Magic Book

Rapport de projet

DEROUIN Auréline 21806986
MARTIN Justine 21909920

Table des matières

1 Présentation du projet

A Présentation de l'application

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

2 Organisation du projet

A Choix des technologies

i Git

Nous avon fais le choix d'utiliser Git comme logiciel de gestion de versions. Les raisons de ce choix sont listées ci-dessous :

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

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

Branches

Afin d'utiliser au mieux Git, nous avons fait le choix de créer deux branches "principales". Il s'agit de *master* et de *develop*.
La branche *master* correspond à une version stable qui peut être mise en production. Ainsi, on ne travaillera jamais sur cette branche. Elle ne nous servira donc qu'à récupérer l'application dans un état stable afin d'y mettre les différentes applications en production.
La branche *develop*, quant à elle, est donc la branche à partir de laquelle nous travaillons. C'est à partir de cette dernière que 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 donc les branches correspondant aux fonctionnalités développées, elles commencent toutes par features/XXX (correspondant à la modification). Par exemple, pour le développement des fourmis, on créera une branche features/fourmis.

Nomenclature

Nous avons choisit d'établir et d'utiliser une nomenclature pour les messages de commit. Chaque message est préficé 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.
Ce qui donne des messages comme celui-ci : *feat : Rend le GraphPane scrollable*.

ii Gradle

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

iii JavaFx

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

B Gestion du projet

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

i GitHub et Forge

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

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

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

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

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

ii Trello

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

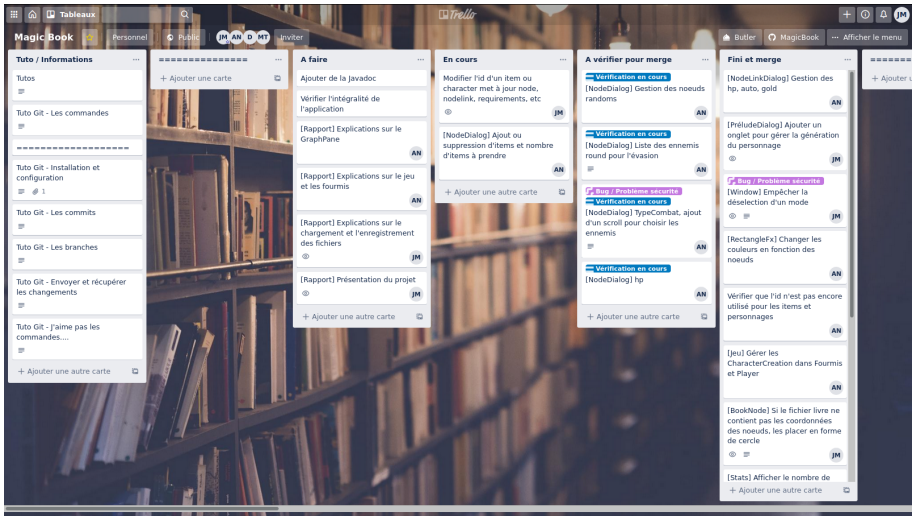

FIGURE 2.1 – Notre tableau Trello

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

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

iii Discord

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

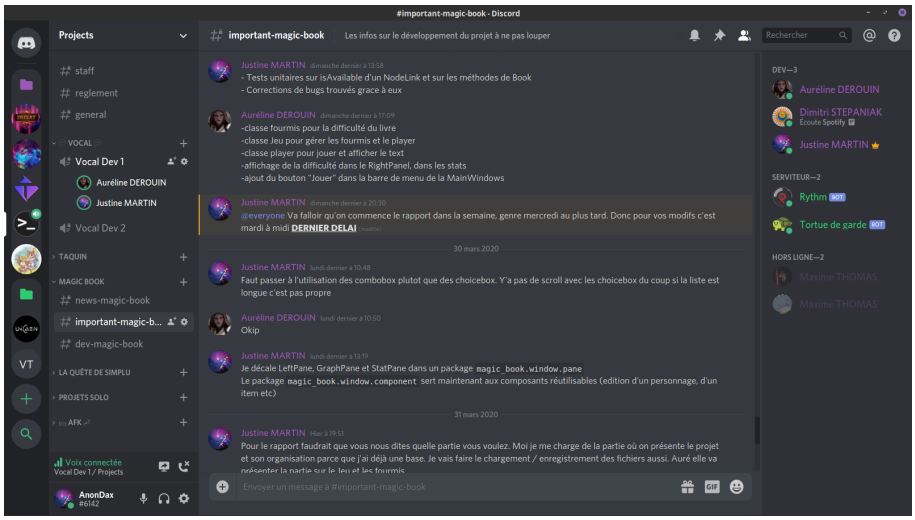

FIGURE 2.2 – Notre serveur Discord

3 Travail de groupe

Tâches effectués	Auréline	Dimitri	Justine	Maxime
Lecture et enregistrement des fichiers				
Classes pour parser le JSON			X	X
Lecture d'un fichier JSON			X	
Enregistrement d'un fichier JSON			X	
Livre				
Classe Book			X	
Classes pour représenter les noeuds et les liens	X		X	
Classe BookCharacter			X	X
Classes pour les Requirement	X		X	
Classes pour représenter les différents types d'items			X	
Classes pour la "Création du personnage"			X	
Classe pour représenter des skills			X	
Ajout des classes pour le pattern observer (uniquement celles qui concernent le livre)			X	X
Jeu et export au format texte				
Classe Jeu, partie commune au joueur et la fourmis	X			
Classe pour la logique de la fourmis	X			
Classe pour la logique du joueur	X			
Permettre une estimation de la difficulté du livre	X			
Generation du livre en format texte			X	
Classe BookState			X	
Conception d'un Parser pour le texte des liens et des paragraphes			X	
Version primitive de l'estimation de la difficulté d'un livre				X
Fenêtre				
Fenêtre principale		X	X	
Permettre la conception d'un nouveau livre, l'ouverture d'un ancien livre sauvegarder, la sauvegarde et "sauvegarde sous" du livre courant			X	
Lister et permettre l'ajout d'items et de personnages sur le panel de gauche		X		
Permettre d'editer ou supprimer un item ou un personnage du livre			X	
Statistiques concernant les noeuds			X	
Statistique sur le niveau de difficulté du livre	X			
Cacher panel des statistiques si l'on décoche une case dans le menu			X	
Cache le panel de gauche si l'on décoche une case dans le menu				X
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, une phase de la "Création d'un personnage", sélectionner une liste d'items			X	
Boites de dialogues				
Classe mère pour les boites de dialogue	X			
Boite de dialogue pour les noeuds	X			
Boite de dialogue pour les liens entre les noeuds	X			
Boite de dialogue pour les items	X			
Boite de dialogue pour les personnages			X	
Boite de dialogue pour le prélude			X	
Boite de dialogue pour la "Création du personnage"			X	
Boite de dialogue pour le personnage par défaut			X	
Zone d'édition				
Classe pour représenter un noeud graphique	X			
Ajout d'un noeud	X			
Modification d'un noeud	X			
Suppression d'un noeud	X			
Classe pour représenter un lien entre 2 noeuds			X	
Ajout d'un lien entre 2 noeuds (NodeLinkFx)			X	
Un lien suit les noeuds auxquelles il est attaché			X	
Modification d'un lien entre 2 noeuds	X			
Suppression d'un lien entre 2 noeuds	X			
Classe mère commune pour représenter un prélude et un noeud (RectangleFx)			X	
Permettre le déplacement des noeuds	X			
Détecter un clique sur un noeud ou lien (classes observer)	X		X	
Gestion des actions en fonction du mode	X			
Afficher un rectangle qui représentera le prélude			X	
Gestion du texte de prélude			X	
Gestion du personnage par défaut			X	
Gestion de la "Conception du personnage"			X	
Changer le premier noeud du livre			X	
Répartition des différents noeuds lors de l'ouverture d'un fichier			X	
Gestion du niveau de zoom			X	
Rend le GraphPane scrollable			X	
Change la couleur d'un noeud en fonction de son type (normal, aléatoire, combat, victoire, ...)	X			
Mettre en valeur un noeud lorsque l'on passe la souris dessus	X			
Autre				
Rapport	X		X	
Restructuration du livre fournis pour les tests (fotw.json)			X	
Création de tests unitaires	X		X	
Javadoc	X			
Revue de code avant de merge			X	

A Idées d'améliorations

Notre application n'ayant pu être terminé faute de temps, voici la liste des améliorations que nous aurions voulus 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 transfert 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
- Gérer les shops(jeu et gui), champs auto (jeu uniquement) et skills (gui)
- Afficher les personnages et items inutilisés
- Indiquer si l'estimation de la difficulté est à jour ou non
- Gestion des prérequis sur les boites de dialogue des noeuds
- Améliorer l'intelligence de la fourmis (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 (plutot que d'avoir un simple nom)
- Afficher les chemins gagnants
- "Langage" simple permettant de manier des conditions et variables pour des prérequis notamment
- Possibilité d'avoir des pnj qui pourraient nous suivre dans l'aventure pour combattre ou pour dévérouiller certains passages par exemple.

B Bugs et problèmes connus

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

- Tests incomplets sur le Book et le Jeu
- Le changement d'id d'un personnage ou d'un item ne met pas à jour les différents élements du livre (noeuds, choix, ...)
- Diverses bugs visuels concernant la boite de dialogue sur le Prélude
- Le zoom ne se fait pas selon la position actuel de la souris mais du point supérieur gauche du GraphPane

4 Architecture du projet

A Arborescence du projet

B Présentation des packages

5 Aspects techniques

A Représentation d'un livre

i Représentation des noeuds et des liens

ii Quelques algorithmes

B Lecture et écriture d'un livre

C Edition d'un livre

D Rendre le livre jouable et estimer sa difficulté

i Jeu

Une classe à été créer se nommant **Jeu**, permettant de gérer les méthodes de jeu communes entre le *Player* et les *Fourmis*.
Un construteur est d'abord appelé, à partir de la MainWindows, afin d'envoyer le livre contenant toutes les informations. Puis, en fonction du mode sélectionner ("Générer la difficulté" ou "jouer"), la méthode correspondante au player est appelé.
Une fois dans la méthode choisis, le livre est alors copié afin de ne pas le modifier par erreur pendant le jeu. Un BookState, qui correspond à la sauvegarde de la partie, est alors créé. Si le prélude contient un personnage principal, alors celui ci est enregistré dans la sauvegarde de la partie (le BookState). Dans le cas contraire, un autre personnage principal est créer afin de pouvoir jouer au jeu.
Enfin, si des compétences et/ou des items sont disponible au début de la partie, la méthode de création de joueur est appelé en focntion du player actuel. Comme cela, le player choisit parmi une liste de skill et d'items disponible en début de partie afin de les avoir pour commencer le jeu.
Une fois le BookState créer, le personnage principal initialisé et la copie du livre enregistré, le premier noeud est donc chargé. Une méthode est appelé en fonction de son type de noeud. La méthode correspondante au type de noeud s'exécute et renvoie le noeud de "destination", en fonction du choix du player et/ou de la mort du player. Durant l'exécution des différentes méthodes et en fonction du player, d'autre méthode externe sont appelé nottament dans la classe Fourmis ou Player.
Pour chaque type de noeud, sauf pour un noeud terminal, une méthode commune est appelé afin de savoir si le noeud pris en charge fait gagné/perdre de la vie puis regarde si le player est toujours en vie. Si ce dernier n'est plus en vie, un noeud terminal est alors renvoyé en noeud de destination. S'il est encore en vie et que le noeud propose des items, ils sont proposés au player en appelant la méthode correspondante entre Fourmis ou Player. A chaque detination choisi, une autre méthode est appelé afin de regarder si le lien entre le noeud de départ et de destination fait perdre ou gagner de la vie et/ou de l'argent.

Si un noeud est de type basic, il est alors pris en charge dans la méthode *execNodeWithChoices*. Cette

dernière renvoi un noeud terminal si aucun choix n'est valide, ce qu'il veut dire, si le player n'a aucun choix ou s'il ne possède pas les items/skill pour aller vers ce choix.
Si le player est encore en vie et si il peut au moins choisir une destination, un choix est demandé parmis toutes les destinations faisant un appel à la méthode en fonction du player. Si le player a les prérequis pour aller vers cette destination, alors le noeud choisi est renvoyé en noeud de destination. Sinon, le player doit faire un autre choix.

Si c'est un noeud de type combat, une vérification est réalisé afin savoir si le noeud contient des ennemis. S'il n'y a pas d'ennemis, le noeud en cas de victoire est envoyé en noeud de destination. Sinon, une liste d'ennemis est créer afin de ne pas modifié la vie des ennemis. Car ces derniers ne sont pas lié au noeud, mais c'est l'ID de l'ennemis qui est lié au noeud permettant de les appelés plusieurs fois dans plusieurs ou dans le même noeud.
Le combat commence alors. Le choix est défini par la méthode du player correspondant. Trois choix sont possible :

- Attaque : un autre choix est demandé permettant de sélectionner l'ennemi à attaquer parmi la liste des ennemis encore en vie. Une fois l'ennemi sélectionné, une méthode attaque est appelé apellant elle même une autre méthode commune entre l'attaque du player et l'attaque d'ennemi. Nommé getDamageAmount, elle permet de savoir le nombre de dommage réalisé en fonction des point d'attaque de l'attaquant, de son double dommage décider en random si ce boolean est défini en true, d'un coup critique décider aussi en random, de l'arme de l'attaquant et de l'item de défense de l'attaqué. L'attaquant et l'attaqué est défini en fonction de la permière méthode qui l'appel. Ici c'est la méthode d'attaque du player.
  Une fois l'attaque effectué, si l'ennemi attaqué est mort, il est supprimer de la liste des ennemis.
- Inventaire : si ce choix est fait par le player, la méthode appelé permettant d'utiliser son inventaire est elle même gérer dans la classe Player ou la classe Fourmis. Elle permet alors de choisir une potion, une arme et ou un item de défense.
- Evasion : si le tour avant évasion est inférieur ou égal à 0 et si un noeud de d'évasion existe, le player peut alors s'enfuir. Sinon cela lui passe son tour.

Une fois le tour du joueur fini, vient le tour de l'ennemi. Il appel une méthode envoyant la liste d'ennemis restant. Cette méthode appel getDamageAmount permettant aux ennemis d'attaquer un par un.

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

Si le noeud est de type aléatoire, la méthode commune est appelé afin de savoir si le player est encore en vie. Puis une autre méthode est appelé afin de déterminer le noeud de destination en fonction des chances attribué à chacun de ces choix.

Si le noeud est de type terminal, la partie est alors terminé et renvoie un bolean sur l'état de la fin de partie.

ii Interface Player / Foumis

Une interface **InterfacePlayerFoumis** à été créer permettant une mise en commun des codes Player et Fourmis. Ces méthodes permettent de faire un choix, prendre les items disponibles, créer un perso-

nage lambda, aller dans l'inventaire, choisir son ennemis ou encore combatre. Elles sont appelé au même moment. La méthode sera alors exécuté différément en fonction du player.

execPlayerCreation permet de choisir les skill et les items disponible au début de la partie. Ces derniers sont défini lors de la création du prélude. Pour l'ajout des items, la méthode *prendItems* est appelé.

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

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

prendItems permet de prendre un item parmi la liste d'items disponible. Cette liste est pris en paramètre ainsi que la sauvegarde de la partie et le nombre d'item maximum pouvant être pris.

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

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

iii Player

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

iv Fourmis

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

6 Conclusions

A Éléments à améliorer

B Avis personnels

7 Ressources utiles et sources utilisés