Architecture Logicielle et Qualité

Documentation

Rapport de Conception et d'Implémentation

Élèves:
Yannis Mahiou
Ilyace Benjelloun

Enseignant :
David Hill

Janvier-Avril 2020

Table des Matières

1 Introduction

L'objectif de ce TP était de concevoir une Simulation Multi-Agent(SMA). Notre idée pour la conception d'une SMA était la suivante : réaliser une arène de combat, où les agents seraient des gladiateurs. Nous avons dans un premier temps réalisé un dossier sur la partie analyse et conception de notre projet, et ce dossier s'inscrit dans la continuité de ce travail. Il reprend les éléments de conception que nous avons choisis au préalable, et met en valeur les choix et changements que nous avons opérés par rapport à ce premier dossier. Il présente également l'architecture actuelle de ce projet, ainsi que les différents outils que nous avons exploités pour arriver à ce résultat.

Au cours de la première partie, nous détaillerons les changements que nous avons réalisés par rapport à ce premier rapport, notamment au niveau des choix que nous avons réalisés, et des problèmes que nous avons rencontré. Nous présenterons l'architecture logicielle de notre Simulation Multi-Agent actuelle, avec notamment un diagramme de classes, et une présentation des patrons de conception que nous avons implémentés. Finalement, nous présenterons l'utilisation sur notre projet des différents outils que nous avons choisis en amont.

2 Changements de Conception

2.1 Choix de Conception

Notre phase d'analyse préalable nous faisait ressortir 3 patrons de conception qui étaient les suivants :

- Observateur : Gestion des interdépendances entre agents par une classe d'observation
- Factory : Point de création unique des agents
- Mediateur : Objet d'encapsulation des différentes interactions complexes entre agents

La Factory était un patron de conception important pour nous à implémenter. En effet, avoir un lieu unique de création des agents était primordial, afin de pouvoir contrôler facilement les statistiques des agents (points de vie et armure par exemple). Nous sommes donc restés sur ce patron de conception et nous l'avons implémenté rapidement lors de la phase de développement de notre SMA.

Par ailleurs, nous avions pensé à implémenter le patron observateur à la fin de notre projet. Cependant, nous avons choisi d'implémenter l'ensemble des fonctionnalités importantes (gestion, création, déplacements, combats) des agents en premier, afin d'obtenir rapidement un prototype fonctionnel. Il s'est avéré à la fin de cette phase de développement que nous n'avions pas réellement besoin d'implémenter le patron de conception Observateur. Ce patron de conception nous aurait permis de gérer les interdépendances entre agents, notamment une notifications des états des agents par rapport aux autres. Or, dans notre conception, nous avions réalisé que la modification de l'état d'un agent n'entraîne pas une modification de l'état des autres agents sur le terrain. Nous avons donc choisi de ne pas l'implémenter.

Concernant le patron de conception Médiateur, nous l'avions également gardé pour la fin du projet. Nos agents étant peu inter-dépendants, nous avons conclus que ce patron de conception n'était également pas nécessaire à implémenter. La réutilisation de nos agents était facile et la communication entre agents n'étais pas un point pertinent dans notre modélisation. Nous avons donc choisi de ne pas l'implémenter.

Par ailleurs, nous avons choisis d'implémenter 2 autres patrons de conception, qui nous semblaient plus judicieux dans notre conception.

Tout d'abord, nous avons implémenté le patron de conception Stratégie. Ce patron de conception permet de définir une famille d'algorithmes, permettant de les rendre interchangeables. En effet, l'utilisateur peut influencer le cours de la simulation. Pour cela, il peut donner un bonus a une des équipes au début de la partie. Ainsi, nous avons conçu, par l'intermédiaire de la stratégie, une famille d'algorithmes de génération d'agents. L'utilisateur peut ainsi choisir de générer les équipes de façon standard, c'est à dire que toutes les équipes sont équilibrées, il peut également décider d'en générer une en donnant un avantage à l'une des équipes (avantage fixé choisi par l'utilisateur sur les statistiques de base des agents d'une équipe (la bleue)), une génération avec des équipes avec une composition randomisée, et pour finir une génération chaos avec des équipes déséquilibrées, avec chaque agent pouvant avoir un bonus ou un malus aléatoire, allant de +5 à toutes les statistiques ou -5 pour ces dernières. Cette stratégie permet alors facilement, de gérer la création des agents, et ajoute une fonctionnalité supplémentaire à

notre SMA.

Par la même occasion, nous avons choisi d'implémenter le patron de conception Singleton. Ce patron de conception permet un point d'accès unique global à une classe. Dans notre cas, ce patron de conception était judicieux à mettre en place, car nous avions beaucoup de nombres aléatoires à générer dans notre projet.

Pour obtenir de la répétabilité dans nos expériences aléatoire, générer l'ensemble des nombres aléatoire par un seul générateur accessible partout dans notre modèle nous semblait primordial. Nous avons donc créé une classe RandomSingleton, qui possède un générateur initialisé avec un seed (graine) qui permet d'obtenir la répétabilité des expériences aléatoires de notre SMA.

Nous avons ainsi modélisé et implémenté 3 patrons de conceptions différents qui seront présentés dans la partie suivante de ce rapport.

Concernant le reste du projet, nous avons réalisé un changement majeur. En effet, nous voulions faire déplacer nos agents équipe par équipe à la manière d'un jeu tour par tour cependant, nous nous sommes rendus compte que cela donnait un avantage bien trop important à l'équipe qui commençait en premier. Notre objectif étant en premier lieu avoir des taux de victoire de chaque équipe s'approchant de 50% pour des compositions équilibrées, nous avons donc décidé de faire déplacer un agent de chaque équipe chacun son tour et de rendre la position de départ beaucoup plus aléatoire et non cantonnée à une partie du terrain en fonction de l'équipe. Pour les autres aspects du projet, nous sommes restés identiques aux choix que nous avions réalisés au dans le rapport d'analyse et conception. Nous avons conçu l'interface console que nous voulions, et nous avons pu également concevoir et manipuler nos agents de la manière que nous voulions.

Nous avons ajouté une classe de statistique qui nous semblait manquante par rapport à la phase d'analyse. En effet, il nous semblait important d'obtenir des résultats sur les différentes expériences de nous réalisions sur les agents durant la SMA. Ainsi, nous avons une classe de statistique, qui permet d'obtenir des résultats tels que le taux de victoires d'une équipe par rapport à une autre, ainsi que des statistiques sur le nombres de coups reçus, ou bien le nombre d'unités tuées par exemple.

Nous avons présenté l'ensemble des modifications de conception que nous avons pu réalisé au cours de ce projet, nous allons maintenant présenter l'architecture logicielle de notre projet et détailler plus en profondeur certaines parties algorithmique du logiciel que nous avons développé.

3 Architecture logicielle

3.1 Diagramme de package

Notre projet est composé de 6 packages, qui ont chacun un rôle précis dans notre modèle. On retrouve les packages suivants :

- Agent : contient toutes les classes d'agents et relatives à ces derniers, ainsi que les algorithmes qui permettent de les manipuler
- Factory : contient les classes du pattern Factory, point de création unique des agents dans notre modèle
- Game : contient la boucle de simulation principale, ainsi que le Singleton pour le générateur de nombres aléatoires. Il contient également une classe de statistiques permettant d'obtenir des résultats statistiques sur la simulation
- Serialization : contient les classes de serialization, nécessaires à la répétabilité des expériences aléatoires dans notre simulation
- Strategy : contient le patron de conception strategy, utilisé pour choisir le mode de création des équipes d'agents dans le cadre de notre simulation
- Terrain : contient les classes de modélisation du terrain, ainsi que les primitives de placement des agents et de gestion de l'interface utilisateur

Cette division en package nous a permis d'obtenir une responsabilité unique associée à chaque package, et ainsi obtenir une division efficace de nos classes. Nous pouvons ainsi représenter le diagramme de package de la manière suivante (voir figure 1)

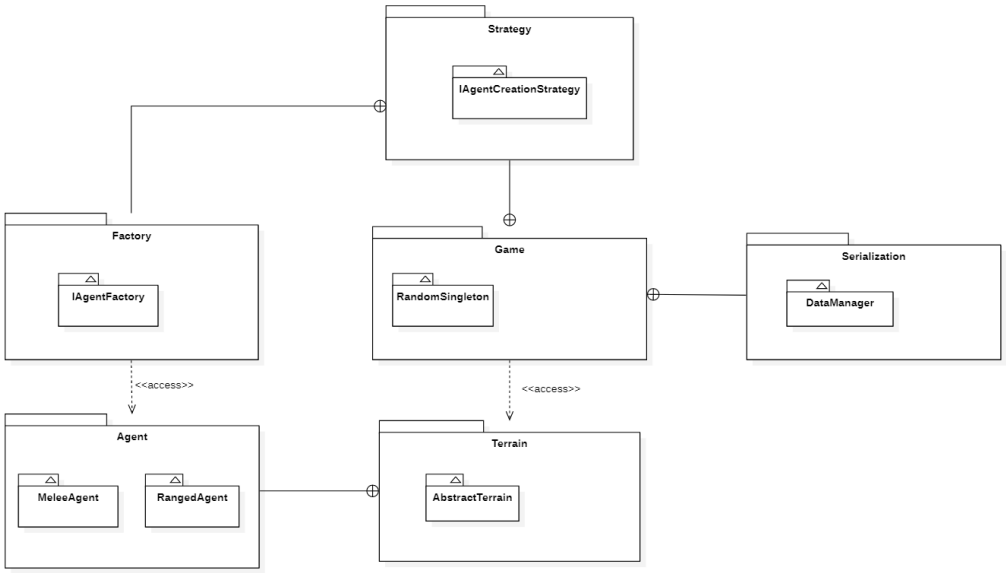


Figure 1: Diagramme de package de notre projet SMA

Dans la partie suivante, nous présenterons le diagramme de classe d'analyse, ainsi que le diagramme de classe de conception, que nous avons conçu avec l'outil Dia.

3.2 Diagramme de classe

3.3 Diagramme de classe d'analyse

Ce diagramme de classe a été réalisé lors de la phase d'analyse. Cette phase préliminaire à la phase de développement nous a permis de sortir le diagramme de classe suivant :

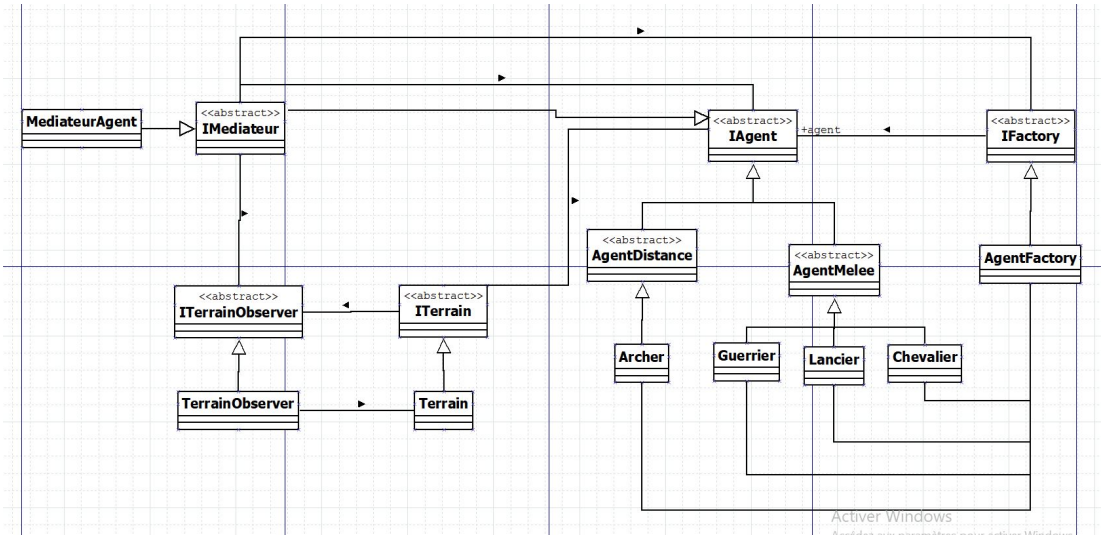


Figure 2: Diagramme de classe d'analyse

On y retrouve l'architecture que nous avions établie au préalable lors de cette phase d'analyse. Il y figure les patrons de conception que nous avons changé, l'architecture principale des agents, et le terrain. Ce diagramme de classe nous a permis tout de suite de nous rendre compte de l'ordre dans lequel nous allions procéder à notre développement. Nous avions vu immédiatement que l'architecture des agents était la structure à implémenter en premier. La Factory et le terrain venaient ensuite, pour finir avec les autres patrons de conception.

Ce diagramme de classe est cependant bien différent du diagramme de classe que nous avons obtenus par la suite et qui est présenté dans la partie suivante. (figure 3)

3.4 Diagramme de classe de conception

Une fois la phase de développement réalisée, il nous a été primordial de modifier notre diagramme de classe d'analyse afin de construire un nouveau diagramme de classe : le diagramme de classe de conception.

Ce diagramme de classe comporte l'ensemble des méthodes utilisées dans notre simulation, ainsi que les différents liens entre nos classes (héritage, réalisation d'interface, utilisation). Nous l'avons représenté à l'aide de l'outil Dia dans la figure 3

Ce diagramme de classe comporte de nombreuses architectures. On retrouve notamment la structure d'héritage que nous avions préalablement imaginée lors de la phase d'analyse et qui n'a pas changé lors de la phase de conception. La modélisation du terrain est également restée

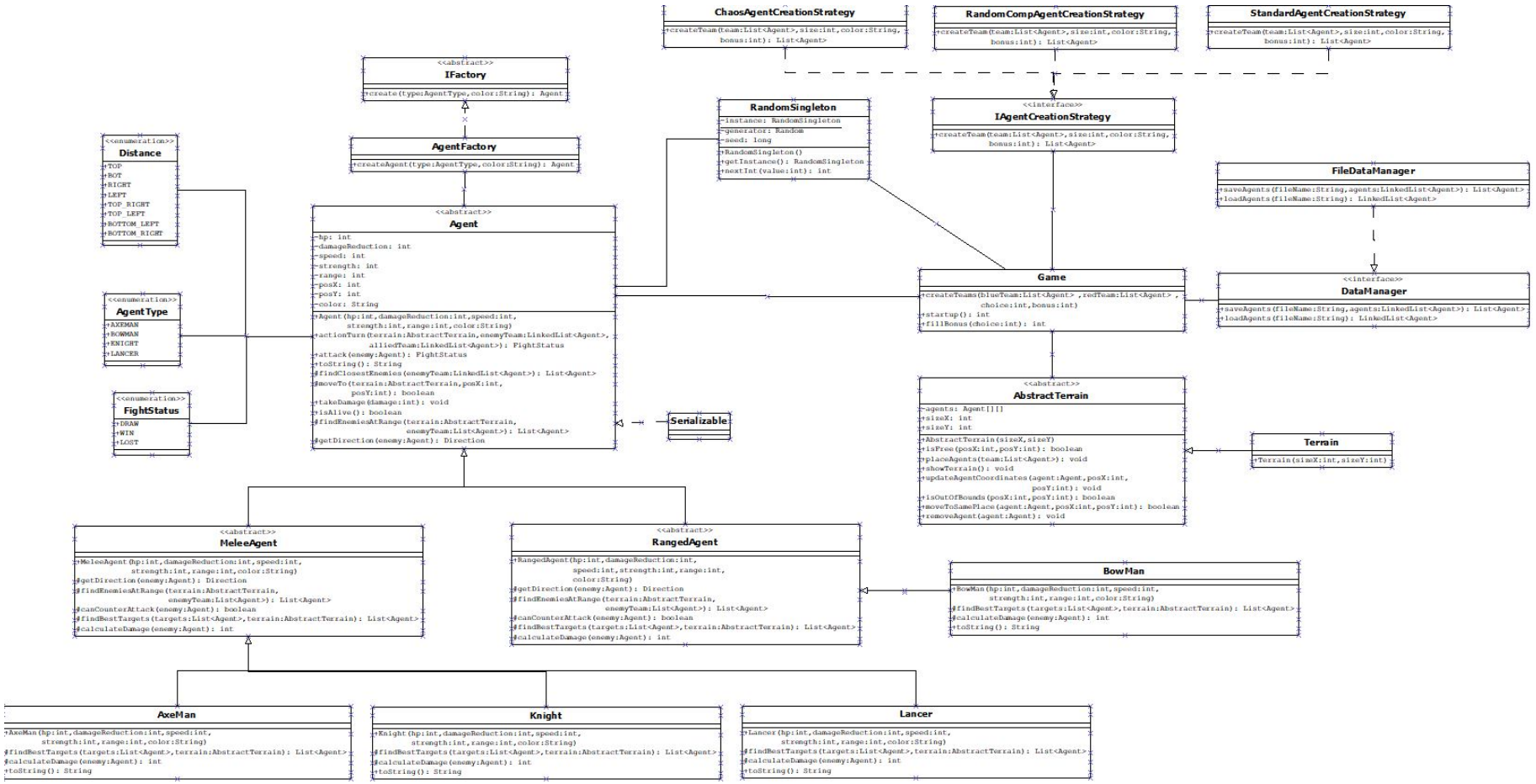


Figure 3: Diagramme de classe de conception

identique par rapport au précédent diagramme. Par ailleurs, on y retrouve les nouvelles fonctionnalités que nous avons implémenté (Serialization, Génération de nombres aléatoires, Factory) ainsi que les nouveaux patrons de conceptions que nous avons choisis (Singleton, Stratégie). Ces nouveaux patrons seront expliqués dans la partie suivante, avec notamment, un accent sur les principes SOLID, que nous avons voulu respecter, tout au long de notre phase de développement.

3.5 Patrons de conception

3.6 Singleton

Le singleton est un patron de conception utilisé lorsqu'une classe doit être instanciée une seule et unique fois et accessible à plusieurs endroits différents. Le singleton englobe une instance de la classe à instancier qu'une seule fois et possède une méthode unique et statique getInstance(). Celle-ci peut se charger de la première instanciation de classe si nécessaire.

Dans notre projet, nous avons utilisé ce patron de conception car nous voulions avoir un seul et unique générateur de nombres aléatoires initialisé avec le seed (ou graine) que nous avions décidé pour la répétabilité de notre simulation. De ce fait nous appelons le générateur créé et initialisé dans notre RandomSingleton à chaque fois que nous avons besoin de générer un quelconque nombre aléatoire.

Nous avons représenté le Singleton de la manière suivante (figure 4)

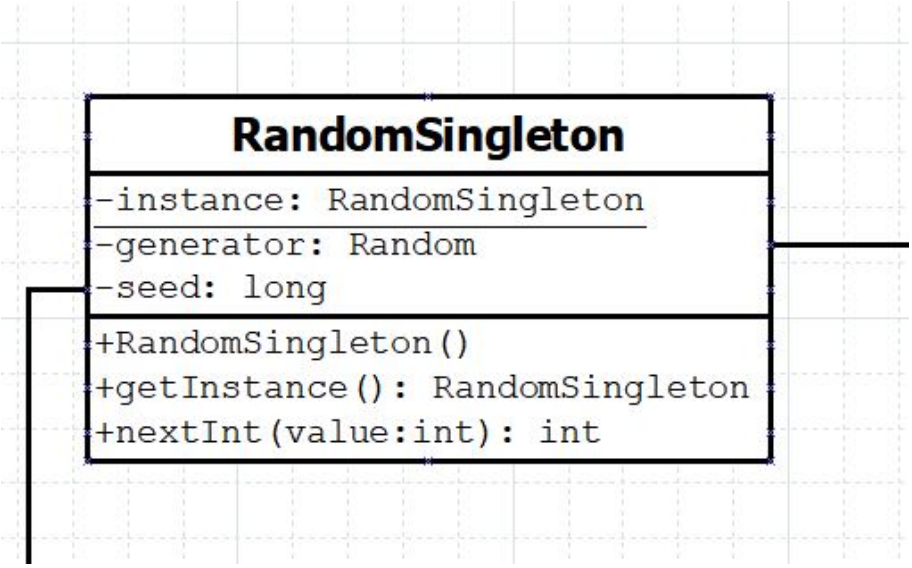


Figure 4: Patron de conception Singleton utilisé pour la génération de nombres aléatoires

3.7 Factory

La Factory (ou Fabrique) est un patron de type créateur, il permet de fabriquer des produits dont on ne connaît pas nécessairement le type avant l'instanciation, par exemple dans notre projet des

Chevaliers ou des Lanciers héritant d'une classe mère Agent, ou objets composés en déléguant les instanciations des sous classes. Concrètement, lors de l'exécution d'un programme, la fabrique est instanciée par la méthode qui en aura besoin puis créera tous les objets nécessaires à partir de celle-ci. Cependant, la fabrique n'instancie qu'un seul objet à la fois. Il faut donc faire une boucle for pour en créer plusieurs.

Notre fabrique AgentFactory permet de créer une instance d'objet Agent. Cette responsabilité lui est déléguée et c'est la seule qu'elle assume. Toutes les instanciations de Agent sont donc assurées par cette Factory et on s'assure ainsi que toutes nos classes n'ont qu'une seule responsabilité.

Nous avons représenté la Factory de la manière suivante (figure 5)

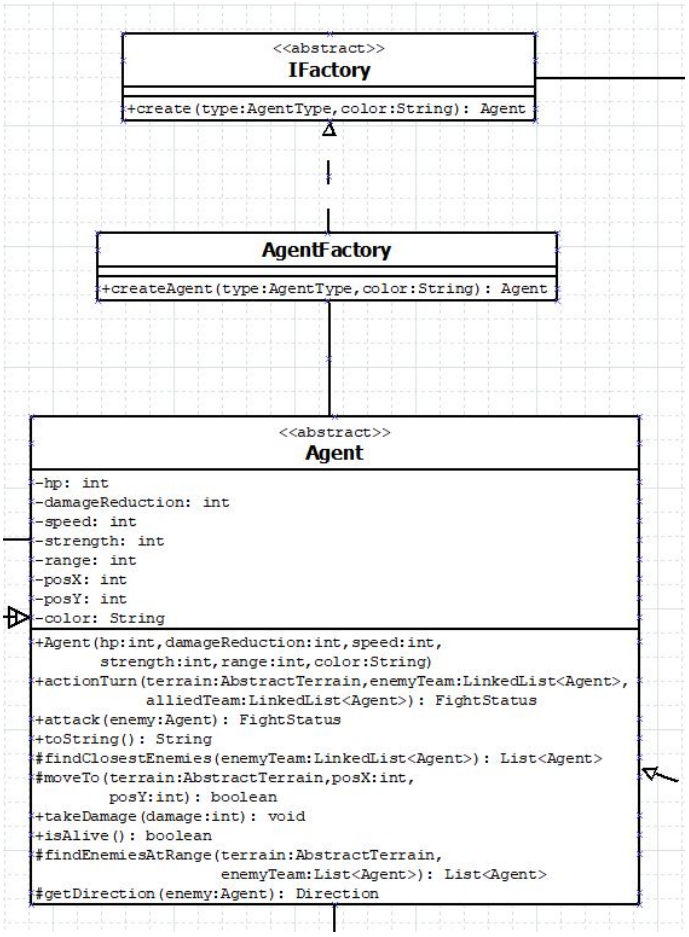


Figure 5: Patron de conception Factory utilisé pour la création des agents

3.8 Stratégie

La Stratégie est un patron de type comportemental. Il permet de définir une famille d'algorithmes interchangeables. Ce patron permet aux algorithmes d'évoluer sans influer sur le reste de l'application.

Nous avons implémenté en soit 2 Stratégies différentes : Une pour la création des équipes comme énoncé plus haut et une autre pour les comportements de chaque Agents. Un agent (gladiateur) attaquant en mêlée n'aura pas le même comportement qu'un agent attaquant à distance, de une sur ses cibles privilégiées et de deux sur son placement par rapport à l'adversaire. En descendant à un niveau encore plus bas, un Chevalier aura plus tendance à attaquer un gladiateur muni d'une hache qu'un gladiateur muni d'une lance, tout comme un gladiateur muni d'une lance éviterai les confrontations avec un gladiateur muni d'une hache. Quant à eux les gladiateurs attaquant à distance ont tout intérêt à attaquer ceux attaquant en mêlée pour ne pas se prendre une contre attaque d'un autre archer!

La stratégie nous permet de pouvoir faire plusieurs types de création d'agents. Ainsi, on peut ajouter autant de types de création d'agents que l'on veut seulement par extension et sans modification de code. Le modèle de conception est alors plus facilement maintenable et évolutif.

Nous avons représenté la Stratégie de la manière suivante (figure 6)

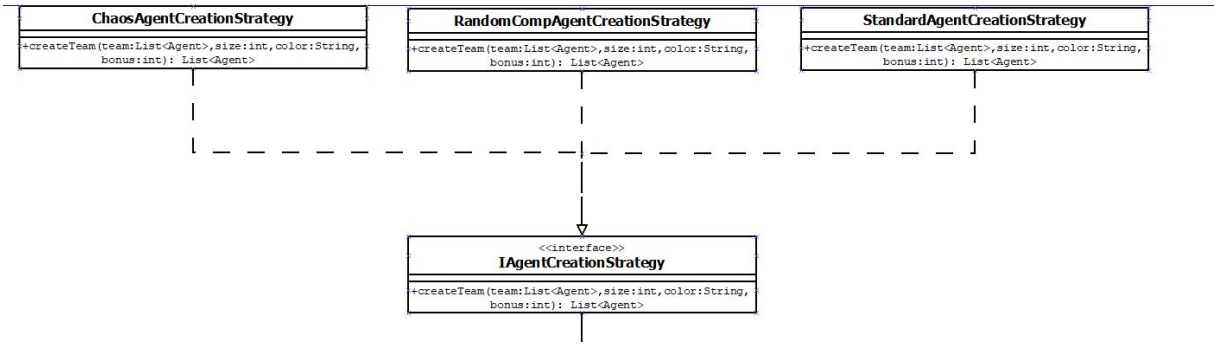


Figure 6: Patron de conception Stratégie utilisé pour la méthode de création des agents

3.9 Éléments importants

Nous avons cherché à respecter au mieux les principes SOLID.

3.9.1 S : Principe de responsabilité unique

Chaque classe doit avoir une responsabilité unique.

Appliquer ce concept permet de découpler les responsabilités pour que le changement d'une classe n'ait pas d'impact sur plusieurs responsabilités.

Nous avons appliqué ce principe dans notre projet et chaque classe a une responsabilité unique.

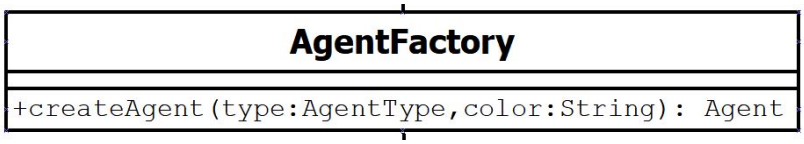


Figure 7: Principe de responsabilité unique

Par exemple, nous avons créé une Factory qui n'a pour seule responsabilité que de créer un objet Agent. Cette responsabilité est unique et propre à cette classe. Ainsi, en cas de modification de cette Factory aucune autre responsabilité n'en sera affectée.

3.9.2 O : Principe de l'ouverture / fermeture

On doit créer une application dont le modèle est ouvert à l'extension et fermé à la modification.

Nous avons essayé de respecter au mieux ce principe dans notre projet, en voici un exemple :
Pour réaliser la création des agents, nous avons utilisé une interface IAgentCreationStrategy. C'est cette dernière qui est manipulée dans le projet. Ainsi, peu importe le type de création d'agent utilisé, il suffit simplement de choisir si on veut utiliser une création chaotique, aléatoire, ou standard.

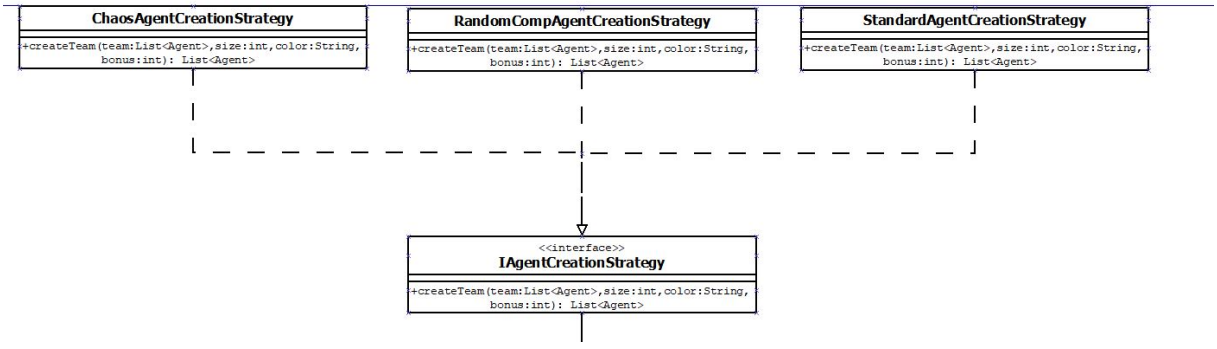


Figure 8: Principe de l'ouverture / fermeture

De plus, si on veut changer de type de création et implémenter une création différente, aucune classe n'est modifiée mais l'interface IAgentCreationStrategy est étendue par héritage. L'application est donc beaucoup plus facilement maintenable et permet d'éviter l'injection de bugs.

3.9.3 L : Principe de la substitution de Liskov

Une fonction qui utilise un objet d'une classe mère doit pouvoir utiliser toute instance d'une classe dérivée sans avoir à la connaître.

C'est le principe qui permet de profiter énormément du polymorphisme et qui joue sur la substituabilité. Ainsi, on va chercher à placer les méthodes le plus haut possible pour que toutes les classes filles puissent y avoir accès.

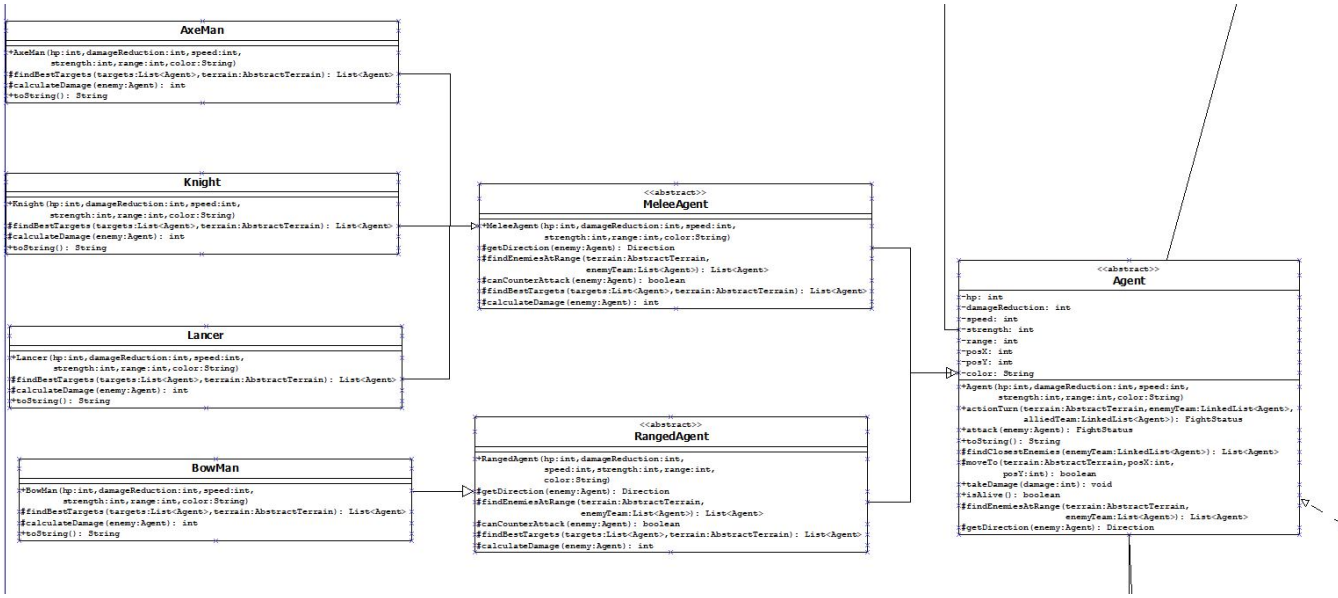


Figure 9: L : Principe de la substitution de Liskov

Ainsi, le maximum de méthodes sont remontées et s'appliquent sur tous les enfants. On peut alors manipuler un Agent sans problème et sans savoir si ce dernier est un Agent mêlée ou bien un Agent à distance.

3.9.4 I : Principe de ségrégation d'interface

Ce principe consiste à découper une interface en plusieurs petites interfaces si celle-là s'avère trop importante.

Nous n'avons d'interfaces qui sont trop importantes. Nous n'avons donc pas eu besoin de découper nos interfaces en interfaces plus petites.

3.9.5 D : Principe d'inversion de dépendance

Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau et les modules de bas niveau doivent dépendre d'abstraction.

Nous avons essayé de respecter au mieux de principe, en voici un exemple :

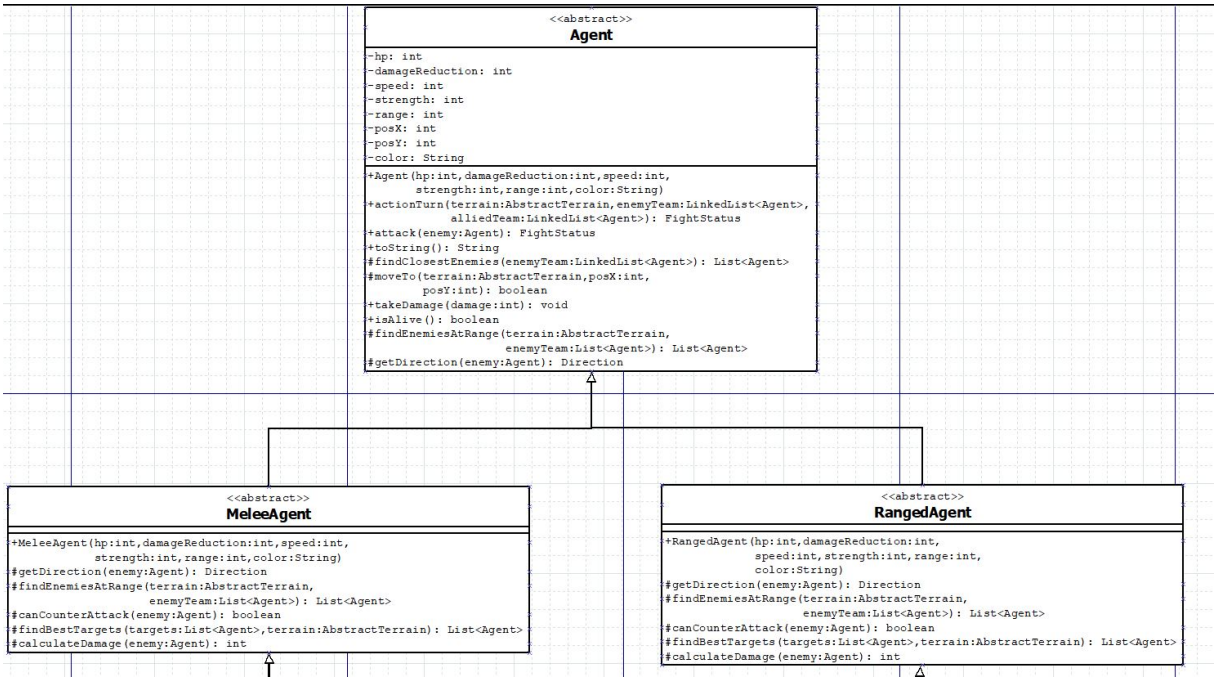


Figure 10: D : Principe d'inversion de dépendance

Il y a un contrat passé entre la classe abstraite Agent, RangedAgent et MeleeAgent . Ainsi, de l'extérieur du modèle on ne manipule que des Agent pour protéger les classes inférieures. Le module du bas ne dépend donc que d'une couche d'abstraction. On crée ainsi une boîte noire et l'on peut utiliser des Agent sans savoir ce qu'il se passe derrière.

4 Mise en oeuvre des outils

4.1 Dia : Ilyace BENJELLOUN

Nous avons décidé de réaliser notre conception dès le départ en utilisant des outils de création de diagrammes UML, après avoir cherché différents outils numériques libres on nous a conseillé d'utiliser Dia, un logiciel complet qui nous permettrai de pouvoir réaliser des diagrammes facilement modifiables et très parlant.

Nous avons tout débord réalisé un premier diagramme de classe lors de la phase d'analyse avec Dia que nous avons repris et modifié quelques fois par la suite lors de la phase d'analyse puis de développement. Nous avons par exemple supprimé une interface qui n'était au final pas si utile ou ajouté le RandomSingleton auquel nous n'avions pas pensé au départ mais qui était primordial pour la répétabilité de notre simulation.

Ensuite nous avons réalisé un diagramme de UseCase permettant de savoir comment pourrait influencer l'utilisateur sur la simulation. Nous avons depuis ce diagramme ressorti la trame principale des actions possibles pour l'utilisateur.

Pour finir, nous avons réalisé un diagramme de package afin de bien pouvoir visualiser les différents composants de notre application.

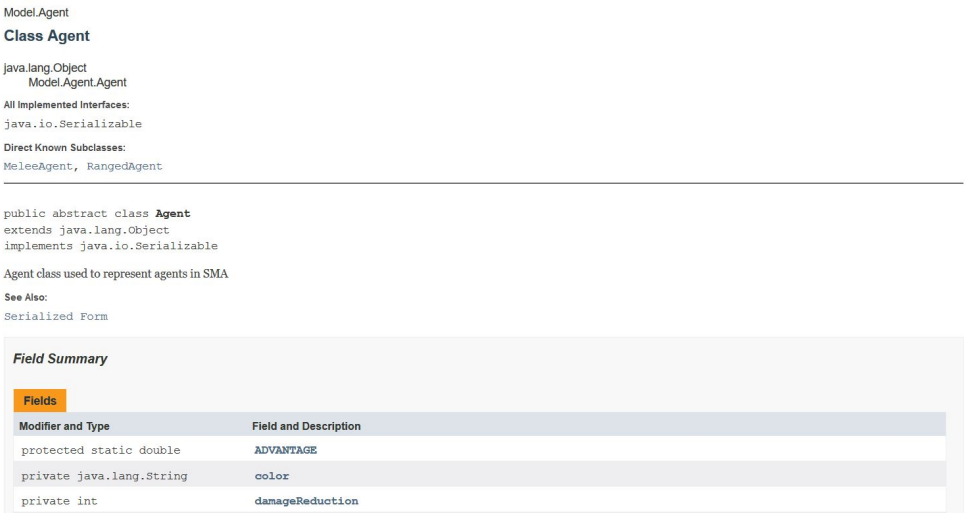
4.2 JavaDoc : Yannis MAHIOU

La mise en oeuvre de la JavaDoc était plus qu'évident. La documentation nous permettait de comprendre le code qui était écrit par l'autre, et ce de manière claire et compréhensible. Nous l'avons utilisé dès le début de la phase de conception, ce qui nous a permis d'obtenir rapidement un code documenté.

Une fois le projet réalisé et la documentation de type JavaDoc réalisée, nous avons pu la générer par l'intermédiaire de l'IDE que nous avons utilisé : IntelliJ.

Il nous suffisait alors d'aller dans le menu, de choisir la niveau de visibilité de la javadoc que nous voulions (dans notre cas : public afin d'obtenir l'ensemble des commentaires que nous avions écrits, peut importe la visibilité objet). Par la suite, il nous suffisait de générer la JavaDoc dans un dossier de notre choix. Nous avons choisi de placer la JavaDoc à la racine de notre projet, afin qu'elle soit facilement accessible. On la retrouve ainsi dans le dossier *javadoc*.

Voici un exemple de la documentation de la classe Agent:


Model.Agent

Class Agent

java.lang.Object
Model.Agent.Agent

All Implemented Interfaces:
java.io.Serializable

Direct Known Subclasses:
MeleeAgent, RangedAgent

public abstract class **Agent**
extends java.lang.Object
implements java.io.Serializable

Agent class used to represent agents in SMA

See Also:
Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
protected static double	**ADVANTAGE**
private java.lang.String	**color**
private int	**damageReduction**

Figure 11: Exemple de documentation JavaDoc pour la classe Agent

On peut cliquer sur une des méthodes montrée en bas de la figure 11 afin de montrer la déclaration de la méthode et obtenir plus d'informations sur les arguments.

5 Résultats obtenus

Avant de commencer, tous les résultats ont étés obtenus en utilisant la même graine (ou seed) pour le Générateur de Nombre Pseudo-Aléatoires.

Afin de voir si notre algorithme donnait des chances équivalentes aux deux équipes, nous avons procédé de la manière suivante : pour un nombre fini d'expériences nous générons chacune des équipes et plaçons chacun des agents sur le terrain de façon aléatoire selon les règles établies puis sauvegardons chacune des équipes dans un fichier. Nous lançons la simulation et incrémentons un compteur lorsqu'une équipe est victorieuse, puis rechargeons les équipes et leur positions afin de recommencer l'expérience. En effet le choix de la cible final de chaque agent et la position qu'il prendra face à son adversaire étant aléatoire, il se peut que le cours d'une partie puisse entièrement changer en fonction des choix des agents. Nous répétons cette procédure 30 fois. Nous avons cherché à savoir si dans certaines conditions nous retrouvions un penchant avantageux pour l'équipe bleu qui possède le premier tour (première équipe avec une unité attaquant).

En fonction des stratégies de création des équipes nous constatons les résultats suivants :

- Mode Standard : Avantage faible pour l'équipe Bleue, cet avantage est sûrement expliqué par le fait que l'équipe bleue possède la première unité à pouvoir se déplacer et donc attaquer.
- Mode chaos : C'est le mode le plus déséquilibré, on constate très souvent des taux de victoire supérieurs à 80% et dans certains cas plus de 95%
- Mode Avantage (avec bonus) ; Avec un avantage de 1 sur les statistiques, l'équipe bleue gagne plus de 95% des matchs, cela est du au fait qu'un gladiateur avec un avantage peut battre un autre gladiateur sur lequel il a un avantage en 2 combats seulement, de plus, alors qu'il en faudrait 3 pour une unité de l'équipe bleue. A partir de 2 points bonus dans toutes les statistiques, le taux de victoire de l'équipe bleue est à 100%, il faut au minimum 4 combats à chaque gladiateur de l'équipe rouge ayant un avantage pour battre un agent de l'équipe bleue. A partir d'un bonus de 5 points, il n'est plus possible pour un membre de l'équipe bleue n'ayant pas de désavantage de ne pas tuer un gladiateur de l'équipe rouge en moins d'un seul combat à cause des doubles coups.
- Mode Avantage (avec Malus) : Avec une pénalité de 1 sur les statistiques, l'équipe bleue perd plus de 99, il devient possible pour un gladiateur de l'équipe rouge sans désavantage de battre un gladiateur de l'équipe rouge en deux combats. Puis nous constatons les mêmes effets qu'avec un avantage mais cette fois-ci dans le sens de l'équipe bleue, le taux de victoire atteint 100% à partir de 2 points de pénalité.
- Mode Équipes Aléatoires : C'est également un mode très déséquilibré. En effet en fonction de la composition des équipes, il est possible qu'une équipe n'aie quasiment aucune chance de gagner, comme une équipe composée majoritairement de lanciers contre une équipe composée majoritairement de gladiateurs à la hache donnant un avantage certain à cette dernière...

Nous avons également mis au point un système qui nous permet de savoir où se situe en majeure partie les combats. A la fin de chaque chaque, quand chaque agent a effectué au moins une action (ou tenté), nous enregistrons la position de chaque unité. En modifiant les règles de position de départ pour les rendre complètement aléatoires, nous nous attendions à ce que les combats se concentrent bien plus sur le centre du terrain que le haut ou le bas de carte. Les résultats obtenus confirment nos attentes principales, les cases en milieu de terrain sont bien plus souvent occupées que les cases sur le bord du terrain. La case en plein centre du terrain est également la plus occupée tandis que les cases aux angles du terrain sont les moins occupées. Nous observons une diminution progressive du nombre de fois qu'un agent se trouvait sur une dite case à mesure que nous éloignons du centre.

Pour finir nous nous sommes également intéressés à la proportion de combats gagnés, perdus ou finissants sur une égalité (les deux agents sont en vie) ainsi que la proportion de tours sans combats. Il est possible qu'un agent n'engage pas de combat si la case sur laquelle il souhaitait se rendre est occupée, dans ce cas il ne se déplace pas et nous comptons cette éventualité.

En fonction des stratégies de création des équipes nous constatons les résultats suivants :

- Modes Standard, Chaos et Équipes Aléatoires : Comme on devrait le deviner plus de la moitié des combats finissent sur une égalité, nous constatons 8% de dégâts perdus et 32% de combats gagnés. 12% des tours ne donnent pas lieu à un combat.
- Mode Avantage (avec bonus de 5) : Comme expliqué plus haut, nous retrouvons un nombre de combats gagnés très grand (plus de 60%) et un nombre de combats finis sur une égalité en large baisse (aux alentours de 30%). Les gladiateurs attaquant en priorité les adversaires sur lesquels ils ont un avantage, il est très peu probable qu'un gladiateur meure au combat en l'initiant. De plus on dénombre moitié moins de combats et d'actions que dans les autres modes.
- Mode Avantage (avec Malus) : Nous observons sensiblement les mêmes résultats que pour le mode avec avantage de 5 points. Cependant, l'équipe bleue étant la première à attaquer, il y a bien plus de chances que ses gladiateurs meurent en initiant un combat, 20% des combats se terminent sur une défaite de l'attaquant, et 20% sur une égalité.. On constate encore moins de combats et d'actions et une augmentation nette

6 Présentation Des Outils

6.1 JavaDoc : Yannis MAHIOU

6.1.1 Introduction

JavaDoc est un système de documentation développé par Sun MicroSystems (rachat par Oracle) pour les programmes écrits en Java.

Cet outil permet de générer une documentation d'API en format HTML, depuis les commentaires d'un type particulier présents dans un code source en Java.

JavaDoc est le standard industriel pour la documentation des classes Java. La plupart des IDE (IntelliJ, NetBeans...) génèrent automatiquement la JavaDoc au format HTML. Ces pages HTML contiennent au minimum la liste des classes, la liste des méthodes et la liste des variables utilisées.

L'avantage du format HTML utilisé dans la documentation produite tient dans la présence de liens hypertextes. Il devient très facile de naviguer dans la documentation, au fil de la lecture.

Par ailleurs, une JavaDoc peut être générée dans différents formats via l'utilisation de Doclets. En effet, il existe un grand nombre de Doclets, qui permettent d'exporter la JavaDoc en différents formats, tels que PDF, XML, Ant Tasks, LaTeXpar exemple. De base, JavaDoc utilise le Doclet standard, si aucun autre Doclet est spécifié.

6.1.2 Procédure d'installation

Pour les plates-formes Windows et Unix, l'outil JavaDoc est inclus dans tous les JDK (ou SDK) de Java. Pour télécharger la dernière version du JDK, il faut se rendre sur la page de téléchargement de Sun :

https://www.oracle.com/java/technologies/javase-downloads.html

L'exécutable utilisé pour la génération de la JavaDoc, c'est à dire de l'exécutable javadoc.exe qui se trouve dans le répertoire *bin* du jdk.

Pour installer les Doclets, il suffit de se rendre sur le site :

http://doclet.com/

On retrouve de nombreux Doclets, tel que TeXDoclet ou PDFDoclet, qui permet de produire une documentation JavaDoc en version LaTeX. Leurs utilisations est expliquée dans la partie Guide d'utilisation.

6.1.3 Fonctionnalités

L'outil JavaDoc utilise plusieurs types de fichiers pour générer la documentation :

- Fichiers sources .java

- Fichiers de commentaires d'ensemble
- Fichiers de commentaires de packages
- fichiers de type images, fichier HTML

En fonction des paramètres fournis à l'outil, ce dernier recherche les fichiers source .java concernés. Les sources de ces fichiers sont scannées pour déterminer leurs membres, extraire les informations utiles et établir un ensemble de références croisées.

Le résultat de cette recherche peut être enrichie avec des commentaires de code. Ces commentaires doivent immédiatement précéder l'entité qu'ils concernent (classe, interface, méthode, constructeur ou champ). Seul le commentaire qui précède l'entité est traité lors de la génération de la documentation.

L'utilisateur peut également utiliser des tags définis par JavaDoc. Cet outil traite les tags d'un certaine manière, et en définit plusieurs qui permettent de mieux décrire un élément. Les tags commencent toujours par le caractère @.

On retrouve par exemple les tags suivants :

- @param : permet de documenter un paramètre de l'élément
- @author : permet de préciser un ou des auteurs d'un élément
- @return : permet de décrire la valeur de retour d'une méthode
- @deprecated : permet de spécifier qu'une entité ne devrait plus être utilisée
- ...

Une liste des tags JavaDoc est présente sur la page de documentation officielle de l'outil Oracle :

https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

De plus, l'utilisateur peut inclure des balise HTML dans cette javadoc, ce qui peut être utile lorsque la description d'un objet est longue et prend plusieurs ligne. On peut ainsi utiliser une balise HTML p, qui permet alors de mieux structurer la description.

Concernant le code, les commentaires doivent suivre des règles précises. Le format de ces commentaires commence par /** et se termine par */. Il peut contenir un texte libre et des balises particulières.

Le commentaire peut être sur une ou plus généralement sur plusieurs lignes. Les caractères d'espacement (espace et tabulation) qui précèdent le premier caractère * de chaque ligne du commentaire ainsi que le caractère lui-même sont ignorés lors de la génération. Ceci permet d'utiliser le caractère * pour aligner le contenu du commentaire.

Voici un exemple d'utilisation de la JavaDoc tiré de notre projet :

```java
/**
 * Checks if the posX and posY given aren't out of bounds from the Terrain
 *
 * @param posX the X coordinate
 * @param posY the Y coordinate
 * @return true if out of bounds
 * @author Yannis MAHIOU
 */
public boolean isOutOfBounds(int posX, int posY) {
    boolean isOut = false;
    if (posX < 0 || posY < 0 || posX >= NB_AGENTS || posY >= NB_AGENTS)
        isOut = true;
    return isOut;
}

/**
 * Checks if the agent wants to move to the same [X][Y] coordinates
 *
 * @param agent the agent
 * @param posX the X coordinate
 * @param posY the Y coordinate
 * @return true if the coordinates correspond, false else
 * @author Ilyace BENJELLOUN
 */
public boolean moveToSamePlace(Agent agent, int posX, int posY) {
    return agent.getPosX() == posX && agent.getPosY() == posY;
}
```

6.1.4 Guide d'utilisation

La JavaDoc se construit par l'intermediaire de la commande javadoc. Cette commande s'utilise de la manière suivante :

```
$ javadoc [options] [packagenames] [sourcefiles] [classnames] [@files]
```

Les options utilisables avec cette commande sont :

- public seuls les attributs et méthodes publics.
- protected seuls les attributs et méthodes publics et protégés.
- package seuls les attributs et méthodes public, protected et paquetage.
- private tous les attributs et toutes les méthodes.
- doclet permet de spécifier un doclet autre que le standard. Un doclet permet une présentation différente de la documentation.
- docletpath le chemin où trouver le doclet à utiliser.
- sourcepath sourcepathlist spécifie le chemin où trouver les fichiers sources.

- classpath classpathlist spécifie le chemin où trouver les fichiers .class.
- exclude liste des paquetages à exclure.
- subpackages liste des sous paquetages à inclure.
- verbose affichage des messages pendant la construction de la documentation

L'utilisateur peut également lancer la commande :

```
$ javadoc -help
```

afin d'obtenir une aide sur les commandes à utiliser.
Un exemple d'utilisation de cette commande :

```
$ javadoc -d doc path/to/files/*.java
```

Cette commande va créer dans le dossier *doc* l'ensemble de la documentation des classes .java spécifiée dans le chemin indiqué par l'utilisateur. On retrouve alors un dossier, qui contient différents fichiers HTML, un fichier css pour l'affichage de la documentation, l'arborescence du projet sous forme de documentation, ainsi qu'un index (fichier HTML), qui permet d'obtenir un "menu" de la JavaDoc.

On peut obtenir une documentation comme ceci pour le package agent :

Package Model.Agent

Class Summary

Class	Description
Agent	Agent class used to represent agents in SMA
AxeMan	Melee Agent with an axe
BowMan	Ranged Agent with a bow
Knight	Melee Agent with a sword
Lancer	Melee Agent with an axe
MeleeAgent	Melee Agent : Axe, Sword, Lance
RangedAgent	Ranged Agent : Bow

Enum Summary

Enum	Description
AgentType	Enumeration used to propose several types of Agents in the SMA
Direction	Directions of moves used by moves algorithms
FightStatus	Status of the fight

On obtient des liens cliquables qui nous permettent de nous déplacer dans l'arborescence du projet, et d'obtenir d'avantage d'informations sur une classe, ou une méthode en particulier. Par ailleurs, on peut naviguer dans la documentation par package, ce qui permet une meilleure cohérence dans la présentation de la documentation. On peut ensuite se diriger dans les classes, et par la suite dans les usages d'un élément dans le projet. Cette structure arborescente est pratique car elle permet de descendre par couche d'abstraction dans l'arborescence du code.

6.1.5 Points clés

- Support de multiples formats d'affichage de JavaDoc via les différents Doclets (LaTeX, PDF...)
- Personnalisation de la documentation via les balises HTML et les stylesheets CSS
- Documentation de classes qui ne sont pas documentée afin de connaître leurs structures
- Lisibilité de la documentation : claire et synthétique
- Grande compatibilité avec de nombreux IDE qui génèrent automatiquement la JavaDoc
- Facilité d'utilisation et des commentaires et génération de JavaDoc

6.2 Dia : Ilyace BENJELLOUN

6.2.1 Introduction

Dia est un logiciel libre de création de diagramme développé à l'aide du langage C et distribué sous la licence GNU GPL.

Il permet de réaliser jusqu'à 30 types diagrammes de toutes sortes dont des diagrammes de classes et autres diagrammes de type UML, des modèles de base de données, des circuits électroniques tout en passant par la confection d'un puzzle.

Dia ne pose aucune restriction sur les composants de diagrammes utilisés. Ainsi il est possible de mettre des formes et des connecteurs provenant de plusieurs catégories différentes. Le logiciel propose également une fonctionnalité appelée dia2code permettant de générer du code PHP, JAVA, C++... à partir d'un diagramme UML réalisé à l'aide de l'outil.

Dia est un outil portable fonctionnant sur la plupart des plate-formes UNIX. Des exécutables pour Linux, Windows et Mac OS X sont disponibles.

6.2.2 Procédure d'installation

Dia est un logiciel portable, il est exécutable à partir d'une clé USB. Il suffit simplement de télécharger l'installateur sur le site http://dia-installer.de/download/index.html et de sélectionner la version pour le système d'exploitation désiré parmi Linux, Mac OS X et Windows, vous trouverez également la version pour clé USB sur cette page.

Il ne reste plus qu'à exécuter le logiciel d'installation et de suivre les instructions affichées sur l'écran.

6.2.3 Fonctionnalités

Dia permet de réaliser différents diagrammes tout en permettant de mélanger les différents composants des différentes catégories sans aucune restriction.

Les diagrammes générés peuvent être enregistrés puis modifiés à tout moment grâce à un format XML personnalisé propre à l'application mais il est également possible d'exporter les diagrammes sous différents formats comme EPS, SVG, XFIG, WMF ou encore PNG.

Possibilité pour l'utilisateur d'ajouter des formes personnalisées à inclure aux différents diagrammes. Un tutoriel détaillé est disponible à l'adresse suivante :

http://dia-installer.de/howto/create_shape/index.html

Une extension de fonctionnalité appelé Dia2code permet de générer partir des diagrammes UML conçus à l'aide de l'application du code PHP, Java, C, Python, C++, Ruby, C ou encore SQL. Cette extension est disponible à l'adresse suivante :

http://dia2code.sourceforge.net/index.html

6.2.4 Guide d'utilisation

L'utilisation de dia est relativement simple. Il suffit tout simplement de sélectionner la forme voulue puis de cliquer là où l'on veut la placer sur l'écran. Il est possible de redimensionner la plupart des formes, d'ajouter des commentaires ou encore de changer le contenu de la forme en effectuant simplement un double clic dessus.

En fonction de la forme choisie, un sous-menu s'ouvre avec un ou plusieurs onglets dans lesquels il est possible de renseigner plusieurs types d'informations. Par exemple pour une forme de type Classe, il est possible de renseigner le nom de la classe. Les cases à cocher en dessous de la case nommée Abstraite permettent de modifier l'affichage final sur le graphe. Veuillez noter que remove a été mal traduit par supprimer. Les deux cases sur la droite de la capture d'écran permettent de cacher les attributs et les opérations (méthodes ou fonctions).

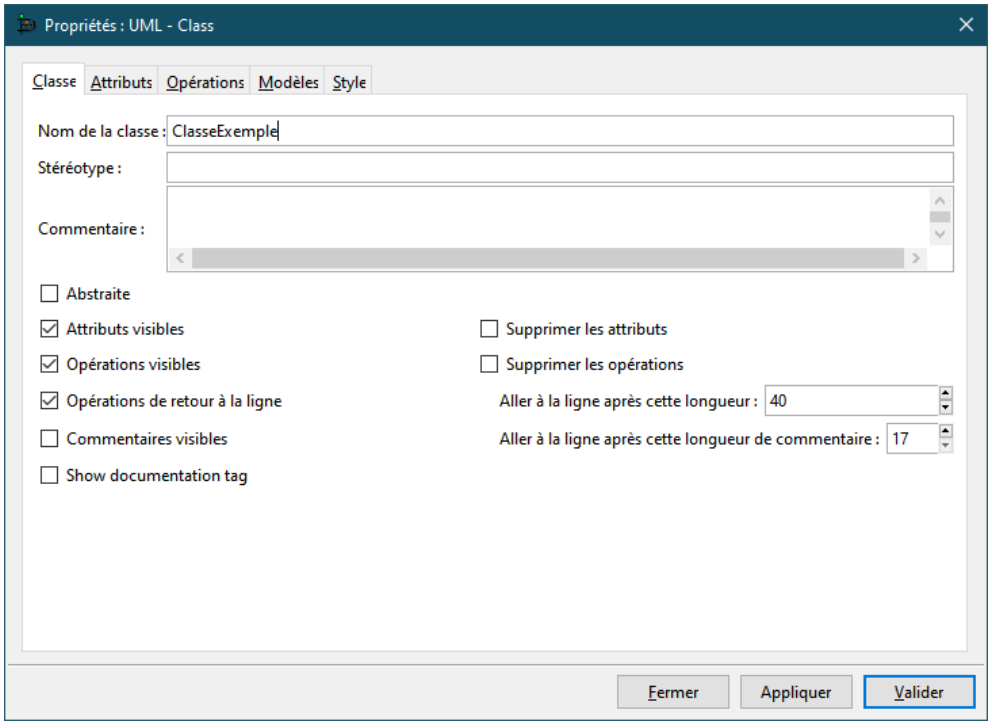


Figure 12: Onglet Classe permettant de saisir les informations générales d'une forme de type classe

Il est par la suite possible de renseigner les attributs de la classe, et de les ordonner dans l'affichage final, sur l'exemple ci-dessous, nous avons renseigné une nouvel attribut nommé nom de type String qui aura pour valeur par défaut exemple. Il est possible de choisir la visibilité de l'attribut saisi.

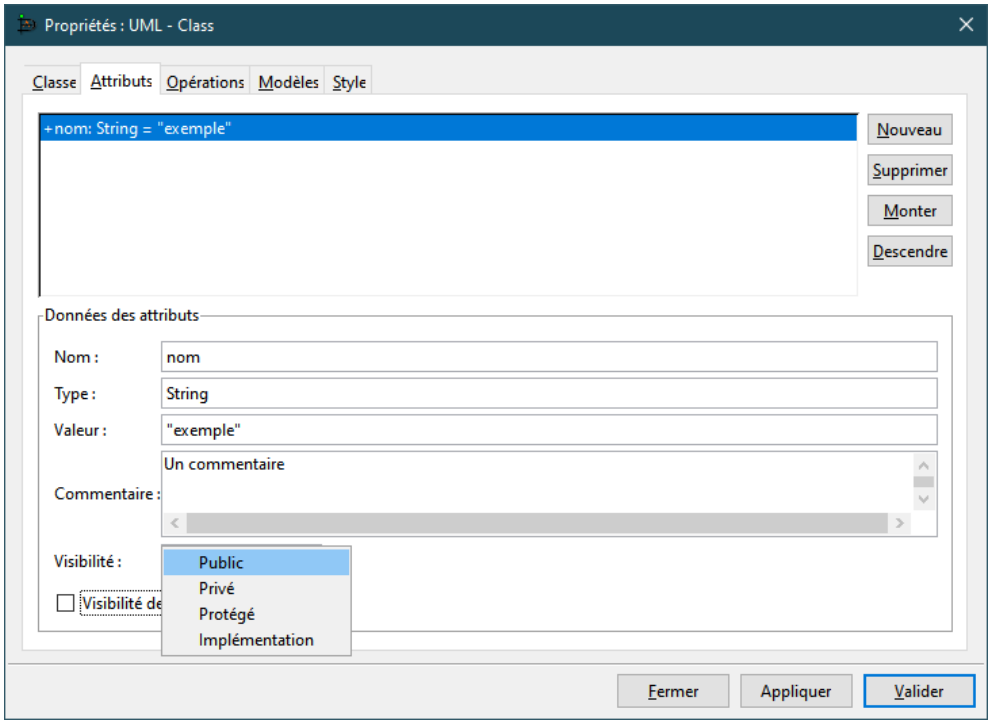


Figure 13: Onglet Attributs permettant de saisir les attributs d'une forme de type classe

Enfin, il est également possible de saisir les méthodes et les fonctions d'une classe. De la même manière que pour les attributs.

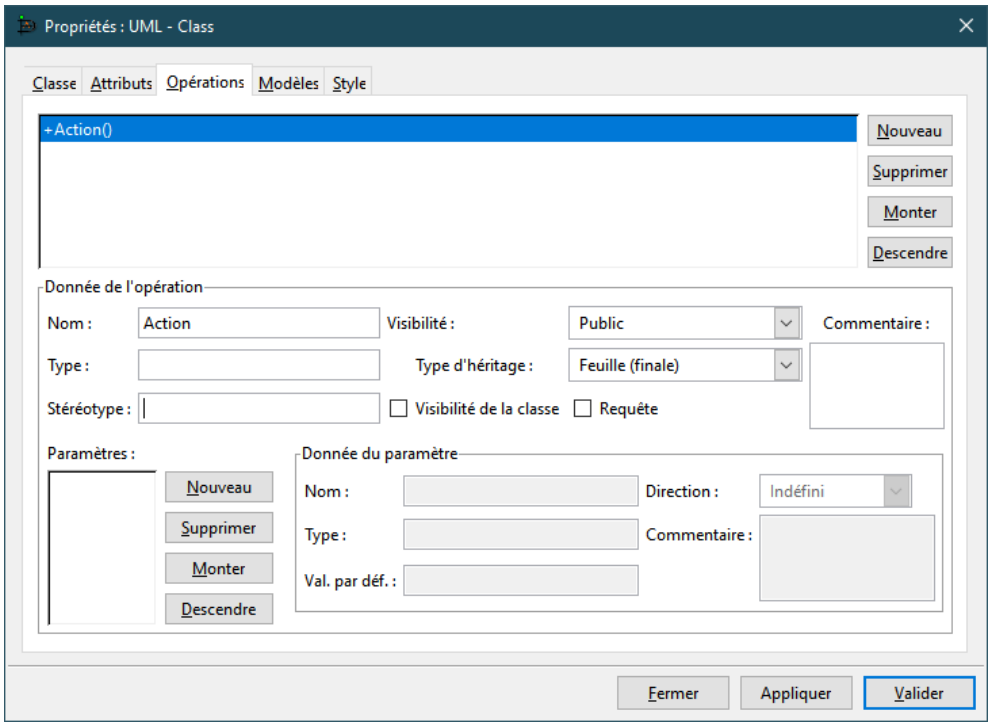


Figure 14: Onglet Opérations permettant de saisir les méthodes d'une forme de type classe

Pour finir les onglets Modèles (ou Template) et Style permettent de définir la notion de Template en C++ ou l'affichage final sur le graphe comme les couleurs ou la police d'écriture.

6.2.5 Les Points clés

Dia est une application portable pouvant être exécutée sur les systèmes d'exploitation principaux. Il dispose d'un large panel de diagrammes réalisables, allant du simple diagramme UML à la réalisation de circuits électriques tout en passant par la confection de puzzle tout en donnant la possibilité à l'utilisateur de créer ses propres formes afin de créer un éventail de possibilité encore plus large.

Une extension de fonctionnalité permet de générer du code à partir des diagrammes réalisés. Dia donne la possibilité à ses utilisateurs de réaliser des diagrammes de qualité et commentés. Pour finir, dia est facile d'utilisation et facile à prendre en main.

7 Compilation du projet

Pour compiler le projet, il faut se mettre à la racine du projet, et utiliser les commandes :

```
$ cd src/Model
$ find . -name "*.java" > sources.txt
$ javac @sources.txt
$ java Game
```

8 Conclusion

Au-delà des aspects techniques, la réussite de ce projet était de s'assurer que la simulation soit effective et fonctionne selon les contraintes que nous avions énoncées au départ. Nous avons obtenu une simulation fonctionnelle, et avons pu réaliser des statistiques sur cette simulation. Nous avons également pu implémenter 3 patrons de conception que nous avons jugé utiles lors de notre développement et pour un potentiel développement futur de notre simulation.