

World of Heroes
Projet logiciel de Licence Informatique

Erik Clairiot, Ronan Le Fayer

6 mai 2007

Table des matières

1 Règles du jeu	3
1.1 Objectifs	3
1.2 Les personnages	3
1.3 Objets	3
1.4 Le monde	3
2 Comment jouer ?	5
2.1 Pour commencer	5
2.2 Choisir la langue du jeu	5
2.3 Charger un nouveau monde	5
2.4 Quitter le jeu	5
2.5 Génération d'un monde par fichier XML	5
2.5.1 Ajouter un type de terrain	6
2.5.2 Ajouter un objet	6
2.5.3 Ajouter votre héros	7
2.5.4 Ajouter un personnage	8
3 API	10
3.1 UML	10
3.1.1 Paquetage Living	10
3.1.2 Paquetage Item	11
3.1.3 Paquetage Action	12
3.1.4 Paquetage Map	13
3.1.5 Paquetage Choice	13
3.1.6 Paquetage World	13
3.1.7 Paquetage Exception	14
3.1.8 Paquetage Misc	14
3.2 Utilisation des Properties	14
3.3 Mise à jour de l'API	15
3.3.1 Créer un nouveau terrain	15
3.3.2 Créer un nouvel objet	16
3.3.3 Créer un nouveau personnage	17
3.3.4 Créer une nouvelle action	18
3.3.5 Créer un nouveau monde	21
3.4 Originalité du projet ?	22
4 Conclusion	22

1 Règles du jeu

1.1 Objectifs

Ce jeu est un jeu d'aventure. Et vous en êtes le héros. Cool! Vous vivez dans un monde plein de personnages, qui n'ont pour but que le vôtre. Grâce à un menu, vous contrôlerez aisément votre héros. Suivant la situation dans laquelle vous vous trouverez, vous aurez le choix entre plusieurs actions. Nous vous donnerons l'opportunité de battre vos voisins, d'utiliser les objets contenus dans votre besace, de collecter les objets honteusement jetés au sol ou encore d'interagir avec d'autres personnages... Par votre intelligence, mais aussi et surtout votre fourberie, vous pourrez essayer de sauver une princesse, qui hélas à déjà rendu l'âme.

Comprenez bien qu'il n'y a que deux moyens de finir ce jeu : mourir ou survivre.

1.2 Les personnages

Tous les personnages ont les mêmes caractéristiques, à savoir de l'énergie et de la puissance. L'énergie correspond à la santé du personnage. La puissance est ce que votre héros perd lorsqu'il est attaqué par un autre personnage.

Quand un personnage voit son énergie tomber à 0, nous pensons que nous pouvons considérer d'un commun accord que ce personnage est décédé.

Chaque personnage est doté d'un inventaire et se verra attribuer un peu d'or. Quand un personnage meurt, il lache tous ces biens sur la case. Voilà une magnifique occasion de tout ramasser, le fourbe que vous êtes ne nous dira pas le contraire.

Plusieurs actions différentes peuvent être attribuées aux personnages. Les actions originellement disponibles sont : se battre ou interagir avec un autre personnage, collecter ou utiliser des objets, bouger ou ne pas bouger...

Concernant les personnages contrôlés par l'ordinateur, leurs actions sont choisies aléatoirement. Vous, par contre, devrez choisir ce que vous voulez faire grâce à un menu d'une sobriété classique.

Excepté vous qui avez l'immense chance et l'unique privilège d'être un Héros, ce jeu contient (uniquement) deux autres sortes de personnage. Faisons leur connaissance sans plus attendre!

Le Guerrier : Un monstre énorme muni d'une machette, d'un couteau et d'une épée légendaire gigantesque. Il n'a pas de pistolet mais un bazooka (car les guerriers n'aime pas les balles ni le lait). Il n'a pas de cerveau mais uniquement un réseau de nerfs primitifs qui lui permet bien heureusement de s'adonner au plaisir de se battre avec tout ce qui bouge... ou pas. Il est doté de pieds, qui lui permettent de marcher... ou pas.

Le Guérisseur : Un petit gnome, vraiment petit, avec une énorme moustache rose et un collier de hippie autour du coup. Il est doté de pieds comme le guerrier, mais les pas d'un gnome étant ridiculeusement court, et la marche n'étant pas ce qui soit le plus rentable d'un point de vue pécunier, il ne risquera jamais de se fatiguer en marchant. Malgré cette fénéantise, il n'hésitera jamais à vous soigner. Si vous avez de l'argent. Et si vous lui demandez. Plutôt égoïste ce gnome.

1.3 Objets

Dans la version originale nous avons fait deux types (seulement) d'objets.

La nourriture : C'est de la nourriture. Hmmm, c'est encore chaud. Quand un personnage en mange, il récupère de l'énergie.

L'argent : C'est de l'argent. Quand un personnage ramasse de l'argent, il gagne de l'argent. Evident.

Il y a des objets partout dans le monde, et le plus souvent juste à côté de vos pieds. Si, si, regardez!! Quand les objets sont au sol, vous pouvez les ramasser. Un par un, un par tour.

Mais il vous faut savoir que votre besace n'est pas sans fond. Elle ne peut pas contenir plus de dix-huit objets.

Quand vous utiliserez un objet en votre possession, cet objet disparaîtra de votre inventaire. Ne nous envoyez pas de courriels, nous ne vous avons pas volé. Et ce n'est pas non plus un bug, ceux-ci sont principalement cachés dans la gestion de votre argent... En résumé, un objet utilisé correspond à un espace qui se libère dans votre sacoche.

1.4 Le monde

Les mondes dans lesquels vous vivrez prochainement sont automatiquement générés grâce aux spécifications que vous aurez ajouté au fichier XML. Vous êtes donc libre de créer un monde rempli de paix et d'amour ou de vivre dans une ville où règne la loi du plus fort.

Vous voyagerez à travers le monde en choisissant à chaque tour une direction, si vous désirez vous déplacer bien entendu.

Mais vous devez savoir que se déplacer fait baisser l'énergie de votre héros, selon le type de terrain que vous traverserez. Alors n'oubliez pas de boire et de manger tout le long de votre voyage si vous voulez survivre.

Amusez vous bien !

2 Comment jouer ?

2.1 Pour commencer

Vous venez de récupérer l'archive du jeu que nous nommerons *game.tar.gz*. En premier lieu, décompressez l'archive dans un répertoire avec `tar -xzf game.tar.gz`. Voici l'arborescence à laquelle vous allez être confronté.

- *build/* Contient l'archive executable du jeu
- *doc/* Contient la javadoc du projet
- *lib/* Contient les bibliothèques externes au projet, ici uniquement *jdom.jar*
- *classes/* Contient les classes du projet
- *rapport/* Tout ce qui vous permettra d'en savoir plus sur l'api du jeu
- *xml/* Fichiers xml de configuration du jeu (packs de langues, generation de cartes)

Pour jouer, il vous faut la machine virtuelle java d'installée.

A partir de la racine du répertoire de décompression de l'archive, exécutez la commande suivante dans un terminal : `java -jar build/game.jar`

2.2 Choisir la langue du jeu

Ce jeu a la possibilité d'être joué en plusieurs langues. Chaque pack de langue est composé d'un seul fichier de propriétés, fichiers qui sont présents dans le répertoire *xml/lang*.

Actuellement, seules deux langues sont supportées : l'anglais et le français.

Pour changer de langue, il suffit de choisir cette option dans le menu principal. Un fichier de langue vous sera demandé, il suffit d'entrer le nom de fichier. A noter que le nom du répertoire est inutile, les fichiers étant automatiquement cherchés à partir de *xml/lang/*.

Exemple Liste des langues gérés avec leurs noms de fichiers :

Français *lang-fr.xml*
Anglais *lang-en.xml*

2.3 Charger un nouveau monde

Pour charger un nouveau monde, il faut passer par le chargement d'un fichier xml contenant les informations de ce monde. Tout comme les fichiers de langue, une option dans le menu principal vous permettra de charger un fichier xml. Les fichiers xml de génération de carte sont dans le répertoire *xml/world*, et donc encore, comme les fichiers de langues, il ne faut point mettre le dossier contenant ces fichiers.

Exemple Liste des mondes existants avec leurs noms de fichiers :

Carte classique *classic.xml*
Carte originale *original.xml*

2.4 Quitter le jeu

Si vous n'avez plus de pizza sous la main, si vous n'avez plus de schnaps ou pire, si l'envie de parler à des humanoïdes vous prend, et que vous êtes prêt à affronter un soleil meurtrier, la possibilité de quitter le jeu vous est proposé.

A une condition tout de même, que votre quête dans ce monde soit achevée. Une mort sanglante ou une victoire de fourbe importe peu. Finissez vos objectifs ! Sachez juste que nous pourrions vous empêcher d'arrêter jouer, mais nous comptons sur vous pour sauver le monde.

2.5 Génération d'un monde par fichier XML

Ce jeu ne possède aucun monde défini. Le seul moyen de visiter un monde est de le générer à l'aide d'un fichier XML. Cette section va vous présenter le type de document XML à produire pour créer le monde dont vous aviez toujours rêvé.

Si vous n'avez aucune idée de ce qu'est le XML, ne vous inquiétez pas. Nous espérons que la suite vous permettra de manipuler suffisamment correctement ces fichiers pour créer un fichier de génération. Si par contre vous êtes déjà habitué à ce langage, vous pouvez d'ores et déjà jeter un oeil sur le type de document XML à respecter en consultant la DTD (*xml/world/carteConfig.dtd*). Nous allons au fur et à mesure de cette partie afficher l'évolution du fichier XML que nous allons créer.

Tous les fichiers de génération de monde devraient être dans le dossier *xml/world*.

Une carte du monde est constituée d'informations concernant le type du monde et ses différents types de terrains, le héros,

les personnages susceptibles de rencontrer, les objets éparpillés dans le monde.

Le langage XML fonctionne sur un principe de balises ouvrantes et fermantes. Tout élément compris entre deux balises de même type peut-être perçu comme un sous-ensemble de l'ensemble défini par les balises parentes.

Une carte est donc prioritairement défini par les balises `<carte >` et `</carte >`. Toutes les informations futures seront donc entre ces deux balises.

```
1 <carte >
2
3 </carte >
```

A chaque carte est associé un type de monde. Par exemple, il peut s'agir d'un monde rectangulaire, ou chaque case possède un voisin au nord, sud, est ou ouest. Il pourrait aussi être un monde où vous pourriez vous déplacer, en plus de ces 4 directions, vers le haut ou vers le bas. Cela pourrait aussi être un monde où la téléportation serait possible, ou 2 cases lointaines pourraient être accessibles entre-elles par un portail.

Quoiqu'il en soit, il va falloir définir lors de la génération du monde le type de monde que vous voulez créer, tout en renseignant sur les informations dont il a besoin pour se créer correctement. Par exemple, un monde en 2 dimensions aura besoin de 2 dimensions pour définir correctement sa taille. Un monde 3D en aura besoin de 3 par exemple.

Nous ne saurons que vous conseiller de regarder la documentation Javadoc de l'application présente dans le dossier *doc* pour savoir précisément les informations requises par un monde pour être correctement créé.

Le nom de la classe du monde à charger est à mettre entre les balises de type `<classeCarte >`.

Chaque information nécessaire à la création du monde doit être insérée entre les balises de type `<param >`. Il est important de respecter l'ordre d'insertion de ces paramètres par rapport à ce qu'il est dit dans la Javadoc.

Par exemple, pour un monde en 2D (classe *map.lib.World2D*) qui va disposer de 10 cellules en largeur et de 20 cellules en longueur, voici ce que le fichier XML devient :

```
1 <carte >
2     <classeCarte>map.lib.World2D</classCarte >
3     <param>10</param >
4     <param>20</param >
5 </carte >
```

2.5.1 Ajouter un type de terrain

Chaque case de la carte est doté d'un paysage. Chaque paysage possède un cout d'énergie pour les personnages qui le traversent, ce qui est donc intéressant est de varier le type de paysage afin de faire réfléchir les personnages quant au trajet qu'ils vont adopter.

L'ensemble comprenant les informations sur les différents types de terrains est délimité par les balises `<terrains >` et `</terrains >`.

Chaque type de terrain a un certain pourcentage de chance d'être affecté à une case du monde. Ce pourcentage doit être inséré entre les balises `<pourcentage >` et `</pourcentage >`.

Le nom de la classe du terrain doit être quand à elle insérée entre les balises `<terrain >` et `</terrain >`, les classes de terrains étant dans le paquetage *map.lib*.

Ajouter un terrain consiste donc à ajouter le couple de pourcentage de chance d'apparaître et de la classe du terrain associé à ce pourcentage.

Exemple Voici un monde où il y a 30% de chance de voyager dans des montagnes (classe *map.lib.Mountain*) et 70% donc de rencontrer des plans d'eaux (classe *map.lib.Sea*).

```
1 <terrains >
2     <terrain>map.lib.Mountain</terrain >
3     <pourcentage >30</pourcentage >
4     <terrain>map.lib.Sea</terrain >
5     <pourcentage >70</pourcentage >
6 </terrains >
```

2.5.2 Ajouter un objet

Des objets sont éparpillés partout dans le monde. Lorsque un monde est généré, nous appliquons pour toutes les cases un certain pourcentage de chance qu'un objet soit posé sur une case ou non.

S'il est décidé qu'un objet soit posé sur une case, de nouveaux pourcentages entrent en jeu. En effet, chaque objet possède un certain pourcentage de chance d'être choisi parmi les autres objets du monde pour être posé sur la case.

Il apparaît clair que pour qu'un objet soit inséré dans un jeu, il doit uniquement posséder un certain pourcentage d'apparition. De plus, chaque monde possède un certain pourcentage de présence d'objet.

Dans le fichier XML, la partie concernant les objets est réunie dans le balisage `<choses >`. Ce qui signifie que tout ce qui est compris entre `<choses >` et `</choses >` concernent les objets, et que toutes les informations qui suivent à ajouter devront être ajoutées entre ces 2 balises.

Le pourcentage de présence d'objet doit être spécifié entre les balises `<probaChose >` et `</probaChose >`. Un entier compris entre 0 et 100.

Pour ajouter l'occurrence d'un objet dans un monde, il faut créer une sous-partie de `<choses >`. Cette sous-partie est définie par la balise `<chose >` et par sa balise fermante `</chose >`.

Dans la partie `<chose >`, 2 informations sont importantes : le nom de la classe de l'objet, et son taux de chance d'apparition par rapport aux autres objets.

Le nom de la classe objet doit être situé entre les balises `<classChose >`.

Le pourcentage d'apparition doit être situé lui entre les balises `<pourcentage >`.

Vous pouvez donc ajouter autant de type d'objets que vous le désirez, en mettant simplement à la suite toutes les parties `<chose > ... </chose >`. Remarquez tout de même qu'il va de soi que la somme des pourcentages d'apparition d'objets doit s'approcher de 100.

Exemple Un exemple de monde où les cases ont 100% de chance de posséder un objet, objet qui est choisi entre 2 objets possédant le même taux d'apparition de 50%.

```
1 <choses >
2     <probaChose >100</probaChose >
3     <chose >
4         <classChose >item.lib.CheatedItem</classChose >
5         <pourcentage >50</pourcentage >
6     </chose >
7     <chose >
8         <classChose >item.lib.Food</classChose >
9         <pourcentage >50</pourcentage >
10    </chose >
11 </choses >
```

Exemple Vous voulez ajouter un objet du nom de classe `FacticeItem`, avec un taux d'apparition de 2%. Notez qu'ajouter un objet vous oblige à modifier les taux d'apparitions des autres objets déjà renseignés. Voici le code à insérer dans la partie `<choses > ... </choses >`.

```
1 <chose >
2     <classChose >item.lib.FacticeItem</classChose >
3     <pourcentage >2</pourcentage >
4 </chose >
```

Maintenant que vous savez contrôler l'apparition d'un objet par case, vous allez savoir comment faire apparaître plus d'un objet par case. Il suffit pour cela d'ajouter autant de parties `<choses >` et `</choses >` que de nombre d'objets maximum possible par case.

2.5.3 Ajouter votre héros

Il est temps d'ajouter un héros à votre monde. A chaque monde un seul héros. Dans l'implémentation originale de ce jeu, un seul type d'héros existe, donc la question de choisir la classe ne se pose pas. L'ensemble XML réservé pour le héros est compris entre les balises `<heros >` et `</heros >`.

Cette partie concernant le héros sera très similaire à celle concernant les personnages, un héros étant un type de personnage. Nous développerons donc plus en détail la manière d'ajouter un héros, le procédé étant quasiment identique pour les personnages.

Pour charger un type de héros, il suffit de mettre le nom de la classe du héros entre les balises `<classHero >` et `</classHero >`.

Tout comme un monde, un personnage est doté de paramètres. Ces paramètres sont par exemple un nom, de l'énergie, de la force, de l'argent... Et tout comme les mondes, nous vous encourageons à lire la documentation afin de savoir précisément à quoi correspondent ces paramètres, et ce qui est particulièrement important, l'ordre d'insertion. Ces paramètres sont à insérer entre les balises `<param >` et `</param >`.

De plus, des actions enrichissent les personnages. Ainsi à chaque type de personnage, il est très facile de leur attribuer tout type d'actions. Actions qui sont par exemple "bouger", "attaquer", "ramasser un objet", "utiliser un objet"... Pour ajouter une action à un personnage, il suffit d'ajouter les balises `<capacite >` avec en leur sein le nom de la classe de l'action.

Exemple Voyons donc ce que cela donne si l'on désire avoir comme Héros un archimage (classe `living.lib.Archimage`), qui accepte 5 paramètres, qui sont respectivement son nom (Bobbafet), son énergie (7000), sa force (1), sa quantité de mana (8000) et sa pointure (32). Nous allons le doter des classes d'actions `actions.lib.UniversalPower` et `actions.lib.Sheep`. Voici le code XML à ajouter à notre précédent code (toujours entre `<carte >` et `</carte >`, à la suite des balises `<param >` concernant le monde) :

```
1 <heros >
2     <heroClass>living.lib.Archimage</heroClass >
3     <param>Bobbafet</param >
4     <param>1</param >
5     <param>7000</param >
6     <param>8000</param >
7     <param>32</param >
8     <capacite>actions.lib.UniversalPower</capacite >
9     <capacite>actions.lib.Sheep</capacite >
10 </heros >
```

2.5.4 Ajouter un personnage

Ajouter un personnage est très similaire à l'ajout du héros, nous vous encourageons donc à lire et relire la précédente partie jusqu'à ce que la compréhension vous arrive, cette partie étant moins développée que celle concernant le héros.

Dans le fichier XML, l'ensemble réservé pour les personnages est compris entre les balises `<personnages >` et `</personnages >`.

Dans un monde, des personnages évoluent. Lors de la création du monde, tout comme les objets, un pourcentage est appliqué sur chaque case de la carte pour décider ou non si un personnage y sera chargé. Ce pourcentage est contenu entre les balises `<probaPerso >` et `</probaPerso >`.

Chaque personnage est lui compris entre les balises `<personnage >` et `</personnage >`.

Si un personnage doit être chargé sur la case d'une carte, un nouveau tirage entre tous les différents types de personnages à charger est effectué. Ce pourcentage est compris entre les balises `<pourcentage >` et `</pourcentage >`.

Un personnage est composé de caractéristiques et de capacités. Les caractéristiques dépendent de la classe du personnage que vous désirez ajouter, et une fois de plus, n'hésitez pas à lire la Javadoc pour connaître ces caractéristiques. Les capacités sont indépendantes des classes de personnages, et sont contenues dans le paquetage `action.lib`.

Exemple On veut ajouter un guérisseur (classe `living.lib.Healer`) qui ait 1% de chance d'apparaître dans le monde, qui soit nommé Froissard, qui possède 3000 points de vie, une force de 100, un besace contenant initialement 100 pièces d'or, et un ratio de heal de 80 points d'énergie donné pour 10 pièces d'or payées. En action, ce guérisseur aura tout de même la capacité de se battre contre un ennemi (classe `actions.lib.Fight`) et pourra se déplacer dans le monde (classe `actions.lib.Move`).

```
1 <personnage >
2     <classPerso>living.lib.Healer</classPerso >
3     <pourcentage>1</pourcentage >
4     <param>Froissard</param >
5     <param>100</param >
6     <param>3000</param >
7     <param>100</param >
8     <param>80</param >
9     <capacite>actions.lib.Fight</capacite >
10    <capacite>actions.lib.Move</capacite >
```

Chaque ensemble compris entre les balises `<personnage >` et `</personnage >` va potentiellement faire apparaître des personnages de ce type dans le monde. Pour ajouter plusieurs types de personnages, il suffit d'insérer plusieurs ensembles de ce type, de les mettre à la suite, et tous évidemment dans le sous-ensemble réservé aux personnages. Ce qui est censé vous donner si vous désirez insérer deux types de personnages, avec une chance d'apparition d'un personnage sur une case de 60% :

```
1 <personnages >
2     <probaPerso>60</probaPerso >
3     <personnage >
4         ...
5     </personnage >
6     <personnage >
7         ...
8     </personnage >
9 </personnages >
```

Tout comme les objets, pour faire apparaître plusieurs personnages par cellule, il suffit d'ajouter plusieurs balisages `<personnages >` à la suite.

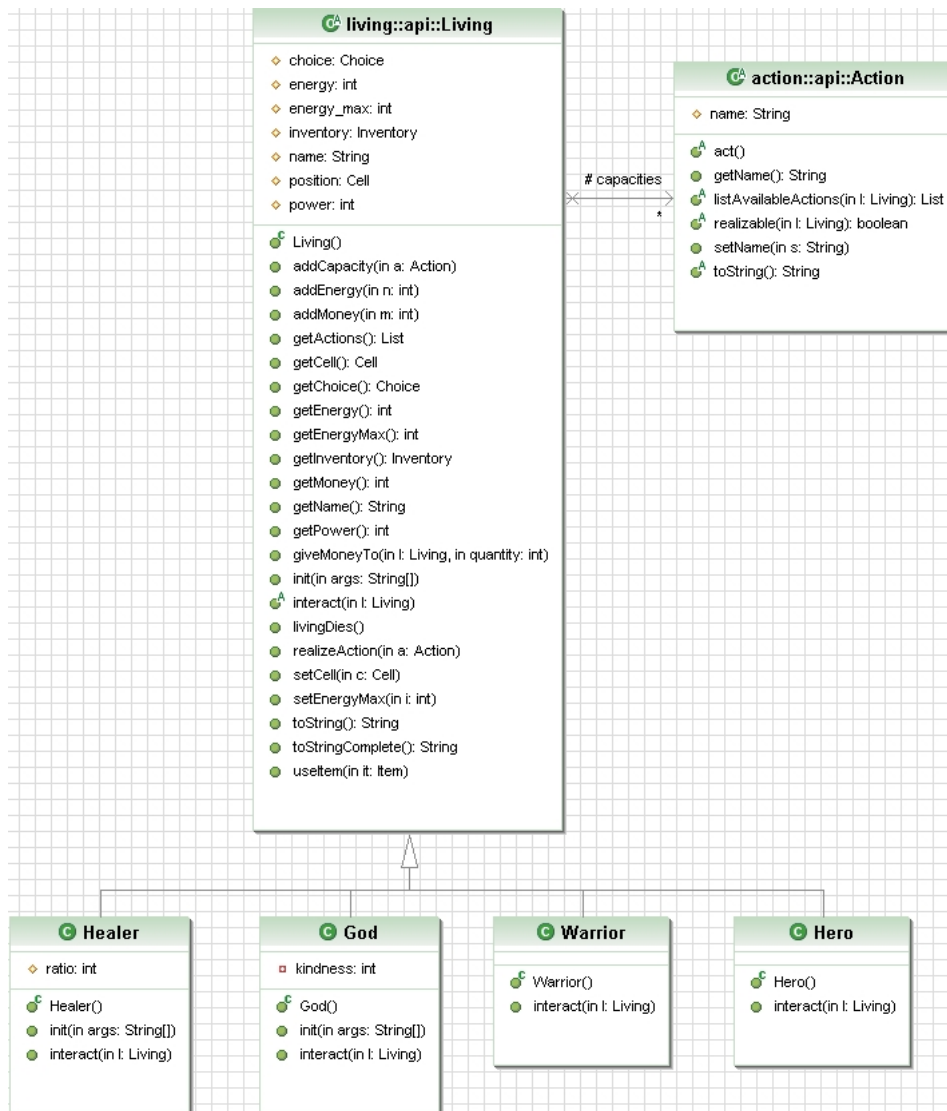
3 API

Cette partie servira aux programmeurs souhaitant modifier le jeu ou en comprendre son fonctionnement. L'application sera en premier lieu présentée sous la forme de diagrammes UML, puis nous expliquerons l'utilisation de paramètres externes notamment pour la gestion du multi-language. Et pour finir, des tutoriels vous présenteront succinctement comment mettre à jour le jeu.

3.1 UML

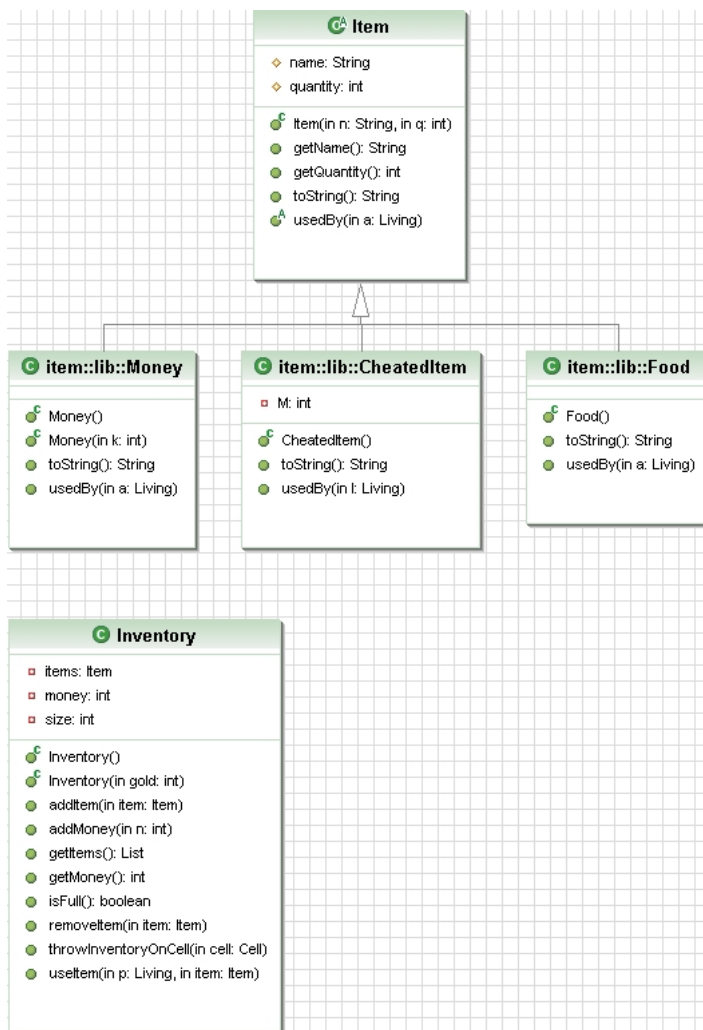
3.1.1 Paquetage Living

Définition d'un personnage et tous les types de personnages.



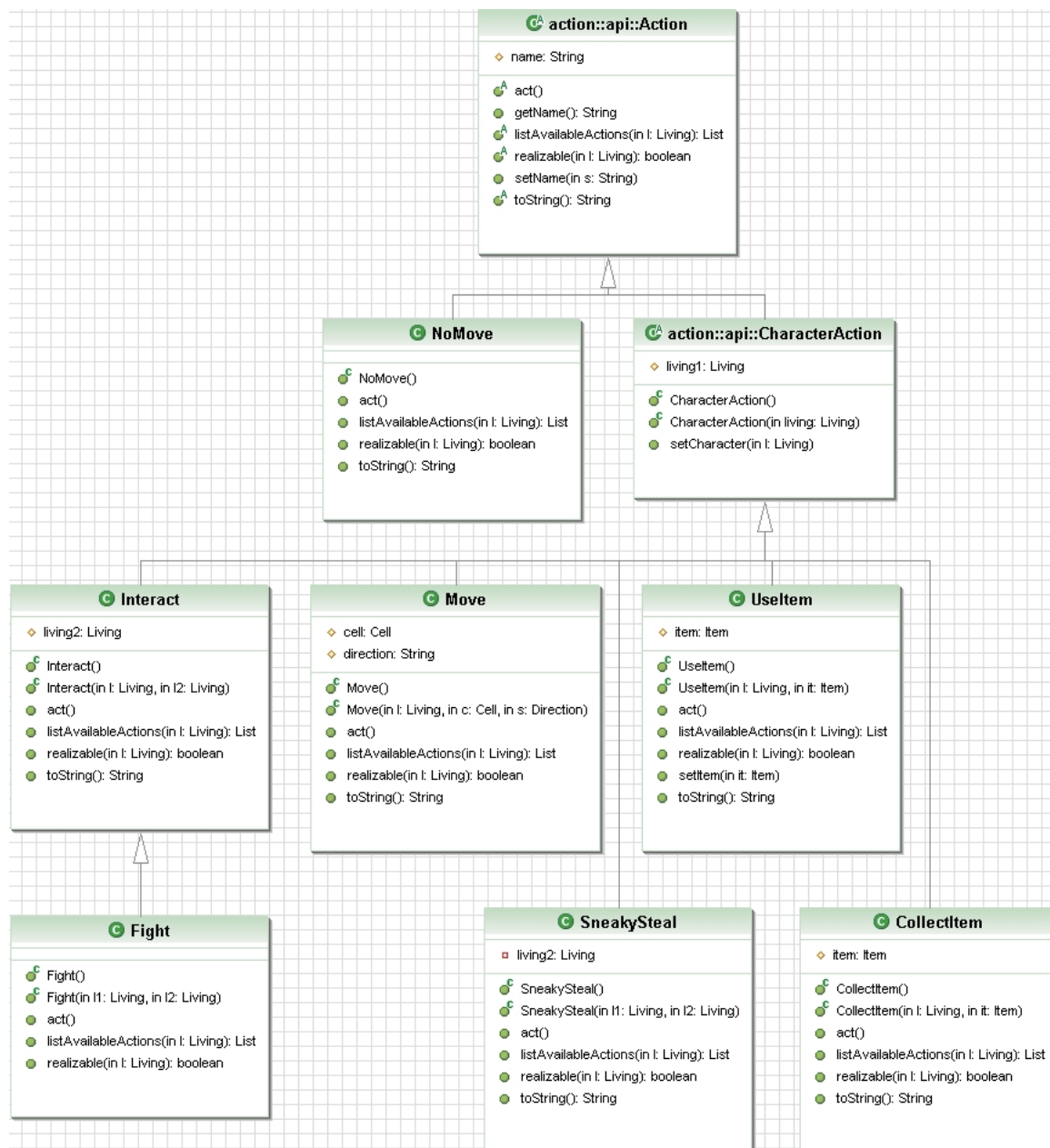
3.1.2 Paquetage Item

Définition des objets et de l'inventaire des personnages.



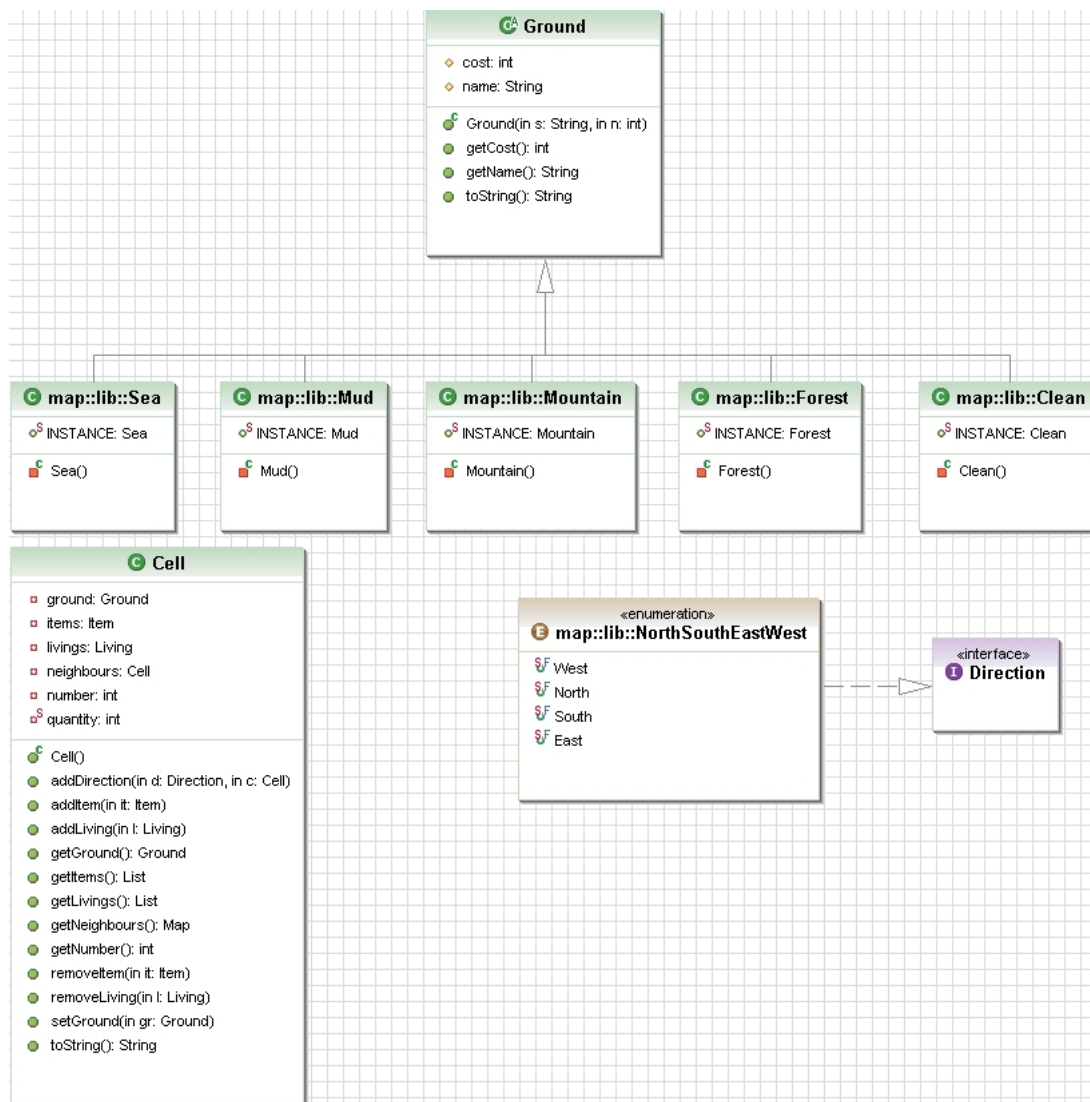
3.1.3 Paquetage Action

Actions réalisables par les personnages.



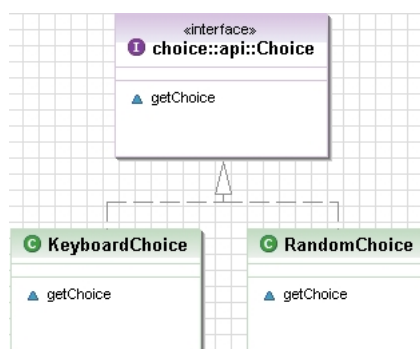
3.1.4 Paquetage Map

Regroupe toutes les classes utilisées dans un monde : cases, types de terrains, directions du voisinage.



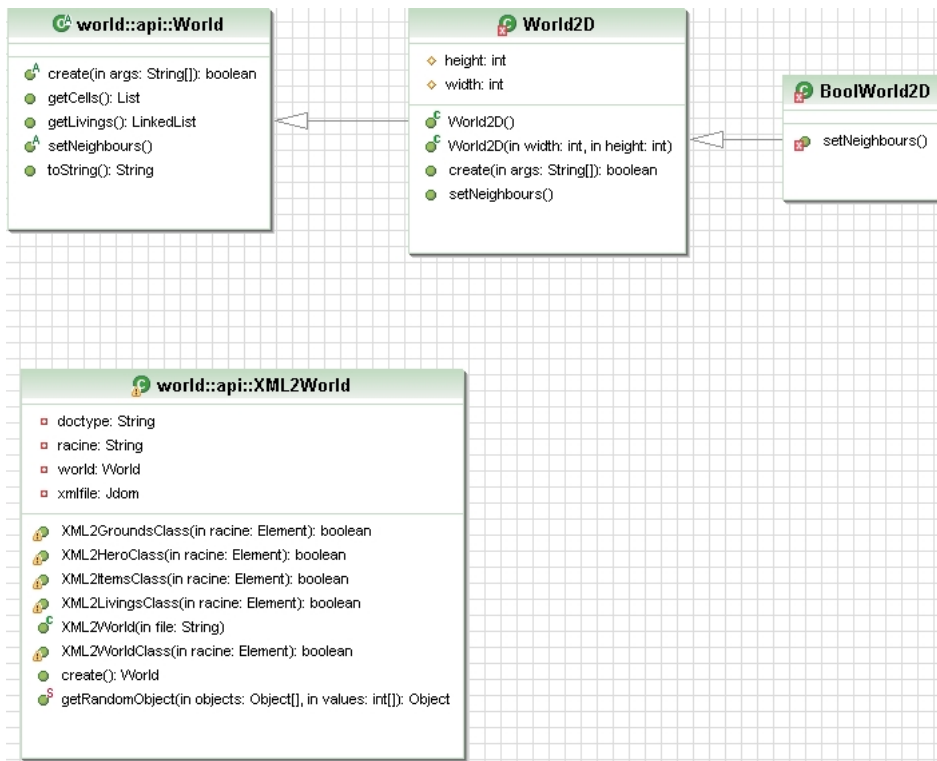
3.1.5 Paquetage Choice

Implémentation des différents types de choix possibles pour les actions des personnages, notamment le choix au clavier ou le choix aléatoire.



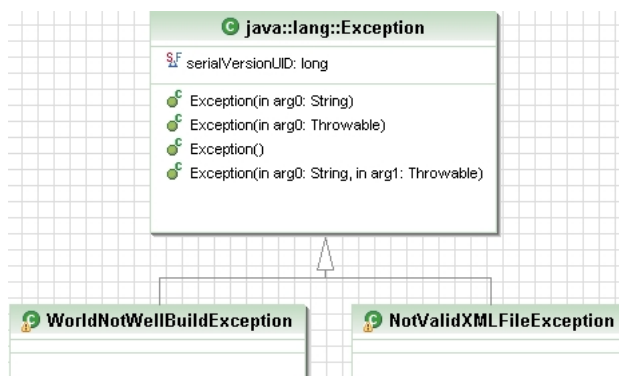
3.1.6 Paquetage World

Regroupe toutes les classes nécessaires à la création d'un monde, et les différentes sortes de mondes jouables.



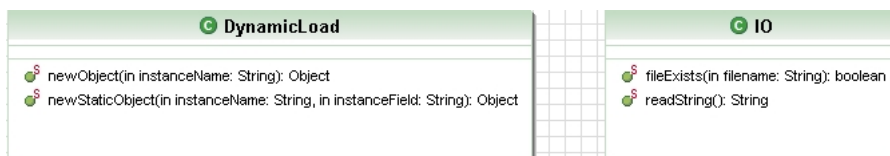
3.1.7 Paquetage Exception

Exceptions levables pendant l'exécution du jeu.



3.1.8 Paquetage Misc

Classes pratiques pour l'utilisation des classes utilisées et citées plus haut (chargement dynamique d'objets, gestion des entrées au clavier...)



3.2 Utilisation des Properties

Cette application utilise les propriétés (class *Properties*) pour la gestion du multi-langues. Par défaut, la langue anglaise est chargée. Il est possible de changer de langue en utilisant le menu du jeu.

Nous avons décidé d'inclure directement les propriétés de langue dans les propriétés gérées par la classe *System*. Cela permet notamment d'avoir accès de n'importe quelle classe aux propriétés créés par les fichiers de langues, sans avoir à transmettre entre chaque objet une référence sur des propriétés.

Pour tout objet, mot ou phrase qui pourraient être affichés, une entrée dans le fichier XML de langue est créée, avec en objet associé la traduction de l'objet dans la langue correspondante au fichier. Cela implique que toute clé présente dans un fichier XML de langue se doit d'être présente dans tous les fichiers XML. Si cela ne se vérifie pas, un simple terme "null" est affiché à la place de la traduction. Il suffit donc pour palier au problème que chaque fichier XML de langues ait exactement

les mêmes clés en valeur et en quantité totale.

Tous les fichiers XML de langues sont dans le dossier *xml/lang/* de l'application. Nous vous encourageons à leur donner des noms relativement explicites.

Nous avons essayé d'organiser les fichiers XML, en groupant par les mêmes préfixes les objets de même type. Voici une liste des groupes, et comme les noms de fichiers, nous vous encourageons à regrouper vos nouveaux objets dans les groupes déjà existants, s'ils existent. Mais vous êtes totalement libre du nom de la clé après tout.

Game-XXX Toutes les phrases affichées dans le déroulement du jeu

Menu-XXX Toutes les phrases contenues dans le menu de jeu

Word-XXX Mot basique

Action-XXX Traduction de chaque action, ces phrases seront affichées dans le menu

Item-XXX Traduction de chaque objet, phrases affichées dans les menus

Exception-XXX Explications de toutes les exceptions attrapables

Living-XXX Phrases concernant les joueurs

Map-Cell Traduction du mot cellule

Map-Ground-XXX Traduction d'un terrain

Map-Direction-XXX Traduction des valeurs des directions

3.3 Mise à jour de l'API

Sous la forme de tutoriels, vous apprendrez comment créer de nouveaux types de personnages, d'objets, d'actions... Notez que les menus évoluent dynamiquement quelque soient les modifications effectuées.

3.3.1 Créer un nouveau terrain

1. Pour ajouter un nouveau terrain, vous devez créer une sous-classe de *map.api.Ground* et l'insérer dans le paquetage *map.lib*.
2. Ensuite, il vous faut créer un constructeur privé sans paramètres. Ce constructeur devra appeler le super-constructeur, qui requiert 2 paramètres. Le premier est la clé associée au nom du terrain dans le fichier XML de langage. Le second paramètre est un entier qui correspond au coût d'énergie de la traversée du terrain.
3. Pour finir, il faut juste créer un attribut statique nommé *INSTANCE* du même type que la classe que vous avez créée, qui invoque le constructeur privé de la classe.

Exemple Pour présenter la création de terrain, nous allons créer un terrain constitué de boue. Traverser cette vallée de boue coûtera 60 points d'énergie aux aventuriers qui oseraient la traverser.

Filename map.lib.Mud

```
1 package map.lib;
2
3 import map.api.Ground;
4
5 /* Mud extends superclass Ground */
6 public class Mud extends Ground {
7
8     /* All grounds have a public static instantiation */
9     public static Mud INSTANCE = new Mud();
10
11     /* A private constructor. "Map-Ground-Mud" is the
12      * name designing this ground in the xml language file.
13      * 60 is the cost of crossing this ground
14      */
15     private Mud(){
16         super("Map-Ground-Mud", 60);
17     }
18 }
```

3.3.2 Créer un nouvel objet

1. Créez une classe dans le paquetage *item.lib* qui hérite de la classe *item.api.Item*.
2. Vous aurez juste à redéfinir une méthode et à implémenter un constructeur sans paramètre.
3. Le constructeur devra juste invoquer le super-constructeur qui demande 2 paramètres. Le premier paramètre est le nom de l'objet qui doit être la clé du vrai nom de l'objet dans le fichier XML de langage. Un nom correct est *Item-ClassName*, ou *ClassName* est le nom de la classe de l'objet. Le second paramètre est un entier qui sert à donner une valeur à l'action de l'objet lorsqu'on l'utilise.
4. La méthode à redéfinir est *usedBy(Living)*. Cette méthode sert à mettre à jour les informations du personnage qui utilise l'objet. Jetez un oeil à la Javadoc pour voir toutes les méthodes utilisables sur les personnages (*living.api.Living*).
5. Il est aussi possible de redéfinir la méthode *toString()*. Cette méthode est défini par la superclasse et retourne le nom traduit de l'objet.

Exemple Nous voulons ajouter un nouvel objet qui aura pour effet de faire des dégâts de zone, c'est à dire sur plusieurs personnages à la fois. Lors de l'utilisation de l'objet, tous les personnages sur la case à l'exception du possesseur de l'objet vont être affectés. Ils perdront N dommages. Tous ces N dommages cumulés seront transformés en $N * M$ pièces d'or, qui seront attribués à l'utilisateur de l'objet. Quel bel objet n'est ce pas?

Notez ici que l'on a besoin d'un attribut en plus pour bien calculer les effets du sort. L'on va se servir de l'attribut hérité pour y affecter la quantité de dégât qui sera occasionné sur chaque personne, et nous ajoutons un nouvel attribut M dans la sous-classe qui sera le ratio points de dégâts / pièces d'or.

Filename src/item/lib/CheatedItem.java

```
1 package item.lib;
2
3 import java.util.List;
4 import living.api.Living;
5 import item.api.Item;
6
7 /* Point 1 : the item class extends Item */
8 public class CheatedItem extends Item {
9     /* Item got a protected integer attribute, we decided to grant him the role
10        of the quantity of dommages done per living. We have to add a new
11        attribute which will serve to calculate how many golds the user will get
12        */
13     private int M;
14
15     /* Point 2 - 3 : A constructor without parameters, giving the name of the
16        entry item in the xml language file */
17     public CheatedItem(){
18         super("Item-CheatedItem", 10);
19         this.M = 2;
20     }
21
22     /* Point 2 - 4 : This method affects dommages on every player on the cell
23        excepts the living in parameter, and then gives money tho the living which
24        used this object */
25     @Override
26     public void usedBy(Living l){
27         List<Living> livingsOnCell = l.getCell().getLivings();
28         int totalGolds = 0;
29         for (Living opponent : livingsOnCell){
30             if (!(opponent == l)){
31                 totalGolds += this.M * this.quantity;
32                 opponent.addEnergy(-this.quantity);
33             }
34         }
35         l.addMoney(totalGolds);
36     }
37
38     /* Point 5 : We give more information about the item, like the ratio dommages
39        done / golds win */
```

```

33     @Override
34     public String toString(){
35         String s = super.toString();
36         s += "< " + this.quantity + " " + System.getProperty("Word-Dommages") + "
           = ";
37         s += this.M + " " + System.getProperty("Word-Gold") + " :: ";
38         s += System.getProperty("Word-AoE") + " >";
39         return s;
40     }
41 }

```

3.3.3 Créer un nouveau personnage

Il y a 2 types de personnages : ceux contrôlés par le joueur, et ceux contrôlés par l'ordinateur. Les actions du 2ème type de personnages sont choisies aléatoirement.

Originellement, le jeu a été conçu pour compter un seul et unique héros, et plus d'un personnages contrôlés par l'ordinateur. Nous allons en premier lieu vous expliquer comment créer un nouveau personnage, puis ensuite vous présenter les modifications à faire pour l'adapter en un héros.

Chaque personnage possède quatre paramètres à leurs création : Nom, force, énergie, nombre de pièces d'or.

Il est possible d'ajouter de nouveaux paramètres aux personnages que vous créerez. Mais si vous ajoutez un paramètre comme du mana par exemple, il est important de savoir qu'aucune action ne pourra exercer de modifications sur ce paramètre, ainsi que sur tous ceux ajoutés aux 4 paramètres originaux. Il serait cependant possible de forcer des modifications éventuelles dans des actions, mais l'on perdrait en "open/close" dans l'api.

La plupart des différences entre les personnages sont dans la méthode *interact(Living)*. Cette méthode définit la réaction d'un personnage lorsqu'un autre interagit avec.

1. Créer une sous-classe de *living.api.Living* dans le paquetage *living.lib*.
2. Créer un constructeur sans paramètre. Ce constructeur doit initialiser l'attribut *choice*, avec une instantiation d'une classe contenue dans le paquetage *choice.lib*.
Si vous voulez que les actions du personnage soit choisies aléatoirement, vous pouvez utiliser la classe *choice.lib.RandomChoice* proposée par l'API. Si vous voulez contrôler le personnage au clavier, vous pouvez utiliser *choice.lib.KeyboardChoice*. Il est aussi tout à fait possible d'implémenter une nouvelle possibilité de choix en créant une classe qui implémente l'interface *choice.api.Choice*.
3. Ce point n'est pas forcément obligatoire. Si votre personnage utilise uniquement les 4 attributs hérités par la superclasse, vous n'avez pas besoin de modifier la méthode *init(String[])*. Mais si vous désirez ajouter des attributs en plus de ces 4 hérités, vous devrez redéfinir cette méthode.
La méthode de la superclasse *init(String[])* attribue les 4 premiers éléments du tableau passé en paramètre aux attributs de *living.api.Living*. Donc si vous voulez ajouter des attributs en plus, vous devrez redéfinir la méthode. La redéfinition de cette méthode consisterait à invoquer cette même méthode de la superclasse qui s'occupera donc des 4 premiers éléments du tableau, puis de s'occuper du reste des éléments du tableau.
Remarquez donc qu'il est conseillé d'ajouter des valeurs par défaut aux attributs que vous ajoutez à vos classes, au cas ou vous oublieriez de renseigner suffisamment le tableau d'initialisation.
4. Bienvenue dans la méthode *interact(Living l)*. Vous allez devoir définir cette méthode. C'est ici que votre personnage prendra toute sa dimension. Comme un autre personnage interagira avec vous, que souhaiteriez-vous lui faire ? Lui voler de la vie ? De l'argent ? Tous ses objets ? Lui donner de la nourriture ? L'assassiner ? Lui parler ? Faites donc confiance en votre imagination et codez cette méthode ;)

Si vous voulez créer un héros, vous pouvez suivre ces 4 points. Mais à la place d'hériter de la classe *living.api.Living*, il est préférable d'étendre la classe *living.lib.Hero*. Et ainsi, vous n'aurez pas besoin de créer de nouveau constructeur.

Exemple Nous voulons ajouter comme nouveau personnage Dieu.

Quand quelqu'un interagit avec Dieu, Dieu juge cette personne. Si ce personnage est assez pure pour parler affaire avec Dieu, Dieu le bénira en lui doublant son nombre de points d'énergie. Sinon, Dieu le punira en le baffant jusqu'à ce que ce personnage se retrouve avec N points d'énergie.

Le jugement de Dieu peut être considéré comme un tirage aléatoire. La bonté de Dieu étant mesuré par l'attribut N.

Filename `src/living/lib/God.java`

```

1 package living.lib;
2

```

```

3  /* Beware of packages you need to import */
4  import java.util.Random;
5
6  import choice.lib.RandomChoice;
7  import living.api.Living;
8
9  public class God extends Living {
10     /* Kindness values per default is set to 1 */
11     private int kindness = 1;
12
13     /* An empty constructor, god will randomly select his actions */
14     public God(){
15         super();
16         this.choice = new RandomChoice();
17     }
18
19     /* As we have add a new parameter, we have to override init method. */
20     public void init(String args []){
21         /* The three first elements of args[] will be assigned to the 3
22          * parameters of living.api.Living, we have to check the 4th */
23         super.init(args);
24         try {
25             if (args.length > 3) this.kindness = Integer.parseInt(args
26                 [3]);
27         }
28         /* If an exception occured, kindness is set to default, here it's
29          1 */
30         catch (NumberFormatException e){
31         }
32     }
33
34     /* We will make a random, if random = 0, god lose, and living gives life */
35     /* Else if random = 1, god wins, and punish 1 */
36     public void interact(Living l){
37         Random random = new Random();
38         if (random.nextBoolean()){
39             /* God wins */
40             l.addEnergy(this.kindness - l.getEnergy());
41         }
42         else {
43             /* God lose */
44             l.setEnergyMax(l.getEnergy()*2);
45             l.addEnergy(l.getEnergy());
46         }
47     }
48 }

```

3.3.4 Créer une nouvelle action

Chaque capacité d'un personnage est définie par une unique action : La capacité de se battre est définie par l'action *actions.lib.Fight*, la capacité de ramasser un objet par *action.lib.CollectItem...*

Il est important de comprendre la méthode de selection des actions afin de bien saisir les différentes méthodes à redéfinir. Chaque action que l'on attribue à un personnage s'ajoute à la liste d'actions que gère ce personnage. Ces actions ne sont à ce moment aucunement paramétrées, il s'agit simplement d'instanciations primaire de chaque classe d'action. Le correct paramétrage des actions nécessite de connaitre diverses informations, selon le type d'action. Ainsi, l'action de ramasser un objet aura besoin de disposer de la référence vers la case pour pouvoir avoir accès à tous les objets posés sur cette case; l'action de frapper quelqu'un aura besoin de la case pour disposer de la liste des personnages présents sur la case... Etant donné que l'on ne peut agir que sur les éléments qui sont disponibles sur la case du personnage, le simple fait de connaitre le personnage qui lance l'action permet de disposer de tous les éléments potentiellement nécessaires au bon paramétrage de l'objet. Ainsi, prenons le cas ou nous donnons à un personnage la capacité de se battre et de ramasser un objet. Sur la case ou ce personnage se trouve, il y a deux autres personnages et aucun objet au sol. Une seule des actions est réalisable, car il n'y aucun item à ramasser, mais au moins un personnage sur la case. La méthode *action.api.realizable()* permet de déterminer si oui ou non une action est réalisable. Donc la première action générique choisie sera automatiquement celle de se battre.

Deux actions plus spécifiques en découlent, qui sont de se battre avec l'un ou l'autre des deux personnages. La méthode `action.api.listAvailableActions(Living)` se charge de créer une liste d'actions correctement paramétrées cette fois-ci, donc en renseignant le battant et le battu à partir des informations fournies par le paramètre `Living`. Cette liste d'actions servira ensuite pour le choix de l'action spécifique, qui ensuite pourra être convenablement exécutée selon la méthode `action.api.act()`.

`action.api.Action` est une classe abstraite ne contenant en attribut qu'un seul nom, qui sert notamment à l'affichage dans les menus de la traduction de l'action. Si vous désirez ajouter une action n'influençant aucunement sur le personnage qui l'exerce, ni sur ce qui pourrait être contenu sur la case de ce personnage, étendez cette classe.

`action.api.CharacterAction` est sous-classe abstraite de `action.api.Action` qui ajoute un attribut `Living` qui représente la personne qui lance l'action. Si vous désirez ajouter une action modifiant un élément du personnage, étendez cette classe.

Par exemple, `action.api.Interact` est sous-classe de `action.api.CharacterAction` et ajoute un nouvel attribut `Living`. Toutes les actions entre 2 personnages peuvent donc hériter de cette classe. De même, `action.api.UseItem` est sous-classe de `action.api.CharacterAction` et ajoute un attribut `Item`.

Voyons en quelques étapes comment ajouter une action.

1. En premier lieu, choisissez correctement la classe `Action` à hériter. Créez cette classe dans le paquetage `action.lib`.
2. Il est important que chaque `Action` ait un constructeur sans paramètre, pour les actions génériques, et d'un autre réclamant la totalité des paramètres, pour les actions spécifiques. Ces deux constructeurs devront appeler la méthode `action.api.Action.setName(String)` afin d'initialiser le nom de l'action, qui est la clé de la propriété correspondante dans le fichier XML de langue.
3. Redéfinissez la méthode `action.api.realizable(Living)` qui est la méthode qui permet de déterminer si oui ou non une action est réalisable, selon les informations récoltables par le paramètre `Living`.
4. Redéfinissez la méthode `action.api.act()` qui règle le déroulement de l'action. Une fois que cette action a été choisie par le personnage, cette méthode est appelée. Elle utilise les attributs de l'objet `Action` pour effectuer les éventuelles modifications.
5. Redéfinissez la méthode `action.api.listAvailableActions(Living)`. C'est cette méthode qui permettra de définir pour un personnage donné quelles actions de ce type il pourra exécuter et combien au maximum. Il est courant de recourir à la méthode `living.api.getCell()` pour obtenir une cellule qui recense tous les objets présents dessus.
6. Redéfinissez éventuellement la méthode `action.api.toString()` qui se chargera de renvoyer le texte nécessaire à l'affichage de l'action détaillée, c'est à dire lorsque l'action est totalement renseignée et devient donc applicable.

Exemple Pour illustrer la création d'action, nous allons créer une action que nous appellerons "SneakySteal". Le personnage disposant de cette capacité pourra, lorsque c'est à lui de jouer, choisir de voler un personnage. Nous n'allons pas pousser le vice trop loin, le vol ciblera uniquement les pièces d'or. Seulement, si un personnage n'a pas d'argent, nous allons à ce moment là lui voler l'ensemble de son inventaire, dans la limite de nos places libres dans l'inventaire.

Cette action nécessite donc pour fonctionner de deux caractères : celui qui vole, et celui qui est volé. Nous allons donc étendre la classe `action.lib.CharacterAction` et ajouter un attribut `Living` précisant le personnage volé. La méthode `action.lib.SneakySteal.realize(Living)` se contentera donc de renvoyer vrai si au moins un caractère sur la case dispose d'un pécule ou d'un objet. La méthode `action.lib.SneakySteal.listAvailableActions(Living)` quand à elle listera tous les vols possibles sur les caractères répondant aux deux critères cités juste précédemment. Et la méthode `action.lib.SneakySteal.act()` activera la fouille des poches.

```
1 package action.lib;
2
3 import item.api.Item;
4 import java.util.ArrayList;
5 import java.util.List;
6 import living.api.Living;
7 import action.api.Action;
8 import action.api.CharacterAction;
9
10 /** SneakySteal is an action that steals money to characters, or objects if
11     characters have no money.
12     */
13 public class SneakySteal extends CharacterAction {
14     private Living living2;
15
16     public SneakySteal(){
17         super();
18         this.setName("Action-SneakySteal");
19     }
20 }
```

```

20 public SneakySteal(Living l1, Living l2){
21     super(l1);
22     this.living2 = l2;
23 }
24
25 /** Steals money to living2.
26  * If living2 got no money, steals all objects from living2' inventory.
27  */
28 @Override
29 public void act() {
30     int money = living2.getMoney();
31     if (money > 0){
32         living2.giveMoneyTo(living1, money);
33     }
34     else {
35         List<Item> items = living2.getInventory().getItems();
36         for (Item it : items){
37             if (!living1.getInventory().isFull()) living1.getInventory().
38                 addItem(it);
39         }
40     }
41
42 /** Fills a list with realizable actions.
43  * @return a List of actions of this type.
44  */
45 @Override
46 public List<Action> listAvailableActions(Living l) {
47     List<Action> actions = new ArrayList<Action>();
48     List<Living> livings = l.getCell().getLivings();
49     for (Living liv : livings){
50         if (liv != l && (liv.getMoney() > 0 || l.getInventory().getItems().
51             size() > 0)){
52             actions.add(new SneakySteal(l, liv));
53         }
54     }
55     return actions;
56 }
57
58 /** Determinates if the action is realizable or not.
59  * Action is possible if there is at least one character on the cell with
60  * some golds or some objects.
61  * @return True if realizable.
62  */
63 @Override
64 public boolean realizable(Living l) {
65     List<Living> livings = l.getCell().getLivings();
66     for (Living liv : livings){
67         if (liv != l && (liv.getMoney() > 0 || l.getInventory().getItems().
68             size() > 0)) return true;
69     }
70     return false;
71 }
72
73 /** Returns a string representation of whats going to be steal. Count of
74  * items, or golds quantity. */
75 @Override
76 public String toString() {
77     String s = new String(this.living2.getName() + " : ");
78     if (this.living2.getMoney() > 0) s += this.living2.getMoney() + " " +
79         System.getProperty("Word-Gold");
80     else s += this.living2.getInventory().getItems().size() + System.
81         getProperty("Word-Item");

```

```

76     return s;
77 }
78 }

```

3.3.5 Créer un nouveau monde

Actuellement, un seul type de monde existe : Un monde délimité par une certaine longueur et largeur, ou chaque case dispose de quatre voisins lorsque cela est possible (comprenez que cela est impossible lorsque la case se trouve à une bordure du monde), qui sont dénommés par les classiques termes "Nord", "Sud", "Est", "Ouest". Deux choses sont particulièrement importantes pour créer un monde, à savoir la représentation spatiale du monde, et les liens de voisinages que chaque case ont entre elles. Le voisinage est organisé par des directions, qui doivent implémenter l'interface *map.api.Direction*. Les directions d'un monde doivent simplement être une énumération.

L'affichage du monde ne sera pas détaillé ici, sachez juste que la méthode *world.api.toString()* est la pour être redéfinie si vous voulez.

La mise à jour des voisins se fait via la méthode *world.api.World.setNeighbours()*.

La création des cellules se fait par la méthode *world.api.World.create(String[])* qui selon les paramètres passés dans le tableau, crée un certain nombre de cellules. Cette méthode doit appeler ensuite la méthode *setNeighbours()*. Ensuite, le monde peut être considéré comme fonctionnel.

Exemple Imaginons par exemple que vous vouliez créer un monde du même type que le monde 2D que nous avons implémenté originellement dans le jeu. Votre seul désir est qu'il n'y ait pas de bordure dans ce monde, que votre monde ait la forme d'une boule. C'est à dire que lorsque l'on est situé tout en haut du plateau, se déplacer au nord soit possible et que cela nous amène au sud de la carte. De même pour les autres directions.

1. Comme notre monde se trouve être une extension d'un monde en 2 divisions, notre classe *world.lib.BoolWorld2D* héritera de *world.lib.World2D*.
2. Nous ne nous occuperons pas de la méthode *world.api.World.create(String[])* qui finalement se contente de créer un certain nombre de cases selon la largeur et longueur demandée pour le monde. Celle-ci ne sera pas redéfinie, car l'architecture de notre nouveau monde ne variera pas, seule le voisinage le sera.
3. Redéfinir la méthode *world.lib.World2D.setNeighbours()* qui a pour objectifs de relier les cases entre eux. Sans l'invocation de cette méthode, vos personnages auront extrêmement de mal à se déplacer. Ici, l'on invoque la super méthode qui s'occupe des reliages à faire. Nous ajouterons juste 2 boucles pour rendre voisins les cases en bordures uniquement.

```

1  package world.lib;
2
3  import map.api.Cell;
4  import map.lib.NorthSouthEastWest;
5
6  /** BoolWorld2D has the same construction than World2D, excepts all cells have 4
7     neighbours. */
8  public class BoolWorld2D extends World2D {
9
10     /** Sets new neighbours to every border cells. */
11     public void setNeighbours(){
12         super.setNeighbours();
13         /* When the cell is at the top north, it gets the cell at the extrem
14            south
15            * as neighbour associated with the north direction */
16         for (int i = 0; i < this.height; i++){
17             Cell cell1 = this.cells.get(i * this.width);
18             Cell cell2 = this.cells.get((i+1) * this.width - 1);
19             cell1.addDirection(NorthSouthEastWest.West, cell2);
20             cell2.addDirection(NorthSouthEastWest.East, cell1);
21         }
22         /* When the cell is at the top east, it gets the cell at the extrem west
23            * as neighbour associated with the east direction */
24         for (int i = 0; i < this.width; i++){
25             Cell cell1 = this.cells.get(i);
26             Cell cell2 = this.cells.get(i + this.width * (this.height-1));
27             cell1.addDirection(NorthSouthEastWest.North, cell2);
28             cell2.addDirection(NorthSouthEastWest.South, cell1);
29         }
30     }
31 }

```

3.4 Originalité du projet ?

Ce jeu, par la manière dont il a été structuré et pensé, permet une évolution facile et rapide.

Pour ce qui est du simple joueur, la simple possibilité de configurer le monde, de gérer les actions des différents personnages, leurs caractéristiques... donnent une durée de vie assez grande.

Pour ce qui est du programmeur, le principe "open/close" étant respecté (sauf pour le hack des objets de richesses qui ne sont pas stockables en inventaire), il est aisé de mettre à jour le projet. Comme présenté dans la partie de mise à jour de l'API, il est facile d'ajouter actions, personnages, objets. Par exemple, il serait intéressant d'implémenter un système d'évolution des caractéristiques des personnages selon leurs rencontres dans le monde. Un système de faction le serait tout autant, donner un soupçon d'intelligence aux personnages afin qu'ils ne se suicident plus en marchant d'un territoire à l'autre le serait encore plus...

Un défaut à noter serait situé dans le code de définitions des règles du jeu, dans la classe *game.api.Game*, qui est trop lié à l'affichage du jeu. Ajouter une IHM ne serait possible sans redéfinir les règles, même si celles-ci ne changent pas.

4 Conclusion

Ce projet aura été l'occasion de découvrir de nouvelles techniques de programmation, notamment la version 1.5 de Java, la librairie Jdom et l'utilisation des fichiers XML, l'externalisation de propriétés dans une application, ainsi que de nouvelles applications comme Ant.

La rédaction d'une documentation complète aura permis de se rendre compte de l'ampleur d'une telle tâche, documentation nécessaire afin que non seulement l'utilisateur de l'application ne soit pas désappointé lors de l'utilisation du programme, mais surtout pour aider le programmeur qui souhaite modifier et améliorer cette application.

Nous espérons que notre jeu vous plaira, et, que cela se vérifie ou non, il ne tient qu'à vous d'en faire un meilleur jeu !