

**Année universitaire**

**2022 – 2023**

# 22\_23\_4IRC\_08\_COMSC2\_103\_MAJ

**Conception Orientée Objet &**

**Design Patterns**

**Ateliers**

## Bibliographie

### Cours en ligne Design Pattern

* <https://refactoring.guru/fr/design-patterns>
* <https://www.dofactory.com/net/design-patterns>
* <https://www.baeldung.com/tag/pattern/>
* <https://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/>

### Cours en ligne Accès concurrents

* <https://www.baeldung.com/a-guide-to-java-sockets>
* <https://docs.oracle.com/javase/tutorial/networking/sockets/index.html>
* <https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm>
* Pour se remettre dans le « JavaFX application thread » en cas de multithread : <https://thierrywasyl.wordpress.com/2014/02/09/update-your-scene-in-javafx/>

**Françoise PERRIN**

# Préambule

## Compétences visées

Être capable de concevoir et développer (en Java) des applications :

* **Faciles à développer**
* **Faciles à maintenir** (corrections et évolutions mineures)
* **Faciles à étendre** (ajout nouvelles fonctionnalités)
* **Dont on peut facilement réutiliser des composants.**

Pour ce faire, être capable, lorsque l’on conçoit une application, de repérer en quoi notre problématique est analogue à certains problèmes « types » et d’identifier, d’adapter et de mettre en œuvre le ou les **Design Patterns** nécessaires à sa résolution.

**Les Design Patterns sont des solutions génériques à des problèmes classiques et récurrents de la conception de logiciels.**

## Moyens

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

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. La fin de partie n’est pas gérée (mat, pat, etc.). L’architecture globale de l’application est donnée en annexe.

Vous devrez vous approprier le code existant et le compléter en respectant les bonnes pratiques.

## 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 (8 équipes).
* 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.).
* Une évaluation à travers un DS papier/crayon.

# 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),
* Identifier les patterns mis en œuvre dans l’application et comprendre ce qui justifiait leur utilisation,
* Répondre aux questions « Ce programme est il bien conçu ? Respecte-t-il les principes objets ? ».

## 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 d’un pattern et identification dans l’application (lecture en détail cette fois).
  + Conception d’un Poster/Powerpoint (Ressources en annexe). Quand validé par un prof.
* Restitution aux autres équipes – questions/réponses :
  + Dans l’ordre des N° de patterns pour ceux mis en œuvre dans le programme,
  + Chacun complète son Carnet de bord (MAJ fiche des patterns).
* Brainstorming/brainsketching :
  + Qu’est-ce qu’une bonne conception objet ?
  + Quels principes objets garantissent une bonne conception ?
  + Sur quels piliers objet s’appuient-ils ?

## 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 du Carnet de Bord).
* 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. Facade + MVC (…)
2. Mediator (controller) + Singleton
3. Bridge (ChessModel et ChessImplementor)
4. Builder (LauncherWithBuilder)
5. Factory Method (LauncherWithBuilder) et Prototype
6. Composite (view)
7. Iterator (…)
8. Anti-patterns

# 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

### 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 ChessPieceModel (PieceModelFactory).

### Modifiez à la volée la texture ou la couleur des SquareGUI

### Objectif

La coloration actuelle des cases est dégradée. La fenêtre de votre application est déjà munie d’un menu qui permet de changer le style (uni ou dégradé) ou la couleur des SquareGui.

Les valeurs sont bien changées dans la GuiFactory. Pour autant, il n’y a pas d’effet immédiat sur l’ensemble des cases et surtout, cela n’a d’impact que sur la case vers laquelle on déplace la prochaine pièce, ce qui n’est pas tout à fait le résultat escompté…

### Patterns à mettre en œuvre

* Observer.

### Trucs et astuces

* Programmez votre propre mécanisme ou utilisez celui de javafx qui s’appuie sur les interfaces ChangeListener et ObservableValue.
* Dans un 1er temps, modifiez la classe SquareGui existante (sauvegardez la avant). Vous apprendrez dans l’exercice suivant à éviter de modifier une classe existante.

### Reprogrammez la même action sans modifier une classe existante

### Objectif

Vous avez modifié une classe existante ce qui n’est pas toujours possible (ne vous appartient pas ou bien est déjà en production, ou est déclarée final).

Restaurez la classe SquareGui dans sa version initiale et programmer le DP qui permet de résoudre ce pb.

### Patterns à mettre en œuvre

* Decorator.

### Trucs et astuces

* Il existe une ligne à dé-commenter dans la GuiFactory.

### 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.

Complétez la classe MenuView (inutile de la décorer ☺) avec un nouveau menu qui propose les options undo et redo pour annuler ou rejouer vos commandes.

### Patterns à mettre en œuvre

* Command.
* Ce pattern peut être implémenté de 3 manières (Compensation, Memento, Replay). Choisissez celle que vous préférez mais les 3 doivent être implémentées pour pouvoir les comparer ensuite.

### Trucs et astuces

* Modifiez dans un 1er temps les écouteurs de menus de manière à ne plus modifier directement les properties de la classe GuiFactory mais d’invoquer (do) des instances de Command. Testez le « do » avant de mettre en place le « undo/redo ».
* DP Command : <http://zenika.developpez.com/tutoriels/java/patterns-command>[/](http://zenika.developpez.com/tutoriels/java/patterns-command/) (code sur e-campus)

# 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

### 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 (ChessPieceModel) 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 (l’écoute du clic droit de l’utilisateur sur le damier pour choisir le mode « Normal ou Tempête » est déjà codé).

### Patterns à mettre en œuvre

* Strategy : les ChessPieceModel 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.
* Adapter : les fabriques ont besoin du nom et des coordonnées de la ChessPieceModel pour fabriquer les Strategy. Vous leur fournirez soit ces 2 paramètres, soit un Adapter de ChessPieceModel (intérêt uniquement pédagogique…).

### 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.
* Une fabrique de MovementStrategy est disponible sur le e-campus. Attention, elle considère déjà que les Strategy sont des Singleton.

### 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 à effectuer ni sur les classes métier, ni sur les classes de la view. Il suffit que les controller invoquent respectivement les méthodes du model ou de la view d’une part et envoient 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 de la view 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.
* Un package de communication à travers des sockets est disponible dans le code fourni. Il permet de découpler au maximum le mécanisme de communication du reste de l’application et ainsi servir dans d’autres projets…

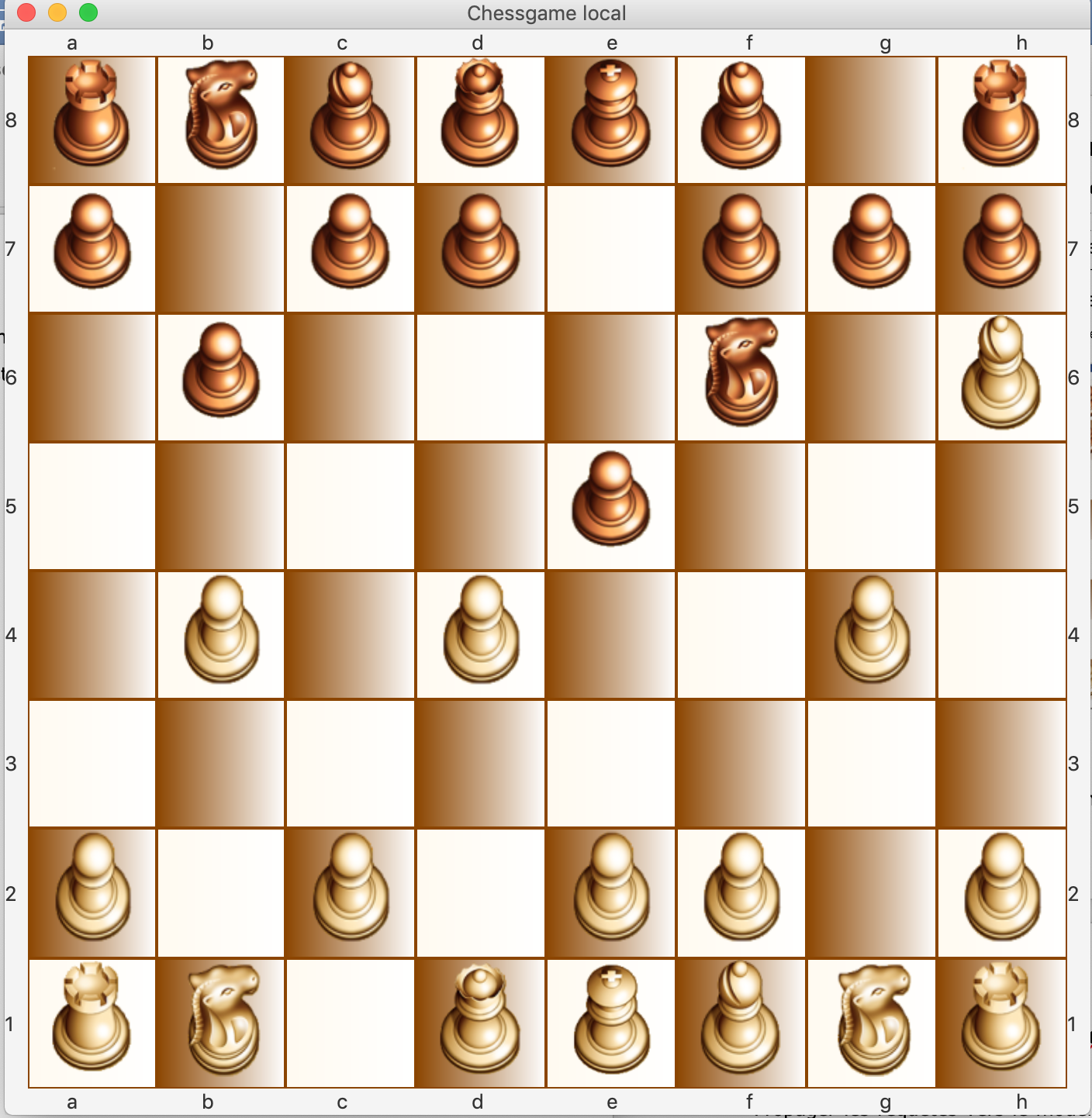
### Complétez les classes « métier »

### 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

## 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.

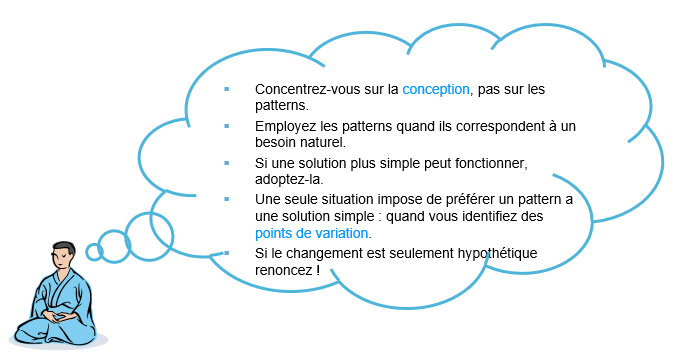
* Principe d’Inversion de Dépendances (DIP) : « High level modules should not depend on low level modules. Both should depend on abstractions. » « Abstractions should not depend on details. Details should depend on abstractions. »

⇨ 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.



## But du programme

Il s’agit d’un jeu d’échec.

Dans la version qui vous est fournie, 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 (implémenté par classe Model) 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 ChessPieceModel : 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 (implémenté par classe Implementor) qui permet à l’objet ChessModel de s’affranchir de toute la communication avec les ChessPieceModel: il crée les ChessPieceModel et sait les manipuler. La fabrique PieceModelFactory lui permet de créer une liste de ChessPieceModel. 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 ChessPieceModel sans passer par les méthodes du ChessModel (en réalité, à travers un objet ChessController – Cf. plus loin).

### Interfaces et classes de la view

La view (ChessView implémentée par classe View) est composée d’un damier (GridView), lui-même composé de cases (ChessSquareGUI) sur lesquelles sont déposées ou non des (ChessPieceGUI).

Les écouteurs de MouseEvent sur les ChessSquareGUI ou les ChessPieceGUI invoquent les méthodes du controller :

* Lorsque la pièce est sélectionnée (MousePressed) -> actionsWhenPieceIsSelectedOnGui().
* Au début du déplacement (DragDetected) -> actionsWhenPieceIsDraggedOnGui().
* En cours de déplacement (DragOver) : l’image de la pièce se déplace visuellement sur le damier.
* À la fin du déplacement (DragDropped) -> actionsWhenPieceIsMovedOnGui() .
* Lorsque la souris est relâchée (MouseReleased) -> actionsWhenPieceIsReleasedOnGui().

En revanche, la classe View sert de façade à l’ensemble de la vue et les seules interactions venant du controller se feront à travers la classe View qui les propagera au damier (GridView).

### Interfaces et classes du controller

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

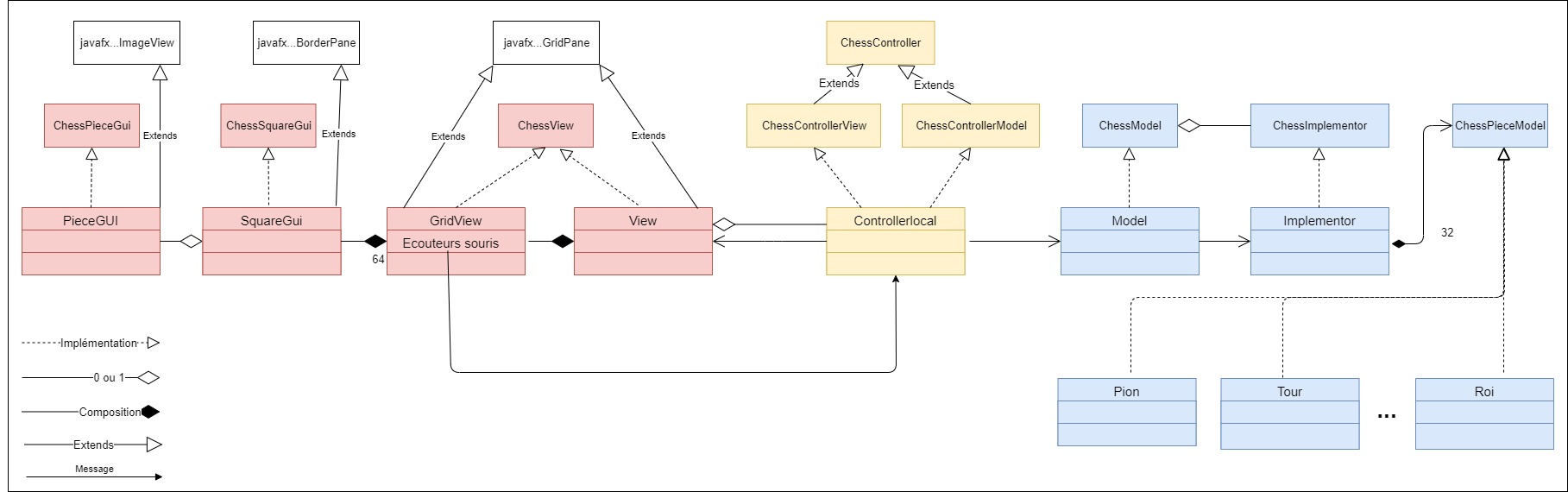
Les méthodes du controller sont invoquées par les écouteurs de MouseEvent.

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) view(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 view vers celles du model. En effet, le damier (GridView) sera implémenté par un GridPane dont les cases sont numérotées de [0, 0] en haut à gauche à [7, 7] en bas à droite, 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 (ControllerLocal), 1 instance de view. Les 2 utilisateurs jouent sur la même vue.
* En mode client/server il existera (diagramme à :
  + Sur le serveur 1 instance de model et son instance de controller (instance de ChessControllerModel).
  + Sur chaque client (joueur noir et joueur blanc) 1 instance de view et son instance de controller (instance de ChessControllerView).