Conception dirigée par les responsabilités

Nous avons vu au paragraphe précédent que l'utilisation de l'encapsulation réduit le couplage et peut diminuer de manière significative la charge de travail nécessaire pour modifier une application. Cependant, l'encapsulation n'est pas le seul facteur qui influence le degré de couplage ; la *conception dirigée par les responsabilités* joue également un rôle important.

Concept

La **conception dirigée par les responsabilités** est le processus de création de classes correspondant à des responsabilités bien définies pour chaque classe. Cette approche peut être utilisée pour décider de la classe qui doit implanter telle ou telle partie des fonctionnalités d'une application.

La conception dirigée par les responsabilités exprime l'idée que chaque classe doit être responsable de la gestion de ses propres données. Nous sommes souvent amenés à ajouter de nouvelles fonctionnalités à une application. Nous devons alors nous demander dans quelle classe il faut ajouter une méthode pour implanter cette nouvelle fonction. Quelle classe devrait être responsable de cette tâche ? La réponse est que la classe responsable du stockage de données D devrait aussi être responsable de la manipulation des données D.

Le bon ou mauvais usage de la conception dirigée par les responsabilités influence le degré de couplage et donc, à nouveau, la facilité avec laquelle une application peut être modifiée ou étendue. Comme d'habitude, nous allons étudier plus en détail ce point à l'aide de notre exemple.

Responsabilités et couplage

Les changements apportés à la classe Room, au paragraphe « Utilisation de l'encapsulation pour réduire le couplage », facilitent l'ajout des nouvelles directions pour les déplacements vers le haut et le bas dans la classe Game. Nous allons vérifier ce point à l'aide d'un exemple : supposons que nous voulions ajouter une nouvelle pièce (la cave) sous le bureau. Tout ce que nous avons à faire dans ce cas est d'apporter quelques petites modifications à la méthode createRooms de la classe Game pour créer la pièce et réaliser deux appels pour créer les sorties :

```
private void createRooms()
{
  Room outside, theatre, pub, lab, office, cellar;
  ...
  cellar = new Room('dans la cave');
  ...
  office.setExit('bas', cellar);
  cellar.setExit('haut', office);
}
```

Grâce à la nouvelle interface de la classe Room, cela fonctionnera sans difficulté. Ce changement est maintenant très facile à apporter et confirme que nous avons amélioré la conception.

D'autres faits liés à cette meilleure conception peuvent être mis en évidence si nous comparons la version originale de la méthode printLocationInfo présentée au code 7.2 avec la méthode getExitString présentée au code 7.6 qui est une solution de l'exercice 7.7.

Code 7.6 • La méthode getExitString de la classe Room.

```
    /**
     * Renvoie une description des sorties d'une pièce,
     * par exemple : 'Sorties : nord ouest '.
     * @return Une description des sorties possibles.
     */
    public String getExitString()
    {
        String exitString = "Sorties :";
        if(northExit != null)
          exitString += 'nord ';
        if(eastExit != null)
          exitString += 'est ';
        if(southExit != null)
          exitString += 'sud ';
        if(westExit != null)
          exitString += 'ouest ';
        return exitString;
    }
```

Comme les informations de sortie sont maintenant stockées par la classe Room elle-même, cette dernière est responsable de la communication de ces informations. L'objet de type Room peut accomplir cette tâche beaucoup mieux que n'importe quel

autre objet, car il possède toute la connaissance sur la structure de stockage interne des sorties. À l'intérieur de la classe `Room`, nous pouvons maintenant tirer parti du fait que les sorties sont stockées dans un objet de type `HashMap` et nous pouvons réaliser une itération sur cet objet pour décrire les sorties.

Par conséquent, nous remplaçons la version de `getExitString` présentée au code 7.6 par la version du code 7.7. Cette méthode trouve tous les noms des sorties dans l'objet de type `HashMap` (ses clés sont les noms des sorties) et les concatène en une seule chaîne `String` qui est ensuite renvoyée (nous devons importer les classes `Set` et `Iterator` de `java.util` pour que cela fonctionne).

Code 7.7 • Une version révisée de la méthode `getExitString`.

```
/**
 * Renvoie une description des sorties d'une pièce,
 * par exemple : "Sorties : nord ouest ".
 * @return Une description des sorties disponibles.
 */
public String getExitString()
{
    String returnString = "Sorties :";
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
}
```

Exercice 7.9

Étudiez la méthode `keySet` dans la documentation de la classe `HashMap`. Que fait-elle ?

Exercice 7.10

Expliquez en détail et par écrit comment fonctionne la méthode `getExitString` présentée au code 7.7