Frogger

Tony Clavien, Maxime Guillod, Gabriel Luthier & Guillaume Milani

13 juin 2017

Table des matières

1 Introduction **5**

2 Fonctionnement général du jeu **5**

2.1 Objectifs de base 5

2.2 Utilisation de l'applicatif 6

2.3 Règles du jeu 6

2.4 Contraintes 6

2.5 Priorités de développement 6

2.6 Base de données 6

3 Analyse **7**

3.1 Partage des responsabilités entre le serveur et le client 7

3.1.1 Responsabilités du client 7

3.1.2 Responsabilités du serveur 7

3.2 Diagramme d'activité général 7

3.3 Rôle des participants 7

3.4 Cas d'utilisation 8

3.4.1 Diagramme général de contexte 8

3.4.2 Description des acteurs 8

3.4.3 Scénario principal 8

3.4.4 Extensions (ou scénarios alternatifs) 8

3.4.5 Scénario d'administration 8

3.5 Modèle de domaine 9

3.5.1 Modèle de domaine pour le client 9

3.5.2 Modèle de domaine pour le serveur 9

3.6 Base de données 9

3.6.1 Modèle conceptuel (entité-associations) 9

4 Conception du projet **9**

4.1 Protocole d'échange entre le client et le serveur 9

4.1.1 Communication 9

4.1.2 Données échangées 9
4.2 Diagrammes de classes du serveur et du client 10
4.3 Modèle conceptuel & relationnel de la base de données 10

5 Implémentation du projet 10

5.1 Structure du projet 10
5.1.1 GEN-protocol 10
5.1.2 GEN-BDD 10
5.1.3 GEN-server 10
5.1.4 GEN-admin 10
5.1.5 GEN-Frogger 10
5.2 Technologies, langages, bibliothèques utilisées 10
5.3 Technologies "originales" (autres que les technologies étudiées à l'école) 11
5.3.1 Guava 11
5.4 Problèmes éventuels rencontrés et solutions apportées 11
5.4.1 Logique FX 11
5.4.2 ReadLine et $\backslash n$ 11
5.4.3 Partage de classes/code entre plusieurs projets 11

6 Gestion du projet 12

6.1 Rôle des participants au sein du groupe de développement 12
6.2 Plan d'itérations initial 12
6.2.1 Création de l'interface de base 12
6.2.2 Ajout du 1er joueur avec les obstacles 12
6.2.3 Ajout du 2ème joueur en local 12
6.2.4 Mise en place des logiques de jeu 13
6.2.5 Ajout de la communication avec le serveur 13
6.2.6 Application administrateur 13
6.2.7 Ajout du Lobby 13
6.2.8 Finalisation 14
6.3 Suivi du projet 14
6.3.1 Itération 1 : Création de l'interface de base 14

6.3.2 Itération 2 : Ajout du 1er joueur avec les obstacles . . . 14
6.3.3 Itération 3 : Ajout du 2ème joueur en local . . . 14
6.3.4 Itération 4 : Mise en place des logiques de jeu . . . 14
6.3.5 Itération 5 : Ajout de la communication avec le serveur . . . 15
6.3.6 Itération 6 : Application administrateur . . . 15
6.3.7 Itération 7 : Ajout du Lobby . . . 15
6.3.8 Itération 8 : Finalisation . . . 15
6.4 Stratégie de tests . . . 16
6.4.1 Utilisation de JUnit pour une classe définissant le protocole . . . 16
6.4.2 Résultats des tests . . . 17
6.5 Stratégie d'intégration du code de chaque participant . . . 17

7 État des lieux 18
7.1 Ce qui fonctionne (résultats des tests) . . . 18
7.2 Ce qu'il resterait à développer (en proposant une planification) . . . 18

8 Auto-critique 18

9 Conclusion 18

A Manuel d'utilisation 19

Table des figures

1 Maquette d'une partie . . . 5
2 Diagramme d'activité . . . 20
3 Diagramme d'utilisation . . . 21
4 Modèle de domaine . . . 22
5 Modèle conceptuel . . . 23
6 Résultats des tests JUnit . . . 23

1 Introduction

Dans le cadre du cours GEN (Génie logiciel), nous allons créer une application client-serveur par groupe de 4 personnes. Nous avons choisi de développer un jeu se basant sur *Frogger*, qui consiste à faire traverser des routes pleines de trafic à une grenouille sans se faire écraser.

Les spécifications de bases de ce projet sont de faire une application client-serveur, qu'elle comporte une base de données (dont le format est libre), qu'elle comporte deux acteurs, que le développement soit géré avec git et que des tests soient effectués avec JUnit.

Nous avons choisi de développer ce jeu avec Java, car c'est un langage que l'on maitrise tous bien, qui nous plait et qui est adapté à ce type de contraintes. On peut en effet facilement mettre en place une communication client-serveur (en tout cas pour une application simple). La gestion de requêtes SQL et aussi aisée.

On présente dans ce rapport toutes les phases de la méthode de développement UP : initialisation, élaboration, construction, puis transition. À part la dernière qui n'a pas vraiment été atteinte, on va expliciter dans ce rapport quel est le travail qui a été fourni pour gérer ce projet avec cette méthode de développement.

2 Fonctionnement général du jeu

2.1 Objectifs de base

Le but du jeu sera de faire descendre une piste de ski à un Valaisan (*joueur skieur*). Il devra éviter les obstacles la traversant qui seront envoyé par le *défenseur*.

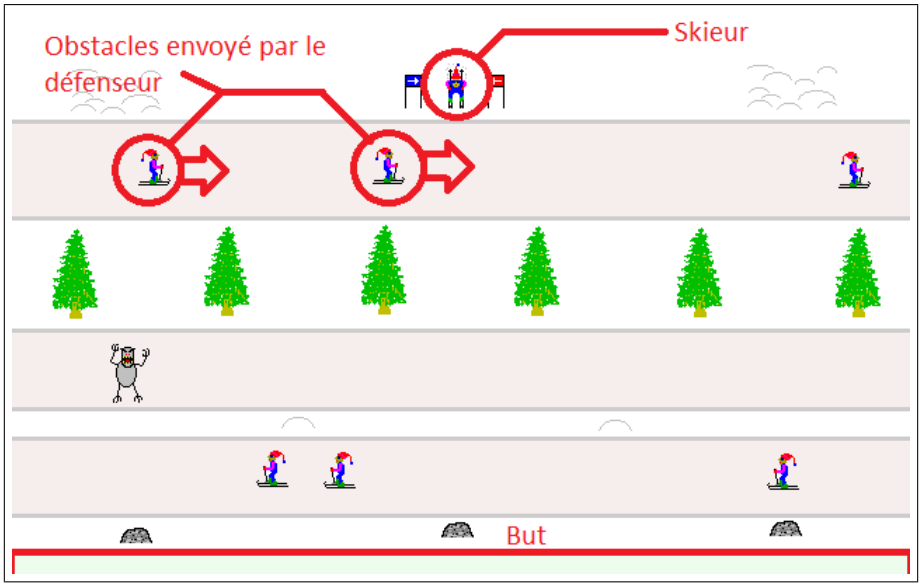

FIGURE 1 – Maquette d'une partie

Les zones rouges sont les zones à disposition du joueur 2 pour placer les obstacles, les sapins sont des obstacles fixes. Le joueur 1 gagne s'il atteint la zone en vert sans heurter d'obstacle.

2.2 Utilisation de l'applicatif

Le joueur 1 contrôle son personnage à l'aide des flèches du clavier (← : à gauche, ↓ : en bas, → : à droite) et tente d'éviter les obstacles. Le joueur 2 place les obstacles dans les colonnes, ceci fait que l'obstacle se met en mouvement dans la colonne. Une bouteille de Fendant^{TM}se vide au fur et à mesure que le joueur 2 place des obstacles. Il doit ensuite attendre qu'elle se remplisse à nouveau pour placer de nouveaux obstacles.

L'administrateur peut régler les paramètres du jeu (« skin » du jeu, vitesse des obstacles, vitesse de rechargement de la bouteille de Fendant^{TM}). L'administrateur est un rôle hérité de celui de joueur (un administrateur est donc avant tout un joueur). Ce droit est stocké dans la base de données.

2.3 Règles du jeu

Pour le joueur 1 : le but est d'arriver en bas de la piste sans avoir heurté d'obstacle.
Pour le joueur 2 : le but est que le joueur 1 heurte un obstacle.

2.4 Contraintes

Le joueur 1 ne peut pas revenir en arrière, une fois descendu une portion de piste il peut s'arrêter, changer de «colonne» ou de continuer à descendre.

Le joueur 2 peut envoyer des obstacles pour autant que la bouteille de Fendant^{TM}ne soit pas vide.

2.5 Priorités de développement

1. Première version du jeu «standalone» avec un seul joueur qui prend le rôle du joueur 1 (descendre la piste). Les obstacles sont envoyés aléatoirement. (En parallèle, développement du serveur et de l'API).

2. Ajout du deuxième joueur en local (les deux jouent sur la même machine avec des touches différentes du clavier).

3. Les joueurs jouent chacun sur leur machine et se connectent à un serveur distant centralisant la partie. Lorsqu'un joueur se connecte, il voit la liste des parties créées (soit en attente d'un joueur, soit en cours). Il peut soit rejoindre une partie en attente d'un joueur soit créer une nouvelle partie.

4. L'administrateur peut configurer le serveur (paramètres des parties).

2.6 Base de données

La base de données stocke et partage les informations suivantes entre les joueurs et l'administrateur :

- Informations des joueurs (nom d'utilisateur, mot de passe, rôle, points totaux gagnés)

- Informations des parties (joueurs, points remportés par chacun)
- Paramètres de l'application (ce qui peut être configuré par l'administrateur)
- Logs de l'application

3 Analyse

3.1 Partage des responsabilités entre le serveur et le client

3.1.1 Responsabilités du client

Le client est en charge de transmettre les actions effectuées par le joueur au serveur. Ces actions sont en fait les touches appuyées qui correspondent à une action spécifique dans le jeu (comme par exemple aller à droite ou à gauche, ou envoyer un obstacle à un endroit spécifique). Le client doit aussi gérer l'affichage du jeu, en récupérant les emplacements de tous les éléments du jeu (skieur, obstacles fixes et obstacles dynamiques) à partir du serveur.

3.1.2 Responsabilités du serveur

Le serveur lui doit gérer la logique du jeu. Pour chaque action effectuée par un des joueurs, il va mettre à jour les emplacements des éléments du jeu et vérifie si la partie est gagnée (ou perdue) en regardant s'il y a eu une collision ou si le skieur est arrivé en bas de la piste de ski. Il fait ces mises à jour de manière asynchrone, mais il va contacter les clients de manières synchrones (toutes les x millisecondes) pour que les deux joueurs aient le jeu dans le même état chacun.

3.2 Diagramme d'activité général

La figure 2 représente le diagramme d'activité qui illustre la répartition des responsabilités entre le client et le serveur.

3.3 Rôle des participants

Joueurs : 2 personnes par partie, le *skieur* ayant pour but de réussir à descendre la piste, tandis que l'autre essaye de l'en empêcher (*défenseur*).

Administrateur : Peut changer les règles générales du jeu (configuration des difficultés, tailles des cartes, etc...), et s'occuper des données (nettoyer, modifier des joueurs, etc...)

3.4 Cas d'utilisation

3.4.1 Diagramme général de contexte

3.4.2 Description des acteurs

1. Joueurs, dans une partie nommée : *Skieur* et *Défenseur*
2. Administrateur

3.4.3 Scénario principal

1. L'un des deux joueurs se connecte au lobby et crée une partie en spécifiant les paramètres qu'il souhaite.
   (a) Difficulté
   (b) Taille de la carte
   (c) Son rôle (*Skieur* ou *Défenseur*)
2. Le second joueur rejoint la partie libre du joueur 1 dans le lobby et prend le rôle restant.
3. Lorsque les 2 joueurs sont prêts, la partie commence.
   (a) "*Le skieur*" tente de traverser la piste en déplaçant son personnage de haut en bas.
   (b) "*Le défenseur*" tente de l'en empêcher en envoyant des obstacles de gauche à droite.
   (c) La partie se termine si le skieur n'a plus de vie ou si il a atteint l'autre côté (le but).
4. À la fin de la partie, soit les rôles s'inversent et on recommence une partie à l'étape 3, soit les joueurs quittent la partie. Cette dernière est donc interrompue et les joueurs retournent au lobby.

3.4.4 Extensions (ou scénarios alternatifs)

Pour permettre la récupération des parties de manière correcte, il faut s'assurer que tous les états et les événements sensibles du système peuvent être récupérés à n'importe quelle étape du scénario.

Lors d'une partie, si l'un des deux joueurs est déconnecté, l'adversaire gagne automatiquement la partie et est renvoyé au lobby.

3.4.5 Scénario d'administration

1. l'administrateur se connecte à l'interface.
2. Il modifie l'une des options à sa disposition :
   (a) Les paramètres du jeu.
   (b) Les données d'un joueur.
3. puis il peut se déconnecte ou recommencer à l'étape 2.

3.5 Modèle de domaine

La figure 4 représente une ébauche du modèle de domaine.

3.5.1 Modèle de domaine pour le client

3.5.2 Modèle de domaine pour le serveur

3.6 Base de données

L'objectif de la base de données est principalement de stocker les paramètres du jeu et les données utilisateurs. Nous avons fait le choix de ne pas stocker d'informations concernant les parties en cours qui seront chargées uniquement dans la mémoire du serveur.

On trouve donc une entité `User` qui peut être `Administrator`, dont on stocke le login et le nombre de victoires. `Settings` modélise les réglages généraux du jeu, `MapSize` permettra à l'utilisateur qui crée une partie de choisir parmi plusieurs tailles de carte. `DifficultyLevel` modélise les niveaux de difficulté parmi lesquels l'utilisateur créant la partie aura le choix.

3.6.1 Modèle conceptuel (entité-associations)

4 Conception du projet

4.1 Protocole d'échange entre le client et le serveur

4.1.1 Communication

Nous allons utiliser JSON [1] comme protocole d'échange entre le client et le serveur.

En effet, de par la structure de son contenu, l'échange d'information entre les deux intervenants est très facilement sérialisable/désérialisable, sans compter sur le fait que le contenu est lisible par un individu (contrairement à un contenu binaire).

4.1.2 Données échangées

L'API de communication en cours d'élaboration se trouve sur le wiki de notre projet GitHub
`https://github.com/gluthier/GEN-projet/wiki/Communication-API`
L'idée est que chaque joueur envoie au serveur ses commandes : pour le skieur s'il souhaite tourner à gauche ou à droite, pour le Valaisan sur quelle ligne il envoie un obstacle. Le serveur s'occupe de placer les obstacles et le skieur et envoie les positions de manière régulière aux deux clients. Les clients ne font que l'affichage et l'envoi de commande alors que le serveur s'occupe de la synchronisation, gestion des collisions, etc.

1. JSON : JavaScript Object Notation

4.2 Diagrammes de classes du serveur et du client

4.3 Modèle conceptuel & relationnel de la base de données

5 Implémentation du projet

5.1 Structure du projet

Pour partager les différentes parties de l'applicatif, nous avons séparé le code en plusieurs projets afin d'exposer que le code qui est nécessaire. Le client, par exemple, n'a pas besoin d'avoir accès au code qui gère la base de données ni à code de l'applicatif administrateur.

5.1.1 GEN-protocol

`GEN-protocol` définit le protocole utilisé par le client et le serveur pour communiquer entre eux.

5.1.2 GEN-BDD

`GEN-BDD` offre une interface pour communiquer avec la base de données du jeu.

5.1.3 GEN-server

`GEN-server` est la partie que gère la logique du jeu et la synchronisation des joueurs entre eux.

5.1.4 GEN-admin

`GEN-admin` est l'applicatif administrateur qui permet de gérer la base de données, ajouter des comptes utilisateurs, consulter les logs en offrant une interface graphique.

5.1.5 GEN-Frogger

`GEN-Frogger` est le code destiné au client qui gère l'affichage graphique du jeu, ainsi que les interactions du joueur.

5.2 Technologies, langages, bibliothèques utilisées

Ce projet a été réalisé entièrement en Java, que ce soit le client ou le serveur. Nous avons utilisé JavaFX pour gérer la partie graphique du jeu. Il s'agit d'une bibliothèque de création d'interfaces graphiques pour le langage Java. Elle contient notamment des outils pour du graphisme 2D que nous avons utilisés pour la gestion des éléments affichés à l'écran.

La bibliothèque TestFX a été utilisée pour mettre en place des tests qui fonctionnent avec JavaFX, car celui-ci demande a besoin que l'application s'exécute réellement pour fonctionner correctement. Les tests unitaires classiques (comme JUnit) ne peuvent pas être utilisés pour tester les parties gérées par JavaFX.

5.3 Technologies "originales" (autres que les technologies étudiées à l'école)

5.3.1 Guava

1. Descriptif Guava est une ensemble de bibliothèque open source développé par Google en Java. Nous l'utilisons principalement pour ses outils de hachage et sur les immutable.

2. Avantages Elles offrent pas mal d'outils au niveau des objets immutable, comme par exemple des table d'association immutable que nous utilisons, et des fonctionnalité au niveau du hachage

3. Limitations La librairie est lourde ce qui peut alourdir notablement le projet.

4. Remarques personnelles Très pratique et simple d'utilisations. Comme elle est utilisée par pas mal de monde, on n'a pas de problème à trouver des solutions aux éventuels problèmes que l'on pourrait rencontré avec.

5.4 Problèmes éventuels rencontrés et solutions apportées

5.4.1 Logique FX

Venant de *Swing*, nous avons eu pas mal de problèmes avec l'utilisation de *JavaFx* et des nouvelles logiques de conception que ça amener. Nous les avons peu à peu résolu, en reprenant bien ce que nous avions fait et en apprenant tout depuis le début.

5.4.2 ReadLine et $\backslash n$

Dans notre protocole d'échange nous utilisons du *json* pour communiquer les différents messages nécessaires au bon fonctionnement du jeu, afin de lire les messages nous utilisons la méthode *readLine* qui est bloquante et permet d'attendre sur une notification. Cependant nous avons remarqué que pour utiliser readLine, il fallait que nos messages soient bien terminé (via un $\backslash n$), pour que la commande nous retourne le message reçu et le traite. Nous avons donc ajouter à la fin des paquets *json* un $\backslash n$ pour nous assurer que cela fonctionnait bien.

5.4.3 Partage de classes/code entre plusieurs projets

Notre structure étant composé de 4 projets distincts, nous avions besoin de pouvoir partagé des fonctions entre eux de manière simple, et surtout sans copié-collé. Pour palier à ça, nous utilisons le système de gestion *maven*, ce qui nous permet de lié les projets entre eux, d'assurer une dépendance toujours à jour et de facilement construire nos projets.

6 Gestion du projet

6.1 Rôle des participants au sein du groupe de développement

1. Tony Clavien : Analyste, design manager, programmeur, représentant valaisan, responsable apéro
2. Maxime Guillod : Chef de projet, programmeur
3. Gabriel Luthier : Responsable des tests, programmeur
4. Guillaume Milani : Architecte, concepteur en chef, programmeur, community manager

6.2 Plan d'itérations initial

6.2.1 Création de l'interface de base

Objectif : Mise en place de l'interface utilisateur de l'écran principal du jeu.

Durée : 1 semaine

Date du rendu : 26.04.2017

Partage des tâches : Création des images/sprites, affichage de l'UI, mise en place des obstacles fixes

Effort : 15 heures

6.2.2 Ajout du 1er joueur avec les obstacles

Objectif : Développement de la logique du joueur "skieur" qui descend la piste.

Durée : 1 semaine

Date du rendu : 03.05.2017

Partage des tâches : interaction avec les touches du clavier, déplacement du skieur sur la carte

Effort : 20 heures

6.2.3 Ajout du 2ème joueur en local

Objectif : Développement de la logique du joueur "défendant" qui envoie des obstacles contre l'autre joueur.

Durée : 1 semaine

Date du rendu : 10.05.2017

Partage des tâches : interaction avec les touches du clavier, logique de déplacement des obstacles sur la carte

Effort : 20 heures

6.2.4 Mise en place des logiques de jeu

Objectif : Détection des collisions entre les obstacles (fixes et mobiles) avec le joueur "skieur", des conditions de victoire et défaite et barre de recharge pour la pose d'obstacles.

Durée : 1 semaine

Date du rendu : 17.05.2017

Partage des tâches : détection de collisions, conditions de victoire, conditions de défaite, barre de recharge pour la pose d'obstacles

Effort : 20 heures

6.2.5 Ajout de la communication avec le serveur

Objectif : Développement du serveur pour pouvoir jouer à distance.

Durée : 1 semaine

Date du rendu : 24.05.2017

Partage des tâches : connexion des deux joueurs à une partie, communication réseau entre le serveur et les deux joueurs, gestion de l'état d'une partie

Effort : 30 heures

6.2.6 Application administrateur

Objectif : Développement de l'application administrateur utilisée afin de modifier les paramètres des différents modes de jeu et des données des joueurs.

Durée : 1 semaine

Date du rendu : 31.05.2017

Partage des tâches : UI de l'applicatif, connexion à la base de données, fonctions de modification de la BD

Effort : 20 heures

6.2.7 Ajout du Lobby

Objectif : Gestion du lobby (UI et logique).

Durée : 1 semaine

Date du rendu : 07.06.2017

Partage des tâches : UI du lobby (menu), récupérer les informations sur les parties en recherche de joueurs, création d'une nouvelle partie

Effort : 20 heures

6.2.8 Finalisation

Objectif : Finalisation de l'application et ajout de bonus.

Durée : 1 semaine

Date du rendu : 14.07.2017

Partage des tâches : derniers détails, mise en places des bonus/easter eggs

Effort : 15 heures

6.3 Suivi du projet

6.3.1 Itération 1 : Création de l'interface de base

Cette itération a été consacrée à la découverte de la bibliothèque JavaFX pour afficher l'interface de base. L'image du fond et les différents sprites du jeu ont été mis en place, ainsi que la grille où tous les éléments seront affichés. Pas de problèmes ont été rencontrés et donc aucune replanification n'est nécessaire.

6.3.2 Itération 2 : Ajout du 1er joueur avec les obstacles

Le premier joueur (le skieur) est mis en place et peut se déplacer à travers la grille (pas case par case, cela sera développé plus tard). Le premier problème rencontré a été d'écrire des tests qui fonctionnent avec JavaFX. En effet, des tests unitaires (JUnit) classiques ne fonctionnent pas, car JavaFX a besoin que le jeu s'afficher "en vrai" pour pouvoir s'exécuter correctement (par exemple pour répondre à un clique du joueur, il a besoin de savoir à quelle position il a cliqué pour éventuellement interagir avec l'élément cliqué). Aucune replanification a été faite, les tests seront simplement développés ensuite au fil des itérations.

6.3.3 Itération 3 : Ajout du 2ème joueur en local

La création des obstacles dynamiques qui sont créés par le 2ème joueur (à l'aide de son clavier) a été mise en place. Les collisions avec le skier ont aussi été développée, mais le code n'a pas encore été mergé à ce moment-là et les deux fonctionnalités marchent, mais séparément. La bibliothèque TestFX a été choisie et testée pour effectuer nos tests fonctionnant avec JavaFX. Pas de replanification a été faite, mais le bug des collisions devra être corrigé par la suite.

6.3.4 Itération 4 : Mise en place des logiques de jeu

Remise en question générale et changement du déplacement du skieur pour rendre la détection des collisions plus facile et aussi par anticipation à la mise en place du serveur. En effet, comme c'est lui qui devra gérer la logique du jeu, on diminue ainsi le nombre de requêtes qui lui seront transmises. Une grosse partie de travail effectué lors de cette itération à été de changer le fonctionnement interne du jeu, l'ajout de nouvelles fonctionnalités en a donc péri.

Cette fois-ci, une replanification est nécessaire. Les trois cas d'utilisation qui auraient dû être développés se voient partagés dans les itérations suivantes. L'UC *Jouer une partie* et l'UC *S'identifie* sont reporté à l'itération 5 tandis que l'UC *Voir ses statistiques* se reportent à l'itération 6.

6.3.5 Itération 5 : Ajout de la communication avec le serveur

Cette itération nous a pris beaucoup plus de temps que prévu. On a décidé de se concentrer sur le case d'utilisation *Jouer une partie*, qui constitue le cœur du jeu. Il a été possible de finaliser ce case d'utilisation, mais cela nous a demandé de refactorer presque l'ensemble du projet pour implémenter les collisions entre les obstacles, la logique de victoire/défaite d'une partie et l'interface pour pouvoir relancer une partie.

L'UC *S'identifie* a quand même été entamée. La base de données qui contiendra les informations des utilisateurs, les logs ainsi que les paramètres du jeux a été mise en place. Il reste cependant à faire communiquer le client avec le serveur pour effectivement pouvoir s'identifier.

Pour pouvoir finir ce cas d'utilisation, on effectue une replanification et déplace l'UC *S'identifie* à l'itération suivante (la 6).

6.3.6 Itération 6 : Application administrateur

L'application de l'interface d'administration est développée avec une interface sommaire pour le moment. Un affichage des logs du serveur avec un système d'ID pour le suivi de chaque instance (de chaque joueur qui se connecte au serveur) est implémenté. L'interface du login est aussi mise en place, mais la communication avec le serveur n'est toujours pas fonctionnelle. Un gros travail sur les divers éléments de communication est fourni, mais rien n'est encore fonctionnel à 100%.

Une replanification est faite pour reporter tous les cas d'utilisation qui nécessitent de communiquer avec le serveur à l'itération suivante.

6.3.7 Itération 7 : Ajout du Lobby

Les paramètres des jeux sont maintenant configurables dans un onglet dans l'interface administrateur. Tous ces paramètres sont fonctionnels, enregistrés dans la base de données et communiqués par la suite au client par le serveur. Le principal du travail a été fourni sur le backend, il n'y a donc pas grand-chose de nouveau à montrer. Le problème principal est que tout a été très bien avancé, mais rien à 100%. On a donc pour l'instant une application qui n'est pas fonctionnelle.

La replanification est plutôt simple : on reporte tout pour la dernière itération.

6.3.8 Itération 8 : Finalisation

Pour cette itération, on a du travailler sur tout le projet en général, car il a fallut terminer de développer le maximum de fonctionnalité qu'il nous manquait et surtout corriger les bugs pour rendre l'application fonctionnelle.

6.4 Stratégie de tests

Nous avons effectué quelques tests pour les parties importantes de l'application. Généralement, la personne qui développait une fonctionnalité faisait elle-même les tests nécessaires pour assurer le bon fonctionnement du programme. Tout n'a pas été testé, car on a été rattrapé par les deadlines.

Pour faire ces tests, on a utilisé un mélange de JUnit pour tout ce qui est une application Java uniquement et TestFX pour les parties utilisant JavaFX (il s'agit de la partie client du jeu).

6.4.1 Utilisation de JUnit pour une classe définissant le protocole

```
1  package ch.heigvd.protocol;
2
3  import com.google.common.hash.Hashing;
4  import org.json.JSONArray;
5  import org.json.JSONObject;
6  import org.junit.Test;
7  import java.nio.charset.StandardCharsets;
8  import java.util.LinkedList;
9
10 import static org.junit.Assert.*;
11
12 /**
13  * @author Tony Clavien
14  * @author Maxime Guillod
15  * @author Gabriel Luthier
16  * @author Guillaume Milani
17  */
18 public class ProtocolTest {
19
20     @Test
21     public void testFormatArraytoJson() throws Exception {
22         LinkedList<Sendable> testList = new LinkedList<Sendable>();
23         testList.add(new Skier(0, 1));
24         testList.add(new Difficulty(1, "medium", 5, 4, 3, 20));
25
26         String message = Protocol.formatArraytoJson(testList);
27         JSONArray testArray = new JSONArray(message);
28
29         JSONObject testSkierObject = testArray.getJSONObject(0);
30         JSONObject testDifficultyObject = testArray.getJSONObject(1);
31
32         assertEquals(0, testSkierObject.get("x"));
33         assertEquals(1, testSkierObject.get("y"));
34
35         assertEquals(1, testDifficultyObject.get("id"));
36         assertEquals("medium", testDifficultyObject.get("name"));
37         assertEquals(5, testDifficultyObject.get("manaRegenerationSpeed"));
38         assertEquals(4, testDifficultyObject.get("playerMoveSpeed"));
39         assertEquals(3, testDifficultyObject.get("obstacleMoveSpeed"));
40         assertEquals(20, testDifficultyObject.get("obstacleWidth"));
41     }
42
43     @Test
44     public void testGetJsonParam() throws Exception {
45         // {"command":"login","param":{"user":"maxime","password":"coucou"}}
46         String message = "{\"command\":\"login\",\"param\":{\"user\":\"maxime\",\"password\":\"
               coucou\"}}";
47
48         assertEquals("maxime", Protocol.getJsonParam(message, "param", "user"));
49         assertEquals("coucou", Protocol.getJsonParam(message, "param", "password"));
50
51         assertEquals("login", Protocol.getJsonParam(message, null, "command"));
52     }
53
54     @Test
55     public void testFormatLoginSend() throws Exception {
56         String message = Protocol.formatLoginSend("test", "1234");
57
```

```
58          JSONObject test = new JSONObject(message);
59          assertEquals("login", test.getString("command"));
60
61          JSONObject param = test.getJSONObject("param");
62          assertEquals("test", param.getString("user"));
63          assertEquals(Hashing.sha256().hashString("1234", StandardCharsets.UTF_8).toString(),
                param.getString("password"));
64      }
65
66      @Test
67      public void testGetFormatLoginUser() {
68          String message = Protocol.formatLoginSend("test", "1234");
69          assertEquals("test", Protocol.getFormatLoginUser(message));
70      }
71
72      @Test
73      public void testGetFormatLoginPassword() {
74          String message = Protocol.formatLoginSend("test", "1234");
75          assertEquals(Hashing.sha256().hashString("1234", StandardCharsets.UTF_8).toString(),
                Protocol.getFormatLoginPassword(message));
76      }
77
78      @Test
79      public void testFormatWrongLoginAnswer() throws Exception {
80          String message = Protocol.formatWrongLoginAnswer();
81          JSONObject test = new JSONObject(message);
82          assertEquals("", test.getString("token"));
83      }
84
85      @Test
86      public void testFormatLoginAnswer() throws Exception {
87          LinkedList<Difficulty> difficulty = new LinkedList<Difficulty>();
88          LinkedList<MapSize> map = new LinkedList<MapSize>();
89
90          difficulty.add(new Difficulty(1, "medium", 5, 4, 3, 20));
91          map.add(new MapSize(1, "medium", 200, 100));
92
93          String answer = Protocol.formatLoginAnswer("1234", difficulty, map);
94
95          JSONObject test = new JSONObject(answer);
96          assertEquals("1234", test.get("token"));
97          JSONArray difficulties = test.getJSONArray("difficulties");
98          JSONArray mapSizes = test.getJSONArray("mapSizes");
99
100         assertEquals(difficulty.get(0), new Difficulty(difficulties.getJSONObject(0)));
101         assertEquals(map.get(0), new MapSize(mapSizes.getJSONObject(0)));
102     }
103
104     @Test
105     public void testGetFormatLoginToken() throws Exception {
106         // {"token":"9164108374"}
107         String message = "{\"token\":\"9164108374\"}";
108         assertEquals("9164108374", Protocol.getFormatLoginToken(message));
109     }
110 }
```

6.4.2 Résultats des tests

La figure 6 montre le résultat des tests JUnit.

6.5 Stratégie d'intégration du code de chaque participant

Afin de collaborer efficacement, nous avons utilisé Git (avec la plateforme Github) pour gérer l'intégration du code de chaque développeur. Pour chaque fonctionnalité, une branche était créée pour permettre à différentes personnes de collaborer dessus. Puis, quand la fonctionnalité était finie, une pull request était ouverte sur Github pour permettre aux autres membres du groupe d'analyser et critiquer (si besoin) les changements effectués. Une fois tout le monde satisfait, la

pull request était mergée sur la branche master qui correspond à l'ensemble de fonctionnalités finies.

7 État des lieux

7.1 Ce qui fonctionne (résultats des tests)

7.2 Ce qu'il resterait à développer (en proposant une planification)

8 Auto-critique

Notre principale difficulté a été de commencer à développer l'application en local puis ensuite de vouloir la transformer pour pouvoir jouer à travers un serveur. Beaucoup d'effort a été fourni pour la rendre fonctionnelle à deux joueurs sur un ordinateur et il a fallu presque tout refaire la logique du jeu pour pouvoir communiquer avec le serveur.

Un autre aspect qu'on aurait pu mieux faire a été notre dispersion. Au fil des itérations et surtout des replanifications, la charge de travail c'est accumulée et on avait de plus en plus de fonctionnalités à développer. Au lieu de se focaliser sur l'essentiel, on a profité d'être 4 et à chaque fois on a partagé le travail dans l'espoir de tout finir. On a donc pas pu développer toutes les fonctionnalités que l'on avait prévu lors de la planification du projet.

On a aussi eu plus de peine à gérer la fin de semestre et tous les rendus des autres cours qui viennent en même temps. Pour améliorer ça, on aurait pu, lors des replanifications, oser redéfinir les fonctionnalités pour enlever celles qui ne sont pas essentielles (notamment les statistiques ou le lobby par exemple) pour pouvoir se concentrer sur l'aboutissement du cœur du jeu.

9 Conclusion

Pour conclure, ce projet aura été une belle expérience. En effet, on a beaucoup appris des difficultés qui peuvent survenir lorsque l'on entreprend de développer un projet de plus grande envergure que ce à quoi on a été habitué de développer de notre côté. Une des leçons a été l'importance d'une bonne planification et replanification si besoin. On a plutôt eu tendance à simplement repousser le travail des itérations qui n'avait pas été fini aux itérations suivantes sans réfléchir vraiment si c'était une bonne idée.

Malgré l'utilisation d'outils tels que Trello et Github, la collaboration n'a pas été aisée. Planifier les itérations et ensuite partager le travail demande pas mal de temps. Ensuite, plus le projet avance, plus l'on est dépendant du bon développement des collègues. Il nous est arrivé de nous retrouver dans la situation que, malgré l'accès à l'historique des modifications apportées par chacun, de remodifier le changement et ainsi revenir en arrière parce que l'on n'avait pas aperçu la raison du changement plus loin dans le code.

On peut également ajouter que le temps nécessaire à une programmation propre, claire et facilement adaptable par la suite a été négligé. En effet, bon nombre de changements ont du s'effectuer a posteriori de leur développement, ce qui de par la qualité du code des premières itérations, nous

à demandé un effort supplémentaire et non prévu.

Malgré tout, même si l'on n'a pas réussi à terminer entièrement le projet tel que l'on avait planifié en début de semestre, on a pu développer une bonne partie, en prenant soin de bien structurer notre application. On a pu mettre en place une base de données ainsi qu'une API permettant d'y accéder sans devoir écrire toutes les requêtes SQL "à la main". Notre partie administration fonctionne bien aussi. On peut accéder aux logs de tous ce qu'il se passe dans l'application, ce qui est utile non seulement en phase debug, mais aussi lors du cours normal pour pouvoir vérifier que tout fonctionne correctement. Le protocole que l'on a défini est aussi bien abouti. Le fait de l'avoir séparé du client et du serveur nous permet de simplement le définir une seule fois et laisser ensuite et le client et le serveur le soin d'aller chercher les informations qu'ils ont besoin pour communiquer entre eux. Et surtout, on peut effectivement jouer en réseau sur des machines différentes. Le jeu n'est pas extrêmement équilibré, mais au moins il est jouable.

En somme, ce projet nous a beaucoup appris sur le cycle complet (accéléré dans notre cas) d'un développement à partir de la phase d'initialisation jusqu'à la fin de celle de construction, en passant par la phase d'élaboration et en s'arrêtant avant celle de transition qui serait celle où l'on livre l'application finale au client. Ces connaissances sont très utiles pour la suite dans le monde professionnel.

A Manuel d'utilisation

1. Installation
2. Utilisation (avec copies d'écran)

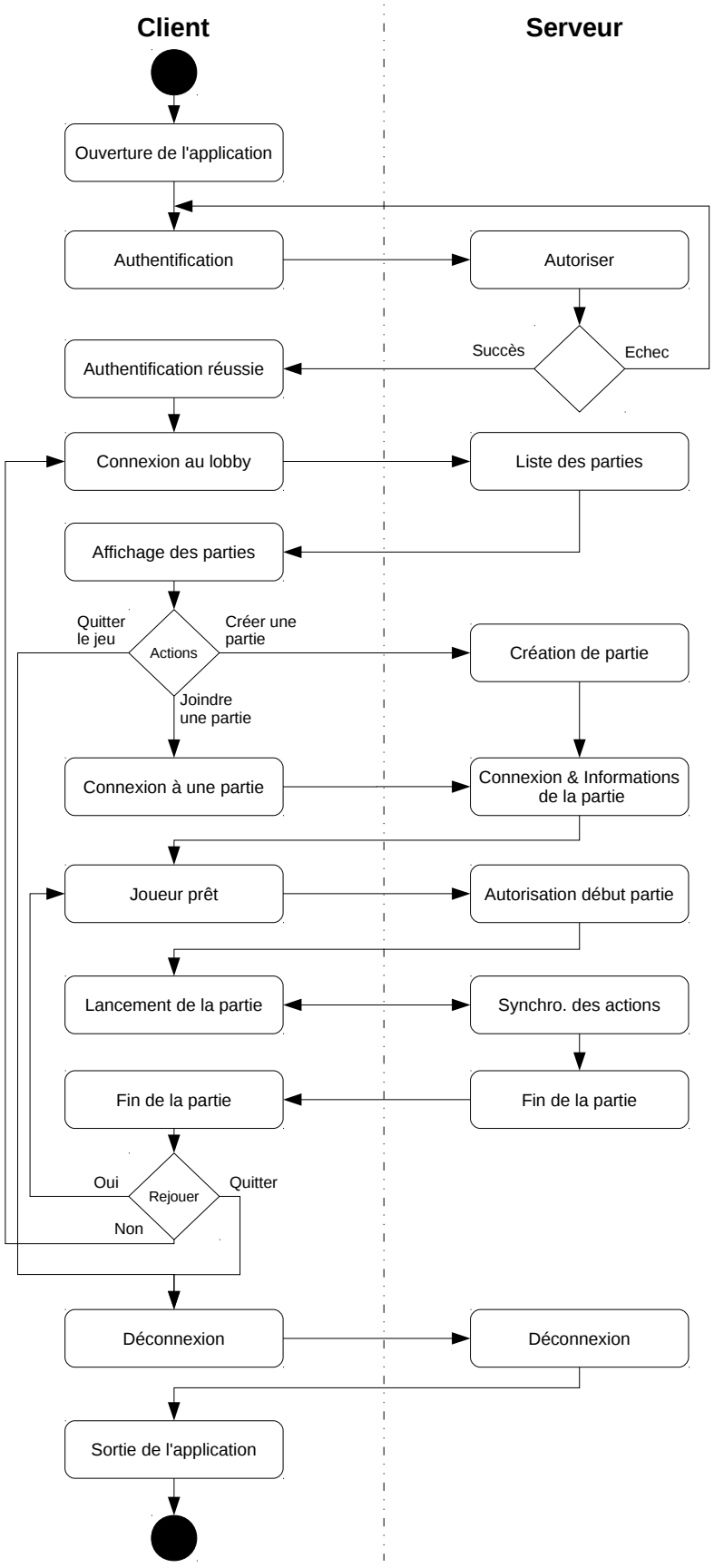

FIGURE 2 – Diagramme d'activité

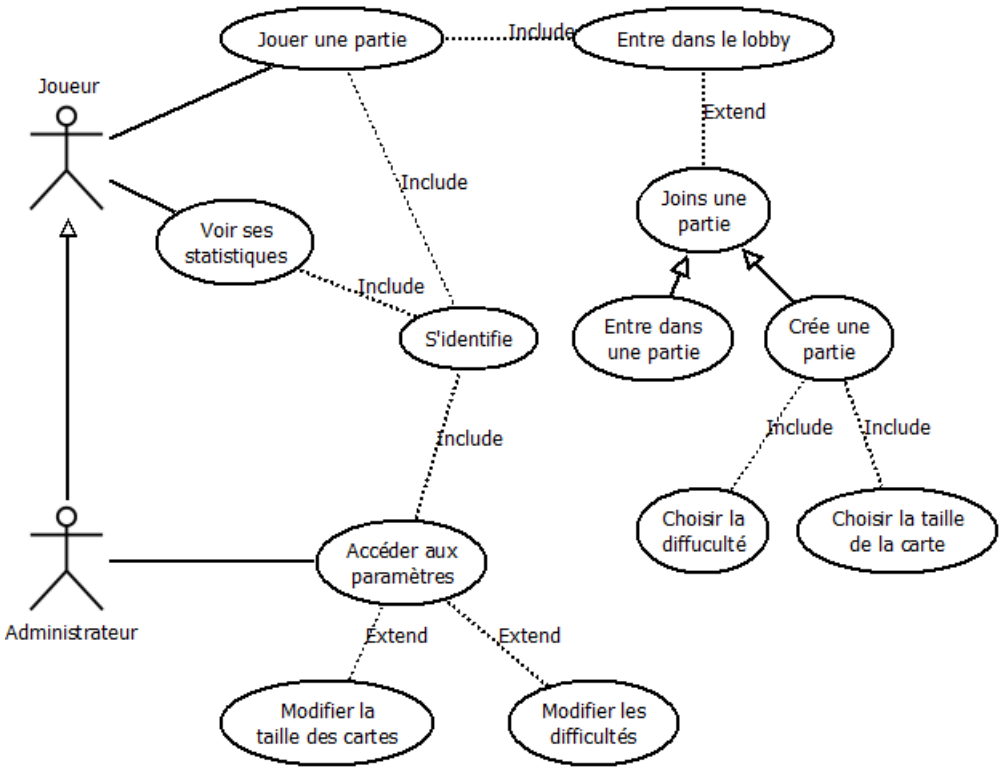

FIGURE 3 – Diagramme d'utilisation

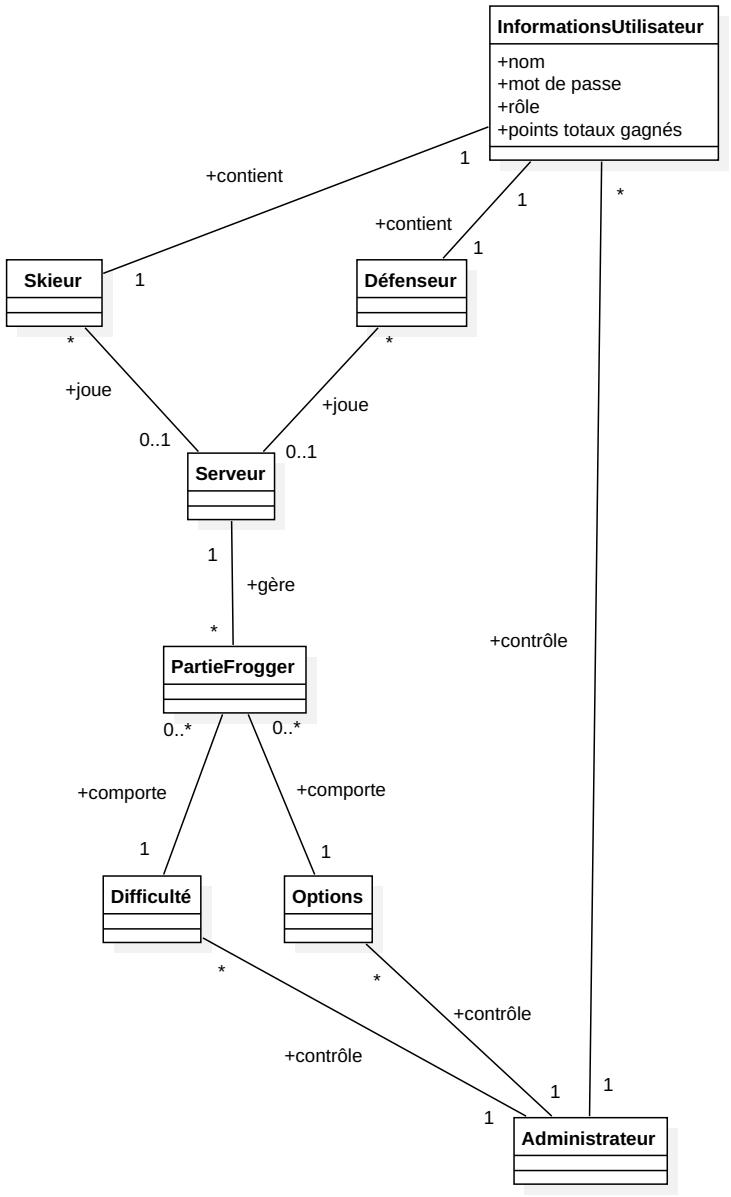

FIGURE 4 – Modèle de domaine

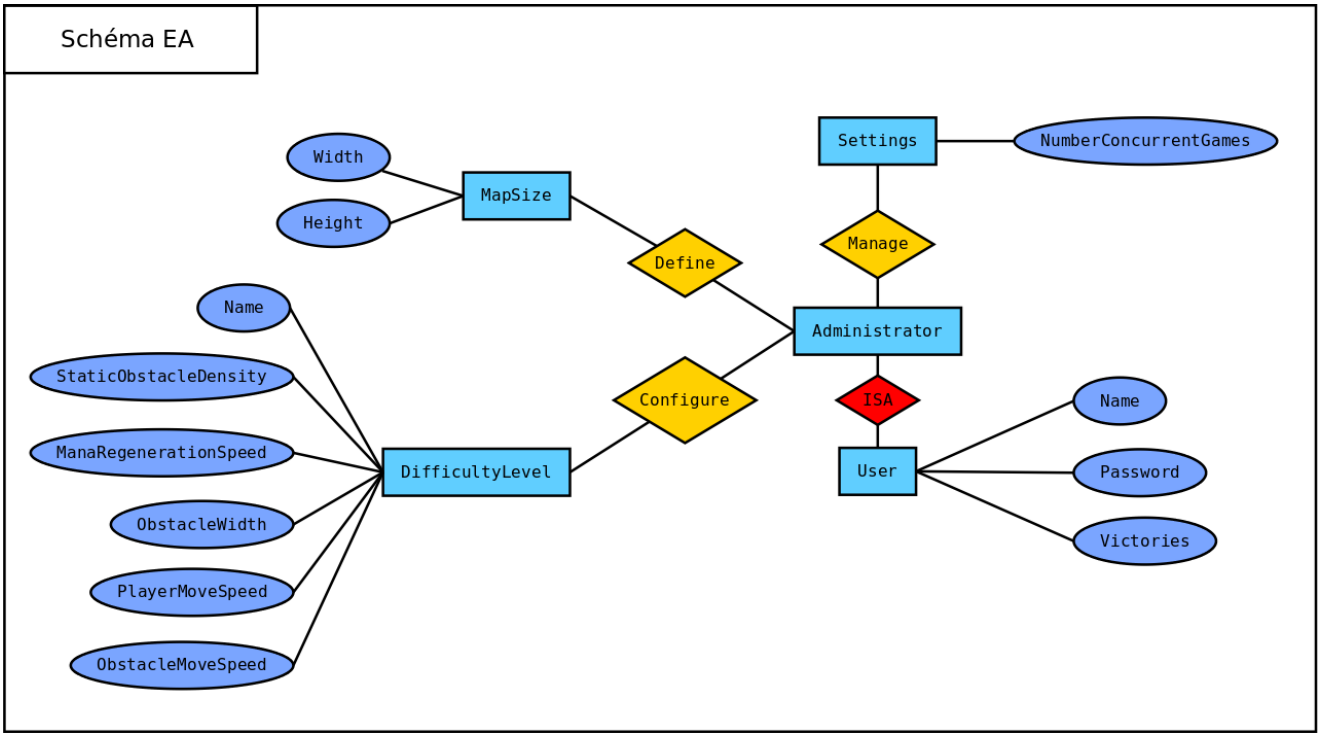

FIGURE 5 – Modèle conceptuel

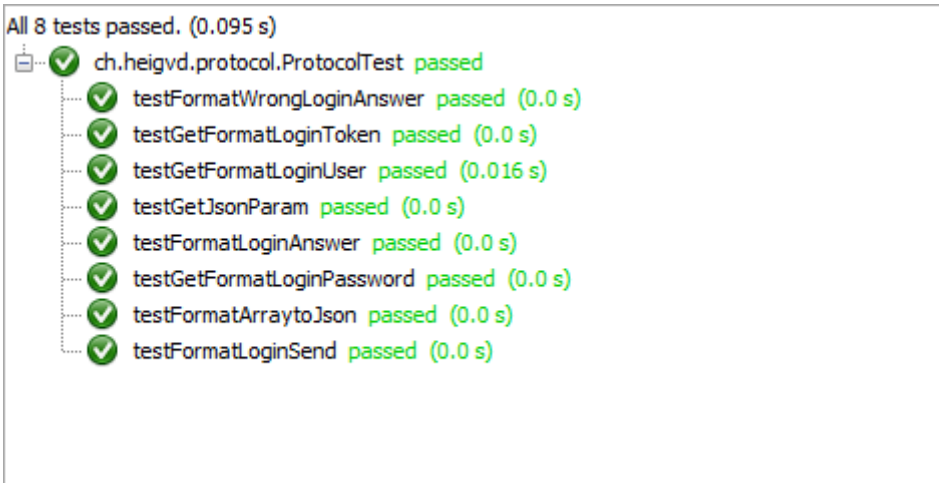

FIGURE 6 – Résultats des tests JUnit