Chapitre

8

Concevoir des cours

Principaux concepts abordés dans ce chapitre :

- conception axée sur la responsabilité
- cohésion
- couplage
- refactorisation

Constructions Java abordées dans ce chapitre :

types énumérés, commutateur

Dans ce chapitre, nous examinons certains des facteurs qui influencent la conception d'une classe. Qu'est-ce qui fait qu'une conception de classe est bonne ou mauvaise ? Écrire de bonnes classes peut demander plus d'efforts à court terme qu'écrire de mauvaises classes, mais à long terme, cet effort supplémentaire sera presque toujours justifié. Pour nous aider à écrire de bonnes classes, il y a quelques principes que nous pouvons suivre. En particulier, nous introduisons le point de vue selon lequel la conception des classes doit être axée sur la responsabilité et que les classes doivent encapsuler leurs données.

Ce chapitre est, comme beaucoup de chapitres précédents, structuré autour d'un projet. Il peut être étudié simplement en le lisant et en suivant notre ligne d'argumentation, ou il peut être étudié beaucoup plus en profondeur en faisant les exercices du projet en parallèle tout en travaillant sur le chapitre.

Le travail du projet est divisé en trois parties. Dans la première partie, nous discutons des changements nécessaires au code source et développons et montrons des solutions complètes aux exercices.
La solution pour cette partie est également disponible dans un projet accompagnant ce livre. La deuxième partie suggère plus de changements et d'extensions, et nous discutons des solutions possibles à un niveau élevé (le niveau de conception de classe), mais laissons aux lecteurs le soin de faire le travail de niveau inférieur et de terminer l'implémentation. La troisième partie propose encore plus d'améliorations sous forme d'exercices. Nous ne donnons pas de solutions - les exercices appliquent le matériel discuté tout au long du chapitre.

La mise en œuvre de toutes les parties constitue un bon projet de programmation sur plusieurs semaines. Il peut également être utilisé avec beaucoup de succès en tant que projet de groupe.

8.1 Présentation

Il est possible d'implémenter une application et de lui faire exécuter sa tâche avec des classes mal conçues. Le simple fait d'exécuter une application terminée n'indique généralement pas si elle est bien structurée en interne ou non.

Les problèmes surviennent généralement lorsqu'un programmeur de maintenance souhaite apporter des modifications à une application existante. Si, par exemple, un programmeur tente de corriger un bogue ou souhaite ajouter de nouvelles fonctionnalités à un programme existant, une tâche qui pourrait être facile et évidente avec des classes bien conçues peut très bien être très difficile et impliquer beaucoup de travail si le les classes sont mal conçues.

Dans les applications plus importantes, cet effet se produit plus tôt, lors de la mise en œuvre initiale. Si l'implémentation commence avec une mauvaise structure, la terminer peut ensuite devenir trop complexe et le programme complet peut soit ne pas être terminé, contenir des bogues, soit prendre beaucoup plus de temps à construire. En réalité, les entreprises entretiennent, étendent et vendent souvent une application pendant de nombreuses années. Il n'est pas rare qu'une implémentation d'un logiciel que nous pouvons acheter aujourd'hui dans un magasin de logiciels ait été lancée il y a plus de dix ans. Dans cette situation, un éditeur de logiciels ne peut pas se permettre d'avoir un code mal structuré.

Étant donné que de nombreux effets d'une mauvaise conception de classe deviennent plus évidents lorsque vous essayez d'adapter ou d'étendre une application, nous ferons exactement cela. Dans ce chapitre, nous utiliserons un exemple appelé world-of-zuul, qui est une implémentation rudimentaire d'un jeu d'aventure textuel. Dans son état d'origine, le jeu n'est pas vraiment très ambitieux, d'une part, il est incomplet. À la fin de ce chapitre, cependant, vous serez en mesure d'exercer votre imagination, de concevoir et de mettre en œuvre votre propre jeu et de le rendre vraiment amusant et intéressant.

world-of-zuul Notre jeu world-of-zuul est calqué sur le jeu d'aventure original qui a été développé au début des années 1970 par Will Crowther et étendu par Don Woods. Le jeu original est aussi parfois connu sous le nom de Colossal Cave Adventure. C'était un jeu merveilleux, entièrement imaginatif et sophistiqué pour l'époque, impliquant de trouver son chemin à travers un système complexe de grottes, de localiser des trésors cachés, d'utiliser des mots secrets et d'autres mystères, le tout dans le but de marquer le maximum de points. Vous pouvez en savoir plus à ce sujet sur des sites tels que http://jerz.setonhill.edu/if/canon/Adventure.htm et http://www.rickadams.org/adventure/, ou essayez de faire une recherche sur le Web pour "Colossal Aventure dans la grotte."

Pendant que nous travaillons sur l'extension de l'application d'origine, nous profiterons de l'occasion pour discuter des aspects de la conception de sa classe existante. Nous verrons que l'implémentation avec laquelle nous commençons contient des exemples de mauvaise conception, et nous pourrons voir comment cela impacte nos tâches et comment nous pouvons les corriger.

Dans les exemples de projets de ce livre, vous trouverez deux versions du projet zuul : zuul-bad et zuul-better. Les deux implémentent exactement la même fonctionnalité, mais une partie de la structure de classe est différente, représentant une mauvaise conception dans un projet et une meilleure conception dans

L'autre. Le fait que nous puissions implémenter la même fonctionnalité dans le bon sens ou dans le mauvais sens illustre le fait qu'une mauvaise conception n'est généralement pas la conséquence d'un problème difficile à résoudre. Une mauvaise conception a plus à voir avec les décisions que nous prenons lors de la résolution d'un problème particulier. Nous ne pouvons pas utiliser l'argument selon lequel il n'y avait pas d'autre moyen de résoudre le problème comme excuse pour une mauvaise conception.

Nous allons donc utiliser le projet avec une mauvaise conception afin de pouvoir explorer pourquoi il est mauvais, puis l'améliorer. La meilleure version est une implémentation des modifications dont nous discutons ici.

Exercice 8.1 Ouvrez le projet zuul-bad. (Ce projet est appelé "mauvais" car sa mise en œuvre contient de mauvaises décisions de conception, et nous ne voulons laisser aucun doute sur le fait que cela ne doit pas être utilisé comme exemple de bonne pratique de programmation !) Exécutez et explorez l'application. Le commentaire du projet vous donne des informations sur la façon de l'exécuter.

Tout en explorant l'application, répondez aux questions suivantes : ■■ Que fait cette application ? ■■ Quelles commandes le jeu accepte-t-il ? ■■ Que fait chaque commande ? ■■ Combien y a-t-il de pièces dans le scénario ? ■■

Dessinez un plan des pièces existantes.

Exercice 8.2 Une fois que vous savez ce que fait l'ensemble de l'application, essayez de découvrir ce que fait chaque classe individuelle. Notez pour chaque classe son objectif. Vous devez regarder le code source pour le faire. Notez que vous ne comprenez peut-être pas (et n'avez pas besoin) tout le code source. Souvent, il suffit de lire les commentaires et de regarder les en-têtes de méthode.

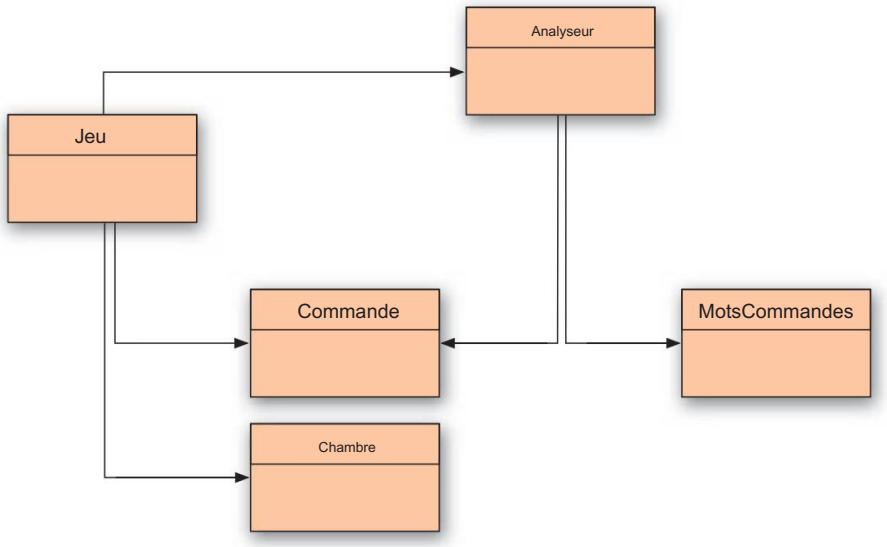
8.2 L'exemple du jeu world-of-zuul

À partir de l'exercice 8.1, vous avez vu que le jeu zuul n'est pas encore très aventureux. Il est, en fait, assez ennuyeux dans son état actuel. Mais cela nous fournit une bonne base pour concevoir et implémenter notre propre jeu, qui, espérons-le, sera plus intéressant.

Nous commençons par analyser les classes qui sont déjà présentes dans notre première version et essayons de découvrir ce qu'elles font. Le diagramme de classes est illustré à la figure 8.1.

Le projet montre cinq classes. Ce sont Parser, CommandWords, Command, Room et Game. Une enquête sur le code source montre, heureusement, que ces classes sont assez bien documentées, et nous pouvons avoir un premier aperçu de ce qu'elles font en lisant simplement le commentaire de classe en haut de chaque classe. (Ce fait sert également à illustrer qu'une mauvaise conception implique quelque chose de plus profond que simplement l'apparence d'une classe ou la qualité de sa documentation.) Notre compréhension du jeu sera facilitée par l'examen du code source.

Figure 8.1
Diagramme de classes Zuul



pour voir quelles méthodes chaque classe possède et ce que certaines méthodes semblent faire. Ici, nous résumons le but de chaque classe :

■■ CommandWords La classe CommandWords définit toutes les commandes valides dans le jeu. Pour ce faire, il contient un tableau d' objets String représentant les mots de commande.

■■ Analyseur L'analyseur lit les lignes d'entrée du terminal et essaie de les interpréter comme des commandes. Il crée des objets de classe Command qui représentent la commande qui a été entrée.

■■ Command Un objet Command représente une commande entrée par l'utilisateur. Il a des méthodes qui nous permettent de vérifier facilement s'il s'agissait d'une commande valide et d'obtenir les premier et deuxième mots de la commande sous forme de chaînes distinctes.

■■ Pièce Un objet Pièce représente un emplacement dans un jeu. Les chambres peuvent avoir des sorties qui mènent à autres pièces.

■■ Jeu La classe Jeu est la classe principale du jeu. Il configure le jeu puis entre dans une boucle pour lire et exécuter des commandes. Il contient également le code qui implémente chaque commande utilisateur.

Exercice 8.3 Concevez votre propre scénario de jeu. Faites cela loin de l'ordinateur.
Ne pensez pas à la mise en œuvre, aux classes ou même à la programmation en général. Pensez à inventer un jeu intéressant. Cela pourrait être fait avec un groupe de personnes.

Le jeu peut être tout ce qui a pour structure de base un joueur se déplaçant à travers différents endroits. Voici quelques exemples : ■■ Vous êtes un globule blanc voyageant

dans le corps à la recherche de virus pour
attaque . . .

■■ Vous êtes perdu dans un centre commercial et devez trouver la sortie. . . ■■
Vous êtes une taupe dans son terrier et vous ne vous souvenez plus où vous avez stocké votre
réserves alimentaires avant l'hiver. . .

■■ Vous êtes un aventurier qui cherche dans un donjon plein de monstres et
autres personnages . . .

■■ Vous faites partie de l'escouade anti-bombes et devez trouver et désamorcer une bombe avant qu'elle ne parte
désactivé . . .

Assurez-vous que votre partie a un but (afin qu'elle ait une fin et que le joueur puisse « gagner »). Essayez de penser à beaucoup de choses pour rendre le jeu intéressant (trappes, objets magiques, personnages qui ne vous aident que si vous les nourrissez, limites de temps... tout ce que vous voulez). Laissez courir votre imagination.

À ce stade, ne vous inquiétez pas de la façon de mettre en œuvre ces choses.

8.3 Introduction au couplage et à la cohésion

Concept

Le terme couplage décrit l'interconnexion des classes. Nous nous efforçons d'obtenir un couplage lâche dans un système, c'est-à-dire un système où chaque classe est largement indépendante et communique avec les autres classes via une petite interface bien définie.

Si nous voulons justifier notre affirmation selon laquelle certaines conceptions sont meilleures que d'autres, nous devons définir certains termes qui nous permettront de discuter des problèmes que nous considérons comme importants dans la conception de classe. Deux termes sont centraux lorsqu'on parle de la qualité d'une conception de classe : couplage et cohésion.

Le terme couplage fait référence à l'interdépendance des classes. Nous avons déjà expliqué dans les chapitres précédents que nous visons à concevoir nos applications comme un ensemble de classes coopérantes qui communiquent via des interfaces bien définies. Le degré de couplage indique à quel point ces classes sont étroitement liées. Nous nous efforçons d'obtenir un faible degré de couplage, ou un couplage lâche.

Le degré de couplage détermine à quel point il est difficile d'apporter des modifications à une application. Dans une structure de classe étroitement couplée, un changement dans une classe peut rendre nécessaire le changement de plusieurs autres classes également. C'est ce que nous essayons d'éviter, car l'effet d'un petit changement peut rapidement se répercuter sur une application complète. De plus, trouver tous les endroits où des changements sont nécessaires et effectuer les changements peut s'avérer difficile et prendre du temps.

Dans un système faiblement couplé, en revanche, nous pouvons souvent changer une classe sans apporter de modifications aux autres classes, et l'application fonctionnera toujours. Nous aborderons des exemples particuliers de couplage serré et lâche dans ce chapitre.

Le terme cohésion fait référence au nombre et à la diversité des tâches dont une seule unité d'une application est responsable. La cohésion est pertinente pour les unités d'une seule classe et d'une méthode individuelle.[1]

[1] Nous utilisons aussi parfois le terme module (ou package en Java) pour désigner une unité multi-classes. La cohésion est également pertinente à ce niveau.

Concept

Le terme cohésion décrit dans quelle mesure une unité de code correspond à une tâche ou une entité logique. Dans un système hautement cohérent, chaque unité de code (méthode, classe ou module) est responsable d'une tâche ou d'une entité bien définie. Une bonne conception de classe présente un degré élevé de cohésion.

Idéalement, une unité de code devrait être responsable d'une tâche cohérente (c'est-à-dire une tâche qui peut être considérée comme une unité logique). Une méthode doit implémenter une opération logique et une classe doit représenter un type d'entité. La raison principale derrière le principe de cohésion est la réutilisation : si une méthode ou une classe est responsable d'une seule chose bien définie, alors il est beaucoup plus probable qu'elle puisse être réutilisée dans un contexte différent. Un avantage complémentaire à suivre ce principe est que, lorsqu'il est nécessaire de modifier certains aspects d'une application, nous sommes susceptibles de trouver toutes les pièces pertinentes situées dans la même unité.

Nous discuterons avec des exemples ci-dessous comment la cohésion influence la qualité de la conception des classes.

Exercice 8.4 Dessinez (sur papier) une carte pour le jeu que vous avez inventé dans l'exercice 8.3. Ouvrez le projet zuul-bad et enregistrez-le sous un nom différent (par exemple, zuul). C'est le projet que vous utiliserez pour apporter des améliorations et des modifications tout au long de ce chapitre. Vous pouvez omettre le suffixe -bad , car il ne sera (espérons-le) bientôt plus aussi mauvais.

Dans un premier temps, modifiez la méthode createRooms dans la classe Game pour créer les salles et les sorties que vous avez inventées pour votre jeu.

8.4 Duplication de codes

La duplication de code est un indicateur de mauvaise conception. La classe Game présentée dans le code 8.1 contient un cas de duplication de code. Le problème avec la duplication de code est que toute modification d'une version doit également être apportée à une autre si l'on veut éviter les incohérences. Cela augmente la quantité de travail qu'un programmeur de maintenance doit effectuer et introduit le danger de bogues. Il arrive très facilement qu'un programmeur de maintenance trouve une copie du code et, après l'avoir modifié, suppose que le travail est terminé. Rien n'indique qu'une deuxième copie du code existe, et il peut rester inchangé à tort.

Code 8.1 Sections sélectionnées de le (mal conçu) Classe de jeu

```
public class Game
{
    Some code omitted.
    private void createRooms()
    {
        Room outside, theater, pub, lab, office;

        // create the rooms
        outside = new Room("outside the main entrance of the university");
        theater = new Room("in a lecture theater");
        pub = new Room("in the campus pub");
        lab = new Room("in a computing lab");
        office = new Room("in the computing admin office");
```

Code 8.1
suite
Sections sélectionnées de
le (mal conçu)
Classe de jeu

Concept

La duplication de code (avoir plusieurs fois le même segment de code dans une application) est un signe de mauvaise conception. Il devrait être évité.

```
        // initialize room exits
        outside.setExits(null, theater, lab, pub);
        theater.setExits(null, null, null, outside);
        pub.setExits(null, outside, null, null);
        lab.setExits(outside, office, null, null);
        office.setExits(null, null, null, lab);

        currentRoom = outside;  // start game outside
    }

    Some code omitted.

    /**
     * Print out the opening message for the player.
     */
    private void printWelcome()
    {
        System.out.println();
        System.out.println("Welcome to the World of Zuul!");
        System.out.println("Zuul is a new incredibly boring adventure game.");
        System.out.println("Type 'help' if you need help.");
        System.out.println();
        System.out.println("You are " + currentRoom.getDescription());
        System.out.print("Exits: ");
        if(currentRoom.northExit != null) {
            System.out.print("north ");
        }
        if(currentRoom.eastExit != null) {
            System.out.print("east ");
        }
        if(currentRoom.southExit != null) {
            System.out.print("south ");
        }
        if(currentRoom.westExit != null) {
            System.out.print("west ");
        }
        System.out.println();
    }

    Some code omitted.

    /**
     * Try to go in one direction. If there is an exit, enter
     * the new room, otherwise print an error message.
     */
    private void goRoom(Command command)
    {
        if(!command.hasSecondWord()) {
            // if there is no second word, we don't know where to go...
            System.out.println("Go where?");
            return;
        }
```

Code 8.1 suite
Sections sélectionnées de le (mal conçu) Classe de jeu

```
        String direction = command.getSecondWord();

        // Try to leave current room.
        Room nextRoom = null;
        if(direction.equals("north")) {
            nextRoom = currentRoom.northExit;
        }
        if(direction.equals("east")) {
            nextRoom = currentRoom.eastExit;
        }
        if(direction.equals("south")) {
            nextRoom = currentRoom.southExit;
        }
        if(direction.equals("west")) {
            nextRoom = currentRoom.westExit;
        }

        if (nextRoom == null) {
            System.out.println("There is no door!");
        }
        else {
            currentRoom = nextRoom;
            System.out.println("You are " + currentRoom.getDescription());
            System.out.print("Exits: ");
            if(currentRoom.northExit != null) {
                System.out.print("north ");
            }
            if(currentRoom.eastExit != null) {
                System.out.print("east ");
            }
            if(currentRoom.southExit != null) {
                System.out.print("south ");
            }
            if(currentRoom.westExit != null) {
                System.out.print("west ");
            }
            System.out.println();
        }
    }

    Some code omitted.
}
```

Les méthodes printWelcome et goRoom contiennent les lignes de code suivantes :

```
System.out.println("Vous êtes" + currentRoom.getDescription());
System.out.print("Sortie :");
if(currentRoom.northExit != null)
        { System.out.print("north");

} if(currentRoom.eastExit != null)
        { System.out.print("east");
}
```

```
if(currentRoom.southExit != null)
        { System.out.print("south");

} if(currentRoom.westExit != null)
        { System.out.print("west");
}
System.out.println();
```

La duplication de code est généralement le symptôme d'une mauvaise cohésion. Le problème ici a ses racines dans le fait que les deux méthodes en question font deux choses : printWelcome imprime le message de bienvenue et imprime les informations sur l'emplacement actuel, tandis que goRoom change l'emplacement actuel puis imprime les informations sur le (nouveau) emplacement actuel.

Les deux méthodes affichent des informations sur l'emplacement actuel, mais aucune ne peut appeler l'autre, car elles font aussi d'autres choses. C'est une mauvaise conception.

Une meilleure conception utiliserait une méthode distincte et plus cohérente dont la seule tâche est d'imprimer les informations de localisation actuelles (code 8.2). Les méthodes printWelcome et goRoom peuvent alors appeler cette méthode lorsqu'elles ont besoin d'imprimer ces informations. De cette façon, on évite d'écrire le code deux fois, et quand on a besoin de le changer, on n'a besoin de le changer qu'une seule fois.

Code 8.2
printEmplacement
Info en tant que méthode distincte

```
private void printLocationInfo()
{
    System.out.println("You are " + currentRoom.getDescription());
    System.out.print("Exits: ");
    if(currentRoom.northExit != null) {
        System.out.print("north ");
    }
    if(currentRoom.eastExit != null) {
        System.out.print("east ");
    }
    if(currentRoom.southExit != null) {
        System.out.print("south ");
    }
    if(currentRoom.westExit != null) {
        System.out.print("west ");
    }
    System.out.println();
}
```

Exercice 8.5 Implémentez et utilisez une méthode distincte printLocationInfo dans votre projet, comme indiqué dans cette section. Testez vos modifications.

8.5 Faire des extensions Le projet zuul-bad fonctionne.

Nous pouvons l'exécuter, et il fait correctement tout ce qu'il était censé faire. Cependant, il est à certains égards assez mal conçu. Une alternative bien conçue fonctionnerait de la même manière ; nous ne remarquerions aucune différence simplement en exécutant le programme.

Cependant, une fois que nous essaierons d'apporter des modifications au projet, nous remarquerons des différences significatives dans la quantité de travail impliquée dans la modification d'un code mal conçu, par rapport aux modifications apportées à une application bien conçue. Nous étudierons cela en apportant quelques modifications au projet. Pendant que nous faisons cela, nous discuterons des exemples de mauvaise conception lorsque nous les verrons dans la source existante, et nous améliorerons la conception de la classe avant d'implémenter nos extensions.

8.5.1 La tâche

La première tâche que nous tenterons est d'ajouter une nouvelle direction de mouvement. Actuellement, un joueur peut se déplacer dans quatre directions : nord, est, sud et ouest. Nous voulons autoriser les bâtiments à plusieurs niveaux (ou caves, ou donjons, ou tout ce que vous souhaitez ajouter ultérieurement à votre jeu) et additionner les directions possibles . Un joueur peut alors taper "descendre" pour descendre, disons, dans une cave.

8.5.2 Trouver le code source pertinent

L'inspection des classes données nous montre qu'au moins deux classes sont impliquées dans ce changement : Room et Game.

Room est la classe qui stocke (entre autres) les sorties de chaque room. Comme nous l'avons vu dans le code 8.1, dans la classe Game , les informations de sortie de la salle actuelle sont utilisées pour imprimer des informations sur les sorties et pour se déplacer d'une salle à l'autre.

La classe Room est assez courte. Son code source est présenté dans le Code 8.3. En lisant la source, nous pouvons voir que les sorties sont mentionnées à deux endroits différents : elles sont répertoriées sous forme de champs en haut de la classe et elles sont affectées dans la méthode setExits . Pour ajouter deux nouvelles directions, il faudrait ajouter deux nouvelles sorties (upExit et downExit) à ces deux endroits.

Code 8.3
Code source du (mal conçu)
Classe de chambre

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

    /**
     * Create a room described "description."  Initially, it has
     * no exits. "description" is something like "a kitchen" or
     * "an open court yard."
     * @param description The room's description.
     */
    public Room(String description)
    {
        this.description = description;
    }
```

Code 8.3
suite
Code source du
(mal conçu)
Classe de chambre

```
    /**
     * Define the exits of this room.  Every direction either leads
     * to another room or is null (no exit there).
     * @param north The north exit.
     * @param east The east east.
     * @param south The south exit.
     * @param west The west exit.
     */
    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;
        }
    }

    /**
     * @return The description of the room.
     */
    public String getDescription()
    {
        return description;
    }
}
```

C'est un peu plus de travail pour trouver tous les endroits pertinents dans la classe Jeu . Le code source est un peu plus long (il n'est pas entièrement montré ici), et trouver tous les endroits pertinents demande de la patience et du soin.

En lisant le code montré dans le Code 8.1, nous pouvons voir que la classe Game fait un usage intensif des informations de sortie d'une pièce. L' objet Game contient une référence à une salle dans la variable currentRoom et accède fréquemment aux informations de sortie de cette salle.

- Dans la méthode createRoom , les sorties sont définies.
- Dans la méthode printWelcome , les sorties de la salle actuelle sont imprimées afin que le joueur sache où aller lorsque le jeu commence.
- Dans la méthode goRoom , les sorties sont utilisées pour trouver la pièce suivante. Ils sont ensuite réutilisés pour imprimer les sorties de la pièce suivante dans laquelle nous venons d'entrer.

Si nous voulons maintenant ajouter deux nouvelles directions de sortie, nous devrons ajouter les options haut et bas à tous ces endroits. Cependant, lisez la section suivante avant de faire cela.

8.6 Couplage

Le fait qu'il y ait tant d'endroits où toutes les sorties sont énumérées est symptomatique d'une mauvaise conception des classes. Lors de la déclaration des variables de sortie dans la classe Room , nous devons lister une variable par sortie ; dans la méthode setExits , il y a une instruction if par sortie ; dans la méthode goRoom , il y a une instruction if par sortie ; dans la méthode printLocationInfo , il y a une instruction if par sortie ; et ainsi de suite. Cette décision de conception crée maintenant du travail pour nous : lors de l'ajout de nouvelles sorties, nous devons trouver tous ces endroits et ajouter deux nouveaux cas. Imaginez l'effet si nous décidions d'utiliser des directions telles que le nord-ouest, le sud-est, etc. !

Pour améliorer la situation, nous décidons d'utiliser un HashMap pour stocker les sorties, plutôt que des variables séparées. En faisant cela, nous devrions être capables d'écrire du code qui peut faire face à n'importe quel nombre de sorties et ne nécessite pas autant de modifications. Le HashMap contiendra un mappage d'une direction nommée (par exemple, "nord") à la pièce qui se trouve dans cette direction (un objet Room). Ainsi, chaque entrée a une chaîne comme clé et un objet Room comme valeur.

Il s'agit d'un changement dans la façon dont une pièce stocke en interne des informations sur les pièces voisines. Théoriquement, il s'agit d'un changement qui ne devrait affecter que l' implémentation de la classe Room (comment les informations de sortie sont stockées), pas l' interface (ce que la room stocke).

Idéalement, lorsque seule l'implémentation d'une classe change, les autres classes ne devraient pas être affectées. Ce serait un cas de couplage lâche .

Dans notre exemple, cela ne fonctionne pas. Si nous supprimons les variables de sortie de la classe Room et les remplaçons par un HashMap, la classe Game ne compilera plus. Il fait de nombreuses références aux variables de sortie de la salle, qui provoqueraient toutes des erreurs.

Nous voyons que nous avons ici un cas de couplage étroit . Afin de nettoyer cela, nous allons découpler ces classes avant d'introduire le HashMap.

8.6.1 Utilisation de l'encapsulation pour réduire le couplage

Concept

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

L'un des principaux problèmes de cet exemple est l'utilisation de champs publics. Les champs de sortie de la classe Room ont tous été déclarés publics. De toute évidence, le programmeur de cette classe n'a pas suivi les directives que nous avons énoncées plus haut dans ce livre ("Ne jamais rendre les champs publics !"). Nous allons maintenant voir le résultat. La classe Game dans cet exemple peut accéder directement à ces champs (et elle utilise largement ce fait). En rendant les champs publics, la classe Room a exposé dans son interface non seulement le fait qu'elle a des sorties, mais aussi exactement comment les informations de sortie sont stockées. Cela enfreint l'un des principes fondamentaux d'une bonne conception de classe : l'encapsulation.

La directive d'encapsulation (cacher les informations d'implémentation de la vue) suggère que seules les informations sur ce qu'une classe peut faire doivent être visibles de l'extérieur, pas sur la façon dont elle le fait. Cela a un grand avantage : si aucune autre classe ne sait comment nos informations sont stockées, alors nous pouvons facilement changer la façon dont elles sont stockées sans casser les autres classes.

Nous pouvons imposer cette séparation entre quoi et comment en rendant les champs privés et en utilisant une méthode d'accès pour y accéder. La première étape de notre classe Room modifiée est illustrée dans le code 8.4.

Code 8.4

Utilisation d'une méthode d'accès pour diminuer le couplage

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

    Existing methods unchanged.

    public Room getExit(String direction)
    {
        if(direction.equals("north")) {
            return northExit;
        }
        if(direction.equals("east")) {
            return eastExit;
        }
        if(direction.equals("south")) {
            return southExit;
        }
        if(direction.equals("west")) {
            return westExit;
        }
        return null;
    }
}
```

Après avoir apporté cette modification à la classe Room , nous devons également modifier la classe Game . Partout où une variable de sortie a été accédée, nous utilisons maintenant la méthode d'accès. Par exemple, au lieu d'écrire

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

nous écrivons maintenant

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

Cela rend également le codage d'une section de la classe Game beaucoup plus facile. Dans la méthode goRoom , le remplacement suggéré ici se traduira par le segment de code suivant :

```
Salle nextRoom = null;
if(direction.equals("north")) { nextRoom =
        currentRoom.getExit("nord");

} if(direction.equals("east")) { nextRoom
        = currentRoom.getExit("east");

} if(direction.equals("south")) { nextRoom
        = currentRoom.getExit("south");
}
```

```
if(direction.equals("west")) { nextRoom
      = currentRoom.getExit("west");
}
```

Au lieu de cela, tout ce segment de code peut maintenant être remplacé par :

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

Exercice 8.6 Apportez les modifications que nous avons décrites aux classes Room et Game .

Exercice 8.7 Apportez une modification similaire à la méthode printLocationInfo de Game afin que les détails des sorties soient maintenant préparés par la Room plutôt que par Game. Définissez une méthode dans Room avec l'en-tête suivant :

```
/**
 *
    Renvoyer un descriptif des sorties de la salle,
 * par exemple, "Sorties : nord ouest".
 *
    @return Une description des sorties disponibles. */

chaîne publique getExitString()
```

Jusqu'à présent, nous n'avons pas modifié la représentation des sorties dans la classe Room . Nous avons seulement nettoyé l'interface. Le changement dans la classe Game est minime - au lieu d'un accès à un champ public, nous utilisons un appel de méthode - mais le gain est spectaculaire. Nous pouvons maintenant modifier la façon dont les sorties sont stockées dans la pièce, sans avoir à vous soucier de casser quoi que ce soit dans la classe Jeu . La représentation interne dans Room a été complètement découplée de l'interface. Maintenant que la conception est telle qu'elle aurait dû être en premier lieu, il est facile d'échanger les champs de sortie séparés contre un HashMap . Le code modifié est indiqué dans le code 8.5.

Code 8.5
Code source du
Classe de chambre

```
import java.util.HashMap;

Class comment omitted.

public class Room
{
    private String description;
    private HashMap<String, Room> exits;        // stores exits of this room.

    /**
     * Create a room described "description." Initially, it has no exits.
     * "description" is something like "a kitchen" or "an open court yard."
     * @param description The room's description.
     */
    public Room(String description)
    {
        this.description = description;
        exits = new HashMap<>();
    }
```

Code 8.5
a continué
Code source du
Classe de chambre

```
    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbor  The room to which the exit leads.
     */
    public void setExit(Room north, Room east, Room south, Room west)
    {
        if(north != null) {
            exits.put("north", north);
        }
        if(east != null) {
            exits.put("east", east);
        }
        if(south != null) {
            exits.put("south", south);
        }
        if(west != null) {
            exits.put("west", west);
        }
    }

    /**
     * Return the room that is reached if we go from this room in direction
     * "direction."  If there is no room in that direction, return null.
     * @param direction The exit's direction.
     * @return The room in the given direction.
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }

    /**
     * @return The description of the room
     * (the one that was defined in the constructor).
     */
    public String getDescription()
    {
        return description;
    }
}
```

Il convient de souligner à nouveau que nous pouvons effectuer ce changement maintenant sans même vérifier si quelque chose va se casser ailleurs. Étant donné que nous n'avons modifié que les aspects privés de la classe Room , qui, par définition, ne peuvent pas être utilisés dans d'autres classes, ce changement n'a pas d'impact sur les autres classes. L'interface reste inchangée.

Un sous-produit de ce changement est que notre classe Room est maintenant encore plus courte. Au lieu d'énumérer quatre variables distinctes, nous n'en avons qu'une. De plus, la méthode getExit est considérablement simplifiée.

Rappelons que l'objectif initial qui a déclenché cette série de changements était de faciliter l'ajout des deux nouvelles sorties possibles dans le sens montant et descendant . C'est déjà devenu beaucoup plus facile. Parce que nous utilisons maintenant un HashMap pour stocker les sorties, le stockage de ces deux directions supplémentaires fonctionnera sans aucun changement. Nous pouvons également obtenir les informations de sortie via la méthode getExit sans aucun problème.

Le seul endroit où les connaissances sur les quatre sorties existantes (nord, est, sud, ouest) sont encore codées dans la source se trouve dans la méthode setExits . C'est la dernière partie qui doit être améliorée. Pour le moment, l'en-tête de la méthode est

```
public void setExits(Pièce nord, Pièce est, Pièce sud, Pièce ouest)
```

Cette méthode fait partie de l'interface de la classe Room , donc toute modification que nous lui apportons affectera inévitablement d'autres classes en vertu du couplage. On ne peut jamais complètement découpler les classes dans une application ; sinon, les objets de classes différentes ne pourraient pas interagir les uns avec les autres. Nous essayons plutôt de maintenir le degré de couplage aussi bas que possible.
Si nous devons de toute façon modifier setExits pour prendre en charge des directions supplémentaires, notre solution préférée consiste à le remplacer entièrement par cette méthode :

```
/**
 * Définir une sortie de cette pièce.
 *
 * @param direction La direction de la sortie. @param voisin La pièce dans
 la direction donnée. */

public void setExit (direction de la chaîne, voisin de la pièce) {

    exits.put(direction, voisin);
}
```

Maintenant, les sorties de cette pièce peuvent être définies une sortie à la fois, et n'importe quelle direction peut être utilisée pour une sortie. Dans la classe Game , le changement résultant de la modification de l'interface de Room est le suivant. Au lieu d'écrire

```
lab.setExits(dehors, bureau, null, null);
```

nous écrivons maintenant

```
lab.setExit("nord", extérieur); lab.setExit("est",
bureau);
```

Nous avons maintenant complètement supprimé la restriction de Room selon laquelle il ne peut stocker que quatre sorties. La classe Room est maintenant prête à stocker les sorties montantes et descendantes , ainsi que toute autre direction à laquelle vous pourriez penser (nord-ouest, sud-est, etc.).

Exercice 8.8 Implémentez les modifications décrites dans cette section dans votre propre projet zuul .

8.7 Conception axée sur la responsabilité

Nous avons vu dans la section précédente que l'utilisation d'une encapsulation appropriée réduit le couplage et peut réduire considérablement la quantité de travail nécessaire pour apporter des modifications à une application.
L'encapsulation, cependant, n'est pas le seul facteur qui influence le degré de couplage. Un autre aspect est connu sous le terme de conception axée sur la responsabilité.

La conception axée sur la responsabilité exprime l'idée que chaque classe devrait être responsable de la gestion de ses propres données. Souvent, lorsque nous devons ajouter de nouvelles fonctionnalités à une application,

nous devons nous demander dans quelle classe nous devons ajouter une méthode pour implémenter cette nouvelle fonction. Quelle classe devrait être responsable de la tâche ? La réponse est que la classe responsable du stockage de certaines données devrait également être responsable de leur manipulation.

La manière dont la conception axée sur la responsabilité est utilisée influence le degré de couplage et, par conséquent, encore une fois, la facilité avec laquelle une application peut être modifiée ou étendue. Comme d'habitude, nous en discuterons plus en détail avec notre exemple.

8.7.1 Responsabilités et couplage

Concept

La conception axée sur la responsabilité est le processus de conception de classes en attribuant des responsabilités bien définies à chaque classe. Ce processus peut être utilisé pour déterminer quelle classe doit implémenter quelle partie d'une fonction d'application.

Les changements apportés à la classe Room dont nous avons parlé dans la section 8.6.1 permettent maintenant d'ajouter assez facilement les nouvelles directions pour le mouvement vers le haut et vers le bas dans la classe Game . Nous étudions cela avec un exemple. Supposons que nous voulions ajouter une nouvelle pièce (la cave) sous le bureau. Tout ce que nous avons à faire pour y parvenir est d'apporter quelques petites modifications à la méthode createRooms de Game pour créer la salle et de faire deux appels pour définir les sorties :

```
privé void createRooms() {

        Salle à l'extérieur, théâtre, pub, labo, bureau, cave ;
        ...
        cave = new Room("dans la cave");
        ...
        office.setExit("bas", cave);
        cave.setExit("haut", bureau);
}
```

Grâce à la nouvelle interface de la classe Room , cela fonctionnera sans problème. Le changement est maintenant très facile et comme que le design s'améliore.

Une preuve supplémentaire de ceci peut être vue si nous comparons la version originale de la méthode printLocationInfo montrée dans le Code 8.2 avec la méthode getExitString montrée dans le Code 8.6 qui représente une solution à l'Exercice 8.7.

Étant donné que les informations sur ses sorties ne sont désormais stockées que dans la salle elle-même, c'est la salle qui est chargée de fournir ces informations. La salle peut le faire bien mieux que tout autre objet, car elle possède toutes les connaissances sur la structure de stockage interne des données de sortie. Maintenant, à l'intérieur de la classe Room , nous pouvons utiliser la connaissance que les sorties sont stockées dans un HashMap, et nous pouvons parcourir cette carte pour décrire les sorties.

Par conséquent, nous remplaçons la version de getExitString indiquée dans le code 8.6 par la version indiquée dans le code 8.7. Cette méthode trouve tous les noms des sorties dans le HashMap (les clés du HashMap sont les noms des sorties) et les concatène en une seule chaîne, qui est ensuite renvoyée. (Nous devons importer Set depuis java.util pour que cela fonctionne.)

Exercice 8.9 Recherchez la méthode keySet dans la documentation de HashMap.
Qu'est ce que ça fait?

Exercice 8.10 Expliquez, en détail et par écrit, comment fonctionne la méthode getExitString présentée dans le Code 8.7.

Code 8.6
La méthode getExit String de Room

```
/**
 * Return a description of the room's exits,
 * for example, "Exits: north west."
 * @return A description of the available exits.
 */
public String getExitString()
{
    String exitString = "Exits: ";
    if(northExit != null) {
        exitString += "north ";
    }
    if(eastExit != null) {
        exitString += "east ";
    }
    if(southExit != null) {
        exitString += "south ";
    }
    if(westExit != null) {
        exitString += "west ";
    }
    return exitString;
}
```

Code 8.7
Une version révisée de getExitString

```
/**
 * Return a description of the room's exits,
 * for example "Exits: north west."
 * @return A description of the available exits.
 */
public String getExitString()
{
    String returnString = "Exits:";
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
}
```

Notre objectif de réduire le couplage exige que, dans la mesure du possible, les modifications apportées à la classe Room ne nécessitent pas de modifications de la classe Game . Nous pouvons encore améliorer cela.

Actuellement, nous avons toujours encodé dans la classe Game la connaissance que les informations que nous voulons d'une pièce se composent d'une chaîne de description et de la chaîne de sortie :

```
System.out.println("Vous êtes                    " + currentRoom.getDescription());
System.out.println(currentRoom.getExitString());
```

Et si nous ajoutions des objets aux pièces de notre jeu ? Ou des monstres ? Ou d'autres joueurs ?

Lorsque nous décrivons ce que nous voyons, la liste des objets, des monstres et des autres joueurs doit être incluse dans la description de la pièce. Nous aurions besoin non seulement d'apporter des modifications à la classe Room pour ajouter ces éléments, mais également de modifier le segment de code au-dessus de l'endroit où la description est imprimée.

Il s'agit là encore d'une violation de la règle de conception axée sur la responsabilité. Étant donné que la classe Room contient des informations sur une pièce, elle doit également produire une description pour une pièce. Nous pouvons améliorer cela en ajoutant à la classe Room la méthode suivante :

```
/**
 *
 *  Renvoie une longue description de cette salle, de la forme :
 *        Vous êtes dans la cuisine.
 *
 *        Sorties : nord ouest
 *
    @return Une description de la salle, y compris les sorties. */

    chaîne publique getLongDescription() {

        retour "Vous êtes                "
                                          + description + ".\n" + getExitString();
}
```

Dans la classe Game , on écrit alors

```
System.out.println(currentRoom.getLongDescription());
```

La "description longue" d'une pièce comprend désormais la chaîne de description et des informations sur les sorties, et peut à l'avenir inclure tout ce qu'il y a à dire sur une pièce.
Lorsque nous réaliserons ces futures extensions, nous devrons apporter des modifications uniquement à la classe Room .

Concept

L'un des principaux objectifs d'une bonne conception de classe est de localiser le changement : apporter des modifications à une classe devrait avoir des effets minimes sur les autres classes.

Exercice 8.11 Implémentez les modifications décrites dans cette section dans votre propre projet zuul .

Exercice 8.12 Dessinez un diagramme d'objets avec tous les objets de votre jeu, tels qu'ils sont juste après le début du jeu.

Exercice 8.13 Comment le diagramme d'objets change-t-il lorsque vous exécutez une commande go ?

8.8 Localisation du changement

Un autre aspect des principes de découplage et de responsabilité est celui de la localisation du changement.
Notre objectif est de créer une conception de classe qui facilite les changements ultérieurs en localisant les effets d'un changement.

Idéalement, une seule classe doit être changée pour effectuer une modification. Parfois, plusieurs classes doivent changer, mais nous visons alors à ce qu'il y ait le moins de classes possible. De plus, les changements nécessaires dans les autres classes doivent être évidents, faciles à détecter et faciles à effectuer.

Dans une large mesure, nous pouvons y parvenir en suivant de bonnes règles de conception telles que l'utilisation d'une conception axée sur la responsabilité et en visant un couplage lâche et une cohésion élevée. De plus, cependant, nous devons avoir à l'esprit les modifications et les extensions lorsque nous créons nos applications. Il est important d'anticiper qu'un aspect de notre programme pourrait changer, afin de faciliter tout changement.

8.9 Couplage implicite

Nous avons vu que l'utilisation des champs publics est une pratique susceptible de créer une forme inutilement étroite de couplage entre classes. Avec ce couplage étroit, il peut être nécessaire d'apporter des modifications à plus d'une classe pour ce qui aurait dû être une simple modification. Par conséquent, les champs publics doivent être évités. Cependant, il existe une forme de couplage encore pire : le couplage implicite.

Le couplage implicite est une situation où une classe dépend des informations internes d'une autre, mais cette dépendance n'est pas immédiatement évidente. Le couplage étroit dans le cas des champs publics n'était pas bon, mais au moins c'était évident. Si nous modifions les champs publics d'une classe et oublions l'autre, l'application ne se compilera plus et le compilateur signalera le problème. En cas de couplage implicite, l'omission d'un changement nécessaire peut passer inaperçue.

Nous pouvons voir le problème se poser si nous essayons d'ajouter d'autres mots de commande au jeu.

Supposons que nous voulions ajouter la commande look à l'ensemble des commandes légales. Le but de look est simplement d'imprimer à nouveau la description de la pièce et des sorties (on "regarde autour de la pièce"). Cela peut être utile si nous avons entré une séquence de commandes dans une pièce de sorte que la description a défilé hors de vue et que nous ne pouvons pas nous rappeler où se trouvent les sorties de la pièce actuelle.

Nous pouvons introduire un nouveau mot de commande simplement en l'ajoutant au tableau des mots connus dans le tableau validCommands de la classe CommandWords :

```
// un tableau constant contenant tous les mots de commande valides
private static final String validCommands[] = { "go", "quit", "help",
        "look"
} ;
```

Cela montre un exemple de bonne cohésion : au lieu de définir les mots de commande dans l'analyseur, ce qui aurait été une possibilité évidente, l'auteur a créé une classe séparée juste pour définir les mots de commande. Cela nous permet maintenant de trouver très facilement l'endroit où les mots de commande sont définis, et il est facile d'en ajouter un. L'auteur pensait évidemment à l'avenir, en supposant que d'autres commandes pourraient être ajoutées plus tard, et a créé une structure qui rend cela très facile.

Nous pouvons déjà tester cela. Lorsque nous apportons ce changement, puis exécutons le jeu et tapons la commande look, rien ne se passe. Cela contraste avec le comportement d'un mot de commande inconnu ; si nous tapons un mot inconnu, nous voyons la réponse

```
Je ne sais pas ce que tu veux dire...
```

Ainsi, le fait que nous ne voyons pas cette réponse indique que le mot a été reconnu, mais rien ne se passe car nous n'avons pas encore implémenté d'action pour cette commande.

Nous pouvons résoudre ce problème en ajoutant une méthode pour la commande look à la classe Game :

```
regard vide privé() {

        System.out.println(currentRoom.getLongDescription());
}
```

Vous devriez, bien sûr, également ajouter un commentaire pour cette méthode. Après cela, nous n'avons qu'à ajouter un cas pour la commande look dans la méthode processCommand , qui invoquera la méthode look lorsque la commande look sera reconnue :

```
if(commandWord.equals("help")) { printHelp();


} sinon if(commandWord.equals("go")) {
    goRoom(commande);

} else if(commandWord.equals("look")) { look();


} else if(commandWord.equals("quit")) { wantToQuit
    = quit(command);
}
```

Essayez ceci, et vous verrez que cela fonctionne.

Exercice 8.14 Ajoutez la commande look à votre version du jeu zuul.

Exercice 8.15 Ajoutez une autre commande à votre jeu. Pour commencer, vous pouvez choisir quelque chose de simple, comme une commande manger qui, une fois exécutée, affiche simplement "Vous avez mangé maintenant et vous n'avez plus faim".
Plus tard, nous pourrons améliorer cela afin que vous ayez vraiment faim au fil du temps et que vous ayez besoin de trouver de la nourriture.

Jusqu'à présent , le couplage entre les classes Game, Parser et CommandWords semble avoir été très bon - il était facile de créer cette extension et nous l'avons fait fonctionner rapidement.

Le problème mentionné précédemment - le couplage implicite - devient apparent lorsque nous émettons maintenant une commande d'aide. La sortie est

```
Vous êtes perdu. Vous êtes seul. Vous flânez à
l'université.
Vos mots de commande sont :
allez quitter aidez-moi
```

Maintenant, nous remarquons un petit problème. Le texte d'aide est incomplet : la nouvelle commande, look, n'est pas répertoriée.

Cela semble facile à résoudre : nous pouvons simplement modifier la chaîne de texte d'aide dans la méthode printHelp du jeu . Cela se fait rapidement et ne semble pas être un gros problème. Mais supposons que nous n'ayons pas remarqué cette erreur maintenant. Avez-vous pensé à ce problème avant de lire à ce sujet ici?

C'est un problème fondamental, car chaque fois qu'une commande est ajoutée, le texte d'aide doit être modifié, et il est très facile d'oublier de faire ce changement. Le programme se compile et s'exécute, et tout semble correct. Un programmeur de maintenance peut très bien croire que le travail est terminé et publier un programme qui contient maintenant un bogue.

Ceci est un exemple de couplage implicite. Lorsque les commandes changent, le texte d'aide doit être modifié (couplage), mais rien dans le source du programme ne signale clairement cette dépendance (donc implicite).

Une classe bien conçue évitera cette forme de couplage en suivant la règle de la conception axée sur la responsabilité. Étant donné que la classe CommandWords est responsable des mots de commande, elle doit également être responsable de l'impression des mots de commande. Ainsi, nous ajoutons la méthode suivante à la classe CommandWords :

```
/**
 *
     Affiche toutes les commandes valides sur System.out.
 */
public void showAll() {

        for(String command : validCommands) {
                System.out.print(commande +            " ");
        }
        System.out.println();
}
```

L'idée ici est que la méthode printHelp dans Game, au lieu d'imprimer un texte fixe avec les mots de commande, appelle une méthode qui demande à la classe CommandWords d'imprimer tous ses mots de commande. Cela garantit que les mots de commande corrects seront toujours imprimés, et l'ajout d'une nouvelle commande l'ajoutera également au texte d'aide sans autre modification.

Le seul problème restant est que l' objet Game n'a pas de référence à l' objet CommandWords . Vous pouvez voir dans le diagramme de classes (Figure 8.1) qu'il n'y a pas de flèche de Game à CommandWords. Cela indique que la classe Game ne connaît même pas l'existence de la classe CommandWords . Au lieu de cela, le jeu a juste un analyseur, et l'analyseur a des mots de commande.

Nous pourrions maintenant ajouter une méthode à l'analyseur qui transmet l' objet CommandWords à l' objet Game afin qu'ils puissent communiquer. Cela augmenterait cependant le degré de couplage dans notre application. Le jeu dépendrait alors de CommandWords, ce qu'il ne fait pas actuellement. De plus, nous verrions cet effet dans le diagramme de classes : Game aurait alors une flèche vers CommandWords.

Les flèches dans le diagramme sont, en fait, une bonne première indication du degré de couplage d'un programme - plus il y a de flèches, plus il y a de couplage. Comme approximation d'une bonne conception de classe, nous pouvons viser à créer des diagrammes avec peu de flèches.

Ainsi, le fait que Game n'ait pas de référence à CommandWords est une bonne chose ! Nous ne devrions pas changer cela. Du point de vue de Game , le fait que la classe CommandWords

existe est un détail d'implémentation de l'analyseur. L'analyseur renvoie des commandes, et s'il utilise un objet CommandWords pour y parvenir ou quelque chose d'autre dépend entièrement de l'implémentation de l'analyseur.

Une meilleure conception permet simplement au jeu de parler à l' analyseur, qui à son tour peut parler aux mots de commande. Nous pouvons implémenter cela en ajoutant le code suivant à la méthode printHelp dans Game :

```
System.out.println("Vos mots de commande sont :"); parser.showCommands();
```

Il ne manque donc que la méthode showCommands dans l' analyseur, qui délègue cette tâche à la classe CommandWords . Voici la méthode complète (dans la classe Parser):

```
/**
 *     Imprimez une liste de mots de commande valides.
 */
public void showCommands() {

        commandes.showAll();
}
```

Exercice 8.16 Implémentez la version améliorée de l'impression des mots de commande, comme décrit dans cette section.

Exercice 8.17 Si vous ajoutez maintenant une nouvelle commande, avez-vous encore besoin de changer la classe Game ? Pourquoi?

L'implémentation complète de tous les changements discutés dans ce chapitre jusqu'à présent est disponible dans vos exemples de code dans un projet nommé zuul-better. Si vous avez déjà fait les exercices, vous pouvez ignorer ce projet et continuer à utiliser le vôtre. Si vous n'avez pas fait les exercices mais que vous voulez faire les exercices suivants dans ce chapitre comme projet de programmation, vous pouvez utiliser le projet zuul-better comme point de départ.

8.10 Anticiper

La conception que nous avons maintenant est une amélioration importante par rapport à la version originale. Il est cependant possible de l'améliorer encore plus.

L'une des caractéristiques d'un bon concepteur de logiciels est sa capacité à anticiper. Qu'est-ce qui pourrait changer ? Que pouvons-nous supposer en toute sécurité qu'ils resteront inchangés pendant toute la durée de vie du programme ?

Une hypothèse que nous avons codée en dur dans la plupart de nos classes est que ce jeu fonctionnera comme un jeu basé sur du texte avec une entrée et une sortie terminales. Mais en sera-t-il toujours ainsi ?

Cela pourrait être une extension intéressante plus tard pour ajouter une interface utilisateur graphique avec des menus, des boutons et des images. Dans ce cas, nous ne voudrions pas imprimer les informations dans le texte

terminal plus. Nous pourrions toujours avoir des mots de commande, et nous pourrions toujours vouloir les afficher lorsqu'un joueur entre une commande d'aide. Mais nous pourrions alors les afficher dans un champ de texte dans une fenêtre, plutôt que d'utiliser System.out.println.

C'est une bonne conception d'essayer d'encapsuler toutes les informations sur l'interface utilisateur dans une seule classe ou un ensemble de classes clairement défini. Notre solution de la section 8.9, par exemple, la méthode showAll dans la classe CommandWords , ne suit pas cette règle de conception. Il serait bien de définir que CommandWords est responsable de la production (mais pas de l'impression !) de la liste des mots de commande, mais que la classe Game devrait décider comment elle est présentée à l'utilisateur.

Nous pouvons facilement y parvenir en modifiant la méthode showAll afin qu'elle renvoie une chaîne contenant tous les mots de commande au lieu de les imprimer directement. (Nous devrions probablement la renommer getCommandList lorsque nous apporterons cette modification.) Cette chaîne peut ensuite être imprimée dans la méthode printHelp de Game.

Notez que cela ne nous rapporte rien pour le moment, mais nous pourrions profiter de la conception améliorée à l'avenir.

Exercice 8.18 Mettez en œuvre le changement suggéré. Assurez-vous que votre programme fonctionne toujours comme avant.

Exercice 8.19 Découvrez ce qu'est le modèle modèle-vue-contrôleur . Vous pouvez faire une recherche sur le Web pour obtenir des informations, ou vous pouvez utiliser toute autre source que vous trouvez. Quel est le lien avec le sujet traité ici ? Que suggère-t-il ? Comment pourrait-il être appliqué à ce projet ? (Discutez uniquement de son application à ce projet, car une mise en œuvre réelle serait un exercice de défi avancé.)

8.11 Cohésion

Nous avons introduit l'idée de cohésion dans la section 8.3 : une unité de code doit toujours être responsable d'une et d'une seule tâche. Nous allons maintenant approfondir le principe de cohésion et analyser quelques exemples.

Le principe de cohésion peut s'appliquer aux classes et aux méthodes : les classes doivent afficher un haut degré de cohésion, et les méthodes aussi.

8.11.1 Cohésion des méthodes

Lorsque nous parlons de cohésion des méthodes, nous cherchons à exprimer l'idéal selon lequel toute méthode devrait être responsable d'une et d'une seule tâche bien définie.

Nous pouvons voir un exemple de méthode cohésive dans la classe Game . Cette classe a une méthode privée nommée printWelcome pour afficher le texte d'ouverture, et cette méthode est appelée lorsque le jeu démarre dans la méthode play (Code 8.8).

Code 8.8
Deux méthodes avec une bonne cohésion

```
/**
 *  Main play routine. Loops until end of play.
 */
public void play()
{
    printWelcome();

    // Enter the main command loop. Here we repeatedly read commands and
    // execute them until the game is over.

    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }
    System.out.println("Thank you for playing.  Good bye.");
}
```

```
/**
 * Print out the opening message for the player.
 */
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to the World of Zuul!");
    System.out.println("Zuul is a new, boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentRoom.getLongDescription());
}
```

Concept

Cohésion de la méthode

Une méthode cohésive est responsable d'une et d'une seule tâche bien définie.

D'un point de vue fonctionnel, nous aurions pu simplement saisir les instructions de la méthode printWelcome directement dans la méthode play et obtenir le même résultat sans définir de méthode supplémentaire ni appeler de méthode. Soit dit en passant, la même chose peut être dite pour la méthode processCommand qui est également invoquée dans la méthode play : ce code aurait également pu être écrit directement dans la méthode play .

Il est cependant beaucoup plus facile de comprendre ce que fait un segment de code et d'apporter des modifications si des méthodes courtes et cohérentes sont utilisées. Dans la structure de méthode choisie, toutes les méthodes sont raisonnablement courtes et faciles à comprendre, et leurs noms indiquent assez clairement leurs objectifs. Ces caractéristiques représentent une aide précieuse pour un programmeur de maintenance.

8.11.2 Cohésion des classes

La règle de cohésion des classes stipule que chaque classe doit représenter une seule entité bien définie dans le domaine du problème.

Concept

Cohésion de classe Une classe cohésive représente une entité bien définie.

Comme exemple de cohésion de classe, nous discutons maintenant d'une autre extension du projet zuul . Nous voulons maintenant ajouter des éléments au jeu. Chaque pièce peut contenir un objet, et chaque objet a une description et un poids. Le poids d'un article peut être utilisé ultérieurement pour déterminer s'il peut être ramassé ou non.

Une approche naïve consisterait à ajouter deux champs à la classe Room : itemDescription et itemWeight. Nous pourrions maintenant spécifier les détails de l'article pour chaque pièce, et nous pourrions imprimer les détails chaque fois que nous entrons dans une pièce.

Cette approche ne présente cependant pas un bon degré de cohésion : la classe Room décrit désormais à la fois une pièce et un élément. Cela suggère également qu'un élément est lié à une pièce particulière, ce que nous ne souhaiterions peut-être pas être le cas.

Une meilleure conception créerait une classe distincte pour les éléments, probablement appelée Item. Cette classe aurait des champs pour une description et un poids, et une pièce contiendrait simplement une référence à un objet d'élément.

Exercice 8.20 Étendez votre projet d'aventure ou le projet zuul-better afin qu'une pièce puisse contenir un seul élément. Les articles ont une description et un poids.
Lors de la création de salles et de la configuration de leurs sorties, des éléments pour ce jeu doivent également être créés. Lorsqu'un joueur entre dans une pièce, des informations sur un élément de cette pièce doivent être affichées.

Exercice 8.21 Comment produire l'information sur un élément présent dans une pièce ? Quelle classe doit produire la chaîne décrivant l'élément ?
Quelle classe doit l'imprimer ? Pourquoi? Expliquez par écrit. Si la réponse à cet exercice vous donne l'impression que vous devriez modifier votre implémentation, allez-y et apportez les modifications.

Les avantages réels de la séparation des pièces et des éléments dans la conception peuvent être constatés si nous modifions un peu les spécifications. Dans une autre variante de notre jeu, nous souhaitons autoriser non seulement un seul objet dans chaque pièce, mais un nombre illimité d'objets. Dans la conception utilisant une classe Item séparée , c'est facile. Nous pouvons créer plusieurs objets Item et les stocker dans une collection d'éléments dans la pièce.

Avec la première approche naïve, ce changement serait presque impossible à mettre en œuvre.

Exercice 8.22 Modifiez le projet afin qu'une pièce puisse contenir n'importe quel nombre d'éléments. Utilisez une collection pour ce faire. Assurez-vous que la pièce dispose d'une méthode addItem qui place un élément dans la pièce. Assurez-vous que tous les éléments sont affichés lorsqu'un joueur entre dans une pièce.

8.11.3 Cohésion pour la lisibilité

Il existe plusieurs façons dont une cohésion élevée profite à une conception. Les deux plus importants sont la lisibilité et la réutilisation.

L'exemple discuté dans la section 8.11.1, cohésion de la méthode printWelcome , est clairement un exemple dans lequel l'augmentation de la cohésion rend une classe plus lisible et donc plus facile à comprendre et à maintenir.

L'exemple de cohésion de classe de la section 8.11.2 a également un élément de lisibilité. S'il existe une classe d'articles distincte , un programmeur de maintenance reconnaîtra facilement par où commencer la lecture du code si une modification des caractéristiques d'un article est nécessaire. La cohésion des classes augmente également la lisibilité d'un programme.

8.11.4 Cohésion pour la réutilisation

Le deuxième grand avantage de la cohésion est un potentiel de réutilisation plus élevé.

L'exemple de cohésion de classe de la section 8.11.2 en montre un exemple : en créant une classe Item distincte , nous pouvons créer plusieurs éléments et ainsi utiliser le même code pour plusieurs éléments.

La réutilisation est également un aspect important de la cohésion de la méthode. Considérez une méthode dans la classe Room avec l'en-tête suivant :

```
salle publique leaveRoom (direction de la chaîne)
```

Cette méthode pourrait renvoyer la pièce dans la direction donnée (afin qu'elle puisse être utilisée comme nouvelle pièce courante) et également imprimer la description de la nouvelle pièce que nous venons d'entrer.

Cela semble être une conception possible, et cela peut en effet être fait pour fonctionner. Dans notre version, cependant, nous avons séparé cette tâche en deux méthodes :

```
public Room getExit(String direction) public String
getLongDescription()
```

Le premier se charge de rendre la salle suivante, tandis que le second produit la description de la salle.

L'avantage de cette conception est que les tâches séparées peuvent être réutilisées plus facilement. La méthode getLongDescription , par exemple, est désormais utilisée non seulement dans la méthode goRoom , mais également dans printWelcome et l'implémentation de la commande look . Cela n'est possible que parce qu'il affiche un haut degré de cohésion. La réutiliser ne serait pas possible dans la version avec la méthode leaveRoom .

Exercice 8.23 Implémentez une commande back . Cette commande n'a pas de second mot. Entrer la commande de retour emmène le joueur dans la pièce précédente dans laquelle il se trouvait.

Exercice 8.24 Testez votre nouvelle commande. Fonctionne-t-il comme prévu ? En outre, les cas de test où la commande est utilisée de manière incorrecte. Par exemple, que fait votre programme si un joueur tape un deuxième mot après la commande retour ? Se comporte-t-il raisonnablement ?

Exercice 8.25 Que fait votre programme si vous tapez « retour » deux fois ? Ce comportement est-il judicieux ?

Exercice 8.26 Exercice de défi Implémentez la commande back afin que son utilisation répétée vous ramène plusieurs pièces en arrière, jusqu'au début du jeu si elle est utilisée assez souvent. Utilisez une pile pour ce faire. (Vous devrez peut-être vous renseigner sur les piles. Consultez la documentation de la bibliothèque Java.)

8.12 Refactorisation

Concept

Le refactoring est l'activité de restructuration d'une conception existante pour maintenir une bonne conception de classe lorsque l'application est modifiée ou étendue.

Lors de la conception d'applications, nous devons essayer d'anticiper, d'anticiper les changements possibles dans le futur et de créer des classes et des méthodes hautement cohérentes et faiblement couplées qui facilitent les modifications. C'est un objectif noble, mais nous ne pouvons pas toujours anticiper toutes les adaptations futures, et il n'est pas possible de se préparer à toutes les extensions possibles auxquelles nous pouvons penser.

C'est pourquoi le refactoring est important.

La refactorisation est l'activité de restructuration des classes et des méthodes existantes pour les adapter aux fonctionnalités et aux exigences modifiées. Souvent, dans la durée de vie d'une application, des fonctionnalités sont progressivement ajoutées. Une conséquence courante est que, comme effet secondaire, les méthodes et les classes augmentent lentement en longueur.

Il est tentant pour un programmeur de maintenance d'ajouter du code supplémentaire aux classes ou méthodes existantes. Faire cela pendant un certain temps, cependant, diminue le degré de cohésion. Lorsque de plus en plus de code est ajouté à une méthode ou à une classe, il est probable qu'à un moment donné, il représentera plus d'une tâche ou entité clairement définie.

La refactorisation consiste à repenser et à reconcevoir les structures de classes et de méthodes. Le plus souvent, l'effet est que les classes sont divisées en deux ou que les méthodes sont divisées en deux méthodes ou plus. La refactorisation peut également inclure la réunion de plusieurs classes ou méthodes en une seule, mais cela est moins courant que le fractionnement.

8.12.1 Refactorisation et tests

Avant de donner un exemple de refactorisation, nous devons réfléchir au fait que, lorsque nous refactorisons un programme, nous proposons généralement d'apporter des modifications potentiellement importantes à quelque chose qui fonctionne déjà. Lorsque quelque chose est modifié, il est probable que des erreurs soient introduites. Par conséquent, il est important de procéder avec prudence ; et, avant le refactoring, nous devons établir qu'un ensemble de tests existe pour la version actuelle du programme. Si les tests n'existent pas, nous devons d'abord décider comment nous pouvons raisonnablement tester la fonctionnalité du

programmer et enregistrer ces tests (par exemple, en les écrivant) afin que nous puissions répéter les mêmes tests plus tard. Nous aborderons les tests plus formellement dans le chapitre suivant. Si vous êtes déjà familiarisé avec les tests automatisés, utilisez des tests automatisés. Sinon, des tests manuels (mais systématiques) suffisent pour l'instant.

Une fois qu'un ensemble de tests a été décidé, le refactoring peut commencer. Idéalement, le refactoring devrait alors se dérouler en deux étapes :

- La première étape consiste à refactoriser afin d'améliorer la structure interne du code, mais sans apporter de modifications aux fonctionnalités de l'application. En d'autres termes, le programme doit, lorsqu'il est exécuté, se comporter exactement comme il le faisait auparavant. Une fois cette étape terminée, les tests précédemment établis doivent être répétés pour s'assurer que nous n'avons pas introduit d'erreurs involontaires.

- La deuxième étape n'est effectuée qu'une fois que nous avons rétabli la fonctionnalité de base dans la version refactorisée. Nous sommes alors dans une position sûre pour améliorer le programme. Une fois cela fait, bien sûr, des tests devront être effectués sur la nouvelle version.

Faire plusieurs changements en même temps (refactoring et ajout de nouvelles fonctionnalités) rend plus difficile la localisation de la source des problèmes lorsqu'ils surviennent.

Exercice 8.27 Quel type de tests de fonctionnalité de base pourrions-nous souhaiter établir dans la version actuelle du jeu ?

8.12.2 Un exemple de refactorisation

A titre d'exemple, nous continuerons avec l'extension de l'ajout d'objets au jeu. Dans la section 8.11.2, nous avons commencé à ajouter des éléments, suggérant une structure dans laquelle les pièces peuvent contenir n'importe quel nombre d'éléments. Une extension logique de cet arrangement est qu'un joueur devrait être capable de ramasser des objets et de les transporter. Voici une spécification informelle de notre prochain objectif :

- Le joueur peut ramasser des objets dans la pièce actuelle.
- Le joueur peut transporter n'importe quel nombre d'objets, mais seulement jusqu'à un poids maximum.
- Certains articles ne peuvent pas être ramassés.
- Le joueur peut déposer des objets dans la salle actuelle.

Pour atteindre ces objectifs, nous pouvons faire ce qui suit :

- Si ce n'est déjà fait, nous ajoutons une classe Item au projet. Un élément a, comme indiqué ci-dessus, une description (une chaîne) et un poids (un entier).
- Nous devons également ajouter un nom de champ à la classe Item . Cela nous permettra de faire référence à l'article avec un nom plus court que celui de la description. Si, par exemple, il y a un livre dans la salle actuelle, les valeurs de champ de cet élément peuvent être :

  nom : livre

  description : un vieux livre poussiéreux relié en cuir gris poids : 1200

Si nous entrons dans une pièce, nous pouvons imprimer la description de l'objet pour indiquer au joueur ce qui s'y trouve. Mais pour les commandes, le nom sera plus facile à utiliser. Par exemple, le joueur peut alors taper prendre un livre pour ramasser le livre.

- Nous pouvons nous assurer que certains objets ne peuvent pas être ramassés, en les rendant simplement très lourds (plus qu'un joueur ne peut en porter). Ou devrions-nous avoir un autre champ booléen canBePickedUp ? Selon vous, quel est le meilleur design? Est-ce que ça importe? Essayez de répondre à cette question en pensant aux futurs changements qui pourraient être apportés au jeu.
- Nous ajoutons des commandes take and drop pour ramasser et déposer des éléments. Les deux commandes ont un nom de l'élément comme deuxième mot.
- Quelque part, nous devons ajouter un champ (contenant une forme de collection) pour stocker les objets actuellement portés par le joueur. Nous devons également ajouter un champ avec le poids maximum que le joueur peut porter, afin que nous puissions le vérifier chaque fois que nous essayons de ramasser quelque chose. Où devraient-ils aller? Encore une fois, pensez aux futures extensions pour vous aider à prendre la décision.

C'est cette dernière tâche que nous allons aborder plus en détail maintenant, afin d'illustrer le processus de refactoring.

La première question à se poser lorsque l'on réfléchit à la façon de permettre aux joueurs de transporter des objets est la suivante : où devrions-nous ajouter les champs pour les objets actuellement transportés et le poids maximum ? Un rapide coup d'œil sur les classes existantes montre que la classe Game est vraiment le seul endroit où elle peut être intégrée. Elle ne peut pas être stockée dans Room, Item ou Command, car il existe de nombreuses instances différentes de ces classes au fil du temps, qui sont pas tous toujours accessibles.
Cela n'a pas non plus de sens dans Parser ou CommandWords .

Renforcer la décision de placer ces changements dans la classe Jeu est le fait qu'elle stocke déjà la pièce actuelle (informations sur l'endroit où se trouve le joueur en ce moment), donc l'ajout des éléments actuels (informations sur ce que le joueur a) semble correspondre à cela plutôt bien.

Cette approche pourrait être mise en œuvre. Ce n'est cependant pas une solution bien conçue.
La classe Game est déjà assez grande, et il y a un bon argument qu'elle en contient trop telle quelle. Ajouter encore plus ne rend pas cela meilleur.

Nous devons nous demander à nouveau à quelle classe ou objet cette information doit appartenir. En réfléchissant bien au type d'informations que nous ajoutons ici (objets transportés, poids maximum), nous nous rendons compte qu'il s'agit d'informations sur un joueur ! La chose logique à faire (en suivant les directives de conception axées sur la responsabilité) est de créer une classe Player . Nous pouvons ensuite ajouter ces champs à la classe Player , et créer un objet Player au début du jeu, pour stocker les données.

Le champ existant currentRoom stocke également des informations sur le joueur : l'emplacement actuel du joueur. Par conséquent, nous devrions maintenant également déplacer ce champ dans la classe Player .

En l'analysant maintenant, il est évident que cette conception correspond mieux au principe de la conception axée sur la responsabilité. Qui devrait être responsable du stockage des informations sur le joueur ? La classe Joueur , bien sûr.

Dans la version originale, nous n'avions qu'une seule information pour le joueur : la salle actuelle. La question de savoir si nous aurions dû avoir une classe Player même à l'époque est à discuter.

Il y a des arguments dans les deux sens. Cela aurait été un beau design, alors, oui, peut-être que nous devrions. Mais avoir une classe avec un seul champ et aucune méthode qui fait quoi que ce soit d'important aurait pu être considéré comme exagéré.

Parfois, il y a des zones grises comme celle-ci, où l'une ou l'autre décision est défendable. Mais après avoir ajouté nos nouveaux champs, la situation est assez claire. Il existe maintenant un argument de poids en faveur d'une classe Player . Il stockerait les champs et aurait des méthodes telles que dropItem et pickUpItem (qui peuvent inclure la vérification du poids et pourraient renvoyer false si nous ne pouvons pas le transporter).

Ce que nous avons fait lorsque nous avons introduit la classe Player et déplacé le champ currentRoom de Game vers Player était une refactorisation. Nous avons restructuré la façon dont nous représentons nos données, pour parvenir à une meilleure conception dans le cadre de nouvelles exigences.

Des programmeurs moins bien formés que nous (ou simplement paresseux) auraient peut-être laissé le champ currentRoom là où il se trouvait, voyant que le programme fonctionnait tel qu'il était et qu'il ne semblait pas nécessaire de faire ce changement. Ils se retrouveraient avec une conception de classe désordonnée.

L'effet du changement peut être vu si nous réfléchissons un peu plus loin. Supposons que nous voulions maintenant étendre le jeu pour permettre plusieurs joueurs. Avec notre joli nouveau design, c'est soudainement très facile. Nous avons déjà une classe Player (le Game contient un objet Player), et il est facile de créer plusieurs objets Player et de stocker dans Game une collection de joueurs au lieu d'un seul joueur. Chaque objet joueur contiendrait sa propre pièce actuelle, ses objets et son poids maximum. Différents joueurs pourraient même avoir des poids maximum différents, ouvrant le concept encore plus large d'avoir des joueurs avec des capacités assez différentes - leur capacité de transport n'étant qu'une parmi tant d'autres.

Cependant, le programmeur paresseux qui a quitté currentRoom dans la classe Game a maintenant un sérieux problème. Étant donné que le jeu entier n'a qu'une seule pièce actuelle, les emplacements actuels de plusieurs joueurs ne peuvent pas être facilement stockés. Une mauvaise conception revient généralement pour créer plus de travail pour nous à la fin.

Faire un bon refactoring, c'est autant penser dans un certain état d'esprit que des compétences techniques. Pendant que nous apportons des modifications et des extensions aux applications, nous devons régulièrement nous demander si une conception de classe originale représente toujours la meilleure solution. Au fur et à mesure que la fonctionnalité change, les arguments pour ou contre certaines conceptions changent. Ce qui était une bonne conception pour une application simple peut ne plus l'être lorsque certaines extensions sont ajoutées.

Reconnaître ces changements et apporter les modifications de refactorisation au code source permet généralement d'économiser beaucoup de temps et d'efforts à la fin. Plus tôt nous nettoyons notre conception, plus nous économisons de travail.

Nous devons être prêts à factoriser les méthodes (transformer une séquence d'instructions du corps d'une méthode existante en une nouvelle méthode indépendante) et les classes (prendre des parties d'une classe et en créer une nouvelle). Envisager de refactoriser régulièrement garde notre conception de classe propre et économise du travail à la fin. Bien sûr, l'une des choses qui signifiera réellement que la refactorisation rend la vie plus difficile à long terme est si nous ne parvenons pas à tester correctement la version refactorisée par rapport à l'original. Chaque fois que nous nous lançons dans une tâche majeure de refactoring, il est essentiel de s'assurer que nous testons bien, avant et après le changement. Faire ces tests manuellement (en créant et en testant des objets de manière interactive) deviendra rapidement fastidieux. Nous étudierons comment nous pouvons améliorer nos tests, en les automatisant, dans le chapitre suivant.

Exercice 8.28 Refactorisez votre projet pour introduire une classe Player distincte . Un objet Player doit stocker au moins la salle actuelle du joueur, mais vous pouvez également stocker le nom du joueur ou d'autres informations.

Exercice 8.29 Implémentez une extension qui permet à un joueur de ramasser un seul objet. Cela inclut l'implémentation de deux nouvelles commandes : take et drop.

Exercice 8.30 Étendez votre implémentation pour permettre au joueur de transporter n'importe quel nombre d'objets.

Exercice 8.31 Ajoutez une restriction qui permet au joueur de transporter des objets uniquement jusqu'à un poids maximum spécifié. Le poids maximum qu'un joueur peut porter est un attribut du joueur.

Exercice 8.32 Implémentez une commande items qui imprime tous les articles actuellement transportés et leur poids total.

Exercice 8.33 Ajoutez un cookie magique à une pièce. Ajoutez une commande manger un cookie . Si un joueur trouve et mange le cookie magique, cela augmente le poids que le joueur peut porter. (Vous voudrez peut-être le modifier légèrement pour mieux l'adapter à votre propre scénario de jeu.)

8.13 Refactoring pour l'indépendance linguistique

Une caractéristique du jeu zuul que nous n'avons pas encore commentée est que l'interface utilisateur est étroitement liée aux commandes écrites en anglais. Cette hypothèse est intégrée à la fois dans la classe CommandWords , où la liste des commandes valides est stockée, et dans la classe Game , où la méthode processCommand compare explicitement chaque mot de commande à un ensemble de mots anglais. Si nous souhaitons modifier l'interface pour permettre aux utilisateurs d'utiliser une langue différente, nous devrons alors trouver tous les endroits du code source où les mots de commande sont utilisés et les modifier. Ceci est un autre exemple d'une forme de couplage implicite, dont nous avons discuté dans la section 8.9.

Si nous voulons avoir l'indépendance de la langue dans le programme, alors idéalement nous devrions avoir un seul endroit dans le code source où le texte réel des mots de commande est stocké et avoir partout ailleurs référence aux commandes d'une manière indépendante de la langue. Une fonctionnalité du langage de programmation qui rend cela possible est les types énumérés, ou énumérations. Nous allons explorer cette fonctionnalité de Java via les projets zuul-with-enums .

8.13.1 Types énumérés

Le code 8.9 montre une définition de type énuméré Java appelée CommandWord.

Code 8.9

Un type énuméré pour les mots de commande

```
/**
 * Representations for all the valid command words for the game.
 *
 * @author  Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */
public enum CommandWord
{
    // A value for each command word, plus one for unrecognized commands.
    GO, QUIT, HELP, UNKNOWN
}
```

Dans sa forme la plus simple, une définition de type énuméré consiste en un wrapper externe qui utilise le mot enum plutôt que class, et un corps qui est simplement une liste de noms de variables indiquant l'ensemble de valeurs qui appartiennent à ce type. Par convention, ces noms de variables sont entièrement en majuscules. Nous ne créons jamais d'objets d'un type énuméré. En effet, chaque nom dans la définition de type représente une instance unique du type qui a déjà été créée pour que nous l'utilisions. Nous appelons ces instances CommandWord.GO, CommandWord.QUIT, etc. Bien que la syntaxe pour les utiliser soit similaire, il est important d'éviter de considérer ces valeurs comme étant comme les constantes de classe numériques dont nous avons parlé dans la section 6.14. Malgré la simplicité de leur définition, les valeurs de type énumérées sont des objets propres et ne sont pas identiques aux entiers.

Comment pouvons-nous utiliser le type CommandWord pour faire un pas vers le découplage de la logique de jeu de zuul d'un langage naturel particulier ? L'une des premières améliorations que nous pouvons apporter concerne la série de tests suivante dans la méthode processCommand de Game :

```
if(command.isUnknown()) {
        System.out.println("Je ne sais pas ce que vous voulez dire..."); retourner
        faux ;
}
Chaîne commandWord = command.getCommandWord();
if(commandWord.equals("help")) { printHelp();


} else if(commandWord.equals("go"))
        { goRoom(commande);

} else if(commandWord.equals("quit")) { wantToQuit
        = quit(command);
}
```

Si commandWord est conçu pour être de type CommandWord plutôt que String, cela peut être réécrit comme suit :

```
if(commandWord == CommandWord.UNKNOWN)
        { System.out.println("Je ne sais pas ce que vous voulez dire...");

} else if(commandWord == CommandWord.HELP)
        { printHelp();
}
```

```
else if(commandWord == CommandWord.GO)
        { goRoom(command);

} else if(commandWord == CommandWord.QUIT)
        { wantToQuit = quit(command);
}
```

En fait, maintenant que nous avons changé le type en CommandWord, nous pourrions également utiliser une instruction switch au lieu de la série d'instructions if. Cela exprime un peu plus clairement l'intention de ce segment de code.2

```
switch (commandWord) { cas
        INCONNU :
                System.out.println("Je ne sais pas ce que vous voulez dire..."); casser;
                cas
        AIDE :
                printAide();
                casser;
        case GO :
                goRoom(commande) ;
                casser;
        cas QUITTER :
                wantToQuit = quitter(commande);
                casser;
}
```

Concept

Un interrupteur instruction sélectionne une séquence d'instructions à exécuter parmi plusieurs options différentes.

L'instruction switch prend la variable entre parenthèses après le mot-clé switch (commandWord dans notre cas) et la compare à chacune des valeurs répertoriées après les mots-clés case . Lorsqu'un cas correspond, le code qui le suit est exécuté. L' instruction break provoque l'abandon de l'instruction switch à ce stade et l'exécution se poursuit après l'instruction switch. Pour une description plus complète de l'instruction switch, voir l'annexe D.

Il ne nous reste plus qu'à faire en sorte que les commandes saisies par l'utilisateur soient mappées sur les valeurs CommandWord correspondantes . Ouvrez le projet zuul-with-enums-v1 pour voir comment nous avons procédé. Le changement le plus significatif se trouve dans la classe CommandWords . Au lieu d'utiliser un tableau de chaînes pour définir les commandes valides, nous utilisons maintenant une carte entre les chaînes et les objets CommandWord :

```
public MotsCommande() {

        validCommands = nouveau HashMap<>();
        validCommands.put("go", CommandWord.GO);
        validCommands.put("help", CommandWord.HELP);
        validCommands.put("quit", CommandWord.QUIT);
}
```

2 En fait, les littéraux de chaîne peuvent également être utilisés comme valeurs de cas dans les instructions switch, afin d'éviter une longue séquence de comparaisons if-else-if.

La commande tapée par un utilisateur peut maintenant être facilement convertie en sa valeur de type énumérée correspondante.

Exercice 8.34 Passez en revue le code source du projet zuul-with-enums-v1 pour voir comment il utilise le type CommandWord . Les classes Command, CommandWords, Game et Parser ont toutes été adaptées à partir de la version zuul-better pour s'adapter à ce changement. Vérifiez que le programme fonctionne toujours comme prévu.

Exercice 8.35 Ajoutez une commande look au jeu, selon les lignes décrites dans la section 8.9.

Exercice 8.36 "Traduire" le jeu pour utiliser des mots de commande différents pour les commandes GO et QUIT . Ceux-ci peuvent provenir d'une langue réelle ou simplement de mots inventés. Avez-vous seulement besoin de modifier la classe CommandWords pour que cette modification fonctionne ? Qu'est ce que cela signifie?

Exercice 8.37 Changez le mot associé à la commande HELP et vérifiez qu'il fonctionne correctement. Après avoir effectué vos modifications, que remarquez-vous concernant le message de bienvenue qui s'affiche au démarrage du jeu ?

Exercice 8.38 Dans un nouveau projet, définissez votre propre type énuméré appelé Direction avec les valeurs NORD, SUD, EST et OUEST.

8.13.2 Découplage supplémentaire de l'interface de commande

Le type CommandWord énuméré nous a permis de découpler de manière significative le langage de l'interface utilisateur de la logique du jeu, et il est presque tout à fait possible de traduire les commandes dans un autre langage simplement en éditant la classe CommandWords . (À un moment donné, nous devrions également traduire les descriptions de pièces et d'autres chaînes de sortie, probablement en les lisant à partir d'un fichier, mais nous laisserons cela pour plus tard.) Il y a un autre élément de découplage des mots de commande que nous aimerions effectuer.
Actuellement, chaque fois qu'une nouvelle commande est introduite dans le jeu, nous devons ajouter une nouvelle valeur au CommandWord, et une association entre cette valeur et le texte de l'utilisateur dans les classes CommandWords . Il serait utile que nous puissions rendre le type CommandWord autonome - en fait, déplacer l'association texte-valeur de CommandWords vers CommandWord.

Java permet aux définitions de types énumérées de contenir bien plus qu'une liste des valeurs du type. Nous n'explorerons pas cette fonctionnalité en détail, mais nous vous donnerons juste un avant-goût de ce qui est possible. Le code 8.10 montre un type CommandWord amélioré qui ressemble assez à une définition de classe ordinaire. Cela peut être trouvé dans le projet zuul-with-enums-v2 .

Code 8.10
Associer des chaînes de commande à des valeurs de type énumérées

```
/**
 * Representations for all the valid command words for the game
 * along with a string in a particular language.
 *
 * @author  Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */
public enum CommandWord
{
    // A value for each command word along with its
    // corresponding user interface string.
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?");

    // The command string.
    private String commandString;

    /**
     * Initialize with the corresponding command string.
     * @param commandString The command string.
     */
    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    /**
     * @return The command word as a string.
     */
    public String toString()
    {
        return commandString;
    }
}
```

Les principaux points à noter concernant cette nouvelle version de CommandWord sont les suivants :

- Chaque valeur de type est suivie d'une valeur de paramètre — dans ce cas, le texte de la commande associée à cette valeur.
- Contrairement à la version dans Code 8.9, un point-virgule est requis à la fin de la liste des valeurs de type.
- La définition de type inclut un constructeur. Cela n'a pas le mot public dans son en-tête. Les constructeurs de types énumérés ne sont jamais publics, car nous ne créons pas les instances. Le paramètre associé à chaque valeur de type est passé à ce constructeur.
- La définition de type inclut un champ, commandString. Le constructeur stocke le chaîne de commande dans ce champ.
- Une méthode toString a été utilisée pour renvoyer le texte associé à un type particulier valeur.

Avec le texte des commandes stockées dans le type CommandWord , la classe CommandWords dans zuul-with-enums-v2 utilise une manière différente de créer sa carte entre le texte et les valeurs énumérées :

```
validCommands = new HashMap<String, CommandWord>();
for(CommandWord command : CommandWord.values())
        { if(command != CommandWord.UNKNOWN)
                { validCommands.put(command.toString(), command);
        }
}
```

Chaque type énuméré définit une méthode de valeurs qui renvoie un tableau rempli avec les objets de valeur du type. Le code ci-dessus parcourt le tableau et appelle la méthode toString pour obtenir la commande String associée à chaque valeur.

Exercice 8.39 Ajoutez votre propre commande look à zuul-with-enums-v2. Avez-vous seulement besoin de changer le type CommandWord ?

Exercice 8.40 Modifier le mot associé à la commande help dans CommandWord. Ce changement est-il automatiquement reflété dans le texte de bienvenue lorsque vous démarrez le jeu ? Jetez un œil à la méthode printWelcome dans la classe Game pour voir comment cela a été réalisé.

8.14 Directives de conception

Un conseil souvent donné aux débutants concernant l'écriture de bons programmes orientés objet est : « Ne mettez pas trop de choses dans une seule méthode » ou « Ne mettez pas tout dans une seule classe ». Les deux suggestions ont du mérite, mais conduisent souvent aux contre-questions : "Combien de temps une méthode doit-elle durer ?" ou "Combien de temps doit durer un cours ?"

Après la discussion dans ce chapitre, ces questions peuvent maintenant trouver une réponse en termes de cohésion et de couplage. Une méthode est trop longue si elle effectue plus d'une tâche logique. Une classe est trop complexe si elle représente plus d'une entité logique.

Vous remarquerez que ces réponses ne donnent pas de règles claires qui spécifient exactement ce qu'il faut faire. Des termes tels qu'une tâche logique sont encore ouverts à l'interprétation, et différents programmeurs décideront différemment dans de nombreuses situations.

Ce sont des lignes directrices et non des règles immuables. Garder cela à l'esprit, cependant, améliorera considérablement la conception de votre classe et vous permettra de maîtriser des problèmes plus complexes et d'écrire des programmes meilleurs et plus intéressants.

Il est important de comprendre les exercices suivants comme des suggestions et non comme des spécifications fixes. Ce jeu a de nombreuses possibilités d'extension, et vous êtes encouragé à inventer vos propres extensions. Vous n'avez pas besoin de faire tous les exercices ici pour créer un jeu intéressant ; vous voudrez peut-être en faire plus,

ou vous voudrez peut-être en faire d'autres. Voici quelques suggestions pour vous aider à démarrer.

Exercice 8.41 Ajoutez une certaine forme de limite de temps à votre jeu. Si une certaine tâche n'est pas terminée dans un délai spécifié, le joueur perd. Une limite de temps peut facilement être mise en place en comptant le nombre de coups ou le nombre de commandes saisies. Vous n'avez pas besoin d'utiliser le temps réel.

Exercice 8.42 Mettez en place une trappe quelque part (ou une autre forme de porte que vous ne pouvez franchir que dans un sens).

Exercice 8.43 Ajoutez un projecteur au jeu. Un projecteur est un appareil qui peut être chargé et déclenché. Lorsque vous chargez le projecteur, il mémorise la pièce actuelle. Lorsque vous déclenchez le projecteur, il vous ramène immédiatement dans la pièce dans laquelle il a été chargé. Le projecteur peut être soit un équipement standard, soit un objet que le joueur peut trouver. Bien sûr, vous avez besoin de commandes pour charger et déclencher le projecteur.

Exercice 8.44 Ajoutez des portes verrouillées à votre jeu. Le joueur doit trouver (ou obtenir autrement) une clé pour ouvrir une porte.

Exercice 8.45 Ajoutez une salle de téléportation. Chaque fois que le joueur entre dans cette pièce, il est transporté au hasard dans l'une des autres pièces. Remarque : Trouver un bon design pour cette tâche n'est pas anodin. Il pourrait être intéressant de discuter d'alternatives de conception pour cela avec d'autres étudiants. (Nous discutons des alternatives de conception pour cette tâche à la fin du chapitre 11. Le lecteur aventureux ou avancé voudra peut-être sauter devant et jeter un coup d'œil.)

8.15 Résumé

Dans ce chapitre, nous avons abordé ce que l'on appelle souvent les aspects non fonctionnels d'une application. Ici, le problème n'est pas tant de faire en sorte qu'un programme exécute une certaine tâche, mais de le faire avec des classes bien conçues.

Une bonne conception de classe peut faire une énorme différence lorsqu'une application doit être corrigée, modifiée ou étendue. Cela nous permet également de réutiliser des parties de l'application dans d'autres contextes (par exemple, pour d'autres projets) et crée ainsi des avantages plus tard.

Il existe deux concepts clés sous lesquels la conception de classe peut être évaluée : le couplage et la cohésion. Le couplage fait référence à l'interdépendance des classes, la cohésion à la modularisation en unités appropriées. Une bonne conception présente un couplage lâche et une cohésion élevée.

Une façon d'obtenir une bonne structure est de suivre un processus de conception axée sur la responsabilité. Chaque fois que nous ajoutons une fonction à l'application, nous essayons d'identifier quelle classe doit être responsable de quelle partie de la tâche.

Lors de l'extension d'un programme, nous utilisons une refactorisation régulière pour adapter la conception aux exigences changeantes et pour garantir que les classes et les méthodes restent cohérentes et faiblement couplées.

Termes introduits dans ce chapitre :

duplication de code, couplage, cohésion, encapsulation, responsability-driven design, couplage implicite, refactoring

Résumé du concept

- couplage Le terme couplage décrit l'interconnexion des classes. Nous nous efforçons de couplage lâche dans un système, c'est-à-dire un système où chaque classe est largement indépendante et communique avec les autres classes via une petite interface bien définie.
- Cohésion Le terme cohésion décrit dans quelle mesure une unité de code correspond à une tâche ou une entité logique. Dans un système hautement cohérent , chaque unité de code (méthode, classe ou module) est responsable d'une tâche ou d'une entité bien définie. Une bonne conception de classe présente un degré élevé de cohésion.
- duplication de code Duplication de code (avoir le même segment de code dans une application plus d'une fois) est un signe de mauvaise conception. Il devrait être évité.
- encapsulation Une bonne encapsulation dans les classes réduit le couplage et conduit ainsi à une meilleure conception.
- conception axée sur la responsabilité La conception axée sur la responsabilité est le processus de conception classes en attribuant des responsabilités bien définies à chaque classe. Ce processus peut être utilisé pour déterminer quelle classe doit implémenter quelle partie d'une fonction d'application.
- Localiser le changement L'un des principaux objectifs d'une bonne conception de classe est de localiser le changement : apporter des modifications à une classe devrait avoir des effets minimaux sur les autres classes.
- Cohésion de la méthode Une méthode cohésive est responsable d'une, et d'une seule, bien définie tâche.
- cohésion de classe Une classe cohésive représente une entité bien définie.
- refactoring Le refactoring est l'activité de restructuration d'une conception existante pour maintenir une bonne conception de classe lorsque l'application est modifiée ou étendue.
- Instruction switch Une instruction switch sélectionne une séquence d'instructions à exécuter parmi plusieurs options différentes.

Exercice 8.46 Exercice de défi Dans la méthode processCommand de Game, il existe une instruction switch (ou une séquence d'instructions if) pour envoyer des commandes lorsqu'un mot de commande est reconnu. Ce n'est pas une très bonne conception, car chaque fois que nous ajoutons une commande, nous devons ajouter un cas ici. Pouvez-vous améliorer cette conception? Concevez les classes de sorte que la gestion des commandes soit plus modulaire et que de nouvelles commandes puissent être ajoutées plus facilement. Mettre en œuvre. Essaye-le.

Exercice 8.47 Ajoutez des personnages au jeu. Les personnages sont similaires aux objets, mais ils peuvent parler. Ils parlent un texte lorsque vous les rencontrez pour la première fois, et ils peuvent vous aider si vous leur donnez le bon article.

Exercice 8.48 Ajouter des personnages en mouvement. Ce sont comme les autres personnages, mais chaque fois que le joueur tape une commande, ces personnages peuvent se déplacer dans une pièce adjacente.

Exercice 8.49 Ajoutez une classe appelée GameMain à votre projet. Définissez simplement une méthode principale à l'intérieur de celle-ci et demandez-leur de créer un objet Game et d'appeler sa méthode play .