UNIVERSITY PIERRE ET MARIE CURIE

Projet River City Ransom

MI047 : Composant

Thibaut Fleury - Jérôme Rahault

15/04/2014

Table des matières

I. Architecture générale

Nous nous sommes limités à 8 services afin d'éviter une implémentation trop lourde. Les services Objet, Bloc et Terrain servent à organiser des données utilisées par les Personnages et le service GestionCombat. Personnage est décomposé en deux types de service, à savoir les Joueurs (les personnages contrôlés par les utilisateurs) et les gangsters (contrôlés par la simulation).
Nous avions hésité à garder uniquement Personnage pour gérer n'importe quel type de personnage, mais les services Joueur et Gangster étaient trop différents, d'une part dans leur construction puisqu'un joueur n'a nullement besoin de drop et un gangster ne doit pas avoir d'inventaire ou d'argent. Rallier les deux sous un même service aurait produit trop de champs inutilisés et trop de confusions dans leur implémentation.

Nous nous sommes posés les mêmes questions pour le service Objet qui, suivant notre première conception, devait être non instanciable et implémentable via soit un ObjetMarchandable (ou ObjetConsommable) soit un ObjetEquipable.
ObjetEquipable devait hériter d'un autre service (que nous avons finalement choisi de supprimer) nommé Equipable, et dont Personnage devait aussi hériter. Le problème était, d'une part, que les paramètres à gérer lorsqu'un personnage est équipé/jeté étaient nettement plus complexes que lorsque c'est un objet qui est équipé/jeté et d'autre part, qu'ObjetEquipable héritait à la fois d'Objet et d'Equipable, ce qui sous-entendait un héritage multiple (ce qui après renseignement, nous a été fortement déconseillé par notre chargé de TD). Nous avons finalement conclu sur un seul service Objet avec un booléen *marchandable* et un champs valeur (utilisable par les deux types d'objets) et un champs coût utile uniquement aux objets marchandables (détaillé plus précisément dans la partie suivante).

Enfin, GestionCombat est le cœur de la simulation, c'est le service qui construit le niveau (Terrain), les avatars des Joueurs et Gangsters et organise les interactions entre eux, MoteurJeu ne fait qu'appeler GestionCombat à chaque pas de jeu.

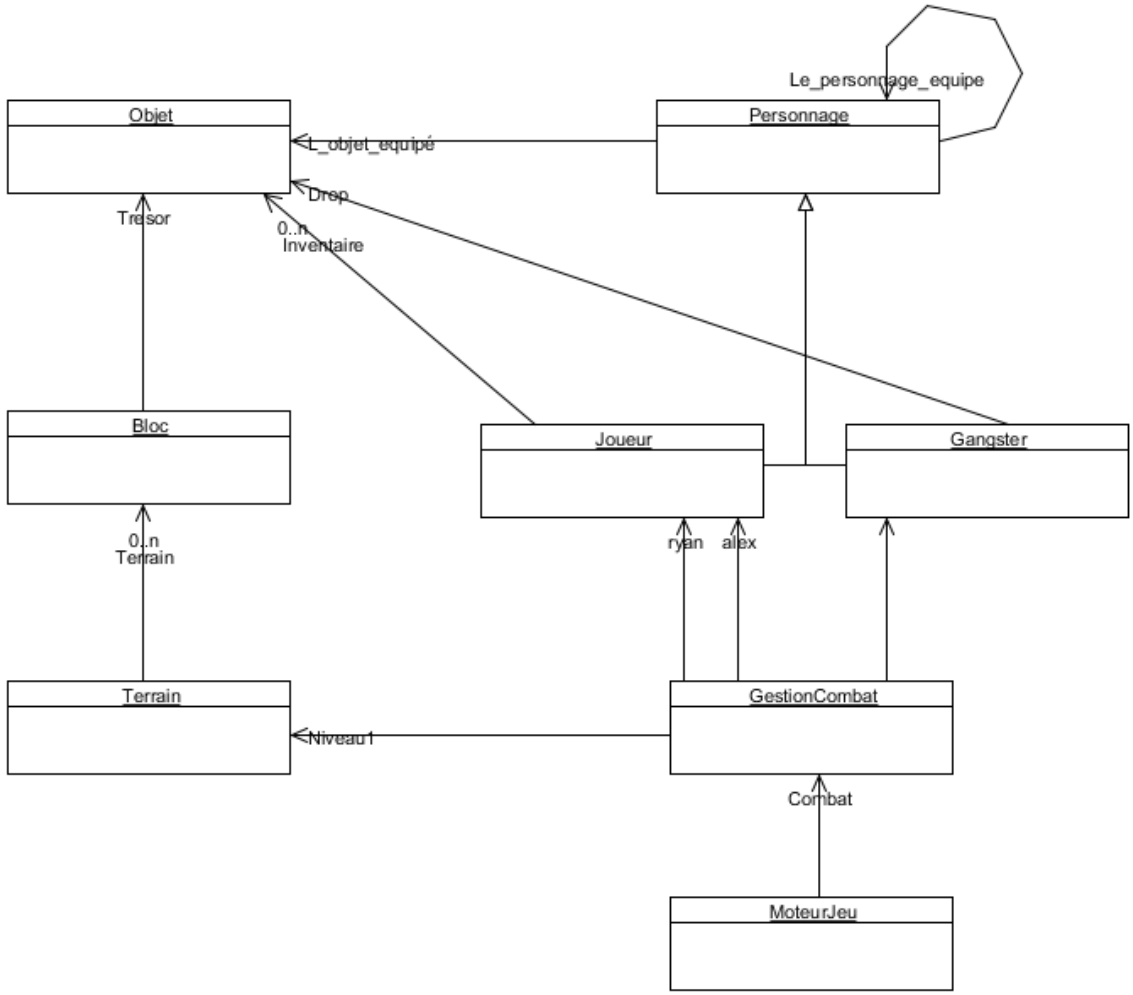

Figure 1 : Diagramme de classe

II. Les services de stockage : Objet, Bloc, Terrain

Le terrain est un service qui possède trois constantes : sa hauteur, sa largeur et sa profondeur. La largeur correspond à l'axe de la rue dans laquelle se déroule le niveau, la profondeur correspond à l'espace entre l'écran et le mur du fond de la rue, et enfin la hauteur qui correspond à l'axe utilisé pour les sauts et les obstacles. Le terrain est un ensemble de blocs possédants chacun des coordonnées.
Nous avons donc utilisé un simple tableau à trois dimensions pour modéliser ces coordonnées. Chaque dimension de ce tableau correspond à l'une des grandeurs (largeur, hauteur, profondeur) données en paramètres. Certains mécanismes de construction du terrain (de l'organisation des blocs) ont dû être spécifiés. Pour chaque croisement des axes de la largeur et de la hauteur, on utilise systématiquement le bloc le plus haut comme hauteur de référence. Ainsi, pour chaque profondeur et chaque largeur, tout ce qui est au-dessus du bloc non-null le plus haut est considéré comme un espace accessible, alors que tout ce qui se trouve en-dessous (blocs null et blocs non-null) est considéré comme un obstacle et est donc inaccessible.

Le bloc est un service très simple qui n'a que deux observateurs. Tout d'abord un observateur indiquant son type : Vide ou Fosse.
Ceci permettant de savoir si le personnage peut marcher sur ce bloc ou non (si un personnage devait marcher sur un fossé, celui-ci serait mort instantanément). Et également un observateur indiquant le trésor éventuel qu'il peut y avoir sur ce bloc. Cet observateur est donc soit null soit de type Objet. Le bloc n'ayant qu'un seul champ Objet pour le trésor, chaque nouveau trésor qui tomberait sur le bloc écraserait le précédent trésor.

L'Objet est un service très générique qui peut contenir n'importe quel type d'objet. En effet il peut exister des objets de type arme qui peuvent être équipés et ne peuvent pas être vendus. A contrario, certains objets sont de type marchandable et peuvent être soit consommés soit vendus. Afin de gérer tous ces cas, le service Objet possède quatre observateurs constants (un objet crée ne sera jamais modifié d'aucune manière). On dispose tout d'abord du type de contenu qui est une enum listant tous les objets disponibles dans le jeu. On a ensuite un champ valeur. Dans le cas d'un objet de type arme, il est utilisé pour connaitre les dégâts qui peuvent être occasionnés par l'arme. Dans le cas d'un objet de type marchandable, il est utilisé pour pouvoir quantifier les effets d'un objet consommable (les pv d'une potion de vie par exemple). Ensuite, marchandable est un booléen qui nous indique si l'objet est marchandable ou équipable (false pour équipable). Et pour finir, on a un champ coût qui est utilisé si l'objet est marchandable (pour une implémentation future des magasins). L'objet qui est donc constant sera simplement échangé d'un personnage à un bloc tout au long de la partie.

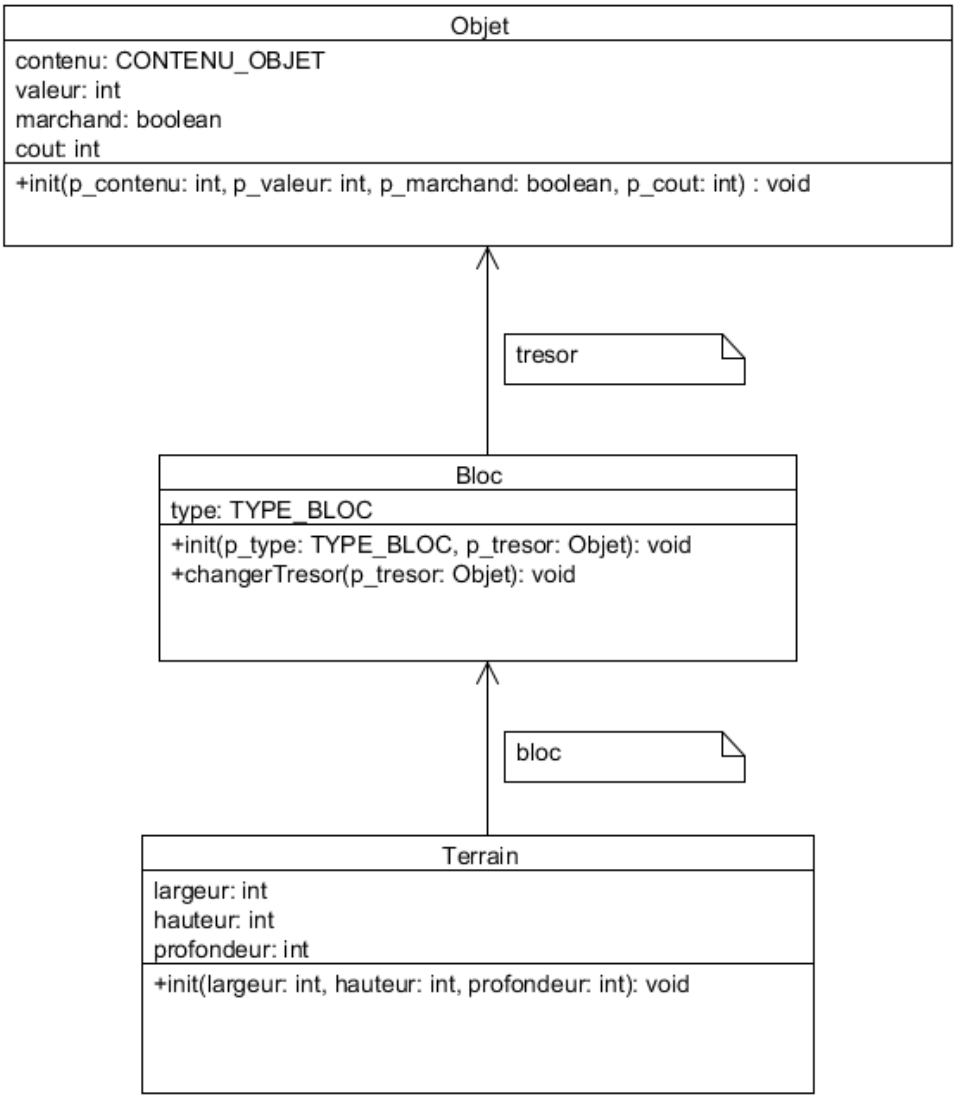

Figure 2 : Diagramme détaillé Objet/Bloc/Terrain

III. Les services de personnage : Personnage, Joueur, Gangster

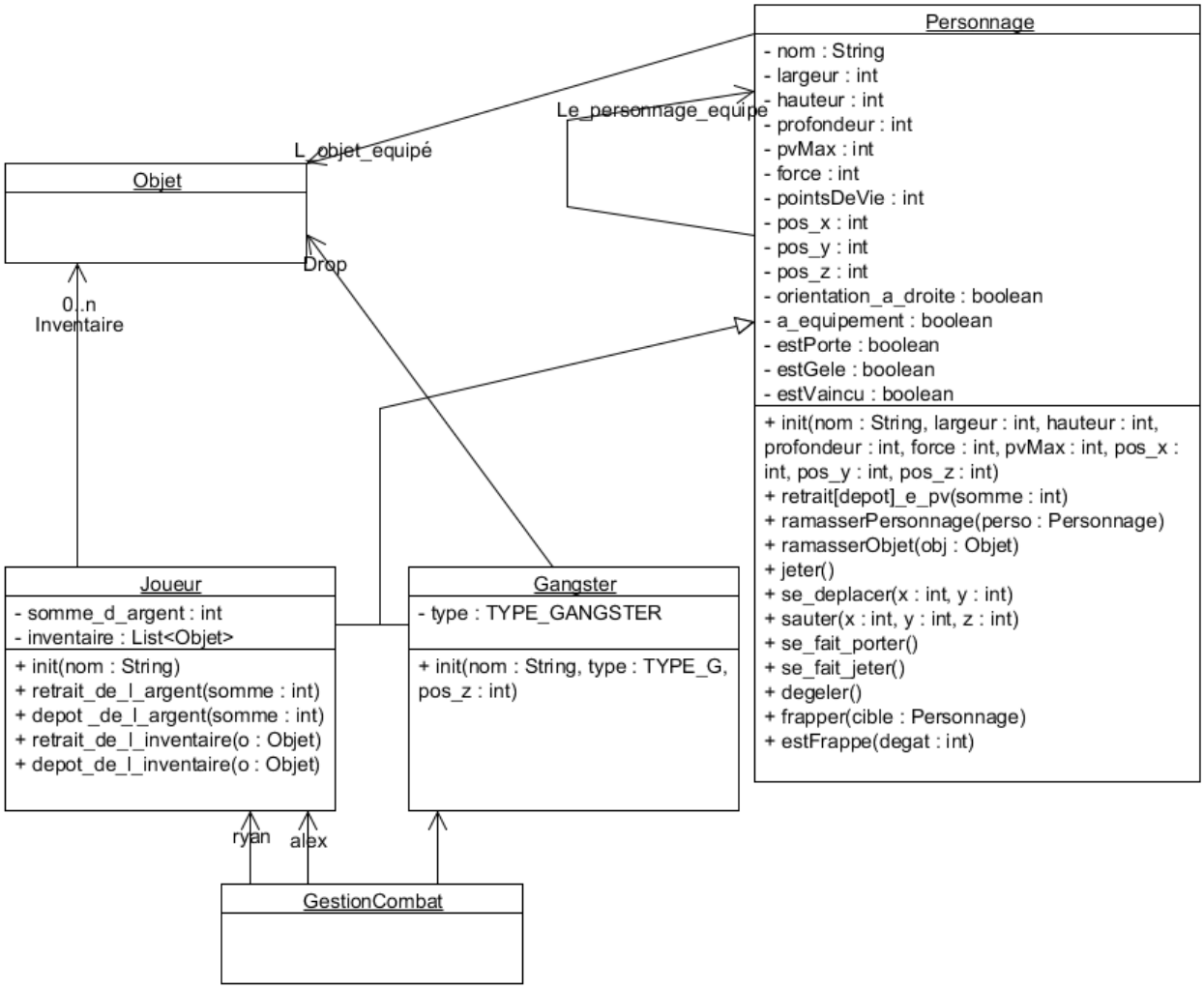

Figure 3 : Diagramme détaillé Personnage/Joueur/Gangster et interactions

C'est le service Personnage qui stock les informations communes aux joueurs et aux gangsters, à savoir leurs caractéristiques (nom, dimensions, positions, points de vie, force et orientation) ainsi que leurs états (est ce qu'ils ont un équipement en main ?/est ce qu'ils sont portés ?/gelés ?/vaincus ?). Les états renvoient tous un booléen à l'exception de l'état de gel (qui renvoie un entier) qui permet de gérer l'état sur un nombre de pas de jeu fixé, chaque pas de jeu décrémentant le gel de 1 (via *degeler()*) jusqu'à atteindre 0 considéré comme l'état non gelé.

Certaines opérations permettent simplement de modifier un attribut précis de Personnage comme la modification de son nombre de pv (*depot_de_pv* s'il mange de la nourriture/*retrait_de_pv* s'il se fait frapper) ou le ramassage (*ramasserPersonnage(p)* => le_personnage_equipe=p / *ramasserObjet(o)* => l_objet_equipe=o). *se_deplacer* modifie ses coordonnées 2D (la hauteur n'est pas modifier) tandis que *sauter* remplace ses coordonnées 3D (hauteur comprise donc).
L'opération *frapper(cible)* s'occupe de gérer le gel de l'agresseur ainsi que le retrait de pv et le gel de

la cible via l'appel de *estFrappe(degat)* sur celle-ci.
Le calcul de dégât se fait de la manière suivante :

- Dégâts = Force de l'agresseur (s'il n'a pas d'équipement) ;
- Dégâts = Force de l'agresseur + valeur de l'objet équipé (s'il est équipé d'un objet) ;
- Dégâts = Force de l'agresseur + force du personnage porté (s'il est équipé d'un personnage).

Ce dernier point n'est pas spécialement logique niveau « role-play » mais beaucoup plus niveau « gameplay », on a pensé que ça pimenterait le jeu d'essayer de s'équiper du personnage le plus fort plus faire plus de dégâts. D'autant plus qu'à deux joueurs, ça peut être un gros avantage de s'équiper de son coéquipier pour accroître les dégâts.

Le joueur possède en plus d'un personnage normal de l'argent ainsi qu'un inventaire (qui est une liste d'objets marchandables). Ses opérations sont uniquement l'ajout et la suppression d'argent et d'objet de son inventaire. Inutile de s'attarder dessus.

Le gangster lui est d'un certain type (prédéfinie par une énumération) et peut avoir un drop (c'est-à-dire un objet qu'il laissera aux joueurs à sa mort). Son type permet à la Gestion du combat de choisir ses caractéristiques à sa création. Tous les gangsters d'un même type ont le même nombre de point de vie, la même force, les même dimensions etc… (Mais pas les mêmes coordonnées bien sûr mais on y reviendra quand on présentera le service GestionCombat)

IV. Les services de gestion de l'animation : MoteurJeu, GestionCombat

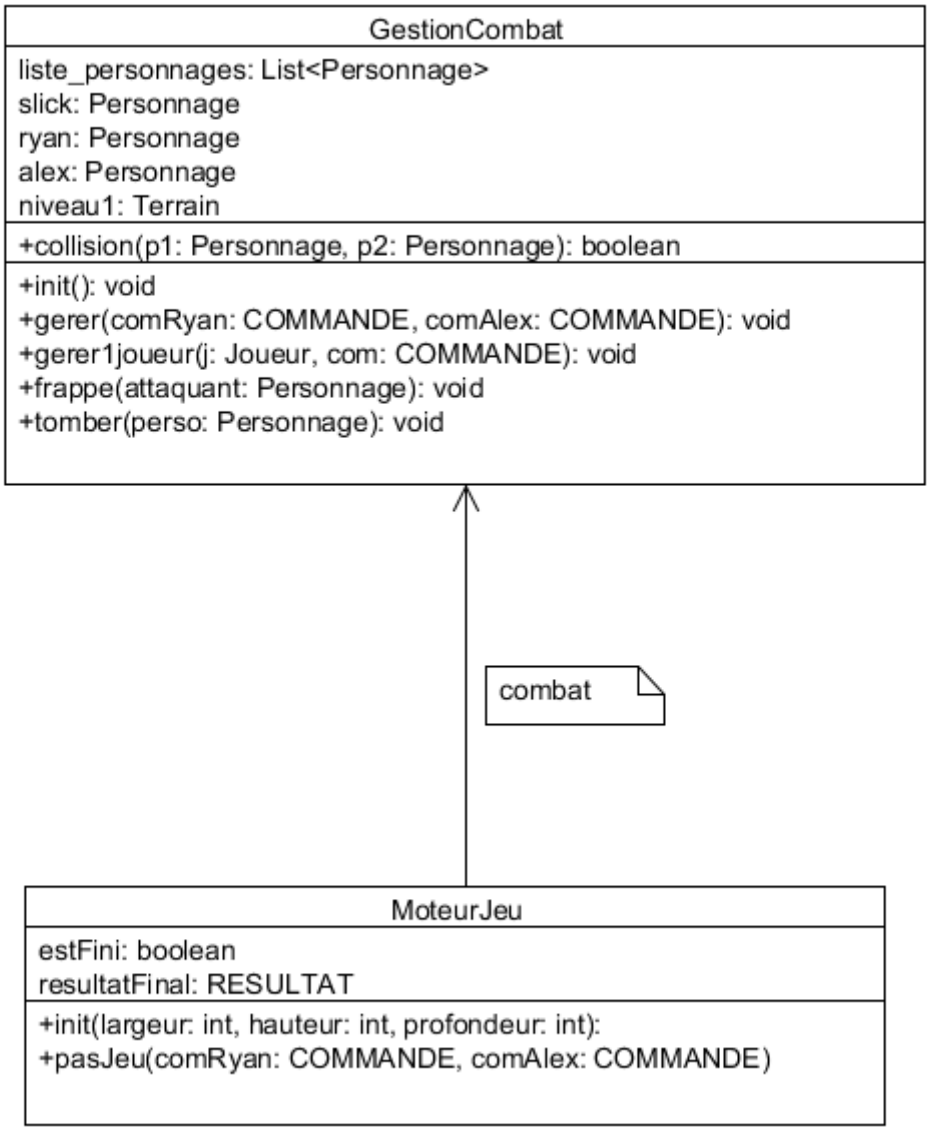

Figure 4 : Diagramme MoteurJeu/GestionCombat

Le service MoteurJeu gère le déroulement temporel (via des pas de jeu) de la simulation tant qu'elle n'est pas terminée (estFini=false). Elle n'est terminé que lorsque les deux joueurs sont morts (défaite : « Game Over ») ou que le boss final Slick est vaincu (victoire : « End Game »). Le resultatFinal ne fait qu'indiquer les survivants, notamment pour savoir qui « remporte » la fille.
Le constructeur ne fait que construire le service GestionCombat avec les mêmes paramètres.
L'opération *pasJeu* est appelée à chaque pas de la simulation et appelle la fonction gérer de GestionCombat avec les commandes des deux joueurs.

Le service GestionCombat est le cœur de la simulation, il crée le terrain et les protagonistes du jeu et s'occupe des interactions entre eux.
Par exemple un personnage a des coordonnées mais n'a pas de vision du Terrain, il ne peut donc pas

savoir lorsqu'il en sort, lorsqu'il est bloqué par un mur, lorsqu'il marche sur un bloc fosse ou lorsqu'il ne touche pas sol et doit donc tomber sous l'effet écrasant de la gravité. C'est donc la gestion du combat qui permet, en plus du combat à proprement parler (frappe/ramasse/jet/saut), d'effectuer ces actions de déplacement tout en gardant une cohérence vis-à-vis du niveau parcouru.
init appelle le constructeur de Terrain avec les données passées en argument pour créer le niveau1, deux fois le constructeur de Joueur pour créer Alex et Ryan avec des inventaires vides et 20 d'argent chacun et un certain nombre de fois (nous avons choisi quatre pour simplifier) le constructeur de Gangster avec pour chacun un type et un drop aléatoire. Le constructeur de gangster n'étant pas assez précis, nous avons choisi d'utiliser le constructeur de Personnage pour Slick qui n'a d'ailleurs pas besoin de drop puisque la partie se termine à sa mort, on a pu lui passer une force et des pv beaucoup plus grand que chaque type de gangster ainsi que de grande dimension pour un boss final imposant.
Les positions initiales des joueurs sont à une case du bord gauche du terrain et Slick est à une case du bord droit (afin que les joueurs aient à traverser tout le terrain pour l'affronter), les positions des gangsters sont parfaitement aléatoires.
Tous les personnages créés sont ajoutés à la liste de personnage afin de pouvoir tous les parcourir facilement à l'occasion.

L'opération principale de ce service est *gerer(comRyan,comAlex)* qui s'occupe de faire agir la gravité en appelant *tomber(perso)* sur chaque personnage qui déplace un personnage d'une case vers le bas s'il ne touche pas sol au pas courant. *gerer* tue les personnages qui marchent sur des blocs de type fosse et supprime les gangsters vaincus de la liste de personnage et gèle ceux qui sont portés. Elle délègue son travail à *gerer1Joueur(ryan,comRyan)* si ryan n'est ni porté, ni vaincu et à *gerer1Joueur(alex,comAlex)* avec les mêmes conditions pour alex.

gerer1Joueur(joueur,com) produit une action sur le joueur en fonction de la commande reçue avec l'aide d'un switch sur toute l'énumération de COMMANDE.
Si la commande est GAUCHE, DROITE, HAUT, BAS, on appelle *se_deplacer* sur le joueur pour effectuer le déplacement associé, en vérifiant qu'il ne sort pas du terrain, idem pour les SAUT_SUR_PLACE, SAUT_GAUCHE, SAUT_DROIT avec la fonction *sauter*.
Lorsque c'est la commande PREND qui est choisie, si le joueur n'est pas déjà équipé, on regarde s'il y a un objet ou un personnage gelé sur la case du joueur et on appelle *ramasserObjet* ou *ramasserPersonnage* en conséquence. Un objet ramassé disparait bien sûr du terrain et est ajouté à l'inventaire s'il est marchandable ou équipé autrement, un personnage ramassé est continuellement gelé (jusqu'à ce qu'il soit jeté).
Lorsque c'est la commande JET qui est effectuée, on place l'objet/le personnage précédemment équipé sur la case adjacente au joueur et on prévient le personnage qu'il n'est plus porté. On utilise le booléen donnant l'orientation du joueur pour choisir si l'élément jeté doit être placé à sa gauche ou à sa droite. Le jet d'un personnage ou d'un objet ne blesse personne, c'est plus un lâché qu'un jet, nous n'avons pas eu le temps d'implémenter plus précisément ce mécanisme.
La commande FRAPPE est sans doute la plus complexe, elle met en scène deux personnages, un qui effectue un coup (qui *frappe*) et un autre qui reçoit le coup (qui *estFrappé*). Elle nécessite notamment de savoir lorsque deux personnages sont en collisions, c'est-à-dire lorsque qu'ils ont au moins un pixel en commun, pour cela on fait appel à la méthode *collision(agresseur,cible)* (l'ordre des paramètres n'a pas d'importance). L'agresseur est donc le joueur qui utilise la commande FRAPPE et

on effectue le test sur chacun des personnages de la liste, plusieurs personnages en même temps pouvant être frappés (du moment qu'ils sont en collision avec l'agresseur). Les personnages touchés sont à la fois gelés, perdent des points de vie (calcul expliqué dans la présentation du service Personnage) et sont aussi repoussés d'une case suivant l'orientation de l'agresseur (à droite s'il est orienté vers la droite et respectivement pour la gauche). Les personnages frappés perdent aussi l'équipement qu'ils portent éventuellement, cette perte équivaut à un jet de leur part (puisque notre jet actuel ne blesse pas).

Conclusion

Ce projet nous a demandé beaucoup d'effort de conception, j'ai passé plusieurs heures avec mon binôme à discuter de la façon dont nous allions spécifier les services. Partir sur de bonnes bases était essentiel puisque les contrats et l'implémentation allaient découler des spécifications de manière plus ou moins fluide en fonction de leur qualité. Et plus les spécifications avaient été bien choisies et moins nous avions besoin de revenir dessus en même temps que nous implémentions. Après avoir écouté les conseils de notre chargé de TD, nous avons minimisé le nombre de service pour en avoir huit (au lieu de onze dans notre conception la plus lourde). Nous sommes plutôt satisfait de nos choix dans l'ensemble même si j'avoue que nous avons dû revoir quelques détails que nous avions omis (par exemple les deux méthodes *ramasserPersonnage* et *ramasserObjet* qui étaient seulement *ramasser* sans paramètre). Nous avons un peu pesté contre la redondance du travail demandé bien qu'avec du recul la réalisation du projet nous a appris pas mal de chose sur la conception par contrat et sur les couvertures de test. C'est peut être la première fois que l'on accorde plus d'importance aux objectifs de test (et aux tests unitaires) qu'à l'implémentation d'une interface graphique et donc à la réalisation d'un projet abouti.