CARBONNIER Nicolas
WYSOCKI Tom
WATELET Manon

L2 – MIASHS
Sciences Cognitives
2018 - 2019

Programmation Orientée Objet
ft Intelligence Artificielle
CASTAGNOS Sylvain – BOYER Anne

Projet de groupe – Rapport JAVA

Introduction

1. Descriptif des fichiers et des classes + UML
2. Présentation et fonctionnement de l'application
3. Présentation de l'Interface Graphique
4. Stratégie d'attaque (IA)

Introduction

Si le jeu Finstere Flure semble assez complexe dans sa stratégie d'attaque, il n'en est pas moins de sa création … et de sa programmation. Nous avons pu (du!) nous y atteler sérieusement , réalisant au fur et à mesure qu'il s'agissait plus qu'un simple jeu de l'oie .

Après quelques parties en équipe, on prend goût au jeu. Nous commençons à développer des pistes d'éléments de programmation. Progressivement, après avoir tracé les grandes lignes du jeu et simulé un début de partie, les nombreux détails et problèmes de résolution sont venus se mettre en orbite, pour au final se raccrocher à l'édifice du jeu, et y apporter sa contribution.

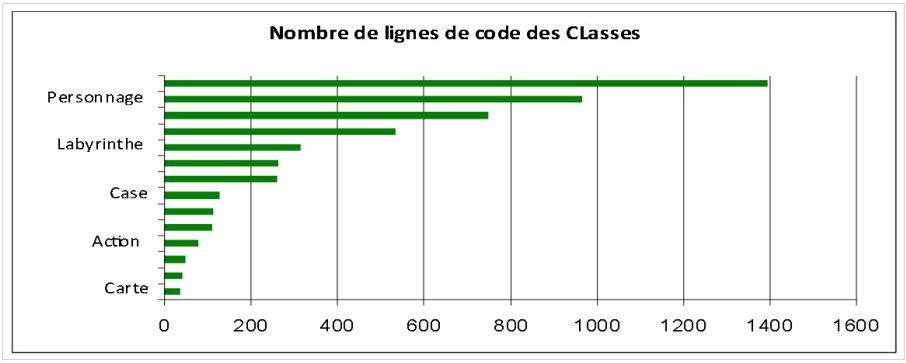
Superposition des maps et des classes, affichage du monstre, conditions de déplacement, graphisme… Le jeu de l'oie fut en réalité la réalisation des classes et méthodes ! Fort heureusement, à coup de sauvegardes et de GitHub, nous avons évité le pire et sommes arrivés, non sans encombres, à la case « compilation réussie ».

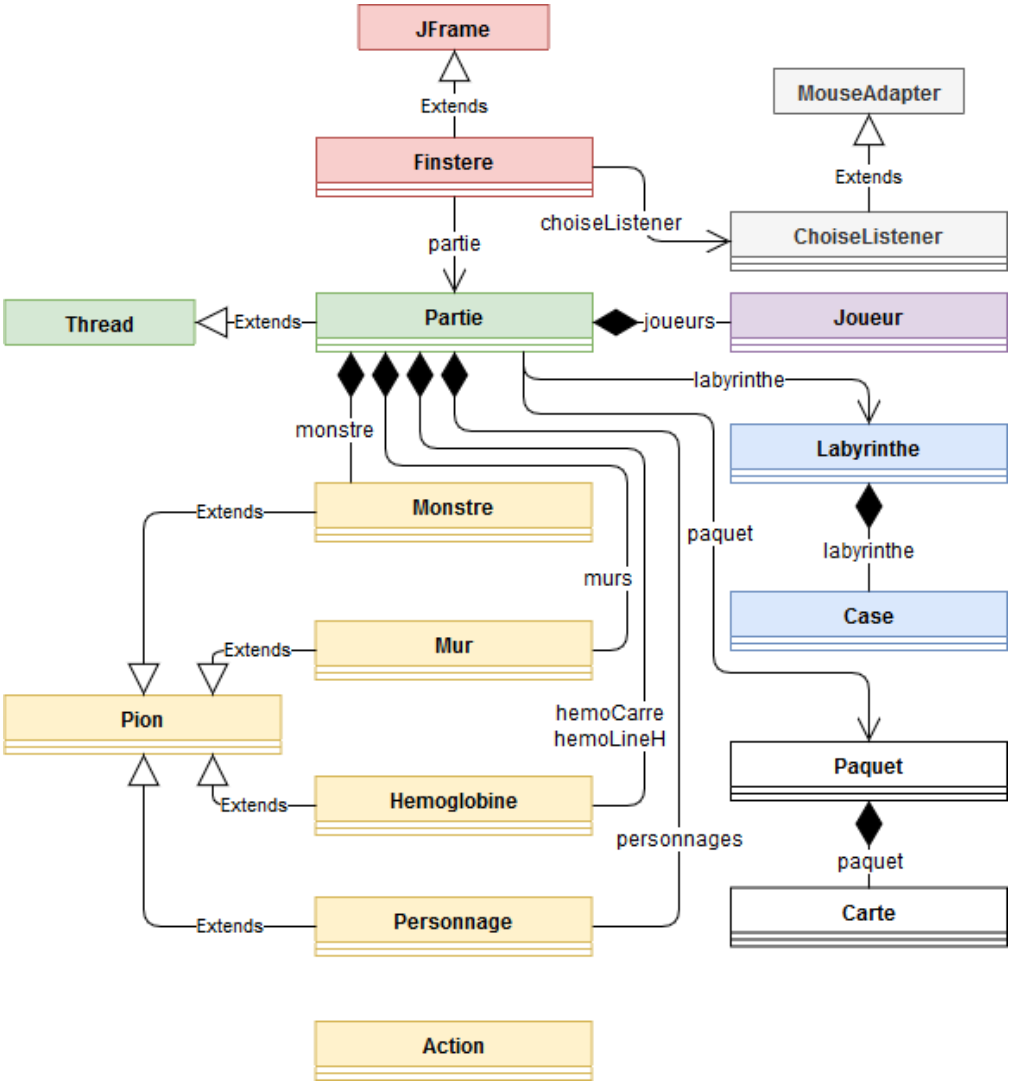
Le chemin n'est d'autant pas terminé. L'IA et la soutenance pourraient s'avérer être plus pénalisants que nous le pensons, bien que le hasard n'ai pas son mot à dire ici. Espérons qu'il s'agira d'avantage d'un tremplin !

En vous souhaitant une bonne lecture.
L'arrivée est proche ...

Nicolas, Tom & Manon

1. Descriptif des fichiers et des classes





Finstere

La classe `Finstere` correspond l'interface graphique du jeu et la classe principale. Elle est donc étendue de JFrame. Elle possède comme attributs tous les éléments graphiques de l'interface, ainsi que des constantes statiques utilisées dans les autres classes de programme (directions, couleurs, type des flaques d'hémoglobine…).

Cette classe possède également des constantes utilisées par l'interface, pour définir les dimensions d'une case, les coordonnées du début de la grille du labyrinthe et les couches des JlayeredPane pour chaque type d'élément (plateau, monstre, mur, personnage, action…).

Parmi ses attributs privés, on retrouve la `partie`, qui contient la boucle de jeu et les éléments fonctionnels du jeu, ainsi que des **JLabel** pour afficher les éléments dynamiques de l'interface tels que le monstre ou les personnages. La classe possède un entier `choix` qui sera récupéré par la classe partie pour connaître le choix du Joueur quant au personnage sélectionné ou à l'action qu'il doit réaliser.

Les méthodes de cette classe sont principalement d'ordre graphique, pour par exemple mettre à jour la position des personnages ou du monstre sur le labyrinthe, ou encore pour actualiser les informations telles que le tour, la manche ou l'historique de déplacement du monstre.

Certaines méthodes vont afficher une seconde fenêtre pour lancer une nouvelle partie, saisir le nom des joueurs ou encore informer de la fin de la partie et du nom du gagnant.

Partie

La classe `Partie` correspond à la structure du jeu et à la boucle de jeu. Cette dernière est étendue de Thread afin de mettre en pause la boucle lorsqu'un joueur doit effectuer un choix (personnage puis action) lors d'une partie avec l'interface graphique (dans une partie avec le terminal, la boucle sera obligatoirement bloquée par le Scanner).

Elle possède des attributs pour les éléments du jeu (labyrinthe, joueurs, personnages, monstre, murs…) ainsi que pour les informations de la partie (numéro de tour et de manche, partie dans le terminal ou non…).

Parmi ses méthodes on y retrouve notamment une méthode pour initialiser `la Partie`, en créant les joueurs et leurs personnages, ainsi qu'une surcharge de cette dernière lorsque la partie se déroule dans le terminal et que les Joueurs doivent saisir leurs informations sur ce dernier. On retrouve aussi la méthode chargée de générer le labyrinthe et d'y disposer les flaques d'hémoglobine et les murs.

L'élément le plus important de cette classe est la boucle de jeu, représentée par la méthode `run()` qui est une réécriture de la méthode `run()` de la classe Thread. C'est depuis cette méthode que la partie va se dérouler. En demandant aux joueurs de choix un personnage puis une action, puis en déplaçant le monstre et en cherchant un gagnant après chacune des actions.

Pour plus de clarté dans le code, la méthode `run()` fait appel à d'autres méthodes de la classe. Notamment, la méthode `coupJoueur`, dont la signature comporte l'élément synchronized, permet d'utiliser la méthode `wait()` qui permet de mettre en pause la boucle.

Labyrinthe

La classe `Labyrinthe` correspond au labyrinthe de la partie. Il est caractérisé par un tableau à deux dimensions de cases.

Cette classe possède notamment une méthode pour générer le labyrinthe, en y créant les cases et en définissant les possibles coordonnées de téléportation du `Monstre` pour chacune d'elles.

Les setters et les getters de cette classe possèdent en paramètre des coordonnées x et y car ils font appel à ceux de la case de coordonnées (x, y) .

Case

La classe `Case` correspond à une case du `Labyrinthe` caractérisée par des coordonnées x et y, deux booléens pour déterminer si la case est en bordure ou si elle est bloquée, ainsi qui par une Map `tp` dont la clé correspond à une direction et la valeur aux coordonnées de téléportation (la Map sera vide si la case n'est pas en bordure).

Une case dite bloquée est une case qui ne fait pas partie du labyrinthe, vu que ce dernier n'est pas totalement rectangulaire. L'attribut `tp` permettra de déterminer si le monstre doit être téléporté en fonction de sa position et de sa direction.

Pour simplifier l'appel au constructeur, celui-ci dispose de trois surcharges pour les cas suivants : case en bordure, case normale et case bloquée.

Pion

La classe `Pion` correspond à un pion du jeu caractérisé par des coordonnées x et y.
Un pion peut être un personnage, un monstre, un mur ou une flaque d'hémoglobine.

Personnage

La classe Personnage correspond à un personnage qui, en plus de ses coordonnées, est caractérisé par ses points de mouvement pour chaque face et sa couleur. Il possède également d'autres attributs, notamment des booléens : pour déterminer sa face actuelle, si il a été joué durant le tour, si il est mort ou sorti. Également pour déterminer son classement une fois qu'il est sorti.

Un personnage possède aussi deux booléens `auDessus` et `enDessous` qui permettent de savoir quel personnage doit être affiché dans le cas où deux personnages se retrouveraient aux mêmes coordonnées suite à un déplacement.

Les méthodes de la classe `Personnage` correspondent principalement aux actions qu'un personnage peut effectuer (se déplacer, pousser un mur, glisser sur une flaque et sortir) ainsi qu'à des méthodes pour déterminer si une action est réalisable.

La méthode la plus important de cette classe est la méthode getActions() qui retourne une Map<Integer, Action> qui construit et retourne la liste de toutes les actions qu'un personnage peut réaliser au moment de l'appel de la méthode. Il s'agit donc de la méthode appelée lors ce que le jeu va proposer les actions au joueur.

Monstre

La classe Monstre correspond au monstre du jeu qui, en plus de ses coordonnées, est caractérisé par une direction.

Les méthodes de cette classes permettent principalement de déplacer le monstre en fonction de la carte tirée et de téléporter celui-ci si jamais la case sur laquelle il est est une bordure et si sa direction correspond à un élément de l'attribut `tp` de la case en question. Ces méthodes retournent une liste de personnages qui correspondent aux personnages dévorés ou écrasés par le monstre lors de son déplacement.

La classe possède également une méthode qui permet de déterminer la direction que le monstre doit prendre en fonction des personnages qu'il peut voir.

Mur

La classe `Mur` correspond à un mur, caractérisé uniquement par ses coordonnées.

Cette classe possède deux méthodes qui permettent de pousser le mur, la première est appelée lorsqu'un personnage pousse le mur, et la seconde lorsque le monstre pousse le mur. La différenciation est nécessaire car le monstre peut pousser n'importe quel mur et peut possiblement écraser un personnage (auquel cas celui-ci sera retourné par la méthode) alors qu'un personnage peut pousser uniquement les murs dits `poussables` dans la direction donnée.

La méthode `poussable()` permet donc de déterminer si un mur est déplaçable dans la direction donnée.

Hemoglobine

La classe `Hemoglobine` correspond à une flaque d'hémoglobine caractérisée par son type (carré ou linéaire horizontale) et ses coordonnées.

Elle possède des méthodes pour réaliser la glissade d'un mur ou d'un personnage, ainsi que des méthodes pour déterminer les coordonnées d'arrivée d'un pion, la zone d'interaction avec la flaque ainsi que si des coordonnées données sont sur la flaque ou non.

Joueur

La classe `Joueur` correspond à un joueur. Il est caractérisé par son nom et ses personnages. La classe possède également un attribut `nbRestant` qui représente le nombre de personnages restant au joueur (non morts et non sortis), utilisé dans la boucle de jeu afin de ne pas bloquer la partie si jamais il reste moins de personnage à l'un des joueurs.

Cette classe possède également une méthode qui retourne une `Map` qui correspond à la liste des personnages encore jouable pour ce tour. Cette méthode est appelée lorsque le jeu demande à un joueur de choisir un personnage.

Action

La classe `Action` correspond à une action réalisable par un personnage. Elle est caractérisée par une description de l'action, un nom de méthode, une méthode (de type `Method`), un tableau de paramètres et un coût.

Cette classe permet la proposition dynamique des actions des personnages car la méthode appelée, une fois le choix effectué, est stockée dans l'objet de type Method, qui sera appelé dans la boucle de jeu avec comme paramètres ceux contenus dans le tableau en attribut de la classe.

La coût est utilisé pour afficher la bonne image pour l'action en question lors d'une partie avec l'interface graphique.

Paquet

La classe Paquet correspond au paquet de `cartes`. IL est caractérisé par un tableau de `carte` et deux listes de `String` pour les cartes dans la pioche et dans la défausse. Les listes sont utilisées dans l'interface graphique pour afficher la liste de la pioche est de la défausse.

La principale méthode de la classe permet de mélanger les cartes. Elle est appelée au début de chaque manche et intègre une boucle qui permet de s'assurer que la carte « Une proie » ou « Deux proies » ne sera pas piochée en premier.

Carte

La classe Carte correspond à une carte, caractérisée par sa valeur.

ChoiceListener

La classe `ChoiceListener` correspond à l'écouteur utilisé par l'interface graphique lors du choix du personnage ou de l'action. Elle est étendue de la classe `MouseAdapter` et réécrit la méthode `mouseClicked`.

Ainsi lors d'un clique sur un JLabel ou un JButton ayant cet écouteur, la méthode mouseClicked définira l'attribut choix de `finstere` avec l'entier correspondant au personnage ou à l'action choisi puis notifiera la `partie` que le choix a été effectué et que la boucle de jeu peut reprendre.

2. Présentation et fonctionnement de l'application

→ La présentation ...

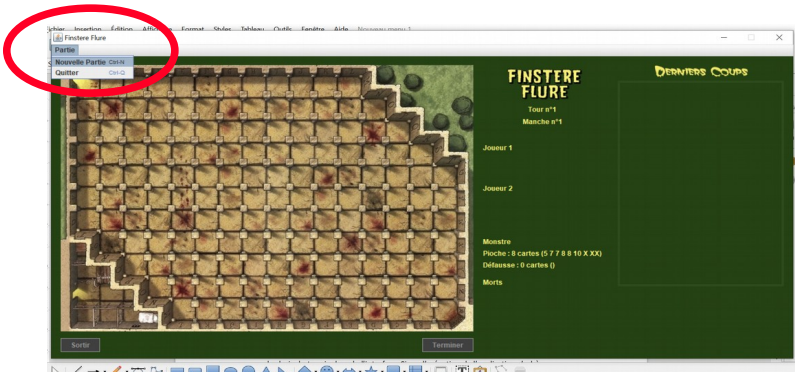
Notre application est un logiciel Java permettant de jouer au jeu de plateau Finstere Flure. Pour les besoins du projet, notre version du jeu est conçue uniquement pour deux joueurs. Bien que renseigné comme possible par l'interface graphique, l'utilisation du bot n'est pas possible. Le joueur 2 portera le nom de « Bot » mais sera contrôlé par l'utilisateur

L'application dispose d'une interface graphique, affichée dès le lancement du programme, mais permet tout de même de lancer une partie dans le terminal (le lancement se fera depuis l'interface graphique).

→ ... et le fonctionnement

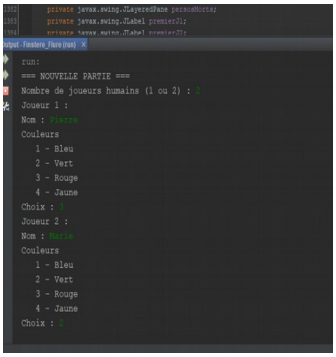
L'installation de notre application est relativement simple. En effet, puisque nous n'avons atteint que le **défi 3**, notre application ne dispose pas d'une base de données et donc l'installation se fait juste en ouvrant le répertoire du projet depuis NetBeans.

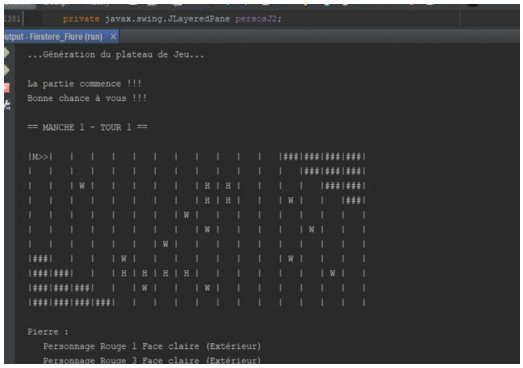
Pour lancer l'application, il faut exécuter le fichier Finstere.java qui est l'interface graphique de notre programme, ainsi que la classe principale. La fenêtre principale de l'application s'ouvre alors. Pour lancer une nouvelle partie, il faut ouvrir la fenêtre de création de partie, soit depuis le menu Partie > Nouvelle Partie, soit directement avec le raccourci clavier Ctlr+N. Deux possibilités s'offre alors à vous : lancer une Partie dans le terminal ou lancer une Partie avec l'Interface.



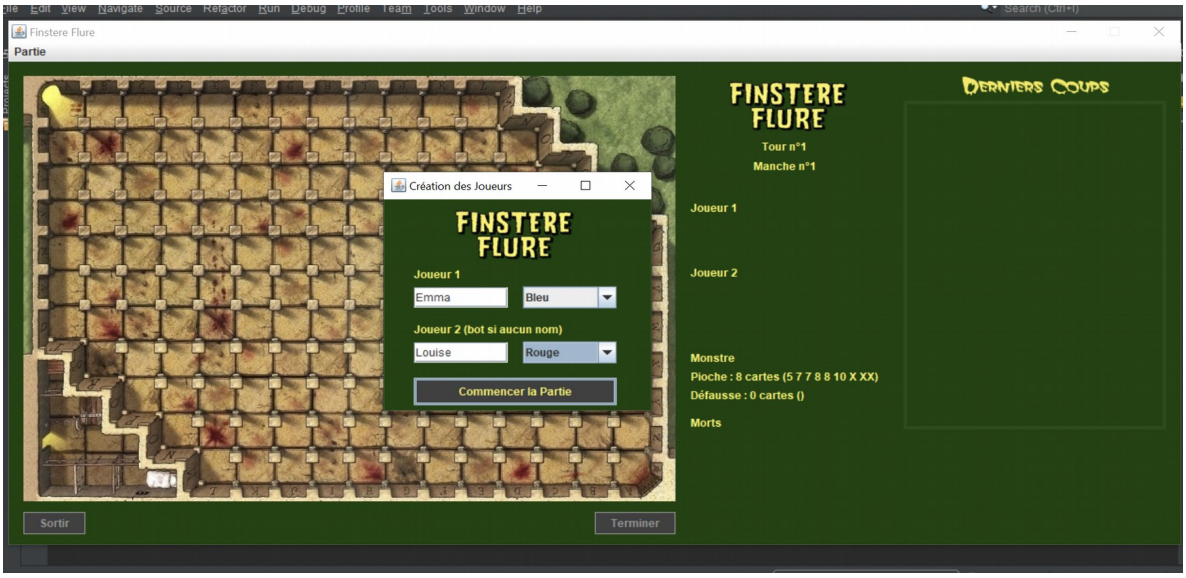
1
2

Si vous choisissez la première option, la fenêtre se ferme et la suite de l'exécution se fait dans le terminal. Il vous sera alors demander de saisir le nombre de joueurs humains (1 ou 2) puis de choisir leur nom et la couleur de leurs personnages. La partie se lance ensuite et les joueurs devront successivement choisir un de leurs personnages, puis une ou des actions à réaliser avec ce dernier. La partie se déroule donc comme une partie de Finstere Flure classique. À la fin de la partie, une fois le gagnant affiché, il vous sera demander si vous souhaitez relancer une partie ou non. Si vous choisissez de relancer une partie, l'interface se ré-affiche, avec la fenêtre de création de partie, vous laissant de nouveau le choix du terminal ou de l'interface. Sinon l'exécution de l'application s'achève.





Si vous choisissez de lancer une partie avec l'Interface, une nouvelle fenêtre s'ouvre et demande le nom des joueurs ainsi que la couleur de leurs personnages. Le bouton Commencer la Partie reste grisé tant que aucun nom n'a été saisi pour le Joueur 1 ou tant que la couleur des personnages des deux joueurs est identique dans le cas où un deuxième nom est saisi. La partie peut ensuite commencer, les joueurs devront choisir un de leur personnage depuis la liste à droite du labyrinthe, puis choisir une action à réaliser. Les boutons Sortir et Terminer s'activent en fonction de la situation, si le personnage a suffisamment de points de mouvement pour sortir pour le bouton Sortir, et si le personnage est sur le labyrinthe et pas sur un autre personnage pour le bouton Terminer (qui fera donc passer la main au joueur suivant). Les utilisateurs ont la possibilité d'interrompre la partie à tout instant en en relançant une nouvelle, ou en fermant la fenêtre, qui arrêtera l'exécution de l'application.



→ Bonus : les bugs notables

À l'heure actuelle, le seul bug connu est dans la proposition de certaines actions. En effet, nous n'avons pas réussi à éliminer certains déplacements normalement impossibles. La zone de déplacement autour d'un personnage étant un cercle de rayon les points de mouvement du personnage, il nous a fallu éliminer les cases inaccessibles, augmenter le coup d'un déplacement s'il était nécessaire de contourner un obstacle, etc. Mais nous n'avons malheureusement pas trouvé de condition pour éliminer des possibilités dans une situation comme celle montrée ci-dessous.


2
5
2
1
2
2
1
4
1
2
1
2
1
2

3. Présentation de l'Interface Graphique

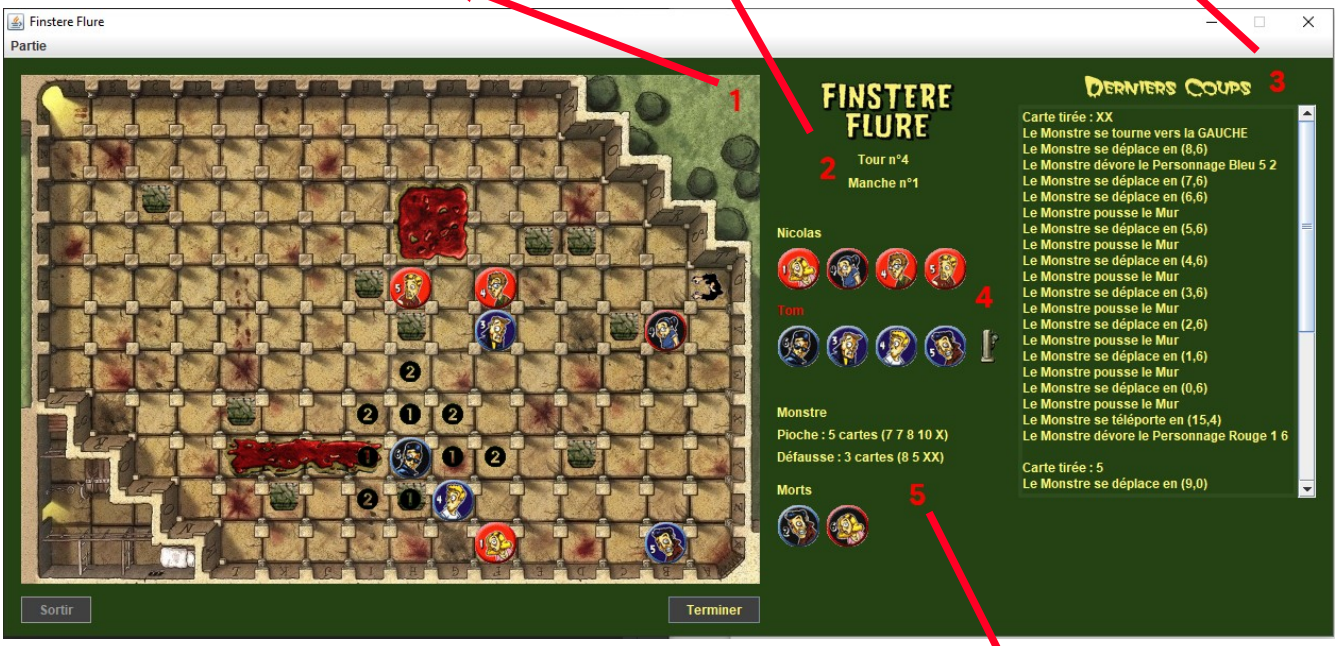
a) Fenêtre principale

La fenêtre principale du logiciel dispose de tous les éléments important pour le déroulement de la partie :

❶Plateau de jeu : il s'agit du labyrinthe sur lequel seront affichés les personnages sortis, ainsi que le monstre, les murs, les flaques d'hémoglobine et les actions possibles lorsque vient le moment de choisir une action

❷Information sur la partie : c'est ici que sont affichées les informations sur le tour et la manche en cours

❸Derniers coups : sont affichés ici les déplacements du monstre, sous forme textuelle avec la carte tirée, les changements de direction, les personnages dévorés, etc



❹Liste des personnages : c'est ici que l'on visualise les personnages de chaque joueur, on peut ainsi savoir si un personnage est joué, sorti ou mort à tout moment. La pierre de premier joueur indique le premier joueur pour ce tour et le nom du joueur devient rouge quand vient son tour de jouer. Les personnages qu'il pourra jouer seront alors affiché avec un carré translucide juste au dessus, le joueur aura juste à cliquer sur l'un d'eux pour le choisir

❺Informations sur le monstre et liste des morts : on retrouve dans cette zone la liste des cartes de la pioche et de la défausse, ainsi que la liste des personnages dévorés ou écrasés durant le dernier déplacement du monstre

b) Nouvelle partie

Cette fenêtre s'affiche lorsque l'utilisateur demande à créer une nouvelle partie, soit depuis le menu Partie > Nouvelle Partie, soit avec le raccourci Ctrl+N. Elle offre deux options : lancer une partie dans le terminal ou lancer une partie avec l'interface graphique.

c) Création des joueurs

Cette fenêtre s'affiche si l'utilisateur demande à créer une partie avec l'interface graphique. Elle dispose de deux zones de saisies pour les noms des joueurs et de deux menus déroulants pour le choix des couleurs.

d) Victoire (si vous parvenez jusque là …)

Une fenêtre s'affiche à la fin de la partie si celle-ci se déroule avec l'interface graphique. Elle informe du gagnant, des personnages qu'il a réussi à sortir et propose de créer une nouvelle partie.
Ouf, tout est bien qui finit bien.

4. Stratégie d'attaque (IA)

Tous les problèmes étudiés jusqu'à présent se présentaient sous cette forme : on avait un problème et une solution finale à laquelle aboutir (jeu du taquin, cannibales et missionnaires etc.).
Pour cela plusieurs chemins sont parfois possibles et on utilisait 3 algorithmes piliers dans la résolution de ce problème :
- Un **SSALGO But?** qui vérifie si l'état final e_f (=solution attendue) est atteint ou non après chaque opération effectuée
- un **SSALGO Successeur** qui génère tous les états e_{n+1} à partir d'un état e_n
- un **SSALGO Recherche** qui donne la liste des chemins qui mènent jusqu'à cet état final (selon le nombre de possibilités) on peut retourner : une liste vide, d'un seul élément ou plusieurs) autrement une la liste de toute les actions vers le résultats final attendu.

Dans le cas du jeu **Finstere Flure**, on a une omniscience sur la map, où tout est observable (pions alliés, adverses, monstre et la nature de la case – mur, flaque de sang ou occupée) mais on a un aspect indéterministe ou imprévisible pour les déplacements adverses et ceux du monstre.
C'est pour cela que l'usage du **SSALGO But?** est plus complexe à mettre en place et est différent de ceux qu'on a connu jusqu'à présent, dans la mesure où le **SSALGO Successeur** retourne tous les coups n+1 accessibles (au coup n) et que l'environnement évolue après chaque coup exécuté.

Le **SSALGO Recherche** retourne, quant à lui, la liste **lvictoire** de ceux qui favorisent une stratégie gagnante (en se rapprochant le plus de la sortie, en sortant, ou en sauvant un pion proche du monstre) à chaque étape.