

UNIVERSITÉ DE BORDEAUX

PROJET DE PROGRAMMATION 2018

# Jeu de la Guerre

Mémoire

*Emile* BARJOU

*Christophe* CAUBET

*Adrien* HALNAUT

*Romain* ORDONEZ

Sujet proposé par

Philippe NARBEL

Supervisé par

Emmanuel FLEURY

## Résumé

Dans ce document, nous détaillerons la manière dont nous avons mené notre Projet de Programmation réalisé dans le cadre de notre première année de Master Informatique à l'Université de Bordeaux.

Ce sujet, proposé par M. Narbel, porte sur la compréhension, puis le développement d'une application permettant de jouer au *Jeu de la Guerre* tel que décrit par Guy Debord.

En vue d'une reprise potentielle du projet, un module d'analyse et une première approche de la réalisation d'un système expert en mesure de décider des coups, pouvant prendre la place d'un joueur réel, sont fournis.

## Abstract

In this document, we will explain how we led our Programming Project developed in the context of our Master degree in Computer Sciences at the University of Bordeaux.

This subject, given by Mr. Narbel, is about the understanding, and development of an application allowing to play «*Jeu de la Guerre*» as described by Guy Debord.

In the perspective of our project being continued, an analysis module and a first approach of the realisation of an expert system able to decide a move, that can replace a real player, are given.

# Remerciements

Nous souhaitons tout d'abord remercier M. Narbel de nous avoir proposé ce sujet, pour son temps et son investissement lors des nombreuses séances de discussions que nous avons pu avoir autour de ce projet.

Nous remercions aussi M. Fleury, pour son aide et sa patience lors de nos séances de TD ainsi que pour ses conseils avisés qui nous ont grandement aidé à mener à bien ce projet et ce document.

Aussi, nous souhaitons remercier le jury de l'audit, composé de M. Desbarat, M. Mansencal et de M. Lanterne, pour toutes les remarques constructives qu'ils nous ont fourni sur notre présentation et sur l'architecture de notre projet.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Domaine . . . . .	5
1.2	Sujet . . . . .	6
1.3	<i>Le Jeu de la Guerre</i> . . . . .	6
<b>2</b>	<b>Existant</b>	<b>10</b>
2.1	Assimilation du sujet et prototype . . . . .	10
2.2	Autres projets . . . . .	11
<b>3</b>	<b>Cahier des besoins</b>	<b>15</b>
3.1	Besoins primaires . . . . .	15
3.2	Besoins secondaires . . . . .	16
3.3	Besoins tertiaires . . . . .	16
3.4	Besoins non-fonctionnels . . . . .	18
<b>4</b>	<b>Présentation du logiciel</b>	<b>19</b>
4.1	Interface . . . . .	19
4.2	Commandes . . . . .	20
4.3	Messages . . . . .	21
<b>5</b>	<b>Architecture</b>	<b>23</b>
5.1	Vue d'ensemble . . . . .	23
5.2	Moteur de règles . . . . .	24
5.3	État du jeu . . . . .	25
5.4	Moteur de jeu . . . . .	26
5.5	Interface utilisateur . . . . .	27
5.6	Module d'analyse . . . . .	28
<b>6</b>	<b>Implémentation</b>	<b>30</b>
6.1	Modules principaux . . . . .	30
6.2	Moteur de jeu . . . . .	30
6.2.1	Jeu (Classe Game) . . . . .	30
6.2.2	Plateau de jeu (Classe Board) . . . . .	31
6.2.3	État du jeu (Classe GameState) . . . . .	32
6.2.4	Joueur (Interface Player) . . . . .	32
6.3	Moteur de règles . . . . .	35
6.3.1	Vérificateur de règles (Classe RuleChecker) . . . . .	35
6.3.2	Résultat d'une vérification (Classe RuleResult) . . . . .	35
6.3.3	Gestion des règles . . . . .	37
6.4	Interface utilisateur (Paquetage UI) . . . . .	41
6.5	Module d'analyse . . . . .	42
6.5.1	Présentation . . . . .	42
6.5.2	Recherches et observations sur le jeu et ses configurations . . . . .	43
6.5.3	Pistes d'exploitations possibles . . . . .	45
<b>7</b>	<b>Tests</b>	<b>46</b>
7.1	Les tests unitaires . . . . .	46

7.2	Les tests d'intégration . . . . .	47
7.3	Les tests système . . . . .	48
7.4	La couverture de code . . . . .	49
7.5	Tests des besoins . . . . .	49
<b>8</b>	<b>Conclusion</b>	<b>51</b>
8.1	Perspectives et extensions . . . . .	51
	<b>Bibliographie</b>	<b>53</b>
<b>A</b>		<b>55</b>

# 1

## Introduction

### 1.1 Domaine

Le *Jeu de la Guerre*[3] est un jeu de plateau imaginé par Guy Debord dans son livre éponyme.

Guy Debord (1931-1994) est un écrivain et philosophe situationniste principalement connu pour son ouvrage *Société du spectacle*(1967). Il crée la société "Les Jeux stratégiques et historiques" avec Gérard Lebovici, et imaginent les règles du *Jeu de la Guerre* puis les publient en 1987 dans *Le Jeu de la Guerre : Relevé des positions successives de toutes les forces au cours d'une partie*[3].

C'est un jeu au tour par tour qui est centré sur la stratégie militaire et les contraintes auxquelles font face les armées en temps de guerre. Il oppose deux armées ayant pour objectif de détruire les troupes adverses ou ses moyens de communication.

Guy Debord s'est inspiré du jeu *Kriegsspiel* (voir figure 1.1), créé par Von Reiswitz au XIX<sup>ème</sup> siècle, qui permet de simuler une guerre entre deux armées permettant d'affiner et de prévoir les tactiques militaires et les affrontements, tout en s'initiant à l'art de la guerre. Ce jeu a été utilisé pendant des décennies par des officiers et commandants, d'abord allemands puis de diverses nationalités, afin de pratiquer des batailles et de prévoir des stratégies d'action. Guy Debord s'est également inspiré de plusieurs des ouvrages de Carl von Clausewitz, dont le plus connu, *De la guerre* (ou *vom Kriege*)[2], écrits à la même époque qu'a été créée le *Kriegsspiel* de von Reiswitz[10].

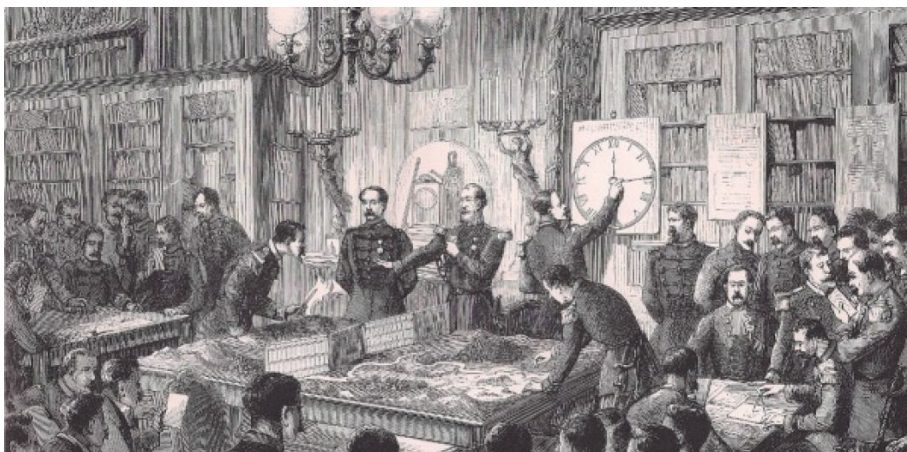


FIGURE 1.1 – "Le Jeu de la Guerre" (Kriegsspiel) dans *L'Illustration* du 22 août 1874

Les jeux de guerre, ou Wargames, sont le plus souvent des jeux de plateau faisant affronter des camps entre eux pour atteindre un objectif, que ce soit détruire entièrement les effectifs adverses ou bien atteindre une position clé sur le terrain.

De nos jours, ils sont assez répandus, même si la plupart d'entre eux s'adressent à des joueurs confirmés qui veulent principalement rejouer certaines batailles historiques en employant diverses stratégies. On peut notamment citer *DesertFox*, qui permet de rejouer certaines batailles datant de la Seconde Guerre Mondiale en Afrique du Nord, sur un plateau à cases hexagonales. Il est apprécié pour son niveau de complexité et de détails dans la gestion des armées et de la mise en pratique des stratégies.

Ce genre de jeu s'est surtout popularisé grâce à l'arrivée des jeux vidéo qui a permis une simplification dans la gestion des unités et un automatisme dans le calcul de certains mécanismes, l'un des plus répandus est *Civilization* qui met en compétition plusieurs joueurs, automatiques ou non, dans une course à l'expansion dans l'un des domaines permettant la victoire, que ce soit purement militaire ou scientifique et en passant par des victoires plus fines comme religieuse ou politique. Aussi, l'une des particularités de *Civilization* est qu'il joue avec les anachronismes, permettant de faire affronter Cléopâtre avec Roosevelt sur une carte totalement fictive.

D'autres jeux, comme la série *Cossacks* ou les *Total War*, ayant un niveau de gestion simplifié mais plus focalisé sur l'aspect militaire, visent le même public que *Civilization*, et d'autres comme la série *Wargame* se rapprochent plus de celui visé par *DesertFox*, mais sans le système de tour par tour.

## 1.2 Sujet

Ce sujet nous a été proposé par Philippe Narbel, avec qui nous avons pu préciser les besoins autour du projet, et encadré par Emmanuel Fleury avec qui nous avons pu discuter et critiquer nos choix d'implémentations. L'objectif est d'implémenter une version jouable du *Jeu de la Guerre* pour ensuite être en mesure d'analyser une partie.

Il s'agit tout d'abord de comprendre les règles décrites dans l'ouvrage de référence[3] pour pouvoir les implémenter dans le moteur de jeu, puis d'ajouter une interface pour rendre le jeu jouable. Enfin, nous devons extraire des informations pertinentes du jeu qui puissent être utilisées pour élaborer des stratégies, pouvant mener à l'implémentation d'un système expert capable de proposer des coups valides.

Lors de la création de tests, ceux-ci se basant sur des situations présentes dans ce livre, nous nous sommes rendus compte de quelques erreurs d'application des règles. Nous pensons que celles-ci viennent principalement du fait qu'il est assez complexe de décider d'une action dans ce jeu : les joueurs ont pu déplacer des unités pour mieux visualiser leurs actions pendant leur tour, et avant d'effectuer l'action qu'ils allaient réellement jouer, ont pu oublier de replacer correctement leurs unités. Nous pouvons noter une première erreur entre les positions 9 et 9', concernant le tour du camp Sud, où une infanterie se déplace de deux cases arrivant à la case I17, alors qu'elle ne devrait se déplacer que d'une seule case. D'autres erreurs ont été trouvées et répertoriées par Radical Software Games sur leur site Internet[6]. Il faudra cependant remarquer que l'erreur soulignée par Radical Software Games à propos de la position 46' de la version 2006 n'en est pas réellement une, car le tour est parfaitement jouable selon les règles du livre.

Étant un point crucial du projet, le moteur de règles doit être entièrement testé pour en garantir sa robustesse et son bon fonctionnement.

## 1.3 Le Jeu de la Guerre

Le projet se base sur le livre[3] écrit par Guy Debord, qui est donc notre principale référence, dans lequel est décrit le déroulement complet d'une partie servant d'introduction à son jeu, ainsi que la liste des règles de ce dernier (voir [3], pages 135-160).

C'est un jeu de plateau(1.3) historique, joué tour par tour, dont le but est de retranscrire les difficultés de mener une bataille, axée sur l'élaboration de stratégies et la mise en place de tactiques. Le jeu consiste essentiellement à déplacer ses unités et attaquer les unités et bâtiments adverses tout en protégeant les siens.

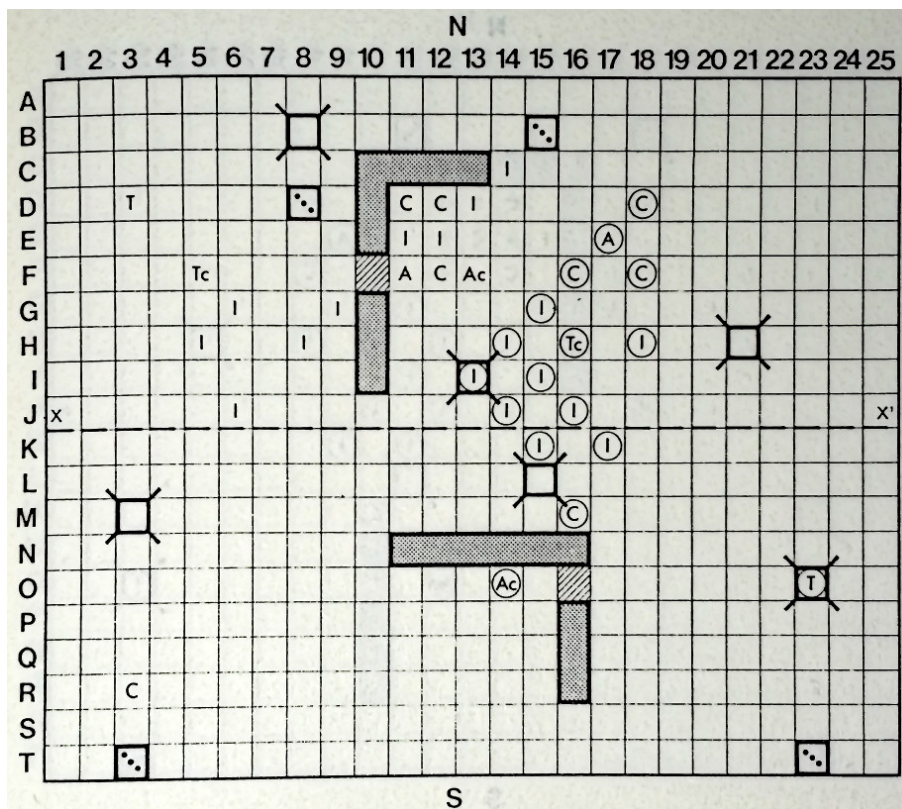


FIGURE 1.2 – Représentation du plateau de jeu décrit dans le livre du *Jeu de la Guerre*

## Éléments du plateau

Le plateau est composé de cases carrées, 25 de long, et 20 de hauteur, divisé en deux parties égales par une ligne sur le sens de la longueur. Chacune de ces divisions du plateau constitue l'un des deux camps adverses et possède 3 cases forteresses, 2 arsenaux, 9 montagnes et 1 col placé entre deux cases montagnes. Ces éléments ne peuvent se superposer sur une même case. La partie supérieure du plateau représente la zone associée au camp Nord, et la partie inférieure au camp Sud.

Les montagnes ne peuvent pas être occupées ou traversées par une unité, et une attaque ne peut être portée à travers celles-ci.

Les forteresses, ne pouvant être détruites, ainsi que les cols apportent un bonus défensif, respectivement de 4 et de 2, au coefficient de défense aux unités les occupant sans se soucier du côté du terrain où ils sont situés.

Un arsenal est considéré comme détruit lorsqu'une unité combattante du camp adverse a occupé sa case. Aucune unité adverse ne doit être présente sur la case de l'arsenal pour qu'il puisse être détruit.

Chaque camp possède également plusieurs types d'unités, définis par 4 caractéristiques : le coefficient d'attaque, le coefficient de défense, la portée et les points de mouvement.

## Initialisation de la partie

Les joueurs doivent disposer ensemble les montagnes et les cols sur le plateau, en respectant les contraintes précédentes. Aussi ces éléments ne doivent pas être disposés de manière symétrique par rapport à la ligne séparant les deux camps.

Les joueurs devront placer leurs bâtiments et leurs unités sur le terrain sans avoir connaissance du placement des effectifs adverses. Les forteresses doivent être posées sur des cases en communication directe (voir section [1.3 Communications](#)) avec au moins un arsenal allié. Le joueur qui



commence la partie est désigné de manière aléatoire.

### Déroulement d'un tour

À son tour, un joueur peut déplacer jusqu'à 5 de ses unités. Une unité ne peut se déplacer qu'une fois par tour, et elle ne peut se déplacer, attaquer ou défendre que si elle est alimentée en communication. À l'issue de chaque mouvement, le joueur peut attaquer une unité ennemie à portée d'attaque de l'unité qu'il vient de déplacer.

### Conditions de victoire

Il existe deux moyens de remporter la victoire, le premier est de détruire les deux arsenaux de l'adversaire, le deuxième est de détruire toutes les unités combattantes adverses.

### Communications

Les cases-Arsenal diffusent les lignes de communication dans les 8 directions (horizontales, verticales et diagonales), sans limitation de portée. La diffusion dans l'une des directions est stoppée dès lors qu'elle croise une unité ennemie combattante ou une montagne.

Les unités non-combattantes (les unités relais) alimentées en communication diffusent l'information de la même manière que les arsenaux. Les unités combattantes alimentées en communication diffusent l'information sur leurs 8 cases adjacentes, les unités se trouvant sur ces cases diffusent à leur tour l'information.

### Attaquer et défendre

- L'attaque se déroule sur une case définie par le joueur attaquant, à savoir la case où se trouve l'unité attaquée.
- Toutes les unités du joueur à portée d'attaquer cette même case additionnent leurs coefficients offensifs. Les unités du joueur défendant à portée d'attaquer cette même case additionnent leurs coefficients défensifs.
- Le résultat de la soustraction des coefficients offensifs par les coefficients défensifs permet trois situations :
  1. Le résultat est inférieur ou égal à 0, il ne se passe rien.
  2. Le résultat est supérieur ou égal à 2, l'unité attaquée est détruite.
  3. Le résultat est égal à 1, lors du prochain tour, l'adversaire devra, avant toute autre action, utiliser un de ses coups pour déplacer l'unité attaquée vers une case vide. Cette unité ne pourra participer à aucune offensive pendant ce prochain tour.
- Une unité ayant attaqué ne peut plus se déplacer jusqu'à la fin du tour.
- Une unité de cavalerie peut, à l'issue de son déplacement, initier une charge contre une unité ennemie se trouvant sur une des 8 cases adjacentes à une unité de cavalerie. Cette unité peut être la même que celle qui initie l'attaque, et doit se trouver sur l'axe reliant l'unité attaquée et celle attaquante sans discontinuité de cases entre elles. La charge donne un coefficient offensif de 7 à cette unité de cavalerie, et permet de faire participer à l'offensive les autres unités de cavalerie alliées se trouvant dans le même alignement de cases sans discontinuité de cases entre elles. Ce faisant, les unités de cavalerie participant à la charge voient leurs coefficients offensifs montés à 7 (voir figure 1.3).

	A	B	C	D	E	
1						
2		I				
3			C			
4				C		
5					C	

FIGURE 1.3 – Exemple d’une charge de 3 cavaliers sur l’unité en B2 initiée par le cavalier en E5

- Une charge ne peut être initiée contre une unité se situant sur un col de montagne ou dans un fort, et une unité se situant dans un fort ne peut y participer.
- La prise et destruction d’un arsenal ennemi est considérée comme une attaque, et l’unité à l’origine de cette prise ne peut donc pas initier d’offensive à l’issue de son déplacement.

## 2

# Existant

### 2.1 Assimilation du sujet et prototype

Les règles de ce jeu demandant un peu de temps à bien être comprises, nous avons décidé de faire quelques parties avant de se lancer dans le développement du programme, nous permettant de mieux comprendre les mécanismes du jeu et les enjeux stratégiques derrière celui-ci.

Cependant, nous nous sommes rendus compte assez vite qu'il n'existait pas de support jouable, ni plateau réel ou même virtuel. Nous avons donc décidé de faire un prototype papier (voir figures 2.1 et 2.2).



FIGURE 2.1 – Prototype papier du *Jeu de la Guerre*

Cette version nous a permis de découvrir la dimension tactique du jeu et que certains positionnements d'unités, notamment un groupe de 5 unités "en étoile" (Vu en détail en section 6.5.2 Référencessec :analyseEtoile), sont plus efficaces que d'autres dans l'optique de conserver un maximum de potentiel d'attaque et de défense tout en avançant vers les lignes adverses. Nous nous sommes également rendus compte lors de ces parties que nous passions la majeure partie du temps à vérifier l'alimentation des communications et à calculer les valeurs d'attaque et de défense.



FIGURE 2.2 – Prototype papier du *Jeu de la Guerre* - autre point de vue

## 2.2 Autres projets

Il existe d'autres projets autour du *Jeu de la Guerre*, l'un d'entre eux venant d'un ancien projet de programmation dans le cadre de cette UE.

### PdP 2014

Ce projet[4] permet la visualisation d'une instance d'une partie sans interaction possible avec le plateau. L'interface permet la visualisation des lignes de communication (voir figure 2.3), la portée de déplacement d'une unité ainsi que le potentiel offensif (voir figure 2.5) et défensif (voir figure 2.6) de chaque camp pour chaque case du plateau. Ce projet nous a donné des idées de fonctionnalités à implémenter concernant l'interface graphique pour l'utilisateur.

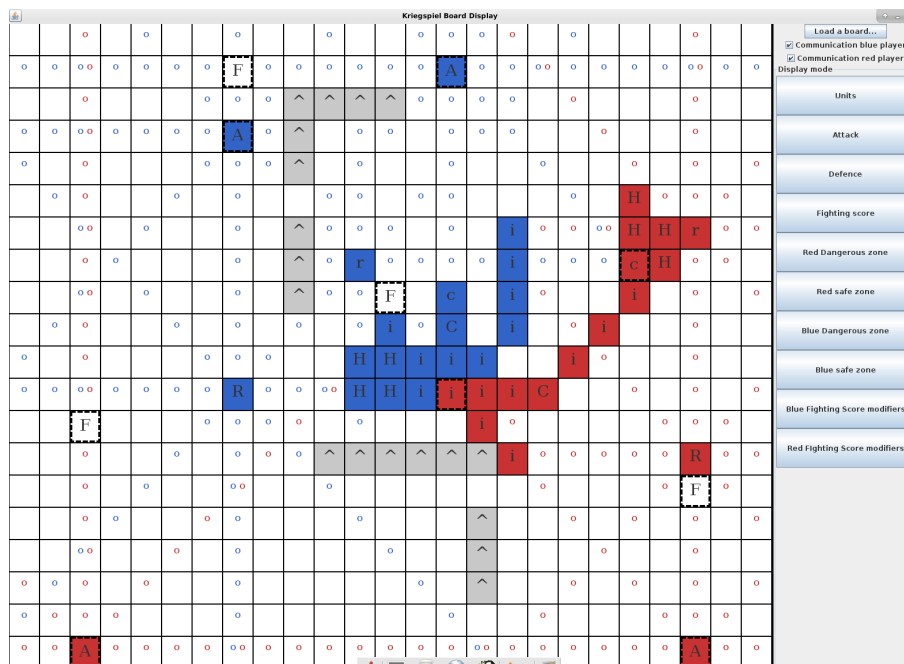


FIGURE 2.3 – Capture d’écran du programme développé du PdP 2014

### *Kriegspiel* par Radical Software Games[7]

Ce projet n’est plus disponible sur le site de Radical Software Games, mais nous disposons grâce à M. Narbel des sources du jeu. Il permettait de jouer en réseau contre un autre joueur au *Jeu de la Guerre* avec des règles simplifiées. Des aides visuelles sont présentes sur le plateau du jeu, permettant de voir les lignes de communication de chaque camp, les cases où peuvent se déplacer les unités et si ces cases sont à portée d’une ligne de communication (fig. 2.4). Il semble bien fonctionnel, mais a été retiré du site depuis. Il semblerait que ce soit la veuve de Guy Debord, Alice Becker-Ho, qui soit à l’origine du retrait[8] du jeu.

Cependant, selon un article datant de 2011 de *l’Express* [11], les archives de Guy Debord sont désormais la propriété de la Bibliothèque Nationale de France.



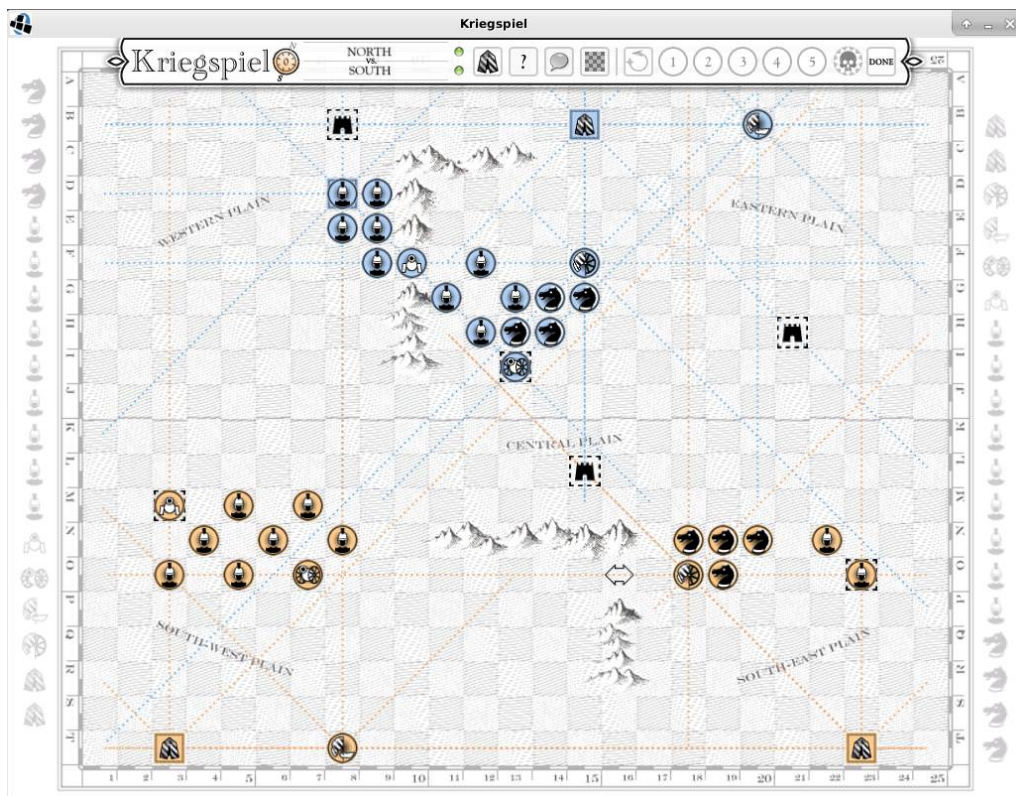


FIGURE 2.4 – Interface de *Kriegspiel* par Radical Software Games

## Conclusion

À notre connaissance, et après des recherches, nous pensons donc qu'il n'existe pas de version jouable du *Jeu de la Guerre* actuellement disponible. Cependant, il existe d'autres jeux semblables au jeu de Debord partageant le principe de simuler une guerre sur un plateau afin de revivre des batailles historiques et/ou de comprendre les tactiques derrière ces affrontements. On peut notamment citer *DesertFox*<sup>[1]</sup> qui était à l'origine le jeu proposé pour ce sujet, mais d'une complexité bien supérieure à celle du *Jeu de la Guerre*. Nous avons donc dû créer les bases de notre projet, mais les projets que nous avons cité nous ont tout de même servi dans notre réflexion sur comment mener à bien le notre, ainsi que sur quelques pistes notamment pour concevoir l'interface utilisateur.

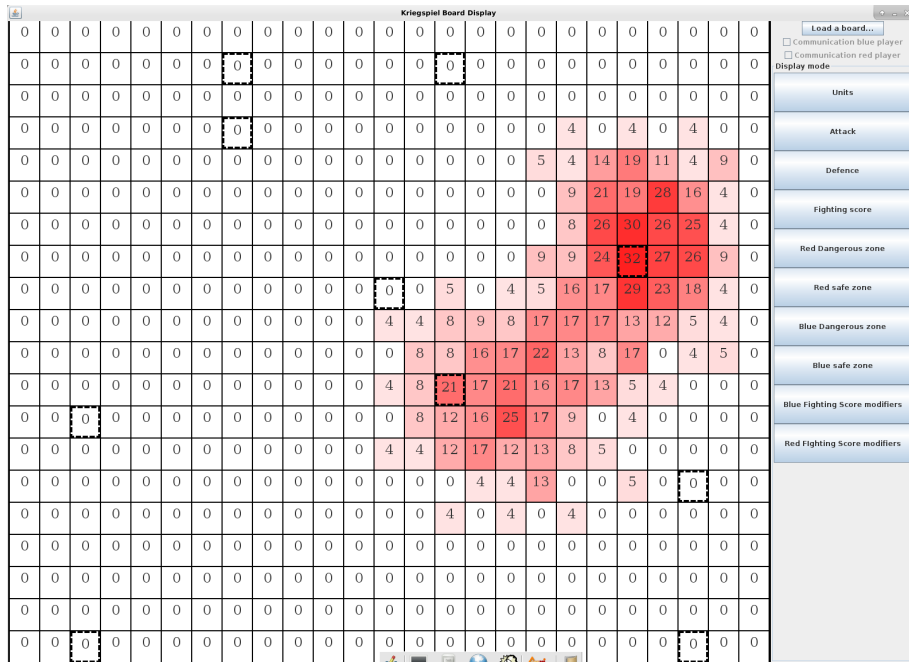


FIGURE 2.5 – Potentiel d'attaque des unités rouges (PdP 2014)

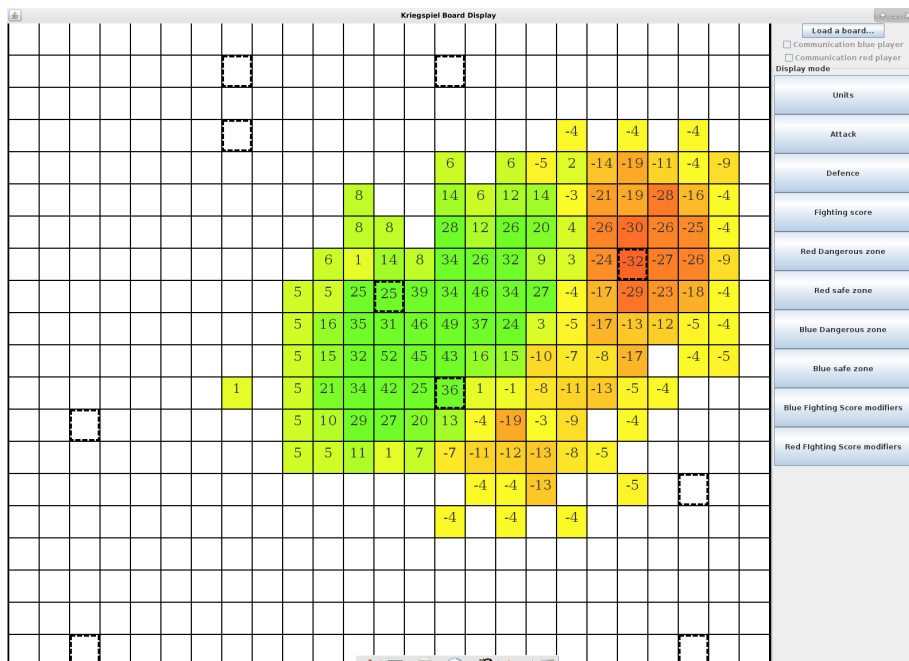


FIGURE 2.6 – Variation de potentiel de défense des unités bleues (PdP 2014)

## 3

# Cahier des besoins

La liste des besoins et leur ordre de priorité ont été établis avec le client M. Narbel et nous avons déterminé trois niveaux de besoins : primaire, secondaire et tertiaire.

Les besoins primaires sont la liste des services que notre programme doit impérativement fournir pour le rendu final du projet et qui doivent être testés au maximum pour en garantir la robustesse de l'implémentation.

Nous devons également implémenter les besoins secondaires autant que possible, ou à défaut en faciliter l'implémentation pour une possible reprise du projet. Enfin, les besoins tertiaires sont des pistes d'évolutions qui ne seront pas nécessairement finalisés, mais l'architecture du projet doit être adaptée à leur ajout.

### 3.1 Besoins primaires

1. Lire les commandes écrites par l'utilisateur dans un champ de saisie de texte et les traduire en coup (Détailé en figure 4.2) :
  - La position de chaque case est décrite par l'utilisateur à l'aide d'une ou plusieurs lettres suivies, sans espace, d'un nombre représentant respectivement l'axe des ordonnées et des abscisses. Ainsi, la case supérieure gauche est A1 et la case inférieure droite est T25.
  - Déplacer une unité sur une case ciblée vers une autre case : **move** <C1> <C2> où C1 est la case de l'unité alliée ciblée, C2 la case d'arrivée
  - Attaquer une case en ciblant la case de l'unité attaquante : **attack** <C1> <C2> où C1 est la case de l'unité alliée ciblée, C2 la case où se situe l'unité ennemie attaquée.
  - Déclarer la fin de son tour : **end**
2. Vérifier qu'un coup est valable pour l'état de jeu actuel à l'aide d'un moteur de règles implémentant celles décrites dans le livre *Jeu de la Guerre*.
  - Vérifier que la commande est bien demandée par le joueur dont c'est le tour.
  - Vérifier que le joueur peut encore effectuer des actions pendant ce tour.
  - Vérifier les règles spécifiques aux mouvements et attaques.
  - Vérifier qu'une retraite ne doit pas être réalisée avant tout autre coup.
3. Appliquer un coup validé en modifiant l'état de jeu actuel.
4. Simuler un coup sur une instance d'un plateau sans le modifier pour aider un joueur automatique.
5. Permettre à deux utilisateurs de jouer sur une même interface graphique à tour de rôle.
6. Afficher l'état du plateau actuel, comprenant les unités et les bâtiments à l'aide d'icônes différenciables. (voir figure 3.2)
7. Afficher le joueur dont c'est le tour.
8. Afficher le nombre de coups restant au joueur ce tour-ci.
9. Déterminer et afficher les informations de fin de partie et du vainqueur.



## 3.2 Besoins secondaires

1. Lorsqu'un coup est refusé par le moteur de règles, afficher une explication du refus.
2. Générer par un clique gauche, dans la ligne de commande, les coordonnées correspondant à la case cliquée par l'utilisateur dans le format attendu par le jeu.
3. Sauvegarder l'état du plateau dans un fichier texte, en codage ASCII, en respectant un ordre logique des informations stockées, le fichier sera lisible par l'utilisateur, dans le format détaillé en figure 3.1. Cette action sera possible à l'aide de la commande : `save <file>`. Un exemple d'un tel fichier est donné en annexe A.1.

FIGURE 3.1 – Format du fichier de sauvegarde

```
<Largeur>;<Hauteur>
<Joueur actuel>;<Nombre de coup restant>
<ID d'un terrain>;<X>;<Y>;<Joueur>
...
<ID d'une unité>;<X>;<Y>;<Prioritaire ?>;<Peut bouger ?>;<Dernière unité déplacée ?>;<Peut attaquer ?>;<Joueur>
...
```

4. Instancier une partie à partir d'un fichier texte valide, en codage ASCII, respectant l'ordre et la syntaxe décrits par la sauvegarde. Cette action sera possible à l'aide de la commande : `load <file>`.
5. Annuler la dernière action effectuée par l'utilisateur à l'aide de la commande : `revert`.
6. Extraire des informations sur l'état du jeu telles que les lignes de communications, la carte des potentiels d'attaque, de défense ou de déplacement.
7. Afficher sur le plateau des aides visuelles à l'aide de couleurs pour mettre en avant les informations extraites.
8. Choisir quel type et de quel joueur afficher les informations à l'aide d'un menu.
9. Afficher une légende des unités et bâtiments, comprenant leurs icônes et leurs attributs.
10. Pouvoir positionner sur le plateau les unités à l'aide d'une commande dédiée lors de la phase de positionnement des unités.
11. Pouvoir remplacer un joueur humain par un joueur automatique pouvant générer des coups pendant son tour.

## 3.3 Besoins tertiaires

1. Évaluer ("Comprendre") une situation du jeu à un moment donné, (Spatial Reasoning[9] - Potential fields - PathFinding) Détermination de "points décisifs".
2. Fournir une représentation visuelle de ces évaluations en plus des informations extraites du jeu.
3. Évaluer ("Comprendre") une situation du jeu sur quelques coups ou tours. Prise en compte de techniques de planification (Planning). Représentation graphique de cette évaluation.
4. Élaborer et mettre en place une stratégie permettant de trouver le chemin le plus efficace pour atteindre les arsenaux de l'adversaire (analyse des "vallées" ou faiblesses dans la formation adverse pour atteindre l'objectif).
5. Élaborer et mettre en place une tactique permettant d'obtenir une configuration optimale dans la planification d'une attaque ou d'une défense (utilisation d'une cartographie des potentiels d'attaque et de défense).

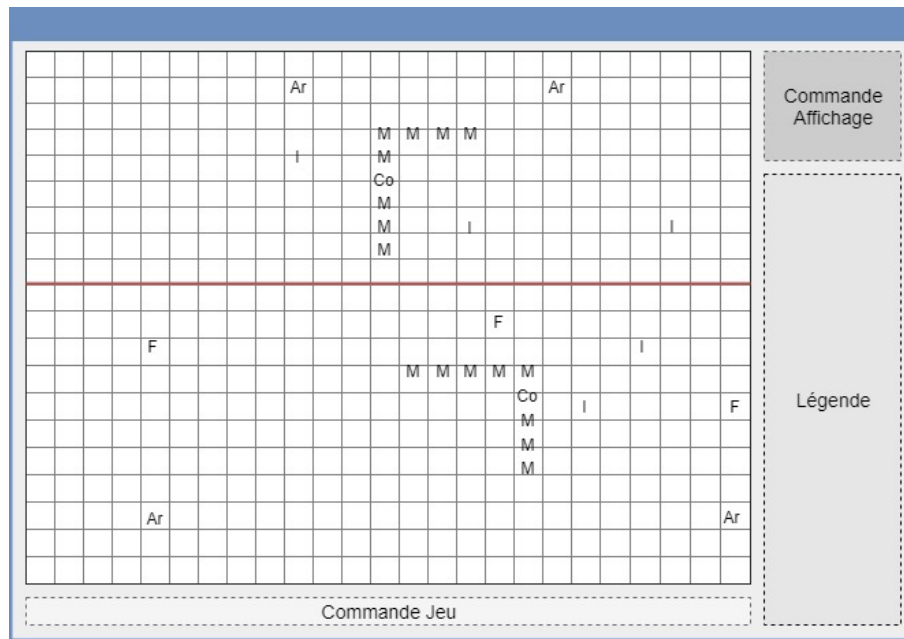


FIGURE 3.2 – Ébauche d'interface commande et d'aperçu de l'état du plateau

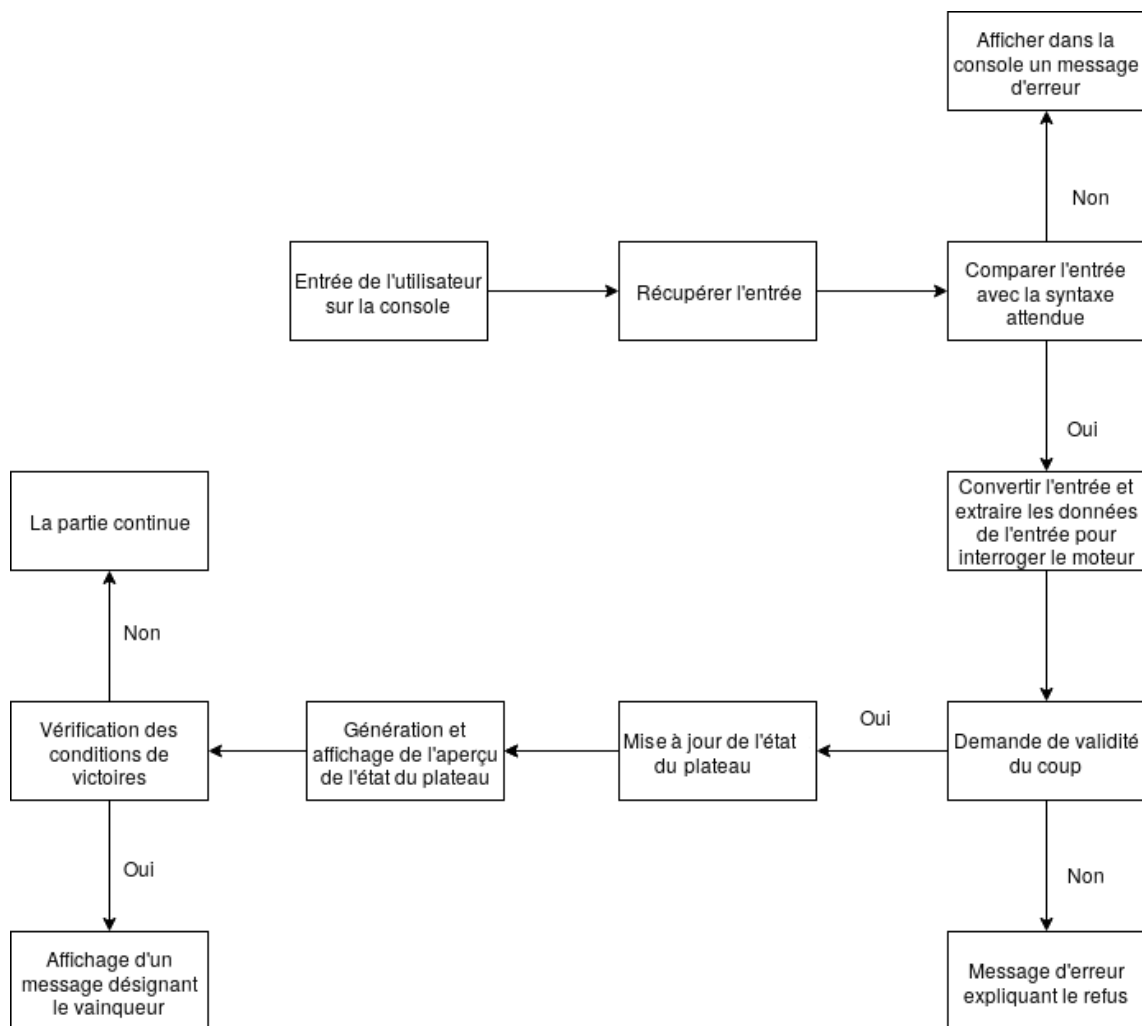


FIGURE 3.3 – Scénario d'utilisation pour un coup proposé

### 3.4 Besoins non-fonctionnels

1. Le code du moteur de règles devra avoir un certain degré de généricité et de modularité pour pouvoir altérer les règles du jeu.
2. Le temps d'attente entre un coup proposé et sa validité évaluée par le moteur de règles devra être de l'ordre de la seconde.
3. Le jeu doit fonctionner sur un système UNIX.
4. Le jeu sera codé dans le langage de programmation Java.
5. Une couverture des tests sur les différents modules du jeu est attendue lors du développement. Ces tests devront être implémentés par les développeurs travaillant sur leurs modules au fur et à mesure.
6. Les actions de déplacement, d'attaque et de fin de tour devront être testées, aussi bien pour des actions valides qu'invalides, pour une instance du plateau donnée. De plus, certains de ces tests devront inclure des instances continues de la partie de Guy Debord où le mécanisme testé est présent.
7. Des tests en boîte noire devront être fournis, permettant de valider les interactions entre l'utilisateur et le programme. Certains de ces tests se baseront sur les configurations du plateau de la partie décrite dans le livre de Debord (voir [3], pages 14-24), notamment pour vérifier si l'état du plateau résultant du test sur l'état initial est bien conforme à ce qui est décrit. Cette partie servira aussi de témoin principal pour confirmer la validité des règles implémentées.

## 4

# Présentation du logiciel

## 4.1 Interface

Le logiciel se présente sous la forme d'une fenêtre graphique (voir figure 4.1) comprenant l'affichage du plateau de jeu, un champ de texte pour y entrer les commandes, une légende et un menu pour indiquer quel type d'information afficher sur le plateau (par exemple les lignes de communications, les potentiels d'attaque ou de défense, voir section 6.5 Module d'analyse).

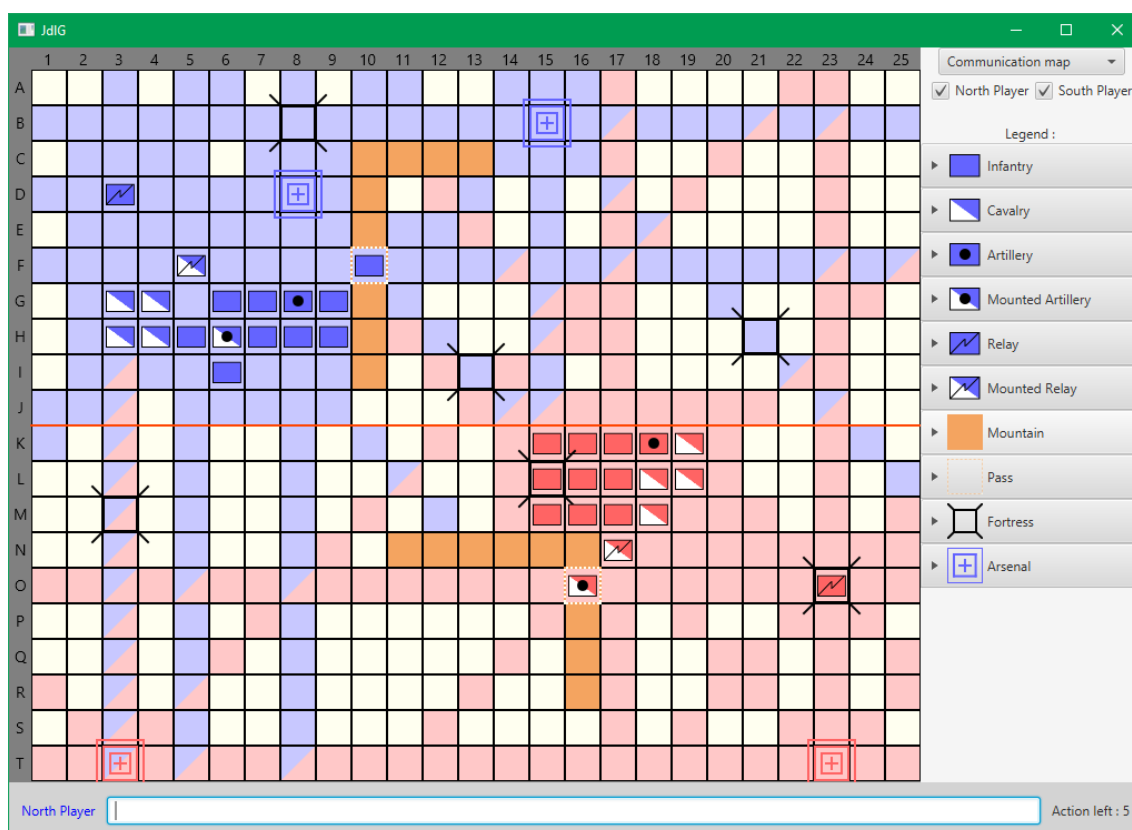


FIGURE 4.1 – Interface graphique du logiciel

Le plateau est la partie centrale de la fenêtre, et est redessiné dynamiquement lors du redimensionnement de celle-ci. Dessus y sont dessinées les cases avec leurs unités et bâtiments, et l'information choisie (par défaut les lignes de communications).

La barre de commande sur la partie inférieure de la fenêtre comporte, en plus du champ de texte, le nom du joueur auquel c'est au tour de jouer et du nombre de coups qu'il lui reste.

Enfin le panneau latéral comporte en haut le menu déroulant pour l'information ainsi que deux cases à cocher pour choisir de quel joueur afficher cette information. En dessous se trouve la légende faisant correspondre les unités et leurs icônes, qui peut être étendue pour afficher les informations relatives à l'unité.

Lors du lancement du logiciel, la partie commence directement, dans la configuration initiale de la partie du livre de référence.

Commandes de jeu	
Ecriture d'une coordonnée : C12, B6, A20 De A1 (coin sup. gauche) à T25 (coin inf. droit)	
Mouvement de l'unité sur la case C1 vers C2	move <C1> <C2>
Attaque par l'unité sur la case C1 de C2	attack <C1> <C2>
Termine le tour du joueur courant	end
Commandes générales	
Afficher la liste des commandes	help
Sauvegarder la partie dans le fichier FILE	save <FILE>
Charger la partie depuis le fichier FILE	load <FILE>
Annuler les dernières commandes	revert











Nom complet	symbole	Icône
Infanterie	I	
Cavalier	C	
Artillerie	A	
Artillerie montée	Ac	
Communication	R	
Communication montée	Rc	
Montagne	M	
Col	Co	
Fort	F	
Arsenal	Ar	

FIGURE 4.2 – Commandes et identifiants des unités et terrains employés

## 4.2 Commandes

Les commandes disponibles pour la barre de commande sont :

- **help** : Affiche la liste des commandes disponibles. La syntaxe de ces commandes est aussi référencée en figure 4.2.
- **load <file>** : Permet de charger une configuration de partie à partir du fichier **file**. Le format de tels fichiers est détaillé en figure 3.1.
- **save <file>** : Permet de sauvegarder la configuration de partie actuelle dans le fichier **file**.
- **move <c1> <c2>** : Demande de déplacer l'unité située en case **c1** à la case **c2**. Les cases sont référencées en <ordonnée><abscisse> avec **ordonnée** une lettre et **abscisse** un nombre. Cette commande peut aussi être raccourcie en '**m <c1> <c2>**' ou même '<c1> <c2>'. Cette commande sert aussi pour la prise d'arsenal en déplaçant une unité alliée dessus.
- **attack <c1> <c2>** : Demande d'attaquer l'unité située en case **c2** avec l'unité située en case **c1**. L'utilisation de la charge est automatique (si faisable) avec cette commande. Cette

- commande peut être raccourcie par '`a <c1> <c2>`'.
  - **revert** : Demande à annuler le coup joué précédemment, cela peut être un mouvement, une attaque ou même un changement de joueur.
  - **end** : Termine le tour pour laisser la main à l'autre joueur.
  - **exit** : Ferme le programme.

Pour faciliter l'écriture des coordonnées, un clic sur une case du plateau permet de concaténer ses coordonnées à la commande. On notera qu'un déplacement peut donc être effectué en cliquant successivement sur deux cases du plateau, avant d'appuyer sur Entrée.

## Lancement d'un joueur automatique

Nous avons ajouté la possibilité de lancer le programme avec un ou les deux joueurs remplacé par un *bot*<sup>1</sup>. Le système pour proposer des coup est simpliste, se contentant de dresser la liste des coups possible et d'en choisir un au hasard ; cet ajout sert surtout à donner un exemple sur où implémenter un système expert par la suite, et comment le faire interagir avec le module d'analyse.

L'utilisation de bots passe par des lignes de commandes, `-bot1` pour remplacer le second joueur par un bot, et `-bot2` pour que les deux joueurs soient des bots. Un bot joue un coup lorsque l'on envoie une commande vide à son tour, il faut donc appuyer sur Entrée lors du tour du bot et il jouera un coup, ou mettra fin à son tour si il n'a plus de coups à jouer.

## 4.3 Messages

### Messages d'erreur

Si une commande entrée est incorrecte, un message d'erreur, aussi détaillé que possible (voir figure 4.3), expliquera les raisons pour lesquelles le coup a été refusé. Cela fonctionne pour une erreur au sein de la vérification de la validité du coup, ou pour une erreur sur le traitement de la commande.

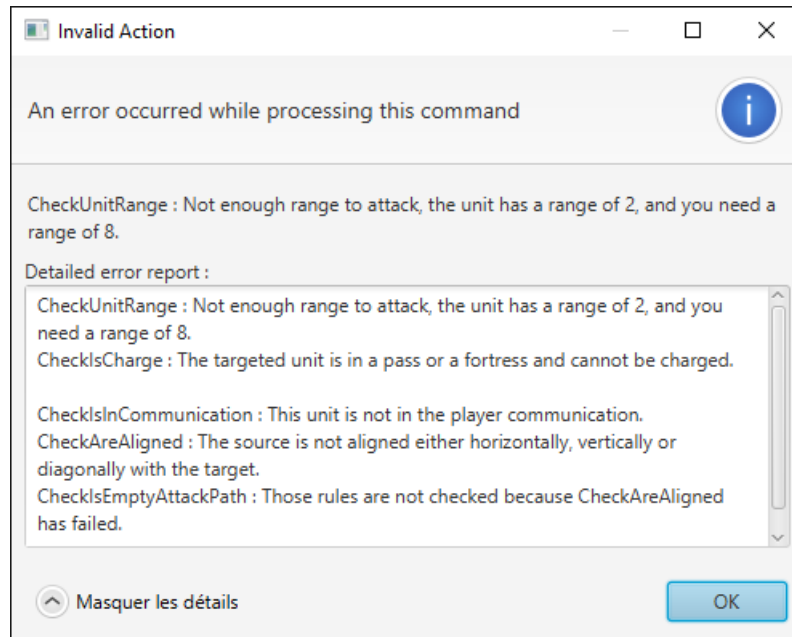


FIGURE 4.3 – Message d'erreur d'une commande d'attaque invalide

1. Joueur automatique.

## Résultats d'affrontement

Après une attaque, un message affichant le résultat de l'affrontement obtenu s'affiche, qu'il soit réussi ou non (voir figure 4.4).

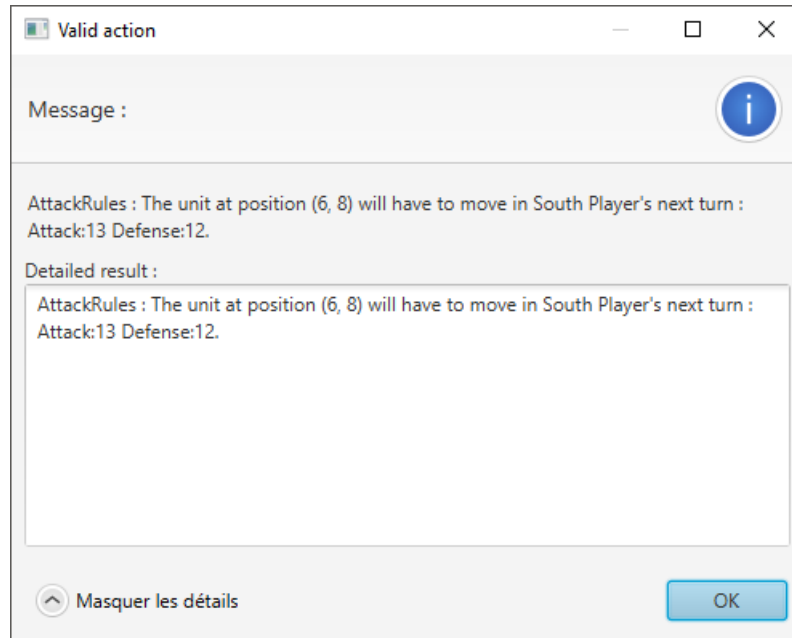


FIGURE 4.4 – Message d'information à la suite d'une attaque

## Victoire

Quand l'action d'un joueur mène à sa victoire, un simple message s'affiche pour l'en informer (voir figure 4.5).

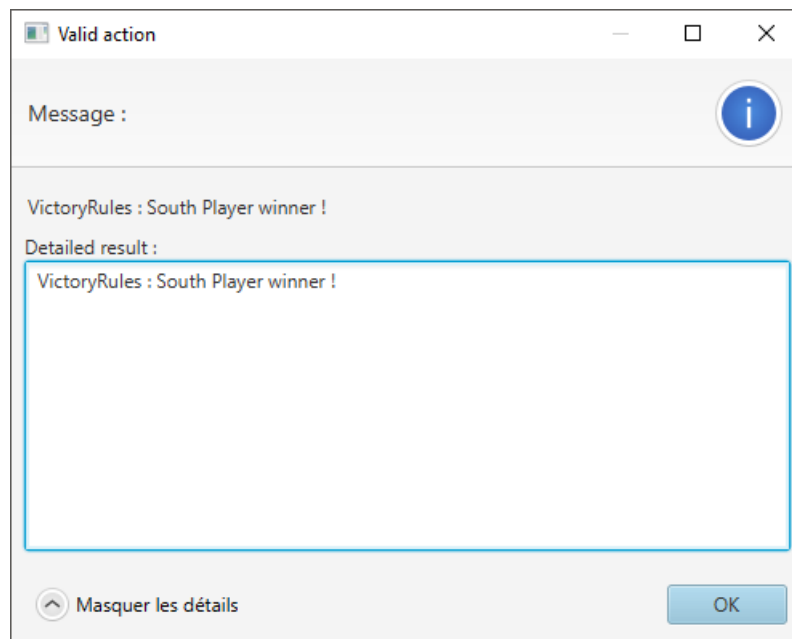


FIGURE 4.5 – Message d'information à la suite d'une attaque

## 5

# Architecture

## 5.1 Vue d'ensemble

Lors de notre réflexion sur la structure générale du projet, nous avons très rapidement défini, avec notre chargé de TD, cinq modules principaux qui divisent les grandes lignes de notre projet. (voir 5.1)

1. Le moteur de règles, permettant de vérifier la validité d'un coup et de l'appliquer selon les règles décrites dans *Jeu de la Guerre*.
2. L'état du jeu, qui stocke toutes les données nécessaires au déroulement de la partie et gère les accès au plateau.
3. Le moteur de jeu, qui est le cœur du programme. Il détient la boucle principale, et lie tous les autres modules entre eux.
4. L'interface "joueur", qui représente un joueur réel ou artificiel.
5. Le module d'analyse, qui construit, à partir d'un état de jeu, des cartes d'informations diverses qui seront utiles autant pour un joueur réel que pour un système expert.

Une documentation Javadoc est disponible concernant notre projet. Nous avons également utilisé l'outil Doxygen qui permet d'avoir, en plus de la génération de la documentation, une représentation plus visuelle de l'architecture et des dépendances entre les différents composants de notre projet.

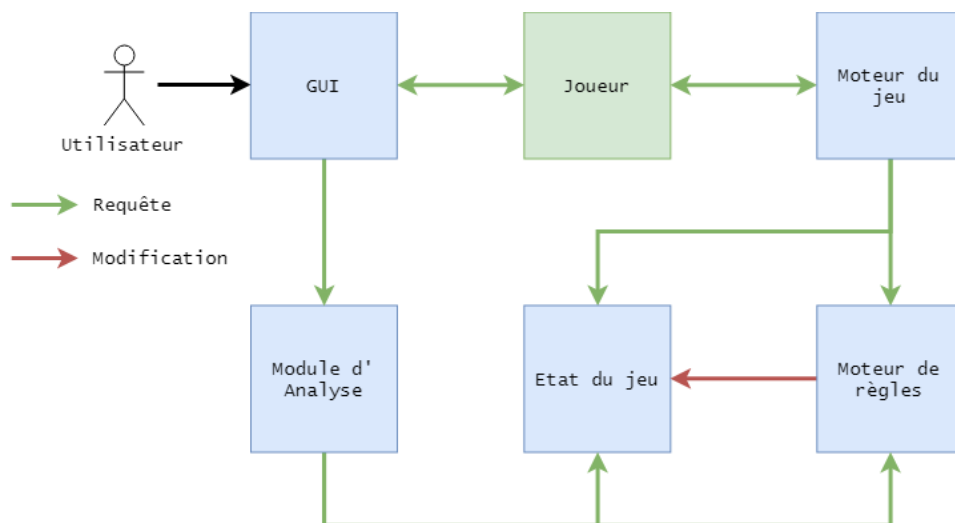


FIGURE 5.1 – Interactions au sein de l'architecture du programme



## 5.2 Moteur de règles

L'un des objectifs premiers lors de la conception du moteur de règles a été de le rendre le moins dépendant possible du reste du jeu. Le moteur de règles se divise en trois parties, **RuleChecker** qui s'occupe d'associer les actions à une des règles et de la tester, cette classe représente le point d'entrée du module utilisé par le jeu et par le module d'analyse. Nous avons aussi une classe représentant le résultat produit par la vérification des règles, à savoir le **RuleResult** qui sera transmis à l'issue d'une requête, et enfin la partie concernant les règles et leurs associations.

Pour faciliter la communication avec les autres modules, **RuleChecker** utilise une énumération, **EGameActionType**, qui représente chaque mécanisme décrit par le *Jeu de la Guerre*. La figure 5.2 représente un schéma UML des interactions entre le **RuleChecker** et les autres modules, ainsi que les classes du moteur de règles.

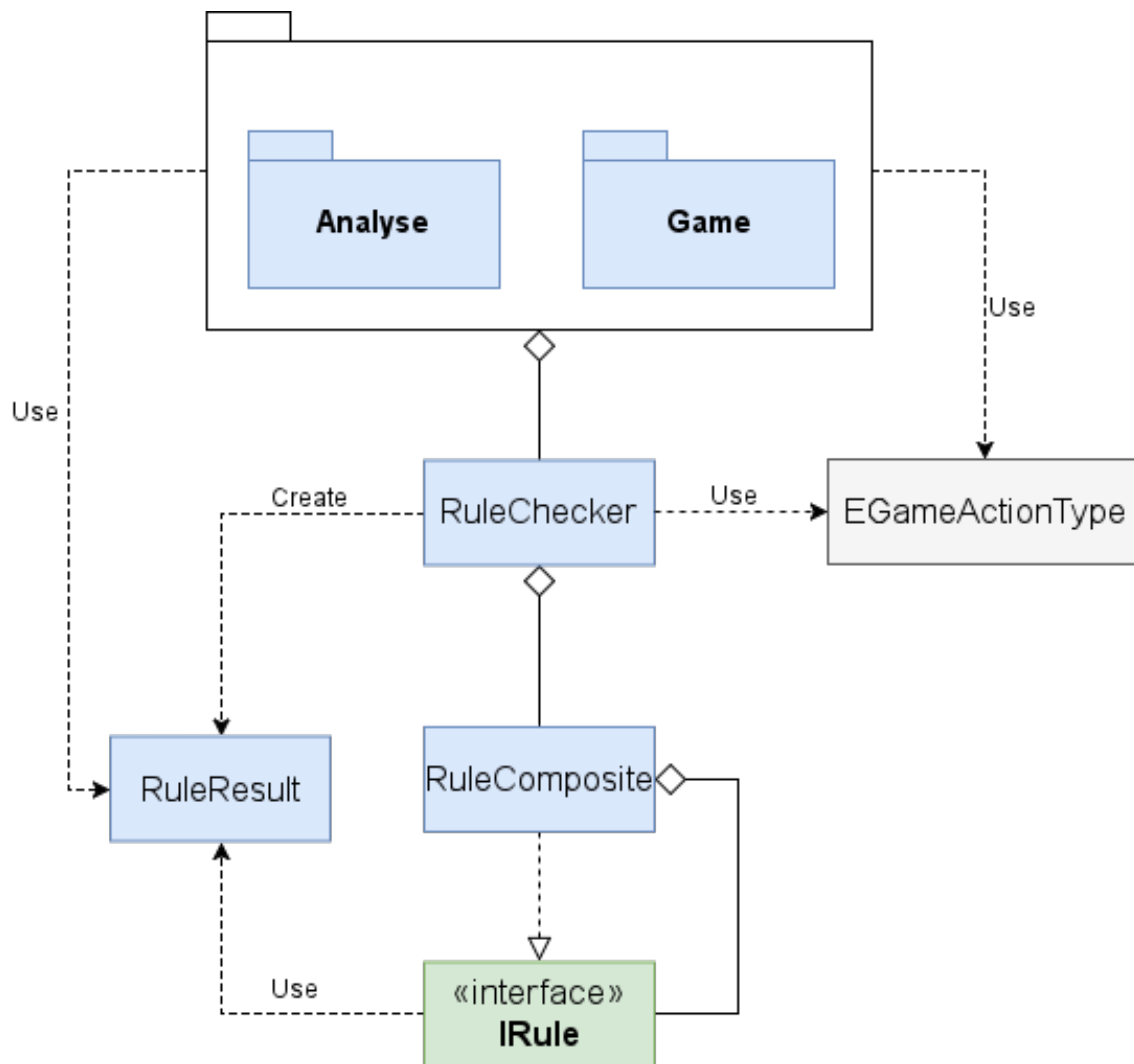


FIGURE 5.2 – Schéma UML du moteur de règles

Afin de conserver une certaine cohérence entre les sous-règles, un système de dépendances entre elles a dû être mis en place. Pour répondre à ce besoin, nous avons utilisé le patron de conception *Composite* (voir figure 5.3). Le Composite nous permet d'avoir une structure d'arbre, où nous ne nous soucions pas de savoir si les règles partageant l'interface **IRule** sont des sous-règles, des règles ou des groupes de règles.

Dans notre cas, ce patron nous permet de générer une structure d'arbre, où la partie composite du pattern, à savoir les nœuds, représente les opérateurs logiques *ET*, *OU* et *NON*, et leurs équi-

valents paresseux servant de dépendances, et où les feuilles sont des sous-règles. Ainsi les règles, ou *MasterRules*, étendent un des opérateurs logiques et spécialisent la méthode `applyResult()` afin d'appliquer au jeu les modifications associées à la règle.

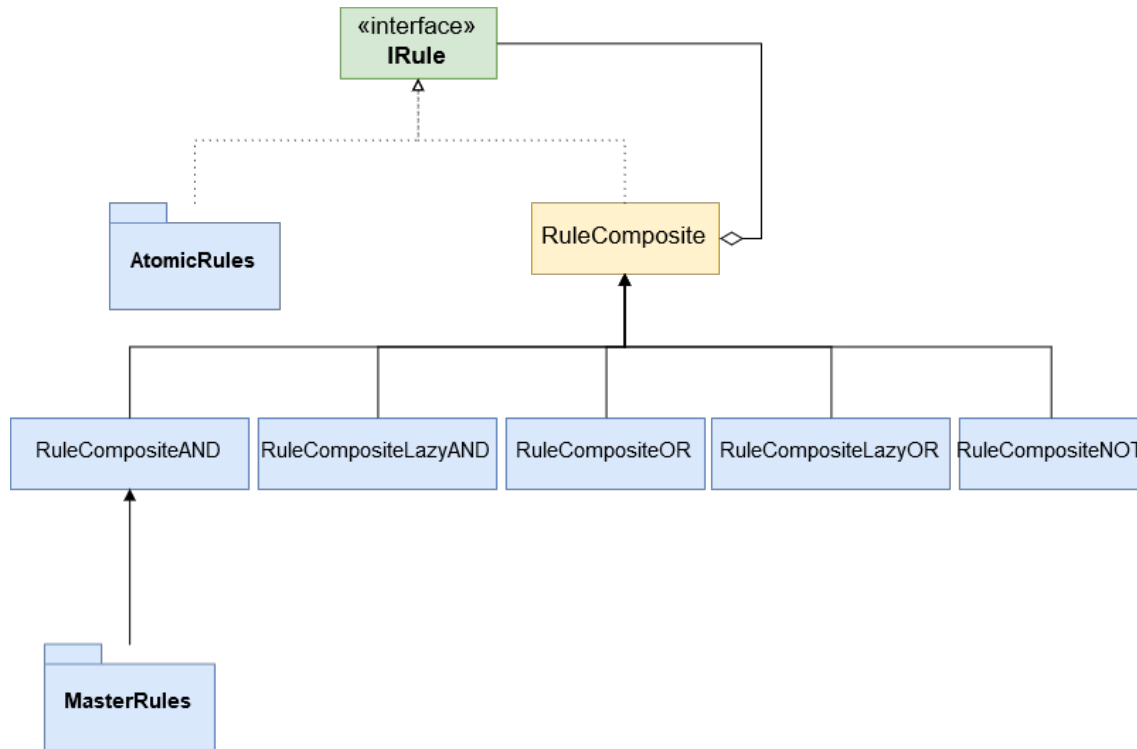


FIGURE 5.3 – Schéma UML des règles composites

## 5.3 État du jeu

La classe **GameState** contient toutes les informations qui caractérisent une partie à un instant donné, et sert donc essentiellement de classe de stockage. Elle contient un plateau, les listes des bâtiments et des unités ainsi que l'identifiant du joueur courant.

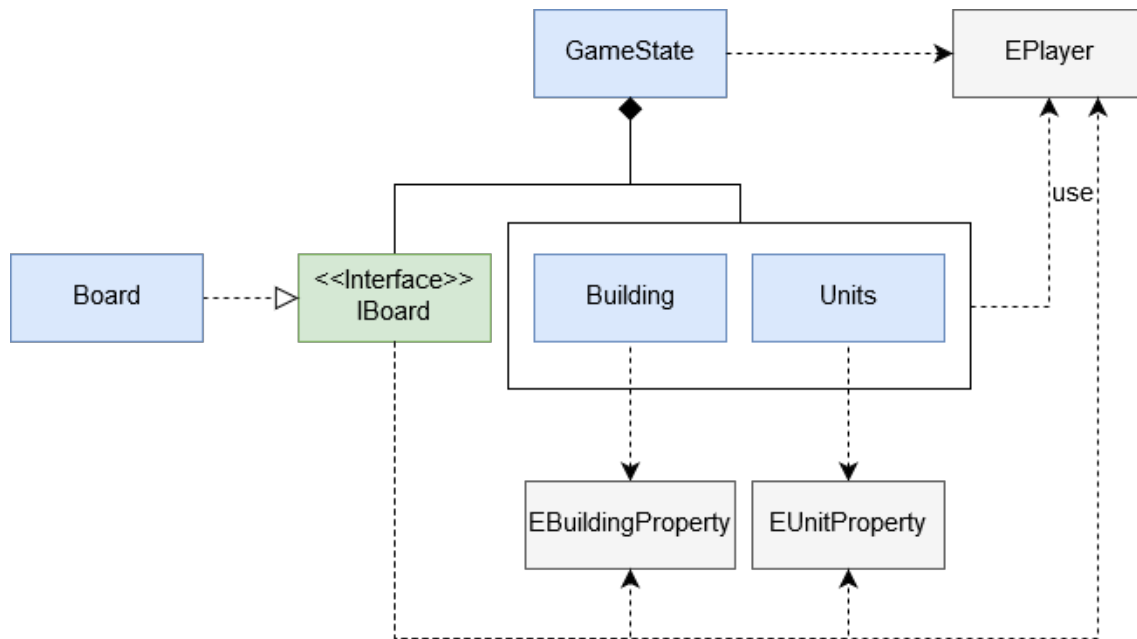


FIGURE 5.4 – Schéma UML de l'état du jeu

## 5.4 Moteur de jeu

Le moteur de jeu est la classe centrale du programme, qui relie les autres modules et les fait interagir entre eux. Il prend les commandes aux objets **Player**, donne le coup correspondant au moteur de règle avec le **GameState** à modifier, puis retourne la réponse au **Player**.

La commande est récupérée dans un **UIAction**, qui peut être interprété comme une commande pour agir sur le jeu ou bien effectuer une commande propre au programme, comme de la manipulation de fichier. La réponse est elle envoyée sous la forme d'un **GameResponse**.

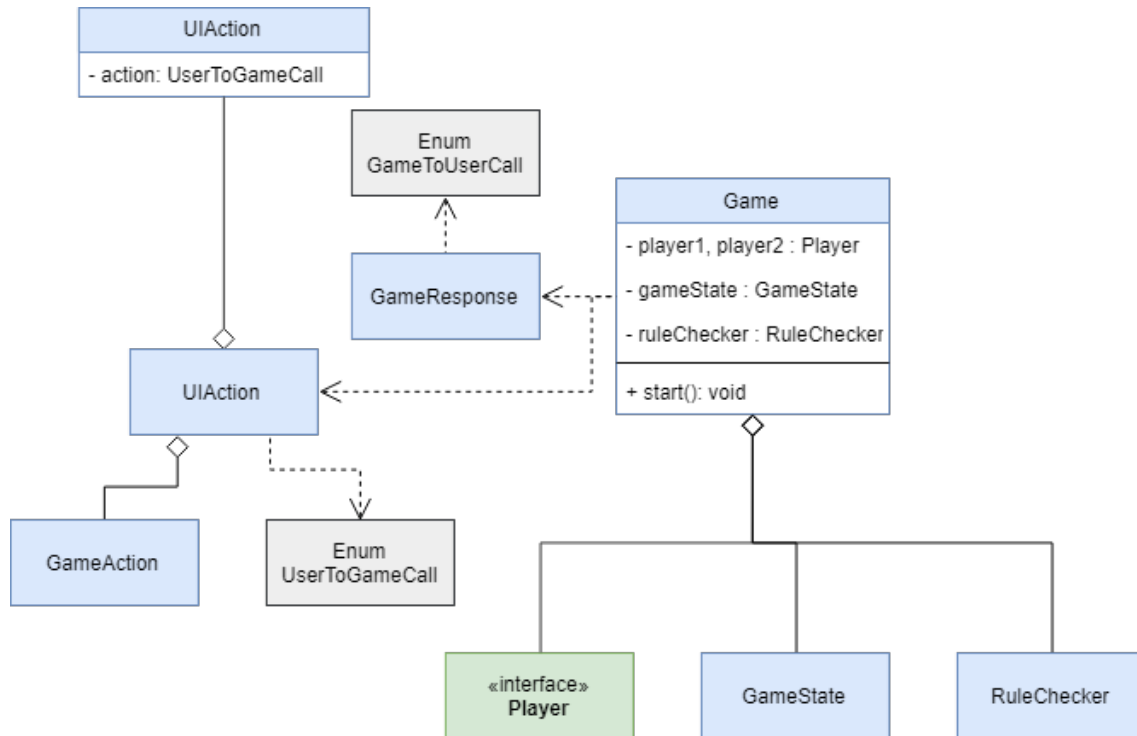


FIGURE 5.5 – Schéma UML du moteur de jeu

## 5.5 Interface utilisateur

L'interface est centrée autour de la classe **TermGUI**, héritant de **Application** de JavaFX. Elle est composée de trois "Pane", un pour le panneau contenant les éléments de commandes, un autre les éléments d'informations et enfin un contenant les éléments du jeu.

Le schéma UML complet de ce module est relativement important (Voir figure 5.6), en partie car les éléments d'affichage sont séparés eux-même en plusieurs classes pour mieux diviser le code de l'interface.

C'est par le biais de **GUIPlayer**, implémentant **Player**, que l'interface enverra les commandes et recevra les réponses du jeu.

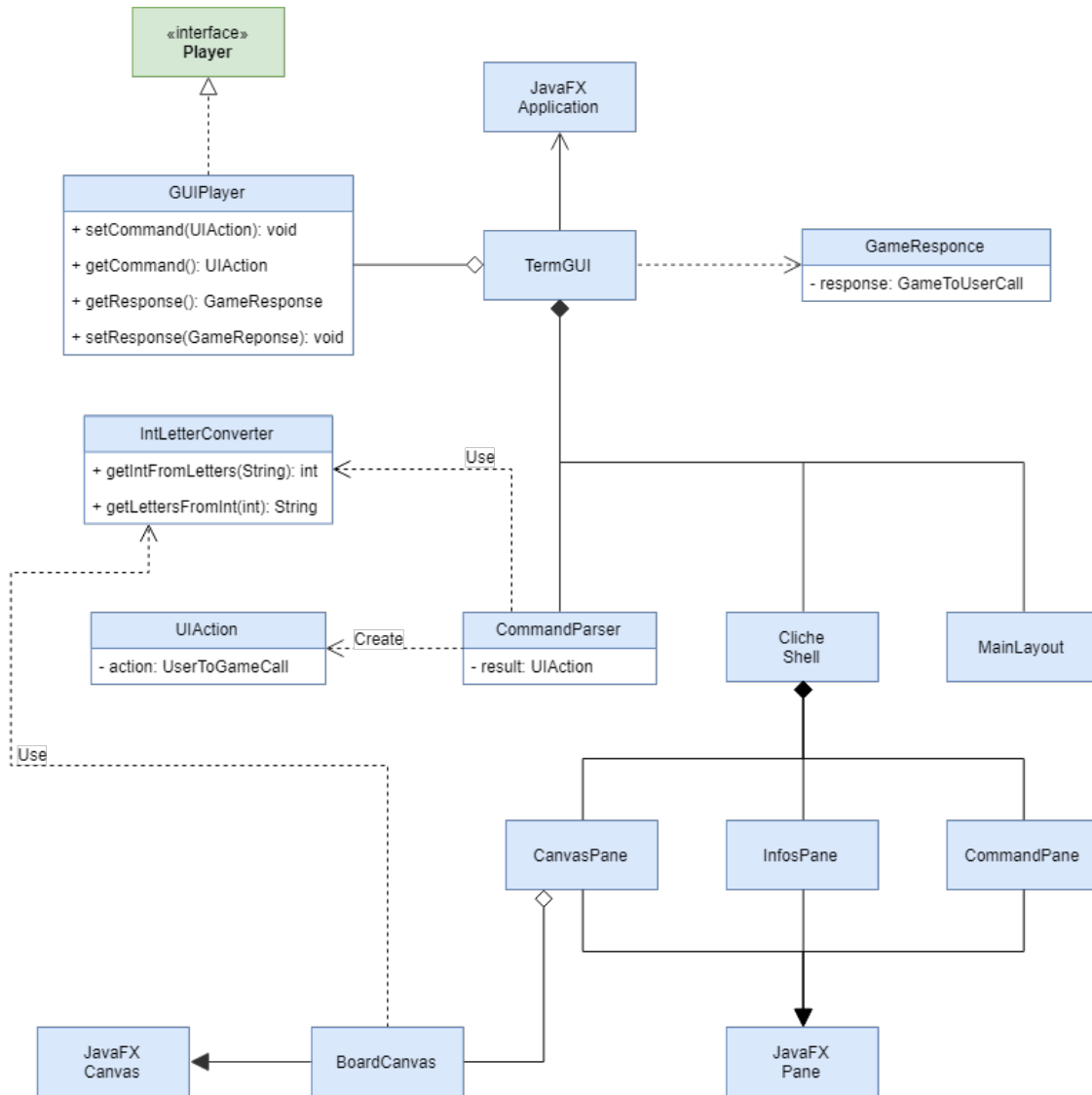


FIGURE 5.6 – Schéma UML de l'interface utilisateur

## 5.6 Module d'analyse

Le module d'analyse n'étant pas essentiel au déroulement d'une partie, celui-ci se doit de ne pas servir de dépendance aux autres modules, tout en restant accessible au niveau d'un objet implémentant l'interface **Player**, afin qu'une interface graphique ou bien un joueur artificiel puisse l'exploiter.

Le point d'entrée de ce module se fait par la classe **InfoModule** via des méthodes publiques et statiques, pour obtenir une architecture "annexe" de ce que nous avons déjà produit.

Chaque algorithme d'analyse implémente une interface définissant le type informatique de la donnée retournée (actuellement, **IMetricMapType** et **IMetricMoveType** retournent respectivement un tableau en deux dimensions de **double** et une liste de mouvements). **InfoModule** s'adapte à ces interfaces et sert de passerelle pour accéder à ces informations. Les algorithmes sont ensuite répertoriés dans des énumérations, ces dernières possédant des références vers ces dits algorithmes. Une requête au module d'analyse se fait donc en appelant **InfoModule** et en indiquant l'information désirée à l'aide de ces énumérations.

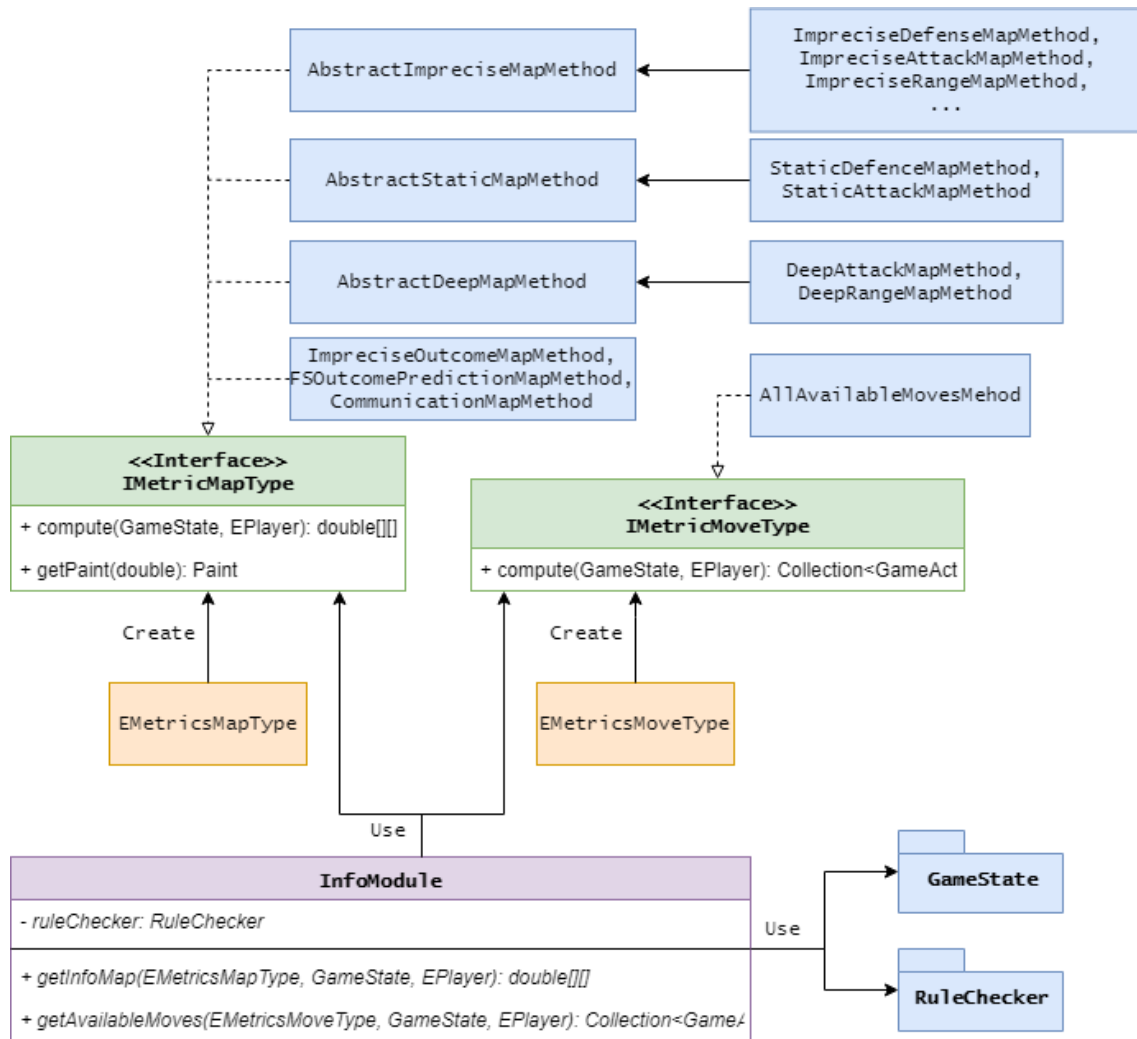


FIGURE 5.7 – Schéma UML du module d'analyse

## 6

# Implémentation

## 6.1 Modules principaux

Si, lors du développement, nous avons changé un certain nombre des décisions prises plus tôt dans le projet, les modules ont très peu été altérés jusqu'à la fin du développement, et représentent la base sur laquelle nous avons imaginé et créé nos classes pour tout le programme.

Dans les sections suivantes, nous détaillons le fonctionnement et l'implémentation de ces modules.

## 6.2 Moteur de jeu

### 6.2.1 Jeu (Classe Game)

La classe **Game** comprend deux joueurs, un état de jeu (dont le plateau), et le moteur de règles.

Son but est de faire communiquer ses différents membres entre eux pour s'assurer du bon déroulement du jeu.

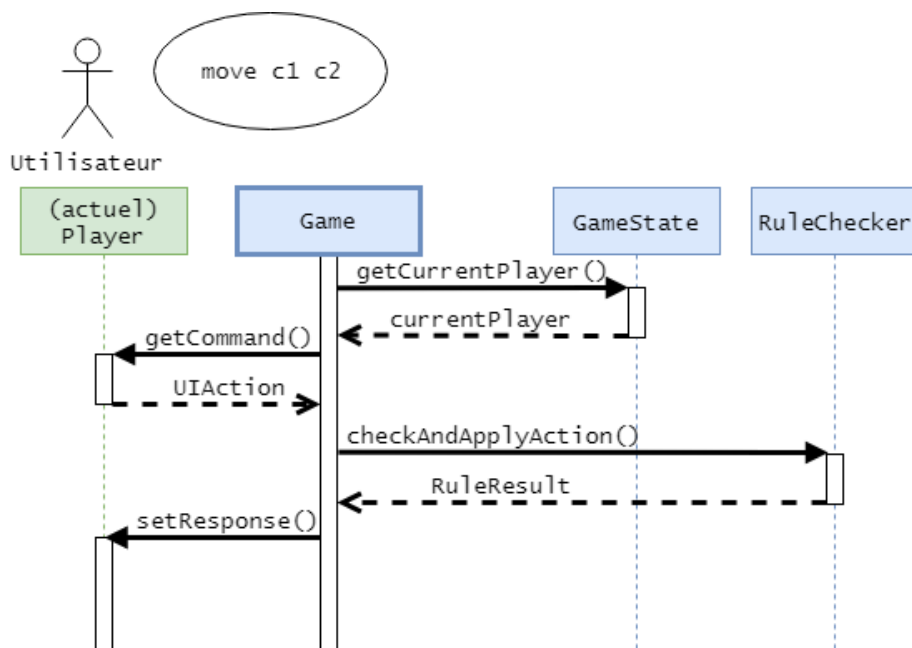


FIGURE 6.1 – Diagramme de séquence du Jeu

Dans un premier temps, **Game** demande une commande au joueur dont c'est le tour, la donne au moteur de règles pour la vérifier et l'appliquer si elle est correcte, puis retourne la réponse et le nouvel état du plateau au joueur. Deux types de commandes sont différenciés : les commandes de jeu (**move** et **attack**), qui seront transmises au moteur de règles, et les commandes **save**, **load** et **revert** qui sont gérées au sein de cette classe.

L'historique des états de jeu est également stocké dans cette classe, lorsqu'une modification se produit sur l'état actuel du jeu, sous la forme d'une pile de **GameState** dont nous pouvons dépiler la tête lorsque l'utilisateur appelle la commande **revert**.

Nous nous sommes efforcés de garder cette classe aussi réduite que possible, afin d'éviter de lui donner trop de responsabilités, car sa situation centrale lui donne beaucoup de pouvoir sur les objets qui la composent.

### 6.2.2 Plateau de jeu (Classe Board)

Nous aurions pu représenter le plateau sous la forme de listes d'objets et de coordonnées, mais cela aurait rendu sa manipulation (tel que savoir si une unité peut se déplacer sur une case) et surtout l'analyse (tel que le calcul et l'affichage des communications), beaucoup plus difficile. Nous avons néanmoins rapidement pris en compte le fait que l'implémentation d'un module d'analyse pourrait avoir besoin de nombreuses copies du plateau en différents états, aussi l'occupation mémoire et le temps de clonage a été un point crucial lors de sa conception.

Nous avons donc rapidement abandonné la représentation sous la forme d'un tableau d'objets cellules contenant des informations, car trop coûteux, pour une représentation par tableaux de primitives, manipulés à l'aide d'opérations sur les bits.

Le plateau est représenté par deux tableaux de **short** (unités et bâtiments) et un tableau de **byte** pour représenter les communications, chacun de taille *hauteur \* largeur* 6.2.

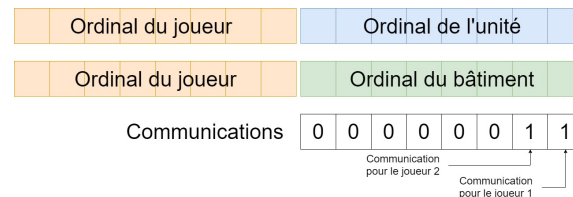


FIGURE 6.2 – Contenu des primitives de Board

Cela nous donne, pour le plateau de *Jeu de la Guerre* de taille  $25 * 20$ , deux tableaux de 500 **short** et un tableau de 500 **byte**, ce qui demande donc  $2 * 500 * (2octets) + 500 * (1octet) = 2.5Ko$  pour représenter le plateau.

Un certain nombre de méthodes privées dédiées à abstraire les manipulations bit à bit ont été créées pour réduire au maximum la nécessité de les manipuler au sein même de la classe :

- **getOffset()** sert à atteindre la case correspondantes aux coordonnées  $[x,y]$  dans des tableaux à une dimension.
- **getUnitType()**, **getBuildingType()** et **setItemType()** permettent d'atteindre la partie de l'octet correspondant au type des bâtiments ou unités de la case.
- **getPlayer()**, **setItemPlayer()** permettent d'atteindre le joueur propriétaire.

Les autres méthodes, à part pour les communications, n'ont donc pas à manipuler les primitives mais peuvent utiliser directement les énumérations des joueurs, unités et bâtiments en composant les appels de ces méthodes privées.

Les méthodes sont faites pour être appelées sur des cases valides (par exemple **getUnitPlayer()** doit être appelée pour une case avec une unité), et lèvent l'exception personnalisée **IllegalBoardCallException** si ce n'est pas le cas.

Étant donné que **Board** fournit les méthodes pour s'assurer qu'un appel est valide (tel que **isUnit()** pour le cas précédent), cette exception hérite de **RuntimeException** et n'est donc pas nécessairement testée par les appelants.

Nous avons également ajouté un tableau et les accesseurs associés pour effectuer un marquage



sur les cases, qui est notamment utilisé pour le calcul des communications et peut être utilisé pour le module d'analyse. Également, la méthode `clone()` du plateau, massivement utilisée par ce module, utilise le fait que Java puisse copier très efficacement les tableaux de primitives en natif.

### 6.2.3 État du jeu (Classe GameState)

La classe `GameState` a été créée afin de représenter l'état du jeu à un moment donné, et sera par exemple utilisée, en plus du jeu, par le moteur de règles et le module d'analyse, ainsi que pour la commande `revert`.

Cette classe comprend une représentation du plateau, et des informations complémentaires comme le tour du joueur actuel, le nombre de coups restant, la liste des unités de chaque joueur et quelques autres informations nécessaires au moteur de règles. Nous avons choisi de créer ces listes afin d'avoir accès sans itérer sur le plateau aux unités et aux bâtiments.

Une liste des unités battant en retraite à dû être ajoutée à cela afin d'être communiquée au moteur de règle.

Le plateau est totalement abstrait par cette classe, afin que seule cette classe puisse le modifier, elle comprend donc un grand nombre de méthodes "wrapper" qui se contentent de passer des appels au `Board`.

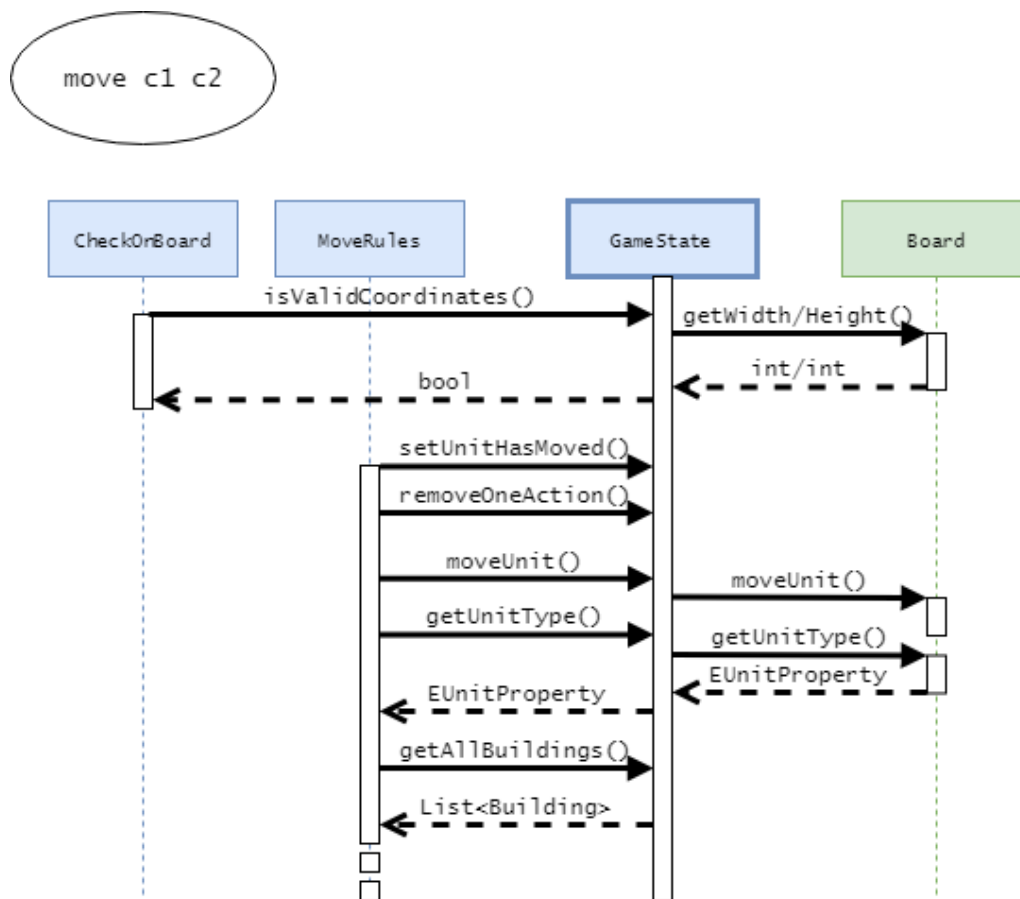


FIGURE 6.3 – Diagramme de séquence du `GameState`

### 6.2.4 Joueur (Interface Player)

Les joueurs sont les objets auxquels la classe `Game` s'adresse directement pour obtenir les commandes à jouer. Ils jouent le rôle d'interface entre le joueur "réel" (humain ou machine) et le jeu.

Afin de renforcer l'indépendance entre Game et les Player, les classes de joueurs implémentent l'interface Player qui se résume à quatre méthodes : `get/setCommand()` et `get/setResponse()`. Ainsi, le jeu pourra appeler le `getCommand()` du joueur actuel, et après avoir traité la commande donnera sa réponse par `setResponse()`.

En parallèle, la classe qui est chargée de donner les commandes (GUI par exemple), appellera `setCommand()` puis `getResponse()`.

Bien entendu, les appels n'étant pas synchrones, la classe Player se charge donc de la synchronisation, de telle sorte qu'elle soit invisible pour les threads appelant : les commandes `get()` sont bloquantes et retournent toujours le résultat attendu dès qu'il est disponible [6.4](#).

Nous avons deux implémentations de joueurs :

La première est `GUIPlayer`, qui représente un joueur "Humain" qui communique via l'interface. La construction de la classe rend la nature asynchrone de l'interface JavaFX très facile à gérer, il suffit d'appeler `setCommand()` lorsque la commande est rentrée par l'utilisateur.

La seconde implémentation est le `BotPlayer`, qui est une ébauche de système expert destinée à déterminer des coups à jouer. Dans sa forme actuelle, les commandes sont générées à l'appel de `getCommand()`. Il faudrait à terme avoir un thread séparé qui les génère de manière asynchrone, mais cela nécessiterait quelques modifications que nous n'avons pas eu le temps d'apporter.

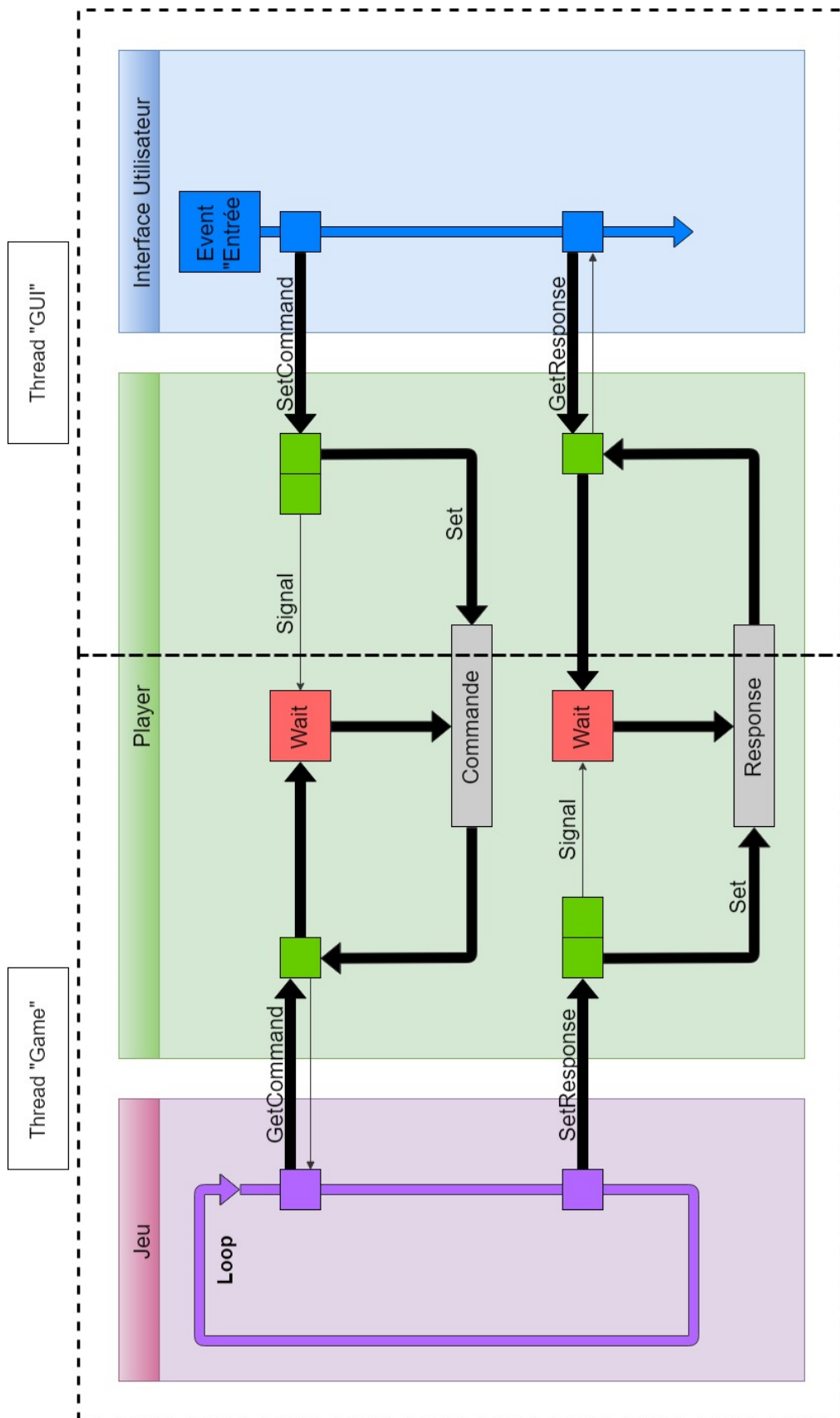


FIGURE 6.4 – Schéma de la synchronisation entre le Jeu (Game) et l'interface utilisateur

## 6.3 Moteur de règles

Comme décrit dans la partie 5.2, le moteur de règles est un point essentiel du projet, et son implémentation a évolué de nombreuses fois, notamment pour développer le système de dépendances que nous décrirons plus tard.

L'objectif principal que nous avions était d'avoir un moteur de règles permettant de modifier le contenu de ces règles sans modifier le code du moteur.

Lors de la retranscription informatique des règles, nous nous sommes rendu compte de duplication de vérification, et donc de code. Afin de conserver l'atomicité des sous-règles, et permettre de générer des messages de refus exhaustifs sans générer de problème lors de la vérification, nous avons dû mettre en place un système de dépendances. Par exemple, nous voulions vérifier une seule fois si les coordonnées entrées par l'utilisateur représentaient bien une case présente sur le plateau, afin d'éviter aux autres sous-règles demandant des accès au plateau de les vérifier à nouveau.

Dans un premier temps, ces dépendances étaient gérées par les master-rules via un système de *Map*, qui, à chaque règle vérifiée, attribuait en valeur le résultat de cette règle. Ces résultats étaient vérifiés à chaque fois qu'une sous-règle était appelée, mais cette méthode s'est avérée difficilement maintenable. Ce système de dépendance a donc été ré-implémenté sous la forme d'un arbre, parcouru par les master-rules suivant la nature des nœuds de l'arbre, pouvant être des règles, ou bien des opérateurs logiques *ET*, *OU*, ou *NON*, avec leurs équivalents paresseux. Ces arbres sont décrits dans la section 6.3.3.

Notre moteur de règles est défini par trois modules, le RuleChecker, qui communique avec le Game via le RuleResult, collectant les messages générés par les règles sur une action et stocke la validité de celle-ci, et les règles qui sont définies en trois parties, les sous-règles qui ont une unique responsabilité, les règles qui sont la représentation des actions décrites dans le livre, et les composites logiques, qui servent de liant entre les règles et les sous-règles.

### 6.3.1 Vérificateur de règles (Classe RuleChecker)

La classe RuleChecker est la classe communiquant avec le jeu (Game), elle représente le point d'entrée du moteur de règles et permet donc de vérifier la règle associée à l'action proposée par celui-ci sur une instance du plateau. Avant de commencer la vérification de la règle, elle crée un RuleResult qu'elle donnera à celle-ci afin de récupérer les messages et la validité de l'action avant de les renvoyer au jeu.

Cette classe fournit deux services, la simulation d'une action à l'aide de la méthode `checkAction()`, et la vérification suivie de l'application de la règle à l'aide de la méthode `checkAndApplyAction()`. Lors du déroulement d'une partie, la méthode `checkAndApplyAction()` est celle utilisée pour traiter une entrée provenant d'un joueur. En effet, la simulation sert principalement au joueur automatique à travers le module d'analyse.

### 6.3.2 Résultat d'une vérification (Classe RuleResult)

La classe RuleResult permet de recueillir les messages produits par les règles ainsi que la validité de l'action proposée. La validité de l'action est initialisée à vrai lors de la création du RuleResult, et peut être invalidée par les règles lors de la vérification, cependant elle ne pourra plus être remise à valide.

Ces messages sont en lien direct avec le besoin concernant la génération de messages expliquant le refus de l'action proposée, ils sont générés par les sous-règles qui explicitent ce qui n'allait pas avec la proposition de l'utilisateur en fonction de l'état du jeu actuel.



### 6.3.3 Gestion des règles

#### Les sous-règles

Les sous-règles sont décrites comme des règles "atomiques" car on leur attribue une unique responsabilité, une seule vérification, la plus fine possible. Elles implémentent l'interface `IRule` et n'utilisent que la méthode `checkAction()`, qui prend une instance du plateau et un `RuleResult`, afin d'invalider ou non le résultat et d'y ajouter un message correspondant à l'invalidation de l'action.

Par exemple, la règle `CheckIsAllyUnit` va vérifier si les coordonnées sources contenues dans le `GameAction` pointent bien vers une unité alliée sur le plateau, sinon, la règle va ajouter un message expliquant le problème (si c'est une unité ennemie ou si tout simplement il n'y a pas d'unité sur cette case) et invalider le résultat.

#### Les règles composites logiques

Ce type de règles nous permet d'utiliser 5 opérateurs logiques, le ET et le OU, leurs équivalents paresseux, et le NON. Elles spécialisent toutes la méthode `checkAction()` afin de correspondre à l'opérateur logique associé, ainsi les règles composant l'un de ces opérateurs logiques seront vérifiées et modifieront le résultat si les règles satisfont l'opérateur en question.

Pour gérer les dépendances, les opérateurs paresseux ET et OU sont utilisés. Le ET paresseux vérifie les règles dans l'ordre, et lorsqu'une règle échoue, il arrête d'appeler la méthode `checkAction()` des autres règles et génère un message expliquant que les règles suivant celle qui a échoué en sont dépendantes, et donc n'ont pas été vérifiées. Le OU paresseux fonctionne de la même manière, à la différence qu'il s'arrête de vérifier les règles dès qu'une est valide.

Les classes associées aux opérateurs ET, ET paresseux, OU, OU paresseux et NON sont respectivement `RuleCompositeAND`, `RuleCompositeLazyAND`, `RuleCompositeOR`, `RuleCompositeLazyOR` et `RuleCompositeNOT`. Elles étendent toutes la classe abstraite `AbsRuleComposite` qui implémente l'interface `IRule` et qui contient une liste chaînée de règles `IRule`, cette liste servira à la vérification des règles lors de l'appel à `checkAction()`.

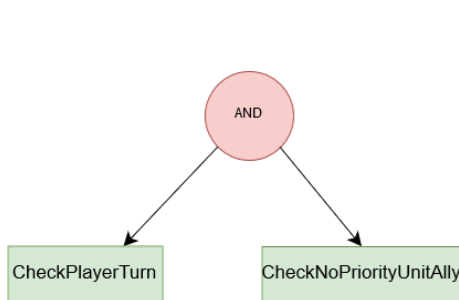


FIGURE 6.6 – Arbre des règles de fin de tour

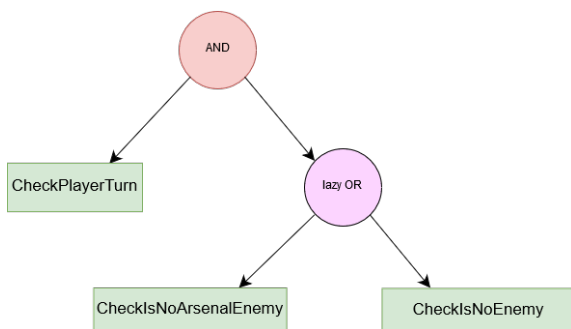


FIGURE 6.7 – Arbre des règles de victoire

#### Les règles (MasterRule)

Les règles mères sont associées à un mécanisme explicite, comme le déplacement décrit dans le livre, ou implicite, comme les conditions pour passer son tour. Elles étendent l'une des règles composites logiques afin de définir l'ordre des vérifications et les dépendances entre les règles qui la composent.

La composition logique des sous-règles se fait dans le constructeur de la règle mère, c'est ici que l'on crée et assemble les groupes de règles en leur associant un opérateur logique.

Ainsi nous avons 5 règles mères différentes, la règle de déplacement `MoveRules` 6.8, la règle d'attaque `AttackRules` 6.9, la règle de fin de tour `EndRules` 6.6, la règle de victoire `VictoryRules`

6.7 et la règle de communication `CommRules`. Chacune peut être représentée par un arbre grâce au patron de conception `Composite` (voir 5.2), l'ordre de vérification effectué lors de l'appel à `checkAction()` correspond à l'ordre préfixe des arbres.

Ces règles doivent spécialiser la méthode `applyResult()` afin d'apporter les modifications associées à elles à l'état du jeu passé en paramètre.

Afin d'afficher dans la console une représentation textuelle des arbres, la méthode `getName()` permet aux classes composites de retourner une chaîne de caractère qui est composée du nom des règles qui les composent ainsi que des opérateurs logiques qui les lient.

La règle de communication étant un comportement spécifique au jeu mais ne dépendant d'aucune règle, donc n'en vérifiant aucune, elle permet simplement de recalculer les communications sur l'état du jeu tel qu'il est défini dans le livre lors de l'appel à la méthode `applyResult()`.

## Calcul des communications

Le calcul des communications est le mécanisme le plus important du *Jeu de la Guerre*, toutes les actions de déplacement, d'attaque et de défense en sont dépendantes.

Notre implémentation du calcul des communications est une fonction récursive, qui propage la communication depuis une case dans une direction, avec une limite de portée, -1 si infinie. La communication étant créée et diffusée par les arsenaux, nous exécutons cette fonction récursive dans les 8 directions pour chaque arsenal présent sur le plateau.

Cette fonction récursive s'arrête lorsqu'elle se trouve sur une case hors du plateau, sur une montagne, sur une unité ennemie combattante ou si nous avons atteint la limite de portée définie par l'arsenal ou unité à l'origine de cette communication. Nous ignorons les cases contenant des unités alliées déjà visitées. La récursivité de cette fonction doit s'arrêter grâce aux coordonnées qui évoluent dans une seule direction, ainsi à un moment nous atteindrons forcément l'un des bords du plateau, et aussi du fait que nous ne considérons plus les unités déjà traitées, ces conditions permettant l'arrêt la fonction.

La complexité de notre implémentation (voir algorithme 1) peut s'écrire sous la forme

$$O((nbRelay + nbArsenal) * (3 * \min(H, W) + \max(H, W)) + nbUnits * 8)$$

où chaque symbole représente :

- *nbRelay* le nombre d'unités relais,
- *nbArsenal* le nombre d'arsenaux,
- *H* la hauteur du plateau,
- *W* la largeur du plateau,
- *nbUnits* le nombre d'unités combattantes.

---

**Algorithm 1** Calcul des lignes de communication

---

```

1 : procedure computeCommunication(gameState)                                ▷ Fonction entrante
2 :   clearCommunication(gameState)
3 :   clearMarked(gameState)                                                ▷ On remet à zéro le marquage du plateau
4 :   for all arsenal ∈ gameState do
5 :     coordinates ← getCoordinates(arsenal)
6 :     player ← getPlayer(arsenal)
7 :     setInCommunication(gameState, player, coordinates, true)
8 :     setMarked(gameState, coordinates, true)
9 :     for all d ∈ Directions do                                          ▷ Dans les 8 directions
10 :      createCom(gameState, coordinates, d, player, -1)

11 : procedure createCom(gameState, coordinates, direction, player, rangeMax)
12 :   coordinates ← coordinates + direction
13 :   distance ← 1
14 :   while canContinue(gameState, coordinates, player, distance, rangeMax) do
15 :     setInCommunication(gameState, player, coordinates, true)
16 :     if isAllyUnitUnmarked(gameState, coordinates, player) then
17 :       setMarked(gameState, player, coordinates, true)
18 :       range ← 1
19 :       if isRelayUnit(gameState, coordinates) then
20 :         range ← -1
21 :       for all d ∈ Directions do
22 :         createCom(gameState, coordinates, d, player, range)
23 :       coordinates ← coordinates + direction

24 : function canContinue(gameState, coordinates, player, distance, rangeMax)
25 :   canContinue ← isNotFightingEnemyUnit(gameState, coordinates, player)
26 :   canContinue ← canContinue ∧ isNotMountain(gameState, coordinates)
27 :   canContinue ← canContinue ∧ (rangeMax < 0 ∨ rangeMax ≥ distance)
28 :   return canContinue

```

---



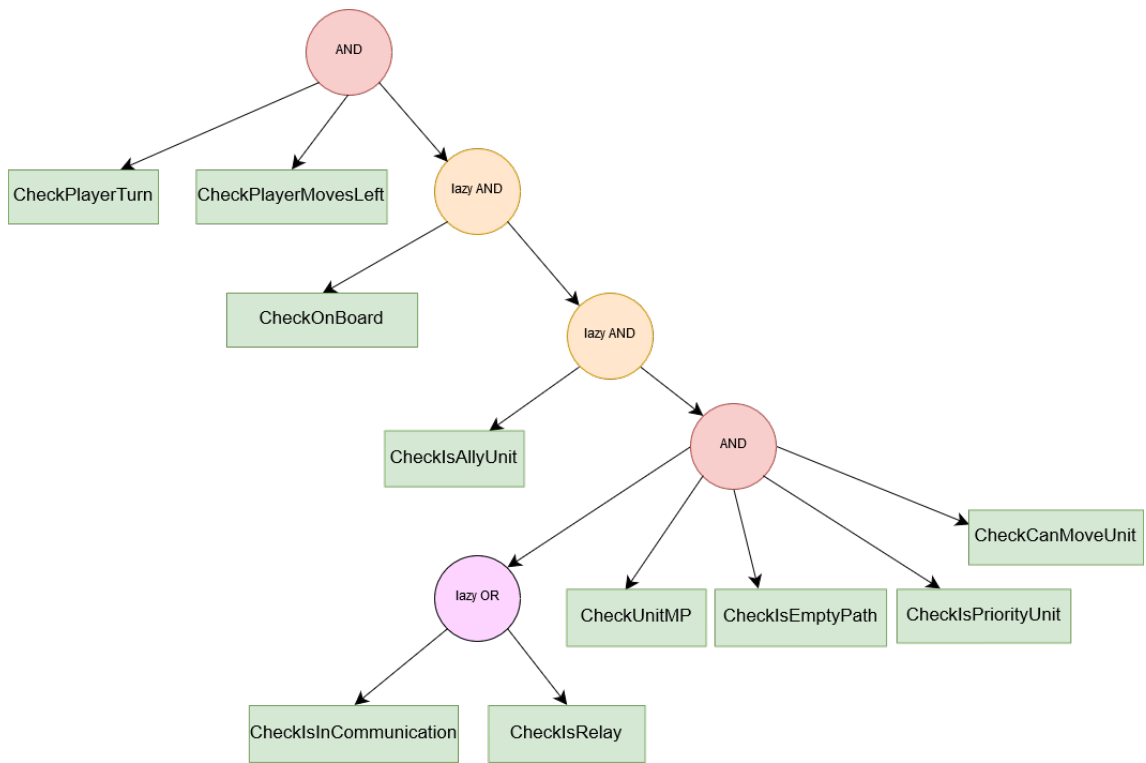


FIGURE 6.8 – Arbre des règles de déplacement

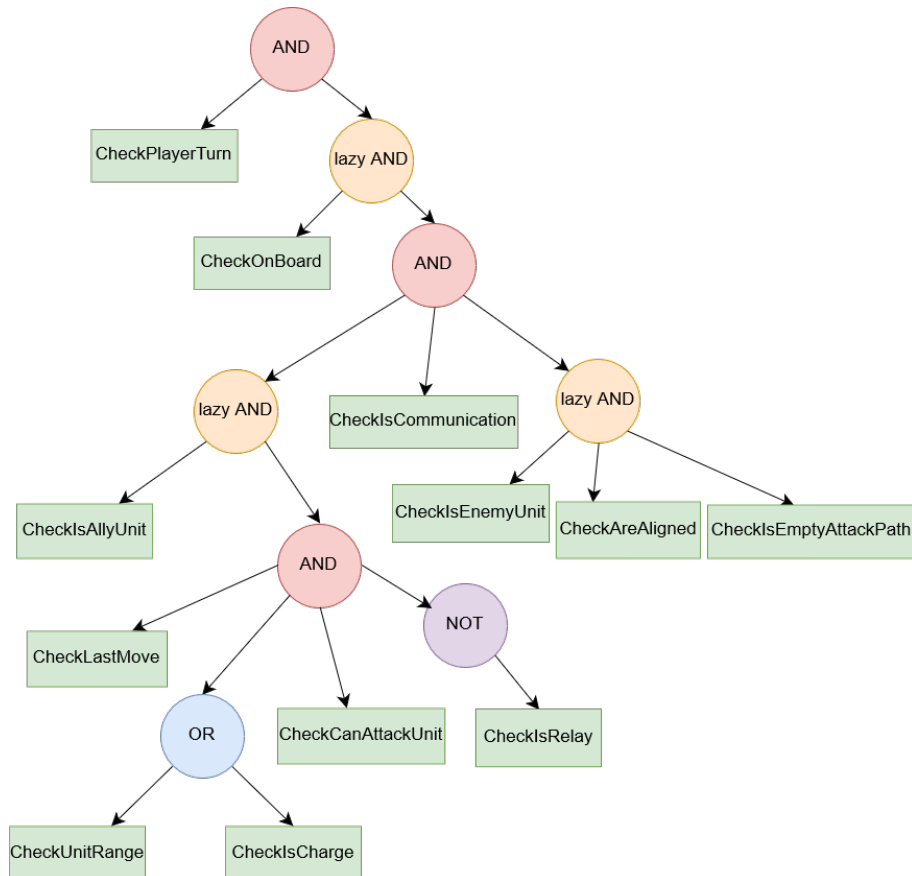


FIGURE 6.9 – Arbre des règles d'attaque

## 6.4 Interface utilisateur (Paquetage UI)

L'interface utilisateur est un élément qui a beaucoup évolué au long du développement du projet. Nous avons commencé par un affichage simple du plateau dans une fenêtre JavaFX et les commandes étaient envoyées par le terminal. Pour l'analyse syntaxique, nous avons utilisé la bibliothèque Cliche [15], qui nous permet très facilement de traduire du texte vers des appels de méthode. La classe `CommandParser` endosse ce rôle, à travers des méthodes comme `move(src, dst)`, appelées par Cliche lorsqu'on lui donne une chaîne de caractères "move [SRC] [DST]". Les coordonnées données sont au format lettres+chiffres comme précisé dans le cahier des besoins, et les lettres sont donc traduites de la base 26 vers la base 10 avant d'être utilisées.

Nous avons rapidement déplacé les commandes du terminal vers un champ de texte sur l'interface pour en simplifier l'utilisation, puis avons amélioré l'affichage avec des icônes, une légende, et enfin un menu pour choisir quel type de données afficher en fond sur le plateau.

La classe principale de l'interface est `TermGUI` (NOM A CHANGER?), héritant d'`Application` de JavaFX. Les différents éléments d'interface sont ensuite factorisés en plusieurs classes pour éviter de déclarer tous les éléments à la suite dans une seule méthode. La plupart sont donc déclarées comme héritant d'une classe JavaFX, comportent très peu de code, et sont liées entre elles afin de respecter les changements de la taille de la fenêtre.

Le `BoardCanvas`, lui, est relativement chargé, c'est lui qui s'occupe d'afficher le plateau à l'aide de méthodes de dessin sur canvas. Afin de ne pas surcharger la classe avec beaucoup d'appels de dessin, une partie est déportée vers des méthodes statiques de la classe `BoardDrawer` qui se charge par exemple d'afficher les icônes. (voir 6.10).

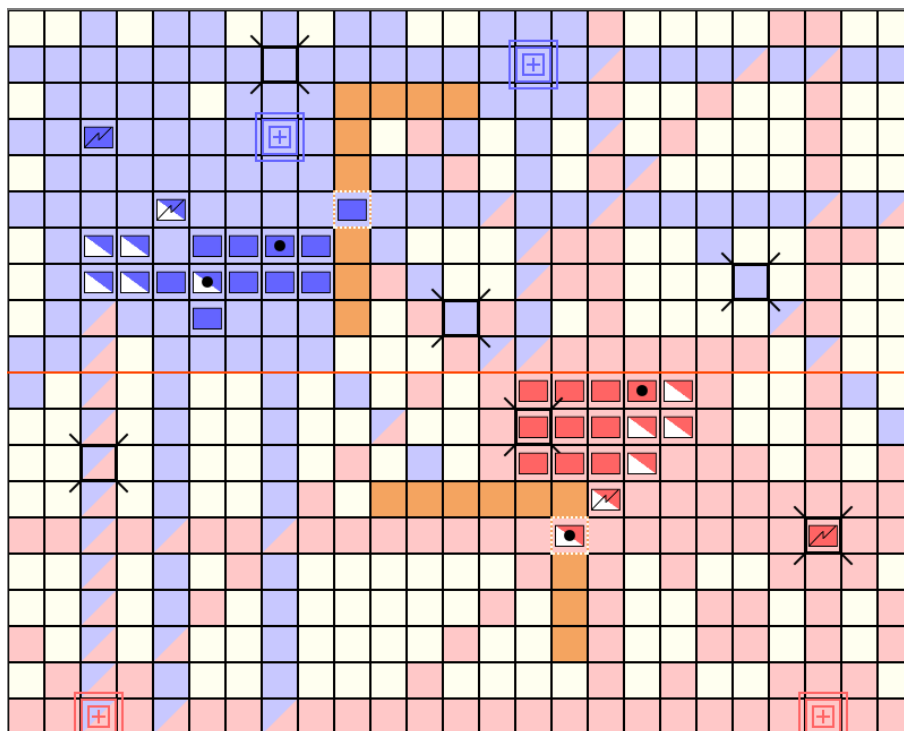


FIGURE 6.10 – Représentation du plateau

L'interface répond uniquement aux événements d'entrée et ne rafraîchit l'affichage que lorsque le plateau ou la fenêtre ont changé pour ne pas utiliser inutilement des ressources.

De même, afin de ne pas briser l'idée de segmentation de base du projet (avec les quatre modules), l'interface communique uniquement avec les objets `Player` à l'aide successivement de `setCommand()` puis de `getResponse()` lorsque l'utilisateur envoie une commande. Si la réponse fait écho à une commande invalide ou une fin de partie, cette information est affichée par une fenêtre 'pop-up'.

L'interface graphique fait néanmoins appel aux méthodes statiques du module d'analyse pour calculer les valeurs à afficher selon le type d'information demandé par l'utilisateur (voir 6.11).

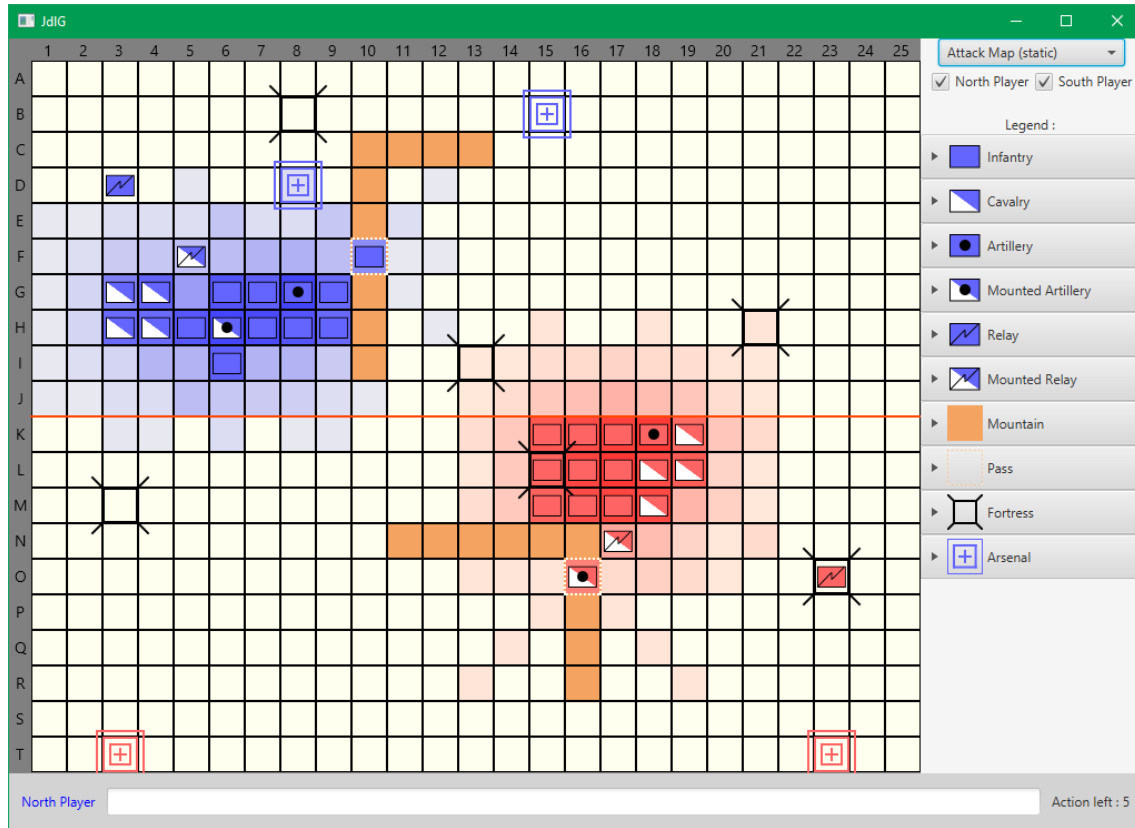


FIGURE 6.11 – Affichage de la carte d'attaque

## 6.5 Module d'analyse

### 6.5.1 Présentation

Le module d'analyse a été implémenté de façon à n'avoir aucun impact sur le déroulement du programme, et peut donc fournir des informations relatives à la partie sur demande d'un objet client, comme un joueur voulant bénéficier d'une aide visuelle ou un joueur automatique lors d'une prise de décision. Le module d'analyse permet actuellement de générer des cartes d'informations, mais aussi de générer une liste de coups en suivant des directives.

Des algorithmes d'analyse simples ont été implémentés, sans se soucier de leur complexité, pour montrer de quoi était capable le programme. Ces algorithmes sont le calcul des potentiels d'attaque, de défense, de portée et de prédiction de résultats d'affrontement. Les résultats de ces calculs peuvent être visualisés sur le GUI sous forme de cartes à nuances de couleurs. Une liste exhaustive de mouvements valides peut aussi être générée, et est utilisée par le joueur automatique implémenté.

Les algorithmes de potentiels d'attaque et de défense utilisent :

- L'état statique du plateau, sans considérer de mouvements. C'est la méthode utilisée dans le programme développé par le groupe de PdP 2014, et est utile surtout pour calculer les portées de défense des unités (adverses ou alliées), vu qu'une attaque demande un déplacement préalable de l'unité attaquante.
- L'état du plateau après le déplacement valide d'une unité (n'importe laquelle). Actuellement la méthode la plus gourmande en performances, elle récupère chaque mouvement valide (testé au préalable) d'unité, et ajoute le coefficient d'attaque ou de défense sur l'ensemble

des cases à sa portée. On obtient ainsi des "points chauds" représentant les zones les plus protégées ou plus dangereuses.

- L'état du plateau après le déplacement non-vérifié de toutes les unités. Cette méthode utilise une propriété observée et décrite dans la partie 6.5.2, et permet de calculer les zones protégées et dangereuses très rapidement à défaut de son imprécision car il n'y a pas de vérification quant à la validité des mouvements des unités impliquées.

Des tests de performance ont été effectués sur ces algorithmes, et ont permis de remarquer que le [Calcul des communications](#) était anormalement long. Cette durée était provoquée par la présence de *try/catch* dans les fonctions, appelées un nombre conséquent de fois à cause de la récursivité du calcul.

## 6.5.2 Recherches et observations sur le jeu et ses configurations

### Positionnement optimal d'unités pour l'attaque et la défense

Lors de nos parties réelles sur un plateau de jeu, nous avons remarqué que, compte tenu des calculs effectués afin de remporter un affrontement de deux groupes d'unités, un positionnement d'unités donnait la meilleure configuration que ce soit pour défendre ou pour attaquer (voir 6.13 et 6.14). Lors d'une attaque, les unités peuvent se déplacer pour être à portée de la case ciblée de façon à maximiser leur potentiel offensif. Nous avons appelé cette configuration le "positionnement en étoile". Cette configuration a pu être testée et confirmée sur le programme, rendant un tel groupe d'unités positionné autour d'un fort difficile à déstabiliser, sauf erreur humaine.

### Portée maximale d'une unité

Pendant l'implémentation des algorithmes de calcul, nous avons noté que toutes les cases à portée d'une unité, en comptant son déplacement initial et en faisant abstraction des contraintes liées à la communication et au terrain, reviennent à dessiner autour de cette unité "un carré de rayon  $r$ " avec  $r = range + movementPoints$  (figure 6.12).

Le remplissage d'un tableau étant une opération triviale algorithmiquement, cette observation nous permet de calculer très rapidement les positions dangereuses quelque soient les unités déplacées pendant le tour. Comme indiqué précédemment, cette méthode ne prend pas en compte les contraintes des communications ou des terrains, et n'est donc pas la solution parfaite.

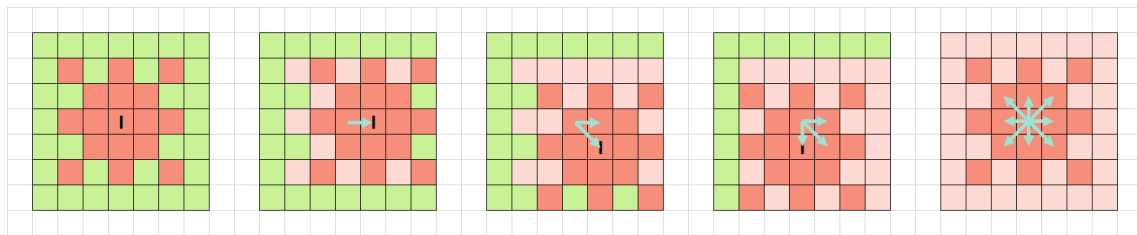


FIGURE 6.12 – Récupération du "carré" de portée d'une infanterie

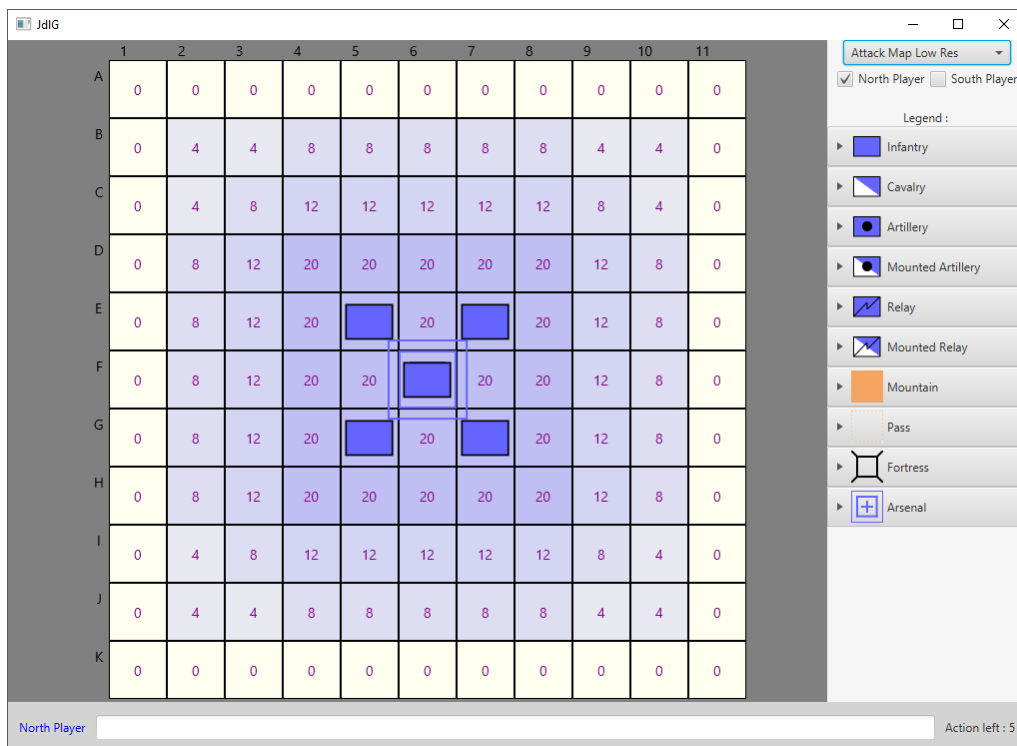


FIGURE 6.13 – Potentiel d’attaque d’une configuration ”en étoile”

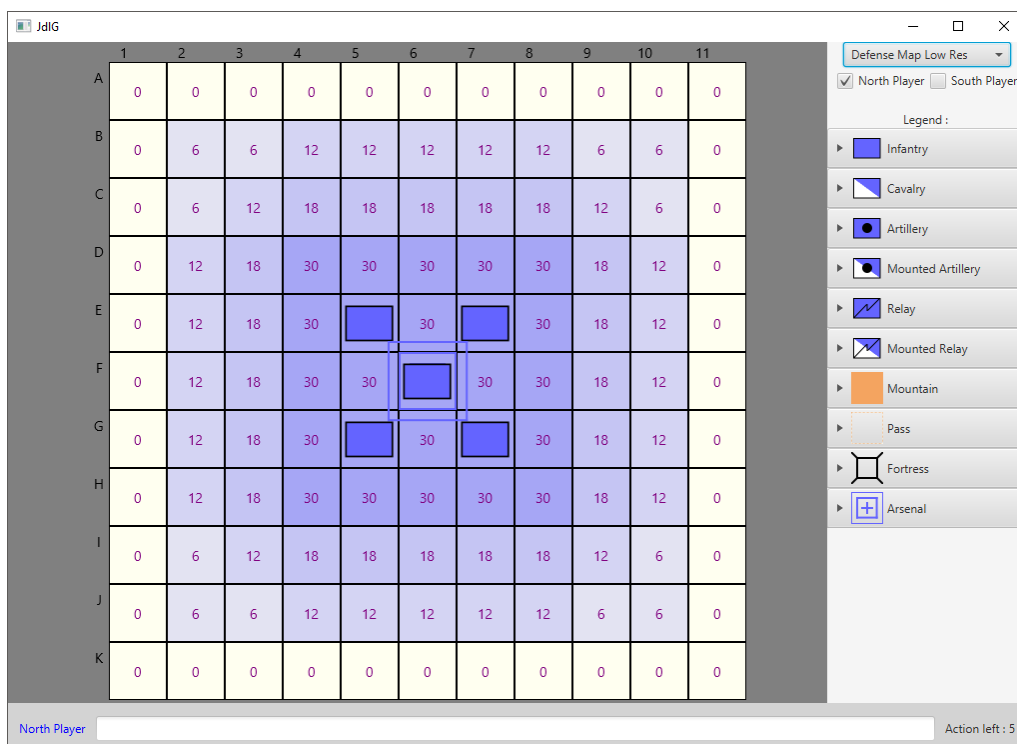


FIGURE 6.14 – Potentiel de défense d’une configuration ”en étoile”

### 6.5.3 Pistes d'exploitations possibles

Les algorithmes implémentés sont largement améliorables :

Le calcul de carte en comptant le déplacement d'une unité peut être étendu jusqu'à cinq coup, étant le nombre d'unités déplaçables par tour, donnant ainsi les meilleurs potentiels possibles, le tout de façon récursive.

Les cartes d'attaques pourraient être "mémorisées" pour avoir des tuples {positionnement, carte d'attaque} pour pouvoir placer de façon optimiser un groupe d'unité pour attaquer une case en particulier, et de même pour la défense. Ces optimisations seraient tout aussi utiles pour un joueur humain que pour un joueur automatique.

Ces calculs demandant plus de performances, la prévision de coup pour un joueur automatique peut devenir problématique. Effectuer ces calculs sur une zone du plateau réduite pourrait aider à résoudre ce problème. Ainsi, on obtiendrait des calculs précis lors des affrontements (et donc utiles à la tactique), et on pourra utiliser des calculs moins précis, comme l'approche "rapide" décrite en 6.5.2, pour avoir une idée générale de la situation du jeu (pour définir une stratégie).

Une première configuration optimale a été remarquée plus tôt, et peut être utilisée pour influencer les coups à prendre pour un joueur artificiel, de même qu'une configuration pour se déplacer rapidement en un point critique, comme dans la partie présentée dans le livre de Debord, où un cavalier du camp Nord part attaquer seul un arsenal adverse. À l'inverse, ces configurations particulières pourraient aider à déterminer les intentions de l'adversaire et à les contrer de manière efficace.

Le *Jeu de la Guerre* étant particulier pour son système de communications, l'analyse de ces lignes pourraient révéler un point "fragile" dans celles-ci et ce point deviendrait un objectif envisageable pour un joueur automatique afin d'affaiblir considérablement son adversaire.

# 7

## Tests

Les tests constituent une partie critique du projet. C'est eux qui nous permettent de nous assurer que le programme fait ce qu'on attend de lui à différents niveaux. Nous avons implémenté des tests unitaires, d'intégration et systèmes, à l'aide des bibliothèques *JUnit4* et *Mockito*.

Très peu de tests ont pu être effectués sur l'interface graphique de l'utilisateur, à cause de la difficulté à la rendre sous une forme exploitable et testable, comme le *XML* ou *SVG*. Nous ne nous sommes pas appuyés sur cette difficulté en raison de sa priorité moindre par rapport aux tests menés sur le reste de l'architecture.

### 7.1 Les tests unitaires

Les tests unitaires ont été implémentés au travers de différents scénarios, aussi bien valides qu'invalides, afin de tester nos méthodes en s'assurant ainsi qu'elles renvoient bien le résultat attendu. Pour chaque test de cas invalide, on vérifie que le message renvoyé pour signaler l'origine du problème est cohérent avec le problème provoqué. Le résultat des sous-règles dépendant généralement du contenu d'autres classes, à savoir **GameState** et **GameAction**, on utilise à la place des Mocks, ou "faux-objets", pour simuler leur comportement comme on le souhaite, donc en considérant le reste du programme comme infallible. L'un des facteurs dont nous nous sommes le plus inspirés pour ces tests est la couverture de code, permettant de vérifier la pertinence du dit code.

#### Exemple : CheckIsChargeTest

On effectue le test de la sous-règle **CheckIsCharge** en utilisant des unités de cavalerie et d'infanterie, représentant respectivement des unités avec la possibilité de charger et sans. On fait abstraction de la présence d'unités ennemies dans les cases ciblées, de la communication et des autres facteurs nécessaires à l'exécution d'une attaque car ce n'est pas dans le rôle de la règle testée de vérifier leur validité.

On tente de charger, avec une infanterie :

- Une case à proximité (Invalide).
- Jusqu'au bout d'un alignement d'elle-même et de trois cavaleries (Invalide).
- En participant à une charge menée par une autre unité de cavalerie (Invalide).

On tente de charger, avec 3 cavaleries alignées :

- Tel quel (Valide).
- L'une des cavaleries de l'alignement ayant été forcée à battre en retraite (Invalide).
- La cible n'est pas située au contact immédiat des cavaleries (Invalide).
- La cavalerie initiant l'attaque positionnée dans une forteresse (Invalide).
- Une autre cavalerie de l'alignement est positionnée dans une forteresse (Invalide).
- La case ciblée contient une forteresse (Invalide).
- La cavalerie initiant l'attaque est positionnée dans un col de montagne (Valide).

- La case ciblée contient un col de montagne (Invalide).
- L'une des cavaleries de l'alignement n'appartenant pas au joueur initiant l'attaque (Invalide).
- La case cible n'est pas alignée avec la case de la cavalerie initiant la charge (Invalide).

## 7.2 Les tests d'intégration

Les tests d'intégration ont été imaginés afin de tester les interactions des classes et modules entre eux, c'est-à-dire cette fois sans utiliser de Mocks, en ayant le moteur de règles comme acteur principal. Certains de ces tests chargent des configurations spécifiques du plateau. Comme pour les tests unitaires, pour chaque cas invalide, on vérifie que le message retourné correspond à l'erreur provoquée.

### Exemple : AttackRulesTest

La liste des sous-règles impliquées dans la règle d'attaque est indiquée à la figure 6.9, et la configuration de jeu utilisée est celle indiquée à la figure 7.1. Ici, les tests doivent non seulement vérifier si une attaque est valide ou non, mais également que le résultat à l'issue de l'affrontement est correct, à savoir l'issue de cet affrontement, les valeurs d'attaque et les valeurs de défense calculées.

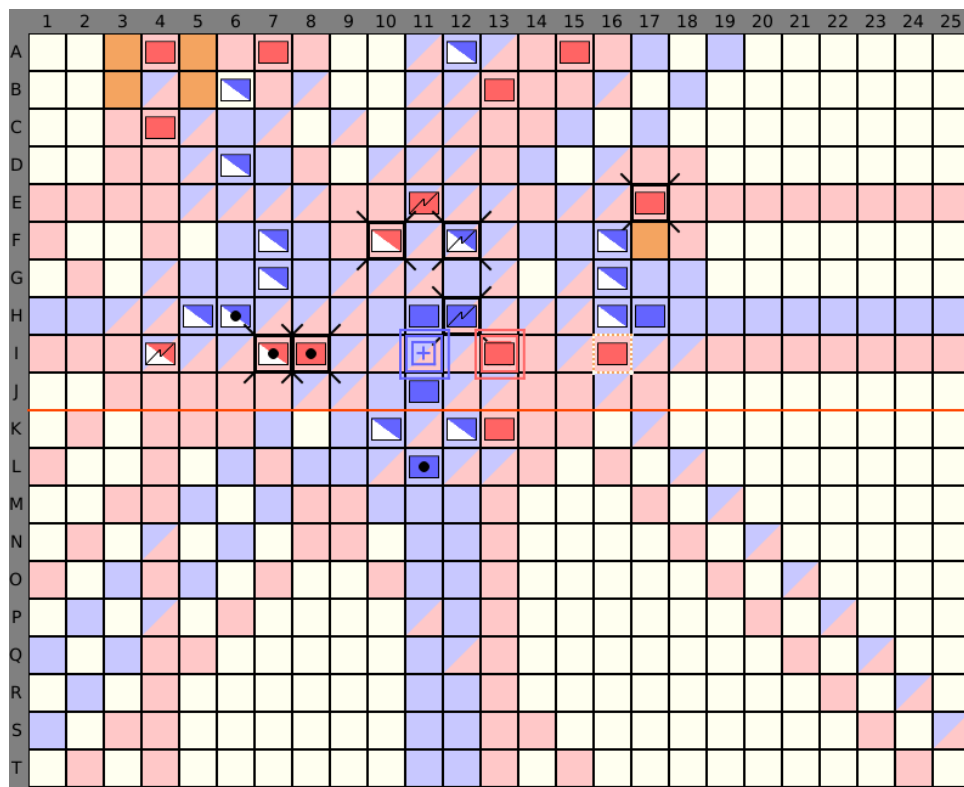


FIGURE 7.1 – Plateau de jeu des tests d'attaque

On vérifie les cas invalides :

- La case source de l'attaque ne contient pas d'unité alliée.
- La case source de l'attaque contient une unité ennemie.
- La case cible de l'attaque ne contient pas d'unité ennemie.
- La case cible de l'attaque contient une unité alliée.
- L'attaque n'a pas été précédée par un déplacement d'unité.
- L'unité attaquante désignée n'est pas celle qui a effectué le dernier déplacement.
- L'unité attaquante a déjà attaqué une fois à l'issue de son déplacement.



- L'unité attaquante est une unité non combattante (unité relais).
- La case ciblée par l'attaque est hors de portée de l'unité attaquante et il n'y a pas de charge possible entre ces deux positions. Ce cas est vérifié pour tous les types d'unités (sauf relais) et dans plusieurs configurations à la fois considérant la portée et la charge.
- L'unité attaquante n'est pas alignée avec l'unité ciblée.
- L'unité attaquante n'est pas alimentée en communication.
- Il y a un obstacle empêchant l'unité attaquante de combattre l'unité ciblée.
- L'unité attaquante a été forcée à battre en retraite.

Les tests valides étant nombreux, ils ne seront pas tous listés dans ce document. Il s'agit de vérifier en plus que les calculs d'attaque et de défense renvoient les bonnes valeurs, et que l'issue d'un combat (l'unité attaquée est intacte, doit battre en retraite, ou est détruite) est conforme aux valeurs renvoyées pour l'attaque et la défense.

## 7.3 Les tests système

Ces tests complètent les précédents, en s'assurant de la cohésion des éléments du jeu et en essayant de jouer une partie réelle via notre moteur de jeu. Ils s'effectuent sur le moteur de jeu et testent le plateau, le module système et le moteur de règles. Ces tests peuvent être étendus pour impliquer également l'interface utilisateur afin de jouer les coups en tant que joueurs "humain" et de tester le jeu dans son ensemble.

Étant donné que nous nous basons sur la partie et les règles du livre de Debord[3], notre test principal est joué sur des instances de la partie décrite. Cependant, des erreurs dans la partie illustrée dans le livre (voir section 1.2 [Sujet](#)) empêchent de jouer la partie dans son ensemble en suivant les règles et sans utiliser de moyen détourné (rajouter des tours factices permettant de jouer des coups autrement impossibles), nous avons décidé de n'en reproduire et de ne tester que quelques portions démontrant les aspects les plus importants du jeu. La partie elle-même ne présentant que certaines actions redondantes et peu de mécaniques, ces tests peuvent être complétés par des tests s'appuyant sur d'autres parties jouées afin de s'assurer que des parties plus complètes, plus complexes et moins erronées peuvent également être jouées.

Dans ces tests également, les configurations sont chargées à partir de fichiers (comme celle en figure 7.2) en début de test. Les actions (déplacement, attaque, fin de tour) sont jouées une par une et leur validité est vérifiée, et à chaque fin de tour un nouveau plateau est créé, permettant de vérifier manuellement qu'il correspond au résultat attendu. D'autres vérifications sont également possibles, par exemple dans le cas d'une attaque et destruction, on effectue une vérification avant l'attaque de la présence de l'unité attaquée à la case ciblée, et après l'attaque, de l'absence de cette même unité dans cette même case.

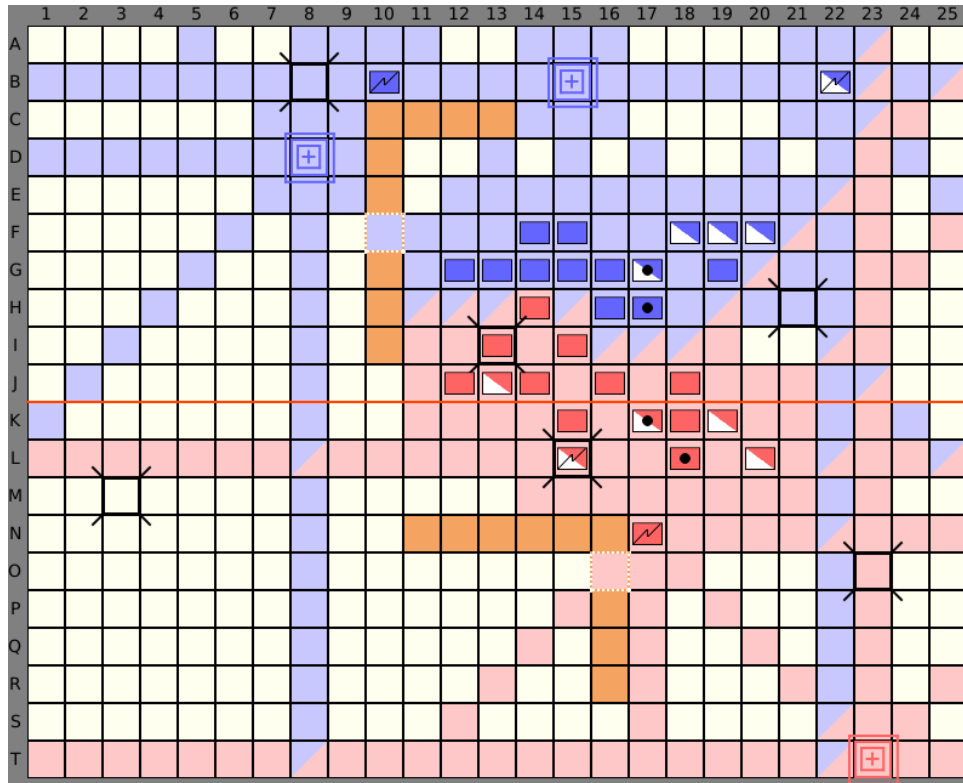


FIGURE 7.2 – Plateau de jeu du tour 42 du livre

## 7.4 La couverture de code

Le cœur du projet, le module de règles, est presque entièrement couvert (92%), et seuls certaines des fonctions renvoyant des chaînes de caractères (comme `toString()`), ainsi que certains renvois d'exceptions, ne sont pas couverts.

Le moteur de jeu et le déroulement du programme sont couverts à près de 80% et le module d'analyse a également été couvert par des exemples de test.

Cependant, la partie interface utilisateur a été délaissée en raison de son implémentation (rendu graphique et synchronisation des threads).

## 7.5 Tests des besoins

Les points des besoins concernant l'affichage et l'interface graphique ne sont que très peu testés dans le projet, mais restent visuellement vérifiables sur l'interface graphique une fois une partie commencée.

**Besoins primaires** La lecture, la traduction en coup des commandes écrites par l'utilisateur, la vérification de la validité des coups envoyés au jeu et l'application des coups validées sont bien testés et valides. De plus, la simulation de coup est très utilisée dans le module d'analyse, lui-même testé. La mécanique de changement de tour et d'alternance entre les deux joueurs est utilisée par les tests système et certains tests d'intégration.

**Besoins secondaires** Le refus des coups et l'explication du refus sont vérifiés par les tests unitaires et d'intégration utilisant les règles, et les commandes de chargement et de sauvegarde de l'état du plateau, suivant la syntaxe définie, par les tests système.

Les extractions d'informations par le module d'analyse et le remplacement du joueur humain

par un joueur automatique sont testables mais n'est pas couverts par nos tests.

**Besoins tertiaires** Ceux-ci n'ont pas été implémentés et aucun test n'a été produit pour s'assurer de leur validité.

**Besoins non-fonctionnels** Des tests de performance ont également été effectués, et le temps de réponse du moteur de règles à un coup proposé correspond aux besoins énoncés, et ces tests ont été réalisés à l'aide de *profilers*.

# Conclusion

## 8.1 Perspectives et extensions

**Aspect jeu** Actuellement, le moteur de règles permet, à lui seul, d'altérer radicalement les règles du *Jeu de la Guerre*, cela peut être de la modification ou l'ajout d'unités, de terrain, des conditions de victoire, des méthodes de calcul pour la gestion de l'attaque, des déplacements ou même des communications.

Avec ce que nous avons et ce que nous permet le moteur de règles, nous pensons pouvoir implémenter d'autres jeux tour par tour sur plateau pouvant supporter plusieurs joueurs, comme les échecs, les dames ou encore le *Reversi*.

Cependant, pour ce qui est d'une conversion totale du jeu, la liaison entre le moteur de règles et **GameState** est encore trop restreinte au *Jeu de la Guerre* pour obtenir un plateau de cases hexagonales par exemple (comme dans *DesertFox*), ou même sans cases (comme la série des *Warhammer*). Une conversion vers un genre de jeu avec une gestion plus profonde comme pour *DesertFox*, *Civilization* ou d'autres jeux de guerre reste possible mais à un degré assez limité. Par exemple, on peut implémenter une gestion de champ de vision ou de voie commerciale, de manière analogue aux communications, en modifiant de façon appropriée les *flags* dans **Board**. La conversion à un jeu d'une autre nature, comme un jeu de cartes ou un puzzle, est cependant difficilement envisageable : il serait préférable de refaire un moteur de règles ayant comme objectif d'être adaptable à n'importe quel jeu plutôt que de modifier celui-ci.

En soi, le moteur de règles possède un degré de généricité étendu pour un jeu au tour par tour à plusieurs joueurs, avec ou sans plateau. Afin de pouvoir être exploité au maximum, il aurait fallu rendre **GameState** et **Board** bien plus génériques qu'ils ne le sont actuellement, mais cela sort des objectifs de ce projet.

En ce qui concerne le *Jeu de la Guerre*, il reste à implémenter les règles d'initialisation de la partie, comme le positionnement des bâtiments et des unités tout en cachant les effectifs adverses, ainsi que pour déterminer le joueur qui débutera la partie.

Aussi, il faudrait implémenter une règle permettant de s'assurer qu'une instance du plateau respecte bien les conditions décrites dans le livre, par exemple les forteresses sur des cases en liaison directe avec ses arsenaux, le nombre de montagnes ou encore l'asymétrie du plateau. Le reste des règles est implémenté, testé et fonctionne dans l'exécutable produit.

**Système expert** Concernant un système expert, comme nous l'avons décrit dans les parties précédentes, nous avons une première implémentation d'un joueur automatique ainsi qu'un module d'analyse. Actuellement, nous calculons principalement des cartes d'attaques et de défenses de chaque camps en prenant en compte un et aucun déplacement, ainsi que des déplacements valides d'unités.

Actuellement notre joueur automatique n'utilise que la génération des déplacements valides afin d'en jouer un au hasard. Nous avons pensé à diverses pistes afin de rendre ses déplacements plus

cohérents.

Tout d'abord nous avons pensé à trier la liste des déplacements possibles par le nombre total de cases en communication que ça engendrerait, et de choisir parmi les 10% meilleurs résultats un mouvement au hasard. Cela pourrait peut-être faire apparaître des positionnements d'unités plus favorables permettant de favoriser la communication lors de déplacement. Une autre façon d'aborder ce cas pourrait être de trier les déplacements en fonction du nombre de déplacements valides qui serait possible d'effectuer à la suite du déplacement, ainsi cela favoriserait une plus grande liberté de mouvement.

De manière analogue en utilisant les cartes d'attaques/défenses, privilégier les déplacements qui permettent de conserver un maximum de potentiel d'attaque ou de défense lors de déplacement, et à l'issue de ce déplacement avoir une liste des attaques possibles qui permettent la retraite ou la destruction d'une unité ennemie.

**Stratégies applicables au *Jeu de la Guerre*** Lors de nos différentes parties sur le jeu, nous avons pu constater quelques stratégies qui pourraient être applicable à un système expert. Nous détaillerons dans cette partie les différentes stratégies que nous avons pu remarquer.

Tout d'abord une première stratégie, qui se trouve appliquée dans les premiers tours de la partie décrite dans le livre, pourrait être de focaliser ses forces vers les arsenaux ennemis à l'aide d'unités rapides.

Un autre plan que pourrait utiliser le système expert serait d'affiner la stratégie précédente en calculant les faiblesses dans la formation des unités ennemies permettant au joueur automatique de mieux déplacer ses unités afin d'atteindre plus facilement les arsenaux.

Une autre stratégie envisageable serait de jouer sur la défensive, et donc de privilégier la protection de ses arsenaux et de ses unités relais afin de forcer l'ennemi à attaquer.

L'utilisation des cartes d'attaque et de défense semble cruciale pour déterminer les mouvements globaux des troupes, et en garder une trace pourrait permettre de mieux discerner les intentions de l'ennemi pour plus facilement le contrer.

**Bilan** Nous pensons avoir atteint la majorité des objectifs que nous nous étions fixés, mais avons manqué de temps pour tous les réaliser.

La phase d'initialisation de la partie n'a pas du tout été implémentée, et nécessiterait en plus de l'ajout de quelques règles une modification de la classe **Game**. En l'état, cette classe n'est pas capable de gérer des étapes au cours d'une partie.

Nous n'avons pas non plus implémenté de stratégie pour permettre au système expert de pondérer l'intérêt d'un coup, celui-ci le choisissant donc aléatoirement. Des recherches on néanmoins été faites sur ces stratégies, que nous détaillons en section 6.5.

Finalement, réaliser ce projet nous a appris de nombreuses choses sur la manière de mener le développement d'un logiciel. Le temps que nous avons passé à réfléchir au projet avant de commencer le développement nous a permis de mieux commencer la phase de code, nous a permis d'éviter d'erreurs de conceptions qui auraient pu gravement nuire au futur développement, ce qui à terme nous a conduit à avoir des bases plus solides que lors des projets précédents que nous avons pu faire.

Nous avons également appris à nous servir de différents outils pour nous aider dans notre tâche, tels que Doxygen avec lequel nous avons pu nettement améliorer l'organisation de notre code, ou encore de profilers.

Enfin, nous espérons que le travail conduit au cours de ce projet pourra être réutilisé par d'autres groupes de Projet de Programmation dans les années à venir.

# Bibliographie

- [1] Richard BERG. *Desert Fox*. Six Angles, Simulations Publications Inc., 1981.
- [2] Carl von CLAUSEWITZ. *Vom Kriege*. 1832.
- [3] Guy DEBORD et Alice BECKER-HO. *Le Jeu de la Guerre - Relevé des positions successives de toutes les forces au cours d'une partie*. Gallimard, 2006.
- [4] Guillaume DESBIEYS, David CHEMINADE, Hubert MONDON et Quentin MICHAUD. *Jeu de la guerre - Kriegspiel (PdP 2014)*. <https://services.emi.u-bordeaux.fr/projet/savane/projects/kriegspiel/>. Visité : 12/01/2018.
- [5] *Documentation de l'API Javafx*. <https://docs.oracle.com/javase/8/javafx/api/toc.htm>. Visité le 01/04/2018.
- [6] Alexander GALLOWAY et Stephen KELLY. *Errata and Commentary on Debord's Published Rules*. <http://r-s-g.org/kriegspiel/errata.php>. Visité : 29/03/2018.
- [7] *Kriegspiel*. <http://r-s-g.org/kriegspiel/index.php>. Visité : 12/01/2018.
- [8] *KriegSpiel : Soyons plus Debord que Madame Debord*. <https://sites.google.com/site/lefinmotdelhistoire/home/kriegspiel>. Visité : 02/04/2018. Avril 2008.
- [9] Michael LEECE et Arnav JHALA. *Reinforcement Learning for Spatial Reasoning in Strategy Games*. <https://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7407/7603>. Visité : 10/02/2018.
- [10] Bill LEESON et Johan HÖRBERG. *Kriegsspiel - The Prussian Army Wargame*. Reproduction du jeu original du XIXème siècle des pères et fils von Reisswitz. Too Fat Lardies, 1983.
- [11] L'EXPRESS.FR AVEC AFP. "Les archives de Guy Debord entrent à la BNF". *l'Express* (2011). [https://www.lexpress.fr/culture/livre/les-archives-de-guy-debord-entrent-a-la-bnf\\_965886.html](https://www.lexpress.fr/culture/livre/les-archives-de-guy-debord-entrent-a-la-bnf_965886.html).
- [12] Funge MILLINGTON. *AI for Games : chapters 5 - 6 : Decision Making - Tactical and Strategic AI*. Morgan Kaufmann Publishers, 2009.
- [13] Santiago ONTAÑON, Gabriel SYNNAEVE, Alberto URIARTE, Florian RICHOUX, David CHURCHILL et Mike PREUSS. *A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft*. <https://hal.archives-ouvertes.fr/hal-00871001/document>. Visité : 10/02/2018.
- [14] SECARDS. *Kriegspiel ou le jeu de la guerre*. <https://secardartiste.wordpress.com/2016/02/22/kriegspiel-ou-le-jeu-de-la-guerre/>. Visité le 10/01/2018. 2016.
- [15] *Site de la librairie Cliche*. <http://cliche.sourceforge.net/>. Visité le 01/04/2018.

# Table des figures

1.1	"Le Jeu de la Guerre" (Kriegsspiel) dans <i>L'Illustration</i> du 22 août 1874 . . . . .	5
1.2	Représentation du plateau de jeu décrit dans le livre du <i>Jeu de la Guerre</i> . . . . .	7
1.3	Exemple d'une charge de 3 cavaliers sur l'unité en B2 initiée par le cavalier en E5 . . . . .	9
2.1	Prototype papier du <i>Jeu de la Guerre</i> . . . . .	10
2.2	Prototype papier du <i>Jeu de la Guerre</i> - autre point de vue . . . . .	11
2.3	Capture d'écran du programme développé du PdP 2014 . . . . .	12
2.4	Interface de <i>Kriegsspiel</i> par Radical Software Games . . . . .	13
2.5	Potentiel d'attaque des unités rouges (PdP 2014) . . . . .	14
2.6	Variation de potentiel de défense des unités bleues (PdP 2014) . . . . .	14
3.1	Format du fichier de sauvegarde . . . . .	16
3.2	Ébauche d'interface commande et d'aperçu de l'état du plateau . . . . .	17
3.3	Scénario d'utilisation pour un coup proposé . . . . .	17
4.1	Interface graphique du logiciel . . . . .	19
4.2	Commandes et identifiants des unités et terrains employés . . . . .	20
4.3	Message d'erreur d'une commande d'attaque invalide . . . . .	21
4.4	Message d'information à la suite d'une attaque . . . . .	22
4.5	Message d'information à la suite d'une attaque . . . . .	22
5.1	Interactions au sein de l'architecture du programme . . . . .	23
5.2	Schéma UML du moteur de règles . . . . .	24
5.3	Schéma UML des règles composites . . . . .	25
5.4	Schéma UML de l'état du jeu . . . . .	26
5.5	Schéma UML du moteur de jeu . . . . .	27
5.6	Schéma UML de l'interface utilisateur . . . . .	28
5.7	Schéma UML du module d'analyse . . . . .	29
6.1	Diagramme de séquence du Jeu . . . . .	30
6.2	Contenu des primitives de <b>Board</b> . . . . .	31
6.3	Diagramme de séquence du GameState . . . . .	32
6.4	Schéma de la synchronisation entre le Jeu (Game) et l'interface utilisateur . . . . .	34
6.5	Diagramme de séquence du RuleChecker . . . . .	36
6.6	Arbre des règles de fin de tour . . . . .	37
6.7	Arbre des règles de victoire . . . . .	37
6.8	Arbre des règles de déplacement . . . . .	40
6.9	Arbre des règles d'attaque . . . . .	40
6.10	Représentation du plateau . . . . .	41
6.11	Affichage de la carte d'attaque . . . . .	42
6.12	Récupération du "carré" de portée d'une infanterie . . . . .	43
6.13	Potentiel d'attaque d'une configuration "en étoile" . . . . .	44
6.14	Potentiel de défense d'une configuration "en étoile" . . . . .	44
7.1	Plateau de jeu des tests d'attaque . . . . .	47
7.2	Plateau de jeu du tour 42 du livre . . . . .	49

# Annexe A

Listing A.1 – Fichier pour l'état initial présenté dans le livre *Le Jeu de la Guerre*

```
25;20
1;5
F;8;2;1
F;21;8;1
F;13;9;1
AR;15;2;1
AR;8;4;1
M;10;3;1
M;10;4;1
M;10;5;1
M;10;7;1
M;10;8;1
M;10;9;1
M;11;3;1
M;12;3;1
M;13;3;1
CO;10;6;1
F;3;13;2
F;15;12;2
F;23;15;2
AR;3;20;2
AR;23;20;2
M;11;14;2
M;12;14;2
M;13;14;2
M;14;14;2
M;15;14;2
M;16;14;2
M;16;16;2
M;16;17;2
M;16;18;2
CO;16;15;2
R;3;4;false,true,false,true;1
RC;5;6;false,true,false,true;1
C;3;7;false,true,false,true;1
C;4;7;false,true,false,true;1
C;3;8;false,true,false,true;1
C;4;8;false,true,false,true;1
I;5;8;false,true,false,true;1
I;10;6;false,true,false,true;1
I;6;7;false,true,false,true;1
I;7;7;false,true,false,true;1
I;9;7;false,true,false,true;1
I;7;8;false,true,false,true;1
I;8;8;false,true,false,true;1
I;9;8;false,true,false,true;1
I;6;9;false,true,false,true;1
A;8;7;false,true,false,true;1
AC;6;8;false,true,false,true;1
R;23;15;false,true,false,true;2
RC;17;14;false,true,false,true;2
C;19;11;false,true,false,true;2
C;19;12;false,true,false,true;2
C;18;12;false,true,false,true;2
C;18;13;false,true,false,true;2
I;15;11;false,true,false,true;2
I;15;12;false,true,false,true;2
I;15;13;false,true,false,true;2
I;16;11;false,true,false,true;2
I;16;12;false,true,false,true;2
I;16;13;false,true,false,true;2
I;17;11;false,true,false,true;2
I;17;12;false,true,false,true;2
I;17;13;false,true,false,true;2
A;18;11;false,true,false,true;2
AC;16;15;false,true,false,true;2
```