TD SAE - Projet Zeldiablo

Ce TP est le début de la `SAE_2.01`. Ce TP a pour objectif de vous fournir un point de départ pour l'application à développer et de vous faire découvrir la démarche suivie dans le cadre du projet. Il est extrêmement important de suivre l'ensemble des consignes afin de disposer de la base nécessaire pour accomplir la `SAE_2.01`.

L'objectif de la `SAE_2.01` sera de développer une application en mettant l'accent sur **la partie conception et la structure** de votre code. Cela impliquera :

- de choisir des fonctionnalités à implémenter ;
- de réfléchir à la manière de modifier votre code existant pour les intégrer ;
- de proposer des diagrammes de classe et de séquence qui expliquent comment faire ;
- de modifier le code pour ajouter ces fonctionnalités ;
- de les tester et de les valider.

La `SAE_2.01` se fera par groupe de 4 et utilisera `git` pour coordonner le travail. Un seul dépôt sera utilisé et partagé par groupe.

Attention :
Comme la `SAE_2.01` est centrée sur la partie conception et IHM, la structure de vos classes ainsi que les diagrammes de classe et de séquence auront une grande importance dans l'évaluation de votre projet.

Première partie

Moteur de Jeu

1 Mise en place du projet

1.1 Structure du projet

Les classes utiles pour la mise en place du projet sont fournies dans un fichier `zip` disponible sur arche.

Question 1
Créez sous GitHub un dépôt `git` nommé « `2022_Zeldiablo_`*`logins`* » où *logins* représente l'ensemble des logins du groupe. Partagez le dépôt entre vous et avec votre enseignant.

Pour le moment, **N'UTILISEZ PAS** `IntelliJ`. La configuration d'`IntelliJ` sera abordée dans une partie spécifique.

Question 2
Récupérez les fichiers fournis sur arche et ajoutez les au dépôt en respectant la structure des fichiers fournis.

À l'issue de cette étape, votre dépôt doit être structuré de la manière suivante :

- un répertoire « `projet_zeldiablo` » destiné à contenir votre projet `IntelliJ` (cf suite) ;
- un répertoire « `documents` » destiné aux documents de conception de chaque version ;
- un répertoire « `documents/version_0` » contenant déjà les diagrammes de classe et de séquence de la version fournie ;
- un répertoire « `documents/version_1` » destiné à contenir vos fonctionnalités, diagrammes de classe et de séquence de la version 1 ;
- un répertoire « `documents/version_2` », ... pour les documents de conception des versions futures.

Question 3
Ajouter à la racine du dépôt un fichier « `README.md` » contenant les noms des

. membres de votre groupe.

Question 4

Ajouter à la racine du dépôt un fichier « `.gitignore` » permettant de filtrer les fichiers du dépôt. Parmi ceux-ci, vous penserez à filtrer :

- les fichiers liés à IntelliJ (en particulier les fichier `.iml` et le répertoire de configuration `.idea`) ;
- les fichiers `.class` ;
- les fichiers liés au système d'exploitation utilisé (ex `Thumbs.db`).

A l'issue de cette étape, votre projet doit être structuré selon la figure **??**.

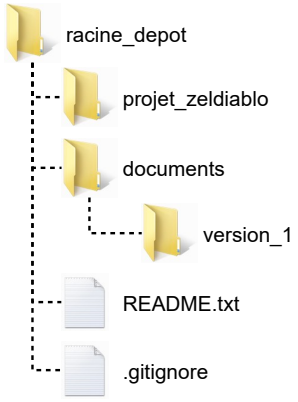

FIGURE 1 – Structuration du dépôt

1.2 Configuration d'`IntelliJ`

On souhaite respecter les contraintes suivantes

- avoir un projet sous `Java 11` pour être compatible avec les machines IUT ;
- avoir les bibliothèques `JavaFX` correctement installée.

Question 5

Une fois que chaque membre du groupe a clone le dépôt, suivez les étapes ci-dessous une par une pour construire un projet valide.

1. Créer un projet `IntelliJ` « `Java 11` » dans le répertoire « `zeldiablo` » contenu dans votre dépot.
2. Pour ajouter des bibliothèques, aller dans le menu `File -> Project structure...`

3. Sélectionner `Librairies -> '+' -> From Maven...`
4. Chercher « `org.openjfx:javafx-base:11.0.1` » et ajouter la bilbiothèque.
5. Chercher « `org.openjfx:javafx-controls:11.0.1` » et ajouter la bilbiothèque.
6. Valider ces bibliothèques.

Question 6
Vérifier que la classe `gameArkanoid.MainArkanoidRapide` compile et se lance bien (les touches pour déplacer la raquette sont `'Q'` et `'D'`).

2 Analyse de la classe `Labyrinthe` fournie

Dans les fichiers fournis, vous trouverez des classes `Labyrinthe` et `Perso` qui correspondent à une version très simplifiée du projet du module « `Qualité de développement` » [1]. Ces classes constitueront le point de départ du projet.

Attention :
ATTENTION : contrairement au projet, l'axe `x` correspond à l'axe des abscisses et l'axe `y` correspond à l'axe des ordonnées (orienté vers le bas).

Question 7
Dans le répertoire « `documents/version_1` », faites un diagramme de classe qui décrit les relations entre ces classes.

Question 8
Dans le répertoire « `documents/version_1` », faites un diagramme de séquence qui illustre ce qui se passe lorsque l'on demande de déplacer le personnage dans la direction « `Gauche` ».

Question 9
Ajouter et envoyer les diagrammes dans votre dépôt distant.

1. et N'EN constituent PAS un corrigé.

3 Moteur de Jeu

L'objectif de cette partie est d'intégrer un moteur de jeu graphique dans votre projet. Des classes basées sur `JavaFX` vous sont **fournies** dans les fichiers téléchargés (plus précisément dans « `zeldiablo/src/moteurJeu` »), le travail sur cette partie va consister à vous interfacer avec ces classes.

3.1 Objectif d'un moteur de jeu

Un **moteur de jeu** est en charge d'organiser l'exécution du jeu. Globalement, un moteur de jeu a pour objectif

- de lancer le jeu ;
- de gérer l'interaction avec l'utilisateur ;
- de gérer l'évolution du jeu au cours du temps ;
- de gérer l'affichage du jeu ;
- d'attendre un court temps pour assurer un framerate constant.

3.2 Description des classes du moteur

Le moteur fourni est composé de plusieurs classes décrites dans le diagramme de classe présenté en figure **??**. Les classes présentées se trouvent dans le package `moteurJeu` et permettent l'exécution et l'affichage d'un `Jeu`.

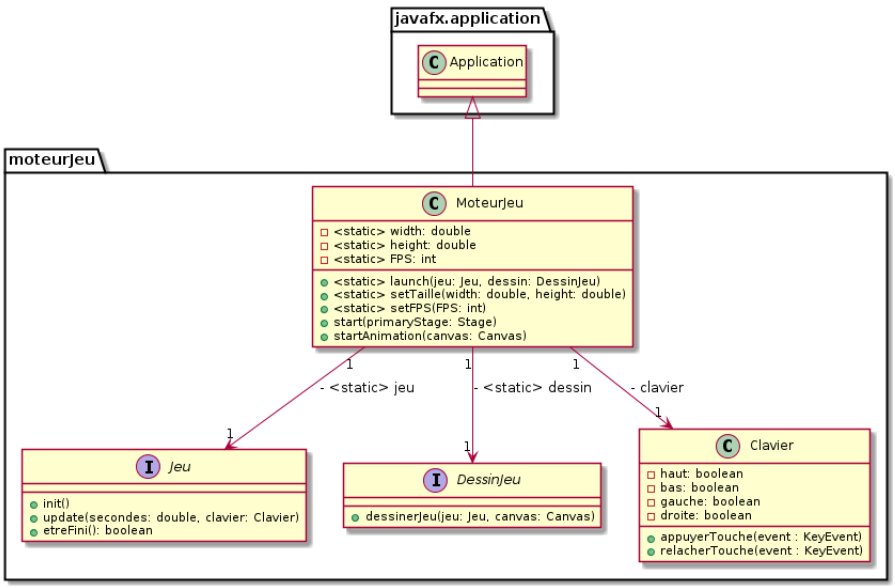

FIGURE 2 – Diagramme de classe décrivant les classes du package `moteurJeu`

- le point d'entrée pour exécuter un jeu est la classe `MoteurJeu` qui hérite de `javafx.application.Application` et possède trois attributs
  - ☐ un objet `jeu` ;
  - ☐ un objet `dessinJeu` ; et
  - ☐ un objet `clavier`.
- l'objet `jeu` est en charge de stocker les données du jeu et de gérer leur évolution. C'est une interface qu'il faut implémenter lorsqu'on souhaite faire un nouveau jeu. Un jeu est caractérisé par la méthode `void update(double secondes, Clavier clavier)` qui fait évoluer les données du jeu en fonction de la commande utilisateur stockée dans l'objet `clavier`.
- l'objet `dessinJeu` est un objet en charge de dessiner le jeu. C'est une interface dont il faut définir la méthode `dessinerJeu(Jeu jeu, Canvas canvas)` pour l'adapter à son jeu (cf ci-dessous). La méthode `dessinerJeu` a pour objectif de dessiner l'image du jeu passé en paramètre `jeu` sur le `canvas` passé en paramètre.
- l'objet `clavier` est géré en interne et s'occupe de stocker les touches appuyées par l'utilisateur via un `KeyHandler`.

3.3 Description du fonctionnement du moteur

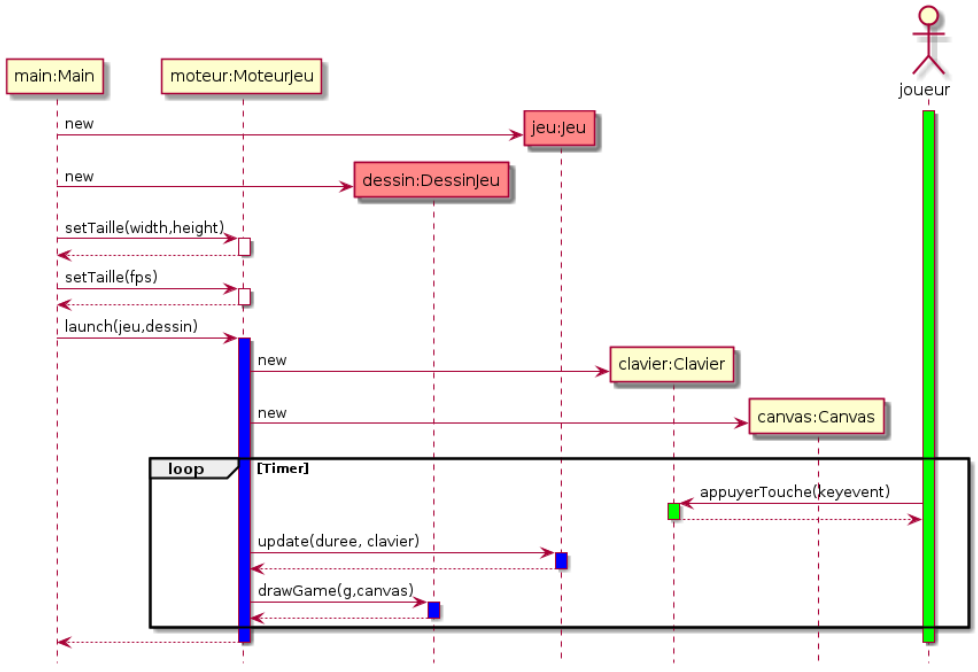

FIGURE 3 – Diagramme de séquence décrivant comment fonctionne le moteur de jeu. Les objets à fournir au moteur de jeu sont indiqués en rouge. Les périodes d'activité bleu correspondent à la boucle de jeu et les périodes d'activité en vert aux actions du joueur effectuées en parrallèle.

A l'utilisation, on doit disposer d'un objet `jeu` implémentant `Jeu` et d'un objet `dessin` implémentant `DessinJeu`. Lorsqu'on dispose de ces deux objets, lancer un jeu fonctionne de la manière suivante (cf diagramme de séquence de la figure **??**)

- on paramètrise le `MoteurGraphique` via les méthodes statiques `setTaille` et `setFPS` ;
- on lance le `MoteurGraphique` via la méthode statique `launchGame` en passant les objets `jeu` et `dessin` créés auparavant en paramètre ;
- le moteur se charge alors de construire l'interface graphique et les composants ;
- la boucle de jeu (représentée en bleue) se lance et consiste à chaque itération à
  - ☐ faire évoluer le jeu en fonction des commandes utilisateurs stockées dans la variable `clavier` en appelant la méthode `update(Jeu jeu, Clavier clavier)` de l'objet `jeu` passé au lancement du moteur ;
  - ☐ demander l'affichage du jeu via la méthode `dessinerJeu` de l'objet `dessin` passé au lancement du moteur.

En parralèle, l'utilisateur peut interagir avec l'application dans son propre `Thread` (représenté en vert)

- Lorsque l'utilisateur appuie sur (ou relâche) une touche, le `KeyHandler` transmet l'événement à l'objet `clavier`.
- Le `clavier` met à jour les booléens interne qui rendent compte de l'état du clavier (touche appuyée/relachée).
- L'objet `jeu` peut alors accéder à ces informations lorsque sa méthode `update` est appelée (avec ce `clavier` en parallèle).

Attention :
Les touches utilisées par défaut sont les touches `QSDZ`.

3.4 Utilisation du moteur

Lorsqu'on souhaite créer un nouveau jeu, il suffit simplement de définir une nouvelle classe implémentant `Jeu` et une nouvelle classe implémentant `DessinJeu` puis de passer les objets correspondant lors de la construction du `MoteurGraphique`.

Vous trouverez un exemple de jeu utilisant le moteur de jeu dans le package `arkanoidJeu` fourni sur arche (cf diagramme de classes figure **??**). Il est possible de lancer l'application à l'aide de la classe `MainArkanoid` qui construit les objets utiles et lance le moteur.

Le jeu fourni est un jeu de type casse-brique très simple. Il est structuré de la manière suivante (cf diagramme figure **??**) :

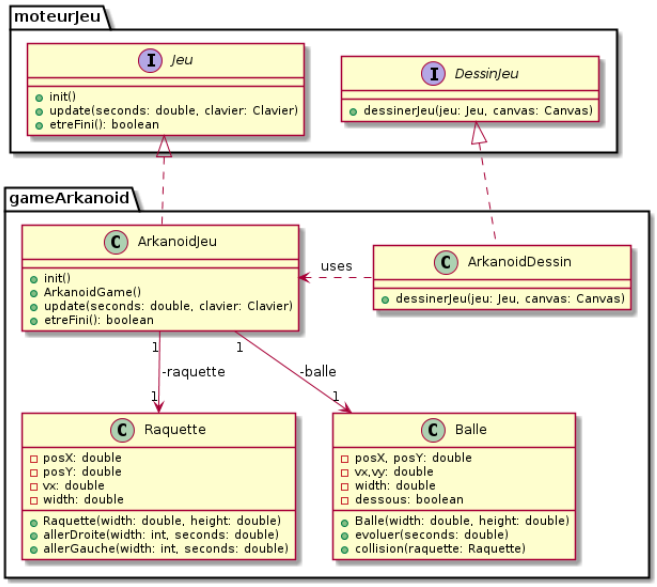

FIGURE 4 – Diagramme de classe décrivant la manière d'utiliser le moteur de jeu

- la classe `ArkanoidJeu` est le jeu à lancer, il implémente l'interface `Jeu`, contient les données du jeu et définit la méthode `update` qui encapsule les règles du jeu ;
- la classe `ArkanoidDessin` correspond à l'afficheur permettant d'afficher le jeu `jeu` et hérite de `DessinJeu`. La classe `ArkanoidDessin` accède au jeu `Jeu`[2] car il passé en paramètre à chaque fois que la méthode `dessinerJeu` est appelée par le moteur. La méthode peut alors dessiner l'état du jeu dans le canvas ;
- d'autres classes liées au jeu sont nécessaires pour manipuler correctement les données (comme la classe `Balle` et `Raquette`), mais ces classes sont spécifiques au jeu en question.

Question 10

Parcourez les classes `MainArkanoid`, `ArkanoidJeu` et `ArkanoidDessin` pour comprendre ce qu'il est nécessaire d'écrire pour proposer un jeu.

4 Utilisation du moteur et de labyrinthe

L'objectif de cette partie est d'adapter les classes fournies avec `Labyrinthe` pour avoir un jeu simple avec un rendu graphique temps réel.

L'évolution du jeu consiste simplement à déplacer le personnage dans la direction

2. ce qui est représenté par la flèche en pointillé qui est une une relation de dépendance - ce N'est PAS une association.

des touches appuyées par le joueur (tout en vérifiant que le personnage ne traverse pas un mur).

Question 11

Créer une classe `LabyJeu` qui implémente `Game` et qui utilise la classe `Labyrinthe`. On utilisera par défaut le labyrinthe fourni « `laby1.txt` ».

L'affichage graphique consiste simplement à afficher le personnage sous la forme d'un cercle rouge et les murs sous la forme de carrés noirs (cf figure **??**).

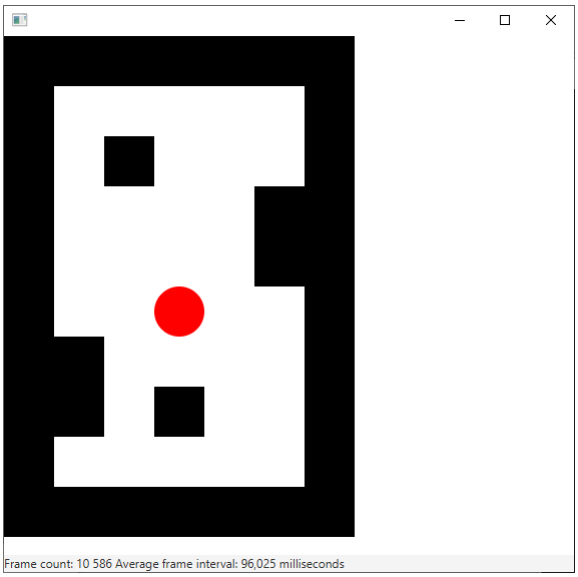

FIGURE 5 – Rendu graphique attendu (ici pour « `laby1.txt` ») : les murs sont représentés par des carrés noirs et le personnage par un cercle rouge.

L'objectif de cette partie est juste d'avoir une base pour votre jeu. Vous pourrez bien sûr faire évoluer cet affichage ultérieurement au cours de la `SAE_2.01`.

Question 12

En utilisant les méthodes vues en IHM pour modifier le contenu d'un objet `Canvas`, écrire une classe `LabyDessin` qui implémente `DessinJeu`.

En utilisant le diagramme de séquence fourni et en vous inspirant du package `arkanoid`, on souhaite mettre les différents éléments en place pour lancer le jeu sous une application graphique.

Question 13

Ecrire une classe `MainLaby` en charge de créer les objets `JeuLaby` et `LabyDessin` utiles et de lancer l'application.

Question 14

Créez des nouveaux diagrammes de classe et de séquence pour montrer comment vos classes fonctionnent avec le moteur de jeu.

Question 15

Ajouter le tag « `v1` » et stocker les différents documents dans votre dossier document :
« `documents/version1` ».

5 Bilan de la mise en place

La version « `v1` » de votre projet est finalisée. La partie suivante aborde la version « `v2` » montre comment faire une itération selon la démarche à suivre dans la SAE.

Deuxième partie

Première itération

6 Principe de la démarche incrémentale

La démarche suivie dans le cadre du projet sera une **démarche incrémentale** consistant à ajouter progressivement des fonctionnalités au projet.

Attention :
Cette démarche se fera en plusieurs itérations. Une itération dure 4h et consiste à

- décider des fonctionnalités (deux max) à ajouter à cette version (10 min) ;
- expliciter les critères de validation (10 min) ;
- réfléchir collectivement à une conception (1h) ;
  - ☐ pour chaque fonctionnalité, dessiner un ou plusieurs diagrammes de séquence ;
  - ☐ faire des diagrammes de séquence pour les cas très particuliers ;
  - ☐ mettre à jour votre diagramme de classe en conséquence.
- se répartir le travail de manière équitable à partir de la conception (10 min) ;
- écrire les tests à valider et réaliser le code de l'application(1h30 min) ;
- valider grâce aux tests les nouvelles fonctionnalités par rapport aux critères exprimés et vérifier que le reste de votre application fonctionne encore correctement (30 min).

Une fois que cette version est validée, on peut passer à la suivante.

Attention :
Le cœur de cette démarche est de ne considérer qu'une ou deux fonctionnalités à la fois. Ne vous concentrez que sur une fonctionnalité et ne passez à la suite que lorsque cette fonctionnalité sera validée.

Dans cette partie, on suivra la démarche sur un cas particulier : l'ajout d'un monstre dans le labyrinthe. Dans la `SAE_2.01`, vous serez amené à reproduire la démarche de manière autonome.

7 Analyse et conception (1h)

Avant tout codage, on attend une analyse détaillée des fonctionnalités à ajouter et des documents de conception rendant compte de votre travail. Cette analyse se fait en trois étapes :

- Réfléchir aux fonctionnalités et aux critères de Validation ;
- Écrire collectivement des diagrammes de classe et de séquence ;
- Vous partager le travail de réalisation.

7.1 Fonctionnalités liées à l'ajout d'un monstre dans l'application

Une **fonctionnalité** correspond à un comportement attendu de l'application. Elle est définie par un descriptif textuel accompagné de critères de validation.

- le **descriptif textuel** doit exprimer en une ou deux phrase l'objectif de la fonctionnalité.
- les **critères de validation** doivent exprimer ce que l'application devra vérifier une fois que la fonctionnalité est implémentée et quels seront les tests associés.

Cette partie est très importante car c'est elle qui va définir l'objectif à atteindre (tests, validation). Si ces descriptifs sont flous, vous risquez de ne pas avoir le même point de vue sur ce qu'il y a à faire et développer du code non compatible entre vous.

Sur cet exemple, il s'agit de prendre quelques minutes pour réfléchir à ce que signifie ajouter un monstre dans l'application.

Question 16

Dans un fichier texte, listez les fonctionnalités les plus simples possibles associées à l'ajout d'un monstre dans l'application. Précisez à chaque fois, les critères de validation.

7.2 Fonctionnalités retenues

De manière plus précise, dans ce TP, on se limitera aux fonctionnalités suivantes :

1. **Donner une position initiale au monstre**
   - **Descriptif** :
     - ☐ le monstre débute sur une case décrite dans le fichier labyrinthe.
   - **Critères de validation** :
     - ☐ le monstre doit avoir une position initiale.

- ☐ le monstre se trouve sur la case indiquée dans le fichier labyrinthe.
- ☐ le monstre est représenté par le caractère `'M'` dans le fichier labyrinthe.
- ☐ le monstre ne se trouve pas sur la même case que le personnage.

2. **Afficher le monstre**
   - **Descriptif** :
     - ☐ le jeu doit afficher le monstre à sa position.
   - **Critères de validation** :
     - ☐ le monstre doit être affiché à la bonne position dans le labyrinthe.
     - ☐ le monstre sera représenté sous la forme d'un cercle violet de la taille du personnage.
3. **Considérer le monstre dans les déplacements du personnage**
   - **Descriptif** :
     - ☐ lorsque le jeu évolue, le personnage ne peut pas se déplacer sur la case du monstre.
   - **Critères de validation** :
     - ☐ le monstre constitue un obstacle pour le personnage.
     - ☐ le monstre et le personnage ne peuvent pas se trouver sur la même case.
     - ☐ le personnage ne peut pas traverser la case du monstre.
4. **Déplacer le monstre** (optionnel si les autres fonctionnalités sont finies)
   - **Descriptif** :
     - ☐ lorsque le jeu évolue, le monstre choisit une case adjacente de manière aléatoire et tente de s'y déplacer.
   - **Critères de validation** :
     - ☐ le monstre doit se déplacer sur une case adjacente. Il considère les 4 directions de déplacement possibles.
     - ☐ le monstre ne peut pas se déplacer sur mur. S'il tente de se déplacer sur cette case, il ne bouge pas.
     - ☐ le monstre ne peut pas se trouver sur la même case que le personnage. Lors d'une évolution du jeu, il se déplace aprés le personnage héros.

Ces fonctionnalités sont déjà incluses dans le fichier « `fonctionnalite.txt` » dans le répertoire « `document/version_2` » fourni dans le fichier `zip`. Un tel fichier doit être produit à chaque nouvelle version de l'application. Le descriptif des fonctionnalités s'exprime simplement sous la forme suivante.

```
- fonctionnalité 1 : nom
- descriptif texte : [...]
- critères de validation :
    - critère 1 [...]
    - critère 2 [...]

[....]
```

7.3 Diagramme de Classe et de séquence

Maintenant que les fonctionnalités ont été explicitées, cette partie, **fondamentale dans le cadre du projet**, consiste à faire des choix de conception pour implémenter les fonctionnalités choisies.

Question 17
Pour **chaque** fonctionnalité, proposez un diagramme de séquence qui rend compte de la manière dont votre programme fonctionnera. Cela nécessitera de définir les méthodes utiles qu'il faudra ajouter à votre diagramme de classe.

Question 18
Proposer un diagramme de classe qui rende compte de la nouvelle forme de votre application pour implémenter les fonctionnalités choisies.

Question 19
Déposez l'ensemble des diagrammes dans le répertoire « `document/version_2` » associé à cette version. Cela inclut les fichiers `plantUML` et les fichiers images générés.

7.4 Partage des tâches

L'intérêt de disposer de diagrammes de classe et de séquence est que vous savez désormais quelles sont les méthodes à écrire et les attributs à ajouter.

Il devient donc possible de se répartir le travail sur cette base.

Question 20
A partir des fonctionnalités et des diagrammes, répartissez-vous le travail entre vous. N'oubliez pas de penser aux tests JUnit à écrire et à vous les répartir.

Question 21
Faites apparaître cette répartition dans le fichier décrivant votre itération.

8 Implémentation (1h)

8.1 Implémentation

Pour simplifier le travail, avant de remplir les méthodes, il peut être utile de créer des méthodes vides ne contenant que des

```
throw new Error("TODO");
```

Cela permet de compiler les classes dès le départ et de s'assurer que tous le monde utilisera bien les mêmes noms de méthodes.

Question 22
La répartition des tâches ayant été faite, chaque membre du groupe peut développer la partie qui lui a été dédiée. Les tests, eux aussi à écrire à cette étape, peuvent se développer en parallèle.

8.2 Validation et Test

Question 23
Validez les tests qui ont été écrits.

Question 24
En cas de problème, utilisez le debugger pour localiser votre erreur, la corriger et valider les tests.

8.3 Finalisation et tag

Les tags permettent de nommer des versions de votre application. Cela permet de pouvoir facilement revenir à une version antérieure et fonctionnelle de l'application.

Les tags seront aussi utiles pour pouvoir vérifier l'évolution de votre code entre différentes versions.

Question 25
Une fois les tests passés, ajouter un fichier texte « `bilan.txt` » qui fait le bilan du travail réalisé lors de cette version (quels sont les bugs restants, qu'est ce qui

a été validé, ...).

Question 26
Enfin, ajouter un tag « `v2` » correspondant à votre numéro de version et pusher ce tag.

9 Bilan

Á l'issue de ce TP, vous avez effectué une itération complète.

Vous devez avoir produit :

- un document « `document/version_2/fonctionnalites.txt` » décrivant les **fonctionnalités développées** ;
- un **diagramme de classe et des diagrammes de séquence** pour chaque fonctionnalité dans le répertoire « `document/version_2` » décrivant le fonctionnement de votre application ;
- un **code** mis à jour qui fonctionne conformément à vos choix de conception ;
- un ensemble de **tests** qui vérifient les critères de validation des différentes fonctionnalités ;
- un fichier **bilan** présentant un état de votre version (bug, validation, ...) ;
- un **tag de version** finalisant cette version.

Chaque itération de 4h de la `SAE_2.01` devra fonctionner sur ce principe. A la fin de la `SAE_2.01`, vous devrez donc disposer des répertoires

- « `document/version_3` »
- « `document/version_4` »
- « `document/version_5` »

Chacun de ces répertoires possède les documents demandés (fonctionnalités, diagrammes de classes et de séquence, résultat du test).

La `SAE_2.01` étant une `SAE_2.01` centrée sur la conception et l'IHM, ces documents sont **attendus** et constitueront **une partie importante de l'évaluation**.