Localisation des modifications

Un autre aspect du découplage et de la conception dirigée par les responsabilités est la *localisation des modifications*. Nous souhaitons créer une conception des classes qui facilite les modifications en localisant les effets d'un changement.

Idéalement, une seule classe devrait être changée pour apporter une modification. Il est parfois nécessaire de modifier plusieurs classes, mais alors nous cherchons à minimiser le nombre de classes concernées. De plus, les modifications nécessaires aux autres classes devraient être évidentes, faciles à détecter et à apporter.

Concept

Un des principaux objectifs d'une bonne conception des classes est la **localisation des modifications** : modifier une classe devrait avoir un minimum d'effets sur les autres classes.

Nous pouvons, dans une large mesure, atteindre ce but en suivant de bonnes règles de conception telles que la conception dirigée par les responsabilités, réduire le couplage et augmenter la cohésion. Cependant, nous devons aussi garder à l'esprit les modifications et extensions possibles quand nous créons nos applications. Il est important d'anticiper les parties de notre programme qui pourraient évoluer afin de faciliter ces évolutions.

Couplage implicite

Nous avons déjà montré que l'utilisation de champs publics est une pratique qui peut créer inutilement une forme étroite de couplage entre classes. Ce couplage fort peut rendre nécessaire la modification de plusieurs classes pour ce qui aurait dû être une modification simple. C'est pourquoi il faut éviter les champs publics. Cependant, il existe une forme encore plus grave de couplage : le *couplage implicite*.

Le couplage implicite est une situation dans laquelle une classe dépend de l'information interne d'une autre, sans que cette dépendance soit visible de manière évidente. Le couplage élevé induit par les champs publics n'est pas souhaitable, mais il est au moins évident : si nous changeons les champs publics d'une classe et oublions de modifier une autre classe, l'application ne pourra plus être compilée et le compilateur mettra le problème en évidence. Dans le cas d'un couplage implicite, omettre un changement indispensable peut passer inaperçu.

Nous pouvons rencontrer ce problème si nous essayons d'ajouter de nouvelles commandes à notre jeu.

Supposons que nous voulions ajouter la commande regarder à l'ensemble des commandes possibles. L'objectif de cette commande est tout simplement de réafficher la description de la pièce et de ses sorties (nous « jetons un coup d'œil dans la pièce »). Cela pourrait être utile, par exemple, si nous avons donné une série de commandes dans une pièce, que la description ne soit plus visible à l'écran et que nous ne nous souvenions plus de ses sorties.

Nous pouvons introduire une nouvelle commande en l'ajoutant simplement au tableau des commandes connues `validCommands` de la classe `CommandWords` :

```
// un tableau constant qui conserve les commandes valides
private static final String validCommands[] = {
   "aller", "quitter", "aide", "regarder"
} ;
```

Voilà un exemple de bonne cohésion : au lieu de définir les commandes dans l'analyseur syntaxique, ce qui est une possibilité évidente, l'auteur a créé une classe distincte

uniquement pour définir les commandes. Ce choix facilite maintenant la recherche de l'endroit où les commandes sont définies et l'insertion d'une nouvelle commande. De façon évidente, l'auteur a réfléchi à cet aspect ; il a considéré que de nouvelles commandes pourraient être ajoutées plus tard et il a créé une structure qui facilite cette tâche.

Nous pouvons déjà tester cette propriété. Cependant, après avoir apporté la modification, quand nous exécutons le jeu et saisissons la commande `regarder`, rien ne se produit. Ce comportement est très différent de celui que nous avons observé lors de la saisie d'une commande inconnue : si nous saisissons une commande inconnue, nous lisons la réponse :

```
Je ne comprends pas cette commande…
```

Ce message n'est pas visible, ce qui signifie que le mot a bien été reconnu ; cependant, rien ne se passe car nous n'avons pas encore implanté d'action pour cette commande.

Nous pouvons corriger ce problème en ajoutant une méthode à la classe `Game` pour la commande `regarder` :

```
private void look()
{
  System.out.println(currentRoom.getLongDescription());
}
```

Ensuite, il ne nous reste plus qu'à ajouter un cas pour la commande `regarder` dans la méthode `processCommand`, qui appellera la méthode `look` quand la commande `regarder` sera reconnue :

```
        if(commandWord.equals("aide")) {
            printHelp();
        }
        else if(commandWord.equals("aller")) {
            goRoom(command);
        }
        else if(commandWord.equals("regarder")) {
            look();
        }
        else if(commandWord.equals("quitter")) {
            wantToQuit = quit(command);
        }
```

Essayez cette modification et vous verrez que cela fonctionne.