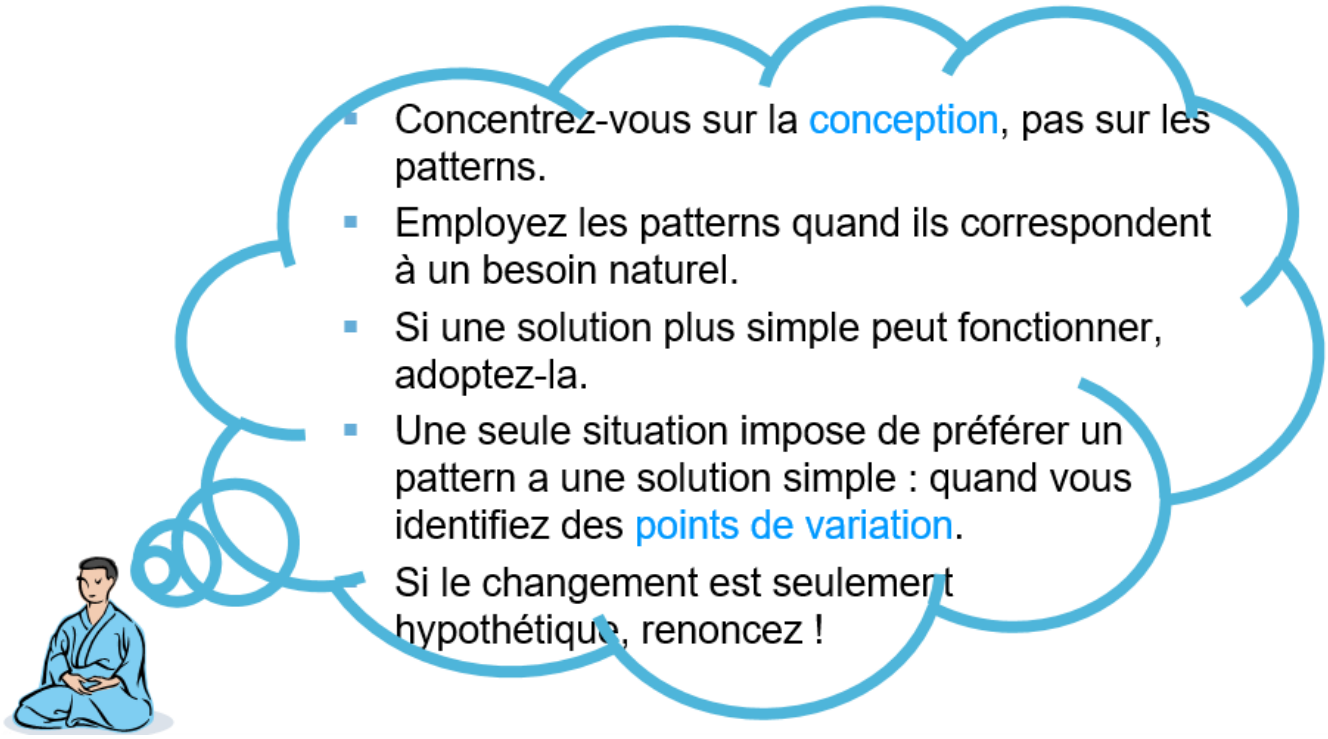
Année universitaire

2019 - 2020

Module 4-6-COMSC2-C

Conception Orientée Objet & Design Patterns

Ateliers



Françoise PERRIN

Préambule

Compétences visées

Être capable de concevoir et développer en Java des **programmes souples, extensibles, réutilisables et faciles à maintenir.**

Plus précisément, être capable d'identifier et de mettre en œuvre les **Design Patterns** nécessaires à la résolution de certains problèmes « types ».

Moyens

Mettre en œuvre, conjointement dans une même application (jeu d'échec - Cf. annexes), plusieurs Design Patterns (GOF) en appliquant les bonnes pratiques de conception objet (Cf. annexes).

Déroulement

- Un travail en 3 temps sur une application dont la conception et le développement sont à enrichir :
    - Identification et compréhension des design Patterns mis en œuvre dans la base fournie (Diagramme de classes en annexe).
    - Correction de quelques faiblesses de conception et évolutions cosmétiques mineures.
    - Évolutions fonctionnelles majeures.
- Une pédagogie active :
    - Des problèmes à résoudre à partir des éléments fournis (suggestion de patterns, ressources en ligne à étudier).
    - Des recherches à effectuer en équipe avec partage et transfert de compétences auprès des membres de l'équipe et des autres équipes (9 équipes).
    - 3 à 4 enseignants facilitateurs qui vérifient la bonne acquisition des compétences et qui occasionnellement fournissent des compléments d'information.
    - Un cahier de notes à compléter au fur et à mesure (schémas, questions, etc.).

Évaluation

- Participation active aux différents ateliers : 20%.
- DS1 papier/crayon : 20%.
- DS2 papier/crayon : 60%.

Atelier N°1

Objectif de l'atelier

L'objectif de ce 1er atelier est de vous approprier la conception et le code existant (e-campus), d'identifier les patterns mis en œuvre dans l'application et de comprendre ce qui justifiait leur utilisation.

Le programme fourni permet de simuler les déplacements simples (pas de prise en passant, roque du roi, etc.) des pièces d'un jeu d'échec sur un échiquier et d'afficher la trace d'exécution. La fin de partie n'est pas gérée (mat, pat, etc.).

L'architecture globale de l'application est donnée en annexe. Elle est déjà pensée pour qu'une exécution de l'application en mode client/server soit possible : 1 server avec le modèle et son contrôleur, 2 clients avec chacun 1 vue (joueur blanc, joueur noir) et 1 contrôleur.

Déroulement de l'atelier

- Installation des sources et exécution du programme. N'hésitez pas à verbaliser la séquence des appels de fonctions (sans les lire en détail), en binôme ou en équipe, pour vous approprier l'architecture.
- Par équipe :
    - Étude bibliographique du ou des patterns de son N° d'équipe et identification dans l'application (lecture en détail cette fois).
    - Conception et réalisation Diaporama. Quand validation par prof, dépôt sur espace partagé.
- Restitution aux autres équipes – questions/réponses.
    - Dans l'ordre des N° de patterns pour ceux mis en œuvre dans le programme, puis les autres (en italique)
    - Chacun complète son carnet de bord (MAJ fiche des patterns).

Pour chaque Design Patterns

Vous devez être capable de préciser :

- Sa classification.
- Son intention dans l'absolu et surtout **dans notre application** (pourquoi l'a-t-on utilisé).
- Ses participants dans notre application (les ajouter sur le diagramme de classe).
- Les interactions entre ses participants (diagramme de séquence ou autre).
- Si nous l'avons utilisé dans sa forme canonique et, si non, pourquoi.

et de proposer un autre exemple d'utilisation qui vous parle (pris dans votre contexte professionnel ou autre).

Liste des Design Patterns à étudier

1. *Anti Patterns*
2. Facade + MVC
3. Mediator, *Singleton*
4. Bridge
5. Factory Method, *prototype*
6. Builder
7. Composite
8. Iterator
9. *Flyweight*

Atelier N°2

Objectif de l'atelier

L'objectif du 2ème atelier est de corriger quelques faiblesses de conception et de rendre le logiciel plus convivial, sans évolution fonctionnelle majeure (impact fonctionnel uniquement sur les classes de la vue).

Les Design Patterns à étudier et à mettre en œuvre sont : **Template Method, Observer, Decorator, Command**.

Déroulement de l'atelier

- Par équipe :
  - Pour chaque exercice lisez bien l'énoncé dans son intégralité (Objectif, DP, Trucs et astuces).
  - Étude bibliographique du pattern nécessaire - ressources en annexe.
  - Réflexion sur la manière dont il va permettre de résoudre le problème.
  - Mise à jour du diagramme de classes à faire valider avant de commencer à coder.
  - Préparation d'un Diaporama de présentation du pattern.
  - En binôme ou individuellement ou en équipe, développement de la solution.
- Présentation du pattern par l'équipe choisie par les profs. Chacun complète son carnet de bord.
- Débriefe collectif sur les compétences acquises

Fonctionnalités à programmer

1. Supprimez la redondance dans la classe `ChessGUI`

Objectif

Vous pouvez constater que dans la classe `ChessGUI`, on répète 4 fois le même code, les seuls éléments qui varient étant les valeurs portées par les axes (1 à 8 et 'a' à 'h') et le sens d'affichage (8 -> 1 et 'a' -> 'h'). Optimisez ce code pour supprimer la redondance.

Patterns à mettre en œuvre

- Aucun ☺.
- Il s'agit juste d'un échauffement pour vous re-familiariser avec le langage Java (n'y passez-pas plus de 15' tests compris, cela ne rend pas le programme plus convivial, seulement plus facile à maintenir).

2. Supprimez le test de couleur dans la classe `Pion`

Objectif

Vous pouvez constater que dans la classe `Pion`, plusieurs méthodes ont des comportements différents en fonction de la couleur du pion. Ce n'est pas très objet : normalement un `Pion` « sait » qu'il est un pion blanc ou noir et agit en conséquence ; il ne doit pas avoir à « se demander » s'il est un pion blanc pour décider de son comportement.

Patterns à mettre en œuvre

- Template Method.

Trucs et astuces

- Pour tester, commentez et dé-commentez les lignes nécessaires dans la fabrique de `PieceModel`.

3. Modifiez à la volée la texture des `ChessSquareGUI`

Objectif

La coloration actuelle des cases est dégradée. A travers un menu, proposez à l'utilisateur de choisir un affichage uni ou dégradé. L'effet devra être immédiat (sans qu'il ne soit besoin de déplacer une pièce).

Patterns à mettre en œuvre

- Observer.
- Programmez votre propre mécanisme ou, si vous avez intégré le concept de délégation, utilisez celui de Java qui s'appuie sur les classes `PropertyChangeSupport et PropertyChangeListener`.

Trucs et astuces

- Ajoutez un attribut `Enum` (uni - dégradé) dans l'objet de configuration de la vue (`ChessGUIConfig`) et rendez-le observable.
- Utilisation des menus : des `JMenu` composés de `JMenuItem` le tout stocké dans une `JMenuBar`. Certains `JMenuItem` sont des `JMenu` ou des `JRadioButtonMenuItem`. On peut grouper les `JMenuItem` dans un `ButtonGroup` pour rendre les choix exclusifs.

4. Changez leur fond et encadrez les JPanel des axes et de la trace d'exécution

Objectif

Dis comme ça (titre de l'exercice), ça n'a pas l'air bien sorcier ☹ (ajout instructions `setBorder()` et `setBackground()` sur les `JPanel` de la classe `ChessGUI` devrait être suffisant).

Maintenant si je vous dis : chez certains clients, les `JPanel` seraient vierges, chez d'autres, ils auraient un fond ou encore un fond et une bordure, et puis ils pourraient en plus clignoter quand la souris les survole, et puis ce serait sympa si tout cela pouvait s'appliquer également aux `JButton`, aux `JLabel`, bref à **tous** les `JComponent` utilisés dans **tous** les programmes de l'entreprise, etc. etc. etc. Là ça devient plus rigolo…

Patterns à mettre en œuvre

- Decorator.

Trucs et astuces

- Inutile de programmer le clignotement, en revanche, il faut que cela s'applique potentiellement à tous les `JComponent`.

5. Programmez un mécanisme undo/redo des actions sur le menu

Objectif

L'idée est d'annuler ou de rejouer, parfois plusieurs fois de suite des actions de l'utilisateur sur le menu.

Pour que ce soit significatif, ajoutez un nouveau menu à la barre de menu pour changer la couleur d'affichage des cases de destination (l'attribut correspondant est déjà prévu dans la classe `ChessGUIConfig`).

Ajoutez aussi un nouveau menu avec les options undo et redo (+ un attribut dans la classe `ChessGUIConfig`) pour tester votre mécanisme.

Patterns à mettre en œuvre

- Command.
- Ce pattern peut être implémenté de différentes manières (Compensation, Memento, Replay). Choisissez d'implémenter celle que vous préférez. Les 3 méthodes doivent être implémentées entre les différentes équipes pour pouvoir les comparer ensuite.

Trucs et astuces

- Pour choisir une couleur : `JColorChooser.showDialog(…)`

Atelier N°3

Objectif de l'atelier

L'objectif du 3ème atelier est d'enrichir le métier avec de nouvelles règles du jeu (impact sur les classes du model) et de modifier l'architecture en proposant une version client/server (impact sur les classes du controller).

Les Design Patterns à étudier et à mettre en œuvre sont : **Strategy, Singleton, Adapter, Abstract Factory, Proxy**.

Les techniques Java à étudier et à utiliser sont : **programmation concurrente, communication à travers des sockets**.

Déroulement de l'atelier

- Toutes les équipes étudient et programment la 1ère fonctionnalités (« Tempête sur l'échiquier ») puis choisissent entre la 2ème et la 3ème.
- Par équipe :
    - Pour chaque exercice lisez bien l'énoncé dans son intégralité (Objectif, DP, Trucs et astuces).
    - Étude bibliographique du pattern nécessaire - ressources en annexe.
    - Réflexion sur la manière dont il va permettre de résoudre le problème.
    - Mise à jour du diagramme de classes à faire valider avant de commencer à coder.
    - Préparation d'un Diaporama de présentation du pattern.
    - En binôme ou individuellement ou en équipe, développement de la solution.
- Présentation du pattern par l'équipe choisie par les profs. Débriefe. Chacun complète son carnet de bord.

Fonctionnalités à programmer

1. Changez les règles du jeu : « Tempête sur l'Échiquier »

Objectif

L'objectif est de programmer une version très simplifiée de « Tempête sur l'Échiquier » qui propose plusieurs variantes (https://fr.wikipedia.org/wiki/Temp%C3%AAte_sur_l%27%C3%A9chiquier)

Une 1ère variante consiste à considérer qu'une pièce (`PieceModel`) n'a plus toujours son algorithme de déplacement propre mais a un algorithme de déplacement différent selon sa position sur le damier. Elle garde son apparence (image de `ChessPieceGUI`), mais n'a plus toujours le même comportement.

Son comportement est défini par colonne :

- Toutes les pièces positionnées sur les colonnes 'a' et 'h' ont un comportement de Tour.
- Toutes les pièces positionnées sur les colonnes 'b' et 'g' ont un comportement de Cavalier.
- Toutes les pièces positionnées sur les colonnes 'c' et 'f' ont un comportement de Fou.
- Toutes les pièces positionnées sur les colonnes 'd' et 'e' ont leur comportement du mode « Normal » (un Fou a un comportement de Fou, etc.).

Une 2ème variante du mode « Tempête », consisterait à imaginer que certaines pièces ne seraient plus supprimées du jeu lorsqu'elles seraient prises mais retourneraient à leur position initiale si elle n'était pas occupée. Nous ne programmerons pas les détails de cette variante, mais juste la signature des méthodes dans les classes adéquates. Une 3ème variante, …etc.

Le choix du mode de jeu est défini au lancement du programme mais peut être modifié en cours d'exécution (lancez une pop-up en réaction à un clic droit de l'utilisateur sur le damier pour choisir le mode).

Patterns à mettre en œuvre

- Strategy : les `PieceModel` délèguent à leur stratégie de déplacement (`RoiMovementStrategy` pour le `Roi`, etc.), le soin de vérifier que le déplacement est possible.
- Singleton : chaque Strategy / Factory est un Singleton.
- Abstract Factory : les fabriques servent à fabriquer la bonne Strategy de déplacement (`MovementStrategy`) en fonction de la position de la pièce et/ou de son nom selon le mode de jeu (« Normal » / « Tempête »).
- Adapter : les fabriques ont besoin du nom et des coordonnées de la `PieceModel` pour fabriquer les Strategy. Vous leur fournirez soit ces 2 paramètres, soit un adaptateur de `PieceModel`.

Trucs et astuces

- Codez et testez dans un 1er temps « en dur » le mode « Normal » en utilisant les Strategy. Vous devez obtenir le même comportement que précédemment. Puis codez et testez « en dur » le mode « Tempête » avant de programmer (et tester) les fabriques abstraites.
- Ajoutez un attribut `Enum` (normal - tempête) dans l'objet de configuration du model (`ChessModelConfig`).
- Une fabrique de `MovementStrategy` est disponible sur le e-campus.

2. Changez l'architecture : mode client/server

Objectif

Il s'agit cette fois d'avoir 2 instances, de ce qui semble être le même programme, qui s'exécutent en même temps et qui communiquent ensemble. Après chaque déplacement autorisé sur l'un des damiers, le déplacement doit être « visible » sur le damier de l'autre joueur (Cf. diagramme d'architecture en annexe).

La conception respectée jusqu'alors fait qu'il n'y aura aucun changement a effectuer ni sur les classes métier, ni sur les classes de la vue. Il suffit que les contrôleurs invokent respectivement les méthodes du modèle ou de la vue d'une part et envoie des messages à travers un canal de communication (sockets) d'autre part. Ces messages une fois décodés, permettront d'invoquer les méthodes du model ou celles de la vue de chaque joueur.

Patterns à mettre en œuvre

- Proxy.

Trucs et astuces

- Dessinez ou verbalisez et faites valider le diagramme de séquence avant tout développement.

3. Complétez les classes « métier »

Objectif - Vous pouvez :

- Programmer le roque du roi, la prise en passant, etc.
- Arrêter la partie lorsqu'elle est gagnée (échec et mat) ou lorsqu'elle est nulle.

Trucs et astuces

- Identifiez bien les interfaces/classes et méthodes responsables des nouvelles actions en veillant à respecter l'encapsulation initiale et à limiter l'impact des évolutions sur les classes existantes.
- Réfléchissez aux impacts sur l'IHM (prise en passant, roque, etc.).

Annexes

Bibliographie

Cours en ligne Design Pattern

- https://www.dofactory.com/net/design-patterns
- https://www.baeldung.com/tag/pattern/
- https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/
- https://www.jmdoudoux.fr/java/dej/chap-design-patterns.htm

Exemple avancé

DP Command : http://zenika.developpez.com/tutoriels/java/patterns-command/

But du programme

Il s'agit d'un jeu d'échec.

Dans la version initiale, les deux joueurs peuvent jouer sur le même ordinateur. A termes, ils pourront jouer sur 2 ordinateurs différents (et distants), la communication des données entre les deux se faisant à travers des sockets.

Seules les pièces du joueur courant (blanc ou noir) peuvent être déplacées. Le joueur blanc commence.

Lorsque le joueur sélectionne une pièce sur l'échiquier avec sa souris, les cases où la pièce peut être déplacée sont affichées d'une autre couleur.

Lorsqu'il sélectionne une pièce et la déplace sur une case de destination, le déplacement de la pièce est visible (drag&drop).

Si le déplacement est légal (selon les règles du jeu d'échec), la pièce reste sur la case de destination et « prends » l'éventuelle pièce du joueur adverse qui s'y trouvait, sinon, elle retourne à sa position initiale.

Conception du programme

Interfaces et classes du model

La conception du projet a permis d'identifier un certain nombre d'objets « métier » et en particulier :

- 1 `ChessModel` qui gère toutes les règles du jeu d'échec : il sait si les déplacements sont légaux (et dans ce cas les autorise), si le roi est en échec, si la partie est finie, etc.
- Des `PieceModel` : `Roi`, `Reine`, `Fou`, `Pion`, `Cavalier`, `Tour`. Il existe 16 pièces blanches et 16 pièces noires. Chaque pièce sait dire comment et où elle peut se déplacer.
- 1 `ChessImplementor` qui permet à l'objet ChessModel de s'affranchir de toute la communication avec les `PieceModel` : il crée les `PieceModel` et sait les manipuler. La fabrique `ChessPieceModelFactory` lui permet de créer une liste de `PieceModel`. Elle s'appuie sur les méthodes d'une classe `d'Introspection` qui sait créer un objet connaissant le nom de sa classe (nom donné sous forme de `String`).

Les classes sont donc parfaitement bien encapsulées et les seules interactions possibles avec une IHM se font à travers le `ChessModel` et en aucun cas une IHM ne pourra directement déplacer une `PieceModel` sans passer par les méthodes du `ChessModel` (en réalité, à travers un objet `ChessGameController` – Cf. plus loin).

Interfaces et classes de la view

La vue (classe `ChessGUI`) est composée d'un damier (`ChessGridGUI`), lui-même composé de cases (`ChessSquareGUI`) sur lesquelles sont déposées ou non des (`ChessPieceGUI`). La classe `ChessGUI` implémente l'interface `ChessGUI`.

Les écouteurs de `MouseEvent` sur les `ChessSquareGUI` ou les `ChessPieceGUI` pourront invoquer directement les méthodes du controller. En revanche, la classe `ChessGUI` sert de façade à l'ensemble de la vue et les seules

Interfaces et classes du controller

Le controller sert d'interface entre le model et la view.

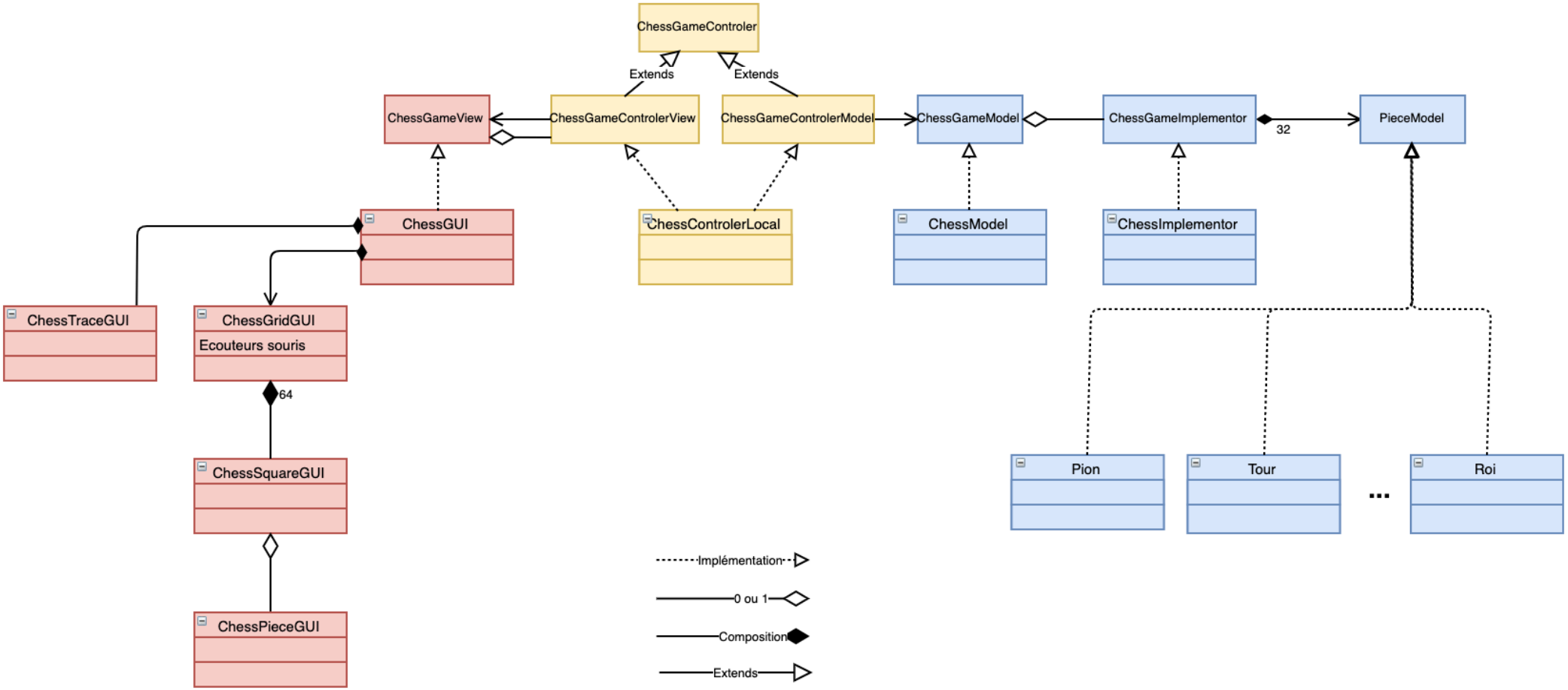
Les méthodes du controller sont invoquées par les écouteurs de `MouseEvent`.

Il a 3 responsabilités :

- Propager les requêtes vers le model. Par exemple lorsque l'utilisateur a déplacé la `ChessPieceGui` sur la view, le controller invoque la méthode de déplacement du model (`move()`).
- Indiquer à la (ou les) vue(s) les actions à effectuer : en fonction du retour de la méthode `move()` du model, la `ChessPieceGui` déplacée peut être replacée à sa place initiale si le déplacement n'était pas légal ou effectivement positionnée à sa nouvelle place.
- Convertir les coordonnées de la vue vers celles du model. En effet, les indices des `ChessSquareGUI` sur le `ChessGridGUI` varient de 0 à 63, alors que les coordonnées des `ChessPieceModel` sont stockées sous la forme ['a', 8] en haut à gauche, et ['h', 1] en bas à droite.

Une classe `AbstractController` est fournie. Elle gère en particulier les conversions de coordonnées. La classe `ControllerLocal` sert pour le jeu en « local ». D'autres classes seront à créer pour le mode Client/Server.

Diagramme de classe de l'application existante



L'architecture permet de simuler le jeu d'échec de 2 manières :

- En mode local, il existe 1 instance de model, 1 instance de controller, 1 instance de view. Les 2 utilisateurs jouent sur la même vue.
- En mode client/server il existera :
    - Sur le serveur 1 instance de model et son instance de controller.

  Sur chaque client (joueur noir et joueur blanc) 1 instance de view et son instance de controller.

Bonnes pratiques

Principe à respecter pour une bonne conception

- Principe de responsabilité unique (SRP) : « A class should have one reason to change. »

  ⇨ Une classe ne doit posséder qu'une et une seule responsabilité, et réciproquement, une responsabilité ne doit pas être partagée par plusieurs classes.

- Principe d'ouverture fermeture (OCP) : « Classes, methods should be open for extension, but closed for modifications. »

  ⇨ Une classe, une méthode, un module, un système doivent pouvoir être étendus, supporter différentes implémentations (Open for extension) sans pour cela devoir être modifiés (closed for modification).

- Principe de Substitution de Liskov (LSP) : « Subtypes must be substituable for their base types. »

  ⇨ Les sous classes doivent pouvoir être substituées à leur classe de base sans altérer le comportement de ses clients. Dit autrement, un client de la classe de base doit pouvoir continuer de fonctionner correctement si une instance de classe dérivée de la classe de base lui est fournie à la place.

- Principe de Ségrégation des Interfaces (SIP) : « Clients should not be forced to depend on methods that they do not use. »

  ⇨ Le but de ce principe est d'utiliser les interfaces pour définir des contrats, des ensembles de fonctionnalités répondant à un besoin fonctionnel. Il en découle une réduction du couplage, les clients dépendant uniquement des services qu'ils utilisent. En conséquence, toute classe implémentant une interface doit implémenter chacune de ses fonctionnalités.

- Principe d'Inversion de Dépendances (DIP) : « High level modules should not depend on low level modules. Both should depend on abstractions. » « Abstractions should not depend on details. Details should depend on abstractions. »

  ⇨ Les entités logicielles de haut niveau ne doivent pas dépendre des entités logicielles de bas niveau. Chacune doit dépendre d'abstractions. Ce principe est illustré par le principe dit d'Hollywood « Ne nous appelez pas, nous vous appellerons ». Mais pourquoi ? En règle générale les modules de haut niveau contiennent le cœur – business – des applications. Lorsque ces modules dépendent de modules de plus bas niveau, les modifications effectuées dans les modules « bas niveau » peuvent avoir des répercussions sur les modules « haut niveau » et les « forcer » à appliquer des changements, ce qui les rendrait incompatibles avec toute velléité de réutilisation.

  Donc pour découpler les dépendances entre objets/systèmes (les rendre indépendant), il faut toujours travailler avec des abstractions.

Comment faire ?

- Encapsuler ce qui varie.
- Préférer la composition à l'héritage (au sens délégation).
- Programmer des interfaces (au sens abstraction) et non des implémentations :
  - En Java une abstraction est une interface. Une classe abstraite n'est qu'un outil qui permet de factoriser du code.
  - En Java, une implémentation est une classe instanciable.
- Coupler faiblement les objets qui interagissent.