MEHDI Rayan
TELLIER Olivier
MODOUX Josua
CHABOUD Thomas

Portage de A.M.E.S sur Android

Rapport de projet tuteuré

“ Bon est-ce que vous avez déjà vu fantôme? Vous pouvez m’affirmer que ça clignote pas? ¯_(ツ)_/¯ ” - David goodenough

Sommaire

1. Objectif à atteindre

1.1. Présentation de A.M.E.S

A.M.E.S (Amplificateur Mediumnique ExtraSensoriel) est une application développée par Jean-Philippe Farrugia, Pa Ming Chiu et Jérôme Farrugia sous IOS et disponible gratuitement sur l'App Store.

Jean-Philippe Farrugia

Pa Ming Chiu

Jérôme Farrugia

Elle se présente sous la forme d'un jeu qui invite l'utilisateur à tester ses capacités médiumniques avec l'aide d'une technicienne nommée Diane qui nous accompagnera tout au long de l'aventure. Le but de ce jeu est de détecter et chasser des fantômes se trouvant tout autour de nous.

A.M.E.S à été développée en Objective-C et fonctionne grâce à des fichiers Plist (format de données propre à Apple) contenant plusieurs séquence, chacun composées de différents évènements (affichage d'une image, d'un bouton, déclenchement de la caméra, du son,...). Le principe de l'application est de lire ces fichiers Plist pour effectuer le déroulement de l'histoire.

1.2. Description du projet

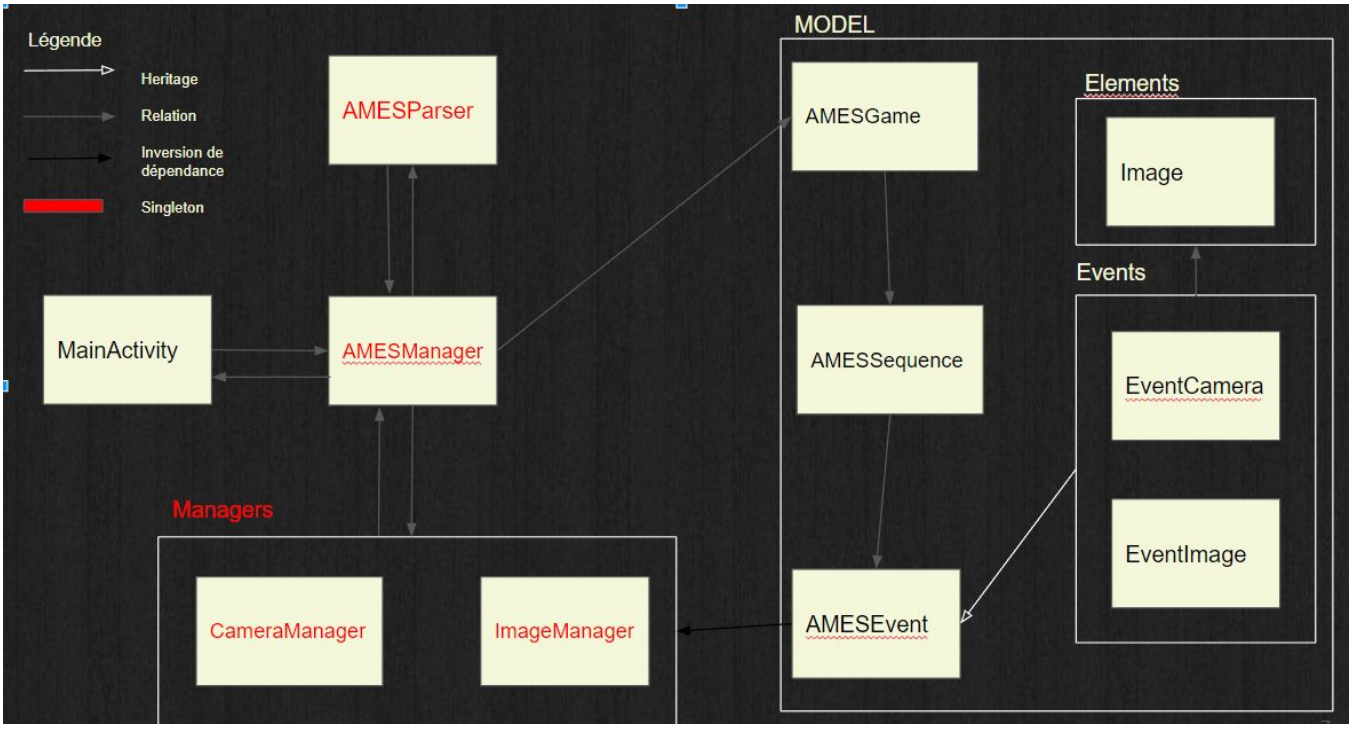
Le projet qui nous a été confié par Jean-Philippe Farrugia est de faire un portage de A.M.E.S sur Android afin de pouvoir par la suite mettre l'application sur le Play Store et la rendre disponible aux utilisateurs cette OS.

Pour cela, il nous fallait déterminer sur quelle plateforme nous allions la développer. Nous pouvions partir sur un developpement hybride ou sur du natif. Un groupe avait déjà travaillé sur un portage Android de cette application et avait choisi de le faire avec Unity (développement hybride). D'après les retours que nous avons eu, ce groupe avaient rencontré des difficultés pour gérer certains éléments (notamment la torche du téléphone). Suite à cela et comme la plupart

d'entre nous n'avait pas les connaissances necessaires en Unity, nous avons choisi de développer le nouveau A.M.E.S en natif.

2. Travail réalisé et difficultées rencontrées

Voici le diagramme de classe de la version d'A.M.E.S réalisé sur Android Studio, qui vous sera, dans un premier temps, détaillé pour comprendre l'architecture que nous avons appliqué à partir des sources fournies par M. Farrugia et du fonctionnement spécifique d'Android pour avoir une structure la plus générique possible, permettant de lire les séquences du jeu déjà existante ainsi que les futurs créations de l'équipe originel :



Les besoins structurel de ce projet étant très spécifique, avec notamment la présence d'une seule vue, constamment mis à jour par des objets graphiques créés dynamiquement, nous avons mis en place un pattern s'apparentant à Pattern MVC (Model - Vue - Contrôleur) revisité permettant d'utiliser au mieux ces spécificités par notre propre équipe ou par M. Farrugia.

Résumé : Comme décrit dans la première partie de ce document, notre application n'est en fait qu'un interpréteur de scénario où chaque avancées (appelé Event) permet d'afficher à l'écran une action prédéfinie, comme du texte ou une image.
Au lancement de l'application, notre classe AMESApplication instancie la classe qui est la plaque tournante de tout le jeu : AMESManager (Singleton qui va mettre en place toutes les classes indispensable au bon déroulement du jeu), il crée tout d'abord un objet "AMESGame" dont la fonction est de gérer le déroulement du jeu en stockant les Séquences contenant chacune les Event lu par le Parser via les fichiers XML, puis crée tous les Managers, singleton eux aussi, qui sont nos contrôleurs, appliquant le cycle de vie d'un Event spécifique : son lancement et son arrêt.

Nous fonctionnons avec l'unique Activity "MainActivity", et son layout, qui nous sert de page principal du jeu ayant accès à notre AMESManager, où l'appel vers le Parser est effectué, créant les différents Event et lançant le jeu une fois tous les Event créés.

Vue / Parser :

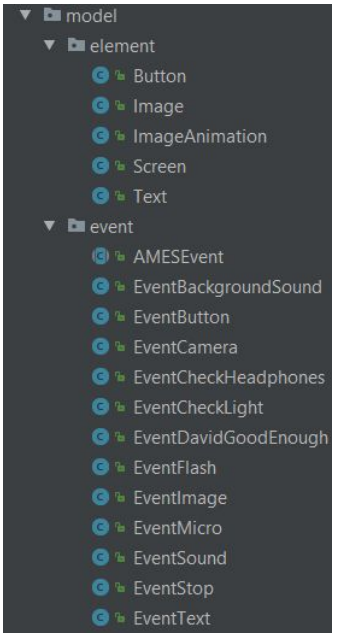
Comme décrit ci-dessus, la MainActivity n'a pour seul intérêt que de faire office de vitrine, un layout vide où chaque élément graphique seront apporté et retirer dynamiquement par les Event et de lancer la création de ces Event via le Parser en lisant les fichiers XML sous format PList qui y sont inscrit. Ces Event sont créés à partir du mot clef TYPE de chaque Objet du fichier XML où un Switch case récupère les clefs présente sur cet objet pour instancier un Event qui sera directement ajouter dans une séquence (une séquence par fichier lu).

```
switch (getValueInString(pListEvent,TYPE)){
    case "animated text":
        if(pListEventParameter.has(DISPLAY_OFF)){
            event = new EventStop(amesEventName, amesEventType, amesEventDelay);
        }else {

            Text text = new Text(getValueInString(pListEventParameter,DISPLAYED_TEXT),
                    Double.parseDouble(getValueInString(pListEventParameter, X_LOCATION)),
                    Double.parseDouble(getValueInString(pListEventParameter, Y_LOCATION)),
                    Double.parseDouble(getValueInString(pListEventParameter, WIDTH)),
                    Double.parseDouble(getValueInString(pListEventParameter, HEIGHT)),
                     isAnimated: true,
                    Double.parseDouble(getValueInString(pListEventParameter, TEXT_SPEED)));
            event = new EventText(amesEventName,amesEventType,amesEventDelay, text);
        }
        break;
```

Ci dessus un exemple où le mot clé "animated text" permet de créer un EventText, gérant les textes simples ainsi que ceux apparaissant avec du delay entre les caractères.

Model :

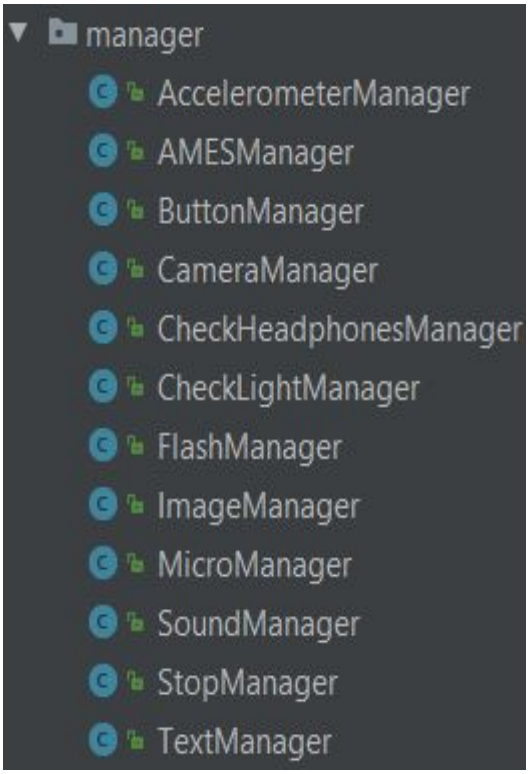


Nos modèles sont donc ces Event héritant tous de notre classe générique AMESEvent et possédant comme paramètres : le lien vers la ressource (ou élément) associé , un nom pour les EventStop (voir plus bas), ainsi que d'un délais avant le lancement du prochain Event de la Séquence.
Chaque Event possèdent aussi deux fonctions d'appel à son Manager : run et stop qui sont appelés lorsque l'Event est lancé ou doit être détruit, tout ce travail est réalisé par les Managers.

Parmi nos Event, nous en avons un particulier : l'EventStop, celui-ci ne possèdent pas d'autre paramètre que le nom et le type hérité d'AMESEvent, celui-ci est créer lorsque, dans le fichier XML, il est demandé à un Event de s'arrêter à un moment précis du déroulement de la Séquence, possédant le

même nom et type que l'Event originel, il réutilise ces informations pour aller appeler la fonction stop de l'Event souhaiter.

Contrôleur / Managers :



Il existe deux types de Managers : L'AMESManager qui est le singleton instanciant tous les autres et par lequel il faut passer pour accéder à ceux-ci ainsi qu'au Singleton Game et les Manager d'Event qui existe pour effectuer la logique métier des Event et sont appelé directement par ceux-ci.

Un manager d'Event possèdent deux rôle essentiel : Celui de lancer l'événement, créant un objet View (comme une TextView pour un texte) qui viendra s'ajouter au layout puis, ayant initialisé et lancer l'Event, il effectueras généralement un CountTimer au bout duquel il demandera à la Game de lancer le prochain Event de la Séquence en cours qui se fait de manière Asynchrone, le delay de ce timer est le delay de l'Event sauf cas particulier comme un bouton ou un texte bloquant avec un delay de 0, demandant une action de l'utilisateur avant de s'autodétruire et appelant l'Event suivant.

Son autre tâche est donc de détruire la View qu'il avait mis en place au moment où il le faut, dépendant de l'Event certains utilisant un EventStop, d'autres se détruisant seul comme les boutons.

Difficultées rencontrées

Lors de ce projet, nous avons dû affronter différents problèmes, certains liés à la contrainte matériel d'Android, d'autre lié à l'organisation, étant notre première expérience à tous dans la tâche de portage de code d'un support à un autre, voici une liste résumant certains d'entre eux :

Parser :
C'est ici qu'est née notre première grande difficulté : trouver un Parser pour lire les données, étant gérée nativement sur le support Apple. Notre premier réflexe fut d'en créer un de nous même mais certaines balise posait problème, notamment celle de Texte, notre Parser n'arrivant pas à lire un String sur plusieurs lignes ou avec des saut de lignes mais notre méthode de travail, qui était de traiter le Parser et le fonctionnement des Event en les prenant un à un dans leur ordre d'arrivé de la première séquence, a fait que nous nous sommes rendu compte assez tardivement de ce problème, il a été résolue en changeant entièrement le Parser, utilisant une librairie décrite dans la partie planning.

Camera :
La caméra fut un gros morceau avec plusieurs type de problèmes. Tout d'abord nous sommes partie sur l'utilisation de Camera 2, étant plus récente pour avoir le moins de soucis avec tous les type de SmarthPhone Android, mais nous n'avons pas réussis à correctement afficher la Preview de la Caméra, faute de documentations sur le sujet sur les divers sites parlant du sujet.
Ensuite fut pris le choix de revenir sur Camera 1 avec laquelle nous avons réussit à afficher la preview de la caméra en cours grâce à une textureView. Mais, quelques temps plus tard, une autre difficultés apparut, le fait de superposer les layout pour afficher correctement les images au dessus de la caméra, hors avec une textureView, il nous était impossible de l'ajouter aux view Classique de notre layout. Nous avons donc dû faire des recherches pour finalement recourir à une surfaceView, avec laquelle nous avons réussit à faire fonctionner tout ce qu'il fallait.

EventStop :
Pour qu'un Event s'arrête à un moment précis choisi par le déroulement de la Séquence et non un timer classique, le problème de gérer son arrêt nous a été posé et nous devions trouver un moyen efficace et restant le plus générique possible pour permettre l'ajout de nouveau contenue sans trop de difficultées. Après une discussion avec tous les membres de l'équipe notre choix se porta sur la création d'un EventStop décrit dans la partie model.

GIF :
Dans le code IOS, les GIF sont en fait l'affichage de toutes les images souhaiter avec un délais précis entre chaque image mais nous avons voulu essayer de les gérer plus simplement avec une librairie Android permettant d'afficher un GIF créer par nos soins, cela aurait allégé les ressources et de la mémoire. Nous avions réussis avant de comprendre en revoyant l'application tourner sur IOS, qu'à la fin du déroulement d'un GIF, la dernière image de celui-ci devait rester comme image fixe le temps qu'un EventStop vienne la détruire. Hors, cela était impossible avec notre librairie, nous avons donc changer en reprenant une façon de faire plus proche du code original.

Scale :
M.Farrugia ayant utilisé, lors de la période de développement, des IPhone ayant des résolution proche, le fichier XML ne contient pas de notion de scaling de la taille de l'image par rapport à l'écran. Après avoir compris ce soucis, nous sommes aller en parler directement avec lui, qui nous a dit accepter le fait que nous rajoutions des paramètres dans le fichier XML, ne changeant rien du coté IOS car ces informations ne serait simplement pas traiter dans son Parser.

Mode d’emploi pour de futurs ajouts

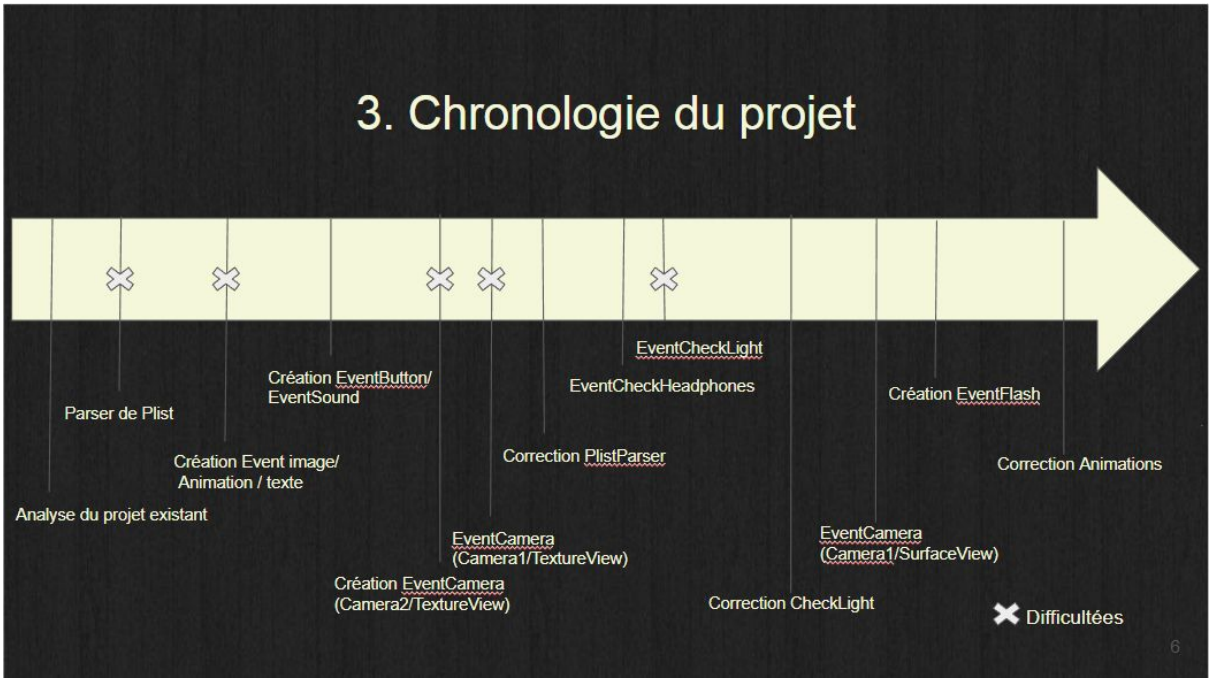
Notre projet étant le plus générique possible, même si certaines facilités ayant été nécéssaire, déjà présente dans le code IOS car le fichier XML ne nous permettant pas de le traiter autrement, l’ajout de nouvelles séquences ou Event est en effet très simple à faire.

-Pour ajouter une séquence ou modifier leur ordre d’apparition, vous devez, dans la Main Activity modifier la fonction loadSequenceFile en ajoutant votre fichier qui devra se trouver dans le dossier ressource RAW

-Pour ajouter de nouveaux Event : Ajouter dans le Switch Case de notre Parser le mot clé associé et créer un Event ainsi qu’un Manager en utilisant ceux déjà présent comme exemple : le Modèle hérite d’AMESEvent et Override les fonction Run et Stop, le Manager ayant une fonction pour initialiser la view, l’ajoutant au layout principal et démarrant le prochain Event après un Timer ou un listener précis, et une fonction pour détruire une View précédemment créée.

3. Organisation et Planning

En ce qui concerne l'organisation du projet ainsi que la répartition des rôles, nous nous sommes servis de Trello afin de structurer et de découper le projet de manière à être le plus efficace possible.



Les deux étapes principales du projet peuvent être résumé ainsi: créer (ou récupérer) un parser de plist qui était le coeur de l'application, puis gérer les différents événements existant pour les incorporer un à un au projet.

Au niveau de la répartition des tâches sur les deux semaines, nous avons tout d'abord accordé 3 jours à la compréhension du code ainsi qu'à la réalisation de la structure générale du projet. Ensuite, la durée des différentes tâches a varié en fonction de la difficultée de celles ci: les evenements simple tel que le CheckHeadphones ont mis un peu moins d'une demi journée a être réalisé, tandis que d'autres tel que la camera ou encore le parser qui nécessitait des des modifications constantes ont été réalisé en plusieurs jours (2 à 3 jours au total).

Pour le parser de plist, c'est Josua qui s'en est chargé dans un premier temps en essayant de créer un parser personnalisé, puis dans un second temps en incorporant une librarie trouvée sur github (https://github.com/keiji/plist-parser-java) afin de résoudre les problème de la première version du parser.

Pour les différents événements, nous travaillions en général par binômes lorsque des difficultées se faisaient ressentir afin de régler les problèmes aussi vite que possible.

La répartition c'est fait ainsi : les événements liées aux images(Images et animation) aux textes(textes et animated texte) ainsi que la gestion du son et du micro ont été géré par Thomas et Olivier. Les événements camera et boutons ont été géré par Josua et Rayan. les evenements utilisant le capteur de luminosité et la reconnaissance des écouteurs ont été géré par Olivier et Rayan.

Enfin, l'harmonisation des différentes fonctions d'affichage ont été réalisé par Josua et Olivier.

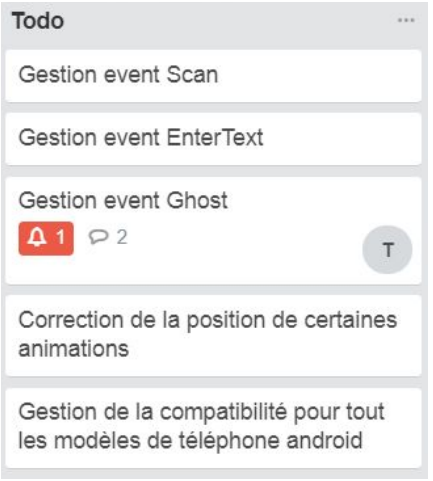
En parallèle de tout cela, nous avions confié à Thomas la tâche de la création de l'événement Ghost qui est le coeur du jeu. Celui ci a pu commencer à explorer certaines fonctionnalités tel que le gyroscope du téléphone pour percevoir les mouvement de l'utilisateur. Cependant, comme l'événement Ghost nécessitait l'utilisation de la majorité des autres événements (Animations, son, image, camera…) et que nous avons eu de grosses difficultées pour en traités certains (cf. *3.Travail effectué et difficultés rencontrées*), nous avons décidé de nous recentrer sur les événements que nous pensions réalisable dans le temps imparti.

4. Bilan et rétrospective

L'application peut actuellement effectuer les 2 premières séquences (sur les 4), c'est à dire toute la partie d'introduction du jeu sans l'apparitions des fantômes. Le projet comporte donc le parser de Plist, qui nous permet de récupérer toutes les séquences et leurs événements et de les rentrer dans notre modèle. Une architecture pensée pour être la plus générique possible, avec un modèle pour chaque type d'événement, et aussi pour être la plus organisée et la plus découpée : avec un manager qui gère tout les autres managers tel que la caméra ou encore l'affichage d'image.
Nous avons aussi un début de recherche sur l'utilisation de l'accéléromètre pour l'événement ghost (sur la branche Feature/EventGhost).

Pour les dernières tâches à effectuer, nous pouvons les retrouver sur notre Trello qui sont :

- L'événement ghost : qui est l'événement utilisant le plus de fonctionnalité et le plus complexe à mettre en place dans l'application : accéléromètre, animations, sons, tolérance : pour la zone ou on considère que l'utilisateur a bien touché la cible).
- L'événement scan : qui est un bouton servant au moment de la chasse aux fantômes.
- L'événement input Text : qui sert seulement à la toute fin pour entrer l'email de l'utilisateur.



Si nous devrions refaire ce projet, nous aurions chercher un bon parser de Plist dès le début, ou encore utiliser directement l'API camera au lieu d'essayer d'utiliser la version 2 qui est beaucoup moins bien documentée et plus compliquée à mettre en place. Ces éléments de l'application nous ont fait perdre beaucoup de temps, qui nous aurait pu nous servir à faire l'événement ghost.
Nous aurions dû aussi demander dès le début la possibilitée d'ajouter des éléments dans les fichiers PList à M. Farrugia, cela nous aurait facilité l'affichage des différents éléments graphique.

Pour conclure, ce que nous avons retenu de ce projet c'est de bien mettre en place un cahier des charges que ce soit au niveau des contraintes (on ne devait pas modifier les fichiers PList), ce que nous avons bien fait. Mais aussi des possibilités autorisés (ajouter des champs au fichiers PList) ce que nous avons demandés mais seulement à la moitié du projet. A l'avenir nous serons aussi plus prudent au moment de la mise en place de nos tests qui des fois nous ont fait perdre du temps. Car des fois nous testions une fonctionnalité en croyant qu'elle n'était pas bonne alors que c'était juste que certains tests était trop particulier. Malgré les difficultés rencontrées dans le projets notre architecture était bien pensée dès le début, elle nous a permit de bien structuré et de partager notre code. Ainsi que notre gestion du dépôt Git dans le groupe (avec les branches et les Tags), et de notre planning sur Trello : ce qui nous a permit d'éviter un maximum d'avoir des conflits entre les différentes version du code, et de bien distribuer les tâches au sein du groupe.