Projet Clavardage - Rapport COO

Léonie Gallois, Théau Giraud, Alexandre Gonzalvez

4IR-A | 2020/2021

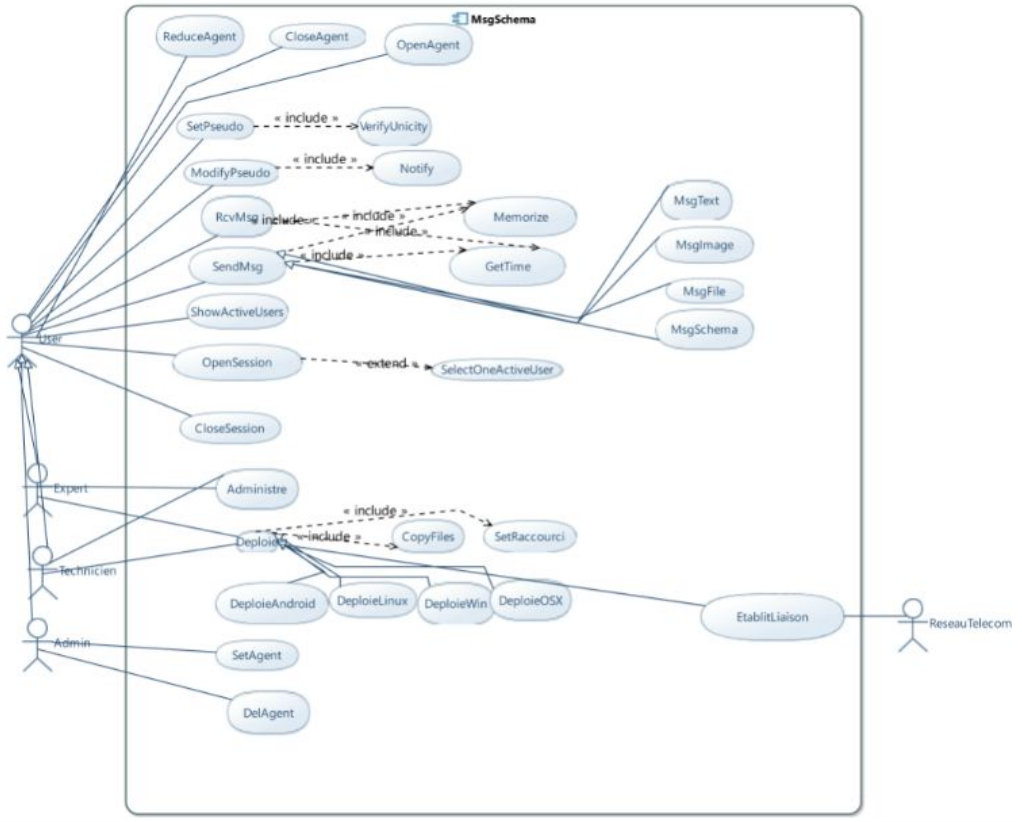
Dans le cadre du projet POO/COO de cette année nous avons dû implémenter un système de clavardage, c.à.d une application de chat live entre utilisateurs distants. La partie COO de ce projet consistait à réaliser plusieurs diagrammes UML de façon à éclairer l'implémentation et l'utilisation de notre application.

L'approche COO au projet nous a permis de visualiser les besoins et fonctionnalités de l'application que nous allions développer. Cela nous a aussi permis de faire certains choix sur l'architecture et le layout des différents objets et classes construisant l'application. Nous avons aussi pu nous remettre en question par rapport à ce que nous faisions et par rapport à nos choix initiaux comparés aux contraintes que nous avons eu pendant le développement.

Ce rapport vient en complément au rapport POO (notamment la partie 1) qui se trouve sur le git avec celui-ci. Les diagrammes sont aussi en version jpg sur le git pour pouvoir être visualisés en plus de détails.

1. Diagramme de cas d'utilisation:

Le diagramme de cas d'utilisation a été notre premier travail de modélisation. Cela nous a permis de nous poser des questions sur les interactions basiques des différents éléments entre eux. Voici notre première ébauche:



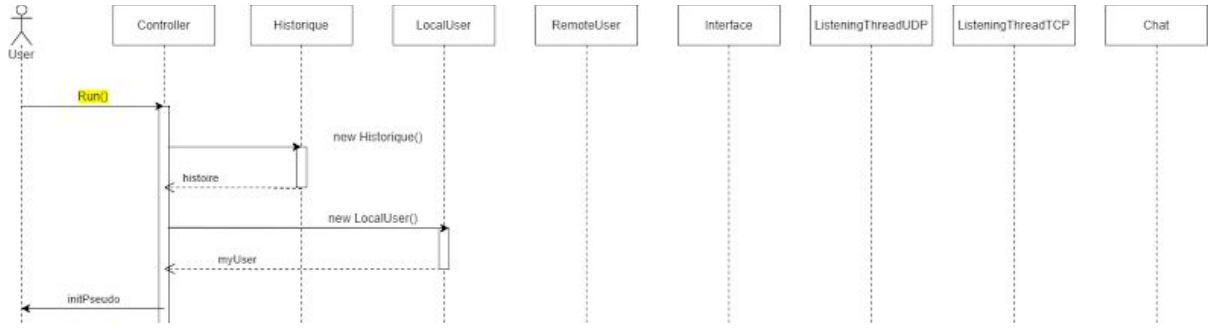
A cette étape, la notion de *Controller* n'avait pas encore été discutée. Nous nous étions basés sur le cahier des charges donné et avions rajouté une fonction pour chaque aspect demandé. Cette première étape nous a permis de mieux visualiser le travail à faire.

2. Diagramme de séquence:

Un diagramme de séquence plutôt simpliste a été fait, puis il a été revu au fur à mesure de notre avancement dans le projet. Ce diagramme nous a permis d'approfondir notre idée des différentes interactions entre les évènements et le fait d'imaginer qu'il y ait un ordre très clair des méthodes appelées nous a aidé plus tard à bien découper les parties à implémenter.

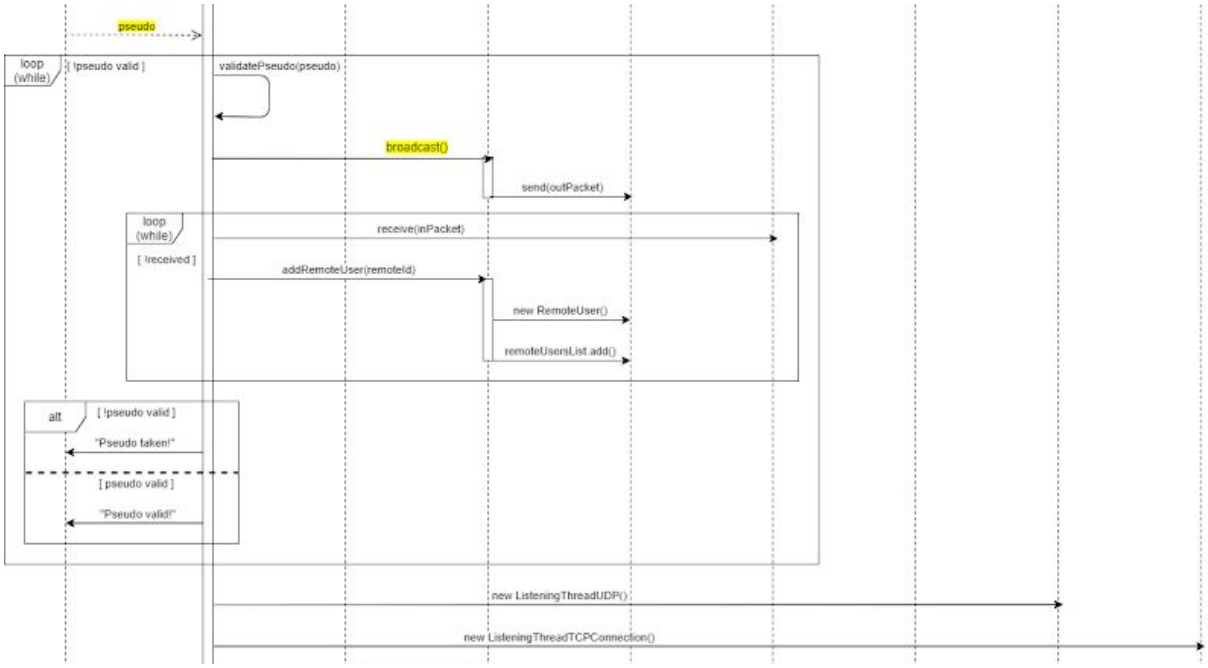
Le diagramme final est un exemple d'une séquence de déroulement complète d'utilisation de l'application. Nous avons ici découpé le diagramme de séquence en étapes importantes (les fonctions appelées) pour pouvoir mieux les commenter. Voici le diagramme:

Lancement de l'application:



Le lancement de l'application crée une nouvelle instance de *Controller*, qui crée son *Historique* local ainsi qu'une instance de *User* qui sera l'objet représentant l'utilisateur local.

Validation du pseudo:

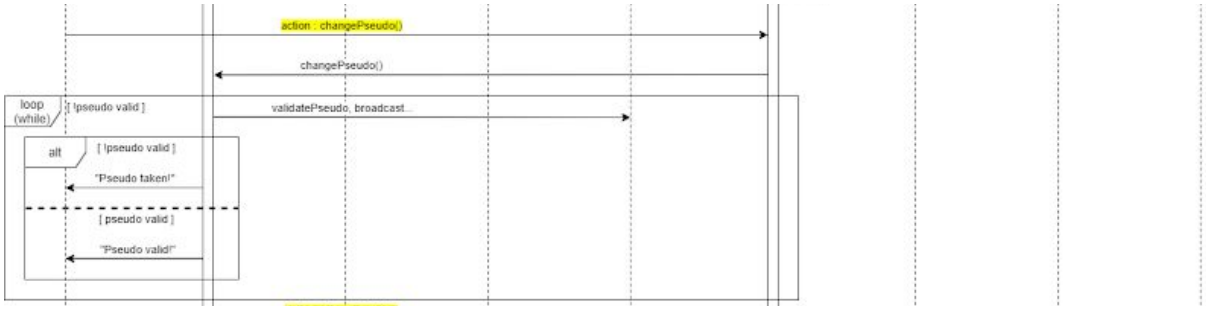


L'utilisateur donne son pseudo en entrée. Le *Controller* appelle sa méthode *validatePseudo* qui fait un broadcast pour demander leurs pseudos aux utilisateurs distants. Il attend (boucle while *!received*) de recevoir toutes leurs réponses et crée une liste d'utilisateurs distants localement. Une autre vérification est ensuite faite par le *Controller* avec sa liste pour vérifier si le pseudo n'est pas déjà pris. S'il l'est déjà, on redemande un pseudo (boucle while *!not_valid*). Le Controller crée aussi deux nouveau threads d'écoute; un pour les communications UDP, *ListeningThread* et un pour les TCP, *ListeningThreadTCP*.

Notification des RemoteUser:

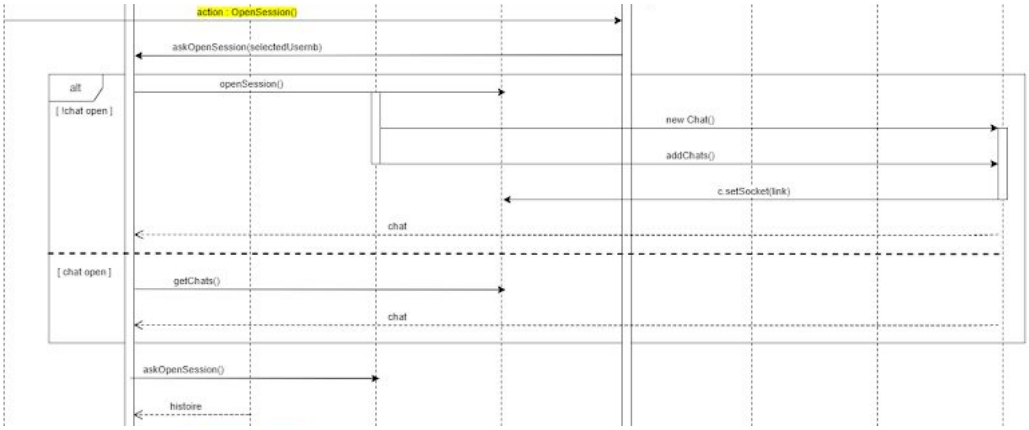
Le *Controller* donne son nouveau pseudo aux utilisateurs distants qui l'ajoutent à leur liste locale de distant users. Il crée ensuite l'interface graphique.

Changer son pseudo:



Le *Controller* fait un autre broadcast aux utilisateurs distants pour récupérer leurs pseudos, puis vérifie, etc. C'est le même mécanisme que *setPseudo*.

Ouvrir une session chat:

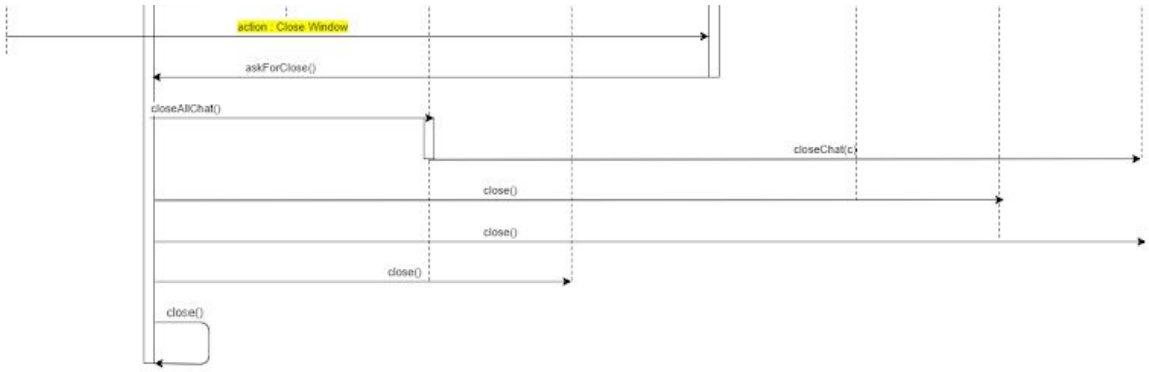


Ouvrir une session chat appelle une méthode de l'interface du *User* vers le *Distant User* qui répond ensuite en acceptant la requête de connexion TCP. Si un chat n'a pas déjà été ouvert depuis le lancement de l'application, le Controller crée un objet *Chat*. Le Controller récupère aussi l'historique de chat précédents en utilisant l'adresse IP (dans notre cas le numéro de port) du *Distant User*.

Envoi de message:

L'envoi de message effectué par l'utilisateur est perpétré à l'utilisateur distant qui confirme la réception. Le *Controller* de l'application ayant envoyé le message ajoute ce message dans le log des messages envoyés dans la base de données. Le message est ensuite sauvegardé localement.

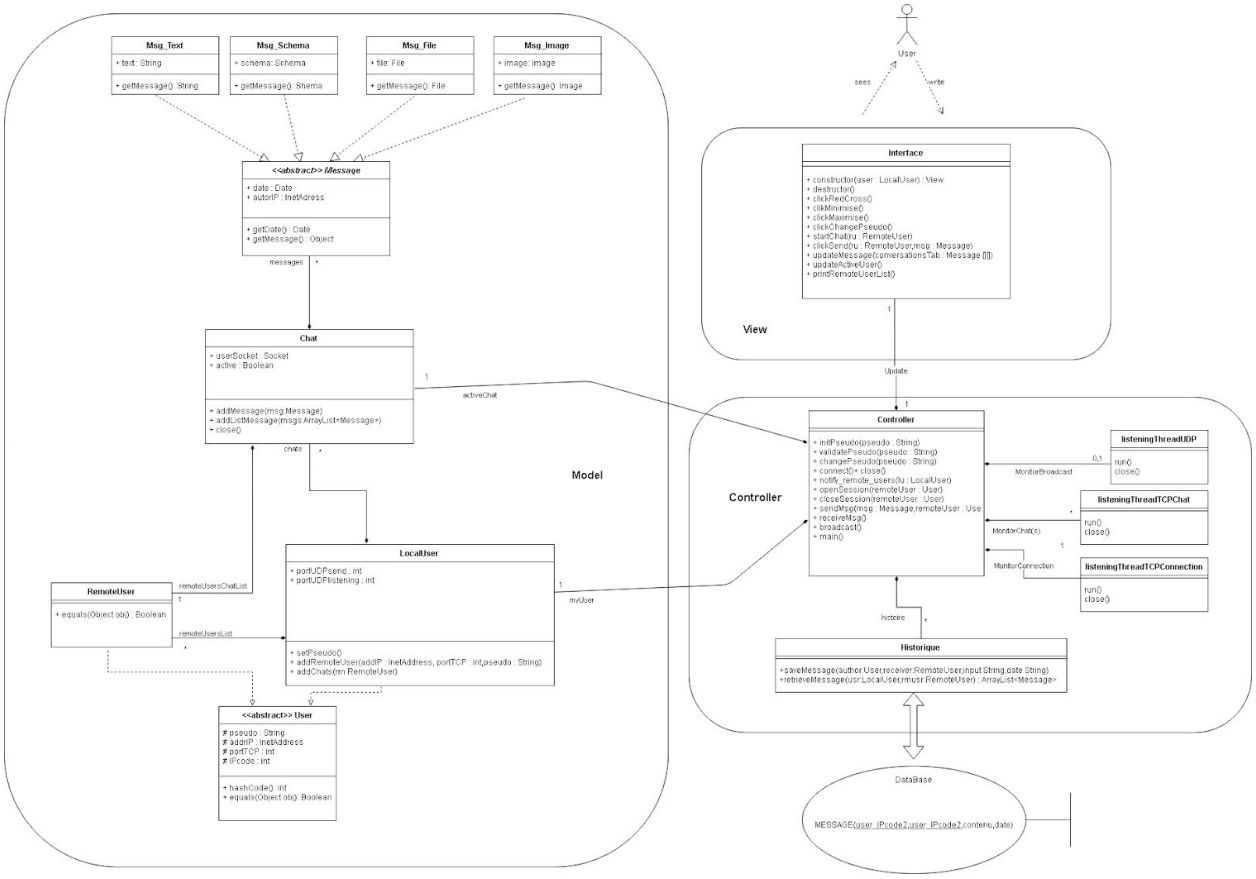
Fermeture de l'application:



L'utilisateur clique sur la croix rouge de l'interface graphique et celle-ci déclenche la fermeture du *Chat* dans *Controller* qui ferme la connexion avec le/les utilisateurs distants, ferme les threads d'écoute et enfin sort de sa boucle while.

3. Diagramme de classe:

Nous avons opté pour une architecture *Model View Controller (MVC)*. L'utilisateur interagit avec la vue. La partie *Model*, représente les différentes données, leurs relations et la façon de les traiter. Le *Controller* est un agent permettant d'exécuter les requêtes des utilisateurs (via l'interface) et donc d'interagir avec le *Model* et de communiquer avec les autres agents.

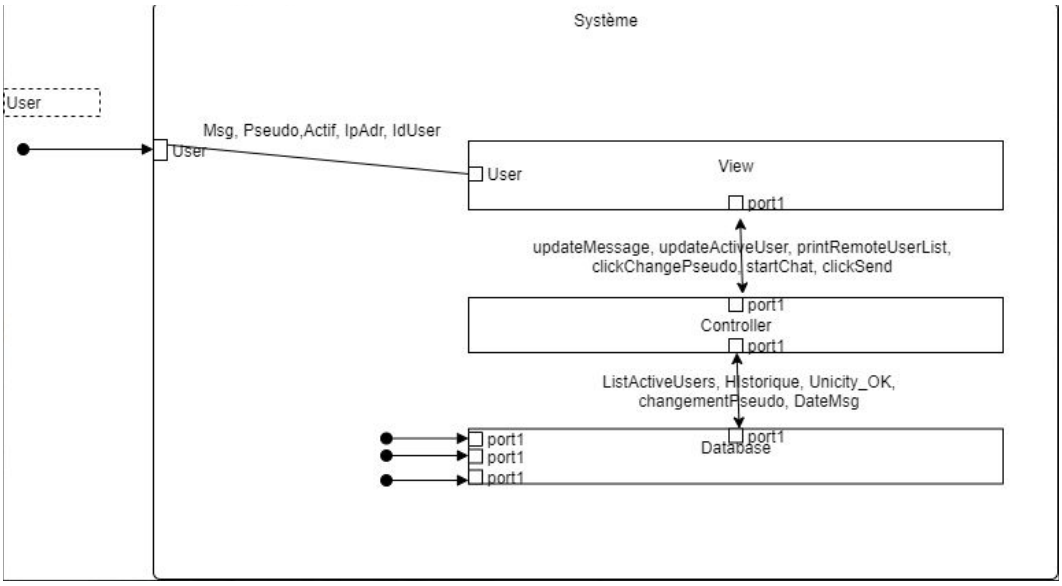


La classe *Interface* permet d'afficher le retour visuel pour l'utilisateur. Elle possède différents observateurs d'événement, qui appellent les fonctions associées dans la classe Controller. La classe *Controller* est centrale à l'application, elle possède une référence vers la vue (Interface) afin de l'actualiser, vers son utilisateur (LocalUser) et vers la session active (Chat). Il possède 1 thread d'écoute UDP, 1 thread d'écoute pour les nouvelles sessions et 1 thread d'écoute pour chaque session.

La classe *Historique* permet de gérer les traitements vers la base de données. Le package *Model* représente les différentes informations d'un utilisateur (LocalUser) : il possède une liste de potentiel correspondant actif (RemoteUser) et une liste de sessions (Chat). Chaque session possède un correspondant (RemoteUser) et une liste de messages (Message).

4. Diagramme de structure composite:

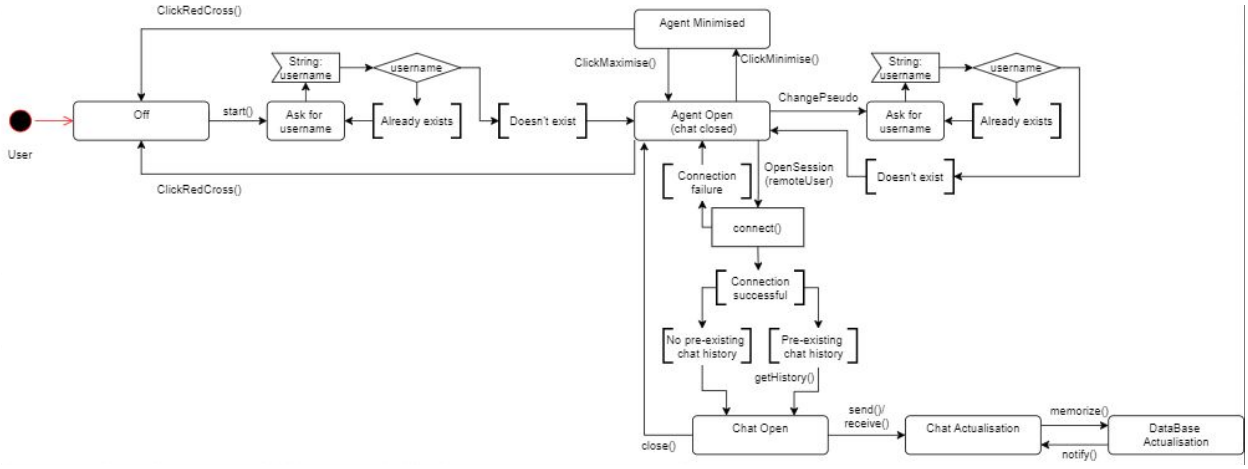
Ce diagramme a été fait en début de conception de projet mais n'a pas été repris après. On peut notamment distinguer la première fois qu'il nous est venu l'idée d'utiliser un *Controller* pour gérer le système.



5. Machine à états:

Enfin, faire un diagramme de machine à états nous a permis de mieux discerner les “boucles” importantes où une mauvaise utilisation/implémentation aboutirait à un crash de l’application. Nous avons pu distinguer les grandes lignes à suivre pour coder notre système.

Ce diagramme de machine à états est comme une version simpliste de notre diagramme de séquence. On peut y voir la validation de pseudo qui boucle en cas de mauvais pseudo, l’ouverture d’un chat avec une bonne connexion du côté de l’utilisateur distant, le changement de pseudo ainsi que la sauvegarde du chat dans la base de donnée.



Conclusion:

Avec notre analyse du cahier de charge et la traduction vers la conception orientée objet que nous avons fait, nous avons pu mieux cibler notre projet en Java. La diagrammes UML permettent de visualiser les classe et méthodes importantes ainsi que l'ordre dans lequel il faut les appeler pour un bon déploiement de l'application. Chaque diagramme permet d'avoir une vue plus précise d'un aspect ou un autre du projet.

Nous pensons avoir gagné en temps sur l'implémentation de l'application grâce à la conception faite au préalable. Nous avons aussi appris des méthodes sûres de conception d'un gros projet en programmation orientée objet et pourrons appliquer ces concepts dans nos projets futurs.