RAPPORT GÉNÉRAL
Itération 3 - Fin

[redacted] Édouard Lumet

15 juin 2018

Résumé

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.

Cet outil, nommé UML7, est intégralement développé en JAVA. Sa conception permet un support de différents langages de programmation orientée objet quant à la génération de code.

L'objectif de l'outil UML7 est de permettre au développeur de passer rapidement de l'étape conception en éditant un diagramme UML à l'étape de réalisation. En effet, UML7 génère un squelette de code dans le langage souhaité (JAVA uniquement pour l'instant), il ne reste alors plus qu'à écrire le corps des classes dans les différents fichiers obtenus dont figure la classe avec ses attributs et ses méthodes « vides ».

Sommaire

1 Gestion de projet

1.1 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>

Lors de la dernière itération, nous avons effectué les *User Stories* suivantes :

— Modifier un attribut via l'interface graphique (ÉDITER) <8>→<5>
— Supprimer une méthode dans un composant UML via l'interface graphique (ÉDITER) <5>
— Gérer les erreurs dans la création/modifciation d'un composant UML (GÉNÉRER) <5>→<20>
— Sauvegarder le diagramme UML en format uml7 <5>→<13>
— Charger un diagramme UML depuis le format uml7 <5>
— Afficher un message popup lors d'une erreur (VISUALISER) <2>
— Créer une méthode via l'interface graphique avec des paramètres pour un éléments UML et le type de retour (ÉDITER) <8>→<13>
— Modifier une méthode via l'interface graphique (ÉDITER) <8>→<13>
— Exporter un diagramme de classe en JAVA dans plusieurs fichiers et dossiers (EXPORTER) <5>→<8>
— Créer et supprimer les attributs via l'interface graphique pour un élément UML avec modificateur et visibilité (ÉDITER) <8>
— Créer et supprimer un élément UML (classe, interface, ..etc) vide via l'interface graphique

avec un nom (ÉDITER) <13>→<8>

Les *User Stories* techniques sont :

- Faire le diagramme de cas d'utilisation de l'application pour la présentation finale <3>
- Factorisation du nom dans UmlEntity <1>
- Ecrire les tests pour le model d'élements UML (classe, enum, ...etc) (GENERER) <13>→<5>
- Créer le diagramme de classe du contrôleur <3>
- S'expliquer les différentes parties du code <13>
- Terminer la Javadoc et la générer <2>→<8>
- Créer le diagramme général de toute l'architecture (tous les packages dans un diagramme) <3>→<8>
- Mettre à jour tous les diagrammes <5>
- Faire le rapport général LaTeX<5>→<13>
- Prise en compte des méthodes 'final' dans une classe abstraite <2>
- Amélioration génération Java (conformité checkstyle) (GÉNÉRER) <5>
- Implanter retour des méthodes selon type primitif (GÉNÉRER) <3>
- Mettre à jour les tests (GÉNÉRER) <3>
- Écrire les tests pour le modèle (classe, enum, ...etc) (GÉNÉRER) <13>→<5>

1.2 Sprint reviews

À la fin de chaque itération, nous avons vérifié différents points tels que :

- la réalisation des objectifs,
- la validation des critères d'acceptation,
- la vélocité,
- la valeur métier,
- le *Definition of Done*.

Pour la valeur métier, nous nous sommes appuyés sur le système *MoSCoW* qualifiant nos *US*. Nous avons considéré qu'un *Must* équivalait à 200 de valeur, 150 pour un *Should*, 100 pour un *Could* et 50 pour un *Would*.

Notre *Definition of Done* est commun à toutes les *User Stories* et permet de les valider ainsi que de valider la fin de l'itération. Il est le suivant :

- Respecter les objectifs pédagogiques
  - Appliquer de la programmation orientée objet (POO)
  - Avoir une interface graphique
  - Utiliser un MVC
- Javadoc en anglais
- Tests unitaires JUnit (partie non graphique)
- Validation manuelle par deux autres personnes (partie graphique)
- Tests d'intégration
- GIT propre : cf référentiel Git
- Rapports d'itération complets
- Diagrammes de classe complétés au fur-et-à-mesure
- Validation Sonarqube

1.2.1 Itération 1

Notre première itération n'a pas été correctement gérée d'un point de vue gestion de projet Agile, comme décrit dans la section suivante. Nous avions tout de même définie une vélocité attendue puis calculée la vélocitée effective ainsi que la valeur métier en fin d'itération.

Notre vélocité prévue était de 80 pour une vélocité réalisée de 40. Cette différence s'explique par la non prise en compte des *User Stories* techniques dans notre sprint ainsi que la mauvaise gestion et entente dans l'équipe.

La valeur métier livrée fut quant à elle de 800, la majeure partie du développement étant consacrée au modèle. Nous pouvions alors montrer l'affichage d'une méthode et d'un attribut dans une fenêtre contenant un *exit*.

1.2.2 Itération 2

L'objectif de notre itération 2 était d'afficher un diagramme simple. Nous avions une vélocité estimée de 84 et notre vélocité réelle fut de 89. Nous avons par ailleurs atteint notre objectif. En effet, nous pouvions afficher un diagramme avec deux classes contenant attributs et méthodes et une relation entre ces deux classes. La valeur métier livrée était alors de 1150.

Nous avons également validé avec succès le *Definition of Done*. Quant aux critères d'acceptation ; voici un exemple pour une *US* :

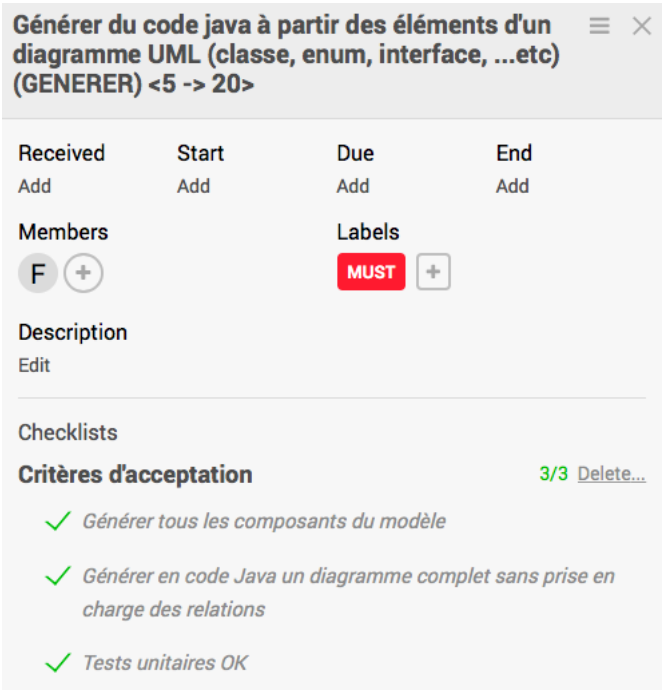


FIGURE 1 – Contenu d'une *US* avec ses critères

1.2.3 Itération 3

Les objectifs de notre itération 3 étaient les suivants :
— Sauvegarder et charger un diagramme UML
— Créer des éléments via l'interface graphique

— Traiter les cas d'exception
— Éditer les éléments UML

Nous avons réalisé tous nos objectifs et nous couvrons toutes nos *features*.

Nous avions estimé en début de sprint une vélocité de 127 et avons atteint une vélocité de 159. De plus, nous avons livré 1800 points de valeur métier. Cela est cohérent étant donné l'avancement du projet. Nous pouvons effectivement générer puis exporter du code JAVA dans des fichiers à partir d'un diagramme édité directement dans l'interface graphique. Il est également possible de sauvegarder son travail pour reprendre plus tard via des fichiers d'extension uml7.

Quant au *Definition of Done*, un point n'a pas pu être validé qui est la réalisaiton de tests JUnit. En effet, par manque de temps, nous avons dû nous en tenir aux tests réalisés lors de la précédente itération.

1.3 Organisation

1.3.1 Aspect humain

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.

La dernière itération fut un succès à différents niveaux. Concernant l'entente au sein de l'équipe, en poursuivant nos efforts d'écoute, nous avons réussi à conserver l'ambiance de l'itération 2.

Quant à l'efficacité de l'équipe, elle fut accrue car nous n'avions pas besoin de rattraper les difficultés de l'itération précédente comme lors du début d'itération 2. C'est ce qui explique l'évolution de notre vélocité, atteignant ainsi notre « vitesse de croisière ».

1.3.2 Aspect technique

D'un point de vue technique, nous avons utilisé Git pour la gestion de versions de notre code et Github pour la centralisation de celui-ci. Nous avons donc défini une utilisation dans un référentiel afin que tous l'emploient de la même manière. Ce référentiel est fourni dans le dossier *doc* du dépôt SVN ou ici.

2 Présentation technique

2.1 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.
- Un sous-système dédié à l'exportation permettant d'exporter un diagramme de classe en fichier .uml7 et en code JAVA. Ce sous système utilise le générateur de code JAVA.
- Un sous-système dédié à importation de fichier avec l'extension .uml7.

2.2 Diagrammes uml

Tout d'abord, le diagramme de cas d'utilisation explicitant les process majeurs de notre application est présentée en annexe 1.

Les diagrammes de classe modélisant notre projet, notre application sont multiples car nous avons un diagramme par fonctionnalité.
En annexe 2 figure le diagramme modélisant la partie modèle du sous-système éditeur.
En annexe 3 figure le diagramme modélisant la partie vue du sous-système d'édition.
En annexe 4 figure le diagramme modélisant la partie contrôleur du sous-système d'édition.
En annexe 5 figure le diagramme modélisant le sous-système de génération.
En annexe 6 figure le diagramme modélisant le sous-système d'exportation.
En annexe 7 figure le diagramme modélisant le sous-système d'importation.

2.3 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).

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 ?). Par manque de temps, la gestion des relations n'a pas été finalisé lors de l'itération 3.

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 implanter 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), l'architecture actuelle est évolutif, on pourra aisément supporter la prise en charge d'un nouveau langage.

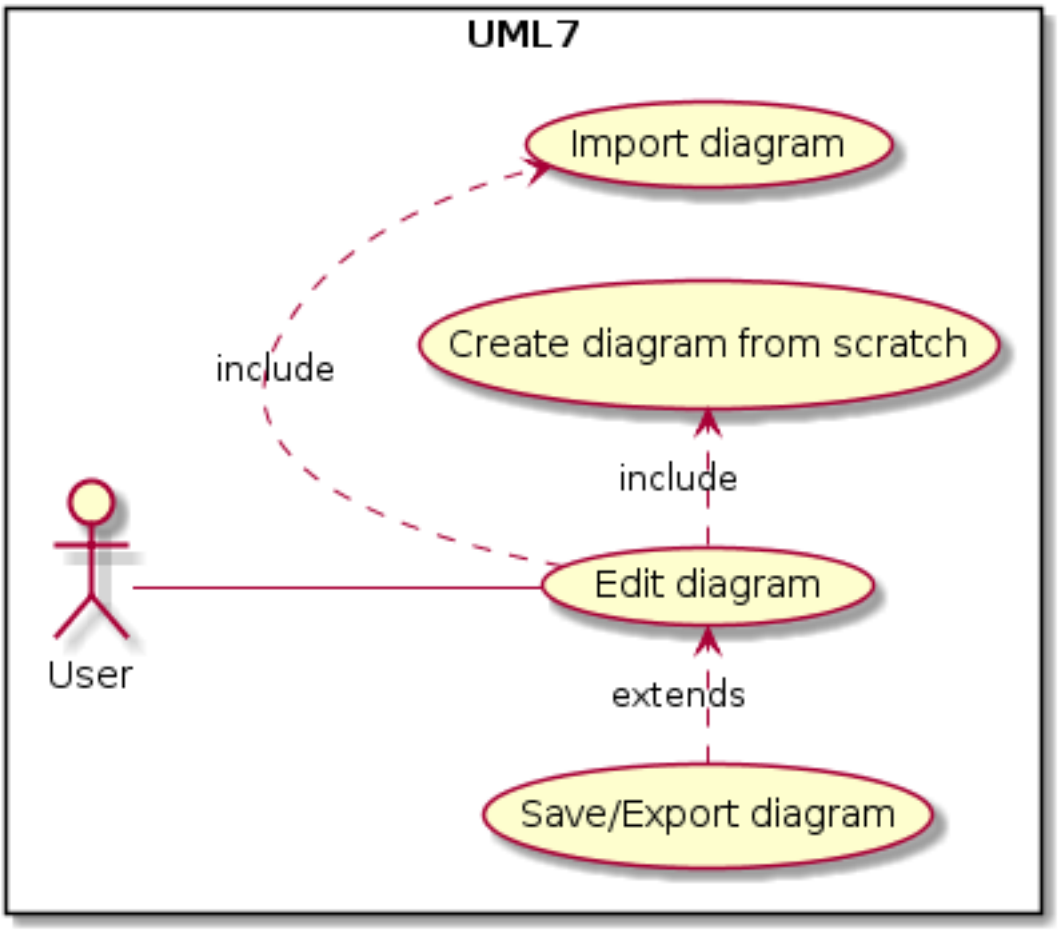
Pour l'instant seul le générateur de code java à été implémenté permettant de générer du code d'élément UML (mais pas leurs relation). Sa complexité réside dans le respect du "checkstyle" lors de la réalisation du code.

Un des problèmes fut la mise en place du patron MVC actif, du fait qu'on dispose d'un projet avec beaucoup d'interaction à la vue complexe, il existe entre le modèle et la vue un fort couplage car la vue d'un élément UML a besoin de connaitre le modèle dès sa création. Ce problème ne peut être résolu à moins d'utiliser un patron de conception de type MVVM (modèle-vue vue-modèle) plus adapté.

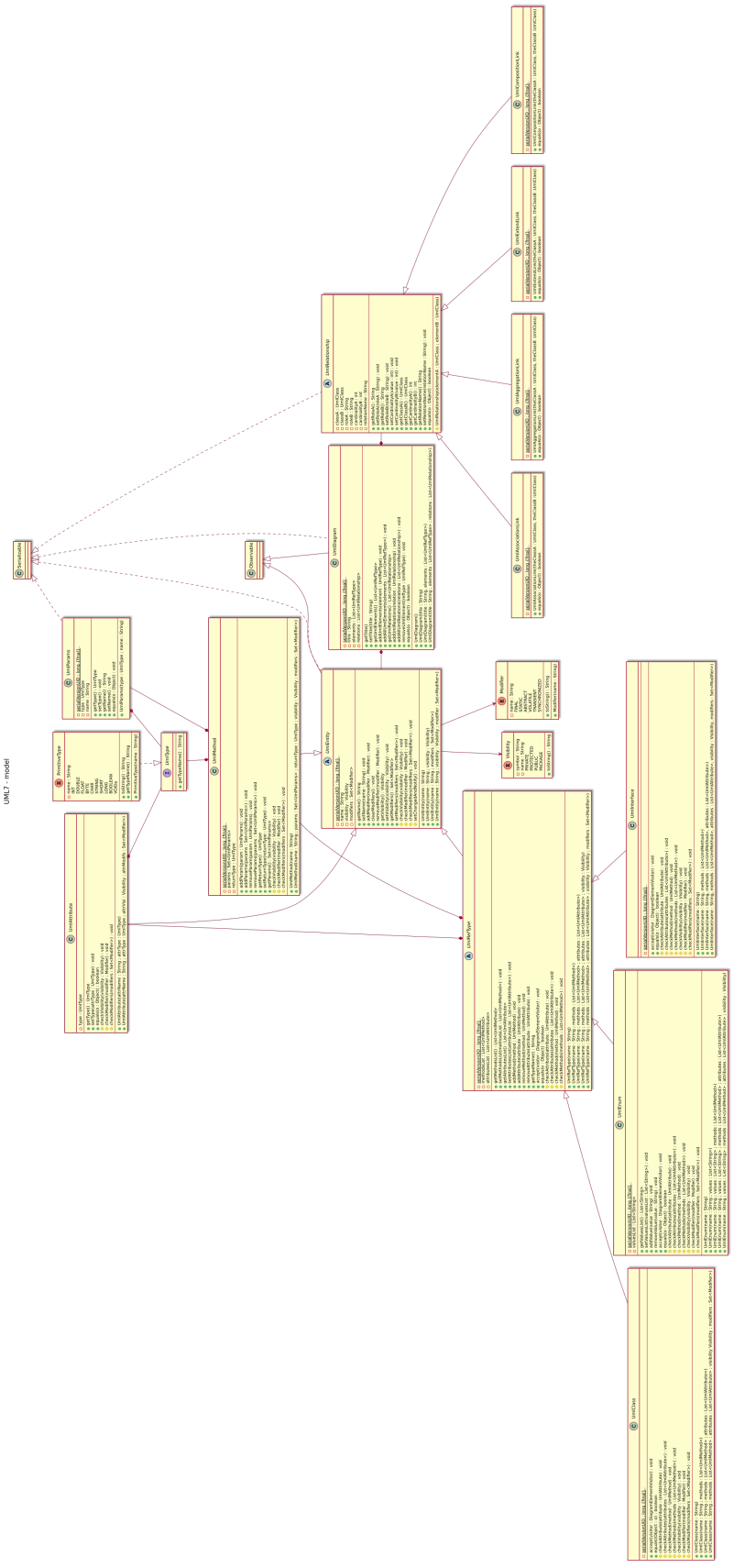
3 Conclusion

Le projet a été très enrichissant pour l'ensemble de l'équipe, aussi bien sur l'aspect humain que l'aspect technique. Il nous a enseigné les risques liés à un tel projet. En effet, différents points peuvent être une difficulté comme le nombre de personnes, l'ambition initiale, la gestion du temps, la provenance des membres (réseau, informatique, ...) ou encore la barrière de la langue.

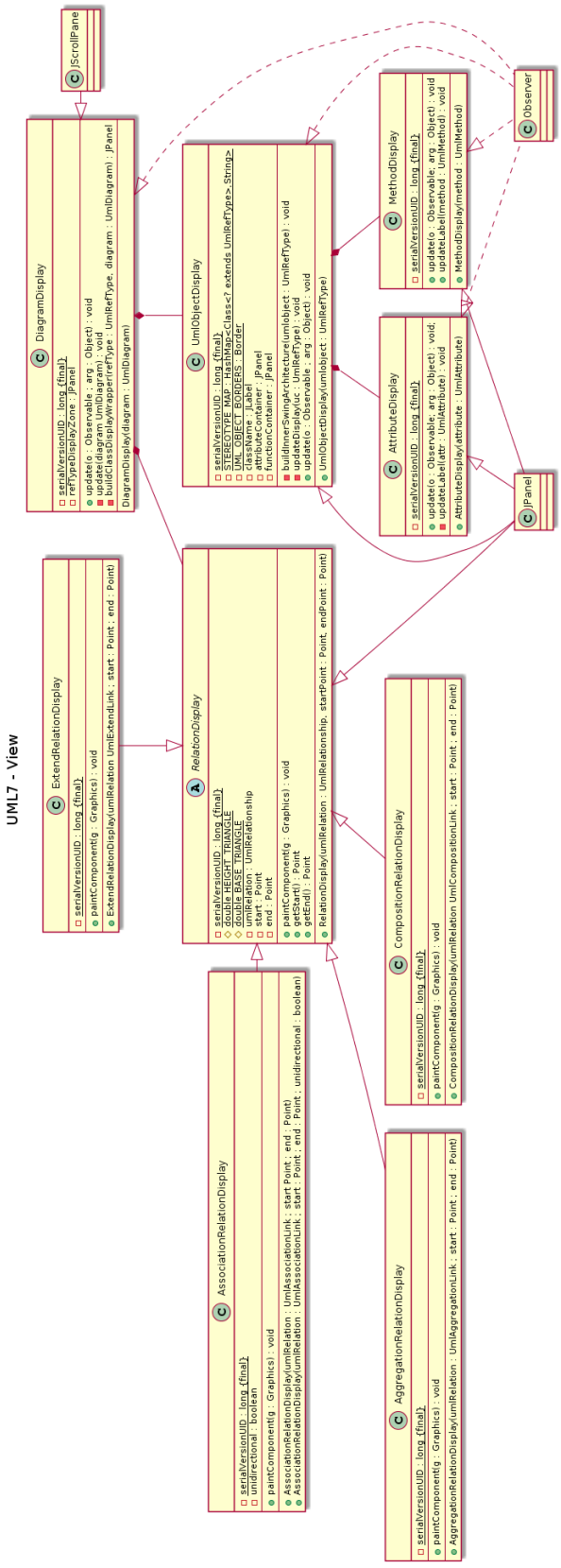
A Annexe 1


UML7
Import diagram
Create diagram from scratch
include
include
Edit diagram
User
extends
Save/Export diagram

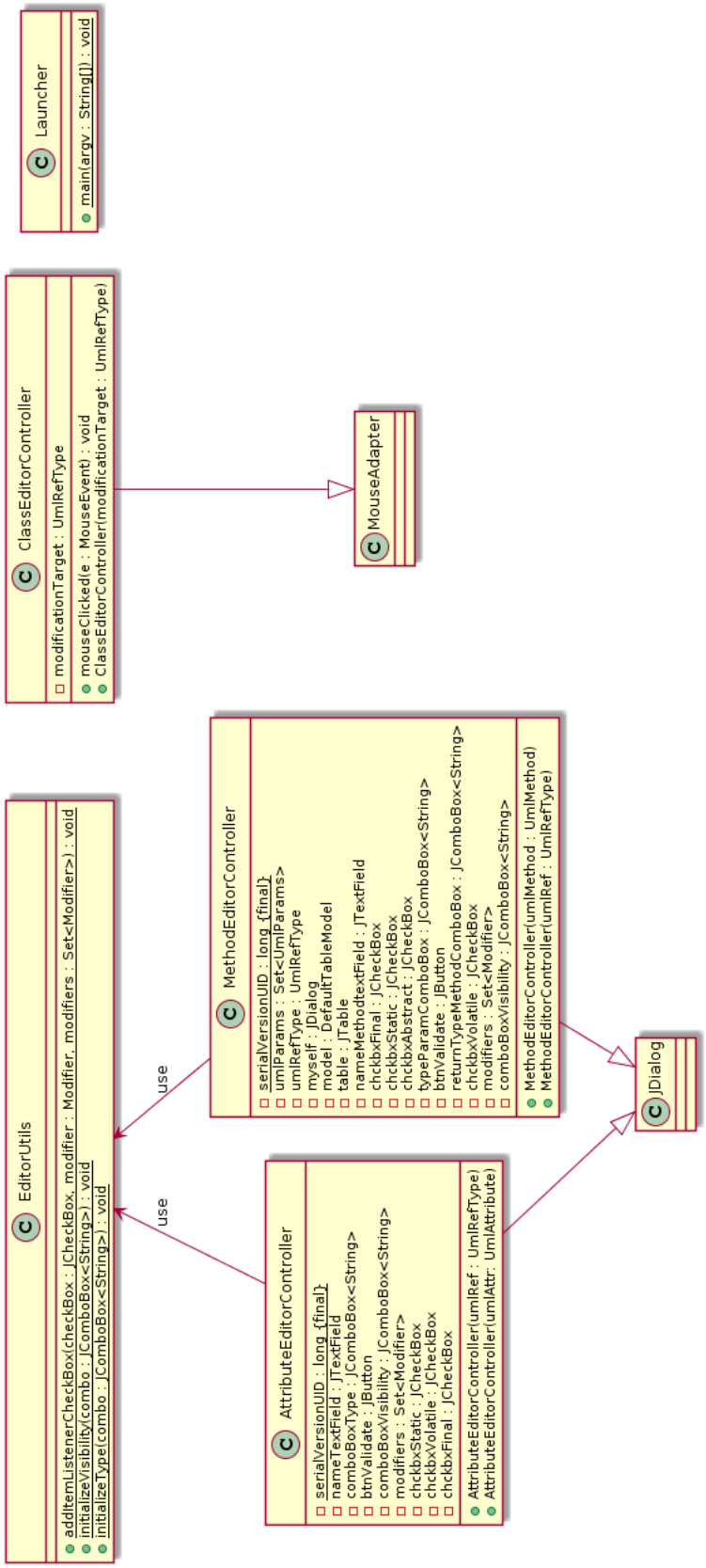
B Annexe 2



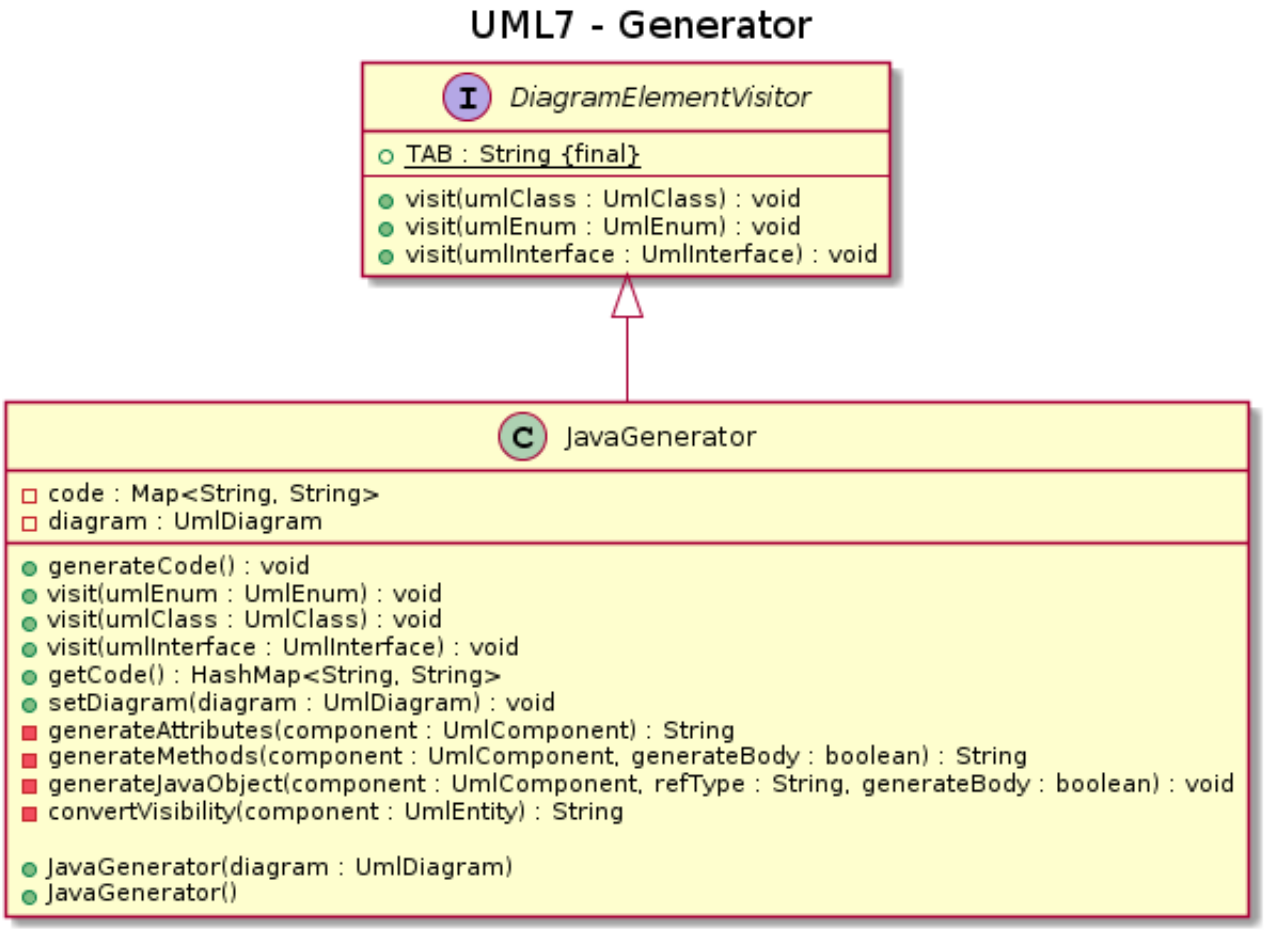
C Annexe 3


UML7 - View
JScrollPane
DiagramDisplay
UmlObjectDisplay
MethodDisplay
Observer
AttributeDisplay
JPanel
ExtendRelationDisplay
RelationDisplay
AssociationRelationDisplay
CompositionRelationDisplay
AggregationRelationDisplay

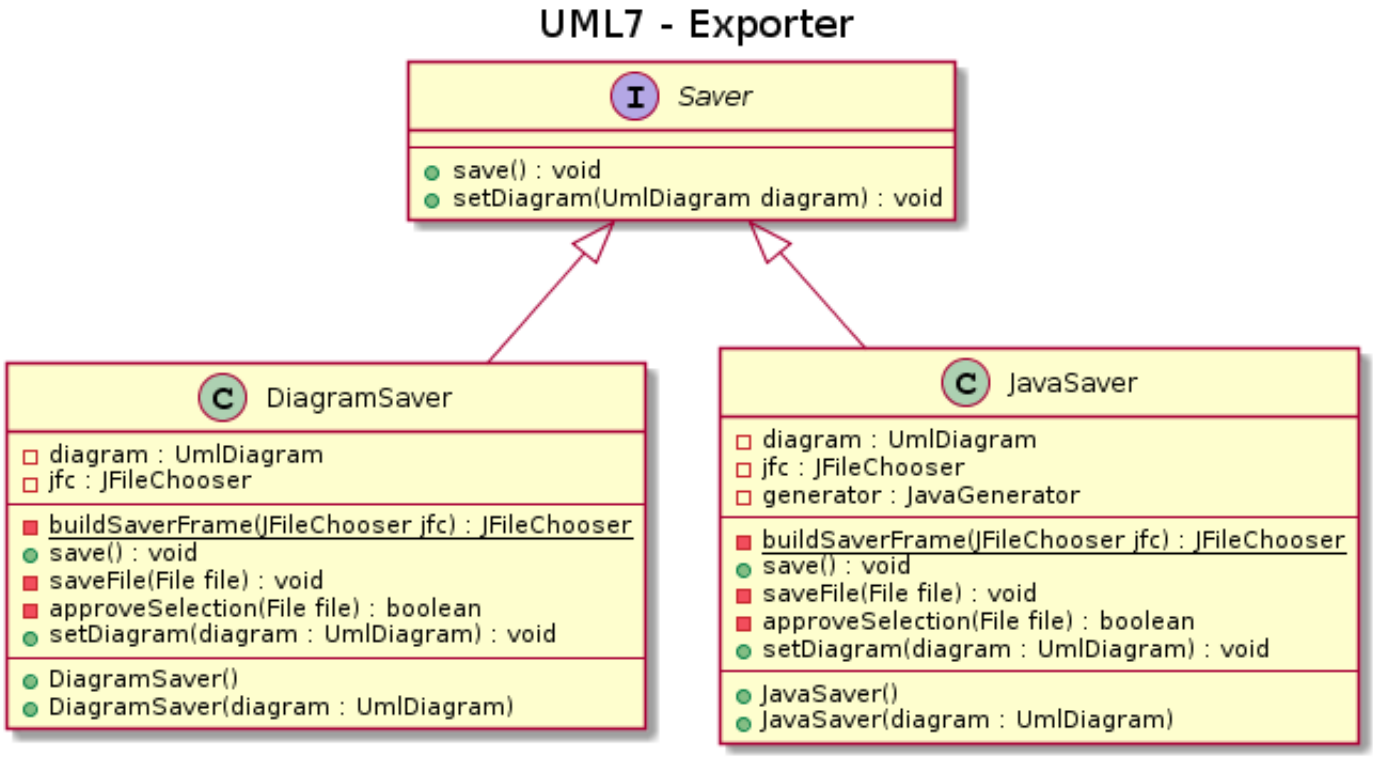
D Annexe 4


Launcher
main(argv : String[]) : void
ClassEditorController
modificationTarget : UmlRefType
mouseClicked(e : MouseEvent) : void
ClassEditorController(modificationTarget : UmlRefType)
MouseAdapter
MethodEditorController
serialVersionUID : long {final}
umlParams : Set<UmlParams>
umlRefType : UmlRefType
myself : JDialog
model : DefaultTableModel
table : JTable
nameMethodtextField : JTextField
chckbxFinal : JCheckBox
chckbxStatic : JCheckBox
chckbxAbstract : JCheckBox
typeParamComboBox : JComboBox<String>
btnValidate : JButton
returnTypeMethodComboBox : JComboBox<String>
chckbxVolatile : JCheckBox
modifiers : Set<Modifier>
comboBoxVisibility : JComboBox<String>
MethodEditorController(umlMethod : UmlMethod)
MethodEditorController(umlRef : UmlRefType)
EditorUtils
addItemListenerCheckBox(checkBox : JCheckBox, modifier : Modifier, modifiers : Set<Modifier>) : void
initializeVisibility(combo : JComboBox<String>) : void
initializeType(combo : JComboBox<String>) : void
use
use
AttributeEditorController
serialVersionUID : long {final}
nameTextField : JTextField
comboBoxType : JComboBox<String>
btnValidate : JButton
comboBoxVisibility : JComboBox<String>
modifiers : Set<Modifier>
chckbxStatic : JCheckBox
chckbxVolatile : JCheckBox
chckbxFinal : JCheckBox
AttributeEditorController(umlRef : UmlRefType)
AttributeEditorController(umlAttr : UmlAttribute)
JDialog

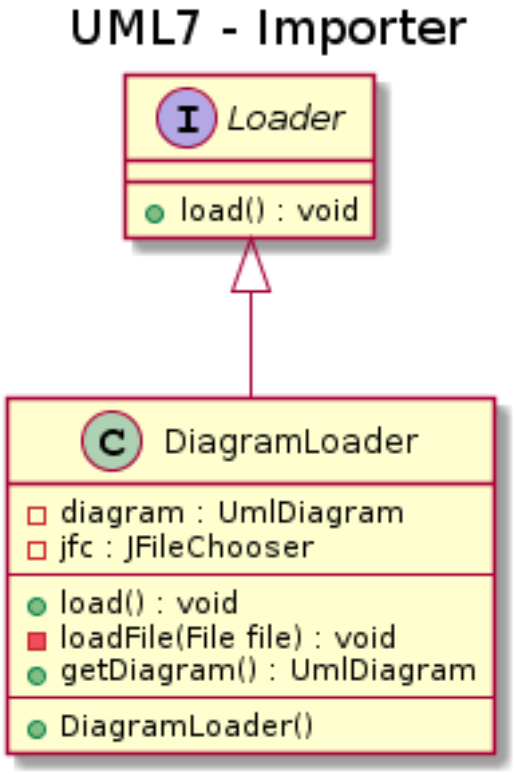
E Annexe 5


UML7 - Generator
DiagramElementVisitor
TAB : String {final}
visit(umlClass : UmlClass) : void
visit(umlEnum : UmlEnum) : void
visit(umlInterface : UmlInterface) : void
JavaGenerator
code : Map<String, String>
diagram : UmlDiagram
generateCode() : void
visit(umlEnum : UmlEnum) : void
visit(umlClass : UmlClass) : void
visit(umlInterface : UmlInterface) : void
getCode() : HashMap<String, String>
setDiagram(diagram : UmlDiagram) : void
generateAttributes(component : UmlComponent) : String
generateMethods(component : UmlComponent, generateBody : boolean) : String
generateJavaObject(component : UmlComponent, refType : String, generateBody : boolean) : void
convertVisibility(component : UmlEntity) : String
JavaGenerator(diagram : UmlDiagram)
JavaGenerator()

F Annexe 6


UML7 - Exporter
Saver
save() : void
setDiagram(UmlDiagram diagram) : void
DiagramSaver
diagram : UmlDiagram
jfc : JFileChooser
buildSaverFrame(JFileChooser jfc) : JFileChooser
save() : void
saveFile(File file) : void
approveSelection(File file) : boolean
setDiagram(diagram : UmlDiagram) : void
DiagramSaver()
DiagramSaver(diagram : UmlDiagram)
JavaSaver
diagram : UmlDiagram
jfc : JFileChooser
generator : JavaGenerator
buildSaverFrame(JFileChooser jfc) : JFileChooser
save() : void
saveFile(File file) : void
approveSelection(File file) : boolean
setDiagram(diagram : UmlDiagram) : void
JavaSaver()
JavaSaver(diagram : UmlDiagram)

G Annexe 6


UML7 - Importer
Loader
load() : void
DiagramLoader
diagram : UmlDiagram
jfc : JFileChooser
load() : void
loadFile(File file) : void
getDiagram() : UmlDiagram
DiagramLoader()