

PROGRAMMATION DU JEU "Lionheart"



Etudiants :

COISPEL Aldrik
DUONG Jean-Luc
BOUTANT Thomas
MANZANO Nicolas

Encadrant:

Djamel BELAÏD

Date: 22/05/2016

TABLE DES MATIÈRES

Introduction.....	3
1. Cahier des charges	4
2. Développement	
2.1 Analyse du problème et spécification fonctionnelle.....	8
2.2 Conception préliminaire	9
2.3 Conception détaillée	12
3. Tests	17
Conclusion	20
Annexes	
A. Ressources	22
B. Manuel de l'utilisateur	23
C. Règles du jeu	24

Introduction

Ce document est le rapport final du projet "Lionheart", réalisé dans le cadre du module PRO3600 Développement informatique. Il donne une vision générale de ce qui a été fait au travers de notre progression, des contraintes et difficultés rencontrées, ainsi que de l'évolution du codage.

Ce document comporte trois parties à la suite de la présente introduction:

- une présentation du cahier des charges où nous comparerons les objectifs initiaux avec ce qui a finalement été réalisé;
- la description du développement, qui se décompose en 3 parties : une analyse du problème, une conception préliminaire et une conception détaillée
- des tests permettant de vérifier la cohérence du code du jeu avec les règles

Il se termine par une conclusion résumant les points importants. En fin de document, une annexe présente le manuel utilisateur et les règles du jeu "Lionheart" que nous avons commenté pour permettre une meilleure compréhension - et pour donner envie d'y jouer.

1. Cahier des charges

"Lionheart" est un jeu de plateau opposant 2 joueurs et faisant intervenir de l'aléatoire au moyen d'un lancer de dés.

Le but initial du projet est de coder "LionHeart", en partant d'une modélisation simplifiée du jeu, pour arriver à la réalisation du jeu dans son intégralité.

Le jeu "Lionheart" comporte à l'origine 2 niveaux de jeu. Initialement nous pensions qu'il nous serait possible dans les contraintes de temps fixées et avec nos capacités, de réaliser les fonctionnalités suivantes :

- *jouer en mode solo au niveau 1 ou 2*
- *jouer en multijoueur (1 vs 1) au niveau 1 ou 2*

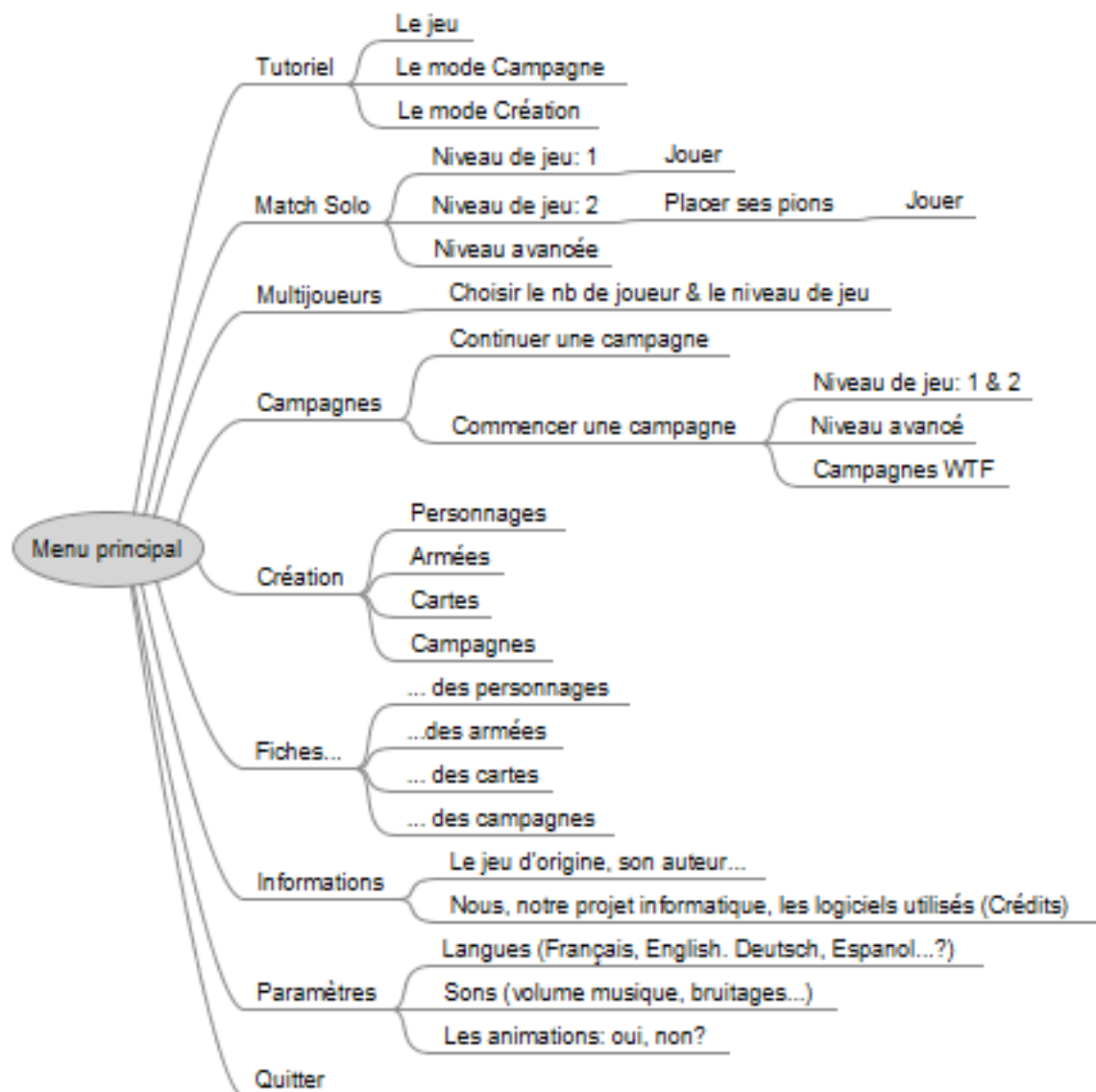
Seul le jeu en multijoueur au niveau 1 a été finalement implanté.

Les contraintes techniques sont les suivantes:

- *le langage de développement choisi est JAVA.*

Nous envisagions dans un deuxième temps de réaliser les autres développements présentés dans ce Mind Mapping:

Nous considérons déjà cela comme du "bonus" et l'arbre ci-dessous représentait l'idéal que nous souhaitions atteindre. Comme vous avez pu le comprendre, cela n'a pu se faire.



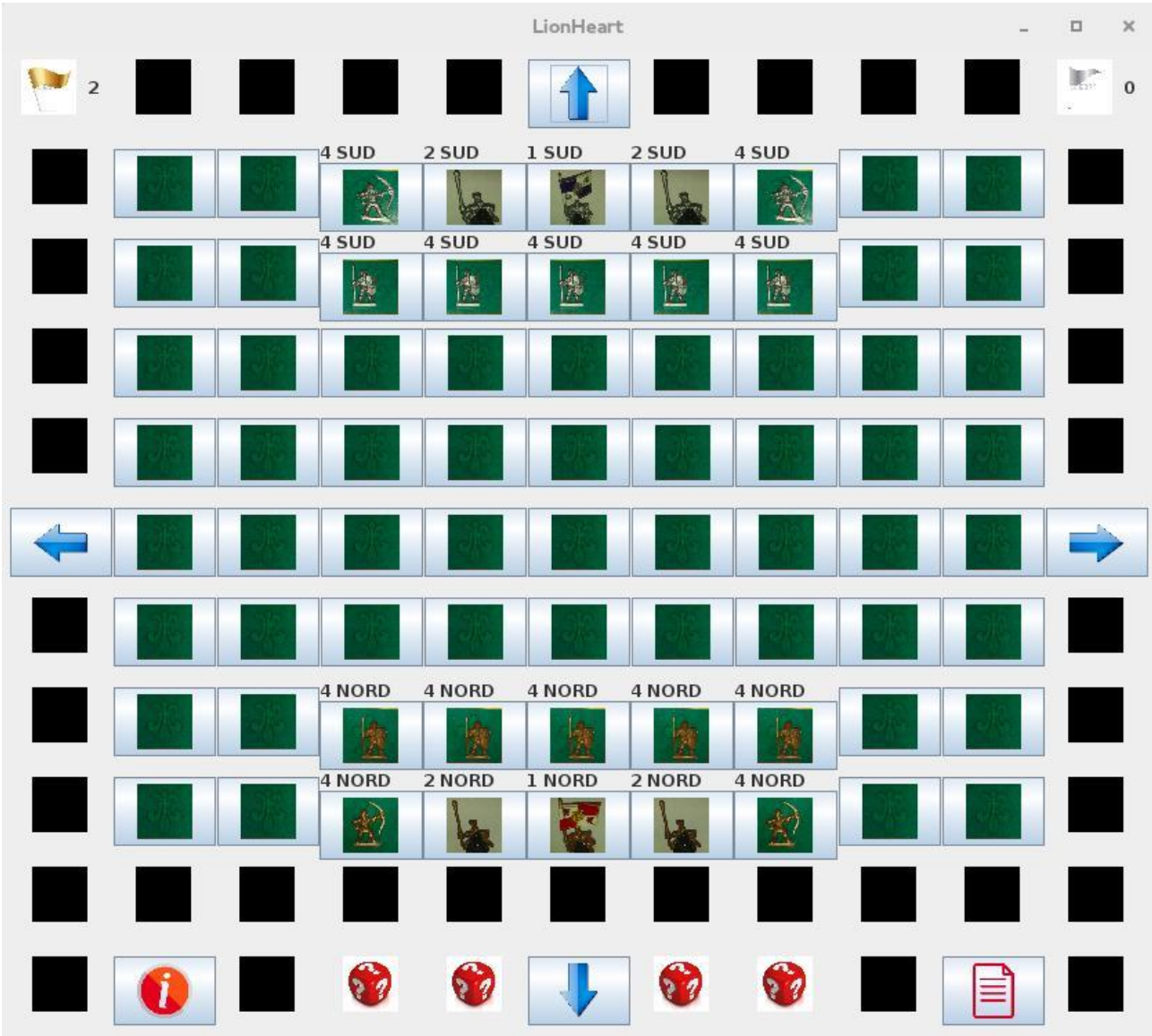
Une fois l'étude du jeu complet réalisée, nous nous sommes concentrés sur certains aspects afin d'avoir un produit fini dans les temps tout en gardant à l'esprit les éventuels développements réalisables. Nous nous sommes donc concentrés sur le niveau 1 opposant 2 joueurs humains.

Toutefois nous avons aussi joint trois choses au jeu afin d'aider le joueur :

Une aide personnalisée en fonction de l'avancée du jeu, cliquable par un JButton sur le plateau de jeu.

Nous avons également joint les règles du jeu accessible par un JButton sur le plateau de jeu.

Un Compteur de Points d'Action Or et Argent pour que chacun visualise en temps réel le nombre d'actions qu'il peut encore réaliser.



2. Développement

L'organisation de cette partie est la suivante: une première section est dédiée à l'analyse du problème et à la spécification fonctionnelle, partie durant laquelle on étudiera ce qui a finalement été réalisé; La seconde partie aborde la conception préliminaire du projet. Enfin, la troisième partie présentera la conception finale et détaillée.

2.1 Analyse du problème et spécification fonctionnelle : Evolution et comparaison au livrable 1

Pour pouvoir jouer à "Lionheart", on doit tout d'abord connaître :

- le nombre de joueur (1 ou 2 selon si on se place en mode solo ou en multijoueur).
- le niveau auquel on veut jouer (niveau 1 ou 2)

Le mode multijoueur étant "plus raisonnable et accessible" à **concevoir** que le mode solo, puisqu'il n'y a pas besoin de programmer l'intelligence artificielle de l'ordinateur

Commençons donc par analyser le mode multijoueur :

- Nous pensions au départ à procéder à une sorte d'authentification des deux joueurs afin de démarrer une partie.

Solution retenue : En définitive , nous n'avons pas eu besoin d'utiliser une authentification étant donné que le jeu que nous avons réalisé est uniquement jouable par 2 joueurs sur un même ordinateur.

- Ensuite, nous pensions laisser chaque joueur choisir sa couleur et son côté (or ou argent ; En haut ou en bas) . Pour déterminer quel joueur allait commencer à jouer, nous pensions à différents systèmes :
- Au hasard, par un lancer de dés - ce que voudrait la règle du jeu
- ou par un choix arbitraire.

Solution retenue : Finalement, nos choix ont été les suivants : le premier joueur à jouer est le joueur Or, qui se situe au bord inférieur du plateau.

- **Concernant le niveau 2 : il devait permettre aux joueurs de placer leurs pièces selon leur choix avant de débiter la partie.**

Toutefois , nous n'avons pas développé cette fonctionnalité.

- Dans les deux cas (solo ou multijoueur), nous retrouvons les obligations de définir les différentes actions possibles (avancer, attaquer et tourner), ainsi que d'instaurer un système renvoyant aléatoirement une valeur entre les entiers de 1 à 6 (pour pouvoir attaquer). D'ailleurs, nous avons aussi réfléchi à la possibilité de laisser les joueurs gérer ces lancers de dés en leur proposant d'inscrire eux-mêmes la valeur des dés – certains pourraient ne pas croire au caractère aléatoire généré par

l'ordinateur et voudraient lancer les dés eux-mêmes; Il faudrait alors pouvoir les laisser attribuer un entier entre 1 et 6 pour chacune des valeurs {hache, flèche, panique} **et les autoriser à donner le résultat.**

Solution retenue : Finalement , nous nous sommes accordés que le modèle aléatoire généré par l'ordinateur était honnête et que les joueurs auront confiance en notre impartialité et en celle de l'ordinateur.

Les trois actions DEPLACER, ATTAQUER et TOURNER, ont bien été utilisées (le descriptif de ces actions sera expliqué dans la partie "conception").

- Enfin, il nous faudra définir toutes les pièces et, au final, décrire les conditions de la fin de la partie.

Tout ceci a été fait, l'explication de ces différents éléments est expliqué plus tard.

Nous pourrions alors tester notre programme en créant un tutoriel: cela reviendrait à enregistrer une partie entre deux joueurs, et la commenter.

Cependant , nous n'avons pas développé cette fonctionnalité.

Nous avons aussi pris des propositions concernant de futurs problèmes:

- Concernant le fait de revenir en arrière (Undo): on envisageait de l'interdire après un lancer de dés et donc de sauvegarder la partie après chaque lancer de dés.

Solution retenue : En définitive, c'est bien ce que nous avons fait. Nous n'avons pas créer l'action Undo compte tenu du fait que le joueur n'a pas de limite de temps pour effectuer son action et qu'une erreur de clic est très peu probable puisque chaque action autre qu'orienter (qui est réalisée à partir d'un clic sur la cellule depuis laquelle on effectue l'action , puis par un clic sur une flèche directionnelle (nord , est , sud , ouest)) nécessite 2 clics sur des bases : le premier sur la base depuis laquelle on effectue l'action, le second sur la base "cible" .

- En mode multijoueur: si un joueur décide d'arrêter la partie et que le second joueur veut la continuer en jouant contre l'ordinateur, le peut-il ?

On considère que ce cas n'arrivera pas, donc "abandonner une partie" ramènera automatiquement au menu principal.

Solution retenue : Ce problème n'en est plus un : nous n'avons pas développé de mode de jeu contre l'ordinateur; Et, il n'y a pas de menu principal , le jeu se lance par l'exécution de Lionheart sous Eclipse.

2.2 Conception préliminaire

Cette partie présente les pistes que nous suivions au moment de la réalisation du livrable 1.

A ce moment-là nous avons encore une idée assez vague de ce que nous allons vraiment utiliser. Néanmoins nous parvenions déjà à distinguer certaines des fonctions et modules que nous pourrions utiliser.

- *Main* : lancer le jeu, faire apparaître le menu
- *Menu* : le menu principal == interface avec le(s) joueur(s)
- *Jeu* : board, player1, player2, move_History, next_to_play, select_Pièces_type, start_game, announce_winner
- *Plateau* :
 - box, removed_pieces
 - Position(int x, int y)
 - OrientationPieces: nord, est, sud, ouest
 - ValeurSurCase : TypePiece + CouleurPiece
 - CouleurPiece: or ou argent
- *Pieces* :
 - CouleurPiece, PieceAt, Bougés
 - TypePiece: Roi, Cavaliers, Archers, Soldats, Mercenaires, Infanterie lourde, Paysans
- *Joueur* :
 - CouleurPiece
 - HumanPlayer
 - ComputerPlayer
- *Actions* :
 - Avancer
 - Tourner
 - Attaquer: lancerDé, dégâts
- *ActionsPossibles*:
 - TypePiece, ValeurSurCase
- *Simulation* :
 - SimulationDés
 - SimulationsParties
- *ValeurDés* :
 - Hache
 - Flèche
 - Panique
- *Compteurs* :
 - Dégâts
 - Points d'action
 - NbdePieces
 - TestVictoire
- *Ressources* :
 - toutes les images visibles possibles
 - Règles du jeu.pdf

Depuis lors, nous avons beaucoup réfléchi à comment mieux articuler les différents modules entre eux. Une grande partie de ces modules ont été soit abandonnés par manque de temps pour les mener à bien, soit changés, renommés et ajustés afin que les appels entre ces différents modules fonctionnent bien durant le codage qui a suivi cette conception préliminaire.

La conception détaillée qui suit, présente et explique le rôle des différents modules qui ont finalement été utilisés lors du codage du jeu .

2.3 - Conception détaillée

Cette étape de la conception fournit le contenu détaillé des différentes structures implémentées dans le code du jeu "LionHeart".

Pour faciliter la compréhension de chacun, les algorithmes présentés seront décrits à l'aide de pseudo-langage.

Nous allons commencer par décrire rapidement de proche en proche toutes les classes et énumérations nécessaires.

-> **class LionHeart()**
 public LionHeart()
 public static void main()

-> C'est la fonction centrale de notre code , c'est en exécutant celle ci que l'on lance le jeu et la virtualisation du plateau.

-> **class Plateau()**
 public Plateau()
 public void affichePlateau()
 public void updateCellule(int i, int j)
 public void initPlateauVide()
 public void initPlateau()
 public void initPlateauTestPanique()
 void cliqued(int i, int j)
 public int getPA_OR()
 public int getPA_ARGENT()
 public void initPlateauTestHelpArcher()

-> C'est à partir de cette fonction que l'on crée les paramètres du tableau de jeu (taille , image sur case , type de cases (unités , case vide , case noire ...)).

-> Cette fonction permet aussi au moyen de UpdateCellule , de mettre à jour l'état de la cellule après chaque action exécutée (orientation, attaque , déplacement , mort de la cellule..)

-> Les "init plateau" permettent d'associer à certaines cases des images comme les flèches directionnelles ou les dés. initPlateauTestPanique et initPlateauTestHelpArcher servent notamment à des tests dont les modalités seront détaillés dans la partie test.

-> Grâce à getPA_Or et getPA_Argent, on récupère les points d'action restants de chaque joueur . Ces points d'actions seront ensuite affichés sur le plateau de jeu par initPlateau.

-> Enfin, "cliqued" correspond à la logique du jeu. C'est à dire à l'exécution des actions (orienter , déplacement et attaquer). Elle fait appel à différents tests comme AttaquePossible et Déplacement possible et regarde si le compteur PA (= Points d'Action) est non nul et le décrémente une fois l'action réalisée. De plus à l'intérieur de

cette action , on a du code traduisant les règles du jeu (impossibilité d'attaquer 2 fois avec une même unité dans un même tour)

-> **class Cellule()**

```
    public Cellule(TypeCellule typeCellule, Plateau plateau, int positionX, int positionY)
    public int getPositionX()
    public void setPositionX(int positionX)
    public int getPositionY()
    public void setPositionY(int positionY)
    public void setPositionXY(int positionX, int positionY)
    class NotreActionListener
```

-> A chaque cellule sont associés 5 arguments : son TypeCellule , ses position x et y sur le plateau , son nombre de pièces et son orientation.

-> On récupère ces différentes positions pour les associer à une cellule sur le tableau.

-> NotreActionListener permet au logiciel de comprendre ce qui se déroule lorsque l'on clique sur un bouton.

-> **class TypeCellule()**

```
    private int nbDePieces
    private Orientation orientation
    public int getNbDePieces()
    public Orientation getOrientation()
    public TypeCellule(TypePiece typePiece, Couleur couleur, int nbDePieces, Orientation orientation)
    public TypeCellule(int nbDePieces)
    public String getTexte()
```

-> TypeCellule créé pour chaque état, un TypeCellule. C'est à dire que pour chaque pièce de même TypePièce , de chacune des 2 couleurs , de chaque orientation et de chaque nombre , il y a un TypeCellule correspondant .

Exemple : Il y'a un TypeCellule associé à chaque cellule ayant 4 soldats , orientée Nord de couleur Or.

-> On récupère le nombre de pièces et l'orientation pour les associer à une cellule sur le tableau.

-> TypeCellule(int nbDePieces) permet d'associer à un nombre de pièce négatif (ce qui n'arrive jamais) une image (autre que des images d'unités) à une cellule comme les flèches directionnelles et les dés. Un nombre de pièce nul correspondant à une cellule vide.

-> **enum Image**

```
    private int tailleCellule
```

-> Dans cet enum se trouvent toutes les images utilisées dans le tableau de jeu .

-> enum Action

-> Dans cet enum se trouvent toutes les actions effectuelles pas les joueurs : Orienter , Attaquer et Déplacer

-> class Attaque()

```
public boolean attaquePossible(int i1, int j1, int i2, int j2, TypePiece typePiece,
Orientation orientation)
public int degats(ValeurDe[] valeurDes, TypePiece typePiece0)
public void panique(TypeCellule typeCellule, int i, int j, Cellule[ ][ ] plateau)
public boolean mortCellule(int pdd, int nbDePieces, TypePiece typePiece)
public int nbPiecesRestantes(int pdd, int nbDePieces, TypePiece typePiece)
public boolean testvictoire(Cellule[ ][ ] plateau)
public boolean testPanique(ValeurDe[] valeurDes)
public void attaquer(int i1, int j1, int degats, TypeCellule typeCellule1, Cellule[ ][ ]
plateau)
```

-> Dans Attaque se situe attaquePossible qui teste en fonction de la position, de l'orientation et du typePiece de la cellule qui attaque (un archer pouvant attaquer dans un carré de 3 cases devant lui ; les autres unités du niveau 1 attaquant 1 case devant eux) , et la position de la cellule attaquée si l'attaque est possible.

-> La fonction "dégâts" renvoie un entier PDD correspondant aux dégâts infligés par une cellule à une cellule ennemie. Cette fonction dépend du résultat des dés et du type de pièce qui attaque.

-> La fonction panique correspondant au cas où tous les dés lancés ont pour résultat "Panique" . Dans ce cas Panique provoque le recul de la base qui attaque d'une case en changeant l'orientation dans la direction opposée à la direction initiale. Cette panique peut se transmettre au unités alliées dans le cas où la cellule serait devant une cellule alliée (autre que le roi , qui ne panique jamais) d'une case . La panique entraîne la mort de la cellule dans la cellule doit prendre la position d'une cellule ennemie. Enfin , si une cellule subissant "panique" venait à sortir des cases jouables du plateau , elle mourrait (cf mortCellule ci-dessous)

-> La fonction "MortCellule" teste si la cellule meurt , c'est à dire si le nombre de pièces sur la cellule est nul. Dans les faits , on regarde si les PDD (Points De Dégâts) sont supérieurs ou égaux au nombre d'unités restant sur la cellule.

-> Comme son nom l'indique , NbPiecesRestantes renvoie le nombre de pièces restantes après une attaque. On soustrait au nombre de pièces initialement présentes les PDD .

-> Après chaque résultats de lancer de dés , on applique testPanique pour voir si Panique s'applique.

-> Après chaque attaque , on applique testVictoire pour voir si la partie se termine. Concrètement , une partie se termine lorsque :

- toutes les pièces (autres que le roi) d'une même couleur meurent
- Le roi meurt

-> Enfin , on exécute attaque . Pour cela , on effectue d'abord le test AttaquePossible , puis l'on lance les dés en fonction du nombre de Pièces sur la cellule (2 dés pour 1 cavalier ou un roi , 1 dés pour les archers et les soldats).

-> **enum Couleur**

-> Dans cet enum se trouvent les deux couleurs : Or et Argenté

-> **class Deplacement()**

```
public boolean deplacementPossible(int i0, int j0, int i1, int j1, TypePiece  
typePiece, Orientation orientation, Cellule[ ][ ] plateau)
```

-> On effectue le test deplacementPossible pour lequel on regarde la position initiale de la Cellule , la position finale souhaitée , l'orientation de la cellule , sa position sur le plateau et le type de pièces sur la cellule (un roi et un cavalier pourront se déplacer de plusieurs cases en ligne droite ; les archers et soldats se déplaçant d'une case devant eux) . La notion d'obstacle pouvant rencontrer en compte dans le cas du déplacement du roi et cavalier est traitée . Si obstacle il y a , "Déplacement impossible" est renvoyé dans la console.

-> **class Des()**

```
public void affiche()  
public Des(int nbDePieces, TypePiece typePiece, Joueur joueur)  
public void listeValeursDes(int nbLancerDe, Joueur joueur)  
private ValeurDe conversion_Nombre_ValeurDe(int nombre)  
private int nbLancerDe(int NbDePieces, TypePiece typePiece)
```

-> affiche permet d'afficher le résultat littéral du lancer de dés dans la console .

-> Dans listeValeurDes , on prends 1+ la partie entière d'un nombre décimal compris entre 0 inclus et 6 exclu , délivré par mathrandom.

-> Dans ValeurDe conversion , On associe les valeurs 1,2,3 à une hache ; les valeurs 4,5 à une flèche ; la valeur 6 à un panique . La probabilité d'une hache étant de $\frac{1}{2}$, celle d'une flèche de $\frac{1}{3}$ et celle d'un panique de $\frac{1}{6}$.

-> Dans listeValeursDes , on associe une valeur à un résultat de dé (hache , flèche , panique) .

-> Dans NbLancerDe , on associe à une unité un nombre de dé lancable . Un archer et un soldat compte pour 1 dé tandis qu'un cavalier et un roi comptent pour 2 dés .

-> **enum Joueur**

-> Dans cet enum on associe une couleur à un joueur

-> **enum Orientation**

-> Dans cet enum se trouvent toutes les orientations disponible : Nord , Est , Sud , Ouest auquel on associe une lettre N pour Nord , E pour Est , S pour Sud , O pour Ouest .

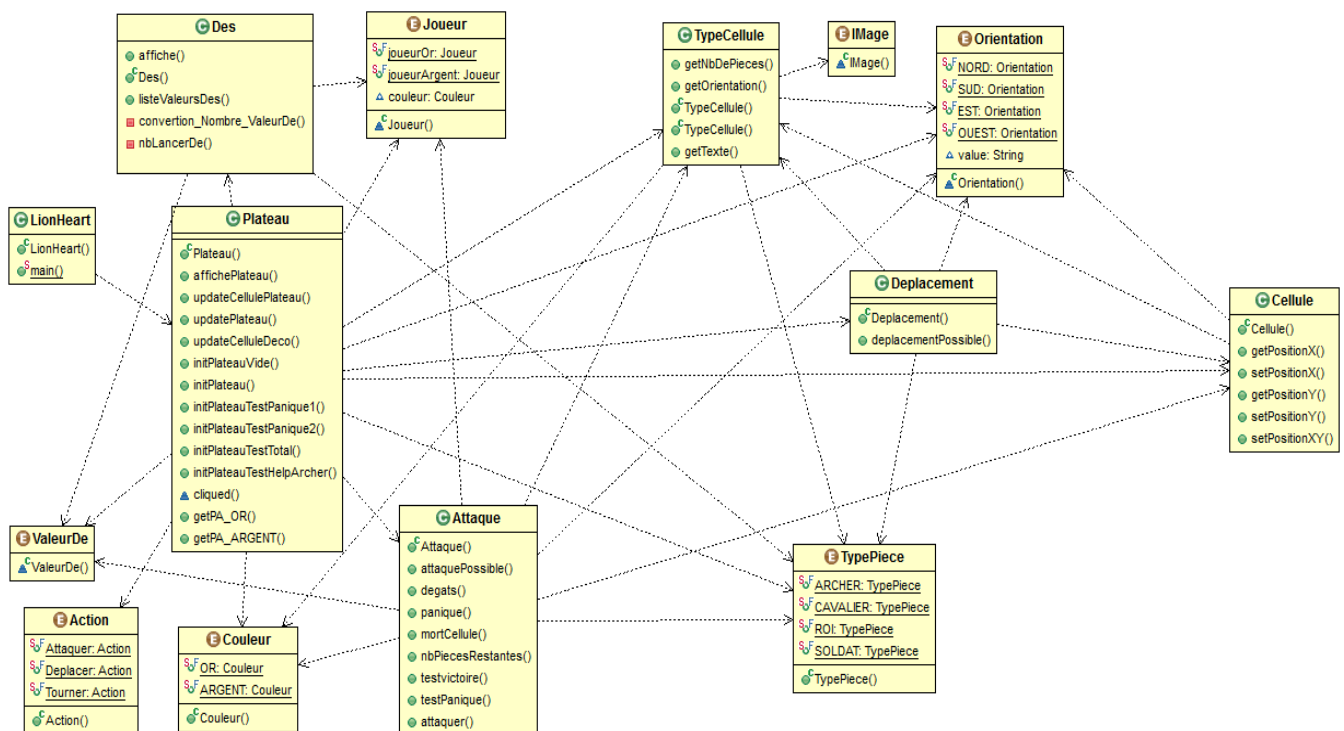
-> **enum TypePiece**

-> Dans cet enum se trouvent tous les types de pièces : soldat , archer , cavalier , roi.

-> **enum ValeurDe**

-> Dans cet enum se trouvent toutes les résultats de dés : Hache , Flèche , Panique pour lesquels on associe une lettre : H pour Hache , F pour Flèche , P pour Panique.

On peut résumer les appels entre class et enums sous le diagramme UML suivant :



3. Tests

Nous distinguons deux sortes de tests.

Les tests nécessaires pour le déroulement du jeu et les tests permettant de vérifier et de simplifier le codage du jeu.

Dans la première catégorie nous avons les test de possibilité d'actions à savoir attaquepossible, déplacementpossible ainsi que les tests d'états du jeu : testPanique, testvictoire, mortCellule

Les deux premiers permettent la validité de l'action envisagée par l'utilisateur. Ils sont utilisée à chaque fois que l'utilisateur veut effectuer un déplacement ou une attaque. Ils sont situés respectivement dans déplacer et attaquer.

attaquePossible(int , int, int, int, TypePiece, Orientation)

prend en argument les coordonnées de la base qui attaque ainsi que la base ciblée. le type de pièce de la base qui attaque (les archers ont une attaque différentes des autres pièces) ainsi que l'orientation de la base qui attaque.

deplacementPossible(int, int, int, int, TypePiece, Orientation, plateau)

prend en argument les coordonnées de départ ainsi que les cordonnées d'arrivées, le type de la pièce (les cavaliers et le roi peuvent se déplacer de plusieurs cases), l'orientation, ainsi que le plateau afin de connaître les dimensions ainsi que la positions des autres pièces sur le plateau.

Les trois test suivants interviennent pour connaître l'état du jeu. Ils sont tous les trois située dans la classe attaque. Ils ne sont en effet necessaire qu'à la suite d'une attaque.

testPanique (valeurDes)

intervient après un lancer de dès. Il permet de détecter l'état de panique et testant la valeur du lancé de dès

testvictoire (plateau)

Après chaque attaque on compte le nombre de pièces restantes ainsi que de roi restant par couleur. Si le roi meurt l'adversaire gagne, si la dernière unité meurt (le roi exclu) l'adversaire gagne.

mortCellule(int pdd, int nbDePieces, typePiece)

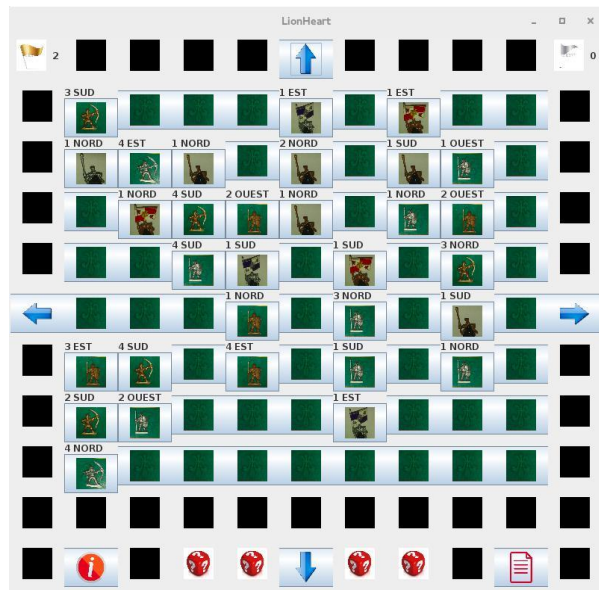
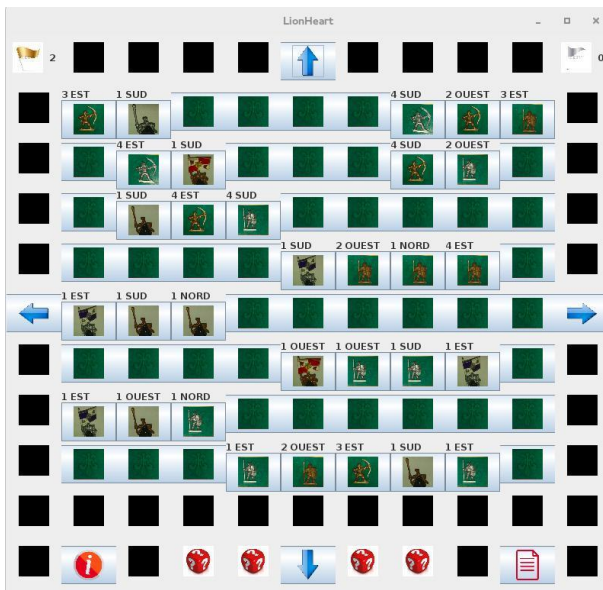
Après chaque attaque on regarde si le nombre de dégât infligés est supérieur ou égale on nombre de point de vie de l'unité visée. si c'est le cas alors la cellule est rendue vide. Le nombre de point de vie est fonction du nombre d'unité sur la base ainsi que du type de la pièce (les rois et cavaliers ont une résistance de 2)

Afin de tester la fonctionnalité des différentes méthodes nous avons mis en place plusieurs configurations initiales du plateau.

En effet le jeu se présente lui même comme étant un test fiable puisque chaque méthodes est accompagnée d'une écriture dans la console pour permettre au joueur de savoir ou il en est dans le jeu.

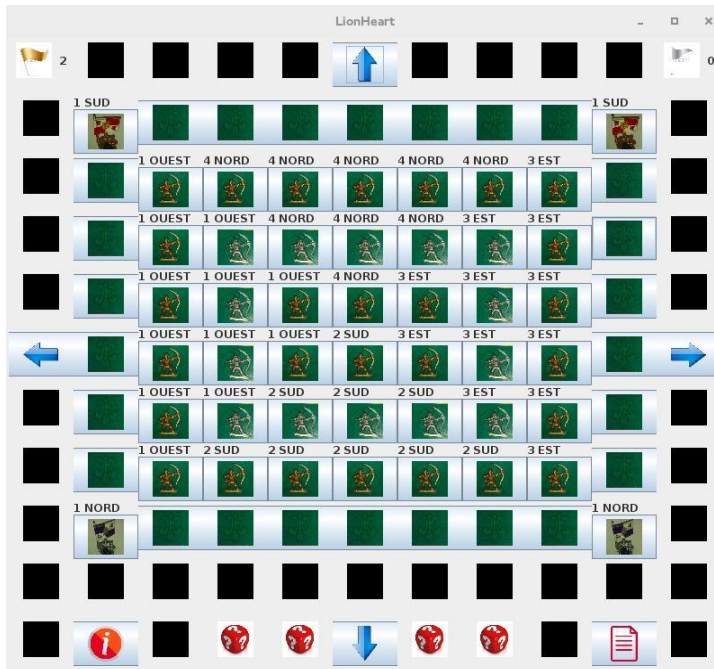
initPlateauTestPanique1 et 2 sont deux configurations nous permettant de tester l'état de panique.

Pour les activer il suffit de supprimer les '/' qui le précède dans Plateau et de passer initplateau() en commentaire de plus il faut forcer l'état de panique en passant de[i] = 6 dans Des.



initPlateauTestHelpArcher De même que pour l'état de panique, la commande Help nécessitait nombres de conditions et afin de vérifier que les affichages consoles correspondaient bien avec les actions possibles, nous avons initialiser le plateau de tel sorte à pouvoir rapidement tester l'intégralité des options et vérifier la cohérence de l'affichage console.

A l'instar du test précédant, il suffit de passer en commentaire initplateau() et de décommentariser notre test.



Conclusion

Cet exercice nous a permis de mener à bien un projet concret, en groupe, en partant de rien.

La première étape a été de repenser le jeu de plateau en un logiciel. Cette étape de préconception a permis à chacun d'apporter sa vision du jeu vidéo et donc d'imaginer de nombreuses fonctionnalités annexes ainsi que des développements éventuels que ne permet pas le jeu de plateau.

Une fois cette étape réalisée, il a fallu se concentrer sur un aspect bien précis du jeu afin de pouvoir jouer une partie basique dans son intégralité. Nous avons donc concentré nos efforts sur le niveau 1 de jeu opposant deux joueurs réels sur le même ordinateur.

Cet objectif a été réalisé.

Comme dans chaque projet, les débuts ont été lents puisqu'il a fallu trouver une dynamique de groupe et segmenter intelligemment le travail en fonction de chacun.

Fort de notre travail sur ce projet nous pourrions développer les autres fonctionnalités (niveau 2, campagne, menu) si le temps imparti était plus important.

Annexes

Les annexes sont découpées en trois parties: les ressources que nous avons utilisé, le manuel utilisateur et les règles du jeu détaillés et commentés de notre part.

A. Ressources

L'espace numérique Moodle de Télécom SudParis pour:

- la formation à SVN, un logiciel de gestion de versions
- la formation à la programmation d'interfaces de type graphique (awt)

B. Manuel d'utilisateur

LionHeart est un jeu de combat sur plateau au tour par tour durant la période du Moyen Âge. A la tête de votre armée, vous chercherez à annihiler votre adversaire en mettant en oeuvre toute votre stratégie militaire.

Pressé de rentrer dans la bataille?

Pour cela, il vous suffit de décompresser l'archive, de l'ouvrir sur Eclipse et de l'exécuter en tant qu'application Java.

Ensuite prenez le contrôle de votre armée et défiez un ami sur votre machine.

Un souci avec les règles?

La partie suivante pourra éclairer vos lanternes ainsi que la touche "Information" située en bas à gauche du plateau.

Bonne partie et que le meilleur gagne !

C. Règles du jeu

I - Règles de base

Ça se joue à 2 : 2 armées s'affrontent.

I.1 - Composition des armées

			
Le roi	Les cavaliers	Les archers	L'infanterie

On les place ensuite de la façon suivante:

		L'infanterie (4)	L'infanterie (4)	L'infanterie (4)	L'infanterie (4)	L'infanterie (4)		
		Les archers (4)	Les cavaliers (2)	Le roi (1)	Les cavaliers (2)	Les archers (4)		

Explication du nombre après le nom de la pièce: prenons, pour exemple, “Les archers (4)”. Cela veut dire que sur l’unité, il y a 4 archers. Cela donne ça :



I.2 - But du jeu

Il y a deux façons de gagner:

- éliminer le roi adverse
- éliminer tous les guerriers de l'armée adverse

I.3 - Déroulement du jeu

- Qui commence? Les 2 joueurs lancent les 4 dés à la fois chacun leur tour. Celui qui a obtenu le plus de haches de guerre commence à jouer.
- C'est à votre tour. Que pouvez-vous faire? Réponse: effectuer 2 des actions suivantes:

	Ce qu'on peut faire	Ce qu'on ne peut pas faire
AVANCER	<p>-L'infanterie et les archers peuvent seulement avancer d'une case</p> <p>-Les rois et les cavaliers (qui sont donc sur des chevaux...) peuvent eux avancer du nombre de cases qu'ils veulent, dans la rangée de cases devant eux</p> <p>Pour les 4 pièces ci-dessus, un déplacement = une action</p>	<p>-une unité ne peut pas reculer</p> <p>-interdiction de se déplacer en diagonale</p> <p>-interdiction de se déplacer à travers les autres unités (!= cavalier dans les échecs)</p>
TOURNER	<p>Que les unités tournent d'un quart de tour ou d'un demi-tour, cela compte pour une action.</p> <p><u>Remarque:</u> une unité peut donc reculer à la condition d'utiliser 2 actions (tourner et avancer) et de ne pas avoir d'unité derrière elles (à cause de l'interdiction de se déplacer à travers les autres unités)</p>	
ATTAQUER	<p>Chaque attaque = une action</p> <p>Une unité ne peut attaquer d'une fois par tour (ce qui est normal!), mais deux unités peuvent attaquer chacune pendant le même tour.</p> <p>Une fois en position pour attaquer, ANNONCER quelle unité adverse on vise, PUIS le joueur lance autant de dés que</p>	<p><u>Remarque:</u> Pourquoi tourner? Et bien parce qu'on ne peut attaquer, en général, qu'en étant en face de la cible. Interdiction d'attaquer sur les côtés par exemple!</p>



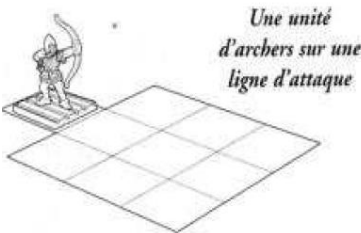

	l'unité (alliée) comprend d'hommes.	
--	-------------------------------------	--


Restons sur l'action d'attaquer. Les effets de celle-ci dépendent de la pièce... et des dés!

Sur 1 dé, il y a :

- 3 faces "haches de guerre"
- 2 faces "flèche"
- 1 face "panique"

Voici les détails des actions des pièces (il y a un résumé à la fin):

Attaquer avec...	Dégâts causés	Vies
 <p>L'infanterie</p>	<p><u>Position:</u> être face à l'unité adverse visée</p> <p>1 hache de guerre = 1 coup porté (1 chance sur 2)</p>	<p>1 coup porté pour l'éliminer</p>
 <p>Les archers</p>	<p><u>Position pour attaquer:</u></p>  <p>Les archers peuvent lancer des flèches au-dessus des troupes qui se trouvent entre eux et l'unité adverse visée (qui elle se trouve dans le carrée 3 x 3 devant les archers).</p> <p>1 flèche = 1 coup porté (1 chance sur 3)</p>	<p>1 coup porté pour l'éliminer</p>
	<p>Comme l'infanterie (1 hache de guerre = 1 coup porté), SEULEMENT chaque cavalier lance 2 dés!</p>	<p>2 coups portés pour l'éliminer</p> <p><u>Exception:</u> si l'unité qui attaque n'a plus qu'un seul homme d'infanterie</p>

Les cavaliers		ou un archer, on peut lancer le dé 2 fois, à condition d'obtenir un coup approprié au premier lancer.
 <p>Le roi</p>	Comme un cavalier.	Comme un cavalier.

PANIQUE!

- Obtenir le symbole "PANIQUE" sur tous les dés pendant l'attaque = l'unité complète (qui attaquait) panique, fait demi-tour: tourne de 180 degrés, puis avance d'une case.
- Les rois ne paniquent jamais



Cas particuliers:


- Si une unité, en pleine panique, déborde du plateau de jeu, ou est bloquée par son propre roi ou par n'importe quelle unité adverse, tous les hommes de cette unité sont éliminés
- Si elle est bloquée par une ou plusieurs de ses propres unités (mis à part le roi!), ces autres unités paniquent également!

II - Deuxième niveau de jeu: on agrandit l'armée!

Chaque joueur a 10 bases de départ sur le plateau de jeu (comme au I-). Seule différence: les unités autre que le roi peuvent être placées partout sur les 2 premières rangées. Dans le "jeu réel", on place même la boîte de jeu debout au milieu du plateau afin que chaque joueur puisse positionner son armée en secret. (et bien sûr, on enlève la boîte du plateau avant de jouer)

On peut aussi ajouter des guerriers supplémentaires:

Attaquer avec...	Dégâts causés	Vies
 <p>Les paysans</p>	<p>1, 2, 3 ou 4 paysans par base</p> <p>Liberté de choisir le nombre de paysans que l'on prend dans l'armée</p> <p>Déplacement, rotation, conditions pour attaquer : comme l'infanterie</p> <p>Ce qui change:</p> <ul style="list-style-type: none"> • <u>Avantage</u>: ils marquent des coups avec les haches de guerre OU les flèches (le "OU" étant malheureusement exclusif : il faut prendre en compte le symbole obtenu le plus souvent) • <u>Inconvénient</u>: après avoir attaqué, l'unité de paysans doit tourner et reculer d'une case pour chaque symbole "Panique" obtenu! 	<p>1 seul coup pour en éliminer 1</p>
 <p>Les mercenaires</p>	<p>Il ne peut y avoir qu'une unité de mercenaires dans chaque armée. Fixer un drapeau de la couleur de l'armée sur la base de l'unité de mercenaires.</p> <p>Chaque unité est composée d'1 ou 2 mercenaires</p> <p>Déplacement, rotation, position pour attaquer: comme l'infanterie</p> <p>Pour attaquer lancer 2 dés pour chaque mercenaire de l'unité!</p> <p>1 hache de guerre = 1 coup porté</p> <p>Symbole "Panique": l'inverse des paysans, c'est-à-dire que pour chaque symbole "Panique" obtenu, c'est l'unité adverse qui recule d'une case!</p> <p>ET: Les mercenaires ne paniquent jamais!</p>	<p>1 seul coup pour en éliminer 1...</p> <p>Mais est-ce le plus judicieux?:</p> <p>-Si le roi attaque des mercenaires adverses, il ne lance pas de dé, mais utilise une action pour enrôler dans son armée ces mercenaires.</p> <p>L'UNITÉ DES MERCENAIRES ACCEPTENT TOUJOURS LE MARCHÉ, SAUF si elle se trouve à côté de son propre roi et que celui-ci lui fait face. Dans le cas où elle accepte le marché, il faut remplacer son drapeau pour indiquer qu'elle a changé de camp!</p>

 <p>L'infanterie lourde</p>	<p>Ce qu'il faut retenir: <i>"elle avance lentement, mais est très dangereuse"</i></p> <p>Chaque base est composé d'1 ou 2 hommes.</p> <p>Elles avancent d'une case à la fois devant elles. 1 déplacement d'une case = 2 actions !!! 1 rotation = 2 actions !!! 1 attaque = 1 action (ouf!)</p> <p>1 hache de guerre = 1 coup porté</p> <p>Ils peuvent attaquer des unités ennemies situées sur TOUTES LES CASES ADJACENTES (devant, à côté, en diagonale, ou derrière)!!! (et ça, c'est cool!)</p>	<p>2 coups sont nécessaires pour en éliminer un.</p>
<p>L'infanterie massive</p> <p>= une unité de 10 hommes d'infanterie</p>	<p>Ces bases se déplacent et attaquent de la même manière que l'infanterie normale.</p> <p>Pour chaque attaque, tant que l'unité comprend plus de 4 hommes, il faut lancer 4 dés. (Si par exemple, il y a 7 hommes, on ne lance par 7 fois un dé!)</p>	<p>Comme l'infanterie normale: 1 coup porté = 1 homme éliminé</p>

Cela peut ressembler à ça:



(il n'y a pas unicité de la mise en place de départ. Seul le roi a une position pré-établie.)

III - Tableau de mise en place et d'action des Guerriers

Voici le fameux tableau récapitulatif:

Nom	Nb par joueur	Nb par case	Nb de dés lancés	Les coups portés	Nb de coups pour éliminer chaque personnage	Panique?	Commentaires
Roi	1	1	2	hache de guerre	2	Jamais!	-Avance aussi loin que possible tant qu'il n'est pas bloqué
Cavalier	4	2	2 par cavalier	hache de guerre	2	Oui	-Avance aussi loin que possible tant qu'il n'est pas bloqué
Infanterie	20	4	1 par homme	hache de guerre	1	Oui	
Archers	8	4	1 par homme		1	Oui	-Attaquent dans une grille de 3 x 3 et peuvent lancer leurs flèches au-dessus des autres unités
Infanterie de masse	20	10	max 4	hache de guerre	1	Oui	
Infanterie lourde	2	1 ou 2	2 par homme	hache de guerre	2	Oui	-Avancer ou tourner compte pour deux actions -Attaquer = 1 action -Peut attaquer verticalement, horizontalement ou en diagonale (donc pas nécessaire d'être en face pour attaquer)
Paysans	4	1, 2, 3 ou 4	1 par paysan	hache de guerre ou flèche	1	Oui... souvent!	-Reculer d'une case après une bataille réussie pour chaque symbole "Panique" obtenu
Mercenaires	2	1 ou 2 (+1 drapeau par mercenaire)	2	hache de guerre	1	Jamais!	-Les unités attaqués doivent reculer d'une case pour chaque symbole "Panique" obtenu -Change de camp s'il est attaqué par le roi adverse (sauf si son propre roi est juste à côté)

