LOPES Guillaume
LARMURIER Cyril

12 / 01 / 2005

Rapport de Programmation Orientée Objet

Borodino

Licence 3 informatique

Objectif

On déterminera après analyse puis on implémentera en langage java les structures objets réalisant un jeu baptisé **Borodino**.

Dans cet exposé, nous nous attacherons à détailler les particularités de notre programme, les réflexions menées, les raisons de certains choix, en particulier ceux liés aux concepts objets réalisés, et les difficultés rencontrées.

Règles du jeu

Les pions sont représentés sous deux formes, et ils possèdent trois paramètres.

- Les **pions-joueurs** qui représentent les joueurs humains.
  Chaque participant possède son propre pion et le manipule à son grès.
  La force et la vitalité des pions-joueurs sont initialisées à 100.
  Initialisation :
  Force = [100]
  Vitalité = [100]
  Héritage = [0 ; 25] %

- Les **pions-méchants** qui représentent les joueurs simulés.
  Ces pions sont pilotés par la machine, et répondent à l' I.A. implémentée.
  La force et la vitalité des pions-méchants sont générées aléatoirement dans l'intervalle [65 ; 135]. Cet écart-type de 35 a été choisi après de nombreux tests, il offre un bon compromis du rapport victoires / défaites.
  Initialisation :
  Force = [65 ; 135]
  Vitalité = [65 ; 135]
  Héritage = [0 ; 25] %

  Sachant que l'héritage est *la vitalité transmise à son adversaire lorsque celui-ci le vainc* et que la force *augmente à chaque combat gagné*, nous avons décidé que l'héritage soit un pourcentage initialisé aléatoirement mais fixe dans le temps, et dans l'intervalle [0 ; 25]. Il se révèle déterminant lorsque l'on doit choisir entre deux pions à peu près aussi faibles. L'appât du gain défini par l'héritage nous indique le « gibier » le plus intéressant : celui qui a l'héritage le plus important.

Un pion ne passe d'une case à une autre que si les deux cases concernées sont directement voisines.

Lorsqu'au moins deux pions se situent sur la même case, un combat à mort s'engage, à l'issue du-ou-desquel(s) il ne reste qu'un seul gagnant. (Un combat ne faisant intervenir que deux protagonistes à la fois.)

Les dommages sont fonction de la force de frappe de l'attaquant, et constitue pour le défenseur une perte correspondante de vitalité.

Dommages :

vitalité de la victime = vitalité de la victime - `random`(force de l'attaquant)

Héritages :

force du gagnant = force initiale du gagnant + héritage* du perdant x force du perdant

vitalité du gagnant = vitalité initiale du gagnant + héritage* du perdant x vitalité du perdant

*On rappelle que l'héritage est un pourcentage, et qu'il est fixe dans le temps.

Comment le jeu se termine-t-il ?

Même si les pions-méchants peuvent se manger entre eux, et qu'ils ont une véritable indépendance du joueur humain, il nous est apparût peu intéressant de continuer la partie dès lors qu'il n'y a plus de pions-joueurs, ce qui est synonyme d'échec. Dans le cas contraire, le jeu s'arrête s'il ne reste plus de pions-méchants et un seul pion-joueur victorieux.

La stratégie

Ainsi, chaque pion peut se déplacer dans l'une des quatre directions cardinales ; cependant lorsqu'un pion est à la portée d'un autre dans sa diagonale, il peut être attaqué, et ce quelque soit son type. Cette particularité est due à l'intelligence artificielle des pions-méchants, qui était infaillible et nous faisait tourner en rond indéfiniment. Il fallut introduire une brèche : « l'attaque diagonale ».

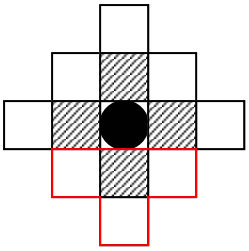
On a pensé l'intelligence artificielle en fonction de trois cas :

- Au moins un pion plus faible est repéré.
- Aucun pion plus faible n'est trouvé, quelles directions sont libres ?
- Au moins un pion plus faible et au moins un pion plus fort sont présents.

Les pions-méchants possèdent une « vision » égale à un tiers du côté le plus petit du damier. Une vision est une zone carrée autour du pion, dans laquelle tout pion plus faible que lui est repéré. Si plusieurs sont repérés, le plus faible d'entre tous est choisi ; il se peut aussi que plusieurs pions soient aussi faibles les uns que les autres (cas probable en début de partie avec plusieurs pions-joueurs à 100), et dans ce cas, le plus proche est pistée. Lors de la poursuite, il commence par combler le retard le plus important entre la longueur et la largeur, de manière à ne pas le perdre.

Lorsqu'un pion-méchant piste un pion plus faible que lui et que celui-ci est attaqué par un autre assaillant, le pion-méchant ne se jette dessus que si la case ne contient que des pions plus faibles. C'est un choix de précaution qui va dans le sens du pion-méchant ; quant aux pions-joueurs, ils peuvent aussi aller au combat et s'ajouter à la liste des duels qui auront lieu sur la même case.

Dans ce cas précis, les duels s'engagent alors au hasard, et ceci pour ajouter un peu de suspens. Il est à noter que l'assaillant n'a pas forcément la primeur de l'attaque, toujours pour ne pas avantager un joueur plus qu'un autre pendant un combat.
Pourquoi pas de vision totale ?
Si tel avait été notre choix, tous les pions se seraient agglutinés vers le même pion, et il aurait fallu temporiser avec la distance à un pion moins faible plus près. Le pion aurait eu une seule politique : l'attaque, ce qui est peu naturel et laisse peu de place à la stratégie.
L'effet produit par une vision plus courte offre un compromis entre la découverte rapide d'un pion faible et des scènes d'actions décentralisées. En outre, ce choix développe l'aspect stratégique du jeu, où les notions d'itinéraire et d'« évitement » prennent tout leur sens.
En effet, le pion simulé montre des réactions très proche d'un joueur : il évalue son environnement, ou plutôt sa « zone », et choisit entre l'attaque d'un pion plus faible et la défense qui consiste à fuir un pion plus fort dans la direction opposée, dans l'espoir de croiser un autre pion plus faible que lui. Le plus faible serait alors soit mangé par le pion simulé, qui aurait plus de chances de battre son poursuivant ; soit privilégié par son poursuivant. Enfin, s'il ne croise personne, le combat avec le pion plus fort devient inévitable au bout d'un certain temps.
Cette fuite est aussi liée à un système de vision particulier, détaillée sur le schéma ci-après :



Les cases grisées représentent ses déplacements potentiels, en supposant qu'il n'y ait aucun pion plus faible que lui dans l'une de ses diagonales.

La zone délimitée en rouge représente les cases de la direction Sud, celles-ci doivent être vide pour qu'il aille dans cette direction. Si toutes les directions sont occupées par des pions plus forts, le pion attend dans un ultime espoir jusqu'à ce que l'une des directions se libère, sinon il choisit au hasard une direction parmi celles qui sont inoccupées. Les trois cases blanches d'une direction doivent être testées car elles sont autour du déplacement potentiel grisé, et si un autre pion plus fort se trouve dessus et qu'il n'a pas encore joué (déplacement), on est à sa merci.

En conclusion, toute la stratégie du jeu consiste à manger les pions les plus faibles avec les héritages les plus importants, et ce avant les autres. En un mot, c'est un jeu capitaliste, où plus on est fort et plus on peut manger. En dépit de cela, la chance (représentée par le générateur pseudo aléatoire), est là pour ajouter une touche de piquant : ne pas être obligé de remporter un combat lorsque celui-ci semble gagné, ou vaincre à un moment inattendu…

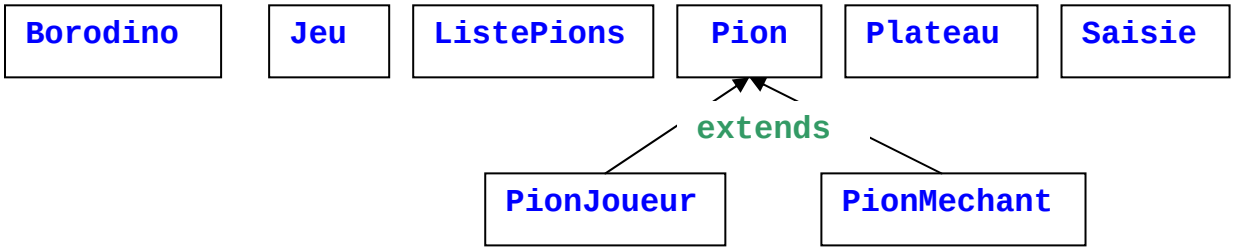
Le déroulement du jeu est strictement conforme au cahier des charges.

La programmation

Les classes (dans l'ordre alphabétique) :

`Borodino` : lance le jeu en créant une nouvelle partie.
`Jeu` : initialise le plateau et les pions, gère les tours de jeu.
`ListePions` : liste globale des pions, et opérations sur cette liste.
`Pion` : attributs propres à un pion : déplacement, combat.
`PionJoueur` : construit un PionJoueur, hérite des attributs de la classe Pion.
`PionMechant` : méthodes spécifiques à un pion simulé (I.A).
`Plateau` : construit un plateau, et l'affiche avec ses pions.
`Saisie` : toute forme de saisie, en particulier le déplacement des joueurs.

Diagramme :



Classe `Borodino`

Méthode :
- `public static void main(String[] args) ;`

L'exécution de la classe Borodino par la machine Java est rendue possible grâce à la présence indispensable de la méthode `main`, dont les attributs sont nécessairement `public` et `static`.
Une nouvelle partie est instanciée et le déroulement du jeu (tours) peut commencer. La partie continue tant que la méthode `fin` de la classe `Jeu` renvoie faux, elle permet de tester après chaque tour de jeu si la partie est terminée.

Classe `Jeu`

Variables :
- `private ListePions lp ;`
- `private Plateau Plat ;`

Constructeur :
- `Jeu(int longueur, int largeur) ;`

Méthodes :

- `public void tour() ;`
- `public boolean fin() ;`

Ses deux variables sont `private` parce qu'ils ne doivent pas être manipulés en dehors de leur classe. Cette liste de tous les pions (vivants) nous permet de gérer les pions sans reparcourir tout le plateau à chaque tour. D'autre part, c'est bien le même objet qui se trouve à la fois sur le plateau et dans la liste.

Le constructeur lance la construction d'un plateau, et la saisie des pions. De là peut commencer une partie.

Les deux méthodes de la classe sont nécessairement `public` pour pouvoir être appelées dans la classe `Borodino`.
La méthode `tour` :
La partie est une séquence de « tours ». Chaque tour est un déplacement de tous les pions de la liste excepté ceux qui sont attaqués sur leur case, dans ce cas, le combat est inévitable, et leur tour de déplacement est sauté. Le tour des pions-joueurs est saisi avant celui des pions-méchants, car il faut être honnête, et donner la position réelle des pions simulés. Le déplacement des pions simulés est généré grâce à la méthode déplacer de leur classe `PionMechant`.
Après les déplacements s'en suit un enchaînement de combats, si tant est qu'ils y en aient. C'est à ce moment que la liste de pions est parcourue, et que tous les pions ayant la même position (x, y) sur le plateau se déplacent. Il peut y avoir plusieurs combats sur une même case, mais un seul gagnant. Les perdants sont alors supprimés du plateau et de la liste.
La méthode `fin` appelle la méthode d'instance `finDePartie` de la classe `ListePions`.

Classe `ListePions`

Variable :

- `private ArrayList lp ;`

Constructeurs :

- `public ListePions() ;`
- `public ListePions(int nbPions) ;`

Méthodes :

- `public void ajoutePion(Pion p) ;`
- `public void supprimePion(Pion p) ;`
- `public int taille() ;`
- `public Pion getPion(int i) ;`
- `public void affiche() ;`
- `public boolean finDePartie() ;`

Pour des raisons de convention et de robustesse du code, la variable d'instance `lp` est `private`.

Il y a une surcharge du constructeur pour pouvoir initialiser une liste de pions avec un nombre de pions par défaut et un nombre de pions donné.

Les méthodes de cette classe sont des méthodes d'opérations basiques sur des `ListePions`. Elles cachent des appels à des fonctions déjà implémentées dans la classe `ArrayList`, en fait les `ListePions` sont des `ArrayList` de pions.
Les méthodes `ajoutePion`, `supprimePion`, `taille`, et `getPion` parlent pour elles.
La méthode `getPion` caste en `Pion` l'`Object` retourné par la fonction `get` de la classe `ArrayList`.
La méthode `affiche` se contente d'afficher les stats de tous les pions de la liste, il s'agit plus précisément d'une légende qui est invoquée à chaque tour de jeu, et dont l'affichage se situe juste en dessous du damier, il va donc de paire avec celui du plateau.
La méthode `finDePartie` est uniquement appelée par la méthode `fin` de la classe `Jeu`.
Et comme les règles du jeu énoncées précédemment l'ont spécifié, une partie se termine s'il n'y a plus aucun joueur, ou s'il ne reste qu'un seul pion-joueur sur tout le plateau, un booléen est alors retourné.

Classe `Pion`

Variables :

- `protected int x ;`
- `protected int y ;`
- `protected int force ;`
- `protected int vitalite ;`
- `protected float heritage ;`
- `protected String nom ;`

Constructeurs :

- `public Pion(Plateau plat) ;`
- `public Pion(Plateau plat, int force, int vitalite) ;`

Méthodes :

- `private void genereHeritage() ;`
- `private void generePosition(Plateau plat) ;`
- `public boolean estHorsPlateau(int x, int y, Plateau plat)`
- `protected void deplacer(int deplacement, Plateau plat) ;`
- `private void frappe(Pion victime) ;`
- `public void combat(Pion attaque) ;`

Les variables d'instance de la classe sont `protected`, c'est un choix qui s'oppose à la convention habituelle, ainsi les variables étaient accessibles dans les sous-classes. Dans le cas contraire (`private`), le nombre d'appels de fonctions aurait été considérable et la syntaxe s'en serait trouvé alourdi. Il nous a donc semblé naturel

qu'un `PionJoueur` ou un `PionMechant` puisse accéder directement à ses caractéristiques intrasèques. En outre, le niveau de protection est acceptable puisque les variables sont visibles uniquement dans les sous-classes.

Les méthodes `genereHeritage` et `generePosition` sont `private` car on ne s'en sert que dans le constructeur de la classe. La première génère un héritage dans l'intervalle [0 ; 25] comme on l'a vu dans les règles du jeu, et la seconde génère une position valide, c'est-à-dire sans pion dans une case mitoyenne à celle générée.
La méthode `estHorsPlateau` est public parce qu'on l'appelle hors de sa classe et de ses sous-classes (par ex. dans la classe `Saisie`). Comme son nom l'indique, elle teste si les coordonnées (x, y) sont hors du plateau, et retourne logiquement un booléen.
La méthode `deplacer` de classe mère est `protected` car elle n'est utile que dans ses sous-classes `PionJoueur` et `PionMechant`. Il s'agit ici du déplacement purement « physique » du pion sur une case mitoyenne.
La méthode `frappe` est `private` parce qu'elle uniquement appelée dans sa classe par la méthode `combat`. D'après les règles, elle soustrait au pion `victime` une part de vitalité correspondant à la force de frappe de l'attaquant, mais n'oublions pas que la puissance du coup obéit à un `random` sur la force maximale de celui qui le porte. La violence des coups portés et des dommages causés sont affichés.
La méthode `combat` est `public` car elle est appelée dans la class `Jeu` une fois les pions tous déplacés. Cette dernière appelle la méthode `frappe` en alternant le frappeur et le frappé jusqu'à ce que l'un des deux meurt. Celui qui a engagé le combat a été choisi au hasard pour ne pas avantager un pion au détriment d'un autre. Une fois le combat terminé, le pion reconstitue sa vie et hérite d'une partie des points de vie et de force de son adversaire, ces points sont proportionnels à l'héritage du pion vaincu. Une attente de trois secondes est prévue pour visualiser les détails du combat.

Classe `PionJoueur extends Pion`

Variable :
- `private static int numero ;`

Constructeur :
- `PionJoueur(Plateau plat) ;`

Méthode :
- `public void deplacer(int deplacement, Plateau plat) ;`

Dans cette sous-classe sont récupérés tous les attributs de la classe mère `Pion` (variables et méthodes), et seront accessibles seuls ceux qui ne seront pas `private`.

La variable de classe est `private` parce qu'elle ne doit pas être manipulée en dehors de sa classe, d'autre part elle est `static` car on peut avoir besoin de réutiliser cette variable à un autre moment du programme. Le numéro qu'elle représente est simplement celui du joueur, en particulier lors de l'affichage de la

saisie de son nom. On aurait pu le passer en paramètre dans le constructeur, mais en le laissant `static`, on pensait garder une ouverture sur un développement plus poussé du jeu où d'autres joueurs pourraient entrer en cours de partie.

Le constructeur appelle celui de la super-classe (`Pion`). Puis le champ `nom` hérité de la classe mère est seulement initialisé ici ; il est directement saisi dans ce constructeur, ce qui explique la gestion d'erreur d'entrée / sortie.

La méthode de déplacement est `public` car elle est appelée depuis la class `Jeu` à chaque tour et pour chaque joueur, elle appelle la méthode de la super-classe (`Pion`).

Classe `PionMechant extends Pion`

Variable :

- `private static int numero ;`

Constructeur :

- `PionMechant(Plateau plat) ;`

Méthodes :

- `private boolean deplacementPossible(ArrayList positionSuivante, ArrayList cibles, Pion faible) ;`
- `public void deplacer(Plateau plat) ;`

Dans cette sous-classe sont récupérés tous les attributs de la classe mère `Pion` (variables et méthodes), et seront accessibles seuls ceux qui ne seront pas `private`.

La variable de classe est `private` et `static` pour les mêmes raisons que dans la classe `PionJoueur`. Cependant, l'itérateur sert cette fois-ci à construire le nom d'un pion-méchant et non plus à le saisir : Chaque robot possède un numéro à la fin de son nom, ce numéro est affiché dans le plateau et permet de les différencier.

Le constructeur appelle celui de la super-classe (`Pion`). Là aussi le champ `nom` hérité de la classe mère est initialisé dans le constructeur de la sous-classe.

La méthode `deplacementPossible` est `private` parce qu'elle est uniquement appelée dans sa classe par la méthode `deplacer`. Cette dernière teste si un pion en poursuivant un autre a le droit de se déplacer sur la case suivante : Il s'y déplace si la case suivante est vide, soit si elle contient nécessairement le pion cible et au pire d'autres pions plus faibles que lui.
La méthode de `deplacer` est `public` car elle est appelée depuis la class `Jeu` à chaque tour et pour chaque robot. Une fois que la stratégie à appliquer est définie, soit l'attaque soit la fuite, elle appelle la méthode `deplacer` de la super-classe (`Pion`) pour procéder au déplacement réel du pion.
Cette méthode constitue le cœur de l'intelligence artificielle du jeu, elle est représentative des stratégies décidées et de la façon dont le programme a été

pensé. Nous n'en parlerons pas plus ici étant donné qu'elle fait l'objet d'une étude exhaustive au cours de la partie **stratégie** du rapport.

Classe `Plateau`

Variables :

- `private int longueur ;`
- `private int largeur ;`
- `private static ArrayList [] [] plateau ;`

Constructeur :

- `public Plateau(int longueur, int largeur) ;`

Méthodes :

- `public int getLongueur() ;`
- `public int getLargeur() ;`
- `public static ArrayList [] [] getPlateau() ;`
- `public void affiche() ;`

Ses trois attributs sont `private` parce qu'on ne souhaite pas manipuler ces variables d'instance directement dans une autre classe, et ceci pour des raisons de robustesse. Entre autres, cela fait partie des conventions du style de programmation orientée objet même si ce choix n'est pas indispensable ici.

Le plateau est `static` car le programme s'appuie sur un et un seul plateau.

Le constructeur prend en argument la longueur et la largeur qui auront été saisi.
On obtient un tableau de tableau d'`ArrayList`.

Les trois premières méthodes `public` sont dites spécialisées, elles existent uniquement parce que les variables d'instance sont `private`, elles permettent d'accéder en lecture aux champs de la classe à partir d'une autre classe, d'où `public`.
La méthode `affiche` est un affichage du plateau sous forme de damier (simple grille de cases) et de ses pions sur la console.
Elle est appelée après chaque tour de jeu.
Les pions-joueurs sont représentés par la première lettre du nom entré en début de partie. Les pions-méchants sont représentés par un entier dans l'intervalle [1 ; 99]. C'est donc l'affichage qui limite le nombre de pions sur le plateau, et pas l'architecture du programme, qui, elle, le permet. D'autre part, il nous faut un tableau de 23 x 23 pour avoir un maximum de 95 pions-méchants, donc on ne se sent pas frustré par la taille du tableau. Mais n'y a-t-il pas 529 cases sur un tableau de 23 x 23 ? Oui, c'est vrai mais pour que les départs soient à peu près équitables, on construit un pion s'il n'y a personne sur une case juxtaposée, on comprend aisément pourquoi, et cela diminue de fait le nombre de pions sur le plateau.

Classe `Saisie`

Variable :

- `private static ArrayList directions ;`

Méthodes :

- `public static int dimension(String cote) throws IOException ;`
- `public static void pions(Plateau plat, ListePions lp) throws IOException ;`
- `public static int deplacement(PionJoueur pion, Plateau plat) throws IOException ;`

La variable de classe *directions* est `private` car nécessaire pendant toute la durée du programme, en effet elle contient toutes les directions possibles où puisse se déplacer un joueur. Ceci évite de réinstancier à chaque tour et pour chaque joueur un tableau de type `ArrayList`.

La méthode de classe `dimension` saisit soit la longueur soit la largeur d'un plateau. Les dimensions minimums sont 4 x 4 à cause du calcul du nombre de pions maximal.
La méthode de classe `pions` saisit à la fois le nombre de joueurs, leurs noms respectifs, et le nombre de robots.
Il doit y avoir au moins 1 joueur pour démarrer une partie, et maximum 99 robots sur un plateau suffisamment grand pour les contenir. Cette méthode est uniquement appelée en début de partie, dans le constructeur `Jeu`, pour initialiser la partie.
La méthode de classe `deplacement` saisit le déplacement d'un joueur.
La direction 5 (rester sur la même case) est toujours incluse, et les autres directions sont ajoutées au fur et à mesure qu'elles sont évaluées dans la méthode.
Parallèlement aux ajouts de directions, on construit une chaîne de caractères récapitulant toutes les directions possibles. Celle-ci est affichée lors de la saisie du déplacement, et l'entier saisi est ensuite retourné vers la méthode `deplacer` de la classe `PionJoueur`.