Le groupe étant composé de 10 étudiants (Alexandre A., Alexandre B., Ismail, Raphaël, Philippe, Lina, Brenno, Wassim, Romain, Nicolas) de l'École Polytechnique de Bruxelles. La vélocité du groupe est de 28h par itération.

L'objectif de ce document est de décrire le suivi de planification du projet. Il reprend notamment les explications des fonctionnalités implémentées et les méthodes utilisées pour contourner les différentes difficultés techniques. Il reprend également les différents choix pris lors de la conception du projet.

Gestion de projet

Afin de s'assurer que tout soit bien réalisé à la fin de l'itération, il a été décidé d'utiliser un Microsoft To-Do pour lister et s'assurer que les tâches et documents avaient bien été complétés.

Suivi de planification

Itération 1

Lors de la première itération, certaines fonctionnalités n'ont pas su être implémentées, principalement au niveau de l'affichage de l'application. Cela est dû à une mauvaise répartition des tâches et à un apprentissage qui s'est avéré plus complexe que prévu du JavaFX.

En effet, il y a eu beaucoup de problème au niveau de la structure même du projet qui ont été réglés grâce à l'utilisation de Maven. L'outil Maven a permis de faciliter la construction, la gestion des dépendances et le déploiement de l'application.

Afin de s'assurer que toutes les tâches soient réalisées lors de l'itération suivante, une réunion a été faite afin de s'assurer que l'équipe communique bien ses avancées et que les tâches soient définies de façon claire pour tous les membres de l'équipe.

Afin de faciliter et de gagner du temps lors de la programmation, le module Lombok a été implémenté (voir Lombok.md).

Itération 2

À partir de l'itération 2, après en avoir parlé en réunion, il a été décidé de définir 2 personnes par itération qui définissent les tâches et les écrivent dans le document. De la sorte, toutes les tâches sont clairement définies et il est facilement possible de communiquer entre binômes.

L'affichage des cartes étant une tâche principale, plusieurs autres tâches ont dû être dépendante de celle-ci, ralentissant le travail.

Il faudra dès lors assurer une meilleure communication et création des tâches.

L'histoire 5 n'a pas pu être terminée, les différents types de cartes sont créés, stockés et l'affichage fonctionne. Mais il n'est pas possible d'éditer et d'afficher les différents types de cartes sur l'application. Il a fallu trop de temps pour créer l'édition du paquet et l'ajout de cartes dans ce dernier. Dès lors il a été décidé de termine l'histoire 5 à l'itération 3 avec 3 points supplémentaires. C'est notamment pour cela que certaines fonctions ne sont pas utilisées dans l'affichage, elle seront donc utilisées dès le début de l'itération 3.

Itération 3

Dans l'ensemble, toutes les tâches ont été réalisées.

L'histoire 8 a été terminée et implémentée.

L'histoire 10 a été implémentée mais il en résulte pas mal d'erreurs qui n'avaient pas été prévues.

L'histoire 11 a été commencée, mais pas mal d'éléments ont été sous-estimés. Il ne reste plus qu'à faire le download depuis le store et de vérifier qu'il n'y ait pas de doublons dans le store.

L'histoire 14 a été implémentée et fonctionne totalement.

Plusieurs erreurs imprévues ont fait leurs apparitions au cours de la création du serveur et du store. Il a donc été décidé de créer une histoire de 6 points permettant de résoudre toutes ces erreurs et incompatibilités non traitées.

Lors du *code review* de l'itération 2, nous avons remarqué que nous appelions le modèle dans la vue à plusieurs reprises. Cela a été enlevé à certains endroits, mais dans l'affichage de la liste des paquets. Cela aurait pris trop de temps en plus des tâches définies.

Il a donc été décidé de prendre des points de refactoring et de traiter cela à l'itération 4.

Itération 4

Histoire 15 – Import et Export des cartes

L'export et l'import des cartes se sont déroulés légèrement plus rapidement que prévu. Ce qui a permis de libérer un peu plus de temps sur le refactoring.

Histoire 17 – Cartes HTML et LaTeX

L'histoire a été séparée en 3 tâches : l'affichage du HTML, l'affichage du latex et la modification de l'édition. Les 3 tâches ont été réalisées sans problèmes majeurs.

Histoire 25 – Terminer le server et le store

Cette histoire a été répartie en plusieurs tâches qui ont chacune été réalisées dans les temps.

Histoire 26 – Refactoring

Il a été décidé de mettre la priorité sur la propreté du MVC et de la transition du stockage de données.

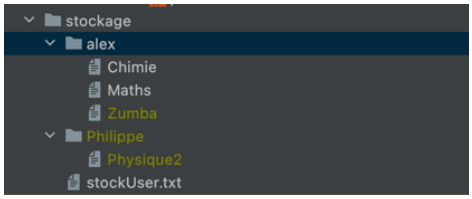
Histoire 27 – Optimisation et debug du code

L'histoire 27 a été rajoutée par suite de la modification de la deadline de l'itération 4. Il a été convenu avec le client de rajouter 14 points pour l'optimisation du code et sa propreté. Dès lors différentes tâches ont été créées afin de rendre le code plus propre et optimal.

Choix de conception

Itération 1

L'utilisation des fichiers pour le stockage de données a été implémenté pour faciliter l'accès aux données. Les utilisateurs sont stockés dans "stockUser.txt" dans le dossier ''stockage''. Chaque utilisateur a un dossier personnel avec à l'intérieur une série de fichiers texte correspondant aux paquets.



Les couples utilisateurs / mot de passe sont structurés comme suit "utilisateur#motdepasse". C'est une solution d'encodage qui est très basique, facile à utiliser dans le code et qui n'a pas de contraintes sur la sécurité pour l'instant.

“stockUser.txt“

```
alex#pomme
philippe#orange
```

Les fichiers “paquets“ sont structurés comme suit : la première ligne correspond au nom du paquet et les lignes d’après sont les questions et réponses, séparés par des “#“.

“Physique2“

```
Physique2
#Qu'est ce que la divergence du rotationnel ?#Une bien belle chose
#PV ?#=NRT bien sûr !
```

La structure MVC n’a pas été implémentée et n’a pas été prise en compte lors du développement du code de l’itération 1. Par conséquent, des parties du code se sont mélangées ne suivant pas les principes du MVC. Ce qui rendait l’utilisation et l’assemblage des différentes parties du programme très complexe et non optimal.

Itération 2

Lors de l’itération 2, il a été décidé d’implémenter la structure MVC recommandée lors du cours théorique. Elle facilite grandement l’implémentation de nouvelles fenêtres et la lecture du code.

Pour l’application de l’histoire 8 (cartes textes à trous et QCM), il a été décidé dans un premier temps de ne pas faire d’héritage au niveau des cartes car cela allait compliquer fortement le stockage de ces dernières. La solution adoptée a été d’écrire dans les String *recto* et *verso* les questions et réponses sous des formats différents selon le type de la carte. Des fonctions permettant de décrypter ces formats ont été implémentées afin de faciliter l’affichage et l’utilisation de ces String *recto* et *verso.*

La gestion des exceptions n’étant pas implémentée de manière correcte, elle a dû être remaniée.

Il a été décidé que chaque paquet de cartes soit stockés suivant la structure suivante :

“Nom du paquet“

```
Nom du paquet
#Categorie1#Categorie2#...#CategorieN
#TypeDeCarte#Recto#Verso#Score
#TypeDeCarte#Recto#Verso#Score
#TypeDeCarte#Recto#Verso#Score
#TypeDeCarte#Recto#Verso#Score
```

Pour l’application de l’histoire 8 (cartes textes à trous et QCM), il a été décidé dans un premier temps de ne pas faire d’héritage au niveau des cartes car cela allait compliquer fortement le stockage de ces dernières. La solution adoptée a été d’écrire dans les String *recto* et *verso* les questions et réponses sous des formats différents selon le type de la carte. Des fonctions permettant de décrypter ces formats ont été implémentées afin de faciliter l’affichage et l’utilisation de ces String *recto* et *verso.*

Pour le format texte à trou (*tt)* : `#tt#débutdephrase§findephrase#réponse#Score`

Pour le format qcm (*qcm)* : `#qcm#question§choix1§choix2§choix3#réponse#Score`

Itération 3

Types de cartes

Afin de faciliter l’utilisation des différents types de cartes, il a été décidé d’utiliser de l’héritage. Au niveau du stockage, cela ne change pas. Mais chaque carte a des attributs différents.

La carte TT (Texte à trou) possède les attributs (String) : *begin*, *gap* et *end*.

La carte QCM (Question Choix Multiple) possède les attributs (String) : *question*, *rep1, rep2 , rep3* et *answer*.

La carte Simple possède les attributs (String) : *recto* et *verso*.

Pour pouvoir éditer les paquets de cartes et gérer l'affichage lors du jeu avec différents types de cartes. Il a été décidé d'utiliser la propriété « *visible* » des éléments fxml. C'est un choix qui a été fait après de nombreuses heures passées à essayer d'implémenter des fichiers fxml dans d'autres fichiers fxml.

Serveur avec Spring Boot

Pour les histoires 10 et 11, nous avons choisi d'utiliser Spring Boot en raison de sa facilité d'utilisation et de création. Nous avons opté pour une architecture composée de plusieurs couches distinctes, chacune remplissant un rôle spécifique.

Tout d'abord, nous avons la couche "Model", qui représente la forme de données manipulées et suit le modèle MVC classique. Ensuite, nous avons les "Data Access Objects" (DAO), qui permettent d'interagir avec la base de données. La couche "Service" est intermédiaire entre les DAO et les API, elle se charge du traitement des données et d'effectuer des opérations spécifiques sur celles-ci.

Enfin, nous avons la couche "API" qui s'occupe de gérer les requêtes HTTP. Dans l'ensemble, cette architecture est bien pensée et permet une gestion claire et efficace des différents composants de l'application. Grâce à l'utilisation de Spring Boot, nous avons pu développer notre application plus rapidement et de manière plus fluide.

En examinant le code du server, on peut constater que les classes (DAO, API, ...) ne sont pas appelées. Cela est dû au fait que Spring Boot se charge de créer les objets de ces classes. Ainsi, les constructeurs de ces classes ne prennent pas d'arguments, ou s'ils en prennent (pour effectuer de l'injection de dépendance), ils ont l'annotation "#Autowired" placée devant pour que Spring Boot sache quels objets y injecter.

Gestion de la base de données du serveur

L'application possède son propre système de stockage permanent pour les paquets et les utilisateurs, comme décrit dans le chapitre de la première itération. Pour mettre en place le serveur, il est nécessaire de séparer la base de données locale de l'utilisateur et la base de données du serveur, qui contient tous les paquets et utilisateurs existants. Au niveau du serveur, les fichiers de stockage sont gérés par des classes de type DataAccessService. Elles sont responsables de charger les données existantes dans les fichiers texte en mémoire et de les conserver. De plus, ces classes sont responsables de sauvegarder l'état actuel de la base de données à chaque modification. Au niveau du client, les anciennes classes GestionnaireUtilisateur et GestionnairePaquet continuent à gérer le stockage local.

Actuellement, seules les cartes de type Recto-Verso peuvent être sauvegardées et chargées en mémoire au niveau du serveur. Alors, dans la prochaine itération, il serait nécessaire de gérer les stockage des autres types de cartes et aussi, synchroniser les deux bases de données.

Pour envoyer les paquets du client vers le serveur, on transforme les paquets en objet JSON avec un objet « objetMapper » et on l'envoie sous forme de String vers le serveur. Au niveau du serveur, cet objet JSON est récupéré et on recrée un objet paquet avec une fonction de parsing. Au niveau de l'application il y a également une fonction de parsing pour la communication dans le sens inverse.

Histoire 14

L'application utilise la librairie FreeTTS pour intégrer une synthèse vocale a notre application. C'est une des synthèses vocales les plus utilisées ce qui facilite la documentation.

Malheureusement, aucune librairie de synthèse vocal française n'a été trouvée pour l'instant.

Itération 4

Histoire 15 – Import et Export de paquets

Les objets FileDialog ont été utilisés afin de se déplacer dans l'explorateur de fichier de l'OS. C'est un objet efficace et simple d'utilisation. Une vérification que le fichier s'importe correctement a lieu afin de s'assurer que l'application ne plante pas.

Histoire 17 – Cartes HTML et LaTeX

Premièrement, nous avons étendu la classe Carte en CarteSpec. Cette classe est de type « spec ». Elle possède un attribut supplémentaire : « lang » qui représente le language utilisé (html ou latex). En réalité c'est simplement une carte Recto/Verso simple qui peut afficher du HTML ou Latex.

En fonction du langage on appelle la bonne fonction d'affichage. Pour afficher du HTML, on instancie le composant javaFX appelé Webview et on charge le contenu écrit en HTML dans un String. Pour le latex, la solution trouvée a été d'utiliser JLatexMath. Cette classe permet de charger un String écrit en Latex et réaliser un rendu sous forme d'image. Cette image est simplement affichée dans une « imageView ».

Histoire 25 – Synchronisation

Afin de synchroniser les données entre le serveur et l'application, nous offrons à l'utilisateur la possibilité de choisir lorsqu'il se connecte en ligne. L'utilisateur peut choisir de télécharger ses paquets stockés sur le serveur ou d'envoyer ses paquets locaux pour qu'ils soient stockés sur le serveur.

Histoire 26 – Refactoring

Étant donné que la communication entre le client et le serveur s'effectue via JSON, il a été déterminé que le formatage que nous avions créé comportait un nombre excessif de règles, ce qui rendait difficile le stockage de tous les types de cartes. Par conséquence, afin d'uniformiser le formatage de données dans le stockage, la décision a été prise d'abandonner l'ancien formatage et d'adopter le JSON. Ainsi, tout type de carte peut être pris en compte et différencier par un simple attribut type et la création et la lecture des JSON est facilitée par la librairie Jackson, qui utilise des annotations dans les classes du model pour les générer et instancier des objets à partir de JSON.

Il a également été remarqué que certains modèles se retrouvaient dans la vue car cette dernière affichait plusieurs éléments de ces modèles. Or, il se trouve que ces éléments requis pour la vue étaient très souvent les mêmes. Afin de respecter l'architecture MVC, il a donc été décidé de créer des DTO comme objets intermédiaires, immutables, stockant les éléments nécessaires pour l'affichage. Ceux-ci sont stockés dans la vue.