final String pName, final String pDescription, final String
355 pImageName)
355 {
356     // initialisation des variables d'instance
357     super(pName, pDescription, pImageName);
358     this.aRoomRandomizer = new RoomRandomizer();
359 } //TransporterRoom()
```

On crée l'accesseur qui retourne l'attribut :

```
884 /**
885 Accesseur pour l'objet RoomRandomizer de la pièce Transporter.*/
886 @return L'objet RoomRandomizer associé à la pièce Transporter.
887 */
888 public RoomRandomizer getRoomRandomizer(){
889     return this.aRoomRandomizer;}//getRoomRandomizer()
```

On doit redéfinir la fonction getExit() dans la classe TransporterRoom puisqu'elle est définie normalement dans la classe Room. Cette méthode appelle la méthode getRandomRoom, indépendamment de la direction de la sortie, que l'on crée au préalable. Cette méthode retourne la méthode findRandomRoom().

```
861 /**
862
863 Redéfinition de la méthode getExit de la classe mère Room.
864 Cette méthode renvoie une pièce aléatoire à la place de la pièce de sortie
865 normale.
866 * @param pDirection La direction de sortie de la pièce.
867 * @return Une pièce aléatoire sélectionnée par le RoomRandomizer.
868 */
869 @Override
870 public Room getExit(final String pDirection)
871 {
872     return getRandomRoom();
873 } //getExit()
```

```
875 /**
876
877 Renvoie une pièce aléatoire.*/
878 @return Une pièce aléatoire sélectionnée par le RoomRandomizer.
879 */
880 public Room getRandomRoom(){
881     return this.aRoomRandomizer.findRandomRoom();
882 } //getRandomRoom()
```

Pour finir les modifications de cet exercice, on doit dans la classe GameEngine, dans la méthode createRooms(), créer une nouvelle TransporterRoom et ajouter les pièces dans lesquelles le joueur peut être téléporté. Comme précédemment, puisque l'on ajoute une nouvelle pièce on définit de nouveau les sorties possibles de celle-ci.

```

963 TransporterRoom vTransporterRoom = new TransporterRoom("TransporterRoom", "une
pièce qui vous téléporte aléatoirement dans une autre lorsque vous sortez.",
"./Images/transporter_room.png");
964 vTransporterRoom.getRoomRandomizer().addRoom(vKairo);
965     vTransporterRoom.getRoomRandomizer().addRoom(vKaresansui);
966     vTransporterRoom.getRoomRandomizer().addRoom(vGate);
967     vTransporterRoom.getRoomRandomizer().addRoom(vKyozo);
968 vHojo.setExit("ouest", vTransporterRoom);
969 vHatto.setExit("est", vTransporterRoom);

```

Exercice 7.46.1 :

L'objectif de cet exercice est de créer une nouvelle commande alea qui force l'aléatoire pour que lors de l'exécution des fichiers test, il soit possible de les continuer même après le passage dans la TransporterRoom. On modifie tout d'abord la classe RoomRandomizer en créant un nouvel attribut aForcedRoom.

```

905     private String aForcedRoom;

```

On doit ensuite créer une fonction setForcedRoom() qui prend en paramètre une chaîne de caractère. Si la longueur de la String prise en paramètre est égal à 0 alors l'attribut est égal à une chaîne de caractère non vide. Pour toutes les pièces appartenant à la liste des pièces, dans le cas où le paramètre est égal au nom de la pièce alors l'attribut aForcedRoom est égal au paramètre et retourne une chaîne de caractère. Sinon, une chaîne de caractère est renvoyée.

```

934 /**
935 Définit une pièce forcée.
936 @param pName Le nom de la pièce à forcer.
937 @return Un message indiquant si la pièce a été forcée ou si elle n'existe pas.
938 */
939 public String setForcedRoom(final String pName) {
940     if (pName.length() == 0) {
941         this.aForcedRoom = null;
942         return "Le tirage aléatoire a été réinitialisé. ";} else {
943             for(Room vRoom : this.aRooms){
944                 if(pName.equals(vRoom.getName())){
945                     this.aForcedRoom = pName;
946                     return "Le tirage aléatoire a été forcé sur " + pName;}}
947             return "Cette pièce n'existe pas ou vous ne pouvez pas vous y transportez.
948        ";}
949     } //setForcedRoom()

```

On a besoin de modifier dans la classe RoomRandomizer l'accesseur getRandomRoom. Si l'attribut aForcedRoom n'est pas égal à null : alors pour chaque pièce de la liste des pièces dans lesquelles on peut être téléporté et si le nom d'une de ces pièces est le même que la chaîne de caractère stockée dans aForcedRoom alors on retourne la pièce. Dans le cas où ce n'est pas le même nom alors on doit retourner null. Dans le cas où aForcedRoom est null alors on tire un index de manière aléatoire et on retourne la pièce correspondante à cet index.

```

924 /**
925 Recherche une pièce aléatoire.
926 @return Une pièce aléatoire sélectionnée dans la liste de pièces.
927 */
928 public Room findRandomRoom() {
929     if (this.aForcedRoom != null) {
930         for(Room vRoom : this.aRooms)
931         {
932             if(vRoom.getName().equals(this.aForcedRoom))
933             {
934                 return vRoom;
935             }
936         }
937         return null;
938     }
939     else
940     {
941         int vNb = this.aRandom.nextInt(this.aRooms.size());
942         return this.aRooms.get(vNb);
943     }
944 } //findRandomRoom()

```

Puisqu'il s'agit de la création d'une nouvelle commande, dans la classe GameEngine, dans la méthode interpretCommand(), on rajoute la commande alea de la manière suivante :

```

959 case "alea" :
960         this.alea(vCommand);
961         break;

```

On n'oublie pas d'ajouter la commande alea dans la liste des commandes valides :

```

963     private static final String[] aValidCommands =
964     {"aller","aide","quitter","regarder","ingérer","reculer",
965     "test","prendre","jeter","inventaire","charger","décharger","carte","alea","parler"
966     ",répondre","gagner"};

```

Pour finir les modifications de cet exercice, il ne nous reste plus qu'à créer la commande alea. On ajoute donc dans la classe GameEngine une procédure alea dans laquelle, on crée une pièce TransporterRoom qui sera la pièce de téléportation. Dans le cas où la commande entrée par l'utilisateur possède un deuxième mot alors on affiche un message qui correspond grâce à la méthode setForcedRoom(). Dans le cas contraire, on attribue à setForcedRoom la valeur null et on affiche un message.

```

965 /**
966 Cette méthode permet à l'utilisateur de spécifier si la salle choisie dans la
967 pièce de téléportation doit être sélectionnée de manière aléatoire ou si une salle
968 spécifique doit être forcée.
969 L'utilisateur peut fournir une commande avec un deuxième mot pour spécifier la
970 salle à forcer, ou ne pas fournir de deuxième mot pour réinitialiser la sélection
971 aléatoire.
972 @param pCommand La commande spécifiant l'action à effectuer. Si elle contient un
973 deuxième mot, celui-ci sera utilisé comme salle à forcer.
974 */
975 public void alea(final Command pCommand) {
976     TransporterRoom vTransporterRoom2 = (TransporterRoom)
977     this.aRooms.get("TransporterRoom");
978     if(pCommand.hasSecondWord()){
979
980         this.aGui.println(vTransporterRoom2.getRoomRandomizer().setForcedRoom(pCommand.get
981         SecondWord()));} else {
982             vTransporterRoom2.getRoomRandomizer().setForcedRoom("");
983             this.aGui.println("La sortie est de nouveau définie de manière
984             aléatoire");}
985     } //alea()

```

Exercice 7.48 :

L'objectif de cet exercice est de créer des personnages dans le jeu. Pour ce faire, tout d'abord, on ajoute une classe Character. Cette nouvelle classe représente un personnage dans un jeu. Elle contient les attributs suivants :

- aName qui représente le nom du personnage (de type String)
- aDialog qui représente le dialogue du personnage (de type String)
- aCorrectAnswer qui représente la réponse correcte à une énigme posée par le personnage (de type String)
- aReward qui représente la récompense attribuée par le personnage (de type Item)
- aNbError qui représente le nombre d'erreurs commises par le joueur (de type int)

```
194     /**
195      * La classe Character représente un personnage dans le jeu.
196      * Un personnage possède un nom, un dialogue, une réponse correcte à une énigme,
197      * une récompense.
198      *
199      * @author HAKIM Justine
200      * @version 21/05/2023
201      */
202  public class Character
203  {
204      // variables d'instance
205      private String aName ;
206      private String aDialog;
207      private String aCorrectAnswer;
208      private Item aReward;
209      private int aNbError;
```

On initialise les attributs dans le constructeur de la classe Character qui prend en paramètres le nom du personnage, son dialogue, la réponse correcte à l'éénigme posée par le personnage et la récompense attribuée.

```
210      /**
211       * Constructeur d'objets de classe Character
212       *
213       * @param pName Le nom du personnage.
214       * @param pDialog Le dialogue du personnage.
215       * @param pCorrectAnswer La réponse correcte à l'éénigme posée par le
216       * personnage.
217       * @param pReward La récompense attribuée par le personnage.
218       */
219   public Character(final String pName, final String pDialog, final String
220   pCorrectAnswer, final Item pReward)
221   {
222       // initialisation des variables d'instance
223       this.aName = pName;
224       this.aDialog = pDialog;
225       this.aCorrectAnswer = pCorrectAnswer;
226       this.aReward = pReward;
227       this.aNbError= 0;
228   } //Character()
```

Les méthodes créées dans cette classe sont :

- getNbError() qui est un accesseur qui retourne le nombre d'erreurs commises par le joueur.

```
228      /**
229       * Accesseur du nombre d'erreurs.
230       *
231       * @return Le nombre d'erreurs.
232       */
233   public int getNbError(){
234       return this.aNbError;
235   } //getNbError()
```

- upNbError() qui incrémente le nombre d'erreurs commises par le joueur.

```

237     /**
238      * Incrémenté le nombre d'erreurs.
239      */
240     public void upNbError(){
241         this.aNbError += 1;
242     }//upNbError()

```

- getName() qui est un accesseur qui retourne le nom du personnage

```

244     /**
245      * Accesseur du nom du personnage.
246      *
247      * @return Le nom du personnage.
248      */
249     public String getName(){
250         return this.aName;
251     }//getName()

```

- getDialogString() qui retourne le dialogue complet du personnage

```

253     /**
254      * Retourne le dialogue complet du personnage.
255      *
256      * @return Le dialogue du personnage.
257      */
258     public String getDialogString() {
259         return this.aName + ": " + this.aDialog ;
260     }//getDialogString()

```

- checkAnswer() qui vérifie si la réponse du joueur est correcte en comparant avec la réponse correcte du personnage. Elle retourne true si la réponse est correcte et false dans le cas contraire.

```

262     /**
263      * Vérifie si la réponse du joueur est correcte.
264      *
265      * @param pPlayerAnswer La réponse du joueur.
266      * @return true si la réponse est correcte, false sinon.
267      */
268     public boolean checkAnswer(final String pPlayerAnswer) {
269         return pPlayerAnswer.equals(this.aCorrectAnswer);
270     }//checkAnswer()

```

- getReward() qui est un accesseur qui retourne la récompense attribuée par le personnage.

```

272     /**
273      * Accesseur de la récompense attribuée par le personnage.
274      *
275      * @return La récompense.
276      */
277     public Item getReward() {
278         return this.aReward;
279     }//getReward()

```

Dans mon jeu, j'ai besoin d'un personnage spécial qui vérifie si le joueur à récolter tous les objets et s'il peut recevoir le titre de samouraï. J'ai donc du implémenter une classe Special Character. La classe SpecialCharacter représente un personnage spécial dans un jeu, et elle hérite de la classe "Character". On a donc la signature suivante :

```
282 /**
283 * La classe SpecialCharacter représente un personnage spécial dans le jeu.
284 * Elle hérite de la classe Character.
285 * Un personnage spécial possède un nom, un dialogue, une réponse correcte à une
286 * énigme et une récompense.
287 *
288 * @author HAKIM Justine
289 * @version 21/05/2023
290 */
291 public class SpecialCharacter extends Character
```

Le constructeur de la classe prend en paramètres le nom du personnage spécial, son dialogue, la réponse correcte à l'énigme posée par le personnage spécial, et la récompense attribuée. Il appelle le constructeur de la classe mère pour initialiser les variables d'instance héritées.

```
293 /**
294 * Constructeur d'objets de classe SpecialCharacter
295 *
296 * @param pName Le nom du personnage spécial.
297 * @param pDialog Le dialogue du personnage spécial.
298 * @param pCorrectAnswer La réponse correcte à l'énigme posée par le
299 * personnage spécial.
300 * @param pReward La récompense attribuée par le personnage spécial.
301 */
301 public SpecialCharacter(final String pName, final String pDialog, final String
302 pCorrectAnswer, final Item pReward)
302 {
303     // initialisation des variables d'instance
304     // Appel du constructeur de la classe mère (Character) pour initialiser
305     // les variables d'instance
306     super(pName, pDialog, pCorrectAnswer, pReward);
306 }//SpecialCharacter()
```

Cette nouvelle classe possède une unique méthode qui permet de vérifier si le joueur possède tous les items nécessaires dans son inventaire afin de pouvoir recevoir le titre de samouraï. Cette méthode s'appelle checkAllItems() et vérifie si tous les objets requis sont présents dans une liste d'objets. Elle prend en paramètre la liste d'objets, un type ItemList et retourne true si tous les objets requis sont présents dans la liste et false dans le cas contraire. Les objets requis sont vérifiés en utilisant les méthodes hasItem() de la classe ItemList() pour chaque objet requis.

```

308     /**
309      * Vérifie si tous les objets requis sont présents dans une liste d'objets.
310      *
311      * @param pItems La liste d'objets à vérifier.
312      * @return true si tous les objets requis sont présents, false sinon.
313      */
314     public boolean checkAllItems(final ItemList pItems){
315         // Vérifie si tous les objets requis sont présents dans la liste d'objets
316         return pItems.hasItem("Jinbaori") && pItems.hasItem("Kote") &&
317             pItems.hasItem("Hitatare.et.hakama") && pItems.hasItem("Mengu") &&
318             pItems.hasItem("Kutsu") && pItems.hasItem("Do") && pItems.hasItem("Sode") &&
319             pItems.hasItem("Suneate") && pItems.hasItem("Kusazuri.et.Haidate") &&
320             pItems.hasItem("Kabuto");
321     }//checkAllItems()

```

Afin que le joueur puisse interagir avec les personnages, on crée deux méthodes, qui sont deux commandes dans la classe Player. La première est talk(). La méthode talk() est une procédure qui permet d'engager une conversation avec un personnage spécifique. Tout d'abord, la méthode vérifie si la commande entrée par l'utilisateur a un deuxième mot. Ensuite, la méthode récupère le deuxième mot de la commande, qui est supposé être le nom du personnage avec lequel l'utilisateur veut parler. Si le personnage est inexistant (c'est-à-dire que vCharacter est null), un message indiquant que cette personne n'existe pas est affiché, et la conversation se termine.

Par exemple, si le personnage est Yakushi.Nyorai ou Statue.Sacrée, la méthode vérifie si le joueur a l'objet ¥50 dans son inventaire. Si oui, l'objet est récupéré et retiré de la liste d'objets du joueur. Le poids du joueur est également mis à jour en conséquence. En prévision afin d'éviter que le joueur reçoive la récompense à l'infini, la méthode vérifie si le joueur n'a pas déjà reçu la récompense du personnage. De plus, la méthode vérifie si le poids du joueur actuel lui permet de porter la récompense à l'aide de la méthode canTake(). Si ces deux conditions sont remplies, alors la récompense du personnage est ajoutée à l'inventaire du joueur, et le poids du joueur est mis à jour. Sinon, un message indiquant que le joueur a déjà reçu la récompense ou porte une charge trop lourde est affiché. Si le joueur n'a pas l'objet, un message lui demandant d'avoir une pièce pour vénérer la statue est affiché. Toutes les autres conditions fonctionnent sur le même modèle.

Si le personnage est une instance de la classe SpecialCharacter, la méthode vérifie si le joueur a tous les objets requis pour le personnage spécifique. Si les conditions sont remplies, la récompense du personnage est ajoutée à la liste d'objets du joueur, et le poids du joueur est mis à jour. Sinon, un message indiquant que le joueur est encore un samouraï junior est affiché.

Enfin, si aucun des cas précédents n'est vérifié, le dialogue du personnage est affiché. Le code pris en capture d'écran n'est qu'un échantillon du code complet.

```

17  /**
18   * Procédure permettant d'engager une conversation avec un personnage donné.
19   *
20  * @param pCommand La commande de conversation entrée par l'utilisateur
21  */
22 public void talk(final Command pCommand) {
23     if(!pCommand.hasSecondWord()){
24         this.aEngine.getGUI().println("Vous parlez à vous même ?");
25         return;
26     }//if
27     String vSecondWord = pCommand.getSecondWord();
28     Character vCharacter = this.aCurrentRoom.getCharacter(vSecondWord);
29     if(vCharacter == null){
30         this.aEngine.getGUI().println("Cette personne n'existe pas...");
31         return;
32     }//if
33     if(vCharacter.getName().equals("Yakushi.Nyorai") ||
vCharacter.getName().equals("Statue.Sacrée")){
34         if(this.aItems.hasItem("\u00a350")){
35             Item vYen50 = this.aItems.getItem("\u00a350");
36             this.aItems.removeItem("\u00a350", vYen50);
37             this.aWeight -= vYen50.getWeight();
38             if (!this.aItems.hasItem(vCharacter.getReward().getName()) &&
(canTake(vCharacter.getReward().getWeight()))) {// Vérification si le joueur n'a
pas déjà la récompense
39                 this.aItems.addItem(vCharacter.getReward().getName(),
vCharacter.getReward());
40                 this.aWeight += vCharacter.getReward().getWeight();
41                 this.aEngine.getGUI().println(this.aPseudo + " a vénétré la
statue. Pour son culte, " + this.aPseudo + " reçoit :" +
vCharacter.getReward().getName());
42                 return;
43             } else {
44                 this.aEngine.getGUI().println(this.aPseudo + " a déjà reçu la
récompense ou porte déjà une charge trop lourde.");
45                 return;
46             }
47         } else {
48             this.aEngine.getGUI().println(this.aPseudo + " a besoin d'une
pièce pour vénétrer la statue.");
49             return;
50         }
51     }

```

La deuxième commande est reply(), qui permet au joueur de proposer sa réponse à l'énigme posée par le personnage. La méthode reply() permet de répondre à un personnage dans le jeu. Tout d'abord, la méthode vérifie si la commande de l'utilisateur comporte un deuxième mot. Ensuite, la méthode divise le deuxième mot de la commande en deux parties séparées par ":". Si la division ne produit pas exactement deux parties, la méthode affiche un message indiquant à l'utilisateur qu'il doit fournir une réponse sous la forme "répondre personnage:réponse". Si aucun personnage ne correspond au nom donné, la méthode affiche un message indiquant que la personne n'existe pas. Le nombre d'erreurs est augmenté de un à chaque fois qu'une réponse est fournie. La méthode vérifie si la réponse fournie est correcte en faisant appel à la méthode checkAnswer(). Si la réponse est correcte, la méthode vérifie également si le personnage a une récompense associée. Si le personnage a une récompense et que le joueur ne l'a pas déjà reçue, et si le joueur peut porter le poids de la récompense, la récompense est ajoutée à son inventaire, et le poids du joueur est mis à jour. Si le personnage a une récompense mais le joueur l'a déjà reçue ou porte une charge trop lourde, un message est affiché.

Si la réponse fournie par le joueur est incorrecte, la méthode vérifie si le nombre d'erreurs du personnage dépasse 10. Si c'est le cas, un message de fin de jeu est affiché. Si la réponse est incorrecte mais le nombre d'erreurs du personnage est inférieur ou égal à 10, un message est affiché pour indiquer que la réponse était incorrecte.

```
149     /**
150      * Procédure permettant de répondre à un personnage.
151      *
152      * @param pCommand La commande entrée par l'utilisateur.
153      */
154     public void reply(final Command pCommand){
155         if(!pCommand.hasSecondWord()){
156             this.aEngine.getGUI().println("Quel est votre réponse ?");
157             return;
158         }//if
159         String [] vSecondWord = pCommand.getSecondWord().split(":");
160         if(vSecondWord.length < 2){
161             this.aEngine.getGUI().println("Vous devez renseigner : répondre
162             personnage:réponse");
163             return;
164         }
165         String vName = vSecondWord[0];
166         String vReply = vSecondWord[1];
167         Character vCharacter = this.aCurrentRoom.getCharacter(vName);
168         if(vCharacter == null){
169             this.aEngine.getGUI().println("Cette personne n'existe pas...");
170             return;
171         }//if
172         vCharacter.upNbError();
173         if(vCharacter.checkAnswer(vReply)){
174             if(vCharacter.getReward() != null) {
175                 if(!this.aItems.hasItem(vCharacter.getReward().getName()) &&
176                     (canTake(vCharacter.getReward().getWeight())))
177                     this.aEngine.getGUI().println("Bravo ! " + this.aPseudo + " a
178                     répondu correctement et gagne donc : " + vCharacter.getReward().getName());
179             this.aItems.addItem(vCharacter.getReward().getName(),vCharacter.getReward());
180             this.aWeight += vCharacter.getReward().getWeight();
181             return;
182         }
183     }
184     } else {
185         if(vCharacter.getNbError() > 10){
186             JOptionPane.showInternalMessageDialog(null,"Désolé...Vous avez
187             fait trop d'essais... \n Merci d'avoir joué au jeu " + this.aPseudo + ". Au
188             revoir !", "Message d'au revoir", JOptionPane.INFORMATION_MESSAGE);
189             System.exit(0);
190             return;
191         }
192     }
193 }
```

Puisqu'il s'agit de deux nouvelles commandes, elles doivent bien évidemment être ajoutées dans la liste des commandes valides et dans la méthode interpretCommand() dans la classe GameEngine. L'implémentation se fait de la même manière que d'habitude.

Cette nouvelle implémentation nécessite également des modifications dans la classe Room. On a besoin de créer une HashMap de personnages qui associe la String du nom du personnage avec le personnage, que l'on initialise dans le constructeur de la classe :

```
337     private HashMap<String, Character> aCharacters;//HashMap("nom", personnage)
338     this.aCharacters = new HashMap <String, Character>();
```

On doit ajouter à présent des méthodes qui permettront la gestion des personnages dans les pièces.

La première méthode est addCharacter() qui permet d'ajouter un personnage à la pièce. Elle prend en paramètre un objet de type Character et l'ajoute à la HashMap en utilisant le nom du personnage comme clé.

```
339 /**
340 * Ajoute un personnage à la pièce
341 *
342 * @param pCharacter Le personnage à ajouter
343 */
344 public void addCharacter(final Character pCharacter){
345     this.aCharacters.put(pCharacter.getName(), pCharacter);
346 } //addCharacter()
347
```

La deuxième méthode est getCharacterString() qui renvoie une description des personnages présents dans la pièce. Si la HashMap est vide, un message indiquant qu'il n'y a pas de personnages est ajouté à la chaîne. Sinon, pour chaque nom de personnage dans la HashMap, le nom est ajouté à la chaîne.

```
348 /**
349 * Renvoie une description des personnages présents dans la pièce.
350 *
351 * @return Une chaîne de caractères indiquant les personnages présents dans la
352 * pièce.
353 */
354 private String getCharacterString() {
355     StringBuilder returnString = new StringBuilder("Les personnages sont : ");
356     if(this.aCharacters.isEmpty())
357     {
358         returnString.append("Désolé...Il n'y a pas de personnages...");
359     } else {
360         for(String vJ : this.aCharacters.keySet())
361             returnString.append( " " + vJ + " - ");
362     }
363     return returnString.toString();
364 } //getCharacterString()
```

La troisième méthode s'appelle `getCharacter()` et elle permet de renvoyer le personnage correspondant au nom donné. Elle prend en paramètre le nom du personnage recherché et utilise ce nom pour accéder à la `HashMap` et récupérer le personnage correspondant. Si aucun personnage n'est trouvé, la méthode renvoie null.

```
366     /**
367      * Renvoie le personnage correspondant au nom donné.
368      *
369      * @param pCharacter Le nom du personnage recherché.
370      * @return Le personnage correspondant au nom donné, ou null s'il n'existe
371      * pas.
372      */
373     public Character getCharacter(final String pCharacter){
374         return this.aCharacters.get(pCharacter);
375     }//getCharacter()
```

La méthode `getLongDescription()` a quant à elle subit une modification depuis sa création initiale. En effet, elle renvoie à présent : le nom de la pièce, une description détaillée de la pièce, les sorties disponibles, les objets présents dans la pièce et ainsi les personnages présents. Elle utilise donc à présent la méthode `getCharacterString()` en plus afin d'obtenir la description des personnages présents.

```
376     /**
377      * Renvoie une description détaillée de cette pièce sous la forme :
378      * Vous êtes dans (pièce).
379      * Sorties : nord sud.
380      * Les objets dans cette pièce sont : (items)
381      * Les personnages dans cette pièce sont : (personnages)
382      *
383      * La description inclut le nom de la pièce, les sorties disponibles, les
384      * objets présents
385      * et les personnages présents.
386      *
387      * @return Une description de la pièce, ainsi que les sorties, les objets et
388      * les personnages.
389      */
390     public String getLongDescription()
391     {
392         return "Vous êtes dans " + this.aName + ", " + this.aDescription + "\n" +
393             this.getExitString() + "\n" + "Les objets dans cette pièce sont :" +
394             this.aRoomItems.getItemString() + "\n" + this.getCharacterString();
395     }//getLongDescription()
396 
```

Pour finir les modifications afin que les personnages soient correctement créés, il faut les ajouter dans la classe GameEngine, dans la méthode createRooms(). Pour ce faire, on ajoute un attribut dans cette même classe de type Character que l'on appelle aCharacter.

336

```
private Character aCharacter;//Personnages dans le jeu
```

Il ne nous reste plus qu'à créer tous les personnages de la manière suivante :

```
321      vChinjusha.addCharacter(new Character("Shizue","Je vois que tu possèdes le  
titre de samouraï et le shōjin ryōri qui atteste de ta méditation. Tu mérites donc  
de récupérer ce qui t'appartient.",null,vShi));  
322      vHokke_do.addCharacter(new Character("Sogen","Une récompense  
traditionnelle après une méditation intense et prolongée est appelée shōjin ryōri.  
Le shōjin ryōri est associé à la pratique bouddhiste zen et est réputé pour sa  
simplicité, sa fraîcheur et son équilibre.",null,vMeditation));  
323      vKaresansui.addCharacter(new SpecialCharacter("Haruki",null,null,vBadge));  
324      vKoro.addCharacter(new Character("Katakuri","Si tu réponds correctement à  
cette énigme, je te donnerai une récompense. Voici mon énigme : \n Cette chose,  
toutes choses dévore. Oiseaux bêtes arbres fleurs. Il ronge le fer et mord  
l'acier. Il réduit les cailloux en poussière. Il tue les rois et détruit les  
villes. Qui est-ce ?","temps", vKutsu));  
325      vHatto.addCharacter(new Character("Abbé","Le Bouddha a dit : Comme une  
mère au risque de sa vie protège son unique enfant, ainsi, avec un cœur sans  
bornes, devrais-je chérir tous les êtres vivants.", null, null));  
326      vKaresansui.addCharacter(new Character("Yuri", "Bienvenue dans le temple  
de Shizue. Je suis Yuri, un moine.\n Shi se trouve dans la pièce Chinjusha. Pour y  
accéder tu dois te munir d'une armure complète de samouraï. \nJe te conseille de  
charger ton Shukkō ki dans la pièce où tu le trouveras...", null,null));  
327      vDojo_1.addCharacter(new Character("Maître.Kai", "Je vois que tu  
progresses, prends cet objet, il t'aidera à progresser davantage.", null,vDo));  
328      vDojo_2.addCharacter(new Character("Maître.Zoro", "Je vois que tu  
progresses, prends cet objet, il t'aidera à progresser davantage.", null,vSode));  
329      vYakushi_do.addCharacter(new Character("Yakushi.Nyorai", null,  
null,vKabuto));  
330      vMi_do.addCharacter(new Character("Statue.Sacrée", null, null,vKusazuri));  
331      vKuri.addCharacter(new Character("Sanji","Bienvenue dans ma cuisine. Je  
suis Sanji, le cuistot du temple.\n Je travaille sur une recette et il me manque  
un ingrédient. As-tu une idée de ce que je pourrai rajouter ?\n Pourrais tu me  
l'amener ?",null, vSuneate ));  
332      vKyozo.addCharacter(new Character("Dan", "Je suis Dan et je suis en charge  
de protéger l'histoire du temple.\n Je connais absolument tout de l'histoire des  
temples. J'ai perdu cependant un livre. Pourrais-tu me l'amener ?", null,vMengu));  
333      vHojo.addCharacter(new Character("Prêtre", "Si tu réponds correctement à  
cette énigme, je te donnerai une récompense. Voici mon énigme : \n Tu mesures ma  
vie en heures et je te sers en expirant.\n Je suis rapide quand je suis mince et  
lente quand je suis grosse.\n Le vent est mon ennemi.\n Qui suis-je ?","bougie",  
vClé));
```

III) Ajout de commandes

J'ai eu besoin de rajouter deux commandes qui n'étaient pas demandé dans les exercices. La première est toWin() qui permet de vérifier si le joueur peut remporter la partie. Tout d'abord, la méthode vérifie si la salle actuelle du joueur a pour nom "Chinjusha". Si ce n'est pas le cas, cela signifie que le joueur n'est pas dans la bonne pièce pour récupérer l'objet "Shi" nécessaire pour gagner la partie. Un message est affiché, indiquant que le joueur doit continuer son aventure. Si la salle actuelle du joueur a pour nom "Chinjusha", la méthode vérifie si le joueur possède deux objets spécifiques : "Badge" et "Shi". Si le joueur a ces deux objets, cela signifie qu'il a récupéré "Shi" et peut donc gagner la partie. Le joueur est informé qu'il a gagné en récupérant "Shi", et un message de fin de jeu est affiché. Ensuite, le jeu se termine en utilisant System.exit(0) pour fermer l'application. Si le joueur ne possède pas les deux objets nécessaires, un message est affiché indiquant que le joueur n'a pas encore récupéré "Shi" et doit continuer son aventure.

```
1  /**
2   * Procédure permettant de vérifier si le joueur peut remporter la partie.
3   */
4  public void toWin(){
5      if(this.aCurrentRoom.getName().equals("Chinjusha")){
6          if(this.aItems.hasItem("Badge") && this.aItems.hasItem("Shi")) {
7              JOptionPane.showInternalMessageDialog(null,"Bravo ! Vous avez
gagné ! Vous avez récupérer Shi. \n Merci d'avoir joué au jeu " + this.aPseudo +
". Au revoir !", "Message d'au revoir", JOptionPane.INFORMATION_MESSAGE);
8              System.exit(0);
9          } else {
10              this.aEngine.getGUI().println("Désolé... Vous n'avez pas récupérer
Shi. Continuez votre aventure.");
11          }
12      } else {
13          this.aEngine.getGUI().println("Désolé... Vous n'êtes pas dans la bonne
pièce pour récupérer Shi. Continuez votre aventure.");
14      }
15  }//toWin()
```

Mon jeu a été testé par un camarade, et au vu du mapage de mon jeu, il m'a conseillé de créer une méthode carte afin d'afficher au joueur le plan de mon jeu pour lui faciliter la tâche. J'ai donc crée dans la classe GameEngine une commande carte. La méthode showMap() affiche la carte du jeu dans une boîte de dialogue en utilisant une image spécifiée. Tout d'abord, elle crée une instance de la classe ImageIcon en utilisant la méthode getResource() de la classe getClass(). Cette méthode permet de charger une ressource en utilisant un chemin d'accès. Cette utilisation a été donné dans le code de Bureau dans la classe UserInterface. Le chemin de l'image de la carte est définie par l'attribut suivant :

```
404 private static final String MAP_IMAGE = "./Images/Plan.png"; //Chemin d'accès à  
l'image de la carte du jeu
```

Ensuite, la méthode `showMessageDialog()` de la classe `JOptionPane` est appelée pour afficher une boîte de dialogue. Cette boîte de dialogue contient l'image de la carte. L'image est passée en tant que deuxième argument, après `null`, qui représente le composant parent de la boîte de dialogue. Le troisième argument est la chaîne de caractères "Carte", qui est le titre de la boîte de dialogue. Enfin, le dernier argument `JOptionPane.PLAIN_MESSAGE` spécifie le type de message à afficher.

```
394 /**
395 * Affiche une carte dans une boîte de dialogue.
396 * Utilise l'image de la carte spécifiée pour afficher la carte.*/
397 private void showMap(){//L'image de la carte.
398     final ImageIcon vMapImage = new
399         ImageIcon(getClass().getResource(MAP_IMAGE));//Affiche la boîte de dialogue
400         //contenant l'image de la carte.//Le titre de la boîte de dialogue est "Carte".
401         JOptionPane.showMessageDialog(null, vMapImage, "Carte",
402             JOptionPane.PLAIN_MESSAGE);
403     } //showMap()
404 }
```

Ces deux commandes possèdent des boutons et ont été ajoutées à la liste des commandes valides et au switch dans la méthode `interpretCommand()` dans la classe `GameEngine`.

IV. Déclaration obligatoire anti-plagiat

J'ai reçu l'aide de mon camarade Aman GHAZANFAR pour certains exercices compliqués, notamment les derniers. Lorsqu'une source extérieur m'a permis d'écrire le code, alors cette source est normalement renseigné directement dans l'exercice.