

**Rapport de conception**

**Projet Analyse, Conception et P.O.O.**

**4INFO G2.2B 2014/2015**

**Rapport de conception**

Projet Analyse, Conception et P.O.O.

*Jeu Small World*

Matry Bruno

El Alami Trebki Sami

**Matière :**

Modélisation Orientée Objet

**Professeurs :**

Arnaud Blouin

Eric Anquetil

Maud Marchal

Manuel Bouillon

Grégoire Richard

Remerciements

Nous remercions Eric Anquetil, Arnaud Blouin, Maud Marchal et Manuel Bouillon pour leurs cours et leurs explications sur la programmation et la modélisation orientée objet, ainsi que pour leurs conseils avisés qui nous ont permis la réalisation de ce projet.

Sommaire

Contenu

[Introduction 2](#_Toc372574984)

[I- Les commandes utilisateur 3](#_Toc372574985)

[1. Création de la partie 3](#_Toc372574986)

[2. Déroulement d’un tour 4](#_Toc372574987)

[II- Fonctionnement du logiciel 5](#_Toc372574988)

[1. Création de la partie 5](#_Toc372574989)

[2. Cycle de vie d’une unité 7](#_Toc372574990)

[3. Détail du déroulement d’une phase d’attaque 9](#_Toc372574991)

[III- Diagramme de classe 12](#_Toc372574992)

[1. Monteur Partie 12](#_Toc372574993)

[2. Strategie Partie 13](#_Toc372574994)

[3. Carte 14](#_Toc372574995)

[4. Cases 15](#_Toc372574996)

[5. Partie 16](#_Toc372574997)

[6. Joueur et Peuple 17](#_Toc372574998)

[7. Fabrique d’unités 18](#_Toc372574999)

[8. Unité 19](#_Toc372575000)

[9. Classe et énumérateurs auxiliaires 20](#_Toc372575001)

[Conclusion 21](#_Toc372575002)

# Introduction

Small World est « un jeu tour-par-tour où chaque joueur dirige un peuple. Le but du jeu est de gérer des unités sur une carte pour obtenir le plus de points possible à la fin d’un certain nombre de tours. Pour ce faire, chaque joueur commence avec des unités qu’il doit repartir sur la carte. Le placement de chaque unité rapporte plus ou moins de points. Les unités d’un joueur peuvent également attaquer les unités d’un autre joueur pour détruire des unités (limitant ainsi l’acquisition de points de l’adversaire) et occuper une case de la carte. Les points sont comptés à la fin de la partie, c.-à-d. après un nombre prédéfini de tours. Le jeu se déroule sur une carte du monde sur laquelle les unités se déplacent. »

Dans le cadre du projet de conception de ce jeu, nous avons établi ce rapport qui a pour objectif de modéliser la conception du logiciel en utilisant le langage graphique UML et donc de préparer la programmation de ce jeu.

L’analyse comporte ainsi l’étude des cas d’utilisation, un diagramme de classes représentant la modélisation globale du jeu, que nous avons réalisé à l’aide des patrons de conceptions étudiés, un diagramme d’état transition modélisant le cycle de vie d’une unité ainsi que des diagrammes d’interactions. L’ensemble de ces diagrammes nous servirons de point de départ lors de la phase d’écriture du code source de l’application.

Dans ce document nous ne détaillerons pas le fonctionnement de la partie graphique (Interface Homme-Machine) ainsi que l’implémentation des différentes méthodes.

# Les commandes utilisateur

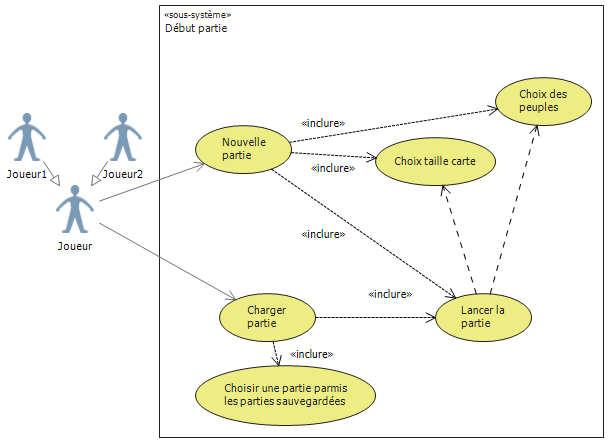
Les joueurs doivent bien entendu être en mesure d’interagir avec l’interface du jeu. Dans le cahier des charges proposé, tous les joueurs jouent sur le même poste. Il ne faudra pas faire communiquer plusieurs installations du logiciel.

On pourra distinguer deux phases d’interaction utilisateur dans la partie :

* La création de la partie
* Les différents tours de jeu

Nous allons donc détailler ces deux phases par le biais de diagrammes d’interaction.

## Création de la partie



*Figure 1. Diagramme d’interaction – Création d’une partie*

La figure 1 présente la première phase de jeu qui est le lancement de la partie. Les joueurs ont deux moyens de lancer une partie :

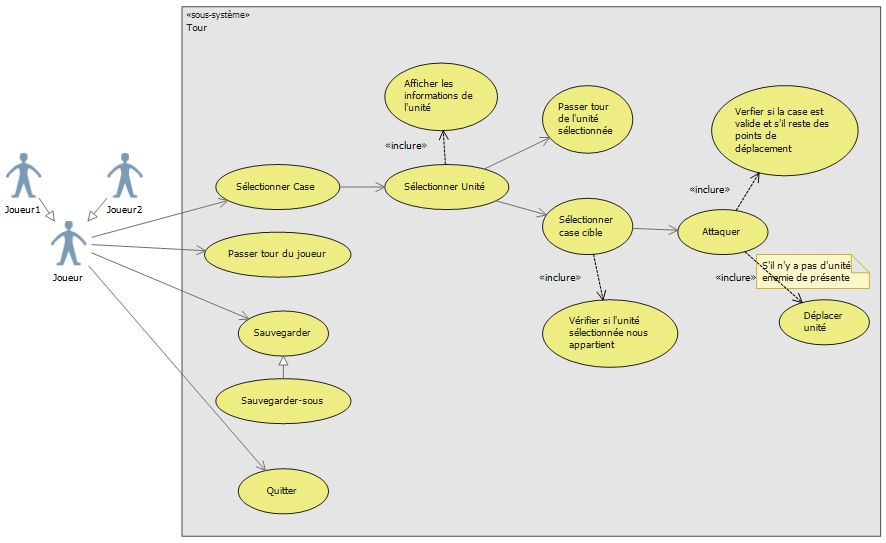
Ils peuvent lancer une nouvelle partie en réalisant le choix des peuples et le choix du type de jeu (demo, petit, normal) avec leurs caractéristiques associées :

* Démo : 2 joueurs, 5 cases × 5 cases, 5 tours, 4 unités par peuples.
* Petite : 2 joueurs, 10 cases × 10 cases, 20 tours, 6 unités par peuples.
* Normale : 2 joueurs, 15 cases × 15 cases, 30 tours, 8 unités par peuples.



Ils peuvent aussi choisir de charger une partie à partir d’un fichier de sauvegarde.

## Déroulement d’un tour



*Figure 2. Diagramme d’interaction – Déroulement d’un tour*

Lors du déroulement d’un tour, le joueur principal a le choix entre jouer, passer son tour, sauvegarder et quitter.

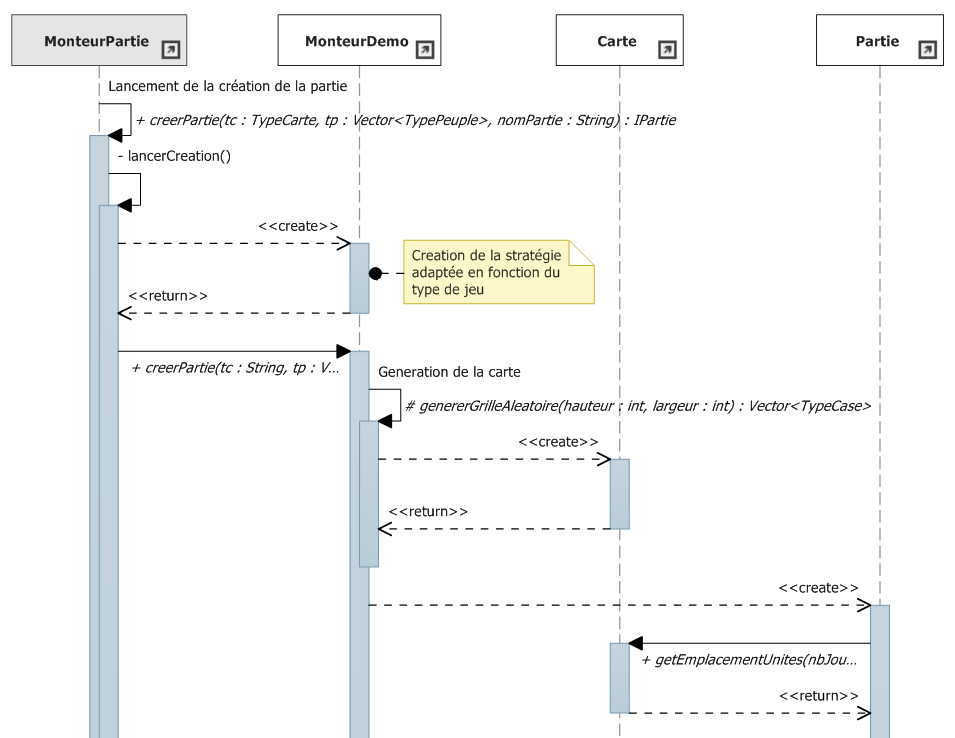
Si le joueur souhaite jouer, il choisit la case et l’unité qu’il souhaite contrôler (*Sélectionner Case* puis *Sélectionner Unité*). S’il y a une seule unité sur la case, elle sera sélectionnée par défaut. Lors de la sélection de l’unité le joueur a accès à ses informations : Point de vie, Point d’attaque, Point de défense.

Il peut alors juger s’il est judicieux de passer le tour de l’unité ou d’effectuer une action (*Attaquer* par défaut et *Déplacer l’unité* dans le cas ou la case ne contient pas d’ennemi), pour cela il doit sélectionner une case cible. En arrière-plan, le programme vérifiera si la case est valide pour l’action sélectionnée et si le joueur possède assez de points de déplacements, si c’est le cas l’action sera exécutée : si la case cible est une case ennemie, l’unité décidera d’attaquer, sinon l’unité effectuera un simple déplacement sur la case cible.

# Fonctionnement du logiciel

Pour simplifier la lecture des diagrammes de cette partie, nous emploierons le nom des classes et non le nom des interfaces même si ce sont ces dernières que nous manipulons dans les méthodes (voir partie III. Diagramme de classe).

## Création de la partie



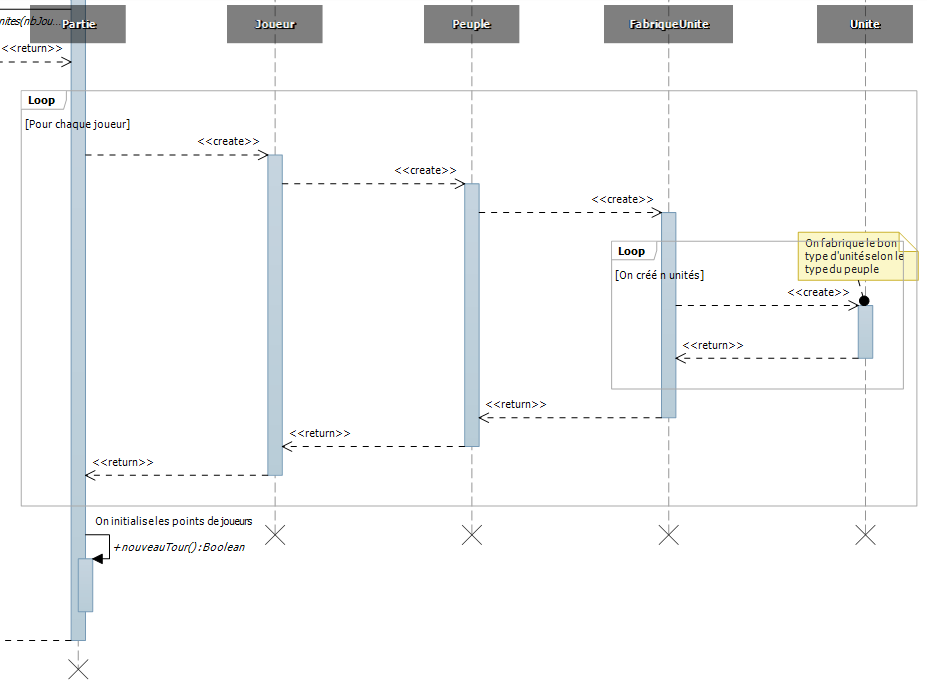
*Figure 3. Diagramme de séquence – Création de la partie (1/2)*

Lors de la création d’une nouvelle partie, on va faire appel à l’objet MonteurPartie qui va créer une partie soit :

* à partir d’un fichier
* à partir du type de la carte (représenté sous la forme d’un énumérateur), d’un tableau de type de peuple (à nouveau représenté sous la forme d’un énumérateur), et d’un nom de partie.

L’objet MonteurPartie va créer l’objet adapté selon le type de carte (utilisation du pattern Stratégie). Dans la figure 3, nous prenons l’exemple d’une carte demo. Nous créons donc un objet MonteurDemo. On appelle ensuite la méthode creerPartie sur l’objet MonteurDemo qui a été instancié.

Cet appel déclenche la génération aléatoire de la grille de la carte qui permet ensuite d’instancier un objet de la classe Carte. L’objet Partie est ensuite généré à partir de la carte qui lui sera passé en paramètre. Lors de son instanciation, la Partie va dans un premier temps appeler la méthode getEmplacementUnites de la Carte afin d’obtenir les coordonnées idéales pour placer les unités des joueurs.



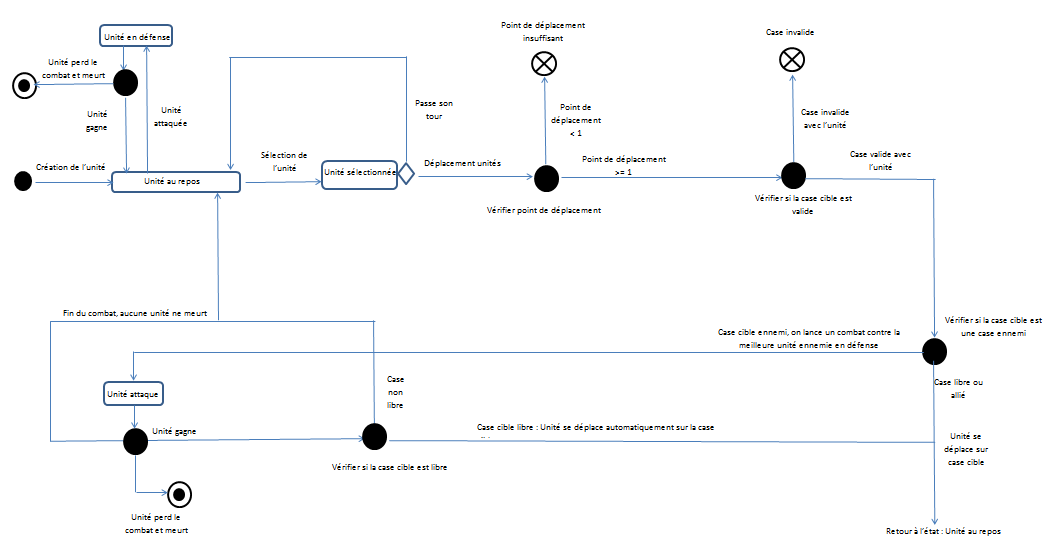
*Figure 4. Diagramme de séquence – Création de la partie (2/2)*

La partie va ensuite instancier chaque joueur qui va instancier le peuple qui lui est associé qui va instancier une fabrique unité qui elle-même va créer un nombre donné (selon la taille de la carte) d’unités du bon type (selon le peuple choisi).

Toujours pendant son instanciation, la classe partie va appeler la méthode nouveauTour afin d’initialiser les points des unités (selon le type de case sur lequel ils se trouvent).

## Cycle de vie d’une unité

Dans ce diagramme d’états-transition, nous décrivons le cycle de vie d’une unité :



*Figure 5. Diagramme d’état-transition – Cycle de vie d’une unité*



A la création de l’unité, elle est à l’état « Au repos », le joueur a donc 3 choix pour modifier cette état: Passer son tour, déplacer l’unité sur une case libre, attaquer une unité ennemie (ce qui équivaut à lancer un déplacement sur une case ennemie sur le diagramme).

Si le joueur choisit de passer son tour, l’unité reste au repos, aucun changement d’état n’a lieu.

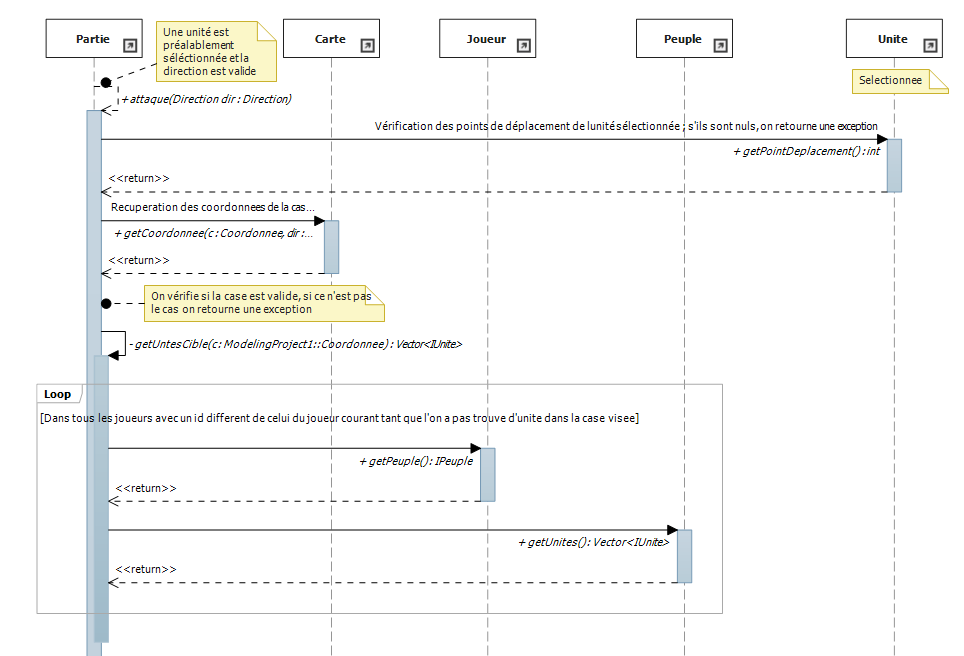
Si le joueur choisit de déplacer une unité sur une case libre, on vérifie que le joueur possède assez de points de déplacement et que la case cible est valide pour l’unité (Par exemple, l’unité Viking a la capacité de se déplacer sur l’eau alors que les autres ne peuvent pas) ensuite on vérifie que la case cible est libre, une fois ces trois conditions vérifiées, l’unité se déplace sur la case cible puis revient de l’Etat « sélectionnée »  à l’état « au repos ».

Si le joueur choisit d’attaquer, ce qui revient à déplacer l’unité sur une case ennemie, on fait les même vérifications que pour les déplacements, ensuite on vérifie que la case cible est bien une case ennemie. Si l’unité gagne, on vérifie qu’il ne reste pas d’autres ennemis sur la case cible si c’est le cas l’unité se déplace automatiquement sur la case vaincue. Si l’unité perd le combat, l’unité est détruite. Si aucune unité ne meurt, on repasse à l’état « au repos ».

L’unité peut également être attaquée, dans ce cas-là l’unité passe de l’état « au repos » à l’état « défense », si l’unité perd elle est détruite, sinon elle repasse à l’état « au repos ».

Lorsqu’on entre dans un cas d’erreur, l’unité retourne à l’état « au repos ».

## Détail du déroulement d’une phase d’attaque

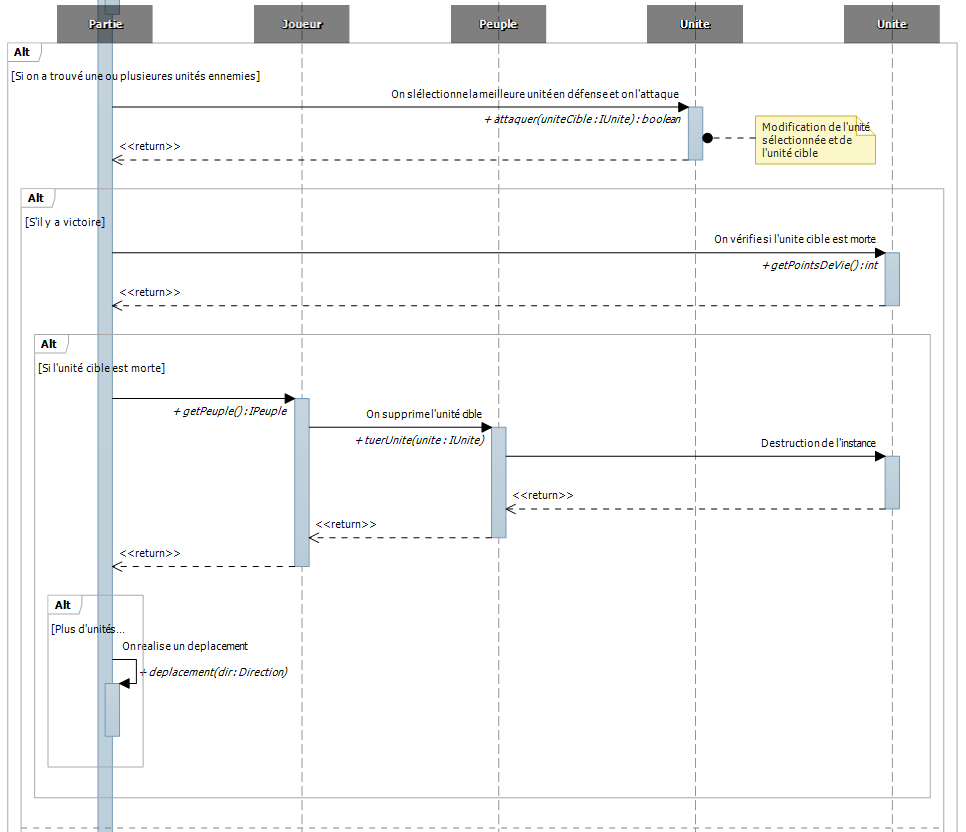


*Figure 6. Diagramme de séquence – Déroulement d’une phase d’attaque (1/3)*

Pour réaliser une attaque il suffit d’appeler la méthode *attaque(Direction)* en lui indiquant la direction de l’attaque.

Les conditions nécessaires pour effectuer une attaque sont :

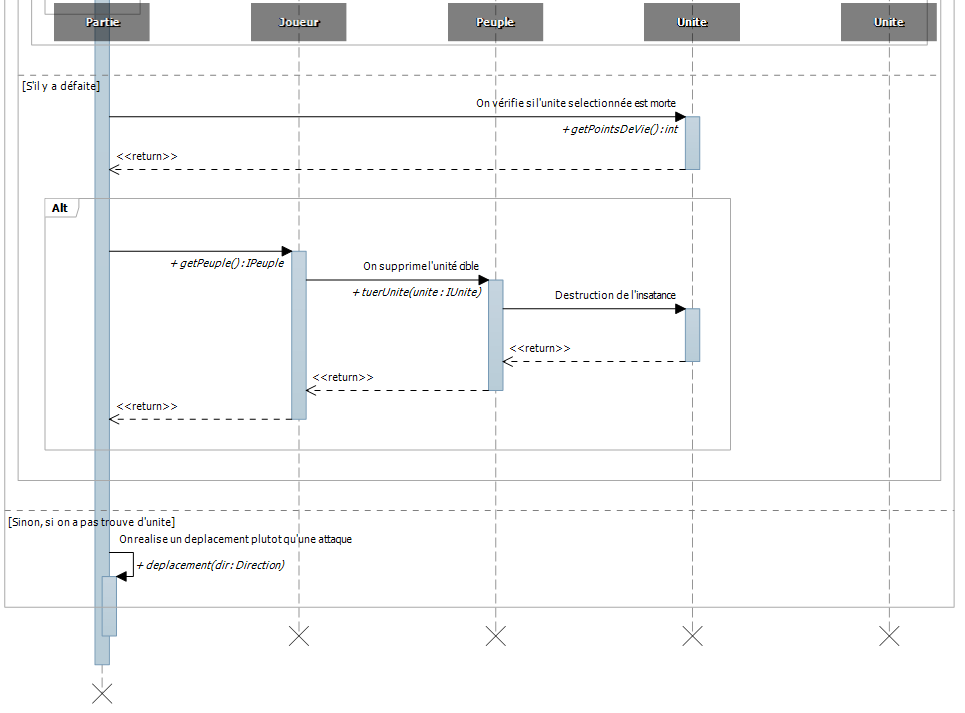
* Les points de déplacements qui doivent être supérieurs ou égal à 1 ; ce que l’on obtient par la méthode *getPointDeplacement()*
* La case cible qui doit être une case valide, c’est-à-dire que les unités ennemies doivent se situer sur des cases juxtaposées ; ce que l’on vérifie par la méthode *getCoordonnee(coordonneesActuelles, direction)* qui rend les coordonnées de la case cible.

**

*Figure 7. Diagramme de séquence – Déroulement d’une phase d’attaque (2/3)*

Une fois des unités ennemies détectées sur la case cible, on sélectionne la meilleure unité en défense et on l’attaque en utilisant la méthode *attaquer(uniteCible)*. Si on gagne, on vérifie que l’unité cible est morte, en utilisant la méthode *UniteCible.getPointsDeVie()* et en vérifiant si elle retourne 0, si c’est le cas on la détruit avec la méthode *tuerUnite(uniteCible)*.

Puis on vérifie si la case cible est libre, si c’est le cas l’unité sélectionnée se déplace automatiquement dessus en appelant la méthode *déplacement(Direction)*.



*Figure 8. Diagramme de séquence – Déroulement d’une phase d’attaque (3/3)*

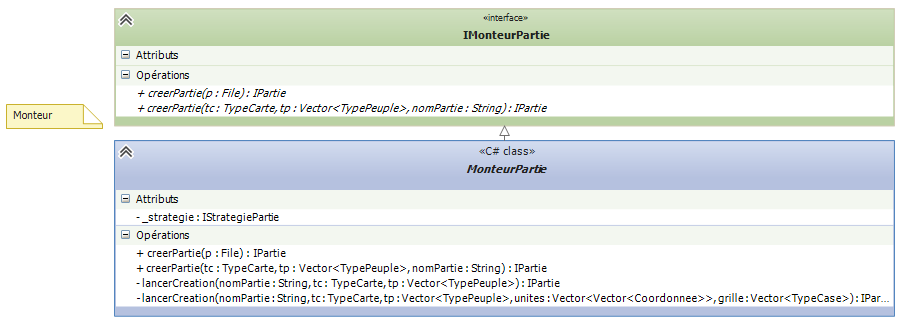
Si on perd, on vérifie que l’unité sélectionnée est morte ou non ; si c’est le cas on la détruit en utilisant la même méthode que précédemment.

Si l’attaque a lieu sur une case vide ou allié, on réalise un déplacement sur la case cible.

# Diagramme de classe

Les classes du diagramme vous seront présentées en suivant une certaine logique : leur ordre d’apparition lors de la création d’une nouvelle partie.

## Monteur Partie



*Figure 9. Extrait du diagramme de classe (1/9) – MonteurPartie*

Nous avons fait ici (Figure 9) le choix d’utiliser un patron de conception du type *monteur* pour créer la partie afin de séparer la création de la partie de sa représentation (la partie est un objet complexe).

L’unique méthode accessible pour le Monteur (utilisation d’une interface *IMonteurPartie*) est la méthode *creerPartie* qui a été surchargée afin de pouvoir charger une partie sauvegardée auparavant ou en créer une nouvelle.

Lors de la création d’une partie, on appelle la méthode privée (et surchargée) *lancerCreation* qui :

* dans le cas d’un chargement de partie enregistrée ; après interprétation du fichier par la méthode *creerPartie* chargera les différents paramètres de la partie sauvegardée.
* dans le cas de la création d’une nouvelle partie, lancera simplement la génération d’une nouvelle partie avec une carte aléatoire en créant la *StrategiePartie* adaptée (voir extrait de diagramme suivant).

## Strategie Partie



*Figure 10. Extrait du diagramme de classe (2/9) – StrategiePartie*

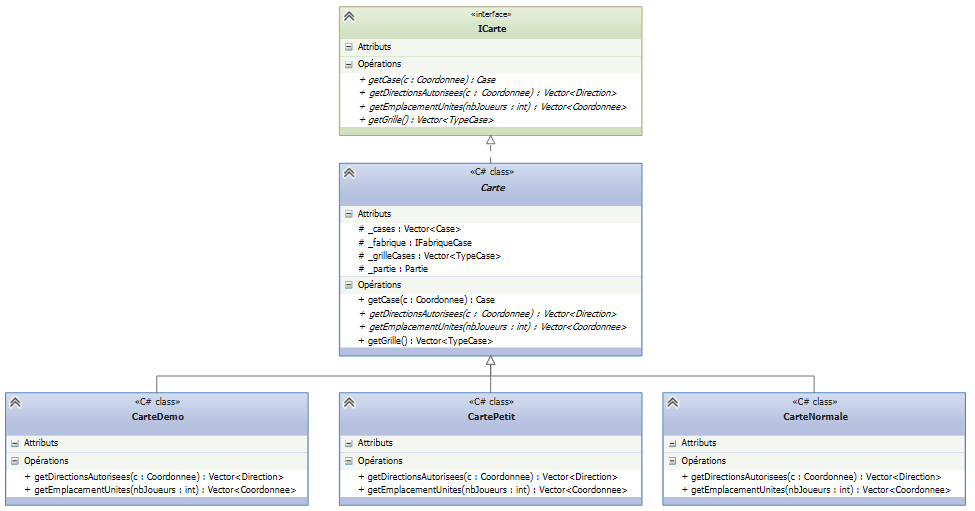
Le choix de l’utilisation d’un patron de conception de type *stratégie* (Figure 10) afin de pouvoir réaliser dynamiquement le montage d’une carte selon le type recherché (demo, petite, grande).

Ces monteurs implémentent aussi une méthode *creerPartie* afin de permettre une plus grande généricité de notre code. En effet, si l’on créé un nouveau type de partie avec plus de 2 joueurs, il suffira simplement d’implémenter un nouveau monteur. Ce qui ne nécessitera pas de modifier l’ensemble de notre code.

Ces monteurs gèrent donc le montage de la partie mais surtout le montage de l’objet *Carte*.

Ici encore, on ne manipulera que l’interface *IStrategiePartie* et nous ne ferons le choix de l’instanciation d’un objet concret qu’au dernier moment afin de minimiser les modifications dans les différentes classes en cas de modification de l’implémentation de la classe *StrategiePartie* ou de ses classes filles. Nous suivrons cette logique pour l’ensemble du projet.

## Carte



*Figure 11. Extrait du diagramme de classe (3/9) – Carte*

Dans la Figure 11, nous voyons à nouveau apparaître une interface *ICarte* qui servira à manipuler les objets concrets *CarteDemo*, *CartePetit* et *CarteNormale*.

Les objets implémentant l’interface carte doivent être en mesure de donner la *Case* qui se situe à une coordonnée donnée (retourne une exception si cette coordonnée est hors de la carte, ce qui ne devrait normalement jamais arriver puisqu’on ne permet pas au joueur de pointer une case hors de la carte).

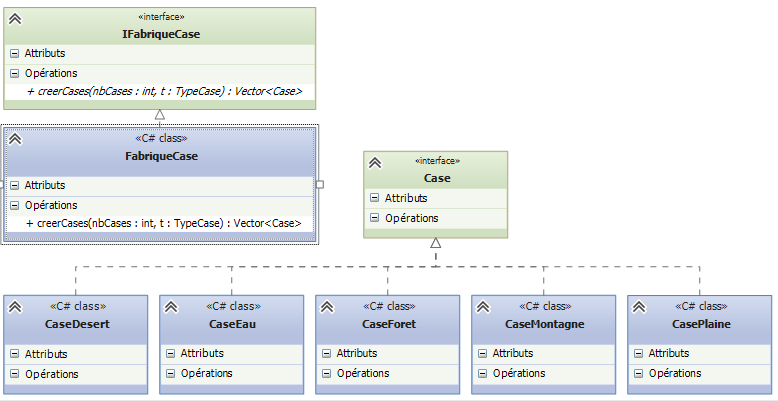
Ils doivent aussi être en mesure de fournir les directions autorisées en fonction d’une coordonnée donnée (attention, cette fonction ne garantie pas complètement la validité de la case cible puisque certaines unités ne peuvent pas se déplacer sur l’eau par exemple).

Pour le placement optimal des unités en début de partie, la fonction *getEmplacementUnites* retourne les coordonnées optimales de placement des unités en début de partie en fonction du nombre de joueur. Nous utiliserons le polymorphisme afin d’adapter ce placement à la taille des cartes.

Enfin, l’accesseur *getGrille* permettra la récupération de la grille des cases sous la forme d’un tableau de *TypeCase* (énumérable).

Nous utilisons ici un patron de conception de type *poids mouche* afin d’éviter la création d’un nombre important d’instances des différentes classes filles de la classe *Case* (voir diagramme suivant).

## Cases

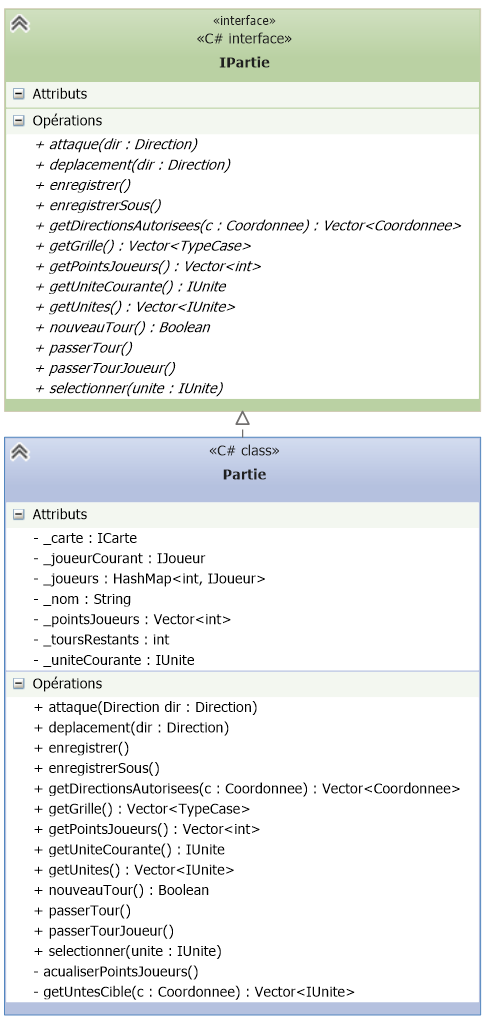


*Figure 12. Extrait du diagramme de classe (4/9) – Case*

Les cases seront générées par le biais d’une fabrique modélisée dans la figure 12 par l’interface *IFabriqueCase* et la classe *FabriqueCase*. Ces fabriques généreront un nombre *nbCases* du type *t* à l’aide de la méthode *creerCases*.

L’ensemble des objets concrets cases (*CaseDesert*, *CaseEau*, …) implémente directement l’interface *Case*.

## Partie



L’interface *IPartie* et l’objet *Partie* (figure 13) représente en quelque sorte le noyau du jeu.

La totalité des actions des joueurs peut être réalisée en passant par cette classe.

La sélection d’une unité, l’attaque ou le déplacement dans une direction (qui pourra éventuellement être surchargée pour prendre en paramètre une coordonnée en plus de la méthode permettant de prendre en paramètre une direction), le passage du tour par l’unité, le passage du tour par le joueur, l’enregistrement de la partie (et l’enregistrement-sous).

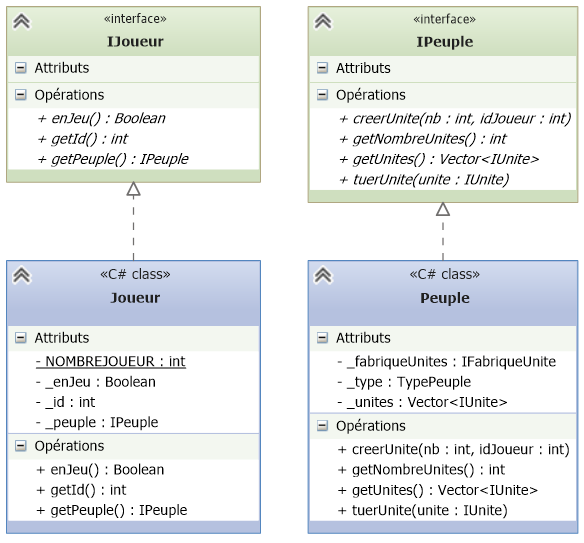
Des getters sont aussi présents afin de récupérer la grille, les points des joueurs, l’unité courante, et l’ensemble des unités. Ces getters seront utiles lorsqu’il faudra réaliser l’affichage de la partie.

Notez que l’on pourra aussi implémenter une méthode *getUniteSuivant*eafin de faciliter la sélection d’unité par les joueurs.

*Figure 13. Extrait du diagramme de classe (5/9) – Partie*

*Figure 13. Extrait du diagramme de classe (5/9) – Partie*

## Joueur et Peuple

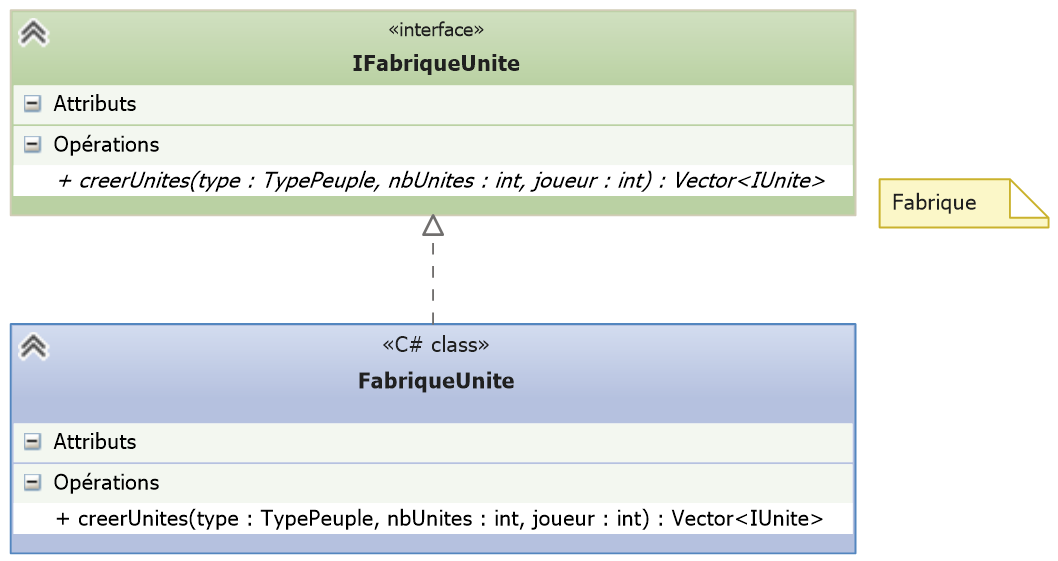


*Figure 14. Extrait du diagramme de classe (6/9) – Joueur et Peuple*

La classe *Joueur* (manipulée par le biais de l’interface *IJoueur*), contiendra un identifiant de joueur afin de différencier les différents joueurs ainsi qu’une instance de *Peuple* (manipulée par le biais de l’interface *IPeuple*). La méthode *enJeu* permet de vérifier si le joueur est encore en jeu ; cette méthode sera particulièrement utile dans le cas d’une partie où le nombre de joueur sera supérieur à 2.

La classe *Peuple* contiendra une instance de *FabriqueUnite*, son type, ainsi qu’un tableau d’unités (qui sera instancié par la fabrique, voir diagramme suivant). Cette classe sera en mesure de donner le nombre d’unités qu’il lui reste, de tuer une unités (suppression de sa liste d’unités), de donner les unités qu’elle contient ainsi que de créer les unités (par le biais de la *fabrique*) à partir de l’id du Joueur.

## Fabrique d’unités

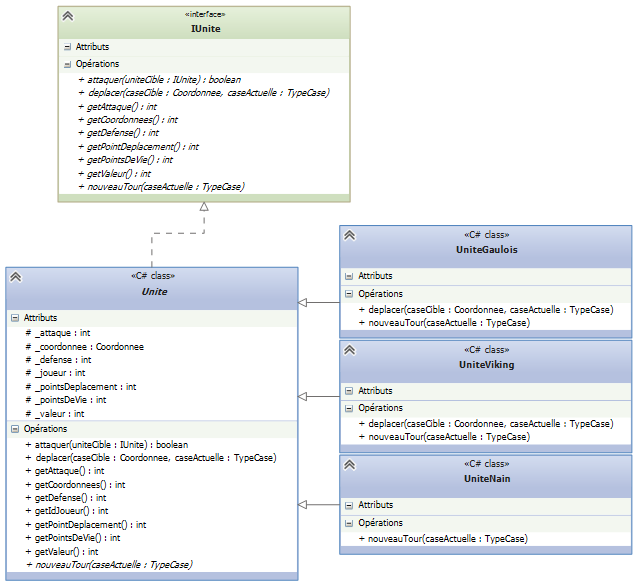


*Figure 15. Extrait du diagramme de classe (7/9) – Fabrique Unités*

Comme son nom peut le laisser penser, la *FabriqueUnite* (figure 15) est un patron de conception de type *fabrique*.

Cette fabrique construit un nombre *nbUnites* d’unités de type *type* associées au joueur d’id *joueur*.

## Unité



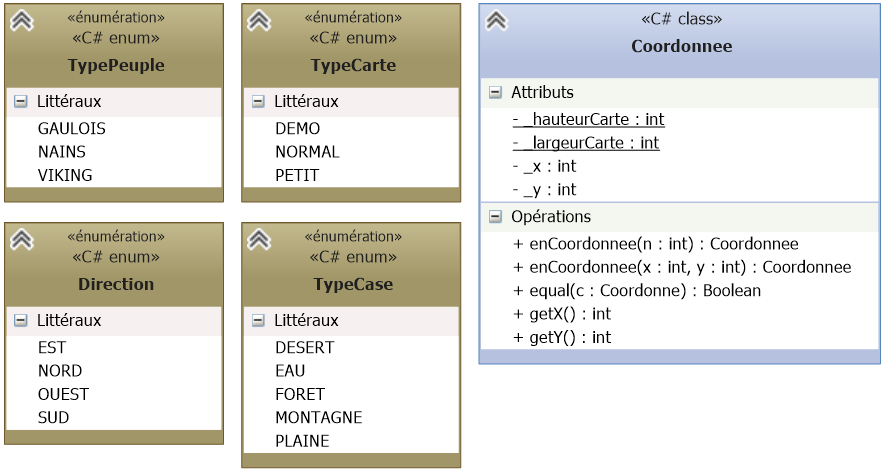
*Figure 16. Extrait du diagramme de classe (8/9) – Unité*

La classe *Unité* (figure 16) est encore une fois manipulée par le biais d’une interface (*IUnite*). Les classes filles *UniteGaulois*, *UniteViking*, et *UniteNain* utilisent le polymorphisme sur les méthodes *déplacer et NouveauTour* car ce sont les deux seules méthodes qui sont dépendantes du type de l’unité.

Pour la plupart des attributs, on a des getters qui serviront à l’affichage des attributs de l’unité lorsqu’elle sera sélectionnée. On dispose aussi d’un getter pour les coordonnées de l’unité qui permettra de placer plus facilement les instances d’*Unite* sur la *Carte*.

La méthode *attaquer* permettra de lancer une attaque contre une unité ciblée, la méthode *déplacer* sert, bien entendu, au déplacement de l’unité et la méthode *nouveauTour* à la mise à jour des champs internes de l’unités (points de déplacement, valeur, …).

## Classe et énumérateurs auxiliaires



*Figure 17. Extrait du diagramme de classe (9/9) – Classe et énumérateurs auxiliaires*

La classe *Coordonnee* contiendra en attribut *static* la hauteur et la largeur de la carte. On pourra aussi utiliser la méthode *static* surchargée *enCoordonnee* afin d’obtenir une instance de *Coordonnee* en fonction d’un entier ou d’une paire d’entiers.On aura aussi des getters pour obtenir l’attribut *x* et l’attribut *y*.

Les énumérateurs permettront de simplifier la lisibilité du code.

# Conclusion

Ces diagrammes nous serviront donc de base pour implémenter notre code. Le diagramme de classe nous permet même de générer automatiquement le squelette de notre code.

Bien entendu, ils seront modifiés de façon mineure lors de l’implémentation afin de rajouter des fonctionnalités par exemple.