Jeton Fou

1) Elaboration

- Départ sous forme de quatre classe (Jeton,Damier,JetonFou,Direction) mais Direction pour peu d'argument et utilité retirer
- Partit sous forme complexe du problème (multitude de variable) ainsi de gérer la réservation des cases. Modèle trop complexe --> Simplification.

2) Modèle finale

UML :

<Jeton>
chaine caractere / entier jeton / entier dirx, diry / booleen depla
Jeton() / Jeton (Random rnd) / Jeton (entier)
getCaractere() / getDirx() / getDiry() / getDepla() / setDepla() / change() / directionJeton() / directionJetonV2() / detruit() / deviens() / placeJeton() / placeJetonV2()

<Damier>
Tableau de Jeton [ ][ ] / entier nbrJeton / entier N, M
Damier()
getNbrJeton() / demandeJeton() / demandeTaille() / deplacementJeton() / deplacementJetonV2() / initTab() / resetDepla() / toString(entier)

<JetonFou>
JetonFou()
actualisationGraphique(FenetreGraphique, Tableau de Jeton[ ][ ]) / Version1() / Version2() / Version3() / main()

3) Comment fonctionne le programme

La classe JetonFou possède la méthode principale c'est elle qui lance et ajuste les paramètres. L'utilisation d'un "tant que"(l.129) permet de boucler sur les versions du programmes qui seront ensuite choisit dans un "switch"(l.138). Ce switch permet de lancer une méthode ou de quitter afin de choisir le cas le plus approprier.

--> Version1()

La Version1 comme la Version2 et 3 crée un nouvelle instance de Damier et gère les paramètres d'initialisation. Puis boucle afin de crée le nombre de Jeton nécéssaire. Avant de boucler sur le deplacement et l'affichage des Jetons.

--> Version2()

Identique a la Version1() sauf qu'on l'on rajoute une graine, on remplace les Math.random() par des nextInt(a) *(permet de choisir sur une série de bits un entier inférieur a "a")*.

--> Version3()

Même chose que la Version1() sauf que la méthode toString() n'est pas utilisée, on utilise un affichage graphique. Pour cela on décompose le code deux grosse partie. La partie initialisation et création des Jetons et la partie affichage, avec la création est l'actualisation de la fenetre graphique.

4) Quels sont les défauts du programme.

Les Versions sont facilement optimisable dans la mesure ou elle ne comporte qu'une suite de méthode. Pour optimiser le programme il suffit de modifier le code dans les méthodes. Les Version1 et Version2, sont difficilement optimisable été donnée que le travail le plus pour l'algorithme et de parcourir le tableau 2 fois pour 1 déplacement. On pourrait régler le problème par un tableau de coordonée de Jeton qui nous donne directement les coordonées du Jeton mais cela se fait au détriment de l'espace mémoire, or dans notre cas le tableau reste assez petit ce qui n'est pas dérangant. La Version qui manque le plus d'optimisation est la Version3, l'interface graphique étant lourde, nous ne pouvons pas nous permettre de commetre certaine erreur. Ici l'interface graphique s'actualise à chaque "frame" alors que le Damier et les lignes de celle-ci sont fixe, on perd donc des performances a actualiser quelque chose de fixe. Il faudrait donc pouvoir ne pas actualiser cette partie de l'interface.

5) Question/Réponse

- Pourquoi avoir utilisé des méthodes Getter/Setter alors qu'un attribut publique aurait été mieux ?

  --> Cela nous permet si on améliore le code (optimisation/correction de bug) de modifier uniquement les méthodes de Getter/Setter afin de limiter l'utilisation abusive.

- Pourquoi autant de méthode ?

  --> Que ce soit pour la correction de bug ou de travaille en équipe il était plus simple et plus intelligent de séparer le travail en petit algorithme afin de pouvoir mieux gérer le travail. Certains sous-algorithme on ainsi fusioner (exemple detruit() et change() dans la classe Jeton).