

Projet LO23

ChessP2P

22/10/12

Auteur(s) :

- Communication et traitements
- Gestion de données
- IHM Connexion
- IHM Grille

Dossier de conception

Réalisation d'un jeu d'échec en réseau décentralisé

Table des matières

| | | |
|-------|---|----|
| 1 | Le projet : Chess P2P | 3 |
| 1.1 | Choix de conception..... | 4 |
| 1.2 | Cas d'utilisation | 5 |
| 2 | Les sous-projets | 6 |
| 2.1 | Communication et traitement | 6 |
| 2.1.1 | Analyse préliminaire..... | 6 |
| 2.1.2 | Choix de conception..... | 7 |
| 2.1.3 | Diagramme de cas d'utilisation | 10 |
| 2.1.4 | Diagrammes de séquence | 11 |
| 2.1.5 | Diagramme de classes..... | 15 |
| 2.2 | Gestion des données..... | 16 |
| 2.2.1 | Documents d'étude | 16 |
| 2.2.2 | Choix de conception | 17 |
| 2.2.3 | Diagramme de classes et interfaces..... | 20 |
| 2.3 | IHM connexion | 21 |
| 2.3.1 | Documents d'étude | 21 |
| 2.3.2 | Cas d'utilisation | 23 |
| 2.3.3 | Diagramme de séquence..... | 24 |
| 2.3.4 | Diagramme de classes et interfaces..... | 34 |
| 2.3.5 | Maquettage..... | 37 |
| 2.4 | IHM grille..... | 39 |
| 2.4.1 | Documents d'étude | 39 |
| 2.4.2 | Choix de conception | 40 |
| 2.4.3 | Cas d'utilisation | 40 |
| 2.4.4 | Diagramme de séquence..... | 41 |
| 2.4.5 | Diagramme de classes et interfaces..... | 44 |
| 2.4.6 | Maquettage..... | 46 |

1 Le projet : Chess P2P

Dans le cadre de l'unité de valeur LO23 à l'Université de Technologie de Compiègne, concernant la conduite de projets informatiques, il nous a été demandé de réaliser un jeu d'échec en réseau. Le projet a pour objet toutes les étapes depuis la conception jusqu'à l'intégration, en passant par la réalisation.

La particularité de ce projet est son nombre important de contraintes. Des contraintes de temps, d'abord, puisqu'il doit être finalisé pour la fin du semestre d'automne 2012. La difficulté, ensuite, vient de l'organisation des acteurs : vingt-trois étudiants doivent travailler en accord pour mener à bien le projet autour d'un découpage structurel imposé en plusieurs sous-projets :

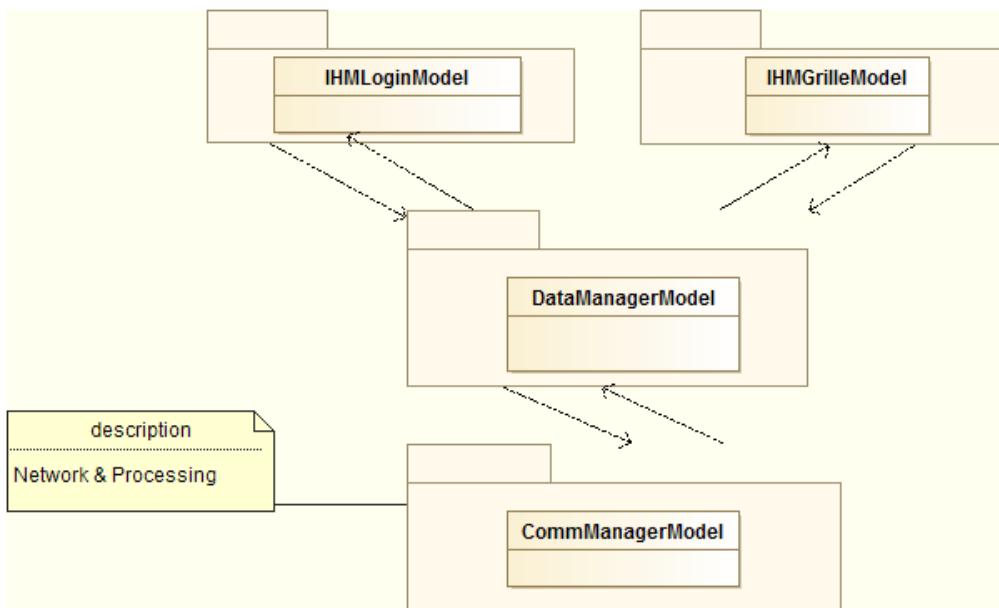
- Communication et traitement des données
- Gestion des données
- Interface humain-machine pour la connexion au jeu
- Interface humain-machine dans le jeu (grille de jeu)

Enfin, il s'agit de respecter les contraintes techniques imposées par le cahier des charges. Le logiciel utilise un réseau décentralisé, ce qui signifie que chaque joueur est propriétaire de ses données. Il a été demandé, de plus, d'intégrer les fonctions suivantes :

- Création du profil
- Connexion et découverte de joueurs connectés
- Lancement d'une partie
- Déroulement d'une partie
- Modification et portabilité du profil
- Sauvegarde d'une partie
- Visionnage d'une partie enregistrée

1.1 Choix de conception

Pour l'architecture globale, nous avons choisi une architecture en étoile : le module « gestion de données » est central et assure la communication avec les autres modules.



D'autre part, toutes les communications entre les différents modules s'effectuent en mode asynchrone afin d'éviter les blocages lorsqu'un module ne répond pas.

1.2 Cas d'utilisation

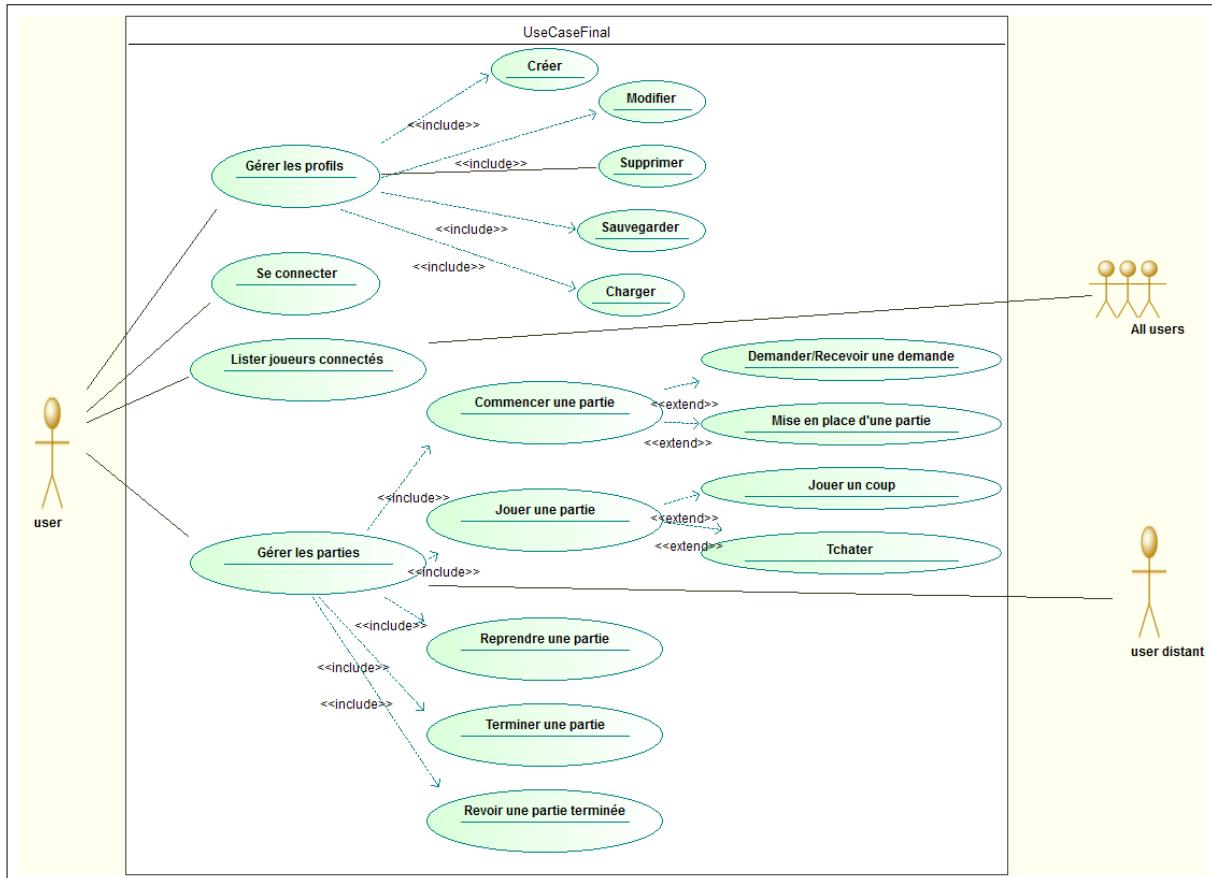


Figure 1 : diagramme de cas d'utilisation global du projet

L'ensemble des équipes a défini un diagramme de cas d'utilisation global. Ce diagramme a été défini sur un plan de haut niveau : pour plus de détails, on ira voir les diagrammes spécifiques à chaque équipe. L'intérêt de ce travail a été de définir plus précisément le rôle de chaque équipe tout en ayant une vision globale sur le projet général.

2 Les sous-projets

2.1 Communication et traitement

Ce module a la tâche de gérer l'acheminement des données dans le réseau décentralisé. Le but est donc de définir la logique d'envoi et de réception des messages dans les différentes situations imposées par les fonctionnalités offertes par le logiciel.

2.1.1 Analyse préliminaire

Dans les prémisses du projet, notre équipe a réfléchi sur l'impact de la logique de communication au cours des différentes situations d'utilisation du jeu. Nous avons ainsi déterminé qu'il y avait deux cas à traiter : la communication joueur à joueur au cours d'une partie et la découverte des joueurs.

a) La communication avec l'adversaire

Il s'agit d'une communication bidirectionnelle entre deux joueurs. Dans cette situation, on doit pouvoir être informé que le message réseau a bien été reçu par le destinataire. Par exemple, si on envoie un message indiquant qu'un joueur a déplacé une pièce, il est primordial qu'il sache que l'autre joueur a reçu son message et que maintenant c'est à lui de déplacer une pièce. Il faut donc utiliser un protocole en mode connecté.

b) La découverte des joueurs

Pour la découverte des joueurs sur le réseau, il faut pouvoir envoyer un message à tous les joueurs disponibles. Deux solutions sont envisageables : soit en mode « broadcast », qui consiste à envoyer un message à toutes les machines d'un réseau, soit en mode « multicast », qui permet d'envoyer un message à un groupe de machines. Ces formes particulières de communication imposent d'utiliser un protocole en mode non-connecté car la communication est unidirectionnelle : le message part d'une machine pour arriver à un ensemble de machines. Dans le cadre de notre projet, une machine A enverrait d'abord un message à un ensemble de machines B pour demander de se présenter (profil + disponibilité) puis chaque machine B répondrait à la machine A en envoyant les informations nécessaires.

2.1.2 Choix de conception

Durant la conception, l'équipe a été amenée à effectuer de nombreux choix importants parfois décisifs pour le bon déroulement du projet. Voici donc, en détails, les principales questions que s'est posée l'équipe, accompagnées des solutions possibles, d'arguments puis de la décision finale.

a) Objet brut ou objet intermédiaire ?

L'équipe s'est confrontée à un choix de conception dès la première séance. En effet, concernant l'envoi des données, la question était de savoir si les données envoyées à travers le réseau allaient être simplement celles offertes par le module *Gestion de données* à travers leur interface ou si notre équipe décidait de convertir cette structure en une autre plus adaptée à notre module et au transfert à travers le réseau.

Dans le premier cas, on récupère l'objet du module *Gestion de données* à travers l'interface liant nos deux modules et on décide de l'envoyer directement à travers le réseau, puis de le récupérer (au sein d'une autre application cliente) et de le rendre au module *Gestion de données*.

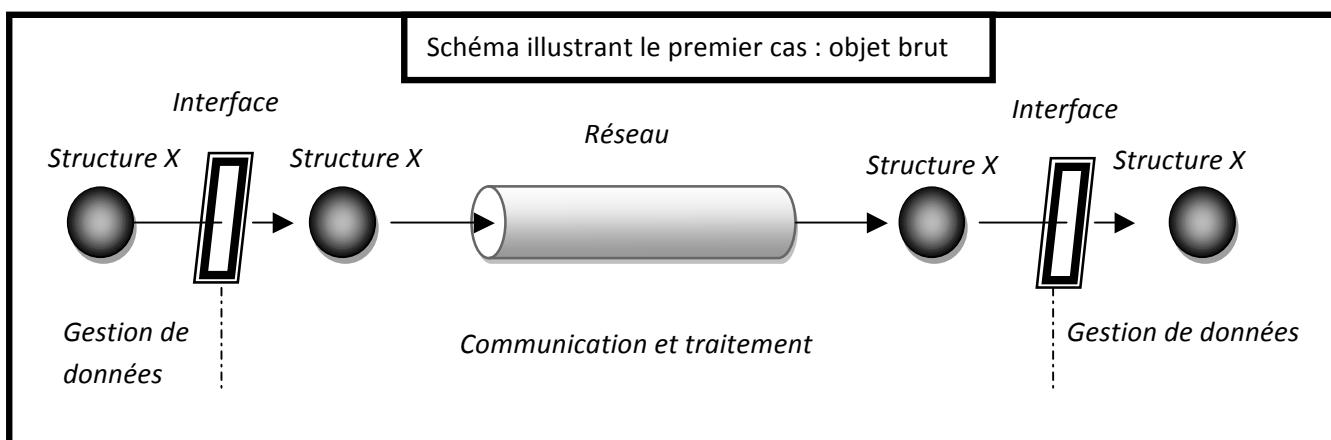


Figure 2 : cas d'un objet brut

Dans le deuxième cas, on récupère l'objet du module *Gestion de données* à travers l'interface et on décide de créer notre propre structure, de récupérer les paramètres indispensables et de les injecter dans notre objet. Ensuite, on peut alors envoyer l'objet nouvellement créé à travers le réseau, le récupérer, le reconvertir en un objet manipulé par le module *Gestion de données* avant de leur rendre à travers l'interface.

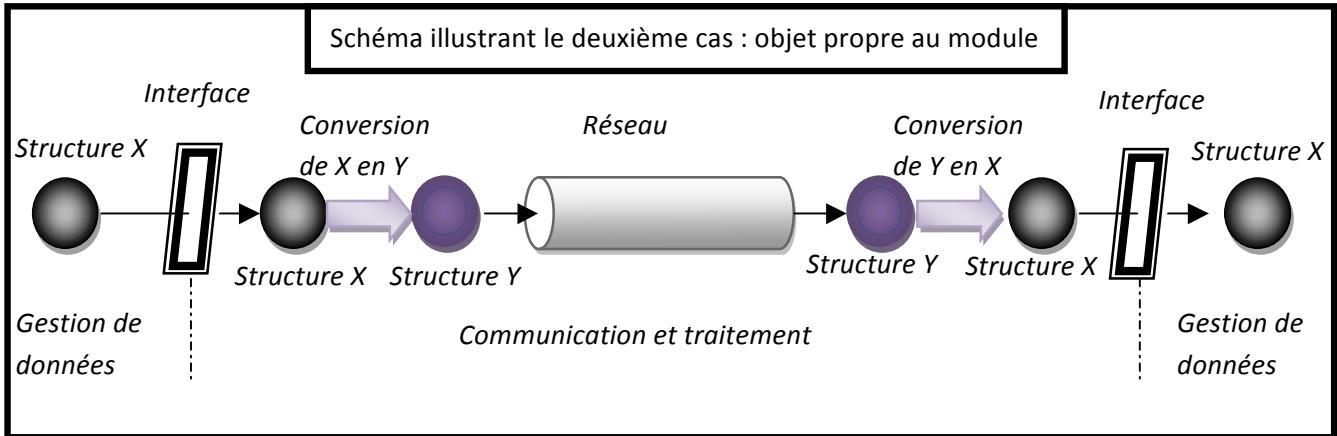


Figure 3 : cas d'un objet propre au module

L'avantage évident de la première solution est sa clarté et sa simplicité. Néanmoins, après réflexion, on se rend compte que cette solution lie de façon dérangeante notre équipe à l'avancement du module Gestion de données. En effet, dans ce premier cas, on utilise leur objet ; on se doit donc d'attendre que celui-ci soit finalisé pour pouvoir développer le module de communication à travers le réseau.

La deuxième solution permet, quant à elle, de gagner en indépendance par rapport à l'autre équipe (responsable de la gestion de données). On peut alors concevoir toute la partie envoi et réception de données à travers le réseau en se basant sur la donnée proprement créée pour transiter à travers le réseau. Une fois que le module Gestion de données nous fait parvenir la structure qu'elle veut faire transiter, il suffit alors à notre équipe de concevoir les 2 convertisseurs (un dans chaque sens) et la liaison sera opérationnelle. La seconde approche est, pour résumer, plus modulaire et apporte donc de la souplesse même si elle complexifie sensiblement le processus en passant par une structure intermédiaire pour le transfert à travers le réseau. Un autre avantage à la deuxième solution est le fait que cette structure intermédiaire ne contiendra que les informations indispensables contrairement au premier cas où l'on envoie l'objet entier d'origine.

L'équipe a donc choisi la deuxième solution (création d'un objet intermédiaire pour le transit) pour des raisons de souplesse principalement et donc pour conserver une certaine autonomie.

b) Structure globale

Concernant l'architecture globale de notre application, nous nous sommes entretenus à ce sujet, en premier lieu, avec le groupe relatif au module *Gestion de données*. Selon nous, il est plus propre et plus structuré de centraliser les communications inter modules autour de celui relatif à la gestion des données. En effet, dans le cas contraire, les modules IHM auraient de temps à autres court-circuité le module *Gestion de données* afin de s'adresser à nous directement, ce qui obligeait à prévoir une interface faisant à la fois référence à des fonctions appelées par le module *Gestion de données* ainsi que ceux liés à l'IHM. Cela ne nous semblait pas très propre d'un point de vue structurel. A cet effet, après diverses réunions avec l'intégralité des équipes, nous avons adopté le schéma structurel ci-après.

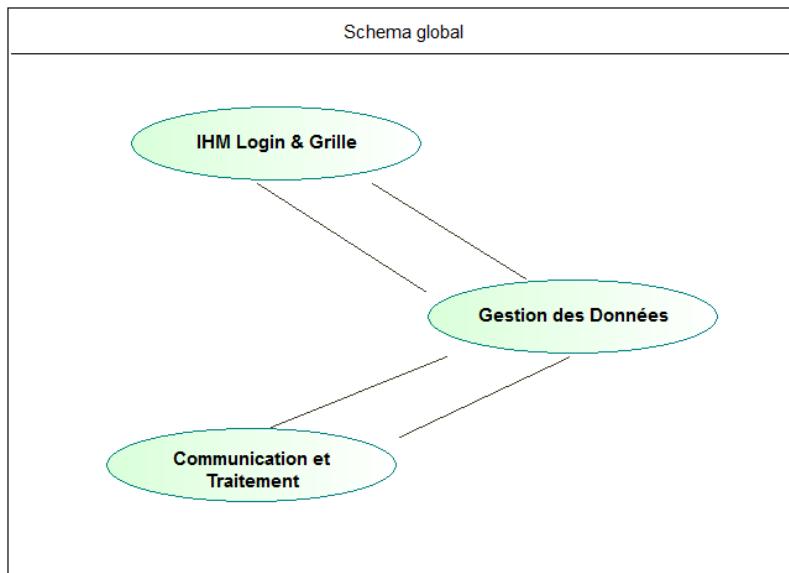


Figure 4 : schéma structurel des équipes

c) Interface

Concernant l'interface que nous allons fournir au module *Gestion de données*, nous avions le choix entre une seule méthode générale possédant de multiples paramètres en fonction du contenu des messages à transmettre ou l'utilisation de plusieurs méthodes différentes. Le module partenaire ne s'étant pas prononcé, nous avons décidé de fournir diverses méthodes afin de clarifier la structure des communications inter modules. De plus, ce choix nous permettra d'aisément répartir le travail de développement.

2.1.3 Diagramme de cas d'utilisation

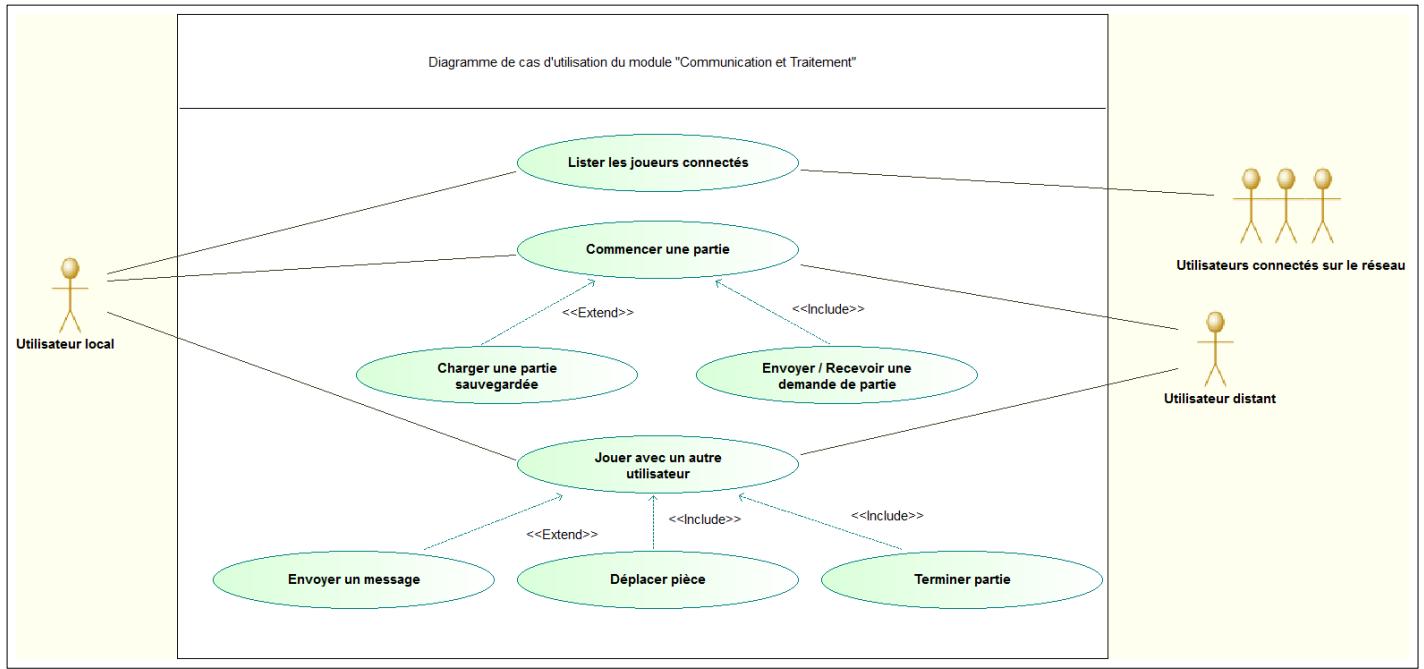


Figure 5 : diagramme de cas d'utilisation du module Communication et Traitement

Suite à la mise en place de ce diagramme, nous avons donc décidé d'étudier les diagrammes de séquence suivants :

- Lister les joueurs connectés
- Envoyer / Recevoir une demande de partie (chargement inclus)
- Envoyer un message
- Envoyer un statut
- Déplacer pièce
- Déconnexion

2.1.4 Diagrammes de séquence

a) Lister les joueurs connectés

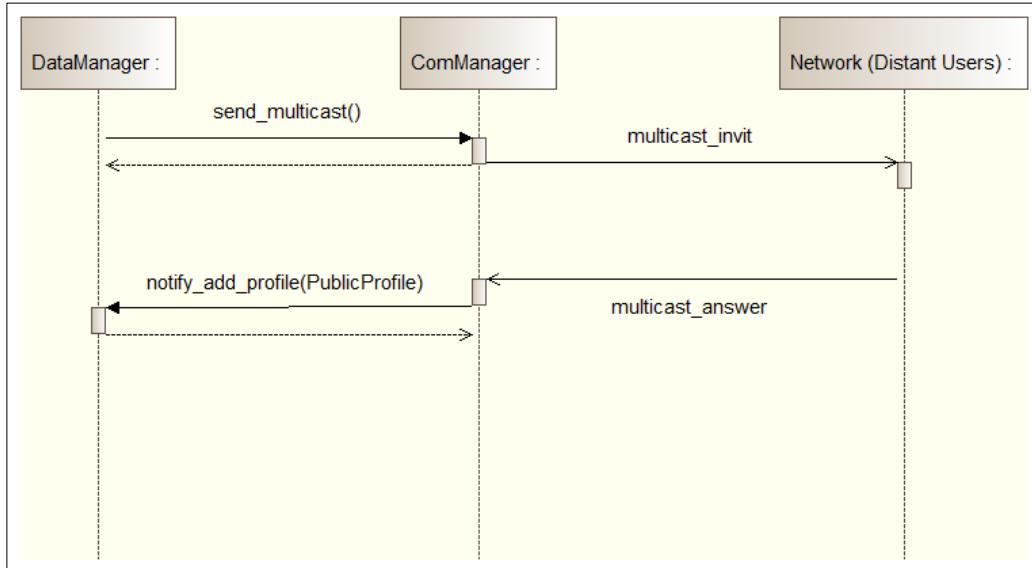


Figure 6 : diagramme de séquence de recherche des joueurs

On envoie un multicast sur le réseau local afin d'envoyer les utilisateurs détectés au « DataManager ».

b) Envoyer / Recevoir une demande de partie (chargement inclus)

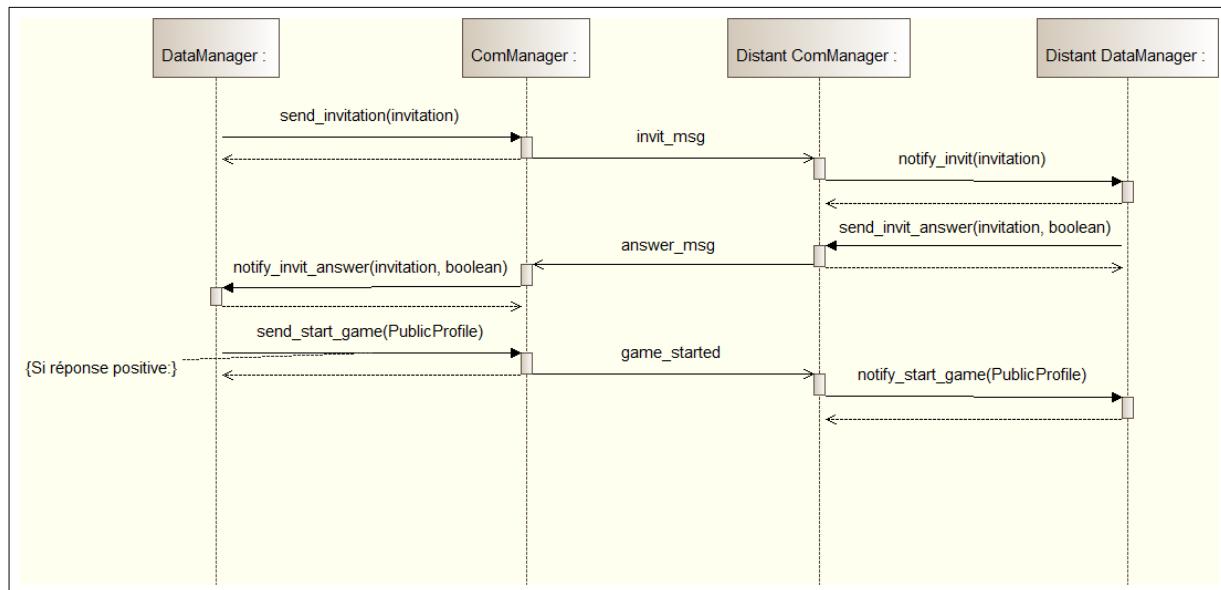


Figure 7 : diagramme de séquence de lancement d'une partie

Ici, on note que les requêtes sont envoyées de manière asynchrone, ce qui implique l'utilisation d'une pile de demandes afin de pouvoir permettre à l'utilisateur de répondre à celle de son choix. Ceci nous permettra également de gérer le cas de demandes de connexion simultanées. Néanmoins, lorsqu'une demande de connexion est acceptée, il faudra bien veiller à ce qu'un « flag »

ou autre marqueur bloque l'acceptation de toute autre demande. On note également que ce scénario gère les demandes de nouvelles parties ainsi que les demandes de chargement de parties. A cet effet, la structure invit_msg contiendra soit des éléments relatifs à une nouvelle partie, soit une partie sauvegardée localement qui sera envoyé à l'adversaire concerné.

c) Envoyer un message

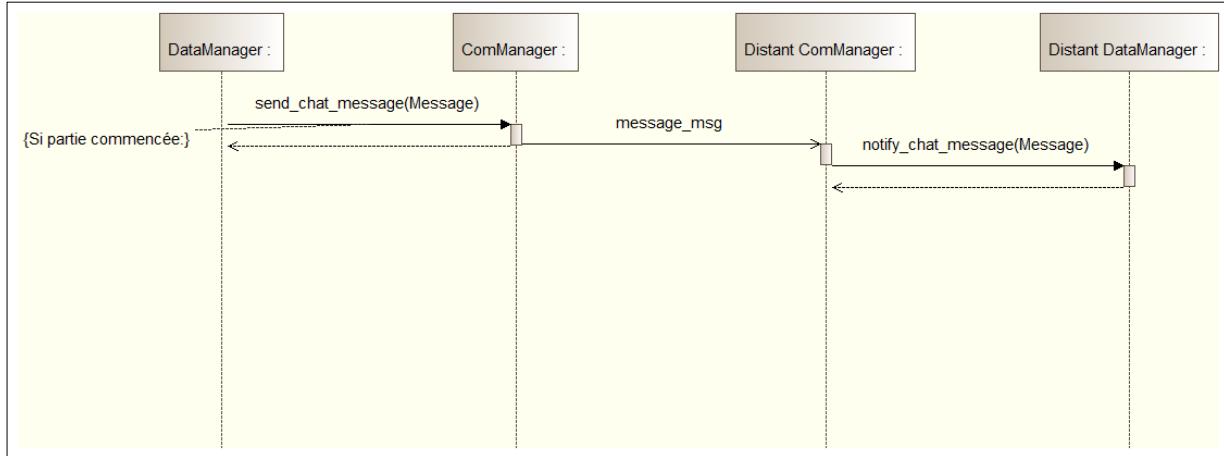


Figure 8 : diagramme de séquence d'envoi d'un message par chat

d) Envoyer un statut

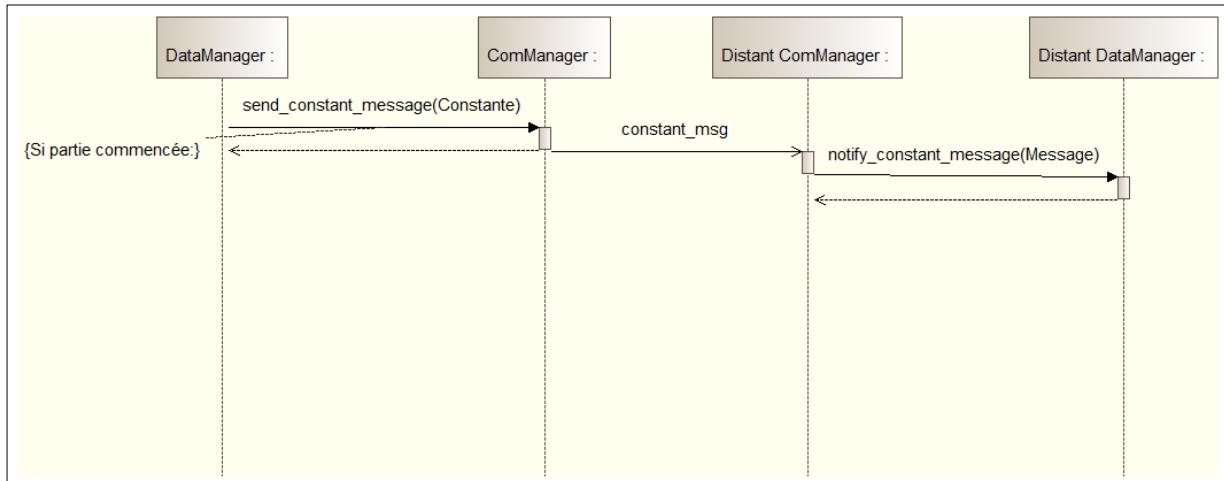


Figure 9 : diagramme de séquence d'envoi d'un message par chat

On envoie ici le statut de la partie. En effet, lorsqu'un joueur demande un match nul, qu'il abandonne ou qu'il quitte la partie, un message de type constant_msg est envoyé pour prévenir le client adverse.

e) Déplacer pièce

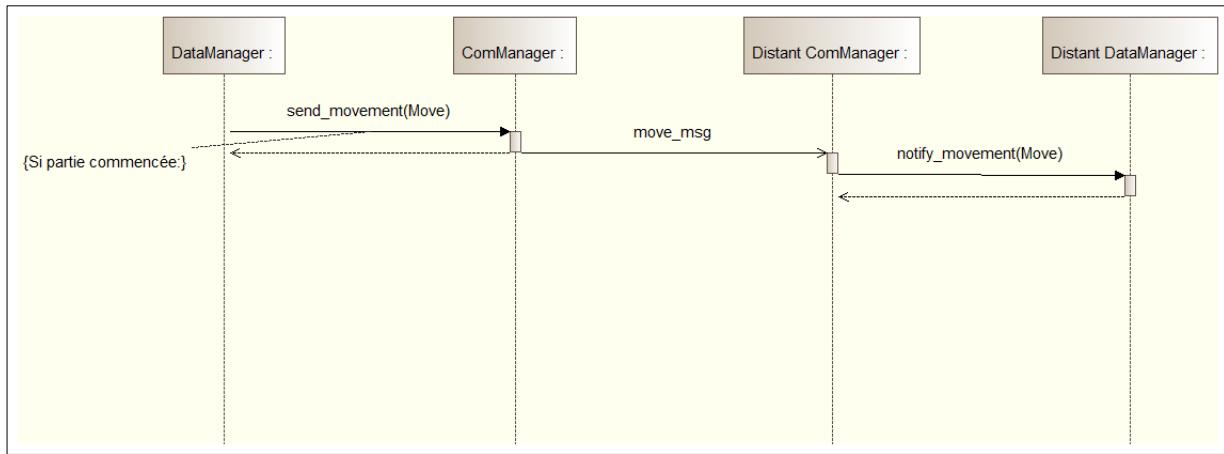


Figure 8 : diagramme de séquence de déplacement d'une pièce

On transmet le déplacement au joueur adverse par le biais de la structure `move_msg`.

f) Déconnexion de partie

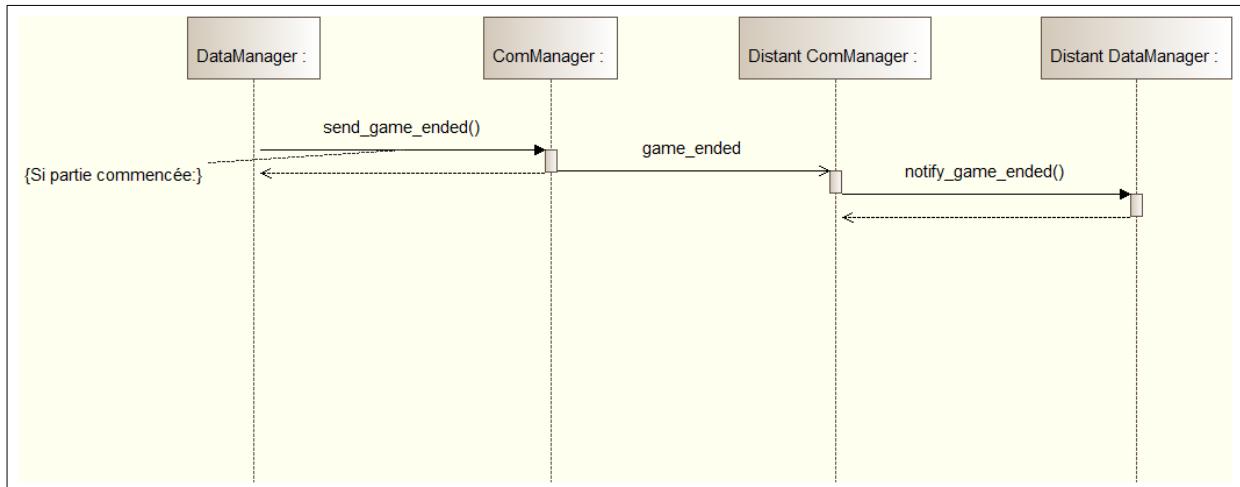


Figure 9 : diagramme de séquence de fin de partie

Une fois la partie finie ou si un des joueurs décide de quitter la partie, un message est envoyé à l'adversaire afin de lui notifier cette décision.

g) Interface

Etant donné que notre module communique uniquement avec le module *Gestion de données*, voici l'interface que nous avons définie avec eux :

| | ISend | IReceive |
|--|--|--|
| Lister les joueurs connectés | Send_multicast() | Notify_add_profile(PublicProfile) |
| Invitation Nouvelle partie / Chargement | Send_invit(Invitation) Send_invit_answer(Invitation, boolean) | Notify_invit(Invitation) Notify_invit_answer(Invitation, boolean) |
| Début de partie Connexion | Send_start_game(PublicProfile) | Notify_start_game(PublicProfile) |
| Chat | Send_chat_message(Message) | Notify_chat_message(Message) |
| Fin de partie (Match nul, se render...) | Send_constant_message(Constant) | Notify_constant_message(Constant) |
| Jouer | Send_movement(move) | Notify_movement(move) |
| Deconnexion | Send_game_ended() | Notify_game_ended() |

Les fonctions situées dans la colonne « ISend » sont celles qui seront implémentées par notre module ; celles situées dans la colonne « IReceive » seront implémentées par le module *Gestion de données*. A cet effet, nous allons développer une interface en JAVA afin que le module *Gestion de données* puisse commencer à développer et vice versa.

2.1.5 Diagramme de classes

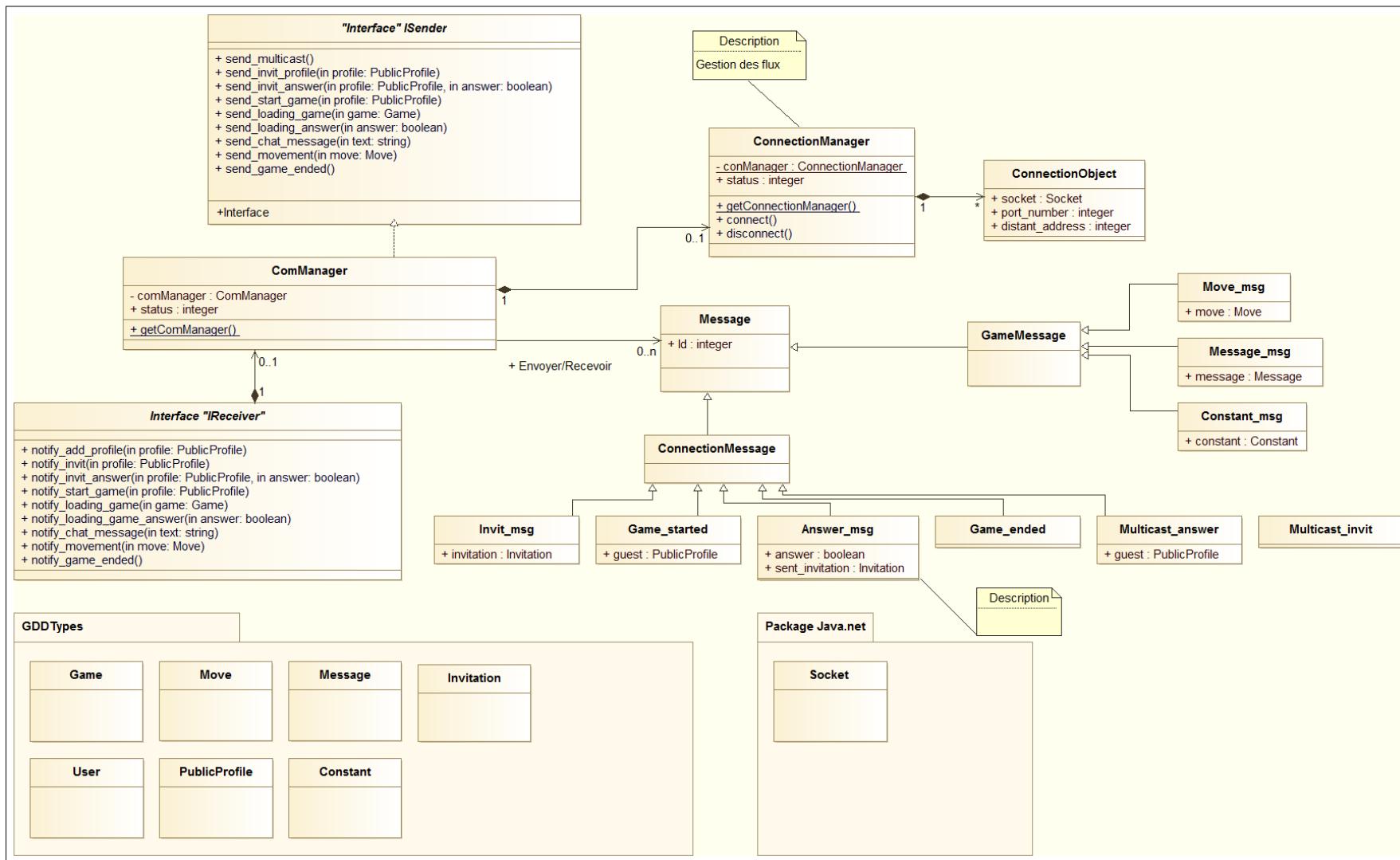


Figure 10: diagramme de classes du module Communication et Traitement

2.2 Gestion des données

2.2.1 Documents d'étude

Le module gestion de données a pour but de mettre en place toute la structure de données permettant de cadrer les échanges entre les différents modules. L'objectif est de permettre la manipulation des données. Ces données se séparent en deux grands ensembles : les profils et les parties.

Dans la gestion du profil, on retrouve les données suivantes :

- Le profil de l'utilisateur courant (privé et public).
- La liste des profils des joueurs connectés.
- La liste des invitations (émises ou en cours).

Dans la gestion des parties, on retrouve les données suivantes :

- La liste des parties en cours ainsi que leur historique.
- La liste des pièces sur l'échiquier d'une partie.
- Le chat lié à une partie.
- Les joueurs d'une partie.

Pour manipuler ces données, nous avons opté pour une architecture à base de "Manager" (gestionnaire). Ces managers sont des classes qui permettent la manipulation des objets du modèle à commencer par le "CRUD" (Create, Read, Update, Delete).

Ces managers (un pour les profils et un pour les parties) sont les interfaces de la gestion des données avec les autres modules du projet. Leurs méthodes ont donc été conçues en étroite collaboration avec les autres.

La dernière partie de la gestion des données est la nécessité d'assurer la persistance des données. Pour cela nous avons réalisé une étude sur la meilleure façon d'exécuter cette tâche.

Nous avons aussi choisi de ne pas utiliser de design pattern de Singleton pour les managers. Cependant, ces derniers seront tous instanciés dans la classe ApplicationModel et par la même, n'auront qu'une instance unique accessible à tout moment.

Enfin, nous avons aussi travaillé plus en détail avec la gestion des connexions réseaux afin de mettre au point l'interface que nous utiliserons pour transmettre des données aux autres joueurs.

Après avoir présenté globalement le travail de conception du groupe, détaillons les choix de conceptions.

2.2.2 Choix de conception

a) Le profil et la gestion des profils

[NB : les classes du diagramme concernées par cette partie sont : Profile, PublicProfile, Invitation, NewInvitation, ResumeGame, ProfileManager].

La classe Profile représente un profil utilisateur. À noter que chaque Profile est unique grâce à un identifiant généré (c'est un autre groupe du projet qui s'occupe de la génération de l'identifiant). Chaque objet Profile possède son pendant public, un objet PublicProfile qui n'est autre qu'un objet Profile amputé du mot de passe. C'est l'objet PublicProfile qui sera transmis aux autres joueurs pour des raisons de sécurité.

Plusieurs solutions pour la représentation du PublicProfile ont été envisagées :

- une classe qui contient une instance d'une classe Profile et dont les getters font référence aux attributs de l'objet Profile (1)
- une sous-classe spécialisant la classe Profile en redéfinissant le "getter" de l'accès à l'attribut password (2)
- une nouvelle classe qui serait instanciée à partir des valeurs d'un objet Profile (3)

Les solutions (1) et (2), bien que plus élégantes en apparence, ne sont pas sûres : dans les deux cas l'attribut password existe en mémoire et peut-être lu.

La troisième nous a semblé meilleure puisqu'elle résout le problème initial sans défaut.

Les invitations permettent aux joueurs d'interagir, c'est-à-dire d'inviter un joueur à commencer une partie ou d'inviter un joueur à continuer une partie. On aurait pu ne faire qu'une classe et utiliser un champ "type" mais la différence des attributs entre les classes NewInvitation et ResumeGame justifie clairement un héritage.

Le Manager de Profils, le ProfileManager, permet à la fois d'assurer le "CRUD" des objets profils mais aussi d'envoyer et recevoir des invitations. C'est aussi lui qui assure la connexion du joueur à son profil lors du lancement du jeu et qui est chargé de récupérer la liste des joueurs connectés sur le réseau.

b) Les parties et la gestion des parties

[NB : les classes du diagramme concernées par cette parties sont : Player, Game, GamePiece et ses sous-classes, Event et ses sous-classes, GameManager]

Les parties sont représentées par trois sous-ensembles : le représentant du joueur, la grille d'échec et les événements.

Le représentant du joueur est la classe Player. C'est en quelque sorte l'instance du Profile pour la partie. La classe Player possède les attributs concernant la couleur et le timer du joueur dans la partie. Nous avons trouvé cette solution plus élégante et efficace que de mettre les notions de couleur et timer directement dans la classe Game pour chaque joueur (ce qui aurait été possible) sous forme de blackPlayer, whitePlayer, blackPlayerTimer, etc....

La grille d'échec n'est pas représentée sous la forme d'une classe. A l'origine c'était le cas mais après consultation il est apparu que cela alourdissait le diagramme pour un bénéfice nul (à l'exception de la

sémantique d'avoir un objet Grille). Elle est à présent représentée sous la forme d'une double collection d'objets de type GamePiece.

GamePiece est une classe abstraite qui représente une pièce. Sa méthode getPossibleMoves() est aussi abstraite et devra être implémentée par ses sous-classes représentant le Roi, la Reine, les Tours, les Fous, les Cavaliers et les Pions. L'héritage était ici une évidence à cause de la redéfinition de cette méthode.

Certaines de ces sous-classes possèdent des attributs supplémentaires, par exemple pour savoir si le Roi a été mis en échec ou s'il a déjà bougé. Ces attributs seront utilisés pour implémenter des règles spéciales des échecs par le module IHM Grille.

Les événements permettent trois choses : transmettre les coups joués, l'état de la partie et envoyer des messages textuels à l'autre joueur. C'est le rôle de la classe Event et respectivement des classes Move, Constant et Message.

En gardant les événements on peut aussi enregistrer le déroulement d'une partie et ainsi obtenir la notion de "replay" voulue par le client.

Ces différents éléments sont centralisés dans la classe Game qui contient des indications sur la durée de la partie et des méthodes pour lancer/arrêter les timers ainsi qu'une méthode d'initialisation de la grille d'échec.

Enfin le GameManager, comme son homologue pour les profils permet avant tout le CRUD de parties et d'événements. C'est aussi lui qui sera chargé de "jouer un coup" grâce à la méthode playMove. En combinant les méthodes getHistory() et playMove() on peut facilement obtenir le replay de la partie.

c) La persistance

La persistance ou sérialisation est un mécanisme technique qui permet de stocker les objets dans des fichiers (on aurait aussi pu imaginer une base de données mais cette solution n'est pas adaptée dans notre cas).

Nous avons eu une réflexion sur le format de la persistance (XML ? JSON ? binaire ?) qui a donné lieu à une étude technique.

Au début, nous nous étions mis d'accord sur une persistance en JSON (privilégié par rapport à XML pour sa plus grande concision) puisque ce format aurait permis une évolution de l'application plus intéressante (réutilisation par des terminaux mobiles par exemple) par rapport à une persistance binaire.

Bien qu'il existe des librairies très puissantes pour la sérialisation en JSON (par exemple google-gson, développée par Google : <http://code.google.com/p/google-gson/>), nous avons finalement privilégié la sérialisation Java native (binaire). En effet, celle-ci est la seule à supporter les références circulaires dans les objets sérialisés. Étant donné que supprimer les références circulaires dans notre graphe d'objet aurait représenté un travail important, nous avons décidé que le jeu n'en valait pas la chandelle.

La seule contrainte de ce mode de persistance est la nécessité de l'implémentation de l'interface Java Serializable. Cette dernière ne nécessitant aucune implémentation de méthode, la contrainte est extrêmement faible.

De plus, nous avons du réfléchir à la mise en place de la persistance (méthode de sauvegarde/chargement dans les objets vs objet externe chargé de la persistance).

Comme nous avons opté pour une architecture données/manager, il nous a semblé naturel de respecter le même schéma pour la persistance, pour des raisons de cohérence. Une classe (le Serializer, non présenté sur le diagramme de classe dans un souci de clarté, ce dernier n'apportant rien à la compréhension du problème) aura donc la responsabilité de faire sauvegarder et charger les objets dans des fichiers.

2.2.3 Diagramme de classes et interfaces

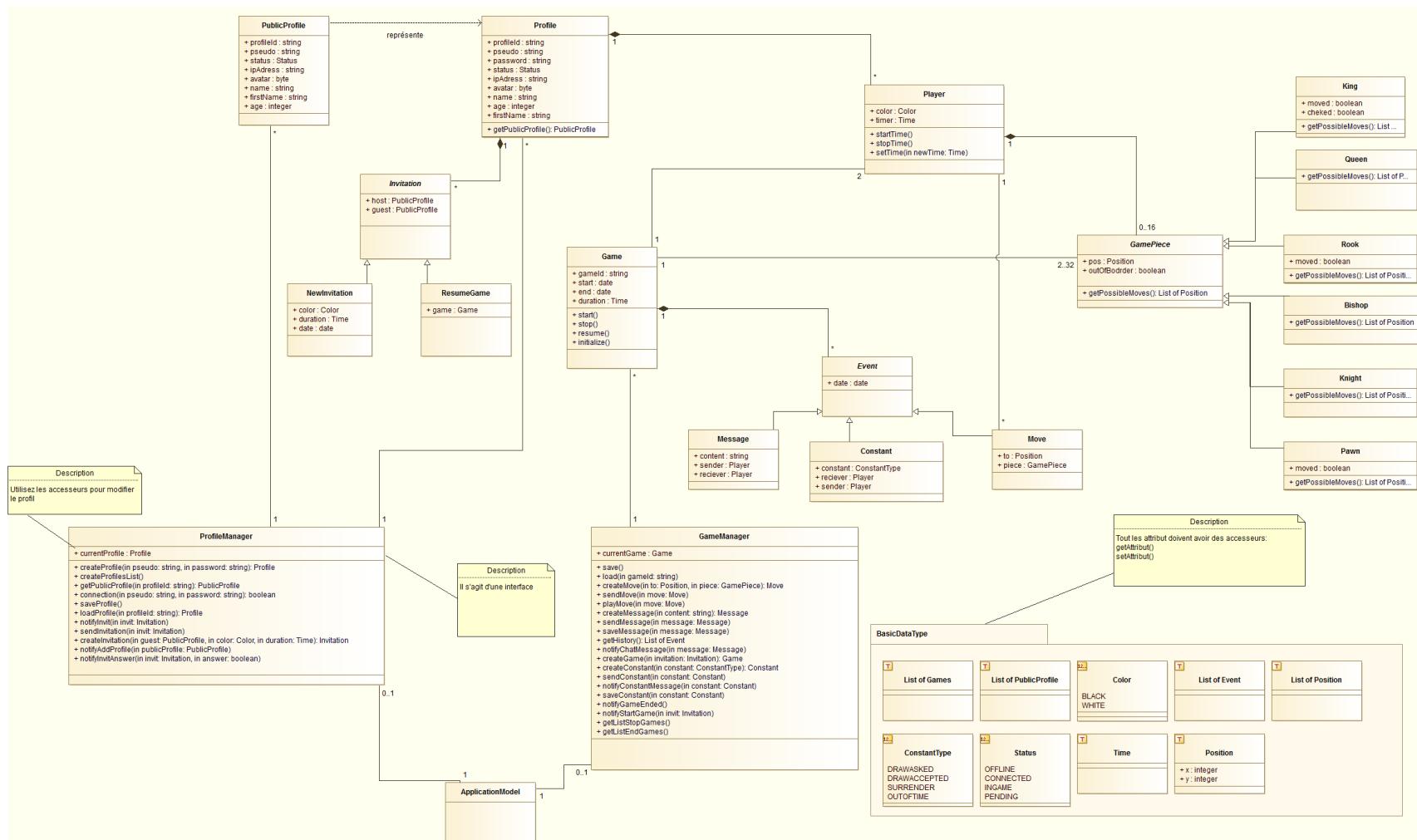


Diagramme de classe obtenu suite aux différents choix de conception

2.3 IHM connexion

L'objectif de ce module du projet est de réaliser l'interface Homme-Machine de la connexion du joueur à l'application p2p-chess. Cette connexion à l'application nécessite plusieurs fenêtres différentes qui se succéderont selon les actions de l'utilisateur. Nous pouvons compter parmi ces fenêtres :

- Une fenêtre de connexion
- Une fenêtre de création de profil
- Une fenêtre montrant la liste des joueurs connectés
- Une fenêtre de modification et d'export du profil

Cet IHM doit être ergonomique et confortable pour l'utilisateur. De plus, la position des boutons doit rendre la prise en main de l'application très simple et compréhensible rapidement.

Par rapport aux autres modules, celui-ci se positionne au tout début, lorsque l'utilisateur lance l'application. Une fois qu'un joueur décide de rentrer dans une partie, l'IHM grille prend le relais. La réception et la vérification des données s'effectuent grâce aux modules Communication et DataManager.

Nous allons donc présenter le dossier de conception de l'IHM connexion dans la suite de ce rapport.

2.3.1 Documents d'étude

a) Java

Nous avons choisi d'utiliser le langage Java pour implémenter notre application. En effet, Java est un langage moderne permettant la programmation orientée objet ce qui facilite la programmation de composants modulaires. Ce langage est connu par la majorité des membres de l'équipe et il est facile de l'apprendre pour ceux qui connaissent déjà un langage orienté objet comme le C++. Utilisé massivement dans l'industrie, Java a fait ses preuves comme étant un langage robuste et permettant la collaboration entre plusieurs équipes grâce au concept d'interface qui permet la communication entre les différentes parties du code.

b) Netbeans

Nous avons choisi d'utiliser l'IDE Netbeans pour coder notre application. Ce choix s'est fait en le comparant à Eclipse, l'éditeur de choix pour coder du Java il y a quelques années. De nos jours, il semble que Netbeans soit moins lourd dans son interface tout en gardant les mêmes fonctionnalités. De plus, la majorité des personnes de l'équipe connaît ce logiciel ce qui a permis une prise en main simplifiée. Même s'il est moins lourd qu'Eclipse, il supporte quand même des fonctionnalités avancées comme le renommage de variable dans tout le code (*refactoring*), l'autocomplétion, l'intégration à des logiciels de gestion de version comme SVN ou Git ou l'ajout de getters/setters automatique.

c) Swing

Nous avons choisi d'utiliser la librairie Swing pour créer l'interface. Cette librairie est intégrée à J2SE et permet un développement d'interfaces rapide et multi-plateformes car elle est écrite complètement en Java. Cela lui permet d'avoir le même comportement sur plusieurs plateformes différentes au prix de performances un peu moindres qu'avec Abstract Window Toolkit dont les composants sont écrits directement pour une plateforme spécifique. Cette perte de performance est très faible et est apparemment imperceptible pour les utilisateurs finaux. Swing permet d'utiliser l'architecture MVC où les données et leur représentation sont découplées ce qui permet une plus grande modularité et une plus grande séparation des préoccupations. Elle utilise la programmation par événements pour signaler des changements du modèle. Par exemple si la liste des joueurs change, un signal sera envoyé à Swing qui "repeindra" le composant qui dépend de cette liste (modèle).

Il est possible d'utiliser plusieurs threads avec Swing ce qui est très important pour notre application qui devra avoir un thread pour écouter sur le réseau et un thread pour l'interface graphique. De plus, Swing est intégré à Netbeans ce qui permet de prototyper rapidement des interfaces grâce au designer d'interface. Il permet d'utiliser une interface WYSIWYG pour coder les différentes GUI.

d) Mockup Screens

Nous avons choisi de réaliser les maquettes de l'IHM Login à l'aide du logiciel Mockup Screens. Il existe de nombreux logiciels permettant de réaliser des maquettes d'interfaces (exemple : balsamiq, mockflow...) mais ceux-ci sont généralement utilisables directement en ligne. Nous avons donc dans un premier temps testé ces différentes applications en ligne pour nous rabattre vers un logiciel installé localement. Cela assure ainsi une meilleure interface, une meilleure fluidité de l'application ainsi qu'un meilleur rendu.

2.3.2 Cas d'utilisation

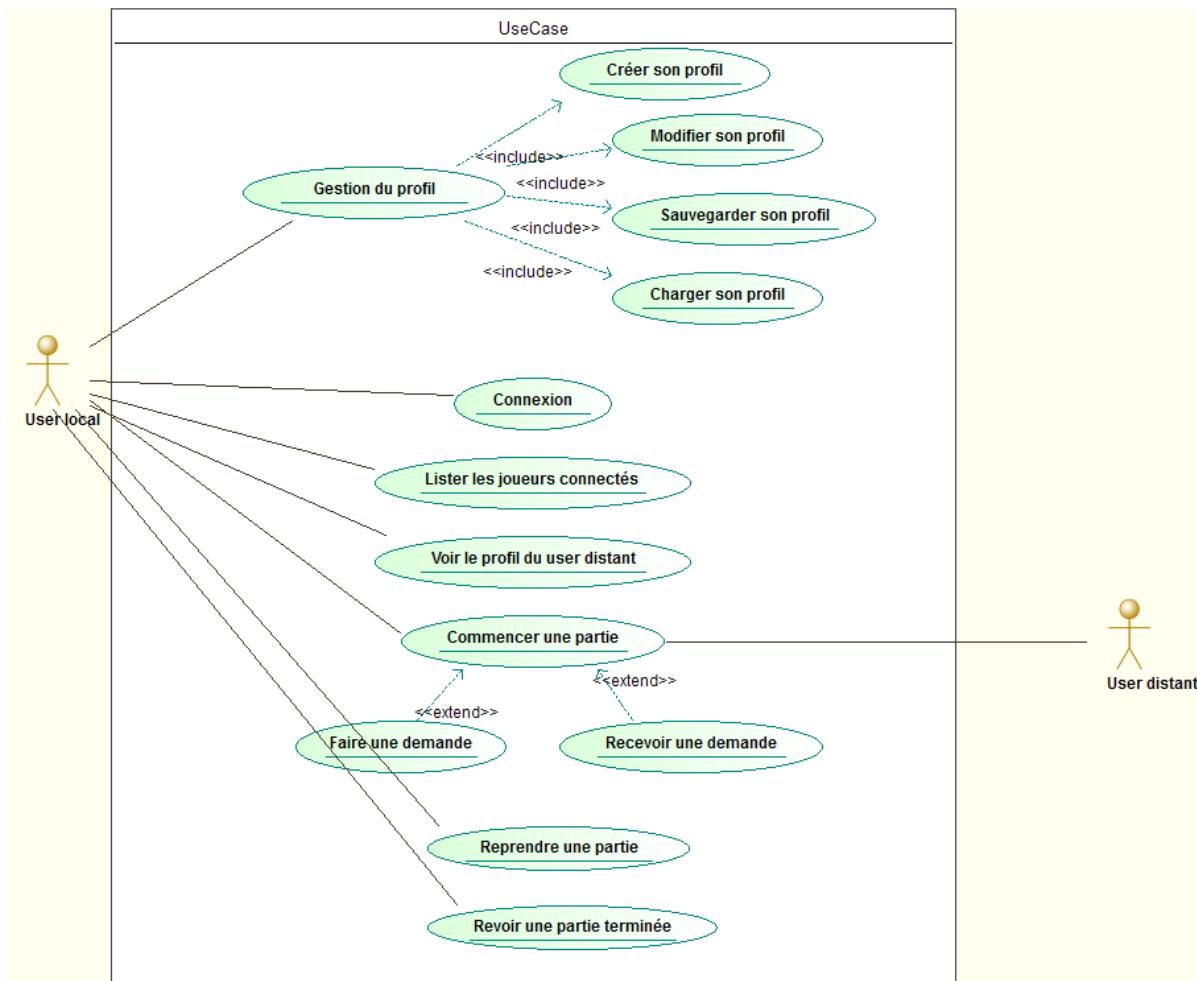


Figure 1 : Use Case spécifique du module IHM-Login

Gestion du profil

L'utilisateur peut gérer son profil après sa création. Plusieurs fonctionnalités seront proposées : modification du profil, chargement d'un profil (importer) et sauvegarde de son profil en local (exporter).

Connexion

L'utilisateur renseigne son pseudo et son mot de passe pour se connecter dans l'application. Il peut aussi décider de charger un autre profil.

Lister les joueurs connectés

Une fois l'utilisateur connecté, il aura accès à la liste des joueurs connectés ainsi qu'à leurs statuts de disponibilité.

Voir le profil du joueur distant

L'utilisateur peut visualiser le profil de l'ensemble des joueurs connectés, des informations telles que le nom, le pseudo, l'âge, l'avatar, ...

Commencer une partie

L'utilisateur peut demander à jouer une partie contre un joueur adverse "disponible". Il peut de même recevoir une demande de la part d'un des joueurs connectés.

Reprendre une partie

L'utilisateur peut reprendre une partie avec son adversaire, dans le cas où la partie a été interrompue et sauvegardée.

Revoir une partie terminée

L'utilisateur peut revoir une partie terminée, dans le cas où la partie a été sauvegardée.

2.3.3 Diagramme de séquence

a) Créer un profil

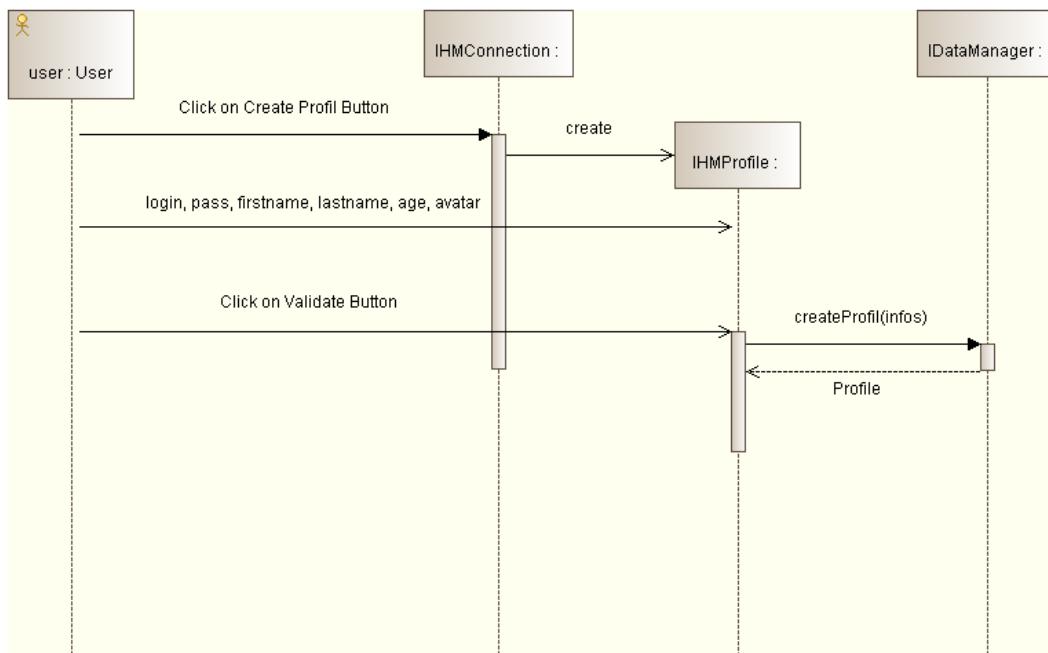


Figure 2 : Séquence d'état de la méthode « Crée un profil »

Ce diagramme illustre le fait qu'un utilisateur peut créer un profil.

b) Modifier un profil

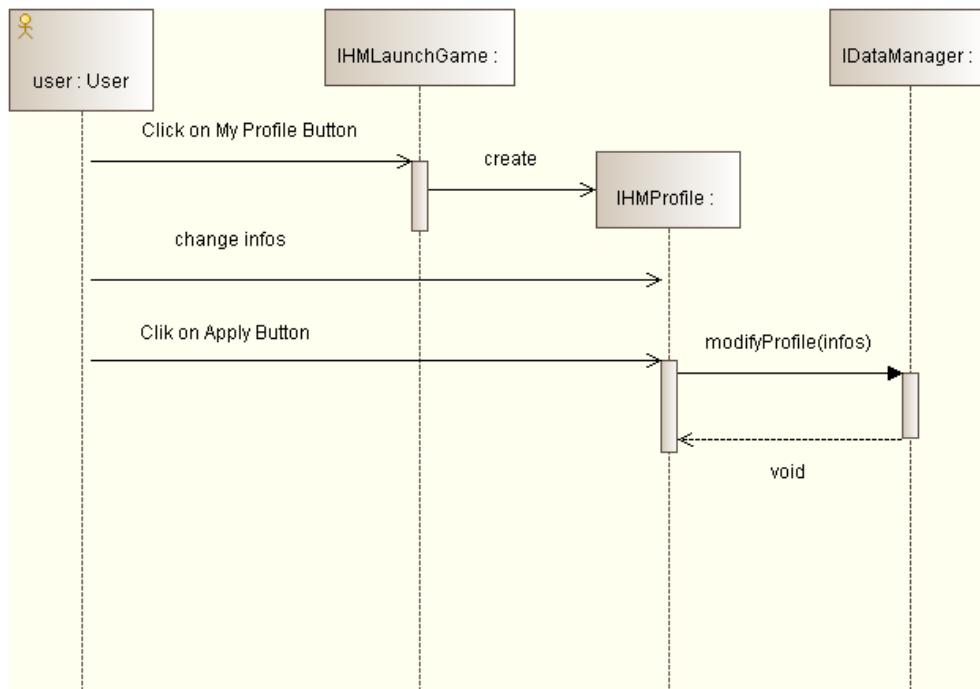


Figure 3 : Séquence d'état de la méthode « Modifier un profil »

c) Sauvegarder un profil

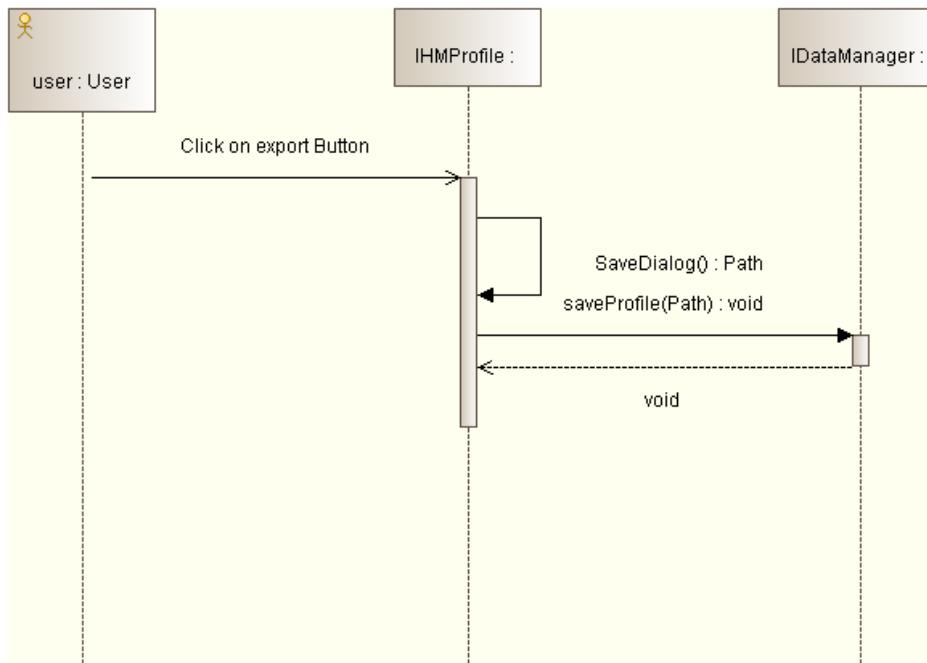


Figure 4 : Séquence d'état de la méthode « sauvegarder un profil »

Cette séquence d'état illustre la possibilité d'exporter son profil dans un fichier en local.

d) Charger un profil

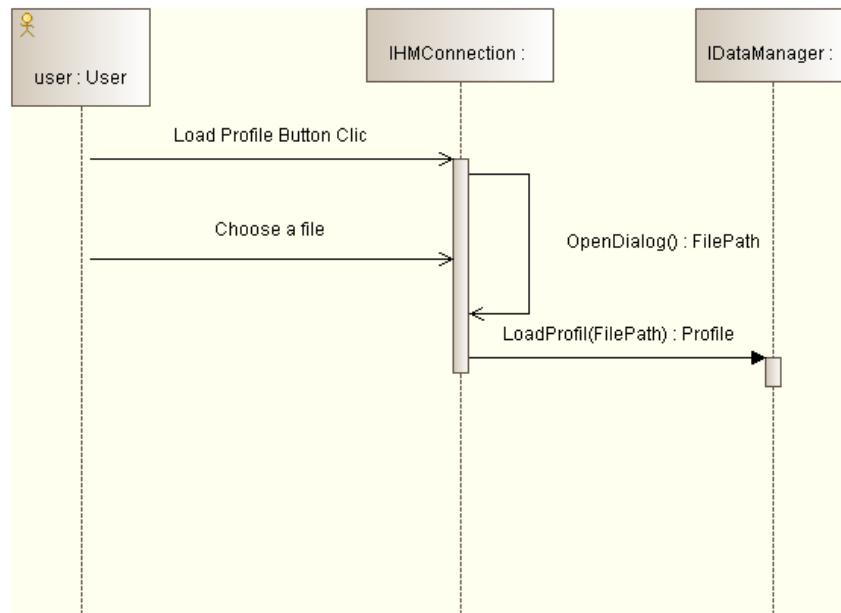


Figure 5 : Séquence d'état de la méthode « Charger profil »

Cette séquence illustre la possibilité de charger « importer » un profil localement. L'utilisateur choisit le fichier à importer.

e) Connexion

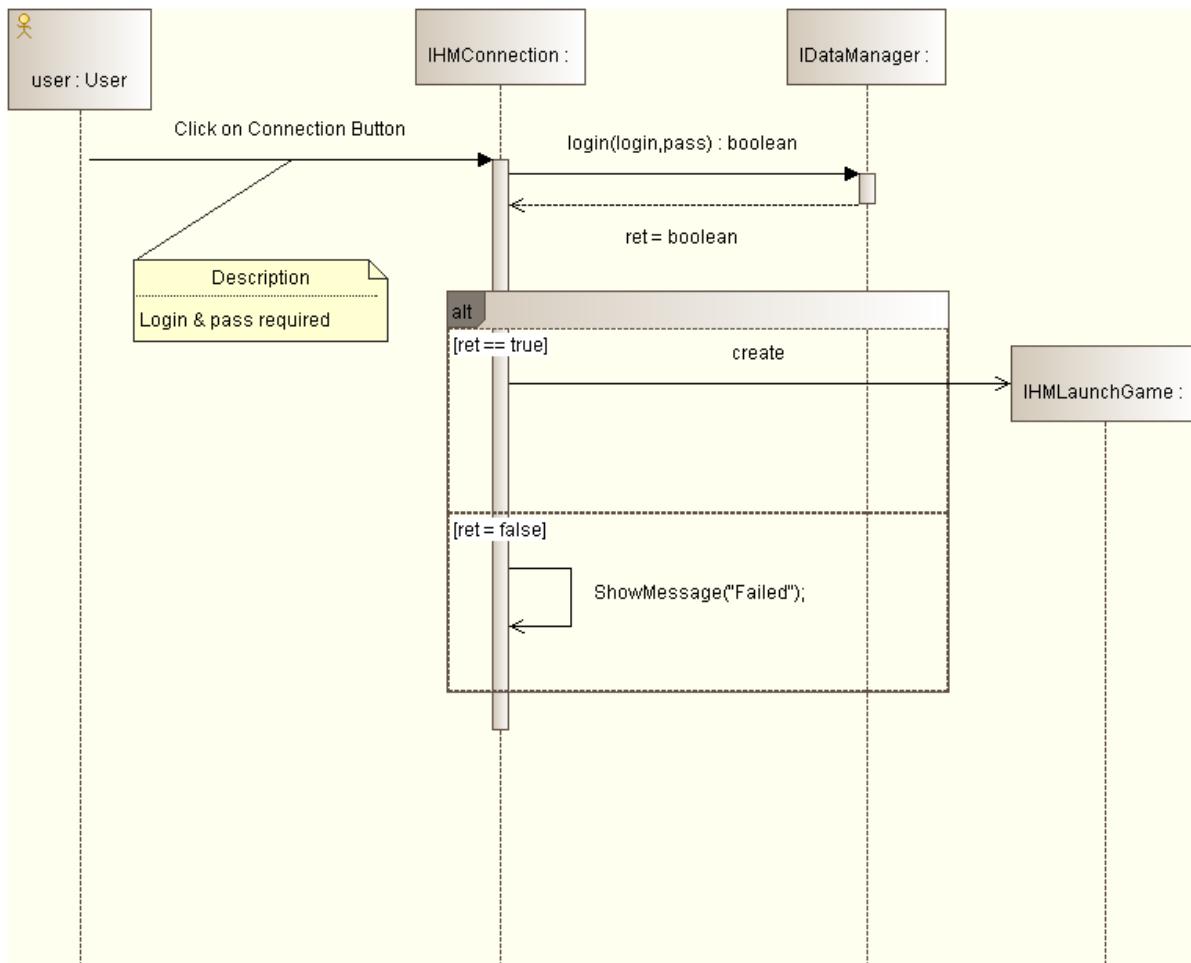


Figure 6 : Séquence d'état de la méthode « connexion »

Cette séquence d'état illustre le cas d'utilisation du module « connexion » où l'utilisateur saisit son pseudo et mot de passe via le formulaire de connexion.

f) Liste des joueurs connectés et mise à jour de la liste

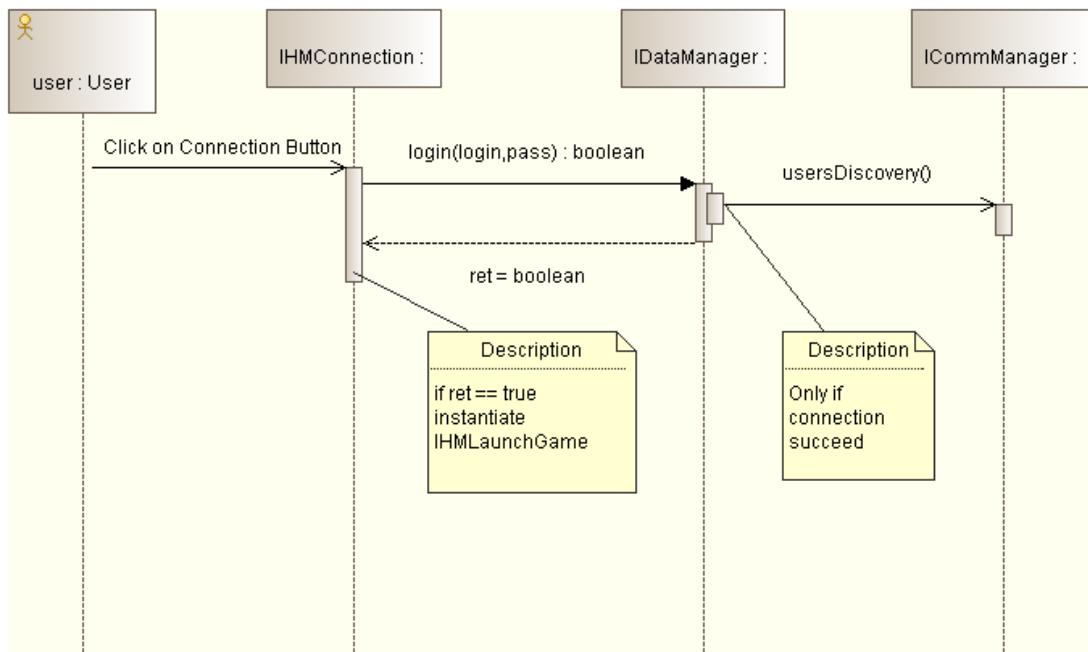


Figure 7 : Séquence d'état de la méthode « lister les joueurs connectés »

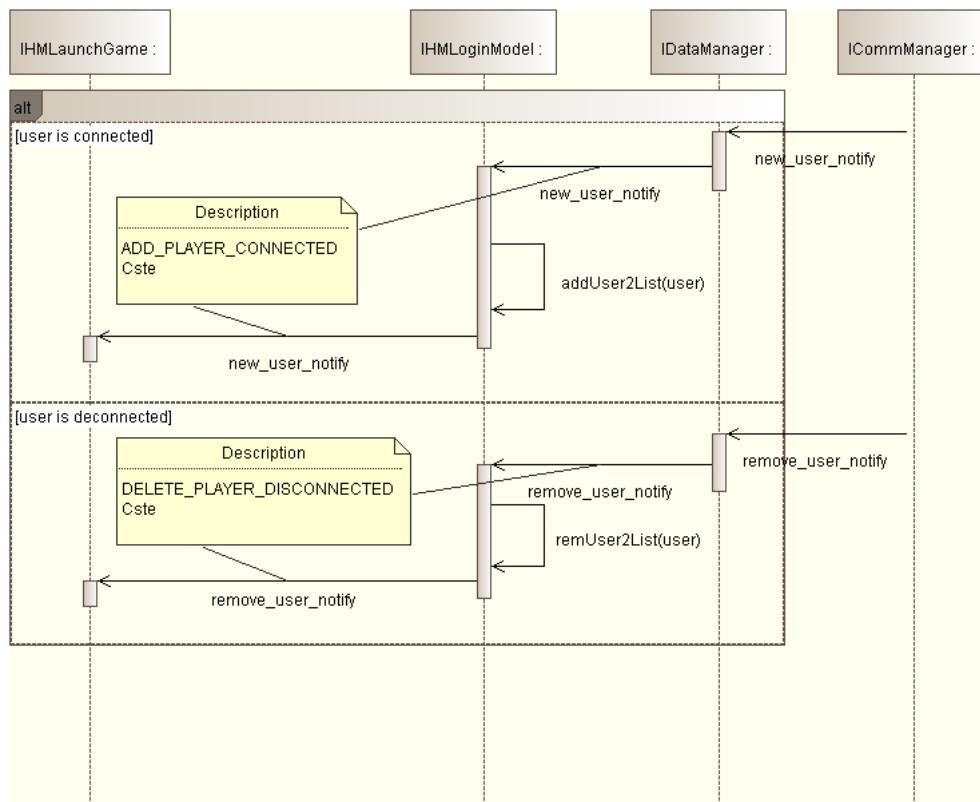


Figure 8 : Séquence d'état de la méthode « mise à jour de la liste des joueurs connectés »

g) Visualiser le profil de l'adversaire

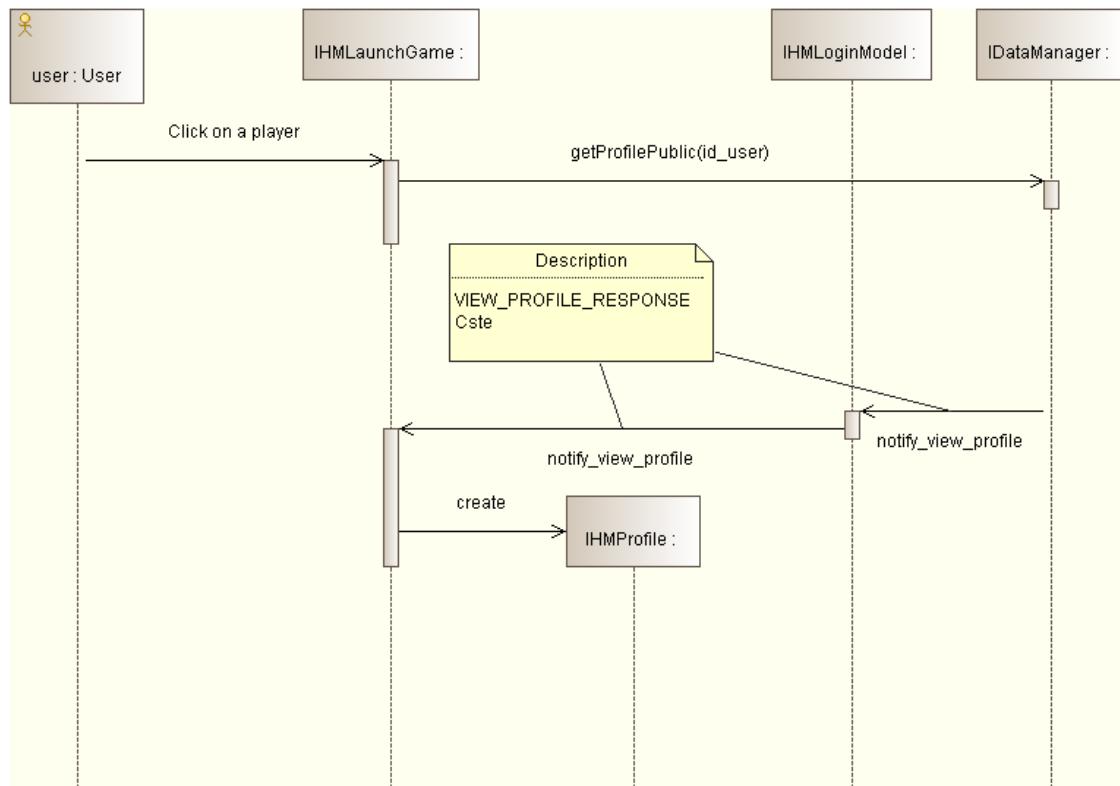


Figure 9 : Séquence d'état de la méthode « Visualiser un profil distant »

h) Commencer une partie

i. Envoyer une invitation

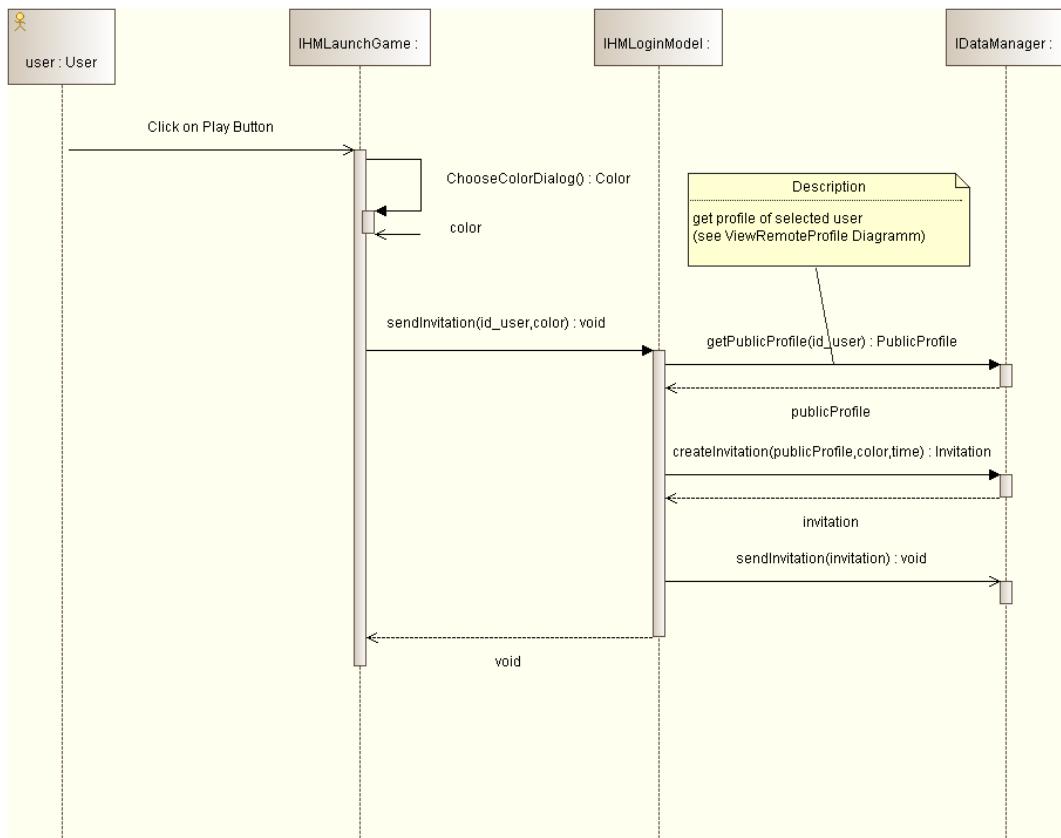


Figure 10 : Séquence d'état de la méthode «Envoyer une invitation»

ii. Recevoir une invitation

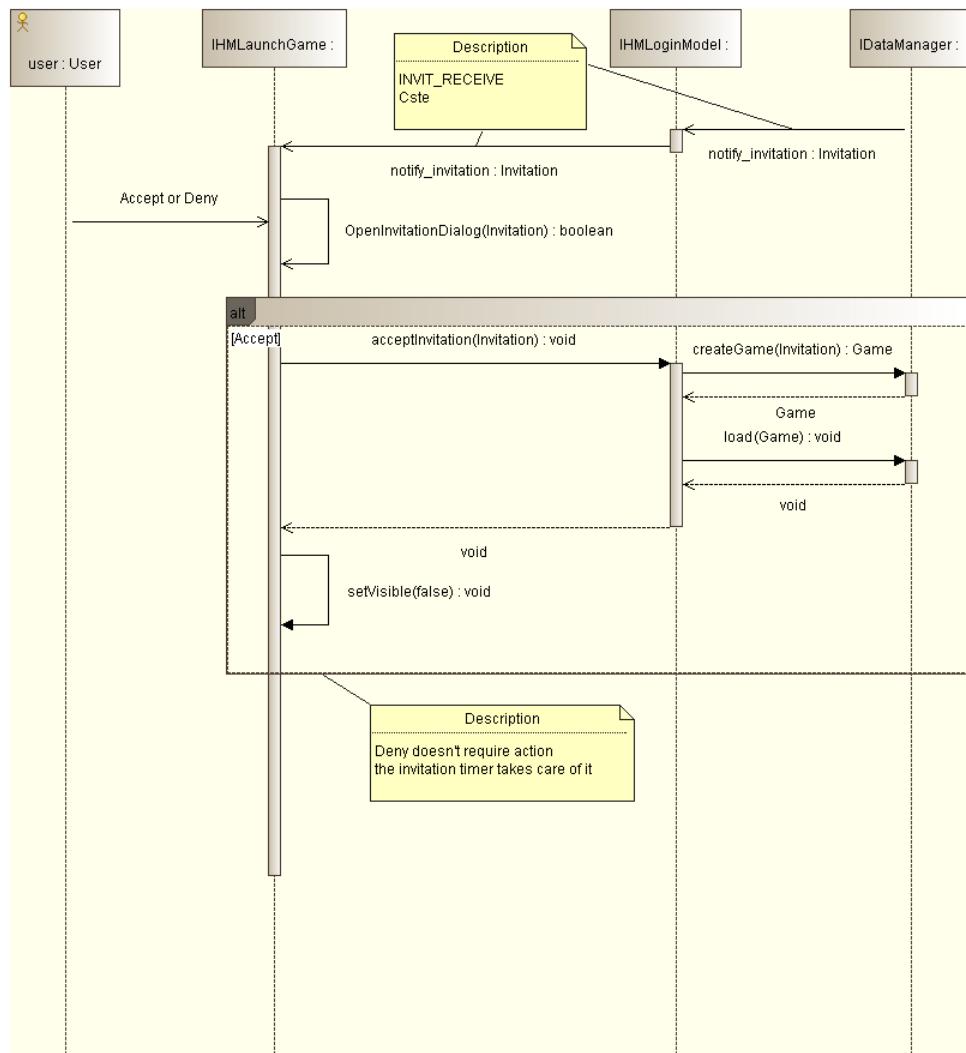


Figure 11 : Séquence d'état de la méthode «Recevoir une invitation»

iii. Lancer une partie

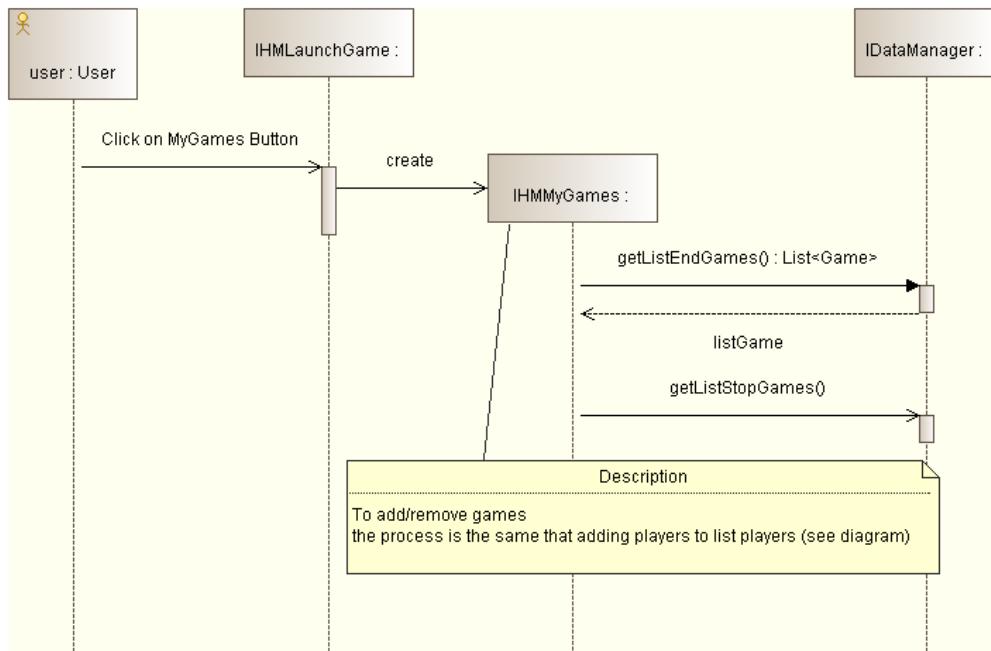


Figure 12 : Séquence d'état de la méthode «lancer une partie»

iv. Invitation acceptée

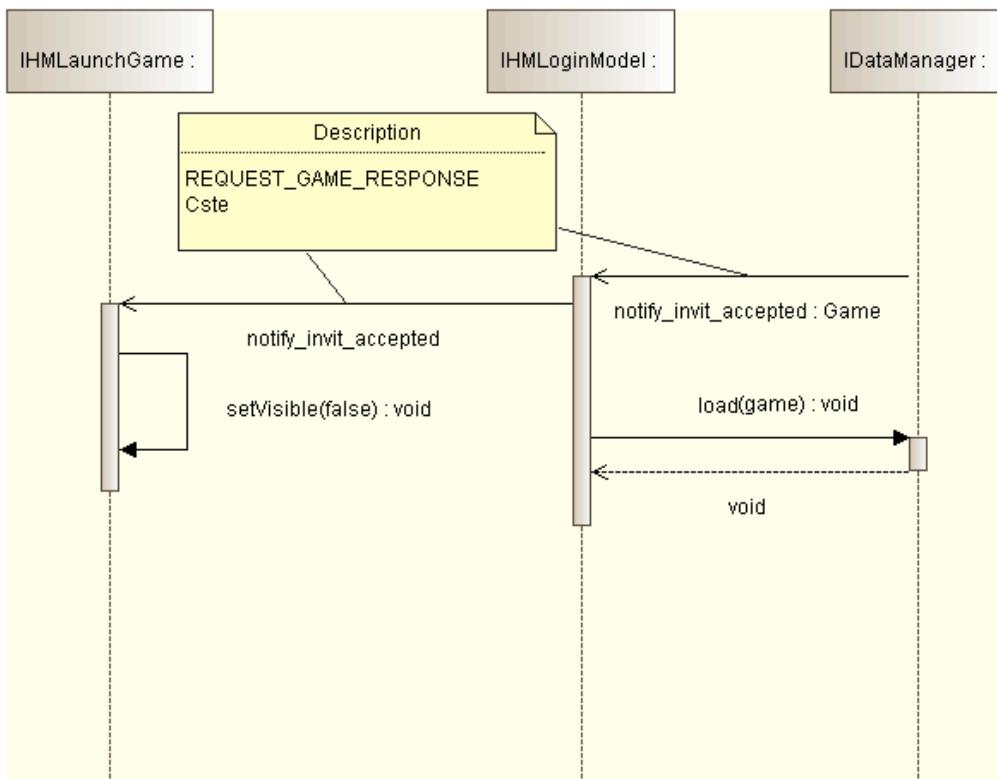


Figure 13 : Séquence d'état de la méthode «invitation acceptée»

v. Time out d'une invitation

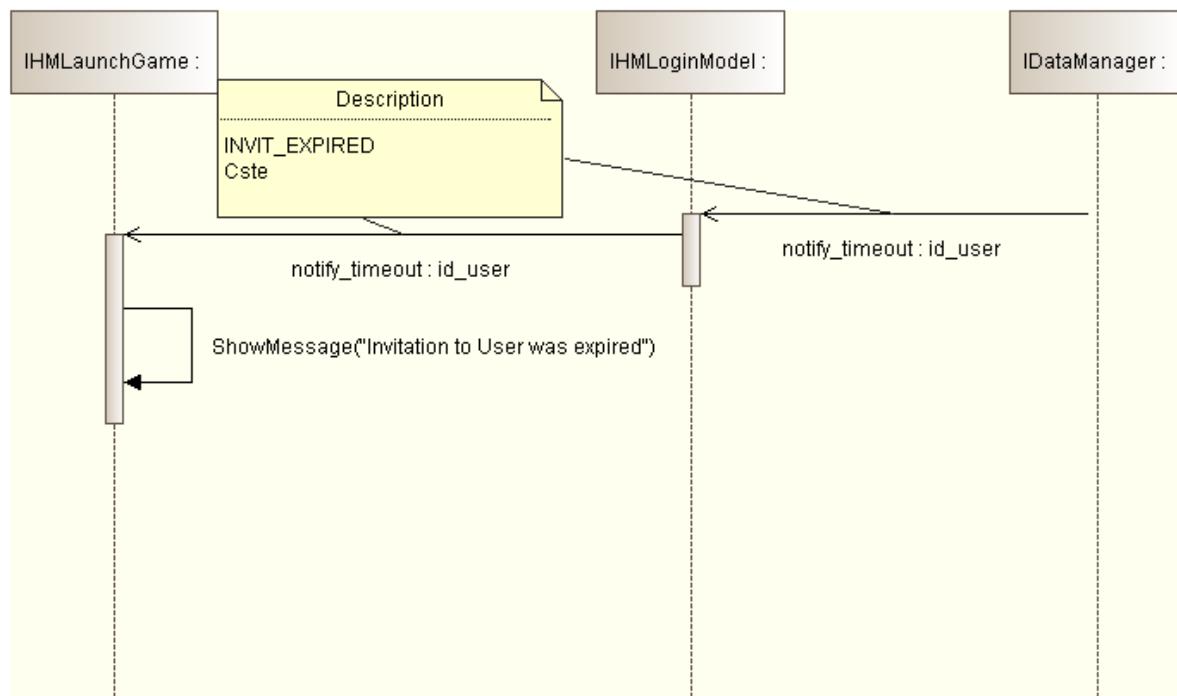
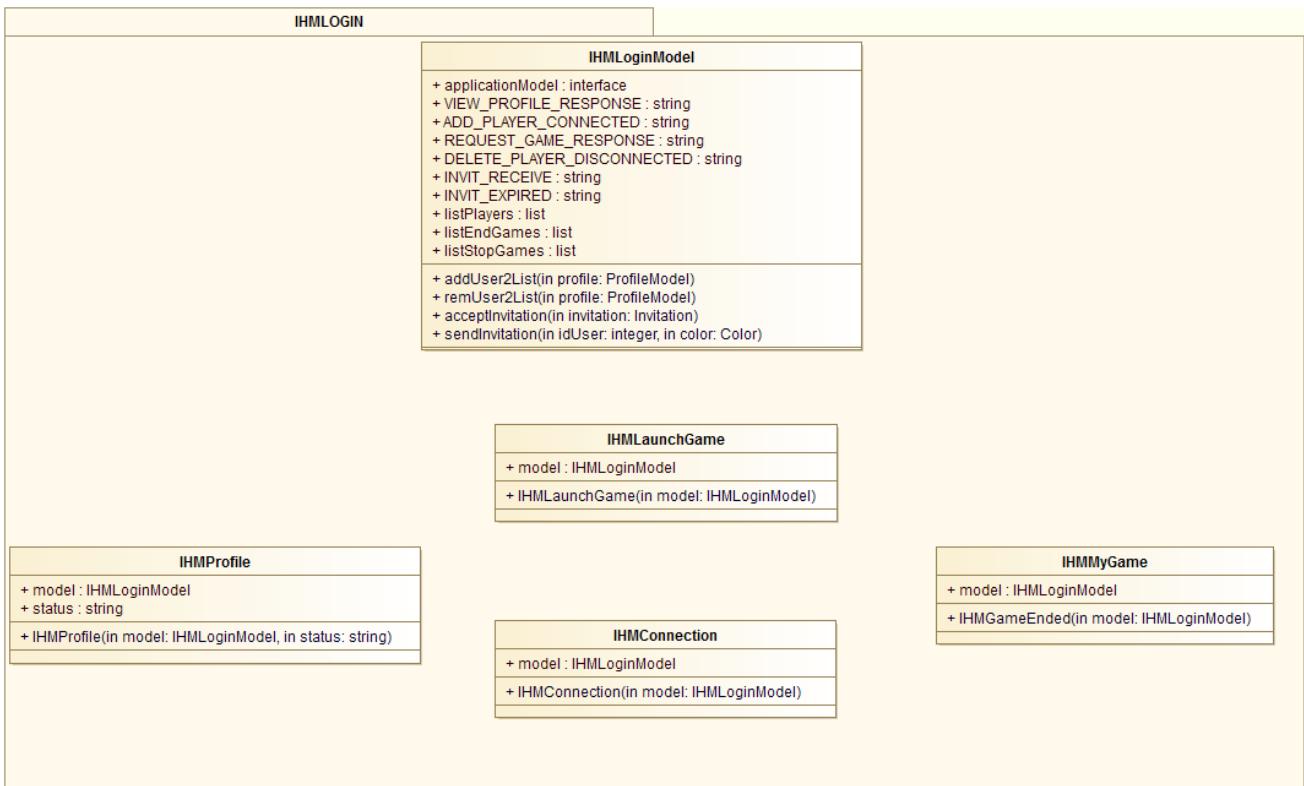


Figure 14 : Séquence d'état de la méthode «timeout d'une invitation»

2.3.4 Diagramme de classes et interfaces

Nous allons présenter l'architecture retenue pour le module « IHM Login », en accord avec les trois autres modules. Dans un premier temps, nous verrons l'architecture du modèle ainsi que les différentes interfaces avec le module Data. Ensuite nous présenterons le rôle de chacune des quatre frames retenues en justifiant nos choix de conceptions lorsque cela est pertinent.



Note : la propriété « status » de la classe **IHMProfile** permet de déterminer trois modes de création de l'IHM : édition, lecture seule ou création, respectivement la modification du profil, la consultation du profil distant et la création d'un profil.

a) IHMLoginModel

Il s'agit du modèle de l'IHM Login qui sera instancié une fois et utilisé dans chaque frame. Cette classe contient le modèle du profil de l'utilisateur courant (**currentProfile**) dont la classe est implémentée dans le module « Gestion de données ». Ce profil est sauvegardé en local sur la machine utilisateur mais peut être importé d'un autre ordinateur à l'aide d'une option d'importation et d'exportation de profile.

Cette classe dispose également de méthodes permettant de lancer l'ensemble des actions réalisables par l'utilisateur dans le cadre du module IHM Login. En voici une brève description :

- **addUser2List** qui permet d'ajouter un joueur à la liste des joueurs après un signal ajout d'un joueur par le DataManager.
- **remUser2List** qui permet de supprimer un joueur de la liste des joueurs après un signal suppression d'un joueur par le DataManager.
- **acceptInvitation** est la méthode appelée lorsque l'utilisateur accepte l'invitation d'un autre joueur à démarrer une partie.
- **sendInvitation** est la méthode appelée lorsque l'utilisateur envoie une demande de jeu à un autre utilisateur.

Le dataManager possède des méthodes que nous appellerons pour récupérer les données provenant des autres modules et qui nous sont utiles. Voici les méthodes dont nous aurons besoin :

- Dm.createProfile(infos) : Crée un nouveau profile avec les informations saisies par l'utilisateur. La méthode renvoie le profile créé.
- Dm.modifyProfile(infos) : Mets à jour le profile courant.
- Dm.saveProfile(path) : Permet de sauvegarder un profile en local.
- Dm.loadProfile(FilePath) : Charge un profile présent en local.
- Dm.login(login, password) : Permet de se connecter dans le jeu. À ce moment, la recherche des joueurs connectés est lancée. La liste des joueurs connectés sera mise à jour en continu à l'aide de nos variables ADD_PLAYER_CONNECTED, quand un nouveau joueur arrive sur le réseau, et DELETE_PLAYER_DISCONNECTED quand un joueur quitte le jeu.
- Dm.getProfilePublic(id_user) : Permet de voir un profile distant. Cette demande est asynchrone. La variable VIEW_PROFILE_RESPONSE permet de lancer l'affichage du profile lorsque l'utilisateur l'a reçu depuis le réseau.
- Dm.createInvitation(publicProfile,color,time) : Crée une invitation avec les paramètres du jeu souhaité. Cette méthode renvoie une invitation.
- Dm.sendInvitation(invitation) : Envoie une invitation à l'utilisateur distant. Le jeu n'est pas créé pour autant. La réponse de l'utilisateur distant se fera par la variable REQUEST_GAME_RESPONSE.
- Dm.createGame(invitation) : Lorsqu'un utilisateur accepte une invitation de partie, il crée le jeu directement à l'aide de cette méthode.
- Dm.load(game) : Lance une partie. Cette partie peut être une nouvelle partie, une partie interrompue qui est reprise ou une partie terminée. L'ensemble des informations sur la partie lancée étant contenues dans le GameModel.
- Dm.getListEndGame() : Permet de récupérer l'ensemble des parties terminées pour un joueur.
- Dm.getListStopGame() : Permet de récupérer l'ensemble des parties interrompues pour un joueur.

Toutes les différentes fenêtres ont accès à notre modèle car leur constructeur le prend en paramètre.

b) IHMConnection

Il s'agit de la frame d'accueil qui gère la connexion. Comme expliqué ci-dessus, chaque frame possède un constructeur prenant en paramètre l'instance du model IHMLogin. Dans la maquette, cela correspond à la fenêtre « Connexion ».

c) IHMProfile

Cette frame correspond à la visualisation du profile. Le choix technique retenu est d'utiliser la même frame pour la visualisation, la création et la mise à jour d'un profil. Pour cela, le constructeur prend un paramètre un code statu qui indique le type de visualisation. S'il s'agit d'une consultation d'un profil distant, l'ensemble des champs ne seront pas éditables et l'utilisateur n'aura donc qu'un accès en lecture du profile. S'il s'agit d'une création d'un profil, un nouveau modèle de profile sera créé et à la fin la méthode dm.createProfile(infos) sera appelée. Enfin, si il s'agit d'une modification du profil utilisateur, le profile actuel sera chargé dans les champs qui seront éditables. À la validation, la méthode dm.modifyProfile(infos) sera appelée.

Avec ce choix de conception, nous optimisons donc le code produit et assurons une réutilisabilité du code. Dans la maquette, cela correspond aux fenêtres « Inscription », « Fiche Profile » et « Gestion du profil ».

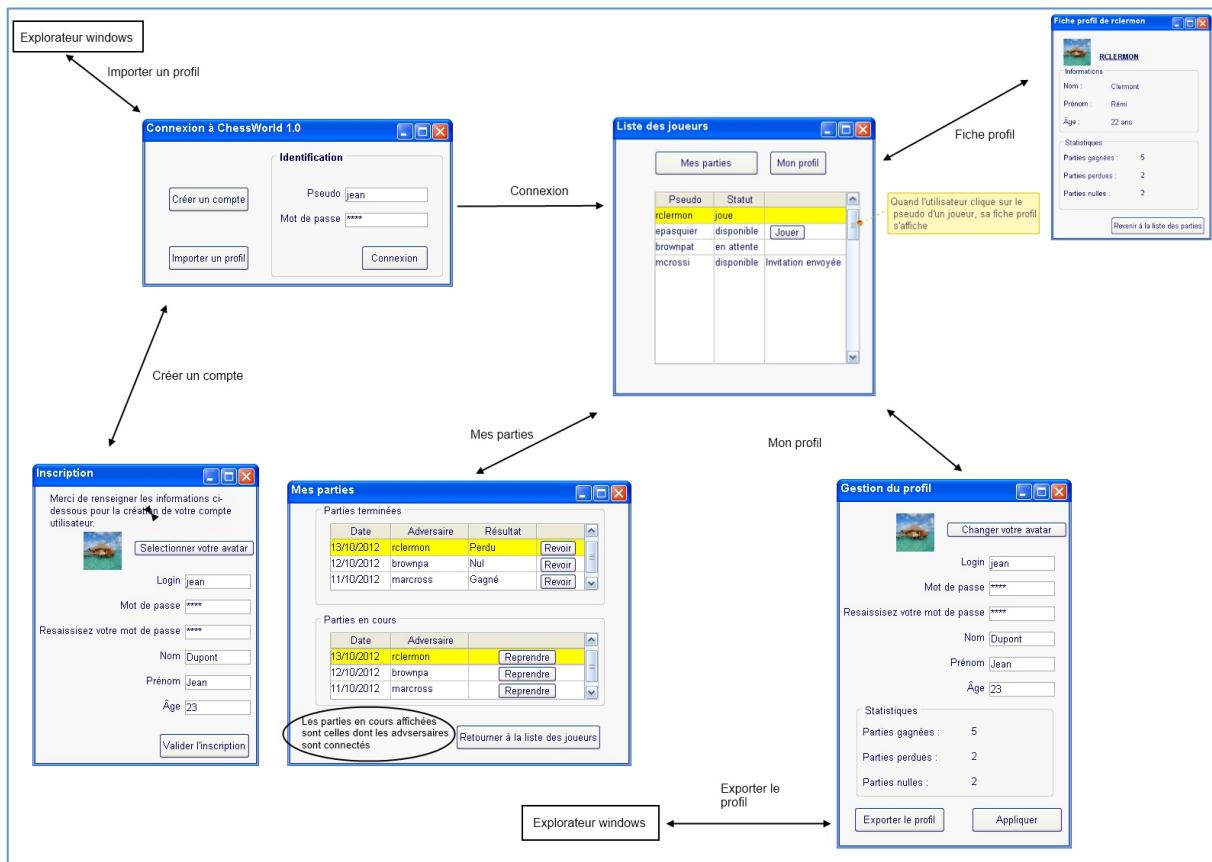
d) IHMLaunchGame

Cette vue gère l'ensemble du processus de lancement d'une partie. Dans la maquette, cela correspond aux fenêtres « Liste des joueurs » ainsi qu'aux cinq fenêtres associées au lancement d'une partie.

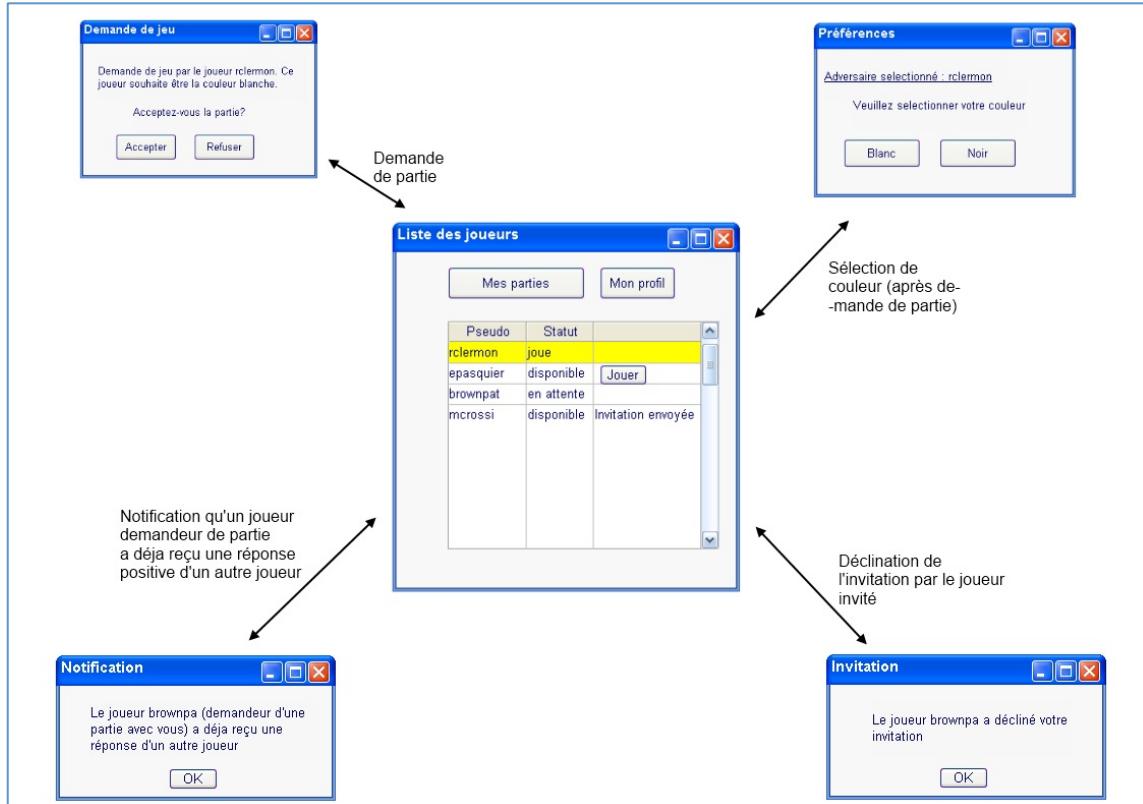
e) IHMMyGame

Dans cette frame, nous aurons la liste des différentes parties terminées et celles interrompues. L'utilisateur passera donc sur cette vue dès qu'il veut revoir une de ses parties ou consulter son historique de jeux. Dans la maquette, cela correspond à la fenêtre « Mes Parties ».

2.3.5 Maquettage



Maquette 1



Maquette 2

a) Maquette 1

Elle montre à quoi ressembleront les différentes fenêtres générales de l'application d'après le use case. Celle-ci se compose donc de :

La fenêtre de connexion : elle permet à l'utilisateur de se connecter, de créer un compte ou d'importer un fichier profil (via l'explorateur de fichiers).

La fenêtre de création de compte utilisateur : elle comporte les informations essentielles à toute création d'un compte utilisateur (login, mot de passe) mais aussi quelques informations supplémentaires personnelles sur le joueur (nom, prénom, âge) ainsi que la possibilité de choisir un avatar associé à son profil.

La fenêtre de liste des joueurs : elle permet à l'utilisateur de voir tous les joueurs connectés sur le réseau en temps réel. C'est la fenêtre principale qui permet d'accéder à son profil et à ses parties (sauvegardées ou terminées). Lors d'un clique sur le pseudo d'un joueur, sa fiche profile s'ouvrira dans une nouvelle fenêtre. Elle permet ensuite d'effectuer la demande de partie.

La fenêtre « Mes parties » : elle permet de visualiser les différentes parties que l'utilisateur peut reprendre ou les parties terminées si celui-ci souhaite analyser une ancienne partie. Les parties à reprendre affichées sont uniquement celles dont les adversaires sont connectés.

La fenêtre de profil d'un utilisateur connecté : elle affiche les différentes informations du joueur (pseudo, nom, prénom, âge). Elle inclue aussi un système de statistiques du joueur (parties gagnées, perdues, nulles).

La fenêtre de gestion de son profil : l'utilisateur peut modifier son profil via cette fenêtre. Il peut aussi choisir d'exporter son profil pour utiliser l'application sur un autre ordinateur.

b) Maquette 2

Elle montre les différentes fenêtres qui pourront être affichées lors d'une demande de partie. Le scénario est le suivant :

Soient des joueurs connectés (Joueur 1 et des autres joueurs), le Joueur 1 souhaite effectuer une partie avec une personne connectée. Il clique donc sur le bouton « Jouer » correspondant à chacun des joueurs avec lesquels il souhaite jouer. Pour chaque proposition, la fenêtre de préférences s'affiche alors sur l'écran du joueur 1 pour lui demander sa couleur désirée (Blanc ou Noir). Les joueurs invités reçoivent alors sur leur écran la demande de partie (via la fenêtre de demande de jeu) avec les informations correspondantes au joueur adverse ainsi que la couleur que celui-ci a choisi. Le joueur 1 attend donc que l'un des joueurs invités accepte ou décline son invitation. Si un des joueurs accepte la partie, le jeu se lance et les autres joueurs invités reçoivent alors une notification que le Joueur 1 a déjà trouvé une partie.

2.4 IHM grille

Le rôle de l'IHM Grille est de gérer les interactions avec l'utilisateur. Il représente l'interface avec laquelle celui-ci utilisera le jeu, et donc interagira avec toutes les fonctionnalités (jouer une partie, jouer un coup, discuter avec l'adversaire, revoir une partie). Il est donc nécessaire qu'elle soit ergonomique, efficace, facile à utiliser et agréable visuellement.

L'objectif de notre module est donc de créer un environnement graphique plaisant pour l'utilisateur lui permettant de jouer à ce jeu d'échec dans les meilleures conditions possibles.

2.4.1 Documents d'étude

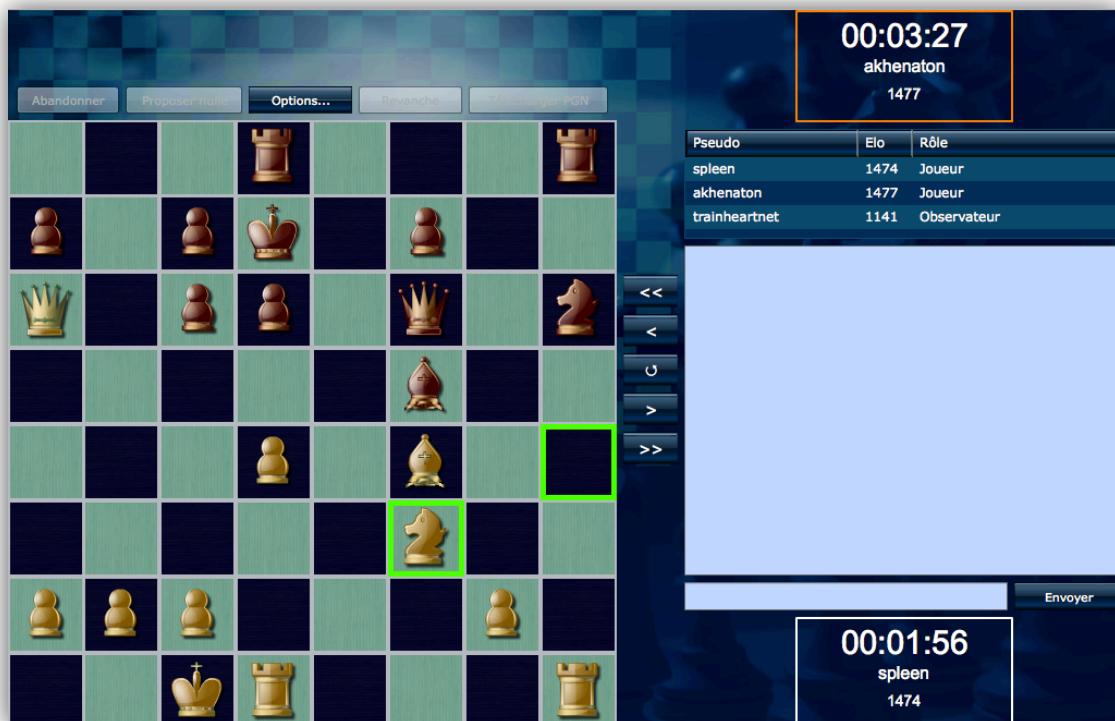


Figure 11 : Interface du site www.echechs-online.fr

Une étude a été menée en début de conception afin d'avoir une idée des interfaces déjà existantes et de ne pas risquer d'oublier certains points importants lors de la conception. Voici un exemple de site sur lequel c'est basé notre étude.

L'étude des différentes interfaces disponibles a permis de mettre en avant certaines fonctionnalités à implémenter pour le jeu. Comme on le voit sur l'image ci-dessus, il faut permettre à l'utilisateur de voir les emplacements où il peut placer la pièce sélectionnée, consulter le temps sur son horloge et celle de l'adversaire ou encore échanger des messages avec lui et consulter l'historique.

2.4.2 Choix de conception

Le dossier de conception IHM grille est le résultat de l'étude de conception de l'interface nous permettant de définir l'ensemble des moyens techniques et humains capables de satisfaire les besoins de l'utilisateur et de répondre aux contraintes du projet.

Certaines parties comme le diagramme de classes ou encore les interfaces entre l'IHM grille et le data Manager permettent d'assurer la cohérence des classes, des données et des fonctions entre les différents modules. Le futur programme sera ainsi clairement structuré afin d'éviter tous types de problèmes.

Le choix de conception IHM Grille repose principalement sur le diagramme de classe qui contient l'ensemble des informations nécessaires aux éléments utilisés dans le module. Les diagrammes de séquences détaillent chacun les déroulements d'une utilisation spécifique de l'interface.

Pour l'échange entre les interfaces, il a été choisi de générer différents événements (Event) géré par le Data Manager qui permettra de notifier l'interface quand un changement sera opéré, permettant ainsi de communiquer entre l'IHM locale et l'IHM distante.

2.4.3 Cas d'utilisation

L'interface concerne le joueur local qui est en lien indirect avec le joueur distant. Différentes actions s'offrent à ces 2 joueurs (voir diagramme ci-dessous) pour qu'ils puissent communiquer ensemble.

Interactions mobilisant les 2 joueurs :

- le plateau de jeu, avec la mise en place d'une partie, la possibilité de jouer un coup et de déplacer une pièce ;
- Le chat, envoi et réception de messages ;
- Terminer une partie : quitter la partie, annoncer une partie nulle.

Interactions propres à un joueur :

- Visualisation des coups possibles d'une pièce
- Visualisation de l'historique
- Revoir une partie
- Revoir anciens messages du chat
- Sauvegarder une partie
- Visualiser le profil du joueur adverse

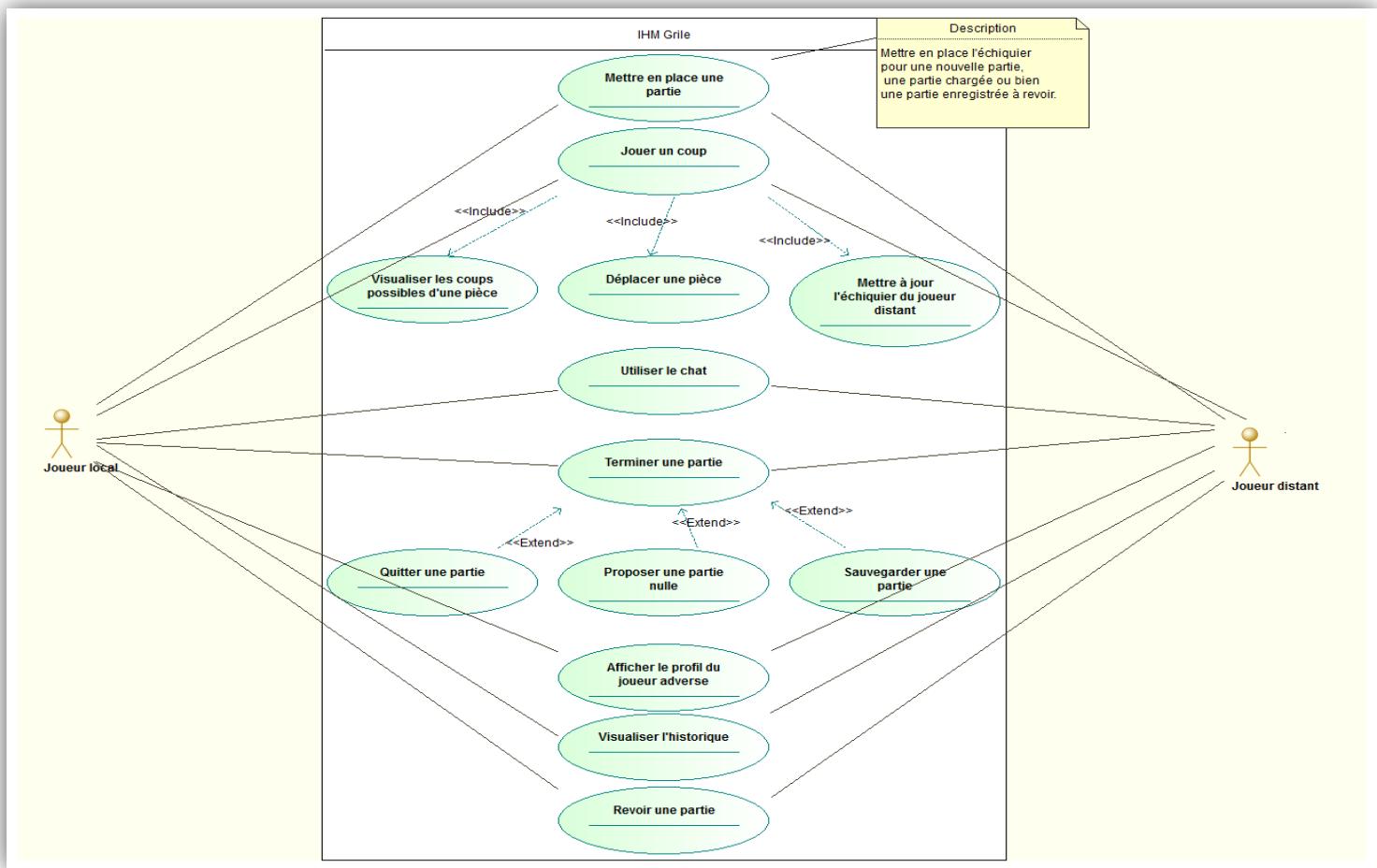


Diagramme d'utilisation IHM Grille

2.4.4 Diagramme de séquence

On trouve ici trois diagrammes de séquences importants : la mise en place d'une partie, la gestion d'un coup joué et l'envoi d'un message dans le chat.

a) Mise en place d'une partie

Le diagramme suivant correspond à la mise en place d'une nouvelle partie : à la création de la partie (Game), l'IHM est lancée et construit les échiquiers.

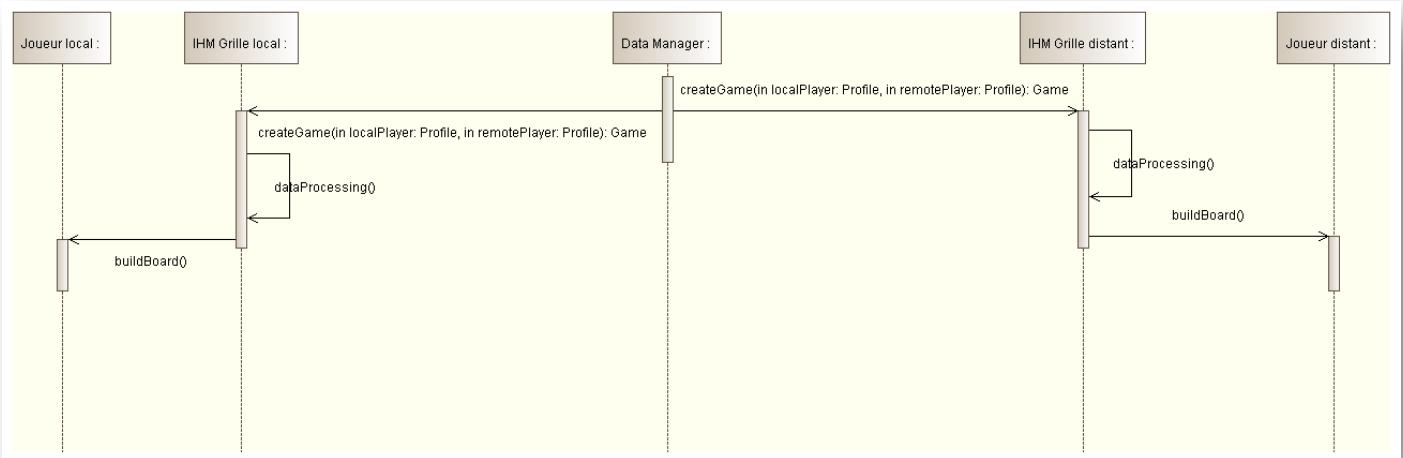


Diagramme séquentiel « mise en place d'une partie »

b) Jouer un coup

Le diagramme suivant correspond au déplacement d'une pièce et à la réalisation complète d'un coup. Le coup se décompose en une sélection de pièce, un déplacement (choix d'un coup valide), un envoi et enfin une mise à jour des deux échiquiers.

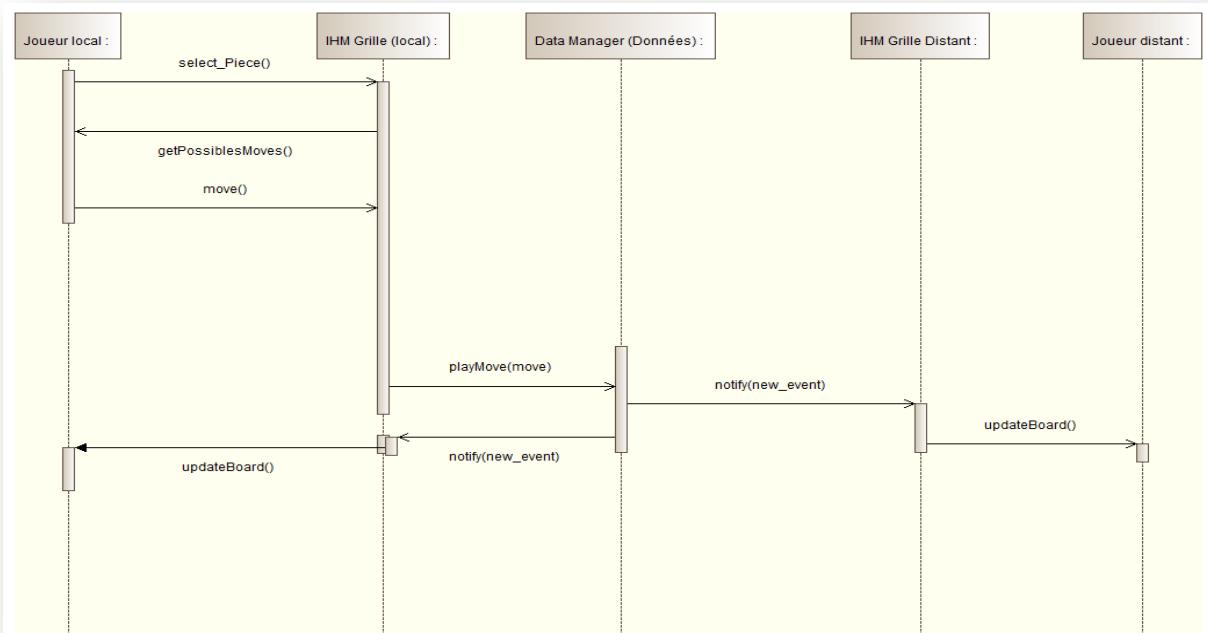


Diagramme séquentiel « jouer un coup »

c) Envoi d'un message

Le diagramme suivant correspond à l'utilisation du chat, le joueur local rédige et envoie un message au joueur distant.

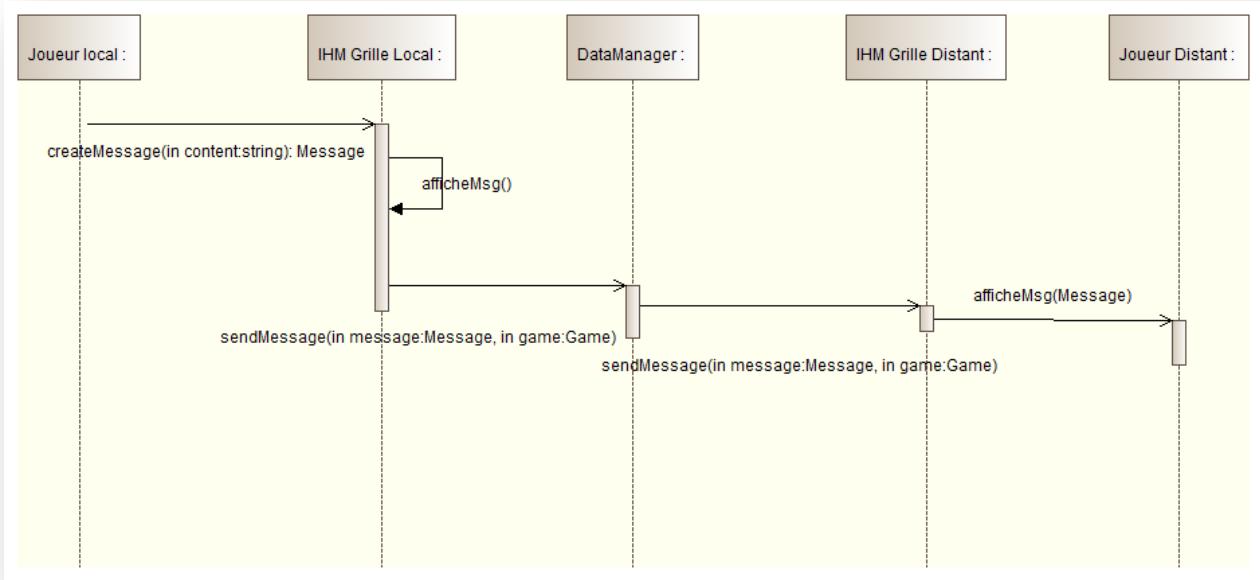


Diagramme d'envoi d'un message sur le chat

2.4.5 Diagramme de classes et interfaces

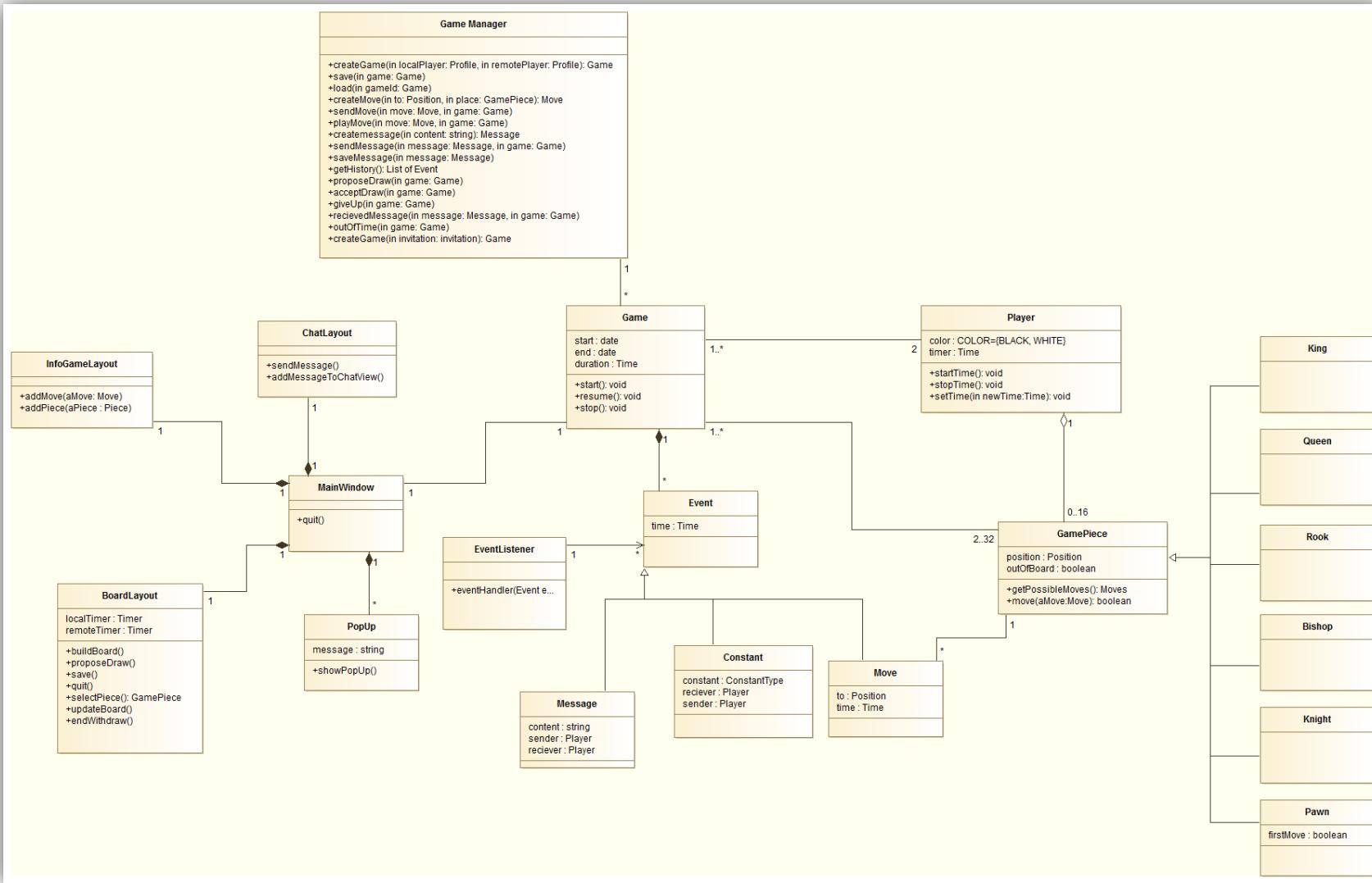


Diagramme de classe IHM Grille

Liste des interfaces du GameManager utilisées par IHM Grille :

a) Jouer un coup:

createMove(in to: Position, in piece: GamePiece): Move

Permet de créer un déplacement d'une pièce en fonction de sa position.

sendMove(in move:Move, in game:Game)

Permet d'envoyer un déplacement après action d'un joueur local vers un joueur distant.

b) Gestion partie :

giveUp(in game: Game)

Permet de quitter une partie en abandonnant.

proposeDraw(in game:Game)

Permet de proposer un match nul à l'adversaire.

acceptDraw(in game:Game)

Permet d'accepter une proposition de match nul d'un adversaire.

save(in game:Game)

Permet de sauvegarder une partie.

getHistory():List of Event

Permet de récupérer la liste d'évènements de la partie en cours.

outOfTime(in game:Game)

Permet de mettre fin à la partie en cas de temps épuisé par un joueur.

c) Gestion chat:

createMessage(in content:string): Message

Permet de créer un message à partir d'une chaîne de caractère.

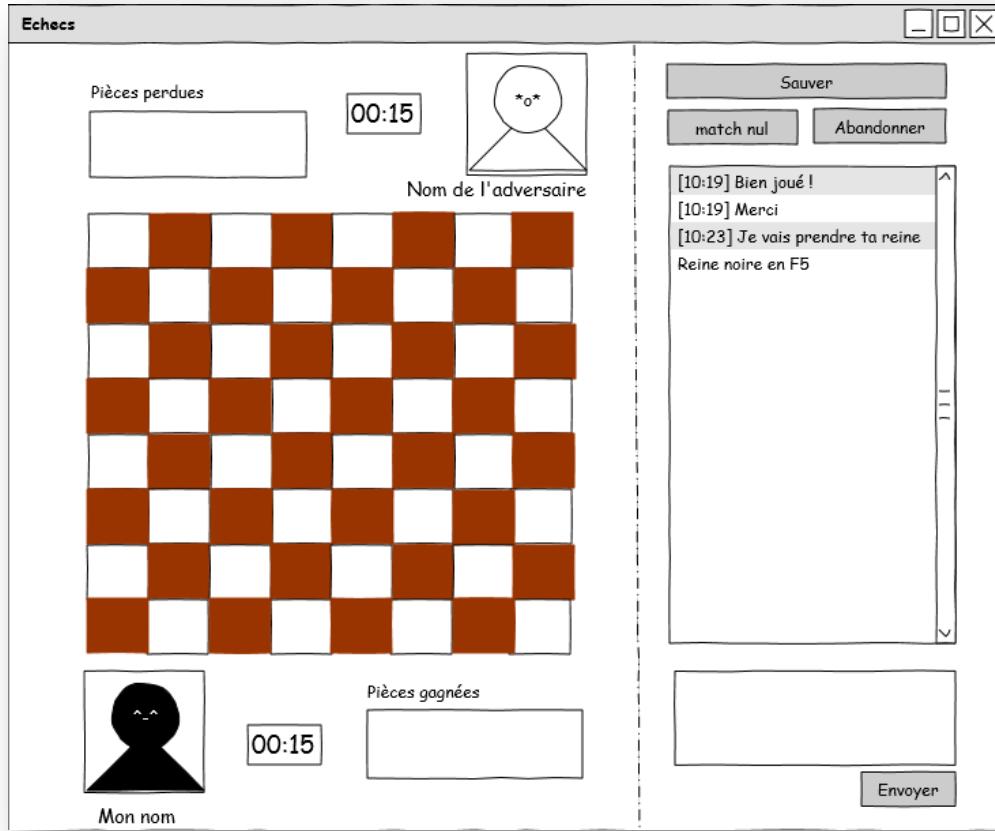
sendMessage(in message:Message, in game:Game)

Permet d'envoyer un message d'un joueur local à un joueur distant.

2.4.6 Maquettage

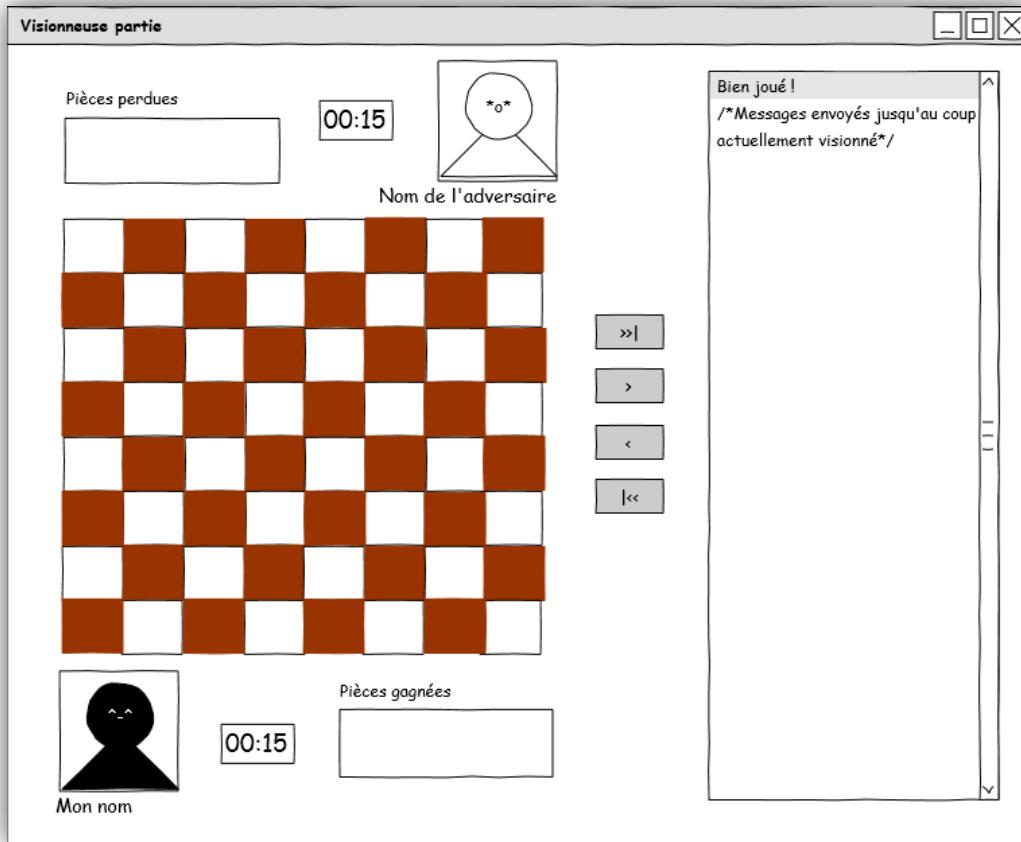
a) Apparence générale de l'interface

Des maquettes ont été réalisées pour essayer de répondre au mieux aux attentes et intégrer le chat et l'historique à la fenêtre contenant l'échiquier et le profil des joueurs s'affrontant.



Maquette de conception de l'interface coté utilisateur

L'interface est modulable, ainsi le joueur peut, selon ses préférences, placer à droite ou à gauche de l'échiquier la fenêtre de chat et d'historique. Dans le cas du visionnage d'une partie, des boutons permettant de naviguer entre les différents événements sont ajoutés à la fenêtre de jeu comme on peut le voir ci-dessous.

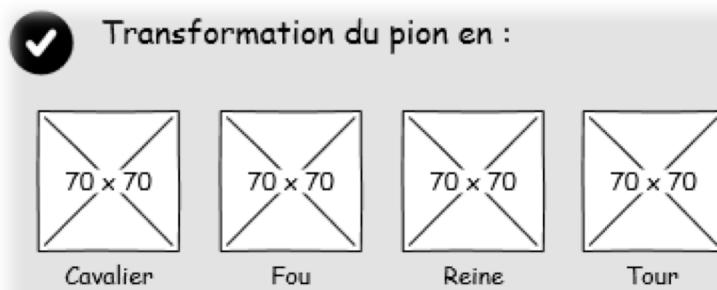


Maquette de l'interface de visionnage de parties terminées

b) Fenêtres pop-up interactives

Certaines fonctionnalités nécessitent l'utilisation de fenêtre pop-up notamment pour les actions de changement de pièce (transformation du pion) ou pour les demandes d'abandon ou de match nul. Certaines de ces pop-up apparaissent du côté du joueur ayant fait la demande tandis que d'autres apparaissent sur l'interface de l'adversaire (demande d'accord pour un match nul).

Voici un exemple d'une fenêtre pop-up interactive, lorsque le pion arrive dans la dernière ligne du damier du camp adverse :



Fenêtre pop-up, changement de pion