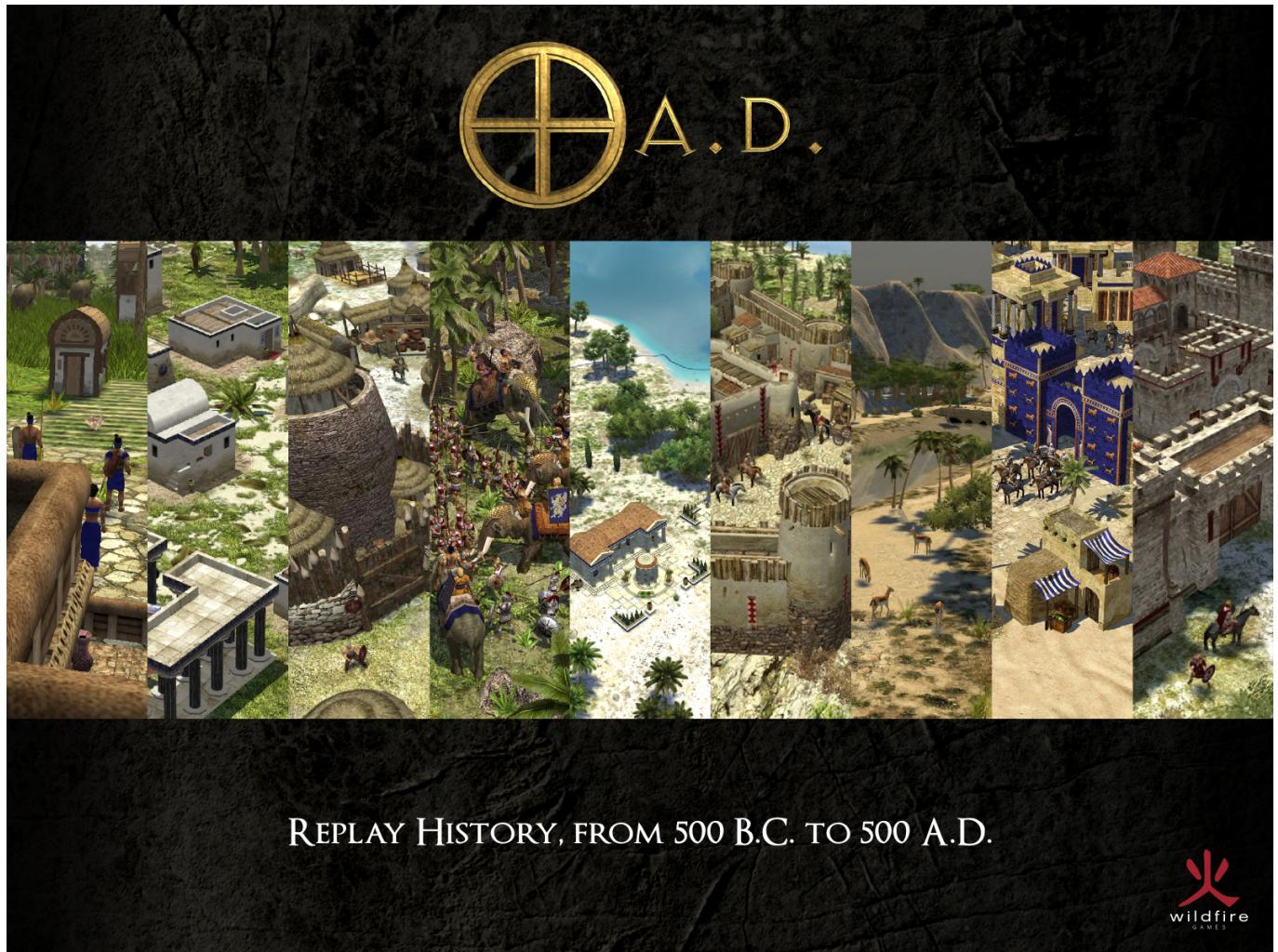


# Analyse logiciel du jeu libre 0 A.D.



Duraj Bastien et Carreteros Laetitia UQAC, 05 décembre 2018

## Plan

1. Introduction
2. Description du logiciel
3. Analyse architecturale
  1. Les différents acteurs et scénarios possibles
    1. Présentation des grandes fonctionnalités
    2. Disposition des fonctionnalités
    3. Détail d'une fonctionnalité
  2. Development View
    1. Disposition des packages
    2. Répartition du code dans l'architecture
    3. Lien entre les packages
  3. Logical View
    1. Structure globale du code C++
    2. Implémentation de la gestion des unités
    3. Problème de la structuration des classes
  4. Process View
    1. Diagramme de séquences lors de l'action d'un joueur

5. Conclusion sur l'architecture
4. Analyse de la qualité du code
  1. source/simulation2
  2. binaries/data
  3. Design Patterns
  4. Conclusion sur la qualité du code
5. Conclusion

## Introduction

Nous allons ici analyser 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. Celui-ci propose 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.

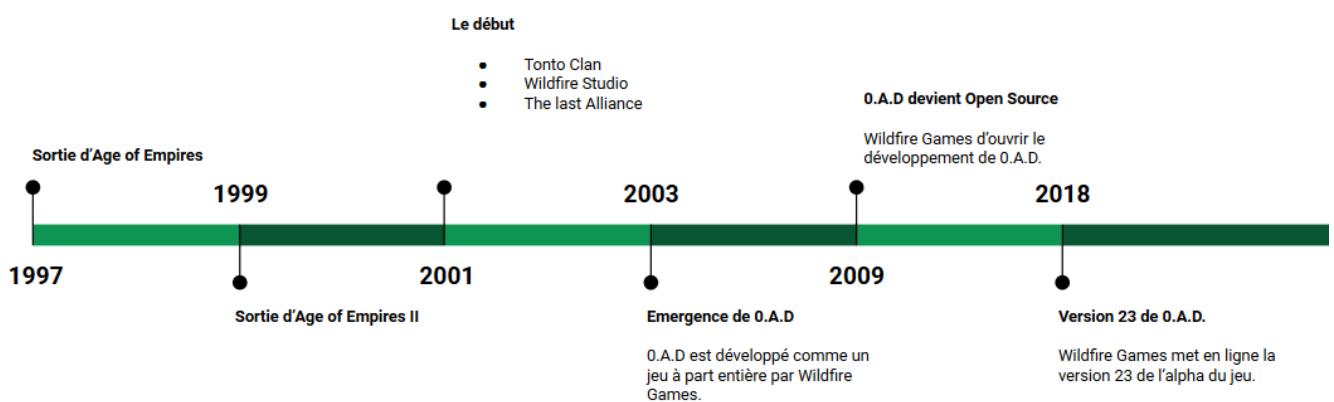


fig. 0 Chronologie simplifiée de 0.a.d

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 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 développé par Wildfire Studio , cependant ils ont été frustré par la limitation à ne pas pouvoir accéder à tout le code. Il y a eu d'autres organisations : Tonto Clan , qui ont voulu proposer un mod à Age of Empire II mais qui à été recalé car la société était en développement de Age of Mythology. Les autres participants aux prémisses de 0.A.D sont les développeur du mod The Last Alliance. Tous les trois vont former Wildfire Games.

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.

Pour commencer nous allons expliquer ce qui nous a intéressé dans ce projet. Tout d'abord nous avons choisi un jeu vidéo car cela nous semblait intéressant de voir leur construction et leur

architecture mais également leur évolution au cours du développement. Les jeux vidéos sont des logiciels de divertissement, ils ont donc des particularités qui leur sont propres notamment celle de faire collaborer des artistes et des designer avec des programmeurs, ou encore de devoir, sur un projet à long terme, changer de moteur graphique pour un meilleur rendu par exemple. Nous sommes donc intéressés pour voir comment ces contraintes sont gérées dans un développement ouvert.

Ensuite notre choix s'est porté sur 0 A.D. car il est encore assez 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.

De plus depuis l'ouverture du projet celui-ci a beaucoup progressé et est devenue très complet : partie solo, partie multijoueur, didacticiel, éditeur de scénarios, support pour les mods, ect... , cherchant à atteindre un niveau de qualité proche des jeux commerciaux.

Il serait donc intéressant d'analyser ce logiciel et ceci sous plusieurs angles en commençant par une analyse architecturale en prenant en compte 4 vues : Logical view, Process view, Development view et Deployment view. Une fois cette analyse effectuée nous analyserons la qualité du code grâce à plusieurs métriques puis nous finirons avec l'analyse des Design Patterns présent dans le projet ainsi que leurs avantages et inconvénients.

## Description du logiciel

Avant de nous plonger dans l'analyse formelle du logiciel nous allons décrire le jeu dans son fonctionnement et ses principales interfaces. Il s'agit d'un jeu de stratégie, le but est de battre l'adversaire ou les adversaires suivant un objectif précis. Pour cela on choisit une civilisation que l'on va diriger sur plusieurs aspects. Le jeu se présente avec une Caméra en vue de dessus assurant une vue globale sur la partie et permettant de mieux gérer ses unités et ses bâtiments. La figure 5 permet d'avoir un aperçu de ce concept. Passons maintenant à la présentation du jeu tel que les développeurs l'on conçut.

Au lancement du logiciel nous nous retrouvons donc avec une interface nous permettant soit d'apprendre à jouer, de jouer une partie en solo, de jouer en multijoueur, de régler les paramètres ainsi que d'autres fonctionnalités pouvant aider à l'amélioration du jeu. L'image ci-dessous permet de voir comment ce menu est présenté.



fig. 1 Menu principale du jeu

Si nous décidons de jouer une partie, ce qui est la fonctionnalité centrale de tout jeu vidéo, nous pouvons , comme montré à la figure 2 , 3 et 4, choisir la faction que nous voulons jouer, le type de carte (qui définit le nombre de joueurs), la difficulté des IA , et la possibilité de créer des équipes. Pour le choix du type de cartes il y a à disposition différents filtres : nouvelles cartes, cartes navales, carte de démonstration , carte à déclencheurs et toutes les cartes, ceci permet de ne montrer que les cartes qui nous intéressent. Une fois la carte choisie nous pouvons décider si nous voulons avoir la carte révélée ou non ainsi que la présence de trésors en jeu.

Nom du joueur	Couleur	Position du joueur	Civilisation	Équipe
Joueur 1	▼ murph	▼ Aléatoire	▼ 1	
Joueur 2	▼ IA : Petra Bot	▼ Aléatoire	▼ Aucune	
Joueur 3	▼ IA : Petra Bot	▼ Aléatoire	▼ Aucune	
Joueur 4	▼ IA : Petra Bot	▼ Aléatoire	▼ 1	

**Configuration**

**Plaines de Thessalie (4)**

**Paramètres joueurs**

**Paramètres carte**

**Détails carte**

**Plaines de Thessalie (4)**

**Type de carte:** Escarmouche

**Filtre de carte:** Toutes les cartes

**Choisir la carte:** Plaines de Thessalie (4)

**Désactiver les trésors:**

**Carte explorée:**

**Carte révélée:**

**Conquête** Vaincre les adversaires en tuant toutes leurs unités et en détruisant toutes leurs structures.

**Diplomatie** Les joueurs peuvent établir des alliances et déclarer la guerre à leurs alliés.

**Victoire alliée** Si un joueur gagne, ses alliés gagnent aussi. S'il reste un groupe d'alliés, ceux-ci gagnent.

**Cessez-le-feu désactivé**

**Nom de la carte** Plaines de Thessalie (4)

**Description de la carte** La houleuse plaine de Thessalie est traversée par des ruisseaux étroits, faciles à traverser. De larges espaces ouverts permettent l'expansion massive, et chaque joueur commence la partie à l'abri en haut d'une large acropole.

**Type de carte** Escarmouche

**Filtre de carte** Toutes les cartes

**Centre-villes** Les joueurs commencent avec un Centre-ville

**Ressources de départ** Faibles (300)

**Limite de population** 300

**Retour** **Démarrer !**

fig. 2 Configuration nouvelle partie / Paramètres carte

Une autre possibilité est de choisir la capacité limite de population dans une civilisation ainsi que le nombre de ressources de départ pour personnaliser sa partie et ne pas vivre toujours la même expérience de jeu.

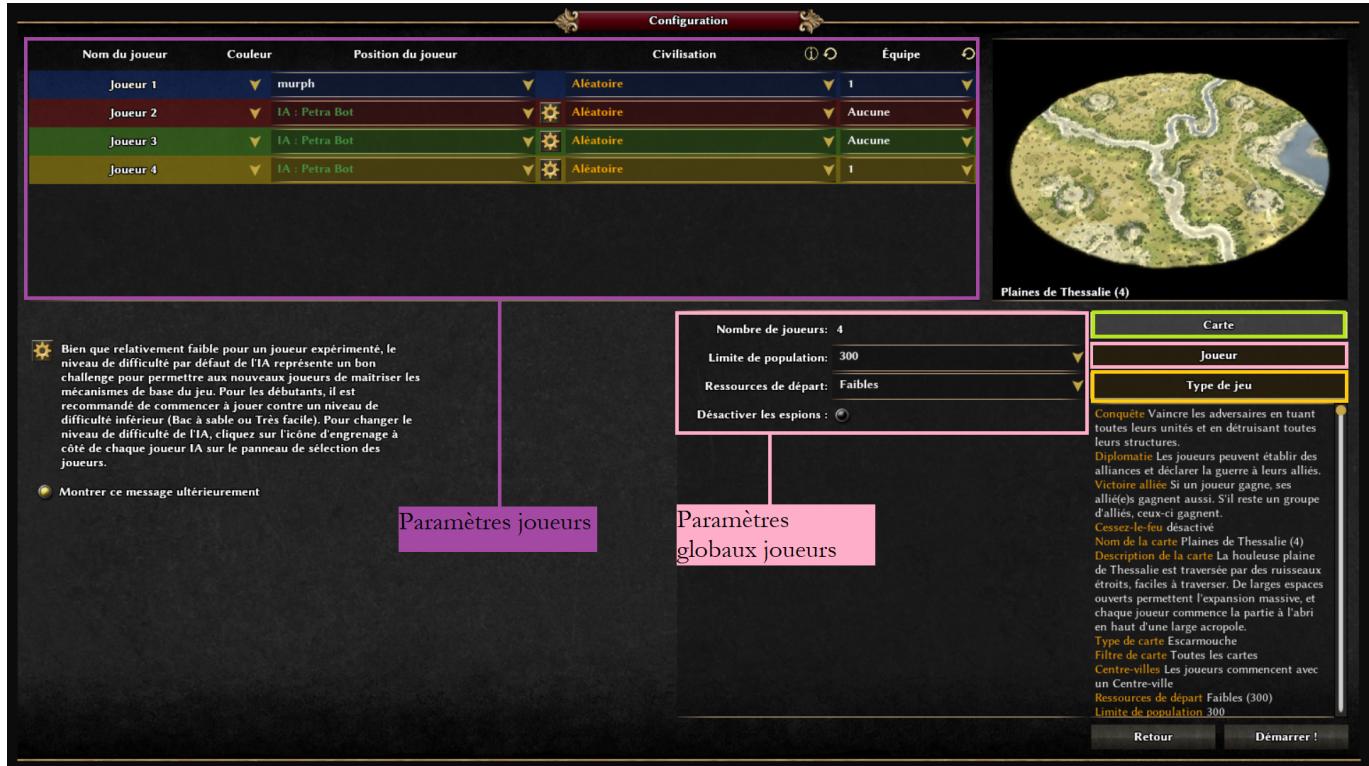
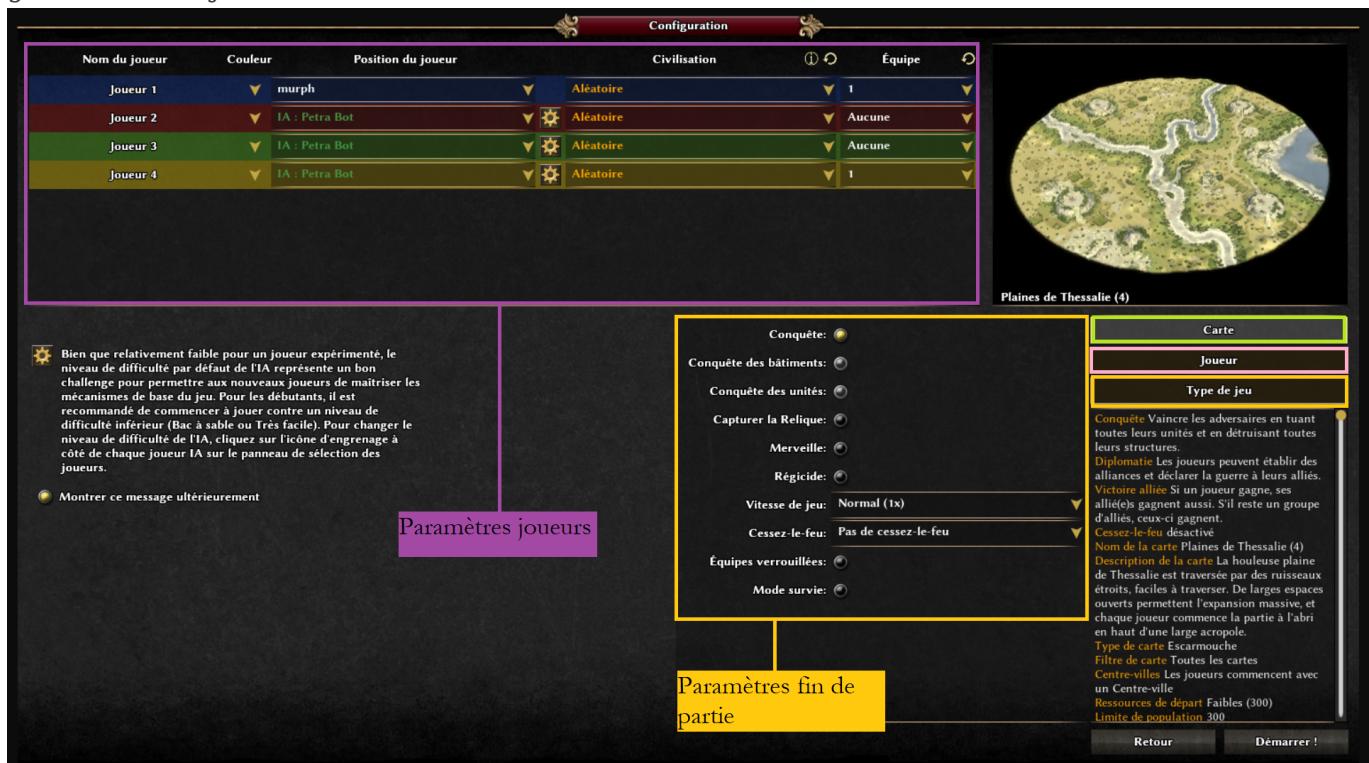


fig. 3 Configuration nouvelle partie / Paramètres global joueurs

La manière de terminer / gagner une partie peut également être déterminée par l'onglet "type de jeu". Nous pouvons aussi ajouter plusieurs conditions simultanées pour gagner ce qui complexifie grandement le jeu.



#### fig. 4 Configuration nouvelle partie / Paramètres fin de partie

La configuration d'une partie est donc très complète et au même niveau que des jeux tels que Starcraft ou encore Age of Empires.

Une fois la partie démarrée nous nous retrouvons avec notre civilisation et comme point de départ un centre ville entouré de quatre ouvrières, deux soldats de corps à corps (infanterie), un cavalier, et deux soldats de combat à distance (archerie). Ces unités de départ peuvent récupérer différentes ressources et chacune des unités a des bonus différents dans la récolte d'une ressource ce qui permet de les différencier lorsque l'on doit récolter. Il existe plusieurs ressources récoltables qui sont la nourriture, les pierres, le métal et le bois. Grâce à cela les unités peuvent construire des bâtiments pour faire évoluer et développer notre civilisation. L'évolution se fait à travers 3 phases. La première de base est celle des villages, la seconde la phase des villes, et la dernière celle des cités. Le fait de changer de phase permet de débloquer de nouveaux bâtiments avec de nouvelles fonctionnalités. Mais pour débloquer ces phases avec les avantages qu'elles possèdent il faut remplir des conditions, par exemple un nombre de population, la présence de certains bâtiments de l'âge actuel, forçant le joueur à devoir se développer un peu avant de voir les meilleures unités débloquées.

Pour combattre les autres civilisations nous utilisons les unités à disposition qu'il faut entraîner dans des bâtiments spéciaux comme les casernes. Le jeu propose d'ailleurs une grande diversité suivant la civilisation choisie lors de la configuration du jeu. Le but est donc, dans le cas d'une victoire militaire, de détruire les bâtiments et unités adverses grâce à nos propres unités. Il existe plusieurs catégories : l'infanterie, l'archerie, la cavalerie et les armes de siège. Chacune ayant un rôle précis et devant être utilisée correctement pour s'assurer la victoire.

Nous pouvons également mettre des unités dans certains bâtiments soit pour les protéger soit pour qu'ils attaquent automatiquement les ennemis. Mais le jeu ne se limite pas à la guerre on peut également faire du commerce avec des joueurs neutres ou alliés, ou utiliser la diplomatie pour gagner contre un joueur puissant.

Le jeu intègre également une notion de territoires qui limite la construction des bâtiments à notre territoire uniquement (sauf les postes avancés qui peuvent être en territoire neutre). Pour étendre notre territoire nous devons bâtir nos bâtiments proches de nos frontières, cela va participer à son expansion au fil du temps et ainsi notre domination sur la carte. Il existe une façon plus rapide de s'étendre, la construction de nouveaux centres de ville sur la carte. Ceux-ci peuvent se construire sur les territoires neutres. Cette mécanique n'est pas à prendre à la légère puisqu'elle permet de bloquer l'avancée d'autres civilisations en récupérant des ressources.



fig. 5 Interface de jeu lors d'une partie

Ce premier pas dans les nombreuses fonctionnalités du jeu permet dans un premier temps de comprendre le but recherché par les développeurs : créer un jeu de stratégie open source le plus complet possible avec un grand niveau de qualité. Puis dans un second temps de mieux comprendre l'analyse des fonctionnalités présente dans la prochaine partie.

## Analyse architecturale

### Les différents acteurs et scénarios possibles

Pour comprendre le fonctionnement de ce logiciel nous avons tout d'abord analysé l'architecture de celui-ci en commençant par identifier les différents acteurs qui peuvent intervenir dans le déroulement du jeu et surtout les fonctionnalités proposées par le jeu. 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 détaillant de manière hiérarchique les fonctionnalités proposées dans le jeu. Pour finir sur cette partie nous avons créé un diagramme des cas d'utilisation pour la partie la plus intéressante et la plus complète : le déroulement d'une partie.

#### 1. Présentation des grandes fonctionnalités

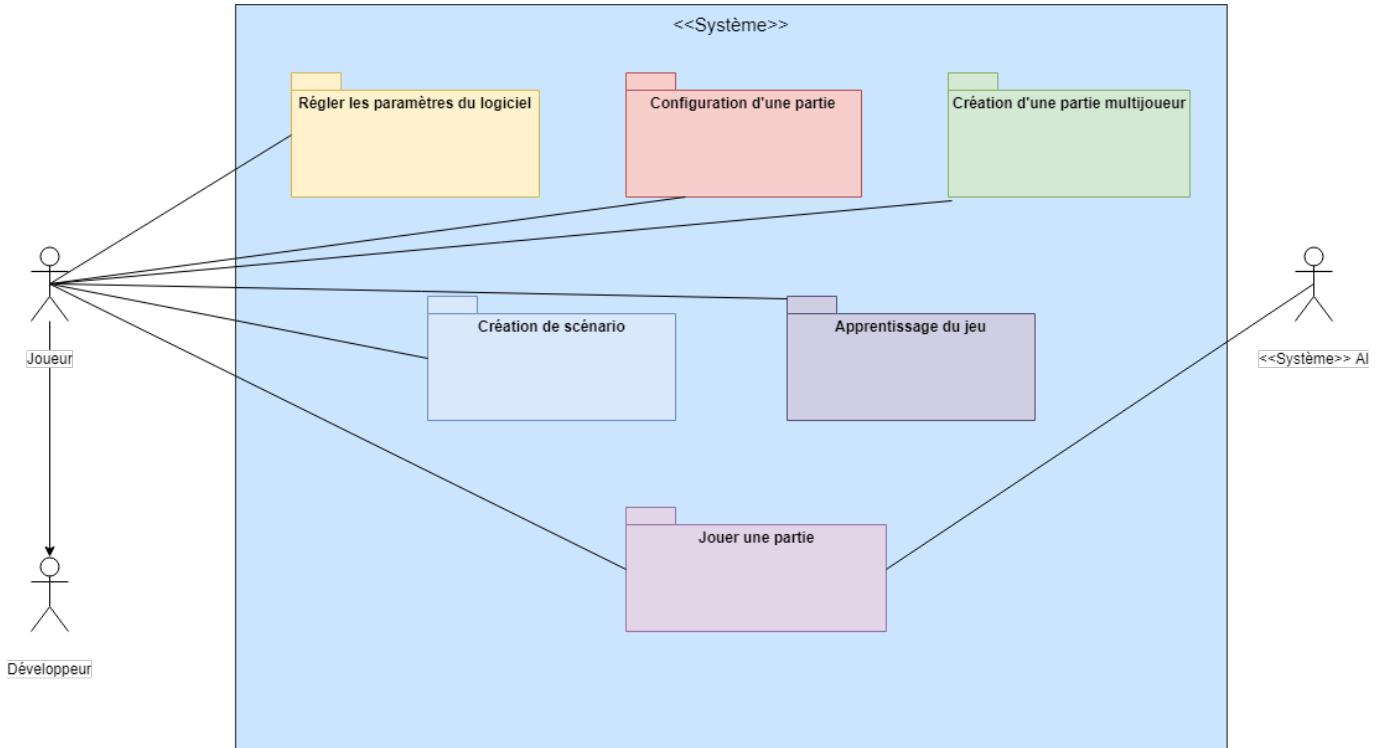
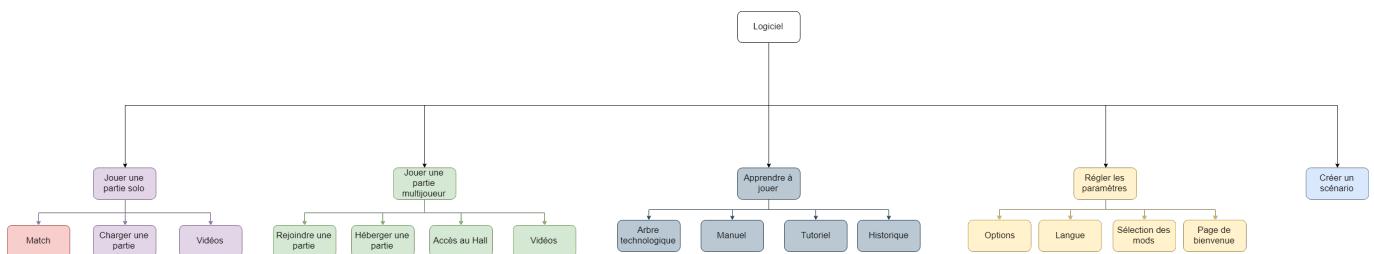


fig. 6 Diagramme représentant les grandes fonctionnalités du jeu

Nous avons donc diviser en 6 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 joueur peut, en plus de la possibilité d'utiliser une console, l'IA ne peut que jouer une partie. Le joueur a donc un nombre important de fonctionnalités à disposition ce qui montre la volonté des développeurs de créer un jeu rivalisant en terme de fonctionnalité avec des jeux commerciaux. Cet aspect est important car si le jeu a le mérite de proposer beaucoup de fonctionnalités nous verrons par la suite que l'architecture en souffre quelque peu.

## 2. Présentation des fonctionnalités



En ce qui concerne l'organisation des fonctionnalités, celle-ci est très classique et on sent que le jeu s'est inspiré de son père spirituel Age of Empires. Les parties solo et multijoueur du titre sont séparées ce qui montre une envie de proposer plusieurs façons de jouer. On trouve également en premier dans le menu du jeu la possibilité d'apprendre à jouer avec un manuel, des tutoriels et des informations sur les éléments du jeu, encore une fois beaucoup de possibilités et une envie évidente d'inviter le joueur à rejoindre la communauté du jeu dans les meilleures conditions. Le jeu propose encore deux fonctionnalités. La première permet de régler le jeu selon ses envies : graphismes, contrôle, son, ... on peut même apercevoir la gestion des mods qui permet à qui le veut de

personnaliser le logiciel de manière très poussé. Encore une fois très complet. La dernière très intéressante permet de créer à l'envie des cartes pour le jeu, cet outil très puissant est d'ailleurs utilisé par les game designers pour générer des fichiers xml automatiquement représentant la carte à intégrer dans le jeu. Elle est ensuite lu par le moteur qui pourra la stocker en mémoire et l'afficher. L'outil qui s'appelle Atlas propose de nombreuses fonctionnalités non détaillé ici comme la création de terrains avec différents peinceaux, la possibilité de peindre le terrain avec des textures, ... . Le jeu se veut donc à tout point de vue complet et intègre un nombre impressionnant de fonctionnalité.

### **3. Cas d'utilisation d'une fonctionnalité**

Pour finir notre analyse des fonctionnalités proposées par le jeu nous avons détaillé une des grandes fonctionnalités 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.

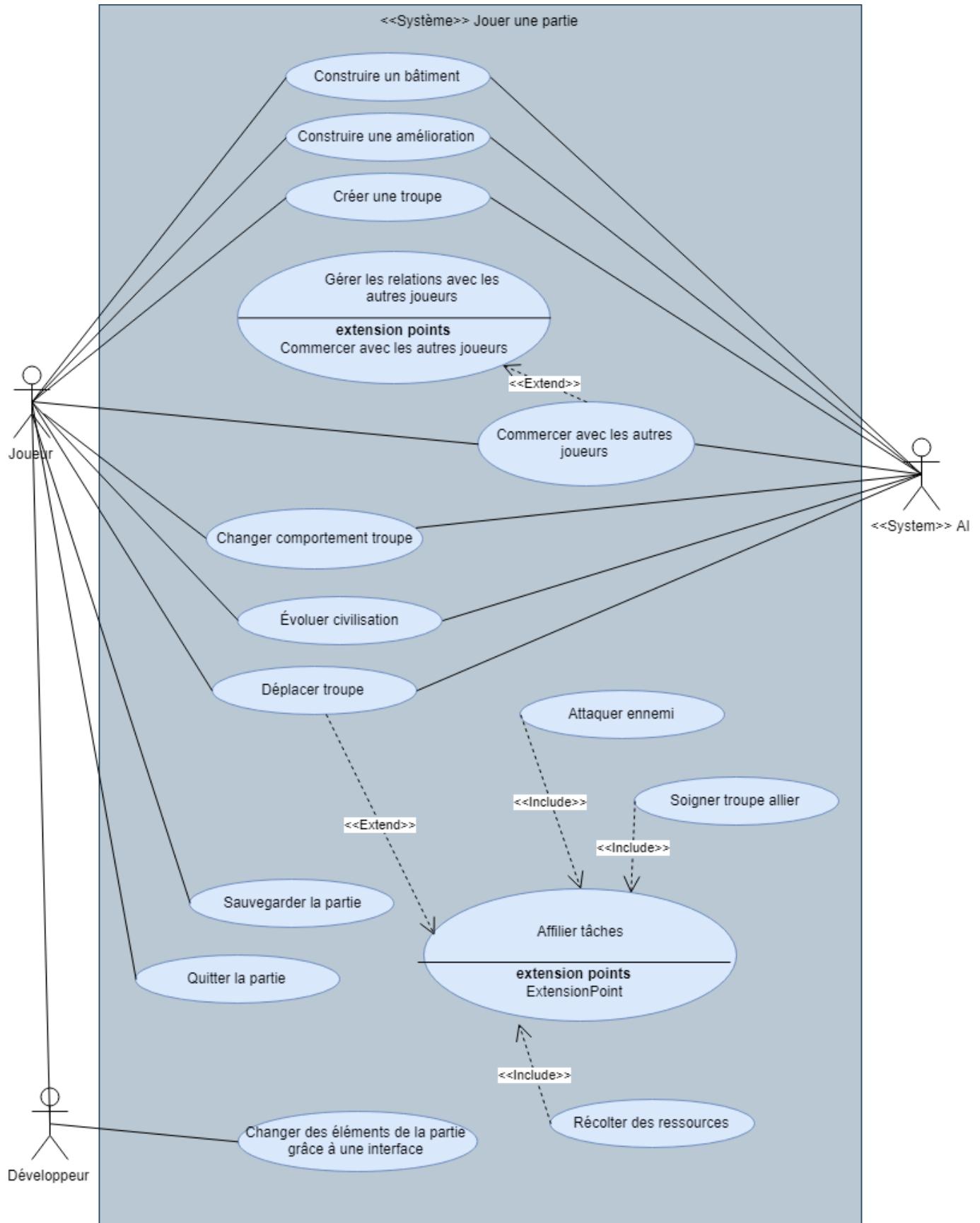


fig. 7 Diagramme des cas d'utilisations pour la fonctionnalité "Jouer partie"

Le diagramme qui nous intéresse 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. Le jeu a donc besoin d'implémenter un nombre important d'élément comme la gestion des troupes, la construction de bâtiment, le commerce, la guerre, la récolte de ressources, ... . 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 éléments non implémentés.

Ce diagramme permet donc de mieux appréhender les fonctionnalités mises en avant dans le jeu et de voir comment ces fonctionnalités ont été intégré dans l'architecture du projet grâce à la partie suivante portant sur les packages.

## **Diagramme des packages (Development View)**

Notre analyse des packages a pour but de mieux comprendre comment le projet a été construit et s'il mériterait ou non des améliorations. Comme premier point nous avons remarqué que les packages du projet représentent souvent la fonctionnalité "technique" qu'ils traitent comme le rendu, les graphismes, ou encore l'intégration des scripts dans le moteur. Ce qui donne le diagramme suivant:

### **1. Disposition des packages**

Comme point de départ pour comprendre la répartition des packages dans le code source nous avons utilisé le diagramme fourni par CodeScene ci-dessous.

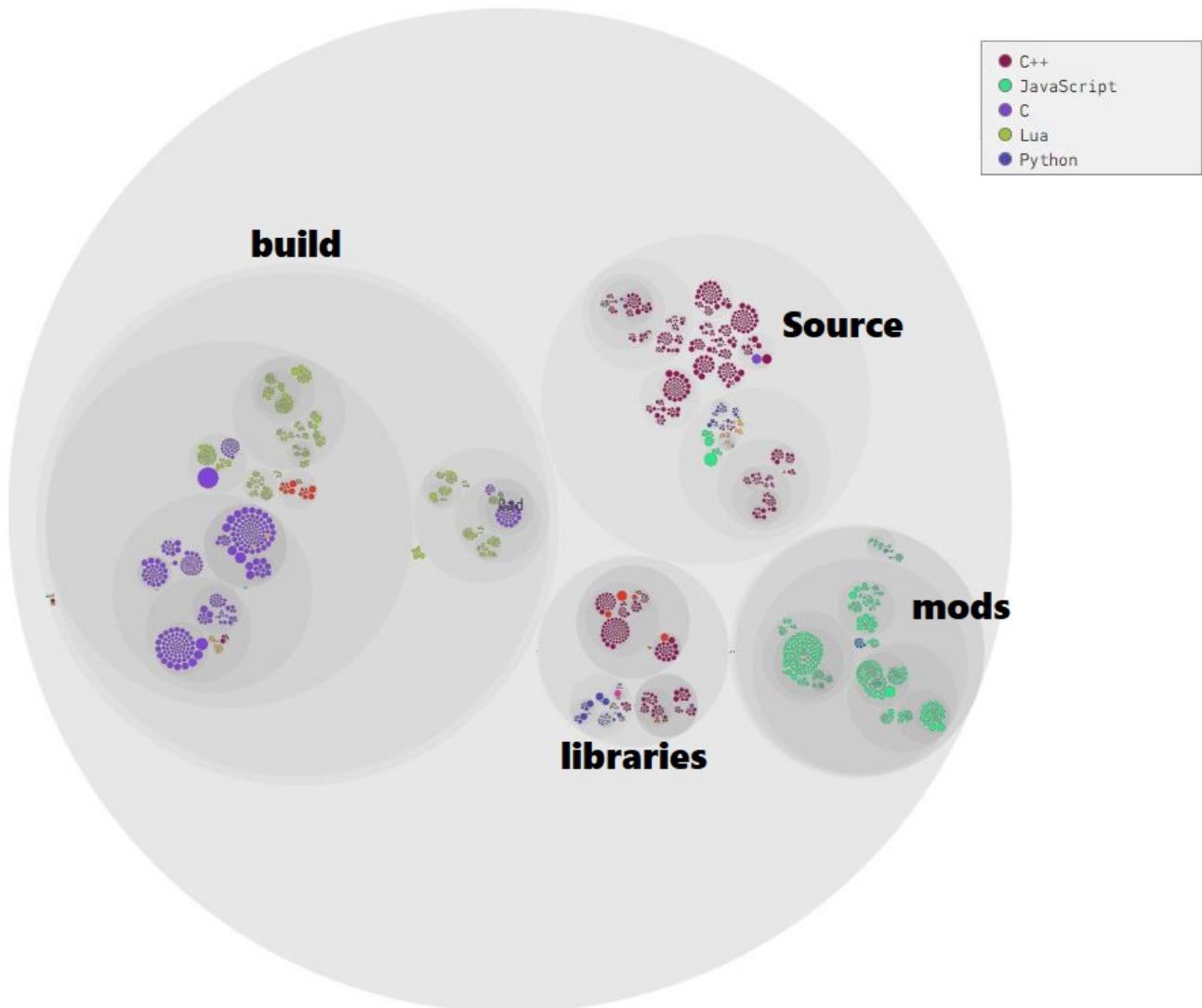


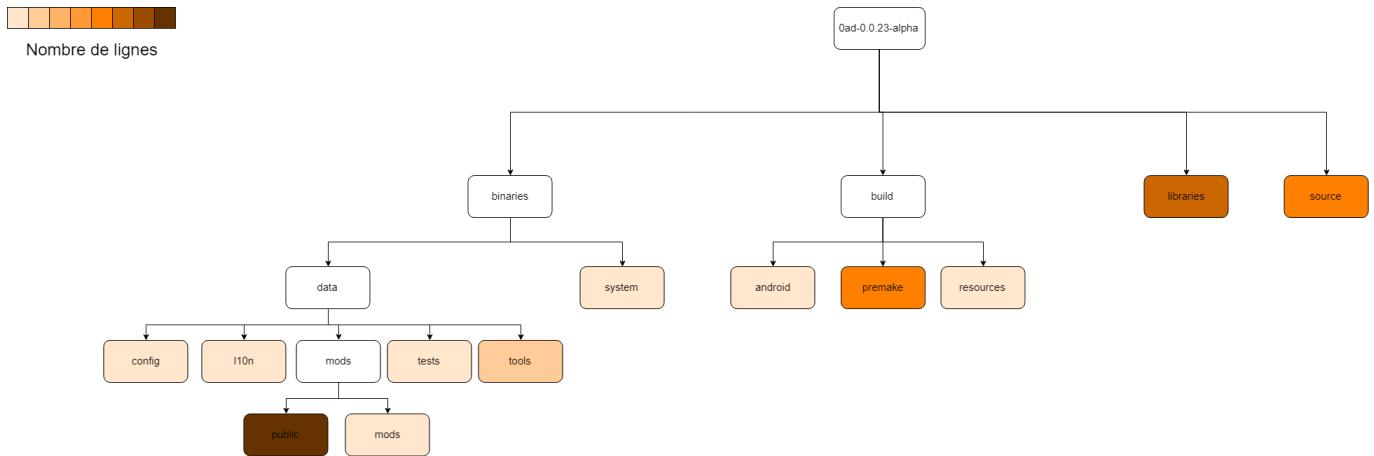
fig. 8 Diagramme CodeScene avec les langages les plus utilisées par packages

Celui-ci a la particularité de montrer la répartition des langages à travers les packages mais aussi un autre problème dont nous parlerons en détail dans la partie suivante. Dans source on retrouve majoritairement le moteur du jeu écrit en C++. Dans le dossier build on retrouve les outils externes au logiciel qui seront ajoutés au moment de la compilation comme Premake un outil open source pour optimiser la compilation, les langages majoritaires sont le C pour premake notamment et le lua. Libraries contient comme son nom l'indique les libraries que le projet utilise, le langage utilisé est ici majoritairement du C++. Enfin le dossier mods qui se situe non à la racine du code source mais dans Binaries contient comme son nom l'indique les mods mais également les scripts javascript qui seront interprétés par le moteur Pyrogenesis.

Grâce à ce diagramme nous avons pu voir que le jeu était en fait divisé en 2 parties. D'un côté le moteur du jeu Pyrogenesis écrit en C++ dans le dossier source et de l'autre les éléments du jeu lui-même comme le joueur ou l'IA dans le dossier mods/public/ écrit en javascript et relié dans le code source du moteur par une classe C++ : CScriptInterface. Celle-ci permet d'appeler des méthodes ou de récupérer des données d'un script.

## 2. Répartition du code dans l'architecture

Nous allons maintenant aborder un point important de l'architecture et qui selon mériterait une amélioration. Le problème qui concerne la répartition du code source dans les différents dossiers est illustré dans le diagramme suivant.



Le diagramme représente les principaux packages du projet organisé sous la forme d'un arbre. Le but de ce diagramme est de montrer la répartition des lignes de codes à travers les packages. La première chose qu'on peut remarquer avec ce diagramme c'est que parmi tous les package représentés, seuls 4 d'entre eux intègrent la quasi totalité du code source dont le plus important "public" avec ses plus de 2 000 000 de lignes dont 112.152 lignes de javascript, le reste étant essentiellement des fichiers xmls servant à stocker les informations. Le second point concerne la profondeur de l'arbre à laquelle se trouve les fichiers javascript dans "public", ils sont difficiles à trouver de plus leur emplacement ne mets pas en avant leur lien très étroit avec le moteur du jeu, il est facile de s'imaginer qu'il s'agit de modes supplémentaires et optionnelles alors qu'il n'en est rien. Nous avons donc affaire ici à un problème de visibilité sur le projet qui impacte la compréhension du projet plus que les performances par exemple.

Pour résoudre ce problème nous pourrions imaginer une organisation différente du code source notamment en ce qui concerne le javascript du jeu. Ceci obligera par contre une distinction dans le projet entre les modes des utilisateurs et les modes dit public ce qui demandera une réécriture partielle du moteur pouvant créer des instabilités involontaires.

## 3. Lien entre les packages

Le second problème de cette architecture c'est qu'elle occasionne beaucoup de dépendances entre les packages ce qui peut compliquer énormément le refactoring. Il aurait pu être intéressant de diviser le projet en services et fonctionnalités orientées utilisateurs à la place comme on peut le voir avec le package "Atlas" qui est l'éditeur de scénarios du jeu. Le diagramme des dépendances qui suit montre bien le problème. On remarque rapidement que les packages s'appellent beaucoup entre eux ce qui rend le tout assez illisible.

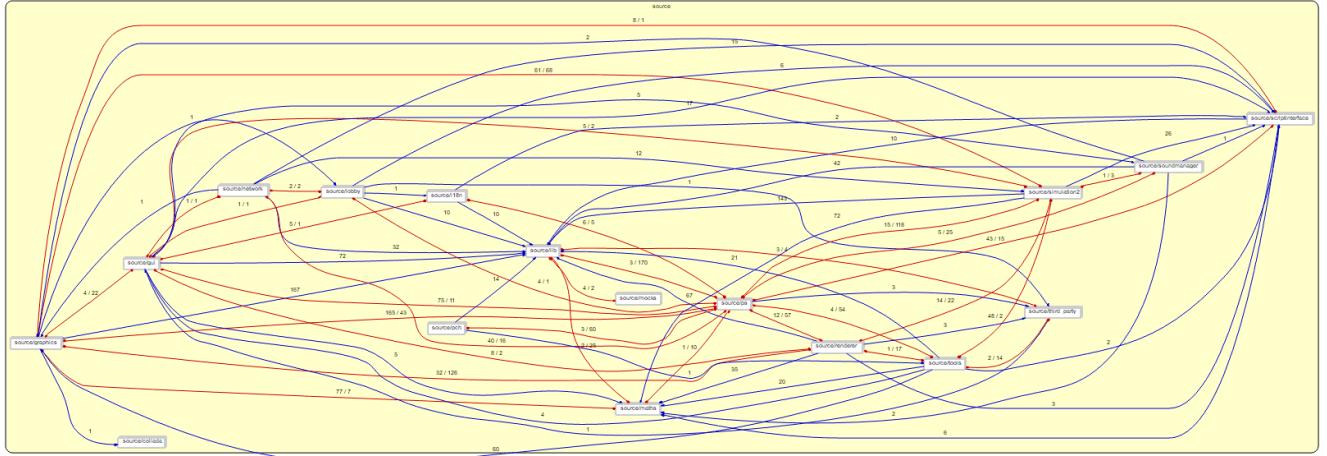


fig. 9 Diagramme des dépendances entre packages

Celui-ci permet de remarquer que les différents packages ne sont pas tous bien organisés en dossiers plus petit, ce qui peut poser problème pour la compréhension du projet, par exemple dans le package "graphics" qui est très petit, on trouve 118 fichiers, 40 classes et 25 735 lignes de code ce qui rends l'exploration laborieuse. De plus le diagramme de package ne permet pas sous cette forme de reconnaître les éléments les plus importants de l'architecture comme la gestion des graphismes dans "graphics". En effet ce package possède un diagramme de classe très imposant comme montré ci-dessous à la figure 11.

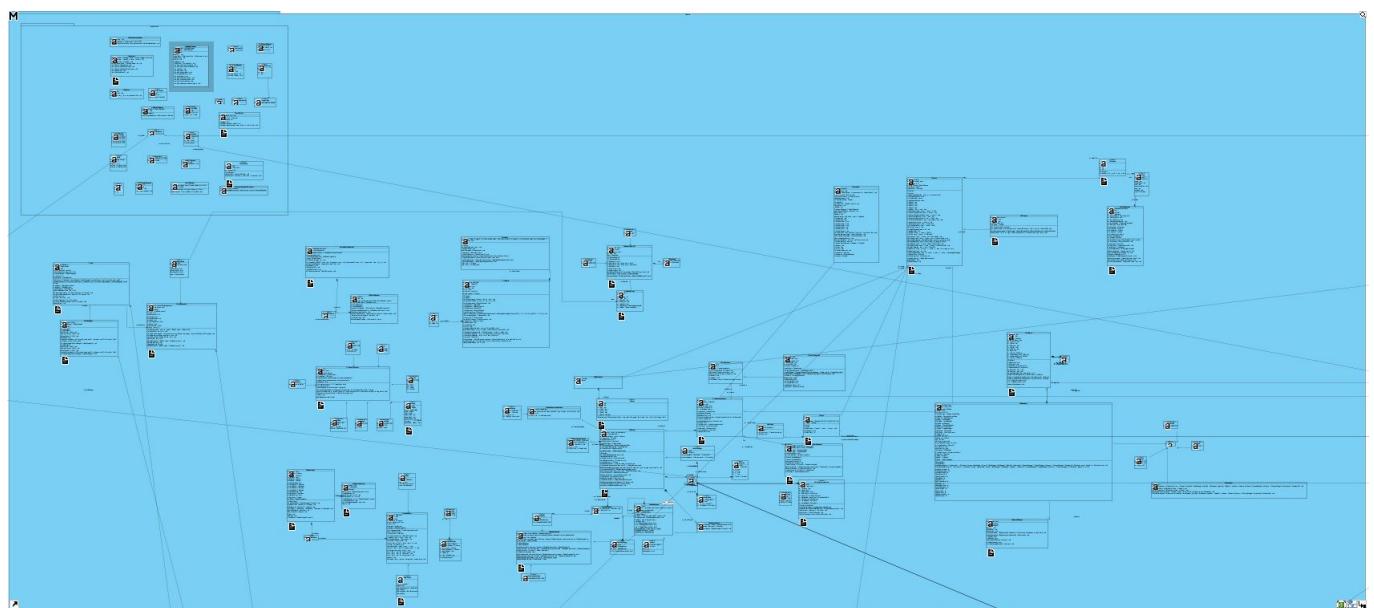


fig. 11 Diagramme des classes du package "graphics"

Mais celui-ci n'est pas divisé en sous-partie comme dans le package tools (voir figure 12).

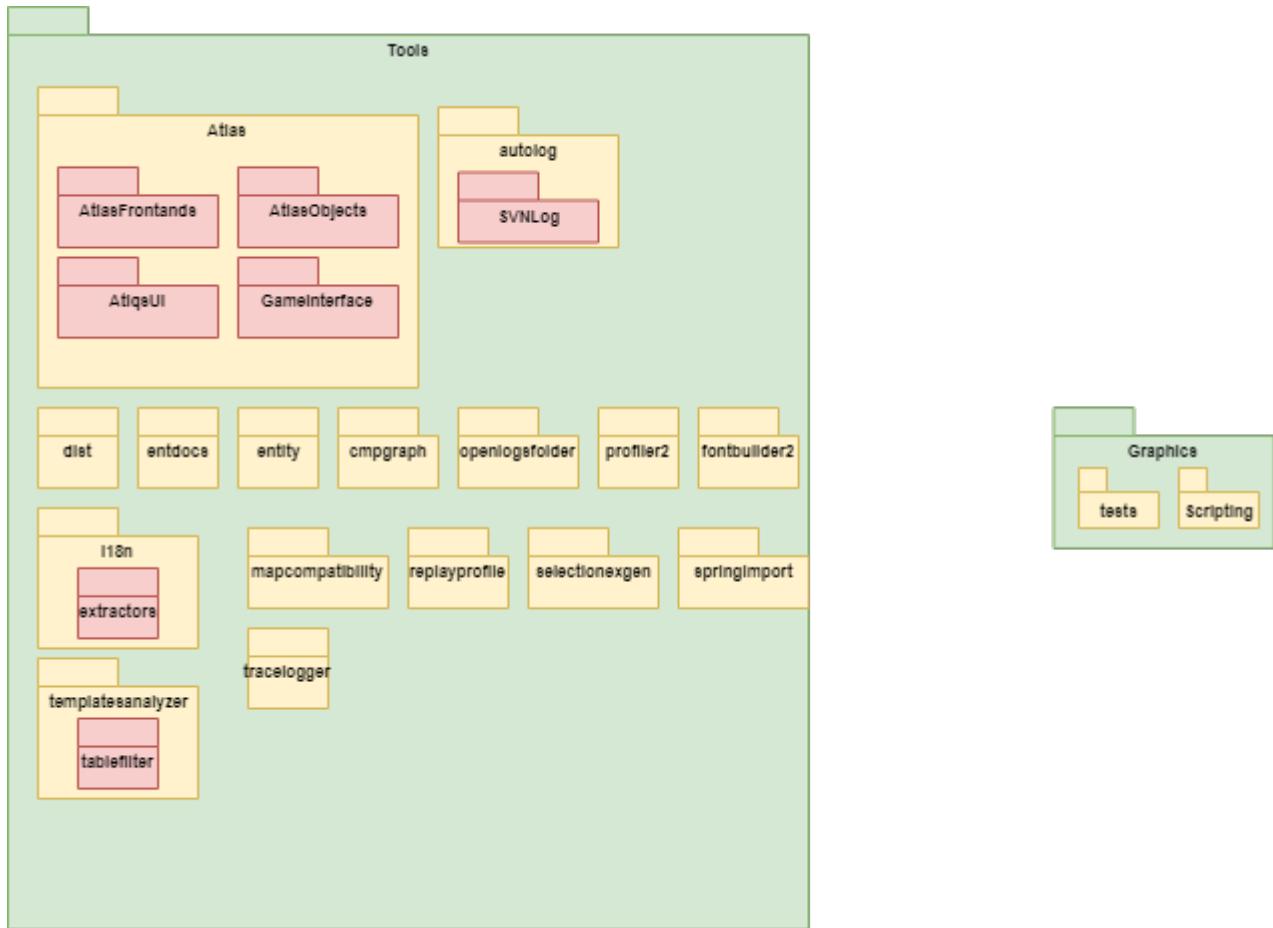


fig. 12 Diagramme de packages du package "Tools" et du package "Graphics"

L'organisation des packages pourrait donc être un point intéressant à améliorer pour le projet pour offrir une meilleure visibilité et faciliter le refactoring de certaines parties du code.

## Logical View

Le diagramme des classes étant assez grands et complexes, regroupant 655 classes dans 17 packages. Il comprend également 298 228 lignes de code, en voici un aperçu non exhaustif:

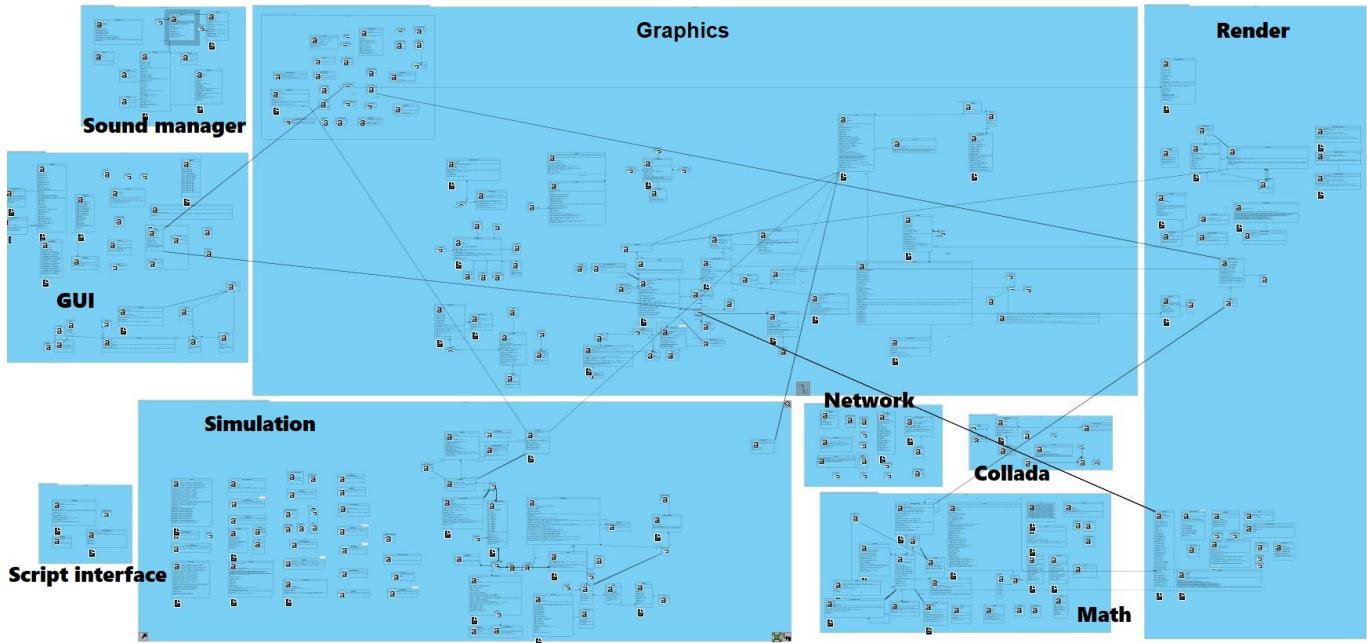
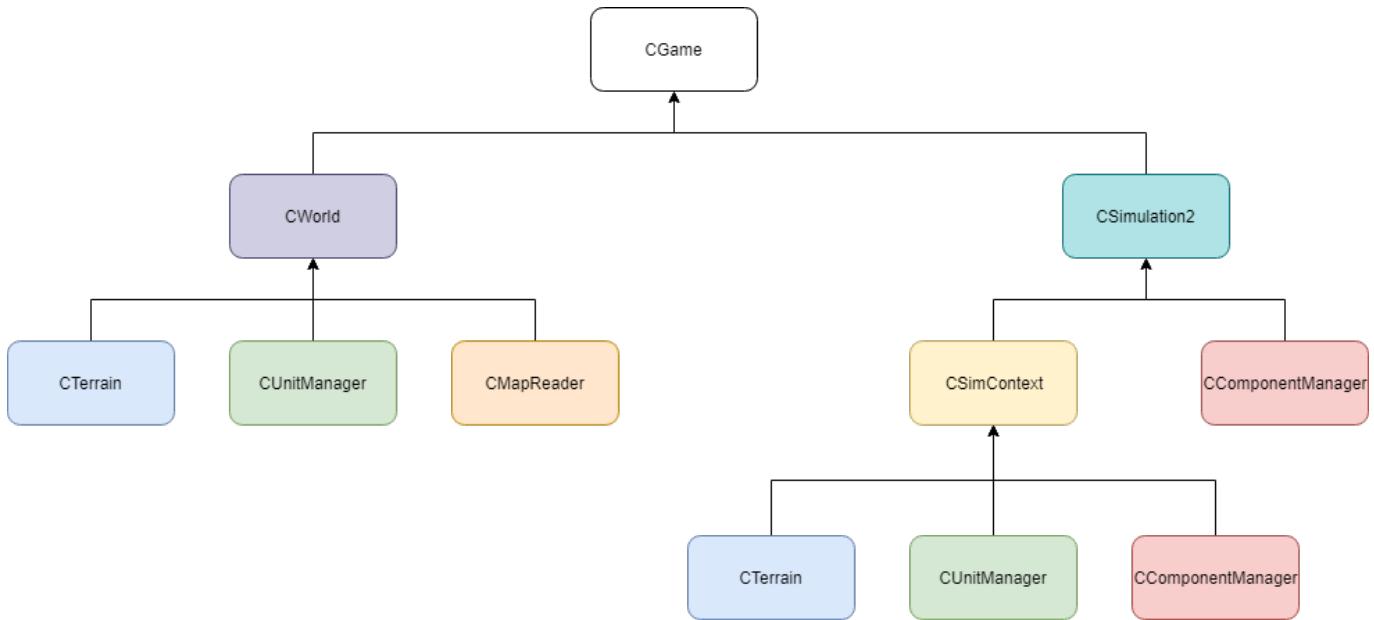


fig. 13 Diagramme des classes non exhaustif du package source

## 1. Structure globale du code source

Pour représenter la structure globale du package nous avons fait un diagramme avec les principales classes permettant de gérer le jeu.



Le principale point à retenir est cette architecture en arbre. Il existe plusieurs grandes classes dans le code qui regroupe d'autres éléments plus petits et ainsi de suite on retrouve donc une certaine hiérarchie dans la structure logique du moteur. Globalement le jeu est orchestré par la classe qui contient le monde de manière générale et qui permet de connecter les éléments entre eux pour créer un monde vivant. On utilise donc ici des classes englobantes qui favorisent la composition plutôt que l'héritage entre les classes, ceci permet également de mieux organisé le code.

## 2. Implémentation de la gestion des unités

Pour donner un exemple plus précis nous avons donc décidé de présenter un élément en particulier du diagramme et une des fonctionnalités de base du jeu : la gestion des unités.

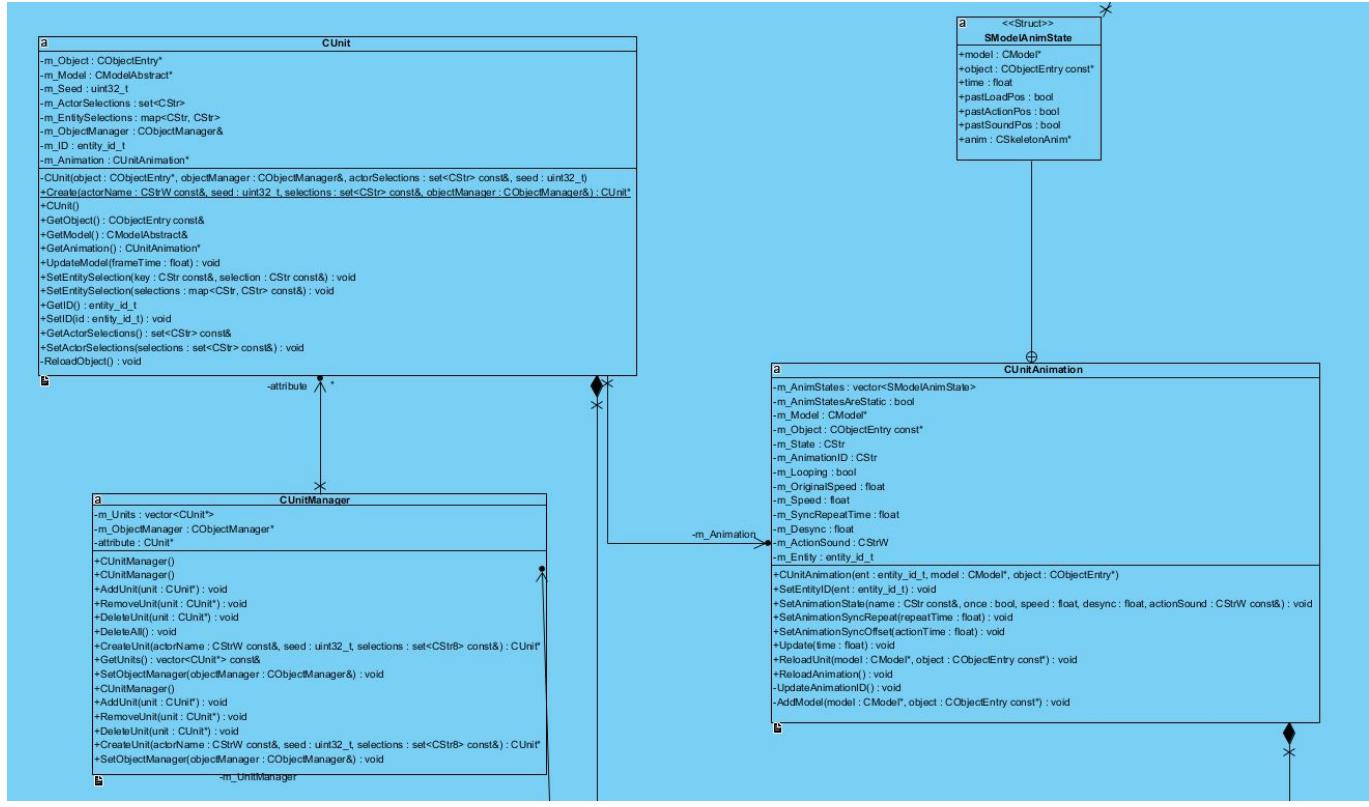


fig. 14 Partie du diagramme des classes concernant la gestion des unités

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 code que voici.

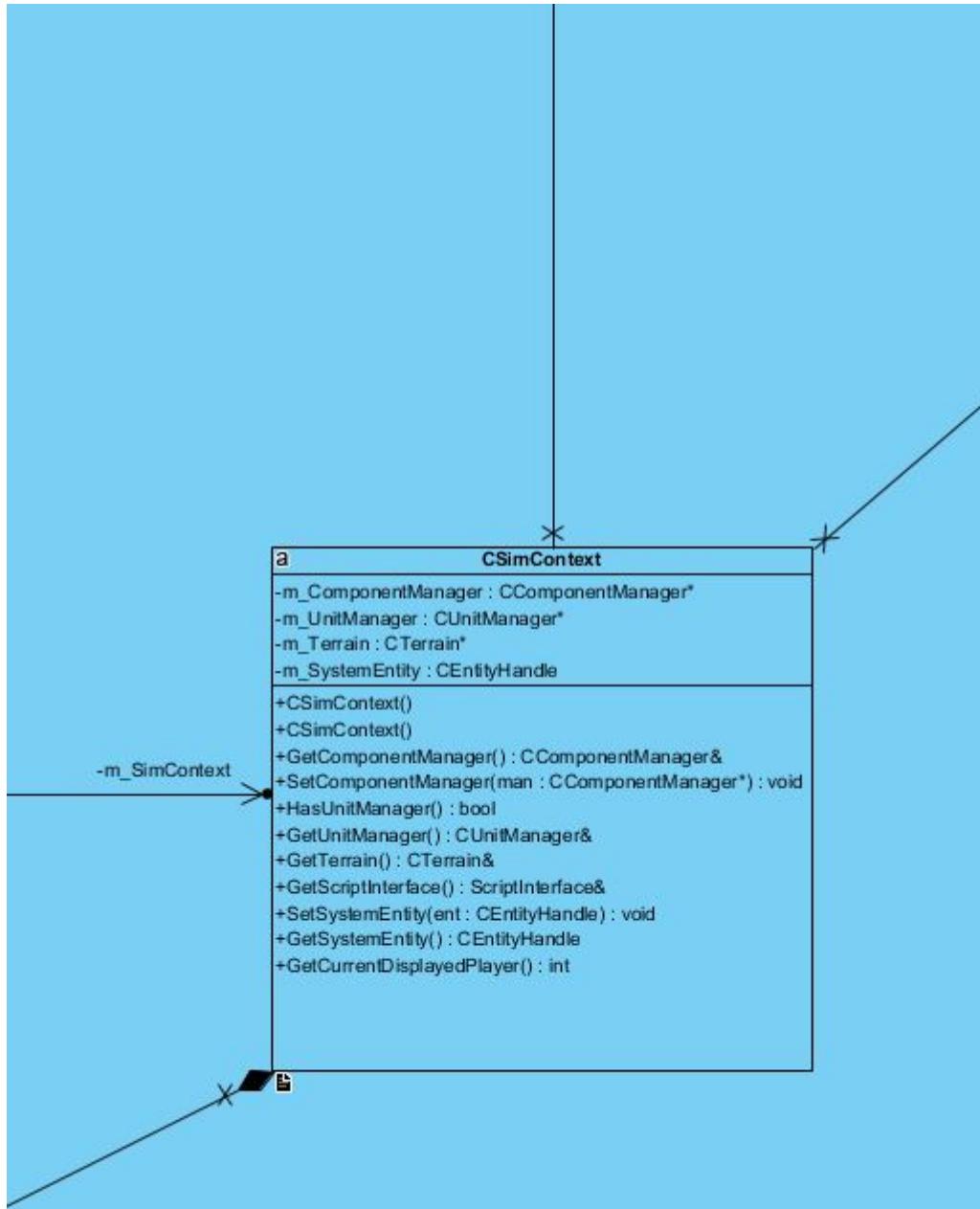


fig. 15 Diagramme UML de la classe "CSimContext" du package Simulation2

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.

### 3. Problème de la structuration des classes

Dans cette partie nous allons juste aborder un point intéressant dans lequel nous reviendrons dans la partie sur la qualité du code. Un problème flagrant lorsque l'on analyse les classes c'est la taille de certaines d'entre elles. En effet certaines classes arrivent à dépasser le milliers de lignes. Il pourrait donc être intéressant de subdiviser ces classes en classes plus petites ce qui encore une fois permettrait de débuguer ou de faire du refactoring plus facilement. De plus cette façon de faire irait dans le sens de la structure logique générale du code.

## Process View

### 1. Diagramme de séquences lors de l'action d'un joueur

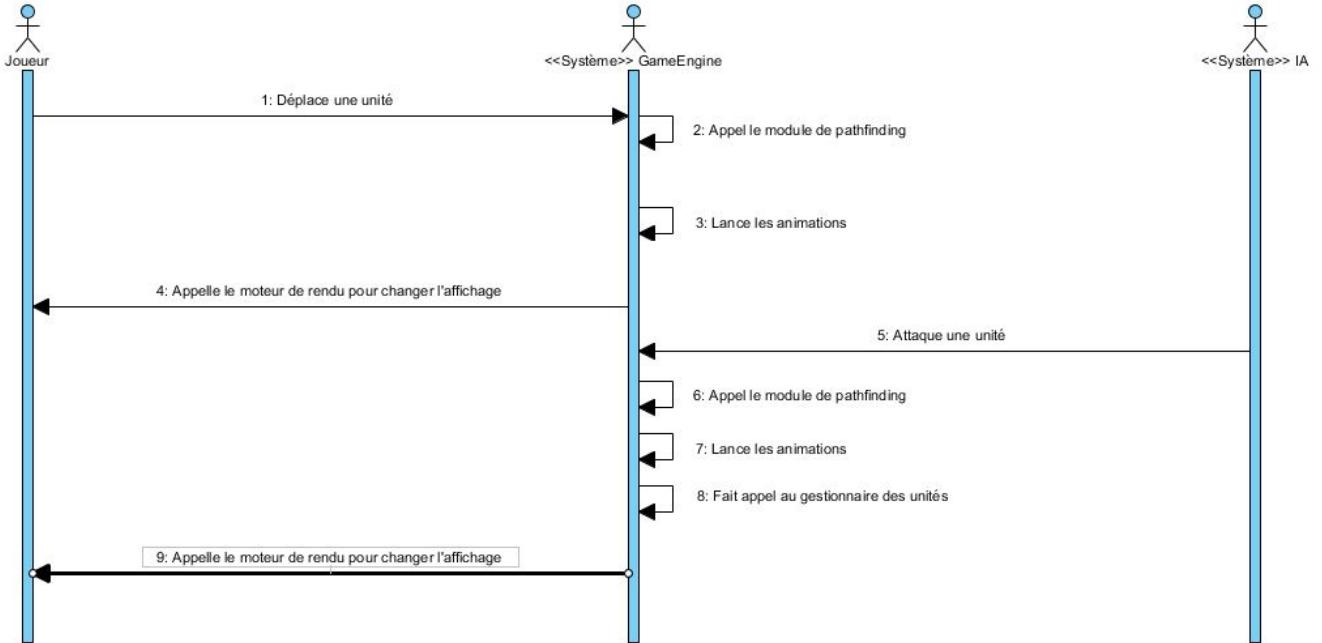


fig. 16 Diagramme de séquence dans le cas d'une partie jouée contre une IA

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 appelle en conséquence les bons modules chacun gérant une partie du logiciel comme l'IA, le rendu, la caméra, le pathfinding, les animations, etc ... .

### Conclusion sur l'architecture

D'après notre analyse le logiciel possède deux problèmes qui mériteraient une amélioration, le problème des packages pas toujours très clair et le problème des classes trop grandes qui mériteraient d'être subdiviser.

### Analyse de la qualité du code

Pour analyser la qualité du code nous allons utiliser différentes métriques sur certaines parties de celui-ci. Pour déterminer si le code répond à ces critères nous avons utilisé différents logiciels , notamment CodeScene , BetterCodeHub et CppCheck. Les critères que nous avons retenu sont :

La longueur de classe /

La longueur des méthodes /

Le nombre de paramètres /

Sur les 3.436.103 de lignes de code disponibles nous avons porté notre analyse sur certains dossiers

qui nous semble plus important pour le jeu.

## source/simulation2

Nous avons choisi de regarder ce dossier car il nous semble une des parties centrales du jeu. Il définit les classes utilisées par la simulation. Les composants qui implémentent la logique du moteur sont inclus ici, ainsi que les classes de sérialisation et d'aide. Certains composants sont implementés en JavaScript et définissent optionnellement les interfaces JavaScript et C++ (ces composants sont inclus par mods, dans le répertoire binaries\data).

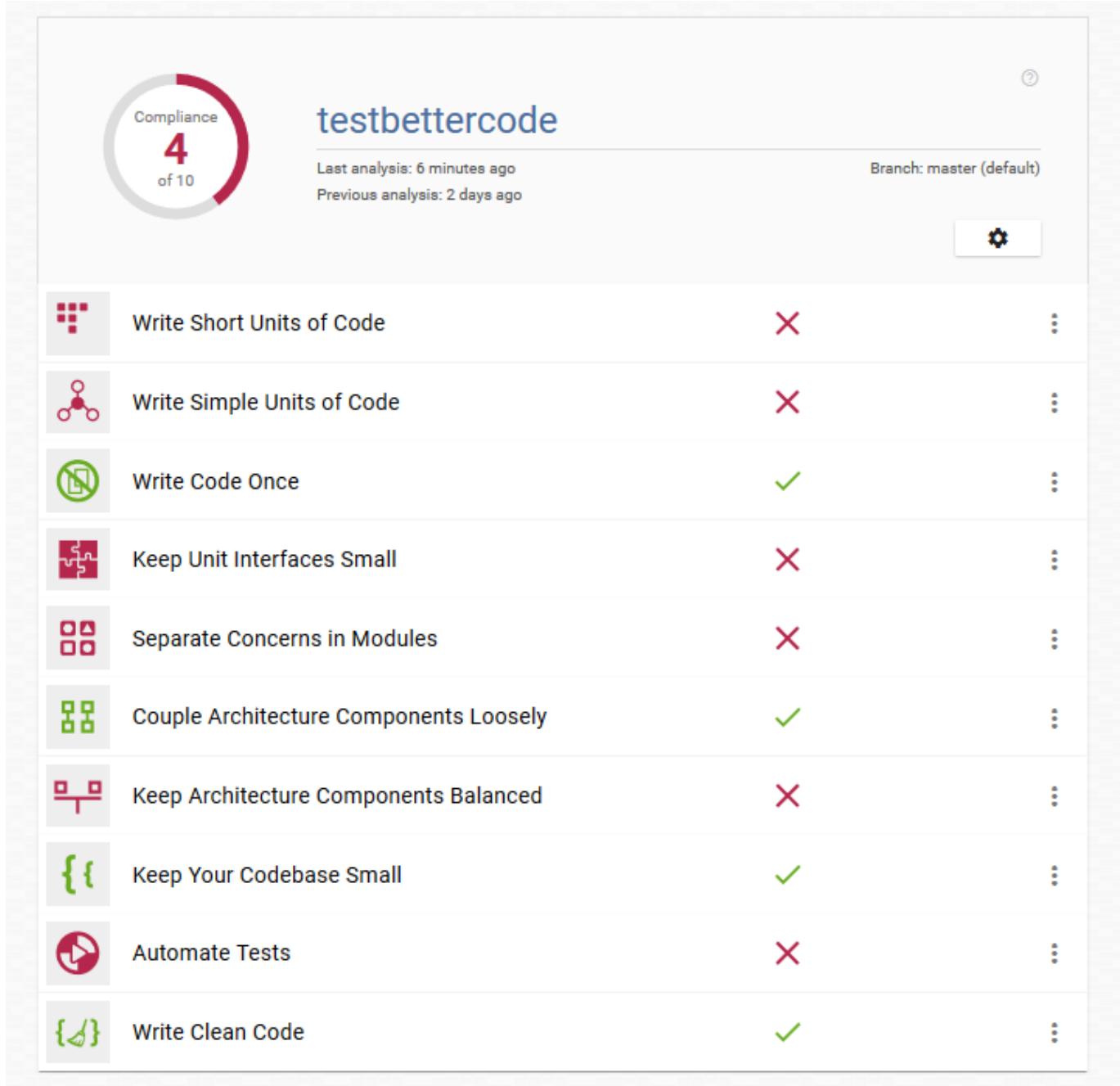


fig. 17 Analyse de la qualité du dossier simulation2 avec BetterCodeHub

D'après la figure ci-dessus nous pouvons constater que ce dossier malgré son importance comporte de nombreux points faibles du point de vue de BetterCodeHub.



fig. 18 Analyse de la qualité sur la taille des méthodes

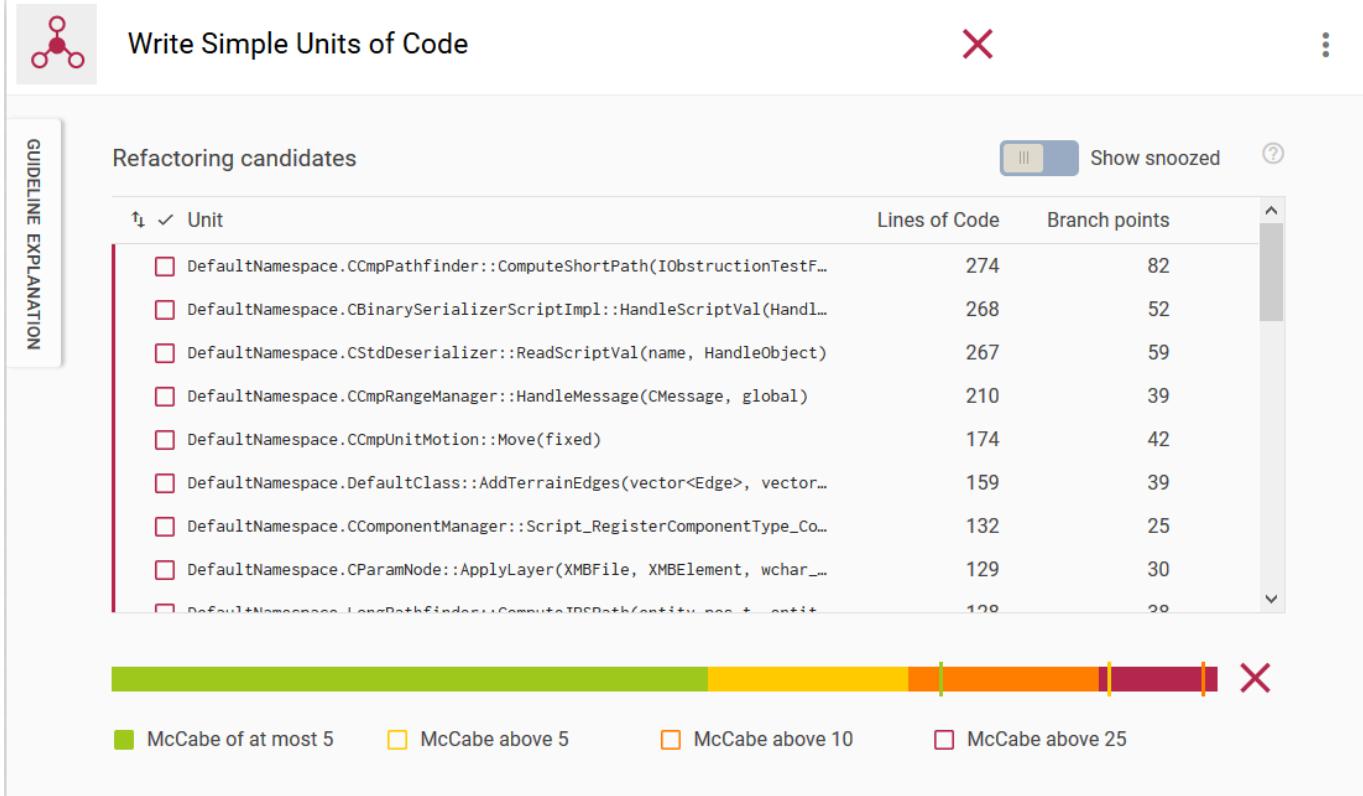


fig. 19 Analyse de la qualité sur la simplicité des méthodes

Dans ces analyses nous pouvons voir que certaines méthodes font largement plus que les 15 lignes de codes recommandées. En regardant plus attentivement les fichiers concernés , nous pouvons en déduire qu'il serait effectivement possible de plus fractionner en des sous fonctions pour réduire ce nombre de lignes. Ce qui aurait aussi comme conséquences d'augmenter la simplicité des méthodes.



## Keep Unit Interfaces Small



GUIDELINE EXPLANATION

### Refactoring candidates

[Show snoozed](#) [?](#)

Unit	Lines of Code	Parameters
DefaultNamespace.CCmpPathfinder::ComputeShortPath(IObstructionTestF...	274	8
DefaultNamespace.DefaultClass::AddTerrainEdges(vector<Edge>, vector...	159	8
HierarchicalPathfinder.Chunk::RegionNearestNavcellInGoal(u16, u16, ...	96	7
DefaultNamespace.CCmpObstructionManager::TestStaticShape(IObstructi...	52	7
DefaultNamespace.DefaultClass::SplitAAEdges(CFixedVector2D, vector<...	48	8
DefaultNamespace.EntitySelection::PickSimilarEntities(CSimulation2,...	44	8
DefaultNamespace.DefaultClass::ConstructCircleOrClosedArc(CSimConte...	42	10
DefaultNamespace.CCmpObstructionManager::TestLine(IObstructionTestF...	37	7
DefaultNamespace.SimBoundary::ConstructSquareOnGround(CSimContour, f1	21	0



at most 2 parameters    more than 2 parameters    more than 4 parameters    more than 6 parameters

fig. 20 Analyse BetterCodeHub sur le nombre de paramètres des méthodes

Nous pouvons voir un problème sur le nombre de paramètre pouvant atteindre le nombre de 10. Certains cas pourraient être évités, notamment en créant des structures. Comme par exemple dans la fonction suivante où une structure de coordonnées aurait pu être mise en place :

```
static void ConstructCircleOrClosedArc(
    const CSimContext& context, float x, float z, float radius,
    bool isCircle,
    float start, float end,
    S0OverlayLine& overlay, bool floating, float heightOffset)
```

De plus certains paramètres ne sont pas clairs sur leur signification comme par exemple dans la méthode :

```
static void AddTerrainEdges(std::vector<Edge>& edges, std::vector<Vertex>&
    vertexes,
    int i0, int j0, int i1, int j1,
    pass_class_t passClass, const Grid<NavcellData>& grid)
```

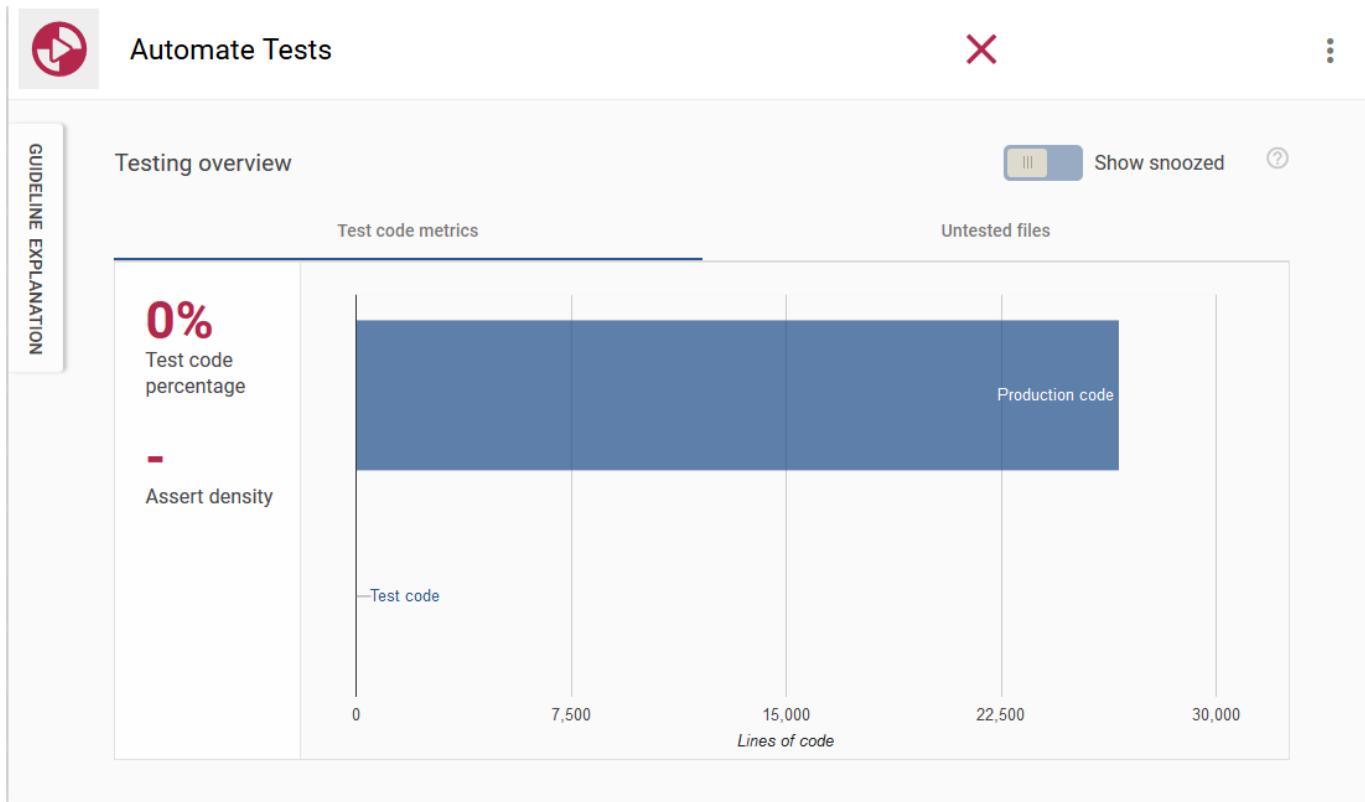


fig. 21 Analyse BetterCodeHub sur l'automatisation des test

Nous pouvons voir que d'après BetterCodeHub il n'y a aucun test automatisé. Ce serait donc un point à améliorer. Cependant il y a quand même un sous-dossier tests dans cette section.

## binaries/data

Binaries/data nous semblait un dossier également intéressant à analyser. Avec CodeScene nous avons pu retirer quelques informations parmi les fichiers proposés:

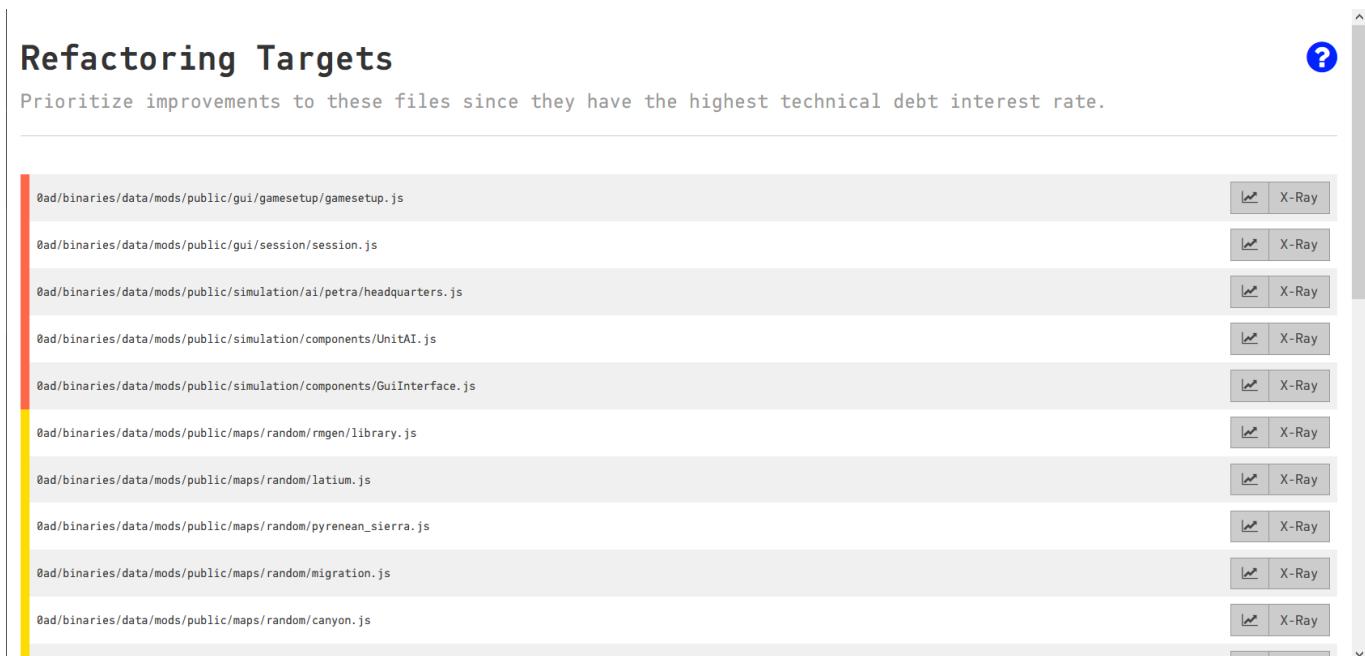


fig. 22 Analyse CodeScene pour trouver les candidats au refactoring

Nous avons regardé le fichier UnitIA.js suite à cela , et nous avons pu noter la grosseur d'un tel

fichier : 6140 lignes. Nous émettons donc l'hypothèse que ce fichier pourrait être réduit en étant séparé en plusieurs modules.

Cependant parmi tous les fichiers Javascript disponibles nous trouvons que la quantité de fichiers proposés en refactoring était moindre.

Gamesetup.js possède également un nombre de ligne assez conséquent : 2331 , qui devrait probablement être remanié.

### Analyse avec BetterCodeHub de binaries/data

Nous avons également analyser binaries/data avec BetterCode et voici ce que nous en avons retiré :

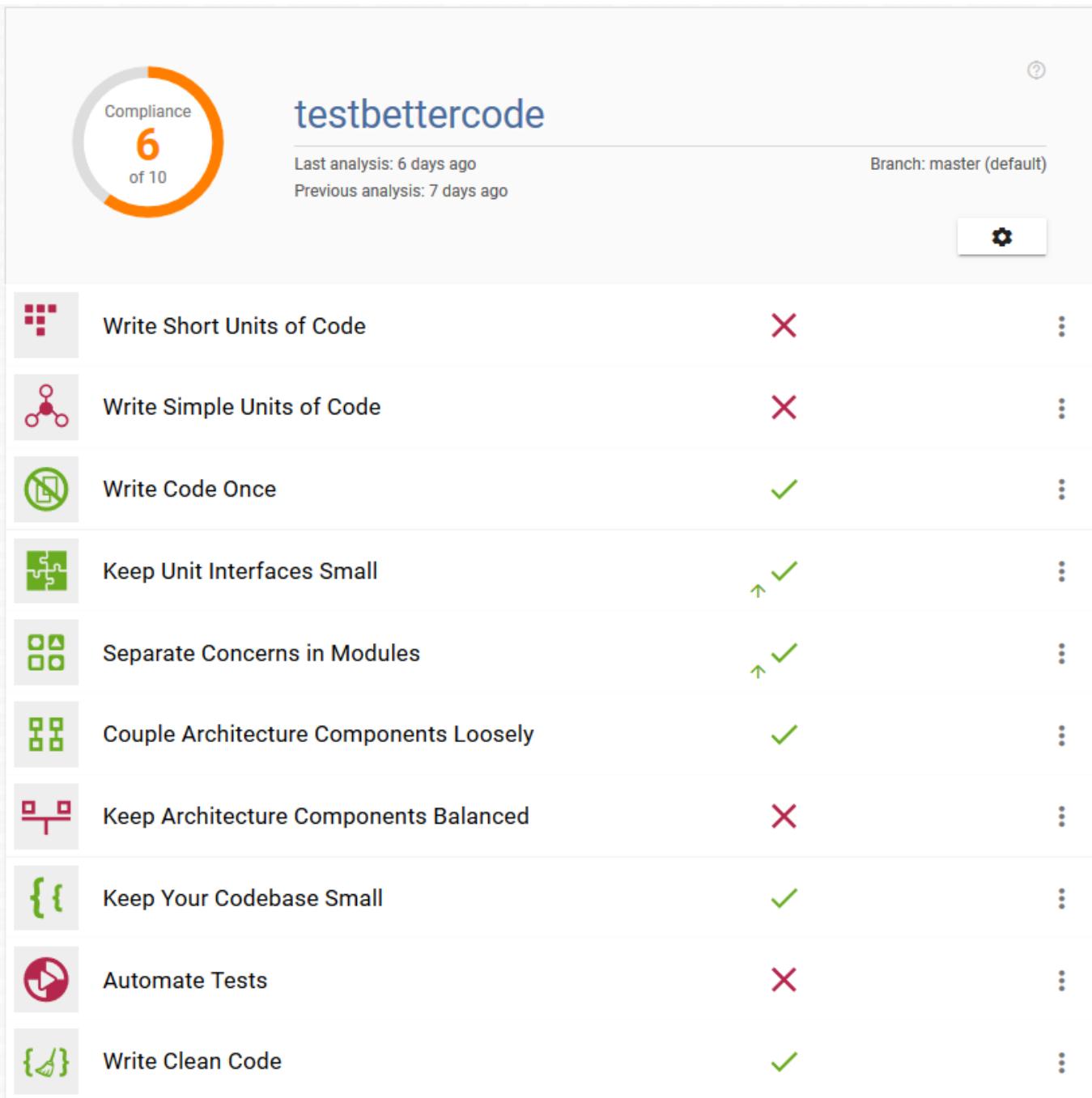


fig. 23 Analyse BetterCodeHub sur le dossier binaries

Nous pouvons voir que sa notation est plus élevée que l'analyse de simulation2 , cependant il y a des

problème récurrents , avec les métriques de qualités de : Write Short Units of Code et Write Simple Units of Code.

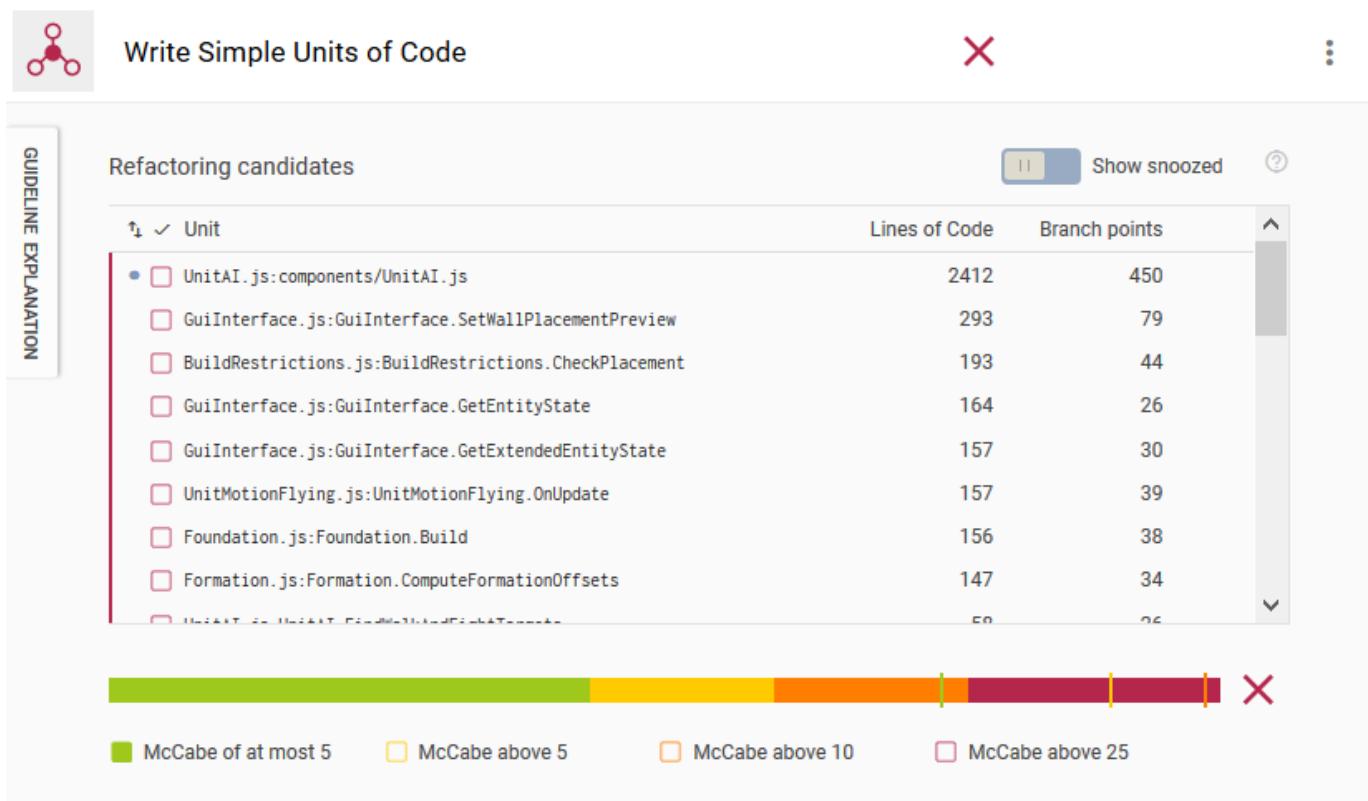


fig. 24 Analyse BetterCodeHub Write Simple Units of Code sur le dossier binaries

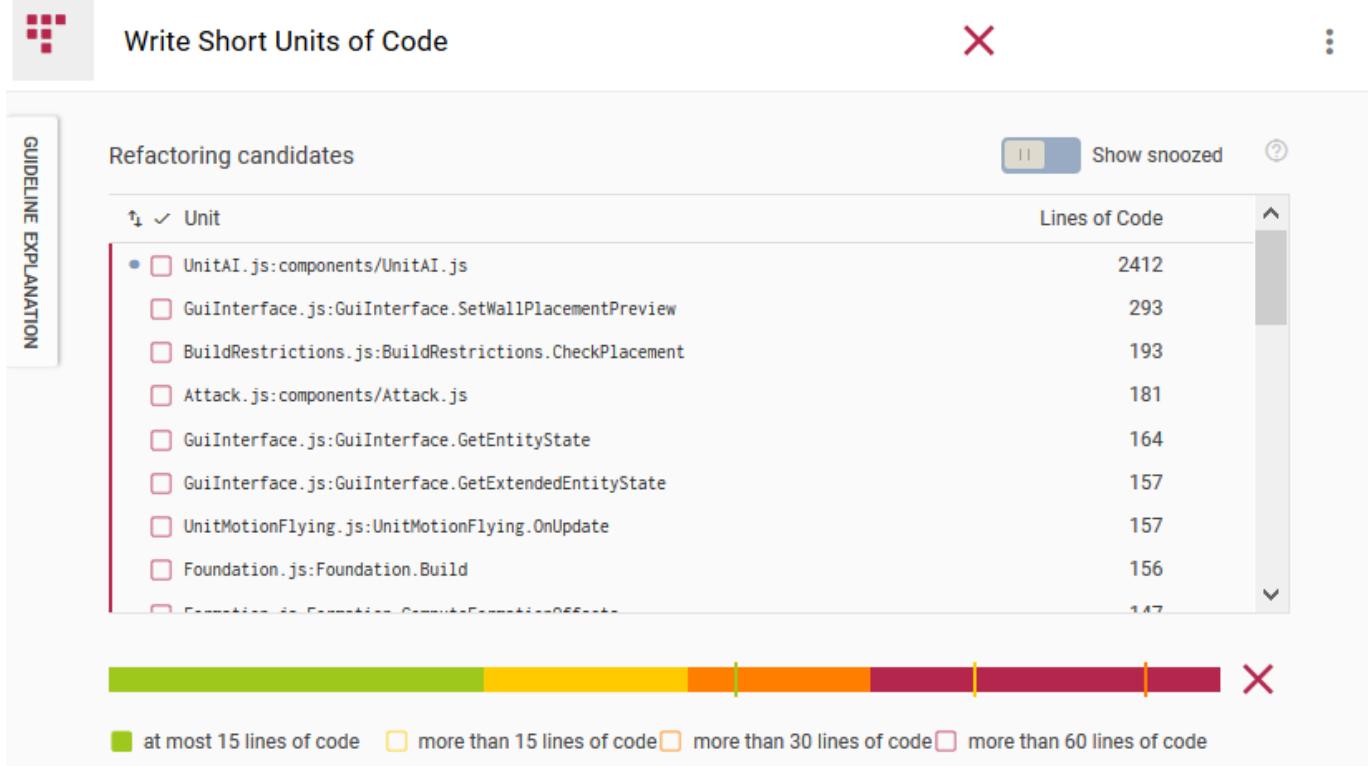


fig. 25 Analyse BetterCodeHub Write Shorts Units of Code sur le dossier binaries

## **Qualité du travail d'équipe**

Malgré quelques problèmes de qualité , nous trouvons que pour un logiciel libre de jeu vidéo nous avons une bonne documentation du projet. Cette documentation explique l'utilité des différents dossiers pour les programmeurs et les artistes. Nous pouvons trouver également un système de tickets répertoriant les différentes tâches à faire ou qui sont en cours de production , ainsi qu'un forum. Nous pouvons trouver aussi des informations sur la raison du choix des langages , des consignes de conventions de codage , la suggestion d'utiliser des objets mocks, un endroit où reporter les bugs. Cependant il est précisé à certains endroits que certaines informations peuvent être datées , notamment en ce qui concerne le design. Le code possède des commentaires clairs pour des sujets importants. Notamment sur l'utilisation de design pattern. Nous avons pu trouver des fichiers singleton avec en commentaires leurs utilité et dans quel cas les utiliser. Ils font également partie des recommandations sur l'utilisation de ce design pattern dans la documentation du logiciel.

## **Design Pattern**

Notamment sur l'utilisation de design pattern. Nous avons pu trouver des fichiers singleton avec en commentaires leurs utilité et dans quel cas les utiliser. Ils font également partie des recommandations sur l'utilisation de ce design pattern dans la documentation du logiciel. Nous avons également trouvé un boost flyweight qui pourrait suggérer l'utilisation du pattern flyweight.

## **Conclusion sur la qualité du code**

Il y a plusieurs points faibles dans les parties que nous avons étudié. Cependant on peut noter une envie d'accomplir ce projet dans de bonnes conditions structurées, mise en évidence par la documentation. Cette documentation a beaucoup de détails sur comment les développeurs doivent travailler par exemple :

## How to write components

See the [Trac wiki](#) for more documentation about this system.

- [Defining interfaces in C++](#)
- [Interface method script wrappers](#)
- [Script type conversions](#)
- [Defining component types in C++](#)
  - [Message handling](#)
  - [Component creation](#)
  - [Component XML schemas](#)
  - [System components](#)
- [Allowing interfaces to be implemented in JS](#)
- [Defining component types in JS](#)
- [Defining interface types in JS](#)
- [Defining a new message type in C++](#)
- [Defining a new message type in JS](#)
- [Component communication](#)
  - [Message passing](#)
  - [Retrieving interfaces](#)
- [Testing components](#)
  - [Testing C++ components](#)
  - [Testing JS components](#)

## Defining interfaces in C++

Think of a name for the component. We'll use "Example" in this example; replace it with your chosen name in all the filenames and code samples below.

(If you copy-and-paste from the examples below, be aware that the [coding conventions](#) require indentation with tabs, not spaces, so make sure you get it right.)

fig. 26 Exemple de la documentation

Cependant les ajouts qui pourraient être faits sont par exemple une section de mise en évidence de design pattern à utiliser et dans quels cas, une section expliquant l'importance de la création de sous-méthode , la gestion de paramètre. Ce genre de projet devrait conseiller l'utilisation de logiciel comme Better Code Hub et CodeScene lors du développement pour éviter les problèmes que nous avons rencontré.

## Conclusion

En conclusion le projet est plutôt bien structuré et beaucoup d'efforts sont fait dans ce sens. Hormis les deux problèmes architecturaux soulignés dans la partie correspondante le projet fonctionne de toute évidence bien. Du côté de la qualité du code source de gros efforts sont fait pour instaurer de bonnes pratiques lors de la création du code cependant on remarque sur les fichiers analysés un manque de synthétisations des données dans les méthodes par exemple. Un manque du côté des tests automatisés d'après BetterCodeHub est également présent. Le projet pourrait donc être améliorer en suivant des pratiques supplémentaires comme l'automatisation des tests, en conseillant l'utilisation de logiciel comme BetterCodeHub pour le développement.