RAPPORT GÉNÉRAL
ITÉRATION 2

[redacted] Édouard LUMET
[redacted]

11 juin 2018

Table des matières

1 Rappel du sujet **2**

2 Fonctionnalités et *User Stories* **2**

3 Découpage de l'application **2**

4 Diagrammes de classe **3**

5 Choix de conception, problèmes et solutions **3**

6 Organisation et méthode agile **4**

A Annexe 1 **5**

B Annexe 2 **6**

C Annexe 3 **7**

1 Rappel du sujet

Notre projet consiste en la création d'une application avec interface graphique ayant pour utilité de :

- visualiser des diagrammes de classe UML,
- générer un diagramme de classe UML à partir du code JAVA, et inversement,
- éditer un diagramme ou du code JAVA.

2 Fonctionnalités et *User Stories*

Les principales fonctionnalités sont la visualisation, la génération, l'édition et l'exportation.

Les *User Stories* effectuées lors de la première itération sont les suivantes :

- En tant qu'utilisateur je souhaite visualiser le diagramme de classe et le diagramme d'analyse afin d'avoir une représentation conceptuelle et visualiser l'implantation du code. <40>
- En tant qu'utilisateur je souhaite à partir du code JAVA, générer un diagramme de classes afin de faciliter la visualisation du code. <40>

Les *User Stories* effectuées lors de la deuxième itération sont les suivantes :

- Afficher deux classes reliées par une relation de tous types (agrégation, composition, ...etc) (VISUALISER) <13>
- Générer du code java à partir des éléments d'un diagramme UML (classe, enum, interface, ...etc) (GÉNÉRER) <5>→<20>
- Intégrer l'affichage des méthodes et des attributs dans l'affichage d'une classe <3>
- Mettre hashmap pour une future exportation/affichage dans un fichier et faciliter la manipulation du code généré (GÉNÉRER) <3>
- Concevoir un diagramme UML comportant des élements et des relations UML <3>
- Afficher les éléments UML (classes, enum, ...etc) avec un nom et vide de contenu (VISUALISER) <8>
- Modifier le nom d'une classe via l'interface graphique (ÉDITER) <13>
- Créer le diagramme de classe pour le package generator <2>
- Afficher une méthode avec ses paramètres, sa visibilité et ses modificateurs (VISUALISER) <3>
- Afficher les attributs d'une classe avec modificateurs et visibilité (VISUALISER) <3>
- Concevoir le modèle permettant à tous les éléments d'avoir une visibilité et des modificateurs <5>

3 Découpage de l'application

Le programme suit une architecture en deux sous-systèmes majeurs :

- Un sous-système d'édition de diagramme, qui contient lui-même trois sous-système :
  - modèle : la représentation en Java d'un diagramme UML et de ces éléments,
  - vue : La représentation graphique de l'éditeur,

  - contrôleur : Le code java permettant à l'utilisateur de manipuler la vue
- Un sous-système dédié à la génération de code à l'aide du patron visiteur.

4 Diagrammes de classe

Les diagrammes de classe modélisant notre projet, notre application sont multiples car nous avons un diagramme par fonctionnalité.

En annexe 1 figure le diagramme modélisant la partie modèle du sous-système éditeur.
En annexe 2 figure le diagramme modélisant la partie vue du sous-sytème d'édition.
En annexe 3 figure le diagramme modélisant le sous-système de génération.

5 Choix de conception, problèmes et solutions

Les principaux choix de conception et réalisation ont été tout d'abord d'appliquer un patron d'architecture qui est MVC (modèle-vue-contrôleur).

Le patron Visiteur a été pensé pour la mise en place des fonctionnalités futures traitant de la génération de code.

Lors du développement divers problèmes ont été soulevés en rapport avec la visualisation des éléments. Tout d'abord le framework JavaFX a été proposé comme base graphique pour ses nombreux composants qui pouvait nous faciliter le développement mais ce dernier étant très récent et par manque d'expérience au sein de l'équipe, le choix s'est porté sur Swing afin d'éviter toute perte de temps.

La représentation des éléments est un second problème, entre autre l'organisation des relations dans le plan (comment savoir si deux relations ne se recoupent pas ? comment faire pour qu'une relation ne passe pas sur une entité ?).

Le patron Visiteur se compose d'une interface (nom de l'interface). Celle-ci permet l'implémentation d'autres générateurs de code et définit les méthodes que chaque générateur doit implémenter. Chaque composant UML (class, enum, ...etc) doit également implémenter une méthode (appelé "accept" dans notre projet). Elle va appeler une méthode du générateur de code lui permettant de générer du code spécifique pour chaque composant UML (enum, class, interface, ...etc).
Pour l'instant seul le générateur de code JAVA à été implémenté permettant de générer du code d'élément UML (mais pas leurs relation). Sa complexité réside dans le respect du "checkstyle" lors de la réalisation du code.

6 Organisation et méthode agile

L'organisation consiste en une répartition individuelle des tâches techniques entre les membres du groupe, tout en utilisant Github pour la gestion de sources et WeKan pour la gestion de projet (backlog, *US*, ...).

La mise en oeuvre des méthodes agiles a conduit dans un premier temps à l'écriture de *User Stories* et de *Technical Tasks* ainsi qu'à une prévision du travail effectué lors du Sprint 1.

Cependant cette première mise en oeuvre s'est avérée être un échec, l'ensemble du groupe étant parti directement sur une conception de l'ensemble de la solution avec la création d'un diagramme de classe UML. Ce problème ne nous a pas permis de terminer l'ensemble des tâches dans les *US*.

Après en avoir pris conscience au cours du Sprint 1, le backlog a été retravaillé et les User Stories re-validées afin de préparer au mieux le Sprint 2. Le diagramme produit n'a pas été utilisé tel quel mais a permis d'avoir une idée plus claire de l'architecture générale du programme.

Comparé à la première itération, la gestion de projet Agile fut bien meilleure cette fois-ci. Un terrain d'entente a été trouvé et nous avons réussi à appliquer les différents conseils et à considérer les différentes remarques de la séance SCRUM du 6 juin.

Des réunions ont été organisé au sein de l'équipe afin de valider certains points durant l'itération et de tenir les autres membres au courant d'évolutions apportées.

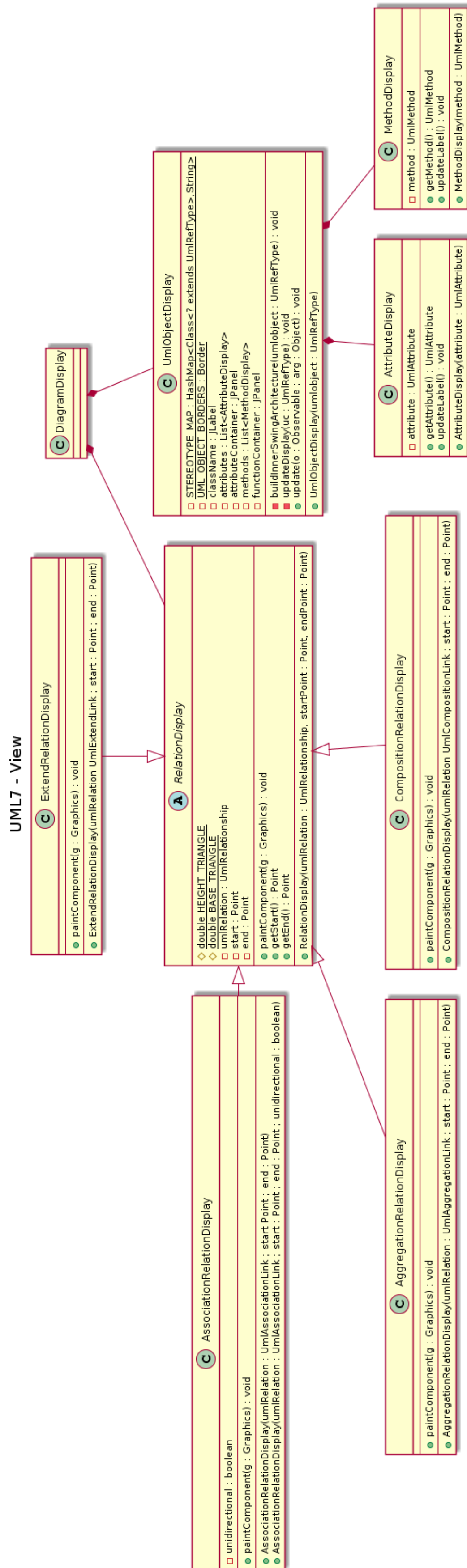
L'équipe a ainsi gagné en efficacité et le travail a été plus fluide. Chaque membre d'équipe a sa propre US et en sélectionne une nouvelle une fois la précédente terminée ou en phase de testing. Les US sont clairement définies et suffisamment atomiques.

Les objectifs de début d'itération ont été dépassé et des tâches supplémentaires ont pu être entamé.

A Annexe 1

UML7 - model

B Annexe 2



C Annexe 3