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.

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.

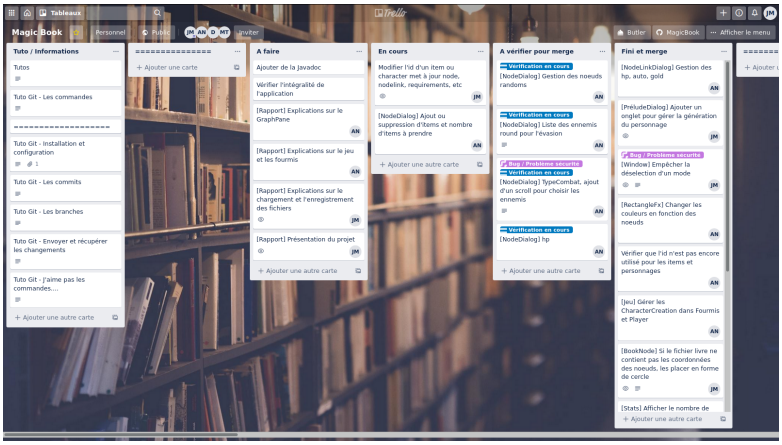

FIGURE 2.1 – Notre tableau Trello

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.

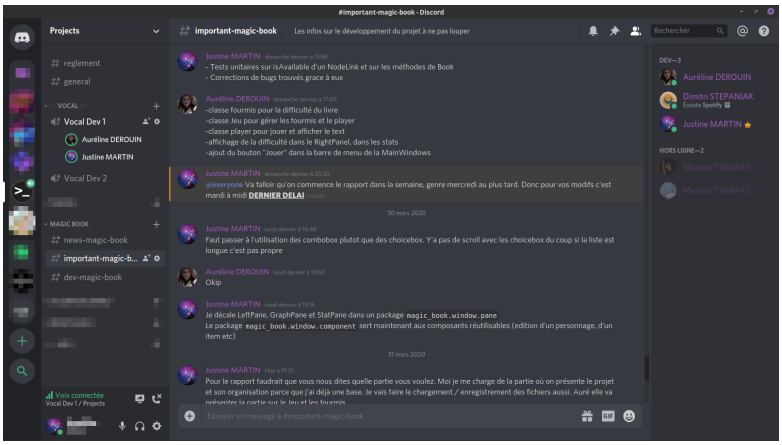

FIGURE 2.2 – Notre serveur Discord

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	
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 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	
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			
Autre				
Rapport	X		X	$\sim$
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 :

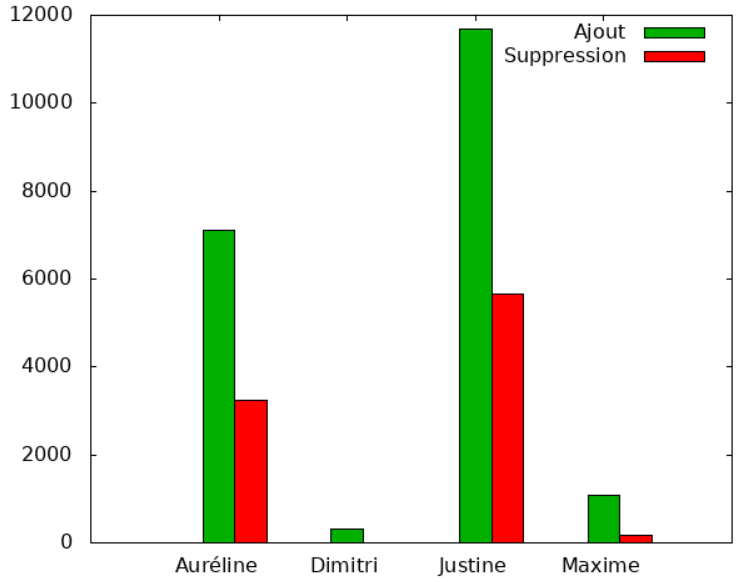

FIGURE 3.1 – Nombre de lignes ajoutés et supprimés par personne

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 17)
- 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

Il y a quatre types de noeuds représentées par des classes (Voir figure 5.1) : **BookNodeWithChoices**, **BookNodeWithRandomChoices**, **BookNodeCombat** et **BookNodeTerminal**.

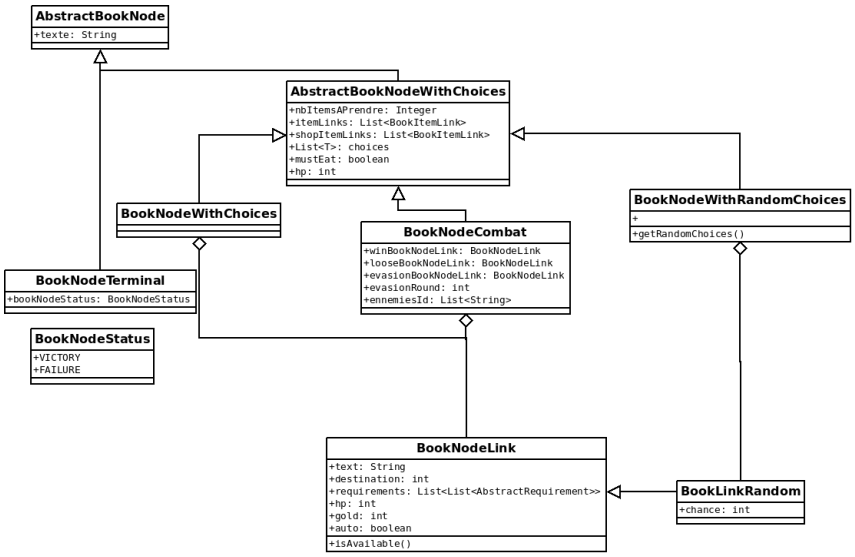

FIGURE 5.1 – BookNode

Tous extends direct ou indirectement de **AbstractBookNode**. Cette dernière contient juste un texte correspondant aux paragraphes de chaque noeuds.

Pour le **BookNodeTerminal**, il prend en compte, en plus du texte de **AbstractBookNode**, une énumération (**BookNodeStatus**) qui ne prend en compte que **FAILURE** ou **VICTORY**.

Pour l'ensemble des autres noeuds, on retrouve une liste de choix et une perte/gain de vie. Pour cette raison, elles extends 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 de choix disponible. Ce dernier est défini par une List<T> car cela peut être un **BookNodeLink** ou un **BookNodeLinkRandom**. Nous avons fait le choix d'utiliser un générique sur 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 3 BookNodeLink afin de représenter les différentes issues du combat : victoire, evasion, defaite. Une liste contenant les ID des ennemis que l'on doit combattre y est également renseignée. De plus, ce noeud peut également posséder 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 dans une liste nommée listNodeLinkDisponible. 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. Dans le cas contraire, on choisit un nombre aléatoire en fonction de la somme totale des probabilités des liens valides (Voir listing 5.1).
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).

```
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 défini 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 classe Book à la page 15. 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. Pour savoir si le player peu aller vers ce lien, un appel de la fonction isSatisfied()5.3, vu un peu plus loin, sera utilisé afin de savoir si le personnage rempli la condition demandée (item et/ou skill et/ou monnaie). Si le player satisfait tous les prérequis demandés, cela retourne true. False dans le cas contraire.

```
1     for(List<AbstractRequirement> groupRequirement : requirements) {
2         boolean satisfied = true;
3         for(AbstractRequirement r : groupRequirement) {
4                 if(!r.isSatisfied(state)) {
5                     satisfied = false;
6                     break;
```

```
                }
            }

        if(satisfied)
            return true;
    }
```

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 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). Tous prennent un ID en compte, permettant de retrouver l'item / skill / monnaie demandé. Seul le RequirementMoney possède une méthode qui diffère, c'est la quantité de money requis.
Pour savoir si un item / skill est présent dans l'inventaire du personnage principal, une for est utilisée afin de regarder les items / skill possédés. Si l'ID de l'item / skill 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 player 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.

```
public boolean isSatisfied(BookState state) {
    for (String i : state.getMainCharacter().getItems()){
        if(i.equals(itemId)) {
            return true;
        }
    }

    return false;
}
```

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.

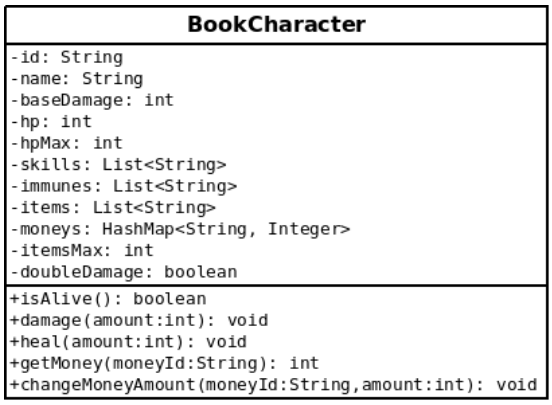

FIGURE 5.2 – UML sur la gestion des personnages

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.

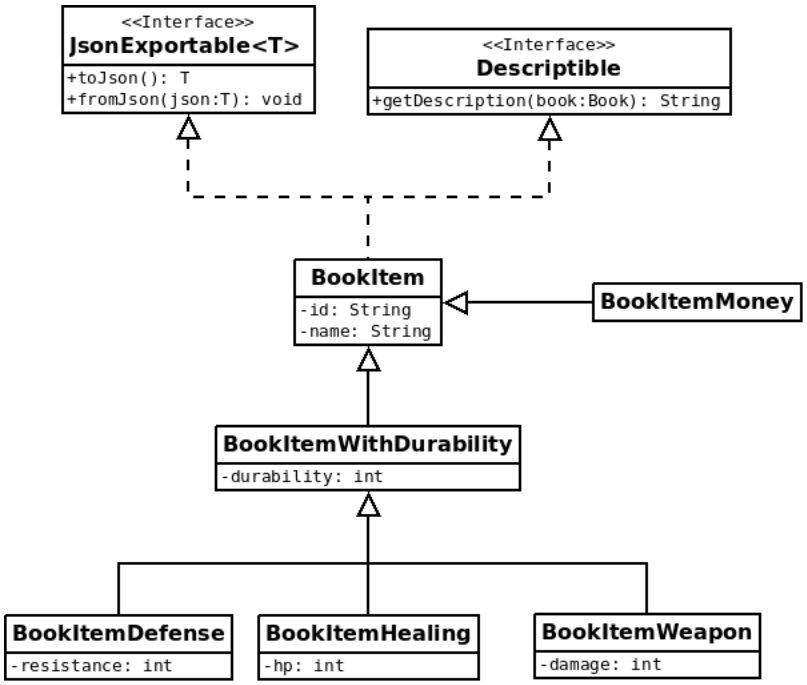

FIGURE 5.3 – UML sur la gestion des items

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)*.

```
@Override
public String getDescription(Book book) {
    StringBuffer buffer = new StringBuffer();

    buffer.append(super.getDescription(book));

    buffer.append("Dégats : ");
    buffer.append(damage);
    buffer.append("\n");

    return buffer.toString();
}

@Override
public ItemJson toJson() {
    ItemJson itemJson = super.toJson();

    itemJson.setDamage(damage);
    itemJson.setItemType(ItemType.WEAPON);

    return itemJson;
}
```

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.

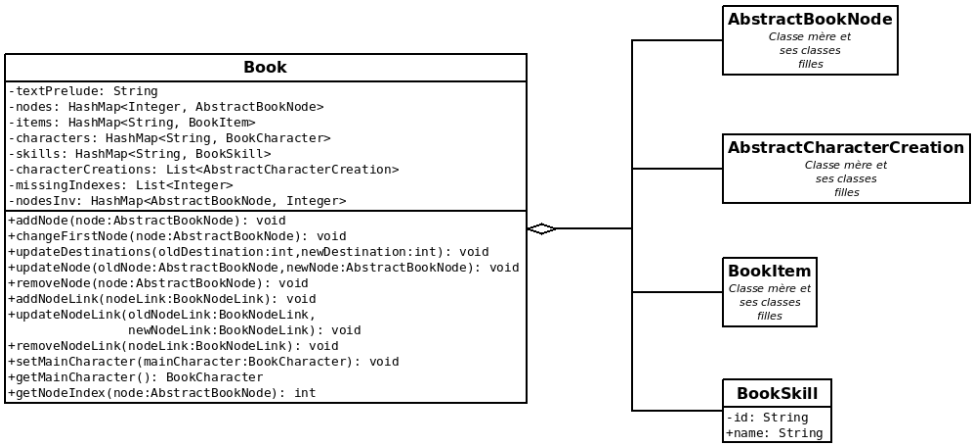

FIGURE 5.4 – UML sur la classe Book

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.

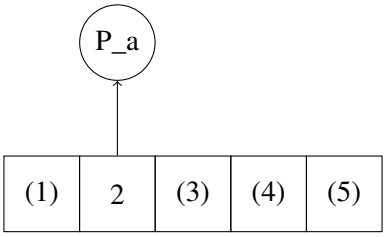

FIGURE 5.5 – Ajout du paragraphe A

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.

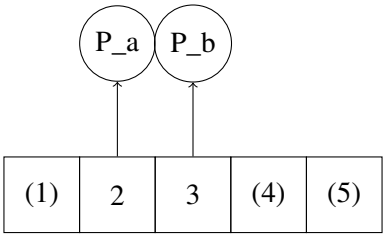

FIGURE 5.6 – Ajout du paragraphe B

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.

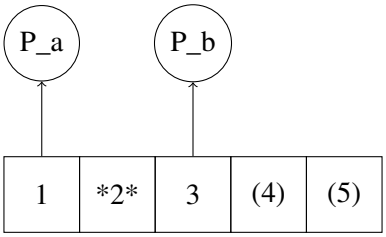

FIGURE 5.7 – Le paragraphe A devient le noeud de départ

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.

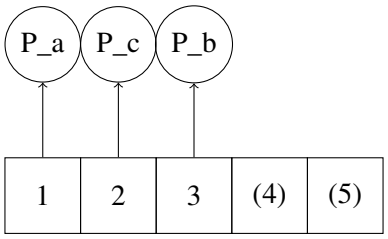

FIGURE 5.8 – Ajout du paragraphe C

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

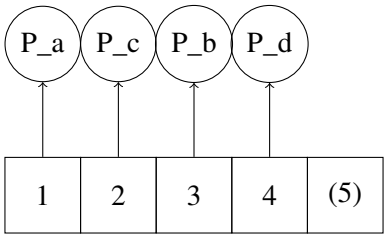

FIGURE 5.9 – Ajout du paragraphe D

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 :

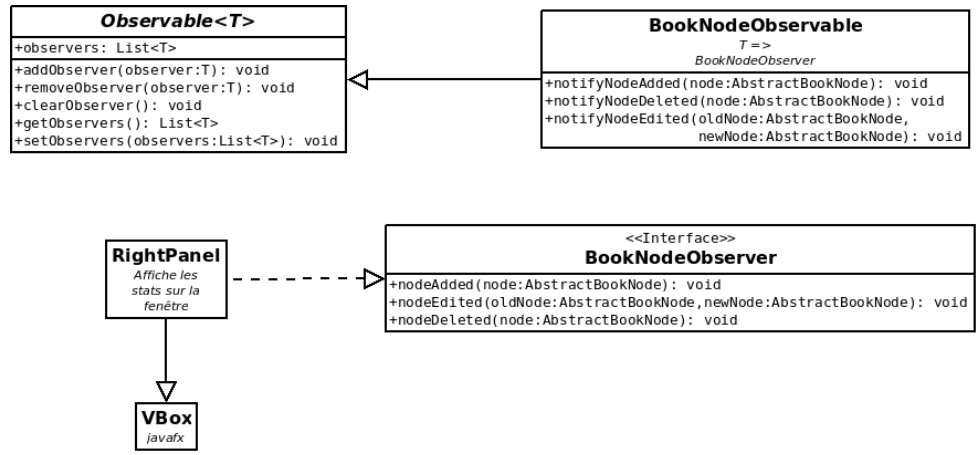

FIGURE 5.10 – UML d'exmple sur le pattern observer

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 skills 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"
            }
```

```
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.

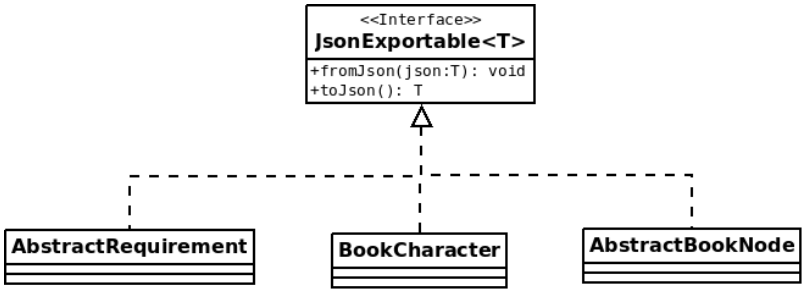

FIGURE 5.11 – L'interface JsonExportable et quelques classes qui l'implémentent

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 :

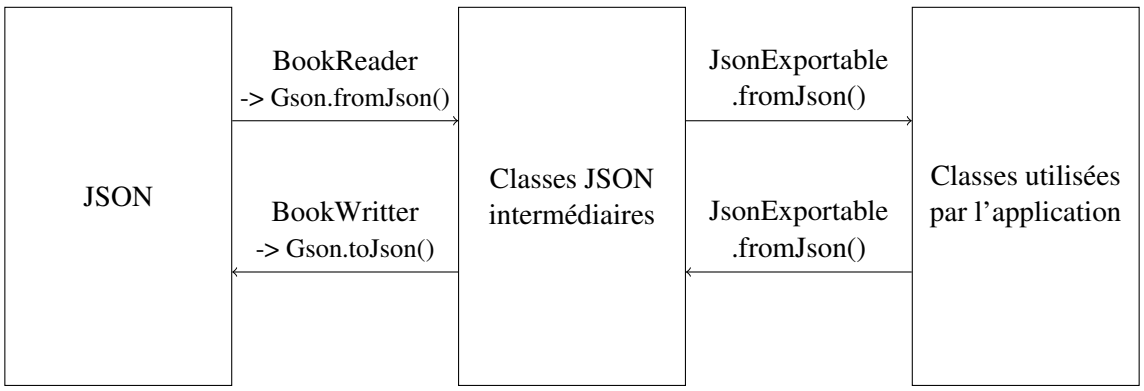

FIGURE 5.12 – Échanges pour la lecture / écriture

C Edition graphique d'un livre

i MainWindow

La MainWindow est notre fenêtre principale. Elle contient un Menu permettant de réaliser plusieurs actions 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 et de personnage 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.

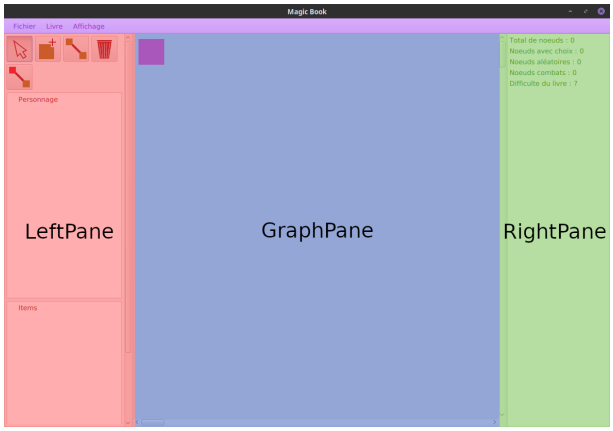

FIGURE 5.13 – MainWindow

Tout d'abord, dans la barre de menu de la fenêtre (menuBar), se trouve plusieurs onglets. Premièrement, dans le menu nommé **Fichier**, l'utilisateur peut alors ouvrir un nouveau livre vide en cliquant sur **Nouveau**. Le nouveau livre s'ouvre alors en mettant à jour tous 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 correcte, le livre est chargé en mettant tous les panels à jour. Il peut aussi enregistrer ou enregistrer-sous le livre en faisant appel au FileChooser et au File.
Deuxièmement, dans le menu nommé **Livre**, l'utilisateur peut **Jouer** ou **Estimer la difficulté**. Ces deux MenuItem utilisent tous les deux la classe Jeu, décrit un peu plus loin Rendre le livre jouable et estimer sa difficulté à la page 30. 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.
Troisièmement, dans le menu nommé **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.

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. Un évènement est associé à chaque boutons permettant de changer le mode. Chaque mode permet d'executer des actions différentes, présenté ci-dessous. Les mêmes évènements de sélection, ainsi que de suppression, sont associés au lien.

```
class NodeFxListener implements RectangleFxObserver {
    @Override
    public void onRectangleFXClicked(RectangleFx rectangleFx, MouseEvent event)
    {
        NodeFx nodeFx = (NodeFx) rectangleFx;
        if(mode == Mode.SELECT){
            //sauvegarde du premier noeud cliqué
            if(event.getClickCount() == 2) {
                //Ouverture d'un NodeDialog
            }
        } else if(mode == Mode.ADD_NODE_LINK) {
            if(selectedNodeFx == null && !(nodeFx.getNode() instanceof
    AbstractBookNodeWithChoices)) {
                return;
            } else if(selectedNodeFx == null && nodeFx.getNode() instanceof
    AbstractBookNodeWithChoices) {
                //sauvegarde du premier noeud cliqué
            } else {
                //Création d'un lien entre les différents type du premier noeud
            }

        } else if(mode == Mode.DELETE) {
            //Supprime le noeud
        } else if(mode == Mode.FIRST_NODE) {
            //Modifie la destination du prélude
        }
        event.consume();
    }
}
```

Listing 5.13 – Modification Mode

Une VBox est ensuite créé afin de mettre deux TreeView contenant un TreeItem pour la liste des items et un autre TreeItem pour celle des personnages. Il est possible d'ajouter des items et des personnages à partir de MenuItem (Ajout, Modification, Suppression) qui sont contenu dans un ContextMenu. Des évenèments sont liés à ces MenuItem afin de modifier le livre permettant d'appeler des méthodes afin de modifier la liste des items ou la liste des personnages.

Prenons l'exemple d'un ajout d'un item. Un clique droit est alors enregistrer en tant qu'évènement. Ce clique permet de faire appel à la méthode handle() afin de créer un **ItemDialog** (boite de dialogue). La boite de dialogue est donc affiché, les valeurs peuvent être rentré. Si la boite n'est pas validé, aucun changement n'est effectué. Mais si l'utilisateur valide la boite, un **BookItem** est créé en fonction du type d'item (arme, un item de soin, de défense ou un simple item). L'item est alors ajouter dans le livre à l'aide d'une méthode addItem(). Cette méthode est présente dans la classe **Book**, permettant d'ajouter l'item et, dès ce faisant, notifier la classe **BookItemObservable** qu'un item a été ajouté. La méthode itemAdded() du **LeftPane** est alors appelé afin d'ajouter au treeView une vue sur l'item créer. Vous trouverez ci-dessous, la méthode si la captation d'un évènement "Ajouter un item" est enregistré. L'ajout d'un personnage se fait exactement dans la même idée.

```
menuItemAdd.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        ItemDialog itemDialog = new ItemDialog(LeftPane.this.book);
        BookItem item = itemDialog.getItem();
        if(item != null) {
            book.addItem(item);
        }
    }
});
```

Listing 5.14 – Ajout d'item

Prenons maintenant l'exemple d'une modification d'un item. Le même principe est effectué, sauf que l'item où il a été enregistré le clique, est envoyé dans la classe **ItemDialog**. L'item est alors modifié en fonction de son instanceof. Puis est modifié si le bouton "Valider" a été appuyé. La méthode updateItem est alors appelé de la classe **Book**. L'observeur fait maintenant appel à itemEdited(). Cette méthode permet de changer la valeur de l'item à l'aide d'un setValue. Vous trouverez ci-dessous, un exemple d'une modification d'un item.

```
menuItemUpdate.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        TreeItem<BookItem> selectedItem = treeViewItem.getSelectionModel().
    getSelectedItem();
        if(selectedItem != null) {
            BookItem oldItem = selectedItem.getValue();

            ItemDialog newItemDialog = new ItemDialog(oldItem, LeftPane.this.
    book);
            BookItem newItem = newItemDialog.getItem();

            if(newItem == null){
                return;
            }

            book.updateItem(oldItem, newItem);
        }
    }
});
```

Listing 5.15 – Modification d'item

Maintenant, prenons l'exemple d'une suppression d'item. Un enregistrement d'évènement est alors effectué sur "Supprimer un item". Une notification est envoyé directement dans le book, supprimant ainsi l'item et appelant la méthode du **LeftPane**, *itemDeleted*. Cette méthode enlève alors du TreeView l'item supprimé à l'aide d'un remove(). Vous trouverez ci-dessous, le code de suppresion lors de l'enregistrement de l'évènement.

```
menuPersoDel.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        TreeItem<BookCharacter> selectedItem = treeViewPerso.getSelectionModel()
    .getSelectedItem();
        book.removeCharacter(selectedItem.getValue());
    }
});
```

Listing 5.16 – Supression d'un item

iii GraphPane

Le prélude est déjà situé en haut à gauche du GraphPane. Ce prélude, comme les noeuds, est représenté par un carré. Il contient un PréludeFx faisant le lien entre le noeud du prélude et le Rectangle le contenant. Ce prélude contient trois Tab. La première Tab contient le texte du prélude. La deuxième contient la création du personnage. Et la troisième la création du personnage principal.
La deuxième partie du TabPane contient un Accordion. Cette accordion est affiché grâce au bouton **Ajouter**. Il est composé d'un CharacterCreationComponent comprenant un Texte, une CheckBox (Texte, Item), et d'un bouton **Supprimer cette partie**.
La checkBox n'affiche que **Texte** si aucun item n'est disponible. Mais si un item est créé, l'utilisateur peu alors ajouter un ItemListComponent en sélectionnant **Item** dans la CheckBox. Cette classe est affiché sur

tout les noeuds, permettant d'ajouter des items ainsi que la quantité disponible de chaque item, qui ici, sera disponible en début de partie. L'item ajouté peut être supprimé grâce à un clic droit de la souris.
La troisième partie du TabPane permet quant a elle, de créer le personnage principal grâce au CharacterComponent. Cette classe permet de créer un personnage. Mais si la valeur boolean de mainCharacter est true, les TextField pour le nombre d'item maximum ainsi que la quantité d'argent est ajouté. Après validation, le textPrelude (texte du prélude), mainCharacter (personnage principal) et le characterCreations (item et texte à afficher après le prélude) est créer.

Maintenant que vous avez une vision du prélude, une présentation de son code s'impose. Tout d'abord, le prélude contient donc la classe **RectangleFx**. Cette classe envoie donc une notification si un évènement *Mousse* est enregistré. Cette évènement est recu par le **PréludeFx**, qui lui, regarde si le mode **SELECT** est activé. Si ce mode est activé et deux cliques sont enregistré sur le prélude, ce dernier ouvre sa boite de dialog avec, ou non, des informations à l'intérieur (si c'est une modification).
Dès que le prélude est valider, le book modifie son prélude, le personnage principal ainsi que le characterCreations (ajout des items, texte). Ci dessous, vous découvrirez le code de modification du prélude.

```
1  private void createNodePrelude() {
2      PreludeFx preludeFx = new PreludeFx(zoom);
3      preludeFx.setRealX(10);
4      preludeFx.setRealY(10);
5
6      preludeFx.addNodeFxObserver((RectangleFx rectangleFx, MouseEvent event) -> {
7          if(mode == Mode.SELECT) {
8              if(event.getClickCount() == 2) {
9                  PreludeDialog dialog = new PreludeDialog(book);
10
11                 if(dialog.getTextePrelude() != null) {
12                     book.setTextPrelude(dialog.getTextePrelude());
13                     book.setMainCharacter(dialog.getMainCharacter());
14                     book.setCharacterCreations(dialog.getCharacterCreations());
15                 }
16             }
17         }
18     });
19
20     preludeFxFirstNodeLine.startXProperty().bind(preludeFx.xProperty().add(
       preludeFx.widthProperty().divide(2)));
21     preludeFxFirstNodeLine.startYProperty().bind(preludeFx.yProperty().add(
       preludeFx.heightProperty().divide(2)));
22
23     rootPane.getChildren().add(preludeFx);
24     this.setPreludeFx(preludeFx);
25 }
```

Listing 5.17 – Ajout / modification du prélude

Concernant l'ajout d'un noeud, on clique sur le mode **ADD_NODE**,puis on clique sur le GraphPane. Une méthode *createNodeFxDialog* est alors appelé.

```
1      public NodeFx createNodeFxDialog(MouseEvent event){
2          NodeDialog nodeDialog = new NodeDialog(book);
3          AbstractBookNode node = nodeDialog.getNode();
4
5          if(node != null) {
6              lastXClick = event.getX();
7              lastYClick = event.getY();
8
9              book.addNode(node);
10         }
11
12         return null;
```

```
13      }
```

Listing 5.18 – ADD NODE()

Une fenêtre de dialog est alors ouverte afin de renseigner le paragraphe à afficher, ainsi qu'une CheckBox permettant de sélectionner le type de noeud. L'affichage change en fonction du type de noeud sélectionner dans cette CheckBox, qui elle, est prédéfini sur **Basic**.

Si c'est un noeud basic ou aléatoire, les renseignements suivant sont demandé : le texte à afficher, le nombre de points de vie à gagner/perdre, les items disponible, le nombre d'item disponible. Rien ne change sur la structure de la boite de dialog entre ces deux types. Mais au moment de la validation, le type de BookNode créer se fera en fonction de la valeur de la choiceBox : BoockNodeWithChoices, BookNodeWithRandomChoices. Des try/catch sont utilisé afin de changer certaine valeurs de saisie en Integer. Cela permet d'être sûr que ce sont des chiffres. Si la conversion ne peut pas se faire, une boite de dialog affichant l'erreur apparait, notifiant ainsi où l'erreur se situe.

Si c'est un noeud terminal, une ChoiceBox est juste rajouté afin de savoir si c'est un noeud gagnant ou perdant. Lors de la validation, un BookNodeTerminal est créer avec comme valeur un BookNodeStatus défini en fonction de la ChoiceBox.

Dès qu'un noeud est validé, la méthode *createNodeFxDialog* envoie une notification au **Book** permettant que le noeud créer soit ajouté dans la liste. Une fois cela effectué, un observeur est notifié afin d'effectuer un appel à la méthode *nodeAdded* dans le **GraphPane**. Cette méthode permet de créer le **NodeFx** qui extend de **Rectangle Fx**.

```
1  public NodeFx createNode(AbstractBookNode node, double x, double y) {
2      NodeFx nodeFx = new NodeFx(node, zoom);
3      nodeFx.setRealX(x/zoom.get());
4      nodeFx.setRealY(y/zoom.get());
5
6      nodeFx.addNodeFxObserver(new NodeFxListener());
7
8      listeNoeud.add(nodeFx);
9      rootPane.getChildren().add(nodeFx);
10
11     lastXClick = 0;
12     lastYClick = 0;
13
14     return nodeFx;
15 }
```

Listing 5.19 – Création du NodeFxDialog

Il contient alors le noeud créer et affiche un Rectangle là où la souris a été cliqué dans le graphPane, à la création du noeud. La couleur du carré change en fonction du type de noeud. Un RectangleFX est donc mis en place permettant d'envoyé une notification si jamais la souris passe dessus (opacité du Rectangle qui change), réalise un clique maintenu (déplacement du rectangle en fonction de la souris), réalise un clique simple. Pour ce dernier, cela envoie une notification à l'observable permettant de prévenir l'observeur qu'on a cliquer dessus. Cela sert si un double clique est produit (permet de procédé à une modification si le mode activé est **SELECT**, a une suppression si c'est le mode **DELETE** qui est activé ou encore a une création de lien entre le prélude et le noeud sélectionné si le mode **FIRST_NODE** est activé) ou si un deux cliques espacé sont réalisé (permet de procédé à la création d'un lien si le mode **ADD_NODE_LINK** est sélectionné).

Une fois un ou plusieurs noeud créer, un lien peut être donc être effectué. Le lien peut être fait sur le même noeud, si une boucle est voulu. Mais il peut aussi être réalisé entre deux noeuds. Pour ce faire, le mode **ADD_NODE_LINK** doit être activé. Le premier clique permet de sauvegarder le noeud de départ. Le deuxième permet d'avoir le noeud de destination. Une fois les deux cliques détecté, une fenêtre de dialog apparait alors.

Il y a trois affichage différent. L'affiche ce fait en fonction du premier noeud. Tout d'abord, l'affichage commun entre ces noeuds est un simple texte, deux TextField ainsi qu'un CheckBox. Les TextField permettent d'ajouter un gain ou une perte de vie/d'argent. La CheckBox, quant a elle, permet d'aller vers ce choix obligatoirement ou non.
Si le premier noeud est un noeud de combat, une ChoiceBox apparait alors permettant de choisir si c'est un lien gagnant, perdant ou un lien d'évasion. Cette liste de choix est créer si le permier noeud à encore des choix de libre. S'il n'a plus de choix libre, une boite d'alerte s'affiche en prévenant qu'il n'y a plus aucun lien disponible.

Si le permier noeud est de type aléatoire, le joueur doit alors renseigner la chance sur chaque lien.

Une fois la boite de dialogue validé, une méthode est appelée dans la Classe **Book**, afin d'ajouter le lien ainsi que de notifier le **BookNodeLinkObservable**. Cet observeur va alors appeler la méthode *nodeLinkAdded* permettant de créer une ligne entre le NodeFx de départ et le NodeFx de destination. Pour différencier les deux, un cercle est créé et est affiché au noeud de destination. Tout cela est géré par NodeLinkFx qui a aussi un observeur permettant d'envoyer une notification si jamais la ligne ou le cercle enregistre un événement du style "pressed". Cela permet de savoir si le lien a été double cliqué pour réaliser une modification.
Ce NodeLinkFx est créer dans la méthode *createNodeLink*, défini ci-dessous.

```
1
2  public NodeLinkFx createNodeLink(BookNodeLink bookNodeLink, NodeFx firstNodeFx,
       NodeFx secondNodeFx) {
3      NodeLinkFx nodeLinkFx = new NodeLinkFx(bookNodeLink, firstNodeFx,
       secondNodeFx, zoom);
4      nodeLinkFx.addNodeLinkFxObserver(new NodeLinkFxListener());
5
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 }
```

Listing 5.20 – Création du lien NodeLink

iv RightPane

Ce panel contient tout ce qui représente les statistiques. Dès qu'un noeud est ajouté ou supprimé, une méthode de la class *RightPane* est utilisé afin de mettre à jour le nombre de noeuds existant. Pour la difficulté du livre, elle est mise à jours dès que **Estimé la difficulté du livre** est sélectionné.
Si un nouveau livre est chargé, tout ces statistiques ce remmettent à zéro.

D Rendre le livre jouable et estimer sa difficulté

i Jeu

Une classe à été créer se nommant **Jeu**, permettant de gérer les méthodes de jeu communes entre le *Player* et les *Fourmis*.
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 player est appelé.
Une fois dans la méthode choisis, le livre est alors copié afin de ne pas le modifier par erreur pendant le jeu. Un BookState, qui correspond à la sauvegarde de la partie, est alors créé. Si le prélude contient un personnage principal, alors celui ci est enregistré dans la sauvegarde de la partie (le BookState). Dans le cas contraire, un autre personnage principal est créer afin de pouvoir jouer au jeu.

```
1  BookState newState = new BookState()
2  if(this.book.getMainCharacter() == null){
3      //Création d'un BookCharacter
4  } else {
5      BookCharacter bookCharacterMain = this.book.getMainCharacter();
6  }
7  newState.setBook(this.book);
8  for(AbstractCharacterCreation characterCreation : this.book.
       getCharacterCreations())
9      player.execPlayerCreation(book, characterCreation, newState);
10
11 return newState;
```

Enfin, si des compétences et/ou des items sont disponible au début de la partie, la méthode de création de joueur est appelé en focntion du player actuel. Comme cela, le player choisit parmi une liste de skill et d'items disponible en début de partie afin de les avoir pour commencer le jeu.
Une fois le BookState créer, le personnage principal initialisé et la copie du livre enregistré, le premier noeud est donc chargé. Une méthode est appelé en fonction de son type de noeud. La méthode correspondante au type de noeud s'exécute et renvoie le noeud de destination, en fonction du choix du player et/ou de la mort du player. Durant l'exécution des différentes méthodes et en fonction du player, d'autre méthode externe sont appelé 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;
```

Pour chaque type de noeud, sauf pour un noeud terminal, une méthode commune est appelé afin de savoir si le noeud pris en charge fait gagné/perdre de la vie puis regarde si le player est toujours en vie. Si ce dernier n'est plus en vie, un noeud terminal est alors renvoyé en noeud de destination. S'il est encore en vie et que le noeud propose des items, ils sont proposés au player en appelant la méthode correspondante entre Fourmis ou Player. A chaque detination choisi, une autre méthode est appelé 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 un noeud est de type basic, il est alors pris en charge dans la méthode *execNodeWithChoices*. Cette dernière renvoi un noeud terminal si aucun choix n'est valide, ce qu'il veut dire, si le player n'a aucun choix ou s'il ne possède pas les items/skill pour aller vers ce choix.
Si le player est encore en vie et si il peut au moins choisir une destination, un choix est demandé parmis toutes les destinations faisant un appel à la méthode en fonction du player. Si le player a les prérequis pour aller vers cette destination, alors le noeud choisi est renvoyé en noeud de destination. Sinon, le player doit faire un autre choix.

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. Sinon, une liste d'ennemis est créer afin de ne pas modifié la vie des ennemis. Car ces derniers ne sont pas lié au noeud, mais c'est l'IDde l'ennemis qui est lié au noeud permettant de les appelés plusieurs fois dans plusieurs ou dans le même noeud.
Le combat commence alors. Le choix est défini par la méthode du player correspondant. Trois choix sont possible :

Attaque : un autre choix est demandé permettant de sélectionner l'ennemi à attaquer parmi la liste des ennemis encore en vie. Une fois l'ennemi sélectionné, une méthode attaque est appelé apellant elle même une autre méthode commune entre l'attaque du player 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. Une fois l'attaque effectué, si l'ennemi attaqué est mort, il est supprimer de la liste des ennemis.

Inventaire : si ce choix est fait par le player, 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.

Evasion : si le tour avant évasion est inférieur ou égal à 0 et si un noeud de d'évasion existe, le player peut alors s'enfuir. Sinon cela lui passe son tour.

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.

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.

La fin de combat est déterminé si la liste d'ennemis est vide ou si le player n'est plus en vie. Le noeud de destination est alors défini en fonction du résultat en fin de combat.

Si le noeud est de type aléatoire, la méthode commune est appelé afin de savoir si le player est encore en vie. Puis une autre méthode est appelé 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 bolean sur l'état de la fin de partie.

ii Interface Player / Foumis

Une interface **InterfacePlayerFoumis** à été créer 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 player.

execPlayerCreation permet de choisir les skill et les items disponible 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.

combatChoice, prend en paramètre le noeud de Combat et le nombre de tour avant l'évasion ainsi que le BookState, permet de faire un choix lors du tour du player 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.

chooseEnnemi permet de choisir l'ennemi à attaquer parmi la liste de tout les ennemis encore en vie.

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.

makeAChoice permet de faire un choix en fonction des différentes destinations proposé par le noeud.

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 actualisé.

iii Player

La classe **Player** permet de jouer au jeu en tant que joueur. Elle permet de faire des choix grâce aux Scanner. Des messages sont aussi affiché afin de guider le joueur dans ses choix.
Notament 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 un skill.
Pour la méthode commune *prendItems*, cette dernière fait appel à d'autre méthode dans la classe Player. Comme itemAdd() permettant de choisir l'item à ajouter dans l'inventaire. Ou encore itemPlein() qui demande au joueur s'il veut supprimer un item. Si le joueur répond oui, la méthode itemSupp() est appelé afin de choisir l'item à supprimer et à mettre à jour l'inventaire.
Pour la méthode *execPlayerCreation* au moment de l'ajout des skill, le joueur doit confimer ou non s'il veut un skill. Si oui, la méthode skillAdd est appelé jusqu'à ce que le maximum d'item à été pris ou qu'ilne reste plus de skill à prendre.

iv Fourmis

La classe **Fourmis** permet de jouer en tant que joueur fictif. Elle effectue des choix random en fonction des différentes méthodes de l'interface. Comme par exemple, pour prendre des items ou des skill, la fourmi en prend autant que possible et en supprime obligatoirement en aléatoire si il n'a plus de place dans l'inventaire. Nous avons décider de réalisé cette méthode comme cela afin de pouvoir aller dans le maximum de noeud s'ilsont des prérequis ou alors avoir le maximum d'items pour les combats.
Pour la méthode *chooseEnnemi*, la fourmi envoyé prend obligatoirement le premier ennemi permettant de tué le maximum d'ennemis en attaquant toujours le même ennemis.
Et enfin, la méthode *combatChoice* permet de choisir entre ATTAQUE, EVASION, INVENTAIRE. Nous avons choisir de faire un random sur les trois choix et non pas sur deux choix même si le tour d'évasion n'est pas disponible afin de passer le tour, comme le joueur, afn d'avoir la même chance lors des combats.

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