CHARPENTIER Romain
CORNET Florian
L3 Informatique

Projet de Programmation Orientée Objet

Un petit jeu d'aventure en mode texte

<u>Date :</u> Vendredi 2 Décembre 2016

Table des matières

Avant de commencer, nous souhaitons préciser que l'intégralité du digramme UML de notre projet est à trouver dans le dossier de notre projet pour des raisons de lisibilité. Vous trouverez cependant des parties de notre UML dans notre rapport.

I Présentation générale

Le jeu se joue avec la console. Le joueur incarne un personnage sans nom qui se réveille après une soirée dans un bar. Il veut vite rentrer chez lui avant que son conjoint ne réveille. Malheureusement il ne sait plus où est sa maison et a perdu ses clefs… Il devra donc aller de maison en maison et de rue en rue en quête de ces derniers dans un temps limité !

Le héros pourra rencontrer pendant sa « quête » plusieurs types de PNJs et ramasser plusieurs types d'objets. Le héros gagnera en puissant à chaque fois qu'il boira de l'alcool, son taux d'alcoolémie est son plus grand atout.

A chaque partie, les bases changent, les rues n'ont pas la même taille, les PNJs ne s'appellent plus pareil, ne sont plus au même endroit et de même pour les objets ! Chaque partie sera unique et cela permet une meilleure expérience de jeu. Le joueur peut aussi choisir la taille de la carte (il choisit le nombre de rues pour des parties plus ou moins longues).

On utilise dans ce projet un package « util » qui nous sert à diverses méthodes utilitaires tel que pour générer un nombre au hasard, faire un affichage caractère par caractère choisir une ligne au hasard dans un fichier ou encore pour utiliser les interactions homme-machine (on utilise ceci pour permettre par exemple une interface graphique et ne pas contraindre le joueur à la console).

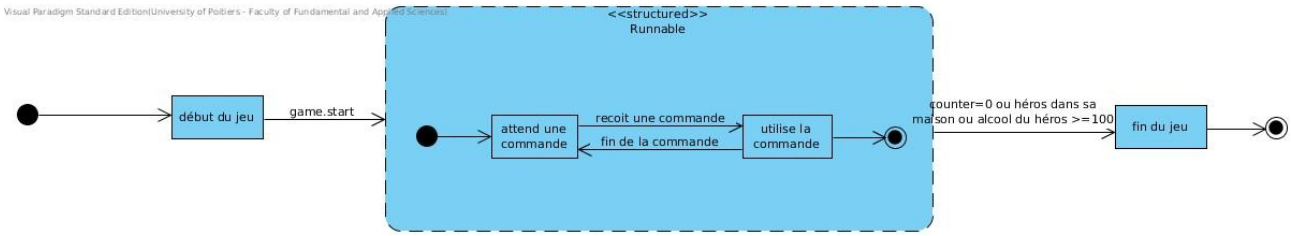


Diagramme d'états du jeu

Pour la suite, nous avons choisi de modéliser les diagrammes de séquences des méthodes que nous avons jugé les plus importantes et avec le plus d'éléments.

II Les commandes

Pendant le déroulement de la partie, les commandes sont récupérées à partir d'un scanner (on utilise une classe intermédiaire pour permettre une interface graphique par exemple et ne pas se

limiter à la console). On récupère ainsi un String qu'on va split en fonction des espaces pour ensuite séparer chaque argument dans un tableau de String.

Ainsi on va obtenir un tableau avec comme premier élément la commande et ensuite les arguments de la commande. Avec ce premier élément, on va trouver la commande avec « Command.valueof(monString.toUpperCase()) ». On doit mettre le string en majuscule car les commandes sont écrites dans le type enum en majuscules.

II.1 Attack

Le héros attaque un PNJ face à lui. Il ne peut y avoir qu'un seul PNJ devant lui donc cette commande ne nécessite pas de paramètre. Toutefois si aucun PNJ ne se trouve face à lui, ce dernier attaque le « vide » (sous l'effet de l'alcool).

La commande appelle la méthode « attack » du héros sur le pnj qui se trouve dans le lieu où se trouve le héros. Le pnj va donc perdre des points de vie (avec une méthode dans la classe NPC qui lui enlève de la vie) en fonction de l'attaque et de l'alcoolémie du héros. Ensuite si le PNJ est est un « Enemy » alors il va riposter et on doit alors tester si le héros est encore en vie, s'il ne l'est pas c'est une défaite et le programme va s'arrêter.

Voici le diagramme de séquence de la méthode attack de Hero (qui est appelée quand on utilise la commande « Attack ») :

(Nous devons ici gérer tous les cas, par exemple celui où le héros meurt)

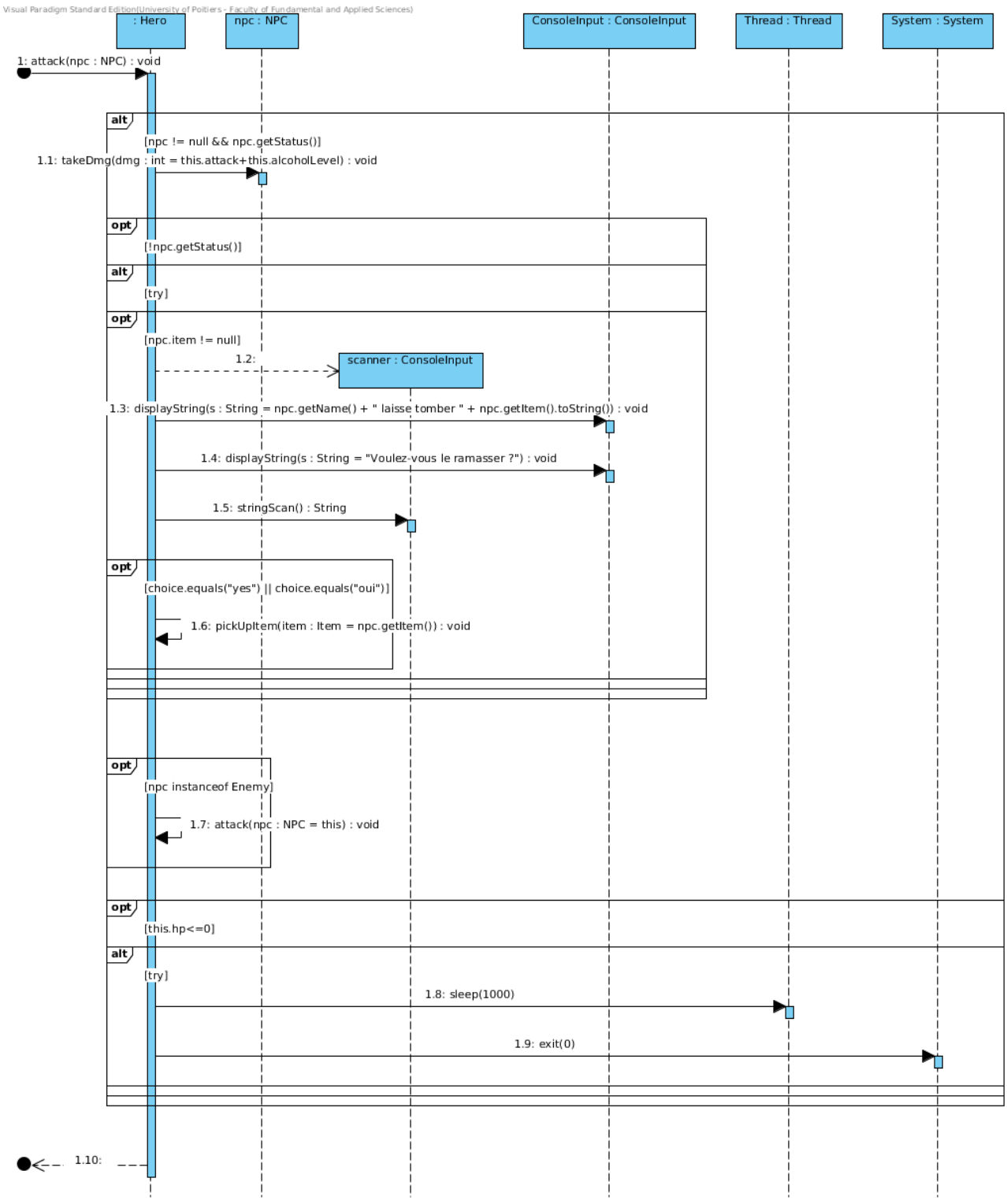


Diagramme de séquence de la méthode attack de Hero

II.2 Delete

La commande delete retire un objet de l'inventaire du héros. Pour cela, on parcourt l'inventaire du héros et on s'arrête quand on trouve l'objet qui a le même nom (string) que l'argument que l'on donne à la commande delete. Si on le trouve (on prend le premier item avec le

bon nom), on le supprime de la liste (l'inventaire du héros) avec la méthode remove et sinon on ne fait rien.

II.3 Go

La commande go permet au héros de se déplacer sur la carte. Elle nécessite un argument qui est la direction (String) où le joueur souhaite se rendre. La fonction va tester si cette issue est possible, c'est à dire si elle se trouve dans la HashMap « exits » qui est la liste des sorties d'une « Place ».

Si la sortie est possible, on va aussi tester si c'est une maison car une maison peut posséder des entrées bloquées. Si tel est le cas alors le joueur doit ouvrir la porte (clé, énigme…) sinon il va simplement se déplacer dans la direction qu'il a renseigné.

Cette commande utilise la méthode « go » du héros qui va modifier sa Place actuelle (qu'il a en attributs), on passe aussi par l'intermédiaire des méthodes de sa Place actuelle et la sortie qui est liée à la direction renseignée. C'est donc le héros qui se déplace et non pas la carte qui déplace le héros car seul celui-ci connaît sa position. Il change donc sa Place pour la Place de l'Exit de la direction qu'il a renseigné (qu'on retrouve avec la méthode « getNextPlace » de la classe « Place ».

Voici le diagramme de séquence de la méthode « go » dans la classe Hero (méthode qui est appelée par la commande) :

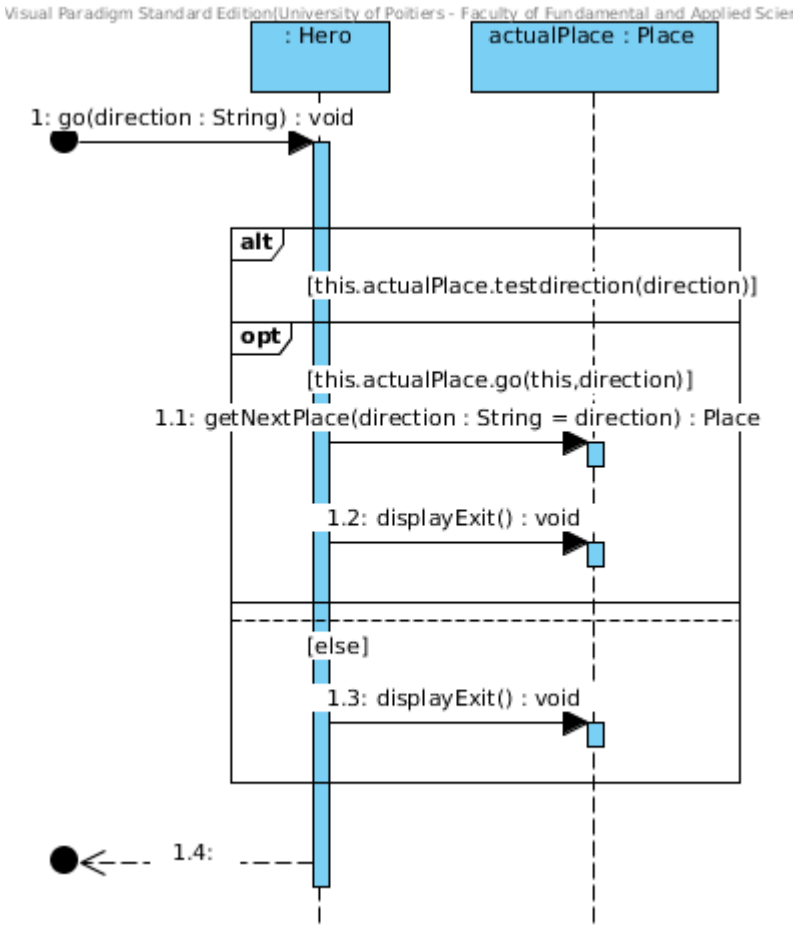


Diagramme de séquence de la méthode go du Hero

II.4 Help

La commande help affiche toutes les commandes disponibles avec un petit descriptif. Pour cela, elle parcourt les valeurs du type enum « Commandes » avec un for..each et affiche leurs noms et leurs descriptions (qui est en attribut de chaque valeur du type enum).

II.5 Inventory

Cette commande affiche l'inventaire du héros avec la commande printInventory de Hero. Elle parcourt donc la liste d'objets du héros (donc l'inventaire de ce dernier) et affiche le nom de chaque objet qui la compose (chaque item redéfinit la méthode toString),

II.6 Look

Cette commande permet de décrire un lieu ou un objet avec la méthode look de Hero. Si la commande n'a pas d'argument, elle va décrire le lieu actuel du héros. Si l'argument est une direction, elle va décrire le lieu suivant la direction qu'indique le joueur. Si l'argument est un objet que possède le héros alors elle va décrire l'objet. Cette commande utilise les méthodes « describe » des classes Item et Place.

II.7 Quit

La commande quit permet simplement de quitter la partie. Elle utilise la commande « System.exit » pour arrêter le programme. Elle utilise aussi un thread pour une petite animation lors de l'exécution de la commande. En utilisant cette commande, on quitte la partie après 2 secondes.

II.8 Status

Cette commande affiche les caractéristiques du héros. Cela permet au joueur de connaître sa vie, son alcoolémie et son arme équipée. Cette commande est surtout utile pour ne pas devoir retenir sa vie ou devoir le noter. Elle utilise simplement les getters de la classe Hero.

II.9 Take

La commande take permet au joueur de ramasser les objets contenus dans une maison. Pour cela le joueur doit saisir en argument le nom de l'objet à ramasser. Evidemment le joueur ne peut pas ramasser un objet qui ne se trouve pas dans la maison. Cette commande appelle la méthode pickUpItem de la classe Hero.

II.10 Talk

La commande talk sert à communiquer avec le pnj s'il y en a un présent dans la pièce. On peut cependant utiliser la commande s'il n'y a pas de pnj, il y aura alors simplement un affichage de texte. Si le pnj est présent, en fonction de sa catégorie (Enemy or Neutral) les actions varient. S'il est neutral alors il accueille le héros et lui propose gentiment son objet, il peut alors partir de la

maison (le joueur doit accepter ou refuser avec la console (oui ou non)). S'il est enemy alors il refuse de parler au héros et l'attaque avec la méthode attack qu'il y a dans la classe Enemy.

On teste donc s'il y a un pnj dans la place actuelle du héros et s'il y en a un, on utilise sa méthode « talk ».

II.11 Unequip

Si le héros possède une arme, la commande appelle la méthode unequip de Hero qui range l'arme dans l'inventaire s'il y a de la place ou la détruit et remplace donc l'attribut weapon de Hero par null (qui signifie donc « sans arme »).

II.12 Use

La commande USE se distingue en 2 méthodes. La première consiste à utiliser un objet contenu dans l'inventaire du héros, c'est-à-dire manger, boire ou s'équiper de l'objet utilisé. Pour cela on appelle la méthode "use" de la classe Hero ayant comme prototype : use(String item). Quant à la seconde méthode elle appelle une autre fonction "use" de la classe Hero mais qui à comme prototype : use(String item1, String item2).

Cette méthode ne permet pas d'utiliser directement un objet sur le héros mais plutôt de fusionner 2 objets de même type posséder par le héros pour en créer un meilleur afin de l'utiliser plus tard.

Voici le diagramme de séquence de la méthode use d'un objet :

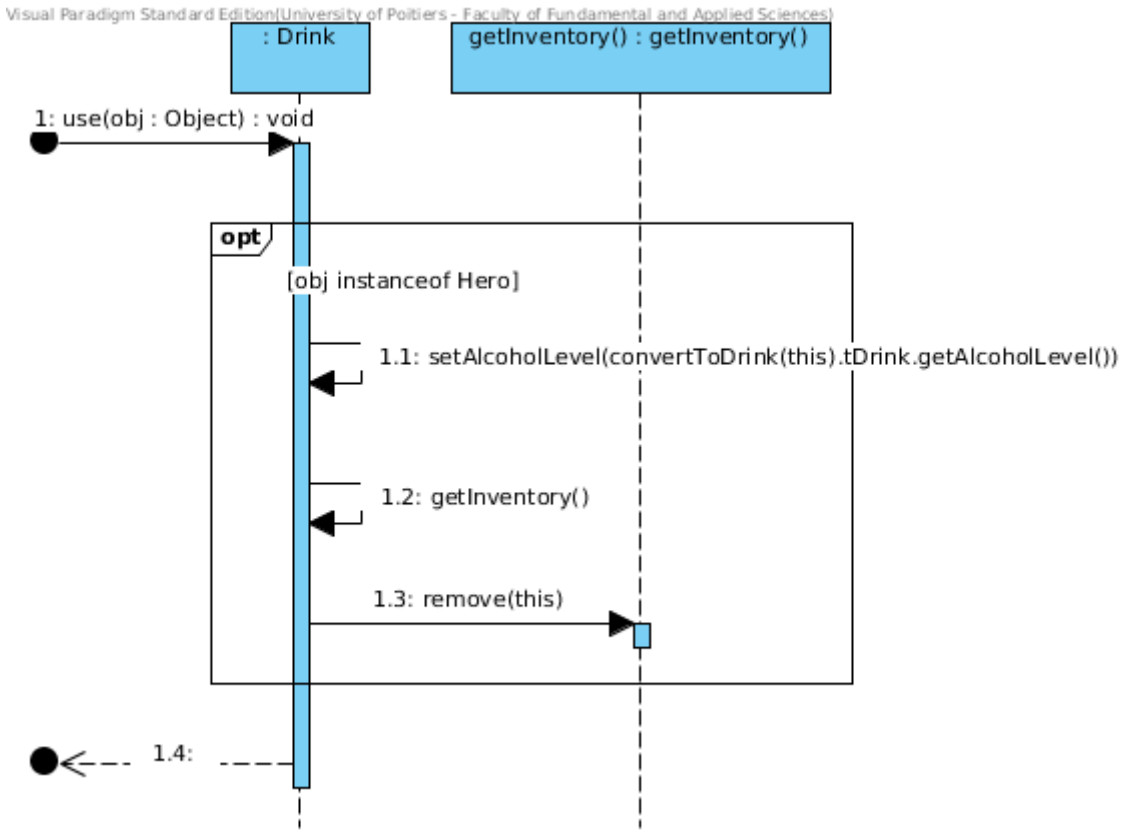
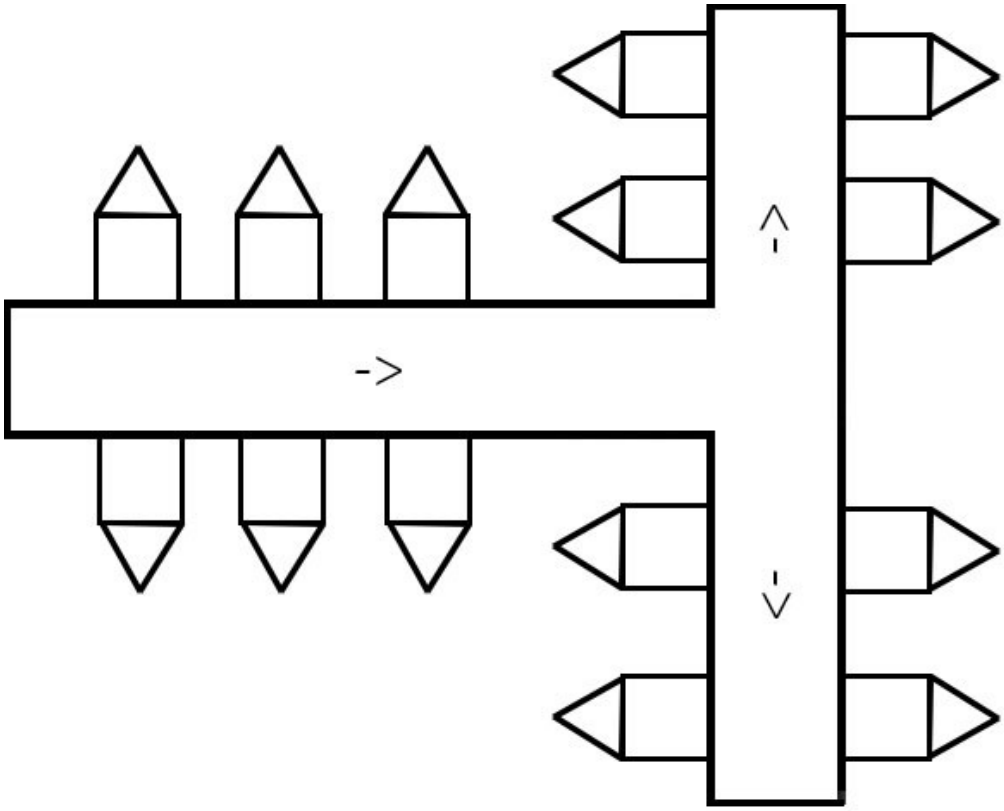


Diagramme de séquence de la méthode use d'un Item

III La carte



Exemple de carte (3 rues)

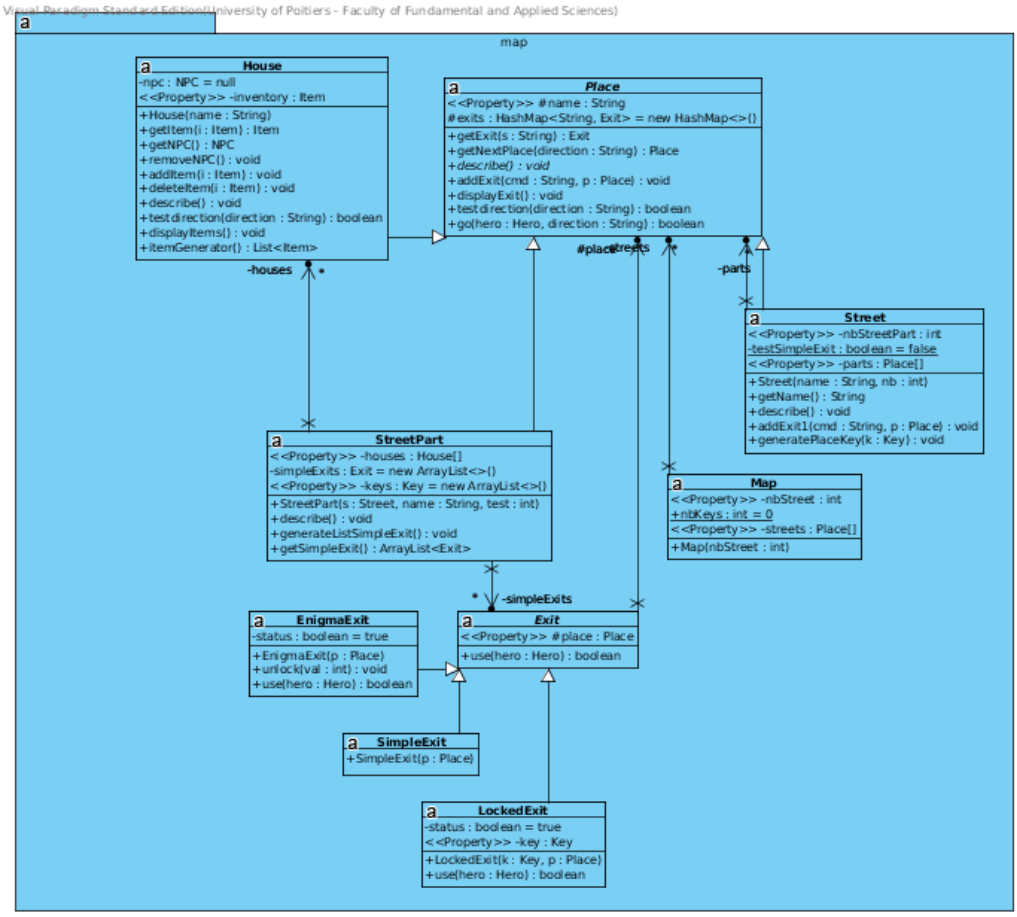
La carte du jeu se génère aléatoirement en fonction de la partie. Le joueur a aussi le choix entre une carte avec 1 ou 3 rues. La carte se divise en rues « Street » puis en parcelles de rues « StreetPart » puis en maison « House ». Tout est généré aléatoirement, nous devons donc faire en sorte de ne pas bloquer le joueur. Nous avons donc choisi de « forcer » l'ajout de portes verrouillées ou non dans un « StreetPart », c'est pourquoi il y a un entier « test » en paramètre du constructeur. Dans chaque rue, nous allons ajouter de force une porte classique et une porte verrouillée (des « Exit » qui accèdent aux maisons) à des maisons choisies aléatoirement.

La rue contient un nombre entre 3 et 6 de parcelles et une parcelle contient toujours 2 maisons sur chaque côté (voir le schéma). Les maisons peuvent avoir des portes verrouillées ou non (choisi aléatoirement). Chaque rue a un nom choisi aléatoirement dans une liste, de même pour les maisons (le nom sur les boites aux lettres, le nom peut être inconnu).

Le joueur doit se déplacer sur la carte avec la commande « Go » suivi de la direction. Il faut savoir que pour se diriger vers une maison il faut indiquer le numéro de la maison (1 ou 2) comme ceci : « go house2 ». Les maisons peuvent très bien avoir les mêmes noms, donc nous ne pouvons pas utiliser la commande go avec comme argument le nom de la maison. Par contre le joueur écrit bien le nom de la rue pour aller d'une rue à l'autre car les rues ont chacune un nom différent. Nous

avons essayé de rendre la carte plutôt réaliste. Le joueur peut aussi, bien entendu, se déplacer d'avant en arrière dans la rue en indiquant « forward » ou « backward ».

La carte est essentiellement composé des classes « Place » et « Exit », vous pouvez avoir un aperçu de la conception de la carte avec l'UML suivant :



UML du package Map

III.1 Place

La classe « Place » est l'ensemble des lieux possibles dans la Map. Nous retrouvons ici « Street », « StreetPart » et « House » dont nous avons parlé précédemment. Ces classes héritent toutes de Place pour pouvoir rendre possible le déplacement du héros sur n'importe quelle Place. Cela nous permet également d'y mettre les méthodes identiques pour tous les lieux, comme par exemple la méthode qui affiche les sorties possibles du lieu.

On peut remarquer que la maison a beaucoup de méthodes en plus, c'est le cas car elle est assez différentes des Place « classiques », en effet elle a en plus un PNJ et des objets. Nous avons donc dû y intégrer des méthodes pour interagir avec le contenu des maisons.

Chaque Place connaît également la liste des sorties possibles. On les contient dans une HashMap qui permet aussi d'indiquer le type de direction (en String) pour utiliser la sortie.

Plusieurs méthodes sont en interaction avec cette HashMap notamment « testdirection » et « go ». La première vérifie si la direction en paramètre est présente dans la HashMap, la deuxième sert à utiliser une sortie (en fonction d'une direction), si elle est bien dans la HashMap. Elle renvoie un booleen car le déplacement n'est pas forcément possible et il faut l'indiquer au héros.

III.2 Exit

(Précision : une Place peut avoir une porte verrouillée entre une Place alors que cette dernière a une porte simple avec l'autre.)

Dans la classe Exit, on voit peut d'éléments car chaque type d'Exit est très différent. Ils vont cependant tous avoir la méthode « use » qui permet de se rendre dans la Place qu'ils ont en attribut. Cette méthode est souvent « Override » par les classes filles car elles ont des fonctionnement différents. La classe contient aussi un getter sur la Place en attribut car nous devons l'utiliser pour les méthodes de déplacement de la classe « Place ».

Nous allons maintenant détaillé la liste des sorties possibles pour que vous compreniez le fonctionnement de chacune.

SimpleExit

Cette sortie est une sortie classique sans condition. Elle permet notamment de se déplacer dans la rue et vers certaines maisons. Cette classe est surtout utilisé pour ne pas bloquer le joueur. En effet, nous avons choisi de lister dans chaque « StreetPart » la liste des « SimpleExit » pour choisir dans chaque rue, une maison avec une « SimpleExit » qui possède une clé afin de ne pas bloquer le joueur tout en conservant une carte générée aléatoirement.

Il y a nécessairement une SimpleExit dans chaque rue pour ne pas bloquer le joueur.

EnigmaExit

La sortie EnigmaExit a d'enigme que de nom, elle demande simplement au joueur la réponse à une multiplication (c'est vrai que pour certains la réponse peut sembler impossible). Les nombres de l'opération sont aléatoires entre 0 et 15 et sont générés à chaque fois que le héros veut utiliser la sortie pour se rendre dans la maison correspondante. Ces nombres changent donc à chaque fois que le héros réessaie.

On utilise ici un scanner (dans le package « util ») pour demander une entier au joueur, il peut également abandonner en écrivant « 0 » (cela ne crée pas de conflit si la réponse est « 0 », la porte va s'ouvrir normalement). Une fois, déverrouillée, la porte le reste avec un booleen en attribut (il renseigne si la porte est verrouillée ou non). Si la porte a été déverrouillée, on utilise la méthode « use » de la classe mère, elle devient comme une « SimpleExit ».

LockedExit

Cette sortie est la plus compliquée des trois, elles utilisent un objet pour être ouverte, une clé. A chaque fois que l'on va ajouter une sortie de ce type (qui est, je le rappelle un choix

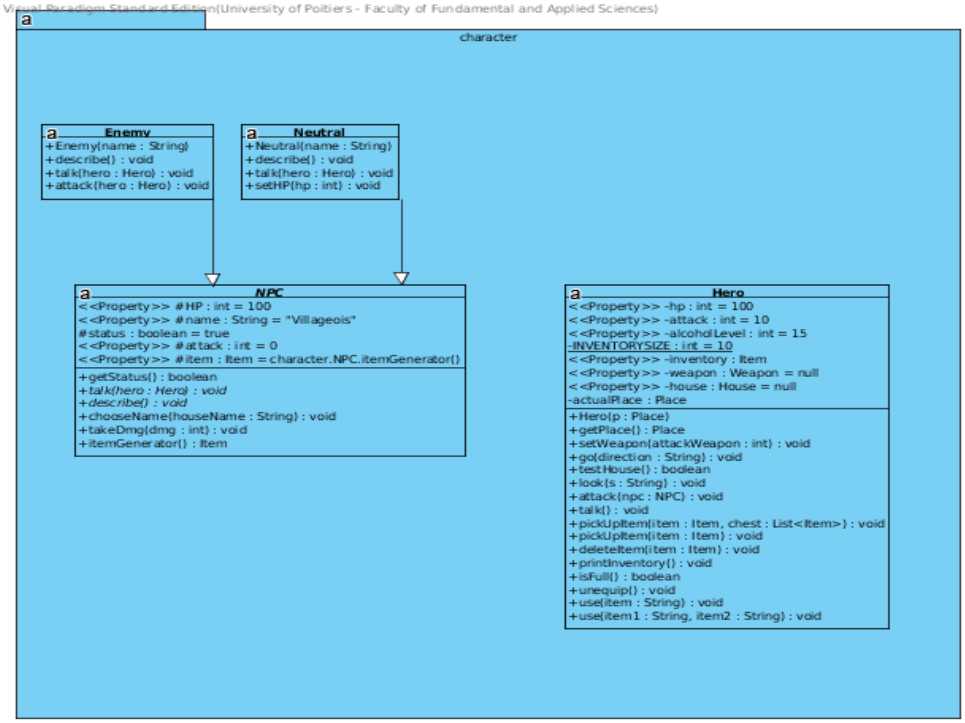
aléatoire), nous devons donc ajouter la clé sur la carte. Nous avons choisi de juste ajouter la clé dans une maison de la même rue. Pour cela, nous avons listé dans chaque « StreetPart » la liste des clés qu'il faut générer. Liste que nous reprenons dans la classe « Street », on utilise la méthode « generatePlaceKey » de la classe pour chaque clé de chaque « StreetPart ». Cette méthode va générer la liste des sorties possibles (vers une maison) et en choisir une aléatoirement pour y placer la clé.

On utilise un attribut booleen en static « testSimpleExit » pour placer la première clé (et seulement la première clé de la première rue) dans une maison avec une porte classique (non verrouillée) pour ne pas bloquer le joueur en quête de ces clés.

On fait toutes ces opérations dans les constructeurs des classes pour ne pas avoir à le faire manuellement à l'initialisation du jeu.

IV Les personnages

Les personnages sont composés du héros et des PNJs. Le héros est unique et est incarné par le joueur durant la partie. Les PNJs ont 2 « types » et peuvent être de la classe Neutral ou Enemy. Chaque personnage possède des points de vie (=100) et une valeur d'attaque. Le héros et les PNJs interagissent entre eux par le biais de la carte (les maisons connaissent les PNJs qui la composent), vous pourrez l'observer sur l'UML complet de notre projet, voici un aperçu des méthodes qui composent le package des personnages :



UML du package Character

IV.1 Le héros

C’est le personnage que le joueur va incarné pendant la partie, il suit les directives du joueur et le jeu est centré sur lui. La plupart des commandes utilisent donc des méthodes dans la classe « Hero ». Au début du jeu, on lui attribue une maison avec une porte fermée (choisie aléatoirement) qu’il doit retrouver pour gagner la partie avec le temps imparti.

En plus des points de vie et de l’attaque, il possède aussi un inventaire (une liste de taille prédéfinie par une constante), un niveau d’alcool qui sert à augmenter sa force (à consommer avec modération pour éviter un coma éthylique).

Il connaît sa position actuelle (contrairement à la carte) et sa maison (même si dans l’histoire le héros ne connaît pas sa maison et doit la retrouver, cet élément sert à savoir si le héros parvient à rentrer chez lui, ce qui est la condition de victoire). Le héros peut également s’équiper d’une arme avec laquelle il fera plus de dommages sur les PNJs.

Le héros peut se déplacer sur la carte, entre chaque Place. Il peut également interagir avec les objets et les PNJs.

IV.2 Les PNJs

Les PNJs sont séparés en 2 catégories, on peut dire qu’il y a les « gentils » (Neutral) et les « méchants » (Enemy). Chaque PNJ a un nom généré aléatoirement à partir d’une liste de nom (il a comme nom par défaut « Villageois », il peut le garder en fonction d’un random) et d’une méthode dans la classe Name du package « util ». Ils ont aussi un « status » qui signifie s’ils sont en vie ou non et un objet sur eux généré aléatoirement (on a ici choisi d’en mettre un seul pour ne pas surcharger le joueur d’objets, il y en a déjà assez dans les maisons). Chaque PNJ possède aussi une méthode « describe » qui sert de message d’accueil quand le héros rentre chez eux. Les phrases dites par les PNJs sont choisies aléatoirement.

Neutral

Ce type de PNJ ne s’attaque pas au héros, il peut cependant être attaqué et mourir, c’est une façon de récupérer son objet. Cependant, en lui parlant il proposera de donner son objet, c’est une méthode plus rapide !

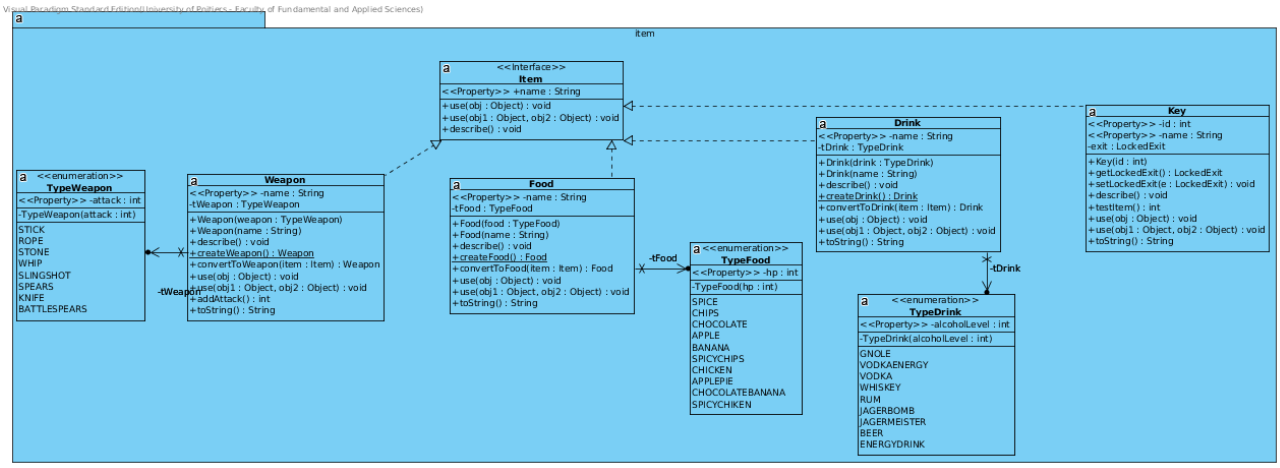
Enemy

Ce type de PNJ peut s’attaquer au héros s’il est attaqué avant ou si le héros essaie de lui parler. Il est donc agressif et vous empêchera de partir de chez lui (ce qui est un peu illogique car il vous demande de partir, mais une personne réveillée tard dans la nuit par un ivrogne peut se montrer illogique).

Contrairement au PNJ Neutral, il possède une attaque qui a une valeur aléatoire entre 10 et 25 et sera donc plus difficile à tuer. Il n’y a pas de méthode plus rapide ici, il faut le tuer pour avancer et récupérer les objets de la maison et aussi sortir de la maison.

V Les Objets

Parlons maintenant des objets. Parmi eux, on retrouve 4 classes qui héritent toutes d'une classe commune appelé Item. Nous allons, dans ce rapport, distinguer 2 catégories.
La première qualifiable de "consommable" qui contient les objets de type nourriture (Food), ceux de type boisson (Drink) et enfin ceux de type arme (Weapon).
La seconde catégorie quant à elle ne contient que la classe des clefs (Key).



UML du package Item

V.1 Les consommables

La catégorie des consommables comporte donc 3 classes d'objets qui sont chacune liée à des types énumérés (respectivement TypeFood, TypeDrink et TypeWeapon) qui contiennent les différentes valeurs que peuvent prendre ces différents objets.

A la place de passer par des types énumérés nous aurions pu définir une classe pour chaque type d'objet, cependant chaque objet du même type apportera toujours la même chose. Par exemple, si on créer un objet "pomme", lorsque le joueur l'utilisera il récupérera toujours 3 points de vie. L'intérêt de passer par une classe spécifique Pomme aurai été plus avantageuse si le nombre de vie rendu par une pomme pouvait varier. Ainsi nous avons préféré l'utilisation du type énuméré à celle de classe spécifiquement faite pour un objet qui ne changerait jamais.

Pour cette catégorie, la génération des objets se fait de manière aléatoire dans les maisons et sur les PNJ à l'aide de la méthode "itemGenerator".

En ce qui concerne le contenu des 3 classes de cette catégorie nous y retrouvons des méthodes similaires. Nous allons étudier les plus importantes à travers l'exemple de la classe Food.

Premièrement, on retrouve createFood() qui renvoie un Item de type Food avec pour valeur une valeur choisie aléatoirement dans le type énuméré TypeFood.

Nous retrouvons ensuite la méthode convertToFood(Item item) qui permet à partir d'un

objet de type Item d'obtenir un objet de type Food en vérifiant si le nom de l'item est bien dans l'énumération TypeFood.

Ensuite il y a la méthode use(Object obj) qui permet d'utiliser un objet de type Food sur le héros. Pour cela on test si l'objet mit en paramètre correspond bien au héros avec "instanceof" puis nous ajoutons la valeur de l'item de type Food utilisé au point de vie du héros, sachant que les points de vie ont pour valeur maximale 100.

Enfin, la dernière méthode use(Object obj1, Object obj2) permet de combine 2 items de même type (ici 2 Food) afin d'en obtenir un plus avantageux. Pour cela on commence par vérifier que le premier objet est bien un objet de type Food et que le second correspond bien à nouveau au héros. On regarde ensuite les 2 items afin de renvoyer l'objet de fusion souhaité (exemple, avec 2 pommes on obtient une tarte aux pommes).

Regardons désormais ces classes une par une pour comprendre leur utilité.

La nourriture

La classe nourriture (Food) est liée à l'énumération TypeFood regroupant donc les différentes nourritures que le joueur pourra retrouver dans le jeu. Il est également possible de fusionner certains de ces objets pour en obtenir de nouveau via la méthode USE décrite précédemment. L'utilisation d'un tel objet par le joueur lui permet de regagner un certain nombre de point de vie.

Liste des objets fusionnables :

Objet 1	Objet 2	Résultat de la fusion
Chocolate	Banana	Chocolatebanana
Apple	Apple	Applepie
Spice	Chips	Spicychips
Spice	Chicken	SpicyChicken

Les boissons

La classe boisson (Drink) est liée à l'énumération TypeDrink qui regroupe les différentes boissons que le joueur pourra utiliser. Là encore 2 objets de type boisson peuvent être fusionnés pour en obtenir un autre. Utiliser ces objets permettent au joueur d'augmenter son alcoolémie et donc indirectement d'augmenter son attaque.

Liste des objets fusionnables :

Objet 1	Objet 2	Résultat de la fusion
Energydrink	Jagermeister	Jagerbomb
Energydrink	Vodka	Vodkaenergy
X	X	Gnole

X : N'importe quelle boisson sauf Energydrink

Les armes

Enfin, la classe d'arme (Weapon) est lié à l'énumération TypeWeapon qui une fois encore regroupe les différentes armes disponible dans le jeu. Les armes aussi peuvent être fusionnées ensemble pour en obtenir une meilleure. L'utilisation d'une arme permet au héros d'augmenter ses dégâts d'attaque.

Liste des objets fusionnables :

Objet 1	Objet 2	Résultat de la fusion
Stick	Rope	Whip
Stick	Stone	Spears
Rope	Stone	Slingshot
Stick	Knife	Battlespears

V.2 Les clefs

Passons maintenant au dernier type d'objet, il s'agit donc des clefs. Ces dernières sont légèrement différentes puisqu'elles ne sont pas liées à un type énuméré et qu'elles ne peuvent pas être fusionnées. Cette classe hérite quand même de la classe Item puisqu'il nous fallait pouvoir ajouter des clés dans l'inventaire du héros mais elle ne redéfini pas complètement toutes les méthodes.
L'utilisation d'une clef se fait de manière automatique lorsque l'on traverse la porte correspondante. Quant à la génération des clés elle est un peu différente de la génération des objets car il faut autant de clés que de maison, et chaque clé doit correspondre à une porte. Cette méthode de génération est d'avantage détaillé dans le paragraphe de la classe "LockedExit".

VI Les tests

Pour les tests, nous voulons surtout tester les méthodes les plus importantes. Pour cela, nous avons essayé entre autre de vérifier manuellement les commandes et comparer le résultat du test avec notre résultat attendu. Avec une génération aléatoire de la carte, il est important que tout les cas soient bien encadrés et corrects.

Nous ne pouvons pas réellement tester notre carte avec des test, enfin nous ne pensons pas mais il est possible de reproduire plusieurs conditions qui peuvent survenir dans cette génération, ce qui est d'une grande aide pour vérifier tout les cas possibles et ne pas attendre simplement que l'erreur survienne car les chances peuvent être infimes.

VII Conclusion

La réalisation de ce projet a permis de développer notre créativité… mais surtout de développer un vrai projet en java avec un temps limite bien imparti. Il nous a permis de mettre en place toutes nos connaissances dans un sujet très libre où nous avons un peu pu faire ce que nous désirions. La réalisation d'un tel projet n'a pas été sans difficulté, surtout pour adopter un mode de pensée adaptée à la programmation orientée objet et ne pas penser à une méthode de programmation impérative. La carte générée aléatoirement n'a pas été sans soucis non plus car il faut bien gérer les différents cas, qui ne vont pas créer des erreurs à la compilation mais quelques fois dans le jeu.