Réingénierie et indépendance linguistique

Nous n'avons pas encore abordé le fait que l'interface utilisateur est étroitement liée à des commandes rédigées en français. Cette hypothèse est liée à la classe `CommandWords`, qui stocke la liste des commandes, et à la classe `Game`, dans laquelle la méthode `processCommand` compare explicitement chaque mot de commande à un ensemble de mots français. Si nous souhaitons modifier l'interface pour permettre à des utilisateurs d'utiliser une autre langue, il faut retrouver tous les emplacements dans le code source où les mots de commande sont utilisés, afin de les modifier. Il s'agit d'un nouvel exemple de couplage implicite, que nous avons déjà évoqué.

Pour parvenir à ce que la langue soit indépendante du programme, il faudrait, dans l'idéal, un emplacement dans le source qui stockerait le texte des commandes. Tous les autres endroits devront alors faire référence aux commandes indépendamment de la langue. Un aspect du langage de programmation permet cela : les *types énumérés*. Nous les étudierons grâce aux projets *zuul-with-enums*.

Types énumérés

Le code 7.9 présente une définition de type énuméré Java appelé `CommandWord`.

Code 7.9 • Un type énuméré pour les mots des commandes.

```
/**
 * Représentation de tous les mots de commande valables
 * pour le jeu.
 *
 * @author Michael Kolling et David J. Barnes
 * @version 2008.03.30
 */
public enum CommandWord
{
    // Une valeur pour chaque mot de commande, plus un
    // pour les commandes non reconnues
    ALLER, QUITTER, AIDE, INCONNU;
}
```

Sous sa forme la plus simple, une définition de type énuméré est composée d'une enveloppe extérieure employant le mot `enum` et non `class`, ainsi que d'un corps, qui est simplement une liste de noms de variables présentant l'ensemble de valeurs appartenant à ce type. Par convention, ces noms s'écrivent en majuscules. Nous ne créons jamais d'objets de type énuméré. En effet, chaque nom dans la définition de type représente une instance unique du type déjà créé pour qu'on l'utilise. Nous désignons ces instances sous les termes `CommandWord.GO`, `CommandWord.QUIT`, etc. Bien que leur syntaxe soit similaire, il faut absolument éviter de considérer ces valeurs comme des constantes de classes numériques, traitées au paragraphe « Variables de classes et constantes » du chapitre 5. Malgré la simplicité de leur définition, les valeurs des types énumérés sont des objets propres, différents des entiers.

Comment utiliser le type `CommandWord` pour avancer vers le découplage entre la logique de jeu de zuul et une langue donnée ? L'une des premières améliorations à apporter concerne la série de tests suivants dans la méthode `processCommand` de la classe `Game` :

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

Si `commandWord` obtient le type `CommandWord` au lieu de `String`, ce code peut être réécrit ainsi :

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

Il nous suffit maintenant de faire concorder les commandes tapées par l'utilisateur avec les valeurs correspondantes de `CommandWord`. Ouvrez le projet *zuul-with-enums-v1* pour voir comment nous avons procédé. Le changement le plus significatif a lieu dans la classe `CommandWords`. Au lieu d'employer un tableau de chaînes pour définir les commandes valables, nous utilisons désormais une association entre chaînes et objets `CommandWord` :

```
public CommandWords()
{
    validCommands = new HashMap<String, CommandWord>();
    validCommands.put("aller", CommandWord.GO);
    validCommands.put("aide", CommandWord.HELP);
    validCommands.put("quitter", CommandWord.QUIT);
}
```

La commande saisie par un utilisateur peut maintenant être facilement transformée en un type de valeur énumérée correspondant.