Analyse logicielle du jeu libre 0 A.D.

Duraj Bastien et Carreteros Laetitia
UQAC, 20 octobre 2018

Plan

1. Quelques mots sur 0 A.D.
2. Analyse architecturale
    i. Les différents acteurs et scénarios possibles (Diagramme des cas d'utilisation)
    ii. Diagramme des classes (Logical View)
    iii. Diagramme des packages (Development View)
    iv. Diagramme de séquences (Process View)
    v. Rétrospective
3. Analyse de la qualité du code
4. Design Patterns
5. Problème de conception
6. Conclusion

Quelques mots sur 0 A.D.

Notre choix s'est porté sur un jeu vidéo libre, 0 A.D. . Il s'agit d'un jeu de stratégie en temps réel, en opposition au jeu de stratégie au tour par tour, proposant de jouer avec une dizaine de civilisations ayant vécu de l'an 500 av. JC jusqu'à l'an 0. Le jeu peut être joué aussi bien tout seul qu'à plusieurs.

Né du jeu de stratégie en temps réel, Age of empires, lequel a été développé en parti par Relic Entertainment et édité par Microsoft

Studios, le jeu était au départ, comme beaucoup de jeu vidéo libre, un mod, c'est à dire qu'il utilisait les technologies et les ressources d'un jeu existant. Dans ce cas précis il s'agissait d'un mod du second volet de la saga: Age of empires II: The Age of Kings. Le but premier était de recréer Age of empires avec les technologies du second jeu mais le projet a fini par évoluer et devenir totalement autonome.

0 A.D. a dans un premier temps été en développement fermé avec la société Wildfire Games en 2002. Mais au cours de l'année 2009 l'entreprise décide d'ouvrir le développement et le code, permettant ainsi à qui le souhaite de participer.

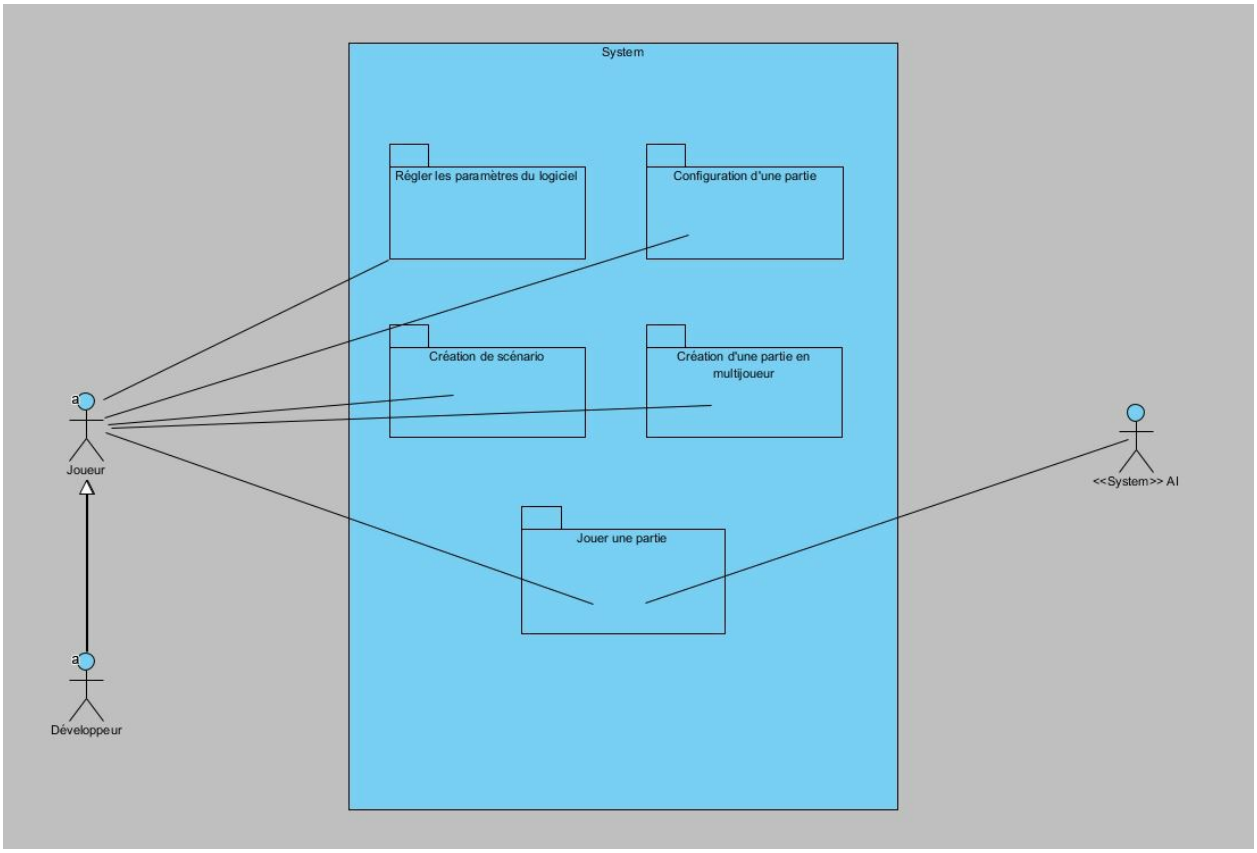
Ensuite notre choix s'est porté sur 0 A.D. car il est très actif et ceci depuis plusieurs années : la dernière mise à jour date du 17 mai 2018 ce qui est plutôt récent pour un tel projet. La documentation du projet est également très complète et aborde tous les sujets: comment participer, documentation technique, développer un patch, comment apporter une traduction, … . Le code et les ressources ainsi que l'exécutable sont facile d'accès et le forum très actif, permettant ainsi de poser des questions s'il venait à nous manquer des informations pour notre travail.

Dernier point intéressant pour ce projet : le jeu est multi-plaforme, il serait donc intéressant de voir comment cela a-t-il été intégré au projet.

Analyse architecturale

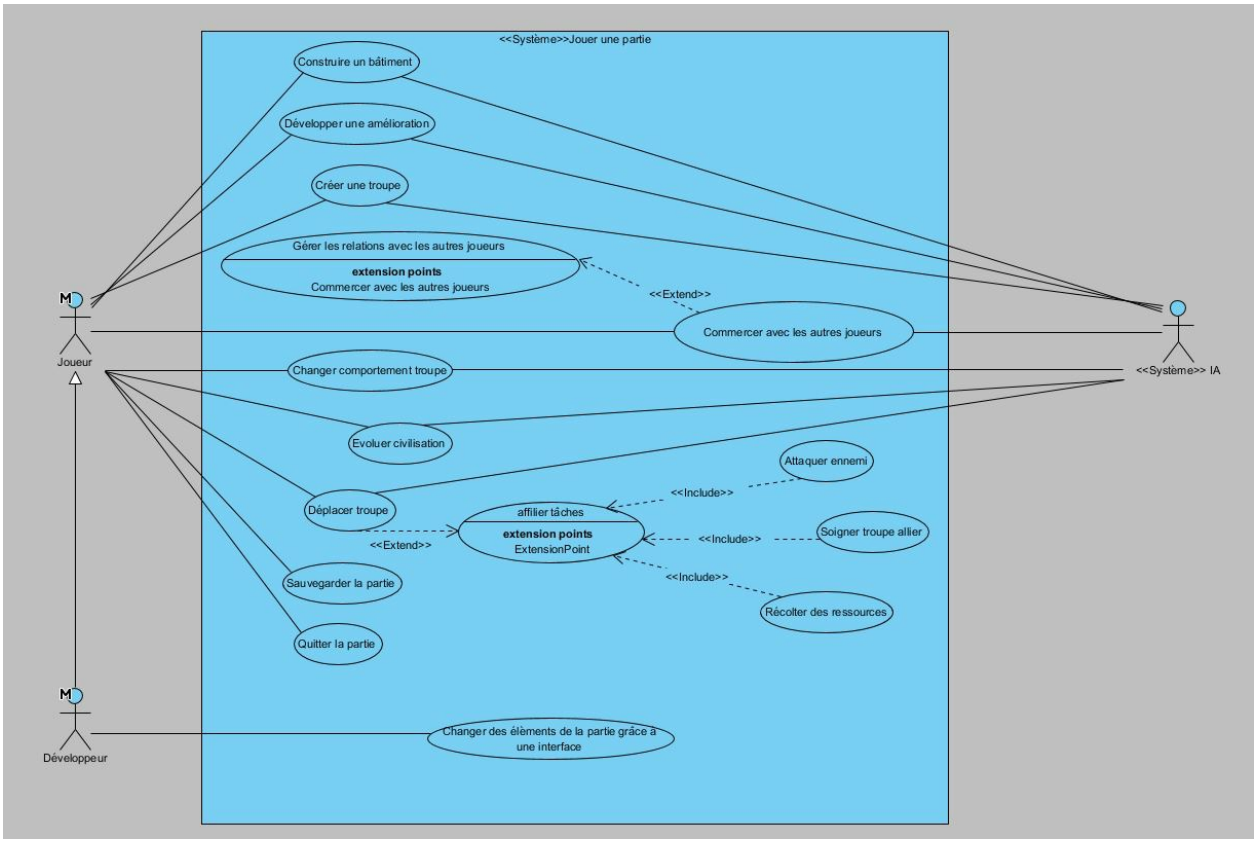
Les différents acteurs et scénarios possibles

Pour ce travail nous avons tout d'abord analysé l'architecture du logiciel en commençant par identifier les différents acteurs qui peuvent intervenir dans le déroulement du programme. Pour représenter ces acteurs et les fonctionnalités auxquels ils sont rattachés nous avons décidé de modéliser nos résultats avec, dans un premier temps, un diagramme regroupant les différentes parties du logiciel et les acteurs, puis dans un second temps un diagramme des cas d'utilisation pour chaque partie que nous avons identifié.



Nous avons donc diviser en 5 parties les différentes utilisations possibles du jeu avec 2 acteurs humains, les joueurs et les développeurs, et un acteur système représentant l'intelligence artificiel du jeu: Petra. Alors que le développeur peut faire tout ce que le jeoueur peut, en plus de la possibilité d'utiliser une console, l'IA ne peut que joueur une partie.

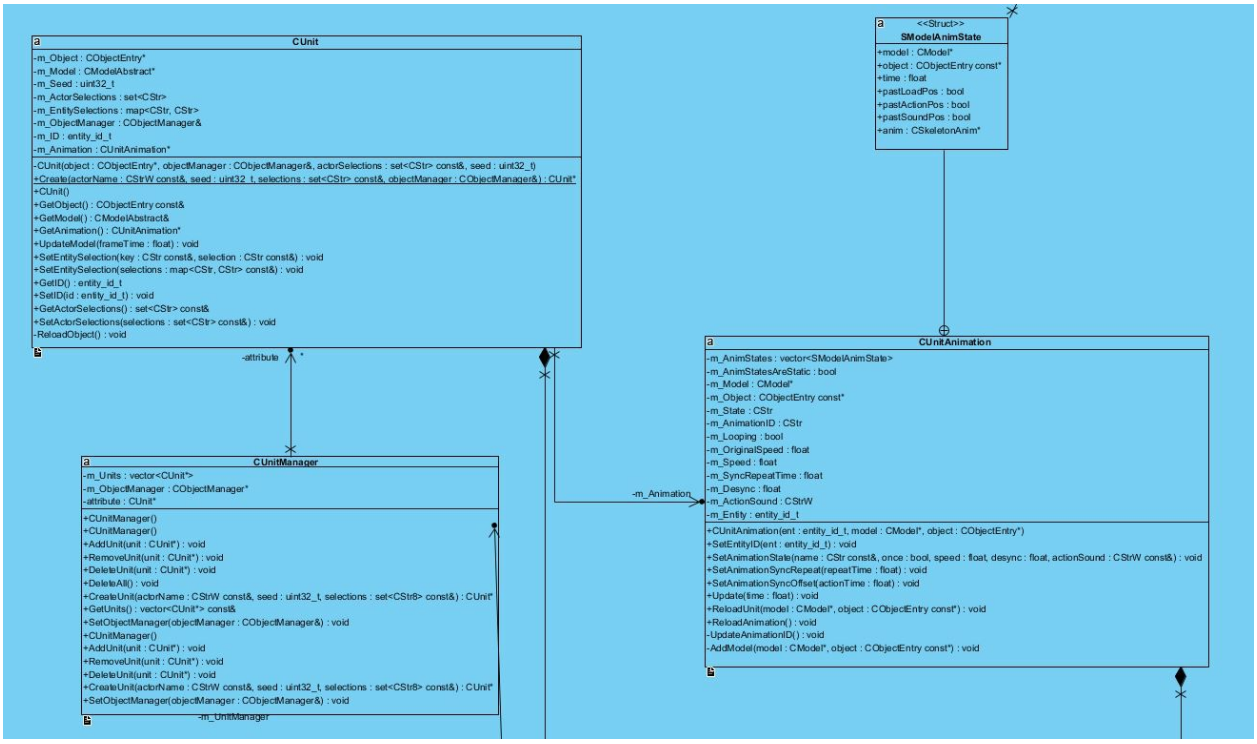
Nous avons ensuite détaillé chaque partie à l'aide d'un diagramme de cas d'utilisation pour nous permettre de mieux comprendre les objectifs et les besoins d'un tel logiciel. Nous n'analyserons pas tous les diagrammes de cas d'utilisation car ils ne sont pas tous interessant.



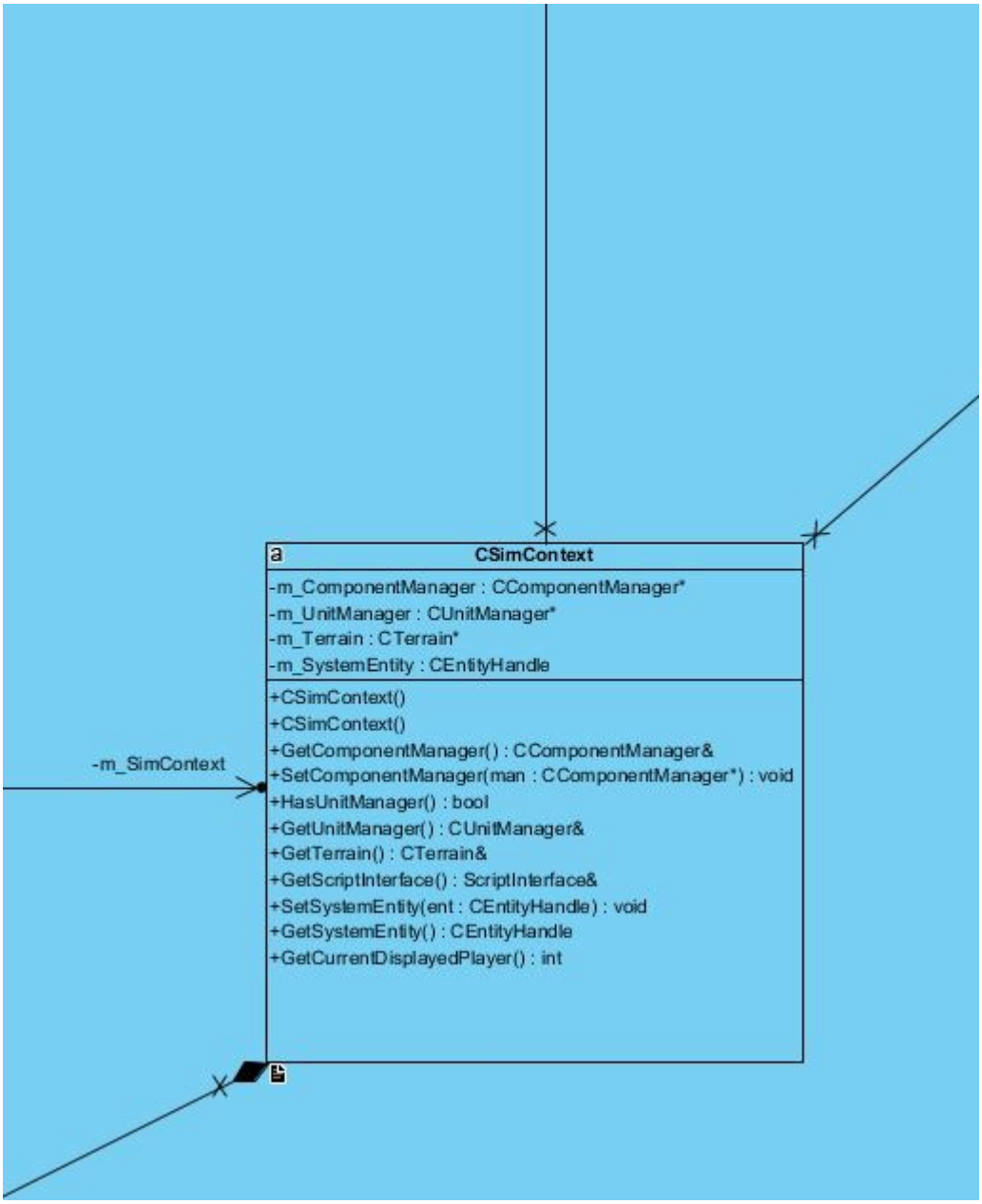
Le diagramme qui nous intéresse le plus ici est celui qui concerne le déroulement d'une partie. Il est le centre du jeu et justifie l'existence même du moteur graphique, c'est aussi l'une des parties les plus complètes où chaque acteur à un rôle à jouer. Ce diagramme montre bien les nombreuses actions que le joueur ou l'IA peuvent faire pour remporter le jeu, qu'ils s'agissent des relations diplomatiques avec d'autres joueurs ou de la création de troupe. Il met également en avant un point important de tout jeu vidéo d'envergure, la possibilité de faire apparaître une console qui permet d'appeler directement des fonctions javascript pour par exemple tester de nouveaux élèments non implémentés.

Diagramme des classes (Logical View)

Le diagramme des classes étant assez grands et complexes nous avons décidé de réduire la présentation de cette partie à un élèment unique du logiciel : la gestion des unités.

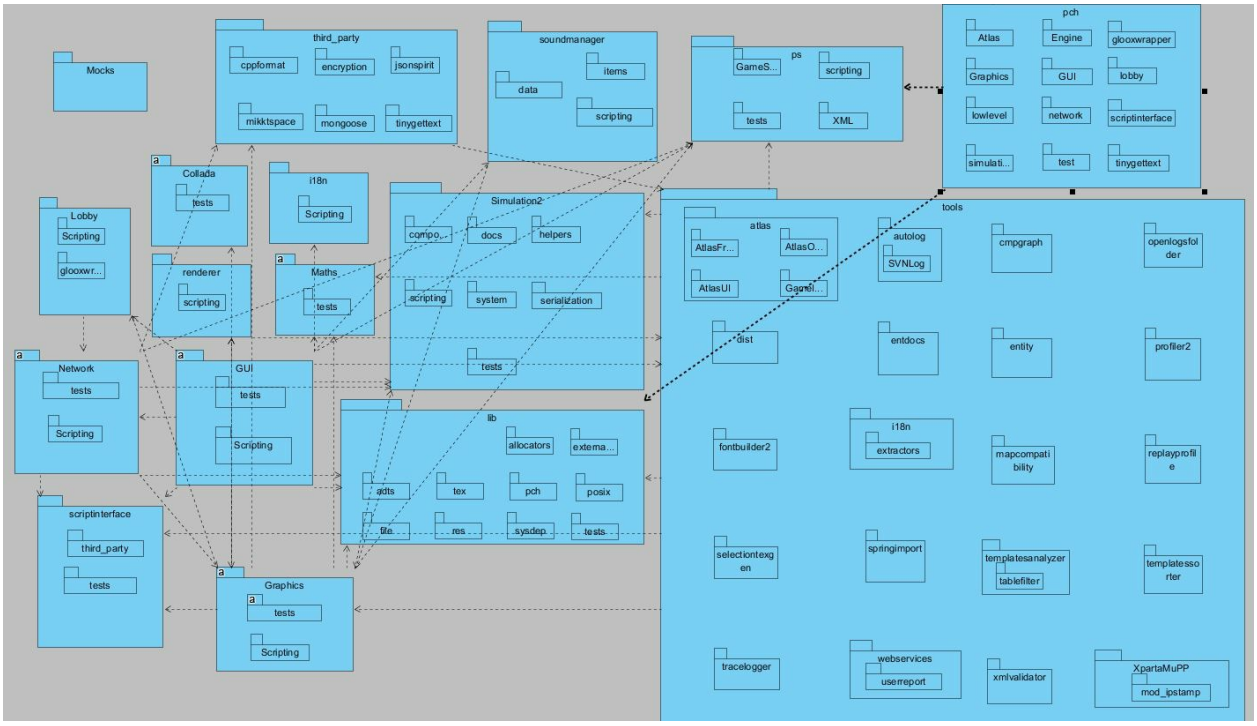


Cette partie du diagramme est assez représentative du reste du code. On trouve une classe de base gérant les unités appelées "CUnit" celle-ci comprend notamment un objet animation et un objet model pour la représentation graphique de l'unité. Ce qui est intéressant ici c'est qu'il existe une classe spécialisée appelée "CUnitManager" qui permet de gérer toutes les unités au même endroit et de rapidement faire des opérations dessus. Ce schéma se retrouve souvent dans le logiciel sans doute pour pourvoir plus facilement optimiser l'usage de la mémoire, ce problème étant très courant dans les jeux vidéos. La classe "CUnit" utilise également un Abstract Design Pattern avec un constructeur privée et une méthode static "Create" se chargeant des allocations mémoires à faire, mais nous reviendrons sur les Design Patterns dans une autre partie. La classe est ensuite relié à une autre partie du logiciel que voici.



Il s'agit du contexte auquel va se référer le logiciel lorsqu'une partie sera lancée. Ce contexte possède donc un gestionnaire d'unités avec un terrain, un gestionnaire d'entité regroupant tous les types d'objet qui peuvent intervenir dans une partie et un gestionnaire de composants. Cet object regroupe donc un grand nombre d'informations essentielles au bon déroulement d'une partie.

Diagramme des packages (Development View)



Le diagramme des packages ci-dessous permet surtout de remarquer que les différents packages ne sont pas tous bien organisé en dossiers plus petit, ce qui peut poser problème pour la compréhension ou la refacturation du code, par exemple dans le package "graphics" qui très petit, on trouve 118 fichiers ce qui rends l'exploration laborieuse. De plus le diagramme de package ne permet pas sous cette forme de reconnaître les élèments les plus importants de l'architecture comme la gestion des graphismes dans "graphics" qui possède l'un des diagrammes de classe les plus imposants comme montré ci-dessous.

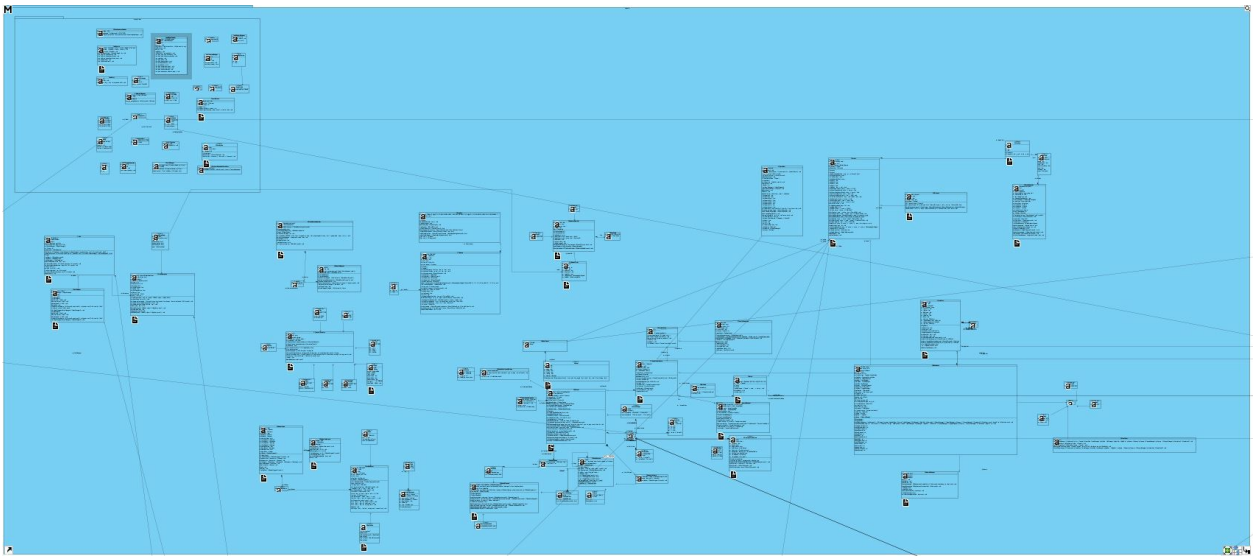
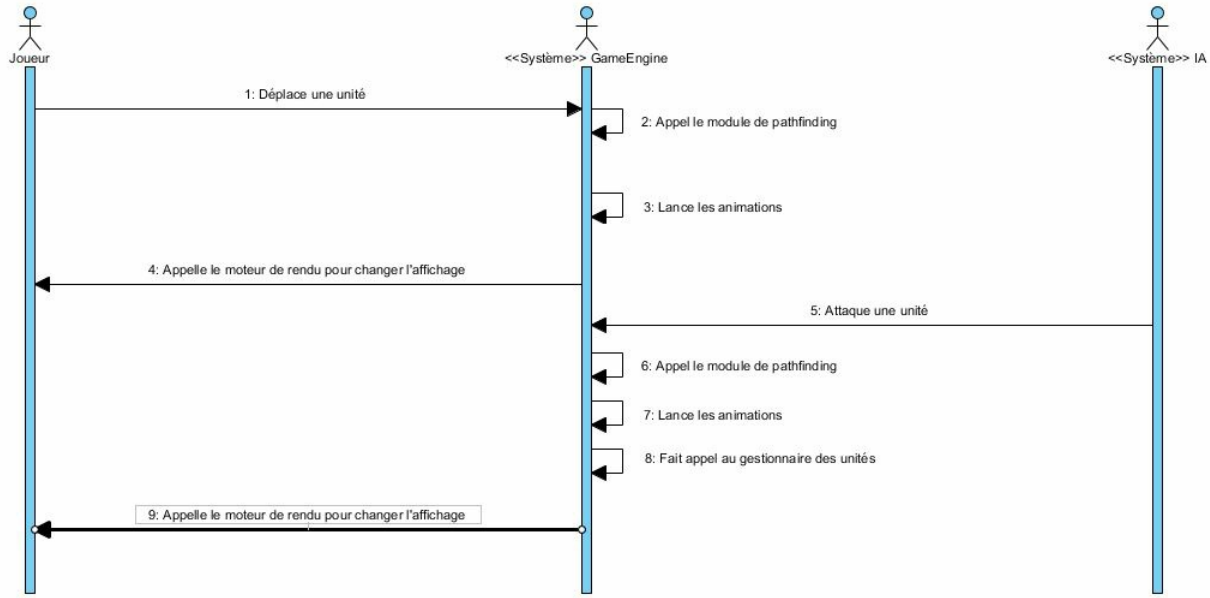


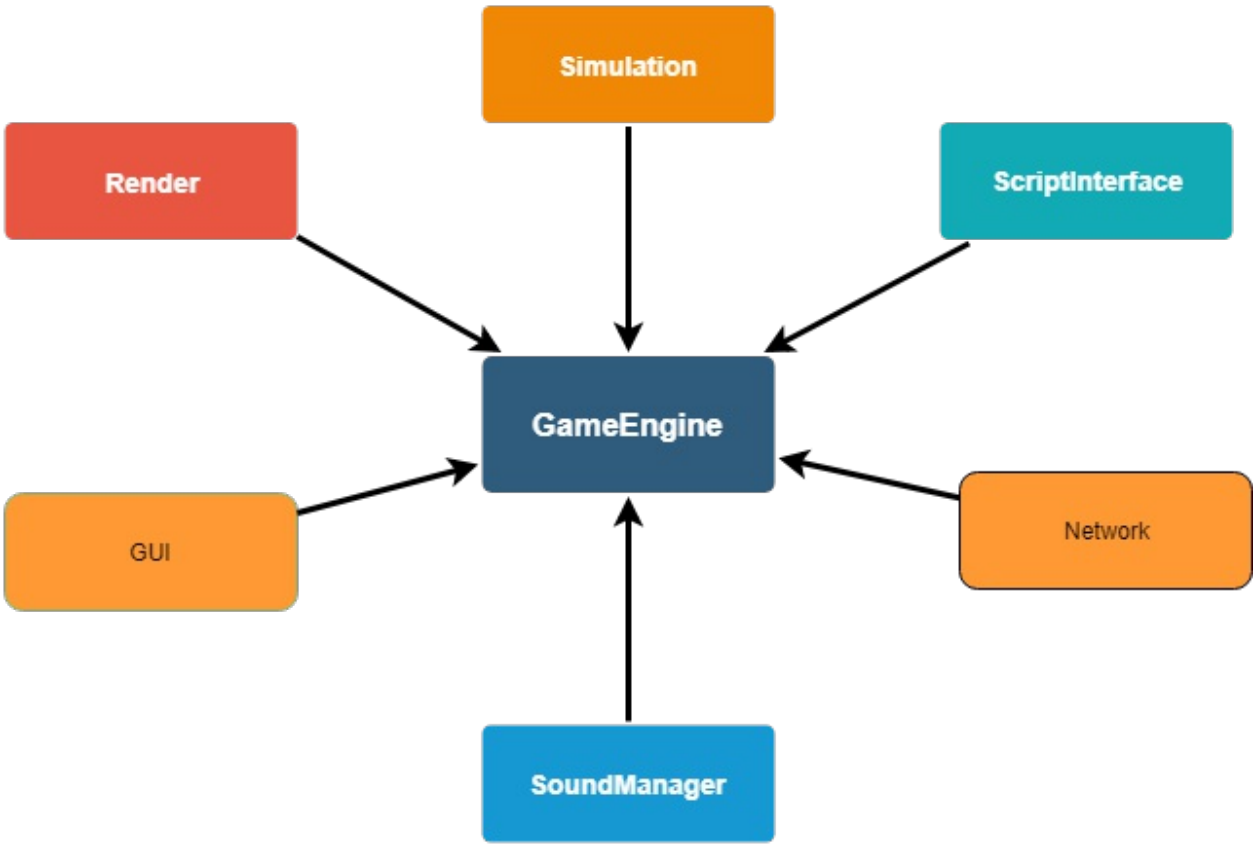
Diagramme de séquences (Process View)



Ce diagramme de séquence nous apprends un peu plus de chose quand au fonctionnement du logiciel. Le moteur de jeu gère ici l'aspect controller, il reçoit des évènements puis appel en consèquence les bons modules chacun gèrant une partie du logiciel comme l'IA, le rendu, la caméra, le pathfinding, les animations, ect … .

Rétrospective

Les différents diagrammes que nous avons présentés nous ont permis d'en déduire une architecture pour le logiciel que voici:



L'architecture est donc centré autour du moteur de jeu qui regroupe les différents composants essentiels au fonctionnement du jeu. Le moteur s'occupe de récupérer et d'interpréter les évènements transmis par le joueur (selectionner un élèment, charger une partie, …) et de synchroniser l'utilisation de tous ces composants

pour éviter les problèmes de concurrences.