



UNIVERSITÉ
CAEN
NORMANDIE

Magic Book

Rapport de projet

DEROUIN Auréline 21806986

MARTIN Justine 21909920

THOMAS Maxime 21810751

Table des matières

1	Présentation du projet	1
A	Présentation de l'application	1
2	Organisation du projet	2
A	Choix des technologies	2
i	Git	2
ii	Gradle	2
iii	JavaFx	3
B	Gestion du projet	3
i	GitHub et Forge	3
ii	Trello	3
iii	Discord	4
3	Travail de groupe	5
A	Répartition des tâches	5
B	Idées d'améliorations	7
C	Bugs et problèmes connus	8
4	Architecture du projet	9
A	Arborescence du projet	9
B	Présentation des packages	9
5	Aspects techniques	11
A	Représentation d'un livre	11
i	Représentation des noeuds	11
ii	Représentation des liens	12
iii	Prérequis pour un choix	13
iv	Représentation des personnages, items	13
v	La classe Book	15
vi	Le pattern observer et la classe Book	18
B	Lecture et écriture d'un livre	19
i	La structure du JSON	19
ii	La lecture et l'écriture	22
C	Edition graphique d'un livre	23
i	MainWindows	24
ii	LeftPane	24
iii	GraphPane	25
iv	RightPane	27
D	Rendre le livre jouable et estimer sa difficulté	27
i	Jeu	27
ii	Interface Player / Fournis	29
iii	Player	29
iv	Fourmis	30

6 Conclusion	31
7 Ressources utiles et sources utilisés	32

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 pélude, 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 créé. 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 fait le choix 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 décentralisé, une interruption de service d'un hébergeur n'empêche pas de continuer le travail et il est facile d'héberger son code sur une autre 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 donc la branche à partir de laquelle 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 donc 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 dépôts, 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'exécution 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 bibliothèques 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 vu 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 dépôt. 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 modifications 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 événement 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'entièreté des commits est conservée. Des pushes sur la Forge sont donc réalisés toutes les semaines afin d'actualiser le dépôt. Bien entendu un push final a été réalisé sur la Forge pour rendre le projet.

ii Trello

Concernant la répartition et le "listing" du travail à effectuer, nous avons fait le choix d'utiliser [Trello](#), 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 ce 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 vérifier" 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 du 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* la déplacer dans la seconde colonne.

iii Discord

Afin de faciliter la communication au sein du groupe, nous avons utilisé 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" nous permettait d'obtenir toutes les informations sur les push, pull-request, résultats des tests concernant le dépôt sur GitHub. "important-magic-book" permet de transmettre des messages importants sur ce qui a été fait, sur des changements importants concernant le projet, etc. Enfin, "dev-magic-book" était une discussion beaucoup plus générale dans laquelle on pouvait demander de l'aide, aider des membres en difficulté, ou même de discuter de certains choix à faire.

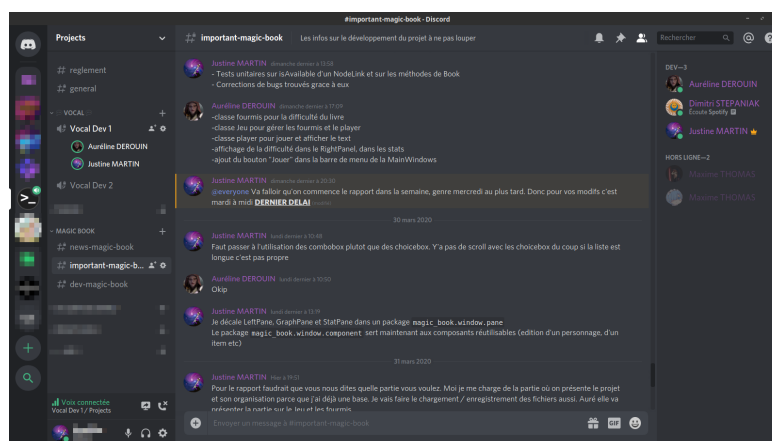


FIGURE 2.2 – Notre serveur Discord

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 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	
Ajout des classes pour le pattern observer (uniquement celles qui concernent le 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	
Conception d'un Parser pour le texte des liens et des paragraphes			X	
Version primitive de l'estimation de la difficulté d'un livre				X
Fenêtre				
Fenêtre principale		X	X	
Permettre la conception d'un nouveau livre, l'ouverture d'un ancien livre sauvegarder, la sauvegarde et "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	
Cache le panel de gauche si l'on décoche une case dans le menu				X

Tâches effectués	Auréline	Dimitri	Justine	Maxime
Séparation des différentes parties de la fenêtre en plusieurs classes (LeftPane, GraphPane, RightPane)	X			
Composants réutilisables pour créer des personnages, une phase de la "Création d'un personnage", sélectionner une liste d'items			X	
Boîtes de dialogues				
Classe mère pour les boîtes de dialogue	X			
Boîte de dialogue pour les noeuds	X			
Boîte de dialogue pour les liens entre les noeuds	X			
Boîte de dialogue pour les items	X			
Boîte de dialogue pour les personnages			X	
Boîte de dialogue pour le prélude			X	
Boîte de dialogue pour la "Création du personnage"			X	
Boîte 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 (NodeLinkFx)			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			
Classe mère commune pour représenter un prélude et un noeud (RectangleFx)			X	
Permettre le déplacement des noeuds	X			
Détecter un clique sur un noeud ou 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	~
Restructuration du livre d'exemple (fotw.json)			X	
Création de tests unitaires	X		X	
Javadoc	X			

Tâches effectués	Auréline	Dimitri	Justine	Maxime
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 personnes. 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 ajoutés, et 10 460 lignes supprimés sont données à la personne qui les ont commit, Justine dans notre cas.

Après avoir lancé un script ([git-stats](#)) pour connaître le nombre de lignes par commit de chacun, voici ce que donnerait le graphique si l'on enlevait toutes ces lignes à Justine :

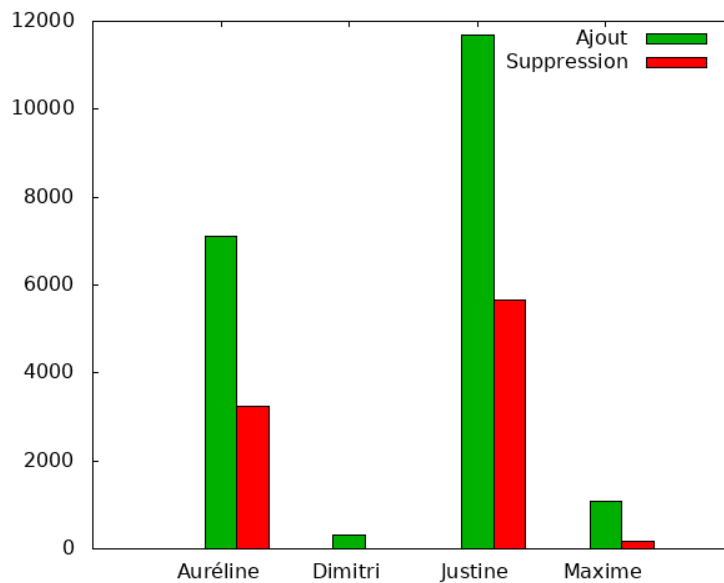


FIGURE 3.1 – Nombre de lignes d'ajoutés et supprimés par personnes

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 transfère 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 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 boîtes 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)
- "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 déverrouiller certains passages par exemple.

C Bugs et problèmes connus

Certains et 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 éléments du livre (noeuds, choix, ...)
- Diverses bugs visuels concernant la boîte 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 destiné 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, musiques, ...).

test : Code des tests unitaires.

java : Code source.

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

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 documentation 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 intermédiaire 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** : Classe 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 type de noeud représenté par des classe (Voir ??) : **BookNodeWithChoices**, **BookNodeWithRandomChoices**, **BookNodeCombat** et **BookNodeTerminal**. Tous extends direct ou indirectement de **AbstractBookNode**. Cette dernière contient juste un texte pour le constructeur car toutes les méthodes contiennent au moins un texte.

Pour l'ensemble des méthode représentant les différents type de noeuds, sauf le **BookNodeTerminal**, chacune d'entre elles contiennent des choix et une perte/gain de vie. Pour cela, ces dernières extends de la classe **AbstractBookNodeWithChoice** permettant ainsi de définir des variable tel que le nombre d'item à prendre, la liste des items disponible sur ce noeuds, la liste des items à acheter, un boolean qui oblige l'utilisateur à manger pour continuer, le nombre de vie perdu/gagner sur ce noeud, la liste de choix disponible. Ce dernier est défini par une `List<T>` car cela peut être un **BookNodeLink** ou un **BookNodeLinkRandom**. Ce qui veut dire que ça peut être soit un lien normal, soit un lien venant d'un noeud aléatoire.

Pour le **BookNodeWithRandomChoices**, une méthode, en plus de son constructeur à été ajouté. En effet, elle permet de donner un choix random en fonction du nombre aléatoire généré ainsi que le nombre de chance de chaque lien. Cela ajoute d'abord tout les liens disponibles, c'est à dire, tout les liens où le joueur peut avoir accès en fonction des prérequis demandés, dans une liste nommé `listNodeLinkDisponible`. Si aucun lien n'est disponible, cela retourne null. Dans le cas contraire, cela choisi un nombre aléatoire en fonction de la somme total de chance de tout les liens valide.

Puis, en fonction de ce nombre, la chance de chaque lien valide est enlever du nombre random jusqu'à ce que ce nombre soit égal ou inférieur à zéro. Le choix de la destination est alors retourné.

```
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) ;
15
```

Pour le **BookNodeTerminal**, il prend en compte, en plus du texte de **AbstractBookNode**, un `BookNodeStatus` défini soit sur **FAILURE** soit sur **VICTORY**.

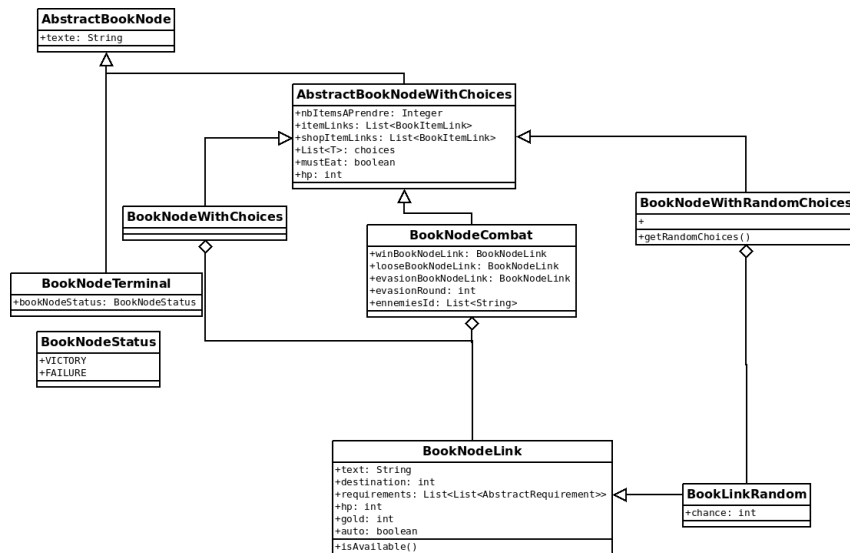


FIGURE 5.1 – BookNode

ii Représentation des liens

Les noeuds sont lié par des liens. C'est liens sont soit défini par la classe **BookNodeLink** ou **BookNodeLinkRandom**. Pour évité 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 pré-requis. Pour la destination, nous avons d'abord mis un **AbstractBookNode**, mais des instanceof était requis pour chaque appel de la destination. Nous avons alors choisis de changer et de mettre le numéro du choix correspondant.

Pour le **BookNodeLinkRandom**, la classe à besoin d'une variable chance, afin de définir la chance d'aller vers ce noeud. Cette chance est ensuite totalisé sur tout les noeuds disponible, comme vu précédementi.

La classe **BookNodeLink** est composé d'une méthode, nommé **isAvailable()** permettant de savoir si le personnage principal à les items requis (défini par une liste de **List<AbstractRequirement>**) pour aller vers ce lien. S'il n'y a pas d'item requis, le player peut alors aller vers ce lien. Sinon, un appel de la fonction **isSatisfied()**5.2, vu un peu plus loin, sera utilisé afin de savoir si le personnage à chaque item/s-kill/money demandé. Si le player satisfait tout les prérequis demandé, 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;
7              }
8          }
9
10         if(satisfied)
11             return true;
12     }
13

```

Listing 5.1 – exemple de isAvailable()

iii Prérequis pour un choix

Les prérequis des liens sont définis par une classe `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 utiliser la méthode `isSatisfied()`, prenant en paramètre un `BookState`. Tous prennent un id en compte, permettant de retrouver l'item/skill/money demandé. Car chaque type possède un id différent afin de pouvoir retrouver l'objet/la compétence correspondant. La seule chose qui diffère c'est la quantité de money requis.

Pour savoir si un item/skill est présent dans l'inventaire du personnage principal, un `for` est utilisé afin de regarder les item/skill possédés. Si l'ID de l'item/skill n'est pas possédé, le personnage principal ne peut donc satisfaire tout les prérequis. Pour l'argent, une simple ligne permet de savoir, en fonction de l'ID de la money, savoir si le player en possède suffisamment. Actuellement, une seule money est disponible, se nommant "gold". En effet, nous n'avons pas eu le temps de créer une boîte de dialog ainsi qu'un espace dédié pour la création de différentes money.

```

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      }
10

```

Listing 5.2 – 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 définis 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.

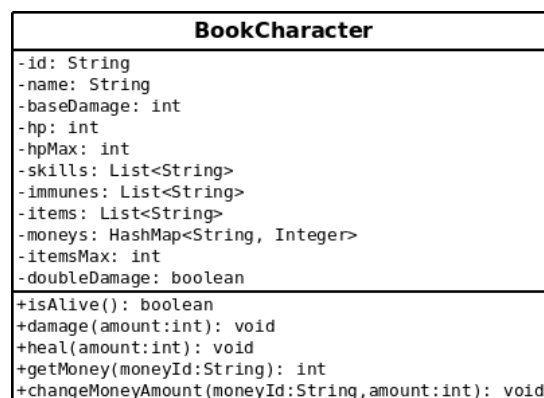


FIGURE 5.2 – UML sur la gestion des personnages

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 nous permet aux

items et compétences de n'être référencés qu'à un seul et même endroit, c'est à dire, dans la classe Book. Pour la monnaie, le choix a été fait 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`.

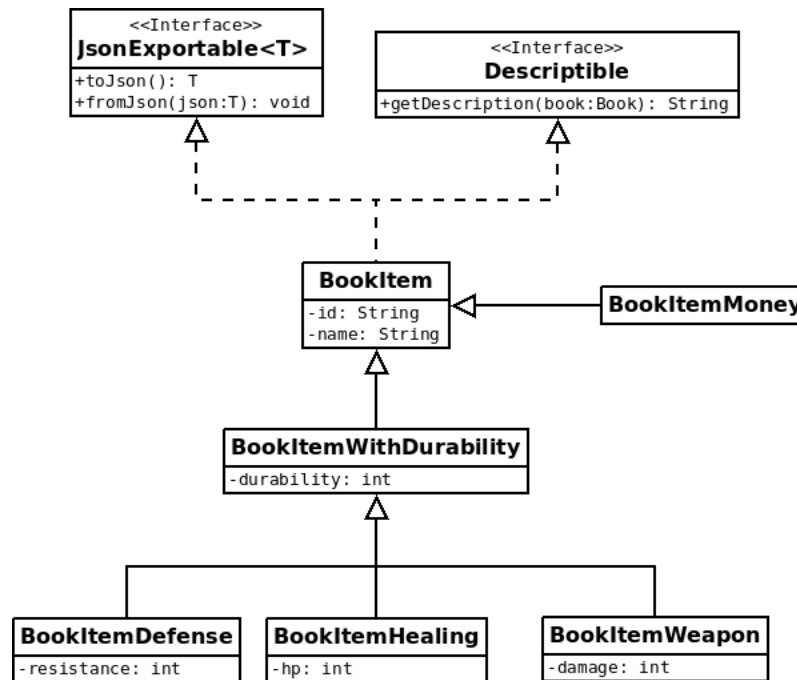


FIGURE 5.3 – UML sur la gestion des items

Avant de détailler notre choix, nous allons expliquer brièvement les deux interfaces que l'on observe. L'une se nomme `Describable` 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 expliquée plus en détails dans [La lecture et l'écriture](#) à la page 22. 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'attribut spécifique à notre classe fille. Par exemple, il serait étrange d'avoir un attribut pour savoir combien de dégât un item de type `money` 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 notera l'appel à la méthode définie 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 5.3 – 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 tous les différents éléments qu'on a pu évoquer avant. C'est en effet ce que l'on peut observer sur la figure suivante.

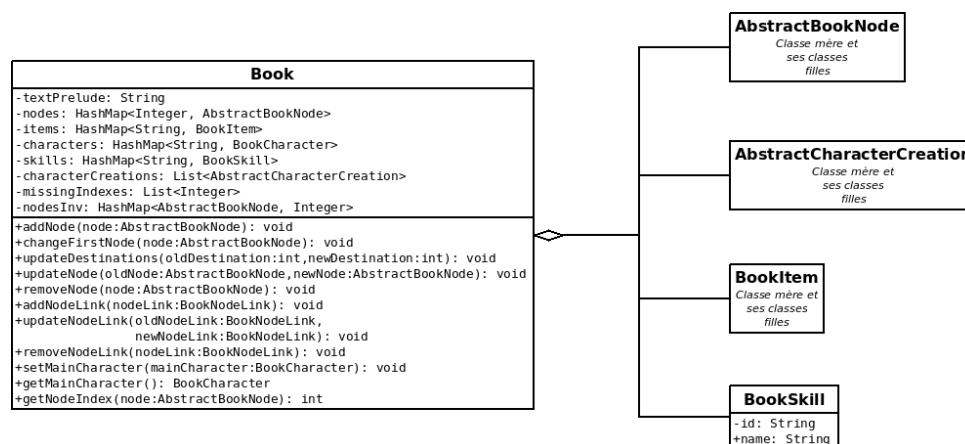
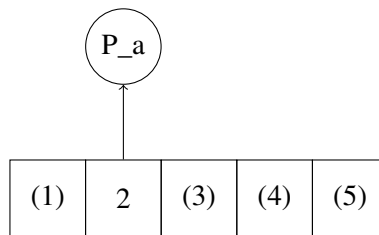


FIGURE 5.4 – UML sur la classe Book

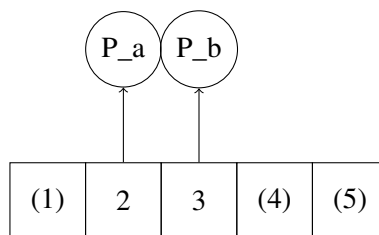
Par soucis de place, les getters et setters n'ont pas été renseignés. Il en va de même pour le détail des différentes classes (**BookCharacter**, **BookItem**, ...). Enfin, les observateurs sont volontairement omis car ils seront détaillés un peu plus tard (dans [Le pattern observer et la classe Book](#) à la page 18).

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

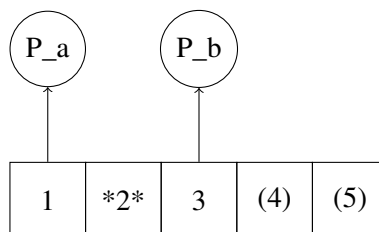
Cette `HashMap` possède un `int` comme clé, clé qui correspond au numéro du paragraphe. Lorsque l'on ajoute un noeud au livre, on doit d'abord lui trouver un numéro. Nous avons décidé de plusieurs règles. Les paragraphes commencent à partir du numéro 1. Le numéro 1 représente **toujours** le premier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

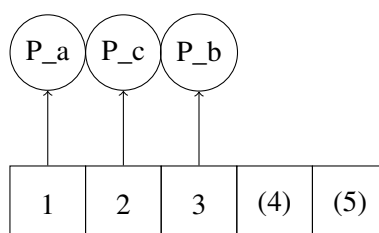
Les autres numéros sont représentés mais mis entre parenthèse car ils n'existent pas dans la Map. Ils sont uniquement là pour nous aider à bien visualiser ce dont on parle. Si nous décidons maintenant d'ajouter un second paragraphe alors celui-ci sera à la position 3.

**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 compléter 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.

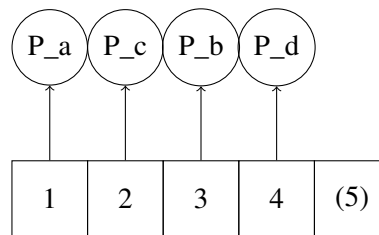


FIGURE 5.9 – Ajout du paragraphe D

Voici quelques précisions supplémentaires :

- Le principe d'indice manquant fonctionne également pour la suppression d'un noeud
- Afin de gagner en performance dans la recherche d'un numéro d'un noeud (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 miroir pour celle des noeuds, on lui passe un noeud en clé et on obtient son indice. Il est donc extrêmement important que les deux maps soient parfaitement identiques pour éviter tous bugs.
- Pour déterminer l'indice d'un noeud, 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 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 noeud le placerait à la position $2 + \text{tailleDuTableau}$ soit $2 + 1$ c'est à dire 3. On a un décalage de un. 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 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 not (node in nodes) then
2   |   addNode(node)
3
4 updateDestinations(1, -1)
5
6 indexOfNode : int
7 indexOfNode ← nodesInv[node]
8
9 oldNode : AbstractBookNode
10 oldNode ← nodes[1]
11
12 updateDestinations(indexOfNode, 1)
13
14 nodes[1] ← node
15 nodesInv[node] ← 1
16
17 if oldNode != null then
18   |   nodes[indexOfNode] ← oldNode
19   |   nodesInv[oldNode] ← indexOfNode
20   |   updateDestinations(-1, indexOfNode)
21 else
22   |   missingIndexes.add(indexOfNode)
23   |   nodes.remove(indexOfNode)
```

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

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 le faire de la manière suivante :

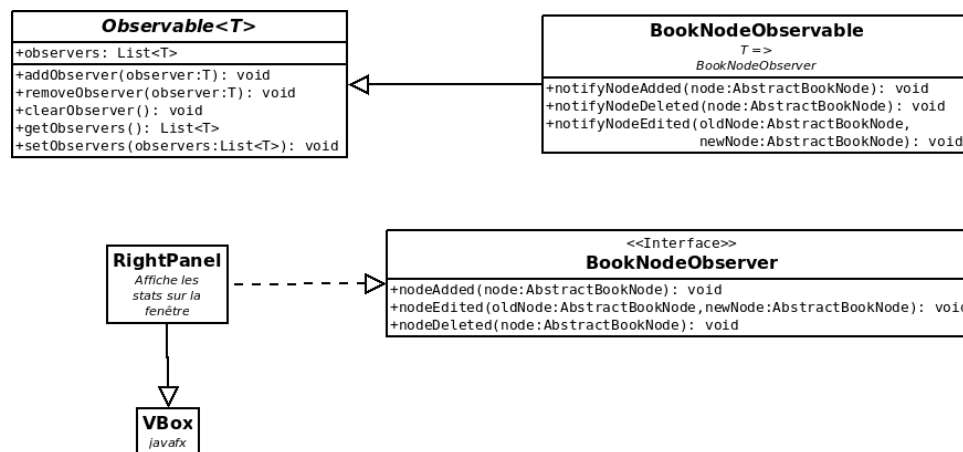


FIGURE 5.10 – UML sur la classe Book

Tous les observers sont dispo sauf pour le changement de 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 fourni 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 ressemblent :

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   },
9   "sections": {
10    "1" : {
11      "text": "Vous être en train d'étudier notre projet",
12      "choices": [
13        {
14          "text": "Mettre une bonne note",
15          "section": 3
16        },
17        {
18          "text": "Mettre une mauvaise note",
19          "section": 2
20        }
21      ]
22    },
  
```

```

23     "2": {
24         "text": "Les étudiants du projet sont tristes",
25         "end_type": "FAILURE"
26     },
27     "3": {
28         "text": "Les étudiants sont satisfait de leur travail",
29         "end_type": "VICTORY"
30     }
31 }
32 }

```

Listing 5.4 – 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 fréquemment 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 connaître la force ajoutée dans un combat, la quantité de soins à rendre par noeuds, par exemple.

```

1 {
2     "id": "sixth_sense",
3     "name": "Sixième sens"
4 }

```

Listing 5.5 – 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 dégâts ("damage") par exemple.

```

1 {
2     "id": "backpack",
3     "name": "Backpack",
4     "item_type": "KEY_ITEM"
5 },
6 {
7     "id": "healing_potion_4",
8     "name": "Potion de soins (4HP)",
9     "hp": 4,
10    "durability": 1,
11    "item_type": "HEALING"
12 }

```

Listing 5.6 – 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 dégâts ("double_damage"), ainsi que "combat_skill" qui représente le montant de ses dégâts.

```

1 {
2     "id": "zombie_captain",

```

```

3     "name": "Zombie Captain",
4     "hp": 15,
5     "double_damage": true,
6     "combat_skill": 2
7 }

```

Listing 5.7 – Exemple de personnage

Les `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 débiter notre aventure ainsi que le nombre que l'on doit en choisir ("`amount_to_pick`").

```

1 {
2     "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,
[...]",
3     "type": "SKILL",
4     "skills": [
5         "camouflage",
6         "hunting",
7         "sixth_sense",
8         "tracking",
9         "healing",
10        "weaponskill",
11        "mindshield",
12        "mindblast",
13        "animal_kinship",
14        "mind_over_matter"
15    ],
16    "amount_to_pick": 5
17 }

```

Listing 5.8 – 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, terminaux. Tous possèdent un texte. Les noeuds terminaux possèdent un type de fin ("`end_type`") afin savoir si l'on a gagné ou pas (cf : Listing 5.4). Les noeuds aléatoires eux, possède un attribut "`is_random_pick`" qui vaut true. Pour tous les autres types de noeuds, on retrouve parmi 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`"), des items disponibles à l'achat ("`shop`").

```

1 {
2     "text": "The back door opens [...]",
3     "items": [
4         {
5             "id": "gold",
6             "amount": 5
7         },
8         {
9             "id": "dagger"
10        },
11        {
12            "id": "seal_hammerdal"
13        }
14    ],
15    "amount_to_pick": 2
16    "choices": [
17        {

```



```

18         "text": "Return to the tavern.",
19         "section": "177"
20     },
21     {
22         "text": "Study the tomb.",
23         "section": "24"
24     }
25 ]
26 }

```

Listing 5.9 – Exemple de paragraphe

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

```

1 {
2     "text": "The dead zombies lie [...]",
3     "combat": {
4         "win": {
5             "text": "If you win the combat.",
6             "section": "309"
7         },
8         "enemies": [
9             "zombie_captain"
10        ]
11    }
12 }

```

Listing 5.10 – 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'une 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 aléatoire, un poids est également présent ("weight").

```

1 {
2     "text": "If you have the Kai Discipline of Tracking.",
3     "section": "182",
4     "hp": -5,
5     "requirements": [
6         [
7             {
8                 "id": "tracking",
9                 "type": "SKILL"
10            }
11        ]
12    ]
13 }

```

Listing 5.11 – Exemple de choix

ii La lecture et l'écriture

Du fait que la structure en Json n'est pas identique à celle détaillée 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 conversion 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.

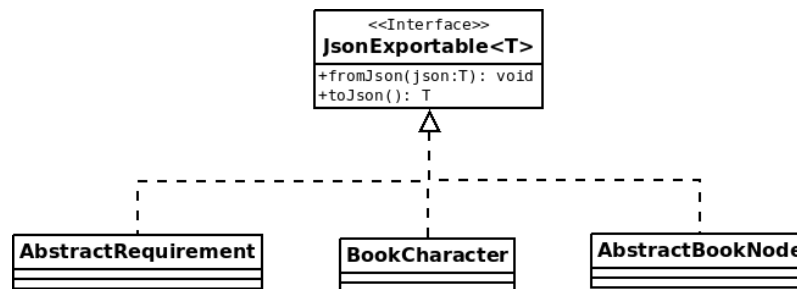


FIGURE 5.11 – L'interface *JsonExportable* et quelques classes qui l'implémentent

Enfin, les classes *BookReader* et *BookWriter* 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 :

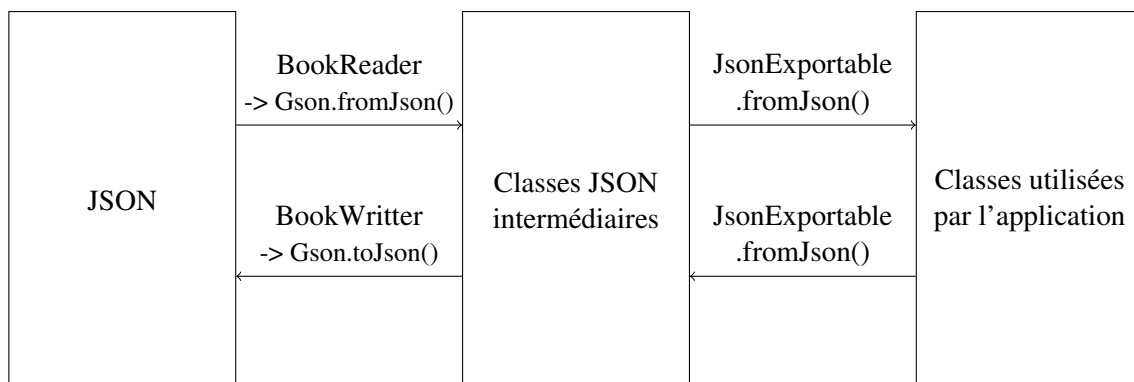


FIGURE 5.12 – Échanges pour la lecture / écriture

C Edition graphique d'un livre

Afin d'éditer un livre, nous avons différents boutons permettant de changer de mode. Chaque mode permet d'ajouter, de supprimer, de sélectionner, un noeud ou un lien. Ces modes sont gérés à l'aide de la *MainWindows* qui est une fenêtre qui affiche tout les différents panel. Il existe trois grands panels (Voir 5.13) comportant eux même d'autres petits panels. Le premier panel se nomme le **LeftPane**. Il se situe à gauche et est composé des boutons de mode, d'une liste d'item existant ainsi que d'une liste de personnage existant. Le deuxième panel s'appelle le **GraphPane**. Il est au centre et permet d'ajouter les noeuds ainsi que les liens entre les noeuds. Il contient également le préluce. Le troisième panel se nomme le **RightPane**. Il permet d'afficher les statistiques du livre comme par exemple, le nombre de noeud et l'estimation de la difficulté du livre.

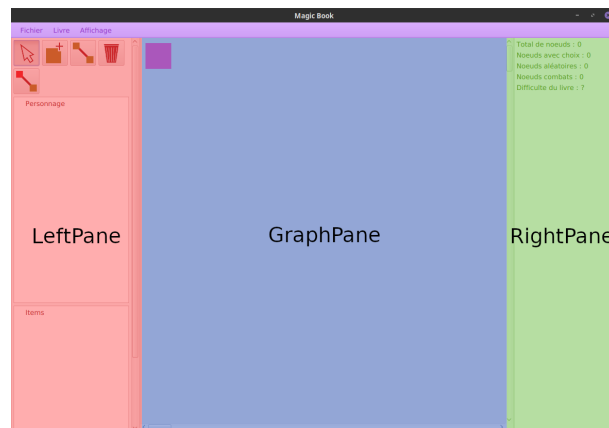


FIGURE 5.13 – MainWindows

i MainWindows

La MainWindows, comme mentionné précédemment, contient toute la fenêtre. Elle contient également un Menu permettant de réaliser plusieurs actions.

Premièrement, dans le menu nommé **Fichier**, l'utilisateur peut alors ouvrir un nouveau livre vide en cliquant sur **Nouveau**. Un nouveau livre est alors ouvert en mettant tout les panel (LeftPane, GraphPane, RightPane) à jour, permettant ainsi de remettre tout à zéro. Ce menu comporte aussi un MenuItem nommé **Ouvrir**, permettant ainsi d'ouvrir un fichier Json ou txt. Si ce fichier n'est pas bien indenté, il manque une virgule ou une accolade ou autre ne permettant pas la lecture du fichier en Json (Voir [B](#)), un message d'erreur apparaît. Si le fichier n'est pas bien fait, ne permettant pas l'ouverture de ce fichier, une classe nommé **BookValidator** devait "valider" le livre. Par manque de temps, cette classe n'est pas fonctionnelle. Si le fichier est bien indenté et peut s'ouvrir, le livre est changé en mettant tout les panel 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 toutes les deux la classe Jeu, décrit un peu plus loin [D](#). Un autre MenuItem est aussi présent, permettant de **Générer le livre en txt**. Cela permet à l'utilisateur de pouvoir avoir un livre propre contenant les paragraphes et le choix de chaque paragraphes.

Troisièmement, dans le menu nommé **Affichage**, l'utilisateur peut afficher ou non, le LeftPane et/ou le RightPane. Cela permet notamment 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 parmi cinq modes : **SELECT**, **ADD NODE**, **ADD NODE LINK**, **DELETE**, **FIRST NODE**. Chacun de ces mode fait appel à une méthode permettant de créer le bouton. Cette méthode prend en paramètre une image et un des modes de la classe Mode.

Ensuite, un TreeItem contenant des BookCharacter est créé permettant de lister tout les personnages créé à l'aide de TreeView. Le même procédé est réalisé pour les items.

Puis, pour permettre à l'utilisateur de créer des items et des personnages, un ContextMenu est ajouté au TreeView des items, puis au TreeView des personnages. Une action est implémenté pour chaque choix de ces ContextMenu : Ajout, Modification, Suppression. Vous trouverez ci dessous, un exemple pour les personnages.

```
1 TreeItem<BookCharacter> rootPerso = new TreeItem<> (new
    BookCharacter("0", "Personnage", 0, 0, null, null, 0));
```

```

2      rootPerso.setExpanded(true);
3      treeViewPerso = new TreeView<> (rootPerso);
4
5      ContextMenu contextMenuPerso = new ContextMenu();
6      MenuItem menuPersoAdd = new MenuItem("Ajouter un Personnage");
7      MenuItem menuPersoUpdate = new MenuItem("Modifier un Personnage"
8  );
9      MenuItem menuPersoDel = new MenuItem("Supprimer un Personnage");

```

Si le choix d'ajout est cliqué, une boîte de dialog apparait appelant `CharacterDialog` qui elle même appel `CharacterComponent` pour la création d'un personnage ou `ItemDialog` pour la création d'un item. Pour ce dernier, l'`ItemDialog` à une `ChoiceBox` permettant de choisir si l'item créer est une arme, un item de soin, de défense ou un simple item. Le `BookItem` est créer en fonction du type d'item.

The image shows two side-by-side dialog boxes. The left dialog is titled 'Ajout d'un item' and contains fields for 'Id', 'Name', 'Choix du type d'item' (a dropdown menu currently showing 'Weapon'), 'Point d'attaque', and 'Usure du matériel'. The right dialog is titled 'Ajout d'un personnage' and contains fields for 'Id', 'Name', 'Hp', and 'Dégats', along with a checkbox labeled 'Double dégats'. Both dialogs have 'Annuler' and 'Valider' buttons at the bottom.

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**. Chaque `accordion` 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.

Pour ajouter un noeud, on clique sur le mode **ADD NODE**, puis on clique sur le `GraphPane`. 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 de combat..... JUJU :p

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, un NodeFx est créer. 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'observateur 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. 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 premier 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 premier noeud est de type aléatoire, le joueur doit alors renseigner la chance sur chaque lien.

Le lien est représenté par une ligne entre le noeuds de départ et d'arrivé. Pour les différencier, un cercle est créer et est affiché au noeud de destination. Tout cela est géré par NodeLinkFx qui a aussi un observateur permettant d'envoyé 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.

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 remettent 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 *MainWindows*, 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;
12

```

Enfin, si des compétences et/ou des items sont disponible au début de la partie, la méthode de création de joueur est appelé en fonction 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 notamment 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 gagnée, win = true
14         } else {
15             //BookNodeTerminal FAILURE
16         }
17     }
18     return win;
19

```

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 destination 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 parmi 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'ID de l'ennemi 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 appelant 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 première 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 appelle *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 appelée afin de savoir si le player est encore en vie. Puis une autre méthode est appelée 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 terminée et renvoie un booléen sur l'état de la fin de partie.

ii Interface Player / Fournis

Une interface **InterfacePlayerFournis** a été créée permettant une mise en commun des codes Player et Fournis. Ces méthodes permettent de faire un choix, prendre les items disponibles, créer un personnage lambda, aller dans l'inventaire, choisir son ennemi ou encore combattre. Elles sont appelées au même moment. La méthode sera alors exécutée différemment en fonction du player.

execPlayerCreation permet de choisir les skills et les items disponibles au début de la partie. Ces derniers sont définis lors de la création du prélude. Pour l'ajout des items, la méthode *prendItems* est appelée.

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 choisit l'inventaire, on va alors dans une autre méthode appelée *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 choisit un autre choix, cet 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 tous les ennemis encore en vie.

prendItems permet de prendre un item parmi la liste d'items disponibles. Cette liste est prise 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 proposées par le noeud.

useInventaire 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 mise à jour. Si un item de soin est choisi, les points de vie du joueur sont alors actualisés.

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 affichés afin de guider le joueur dans ses choix.

Notamment 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 confirmer 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'il ne 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