Magic Book

Rapport de projet

DEROUIN Auréline 21806986
MARTIN Justine 21909920
THOMAS Maxime 21810751

Table des matières

1 Présentation du projet

A Présentation de l'application

Magic Book est un éditeur de livre permettant de créer un livre à choix multiples pouvant contenir des conditions pour certains d'entre eux, des choix aléatoires, des combats, etc.
On peut donc créer des paragraphes, appelés des "noeuds", reliés entre eux par des liens. L'application comprend aussi la création d'un prélude, de personnages et d'items.
Une fois le livre créé, nous pouvons alors obtenir une estimation de sa difficulté en choisissant l'option correspondante dans la barre de menu en haut. Cette difficulté est ensuite affichée dans le panel des stats.
Une option est également disponible pour permettre de jouer à l'histoire créée. Enfin, il est également possible d'exporter le livre dans un format texte.
Bien entendu, il est possible d'enregistrer notre livre afin de le réouvrir pour continuer l'édition de celui-ci.

FIGURE 1.1 – MagicBook

2 Organisation du projet

A Choix des technologies

i Git

Nous avons choisi d'utiliser Git comme logiciel de gestion de versions. Quelques-unes des raisons de ce choix sont listées ci-dessous :

- La gestion des branches est efficace
- Logiciel de gestion de versions décentralisées, une interruption de service d'un hébergeur n'empêche pas de continuer le travail et il est facile d'héberger son code sur une nouvelle plateforme
- Meilleure gestion des commits et des conflits que SVN

Voici quelques informations supplémentaires concernant notre utilisation de celui-ci.

Branches

Afin d'utiliser au mieux Git, nous avons fait le choix de créer deux branches "principales". Il s'agit de *master* et de *develop*.
La branche *master* correspond à une version stable qui peut être mise en production. Ainsi, on ne travaillera jamais sur cette branche.
La branche *develop*, quant à elle,est celle à partir de laquelle nous créerons les différentes branches pour le développement de nos fonctionnalités. Ne sont poussées sur celle-ci que les nouvelles fonctionnalités opérationnelles des applications. C'est donc la version en cours de développement.
Les branches créées à partir de *develop* sont les branches correspondant aux fonctionnalités développées, elles commencent toutes par *features/* (correspondant à la modification). Par exemple, pour le développement des fourmis, on créera une branche *features/fourmis*.

Nomenclature

Nous avons choisi d'établir et d'utiliser une nomenclature pour les messages de commit. Chaque message est préfixé par un mot qui permet d'identifier le type de modification apportée. Nous pouvons par exemple citer l'ajout de fonctionnalités sous le préfixe de *feat*, *fix* pour les corrections de bug, *doc* pour la documentation, etc.

ii Gradle

Gradle est un "build automation system". Il est un équivalent plus récent et plus complet à Maven. Il possède de meilleures performances, un bon support pour de nombreux IDE et permet d'utiliser de nombreux dépots, dont ceux de Maven, pour télécharger les dépendances dont le projet a besoin. Cet outil se révèle pratique car il automatise complètement la réalisation des tâches usuelles tel que la compilation, l'éxécution et les tests unitaires du code source, etc. Il est également possible de créer ses propres "tasks", afin d'automatiser des actions récurrentes, ou de concevoir et utiliser des plugins pour faciliter la configuration de certains projets (JavaFx11 et plus, Android, ...).

iii JavaFx

Il s'agit d'une technologie plus récente que Swing. De ce fait, beaucoup plus de composants modernes sont disponibles contrairement à Swing. Nous avons fait le choix d'utiliser cette technologie notamment pour élargir nos connaissances sur Java et les librairies usuelles.

B Gestion du projet

Afin de faciliter la communication et le bon déroulement de la conception de notre application, divers moyens ont été mis en oeuvre.

i GitHub et Forge

Bien que nous devions rendre le projet sur la forge, nous avons fait le choix d'utiliser GitHub afin d'héberger et de travailler sur le projet. Ce choix s'est fait au vue de la liste des avantages que cette plateforme apporte :

Webhooks : Ils permettent d'obtenir facilement toutes les informations sur ce qui se passe concernant le dépot. Cela est d'autant plus intéressant que Discord permet d'exploiter ces webhooks.

Pull Requests : Elles permettent de demander une fusion entre deux branches tout en visualisant toutes les mofifications effectuées depuis le dernier commit en commun. Cette fonctionnalité nous a notamment été utile pour effectuer les revues de code.

Actions : Il est possible d'exécuter certaines actions, par exemple, lorsqu'un évènement se déclenche. Nous avons utilisé cette fonctionnalité afin de lancer automatiquement les tests unitaires à chaque push et pull request. On était alors prévenu dès qu'ils échouaient.

De plus, grâce à git, il suffit simplement d'ajouter une remote vers la forge afin de push les changements sur celle-ci. Cela est d'autant plus pratique que l'entiereté des commits est conservé. Des pushs sur la Forge sont donc réalisés toutes les semaines afin d'actualiser le dépôt. Bien entendu un push final a été effectué sur la Forge pour rendre le projet.

ii Trello

Concernant la répartition et le "listing" du travail à produire, nous avons fait le choix d'utiliser Trello. C'est une plateforme qui nous permet d'utiliser des tableaux pour planifier un projet.

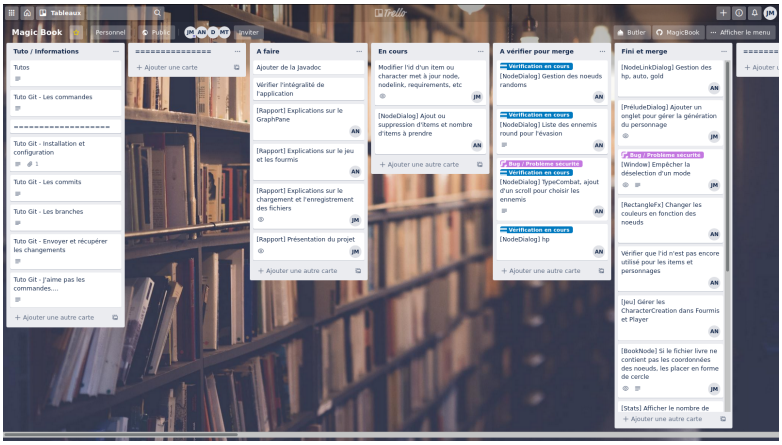

FIGURE 2.1 – Notre tableau Trello

Ainsi, comme nous pouvons le constater, les différentes tâches passent par différents états, "*A faire*", "*En cours*", "*A vérifier*", "*Fini et merge*". Enfin, bien que cela ne soit pas visible sur l'image 2.1, il existe un "*Backlog*" sur la droite qui contient les différentes tâches restantes à accomplir. Celles-ci peuvent ensuite être déplacées dans la colonne "*A faire*" au moment où nous jugeons qu'elles peuvent être réalisées.

Les colonnes "*A verifier*" et "*Fini et merge*" nécessitent quelques précisions. Pour la première, lorsqu'une tâche est terminée, elle est soumise à évaluation et relecture. Cela permet d'obtenir un avis sur la fonctionnalité et d'éviter d'éventuels bugs par la suite mais aussi de garder une cohérence au travers le code. Raisons pour lesquelles les personnes qui effectuent cette relecture sont souvent les mêmes. Enfin, quand celle-ci est vérifiée et validée, on peut alors merge la branche *feature* dans *develop* et ainsi, la déplacer dans la seconde colonne.

iii Discord

Afin de faciliter la communication au sein du groupe, nous utilisons le service de messagerie Discord car tous les membres du groupe l'utilisaient déjà de manière personnelle. Celui-ci permet de parler par le biais de "serveurs" gratuits dans lesquels nous pouvons ajouter des salons textuels ou des salons vocaux à volonté. Ainsi, nous avons trois salons de discussion. L'un nommé "*news-magic-book*", qui nous permet d'obtenir toutes les informations sur les push, pull-request, résultats des tests concernant le dépot sur GitHub. "*important-magic-book*" permet de transmettre des messages importants, sur ce qui a été fait, sur des changements, sur les dates limites concernant le projet, etc. Enfin, "*dev-magic-book*" est une discution beaucoup plus générale dans laquelle on peut demander de l'aide, aider des membres en difficulté, ou même de discuter de certains choix à faire.

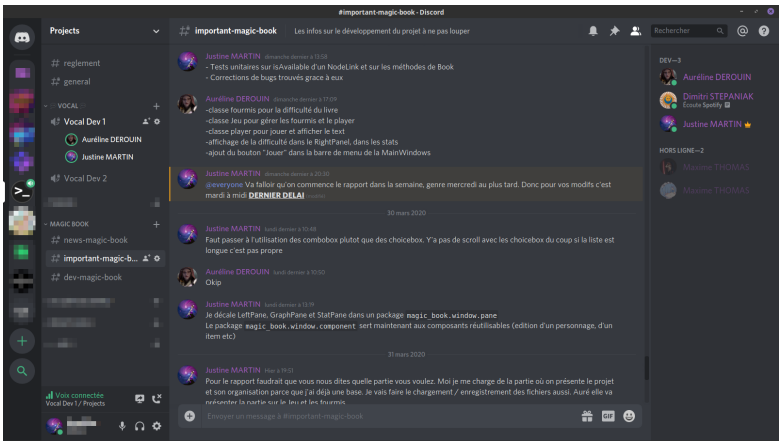

FIGURE 2.2 – Notre serveur Discord

3 Travail de groupe

A Répartition des tâches

Tâches effectués	Auréline	Dimitri	Justine	Maxime
Lecture et enregistrement des fichiers				
Classes pour parser le JSON			X	X
Lecture d'un fichier JSON			X	
Enregistrement d'un fichier JSON			X	
Livre				
Classe Book			X	
Classes pour représenter les noeuds et les liens	X		X	
Classe BookCharacter			X	X
Classes pour les prérequis (Requirement)	X		X	
Classes pour représenter les différents types d'items			X	
Classes pour la "Création du personnage"			X	
Classe pour représenter des skills			X	
Classes pour le pattern observer du livre			X	X
Jeu et export au format texte				
Classe Jeu, partie commune au joueur et la fourmis	X			
Classe pour la logique de la fourmis	X			
Classe pour la logique du joueur	X			
Permettre une estimation de la difficulté du livre	X			
Generation du livre en format texte			X	
Classe BookState			X	
Création d'un Parser pour le texte			X	
Version primitive de l'estimation de la difficulté d'un livre				X
Fenêtre				
Fenêtre principale		X	X	
Gérer la création d'un nouveau livre, l'ouverture d'un ancien livre, sauvegarde/sauvegarde-sous du livre courant			X	
Lister et permettre l'ajout d'items et de personnages sur le panel de gauche		X		
Permettre d'editer ou supprimer un item ou un personnage du livre			X	
Statistiques concernant les noeuds			X	
Statistique sur le niveau de difficulté du livre	X			
Cacher panel des statistiques si l'on décoche une case dans le menu			X	
Cacher le panel de gauche si l'on décoche une case dans le menu				X
Séparation des différentes parties de la fenêtre en plusieurs classes (LeftPane, GraphPane, RightPane)	X			

Tâches effectués	Auréline	Dimitri	Justine	Maxime
Composants réutilisables pour créer des personnages, créer une phase de la "Création du personnage", sélectionner une liste d'items			X	
Création d'une classe mère pour l'affichage des items/personnages/compétences			X	
Création d'une classe fille AbstractBookTreeView			X	
Boites de dialogue				
Classe mère pour les boites de dialogue	X			
Boite de dialogue pour les noeuds	X			
Boite de dialogue pour les liens entre les noeuds	X			
Boite de dialogue pour les items	X			
Boite de dialogue pour les personnages			X	
Boite de dialogue pour les compétences			X	
Boite de dialogue pour le prélude			X	
Boite de dialogue pour la "Création du personnage"			X	
Boite de dialogue pour le personnage par défaut			X	
Zone d'édition				
Classe pour représenter un noeud graphique	X			
Ajout d'un noeud	X			
Modification d'un noeud	X			
Suppression d'un noeud	X			
Classe pour représenter un lien entre 2 noeuds			X	
Ajout d'un lien entre 2 noeuds			X	
Un lien suit les noeuds auxquelles il est attaché			X	
Modification d'un lien entre 2 noeuds	X			
Suppression d'un lien entre 2 noeuds	X			
Un lien suit les noeuds auxquelles il est attaché			X	
Classe mère commune pour représenter un prélude et un noeud (RectangleFx)	X		X	
Permettre le déplacement des noeuds	X			
Détecter un clique sur un noeud ou un lien (classes observer)	X		X	
Gestion des actions en fonction du mode	X			
Afficher un rectangle qui représentera le prélude			X	
Gestion du texte de prélude			X	
Gestion du personnage par défaut			X	
Gestion de la "Conception du personnage"			X	
Ajout du shop dans la "Conception du personnage"	X			
Changer le premier noeud du livre			X	
Répartition des différents noeuds lors de l'ouverture d'un fichier			X	
Gestion du niveau de zoom			X	
Rend le GraphPane scrollable			X	
Change la couleur d'un noeud en fonction de son type (normal, aléatoire, combat, victoire, ...)	X			
Mettre en valeur un noeud lorsque l'on passe la souris dessus	X			
Gestion des prérequis dans les lien	X			

Tâches effectués	Auréline	Dimitri	Justine	Maxime
Gestion des shops dans les noeuds	X			
Ajout d'un scroll si trop de personnages dans les noeuds de Combat			X	
Autre				
Rapport	X		X	~
Diapositives de présentation			X	
Texte de présentation diapositives			X	
Documentation Utilisateur	X			
Restructuration du livre d'exemple (fotw.json)			X	
Création de tests unitaires	X		X	
Javadoc	X		X	
Revue de code avant de merge			X	

Nous avons décidé de ne pas inclure le graphique de Forge concernant le nombre de lignes de code commités par personne. En effet, l'ajout de Gradle et du livre d'exemple font à eux seul 10 000 lignes. De plus, ce livre a été restructuré. De ce fait, après vérification, 17 150 lignes ajoutées, et 10 460 lignes supprimées sont données à la personne qui les a commit, Justine dans notre cas.

Après avoir lancé un script (git-stats) pour connaitre le nombre de lignes par commit de chacun, voici ce que donnerait le graphique si on enlevait toutes ces lignes de Justine :

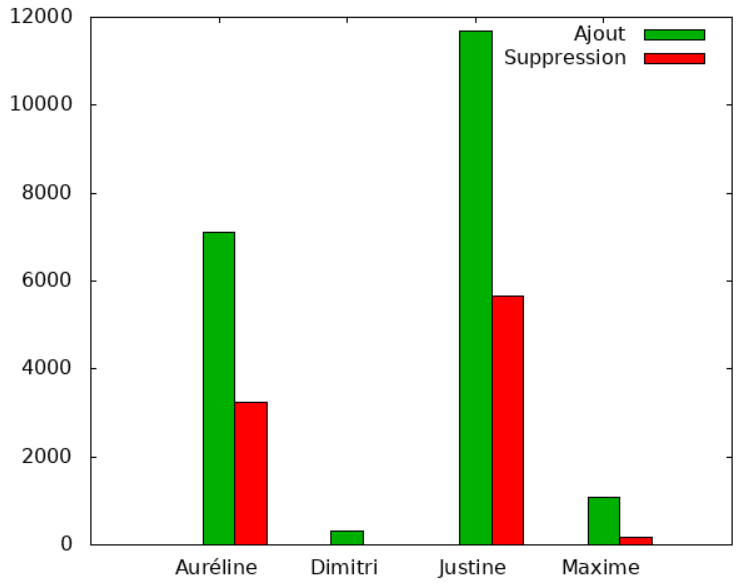

FIGURE 3.1 – Nombre de lignes ajoutés et supprimés par personne

B Idées d'améliorations

Notre application n'ayant pu être terminée faute de temps, voici la liste des améliorations que nous aurions voulu faire et celles qui seraient possibles d'implémenter ensuite :

- Concevoir deux types de fichier, l'un pour l'éditeur et l'autre pour le jeu. Le jeu serait une version épurée de celui de l'éditeur et ne contiendrait pas la position des noeuds, par exemple
- Une mise à jour d'un noeud qui transformerait/transférerait correctement les différents liens (au lieu de les supprimer dans la plupart des cas)

- Vérifier que le livre est valide pour être joué
- Créer une classe mère pour les listes (TreeView) sur le côté gauche de l'application (Item et personnage)
- Une fois la classe mère codée, ajouter une liste à gauche pour gérer les skills dans l'éditeur
- Déclencher plus d'exception si le livre est incorrect
- Gérer les shops (jeu et gui), champs auto (jeu uniquement)
- Afficher les personnages et items inutilisés
- Indiquer si l'estimation de la difficulté est à jour ou non
- Gestion des prérequis sur les boites de dialogue des liens
- Améliorer l'intelligence de la fourmi (pouvoir estimer si un item est plus important qu'un autre, meilleure gestion des combats, ...)
- Ajouter et supprimer des skills au fil du jeu
- Ajout de paramètres aux skills (plutôt que d'avoir un simple nom)
- Afficher les chemins gagnants
- Enlever ou ajouter une somme d'argent à un personnage se fait sur une monnaie précise (ex : -5 dollards, +15 euros, etc)
- Ajout d'un "Langage" simple permettant de manier des conditions et variables pour des prérequis notamment
- Possibilité d'avoir des pnj qui pourraient nous suivre dans l'aventure pour combattre ou pour dévérouiller certains passages par exemple.

C Bugs et problèmes connus

Certains problèmes sont connus, en voici une liste non exhaustive une fois de plus :

- Tests incomplets sur le Book et le Jeu
- Si un numéro de noeud est manquant à l'ouverture du fichier, des noeuds peuvent se faire écraser (cf : la liste des précisions dans La classe Book page 18)
- Le BookNodeCombat n'est pas du tout pratique à utiliser
- Le changement d'ID d'un personnage ou d'un item ne met pas à jour les différents élements du livre (noeuds, choix, ...)
- Diverses bugs visuels concernant la boite de dialogue sur le Prélude
- Le zoom ne se fait pas selon la position actuel de la souris mais du point supérieur gauche du GraphPane

4 Architecture du projet

A Arborescence du projet

.github : Fichiers spécifiques à GitHub.

workflows : Fichiers destinés au module d' "Actions" de GitHub. Nous nous en sommes servis pour lancer automatiquement les tests unitaires lors d'un push ou d'une pull-request.

app : Contient tout le code source de notre application.

gradle : Wrapper de gradle.

livre : Exemples de livre.

src : Contient les codes sources, ressources et tests unitaires.

main : Code principal de l'application.

java : Code source.

resources : Ressources pour l'application (images, ...).

test : Le code des tests unitaires.

java : Code source.

resources : Ressources utiles pour les tests uniquement (images, ...).

build.gradle : Script de configuration du projet (dépendance, classe principale, ...).

gradlew : Script pour les systèmes Unix afin d'exécuter le Wrapper de Gradle.

gradlew.bat : Script pour les systèmes DOS afin d'exécuter le Wrapper de Gradle.

settings.gradle : Configuration sur les modules à inclure, les noms de ceux-ci, etc.

.gitattributes : Permet de fixer la fin de ligne pour les scripts Unix et DOS.

doc : Contient toute la documentations du projet, notamment le rapport.

.gitignore : Fichier ignorant les changements sur certains fichiers ou dossier sur Git.

CONVENTIONS.md : Conventions de nommage concernant le projet et les commits.

LICENSE : Licence du projet.

README.md : README pour présenter notre projet et expliquer la compilation de celui-ci.

Pour mieux comprendre la structure de gradle les liens suivants sont utiles https://guides.gradle.org/creating-new-gradle-builds/ et https://docs.gradle.org/6.3/userguide/gradle_wrapper.html

B Présentation des packages

Notre application contenant beaucoup de classes, celles-ci sont réparties en packages que nous allons détailler :

core : Classes principales de l'application

exception : Classes d'exceptions

file : Classes utiles à la lecture, l'écriture de fichier (json et texte)

deserializer : Classes qui héritent de JsonDeserializer (provient de GSON)

json : Classes JSON intermédiaires pour la lecture et l'écriture avec GSON

- **game :** Classes spécifiques au jeu (Personnage, Skill, BookState, ...)
  - **character_creation :** Classes qui représentent une étape de la "Création du personnage"
  - **player :** Classes qui permettent de jouer au jeu (Joueur ou fourmis)
- **graph :** Classes qui représentent les noeuds et les liens
  - **node :** Classes pour les noeuds
  - **node_link :** Classes pour les liens
- **item :** Classes qui représentent les items
- **parser :** Classes qui permettent de parser un texte pour afficher le nom de l'item ou du personnage
- **requirement :** Classes pour gérer les prérequis sur un noeud

observer : Classes pour le pattern observer

- **book :** Classes pour le pattern observer du livre
- **fx :** Classes pour le pattern observer des éléments JavaFx

window : Classes pour l'affichage avec JavaFx

- **component :** Composants réutilisables à différents endroits (dans plusieurs boites de dialogues par exemple)
- **dialog :** Les différentes boites de dialogue
- **gui :** Les différents éléments graphiques pour JavaFx (NodeFx, NodeLinkFx, PreludeFx)
- **pane :** Les différentes parties qui composent notre affichage sur la fenêtre (Partie de gauche, centrale, droite)

5 Aspects techniques

A Représentation d'un livre

i Représentation des noeuds

Nous étions d'abord partie sur une forme basique de noeud. C'était un noeud tout simple qui comportait un paragraphe, un type et une liste de choix. Sauf que cela ne suffisait pas, car même si on définissait plusieurs **BookNodeType**, on ne pouvait pas implenter tout ce qu'on voulait. En effet, on devait penser autrement, c'est a dire, une classe par type. On pourait donc plus facilement gérer les différents lien, méthode propre au types de noeud ainsi que leurs différents valeurs notament, par exemple , au noeud de combat.

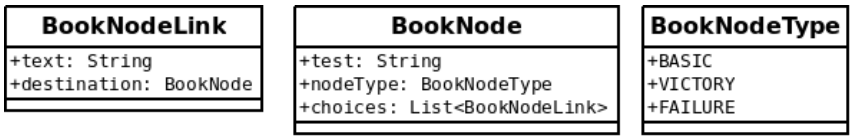

FIGURE 5.1 – Représentation des Node avant

Il y a donc, quatre types de noeuds représentées par des classes (Voir figure 5.2) : **BookNodeWithChoices**, **BookNodeWithRandomChoices**, **BookNodeCombat** et **BookNodeTerminal**. Toutes ces classes représentes les différents type de noeuds existants. Cela permet donc d'avoir différentes méthodes qui les représentes. Mais tous extends direct ou indirectement de **AbstractBookNode**. Cette dernière contient juste un texte correspondant aux paragraphes de chaque noeuds.

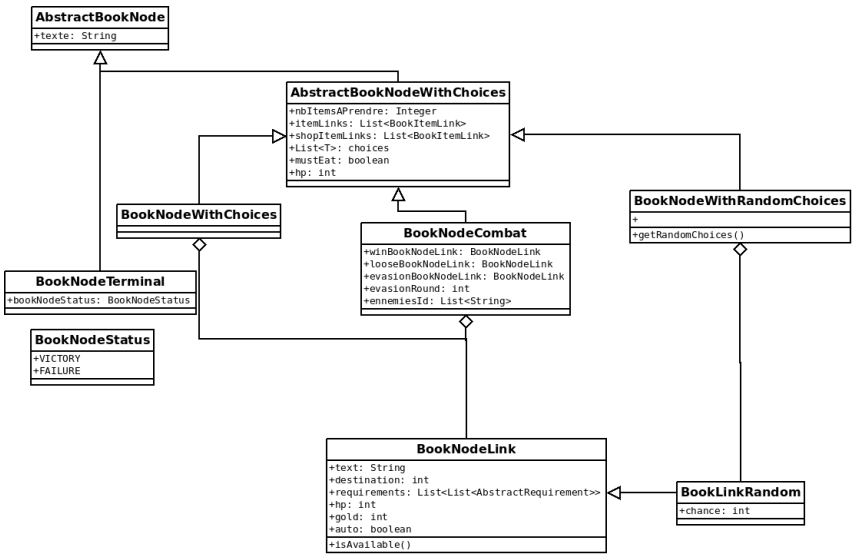

FIGURE 5.2 – BookNode

Pour le **BookNodeTerminal**, il prend en compte, en plus du texte de **AbstractBookNode**, une énumération de la classe **BookNodeStatus** : **FAILURE** ou **VICTORY**. C'est donc cette classe qui va déterminer la finalité du livre : si le joueur à gagné ou perdu.

Pour l'ensemble des autres noeuds, on retrouve une liste de choix et une perte/gain de vie. Pour cette

raison, elles extendent toutes d'une même classe, **AbstractBookNodeWithChoice**, permettant ainsi de définir des variables tel que : la liste des items disponible sur ce noeud, le nombre d'item maximum qu'on peut en prendre, la liste des items qu'il est possible d'acheter, le nombre de vie perdue / gagnée sur ce noeud, mais aussi, et surtout, la liste des choix disponible. Ce dernier est défini par une List<T> car cela peut être un choix de type **BookNodeLink** ou un choix de type **BookNodeLinkRandom**. Tout depend du type de noeud. Car les lien "aléatoire", donc de type **BookNodeLinkRandom**, a besoin d'une valeur de plus : la chance. Pour plus d'explication sur ces liens, vous pouvez aller voir la subsection ii à la page 13. Nous avons donc fait le choix d'utiliser un générique dans la classe **AbstractBookNodeWithChoice** afin de définir la classe utilisé pour la liste de choix. De cette manière, on empêche un noeud random (**BookNodeWithRandomChoices**) de contenir des choix "normaux" (**BookNodeLink**). Il ne peut contenir que des **BookNodeLinkRandom**, c'est à dire des choix avec une probabilité d'être choisi.

Pour le **BookNodeCombat**, le choix a été fait d'ajouter trois **BookNodeLink** afin de représenter les différentes issues du combat : victoire, evasion, defaite. Cela permet donc d'avoir, au maximum, un lien en fonction de la finalité du combat. Bien sûr tout les liens ne sont pas obligatoire. Mais cela génère alors un noeud terminal de défaite si le lien est inexistant, peu importe la finalité du combat (voir la section sur le Jeu D à la page 35). Une liste contenant les ID des ennemis que l'on doit combattre y est également renseignée. Les ID seuls sont nécessaire car ils sont uniques, donc peut être facilement retrouvé dans la liste des personnages présente dans la sauvegarde du livre. Ce noeud posséde les même attributs que AbstractBookNodeWithChoices, raison pour lesquels il en hérite. Nous avons donc redéfini les méthodes spécifiques aux choix, c'est à dire celle pour récupérer les choix, celle pour les supprimer et celle pour les mettre à jour. La liste de la classe mère n'est plus utilisable. Il s'agit d'une grosse erreur de notre part. En effet, cette classe demande un traitement spécial à presque tous les endroits où l'on peut gérer des noeuds. Il aurait plutot fallu ajouter les choix dans la liste parente et avoir un moyen de faire une distinction afin de savoir lequel est celui de victoire, de défaite ou d'évasion. On aurait pas eu à redéfinir autant de fois des comportements particuliers pour ce noeud avec des instanceof. Par manque de temps et au vu des changements importants que cela nécessitait, pour faire cela proprement nous n'avons pas pus changer cela.

Pour le **BookNodeWithRandomChoices**, une méthode a été ajoutée, afin de sélectionner un choix de manière aléatoire en fonction de la probabilité de chaque lien. Pour cela, on ajoute d'abord tous les liens disponibles. C'est à dire, les liens où le joueur peut avoir accès en fonction des prérequis demandés. Si aucun lien n'est disponible, cela retourne null. La suite ce passe dans la partie jeu, soit à la section D à la page 35. Dans le cas contraire, cela choisit un nombre aléatoire en fonction de la somme totale des probabilités des liens valides (Voir listing 5.1). Puis, en fonction de ce nombre, la probabilité de chaque lien valide est enlevée du nombre tiré jusqu'à ce qu'il soit égal ou inférieur à zéro. Le choix sélectionné est alors retourné (c'est la destination). Nous avons décider de procéder comme cela car, c'était pour nous, la meilleur solution pour effectuer un choix le plus aléatoirement possible, tout en ayant des valeurs défini par l'utilisateur. Nous aurions pu simplement faire simple choix aléatoire, mais étant un éditeur d'un livre de l'utilisateur, nous avons pensé qu'un ajout de "chance" serait plus apprécié par celui-ci.

```
1     int nbrChoice = 0;
2     Random random = new Random();
3     int nbrRandomChoice = random.nextInt(somme);
4     for (int i = 0 ; i < listNodeLinkDisponible.size() ; i++){
5         if(!this.getChoices().get(i).isAvailable(state)){
6             continue;
7         }
8         nbrRandomChoice -= this.getChoices().get(i).getChance();
9         if(nbrRandomChoice < 0){
10            nbrChoice = i;
11            break;
12        }
13    }
```

```
14      return this.getChoices().get(nbrChoice) ;
```

Listing 5.1 – getRandomChoice()

ii Représentation des liens

Les noeuds sont liés par des liens. Ces liens sont soit définis par la classe **BookNodeLink** ou **BookNodeLinkRandom**. Pour éviter les redondances, cette dernière extends de **BookNodeLink**. Elles prennent donc toutes les deux un texte, une destination (défini par le numéro du noeud) et enfin, une liste de prérequis. Pour la destination, nous avons d'abord mis un **AbstractBookNode**, mais beaucoup de manipulations étaient nécesssaires lorsqu'un changement était apporté au noeud. Nous avons alors choisi de changer et de mettre le numéro du noeud suivant. Pour plus d'information sur la gestion actuelle des noeuds, voir la subsection La classe Book à la page 16.

Pour le **BookNodeLinkRandom**, la classe à besoin d'une variable pour gérer la probabilité, afin de définir la chance d'aller vers ce noeud. Cette probabilité est ensuite totalisé sur tous les noeuds disponibles, comme vu précédement getRandomChoice().

La classe **BookNodeLink** est composé d'une méthode, nommé *isAvailable()* permettant de savoir si le personnage principal rempli les conditions, défini par List<List<AbstractRequirement», pour aller vers ce lien. Nous avons décidé de réalisé une liste de liste qui contiendrait les types de Requirement possible afin d'avoir la possibilité d'inclure un "ou" entre les listes. Par exemple : les conditions emprunter un lien peut est soit ["Arme","Potion"] ou ["Flèche","Lancer un sort"].
La classe **Jeu**, décrite un peu plus bas (voir D), veut alors aller a une destination mais doit emprunter un lien. La méthode *isAvailable()* est donc appelé pour savoir si le joueur peu aller vers ce lien. Cette méthode appelle une fonction nommé *isSatisfied()* (voir 5.3), situé dans le AbstractRequirement. Elle permet de comparer l'inventaire et les compétences du joueur en parcourant tout les prérequis demandés. Si c'est le cas, cela retourne true. False dans le cas contraire. Le joueur peu alors accéder ou pas à ce lien, et donc au noeud choisis.

```
1   public boolean isAvailable(BookState state) {
2       if(requirements.isEmpty())
3           return true;
4
5       for(List<AbstractRequirement> groupRequirement : requirements) {
6           boolean satisfied = true;
7           for(AbstractRequirement r : groupRequirement) {
8                   if(!r.isSatisfied(state)) {
9                       satisfied = false;
10                      break;
11                  }
12              }
13
14          if(satisfied)
15              return true;
16      }
17
18      return false;
19  }
```

Listing 5.2 – exemple de isAvailable()

iii Prérequis pour un choix

Les prérequis des liens sont défini par une classe mère AbstractRequirement puis une classe par type de prérequis (requirements). Il y a la classe **RequirementItem**, **RequirementMoney** ainsi que **RequirementSkill**. Tous extends de AbstractRequirement afin de pouvoir redéfinir la méthode *isSatisfied()*, prenant en paramètre l'état du jeu (BookState). Ce paramètre permet donc de comparer le personnage principal, donc le joueur, avec le prérequi demandé par le lien. Cela renvoie donc un boolean sur la finalité de cette comparaison. Tous prennent un ID en compte, permettant de retrouver l'item / compétence / monnaie demandé. Seul le RequirementMoney possède une méthode qui diffère, c'est la quantité de money requis.
Pour savoir si un item / compétence est présent dans l'inventaire du personnage principal, une for est utilisée afin de regarder les items / compétences possédés. Si l'ID de l'item / compétence n'est pas possédé, le personnage principal ne peut satisfaire les prérequis. Pour l'argent grâce à l'ID de la monnaie, cela permet de savoir si le joueur en possède suffisament. Actuellement, une seule monnaie est disponible, se nommant "gold". En effet, nous n'avons pas eu le temps de le gérer dans les fichiers et dans le jeu.

```
1 public boolean isSatisfied(BookState state) {
2     for (String i : state.getMainCharacter().getItems()){
3         if(i.equals(itemId)) {
4             return true;
5         }
6     }
7
8     return false;
9 }
```

Listing 5.3 – exemple de isSatisfied()

iv Représentation des personnages, items

Nous allons maintenant parler des personnages et des items du livre. Bien qu'il n'y ai pas grand chose à expliquer sur eux, car il s'agit de classes possédant beaucoup de getter et setter, nous souhaitons détailler certains choix faits.

Commençons par les personnages. Ceux ci sont définits par la classe **BookCharacter** dans le package magic_book.core.game. Cette classe possède presque uniquement des getter et setter bien qu'elle possède également quelques méthodes utilitaires.

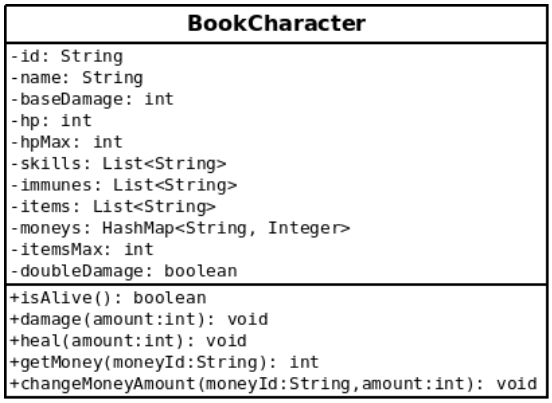

FIGURE 5.3 – UML sur la gestion des personnages

Concernant les listes de String (skills, immunes, items), il s'agit d'une liste contenant les id. Pour "immunes", il s'agit des ID des skills contre lesquels le personnage est immunisé. Cela permet aux items

et aux compétences de n'être référencés qu'à un seul et même endroit, c'est à dire, dans la classe **Book**. Pour la monnaie, on a choisi d'utiliser une HashMap<String, Integer> afin de pouvoir gérer différents types de monnaie et leur montant dans une même histoire. Malheureusement, notre format de livre, et donc notre application, ne permet pour le moment pas de gérer pleinement cette fonctionnalité.

Passons maintenant aux items. Ceux-ci possèdent une classe mère BookItem dans le package magic_book.core.item.

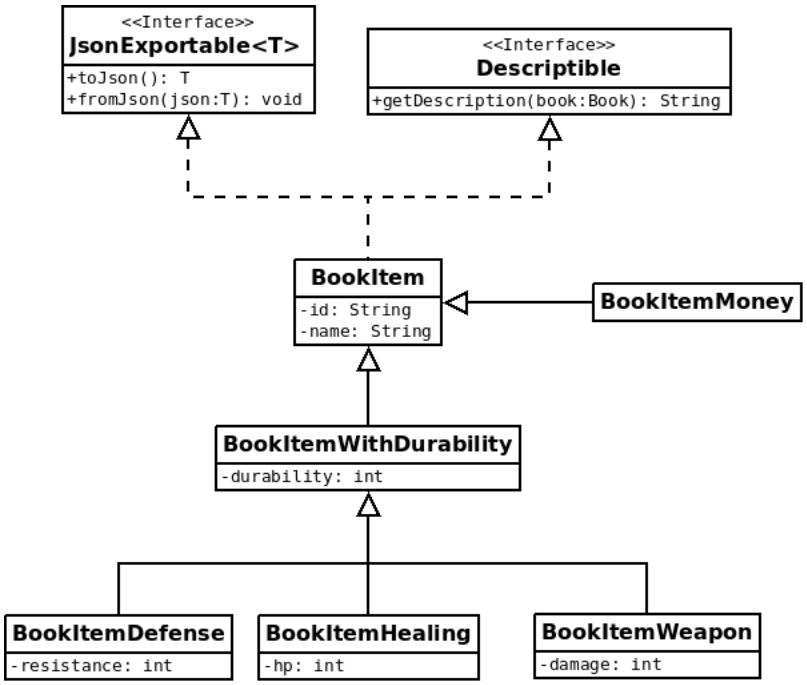

FIGURE 5.4 – UML sur la gestion des items

Avant de détailler notre choix, nous allons expliquer brièvement les deux interfaces que l'on observe. L'une se nomme **Descriptible** et permet à l'item de se décrire sous forme de String. Bien que la méthode *String toString()* soit déjà prévu à cet effet, celle ci ne permet pas de prendre en argument un livre (classe **Book**). Bien entendu, c'est logique mais certains objets ont besoin du livre pour se décrire, par exemple, pour retrouver un item / personnage à partir de son id. La seconde interface, **JsonExportable** sera expliqué plus en détails dans La lecture et l'écriture à la page 24. Pour rester bref, disons qu'elle permet, la lecture et l'écriture du fichier en JSON.

Dès lors, l'héritage prend son sens et permet une spécialisation d'un item de deux façons. La première, qui est la plus logique, permet l'ajout d'attributs spécfiques à notre classe fille. Par exemple, il serait étrange d'avoir un attribut pour savoir combien de dégat un item de type monnaie inflige. Deuxièmement, cette spécialisation intervient dans la redéfinition, par les classes filles, des méthodes des deux interfaces. En effet, chaque classe fille apporte ses propres attributs à chacune des différentes méthodes comme on peut le voir sur le listing ci-dessous. On nottera l'appel à la méthode définit dans la classe mère par le mot clé *super.nomMethode(arguments)*.

```
1  @Override
2  public String getDescription(Book book) {
3      StringBuffer buffer = new StringBuffer();
4
5      buffer.append(super.getDescription(book));
6
7      buffer.append("Dégats : ");
8      buffer.append(damage);
9      buffer.append("\n");
10
```

```
11      return buffer.toString();
12  }
13
14  @Override
15  public ItemJson toJson() {
16      ItemJson itemJson = super.toJson();
17
18      itemJson.setDamage(damage);
19      itemJson.setItemType(ItemType.WEAPON);
20
21      return itemJson;
22  }
```

Listing 5.4 – Exemple de spécialisation des items

v La classe Book

Cette classe est la plus importante de tout le projet. En effet, c'est elle qui met en lien tout les différents éléments qu'on a pu évoquer avant. C'est en effet ce que l'on peut observer sur la figure suivante.

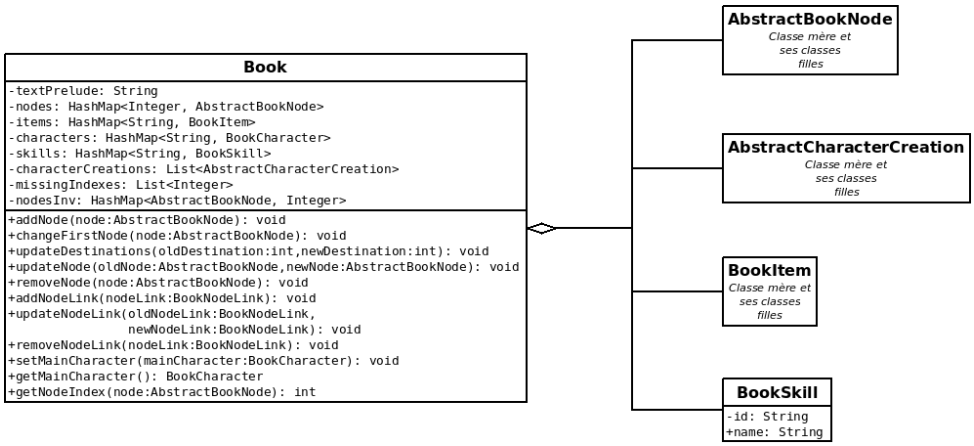

FIGURE 5.5 – UML sur la classe Book

Par soucis de place, les getters et setter n'ont pas été renseignés. Il en va de même pour le détails des différentes classes (**AbstractBookNode**, **BookItem**, ...). Enfin, les observateur sont volontairement omis car ils seront détaillés un peu plus tard (dans Le pattern observer et la classe Book à la page 19).

Tout d'abord, commençons par expliquer comment sont sauvegardés les noeuds et les liens dans la HashMap "nodes".

Cette HashMap possède un int comme clé, qui correspond au numéro du paragraphe. Lorsque l'on ajoute un noeud au livre, on doit d'abord lui trouver un numéro. Nous avons décidé de plusieurs règles. Les paragraphes commencent à partir du numéro 1. Le numéro 1 représente **toujours** le premier paragraphe du livre. De ce fait, si l'on ajoute un noeud, il devra avoir pour numéro le 2.

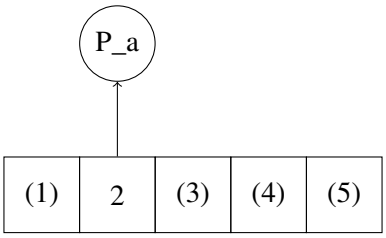

FIGURE 5.6 – Ajout du paragraphe A

Les autres numéros sont représentés mais mis entre parenthèse car ils n'existent pas dans la Map. Ils sont uniquement là pour nous aider à bien visualiser ce dont on parle. Si nous décidons maintenant d'ajouter un second paragraphe alors celui-ci sera à la position 3.

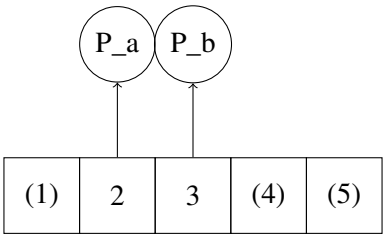

FIGURE 5.7 – Ajout du paragraphe B

Maintenant, supposons que nous souhaitons que notre paragraphe A soit le premier noeud du livre, alors il suffira de l'ajouter dans la map l'indice 1 et de supprimer la clé 2 de notre Map.

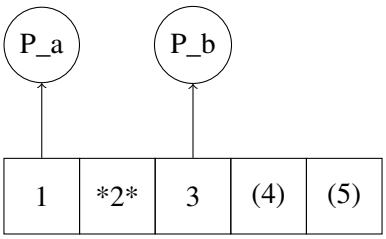

FIGURE 5.8 – Le paragraphe A devient le noeud de départ

Dès lors, une case vide se retrouve disponible (symbolisé par des **). Ainsi, si l'on souhaite ajouter un noeud, il faudra d'abord combler ce vide. Le prochain paragraphe, le C donc, aura pour numéro le 2.

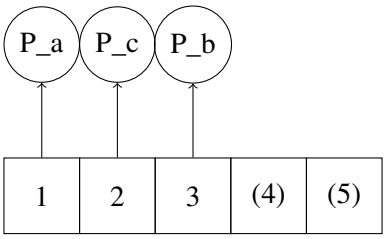

FIGURE 5.9 – Ajout du paragraphe C

Enfin, notre application peut recommencer à ajouter des noeuds à la "fin" de notre Map.

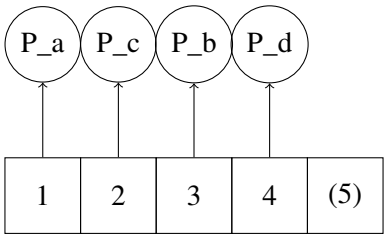

FIGURE 5.10 – Ajout du paragraphe D

Voici quelques précisions supplémentaires :

- Le principe d'indice manquant est le même pour la suppression d'un noeud
- Afin de gagner en performance pour déterminer le numéro d'un noeud déjà présent (pour savoir quel numéro sera manquant par exemple), une autre HashMap est mis à jour en même temps que celle des noeuds. Il s'agit de nodeInv. Cette map n'est rien de plus qu'une sorte de mirroir pour celle des noeuds, on lui passe un noeud en clé et on obtient son numéro. Il est donc extrêmement important que les deux maps soient parfaitement identique pour éviter tout bug.
- Pour déterminer l'indice d'un noeud que l'on ajoute, nous nous basons sur la taille de la Map. Cela pose problème dans un cas particulié. En effet, si nous supprimons un paragraphe au milieu de la map, le noeud à l'indice 3 par exemple, alors cet indice sera libre. Si nous sauvegardons et décidons d'ouvrir de nouveau le fichier, alors nous n'aurons plus cette liste. Il faudrait que nous parcourions la Map pour trouver les indices manquant à la lecture du livre, cependant, comme évoqué un peu partout dans ce rapport, nous avons manqué de temps, d'autant plus que nous avons pensé à ce détails quelques jours avant de rendre le rapport.

De ce fait, voici l'algorithme que nous avons mis en place pour l'ajout d'un noeud.

Algorithm 1: Ajout d'un noeud

```
Input: le noeud à ajouter : node
Data: nodes : Map<Integer, AbstractBookNode> liste des noeuds
      nodesInv : Map<AbstractBookNode, Integer> liste inversée des noeuds
      missingIndexes : List<Integer> liste des indices libres
1  if node in nodes then
2      return
3  if missingIndexes.length == 0 then
4      offset : int
5      offset ← (1 in nodes) ? 1 : 2
6      nodes[nodes.length + offset] ← node
7      nodesInv[node] ← nodesInv.length + offset
8  else
9      nodes[missingIndexes[0]] ← node
10     nodesInv[node] ← missingIndexes[0]
11     missingIndexes.remove(0)
12 notifyNodeAdded(node)
```

La variable offset correspond au décalage à ajouter pour placer le noeud. Comme on commence à 2 un décalage de 2 est nécessaire. Supposons que le premier noeud est renseigné et qu'il est le seul du tableau, ajouter un nouveau noeud le placerait donc celui-ci à la position 2 + tailleDuTableau soit 2 + 1 c'est à dire 3. On a alors un décalage d'une "case". De ce fait, on doit faire un décalage de 1 uniquement si le premier noeud est renseigné.

Voyons maintenant celui mis en place pour le changement du premier noeud.

Algorithm 2: Changement du premier noeud

```
Input: le nouveau premier noeud : node
Data: nodes : Map<Integer, AbstractBookNode> liste des noeuds
nodesInv : Map<AbstractBookNode, Integer> liste inversée des noeuds
missingIndexes : List<Integer> liste des indices libres
if not (node in nodes) then
    addNode(node)

updateDestinations(1, -1)

indexOfNode : int
indexOfNode ← nodesInv[node]

oldNode : AbstractBookNode
oldNode ← nodes[1]

updateDestinations(indexOfNode, 1)

nodes[1] ← node
nodesInv[node] ← 1

if oldNode != null then
    nodes[indexOfNode] ← oldNode
    nodesInv[oldNode] ← indexOfNode
    updateDestinations(-1, indexOfNode)
else
    missingIndexes.add(indexOfNode)
    nodes.remove(indexOfNode)
```

NB : updateDestinations permet de changer les numéro de destination des BookNodeLink d'un ancien numéro, vers un nouveau

Si le noeud n'est pas présent dans le livre, nous commençons par l'ajouter. Les numéros des paragraphes vont êtres amenés à changer, de ce fait, il est important de mettre à jour les numéros de destination des différents liens. Nous commençons par déplacer les références du noeud 1 vers -1. En effet, cet indice n'est jamais renseigné. Ensuite, nous changeons les destination des liens qui allaient vers le "noeud à placer en premier" pour qu'elles pointent vers le premier noeud. Nous ajoutons le noeud à cette premiere "case". Deux options sont maintenant possibles. Il y avait déjà un premier noeud auparavant auquel cas il faut maintenant le placer là où se trouvait l'ancien et donc, mettre à jour les liens qui vont de -1 vers ce nouvel emplacement. Si jamais il n'y avait pas de premier noeud, alors une place est maintenant manquante. On l'ajoute donc à la liste des emplacements à combler avant de pouvoir de nouveau ajouter des noeuds normalement.

vi Le pattern observer et la classe Book

Le pattern observer est essentiel pour la mise en place du pattern MVC. Nous avons décidé de procéder de la manière suivante :

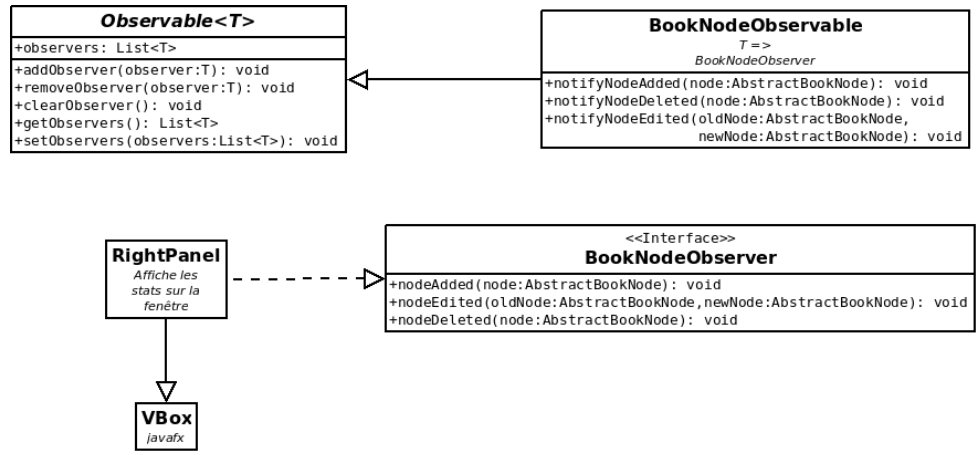

FIGURE 5.11 – UML d'exmple sur le pattern observer

Comme on peut le voir, une classe mère **Observable<T>** détient une List<T> d'observers. Une méthode d'ajout, et de suppression permettent de modifier cette liste. Dès lors que l'on souhaite ajouter un nouveau type d'observer, on doit commencer par créer une nouvelle Interface avec les méthodes que l'on souhaite fournir, dans notre exemple il s'agit de nodeAdded, nodeEdited, nodeDeleted. Une fois cette interface faite, on doit alors faire une nouvelle classe Observable, BookNodeObservable dans notre cas, qui hérite de Observable<T>, T étant l'observer que l'on souhaite utiliser, donc BookNodeObserver. Ainsi, chaque Observable que l'on fera ne sera qu'une définition des méthodes pour notifier qu'un évènement s'est produit.

Le choix à été fait de séparer les différentes parties (noeuds, liens, items, ...) du livre en différents observers afin de ne pas surcharger le nombre de méthodes à redéfinir dans les classes Observer, et donc de n'observer que ce dont on a besoin. De ce fait, tous les observers concernant le livre sont sont disponibles sauf celui pour notifier d'un changement concernant le premier noeud.

B Lecture et écriture d'un livre

L'objectif de l'application étant de concevoir un éditeur, il était important de permettre la sauvegarde et la lecture du livre que l'on édite. Le choix du format JSON est rapidement survenue. Premièrement car un fichier d'exemple qui nous a été fournis était sous ce format mais aussi car il s'agit d'une structure simple et très facile à lire. Nous avons alors utilisé GSON, une librairie, conçue par Google, extrêmement simple. Elle permet de retranscrire sous forme d'objet Java un fichier JSON structuré, c'est à dire où l'on distingue très clairement des objets qui se répète.

Afin de lire un fichier JSON, avec cette librairie, il suffit de concevoir des objets Java avec les même attributs que ceux du fichier à lire ou à écrire. Voici un exemple très court de ce à quoi nos fichiers ressembles :

i La structure du JSON

```
1 {
2     "prelude": "Vous êtes l'enseignant qui note notre projet",
3     "setup": {
4         "skills": [],
5         "items": [],
6         "characters": [],
7         "character_creation": []
8     },
```

```
    "sections": {
        "1" : {
            "text": "Vous être en train d'étudier notre projet",
            "choices": [
                {
                    "text": "Mettre une bonne note",
                    "section": 3
                },
                {
                    "text": "Mettre une mauvaise note",
                    "section": 2
                }
            ]
        },
        "2": {
            "text": "Les étudiants du projet sont tristes",
            "end_type": "FAILURE"
        },
        "3": {
            "text": "Les étudiants sont satisfait de leur travail",
            "end_type": "VICTORY"
        }
    }
}
```

Listing 5.5 – Exemple de livre très simple

On retrouve plusieurs éléments différents. On remarque par exemple un attribut "prelude", ainsi que deux grosses parties, "setup" et "sections". Dans la suite, nous détaillerons uniquement les attributs les plus frequemment présents.

Setup

Commençons par détailler "setup". Ce passage contient toutes les informations générales à notre livre. On y retrouve la liste des compétences ("skills"), la liste des items ("items") et la liste des personnages ("characters"). "character_creation", lui, détaille toutes les étapes lors de la conception du personnage qui intervient au tout début. Celle-ci permet de sélectionner des compétences et items de départ.

Pour le moment les compétences sont uniquement composé d'un ID et d'un nom. Dans une future mise à jour il serait intéressant d'ajouter des propriétés pour connaitre la force ajouté dans un combat, la quantité de soins à rendre par noeuds, par exemple.

```
{
    "id": "sixth_sense",
    "name": "Sixième sens"
}
```

Listing 5.6 – Exemple de compétence

Les items peuvent être de différents types : KEY_ITEM, WEAPON, DEFENSE, MONEY, HEALING. On retrouve pour tous les items un id et un nom ("name"). Pour certains types, des attributs supplémentaires sont présent. Par exemple, un attribut "durability" peut être présent. Il permet de déterminer le nombre d'utilisation maximum d'un item. Un item de type HEALING possède un nombre de pv à rendre ("hp") tandis que ceux type WEAPON possède un montant de dégats ("damage") par exemple.

```
{
    "id": "backpack",
    "name": "Backpack",
    "item_type": "KEY_ITEM"
},
{
```

```
    "id": "healing_potion_4",
    "name": "Potion de soins (4HP)",
    "hp": 4,
    "durability": 1,
    "item_type": "HEALING"
}
```

Listing 5.7 – Exemple d'items

Concernant les personnages on y retrouve un id, un nom ("name"), un nombre de pv maximum ("hp"), un boolean pour indiquer s'il a beaucoup de chance que ses coups fassent le double des dégats ("double_damage"), ainsi que "combat_skill" qui représente le montant de ses dégats.

```
{
    "id": "zombie_captain",
    "name": "Zombie Captain",
    "hp": 15,
    "double_damage": true,
    "combat_skill": 2
}
```

Listing 5.8 – Exemple de personnage

Les character character_creation peuvent être de simple texte ou de type "ITEM" ou "SKILL". On y retrouve les différents skills ou items que l'on peut prendre pour débuter notre aventure ainsi que le nombre maximum que l'on peut choisir ("amount_to_pick").

```
{
    "text": "Kai Disciplines\n\nOver the centuries, the Kai monks have mastered
    the skills of the warrior. These skills are known as the Kai Disciplines,
    [...]",
    "type": "SKILL",
    "skills": [
        "camouflage",
        "hunting",
        "sixth_sense",
        "tracking",
        "healing",
        "weaponskill",
        "mindshield",
        "mindblast",
        "animal_kinship",
        "mind_over_matter"
    ],
    "amount_to_pick": 5
}
```

Listing 5.9 – Exemple de character_creation

Sections

La partie "sections" est une map qui représente le numéro d'un paragraphe ainsi que le paragraphe associé. Il existe différents types de paragraphes : à choix, à choix aléatoire, avec des combats et terminaux. Tous possède un texte. Les noeuds terminaux possèdent un type de fin ("end_type") afin savoir si l'on a gagné ou pas (cf : Listing 5.5). Les noeuds aléatoires eux, possèdent un attribut "is_random_pick" qui vaut true. Pour tous les autres types de noeuds, on retrouve parmis les attributs les plus importants une liste d'items qu'il est possible de prendre, un montant d'item maximum qui peut être pris ("amount_to_pick") et enfin des items disponibles à l'achat ("shop").

```
{
    "text": "The back door opens [...]",
```

```
    "items": [
        {
            "id": "gold",
            "amount": 5
        },
        {
            "id": "dagger"
        },
        {
            "id": "seal_hammerdal"
        }
    ],
    "amount_to_pick": 2
    "choices": [
        {
            "text": "Return to the tavern.",
            "section": "177"
        },
        {
            "text": "Study the tomb.",
            "section": "24"
        }
    ]
}
```

Listing 5.10 – Exemple de paragraphe

Certains paragraphes peuvent contenir un attribut "combat". Dès lors on peut connaitre le choix en cas de victoire ("win"), de défaite ("loose") ou d'évasion ("evasion). Si l'évasion est possible seulement à partir d'un certains nombre de tour on retrouve alors un attribut nommé "evasion_round". Pour finir, un attribut "enemies" permet de connaitre les personnages que l'on combat.

```
{
    "text": "The dead zombies lie [...]",
    "combat": {
        "win": {
            "text": "If you win the combat.",
            "section": "309"
        },
        "enemies": [
            "zombie_captain"
        ]
    }
}
```

Listing 5.11 – Exemple de paragraphe avec des combats

Pour représenter un lien vers un autre paragraphe on retrouve une liste de choix ("choices"). Ils possèdent également un texte qui correspond à l'intitulé du choix, le numéro du paragraphe suivant ("section"), un nombre d'hp à retirer, un nombre d'argent à ajouter ainsi qu'un liste de prérequis ("requirements"). Comme pour les BookNodeLink, il s'agit d'un tableau à deux dimensions. Le premier représente une liste de condition en OU et le second une liste de condition en ET. Enfin, pour les noeuds aléatoires, un probabilité est également présent ("weight").

```
{
    "text": "If you have the Kai Discipline of Tracking.",
    "section": "182",
    "hp": -5,
    "requirements":  [
        [
            {
                "id": "tracking",
                "type": "SKILL"
```

```
10                }
11            ]
12        ]
13    }
```

Listing 5.12 – Exemple de choix

ii La lecture et l'écriture

Du fait que la structure en Json n'est pas identique à celle détaillé dans **??** (page **??**), nous avons fait des classes intermédiaires pour permettre cette lecture. Celles-ci sont disponibles dans le package *magic_book/core/file/json* et ne contiennent rien de plus que des getter et setter. Aussi, afin de permettre une convertion entre les classes faites pour représenter un fichier json et celles faites pour être utilisées par l'application, une interface *JsonExportable* existe. Celle ci permet de redéfinir 2 méthodes. L'une renvoyant la classe JSON associé à notre classe actuelle, l'autre permettant à partir d'une classe JSON d'obtenir la classe Java correspondante.

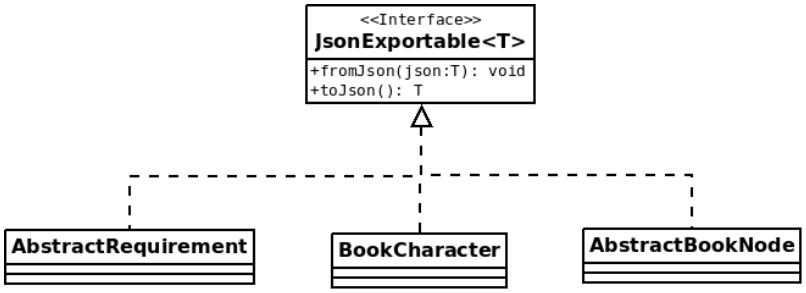

FIGURE 5.12 – L'interface JsonExportable et quelques classes qui l'implémentent

Enfin, les classes BookReader et BookWritter permettent de récupérer toutes les classes JSON intermédiaires pour les regrouper dans le BookJson qui correspond à la structure complète de notre livre. Ces classes sont également une couche d'abstraction à GSON car c'est elles qui se chargent d'écrire le JSON correspondant dans un flux.

Pour résumer, on peut schématiser ces échanges de telle sorte :

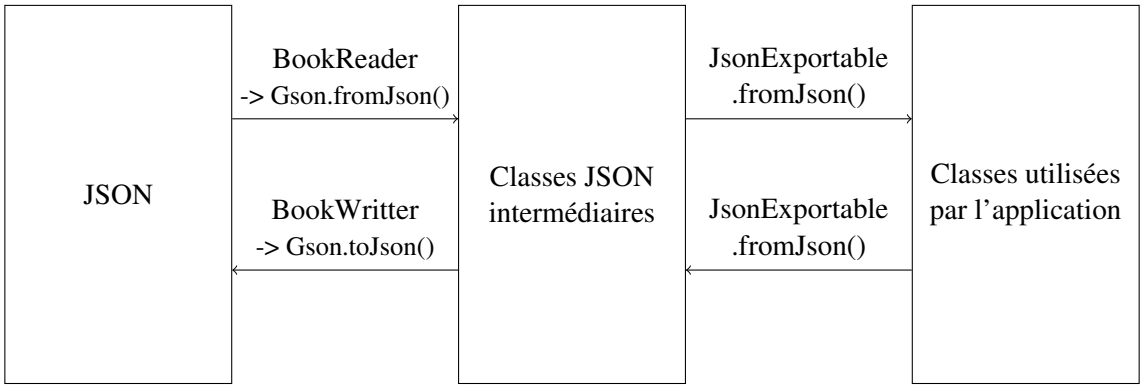

FIGURE 5.13 – Échanges pour la lecture / écriture

C Edition graphique d'un livre

i MainWindow

La MainWindow est notre fenêtre principale. Elle contient un Menu permettant de réaliser plusieurs actions (exporter le livre, jouer, générer la difficulter...) ainsi que trois zones différentes (panel). Le premier panel se nomme le **LeftPane**. Il se situe à gauche et est composé de différent boutons permettant de changer de mode, de la liste des items, de personnage, de compétences du livre. Le deuxième panel s'appel le **GraphPane**. Il est au centre et permet d'ajouter les noeuds ainsi que les liens entre les noeuds, c'est donc la zone d'édition de notre livre. Il contient également le prélude. Le troisième panel se nomme le **RightPane**. Il permet d'afficher les statistiques du livre comme par exemple, le nombre de noeud ainsi que l'estimation de la difficulté du livre.

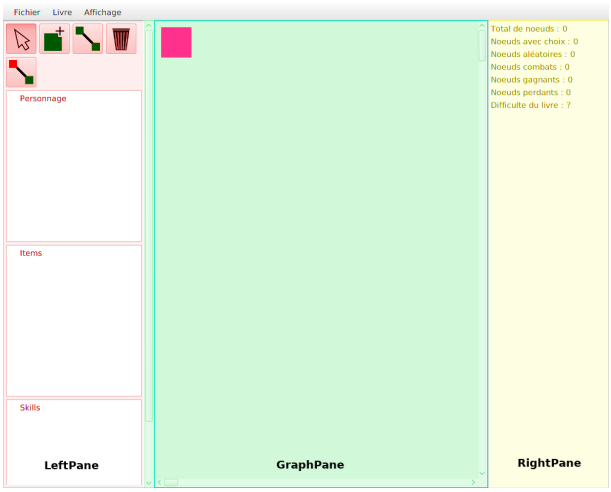

FIGURE 5.14 – MainWindow

Pour afficher cette fenêtre, plusieurs classe sont nécessaires (voir ci-dessous). La **MainWindow** gère la création de la fenêtre en créant d'abord la barre de menu grâce à la méthode *createMenuBar()*, appeler dans son constructeur, ainsi que les options de la barre. Elle appele également, grâce à son constructeur, les classes **LeftPane**, **GraphPane** ainsi que **RightPane** qui construisent les trois panels principaux. Toutes ces composantes constituant la fenêtre, sont ajouter au BorderPane permettant ainsi de les positionner.

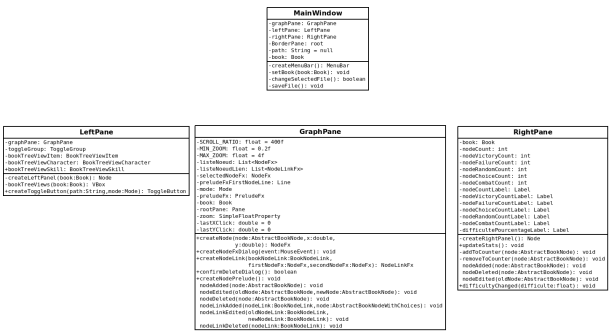

FIGURE 5.15 – Classes lié à la fênetre

Fichier

L'utilisateur peut ouvrir un nouveau livre vide en cliquant sur Nouveau. La classe **MainWindow** modifie alors le livre actuel en créant un livre vierge. Une méthode *setBook()* est alors appelé afin de

modifier le livre dans tout les panels (LeftPane, GraphPane, RightPane), permettant ainsi de partir sur un nouveau livre.

Ce menu comporte aussi un MenuItem nommé Ouvrir, permettant ainsi d'ouvrir un fichier Json ou txt. Si dans ce fichier, il manque une virgule ou une accolade ou autre ne permettant pas la lecture du fichier en Json (Voir Lecture et écriture d'un livre à la page 20), un message d'erreur apparait. Si le fichier n'est pas bien fait, ne permettant pas l'ouverture de ce fichier, une classe nommée **BookValidator** devait "valider" le livre. Par manque de temps, cette classe n'est pas fonctionnelle. Si le fichier est valide, cela appel alors la classe **BookReader** permettant ainsi de lire le fichier en json (pour plus d'information, voir la subsection ii). La méthode *setBook()* est ensuite appelé afin de modifier le livre dans tout les panels (LeftPane, GraphPane, RightPane).

Les options Enregistrer ainsi que Enregistrer-Sous sont également présente. Elles appellent toutes les deux les même méthodes : *changeSelectedFile()* ainsi que *saveFile()*. La première méthode permet d'enregistrer le fichier. Si l'utilisateur annule la boite de dialog permettant d'enregistrer le fichier, la méthode *saveFile()* n'est donc pas appelé. Dans le cas contraire, la méthode *saveFile()* permet de sauvegarder le document en fonction du chemin du fichier, défini par la variable "path". Pour "écrire" le fichier qui sera enregistrer, permettant par la suite de l'ouvrir, la classe **BookWritter** est donc utilisé (pour plus d'information, voir la subsection ii). Si la moindre erreur survient lors de l'enregistrement du fichier écrit, cela annule l'enregistrement. La seule diférence entre Enregistrer et Enregistrer-Sous, ce passe si la variable "path", soit le chemin du fichier, soit déjà défini. Si c'est le cas, Enregistrer appel directement la méthode *saveFile()* car l'utilisateur n'a pas besoin de redéfinire le chemin d'enregistrement du fichier.

```
1         private void saveFile() {
2             File saveFile = new File(path);
3 
4             if(!saveFile.getParentFile().exists())
5                 saveFile.mkdirs();
6 
7             BookWritter bookWritter = new BookWritter();
8             try {
9                 bookWritter.write(path, book);
10            } catch (IOException ex) {
11                Alert a = new Alert(AlertType.ERROR);
12                a.setTitle("Erreur lors de l'écriture du fichier");
13                a.setHeaderText("Impossible d'écrire le fichier sur le disque");
14                a.show();
15            }
16        }
17 
```

Listing 5.13 – Enregistrement du livre

Livre

L'utilisateur peut Jouer ou Estimer la difficulté. Ces deux MenuItem utilisent toutes les deux la classe **Jeu** décrit un peu plus loin Rendre le livre jouable et estimer sa difficulté à la page 35. Le livre est alors charger dans cette classe afin d'avoir accès à toutes les informations pour pouvoir jouer.

Pour l'option Jouer, elle utilise la méthode *play()* de la classe **Jeu**. Si cette méthode relate la moindre erreur, celle si remonte jusqu'à la **MainWindow**. Un message d'erreur apparait alors. Cette erreur survient normalement si un chemin ne conduit sur aucun lien terminaux ou si un noeud se retrouve sans lien.

Pour l'option Estimer la difficulté, elle utilise la méthode *fourmis()* de la classe **Jeu**. Une simple valeur indiquant le nombre de "fourmis" voulant être lancé est envoyé dans cette méthode. Puis, une fois la difficulté calculé, ce pourcentage apparait dans le panel des statistiques, soit dans la partie droite de la fenêtre grâce à la méthode *difficultyChanged()* de la classe **RightPane**.

Un autre MenuItem est aussi présent, permettant de Générer le livre en txt. Cela permet à l'utilisateur de pouvoir avoir un livre propre au format texte. Pour ce faire, un lien entre le prélude et le premier noeud doit être généré. Sinon une erreur apparait en indiquant le problème de l'erreur. En effet, le prélude apparait d'abord, puis le premier noeud. Cela ne peux pas se faire tant que ce lien n'est pas défini. Si le lien est générer, l'utilisateur doit alors indiquer où enregistrer le fichier génerer à l'aide de la classe **FileChooser**. Si l'utilisateur annule, la génération est annulé. Sinon, cela génère le fichier à l'aide de la classe **BookTextExporter** à l'adresse spécifié dans **FileChooser**. Bien sûr, cela s'annule à la moindre erreur.

```
1        MenuItem menuBookGenerate = new MenuItem("Générer le livre en txt");
2            menuBookGenerate.setOnAction((ActionEvent e) -> {
3            NodeFx firstNodeFx = graphPane.getPreludeFx().getFirstNode();
4            if(firstNodeFx == null) {
5                Alert a = new Alert(Alert.AlertType.WARNING);
6                a.setTitle("Erreur lors de l'export");
7                a.setHeaderText("Merci de sélectionner au préalable le noeud de
    départ");
8                a.show();
9
10               return;
11           }
12
13           FileChooser fileChooser = new FileChooser();
14           fileChooser.setTitle("Exporter en texte");
15
16           File selectedFile = fileChooser.showSaveDialog(this);
17           if (selectedFile == null) {
18               return;
19           }
20
21           try {
22               BookTextExporter.generateBook(book, selectedFile.getAbsolutePath
    ());
23           } catch (IOException ex) {
24               Alert a = new Alert(Alert.AlertType.ERROR);
25               a.setTitle("Erreur lors de l'export du fichier");
26               a.setHeaderText("Impossible de sauvegarder le fichier sur le
    disque");
27               a.show();
28           }
29       });
30
```

Listing 5.14 – Génération du livre en txt

Affichage

L'utilisateur peut afficher ou non, le **LeftPane** et/ou le **RightPane**. Cela permet notament à l'utilisateur de mieux voir la partie édition. Mais était surtout réalisé pour un meilleur affichage en vidéo projecteur de notre projet. Une simple modification de ce qui est affiché à gauche ou à droite du BorderPane (soit de la variable "root" contenant toute la fenêtre) est nécessaire.

ii LeftPane

Ce panel contient tout d'abord des ToggleButton permettant de sélectionner un mode parmis cinq modes : SELECT, ADD NODE, ADD NODE LINK, DELETE, FIRST NODE. Ces modes sont sélectionnables à l'aide de bouton créé a partir de la méthode *createToggleButton()*. Cette méthode prend en paramètre une image et un des modes de la classe **Mode**, qui est une classe ennumérant tout les modes.

Un évènement est associé à chaque boutons permettant de changer le mode (voir iii). Le mode ensuite déjà selectionné est toujours le mode SELECT. On a créer des ToggleButton afin d'avoir deux états : un état on et un état off. Cela permet de "désactiver" les boutons non sélectionné et de n'avoir qu'un seul bouton non sélectionné. Pour sela, un **ToggleGroup** permettant que dès qu'un bouton est "cliquer" que celui-ci soit en "on" et les autres en "off".

```
1    private ToggleButton createToggleButton(String path, Mode mode) {
2        InputStream stream = ClassLoader.getSystemClassLoader().
     getResourceAsStream(path);
3        ImageView imageView = new ImageView(new Image(stream));
4        imageView.setFitHeight(40);
5        imageView.setFitWidth(40);
6        ToggleButton toggleButton = new ToggleButton("", imageView);
7
8        imageView.setPreserveRatio(true);
9        toggleButton.setMinSize(UiConsts.CONTROL_BUTTON_SIZE, UiConsts.
     CONTROL_BUTTON_SIZE);
10       toggleButton.setMaxSize(UiConsts.CONTROL_BUTTON_SIZE, UiConsts.
     CONTROL_BUTTON_SIZE);
11
12       toggleButton.setOnAction((ActionEvent e) -> {
13           if(toggleGroup.getSelectedToggle() == null) {
14               toggleGroup.selectToggle(toggleButton);
15               return;
16           }
17
18           graphPane.setMode(mode);
19           graphPane.setSelectedNodeFx(null);
20       });
21
22       if (this.toggleGroup == null) {
23           this.toggleGroup = new ToggleGroup();
24       }
25
26       this.toggleGroup.getToggles().add(toggleButton);
27
28       return toggleButton;
29   }
```

Listing 5.15 – Création du ToggleButton

Une VBox est ensuite créé afin de mettre les trois TreeView contenant un les personnages, items et compétences. Pour cela, la méthode *bookTreeView()* est appelé afin d'appeler chaque classe nécessaire à l'affichage de ces TreeView : **BookTreeViewCharacter** pour la liste des personnages, **BookTreeViewItem** pour la liste des items et **BookTreeViewSkill** pour la liste des compétences.

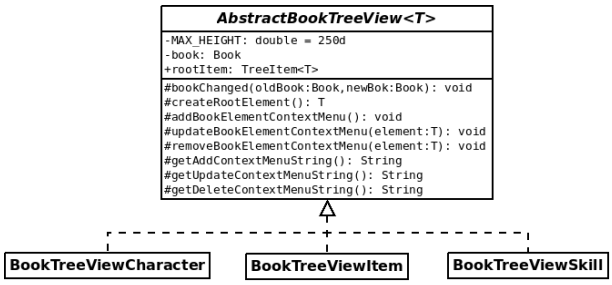

FIGURE 5.16 – Constitution de la vue des personnages/item/compétences

Comme vous pouver le voir dans la figure ci-dessous, une classe AbstractBookTreeView implique ces trois classe. Elles ont donc toutes les même méthode mais défini en fonctions de ce qu'ils affichent.

- bookChanged() permet de mettre à jour la liste si un changement de livre est effectué. Elle est donc appelé par la méthode *setBook()*.
- createRootElement() permet de créer l'item qui sera afficher dans la vue et non modifiable. Par exemple, pour la liste des items, "Items" sera donc afficher tout en haut.
- addBookElementContextMenu() permet d'ajouter un personnage / item / compétence. Cela appel alors la boite de dialogue en fonction du TreeView appelé. Les classes **CharacterDialog**, **ItemDialog**, **SkillDialog** sont alors appelé. Si l'utilisateur valide la boite de dialogue, cela l'ajoute dans le HashMap prévu à cette effet dans le livre. Par exemple si c'est une compétences, *addSkill()* est alors appelé.
- updateBookElementContextMenu() permet de modifier l'élément sélectionné. Cela réouvre la boite de dialogue. Si l'utilisateur valide la boite de dialogue, cela modifie l'élément sélectionné dans le HashMap du livre. C'est pour cela qu'un HashMap a été créer et non une liste.
- removeBookElementContextMenu() permet de sélectionné l'élément sélectionné. Cela supprime donc l'élément dans la HashMap du livre.
- getAddContextMenuString() permet d'afficher le texte permettant d'ajouter le personnage / l'item / la compétence
- getUpdateContextMenuString() permet d'afficher le texte permettant de modifier le personnage / l'item / la compétence
- getDeleteContextMenuString() permet d'afficher le texte permettant de supprimer le personnage / l'item / la compétence

Dans ces différentes classes, un Observeur est ajouter sur chaque classes. Ce qui permet que le moindre clique sur "ajouter", "modifier" ou "supprimer" permet d'effectuer les actions expliqué dans la liste ci-dessus. Un Observeur est créer par classe, il y a donc trois observeur et trois observable. Si un item est ajouté par exemple, c'est la classe **BookTreeViewItem** qui est appelé. Puis la méthode *addBookElementContextMenu()* est appelé et si l'utilisateur valide la boite de dialogue, la méthode *addItem()* dans la classe **Book** est appelé. Cela permet d'ajouter l'item dans le HashMap d'items. Puis cette méthode appelle *notifyItemAdded()* de la classe **BookItemObservable** qui elle, notifie *itemAdded()* de l'ajout d'un item. Cette méthode est dans la classe **BookTreeViewItem**. Elle permet donc d'ajouter l'item en question sur le Pane, permetant de le voir. La "modfication" et la "supression" est sur le même principe.

Grâce au constructeur de **AbstractBookTreeView** (voir ci-dessous), le titre du TreeItem est ajouter. Puis un ContextMenu() est créer et Chaque MenuItem est ajouter ("ajouter", "modifier" ou "supprimer") à ce ContextMenu.

```
1   public AbstractBookTreeView(Book book){
2       rootItem = new TreeItem<>(createRootElement());
3       rootItem.setExpanded(true);
4
5       this.setRoot(rootItem);
6
7       ContextMenu contextMenu = new ContextMenu();
8       MenuItem menuAddElement = new MenuItem(getAddContextMenuString());
9       MenuItem menuUpdateElement = new MenuItem(getUpdateContextMenuString());
10      MenuItem menuDeleteElement = new MenuItem(getDeleteContextMenuString());
11
12      menuAddElement.setOnAction((ActionEvent event) -> {
13          addBookElementContextMenu();
14      });
15
16      menuUpdateElement.setOnAction((ActionEvent event) -> {
17          TreeItem<T> selectedItem = this.getSelectionModel().getSelectedItem
    ();
18
19          if(selectedItem != null && selectedItem != rootItem) {
20              updateBookElementContextMenu(selectedItem.getValue());
21          }
22
```

```
23              this.refresh();
24          });
25
26          menuDeleteElement.setOnAction((ActionEvent event) -> {
27              TreeItem<T> selectedItem = this.getSelectionModel().getSelectedItem
    ();
28
29              if(selectedItem != rootItem)
30                  removeBookElementContextMenu(selectedItem.getValue());
31          });
32
33          contextMenu.getItems().addAll(menuAddElement, menuUpdateElement,
    menuDeleteElement);
34          this.setContextMenu(contextMenu);
35
36          this.setBook(book);
37
38          this.setMaxHeight(MAX_HEIGHT);
39      }
```

Listing 5.16 – Constructeur de AbstractBookTreeView()

iii GraphPane

Le GraphPane lui affiche la zone d'édition. Cette zone contient des méthodes permettant l'ajout de lien ainsi que l'ajout de noeud. Elle contient également deux classes appelé **NodeFxListener** ainsi que **NodeLinkFxListener**. Elle permettent de réalisé des actions en fonction des modes sélectioné dans le LeftPane.

Son constructeur permet de créer le prélude ainsi que d'ajouter des noeud si le mode "ADD_NODE" est sélectionné. Si ce bouton est sélectionné, et qu'un clique est observer sur le GraphPane, la méthode *createNodeFxDialog()* permet d'ouvrir un **NodeDialog**. C'est la boite de dialogue permettant de créer le noeud. Si l'utilisateur valide le noeud, la méthode *addNode()* de la classe **Book** est appelé permattant de rajouter les coordonnée du noeud pour le ZOOM. Puis une notification est envoyé dans le **BookNodeObservable** permettant d'appeler la méthode *nodeAdded()* dans le GraphPane. Cette méthode permet de créer le noeud visuellement. Pour cela, un NodeFx est nécessaire. En effet, ce NodeFx fera le lien entre le RectangleFx qui sera le visuel sur noeud (soit un carré) et le noeud.

NodeFxListener

Si le mode "SELECT" est sélectionné et si la souris maintient le clique appuyé (soit *setOnMouseDragged()*), la classe **RectangleFx** rattaché au noeud est appelé afin de modfier le rectangle de place. Ce qui veux dire que le rectangle à les même coordonnées que la souris. Mais la méthode empêche le Rectangle de sortir de la zone d'édition. Si un double clique est effectué, le deuxième constructeur de la classe **NodeDialog** est appelé avec le noeud sélectionné en argument. Cela permet donc de modifier la saisie sur ce noeud. Si l'utilisateur valide la boite de dialogue, le livre met a jour le noeud contenue dans un HashMap, ce qui permet de le retrouver.

```
1       if(mode == Mode.SELECT){
2           selectedNodeFx = nodeFx;
3           if(event.getClickCount() == 2) {
4               NodeDialog dialog = new NodeDialog(book, selectedNodeFx.getNode
    ());
5               if(dialog.isValid()) {
6                   book.updateNode(nodeFx.getNode(), dialog.getNode());
7               }
8           }
```

```
9         }
10
```

Listing 5.17 – Classe NodeFxListener avec le mode SELECT

La méthode *updateNode()* est alors appelé mettant à jour le noeud. Si un moindre changement du type de noeud est effectué, le lien est alors supprimé grâce au **BookNodeLinkObservable** et de la méthode *notifyNodeLinkDeleted()*. Puis celle-ci fait appel l'observeur **BookNodeObservable** permettant de notifier *nodeEdited()* de la classe **GraphPane**. Cette méthode permet de modifier le noeud dans le NodeFx qui a été cliqué.

Si le mode ADD_NODE_LINK est sélectionné, et que le premier clique ce fait sur un noeud qui n'a pas de choix, donc le prélude ou un noeud terminal, cela ne fait rien. Si au contraire, le noeud est autorisé à avoir des choix, le prmier clique enregistre donc le noeud sélectionné afin d'avoir le départ du lien. Si un autre clique est effectué et que la variable contenant le premier noeud sélectionné n'est pas vide, alors le noeud est créer en fonction du type du premier noeud. En effet, si le premier noeud est un noeud de type combat, et si tout ces choix sont déjà défini (combat gagner, combat perdu, evasion), le lien ne peux pas être effectué. Un message d'erreur s'affiche alors, puis la variable contenant le premier noeud est supprimé. Dans le cas contraire, la boite de dialogue s'affiche. Nous n'avons pas voulu faire d'autre condition car si l'utilisateur veux créer un lien qui effectue une boucle ou plusieurs choix qui mène aux même noeud, il est libre de le faire. Une fois la boite de dialogue valider, et si c'est un noeud de combat, on met à jour le lien parmis ses destinations. Sinon, on ajoute le lien dans le HashMap contenant les liens. Puis on met à zéro la variable contenant le premier noeud.

```
1         if(mode == Mode.ADD_NODE_LINK) {
2             if(selectedNodeFx == null && !(nodeFx.getNode() instanceof
    AbstractBookNodeWithChoices)) {
3                 return;
4             } else if(selectedNodeFx == null && nodeFx.getNode() instanceof
    AbstractBookNodeWithChoices) {
5                 selectedNodeFx = nodeFx;
6             } else {
7                 if (selectedNodeFx.getNode() instanceof BookNodeCombat){
8                     BookNodeCombat firstNodeCombat = (BookNodeCombat)
    selectedNodeFx.getNode();
9                     if(firstNodeCombat.getEvasionBookNodeLink() != null
10                            && firstNodeCombat.getLooseBookNodeLink() != null
11                            && firstNodeCombat.getWinBookNodeLink() != null) {
12                        Alert alertDialog = new Alert(Alert.AlertType.ERROR);
13
14                        alertDialog.setTitle("Erreur");
15                        alertDialog.setHeaderText("Veuillez supprimer un lien de
     victoire / defaite / evasion pour pouvoir rajouter un autre lien.");
16                        alertDialog.show();
17
18                        return;
19                    }
20                }
21
22                NodeLinkDialog nodeLinkDialog = new NodeLinkDialog(
    selectedNodeFx.getNode(), book);
23
24                if(nodeLinkDialog.isValid()) {
25                    BookNodeLink bookNodeLink = nodeLinkDialog.getNodeLink();
26                    bookNodeLink.setDestination(book.getNodeIndex(nodeFx.getNode
    ()));
27
28                    if(selectedNodeFx.getNode() instanceof BookNodeCombat) {
29                        BookNodeCombat bookNodeCombat = (BookNodeCombat)
    selectedNodeFx.getNode();
30
```

```
31                          if(nodeLinkDialog.getLinkType() == NodeLinkDialog.
    EVASION) {
32                              bookNodeCombat.setEvasionBookNodeLink(bookNodeLink);
33                          } else if(nodeLinkDialog.getLinkType() == NodeLinkDialog
    .PERDRE) {
34                              bookNodeCombat.setLooseBookNodeLink(bookNodeLink);
35                          } else if(nodeLinkDialog.getLinkType() == NodeLinkDialog
    .GAGNE) {
36                              bookNodeCombat.setWinBookNodeLink(bookNodeLink);
37                          }
38
39                          createNodeLink(bookNodeLink, selectedNodeFx, nodeFx);
40                      } else {
41                          book.addNodeLink(bookNodeLink, (
    AbstractBookNodeWithChoices) selectedNodeFx.getNode());
42                      }
43                  }
44
45
46                  selectedNodeFx = null;
47              }
48          }
49
```

Listing 5.18 – Classe NodeFxListener avec le mode ADD NODE LINK

La méthode *addNodeLink()* ajoute donc le lien puis appel l'observeur **BookNodeLinkObservable** permettant de notifier *nodeLinkAdded()* de la classe **GraphPane**. Cette méthode permet de trouver le premier et le deuxième noeud parmi la liste des noeuds existant. Dès qu'ils ont été trouvé, elle fait appel à la méthode *createNodeLink()* permettant de créer un lien entre les deux noeuds. Pour cela, un NodeLinkFx est d'abord créer permettant de pouvoir changer de valer si un des noeuds changeait d'emplement ainsi que de changer de position en fonction du zoom. Ce lien est ensuite ajouter dans une liste de lien permettant de le retrouver.

```
1     public NodeLinkFx createNodeLink(BookNodeLink bookNodeLink, NodeFx
    firstNodeFx, NodeFx secondNodeFx) {
2         NodeLinkFx nodeLinkFx = new NodeLinkFx(bookNodeLink, firstNodeFx,
    secondNodeFx, zoom);
3         nodeLinkFx.addNodeLinkFxObserver(new NodeLinkFxListener());
4
5         // Lie la line avec les deux autres noeuds
6         nodeLinkFx.startXProperty().bind(firstNodeFx.xProperty().add(firstNodeFx
    .widthProperty().divide(2)));
7         nodeLinkFx.startYProperty().bind(firstNodeFx.yProperty().add(firstNodeFx
    .heightProperty().divide(2)));
8
9         nodeLinkFx.endXProperty().bind(secondNodeFx.xProperty().add(secondNodeFx
    .widthProperty().divide(2)));
10        nodeLinkFx.endYProperty().bind(secondNodeFx.yProperty().add(secondNodeFx
    .heightProperty().divide(2)));
11
12        listeNoeudLien.add(nodeLinkFx);
13
14        nodeLinkFx.registerComponent(rootPane);
15
16        return nodeLinkFx;
17    }
18
```

Listing 5.19 – nodeLinkAdded()

Si le mode "DELETE" est sélectionné la méthode *removeNode()* de la classe Book est appelé. Une boite de dialogue d'affiche demandant confirmation. Si l'utilisateur annule, rien ne se passe. S'il valide,

une boucle "for" est parcouru afin d'envoyer une notification dans le **BookNodeLinkObservable** et la méthode *notifyNodeLinkDeleted()* qui appel *nodeLinkDeleted()* permettant de supprimer les liens lié à ce noeud. Puis une autre notification est envoyé à la classe **BookNodeLinkObservable**, qui lui, supprime le noeud en question grace à *notifyNodeLinkDeleted()* qui appel *nodeLinkDeleted()*.

```
1         if(mode == Mode.DELETE) {
2             if(confirmDeleteDialog()){
3                 book.removeNode(nodeFx.getNode());
4             }
5         }
6
```

Listing 5.20 – Classe NodeFxListener avec le mode DELETE

Si le mode "FIRST_NODE" est sélectionné, le lien entre le prélude et le noeud sélectionné est ajouté. Pour cela, un controle est fait si c'est bien un noeud. Si oui, le tracer est fait entre le prélude et le noeud sélectionné. Le "premier noeud" est donc changé dans le livre.

```
1         public void setFirstNode(NodeFx newFirstNode) {
2             preludeFx.setFirstNode(newFirstNode);
3
4             if(newFirstNode == null) {
5                 preludeFxFirstNodeLine.setVisible(false);
6             } else {
7                 preludeFxFirstNodeLine.endXProperty().bind(newFirstNode.
    xProperty().add(newFirstNode.widthProperty().divide(2)));
8                 preludeFxFirstNodeLine.endYProperty().bind(newFirstNode.
    yProperty().add(newFirstNode.heightProperty().divide(2)));
9
10                preludeFxFirstNodeLine.setVisible(true);
11
12                if(!rootPane.getChildren().contains(preludeFxFirstNodeLine)) {
13                    rootPane.getChildren().add(preludeFxFirstNodeLine);
14                }
15
16                book.changeFirstNode(newFirstNode.getNode());
17            }
18        }
19
```

Listing 5.21 – Classe NodeFxListener avec le mode FIRST NODE

NodeLinkFxListener

Cette classe permet de savoir si un évenement a été détecté sur un lien à l'aide du **NodeLinkFx** lié à chaque lien.

Si le mode "SELECT" est sélectionné et un double clique est effectué sur un lien, le lien est donc modfiable. La boite de dialogue sur lien s'affiche. Si la boite de dialogue est validé, les liens sont mis à jour grâce à la méthode *removeNodeLink()* présente dans la classe **Book**. Cette méthode notifie le **BookNodeLinkObservable** grâce à la méthode *notifyNodeLinkEdited()* qui elle appel la méthode *nodeLinkEdited()* du **GraphPane**. Cette méthode permet de remplacer le lien existant dans le **NodeLinkFx** par le nouveau lien modifié.

```
1         if(mode == Mode.SELECT) {
2             if(event.getClickCount() == 2) {
3                 NodeLinkDialog nodeLinkDialog = new NodeLinkDialog(nodeLinkFx.
    getNodeLink(), nodeLinkFx.getStart().getNode(), book);
4
5                 // Si on a validé les modifications sur un lien
6                 if(nodeLinkDialog.isValid()) {
```

```
7                       nodeLinkDialog.getNodeLink().setDestination(book.
    getNodeIndex(nodeLinkFx.getEnd().getNode()));
8
9                       //On met à jour les liens pour un noeud de combat
10                      if(nodeLinkFx.getStart().getNode() instanceof BookNodeCombat
    ) {
11                          BookNodeCombat bookNodeCombat = (BookNodeCombat)
    nodeLinkFx.getStart().getNode();
12
13                          book.removeNodeLink(nodeLinkFx.getNodeLink());
14
15                          if(nodeLinkDialog.getLinkType() == NodeLinkDialog.
    EVASION) {
16                              bookNodeCombat.setEvasionBookNodeLink(nodeLinkDialog
    .getNodeLink());
17                          } else if(nodeLinkDialog.getLinkType() == NodeLinkDialog
    .PERDRE) {
18                              bookNodeCombat.setLooseBookNodeLink(nodeLinkDialog.
    getNodeLink());
19                          } else if(nodeLinkDialog.getLinkType() == NodeLinkDialog
    .GAGNE) {
20                              bookNodeCombat.setWinBookNodeLink(nodeLinkDialog.
    getNodeLink());
21                          }
22
23                          createNodeLink(nodeLinkDialog.getNodeLink(), nodeLinkFx.
    getStart(), nodeLinkFx.getEnd());
24                      } else {
25                          book.updateNodeLink(nodeLinkFx.getNodeLink(),
    nodeLinkDialog.getNodeLink());
26                      }
27                  }
28              }
29          }
30
```

Listing 5.22 – Classe NodeLinkFxListener avec le mode SELECT

Si le mode "DELETE" est sélectionné et un clique est observé et l'utilisateur confirme la supression par la une réponse positive sur la boite de dialogue, la méthode *removeNodeLink()* de la classe **Book** est appelé. Ces liens sont alors supprimer des choix du noeuds considéré en "premier noeud". Puis le lien est alors supprimé grâce à la notification du **BookNodeLinkObservable** au **GraphPane**. Il est aussi supprimé de la liste des liens existants.

```
1           public void removeNodeLink(BookNodeLink nodeLink) {
2               List<AbstractBookNodeWithChoices> postRemove = new ArrayList<>();
3
4               for(Entry<Integer, AbstractBookNode> entry : this.nodes.entrySet())
    {
5                   if(!(entry.getValue() instanceof AbstractBookNodeWithChoices))
6                       continue;
7
8                   AbstractBookNodeWithChoices currentChoice = (
    AbstractBookNodeWithChoices) entry.getValue();
9                   for(BookNodeLink nl : entry.getValue().getChoices()) {
10                      if(nl == nodeLink) {
11                          postRemove.add(currentChoice);
12                          break;
13                      }
14                  }
15              }
16
17              for(AbstractBookNodeWithChoices node : postRemove) {
```

```
18                  node.removeChoice(nodeLink);
19              }
20
21              bookNodeLinkObservable.notifyNodeLinkDeleted(nodeLink);
22          }
23
```

Listing 5.23 – Classe NodeLinkFxListener avec le mode DELETE

iv RightPane

Ce panel contient tout ce qui représente les statistiques. Dès qu'un noeud est ajouté ou supprimé, une méthode de la classe **RightPane** est utilisé afin de mettre à jour le nombre de noeuds existant. Cette mise à jour est possible grâce au Observeur et Obervable entre le **GraphPane** et le **Book**. En effet, nous venons de voir que le **GraphPane** appelait une méthode dans le **Book**. Puis celui-ci notifiait grâce au **BookNodeObservable** le **GraphPane** de l'ajout/modfication/suppression de ce noeud. Mais en réalité elle notifie le **GraphPane** et le **RightPane**. Grâce à ce **BookNodeObserver**, les statistiques sont donc mis à jours en ajoutant ou en enlevant 1 aux types de noeuds.

Pour la difficulté du livre, elle est mise à jours dès que **Estimé la difficulté du livre** est sélectionné grâce à la méthode *difficultyChanged()* qui se situe dans la classe **RightPane**. Le pourcentage de difficulté est donc en paramètre de la méthode permettant à celle-ci de la mêttre à jour.

Si un nouveau livre est changé, tout ces statistiques ce remmettent à zéro grâce au *setBook()*.

D Rendre le livre jouable et estimer sa difficulté

i Jeu

Une classe a été créée se nommant **Jeu**, permettant de gérer les méthodes de jeu communes entre les classes *Player* et *Fourmis*. Cela permet donc d'avoir une classe qui gère tout le jeu en faisant appel aux différentes méthodes et classes en fonctions du types de noeud et du type de joueur.
Un construteur est d'abord appelé, à partir de la MainWindow, afin d'envoyer le livre contenant toutes les informations. Puis, en fonction du mode sélectionner (**Générer la difficulté** ou **Jouer**), la méthode correspondante au joueur est appelé.

Soit la méthode *play()* est appelé si **Jouer** est sélectionné. C'est donc le player qui va jouer. Cela veut dire que les messages vont alors s'afficher dans le terminal (*showMessages*) et le player est défini sur la classe *Player*. Cela permetra d'appeler le bon joueur lors de l'appélation des méthodes où un choix est nécessaires. Une fois le jeu lancer grâce au *runGame()*, expliqué dans le code 5.28, les ressources temporaires qui ont été généré sont supprimés.

```
1           public void play() throws IOException, BookFileException {
2               player = new Player();
3
4               showMessages = true;
5
6               runGame();
7
8               cleanUp();
9           }
10
```

Listing 5.24 – play()

Soit la méthode *fourmis()* est appelé si **Générer la difficulté** est sélectionné. Ce sont donc plusieurs fourmis qui vont jouer une par une afin de générer la difficulté du livre. Les messages sur le terminal sont donc non affiché. Puis vient le lancement des fourmis. Pour cela une boucle "for" est simplement necessaire. Le player est donc défini sur la classe **Fourmis** permettant ainsi d'utiliser les méthodes de cette classe afin de réaliser des choix random. Puis la méthode *runGame()*, expliqué dans le code 5.28, renvoi donc la finalité du livre : soit un boolean true en cas de victoire ou false en cas de défaite.
On a donc une classe qui permet réaliser des choix random ainsi qu'une méthode qui renvoie la finalité du livre. Cela permet donc de lancer un certain nombre de "fourmis" qui vont parcourir tout le livre, réalisé des choix, gagner ou perdre. Toutes les victoires vont être comptés, ce qui va permettre de savoir combien de fourmis ont gagnés. Soit calculé la difficulté du livre.

```
1            public float fourmis(int nbrFourmis) throws IOException,
   BookFileException {
2                showMessages = false;
3
4                int victoire = 0;
5                for(int i = 0 ; i < nbrFourmis ; i++){
6                    player = new Fourmi();
7
8                    if(runGame()) {
9                        victoire++;
10                   }
11               }
12
13               cleanUp();
14
15               return ((float)victoire / (float)nbrFourmis) * 100f;
16           }
17
```

Listing 5.25 – fourmis()

Une fois dans la méthode choisis, la méthode *runGame()* est alors appelé. C'est la méthode principal de la classe **Jeu** car c'est elle qui lance le jeu. Tout d'abord, la méthode copie le livre afin de ne pas le modifier par erreur pendant le jeu. Car même si on a fait attention, il suffit d'une erreur pour modifier tout le livre principal qui gère le jeu. Puis un BookState, qui correspond à l'état de la partie, est créé. Ce BookState permettera donc de suivre l'état du livre ainsi que du joueur (son inventaire, sa santé...). D'ailleur, le joueur enregistré dans le BookState est soit le personnage principal créer dans le prélude, soit, si inexistant, un personnage préexistant permettant ainsi de jouer au jeu sans pour autant créer un personnage principal (voir méthode si dessous).

```
1            BookCharacter bookCharacter;
2            BookState newState = new BookState();
3
4            if(this.book.getMainCharacter() == null){
5                //Création d'un personnage
6            }
7            else {
8                //Récupération du personnage principal
9            }
10
11           //Affichage pour le joueur du personnages principal
12
13           newState.setBook(this.book);
14
15           //Conception du personnage (ajout de compétences, items)
16
17           return newState;
18
```

Listing 5.26 – createNewState()

Enfin, si des compétences et/ou des items sont disponible au début de la partie, la méthode de création de joueur (*execPlayerCreation*, voir 5.27) est appelé en fonction du player actuel. Comme cela, le player choisit parmi une liste de compétences et d'items disponible en début de partie afin de les avoir pour commencer le jeu. Ci-dessous, vous pouvez voir le code permettant, ici, au player de choisir, d'après une liste, avec quels item ou compétences le joueur veut-il commencer le jeu. Cela permet ainsi d'avoir différentes finalité en fonction de ce que choisis le joueur.

```
1        public void execPlayerCreation(Book book, AbstractCharacterCreation
    characterCreation, BookState state){
2
3                if(characterCreation instanceof CharacterCreationItem){
4                    CharacterCreationItem characterCreationItem = (
    CharacterCreationItem) characterCreation;
5                    prendreItems(state, characterCreationItem.getItemLinks(),
    characterCreationItem.getAmountToPick());
6                }
7
8                else if(characterCreation instanceof CharacterCreationSkill){
9                    CharacterCreationSkill characterCreationSkill = (
    CharacterCreationSkill) characterCreation;
10
11                   int nbItemMax = characterCreationSkill.getAmountToPick();
12
13                   while(nbItemMax != 0 && !characterCreationSkill.
    getSkillLinks().isEmpty()){
14                       //Affiche les compétences disponibles
15
16                       skillAdd(state, characterCreationSkill);
17                       nbItemMax--;
18                   }
19               }
20           }
21
```

Listing 5.27 – Méthode execPlayerCreation() du player

Une fois le **BookState** créé, le personnage principal initialisé et la copie du livre enregistré, le premier noeud est donc chargé. La boucle while du *runGame()* est donc lancé et ne s'arrètera qu'une fois la partie terminée. Comme vous pouvez le remarquer, sur le code ci-dessous, chaque méthode correspond à un type de noeud en particulier. Les méthodes s'exécutent alors, puis renvoie le noeud de destination, en fonction du choix du joueur et / ou de la mort du joueur. Durant l'exécution des différentes méthodes et en fonction du joueur, d'autres méthodes externe sont appelées nottament dans la classe **Fourmis** ou **Player**.

```
1            while(!gameFinish){
2                if(currentNode instanceof BookNodeCombat){
3                    //execNodeCombat(bookNodeCombat);
4                }
5                else if(currentNode instanceof BookNodeWithChoices){
6                    //execNodeWithChoices(bookNodeWithChoices);
7                }
8                else if(currentNode instanceof BookNodeWithRandomChoices){
9                    //execNodeWithRandomChoices(bookNodeWithRandomChoices);
10               }
11               else if(currentNode instanceof BookNodeTerminal){
12                   //execNodeTerminal(bookNodeTerminal);
13                   //Si partie gagné, win = true
14               } else {
15                   //BookNodeTerminal FAILURE
16               }
17           }
```

```
18              return win;
19
```

Listing 5.28 – Méthode runGame()

Pour chaque type de noeud, sauf pour un noeud terminal car il ne contient pas ces options, une méthode commune, nommé *execAbstractNodeWithChoices()* est appelé afin de savoir si le noeud pris en charge fait gagné/perdre de la vie (méthode *execNodeHp()*), puis regarde si le joueur est toujours en vie. Si ce dernier n'est plus en vie, un noeud terminal de défaite est alors renvoyé en noeud de destination. S'il est encore en vie et que le noeud propose des items, ils sont proposés au joueur en appelant la méthode correspondante entre **Fourmis** ou **Player**. Ces classes sont appelé car le player doit faire un choix grâce au Scanner et la fourmis doit effectuer un choix random.

```
1         public AbstractBookNode execAbstractNodeWithChoices(
      AbstractBookNodeWithChoices node){
2             //Affichage pour le joueur
3
4             execNodeHp(node);
5
6             if(!state.getMainCharacter().isAlive()){
7                 //Création du noeud terminal de défaite
8                 return nodeTerminal;
9             }
10
11            if(!node.getItemLinks().isEmpty())
12                chooseItems(node);
13
14            return null;
15        }
16
```

Listing 5.29 – Méthode execAbstractNodeWithChoices()

Puis, à chaque destination du joueur, sauf si celui ci a été entrainer vers un noeud terminal de défaite créer par la classe jeu et donc, non existante, une autre méthode est appelé (nommé *execBookNodeLink()*) afin de regarder si le lien entre le noeud de départ et de destination fait perdre ou gagner de la vie et/ou de l'argent. Si c'est le cas, le joueur est alors "mis à jour". Si le joueur n'a plus de vie, même si le lien le portait vers un noeud de victoire, un noeud de défaite est créé. Le joueur à alors perdu.

Si un noeud est de type basic, il est alors pris en charge dans la méthode *execNodeWithChoices()*, comme vous pouvez le constater dans la figure 5.19.
Tout d'abord, comme tout noeuds non terminaux, la méthode fait appel à *execAbstractNodeWithChoices()*, vu un peu plus haut (voir 5.29). Puis, vérifie si un choix est possible : si le joueur n'a aucun choix ou s'il ne possède pas les items/compétences pour aller vers ce choix. Si ce n'est pas le cas, la méthode renvoi un noeud terminal de défaite. Ainsi, le joueur n'est pas bloqué. Dans le cas contraire, s'il peut au moins choisir une destination, un choix est demandé parmis toutes les destinations faisant un appel à la méthode en fonction du joueur (*player.makeAchoice()*). Car ici, un choix est demandé, soit le Scanner du player ou le random de la fourmis. Une fois le choix décidé, si le joueur a les prérequis pour aller vers cette destination (si *isAvailable()* renvoie true), alors la méthode *execBookNodeLink()*, vu juste en dessous de la figure du haut, est appelé. Si le joueur est toujours en vie, le noeud choisi est renvoyé en noeud de destination. Sinon, le joueur doit faire un autre choix car il y a minimum un choix possible.

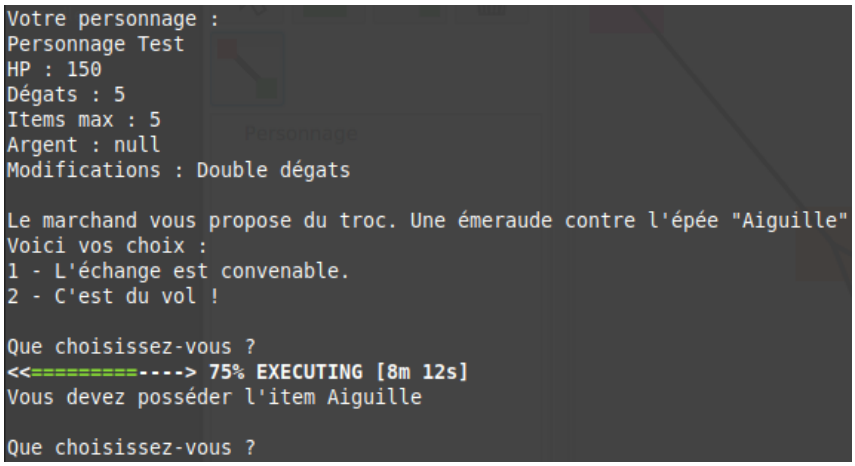

FIGURE 5.17 – Jeu prérequis

Le problème de la méthode *isAvailable()*, c'est que cela peut créer un livre sans fin. En effet, prenons l'exemple d'un joueur avec un espace inventaire maximal de 1. Le joueur a un item, dans le premier noeud, "Potion" et décide de prendre l'item. Il n'a donc plus de place dans son inventaire. Le deuxième noeud, il décide de prendre l'item "Arme" car il veut pouvoir se défendre, l'item "Potion" est donc supprimé. Sauf qu'il se retouve en face de deux choix : Choix 1 avec de prérequi "Potion" qu'il vient de suprimmer, ainsi que le choix 2 qui le renvoi sur ce noeud. Le problème c'est que la méthode *isAvailable()* renvoie true, car le joueur peut accéder au choix 2. Le joueur est donc coincé. Cela pose un problème que cela soit dans la génération de la difficulté et la jouabilité du livre.
Cela ouvre donc une piste de réflexion. Il faudrait peut être envoyer des fourmis à chaque "destination" de noeud trouvé. Si une fourmis trouve une boucle infini (retombe sur un même noeud "n" fois, avec un nombre assez grand permettant d'être sûr que c'est bien une boucle infini), elle renvoie un false et le numéro du noeud. Si le joueur tombe sur ce paragraphe, un noeud terminal défaite est généré en indiquant sur celui ci "le livre n'est plus possible, il vous manque [prerequi]". Ces fourmis et ce boolean sera généré a chaque destination chargé. La fourmis sera la copie exacte du personnage au moment même du chargement du noeud de destination. Mais cette méthode est trop importante et une autre, plus simple, serait peut être possible.

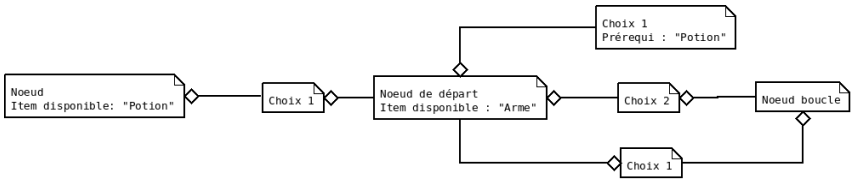

FIGURE 5.18 – Exemple de boucle infini

Une fois qu'au moins une destination est possible, nous avons décidé de ne pas afficher que les choix possibles afin de ne pas aveugler le joueur de ses possibilités. S'il veut rejouer au livre, il saura les prérequis de certain chemins.

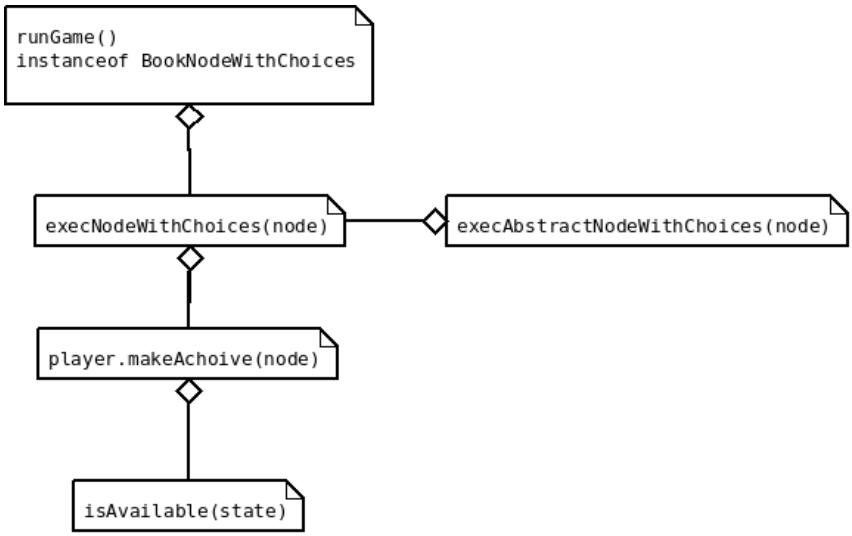

FIGURE 5.19 – Lien entre les méthodes pour un noeud basique

Si c'est un noeud de type combat, une vérification est réalisé afin savoir si le noeud contient des ennemis. S'il n'y a pas d'ennemis, le noeud en cas de victoire est envoyé en noeud de destination. S'il ny a pas de noeud de victoire, un noeud de défaite est automatiquement transmit afin de ne pas avoir d'erreur sur ce noeud. Si une liste d'ennemis est présente, cette dernière est copié est créé afin de ne pas modifier la vie des ennemis. Car ces derniers ne sont pas lié au noeud, mais c'est l'ID de l'ennemi qui est lié au noeud. Cela permet donc de les appelés plusieurs fois dans plusieurs ou dans le même noeud.
Le combat commence alors. Le choix est défini par la méthode du joueur correspondant car un Scanner ou un random est requis pour ce type de choix. Ici, trois choix sont possibles :

Choix Attaque : que ce soit dans la classe **Player** ou **Fourmis**, un autre choix (cf ligne 2) est demandé permettant de sélectionner l'ennemi à attaquer parmi la liste des ennemis encore en vie. Cela permet donc de créer un dynamisme et une certaine intélligence dans le combat. Je vous invite à voir les subsection fourmis iv et player iii pour plus de détail dans le choix. Une fois l'ennemi sélectionné, une méthode attaque (cf ligne 7) est appelé apellant elle même une autre méthode commune entre l'attaque du joueur et l'attaque d'ennemi. Nommé *getDamageAmount()*, elle permet de savoir le nombre de dommage réalisé en fonction des point d'attaque de l'attaquant, de son double dommage décider en random si ce boolean est défini en true, d'un coup critique décider aussi en random, de l'arme de l'attaquant et de l'item de défense de l'attaqué. L'attaquant et l'attaqué est défini en fonction de la permière méthode qui l'appel. Ici c'est la méthode d'attaque du player. Cette méthode commune permet donc d'éviter une répétition dans le code. Une fois l'attaque effectué, si l'ennemi attaqué est mort, il est supprimé de la liste des ennemis (cv logne 9) avant sont tour pour éviter que l'ennemi mort attaque ou que l'on rattaque cette ennemi.

Inventaire : si ce choix est fait par le joueur, la méthode appelé permettant d'utiliser son inventaire est elle même gérer dans la classe Player ou la classe Fourmis. Elle permet alors de choisir une potion, une arme et ou un item de défense. Pour plus de détail sur ce choix, je vous invite à regarder les subsection fourmis iv et player iii.

Evasion : si le tour avant évasion est inférieur ou égal à 0 (voir cf 13) et si un noeud de d'évasion existe, le joueur peut alors s'enfuir. Sinon cela lui passe son tour.Nous avons décidé de ne pas relancer les choix à ce moment, car cela peut être une stratégie du joueur qui, dans une prochaine mise à jour d'implentation de pouvoir lié à des compétences, en cas d'une compétence de "renvoi" (qui se désactive en cas d'attaque de la part du joueur), l'attaque de l'ennemi est alors "retourné" vers lui même.

```
        while(!finCombat){
            ChoixCombat choixCombat = player.combatChoice(node, evasionRound,
    state);

            if (choixCombat == ChoixCombat.ATTAQUER) {

                BookCharacter ennemi = player.chooseEnnemi(listEnnemis);
                attaque(ennemi);
                if(!ennemi.isAlive()){
                    listEnnemis.remove(ennemi);
                }
            }
            else if(choixCombat == ChoixCombat.EVASION) {
                if(evasionRound <= 0 && node.getEvasionBookNodeLink() != null){
                    execBookNodeLink(node.getEvasionBookNodeLink());
                    return book.getNodes().get(node.getEvasionBookNodeLink().
    getDestination());
                }
            }

            ennemiTour(listEnnemis);

            if(!state.getMainCharacter().isAlive()) {
                //Vérification d'un noeud Failure
                if(node.getLooseBookNodeLink() != null)  {
                    execBookNodeLink(node.getLooseBookNodeLink());
                    return book.getNodes().get(node.getLooseBookNodeLink().
    getDestination());
                } else
                    return new BookNodeTerminal("Vous succombez à vos blessures
    ", BookNodeStatus.FAILURE);
            }

            if(listEnnemis.isEmpty())
                finCombat = true;

            evasionRound -= 1;
            }

```

Listing 5.30 – JeuCombat

Une fois le tour du joueur fini, vient le tour de l'ennemi. Il appel une méthode envoyant la liste d'ennemis restant. Cette méthode appel *getDamageAmount()* permettant aux ennemis d'attaquer un par un. Cette liste d'ennemis et la méthode commune, *getDamageAmount()*, est alors très pratique pour l'attaque de l'ennemis et est, on pense, le meilleur moyen utiliser afin de gérer les attaques.

La fin de combat est déterminé si la liste d'ennemis est vide ou si le joueur n'est plus en vie. Le noeud de destination est alors défini en fonction du résultat en fin de combat. Bien sur, si le noeud de destination n'est pas existant, la noeud terminal de défaite est créer pour éviter de bloquer le jeu.

Si le noeud est de type aléatoire, la méthode commune est appelé afin de savoir si le joueur est encore en vie. Puis une autre méthode, nommé *getRandomChoices()* (Voir 5.1 à la page 12) est appelé de la classe BookNodeLinkRandom afin de déterminer le noeud de destination en fonction des chances attribué à chacun de ces choix.

Si le noeud est de type terminal, la partie est alors terminé et renvoie un boolean sur l'état de la fin de partie. Ce boolean permet de quitter la boucle while de la méthode *runGame()* et permet ainsi donc d'arreter le jeu. Cela permet de continuer l'édition du livre pour le player et permet de savoir si la partie est gagné ou perdu pour la fourmis (voir la subsection fourmis iv).

ii Interface Player / Foumis

Une interface **InterfacePlayerFoumis** a été créé permettant une mise en commun des codes **Player** et **Fourmis**. Ces méthodes permettent de faire un choix, prendre les items disponibles, créer un personage lambda, aller dans l'inventaire, choisir son ennemis ou encore combatre. Elles sont appelé au même moment. La méthode sera alors exécuté différément en fonction du joueur. Ce sont des méthode communes car un choix est demandé, et la décision du choix est différentes en fonction d'un joueur ou d'une fourmis. En effet, un Scanner est utilisé afin de prendre en compte le choix du player et un nombre aléatoire est généré pour la fourmis.

Si dessous, toutes les méthodes communes sont présentés de façon général. Mais elles seront toutes expliquer dans les subsection fourmis iv et playeriii, vu un peu plus bas.

execPlayerCreation permet de choisir les compétences et les items disponibles au début de la partie. Ces derniers sont défini lors de la création du prélude. Pour l'ajout des items, la méthode *prendItems* est appelé. Cette méthode permet donc de jouer au livre avec peut être, différentes finalités en fonction de ces choix.

combatChoice(), prend en paramètre le noeud de Combat et le nombre de tour avant l'évasion ainsi que le BookState. Elle permet de faire un choix lors du tour du joueur dans un combat. On peut alors choisir d'attaquer, d'aller dans l'inventaire ou alors de s'évader. Si on choisi l'inventaire, on va alors dans une autre méthode appelé *useInventaire()* qui prend le BookState en paramètre. On peut alors utiliser une potion, prendre un objet de défense ou alors une arme. Si l'on choisis un autre choix, cette objet n'est pas utilisable lors d'un combat (comme par exemple de l'argent). Une fois l'objet pris, on retourne dans les choix du combat. On peut alors, soit retourner dans l'inventaire pour prendre un autre objet, soit attaquer ou s'évader. Cela donne donc libre choix au joueur sur le nombre d'objet à utiliser (comme par exemple, l'ajout d'une arme ainsi qu'un bouclier).

chooseEnnemi() permet de choisir l'ennemi à attaquer parmi la liste de tout les ennemis encore en vie. Cela rend le système de combat un peu plus "intelligent".

prendItems() permet de prendre un item parmi la liste d'items disponible. Cette liste est pris en paramètre ainsi que la sauvegarde de la partie et le nombre d'item maximum pouvant être pris. Cela permet ainsi d'avoir une méthode complète qui peut être utiliser pour prendre les items disponibles aux prélude comme sur les différents noeuds du livre.

makeAChoice() permet de faire un choix en fonction des différentes destinations proposé par le noeud. Cette méthode est ce qui défini un "livre à choix".

useIventaire() permet d'utiliser son inventaire lors d'un noeud de combat. La mise à jour d'un port d'item de défense ou d'arme est alors mis à jour. Si un item de soin est choisi, les points de vie du joueur sont alors actualisée.

Malgré une séparation entre le code en "communs" visible dans la classe **Jeu** et le code normalement "non communs" visible dans les classes **Player** et **Fourmis**, vous allez remarquer dans l'explication de ses classes, qu'une partie est encore commune. Ce problème n'a pas été réglé par manque de temps.

iii Player

La classe **Player** permet de jouer au jeu en tant que player. Elle permet de faire des choix grâce aux Scanner permettant ainsi l'avancement du jeu. Des messages sont aussi affiché afin de guider le player dans ses choix.
Nottament la méthode *choixYesNo()* qui permet de choisir Oui ou Non et de renvoyer le boolean true ou false. Cette méthode permet, par exemple, de savoir si le player veut supprimer, prendre un item ou une compétence. Elle a donc été créé évitant ainsi la redondance de code.

Pour la méthode commune *prendItems()*, cette dernière fait appel à d'autre méthodes présente dans la classe **Player**.

Par exemple, *itemAdd()* (cf ligne 20 dans le code 5.34) qui elle permet de choisir l'item à ajouter dans l'inventaire. Une variable "choixValide" permet de ne pas valider un numéro de choix non valide. Le player peut donc alors répondre entre 0 et le nombre d'items disponibles ou "-1". Le "-1" permet d'annuler l'ajout d'item. Une fois l'item choisi, l'ID de l'item est alors ajouter dans la liste de l'inventaire afin de pouvoir retrouver l'item si le joueur l'utilise.
Cette méthode a été créé permettant de séparer le code de la méthode *prendItems()* pour une meilleure visibilité. Elle pourrait aussi servir, par exemple, dans l'ajout d'un item obligatoire. Dans ce cas, la méthode *prendItems()* n'est plus uilisé mais directement *itemAdd()*. Mais cette exemple n'est pas représentatif de l'état actuel de MagicBook qui ne permet pas de rendre un item obligatoire. Mais c'est une idée d'amélioration.

```
1  private void itemAdd(BookState state, List<BookItemLink> bookItemLinks){
2      System.out.println("Quel item voulez-vous ?");
3      boolean choixValide = false;
4      int choix = -1;
5
6      while(!choixValide){
7          Scanner scanner = new Scanner(System.in);
8          choix = scanner.nextInt();
9
10         if(choix >= 0 && choix <= (bookItemLinks.size()-1)){
11             choixValide = true;
12         } else {
13             System.out.println("vous ne pouvez pas effectuer ce choix");
14         }
15     }
16
17     BookItemLink itemLink = bookItemLinks.get(choix);
18     System.out.println("L'item "+state.getBook().getItems().get(itemLink.getId()
       ).getName()+" a été rajouté");
19     state.getMainCharacter().getItems().add(itemLink.getId());
20
21     itemLink.setAmount(itemLink.getAmount()-1);
22
23     if(itemLink.getAmount() == 0)
24         bookItemLinks.remove(itemLink);
25 }
```

Listing 5.31 – itemAdd()

La méthode *itemPlein()* (cf ligne 11 dans le code 5.34), qui elle, affiche les items à supprimer.

```
1  private void itemPlein(BookState state){
2      System.out.println("Votre inventaire est plein");
3      System.out.println("Voulez vous supprimer un item ?");
4      System.out.println("Vos Item: ");
5
6      int i = 0;
7      for(String itemState : state.getMainCharacter().getItems()){
8          System.out.println(i + " - "+state.getBook().getItems().get(itemState));
9          i++;
10     }
11 }
```

Listing 5.32 – itemPlein()

Un choix *choixYesNo()* est alors posé. Si le player répond oui (cf ligne 15 dans le code 5.34), la méthode *itemSupp()* (cf ligne 18 dans le code 5.34) est appelé afin de choisir l'item à supprimer. Une

boucle while est alors parcouru afin d'être sûr d'avoir un choix valide. Le player peut donc alors répondre entre 0 et le nombre d'item dans l'inventaire ou "-1". Cette dernière valeur permet d'annuler la supression d'item et de retourner au choix de l'item. Une fois l'item choisi, cela met l'inventaire du player à jour grâce au state pris en charge dans la méthode *prendreItems()*. S'il répond non au choix *choixYesNo()*, la boucle (lancé en cf ligne 2 dans le code 5.34) est arrété, le jeu continu.

```
1  private void itemSupp(BookState state){
2      System.out.println("Quel item voulez-vous supprimer ?");
3      boolean choixValide = false;
4
5      while(!choixValide){
6          Scanner scanner = new Scanner(System.in);
7          int choix = scanner.nextInt();
8
9          if(choix >= 0 && choix <= (state.getMainCharacter().getItems().size()-1)
       ){
10             state.getMainCharacter().getItems().remove(choix);
11             choixValide = true;
12         } else if(choix == -1) {
13             choixValide = true;
14         } else {
15             System.out.println("vous ne pouvez pas effectuer ce choix");
16         }
17     }
18
19 }
```

Listing 5.33 – itemSupp()

Bien sûr cette méthode n'est appelé que si l'inventaire est plein. Autrement, *itemAdd()* est directement appelé. On aurait pu très bien laisser le choix au player de supprimer quand même un item si son inventaire n'était pas plein. Mais c'est un livre à choix avec des prérequis, tout les items peuvent donc être important, et il n'y a pas de pénalité de déplacement si l'inventaire est plein. Cette fonctionalité de supprimer un item dans l'inventaire n'a alors pas été jugé nécessaire à ajouter en option libre.

```
1  public void prendreItems(BookState state, List<BookItemLink> bookItemLinks, int
       nbItemMax){
2      while(nbItemMax != 0 && !bookItemLinks.isEmpty()){
3          System.out.println("Les items suivant sont disponible:");
4          //Affiche les items
5
6          System.out.println("Voulez vous un item ?");
7          if(choixYesNo()){
8              int itemMax = state.getMainCharacter().getItemsMax();
9
10             if(itemMax == state.getMainCharacter().getItems().size()){
11                 itemPlein(state);
12
13                 System.out.println("Voici vos choix:");
14
15                 if(choixYesNo())
16                     nbItemMax = 0;
17                 else
18                     itemSupp(state);
19             } else {
20                 itemAdd(state, bookItemLinks);
21             }
22
23         } else {
24             nbItemMax = 0;
25         }
26
27         if(bookItemLinks.isEmpty())
```

```
28              nbItemMax = 0;
29      }
30  }
```

Listing 5.34 – prendreItems() du Player

Pour la méthode *execPlayerCreation()* est est presque la même que celle de la fourmis : elle utilise aussi la méthode *prendItems()*, défini ici5.34 au moment de l'ajout des items. Mais au moment de l'ajout de compétences, la méthode *skillAdd()* est appelé jusqu'à ce que le maximum de compétences autorisé a été pris ou qu'il n'en reste plus à prendre.

```
1  public void execPlayerCreation(Book book, AbstractCharacterCreation
       characterCreation, BookState state){
2      //Affichage du texte du début dans "création du personnage"
3
4      if(characterCreation instanceof CharacterCreationItem){
5          CharacterCreationItem characterCreationItem = (CharacterCreationItem)
       characterCreation;
6          prendreItems(state, characterCreationItem.getItemLinks(),
       characterCreationItem.getAmountToPick());
7      }
8      else if(characterCreation instanceof CharacterCreationSkill){
9          CharacterCreationSkill characterCreationSkill = (CharacterCreationSkill)
        characterCreation;
10
11         int nbItemMax = characterCreationSkill.getAmountToPick();
12
13         while(nbItemMax != 0 && !characterCreationSkill.getSkillLinks().isEmpty
       ()){
14             System.out.println("Les compétences suivant sont disponible:");
15             //Affichage des compétences
16             }
17
18             skillAdd(state, characterCreationSkill);
19             nbItemMax--;
20         }
21     }
22 }
```

Listing 5.35 – execPlayerCreation() du Player

Le player doit alors confirmer ou non s'il veut une compétence grâce à la méthode *choixYesNo()*, permettant ainsi de ne pas obliger le joueur à prendre une compétence. Si "true" est retourné, le choix de la compétence est alors demandé, puis celle-ci est ajouté. Si "false" est retourné, la liste est supprimé. Il n'y a donc plus de compétences à prendre, le jeu continue.

```
1  private void skillAdd(BookState state, CharacterCreationSkill
       characterCreationSkill){
2      System.out.println("Voulez vous un skill ?");
3      if(choixYesNo()){
4          System.out.println("Quel skill voulez-vous ?");
5          boolean choixValide = false;
6          int choix = -1;
7
8          while(!choixValide){
9              Scanner scanner = new Scanner(System.in);
10             choix = scanner.nextInt();
11
12             if(choix >= 0 && choix <= (characterCreationSkill.getSkillLinks().
       size()-1)){
13                 choixValide = true;
14             } else {
```

```
                System.out.println("vous ne pouvez pas effectuer ce choix");
            }
        }

        String skill = characterCreationSkill.getSkillLinks().get(choix);
        state.getMainCharacter().addSkill(skill);
        System.out.println("Le skill "+skill+" a été rajouté");
        characterCreationSkill.getSkillLinks().remove(skill);

        characterCreationSkill.setAmountToPick(characterCreationSkill.
    getAmountToPick()-1);
    } else {
        characterCreationSkill.getSkillLinks().clear();
    }
}
```

Listing 5.36 – skillAdd()

iv Fourmis

La classe **Fourmis** permet de parcourir le livre en tant que joueur fictif. Elle effectue des choix random en fonction des différentes méthodes de l'interface. Cela permet donc de terminer le livre grâce aux choix random.

La méthode *combatChoice* permet de choisir entre ATTAQUE, EVASION, INVENTAIRE. Nous avons choisi de faire un random sur les trois choix et non pas sur deux choix même si le tour d'évasion n'est pas disponible, permettant ainsi d'avoir la possibilité que la fourmi passe aussi son tour, comme le joueur, afin d'avoir la même chance lors des combats. Elle peut aussi utiliser son inventaire lors du combat.

```
public ChoixCombat combatChoice(BookNodeCombat bookNodeCombat, int
    remainingRoundBeforeEvasion, BookState state) {
    boolean choixValide = false;
    Random random = new Random();
    ChoixCombat choixCombat = null;
    int choix;

    while(!choixValide){
        choix = random.nextInt(ChoixCombat.values().length);
        choixCombat = ChoixCombat.values()[choix];

        if (choixCombat == ChoixCombat.INVENTAIRE){
            if(!state.getMainCharacter().getItems().isEmpty())
                useInventaire(state);
        } else {
            choixValide = true;
        }
    }

    return choixCombat;
}
```

Listing 5.37 – combatChoice() de Fourmis

Ainsi, elle peut se soigner, prendre des armes et/ou des items de défense. Malheureusement, la fourmis n'est pas "intelligente" et ne prend pas donc la meilleur arme ou ne se soigne pas alors qu'elle va mourir. Mais c'est un gros point à améliorer. Elle choisi donc au hasard un des items et peut même prendre un items non utile dans le combat. Elle revient donc sur les trois choix ATTAQUE, EVASION, INVENTAIRE sans avoir pris une arme ou autre lors du choix de l'item dans son inventaire.

```java
public void useInventaire(BookState state){
    List<String> listItemState = state.getMainCharacter().getItems();

    Random random = new Random();
    int choix = random.nextInt(listItemState.size());
    BookItem bookItem = state.getBook().getItems().get(listItemState.get(choix))
    ;

    if(bookItem instanceof BookItemDefense){
        state.setBookItemDefense((BookItemDefense) bookItem);
    }
    else if(bookItem instanceof BookItemHealing){
        BookItemHealing bookItemHealing = (BookItemHealing) bookItem;
        state.getMainCharacter().heal(bookItemHealing.getHp());
        state.getMainCharacter().getItems().remove(listItemState.get(choix));
    }
    else if(bookItem instanceof BookItemWeapon){
        state.setBookItemArme((BookItemWeapon) bookItem);
    }
}
```

Listing 5.38 – useInventaire() de Fourmis

Autre point, lors de l'utilisation de la méthode *prendreItems()*, la fourmis prend tout les items disponible, tant qu'elle a encore de la place, que le maximum d'objet à prendre n'est pas à 0 et si des objets sont encore disponible.

```java
public void prendreItems(BookState state, List<BookItemLink> bookItemLinks, int
    nbItemMax){
    int choix = 0;

    while(nbItemMax != 0){
        int itemMax = state.getMainCharacter().getItemsMax();
        //Vérifie item maximum et si il y a encore des items à prendre
        if(state.getMainCharacter().getItems().size() < itemMax && !
    bookItemLinks.isEmpty()){
            //Choix
            Random random = new Random();
            choix = random.nextInt(bookItemLinks.size());

            BookItemLink itemLink = bookItemLinks.get(choix);

            state.getMainCharacter().addItem(itemLink.getId());
            itemLink.setAmount(itemLink.getAmount()-1);

            if(itemLink.getAmount() == 0)
                bookItemLinks.remove(itemLink);

            if(nbItemMax != -1)
                nbItemMax--;
        } else {
            nbItemMax = 0;
        }
    }
}
```

Listing 5.39 – prendreItems() de Fourmis

Des boucles infini sont donc à prévoir si par exemple, un objet est requis pour accéder au choix 2 et qu'il faut aller dans le choix 1 pour prendre l'item. Vu qu'au moins un choix est disponible, aucun noeud terminal n'est généré. Sauf que si l'item de la fourmis est plein, l'item ne peut pas être pris et aucun item est supprimé. Si la supression est active, quel est l'item à suprimmer ? Comment définir un item important ? A la santé du personnage, au nombre d'objet non possédé d'un certain type (exemple :

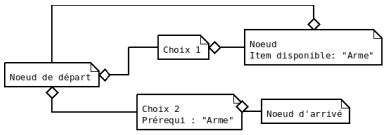

FIGURE 5.20 – prendreItem exmple de boucle infini

defense) ? A un nombre "d'importance" prédéfini à l'édition du livre ? Cette piste de réflexion serait donc à exploiter dans les améliorations à apporter.

A la méthode *execPlayerCreation()*, soit, à la création du personnage, le même problème se pose. En effet, la méthode *prendreItems()* est utilisé. Pour l'ajout de compétence, le problème est moins important. En effet, les compétences ne peuvent pas être aprise autre part qu'à la création du personnage. En effet, c'est une autre amélioration prévu. Nous avons donc décidé d'ajouter le nombre maximal de compétence. Mais si les compétences avait des effets, comment définir qu'une compétence est plus importante qu'une autre ? Peur être avoir aussi un nombre prédéfini, comme les items, qui désigne son importance. Puis copier la méthode pour choisir un noeud aléatoire. Il faudrait aditionner tout les nombres désignant l'importance, choisir un nombre aléatoire, puis soustraire chaque nombres désignant l'importance jusqu'à être égal ou inférieur à zero. Ces une piste de réflexion possible, mais pas la seule solution.

```
1  public void execPlayerCreation(Book book, AbstractCharacterCreation
       characterCreation, BookState state) {
2
3      if(characterCreation instanceof CharacterCreationItem){
4          CharacterCreationItem characterCreationItem = (CharacterCreationItem)
       characterCreation;
5
6          prendreItems(state, characterCreationItem.getItemLinks(),
       characterCreationItem.getAmountToPick());
7      }
8      else if(characterCreation instanceof CharacterCreationSkill){
9          CharacterCreationSkill characterCreationSkill = (CharacterCreationSkill)
        characterCreation;
10
11         Random random = new Random();
12         int amountToPick = characterCreationSkill.getAmountToPick();
13         while(amountToPick != 0 && !characterCreationSkill.getSkillLinks().
       isEmpty()) {
14             int choix = random.nextInt(characterCreationSkill.getSkillLinks().
       size());
15             state.getMainCharacter().addSkill(characterCreationSkill.
       getSkillLinks().get(choix));
16             characterCreationSkill.getSkillLinks().remove(choix);
17             amountToPick--;
18         }
19     }
20 }
```

Listing 5.40 – execPlayerCreation() de Fourmis

Pour la méthode *chooseEnnemi*, la fourmi envoyé prend obligatoirement le premier ennemi permettant de tuer le maximum d'ennemis en attaquant toujours le même ennemi. Nous avons choisi cette solution pour éviter que la fourmi attaque tout les ennemis sans en tuer un seul. Sauf qu'à chaque tour, chaque ennemi attaque. Donc moins il y a d'ennemis, plus sont les chances de remporter le combat.
Mais un autre problème se pose. Imaginons une fourmi avec 15 points de vie 10 points d'attaque. Imaginons une liste ennemis : [20pts,5pts,5pts,5pts]. Si le premier ennemi à 20 point de vie et les troix suivant ont 5 points de vie, serait-il plus judicieux alors de tuer les troix ennemis avec 5 points de vie avant d'attaquer le premier ennemis avec 20 points de vie. Probablement oui. Voyons le problème de plus près.

- Attaque de la fourmis : deuxième de la liste

- Reste liste ennemis : [20pts,5pts,5pts]
- Attaque des ennemis : Le premier ennemi attaque (-5 points de vie à la fourmi, soit 10pts vie restant), les autres attaque aussi (-0pts de vie à la fourmi, soit 10pts vie restant)

En tant que player, il changerait d'ennemis. La fourmi devra alors avoir l'intelligence de changer d'adversaire. Elle gagnera alors de combat.

- Attaque de la fourmis : premier de la liste
- Perte de 10pts de vie sur le premier ennemi
- Reste liste ennemis : [10pts,5pts,5pts]
- Attaque des ennemis : Le premier ennemi attaque (-5 points de vie à la fourmi, soit 5pts vie restant), les autres attaque aussi (-0pts de vie à la fourmi, soit 5pts vie restant)
- Attaque de la fourmis : premier de la liste
- Reste liste ennemis : [5pts,5pts]
- Attaque des ennemis : les autres attaque aussi (-0pts de vie à la fourmi, soit 5pts vie restant)
- ...

6 Conclusion

Ce projet a permis aux différents membres de s'améliorer, que ce soit dans l'apprentissage d'une librairie, la mise en application des concepts vus en cours ou bien encore, dans l'art de la procrastination. Bien que nous ayons été quatre sur le projet, deux de nos camarades n'ont pas fourni suffisamment d'aide dans celui-ci. Il a fallu leur demander à de nombreuses reprises de travailler, des modifications de quelques lignes pouvaient prendre jusqu'à trois semaines pour être rendues, tout en étant parfois incomplètes. Ainsi, pour nous, le groupe était un groupe constitué de deux personnes uniquement. Le projet a été très intéressant mais nous regrettons beaucoup de ne pas avoir pu le compléter et de ne pas avoir rendu le code aussi propre que ce que nous aurions voulu. Nous sommes, par exemple, déçue de ne pas avoir pu fournir certaines fonctionnalités qui sont prises en compte dans le jeu mais pas dans l'éditeur (gestions des skills, des prérequis, ...). L'application reste cependant fonctionnelle et plutôt complète.

7 Ressources utiles et sources utilisés