Patrons : *Adapter, Builder,* fabriques

Théorie et pratique de la programmation
Michel Schinz – 2013-04-22

Patron n°6 : *Adapter*

Illustration du problème

Admettons l'existence d'une méthode permettant de mélanger une liste d'éléments :

```
<E> void shuffleList(List<E> list) { … }
```

Est-il possible d'utiliser cette méthode pour mélanger un tableau Java ?

Directement, cela n'est clairement pas possible, les tableaux n'étant pas des listes. Mais existe-t-il un moyen indirect d'y parvenir ?

Solution

Pour permettre l'utilisation d'un tableau Java là où une liste est attendue, il suffit d'écrire une classe qui adapte le tableau en le présentant comme une liste.

Cette classe doit implémenter l'interface `List` du paquetage `java.util` et effectuer les opérations de cette interface directement sur le tableau qu'elle adapte.

Adaptateur pour tableaux

```
import java.util.List;
class ArrayAdapter<E> implements List<E> {
  private E[] array;

  public ArrayAdapter(E[] array) {
    this.array = array;
  }
  public E get(int index) {
    return array[index];
  }
  // ... les 22 autres méthodes de List
}
```


Utilisation de l'adaptateur

Une fois l'adaptateur défini, il est possible de l'utiliser pour mélanger un tableau au moyen de la méthode de mélange de liste :

```
<E> void shuffleList(List<E> list) { … }

String[] array = …;
List<String> adaptedArray =
  new ArrayAdapter<String>(array);
// mélange les éléments de array.
shuffleList(adaptedArray);
```

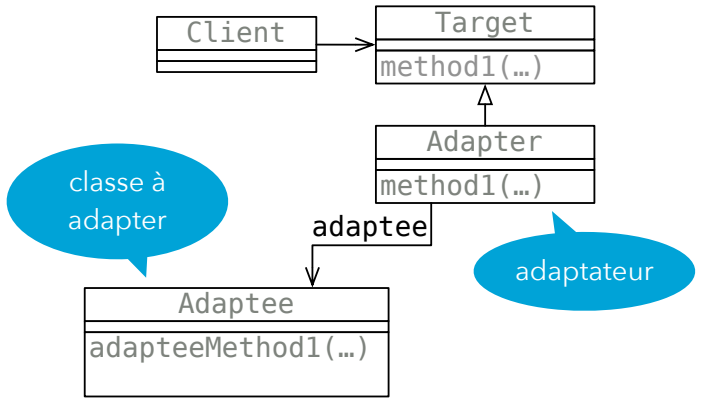

Généralisation

De manière générale, lorsqu'on désire utiliser une instance d'une classe A là où une instance d'une classe B est attendue, il est possible d'écrire une classe servant d'adapteur.

Bien entendu, il faut que le comportement des classes A et B soit relativement similaire, sans quoi l'adaptation n'a pas de sens.

Le patron de conception ***Adapter*** décrit cette solution.

Diagramme de classes



Exemple réel

La classe `java.util.Arrays` offre, sous la forme de méthodes statiques, un grand nombre d'opérations sur les tableaux.
Une de ces méthodes, nommée `asList`, permet justement d'adapter un tableau en liste.
Cette méthode utilise le patron *Adapter* pour faire cette adaptation, exactement comme nous l'avons fait.

`AbstractList` (digression)

`java.util.List`

L'interface `java.util.List` contient 23 méthodes. Ecrire une classe implémentant cette interface représente donc un travail important.
Toutefois, un examen de cette interface montre d'une part que 9 méthodes sont optionnelles, et d'autre part que plusieurs méthodes peuvent facilement s'exprimer en termes d'un petit nombre de méthodes de base.

Opérations optionnelles

Dans la bibliothèque Java, certaines interfaces possèdent des méthodes correspondant à des **opérations optionnelles**.
Une classe implémentant une telle interface peut ne pas fournir ces opérations, c-à-d faire en sorte que les méthodes correspondantes lèvent l'exception `Unsupported-OperationException`.
Exemple que vous connaissez : `remove` de l'interface `Iterator`.

Opérations optionnelles

Dans la bibliothèque Java, les méthodes qui *modifient* les structures de données sont en général optionnelles. Cela permet la définition de structures de données non modifiables.
Ainsi, les méthodes `add`, `addAll`, `clear`, `remove`, `removeAll`, `retainAll` et `set` de l'interface `List` sont optionnelles.

Méthodes de base de `List`

L'interface List possède plusieurs méthodes qui sont facilement exprimables en fonction de méthodes de base.
Par exemple, au moyen de la méthode `size` il est trivial de fournir la méthode `isEmpty` :

```
public boolean isEmpty() {
  return size() == 0;
}
```


La classe `AbstractList`

L'idée de la classe `java.util.AbstractList` est de définir autant de méthodes de `List` que possible en termes d'un petit nombre de méthodes de base, qui sont laissées abstraites. Les méthodes correspondant à des opérations optionnelles lèvent quant à elle l'exception `UnsupportedOperationException`
En héritant de cette classe et en définissant les méthodes de base, on peut ainsi facilement définir un nouveau type de listes.

Adaptateur pour tableaux

La classe `AbstractList` nous permet d'écrire beaucoup plus simplement notre adaptateur pour tableaux.
Au lieu de définir les 23 méthodes de l'interface `List`, il nous suffit d'hériter d'`AbstractList` et de définir les trois méthodes de base requises : `get`, `set` et `size`. C'est tout !

Adaptateur pour tableaux

```
import java.util.AbstractList;
class ArrayAdapter<E>
    extends AbstractList<E> {
  private E[] array;
  public ArrayAdapter(E[] array) {
    this.array = array;
  }
  @Override
  public E get(int index) { ??? }
  @Override
  public E set(int index, E element) { ??? }
  @Override
  public int size() { ??? }
}
```


Patron n°7 : *Builder*

Illustration du problème

Admettons que l'on désire écrire une bibliothèque de calcul matriciel. Un composant central d'une telle bibliothèque est la (ou les) classe(s) modélisant les matrices.

Comme toutes les classes représentant des entités mathématiques, ces classes devraient être non modifiables. Cela implique que la totalité des éléments d'une matrice doivent être spécifiés lors de sa construction. Pour les grosses matrices, ou les matrices creuses, cela peut s'avérer pénible.

Comment résoudre ce problème ?

Solution

Une solution consiste à offrir un bâtisseur de matrice, c'est-à-dire une classe dont le seul but est de stocker le contenu d'une matrice en cours de construction.

Le bâtisseur est modifiable, et possède des méthodes pour changer les différents éléments de la matrice en cours de construction. Il possède également une méthode pour construire la matrice.

Matrices

```
public interface Matrix {
  double get(int r, int c);
  Matrix transpose();
  Matrix inverse();
  Matrix add(Matrix that);
  Matrix multiply(double that);
  Matrix multiply(Matrix that);
  …
}
public class DenseMatrix implements Matrix {
  …
}
public class SparseMatrix implements Matrix {
  …
}
```


Bâtisseur de matrice

```
public class MatrixBuilder {
  private double[][] elements;
  public MatrixBuilder(int r, int c) {
    elements = new double[r][c];
  }
  public double get(int r, int c) {
    return elements[r][c];
  }
  public void set(int r, int c, double v) {
    elements[r][c] = v;
  }
  public Matrix build() {
    return new DenseMatrix(elements);
  }
}
```


Types de matrices

Outre le fait que le bâtisseur permet de construire une matrice en plusieurs étapes, il permet également de choisir une représentation pour la matrice qui soit appropriée à son contenu !

Par exemple, une matrice creuse – c-à-d dont la plupart des éléments valent 0 – peut être représentée au moyen d'une table associative contenant les éléments non-nuls. Une matrice dense peut être représentée par un tableau bi-dimensionnel.

Le choix entre les deux représentations peut être fait par le bâtisseur au moment de la création de la matrice.

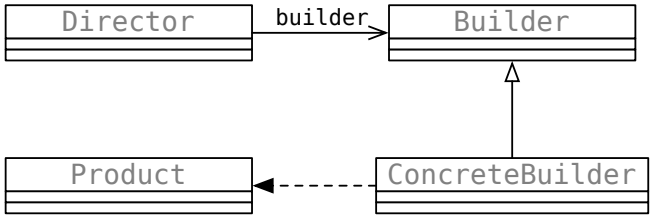
Bâtisseur intelligent

```
public class MatrixBuilder {
  … // comme avant
  public Matrix build() {
    if (density() > 0.5)
      return new DenseMatrix(elements);
    else
      return new SparseMatrix(elements);
  }
}
```

Généralisation

Chaque fois que le processus de construction d'une classe est assez difficile pour que l'on désire le découper en plusieurs étapes, on peut utiliser un objet pour stocker l'état de l'objet en cours de construction.
C'est l'idée du patron de conception ***Builder***.

Diagramme de classes



Exemple réel

La classe `String` (de `java.lang`) modélise les chaînes de caractères, qui ne sont pas modifiables.
La classe `StringBuilder` sert de bâtisseur pour les chaînes de caractères. Elle possède entre autres une méthode `append` pour ajouter la représentation textuelle d'un objet à la chaîne en cours de construction, et la méthode `toString` pour obtenir la chaîne construite.
(Notez que la concaténation de chaînes de caractères au moyen de l'opérateur `+` est traduite via `StringBuilder`).

Patron n°8 : *Factory Method*

Illustration du problème

Plusieurs jeux récents sont extensibles par programmation : l'utilisateur a la possibilité de modifier ou améliorer certains de leurs aspects en écrivant du code.
Dans un langage orienté-objets, une manière naturelle de permettre ce genre d'extensions consiste à laisser l'utilisateur définir ses propres sous-classes des classes principales du jeu.

Illustration du problème

Pour rendre le problème plus concret, imaginons un jeu dans lequel chaque joueur possède un bateau, modélisé par une classe `Ship`.
Afin d'étendre le jeu, un utilisateur pourrait définir sa propre sous-classe de `Ship`, p.ex. `MagicShip`.
Une question se pose alors : comment s'assurer que tous les bateaux créés lors du jeu soient des instances de `MagicShip` et pas de `Ship` ?

Illustration du problème

```
class Game {
  public Game(int pCount) {
    Player[] players = new Player[pCount];
    Ship[] ships = new Ship[pCount];
    for (int i = 0; i < pCount; ++i) {
      players[i] = new Player();
      ships[i] = new Ship();
      ships[i].setPosition(...);
      ships[i].setOwner(players[i]);
    }
    // ... initialisation complète du jeu
  }
}
```

problème

Solution

Afin de résoudre ce problème, une solution est de définir une méthode dans la classe `Game` pour créer un nouveau bateau, puis de l'utiliser systématiquement à la place de l'opérateur `new`.
Cette méthode peut être redéfinie dans une sous-classe afin de créer un autre type de bateau.
Une telle méthode est appelée une **méthode fabrique** (*factory method*) ou **constructeur virtuel** (*virtual constructor*).

Jeu extensible

```
class Game {
  Ship newShip() { return new Ship(); }
  public Game(int pCount) {
    Player[] players = new Player[pCount];
    Ship[] ships = new Ship[pCount];
    for (int i = 0; i < pCount; ++i) {
      players[i] = new Player();
      ships[i] = newShip();
      ships[i].setPosition(...);
      ships[i].setOwner(players[i]);
    }
    // ... initialisation complète du jeu
  }
}
```

fabrique

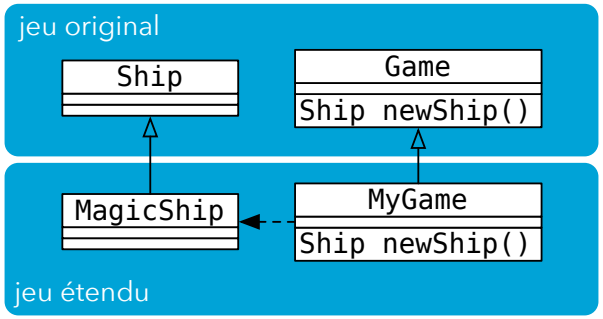
tous les bateaux sont créés via la fabrique

Jeu étendu

```
class MagicShip extends Ship {
  // ...
}
class MyGame extends Game {
  @Override
  Ship newShip() {
    return new MagicShip();
  }
  public MyGame(int playersCount) {
    super(playersCount);
  }
}
```

nouvelle version de la méthode fabrique

Diagramme de classes



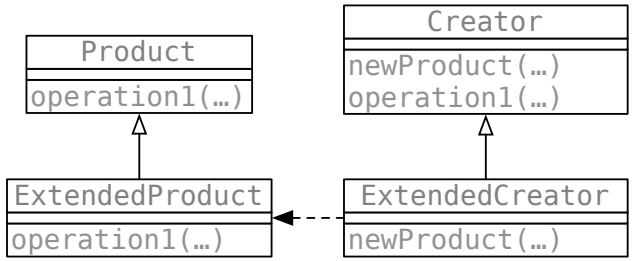
Généralisation

Chaque fois qu'une classe doit créer une instance d'une autre classe qui n'est pas connue à l'avance, elle peut utiliser une méthode fabrique.

En redéfinissant la méthode fabrique dans une sous-classe, il est possible de spécifier la classe exacte à créer.

Le patron de conception ***Factory Method*** décrit cette solution.

Diagramme de classe



Chaque fois que le créateur `Creator` désire créer un produit `Product`, il utilise la méthode `newProduct`.

Exemples réels

Les tests unitaires que nous avons écrits pour les collections utilisaient des méthodes fabriques pour créer les instances, afin de pouvoir réutiliser les tests. Exemple :

```
abstract class ListTest {
  abstract <E> List<E> newList();
  // chaque test utilise newList pour
  // obtenir une liste à tester.
  …
}
```


Patron n°9 :
Abstract Factory

Illustration du problème

Continuons avec notre exemple de jeu extensible, mais en imaginant une situation plus complexe. Plutôt que de n'avoir que la seule classe `Ship` qui soit extensible par l'utilisateur, imaginons que nous en ayons trois: `Ship`, `Treasure` et `Island`.

Posons-nous la même question qu'avant : comment s'assurer que chaque fois qu'un bateau, trésor ou île est créé, la bonne classe est utilisée ?

Solution

Une solution consiste bien entendu à définir trois méthodes fabriques dans la classe `Game`.

Toutefois, lorsque le nombre de méthodes fabriques commence à devenir important et que les classes dont elles créent des instances sont liées, il peut devenir préférable de les extraire de la classe `Game` pour les mettre dans une interface séparée.

Une telle interface, qui ne contient que des méthodes fabriques, est appelée **fabrique abstraite** (*abstract factory*)

Fabrique abstraite

Pour notre exemple, la fabrique abstraite se présente ainsi :

```
interface GameFactory {
  Ship newShip();
  Island newIsland();
  Treasure newTreasure();
}
```

On peut utiliser un objet implémentant cette interface pour créer tous les bateaux, îles et trésors nécessaires à un jeu.

Jeu extensible

```
class Game {
  private final Ship[] ships;
  private final Island[] islands;
  private final Treasure[] treasures;
  public Game(GameFactory factory) {
    ships = new Ship[] {
      factory.newShip(), … };
    islands = new Island[] {
      factory.newIsland(), … };
    treasures = new Treasure[] {
      factory.newTreasure(), … };
  }
}
```

tous les objets extensibles sont créés via la fabrique

Fabrique concrète

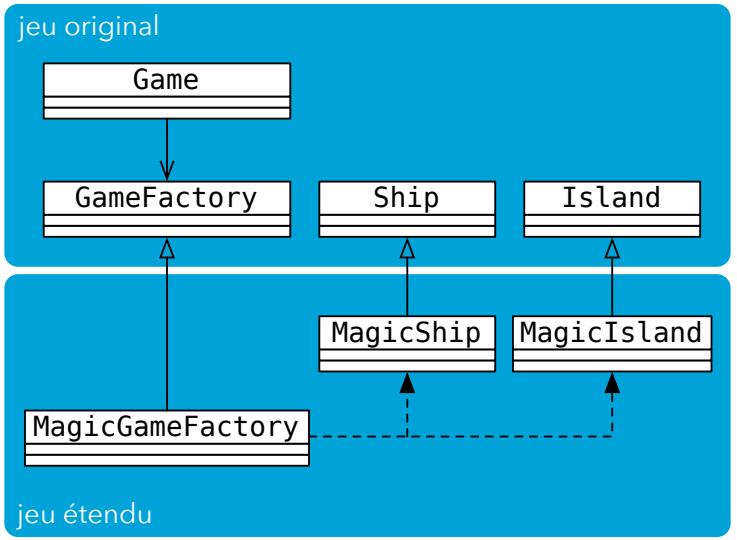
```
class MagicShip extends Ship { … }
class MagicIsland extends Island { … }
class MagicTreasure extends Treasure { … }
class MagicGameFactory
    implements GameFactory {
  public Ship newShip()
    { return new MagicShip(); }
  public Island newIsland()
    { return new MagicIsland(); }
  public Treasure newTreasure()
    { return new MagicTreasure(); }
}
```

Jeu étendu

Pour obtenir une version magique du jeu, il suffit de créer une instance de la classe `Game` en lui passant la fabrique correspondante :

```
// Classe principale de la version magique
// du jeu.
class MagicGameMain {
  public static void main(String[] args) {
    // crée et démarre le jeu
    new Game(new MagicGameFactory());
  }
}
```

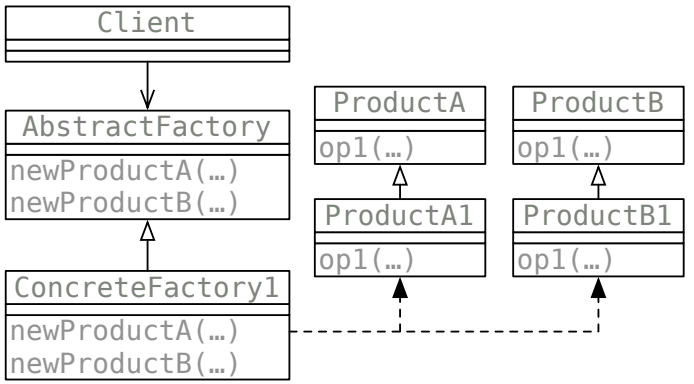

Diagramme de classes



Généralisation

Chaque fois qu'une classe doit créer des instances d'une famille de classes apparentées mais non connues d'avance, elle peut utiliser une fabrique abstraite.
Le patron de conception ***Abstract Factory*** décrit cette solution.

Diagramme de classes



Types de fabriques

Les patrons *Factory Method* et *Abstract Factory* sont très proches. En fait, une fabrique abstraite n'est rien d'autre qu'une collection de méthodes fabriques empaquetées dans un objet.

On utilise en général une fabrique abstraite dès le moment où il doit être possible de créer des instances d'une famille de classes liées.

Exemple réel (ou presque)

La classe `BorderFactory` est une fabrique pour les bordures Swing. Mais attention : elle n'est pas conçue selon le patron *Abstract Factory*, car toutes les méthodes de construction sont statiques – donc non redéfinissables – et la classe elle-même n'est pas abstraite.

`BorderFactory` résout en fait un problème différent de celui résolu par *Abstract Factory*. Son but est d'éviter de créer trop d'instances des bordures, en essayant de les partager, et pas de faire abstraction de la classe des bordures.