université PARIS DIDEROT PARIS 7

Rapport de projet : POO-IG

Akash HOSSAIN, groupe MI2

Introduction

Dans ce rapport, je vais d'abord donner un résumé de ma démarche de travail, dans quel ordre j'ai abordé les parties du projet, quels modifications j'ai dû effectuer... Ensuite, j'expliquerai certains de mes choix (au niveau du code) qui peuvent être discutables. Pour finir je parlerai des problèmes que j'ai rencontré. Ce rapport fait de nombreuses fois référence au code du projet, je recommande donc de lire mes sources avant de lire le rapport.

1 Démarche de travail

Lorsque j'ai reçu le sujet du projet, il y avait certaines parties que j'étais déjà capable d'aborder, et d'autre (comme le swing) qui nécessitaient des cours que je n'avais pas encore reçu. Je me suis immédiatement occupé des parties abordables, ce qui m'a donné une avance solide. De cette manière j'ai codé les classes Alignement, Alignable, Colore, Case, et Plateau (avec des noms différents qui correspondent à la modélisation soumise début Décembre). Mon idée de modélisation n'était alors pas très claire (car le cours n'était pas terminé), mais je savais qu'elle allait inclure ces classes.

J'ai pu ensuite affiner ma modélisation en lisant le TP9, qui utilise une architecture MVC. J'ai ensuite codé le KolorLines selon la modélisation soumise début Décembre. Après debugging et avant de coder le Gomoku, je me suis rendu compte que je pouvais factoriser plus de code, ce qui m'a amené à écrire des classes abstraites intermédiaires. A partir de là l'implémentation du Gomoku, du KolorLines amélioré et du robot stupide on été très rapides car la modélisation était bien optimisée.

J'aurais sans doute dû mieux réfléchir à ma modélisation dès le départ, mais la POO est un peu nouvelle pour moi et ce n'était pas évident. En réalité, la création des classes intermédiaires ne m'a pas pris beaucoup de temps, la partie la plus longue était le debugging des méthodes relatives au swing, sur lequel je n'était pas très à l'aise (c'est tout nouveau pour moi). En plus, contrairement aux interfaces textuelles, un programme qui utilise le swing ne prends pas fin quand on rencontre un bug, mais continue de tourner, et on se rend compte trop tard qu'une erreur est survenue.

2 Quelques choix à discuter

2.1 Les classes Alignable et Case

L'examinateur aura sans doute été surpris par la redondance des méthodes de l'interface Alignable, et des attributs de la classe Case (les 8 voisins). En réfléchissant à comment implémenter la méthode alignements dans Alignable, je n'ai trouvé que deux manières de le faire. La première manière est celle que j'ai choisi, en utilisant une multitude de petites méthodes récursives. La seconde était de trouver les alignements directement dans le tableau à deux dimensions avec des astuces mathématiques, et d'écrire tout le code directement dans alignements, ce qui donnera un bloc de code très gros et très compliqué.

La redondance de la solution que j'ai choisi est certes gênante, mais au moins les méthodes sont relativement simples et claires. L'initialisation des attributs de Case avec la méthode voisins dans Plateau peut paraître longue, mais elle n'est exécutée qu'une fois au début du programme et n'est plus jamais appelée par la suite. De plus ce choix obéit bien au credo du programmeur : décomposer un problème complexe en une multitude de problèmes triviaux. J'en ai discuté avec mon professeur de TD qui soutient mon choix.

2.2 La classe Alignement

J'aurais préféré, dans la classe Alignement, implémenter coupe avec les méthodes mise en commentaires à la fin du programme. L'avantage d'une telle implémentation est que la complexité asymptotique de coupe aurait été constante, tandis qu'avec la méthode de parcours que j'utilise, le temps de calcul dépend de la longueur des alignements étudiés.

J'aurais sans doute pu debugger le code en commentaire, mais je me suis rendu compte que ça m'aurait fait perdre beaucoup de temps, pour un gain en temps de calcul qui n'est pas vraiment nécessaire dans ce projet. Même si elle est moins efficace, la méthode que j'ai écrit à la place a l'avantage d'être bien plus simple à comprendre.

2.3 La classe Pair

Au lieu d'implémenter moi même une classe Pair, j'aurais pu simplement importer le package Point qui fait exactement la même chose. En fait, j'avais écrit la classe Pair longtemps avant, pour un TP d'un semestre antérieur, et depuis je me ressers assez souvent de cette classe pour divers usages. C'est donc plus facile pour moi d'utiliser cette classe que de devoir lire la doc.

2.4 Le robot stupide

Je tiens d'abord à dire que je ne comprends pas vraiment l'intérêt d'implémenter un joueur robot pour des jeux à un joueur comme KolorLines. J'ai réfléchi à cette consigne et je me suis dit que si j'implémentais un joueur ce serait un "joueur" au sens propre, qui n'agit pas directement sur le modèle mais clique virtuellement sur les boutons de la vue.

J'ai réussi à programmer ce joueur en n'effectuant qu'un changement : j'ai dû ajouter un attribut booléen à VueKL pour savoir quand la partie est finie, et que le robot doit arrêter de jouer.

Pour le Gomoku(qui se joue à deux), la présence d'un joueur non-humain, elle, est justifiée, même si il ne s'agit pas d'une IA. J'ai par contre préféré programmer un robot qui agit directement sur le modèle pour la simple raison que c'était plus facile et plus court.

2.5 La classe Plateau

Dans Plateau, j'ai choisi de créer une liste des coordonnées de toutes les cases vides. L'avantage de ce choix est qu'on peut très facilement choisir une case vide au hasard pour faire jouer l'ordinateur (pas le robot stupide) dans KolorLines. J'ai fais ce choix en m'inspirant de l'exercice 1 du TD10 (sur les Bazar).

3 Problèmes connus

Tout d'abord, je tiens à dire que le projet semble fonctionner correctement. Je n'ai rencontré aucun bug après de nombreux tests. Cependant, il y a quelques problèmes génants dans le code ou l'architecture, dont je vais parler ici.

3.1 Le downcasting

Dans les méthodes de ControleurKL, ControleurKLPlus et ControleurGomoku, on est souvent forcés à faire du downcasting. Je n'ai pas de problèmes quand je dois effectuer un cast deux ou trois fois, mais quand je dois le faire un grand nombre de fois en peu de temps, ça me gène et j'estime qu'il y a un problème au niveau de l'architecture. La priorité à été donnée à la factorisation du code, comme l'exige le sujet. Mais si ça ne tenait qu'à moi, j'aurais sans doute codé de manière différente pour éviter les casts.

3.2 Un constructeur inutile

Pour pouvoir écrire le constructeur de KolorLinesPlus et ModeleKLPlus qui étendent KolorLines et ModeleKL, j'ai été forcé d'écrire un constructeur vide pour les classes KolorLines et ModeleKL. Ce qui est génant, c'est que ces constructeurs ne sont pas destinés à être appelés et servent juste à implémenter KolorLinesPlus. J'ai tenté diverses choses pour éviter cela.

J'ai d'abord tenté d'appeler le super constructeur avec comme argument un entier quelconque. Le problème, c'est que le code qui suivait n'était pas exécuté, et je me retrouvais avec un KolorLines et non un KolorLinesPlus. J'ai aussi tenté de créer un constructeur sans argument de KolorLines qui lance une exception

que j'attrape dans le constructeur de KolorLinesPlus. Cela n'a pas fonctionné car la ligne de code "try" rendait impossible un appel du super constructeur.

Au final, j'ai été obligé d'écrire un constructeur vide, inutile et assez disgracieux. Ceci étant dit, le fait que KolorLinesPlus hérite de KolorLines permet de factoriser beaucoup de code.

3.3 Des instructions parfois redondantes

Dans KolorLines, il y a une petite imperfection qui me gène un peu. Quand l'utilisateur déplace une case, le contrôleur appelle d'abord sa méthode joue, qui elle même appelle update, qui appelle setColor dans les CaseVue, ce qui change l'état cliquable de ces cases. Or, immédiatement après, il appelle setEnabled qui change aussi l'état cliquable des cases. Ce sont donc des instructions redondantes.

Pour régler ce problème, j'ai d'abord essayé de supprimer le code de CaseVue.setColor qui change l'état cliquable du bouton, mais cela pose des problèmes quand le bouton est initialisé, car au début du jeu les cases vides sont alors cliquables. J'ai ensuite essayé de n'appeler Vue.setEnabled que quand une case venait d'être sélectionnée. Cela pose aussi un problème car quand l'utilisateur annule sa sélection, update n'est pas appelée car l'ordre de déplacement est invalide, et les cases vides restent cliquables.

Au final je n'ai pas trouvé de moyen de régler ce problème. Cela n'affecte pas le fonctionnement du programme, c'est juste qu'il n'est pas optimisé comme je le voulais.

Conclusion

Pour résumer ce qui a été dit, mon projet fonctionne correctement, et, même si j'ai parfois dû faire des compromis au niveau du code, il est assez bien optimisé et j'en suis globalement satisfait. Nous verrons si il en est de même pour les examinateurs.