Le couplage entre les classes `Game`, `Parser` et `CommandWords` semblait jusqu'à présent très bon — cette extension a été facile à réaliser, et l'ensemble a fonctionné rapidement.

Le problème que nous avons mentionné auparavant — le couplage implicite — devient apparent quand nous traitons la commande `aide`. L'affichage est :

```
Vous êtes perdu. Vous êtes seul. Vous flânez
à travers l'université.

Les commandes sont :
  aller quitter aide
```

Nous découvrons maintenant une petite difficulté. Le texte d'aide est incomplet : la nouvelle commande, `regarder`, n'est pas listée.

Cela semble facile à corriger : nous pouvons simplement éditer la chaîne de caractères, correspondant à l'aide, de la méthode `printHelp` de la classe `Game`. La modification est rapidement faite, et ne semble pas être un gros problème. Mais supposons que nous n'ayons pas remarqué cette erreur maintenant. Aviez-vous pensé à cela avant de lire ce paragraphe ?

C'est une erreur fondamentale, car chaque fois qu'une commande est ajoutée, le texte d'aide doit être changé et il est très facile d'oublier de le faire. Le programme se compile, fonctionne et tout semble parfait. Un programmeur chargé de la maintenance pourrait facilement croire que le travail est fini, et mettre à jour un programme contenant un bogue.

C'est un exemple de couplage implicite : quand les commandes changent, le texte d'aide doit lui aussi être modifié (couplage), mais rien dans le code source du programme n'indique clairement cette dépendance (couplage implicite).

Une classe bien conçue évitera cette forme de couplage en suivant la règle de conception dirigée par les responsabilités : comme la classe `CommandWords` est responsable des commandes, elle doit aussi être responsable de leur affichage. Nous ajoutons ainsi 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(command + "  ");
        }
        System.out.println();
    }
```

Plutôt que d'afficher un texte prédéterminé associé aux commandes, la méthode `printHelp` de la classe `Game` appelle une méthode qui demande à la classe `CommandWords` d'afficher toutes ses commandes. De cette façon, nous sommes sûrs que la liste des commandes valides sera toujours affichée et l'ajout d'une nouvelle commande entraîne son ajout à la commande d'aide sans aucune modification supplémentaire.

Le seul problème restant est que l'objet `Game` ne possède pas de référence vers l'objet `CommandWords`. Il n'existe pas de flèche entre les classes `Game` et `CommandWords` dans le diagramme des classes (figure 7.1). Cela indique que la classe `Game` ne connaît même pas l'existence de la classe `CommandWords`. Au lieu de cela, le jeu inclut un analyseur qui possède des mots de commandes.

Nous pourrions, sur ce point, ajouter à l'analyseur une méthode qui procure à l'objet `Game` un accès à l'objet `CommandWords`, afin qu'ils puissent communiquer. Cependant, cela augmenterait le degré de couplage dans notre application : `Game` dépendrait alors de la classe `CommandWords`, ce qui n'est pas le cas actuellement. Nous verrions cet effet sur le diagramme de classe : il y aurait alors une flèche issue de `Game` en direction de `CommandWords`.

Les flèches du diagramme sont en réalité une bonne indication sur le degré de couplage d'un programme. Plus les flèches sont nombreuses, plus le couplage est fort. Pour s'approcher d'une bonne conception de classes, nous pouvons chercher à créer des diagrammes contenant peu de flèches.

Ainsi, le fait que `Game` ne possède pas de référence vers `CommandWords` est une bonne chose ! Nous ne devrions pas changer cela. Du point de vue de la classe `Game`, l'existence de la classe `CommandWords` est un détail d'implantation de l'analyseur. L'analyseur renvoie des commandes, et l'utilisation d'un objet `CommandWords` pour y arriver ou faire autre chose est complètement enveloppée par l'implantation de l'analyseur.

Il en découle qu'une meilleure conception laisse l'objet `Game` communiquer avec l'objet `Parser` qui, en retour, communique avec l'objet `CommandWords`. Nous pouvons implanter cela en ajoutant le code suivant à la méthode `printHelp` de la classe `Game` :

```
        System.out.println("Vos commandes sont :");
        parser.showCommands();
```

Tout ce qu'il manque alors est la méthode `showCommands` de la classe `Parser`, qui délègue cette tâche à la classe `CommandWords`. Voici la méthode complète (dans la classe `Parser`) :

```
    /**
     * Affiche une liste des commandes valides.
     */
    public void showCommands()
    {
        commands.showAll();
    }
```