TP 6+7 (~projet)

Objectifs

Nous avons abordé, au cours du module de MDI, différents aspects du développement logiciel, principalement à l'aide techniques de modélisation ou programmation orienté objet (e.g., UML, design patterns).
Au cours des deux dernières séances[1], nous revisitons ces techniques de programmation par la pratique, avec un cas concret et une série d'exercices.
L'objectif est de démontrer que vous êtes en mesure :

1. D'identifier et expliquer des design patterns dans un code existant ;
2. De choisir et combiner des design patterns en fonction d'un problème de conception donné ;
3. D'implémenter des design patterns en Java ;
4. De réorganiser du code (« refactoring ») et de le tester.
5. De comprendre une spécification et de la clarifier si besoin

Cette série d'exercice permettra également d'appréhender quelques technologies (Swing, Java properties, Logger, JUnit), et de lire, comprendre, adapter du code existant.
Il s'agit enfin d'avoir un esprit critique sur un code existant.

Modalités

Le travail est à réaliser en binôme (ou seul).
La date de rendu est fixée au 10 mai 23h59.

Il est demandé de rendre :

- Le code source de l'application, incluant les instructions pour installer/exécuter l'application et les cas de test ;
- Un rapport comprenant les réponses aux différents exercices

Le code source peut être rendu via une archive zip ou tar, mais il est également possible de le rendre via un dépôt git ou SVN.
Le rendu s'effectue par mail : mathieu.acher@irisa.fr

[1] Il est fortement conseillé de continuer le travail en dehors des deux (dernières) séances de TP.

Contexte

On souhaite modifier (par restructuration et ajout de fonctionnalités) un programme d'échecs existant.
Vous trouverez une archive à l'adresse suivante :
http://mathieuacher.com/teaching/MDI/ChessMDI1516.zip

Pour ne rien vous cacher, ce programme a été développé par une tierce personne (sur sourceforge).
Il a été légèrement adapté (e.g., avec la suppression de certaines fonctionnalités).

Le code développé présente de nombreux défauts de conception – nous allons nous en rendre compte au cours de la série d'exercices – mais il a le mérite de fournir une implémentation du jeu d'échecs qui fonctionne relativement bien.
Ainsi, on peut jouer contre soi même ou contre un adversaire (modulo le fait qu'il utilise la même machine et la même interface – les aspects réseaux ont été supprimés).
Cela fait partie intégrante de l'exercice de comprendre, corriger, et critiquer le code existant.

La page http://en.wikipedia.org/wiki/Chess est une bonne introduction aux règles du jeu d'échecs et est normalement suffisante pour la série d'exercices à venir.

Vous pouvez utiliser Eclipse ou IntelliJ pour exécuter le projet et développer.

Conseils

Il n'y a pas d'ordre strict dans les exercices, mais il est conseillé de commencer par les exercices 1, 2, et 3 pour bien appréhender le projet.
L'exercice 1 peut se dérouler en plusieurs étapes : un premier jet, un raffinement suite à l'exercice 2, un nouveau raffinement suite à l'exercice 3, et ainsi de suite.

Il est également conseillé de lire l'énoncé de tous les exercices avant de commencer le travail.

Exercice 1. Au commencement

Comprendre et modéliser

- Produire le diagramme de classes UML de l'application à l'origine ;
- Identifier les différents design patterns utilisés. Rappeler la définition du design pattern et justifier (ou critiquer) l'usage du design pattern;
- Expliquer le rôle des entités (e.g., classes, méthodes, attributs) de l'application pour chaque design pattern.

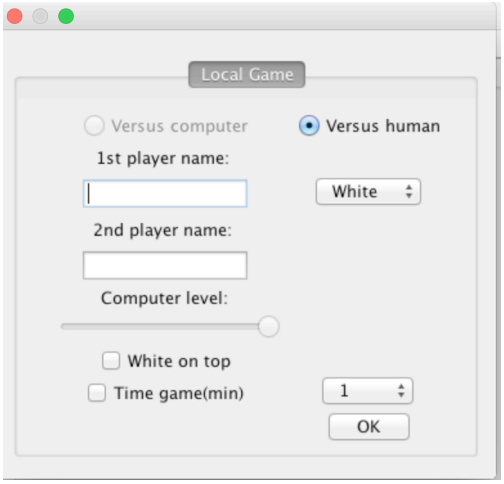
Exercice 2. Nettoyage

Refactoring

Il est très frustrant, lorsqu'on lance l'application, de ne pas avoir un jeu d'échecs (alors que c'est l'essence même de l'application).

Il faut nécessairement passer par un menu pour démarrer une partie (« File -> New »).
Il faut de plus rentrer des informations.
L'expérience utilisateur étant jugé très mauvaise il est demandé :

- De lancer une partie directement au chargement de l'application (le 1er joueur aura pour nom un identifieur unique se terminant par « *White* », *e.g.*, aXzrt18_White ; le 2ème joueur aura pour nom un identifieur unique se terminant par « Black », e.g., yy18Hj_Black), ainsi l'utilisateur n'aura pas à passer par un menu
- Rajouter la possibilité de paramétrer le temps de la partie (Time game (min)) avant le début de la partie (une fois le premier coup joué, cette possibilité disparaîtra). Le choix de l'interaction utilisateur pour paramétrer le temps est laissé libre (e.g., via un boutton ou un menu)
- Corriger le fait que le temps s'écoule alors qu'aucun coup n'a encore été joué, i.e., enclencher le décompte du temps après le 1er coup blanc



Exercice 3. Un vrai bug

Tester et corriger

Exécuter les cas de test contenus dans la class TestPiece.java (package jchess.test).

Il y a un cas de test qui ne passe pas.

- Expliquer, dans votre rapport, la nature du « bug » et la conséquence directe sur l'utilisateur du jeu d'échecs
- Corriger le bug (facile)

Il y a en fait un autre bug avec un autre type de pièce.

- Ecrire un ensemble de cas de tests pour mettre en exergue le bug
- Expliquer, dans votre rapport, la nature du « bug » et la conséquence directe sur l'utilisateur du jeu d'échecs
- Corriger le bug

Exercice 4. Monteur pour la lisibilité

L'écriture de mouvements de coup avec 4 entiers est sujet à erreur et jugée peu lisible.

Proposer deux alternatives :

- Une basée sur une notation algébrique (« e4 »)
- Une autre basée sur l'enchaînement de méthodes (e.g., xFrom(4).yFrom(5).xTo(6).yTo(5))

Implémenter une solution en utilisant un design pattern et expliquer votre implémentation dans votre rapport.

Exercice 5. Variantes

Ecrire trois variantes du jeu d'échecs :

- Au lieu d'avoir un échiquier « 8 par 8 », on aimerait généraliser et avoir un échiquier N x M ($N > 1$, $M > 1$)
- Créer une nouvelle pièce avec de nouveaux déplacements, une nouvelle représentation graphique, etc
- Initialiser la position de départ avec une disposition des pièces aléatoires. Vous définirez également une interface pour permettre à un développeur d'implémenter un algorithme de disposition des pièces (autre que aléatoire)

Implémenter une solution en utilisant un design pattern et expliquer votre implémentation dans votre rapport.

Exercice 6. En visite

Ecrire une méthode m1 permettant de recenser le nombre de types de pièces de chaque côté.
Ecrire une méthode m2 permettant de donner un « score » à chaque camp, le score étant défini comme la somme des valeurs de pièces (pion : 1, fou/cavalier : 3, tour : 5, reine : 10, roi : 1000).
Vous proposerez une implémentation dans laquelle m1 et m2 repose sur la même interface de visite.

Implémenter une solution en utilisant un design pattern et expliquer votre implémentation dans votre rapport.

Exercice 7. Décoration

On souhaite :

- Enregistrer le « temps » associé à chaque coup
- Permettre d'écrire un commentaire textuel associé à un coup

Implémenter une solution en utilisant un design pattern et expliquer votre implémentation dans votre rapport

Exercice 8. AI

Modéliser, implémenter, test

On souhaite écrire un « moteur d'échecs » (chess engine) pour pouvoir jouer contre lui.
Ce moteur doit être capable de fonctionner quelque soit la variante du jeu d'échecs.
On souhaite intégrer trois modes:

- « random » : dans le cas, le moteur joue de manière aléatoire (uniquement des coups légaux évidemment)
- « glouton» : le moteur utilise un algorithme « glouton[2] » dans le sens où à chaque coup, s'il est possible de prendre une pièce adverse, le moteur effectue la prise. Si plusieurs prises sont possibles, alors la pièce adverse qui a le plus de valeurs sera privilégiée
- « min-max » : une implémentation de l'algorithme min-max avec comme fonction d'évaluation la somme des valeurs de pièces dans chaque camp

Vous écrirez quelques cas de tests pour tester la stratégie gloutonne.
Vous écrirez également une procédure permettant de faire jouer « min-max » contre « glouton »

Exercice 9. Bilan

- Produire le diagramme de classes UML de l'application <u>après vos différentes itérations (refactoring, test, correction de bugs, implémentation)</u> ;
- Vous produirez un résumé des différents design patterns que vous avez utilisés (en mentionnant les sections de votre rapport détaillant chaque design pattern).
- Vous produirez une critique du code original suite à votre expérience de développement.

[2] Malheureusement une stratégie gloutonne n'aura pas les propriétés algorithmiques sous-entendues. Il est même possible qu'une stratégie « gloutonne » basée sur la prise de pièces systématique soit inférieure à une stratégie aléatoire