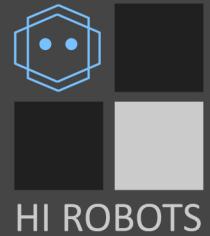


# RAPPORT DE PROJET

## ROBOTS ECRASEURS

2019  
2020



### GROUPE DE PROJET :

KOLOKOLNIKOVA Veronika  
LUTTENSCHLAGER Mélissa  
MOINS Solène  
NDAW Yaya

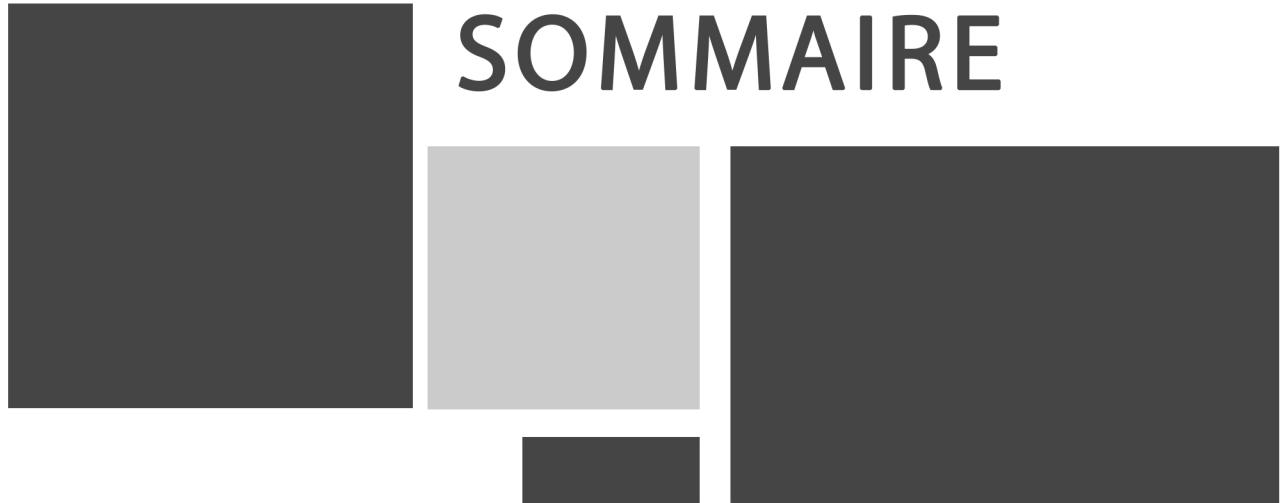
3EME ANNEE  
Licence MIAGE

# REMERCIEMENTS



Nous tenons à remercier M. Stéphane RIVIERE pour l'aide et le suivi apportés durant ce projet, mais également l'ensemble de l'Université de Haute-Alsace qui nous a permis d'étudier dans notre formation et donc d'effectuer ce projet.

# SOMMAIRE



<b>INTRODUCTION .....</b>	<b>page 01</b>
<b>I. LE PROJET .....</b>	<b>page 02</b>
1. Description .....	page 02
2. Répartition .....	page 02
<b>II. OUTILS UTILISÉS .....</b>	<b>page 03</b>
1. Gestion .....	page 03
2. Développement .....	page 04
<b>III. DÉROULEMENT .....</b>	<b>page 05</b>
1. Gestion .....	page 05
2. Développement .....	page 06
3. Interface .....	page 12
<b>IV. PROBLEMES .....</b>	<b>page 14</b>
1. Résolus .....	page 14
2. Non résolus .....	page 14
<b>CONCLUSION .....</b>	<b>page 15</b>
1. Générale .....	page 15
2. Personnelles .....	page 15
<b>TABLE DES MATIERES .....</b>	<b>page 17</b>

# INTRODUCTION



L'Université de Haute-Alsace est une université française, dont les cinq campus sont situés à Mulhouse et à Colmar. Elle compte deux IUT, l'un à Colmar et l'autre à Mulhouse, deux écoles d'ingénieurs: l'ENSCMu et l'ENSISA, à Mulhouse, ainsi que quatre facultés: la FLSH, la FSESJ et la FST à Mulhouse ainsi que la FMA à Colmar.

La Faculté des Sciences et Techniques de Mulhouse offre un grand nombre de formations dont une Licence en Informatique. Cette licence propose une formation universitaire formant à une bonne culture en informatique, en mathématiques et possédant des compétences en méthodologie, communication et langues étrangères. Notre parcours MIAGE, proposent une formation de haut niveau en informatique, permettant à un étudiant diplômé soit de poursuivre ses études en Master, soit de s'intégrer dans la vie active.

C'est dans ce contexte que ce projet est né. Il consiste à développer un jeu de robots écraseurs s'appuyant sur la programmation orientée objet.

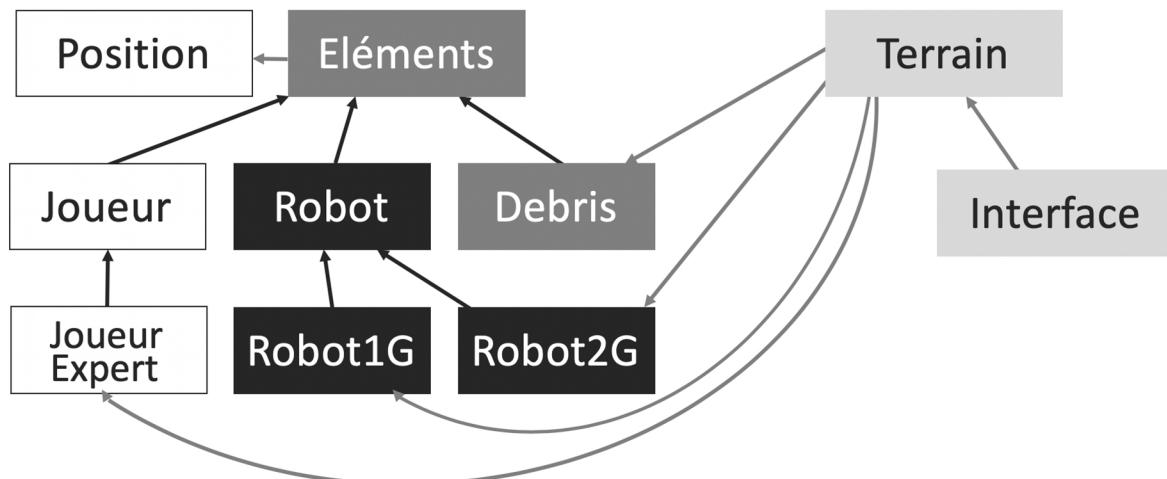
Et c'est donc sur ce sujet que nous pouvons amener la problématique de notre rapport : Comment mettre en place l'architecture du jeu de manière optimisée ? Pour répondre à cette question nous allons d'abord rappeler plus en détail le sujet de ce projet, ensuite nous présenterons tout les outils qui ont été utilisés, on poursuivra en expliquant les détails de la réalisation, et enfin nous consacrerons une partie aux problèmes que nous avons rencontrés et comment nous les avons résolus.

# LE PROJET

## DESCRIPTION

Le projet consiste à créer un jeu où un joueur affronte des robots. Le but du joueur est de gagner en faisant s'entrechoquer les robots et le but des robots est d'écraser le joueur. Il nous est demandé de créer une interface de notre choix, ainsi que des méthodes pour sauvegarder et charger une partie.

## REPARTITION



Yaya

Véronika

Solène

Mélissa

# OUTILS UTILISÉS



GESTION



## Git

Le principal outil de gestion que nous avons utilisé est Git. Il s'agit d'un logiciel de gestion de versions décentralisé. Il nous a permis de partager le code de chacun et de revenir sur nos précédentes versions lorsque nous avions un problème dans notre code.



## Whatsapp



Nous avons également utilisé une messagerie instantanée : Whatsapp, qui nous a permis de communiquer tout au long de notre projet, mais également de nous envoyer des documents.

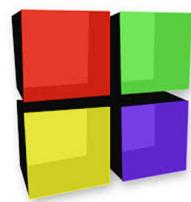


DEVELOPPEMENT



## Code Blocks

Environnement de développement, nous l'avons principalement utilisé pour les tests, mais également utilisé par Yaya pour développer.



## Qt Creator



Principal environnement de développement utilisé pour développer le jeu. Utilisable autant sur Mac OS que sur Windows, il nous a également permis de créer une interface graphique intéressante. Nous avons donc choisi de l'utiliser à la place de la bibliothèque graphique winbgi.

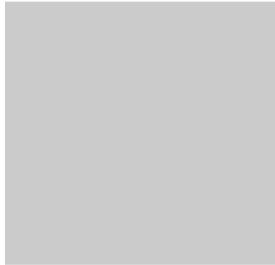
## Photoshop

Logiciel graphique qui nous a permis de concevoir les images intégrées au jeu, ainsi que le logo.



# DÉROULEMENT

GESTION



## Solène



Nous avons créé un projet sur Github afin d'avoir un outil de versioning entre les différents membres de l'équipe. Dès lors nous avons tiré une branche « dev » qui était la branche de tests (ou de pallier) avant de tout mettre sur la branche « master » une fois une version fonctionnelle. Une seule personne devait fusionner sur « dev » mais sur GitHub pour avoir cette option il faut payer donc on a décidé que seulement moi le ferais.

Chacun créait sa branche avec en intitulé ce qu'il allait faire comme tâche par exemple « joueur ». Ensuite, on faisait une consignation avec un message plus détaillé de ce qui avait été fait. Puis on poussait notre travail local. Une fois qu'ils avaient poussé, je me mettais sur une autre branche à partir de « dev » et tirais leur travail dessus. Je faisais des tests pour vérifier que les modifications qui étaient apportées ne contenaient ni d'erreurs, ni de conflits ni de bugs sous Qt.

Une fois tout en ordre, je faisais une « requête de fusion » (= merge request) et puis je me replaçais sur « dev » d'où je tirais à nouveau les modifications faites. Dès que l'un d'entre nous devait réaliser une tâche il récupérait le travail fonctionnel de la personne précédente sur « dev » à partir de laquelle il créait sa nouvelle branche. On a donc pu faire différentes parties notamment les tests des robots et des joueurs simultanément sans avoir de conflit tandis qu'à d'autres moments on devait attendre le travail de l'un des membres par exemple Mélissa qui a fait le terrain devait attendre toutes les classes d'éléments (robots, joueur) et la classe position pour avoir des éléments à positionner sur la grille.

DEVELOPPEMENT



# Véronika

J'ai d'abord commencé par créer les robots de bases en écrivant le constructeur et le destructeur. J'ai codé les déplacements du robot2 en utilisant la classe position. Le groupe s'est rendu compte qu'il y a des méthodes communes entre les classes robot, joueur et débris donc nous avons décidé d'implémenter une classe élément.

```
robot::robot(position*p){_case(p);}  
position* robot::positionRobot() const  
{  
    return _case;  
}  
//detecte si le joueur est present autour de lui  
bool robot::detecterJoueur( const joueur&j) const  
{  
    if(_case == j)  
        return true;  
    else  
        return false;  
}  
void robot::deplacerRobotAuHasard()  
{  
    _caseNew = position(_caseHasard);  
    if(_case == j)  
        // si le robot a détecté un joueur il se déplace sur la case  
        // voisine il se déplace sur une case au hasard se trouvant à côté  
        void robot::deplacerRobot(const joueur&j)  
    {  
        if (detecterJoueur(j))  
        {  
            _caseNew = position(j.positionJoueur);  
        }  
        else  
        {  
            deplacerRobotAuHasard();  
        }  
    }  
}
```

```
2 contributors 23 23%  
16 lines (13 sloc) | 287 Bytes  
Raw  
1 #ifndef ROBOT_H  
2 #define ROBOT_H  
3 #include "joueur.h"  
4  
5 class robot : public element {  
public :  
    robot();  
    robot(position*p);  
    virtual ~robot();  
    virtual void deplacerRobot(const joueur&j) = 0;  
    virtual int type() const;  
    void affichePosition() const;  
13  
15 };  
16 #endif // ROBOT_H
```

Pour le robot1G j'ai calculé la valeur absolue de la distance horizontale et de la distance verticale qui le sépare du joueur. Si la distance verticale est inférieure à l'horizontale le robot se déplacera sur les colonnes sinon sur les lignes. On teste si la distance est négative ou positive puis on déplace le robot.

Le but des robots est de se rapprocher et détecter le joueur.

Le robot2G peut se déplacer sur les diagonales donc il suffit de tester la position du joueur par rapport au robot et le déplacer en haut ou en bas et/ou à droite ou à gauche en fonction des cas.

```
d robot1G::deplacerRobot(const joueur&j){  
int distanceColonne,distanceLigne,distanceAbsolueColonne,distanceAbsolueLigne;  
distanceColonne = positionElement()->numColonne()-j.positionElement()->numColonne();  
distanceLigne = positionElement()->numLigne()-j.positionElement()->numLigne();  
distanceAbsolueColonne = fabs(positionElement()->numColonne()-j.positionElement()->numColonne());  
distanceAbsolueLigne = fabs(positionElement()->numLigne()-j.positionElement()->numLigne());
```

Puis j'ai créé les fonctions Collision qui retourne si le robot se trouve sur la même case qu'un robot, qu'un débris ou que le joueur.

Enfin j'ai fait des tests sur les déplacements des robots en fonction de sa position par rapport au joueur et des tests sur les collisions d'un robot avec les différents éléments.

– remaniements effectués pour pouvoir mieux progresser

Dans un premier temps j'ai voulu déplacer les robots en utilisant la classe position et la classe joueur. Puis j'ai utilisé la classe élément créer par Solène afin de déplacer les robots. J'ai remanié la fonction DeplacerRobot dans la classe du robot1 afin de faire moins de tests et que le code soit lisible. Dans un premier temps je calculais à chaque fois la position du robot par rapport au joueur puis j'ai décidé de faire les calculs au début de la fonction et de faire les tests après.



J'ai commencé par coder une classe terrain qui se construisait avec un joueur (mais la classe joueur était pour l'instant vide), ainsi qu'un nombre de lignes, de colonnes, de robots 1G, de robots 2G et enfin de débris. Mais je l'ai également représenté par une grille (un vecteur à deux dimension) afin de pouvoir le modéliser. J'y ai rajouté les méthodes de base : renvoie les différents nombres ainsi que le joueur et la grille, les méthodes pour lire et sauvegarder le terrain, ainsi que des méthodes pour changer les valeurs.

```
...
1|0|0|0|2|4|0|4|0|
0|0|0|0|0|0|0|4|2|
0|4|0|3|0|0|0|0|0|
0|0|0|0|0|0|0|0|3|
0|0|4|4|0|0|0|0|0|
0|0|0|0|0|4|0|0|0|
0|0|0|0|0|2|0|0|0|
0|0|0|0|0|4|4|3|4|
```

Après cette étape, la classe joueur ainsi que la classe position ont été ajoutée par Yaya. J'ai donc modifié la classe terrain pour qu'elle utilise ces deux classes. J'ai rajouté une méthode position qui renvoyait la position du joueur, j'en avais besoin spécifiquement parce que la position du joueur étant un pointeur, je ne pouvais pas récupérer cette position dans les classes de l'interface. J'ai également ajouté une méthode changePosJoueur qui permettait de mettre à jour la position du joueur dans la grille.

```
private :
    /// le joueur
    joueur d_joueur;
    /// le nombre de ligne
    int d_nbligne;
    /// le nombre de colonne
    int d_nbcollonne;
    /// le vecteur de robots
    std::vector<robot*> d_robot;
    /// le vecteur de débris
    std::vector<debris*> d_debris;
```

```
class terrain
{
    friend class fenetre;
public:
    terrain(); // terrain vide
    terrain(int nbdebris, int nbrobotfirstG, int nbrobotsecondG, int nbligne, int nbcollone, joueur *j);
    int nbColone() const;
    int nbLigne() const;
    int nbDebris() const;
    int nbrobot1G() const;
    int nbrobot2G() const;
    joueur *joueur();
    std::vector<std::vector<int>> grille() const;
    void lireTerrain(const std::string nomFichier); // charge un terrain
    void sauvegardeTerrain(const std::string nomFichier); // sauvegarde le terrain
    void changerTailleGrille(int nbligne, int nbcollone); // change la taille de la grille
    void changeNb(int nbdebris, int nbrobot1G, int nbrobot2G); // change le nombre de débris et de robot
    void InitialisationGrille(int nbdebris, int nbrobot1G, int nbrobot2G); // initialise la grille de maniere a
    void ChangeJoueur(joueur *&j);
    bool joueurPerdu();
    bool terrainOK();

    void afficheGrilleO(); // fonction test affichage
    void affichePositionJoueurO(); //fonction test affichage

private :
    int d_nbdebris;
    joueur *d_joueur;
    int d_nbDebris;
    int d_nbrobotFirstG;
    int d_nbrobotSecondG;
    int d_nbligne;
    int d_nbcollone;

    std::vector<std::vector<int>> d_grille; // 0 = vide, 1 = joueur, 2 = robot1G, 3 = robot2G, 4 = debries
```

J'ai ensuite fait une méthode d'initialisation aléatoire de la grille. Je l'ai codé en tirant au hasard un chiffre et ensuite dans un tableau contenant les chiffres représentant les robots, le joueur et les débris, je récupère la valeur correspondant à l'indice choisi au hasard. Une fois cette valeur placée dans la grille, je mets le compteur correspondant à jour, et ensuite je regarde si ce compteur a atteint sa limite. Si c'est le cas, je supprime la valeur du tableau.

**position\* positionJoueur();**

```
void terrain::changerPosJoueur(position*p){
```

$$\text{d\_grille}[\text{static\_cast<unsigned>}(\text{d\_joueur}-\text{positionJoueur})-\text{nbligne}][\text{static\_cast<unsigned>}(\text{d\_joueur}-\text{positionJoueur})-\text{nbColone}] = \text{d\_grille}[\text{static\_cast<unsigned>}(\text{p}-\text{nbligne})][\text{static\_cast<unsigned>}(\text{p}-\text{nbColone})];$$

$$\text{d\_grille}[\text{static\_cast<unsigned>}(\text{p}-\text{nbligne})][\text{static\_cast<unsigned>}(\text{p}-\text{nbColone})] = \text{d\_grille}[\text{static\_cast<unsigned>}(\text{p}-\text{nbligne})][\text{static\_cast<unsigned>}(\text{p}-\text{nbColone})];$$

$$\text{d\_joueur}.deplacerVers(\text{p}-\text{nbColone}, \text{p}-\text{nbligne});$$
}

```
if(nbalea==1){
    d_joueur.deplacerVers(j,i);
    ++compteurJoueur;}
```

Après ça, nous nous sommes rendus compte que le joueur et les robots possédaient des méthodes similaires, et nous avons décidé de faire une classe élément. Une fois cette classe faite par Solène, et que le joueur et les robots furent remaniés par Yaya et Véronika, j'ai pu à nouveau avancer sur la classe terrain. Tout d'abord, j'ai enlevé les paramètres nombre de robot et nombre de débris et je les ai remplacés par deux vecteurs, un de robots et un de débris.

J'ai également modifié la méthode d'initialisation en créant un nouveau robot ou débris et en l'incluant dans son vecteur respectif à chaque fois qu'il était tiré au hasard. J'ai modifié les méthodes qui retournent les nombres de robots et de débris en les calculant à partir des vecteurs. J'ai supprimé la méthode qui mettait à jour la grille, puisque cette grille n'existe plus.

```

if(nbalea==0){++compteurZero;}
if(nbalea==1){
    d_joueur.deplacerElement(new position{j,i});
    ++compteurJoueur;
}
if(nbalea==2){
    d_robots.push_back(new robot1G(new position{j,i}));
    ++compteurRobot1G;
}
if(nbalea==3){
    d_robots.push_back(new robot2G(new position{j,i}));
    ++compteurRobot2G;
}
if(nbalea==4){
    d_debris.push_back(new debris(new position{j,i}));
    ++compteurDebris;
}

```

```

void terrain::deplacementRobot(){
    for(int i=0;i<static_cast<int>(d_robots.size());++i){
        d_robots[static_cast<unsigned>(i)]->deplacerRobot(d_joueur);
        d_robots[static_cast<unsigned>(i)]->affichePosition(); //Test
    }
    collisionRobotEtRobot();
    collisionRobotEtDebris();
}

```

Les méthodes de collisions testent s'il y a eu une collision entre deux robots ou entre un robot et un débris. Dans la méthode qui teste si deux robots entrent en collision, je supprime ces deux robots du vecteur si le test est positif. Dans la seconde méthode avec le débris, je crée d'abord une nouvelle position qui prend en compte la position de la collision , ensuite je supprime le robot du vecteur, et enfin je crée un nouveau débris à cette position.

J'ai aussi ajouté une méthode deplacementRobot, qui parcourt le vecteur de robot et lance la méthode de déplacement, et ensuite, je lance les méthodes de collision de terrain que j'ai créé.

```

void terrain::collisionRobotEtRobot(){
    for(int i=0;i<static_cast<int>(d_robots.size());++i){
        for(int j=i+1;j<static_cast<int>(d_robots.size());++j){
            if(d_robots[static_cast<unsigned>(i)]->collision(*d_robots[static_cast<unsigned>(j)])){
                std::cout<<"Collision 2 robots"<<std::endl; //Test
                position*p=d_robots[static_cast<unsigned>(i)]->positionElement();
                // supprimerValeurTableau(d_robots,i);
                // supprimerValeurTableau(d_robots,j);

                for(int g=i;g<static_cast<int>(d_robots.size()-1);++g){
                    d_robots[static_cast<unsigned>(g)]=d_robots[static_cast<unsigned>(g+1)];
                }
                d_robots.pop_back();

                for(int g=j;g<static_cast<int>(d_robots.size()-1);++g){
                    d_robots[static_cast<unsigned>(g)]=d_robots[static_cast<unsigned>(g+1)];
                }
                d_robots.pop_back();

                d_debris.push_back(new debris(p));
            }
        }
    }
}

```

```

d terrain::suppValeurTab(std::vector<int>&V,
                           int i) {
    for(int j = 0;j<V.size();j++){
        if(V[j]==i){
            for(int d=j;d<V.size()-1;d++){
                std::swap(V[d],V[d+1]);
            }
            V.pop_back();
        }
    }
}

```

Ensuite, j'ai cherché à optimiser mon code. Pour ça, j'ai créé une méthode suppValeurTab qui supprime la valeur passée en paramètre dans un tableau d'entiers passé aussi en paramètre. Je peux ensuite utiliser cette méthode dans ma méthode d'initialisation afin de réduire la taille du code. J'ai également ajouté des commentaires pour une meilleure compréhension.

Enfin, après l'ajout de la méthode collision dans Element, j'ai pu encore améliorer la classe terrain. J'ai utilisé cette nouvelle méthode dans mes méthodes de collision, mais également dans la méthode joueurAPerdu, en testant si le joueur entrait en collision avec un robot ou un débris. Et pour terminé, j'ai rajouté quelques méthodes pour faire fonctionner le score du joueur.

# Yaya



La classe position :

Cette classe a été choisie pour faciliter l'accès à l'emplacement des personnages du jeu. Ce qui nous permet de regrouper les informations comme la colonne et la ligne qui sont dans la plupart du temps ensemble et d'appliquer des traitements globaux sur ces données.  
La position est créée avec une ligne et une colonne qu'elle encapsule.

Cette classe possède des méthodes comme :

changerPosition : cette méthode permet de modifier la position courante et est utilisée par les autres objets facilitant ainsi les déplacements des personnages.

sauverPosition : cette méthode est utilisée pour afficher la position dans un flux de sortie. Elle s'adapte aux différents flux grâce à l'objet ostream et va nous permettre par exemple de charger une position dans un fichier.

LireDepuis : parallèlement à la sauvegarde elle va permettre de lire une position à partir du flux passé en paramètre.

```
#ifndef POSITION_H
#define POSITION_H
#include <iostream>

class position{
public:
    position();
    position(int ligne,int colonne);
    int numColonne() const;
    int numLigne() const;
    void changerPosition(int col,int ligne);
    void sauverPosition(std::ostream& ost);
    void LireDepuis(std::istream& ist);
    void afficherPosition(std::ostream& ost);
    void DeplacerVers(int lig,int col);

private:
    int d_numLigne;
    int d_numColonne;
};

std::ostream& operator<<(std::ostream& ost, position&p);
std::istream& operator>>(std::istream& ist,position&p);
#endif // POSITION_H
```

numColonne : elle a été définie pour accéder à la colonne où est logée un personnage au lieu d'exposer ces données, ce qui serait dangereux pour assurer la cohérence des données.

Les surcharges des opérateurs comme "<<" et ">>" sont utilisées dans le but de faciliter l'écriture du code et de le rendre plus propre.

La classe joueur :

La classe joueur est définie pour représenter un joueur dans le jeu. Elle nous permet d'effectuer les différents déplacements d'un joueur. Un joueur est un Element qui a en plus un nom, un score et une durée de vie c'est ce qui explique d'ailleurs sa relation d'héritage avec le sur-type ELEMENT. Ainsi, un joueur est construit à partir de sa position, son nom et son score ce qui lui permettra de construire sa partie Element et d'initialiser ses données. En effet, cette classe n'a pas besoin de définir les méthodes relatives aux déplacements car le joueur ne se déplace pas d'une manière différente aux autres personnages du jeu. Seuls les déplacements de l'ELEMENT( son sur-type) lui suffisent pour bien naviguer entre les différentes positions existantes dans le jeu.

Cependant le joueur a besoin de définir certaines méthodes qui lui sont propres du fait que le joueur effectuait ses actions autrement.

`augmenterDureeVie` : cette méthode est définie pour faciliter le calcul du score d'un joueur au fur et à mesure qu'il continue d'effectuer ses déplacements.

`nomJoueur` : permet de connaître le nom du joueur. Ce qui pourrait être intéressant pour savoir la liste de tous les gens qui ont joué.

`sauverJoueur` : comme son nom l'indique elle permet de sauver toutes les données relatives au joueur dans la console, fichiers, chaînes...

`calculScore` : elle met à jour le score du joueur en lui transmettant la valeur stockée dans "duréeVie" (préalablement calculée).

`LireDepuis` : cette méthode permet aussi de récupérer les données d'un joueur grâce aux flux passés en paramètre.

```
#ifndef JOUEUR_H
#define JOUEUR_H
#include <iostream>
#include "element.h"

class joueur: public element
{
public:
    joueur();
    joueur(position*p,const std::string&nom, int score);
    int score()const;
    void calculScore();
    std::string nomJoueur()const;
    int dureeVie()const;
    int nbRobotsDetruits()const;
    void Reinitialiser();

    void augmenterDureeVie();
    void augmenterNbRobotsDetruits();

    void sauverJoueur(std::ostream&ost)const;
    void LireDepuis(std::istream&ist);

private :
    std::string d_nom;
    int d_score;
    int d_dureeVie;
    int d_robotDetruit;
};

std::ostream& operator<<(std::ostream&ost, joueur& j);
std::istream&operator>>(std::istream&ist,joueur&j);

#endif // JOUEUR_H
```

```
#ifndef JOUEUREXPERT_H
#define JOUEUREXPERT_H
#include "joueur.h"
#include<iostream>
class position;

class joueurExpert:public joueur{
public:
    joueurExpert();
    joueurExpert(position*p,const std::string&nom, int score);
    void displaceElementHautDroite()=delete;
    void displaceElementHautGauche()=delete;
    void displaceElementBasDroit()=delete;
    void displaceElementBasGauche()=delete;
};
```

Les tests :

Chacune de ces classes citées ci-dessus ont aussi un fichier qui leur sont propres où toutes les fonctionnalités sont testées. Ces tests ont permis de bien s'assurer que les objets font bien ce qu'on attendait d'eux mais également à remanier certaines classes .

La classe `joueurExpert` :

Un joueurExpert est un joueur sauf qu'il ne peut pas se déplacer en diagonale. Elle est donc un sous-type de la classe joueur par contre le déplacement vers les diagonales n'existe pas. Elle est ainsi construite avec les mêmes données que le joueur à savoir la position, le nom et le score.

```
SUBCASE("la construction du joueur est correcte")
{
    position*p=new position{1,3};
    std::string nom="Hirobot";
    int score=10;
    joueur j{p,nom,score};
    REQUIRE_EQ(j.nomJoueur(),nom);
    REQUIRE_EQ(j.score(),score);
    REQUIRE_EQ(j.positionElement()->numLigne(),1);
    REQUIRE_EQ(j.positionElement()->numColonne(),3);
}
```



## La classe élément :

Nous avons commencé à programmer le projet en se répartissant les différents éléments (terrain, robots, joueurs, débris) et suite à un cours sur l'héritage nous nous sommes rendus compte que plusieurs méthodes notamment les déplacements des robots et des joueurs étaient répétées. J'ai donc programmé cette nouvelle classe « élément » qui regroupe les méthodes de déplacements et de position des différents éléments. Mes coéquipiers ont ensuite récupéré ce code et ont modifié leurs classes en appelant ces nouvelles fonctions au lieu de les redéfinir à chaque fois. Cela a permis d'avoir un code plus clair et plus compréhensif.

```
#ifndef ELEMENT_H
#define ELEMENT_H
#include "position.h"

class element
{
public:
    element(position*p);
    position* positionElement() const;
    void placerElement(position*p);
    void placerElementGauche();
    void placerElementDroite();
    void placerElementBas();
    void placerElementHaut();
    void placerElementHautGauche();
    void placerElementHautDroite();
    void placerElementBasGauche();
    void placerElementBasDroite();
    bool peutSeDeplacer(int nbLignes, int nbColonnes, int typeDeplacement);
    bool collision(element elem);
private:
    position* d_pos;
};

#endif // ELEMENT_H
```

Par la suite, j'ai aussi créé des méthodes de déplacements en diagonale directement de façon à ce qu'on ait pas besoin de répéter deux fois de suite une méthode haut et droite par exemple. De plus, afin d'éviter un bug où le joueur pouvait sortir du terrain défini, j'ai créé une méthode « peutSeDeplacer » qui permet de tester si l'élément en question est autorisé à se déplacer dans telle ou telle direction en vérifiant à partir de sa position initiale s'il est au bord du terrain ou non. La dernière méthode que j'ai dû concevoir qui s'appelle « collision » vérifie que deux éléments se rentrent dedans c'est-à-dire si leur position est égale. Cela évite de répéter dans chaque test (robot - robot ou robot - débris ou joueur - robot ou encore joueur - débris ) la vérification si ces éléments sont entrés en collision et s'il faut alors afficher perdu pour le joueur, faire disparaître le robot entré dans un débris ou créer un débris suite au choc entre deux robots. Cela facilite donc la lecture du code une fois encore.

```
#include<list>
#include"element.h"

class debris:public element
{
public:
    debris();
    debris(position*p);
    void placerElement(position*p)=delete;
    void placerElementGauche()=delete;
    void placerElementDroite()=delete;
    void placerElementBas()=delete;
    void placerElementHaut()=delete;
    void placerElementHautGauche()=delete;
    void placerElementHautDroite()=delete;
    void placerElementBasGauche()=delete;
    void placerElementBasDroite()=delete;
};

#endif // DEBRIS_H
```

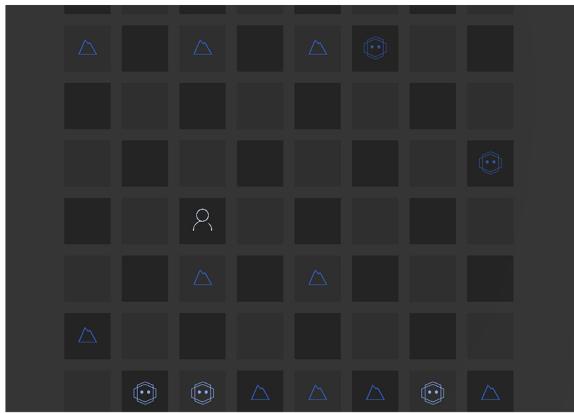
## La classe débris :

Lorsque l'on choisit un niveau de difficulté (ou que l'on paramètre nous-même une grille) un certains nombre de débris apparaissent sur le terrain. Ceux-ci ne possèdent qu'une simple position et ne se déplaceront pas, c'est au joueur de faire attention de ne pas les percuter. En effet, si le joueur arrive malencontreusement dessus, la partie s'arrêtera et il aura perdu. De plus, lors de la collision de deux robots entre eux, un débris se crée à la position où ils se trouvaient au préalable et si le joueur parvient à ce qu'un débris soit percuté par un robot ce dernier disparaîtra et le score du joueur augmentera.

# INTERFACE

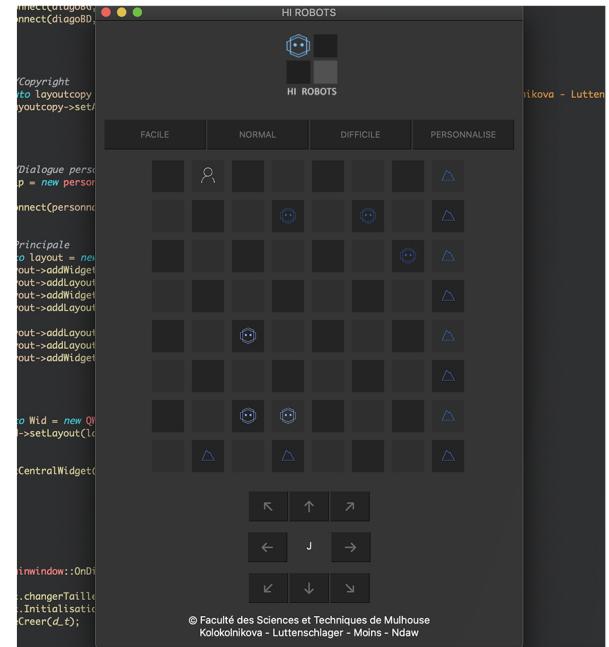
## Mélissa

J'ai commencé l'interface après avoir fait la première version de la classe terrain. J'ai donc crée trois classes pour cette interface : la classe grille de type QWidget qui modélise le plateau de jeu, une classe personnaliserTerrain de type QDialog qui ouvre une boîte de dialogue et permet à l'utilisateur d'entrer ses propres valeurs, et enfin une classe mainwindow de type QMainWindow et qui représente la fenêtre principale. Cette fenêtre principale se construit à partir d'un terrain, et elle affiche la grille correspondant à ce terrain.

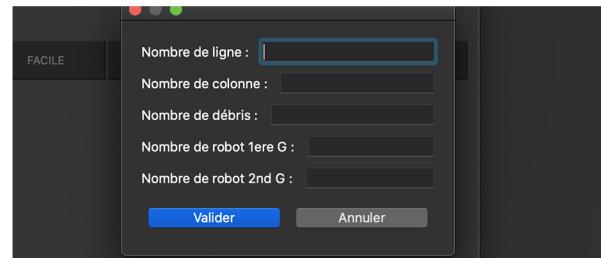


Pour ce qui est de la classe grille, j'ai commencé par créer un paintevent pour pouvoir dessiner le terrain. J'ai utilisé les fonctionnalités de Qt comme QPainter et QRect. J'ai parcouru le terrain qui permet de construire la grille et pour chaque case du vecteur, je dessine un carré, ensuite je teste si cette case contient un joueur (1), un robot1G (2), un robot2G (3), ou un débris (4), et j'affiche l'image correspondante. Enfin, une fois deux cases traitées, j'inverse les couleurs des pinceaux.

Ensuite j'ai fait la classe mainwindow que je construis à partir d'un terrain. La méthode vueCréer va construire l'ensemble de la fenêtre à partir d'un terrain. Je commence par lui donner un titre, et ensuite je fixe une taille à la fenêtre pour une meilleure visibilité. Je crée ensuite une grille grâce à la classe grille et à partir du terrain passé en paramètre. J'ai ensuite créé des boutons de niveaux qui permettent à l'utilisateur de choisir le niveau de jeu qu'il souhaite. Et j'ai également fait des boutons de déplacements pour jouer. Je place ensuite ensemble dans l'ordre voulu dans le layout principal. J'ai rattaché ce layout à un widget et enfin ce widget à la fenêtre principale. Après ça, j'ai connecté les boutons à leur méthode respective.



Pour la classe personnaliserTerrain, j'ai créé des QLineEdit pour chaque valeur à choisir ainsi que deux boutons : un pour valider et un pour annuler. Ensuite lorsqu'on clique sur valider, je récupère les valeurs entrées dans les QLineEdit, je vérifie qu'elles soient valides, et ensuite je les applique au terrain.



Ensuite j'ai refait des modifications dans la classe grille et dans la classe fenêtre après que la classe élément fût ajoutée. Dans la classe grille, je n'utilise plus le vecteur de terrain, puisque je l'avais supprimé, mais j'utilise les positions. Je teste, par exemple, si la position du joueur correspond à la position sur laquelle je suis en train de dessiner, si c'est le cas, j'affiche l'image du joueur. Et je fais la même chose en parcourant le vecteur de débris et de robot.

```
void mainwindow::deplacement(){
    d_t.deplacementRobot();
    d_t.Joueur().changerDureeVie(d_t.Joueur().vueCreer(d_t));
    if(d_t.JoueurAPerdu()){
        perdu();
    }
    if(d_t.JoueurAGagne()){
        gagne();
    }
}
```

Dans la classe mainwindow, je vais modifier les méthodes de déplacements. Je remplace mon test, pour savoir si le joueur peut se déplacer, par celui ajouté dans la classe élément. J'ajoute aussi une méthode déplacement, qui va déclencher le déplacement des robots et ensuite recharger la vue, pour éviter une répétition dans les autres méthodes de déplacements. Je vais aussi ajouter deux méthodes gagne et perdu, qui affiche respectivement un message qui indique que le joueur a gagné ou perdu.



## Solène

Au départ on avait un chemin spécifique vers un dossier avec les images. Or dans l'équipe on a deux utilisateurs sous Mac OS et deux autre sous Windows ce qui fait que les chemins ne sont pas du tout écrit de la même manière. On était donc parti sur une variable avec les chemins au début de certains fichiers et chacun devait le modifier selon son système d'exploitation. Cette solution devait rester provisoire, on a alors découvert les « .qrc ». Ces fichiers ressources permettent de lister les différents fichiers que l'on stocke dans un dossier dans le projet (le même que le « .qrc »). Ceux-ci ont un chemin relatif et non absolu ce qui simplifie le passage d'un système à un autre avec l'utilisation des images. Dans notre cas, on a les différentes images que l'on utilise lors de l'affichage ( par exemple des robots première et deuxième génération, le joueur, des débris ... ).

```
<RCC>
<qresource prefix="/Images">
    <file>Images/debris.png</file>
    <file>Images/explo.png</file>
    <file>Images/hirobot2.png</file>
    <file>Images/joueur.png</file>
    <file>Images/robot1G.png</file>
    <file>Images/robot2G.png</file>
    <file>Images/legende.png</file>
</qresource>
</RCC>
~
~
~
```

# PROBLEMES

RÉSOLUS

Le premier problème que nous avons rencontré est un problème avec les pointeurs. En effet nous avions mis la position du joueur en pointeur, sachant qu'elle change à chaque déplacement. Cependant, lorsque nous lancions le jeu, il s'arrêtait subitement, nous avions l'erreur « le programme s'est terminé subitement ». Pour résoudre ce problème, Mélissa a créé une méthode dans la classe terrain qui renvoyait la position du joueur afin que nous puissions l'utiliser dans la classe mainwindow, sans passer par l'intermédiaire de la méthode qui renvoie le joueur.

Ensuite nous avons rencontré un problème au niveau des boutons de déplacements. Lorsque nous cliquions sur le bouton en bas à gauche ou sur le bouton en haut à droite, il arrivait que le joueur ne se déplace pas. Le problème s'est résolu lorsque Solène a écrit la méthode peutSeDeplacer dans la classe élément.

Nous avons également rencontré un problème au niveau du déplacement des robots, et plus particulièrement pour les robots de première génération. Ces robots ne se déplaçaient pas dans le bon sens parce que nous avions fait des erreurs de calculs. Véronika a fini par résoudre ce problème.

NON RÉSOLUS

Nous n'avons pas réussi à créer une ressource pour le fichier de sauvegarde. Nous n'avons pas non plus réussi à intégrer le joueurExpert à l'interface. Nous avons également rencontré un problème dans la méthode d'initialisation du terrain, en effet nous n'avons pas réussi à faire en sorte de pouvoir entrer par exemple 0 robots de première génération. Mélissa a donc mis une condition pour qu'on ne puisse pas entrer une valeur à 0.

# CONCLUSION

## GÉNÉRALE

Comment mettre en place l'architecture du jeu de manière optimisée ? C'est cette problématique qui était mise en jeu durant ce projet.

Nous avons pu apprendre à mettre en pratique les méthodes apprises en cours durant ce projet, notamment à travers l'outil Git, qui nous était imposé. Git étant très utilisé dans les entreprises, nous avons pu apprendre à l'utiliser. Cet outil nous a permis de gérer de manière optimisée, la conception de notre code, à travers d'éventuels conflits lorsque deux personnes modifiaient le même fichier par exemple. Nous avons appris que la communication était un élément essentiel lors d'un projet, puisque si nous voulions réussir à utiliser Git, nous devions régulièrement échanger et répartir les tâches.

Pour optimiser notre code, nous avons également utilisé les tests avec la syntaxe de doctest, ce qui nous a permis d'être sûr que nos fonctions répondaient à tous les critères voulus.

Nous avons donc réussi à produire un jeu qui fonctionne et est optimisé. Cependant, il reste encore des points à améliorer que nous n'avons pas pu traiter par manque de temps, comme la sauvegarde du jeu par exemple.

## PERSONNELLES

### MELISSA

Ce projet m'a permis de mettre en pratique les méthodes apprises en cours. J'ai pu facilement comprendre comment fonctionnait la qualité de programmation à travers les tests, mais aussi l'optimisation et l'outil Git, que je ne connaissais pas avant. Je suis très contente du résultat que nous avons obtenu, malgré que nous n'avons pas pu aller au bout de certaines choses.

### YAYA

Tout d'abord je pense que la réalisation d'un projet en 3ème année est une très bonne chose car cela nous permet d'avoir un contact avec le monde réel du travail en plus de mettre en pratique certaines notions de cours. Dans les éléments positifs de notre projet, je voudrais mentionner la cohésion du groupe. Personne n'est resté en retrait, les tâches à faire ont été respectées par tous les membres. Ce projet m'a permis d'acquérir d'avantage d'expérience. Je sais par exemple que la gestion du temps est un critère déterminant et il faut se mettre au travail dès les premières semaines sinon ce temps risque de nous manquer à la fin. Ce projet m'a également permis de travailler sur d'autres outils informatiques de gestion de projet que je ne connaissais pas malgré leur efficacité (GitHub à titre d'exemple) mais aussi à travailler en équipe. Pour les tâches que j'ai eu à faire, j'ai rencontré quelques problèmes de conception notamment pour la hiérarchie des classes et l'encapsulation des données, mais on a réussi à résoudre tous ces détails ensemble.

### VERONIKA

Ce projet m'a permis d'approfondir les outils que j'ai pu acquérir grâce au cours de qualité de programmation. Cela m'a permis de comprendre comment collaborer dans un groupe à l'aide de Git. J'ai dû apprendre comment implémenter ma partie en utilisant le code des autres parties. J'ai dû changer mon code en faisant des classes dérivées et des fonctions plus simples. Enfin j'ai pu m'exercer à faire des tests, à commenter clairement un code et écrire du code propre.

### SOLENE

Ce premier projet en groupe de troisième année m'a permis sur un plan personnel d'enrichir mes connaissances en C++. En effet, mes camarades étant légèrement plus avancés que moi, ils ont pu m'aider et être à l'écoute des problèmes et des bugs que j'ai rencontrés. En contrepartie, grâce à mes expériences dans le monde professionnel, j'ai pu aider à la création et la gestion du Git. Pour finir, sur un plan scolaire, j'ai trouvé très intéressant de travailler sur un projet conséquent (pas sur papier) avec des collègues qui n'ont pas eu le même parcours et qui ont donc chacun à leur façon apporter leur pierre à l'édifice.

# TABLE DES MATIERES

INTRODUCTION .....	page 01
I. LE PROJET .....	page 02
1. Description .....	page 02
2. Répartition .....	page 02
II. OUTILS UTILISÉS .....	page 03
1. Gestion .....	page 03
2. Développement .....	page 04
III. DÉROULEMENT .....	page 05
1. Gestion .....	page 05
1.1. Solène .....	page 05
2. Développement .....	page 06
2.1. Véronika .....	page 06
2.2. Mélissa .....	page 07
2.3. Yaya .....	page 09
2.4. Solène .....	page 11
3. Interface .....	page 12
3.1. Mélissa .....	page 12
3.2. Solène .....	page 13
IV. PROBLEMES .....	page 14
1. Résolus .....	page 14
2. Non résolus .....	page 14
CONCLUSION .....	page 15
1. Générale .....	page 15
2. Personnelles .....	page 15
TABLE DES MATIERES .....	page 17