7

Conception des classes

Concepts clés de ce chapitre

- Conception dirigée par les responsabilités
- Cohésion
- Couplage
- Réingénierie

Notions Java

- `static` (pour les méthodes), `Math`, types énumérés

Termes introduits dans ce chapitre

- Duplication du code, couplage, cohésion, encapsulation, conception dirigée par les responsabilités, couplage implicite, réingénierie, méthode de classe

Dans ce chapitre, nous étudierons quelques facteurs qui influencent la conception des classes. Quelle est la différence entre une classe bien ou mal conçue ? Écrire des classes bien conçues demande plus d'efforts à court terme, mais ces efforts supplémentaires sont souvent justifiés à long terme. Nous pouvons respecter plusieurs principes qui nous aideront à bien concevoir nos classes. En particulier, nous introduirons le point de vue suivant : la conception des classes devrait être dirigée par les responsabilités et ces classes devraient encapsuler leurs données.

Ce chapitre, comme les précédents, est organisé autour d'un projet. Il peut être étudié simplement en le lisant et en suivant notre argumentaire, ou plus en profondeur, en effectuant les exercices du projet.

Le travail lié au projet est divisé en trois parties. Dans la première, nous mettons en évidence les modifications indispensables au code source et nous développons des solutions complètes en exercice. La solution de cette partie est disponible sous forme d'un projet accompagnant ce livre. La deuxième partie suggère d'autres changements et modifications, et nous comparons différentes solutions à un haut niveau (au niveau

des classes). Cependant, nous laissons au lecteur le soin d'accomplir le travail de bas niveau et de terminer l'implantation.

La troisième partie suggère de nouvelles améliorations sous forme d'exercices. Nous ne proposons pas de solutions, les exercices mettant en œuvre les techniques présentées au cours de ce chapitre.

Implanter toutes les parties représente un bon projet de programmation sur plusieurs semaines. Il peut aussi très bien être réalisé en groupe.

Introduction

Il est possible d'implanter une application comprenant des classes mal conçues, mais qui produit les résultats attendus. Le simple fait d'exécuter une application terminée n'indique pas en général si elle est bien structurée en interne.

Les problèmes apparaissent typiquement quand un programmeur chargé de la maintenance souhaite modifier une application existante. Par exemple, si un programmeur cherche à corriger une erreur ou souhaite ajouter de nouvelles fonctionnalités à un programme, ce qui devrait être facile et évident lorsque les classes sont bien conçues peut se révéler difficile et demander un travail ardu dans le cas contraire.

Dans le cadre d'applications importantes, cet effet se ressent dès l'implantation initiale. Si l'implantation débute avec une mauvaise structure, sa conclusion peut être très complexe, et le programme global peut ne jamais se terminer, contenir des erreurs ou demander bien plus de temps que nécessaire pour arriver à une version finale. En réalité, les entreprises maintiennent, étendent et vendent une application pendant plusieurs années. Il n'est pas rare que l'implantation d'un logiciel que nous pouvons acheter aujourd'hui ait débuté plus de dix ans auparavant. Dans ce cas, une entreprise ne peut pas se permettre d'avoir un code mal structuré.

Comme les effets liés à une mauvaise conception des classes apparaissent quand nous cherchons à étendre ou à adapter une application, nous allons faire de même. Dans ce chapitre, nous utiliserons un exemple appelé *Le monde de Zuul*, qui réalise une implantation rudimentaire d'un jeu d'aventures en mode texte. Dans son état initial, le jeu n'est pas très ambitieux : d'un certain point de vue, il n'est pas achevé. À la fin de ce chapitre, vous serez cependant en mesure d'exercer votre imagination, de concevoir et d'implanter votre propre jeu, et de le rendre réellement intéressant et amusant.

Tout en travaillant à l'extension de l'application originale, nous en profiterons pour discuter certains aspects de sa structure de classes. Nous nous rendrons compte que l'implantation initiale contient plusieurs mauvais exemples de conception, nous en mesurerons les conséquences sur notre travail et nous chercherons à corriger ces problèmes.

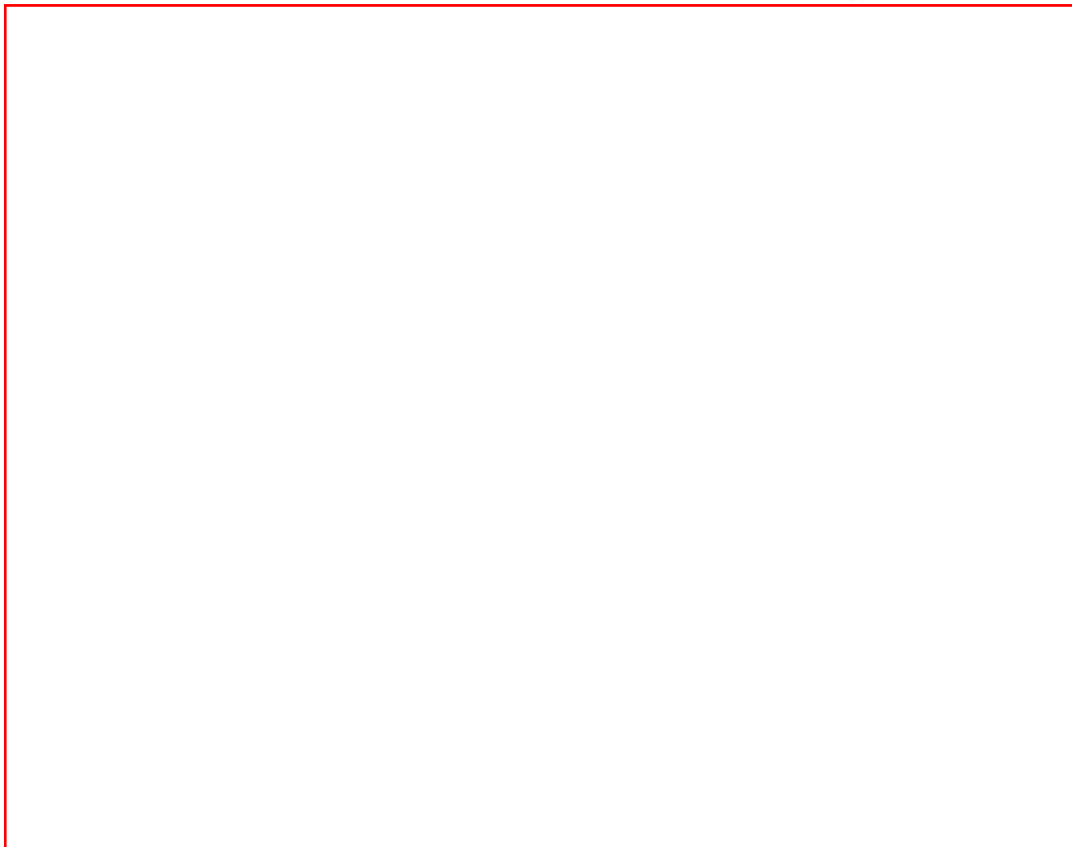
Dans les exemples de projets de ce livre, vous trouverez deux versions du projet *zuul* : *zuul-bad* et *zuul-better*. Chacune implante exactement les mêmes fonctionnalités, mais les structures de certaines classes sont différentes, reflétant une mauvaise conception

Le monde de Zuul

Notre jeu, *Le monde de Zuul*, est basé sur le jeu d'aventures original qui a été développé par Will Crowther et étendu par Don Woods au début des années soixante-dix. Le jeu original est parfois connu sous le nom de *Colossal Cave Adventure*. C'était un jeu très imaginatif et sophistiqué pour son époque, qui demandait au joueur de trouver son chemin dans un système complexe de grottes, de rechercher des trésors cachés, d'utiliser des mots secrets et d'autres objets mystérieux, tout cela pour obtenir un maximum de points. Vous en saurez plus sur ce jeu aux adresses suivantes : **http://jerz.setonhill.edu/if/canon/Adventure.htm** et **http://www.rickadams.org/adventure/** ou en recherchant Colossal Cave Adventure sur le Web.

dans un projet, et une meilleure conception dans l'autre. Le fait de pouvoir implanter les mêmes fonctionnalités d'une manière satisfaisante ou pas prouve qu'une mauvaise conception n'est pas nécessairement une conséquence de la complexité du problème à résoudre. Une mauvaise conception est plutôt liée aux décisions prises lors de la résolution d'un problème particulier. Dire qu'il n'était pas possible de résoudre le problème autrement n'est pas une excuse pour une mauvaise conception.

Nous allons ainsi utiliser le projet mal conçu afin d'étudier en quoi sa conception n'est pas satisfaisante. Nous l'améliorerons par la suite. La meilleure version est une implantation des modifications étudiées.



Exemple : le jeu du monde de Zuul

L'exercice 7.1 vous a permis de vous rendre compte que le jeu de Zuul n'était pas très aventurier. En réalité, ce jeu est plutôt ennuyeux dans sa version actuelle. Il fournit cependant une bonne base de conception et d'implantation de notre jeu, qui devrait être plus intéressant.

Nous commençons par analyser les classes déjà présentes dans notre première version et nous essayons de comprendre ce qu'elles font. Le diagramme de classe est présenté à la figure 7.1.

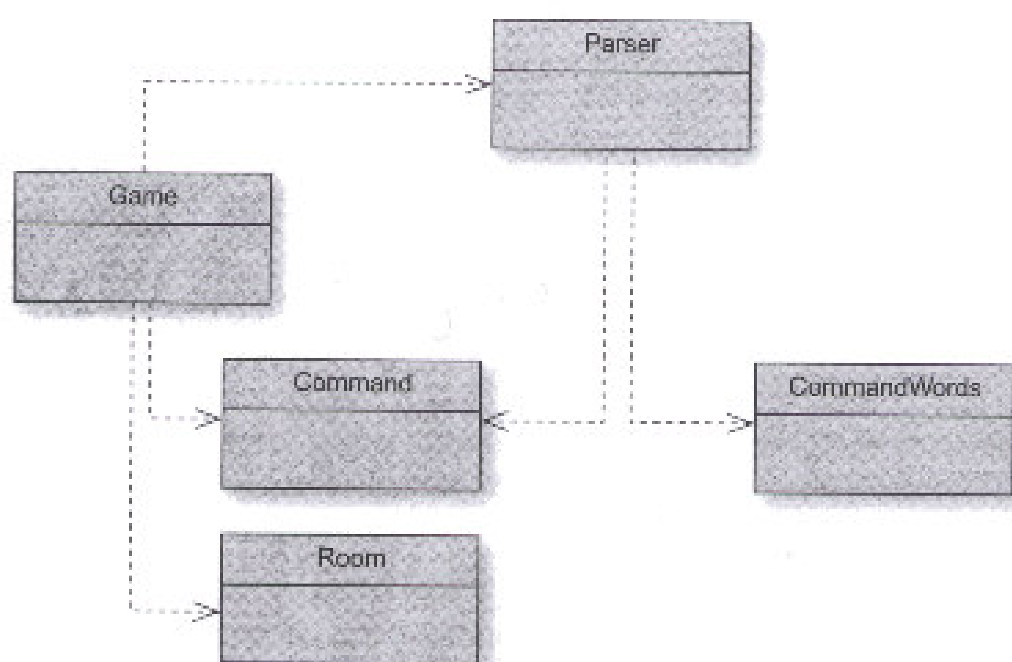


Figure 7.1 • Le diagramme de classe de Zuul.

Le projet inclut cinq classes : `Parser`, `CommandWords`, `Command`, `Room` et `Game`. Un parcours rapide du code source montre que ces classes sont heureusement plutôt bien documentées, et nous obtenons un premier point de vue sur leur travail en lisant simplement les commentaires qui débutent chaque classe (cela illustre aussi le fait qu'une mauvaise conception n'implique pas forcément une mauvaise présentation ou documentation des classes). Notre compréhension du jeu sera facilitée par la lecture du code source pour découvrir les méthodes associées à chaque classe et leur rôle. Voici un résumé du rôle de chaque classe :

- `CommandWords`. Cette classe définit toutes les commandes valides du jeu. Un tableau d'objets `String` représentant tous les mots associés à ces commandes est utilisé pour cette tâche.
- `Parser`. L'analyseur syntaxique lit les lignes entrées au clavier et tente de les interpréter comme des commandes. Il crée des objets de la classe `Command` qui représentent les commandes saisies.

- `Command`. Un objet de type `Command` représente une commande saisie par l'utilisateur. Cette classe contient des méthodes qui facilitent la vérification de la validité d'une commande, et l'obtention du premier et du deuxième mot, saisis sous forme de chaînes de caractères.
- `Room`. Un objet de type `Room` représente une pièce dans le jeu. Les pièces peuvent avoir des sorties qui conduisent à d'autres pièces.
- `Game`. La classe `Game` est la classe principale du programme. Elle initialise le jeu et exécute ensuite une boucle de lecture et d'exécution des commandes. Elle contient également le code qui implante les commandes de l'utilisateur.