Concept Objet – M2 Novembre 2019

Projet *Sangsational Simulator*

BOMY Clara – CHRISTIAENS Mathilde

ISEN
ALL IS DIGITAL!
LILLE
yncréa

TABLE DES MATIERES

INTRODUCTION

Dans le cadre de notre cours de Concept Objet, nous avons réalisé *Sangsational Simulator*, un projet de simulation discrète à pas de temps constant.

Ce projet de simulation, inspiré de l'univers de *Game Of Thrones*, met en scène les célèbres familles *Stark, Lannister, Targaryen* et les *Sauvageons*, sans oublier les redoutables *Marcheurs Blancs* – éléments perturbateurs pouvant compromettre la survie des factions – sur les terres de *Westeros*. Doté de points d'attaque, de vie et d'expérience, chaque individu évolue sur une carte composée d'obstacles et peut être amené à aider voire à combattre jusque mort s'ensuive les autres personnages. Pour se protéger, chaque population – hormis les Marcheurs Blancs – possède une safezone qui leur est propre. En théorie, la simulation se termine lorsqu'il ne reste uniquement que des populations alliées sur la carte. Néanmoins, la simulation pouvant s'éterniser, un mode de calcul basé sur les caractéristiques finales des populations encore en vie nous permet de déterminer la population gagnante.

Pour faciliter la gestion de ce projet, nous avons fait le choix de travailler à deux sur ce projet. Après hésitation avec le Python, le langage de programmation utilisé est le Java, langage sur lequel nous étions toutes deux le plus à l'aise.

1 LES REGLES DE LA SIMULATION

1.1 LA CARTE

La carte est le milieu dans lequel évolue les personnages de la simulation. Le milieu est discrétisé en n^2 cases et comporte plusieurs obstacles, ajoutés aléatoirement (hors safezones) lors de sa création. Notons que chaque case ne peut contenir au maximum qu'un seul individu ou obstacle.

Pour une simulation, l'affichage de la carte se présente comme ci-dessous :

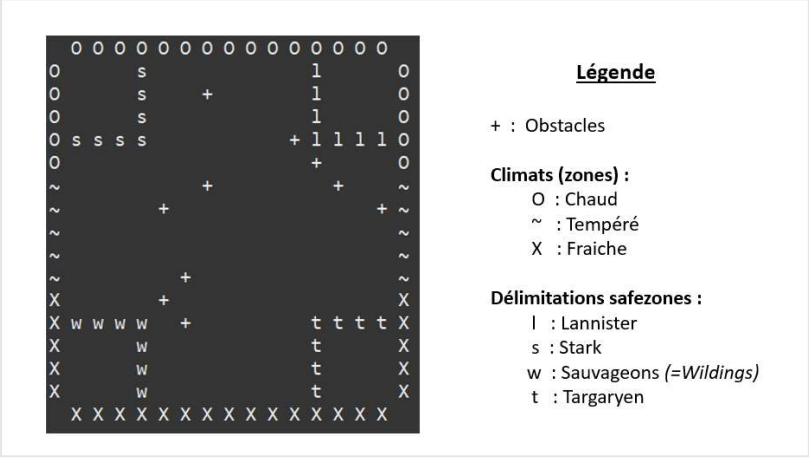

Figure 1 : Exemple de génération de carte

1.1.1 Les safezones

Aux quatre coins de la carte, nous trouvons quatre zones appelées *safezones*. Chaque être humain possède une unique safezone, lui permettant d'y récupérer 10 points d'endurance par case traversée. Ces safezones sont un véritable atout pour les populations puisqu'aucun combat entre humains ne peut y avoir lieu. Lorsque la population propriétaire d'une safezone est éradiquée, celle-ci est détruite.

L'affectation des safezones se fait en début de simulation de manière aléatoire. D'un point de vue graphique, chaque safezone est délimitée par des symboles correspondant à la première lettre en minuscule de la faction propriétaire. Il est ainsi plus simple pour l'utilisateur de repérer les factions propriétaires des safezones.

1.1.2 Les climats

La carte se divise également en plusieurs zones climatiques (zone chaude, tempérée, fraiche ou glaciale).

Pour leur répartition sur la carte, nous nous sommes basées sur la série - c'est pourquoi la zone chaude est au nord et la zone fraiche, au sud. Evidemment, la zone tempérée fait la liaison entre ces deux climats.

Les climats inhospitaliers altèrent la vie des êtres humains qui n'y sont pas habitués *(safezone non présente dans la zone climatique considérée)* : 2 points de dégâts par case traversée.

La zone glaciale est un microclimat spécial, généré par les marcheurs blancs *(=White Walkers)* – ce cas sera expliqué ultérieurement.

1.2 LES PERSONNAGES

Les personnages de notre simulation comportent cinq populations différentes que l'on peut diviser en deux sous-catégories : les êtres humains et les marcheurs blancs *(=White Walkers)*. Chaque individu se déplace sur la carte et peut rencontrer et attaquer d'autres personnages. Ils se caractérisent par leurs points de vie, leurs points d'attaque et leur portée de déplacement relative à la taille de la carte. De plus, chaque individu possède un niveau de chance et de malchance - qui lui permettra de réussir ses actions avec plus ou moins de succès grâce à un mécanisme de lancer de dés.

Le positionnement des personnages se fait de manière aléatoire en début de simulation (hors marcheurs blancs). D'un point de vue graphique, les personnages sont symbolisés par la première lettre en majuscule de leur faction.

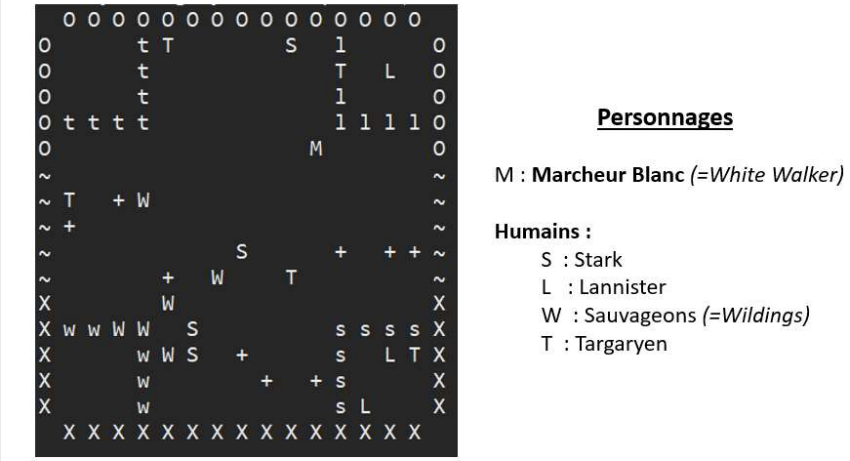

Figure 2 : Représentation des personnages

1.2.1 Les êtres humains

Les *Lannister*, les *Stark*, les *Targaryen* et les *Sauvageons (=Wildings)* représentent les quatre populations humaines évoluant sur la carte. Ces factions sont divisées en deux catégories :

- Les familles du Nord *(=Northerner)* dans lesquels on trouve les *Stark* et les *Sauvageons*.
- Les familles du Sud *(=Southerner)* regroupant les *Lannister* et les *Targaryen*.

Les êtres humains se caractérisent tous par leur nom, leur niveau, leur expérience, leur endurance et leur safezone. Les valeurs attribuées à ces caractéristiques varient en fonction de la faction considérée :

Caractéristiques des humains Vie de départ : 100 Endurance min avant de chercher safezone : 20 Nombre de tours pouvant être passés sans endurance : 2 Amélioration des stats par niveau : 5 Expérience nécessaire pour gagner un niveau : 20	**Caractéristiques des Lannister** Palier de succès critique : 65 Palier d'échec : 5 **Caractéristiques des Targaryen** Palier de succès critique : 95 Palier d'échec : 30
Caractéristiques des familles du Sud Endurance max : 100 Dégâts initiaux : 20 Dégâts max : 120	**Caractéristiques des Stark** Palier de succès critique : 65 Palier d'échec : 15
Caractéristiques des familles du Nord Endurance max : 80 Dégâts initiaux : 30 Dégâts max : 100	**Caractéristiques des Sauvageons** Palier de succès critique : 80 Palier d'échec : 30

Figure 3 : Caractéristiques des êtres humains

Chaque être humain commence la simulation au niveau 1. A chaque case traversée et en survivant aux combats, il amasse de l'expérience jusqu'à en avoir 20 points. Une fois ce palier atteint, l'être humain gagne un niveau : son expérience retombe alors à 0 et sa vie et ses points d'attaque augmentent de 5 points.

1.2.2 Les marcheurs blancs

Les marcheurs blancs sont les derniers personnages que l'on peut rencontrer dans la simulation. Opposés aux êtres humains en général, ils envahissent progressivement la carte et imposent leur propre climat, le froid glacial, sur les cases qu'ils traversent pour quatre tours. Ce climat spécial impacte tous les êtres humains en leur retirant 4 points de vie.

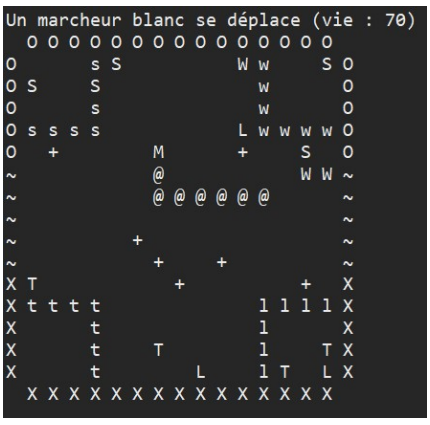

Figure 4 : Illustration du climat glacial (symbole « @ »)

Contrairement aux êtres humains, les marcheurs blancs ne possèdent pas de système de niveaux car ces créatures sont d'office extrêmement puissantes (niveaux de vie et de dégâts importants). En outre, les marcheurs blancs n'ont pas de safezone et ne respectent pas les zones de non-agression des safezones humaines.

1.3 LES DEPLACEMENTS DES PERSONNAGES

A chaque tour de simulation, les personnages peuvent se déplacer un par un sur la carte selon les quatre points cardinaux de l'espace et leurs compositions : *Nord, Nord-Est, Est, Sud-Est, Sud, Sud-Ouest, Ouest et Nord-Ouest*. Leur ordre de prise en charge est déterminé aléatoirement à chaque étape.

Pour se déplacer, un personnage commence par évaluer le nombre de pas qu'il peut effectuer (~portée) via un jet de dé. Il fait alors la liste des directions dégagées à une case de distance dans les limites de la carte puis en sélectionne une au hasard. Si la liste est vide, le personnage se contente de rencontrer les potentiels autres personnages autour de lui. Le personnage se déplace ensuite de case en case dans la limite de sa portée, tant qu'il ne rencontre ni personnage dans les environs, ni obstacle sur sa route.

A chaque case quittée, un marcheur blanc instaure simplement son climat glacial. Il est infatigable (pas de perte d'endurance) et le fait d'appliquer son climat le protège des autres climats. Un humain, quant à lui :

- Gagne un point d'expérience
- Peut perdre un point d'endurance hors safezone ou en gagner 10 dans la safezone de sa faction
- Peut gagner un point de vie ou en perdre 2 à 4 selon sa résistance au climat

Contrairement à un marcheur blanc, un humain peut donc mourir sur son chemin ou tomber à court d'énergie - ce qui le stoppe dans son déplacement.

Cas particuliers :

- Si un humain n'a plus d'endurance, il ne se déplace pas et se contente de rencontrer les personnages autour de lui en attendant un allié (humain de même région ou faction). S'il en rencontre un dans les deux tours, il récupère de l'énergie et peut à nouveau se déplacer ; sinon, il meurt d'épuisement.

- Un humain avec peu d'énergie se concentre sur les trois directions le rapprochant de sa safezone. Il tente ainsi d'accéder le plus rapidement possible à sa safezone.

1.4 LES RENCONTRES ENTRE PERSONNAGES

Au cours de la simulation, les différents personnages peuvent être amenés à se rencontrer. Une rencontre se fait entre deux personnages. Avec nos cinq types de population, plusieurs cas peuvent se produire :

- **Rencontre de même région :** si un des deux personnages est à court d'endurance, l'autre lui donne la moitié de la sienne. Sinon, ils récupèrent tous deux un quart de leur vie maximale.

- **Rencontre de régions opposées :** si un des deux personnages est à court d'endurance, l'autre l'achève. Sinon, un combat se lance entre les deux personnages et le gagnant récupère l'expérience du perdant en tenant compte de son niveau.

- **Rencontre entre humain et marcheur blanc :** elle entraine un combat entre les personnages. Si l'humain gagne, il récupère 100 points d'expérience et 25 points de vie.

Cas particuliers :

- Rencontre de même famille : elle est similaire à la rencontre de même région. Néanmoins, dans le cas où les personnages récupèrent chacun des points de vie, ils gagnent en plus la portée restante convertie en points d'expérience.

- Rencontre entre marcheurs blancs : nous ne comprenons pas ce qu'ils se racontent mais leurs statistiques n'évoluent pas.

1.5 LES COMBATS

Les « mauvaises » rencontres peuvent amener à des combats à mort au « tour par tour ». Chacun leur tour, les personnages déterminent la réussite de leur attaque via un lancer de dé aboutissant soit à un échec, à un succès ou à un succès critique :

- Lors d'un succès critique, les êtres humains lancent une super attaque pouvant permettre de retourner la situation là où les marcheurs blancs réalisent une attaque classique. Chaque faction possède sa propre super attaque :
  - ✓ Un Targaryen tue instantanément l'adversaire avec sa super attaque.
  - ✓ Un Lannister donne 1/6 de ses points de vie actuels pour infliger un nombre de dégâts équivalant à la moitié de ses points de vie maximum.
  - ✓ Un Stark inflige un nombre de dégâts équivalant au tiers de ses points de vie maximum.
  - ✓ Un Sauvageon gagne 10 points de vie et multiplie la puissance de sa prochaine attaque par 1,5.

- Lors d'un succès, les dégâts infligés à l'adversaire correspondent aux points d'attaque.
- Lors d'un échec, l'attaque rate donc aucun dégât n'est infligé.

Le combat se termine lorsqu'un des deux personnages meurt.

1.6 LES CONDITIONS DE FIN

Une simulation peut se finir de différentes manières :

- Le nombre maximum de tour est atteint – auquel cas, on détermine le potentiel futur vainqueur en se basant sur le nombre de survivants de chaque région, puis de chaque faction. En cas d'égalité, nous nous servons du nombre de personnages tués par les factions considérées pour établir un classement. Dans le cas où il reste moins de deux fois plus d'humains que de marcheurs blancs, ces derniers sont considérés vainqueurs.

- Il ne reste plus que des marcheurs blancs : ils dominent Westeros.

- Il ne reste plus que des personnes d'une région (famille du Nord ou famille du Sud) – la famille avec le plus de représentants gagne le pouvoir. En cas d'égalité, nous nous servons du nombre de personnages tués par les factions considérées pour établir un classement.

2 DEROULEMENT D'UNE SIMULATION

2.1 FONCTIONNEMENT GLOBAL

Au lancement du programme, l'utilisateur a le choix entre trois scénarios prédéfinis :

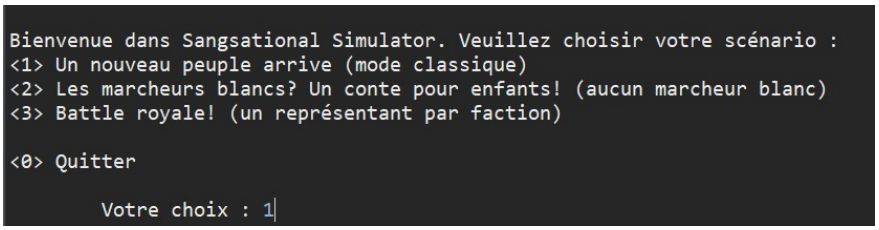

Figure 5 : Lancement du programme

- Un mode classique, fidèle à la série, avec une carte de 12 cases de côtés, des safezones de 3 cases de côtés et 4 individus par famille humaine. Les marcheurs blancs arrivent à partir du 3ème tour puis tous les 4 tours.

- Un mode sans marcheur blanc avec une carte de 12 cases de côtés, des safezones de 4 cases de côtés et 5 individus par famille humaine.

- Un mode « Battle Royale » avec une carte de 9 cases de côtés, des safezones de 2 cases de côtés et un individu pour chaque population.

Il lui suffit alors de se laisser guider jusqu'à la fin de la simulation :

- Tout d'abord l'utilisateur voit la génération de la carte (positionnement des obstacles et attribution des safezones) et le positionnement des personnages. Il peut aussi prendre connaissance des caractéristiques communes aux différentes populations (avec héritage).

- Ensuite, la simulation se déroule au tour par tour – voir détail au point 2.2.

- A la fin d'une simulation, l'utilisateur prend connaissance du gagnant et voit les statistiques de la simulation. Il peut retrouver l'ensemble des événements notables qui sont répertoriés dans un fichier de logs.

Il revient ensuite au menu d'où il peut quitter le programme ou lancer une nouvelle simulation.

2.2 FONCTIONNEMENT DE LA BOUCLE PRINCIPALE

La simulation se déroule sur un nombre de tours compris entre 1 et le maximum défini par le scénario :

- A chaque nouveau tour, on commence par déterminer aléatoirement l'ordre de prise en charge des personnages.

- Les personnages encore en vie se déplacent ensuite un par un sur la carte tandis que ceux qui sont morts sont retirés de la liste. Si ce tour correspond à l'arrivée d'un marcheur blanc, celui-ci est ajouté à la liste de personnages et apparait alors sur la carte – à condition qu'il y reste de la place.

- A chaque fin de tour, on vérifie les conditions de fin : dernier tour atteint ou conditions énoncées au point 1.6 remplies.

2.3 DEROULE D'UN TOUR POUR UN PERSONNAGE

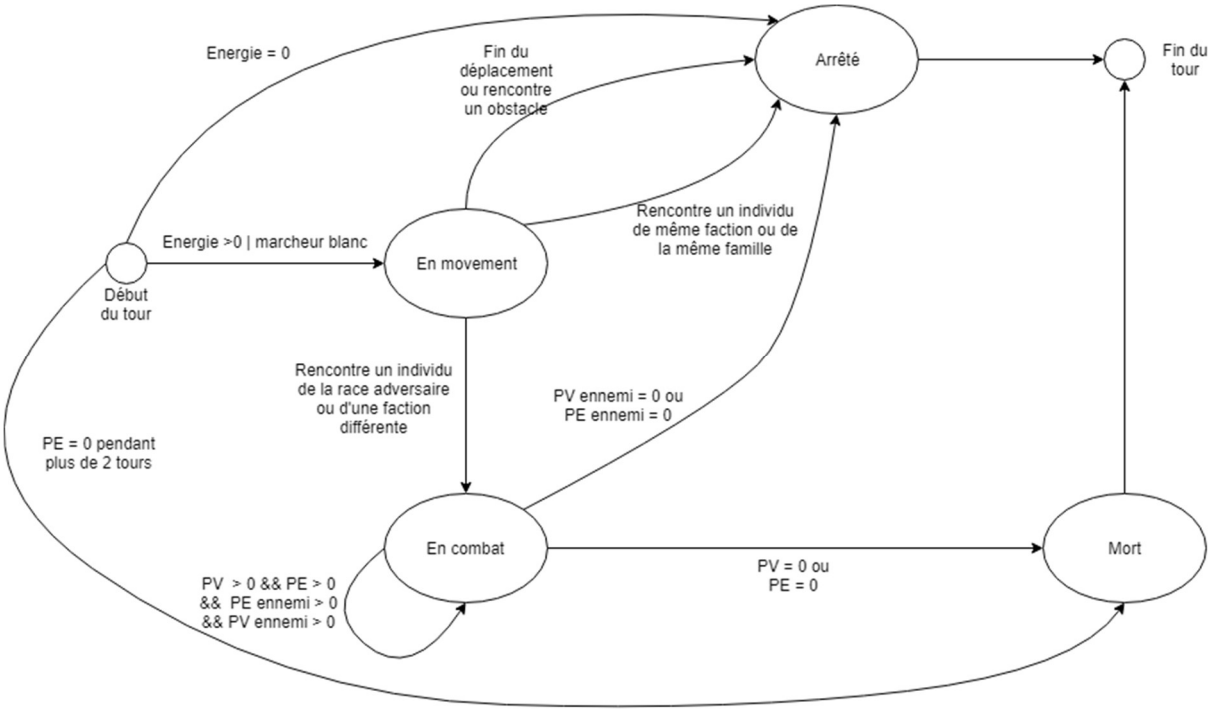

Figure 6 - Diagramme d'Etats-transition du déroulement d'un tour d'un personnage

3 CONCEPTION

3.1 ARCHITECTURE DU PROJET

Pour structurer notre projet, nous avons regroupé de façon logique nos classes dans des packages :

Nom du package	Contenu et utilité
app	Point d'entrée du programme et code relatif au menu de la simulation
character	Arborescence de classes relatives aux personnages
factions	Enumérations : noms des factions et de leurs représentants
gameplay	Ensemble de classes utilitaires nécessaires au fonctionnement de la simulation en général
map	Ensemble de classes gérant la carte et ses différents éléments (hors personnages)

Ces packages sont composés de la façon suivante :

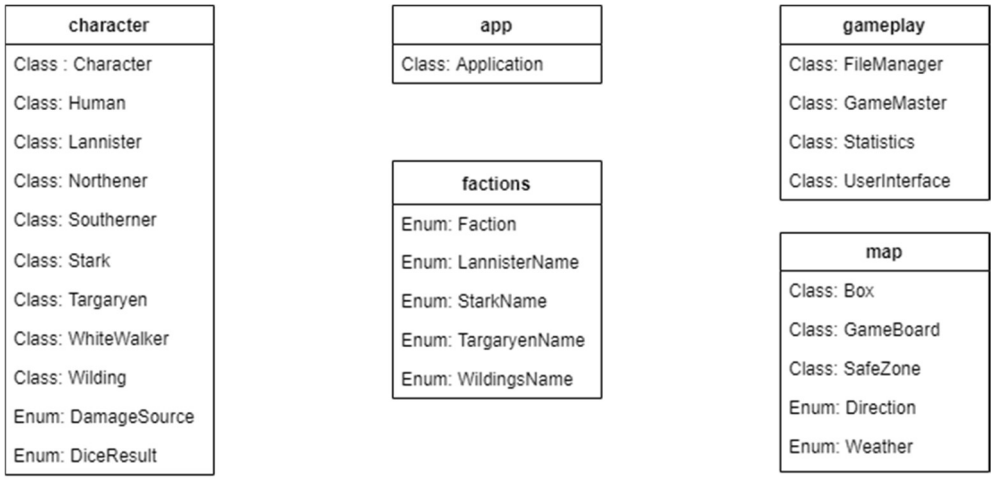

Figure 7 – Contenu des différents packages

3.2 DIAGRAMME DE CLASSES

Le diagramme de classes complet du projet est disponible en annexe.

3.3 INTEGRATION DES CONCEPTS OBJETS

3.3.1 Héritage

La notion d'héritage est exclusivement utilisée dans le package *character*. En effet, les classes finales *(Lannister, Stark, Targaryen, Wilding et White Walker)* sont enrichies par héritages successifs de classes abstraites, plus générales.

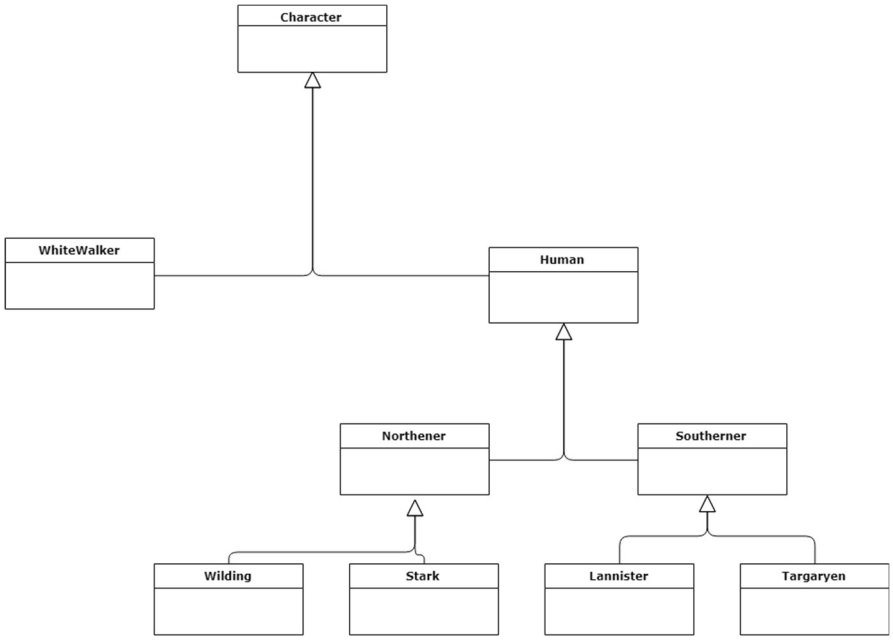

Figure 8 – Diagramme d'héritage de la classe mère Character et de ses classes filles

L'héritage est construit de la façon suivante :

Classe *Character* (classe de base, abstraite) : un personnage se caractérise par sa position sur la carte et sa capacité à s'y déplacer. Un personnage peut rencontrer d'autres personnages spécialisés et les combattre : il a donc un niveau de vie et il inflige des dégâts. Lorsqu'il tombe à court de vie, il meurt. La réussite de ses actions est influencée par sa chance au lancer de dé - ce qui rendrait superflu, en raison de l'héritage mis en œuvre, l'utilisation d'une classe *Random*.

Attributs	Méthodes
Position sur la carte *(westeros, currentBox, MAX_RANGE)*	Se déplacer *(move et méthodes privées associées)* et faire des rencontres *(meet)*
Capacités physiques *(life, maxLife, power, maxPower)*	Attaquer *(attack)* et Mourir *(death)*
Chance *(THRESHOLD_MAX, criticalSuccessThreshold, failureThreshold)*	Lancer de dés *(rollDice)*

Classe *WhiteWalker* (hérite de *Character*, classe instanciable) : cette classe définit les niveaux de vie, de puissance et de chance propres aux marcheurs blancs. Elle peut donc définir le fonctionnement de *meet*, de *attack* et d'une méthode relative à *move* (*movmentConsequences*).

Classe *Human* (hérite de *Character*, classe abstraite) : cette classe définit les niveaux de vie, de puissance et de chance propres aux êtres humains. Elle peut donc définir le fonctionnement de *meet*, de *attack* et d'une méthode relative à *move* (*movmentConsequences*). Les êtres humains sont ainsi des sortes de personnages qui possèdent en plus un nom, un niveau et de l'expérience mais aussi une safezone et de l'endurance ainsi qu'une attaque spéciale, propre à chaque famille.

Attributs	Méthodes
Identification *(name)*	Désignation *(getFullName)*
Expérience *(level, xp, XP_THRESHOLD, GAIN_BY_LEVEL)*	Gagner un niveau (newLevel)
Endurance *(stamina, LOW_STAMINA, maxStamina)*	Récupération en safezone *(safeZoneDirection)*
	Attaque spéciale *(superAttack)*

Classes *Northerner* et *Southerner* (héritent de *Human*, classes abstraites) : ces classes définissent les niveaux d'endurance et de dégâts maximum que peuvent infliger les catégories d'êtres humains. Ces capacités physiques différencient les familles du Nord (=*Northerner*) et les familles du Sud (=*Southerner*).

Classes *Stark*, *Wilding* (héritent de *Northerner*, classes instanciables) et classes *Lannister* et *Targaryen* (héritent de *Southerner*, classes instanciables) : ces classes définissent les niveaux de chance ainsi que les attaques spéciales de chaque famille.

3.3.2 Encapsulation

Pour l'ensemble des classes, les attributs sont tous soit en protected pour être utilisé par héritage, soit en private. De fait, pour y accéder en dehors de la classe ou des classes filles (cas protected), nous utilisons des getters et des setters.

Les seules méthodes publiques sont soit des getters et setters, soit des méthodes « importantes » telles que move de la classe *Character* ou encore *runSimulation* de la classe *GameMaster*, etc. Ainsi, la plupart des méthodes nécessaires au fonctionnement de la classe sont en private ou en protected (héritage) et appelées dans des méthodes publiques. Par exemple, nous utilisons les méthodes privées liées au déplacement d'un personnage (*makeStep*, *getPossibleDirections*, *diagoRight*, etc.) dans la méthode publique move de *Character*.

3.3.3 Polymorphisme

Dans ce projet, nous avons utilisé plusieurs types de polymorphisme. En effet, nous y avons utilisé le polymorphisme dit d'héritage qui consiste à *override* (=redéfinir) dans les classes filles non abstraites une fonction déjà déclarée dans la classe mère, mais aussi un polymorphisme paramétrique qui signifie la redéfinition d'une fonction de même nom qui se comporte différemment selon le type de paramètres.

Une de nos fonctions clefs, la fonction *meet* est une fonction à double polymorphisme : elle est à la fois déclarée abstraite dans la classe mère *Character* et est également déclarée deux fois avec deux types de paramètres différents, elle se comporte différemment suivant le type de personnage qui effectue la rencontre et quel type de personnage il rencontre. Pour illustrer nos propos, il faut s'imaginer qu'un humain n'aura pas la même réaction en rencontrant un autre humain qu'en rencontrant un marcheur blanc. De même, un marcheur blanc ne réagira pas de la même manière face à un de ses congénères que face à un humain. Cette double distinction a également été nécessaire car les marcheurs blancs et les humains n'ont pas les mêmes caractéristiques : un humain peut gagner des points d'expériences et des niveaux alors qu'un marcheur blanc ne peut pas en gagner.

Pour les déplacements, la fonction *movmentConsequences*, déclarée abstraite dans *Character*, est également définie de deux manières différentes dans les classes filles *Human* et *WhiteWalker*. Ce polymorphisme permet de distinguer l'impact d'un pas pour ces deux classes : les notions d'endurance et d'expérience absentes de la classe *WhiteWalker*, et réciproquement, la gestion du climat glacial, spécifique aux marcheurs blancs.

Une autre fonction utilisée par les humains pour laquelle nous utilisons un polymorphisme d'héritage est la fonction *superAttack* uniquement disponible pour les humains : la fonction diffère en fonction de

chaque famille (classe fille) c'est donc pourquoi nous avons dû utiliser ce type de polymorphisme. La particularité de cette fonction explique également un autre polymorphisme d'héritage existant pour la fonction d'attaque. Comme les super attaques ne sont utilisables uniquement par les humains, il a fallu différencier la fonction lorsqu'elle est utilisée par un marcheur blanc : ils ne peuvent donc pas obtenir de succès critique.

Dans un autre package, la méthode *displayConsole* de *UserInterface* bénéficie d'un polymorphisme paramétrique particulier : sa version la plus « simple » ne demande qu'un paramètre et l'ajout successif de paramètres enrichissent ses fonctionnalités. En effet, chaque redéfinition appelle la version précédente en y ajoutant la fonctionnalité liée au paramètre ajouté. Il en va de même pour les méthodes *userChoice* de la classe *UserInterface* et *writeToLogFile* de la classe *FileManager*.

3.3.4 Patrons de conception

Pour les classes *GameMaster* et *GameBoard*, nous avons mis en place un *Design Pattern Singleton* permettant de limiter l'instanciation des classes à un objet car une simulation ne nécessite qu'une seule carte et qu'un seul gestionnaire de simulation.

La conception de ces singletons n'a pas été si simple que ça et nous avons eu quelques difficultés à les mettre en place. En effet, pour pouvoir lancer plusieurs simulations à la suite, il a fallu créer des méthodes d'initialisation et les différencier du constructeur car l'unique instance est utilisée pour toutes les simulations lancées. Avant de séparer l'initialisation, la taille de la carte, des safezones et d'autres éléments restaient fixés sur les valeurs du premier scénario lancé.

3.4 LES CLASSES UTILITAIRES

La classe *FileManager* regroupe l'ensemble des méthodes permettant de gérer le fichier de logs (création, écriture et nettoyage du fichier). Toutes ses méthodes sont statiques – ce qui nous permet de normaliser l'écriture en fichier et de ne pas avoir à créer d'objets.

La classe *UserInterface* regroupe l'ensemble des méthodes d'interaction avec l'utilisateur (affichage, gestion des délais, mise en pause du programme et récupération de l'entrée utilisateur dans un intervalle de valeurs). Toutes ses méthodes sont statiques – ce qui nous permet de normaliser les interactions et de ne pas avoir à créer d'objets.

La classe *Statistics* regroupe l'ensemble des données clés générées par la simulation. Tous ses attributs et ses méthodes sont statiques – ce qui nous permet de ne pas avoir à créer d'objets. Grâce à elle, nous pouvons facilement calculer les statistiques de la simulation *(nombre de tours traversés, nombre de personnages adverses tués, nombre de représentants par famille et nombre de morts au combat/hors combat)*. L'intérêt d'avoir une classe dédiée aux statistiques est de ne pas surcharger les classes propres aux personnages et de pouvoir afficher l'ensemble des données en une seule fois.

La classe *GameMaster* regroupe l'ensemble des méthodes servant à faire tourner une simulation (initialisation, génération et conditions de fin). Nous en avons fait une classe à part entière afin d'encapsuler le fonctionnement global de la simulation : le développeur souhaitant générer des simulations à simplement besoin de fournir les valeurs des paramètres de la méthode *run*.

4 ANALYSE PERSONNELLE

4.1 REPARTITION DES ROLES

Pour ce projet, nous nous sommes réparti les rôles de la manière suivante :

- Clara : création des classes utilitaires (calcul des statistiques, gestion du fichier de logs, gestion du Game Master), création des méthodes liées aux déplacements des personnages, gestion des climats et des safezones, rédaction du rapport.

- Mathilde : création des classes liées aux personnages, création des méthodes liées aux rencontres et aux attaques, création de la carte, rédaction du rapport, conditions de fin, rédaction des messages écrits dans le fichier de logs.

4.2 BILAN

4.2.1 Difficultés rencontrées

En ayant d'autres projets en parallèle, nous avons tout d'abord eu des difficultés à trouver du temps à consacrer à ce projet. Lors des parties de cours consacrées au projet, nous avons également parfois eu l'impression de ne pas être très productives et de perdre du temps sur des choses simples – ce qui pouvait être pour nous source de stress et de frustration.

En dehors du cours de *Concept Objet*, il nous a été difficile de nous organiser et de bien déterminer les tâches de chacune. Dans ce projet, il arrivait que le code de l'une empiète sur le code de l'autre, qu'une fonction codée par l'une soit débuggée par l'autre, etc. Une meilleure attribution des rôles dès le départ nous aurait facilité la tâche. Certes, la communication est plus qu'importante, néanmoins, ayant des emplois du temps relativement différents et d'autres projets à gérer, elle a été pour nous très difficile à maintenir.

D'un point de vue technique, la méthode *move* a subi beaucoup de changements jusqu'à la version actuelle. De même, nous avons eu du mal à conceptualiser et à intégrer le mécanisme des safezones. Enfin, pour la classe *Statistics*, nous n'avons pas trouvé de solution « élégante » à temps, c'est pourquoi elle regroupe autant d'attributs.

4.2.2 Points de satisfaction

Notre plus grand point de satisfaction a été de voir que, malgré les difficultés rencontrées, nous avons été capables de mener à terme ce projet en y intégrant des fonctionnalités supplémentaires (gestion des climats, des marcheurs blancs, d'un fichier de logs, calcul de statistiques, etc.).

De plus, ce projet nous a permis d'approfondir nos connaissances en programmation orientée objet (polymorphisme, héritage et encapsulation), d'identifier nos faiblesses en termes d'organisation de projet et de programmation et de consolider nos compétences en Java.

Nous avons pu également nous améliorer en termes de communication - nos lacunes en matière d'organisation nous ont forcé à discuter ensemble pour trouver des solutions et réussir à rendre en temps et en heure un projet qui nous ressemble et qui est complet.

CONCLUSION

D'une façon générale, ce projet présente l'ensemble des notions présentées en cours de Concept Object : héritage, encapsulation, polymorphisme, diagrammes UML, simulation, etc. Ainsi, ce projet nous a apporté une expérience supplémentaire dans le domaine du développement.

En guise d'améliorations du projet, il serait intéressant d'implémenter des types de déplacements propres à chaque classe personnage (ou aléatoirement) comme mentionné dans l'énoncé du sujet. Nous aurions également pu mettre en place un système d'affichage plus ou moins détaillé selon la volonté de l'observateur ainsi qu'un système de simulation personnalisé, où l'utilisateur peut définir lui-même les valeurs d'initialisation. Pour avoir un suivi de plusieurs parties, il pourrait être intéressant de proposer à l'utilisateur de nettoyer le fichier de logs ou d'en créer un nouveau pour une autre simulation. De plus, nous avions pensé à ajouter un test d'esquive en cas d'attaque simple : si réussite critique, aucun dégât n'est infligé ; si réussite simple, la moitié des dégâts est infligée ; si échec, tous les dégâts sont infligés. Enfin, le jeu gagnerait en dynamisme avec un *pathfinding* entre les personnages pour qu'il y ait plus de rencontres et combats.

En termes de refactorisation et de simplification de code, nous aurions pu simplifier quelques points :

- Remplacer l'énumération des directions par des angles en degrés (valeurs numériques).

- Regrouper les attributs de *Statistics* dans une construction de la forme *Map<String,Map<String,Integer>>*, où le premier *String* correspond à la famille, le second à la statistique *(Added, DeadInBattle, ...)* et le *Integer* au nombre correspondant.

- Créer une classe *Faction* pour regrouper proprement la famille, sa safezone et ainsi gérer une liste de factions dans *GameMaster*. Cela nous permettrait aussi de simplifier la méthode *isFinished.*

ANNEXE : DIAGRAMME DE CLASSES