Mini Projet Java - Canv'Us

Simon Delecourt, Quentin Loiseau, Edouard DONZÉ, Louis BAGOT

21 décembre 2017

A l'attention de M. Vercouter

Table des matières

Introduction

Ce projet de Programmation Orientée Objet se réalisait en groupe de 4 personnes et avait pour but d'utiliser les notions apprises en cours concernant les réseaux et les interfaces graphiques, à savoir le langage Java, les réseaux de type RMI ainsi que les interfaces graphiques Swing.

Une fois notre groupe constitué, nous avons alors commencé à réfléchir à ce que nous voulions faire. Notre choix s'est alors porté vers une toile collaborative. L'idée s'inspire d'un chat (gestion de groupe, de membre ...) où le dialogue serait remplacé par des dessins.

L'application a pour but de permettre à différents utilisateurs de dessiner sur une même toile, d'où l'idée de "Toile collaborative". Afin d'en obtenir un nom un peu plus accrocheur, nous avons pensé à l'appeler "Canv'Us", mixe entre "Canvas" (toile en anglais) et "Us" qui évoque le partage et la collaboration.

1 Description du projet

Avant de se lancer dans le développement d'un projet (informatique ou non), il est important de définir en préliminaires des buts permettant de ne pas s'éparpiller. C'est pourquoi nous avons développé un cahier des charges et une maquette permettant de nous guider par la suite.

1.1 Cahier des charges

Comme il peut y avoir une multitude d'approches à la création d'une telle application, nous avons fait un point sur ce que nous en attendions. Il en est ressorti plusieurs axes différents qui semblaient importants pour la rendre intéressante à développer (car amusante à utiliser) :

- ▷ Pouvoir inscrire un utilisateur et éventuellement lui associer une unique couleur
- ▷ Une fois inscrit, pouvoir se reconnecter grace à un système gérant les membres inscrits
- ▷ Arriver à chaque connexion sur une toile "Public" commune à tout les membres, destinée à devenir un chaos de dessins superposés
- ▷ Pouvoir créer des groupes et y ajouter des membres pour qu'un groupe d'amis ait sa (ou ses) propre(s) toile(s).
- ▷ Avoir à disposition une "boîte à outil" de dessin où il serait possible par exemple de changer la taille du curseur, dessiner des formes géométriques définies ou des aides au dessin.
- ▷ Pouvoir visualiser la toile du groupe en temps réel
- ▷ Importer ou exporter des images, et que ces dernières soient sauvegardées sur le réseau.

1.2 Maquette IHM

Cette application, de par sa nature, était vouée à être très portée sur le visuel. C'est pourquoi nous avons décidé de se mettre d'accord, avant toute ligne de code, sur l'apparence que l'application devrait avoir. Nous avons décidé d'une disposition des différents composants, qui, finalement, a été reportée sur une maquette IHM que voici :

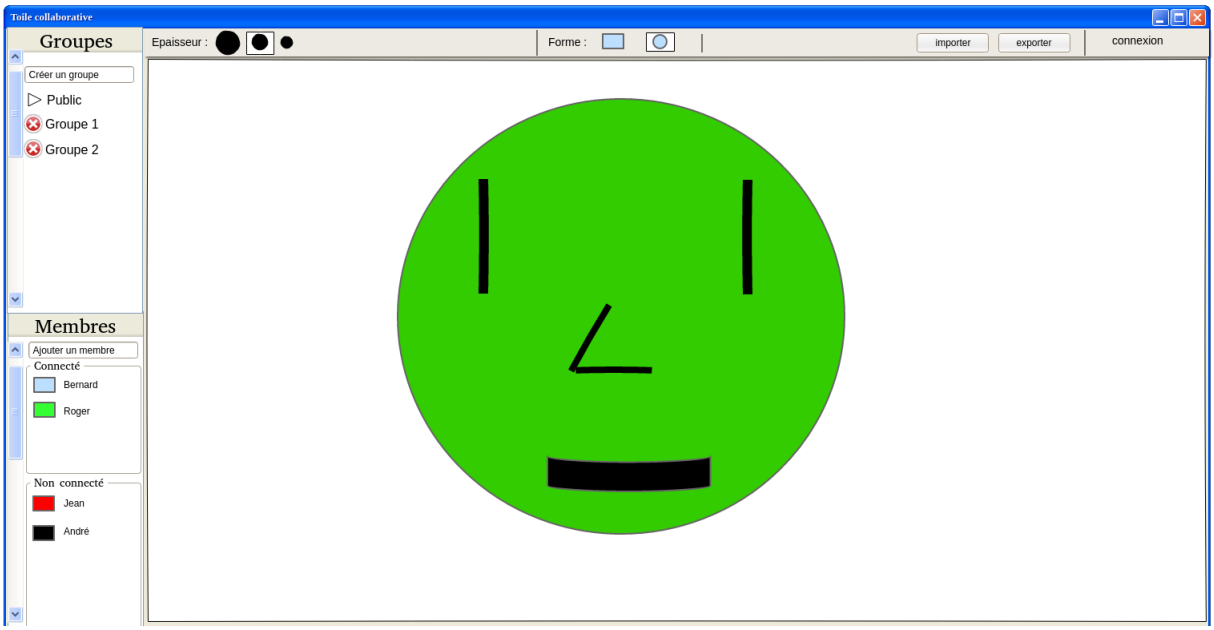

FIGURE 1 – Maquette IHM de l'application Canv'Us

Nous avons donc décidé de placer la boîte à outils en haut de la fenêtre, comme sur plusieurs logiciels de dessin.

Sur la gauche se trouvent les deux 'menus' qui concernent les membres et les groupes :

- ▷ la liste des groupes auquel appartient le membre avec lequel s'est connecté le client
- ▷ la liste des membres du groupe selectionné
- ▷ Deux zones au dessus de chaque liste, permettant respectivement de créer un nouveau groupe, et d'ajouter un membre au groupe selectionné.

Dans le cadre principal se trouve la toile du groupe, sur laquelle l'utilisateur peut dessiner.

2 Modélisation

La Programmation Orientée Objet permet une division claire et instinctive du code, mais il devient alors crucial de définir à l'avance les différents packages et classes qui constitueront le code. La modélisation UML permet de construire un modèle propre du concept de manière à ce que la logique et la lisibilité du code soient les meilleures possibles.

Il nous est cependant important de préciser que ces diagrammes ont inévitablement beaucoup évolué au cours de la programmation, et que ceux que nous présenteront sont les versions finales de ces derniers.

2.1 Fonctionnement général

(diagramme cas d'utilisation)

Définissons d'abord les grandes lignes de l'application. Lorsque le client lance l'application, nous avions d'abord prévu que le client puisse voir et dessiner sur la toile du groupe public avant même de s'authentifier ; cependant l'idée nous est ensuite venue d'associer à chaque membre une couleur unique, qui permettrait de deviner le membre responsable d'une tracé donné.

Le schéma de base a donc légèrement été modifié : au lancement de l'application, l'utilisateur (*client*) va d'abord s'authentifier, et la fenêtre définie plus tôt va ensuite s'afficher en fonction du *membre* que le client a choisi. A partir d'ici, l'utilisateur pourra interagir avec la fenêtre - en dessinant ou utilisant les menus à sa disposition.

Ceci peut être modélisé en UML par un *diagramme des cas d'utilisation* dont la lecture est intuitive :

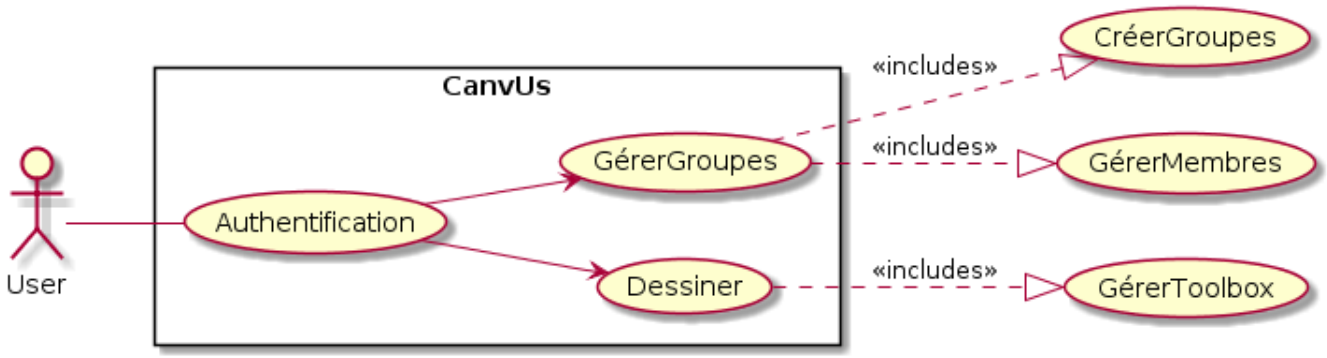

FIGURE 2 – Diagramme des cas d'utilisation de l'application Canv'Us

2.2 Modèle du réseau

L'application nécessitait, de toute évidence, une mise en réseau, et pour ce faire nous avons utilisé l'architecture RMI. Afin d'avoir une architecture structurée du réseau, nous avons décidé de distinguer les différents serveurs :

- ▷ Un serveur principal gérant le déroulement de l'application (appelé donc *ServerApp*). Il gère les fichiers de sauvergarde et l'authentification, mais surtout lie les clients avec leurs groupes :
- ▷ Des serveurs secondaires gérant chaque groupe (*ServerGroup*)
- ▷ Un serveur pour chaque utilisateur (*ServerUser*)

Ceci peut-être modélisé par un *diagramme de déploiement* en ULM :

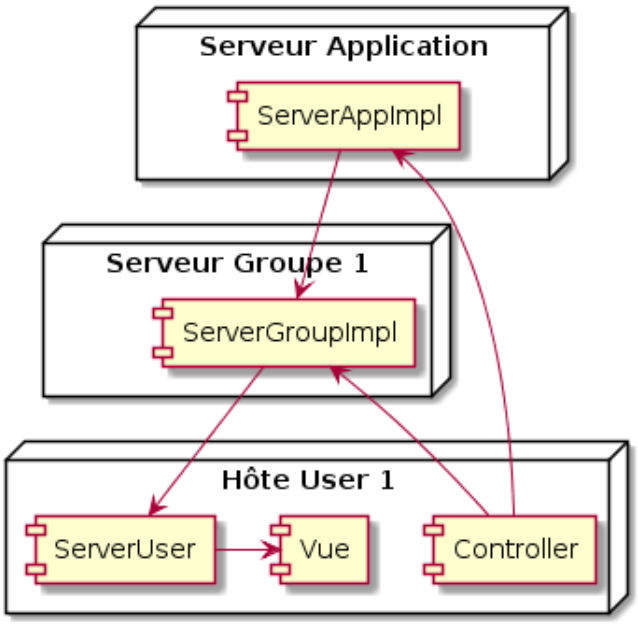

FIGURE 3 – Diagramme de déploiement pour un client

Nous avons d'abord hésité à ne tout gérer qu'avec un seul serveur, qui manipulerait une liste de groupes. Cependant, notre volonté ici était de distinguer les serveurs de groupe du serveur de l'application pour une meilleure gestion. En effet, cela permet de désolidariser le serveur principal de chaque groupe, ce qui implique qu'il ne sera pas impacté par un problème venant d'un groupe en particulier (crash de server, ordre de dessin trop lourd...).

Les communications entre serveurs sont très simples :

- ▷ Les User sont chacun connectés au ServerApp dès leur connexion.
- ▷ Les User ont une connexion à chacun des ServerGroup auxquels ils appartiennent.
- ▷ Le ServerApp communique avec chacun des ServerGroup actuellement en ligne.

Nous pouvons alors en déduire un second *diagramme de déploiement* pour visualiser la distribution à grande échelle :

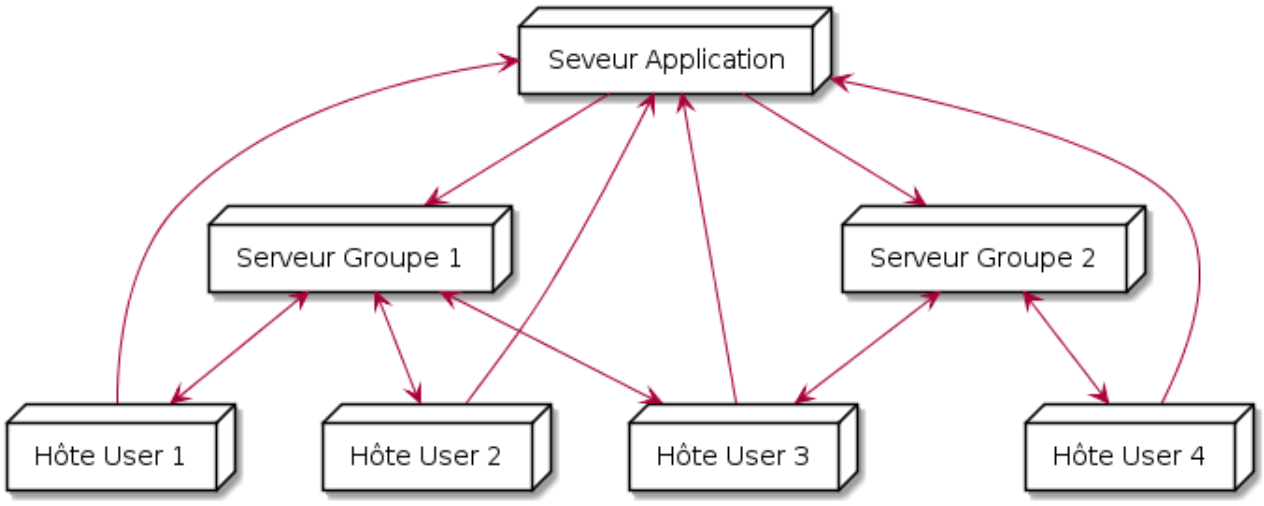

FIGURE 4 – Diagramme de déploiement général

2.3 Conception de l'application

Afin d'assurer la lisibilité et l'adaptabilité du code, nous avons décidé d'utiliser le schéma MVC (Modèle-Vue-Contrôleur) vu en cours. Il permet de dissocier clairement ces trois tâches centrales d'une application, qui interagissent de manière bien définie. Nous avons donc séparé notre code en trois packages portant le nom de chaque partie du modèle :

- ▷ Le package "Model", qui contient toutes les classes gérant la logique métier de l'application. Ceci recouvre les objets Membre, Groupe, Canvas...
- ▷ Le package "Controller", qui est uniquement constitué de classes Listener qui ont pour rôle d'attendre les actions du client et de les relayer aux classes du Modèle concernées
- ▷ Le package "View", qui comporte toutes les classes liées à l'affichage de données sur la fenêtre du client.

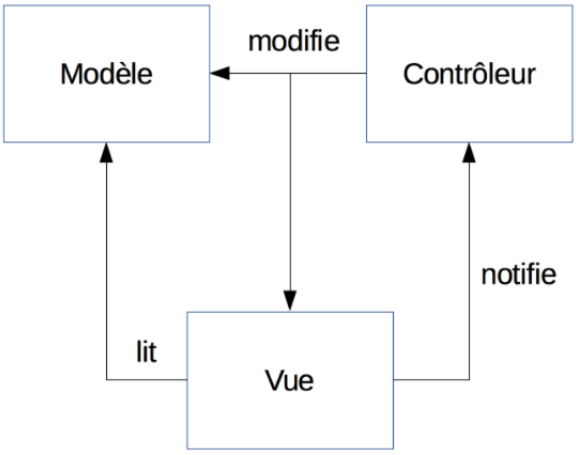

FIGURE 5 – Modèle MVC

Nous avons alors greffé à ce modèle l'idée de réseau, que nous avons modélisé par le package "server", composé principalement de classes évoquées plus tôt et implémentant l'interface RMI Remote.

Chacun de ses packages - et surtout l'ensemble de ses classes - peut alors être représenté sous la forme d'un diagramme UML nommé *diagramme de classes* et qui permet de visualiser les éléments de chaque classe du package et les dépendances entre les classes.

Merci de vous référer à la javadoc fournie avec le code pour les détails sur toutes les classes. Ci-dessous, les diagrammes de classe de chacun des packages :

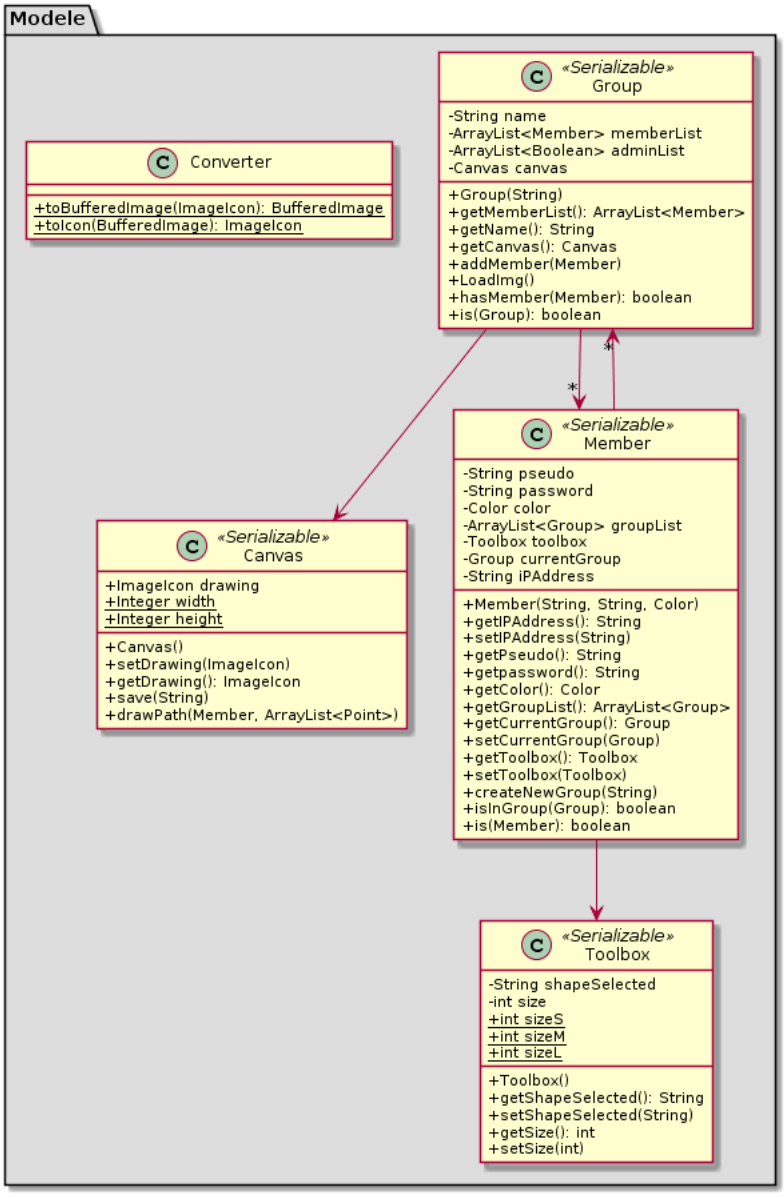

FIGURE 6 – Diagramme de classe du package Model

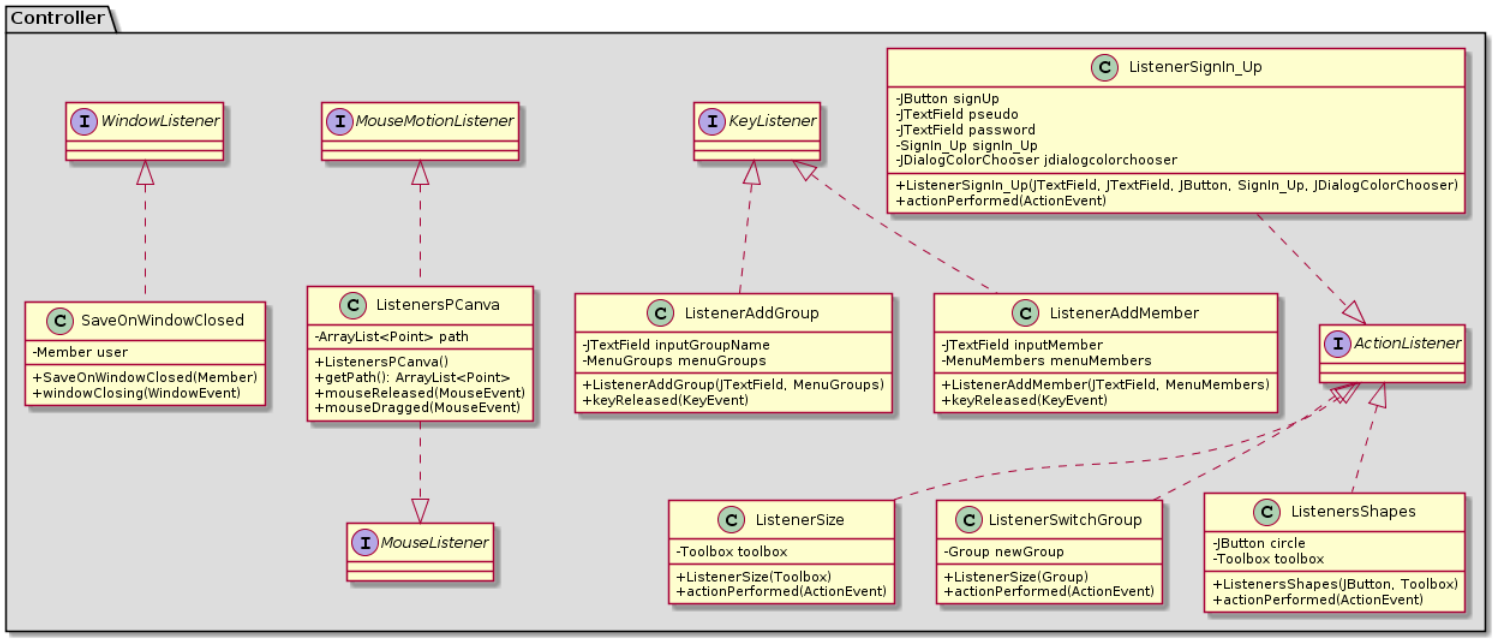

FIGURE 7 – Diagramme de classe du package Controller

Le package Main est très particulier puisqu'il n'existe essentiellement que pour sa méthode main que lance le client (c'est l'action de lancement de l'application) :

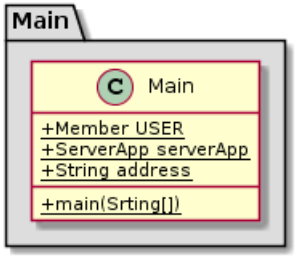

FIGURE 8 – Diagramme de classe du package Main

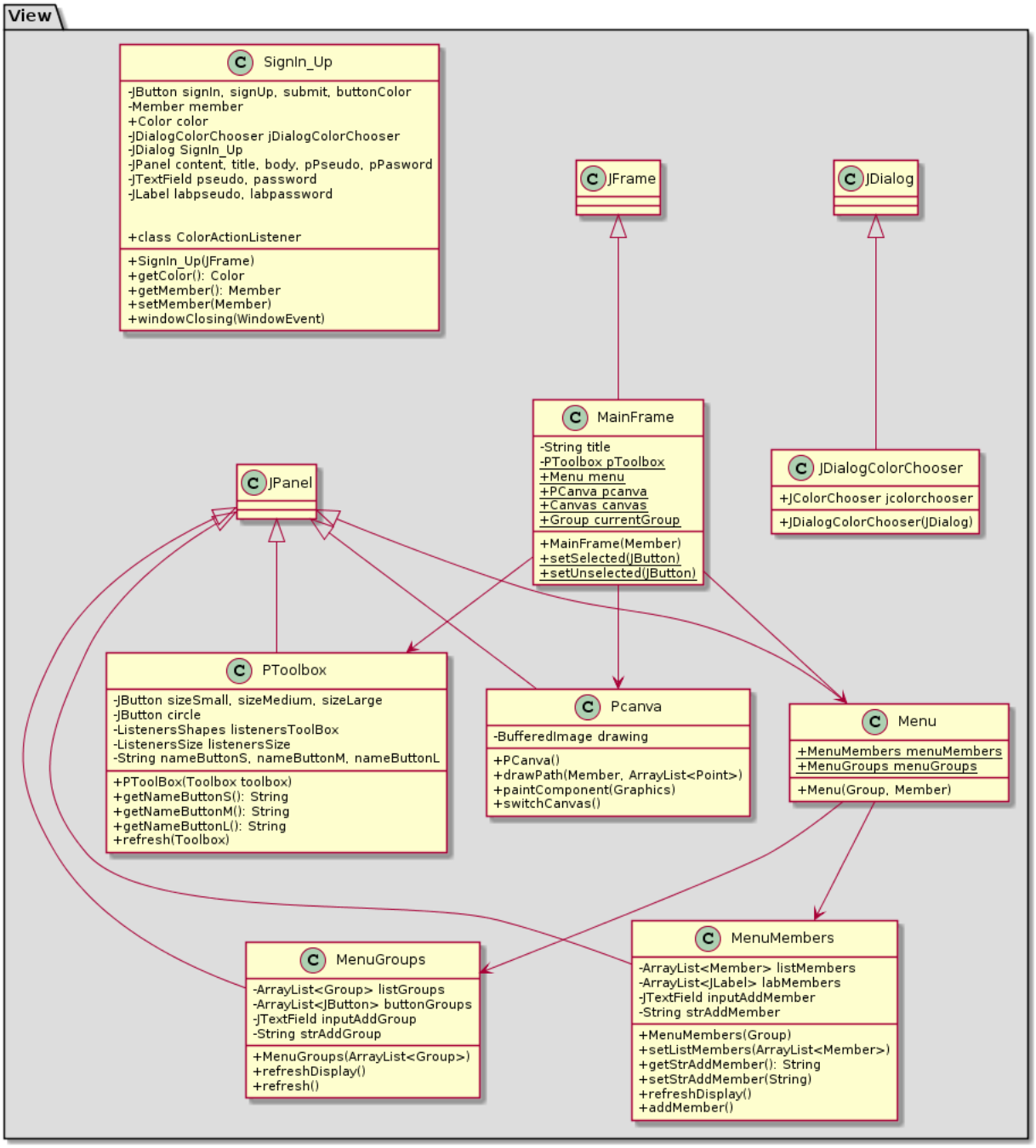

FIGURE 9 – Diagramme de classe du package View

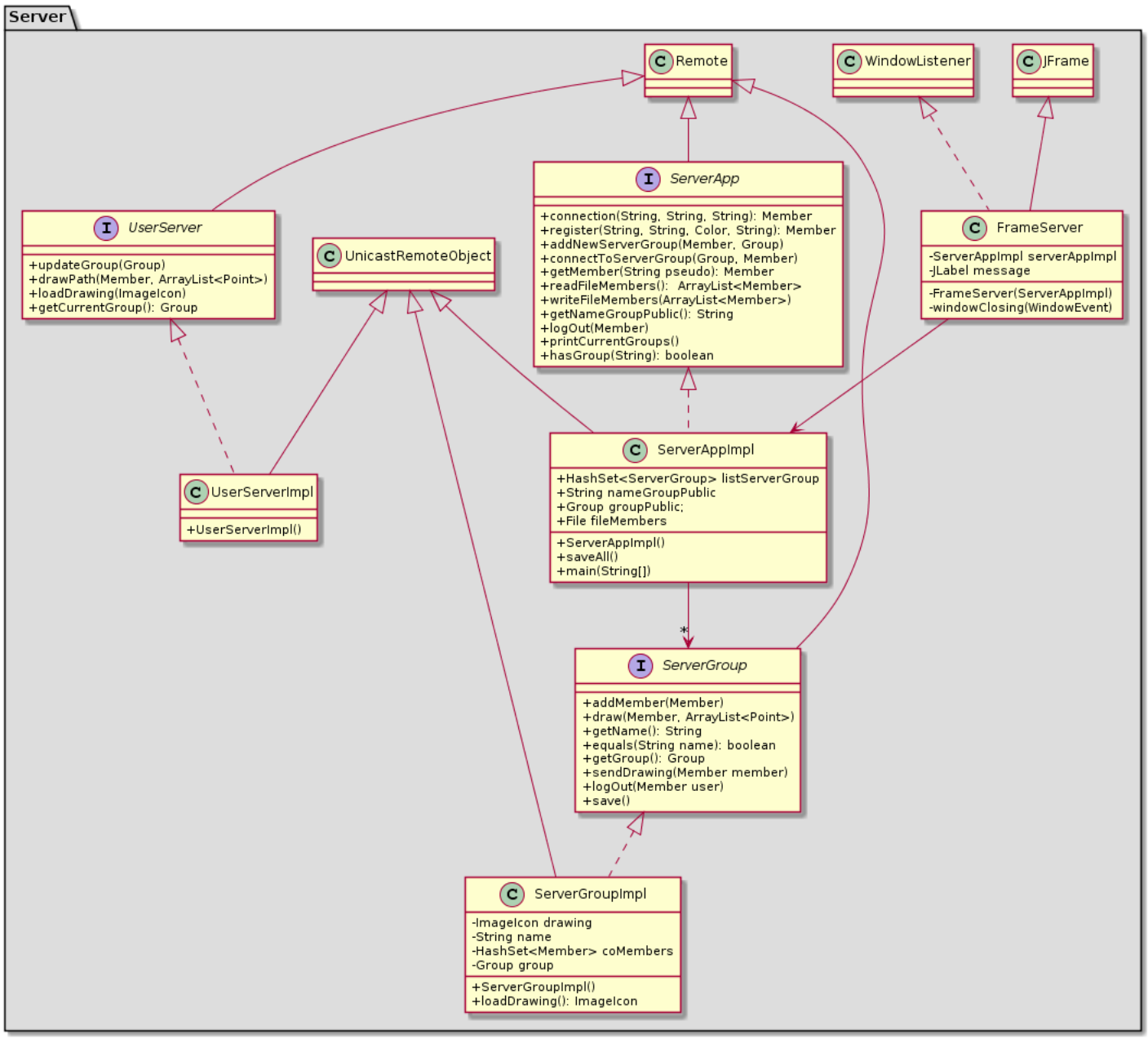

FIGURE 10 – Diagramme de classe du package server

Toutes ces classes organisées dans ces packages sont évidemment destinées à communiquer entre elles et dépendre les unes des autres ; ceci peut-être visualisé à l'aide du même diagramme, mais

généralisé sur l'ensemble du code :

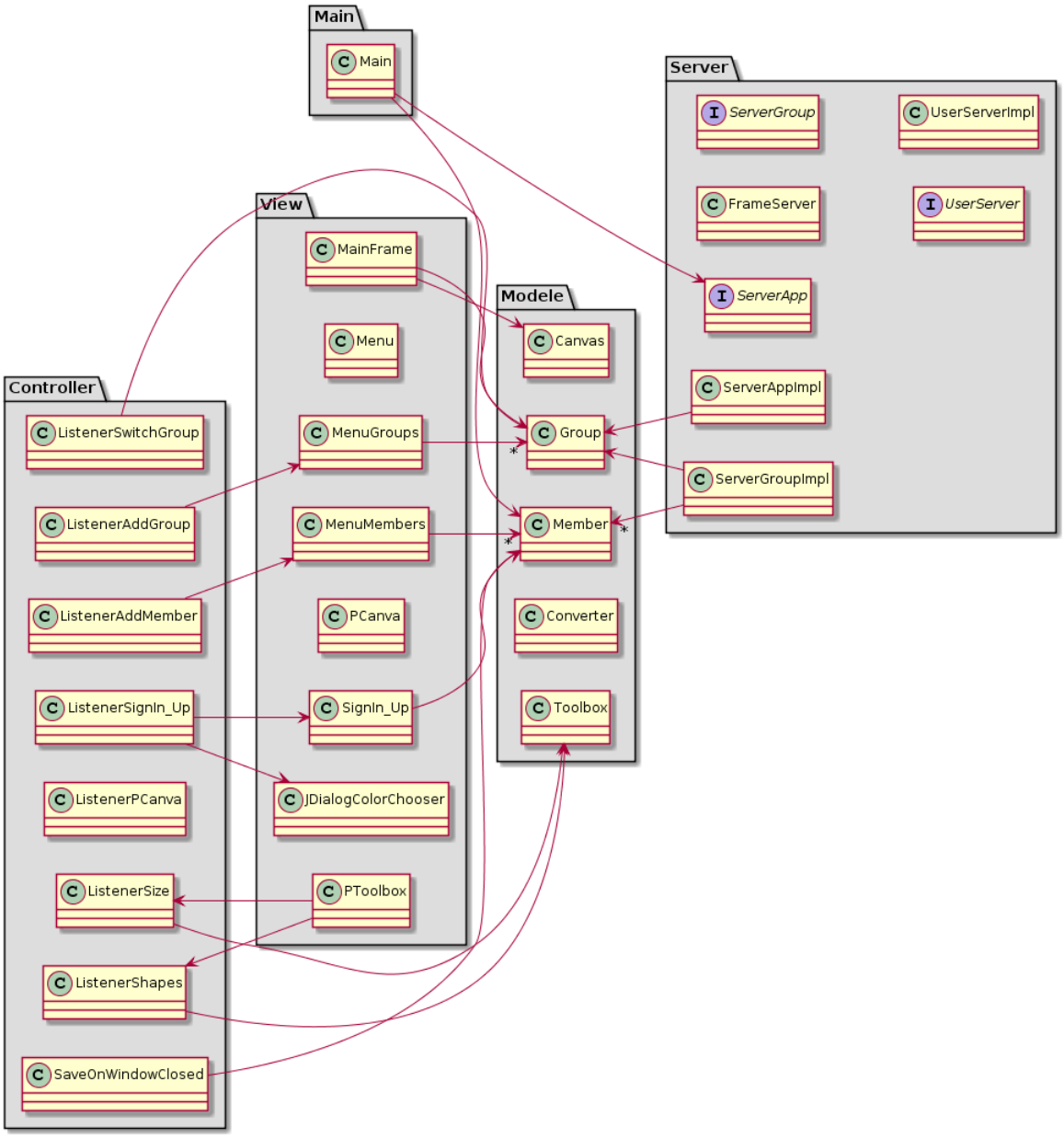

FIGURE 11 – Diagramme de classe du package server

2.4 Déroulement des opérations

Les diagrammes de séquence permettent d'expliciter comment s'éxecute le code une fois un cas d'utilisation débuté. Nous avons décidé de concentrer nos diagrammes de séquence sur les deux cas les plus compliqués qu'offrait l'application :

- ▷ Celui concernant l'authentification d'un utilisateur (inscription ou connexion).
- ▷ Celui concernant un ordre de dessin envoyé par un utilisateur (et tout ce qui en découle).

2.4.1 Authentification :

Une fois que l'utilisateur lance l'application, le Main commence et la fenêtre d'authentification SignIn_Up s'affiche. En fonction des actions de l'utilisateur, il est possible de procéder soit à une connexion (SignIn), soit à une inscription (SignUp).

Une fois cette requête effectuée auprès du ServerApp, une connexion est effectuée pour chaque ServerGroup auquel appartient le membre. Dans le cas d'une inscription, le membre est automatiquement ajouté et connecté au groupe "*Public*".

Une fois cette connexion entre les serveurs établie, un petit message précise en console si l'opération s'est bien déroulée et la MainFrame de l'application est démarée. Celle-ci est initialisée et le membre est envoyé par défaut sur le groupe "Public".

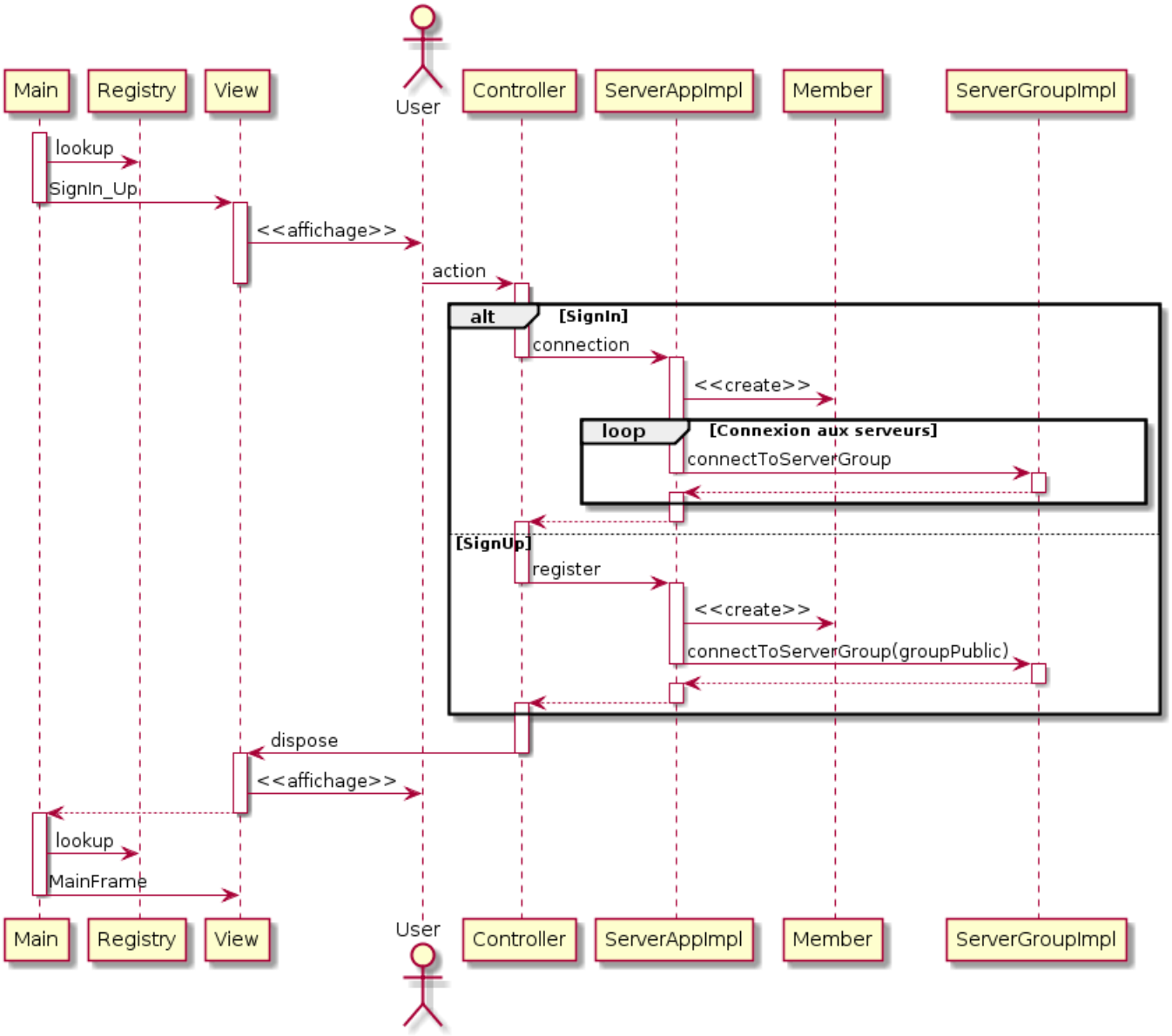

FIGURE 12 – Diagramme de séquence de l'authentification

Comme nous venons de le dire, une autre étape clé sur laquelle nous avons décidé de se concentrer est la partie dessin.

Lorsque l'utilisateur, connecté à un groupe, dessine sur son Canvas, un ordre de dessin (draw) est lancé au serveur de ce groupe. Ce même ordre est alors lancé au Canvas du groupe lié à ce serveur, pour que ce dernier contienne toutes les modifications sans trop d'échange d'information brute (Canvas entier). Cette premiere étape est nécessaire pour que l'utilisateur, lorqu'il rejoint le group ou se connecte

dessus, recupère le Canvas en cours de ce groupe.

Dans une deuxième étape, cet ordre de dessin est transféré à tous les membres connectés (i.e. qui utilisent actuellement l'application). Toutefois une distinction est faite pour ces membres : est-il présent actuellement sur ce groupe ou non ? Si oui, l'ordre de dessin est envoyé au serveur de l'utilisateur pour que la figure s'affiche sur son Canvas, et dans l'autre cas, une notification est envoyée (non implémentée dans la version actuelle).

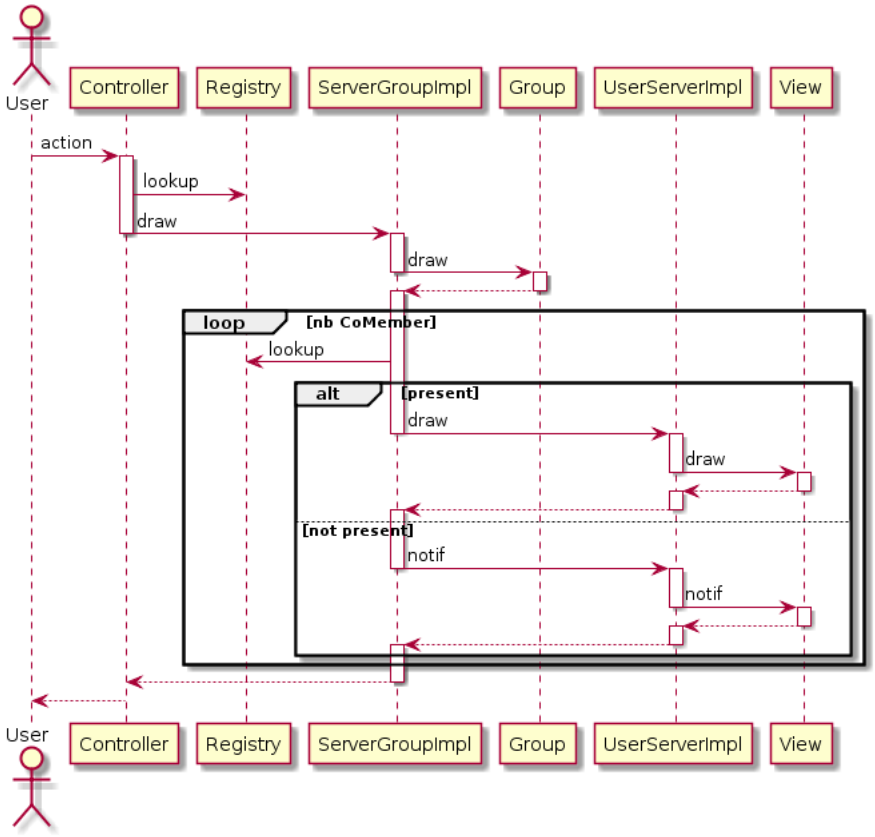

FIGURE 13 – Diagramme de séquence de l'ordre de dessin

3 Résultats obtenus et retrospective

3.1 Résultats

Durant ce projet, nous nous sommes principalement focalisés sur l'aspect visuel d'application et surtout tout ce qui était relatif à la gestion des Canvas. C'est pourquoi les méthodes d'ajout de groupe et de membre ne sont pas entièrement opérationnelles.

Voici une capture d'écran de ce que l'on peut obtenir avec deux membres sur la toile Public :

FIGURE 14 – Exemple d'utilisation sur la toile Public

3.2 Difficultés rencontrées

La première difficulté rencontrée a été la modélisation d'une application dans sa globalité. Nous avons aussi réalisé que travailler par équipe de 4 n'est pas évident. Pour cela nous avons mis en place différents outils pour nous aider à rester au courant du travail des autres membres. Tout d'abord pour le code nous avons créé un "repository" sur github. Pour la liste des tâches effectuées, un "board" sur Trello a également été mis à jour tout au long du projet.

Du coté de l'implémentation, la gestion de la base de données nous a causé des problèmes. Nous avons eu recours à un fichier (car cela avait été vu en cours) pour enregistrer la liste des membres et des différents groupes auxquels ils appartiennent. Or, il est parfois nécessaire de connaître la liste des groupes auxquels appartient un membre, et parfois la liste des membres d'un groupe est requise. Ceci n'est pas évident à faire avec une simple lecture de fichier alors que cela aurait été plus direct avec une vraie base de données telle que JDBI.

La gestion des images côté serveur a engendré des erreurs car le type "BufferedImage" n'implémente pas l'interface "Serializable". Ainsi, nous avons du tricher en stockant sur le serveur une image de type "ImageIcon" qui implemente l'interface Serializable. De ce fait, de nombreuses conversions entre ces deux types sont opérées.

Sur plusieurs machines nous avions des erreurs du côté du client. Nous avons réalisé qu'il nous fallait stocker l'adresse IP de tous les clients et ainsi accéder à leur registre via une requête en spécifiant leur adresse IP. Malheureusement, il subsiste des erreurs de ce côté là que nous n'avons pas réussi à résoudre.

3.3 Organisation du travail

Dans un premier temps, nous avons pensé le projet sans la partie serveur car nous ne l'avions pas encore étudié. Nous avons, d'abord, effectué ensemble une première modélisation de notre projet avec

un diagramme de classe, des descriptions succintes de scénarii et maquette IHM. Simon a ensuite codé les différentes classes et leurs méthodes progressivement jusqu'au fonctionnement de l'application en local.

Dans un second temps, nous avons réfléchi ensemble au diagramme de déploiement. Simon a alors modifié son code pour l'adapter à une architecture serveur. Puis nous avons travaillé par équipe de 2 (Simon/Edouard et Louis/Quentin) afin de corriger différentes erreurs qui apparaissaient à l'exécution.

Une fois que le gros de l'application a fonctionné, Edouard s'est occupé de revoir l'IHM afin de la rendre plus attrayante. Quentin a implémenté des fonctions relatives au dessin ("Toolbox"). Quant à Louis et Simon, ils ont continué à ajouter/corriger des fonctionnalités client/serveur.

En parallèle, Quentin et Edouard ont travaillé sur tous les diagrammes UML et Louis s'est occupé de la javadoc. Pour rester consistant avec le code et pour éviter les problèmes d'accent, la javadoc est en anglais.

3.4 Notice d'utilisation

Pour démarrer l'application sur plusieurs machines distantes en réseau, nous avons utilisé un serveur local lancé par UwAmp où sont stockés les .class des différents classes de l'application. Il faut, ensuite, lancer un rmiregistry sur toutes les machines. Puis exécuter la classe ServerApplmpl sur une machine "serveur". Les autres machines peuvent alors lancer la classe Main en indiquant l'adresse "http ://localhost/classes" et l'adresse IP de la machine serveur.

Sur une seule machine, il suffit d'exécuter ServerAppImpl puis Main.

4 Conclusion

Ce projet en Java est arrivé à point nommé pour nous aider à assimiler les éléments appris en cours de Conception et Programmation Orientée-Objet. Pour chacun de nous, ce fut le premier projet à devoir être réalisé en réseau et un des seuls où nous devions concentrer notre attention sur l'IHM selon un schéma Modèle-Vue-Contrôleur.

Si notre idée initiale de l'application était (probablement) un peu ambitieuse à réaliser de bout en bout, nous sommes assez satisfaits d'avoir rempli une bonne partie du cahier des charges. Il faut noter que c'était un projet conséquent et que chacun a dû y investir un temps non-négligeable.

Si quelques fonctionnalités manquent encore (notification d'un changement auprès de l'utilisateur, possibilité d'importer/exporter une image...), l'application marche globalement selon les critères posés dès le début. L'IHM a elle aussi légèrement évolué (en comparaison avec notre maquette de début) vers quelque chose de plus poussé.

Ce qui est satisfaisant avec la réalisation de ce projet est que nous avons tous eu le sentiment d'être réellement investi, pour produire un livrable répondant aux exigeances et dont les améliorations peuvent être nombreuses. On peut évidemment penser à l'implémentation des fonctionnalités inexistantes, mais nous avons également pensé à renforcer la partie sécurité des informations, où encore utiliser une réelle base de données à la place de notre fichier de données.