

Interface-utilisateur graphique d'une application Java

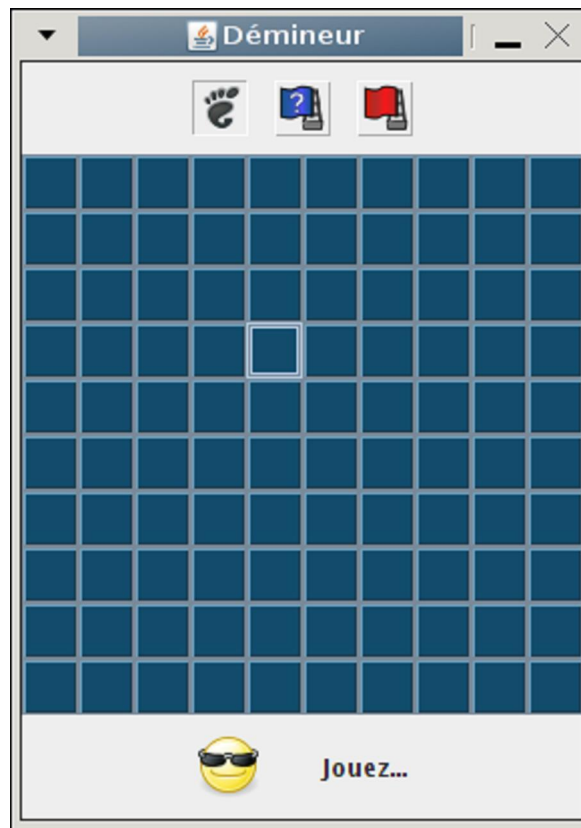
DUT Informatique 1^{ère} année

Cyril Pain-Barre & Henri Garreta

TP 6 - Un démineur

L'objet de ce TP est la création d'un démineur très simple.

Celui-ci se présente sous la forme d'une fenêtre (**JFrame**) :



composée de trois parties :

- en haut se trouve une barre d'action, elle même contenant trois boutons :
 - marcher,
 - marquer comme potentiellement dangereuse,
 - marquer comme contenant une mine ;
- au milieu se trouve la grille du démineur. Selon l'action choisie, on dévoilera une case (si marcher), ou on marquera une case sans la dévoiler (pour les deux autres actions). Seule l'action marcher fait progresser la partie ;
- en bas se trouve la barre d'état. Sur l'image, elle indique que la partie est en cours. En fin de partie, elle indique si le joueur a gagné ou s'il a perdu.

Règles du jeu

Pour gagner une partie, le joueur doit marcher sur toutes les cases ne contenant pas de mine. S'il

marche sur une case minée, celle-ci explose et les autres mines sont dévoilées comme sur l'image suivante :



Durant la partie, le joueur est libre de marquer des cases comme potentiellement dangereuses ou minées, ce qui ne les dévoile pas. Sur l'image suivante, le joueur a marqué des cases minées (drapeau rouge), d'autres comme potentiellement dangereuses (drapeau bleu avec un point d'interrogation) et a laissé des cases manifestement minées sans les marquer :



Le joueur doit marcher sur toutes les cases non minées par gagner, comme sur l'image suivante :



Pour ce TP, le joueur ne pourra faire qu'une partie, mais il sera possible d'étendre le démineur pour en faire plusieurs. Ainsi, lorsqu'une partie est terminée, il n'est plus possible de cliquer sur la grille du jeu.

Les différents composants du démineur

Pour réaliser cette application, on définira 5 classes :

- **FenetreDemineur**, sous-classe de `JFrame`, qui est illustrée par les images précédentes.
- **BarreAction**, sous-classe de `JPanel`, qui est la barre du haut contenant les boutons d'action. En outre, cette classe implémentera l'interface `ActionListener`, pour le traitement des actions sur ces trois boutons.
- **BarreEtat**, sous-classe de `JLabel`, qui est la barre du bas contenant une icône et un texte qui l'accompagne, informant si la partie est en cours, gagnée ou perdue.
- **ChampMine**, sous-classe de `JPanel`, qui est la grille du jeu figurant au centre de la fenêtre. C'est en réalité la classe majeure de l'application.
- **ControleurJeu** est une classe sans apparence visuelle mais implémentant l'interface `ActionListener`, et qui réagit à l'action de l'utilisateur lorsque celui-ci clique sur une case de la grille.

Pour commencer

- créer un projet `demineur`,
- extraire dans le répertoire `demineur` une des archives [images.tgz](#) ou [images.zip](#) contenant les différentes icônes dont vous aurez besoin.

La classe `BarreEtat`

C'est la plus simple des classes de l'application. C'est une extension de `JLabel` qui se contente d'afficher une image et un texte, le tout centré horizontalement. Pour cela, cette classe commence par les définitions des constantes statiques suivantes :

```
public class BarreEtat extends JLabel {  
  
    private static final String cheminBase = "images/";  
  
    private static final ImageIcon iconePartieEnCours =  
        new ImageIcon(cheminBase + "partie_en-cours.png");  
  
    private static final ImageIcon iconePartieGagnee =  
        new ImageIcon(cheminBase + "partie_gagnee.png");  
  
    private static final ImageIcon iconePartiePerdue =  
        new ImageIcon(cheminBase + "partie_perdue.png");  
  
    ...  
}
```

où `cheminBase` doit être adapté à votre arborescence, afin de désigner le répertoire où vous avez extrait `images.tgz`.

La classe `BarreEtat` définit les trois méthodes publiques suivantes :

- `public void setPartieEnCours()` : qui fixe `iconePartieEnCours` comme icône et "Jouez..." comme texte,
- `public void setPartieGagnee()` : qui fixe `iconePartieGagnee` comme icône et "Gagné !" comme texte,
- `public void setPartiePerdue()` : qui fixe `iconePartiePerdue` comme icône et "Perdu !" comme texte.

Le seul constructeur de cette classe a le profil suivant :

```
public BarreEtat()
```

et se contente d'appeler `setPartieEnCours()` puis d'améliorer son rendu visuel :

- les éléments doivent être centrés horizontalement,
- l'espace entre l'icône et le texte doit être de 30 pixels,
- on doit placer autour une bordure vide d'épaisseur 10 pixels.

Exercice 6.1

[\[Correction\]](#)

1. Créer un projet `demineur` et définir la classe `BarreEtat` (prévoir la méthode `main` qui rend cette classe exécutable).
2. Définir la méthode `main` de manière à afficher trois objets `BarreEtat`, un pour chaque état du jeu, dans une fenêtre comme l'illustre l'image ci-dessous :



La classe `BarreAction`

Légèrement plus complexe que la précédente, cette classe est aussi une extension de `JPanel` et a pour objectif principal de fournir les méthodes publiques :

- `public boolean estActionMarcher()` qui renvoie `true` si et seulement si l'action choisie est « marcher »,
- `public boolean estActionMarquerDoute()` qui renvoie `true` si et seulement si l'action choisie est « marquer comme dangereuse »,
- `public boolean estActionMarquerMine()` qui `true` si et seulement si l'action choisie est « marquer comme minée »

Un objet de cette classe doit proposer trois boutons radio, en être l'écouteur (donc implémenter l'interface `ActionListener`), et mettre à jour une variable privée `action` selon le bouton cliqué.

Pour cela, la définition de la classe commence par les déclarations suivantes :

```
public class BarreAction extends JPanel implements ActionListener {

    private static final int MARCHER          = 1;
    private static final int MARQUER_DOUTE    = 2;
    private static final int MARQUER_MINE     = 3;

    private int action; // action choisie




    private JRadioButton boutonMarcher;
    private JRadioButton boutonDoute;
    private JRadioButton boutonMine;

    private static final String cheminBase = "images/";




    ...
}
```

Le constructeur de cette classe (qui est un `JPanel`) doit créer et disposer les boutons, les rendre mutuellement exclusifs (utilisation d'un objet `ButtonGroup`) et se déclarer comme écouteur des boutons. Les trois boutons ont un rendu visuel, défini par une image, dépendant de leur état :



- **boutonMarcher :**


- si sélectionné, l'image est `marcher_on.png` ()
- si non sélectionné, l'image est `marcher_off.png` ()
- pour le rollover, l'image est `marcher_roll.png` ()

- **boutonDoute :**

- si sélectionné, l'image est `marquer_doute_on.png` ()
- si non sélectionné, l'image est `marquer_doute_off.png` ()
- pour le rollover, l'image est `marquer_doute_roll.png` ()

- **boutonMine :**

- si sélectionné, l'image est `marquer_mine_on.png` ()
- si non sélectionné, l'image est `marquer_mine_off.png` ()

- pour le rollover, l'image est `marquer_mine_roll.png` ()

Le constructeur sélectionne par défaut le bouton `boutonMarcher` et initialise en conséquence la variable privée `action`.

Exercice 6.2

[Correction : voir 6.3]

1. Écrire le constructeur de la classe `BarreAction`.
2. Écrire les trois méthodes publiques `estActionMarcher()`, `estActionMarquerDoute()` et `estActionMarquerMine()` qui ne font que comparer la variable `action` et la constante correspondante.

Il reste à définir la méthode :

```
public void actionPerformed(ActionEvent e) {
    ...
}
```

pour implémenter l'interface `ActionListener`. Cette méthode se contente de mettre à jour la variable `action` selon le bouton cliqué.

Exercice 6.3

[\[Correction\]](#)

1. Définir la méthode `actionPerformed`

Cette classe devrait maintenant être opérationnelle. Si vous souhaitez la tester, vous pouvez définir la méthode `main()` pour afficher une instance de la classe `BarreAction` dans une fenêtre et éventuellement écrire sur la console l'action choisie (mais il faut ajouter cette écriture dans `actionPerformed`)

Les classes `ChampMine` et `ControleurJeu`

Ce sont les classes principales de l'application et sont intimement liées, à tel point que `ControleurJeu` pourrait être une classe interne de `ChampMine`. Mais nous avons choisi d'en faire deux classes publiques (donc dans des fichiers séparés).

La classe `ChampMine` crée la grille du jeu, l'initialise (c'est-à-dire y place des mines), puis offre un certain nombre de méthodes permettant de jouer. Si on met de côté l'aspect visuel de la grille et les interactions avec le joueur, une grille de jeu n'est qu'une simple matrice carrée d'entiers déclarée comme variable d'instance privée :

```
private int[][] grille;           // grille du jeu (partie non visible)
```

Elle est de taille `taille x taille`, où `taille` est une variable d'instance privée :

```
private int taille;
```

dont la valeur sera communiquée en paramètre du constructeur de `ChampMine`.

La grille va contenir `nbMines` mines, où `nbMines` est une variable d'instance privée :

```
private int nbMines;
```

dont la valeur sera aussi communiquée en paramètre du constructeur de `ChampMine`.

Fixons les constantes suivantes de la classe `ChampMine` :

```
private final int VIDE = -1;
private final int MINE = -2;
```

En prenant 10 pour `taille` et 15 pour `nbMines` (ce seront les valeurs utilisées pour le TP), après création, initialisation et placement (aléatoire) des mines, la grille peut contenir quelque chose comme :

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-2	-1	-1	-2	-1	-2	-1	-1
-1	-1	-1	-2	-1	-2	-1	-1	-1	-1
-1	-1	-1	-2	-1	-2	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-2	-1	-2	-1	-2	-1	-1	-1	-1
-1	-1	-2	-1	-1	-1	-2	-1	-1	-1
-2	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-2	-1	-1	-1	-1	-2

Exercice 6.4

[\[Correction\]](#)

1. Créer la classe publique exécutable `ChampMine` (on ajoutera l'extension de `JPanel` plus tard).
2. Définir la méthode privée de prototype `private void creerGrille()` qui :
 - crée une grille et initialise toutes ses cases à `VIDE`,
 - puis appelle la méthode `placerMines()`.
3. Définir la méthode privée de prototype `private void placerMines()` qui place aléatoirement `nbMines` dans la grille. Pour cela, on utilisera un objet appelé `generateur`, instance de la classe `java.util.Random`, créé en passant en argument au constructeur l'entier `System.currentTimeMillis()` (pour initialiser la séquence de nombres aléatoires). Pour obtenir un nombre entier aléatoire entre 0 et n (non compris), on utilisera `generateur.nextInt(n)`. Faire attention de ne pas placer deux mines au même endroit.
4. Définir la méthode privée de prototype `private void ecrireGrille()` qui affiche sur la console un rendu textuel de la grille. Pour la grille précédente, l'affichage doit être :

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -2 -1 -1 -2 -1 -2 -1 -1
-1 -1 -1 -2 -1 -2 -1 -1 -1 -1
-1 -1 -1 -2 -1 -2 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -2 -1 -2 -1 -2 -1 -1 -1 -1
-1 -1 -2 -1 -1 -1 -2 -1 -1 -1
-2 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -2 -1 -1 -1 -1 -2
```

5. Définir un constructeur privé de `ChampMine` de prototype `private ChampMine(int taille, int nbMines)` qui :
 - affecte aux variables d'instance `taille` et `nbMines` les valeurs communiquées par le constructeur,
 - appelle `creerGrille()` pour créer une grille.
6. Dans la méthode `main()`, créer un champ de mines de taille 10 contenant 15 mines et faire afficher la grille. Exécuter (plusieurs fois) pour s'assurer que tout se passe bien.

Pré-traitement de la grille

Afin de ne pas ralentir le déroulement du jeu, on va pré-traiter la grille et calculer, pour chaque case ne contenant pas de mine, le nombre de cases voisines (soit au plus 8) qui en contiennent.

Par exemple, pour la case du milieu dans la configuration suivante :

```
-1 -2 -2
-2 -1 -1
-1 -1 -2
```

le nombre de mines voisines est 4.

Exercice 6.5

[\[Correction\]](#)

1. Écrire la méthode privée de prototype `private int nbMinesVoisines(int i, int j)` qui renvoie, pour la case (i,j) qui ne contient pas de mine, le nombre de mines voisines. Attention, il faut tenir compte des bords de la grille.
2. Écrire la méthode privée de prototype `private void mettreAJourVoisinage()` qui, pour toutes les cases de la grille ne contenant pas de mines, remplace sa valeur (`VIDE`) par le nombre de mines voisines. Ainsi, la grille de la figure précédente deviendrait :

0	1	1	1	1	1	2	1	1	0
0	1	-2	2	3	-2	3	-2	1	0
0	1	3	-2	5	-2	4	1	1	0
0	0	2	-2	4	-2	2	0	0	0
1	1	3	2	4	2	2	0	0	0
1	-2	3	-2	2	-2	2	1	0	0
2	3	-2	2	2	2	-2	1	0	0
-2	2	1	1	0	1	1	1	0	0
1	1	0	1	1	1	0	0	1	1
0	0	0	1	-2	1	0	0	1	-2

3. Écrire une méthode privée de prototype `private void ecrireGrilleVoisinage()` qui écrit sur la console la grille dont la mise à jour du voisinage a déjà été effectuée. A la place de la valeur -2, afficher un X. Pour la grille précédente, l'affichage devrait être :


```

0 1 1 1 1 1 2 1 1 0
0 1 x 2 3 x 3 x 1 0
0 1 3 x 5 x 4 1 1 0
0 0 2 x 4 x 2 0 0 0
1 1 3 2 4 2 2 0 0 0
1 x 3 x 2 x 2 1 0 0
2 3 x 2 2 2 x 1 0 0
x 2 1 1 0 1 1 1 0 0
1 1 0 1 1 1 0 0 1 1
0 0 0 1 x 1 0 0 1 x

```

4. Dans la méthode `main()`, à la suite de `ecrireGrille()`, ajouter un appel à `mettreAJourVoisinage()` puis à `ecrireGrilleVoisinage()`.

Avant de passer à la partie graphique de la grille, définissons encore une variable d'instance privée :

```
private int nbCasesDecouvertes;
```

qui contiendra le nombre de cases sur lesquelles le joueur aura marché et qui sera initialisée à 0 au début de la partie.

Enfin, dotons-nous des deux premières méthodes publiques.

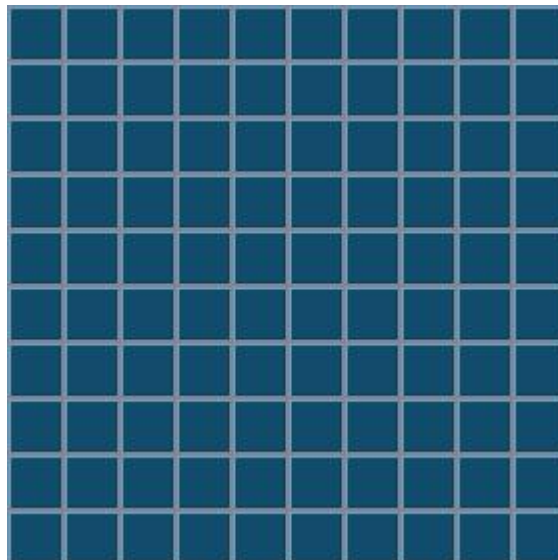
Exercice 6.6

[\[Correction\]](#)

1. Définir la méthode publique de prototype `public boolean aGagne()` qui renvoie `vrai` si et seulement si le joueur a découvert toutes les cases non minées (donc a marché sur toutes les cases moins celles minées).
2. Définir la méthode publique de prototype `public boolean estUneMine(int i, int j)` qui renvoie `vrai` si et seulement si la case (i,j) contient une mine.

La partie graphique de `ChampMine`

Ainsi que nous l'avons dit, `ChampMine` une extension de `JPanel` et contient une grille de `JButton` (un `JButton` par case du jeu). Visuellement, au début du jeu ce `JPanel` se présente ainsi :



puis évolue au cours du temps, lorsque des cases sont découvertes (action marcher) ou marquées, les boutons correspondants changeant alors d'icône.















Exercice 6.7

[\[Correction\]](#)

1. Ajouter l'extension de `JPanel` dans la définition de la classe `ChampMine`.
2. Définir un constructeur public de la classe `ChampMine` de prototype `public ChampMine(int taille, int nbMines, BarreAction barreAction, BarreEtat barreEtat)` qui :
 - initialise les variables privées `taille` et `nbMines`,
 - initialise la variable privée `nbDecouvertes`,
 - fixe un `LayoutManager` adéquat pour notre `ChampMine`.

On complètera ce constructeur plus tard.

Pour une case donnée, les différents icônes que son bouton peut afficher sont :

- pour une case non découverte :
 -  (`case_normale.png`) pour une case non découverte ni marquée,
 -  (`case_marquee_douteuse.png`) pour une case non découverte mais marquée comme dangereuse,
 -  (`case_marquee_minee.png`) pour une case non découverte mais marquée comme minée,
 -  (`case_mine.png`) utilisée en fin de jeu pour montrer où étaient les mines
- pour une découverte :
 -  (`case_mine_explosee.png`) pour une case découverte mais qui contenait une mine.
 -          pour une case découverte ayant 0, 1, 2, 3, 4, 5, 6, 7 ou 8 mines voisines. Pour un nombre de mines voisines n , le fichier image correspondant se nomme `case_n.png`

Les 5 premiers icônes seront chargés dans les variables privées :

```
private ImageIcon iconeCaseNormale;
private ImageIcon iconeMarqueDoute; // icone d'une case marquee douteuse
private ImageIcon iconeMarqueMine; // icone d'une case marquee comme minee
private ImageIcon iconeCaseMinee; // icone d'une case contenant une mine
private ImageIcon iconeCaseExplosee; // icone d'une mine ayant explose
```

alors que les icônes pour une case découverte mais ne contenant pas de mine seront stockés dans le tableau privé :

```
private ImageIcon [] iconesChiffres; // icones des cases decouvertes
```

On se servira à nouveau d'une constante statique privée contenant le chemin des fichiers image :

```
private static final String cheminBase = "images/";
```

Exercice 6.8

[\[Correction\]](#)

Écrire la méthode privée de prototype :

```
private void chargerIcones()
```

qui charge les images dans les objets `ImageIcon` adéquats.

L'ensemble des boutons seront contrôlés par un seul objet de la classe `ControleurJeu` qui sera créé dans le constructeur public de `ChampMine`, qui reste à compléter, et stocké dans la variable privée :

```
private ControleurJeu controleur; // le controleur du jeu: écouteur des boutons
```

Les boutons eux-mêmes sont stockés dans une matrice carrée, de même dimension que la grille, et déclarée comme une variable d'instance privée :

```
private JButton[][] boutons; // boutons du jeu (partie affichée)
```

La seule particularité de ces boutons est que lorsque l'un d'eux est cliqué, le contrôleur doit pouvoir facilement déterminer ses coordonnées afin de traiter l'événement. Pour cela, il faut que chaque bouton définisse une chaîne `actionCommand` (appel de `setActionCommand`) qui indique ses coordonnées et qui soit facilement exploitable. Un moyen simple est par exemple que cette chaîne `actionCommand` contienne l'indice de la ligne suivi d'un espace suivi de l'indice de la colonne. Ensuite, il suffira de traiter cette `actionCommand` afin d'en extraire les informations utiles.

Exercice 6.9

[Correction : voir 6.10]

Écrire la méthode privée de prototype

```
private void creerBoutons()
```

qui :

- créé la matrice des boutons,
- et pour chaque bouton :
 - lui donne comme icône celui représentant une case normale,
 - supprime ses marges,
 - lui donne comme taille préférée celle de l'icône (qu'on obtient par appels des méthodes `getIconWidth()` et `getIconHeight()` de la classe `ImageIcon`),
 - définit comme indiqué sa chaîne `actionCommand`,
 - lui ajoute `controleur` comme écouteur,
 - l'ajoute à l'objet `ChampMine`.

Exercice 6.10

[[Correction](#)]

Afin de faciliter la tâche du contrôleur, définir deux méthodes publiques de prototypes :

```
public int getLigneBouton(String commande)
public int getColonneBouton(String commande)
```

qui, à partir de la chaîne `commande` correspondant à l'`actionCommand` d'un bouton cliqué, retournent respectivement la ligne et la colonne du bouton concerné. Indication : pour traiter la chaîne `commande`, vous aurez certainement besoin des méthodes `substring()` et `indexOf()` de la classe `String`, ainsi que de la méthode `parseInt()` de la classe `Integer`.

La classe `ChampMine` est presque terminée. Il reste à compléter le constructeur (public) et à définir quelques méthodes publiques :

- `public void setMarqueeDoute (int i, int j)` qui change l'icône du bouton (i,j) qui devient `iconeMarqueeDoute`
- `public void setMarqueeMine(int i, int j)` qui change l'icône du bouton (i,j) qui devient `iconeMarqueeMine`
- `public void setExplosee(int i, int j)` qui désactive le bouton (i,j) et place `iconeCaseExplosee` comme icône du bouton désactivé
- `public void setDecouverte(int i, int j)` qui désactive le bouton (i,j) et place comme icône du bouton désactivé celui correspondant à son nombre de mines voisines, et incrémente la variable `nbCasesDecouvertes`
- `public void decouvrirGrille()` qui est appelée en fin de partie pour dévoiler les mines. Cette méthode parcourt la grille des boutons à la recherche de boutons toujours actifs, les désactive, en mettant comme icône du bouton désactivé `iconeCaseNormale` pour ceux ne contenant pas de mines, et `iconeCaseMinee` pour ceux contenant une mine. A la fin de cette méthode, tous les boutons de la grille sont désactivés.

Exercice 6.11

[\[Correction\]](#)

Écrire les cinq méthodes mentionnées ci-dessus.

Exercice 6.12

[\[Correction\]](#)

Compléter le constructeur public de `ChampMine` pour :

- affecter à la variable `controleur` un nouvel objet `ControleurJeu` (`barreAction`, `barreEtat`, `this`),
- créer la grille du jeu,
- mettre à jour le voisinage des mines,
- charger les icônes,
- créer les boutons.

La classe `ControleurJeu`

Cette classe implémente l'interface `ActionListener` et est chargée de réagir au clic de l'utilisateur sur les boutons de la grille du jeu. Elle utilise trois variables d'instance privée :

```
public class ControleurJeu implements ActionListener {
    private BarreAction barreAction;
    private BarreEtat barreEtat;
    private ChampMine champMine;
    ...
}
```

Son unique constructeur prend en paramètres des objets correspondants et ne fait que les affecter à ses variables d'instance :

```
ControleurJeu (BarreAction barreAction, BarreEtat barreEtat, ChampMine champMine) {
    ...
}
```

Exercice 6.13

[\[Correction : voir 6.14\]](#)

Compléter le constructeur de `ContrôleurJeu`.

La classe `ContrôleurJeu` n'a qu'une méthode publique, imposée par sa déclaration d'implémentation de `ActionListener` :

```
public void actionPerformed(ActionEvent e)
```

Cette méthode n'est appelée que lorsque l'utilisateur clique sur un bouton actif de la grille. Cependant, l'effet de ce clic dépend de l'action voulue par l'utilisateur (marcher, marquer comme douteuse ou marquer comme minée).

L'utilisateur ne peut gagner ou perdre qu'en marchant sur une case. S'il marche sur une case non minée, elle doit être découverte et s'il a découvert toutes les cases non minées, alors les mines sont dévoilées et la barre d'état est mise à jour en conséquence. S'il marche sur une case minée, alors celle-ci explose, les autres mines sont dévoilées et la barre d'état est mise à jour en conséquence.

Exercice 6.14

[\[Correction\]](#)

Écrire la méthode `actionPerformed` de `ContrôleurJeu`.

La classe exécutable `FenetreDemineur`

C'est une classe très simple qui est une extension de `JFrame`. Elle crée une barre d'action qu'elle affiche au nord, une barre d'état qu'elle affiche au sud, et un champ de mine de taille 10 x 10 contenant 15 mines, qu'elle affiche au centre.

Exercice 6.15

[\[Correction\]](#)

Écrire la classe `FenetreDemineur`.

Des extensions possibles

- gestion et affichage d'un score
- mise en place d'un menu permettant de :
 - démarrer une nouvelle partie ou arrêter la partie en cours, quitter
 - choisir une taille de grille différente
 - choisir un niveau de difficulté différent (nombre de mines), ce qui peut avoir un impact sur le score

etc.