Introduction aux Design Patterns en Java

Par Alexandre Brillant

Tutoriel d'introduction aux Design Patterns en Java. v1.01

Introduction

L'approche orientée objet tend à éclater les applications en composants plus simples et réutilisables. Cependant, cette approche peut vite devenir un piège lorsque le découpage s'effectue sans règles précises. Le concepteur finit par être saturé par la complexité du codage (effet spaghetti). J'en veux pour preuve ma propre expérience sur une application Java d'environ 40000 lignes de code. Sans architecture de base, cette application est devenue progressivement ingérable avec pour conséquence l'émergence de bugs de plus en plus difficiles à corriger (effet dominos). Pour réduire les risques de maintenance, il a fallu exploiter un niveau supplémentaire dans la conception objet : Les modèles de conception ou design patterns. Mais attention au piège de l'excès inverse, ces modèles sont courants et donc utiles mais à condition de les employer uniquement lorsque c'est vraiment nécéssaire.

1.0 - Quelques rappels sur la conception objet

Contrairement aux langages de type procéduraux comme le C ou le Pascal, la conception objet ne divise pas l'espace de données (attributs) et l'espace de traitements (méthodes). Cet espace commun s'applique à une partie du problème à gérer sous la forme d'une classe. Une classe est une représentation abstraite décrivant comment faire fonctionner des objets. Les objets sont construits à partir de cette classe lors de l'exécution par un processus d'instanciation (en Java l'opérateur new). Chacune des déclarations dans une classe peut être limitée dans sa portée (portée locale ou privée, portée de groupe ou package, portée de descendance ou protégée, portée globale ou publique).

Une classe peut être associée à d'autre classes pour faciliter la réutilisation. L'association la plus commune est l'héritage. L'héritage sert à spécialiser une classe existante (lien de généralisation/spécialisation) par la modification/ l'ajout de nouvelles méthodes et de nouvelles données. Cette spécialisation conduit à la construction de nouvelles classes (appelées aussi sous-classes). Le concept d'héritage peut être vu comme la conception d'un élément "est une sorte de". Par exemple, une Voiture peut être vue comme une sous-classe de la classe Véhicule, une Voiture hérite donc de la classe Véhicule, elle étend les caractéristiques du véhicule par des données et des méthodes supplémentaires (la vitesse de pointe, le moteur...). Lorsqu'on lit "étendre" il faut avoir conscience qu'il y a enrichissement et que l'ensemble des possibilités des enfants est toujours au moins identique ou plus grand que les ancêtres (en objet l'enfant est donc en quelque sorte plus grand que le parent).

Lorsqu'une méthode d'une classe est redéfinie par une sous-classe, on dit qu'il y a surcharge. Comme nous le verrons dans la suite cette possibilité donne une grande souplesse à l'introduction de classes génériques déléguant certains comportements aux sous-classes. Cette surcharge sert également de base au principe du polymorphisme (par héritage car il en existe plusieurs sortes). Le polymorphisme est une manière d'uniformiser les accès entre des objets différents (type) et en s'appuyant sur les accès communs (les classes ancêtres communes).

Certaines classes ne sont pas complètement construites afin d'obliger les sous-classes à effectuer une surcharge. On parle alors de classes abstraites ou d'interfaces (classes totalement abstraites). L'interface est employée en Java et dans les languages objets récents, elle est un peu plus subtile que la classe abtraite au sens que son caractère totalement abtrait évite les problèmes de collisions lors d'un héritage (qui hérite de quoi?) entre plusieurs parents.

Une interface est associée à une ou plusieurs classes d'implémentation (on peut les qualifer de concrètes). Une classe d'implémentation contient donc le code de réalisation de l'interface. Pour simplifier, l'interface est "l'intention" et l'implémentation la "réalisation".

Dans la suite de l'article, nous proposerons des exemples en Java. Ces exemples sont facilement transposables en C++ ou dans tout autre langage objet.

2.0 - Définition des design patterns

Les design patterns ou modèles de conception décrivent des organisations pratiques de classes objets. Ces organisations résultent souvent d'une conception empirique, le concepteur objet tente de faciliter la réutilisation et la maintenance du code. On peut donc concevoir un modèle d'application comme une forme d'organisation transposable à plusieurs applications. Ces systèmes peuvent apparaître complèxes aux débutants voire inutiles, il est pourtant très important d'en connaître plusieurs et de les appliquer systématiquement (dans les cas reconnus comme pouvant évoluer). L'architecte objet se construit petit à petit un "panier" de modèles.

Les design patterns ne sont pas rééllement normalisés, mais on peut les découper en trois grandes catégories :

- Les modèles de création : Ces modèles sont très courants pour déléguer à d'autres classes la construction des objets.
- Les modèles de structure : Ces modèles tendent à concevoir des agglomérations de classes avec des macro-composants.
- Les modèles de comportement : Ces modèles tentent de répartir les responsabilités entre chaque classe (l'usage est plutôt dynamique).

Si on voulait faire un parallèle avec UML, les deux premiers modèles seraient liès à des diagrammes statiques (de classes) alors que le dernier modèle est davantage lié à un diagramme dynamique (de séquence).

3.0 - Les modèles de création

On se trouve en programmation objet souvent confronté au problème d'évolution des classes. Une classe hérite d'une autre classe pour en spécialiser certains éléments. On aimerait donc qu'un objet puisse appartenir à telle ou telle classe (dans une même famille par héritage) sans avoir à chercher la classe de gestion de ces objets et la ligne de code qui effectue l'instanciation. Si on imagine un cas de création d'un objet pour une classe C donnée, réparti dans différents endroits du code, si on décide de faire évoluer la nature de C en passant par une classe descendante (une classe C' héritant de C), il faut donc reprendre l'intégralité du code de création, avec une classe chargée de la création, plus besoin et seule cette dernière est à reprendre.

3.1 - Fabrique de création

Une fabrique de création (ou factory) est une classe qui n'a pour rôle que de construire des objets. Cette classe utilise des interfaces ou des classes abstraites pour masquer l'origine des objets.

Exemple :

```
/** Interface de description d'un point */
public interface Point {
 /** Retourne l'abscisse du point */
 public int getX();
 /** Retourne l'ordonnée du point */
 public int getY();
}

/** Interface de description d'une ligne */
public interface Line {
  /** Retourne les coordonnées du mier point */
  public int getX1();
  public int getY1();
  /** Retourne les coordonnées du deuxième point */
  public int getX2();
  public int getY2();
}

/**  Fabrique  retournant des objets de types point ou ligne */
public class CanvasFactory {
  /** Retourne un Point aux coordonnées x,y */
  public Point getPoint( int x, int y ) {
   return new PointImpl( x, y );
  }
  /** Retourne une Ligne aux coordonnées x1,y1,x2,y2 */
  public Line getLine( int x1, int y1, int x2, int y2 ) {
   return new LineImpl( x1, y1, x2, y2 );
 }
}
```

Dans cet exemple, nous définissons deux interfaces Point et Line représentant deux classes Abstraites. Ces classes Point et Line designent des objets retournées par la classe CanvasFactory. Cette classe masque la véritable nature des objets. Ici nous retournons par les méthodes d'accès des objets PointImpl et LineImpl qui implémentent respectivement les interfaces Point et Line. Ainsi, l'application utilisera la classe CanvasFactory pour obtenir des éléments graphiques, lors d'une évolution l'utilisateur pourra changer facilement la nature des objets (avec d'autres classes implémentant les interfaces Point et Line...).

On distingue parfois deux formes de fabrique :

- Les fabriques abstraites comme celle que nous venons de voir reposant sur l'exploitation de classes génériques (Point et Line).
- Les fabriques concrètes masquant toutes les méthodes nécéssaires à la création et à l'initialisation de l'objet.

3.2 - Singleton

Un singleton sert à contrôler le nombre d'instances d'une classe présent à un moment donné. C'est souvent très pratique pour les classes sans état et effectuant toujours les mêmes traitements.

Un singleton est construit grâce à des méthodes de classes. Ces méthodes appartiennent à une classe et peuvent être accedées indépendemment de l'objet.

Exemple :

```
public CanvasFactory {
  /** Donnée de classe contenant l'instance courante */
  private static CanvasFactory instance = new CanvasFactory();

  /** Constructeur privé interdisant l'instanciation en dehors de cette classe */
  private CanvasFactory () {}
  /** Singleton de la classe courante */
  public static CanvasFactory getInstance() { return instance; }
  ...
}
```

Pour continuer avec l'exemple précédent, nous ajoutons à notre classe CanvasFactory un attribut de classe par la déclaration "private static" représentant l'unique instance disponible de la classe. Nous ajoutons un constructeur de portée privée "CanvasFactory()" pour interdire l'instanciation de cette classe. Enfin la méthode "public static CanvasFactory getInstance" nous retourne l'instance unique de la classe CanvasFactory.

Typiquement l'usage de la classe CanvasFactory se fera sous la forme suivante :

```
CanvasFactory cf = CanvasFactory.getInstance();
```

Le singleton limite le nombre d'instance en mémoire. Il peut être parfois perçu comme une fabrique particulière.

3.3 - Builder

Le Builder ou Monteur est une classe offrant des moyens de construction d'un objet. Par exemple, pour construire un dessin il faut ajouter des points, des lignes, des cercles.... Il ne doit pas être confondu avec la Fabrique.

Le problème d'une Fabrique de création, c'est qu'elle ne permet de définir comment un objet va être construit, certes, il est toujours possible de passer x paramètres dans la méthode de création d'une fabrique mais cela s'avère souvent très réducteurs voire délicat pour la maintenance.

Exemple :

```
/** Builder pour les objets Canvas */
public CanvasBuilder {
   /** Initialise le canvas */
   public void initCanvas( Canvas c ) { ... }
   /** Ajouter un point aux coordonnées x,y */
   public void addPoint( int x, y ) { ... }
   /** Ajouter une ligne aux coordonnées x1,y1,x2,y2 */
   public void addLine( int x1, int y1, int x2, int y2 ) { ... }
}
```

Cet exemple illustre l'usage du Monteur pour construire un espace de dessin (canvas) à l'aide de primitive. Tout d'abord, on appelera la méthode "initCanvas" qui se chargera de désigner une Fabrique pour un Canvas et effacera

l'espace de dessin. Ce canvas pourra être enrichi par les méthodes "addPoint" ou "addLine". L'utilisateur ne manipule donc plus un objet directement, il applique un ensemble d'opérations très naturellement. Le concepteur du Monteur peut à n'importe quel moment changer la nature de l'objet voire l'interprétation des méthodes disponibles.

Exemple à l'usage :

```
//Instanciation du Builder
CanvasBuilder cb = new CanvasBuilder();
Canvas c = new CanvasImpl();
//Montage du Canvas
cb.initCanvas( c );
cv.addPoint( 5, 5 );
cv.addLine(  0, 0 );
cv.addLine( 10,0 );
cv.addLine( 10, 10);
...
```

On peut exploiter les Monteurs en complément d'une Fabrique. Un Fabrique utilise alors le Builder pour "monter" l'objet retourné.

4.0 - Les modèles de structure

Ces modèles de conception tentent de composer des classes pour bâtir de nouvelles structures. Ces structures servent avant tout à ne pas gérer différemment des groupes d'objets et des objets uniques. Tout le monde en utilisant un logiciel de dessin vectoriel est amené à grouper des objets. Les objets ainsi conçus forment un nouvel objet que l'on peut déplacer, et manipuler sans avoir à répéter ces opérations sur chaque objet qui le compose. On obtient donc une structure plus large mais toujours facilement manipulable.

4.1 - Adapter

L'Adapteur ou Adapter est un moyen commode de faire fonctionner un objet avec une interface qu'il ne possède pas. L'idée est de concevoir une classe supplémentaire qui se charge d'implémenter la bonne interface (L'Adapteur) et d'appeler les méthodes correspondantes dans l'objet à utiliser (L'adapté).

Exemple :

```
/** Interface de représentation d'un cercle */
public interface Circle {
  /** Retourne l'abscisse du centre du cercle */
  public int getX();
  /** Retourne l'ordonnée du centre du cercle */
  public int getY();
  /** Retourne le rayon du cercle */
  public int getR();
}

/** Classe implémentant l'interface Circle */
public class CircleImpl implements Circle {
  ...
}

/** Adapteur pour transformer le circle en un point */
public class CircleImplPointAdapter implements Point {
  private Circle c;
  public CircleImplPointAdapter( Circle c ) {
   this.c = c;
  }
  public int getX() { return c.getX(); }
  public int getY() { return c.getY(); }
}
```

Cet exemple tente de convertir un objet implémentant l'interface Circle en un objet de type Point. Une classe CircleImplPointAdapterva réaliser cette liaison se chargeant d'implémenter l'interface Point et d'appeler les méthodes du Cercle analogue à celle du Point, en l'occurrence getX et getY.

L'adapteur sert donc à lier des classes indépendantes n'ayant pas les bonnes interfaces. Cette exploitation est réservée au cas particulier où le changement d'une classe mère serait impossible voire trop complèxe. Attention cependant à ne pas accroître de manière inconsidérer le nombre de classe par l'introduction d'adapteur à tous les niveaux. L'adapteur reste un moyen pour effectuer des "raccords".

4.2 - Proxy

Assez proche de l'Adapteur, le Proxy cherche à ajouter un niveau de redirection entre l'appel d'une méthode d'un objet et l'action associée. A cet effet, on construit une nouvelle classe implémentant l'interface de l'objet à manipuler et déportant toutes les actions sur un autre objet implémentant la même interface. Ce type de structure vise à pouvoir changer l'objet cible sans changer l'objet source (manipulé).

Exemple :

```
/** Interface représentant un espace de dessin */
public interface Canvas {
  /** Ajout d'un objet graphique */
  public void addGraphicElement( GraphicElement ge );
}

/**  Proxy pour l'espace de dessin */
public class CanvasProxy implements Canvas {
  private Canvas c;
  public CanvasProxy( Canvas c ) {
   this.c = c;
  }
  public void addGraphicElement( GraphicElement ge ) {
   c.addGraphicElement( ge );
  }
}
```

Dans cet exemple, une interface Canvas est définie avec une méthode pour ajouter des objets graphiques. Le proxy CanvasProxy est une classe qui se charge d'implémenter l'interface Canvas et d'appeler les méthodes sur l'objet Canvas passé en paramètre de constructeur. On pourrait comme exemple pratique, imaginer que le canvas puisse aussi exister dans sa forme contraire pour faire office de gomme. En passant dans le mode "gomme", tout objet ajouté effacerait une partie du fond comme une empreinte, pour cela, il sufferait dans la classe CanvasProxy de modifier ou changer d'objet Canvas. L'exploitant manipulerait dans tous les cas le même objet.

Les Proxy sont très utilisés pour la gestion d'objets distribués (protocole RMI en Java par exemple). L'idée étant de construire des Proxy capable de communiquer avec des objets distants (usage de la sérialisation en Java) sans que l'exploitant fasse de différences entre un accès locale ou un accès distant.

4.3 - Composite

Le modèle Composite cherche à éliminer toute différence entre un groupe d'objets et un objet. Il s'agit d'une démarche récurrente valable pour tous les problèmes qui font émerger de nouvelles structure par association. L'exemple le plus simple étant celui du groupe d'objets dans un logiciel de dessin comme nous l'avons vu précédemment. La manière la plus simple pour gérer ces modèles est l'exploitation d'une interface unique pour les éléments simples et l'élément "composé".

Exemple :

```
/** Implémentation de l'interface Canvas composite */
public class CanvasImpl implements GraphicElement, Canvas {
/** Ajout d'un élément graphique ou d'un Canvas ! */
public void addGraphicElement( GraphicElement ge ) { ... }
```

Notre exemple se base sur l'exemple précédent, ici nous adoptons deux interfaces pour la classe CanvasImpl :

- L'interface Canvas contenant la méthode "addGraphicElement" pour construire notre espace de dessin
- L'interface GraphicElement pour associer l'espace de dessin à un nouvel élément graphique. Il est donc possible d'ajouter un objet de la classe CanvasImpl dans un objet de la classe CanvasImpl.

L'utilisation d'une interface commune limite les besoins de tests de chaque objet, ici tous les objets sont vus de la même manière. On rencontre cette exploitation dans les arbres DOM avec les documents XML. Un document XML est lui-même une forme de noeud ce qui permet de créer du chainage de documents très facilement.

5.0 - Les modèles de comportement

Le modèle de comportement simplifie l'organisation d'exécution des objets. Typiquement, une fonction est composée d'un ensemble d'actions qui parfois appartiennent à des domaines différents de la classe d'implémentation. On aimerait donc pouvoir "déléguer" certains traitement à d'autres classes. D'une manière générale, un modèle de de comportement permet de réduire la complexité de gestion d'un objet ou d'un ensemble d'objet.

5.1 - Iterator

L'Itérateur ou Iterator est le plus comun des modèles de comportement. L'idée étant de limiter la vision d'une collection par un utilisateur. Typiquement une collection contient un ensemble d'objets stocké par différentes méthodes (un tableau, un vecteur...), l'exploitant qui accède au contenu de la collection ne souhaite pas être concerné par cette manière de gérer les objets. La collection offre donc un point d'accès unique sous la forme d'une interface Iterator.

Exemple :

```
/** Classe de gestion d'un espace de dessin */
public class CanvasImpl implements GraphicsElement, Canvas {
  // Tableau pour stoker les éléments de la collection
  private GraphicsElement[] ge;
  ...
  /** Retourne un itérateur pour accéder aux objets de la collection */
  public Iterator getIterator() { return ArrayIterator( ge ); }
}

/** Interface pour toutes les collections d'objets de GraphicElement */
public interface Iterator {
   public GraphicElement getNextElement();
}

/** Iterateur parcourant un tableau pour retourner des objets de type GraphicElement */
public class ArrayIterator implements Iterator {
  private GraphicElement[] ge;
  private int nge;
  /** Constructeur avec un tableau de données à parcourir */
  public ArrayIterator( GraphicElement[] ge ) {
   this.ge = ge;
   nge = 0;
  }
  /** Retourne chaque élément de la collection ou null */
  public GraphicElement getNextElement() {
   if ( nge >= ge.length ) return null;
   return ge[ nge++ ];
  }
}
```

Cet exemple comprend un itérateur pour la collection CanvasImpl qui contient des objets de type GraphicElement. Ces objets sont stockés dans un tableau, un itérateur ArrayIterator parcourt le tableau et offre chaque élément contenu dans la collection par la méthode getNextElement. Si plus tard, on désire changer de méthode de stockage des objets pour des raisons de performances ou de coûts mémoire, il suffira de réaliser une nouvelle classe implémentant l'interface Iterator.

5.2 - Template

Un Template ou Patron de méthode est très utilisé dans la conception objet. Ce modèle utilise simplement les mécanismes de surcharge pour déléguer à d'autres classes certaines parties d'un algorithme. L'idée ressemble un peu au modèle Builder sauf qu'ici, il sert à construire un traitement et non un objet.

Exemple :

```
/** Template d'extraction d'éléments d'une collection selon différents critères */
public abstract class GraphicElementSorter {
   /** Iterateur d'une collection d'objet GraphicElement */
   public GraphicElementSorter( Iterator t ) {
    ...
   }

  /** Retourne l'objet le plus grand de la collection passé dans le constructeur */
  public GraphicElement getMaxGraphicElement() {
    GraphicElement refGe = t.getNextElement();
    for ( GraphicElement ge = t.getNextElement(); ge != null; ) {
      if ( isGreaterThan( ge, refGe ) ) then refGe = ge;
    }
   return refGe;
  }
  /** Méthode à surcharger retournant l'objet graphique le plus grand entre ge1 et ge2 */
  protected abstract GraphicElement isGreaterThan( GraphicElement ge1, GraphicElement ge2 );
}

/** Cette classe désigne la méthode pour comparer la taille de deux objets */
public class SimpleGraphicElementSorter {
   /** Cette méthode compare deux objets ayant chacun une méthode size retournant la surface */
   public GraphicElement isGreaterThan( GraphicElement ge1, GraphicElement ge2 ) {
    if ( ge1.size() > ge2.size() return ge1; else return ge2;
   }
}
```

Le Template fonctionne par couche. Il permet d'anticiper le champs d'application des algorithmes. Le Template peut exister aussi par l'usage de délégué. Par exemple, on fournit un objet capable de comparer deux Elements à la classe effectuant le traitement principal ce qui évite d'effectuer de l'héritage pour modifier un simple traitement.

6.0 - Pour conclure

Les Design Patterns représentent un espace très riche de composition ou de simplification de votre développement objet. Nous en avons étudié quelques uns ici, mais il en existe beaucoup d'autres et vous serez également amenés à en trouver de nouveaux. Attention à ne pas se laisser "griser" par ces patterns, trop de patterns est un "anti-pattern", une belle architecture est toujours un équilibre entre le possible et le nécéssaire. L'objectif étant de maximiser le "nécéssaire" et donc de faire de bonnes prévisions. Une mauvaise prévision sera d'autant plus pénalisante que votre projet doit avançer à une certaine cadence. Bon design !

7.0 - L'auteur

Pour me présenter rapidement, je suis formateur Java & XML. Je travaille en région parisienne mais je peux me déplaçer dans toute la france si l'hôtel et le déplacement sont pris en compte dans la prestation. Mon CV est disponible ici : **http://www.abrillant.com/cv.html**.

J'ai à mon actif quelques réalisations :

- EditiX : Un éditeur complet XML et XSLT **http://www.editix.com**
- XFlows : Un gestionnaire de flux XML et XSLT **http://www.xflows.com**
- UniMailer : Un gestionnaire de mails **http://www.unimailer.com**
- SwingAll : Un ensemble de composants Swing (Environnement dockable...) **http://www.swingall.com**
- JXMLPad : Un composant Swing d'édition XML : **http://www.jxmlpad.com**
- JFormula : API de calcul : **http://www.japisoft.com/formula**
- JAPISoft : Un ensemble d'API Java (Parser, Implémentation XPath, Librairie de calcul, Plugin Eclipse/Netbeans...) **http://www.japisoft.com**

Vous trouverez davantage d'informations (tarifs, types de cours...) sur mon site personnel : **http://www.abrillant.com**

8.0 - Téléchargements

- **Tutoriel au format HTML** - (**mirroir**)
- **Tutoriel au format PDF** - (**mirroir**)