

**Année universitaire**

**2022 - 2023**

# 3IRC - 3ICS

**Programmation Orientée Objet**

**Ateliers**

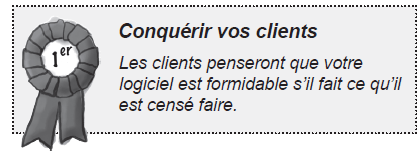
****

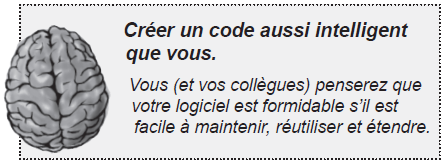
**Françoise PERRIN**

## Compétences visées

A l’issue de ce module, vous serez capable d’identifier et de mettre en œuvre les bonnes pratiques de conception et de programmation objet dans une application simple :

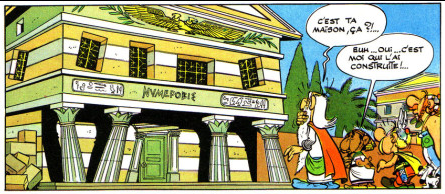
* S : Principe de Responsabilité Unique.
* O : Principe d’Ouverture/Fermeture.
* L : Principe de substitution de Liskov.
* I : Principe de ségrégation des Interfaces.
* D : Principe d’Inversion de Dépendances.





## Moyens

* Un 1er atelier (atelier 0) de prise en main des éléments du langage et des premiers concepts de la programmation objet (ICS : pour tous, IRC : pour les débutants).
* Un projet fil rouge dont l’architecture et le squelette sont définis, qu’il s’agira d’enrichir au fur et à mesure des ateliers à partir, dans un premier temps, de spécifications techniques précises puis de spécifications fonctionnelles détaillées.



## Pédagogie active

* Des vidéos à visionner en amont des séances pour acquérir les connaissances.
* Des recherches à effectuer en équipe avec transfert de compétences.
* Des développements à effectuer individuellement, en binôme ou en équipe.
* Un Carnet de Bord à compléter au fur et à mesure pour capitaliser connaissances et compétences (schémas, quizz…)
* Des enseignants facilitateurs qui vérifient la bonne acquisition des compétences et qui fournissent des compléments d’information.



## Évaluations

* DS papier/crayon (ICS = 70%, IRC = 100% (33% note module) : évalue la capacité à s’approprier la conception et le code d’un programme existant et à le maintenir en Java. Il s’agit d’être capable :
  + De justifier ou critiquer certains choix de conception objet.
  + De justifier ou critiquer le choix de certaines structures de données.
  + De dire ce que font certaines instructions en expliquant leur intérêt pour le programme.
  + De modifier des portions de codes volontairement mal conçues ou mal écrites.
  + D’ajouter des fonctionnalités inexistantes.
* Participation active aux différents ateliers et auto-évaluation (ICS « présentiel » = 30%).
* Rendu projet (ICS « distantiel » = 30%).

## Documentations à consulter

* 
* Documentation (« javadoc ») de la librairie standard : <https://docs.oracle.com/javase/8/docs/api/>
* The JavaTM Tutorials : <http://docs.oracle.com/javase/tutorial/java/index.html> à commencer par "Learning the Java Language" [https://docs.oracle.com/javase/tutorial/java/index.html](https://docs.oracle.com/javase/tutorial/java/index.html%3e)
* Kit de survie langage : <https://learnxinyminutes.com/docs/java/>
* Une mine d’or avec de très bons tutos et exemples : <http://www.java2s.com/>
* Chaîne Youtube de Xavier Blanc – Très pédago et synthétique : <https://www.youtube.com/channel/UCx35YRVHVBU_D6OBG-dqQRA>
* 
* Documentation de JavaFXTM <https://docs.oracle.com/javase/8/javafx/api/toc.htm>
* Pédago, peu théorique et directement opérationnel : <https://openclassrooms.com/fr/courses/1469021-les-applications-web-avec-javafx/1469344-presentation-de-linterface-graphique-en-javafx>
* Vidéos Youtube de Bucky Roberts – en anglais, très clair : <https://www.youtube.com/user/thenewboston/search?query=javafx>
* (Pour projet) Drag&Drop : <https://docs.oracle.com/javafx/2/drag_drop/jfxpub-drag_drop.htm>
* (Pour projet) Accès concurrents : <https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm>
* (Pour projet) Pour se remettre dans le « JavaFX application thread » en cas de multithread : <https://thierrywasyl.wordpress.com/2014/02/09/update-your-scene-in-javafx/>
* 
* <https://www.dofactory.com/net/design-patterns>
* https://refactoring.guru/design-patterns

# Atelier N°0 – 1ère partie (ICS)

## Compétences acquises à travers l’atelier (être capable de…)

* Ecrire du code Java.
* Maitriser les différents éléments du langage.
* Apprécier la philosophie du langage.

## Connaissances acquises à travers l’atelier

* Bases du langage Java (types, structures de contrôle, visibilité/portée des variables/méthodes, fonction main()).
* Bases de la Programmation Orientée Objet en Java : interface, classe, objet, référence, constructeur.

## Déroulement de l’atelier

### Étude préliminaire : vidéos à visionner avant l’atelier

1. Vidéo Youtube de Xavier Blanc « [Programmation Orientée Objet - Cours 1](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10911) [»](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10911) (les 13 premières minutes)
2. Vidéo Youtube de Xavier Blanc [« Programmation Orientée Objet - Cours 2 : La classe »](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10912) (14')
3. Vidéo Youtube de Xavier Blanc [« Programmation Orientée Objet - Cours 2 : Exemple »](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10913) (8')
4. Vidéo Youtube de Xavier Blanc [« Programmation Orientée Objet - Cours 3 : Interfaces »](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10914) (les 6 premières minutes - entendre "client/application cliente" à la place "d'utilisateur".

### Unique activité : les bases du Java

Lisez bien l’intégralité de chaque question (énoncé de la question, et s’il y a lieu : éléments fonctionnels, consignes, trucs et astuces, remarques)

##### **Vos 1ères classes :** suivez les directives de la vidéo « Programmation Orientée Objet - Cours 2 : Exemple » pour créer un projet MaBiblio, les classes Livre, Bibliotheque, MonMain et tester le programme.

**Trucs et astuces**

* + - Définir une classe veut dire définir un nouveau type.
    - Une classe regroupe un ensemble d’attributs et de méthodes.
    - Une classe permet de créer des objets à l’aide d’une méthode particulière appelée le constructeur.
    - Les attributs sont les propriétés de l’objet (ses données, ce qu’il sait/connait).
    - Les méthodes sont les opérations/fonctionnalités dont l’objet est capable (les services qu’il peut rendre, ce qu’il sait faire, etc.).
    - Les méthodes d’un objet ne sont accessibles en Java qu’en passant par une variable qui connait son adresse. Cette variable s’appelle une référence (pointeur en langage C).
    - Les applications clientes de l’objet (ou dit plus rapidement « les clients ») ont besoin que l’objet sache faire des choses (dans ses méthodes) mais ne veulent pas savoir comment il les fait. Dans cet exemple la classe MonMain est le client de Bibliotheque et Bibliotheque est le client des Livre.
    - Les références des objets de type Livre sont stockées dans l’objet de type Bibliotheque dans une Map (et plus précisément dans une HashMap). Une Map est une table associative qui, à la manière d’un dictionnaire, associe une clé et une valeur (par exemple mot/définition, Titre/Livre, etc.).
    - La méthode get(key) de l’interface Map renvoie une référence vers la valeur à laquelle la clé spécifiée est associée : liste.get(nom) renvoie une référence vers le Livre dont le titre est référencé par le paramètre nom.
    - La méthode put(key, value) de l’interface Map Associe la valeur spécifiée à la clé spécifiée. Si la Map contenait auparavant une association pour la clé, l'ancienne valeur est remplacée par la valeur spécifiée, sinon, le couple key/value est ajouté à la Map.

##### **Vos 1ers tests :**

1. Vérifiez qu’un livre existe bien avant de tenter de l’emprunter ou de le rendre. En effet, si vous ajoutez l’instruction biblio.emprunterLivre("Pas là"); à votre fonction main(), vous aurez une erreur d’exécution puisque la méthode emprunterLivre() de la classe Bibliotheque ne vérifie pas que la variable « l » référence réellement un Livre.

** Consignes**

* + - Dans la classe Bibliotheque, modifiez les méthodes emprunterLivre() et rendreLivre() afin de tester que le livre existe.

**Trucs et astuces**

* + - La méthode get(key) de l’interface Map renvoie une référence vers la valeur à laquelle la clé spécifiée est associée, ou null si cette Map ne contient aucune association pour la clé.
    - Syntaxe de l’instruction if-else :

if (condition == true) {

// actions quand condition vraie

}

else {

// actions quand condition fausse

}

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - Modifiez et testez d’abord la méthode emprunterLivre() puis en cas de succès reportez les modifications sur la méthode rendreLivre().

1. Vérifiez qu’un livre ne peut pas être emprunté s’il l’est déjà.

**Quelques éléments fonctionnels**

* + - Un livre ne peut être emprunté que s’il ne l’est pas déjà. Il s’agit donc de le vérifier et de retourner une erreur si l’emprunt est impossible (un booléen de valeur **false**).

** Consignes**

* + - Dans la classe Livre :
      * Initialisez l’attribut emprunt le dans le constructeur.
      * Changez la signature de la méthode emprunter() afin de retourner un booléen (true ou false) et effectuez le test nécessaire.
    - Modifiez la méthode emprunterLivre() de la classe Bibliotheque en conséquence.
    - Testez en ajoutant à la fonction main() les instructions suivantes (copier/coller):

// test emprunt OK livre existant

isEmpruntOk = biblio.emprunterLivre("Java");

**if** (isEmpruntOk == **true**) {

System.***out***.println("Emprunt livre 'Java' : Tout s'est bien passé");

}

// test emprunt KO déjà emprunté

isEmpruntOk = biblio.emprunterLivre("Java");

**if** (isEmpruntOk != **true**) {

System.***out***.println("Emprunt livre 'Java' : KO déjà emprunté");

}

// test emprunt KO livre non existant

isEmpruntOk = biblio.emprunterLivre("Pas là");

**if** (isEmpruntOk != **true**) {

System.***out***.println("Emprunt livre 'Pas Là' : KO pas dans la

bibliothèque");

}

##### **Vos 1ères boucles :**

1. Faites afficher le contenu de la bibliothèque de manière à obtenir le résultat suivant :

Affichage contenu Bibliothèque

Bibliotheque1 [liste={Java=Livre [nom=Java, emprunt=false], Autre=Livre [nom=Autre, emprunt=false]}]

** Consignes**

* + - Dans les classes Bibliotheque et Livre faites générer par votre IDE la méthode toString().
    - Ajoutez les lignes suivantes dans la fonction main() sous la ligne biblio.ajouterLivre("Java");

biblio.ajouterLivre("Autre");

System.***out***.println("Affichage contenu Bibliothèque avant

emprunt livre 'Java' \n" + biblio);

* + - Ajoutez des instructions d’affichage après les emprunts et rendus pour vérifier l’état de chaque Livre dans la fonction main().

**Trucs et astuces**

* + - La méthode toString() de la classe Object retourne une représentation sous forme de String de l’objet. On peut/doit la redéfinir dans chaque classe pour obtenir le résultat escompté.
    - L’instruction System.***out***.println(« Une chaine de caractère ») permet d’afficher du contenu à l’écran.
    - \n permet un saut de ligne.
    - Les instructions System.***out***.println(biblio.toString()) et System.***out***.println(biblio) et sont équivalentes : appel implicite de la méthode dans la 2ème instruction. De même, dans une méthode toString(), l’instruction **return** liste est équivalente à **return** liste.toString().
    - L’opérateur + permet de concaténer des chaines de caractères.

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - Toutes les classes en Java héritent de la classe Object et ses méthodes sont utilisables (Cf. plus tard). Regardez ce qui se passe lorsque vous ne redéfinissez pas la méthode toString() dans la classe Bibliotheque.

1. Nous allons créer une 2ème classe Bibliotheque dans laquelle nous utiliserons un tableau déclaré avec des [] (Array) pour stocker les objets de type Livre. Ce n’est pas forcément une bonne idée (l’utilisation d’une Map est plus judicieuse), c’est uniquement pédagogique, pour utiliser des boucles.

** Consignes**

* + - Renommez la classe Bibliotheque en Bibliotheque1.
    - Dupliquez la classe Bibliotheque1 en Bibliotheque2.
    - Dans la fonction main(), créez un objet de type Bibliotheque2 au lieu de Bibliotheque1 et testez. Normalement, le résultat de l’exécution est le même que précédemment puisque vous n’avez encore effectué aucun changement…
    - Dans la classe Bibliotheque2 remplacez l’instruction **private** Map<String, Livre> liste; par **private** Livre[] liste;
    - Dans le constructeur de la classe Bibliotheque2 remplacez l’instruction **this**.liste = **new** HashMap<String, Livre>(); par **this**.liste = **new** Livre[100];

1. Modifiez la méthode d’ajout de la classe Bibliotheque2 afin de vérifier qu’un nouvel ajout est possible, c’est-à-dire que la capacité du tableau n’est pas dépassée.

** Consignes**

* + - Ajoutez un attribut à la classe Bibliotheque2 pour compter le nombre d’éléments et initialisez le dans le constructeur.
    - Créez une méthode **private** **void** putLivre(Livre l) qui ajoute un Livre dans le tableau sous réserve que la capacité de stockage ne soit pas atteinte et en incrémentant le nombre d’éléments à chaque ajout.
    - Modifiez la méthode ajouterLivre() de la classe Bibliotheque2 en invoquant cette méthode.
    - Dans la fonction main(), commentez les lignes qui gèrent les emprunts et les rendus et testez. Pour vérifier que votre test de capacité fonctionne, réduisez la capacité du tableau à 2 et tentez d’ajouter 3 livres (remettez 100 après…).

**Trucs et astuces**

* + - La capacité d’un tableau est donnée par l’attribut length. Ainsi dans notre exemple, liste.length vaut 100.
    - L’indice des éléments va de 0 à length -1. L’accès au ième élément se fait ainsi : liste[i]
    - L’opérateur ++ permet d’incrémenter une variable.

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - En réalité, votre tableau, comme tous les tableaux, ne contient pas des objets Livre mais des références vers des objets Livre.
    - La valeur de liste n’est pas un tableau mais l’adresse du 1er élément du tableau.
    - Notre algo ne vérifie pas que le livre n’existe pas déjà dans la bibliothèque alors que l’utilisation d’une Map dans la version précédente empêchait d’avoir des doublons. Ne programmez pas pour autant cette vérification.

1. Modifiez la méthode d’emprunt de la classe Bibliotheque2 de manière à tenir compte de cette nouvelle implémentation des données (implémentation = nouveau mode de rangement/stockage).

** Consignes**

* + - Un livre ne peut être emprunté que s’il existe dans le tableau. Autant la méthode get() de la Map le vérifiait pour vous, autant en manipulant un tableau vous devez le vérifier par programme. Vous devez dont vérifier qu’il existe un Livre dans le tableau dont le nom correspond à celui qui doit être emprunter.
      * Munissez la classe Livre d’une méthode getNom() (faites la générer par votre IDE).
      * Dans Bibliotheque2 créez la méthode **private** Livre getLivre(String nom) qui retourne le Livre qui porte ce nom. Utilisez une boucle while pour parcourir le tableau et cherchez l’élément dont le nom est égal à celui passé en paramètre.
      * Modifiez la méthode emprunterLivre() en conséquence.
    - Dans la fonction main(), dé-commentez les lignes qui gèrent les emprunts et testez.

**Trucs et astuces**

* + - La méthode boolean equals(Object obj) vérifie l’égalité sémantique de 2 objets. Elle existe (doit exister) dans toutes les classes du Java (les vôtres et celles du JDK).
    - Syntaxe de l’instruction while :

while (condition)

// actions quand condition vraie

}

1. Modifiez la méthode de rendu de la classe Bibliotheque2 de manière à tenir compte de cette nouvelle implémentation des données.

** Consignes**

* + - Modifiez la méthode rendreLivre() de la classe Bibliotheque2 en conséquence.
    - Modifiez la méthode getLivre() en utilisant cette fois une boucle do-while pour parcourir le tableau.
    - Dans la fonction main(), dé-commentez les lignes qui gèrent les rendus et testez.

**Trucs et astuces**

* + - Syntaxe de l’instruction do-while :

if(condition) {

do {

// actions quand condition vraie

} while (condition) ;

}

1. Modifiez la méthode toString() de la classe Bibliotheque2 de manière à parcourir le tableau élément par éléments.

** Consignes**

* + - Utilisez la syntaxe classique d’une boucle for puis dans un 2ème temps, la syntaxe for-loop utilisable sur les tableaux.
    - Testez successivement les deux versions en ajoutant dans votre fonction main() des instructions pour afficher l’intégralité de la bibliothèque après chaque ajout/emprunt/rendu.

**Trucs et astuces**

* + - Syntaxe de l’instruction for :

for (initialisation; condition d’arrêt; increment)

// actions tant que condition vraie sur refVersLeTableau[i]

}

* + - Syntaxe de l’instruction for-loop :

for (typeElement maRefVersUnElement : refVersLeTableau)

// actions tant qu’il existe des éléments sur maRefVersUnElement

// qui référence successivement tous les éléments du tableau

}

* + - L’opérateur += permet d’ajouter une valeur à une variable en tenant compte de la précédente. Appliqué à un String, cela permet d’ajouter du contenu à la chaine de caractère.

##### **Votre 1ère interface : gestion d’implémentations techniques différentes**

Ce qui intéresse votre application cliente (fonction main()), c’est de gérer une bibliothèque, quel que soit la façon dont le développeur des classes a implémenté le stockage (Map, Array, etc.). Elle attend donc qu’un objet de ce type soit capable d’ajouter, d’emprunter et de rendre des livres. En Java, le comportement que l’on peut attendre des objets est déclaré dans une interface. Ensuite, il faut préciser que la classe qui servira à créer les objets implémente l’interface.

** Consignes**

* + - Dans un nouveau fichier, déclarez l’interface Bibliotheque ainsi.

**public** **interface** Bibliotheque {

**public** **void** ajouterLivre(String nom);

**public** **boolean** emprunterLivre(String nom);

**public** **void** rendreLivre(String nom);

}

* + - Modifiez les classes Bibliotheque1 et Bibliotheque2 afin qu’elles implémentent cette interface : **public** **class** Bibliotheque1 **implements** Bibliotheque

Normalement votre IDE va vous demander de changer la visibilité de vos méthodes en public.

Profitez en pour changer la visibilité de vos attributs en private pour qu’ils ne soient pas accessibles par les clients comme c’est le cas actuellement : à défaut de précision (public, private), les attributs et méthodes sont visibles dans tout le package.

* + - La plupart du temps les applications clientes ne connaissent pas les types réels qui sont implémentés (les classes) et délèguent à une fabrique (Factory) le soin de créer les objets. Dans un nouveau fichier, créez la fabrique suivante :

**public** **class** BiblioFactory {

**private** **static** Map<Integer, Bibliotheque> *mapBiblio*;

**static** {

*mapBiblio* = **new** HashMap<Integer, Bibliotheque>();

*mapBiblio*.put(1, **new** Bibliotheque1());

*mapBiblio*.put(2, **new** Bibliotheque2());

}

**public** **static** Bibliotheque getBiblio(**int** key) {

**return** *mapBiblio*.get(key);

}

}

Dans la fonction main() remplacez l’instruction Bibliotheque1 biblio = **new** Bibliotheque1();

par Bibliotheque biblio = BiblioFactory.*getBiblio*(1); et testez.

**Trucs et astuces**

* + - Une interface définit un Type mais sans se soucier des détails d’implémentation (rangement et stockage des données) comme le fait une classe. Elle se contente de décrire le comportement attendu des objets (déclaration de méthodes).
    - Lorsqu’une application cliente déclare un objet en précisant le nom d’une interface et non d’une classe c’est qu’il attend de l’objet qu’il soit capable de faire des choses, de rendre des services mais sans connaitre le type réel de l’objet (Ex : Un interface munie d’une méthode rouler() pourrait être implémentée par une classe Voiture, Trottinette, Ballon, PetitPois, etc.).
    - Une classe qui implémente une interface doit implémenter toutes ces méthodes (et si elle ne le fait pas, ces classes dérivées devront le faire – Cf. plus tard, notion d’héritage).

##### **Votre 2ème interface : implémentations fonctionnelles différentes**

Imaginons maintenant que la bibliothèque puisse stocker des livres mais aussi des DVD et tout autre type de média mais surtout que la façon d’emprunter et de rendre ces différents médias soit différente. Vu l’extrême simplicité de notre application, nous allons nous contenter d’imaginer mais nous allons néanmoins mettre en œuvre le mécanisme.

** Consignes**

* + - Dans un nouveau fichier, déclarez l’interface Media comme ci-dessous et modifiez la classe Livre pour qu’elle implémente cette interface.

**public** **interface** Media {

**public** **boolean** emprunter() ;

**public** **void** rendre() ;

}

* + - Dans un nouveau fichier, déclarez l’énumération MediaEnum ainsi.

**public** **enum** MediaEnum {

***LIVRE***, ***DVD***;

}

* + - Dans l’interface Bibliotheque
      * Renommez toutes les méthodes en supprimant le suffixe Livre. Si vous utilisez bien votre IDE pour le faire, la modification sera propagée dans les autres classes (Refactor/Rename).
      * Ajoutez une nouvelle méthode ajouter() avec la signature suivante : **public** **void** ajouter(String nom, MediaEnum mediaEnum) ;
      * Ajoutez une annotation sur la méthode ajouter() initiale pour préciser qu’elle n’est plus d’actualité : @Deprecated. Cela permet de ne pas modifier tous les clients qui l’utilisait mais incite les nouveaux clients à ne pas l’invoquer.
    - Dans la classe Bibliotheque1 :
      * Faites que la Map stocke des références vers des Media et non vers des Livre.
      * Faites générer par votre IDE la méthode ajouter() et complétez la ainsi

**public** **void** ajouter(String nom, MediaEnum mediaEnum) {

Media media = **null** ;

**if** (MediaEnum.***LIVRE*** == mediaEnum) media = **new** Livre(nom);

**if** (MediaEnum.***DVD*** == mediaEnum) media = **new** Dvd(nom);

liste.put(nom, media);

}

* + - * Les autres méthodes ne sont pas impactées.
    - Votre IDE vous signale que la classe Dvd n’existe pas. Plutôt que la faire générer, dupliquez la classe Livre, cela ira plus vite.
    - Imaginons qu’un DVD ne peut être rendu que si le disque correspond à la boite/pochette. Ajoutez un test dans la méthode rendre() de la classe Dvd pour le vérifier (test fictif sur un boolean que vous initialiserez à true).
    - Complétez la fonction main() avec les lignes suivantes et testez une 1ère fois avec le booléen à true puis une 2ème fois avec le booléen à false.

// Ajout dvd dans bibliothèque

biblio.ajouter("Windows10", MediaEnum.***DVD***);

biblio.ajouter("NouveauLivre", MediaEnum.***LIVRE***);

System.***out***.println("\nAffichage avant emprunt dvd 'Windows10' " + biblio);

// test emprunt dvd

biblio.emprunter("Windows10");

System.***out***.println("Affichage après emprunt " + biblio);

// test rendu dvd emprunté

biblio.rendre("Windows10");

System.out.println("Affichage après rendu " + biblio);

* + - Inutile de modifier la classe Bibliotheque2. Vous pouvez la déclarer @Deprecated.

**Trucs et astuces**

* + - Une énumération est une classe, dont les valeurs (des constantes) sont les seules instances possibles de cette classe (on peut lui ajouter des constructeurs/méthodes).
    - Les valeurs peuvent être comparées comme n’importe quel objet (Ex : (MediaEnum.***LIVRE)***.equals(mediaEnum)) mais aussi comme n’importe quelle primitive (Ex : MediaEnum.***LIVRE*** == mediaEnum)

##### **Vos 1ères classes/méthodes abstraites**(pour les plus avancés)

Supprimez la redondance visible dans les classes Livre et Dvd en créant la classe abstraite nécessaire.



# Atelier N°0 – 2ème partie (ICS et IRC débutants)

## Compétences acquises à travers l’atelier (être capable de…)

* Lire du code Java.
* Maitriser les différents éléments du langage.
* Comprendre en quoi la Programmation Orientée Objet facilite les tests, l’écriture (débuggage) et la résistance au changement par rapport à la programmation procédurale (1ère illustration du Principe de Responsabilité Unique).

## Connaissances acquises à travers l’atelier

* Piliers de la Programmation Orientée Objet : abstraction, encapsulation.
* Bases du langage Java (types, structures de contrôle, visibilité/portée des variables/méthodes, main()).
* Bases de la Programmation Orientée Objet en Java : interface, classe, objet, référence, constructeur.

## Déroulement de l’atelier

### Étude préliminaire : vidéos à visionner avant l’atelier

1. Vidéo Youtube de Xavier Blanc « [Programmation Orientée Objet - Cours 1](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10911) [»](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10911) (les 13 premières minutes)
2. Vidéo Youtube de Xavier Blanc [« Programmation Orientée Objet - Cours 2 : La classe »](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10912) (14')
3. Vidéo Youtube de Xavier Blanc [« Programmation Orientée Objet - Cours 2 : Exemple »](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10913) (8')
4. Vidéo Youtube de Xavier Blanc [« Programmation Orientée Objet - Cours 3 : Interfaces »](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10914) (les 6 premières minutes - entendre "client/application cliente" à la place "d'utilisateur".
5. Vidéo Youtube de Xavier Blanc « Programmation Orientée Objet - Cours 1 » (la fin : minutes 14 à 34’).
6. Vidéo Youtube de Xavier Blanc « Programmation Orientée Objet - Cours 2 : Conception (Encapsulation et Responsabilités) » (10’).

### Unique activité : « Ça veut dire quoi en fait « Programmer en objet » ? »

##### Exercice collectif « Java ⬄ POO » du Carnet de Bord.

** Consignes**

* + - Lisez tout d’abord les « Trucs et astuces » pour réviser les principaux concepts découverts dans les vidéos.
    - Lisez les codes proposés sur le carnet de bord et commencez à réfléchir aux questions.
    - Notez les réponses au fur et à mesure de la correction sur le Carnet de Bord.

**Trucs et astuces**

* + - Définir une classe veut dire définir un nouveau type.
    - Une classe regroupe un ensemble d’attributs et de méthodes.
    - Une classe permet de créer des objets à l’aide d’une méthode particulière appelée le constructeur.
    - Les attributs sont les propriétés de l’objet (ses données, ce qu’il sait/connait).
    - Les méthodes sont les opérations/fonctionnalités dont l’objet est capable (les services qu’il peut rendre, ce qu’il sait faire, etc.).
    - Les méthodes d’un objet ne sont accessibles en Java qu’en passant par une variable qui connait son adresse. Cette variable s’appelle une référence ( = pointeur en langage C).
    - Les applications clientes de l’objet (ou dit plus rapidement « les clients ») ont besoin que l’objet sache faire des choses (dans ses méthodes) mais ne veulent pas savoir comment il les fait. Dans le 1er exemple la classe TestMorpionBad est le client des classes MorpionBadV1 et MorpionBadV2.
    - L’instruction System.***out***.println(« Une chaine de caractère ») permet d’afficher du contenu à l’écran.
    - \n permet un saut de ligne.
    - Syntaxe de l’instruction if-else :

if (condition == true) {

// actions quand condition vraie

}

else {

// actions quand condition fausse

}

* + - Syntaxe de l’instruction for :

for (initialisation; condition d’arrêt; increment)

// actions tant que condition vraie sur refVersLeTableau[i]

}

* + - Syntaxe de l’instruction for-loop :

for (typeElement maRefVersUnElement : refVersLeTableau)

// actions tant qu’il existe des éléments sur maRefVersUnElement

// qui référence successivement tous les éléments du tableau

}

* + - La capacité d’un tableau est donnée par l’attribut length. Ainsi dans notre exemple, grid.length vaut 9.
    - L’indice des éléments va de 0 à length -1. L’accès au ième élément se fait ainsi : grid[i] et dans un tableau 2D grid[i][j]
    - L’opérateur ++ permet d’incrémenter une variable.

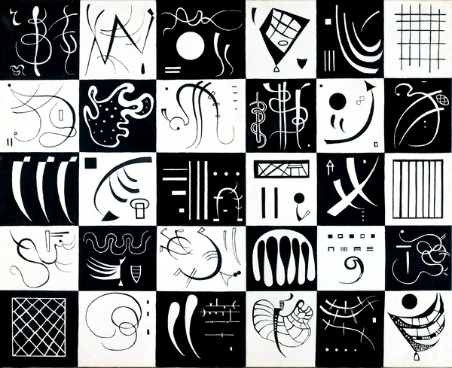
##### Exercice collectif « Magie de l’encapsulation » du Carnet de Bord.

** Consignes**

* + - Lisez tout d’abord les « Trucs et astuces » pour réviser les principaux concepts découverts dans les vidéos.
    - Lisez les codes proposés sur le carnet de bord et commencez à réfléchir aux questions.
    - Notez les réponses au fur et à mesure de la correction sur le Carnet de Bord

**Trucs et astuces**

* + - La méthode toString() de la classe Object retourne une représentation sous forme de String de l’objet. On peut/doit la redéfinir dans chaque classe pour obtenir le résultat escompté.
    - Les instructions System.***out***.println(biblio.toString()) et System.***out***.println(biblio) sont équivalentes : appel implicite de la méthode toString() dans la 2ème instruction.
    - L’opérateur + permet de concaténer des chaines de caractères.
    - Un objet est créé lorsque l’on fait appel à l’un de ses constructeurs (il peut y avoir plusieurs façons de créer l’objet, donc plusieurs constructeurs). Si le programmeur n’a pas explicitement écrit un constructeur le compilateur en synthétise un sans paramètre (en crée un dans le byteCode) qui initialise les attributs avec des valeurs par défaut (0 pour les numériques, false pour les booléens et null pour les références). On appelle ce constructeur le constructeur par défaut (défaut ⬄ absence).
    - La plupart du temps les applications clientes ne connaissent pas les types réels qui sont implémentés (les classes) et délèguent à une fabrique (Factory) le soin de créer les objets.
    - Une interface définit un Type mais sans se soucier des détails d’implémentation (i.e. rangement et stockage des données) comme le fait une classe. Elle se contente de décrire le comportement attendu des objets (déclaration de méthodes).
    - Lorsqu’une application cliente déclare un objet en précisant le nom d’une interface et non celui d’une classe c’est qu’elle attend de l’objet qu’il soit capable de faire des choses, de rendre des services mais n’a pas besoin de connaitre le type réel de l’objet (Ex : Un interface munie d’une méthode rouler() pourrait être implémentée par une classe Voiture, Trottinette, Ballon, etc. La seule info qui intéresse le client est de savoir que l’objet sait rouler()).
    - Une classe qui implémente une interface doit implémenter toutes ces méthodes (et si elle ne le fait pas, ces classes dérivées devront le faire – Cf. plus tard, notion d’héritage).



V. KANDINSKY – Trente (1937))

# Projet Fil-Rouge

## Cahier des charges

Il s’agit de réaliser un jeu de dames.

Les deux joueurs peuvent jouer sur le même ordinateur ou 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.

Pour déplacer une pièce sur le damier, le joueur la sélectionne avec la souris (clic) puis clique sur la case de destination. Si le déplacement est légal (selon les règles du jeu de dames), la pièce apparait sur la case de destination, sinon, elle reste à sa position initiale.

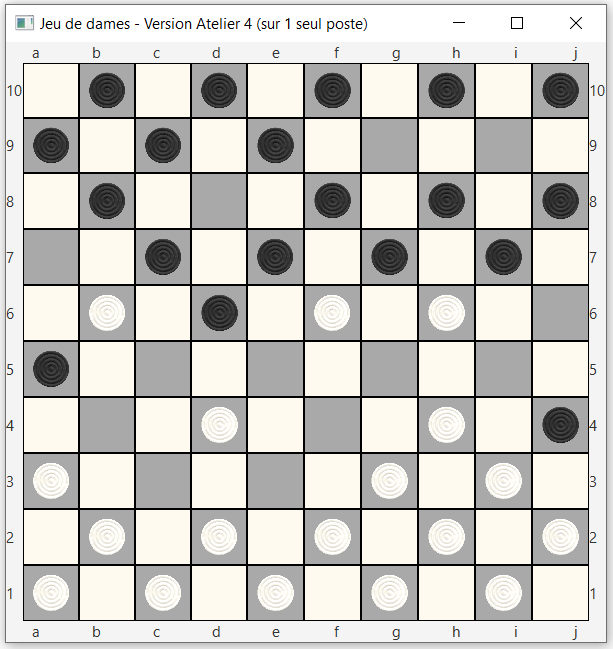
Le déplacement d’un pion est légal s’il est effectué en diagonale dans une case adjacente si la case est vide ou dans la case adjacente à celle d’une pièce à prendre (toujours en diagonale). Une dame peut se déplacer de plusieurs cases, en diagonale à condition qu’il y ait maximum une pièce à prendre sur le trajet.

Les pièces prises disparaissent du jeu.

Le joueur change à chaque déplacement sauf en cas de rafle (plusieurs prises possibles par la même pièce). Dans ce cas, le joueur devra cliquer à nouveau sur la pièce qu’il est en train de déplacer.

Le joueur courant doit jouer le « coup » le plus rentable (celui qui rapporte le plus de prises).

Vous n’aurez peut-être pas tous le temps de programmer toutes les règles (rafles, meilleur « coup » …) ni de tester la fin de partie ni le mode client/server, l’objectif étant pour chacun d’atteindre la fin de l’atelier N°3.



## Architecture logicielle

Le diagramme d’architecture de l’application (telle qu’elle sera à la fin de l’atelier N°3) est disponible en annexe (Figure N°7). Un diagramme de classe simplifié est proposé Figure N°1.

Nous avons choisi de découper l’application en 3 packages qui contiennent chacun plusieurs classes et interfaces :

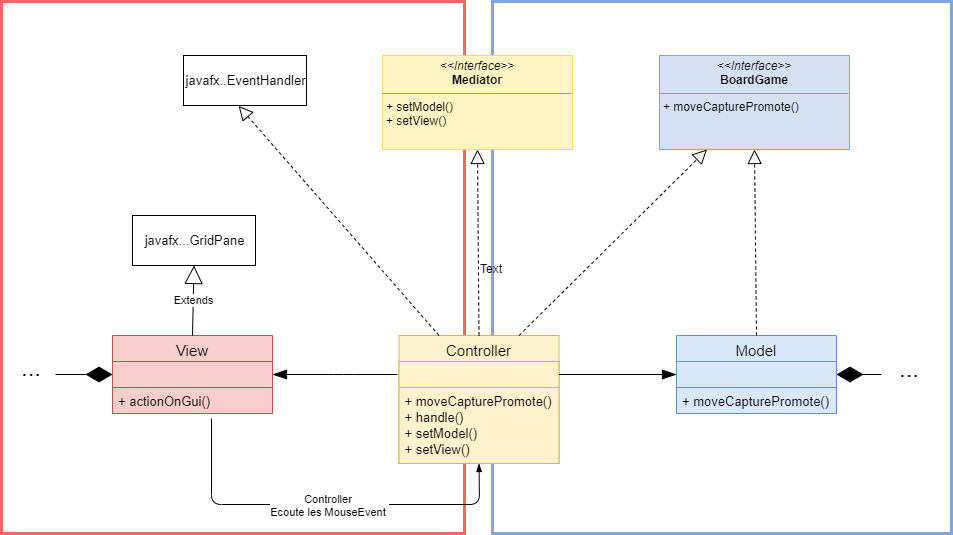
* Le model (bleu) qui contient les classes métiers (gestion des règles du jeu et stockage des données),
* La view (rose) qui s’occupe de l’affichage du damier et des pièces sur le damier,
* Le controller (jaune) qui :
  1. Écoute les interactions de l’utilisateur sur la view (clic de souris)
  2. Interroge le model pour savoir si l’action est légale
  3. Communique la réponse à la vue pour qu’elle rafraichisse l’affichage

Vous pouvez voir sur la Figure N°1 que chaque package contient une classe principale qui sert de façade au package (Design Pattern Facade : une façade sert de point d’entrée à un sous-système et masque les autres classes ce qui simplifie ainsi les échanges entre objets).

Vous pouvez également constater que la classe View ne communique pas directement avec la classe Model mais communique indirectement avec elle à travers la classe Controller. Le Controller se substitue au Model vis-à-vis de la View (ils implémentent tous les 2 la même interface BoardGame). En mode local ce n’était pas indispensable, en revanche en mode Client/Server c’est pratique…

Le Controller est un médiateur (DP Mediator) entre la classe Model et la classe View qui ne se connaissent pas entre elles. Il invoque leurs méthodes respectives et doit donc les connaitre.

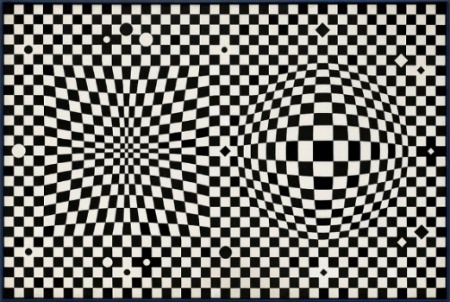
La View et composée (Cf. diagramme complet en annexe) d’un damier (objet instance de la classe GridView) sur lequel sont dessinées des cases blanches ou noires (objets SquareGui) sur lesquelles sont éventuellement posées des pièces noires ou blanches (PieceGui). Contrairement à ce que laisse supposer la Figure N°1, le Controller écoute les clics de souris sur les SquareGui et sur les PieceGui (et non pas sur la View) et pour cela implémente l’interface EventHandler.



**Figure N°1 : Diagramme de classe simplifié**

Nous détaillerons chaque package au fur et à mesure des ateliers.

Toutes les interfaces et classes nécessaires à l’exécution du programme en mode local (sauf 1…) vous sont fournies. Certaines méthodes sont complètes (en particulier toutes celles qui concernent l’affichage graphique) et beaucoup sont à compléter. Certaines autres seront à identifier et à ajouter en fonction des besoins.



V. VASARELY – Vega III (1955)

# Atelier N°1

## Compétences acquises à travers l’atelier (être capable de…)

* S’approprier une petite partie de l’architecture et du code de l’application existante qui servira de base à votre futur jeu de dames.
* S’adapter au mode de pensée d’un autre concepteur/développeur.
* Enrichir une application en respectant sa conception.
* Mettre en œuvre les principes de responsabilité unique et de délégation.
* Tester unitairement une fonctionnalité et intégralement une partie de l’application.

## Connaissances acquises à travers l’atelier

* Piliers de la Programmation Orientée Objet : abstraction, encapsulation.
* Classe Object de Java.
* Interface Comparable de Java

## Déroulement de l’atelier

Dans cet atelier, nous allons commencer à travailler sur notre projet Fil-Rouge. En particulier, nous allons programmer une version simple de la logique « métier » :

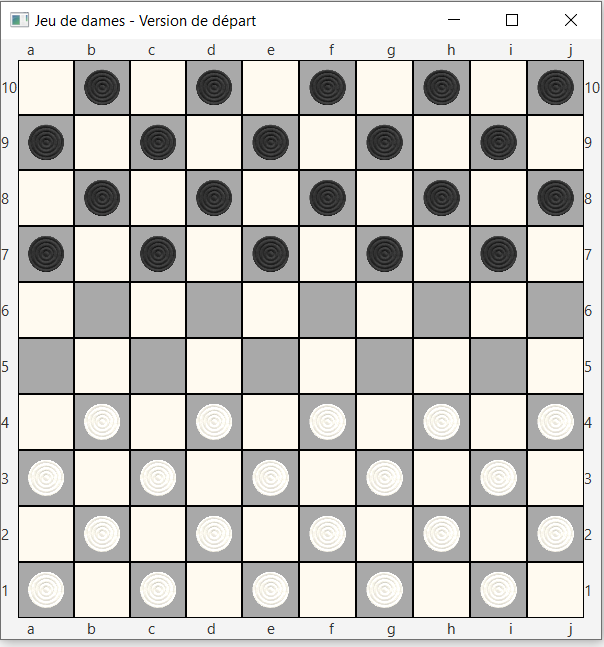
* Le déplacement n’est autorisé que si :
  + la pièce existe,
  + elle appartient au joueur dont c’est le tour de jeu,
  + la case de destination est dans les limites du damier.
* Les pièces peuvent se déplacer d’1 case en diagonale vers le haut ou vers le bas selon leur couleur.
* Toutes les autres fonctionnalités seront implémentées ultérieurement (La prise des pièces de l’adversaire, la promotion des pions en dames, le comportement des dames, la prise successive de pièces par le même joueur (rafle), ou encore la vérification qu’il s’agit du coup le plus rentable pour autoriser le déplacement ou la possibilité de jouer sur 2 ordinateurs différents (pour les plus rapides)).

### Étude préliminaire : tutos à lire avant le 1er atelier

1. Vidéo Youtube de Xavier Blanc « [Programmation Orientée Objet - Cours 1](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10911) [»](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10911) (34’)
2. Vidéo Youtube de Xavier Blanc [« Programmation Orientée Objet - Cours 3 : Interfaces »](https://prod.e-campus.cpe.fr/mod/page/view.php?id=10914) (les 6 premières minutes - entendre "client/application cliente" à la place "d'utilisateur".
3. Vidéo Youtube de Xavier Blanc « Programmation Orientée Objet - Cours 2 : Conception (Encapsulation et Responsabilités) » (10’).

### 1ère activité : installation des sources et 1er test

1. **Créez un projet** dans votre IDE à partir des sources du projet Fil-Rouge disponibles sur le e-campus (respectez bien les packages) et exécutez le programme avec le launcher. Normalement, un damier composé de cases noires (grises dans l’exemple) ou blanches sur lequel sont posées des pièces noires ou blanches doit s’afficher. Il est impossible de déplacer une pièce. Une trace sans pièce doit s’afficher également en mode console.



##### Aucune importance si tout l’environnement JavaFx n’est pas installé (sera utile pour l’atelier N°2) et ne permet pas l’exécution à partir du launcher, nous allons travailler sur les classes du model avec un programme de test unitaire de chaque méthode qui s’exécute en console (TestModel). **Lancez l’exécution de ce client**. Vous devez obtenir la trace d’exécution suivante :

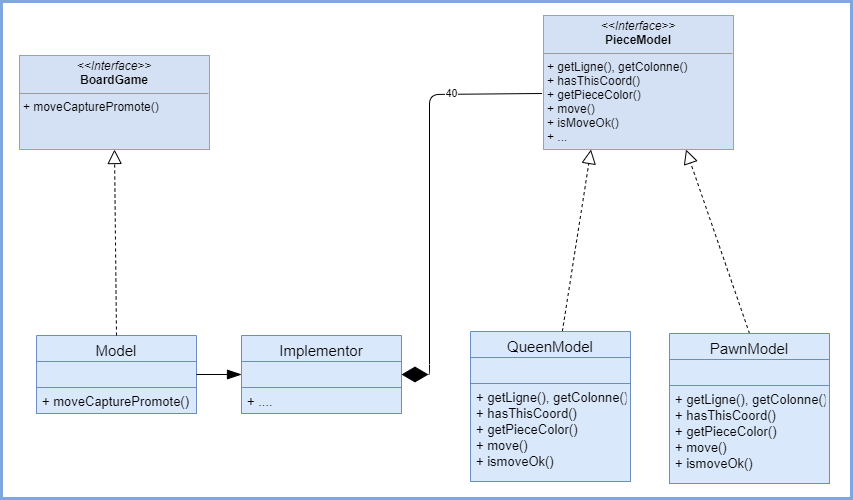
Test classe Coord

MAX = 10

c1 = [7,a]

c2 = [3,b]

### 2ème activité : étude de la conception du model



**Figure N° 2 : Diagramme de classes du model au démarrage du projet**

L’analyse du problème a permis d’identifier 3 objets métiers :

* Le jeu de dames (qui gère les règles du jeu),
* Le plateau de jeu sur lequel évoluent les pièces,
* Les pièces qui sont soit des pions soit des dames. Les pièces sont noires ou blanches.

La conception a conçu 6 classes/interfaces pour modéliser ce problème (Cf. Figure N°2) :

* La classe Model (et l’interface BoardGame) : l’objet instance de cette classe est responsable de la logique métier (règles du jeu). En revanche, il ne stocke pas les pièces, ne connait pas leurs types, n’invoque aucune de leurs méthodes directement mais passe par un objet Implementor pour le faire.
* La classe Implementor : l’objet instance de cette classe est responsable de fabriquer et de stocker les pièces, de les retrouver, de leur dire de se déplacer, etc. Il délègue à une fabrique (ModelFactory) le soin de lui créer une liste de PieceModel (blanches et noires).
* Les objets PieceModel (c’est l’interface qui définit le comportement attendu de chaque pièce) :
  + Sont soit des instances de PawnModel (pion) soit des instances de QueenModel (dame). Au début de la partie, ce sont tous des PawnModel.
  + Ont tous une couleur (de type PieceSquareColor) et une Coordonnée (de type Coord) comprise entre [‘a’..‘j’][10..1], le point [‘a’, 10] étant en haut à gauche du damier.
  + Sont intelligents car, connaissant leur position, ils savent dire s’ils peuvent se déplacer à une coordonnée de destination donnée en fonction de leur algo de déplacement (en diagonale, vers le haut ou le bas selon leur couleur, etc.)

Le diagramme de classes (Figure N°2), les commentaires de chaque interface/classe/méthode dans le code, et le programme de test unitaires doivent vous permettre de bien identifier/comprendre les responsabilités de chaque classe et du coup les mécanismes de délégation. Etudiez en particulier la méthode moveCapturePromote() de la classe Model.

### 3ème activité : développement des classes du model

** Consignes générales**

* Le programme de test est à dé-commenter au fur et à mesure des exercices. Le résultat à obtenir est indiqué en commentaire à la fin de chaque ligne
* Les méthodes sont à compléter au fur et à mesure. Certaines méthodes inexistantes seront à ajouter.
* Testez chaque ligne de test avant de passer à la suivante.
* Complétez le Carnet de Bord au fur et à mesure.

##### **Développez et TESTEZ** les méthodes de la classe Coord.

**Quelques éléments fonctionnels**

* + - Les coordonnées sont valides sous réserve d’être dans les limites du tableau (damier 10\*10).

**Trucs et astuces**

* + - Pour comprendre pourquoi l’instruction « c1.equals(new Coord('a', 7) » retourne « false » alors que les objets semblent égaux (attributs ont les mêmes valeurs), consultez la Javadoc de la classe Object et ajoutez dans la classe Coord les méthodes qui s’imposent.
    - Pour comparer 2 objets d’une même classe, celle-ci doit implémenter l’interface Comparable et la méthode compareTo() (lire Javadoc Comparable).

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - Faites générer autant de code que possible par votre IDE.

##### **Développez et TESTEZ** les méthodes de la classe PawnModel.

**Quelques éléments fonctionnels**

* + - Les PieceModel ont des coordonnées initiales qui seront changées en cas de déplacement.
    - Le déplacement est « légal » quand la pièce se déplace, par rapport à sa position initiale, d’1 case en diagonale vers le haut ou vers le bas selon la couleur ou de 2 cases en diagonale en cas de prise (dans n’importe quel sens).

**Trucs et astuces**

* + - Les fonctions abs et signum de la classe Math peuvent être utiles (Cf. Javadoc).

##### **Développez et TESTEZ** les méthodes de la classe ModelImplementor.

**Quelques éléments fonctionnels**

* + - Cet objet crée et stocke les PieceModel (par délégation à la fabrique).
    - Il invoque leurs méthodes pour leur dire de se déplacer ou pour indiquer au Model que le déplacement est « légal ».

**Trucs et astuces**

* + - Une LinkedList :
      * Est une Collection (objet qui contient plusieurs objets du même type)
      * Et plus précisément, est une List (objet munie d’une relation d’ordre dans lequel la recherche du ième élément a un sens, qui peut être parcouru en séquentiel et qui admet les doublons),
      * Implémentée dans une liste chainée.
      * Un élément est ajouté dans la liste grâce à la méthode add() (Cf. Javadoc).
      * Une collection est parcourue ainsi (Cf. exemple dans ModelFactory) :

For (TypeElementsDeMaCollection refVersUnElement : maCollection) {

// Manipulation de refVersUnElement ;

}

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - La méthode findPiece() doit servir dans toutes les autres méthodes.
    - Dé-commentez les lignes de la méthode toString() pour tester ensuite la class Model.

##### **Etudiez et TESTEZ** les méthodes de la classe Model.

** Consignes**

* + - Etudiez l’algorithme de la méthode moveCapturePromote() et testez. Toutes les méthodes sont écrites et se contentent d’invoquer les méthodes de l’objet ModelImplementor donc, si vous avez bien testé ce qui précède, cette partie devrait s’exécuter sans problème.

### 4ème activité : correction du Carnet de Bord

# Atelier N°2

## Compétences acquises à travers l’atelier (être capable de…)

* S’approprier l’architecture, la conception, le code d’une application dont le développement n’est que partiel.
* S’adapter au mode de pensée d’un autre concepteur/développeur.
* Identifier en quoi l’architecture et la conception proposées facilitent l’écriture, les tests et la future maintenance.
* Dans de futures applications, définir les responsabilités/données de chaque packages/classes en garantissant une forte cohésion et un faible couplage pour que la modification d’une couche n’ait pas d’impact sur les autres.

## Connaissances acquises à travers l’atelier

* Piliers de la Programmation Orientée Objet : abstraction, encapsulation, héritage, polymorphisme.
* Bases du Framework JavaFx et de la programmation événementielle.
* Mise en œuvre simple d’une architecture MVC (Model View Controller).

## Déroulement de l’atelier

Dans cet atelier, nous allons travailler sur les 3 couches du MVC et ainsi :

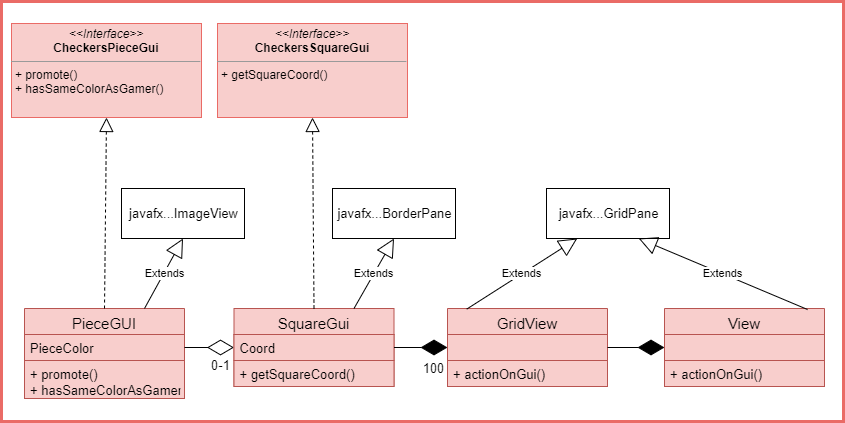
* Etudier rapidement le fonctionnement de la view puis faire évoluer la fabrique des éléments constitutifs de la view pour qu’ils puissent être manipulés par le controller comme des objets fonctionnels et non pas comme des objets graphiques (ce sera plus clair quand vous le programmerez ;-)).
* Connecter le model et la view pour que la logique métier soit prise en compte dans les déplacements effectués par l’utilisateur. Cette connexion s’effectue dans le controller.
* Enrichir le model en programmant la prise d’1 seule pièce de l’adversaire (pas de rafle de plusieurs pièces de l’adversaire pour l’instant).

Toutes les instructions relatives à la programmation graphique et à la programmation événementielles sont déjà écrites, il vous suffira de les comprendre et d’en déplacer certaines.

### Étude préliminaire : vidéos à visionner avant le 2ème atelier

1. Vidéo Youtube de Xavier Blanc « [Programmation Orientée Objet - Cours 3 : Héritage](https://www.youtube.com/watch?v=bLge8v-czPg&t=4s&ab_channel=xavierblanc)» (18’)
2. Vidéo Youtube de Xavier Blanc « [Programmation Orientée Objet - Cours 4 : IHM et MVC](https://www.youtube.com/watch?v=6ZN3-pKU8e4&list=PLuNTRFkYD3u5ibkjD1nHP9QZXPjtvnPXL&index=8&ab_channel=xavierblanc)» (13’). Ne retenez pas les classes et méthodes du package swing, nous allons utiliser JavaFx.
3. Vidéo Youtube de Bucky Roberts (Thenewboston) « [JavaFX Java GUI Tutorial - 1 - Creating a Basic Window](https://prod.e-campus.cpe.fr/mod/url/view.php?id=11809) » (11’)
4. Vidéo Youtube de Bucky Roberts (Thenewboston) « [JavaFX Java GUI Tutorial - 2 - Handle User Events](https://prod.e-campus.cpe.fr/mod/url/view.php?id=11810" \o "JavaFX Java GUI Tutorial - 2 - Handle User Events) » (6’)

### 1ère activité : étude de la conception de la view



**Figure N°3 : Diagramme de classes de la view**

L’analyse du problème a permis d’identifier 4 objets fonctionnels :

* Une fenêtre d’affichage du damier et des axes,
* Le damier sur lequel évoluent les images de pièces,
* Le damier est composé de cases sur lesquels sont éventuellement affiché une image de pièce. Une case connait sa couleur et sait où elle se trouve sur le damier (coordonnées variant de [0..99], la case à l’indice 0 étant celle en haut à gauche du damier)
* Une pièce est représentée par une image, connait également sa couleur. Son image peut changer en cas de promotion (lorsqu’elle correspond à une dame du Model).

La conception a conçu 6 classes/interfaces pour modéliser ce problème (Cf. Figure N°3) :

* La classe View : l’objet instance de cette classe est responsable de l’affichage du jeu. Cet objet ne fait cependant pas grand-chose car il délègue au damier (Board ) le soin de créer les cases (SquareGui) et les pièces (PieceGui) et lui transfère les « ordres » venant du controller pour qu’il rafraichisse l’affichage (image de pièce qui se déplace ou qui disparait en cas de prise).
* La classe Board : l’objet instance de cette classe est responsable de fabriquer et de stocker les cases et les images de pièces puis de déplacer/supprimer ces images. En réalité, il délègue à une fabrique (GuiFactory) le soin de créer les SquareGui (blanches et noires) au bon endroit et d’initialiser le positionnement des PieceGui. Pour que l’affichage se fasse en ligne/colonne, cet objet est une extension de GridPane (classe JavaFx).
* Les cases du damier sont des BorderPane (JavaFx) qui accueilleront potentiellement une ImageView (JavaFx). Pour autant, fonctionnellement, ce sont respectivement des CheckersSquareGui et des CheckersPieceGui qui seront manipulés par le Controller. Les classes SquareGui et PieceGui seront à compléter/utiliser dans l’activité N°2.

Le diagramme de classes (Figure N°3), les commentaires de chaque interface/classe/méthode dans le code, doivent vous permettre de bien identifier/comprendre les responsabilités de chaque classe et du coup les mécanismes de délégation.

### 2ème activité : évolution de la fabrique d’objets graphiques

##### **Créez un projet** dans votre IDE à partir de la correction de l’atelier 1, et testez l’application. Le cas échéant, incluez JavaFx dans le Build path si le programme ne compile/s’exécute pas.

##### La fabrique (GuiFactory) a la double responsabilité de créer des objets et de dessiner ces objets. **Rendez-lui sa vraie vocation** (qui est de créer des objets).

** Consignes**

* + - Complétez les classes SquareGui et PieceGui, et répartissez le code approprié venant de la fabrique dans ces 2 classes (dans leur constructeurs respectifs). Modifiez le code des méthodes createSquare() et createPiece() de la GuiFactory en conséquence et testez. A priori, cela doit fonctionner comme précédemment (affichage du damier sans possibilité de déplacer les pièces)
    - Amusez-vous à changer la couleur des cases dans la classe GuiConfig et testez.

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

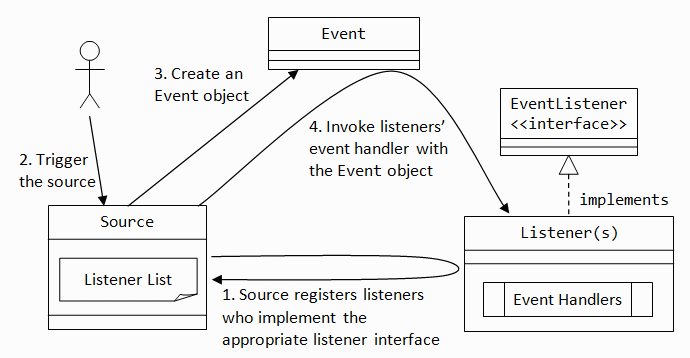
* + - En aucun cas vous ne devez modifier d’autre classes que la GuiFactory et les classes SquareGui et PieceGui.
    - Attention, la classe GuiFactory reste responsable de créer les objets (new).

### 3ème activité : connexion du model et de la view à travers le controller

##### **Intégrez la logique métier** à notre application.

**Quelques éléments fonctionnels**

* + - L’objet Controller en tant que EventHandler écoute les événement souris (clic) sur les SquareGui et PieceGui.
    - Il doit au préalable s’enregistrer en tant qu’écouteur auprès de chacun.
    - Lorsque l’utilisateur clique sur un SquareGui ou une PieceGui, cela crée un événement qui va être géré par l’écouteur.
    - Lorsqu’il gère un évènement (clic), il interroge le Model et renvoie l’information à la View.



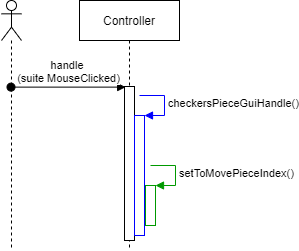
Source : www3.ntu.edu.sg/home/ehchua/programming/java/J4a\_GUI.html

**Figure N°4 : Gestion des évènements**

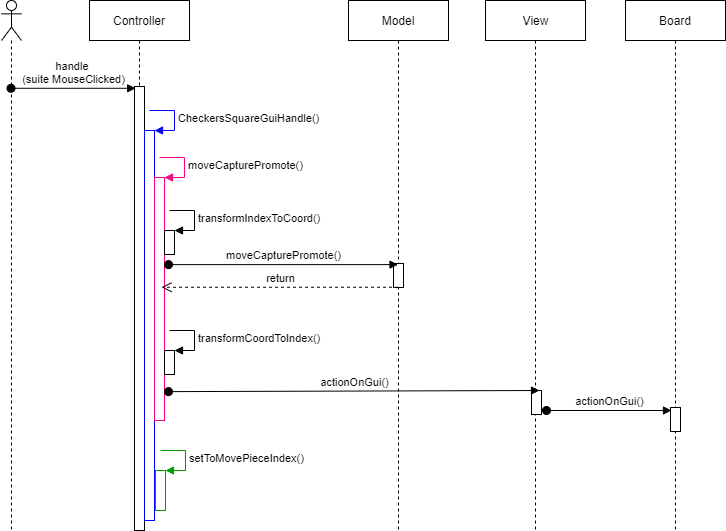
** Consignes**

* + - Etudiez les diagrammes de séquence ci-dessous (figures 5 et 6) et repérez dans les classes les méthodes invoqués (sur les Figures N°2 et 7 et dans le code). Etudiez les valeurs passées en paramètre et retournées par les méthodes (types définis dans package controller).
    - Pour l’instant, seule la gestion des évènements est traitée. Complétez la méthode moveCapturePromote() de la classe Controller selon les indications du diag. de séquence.

**Trucs et astuces**



**Figure N°5 : Diagramme de séquence des échanges lors d’un clic de l’utilisateur sur une PieceGui**



**Figure N°6 : Diagramme de séquence des échanges lors d’un clic de l’utilisateur sur un SquareGui**

##### **Complétez** le Carnet de Bord.

### 4ème activité : évolution du model – gestion des prises

##### **Complétez** les classes du model pour prendre en compte la **capture** d’1 pièce de l’adversaire (1 seule, pas de rafle).

**Quelques éléments fonctionnels**

* + - La prise est « légale », s’il n’y a qu’1 seule pièce d’une autre couleur sur le trajet.
    - La capture (métier) d’une pièce suppose la suppression de la PieceModel correspondante dans la collection gérée par le ModelImplementor.
    - En cas de prise, les coordonnées de la pièces supprimée doivent être retournées au client de la méthode moveCapturePromote() (à travers l’objet d’échange de type OutputModelData).

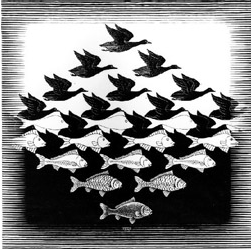
**Trucs et astuces**

* + - Commencez par programmer les méthodes getCoordsOnItinerary() des classes PawnModel puis ModelImplementor qui retournent les coordonnées des cases traversées.
    - La liste de Coord retournée est exploitée par les méthodes isThereMaxOnePieceOnItinerary() et getToCapturePieceCoord() qui retournent respectivement true s’il existe au maximum 1 pièce à prendre sur la trajectoire et les coordonnées de l’éventuelle pièce à prendre.
    - Gardez en tête que dans le prochain atelier vous programmerez le comportement des dames (QueenModel) qui peuvent se déplacer de plusieurs cases à la fois à condition qu’il y ait au plus 1 pièce sur la trajectoire entre leur position de départ et d’arrivée. Pour autant, il ne devra pas y avoir d’impact sur les méthodes modifiées aujourd’hui.

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - Vous pouvez tester unitairement vos méthodes en réutilisant le programme de tests du model (atelier 1) et en le complétant si besoin.
    - Les méthodes isThereMaxOnePieceOnItinerary() et getToCapturePieceCoord() ont beaucoup d’instructions communes. Vous pouvez éventuellement tenter de les factoriser.

### 5ème activité : correction du Carnet de Bord



M.C. ESCHER - Ciel et eau I (1938)

# Atelier N°3

## Compétences acquises à travers l’atelier (être capable de…)

* Enrichir une application Java en respectant les Principes de Responsabilité Unique (SRP), d’Ouverture-Fermeture (OCP) et de Substitution de Liskov (LSP).
* Redéfinir des comportements existants lorsque c’est nécessaire (méthodes polymorphes).
* Choisir l’organisation et le stockage des données appropriés en fonction des besoins et du contexte.
* Identifier les responsabilités de chaque interface/classe et répartir les données et traitements en conséquence, tout en garantissant un couplage faible et une forte cohésion.
* Détecter des dysfonctionnements et les traiter.

## Connaissances acquises à travers l’atelier

* Piliers de la Programmation Orientée Objet : toujours abstraction, encapsulation et d’avantage héritage, polymorphisme.
* Framework de Collections en Java :  interfaces Collection, List, Set, Queue, Map, et leurs implémentations.
* Interfaces Comparable et Comparator. Classe Collections.

## Déroulement de l’atelier

Dans cet atelier, nous allons compléter notre projet Fil-Rouge afin de :

* Enrichir le model en programmant la gestion des dames (comportement des dames et promotion des pions en dames),
* Promouvoir les pions en dames sur la view (visuellement).
* Mais surtout, identifier les types de collection appropriées à notre problématique (nous avons testé avec une LinkedList, nous allons vérifier si d’autre auraient été tout aussi pertinentes).

### Étude préliminaire : tutos à lire avant le 3ème atelier

##### T[uto OpenClassRoom «  Débutez la programmation avec Java - Partie 2 - Gérez les piles de données avec la bonne collection »](https://openclassrooms.com/fr/courses/6173501-debutez-la-programmation-avec-java/6458461-gerez-les-piles-de-donnees-avec-la-bonne-collection)

### 1ère activité : gestion du comportement des dames

##### **Créez un projet** dans votre IDE à partir de la correction de l’atelier 2 et testez l’application.

##### **Complétez** les classes du model pour gérer le **comportement des dames**.

**Quelques éléments fonctionnels**

* + - Les dames peuvent se déplacer en diagonale, de plusieurs cases à la fois, dans n’importe quel sens à condition qu’il y ait au plus 1 pièce sur leur trajectoire.

**Trucs et astuces**

* + - A priori, seule la classe QueenModel devrait être modifiée.
    - Pour autant, vous allez constater quelques redondances avec la classe PawnModel. Profitez-en pour factoriser tous les éléments communs dans une classe abstraite AbstractPieceModel (Cf. Figure N°7).

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - Vous pouvez tester en modifiant la fabrique en décidant temporairement que les pièces blanches sont des dames.

##### **Complétez** les classes du model pour gérer la promotion des pions en dame :

**Quelques éléments fonctionnels**

* + - Lorsqu’un pion blanc arrive sur la ligne du haut (10) ou un pion noir arrive sur la ligne du bas (1), il est promu en dame.

**Trucs et astuces**

* + - Vous devez modifier ou ajouter des méthodes dans plusieurs classes du model. **Réfléchissez bien aux responsabilités** de chacune en misant sur l’encapsulation et la délégation.
    - Il existe une interface Promotable munie d’une méthode isPromotable() et d’une méthode promote() : **faites** que la classe PawnModel implémente cette interface.
    - On ne sait pas changer l’état/la nature d’un PawnModel (sauf à mettre en œuvre le Design Pattern Strategy – à tester pour ceux qui veulent). Donc le ModelImplementor va le supprimer de sa liste de PieceModel et créer un QueenModel aux mêmes coordonnées ; de fait, dans ce contexte, la méthode promote() de la classe PawnModel ne fait rien.
    - L’opérateur instanceof permet de tester si un objet est une instance d’une classe ou d’une interface : if (piece instanceof Promotable)…

**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - Pour l’instant, seul le changement de comportement est visible, vous n’avez pas encore changé le visuel (testez sur des pions noirs, les blancs sont déjà des dames…).

##### Propagez la promotion dans la view :

**Trucs et astuces**

* + - Vérifiez comment cette promotion est effectuée dans la classe Board et complétez la méthode PromotePiece de la classe GuiFactory.
    - Changez le visuel de la pièce (couleur, taille, image, etc.)

### 2ème activité : études du Framework de collections Java

** Consignes générales**

* Il s’agit de tester d’autres types d’implémentation des collections, de trier celles qui peuvent l’être selon différents critères et de vérifier si elles auraient été tout aussi pertinentes dans notre jeu.
* Un programme de test à dé-commenter et tester au fur et à mesure sert de conducteur à cette étude mais il faudra aussi aller tester si les changements d’implémentation ont un impact sur l’exécution du jeu de dames.
* Des méthodes/classes inexistantes seront à ajouter dans les classes du model. D’autres seront à compléter dans le programme de test (TestCollections)

##### **Testez** quelques fonctionnalités possibles sur les **List**

L’interface List précise toutes les méthodes qui permettent à une liste de gérer ses éléments. Il existe une classe Collection**s** qui propose des algorithmes sur les listes (tri, min/max, recherche dichotomique, etc.).

1. Etudiez et exécutez le pgm TestCollection. La liste s’affiche grâce à la méthode getRender() dans l’ordre d’insertion des pièces en affichant 5 pièces par ligne.
2. Modifiez la méthode getRender() pour parcourir la liste des PieceModel en utilisant un itérateur :

**Quelques éléments fonctionnels**

L’affichage doit se présenter sous cette forme :

[W[4,b]] [W[4,d]] [W[4,f]] [W[4,h]] [W[4,j]]

[W[3,a]] [W[3,c]] [W[3,e]] [W[3,g]] [W[3,i]]

[W[2,b]] [W[2,d]] [W[2,f]] [W[2,h]] [W[2,j]]

[W[1,a]] [W[1,c]] [W[1,e]] [W[1,g]] [W[1,i]]

[B[10,b]] [B[10,d]] [B[10,f]] [B[10,h]] [B[10,j]]

[B[9,a]] [B[9,c]] [B[9,e]] [B[9,g]] [B[9,i]]

[B[8,b]] [B[8,d]] [B[8,f]] [B[8,h]] [B[8,j]]

[B[7,a]] [B[7,c]] [B[7,e]] [B[7,g]] [B[7,i]]

**Trucs et astuces**

* + - Un itérateur est un objet qui implémente l’interface java.util.Iterator.
    - Une Collection dispose d’une méthode qui permet d’utiliser son itérateur : pieces.iterator()

1. Affichez la liste dans **l’ordre croissant** des cases dans le damier (de 0 à 99) :

**Quelques éléments fonctionnels**

L’affichage doit se présenter sous cette forme :

[B[10,b]] [B[10,d]] [B[10,f]] [B[10,h]] [B[10,j]]

[B[9,a]] [B[9,c]] [B[9,e]] [B[9,g]] [B[9,i]]

[B[8,b]] [B[8,d]] [B[8,f]] [B[8,h]] [B[8,j]]

[B[7,a]] [B[7,c]] [B[7,e]] [B[7,g]] [B[7,i]]

[W[4,b]] [W[4,d]] [W[4,f]] [W[4,h]] [W[4,j]]

[W[3,a]] [W[3,c]] [W[3,e]] [W[3,g]] [W[3,i]]

[W[2,b]] [W[2,d]] [W[2,f]] [W[2,h]] [W[2,j]]

[W[1,a]] [W[1,c]] [W[1,e]] [W[1,g]] [W[1,i]]

**Trucs et astuces**

* + - La fonction sort() de la classe Collection**s** permet de trier une List. Elle s’appuie sur l’ordre naturel des éléments ou sur un comparateur.
    - Pour mémoire, l’ordre naturel est induit par la méthode compareTo() de l’interface java.util.Comparable. Il s’agit donc de rendre les PieceModel comparables.

** Consignes**

* + - Complétez la classe AbstractPieceModel en ce sens (Pensez délégation…).

1. Affichez la liste dans un autre ordre :

**Quelques éléments fonctionnels**

L’affichage doit se présenter sous cette forme :

[W[1,a]] [W[3,a]] [B[7,a]] [B[9,a]] [W[2,b]]

[W[4,b]] [B[8,b]] [B[10,b]] [W[1,c]] [W[3,c]]

[B[7,c]] [B[9,c]] [W[2,d]] [W[4,d]] [B[8,d]]

[B[10,d]] [W[1,e]] [W[3,e]] [B[7,e]] [B[9,e]]

[W[2,f]] [W[4,f]] [B[8,f]] [B[10,f]] [W[1,g]]

[W[3,g]] [B[7,g]] [B[9,g]] [W[2,h]] [W[4,h]]

[B[8,h]] [B[10,h]] [W[1,i]] [W[3,i]] [B[7,i]]

[B[9,i]] [W[2,j]] [W[4,j]] [B[8,j]] [B[10,j]]

** Consignes**

* + - **Triez** la collection avec un **nouveau comparateur** : d’abord sur les colonnes puis sur les lignes de manière croissante
    - Vous pouvez créer ce comparateur en :
    - Créant explicitement une nouvelle classe pour gérer ce comparateur (classe qui implémente l’interface java.util.Comparator)…
    - Créant la classe de manière anonyme directement dans les paramètres de la fonction sort().

##### **Vérifiez** si le type **ArrayList** aurait été approprié pour gérer les pièces d’un jeu de dames.

1. Continuez de dé-commenter les instructions du pgm de test pour vérifier que l’utilisation d’une ArrayList vous donne exactement les mêmes résultats (affichage et tris).
2. **Testez** l’application Jeu de Dames en activant la ligne qui construit la collection avec une ArrayList dans la méthode createPieceModelCollection() de la classe ModelFactory. Le fonctionnement devrait être le même qu’avec une LinkedList.

##### **Vérifiez** si le type **HashSet** aurait été approprié pour gérer les pièces d’un jeu de dames.

1. Continuez de dé-commenter les instructions du pgm de test pour vérifier utiliser un HashSet.

**Quelques éléments fonctionnels**

L’affichage doit se présenter sous cette forme (avant corrections) - l’ordre d’affichage est différent ce qui est logique puisque les éléments sont rangés en fonction de leur hashCode:

[B[8,d]] [B[8,j]] [W[3,c]] [W[4,j]] [W[2,f]]

[B[10,b]] [B[9,i]] [B[10,h]] [B[9,a]] [B[9,e]]

[B[7,a]] [W[2,d]] [B[8,h]] [W[2,h]] [W[4,b]]

[W[4,d]] [W[1,i]] [B[7,c]] [W[3,a]] [B[7,g]]

[W[1,g]] [B[8,b]] [W[2,j]] [W[3,g]] [W[1,e]]

[W[1,a]] [W[4,h]] [W[2,b]] [B[9,g]] [W[1,c]]

[B[10,f]] [B[8,f]] [B[10,d]] [W[4,f]] [W[3,i]]

[B[7,e]] [W[3,e]] [B[10,j]] [B[7,i]] [B[9,c]]

** Consignes**

* + - Testez dans un 1er temps sans rien changer au reste du pgm : vous obtenez le résultat ci-dessus.
    - Dans la méthode fillCollection(), dé-commentez les dernières lignes de cette méthode pour ajouter à nouveau les pions noirs et testez. Vous devriez voir 60 pièces au lieu de 40 dont toutes les pièces noires en double. C’est ennuyeux, car un Set ne peut/doit pas avoir de doublon par définition. Pourquoi la méthode add() de la classe HashSet donne-t-elle l’impression de ne pas avoir fait son job ? Tout simplement parce que 2 objets égaux doivent avoir le même hashCode et que les méthodes equals() et hashCode() n’ont pas été redéfinies pour les PieceModel. **Complétez** la classe AbstractPieceModel en ce sens et **vérifiez** qu’il n’y a que 40 pièces (L’affichage devrait changer par rapport à la version sans surcharge des méthodes)
    - **Essayez** de trier votre HashSet et confirmez que c’est impossible.

1. **Testez** l’application Jeu de Dames en déplaçant et prenant plusieurs pièces. Normalement, au bout d’un certain temps, cela ne devrait plus fonctionner. Que se passe-il ?

**Trucs et astuces**

* + - Pour comprendre pourquoi l’utilisation d’un HashSet est incompatible avec les algos de notre jeu de dames interrogez-vous sur la façon dont les éléments sont stockés dans un HashSet.

##### **Vérifiez** si le type **TreeSet** aurait été approprié pour gérer les pièces d’un jeu de dames. A priori, pas plus que le HashSet pour la même raison, mais il s’agit juste ici d’en vérifier la faisabilité.

1. Continuez de dé-commenter les instructions du pgm de test pour utiliser un TreeSet.
   * + Lorsque vous le construisez sans paramètre, l’affichage est le même qu’à la question 1.c.
     + Lorsque vous le construisez avec paramètre, l’affichage est le même qu’à la question 1.d.
2. Inutile de tester le Jeu de Dames avec cette implémentation.

##### **Complétez** le Carnet de Bord.

### 3ème activité : correction du Carnet de Bord



S. DALI - La persistance de la mémoire (1931)

# Atelier N° 4 (pour les plus rapides)

## Compétences acquises à travers l’atelier (être capable de…)

* Faire évoluer l’architecture d’une application Java tout en respectant la conception initiale et en garantissant la qualité du programme (respect principes SOLID).
* Concevoir les tests avant de coder.
* Evaluer la conception et son impact sur la facilité de maintenance, d’extension et de réutilisation.

## Déroulement de l’atelier

### 1ère activité : évolution des fonctionnalités ou bien évolution de l’architecture

Il va s’agir sur notre projet Fil-Rouge de :

* Finir de développer le jeu :
  + Coté métier :
    - Mettre en place les rafles c’est-à-dire offrir la possibilité / obliger un joueur à continuer de prendre des pièces (du joueur opposé) tant qu’il le peut.
    - Vérifier qu’il s’agit du coup le plus rentable avant d’autoriser le déplacement.
  + Coté métier/vue :
    - Afficher les scores,
    - Gérer la fin de partie.

** Consignes**

* + - Ne sacrifiez pas l’encapsulation sur l’autel de la facilité.

**Quelques éléments fonctionnels**

* + - Je vous laisse réfléchir...

Ou bien

* Faire évoluer l’architecture de manière à ce que chaque joueur soit sur un ordinateur différent (on commencera à simuler en mode local : « 127.0.0.1 »).

**Quelques éléments fonctionnels**

* + - 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 sur l’un des damiers, le déplacement doit être « visible » sur le damier de l’autre joueur.
    - La conception respectée jusqu’à maintenant fait qu’il n’y aura aucun changement à effectuer sur aucune classe métier, ni sur aucune classe de la vue.

** Consignes**

* + - Il « suffit » de créer de nouveaux controller qui au-delà d’invoquer les méthodes du model et de la view envoient et reçoivent des messages à travers un canal de communication (sockets).
    - On peut concevoir le mode Client/Server de 2 manières, 1 plus simple à mettre en place mais peu usuelle et 1 autre plus logique mais un peu moins simple (à votre choix) :

1ère configuration :

* 1 application coté Server (à lancer en 1er) et 1 application coté Client (à lancer après),
* Chacune ayant 1 instance de model, 1 instance de view, 1 instance de controller (attention, nouvelle classe à créer).
* Dans cette configuration, les messages envoyés à travers les sockets permettront de faire évoluer le model et la view de l’autre joueur.

2ème configuration :

* 1 application coté Server (à lancer en 1er) avec LE model et 1 controller qui sait « parler » au model (controller à créer).
* 2 applications clientes (1 pour le joueur blanc et 1 pour le joueur noir) avec chacune 1 instance de view et 1 instance de controller (à créer) qui sait parler à la view.

**Trucs et astuces**

* + - Dans les 2 cas, une classe abstraite permettant de factoriser les données et comportements des controller client et server sera sans doute nécessaire (gestion des sockets).
    - Dans tous les cas, des Thread et des Task permettront une communication non bloquante entre les applications.
    - Vous trouverez les cours d’introduction aux sockets, de broadcast de message en Java et de programmation concurrente en JavaFX sur les sites suivants.
* Introduction aux sockets : <http://openclassrooms.com/courses/introduction-aux-sockets-1>
* Broadcast de messages : <https://defaut.developpez.com/tutoriel/java/serveur/multithread/>
* Accès concurrents : <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/>

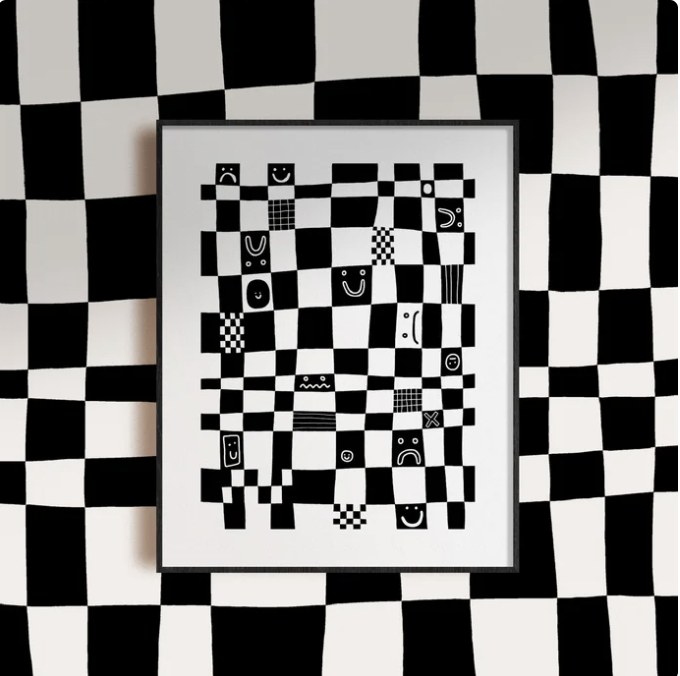
**Notes Post-it jaune vecteur libre avec punaise Clip Art Remarques**

* + - Aucune classe ni du model ni de la view ne doit être modifiée. Seuls des controller seront à créer.
    - Un package qui encapsule tout ce qui est relatif à la communication à travers des sockets est disponible sur le e-campus (reseau.zip dans les Fichiers de base).

### 2ème activité : déclusion

##### Correction Carnet de Bord et lecture « Et pour finir, tout cela était-il bien raisonnable ? ».

##### Auto-évaluation des connaissances et compétences acquises.



Marie CHANTEUR – Checkmate (2020)

## Figure N°7 : Diagramme de classe de l’application

