Bilan
SAE 3.01 – Développement d'une application

Table des matières :

Fonctionnalités réalisées :

Itération 1 :

- Importation et Lecture d'un package .class

Itération 2 :

- Importation et lecture d'un fichier .class
- Générer l'interface graphique

Itération 3 :

- Générer l'interface graphique
- Génération automatique de diagrammes de classes à partir de packages Java.

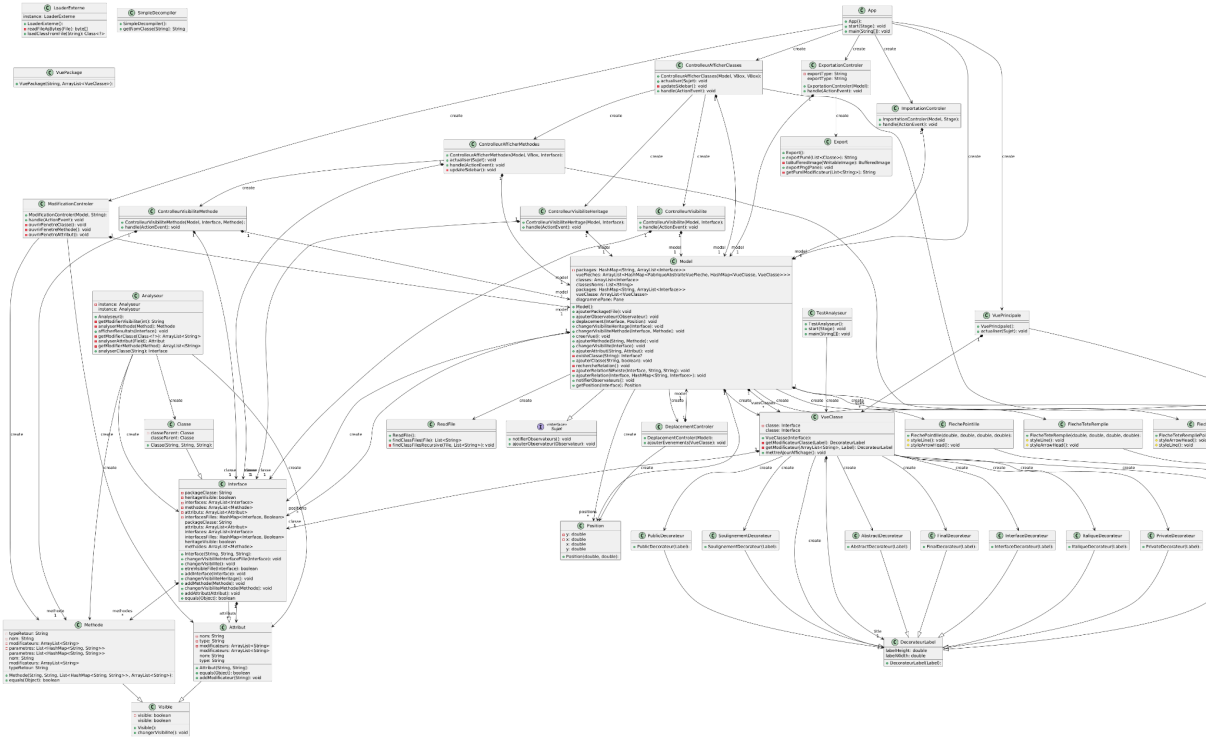
Itération 4 :

- Génération automatique de diagrammes de classes à partir de packages Java.
- Générer l'interface graphique
- Exportation des diagrammes au format image, PlantUML ou résultat compilé.

Itération 5 :

- Modification des diagrammes : ajout de classes, méthodes, etc.
- Afficher ou masquer des éléments (classe et méthodes)
- Afficher ou masquer l'héritage et les implémentations

Diagramme de classe :



Difficultés rencontrées dans le projet.

LoaderExterne: Une classe qui en hérite d'une autre ne peut pas être chargée si elle hérite d'une autre classe ou interface n'étant pas chargée une erreur est créé car j'utilise defineClass.
Complexité du problème :
Si la classe devant être chargée avant n'existe pas sur l'ordinateur, il est impossible d'utiliser defineClass or le seul moyen fiable d'obtenir le nom complet de la classe est de décompiler le fichier, et de l'utiliser dans un loader spécial qui permet de charger.

Potentielle solution : créer une classe (une classe vide avec seulement le nom de la classe) à la volée et la charger puis tenter à nouveau le defineClass qui fonctionnera puisque la classe nécessaire est chargée.
Cette classe sera remplacée lors du chargement de la classe originale.

Potentiel nouveau problème : comment identifier les extends et les implements dans la classe SimpleDecompiler.

Gestion de la Visibilité :
Les contrôleurs sont à la fois des vues donc problèmes de conception due à un manque de temps et au fait que les vues ne sont pas réellement des vues (ne sont pas actualisés par le modèle).

Deuxième problème lors de la création des objets représentant les classes, les booléens permettant de savoir si l'héritage est visible sont null donc impossible de masquer l'héritage, car la méthode renvoies des null pointer exception, les flèches ne se sont pas masquables à cause des mêmes null

Cependant, lorsque la classe est masquée les actions concernant les méthodes et l'héritage est masqué car ça n'a pas de sens de masquer les méthodes d'une classe masquer/afficher ou son héritage.

Erreur de conception sur le modèle MVC car les vues effectuées des opérations et certaines vues n'implémente pas observateur, donc ne sont pas actualisés par le modèle, en d'autres termes c'est ce que l'on appelle un code spaghettis beaucoup d'opérations ne sont pas effectués par les classes qui devraient les effectuer.

Erreur de conception lors de la création de fonction dans l'ensemble des classes, beaucoup trop d'opérations différentes sont effectuées dans la même fonction ce qui rend compliqué la compréhension et rend très difficile la localisation des bugs.

Manque de communication (personne ne sait où on en est dans l'avancement global du projet.)
Solution: éviter dans les réunions de dire que c'est presque fini, mais dire exactement ce qui n'est pas encore fonctionnel, pour permettre de mieux déterminer les futures tâches et ainsi d'éviter de devoir réécrire des gros bouts de codes, car les codes cumulés ne sont pas compatibles.

Répartition du travail:

Mathis Segard et Jules Andre se sont majoritairement concentré sur le code qui n'est pas visible par l'utilisateur (représentation des Classes, leur chargement, gestions des classes)

Gabriel Comte et Valentino Lambert se sont majoritairement concentré sur la partie visuel de l'application

Itération 1 :

- Importation (Valentino et Gabriel) et Lecture (Jules et Mathis) d'un package .class

Itération 2 :

- Importation et lecture d'un fichier .class (Jules et Mathis)
- Générer l'interface graphique (Valentino et Gabriel)

Itération 3 :

- Générer l'interface graphique (Valentino et Gabriel)
- Génération automatique de diagrammes de classes à partir de packages Java. (Jules et Mathis)

Itération 4 :

- Génération automatique de diagrammes de classes à partir de packages Java. (Jules et Mathis)
- Générer l'interface graphique (Gabriel)
- Exportation des diagrammes au format image (Valentino), PlantUML (Jules) ou résultat compilé.

Itération 5 :

- Modification des diagrammes : ajout de classes, méthodes, etc. (Valentino)
- Afficher ou masquer des éléments (classe et méthodes) (Mathis)
- Afficher ou masquer l'héritage et les implémentations (Mathis)
- Correction de bug export (Jules)
- Documentation (Jules)

Présentation des éléments qui nous rendent fiers / originaux :

Chargement des classes :
contrairement à tout les autres groupes pour le chargement des classes on récupère le nom complet de la classe en la décompilant et en concervant uniquement le nom de la classe, ce qui permet de pouvoir charger une classe même si elle n'est pas dans un dossier qui coincide avec

Export du diagramme en Puml :
La spécificité de cette fonctionnalité est qu'elle n'était pas prévue dans la conception (aucun diagramme n'a été fait dessus et les classes ne sont pas dans le diagramme de classe prévu). Nous avons donc dû la faire durant l'itération.
L'export de diagramme fonctionne grâce à un switch dans la classe ExportationControler qui nous permet de choisir le type de l'export (Puml ou Png (non-fonctionnel)). Le controler communique avec l'app qui contient un menubar et des boutons associés aux différents types d'export. Ensuite, après avoir choisi le type voulu, le controler va communiquer avec la classe export contenant les méthodes associées. Pour le cas du Puml, la méthode créée un StringBuilder en ajoutant tout d'abord le prefix utilisé dans le code Puml : @startuml
Ensuite, la méthode boucle sur les classes pour récupérer son nom suivi d'une accolade pour commencer l'énumération des attributs et des méthodes qui seront récupéré de la même façon que le nom de la classe. La seule spécificité est que pour les méthodes et les attributs, on récupère son modificateur et nous le traduisons au format Puml (Public -> + , Private -> -, …). Après avoir terminé la boucle, le StringBuilder va finir le string avec le prefix @enduml. On utilisera ensuite un BufferedWritter pour créer le fichier et l'enregistrer dans le dossier voulu.

Différences avec l'étude préalable :

Les fonctionnalités initialement prévus étaient :

- Importation et lecture d'un fichier .class (pour créer un squelette de class)
- Génération automatique de diagrammes de classes à partir de packages Java.
- Exportation des diagrammes au format image, PlantUML ou résultat compilé.
- Générer l'interface graphique
- Modification des diagrammes : ajout de classes, méthodes, etc.
- Générer les squelettes des classes
- Afficher ou masquer des éléments (classe et méthodes)
- Afficher ou masquer l'héritage et les implémentations

Au final celles réalisées et entièrements fonctionnelles sont :

- Importation et lecture d'un fichier .class (pour créer un squelette de class)
- Génération automatique de diagrammes de classes à partir de packages Java.
- Exportation des diagrammes au format PlantUML.
- Générer l'interface graphique
- Générer les squelettes des classes
- Afficher ou masquer des éléments (classe et méthodes)
- Afficher ou masquer l'héritage et les implémentations

Patrons de conceptions et architecture utilisé :

Dans ce projet réalisé par nos soins, nous prévoyons d'utiliser les patrons singleton et fabrique pour la classe FabriqueBasicGraphicAttributes qui nous permettra de créer des Dans ce projet nous avons d'utilisé les patrons singleton et fabrique pour la classe FabriqueBasicGraphicAttributes qui nous permet de créer des Objet BasicGraphicAttributes avec des paramètres pré remplis pour facilement distinguer visuellement les attributs, les classes et les méthodes et une seule instance sera nécessaire, car les objets produits ne dépendent pas de caractéristiques de cette instance.

Nous avons également utilisé le patron Observateur à travers le modèle MVC, où l'observateur est la classe abstraite Vue, qui se met à jour lorsque le modèle lui ordonne pour changer l'affichage, le modèle lui effectue des opérations lorsque qu'un Contrôleur lui ordonnera de le faire avec éventuellement des informations transmises sur l'opération (comme par exemple les nouvelles coordonnées de la vue et la vue).

Nous utilisons aussi le patron composite pour gérer la composition des classes, où les composants sont les attributs primaires et les classes seront les composites, cela permet de gérer plus efficacement les attributs d'une classe qui peuvent être une classe ou un attribut primaire.

Le patron stratégie lui sert à éviter les copier coller dans les classes en regroupant des éléments communs et à utiliser une méthode sans avoir à se préoccuper de quelle classe est l'instance comme pour les contrôleurs ou les vues, et cela, permet d'alléger les classes Interface, Attribut et Classe via une interface CompositionClasse qui possède des attributs et méthodes communes, puis via un héritage entre Classe et Jonction où Jonction possède les attributs en communs, la même chose est faite avec une interface Visibilite pour gérer les éléments masquables. Les instances d'Interface n'étant pas un attribut de classe, elle implémente l'interface CompositionClasse pour éviter de posséder des instances de cette classe en guise d'attribut.

Graphe de scène :