INSA de Rennes
Département Informatique

Projet de Programmation Orientée Objet

BÉRARD Alexandre, MALKAS Gabriel

Rapport de conception

Année 2012 – 2013

Sommaire

1 Commentaires & Observations

1.1 Diagrammes de séquence

La création d'une partie est orchestrée par le GameBuilder, qui se charge de créer les différentes ressources selon les choix du joueur : créer la map (à l'aide de la future bibliothèque C++), créer les différentes instances de Player, et initialise une instance de Game.

Plusieurs actions sont disponibles pendant le tour d'un joueur. Nous avons choisi un certain ordre d'actions pour réaliser le diagramme, mais il faut garder un tête que c'est un ordre arbitraire. Au final, les actions seront réalisées dans un ordre choisi par le joueur au cours du jeu. Par ailleurs, les actions sont initiées à l'aide de l'interface graphique du jeu, non modélisée dans les diagrammes de classes.

Nous avons donc ajouté une ligne de vie fictive "IHM" afin de préciser d'où les actions sont issues. La nature asynchrone des évènements n'est donc pas visible sur ce diagramme mais devra être prise en compte lors du développement.

1.2 Diagrammes de classe

Note : Nous n'avons pas intégré les Getters/Setters sur les diagrammes de classe pour éviter de les alourdir inutilement.

1.2.1 Explicit interface per class antipattern

Le projet consistant en la réalisation d'une application cliente, et non d'une API, il est inutile voire néfaste de créer une interface pour chaque classe. L'intérêt d'une interface qui n'est implémentée que par une classe est quasi-inexistant, et cela rajouterait une complexité considérable à notre code.

Nous avons donc utilisé des interfaces et classes abstraites là où elles nous semblaient nécessaires, mais pas de façon systématique.

En ce qui concerne cet antipattern, Martin Fowler explique :

> « I come across many developers who have separate interfaces for every class they write. I think this is excessive, especially for application development. [...] I recommend using a separate interface only if you want to break a dependency or you want to have multiple independent implementations. If you put the interface and implementation together and need to separate them later, this is a simple refactoring that can be delayed until you need to do it. »
> *(Patterns of Enterprise Application Architecture, Chapitre 18)*

1.2.2 Pattern Flyweight

Par ailleurs, nous avons choisi d'utiliser le pattern Flyweight pour les textures et non pous les cases. Les cases ont chacune un nombre limité de paramètres (des entiers) qui peuvent varier selon les instances. Appliquer un pattern Flyweight à une telle classe impliquerait l'utilisation d'un contexte, ce qui alourdirait nos méthodes (un paramètre supplémentaire pour passer le contexte). Sachant que les instances des cases sont légères (contrairement aux textures qui contiennent une image), il est plus intéressant de limiter les instances de la classe *Texture*.

Nous utilisons les textures pour représenter graphiquement les cases, mais également les unités.

1.2.3 Redondance des informations

Nous avons également choisi de répéter certaines informations. Par exemple, chaque case dispose d'une liste d'unités, et chaque unité a en retour connaissance de la case sur laquelle elle se trouve. Cette redondance d'information a un impact limité sur les performances (il s'agit de simples références), mais permet de simplifier grandement certaines opérations.

Exemple : non seulement la méthode *MoveUnit(unit :Unit, to :Vector)* de la classe *Game* peut accéder directement à la case sur laquelle se trouve l'unité à déplacer, sans avoir à parcourir la liste des cases et chercher l'unité en question, mais la méthode *RemoveUnit(unit :Unit)* de *Case* peut également simplement

supprimer une unité car chaque case dispose d'une liste des unités présentes. Les deux types de requête peuvent être facilement implémenter grâce à la redondance de l'information concernant la présence d'unités sur certaines cases.

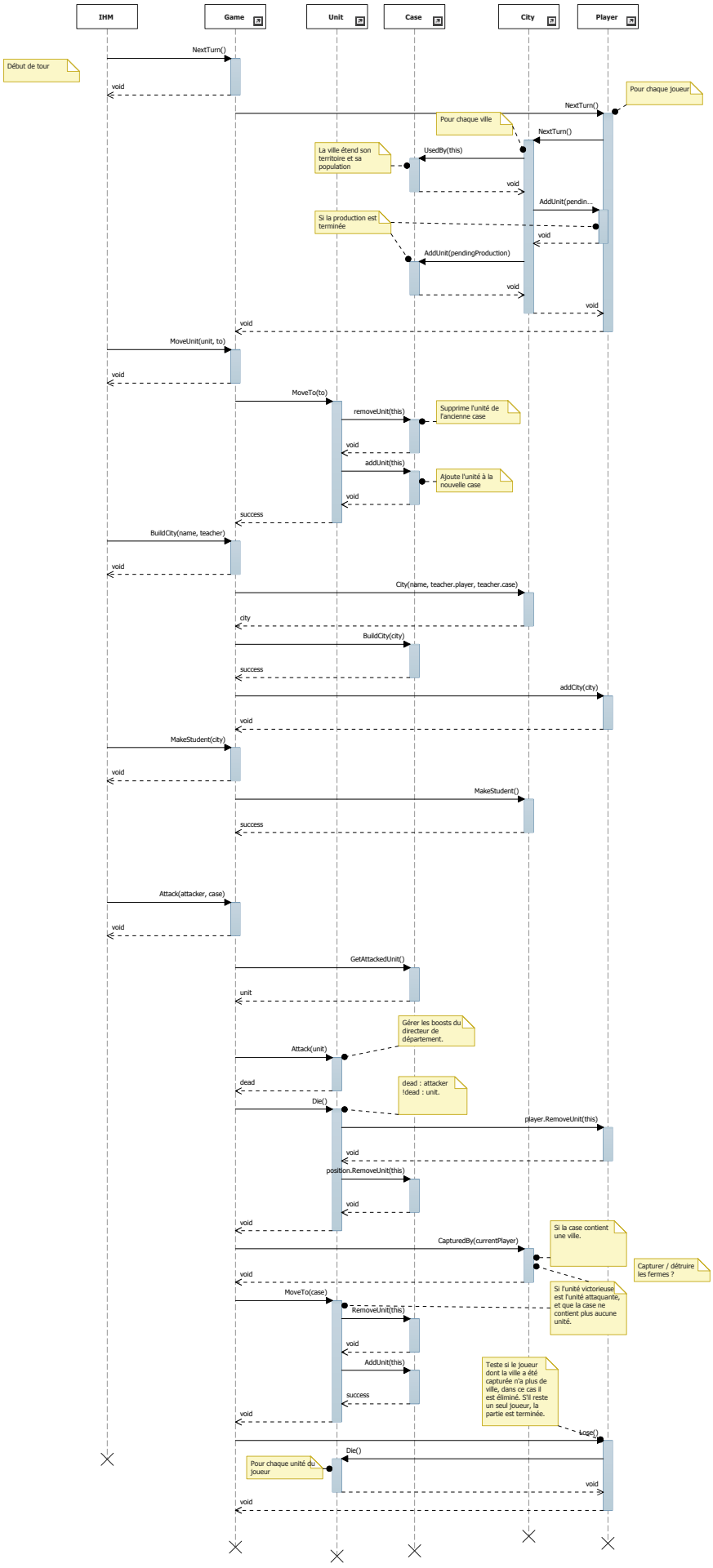
Le seul point négatif est la nécessité de tenir à jour la même information à plusieurs endroits.

1.2.4 État d'une case

Une case peut-être soit occupée par une ville (une ville est construite sur la case), soit utilisée par une ville pour produire des ressources, soit libre. Nous avons choisi de représenter l'état d'une case par une énumération.

2 Diagrammes

sd tour_joueur



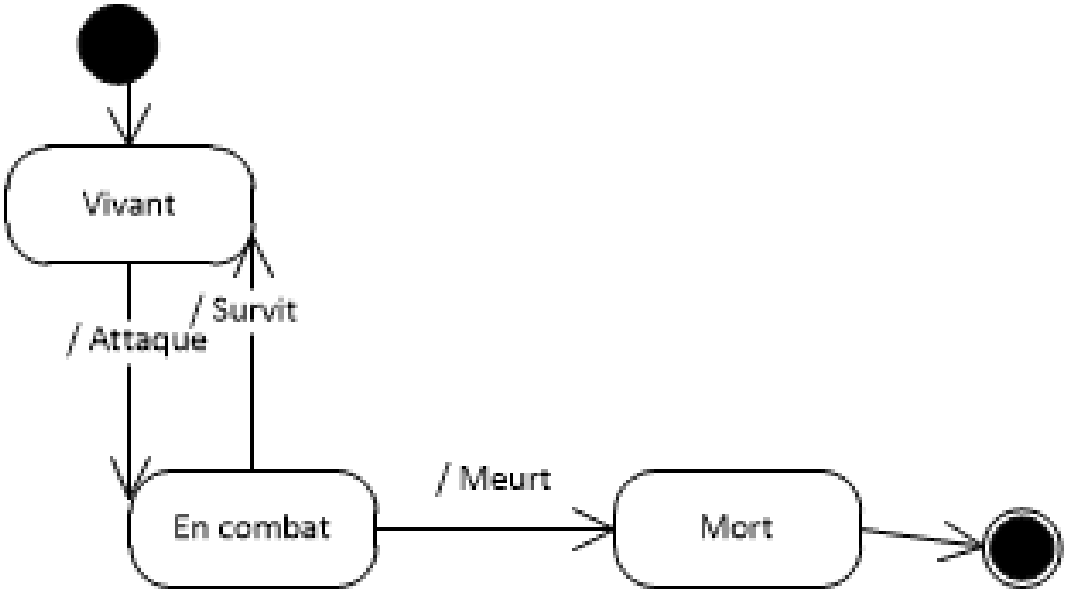


FIGURE 1 – Diagramme d'état-transition d'une unité

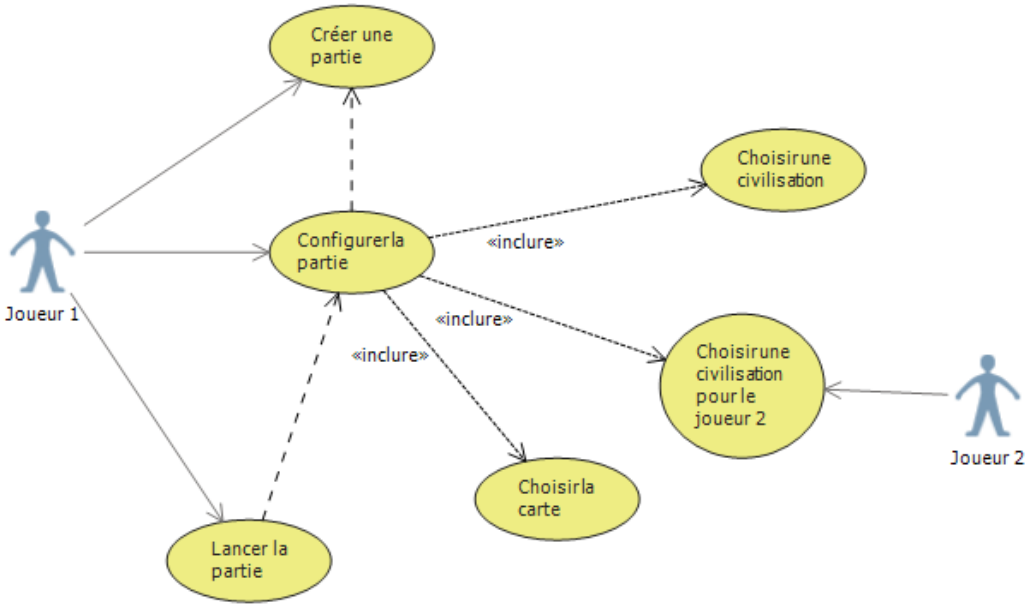


FIGURE 2 – Diagramme de cas d'utilisation de création d'une partie

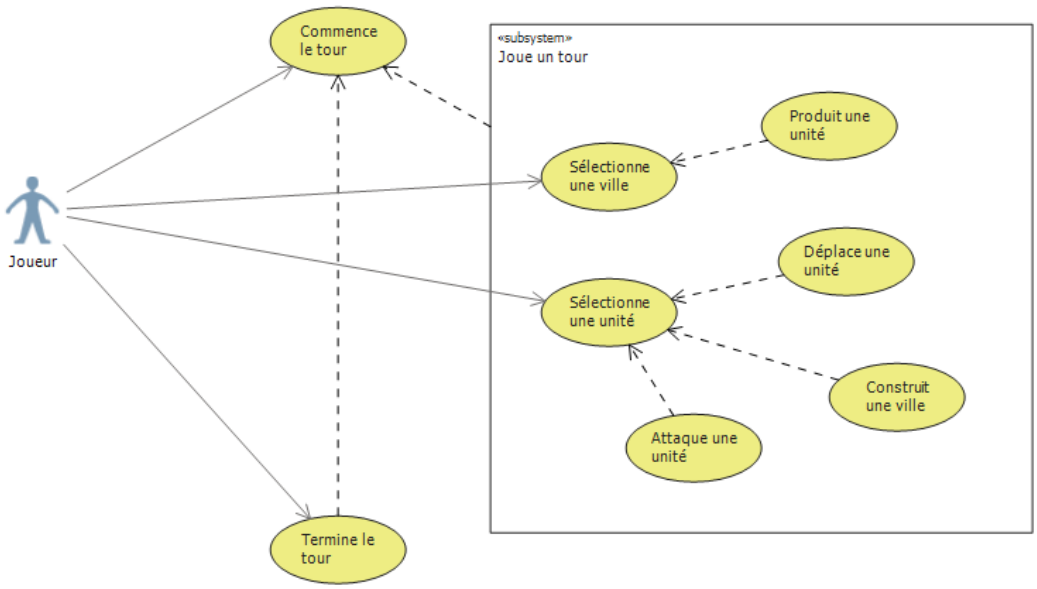


FIGURE 3 – Diagramme de cas d'utilisation de déroulement d'un tour

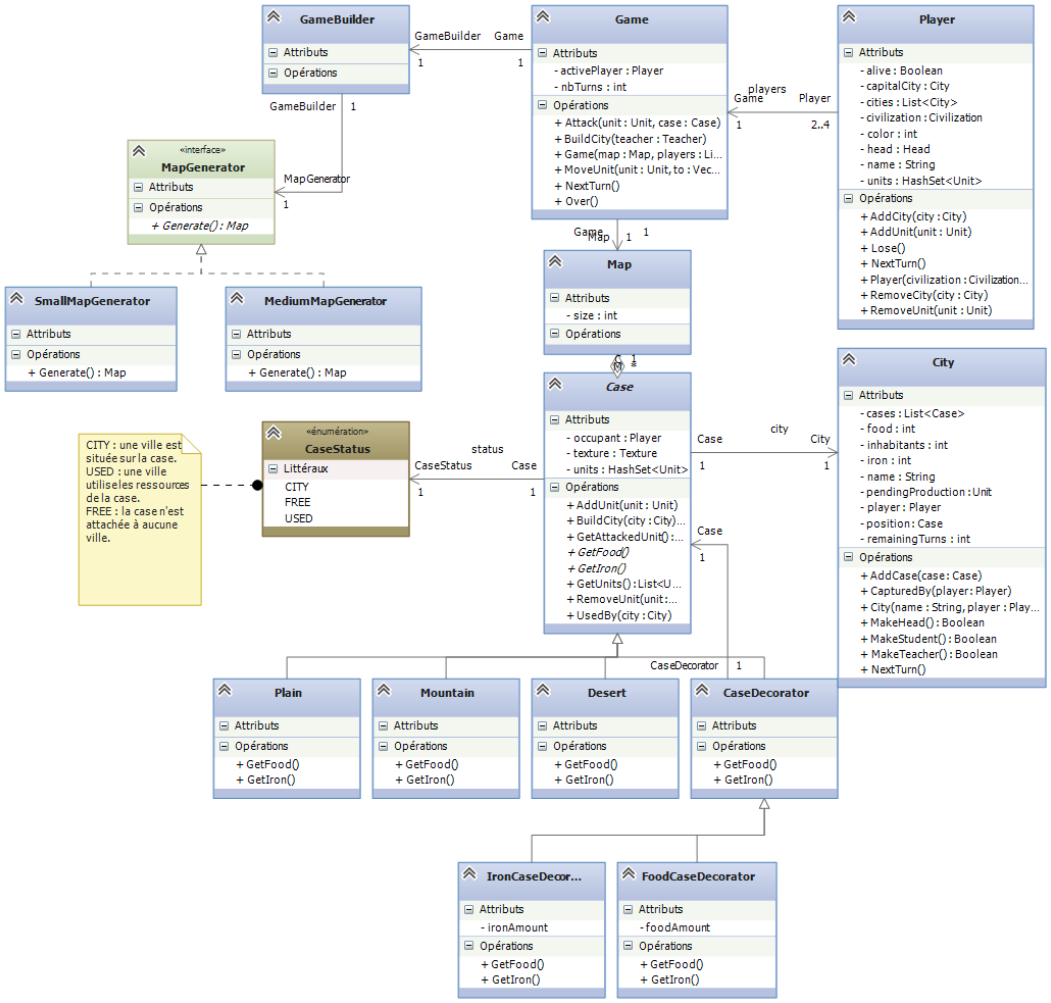


FIGURE 4 – Diagramme de classes générales

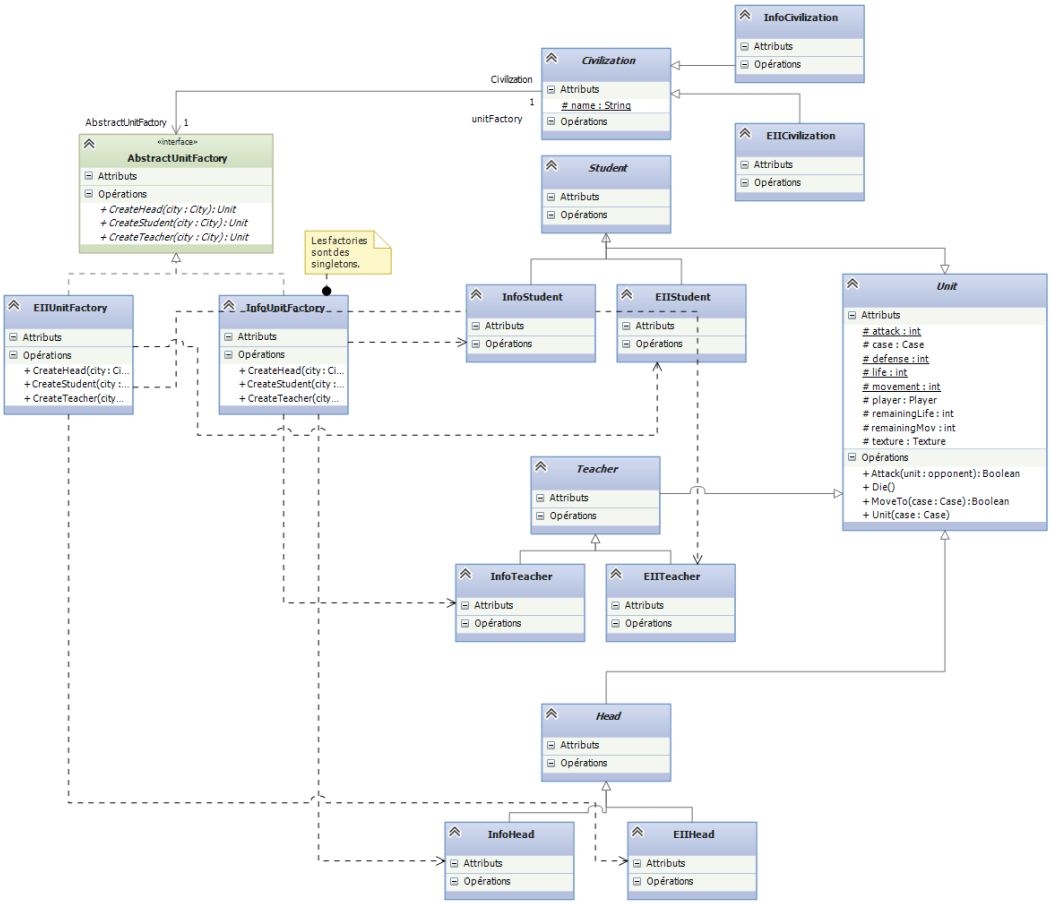


FIGURE 5 – Diagramme de classes des unités

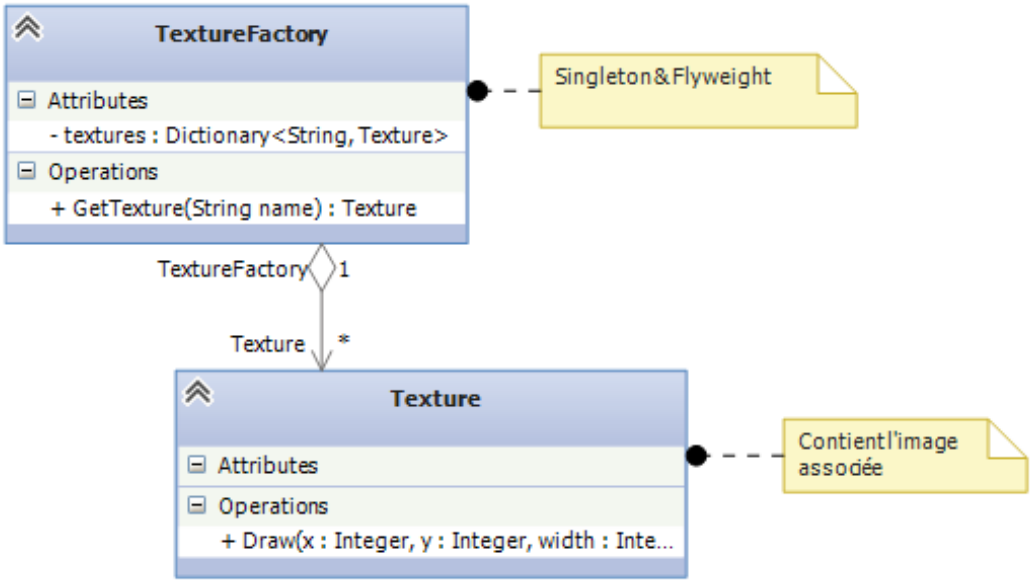


FIGURE 6 – Diagramme de classes des textures

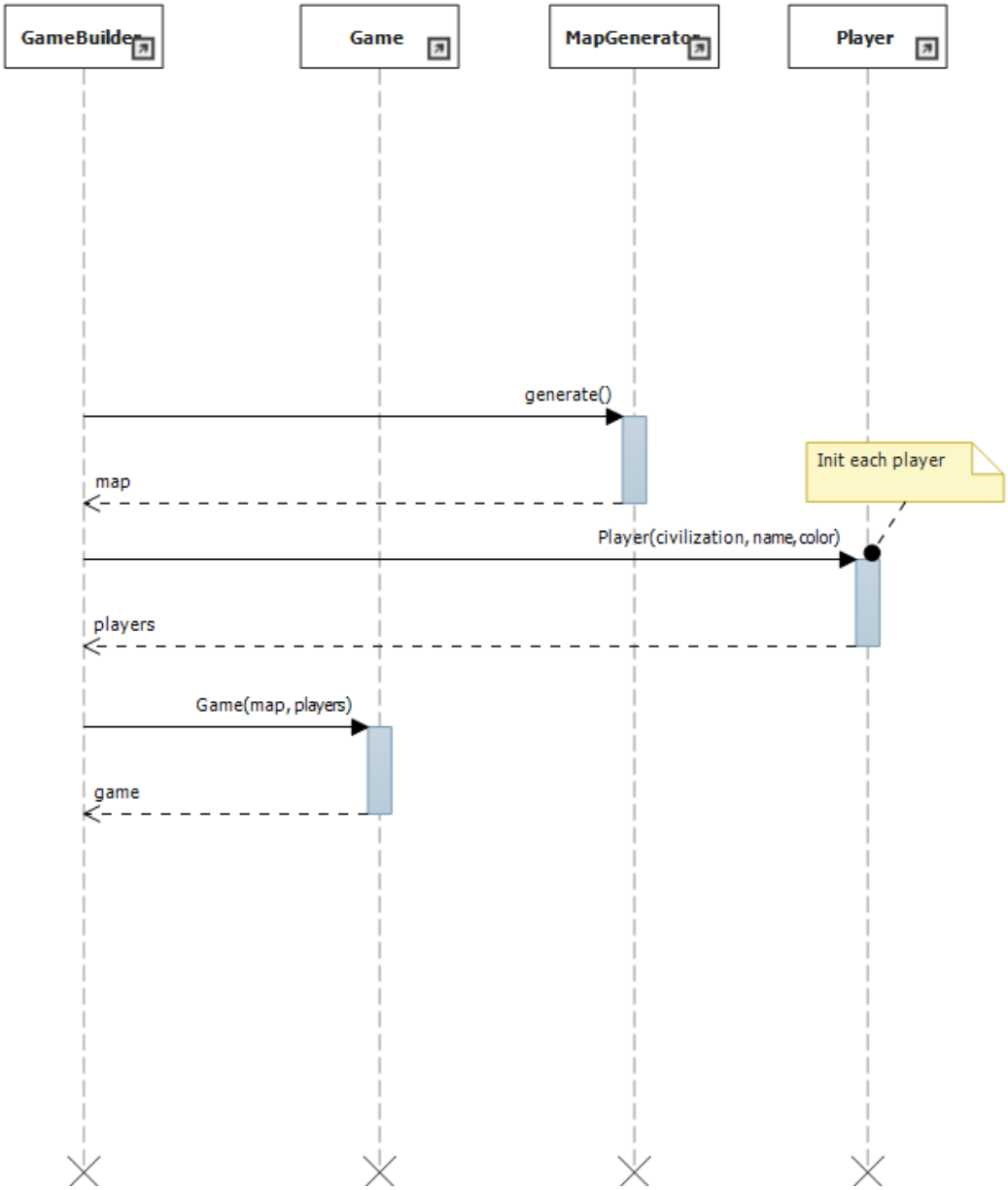


FIGURE 7 – Diagramme de séquence de création d'une partie