Université de Caen Basse-Normandie
U.F.R. de Sciences
Département d'informatique
Bâtiment Sciences 3 - Campus Côte de Nacre
F-14032 Caen Cédex, France

TD-TP $n^o 7$

Niveau	**L3**
Parcours	**Informatique**
Unité d'enseignement	**UE52 - Méthodes et modèles pour le logiciel [Conception et programmation par objets]**
Responsable	**Emmanuel Cagniot Emmanuel.Cagniot@ensicaen.fr**

1 Exercice

Dans un TD-TP précédent, nous avons écrit un modèle pour le jeu de la vie. La Figure 1 rappelle son diagramme de classes.

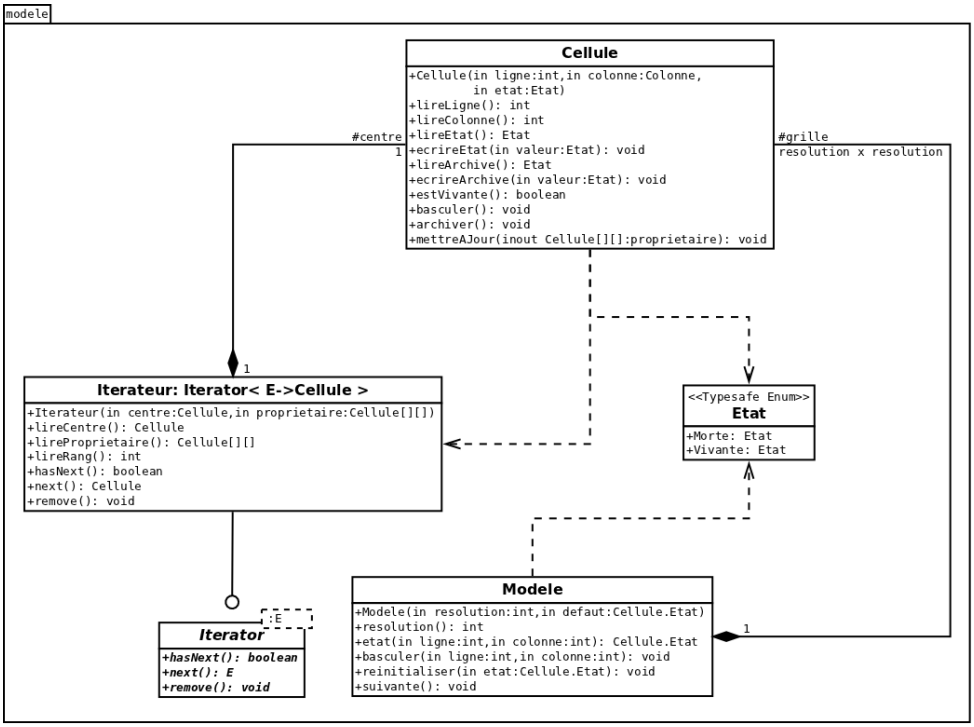

Fig. 1 – Diagramme de classes du modèle.

Nous souhaitons à présent écrire une interface graphique pour ce modèle. La Figure 2 présente le résultat à obtenir.

FIG. 2 – Interface graphique de l'application.

Cette interface associe un bouton à chaque cellule du modèle. Le fait de cliquer sur un bouton provoque le passage de la cellule correspondante d'un état à un autre. Les cellules mortes sont affichées en jaune tandis que les cellules vivantes le sont en rouge. Pour des considérations esthétique, les boutons représentant les cellules du modèle sont entourées d'une bordure de boutons invisibles et inertes laissant apparaître une mosaïque de fond.

La barre de menus et d'outils de l'interface propose quatre commandes :

1. une commande `Quitter` permettant de quitter proprement l'application ;
2. une commande `A propos` permettant d'afficher la version et les auteurs de l'application ;
3. une commande `Suivante` permettant de calculer puis d'afficher la génération de cellules suivante ;
4. une commande `Nouveau` permettant de réinitialiser la génération actuelle à partir de cellules mortes.

L'interface possède également une barre de défilement permettant de contrôler la résolution du jeu. Tout événement sur cette barre provoque l'instanciation puis l'affichage d'un nouveau modèle possédant la résolution demandée.

Une architecture de type MODEL-VIEW-CONTROLLER est particulièrement mal adaptée à ce type d'application graphique. La raison en est la suivante. Dans la plupart des applications graphiques modernes, la vue (c'est à dire l'interface graphique) draine une partie du contrôle puisque toute action de l'utilisateur (le client) sur ses widgets entraîne une modification du modèle et/ou de cette vue. Par conséquent, une redondance s'installe entre le contrôleur, pièce maîtresse du modèle MVC, et la vue qui est ici active et non plus passive. L'architecture MODEL-VIEW-PRESENTER, proposée en 1996 par Taligent Inc., une filiale d'IBM, prend acte de cette évolution. Dans cette nouvelle architecture, le contrôle de l'application est déplacé vers la vue tandis que le présentateur remplace le contrôleur du modèle MVC. Le rôle théorique du présentateur est de mettre en forme les données du modèle pour les rendre affichables par la vue. Plus pratiquement, le présentateur reçoit les requêtes émises par la vue et les répercutent sur le modèle. Comme pour les architectures MVC, les architectures MVP se déclinent en de nombreuses variantes, certaines autorisant la vue à dialoguer directement avec le modèle pour les opérations mineures.

Dans notre cas, nous allons implémenter une architecture MVP classique dans laquelle toutes les requêtes émises par la vue sont traitées par le présentateur. Par conséquent, le modèle est purement passif et ignore l'existence du présentateur et de la vue.

La vue sera réalisée en JAVA SWING. Les événements déclenchés par le client sur les widgets de cette bibliothèque peuvent être interceptés en implémentant le behavioral design pattern `Observer`. La Figure 3 présente son diagramme de classes.

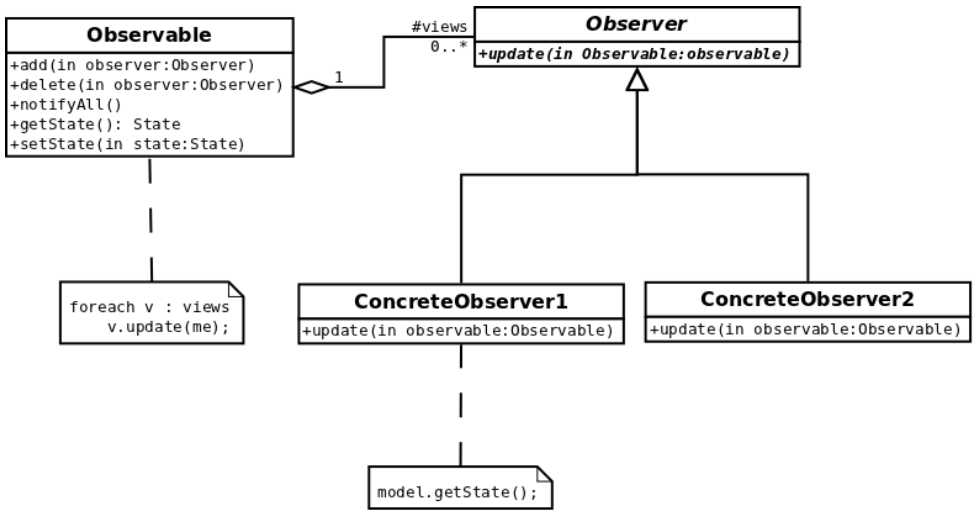

FIG. 3 – Diagramme de classes du behavioral design pattern `Observer`.

Dans ce diagramme, la classe `Observable` représente l'objet observé (un widget dans notre cas). Cet objet possède une liste (attribut `views`) d'observateurs. Les méthodes `add` et `delete` permettent respectivement d'ajouter et de supprimer des observateurs de cette liste. Les méthodes `getState` et `setState` sont des accesseurs permettant de modifier l'état de l'objet observé. Tout modification de cette état est notifiée aux différents observateurs de la liste par le biais de la méthode `notifyAll`.

La classe abstraite `Observer` représente un observateur. Sa méthode abstraite `update` est invoquée par la méthode `notifyAll` de la classe `Observable`. Les classes `ConcreteObserver1` et `ConcreteObserver2` représentent les observateurs physiques qui redéfinissent la méthode `update`. L'obtention du nouvel état de l'objet observé est réalisée en invoquant la méthode `getState` sur l'argument `observable`.

Dans SWING, le pattern `Observer` est implémenté via des interfaces Ainsi, `ActionListener` (Figure 4) est spécialisée dans l'interception des événements de types `ActionEvent`, ceux-ci pouvant être générés, par exemple, en cliquant sur des boutons. De même, `AdjustmentListener` est spécialisée dans l'interception des événements de type `AdjustmentEvent`, ceux-ci pouvant être générés, par exemple, en déplaçant le curseur d'une barre de défilement. Tout widget implémentant l'une de ces interfaces peut se voir ajouter un nouvel observateur via sa méthode `add`.

Les composants de la vue sont affectés au paquetage `vue`.

La Figure 5 présente le diagramme de classes relatif aux actions des barres de menus et d'outils. La classe abstraite `ActionAbstraite` dérive de la classe abstraite `AbstractAction` qui réalise l'interface `ActionListener` sans redéfinir sa méthode `actionPerformed`. Cette classe définit un attribut `vue` qui la relie à sa vue propriétaire. Cette façon de procéder évite la définition de multiples classes anonymes à l'intérieur de la classe `Vue` (Figure 8). Les classes `ActionQuitter` et `ActionAPropos` représentent des actions permettant respectivement de quitter l'application et de présenter cette dernière (nom, version et auteurs). Les classes `ActionSuivante` et `ActionNouveau` représentent des actions permettant respectivement de calculer la génération de cellules suivantes et de réinitialiser la génération actuelle à partir de cellules mortes.

La Figure 6 présente le diagramme de classes relatif à la représentation graphique du modèle. La classe `CelluleGraphique` représente une cellule du modèle. Elle définit trois attributs représentant respectivement le modèle graphique propriétaire et les numéros de ligne et de colonne de la cellule correspondante dans le modèle. Cette classe dérive de `JButton` et réalise l'interface `ActionListener` qui lui permet d'intercepter tout évènement provoqué par un clic sur le bouton. Par conséquent, toute instance de cette classe représente son propre listener. Sa méthode protégée `mettreAJour` est invoquée par la redéfinition

11/03/2012 ActionListener.java 1

```
package java.awt.event;

/**
 * The listener interface for receiving action events. The class that is
 * interested in processing an action event implements this interface, and the
 * object created with that class is registered with a component, using the
 * component's addActionListener method. When the action event occurs, that
 * object's actionPerformed method is invoked.
 */
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */
    void actionPerformed(ActionEvent e);

}
```

11/03/2012 AdjustmentListener.java 1

```
package java.awt.event;

/**
 * The listener interface for receiving adjustment events.
 */
public interface AdjustmentListener extends EventListener {

    /**
     * Invoked when the value of the adjustable has changed.
     */
    void adjustmentValueChanged(AdjustmentEvent e);

}
```

FIG. 4 – Interfaces `ActionListener` (à gauche) et `AdjustmentListener` (à droite).

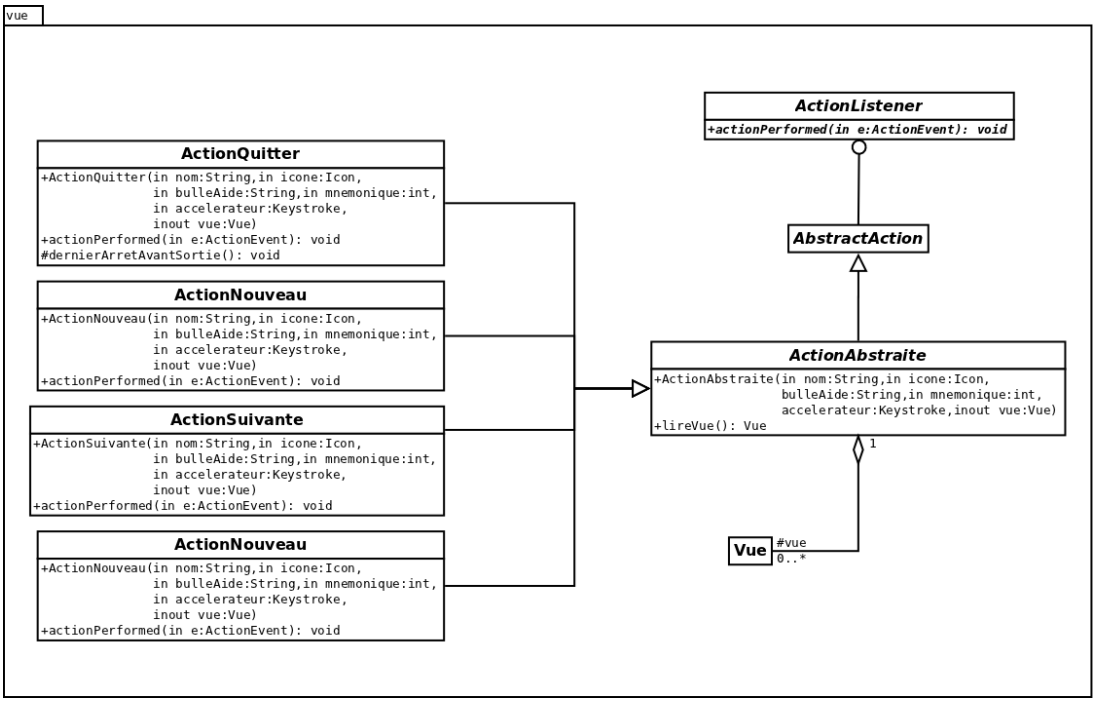

FIG. 5 – Diagramme de classes relatif aux actions des barres de menus et d'outils de la vue.

de la méthode `actionPerformed`. Cette méthode permet de faire passer la cellule correspondante du modèle d'un état à un autre, puis de se mettre à jour en conséquence.

La classe `ModeleGraphique` est la représentation graphique du modèle. Cette classe, qui dérive de `JPanel`, définit un attribut `vue` qui la relie à sa vue propriétaire et un autre, `cellules`, qui représente une grille de cellules graphiques. Sa méthode protégée `mettreAJour` commande à toutes les cellules graphiques de se mettre à jour en fonction de leurs homologues dans le modèle.

La Figure 7 présente le diagramme de classes relatif à la barre de défilement permettant de sélectionner la résolution du modèle. La classe `Resolution`, qui dérive de `JScrollBar`, représente une barre de défilement. Cette classe définit un attribut `vue` qui la rattache à sa vue propriétaire. Elle réalise l'interface `AdjustmentListener` qui lui permet d'intercepter tout événement lié à un déplacement du curseur. Par conséquent, chaque instance de cette classe est son propre listener.

Finalement, la Figure 8 présente le diagramme de classes global de la vue. La classe `Vue`, qui dérive de `JFrame`, représente l'interface graphique proprement dite.

1.1 Question

Notre présentateur sera chargé d'instancier un modèle et sa vue. Il sera également responsable du démarrage de cette vue. Les informations qui lui seront transmises par le client seront la résolution minimum du modèle, sa résolution maximum ainsi que sa résolution de départ. Ces informations sont indispensables à la barre de défilement de la vue. Le modèle instancié par le présentateur sera initialisé à partir de cellules mortes. Le présentateur sera affecté au paquetage `presentateur`.

Après avoir recensé les différentes requêtes pouvant être adressées par la vue à son présentateur, écrivez le diagramme de classe de ce dernier.

Réponse : Figure 9.

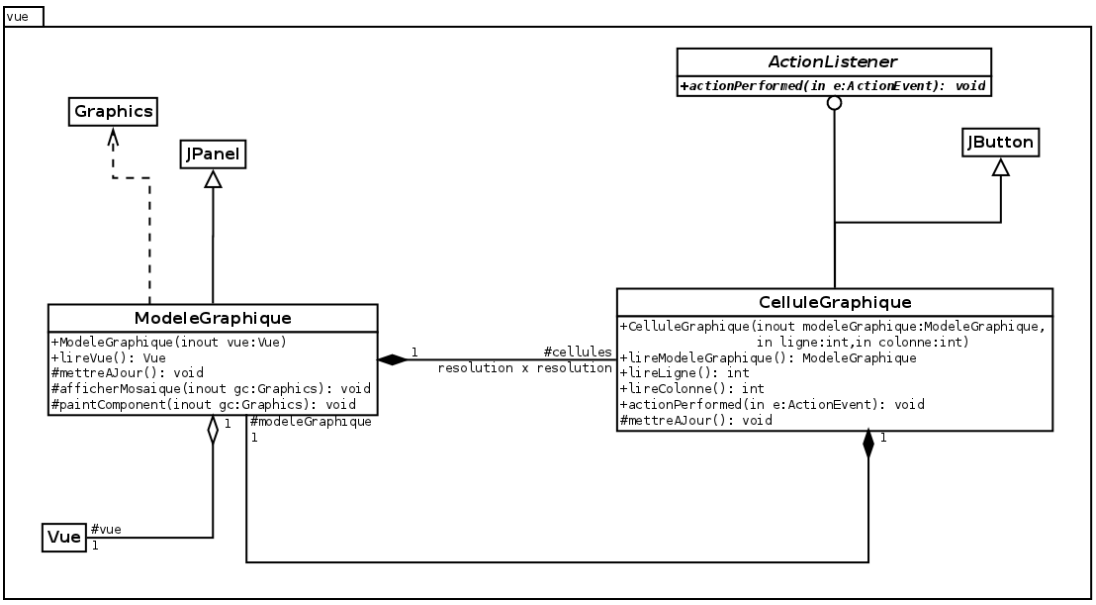

FIG. 6 – Diagramme de classes relatif à la représentation graphique du modèle.

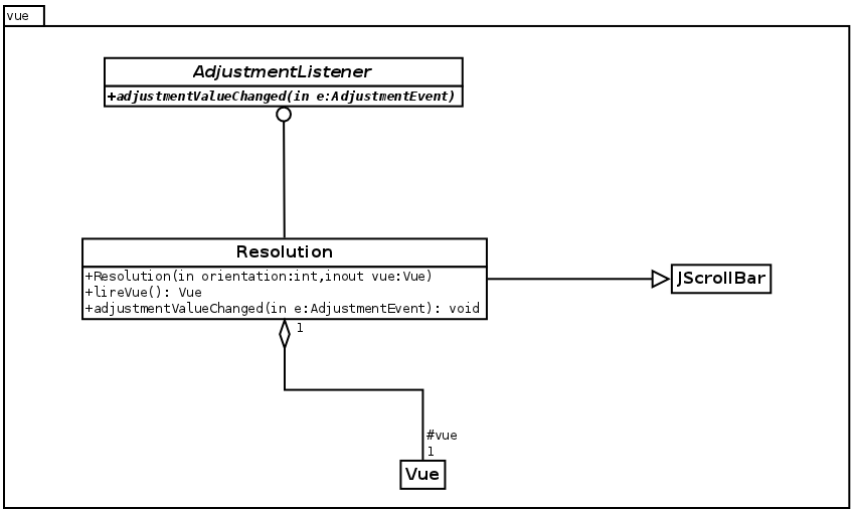

FIG. 7 – Diagramme de classes relatif à la barre de défilement associée à la résolution du modèle.

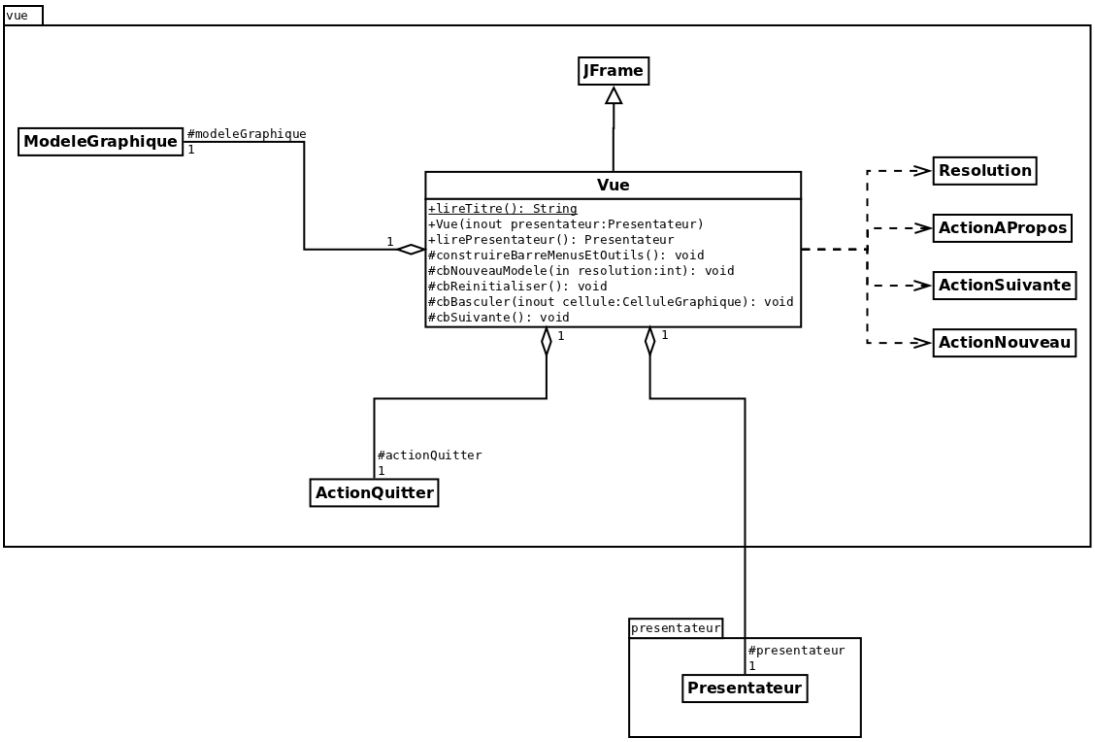

FIG. 8 – Diagramme de classes global de la vue.

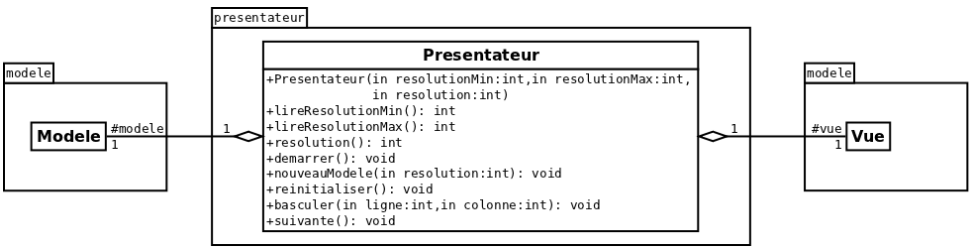

FIG. 9 – Diagramme de classes du présentateur.

1.2 Question

Écrivez la définition de tous les composants mentionnés dans le diagramme de classes du présentateur.

Réponse : voir sources.

1.3 Question

Expliquez précisément la raison pour laquelle la classe `Vue` définit une série de callbacks (méthodes protégées dont le nom est préfixé par `cb`).

Réponse : toutes ces opérations sont mutuellement exclusives. Par conséquent, elles ne peuvent être effectuées qu'après avoir posé un verrou sur la vue. Plutôt que de disperser ces verrous dans toute la vue, il est préférable que la classe `Vue` définissent une série de callbacks `synchronized`.

1.4 Question

Écrivez la définition du callback `cbBasculer` de la classe `Vue`.

Réponse : voir sources.

1.5 Question

Écrivez la redéfinition de la méthode `actionPerformed` de la classe `CelluleGraphique`.

Réponse : voir sources.

1.6 Question

Écrivez la définition de la méthode `mettreAJour` de la classe `CelluleGraphique`.

Réponse : voir sources.

1.7 Question

Écrivez la définition de la méthode `mettreAJour` de la classe `ModeleGraphique`.

Réponse : voir sources.

1.8 Question

Il est possible, en cours d'exécution, de retirer un composant graphique d'un gestionnaire de mise en forme pour le remplacer par un autre. Dans ce cas, l'ancien composant est retiré via la méthode `remove` du gestionnaire dont l'unique argument est l'objet a retirer. L'ajout du nouveau composant est réalisé classiquement via la méthode `add` du gestionnaire. Cependant, comme la vue est déjà affichée, il faut demander le recalcul de la dimension de chaque composant puis son réaffichage. Cette opération est réalisée en invoquant la méthode `validate` des `JFrame`, puis leur méthode `paintComponent`.

Écrivez la définition du callback `cbNouveauModele` de la classe `Vue`.

Réponse : voir sources.

1.9 Question

La méthode `getValue` des `JScrollBar` retourne la graduation entière actuellement repérée par le curseur.

Écrivez la redéfinition de la méthode `adjustmentValueChanged` de la classe `Resolution`.

Réponse : voir sources.

2 Exercice

Le problème du cavalier consiste à faire parcourir toutes les cases d'un échiquier à un cavalier sans jamais passer deux fois de suite par la même case. Le déplacement particulier de cette pièce est rappelé en Figure 10.

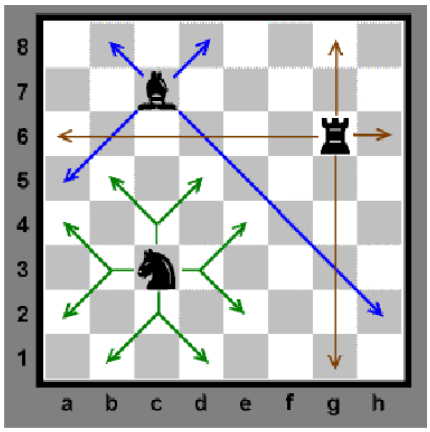

FIG. 10 – Déplacement du cavalier (en bas à gauche) sur un échiquier.

Comme pour le problème des n reines étudié dans un TD-TP précédent, le problème du cavalier est généralisable à des échiquiers de dimension $n \times n$ et peut être résolu par des stratégies exhaustives ou heuristiques.

2.1 Question

En règle générale, un échiquier de dimension $n \times n$ est représenté par une matrice entière de dimension $(n+4) \times (n+4)$. La partie centrale, de dimension $n \times n$, est initialisée à zéro tandis que la partie périphérique est initialisée à une valeur négative quelconque. Lorsqu'un cavalier, positionné sur une case de la partie centrale, tente de se déplacer sur l'une de ses huit cases voisines possibles, trois possibilités sont envisageables :

1. la valeur de la case voisine est négative, ce qui indique que le cavalier est sorti purement et simplement de l'échiquier ;
2. la valeur de la case voisine est $v > 0$, ce qui indique que le cavalier est déjà passé par cette case lors de son $v$ième déplacement ;
3. la valeur de la case voisine est nulle, ce qui indique qu'elle n'a pas encore été visitée par le cavalier.

En vous inspirant des TD-TP précédents ayant pour thèmes le jeu de la vie et la résolution du problème des n reines, écrivez une hiérarchie de classes et/ou d'interfaces telles que :

1. la résolution du problème du cavalier implémente le behavioral design pattern `Strategy`. L'information qui nous intéresse est le nombre de cases de l'échiquier ayant pu être parcourues au départ d'une case donnée ;
2. le parcours des huit cases voisines de celle du cavalier est implémenté via le behavioral design pattern `Iterator`.

Réponse : voir sources.

2.2 Question

Une heuristique relativement efficace consiste à choisir pour prochaine case celle qui minimise le nombre de déplacements possibles par la suite. Cependant, il faut prendre garde aux cases qui n'autorisent plus aucun déplacement par la suite (cases bloquantes). En effet, une telle case ne peut être choisie que si aucune autre case voisine n'autorise de déplacements par la suite.

Écrivez la classe `StrategieLookAhead` qui implémente cette stratégie heuristique. Vous pourrez tirer avantage des `PriorityQueue` du paquetage `java.util` pour la sélection de la prochaine case à occuper.

Réponse : voir sources.

2.3 Question

Écrivez la définition de la classe `Cavalier` qui représente le programme principal. Cette application prendra deux arguments via sa ligne de commandes. Le premier, un entier strictement positif, représentera la dimension du problème à résoudre. Le second, une chaîne de caractères (sans caractère blancs), représentera le nom de la stratégie à mette en œuvre.

L'application commencera par vérifier la validité des arguments. En particulier, elle vérifiera que le nom de stratégie indiqué est connu de sa bibliothèque de stratégies, cette dernière étant représentée par une `HashMap` (attribut de classe). Une fois les vérifications terminées, votre application instanciera la classe `Probleme` puis fera résoudre l'instance correspondante par la stratégie demandée pour chacune de ses cases. Le résultat sera affiché sur la sortie standard sous forme d'un échiquier (matrice entière dont les cases contiennent le nombre de cases couvertes par la stratégie utilisée au départ de cette case).

Une ligne de commandes vide signifiera que l'utilisateur demande de l'aide (il ne s'agit donc pas d'une erreur). Dans ce cas, votre application affichera sur la sortie standard la façon de l'utiliser ainsi que la liste de toutes les stratégies disponibles avec, pour chaque, son nom et sa description succincte.

Réponse : voir sources.