

Support de cours

Cours:

Introduction à la programmation orientée objet (en Java)

Vidéo:

W11-05-morpions-JAVA-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Classe jeumorpion. Méthode placerrond. Utilisateurs de la classe jeumorpion. Choix d'implémentation. Méthode placercoup. Fait évident. Cases vides. Programmeur externe. Version précédente. Message compréhensible. Choix d'implémentation internes. Méthode publique. Façon analogue. Usage de ce boolean. Nombre de décisions.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en Java).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Encapsulation et abstraction : étude de cas

(Partie 2)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s





Pour programmer une classe JeuMorpion proprement encapsulée,

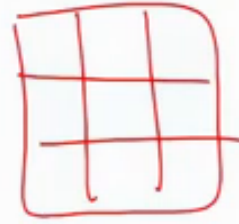
notes

résumé

0m 1s



```
class JeuMorpion {  
    private final static int VIDE = 0;  
    private final static int ROND = 1;  
    private final static int CROIX = 2;  
  
    private int[][] grille;  
  
    public void initialise() {  
        grille = new int[3][3];  
        for (int i=0; i < grille.length; ++i) {  
            for (int j=0; j < grille[i].length; ++j)  
            {  
                grille[i][j] = VIDE;  
            }  
        }  
    }  
    //...  
}
```



il est impératif que les choix d'implémentation ne soient pas accessibles ou visibles, au travers de l'interface d'utilisation offerte au programmeur externe. Par ailleurs, les choix d'implémentation internes, comme par exemple, comment représenter la grille, comment modéliser les jetons qui se posent sur la grille, doivent être du ressort de celui qui implémente la classe JeuMorpion et non pas de celui qui l'utilise. Nous allons étudier dans ce qui suit une variante qui améliore la version précédente. Alors, pour programmer proprement la classe JeuMorpion il va falloir prendre un certain nombre de décisions quant à l'implémentation de la grille, ici on décide de l'implémenter de façon un peu plus parlante et explicite que nous l'avons fait tout à l'heure en utilisant un tableau à deux dimensions qui est plus proche de la représentation effective du Jeu du Morpion. Nous devons aussi décider comment codifier les données que nous plaçons à l'intérieur de la grille, et ici il vaut toujours mieux expliciter le plus possible les choix que l'on fait.

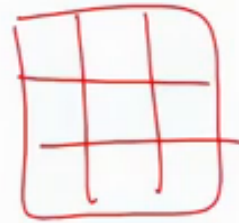
notes

résumé

0m 5s



```
class JeuMorpion {  
    private final static int VIDE = 0;  
    private final static int ROND = 1;  
    private final static int CROIX = 2;  
  
    private int[][] grille;  
  
    public void initialise() {  
        grille = new int[3][3];  
        for (int i=0; i < grille.length; ++i) {  
            for (int j=0; j < grille[i].length; ++j)  
            {  
                grille[i][j] = VIDE;  
            }  
        }  
    }  
    //...  
}
```



Ceci peut se faire par exemple en introduisant un certain nombre de constantes qui vont donner une sémantique claire

notes

résumé

1m 1s



```
public boolean placerRond(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, ROND);  
}  
  
public boolean placerCroix(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, CROIX);  
}  
  
// ici on peut rajouter une methode getJoueurGagnant()  
} // fin de la classe JeuMorpion
```

à la signification des différents entiers qu'on va placer dans notre tableau, Ensuite, contrairement à ce qui a été fait dans la version précédente, ne pas compter sur des valeurs implicites par défaut pour le remplissage initial de la grille, mais rendre ce remplissage le plus explicite possible, ce que nous faisons ici. Donc en lisant par exemple cet portion de code, il est tout à fait évident que l'on est en train de remplir la grille du Jeu Morpion... avec des cases vides.

notes

résumé

1m 7s



```
public boolean placerRond(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, ROND);  
}  
  
public boolean placerCroix(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, CROIX);  
}  
  
// ici on peut rajouter une methode getJoueurGagnant()  
} // fin de la classe JeuMorpion
```

Ensuite l'idée principale est d'offrir aux utilisateurs de la classe JeuMorpion uniquement un ensemble en deux méthodes bien choisies qui lui permettront d'interagir avec le plateau de jeu sans pour autant en connaître les détails d'implémentation. Par exemple, on peut lui fournir une méthode placerRond en lui indiquant que le mode d'interaction, c'est de fournir une ligne et une colonne, ce qui est assez naturel, et ensuite on se préoccupe en interne de comment implémenter concrètement

notes

résumé

1m 34s



```
public boolean placerRond(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, ROND);  
}  
  
public boolean placerCroix(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, CROIX);  
}  
  
// ici on peut rajouter une methode getJoueurGagnant()  
} // fin de la classe JeuMorpion
```

le fait de mettre un rond à une ligne et une colonne données sur la grille de jeu. De façon analogue, on fournirait une méthode publique `placerCroix` qui permettrait donc au programmeur utilisateur du jeu Morpion de placer une croix en spécifiant simplement à quelle ligne et à quelle colonne il désire le faire. Placer un rond ou une croix sur une ligne et une colonne données, se fait exactement selon les mêmes modalités peu importe que ce soit un rond ou une croix que l'on place, donc il est naturel ici de penser à modulariser les traitements en programmant une méthode `placerCoup` qui se charge de placer un coup sur une ligne et une colonne données. Les deux méthodes `placerRond` et `placerCroix` suffisent complètement à l'utilisateur externe pour pouvoir placer un rond ou une croix sur le plateau, il n'est pas nécessaire par conséquent de lui fournir un accès à `placerCoup` ; l'interface d'utilisation est complètement définie pour placer une pièce, uniquement au moyen de ces deux méthodes-là. Ce qui veut dire qu'au moment où on va coder `placerCoup`,

notes

résumé

2m 1s



Jeu de Morpion : bien encapsuler (2)

```
/**
 * Place un coup sur le plateau.
 * @param ligne La ligne 0, 1, ou 2
 * @param colonne La colonne 0, 1, ou 2
 * @param coup Le coup à placer
 */
private boolean placerCoup(int ligne, int colonne, int coup) {
    if (ligne < 0 || ligne >= grille.length
        || colonne < 0 || colonne >= grille[ligne].length) {
        // traitement de l'erreur ici
    }
    if (grille[ligne][colonne] == VIDE) {
        // case vide, on peut placer le coup
        grille[ligne][colonne] = coup;
        return true;
    } else {
        // case déjà prise, on signale une erreur
        //...
        return false;
    } // suite
}
```

on va coder cette méthode comme une méthode privée de la classe qui aide simplement la classe à réaliser les traitements souhaités mais qui n'est pas offerte comme fonctionnalité au programmeur utilisateur de la classe JeuMorpion. Comme c'est le programmeur utilisateur qui va fournir la ligne et la colonne sur lesquelles il souhaite placer le coup, il est nécessaire, si l'on veut rendre les traitements plus robustes aux erreurs, de traiter les cas où ces lignes et ces colonnes ne correspondraient pas aux dimensions souhaitées pour la grille. Donc typiquement si la ligne est en dessous de l'indice 0 ou en dessus de la taille de la grille, en termes de nombre de lignes, alors on doit signaler une erreur. Donc il faudrait rendre les traitements plus robustes en se chargeant de l'erreur. Une fois que l'on a vérifié que la ligne et la colonne transmises par l'utilisateur externe sont bien valides, il nous faut encore vérifier que cette case particulière de la grille est bien vide ; en effet, sur le jeu du morpion, vous n'avez pas le droit de placer un coup sur une case déjà remplie et si ce n'est pas le cas, il faut le signaler à nouveau comme un cas d'erreur. Le boolean qui sert de type de retour à la méthode placerCoup

notes

résumé

3m 1s



```
public boolean placerRond(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, ROND);  
}  
  
public boolean placerCroix(int ligne, int colonne) {  
    return placerCoup(ligne, colonne, CROIX);  
}  
  
// ici on peut rajouter une methode getJoueurGagnant()  
} // fin de la classe JeuMorpion
```

permet d'indiquer si oui ou non on a réussi à placer notre coup sur la ligne et la colonne souhaitées.

notes

résumé

4m 13s



Comment faire maintenant pour faire un rond sur la case en haut à gauche ?

```
→ JeuMorpion jeu = new JeuMorpion();  
→ jeu.initialise();  
→ valide = jeu.placerRond(0, 0); //boolean déclaré plus haut
```

Et pour faire une croix sur la 1^{re} ligne, 2^e colonne ?

```
valide = jeu.placerCroix(0, 1);
```

On aurait pu également décider d'appeler les colonnes 1, 2, 3 au lieu de 0, 1, 2 : c'est une question de convention. C'est justement ce sur quoi il faut se mettre d'accord quand on définit une interface.

Les méthodes placerRond et placerCroix vont faire usage de ce boolean, donc elles vont retourner le boolean issu de l'appel à placerCoup pour indiquer à l'utilisateur externe si oui ou non le placement d'une croix ou d'un rond à l'emplacement souhaitées, s'est bien passé ou non. Voyons maintenant comment toutes ces modifications de la classe JeuMorpion se manifestent pour celui qui l'utilise. Comme tout à l'heure, un programmeur qui souhaiterait utiliser la classe JeuMorpion va déclarer une variable de ce type-là, il va pouvoir l'initialiser au travers des fonctionnalités publiquement offertes mais surtout au moment de placer un coup, il va pouvoir utiliser des fonctionnalités beaucoup plus explicites, comme ici placerRond, et qui ne nécessitent de sa part aucune connaissance de comment sont codifiées les données en interne.

notes

résumé

4m 22s



- **Validation** : il est impossible de mettre une valeur invalide dans le tableau (autre que 0, 1, ou 2)

Donc ici si je veux placer un rond, j'indique que je place un rond dans le jeu à la ligne 0 et à la colonne 0 et je peux également m'informer du fait que le fait de placer ce rond s'est bien passé ou non. On procéderait bien sûr de façon tout à fait analogue pour placer une croix sur une autre case de la grille de jeu. Cette façon d'implémenter la classe JeuMorpion qui permet un usage beaucoup plus robuste et beaucoup plus explicite ne nous affranchit pas cependant de devoir expliciter un certain nombre de conventions d'utilisations. Par exemple ici, nous imposons à celui qui utilise les méthodes fournies, d'utiliser un système de numérotation qui est le suivant pour les lignes et les colonnes ; nous aurions pu en choisir un autre. Alors ces conventions doivent absolument être documentées au niveau de l'interface d'utilisation. La nouvelle implémentation de la classe JeuMorpion comporte de nombreux avantages par rapport à la première version proposée. En effet, elle permet de valider les données,

notes

résumé

5m 13s



- ▶ **Validation** : il est impossible de mettre une valeur invalide dans le tableau (autre que 0, 1, ou 2)
- ▶ **Validation** : il est impossible de cocher une case déjà cochée.
- ▶ **Séparation des soucis** : le programmeur-utilisateur n'a pas besoin de savoir comment le plateau est stocké, ni qu'il utilise des entiers, ni quelles valeurs correspondent à quoi.
- ▶ Le code est compréhensible même par un profane – le nom des méthodes exprime clairement ce qu'elles font et s'explique de lui-même.
- ▶ Si on essaie de faire une opération invalide (cocher deux fois la même case, ou une case en dehors du tableau), on obtient un message compréhensible.

par exemple, il n'est plus possible dans cette nouvelle version de mettre une valeur invalide dans le tableau de jeu. Pourquoi ? Eh bien, parce que celui qui utilise les fonctionnalités de la classe JeuMorpion ne sait même pas que le vide est codifié comme un 0, que le 1 et le 2 représentent des ronds et des croix. Donc il utilise simplement une méthode `placerRond` ou `placerCroix` et c'est l'implémentation en interne qui se charge de la bonne codification. De même il est impossible à l'utilisateur externe de cocher une case déjà cochée. Nous avons vu dans l'implémentation de `placerCoup` que celle-ci vérifiait que la case était bien vide avant de pouvoir y placer un nouveau jeton. Un autre point important, et qui est garant d'une bonne conception orientée objet, est ce qu'on appelle la séparation des soucis : celui qui utilise la classe JeuMorpion n'a pas besoin de savoir comment en interne est représentée la grille et ce qui se passe concrètement lorsqu'il place une pièce sur le plateau. Les seules choses dont il a à se préoccuper c'est de la logique du jeu, de comment il veut placer ses ronds et ses croix. Enfin, le code est compréhensible même pour un profane, lorsque j'écris par exemple `jeu.placerRond(0, 0)` je n'ai pas besoin de beaucoup d'explications pour savoir ce qui se passe concrètement. Ce n'était pas le cas dans la version antérieure où pour placer un jeton, je devais passer par des tournures complètement cryptiques de cette nature. Comparez les deux... et vous voyez que la première version est beaucoup plus explicite et claire. De même, de part le fait que l'on traite explicitement les situations d'erreurs et de façon exhaustive, il devient possible désormais d'avoir un message compréhensible pour toutes les opérations invalides. Dans la première version, lorsque nous essayons par exemple de placer un jeton en dehors des limites du tableau,

notes

résumé

6m 13s



- ▶ **Validation** : il est impossible de mettre une valeur invalide dans le tableau (autre que 0, 1, ou 2)
- ▶ **Validation** : il est impossible de cocher une case déjà cochée.
- ▶ **Séparation des soucis** : le programmeur-utilisateur n'a pas besoin de savoir comment le plateau est stocké, ni qu'il utilise des entiers, ni quelles valeurs correspondent à quoi.
- ▶ Le code est compréhensible même par un profane – le nom des méthodes exprime clairement ce qu'elles font et s'explique de lui-même.
- ▶ Si on essaie de faire une opération invalide (cocher deux fois la même case, ou une case en dehors du tableau), on obtient un message compréhensible.

eh bien, cela se traduisait à l'exécution par un message ésotérique de lancement d'exception et un arrêt abrupt du programme. Il devient désormais possible de donner à l'utilisateur un message compréhensible lui indiquant qu'il essaye de placer un jeton dans une case invalide. un jeton dans une case invalide.

notes

résumé