INSA de Rennes

Programmation Orientée Objet

Projet C++

Rapport de conception

Author :
Jean Guegant
Richard Lagrange

Supervisor :
Alexandru Costan

8 novembre 2012

Table des matières

1 Introduction

Lors de la modélisation de notre version de *Civilization*, nous avons voulu rendre notre code le plus flexible et extensible possible. Nous voulions pouvoir facilement ajouter la gestion des jeux en réseau, pouvoir étendre et créer ses propres classes, etc. Heureusement, ceci est réalisable grâce aux *design patterns* (qui nous sont aussi imposés). Nous avons essayé d'aller plus loin que les 5 patrons de conceptions imposé, afin de réaliser notre objectif d'un jeu pouvant évoluer. Nous justifierons nos différents choix techniques tout au long de ce rapport. Bonne lecture.

2 Diagrammes de cas d'utilisation

2.1 Règles du jeu

Dans ce diagramme, nous avons illustré les intéractions entre les concepts du jeu et les joueurs. (Schéma 3) Les joueurs sont ceux qui créent des villes, créent des unités, bougent, font combattre les unités et capturent des villes. Les dépendances entre les entités du jeu apparaissent le plus clairement possible.

2.2 Cycle de vie d'une partie

Ce diagramme illustre toutes les phases du logiciel, avec les choix potentiels des joueurs.(Schéma 2)

3 Diagrammes de Classe

Nous avons commencé par modéliser l'ensemble des interfaces des entités que nous voudrions manipuler. Ceci nous a permis de ce concentrer sur l'interaction des objets sans se préoccuper de leurs implémentations. Par la suite, nous avons implémenté les différentes interfaces par des classes concrètes. A noter que dans certains cas particulier nous avons regroupé interface et une partie de l'implémentation sous forme de classe abstraite. Ci-dessous vous trouverez les différents les diagrammes de classes finaux de notre application regroupées par espace de noms.

3.1 Civilization : schéma 4

3.1.1 Namespace Civilization

Nous avons utilisé le design pattern Prototype pour pouvoir avoir le plus possible de civilizations différentes. En effet, au départ nous étions partis sur le patron de conception Fabrique Abstraite classique, ce qui aurait donné le Schéma 3. C'est à dire implémenter une classe concrète pour chacune des civilisations. Après réflexion, nous voulions pouvoir étendre facilement le nombre de civilisations (tout joueur n'est pas forcement suffisamment émérite en C# pour pouvoir rajouter sa propre classe). Le joueur pourrait importer sa civilisation créée, sous forme d'un fichier XML par exemple. Donc pour implémenter cette solution, nous avons utilisé une Fabrique Concrète utilisant des prototypes d'unités différents à cloner lorsqu'il désire utiliser telle ou telle civilisation. (Schéma 4) Cette Fabrique Concrète a donc un constructeur qui a comme rôle de récupérer des unités clonables spécifiques à une civilisation (Directeur, étudiant, etc.), pour en fabriquer d'autres, avec des caractéristiques uniques. Ces instances particulières des unités peuvent provenir de 2 endroits :

1. Soit l'utilisateur les a crée lui même, via une interface utilisateur, qui créera un fichier XML contenant le prototype unité sérialisé (l'interface ISerializable de C# permet de définir cette possibilité de sérialisation).
2. Soit ce sont des unités prédéfinies que l'on aura créées auparavant. (Il faudra alors désérialiser l'unité prototype, qui est stockée sous la forme d'un fichier XML comme pour la 1ère méthode.)

Le couplage du pattern Fabrique Abstraite avec le pattern prototype nous permet d'éviter un nombre de sous-classes important. Toutefois il est toujours possible de coder des civilisations en dur en implémentant un autre type de fabrique concrète.

3.1.2 Namespace Unit

Une interface commune IUnit est pratique dans le cas où l'on déciderais de faire une unité qui a une attaque qui dépend de plusieurs facteurs. (Comme par exemple : attaque = attaque de base + attaque de l'épée + attaque du compagnon). Une interface commune permet de réaliser ceci facilement. La classe Unit est *ISerializable* (comme expliqué auparavant) et *IDrawable* (pour avoir les méthodes d'affichage). On a aussi des interfaces spécifiques pour les types d'unités spéciales (IDirector, IStudent, ITeacher), ce qui nous donne une certaine flexibilité. Si jamais l'on décide de changer la gestion des bonus des directeurs de département par exemple, cette interface permettra de le changer sans toucher au code client.

3.1.3 Namespace City

De même que pour IUnit, nous avons pensé à une interface ICity (qui pourrait se revéler pratique si nous voulons étendre un autre concept à des villes de type différent). Par exmple, nous pourrions imaginer une ville dont *TotalAvailableFood* dépend aussi de son commerce avec une autre ville par exemple, nous pouvons changer la manière dont est implémenté cette méthode à chaud sans risquer de changer le code client. Ensuite, nous avons utilisé le

pattern Stratégie afin d'implémenter des algorithmes différents d'extension de la ville (*"Le choix de cette case, sera réalisé par un algorithme que vous développerez."*) Ainsi, on pourra au départ implémenter un algorithme simple et bête qui choisira une case aléatoirement. Puis dans un deuxième temps, choisir d'implémenter un algorithme plus intelligent, qui choisira les case en fonction de leur contenu exploitable (minerai, nourriture). De plus, vu qu'une seule instance de cet algorithme est utile à la fois, nous utiliseront le patron de conception Singleton (le pattern Singleton se retrouve dans toutes les patterns algorithme de notre projet).

3.2 View : schéma 5

3.2.1 Namespace Drawable

TileFlyWeight est un pattern poids-mouche (flyweight) pour les tiles (textures) que nous allons utiliser pour la vue de notre jeu. En effet, de nombreuses cases de la carte de notre jeu ainsi que les différentes unités utilisent les mêmes textures. Afin d'éviter d'encombrer inutilement la mémoire en chargeant plusieurs fois la même texture, nous allons partager une même instance de cette texture aux différents objets la nécessitant. Pour cela, notre pattern flyweight contient une hashmap liant un nom de texture à son contenu. Par exemple si plusieurs objets demandent le tile "water.png", ceux-ci partageront une unique instance du tile "water.png". A noter que si aucun objet ne demande "water.png", la texture ne sera jamais chargé en mémoire. On pourrait se demander si il n'est pas plus judicieux de partager les objets cases de notre carte et pas uniquement leurs textures. En effet nous pourrions décider que toutes les cases "moutain" soit partagées car elles contiennent les mêmes caractéristiques (hormis leur position, une caractéristique qui serait alors passé en état externe lors des manipulations). Nous projetons de créer des cases avec des caractéristiques variant au cours du temps (par exemple, la nourriture diminue en cas de guerre de proche), il est alors difficile envisageable de contenir toutes les caractéristiques en état externe. C'est dans cette optique que nous ne partageons uniquement les textures des cases. Notre poids-mouche est un singleton car il n'y aucune raison de l'instancier plusieurs fois au cours d'une même partie. (hormis leur position, une caractéristique qui serait alors passé en état externe lors des manipulations). Nous projetons de créer des cases avec des caractéristiques variant au cours du temps (par exemple, la nourriture diminue en cas de guerre de proche), il est alors difficile envisageable de contenir toutes les caractéristiques en état externe. C'est dans cette optique que nous ne partageons uniquement les textures des cases. Notre poids-mouche est un singleton car il n'y aucune raison de l'instancier plusieurs fois au cours d'une même partie.

La classe Tile a son constructeur privé car nous voulons que les Tiles soit chargés uniquement à partir du FlyWeight. On remarquera également que la méthode *Draw* prend un paramètre d'état externe qui est la position du tile que nous voulons dessiner (comme vu précédemment le pattern FlyWeight oblige à passer les états non-partagés d'un objet à l'aide de paramètres externes).

Nous avons mit à la disposition des développeurs une interface ISprite pour dessiner des animations. Un sprite consistant en une série de textures que l'on fait défiler afin de donner l'illusion d'une animation. L'implémentation de cette interface est faite avec la classe Sprite qui gère des animations basiques. Encore une fois l'intérêt d'une interface ISprite est de pouvoir facilement changer l'implémentation sans réécrire tout le code client.

Pour la vue, nous avons également l'interface IDrawable qui doit être implémenté par toutes les entités que nous voudrons dessiner (les cases, les unités, ...). Cette interface contient deux fonctions :

- Draw qui permet de dessiner l'entité, le paramètre IGraphiqueEngine sera détaillé par la suite.
- Update qui met à jour l'état graphique de l'entité que nous allons dessiner. En effet, si nous utilisons des animations, il est important de modifier l'état de la prochaine frame que nous allons dessiner. Le paramètre deltaTime est la différence de temps entre la frame précédente et la frame courante. Nous exprimeront l'ensemble de nos animations à l'aide de ce paramètre, ainsi peu importe la puissance de la machine sur laquelle tourne le jeu, les animations seront toujours de même durée et donc fluides.

Nous espérons implémenter par la suite un pattern bridge pour séparer l'interface de notre moteur graphique de son implémentation. En effet, il serait élégant de pouvoir faire tourner notre jeu en dehors de l'environnent WPF (XNA, DirectX, OpenGL par exemple). Pour cela nous avons imaginé une interface IGraphicEngine qui contiendra les appels graphiques nécéssaires au fonctionnement de notre jeux (DrawSprite, CreateSprite, LoadTile, ...). L'implémentation dans le cas de WPF est la classe WPFGraphicEngine, pour XNA se sera XNAGraphicEngine, ... Dans toutes les librairies graphiques nous retrouvons des appels à des primitives graphiques identiques(DrawTexture, LoadPNGTexture, ...). L'interface IGraphicEngine dépend donc de son implémentation bas-niveau qui est l'interface

IGraphicPrimitive qui contient l'ensemble de ces appels primitifs. Pour WPF nous créons alors une classe WPFGraphicPrimitive pour implémenter cette interface IGraphicPrimitive. Il en va des même pour les autres environnements que nous détirons utiliser. Ce pattern bridge ne contient pour l'instant aucune méthode car nous ne sommes pas encore certains des fonctions graphiques nécessaires à notre jeu.

3.3 World : schéma 6

3.3.1 Namespace World

3.3.2 Namespace Square

Dans le namespace Square, nous avons créé une classe pour représenter une case du "damier" de notre jeu. Ces cases peuvent être de natures différentes (Montagne, Plaine, Désert) et avoir des ressources spéciales (Fer, Fruit). Grâce à ces natures et ressources spéciales, on peut avoir de nombreux types de cases différents. Une implémentation idéale de ce cas serait le patron de conception Décorateur. En effet, grâce au décorateur, on peut alors "décorer" les cases Montagne, Plaine ou Désert avec les ressources spéciales Fer et Fruit. Il est aussi facile d'ajouter une nature ou une ressource spéciales, sans tout de même faire exploser le nombre de classes filles.

3.3.3 Namespace Map

L'interface IMapCreate permet d'obliger les classes de type SmallMap ou MediumMap qui l'implémentent d'avoir une méthode CreateMap. Cette architecture peut sembler étrange par la présence du paramètre randomizer. En fait, il s'agit du patron de conception Stratégie appliquée à une interface. Le lien entre l'interface IMapCreate et ISquareRandomizer n'est pas aussi fort qu'une agrégation, mais doit quand même avoir un randomizer passé en paramètre. Il est alors facile d'avoir des stratégies différentes de création de cartes, si on veut des cartes avec plus de mer par exemple, ou plus de montagne. Un autre patron de conception Stratégie se cache derrière SmallMap et MediumMap. La fonction *CreateMap* renvoie une Map, qui sera différente en fonction de la taille (en plus du randomizer). En effet, dans une grande carte on pourrait avoir plus de cases d'eau, chose qu'on ne pourrait se permettre d'avoir quand on a une petite carte.

3.4 Fight : schéma 7

3.4.1 Namespace Fight

Dans le Namespace Fight, nous avons encore une fois utilisé le patron de conception stratégie. Elle nous permet de choisir quel algorithme de combat utiliser. En effet, nous avons imaginé plusieurs algorithmes différents en fonction de différents paramètres :

1. Le combat avec un Directeur de département sur la case comme support.
2. Le combat normal d'une unité contre une autre.
3. Le combat dans une ville (des fortifications doivent sûrement apporter un avantage défensif).

L'intérêt d'une interface IFight est que si l'on veut un combat entre des groupes d'unités (en non plus du 1v1), on pourra implémenter cette interface par une autre classe. On aurait alors les méthodes utiles comme *addDefender()* ou *addFighter()*, qui n'interdisent pas l'ajout de plusieurs unités.

3.5 Game : schéma 8

3.5.1 Namespace Game

Game est la classe centrale de notre jeu. Nous pouvons la considérer comme un pattern médiateur car elle va gérer les communications (la logique) entre les objets de notre jeu. Cette classe implémente l'interface ISerializable afin de pouvoir enregistrer l'état de la partie dans un fichier de notre de choix et de pouvoir le recharger ultérieurement. Nous pourrons donc mettre à disposition un système de sauvegarde pour les joueurs. La fonction MainLoop de notre classe Game est la boucle principale du jeu qui mettra à jour l'état de la partie puis dessinera celle-ci. Si nous voulons changer le comportement de cette boucle, il est toujours possible d'étendre la classe Game et de redéfinir la fonction MainLoop.

Pour gérer les différentes phases du jeu, nous utiliserons le pattern State. Pour cela nous avons créé l'interface IGameState qui contient des fonctions pour charger l'état courant, le démarrer, le stopper... La classe Game contient une pile d'état, l'état courant est alors l'état en sommet de pile. . En effet il est intéressant de garder l'état précédant pour pouvoir retourner facilement dans une phase précédente de la partie.

Pour lier les actions utilisateurs au modèle du jeu, nous utilisons le pattern observateur. Plus précisément, nous avons opté pour le système d'event de C# pour récupérer les mouvements de la souris et les actions du clavier dans les états de Game (respectivement les fonctions KeyboardPressedEventHandle, MouseClickedEventHandler, MouseMovedEventHandler).

3.6 Player : schéma 9

3.6.1 Namespace Player

Afin de pouvoir implémenter plusieurs types de joueurs, nous avons créé une interface commune IPlayer. En effet nous voulons pouvoir gérer aussi bien un joueur humain qu'une intelligence artificielle, qu'un joueur en ligne, ... Nous avons pour l'instant deux implémentations :

- HumanPlayer : qui gère un joueur humain manipulant la machine sur laquelle tourne le jeu. Cette implémentation utilise, comme la classe Game, le pattern observateur pour gérer les entrées utilisateur (la souris et le clavier).
- AIPlayer : qui gère une intelligence artificielle basique.

Les joueurs modifient le cours de la partie à l'aide d'une liste d'action qui est mise à jour à chaque tour. Un joueur possède aussi un état (pattern state) qui influence son comportement. Par exemple un joueur peut être dans l'état actif, l'état inactif, voire l'état déconnecté dans le cas d'un jeu en ligne.

3.6.2 Namespace Player.Actions

Nous avons décidé d'encapsuler les différentes actions des joueurs par un pattern commande. Les communications entre les objets sont donc maintenant contenues dans un objet ayant pour interface IPlayerAction. On peut citer de nombreux avantages à utiliser ce pattern plutôt que des "appels bruts" à des fonctions :

- Loguer les différentes actions des joueurs.
- Débuguer facilement le déroulement d'une partie.
- Si l'on veut implémenter un système de replay, il nous suffira de loguer toutes les actions et de les rejouer.
- Faire un système de retour en arrière.
- Si l'on veut faire un jeu en ligne, il nous suffira d'envoyer les actions des joueurs par paquets sous forme sérializée. Il ne restera plus qu'à désérialiser les actions à la réception des paquets pour synchroniser les deux joueurs.

Pour l'instant nous implémentons cette interface à l'aide d'une classe InGameAction et de nombreuses sous-classes comme AttackAction, BuildCityAction, etc. Nous envisageons de rajouter d'autres classes mères implémentant l'interface IPlayerAction, comme par exemple ChatBoxAction pour la gestion de dialogues.

4 Diagramme d'états-transitions

Nous avons créé deux diagrammes d'états-transitions :

- Un diagramme montrant la recherche d'un Tile dans le flyweight (voir schéma 10).
- Les états-transitions pour déplacer une unité(voir schéma 11).

5 Diagrammes d'interaction

Nous avons créé deux diagrammes d'interaction :

- Un diagramme représentant le lancement d'une partie. Ce qui comprend la création d'une instance de la classe Game, la création d'une carte (Map), la création des deux joueurs et enfin la création du moteur graphique. Le jeu peut ensuite entrer dans l'état LoadingState, puis dans l'état ActiveGameState (la partie a alors commencé, voir schéma 12).
- Un diagramme représentant une itération dans la boucle MainLoop. Cela consiste a dessiner toutes les unités de la map, a mettre à jour les animations pour la prochaine frame et enfin récupérer les actions du joueur actif. On voit dans ce diagramme que le joueur a décidé d'acheter une nouvelle unité Enseignant (voir schéma 13).

6 Annexes

FIGURE 1 – Développement d'une civilisation.

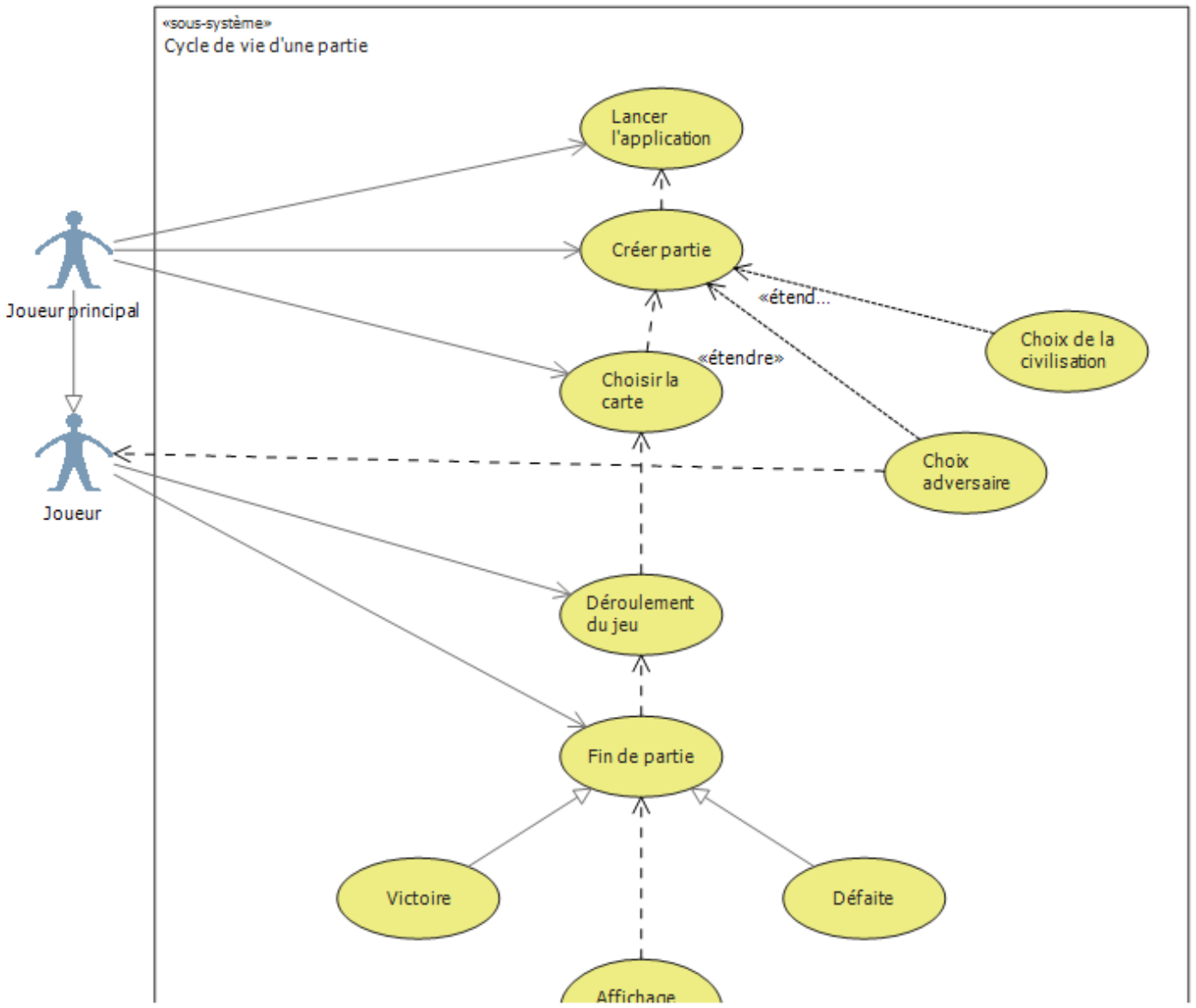

FIGURE 2 – Cycle de vie d'une partie.

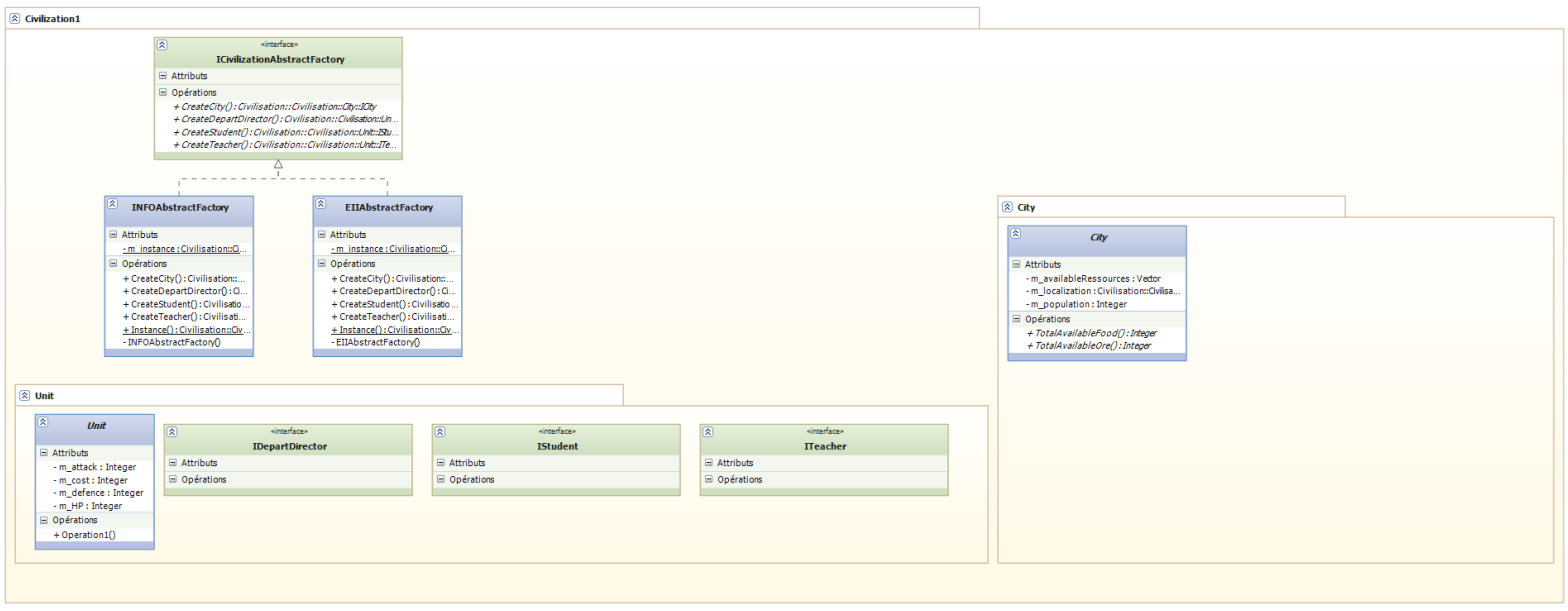

FIGURE 3 – Civilization (ancienne version).

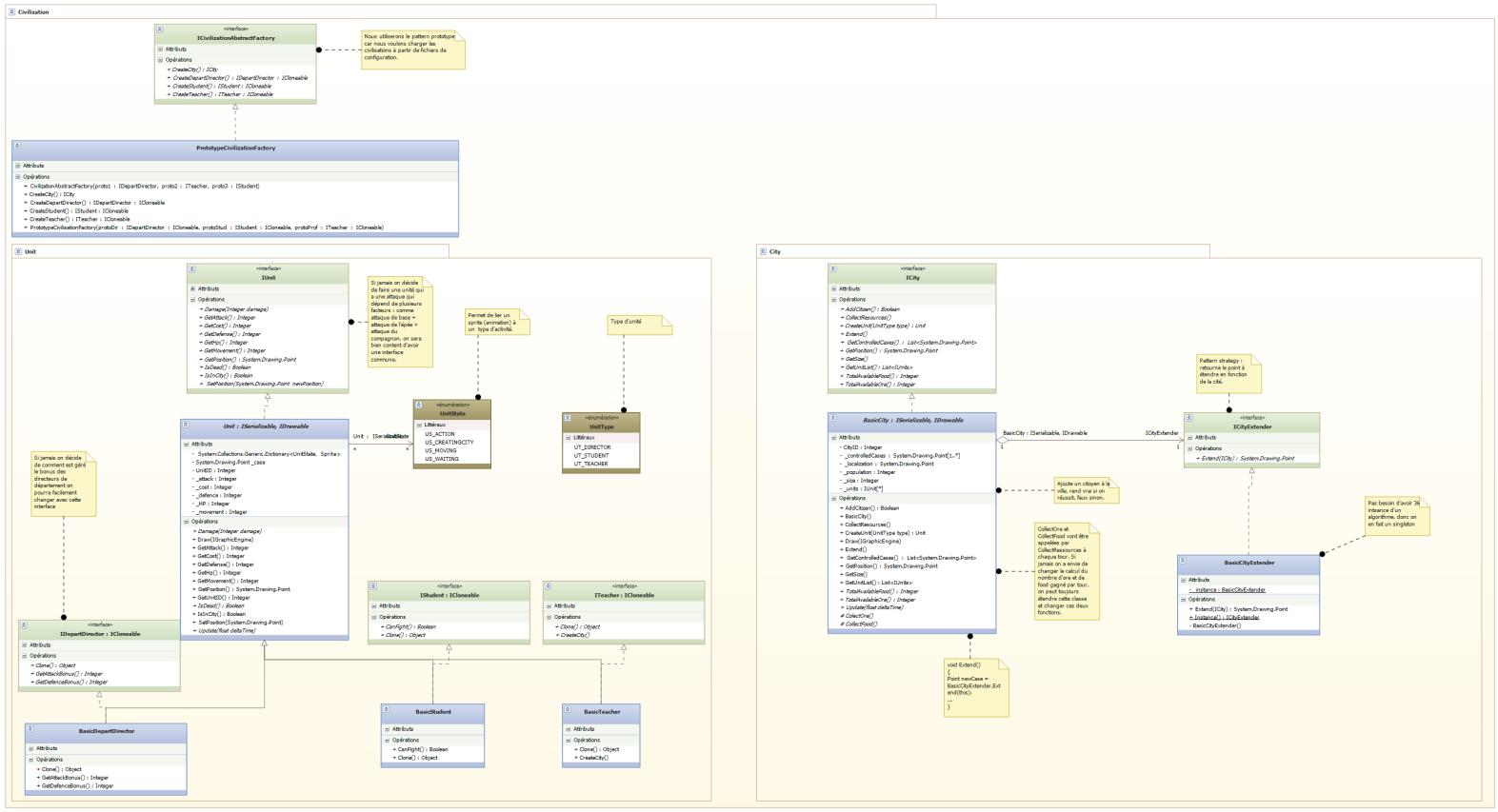

FIGURE 4 – Civilization.

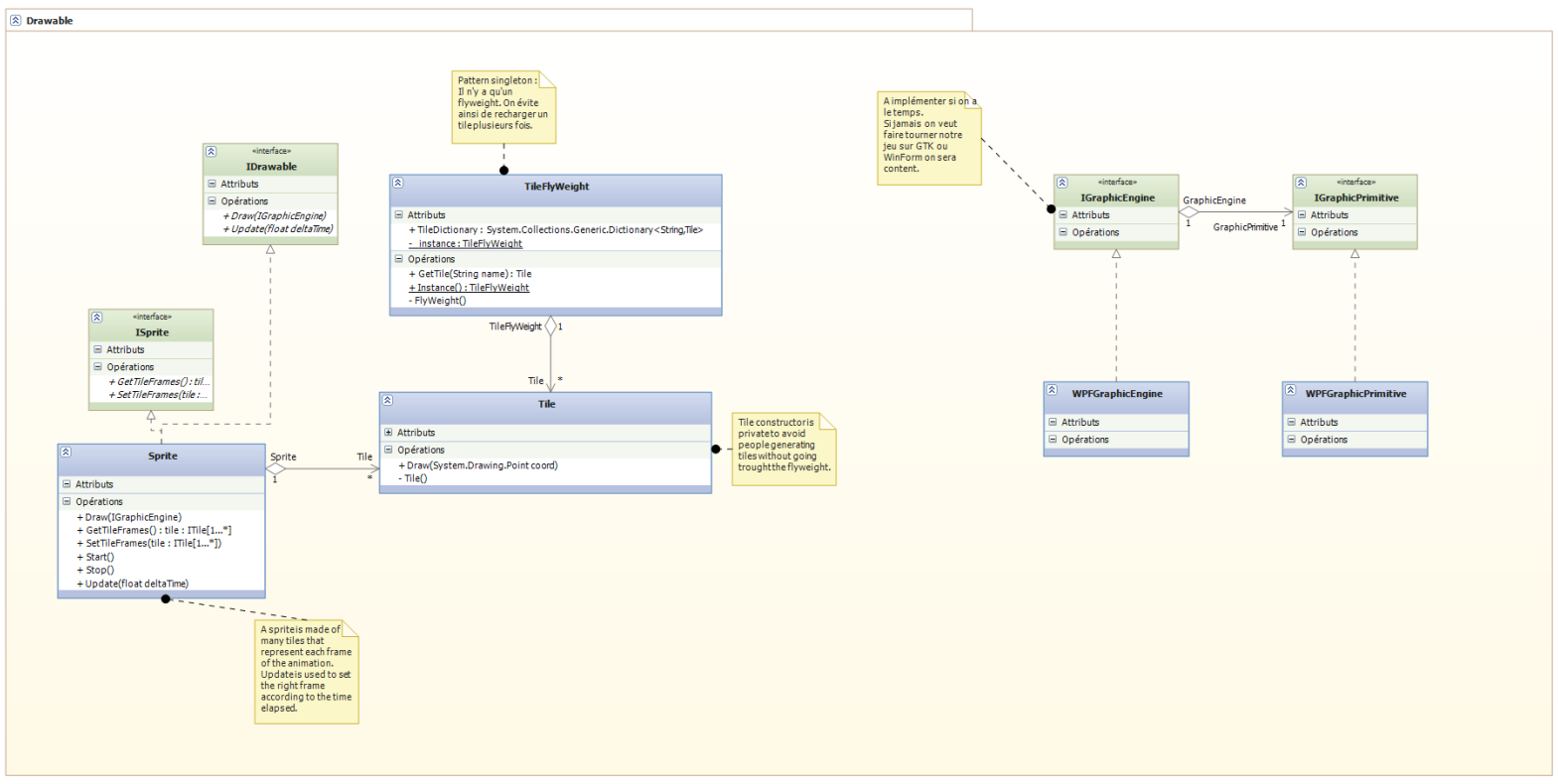

FIGURE 5 – View.

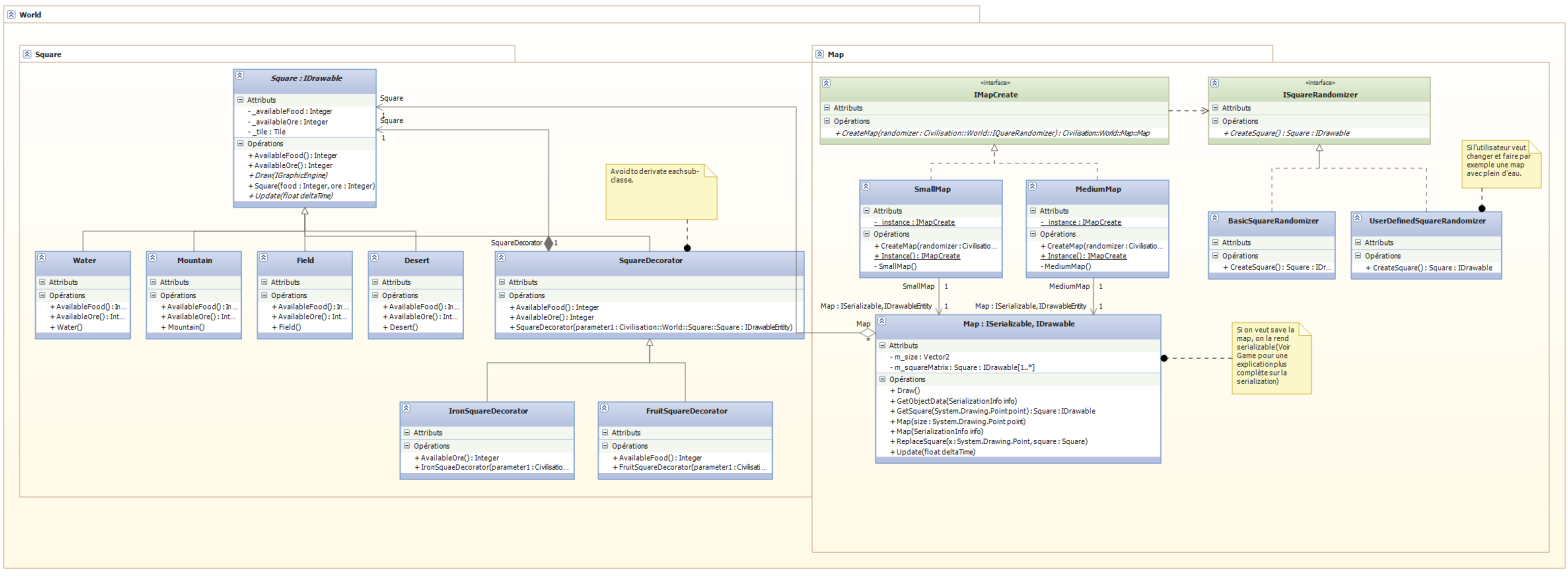

FIGURE 6 – World.

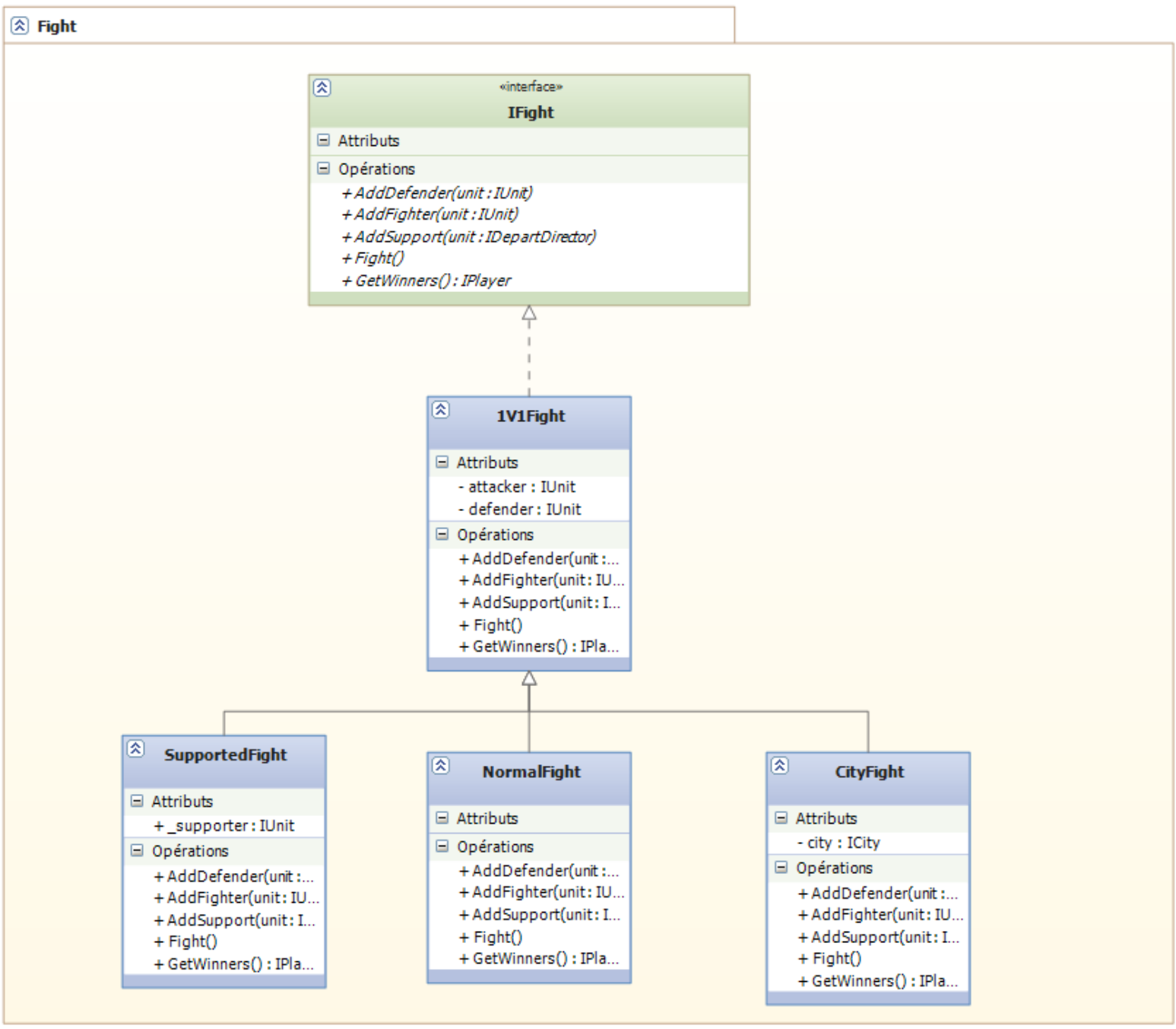

FIGURE 7 – Fight.

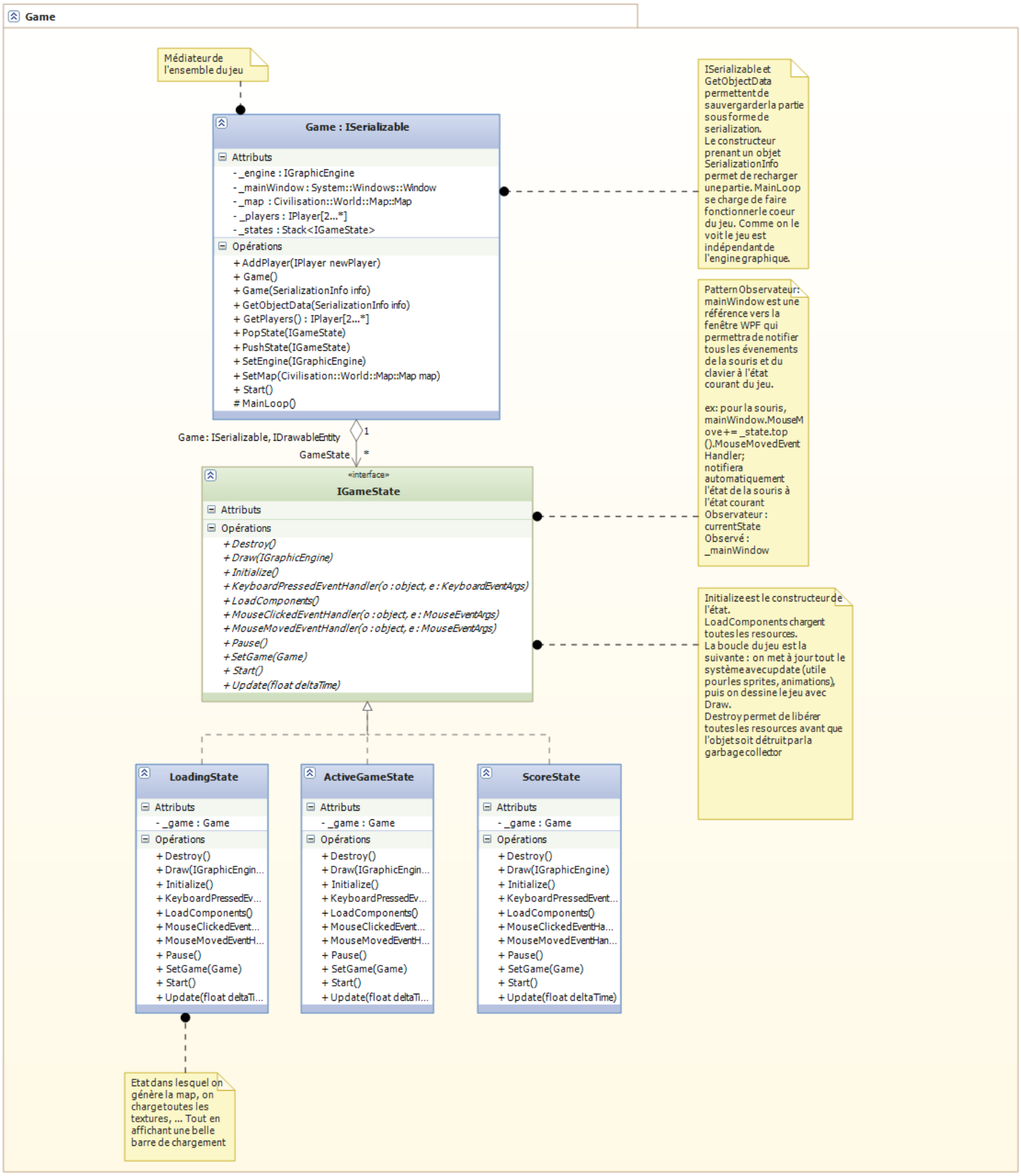

FIGURE 8 – Game.

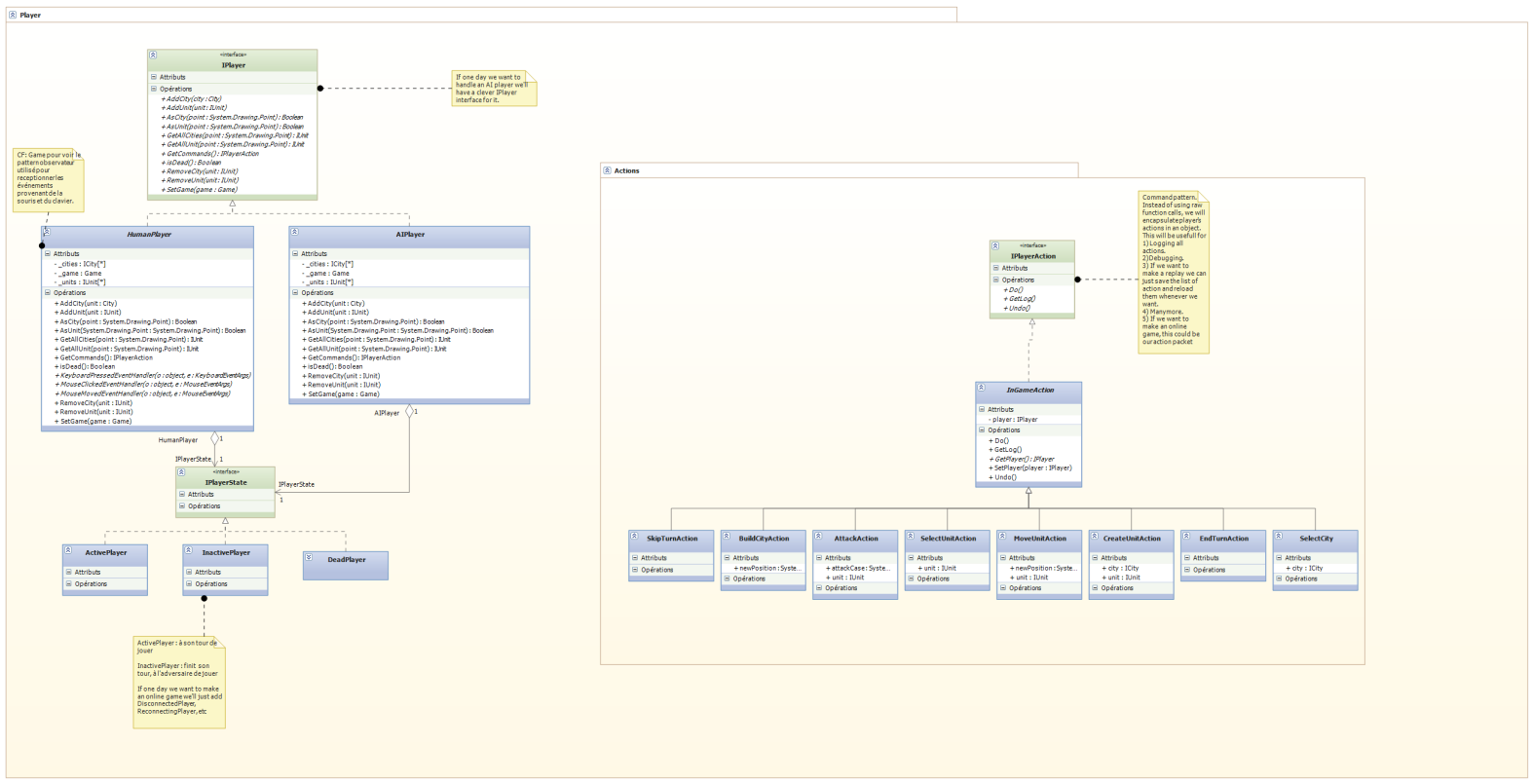

FIGURE 9 – Player.

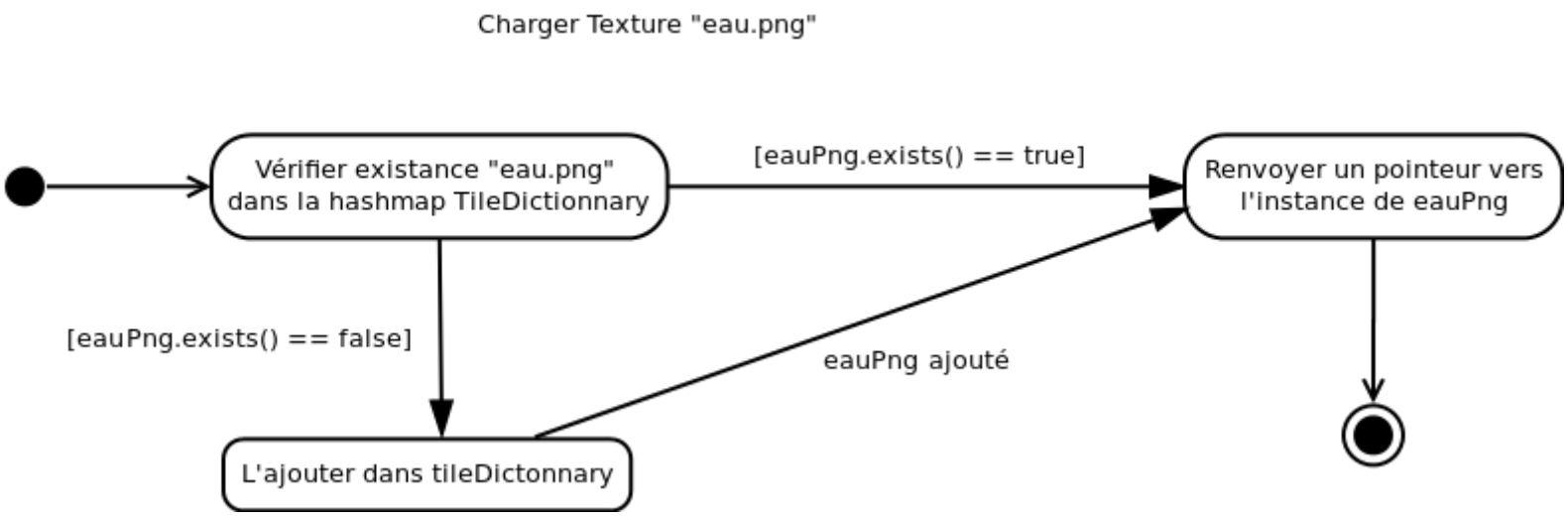

FIGURE 10 – FlyWeight.

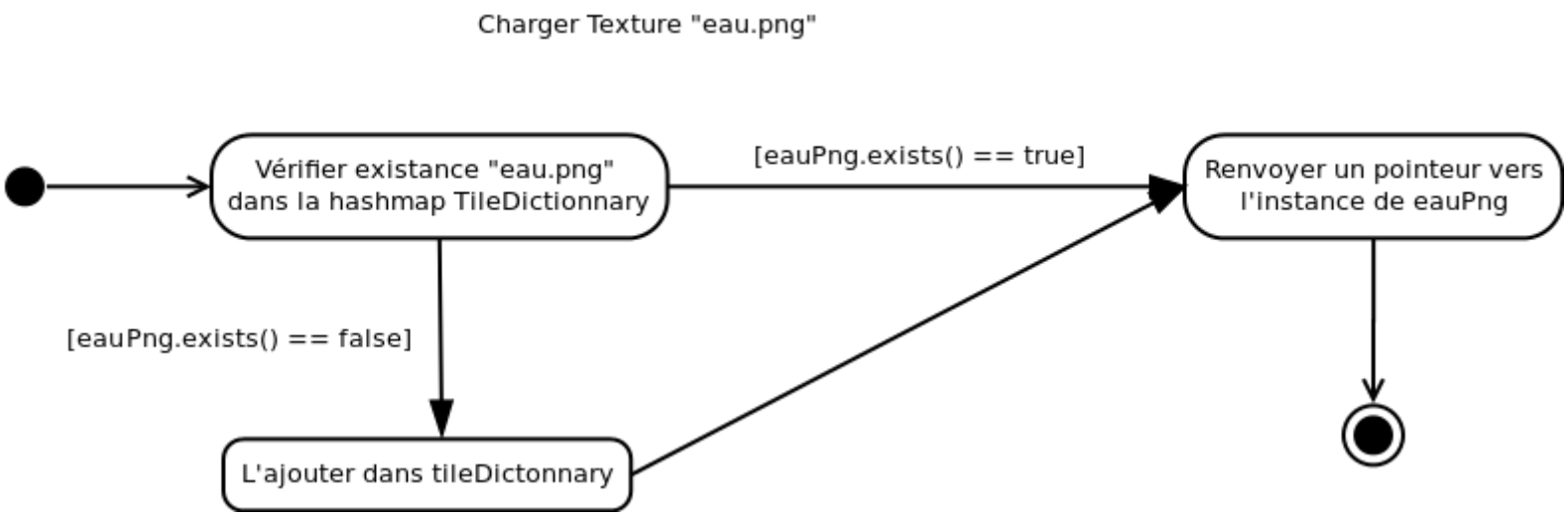

FIGURE 11 – Deplacement d'une unité.

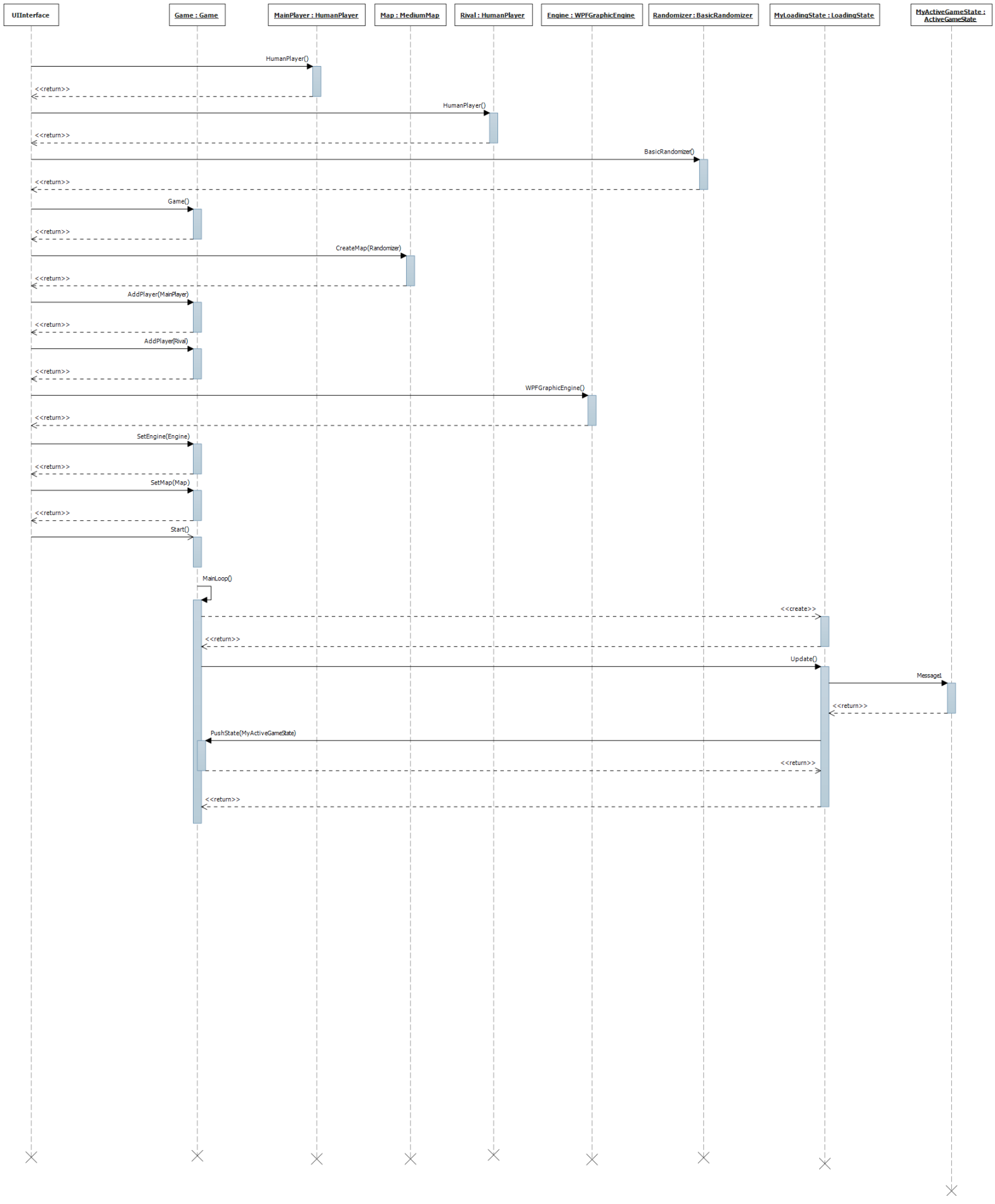

FIGURE 12 – Lancement d'une partie.

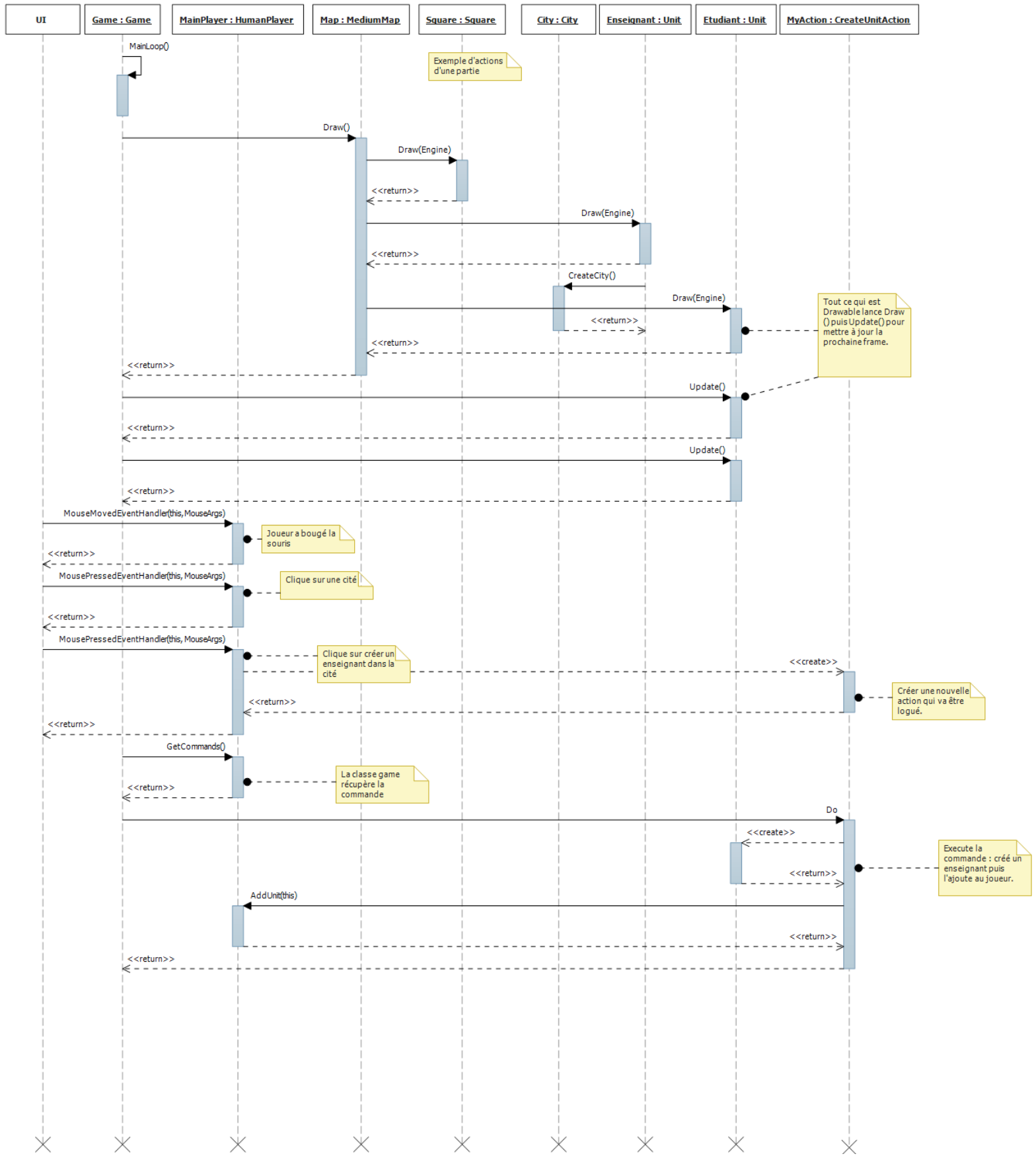

FIGURE 13 – Une itération dans la MainLoop.