



Rapport de Projet

Info4B

Guillaume Beldilmi & Clément Doublet

Table des matières

Introduction	2
I. Conception	3
I.1. Découpage du projet.....	3
I.2. Analyse fonctionnelle.....	3
I.3. Description des structures de données	5
II. Architecture logicielle	6
II.1. Conception en couches.....	6
II.2. Diagrammes de classes	6
III. Description des algorithmes principaux	7
III.1. corewar.mars	7
III.2. corewar.server	10
III.3. corewar.client	13
III.4. corewar.utils	14
IV. Présentation des warriors.....	16
Conclusion.....	17
Annexes.....	18
Annexe 1 : Jeux d'essai.....	18

Introduction

Ce rapport fait l'objet du rendu final du projet du module Info4B. Le but de ce projet est de réaliser le jeu Core War en Java et d'y appliquer les notions de système d'exploitation et réseaux vues en cours.

Notre implémentation doit nous permettre de lancer différentes parties de Core War en parallèle sur le serveur via plusieurs clients. Le serveur est chargé de traiter l'ensemble des demandes et de gérer l'exécution des différentes parties, il doit aussi être capable de générer et sauvegarder le classement des warriors ainsi que le mettre à jour au fil des parties. La soumission des warriors doit se faire via le réseau.

Dans un premier temps, nous aborderont la conception du projet, puis nous traiteront de la structure choisie pour notre implémentation, enfin, nous en verrons la description détaillée, cela nous permettra de voir en profondeur les logiques d'interaction entre les différents éléments de notre projet. Dans un second temps, nous verrons des exemples de warriors et quelques jeux d'essais afin d'illustrer notre implémentation.

Nous concluons ce rapport en évoquant ce que nous a apporté la réalisation de ce projet et les difficultés que nous avons rencontrées, enfin, vous traiterons des pistes d'amélioration de ce projet et de ce que ce dernier pourrait nous permettre de réaliser à l'avenir.

I. Conception

I.1. Découpage du projet

Afin de concevoir notre projet, nous avons commencé par le découper en sous problèmes. Pour rappel, le projet est un jeu de Core War, le but du jeu est de faire combattre différents programmes écrits en redcode appelés warrior dans un espace défini. Dans notre projet, le nombre de warrior maximum n'est pas limité, il faut cependant au minimum deux warrior pour lancer une partie. Le jeu doit fonctionner en réseau, la partie se joue sur le serveur, les clients interagissent avec le serveur et réagissent aux réponses renvoyées par celui-ci. Il faut aussi implémenter un classement global sur le serveur qui pourra être consulté par les clients. Les joueurs peuvent créer et rejoindre des parties, uploader leur warrior et afficher les parties disponibles. Le serveur quant à lui peut gérer plusieurs parties en même temps.

Ce sujet pose plusieurs problèmes qui nous ont permis de le découper en sous problèmes à résoudre. Nous allons définir les questions que nous nous sommes posées et nous nous aiderons de celles-ci pour la rédaction de l'analyse fonctionnelle du projet.

Pour la partie concernant le simulateur MARS, nous nous sommes posé les questions suivantes. Comment le représenter le redcode en mémoire ? Comment manipuler les instructions entre elles ? Comment gérer la mémoire du simulateur en fonction du nombre de warrior et de la taille de leur programme ? De quelle façon doit-on organiser l'exécution des warriors ?

En ce qui concerne la partie client et serveur, nous nous sommes posé les questions suivantes. Comment faire en sorte que plusieurs clients communiquent avec le serveur sans que ce soit bloquant ? Quelle est la meilleure méthode pour communiquer avec le serveur ? Comment gérer différentes parties simultanément ? Comment mettre à jour le classement automatiquement sans problèmes de concurrence ? Comment uploader un warrior sur le serveur ? Quel est le déroulement d'une partie ?

Pour finir, nous avons décidé de nous répartir le projet en deux grosses parties et de rassembler ces deux parties en un bloc à la fin du développement du projet. Guillaume a travaillé sur le simulateur MARS, Clément a réalisé la partie client/serveur et la lecture de fichier.

I.2. Analyse fonctionnelle

Le simulateur MARS

Pour représenter le redcode en mémoire, nous avons choisi de définir un objet représentant une instruction, cet objet aurait un unique attribut qui serait la concaténation de la représentation binaire de tous les champs. De cette façon, on pourrait facilement additionner, soustraire et comparer deux « instructions » entre elles. Régulant ainsi le problème de la manipulation par la même occasion.

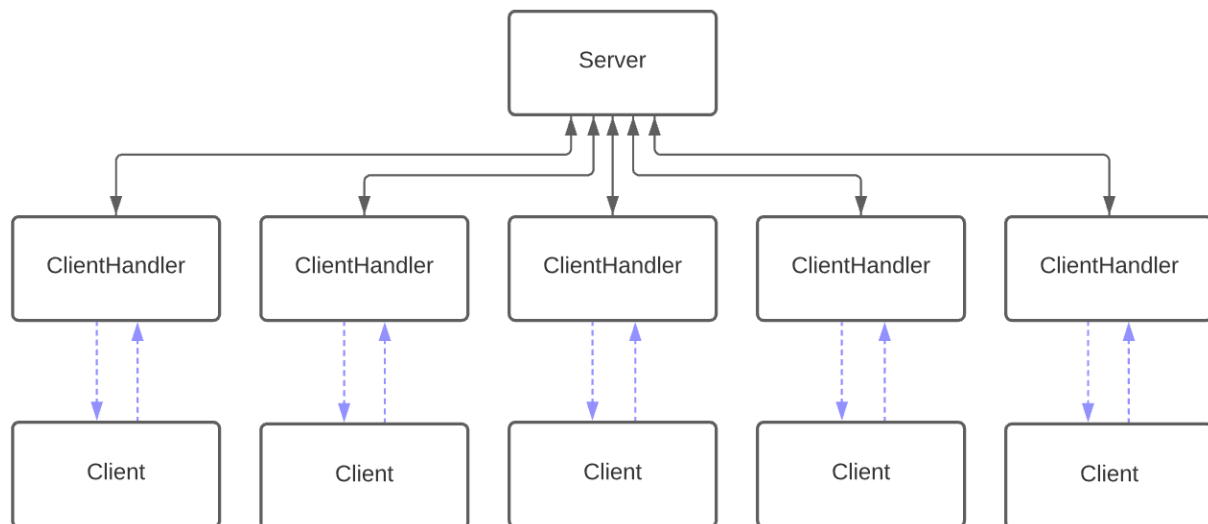
Pour résoudre le problème de la taille de la mémoire du simulateur, deux solutions sont possibles : avoir une taille fixe ou une taille dynamique (bien sûr, la taille de la mémoire ne changera pas en pleine simulation). Une taille fixe limiterait beaucoup les possibilités de simulations et nécessiterait de faire beaucoup de vérifications avant l'initialisation (tailles des programmes, espaces vides, etc...). Une taille dynamique sera plus simple à implémenter : seule la taille du plus grand programme sera prise en compte et la taille de la mémoire serait deux fois la taille du plus long programme multipliée par le nombre de programmes en jeu, ainsi, il y aurait un espace entre chaque programme et il n'y aurait pas de limite de joueurs dans une partie. Le choix le plus simple est donc de donner une taille dynamique à la mémoire.

Pour organiser l'exécution des warriors, il est préférable de faire un ordonnanceur : faire exécuter des threads tour à tour serait contre-productif et cela complexifierait grandement l'implémentation.

La partie client/serveur

Pour faire en sorte que la communication entre le serveur et des clients multiples ne soit pas bloquant. Nous avons décidé de définir une classe ClientHandler côté serveur. Cette classe est un thread et a pour but de communiquer avec une instance de la classe Client. Pour chaque demande de connexion, le serveur initialise une nouvelle instance ClientHandler et lui transmet le socket de communication. Les gestionnaires de client restent alors en écoute et

transmettent les données au serveur qui peut les traiter sans être bloqué par l'écoute continue du flux de données des clients.



Pour que le serveur puisse communiquer avec les clients et surtout interpréter leurs demandes, il faut définir une règle de communication, cette règle et ses codes doivent être les mêmes des deux côtés. Nous avons opté pour la construction d'une api et l'utilisation d'un modèle de requête/réponse. Pour le modèle de requête réponse, le client envoie une requête au serveur et attend une réponse de celui-ci. La requête est reçue par le gestionnaire de client qui demande une réponse au serveur et qui l'envoie au client. Pour ce qui est de l'api, nous avons définis des codes constants représentant les différentes requêtes et réponses possibles ainsi qu'un séparateur. La classe API permet de s'assurer que les requêtes sont correctement formatées, le client construit ses requêtes à partir des méthodes définies par cette classe en lui passant les données nécessaires. C'est cette requête formatée qui est ensuite envoyée et analysée par le serveur.

La gestion simultanée des parties nécessite d'utiliser des threads pour éviter le blocage du serveur lorsqu'une partie est en cours. Lorsqu'un utilisateur décide de créer une partie (classe Game), celle-ci est initialisée, son thread est démarré et elle est ajoutée dans une liste. L'index de cette partie dans la liste sert ensuite d'identifiant pour la retrouver. C'est avec cet identifiant qu'un joueur peut rejoindre une partie.

Pour mettre à jour le classement automatiquement et sans problème de concurrence nous avons opté pour une solution simple. Tout d'abord le gestionnaire de classement (*ClassementHandler*) est un thread, ce qui permet d'éviter le blocage du serveur. Il est initialisé dès le lancement du serveur. Son fonctionnement est simple, il vérifie en boucle l'état de chaque partie. Si le thread d'une partie est mort (ce qui implique que la partie est terminée) il récupère le classement de cette partie et met à jour le fichier de classement. Cette méthode permet de pouvoir traiter toutes les parties, de mettre à jour le classement automatiquement à la fin de chaque partie et surtout de ne traiter qu'une partie à la fois, ce qui permet d'éviter la concurrence.

Pour l'upload d'un warrior sur le serveur, nous utilisons l'api défini dans la classe API. L'utilisateur commence par choisir un warrior en tapant son nom et son extension. Si le fichier existe, une requête est construite avec un code de requête UPLOADWARRIOR le contenu de la requête est défini avec le nom du warrior du joueur et chaque instruction de son programme avec un séparateur entre le nom et chaque instruction. Cette chaîne de caractères est ensuite interprétée par le serveur, le programme est converti en tableau de chaîne de caractères et est utilisé pour initialiser un nouveau warrior et l'ajouter à la partie du joueur.

Pour ce qui est du déroulement d'une partie il est peu pratique de demander à chaque client d'indiquer quand il est prêt pour démarrer la partie. Alors nous avons décidé que la partie serait en attente tant que le nombre maximum de joueurs définis à la création n'est pas atteint et tant que tous les warriors ne sont pas uploadés. De ce fait la partie se lance automatiquement dès que tous les joueurs sont prêts. Ensuite le simulateur est initialisé avec les warriors des joueurs puis il est lancé. A la fin de la simulation, les warriors sont classés du premier au dernier puis ce classement est envoyé à chaque client.

I.3. Description des structures de données

Structures de données du simulateur MARS

La plupart des données manipulées par MARS auront une taille définie et fixe, donc les collections utilisées seront principalement des tableaux.

Afin de simplifier les manipulations des instructions en mémoire (addition, soustraction, comparaisons), nous avons définis des objets *Core* ayant pour unique attribut un entier puis d'appliquer des opérations binaires sur cet attribut pour extraire les différents champs de la valeur interne.

De plus, pour simplifier l'identification des champs des objets *Core*, nous avons créés les énumérations *OpCode* et *AddressMode*. Cela permet de manipuler des objets (*OpCode.DAT*, *OpCode.MOV*, *OpCode.ADD*, etc... et *AddressMode.IMMEDIATE*, etc...) qui sont plus explicites que des entiers au sein du code.

Structures de données pour la partie client/serveur

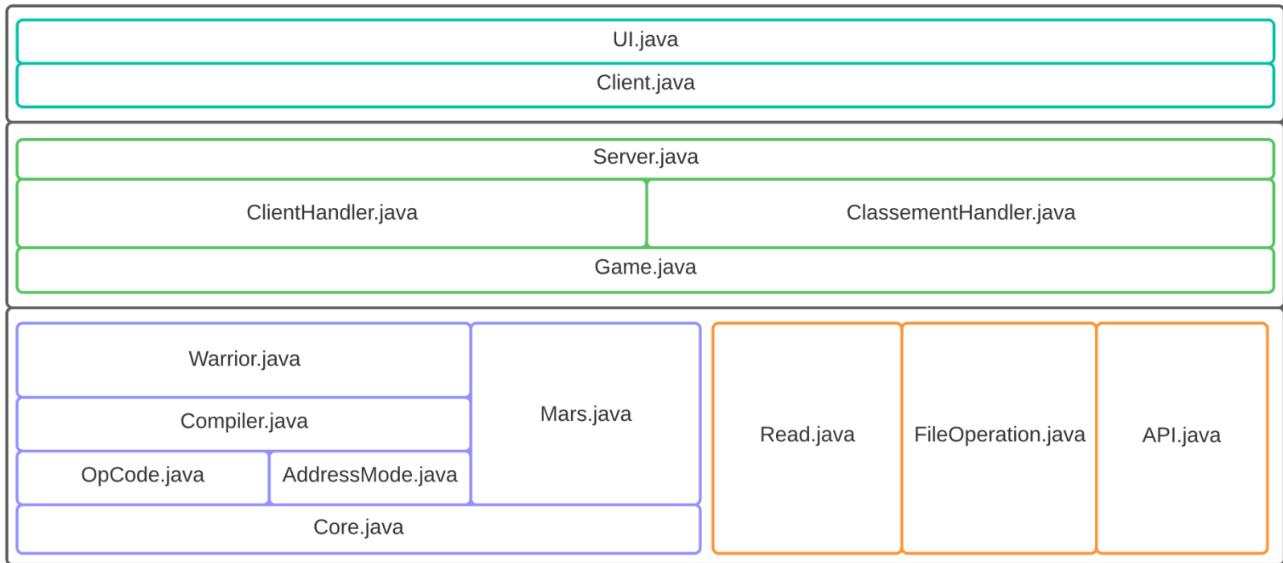
Pour la partie client et serveur Nous avons choisi d'utiliser deux structures de donnée principales. Nous avons utilisé en premier lieu des *ArrayList*. Nous ne connaissons pas à l'avance le nombre de clients qui vont se connecter, le nombre de parties qui seront créées ou encore le nombre d'instructions d'un *Warrior*. Cette structure de donnée nous permet donc de gérer facilement des données de taille variables.

Pour la partie requête/réponse et API, nous avons décidé d'encoder l'information transmise dans des chaînes de caractères, cette représentation est plus simple à gérer dans notre cas car nous n'avons pas beaucoup de données à transmettre lors de nos différentes requêtes et réponses. Les informations sont simplement séparées par un séparateur.

II. Architecture logicielle

II.1. Conception en couches

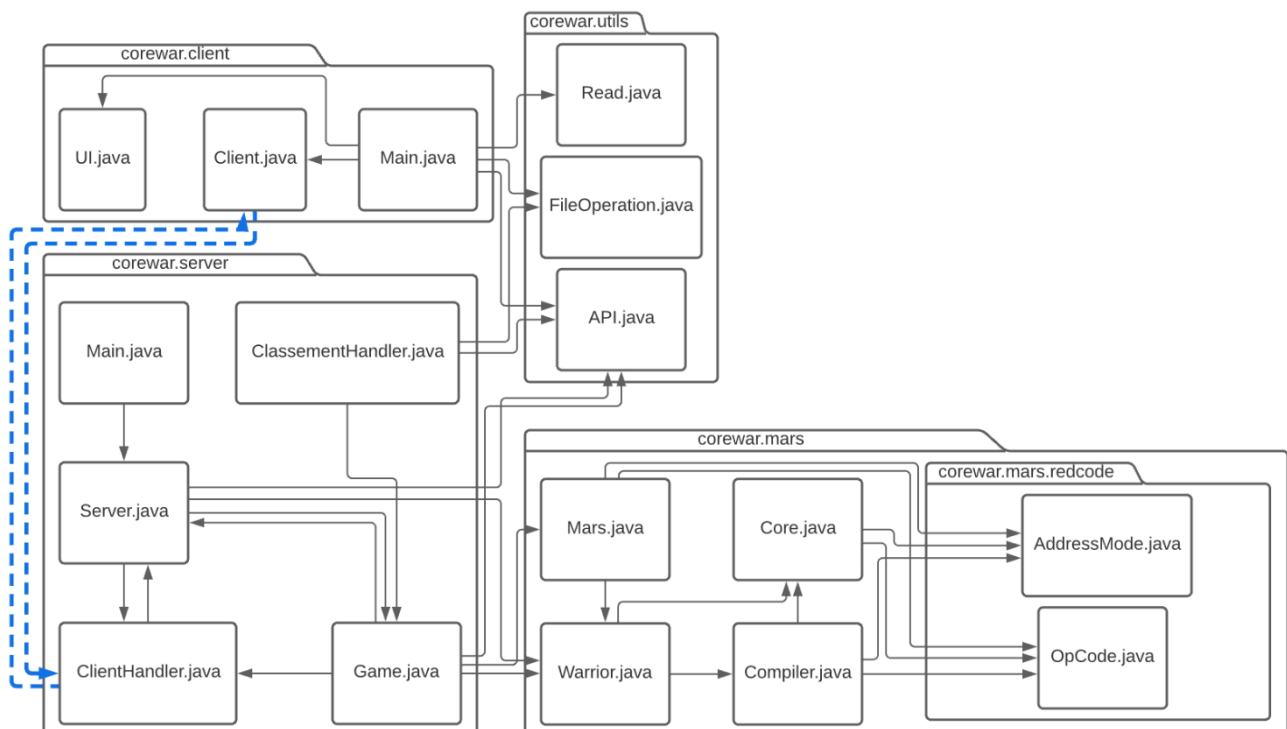
Nous avons construit notre projet sous forme de plusieurs couches fonctionnelles, où les couches supérieures (plus proches de l'utilisateur) utilisent les couches sous-jacentes.



Chaque couleur représente un package principal du projet.

II.2. Diagrammes de classes

Afin de faciliter la compréhension de l'ensemble du projet, nous avons réalisé un diagramme de classes représentant l'ensemble des classes du projet au sein de leur package respectif ainsi que la logique d'interaction entre ces dernières.



Les flèches grises représentent les relations d'utilisations entre les classes. Les flèches bleues représentent les interactions qui se feront via le réseau.

III. Description des algorithmes principaux

III.1. corewar.mars

Mars repose principalement sur la classe *Core*, cette dernière représentera une instruction *redcode* en mémoire. Or, afin de pouvoir manipuler correctement les instances de *Core*, il nous faut déjà définir deux énumérations très utiles : *OpCode* et *AddressMode*. Ces énumérations seront rassemblées dans un sous-package *corewar.mars.redcode*.

corewar.mars.redcode

OpCode.java

Cette énumération nous permettra d'identifier les différents types d'instructions *redcode*. A noter que notre version n'implémente que les *OpCode* suivants : *DAT*, *MOV*, *ADD*, *SUB*, *JMP*, *JMZ*, *JMG*, *DJZ* et *CMP*.

Les deux premières méthodes sont statiques car elles servent à identifier les *opcodes* et à les retourner sous la forme d'instance prédéfinies dans l'énumération (les remplacer par des constructeurs aurait créer une nouvelle instance de l'énumération et aurait rendu l'identification et la comparaison des *opcodes* plus compliquée).

static OpCode getOpCode(String mnemonic) : Cette méthode permet de récupérer l'*OpCode* correspondant à un mnémotique passé en paramètre, si le mnémotique ne correspond à aucun *OpCode* de l'énumération, on lève une *RuntimeException* en spécifiant que le mnémotique n'est pas reconnu.

static OpCode getOpCode(int code) : Cette méthode permet aussi de récupérer un *OpCode* défini dans l'énumération mais, cette fois à partir du code correspondant passé en paramètre. Si le code passé est plus grand que le nombre d'*OpCode* de l'énumération, on ajuste la valeur du code avec un modulo du nombre d'*OpCode* définis dans l'énumération.

Les méthodes suivantes sont non statiques car elles sont propres à l'instance de l'énumération.

int getCode() : Cette méthode permet de récupérer le code correspondant à l'*OpCode*.

String toString() : Cette méthode permet de récupérer le mnémotique correspondant à l'*OpCode* sous forme de chaîne de caractères.

AddressMode.java

Cette énumération nous permettra d'identifier les différents types d'adressages *redcode*, elle partage certaines similarités à l'énumération *OpCode*. A noter que notre version n'implémente que les modes d'adressages immédiat (*IMMEDIATE*), direct (*DIRECT*) et indirect (*INDIRECT*).

Les deux premières méthodes sont statiques car elles servent à identifier les modes d'adressages et à les retourner sous la forme d'instance prédéfini dans l'énumération (pour les mêmes raisons que l'énumération *OpCode*).

static AddressMode getAddressMode(char c) : Cette méthode permet de récupérer l'adressage correspondant à un caractère passé en paramètre (le premier caractère de l'argument), si le caractère est un '#' l'adressage est *IMMEDIATE*, si le caractère est un '@' l'adressage est *INDIRECT*, si le caractère n'est pas reconnu, on suppose que l'adressage est *DIRECT* (car l'argument peut commencer par un chiffre ou un signe '-').

static AddressMode getAddressMode(int code) : Cette méthode permet aussi de récupérer un adressage défini dans l'énumération mais, cette fois à partir du code correspondant passé en paramètre. Si le code passé est plus grand que le nombre d'adressages de l'énumération, on ajuste la valeur du code avec un modulo du nombre d'adressages définis dans l'énumération.

Les méthodes suivantes sont non statiques car elles sont propres à l'instance de l'énumération.

int getCode() : Cette méthode permet de récupérer le code correspondant à l'adressage.

String toString() : Cette méthode permet de récupérer le caractère correspondant à l'adressage sous forme de chaîne de caractères ("#" pour *IMMEDIATE*, "@" pour *INDIRECT*, et une chaîne vide par défaut).

Core.java

Maintenant que nous avons défini les énumérations *OpCode* et *AddressMode*, nous pouvons définir la classe *Core*. Cette classe représente une instruction redcode en mémoire.

Elle possède un unique attribut privé : un entier *value* qui représente le code de l'instruction, cet attribut possède des accesseurs publics en lecture et en écriture. Cet entier stockera le code de l'instruction redcode, les codes d'adressages des deux arguments ainsi que les valeurs des deux arguments le tout sera concaténé en un entier de 32 bits à l'aide des opérations bit à bit (les quatre premiers bits correspondent à l'*OpCode*, les 2 bits suivants à l'adressage du premier argument, les 2 bits suivants à l'adressage du second argument, les 12 bits suivants à la valeur du premier argument et les 12 derniers bits à la valeur du second argument, à noter que les valeurs des arguments sont représentées en complément à deux).

Core() : Constructeur par défaut de la classe *Core*. Initialise l'attribut *value* à 0 (correspondant à l'instruction *DAT 0*).

Core(OpCode opCode, AddressMode addressModeArg1, AddressMode addressModeArg2, int arg1, int arg2) : Ce constructeur initialise l'attribut *value* à la concaténation des codes et des valeurs des arguments passés en paramètre.

Core(int value) : Ce constructeur initialise directement l'attribut *value* à la valeur passée en paramètre.

void add(int value) : Cette méthode permet d'ajouter une valeur à l'attribut *value*.

boolean decrement() : Cette méthode décrémente l'attribut *value* et retourne *true* si la valeur est supérieure à 0, *false* sinon. Cette méthode sera utile pour l'instruction *DJZ*.

Cette classe possède des accesseurs virtuels en lecture pour chaque champs de l'objet *Core*: **OpCode getOpCode()**, **AddressMode getAddressModeArg1()**, **AddressMode getAddressModeArg2()**, **int getArg1()** et **int getArg2()**.

int getArgComplement(int arg) : Cette méthode permet de corriger le complément à deux de la valeur passée en paramètre. Comme la valeur de l'argument est représentée en complément à deux et qu'on la récupère à partir d'un masque de 12 bits, on a besoin de corriger la valeur pour qu'elle corresponde à la valeur attendue si elle est négative.

void sub(int value) : Cette méthode permet de soustraire une valeur à l'attribut *value*.

String toString() : Cette méthode permet de récupérer la représentation de l'objet *Core* sous forme de chaîne de caractères. La première partie prend la forme de code redcode et la seconde partie représente la valeur interne de l'objet *Core* en binaire.

Compiler.java

Cette classe permet de compiler un programme redcode en mémoire. Elle possède une seule méthode statique : **static Core[] compile(String[] program)**. Cette méthode permet de compiler un programme redcode (sous forme de tableau de chaîne de caractère (un élément du tableau correspond à une instruction)) en un tableau d'objets *Core*. Si une instruction n'est pas reconnue ou incorrecte, une exception *RuntimeException* est levée en spécifiant l'origine de l'erreur. Séparer cette méthode du reste du code permet de faciliter la lecture du code.

Warrior.java

Cette classe représente le programme d'un joueur. Elle a pour vocation à être créée avant la création d'une instance de *Mars* afin d'être passée en paramètre au constructeur de cette dernière. Elle évitera à *Mars* d'avoir beaucoup trop d'attributs à manipuler. De plus, elle permettra de faire plus facilement le lien avec le code extérieur au package *corewar.mars* (notamment avec ses attributs *id*, *name* et *rank* et son constructeur).

Concernant les attributs de la classe :

Core[] program : Cet attribut permet de stocker le programme compilé du joueur avant l'initialisation de *Mars*.

int id : Cet attribut permet de stocker l'identifiant du joueur.

int position : Cet attribut permet de stocker la position du joueur dans la mémoire de *Mars*.

int rank : Cet attribut permet de stocker le rang du joueur (0 si le joueur est en vie, puis il correspondra au nombre de joueur vivants toujours en lice à la mort du *Warrior*).

boolean alive : Cet attribut permet de savoir si le joueur est en vie ou non.

String name : Cet attribut permet de stocker le nom du joueur, utile à l'extérieur du package *corewar.mars*.

Pour les méthodes de la classe :

Warrior(int id, String name, String[] program) : Ce constructeur initialise les attributs *id* et *name* aux valeurs passées en paramètre et le reste des attributs à leur valeur par défaut de début de partie. L'initialisation de l'attribut *program* est un peu spécifique car pour passer d'un tableau de chaîne de caractères à un tableau d'objets *Core*, il faut utiliser la méthode *compile(String[])* de la classe *Compiler*. Si la compilation échoue, l'exception levée par la compilation sera levée par le constructeur de la classe.

Cette classe possède des accesseurs en lecture pour chaque attribut de l'objet *Warrior*: **int getId()**, **int getPosition()**, **int getRank()**, **boolean isAlive()**, **String getName()** et **Core[] getProgram()**. Ainsi qu'un accesseur en écriture pour l'attribut position : **void setPosition(int position)**.

void die(int rank) : Cette méthode passe l'attribut *alive* à *false* et met à jour l'attribut *rank* avec la valeur passée en paramètre.

void next(int max) : Cette méthode permet de mettre à jour la position du joueur en incrémentant la valeur de l'attribut *position* de 1. Si la valeur de l'attribut *position* est supérieure à *max*, la valeur de l'attribut *position* est remise à 0.

void programFlush() : Cette méthode, appelée par *Mars* après avoir copiée en mémoire le programme du joueur, met à *null* l'attribut *program* afin que le ramasse-miette puisse libérer la mémoire occupée par ce tableau.

Mars.java (Thread)

Cette classe gère la simulation d'une partie de Core War. Elle hérite de la classe *Thread* afin de pouvoir lancer plusieurs instances en parallèle.

Elle possède deux constantes privées : **long MAX_CYCLE** (qui définit le nombre de cycle maximal d'une partie) et **int MIN_WARRIOR_SIZE** (qui définit la taille minimale dont un programme dispose en mémoire).

Elle possède aussi quatre attributs privés : **Warrior[] warriors** (un tableau contenant l'ensemble des joueurs de la partie en cours), **Core[] memory** (un tableau d'objets *Core* qui représente la mémoire de *Mars*), **long cycle** (un compteur qui compte le nombre de cycle écoulé depuis le début de la simulation), **boolean debug** (un booléen permettant de spécifier si on doit afficher l'état de la mémoire à chaque cycle).

Mars(Warrior[] warriors) : Ce constructeur appelle le second constructeur de la classe *Mars* en spécifiant la valeur du paramètre *debug* à *false*, ainsi, par défaut, on n'affichera pas l'état de la mémoire à chaque cycle. Si l'appel au second constructeur lève une exception, l'exception sera aussi levée par ce constructeur.

Mars(Warrior[] warriors, boolean debug) : Ce second constructeur initialise *Mars* à partir du tableau de *Warriors* passé en paramètre. Il initialise aussi l'attribut *debug* à la valeur passée en paramètre. Si le tableau de *Warriors* contient moins de deux *Warriors*, une exception sera levée en spécifiant qu'il n'est pas possible de jouer avec moins de deux joueurs. Le nombre de cycle écoulé est initialisé à 0, les *Warriors* sont placés dans l'attribut *warriors*, enfin on appelle la méthode *initMemory()*.

void execute(Warrior warrior) : Cette méthode réalise l'exécution d'une instruction du *Warrior* passé en paramètre si ce dernier est toujours « vivant ». Si l'attribut *debug* est à *true*, on affiche l'état de la mémoire avant l'exécution de l'instruction courante. Puis en fonction de l'instruction à la position pointée par le *Warrior*, on exécute les instructions correspondantes en fonction de l'*OpCode* de l'objet *Core* (par exemple, on fait « mourir » le *Warrior* si l'*OpCode* correspond à *DAT*). Enfin, on déplace le *Warrior* à la case mémoire suivante s'il n'a pas fait de saut.

boolean isGameOver() : Cette méthode vérifie qu'il reste au moins un *Warrior* en vie et que le nombre de cycle maximal n'a pas été atteint et retourne *false*, sinon, elle fait « mourir » le ou les *Warrior(s)* restant(s) avec un *rank* à 0 si le nombre de cycle maximal a été atteint, puis elle retourne *true*.

int countAliveWarriors() : Cette méthode sert juste à compter le nombre de *Warriors* toujours en vie et retourne ce nombre.

int getAddress(int position, AddressMode mode, int arg) : Cette méthode traduit l'adresse spécifiée en index dans la mémoire ou en quantité si le mode d'adressage est immédiat à partir de la position du *Warrior* courant, du mode d'adressage et de l'argument passé en paramètre. Si le mode d'adressage est immédiat, on retourne juste la valeur de l'argument, si le mode d'adressage est direct, on retourne la somme de la position du *Warrior* courant et de l'argument, sinon, si le mode d'adressage est indirect, on retourne la valeur de la case mémoire à laquelle pointée par la somme de la position du *Warrior* courant et de l'argument.

Core getCore(int address) : Cette méthode est un accesseur en lecture sur un élément de la mémoire à partir de son adresse.

int getIndex(int address) : Cette méthode permet de corriger l'adresse passée en paramètre en fonction de la taille de la mémoire afin d'éviter les débordements et faire boucler les adresses.

Warrior[] getWarriors() : Cette méthode est un accesseur en lecture sur l'attribut *warriors*.

void initMemory() : Cette méthode initialise l'état de la mémoire à partir des programmes compilés des *warriors* contenus dans leur attribut *program*. Tout d'abord, on initialise le tableau *memory* avec une taille définie par la taille du plus grand programme multipliée par deux (si le résultat est inférieur à *MIN_WARRIOR_SIZE*, alors ce sera la valeur de cette constante qui sera prise en compte) puis multipliée à nouveau par le nombre de *warriors* de la simulation. Ensuite, la mémoire est divisée en *n* parties (*n* étant le nombre de *warriors* de la simulation), pour chacune de ces parties, on copie le programme d'un *Warrior* au début de cette partie, et on positionne le *Warrior* au début de ce dernier, les espaces restants de la partie sont initialisés avec le constructeur par défaut de la classe *Core* (et on répète l'opération pour chaque partie de la mémoire et chaque *Warrior*).

void run() : Cette méthode contient la boucle principale de la simulation. Tant la simulation n'est pas terminée, on exécute tour à tour chaque *Warrior* puis on incrémente le compteur de cycle.

String toString() : Cette méthode retourne une représentation textuelle de l'état de la mémoire sur plusieurs lignes pour chaque objet *Core* du tableau *memory*, chaque ligne commençant par l'adresse de l'objet suivie par la représentation textuelle de ce dernier.

III.2. corewar.server

Server.java

Cette classe possède 4 attributs. Un attribut *ServerSocket socket* correspondant au socket du serveur, une *ArrayList* de *ClientHandler* (gestionnaires de client) qui servent à communiquer avec les différents clients. Une *ArrayList* de *Game* qui correspond à la liste de parties en cours sur le serveur. Le dernier attribut est une instance de *ClassementHandler* qui permet la mise à jour du fichier de classement.

Server(ServerSocket socket) : Le constructeur de cette classe initialise les différents attributs cités ci-dessus. Les *ArrayList* sont créées, l'attribut *socket* est initialisé avec le paramètre du constructeur, une instance de *ClassementHandler* est créée et son thread est lancé.

void start() : Boucle tant que le socket du serveur n'est pas fermé. Crée une nouvelle instance de *ClientHandler* pour chaque demande de connexion en lui passant l'instance du serveur et le socket permettant la communication avec le client. Ajoute cette instance à l'*ArrayList* de *ClientHandler* *clientHandlers* et démarre le thread lié à cette nouvelle instance.

void removeClientHandler(ClientHandler clientHandler) : Cette méthode supprime un gestionnaire de client passé en paramètre de l'*ArrayList* *clientHandlers* et de la partie à laquelle il est associé.

void removeGame(int gameId) : Supprime une partie de l'*ArrayList* *games* tout en réinitialisant les attributs *gameId* et *warriorId* des clients qui étaient dans la partie.

void echoRequest(String username, String request) : Affiche une requête client dans la console du serveur.

void echoResponse(String response) : Affiche une réponse serveur dans la console.

String response(ClientHandler clientHandler, String request) : Retourne la réponse du serveur en fonction de la requête passée en paramètre. Pour retourner la bonne réponse, le serveur exécute certaines de ses méthodes en fonction du code de la requête. Par exemple si ce code correspond à *API.SETUSERNAME* alors la méthode *setUsername(clientHandler, request)* sera exécutée. La méthode fait aussi appel aux méthodes *echoRequest(String username, String request)* et *echoResponse(String response)* qui permet d'afficher le contenu des requêtes et réponses dans la console du serveur.

String setUsername(ClientHandler clientHandler, String request) : Cette méthode récupère le pseudo saisi dans la requête et initialise le pseudo du client seulement si aucun autre client n'a ce pseudo. Si c'est le cas ou si le pseudo du client est déjà initialisé, la méthode retourne un code erreur.

String createGame(ClientHandler clientHandler, String request) : Créé une nouvelle partie en fonction des paramètres passés dans la requête. La méthode ajoute une nouvelle instance de *Game* dans l'*ArrayList* de *Game games*. Cette instance est créée en passant l'instance du serveur le *clientHandler* et le nombre de joueurs maximum pouvant rejoindre la partie *maxPlayers* qui a été transmis dans la requête. Cette nouvelle instance et son thread sont ajoutés au gestionnaire de classement. Pour finir, l'identifiant de la partie correspondant à l'index de l'instance dans *games* est retourné.

void addGame(int i, Game game) : Cette méthode est appelée par *createGame()* elle ajoute l'instance de partie à *games*, démarre le thread de cette instance et l'ajoute au gestionnaire de classement *classementHandler*.

String getGameList() : Construit et retourne un *String* correspondant aux parties que le client peut rejoindre. Le *String* retourné représente chaque partie par son index dans *games* correspondant à l'identifiant de partie ainsi que sa capacité « nombre de joueur en jeu / *maxPlayers* ». Si l'*ArrayList games* est vide la méthode retourne un code erreur.

String joinGame(ClientHandler clientHandler, String request) : Ajoute un client à une instance *Game* stockée dans l'*ArrayList games* en fonction de l'identifiant de partie passé en paramètre de la requête. Si le joueur a pu être ajouté à la partie, la méthode retourne un code valide. Dans le cas contraire elle retourne un code erreur.

String uploadWarrior(ClientHandler clientHandler, String request) : La méthode initialise un tableau de *String* correspondant aux instructions du warrior transmis via la requête, elle crée ensuite une nouvelle instance *Warrior* en passant en paramètre l'identifiant du client, le nom du warrior et le tableau de *String* initialisé plus haut. Pour finir, cette nouvelle instance est ajoutée à la partie dont l'identifiant correspond à celui passé dans la requête. Si le programme envoyé dans la requête est vide la méthode retourne un code erreur.

String getClassement() : Retourne un *String* représentant le classement actuel du serveur. Retourne un code erreur si le classement n'existe pas encore.

ClientHandler.java (thread)

Le gestionnaire de client à 7 attributs. Le premier (*Socket socket*) correspond au socket retourné par le serveur lors de la demande de connexion du client, ce socket est lié à celui-ci pour permettre la communication. L'attribut suivant est l'instance du serveur, pour permettre au gestionnaire de client d'appeler la méthode *response()* du serveur. Cette classe possède deux attributs, *BufferedReader in* pour la réception de données et *BufferedWriter out* pour l'envoi de données. Le gestionnaire de client possède trois autres attributs permettant de stocker des données du client côté serveur. Un *String clientUsername* pour stocker le pseudo du client, un *int gameId* correspondant à l'identifiant de la partie dans laquelle est le client et un *int warriorId* correspondant à l'identifiant du warrior du client dans une partie.

ClientHandler(Socket socket, Server server) : Initialise les attributs *socket*, *server*, *in* et *out* de l'instance. Appelle la méthode *close()* si le constructeur lève une *IOException*.

void run() : Cette méthode boucle tant que le socket du gestionnaire de client est connecté. Une requête est attendue avec la méthode *in.readLine()* si c'est une requête de type *API.ENDCONNECTION*, la méthode *close()* est appelée. Dans le cas contraire, une réponse du serveur est attendue avec la méthode *server.response()*. Pour finir, la réponse du serveur est envoyée avec la méthode *send()*. Si une exception est levée, la méthode *close()* est appelée.

void send(String response) : Envoi des données à l'aide du *BufferedWriter out*. Les méthode *write()*, *newLine()* et *flush()* sont appelée à la suite pour envoyer la donnée. Si une *IOException* est levée, la méthode *close()* est exécutée.

void close() : Ferme proprement l'instance. Un message de déconnection est affiché côté serveur. La méthode *server.removeClientHandler()* est appelée. Le *BufferedReader* de l'instance est fermé, tout comme le *BufferedWriter* et le *Socket*.

Game.java (thread)

Cette classe dispose de 5 attributs. Le premier est l'instance du serveur pour permettre l'accès de la méthode *removeGame()* afin de supprimer la partie de la liste de parties en cours du serveur à la fin de celle-ci. Le second attribut est une instance de la classe *Mars*. Cette classe dispose aussi d'une *ArrayList* de *ClientHandler* qui correspond aux joueurs dans la partie ainsi que d'un *int maxPlayers* permettant de connaître le nombre de joueurs maximum pouvant rejoindre la partie. Le dernier attribut important de cette classe est un tableau de *Warrior*.

Game(Server server, ClientHandler clientHandler, int maxPlayers) : Les différents attributs de l'instance sont initialisés. Le *ClientHandler* passé en paramètre est ajouté à l'attribut *clientHandlers*. La taille du tableau *warriors* est initialisé avec le paramètre *maxPlayers* et chaque case est initialisée à *null*.

void run() : Tant que la partie n'est pas complète et que tous les warriors ne sont pas uploadés, le thread est mis en attente 1 seconde. Si la partie est complète et que tous les warriors ont été uploadés, le simulateur MARS est initialisé avec le tableau de warriors et son thread est démarré. On attend la fin de son exécution avec *mars.join()*. Une fois la simulation terminée, la méthode *setClassement()* et la méthode *sendClassement()* sont appelées. Pour finir, l'instance appelle la méthode *server.removeGame()* pour se supprimer de la liste des parties en cours.

String[] getClassement() : Cette méthode permet de récupérer un classement de partie représenté par une liste de nom de warriors, classés du premier au dernier.

void setClassement() : Classe le tableau *warriors* du premier au dernier en fonction de leur rang à la fin de la partie.

void sendClassement() : Permet d'envoyer le classement de fin de partie à chaque joueurs ayant participé. Le classement est représenté par un couple joueur/warrior classé du premier au dernier.

Warrior[] initWarriors() : Crée un tableau de *Warrior* initialisé à *null* et de taille *maxPlayers*.

boolean isFull() : Retourne *true* si la partie est complète.

boolean isAllWarriorUploaded() : Retourne *true* si tous les warriors ont été initialisés.

int getClientId(String username) : Retourne l'index d'un *ClientHandler* dans l'*ArrayList clientHandlers* en fonction de son pseudo.

synchronized void addWarrior(ClientHandler clientHandler, Warrior warrior) : Ajoute un nouveau *Warrior* au tableau *warriors*.

synchronized void addClient(ClientHandler clientHandler) : Ajoute un nouveau *ClientHandler* à l'*ArrayList clientHandlers*.

synchronized void removeClient(ClientHandler clientHandler) : Permet la suppression d'un client de la partie de façon sécurisée.

String toString() : Retourne la capacité de la partie « nombre de joueurs / maximum de joueurs ».

ClassementHandler.java (thread)

Le gestionnaire de classement a 4 attributs. Un *String* permettant de stocker en mémoire le chemin du fichier de classement. Une *ArrayList* de *Game* servant à renseigner les parties en cours ainsi qu'une *ArrayList* de *Thread* correspondant aux threads associés à chaque partie de cette liste. Et pour finir une *ArrayList* de *String* correspondant à la représentation en mémoire du classement actuel.

ClassementHandler(String path) : Initialise les attributs de cette classe. Vérifie l'existence du fichier dont le chemin est passé en paramètre. Crée le fichier s'il n'existe pas et initialise l'*ArrayList classement* avec son contenu.

void run() : Tant que le thread est en cours, le thread se met en attente pendant 1 seconde. Ensuite il parcourt l'*ArrayList games* correspondant aux parties en cours. Si le thread d'une partie est mort, le gestionnaire de classement récupère le classement de la partie avec la méthode *getClassement()* et met à jour le fichier de classement avec la méthode *updateClassement()*.

void addGame(Game game, Thread thread) : Ajoute une instance de la classe *Game* à l'*ArrayList games* ainsi que son thread dans l'*ArrayList threads*.

void updateClassement(String[] gameClassement) : Cette méthode sépare l'*ArrayList classement* en deux autres *ArrayList* une pour les noms des warrior et une autre pour le nombre de victoires des warriors. Elle parcourt ensuite le classement de la partie. Pour chaque nom de warrior correspondant, si il n'est pas dernier, le nombre de victoire de ce warrior est incrémenté (on considère que ne pas être dernier compte pour une victoire). Si le nom du warrior ne correspond pas, le nom est ajouté et si il n'est pas dernier on lui ajoute une victoire, sinon zéro. Ensuite les listes de noms et de victoires sont triées par ordre croissant. Pour finir, les données sont formatées et écrites dans le fichier de classement.

String getClassement() : Retourne le classement actuel dans une chaîne de caractère.

Main.java

static void main(String[] args) : Initialisation du serveur et de son socket et lancement de la méthode *start()*.

III.3. corewar.client

Client.java

Cette classe possède six attributs. Un *Socket socket* qui sert à se connecter avec un gestionnaire de client pour pouvoir communiquer avec le serveur. Un *BufferedReader in* et un *BufferedWriter out* qui permettront respectivement, l'envoi et la réception de données. Le dernier attribut est un *int gameId* qui correspond à l'id de la partie du joueur. Un attribut *String username* pour stocker le pseudo du client et un *int gameId* qui représente l'identifiant de la partie du joueur.

Client(Socket socket) : Le constructeur initialise les attributs *socket*, *in* et *out* de la classe. Si une *IOException* est levée, la méthode *close()* est appelée.

String request(String request) : Cette méthode permet au client d'envoyer une requête, de recevoir une réponse et de la retourner. Si la requête est de type *API.WAITMSG* l'envoi de la requête est ignoré et la méthode attend directement une réponse. Si une *IOException* est levée, la méthode *close()* est appelée.

void close() : Ferme les attributs *in*, *out* et le socket du client.

UI.java

static void printLogo() : Affiche le logo du jeu.

static void reset() : Permet d'effacer la console de l'utilisateur qu'il soit sur Windows ou Linux.

static String serverConnexion() : Affiche le menu de connexion au serveur. Demande un choix à l'utilisateur entre se connecter et quitter, puis lui demande l'ip du serveur s'il veut se connecter.

static String usernameChoice(String msg) : Demande à l'utilisateur de choisir un pseudo, le paramètre *String msg* sert à afficher un message personnalisé en cas d'erreur.

static String gameMenu() : Affiche le menu principal du jeu et demande à l'utilisateur de sélectionner un choix. Il a la possibilité de créer une partie, en rejoindre une, consulter le classement du serveur ou quitter le jeu.

static int numberOfPlayers() : Demande à l'utilisateur de sélectionner un nombre de joueur (lors de la création d'une nouvelle partie) .

static int gameId(ArrayList<Integer> ids) : Attend que l'utilisateur sélectionne un identifiant de partie valide (contenu dans le paramètre *ids*) lorsqu'il veut rejoindre une partie.

static void waiting() : Attend une entrée utilisateur quelconque afin de mettre l'affichage en pause.

static String selectWarrior() : Demande à l'utilisateur de rentrer le nom du warrior qu'il veut uploader tant que le fichier n'existe pas.

Main.java

static void main(String[] args) : Cette méthode gère le déroulement d'une session côté client. La méthode possède 4 variables importantes. Une variable *Socket socket*, pour l'initialisation du client, une variable *Client client* pour le stocker et deux *String* pour récupérer les inputs utilisateurs et les réponses serveur.

Tant que le socket n'est pas connecté, une connexion utilisateur est demandée avec la méthode *UI.serverConnection()*. Le client est ensuite initialisé avec ce socket. Un pseudo est demandé au client tant que le serveur ne retourne pas une réponse valide. Le pseudo du client est initialisé avec le pseudo choisi. On entre ensuite dans une boucle infinie, une entrée utilisateur est demandée avec la méthode *UI.gameMenu()* si l'attribut *client.gameld* vaut -1.

Si l'utilisateur veut créer une nouvelle partie, on lui demande le nombre de joueurs maximum avec *UI.numberOfPlayers()*, une requête de nouvelle partie est construite avec ce paramètre et envoyée au serveur. Le serveur envoie une réponse avec l'identifiant de la partie créée, cet identifiant est stocké dans l'attribut *client.username*.

Si l'utilisateur décide de rejoindre une partie. Une requête est envoyée au serveur pour récupérer la liste de parties disponibles et cette liste est affichée pour permettre à l'utilisateur de choisir. On lui demande ensuite l'identifiant de la partie qu'il veut rejoindre avec *UI.joinGameRequest()*, l'identifiant de la partie est stocké dans l'attribut *client.gameld*.

Dans le cas d'une demande de classement. Une requête de demande de classement est envoyée au serveur. Si la réponse de celui-ci est valide, le classement est affiché sur l'interface de l'utilisateur, sinon, un message l'informe que le classement est « indisponible ».

Si le client veut quitter le jeu, une requête de fin de connexion est envoyée, cette requête est interceptée par l'instance *ClientHandler* associée au client et comme décrit précédemment, la méthode *close()* du gestionnaire de client est appelée. Enfin, la méthode *close()* du client est appelée et la méthode *exit()* de la classe *System* est exécutée.

Si l'attribut *client.gameld* est différent de -1 et donc que l'utilisateur est dans une partie. On demande au client de saisir le chemin du warrior qu'il souhaite uploader. Le fichier du warrior en question est lu avec *FileOperation.read()* et transmis au serveur via une requête client. Le client envoie ensuite une requête *API.WAITMSG* et attend une réponse. La réponse reçue correspond au classement de fin de partie envoyé par l'instance *Game* liée au client. Ce classement est affiché afin que le client puisse voir le résultat de la partie et l'attribut *client.gameld* est réinitialisé à sa valeur par défaut.

III.4. corewar.utils

API.java

Cette classe possède 12 attributs *final* et *static* ce sont tous des *String*. Ces attributs permettent de coder les différentes requêtes et réponses possibles.

static String[] apiCallToArray(String apiCall) : Retourne un tableau de *String* représentant chaque paramètre d'une requête.

static String getCallType(String apiCall) : Retourne le type de la requête.

static boolean isValidResponse(String response) : Renvoie *true* si la réponse est valide, *false* dans le cas contraire.

static String setUsernameRequest(String username) : Construit une requête de nouveau pseudo.

static String uploadWarriorRequest(ArrayList<String> program) : Construit une requête composée du nom du warrior et de ses instructions.

static String createGameRequest(int maxPlayers) : Construit une requête composée du nombre maximum de joueur de la partie à créer.

static String joinGameRequest(int gameld) : Construit une requête avec comme paramètre, l'identifiant de la partie à rejoindre.

static String getGameListRequest() : Construit une requête de récupération de la liste des parties.

static String waitMsgRequest() : Construit une requête d'attente de message.

static String endConnectionRequest() : Construit une requête fin de connexion.

static String getClassementRequest() : Construit une requête de récupération de classement.

FileOperation.java

static ArrayList<String> read(String path) : Construit et retourne une *ArrayList* constituée du nom du fichier lu et de chaque ligne de ce fichier.

static void write(String path, ArrayList<String> content) : Ecrit dans le fichier indiqué par le paramètre *path*. Le contenu est représenté par l'*ArrayList* passée en paramètre où chaque cellule correspond à une ligne à écrire dans le fichier.

static void create(String path) : Créé un nouveau fichier à l'emplacement indiqué par le chemin *path* passé en paramètre.

IV. Présentation des warriors

dwarf

Dwarf est un programme en redcode capable de "bombarder" la mémoire de "bombes zéro" (des instructions *DAT 0*). Pour cela, il exécute une boucle qui incrémente son compteur de 5 et pose une instruction *DAT 0* à l'adresse pointée par le compteur.

gemini

Gemini est un programme redcode capable de se multiplier en mémoire. Pour cela, il copie juste ses instructions à un autre emplacement de la mémoire puis se déplace sur cette nouvelle copie.

imp

Imp est un warrior composé d'une seule instruction : *MOV 0 1*. Il ne fait que copier son instruction dans la case mémoire suivante puis se rend sur cette case. Si un autre warrior arrive sur une instruction *MOV 0 1* laissée dans le sillage de imp, il devient aussi un programme imp.

snail

Snail est un programme redcode capable de se déplacer en mémoire comme imp. Cependant, puisqu'il est plus conséquent que imp, il se déplace plus lentement que ce dernier.

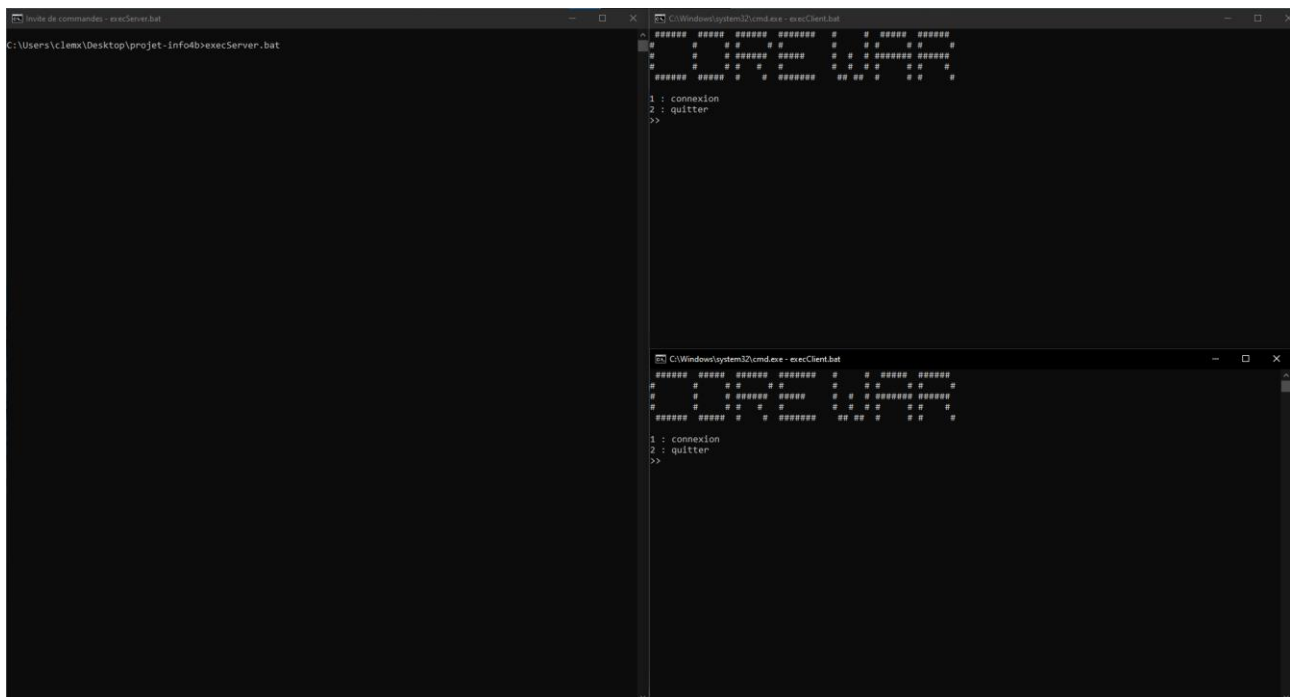
Conclusion

En conclusion, ce projet aura été le projet scolaire le plus long et le plus complet que l'on a réalisé depuis notre admission en licence. Il nous a permis de pleinement nous confronter à un cahier des charges suffisamment conséquent pour mettre à l'épreuve nos connaissances du module d'Info4B, notre organisation et notre collaboration. De plus, les notions abordées dans ce projet, et plus largement ce module, nous ont permis d'atteindre un bon niveau et une certaine polyvalence en programmation.

Annexes

Annexe 1 : Jeux d'essai

Lancement du serveur et des clients



The screenshot shows two command prompt windows. The left window, titled 'invite de commandes - execServer.bat', displays the command 'c:\Users\ciemx\Desktop\projet-info4b\execServer.bat'. The right window, titled 'C:\Windows\system32\cmd.exe - execClient.bat', displays a menu with a decorative border of asterisks. The menu options are '1 : connexion' and '2 : quitter', followed by a prompt '>'. Below this, a second identical window is shown, also displaying the same menu options and prompt.

Connexion des clients au serveur



The screenshot shows two command prompt windows. The left window, titled 'invite de commandes - execServer.bat', displays the command 'c:\Users\ciemx\Desktop\projet-info4b\execServer.bat'. The right window, titled 'C:\Windows\system32\cmd.exe - execClient.bat', displays a menu with a decorative border of asterisks. The menu options are '1 : connexion' and '2 : quitter', followed by a prompt '>'. Below this, a second identical window is shown, also displaying the same menu options and prompt.

Choix des pseudos

The screenshot displays three command prompt windows on a Windows desktop.

- Top-left window:** Titled "invite de commandes - execServer.bat". It shows the following output:


```
C:\Users\clem\Desktop\projet-Info4b>execServer.bat
Nouvelle connexion sur le port : 1234
Nouvelle connexion sur le port : 1234
unknown a choisi le pseudo clem -> OK
unknown a choisi le pseudo clem -> ERR
```
- Top-right window:** Titled "C:\Windows\system32\cmd.exe - execClient.bat". It displays a menu of options and a prompt for a pseudo:


```

      #####  #####  #####  #####  #####
      #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
      #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
      #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
      #####  #####  #####  #####  #####

      Pseudo invalide ou deja pris
      Pseudo >> _
```
- Bottom window:** Titled "C:\Windows\system32\cmd.exe - execClient.bat". It shows the same menu as the top-right window:


```

      #####  #####  #####  #####  #####
      #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
      #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
      #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #
      #####  #####  #####  #####  #####

      1 : creer une nouvelle partie
      2 : rejoindre une partie
      3 : afficher le classement
      4 : quitter
      >>
```

Créer et rejoindre une partie

The screenshot displays three command prompt windows on a Windows desktop. The top-left window, titled "Invite de commandes - exeServer.bat", shows the execution of "exeServer.bat" which starts a server on port 1234. The top-right window, titled "C:\Windows\system32\cmd.exe - exeClient.bat", shows the execution of "exeClient.bat" which connects to the server. The bottom window, also titled "C:\Windows\system32\cmd.exe - exeClient.bat", shows the same client execution but with additional output: "Identifiant de partie : 0", "Nombre de joueurs : 1/2", and "Identifiant de partie" followed by a prompt ">>".

Choix des warriors

```

C:\Users\clem\Desktop\projet-info4b>execServer.bat
Nouvelle connexion sur le port : 1234
Nouvelle connexion sur le port : 1234
unknown a choisi le pseudo clem -> OK
unknown a choisi le pseudo clem -> ERR
unknown a choisi le pseudo guillaume -> OK
guillaume veut creer une nouvelle partie -> OK
clem demande la liste des parties -> OK
clem veut rejoindre la partie 0 -> OK

C:\Windows\system32\cmd.exe - execClient.bat
#####
# # # # #
# # # # #
# # # # #
#####

//\ Votre warrior doit etre dans le dossier du meme nom a la racine du jeu /\
Exemple : dwarf.red

Nom du warrior >> dwarf.red

C:\Windows\system32\cmd.exe - execClient.bat
#####
# # # # #
# # # # #
# # # # #
#####

//\ Votre warrior doit etre dans le dossier du meme nom a la racine du jeu /\
Exemple : dwarf.red

Nom du warrior >> imp.red

```

Fin d'une partie et classement de la partie

```

C:\Users\clem\Desktop\projet-info4b>execServer.bat
Nouvelle connexion sur le port : 1234
Nouvelle connexion sur le port : 1234
unknown a choisi le pseudo clem -> OK
unknown a choisi le pseudo clem -> ERR
unknown a choisi le pseudo guillaume -> OK
guillaume veut creer une nouvelle partie -> OK
clem demande la liste des parties -> OK
clem veut rejoindre la partie 0 -> OK
clem upload son warrior -> OK
guillaume upload son warrior -> OK
la partie 0 a commence !
la partie 0 est terminee !
mise a jour du classement !

C:\Windows\system32\cmd.exe - execClient.bat
#####
# # # # #
# # # # #
# # # # #
#####

Fin de la partie !

1 - clem (imp.red)
2 - guillaume (dwarf.red)

Appuyez sur [Entree] pour retourner au menu

C:\Windows\system32\cmd.exe - execClient.bat
#####
# # # # #
# # # # #
# # # # #
#####

Fin de la partie !

1 - clem (imp.red)
2 - guillaume (dwarf.red)

Appuyez sur [Entree] pour retourner au menu

```

Affichage du classement du serveur

```
C:\Users\clema\Desktop\projet-info4b>execServer.bat
Nouvelle connexion sur le port : 1234
Nouvelle connexion sur le port : 1234
unknown a choisi le pseudo clem -> OK
unknown a choisi le pseudo guillaume -> OK
guillaume veut creer une nouvelle partie -> OK
clem demande la liste des parties -> OK
clem veut rejoindre la partie 0 -> OK
clem upload son warrior -> OK
guillaume upload son warrior -> OK
la partie 0 a commence !
la partie 0 est terminee !
mise a jour du classement !
clem demande le classement du serveur -> OK
```

```
C:\Windows\system32\cmd.exe - execClient.bat

#####  #####  #####  #  #####  #####
#  #  #  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #####  #####  #  #  #####  #####
#  #  #  #  #  #  #  #  #  #  #  #  #
#####  #####  #  #####  ##  #  #  #  #

1 : creer une nouvelle partie
2 : rejoindre une partie
3 : afficher le classement
4 : quitter
>>
```

```
C:\Windows\system32\cmd.exe - execClient.bat

#####  #####  #####  #  #####  #####
#  #  #  #  #  #  #  #  #  #  #  #  #
#  #  #  #####  #####  #  #  #####  #####
#  #  #  #  #  #  #  #  #  #  #  #  #
#####  #  #  #####  ##  #  #  #  #

1 - imp.red | score : 6
2 - dwarf.red | score : 3
3 - gemini.red | score : 2
4 - snail.red | score : 0

Appuyez sur [Entree] pour retourner au menu
```

Nouvelles connexions

The screenshot displays four terminal windows on a Windows desktop, illustrating a client-server application. The top-left window, titled "invite de commandes - execServer.bat", shows a chat log with messages from "unknowm" and "guillaume". The top-right window, titled "C:\Windows\system32\cmd.exe - execClient.bat", shows a menu with options 1 (create new party), 2 (rejoin party), 3 (display ranking), and 4 (quit), followed by a grid of characters. The bottom-left window, titled "invite de commandes - execClient.bat", shows the same menu. The bottom-right window, titled "C:\Windows\system32\cmd.exe - execClient.bat", shows the menu and a grid of characters, similar to the top-right window.

Multi-parties

The image displays three terminal windows from a Windows operating system, showing the execution of a custom networked game.

- Top Window (Server):** The title bar is "C:\Windows\system32\cmd.exe - execServer.bat". The output shows the server starting on port 1234, receiving connections from 'unknown' and 'guillaume', and handling a 'warrior' upload. It also shows a game round starting and ending.
- Bottom-Left Window (Client):** The title bar is "Invite de commandes - execClient.bat". It shows the client's input for the number of parties (0) and players (1/2).
- Bottom-Right Window (Client):** The title bar is "C:\Windows\system32\cmd.exe - execClient.bat". It shows the client's output, including a ASCII art warrior, instructions to place the warrior in a specific directory, and the number of warriors (2).

Choix des warriors

[illegible]

Résultat des parties

The image displays three sequential screenshots of a Windows command prompt window, showing the execution of a client program named 'clemClient.bat'.

First Screenshot (Top): The window title is 'Invité de commandes - clemClient.bat'. The output shows the client connecting to a server, receiving a list of players (guillaume, pilou, azerty), and starting a game. The game is a 'warrior' game where the client uploads a warrior. The game ends with a list of winners: guillaume (imp.red), pilou (gemin1.red), and azerty (dwarf.red).

Second Screenshot (Middle): The window title is 'C:\Windows\system32\cmd.exe - clemClient.bat'. The output shows the client connecting to a server, receiving a list of players (guillaume, pilou, azerty), and starting a game. The game is a 'warrior' game where the client uploads a warrior. The game ends with a list of winners: guillaume (imp.red), pilou (gemin1.red), and azerty (dwarf.red).

Third Screenshot (Bottom): The window title is 'C:\Windows\system32\cmd.exe - clemClient.bat'. The output shows the client connecting to a server, receiving a list of players (guillaume, pilou, azerty), and starting a game. The game is a 'warrior' game where the client uploads a warrior. The game ends with a list of winners: guillaume (imp.red), pilou (gemin1.red), and azerty (dwarf.red).