

# WARTANKS

## Rapport de projet de GEN

---

<b>Auteurs :</b>	Armand Delessert Simon Baehler Benoit Zuckschwerdt Ngueukam Djeuda Wilfried Karel
<b>Destinataires :</b>	Eric Lefrançois Sandy Vibert

---

# Sommaire

Introduction	4
But du projet	4
Déroulement d'une partie	5
Cas d'utilisation	6
Acteurs	6
Scénarios	7
Conception technique	8
Architecture générale	8
Technologies utilisées	9
Décomposition en 3 projets	9
Serveur	10
Architecture	10
Liste des classes	10
Diagrammes UML	12
Diagramme de classes du GameManager	12
Diagramme de la partie réseau	13
Communication réseau	14
Protocole	14
Liste des classes sérialisables	16
Le protocole, étape par étape	16
Base de données	18
Schéma de la base de données	18
Structure du fichier XML	19
Client	20
Architecture	20
List des classes	20
Diagramme UML	22
Début de jeu	23
En jeu	23
Fin de la partie	25
Bonus	26
Bonus speed	26
Mine	26

Ultra Rapide Fire (non implémenté)	26
Laser	26
Foudroiement (non implémenté)	26
Volée de lame	26
Bélier (non implémenté)	27
Alpha Strick	27
Soin	27
La Mort	27
Fenêtres	28
Liste des fenêtres du client	28
Interface du serveur	28
Implémentation du projet	28
Technologies utilisées	28
Tiled	28
Slick2D	28
Gestion du projet	29
Rôles des participants	29
Simon Baehler	29
Suivi du projet	29
Bilans des itérations	29
Stratégie de test	30
Client	30
Stratégie d'intégration du code de chaque participant	32
Etat des lieux	32
Côté serveur	32
Côté client	32
Travail restant	33
Côté client	33
Autocritiques	34
Simon Baehler	34
Armand Delessert	34
Ngueukam Djeuda Wilfried Karel	34
Benoit Zuckschwerdt	34
Conclusions personnelles	35

Simon Baehler	35
Armand Delessert	35
Ngueukam Djeuda Wilfried Karel	36
Benoit Zuckschwerdt	36
Annexes	37
Table des illustrations	37

## Introduction

---

Nous avons réalisé ce travail pour le cours de génie logiciel (GEN), dans le cadre du mini-projet. Le programme développé est un jeu de tanks en deux dimensions en vue de dessus et opposant deux joueurs. Le programme est décomposé en deux parties, un client et un serveur qui gèrent chacun un aspect du jeu :

- Le client sert d'interface graphique pour le joueur. Il se connecte au serveur pour rejoindre une partie. Le joueur n'a pas besoin de compte pour se connecter au serveur. Il suffit juste de connaître l'IP de ce dernier.
- Le serveur gère la partie et la synchronisation des clients. Il permet à 2 clients de se connecter pour une partie.

Ce projet a été réalisé dans le cadre du cours de génie logiciel (GEN) dispensé en 2<sup>ème</sup> année à l'HEIG-VD. D'une durée d'un semestre à raison de 6 heures de travail recommandées/estimées par personne et par semaine, ce projet a été réalisé par les étudiants suivants :

- Armand Delessert (chef du groupe)
- Benoit Zuckschwerdt
- Baehler Simon
- Ngueukam Djeuda Wilfried Karel

## But du projet

---

Le but de ce projet est d'exercer le développement en équipe et de gérer les contretemps qui peuvent survenir lors du développement d'un tel projet. Le développement doit être planifié à l'avance en itérations d'une ou deux semaines. Il faut donc découper le travail en petites parties et répartir la charge de travail entre les itérations. Ensuite, il faut estimer le temps de travail nécessaire pour chaque itération. Lors du développement, il faudra bien sûr essayer le plus possible de respecter les échéances de chaque itération. Si le travail d'une itération n'est pas achevé à temps, une replanification est effectuée. Si le retard ne peut pas être rattrapé, il est possible de retirer une partie des fonctionnalités du programme pour essayer de revenir à une charge de travail raisonnable.

## Déroulement d'une partie

---

### *Démarrage de la partie :*

L'admin démarre une partie depuis le serveur. Une fois la partie créée, les joueurs rejoignent le serveur en saisissant l'IP de celui-ci. Lorsque tout le monde est prêt, l'admin lance la partie.

### *Déroulement de la partie :*

Au début de la partie, chaque joueur apparaît sur la carte. Tous les joueurs possèdent au départ un tank avec une barre de vie. Les joueurs peuvent se déplacer sur les parties libres de la carte (là où il n'y a pas d'obstacle) et tirer. Les déplacements sont limités aux 4 directions nord, sud, ouest, est. Les tirs sont limités à un seul obus à la fois, ce qui veut dire que lorsqu'un joueur tir un coup il ne peut tirer un second coup seulement lorsque le premier tir aura atteint un adversaire ou un obstacle. Chaque tir reçu inflige des dégâts au tank. Lorsque le tank est détruit, le joueur a perdu. Le dernier joueur en vie gagne la partie. S'il reste plus d'un joueur en vie et que le temps de jeu est écoulé, la partie se termine.

### *Les bonus :*

Pendant la partie, des bonus apparaissent sur la carte et les joueurs peuvent les ramasser. Un bonus donne certains avantages au joueur tels qu'un tir plus rapide ou une arme différente ou un bonus de soin.

### *Fin de la partie :*

La partie prend fin dans les 3 cas suivants :

- Lorsqu'il ne reste plus qu'un joueur en jeu.
- Lorsque le temps de la partie est écoulé.
- Lorsque l'admin met fin à la partie depuis le serveur.

Dans les 2 premiers cas, la fenêtre de jeu se ferme et le tableau des scores s'affiche à l'écran de chaque joueur. Dans le troisième cas, la partie s'arrête et l'admin peut choisir de relancer une partie ou de fermer le serveur.

## Cas d'utilisation

La Figure 1 présente le diagramme des cas d'utilisation du programme.

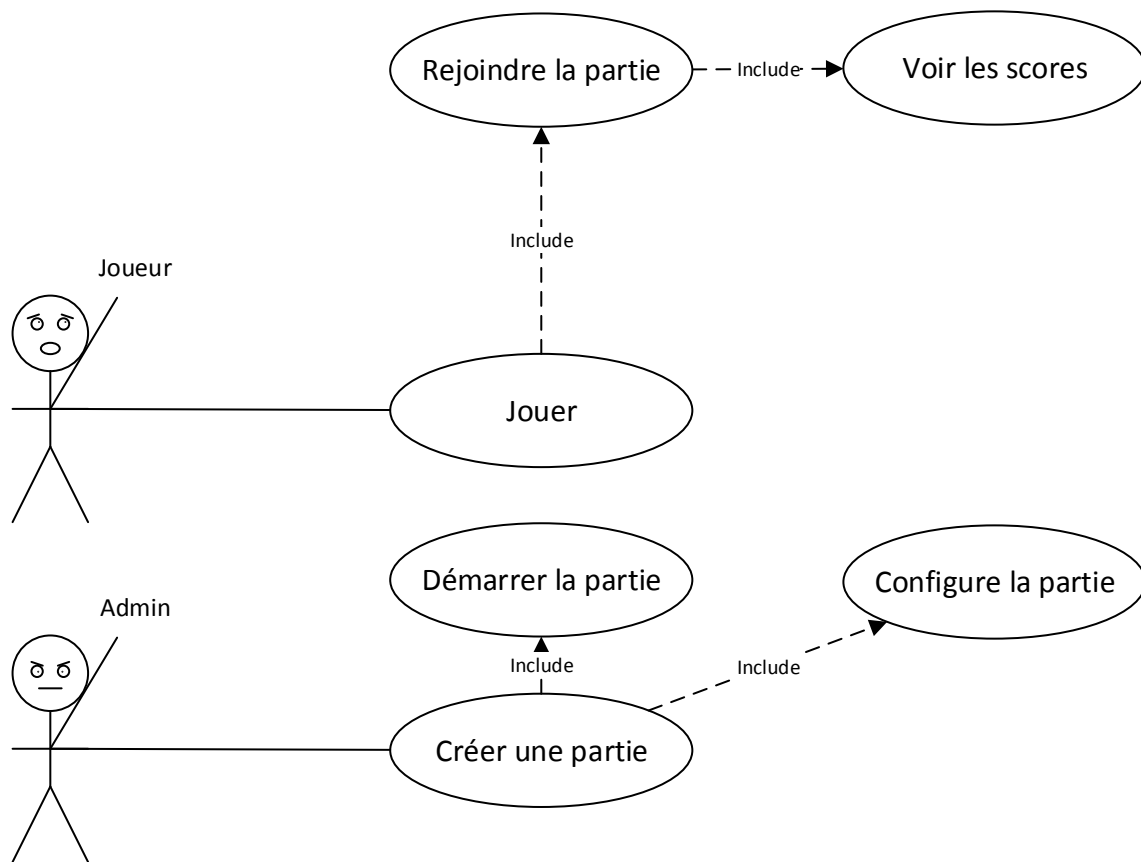


Figure 1

### Acteurs

#### Admin :

Il crée le serveur (nouvelle partie) et le configure. Une fois configuré et lorsque tous les joueurs se sont connectés, il peut lancer la partie.

#### Joueur :

Il rejoint une partie et joue. À la fin de la partie, les scores sont affichés à l'écran.

## Scénarios

1. **Le joueur admin crée une partie :**
  - 1.1. Le joueur admin crée le serveur (le serveur gérera la partie).
  - 1.2. Une fois le serveur configuré, le joueur admin rejoint les autres joueurs (~~choix du tank, etc.~~).
2. Le joueur admin **configure la partie**.
3. **Le joueur rejoint la partie** en se connectant au serveur.
4. Lorsque tous les joueurs ont rejoint la partie, **le joueur admin peut démarrer la partie**.
5. Pendant la partie, **les joueurs jouent** chacun un tank.
  - 5.1. Le joueur peut déplacer son tank.
  - 5.2. Le joueur peut tirer.
  - 5.3. Le joueur peut ramasser des bonus sur la carte.
  - 5.4. Le serveur gère la synchronisation entre les clients.
    - 5.4.1. Plusieurs joueurs peuvent rejoindre la partie et jouer.
    - 5.4.2. Les déplacements sont synchronisés entre les clients.
    - 5.4.3. Les tirs sont synchronisés entre les clients.
6. À la fin de la partie, **le tableau des scores est affiché** chez tous les participants.



## Conception technique

---

### Architecture générale

L'application se compose d'un serveur et d'un client, comme illustré dans la Figure 2. Le découpage des tâches entre le client et le serveur est présenté ci-dessous.

#### *Le client :*

- Se connecte à un serveur pour jouer à une partie.
- Récupère les appuis sur les touches effectués par le joueur et transmet au serveur les commandes correspondantes (avancer, tirer, etc.).
- Reçoit du serveur les mises à jour du plan de jeu et rafraîchit l'affichage à l'écran.
- Reçoit du serveur l'état du jeu (joueurs détruits, points de vie restants, etc.).
- Reçoit du serveur le tableau des scores à la fin de la partie.

#### *Le serveur :*

- Écoute sur le port 1991 les demandes de connexion des clients.
- Lorsqu'un client se connecte, le serveur crée un serveur dédié au client dans un thread.
- Gère la partie (une partie à la fois).
- Reçoit des clients les commandes des joueurs (avancer, tourner, tirer, etc.).
- Contrôle la validité des commandes et met à jour le plateau de jeu.
- Retourne aux clients le plateau de jeu mis à jour.
- Envoie aux clients l'état du jeu (joueurs détruits, temps restant, fin de la partie, etc.).
- Permet la synchronisation entre les clients.
- En fin de la partie, le serveur envoie le tableau des scores aux clients.

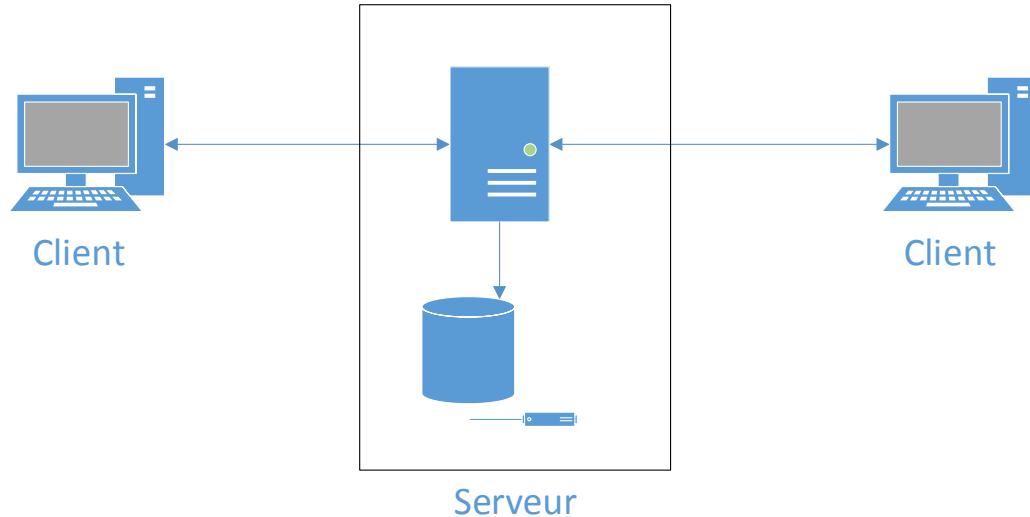


Figure 2

## **Technologies utilisées**

Les technologies utilisées dans ce projet sont les suivantes :

- Langage de programmation : Java
- Interface graphique des fenêtres : Swing
- Interface graphique du jeu : Slick2D
- Base de données : fichiers XML

Nous nous sommes arrêtés sur ces technologies soit parce que nous les connaissions déjà, soit parce que nous les considérons comme étant les plus adaptées à notre utilisation.

## **Décomposition en 3 projets**

Le programme a été décomposé en 3 projets Java développés à l'aide de l'IDE NetBeans. Les 3 projets sont le serveur gérant la synchronisation du jeu, le client graphique et le protocole de communication. Le serveur et le client devant être 2 applications séparées, il était nécessaire de créer 2 projets différents. Cependant pour le debug de la communication client-serveur il aurait été beaucoup trop difficile de jongler entre 2 projets différents. C'est pourquoi nous avons créé un troisième projet nommé « ClientServer » qui regroupe tout le nécessaire à la communication client-serveur. Tout le protocole de communication avec les classes sérialisables et les méthodes nécessaires s'y trouve. L'avantage de cette manière de procéder est que comme le protocole doit être commun au client et au serveur, il suffit d'inclure le projet « ClientServer » en tant que ressource externe au projet client et au projet serveur. Ainsi, le code n'est pas dupliqué et la modification du protocole en est simplifiée.

## Serveur

---

### Architecture

Le serveur est composé de 2 parties qui répondent chacune à une tâche. La première partie est le « GameManager » qui s'occupe de la gestion de la partie et la seconde partie est le couple « ClientListener » et « ClientHandler » qui s'occupe de la communication réseau avec les clients.

### Liste des classes

- Package wartanks
  - WarTanks  
Classe générale qui démarre le serveur de jeu. Cette classe démarre également 2 clients pour les tests lors du debug du serveur.
- Package gamemanager
  - GameManager  
Serveur de jeu. Cette classe gère le déroulement de la partie, c'est-à-dire la vérification et la synchronisation des actions des clients.
    - Package map
      - Map
      - Cell
      - GlobalCell
      - Direction
      - Movable
    - Package player
      - Player
    - Package projectile
      - Projectile
      - Bullet
    - Package stategame
      - StateGameManager
    - Package vehicle
      - Vehicle
      - Tank
- Package network  
Ce package fait partie des dépendances externes du projet « WarTanks » (partie serveur). Il se situe dans le projet « ClientServer ».
  - Package server
    - ClientListener  
Le « ClientListener » est le serveur qui reste à l'écoute de nouveaux clients. Lorsqu'un client se connecte, il crée un « ClientHandler » qui prendra en charge le client.
    - ClientHandler  
Le « ClientHandler » est un serveur dédié à un client. Il s'occupe de recevoir les commandes envoyées par ce client et lui retourne les mises à jour de l'état du jeu.

- Package client
  - Client
 

Ce client sert de test pour la communication. Il implémente le protocole de communication et simule des commandes envoyées au serveur.
- Package protocol
  - CommunicationProtocol
 

La classe « CommunicationProtocol » regroupe toutes les méthodes nécessaires à la communication entre le client et le serveur par sérialisation des classes du package « protocol.messages ».
  - Package messages
 

Toutes ces classes seront transmises par le réseau et sont donc sérialisables.

    - InfoClient
 

Cette classe comporte plusieurs informations concernant le client telles que son adresse IP.
    - InfoPlayer
 

Cette classe comporte plusieurs informations concernant le joueur telles que son ID, son nom ou encore sa couleur.
    - Command
 

Cette classe abstraite permet de regrouper toutes les classes véhiculant une commande du client au serveur sous une même classe par l'héritage. Les classes héritant de la classe « Command » sont listées ci-dessous :

      - Movement
 

Cette classe contient la commande de déplacement du joueur (déplacement vers le haut, à droite, etc.).
      - Position
 

Cette classe contient la position du joueur.
      - Shoot
 

Cette classe informe le serveur que le joueur vient de tirer.
      - UseBonus
 

Cette classe informe le serveur que le joueur utilise un bonus.
    - TiledMapMessage
 

Cette classe contient la carte Tiled avec le design de la carte. Elle est transmise aux clients lors de l'initialisation de la partie.
    - StateGame
 

Cette classe contient tout ce qui est nécessaire pour connaître l'état de la partie. Elle contient entre-autre l'état et la position de chaque joueur, la position et la direction des tirs, la position des bonus, etc. Elle contient également le statut de la partie (en cours, terminée, temps restant, etc.).
    - Map
 

Cette classe contient la position des divers éléments de la carte. Elle est incluse dans la classe « StateGame ».
    - Message
 

Cette classe était utilisée pour le debug de la sérialisation des classes.

## Diagrammes UML

### Diagramme de classes du GameManager

La Figure 3 ci-dessous représente le diagramme de classes du GameManager. Il s'agit de la partie du serveur gérant la partie sans la communication réseau.

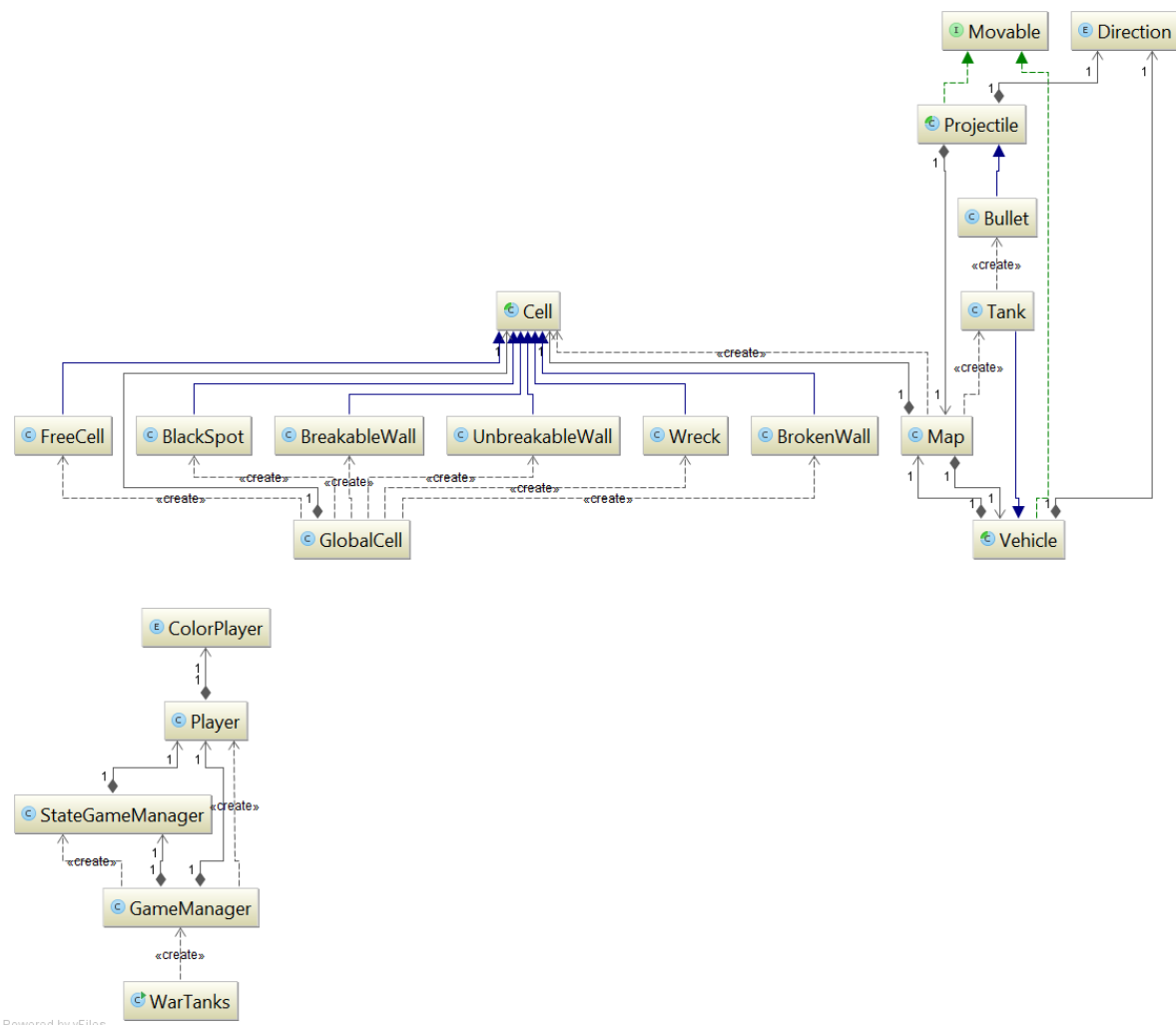


Figure 3

Le diagramme de classes avec les classes complètes (méthodes et attributs) est disponible en annexe.

### Diagramme de la partie réseau

La Figure 4 ci-dessous représente le diagramme de classes du protocole. Il s'agit de la partie gérant la communication réseau.

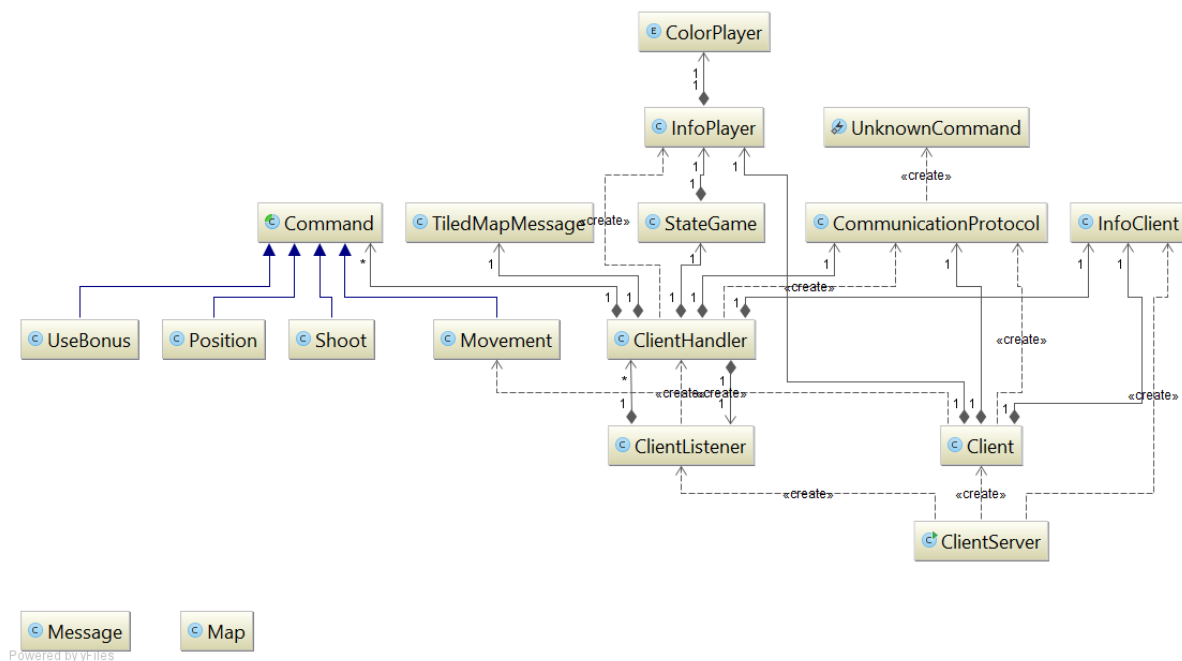


Figure 4

Le diagramme de classes avec les classes complètes (méthodes et attributs) est disponible en annexe.

## Communication réseau

### Protocole

Pour la conception du protocole réseau, nous sommes partis de l'objectif de créer un protocole très simple pour la communication client-serveur en cours de partie. Nous avons prévu d'utiliser la sérialisation de classes avec que 2 classes pour la communication en cours de partie. La première classe servira au client pour lui permettre d'envoyer les commandes au serveur. La seconde classe permettra au serveur d'envoyer l'état du jeu mis à jour aux clients. Ainsi, les clients peuvent à tout moment envoyer une commande que le joueur aurait entré et le serveur s'occupe de maintenir les clients à jour en leur envoyant régulièrement le nouvel état du jeu qui inclut les dernières actions des joueurs. Cependant, l'initialisation de la partie et des clients nécessite aussi l'envoi de plusieurs messages. Il a donc fallu créer plusieurs classes sérialisables pour permettre cette phase d'initialisation.

### *Diagramme d'activité du transfert de message entre le client et le serveur*

La Figure 5 représente le transfert d'un message du serveur au client et vice-versa.

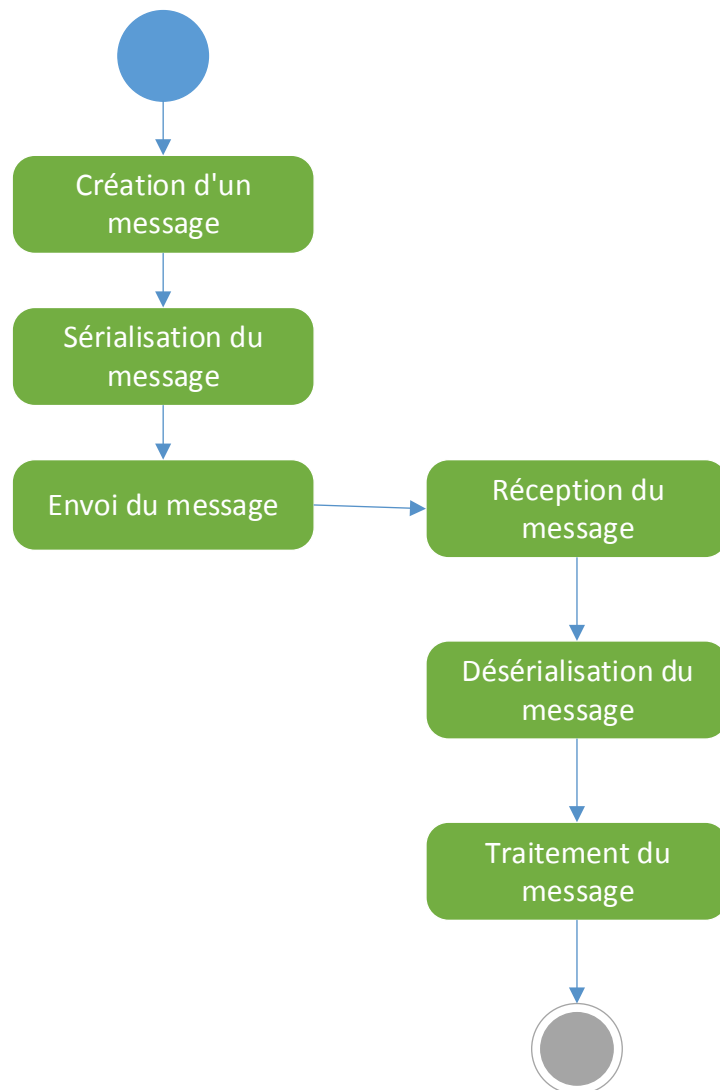


Figure 5

### Diagramme de séquence du transfert de message entre le client et le serveur

La Figure 6 présente le diagramme de séquence de la communication client-serveur.

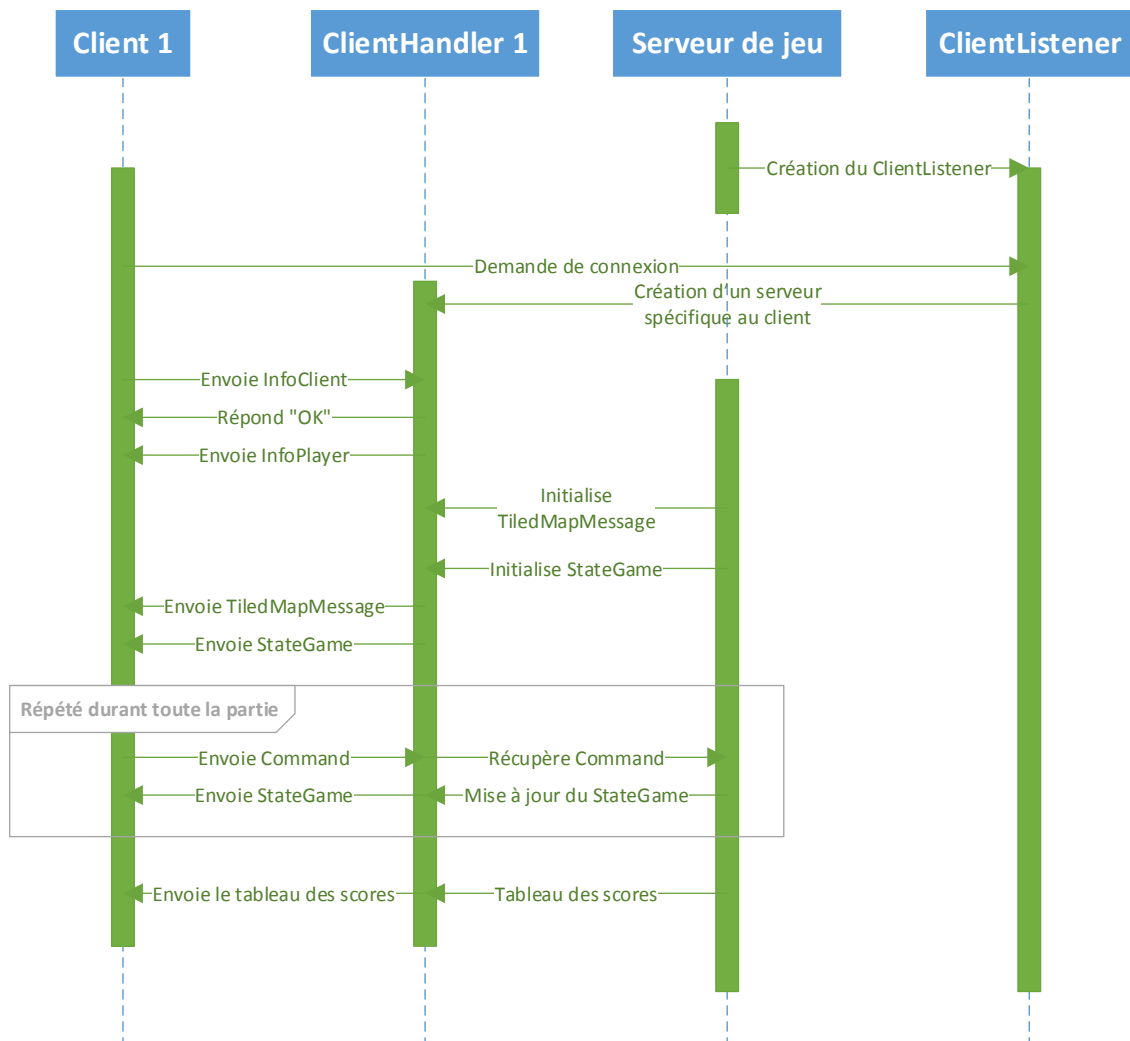


Figure 6



## Liste des classes sérialisables

Toutes ces classes se trouvent dans le projet « ClientServer », dans le package « protocole.messages ».

### *Phase d'initialisation :*

- InfoClient : Demande de connexion de la part du client.
- InfoPlayer : Transmission des infos concernant le joueur.
- TiledMapMessage : Transmission du fichier de la carte aux clients.

### *En cours de partie :*

- Command : Transmission des commandes du client au serveur.
  - Mouvement : Commandes de mouvement du joueur.
  - Position : Transmission de la position du joueur.
  - Shoot : Commande de tir.
  - UseBonus : Utilisation d'un bonus.
- StateGame : Transmission du nouvel état du jeu.

## Le protocole, étape par étape

### *Initialisation :*

- Le serveur général (« ClientListener ») reste à l'écoute de nouveaux clients.
- Lorsqu'un client se connecte au serveur, ce dernier crée un serveur spécifique au client (« ClientHandler ») qui prendra en charge le client.
- Le client envoie ensuite l'objet « InfoClient » au serveur « ClientHandler ».
- Le serveur répond « OK » en cas d'acceptation du client ou « Refused » en cas de refus du client (s'il y a déjà 2 joueurs et que la partie est pleine par exemple).
- Si le serveur « ClientHandler » accepte le client, il lui envoie l'objet « InfoPlayer » contenant l'ID du joueur, son nom (imposé par le serveur, du type « Player 1 »), sa couleur, etc.

À ce moment, l'initialisation de la partie est terminée. Les clients attendent le signal « Start » de la part du serveur.

### *Début de la partie :*

C'est le serveur principal (« ClientListener ») qui s'occupe de synchroniser les « ClientHandler » pour que ceux-ci envoient le signal « Start » simultanément à tous les clients.

- Le « ClientListener » attend sur les « ClientHandler » que ceux-ci aient fini la partie d'initialisation avec leur client.
- Lorsqu'un « ClientHandler » a terminé la partie d'initialisation avec son client, il le signale au « ClientListener » et se bloque.
- Lorsque tous les « ClientHandler » sont prêts, le « ClientListener » libère les « ClientHandler ».
- Une fois libérés, les « ClientHandler » envoient le signal « Start » à leur client et la partie démarre.

### En cours de partie :

En cours de partie, la communication client-serveur se limite à l'envoi des commandes entrées par le client.

- En cours de partie, le client envoie les commandes entrées par le joueur au serveur (« ClientHandler »).
- Chaque « ClientHandler » reste à l'écoute des commandes envoyées par leur client pour les stocker dans une queue.
- Le « GameManager » (serveur gérant la partie) traite les commandes de chacun des « ClientHandler » en vérifiant leur validité et en les exécutant sur l'état du jeu puis retourne l'état du jeu mis à jour aux « ClientHandler ».
- Les « ClientHandler » envoient la mise à jour de l'état du jeu à leur client dès que celle-ci est disponible.

### Schéma de la communication client-serveur :

La Figure 7 présente la communication client-serveur dans son ensemble, telle que présentée plus haut. Les interactions sont représentées par des flèches entre les différentes classes. Les classes « ClientHandler » et « Client » peuvent êtreinstanciées plusieurs fois. Chacune de ces classes est exécutée dans un thread.

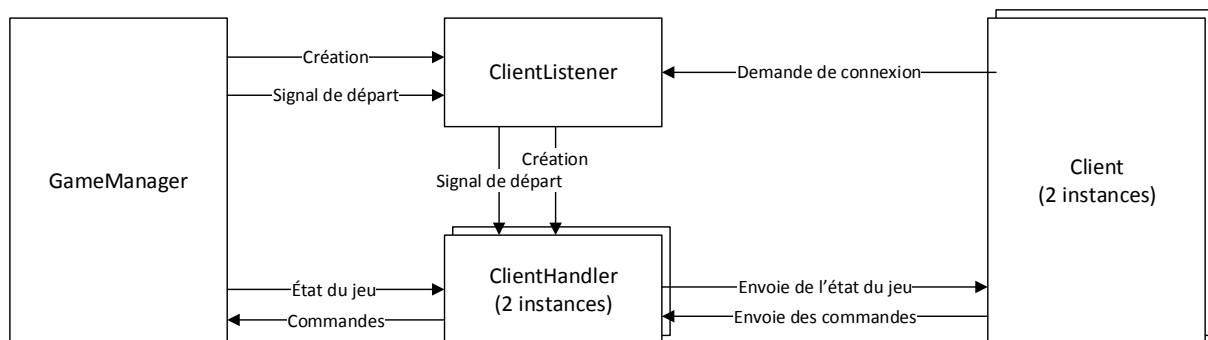


Figure 7

La Figure 8 représente la communication client-serveur en cours de partie seulement. La classe « ClientListener » n'est plus présente car elle n'influe plus sur la communication à ce stade.

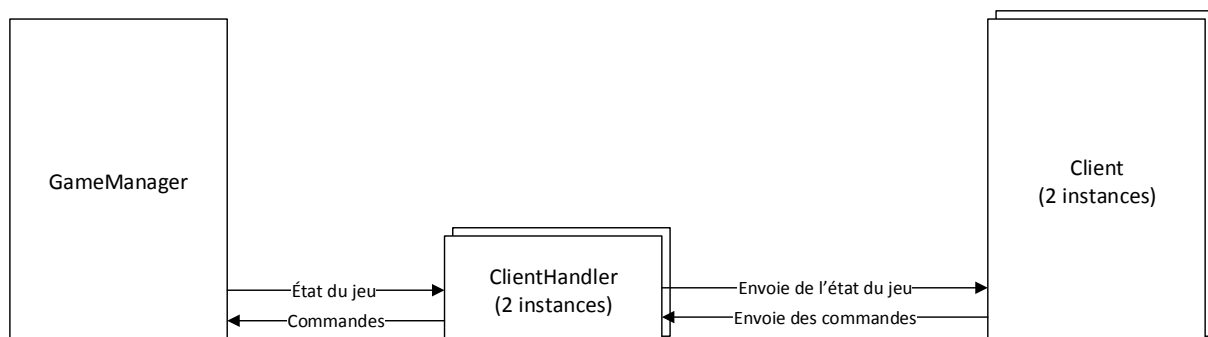


Figure 8

## Base de données

### Schéma de la base de données

La base de données est utilisée pour stocker les scores des parties effectuées. Dans notre application, nous avons choisis de stocker les scores de chaque partie dans un fichier XML. Le diagramme des entités utilisées est donné dans la Figure 9 ci-dessus.

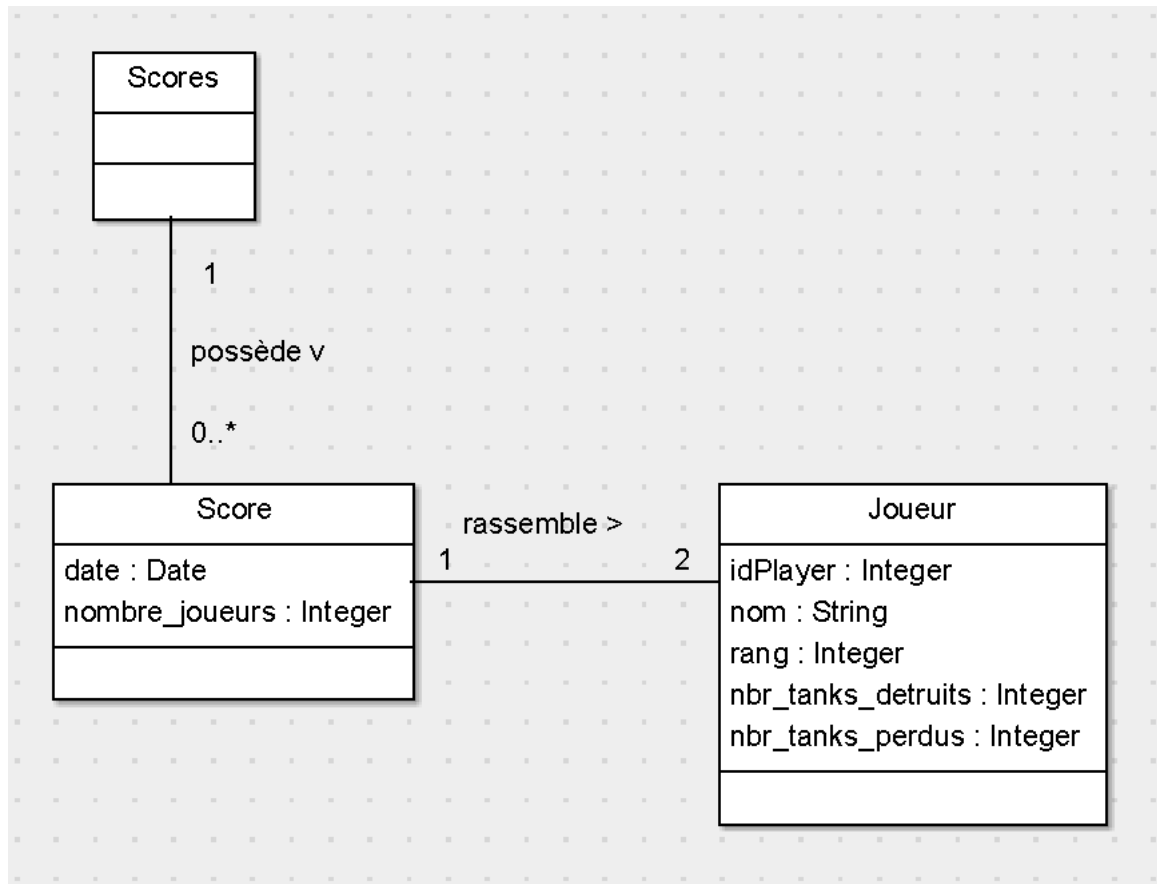


Figure 9

La table « **Score** » de la base de données représente le score d'une partie. Ainsi pour chaque partie, on génère un score avec les détails de chaque les joueurs de la partie. Les scores contiennent la date à laquelle la partie a eu lieu, du nombre de joueurs ayant participé à la partie et la liste des joueurs ayant joué la partie. Pour chaque « **Joueur** », on stocke son rang pour la partie, le nombre de tanks qu'il a détruits et le nombre de vies qu'il a perdu. Un joueur ayant joué plusieurs parties sera inscrit dans la table « **Score** » de chacune des parties auxquelles il a participé, c'est pourquoi il y a une cardinalité de 1 dans la relation liant « **Joueur** » à « **Score** ». Nous avons procédé ainsi car nous ne pensions pas nécessaire de complexifier le schéma pour regrouper chaque instance d'un même joueur. Chaque classe « **Score** » est stockée dans la classe principale du fichier XML « **Scores** ».

## Structure du fichier XML

La structure de notre fichier XML est de la forme suivante :

```
<scores>
  <score>
    <date_partie annee="2015" mois="septembre" jour="16" heure="16"
minute="54"/>
    <nombre_joueurs> 2 </nombre_joueurs>
    <joueur idPlayer="1">
      <nom>player1</nom>
      <rang> 1 </rang>
      <nbr_tanks_detruits> 3 </nbr_tanks_detruits>
      <nbr_tanks_perdus> 2 </nbr_tanks_perdus>
    </joueur>
    <joueur idPlayer="2">
      <nom>player2</nom>
      <rang> 2 </rang>
      <nbr_tanks_detruits> 2 </nbr_tanks_detruits>
      <nbr_tanks_perdus> 3 </nbr_tanks_perdus>
    </joueur>
  </score>
</scores>
```

Comme le montre ce fichier, tous les éléments de notre diagramme conceptuel ont été pris en charge. Le choix entre balise et attribut de balise a été dirigé par l'intuition et non pour une autre raison particulière puisque la seule chose qui va changer c'est la mise en place du parsing du fichier XML.

## Client

---

Notre implémentation possède deux facettes : la partie du client et la partie du serveur. Pour le client nous avons utilisé le moteur graphique 2D nommée Skick2D ainsi qu'une foison d'images, de tuiles et de sprites car la charge de travail pour la création d'image par notre main était trop conséquente. Les différentes images ainsi que leurs sources et décrite plus bas dans la documentation.

## Architecture

Voici la liste des classes du client, le diagramme UML du client est présenté plus bas. Il n'est représenté ici que la partie sans l'infrastructure réseau.

### List des classes

#### Game

Il s'agit de la classe principale qui va lier tous les éléments du jeu. Sa tâche est principalement de créer les objets et de les initialiser. Elle gère aussi toute la partie collision, la pression de touche, et le mouvement de la caméra.

Elle s'occupe de créer et d'initialiser les autres classes :

- Au travers de la factory :
  - Le joueur
  - Les ennemies du joueur
  - Les bonus
- Les scores
- La map
- La HUD du joueur
- L'affichage de fin
  - Victoire
  - Défaite

#### End

Cette classe est la super-classe des deux sous-classes qui sont appelé en fin de partie. Il s'agit d'une classe relativement simple ayant pour but d'afficher l'écran de défait ou de victoire. Et si le temps l'avait permis, l'affichage des scores. Ses sous-classes sont :

- Victory
- Defeat

#### HUD

Cette classe est l'observateur de la classe Player, cette dernière est notifiée quand le joueur passe sur un bonus afin de mettre à jour la barre d'action du joueur. Elle aurait aussi dû afficher les points de vie du joueur, mais faute de temps cette possibilité a été abandonnée au profit d'autres tâches.

## **Factory**

Cette classe est utilisée par la classe Game, elle est chargée de créer les objets de jeux comme les joueurs, les ennemis et les bonus.

## **Bonus**

Cette classe permet la création de bonus sur la map. Une instance de cet objet est créée quand on l'ajout dans la map. Quand un joueur roule sur le bonus une instance de cet objet est également créée dans la classe Player.

## **IFightable**

L'interface IFightable est implémentée par les classes Player et Enemy, ils utilisent tous les deux une classe Score qui contient leurs score respectif. Globalement ces deux classes sont pareils à la différence près que Player est observé par la HUD crée par Game et que le joueur est représenté en bleu et Enemy en rouge. Ce deux classes créent un objet Explosion a leurs mort.

## **Bullet**

La classe bullet est la super-classe des trois différents types de munitions/projectiles listés ci-dessous :

- L'Alpha Strick
- La Mine
- Le Laser

Ces projectiles sont décrits plus loin dans la section dédiée aux bonus.

## **Explosion**

Il s'agit d'une classe relativement simple, elle n'a pour tâche que de gérer l'animation des explosions qui est appelée quand un joueur meurt ou quand le bonus Alpha Strick est lancé.

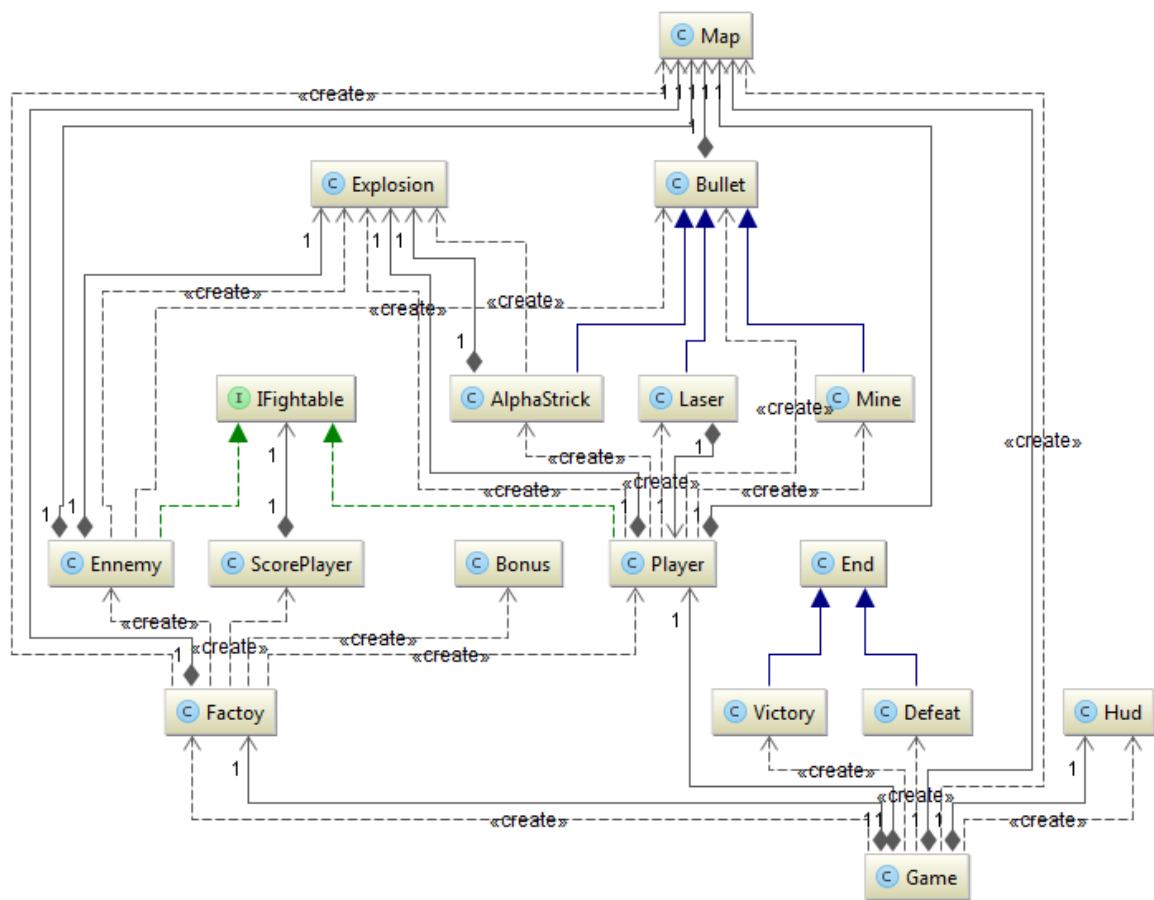
## **Map**

Cette classe gère la génération de la map avec les collisions entre les bords de maps et les objets obstacles.

## **Destroyable**

Cette classe devait implémenter les objets destructibles, mais pour cause de manque de temps et de compétences cette classe a été inachevée.

## Diagramme UML



© **destroyable**

Figure 10

## Début de jeu

---

Idéalement, avant le début de la partie le joueur aurait dû avoir la possibilité de choisir son tank. L'interface a été réalisée sur Photoshop mais n'a pas été intégrée, faute de temps. Il était prévu un bloc de couleur par tank, dans le cas présent il s'agit du tank bleu.



## En jeu

---

Lorsqu'un nombre suffisant de joueurs ont rejoint le serveur, la partie peut commencer. La fenêtre de jeu dispose des éléments suivants :

- Le tank du joueur en bleu
- Le ou les tank(s) ennemi(s) en rouge
  - En noir si détruit
- La HUD, composée de :
  - Un bloc en haut à gauche
  - Une barre de bonus
- Les bonus disponibles sur le sol
- Les projectiles (décrits plus loin)
  - Projectile simple
  - Mine
  - Laser
  - Attaque Alpha Strick



La Figure 11 présente un aperçu de l'interface du jeu en cours de partie avec les explications des différents éléments présentés ci-dessus.



Figure 11

## Fin de la partie

Lorsque la partie se termine, un affichage de fin apparaît informant de la victoire ou de la défaite du joueur et le jeu se bloque. Idéalement nous aurions dû afficher le score des joueurs et avoir la possibilité de relancer la partie en cliquant sur le bouton « Play ». Une illustration est présentée par la Figure 12.



Figure 12

Les images « Play », « Defeat » et « Victory » sont présentées dans les Figure 13, Figure 14 et Figure 15. Le design de ces trois images de fin a été réalisé par les soins de Simon sur le logiciel Photoshop CS6.



Figure 13



Figure 14




Figure 15



## Bonus

Dans notre jeu, le joueur peut ramasser des objets disposés sur le sol. Il peut ensuite les utiliser jusqu'à leur épuisement, chaque bonus ayant un usage limité en nombre d'utilisations ou en temps. Voici un listing des différents bonus.


### Bonus speed

- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Augmente la vitesse de déplacement du tank pour une durée de 10 secondes.



### Mine

- Icône du bonus : 
- Quantité : 3
- Capacité offert : Dépose une mine sur la position courante, représentée par l'image suivante : 


### Ultra Rapide Fire (non implémenté)

- Icône du bonus : 
- Quantité 3
- Capacité offert : Permet de tirer plus rapidement. De base, le tir simple ne doit pas être enchainé.

### Laser

- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Tirer un laser d'une taille de 32x200 pixels devant le joueur, blessant tous les joueurs pris dans le laser. Le laser est représenté par l'image suivante : 


### Foudroisement (non implémenté)

- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Foudroie une zone d'un rayon de 32 pixels autour du joueur toutes les secondes durant 3 secondes, blessant les joueurs dans la zone.


### Volée de lame

- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Chaque tir part dans les 4 directions à la fois.

## Bélier (non implémenté)

- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Fixe des piques sur l'avant du tank. Si le joueur entre en collision avec l'avant de son tank dans un adversaire, le tank adverse subit un point de dégât. Ce bonus est limité à 10 secondes une fois le bonus activé.

## Alpha Strick


- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Effectue une attaque sur une zone de 64x64 pixels devant le tank. Les joueurs qui se situent dans la zone cible prennent un point de dégât. Une animation est déclenchée lors de l'impact.



## Soin

- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Ajoute un point de vie au joueur.

## La Mort

- Icône du bonus : 
- Quantité : 3
- Capacité offerte : Inflige un point de dégât à un joueur choisi au hasard. La cible de ce bonus peut être le joueur l'ayant utilisé.

## Fenêtres

---

### Liste des fenêtres du client

- Fenêtre du launcher :
  - Rejoindre une partie
  - Quitter
- Fenêtre de saisie de l'adresse IP du serveur :
  - Champ de saisie de l'adresse IP du serveur

### Interface du serveur

Le serveur s'utilise via une interface en ligne de commandes.

## Implémentation du projet

---

### Technologies utilisées

#### Tiled

Tiled est un logiciel de conception de carte en 2 dimensions. Bien que ce ne soit pas une technologie en soit, il est tout de même important de citer ce logiciel qui nous a permis de réaliser nos maps du jeu facilement et rapidement.

#### Slick2D

Pour la réalisation de ce projet nous avons utilisé le moteur graphique Slick2D qui permet facilement d'intégrer des maps faites à partir du logiciel Tiled. La principale difficulté avec Slick2D fût le fait que nous ne connaissions pas ce moteur graphique et que pour un problème donné il y a une foison de solutions différentes. Nous avons donc dû naviguer entre les différentes solutions trouvées. La principale erreur commise avec cette technologie fut que je (Simon) suis parti d'un tutoriel que j'ai suivi et j'ai patché le jeu par la suite ce qui a produit du code mal organisé et difficilement compréhensible.

## Gestion du projet

---

### Rôles des participants

**Simon Baehler**

#### *Graphisme*

Je me suis occupé de réaliser la HUD de notre jeu (l'interface en jeu) comme les barres de vie de joueurs, la barre d'action et les affichages de fin de jeu.

#### *Game design*

Je me suis occupé de chercher des images pour les jeux qui m'auraient pris trop de temps à faire moi-même. J'ai également imaginé les bonus des jeux et fait quelque maps.

#### *Programmation*

Programmation de la partie client, j'ai beaucoup cherché d'information sur Slick2D (toute la partie client).

### Suivi du projet

#### **Bilans des itérations**

Les bilans des itérations ainsi que les bilans personnels sont disponibles en annexe.

## Stratégie de test

---

### Client

Durant le développement de la partie client, des tests ont été réalisés une fois chaque fonctionnalité implantée. Ces tests ont été réalisés sur le client hors-réseau.

Test	Ce qui doit se passer	Resultats /Remarques
Chargement d'une map	La map doit apparaitre dans la fenêtre de jeux.	OK
Spawn du personnage	Le personnage doit apparaitre sur la map, dans un premier temps avec des coordonnées fixes puis aléatoires.	OK
Déplacement du personnage	Le personnage doit pouvoir se déplacer vers le haut, le bas, la gauche et la droite, sans prendre en compte les collisions avec les éléments.	OK
Caméra	La caméra suit le personnage s'il s'approche trop du bord de la fenêtre.	OK
Collinson avec les obstacles	Le joueur peut entrer en collision avec un élément.	La collision se faisant dans un premier temps trop tard, puis trop tôt. Désormais elle s'effectue au bon moment mais il arrive qu'on reste bloqué.
Collision avec les bords de la map	Le joueur ne peut pas sortir de la map.	OK
Musique	Une musique se lance au début du jeu et boucle indéfiniment.	OK
Soud Effect	Quand nous tirons un effet sonore se lance qui fait « boom ».	OK
Affichage des HUD	Les hud s'affichent correctement à savoir les points de vie des tanks ennemis, la barre d'action et l'avatar du joueur (bloc en haut à gauche).	Pas de point de vie pour le joueur -> manque de temps.
Spawn de bonus	Les bonus apparaissent aléatoirement sur la map.	OK
Ramassages de bonus	Quand un joueur passe sur un bonus il le ramasse et l'ajoute dans sa barre d'action si elle n'est pas pleine.	OK, les collisions ne sont pas très précises (2 ou 3 pixels de décalage).
Tir	Quand un joueur appuie sur « espace » il tire un projectile.	OK
Utilisation bonus Vitesse	Quand un joueur utilise le bonus « vitesse » sa vitesse augmente correctement, puis au bout de 10sec, sa vitesse de base est restaurée.	OK

Utilisation bonus Mine	Le joueur peut bien déposer une mine derrière lui.	OK
Utilisation bonus URF	Le joueur tire plus rapidement. (3 tirs par seconde au lieu d'un seul). Au bout de 10 secs, le bonus cesse.	NON IMPLEMENTÉ
Utilisation bonus Laser	Créer un laser d'une taille de 32x200 pixels devant le joueur blessant tous les joueurs pris dans le laser.	Il arrive que le laser touche un joueur qui ne se situe pas dans le champ d'action du laser.
Utilisation bonus Foudroiement	Foudroie une zone d'un rayon de 32 pixels autour du joueur toutes les secondes durant 3 secondes, blessant les joueurs dans la zone.	NON IMPLEMENTÉ
Utilisation bonus Volée de lame	Tire un projectile dans les quatre directions.	OK
Utilisation bonus Bélier	Fixe des piques sur l'avant du tank. Si le joueur entre en collision avec l'avant de son tank dans un adversaire, le tank adverse subit un point de dégât. Ce bonus est limité à 10 secondes une fois le bonus activé.	NON IMPLEMENTÉ
Utilisation bonus AlphaStick	Effectue une attaque sur une zone de 64x64 pixels devant le tank, les joueurs dans la zone prennent un point de dégât. Une animation est déclenchée à l'impact.	OK, peut-être un peu trop puissant.
Utilisation bonus Soins	Ajoute un point de vie au joueur.	OK
Utilisation bonus Mort	Inflige un point de dégât à un joueur choisi au hasard. La cible de ce bonus peut être le joueur l'ayant utilisé.	OK
Décrémenter le nombre de munition	Quand nous utilisons X fois un bonus il n'est plus disponible.	OK, par contre il n'y a aucune information sur le nombre de munition, il aurait aussi été bien de pouvoir régler le nombre de munition reçus lors du ramassage.
Animation	Quand un joueur meurt ou que l'Alpha Strick est lancé une animation d'explosion se lance.	OK
Fin de la partie	Quand le joueur meurt ou que tous les ennemis sont morts un écran de fin de partie apparaît.	OK



## **Stratégie d'intégration du code de chaque participant**

---

Nous avons utilisé Git avec un repos sur le site GitHub pour le partage de code et un groupe WhatsApp pour une communication plus rapide entre les membres du groupe.

## **Etat des lieux**

---

### **Côté serveur**

Initialement, la configuration du serveur devait permettre :

- Choix du nombre de joueurs
- Choix de la carte
- Choix du temps de la partie

Finalement, certains paramètres ont été fixés et ne pourront donc pas être modifiés :

- Le nombre de joueur est fixe. Il est néanmoins possible de le changer directement dans le code, avec quelques ajustements.
- La carte ne peut pas être choisie. Il est néanmoins possible de changer ça directement dans le code.
- Il n'y a pas de temps limite de jeu. Le timer n'a pas encore été implémenté.

### **Côté client**

Initialement, le client devait gérer les fonctionnalités suivantes :

- Le choix du serveur sur lequel se connecter. Il devait être également possible de se connecter en tant que joueur ou en tant qu'observateur de la partie.
- Le joueur devait pouvoir choisir son tank au début de partie. Chaque tank devait posséder des caractéristiques différentes telles qu'un nombre de points de vie plus ou moins grand, une vitesse plus ou moins grande.
- Le joueur devait également pouvoir choisir la couleur de son tank.
- Durant la partie, il devait être possible de :
  - Tirer
  - Se déplacer
  - Ramasser des bonus
  - Utiliser les bonus
- A la fin de la partie, il devait être possible de :
  - Consulter les scores (automatique)
  - Relancer la partie

Finalement, ce sont les fonctionnalités suivantes qui ont été implémentées :

- On ne peut que se connecter au serveur en tant que joueur. Le mode observateur a été abandonné.
- Il n'y a qu'un seul modèle de tank disponible, la couleur est imposée.
- Durant la partie, il est possible de :
  - Tirer
  - Se déplacer
  - Ramasser des bonus
  - Utiliser les bonus
- A la fin de la partie, les scores ne s'affichent pas automatiquement.
- La partie ne peut pas être relancée.

## **Travail restant**

---

### **Côté client**

- Début de la partie : Ajoute la possibilité de choisir son tank, soit par la pression de touche soit par clique. Ayant trouvé un tutoriel sur la réalisation d'un écran de démarrage cela devrait prendre environ 3 heures de travail.
- En cours de partie : Réaliser les bonus non implémentés comme le Bélier, le Foudroiement et le URF. Le foudroiement et le Bélier semblent assez complexe à implémenter, ce qui n'est pas le cas du URF. Un temps de travail de 5 heures pour ces 3 bonus devrait jouer.
- Fin de partie : L'affichage des scores en fin de partie devrait être fait rapidement car une structure est déjà existante, il ne reste plus qu'à réaliser qui a touché qui et l'afficher. 3 heures devraient suffire pour cette partie. Pour ce qui en est du relancement d'une partie il s'agit ici de la grande inconnue. En effet, des recherches sur l'implémentation d'une zone cliquable ont été effectuées, malheureusement sans succès.

## Autocritiques

---

### **Simon Baehler**

Comme dit plus loin dans la conclusion, pour moi ce projet est un fiasco. Mon erreur principale fut de partir coder la partie client sans que la partie serveur ne suive, il y a eu un décalage entre les deux parties et le merge trop dure. Mon erreur fut également d'avoir « sur couché » le code réalisé durant le tutoriel au lieu de, une fois l'outil Slick2D connu, refaire un projet depuis 0 avec une implémentation réfléchié et basé sur des cas d'utilisation, diagramme d'activité etc...

### **Armand Delessert**

Le code produit pour la partie serveur et la partie communication réseau reste relativement propre, la structure des classes a été reprise d'un ancien projet de jeu en réseau. Il pourrait très bien être réutilisé dans un futur projet moyennant quelques modifications et corrections de bugs.

### **Ngueukam Djeuda Wilfried Karel**

Le projet ne m'a pas été d'un grand apport car je dois admettre que j'y ai pas beaucoup participé. D'abord le projet ne m'a pas vraiment emballé mais il fallait faire un groupe et aussi toute l'idée n'a pas été discutée avant la mise en place du projet et dont j'étais pas trop au courant de où on allait au début. Aussi avec des difficultés personnelles j'ai pas vraiment voulu m'investir dans le projet car je pensais qu'il était réalisable dans les temps et que avec ma pleine participation ou pas il pourrait être terminé. Une fois que les retards ont commencé à se faire sentir j'ai pas su aider les membres de mon équipe car c'était difficile de comprendre ce qui était déjà fait vu que l'architecture de l'application me semblait pas idéale; mais ça je ne peux m'en prendre qu'à moi car je ne me suis pas investi dans le projet dès le départ. Mais je pense que ce qui a cruellement manqué dans l'équipe c'est la communication au début du projet et le manque de coordination.

### **Benoit Zuckschwerdt**

De mon point de vue, ce projet a mal commencé pour réussir une implémentation de communication client-serveur, avec un serveur gérant toutes les collisions et les mouvements, car les gestions de collisions et de mouvements ont été implémentées côté client, de plus au départ le serveur n'était pas une application à part. Ceci a vraiment freiné le développement de la partie serveur et réseau et il s'est avéré très difficile de rattraper cette erreur en si peu de temps.

Si c'était à refaire, il faudrait mieux se planifier au départ, pour ne pas partir sur une mauvaise voie.

## Conclusions personnelles

---

### **Simon Baehler**

Ce projet se résume à un mot : fiasco. Durant tout le long du projet il n'y a eu aucune communication, dès les 1<sup>ères</sup> itérations la partie réseau a pris du retard. Le rendu des bilans d'itération personnels toujours en retard des fois même de plusieurs semaines, et qui plus pour la plupart du temps minimaliste. Un investissement faible de membre du groupe et un partage totalement inégal du travail et un laxisme de plus énorme.

### **Armand Delessert**

Je ne considère pas ce projet comme un échec complet. Certes, il ne nous a pas été possible de livrer un produit fini qui corresponde aux objectifs de départ dans les temps mais le travail réalisé est tout de même remarquable. Je pense que si je dis que ce projet était très, voir même un peu trop ambitieux par rapport au temps à disposition, je ne vais pas à l'encontre de la pensée de mes collègues. J'avais personnellement assez de temps à consacrer à ce projet en parallèle des autres cours mais je suis conscient que ce n'est pas le cas de tous mes collègues. Malgré l'engouement certains des membres de l'équipe lors du début du projet, la charge de travail était trop importante. Nous avons bien essayé de réduire cette charge en limitant les fonctionnalités du jeu mais sans succès.

L'un des problèmes était le manque de vision sur le développement du projet à long terme. J'avais parfois l'impression d'avancer dans le brouillard, une impression sûrement due à un manque d'expérience dans le développement de projets plus gros qu'un simple labo.

Ce projet m'aura surtout apporté de l'expérience dans le développement en Java et en programmation orientée objet ainsi qu'en programmation réseau. Car il s'agit là de la première fois que je m'attaque à une communication client-serveur complète. Bien que j'aie apprécié développer cette partie, ce n'était pas une tâche facile tant en terme de conception et développement qu'en terme de debug. Malgré tout j'ai trouvé ce challenge très intéressant et très enrichissant. Je suis certes déçu de ne pas avoir pu continuer cette partie jusqu'au bout mais je suis tout de même fier du travail réalisé.

## **Ngueukam Djeuda Wilfried Karel**

D'un point de vue objectif le projet n'a pas été vraiment bénéfique pour moi car j'ai pas appris grand-chose par rapport à l'année dernière si ce n'est le parsing des fichiers XML à l'aide de Jdom2 et de SaxParsers. Mais il m'a surtout permis de me rendre compte des conditions à établir pour mener à bien un projet notamment :

- La communication et la coordination
- La mise en place en commun de l'architecture de base de l'application et de la planification des tâches
- La prise en compte des capacités de chacun des membres du groupe
- L'évaluation minutieuse des fonctionnalités et du temps à disposition avant de proposer une fonctionnalité dans son application

## **Benoit Zuckschwerdt**

Malgré le fait que nous n'avons pas pu mener à bien notre projet, le produit final que l'on rend possède déjà beaucoup de fonctionnalités.

Ce projet m'a permis d'acquérir des connaissances en programmation réseau Java, ainsi qu'une première approche de Slick2D. Il était également enrichissant pour nos futurs travaux en groupe.

Avec ce projet, nous avons été ambitieux vis-à-vis de nos connaissances et du temps à notre disposition, mais je ne le considère pas comme un échec complet, car le point essentiel de ce "laboratoire" projet étant la gestion en soit du projet. Et de ce côté nous retiendrons tous l'importance d'une communication et une coordination parfaite entre les membres, que les spécifications (réseau, applications, configurations) doivent être établis au début de manière à ne plus les toucher si possibles.

## Annexes

---

Bilans des itérations et bilans personnels

Diagramme de classes du projet « WarTanks »

Diagramme de classes du projet « ClientServeur »

Diagramme de classes du projet « WarTanksGUI »

Fichier XML de la base de données

Code source du projet « WarTanks » (serveur)

Code source du projet « WarTanksGUI » (client graphique)

Code source du projet « ClientServeur » (protocole de communication)

## Table des illustrations

Figure 1	6
Figure 2	8
Figure 3	12
Figure 4	13
Figure 5	14
Figure 6	15
Figure 7	17
Figure 8	17
Figure 9	18
Figure 10	22
Figure 11	24
Figure 12	25
Figure 13	25
Figure 14	25
Figure 15	25