Institut national des sciences appliquées de Rouen

INSA de Rouen

Projet POOA GM4

Réalisation d'un Tricount en java

Auteurs :
Thibaut André-Gallis
thibaut.andregallis@insa-rouen.fr
Kévin Gatel
kevin.gatel@insa-rouen.fr
Hugo Merelle
hugo.merelle@insa-rouen.fr
Marion Mesnil
marion.mesnil@insa-rouen.fr

Enseignants :
Cecilia Zanni-Merk
cecilia.zanni-merk@insa-rouen.fr

4 janvier 2022

Table des matières

Table des figures

Introduction

Tricount est une application qui permet d'organiser les dépenses de groupe. Ses utilisations sont multiples, elle permet notamment la répartition équitable des dépenses du groupe et le fait de pouvoir rembourser facilement quelqu'un si besoin. Elle peut également faire l'objet d'un récapitulatif de compte.

Dans ce projet on se propose de réaliser un Tricount à notre échelle, c'est -à -dire une application "locale" entre plusieurs clients (6 maximum) et un serveur. Le terme local est utilisé dans le sens où les clients peuvent se connecter au serveur s'ils partagent le même réseau.

L'utilisateur qui lance le serveur est appelé administrateur. Lui et lui seul a le droit d'ouvrir et/ou fermer l'application lorsqu'il le souhaite. Les autres utilisateurs ne peuvent se connecter à l'application que lorsque le serveur est ouvert au préalable par l'administrateur.

Lorsque le serveur est fermé toutes les données sont perdues. Nous ne stockons pas les données en local, tout se fait dans le serveur lorsqu'il est ouvert.

Dans un premier temps, nous allons voir différents diagrammes afin de clarifier la manière d'aborder le sujet. Puis dans un second temps, nous allons implémenter l'application en Java et commenter les éventuelles difficultées rencontrées.

1. UML Spécification

1.1 Cas d'utilisation

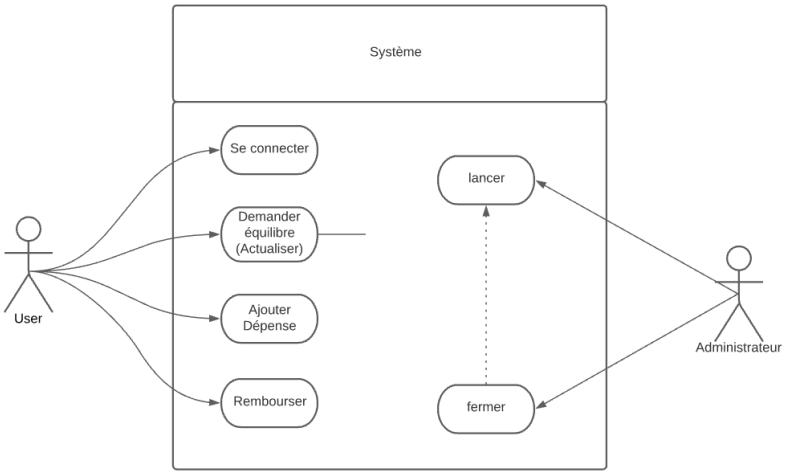

FIGURE 1.1 – Diagramme de cas d'utilisation

Dans ce diagramme on distingue les deux cas suivants : l'administrateur et l'utilisateur de l'application. Le premier permet de lancer l'application puis de l'arrêter, alors que les "users" utilisent l'application suivant quatre options lorsque cette dernière est bien lancée.

1.2 Séquence

Le diagramme de séquence représentant l'option "Se connecter" pour l'utilisateur est représenté sur chacun des trois diagrammes de séquence suivants.

Le diagramme de séquence de l'option "Demander équilibre (Actualiser)" est le suivant :

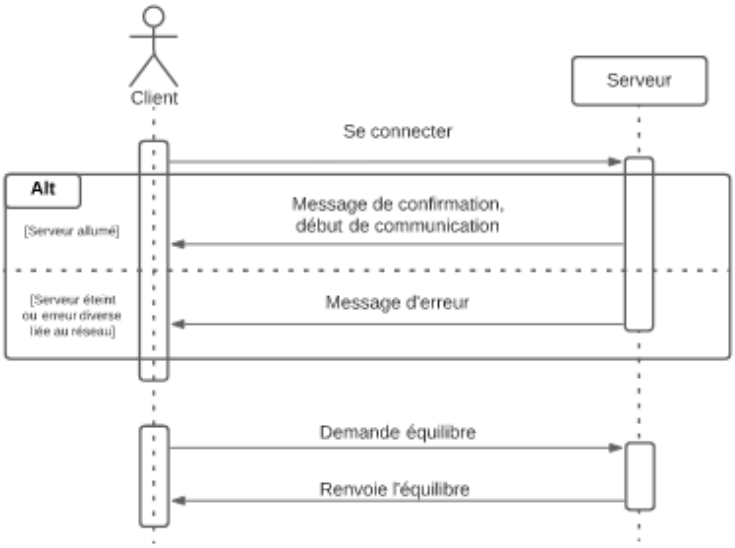

FIGURE 1.2 – Diagramme de séquence "Demander Equilibre (Actualiser)"

Afin de pouvoir “Demander équilibre” au serveur, nous devons nous assurer que l’utilisateur est connecté au serveur. Dans le cas où l’utilisateur est bien connecté, un message de confirmation est envoyé à l’utilisateur et la requête peut commencer à être traitée, dans le cas contraire un message d’erreur est envoyé et la requête est renvoyée. Si l’utilisateur est correctement connecté au Serveur alors le Serveur envoie la liste des utilisateurs de l’application à l’utilisateur en les informant s’ils sont “en avance” ou “en retard” par rapport aux autres utilisateurs selon une somme d’argent indiquée.

Le diagramme de séquence de l’option “Ajouter dépense” est le suivant :

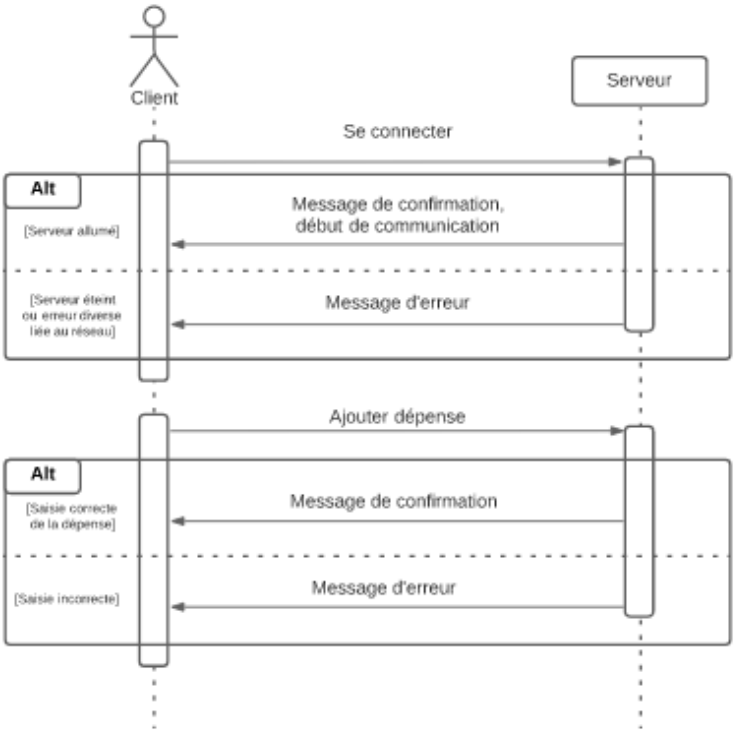

FIGURE 1.3 – Diagramme de séquence "Ajouter dépense"

Même chose que pour “Demander équilibre” l’utilisateur doit être connecté à l’application afin de pouvoir demander cette requête au Serveur. La dépense doit être positive et l’utilisateur doit au moins “cocher” une personne sur les 6 de manière à répartir la dépense équitablement. Le cas où l’utilisateur “coche” uniquement lui-même doit être rejeté. Un message de confirmation est envoyé à l’utilisateur si la requête s’est bien passée, un message d’erreur est retourné dans le cas contraire.

Le diagramme de séquence de l’option “Rembourser” est le suivant :

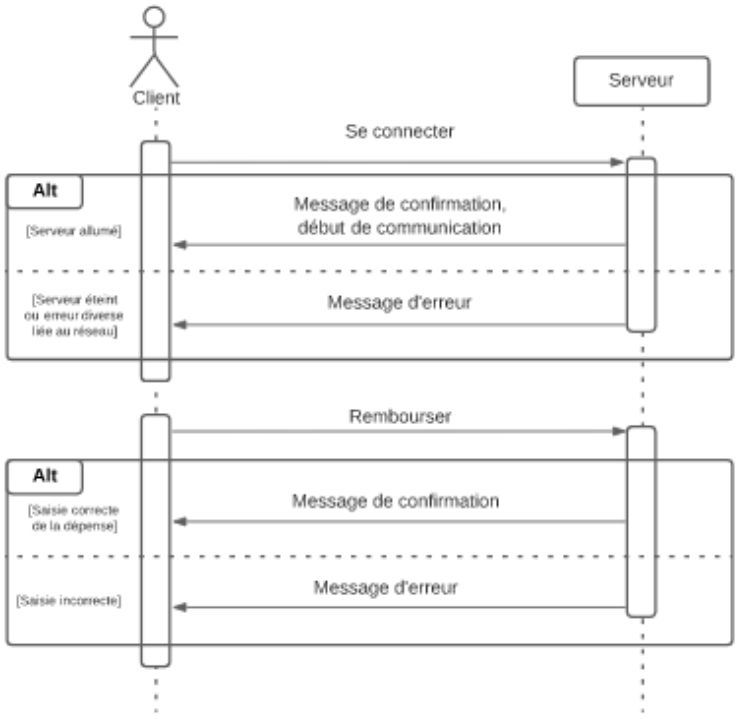

FIGURE 1.4 – Diagramme de séquence "Rembourser"

De la même manière que les précédents, l'utilisateur doit être connecté à l'application afin de pouvoir demander cette requête au Serveur. La somme du remboursement doit être également positive, et l'utilisateur doit "cocher" une et une seule personne (pas lui même) afin de le rembourser. Un message de confirmation est envoyé à l'utilisateur si la requête s'est bien déroulée, un message d'erreur est retourné dans le cas contraire.

1.3 Modèle du domaine

Nous avons réalisé le diagramme de modèle du domaine suivant :

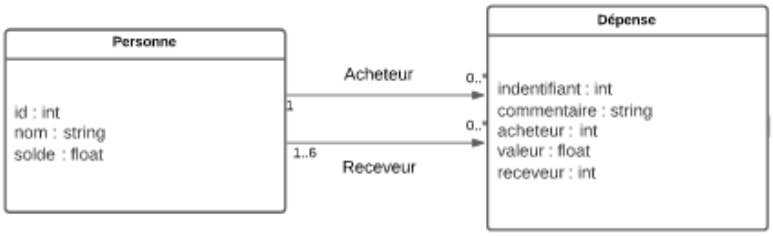

FIGURE 1.5 – Diagramme de modèle du domaine

Comme on peut le voir ci-dessus on peut distinguer 2 classes *Personne* et *Dépense* qui sont liées

entre elles. Commençons d'abord par nous intéresser à chacune d'entre elles :

- *Personne* : La classe Personne représente un utilisateur de l'application tricount. Il va donc posséder deux variables d'instances, son nom qui sera une chaîne de caractères et son solde représenté par un flottant.
- *Dépense* : La classe dépense représente une dépense évidemment mais plus particulièrement une transaction entre deux personnes. Elle est définie par 5 variables : son identifiant qui sera un entier, un commentaire qui décrira à quoi correspond dans la vraie vie cette dépense (ex tickets de bus), une valeur flottante et puis un acheteur et un receveur qui seront des personnes.

Ces deux classes sont liées car une dépense relie deux personnes l'une acheteuse et l'autre receveuse. S'il n'y a pas d'acheteur alors il ne peut y avoir de dépense. Chaque dépense est liée à uniquement un acheteur et un receveur. Cependant un acheteur peut avoir effectué plusieurs dépenses et un receveur en avoir reçu plusieurs également. Il est nécessaire de préciser que dans notre cas nous avons décidé de considérer le receveur comme une seule personne mais nous allons également traiter le cas où une dépense concerne plusieurs receveurs en générant plusieurs dépenses qui posséderont le même identifiant et qui auront une valeur divisée par le nombre de receveurs. Chacune reliera l'acheteur et un des receveurs.

1.4 Maquette de l'application

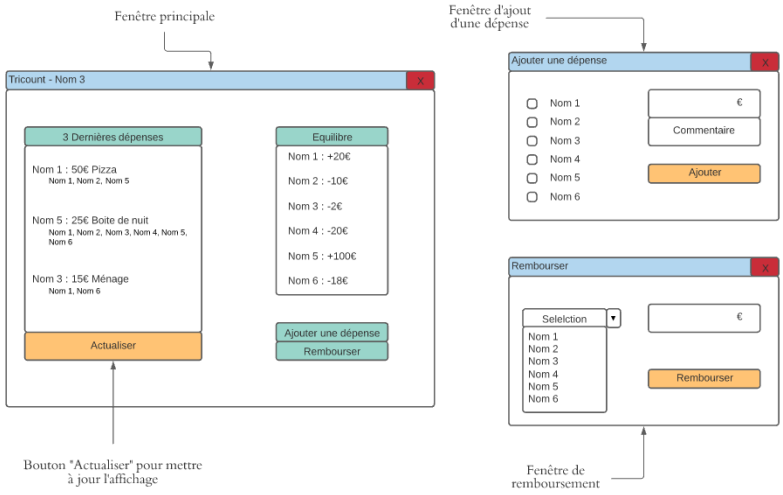

FIGURE 1.6 – Maquette de l'application

Elle peut être expliquée par le diagramme de navigation suivant.

1.5 Navigation

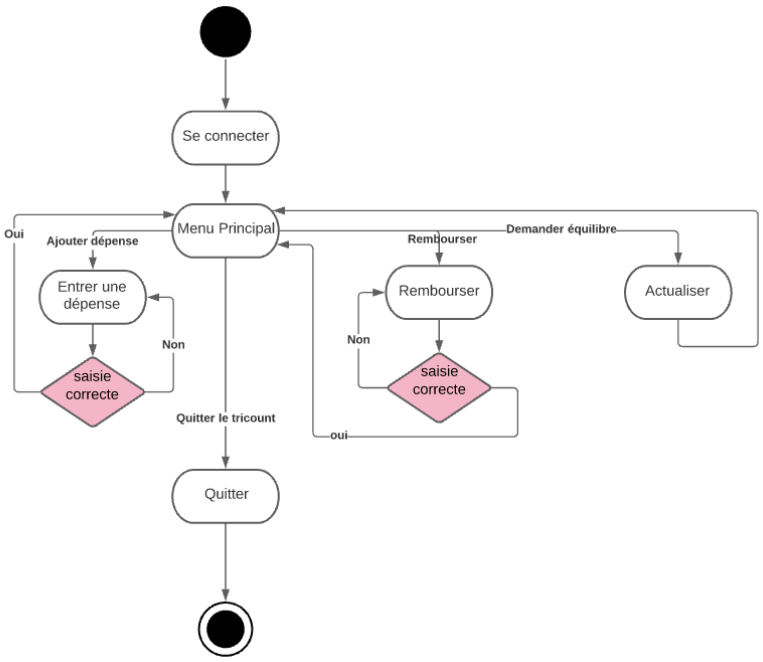

FIGURE 1.7 – Diagramme de navigation

L'utilisateur ouvre le tricount et se connecte, il arrive alors sur la fenêtre principale : le "menu principal". Il peut donc entrer une dépense en cliquant sur le bouton "Ajouter une dépense". Il arrive ensuite sur la fenêtre d'ajout d'une dépense et peut entrer le montant de sa dépense et sélectionner les personnes qui participent à cet achat (il peut se sélectionner lui-même si il paie également). Il clique ensuite sur "Ajouter". Si sa saisie est correcte la dépense est prise en compte et il retourne sur la fenêtre principale, sinon un message d'erreur apparaît et il doit refaire sa saisie. Le principe est le même pour le remboursement. Ensuite, si l'utilisateur sélectionne le bouton "Actualiser" l'affichage s'actualise avec les dernières dépenses (et les derniers participants s'ayant connectés entre temps). Enfin, pour quitter le tricount il suffit de cliquer sur la croix rouge en haut à gauche de la fenêtre principale.

La fonction principale est :

- *public void AddDepense(Int id, String Com, Personne acheteur, Int Val, Personne receveur) throws RemoteException ;* Elle permet d'ajouter une dépense et de rembourser (en faisant une dépense inverse). Les dépenses sont stockées dans un tableau comme vu dans les premiers TDs. Une dépense à un client et un acheteur. Lorsque une dépense est crée elle sera automatiquement stockée plusieurs fois (le nombre de participant à la dépense).

Pour notre Programme nous aurons deux utilisateurs :

- L'administrateur qui lance et ferme le serveur.

- Et les utilisateurs qui se connectent dans un premier temps. Puis ils ont 3 possibilitées :
  - Actualiser les soldes (l'équilibre de chaque personne)
  - Rembourser un autre utilisateur.
  - Ajouter une dépense.

2. UML Conception

2.1 Interaction

On retrouve dans un premier temps le diagramme d'interaction pour la requête "Ajouter dépense" :

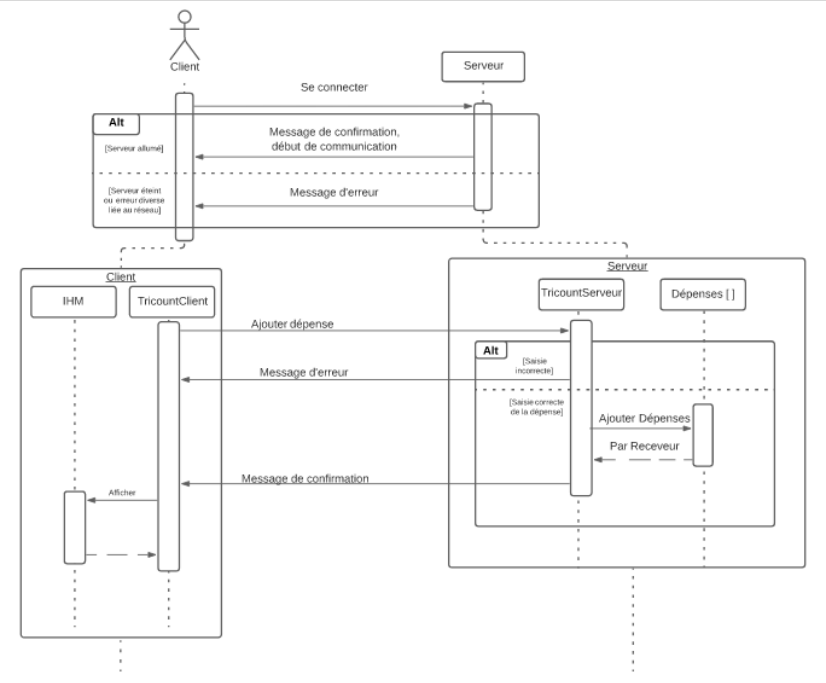

FIGURE 2.1 – Diagramme d'interaction "Ajouter dépense"

On a exactement le même principe que pour le diagramme de séquence. On vérifie d'abord la connexion puis le TricountClient (main) fait une requête d'ajout de dépense au TricountServeur. Dans un premier temps, le Serveur vérifie la saisie. Si cette dernière est incorrecte un message d'erreur est envoyé au client. Si la saisie est correcte alors le serveur ajoute les dépenses au tableau stockées dans TricountServeur. Plus d'informations sont données dans la section Clarification plus bas. Une fois le message de confirmation envoyé, l'IHM est actualisée (la fenêtre est fermée).

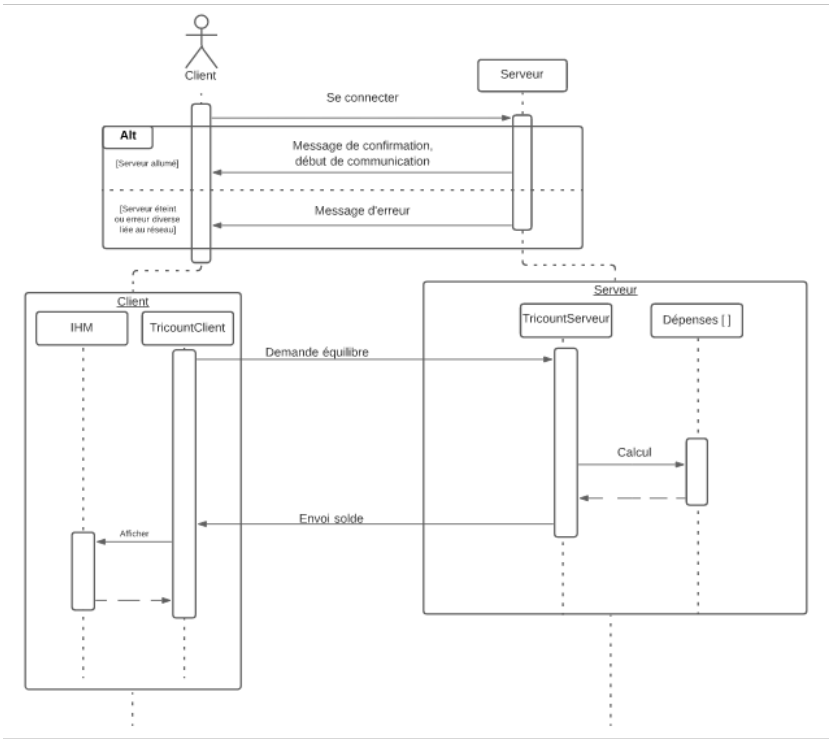

FIGURE 2.2 – Diagramme d'interaction "Demander Equilibre (Actualiser)"

Pour le calcul d'équilibre le principe est similaire au cas précédent. La connexion est vérifiée dans un premier temps et une requête est envoyée par le Client. Ici aucune vérification de saisie n'est nécessaire car l'utilisateur n'a rien à entrer (il clique sur un bouton). Le calcul est effectué côté serveur, encore une fois le détail de calcul est rédigé dans la section Clarification. Une fois le calcul effectué, le résultat est envoyé au TricountClient. Ce dernier met à jour son IHM afin de montrer à l'utilisateur les nouveaux chiffres.

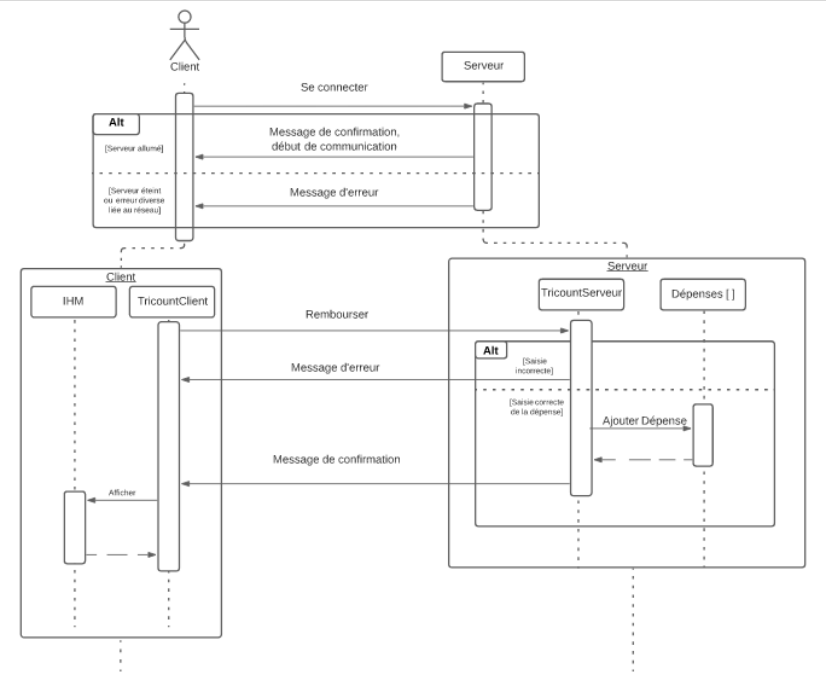

FIGURE 2.3 – Diagramme d'interaction "Rembourser"

Pour le remboursement on effectue la même chose que pour l'ajout de dépenses. Encore une fois vous trouverez plus d'informations dans la section Clarification.

2.2 Classes

Afin de séparer le Serveur et le Client nous avons décidé de faire un diagramme de classe pour chacun des deux.

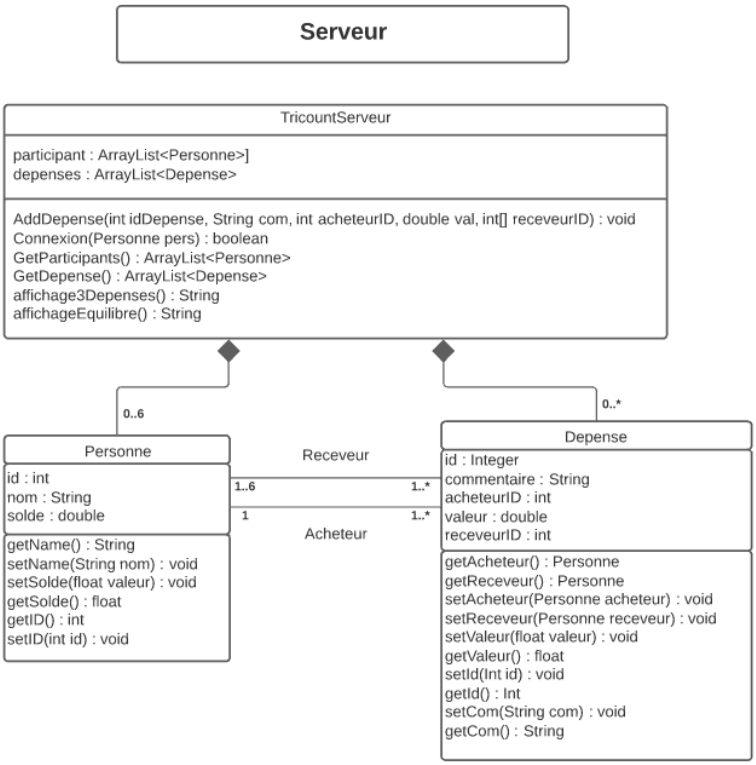

FIGURE 2.4 – Diagramme de classe coté Serveur

Du coté serveur il y aura donc 3 classes pour implémenter ce Tricount.

- La première est *TricountServeur* qui contient le main. Ce sera l'administrateur qui ouvrira et fermera l'application (le serveur) pour permettre ainsi aux clients de s'y connecter par la suite. Elle contient comme attributs un tableau de *Personne* (de taille 6) qui contient tous les participants, et un tableau dynamique de *Depense* regroupant ainsi toutes les dépenses ajoutées depuis le début.

- La classe *Personne* contient le nom et le solde du participant. L'id différente pour chaque personne va permettre de reconnaître deux personnes dans la situation où ils possèdent le même solde et le même nom.

- Enfin la classe *Depense* va avoir comme attribut l'id de la personne qui achète et l'id de la personne qui reçoit la dépense. Pour la même raison, on stockera également une id différente pour chaque dépense pour les reconnaître. On rappelle que s'il y a plusieurs receveurs, il suffit simplement de créer autant de dépenses qu'il y a de receveur pour le même acheteur (Voir la partie clarification un peu plus bas). Enfin on a évidemment le commentaire de la dépense et la valeur de cette dernière.

On remarque bien sur le diagramme que les classes *Depense* et *Personne* dépendent du *ServeurTricont*. Une dépense a bien un seul acheteur avec plusieurs receveurs et chaque personne a son tableau

dynamique de dépense.

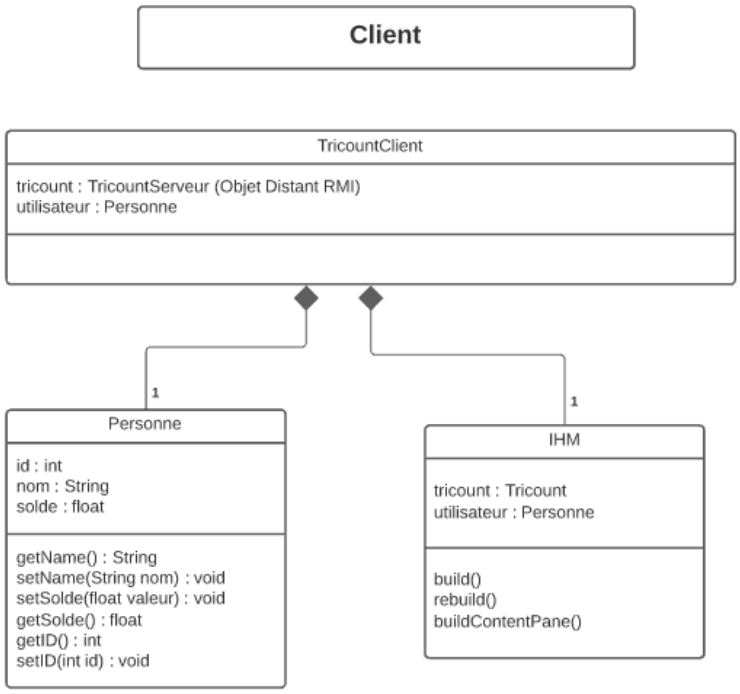

FIGURE 2.5 – Diagramme de classe coté Client

Du côté Client c'est assez similaire. On retrouve deux classes *Personne* et *IHM* qui dépendent de la classe main *TricountClient*. Cette classe *TricountClient* va en réalité chercher à se connecter un Serveur. Le serveur ici est la classe vue précédemment *ServeurTricount*. Une fois le Client connecté, la personne associée aura sa propre interface graphique et va pouvoir ainsi visualiser les dépenses de chacun, calculer l'équilibre ou encore ajouter une dépense. Ceci prend fin lorsque le serveur est fermé.

Ayant implémenté une interface homme-machine (IHM) nous avons également réalisé un diagramme de classe de ce dernier :

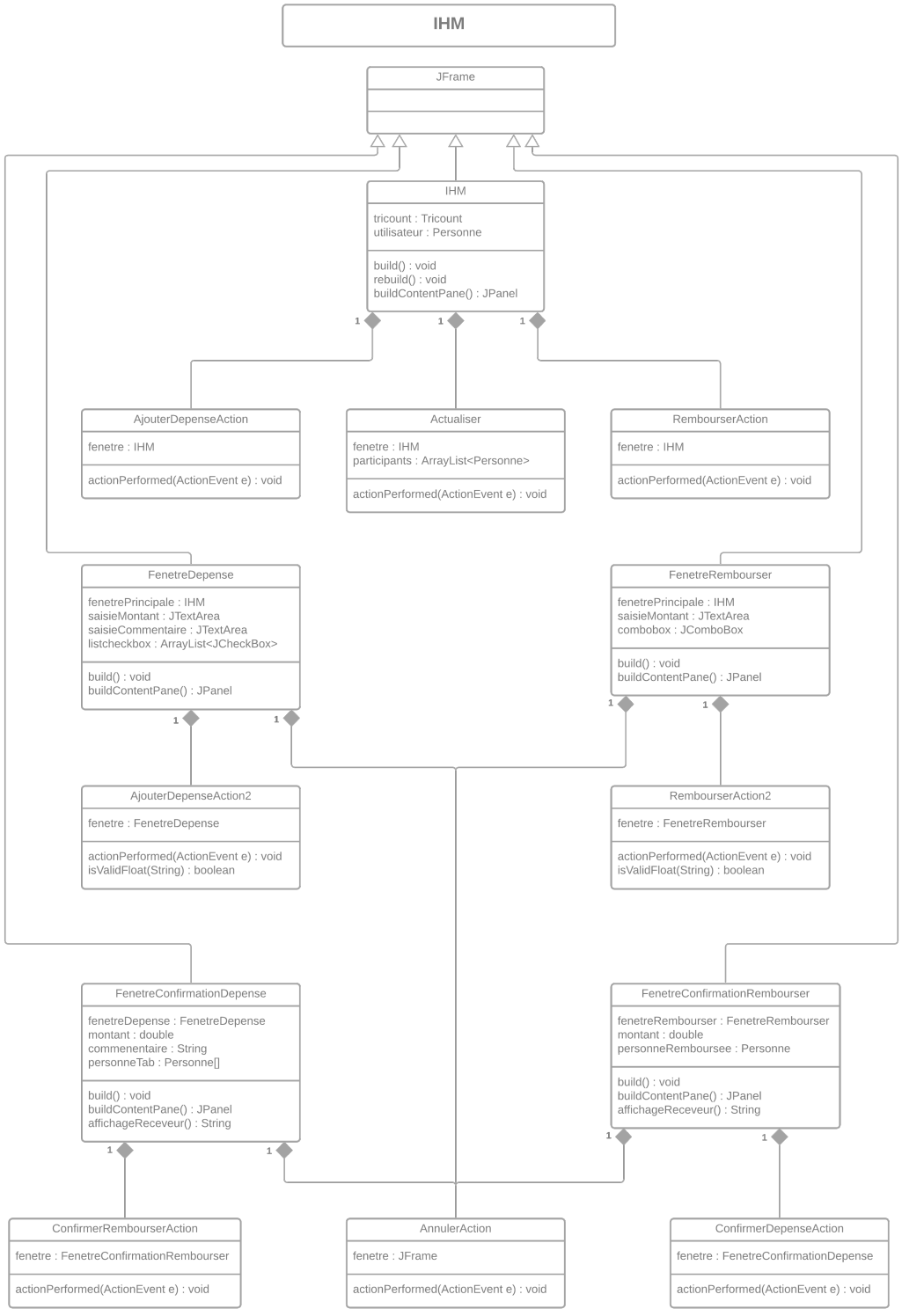

FIGURE 2.6 – Diagramme de classe IHM

Il s'agit essentiellement de manipulations de JFrame et de JPanel sous toute leurs formes, en ajoutant à cela des boutons permettant d'ouvrir ou de fermer les fenêtres. L'interface est légèrement différente de la maquette.

2.3 Paquetage

Nous avons regroupé les éléments de cette façon :

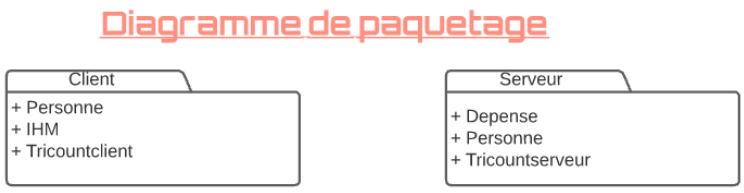

FIGURE 2.7 – Diagramme de paquetage

Le client contient *TricountClient* qui est sa classe principale (le "main"), un IHM sur lequel il pourra naviguer dans l'application et la classe Personne qui décrit le client en question. Le serveur contient la classe Dépense puisqu'il gère l'affichage des dépenses et le calcul de l'équilibre. La classe TricountServeur est sa classe principale (le "main") et la classe Personne lui permet d'associer une dépense à des personnes et de modifier leur solde.

2.4 Composants

Le découpage de composants est représenté comme suit :

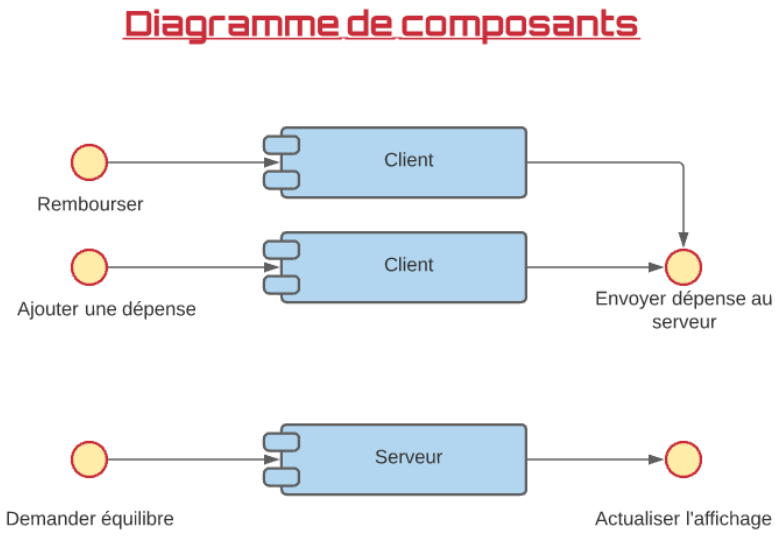

FIGURE 2.8 – Diagramme de composants

Un client peut ajouter une dépense et rembourser quelqu'un. Cette demande est ensuite envoyée au serveur avec le montant et les autres caractéristiques d'une dépense. La demande d'équilibre peut également être demandée au serveur qui, par la suite, va actualiser l'affichage.

2.5 Déploiement

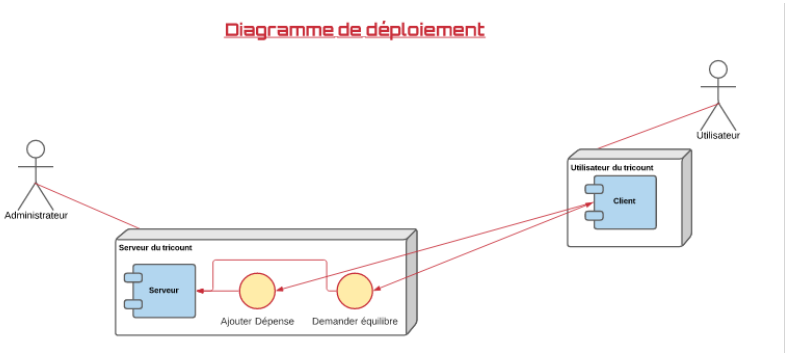

FIGURE 2.9 – Diagramme de composants

Nous avons également fait un diagramme de déploiement pour notre programme à la suite du diagramme de paquetage.

Notre diagramme se décompose en 2 entités :

- **Le client** ou utilisateur qui possède son ordinateur et qui de son côté ne verra que l'IHM après s'être connecté au serveur du Tricount. Il pourra depuis son poste ajouter une dépense ou un remboursement ainsi que demander l'équilibre du Tricount.

- **L'administrateur** qui va s'occuper du serveur qui sera sur un autre appareil que le client. L'administrateur après avoir lancé le serveur permettra au client de s'y connecter et d'effectuer les actions listées précédemment.

Nous aurons donc au moins deux ordinateurs distincts pour ce projet.

3. Clarification pour l'implémentation

- **Stockage des dépenses** : Lors de la création d'une dépense on vient ajouter plusieurs objets au tableau dépenses. Par exemple : Si Paul achète pour 45€ de pizza avec Eugène et Charles. Dans ce cas, on crée 3 dépenses avec le même ID pour pouvoir savoir qu'ils sont liées. Ces trois dépenses ont un montant de 45/3=15€ ils ont l'ID de Paul comme acheteur et l'ID de Eugène, Charles et Paul.

- **Equilibre** : Pour calculer l'équilibre il suffit de sommer les montants où l'identifiant apparaît en tant qu'acheteur. Et ensuite soustraire les moments où l'identifiant est en receveur. Exemple : Dans le cas où Paul achète pour 45€ de pizza avec Eugène et Charles. Si on calcule le cas de Paul on a donc -15 pour sa dépense et ensuite +15*3 dans les 3 dépenses où il est acheteur. Paul aura donc +30 par rapport aux autres.
- **Remboursement** : Pour effectuer un remboursement on doit juste ajouter une dépense "en inverse". Si Paul a remboursé 10€ à Michel dans ce cas on crée une dépense de Paul (acheteur) à Michel (receveur) d'un montant de 10€.

4. Problèmes rencontrés durant l'implémentation

Même si à premiers regards nous pensions que ce projet serait assez simple et rapide vu la simplicité de l'application que tout le monde connaît, il s'en est suivi quelques surprises lors de l'implémentation. En effet ce fut la première fois que nous manipulions le RMI sous forme de projet, et que de plus l'IHM devait gérer les échanges Serveur-Client et Client-Serveur.

La première difficulté fut le temps. En effet ne pouvant pas commencer à programmer avant la validation de l'ensemble des diagrammes, il a fallu s'organiser de manière à être à jour sur les travaux du groupe et finir le projet en deux semaines.

La deuxième difficulté est très vite apparu lors de l'implémentation de l'IHM. Pour beaucoup d'entre nous, la manipulation des JFrame, JPanel, JButton en java ne nous étaient pas familier et il a fallu lire de nombreuses documentations afin de comprendre comment tout cela marchait.

Une fois l'IHM fini, le dernier problème fut de fusionner l'IHM avec le RMI. Lors de partage de code entre chacun d'entre nous grâce à l'outil Git, des fichiers ".class" ne se supprimaient pas ou n'était pas à jour avec un autre camarade ce qui entraînait des erreurs.

Toutes ces erreurs furent minimes et nous ont évidemment appris sur la manière de coder java en général et sur le RMI vu en cours.

Conclusion

Annexe