TD 10 : Commande de robots

Objectifs pédagogiques :

- *Design Pattern Command*
- *Design Pattern Factory*

Dans cet exercice, nous allons contrôler des robots qui peuvent se déplacer dans le plan (axes X et Y) et ont un écran pouvant afficher un message. L'implémentation effective des robots importe peu quand il s'agit de les contrôler. Il suffit de savoir qu'ils obéissent à l'interface suivante :

```
public interface IRobot {
   double getX();
   double getY();
   String getMessage();

   boolean moveX(double distance); // true si succès ; false si le robot n'a pas bougé
   boolean moveY(double distance); // true si succès ; false si le robot n'a pas bougé
   void setMessage(String message); // réussit toujours
}
```

Les méthodes `moveX` et `moveY` renvoient un booléen : il est vrai si le robot a pu effectivement avancer de la distance demandée, et faux si un obstacle l'a empêché d'avancer (dans ce cas, on suppose qu'il ne s'est pas déplacé du tout). L'opération de modification du message, `setMessage`, n'échoue jamais.

10.1 *Design Pattern Command*

Nous souhaitons pouvoir définir un programme de déplacement de robots et pouvoir l'exécuter dans un second temps. La construction du programme consiste donc à planifier, i.e., construire une liste de commandes sans appeler directement les méthodes de `IRobot`. Une classe séparée sera chargée d'exécuter le programme en traduisant la liste en appels effectifs à `IRobot`.

Ainsi nous définissons :

– l'interface `ICommand` suivante qui représente une commande :

```
public interface ICommand {
   public boolean execute();
}
```

– la classe `Executor` suivante :

```
public class Executor {
   public static boolean executePlan(List<ICommand> plan) { /* à faire */ }
}
```

Question 1. Donnez le code de la méthode statique `executePlan` de la classe `Executor`. Celle-ci s'interrompt et renvoie `false` à la première commande qui échoue, et renvoie `true` si toutes les commandes ont été exécutées avec succès.

Nous allons à présent nous intéresser à l'implémentation des commandes. Une première implémentation proposée est la suivante :

```
public class Command implements ICommand {
   private int command;
   private IRobot robotArg;
   private double doubleArg;
```

```
    private String stringArg;

    public static final int commandMoveX = 0;
    public static final int commandMoveY = 1;
    public static final int commandSetMessage = 2;

    public boolean execute() { /* à completer */ }
}
```

où la nature de la commande est encodée dans l'entier `command` (0 pour `moveX`, 1 pour `moveY` et 2 pour `setMessage`), le robot concerné est stocké dans `robotArg`, tandis que les arguments de la commande sont stockés dans `doubleArg` ou `stringArg`, selon leur type.

Question 2. Codez la méthode `execute()` de la classe `Command`. Quels sont les inconvénients d'une telle implémentation ?

Notre prochaine solution est d'avoir une classe d'implémentation par type de commande.

Question 3. Montrez comment définir des classes `CommandRobotMoveX`, `CommandRobotMoveY` et `CommandRobotSetMessage` implémentant `ICommand` pour représenter les différents types de commandes à effectuer sur des `IRobot`.

Question 4. Dans notre implémentation, les commandes contiennent une référence à un robot. Discuter les avantages et les inconvénients que cela apporte. Pouvez-vous imaginer des architectures différentes ?

Rappelons que l'exécution d'une commande peut échouer (`execute()` renvoie `false`). Dans ce cas, au lieu de simplement interrompre l'exécution d'une suite de commandes dans `Executor`, nous souhaitons annuler les commandes déjà effectuées pour revenir au point de départ. Il s'agit d'une opération de *rollback*, liée à la notion de *transaction* : elle évite au client de « voir » des états incohérents dus à une prise en compte partielle d'une suite de commandes. Nous souhaitons, au contraire, voire l'exécution d'une suite de commandes comme une action indivisible : soit elle réussit complètement, soit elle échoue complètement et laisse le robot dans son état original.

Pour cela, nous supposons que l'interface `ICommand` est enrichie en une interface supportant à la fois l'exécution et l'annulation d'une commande :

```
public interface IUndoableCommand extends ICommand {
    public boolean unexecute();
}
```

Bien sûr, les commandes seront annulées dans l'ordre inverse de leur ordre d'exécution.

Question 5. Modifiez les classes commandes de la question 3 pour supporter l'interface `IUndoableCommand`.

Question 6. Modifiez `executePlan` pour effectuer l'annulation en cas d'échec.

Question 7. Une fois une action annulée, est-il possible de l'exécuter à nouveau ?

10.2 *Design Pattern Factory* : Fabriquer des commandes

Un inconvénient, pour le client, de la méthode précédente est la multiplication des classes représentant les différentes méthodes de `IRobot`. Ceci nécessite donc de connaître le nom des classes d'implémentation de commandes et d'appeler le constructeur correspondant. De plus, nous perdons la vue globale sur les fonctionnalités des robots qu'on avait en lisant `IRobot`. Nous cherchons donc à regrouper dans une classe unique un ensemble de méthodes permettant au client de construire toutes les commandes disponibles.

Question 8. Proposez une classe `SimpleCommandFactory` qui centralise la construction de toutes vos commandes grâce à des méthodes statiques.

Les commandes créées précédemment ne sont applicables que sur des objets de type `IRobot`. Nous souhaiterions ajouter à notre programme la possibilité de planifier le mouvement d'objets autres que

des `IRobot`. Nous pouvons imaginer l'existence d'interfaces `ITrain`, `IBateau`, `IAvion`, qui proposent des interfaces de mouvement différentes (e.g., `voler()`, `naviguer()`, etc.).

Question 9. Quelles sont les conséquences pour la classe `SimpleCommandFactory` ?

On propose maintenant d'utiliser une fabrique abstraite qui permet de créer des commandes de mouvement et de message indépendamment du type de l'objet concerné.

```
public interface IAbstractCommandFactory {
   public ICommand newCommandMoveX(double distance);
   public ICommand newCommandMoveY(double distance);
   public ICommand newCommandSetMessage(String message);

}
```

Question 10. Codez la classe `ConcreteFactoryIRobot` permettant de créer les commandes des `IRobot`. Quelles modifications faut-il apporter pour supporter la création de commandes pour des `ITrain`, `IBateau`, `IAvion` ? Quelles conséquences pour le client ?