Sapotache

Projet de programmation C

Calandra Joséphine
Parpaite Thibault
Poncet Clémence
Sarrabayrouse Alexis

Encadrant : M. Desbarats Pascal

ENSEIRB-MATMECA
Filière informatique
Semestre 6
12 mai 2017

Table des matières

Introduction

Contexte

Ce document a pour but de décrire le déroulement de notre projet de programmation C réalisé au cours de notre semestre 6 de cursus informatique à l'ENSEIRB-MATMECA. L'objectif de ce dernier est d'implémenter un jeu de société, **le saboteur**, avec un paradigme client/serveur. Le deuxième point crucial est la mise en place d'une intelligence artificielle capable de jouer au jeu en adoptant différents types de stratégies.

Plan

Dans un premier temps, nous étudierons le **domaine** du projet en explicitant de manière formelle les règles du jeu tout en réalisant une analyse approfondie de l'existant.

Ensuite nous nous intéresserons à la phase de **conception** en rédigeant le cahier des charges décrivant de manière détaillée les besoins et les limites du projet. Cette partie détaillera également l'architecture du code.

Après cela, nous présenterons plus en détails le **logiciel** ainsi que les choix d'**implémentation** pris tout au long du développement.

Enfin nous allons mettre en évidence la véracité et performance de notre travail à l'aide de l'ensemble des jeux de **tests** mis en place.

Méthodes de travail

Nous avons mis en place l'outil CMake pour permettre une compilation dynamique, portable et simplifiée de notre projet.

Est aussi jointe à ce dernier, une documentation au format HTML Doxygen de l'ensemble des fonctions, macros et autres structures de données implémentées tout au long du développement.

Pour la gestion des versions, nous avons utilisé un dépôt *SVN* sur la forge de l'ENSEIRB-MATMECA. Les éditeurs de texte que nous avons privilégiés sont *Emacs* et *Atom*.

Pour suivre l'avancée du projet de manière ergonomique, nous avons mis en place un tableau Trello visible sur la figure . Cela a permis d'évaluer de manière efficace la charge de travail, répartir les tâches et tenir les délais. Le rapport a été rédigé à l'aide de l'éditeur en ligne collaboratif ShareLatex.

Enfin, nous avons choisi de réaliser ce projet en suivant la méthodologie *SCRUM*. En réusmé, *SCRUM* est un processus agile qui procède par itération successives en se focalisant dans un premier temps sur les fonctionnalités essentielles. L'objectif est qu'à la fin de chaque phase de *sprint* (environ deux semaines) un produit **fonctionnel** soit livré. Le livrable implémente seulement une partie des besoins définis dans la cahier des charges (choisis par ordre de priorité), les autres fonctionnalités étant traitées durant des phases de sprint ultérieures.

Par exemple, pour le serveur, nous avons procédé de la manière suivante, en sachant qu'à la fin de chaque phase le serveur était opérationnel :

— Création d'un squelette de serveur initialisant simplement les différents clients
— Création d'une board et initialisation de celle-ci (informations du *parsing*)
— Distribution des cartes
— Système anti-triche, véracité des coups joués
— Gestion des manches et distribution des pépites
— Factorisation du code avec le client

Dans cette optique, nous avons fonctionné en pair programming (programmation en tandem), avec une équipe focalisée sur le serveur et l'autre sur le client afin d'avoir une interaction client/serveur opérationnelle à la fin de chaque séance.

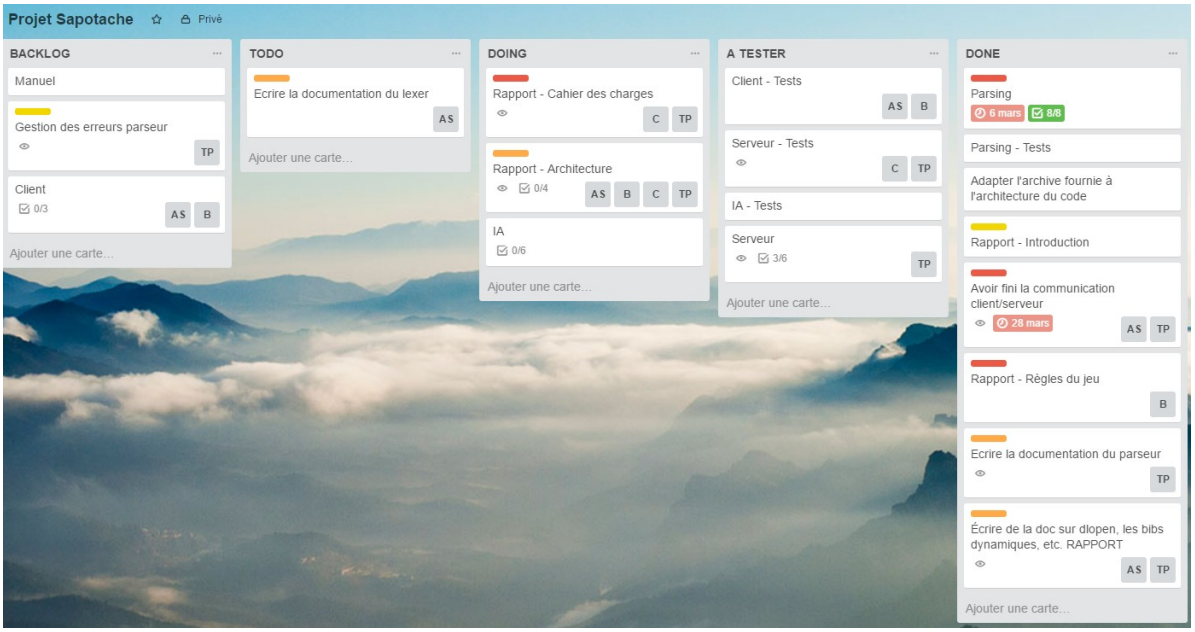


FIGURE 1 – Tableau Trello Sapotache

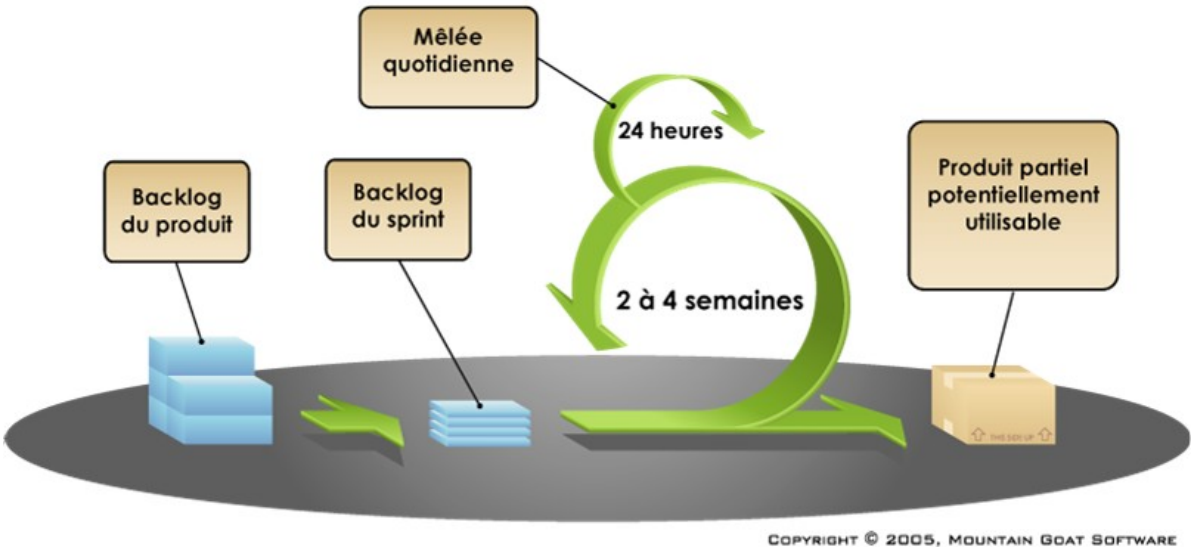


FIGURE 2 – SCRUM - Méthode de gestion de projet Agile

1 Saboteur - Règles du Jeu

Le Saboteur est un jeu de plateau se jouant de 3 à 10 joueurs incarnant des nains. Il existe deux types de nains : les chercheurs d'or, qui ont pour objectif de se frayer un chemin jusqu'au trésor, ainsi que les saboteurs, qui cherchent à empêcher les chercheurs d'or à accéder au trésor. Si les chercheurs d'or trouvent le trésor, ils se partagent les pépites du trésor et les saboteurs ne récupèrent rien. A l'inverse, si les chercheurs d'or ne trouvent rien, les saboteurs se partagent le butin. Les rôles de chaque joueurs sont camouflés jusqu'au partage des pépites d'or.

Au cours de la partie, chaque joueur joue de la manière suivante : soit il pose une carte chemin sur le labyrinthe, soit il pose une carte position permettant de casser ou réparer les outils des autres nains, soit il se défausse d'une carte. Il tire ensuite une nouvelle carte. Le jeu se joue en trois manches au bout desquelles le joueur possédant le plus de pépites gagne la partie.

⚠ Pour jouer une carte chemin, il faut la poser à côté d'une autre carte chemin. De plus, tous les chemins doivent se connecter sur la carte adjacente et les cartes chemin doivent toujours être posées dans le même sens.

FIGURE 1.1 – Le jeu de société Saboteur

2 Analyse de l'existant

Il existe déjà plusieurs implémentations du jeu de société saboteur. L'une d'entre elle est disponible sur le site de jeux de plateaux *boardarena* [1], il s'agit d'une implémentation en JavaScript [2].

Une version mobile a également été développée en JavaScript [3].

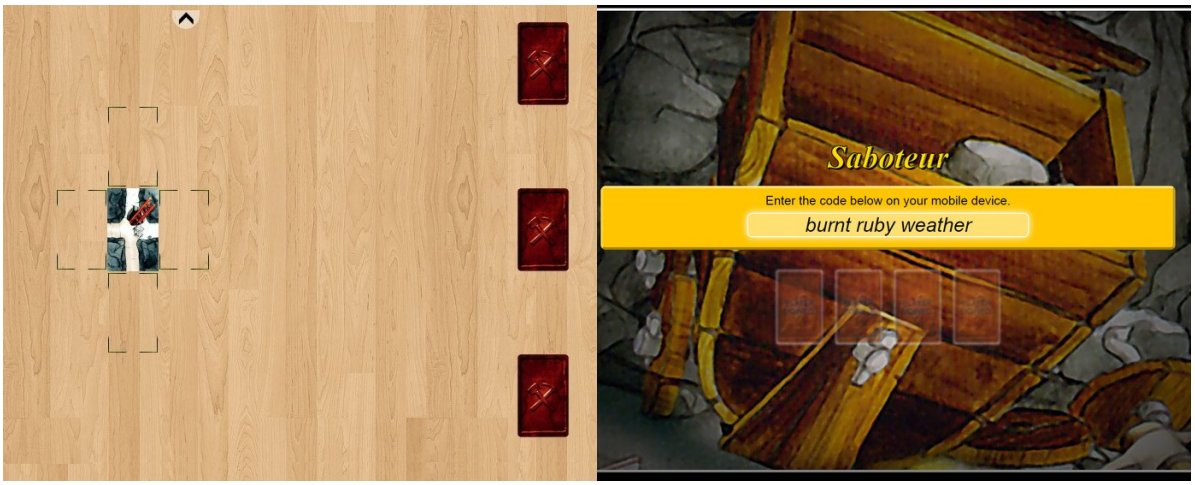


FIGURE 2.1 – Version web (gauche) et version mobile (droite)

1. https://fr.boardgamearena.com/#!gamepanel?game=saboteur
2. https://github.com/smirea/Saboteur
3. https://github.com/annsonn/saboteur

3 Cahier des charges

3.1 Besoins fonctionnels

L'objectif du projet consiste à implémenter un ensemble de fonctions permettant de faire jouer un nombre quelconque de joueurs à une partie de Sapotache. L'application doit se présenter en adoptant un paradigme client/serveur et intégrer les fonctionnalités suivantes :

Le serveur de jeu doit :
— Organiser une partie (initialisation).
— Faire jouer chaque client à tour de rôle en leur précisant la liste des coups joués précédemment (boucle principale).
— Enregistrer le coup du client à son tour (boucle principale).
— Notifier les clients de la fin de la partie (fin).

Chaque client doit :
— Gérer son propre ensemble de carte (lors de l'initialisation, puis à chaque tour).
— Mettre à jour l'état du jeu local grâce aux *previous moves*.
— Jouer selon ses objectifs propres (chercheur d'or ou saboteur).

3.2 Besoins non fonctionnels

Les fonctionnalités décrites dans la première partie du cahier des charges doivent de plus respecter les spécificités suivantes :

— Les différents clients doivent être **interopérables** entre équipes de projets. À cet effet, il est demandé que chaque client soit compilé sous la forme d'une bibliothèque partagée (*SHARED*) et chargeable de manière dynamique, en utilisant les fonctions de la famille *dlopen*, définies dans *dlfcn.h*.
— Il est demandé d'implémenter au moins **deux clients** différents en terme de stratégie utilisée.
— Il est demandé de faire attention à éviter la duplication du code. Pour cela de la **factorisation** de code sera réalisée si cela est nécessaire.

3.3 Contraintes et limites

Plusieurs contraintes nous ont été imposées pour la réalisation de ce projet.

Tout d'abord, la réalisation du parseur n'a pu être réalisée en utilisant uniquement des bibliothèques de la *libc*. L'utilisation des outils classiques d'analyse lexicale et syntaxique tels que flex et bison étaient prohibée. Cela nous a permis d'étudier plus en profondeur les mécanismes du *parsing*.

De plus, le fichier de configuration (qui est analysé par le parseur) servant à décrire le plateau de jeu et les cartes utilisées pour une partie doit suivre un format spécifique. Pour plus de détails, le format d'entrée est disponible en annexe 8.1 et le format de sortie associé (généré par le parseur) est visualisable en annexe 8.2.

Enfin, une interface nous a été fournie. Celle-ci est le **seul** moyen de communication qui sera utilisé pour dialoguer entre le serveur et le client tout au long du projet. Une version simplifiée de cette interface est disponible en annexe 8.3.

4 Architecture

4.1 Interopérabilité client/serveur

Un des points cruciaux dans le développement de ce projet a été **l'interopérabilité** entre le serveur et les clients. Pour cela nous avons fait en sorte de ne faire communiquer les différentes entités entre elles qu'à travers l'interface décrite en annexe 8.3, qui a été choisie suite à un consensus entre les différents groupes.

Afin de vérifier cette interopérabilité entre équipes de projets, nous avons échangé nos bibliothèques dynamiques à la fin de chaque séance de TD et lancé des parties sur tous les serveurs avec chacune d'entre elles, puis avons cherché à connaître les raisons des problèmes rencontrés, s'il y en avait, afin de corriger nos codes s'il y avait lieu de le faire.

4.2 Factorisation du code

Dans un contexte agile, le code a tout d'abord été dupliqué entre client et serveur afin d'obtenir rapidement un prototype fonctionnel. Puis une fois le projet abouti, une factorisation du code a été réalisée.

4.2.1 Etudes des structures dupliquées

Dans le texte ci-dessous, pour plus de clarté, nous considérerons que les structures du projet sont des classes.

Dans un premier temps nous avons réalisé deux diagrammes UML, un pour le coté serveur et un second pour la partie client de notre projet (cf : figure 4.1, figure 4.2). Pour des raisons de lisibilité nous avons choisi de ne pas forcément utiliser les type de C de notre code. A partir de ces derniers, des similitudes entre les classes ont pu être relevées :

- Les classes *board* sont identique, ainsi que *tile* qui compose la matrice représentant le plateau de jeu. Nous pouvons donc mettre en commun ces dernières.
- Les classes *game*, elles, n'ont que certain attributs en commun. Un héritage peut-être envisagé.
- La classe *player* est propre au serveur mais les attributs d'états (*axe_broker, lamp_broker* et *cart_broker*) sont semblables à ceux de la classe *state* du client. Ils pourront donc être stockés dans cette dernière.

Nous avons réalisé un diagramme UML (cf. figure 4.3) représentant les classes du client et du serveur ensemble, sans faire apparaître les éléments de l'interface fournie (*sapotache_interface*) pour plus de lisibilité.

4.2.2 Mise en place

Le langage C ne permettant pas faire d'héritage une solution s'en rapprochant a été implémentée. Un module "père" contenant une structure représentant la classe mère *game* ainsi qu'un champ d'un pointeur générique *extra_data*, la fonction permettant de libérer la mémoire de ce dernier et la structure *tile*.

Le champ *extra_data* permet au module "fils", représentant les classes filles, d'ajouter la structure de leur choix au module "père". Ensuite, chacun des module fils implémente son propre constructeur, c'est ce dernier qui sera utilisé par la suite.

4.3 Algorigramme

L'organigramme de programmation ci-dessous décrit le fonctionnement d'une partie de saboteur (initialisation, *main_loop* et fin). Les interactions entre le client et le serveurs (pointillés) sont réalisées seulement grâce à l'interface.

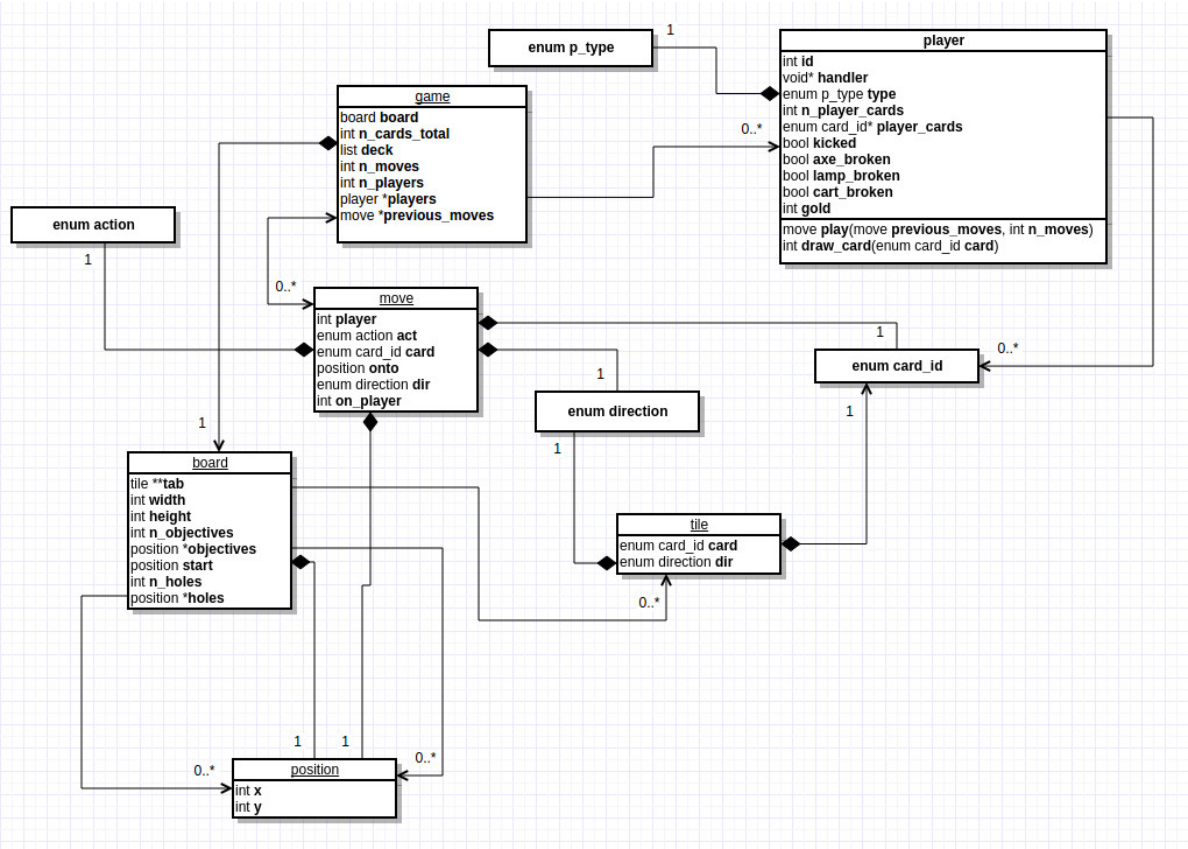


FIGURE 4.1 – Diagramme UML du serveur

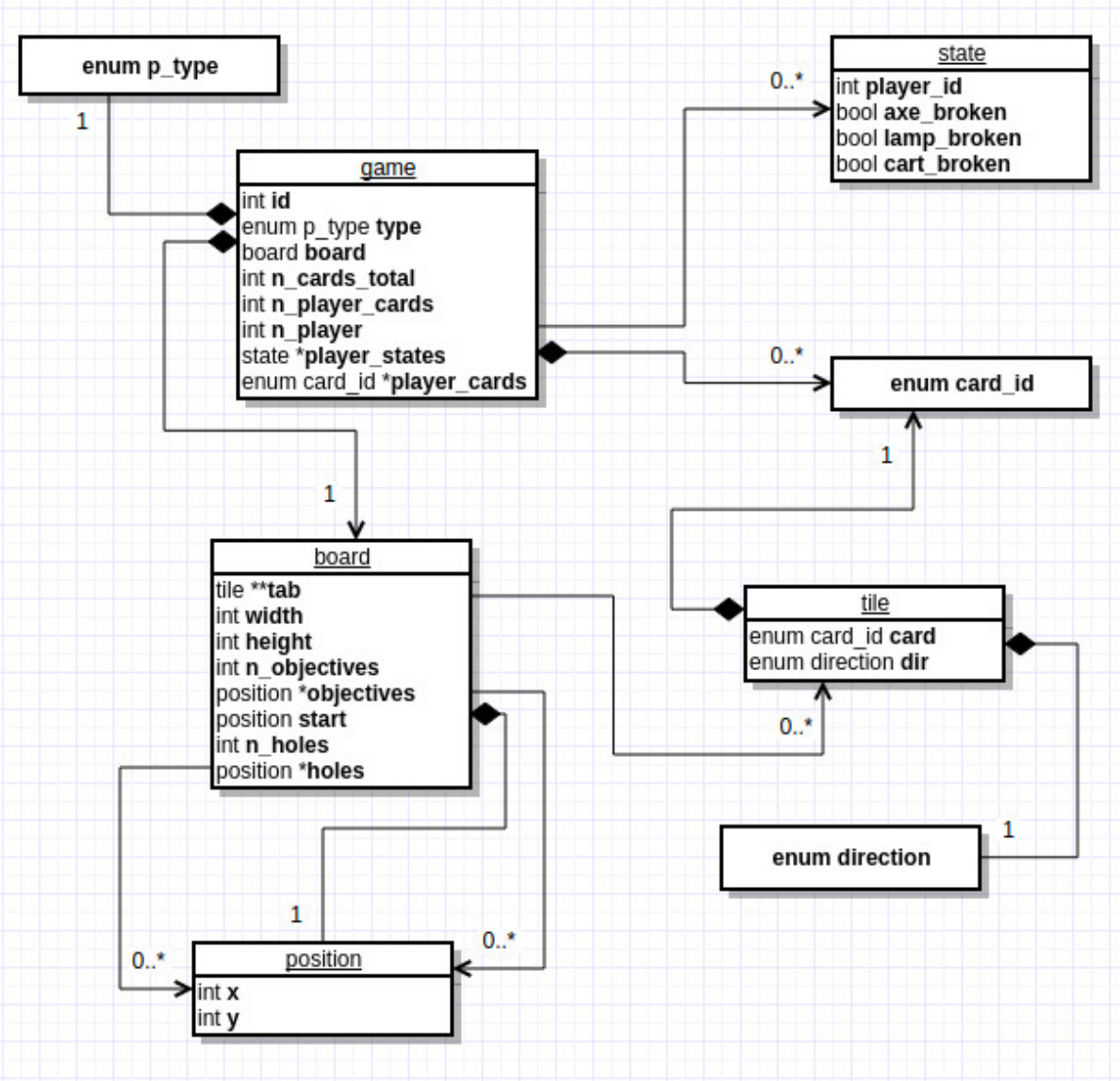


FIGURE 4.2 – Diagramme UML du client

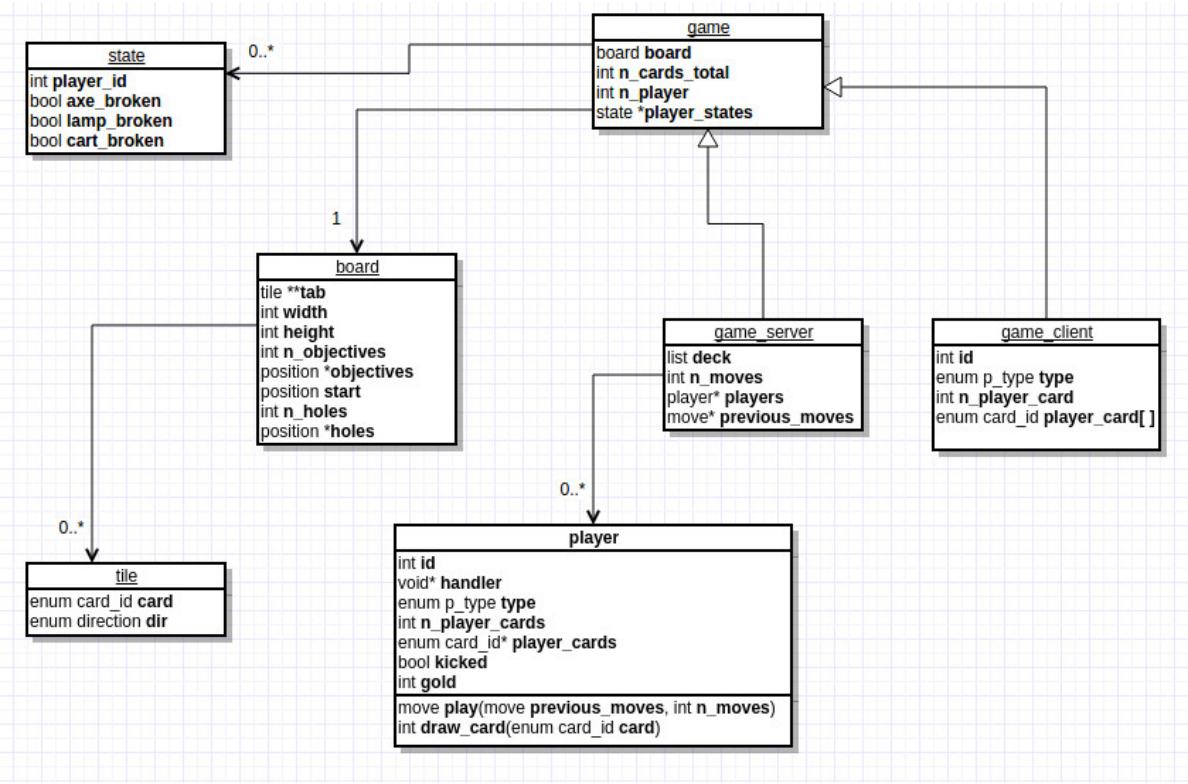
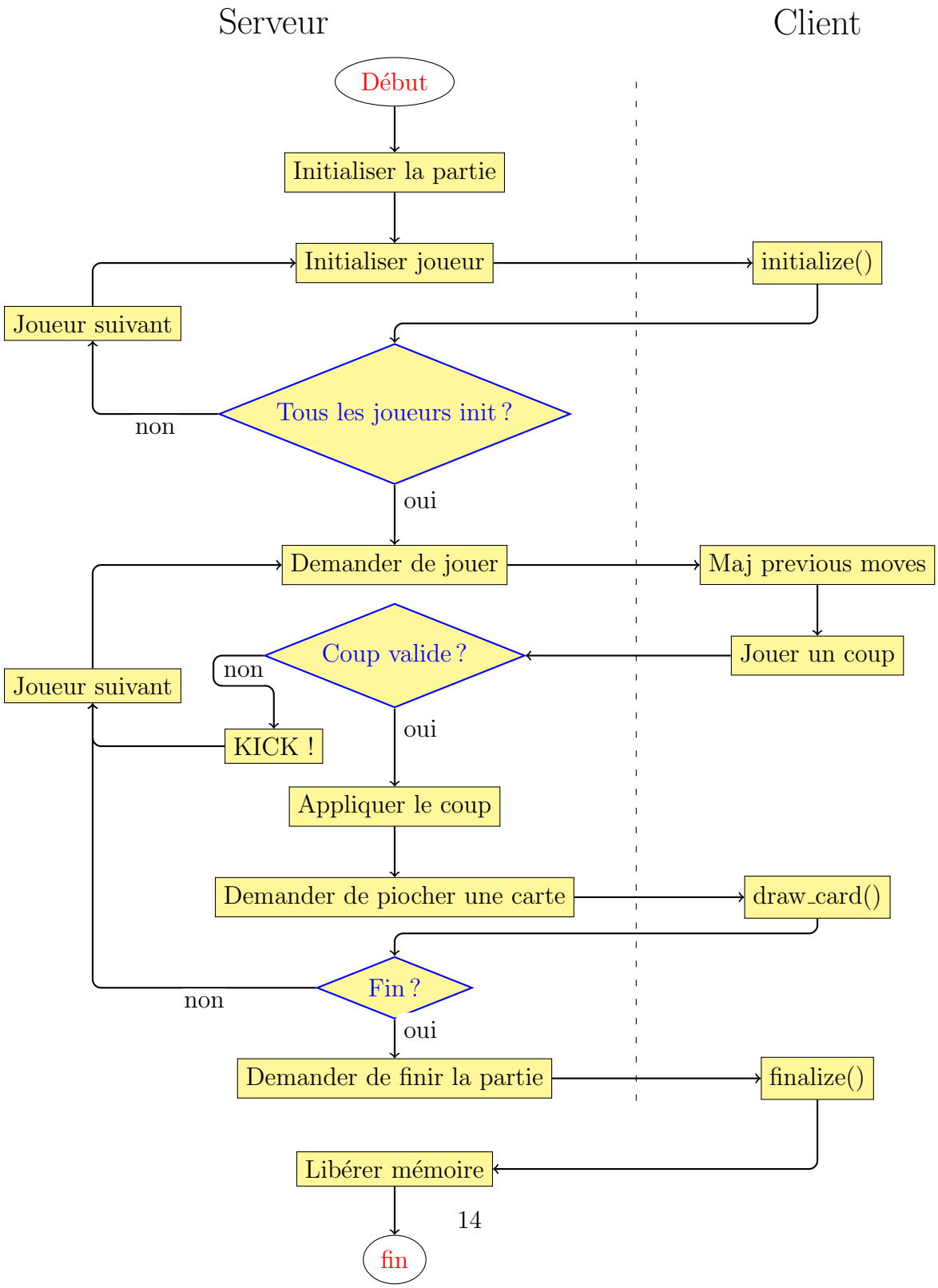


FIGURE 4.3 – Diagramme UML final


Serveur
Client
Début
Initialiser la partie
Initialiser joueur
initialize()
Joueur suivant
Tous les joueurs init ?
non
oui
Demander de jouer
Maj previous moves
Coup valide ?
Jouer un coup
Joueur suivant
non
KICK !
oui
Appliquer le coup
Demander de piocher une carte
draw_card()
Fin ?
non
oui
Demander de finir la partie
finalize()
Libérer mémoire
fin

5 Logiciel

Le projet utilise le moteur de production **CMake** pour générer le Makefile nécessaire à la compilation, à la génération de la documentation et aux tests. Pour utiliser ce dernier il suffit d'exécuter les commandes suivantes :

```
# Creer un dossier build, puis s'y rendre
mkdir build
cd build

# Executer CMake pour generer le Makefile
cmake ..

# Compiler
make

# Lancer les tests
make test

# Generer la documentaion
make doc
```

Pour lancer le programme il faut joindre un fichier de configuration et au minimum trois clients en bibliothèque dynamique (.so)

```
./sapotache config_file.text libclien_1.so libclien_2.so libclien_3.so ...
```

Par exemple dans le repertoire /build/bin :

```
./sapotache ../../src/tests/parser_examples/sapo1.ok.txt ../lib/libhuman.
    so ../lib/libhuman.so ../lib/libhuman.so
```

6 Algorithmes et structures de données

6.1 Parseur

Le parseur implémenté avait initialement pour objectif de générer un fichier de sortie à partir d'un fichier décrivant un plateau de jeu (dans un format spécifique). Dans un second temps, ce fichier de sortie a été remplacé par une structure *memory_t* récupérant l'ensemble des informations liées au plateau de jeu (position du départ, des objectifs, cartes utilisées, etc.). Cette structure mémoire sert à l'initialisation du serveur.

Nous avons choisi d'écrire le parseur en prenant exemple sur les outils de *parsing* déjà existant, dans notre cas *flex* et *bison*. On peut ainsi décomposer la phase de traitement du fichier en deux étapes principales résumées sur la figure 6.1 :

— Une décomposition du fichier en éléments appelés *tokens*, correspondant à des unités d'information contenues dans le fichier (nombre, caractère $, nom d'une carte, espaces ...).
— Une fonction qui, à partir d'un point dans le fichier, lit et renvoie le prochain *token*. Cela revient à lire les mots d'une phrase : c'est le rôle du **lexeur**.
— Une fonction qui récupère les tokens et les traduit en données du programme (en remplissant la structure memoire). Cela revient à structurer une phrase à partir de mots et voir si elle est syntaxiquement correcte : c'est le rôle du **parseur**.

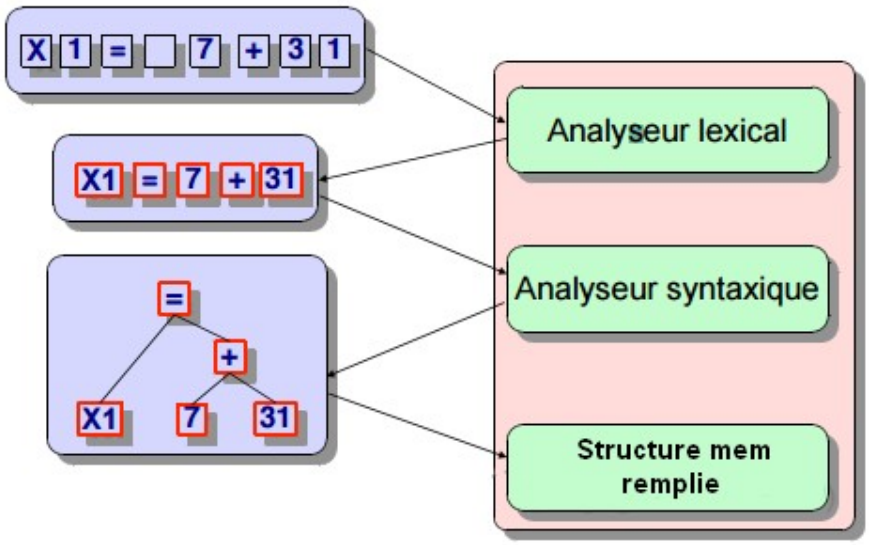


FIGURE 6.1 – Association d'un lexer et d'un parser

6.2 Serveur

6.2.1 Structures de données du serveur

Dans un premier temps, il nous a fallu faire des choix d'implémentation concernant les différentes structures de données que nous serions amenés à manipuler, telles que le plateau, les cartes, ou encore les stratégies.

Nous avons notamment implémenté une structure *player_s*, qui correspond à un joueur, et qui contient donc les informations pertinentes sur celui-ci. Cette structure contient un pointeur générique *handler* qui sert lors de l'utilisation des fonctions dlopen, ainsi que deux fonctions *play* et *draw_card* qui sont chargées depuis la bibliothèque dynamique (autrement dit le client) grâce à dlopen. Elle renseigne également sur les cartes détenues parle joueur en question, à l'aide d'un tableau *player_cards* de *card_id* de taille *n_player_cards*. Nous avons choisi de représenter la main du joueur par un tableau car l'accès aux cartes du joueur se fait en temps constant. De plus, sauf lorsque la pile devient vide, *n_player_cards* ne varie pas.
Dans le cas du plateau de jeu, nous avons implémenté une structure *board_s* dans laquelle se trouvent les informations propres au plateau récupérées lors du parsing, comme sa longueur, sa largeur, ou encore l'emplacement des trous (à l'aide d'un tableau de *positions*) et les *positions* de départ et d'arrivée. De plus, *board_s* contient une matrice de cartes, permettant l'état actuel du plateau de jeu.
Une structure de donnée, *tile_s*, nous permet de modéliser une carte de jeu et la position dans laquelle elle a été posée sur le plateau, si elle s'y trouve. En outre, cette structure contient des informations sur les accès Nord, Sud, Est et Ouest de la carte, afin de pouvoir déterminer si le placement d'une autre carte à sa suite sur le plateau est valide.
Une autre structure utile à notre projet est *game_s*, qui correspond à une manche du jeu, et qui contient un pointeur vers un *board_s*, afin d'avoir accès à l'état actuel du plateau de jeu, un tableau de pointeurs vers des *player_s* permettant de donner tour à tour la main aux différents joueurs de la manche, ainsi qu'un tableau *previous_moves* donnant accès aux derniers coups joués. Cette structure contient également Notre code contient également une liste *deck_s*, qui correspond à la pioche, que l'on a choisi de modéliser par ce type de données, soit une liste *cards* de cartes *tile_s* et le nombre de celles-ci dans la pioche. Le choix d'une implémentation à l'aide d'une liste s'explique par la taille variable d'une pioche, qui diminue à chaque fois qu'un joueur tire une nouvelle carte, et qui peut finir par être vide.

6.2.2 Gestion des clients avec *dlopen*

Le serveur fait jouer à tour de rôle différents joueurs, chacun correspondant à un client chargé dynamiquement (bibliothèque dynamique au format *.so*). On souhaite que chaque client ait une stratégie différente, ainsi chacun dispose de ses propres fonctions (initialisation, play, draw_card) et ses propres variables statiques.

Pour mettre cela en place, on a utilisé les fonctions de la famille *dlopen*[1]. Il est à noter que le flag *LM_ID_NEWLM* est nécessaire dans le cas où on charge plusieurs clients ayant la même stratégie (si on ne le fait pas, les mémoires de ces clients se "superposent" ce qui créé des conflits).

```
void *handler = dlmopen(LM_ID_NEWLM, client_library_paths_a[i], RTLD_NOW)
```

Ensuite, pour charger les différentes fonctions propres au client on utilise dlsym :

```
players_a[id].play = dlsym(handler, "play")
```

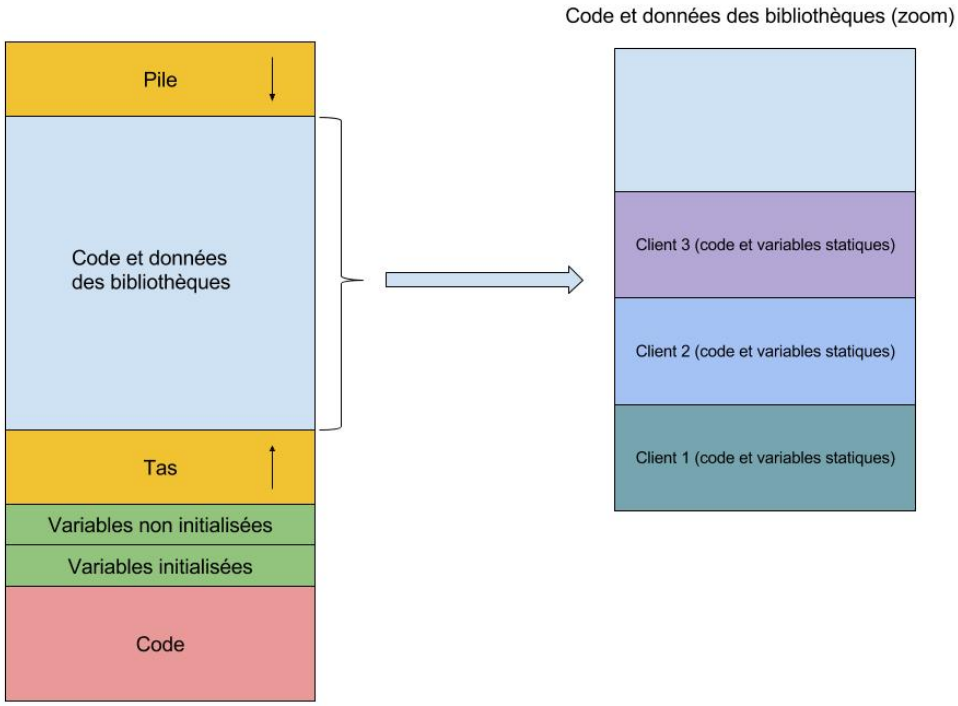


FIGURE 6.2 – Organisation de l'espace d'adressage (mappage) d'un processus

1. http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

6.2.3 Distribution des cartes

Le *deck* (ie. la pioche du jeu) correspond à une liste chaînée (le type *list_t* est implémenté dans un module à part). À l'initialisation, il est créé à partir des informations récupérées lors du *parsing*, puis il est mélangé à l'aide de la routine *list_shuffle*.

La distribution des cartes, puis la pioche après un tour de jeu consiste à récupérer l'élément en tête de cette liste (une fonction *pop* a été écrite à cet effet).

NB : il ne faut pas oublier de demander au client d'appeler la fonction *draw_card* avec la carte piochée pour qu'il se mette à jour.

6.2.4 Système anti-triche, véracité des coups joués

Lorsqu'un joueur a choisi quel coup il allait jouer (à l'aide de la structure *move*), le serveur doit alors vérifier que ce coup est valide, auquel cas il l'applique. Pour cela nous avons introduit une information supplémentaire pour les *tiles*, les points cardinaux indiquant s'il existe un chemin dans cette direction (1 si c'est le cas, 0 sinon). Cette représentation est visible sur la figure 6.2.4.

FIGURE 6.3 – Des cartes chemin positionnées avec leurs points cardinaux

La vérification d'un coup se déroule en plusieurs étapes :

— On vérifie si le joueur a la carte dans sa main
— Carte action : on vérifie que le joueur sur lequel on joue la carte est correct
— Carte chemin : on vérifie que les coordonnées x et y ne débordent pas du plateau
— On vérifie qu'il n'y a pas déjà de carte à cet emplacement
— On vérifie que la carte est adjacente à un chemin déjà existant
— On vérifie que les points cardinaux sont corrects

Si un de ces critères n'est pas respecté, le coup est considéré comme invalide et le joueur est *kické* de la partie.

6.2.5 Gestion des manches et distribution des pépites

Gestion des manches

Distribution des pépites

A la fin d'une manche, chaque joueur du type de nain gagnant se voit distribuer un nombre de pépites. Nous avons donc implémenté pour cela une fonction gold_distribution. Dans cette fonction, on détermine le type de nain gagnant avec la fonction *winner*, puis on applique une stratégie de distribution différente en fonction du type -honest ou saboteur- du nain.

Si les nains sont des sapoteurs, on regarde le nombre de sapoteurs dans la partie. Moins il y a de sapoteurs dans la partie, plus ceux-ci gagnent d'argent. Par exemple, s'il y a un unique sapoteur, celui-ci se voit distribuer 4 pépites et si il y en a 3 ou 4, les sapoteurs gagnent 2 pépites chacun. On modifie ensuite le champ gold des joueurs en conséquence.

Si les nains sont des honest, on prend autant de cartes gold qu'il n'y a de joueurs totaux, sauf s'il y a 10joueurs, auquel cas on prend 9 cartes gold. Ces cartes sont ensuite distribuées dans le sens des aiguilles d'une montre pour chacun des honest, en commençant par le joueur qui a découvert le trésor. Pour cela, on parcourt l'ensemble des joueurs, et si ce sont des honest, on distribue un nombre aléatoire de gold grâce à la fonction *random_gold*.
L'ensemble des cartes golds est modélisé par une structure *gold_cards* contenant un tableau *gold_tab*, ainsi que sa longueur *n_gold_cards*. Le tableau contient des entiers représentant la valeur de ces cartes, c'est à dire 1, 2 ou 3 pépites. On initialise cette structure dans un premier temps avec la fonction *initialize_gold_tab*. On choisit donc

une longueur de NB_GOLD_CARD1 + NB_GOLD_CARD2 + NB_GOLD_CARD3 où NB_GOLD_CARDX sont des variables globales correspondant au nombre de cartes distribuant x pépites, et on remplit un tableau de cette longueur par des -1.
Ensuite, on remplit dans la fonction *fill_gold_tab* un tableau dans l'ordre des indices par la valeur du nombre de pépites sur une carte(1, 2 ou 3), et on réitère l'action autant de fois qu'il y a de pépites correspondant à NB_GOLD_CARDX. On obtient alors un tableau de la forme [1,1,1...2,2...2,3..3]
Puis, on choisit aléatoirement un indice du tableau *gold_tab*, et si ce tableau n'est pas rempli à l'indice donné, on le remplit par le premier élément du tableau construit précédemment. On passe ensuite à l'élément suivant du tableau trié et on choisit un nouvel indice aléatoire pour remplir le tableau vide, et ainsi de suite. Si un indice est déjà occupé par une carte gold, on choisit d'incrémenter de 1 l'indice modulo la taille du tableau jusqu'à obtenir un indice disponible, ce qui nous évite d'obtenir des boucles extrêmement longues dans le cas où le tableau est presque rempli. On obtient ainsi un tableau *gold_tab* mélangé.
Enfin, la fonction *random_gold* consiste alors à récupérer la dernière carte du tableau mélangé, et à réduire de 1 la longeur de ce tableau. Cela permet ainsi de conserver l'aspect tirage de la carte, et donc de réduire les chances d'obtenir un certain nombre de pépites lorsque une carte est déjà tirée.

La fonction *random_gold* nécessite cependant de trouver le type de nain ayant gagné la partie, et plus spécifiquement le nain précis ayant atteint le trésor pour les nains honest. Pour cela, nous avons implémenté la fonction *winner*, qui renvoie un *struct player*. Cette fonction, en considérant les précédents mouvements de tous les joueurs dans le tableau *previous_moves*, vérifie, dans le cas où le joueur a posé une carte position, que celle-ci corresponde à une carte accédant à l'arrivée. La fonction *position_comparison_end* gère la comparaison entre le dernier mouvement d'un joueur précis et une arrivée donnée en paramètre. Si un joueur arrive effectivement sur une arrivée, on récupère alors l'identifiant *id* du joueur, ainsi que son type, et on renvoie la *struct player* correspondante. En revanche, si aucun joueur n'est arrivé sur un trésor et que la pile de carte, ainsi que les mains des joueurs, est vide, alors les sapoteurs sont vainqueurs. On récupère alors le type des vainqueurs et on renvoie la structure correspondante.

6.3 Client

6.3.1 Structures de données du client

Nous avons dû tout d'abord définir les structures de données de notre client. Pour cela, nous avons fait en sorte de ne garder que les informations utiles au client, ce qui permet d'éviter les triches potentielles. Ainsi, nous avons défini les structures suivantes :
- *game_s* définit le jeu du point de vue du joueur. Il contient comme champs l'id du joueur, le type de nain du joueur (SAPOTEUR ou HONEST), le nombre de cartes totales dans le jeu et dans la main du joueur, la contenu de la main du joueur modélisé par un pointeur vers un tableau d'*enum card*, le nombre de joueurs, le plateau, ainsi que l'état des joueurs, c'est-à-dire les outils cassés ou non et éventuellement le niveau de confiance de ce joueur dans le cas d'une stratégie intelligente. Le niveau de confiance du joueur est déterminé par deux variables globales CONFIDENT et UNCONFIDENT qui permettent, si l'entier *confident_lvl* est au-dessus de CONFIDENT ou en-dessous de UNCONFIDENT, de supposer ce joueur comme étant un allié ou non.
- le plateau est modélisé par une *struct board_s*, qui contient la pile de carte, la hauteur et la largeur du plateau, le nombre et la position des trésors, la position de départ, le nombre et la position des trous potentiels sur le plateau.
- la pile de carte est représentée par un tableau de *struct tiles_s*, correspondant à l'identifiant d'une carte, et à sa direction, c'est à dire le sens dans lequel la carte est posée (vers le haut ou vers le bas) .
- l'état de chaque joueur est défini par l'identifiant du joueur, ainsi que par 3 entiers correspondant respectivement à *axe_broken*, *lamp_broken* et *cart_broken*. Ces entiers sont initialisés à 0 pour représenter des outils intacts, et prennent la valeur 1 s'ils sont endommagés. Dans l'interface /textitclient_strategy.c, il existe un champ supplémentaire *confident_lvl*, décrit ci-dessus.

6.3.2 Initialisation et libération de la partie

6.3.3 Action du joueur

L'action du joueur est définie dans la fonction *play*. Cette fonction est divisée en deux partie : Dans une première partie, on regarde les actions itérées par les joueurs durant le précédent tour, et on modifie en conséquence soit le tableau, soit les joueurs visés. Dans une seconde partie, l'action struct_move du joueur concerné par le client est mise à jour. Dans cette seconde partie, la stratégie est implémentée, nous l'aborderons donc plus loin dans ce rapport.

La première partie de la fonction *play* fonctionne de la manière suivante : On parcourt le tableau previous_moves correspondant à la dernière action effectuée par chaque joueur dans le tour. On regarde le type d'action qui est concernée (ADD_PATH_CARD, PLAY_BREAK_CARD, ...)et on modifie les champs nécessaires sur le plateau ou dans le champ d'état des joueurs.

Une fois la carte jouée, le client se débarrasse de celle-ci. Une fonction a été implémentée pour cela, *remove_card*. Cette fonction diminue le nombre de carte dans la main du joueur de 1, et décale les cartes d'indice supérieur à la carte jouée de -1 également.

Le joueur doit tirer une nouvelle carte une fois son coup joué. Pour cela, on a implémenté une fonction *draw_card*, qui prend en argument le card_id d'une carte et associe au dernier element de la main du joueur, présupposé vide à ce moment, la carte donnée en argument. On augmente egalement n_players_cards de 1.

7 Stratégies

Afin de faire jouer le client, nous avons cherché à implémenter des stratégies à différents niveaux de complexité. Dans un premier temps, nous avons implémenté une stratégie *random* permettant de vérifier que le serveur et le client étaient compatibles. Nous avons ensuite développé une stratégie *human*, permettant à un ou plusieurs joueurs d'interagir avec le serveur. Nous avons finalement implémenté une stratégie intelligente, permettant à l'ordinateur de jouer une partie en suivant une stratégie entièrement implémentée sur ordinateur. maladroit
La stratégie employée par le client est implémentée dans la fonction *play*. Cette fonction décrit dans un premier temps les modifications subies par le joueur selon les actions des autres joueurs durant le tour précédent, puis elle modifie dans un second temps le champs *move* du joueur concerné selon l'action choisie par celui-ci. C'est donc dans cette seconde partie que la stratégie du joueur intervient.

7.1 Stratégie random

La première stratégie que nous avons implémentée, et qui est également la plus basique, est la stratégie *random*. Cette stratégie consiste à tirer un nombre aléatoire d'*enum card* parmi toutes les cartes dans la main du joueur, et à jouer celle-ci. Selon le type de carte jouée, les différents paramètres du champ move concernés sont alors modifiés. En effet, si une carte ADD_PATH_CARD ou BOULDER_CARD est jouée, on modifie les champs correspondant aux coordonnées d'emplacement de la carte en tirant aléatoirement des valeurs modulo la largeur et la longueur du plateau pour ces coordonnées. On choisit également une direction de carte aléatoire pour les ADD_PATH_CARD. Si une carte REPAIR_CARD ou BREAK_CARD est jouée, on tire aléatoirement un joueur à qui appliquer cette carte et on modifie le champ du *move* sur lequel la carte est appliquée. Enfin, on retire cette carte de la main du joueur.
Par ailleurs, le serveur vérifie que le coup proposé par le joueur est acceptable. Si il ne l'est pas, le joueur est alors éliminé de la partie. Le joueur ne peut donc continuer à jouer aléatoirement que si son coup est valide, et est éliminé le cas échéant, ce qui limite ses chances de gagner. Par ailleurs, si aucun joueur n'atteint une arrivée, les sapoteurs gagnent la partie par défaut.

7.2 Stratégie joueur

La deuxième stratégie que nous avons cherché à implémenter est une stratégie permettant à un humain d'interagir avec le serveur afin de jouer au jeu du saboteur. Pour cela, nous avons mis en place une interface via le terminal grâce à la fonction *scanf*. Les cartes contenues dans la main du joueur sont affichées à l'écran, et le joueur est invité à choisir la carte qu'il souhaite jouer, ainsi que les coordonnées de l'emplacement où il souhaite placer la carte, ou bien la personne à qui il souhaite appliquer la carte, selon le type de carte choisie par le joueur. Comme pour la stratégie précédente, les champs du *move* du joueur sont alors modifiés en conséquence en récupérant les paramètres entrés par le joueur. Une fonction auxiliaire a dû par ailleurs être implémentée afin de traduire les *enum cards* (ADD_PATH_CARD, BOULDER_CARD...) en chaîne de caractère effectivement lisible par le joueur depuis le terminal.

7.3 Stratégie intelligente

La dernière stratégie que nous avons codé consiste à implémenter un algorithme permettant à la machine de jouer seule un personnage du jeu. Celle-ci est divisée en deux types de stratégie, selon que le bot soit sapoteur ou honest. Selon que le joueur soit un type de joueur ou l'autre, il va choisir son action selon un certain ordre de priorité, en fonction des cartes qu'il possède. En effet, l'ordinateur va parcourir les cartes qu'il a en main en espérant trouver un certain type de carte. Il vérifie qu'il peut jouer ce coup et, le cas échéant, passe au type de carte de priorité inférieure.

7.3.1 ordre de priorité des stratégies honest et sapoteur

Les priorités pour les deux types de joueurs sont donc les suivantes :
pour la stratégie honnête :
- Si le joueur possède une carte qui permet d'avancer sur le plateau, il joue cette carte à l'aide de la fonction *play_path_card_honest*.
- Sinon, il joue une carte permettant de réparer les outils d'un joueur du même camp grâce à la fonction *play_repare_card*.
- Sinon, il joue une une carte permettant de casser l'outil d'un joueur adverse avec la fonction *play_break_card*.
- Sinon, il se défausse d'une carte avec la fonction *play_discard_honest*.

Pour les joueurs sapoteurs, la stratégie est la suivante :
- Si le joueur possède une BOULDER_CARD, celui-ci la joue automatiquement. Ce point

reste critiquable, la BOULDER_CARD pouvant être utilisée de manière stratégique en la gardant en main. Le joueur pourrait alors jouer la carte à un moment opportun afin d'empêcher les honest d'avancer. Nous n'avons néanmoins pas eu le temps d'implémenter une telle stratégie, et la BOULDER_CARD étant extrêmement efficace, nous avons décidé de l'utiliser en première priorité.
- Si le joueur ne possède pas de telle carte, alors le sapoteur regarde si il possède une carte BREAK_CARD et l'utilise avec la fonction *play_break_card*. - Sinon, le joueur utilise une carte qui permet de faire dévier les joueurs honest de leur chemin. On utilise pour cela la fonction *play_path_card_sapoteur*.
- Sinon, le joueur se défausse d'une carte.
Puisque le joueur peut toujours se défausser d'une carte, un sapoteur ne répare jamais les outils d'un joueur, élément qui réduit ses chances d'être pris pour un honest.

7.3.2 Description des fonctions de stratégie

Les fonctions *play_break_card* et *play_repare_card* parcourent l'ensemble des cartes du joueur, et regardent si il possède une BREAK_CARD ou une REPARE_CARD. Si c'est le cas, on parcourt l'ensemble des autres joueurs afin de regarder si l'indice de confiance qu'on leur a attribué est supérieur à la variable globale CONFIDENT (pour réparer les outils) ou inférieur à la variable globale UNCONFIDENT (pour casser les outils), et le joueur agit en conséquence. Pour le premier joueur concerné, on modifie les paramètres de *my_move* en fonction de la carte jouée et du joueur atteint.
Les fonctions *play_discard_honest* et *play_discard_sapoteur* consistent à définir un ordre de priorité pour les cartes à supprimer. En effet, on définit à l'initialisation une carte NO_CARD prioritaire à supprimer, puis on regarde les cartes de la main de joueur une par une, et si la carte suivante à une priorité d'élimination plus elevée (ce qui est nécessaire pour la première carte de la main du joueur), alors celle-ci est retenue à la place. Sinon, on garde la carte de priorité maximale actuelle. Selon que le joueur soit honest ou sapoteur, l'ordre définit n'est pas le même. Un sapoteur éliminera en priorité une REPAIR_CARD, puis une PATH_CARD s'il ne possède pas de carte de réparation, puis une BREAK_CARD, puis une BOULDER_CARD, tandis qu'un joueur honest éliminera d'abords une BOULDER_CARD puis une BREAK_CARD, puis une PATH_CARD, puis une REPAIR_CARD. Cet ordre reste discutable, mais c'est celui qui nous a paru le plus adéquat.

Par manque de temps, nous n'avons pas pu implémenter les fonctions *play_boulder_card*, *play_path_card_honest* ainsi que *play_path_card_sapoteur*

8 Conclusion

8.1 Bilan

Les objectifs principaux de ce projet ont été atteints. En effet nous disposons d'un serveur et de différents clients **fonctionnels** permettant de jouer à des parties de saboteur. De plus ceux-ci sont entièrement **intéropérables**, c'est-à-dire que notre serveur peut accepter des stratégies réalisées par d'autres groupes et vice-versa.

Ce projet nous a permis dans un premier temps de découvrir en profondeur les mécanismes du *parsing*. Il nous a également permis de comprendre le fonctionnement intrinsèque des bibliothèques dynamiques (*dlopen*) tout en consolidant nos acquis de C. Enfin, ce travail nous a permis de renforcer nos connaissances dans la gestion de projet, notamment dans l'application des méthodes agiles (*SCRUM*, Trello), mais aussi dans l'utilisation des outils de développement (CMake, Doxygen, gdb, valgrind).

8.2 Perspectives

Certains aspects du projet auraient plus être plus poussés si le temps nous l'avait permis.

Par exemple une interface graphique permettant de jouer de manière ergonomique (comme présenté dans l'analyse de l'existant figure 2) aurait été un plus.

De plus les stratégies restent assez sommaires. Une version plus poussée utilisant un paradigme min/max avec l'écriture d'une fonction d'évaluation plus complète aurait certainement engendré une intelligence artificielle plus robuste !

Enfin, il est immédiat de constater qu'un portage du jeu en réseau serait plutôt simple en considérant la façon dont le jeu a été programmé (intéropérabilité, communication seulement à l'aide de l'interface, modularité).

Annexes

```
# Les deux premiers entiers designent la largeur-width L et la
# hauteur-height H du jeu.

9 5

# Les positions de jeu sont des couples d'entiers (x,y)

# La specification du plateau de jeu sous la forme de H lignes
# contenant chacune exactement L caract res.
# Un trou est marque par un '%', le d part par un '>',
# une arrivee par un '$', sinon on utilise '*'.
********$
*****%***
>*******$
**%******
********$

# La liste des cartes utilisees dans le jeu sous la forme d'un couple
# (chaine de caracteres du langage "[A-Z_]+", entier).
V_LINE   4
H_LINE   3
V_CROSS  5
H_CROSS  5
X_CROSS  5
L_TURN   5
R_TURN   4
D_END    9

# Optionnellement les cartes de jeu supplementaires
# pour remplacer une carte en jeu
BOULDER 3

# Ainsi que les cartes pour casser et reparer des outils
B_AXE    3
B_LAMP   3
B_CART   3
R_AXE    2
R_LAMP   2
R_CART   2
R_ALL    3
```

Listing 8.1 – Format d'entrée d'un plateau de jeu

```
Configuration : 5x9
Card types    : 16
Nb of cards   : 61
Objectives    : 2 (4,4) (6,2)
Holes         : 2 (5,3) (2,1)
Allow boulder : yes
Allow breaks  : yes
Repair=Break  : yes
```

Listing 8.2 – Format de sortie (généré par le parseur)

```c
#include <stdlib.h>

/* Enum */
enum card_id { NO_CARD, CARD_V_LINE, ... };
enum direction { NORMAL, REVERSED, };
enum action { ADD_PATH_CARD, PLAY_BREAK_CARD, ... };
enum p_type { SAPOTEUR, HONEST};

/* Data structures */
struct position { unsigned int x,y; };
struct move {
  unsigned int player;   // player issueing the move
  enum action act;       // action taken
  enum card_id card;     // specific card
  struct position onto;  // targeted position on the board
  enum direction dir;    // direction of the card
  unsigned int onplayer; // player targeted by the move
};

/* Functions (interaction between server and client) */
char const* get_player_name();
int initialize(unsigned int id,
               enum p_type type,
               unsigned int width,
               unsigned int height,
               struct position start,
               size_t n_objectives,
               struct position const objectives[],
               size_t n_holes,
               struct position const holes[],
               unsigned int n_cards_total,
               size_t n_player_cards,
               enum card_id const player_cards[],
               unsigned int n_players);

struct move play(struct move const previous_moves[], size_t n_moves);
int draw_card(enum card_id card);
int finalize();
```

Listing 8.3 – sapotache_interface.h permettant la communication client/serveur