

Arc Attorney



Costa Bruno, Izzo Valentino et Lopes Da Silva Diogo - Le Callennec Benoît, Rizzotti Aïcha, Senn
Julien

P2, HE-Arc, 15.06.2021

Résumé

Ce document regroupe toutes les informations sur les procédés effectués pour créer le jeu Arc Attorney en Java. De l'introduction à la conclusion, chaque paragraphe permettra de mieux comprendre son fonctionnement.

Arc Attorney est un jeu multijoueur de débat sous forme de chat. Un cas est présenté aux joueurs et le but de la partie est de prouver l'innocence ou la culpabilité d'un suspect. L'avocat de la défense devra proposer des arguments solides tout en déjouant ceux du procureur pour ainsi sauver l'âme en peine.

Quant au procureur, il devra faire son possible pour accuser et mettre en péril le sort du suspect. Finalement, c'est le juge qui possédera le dernier mot. Il donnera un verdict et dépendant de celui-ci, un des deux joueurs remportera la partie.

L'application fonctionne correctement, la connexion au serveur se fait aussi bien localement qu'à distance et une partie peut être déroulée sans problème. Le [cahier des charges](#) est ainsi concrètement rempli.

Des fonctionnalités de gestion de preuves et de questionnement de témoins sont encore nécessaires à être implémentées.

Arc Attorney

Résumé

1. Introduction

- 1.1 Mise en contexte
- 1.2 Le sujet du projet
- 1.3 Les détails du jeu
- 1.4 Cahier des charges
 - 1.4.1 Objectifs du cahier des charges
- 1.5 Le plan d'attaque

2. Analyse

- 2.1 Discussions entre les membres du groupe
- 2.2 Création de croquis des interfaces graphiques
 - 2.2.1 Le menu de connexion
 - 2.2.2 Le lobby
 - 2.2.3 Le jeu côté avocat
 - 2.2.4 Le jeu côté juge
- 2.3 Scénarii
 - 2.3.1 Scénario I
- 2.4 Spécifications
 - 2.4.1 Personnages
 - 2.4.1.1 Avocats
 - 2.4.1.2 Juge
 - 2.4.1.3 Témoins
 - 2.4.2 Tâches bonus

3. Planification

- 3.1 GANTT, Gitlab & Discord
- 3.2 Retard
- 3.3 Améliorations du planning

4. Conception

- 4.1 Langages et/ou frameworks utilisés
 - 4.1.1 Outils externes et librairies
- 4.2 Diagramme UML
 - 4.2.1 Partie graphique
 - 4.2.2 Partie modèle
- 4.3 Architecture réseau
 - 4.3.1 Serveur broadcast
 - 4.3.2 Client
 - 4.3.3 Communication

5. Réalisation

- 5.1 Architecture réseau
 - 5.1.1 Serveur broadcast
 - 5.1.1.1 Server
 - 5.1.1.2 ServerThread
 - 5.1.2 Client
 - 5.1.2.1 Player
 - 5.1.2.2 PlayerThread
 - 5.1.2.3 MessageListener
 - 5.1.3 Message et types de message
 - 5.1.3.1 Type de message
 - 5.1.3.2 Sous-type de message
- 5.2 Fenêtre de connexion
 - 5.2.1 Démarrage d'une partie
 - 5.2.1.1 Rejoindre une partie
 - 5.2.1.1 Créer une partie
 - 5.2.2 Erreur(s)
 - 5.2.3 À Propos
- 5.3 Table

5.4	Skin	
5.5	Fenêtre de lobby	
5.5.1	Gestion des clients dans le lobby	
5.5.2	Erreur(s)	
5.6	Case	
5.7	Holder	
5.7.1	Texture et Atlas	
5.8	MessageArea	
5.9	GameUI	
5.9.1	TextArea	
5.9.2	Bouton Send	
5.9.3	Bouton Objection - GameUIAP	
5.9.4	Bouton Verdict - GameUIJudge	
5.9.4	Réception de messages	
5.10	Historique des messages	
5.10.1	Création	
5.10.2	Ouverture	
5.10.3	Ajout des messages	
5.11	Fenêtre de fin de partie	
6.	Tests	
7.	Résultats	
7.1	Résumé des objectifs du module	
7.2	Résumé des objectifs du projet	
8.	Conclusion	
9.	Bibliographie	
10.	Annexes	

1. Introduction

Ce rapport permet au lecteur de comprendre comment le jeu Arc Attorney a été conçu de la première ligne de code, au dernier placement de bouton.

1.1 Mise en contexte

Ce projet a été réalisé dans le cadre du module de deuxième année, le P2. La durée du développement s'est déroulée sur le second semestre de 2020-2021. L'application sera entièrement conçue avec le framework LibGDX en Java.

1.2 Le sujet du projet

Le projet est très grandement inspiré du jeu de Nintendo "Ace Attorney" qui suit le héros, Phoenix avocat de la défense, dans un système juridique américain. Ce jeu est totalement solo et nous avons voulu ajouter une dimension multijoueur tout en respectant le jeu original via les animations, les objections, les personnages, etc.

1.3 Les détails du jeu

Le but du jeu est d'incarner le juge, le procureur ou l'avocat de la défense.

- Le juge est l'arbitre du jeu, c'est lui qu'il faut convaincre et qui va décider qui remporte la partie.
- Le procureur doit prouver que l'accusé est bien coupable, démontrer les arguments de l'avocat de la défense afin d'avoir un verdict coupable.
- L'avocat de la défense, lui, doit prouver que l'accusé est innocent, il doit apporter des arguments allant dans son sens et selon le témoignage de l'accusé et/ou des potentiels témoins.

1.4 Cahier des charges

En tant que propre client, le cahier des charges a été rédigé par le groupe et validé par les enseignants responsables du module.

1.4.1 Objectifs du cahier des charges

L'objectif principal du [cahier des charges](#) est d'apprendre à utiliser des interfaces graphiques via Swing dans un premier temps bien que, par la suite, le groupe a décidé de migrer vers LibGDX. De plus, l'utilisation d'un schéma client-serveur en Java via les sockets a été ajoutée aux objectifs. Quant aux objectifs du projet, ils sont mis en formes dans le [cahier des charges](#) mis en annexe.

1.5 Le plan d'attaque

Le projet est d'abord passé par une phase d'analyse où plusieurs réflexions sur l'architecture et la forme du projet ont été mises en place. Puis une phase de planification afin de savoir quelles seraient les tâches liées au projet et combien de temps elles prendraient, pour ensuite schématiser son fonctionnement et ainsi comprendre comment l'application fonctionnera lors de la [conception](#). Finalement, la phase de réalisation du dit projet.

2. Analyse

L'analyse du projet est la première phase décisive dans la création d'une application. C'est ici qu'il va falloir établir les "règles" du logiciel, comment il va fonctionner et ce qu'il faudra implémenter.

2.1 Discussions entre les membres du groupe

La première étape était de mettre à jour la compréhension du projet au niveau du groupe, pour que chaque membre sache comment le jeu allait être et quelles seraient ses fonctionnalités. Un "brainstorming" général a alors été fait pour noter les différentes visions et idées sur le déroulement du programme.

Comme Arc Attorney est un jeu du genre visual novel, de multiples hypothèses ont donc été créées sur le suivi du programme en s'inspirant de la célèbre série de jeu vidéo "Ace Attorney", mais en multijoueur.


L'objectif du jeu n'est pas de faire en sorte que le joueur suive une histoire de façon linéaire, mais que les joueurs créent eux-mêmes leur histoire avec leur ingéniosité et créativité.

2.2 Création de croquis des interfaces graphiques

Pour mieux se représenter les besoins de l'utilisateur quant à l'interface graphique, des maquettes basiques ont été conçues. De celles-ci en est ressortie l'idée principale que le logiciel devait avoir 3 fenêtres principales. Quelques autres popups ont été désignés pour servir à l'utilisateur.

2.2.1 Le menu de connexion

Arc'ttorney - Connexion — □ ×



Pseudo :

Adresse IP :




Fig.1 Écran de connexion

2.2.2 Le lobby

Fig.3 Interface de jeu Avocat

2.2.4 Le jeu côté juge

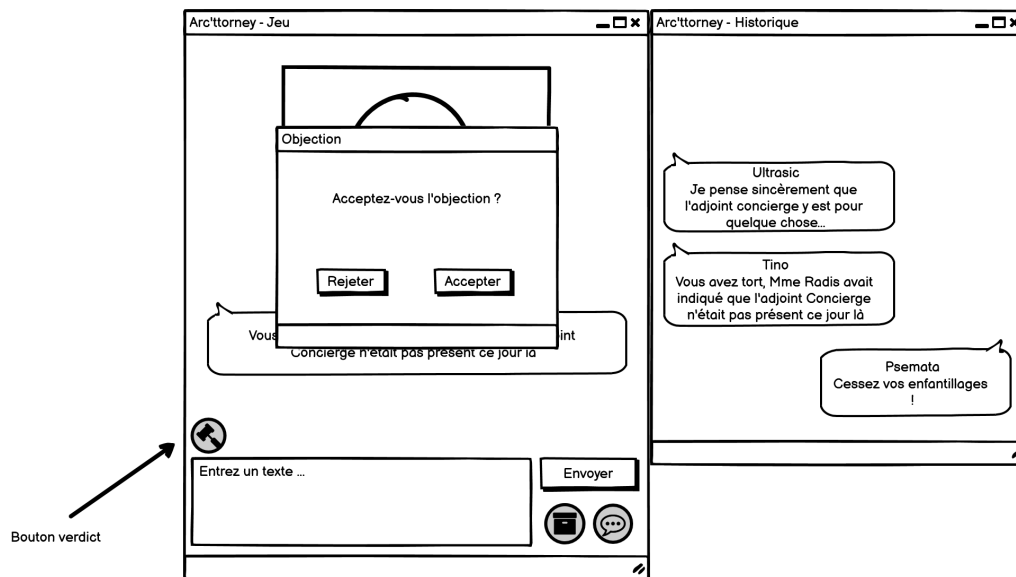


Fig.4 Interface de jeu Juge

2.3 Scénarii

Pour but d'approfondir la recherche, quelques scénarii ont été simulés sur papier et un spécifiquement est ressorti comme une partie "normale"

2.3.1 Scénario I

- Un utilisateur crée la partie et les deux autres la rejoignent.
- Ils arrivent sur un lobby
- L'hôte peut changer le cas courant
- Tant que les utilisateurs n'ont pas tous choisi leur rôle et qu'ils n'ont pas tous mis prêt, l'hôte ne peut pas lancer la partie.
- Une fois la partie lancée, le juge commence avec un petit résumé de l'affaire pour que tout le monde soit au courant.
- C'est aux avocats de prendre les devants et d'émettre leurs arguments quant à l'innocence ou la culpabilité de l'accusé.
- Pour soutenir ses arguments, un avocat peut présenter une preuve.
- Une preuve est récoltée sous forme de texte pour appuyer ses dires.
- Un joueur peut à tout moment consulter l'historique des messages.
- Un avocat peut émettre une objection pour instantanément contredire ce que dit son opposant.
- Le juge peut accepter ou non l'objection.
 - Si acceptée, l'objection se joue
 - Si refusée, rien ne se passe.
- Une fois convaincu, le juge décide de qui gagne et finit la partie.
- Les joueurs peuvent décider de quitter le jeu ou recommencer
 - S'ils décident de recommencer alors ils retournent sur l'interface de connexion.

2.4 Spécifications

Une phase très importante de l'analyse est la [spécification](#). C'est ici que le groupe va documenter tous les détails du projet et, de manière générale, ses fonctionnalités.

2.4.1 Personnages

Le noyau du programme se base sur ses joueurs et donc, ses personnages. L'avocat de la défense, le procureur et le juge forment le système du jeu. En effet, comme dit précédemment, leur ingéniosité et leur créativité vont faire avancer la partie. Sans ces éléments, le jeu n'existe pas.

2.4.1.1 Avocats

Les avocats doivent prouver l'innocence ou la culpabilité de l'accusé et pour se faire possède la capacité d'interagir avec les autres joueurs via discussions, objections et mentions de preuves.

2.4.1.2 Juge

Le juge doit arbitrer la partie, c'est lui que les autres joueurs doivent convaincre pour résoudre l'affaire. Il peut donner un verdict une fois convaincu pour mettre fin à la partie et ainsi condamner ou innocenter l'accusé.

2.4.1.3 Témoins

Il y a des personnages supplémentaires qui ne sont pas jouables qui serait le client ainsi que des témoins. Ils ont des informations clefs pour réussir à démêler le vrai du faux et aident à orienter les joueurs.

2.4.2 Tâches bonus

Le groupe avait également prévu une tâche bonus, à faire si le temps le permet. Le choix s'est porté sur l'utilisation d'un arbre de décision pour les témoins, qui permettrait une interaction plus complexe entre les joueurs et les personnages non joueurs.

3. Planification

La planification est là pour séparer le projet en tâches et les placer dans une échelle temporelle. Ce procédé est fait pour estimer quelles seraient les tâches prenant plus ou moins de temps. Ainsi, une fois préparées dans un échéancier, un procédé, une marche à suivre se démontre en suivant un ordre de priorité des tâches.

Le support choisi pour cet échéancier est le [GANTT](#). De plus, pour ce qui est de la communication, le groupe a décidé d'utiliser Discord.

3.1 GANTT, Gitlab & Discord

Le [GANTT](#) a été un des premiers documents qui a été préparé. En effet, il nous permet d'organiser les différentes opérations à faire et de nous rendre compte de la taille de notre projet et du nombre de tâches qui doivent être réalisées. De plus, nous pouvons y entrer un suivi des heures passées sur les différentes tâches du projet.

Sur la forge de l'école, les fonctionnalités des *issues*, *boards* et *milestones* ont été exploitées. Grâce à celles-ci, nous avons pu mettre en évidence les tâches finies, en cours et à faire de manière ergonomique. Puisque nous voulions un projet centré en un endroit, l'emploi du Wiki de gitlab a été utilisé pour rédiger la documentation, les journaux de travail ainsi que les différents documents liés au projet.

Finalement, Discord était et est toujours l'outil de communication préféré du groupe. Un serveur a donc été créé pour accueillir nos conversations et les différents partages de fichiers.

3.2 Retard

Au cours du projet, le groupe a été pris de court par la quantité de travail à accomplir dans l'ensemble des cours du deuxième semestre de deuxième année. Ce qui a fait que le groupe a dans un premier temps vu la charge de travail un peu à la baisse dans le projet. Bien que tout n'a pas pu être fait, le groupe a réussi à rattraper beaucoup de retard en augmentant le rythme de travail sur le P2.

3.3 Améliorations du planning

C'est après une analyse du planning que nous avons trouvé plusieurs facteurs permettant d'améliorer une future gestion de projet :

- Meilleure analyse des tâches à effectuer et du temps qu'elles demandent pour les finir
- Meilleure répartition des tâches

4. Conception

Après la planification, il faut maintenant définir en détail les bases de l'architecture du logiciel ainsi que ses différentes classes et fonctions.

Une maquette pour une interface graphique, un schéma de classe pour une fonctionnalité, tout est revisité pour avoir un plan d'attaque concret et simple d'exécution.

4.1 Langages et/ou frameworks utilisés

Le langage à utiliser est le Java et comme le groupe a eu des soucis avec Swing, il a été décidé d'utiliser le framework LibGDX qui permet de créer des interfaces graphiques plus souples.

4.1.1 Outils externes et librairies

Les outils utilisés sont :

- Texture packer, un outil utilisé pour créer des packs de texture, il a servi pour créer les fichiers utiles à l'animation.
- Gson, une librairie qui permet de transformer une classe Java sous forme json et vice-versa.

4.2 Diagramme UML

L'architecture du programme est la partie la plus importante de la [conception](#), c'est ici que les premières bases de code se mettent en place. Nous avons donc créé un diagramme UML représentant la structure générale du logiciel.

Ce diagramme est séparé en deux parties, la première est l'ensemble de la partie graphique et la seconde regroupe les classes de la partie modèle.

Ces deux parties sont liées à la classe principale "ArcAttorney" qui représente le jeu en lui-même.

4.2.1 Partie graphique

Dans le package *screens* se trouvent toutes les classes représentant les interfaces graphiques et leurs fonctionnalités. Chaque étape graphique du jeu est une classe qui hérite de Screen afin de pouvoir afficher des menus, des boutons et autres.

1. ConnectionScreen

- La ConnectionScreen est la fenêtre au début du jeu, c'est sur celle-ci que l'on va créer une partie ou se connecter.

2. LobbyScreen

- Le LobbyScreen est une fenêtre de lobby où les joueurs pourront changer de rôle avant le début de la partie. De plus, l'hôte pourra changer l'affaire choisie.

3. EndScreen

- C'est le menu de fin, une fois que la partie est terminée, c'est ici que l'utilisateur arrivera.

4. Interface de jeu

1. GameUijudge

- Si l'utilisateur possède le rôle de juge, son interface sera une GameUijudge qui possède des actions uniques au juge.

2. GameUiAP

- Si l'utilisateur possède soit le rôle d'avocat de la défense, soit le rôle de procureur, son interface sera une GameUiAP.

Les deux types d'interfaces de jeu possèdent des éléments tels que :

- MessageArea : Une zone qui va afficher les messages des joueurs de manière spécifique.
- DisplayHolder : La zone qui contient l'animation du personnage.
- History : Une fenêtre de dialogue qui affiche l'historique des messages envoyés depuis le début de la partie.

4.2.2 Partie modèle

La partie modèle, elle, possède l'architecture logique du projet, là où les calculs et les lignes de code importantes du programme vont se faire.

1. Player

- C'est la classe du client. C'est le Player qui va se connecter au serveur, communiquer et recevoir des données vers et en provenance de celui-ci.

2. PlayerThread

- La classe PlayerThread va servir d'écoute pour le joueur, c'est elle qui va recevoir les informations provenant du serveur.

3. Server

- C'est la classe du serveur. C'est lui qui va retransmettre aux clients les informations qu'il a reçues.

4. ServerThread

- Comme pour le PlayerThread, c'est une classe qui va servir d'écoute pour le serveur. Toutes les informations provenant des clients passent par ici.

5. Role

- Une énumération qui définit le rôle d'un joueur.

6. MessageListener

- Un listener qui va réagir à chaque fois qu'un joueur reçoit un message.

7. Message

- La classe Message est la classe qui contient les informations qui vont être envoyées aux autres clients.

8. MessageType

- Une énumération qui définit le type principal du message. Si c'est un message de modification de données du lobby ou un message de discussion du chat.

9. SubMessageType

- Cette énumération est utilisée pour définir un sous-type aux deux autres types principaux. Par exemple, un sous-type indiquant un changement de rôle lors de la phase de lobby.

10. Case

- Elle va contenir les informations concernant l'affaire sur laquelle les joueurs vont débattre.

4.3 Architecture réseau

Comme l'application se base entièrement sur un serveur et des clients, l'architecture de ce système est ainsi le fondement du bon fonctionnement du jeu. Le groupe étant inexpérimenté, des recherches ont en premier lieu été faites et c'est après avoir trouvé et comparé diverses façons de faire que cette solution a été choisie.

4.3.1 Serveur broadcast

Le jeu a besoin d'un serveur procurant uniquement une fonctionnalité de redirection de messages en broadcast. À chaque message reçu de la part d'un client, il va le rediriger à tous ses clients.

Son fonctionnement est simple, le serveur possède deux threads :

1. Un de connexion, qui va écouter à chaque instant si un client tente de se connecter au serveur, et ce jusqu'à la mort du serveur.
2. Un autre d'écoute, qui lui va écouter les données provenant des clients pour ensuite les retransmettre.

Ce thread d'écoute va être une instance d'une classe nommée ServerThread

4.3.2 Client

Le client va être le joueur se connectant au serveur. Il va posséder les éléments nécessaires pour participer au jeu ainsi que les fonctions et caractéristiques lui permettant la connexion et la communication avec le serveur.

Il ne possède qu'un seul thread d'écoute du nom de PlayerThread, c'est lui qui va récupérer les informations provenant du serveur pour ensuite les traiter.

C'est par l'interface graphique que le client va envoyer des données au serveur, en envoyant un message ou en se mettant prêt lors de la phase de lobby par exemple.

4.3.3 Communication

Pour simplifier la communication, du texte simple ne suffit pas. La classe message a alors été créée pour ainsi contenir toutes les informations nécessaires. Le contenu d'un message, le rôle de l'expéditeur ou encore sa position dans le lobby seront ainsi envoyés de via cet objet qui lui-même sera transformé sous forme json.

5. Réalisation

C'est ici que la création et l'implémentation du projet, de l'architecture réseau aux fonctionnalités du jeu, sont expliquées en détail.

5.1 Architecture réseau

Comme dit précédemment, l'architecture réseau est la base du programme. C'est par ses classes que toutes les autres vont passer pour communiquer et produire des changements dans l'interface graphique et dans le déroulement du jeu. La technologie des sockets a été choisie pour créer et gérer cette architecture. Ainsi, c'est dans cette section que son fonctionnement sera expliqué et détaillé.

5.1.1 Serveur broadcast

5.1.1.1 Server

Le serveur va en tout premier lieu créer un "ServerSocket", c'est le serveur qui va accueillir les connexions et les requêtes provenant des sockets clients. Le serveur possède plusieurs valeurs et caractéristiques importantes : tout d'abord un tableau contenant trois threads d'écoute "ServerThread" qui représentent les clients, un entier indiquant le nombre de clients connectés et finalement un tableau regroupant toutes les informations des clients une fois connectés au lobby. De plus, toutes les communications passent par des I/O tels qu'un BufferedReader et un PrintWriter.

Une fois créé, il va lancer le thread d'écoute de connexion pour permettre à chaque nouveau socket client de correctement se lier au socket serveur.

```
// Server creation
...
try {
    System.out.println("Liaison au port " + port + ", veuillez patienter ...");
    // Launch the server to the specified port
    this.serversocket = new ServerSocket(port);
    System.out.println("Serveur lancé : " + serversocket);
    // Start the connection thread
    start();
} catch (IOException ioe) {
    System.out.println("Ne peut pas se lier au port " + port + " : " +
        ioe.getMessage());
    ...
}
```

À la création de l'objet Server, le ServerSocket est créé et le thread de connexion est également lancé.

```
public void start() {
    if (this.connectionThread == null) {
        this.connectionThread = new Thread(new Runnable() {
            @Override
            public void run() {
                while (connectionThread != null && connectionBoolean.get()) {
                    try {
                        System.out.println("En attente d'un client ... ");
                        // wait for connections
                        addThread(serverSocket.accept());
                    } catch (IOException ioe) {
                        System.out.println("Le serveur n'a pas pu accepter la
connexion : " + ioe);
                        stop();
                    }
                }
            }
        });
    }
}
```

```

        }
    });
    this.connectionThread.start();
}
}

```

Ici, le thread va continuellement attendre des connexions via la fonction bloquante "serverSocket.accept()". Dès qu'une connexion se fait, la fonction va retourner le socket client pour le traiter dans la méthode "addThread()" qui, elle, va créer un thread d'écoute, l'ajouter au tableau de client et traiter les différentes informations importantes.

```

private void addThread(Socket socket) {
    if (clientCount < clientsTab.length) {
        System.out.println("Client accepté : " + socket);
        clientsTab[clientCount] = new ServerThread(socket, this, clientCount);
        try {
            clientsTab[clientCount].open();
            clientsTab[clientCount].start();
            clientCount++;
        } catch (IOException ioe) {
            System.out.println("Erreur lors de l'ouverture du thread");
        }
    } else {
        System.out.println("Client refusé : taille maximum atteinte - " +
            clientCount);
    }
}

```

Finalement, le serveur possède une fonction de traitement et de *broadcast* de données reçues. Cette fonction va d'abord vérifier la donnée reçue avant de la rediriger. En effet, si le message indique :

- Que le client quitte le jeu.
 - Le serveur doit gérer la déconnexion de celui-ci.
 - Change le sous-type du message pour indiquer aux autres joueurs qu'un client quitte le jeu.
- Que c'est un changement dans le lobby.
 - Le serveur doit gérer ces changements.

Pour ensuite rediriger ce message à tous les clients connectés.

```

if(data.getType() == SubMessageType.QUITTING) {
    clientsTab[findClient(id)].send(gson.toJson(data)); // If the client want to
    disconnect, send the quitting data to him
    remove(id);
    // Set the message data as -> someone is quitting
    data = new Message(null, id, data);
    data.setType(SubMessageType.SOMEONE_IS_QUITTING);
} else if (data.getmType() == MessageType.LOBBY) {
    // Add informations to the array so the next client to connect can retrieve
    it
    lobbyData[id] = data;
    if (data.getType() == SubMessageType.ENTERING_LOBBY) {
        // If the client is entering the lobby, it's collecting the data in the
        server to refresh its UI
    }
}

```

```

        data = new Message(gson.toJson(this.lobbyData), id, data);
    } else if(data.getType() != SubMessageType.AFFAIR_CHANGE) {
        // Send the changes to the other clients
        data = new Message(null, id, data);
    }
}

// Serialize the data received and send it to the clients
for (int i = 0; i < clientCount; i++) {
    clientsTab[i].send(gson.toJson(data));
}

```

5.1.1.2 ServerThread

Le ServerThread est l'oreille du serveur, c'est elle qui va lui donner les informations entendues. À sa création, il reçoit un socket client qui est finalement le client auquel ce thread est lié.

Ce thread possède deux fonctions importantes :

1. Sa fonction *run* qui tourne tant que le serveur est vivant et récupère les informations provenant du client.

```

try {
    // Listen data from the client and give it to the server so it can broadcast
    it
    mainServer.handle(this.id, input.readLine());
} catch (IOException ioe) {
    System.out.println(this.id + " n'arrive pas à lire le message " +
ioe.getMessage());
    mainServer.remove(this.id);
    stopThread();
}

```

2. La fonction *send* qui permet d'envoyer une donnée au client.

```

try {
    output.println(msg);
} catch (Exception ioe) {
    System.out.println(this.id + " n'arrive pas à envoyer de message " +
ioe.getMessage());
    mainServer.remove(this.id);
}

```

5.1.2 Client

5.1.2.1 Player

Le client possède un socket lié au serveur, un thread d'écoute PlayerThread ainsi qu'une liste d'abonnés "Listener" pour ce qui est de la communication. Comme autres attributs, il possède un pseudo et un rôle. À sa création, le socket est initialisé et essaie de se connecter au serveur via son adresse IP et son port pour finalement ouvrir ses flux d'écoute et d'envoi de messages.


```

try {
    // Connect to the server
    this.socket = new Socket(serverAddress, serverPort);
    System.out.println("Connecté : " + socket);
    // Open the Output to the server and create the listening server
    open();
} catch (UnknownHostException uhe) {
    System.out.println("Hôte inconnu ou inatteignable : " + uhe.getMessage());
    throw(uhe);
} catch (IOException ioe) {
    System.out.println("Problème de connexion : " + ioe.getMessage());
    throw(ioe);
}

```

Lors de l'ouverture des flux, le `PlayerThread` est initialisé et lancé pour écouter les informations provenant du serveur. Mais c'est via deux fonctions principales que le client va communiquer :

1. La fonction *handle*, qui permet de traiter les données reçues par le serveur en appelant les différents abonnés *listener*

```

// Receive the data from the server
public void handle(String msg) {
    // Deserialize the data received
    Gson gson = new Gson();
    Message data = gson.fromJson(msg, Message.class);

    if (data.getType() == SubMessageType.QUITTING) {
        System.out.println("Déconnexion du client : " + this.pseudo);
        stop();
    } else {
        // Send the data to all the "listeners"
        triggerUiChanges(data);
    }
}

```

2. La fonction *send* qui, elle, permet d'envoyer un message au serveur et ainsi à tous les autres clients.

```

public void send(Message message) {
    Gson gson = new Gson();
    String data = gson.toJson(message);
    // Send the data to the server
    output.println(data);
}

```

5.1.2.2 PlayerThread

Le `PlayerThread` fonctionne exactement comme le `ServerThread`, c'est un thread d'écoute qui va récupérer les informations provenant d'une source pour ensuite les envoyer à un élément qui va traiter cesdites informations. Dans ce cas-ci, c'est du serveur que proviennent les informations et c'est le client qui va les traiter.

5.1.2.3 MessageListener

L'interface `MessageListener` est utilisée pour déclencher un événement à chaque fois que sa fonction `messageReceived()` est appelée. Le `Player` possède une liste d'abonnés qui vont recevoir cet événement à chaque fois que la fonction `triggerUiChanges()` est appelée, cette fonction va contacter tous les abonnés et va lancer la fonction `messageReceived()` qui va transmettre le message reçu.

```
// when a message is received, the ui changes accordingly
private synchronized void triggerUiChanges(Message data) {
    for (MessageListener msgListener : this.listMessageListener) {
        msgListener.messageReceived(data);
    }
}
```

Cette fonction, `triggerUiChanges()`, est appelée lorsque le client reçoit un message, ainsi, à chaque fois que le client reçoit une nouvelle information, les abonnés recevront également ce message pour modifier l'UI ou certaines données selon sa nature.

Par exemple, lorsque l'UI abonnée est le lobby, certaines modifications vont être faites pour les différents clients.

5.1.3 Message et types de message

Message est la classe "conteneur" qui va être envoyée aux autres clients. Elle contient :

- Le pseudo de l'expéditeur
- Le contenu du message
- La preuve mise dans le message
- Le rôle de l'expéditeur
- Le type du message
- Le sous-type du message
- La position du client dans le lobby

5.1.3.1 Type de message

Le message possède deux types, un type lobby qui définit que le message sera lié à la gestion du lobby (connexion, modification, lancement de partie) et le type chat qui définit le fait que le message fera partie du jeu, lorsque les clients discutent entre eux.

5.1.3.2 Sous-type de message

Il existe plusieurs sous-types utilisés pour différents cas :

- `NORMAL` => indiquant que c'est un simple message
- `OBJECTION` => indiquant que le message est une objection
- `VERDICT` => un message de verdict, qui indique la fin de partie
- `REFUSE_OBJECTION` => indiquant que l'objection est refusée
- `ACCEPT_OBJECTION` => indiquant que l'objection est acceptée
- `ROLE_CHANGE` => un changement de rôle dans le lobby
- `READY` => un joueur est prêt dans le lobby
- `NOT_READY` => un joueur n'est pas prêt dans le lobby
- `ENTERING_LOBBY` => un client entre dans le lobby
- `AFFAIR_CHANGE` => l'affaire a été changée par l'hôte de la partie
- `LAUNCHING_GAME` => l'hôte de la partie lance le jeu
- `QUITTING` => un client souhaite quitter la partie
- `SOMEONE_IS_QUITTING` => indique qu'un autre client a quitté la partie

5.2 Fenêtre de connexion

Cette interface permet à l'utilisateur de créer ou rejoindre une partie d'Arc Attorney. Il a aussi accès à la fenêtre de dialogue "À propos".

5.2.1 Démarrage d'une partie

Pour les joueurs qui veulent rejoindre une partie, ils doivent, comme l'hôte, donner leur pseudo et l'adresse IP de l'hôte.

Pour créer la partie, l'option de création de partie doit être cochée par l'utilisateur et le bouton "Rejoindre" se change en "Créer une partie", ce qui va aussi retirer le champ d'entrée de l'adresse IP.

5.2.1.1 Rejoindre une partie

Si l'utilisateur rejoint une partie, un objet client sera créé et la variable "server" sera mise à *null*. C'est ici que le client se connectera au serveur et fera le changement : Menu de connexion ==> Lobby

```
// Create the client
try {
    game.server = null;
    game.client = new Player(textFieldIp.getText(), 5000,
textFieldPseudo.getText(), Role.UNROLLED);
    game.lobbyScreen.setHostRole(false);
    game.lobbyScreen.listenerControl();
    game.client.send(new Message(game.client.getPseudo(), null, null,
game.client.getRole(), MessageType.LOBBY, SubMessageType.ENTERING_LOBBY));
    // Go to the lobby
    game.setScreen(game.lobbyScreen);
} catch (UnknownHostException ioe) {
    labelError.setText("L'hôte est introuvable.");
} catch (IOException ioe) {
    labelError.setText("Un problème de connexion est apparu.");
}
```

5.2.1.1 Créer une partie

Si l'utilisateur crée une partie, la variable "server" sera un objet Server et un client s'y connectera. De plus, le changement d'interface se fera de la même manière qu'avec un client rejoignant le serveur.

```

game.server = new Server(5000);
try {
    game.client = new Player("127.0.0.1", 5000, textFieldPseudo.getText(),
    Role.UNROLLED);
    game.lobbyScreen.setHostRole(true);
    game.lobbyScreen.listenerControl();
    game.client.send(new Message(game.client.getPseudo(), null, null,
    game.client.getRole(), MessageType.LOBBY, SubMessageType.ENTERING_LOBBY));
    // Go to the lobby
    game.setScreen(game.lobbyScreen);
} catch (UnknownHostException ioe) {
    labelError.setText("L'hôte est introuvable.");
} catch (IOException ioe) {
    labelError.setText("Un problème de connexion est apparu.");
}

```

5.2.2 Erreur(s)

Des messages d'erreurs peuvent apparaître à l'écran si les joueurs oublient de renseigner leur pseudo ou si les joueurs tentent de rejoindre une partie inexistante.

5.2.3 À Propos

Dans cette fenêtre de connexion, les joueurs ont aussi accès à une fenêtre de dialogue "À propos" qui est accessible via un bouton sur la fenêtre. On retrouve les informations utiles pour connaître le nom de l'école et les noms des créateurs.

5.3 Table

Pour positionner les éléments sur une *Screen* libGDX, comme des *labels*, des *textfield* ou des boutons. On peut utiliser la classe *Table* de la même librairie. Cette *Table* est comme un tableau. On va pouvoir ajouter des éléments ligne par ligne. On peut choisir aussi des marges entre chaque ligne, le nombre de colonnes que doit prendre un élément, la largeur et hauteur. Voici un échantillon de code d'une disposition d'une *Table*.

```

...
table.add(this.imageLogo).colspan(2).padBottom(30).width(100).height(100);
table.row();
table.add(this.labelPseudo).padBottom(30);
table.add(this.textFieldPseudo).padBottom(30);
table.row();
...

```

5.4 Skin

Les *skins* dans libGDX sont des apparences qu'on peut donner à nos éléments graphiques. On peut soit créer nos propres *skins* qui sont des fichiers "JSON" et "atlas" ou alors utiliser ceux que propose [libGDX](#). Le projet utilise le *skin* par défaut de libGDX.

5.5 Fenêtre de lobby

C'est ici que les joueurs vont pouvoir attendre la configuration de la partie et les autres joueurs. L'hôte de la partie peut sélectionner l'affaire qui va être jouée. Les autres joueurs peuvent sélectionner leur rôle via une *combobox* avec tous les rôles et se mettre prêts via une *checkbox*. Les éléments graphiques de la fenêtre sont disposés via une *Table* comme pour la fenêtre de [connexion](#).

5.5.1 Gestion des clients dans le lobby

Chaque fois qu'un client entre dans le lobby, ses informations (son pseudo, sa position dans le lobby et son rôle) sont mises à jour dans le serveur. De même lorsqu'il modifie son rôle ou lorsqu'il change son état en prêt ou non-prêt. Ainsi, lorsqu'une nouvelle personne entre, les informations concernant les autres joueurs sont directement mises à jour. De plus, à chaque changement, un message est envoyé aux autres clients pour qu'ils mettent à jour l'interface de leur côté.

5.5.2 Erreur(s)

L'hôte de la partie va avoir des erreurs affichées sur son écran s'il veut démarrer la partie alors que les joueurs ne sont pas tous prêts et que les rôles ne sont pas répartis correctement.

5.6 Case

La classe *Case* permet de créer une affaire scénarisée pour le jeu. Chaque *Case* doit avoir une image correspondante à la tête de l'accusé, une description qui explique pourquoi la personne est jugée, un résumé que le juge va prononcer en début de partie et un index qui sert d'*id* pour différencier les affaires.

La description va être utilisée dans la [fenêtre de lobby](#) pour que les joueurs puissent prendre connaissance de l'affaire et pourquoi la personne est accusée.

5.7 Holder

Le *holder* va contenir les différentes animations et positions de ces dernières. Ces animations sont récupérées depuis des atlas.

5.7.1 Texture et Atlas

Le texture packer est un exécutable utilisé par LibGDX pour créer des atlas. Il crée un *texture sheet* de toutes les images et dans un fichier texte (en .atlas) il va indiquer à l'animation quelle image prendre dans quel ordre et où elle se trouve dans le PNG qui pack toutes les images.

5.8 MessageArea

Le *MessageArea* est une classe qui va simplement écrire un caractère tous les ticks du timer pour créer une animation comparable à celle du jeu Ace Attorney. De plus la classe va aussi jouer un son "blip" pour ajouter un petit effet.

5.9 GameUI

C'est la classe principale du jeu (*GameUIAP* pour les avocats et *GameUIJudge* pour le juge). Elle contient une image pour représenter la court avec par-dessus un batch qui dessinera l'animation contenue dans le Holder.

5.9.1 TextArea

Le TextArea dans lequel le joueur peut écrire, a des fonctionnalités limitées pour éviter de casser l'UI. 100 caractères maximum et impossible de faire de retour à la ligne. Lorsque la touche *enter* est pressée, le message est envoyé en enlevant le retour à la ligne. Lorsque le joueur n'a pas le focus sur le TextArea il peut quand même envoyer le message via la touche *enter*.

5.9.2 Bouton Send

La touche *enter* va activer le bouton pour envoyer le message. Ce que ce bouton fait c'est qu'il récupère le texte du textArea, le clear et envoie le texte récupéré au serveur en indiquant son pseudo, son rôle, etc.

5.9.3 Bouton Objection - GameUIAP

Ce bouton fait le même travail que le send à la différence qu'il ne peut être utilisé que toutes les 30 secondes, car lui peut être utilisé même quand un message est encore en train de s'afficher.

5.9.4 Bouton Verdict - GameUIJudge

Lorsque ce bouton est pressé, une fenêtre de dialogue arrive avec un choix pour le verdict. Une fois que le juge a choisi, un message est envoyé au serveur indiquant un verdict.

5.9.4 Réception de messages

Quand un message est reçu, on va vérifier s'il est bien de type chat. Si c'est le cas on vérifie si c'est un message normal, objection ou un verdict.

- Dans les deux premiers cas, le bouton send sera désactivé le temps que le message finisse de s'afficher pour éviter le spam.
 - Les deux afficheront l'animation du joueur correspondant.
 - L'objection aura une animation supplémentaire pour montrer que c'est une objection.
- Lorsque c'est un verdict, on vérifie le résultat (true ou false) et en fonction on affiche le EndScreen avec le texte de qui a gagné.
- On peut aussi recevoir un message indiquant qu'un joueur a quitté, si c'est le cas on ferme l'application.

5.10 Historique des messages

La classe "History" permet de créer et d'afficher une fenêtre de dialogue qui va afficher l'historique des messages envoyés par les joueurs durant la partie. Cette fenêtre est ouvrable via un bouton à droite de leur écran. Elle est utile pour les utilisateurs dans le cas où ils aimeraient revoir d'anciens messages pour se rappeler des dires d'un joueur.

5.10.1 Création

Cette fenêtre de dialogue hérite de la classe "Dialog" de libGDX. Elle est *movable*, ce qui veut dire que l'utilisateur peut la déplacer. Elle n'est pas modale, le joueur peut continuer d'utiliser les autres fonctionnalités de l'application.

5.10.2 Ouverture

Quand l'utilisateur clique sur le bouton historique, il faut modifier le paramètre "Gdx.input.setInputProcessor(stageDialog)". Avec en paramètre le "stage" correspondant à la dialogue, car sinon l'utilisateur ne pourra pas la fermer avec un clic sur le bouton retour.

5.10.3 Ajout des messages

L'ajout de message se fait via une méthode qui est appelée par la GUI qui prend en argument un message. Avec ce message, la méthode crée un *label* qui va être sauvegardé à la fin d'une *Queue List*. Ensuite, la fenêtre de dialogue affiche son message et fait défiler pour que l'utilisateur voie le dernier message.

5.11 Fenêtre de fin de partie

Quand le juge aura rendu son verdict, tous les joueurs seront redirigés sur la fenêtre de fin de partie. Cela va informer les joueurs du choix du juge en affichant si c'est le procureur ou l'avocat de la défense qui a gagné.

Chaque joueur a la possibilité de quitter l'application ou de recommencer une partie qui les redirige directement sur la fenêtre de connexion.

6. Tests

L'application a été testée par un membre de l'équipe. Ces tests permettent de savoir si le jeu est stable ou non et de trouver des bugs. Par manque de temps, ces bugs ne pourront pas être résolus, c'est pourquoi uniquement des hypothèses sur leurs réparations seront rédigées.

- Bugs trouvés
 1. Si une application hôte plante ou se ferme mal. Le serveur local tourne toujours et avec une autre instance on peut rejoindre une partie alors que l'application hôte est fermée. Il est nécessaire de kill l'application dans le gestionnaire des tâches windows.
 2. Si les joueurs décident d'envoyer chacun une pléthore de messages, il se peut qu'une des applications aille planter ou alors que l'un des clients ne puisse plus rien faire.
 3. Le raccourci *enter* n'est pas désactivé lors d'une réception de message et peut donc être spammé
- Hypothèse
 1. Lors d'un crash de l'application, le serveur et le client ne sont pas tués. C'est pourquoi ils tournent toujours en tant que processus de fond. Pour remédier à ce problème, il faudrait appeler la mort du serveur et des clients lorsque l'application se fait fermer de manière inopinée.
 2. Lors d'un envoi trop régulier de données, le serveur n'arrive plus à gérer le broadcast et se ferme de force. Ce qui peut causer une fermeture d'application ou un stop des actions requérant le serveur. Pour régler ce problème, il faudrait mettre en place un système de buffer (BlockingQueue par exemple) afin de mieux traiter l'afflux de message.

Dans le cas où on ne cherche pas à créer des bugs ou à en trouver, l'application permet de jouer au jeu de façon normale sans problème.

7. Résultats

Dans ce projet, le [cahier des charges](#) à été en majeure partie respecté, même si des points importants dans un déroulement normal et propre du jeu manquent. Notamment les preuves qui n'ont pas été implémentées ainsi que les interactions avec témoins / client. Le refus ou la validation de l'objection par le juge ne l'ont également pas été.

Au-delà des points mentionnés plus haut, l'ajout de l'arbre de choix serait un atout majeur qui rendrait l'interaction et l'avancement dans l'affaire beaucoup plus dynamique. De plus, la résolution de bug et l'optimisation des interfaces graphiques et des phases de jeu rendraient l'expérience de jeu plus agréable.

7.1 Résumé des objectifs du module

1. Création d'une application graphique via Java
 - Le jeu Arc Attorney est une application graphique simple et fonctionnelle bien que des améliorations sont les bienvenues. L'objectif est pratiquement atteint.
2. Gérer un projet à plusieurs (planning, suivi, analyse ...)
 - Malgré un problème de retard, le planning et les différentes tâches de la gestion du projet ont été gérés au mieux possible.
3. Rendre un programme stable
 - L'application est stable, mais possède un seul bug capable de faire planter l'application. Cependant, ce bug est difficile à déclencher lors d'une partie normale.

7.2 Résumé des objectifs du projet

1. Créer des interfaces agréables à utiliser avec LibGDX, car Swing est trop rigide.
 - L'objectif est presque atteint, car l'interface du jeu n'est pas très belle et souple.
2. Jeu en multijoueur avec serveur.
 - Objectif rempli, 3 joueurs participent à ce jeu.
3. Trois rôles différents avec chacun leur propre objectif.
 - Objectif rempli, le juge, l'avocat de la défense et le procureur sont implémentés.
4. Jeu interactif avec une histoire de base développée par les joueurs.
 - Objectif rempli, une histoire de base est donnée et les joueurs doivent d'eux-mêmes développer le cours du procès.
5. Interactions avec des témoins / client pour aider les joueurs dans leur avancée.
 - Objectif non atteint, la gestion de témoins n'a pas été implémentée.
6. Objection des avocats pour dynamiser le jeu.
 - Objectif rempli, les avocats peuvent interrompre n'importe qui en émettant une objection.
7. Système de preuve pour que les joueurs puissent appuyer leurs propos.
 - Objectif non atteint, il n'y a pas d'implémentation de preuves.
8. Historique pour que les joueurs puissent relire des événements passés.
 - Objectif rempli, les joueurs peuvent appuyer sur un bouton pour voir l'historique des messages

8. Conclusion

Pour conclure ce projet P2 de deuxième semestre, nous sommes, malgré les problèmes de planning, arrivés à remplir une majorité du [cahier des charges](#).

Le jeu Arc Attorney est stable bien que des bugs restent présents. Malgré une optimisation imparfaite et que certains éléments qui apporteraient une meilleure jouabilité pour les joueurs manquent, le jeu reste totalement jouable et terminable.

Ceci dit, les objectifs du cours ont, eux, été majoritairement remplis.

9. Bibliographie

- [1] Catalin Munteanu, 25.02.18, <https://www.catalinmunteanu.com/design-custom-dialog-libgdx/>
- [2] Creating a simple Chat Client/Server Solution, wachsmut, <http://pirate.shu.edu/~wachsmut/Teaching/CSAS2214/Virtual/Lectures/chat-client-server.html>
- [3] Socket java : Créer une application de chat Client/Serveur, Codeur Java, <http://www.codeurjava.com/2014/11/java-socket-clientserveur-mini-chat.html>
- [4] Socket programming in Java: A tutorial, Steven Haines, 08.01.2020, <https://www.infoworld.com/article/2853780/socket-programming-for-scalable-systems.html>
- [5] Multi-Client Chat Server using Sockets and Threads in Java, Amit Gyawali, 18.11.20, <https://gyawaliamit.medium.com/multi-client-chat-server-using-sockets-and-threads-in-java-2d0b64cad4a7>
- [6] Mans Gezelius, <https://golen.nu/portal/phoenix/>
- [7] Phoenix_Wright GIF, aceattorney.fandom, https://aceattorney.fandom.com/wiki/Phoenix_Wright_-_SpriteGallery?file=PhoenixandDocument2.gif
- [8] Juge GIF, aceattorney.fandom, https://aceattorney.fandom.com/wiki/Judge_-_SpriteGallery?file=JudgeStern2.gif
- [9] Miles Edgeworth GIF, aceattorney.fandom, https://aceattorney.fandom.com/wiki/Miles_Edgeworth_-_Sprite_Gallery?file=AA_Miles_Edgeworth_Court_Smirking_1.gif
- [10] Objection GIF, Medli, <http://fanaru.com/ace-attorney/image/210447/co-coffe-gif/>
- [11] Hammer Logo, FreePick, FlatIcon.com, <https://www.freepik.com>

10. Annexes

- [1] Cahier des charges, <https://gitlab-etu.ing.he-arc.ch/isc/2021/projet-p2-java/arc-attorney/-/wikis/1-Cahier-des-charges>
- [2] Spécification, <https://gitlab-etu.ing.he-arc.ch/isc/2021/projet-p2-java/arc-attorney/-/wikis/2-Sp%C3%A9cification>
- [3] Conception, <https://gitlab-etu.ing.he-arc.ch/isc/2021/projet-p2-java/arc-attorney/-/wikis/4-Conception>
- [4] Journal de travail, Diogo Lopes Da Silva, <https://gitlab-etu.ing.he-arc.ch/isc/2021/projet-p2-java/arc-attorney/-/wikis/Journal-de-travail-Bruno-Costa>
- [5] Journal de travail, Costa Bruno, <https://gitlab-etu.ing.he-arc.ch/isc/2021/projet-p2-java/arc-attorney/-/wikis/Journal-de-travail-Bruno-Costa>
- [5] Journal de travail, Izzo Valentino, <https://gitlab-etu.ing.he-arc.ch/isc/2021/projet-p2-java/arc-attorney/-/wikis/Journal-de-travail-Valentino-Izzo>
- [6] GANTT, Arc Attorney, https://gitlab-etu.ing.he-arc.ch/isc/2021/projet-p2-java/arc-attorney/-/blob/master/Documentation/Gantt_-_Arc_Attorney.xlsx