Création d'extensions

Le projet *zuul-bad* fonctionne. Nous pouvons l'exécuter et il réalise correctement tout ce qu'il est censé accomplir. Cependant, il est, sous certains aspects, plutôt mal conçu. Une solution bien conçue opérerait de la même façon. Nous ne noterions aucune différence en exécutant le programme.

Nous constaterons cependant des différences significatives de charge de travail induite par une mauvaise conception, comparée à celle d'une application bien conçue, quand nous essaierons de modifier le projet. Nous allons étudier ce point en apportant quelques modifications au projet. Tout en le modifiant, nous allons illustrer quelques exemples de mauvaise conception quand nous les rencontrerons dans le code source existant, et nous améliorerons la conception des classes avant d'implanter nos extensions.

La tâche

La première tâche que nous allons accomplir est d'ajouter une nouvelle direction de mouvement. Actuellement, un joueur peut se déplacer dans quatre directions : *nord*, *est*, *sud* et *ouest*. Nous voulons tenir compte des bâtiments à plusieurs niveaux (caves ou donjons ou quoi que vous vouliez ajouter plus tard à votre jeu) et définir les directions *haut* et *bas*. Un joueur peut alors taper « aller bas » pour se déplacer dans une cave, par exemple.

Découverte du code source pertinent

L'étude des classes fournies met en évidence qu'au moins deux classes sont impliquées dans cette modification : `Room` et `Game`.

`Room` est la classe qui mémorise (entre autres) les sorties de chaque pièce. Comme nous l'avons vu dans le code 7.1, cette information est utilisée par la classe `Game` pour afficher les informations concernant les sorties et pour se déplacer d'une pièce à une autre.

La classe `Room` est assez petite. Le code 7.3 présente son code source. En lisant ce code, nous pouvons remarquer que les sorties sont mentionnées à deux endroits : elles sont listées en tant que champ en haut de la classe et elles sont modifiées par la méthode `setExits`. Pour ajouter deux nouvelles directions, nous devrions ajouter deux nouvelles sorties (`upExit` et `downExit`) à ces deux endroits.

Code 7.3 • Code source de la classe (mal conçue) `Room`.

```
public class Room
{
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;

    /**
     * Crée une pièce décrite par la chaîne 'description'.
     * Au départ, il n'existe aucune sortie.
     * "description" est une chaîne comme "une cuisine" ou
     * "une cour de jardin".
     */
    public Room(String description)
    {
        this.description = description;
    }
```

```
    /**
     * Définit les sorties de cette pièce.
     * Chaque direction soit conduit à une autre pièce,
     * soit est null (il n'y a pas de sortie dans cette
     * direction).
     */
    public void setExits(Room north, Room east, Room south,
                         Room west)
    {
        if(north != null) {
            northExit = north;
        }
        if(east != null) {
            eastExit = east;
        }
        if(south != null) {
            southExit = south;
        }
        if(west != null) {
            westExit = west;
        }
    }

    /**
     * Renvoie la description de la pièce
     * (telle que définie par le constructeur).
     */
    public String getDescription()
    {
        return description;
    }

}
```

Trouver tous les endroits pertinents dans la classe `Game` demande un peu plus de travail. Le code source est plus long (il n'est pas présenté entièrement ici), et découvrir tous les endroits pertinents demande de la patience et de l'attention.

En lisant le code 7.1, nous pouvons remarquer que la classe `Game` utilise intensivement les informations de sortie d'une pièce. Un objet de type `Game` conserve une référence vers une pièce à l'aide de la variable `currentRoom` et accède fréquemment aux informations de sortie relatives à cette pièce :

- La méthode `createRoom` définit les sorties.
- Dans la méthode `printWelcome`, les sorties de la pièce courante sont affichées afin de permettre au joueur de savoir comment se déplacer au début du jeu.
- Dans la méthode `goRoom`, les sorties sont utilisées pour identifier la prochaine pièce. Elles sont alors de nouveau utilisées pour afficher les sorties de la pièce dans laquelle nous venons juste d'entrer.

Si nous souhaitons maintenant ajouter deux nouvelles directions de sortie, nous devrons ajouter *haut* et *bas* comme options dans toutes ces méthodes. Lisez cependant le prochain paragraphe avant de le faire.

Couplage

La multiplicité d'endroits où toutes les sorties sont énumérées est symptomatique d'une conception maladroite de classe. Quand les variables de sortie sont définies au niveau de la classe `Room`, nous avons besoin de définir une variable par sortie ; dans la méthode `setExits`, une conditionnelle est définie par sortie ; il en est de même dans la méthode `goRoom` ainsi que dans la méthode `printLocationInfo`, et ainsi de suite. Ces choix de conception sont maintenant à l'origine d'une charge de travail pour nous : quand nous ajoutons de nouvelles sorties, nous devons identifier tous ces endroits et ajouter deux nouveaux cas. Imaginez ce qui se passerait si nous décidions d'utiliser des directions telles que nord-ouest, sud-est, etc. !

Pour améliorer cette situation, nous décidons d'utiliser un objet de type `HashMap` pour stocker les sorties plutôt que des variables distinctes. En procédant ainsi, nous devrions être en mesure d'écrire un code capable de gérer un nombre quelconque de sorties sans trop de modifications. L'objet de type `HashMap` contiendra un lien entre un nom de direction (par exemple « nord ») et la pièce associée à cette direction (un objet de type `Room`). Chaque entrée a ainsi une clé de type chaîne de caractères et un objet de type `Room` comme valeur.

C'est un changement dans la manière utilisée par une pièce pour stocker en interne les informations concernant les pièces voisines. En théorie, cette modification devrait toucher uniquement l'*implantation* de la classe `Room` (*comment* les informations de sorties sont stockées), et non l'*interface* (*ce que* stocke la pièce).

De façon idéale, quand seule l'implantation d'une classe change, les autres classes ne devraient pas être affectées. Cela devrait être le cas quand le couplage est faible.

Dans notre exemple, cela ne fonctionne pas. Si nous supprimons les variables de sortie de la classe `Room` et si nous les remplaçons par un objet de type `HashMap`, la classe `Game` ne pourra plus être compilée. Elle fait de nombreuses références aux variables de sortie qui seront toute à l'origine d'erreurs.

Nous nous rendons compte que nous avons ici à faire à un cas de couplage fort. Afin de résoudre ce problème, nous allons d'abord découpler ces classes avant d'introduire l'objet de type `HashMap`.

Utilisation de l'encapsulation pour réduire le couplage

L'un des principaux problèmes dans cet exemple est l'utilisation de champs publics. Les champs de sortie de la classe `Room` ont tous été déclarés publics. Par conséquent, la classe `Game` peut accéder directement à ces champs (et elle utilise intensivement cette propriété). En rendant ces champs publics, la classe `Room` a exposé dans son interface non seulement le fait qu'elle possède des sorties, mais aussi la manière

dont les informations de sorties sont stockées. Elle viole ainsi un principe fondamental d'une bonne conception de classe : l'*encapsulation*.

Concept

Une **encapsulation** adéquate dans les classes réduit le couplage et conduit ainsi à une meilleure conception.

Le principe de l'encapsulation (cacher les informations liées à l'implantation) suggère que seules les informations au sujet de *ce que* peut faire une classe doivent être visibles de l'extérieur, et non pas *comment* la classe rend ces services. Cela présente un avantage important : si aucune classe ne connaît comment nos informations sont stockées, nous pouvons alors facilement modifier leur stockage sans perturber les autres classes.

Nous pouvons renforcer la séparation entre le *ce que* et le *comment* en rendant les champs privés et en utilisant des accesseurs pour y accéder. La classe `Room` après cette première modification est présentée au code 7.4.

Code 7.4 • Utilisation d'un accesseur pour réduire le couplage.

```
public class Room
{
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;

    // méthodes existantes non modifiées

    public Room getExit(String direction)
    {
        if(direction.equals("nord")) {
            return northExit;
        }
        if(direction.equals("est")) {
            return eastExit;
        }
        if(direction.equals("sud")) {
            return southExit;
        }
        if(direction.equals("ouest")) {
            return westExit;
        }
        return null;
    }
}
```

Une fois la classe `Room` modifiée, nous devons également modifier la classe `Game`. Partout où l'on accède à une variable de sortie, nous utilisons maintenant l'accesseur. Par exemple, au lieu d'écrire :

```
nextRoom = currentRoom.eastExit;
```

nous écrivons maintenant :

```
nextRoom = currentRoom.getExit('est');
```

Cette modification simplifie en outre une partie de la classe `Game`. Dans la méthode `goRoom`, le remplacement suggéré ici aboutit à la partie de code suivante :

```
Room nextRoom = null;
if(direction.equals('nord')) {
    nextRoom = currentRoom.getExit("nord");
}
if(direction.equals('est')) {
    nextRoom = currentRoom.getExit("est");
}
if(direction.equals("sud")) {
    nextRoom = currentRoom.getExit('sud");
}
if(direction.equals("ouest")) {
    nextRoom = currentRoom.getExit("ouest');
}
```

Cette partie entière de code peut maintenant être remplacée par :

```
Room nextRoom = currentRoom.getExit(direction);
```