# Rapport A3P (Java A3P 2025/2026 G2)

## Le retour de Yamata no Orochi

### ヤマタノオロチの帰還

# Table des matières


**I) Informations sur le jeu (scénario e mécaniques)**.....3

**II) Réponses aux exercices (à partir de l'exercice 7.5)**.....7

1. Exercice7.5.....7
2. Exercice7.6.....7
3. Exercice7.7.....10
4. Exercice7.8.....11
5. Exercice7.8.1.....12
6. Exercice7.9.....12
7. Exercice7.10.....12
8. Exercice7.10.1/7.10.2.....12
9. Exercice7.11.....12
10. Exercice7.14/7.15.....13
11. Exercice7.16.....15
12. Exercice7.17.....15
13. Exercice7.18/7.18.2.....15
14. Exercice7.18.1.....16
15. Exercice7.18.3.....16
16. Exercice7.18.4.....16
17. Exercice7.18.6.....16
18. Exercice7.18.8.....17
19. Exercice7.19.2.....18
20. Exercice7.20.....18
21. Exercice7.21.....18
22. Exercice7.22.....18
23. Exercice7.23.....19
24. Exercice7.26.....19
25. Exercice7.28.1.....19
26. Exercice7.29.....20
27. Exercice7.30.....20
28. Exercice7.31.....20
29. Exercice7.31.1.....21
30. Exercice7.32.....21
31. Exercice7.33.....21
32. Exercice7.34.....21
33. Exercice7.42/7.42.1.....22
34. Exercice7.42.2.....22
35. Exercice7.43.....22
36. Exercice7.44.....23
37. Exercice7.46.....23
38. Exercice7.46.1.....24

**III) Mode d'emploi (si nécessaire, instructions d'installation ou pour démarrer le jeu)...25**

**IV) Déclaration anti-plagiat.....25**

# I. Informations sur le jeu(scénario et mécaniques)

## a) Auteur du jeu

Christophe TARATIBU

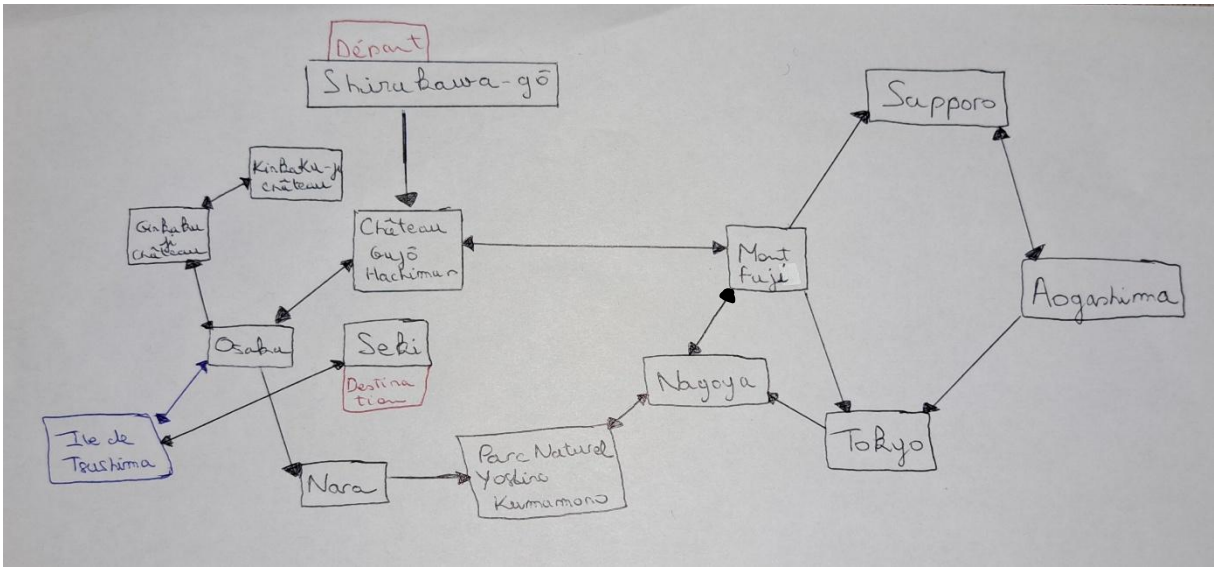
## b) Thème (Phrase thème validée)

Dans le Japon contemporain, le forgeron doit forger un sabre légendaire qui pourfend le mal.

## c) Résumé du scénario (complet)

Le Japon fait face à une crise sans précédent ! Yamata no Orochi le dragon maléfique à 8 têtes à fait son grand retour, ressuscité par des monstres maléfiques appelés Yôkais. Tetsuma Hasegawa, forgeron humain choisi par la déesse du soleil Amaterasu, a pour mission de parcourir le Japon afin de trouver les 9 matériaux qui permettra à Susanoo dieu des tempêtes et frère de la déesse de terrasser le monstre avec le sabre que Tetsuma aura forgé à Seki, le village des forgerons. Néanmoins quelques monstres puissants seront là pour compliquer sa tâche.

## d) Plan du jeu (Villes réelles du Japon)




## e) Scénario complet du jeu

Quand le jeu débute, le joueur se retrouve dans la ville de Sirakawa-gô ville ou le héros est originaire. Il entend alors la voix céleste d'Amaterasu lui disant que le monstre Yamata no Orochi est revenu et qu'il a pour mission de trouver 9 matériaux pour forger la lame de Totsuka, sabre qui permettra de tuer la créature. Ces matériaux sont dispersés à travers tout le Japon et le héros doit tous les trouver et créer la lame dans la ville de Seki, villages des forgerons où de nombreux sabres ont été forgées. Pour bien mener sa mission, le personnage doit passer par le château Gujô Hachiman qui a été transformé en lieu infini par un yokaï mystérieux, la fille à la guitare qui posera une question et suivant la réponse du héros, il sera téléporté à un endroit contenant un matériau ou non. A Osaka, pourra visiter la ville et manger. Sur l'île de Tsushima, il fera la rencontre du dieu des mers Ryûjin qui lui racontera un peu de mythologie du pays et lui donnera comme matériau ses écailles aquatiques puis le héros devra aller dans les châteaux de Kinkaku-ji et Ginkaku-ji pour trouver respectivement les poudres d'or et d'argent qui ont été subtilisés par deux yokaïs. Il devra trouver avoir la bénédiction du temple de Tokyo sinon il se fera tuer instantanément (c'est un humain après tout). Il pourra aller Nara ou il rencontra la déesse cerf (Shika no Megami) qui après avoir combattu les monstres qui attaquaient son sanctuaire fait don au héros de d'un de ses bois, nécessaire à la création de la lame. A l'est, le forgeron, se trouvera dans le parc national Yoshino ou il rencontrera les esprits de la forêt qui lui permettront de couper les

bambous à condition de leur faire une offrande de nourriture (La demande sera aléatoire). A Nagoya, le héros trouvera le Habaki (un collier en cuivre) nécessaire pour l'objet tranchant. Au mont Fuji le joueur sera accueilli par les esprits de la montagne et la récupération des pierres pourra se faire en trouvant la taille de l'ancien volcan dans le temps imparti. Si la hauteur n'est pas trouvée, les esprits banniront le joueur et il aura automatiquement perdu. S'il réussit il pourra récupérer du bois de magnolia à Sapporo et se reposer un peu (il y un sanctuaire de la déesse ici). Il ira à Aogashima récupérer les cendres de l'île mais il devra combattre le grand kappa qui lui posera des devinettes dans un temps imparti avec pour risque de noyer l'île tout entière. Ensuite, il pourra aller à Tokyo recevoir la bénédiction et une amulette qui permettra d'aller dans les châteaux à côté d'Osaka. Il aura fini sa quête et devra trouver son chemin pour aller à Seki forger enfin son sabre.

## f) Détail des lieux, items et personnages

Les salles de ce jeu représentent les villes et lieux du Japon. Une description générale des lieux est faite dans le code du jeu. Les objets du jeu ont aussi leur propre description.
Voici une description des lieux, personnages et objets du jeu :

- Shirakawa-go : C'est le village de départ de notre héros. Il entendra alors le message d'Amaterasu qui précisera sa mission. Il aura avec lui sa pioche et hache qui lui permettra de récupérer des matériaux nécessaires

- Château Gûjo Hachiman : C'est un château qui a été rendu magique par la fille à la guitare qui expliquera le pouvoir du lieu. Cela désorientera le héros. Néanmoins c'est le lieu qui dirigera la quête de ce dernier.

- Osaka : La grande ville du sud-ouest du Japon.

- Île de Tsushima : Île situé entre la Corée du Sud et le Japon, lieu de villégiature du dieu de l'eau et des océans Ryûjin, le joueur pourra récupérer des écailles aquatiques.

- Kinkaku-ji : Château pris en possession par la démone blanche qui contient la poudre dorée. . Le joueur doit avoir l'amulette de Tokyo pour pouvoir récupérer ce matériau.

- Ginkaku-ji : Château pris en possession par le démon noir qui contient la poudre argentée. Le joueur doit avoir l'amulette du dragon des glaces pour pouvoir récupérer ce matériau.

- Nara : Ville du Japon connu pour son parc de cervidés, le joueur rencontrera la déesse cerf blessée qui lui racontera l'attaque de son sanctuaire et lui donnera un de ses bois.

- Parc national Yoshino : Parc connu pour ses forêts des cerisiers (Sakuras). Le joueur trouvera les esprits du parc et pourra récolter des bambous.

- Nagoya : Une des grandes villes du Japon où le joueur pourra trouver le habaki, collier de cuivre nécessaire à la création de l'arme.

- Mt Fuji : Mont montagneux qui fait la célébrité du Japon, le héros rencontrera les esprits de la montagne et pourra récolter la roche qui en est issu.

- Sapporo : Grande ville du Japon, le joueur pourra récolter du magnolia, matériau important pour la confection de l'arme.

- Île d'Aogashima : Îlot de terre perdu au beau milieu de l'océan Pacifique.  Le joueur devra récolter du sel de ce lieu mais devra battre avant le grand kappa, un yokaï des eaux qui mettra le héros à l'épreuve.

- Tokyo : Ville qui fait la renommée de ce pays et c'est la capitale. Le joueur rencontrera la déesse Amaterasu dans son sanctuaire qui le bénira et lui offrira une amulette, symbole de sa bénédiction.

- Seki : Village des forgerons ou le joueur pourra enfin créer l'arme avec les matériaux trouvés. Le joueur aura gagné et le jeu pourra s'achever.

## g) Situations gagnantes et perdantes :

Pour les situations gagnantes, le joueur gagnera dès qu'il aura trouvé tous les matériaux nécessaires à la forge de la lame, il perdra à coup sûr s'il se rend dans les châteaux possédés par les démons sans protection ou s'il ne réussit pas les diverses épreuves qu'il rencontrera dans sa quête.

## h) Eventuellement énigmes, mini-jeux, combats, etc.:

Quelques énigmes et conditions vont êtres implémentés dans ce jeu pour ajouter un peu de réflexion et de difficulté.

### i) Commentaires (ce qui manque, reste à faire, ...) :

Le jeu à une bonne configuration des salles mais manque encore de partie graphique qui affiche les différents lieux explorés par le joueur.

# II. Réponses aux exercices (à partir de l'exercice 7.5)

## 1. Exercice 7.5

En ce qui concerne cet exercice, il y a la présence dans la classe Game, de lignes de codes d—upliquées qui sont présentes dans les procédures `goRooms(Command pCmd)` et `void welcome()` dont l'une permet de changer de pièce et l'autre qui affiche la fenêtre de bienvenue.
Voici les lignes en question :

```
System.out.println(" You are " +
this.aCurrentRoom.getDescription());

System.out.println("Possible exits");
if (this.aCurrentRoom.aEastExit != null) {
      System.out.println("- East");
}
if (this.aCurrentRoom.aWestExit != null) {
      System.out.println("- West");
}
if(this.aCurrentRoom.aNorthExit != null){
      System.out.println("- North");
}
if (this.aCurrentRoom.aNorthExit != null) {
      System.out.println("- North");
}
if (this.aCurrentRoom.aSouthExit != null) {
      System.out.println("- South");
}
```

Ces lignes ci-dessus permettent affichage de la salle courante et L'affichage des sorties possibles de cette dernière matérialisée par les points cardinaux (Nord, Sud, Est et Ouest en anglais). Ainsi pour éviter la duplication de code, la création d'une procédure est nécessaire. C'est là qu'intervient la procédure

privée `printLocationInfo()` qui effectue ce travail.

```
private void printLocationInfo(){

	System.out.println(" You are " +
	this.aCurrentRoom.getDescription());
	System.out.println("Possible exits");
	if (this.aCurrentRoom.aEastExit != null) {
		System.out.println("- East");
	}
	if (this.aCurrentRoom.aWestExit != null) {
		System.out.println("- West");
	}
	if(this.aCurrentRoom.aNorthExit != null){
		System.out.println("- North");
	}
	if (this.aCurrentRoom.aNorthExit != null) {
		System.out.println("- North");
	}
	if (this.aCurrentRoom.aSouthExit != null) {
		System.out.println("- South");
	}

}
```

Les procédures qui contenaient du code dupliqué contiennent dorénavant la ligne de code `this.printLocationInfo();` qui permet l'exécution de la nouvelle procédure.

## 2. Exercice 7.6

Le code du jeu est fonctionnel mais les attributs de la classe Room sont publics et cela ne respecte pas les conventions du langage Java.

```
public Room aNorthExit;
public Room aEastExit;
public Room aSouthExit;
public Room aWestExit;
```

Attributs publics dans la classe Room

Ainsi la classe `Game` avait accès aux attributs de la classe `Room` .

Cependant les sorties étaient énumérées et ceci devenait vite répétitif en cas de multiplication de nouvelles sorties. Afin de compenser cela, on utilise l'encapsulation qui octroie une meilleure protection des attributs en en remplaçant le mot clé java `public` par un autre mot clé `private` . Ceci évite une modification anarchique des données et/ou d'invariants d'une classe qui compromettrait tout le code final. Une méthode publique de la classe `Room Room getExit(String vDirection)` qui fait office de getter permet de récupérer les sorties possibles d'une salle en indiquant un point cardinal.

```
public Room getExit(String vDirection){
	switch (vDirection) {
		case "north":
			return this.aNorthExit;
		case "south":
			return this.aSouthExit;
		case "east":
			return this.aEastExit;
		case "west":
			return this.aWestExit;
		default:
			return null;
	}
}
```

Par conséquent la procédure privée `private void goRooms(Command pCmd)` se retrouve aussi modifiée car les attributs de la classe `Room` sont maintenant privés.

```
private void goRooms(Command pCmd){
	if(!pCmd.hasSecondWord()){
		System.out.println("Go where ?");
	}
	Room vNextRoom = null;
	String vDirection = pCmd.getSecondWord();
	System.out.println(vDirection);
	switch (vDirection) {
		case "north":
		vNextRoom = this.aCurrentRoom.getExit(vDirection);
		break;

		case "south":
		vNextRoom = this.aCurrentRoom.getExit(vDirection);
		break;

		case "east":
```

```
                vNextRoom = this.aCurrentRoom.getExit(vDirection);
                break;
                case "west":
                vNextRoom = this.aCurrentRoom.getExit(vDirection);
                break;
                default:
                System.out.println("Unknown direction");
                break;
        }
        if(vNextRoom == null){
        System.out.println("There's no door");
        } else {
                this.aCurrentRoom = vNextRoom;
                this.printLocationInfo();
        }
}
```

## 3. Exercice 7.7

La classe `Room` a pour mission la création des salles de pour le jeu de zuul. On utilisera la méthode publique `public String getExitString()` qui servira de stockage pour les sorties possibles d'une salle. Donc la classe principale du jeu, `Game` pourra récupérer la « liste » de salles qui est contenue.

```
public String getExitString(){
        String vStrExits = "";
        if(this.getExit("south") != null){
                vStrExits += "South ";
        }
        if(this.getExit("north") != null){
                vStrExits += "North";
        }
        if(this.getExit("east") != null){
                vStrExits += "East";
        }
        if(this.getExit("west")!= null){
                vStrExits += "West";
        }
        return vStrExits;
}
```

## 4. Exercice 7.8

La nouvelle classe `HashMap` va nous permettre de stocker les salles avec leurs sorties respectives. On pourra même ajouter des directions verticales comme bas et haut. L'affectation des sorties avec les salles attribuées sera plus simple avec la méthode `put` propre à la classe choisie. Cela va faciliter l'affectation des sorties possibles des salles sans passer par l'ancienne méthode qui consistait à attribuer les sorties d'une salle par 4 suivant les points cardinaux.

`import java.util.HashMap;` -> permet d'invoquer la classe avec les méthodes qui vont avec
`private HashMap<String, Room> aExits;` -> Sorties possibles d'une pièce représentés par une table de hachage.
Voici une comparaison entre l'ancien code et le nouveau code la procédure `public void setExits`:

```
  // Ancien code
     public void setExits(final Room pNorthExit,final Room pEastExit,
  final Room pSouthExit, final Room pWestExit){
        if(pNorthExit != null){
        this.aExits.put("north", pNorthExit);
    }
    if(pEastExit != null){
        this.aExits.put("east", pEastExit);
    }
    if(pSouthExit != null){
        this.aExits.put("south", pSouthExit);
    }
    if(pWestExit != null){
        this.aExits.put("west", pWestExit);
    }
}

// Nouveau code

public void setExit(String pDirection, Room pNeighbor){

    this.aExits.put(pDirection, pNeighbor);
}
```

On se retrouve ainsi avec du code plus simple et concis. Pour la définition des sorties d'une salle :

```
  vOsaka.setExits(vGinkaku,vGujoHachi,vNara,vTsushi); // Salle qui
  représente la ville d'Osaka

  vOsaka.setExit("north", vGinkaku);
```

```
vOsaka.setExit("east", vGujoHachi);
vOsaka.setExit("south", vNara);
vOsaka.setExit("west", vTsushi);

// Nouvelle représentation des sorties de Salles
```

## 5. Exercice 7.8.1

Déplacement vertical fait : Cependant, il y a un seul mouvement « up » et d'autres mouvement « down » suivant le point cardinal souhaité pour descendre.

## 6. Exercice 7.9 / 7.10

La classe Set du langage java permet de stocker les éléments choisis dans un « ensemble ». Ils sont uniques dans cette structure de donnée et aucun doublon n'est autorisé. La structure de donnée `keySet()` permet de stocker les clés de la `HashMap` (en l'occurrence `aExits.` On utilisera la méthode `keySet()` qui permettra de récupérer les clés de la `HashMap`. Une boucle for each sera utilisée pour itérer sur le set (qui est une liste de string unique de sorties possible d'une salle) et concaténer avec la variable `vExitString` qui contiendra les sorties possibles d'une salle.

Voici le code qui répond à l'exercice demandé :

```
public String getExitString(){
	String vExitString = "Exits:";
	for(String vDirection : this.aExits.keySet()){
		vExitString += " " + vDirection;
	}
	return vExitString;
}
```

## 7. Exercice 7.10.1 / 7.10.2

Javadoc faite.

## 8. Exercice 7.11

Dans cet exercice, le couplage a été réduit afin de faciliter l'ajout de nouvelles descriptions pour la salle. Avec l'ancienne procédure,

la description complète des salles et des sorties était gérée par deux autres procédures de la classe `Room`. Une unique procédure affiche dorénavant la description d'une salle en concaténant la description principale et les sorties possibles :

```
public String getLongDescription(){
	return "You're " + this.aDescription + getExitString();
}
 //Dans la classe Room

 private void printLocationInfo(){

	System.out.println(this.aCurrentRoom.getLongDescription());
 }
 //Dans la classe Game
```

## 9. Exercice 7.14/7.15

Pour limiter le découplage, il est essentiel de localiser les modifications : une classe doit pouvoir être modifiée sans impacter les autres. Les champs publics peuvent introduire divers niveaux de couplage, dont le plus problématique est le couplage implicite, lorsque qu'une classe dépend indirectement des données internes d'une autre. Ce risque apparaît, par exemple, lors de l'ajout d'une nouvelle commande au jeu, en plus de quit, help et go. La commande look, destinée à réafficher la description de la pièce actuelle et ses sorties, doit être implémentée de manière à éviter tout couplage implicite. Dans la classe `CommandWords`, la commande look a été introduite dans le tableau statique des commandes principales et une procédure look, à été codée et réaffiche la description de la salle où se situe le joueur au cas où il serait perdu. La nouvelle commande eat est aussi créée en tant qu'utilitaire et le code est implémenté de la même manière. Voici le code modifié :

Dans la classe `CommandWords`:

```
public CommandWords()

{
this.aValidCommands = new String[5];
this.aValidCommands[0] = "go";
this.aValidCommands[1] = "help";
this.aValidCommands[2] = "quit";
this.aValidCommands[3] = "look";
```

```
        this.aValidCommands[4] = "eat";



        }
```

Et dans la classe Game :

```
        private void look(){

              System.out.println(this.aCurrentRoom.getLongDescription());
        }
        private void eat(){

              System.out.println("You have eaten now and you are not
    hungry any more.");
        }


        private boolean processCommand(final Command pCmd) {

              if (pCmd.isUnknown()) {
                    System.out.println("I don't know what you mean...");
                    return false;
              }
              String vCommandWord = pCmd.getCommandWord();
              if ("help".equals(vCommandWord)) {
                    printHelp();
                    return false;
              } else if ("go".equals(vCommandWord)) {
                    goRooms(pCmd);
                    return false;
                    } else if ("quit".equals(vCommandWord)) {
                          return quit(pCmd);

                    } else if("look".equals(vCommandWord)){
                          this.look();
                          return false;

                    } else if ("eat".equals(vCommandWord)){
                          this.eat();
                          return false;
                    } else {
                                System.out.println("Erreur du programmeur
: commande non reconnue !");
                          return true;
                    }
              }
```

## 10. Exercice 7.16

Après l'ajout des commandes eat et look, la procédure `void printHelp()` se retrouve alors incomplète. Comme solution la procédure `void showAll()`dans la classe `CommandWords` va faire une boucle for each sur la liste des commandes et énumérer les commandes possibles en les affichant en sortie.

```
public void showAll(){
    for(String vCmd : this.aValidCommands){
          System.out.println(vCmd + " ");
    }
    System.out.println();
}
```

## 11. Exercice 7.17

Si le développeur souhaite ajouter une nouvelle commande, il devra modifier la méthode `boolean processCommand(final Command pCmd)` permettant d'ajouter des commandes liées à de nouvelles actions.

## 12. Exercice 7.18/7.18.2

On ne veut plus utiliser `void showAll()` de la classe `CommandWords` pour afficher les commandes valides du jeu. Afin de limiter le couplage implicite de classes, la méthode `String getCommandList()` qui a le même fonctionnement que `void showAll()` mais avec une amélioration particulière : les commandes définies dans le tableau `String[] aValidCommands` sont ajoutées dans nouvelle structure du langage java qui sont les `stringBuilder` qui sont issues de la classe `StringBuilder` qui facilite la manipulation des chaînes de caractères. La méthode `String showCommands()` retourne la chaîne de caractères contenant les commandes possibles du jeu. La procédure `void printHelp()` classe `Game` permettra l'affichage des commandes valides en console.

## 13. Exercice 7.18.1

Après la comparaison avec le projet zuul-better, des modifications ont été faites, notamment du nettoyage de code avec la suppression de lignes redondantes et/ou encore inutiles qui entravent la compréhension et le bon fonctionnement du code.

## 14. Exercice 7.18.3

La recherche d'images pour l'illustration du jeu a été faite.

## 15. Exercice 7.18.4

Le nom choisi pour mon jeu est : Le retour de Yamata no Orochi.
Le message de bienvenue sera affiché dans la classe `Game`.

## 16. Exercice 7.18.6

L'objectif ici est d'ajouter des images dans mon jeu. Avec l'aide du projet « zuul-with-images » laissé pour cette exercice, l'agencement du code du jeu a été modifié. Les classes `CommandWords` et `Command` restent inchangées. En ce qui concerne les autres classes du jeu, de nouvelles classes nommées `GameEngine` et `UserInterface` font leur apparition. Voici les modifications faites dans la nouvelle version du jeu avec les images :

- Classe `Room`: Ajout d'un nouvel attribut permettant de stocker le chemin relatif (dossier ou les images sont localisés du jeu) et d'une nouvelle méthode `String getImageName()` qui permet de récupérer le chemin d'une image tout en modifiant le constructeur.

- Classe `Parser`: Une nouvelle classe du langage Java, `StringTokenizer` est utilisée à la place de la classe `Scanner`. Les commandes ne sont plus lus via le clavier de l'utilisateur. L'attribut `private CommandWords aCommandWords;` est ajouté, le constructeur change aussi.

- Classe `GameEngine`: Cette reprend l'ancien code de la classe `Game` et y ajoute les méthodes `void setGUI()` et les procédures relatives à la fenêtre graphique qui sont `void println()` et `void print()` qui modifient respectivement la fenêtre graphique du jeu (l'interface) et y affiche du texte. La procédure `void printWelcome()` est modifiée avec l'ajout des instructions du jeu relatif au scénario. La procédure `void createRooms()` ajoute à chaque salle, une image associée à sa description. `void interpretCommand()` en est une nouvelle qui remplace l'ancienne procédure `void processCommand()` car cette dernière est utilisée dans la classe `UserInterface`. Cette nouvelle procédure permet d'interpréter les commandes choisies par le joueur au clavier. Les procédures `void endGame()` et `void goRooms(Command pCmd)` permettent respectivement de finir le jeu (aGUI gère la fin du jeu) et de changer de salle en mettant à jour l'image de la prochaine salle.
- Classe `Game`: Le code de la classe est déplacé dans la classe `GameEngine` et le fichier ne fait plus que le code principal du jeu (« le main du jeu »).
- Classe `UserInterface`: Cette sert en grande partie à la gestion de la fenêtre graphique du jeu.

## 17. Exercice 7.18.8

La création des boutons du jeu s'effectue avec la classe `javax.swing.JButton`. Des attributs sont créés dans la classe `UserInterface` pour chaque bouton et sont initialisés par la suite dans le constructeur. Dans la procédure privée qui permet d'organiser la fenêtre graphique `void createGUI()`, les panel (issus de la classe `JPanel`) pour le texte les images (la façon dont les données du jeu y sont organisés) sont déjà créés et 5 boutons sont fabriqués (les points cardinaux et le bouton quitter,

« quit ») et placés en bas de la fenêtre graphique. Afin de permettre que ces boutons soient « cliquables » et réalisent des actions, Un listener (écouteur) est assigné pour vérifier que ces boutons soient bien cliqués. Enfin, les commandes (« go … ») sont associées aux boutons avec la procédure publique `void actionPerformed(final ActionEvent pE)` qui réceptionne les actions faites par le clic (souris) de l'utilisateur.

## 18. Exercice 7.19.2

Les images qui illustrent le jeu, ont été déplacés dans le dossier Images du projet. Le chemin relatif est utilisé dans la classe `GameEngine` qui affiche correctement les images relatives aux salles dans la mesure ou le joueur change de salle.

## 19. Exercice 7.20

Afin d'ajouter des items au jeu, on crée une classe `Item` qui s'occupe de la gestion des objets présents dans les salles (Rooms). Le constructeur de la classe permet son initialisation. Plusieurs getters sont présents qui retournent le nom de l'item, son poids et sa description complète. Un setter présent dans la classe `Room` permet une création d'un seul et unique item propre à la salle avec son poids. Leur définition se fait au même endroit que les salles du jeu, dans la classe `GameEngine`. La méthode publique de cette même classe, `String getLongDescription()` se retrouve modifiée.

## 20. Exercice 7.21

Les informations propres à l'item (nom, poids) sont présentes dans la classe `Item` renvoyées par la méthode publique `String getItemDescription()`. La méthode publique `String getLongDescription()` présente dans la classe `Room` se retrouve alors modifiée et qui affiche en plus les informations concernant l'item présent.

## 21. Exercice 7.22

L'objectif de cet exercice est de mettre plusieurs items dans une salle. Afin de résoudre ce problème, une `HashMap<String, Room>` contenant ayant pour clé le nom de l'item et pour valeur l'item avec sa description et son poids à été implémenté dans la classe `Room` en tant que nouvel attribut de cette classe. Un getter `public Item getItem(final String pName)` et une méthode d'ajout `public void addItem(final String pName, final`

`Item pItem)` ont été ajoutées pour respectivement récupérer un item par son nom et ajouter plusieurs items à une salle. Dans la classe `Items` la méthode publique `public String showItem()` permet d'afficher dans la fenêtre graphique, les objets présents dans la salle.

## 22. Exercice 7.23

La nouvelle commande back a été ajoutée dans la classe `CommandWords` qui contient l'attribut `aValidCommands` contenant lui-même la liste des commandes valides. J'aurais pu commencer par utiliser une un attribut qui permet de mémoriser le changement d'une salle dans la classe `GameEngine` mais j'ai préféré utiliser une LinkedList `private LinkedList<Room> aLastRooms` qui permet de stocker les dernières salles (représentées par le type `Room`) dans des « containers » et qui sont liées suivant le principe d'une liste chaînée. La procédure privée `private void back()` permet alors de retourner dans la dernière salle qui a été visitée par le joueur. Ceci est fait grâce à la méthode `removeLast()` de la classe `LinkedList` qui permet de supprimer le dernier élément de ce type de liste. Cette nouvelle commande est alors ajoutée dans la procédure publique `public void interpretCommand(final String pCmdLine)`.

## 23. Exercice 7.26

La classe `Stack` et les méthodes `empty(),push(Room)` et `pop()` ont été utilisées afin de satisfaire les exigences de la consigne de cet exercice.

## 24. Exercice 7.28.1

L'objectif ici est de tester le jeu, afin d'en voir son bon fonctionnement. Ainsi, 3 fichiers de tests (en extension .txt) ont été écrits : une version de parcours court, une version d'exploration complète et une idéale. Une nouvelle commande, `"test"`, a été implémentée dans la classe `CommandWords` qui gère les commandes et une condition a été ajouté dans la procédure publique `public void interpretCommand(final String pCmdLine)` présente dans la classe `GameEngine`. Dans cette même classe une procédure privée de test, `private void testFile (final Command pCommand)` permet l'exécution d'un

fichier de test en simulant des commandes qui pourraient être faite par un joueur.

## 25. Exercice 7.29

Avec la création de la classe `Player`, les informations qui concernaient le joueur (items, lieu où il se trouve et lil se trouvait) se retrouvent centralisés dans cette classe pour plus de clarté, du « refactoring » a été faite pour cet exercice. Pour créer complètement un joueur, on ajoute comme attribut de classe son nom et toutes les caractéristiques de position qui concernaient le joueur et qui étaient présentes dans la classe `GameEngine`.

## 26. Exercice 7.30

Pour permettre au joueur de prendre et déposer un objet, les nouvelles commandes `"take","drop"`, sont implémentées dans la classe `CommandWords`. Pour permettre au joueur d'avoir des items, on ajoute un attribut de type `Items` avec une procédure d'ajout `public void addItem(final String pName, final Item pItem)` et une autre procédure de suppression `public void removeItem(final String pName)`. La même procédure de suppression est utilisée pour retirer un item dans une salle. Les procédures liées à ces nouvelles commandes, sont ajoutées dans le moteur du jeu et de nouvelles procédures y sont aussi ajoutées.

## 27. Exercice 7.31

L'objectif ici de cet exercice est de permettre au joueur de porter plusieurs objets (Items). Ainsi l'attribut `aItems` est créé. Pour éviter que la collection d'items ne soit manipulée depuis l'extérieur, une `HashMap` qui a pour clé une chaîne de caractères (`String`) et un objet (`Item`) est utilisé comme type d'attribut et est initialisé dans le constructeur de la classe `Player`. Des méthodes utilitaires sont alors créées pour effectuer des opérations sur la collection (`addItem`, `removeItem`, et `getItem`)

## 28. Exercice 7.31.1

La nouvelle classe `ItemList` est créé et contient tout code en lien avec les Items autrement dit tout ce qui est dans les classes `Room`, `Player`.

## 29. Exercice 7.32

Afin de connaître le poids total des objets portés par le joueur, un attribut entier et une constante sont ajoutés dans la classe `Player`: `aCurrentWeight` (pour connaître le poids actuel des objets portés par le joueur) et `MAX_WEIGHT` (une constante exprimant le poids maximal que peut porter le joueur). Un getter et setter sont alors implémentés dans cette même classe (`getWeight` et `setWeight`) pour utiliser/modifier le poids. Les procédures `take` et `drop` de la classe `GameEngine` pour prendre en compte le poids de l'inventaire du joueur. Un message avertissant le joueur est écrit dans la fenêtre graphique dès que le poids de l'inventaire dépasse la limite autorisée.

## 30. Exercice 7.33

L'objectif ici est d'avoir l'inventaire du joueur. Une nouvelle méthode `getItemNames` de la classe `ItemList` qui retourne les items (clés de la HashMap `aItemList`) sous la forme d'une liste de chaîne de caractères. Enfin dans la classe `Player`, une nouvelle méthode est créée : `getItems` qui retourne la liste des items de l'inventaire du joueur séparés par des virgules (grâce à la méthode `join` de la classe `String`), qui appelle la méthode de la classe `ItemList`. Pour finir procédure privée `playerInventory` est créé afin de récupérer les données relatives à l'inventaire du joueur.

## 31. Exercice 7.34

Pour augmenter le poids maximal que l'inventaire peut supporter, et suivant le thème du scénario du jeu un « Anko Mochi » (pâtisserie japonaise a été ajouté) pour doubler le poids maximal supporté dans ce dernier. Pour pouvoir le manger, la procédure `eat` de la classe `GameEngine` a été modifiée pour que le joueur puisse manger ce petit gâteau. Dès lors que le joueur choisit de le manger, cette action exécute la méthode `superPower` de la classe `Player` qui multiplie par 2 le poids maximal de l'inventaire. Le paramètre pris en compte dans la méthode `eat` ici est la commande entière. Puis le second mot est récupéré et comparé pour voir si l'item correspond. Des messages avertissent le joueur dans les cas d'impossibilité : (item non comestible, mauvais item…).

## 32. Exercice 7.42/7.42.1

En ce qui concerne la gestion de la limite du temps, un attribut de type `Timer` a été créé (`aGameTimer`) et une procédure privée (`startGameTimer`) a été implémentée pour pouvoir activer/réactiver le minuteur du jeu. D'autres méthodes privées (`remainingTime` et `formatTime`) présentes dans la classe `GameEngine` permettent respectivement de calculer le temps restant du minuteur et de l'afficher dans la fenêtre graphique. En ce qui concerne l'affichage de l'heure, un attribut `aTimerLabel`, de type `Jlabel` est créé pour permettre son affichage et indiquer au joueur le temps restant. Son affichage est permis par la procédure `updateTimerLabel` présent dans la classe `UserInterface`.

## 33. Exercice 7.42.2

L'interface homme-machine est conservée ici par souci de commodité mais d'autres implémentations ont été ajoutées comme des boîtes de dialogues pour choisir l'objet que le joueur souhaite jeter/prendre, des fichiers à tester ou encore l'ajout d'un minuteur.

## 34. Exercice 7.43

Dans cet exercice, la méthode back peut doit être seulement utilisés lorsqu'il est possible de retourner dans la pièce précédente. La méthode `isExit` qui est présente dans la classe `Room` vérifie que la salle est bien une sortie. Les trapdoors sont ajoutées dans le jeu grâce la méthode publique `setTrapDoorExit` qui est dans cette même classe et permet de mettre les portes « pièges ». On ajoute dans la méthode `back` une condition qui vérifie si la salle a une porte piège à la sortie de la pièce. Il devient alors impossible de retourner en arrière. Cette condition exécute la méthode `emptyPreviousRooms` de la classe `Player` qui vide la pile de salle après le passage des portes pièges.

## 35. Exercice 7.44

Ici, le but de cet exercice est de permettre au joueur de se téléporter d'une salle à l'autre suivant le principe du « beamer » (sujet zuul). L'objet est remplacé par un kunai spatio-temporel

(pour maintenir la cohérence du scénario) dans lequel le joueur peut le planter dans une salle (illustré par la commande `plant`) et puis retourner dans la salle où il est planté (avec la commande `use` qui sont été ajoutées dans la classe `CommandWords`). Afin de mémoriser la salle où l'arme est plantée, un nouvel attribut y est ajouté dans la classe `Player`: `aPlantedRoom`. Des méthodes supplémentaires ont été implémentées pour récupérer et affecter la salle ou l'arme est située (`getPlantedRoom` et `setPlantedRoom`). Dans la classe `GameEngine`, de nouvelles méthodes ont été implémentées `plant` et `use`: qui permettent respectivement de planter l'arme dans une salle (prend une commande en paramètre) « plant Space-time kunai » et d'utiliser le pouvoir de l'arme afin de se téléporter (prend elle aussi une commande en paramètre) « use Space-time kunai ». L'arme est à utilisation unique. Dès lors que la commande d'utilisation est faite le joueur ne peux plus faire de retour en arrière.

## 36. Exercice 7.46

Pour cet exercice, une nouvelle classe est ajoutée : `TransporterRoom` qui se base sur la classe `Room`. Ainsi l'héritage est utilisé pour cette la classe. Son principe est le même que la classe mère mais dès lors que le joueur rentre dans ce type de salle, il est téléporté dans une autre salle aléatoirement. Dans la méthode `createRooms` qui est dans la classe `GameEngine` une salle change alors de type (pour respecter la consigne de cet exercice). La classe `TransporterRoom` contient alors comme attributs la liste des salles possibles (où le joueur peut atterrir, `aPossibleRooms`, arrayList) et la salle choisie (`aChosenRoom`). La méthode `setDestinations` permet alors d'affecter les salles possibles où le joueur peut se rendre de manière aléatoire et une autre méthode : `getRngNumber`, retourne alors un nombre qui est compris entre 0 et la taille de liste des salles possibles avec la méthode `nextInt` de la classe `Random`. Des passages sont ajoutés aussi pour permettre de téléporter plus d'une fois le joueur quand il y retourne. Afin de pouvoir faire fonctionner cette nouvelle classe, une nouvelle condition est ajoutée dans le moteur de jeu. Le type de la salle vérifiée avec le mot clé

`instanceOf`. Par la suite, les anciennes salles parcourues par le joueur sont alors supprimées dans cette même condition.

## 37. Exercice 7.46.1

L'exercice demande l'implémentation d'une nouvelle commande, `alea` pour faciliter les tests de la transporterRoom. Cette nouvelle commande est alors ajoutée dans la liste des commandes prises en compte dans la classe qui gère les commandes `CommandWords` puis une condition est ajoutée dans `interpretCommand` (présente dans le moteur du jeu) pour permettre l'exécution de la méthode éponyme (liée à la commande `alea`). La méthode `testFile` est liée à la commande `test` qui permet de tester le fichier qui est en second mot de la commande. L'attribut `aTestMode` qui est un des composants de la classe `GameEngine` ajoute un mode test à ce projet. Cet attribut prend la valeur `true` après la récupération des commandes présentes dans le fichier de test testé. Lors du test du fichier, les commandes sont faites toutes les 4 secondes : un objet de type `Timer` permet l'espacement et la bonne exécution des fichiers. Concernant la méthode `alea`, le second de la commande « force » la téléportation dans une des salles liées à la transporterRoom, dans une liste qui à été définie lors de la création des salles. Si la commande est lancée sur une salle qui n'est pas liée, un message d'erreur s'affiche sur la fenêtre graphique est le joueur est téléporté de forces dans une des salles qui y sont liées. Pour pouvoir forcer la téléportation dans une des salles, un autre attribut est défini : `aAleaString` stocke la salle dans laquelle test force la téléportation. Au départ, il est nul mais suivant l'avancement de l'exécution du test, il prend comme résultat, le nom de la salle (String). D'autres méthodes/procédures interviennent pour affecter la salle correctement lors de ce processus de test (`setChosenRoom`, de la classe `TransporterRoom` affectant la salle choisie suivant le nombre aléatoire). Sachant que le choix est forcé ici le nombre aléatoire est la position du nom de la salle dans la liste des pièces liées à la transporterRoom.

# III. Mode d'emploi (démarrage du jeu)

Le jeu peut se jouer en ligne de commande (Cependant un programme main est présent pour permettre un bon fonctionnement du jeu). Au démarrage du jeu, l'utilisateur saisit son nom et peut commencer à jouer.

# IV. Déclaration anti-plagiat

Le scénario crée s'inspire beaucoup de la mythologie japonaise et du manga Kimetsu no Yaiba (Les Rôdeurs de la nuit). L'idée du forgeron se base réellement sur de vraies personnes qui ont créé des armes qui ont marqué l'histoire du pays. Les exercices faits ont pour base le Zuul-Bad et les exercices du chapitre 7 du livre de Java. Pour les améliorations de l'IHM, des recherches ont été faites sur Stackoverflow et la documentation du langage java (Swing et autres bibliothèques). Le pouvoir du beamer (kunai spatio-temporel) se base sur une arme existant dans l'univers de Naruto Shippuden. Les images choisies ont été trouvées sur Google (une recherche inverse retrace leur source).