Rapport Projet Zuul

2020/2021

Wandrille Legras G7

Thème : L'enfant de dragon doit tuer le dragon qui rôde en Bordeciel.

Résumé du scénario :

Après avoir échappé à une attaque de dragon en fuyant dans une grotte vous sortez de l'autre côté et cherchez alors à avertir votre Jarl à Blancherive de la présence de dragon en Bordeciel.

Une fois le Jarl informé, celui-ci impressionné de votre initiative à vous être personnellement déplacé pour le prévenir vous invite à aller parler au sorcier de sa cour. Le sorcier vous donne pour mission de récupérer une tablette ancienne pouvant lui donner des informations sur les dragons. Celle-ci se trouve au Tertre des Chutes Tourmenté. Vous partez la chercher, une fois trouvée et arrivé à bout des Draugr qui gardait l'endroit, vous trouvez la tablette dans une tombe ainsi qu'une immense stèle portant des inscriptions indéchiffrables. Vous vous approchez et un souffle puissant vous traverse le corps et une voix semble sortir de nulle part, choqué vous n'oublierez pas de parler de ceci au sorcier de la cour. Vous sortez et rapportez ce que vous avez trouvé au sorcier au sorcier.

Celui-ci vous explique que cela ne peut signifier qu'une seule chose, vous êtes l'enfant de dragon et vous êtes maintenant capable d'utiliser la Voix, la langue des dragons, un pouvoir très puissant. Le mot que vous avez appris est "*Joor Zah Frul*" et il a pour effet de forcer les dragons à se poser au sol et ainsi pouvoir les tuer.

Pendant la discussion avec le sorcier un garde vous interrompt pour vous informer qu'un dragon attaque la tour ouest de Blancherive, vous partez alors au combat avec un petit bataillon et le sorcier combattre le dragon.

Une fois vaincu, sa carcasse prend feu et son âme s'échappe, aspiré par votre corps, Bordeciel est sauvé.

<table>
<tr><td colspan="2"></td><td></td><td></td><td>Chambre Jarl (up)</td><td>Fort-Dragon</td><td></td></tr>
<tr><td colspan="2">Tombeau Boss Stèle ↓</td><td></td><td></td><td></td><td>Centre Ville</td><td>Taverne</td></tr>
<tr><td>Salle Énigme</td><td rowspan="3">Halle du Tertre</td><td></td><td></td><td>Tour West</td><td>Fermes</td><td></td></tr>
<tr><td>Salle Draugr</td><td>Ruines du Tertre</td><td></td><td></td><td rowspan="2">Chemin (loup)</td><td></td></tr>
<tr><td>Salle Coffre</td><td colspan="2">Fort de Bandit</td><td>Pont</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>Rivebois</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>Chemin</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>Sortie Grotte</td><td></td><td></td></tr>
</table>

Items : Tablette ancienne, Hache du Jarl, Poupée, Baies, Cookie magique, Beamer, *Armes*, *Armures*,

Personnages :

Jarl (Balgruuf le Grand), Sorcier (Farengar), Gardes, Loup, Draugr, Dragon

Ralof, Gerdur,

Marchand de Blancherive, Forgeron de Blancherive

Forgeron de Rivebois (Alvor), Marchand de Rivebois (Lucan Valerius), Taverne Blancherive

Gagner : Tuer le Dragon

Perdre : Mourir (HP à 0)

1 énigme à définir

Système de combat, probablement avec des points de vie, lancer de dés et affinité avec certains dés en fonction des items porté par le joueur.

Explication exercice

7.5 (printLocationInfo)

On remarque que les méthodes `printWelcome()` et `goRoom()` utilisent les mêmes instructions pour indiquer à l'utilisateur où il se trouve, c'est une duplication de code.
Pour l'éviter on crée la méthode `printLocationInfo()` qui aura pour rôle de se faire appeler par d'autres méthodes à chaque fois qie l'on aura besoin d'afficher les informations sur la salle courante.

7.6 (getExit)

Actuellement la classe `Game` accède aux sorties d'une salle directement par les attributs publics de la classe `Room`, ce qui renforce le couplage entre ces deux classes. Si nous voulons ajouter des directions supplémentaires pour se déplacer dans le jeu, il nous faudrait modifier le code à plusieurs endroits. Afin d'éviter ceci nous allons doter la classe `Room` d'un accesseur avec un paramètre qui retournera la direction voulue par l'utilisateur : `getExit(String pDirection)`. Dans `goRoom` on n'utilise maintenant plus qu'une seule ligne pour changer de salle via l'utilisation de l'accesseur, ceci indépendamment du nombre de directions.

7.7 (getExitString)

De la même manière que dans la question précédente, nous créons une méthode `getExitString()` dans la classe `Room`. Elle aura pour but de fournir à d'autres méthodes les informations concernant les sorties des salles.
C'est `printLocationInfo()` qui se chargera de les afficher.
Ainsi si de nouvelles directions de sorties sont ajoutées nous n'aurons pas besoin de modifier le code à plusieurs endroits.

7.8 (HashMap, setExit)

Pour pouvoir créer des sorties différentes que "north, south, east, west" pour les salles il nous faut remplacer les quatres attributs de la classe `Room` par une `HashMap`. Celle-ci fonctionne comme un dictionnaire, chaque valeur stockée (de même type, ici `Room`) est associée à une clé unique du type que l'on souhaite (de même type pour la `HashMap` encore, ici `String`). De cette manière nous pouvons ajouter autant de déplacements que voulus de manière très simple, elles correspondent aux clés de la `HashMap`. La salle qui correspond aux déplacements

est définie dans `setExit(pDirection, pRoom)` grâce à la méthode “`put`” du paquetage `java.util.HashMap`.
On adapte aussi le constructeur naturel et l’accesseur.

On peut noter que grâce aux modifications d’encapsulation apportées précédemment nous pouvons faire ce changement sans craindre d’affecter la classe `Game`, il nous faudra seulement modifier l’instanciation des salles.

7.9 (keySet)

`keySet()` est une méthode qui retourne un `Set` (une “sorte de tableau” qui a toutes les propriété d’un ensemble en mathématique) de toutes les clé différentes utilisées par une `HashMap`;

7.10 (getExitString CCM?)

Fonctionnement de la méthode `getExitString()` :
Cette méthode est une fonction qui renvoie une `String` contenant l’ensemble de sorties possibles pour une salle.
On commence par créer la variable `vExits` qui est une `String` contenant la base du message à retourner : "Exits : ".
Ensuite, on crée un `Set` (un ensemble) de `String`, nommé `vKeys` à qui on affecte toutes les différentes sorties disponibles de la salle courante qui sont les clés de la `HashMap aExits`, ceci grâce à l’expression `this.aExits.keySet()`.
Puis on entre dans une boucle `for each`, qui pour chaque `String` contenu dans le `Set vKeys` que nous venons de créer, va concaténer cette `String` à `vExits`.
Une fois tous les éléments parcourus, `vExits` contient maintenant la base de départ ainsi que toutes les sorties disponibles qui peuvent être retournées.

7.10.2 (javadoc)

La classe `Game` contient moins de méthodes dans sa description que `Room` car elle sont toutes privées sauf `play()`. Contrairement à `Room` où elles sont toutes publiques.

7.11 (getLongDescription)

Afin d’encore réduire le couplage entre `Game` et `Room` et en prévision de modification future telle que l'ajout d’objet, de personnages et de monstres dans les pièces. Nous

ajoutons la méthode `getLongDescription()` dans `Room` qui aura pour rôle de fournir une description de tout ce qui se trouve dans la pièce.

7.14 (look)

Nous ajoutons la commande `look`, qui permet de réafficher la description et les sorties d'une salle. Ce rajout peut être effectué facilement grâce à la création préalable de la classe `CommandWords` qui vérifie si une commande existe et réduit le couplage implicite entre les classes.

Nous avons juste à rajouter le mot "`look`" dans le tableau des commandes valides puis à créer la méthode dans game et mettre cette possibilité de commande dans `processCommand()` et c'est bon.

Si après `look`, l'utilisateur rentre un 2e mot, nous lui indiquons qu'il n'y a rien à regarder en particulier pour le moment.

7.15 (eat)

On crée la commande `eat`, elle indique au joueur qu'il a mangé. Comme pour `look`, nous avons juste à rajouter le mot dans la liste des commandes valide puis créer la méthode dans `Game` et enfin l'ajouter comme option possible dans `processCommand()`.

7.16 (showAll, showCommands)

Après avoir ajouté les commandes `look` et `eat`, quand nous tapons "help", si nous ne les avons pas ajoutées manuellement, le programme ne nous propose pas ces deux actions. Cependant, faire cela à la main à chaque fois qu'une nouvelle commande est ajoutée requiert de la maintenance supplémentaire. Afin d'éviter cela nous ajoutons la méthode `showAll()` dans `CommandWords` qui à l'aide d'une boucle `for each` affiche toutes les commandes disponibles à l'utilisateur. La méthode `printHelp()` dans `Game` doit accéder à ces informations, pour ne pas renforcer le couplage entre les classes, nous passons par la classe `Parser` qui, elle, a déjà accès à la classe `CommandWords`. Nous créons donc dans Parser la méthode `showCommands()` qui a pour simple but d'appeler `showAll()`. Nous pouvons maintenant appeler `showCommands()` sur l'attribut de type `Parser` de la classe `Game`.

7.18 (getCommandList)

Dans le but d'attribuer à chaque classe des actions précises et pour anticiper les changements futurs nous changeons la méthode `showAll()` de classe `Room` pour

la remplacer par un accesseur `getCommandList()`. Celle-ci aura pour but d'être appelée par d'autres méthodes et de retourner une String de toutes les commandes que peut effectuer l'utilisateur. Il nous faut donc aussi changer les méthodes `showCommandsList()` dans `Parser` et `printHelp()` dans `Game`.

7.18.3 Images extraites du jeu Skyrim

7.18.5

Dans le but de rendre accessibles les `Room` du jeu aux autres méthodes et classes, nous créons un nouvel attribut `ArrayList<Room> aRooms` dans `GameEngine` et y ajoutons toutes les `Room`, dans `createRooms()`, après les avoir créées. Nous créons donc l'accesseur `getRoom(pIndex)` qui retourne la Room associée au numéro passé en paramètre.

7.18.6 Évolution du projet en Zuul with images

7.18.8 et **7.18.7** (bouton)

Pour pouvoir utiliser un bouton il faut importer `javax.swing.JButton` puis déclarer un attribut de type JButton. Ensuite, dans `createGUI()`, nous initialisons cet attribut puis lui assignons une place dans l'interface et enfin nous le relions à un `ActionListener`, interface qui appelle la méthode `actionPerformed()` lorsqu'une action est faite sur l'objet en question, grâce à `addActionListener()`:

```
this.aButtonHelp = new JButton("Help")
vPanel.add(this.aButtonHelp, BorderLayout.EAST);
this.aButtonHelp.addActionListener(this);
```

Pour finir, dans `actionPerformed()`, qui est une méthode automatiquement appelée en cas d'évènement sur les objets auquel nous avons ajouté un `ActionListener`, nous appelons sur l'attribut de type `GameEngine`, `aEngine` la méthode `interpretCommand()` avec comme paramètre le mot de commande associé. Ceci en vérifiant auparavant que l'objet source de l'appel de `actionPerformed()` correspond bien au bouton voulu grâce à `getSource()` qui retourne la référence vers l'objet qui a généré l'événement :

```
if(pE.getSource() == this.aButtonHelp)
this.aEngine.interpretCommand("help");
```

`actionPerformed()` est une méthode de l'interface `ActionListener` qui a un paramètre de type `ActionEvent` qui représente un événement "significatif" comme par exemple : cliquer sur un bouton.

Une autre solution aurait été d'utiliser `getActionCommand()` qui retourne la `String` correspondant au texte associé grâce à `setActionCommand()` de la manière suivante :

```
if(pE.getActionCommand() == "help")
this.aEngine.interpretCommand(pE.getActionCommand());
```

7.20 (Item)

Ajout d'une classe `Item` au jeu dans le but de permettre aux salles de contenir un objet. Cette classe contient les attributs `aNom`, `aDescription`, `aPoids` et `aPrix`, un constructeur naturel est les accesseurs associés.

Pour qu'un objet soit associé à une salle, il a fallu rajouter un attribut `aItem` à `Room`, mais toutes les salles n'ont pas d'objet donc il n'est pas nécessaire de le rajouter au constructeur. Pour relier un objet à une salle nous créons la méthode `setItem()` qui prend en paramètre l'objet qu'on veut placer et on l'assigne à l'attribut. L'`Item` doit préalablement avoir été créé dans `GameEngine`, de la même manière que nous créons les salles.

L'utilisation de `setItem()` se fait dans la classe `GameEngine` en même temps que d'assigner aux salles leurs sorties.

Pour avoir accès à la description de l'objet d'une pièce nous créons une méthode `getItemString()` qui retourne la chaîne de caractères adéquate selon si la pièce courante possède un objet ou pas. Puis nous ajoutons cette méthode dans `getLongDescription()` pour que la description de l'objet soit affichée quand nous entrons dans la pièce.

7.21(item description)

Ajout d'une méthode `getLongDescription()` dans `Item` qui retourne une chaîne de caractères contenant le nom, le poids, le prix et la description d'un objet de la manière suivante :

"- *Description de l'objet…*

Nom (Poids : *X* kg, Valeur : *X* Septim)"

7.22 (items)

Pour qu'une salle puisse avoir plusieurs objets, on remplace l'attribut de type `Item` par une `HashMap<String, Item>`, où la `String` est le nom de l'`Item` qui lui correspond. On remplace aussi la méthode `setItem()` par `addItem()` qui ajoute l'`Item` passé en paramètre à la `HashMap`.

On modifie aussi `getItemString()` pour qu'elle retourne la chaîne de tous les objets d'une salle les uns à la suite des autres.

7.23 (back)

On souhaite ajouter une commande "back" qui permet de retourner dans la salle précédente. Pour cela on ajoute un attribut de type `Room` : `aPreviousRoom` à la classe `GameEngine`, cet attribut est initialisé la première fois qu'on tape "go *direction*" et prend la valeur de la salle actuelle juste avant que celle-ci soit modifiée. Puis l'attribut est modifié à chaque changement de salle.

On crée ensuite la méthode `back()` qui vérifie s' il existe une salle précédente (au cas où on utilise "back" dans la salle du début); si c'est le cas on attribue à une variable la salle actuelle, puis on modifie la salle actuelle par la salle précédente et enfin la valeur de la salle précédente prend la valeur de la variable qui contient maintenant la nouvelle salle précédente.
Si on tape un deuxième mot après "back", le programme ne réagit pas et fait comme s'il n'avait pas été entré.

```
private void back()
{
    if(this.aPreviousRooms != null){
        Room vCurrentRoom = this.aCurrentRoom;
        this.aCurrentRoom = this.aPreviousRoom;
        this.aPreviousRoom = vCurrentRoom;
        printLocationInfo();
        this.aGui.showImage(this.aCurrentRoom.getImageName());
    }else
        this.aGui.println("Il n'y a pas de salle précédente");
}
```

Avec cette méthode back(), si nous tapons la commande "back" plusieurs fois, nous ne faisons que des allers-retours entre deux salles.

7.26 (Stack)

Cet exercice a pour but d'améliorer la méthode `back()` pour qu'elle permette de retourner plusieurs salles en arrière jusqu'à la 1ère salle. Pour cela on utilise une `collection` de type `Stack` qui correspond à une pile d'objets, seul l'élément au-dessus de la pile est accessible. La méthode `push()` de sa classe permet d'ajouter un élément au-dessus de la pile, on l'utilise dans `goRoom()` pour y ajouter la salle courante juste avant qu'elle ne change. Dans la méthode `back()`, on

remplace la vérification de la présence d'une salle précédente par une vérification de si la pile est vide ou non grâce à la méthode `empty()`. Puis on remplace la création d'une variable contenant la salle courante avant sa modification et la nouvelle attribution de `aPreviousRoom` par la simple méthode `pop()` qui retourne l'élément le plus haut de la pile tout en le retirant.

```
if(!this.aPreviousRooms.empty()){
     this.aCurrentRoom = this.aPreviousRooms.pop();
     printLocationInfo();
     this.aGui.showImage(this.aCurrentRoom.getImageName());
} else this.aGui.println("Il n'y a pas de salle précédente.");
```

Il existe aussi la méthode `peek()` qui retourne l'élément en haut de la pile mais sans le retirer.

7.26.1 (2 javadoc)

7.28.1 (automatisation des tests)

Dans le processus de réingénierie du code, avant de modifier notre jeu avec la nouvelle classe `Player`, nous devons mettre en place un moyen de tester les commandes de notre programme afin de s'assurer qu'après les modifications aucune fonctionnalité n'est été perdue. Pour cela nous créons la méthode `test()` dans `GameEngine` qui aura pour rôle de lire un fichier contenant toutes les commandes que nous voulons tester et de les exécuter.

La méthode `test()` possède un paramètre de type `Command` lui permettant de supporter un second mot qui servira à déterminer quel fichier exécuter.

Dans le corps de la méthode, après avoir testé la présence d'un second mot, nous initialisons la `String pNomFichier` avec celui-ci.

Puis nous entrons dans un corps de `try` / `catch`, cela permet d'exécuter du code dans la partie `try{}` en prévoyant qu'une erreur peut se produire, et y réagir de manière adéquate en exécutant alors le code contenu dans la partie `catch{}` sans stopper l'exécution du programme.

En effet nous voulons attribuer à une variable `vSc` de type `Scanner` le contenu d'un fichier, pour cela, en paramètre du constructeur du Scanner nous instancions le fichier voulu de la manière suivante : `new File(pNomFichier)`, cependant, ce fichier peut ne pas exister, dans ce cas une erreur `FileNotFoundException` ce produit et l'exécution du code passe directement à la partie `catch{}` qui nous informe que le fichier à tester n'a pas été trouvé.

S'il n'y a pas d'erreur, pour la variable `vSc.hasNextLine()` nous attribuons à une variable `String vLigne` cette `vSc.nextLine()` puis nous l'exécutons.

```
private void test(final Command pCommande)
{
     if(!pCommande.hasSecondWord()){
          this.aGui.println("Tester quelle fichier ?");
          return;
     }

     String pNomFichier = pCommande.getSecondWord();
     try{
          Scanner vSc = new Scanner(new File(pNomFichier));
          while(vSc.hasNextLine()){
               String vLigne = vSc.nextLine();
               this.interpretCommand(vLigne);
          }
     }// try
     catch(final FileNotFoundException pFNFE){
          this.aGui.println("Fichier à tester non trouvé.");
     } // catch
}
```

7.28.2 (3 tests)

Le test “court” teste les commandes suivantes : help, look et eat puis avance dans le jeu jusqu’à avoir bougé au moins une fois dans les quatres directions classiques puis fait des back jusqu’à revenir à la 1ère salle.
Le test “ideal” fait uniquement le chemin le plus court possible de toutes les étapes du jeu.
Le test “all” teste toutes les commandes du jeu et fait aller le personnage dans tous les lieux possibles.

7.29 (Player)

Dans le but de permettre à notre personnage de, entre autres, porter des objets dans son inventaire et de respecter la conception dirigée par les responsabilités, nous créons la classe `Player`. Celle-ci contiendra sous la forme d'attributs toutes les informations relatives au personnage pour l’instant : son nom, son poids, le poids maximal qu’il peut porter, sa position ainsi que les lieux précédemment visités et plus tard notamment son inventaire. Son constructeur prend en paramètre uniquement son nom qui pourra être demandé par l’utilisateur. Nous écrivons ensuite tous les accesseurs et modificateurs nécessaires.

Pour faire respecter la conception dirigée par les responsabilités nous devons faire en sorte que ce soit la classe `Player` qui s'occupe de modifier les informations concernant le joueur, donc pour l'instant sa position et l'historique de ses positions. Nous créons donc les méthodes `move()` et `moveBack()` pour remplir ces rôles qui seront appelés dans `goRoom()` et `back()`.

Ainsi, dans la classe `GameEngine` qui ne contient plus la position du personnage, nous devons maintenant utiliser les accesseurs et modificateurs du nouvel attribut `aPlayer`.

7.30 (take, drop)

Pour permettre à notre personnage de transporter et déplacer des objets, nous ajoutons un attribut de type `Item` à la classe `Player` avec un accesseur normal et modificateur sous la forme de la méthode `takeItem()`. Ce constructeur est sous la forme d'une fonction qui vérifie si le poids maximal n'est pas dépassé lorsque l'on essaie de prendre l'objet et qui retourne un `boolean` en fonction de si l'objet a bien été pris ou pas et modifie le poids du joueur. Cette fonction pourra ainsi être utilisée avec un `if else` dans `take()` pour donner la bonne information à l'utilisateur et selon si l'objet a bien été ramassé au pas.

Nous ajoutons aussi la commande `dropItem()` qui modifie l'attribut pour le rendre nul et le poids du joueur, ces deux méthodes seront appelées par les méthodes `take()` et `drop()` dans la classe `GameEngine` et auront pour paramètre le nom de l'objet avec lequel on souhaite interagir. Il ne faut pas oublier d'ajouter les mots de commande "take" et "drop" dans la classe `CommandWords` et de les placer dans la méthode `interpretCommand()`.

Ces deux méthodes fonctionnent de manière similaire, après avoir vérifié que la commande reçue en paramètre a bien un second mot, il est récupéré, puis l'objet associé grâce à la `HashMap` via accesseur créé dans `Room`. S'il n'est pas nul, dans le cas de `take()`, il est donné à l'attribut de `Player` grâce à `takeItem()` et est retiré du lieu par l'appel d'une nouvelle méthode `removeItem()` de `Room`. Dans le cas de `drop()`, l'objet est donné à la salle via la méthode `addItem()` puis il est retiré de l'inventaire grâce à `dropItem()`.

7.31 (porter plusieurs items)

Pour permettre au personnage de transporter plusieurs objets, il suffit de changer le simple attribut `Item` en une `HashMap` d'`Item` avec comme clé leur nom, en n'oubliant pas de l'initialiser dans le constructeur. Puis de rajouter un paramètre de type `String` aux méthodes `takeItem()` et `dropItem()` pour pouvoir identifier les objets dans la `HashMap` et les rajouter dans leur appel dans `GameEngine`.

Un attribut boolean aToujoursDeplacable est rajouté à Item pour permettre le test de la commande take() dans la 1ère pièce, la méthode takeItem() de Player est modifiée en conséquence en ajoutant un "ou logique" au test.

7.31.1 (ItemList)

Afin de centraliser la gestion objets et d'éviter la duplication de code, nous créons le classe `ItemList` qui a pour seul attribut une `HashMap<String, Item>` et qui aura pour rôle la gestion des objets de toutes les autres instances du jeu. Nous y plaçons un accesseur et deux modificateurs, un pour ajouter des `Item` à la `HashMap` et l'autre pour en retirer. Il nous faut donc modifier les classes `Room` et `Player` pour que leur attribut qui contient leurs objets ne soit plus des `HashMap` mais des `ItemList` et donc utiliser les modificateurs `addItem()` et `removeItem()` de `ItemList` quand on veut ajouter ou retirer un objet.

7.32 (poids max)

Fonctionnalité déjà implémentée en même temps que 7.30 mais ajout d'une méthode boolean `assezLege()` utilisée dans `takeItem()`de `Player` pour séparer les tâches des méthodes.

7.33 (inventaire)

Pour pouvoir fournir à `GameEngine` la liste des objets présents dans l'inventaire du personnage et leur poids total nous créons les méthodes `getInventaireString()` et `getPoidsTotal()` dans `ItemList` qui retourne à l'aide de `Set` le nom des objets de l'inventaire et son poids. Nous passons par la classe `Player` avec la méthode `showInventaire()` qui joue le rôle d'intermédiaire et qui rassemble les deux informations et les met en forme.
Dans `GameEngine` nous ajoutons la méthode `printInventaire()` qui affiche simplement la `String` créée par `showInventaire()`. Tout en n' oubliant pas de rajouter la commande "inventaire" à `interpretCommand()` et dans `CommandWords`.

7.34 (magic cookie)

Ajout de la prise en compte d'un second mot à la commande "eat" pour sélectionner l'objet que l'on veut manger. Nous ajoutons aussi un attribut boolean aMangeable à la classe Item. Le fonctionnement de eat() est le suivant, si le second mot est nul, un message prévient l'utilisateur qu'il n'a plus faim et c'est tout, sinon une variable correspondant à l'objet appelé est créée, s'il est nul, l'utilisateur est prévenu que l'objet n'est pas dans son inventaire et s'il n'est pas mangeable il est prévenu aussi.
Un dernier test intervient alors pour vérifier si c'est le cookie, si oui, le poids maximal du joueur est augmenté et l'objet est retiré de l'inventaire du joueur, un message pour prévenir l'utilisateur est aussi affiché.

7.34.1 (mise à jour tests)

Ajout des commandes “take”, “drop”, “inventaire” et “eat Cookie” au “test all”

7.34.2 (mise à jour javadoc)

7.42 (time limit)

Nous ajoutons l’attribut `int aTime` et créons la méthode `boolean time()` à la classe `GameEngine`. Cette méthode ajoute 1 à chaque appel de la méthode et vérifie si `aTime` dépasse 300, si oui l’utilisateur est prévenu que le temps est dépassé et la méthode `endGame()` est appelée puis la méthode retourne `true`. Si le temps n’est pas dépassé elle retourne `false`. La méthode `time()` est appelée à chaque fois qu’une commande valide est entrée.

7.42.2 (IHM graphique) Directement inspiré du projet de Baptiste Espinasse.

Remplacement de l’unique bouton Help sur le bord droit de l’interface par quatre boutons correspondant aux quatre directions principales du jeu : north east west south. Implémentation effectuée par l’ajout de cinq nouveaux attributs, quatre `JButton` et un `JPanel aButton` à la classe `UserIterface`. Nous créons aussi la méthode `makeBoutonBar()` qui a pour rôle de créer ce dernier `JPanel` avec les quatre boutons, notamment grâce à l’aide de la classe `GridLayout` qui permet de scinder l’espace disponible. Ce panel de boutons est ajouté au panel principal déjà présent dans `createGUI()` et est placé à droite de l’interface.

7.43 (trap door) Inspiré du projet de Baptiste Espinasse.

Incorporer un chemin à sens unique est simple, il suffit de n'assigner qu' une sortie réciproquement à deux salles.
Dans le but d’empêcher la méthode `back()` de faire passer le personnage dans des chemins censés être à sens unique, nous ajoutons une méthode boolean `isExit()` dans `Room`. Cette méthode retourne `true` si la salle précédente est accessible depuis la salle courante, ceci grâce à la méthode boolean containsValue() présente dans la classe HashMap. Cette méthode sera appelée dans l’exécution de la commande `moveback()` de `Player` que nous allons aussi modifier. Nous la transformons en fonction du déplacement du Player et qui retourne true si le déplacement a bien été effectué. Ainsi dans GameEngine nous pouvons faire un

`if(this.aPlayer.moveBack())` et si le Player a bien changé de salle, les informations sont affichées.

Nous avons rencontré un problème, lors du test de réciprocité des sorties, nous avons utilisé la méthode `getPreviousRooms()` qui utilisait `pop()` pour accéder à la Room, ce qui fait deux pop() d'un coup. Ceci avait pour effet de retirer la `Room` en haut du `Stack` deux fois et la commande `back()` retournait deux salles en arrière. Pour résoudre ce problème nous avons différencié getPreviousRooms() en getPopPreviousRooms() et getPeekPreviousRooms().

7.44 (beamer) Inspiré du projet de Baptiste Espinasse.

Un Beamer est un objet avec des caractéristiques spécial donc c'est une sorte d'objet. Nous créons notre première sous-classe : Beamer. Celle-ci a deux attributs, `boolean aCharge` et `Room aBeamerRoom` avec leur accesseur respectif, il est envisageable d'en rajouter un troisième : `boolean aModifiable` dans le but de créer des checkpoints représentés dans le jeu par les auberges des villes. Son constructeur naturel à deux paramètres : `pNom` et `pDescription`, l'attibut `aCharge` est pour l'instant tout le temps initialisé à `false`. Nous devons aussi créer deux modificateurs pour charger une `Room` dans le `Beamer` et pour le décharger.

Après ça nous pouvons créer les méthodes qui exécuteront les commandes "charge" et "teleport" : `charge()` et `teleport()`. La première partie de leur corp est la même, elles récupèrent le second mot de la commande pour sélectionner sur quel `Beamer` faire l'action, puis créent une variable `vBeamer` correspondant à celui appelé.

Puis dans le cas de `charge()`, si le `Beamer` appelé existe, celui-ci est chargé et son attribut reçoit la pièce où se trouve le personnage et l'utilisateur est notifié, sinon il est prévenu que le `Beamer` sélectionné n'est pas dans l'inventaire.

Pour `teleport()`, si le `Beamer` existe et qu'il est chargé, la méthode `move(vBeamer.getBeamerRoom())` est appelée sur `aPlayer` puis le `Beamer` est déchargé et l'utilisateur en est informé. Les informations concernant le nouveau lieu et son image sont ensuite affichées mais puisque c'est la troisième fois, nous utilisons les lignes :

```
this.printLocationInfo();
this.aGui.showImage(this.aPlayer.getCurrentRoom().getImageName());
```

Pour éviter la répétition de code nous créons la nouvelle méthode `majInfosRoom()` qui contiendra ces deux lignes et qui sera appelée à chaque fois que le personnage changera de salle.

On fini par rajouter ces deux méthodes dans `interpretCommand()`.

7.45 (optionnel) (locked door)

À faire plus tard.

7.45.1 (mise à jour tests)

Ajout des commandes
“take Beamer”, “charge Beamer”, “teleport Beamer”,
“look rien”, “eat rien”, “take rien”,
“charge rien”, “teleport rien”
au “test all”

7.45.2 (mise à jour javadoc)

7.46 (transporter room)

Dans le but de créer des salles spéciales qui nous téléportent à un endroit aléatoire du jeu quand on en sort, nous créons une sous-classe `TransporterRoom`. Cette sous-classe aura deux attributs: un `RoomRandomizer aRandomRoom` et une `ArrayList<Room> aRooms` contenant toutes les `Room`s du jeu et qui sera initialisé par le constructeur en plus de créer une salle comme d'habitude.
`RoomRandomizer` est une nouvelle classe qui a pour unique rôle de contenir un attribut générateur de nombre aléatoire et de le changer à chaque fois qu’on y accède. Ceci est fait grâce à la classe importée `Random` et la méthode `nextInt(int)` dans l'accesseur `getRandomNumber()` qui fait prendre à un objet `Random` un entier aléatoire selon une loi uniforme entre 0 et l’entier passé en paramètre.
`TransporterRoom` a pour rôle de remplacer la méthode `getExit()` de `Room` affin de retourner une `Room` aléatoire à la place. Pour cela, elle fait appel à la méthode `findRandomRoom()` qui retourne une salle aléatoire de la manière suivante :
`return this.aRooms.get(this.aRandomRoom.getRandomNumber());`
La `TransporterRoom` du jeu est une nouvelle salle créée pour l’occasion, un Taverne à Blancherive.

Concept à expliquer : seed :
Une seed est un code unique associé à une chose précise. Dans notre cas ici, une seed représente une suite de nombres aléatoires, aussi si deux objets Random ont le même seed, alors ils sortiront la même suite de nombre aléatoire.

7.46.1 (commande alea)

En raison du caractère aléatoire des `TransporterRoom`, nous devons trouver un moyen d'annuler ce caractère lors de la lecture des fichiers des tests. Pour cela nous créons la nouvelle commande "alea" qui, si elle contient une deuxième `String` (ici un nombre correspondant à leur place dans la `ArrayList`) l'assigne au nouvel attribut `String aRoomChoisie` de `TransporterRoom`, sinon elle rend cet attribut `null`.
Du côté de `TransporterRoom`, nous ajoutons le modificateur `setRoomChoisie(pNumRoom)` mais cet attribut est initialisé à `null` en temps normal. Il faut aussi rajouter une ligne dans sa méthode `getExit()` pour qu'elle retourne la `Room` choisie si son nouvel attribut n'est pas null :

```
if(this.aRoomChoisie != null){
	return this.aRooms.get(Integer.parseInt(this.aRoomChoisie));}
```

La commande "alea" fonctionne de la manière suivante :

```
private void alea(final Command pCommande)
{
	if(!this.aTestMode){
		this.aGui.println("I don't know what you mean...");
		return;
	}
	String vNumRoom = pCommande.getSecondWord();
	TransporterRoom vTransporterRoom = (TransporterRoom)this.getRoom(12);

	if(!pCommande.hasSecondWord()){
		vTransporterRoom.setRoomChoisie(null);
		this.aGui.println("Lieu supprimé, prochain tirage
		aléatoire." + "\n");
		return;
	}
	else if(Integer.parseInt(vNumRoom) >= this.aRooms.size()){
		this.aGui.println("Nombre trop haut." + "\n");}
	else vTransporterRoom.setRoomChoisie(vNumRoom);
}
```

Affin que cette méthode ne soit accessible que durant l'exécution d'un fichier de test, nous ajoutons un attribut `boolean aTestMode` à `GameEngine` initialisé à `false` par le constructeur mais mis à `true` au début de la commande `test()` puis remis à `false` à la fin.

Ajout de ces commandes au fichier "test all".

7.46.3 (commentaires javadoc)

7.46.4 (générer les javadoc)

7.47 (abstract Command)

Dans le but d'améliorer la conception de l'exécution des commandes, nous allons faire les changements nécessaires pour remplacer la longue suite de `if` dans la méthode `interpretCommand()` et les méthodes des commandes dans `GameEngine` par une simple instruction grâce à une conception par polymorphisme via la classe `Command`.

Nous allons rendre cette classe abstraite avec comme méthode abstraite `execute(Player)` ce qui nous permet de créer des sous-classes pour chaque commande que nous voulons implémenter, elles auront pour méthode principale `execute(Player)` qui pourra être appelée sur un objet `Command` dans `interpretCommand()`.

La nouvelle classe `Command` a trois attributs, `String aSecondWord`, `static UserInterface aGui` et `static boolean aTestMode`. `aSecondWord`, elle était déjà présente et ne change pas, les deux autres sont initialement présentes dans `GameEngine` mais sont placées en `static` car certaines commandes doivent y accéder, celles-ci sont initialisées dans le constructeur de `GameEngine`.

Nous créons ensuite toutes les sous- classes nécessaires et plaçons le corps des méthodes de commandes de `GameEngine` dans celui de la méthode `execute(Player)` de chaque classe et utilisons son paramètre à la place de l'attribut `aPlayer` de `GameEngine`. Elles accèdent à l'interface grâce à l'attribut `static aGui` de `Command` et son accesseur.

Les seules qui nécessitent d'autres changements sont les classes `HelpCommand`, `AleaCommand` et `TestCommand`. Pour `HelpCommand` il faut juste ajouter un attribut de type `CommandWords` pour avoir accès à la liste des mots des commandes possibles. `AleaCommand` a besoin d'un accès à la `ArrayList` des `Room`s, elle lui est fournie par un attribut `static`, et `TestCommand` a besoin de la méthode `interpretCommand()` de `GameEngine` qui lui est fournie de la même manière.

Nous en profitons pour ajouter à la commande "go" une différenciation entre "Direction inconnue." et "Il n'y a pas de chemin." en ajoutant un attribut `ArrayList<String>` à `GoCommand` contenant toutes les directions valides et en effectuant un test juste après avoir récupéré le second mot pour vérifier qu'il est contenu dans la `ArrayList`.

Ensuite, afin d'associer chaque mot de commande à une `Command`, nous créons une `HashMap<String, Command>` dans `CommandWords`, elle est initialisée dans son constructeur que nous créons pour l'occasion, voici un exemple avec "go" :

```
aCommands.put("go", new GoCommand());
```

Nous pouvons aussi supprimer la méthode `isCommand()` qui ne nous servira plus car la méthode `get()` de `HashMap` retourne `null` si la clé entrée n'est pas reconnue.

L'avant-dernière chose à faire est de modifier `Parser`, en effet maintenant que les commandes sont créées dans `CommandWord` et que la `Command` est `null` si le mot de commande n'existe pas nous pouvons simplement l'appeler de la manière suivante :

```
Command vCommand = aCommandWords.getCommand(vWord1);
```

vérifier qu'elle n'est pas `null` et lui attribuer le second mot dans ce cas, sinon la retourner.

Enfin, nous pouvons remplacer la longue chaîne de `if` dans `interpretCommand()` par le simple bout de code :

```
if(vCommand == null) this.aGui.println("Commande non reconnue.");
else vCommand.execute(this.aPlayer);
```

7.47.1 (paquetages)

Nous créons quatre paquetages :

pkg_engine avec les classes GameEngine, UserInterface et Player.

pkg_commands avec les classes Parser, CommandWords, et Command et toutes ses sous-classes .

pkg_rooms avec les classes Room, TransporterRoom et RoomRandomizer.

pkg_items avec les classes Item, ItemList et Beamer.

Seul Game reste à la racine du projet.

Nous importons donc dans chaque classe les paquetages nécessaires à leur fonctionnement.

7.47.2 (générer les javadoc avec paquetages)