UNE VISION DES DESIGN PATTERNS

(brouillon)

Revision: 1.36 — Date: 2012/11/07 11:16:15

Deux approches pour découvrir les *design patterns* : la première dit qu'il faut les comprendre pour comprendre la programmation par objets (héritage, appel de méthode, etc.) tandis que la seconde dit qu'il faut comprendre la programmation par objets pour comprendre les *design patterns*. Ici, on adopte la seconde approche, donc on découvre les *patterns* par nécessité.

1 Introduction

Les *design patterns* désignent des éléments **solutions** pour des **problèmes** souvent rencontrés, de la même manière qu'en mathématiques on voit que $(a+b)^2$ est une **identité remarquable** qui se développe en $a^2+2ab+b^2$: on choisit d'utiliser ces propriétés quand on en a besoin. Il n'est nul besoin de connaître les *design patterns* ou leurs noms d'une manière générale pour programmer, tout comme il n'est nul besoin de reconnaître les identités remarquables, sauf lorsque l'on désire en parler.

> En physique, il n'y a pas besoin de connaître la première loi de Newton [1] si on connaît la deuxième loi de Newton [2], puisque selon la seconde loi, une somme des forces égale au vecteur nul indique soit une masse nulle (ce qui rend l'objet nul, dans cette physique là), soit une accélération nulle (ce qui signifie une variation nulle de vitesse, ce qui est synonyme d'une vitesse constante). De même il n'est pas nécessaire de savoir quelle loi est la première la seconde ou la troisième, sauf quand on les désigne ainsi. Il en va de même pour les *design patterns* : on les retrouve par l'utilisation, il n'y a pas besoin de les connaître, sauf quand on désigne un *design pattern* par son nom auquel cas c'est très obscur si on n'a pas idée de ce qu'il désigne.

Cependant, il est bien pratique de connaître les solutions courantes aux problèmes courants, dans le but d'uniformiser l'architecture des programmes et ainsi faciliter la compréhension d'un code écrit par autrui. Ces solutions ne doivent pas être vues comme parfaites et uniques, puisqu'elles sont souvent le résultat de compromis (composition, héritage, découpage, délégation, visibilité, etc.).

Les **problèmes** dont il est question sont en général inhérents à la programmation par objets qui pose des contraintes architecturales. Mais tous les langages à objets ne sont pas équivalents. Plusieurs paradigmes de programmation par objets existent. Par exemple, Java et Objective C n'autorisent pas l'héritage multiple tandis que C++ et OCaml l'autorisent. Un autre exemple est celui du modèle de programmation par objets à prototypes de JavaScript, qui est analogue à celui d'OCaml, mais se distingue fondamentalement par son typage entièrement dynamique alors que le typage d'OCaml est entièrement statique. Le modèle à objets d'OCaml ne permet pas de pratiquer le *downcast* du fait du typage statique, car le langage permet de ne pas en avoir besoin [3].

Dans le cadre des cours de programmation par objets, il est souvent demandé de connaître au moins les noms des *design patterns* les plus utilisés. Les cours peuvent éventuellement en donner une liste exhaustive à « maîtriser ».

D'autre part, connaître les noms des *design patterns* permet de les désigner facilement plutôt que de devoir systématiquement décrire toute l'architecture à petits grains d'une application : c'est plus facile de dire qu'on a utilisé un « visiteur » pour faire un parcours d'arbres, plutôt que décrire en détails les classes et méthodes implantées : un mot *versus* de longues descriptions truffées d'embûches !

1. « principe de l'inertie » qui dit que si la somme des forces appliquées à un objet est égale au vecteur nul, alors la vitesse de l'objet est nulle
2. « théorème du centre d'inertie » qui dit que la somme des forces appliquées à un objet est égale au produit de la masse par l'accélération
3. Il existe cependant des extensions permettant d'utiliser l'opération de *downcast* avec OCaml, mais ce n'est pas offert par défaut.

1.1 Visibilité des attributs et méthodes

Spécifier la visibilité des attributs et méthodes permet surtout de restreindre les libertés d'action afin d'une part de réduire l'étendu géographique des modifications à apporter en cas d'évolution du code, et d'autre part de boucher certaines potentielles failles de sécurité (exemple : un scénario où un tableau devient vide alors qu'il ne devrait jamais l'être, du fait d'un attribut public au lieu de privé).

L'ordre d'apparition des design patterns *dans ce document devrait correspondre* ***à peu près*** *à leur ordre d'apparition dans les cours 9 et 10 de LI314-2010oct.*

2 Composite

Ce cas de figure se présente lorsque plusieurs classes différentes ont besoin d'être du même type. Par exemple, l'implantation d'une structure arborescente pose des contraintes souvent résolues à l'aide du composite puisqu'une **feuille** doit être de même type qu'un **noeud**, les deux étant des **arbres**... Par exemple, si on veut avoir une liste d'arbres, la liste pourra être de type `List<Arbre>` (ou `List<IArbre>` si le type commun est une interface plutôt qu'une classe abstraite ou non abstraite), auquel cas bien sûr `Feuille` pourra implanter l'interface `IArbre` ou hériter de la classe `Arbre`, et de même pour `Noeud`.

Dans le schéma usuel du Composite :

- On définit une interface ou bien une classe abstraite (usuellement nommée `Composant` ou en anglais `Component` dans le schéma type), pour représenter les points communs des feuilles et des noeuds. Ceci permet aux feuilles et noeuds d'être de même type (et qui ne soit pas `Object`).
- Par définition, les feuilles n'ont pas de fils, et les noeuds ont des fils et ce sont donc ces derniers qui ont les méthodes d'ajouts et de retrait (pour les fils) sauf si le nombre de fils est constant (auquel cas c'est normalement le constructeur qui se charge de récupérer les fils).

Mais il n'est absolument pas nécessaire que les différentes classes définissent une structure arborescente. On peut très bien par exemple n'avoir que des différents types de feuilles (donc sans noeuds) ou encore des feuilles ainsi que des noeuds, etc.

Les utilisateurs utiliseront souvent le type composant (`Composant` ou `Component`), puisque dans le cas contraire, on remettrait en question l'utilité de la mise en place du mécanisme.

Exemple : on définit une classe abstraite `Peinture` (on choisit une classe abstraite plutôt qu'une interface, et ce choix est arbitraire). On définit 3 classes héritant de `Peinture` : `PeintureGouache`, `PeintureAcrylique`, `PeinturePastelle`. On peut ainsi faire une Galerie (par exemple stockée dans une structure `ArrayList<Peinture>`) contenant différentes peintures de type `Peinture`, qui sont à la gouache, à l'acrylique ou en pastelle...

Remarque on peut voir la composition comme signifiant « mettre ensemble », comme quand on mélange du bois avec du plastique pour obtenir du bois composite par exemple.

3 Fabrique (Factory)

Lorsqu'on souhaite regrouper les créations d'objets (avec `new`) géographiquement, la classe qui contient ces créations est une fabrique (en anglais *factory*). Créer tous les objets au même endroit est assez pratique, quand on compare à l'alternative qui consiste à disséminer les créations d'objets. La mise au point (*debug*) est quand même plus simple quand les objets sont plus faciles à localiser et que le flot de contrôle ne zigzague par trop entre les classes.

Parfois la classe qui joue le rôle de la fabrique est celle qui contient la méthode « point d'entrée » `public static void main(String[])`. Un problème est qu'il n'y a pas toujours de méthode `main` ! Utiliser

`main` pour la fabrique est donc un cas particulier (certes souvent utilisé dans les programmes comportant peu de lignes de code).

Les fabriques proposent souvent des méthodes de créations d'objets (ce sont donc des indirections : les méthodes renvoient des objets qu'elles créent avec **new**). Ces méthodes ont rarement uniquement le rôle d'ajouts d'indirections. Souvent elles vont permettre de masquer la manière dont on crée un objet. Par exemple, si on veut créer un objet de type `ClasseAbstraite`, la méthode pourra être

```
public ClasseAbstraite getClasseAbstraite() {
 return new ClasseConcrete(42);
} // en supposant que ClasseConcrete est un sous-type de ClasseAbstraite
```

Ainsi, on ne voit pas de l'extérieur comment est créée l'objet de type `ClasseAbstraite`. (La cachotterie est tout un art en programmation par objets.)

4 Adaptateur (Adapter)

L'adaptateur est un motif la plupart du temps assez simple qui consiste en implanter une classe, satisfaisant une interface, et déléguant le plus possible à une ou plusieurs classes existantes qui ne satisfont pas précisément les contraintes de l'interface à implanter. Certaines délégations peuvent être complexes (c'est-à-dire pas un simple appel de méthode).

Un scénario ce qui pourrait (malheureusement) se passer en France : une entreprise italienne développe ses bibliothèques avec des noms de méthodes en italien, des français qui utilisent ces bibliothèques font des adptateurs pour coller avec leurs interfaces qui ont des noms de méthodes en français, et des allemands rachètent le travail des français et décident (enfin ! !) de faire un adaptateur vers des noms en anglais (l'anglais est la seule langue qui a une chance de mettre tout le monde d'accord, les autres n'en ont simplement aucune...).

Autre scénario qui serait malheureux : prendre du code avec du « franglais » et faire des adaptateurs vers du français ou de l'anglais, histoire d'arrêter les affreux mélanges.

Ces problèmes arrivent notamment quand on n'a pas le droit ou même la possibilité de modifier le code source, et qu'on est obligé d'utiliser des bibliothèques pré-compilées. (Faire du *refactoring* de fichiers `.class` n'est pas impossible mais ce n'est pas trivial à faire...)

5 Singleton

Le singleton est utilisé quand on veut empêcher la création de plusieurs instances d'une classe donnée. Pour ce faire, on privatise le constructeur afin d'interdire aux autres classes d'instancier la classe. En conséquence, le seul endroit où peut se faire l'instanciation devient la la classe elle-même. Attention, on ne peut pas non plus utiliser une méthode non statique pour l'instanciation puisque les méthodes non statiques ne sont accessibles que depuis un objet instance de la classe. Il ne reste plus que deux possibilités :

– soit l'instanciation au lieu de déclaration :

```
private LaClasseSingleton instanceDeLaClasseSingleton=new LaClasseSingleton();
public final static Singleton getInstance() {
   return instanceDeLaClasseSingleton;
}
```

– soit l'instantiation dans une méthode statique qui se charge de vérifier que l'instantiation n'a bien lieu qu'une seule fois, cette méthode statique pourra être la méthode permettant d'accéder à l'instance (par exemple `getInstance`). Si ce n'est pas elle, c'est plus compliqué puisqu'il faut alors gérer le cas d'un retour à `null` au niveau de `getInstance`, ce qui n'est pas génial.

```
private LaClasseSingleton instanceDeLaClasseSingleton;
public final static Singleton getInstance() {
```

```
3   if (instanceDeLaClasseSingleton == null)
4     instanceDeLaClasseSingleton = new LaClasseSingleton();
5   return instanceDeLaClasseSingleton;
6 }
```

La solution présentée permet de limiter le nombre d'instantiations à exactement 1. Il n'est pas compliqué d'adapter le concept à la création d'exactement k classes, k étant une constante définie par les besoins. Si on ne limite pas à 1 ni à k, alors le singleton ne doit pas être appliqué puisqu'il ne servirait alors à rien (sauf se compliquer la vie) à moins que le but ne soit l'obfuscation du code...

6 Proxy

Les différents proxys sont des sortes de délégués qui ajoutent des opérations intermédiaires entre les méthodes d'une classe et leurs appelants. Cela permet de retarder des calculs, d'ajouter des contrôles, etc.

7 Décorateur (Decorator), ou comment étendre sans héritage

Le décorateur consiste à étendre une classe sans utiliser l'héritage, soit parce qu'on souhaite éviter d'utiliser l'héritage (choix d'implantation), soit parce qu'il n'est simplement pas possible d'utiliser l'héritage (contrainte forcée) (un exemple : quand on veut étendre la classe `A` mais qu'on hérite déjà de la classe `B` et qu'on n'a pas d'héritage multiple ; un autre exemple : on veut étendre la classe `A` mais elle est finale et on ne peut donc pas en hériter).

Pour étendre une classe sans utiliser l'héritage, il n'y a pas beaucoup de choix : il faut quelque part avoir un attribut qui est une instance de la classe qu'on souhaite étendre (sinon ça veut dire qu'on n'a pas accès à ce qu'on veut étendre et donc on ne risque pas de l'étendre). Ensuite, selon que la classe `CAE` [4] à étendre est une classe concrète ou une classe abstraite ou encore une interface, la classe contenant le lien vers `CAE` pourra être une interface ou une classe abstraite, ou même directement une classe concrète. Dans tous les cas, la classe concrète (car il en faut bien une tôt ou tard) aura un constructeur paramétré par un objet de type `CAE`.

8 Visiteur (Visitor)

> Pour éviter d'utiliser le modèle du visiteur, on utilise une cascade de (**`if`** + **`instanceof`**) sur les classes des objets avec probablement des transtypages descendants (ou *downcast* an anglais) dans certaines branches. Mais l'utilisation de **`instanceof`** est considéré comme assez coûteuse mais surtout comme une mauvaise pratique de la programmation par objets.

Le modèle du visiteur utilise le *dispatch* [5] de la programmation par objet (et la surcharge) comme filtre sur le type des objets appartenant à une sous-classe d'une classe commune (ça peut être avec une interface commune plutôt qu'une classe commune).

Ainsi, le visiteur est celui qui implante les différentes méthodes `visit` qui sont le point de *dispatch* (même type de retour mais différent type d'entrée). C'est donc lui qui implante le **traitement algorithmique**.

Chaque élément doit implanter une méthode triviale qui se contente d'accepter la visite du visiteur. En fait, celui qui veut invoquer le traitement algorithmique sur un objet `o` doit invoquer `o.accept(v)`, où `v` est le visiteur qui possède la méthode `visit`. Ainsi, `o` va invoquer `v.visit(o)` via **`this`** [6].

4. le choix du nom `CAE` est arbitraire et n'a aucune importance, il sert juste à tenter de simplifier le texte...
5. *dispatch* : le mécanisme de résolution des appels de méthodes.
6. Ça veut dire que la méthode `visit` de `v` va être invoquée par `o` à l'aide de `v.visit(this)`, le **`this`** référant à `o` bien sûr.

On peut voir cela comme le fait que `o.visit(o)` ne peut pas fonctionner directement, parce que ça impliquerait que `o` possède une méthode `visit` compatible avec `o`, ce qui pose problème car il lui faudrait aussi les méthodes `visit` compatibles avec les autres types. Ce qui impliquerait que `o` implante les méthodes `visit`, ce qui veut dire que plein de classes (si pas « plein » alors probablement nul besoin du modèle des visiteurs) implantent ces même méthodes, donc duplication de code ou bien grande verbosité à cause des délégations abusives... (surtout qu'en Java, on n'a pas d'héritage multiple complet [7]).

9 Visiteur (le retour)

9.1 Exemple avec des Arbres Binaires

Prenons l'exemple des arbres binaires (parce que c'est un des exemples les plus simples de structures arborescentes).

Caractéristiques de la structure de données abstraite. L'important est que les arbres binaires ont des **noeuds** et des **feuilles**. Un arbre binaire vide est composé d'une feuille. Les noeuds portent avec eux des valeurs (sinon ça fait des arbres squelettiques moins intéressants). Chaque noeud possède exactement deux fils (qui peuvent être des feuilles).

9.1.1 Comment faire sans visiteur ?

Implantation 1 : une seule classe pour les feuilles et les noeuds. On choisit de proposer une seule classe pour représenter à la fois les feuilles et les noeuds.

```
class ArbreBinaire {
  private ArbreBinaire filsGauche;
  private ArbreBinaire filsDroit;
  private Boolean estUneFeuille;
  private Object valeur;
  // pour créer un noeud
  ArbreBinaire(Object valeur, ArbreBinaire gauche, ArbreBinaire droit){
    this.estUneFeuille = false;
    this.filsGauche = gauche;
    this.filsDroit = droit;
    this.valeur = valeur;
  }
  // pour créer une feuille
  ArbreBinaire() {
    this.estUneFeuille = true;
  }
  // + 4 get methods + 3 set methods ...
}
```

Problèmes.

- pour chaque méthode `get*` il faut vérifier si on est dans un noeud ou une feuille. De même pour chaque méthode `set*`. Il faut gérer les erreurs lorsque les `get*` ou `set*` reçoivent des mauvais paramètres. Exemple : `uneFeuille.getValeur()` doit entraîner une erreur puisqu'une feuille ne

7. L'héritage multiple en Java fonctionne uniquement sur les interfaces, ce qui n'est pas un mécanisme d'héritage multiple complet. L'héritage multiples de classes normales est interdit, que ce soit au niveau du code Java ou dans la JVM. Dans la JVM, il n'y a pas de quoi exprimer nativement l'héritage multiple.

peut pas avoir de valeur. L'implantation est **alourdie** et il y a beaucoup de vérifications dynamiques à implanter à la main.
Le typage statique de Java permet pourtant de pouvoir détecter les erreurs du genre `uneFeuille.getValeur()` (il faut bien sûr modifier le choix d'implantation).
- `valeur` n'est pas générique, mais ce n'est pas difficile de remédier à cela, il suffit de rendre la classe paramétrée.

```
abstract class ArbreBinaire<A> { }
class ArbreBinaireNoeud<A> extends ArbreBinaire<A> {
  private ArbreBinaire<A> filsGauche;
  private ArbreBinaire<A> filsDroit;
  private A valeur;
  ArbreBinaireNoeud(A valeur, ArbreBinaire<A> gauche, ArbreBinaire<A> droit){
    this.filsGauche = gauche;
    this.filsDroit = droit;
    this.valeur = valeur;
  }
  // + 3 get methods + 3 set methods ...
}
class ArbreBinaireFeuille<A> extends ArbreBinaire<A> { }
```

Voilà, cette fois-ci on a une distinction facile entre les noeuds et les feuilles, car on peut déterminer statiquement si on peut invoquer les méthodes ou non. La classe supérieure aux noeuds et feuilles permet d'implanter des traitements qui ne sont statiquement valides que sur les arbres binaires (sans elle, la classe supérieure serait `Object`, et ça serait mauvais car il faudrait vérifier dynamiquement qu'on est bien une feuille ou un noeud).

Remarque : on a paramétré la classe pour qu'elle soit générique vis-à-vis du contenu de l'arbre (et ainsi éviter de forcer le transport de valeurs de type `Object` qui est très (trop) général.)

Problèmes
- Si on veut implanter un algorithme de traitement sur les arbres, il faut soit modifier 3 classes (ou bien les étendre mais ça complique énormément la tâche si on a plusieurs implantations qu'on veut utiliser), soit déporter l'algorithme hors de ces classes mais utiliser le transtypage dynamique.
  - Par exemple, voici une implantation intrusive du comptage du nombre de feuilles.

```
abstract class ArbreBinaire<A> { abstract int nombreFeuilles(); }
class ArbreBinaireNoeud<A> extends ArbreBinaire<A> {
  private ArbreBinaire<A> filsGauche;
  private ArbreBinaire<A> filsDroit;
  private A valeur;
  ArbreBinaireNoeud(A valeur, ArbreBinaire<A> gauche, ArbreBinaire<A> droit){
    this.filsGauche = gauche;
    this.filsDroit = droit;
    this.valeur = valeur;
  }
  // + 3 get methods + 3 set methods ...
  int nombreFeuilles() {
    return this.filsGauche.nombreFeuilles() + this.filsDroit.nombreFeuilles();
  }
}
class ArbreBinaireFeuille<A> extends ArbreBinaire<A> {
  int nombreFeuilles() {
    return 1;
```

```
  }
}
```

– Par exemple, voici une implantation non-intrusive du comptage du nombre de feuilles (où les méthodes sont statiques).

```
abstract class ArbreBinaire<A> { }
class ArbreBinaireNoeud<A> extends ArbreBinaire<A> {
  private ArbreBinaire<A> filsGauche;
  private ArbreBinaire<A> filsDroit;
  private A valeur;
  ArbreBinaireNoeud(A valeur, ArbreBinaire<A> gauche, ArbreBinaire<A> droit){
    this.filsGauche = gauche;
    this.filsDroit = droit;
    this.valeur = valeur;
  }
  // + 3 get methods + 3 set methods ...
}
class ArbreBinaireFeuille<A> extends ArbreBinaire<A> { }
class ArbreBinaireCompteFeuilles {
  static <A> int nombreFeuilles(ArbreBinaire<A> a) {
    if (a instanceOf ArbreBinaireFeuille<A>) {
      return 1;
    } else {
      ArbreBinaireNoeud<A> noeud = (ArbreBinaireNoeud<A>) a; // ICI C'EST BOF
      return nombreFeuilles(noeud.getGauche()) + nombreFeuilles(noeud.getDroit()
    }
  }
}
```

Ce n'est pas génial d'être obligé utiliser le transtypage dynamique. Le compilateur ne sait pas vérifier statiquement que ça fonctionnera à l'exécution, donc il ne nous indiquera rien si jamais on a fait une erreur. Si on a fait une erreur, on le verra à l'exécution et ça pourra potentiellement être difficile de trouver la provenance exacte de l'erreur.

9.1.2 Comment faire avec un visiteur ?

Implantation 3 : le modèle du visiteur, qui combine les deux aspects positifs de l'implantation 2 : être garder le typage entièrement statique (en étant un peu intrusif mais pas trop), et déporter les algorithmes de parcours en dehors des classes. Voici la classe qui va se charger de discriminer le vrai type de l'objet (entre une feuille et un noeud). En fait, on utilise le dispatch pour filtrer le type de l'objet.

```
abstract class ArbreBinaireVisitor<A> {
  public abstract void visit(ArbreBinaireFeuille<A> f);
  public abstract void visit(ArbreBinaireNoeud<A> n);
}
```

Les algorithmes devront implanter cette classe abstraite. Et il faut maintenant s'introduire dans les classes existantes.

```
abstract class ArbreBinaire<A> {
  public abstract void accept(ArbreBinaireVisitor<A> visitor);
}
class ArbreBinaireNoeud<A> extends ArbreBinaire<A> {
  // + 3 fields + 1 constructor + 3 get methods + 3 set methods
  public void accept(ArbreBinaireVisitor<A> visitor) {
    visitor.visit(this);
```

```
  }
}
class ArbreBinaireFeuille<A> extends ArbreBinaire<A> {
  public void accept(ArbreBinaireVisitor<A> visitor) {
    visitor.visit(this);
  }
}
```

> ***Détail technique important*** : pourquoi avons-nous deux méthodes `visit` qui s'appliquent à `ArbreBinaireFeuille` et `ArbreBinaireNoeud` mais jamais à `ArbreBinaire` ? On peut pourtant bien imaginer qu'on crée des objets de type `ArbreBinaire` via des instances de `ArbreBinaireFeuille` et `ArbreBinaireNoeud`.
> La réponse est en fait toute simple : les seuls qui invoquent la méthode `visit` sont les méthodes qui sont implantées au sein même des deux classes concrètes `ArbreBinaireFeuille` et `ArbreBinaireNoeud`, qui se donnent en argument (c'est-à-dire qu'elles invoquent `visit` avec `this` en argument). Donc on n'est jamais dans un cas où on invoque `visit` avec un objet dont on ne sait pas statiquement si c'est un noeud ou une feuille.

Maintenant, on peut implanter l'algorithme sans toucher à ces classes.

```
class CompteFeuilles {
  private class CompteFeuillesVisitor extends ArbreBinaireVisitor {
    int result = 0;
    public void visit(ArbreBinaireFeuille f){
        result++;
    }
    public void visit(ArbreBinaireNoeud n){
        n.getGauche().accept(this); // attention ici
        n.getDroit().accept(this);  // this n'est pas toujours utilisable
    }
    public int getResult() {
        return this.result;
    }
  }
}

// utilisation :
//   CompteFeuillesVisitor c = new CompteFeuillesVisitor().visit(unArbre);
//   int nombreFeuilles = c.getResult();
```

On peut remarquer dans cette implantation que l'utilisation de `this` évite de devoir créer une nouvelle instance de la classe `CompteFeuillesVisitor`. Ceci n'est valable ici que parce que ce n'est pas un problème de partager le champ `result`.

On peut reprocher aux méthodes `visit` de ne rendre que des `void`. Pourquoi `void` ? Parce que sinon, il faudrait utiliser `Object`. Mais si on utilise `Object`, ça veut dire que tôt ou tard il y aura besoin de transtypage dynamique. Alors autant procéder par effet de bord en utilisant `void` et ainsi empêcher le retour direct de valeurs.

Mais en fait, on peut mieux faire, grâce aux méthodes génériques de Java.

```
// A est le paramètre générique des arbres binaires
//   et le type des valeurs transportées

// R est le paramètre générique du visiteur
//   et son type de retour

```

```
abstract class ArbreBinaireVisitor<A,R> {
  public abstract R visit(ArbreBinaireFeuille<A> f);
  public abstract R visit(ArbreBinaireNoeud<A> n);
}

abstract class ArbreBinaire<A> {
  public abstract <R> R accept(ArbreBinaireVisitor<A,R> visitor);
}
class ArbreBinaireNoeud<A> extends ArbreBinaire<A> {
  // + 3 filds + 1 constructor + 3 get methods + 3 set methods
  public <R> R accept(ArbreBinaireVisitor<A,R> visitor) {
    return visitor.visit(this);
  }
}
class ArbreBinaireFeuille<A> extends ArbreBinaire<A> {
  public <R> R accept(ArbreBinaireVisitor<A,R> visitor) {
    return visitor.visit(this);
  }
}
```

Et voici une utilisation du modèle visiteur générique :

```
class CompteFeuilles {
  private class CompteFeuillesVisitor
        extends ArbreBinaireVisitor<Object, Integer> {
    public Integer visit(ArbreBinaireFeuille<Object> f){
        return 1;
    }
    public Integer visit(ArbreBinaireNoeud<Object> n){
        return n.getGauche().accept(this) // "this" est un cas particulier
            +  n.getDroit().accept(this); // ici. Ça veut dire qu'on utilise
                                          // l'algorithme "courant" et qu'il
                                          // est réentrant.
    }
  }
}

// utilisation :
//   int nombreFeuilles = new CompteFeuillesVisitor().visit(unArbre);
```

9.2 Difficultés de ce modèle

- L'algorithme qu'on veut implanter est déporté avec une indirection.
- Le dispatch, qui est une caractéristique du langage, est utilisé comme outil algorithmique, et le site du dispatch est déporté.
- Le nom du modèle visiteur n'est éloquent que lorsqu'il est vraiment bien compris.
- La généricité implique l'utilisation des méthodes génériques.
- Un composite générique combiné avec un visiteur générique, ça fait de la généricité multiparamétrée.
- Dans l'implantation de la classe visiteur (*i.e.*, l'algorithme), l'appel récursif sur les fils revient à invoquer une méthode qui se trouve dans la structure de données avec `this` comme paramètre quand `this` est suffisamment réentrant, sinon il faut créer une nouvelle instance de sa propre classe.

– Dans l'implantation de la classe visiteur (*i.e.*, l'algorithme), l'appel à un autre algorithme sur la structure de données courante revient à invoquer une méthode qui se trouve dans la structure de données avec un **autre visiteur** (qu'il faudra éventuellement instancier s'il ne l'est pas déjà.)

10 Conclusion

Il ne faut pas oublier que les *design patterns* sont des solutions à des problèmes, et comme toutes les solutions, elles n'ont que très peu d'intérêt (voire pas du tout) quand on ne connaît pas les problèmes à qui elles servent.

N.B. Parce que des fois les modèles à objets engendrent des situations locaces, il existe des campagnes politiques sur les bonnes/mauvaises pratiques résultant de ces modèles mal maîtrisés... Vous pouvez visiter par exemple le site `http://www.antiifcampaign.com/` !

11 Questions ?

Si vous avez des remarques ou questions, vous pouvez contacter l'auteur du document par courriel : `Philippe.Wang@LiP6.fr`

Vous ne devez pas faire confiance aveuglément à ce document qui a été très peu relu.