Puissance 4 : Introduction et conception

par Baptiste Wicht

Date de publication : XXX

Dernière mise à jour :

Avec cet article nous allons commencer à concevoir l'application Puissance 4 en Java.

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts

I - Introduction

Cet article est le premier d'une série de plusieurs articles pendant lesquels on va développer entièrement le jeu Puissance 4 en Java. Nous allons commencer dans cette partie à concevoir le Puissance 4 et à définir les classes dont nous aurons besoin pour travailler.

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts

II - Le puissance 4

Le Puissance 4 est un jeu simple qui se joue à deux joueurs dont le but est d'aligner 4 pions de notre couleur. Le plan de jeu est une grille de 6 lignes et de 7 colonnes dans lesquelles les joueurs insérent chacun leur tour des pions. La partie est terminée quand un joueur a aligné 4 pions de sa couleur ou que la grille est pleine.

Si vous voulez plus d'informations sur le Puissance 4, je vous invite à consulter la page de Wikipedia.

II-A - Notre puissance 4

La première version de notre puissance 4 sera très basique et s'étoffera au fur et à mesure de l'avancement des articles.

Dans cette version, nous pourrons jouer à un contre un dans une grille normale affichée dans une fenêtre graphique. La partie devra se terminer dès qu'un joueur aura gagné ou dès le remplissage complet de la grille.

Pour le moment, ce seront les seules exigences du programme.

II-B - Conception

Nous allons maintenant commencer à concevoir notre application.

II-B-1 - Définition des objets

On va d'abord commencer par définir quels sont les beans dont nous avons besoin. Un bean est tout simplement un objet Java composé d'accesseurs et de setteurs et avec pas ou peu de code métier.

Pour définir quels sont les différents beans à utiliser, il faut réfléchir aux différentes choses qui interviennent dans la partie. Tout d'abord nous avons une personne qui va jouer au Puissance 4. Nous aurons même 2 joueurs par partie, il nous faut donc employer une classe *Joueur* dont l'on va définir les propriétés plus tard. Ensuite, les joueurs utilisent des pions. Ceux-ci seront représentés par des objets de la classe *Pion*. Les pions vont aller se poser dans une grille, nous allons utiliser un type *Grille* pour représenter en mémoire la grille. De plus, on accepte que quelqu'un joue à une partie, on aura donc une classe *Partie* qui va représenter une partie en cours.

Ensuite, nous aurons une classe d'entrée dans le programme, cette classe ne s'occuppera que du lancement du programme, il est néanmoins préférable de séparer la méthode main des autres classes. Nous aurons donc une classe *Puissance4*. Il nous faudra également une classe qui va permettre de gérer le tout, cette classe sera la classe *Jeu*. C'est cette classe qui va contenir l'intelligence du jeu, nous en verrons aussi les caractéristiques plus tard.

Ensuite, pour ce qui est des classes d'interface graphique, nous les définirons plus tard, avec l'article dans lequel nous developperons l'interface.

II-B-2 - La classe Joueur

Tout d'abord, il faut réfléchir si nous aurons un seul type de joueur ou bien deux. Et quand on se pose des questions pareilles, il faut aussi voir dans l'avenir. Dans notre cas, pour le moment, on ne va rencontrer que des joueurs humains. Mais si dans l'avenir on ajoute une IA, il se pourrait qu'elle ait un comportement différent de celui d'un joueur humain.

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts

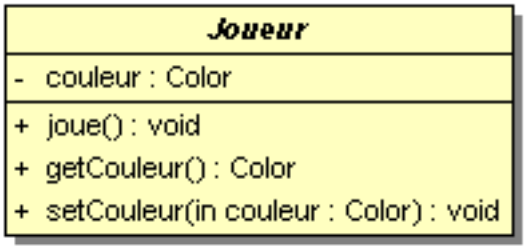
Dans des cas comme celui-ci, il faut donc généraliser nos objets *Joueur*. Pour faire cela, on a deux choix, soit une interface soit une classe abstraite. On va commencer par réfléchir aux propriétés d'un joueur avant de décider de l'implémentation. Un joueur a seulement une couleur qui va définir celle de ces pions. Ensuite, tout son comportement sera défini dans une seule méthode, appellons-la *joue()*.

Maintenant que nous avons toutes ces infos, considérons à nouveau nos deux possibilités :

- Une interface : cela implique de rédefinir toutes les méthodes
- Une classe abstraite : permet d'avoir de l'abstrait et du concret

En considérant ceci, nous voyons qu'il nous faut employer une classe abstraite, car la gestion de la couleur sera exactement la même d'un type de joueur à l'autre quel que soit son comportement. Nous aurons donc une méthode abstraite *joue()* et 2 méthodes concrètes *setCouleur()* et *getCouleur()*.

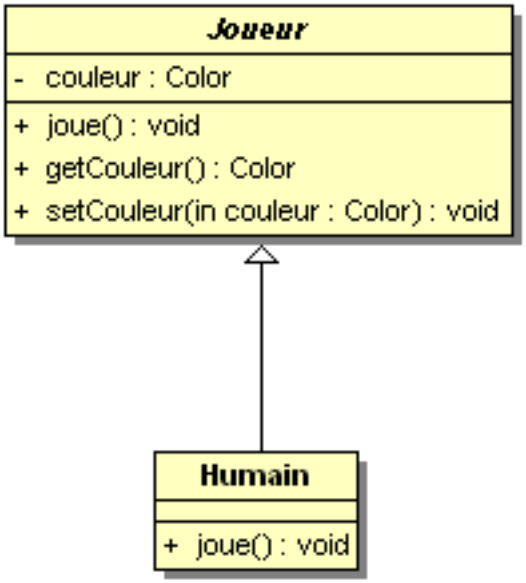
Nous allons donc définir une classe abstraite *Joueur* :



Classe Joueur

Et nous créerons ensuite une classe *Humain* qui étendra *Joueur* et qui redéfinira le comportement de la méthode *joue()*. Il n'aura pas par contre à redéfinir les autres méthodes puisqu'elles sont déja complétement implémentés dans la classe abstraite. Voici donc ce que va donner notre *Humain* :

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts



Classe Humain

II-B-3 - La classe Pion

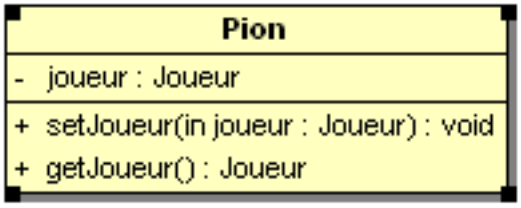
Cette fois-ci, nous n'aurons qu'un seul type de pion. Nous n'avons pas besoin de généraliser les pions, nous n'aurons donc qu'une seule classe *Pion* héritant directement d'*Object*.

Quels sont les caractéristiques d'un pion, vous me direz qu'il a une couleur, mais il appartient à un joueur qui lui décide de la couleur. Un pion a donc un attribut *joueur* par lequel nous accéderons ensuite à la couleur.

Et c'est déja tout ce qui caractérise un pion. On ne stocke pas la position du pion directement dans le pion, c'est à la grille de s'occupper de la gestion des pions et donc de leur position.

Voici donc le diagramme de la classe *Pion* :

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts



Classe Pion

II-B-4 - La classe Grille

Là encore, nous n'aurons qu'une sorte de grille, donc une seule classe *Grille* étendant directement *Object*.

Il nous faut maintenant définir la manière avec laquelle nous allons représenter la grille en mémoire. La grille en elle-même est un ensemble de données fini à deux dimensions, donc nous allons utiliser simplement un tableau à deux dimensions. Ce tableau contiendra soit un *Pion* soit une valeur nulle, la valeur nulle permettra d'indiquer que la case est vide.

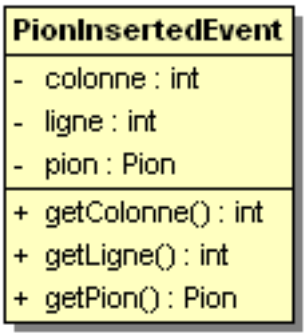
Maintenant que nous avons défini le stockage, on va ajouter des méthodes pour récupérer et modifier ces données. On aura donc une méthode *getPionAt()* et une méthode *setPionAt()* qui prendront toutes deux en paramètres le numéro de la colonne et de la ligne concerné.

On va aussi ajouter maintenant une méthode qui va permettre de vider le tableau que l'on va appeller vider().

Maintenant que va faire d'autre la grille ? Ce n'est pas à elle de contrôler s'il y a un coup gagnant, ce sera le travail de la partie, mais il faut pouvoir contrôler certaines choses. Tout d'abord, il nous faut savoir si une colonne est pleine ou si elle possède encore de la place pour d'autres pions, cela sera contrôlé par une méthode *isColonnePleine()*. Ensuite, il nous faudrait aussi pouvoir savoir la dernière position de libre pour y placer le pion. Ceci sera fait via une méthode que l'on appelera *getDernierePositionLibre()* et que l'on utilisera sur une colonne spécifique. Il nous faudra aussi savoir si la grille est pleine ou non, pour savoir si la partie est terminé ou pas, cette information nous sera donnée par méthode *isGrillePleine()*.

Une dernière chose importante est maintenant que l'interface graphique soit toujours à jour vis à vis de la grille. On va donc appliquer le principe du pattern Observer. On va utiliser pour cela les listeners et les événements sur notre objet *Grille*, à chaque fois qu'un pion sera inséré, on va lancer un événement sur tous nos écouteurs, c'est donc ensuite l'interface graphique qui va les intercepter et les mettre à jour. On aura donc une méthode pour ajouter un listener, une méthode pour en supprimmer et une méthode pour lancer un événement. On va donc aussi directement créer une nouvelle classe pour l'événement, *PionInsertedEvent*. Cette classe contiendra juste la position du pion et le pion qui a été inséré :

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts



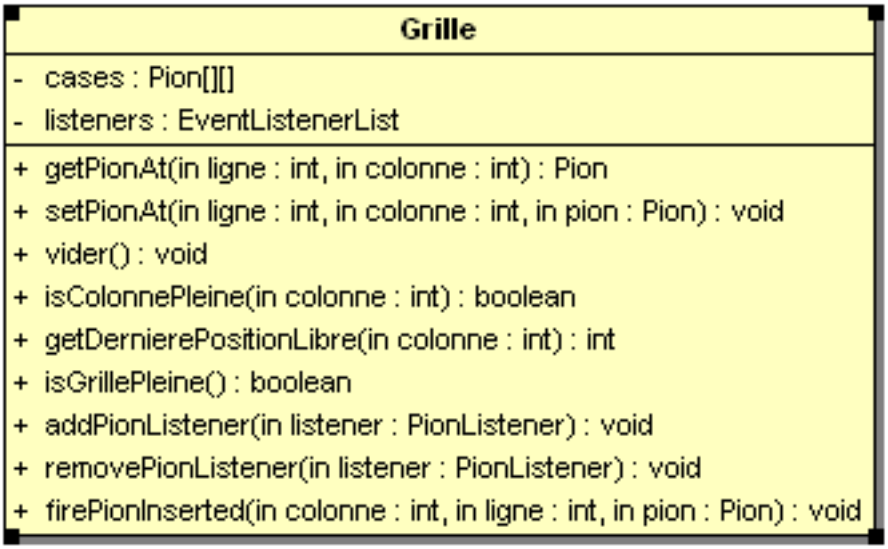
Classe PionInsertedEvent

Ce qui implique aussi de créer un listener. On va l'appeller *PionListener*. Un listener est une interface qui contiendra les différentes méthodes que l'on pourra appeller, dans notre case *pionInserted()* :

Interface PionListener

Voici donc les caractéristiques finales de notre classe *Grille* :

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts



Classe Grille

II-B-5 - La classe Partie

Cette classe représente une partie de Puissance 4, nous n'en aurons donc qu'un seul type.

Quelles sont les caractéristiques d'une partie ? Elle à tout d'abord des joueurs et une indication permettant de savoir quel est le joueur courant. Ensuite, une partie posséde bien entendu une grille de jeu. Ensuite, une partie posséde un état, c'est à dire si elle est en cours ou si elle est terminée. On aura donc une méthode *isTerminee()* pour savoir si la partie est terminée ou non.

Ensuite, on aura une méthode qui permettra de changer de joueur courant, *prochainJoueur()* et une autre méthode nous permettant de récupérer le joueur courant, *getJoueurCourant()*. Ensuite, on aura la méthode principale, que l'on va aussi appeller *joue(int column)*, elle s'occuppera de faire jouer les joueurs et de calculer ce qu'il faut faire.

Voici donc à quoi va ressembler finalement notre classe :

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts

Classe Partie

II-B-6 - La classe Jeu

La classe *Jeu* est la classe permettant de gérer les principales phases du Puissance 4. comme pour le moment, on ne sait pas encore s'il y aura plusieurs types de partie différentes, on ne va pas prendre le risque et on va généraliser notre classe *Jeu*, on aura donc une interface *Jeu* et une implémentation concrète *JeuNormal*. De plus, cette classe se doit d'être unique, nous utiliserons donc le pattern singleton pour empêcher qu'il y ait plusieurs instances de cette classe. C'est aussi elle qui va s'occupper de faire le lien entre l'interface graphique et la partie de puissance 4.

La classe *Jeu* contiendra un objet *Partie* qui sera tout d'abord à null pour indiquer qu'aucune partie n'est encore démarrée. Nous aurons ensuite une méthode permettant de démarrer une nouvelle partie. Appellons-la *nouvellePartie()*.

Comme c'est aussi cette classe qui va se charger de faire l'interface entre le GUI (Graphic User Interface) et le jeu, la classe va donc implémenter l'interface *MouseListener* et redéfinir la méthode *mouseClicked()* de chaque bouton et c'est là-dedans qu'elle va récupérer les clics sur le bouton et agir en conséquence de chacun d'eux.

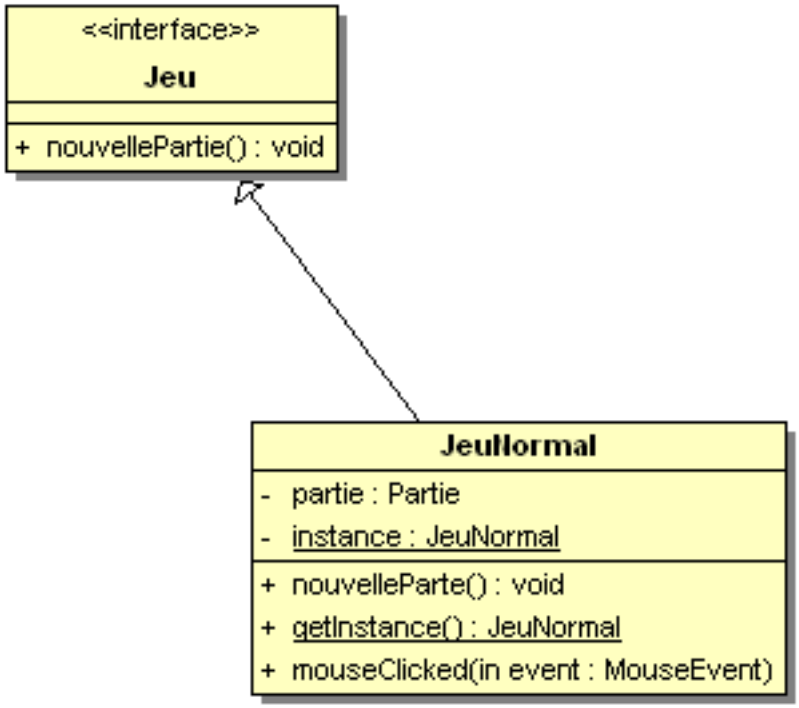
Voilà donc à quoi va ressembler notre interface :

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts

Interface Jeu

Le singleton est un design pattern, permettant d'avoir une seule instance d'une classe dans toute l'application. Pour faire cela, on garde une instance statique de notre classe et on ajoute une méthode getInstance qui va initialiser et récupérer notre instance. Il faut aussi déclarer le constructeur privé pour qu'aucune autre classe puisse instancier un nouvel objet de cette classe.

Voici donc à quoi ressemblera notre classe concrète au final :

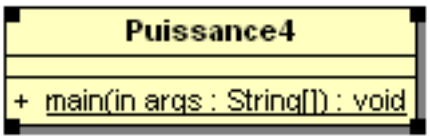


Classe JeuNormal

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts

II-B-7 - La classe Puissance4

Cette classe est très simple, elle ne possédera qu'une seule méthode, la méthode *main* qui va s'occupper du lancement de l'interface graphique.



Classe Puissance4

II-B-8 - Autres classes

En plus des classes déja définies précédemment, nous aurons encore des classes d'utilités qui nous permettront de gérer des objets spécifiques tels que par exemple les images. Nous ne définirons pas ces classes ici, elles vont venir s'ajouter petit à petit au programme au fur et à mesure de son avancement.

Une autre classe que nous utiliserons sera une classe utilisée pour gérer les valeurs constantes de l'application. Appellons-la *Constantes* (original n'est-ce pas ?).

II-C - Conclusion

Nous avons maintenant posé les bases de notre programme. Les prochains articles seront donc consacrés à developper ce que nous avons conçu.

Ce qu'il faut retenir de cet article est qu'il ne faut surtout pas délaisser la partie conception pour aller directement dans le code. Une bonne conception n'est jamais du temps perdu, elle vous permettra d'obtenir ensuite du code stable, réutilisable et facilement extensible. Donc bien que ce ne soit pas la partie la plus intéressante, n'hésitez pas à consacrer du temps à cette partie, vous en récolterez les fruits plus tard.

Dans le prochain article, nous allons construire l'interface graphique du programme. Sachez déja que cette interface graphique se fera en Swing.

II-C-1 - Téléchargements

...

II-C-2 - Remerciements

Je tiens à remercier fearyourself pour ses remarques sur cet article.

Copyright © 2007 Baptiste Wicht . Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts