Projet tuteuré
« Licence pro informatique »

Robocode

année 2012 – 2013

Étudiants ayant participé au projet :
Dorian Coffinet
Thierry Coutant
Charly Poilane

Tuteurs :
Frédéric Lardeux
Frédéric Saubion

Sommaire

I – LE PROJET ROBOCODE

I.1 Le sujet d'étude

I.1.1Sujet proposé

Robocode

Robocode est un jeu vidéo qui a été initialement développé par IBM à des fins pédagogiques. Le jeu repose sur des combats entre des robots (sorte de chars d'assaut) qui peuvent se déplacer et tirer, l'objectif étant de détruire tous les concurrents. Il est ainsi possible de programmer son robot puis de le confronter sur un champ de bataille avec d'autres robots. La programmation s'effectue de manière très intuitive en Java.

Un environnement complet est distribué librement (http://robocode.sourceforge.net/) et permet de développer facilement de nouveaux robots et de les tester.

Ce projet s'adresse à deux équipes A et B dont le but final sera de s'affronter.

Le projet est organisé de la manière suivante :

Phase 1 : étude du jeu et de l'environnement de programmation

Fournir une analyse complète de l'application (modèle UML) et un manuel de programmation des robots
Equipe A : proposer un éditeur de champs de bataille
Equipe B : proposer une interface web pour piloter l'application

Phase 2 : développement de nouveaux robots

Identifier des stratégies de combat
Développer une interface graphique permettant de concevoir rapidement un robot en quelques clics
Programmer un robot efficace
Programmer une équipe de robots

Phase 3 : compétitions de robots

Deux affrontements seront organisés entre les équipes A et B à deux semaines d'intervalle et selon plusieurs type d'évaluation :

Combat 1 contre 1
Combat en équipes de robots
Combat contre des robots de très bonne qualité (déjà implémentés)

I.1.2 Présentation du jeu

Robocode est un jeu qui permet d'apprendre à programmer en java en faisant affronter des chars robotisés virtuels.

C'est une application qui permet de programmer ses chars robotisés, de les tester et de lancer des combats sur un champ de bataille que l'on choisit plus ou moins grand. Pendant ces batailles, 2 à plusieurs chars à 6 roues combattent jusqu'à ce qu'un seul d'entre eux reste « en vie ». Ces « tanks » robotisés peuvent exploser lorsqu'ils sont plusieurs fois touchés par des obus adverses, ou qu'ils perdent trop d'énergie en percutant les murs ou d'autres chars. Ils explosent aussi lorsqu'ils restent trop longtemps inactif.

I.1.3 Paramètres du jeu

Les paramètres pris en compte sont les vitesses, l'accélération, les distances, les puissances, les dommages, les températures, l'énergie. Il faut noter que l'unité utilisée dans Robocode pour les déplacements est le pixel, en sachant que graphiquement l'application peut utiliser des fractions de pixels. Les rotations sont exprimées en degrés ou radians.

Variable ou Constante	**valeurs**
Accélération (a)	1 pixel/tour
Décélération	2 pixel/tour
Vitesse (V) = a*t	Max = 8 pixel/tour
Rotation du Canon	Max = 20 degrés/tour
Rotation du Châssis	Max = 10 degrés/tour (=10 - 0.75 * abs(vitesse)) deg / turn
Rotation du Radar	Max = 45 degrés/tour
Rayon du radar	1200 pixels
PuissanceDeFeu Obus	Min = 0,1 à max = 3
Énergie gagné par coup touché	1,2
Obus Dommage	4 * puissanceDeFeu. Si puissance > 1, it does an additional damage = 2 * (power - 1).
Obus Vitesse	20 - 3 * puissanceDeFeu. entre 11.0 et 19.7
Obus régénération	1 + puissanceDeFeu / 5
Elévation température sur 1 tir	3 * puissanceDeFeu.
Dommage Collision Robot	0,6 pour chaque robot
Dommage Collision Mur	Uniquement Advanced (abs(vitesse) * 0.5 - 1)

I.2 – Fonctionnement du groupe

I.2.1 – Première approche du sujet

Après avoir choisi et installé une version commune de Robocode pour l'ensemble des groupes qui travaillaient sur ce sujet, nous avons tout d'abord répertorier d'une manière générale les différentes classes et interfaces qui constituaient l'application.
Très vite, du fait du nombre très conséquents d'entités, la nécessité de hiérarchiser les classes, de préciser leur nature (abstraite,finale, ou classique), de relever les interfaces qu'elles implémentaient, s'imposait.
L'élaboration fastidieuse du diagramme de classe UML nous a permis de comprendre le fonctionnement général de l'application et d'identifier les classes importantes pour la fabrication d'un robot.
De là, nous avons étudié en détail et traduit en français les méthodes de la classe « Robot », « AdvancedRobot » et « ScannedRobotEvent » qui nous paraissaient les plus importantes pour démarrer la programmation d'un robot.
Parallèlement, nous avons étudié les paramètres du jeu et abordé les règles principales.
Puis, nous avons étudiez le code et les caractéristiques des robots existants dans l'application.
Tout ceci a été réalisé ensemble au sein du groupe pendant les deux premiers jours à l'exception du diagramme UML.

I.2.2 – Répartition du travail (semaine 1 et 2)

Ensuite, nous avons décidez de continuer à travailler ensemble sur le même lieu en se partageant, le travail :

- Trouver les formules mathématiques ou comprendre et traduire celles qui sont présentées dans le wiki pour justifier leur utilisation.

- Réfléchir aux attributs à placer dans l'interface graphique de création de robot. Élaborer le manuel d'installation de l'application.

- Élaborer un tutoriel de création de robot dans l'application

- Recherche de documentations pour l'élaboration de l'interface en java

- Les thèmes suivants ont été abordés par chacun d'entre nous avant mise en commun de nos investigations pour confrontation de nos idées et décisions communes :

    - Tester l'utilisation de méthodes dans le jeu avec un premier robot créé.
    - Thèmes jugés importants pour décider de la stratégie de notre futur robot.
    - Codage du robot

I.2.3 – Répartition du travail (semaine 3 et 4)

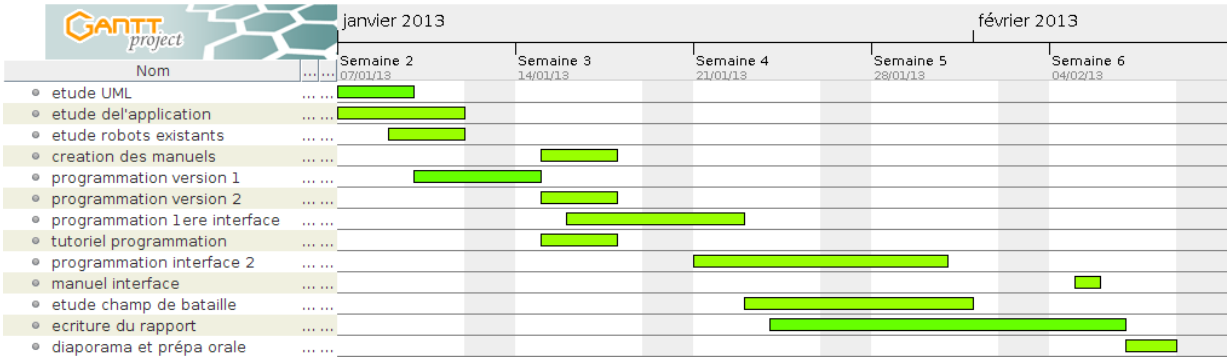
Après avoir fait un bilan collectif approfondi des résultats statistiques de la deuxième série de combats, nous avons établi notre plan de travail pour la semaine 3 du projet en nous répartissant le plus souvent possible les tâches. Pour répondre à notre volonté d'être clair dans nos propos et complet, systématiquement notre travail était repris par un autre pour l'améliorer ou le valider.

Ainsi, nous avons réalisés :

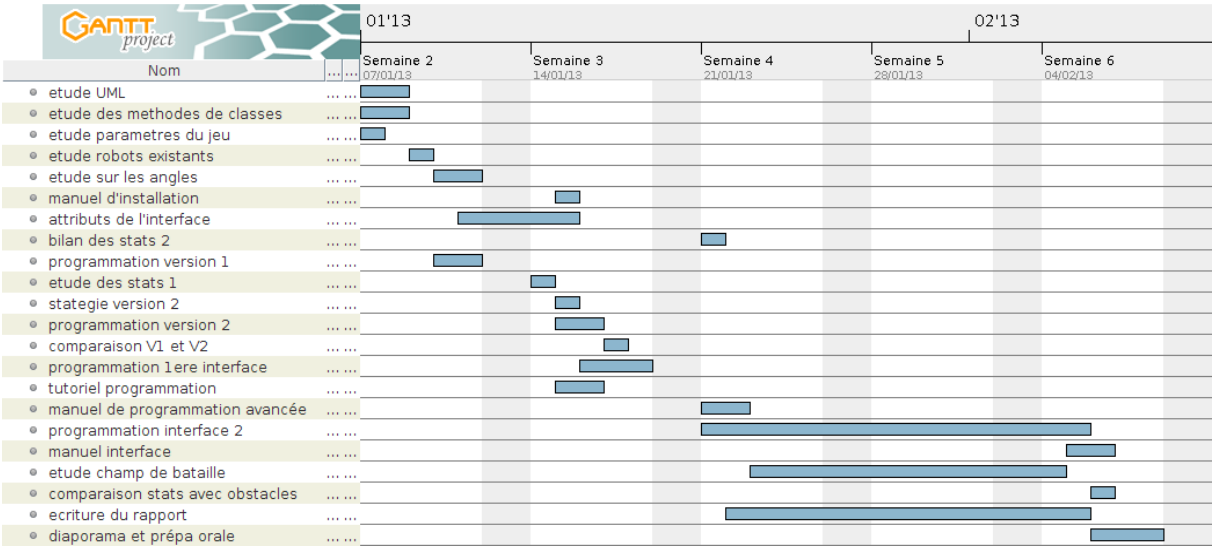
- l'étude des versions 2 des robots des autres groupes
- un manuel avancé de programmation de robots
- le codage de l'interface
- un manuel de présentation de l'interface
- l'étude du codage du champ de bataille

I.2.4 – Diagrammes de Gantt

- Diagramme prévisionnel



- Diagramme réel



II - Etude et utilisation de l'application

II.1 - Etude de l'application

II.1.1 – Fonctionnement général

Cette application écrite en Langage Objet Java, créée pour apprendre ce langage, répond aux caractéristiques des langages objets :

- classes abstraites et interfaces pour contrôler et mutualiser le code
- classes finales gérant les événements et ne pouvant pas avoir de sous-classes
- sous-classes héritant de classes pour élargir le champ des méthodes
- classes implémentant des interfaces pour préciser les méthodes

Le jeu qui permet de créer des objets « robots » utilisent une succession de classes héritant chacune d'une autre classe. Cette hiérarchie de classe permet de concevoir des robots de plus en plus sophistiqués, voire même des équipes de robots :

- la Classe Robot qui définit une grande majorité des méthodes
- la Classe JuniorRobot avec moins de méthodes et surtout des méthodes plus simples
- la Classe AdvancedRobot qui hérite de Robot, pour créer des robots plus avancés
- la Classe TeamRobot qui hérite de AdvancedRobot avec des méthodes pour passer des messages à d'autres robots, pour réaliser des « battles » en équipes.

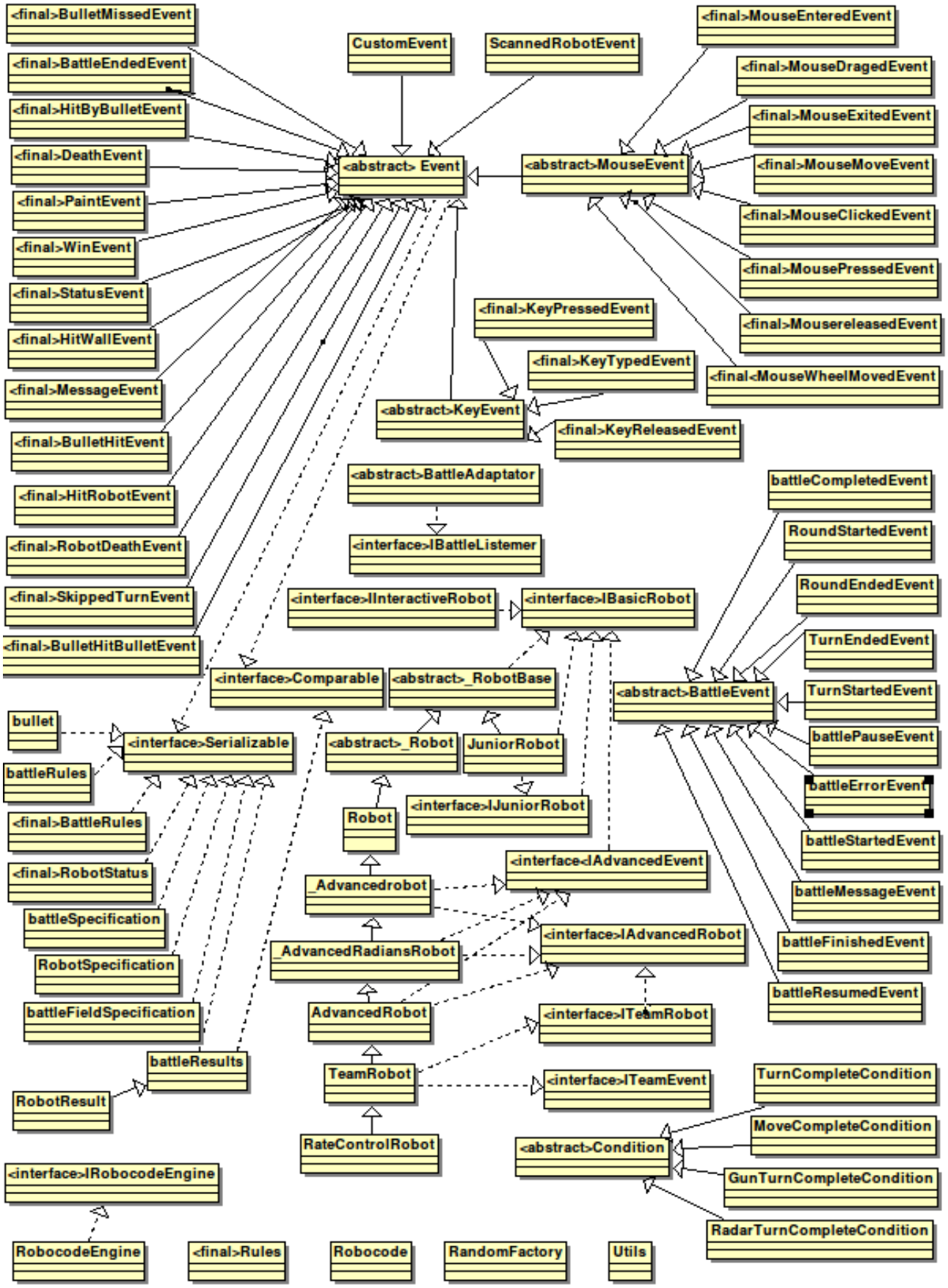
A l'exception de TeamRobot qui implémente IteamEvent et ITeamRobot, les classes Robot et AdvancedRobot implémentent quant à elles les deux interfaces IAdvancedEvent et IAdvancedRobot. La classe JuniorRobot implémente IJuniorRobot.

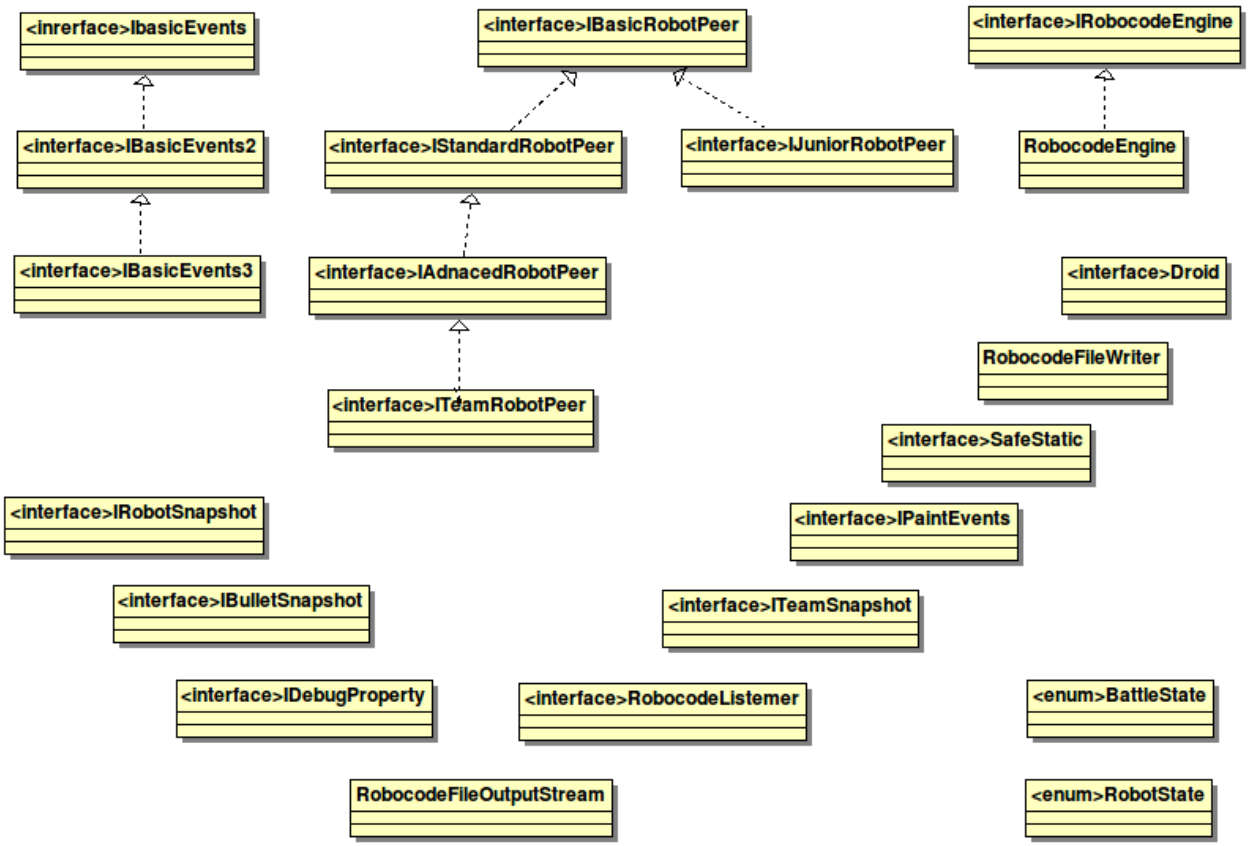
Un certain nombre de classes (battleRules, Bullet, RobotStatus, BattleSpecification, RobotSpecification, BattleFieldSpecification, ...) implémentent l'interface Serialisable qui permet de gérer la synchronisation des événements.
La classe abstraite Event permet de contrôler tous les événement du jeu, de nombreuses classes finales héritent de cette classe abstraite. Elles permettent entre autres de gérer les collisions entre robots,murs et robots, obus et murs, obus et robots, obus et obus, etc...

II.1.2 - Diagramme de classe UML

Après l'étude assez longue de la javadoc, nous avons essayé de représenter graphiquement sous forme de diagramme de classes UML simplifié toutes les entités de cette application codée en langage objet Java. Les champs et méthodes de classes ne sont pas précisés pour ne pas surcharger le schéma. D'ailleurs, cela ne serait pas possible au vu du nombre d'entités assez conséquent de cette application.
Par contre, la nature des classes abstraites ou finales est précisée si tel est le cas. De la même manière, nous avons identifié aussi dans l'intitulé toutes les interfaces. Nous avons représenté les flèches d'héritage et d'implémentation. Ainsi sur deux pages nous pouvons avoir une représentation assez précise de toute l'application.

Diagramme de classe simplifié de l'application





II.1.3 - Manuel d'installation (en annexe)

Ce manuel est réalisé au format PDF en gardant à l'intérieur du document les liens ou renvois entre le sommaire et les chapitres, vers les sites de téléchargements. En s'adressant aux débutants, il permet l'installation de l'application Robocode aussi bien sous Windows que sous Linux.
Après l'installation, ce manuel propose aussi de découvrir succinctement l'application pour pouvoir lancer des « battles ». Toutes les explications sont agrémentées d'images de capture d'écran pour plus de clarté.

II.1.4 - Tutoriel de création de robot (en annexe)

Nous avons envisagez de créer ce tutoriel pour des personnes ne connaissant pas l'application Robocode. Agréablement surpris par la forme des tutoriels que nous propose souvent « le site du Zéro » (site d'échange et d'apprentissage de l'informatique), nous avons essayé de nous en inspirer pour créer un tutoriel pseudo-interactif où le narrateur se pose lui-même les éventuelles questions du lecteur, ou les explications techniques sont entrecoupées de commentaires récréatifs, voire amusants pour détendre le lecteur, entretenir sa motivation, pour l'inciter à continuer la lecture du tutoriel. Il est lui aussi au format PDF en préservant les liens et renvois entre le sommaire et les chapitres.

II.2 – Mathématique des angles

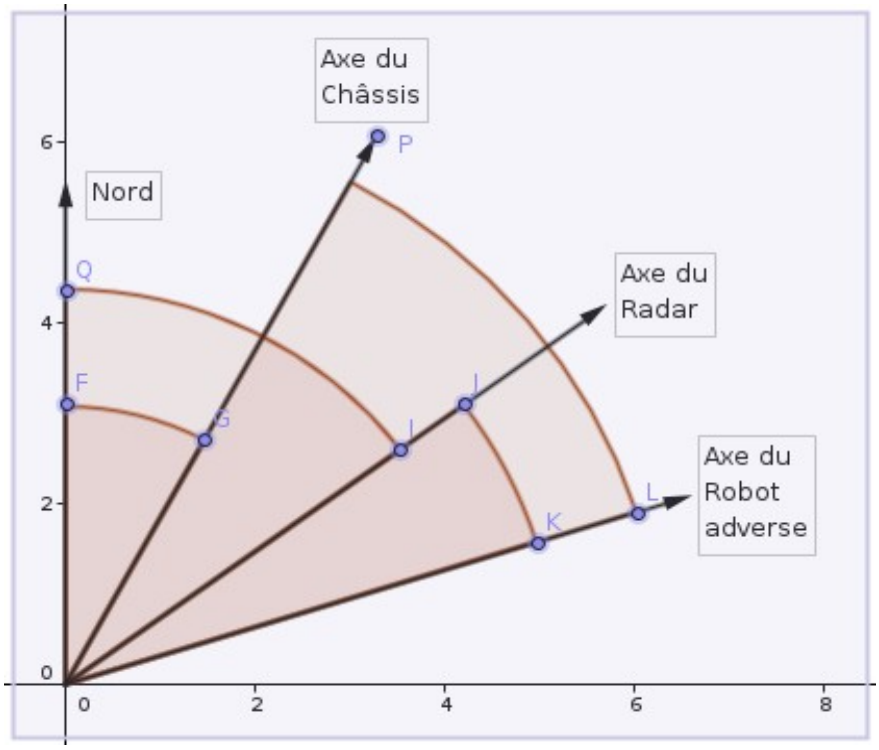
II.2.1 -Principes de base pour les alignements

- ***Alignement du châssis vers le robot adverse*** :

une méthode est déjà définie et utilise l'angle entre l'axe du châssis de notre robot et l'axe de positionnement du robot adverse par rapport à notre robot.

```
setTurnRight(e.getBearing());
```

- ***Alignement du radar sur le robot adverse*** :



Angle FOG : angle du châssis du robot par rapport au nord
compris entre 0 et 2*Π (unité en radians)
correspond à ce que retourne `getHeadingRadians() ;`

Angle QOI : angle du radar par rapport au nord
compris entre 0 et 2*P (unité en radians)
correspond à ce que retourne `getRadarHeadingRadians() ;`

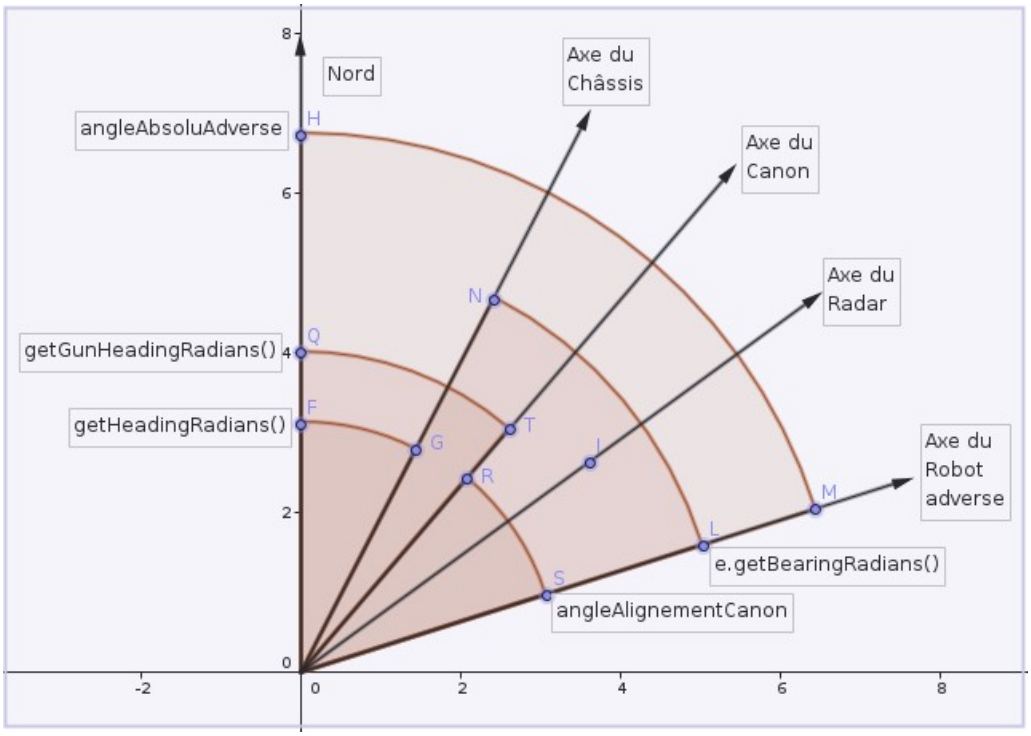
Angle POL : angle entre l'axe du châssis et l'axe du robot adverse scanné
compris entre – Π et + Π `e.getBearingRadians() ;`

Angle LOK : « angleAlignementRadar » angle à prendre en compte pour aligner le radar sur le robot adverse.
Égale à angle FOG + angle POL – angle QOI

```
getHeadingRadians()+e.getBearingRadians()- getRadarHeadingRadians()
```

En utilisant dans la classe `Utils` la méthode `normalRelativeAngle(angleAlignementRadar)` qui transforme un angle absolu (de 0 à 2*Π) en angle relatif compris entre – Π et + Π et en utilisant dans la classe `AdvancedRobot` la méthode `setTurnRadarRightRadians(double radians)` qui fait tourner le radar dès la prochaine exécution, on peut donc aligner le radar de notre robot en direction du robot adverse. Au final, cela donne :
`setTurnRadarRightRadians(Utils.normalRelativeAngle(angleAlignementRadar)) ;`

- ***Alignement du canon sur le robot adverse scanné***



Angle FOG : angle du châssis du robot par rapport au nord
compris entre 0 et 2*Π (unité en radians)
correspond à ce que retourne `getHeadingRadians() ;`

Angle QOT : angle du canon par rapport au nord
compris entre 0 et 2*P (unité en radians)
correspond à ce que retourne `getGunHeadingRadians() ;`

Angle NOL : angle entre l'axe du châssis et l'axe du robot adverse scanné
compris entre – Π et + Π
correspond à ce que retourne `e.getBearingRadians() ;`

Angle HOM : « angleAbsoluAdverse » angle entre le robot adverse et le nord.
Égale à : angle FOG + angle NOL
`getHeadingRadians() + e.getBearingRadians()`

Angle ROS : « angleAlignementCanon » angle à prendre en compte pour aligner le canon sur le robot adverse.
Égale à : « angleAbsoluAdverse » – angle QOT

```
getHeadingRadians() + e.getBearingRadians() - getGunHeadingRadians()
```

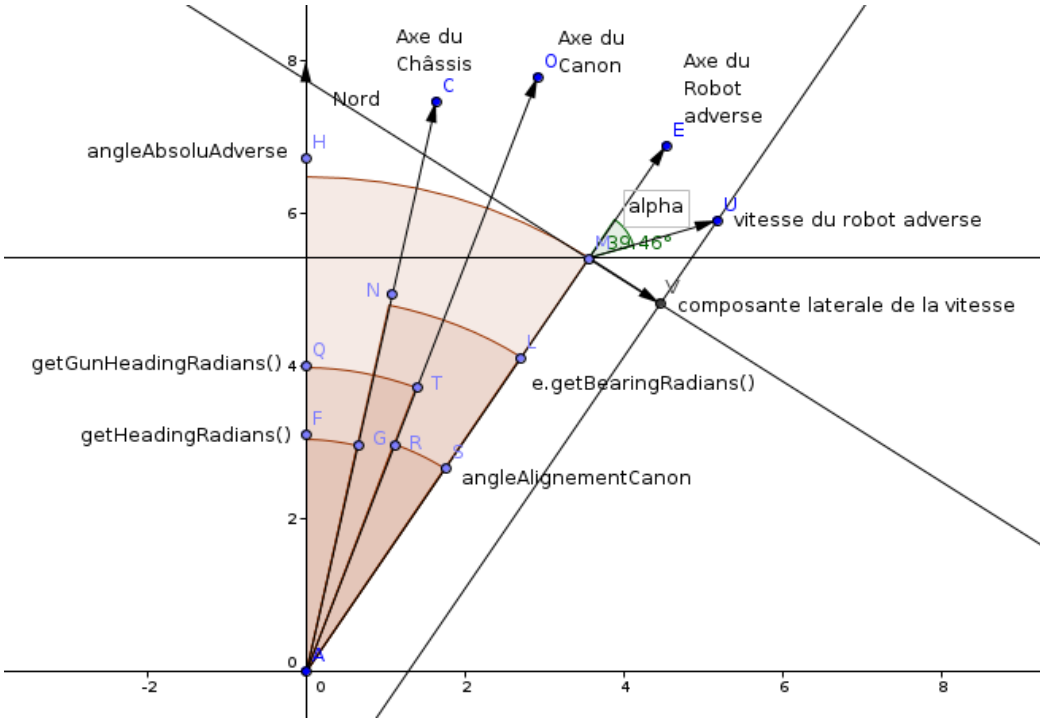
Comme pour l'alignement du radar, et en utilisant aussi dans la classe `AdvancedRobot` la méthode `setTurnGunRightRadians(double radians)` qui fait tourner le canon dès la prochaine exécution, on peut donc aligner le canon de notre robot en direction du robot adverse. Au final, cela donne :

```
setTurnGunRightRadians(Utils.normalRelativeAngle(angleAlignementCanon)) ;
```

II.2.2 – Corrections des angles de visée

- ***Correction de l'angle de tir rapproché***

Ici, nous allons donner (pour le cas de robots proches l'un de l'autre) l'explication mathématique sur le calcul de l'angle, puis pour alléger le code, nous remplacerons certains sous-calculs par des constantes. Nous montrerons ensuite que ce calcul, même s'il manque de rigueur scientifique peut être intéressant.



Dans le déplacement du robot adverse, ce qui nous intéresse de prendre en compte, c'est la composante latérale de son déplacement qui se situe sur l'axe perpendiculaire à l'axe du robot visé :
La vitesse latérale est définie en fonction du sinus (l'angle alpha)
avec alpha = `e.getHeadingRadians()- angleAbsoluAdverse ;`
et aussi en fonction de la vitesse du robot sur son axe de déplacement.
On a donc :
VitLat = `e.getVelocity() * sinus (e.getHeadingRadians() - angleAbsoluAdverse)`
l'angle de visée dépendra aussi de la vitesse de l'obus (force 3, vitesse=11)
l'angle à corriger est donc :
= sinus-1 (VitLat*tempsTrajetObus) / VitObus*tempsTrajetObus)
= sinus-1(VitLat / VitObus)

Par simplification dans le cas present, avec un tir force 3, le calcul de l'angle l'angle peut se traduire approximativement par :
sinus -1(VitLat / VitObus) avec (VitLat / VitObus)=
`e.getVelocity() * Math.sin(e.getHeadingRadians() - angleAsoluAdverse) / 11.0)`
Cet angle calculé reste approximatif puisqu'il ne tient pas compte de la distance entre les deux robots, distance qui peut engendrer des déplacements plus importants de l'adversaire avant que l'obus arrive. Ce code assez léger convient largement pour des tirs à faible ou moyenne distance.

- ***Correction de l'angle de tir éloigné***

Ici, pour nos calculs, nous partons sur l'hypothèse qu'à l'instant où nous décidons de tirer un obus, le robot adverse effectuera une trajectoire rectiligne sans modifications de trajectoire telles que : s'arrêter, changer de sens, ou tourner...
Il est donc important de noter que si cette hypothèse n'est pas vérifiée, l'obus n'a aucune chance de rencontrer la cible visée, à moins qu'elle soit à courte distance, mais dans ce cas tous ces calculs ne se justifient pas.

Les données à prendre en compte :

- la puissance de feu peut être choisi de 0.1 à 3 maxi en sachant qu'elle ne peut pas être supérieure à l'énergie du robot.

```
puissanceDeFeuMaxi = Math.min(3.0, getEnergy());
```

- la vitesse de l'obus en fonction de la puissance de feu :

```
double VitesseObus =  20.0 - 3.0 * puissanceDeFeuMaxi ;
```

- les coordonnées de position de mon robot :

```
double positionX = getX();    double positionY = getY();
```

- angleAbsoluAdverse : pour nous, angle entre le robot adverse et le nord.

```
double angleAbsoluAdverse = getHeadingRadians() + e.getBearingRadians();
```

- l'orientation du châssis adverse par rapport au nord

```
double enemyHeading = e.getHeadingRadians();
```

- la vitesse du robot adverse

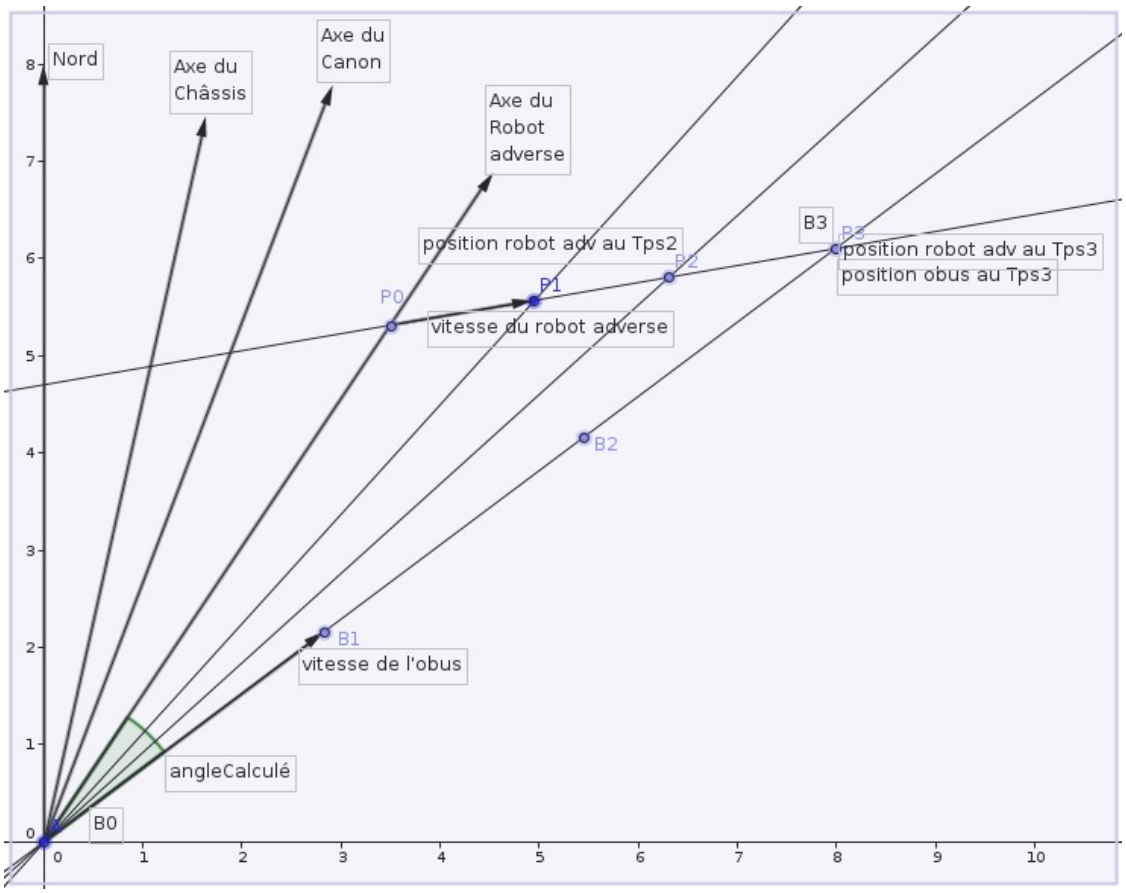
```
double enemyVelocity = e.getVelocity();
```

- les dimensions du champ de bataille

```
double hauteurChampBataille = getBattleFieldHeight();
double largeurChampBataille = getBattleFieldWidth();
```

- nombre d'unité de temps initialisé à 0 au départ du calcul

```
double uniteTemps = 0;
```



Algorithme de calcul du point de rencontre des trajectoires :

Au départ, unité de temps 0 , on calcule les coordonnées du robot adverse :

```
double enemyX = getX() + e.getDistance() * Math.sin(angleAbsoluAdverse);

double enemyY = getY() + e.getDistance() * Math.cos(angleAbsoluAdverse);
```

Les coordonnées prévues du robot adverse à chaque unité de temps sont initialisées au départ :

```
double prevuX = enemyX, prevuY = enemyY;
```

Les nouvelles coordonnées du robot adverse à la fin de la première unité de temps, unité de temps 1 sont :

```
prevuX += Math.sin(enemyHeading) * enemyVelocity;
prevuY += Math.cos(enemyHeading) * enemyVelocity;
```

On va incrémenter la variable temps par pas de 1 jusqu'à ce que la distance parcourue par l'obus tiré pendant ce temps, se rapproche le plus possible de la distance qu'il y aura entre les deux robots :
`(utilisation de import java.awt.geom.* pour la méthode Point2D.Double.Distance)`

Tant que la distance parcourue par l'obus est inférieure à la distance entre notre robot et la position ou se situera le robot adverse à cet instant là

```
while((++uniteTemps)*(20.0-3.0*puissanceFeu)
< Point2D.Double.distance(monX, monY, prevuX, prevuY)){
```

on recalcule les nouvelles coordonnées du robot avderse à la fin de l'unité temps

```
prevuX += Math.sin(enemyHeading) * enemyVelocity;
prevuY += Math.cos(enemyHeading) * enemyVelocity;
```

Si les coordonnées du robot adverse sortent des limites du champ de bataille

```
if(     prevuX < 18.0
        || prevuY < 18.0
        || prevuX > battleFieldWidth - 18.0
        || prevuY > battleFieldHeight – 18.0){
```

on bloque les coordonnées à la limite du champ de bataille sans en sortir

```
prevuX = Math.min(Math.max(18.0, prevuX),
             battleFieldWidth - 18.0);
prevuY = Math.min(Math.max(18.0, prevuY),
             battleFieldHeight - 18.0);
break; //on sort de la boucle car pas de rencontre possible.
 }
}
```

On calcule l'angle de tir en utilisant la tangente -1 pour retrouver l'angle à partir de la tangente (utilisation de la méthode à 2 paramètres de Math.Atan2)

```
double angleCalcule = Utils.normalAbsoluteAngle(Math.atan2(
    prevuX - getX(), prevuY - getY()));
```

On aligne le radar sur le robot adverse

```
setTurnRadarRightRadians(
Utils.normalRelativeAngle(angleAbsoluAdverse –
getRadarHeadingRadians()));
```

On oriente le canon avec la correction et on tire

```
setTurnGunRightRadians(Utils.normalRelativeAngle(angleCalcule -
getGunHeadingRadians()));
fire(puissanceDeFeu);
```

II.3 – Programmation d'un robot

II.3.1 - Version Charthiedor1

Stratégie recherchée pour la version 1 de notre robot

- Récupérer nos coordonnées X, Y au départ d'un round pour se sortir tout de suite de zone jugée éventuellement dangereuse : éviter de rester dans la zone centrale au milieu du champ de tir, éviter d'être coller au mur où l'on a moins de liberté de mouvements.

- Pour bien se défendre, être le plus souvent possible en déplacement

- Pour attaquer efficacement, ne pas gaspiller trop d'énergie en ratant les tirs, on évite de tirer de trop loin et l'on préfère se rapprocher du robot scanné

- Dès que l'on est placé à bonne distance, on oriente le canon vers le robot adverse et on tire tout en se déplaçant de manière à l'encercler.

- On inverse le sens de déplacement, lorsque notre robot est bloqué par un mur ou un autre robot, c'est à dire lorsque notre vitesse est égale à 0.

- Lorsque l'on percute un robot, on oriente le scan vers lui et on le prend comme cible.

Principes de bases utilisés (expliqués précédemment dans le rapport)

- Alignement du radar vers le robot adverse de manière à ne pas le perdre de vue.

```
double angleAlignementRadar =
getHeadingRadians() + e.getBearingRadians() -
getRadarHeadingRadians();
setTurnRadarRightRadians (Utils.normalRelativeAngle
(angleAlignementRadar));
```

- Se rapprocher jusqu'à 100 pixels du robot scanné s'il est à plus de 200 pixels

```
if (e.getDistance() > 200) ; setAhead(e.getDistance() - 100);
```

- Orienter le canon en direction du robot adverse

```
double angleAlignementCanon =
normalRelativeAngleDegrees(e.getBearing() + (getHeading() -
getRadarHeading()));
setTurnGunRight(angleAlignementCanon);
```

- Orienter le chassis vers la cible adverse

```
setTurnRight(e.getBearing()) ;
```

- Encercler le robot adverse

```
setTurnRight(e.getBearing() + 90);
```

```
setAhead(1000 * inverserDirection);
```

- Bloqué, inverser le sens de déplacement
```
if (getVelocity() == 0)
inverserDirection *= -1;
```

- correction de l'angle de visée en tir rapproché en fonction de la direction et de la vitesse du robot adverse
```
double angleAbsoluAdverse = getHeadingRadians() +
e.getBearingRadians();
setTurnGunRightRadians(Utils.normalRelativeAngle(angleAbsoluA
dverse - getGunHeadingRadians() + (e.getVelocity() *
Math.sin(e.getHeadingRadians() -angleAbsoluAdverse) / 11)));
```

Analyse des statistiques des combats

Les tableaux statistiques des résultats de cette première série de « battles » avec nos premières versions de robots montrent que notre char robotisé Chartiedor1 récoltent les meilleurs résultats sur les champs de batailles les plus petits (environ < à 1000 sur 1000) y compris avec d'autres robots proposés par l'application. Par contre dès que le champ de bataille s'agrandit, notre char est trop statique et la portée du radar insuffisante. Il peut être longtemps sans rencontrer d'adversaire et cela le pénalise énormément. Lorsqu'il est le meilleur, son score se répartit de façon équilibré entre les différentes catégories de gains (points survie, points obus, points collisions...)

Vérification de notre code

Contrairement à ce que nous voulions mettre en place pour cette version, et en s'apercevant que notre char tirait au départ d'un scan un obus n'importe où, nous avons corrigé notre oubli en rajoutant les méthodes qui désolidarisent le radar du canon et le canon du châssis.

II.3.2 - Version Chartiedor2

En tenant compte des statistiques des résultats des combats de nos versions 1, en fonction de l'analyse des robots adverses, nous allons tenter d'améliorer notre stratégie.

Stratégie recherchée pour la version 2 de notre robot

- Optimiser sa réaction face à un robot longeant les murs en prenant en compte le fait qu'il peut être à l'arrêt

- Ne pas le faire tirer uniquement de près, mais d'un peu plus loin force 1,1 (pour bénéficier d'un bonus), en essayant d'optimiser l'angle de tir.

- Le mettre en mouvement lorsqu'il ne scanne rien sur les grands champs de bataille

pour augmenter la probabilité d'en rencontrer d'autres.

- Lui imposer de changer de sens juste avant de percuter un mur

- Le faire avancer vers le robot scanné sans utiliser la trace directe

Principes de bases utilisés

- Se décaler d'une ligne de tir après 3 dommages subis par des obus.
- Vérifier que l'on ne soit pas trop près du bord

```
compteurHeurteParObus ++;
// la 3ème fois que l'on est touché par un obus
if(compteurHeurteParObus > 2)
{
compteurHeurteParObus = 0;
// si l'on n'est pas trop près d'un bord, on zig-zag
if((getX()>bord && getX()< largeurChampBataille-
bord)&&(getY()>bord && getY()< hauteurChampBataille-bord))
{
setTurnLeft(45);
setAhead(20);
setTurnRight(45);
}
else // sinon on tourne a 90° et on avance
{
setTurnRight(90);
setAhead(20);
}
}
```

- Optimiser les déplacements sur grand terrain lorsqu'il ne se passe rien pour rencontrer les autres robots.

```
// Boucle principale
while(true) {
turnRadarRight(10);//On tourne le radar vers la droite de 10
degrés
compteurNbDegre += 10;

// Si on à fait un tour complet sans avoir scanné ou heurté
// un ennemi, on bouge
if(compteurNbDegre > 360)
setAhead(getBattleFieldWidth()*0.01);
// Si notre vitesse est nulle en heurtant un mur, on tourne
if (getVelocity()==0) setTurnRight(120);
}
```

Stratégie non efficace et retirée

Nous souhaitions essayer d'éviter les chocs contre les murs (qui engrènent des pertes de vie) en définissant une marge tout autour du champ de bataille. Une fois dans cette zone dangereuse, on change la direction. L'écriture de nombreuses lignes de codes pour prendre en compte les différents bords percutés et l'orientation du châssis n'a pas montré d'amélioration sur les statistiques générales de combats avec les trois autres robots des groupes d'étudiants adverse. Alors nous avons retiré ces lignes de code. Notre robot perdait la cible au moment ou il s'éloignait des murs, ce qui est moins efficace que de toucher une fois le mur et s'écarter sans perdre la cible.

II.2.3 – Comparaison des deux versions

La comparaison des statistiques des résultats de combats entre CharthiedorV1 et CharthiedorV2 face aux autres robots doit permettre d'évaluer la pertinence de notre codage et de nos choix sur l'évolution stratégique recherchée.
De la même manière, pendant l'élaboration de notre version 2, nous réalisions régulièrement à chaque notion travaillée, des tests sur au moins 500 parties pour juger de manière pertinente de l'efficacité de notre codage.

Nous avons réalisé avec les deux versions de notre robot « Charthiedor » des combats de 1000 rounds dans grandeurs de champs de bataille contre les robots versions 1 des trois autres groupes, voici les résultats :

terrain de 600x600

CharthiedorV2

Results for 1000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	CTD.Charthiedor...	313694 (46 %)	126150	21750	143035	21200	1513	46	725	122	104
2nd	ml.Omega*	178625 (26 %)	74250	4470	89046	6800	3720	339	149	412	218
3rd	licpro.Groupe1*	113211 (17 %)	57900	3060	49019	2196	991	45	102	268	317
4th	jhr.Kfc*	69970 (10 %)	41500	720	23049	474	4099	128	24	201	359

Save OK

CharthiedorV1

Results for 1000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	CTD.Charthiedor...	260200 (36 %)	95950	10860	133973	16475	2879	64	364	299	231
2nd	ml.Omega*	259745 (36 %)	99550	13020	127660	13779	5168	568	435	262	163
3rd	licpro.Groupe1*	133374 (18 %)	61850	4710	62222	3087	1423	82	159	244	274
4th	jhr.Kfc*	77488 (11 %)	42500	1320	28079	665	4806	118	45	192	332

Save OK

terrain de 1000x1000

CharthiedorV2

Results for 1000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	CTD.Charthiedor...	324546 (50 %)	128800	22020	149072	23920	688	46	735	144	84
2nd	ml.Omega*	145043 (22 %)	62250	3780	71907	4854	2015	237	129	317	228
3rd	licpro.Groupe1*	109234 (17 %)	59800	2670	44269	1953	527	15	93	295	332
4th	jhr.Kfc*	69809 (11 %)	48900	1410	16851	328	2312	8	47	241	355

Save OK

CharthiedorV1

Results for 1000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	ml.Omega*	267345 (39 %)	106800	16920	123659	16180	3235	551	573	152	123
2nd	CTD.Charthiedor...	204848 (30 %)	75500	5730	108585	12678	2054	299	191	312	314
3rd	licpro.Groupe1*	129959 (19 %)	64500	4380	57348	2735	970	28	155	301	232
4th	jhr.Kfc*	83791 (12 %)	52650	2700	25335	818	2276	12	90	227	331

Save OK

terrain de 5000x5000

CharthiedorV2

Results for 1000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	CTD.Charthiedor...	212399 (39 %)	80650	8010	107785	15916	38	0	269	291	229
2nd	ml.Omega*	146096 (27 %)	69750	5160	62517	8341	288	40	172	272	338
3rd	jhr.Kfc*	97997 (18 %)	80150	10680	6857	230	80	0	380	151	190
4th	licpro.Groupe1*	89037 (16 %)	67700	5370	15376	558	34	0	205	266	240

Save OK

CharthiedorV1

Results for 1000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	ml.Omega*	200445 (39 %)	96400	11340	79220	12709	662	114	386	284	216
2nd	CTD.Charthiedor...	109335 (21 %)	53350	4680	44433	6093	758	20	213	126	243
3rd	licpro.Groupe1*	99871 (20 %)	70350	4590	23056	1542	323	10	211	278	295
4th	jhr.Kfc*	99113 (19 %)	74550	7080	15557	652	1162	112	276	244	231

Save OK

II.2.4 – Analyse des robots adverses

Analyse des robots adverses version1 et version2

	omega	kfc	groupe1
généralités	**Version1** AdvancedRobot il tire beaucoup et se protège avec les murs. **Version2** en se rapprochant des murs il oriente son canon du coté opposé au mur	**V1** classe Robot se déplace sans tenir compte des autres,tourne assez souvent son canon. **V2** AdvancedRobot le canon et le radar est désolidarisé du châssis	**V1** AdvancedRobot se déplace de manière saccadée en tournant beaucoup son canon. Ne cherche pas à se rapprocher des autres **V2** prend vite des infos avec son radar sur 360°et se fixe sur le robot scanné sans forcément cherché à se rapprocher
Déplacement initial	**V1** Se place en contact avec le mur, et tourne autour du champ de bataille vers la droite et fait tourner son canon sur 360°. **V2** idem mais oriente son canon opposé au mur	**V1** Avance s'il n'y a pas d'événements, et tourne son canon sur 360° dès un événement. Ou avance, tourne à gauche, tourne son canon sur 360°, recule et tourne encore son canon sur 360° **V2** tourne uniquement son radar de 45°	**V1** Tourne à droite d'un peu plus de 45° et tourne son canon sur 360° puis se déplace... **V2** de déplace en avant et inverse la direction lorsqu'il heurte un mur
Sur robot scanné	**V1** Oriente son canon sur le robot scanné et tire plus ou moins fort **V2** idem mais tire	**V1** S'il est près tire une fois à 3, s'il est moyennement distant, tire 3 fois à 2 s'il est loin tire à1 **V2** réajuste le radar sur sa cible	**V1** Tire de plus en plus fort s'il a beaucoup d'énergie **V2** **i**l tire beaucoup à

	avec l'énergie 3	et oriente le canon avec une visée corrigée. Tire à énergie 3. S'éloigne après avoir tiré si la distance est trop proche.	énergie 3 mais a une visée pas très optimisée.
Touché par un obus	**V1 et V2** À partir d'un certain nombre de touches, essaie d'éviter en faisant un zig-zag en continuant d'avancer	**V1** Recule et tourne à gauche **V2** rien de particulier	
Touche un mur	**V1 et V2** Tourne 90° à droite	**V1 et V2** Tourne pour éviter le mur	**V1** Tourne à droite moins de 90 degrés et avance **V2** il inverse la direction du déplacement
Touche un robot	**V1 et V2** Oriente son canon sur le robot et tire plein feu	**V1** Tire plein feu un certain nombre de fois **V2** rien de particulier	**V2** rien de particulier

III – Améliorations de l'application

III.1 Conception d'une interface

III.1.1 – Démarche et problèmes rencontrés

Tout d'abord, le premier travail demandé était de créer une interface simple qui puisse permettre de créer un robot, de lui donner un nom, de pouvoir définir ses couleurs dans son « design » et éventuellement un petit comportement à chaque tour, tout ceci en Java.
Ceci fut assez vite réalisé lors des deux premières semaines de notre projet. Ensuite nous nous sommes attaqués à un projet d'interface plus sophistiqué qui puisse permettre de choisir des événements et de pouvoir choisir parmi différentes réactions possibles à ces événements.

Notre première réflexion s'est portée sur le choix des options de l'interface. En effet il faut choisir les différents éléments que l'on souhaite ajouter à l'interface pour avoir un bon compromis entre trop d'options et pas assez d'options. Après discussion avec nos tuteurs nous avons défini les options utiles à l'interface. Cette dernière doit pouvoir créer un squelette de robot pour un utilisateur qui souhaite rentrer dans le code par la suite sans pour autant aller chercher partout des informations pour construire son robot. Nous avons donc retenu les options suivantes pour la création du robot :

- Le nom
- Les couleurs (unies ou personnalisées)
- Les actions du tour
- Les principaux événements possibles (scanner un robot, percuter un robot, un mur)
- Les différentes actions possibles pour chacun de ses événements

Avec ces options, l'utilisateur peut créer un robot viable avec un canevas clair.

La seconde réflexion s'est portée sur la génération de code valide et compilable après les différents choix opérés dans l'interface par l'utilisateur. Nous avons tout d'abord choisi de générer du code et de l'injecter directement dans un fichier texte. Nous nous sommes vite aperçu que ce système n'était pas optimisé, ni très réutilisable si des changements devaient être fait.

Nous avons donc recherché très rapidement une deuxième solution. Après quelques recherches, la solution de générer du code dans un fichier via un générateur de template s'imposait à nous. Et pour cela, nous avons choisi FreeMarker une librairie open source, libre et totalement écrite en java.

Le dernier point de notre réflexion concernait l'intégration de l'interface dans l'application Robocode. Comment intègre-t-on le code de l'interface dans l'application ? Des fichiers pêle-mêle ? Un .jar ? Plusieurs choix sont possibles, nous avons opté pour la solution la plus propre possible : la création d'un jar exécutable utilisable directement via l'interface proposé par Robocode. Avec ce choix nous avons pu intégrer notre interface en modifiant très peu le code source de Robocode. Il suffit d'aller dans le menu de création de robot de Robocode pour y avoir accès. Après les choix effectués et validation via notre interface, un dossier du nom du robot ainsi qu'un fichier .java est créé dans le dossier « robots » présent dans Robocode. Il suffit alors de le compiler pour ensuite pouvoir lancer un combat avec ce nouveau robot.

Le principal souci que nous avons rencontré pour cette intégration c'est trouver ou sont les fichiers java de l'interface de Robocode et surtout les chemins d'enregistrement des fichiers. Nous avons opté pour que cet enregistrement fonctionne peut importe où l'installation de Rocode a été faite.

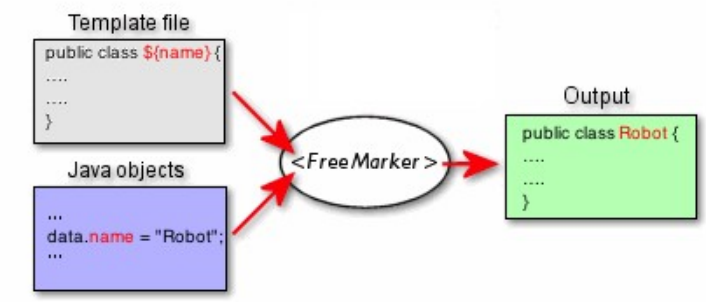
III.1.2– Les choix opérés : FreeMarker

Comment fonctionne FreeMarker ?

Pour arriver à produire un fichier java (sur le schémas « Output ») FreeMarker a besoin de deux éléments :

- Le fichier qui sert de modèle (sur le schéma « Template file ») : c'est un fichier qui définit les règles d'affichages, au format .ftl
- Les objets java (sur le schéma « Java objects », String, int, liste etc) : fichier des données à afficher.

Lorsque la transformation se fait, le « template » reçoit les objets puis FreeMarker interprète les données. Comme dans l'exemple ci-dessous :



Exemple de transformation

Ce système est très puissant et nous avons utilisé seulement une partie infime de ses possibilités. Même si, employer et mettre en place ce genre d'outils prend du temps, lorsque l'on veut faire des modifications on peut les faire très rapidement comparer à la première solution que nous avions choisi.
Les informations disponibles pour FreeMarker sont sur le site officiel de FreeMarler http://freemarker.sourceforge.net/. Un manuel plus des tutoriels sont disponibles à l'adresse suivante : http://freemarker.sourceforge.net/docs/index.html .

III.1.3 - Manuel d'utilisation de l'interface(en annexe)

Nous avons réalisé ce manuel pour expliquer tout d'abord comment lancer l'interface depuis l'application Robocode, ceci se faisant dans le menu Editor où nous avons rajouté des sous-menus. Ensuite, à l'aide de l'image de l'interface, nous avons pu expliquer dans ce manuel le rôle des différentes zones, les différentes étapes de création d'un robot.
Nous avons présenté les différentes fenêtres que l'on peut rencontrer lors de la validation, si les champs renseignés ne sont pas corrects.
Lorsque la validation peut avoir lieu, nous avons expliqué le chemin emprunté pour enregistrer le fichier généré.

Pour finir, nous avons expliqué comment récupérer le fichier de création de robot généré et surtout comment le compiler.

III.2 – Modifications du code source

III.2.1- Intégration de l'interface

Après avoir construit une interface pour créer des robots, il nous semblait intéressant de voir s'il était possible d'intégrer cette interface dans l'application. Pour cela, il fallait pouvoir accéder au code source et ceci est possible à partir de l'URL suivante : http://sourceforge.net/projects/robocode/files/robocode%20sources/
A partir du code source, pour intégrer notre interface, deux ajouts de code ont été écrits dans les fichiers `RobocodeEditor.java` et `RobocodeEditorMenuBar.java`.

Les modifications apportées sont :

- Ajout d'un menu "Interface" et de son sous-menu "Créer un robot via l'interface" qui lance le fichier interface.jar

Voici le chemin des deux fichiers qui ont été modifiés :

- création du menu et sous-menu
`codeSource/robocode.ui.editor/src/main/java/net/sf/robocode/ui/editor/RobocodeEditorMenuBar.java`
- action de lancer le .jar
`codeSource/robocode.ui.editor/src/main/java/net/sf/robocode/ui/editor/RobocodeEditor.java`

Tous nos ajouts de code sont entre les balises:

```
//********************************* DEBUT Ajout Projet Thierry, Dorian, Charly ************************************//
        Code ...
//********************************* FIN Ajout Projet Thierry, Dorian, Charly ************************************//
```

Nous avons créer le dossier `ressources` à la racine du dossier d'installation `robocode/` et nous avons placer dans ce dossier le fichier **interface.jar** de l'interface graphique que nous avons créée.

Il faut passer par cette étape pour avoir accès à toutes les librairies. Ensuite nous avons télécharger le code source de Robocode à l'url suivante : http://sourceforge.net/projects/robocode/files/robocode%20sources/ (Même version que celle du Robocode installé)

Nous avons placé le code source dans : `robocode/codeSource/` en créant ce répertoire.

On se retrouve donc avec tous les `.java` de Robocode. Une fois le `.java` trouvé et modifié, nous le compilons dans le même dossier avant de remplacer le `.class` dans le dossier `.jar` correspondant.

Lorque les `.class` sont remplacés, il suffit maintenant de lancer Robocode

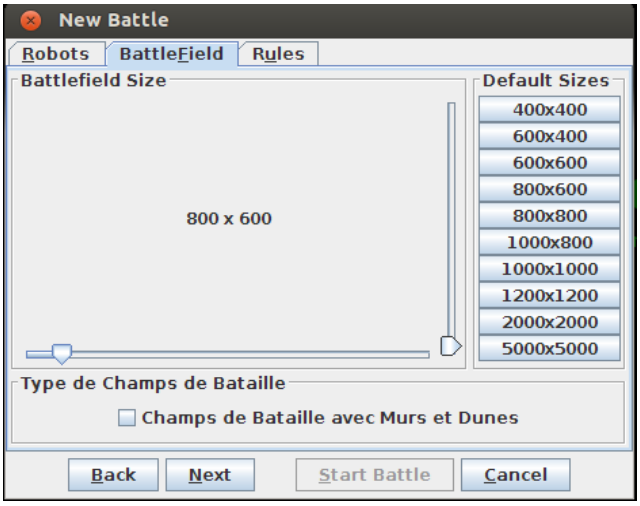
pour profiter des modifications.

III.2.2 – Modification du champ de bataille

Les images utilisées pour modifier le champ de bataille sont placés aussi dans le dossier `ressources/`.

Nous sommes allés dans le dossier libs du répertoire d'installation de robocode `robocode/libs` pour extraire toutes les trois archives `.jar` suivantes : `robocode.battle-1.7.4.2.jar` `robocode.ui.editor-1.7.4.2.jar` `robocode.ui-1.7.4.2.jar`

Nous avons rajouté une « checkbox » dans le menu New Battle / BattleField pour que l'utilisateur puisse choisir ou non de rajouter des obstacles.



Ajout d'obstacles dans le champ

Dans ce mode de jeu, nous avons rajouter sur le champ de bataille 2 obstacles rectangulaires (murs) qui stoppent les robots et les obus, et provoquent une perte d'énergie pour les robots qui viennent les percuter. Nous avons aussi expérimenté des formes circulaires et triangulaires.

Pour réaliser ceci, nous avons rajouter du code dans les fichiers suivants :

- La gestion des collisions et ralentissements des robots: `codeSource/robocode.battle/src/main/java/net/sf/robocode/battle/peer/RobotPeer.java`
- Pour la gestion des collisions des obus: `codeSource/robocode.battle/src/main/java/net/sf/robocode/battle/peer/BulletPeer.java`

Ajout de dunes de sable

Nous avons aussi rajouter 2 bandes de “dune ensablée” qui entraînent le ralentissement des robots lorsqu’ils passent dessus (la vitesse est réduite de moitié).

- Le dessin des dunes de sable: `codeSource/robocode.ui/src/main/java/net/sf/robocode/ui/battleview/BattleView.java`
- La gestion des ralentissements des robots: `codeSource/robocode.battle/src/main/java/net/sf/robocode/battle/peer/RobotPeer.java`

IV – Conclusion générale

IV.1- Les améliorations envisageables

Evolution possible de notre robot

Il est évident que la modification du champ de bataille va certainement influencer les stratégies recherchées. D'ailleurs nous pouvons le justifier en comparant les statistiques de résultats des combats avec champs sans obstacles et champs avec obstacles, et ceci avec les mêmes versions 2 des robots. Nous constatons que notre robot tel qu'il a été conçu pour champ de bataille sans obstacle est moins performant avec obstacles :

Tableaux de fin de batailles :

Results for 1000 rounds

Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	licpro.Groupe1_v3 2....	192903 (29 %)	90950	8940	83732	7797	926	557	308	354	202
2nd	CTD.CharthiedorV2*	192259 (29 %)	80300	8370	91705	11014	804	66	290	245	269
3rd	licpro.Kfc*	188645 (28 %)	72800	7950	94864	10741	2206	84	275	203	250
4th	ml.Omega*	97442 (15 %)	54250	4260	35761	2141	970	61	143	194	273

Save OK

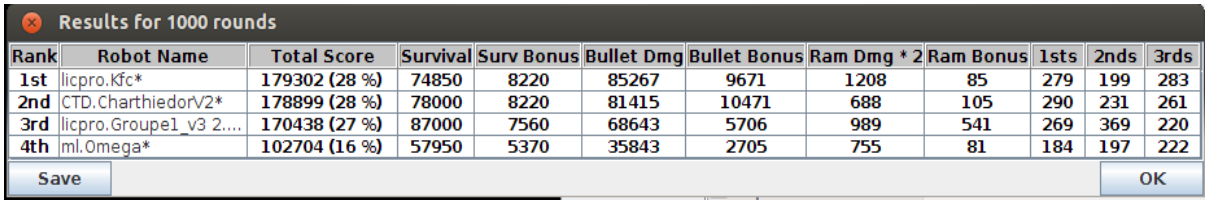
Champ 600x400 AVEC obstacles

Results for 1000 rounds

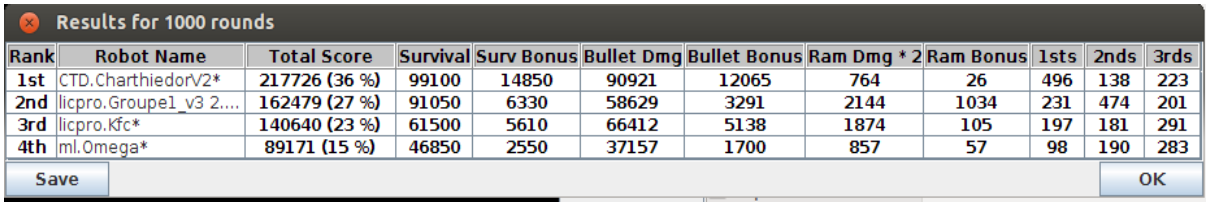
Rank	Robot Name	Total Score	Survival	Surv Bonus	Bullet Dmg	Bullet Bonus	Ram Dmg * 2	Ram Bonus	1sts	2nds	3rds
1st	CTD.CharthiedorV2*	202245 (32 %)	92450	11730	86724	9897	1375	69	396	214	240
2nd	licpro.Groupe1_v3 2....	188953 (30 %)	100700	10020	69056	4548	2921	1709	349	410	166
3rd	licpro.Kfc*	144152 (23 %)	59200	5070	71154	5301	3142	285	179	179	301
4th	ml.Omega*	94870 (15 %)	46450	2640	42094	2041	1423	222	95	181	292

Save OK

Champ 600x400 SANS obstacles



Champ 800x600 AVEC obstacles



Champ 800x600 SANS obstacles

Après observation au ralenti du comportement de notre robot et celui des autres groupes en situation de combats avec obstacles, nous constatons que toutes les stratégies sont à repenser. La traversée des dunes s'effectuent difficilement, les changements de trajectoires lorsque l'on percute un obstacle ne sont pas probants, les tirs de loin peu efficaces.
Plus de paramètres sont à prendre en compte et ils vont complexifier le codage, des nouvelles stratégies vont pouvoir se révéler comme se cacher, se protéger derrière un obstacle.

Modifications du champ de bataille

Nous avons réussi à changer le champ de bataille de manière assez figée en modifiant le code source pour choisir ou non ce mode de jeu. Dans le cas ou l'on choisit des obstacles, on pourrait imaginer que leur positionnement se fasse plutôt aléatoirement ou alors que l'utilisateur puisse choisir les dimensions et leurs emplacements.
On pourrait aussi envisager de proposer la même chose pour les zones de sables qui se placeraient aussi aléatoirement. Pour cela, il faudrait proposer une autre interface graphique pour dessiner le champ de bataille.

Evolution de l'interface

Comme nous avons pu le voir, proposer d'automatiser la génération de « bouts » de codes pour programmer des robots nous obligent à faire un compromis entre une multitude de fonctions très basiques qui alourdissent l'interface créée et des fonctions très évoluées qui ne donnent pas beaucoup de souplesse à l'utilisateur.
L'interface ne remplacera jamais les capacités que permettent l'écriture ligne par ligne du code avec l'utilisation de boucles `for`, `if`, `while`, l'utilisation de variables incrémentables qui génèrent elles-même de nouveaux événements.
Une profonde analyse de ce que pourrait proposer l'interface à l'utilisateur pourrait permettre de la faire évoluer.

IV.2- Conclusion

Ce projet était assez complet et permettait d'investir différents champs du métier de développeur d'applications : analyse d'applications existantes, programmation d'interfaces en langage Java, recherche et étude de librairies pour générer du code, analyse et développement de fonctions mathématiques, d'algorithme mathématique …
Pour ces raisons là, il nous a très vite passionné et intéressé. Il nous imposait de bien se répartir le travail en utilisant les compétences de chacun. Nous nous sommes aperçu d'une forte complémentarité entre nous.
Chacun par son investissement a progressé dans sa partie et a pu aussi en faire profiter les autres. Le rendement de nos apprentissages en a été fortement amélioré, la dynamique de groupe entretenu tout au long du projet sans baisse de motivation.
Aujourd'hui, nous pouvons exprimer tout les bienfaits du travail en groupe et nous pensons avoir développé de véritables compétences professionnelles dans ce domaine. Cela est bien fort à propos dans un cursus universitaire en Licence Professionnelle Informatique.