



UNIVERSITÉ  
CAEN  
NORMANDIE

# Magic Book

Rapport de projet

DEROUIN Auréline 21806986

MARTIN Justine 21909920

THOMAS Maxime 21810751

# Table des matières

<b>1</b>	<b>Présentation du projet</b>	<b>1</b>
A	Présentation de l'application . . . . .	1
<b>2</b>	<b>Organisation du projet</b>	<b>2</b>
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
<b>3</b>	<b>Travail de groupe</b>	<b>5</b>
A	Répartition des tâches . . . . .	5
B	Idées d'améliorations . . . . .	7
C	Bugs et problèmes connus . . . . .	8
<b>4</b>	<b>Architecture du projet</b>	<b>9</b>
A	Arborescence du projet . . . . .	9
B	Présentation des packages . . . . .	9
<b>5</b>	<b>Aspects techniques</b>	<b>11</b>
A	Représentation d'un livre . . . . .	11
i	Représentation des noeuds . . . . .	11
ii	Représentation des liens . . . . .	13
iii	Représentation des personnages, items . . . . .	13
iv	Conditions pour accéder à un choix . . . . .	15
v	Prendre un choix de manière aléatoire . . . . .	16
vi	La classe Book . . . . .	17
vii	Le pattern observer et la classe Book . . . . .	20
B	Lecture et écriture d'un livre . . . . .	21
i	La structure du JSON . . . . .	22
ii	La lecture et l'écriture . . . . .	25
C	Edition graphique d'un livre . . . . .	26
i	La fenêtre principale . . . . .	26
ii	LeftPane . . . . .	27
iii	GraphPane . . . . .	28
iv	RightPane . . . . .	30
D	Rendre le livre jouable et estimer sa difficulté . . . . .	30
i	Jeu . . . . .	31
ii	Interface Player / Fournis . . . . .	37
iii	Player . . . . .	37

iv	Fourmis . . . . .	40
<b>6</b>	<b>Conclusion</b>	<b>44</b>

# 1 Présentation du projet

## A Présentation de l'application

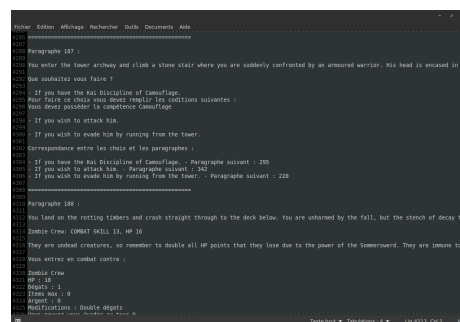
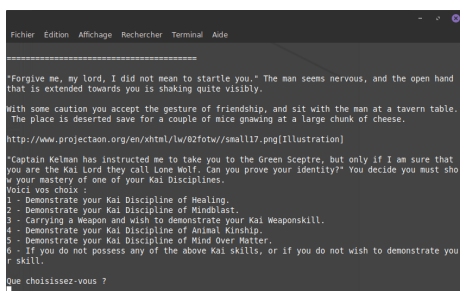
Magic Book est un éditeur de livre permettant de créer un livre à choix multiples pouvant contenir des conditions pour certains d'entre eux, des choix aléatoires, des combats, etc.

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.

Voici, quelques images de notre application :



## 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 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 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 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 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 dépôt 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 discussion 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
<b>Lecture et enregistrement des fichiers</b>				
Classes pour parser le JSON			X	X
Lecture d'un fichier JSON			X	
Enregistrement d'un fichier JSON			X	
<b>Livre</b>				
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
<b>Jeu et export au format texte</b>				
Classe Jeu, partie commune au joueur et la fourmis	X		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		X	
Création d'un Parser pour le texte			X	
Version primitive de l'estimation de la difficulté d'un livre			X	X
<b>Fenêtre</b>				
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		X	X	
Permettre d'editer ou supprimer un item ou un personnage du livre			X	
Lister, ajouter, modifier, supprimer les compétences			X	
Statistiques concernant les noeuds			X	
Statistique sur le niveau de difficulté du livre	X			
Cacher panel des statistiques si l'on décoche une case dans le menu			X	
Cacher le panel de gauche si l'on décoche une case dans le menu				X
Séparation des différentes parties de la fenêtre en plusieurs classes (LeftPane, GraphPane, RightPane)	X			



Tâches effectués	Auréline	Dimitri	Justine	Maxime
Composants réutilisables pour créer des personnages, créer une phase de la "Création du personnage", sélectionner une liste d'items			X	
Création d'une classe mère AbstractBookTreeView			X	
Création d'une classe fille pour l'affichage des items / personnages / compétences sous forme d'"arbre"			X	
<b>Boîtes de dialogue</b>				
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 les compétences			X	
Boîte de dialogue pour gérer le texte du prélude			X	
Onglet pour la conception des phases de "Création du personnage"	X		X	
Onglet pour le "Personnage principal"	X		X	
Gestion des prérequis dans les lien	X			
Gestion des items dans les noeuds			X	
Gestion des shops dans les noeuds	X			
<b>Zone d'édition</b>				
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			
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	
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			
<b>Autre</b>				

Tâches effectués	Auréline	Dimitri	Justine	Maxime
Rapport	X		X	~ <sup>1</sup>
Rédaction du support de soutenance			X	~ <sup>1</sup>
Mémo des diapositives			X	
Documentation Utilisateur	X		X	
Restructuration du livre d'exemple (fotw.json)			X	
Création de tests unitaires	X		X	
Javadoc	X		X	
Revue de code avant de merge			X	
Scripts	X		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, nous avons conçu un script, **scripts/stats.sh**, qui permet de savoir combien de lignes chaque personne a réellement ajouté et supprimé, le tout, sans compter certains fichiers.

Ainsi, voici ce que donne réellement le graphique :

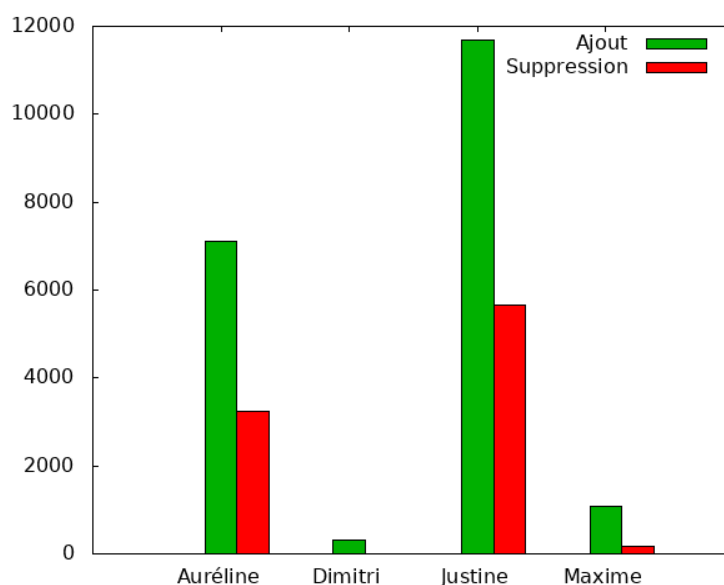


FIGURE 3.1 – Nombre de lignes ajoutés et supprimés par personne

## 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
- Déclencher plus d'exception si le livre est incorrect à la lecture / écriture

1. Mettre son nom. Pour le Rapport, ébauche de la section [Arborescence du projet](#)

- Pouvoir parser les compétences dans du texte
- Gérer les shops et les champs auto dans le jeu)
- Afficher les personnages et items inutilisés
- Indiquer si l'estimation de la difficulté est à jour ou non
- 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). De même pour les prérequis et l'édition du personnage principal.
- 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 déverrouiller 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 Jeu
- 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, ...)
- 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.

**DocUtilisateur** : Contient la documentation à destination de l'utilisateur.

**Rapport** : Contient le rapport du projet.

**Soutenance** : Contient le support et le mémo de la soutenance.

**scripts** : Contient différents scripts pour gérer le projet.

**.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](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).
    - booktreeview** : Les TreeView de l'application..
  - 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

Un livre peut contenir différentes informations. On y retrouve des items, des personnages, des paragraphes (que nous appellerons des noeuds), des choix (que nous appellerons des liens), ... Cette partie présentera la manière dont nous avons pensé ces différents éléments.

Un livre peut contenir différentes informations. On y retrouve des items, des personnages, des paragraphes (que nous appellerons des noeuds), des choix (que nous appellerons des liens), ... Cette partie présentera la manière dont nous avons pensé ces différents éléments.

#### i Représentation des noeuds

Nous étions d'abord parti sur une structure relativement simple pour gérer les noeuds. Ils comportaient un paragraphe, un type et une liste de choix (Voir figure 5.1)

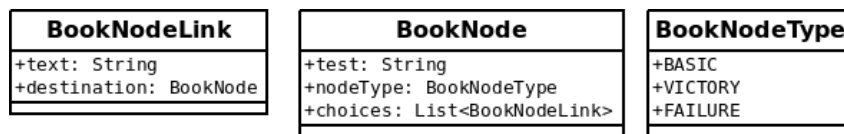


FIGURE 5.1 – Ancienne représentation des noeuds

Par la suite, nous avons voulu utiliser des classes afin de spécialiser certaines méthodes et de bien séparer les différents attributs dans différentes classes. Ainsi, seul la classe **BookNodeCombat**, représentant un noeud de combat, contient les informations sur le combat (liste d'ennemi, etc).

Maintenant, il y a donc quatre types de noeuds représentées par différentes classes (Voir figure 5.3) :

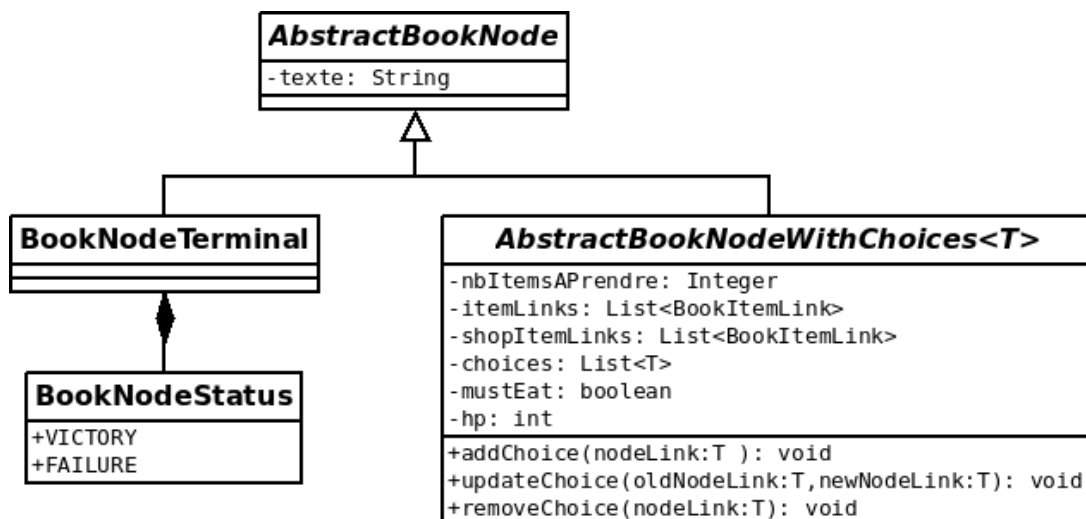


FIGURE 5.2 – Une partie de la nouvelle représentation des noeuds

Tout d'abord on retrouve **AbstractBookNode** qui contient un texte correspondant aux paragraphes du noeud. En découle ensuite une deuxième classe mère **AbstractBookNodeWithChoices<T>** qui

constitue la base de chaque noeud non terminal. Elle apporte divers attributs comme la liste des items disponible sur ce noeud, le nombre d'item maximum qu'on peut en prendre, la liste des items qu'il est possible d'acheter, le nombre de vie perdue / gagnée sur ce noeud, mais aussi, et surtout, la liste des choix disponible. Ce dernier est défini par une `List<T>` car nous aurons différents type de liens et nous souhaitons qu'un noeud ne puisse posséder qu'un et un seul type de lien au sein d'une même liste.

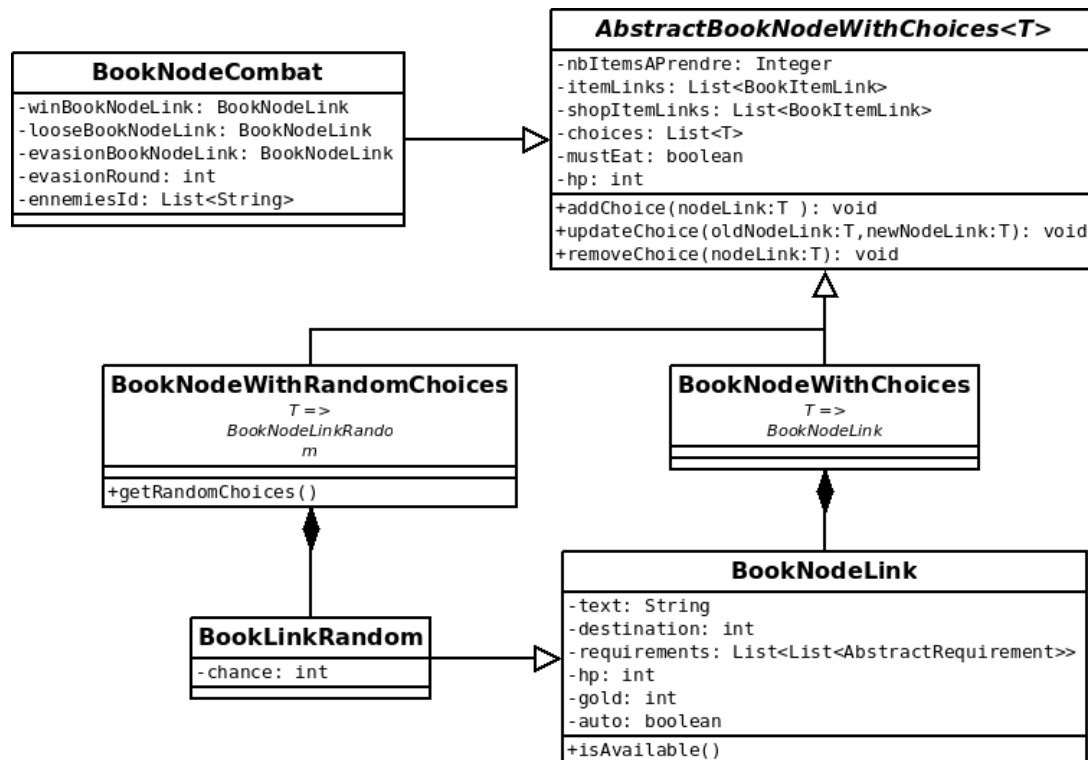


FIGURE 5.3 – L'héritage de `AbstractBookNodeWithChoices<T>`

Parmi les classes filles instanciable, on retrouve **BookNodeWithChoices** pour les noeuds à choix simple, **BookNodeWithRandomChoices** pour les noeuds où l'on prend un choix de manière aléatoire, **BookNodeCombat** pour les noeuds de combat et **BookNodeTerminal** pour ceux de victoire et de défaite.

**BookNodeTerminal** possède une énumération, **BookNodeStatus**, qui permet de connaître la finalité du livre, c'est à dire si le joueur a gagné ou perdu.

Pour le **BookNodeCombat**, le choix a été fait d'ajouter trois **BookNodeLink** afin de représenter les différentes issues du combat : victoire, evasion, défaite. Cela permet donc d'avoir, au maximum, un lien possible pour chaque finalité du combat. 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 plutôt 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. Nous n'aurions 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 pu changer cela. Une liste contenant les ID des ennemis que l'on doit combattre y est également renseignée. Les ennemis peuvent ensuite être retrouvé dans le livre à l'aide de cet ID.

Pour le **BookNodeWithRandomChoices**, la méthode `getRandomChoices` a été ajoutée, afin de sélectionner un choix de manière aléatoire en fonction de la probabilité de chaque lien.

## ii Représentation des liens

Les noeuds sont liés par des liens. Ces liens sont soit définis par la classe **BookNodeLink** ou **BookNodeLinkRandom**. Pour éviter les redondances, cette dernière étend 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 le **BookNodeLinkRandom**, la classe a besoin d'une variable pour gérer la probabilité, afin de définir la chance d'aller vers ce noeud. On comprend maintenant mieux l'utilisation de la généricité, car, si on ne le fait pas, il serait possible de mélanger des choix avec probabilité et d'autres qui n'en possèdent pas.

## iii Représentation des personnages, items

Nous allons maintenant parler des personnages et des items du livre. Bien qu'il n'y ait 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.

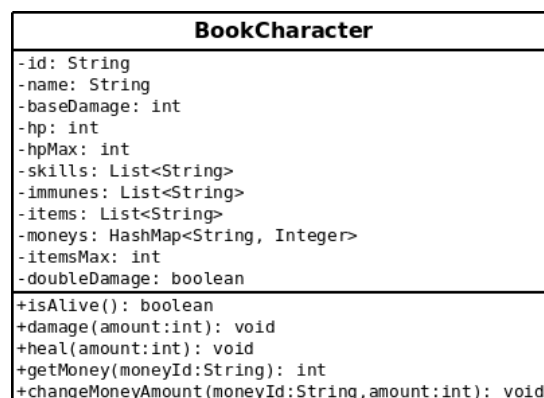


FIGURE 5.4 – UML sur la gestion des personnages

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`.



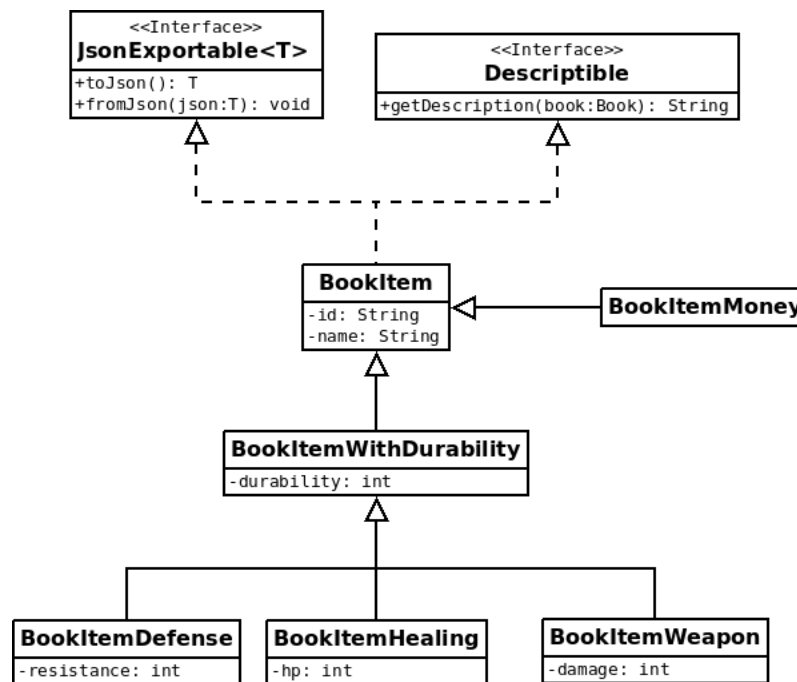


FIGURE 5.5 – UML sur la gestion des items

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 25. 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 spécifiques à notre classe fille. Par exemple, il serait étrange d'avoir un attribut pour savoir combien de dégât 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 notera l'appel à la méthode définit dans la classe mère grâce au mot clé *super*.

```

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 5.1 – Exemple de spécialisation des items

#### iv Conditions pour accéder à un choix

Nous avons décidé de fournir à l'utilisateur la possibilité de définir des listes de prérequis qu'il est nécessaire de remplir afin de pouvoir accéder à un choix. Ainsi, le joueur se verra refusé l'utilisation d'un choix s'il ne possède pas un item en particulier, une certaine somme d'argent, etc.

Les prérequis sont définis par une classe mère `AbstractRequirement`, donne lieu à une classe par type de prérequis. Chaque classe fille doit définir le comportement de la méthode `boolean isSatisfied()`. Ainsi, on retrouve **RequirementItem** pour les prérequis sur la possession d'un item, **RequirementMoney** pour la possession d'une somme d'argent ainsi que **RequirementSkill** pour la possession d'une compétence. Chacun d'entre eux possèdera donc l'id de l'élément qu'il doit vérifier, mais aussi le montant minimum nécessaire dans le cas de l'argent.

Cependant, comment savoir si le prérequis est satisfait ? Comment connaître l'état actuel du jeu ? C'est possible grâce à la classe **BookState** qui contient toutes les informations sur l'état actuel du jeu. Nos prérequis ne portant que sur le joueur principal il est donc uniquement nécessaire d'utiliser la méthode `getMainCharacter`.

Ainsi, pour savoir si un item / une compétence est possédé par personnage principal, il suffit de parcourir la liste de ceux que le personnage possède. Concernant l'argent, on récupère la somme de la monnaie possédée et on vérifie si elle correspond bien au minimum requis.

```

1 public boolean isSatisfied(BookState state) {
2     return state.getMainCharacter().getMoney(moneyId) >= amount;
3 }

```

Listing 5.2 – `RequirementMoney.isSatisfied()`

Nous avons souhaiter fournir encore plus de flexibilité à cette manière de faire, ainsi, il est possible d'avoir d'avoir une liste de prérequis à satisfaire afin de pouvoir accéder au choix voulu. Il est donc maintenant possible de créer un choix qui nécessite à la fois de "posséder un pass" et "50 euros" sur sois par exemple. Là encore, nous avons décidé de fournir une possibilité supplémentaire en permettant de proposer des conditions alternatives. C'es à dire de fournir une liste de prérequis alternative capable de permettre l'accès au choix. Nous avons donc différentes liste contenant elles même des conditions qu'il est nécessaire de satisfaire en même temps. Il est donc maintenant possible de faire des conditions comme suit : ("posséder un pass" ET "50 euros") OU ("pouvoir se téléporter") OU ("posséder 50 euros" ET "posséder 50 yen").

Ainsi, on comprend mieux le rôle de la méthode nommée `isAvailable()` dans **BookNodeLink** (cf : [L'héritage de `AbstractBookNodeWithChoices<T>` page 5.3](#)). Cette méthode se charge d'indiquer si une liste satisfait tous les prérequis. Bien entendu, une seule liste suffit à rendre le choix disponible et un seul prérequis non satisfait suffit à rendre la liste non satisfaite car il s'agit du principe même des OU et des

ET.

**Algorithm 1:** Disponibilité du choix

---

```

Input: state : BookState
Data: requirements : List<List<AbstractRequirement> liste des prérequis
1 if requirements.length == 0 then
2   | return True
3
4 for List<AbstractRequirement> groupRequirement in requirements do
5   | satisfied : bool
6   | satisfied = true
7   | for AbstractRequirement r in groupRequirement do
8     | if not r.isSatisfied(state) then
9       | | satisfied = false
10      | | break
11   | end
12   | if satisfied then
13     | | return true
14 end
15
16 return false

```

---

**v Prendre un choix de manière aléatoire**

Supposons que nous souhaitons faire un paragraphe durant lequel le joueur doit passer sur un pont en bois qui semble plutôt vieux. Il serait totalement ridicule de laisser au joueur la possibilité de choisir si le pont doit céder ou non. Ce choix relève plutôt de l'ordre de l'aléatoire. Nous savons donc maintenant que pour faire cela, une instance de **BookNodeWithRandomChoices** est nécessaire. Mais comment tirer un choix de manière aléatoire ?

Pour cela, on ajoute d'abord tous les liens disponibles, c'est à dire, tous les liens où le joueur peut avoir accès en fonction des prérequis demandés. Si aucun lien n'est disponible, on retourne null symbolisant une erreur. Dans le cas contraire, on commence par calculer la somme des probabilités sur les noeuds disponible. En tire ensuite un nombre aléatoire en fonction de cette somme. Enfin, il ne nous reste plus qu'à retrouver le choix correspondant en enlevant successivement la probabilité de chaque lien jusqu'à atteindre un nombre inférieur à zéro.

```

1 List<BookNodeLinkRandom> listNodeLinkDisponible = new ArrayList();
2 int somme = 0;
3 int nbrChoice = 0;
4
5 // On cherche les noeuds disponibles et on fait la somme des probabilités
6 for (int i = 0 ; i < this.getChoices().size() ; i++){
7   if(this.getChoices().get(i).isAvailable(state)){
8     listNodeLinkDisponible.add(this.getChoices().get(i));
9     somme += this.getChoices().get(i).getChance();
10  }
11 }
12
13 if(listNodeLinkDisponible.isEmpty()){
14   return null;
15 } else {
16   Random random = new Random();
17   int nbrRandomChoice;
18   // Si la somme des proba est à 0, on tire un noeud au hasard
19   if(somme == 0) {

```

```

20     nbrRandomChoice = random.nextInt(listNodeLinkDisponible.size());
21     return this.getChoices().get(nbrRandomChoice);
22 } else {
23     nbrRandomChoice = random.nextInt(somme);
24 }
25
26 // On cherche le choix tiré
27 for (int i = 0 ; i < listNodeLinkDisponible.size() ; i++){
28     if(!this.getChoices().get(i).isAvailable(state)){
29         continue;
30     }
31
32     nbrRandomChoice -= this.getChoices().get(i).getChance();
33     if(nbrRandomChoice < 0){
34         nbrChoice = i;
35         break;
36     }
37 }
38 }
39
40 return this.getChoices().get(nbrChoice);

```

Listing 5.3 – getRandomChoice()

## vi 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.

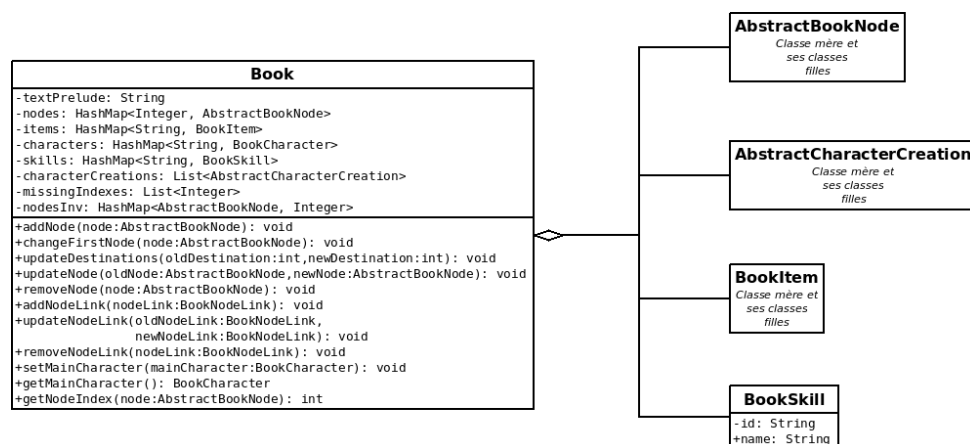
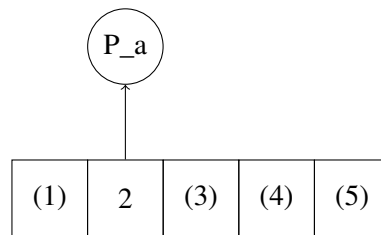


FIGURE 5.6 – UML sur la classe Book

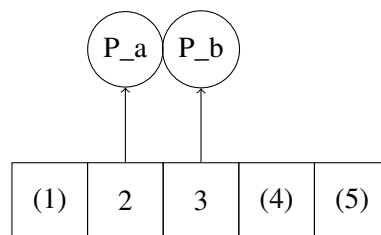
Par soucis de place, les getters et setter n'ont pas été renseignés. Il en va de même pour le détails des différentes classes (**AbstractBookNode**, **BookItem**, ...). Enfin, les observateurs sont volontairement omis car ils seront détaillés un peu plus tard (dans [Le pattern observer et la classe Book](#) à la page 20).

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

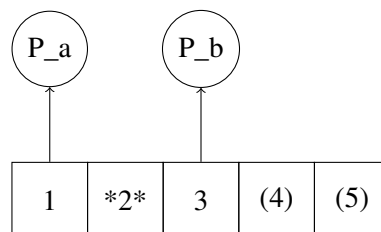
Cette HashMap possède un int comme clé, qui correspond au numéro du paragraphe. Lorsque l'on ajoute un noeud au livre, on doit d'abord lui trouver un numéro. Nous avons décidé de plusieurs règles. Les paragraphes commencent à partir du numéro 1. Le numéro 1 représente **toujours** le premier paragraphe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

**FIGURE 5.7** – Ajout du paragraphe A

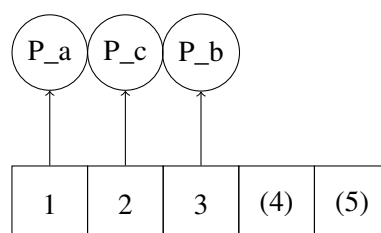
Les autres numéros sont représentés mais mis entre parenthèse car ils n'existent pas dans la Map. Ils sont uniquement là pour nous aider à bien visualiser ce dont on parle. Si nous décidons maintenant d'ajouter un second paragraphe alors celui-ci sera à la position 3.

**FIGURE 5.8** – 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 5.9** – 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 5.10** – Ajout du paragraphe C

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

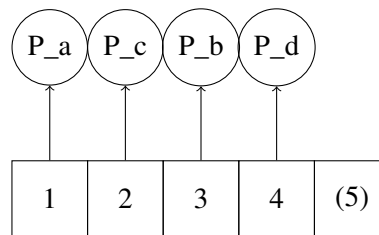


FIGURE 5.11 – Ajout du paragraphe D

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 miroir 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 identiques pour éviter tout bug.
- Pour déterminer l'indice d'un noeud que l'on ajoute, nous basons sur la taille de la Map.

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

---

**Algorithm 2:** 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  $2 + \text{tailleDuTableau}$  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 3:** 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

```

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)
```

---

*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 être 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 première "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.

## vii 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 :

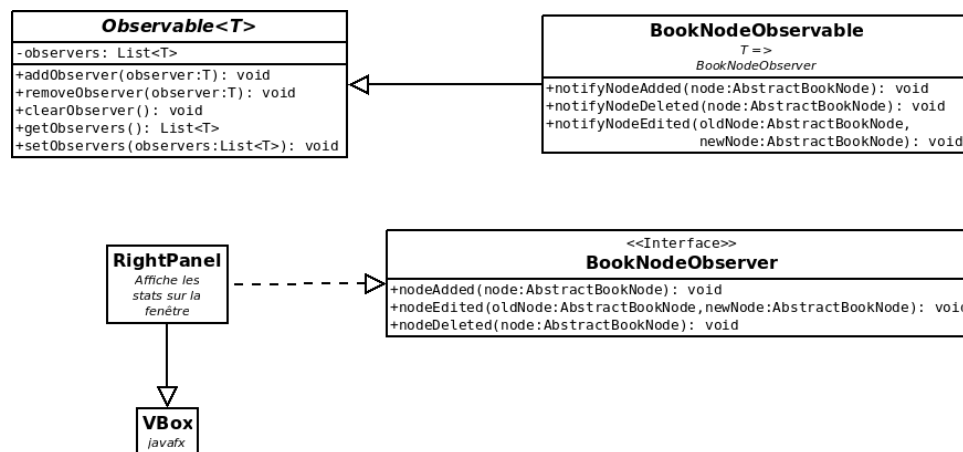


FIGURE 5.12 – UML d'exemple sur le pattern observer

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. Cependant, deux observers sont actuellement manquant, celui pour notifier d'un changement concernant le premier noeud et celui pour notifier un changement sur le personnage principal.

## 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 ressemblent :

```

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

```

Listing 5.4 – Exemple de livre très simple

## i La structure du JSON

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 compétences et items de départ.

Pour le moment les compétences sont uniquement composé d'un ID et d'un nom. Dans une future mise à jour il serait intéressant d'ajouter des propriétés pour connaitre la force 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 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 maximum que l'on peut 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. Pour rappel, il existe différents types de paragraphes : à choix, à choix aléatoire, avec des combats et 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, possèdent 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") et enfin 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 aléatoires, une probabilité est également présente ("weight").

```

1 {
2   "text": "If you have the Kai Discipline of Tracking.",
3   "section": "182",
4   "hp": -5,
5   "requirements": [
6     [
7       {

```

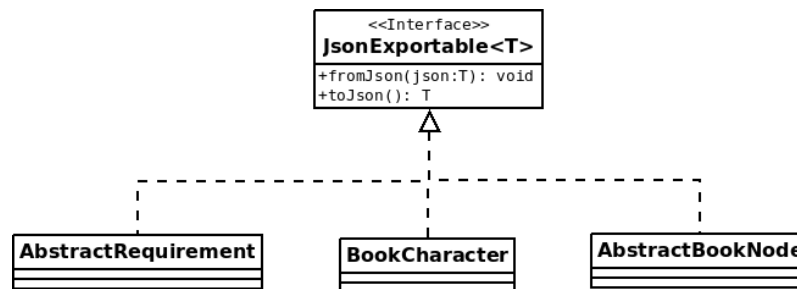
```

8         "id": "tracking",
9         "type": "SKILL"
10    }
11  ]
12 ]
13 }
```

**Listing 5.11** – Exemple de choix

## ii La lecture et l'écriture

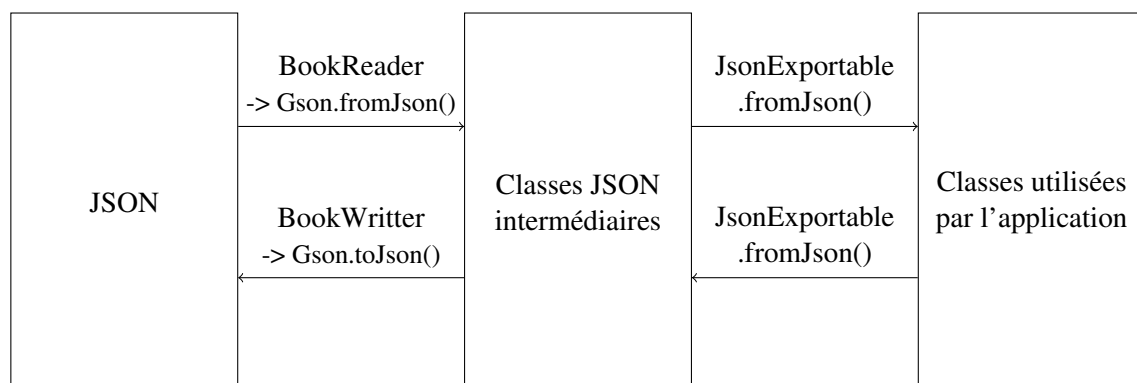
Du fait que la structure en Json n'est pas identique à celle détaillé dans ?? (page ??), nous avons fait des classes intermédiaires pour permettre cette lecture. Celles-ci sont disponibles dans le package *magic\_book/core/file/json* et ne contiennent rien de plus que des getter et setter. Aussi, afin de permettre une 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 5.13** – 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 5.14** – Échanges pour la lecture / écriture

## C Edition graphique d'un livre

### i La fenêtre principale

Notre fenêtre principale est définie par la classe `MainWindow` est notre fenêtre principale. Elle contient un Menu permettant de réaliser plusieurs actions (ouvrir et sauvegarder un fichier, exporter le livre, jouer, estimer la difficulté, ...) ainsi que trois zones différentes, divisées en trois autres classes.

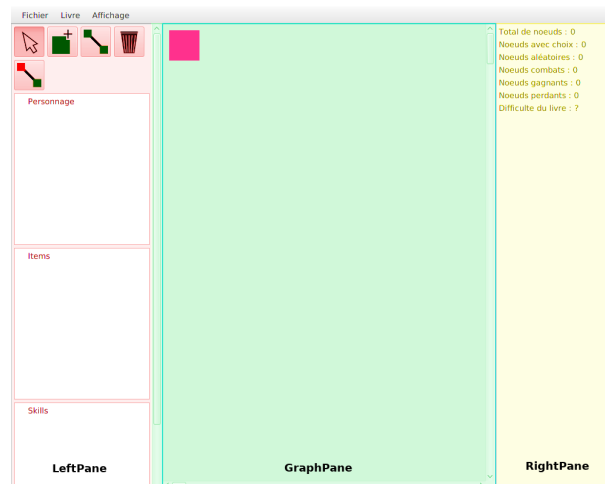


FIGURE 5.15 – MainWindow

La partie de gauche est représentée par la classe **LeftPane**. Celle-ci est composée de différents boutons permettant de changer de mode, d'une liste d'items, de personnage, de compétences constituant le livre. De l'autre côté, à droite, on retrouve une instance de **RightPane**. Cette partie permet d'afficher les statistiques du livre comme par exemple, le nombre de noeud ainsi que l'estimation de la difficulté du livre, etc. Enfin, la partie centrale définie par le **GraphPane** permet d'éditer sous forme graphique le livre. Ces trois parties utilisent le Pattern MVC ce qui leur permet également d'être réactif aux changements du livre notamment grâce au Pattern Observer fournis par celui-ci (cf : [Le pattern observer et la classe Book](#) page 20).

### Ouverture et sauvegarde du livre

Avant de démarrer, il faut préciser qu'une variable *path* est présente dans **MainWindow** et permet de sauvegarder le fichier actuellement ouvert dans l'éditeur. Celle-ci vaut null lorsqu'aucun fichier ne l'est. C'est donc sa valeur au lancement de l'application.

L'utilisateur peut démarrer un nouveau livre, à l'aide du menu, en cliquant sur Fichier -> Nouveau. La classe **MainWindow** fait alors une nouvelle instance de la classe `Book` puis appelle sa méthode `void setBook(Book)` qui se chargera de transmettre à tous les panels (`LeftPane`, `GraphPane`, `RightPane`) le nouveau livre. Les panels propageront alors le livre aux différents éléments qui pourraient en avoir besoin (tel que les listes d'items, de personnages, ...). Enfin, la variable *path* est mise à null.

Il est possible d'ouvrir un livre à l'aide du menu Fichier -> Ouvrir. Une boîte de dialogue s'affiche alors afin de sélectionner le livre que l'on souhaite éditer. Si un livre a bien été sélectionné, alors, **BookReader** sera appelé afin de le lire (cf : [La lecture et l'écriture](#) page 25). Si le fichier est invalide, au niveau de la structure du JSON, une erreur s'affiche et le processus d'ouverture s'en arrêtera là. Une fois de plus, la méthode `void setBook(Book)` de **MainWindow** est appelé afin de transmettre le nouveau livre. Enfin, si l'on a pu faire toutes ces actions sans aucun soucis, la variable *path* retient le chemin vers le

fichier ouvert.

Enfin, il est possible grâce à Fichier -> Enregistrer ainsi que Fichier -> Enregistrer sous d'enregistrer les changements sur le fichier. La sauvegarde d'un fichier se déroule comme suit ; On commence par vérifier si un fichier est déjà ouvert, grace à *path*. Si ce n'est pas le cas, alors on affiche une fenêtre afin de sélectionner l'emplacement de sauvegarde. Si l'utilisateur valide son choix, alors on doit changer la valeur de *path* pour enregistrer cet emplacement. On fera ensuite appel à la classe **BookWriter** pour sauvegarder le livre au format JSON (cf : [La lecture et l'écriture](#) page 25). Cependant, il existe une différence entre Fichier -> Enregistrer et Fichier -> Enregistrer sous. En effet, l'étape de la vérification "du fichier déjà ouvert" ne se fait que pour le premier des deux tandis que l'autre affichera donc toujours la fenêtre de sauvegarde peu importe la valeur de *path*.

## ii LeftPane

### Sélection du mode

Comme mentionné auparavant, ce panel contient plusieurs boutons cliquable en haut afin de sélectionner le mode d'édition. Pour cela, nous avons utilisé des **ToggleButton**, lesquels sélectionnent l'un des 5 modes disponibles par l'énumération **Mode** à savoir : SELECT, ADD\_NODE, ADD\_NODE\_LINK, DELETE, FIRST\_NODE. Ces boutons faisant strictement la même chose, ils sont tous créé à partir de la méthode *ToggleButton createToggleButton(String, Mode)*. Cette méthode prend en paramètre le chemin vers une image ainsi que le mode que l'on souhaite sélectionner lorsque l'on clique dessus, mode qui sera appliqué au **GraphPane**. Afin de ne sélectionner qu'un seul **ToggleButton** à la fois, tous, sont contenu dans un seul et même **ToggleGroup** qui se chargera lui même de désélectionner le **ToggleButton** précédemment sélectionné.

### Gestion des éléments du livre

En dessous des boutons pour sélectionner le mode, nous retrouvons trois listes permettant chacune la gestion d'un des éléments du livre, c'est à dire, des personnages, des items et des compétences.

Bien qu'il s'agisse d'un arbre, au vue du fait qu'ils héritent de **TreeView<T>**, nous avons fait un affichage qui correspond plus à celui d'une liste. Pourquoi ce choix ? Nous aurions voulu pouvoir par la suite organiser cette vue de manière à ce qu'elle puisse contenir différentes catégories tel que, pour les items par exemple, Autre, Arme, Défense, (...) correspondant aux différents types d'items possibles. Chaque catégorie contiendrait alors les items correspondant à ce type. Ainsi, l'UML de ces classes ressemble à celui ci :

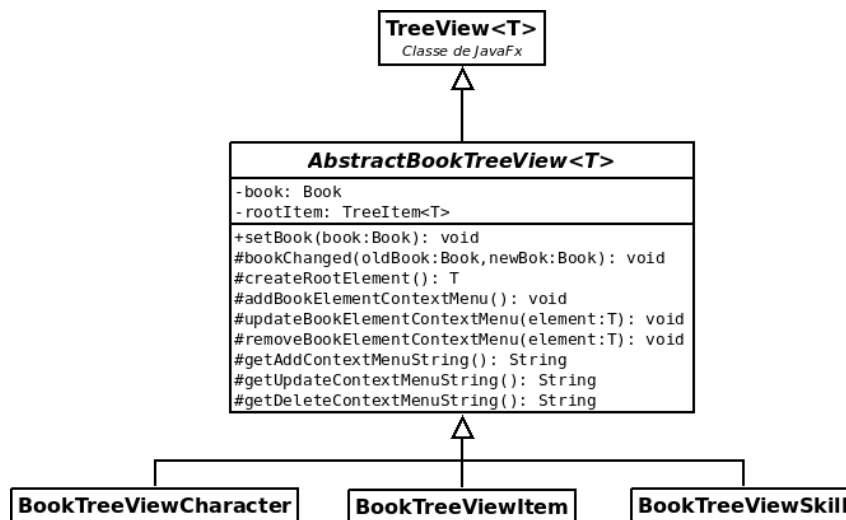


FIGURE 5.16 – UML des TreeView

Comme on pouvait s'en douter, chacune de ces liste hérite de d'une classe mère, **AbstractBookTreeView<T>**, permettant de gérer les comportement en commun. L'utilisation de la généricité permet de définir le type d'élément que la liste se chargera de gérer. Détaillons maintenant les différentes méthodes à redéfinir.

Comme pour tous les autres éléments graphique, les listes sont conçue en utilisant le Pattern MVC et donc également, le Pattern Observer. De ce fait, ils réagissent automatiquement à un changement sur l'éléments qu'ils permettent de gérer (items, personnages, compétences).

La méthode *void setBook(Book)*, permet comme pour tous les autres éléments, de changer le livre actuel. Cependant des actions supplémentaires étaient requises par les listes afin notamment de ne plus écouter les événements sur l'ancien livre ainsi que d'afficher les éléments concernant le nouveau. Nous avons donc conçu la méthode *void bookChanged(Book, Book)* qui est automatiquement appelée par *void setBook(Book)* et permet à la liste d'effectuer ces changements. Nous avons préféré fonctionner de cette manière plutôt que d'utiliser *super* car cela oblige à fournir un comportement pour cette méthode et évite également l'oubli de la redéfinition de *void setBook(Book)* pour traiter correctement le livre et réaliser les actions précédemment évoquées. La méthode *T createRootElement()* quant à elle, permet de créer l'élément servira de label dans le *TreeItem*. La seule vraie valeur que l'on doit y renseigner est le nom de l'élément, les autres valeurs peuvent valoir null car elles ne seront pas utilisées. Les méthodes *void addBookElementContextMenu()*, *void updateBookElementContextMenu(T)*, *void removeBookElementContextMenu(T)* correspondent aux actions à exécuter lorsque l'on clique sur l'action ajouter, modifier ou supprimer du menu lors d'un clic droit sur la liste. Enfin, les méthodes *String getXXXContextMenuString()* permettent de récupérer le texte qui sera affiché dans le menu qui vient d'être évoqué.

### iii GraphPane

Le GraphPane correspond à la zone d'édition. Elle contient des méthodes permettant l'ajout de lien ainsi que l'ajout de noeud. C'est ici que le mode va prendre tout son sens et toute son importance. En effet, un clic sur cette partie de l'application n'aura pas les mêmes effets selon que l'on soit en mode "ADD\_NODE" ou "DELETE".

## Représentation graphique des éléments

Notre application ayant besoin de représenter dans la fenêtre les différents éléments tel que le pré-lude, les noeuds et les liens entre eux, voici la manière dont nous avons procédé.

Commençons par analyser le diagramme montrant l'architecture pour les noeuds :

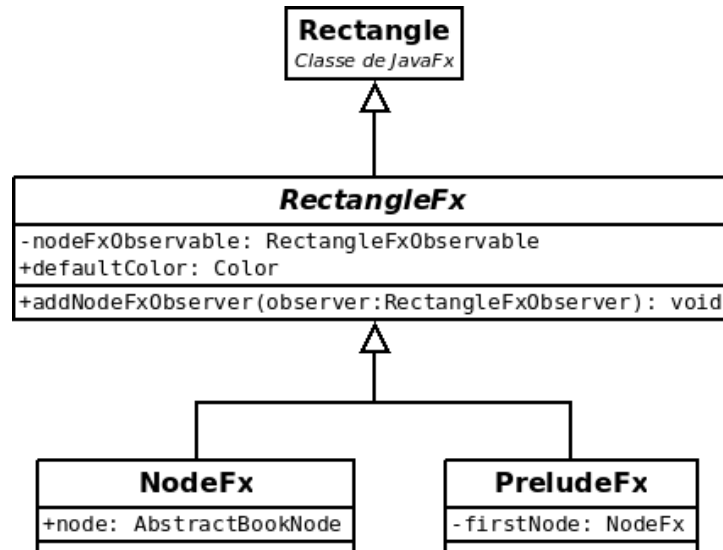


FIGURE 5.17 – UML sur la représentation des noeuds

Nous avons décidé de fournir un wrapper à la classe **AbstractBookNode** afin de les représenter. Nous avons appelé cette classe **NodeFx**. Celle-ci hérite de **Rectangle** qui est fournis par JavaFx. Afin d'éditer les différents éléments du livre il est également nécessaire de fournir un élément visuel représentant le départ du livre et permettant d'éditer les éléments tels que le prélude, le personnage principal, etc. Ceci est possible grâce à **PreludeFx**. Les deux classes ayant des éléments en commun (une couleur et des observers, lesquels seront détaillé après), nous avons donc décidé d'ajouter une "couche" supplémentaire entre **Rectangle** et nos deux classes. C'est ce à quoi sert **RectangleFx**. Chacune des classes filles possède en attribut les éléments qu'elle à besoin, c'est à dire, le noeud représenté pour **NodeFx** tandis que **PreludeFx** a besoin, elle, du premier noeud du livre.

Passons maintenant à la représentation des liens :

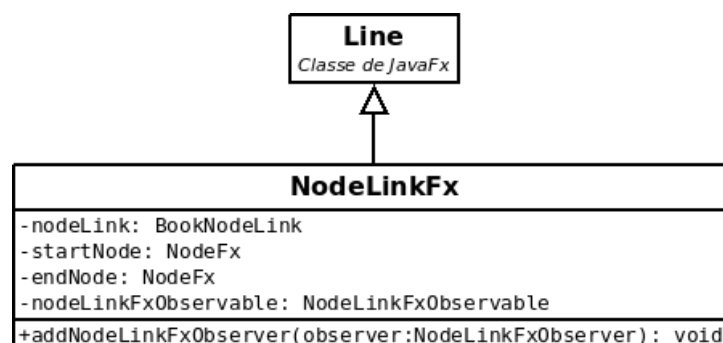


FIGURE 5.18 – UML sur la représentation des liens

Comme on peut le voir, ceux ci héritent de la classe **Line**. De la même manière que **NodeFx**, la classea comme attribut l'objet qu'elle représente, c'est à dire une instance de **BookNodeLink**. On va également retrouver le noeud de départ et de fin ce qui permet de facilement connaitre ces noeuds lorsque



l'on éditera le lien pour, par exemple, ajouter des champs supplémentaires si c'est un noeud de combat ou un noeud aléatoire.

Voici quelques informations supplémentaire :

- Il est important de noter que l'héritage fait avec les éléments de JavaFx permet d'ajouter nos classes comme on pourrait ajouter n'importe quel autre élément de cette librairie.
- L'utilisation du Pattern Observer est fait de tel manière que l'on puisse être notifié dès que l'on clique sur l'élément sans avoir besoin de passer par les événements de JavaFx.

## Gestion des évènements

On distingue l'écoute d'évènement à trois endroits. Sur la zone d'édition, sur un noeud et sur un lien.

Sur la zone d'édition nous surveillons notamment les cliques lorsque nous sommes en mode "ADD\_NODE". Dans le cas où un clique intervient, nous affichons la boîte de dialogue permettant l'ajout d'un noeud puis, si nous avons validé cette dernière, nous ajoutons alors le noeud au livre. On recevra alors une "notification" grâce à la méthode prévue à cet effet par le Pattern Observer que le noeud a été ajouté et qu'il faut donc l'afficher.

Lorsque nous créons un noeud, nous devons faire en sorte qu'il réagisse avec les prochains évènements, de ce fait, nous ajoutons le **GraphPane** à la liste des observers du noeud. Dès lors, un clique sur ce noeud sera automatiquement perçue et traité dans la méthode correspondante.

Afin de réaliser un lien entre les deux noeuds, il faut cliquer sur un noeud puis sur un second tout en ayant sélectionné le mode "ADD\_NODE\_LINK". On retient dans une variable le premier noeud cliqué. Lors d'un clique sur un noeud, cette variable prend alors la valeur de celui-ci et rien de plus ne se passe. Lorsque l'on cliquera de nouveau sur un noeud, alors, comme la variable ne vaut plus "null", on affichera la boîte de dialogue permettant de remplir les informations sur le lien entre ces deux noeuds. Comme pour le noeud, si on valide la boîte de dialogue, alors on ajoute à la classe **Book** le lien et une "notification" est perçue par la méthode correspondante.

De même, afin de recevoir les événements concernant le lien, on enregistre le **GraphPane** comme étant observer de celui-ci. Cela nous permet maintenant de réagir par exemple à la demande de suppression d'un noeud.

## iv RightPane

Ce panel contient toutes les statistiques concernant le livre. Celles-ci sont immédiatement mise à jour lorsqu'un élément du livre change grâce, comme mentionné précédemment, au Pattern Observer.

Pour le moment, les statistiques ne concernent que le nombre de noeuds de chaque type ainsi que la difficulté du livre. Dès qu'un noeud est ajouté, on incrémente les compteurs correspondant. De même, dès qu'un noeud se retrouve supprimé, on décrémente les compteurs qui doivent l'être. Un changement de livre reset les compteurs et, compte le nombre de noeud de chaque type. C'est le seul moment où l'on va parcourir tous les éléments du livre. En effet, le système de compteur permet de gagner un gain de temps sur des livres contenant de nombreux noeuds. Pour la difficulté du livre, il est possible de mettre à jour la valeur affichée avec la méthode *void difficultyChanged(float)*. C'est notamment ce que l'on fait une fois que celle-ci a fini d'être calculée avec le menu Livre -> Estimer la difficulté.

## D Rendre le livre jouable et estimer sa difficulté

## i Jeu

Une classe a été créée, se nommant **Jeu**, permettant de gérer les méthodes de jeu communes entre les classes *Player* et *Fourmis*. Cela permet donc d'avoir une classe qui gère tout le jeu en faisant appel aux différentes méthodes et classes en fonctions du types de noeud et du type de joueur.

Un construteur est d'abord appelé, à partir de la *MainWindow*, afin d'envoyer le livre contenant toutes les informations. Puis, en fonction du mode sélectionner (**Générer la difficulté** ou **Jouer**), la méthode correspondante au joueur est appelé.

Soit la méthode *play()* est appelé si **Jouer** est sélectionné. C'est donc le player qui va jouer. Cela veut dire que les messages vont alors s'afficher dans le terminal (*showMessages*) et le player est défini sur la classe *Player*. Cela permettra d'appeler le bon joueur lors de l'appellation des méthodes où un choix est nécessaires. Une fois le jeu lancer grâce au *runGame()*, expliqué dans le code 5.15 et les ressources temporaires qui ont été généré sont supprimés.

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

**Listing 5.12 – play()**

Soit la méthode *fourmis()* est appelé si **Générer la difficulté** est sélectionné. Ce sont donc plusieurs fourmis qui vont jouer une par une afin de générer la difficulté du livre. Les messages sur le terminal sont donc non affiché. Puis vient le lancement des fourmis. Pour cela une boucle "for" est simplement nécessaire. Le player est donc défini sur la classe **Fourmis** permettant ainsi d'utiliser les méthodes de cette classe afin de réaliser des choix random. Puis la méthode *runGame()*, expliqué dans le code 5.15, renvoi donc la finalité du livre : soit un boolean true en cas de victoire ou false en cas de défaite.

On a donc une classe qui permet réaliser des choix random ainsi qu'une méthode qui renvoie la finalité du livre. Cela permet donc de lancer un certain nombre de "fourmis" qui vont parcourir tout le livre, réalisé des choix, gagner ou perdre. Toutes les victoires vont être comptés, ce qui va permettre de savoir combien de fourmis ont gagnés. Soit calculé la difficulté du livre.

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

**Listing 5.13 – fourmis()**

Une fois dans la méthode choisie, la méthode *runGame()* est alors appelé. C'est la méthode principal de la classe **Jeu** car c'est elle qui lance le jeu. Tout d'abord, la méthode copie le livre afin de ne pas le

modifier par erreur pendant le jeu. Car même si on a fait attention, il suffit d'une erreur pour modifier tout le livre principal qui gère le jeu. Puis un `BookState`, qui correspond à l'état de la partie, est créé. Ce `BookState` permettra donc de suivre l'état du livre ainsi que du joueur (son inventaire, sa santé...). D'ailleurs, le joueur enregistré dans le `BookState` est soit le personnage principal créé dans le prélude, soit, si inexistant, un personnage préexistant permettant ainsi de jouer au jeu sans pour autant créer un personnage principal (voir méthode ci-dessous). Enfin, si des compétences et/ou des items sont disponibles au début de la partie, la méthode de création de joueur (*execPlayerCreation*, voir ??) est appelée en fonction du joueur actuel.

```

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

Listing 5.14 – createNewState()

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

```

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;
```

Listing 5.15 – Méthode runGame()

Pour chaque type de noeud, sauf pour un noeud terminal car il ne contient pas ces options, une méthode commune, nommée *execAbstractNodeWithChoices()* est appelée afin de savoir si le noeud pris en charge fait gagné/perdre de la vie (méthode *execNodeHp()*), puis regarde si le joueur est toujours

en vie. Cela permet donc que chaque exécution de noeud vérifie ceci, permettant ainsi de minimiser la redondance. Si le joueur n'est plus en vie, un noeud terminal de défaite est alors renvoyé en noeud de destination, permettant ainsi de notifier la défaite du joueur. S'il est encore en vie et que le noeud propose des items, ils sont proposés au joueur en appelant la méthode correspondante entre **Fourmis** ou **Player**. Ces classes sont appelées car le player doit faire un choix grâce au Scanner et la fourmis doit effectuer un choix random.

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

**Listing 5.16** – Méthode `execAbstractNodeWithChoices()`

Puis, à chaque destination du joueur, sauf si celui-ci a été entraîné vers un noeud terminal de défaite créer par la classe jeu et donc, non existante, une autre méthode est appelée (nommée `execBookNodeLink()`) afin de regarder si le lien entre le noeud de départ et de destination fait perdre ou gagner de la vie et/ou de l'argent. Si c'est le cas, le joueur est alors "mis à jour". Si le joueur n'a plus de vie, même si le lien le portait vers un noeud de victoire, un noeud de défaite est créé. Le joueur est considéré alors comme perdant. Cela permet donc d'avoir une méthode complète pour la gestion des options dans les liens.

**Si un noeud est de type basic**, il est alors pris en charge dans la méthode `execNodeWithChoices()`, comme vous pouvez le constater dans la figure 5.21. La méthode vérifie si au moins un choix est possible, permettant ainsi de créer, ou non, un noeud terminal de défaite. S'il peut au moins choisir une destination, un choix est demandé parmi toutes les destinations faisant un appel à la méthode en fonction du joueur (`player.makeAchoice()`). Car ici, un choix est demandé, soit le Scanner du player ou le random de la fourmis. Une fois le choix décidé, si le joueur a les prérequis pour aller vers cette destination (si `isAvailable()` renvoie true), alors la méthode `execBookNodeLink()`, vu juste en dessous de la figure du haut, est appelée. Si le joueur est toujours en vie, le noeud choisi est renvoyé en noeud de destination. Sinon, le joueur doit faire un autre choix car il y a minimum un choix possible.

```

Votre personnage :
Personnage Test
HP : 150
Dégats : 5
Items max : 5
Argent : null
Modifications : Double dégats

Le marchand vous propose du troc. Une émeraude contre l'épée "Aiguille"
Voici vos choix :
1 - L'échange est convenable.
2 - C'est du vol !

Que choisissiez-vous ?
<=====--> 75% EXECUTING [8m 12s]
Vous devez posséder l'item Aiguille

Que choisissiez-vous ?

```

FIGURE 5.19 – Jeu prérequis

Le problème de la méthode *isAvailable()*, c'est que cela peut créer un livre sans fin. En effet, prenons l'exemple d'un joueur avec un espace inventaire maximal de 1. Le joueur a un item, dans le premier noeud, "Potion" et décide de prendre l'item. Il n'a donc plus de place dans son inventaire. Le deuxième noeud, il décide de prendre l'item "Arme" car il veut pouvoir se défendre, l'item "Potion" est donc supprimé. Sauf qu'il se retrouve en face de deux choix : Choix 1 avec de prérequis "Potion" qu'il vient de supprimer, ainsi que le choix 2 qui le renvoi sur ce noeud. Le problème c'est que la méthode *isAvailable()* renvoie true, car le joueur peut accéder au choix 2. Le joueur est donc coincé. Cela pose un problème que cela soit dans la génération de la difficulté et la jouabilité du livre.

Cela ouvre donc une piste de réflexion. Il faudrait peut être envoyer des fourmis à chaque "destination" de noeud trouvé. Si une fourmi trouve une boucle infini (retombe sur un même noeud "n" fois, avec un nombre assez grand permettant d'être sûr que c'est bien une boucle infini), elle renvoie un "false" et le numéro du noeud. Si le joueur tombe sur ce paragraphe, un noeud terminal défaite est généré en indiquant sur celui ci "le livre n'est plus possible, il vous manque [prerequi]". Ces fourmis et ce boolean sera généré a chaque destination chargé. La fourmi sera la copie exacte du personnage au moment même du chargement du noeud de destination. Mais cette méthode ralentirais l'application et une autre, plus simple, serait peut être possible.

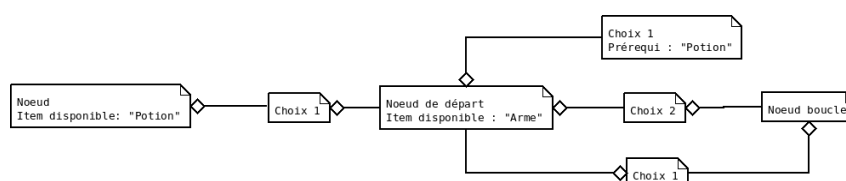


FIGURE 5.20 – Exemple de boucle infini

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

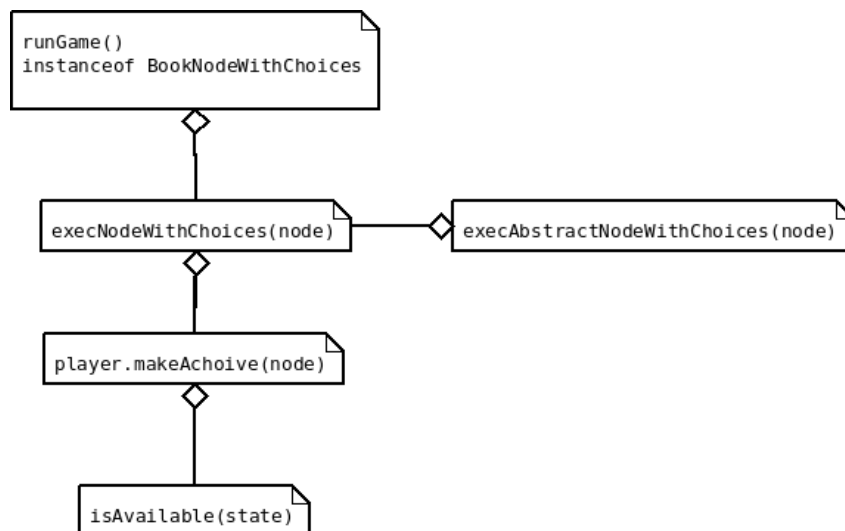


FIGURE 5.21 – Lien entre les méthodes pour un noeud basique

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

**Choix Attaque :** que ce soit dans la classe **Player** ou **Fourmis**, un autre choix (cf ligne 2) est demandé permettant de sélectionner l'ennemi à attaquer parmi la liste des ennemis encore en vie. Cela permet donc de créer un dynamisme et une certaine intelligence dans le combat. Je vous invite à voir les subsections **fourmis iv** et **player iii** pour plus de détail dans le choix. Une fois l'ennemi sélectionné, une méthode attaque (cf ligne 7) est appelée en appelant elle-même une autre méthode commune entre l'attaque du joueur et l'attaque d'ennemi. Nommée *getDamageAmount()*, elle permet de savoir le nombre de dommages réalisés en fonction des points d'attaque de l'attaquant, de son double dommage décidé en random si ce boolean est défini en true, d'un coup critique décidé 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'appelle. Ici c'est la méthode d'attaque du player. Cette méthode commune permet donc d'éviter une répétition dans le code. Une fois l'attaque effectuée, si l'ennemi attaqué est mort, il est supprimé de la liste des ennemis (cf ligne 9) avant son tour pour éviter que l'ennemi mort attaque ou que l'on rattrape cet ennemi.

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

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

```

1 while(!finCombat){
2     ChoixCombat choixCombat = player.combatChoice(node, evasionRound, state);
3
4     if (choixCombat == ChoixCombat.ATTAQUER) {
5
6         BookCharacter ennemi = player.chooseEnnemi(listEnnemis);
7         attaque(ennemi);
8         if(!ennemi.isAlive()){
9             listEnnemis.remove(ennemi);
10        }
11    }
12    else if(choixCombat == ChoixCombat.EVASION) {
13        if(evasionRound <= 0 && node.getEvasionBookNodeLink() != null){
14            execBookNodeLink(node.getEvasionBookNodeLink());
15            return book.getNodes().get(node.getEvasionBookNodeLink().
16                getDestination());
17        }
18    }
19    ennemiTour(listEnnemis);
20
21    if(!state.getMainCharacter().isAlive()) {
22        //Vérification d'un noeud Failure
23        if(node.getLooseBookNodeLink() != null) {
24            execBookNodeLink(node.getLooseBookNodeLink());
25            return book.getNodes().get(node.getLooseBookNodeLink().
26                getDestination());
27        } else
28            return new BookNodeTerminal("Vous succombez à vos blessures",
29                BookNodeStatus.FAILURE);
30    }
31    if(listEnnemis.isEmpty())
32        finCombat = true;
33    evasionRound -= 1;
34 }

```

Listing 5.17 – JeuCombat

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

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

**Si le noeud est de type aléatoire**, la méthode commune est appelée afin de savoir si le joueur est encore en vie. Puis une autre méthode, nommée `getRandomChoices()` (cf [getRandomChoice\(\)](#) à la page 16) est appelée de la classe `BookNodeLinkRandom` afin de déterminer le noeud de destination en fonction des chances attribuées 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. Ce booléen permet de quitter la boucle `while` de la méthode `runGame()` et permet ainsi donc d'arrêter le jeu. L'utilisateur peut alors continuer l'édition du livre pour le player et permet de savoir si la partie est gagnée ou perdue pour la fourmi (voir la subsection fourmi [iv](#)). Bien sûr, une vérification si le joueur est en vie est également exécutée si c'est un noeud de type VICTORY, permettant ainsi que le joueur perde même si son lieu de destination est un noeud de victoire.



## ii Interface Player / Fourmis

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

Ci-dessous, toutes les méthodes communes sont présentés de façon général. Mais elles seront toutes expliquer dans les subsection fourmis [iv](#) et player [iii](#), vu un peu plus bas.

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

*combatChoice*, prend en paramètre le noeud de Combat et le nombre de tour avant l'évasion ainsi que le BookState. Elle permet de faire un choix lors du tour du joueur dans un combat. On peut alors choisir d'attaquer, d'aller dans l'inventaire ou alors de s'évader. Si on choisi l'inventaire, on va alors dans une autre méthode appelé *useInventaire* qui prend le BookState en paramètre. On peut alors utiliser un item de santé, 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. Cela donne donc libre choix au joueur sur le nombre d'objet à utiliser (comme par exemple, l'ajout d'une arme ainsi qu'un bouclier).

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

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

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

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

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

## iii Player

La classe **Player** permet de jouer au jeu en tant que player. Elle permet de faire des choix grâce aux Scanner permettant ainsi l'avancement du jeu. Des messages sont aussi affiché afin de guider le player dans ses choix.

Nottament la méthode *choixYesNo()* qui permet de choisir Oui ou Non et de renvoyer le boolean true ou false. Cette méthode permet, par exemple, de savoir si le player veut supprimer, prendre un item ou une compétence. Elle a donc été créé évitant ainsi la redondance de code.



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

```

1 public void prendreItems(BookState state, List<BookItemLink> bookItemLinks, int
  nbItemMax){
2     while(nbItemMax != 0 && !bookItemLinks.isEmpty()){
3         System.out.println("Les items suivant sont disponible:");
4         //Affiche les items
5
6         System.out.println("Voulez vous un item ?");
7         if(choixYesNo()){
8             int itemMax = state.getMainCharacter().getItemMax();
9
10            if(itemMax == state.getMainCharacter().getItem().size()){
11                itemPlein(state);
12
13                System.out.println("Voici vos choix:");
14
15                if(choixYesNo())
16                    nbItemMax = 0;
17                else
18                    itemSupp(state);
19            } else {
20                itemAdd(state, bookItemLinks);
21            }
22
23        } else {
24            nbItemMax = 0;
25        }
26
27        if(bookItemLinks.isEmpty())
28            nbItemMax = 0;
29    }
30 }

```

**Listing 5.18** – *prendreItems()* du Player

Par exemple, *itemAdd()* (cf ligne 20 dans le code 5.18) qui elle permet de choisir l'item à ajouter dans l'inventaire. Une variable "choixValide" permet de ne pas valider un numéro de choix non valide. Une fois l'item choisi, l'ID de l'item est alors ajouter dans la liste de l'inventaire afin de pouvoir retrouver l'item si le joueur l'utilise.

Cette méthode a été créé permettant de séparer le code de la méthode *prendreItems()* pour une meilleure visibilité. Elle pourrait aussi servir, par exemple, dans l'ajout d'un item obligatoire. Dans ce cas, la méthode *prendreItems()* n'est plus utilisé mais directement *itemAdd()*. Mais cette exemple n'est pas représentatif de l'état actuel de MagicBook qui ne permet pas de rendre un item obligatoire. Mais c'est une idée d'amélioration.

```

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

```

```

15     }
16
17     BookItemLink itemLink = bookItemLinks.get(choix);
18     System.out.println("L'item "+state.getBook().getItems().get(itemLink.getId())
19     ).getName()+" a été rajouté");
20     state.getMainCharacter().getItems().add(itemLink.getId());
21
22     itemLink.setAmount(itemLink.getAmount()-1);
23
24     if(itemLink.getAmount() == 0)
25         bookItemLinks.remove(itemLink);
26 }

```

Listing 5.19 – itemAdd()

Puis la méthode *itemPlein()* (cf ligne 11 dans le code 5.18), qui elle, affiche les items à supprimer. Cette méthode permet d’afficher les items de l’inventaire.

```

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

```

Listing 5.20 – itemPlein()

Pour la méthode *itemSupp*, elle est appelé (cf ligne 18 dans le code 5.18) afin de choisir l’item à supprimer. Une boucle while est alors parcouru afin d’être sûr d’avoir un choix valide. Une fois l’item choisi, cela met l’inventaire du player à jour grâce au state pris en charge dans la méthode *prendreItems*. Bien sûr, l’utilisateur à le choix de ne pas supprimer un item permettant ainsi de garder tout les items de son inventaire. On aurait pu très bien laisser le choix au player de supprimer quand même un item si son inventaire n’était pas plein. Mais c’est un livre à choix avec des prérequis, tout les items peuvent donc être important, et il n’y a pas de pénalité de déplacement si l’inventaire est plein. Cette fonctionnalité de supprimer un item dans l’inventaire n’a alors pas été jugé nécessaire à ajouter en option libre.

```

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

```

Listing 5.21 – itemSupp()

Pour la méthode *itemAdd()*, elle permet d'ajouter un item choisi. Tout ceci est donc géré par la méthode *prendreItemsPlayer*, permettant ainsi d'appeler certaine méthode en fonction des choix du player ainsi que de gérer la prise des items grâce à la boucle while.

Pour la méthode *execPlayerCreation()* est presque la même que celle de la fourmis : elle utilise aussi la méthode *prendreItems()*, défini ici 5.18 au moment de l'ajout des items. Mais au moment de l'ajout de compétences, la méthode *skillAdd()* est appelé jusqu'à ce que le maximum de compétences autorisé a été pris ou qu'il n'en reste plus à prendre. Cette méthode ressemble fortement à la prise d'item. *execPlayerCreation()* sera à implémenté dans la classe *Jeu*, car beaucoup de code est en commun.

```

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

```

Listing 5.22 – execPlayerCreation() du Player

#### iv Fourmis

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

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

```

1 public ChoixCombat combatChoice(BookNodeCombat bookNodeCombat, int
   remainingRoundBeforeEvasion, BookState state) {
2     boolean choixValide = false;
3     Random random = new Random();
4     ChoixCombat choixCombat = null;
5     int choix;
6

```

```

7   while(!choixValide){
8       choix = random.nextInt(ChoixCombat.values().length);
9       choixCombat = ChoixCombat.values()[choix];
10
11       if (choixCombat == ChoixCombat.INVENTAIRE){
12           if(!state.getMainCharacter().getItems().isEmpty())
13               useInventaire(state);
14       } else {
15           choixValide = true;
16       }
17   }
18
19   return choixCombat;
20 }

```

Listing 5.23 – combatChoice() de Fourmis

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

```

1 public void useInventaire(BookState state){
2     List<String> listItemState = state.getMainCharacter().getItems();
3
4     Random random = new Random();
5     int choix = random.nextInt(listItemState.size());
6     BookItem bookItem = state.getBook().getItems().get(listItemState.get(choix))
7     ;
8
9     if(bookItem instanceof BookItemDefense){
10         state.setBookItemDefense((BookItemDefense) bookItem);
11     }
12     else if(bookItem instanceof BookItemHealing){
13         BookItemHealing bookItemHealing = (BookItemHealing) bookItem;
14         state.getMainCharacter().heal(bookItemHealing.getHp());
15         state.getMainCharacter().getItems().remove(listItemState.get(choix));
16     }
17     else if(bookItem instanceof BookItemWeapon){
18         state.setBookItemArme((BookItemWeapon) bookItem);
19     }
20 }

```

Listing 5.24 – useInventaire() de Fourmis

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

```

1 public void prendreItems(BookState state, List<BookItemLink> bookItemLinks, int
  nbItemMax){
2     int choix = 0;
3
4     while(nbItemMax != 0){
5         int itemMax = state.getMainCharacter().getItemsMax();
6         //Vérifie item maximum et si il y a encore des items à prendre
7         if(state.getMainCharacter().getItems().size() < itemMax && !
  bookItemLinks.isEmpty()){
8             //Choix
9             Random random = new Random();
10            choix = random.nextInt(bookItemLinks.size());
11        }
12    }
13 }

```

```

12      BookItemLink itemLink = bookItemLinks.get(choix);
13
14      state.getMainCharacter().addItem(itemLink.getId());
15      itemLink.setAmount(itemLink.getAmount()-1);
16
17      if(itemLink.getAmount() == 0)
18          bookItemLinks.remove(itemLink);
19
20      if(nbItemMax != -1)
21          nbItemMax--;
22  } else {
23      nbItemMax = 0;
24  }
25 }
26 }

```

Listing 5.25 – prendreItems() de Fourmis

Des boucles infini sont donc à prévoir si par exemple, un objet est requis pour accéder au choix 2 et qu'il faut aller dans le choix 1 pour prendre l'item. Vu qu'au moins un choix est disponible, aucun noeud terminal n'est généré. Sauf que si l'item de la fourmis est plein, l'item ne peut pas être pris et aucun item est supprimé. Si la suppression est active, quel est l'item à supprimer? Comment définir un item important? A la santé du personnage, au nombre d'objet non possédé d'un certain type (exemple : defense)? A un nombre "d'importance" prédéfini à l'édition du livre? Cette piste de réflexion serait donc à exploiter dans les améliorations à apporter.

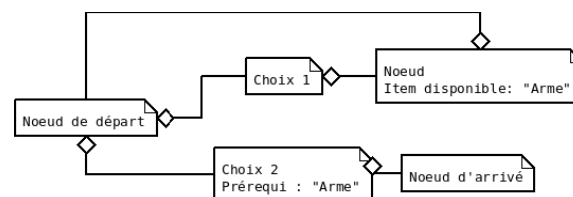


FIGURE 5.22 – prendreItem exemple de boucle infini

[H]

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

```

1 public void execPlayerCreation(Book book, AbstractCharacterCreation
    characterCreation, BookState state) {
2
3     if(characterCreation instanceof CharacterCreationItem){
4         CharacterCreationItem characterCreationItem = (CharacterCreationItem)
        characterCreation;
5
6         prendreItems(state, characterCreationItem.getItemLinks(),
        characterCreationItem.getAmountToPick());
7     }
8     else if(characterCreation instanceof CharacterCreationSkill){
9         CharacterCreationSkill characterCreationSkill = (CharacterCreationSkill)
        characterCreation;

```

```

10
11     Random random = new Random();
12     int amountToPick = characterCreationSkill.getAmountToPick();
13     while(amountToPick != 0 && !characterCreationSkill.getSkillLinks().
isEmpty()) {
14         int choix = random.nextInt(characterCreationSkill.getSkillLinks().
size());
15         state.getMainCharacter().addSkill(characterCreationSkill.
getSkillLinks().get(choix));
16         characterCreationSkill.getSkillLinks().remove(choix);
17         amountToPick--;
18     }
19 }
20 }

```

**Listing 5.26** – `execPlayerCreation()` de Fourmis

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

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

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

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

## 6 Conclusion

Ce projet a permis aux différents membres de s'améliorer, que ce soit dans l'apprentissage d'une librairie, la mise en application des concepts vus en cours ou bien encore, dans l'art de la procrastination. Bien que nous ayons été quatre sur le projet, deux de nos camarades n'ont pas fourni 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 restons cependant satisfaite de notre travail, l'application étant tout de même fonctionnelle et plutôt complète.