**Pattern de construction : Abstract Factory**

abstract class Forme {  
 abstract void affiche( );  
}  
  
class Cercle extends Forme {  
 public void affiche( ) {  
 System.**out**.println( "Logo circulaire" );  
 }  
}  
  
class Rectangle extends Forme {  
 public void affiche( ) {  
 System.**out**.println( "Logo rectangle" );  
 }  
}  
  
abstract class FormeFactory {  
 public abstract Forme getForme( );  
}  
  
public class RandomFormeFactory extends FormeFactory{  
 public Forme getForme( ) {  
 double x = Math.random( );  
 if ( x < 0.5 ) return new Cercle( );  
 else return new Rectangle( );  
 }  
}  
  
  
class OtherFormeFactory extends FormeFactory {  
 public static boolean **indic** = false;  
  
 public Forme getForme( ) {  
 if ( **indic** ) {  
 **indic** = false;  
 return new Cercle( );  
 } else {  
 **indic** = true;  
 return new Rectangle( );  
 }  
 }  
}  
  
public class TestFabrique2 {  
 public static void main( String args[] ) {  
 FormeFactory fab;  
 fab = new RandomFormeFactory ( );  
 System.**out**.println( "--- avec Fabrique au hasard" );  
 for ( int i = 0 ; i < 4 ; i++ ) {  
 Logo l = fab.getForme( );  
 l.affiche( );  
 }  
 fab = new OtherFormeFactory ( );  
 System.**out**.println( "--- avec Fabrique alternee" );  
 for ( int i = 0 ; i < 4 ; i++ ) {  
 Logo l = fab.getForme( );  
 l.affiche( );  
 }  
 }  
}

On voit que le pattern Abstract Factory nous a permis de dissocier la création des objets de leur utilisation. Le client (ici main) peut utiliser des boutons et des cases à cocher, sans en connaître les types concrets. Il sait simplement que ces types dérivent des types abstraits BoutonRadio et CaseCocher. Ici encore, on peut dire qu’on a encapsulé la création des objets et que le client se contente de programmer pour des interfaces, non pour des objets concrets.  
Le pattern Factory Method prévoit qu’il puisse exister plusieurs fabriques qui vont alors implémenter une interface commune.

Comme nous l’avons vu, le pattern Factory Method permet de fournir un ensemble de classes à un client qui n’a pas à connaître la classe concrète des objets instanciés (on parle souvent de « produits » dans ce cas). Il ne précise pas qui a la responsabilité du choix de cette classe.

**Pattern de structure : Composite**

Ces patterns permettent de combiner des classes ou des objets pour créer de nouvelles structures. Lorsqu’ils s’appuient sur l’héritage, la structure est définie à la compilation. Lorsqu’ils s’appuient sur la composition, on obtient une structure dynamique susceptible d’être modifiée durant l’exécution.

abstract class Composant {  
 private String nom;  
  
 public Composant( String nom ) {  
 this.nom = nom;  
 }  
  
 public void ajoute( Composant c ) {  
 }  
  
 public abstract void affiche( ); // a redefinir dans chaque classe concrete  
  
 public String getNom( ) {  
 return nom;  
 }  
}  
  
class Cercle extends Composant {  
 public Cercle( String nom ) {  
 super( nom );  
 }  
  
 public void affiche( ) {  
 System.**out**.println( "Cercle " + getNom( ) );  
 }  
}  
  
class Rectangle extends Composant {  
 public Rectangle( String nom ) {  
 super( nom );  
 }  
  
 public void affiche( ) {  
 System.**out**.println( "Rectangle " + getNom( ) );  
 }  
}  
  
class Groupe extends Composant {  
 private ArrayList< Composant > listeComposants = new ArrayList< Composant >( );  
  
 public Groupe( String nom ) {  
 super( nom );  
 }  
  
 public void ajoute( Composant c ) {  
 listeComposants.add( c );  
 }  
  
 public void affiche( ) {  
 System.**out**.println( "---- Groupe " + getNom( ) + " contenant : " );  
 for ( Composant c : listeComposants ) {  
 c.affiche( );  
 } // depuis JDK5  
 System.**out**.println( "-------- fin groupe " + getNom( ) );  
 }  
}  
  
public class TestComposite {  
 public static void main( String args[] ) {  
 Cercle c1 = new Cercle( "C1" );  
 Cercle c2 = new Cercle( "C2" );  
 Rectangle r1 = new Rectangle( "R1" );  
 c1.affiche( );  
 Groupe ga = new Groupe( "GA" );  
 ga.ajoute( c1 );  
 ga.ajoute( r1 );  
 ga.affiche( );  
 Groupe gb = new Groupe( "GB" );  
 gb.ajoute( ga );  
 gb.ajoute( c2 );  
 gb.affiche( );  
 }  
}

Le design pattern composite permet de gérer un ensemble d'objets en tant qu'un seul et même objet, autrement dit un objet composé de plusieurs autres.

Il permet d'additionner les propriétés des différents objets (par exemple un prix) pour en composer un seul et même. Il simplifie les compositions car on peut ajouter ou supprimer des éléments d'un objet composé grâce à des méthodes, sans avoir à modifier le code de leurs classes.

À ce niveau, on constate que les formes dérivées de Composant vont disposer de cette méthode ajoute au même titre que les groupes, alors qu’elle n’a aucun sens pour elles. C’est là en quelque sorte le prix à payer pour obtenir la récursivité souhaitée

Chaque groupe maintient la référence des objets qu’il contient. Réciproquement, on pourrait envisager que chaque composant comporte une référence à l’éventuel groupe dans lequel il est contenu. Cela pourrait faciliter le travail de certaines méthodes

**Pattern de structure : Decorator**

abstract class Decorator {  
 abstract void affiche( );  
}  
  
class Ballon extends Decorator {  
 private int x, y;  
  
 public Ballon( int x, int y ) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public void affiche( ) {  
 System.**out**.println( "coordonnees = " + x + " " + y );  
 }  
}  
  
class Coloration extends Decorator {  
 private Decorator d;  
 private int c;  
  
 Coloration( Affichable d, int c ) {  
 this.d = d;  
 this.c = c;  
 }  
  
 public void affiche( ) {  
 p.affiche( );  
 System.**out**.println( "+++ couleur " + c );  
 }  
}  
  
class Forme extends Decorator {  
 private Decorator d;  
 private char f;  
  
 Forme( Decorator d, char f ) {  
 this.d = d;  
 this.f = f;  
 }  
  
 public void affiche( ) {  
 p.affiche( );  
 System.**out**.println( "+++ forme " + f );  
 }  
}  
  
public class TestDecorateur {  
 public static void main( String args[] ) {  
 Decorator dc = new Coloration( new Ballon( 6, 5 ), 10 );  
 dc.affiche( );  
 Decorator df = new Forme( new Ballon( 1, 4 ), '\*' );  
 df.affiche( );  
 Decorator dcf = new Forme( pc, '+' );  
 dcf.affiche( );  
 }  
}

Ce pattern permet d’ajouter dynamiquement des fonctionnalités à une classe existante, sans avoir à la modifier. Il est parfois appelé « enveloppeur » (wrapper).

Sur une application qui est sujette à beaucoup d'évolution, la tâche est grandement simplifiée par ce design pattern.

Le pattern Decorator va nous offrir une solution intéressante. Il consiste à créer des classes dites « décorateurs », chaque classe ajoutant une des fonctionnalités à une classe de type Point, conduisant à ce que l’on nomme une « classe décorée ». La classe décorée pourra, à son tour, être décorée, ce qui permettra de combiner l’ajout des fonctionnalités.

Dans ce modèle, on voit plusieurs choses :

* On peut créer plusieurs composants concrets.
* On peut étendre les fonctionnalités de nos composants grâce au décorateur.
* Le décorateur est une classe abstraite qui hérite de Composant et qui a un attribut Composant.
* Chaque Decorateur concret redéfinie les méthodes de Décorateur.

L’idée est que l’objet décorateur dispose de la référence (p) à l’objet à décorer. Elle sera généralement fournie à la construction du décorateur.

L’ajout d’un nouveau décorateur peut se faire, sans remettre en cause, ni les objets à décorer, ni les décorateurs existants. On dit que ce pattern applique le principe « ouvert - fermé » dans lequel les classes sont fermées à la modification, mais ouvertes à l’extension.

**Pattern de structure : Adapter (object : fondé sur la composition)**

class Point {  
  
 private int x, y;  
 public Point( int x, int y ) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public void montre( ) {  
 System.**out**.println( "Coordonnees = " + x + " " + y );  
 }  
  
 public void deltaX( int dx ) {  
 x += dx;  
 }  
  
 public void deltaY( int dy ) {  
 y += dy;  
 }  
  
}  
  
interface ***Pixel*** {  
 void affiche( );  
  
 void deplace( int dx, int dy );  
}  
  
class PointAdapter implements ***Pixel*** // on pourrait aussi extends Pixel si classe  
{  
 private Point px;  
  
 public PointAdapter( Point px ) {  
 this.px = px;  
 }  
  
 public void affiche( ) {  
 px.montre( );  
 }  
  
 public void deplace( int dx, int dy ) {  
 px.deltaX( dx );  
 px.deltaY( dy );  
 }  
}  
  
public class Adaptateur1 {  
 public static void main( String args[] ) { // code existant utilisant déja des implementations de Pixel  
 // .....  
 Point pxa = new Point( 2, 5 );  
 PointAdapter pa = new PointAdapter( pxa );  
 // il peut maintenant utiliser des objets de type Point, encapsules  
 // dans des objets de type PointAdapter, comme des implementations de Pixel   
 pa.affiche( );  
 pa.deplace( 2, 1 );  
 pa.affiche( );  
 }  
}

La démarche consiste à englober un objet de type Point dans un objet d’un nouveau type PointAdapter, implémentant l’interface Pixel, en y encapsulant un objet du type Point qu’on fournira à la construction.

On voit que les appels de affiche ou de deplace sur un objet de type PointAdapter sont en quelque sorte convertis en des appels de montre, deltaX ou deltaY sur l’objet englobé de type Point.

**Pattern de structure : Adapter (class : fondé sur l’héritage)**

class Point {  
  
 private int x, y;  
 public Point( int x, int y ) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public void montre( ) {  
 System.**out**.println( "Coordonnees = " + x + " " + y );  
 }  
  
 public void deltaX( int dx ) {  
 x += dx;  
 }  
  
 public void deltaY( int dy ) {  
 y += dy;  
 }  
  
 public int getX( ) {  
 return x;  
 }  
  
 public int getY( ) {  
 return y;  
 }  
  
}  
  
interface ***Pixel*** {  
 void affiche( );  
  
 void deplace( int dx, int dy );  
}  
  
class PointAdapter extends Point implements ***Pixel*** {  
 public PointAdapter( Point px ) {  
 super( px.getX( ), px.getY( ) );  
 }  
  
 public void affiche( ) {  
 super.montre( );  
 }  
  
 public void deplace( int dx, int dy ) {  
 deltaX( dx );  
 deltaY( dy );  
 }  
}  
  
public class Adaptateur2 {  
 public static void main( String args[] ) { // code existant utilisant déja des implementations de Pixel  
 // .....  
 Point px = new Point( 3, 5 );  
 PointAdapter pa = new PointAdapter( px );  
 // il peut maintenant utiliser des objets de type Point, encapsules dans  
 // des objets de type PointAdapter, comme des implementations de Pixel  
 pa.affiche( );  
 pa.deplace( 2, 1 );  
 pa.affiche( );  
 }  
}

Ce pattern permet de transformer l’interface (au sens général) d’une classe existante pour qu’elle puisse être utilisée par un client qui attend une interface différente.

On va encapsuler l’objet à adapter dans un objet d’une classe ascendante. On crée donc une classe PointAdapter qu’on va faire dériver de Point, tout en implémentant l’interface Pixel et dont le constructeur reçoit en argument un objet de type Point, représentant l’objet à adapter.

On constate cependant qu’ici, il est nécessaire que le constructeur de PointAdapter puisse appeler convenablement le constructeur Point de l’objet encapsulé. Autrement dit, il faut

•   soit que Point dispose d’un constructeur avec un argument de type Point

•   soit que Point dispose de méthodes permettant l’accès à ses coordonnées, par exemple getX et getY

•   soit que les champs x et y de Point soient déclarés protected pour que le constructeur de PointAdapter puisse y accéder.

Un adaptateur de classe souffre de quelques lacunes :

•   comme la classe adaptée (PointAdapte) dérive de la classe à adapter (Point), elle ne peut plus, en Java, dériver d’une autre classse. Ainsi, dans notre dernier exemple, Pixel ne pourait pas être une classe abstraite et encore moins une classe concrète ;

•   l’adaptateur ne peut pas adapter des classes dérivées des classes adaptées ; ainsi, ici, PointAdapter ne permettrait pas d’adapter des classes dérivées de Point ;

•   enfin, la construction de l’objet parent peut poser des problèmes d’accès, comme on l’a vu dans notre exemple.

En revanche, un adaptateur d’objet ne présente aucune des lacunes précédentes. Néanmoins, il peut rendre difficile une éventuelle modification des méthodes de l’objet adapté.

**Pattern comportemental : Strategy**

package org.thib;  
  
class Velo {  
  
 private int x, y;  
  
 private StrategieFreinage strategieFrainage; // strategie d'affichage  
  
 public Velo( int x, int y, StrategieFreinage strategieFrainage ) {  
 this.x = x;  
 this.y = y;  
 this.strategieFrainage = strategieFrainage;  
 }  
  
 void freine( ) {  
 strategieFrainage.freine( x, y );  
 }  
  
}  
  
abstract class StrategieFreinage {  
 public abstract void freine( int x, int y );  
}  
  
class FreinageCourt extends StrategieFreinage {  
 public void freine( int x, int y ) {  
 System.**out**.println( x + " " + y );  
 }  
}  
  
class FreinageLong extends StrategieFreinage {  
 public void freine( int x, int y ) {  
 System.**out**.println( "abscisse = " + x + " ordonnee = " + y );  
 }  
}  
  
public class TestStrategie {  
  
 public static void main( String args[] ) {  
  
 StrategieFreinage court = new FreinageCourt( );  
  
 Velo p1 = new Velo( 2, 9, court );  
 Velo p2 = new Velo( 4, 7, new FreinageLong( ) );  
  
 p1.freine( );  
 p2.freine( );  
  
 }  
}

Il arrive souvent qu’un problème donné puisse être résolu par différents algorithmes. Le passage d’un algorithme à un autre peut alors entraîner des modifications de toutes les classes client concernées par son utilisation.

Le pattern Strategy permet d’encapsuler chacun des algorithmes dans une classe, en permettant au client d’en ignorer les détails internes et de n’en connaître que l’interface.

Le pattern Strategy va nous permettre de découpler la classe Point de l’algorithme d’affichage.

La problématique de ce pattern est plutôt simple : comment faire pour réaliser différentes opérations avec un seul et même objet ? Vous me direz, c’est simple on fait une classe avec toutes les opérations ! Mais non car procéder comme ceci violerait un principe SOLID : le Single Responsibility principle.

Ce pattern permet :

-Une meilleure lisibilité du code.

-D’éviter de violer un principe SOLID.

-Mais avant tout de définir plusieurs algorithmes interchangeables dynamiquement.

Son seul inconvénient ? Il nécessite d’ajouter une classe.

Vous admettrez que cela fait peu d’inconvénients. Malgré cela, ça ne veut pas dire que vous pouvez l’utiliser à tout va, c’est avant tout un pattern destiné à un usage très précis : réaliser différentes opérations avec un seul et même objet.

Mais il peut également servir lorsque l’on est en présence de plusieurs classes ayant la même interface et ne différant que par leur comportement : dans ce cas, il suffit de conserver une seule classe pour la partie commune et d’encapsuler chacune des variantes dans une stratégie. Dans tous les cas, outre le fait que les algorithmes concernés peuvent être cachés au client, le pattern permet également à ce dernier d’éviter des instructions conditionnelles de sélection.

**Pattern comportemental : Template Method**

public abstract class Forme {  
  
 protected int x, y; // pour eviter pb acces dans classes derivees  
  
 public void affiche( ) {  
 afficheNature( );  
 System.**out**.println( "-- Coordonnees = " + x + " " + y );  
 afficheAutresInfos( );  
 }  
  
 abstract public void afficheNature( ); // a redefinir obligatoirement  
  
 public void afficheAutresInfos( ) {  
 } // version par defaut si pas redefinie  
}  
  
class Carre extends Forme {  
 public Carre( int x, int y ) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public void afficheNature( ) {  
 System.**out**.println( "Je suis un Carre" );  
 }  
 // ici, on ne redefinit pas afficheAutresInfos  
}  
  
class Cercle extends Forme {  
 private double r;  
  
 public Cercle( int x, int y, double r ) {  
 this.x = x;  
 this.y = y;  
 this.r = r;  
 }  
  
 public void afficheNature( ) {  
 System.**out**.println( "Je suis un Cercle" );  
 }  
  
 public void afficheAutresInfos( ) {  
 System.**out**.println( "-- Rayon = " + r );  
 }  
}  
  
public class TestTemplateMethod {  
  
 public static void main( String args[] ) {  
  
 Carre carre = new Carre( 2, 5 );  
 carre.affiche( );  
  
 Cercle cercle = new Cercle( 3, 8, 4.5 );  
 cercle.affiche( );  
 }  
}

Ce pattern s’utilise lorsqu’un algorithme applicable à des objets d’un ensemble de classes est constitué d’un squelette bien défini, dans lequel certaines parties peuvent être dépendantes de la classe concernée. Il est alors possible de placer tout ce qui est fixe dans une classe abstraite, dont dériveront les classes concernées.

Notez qu’ici, dans la classe abstraite Forme, nous avons fourni une version "par défaut" de AfficheAutresInfos, ne faisant rien. Cela nous a permis d’éviter d’avoir à la redéfinir dans Point, qui n’avait rien de spécifique à afficher. En revanche, la méthode afficheNature a été déclarée abstraite pour imposer sa redéfinition par les classes dérivées.

**Pattern comportemental : Observer**

class Point extends Observable {  
 String nom;  
 private int x, y;  
  
 public Point( String nom, int x, int y ) {  
 this.nom = nom;  
 this.x = x;  
 this.y = y;  
 }  
  
 public void deplace( int dx, int dy ) {  
 x += dx;  
 y += dy;  
 if ( dx != 0 ) {  
 setChanged( );  
 }  
 notifyObservers( );  
 }  
  
 public int getX( ) {  
 return x;  
 }  
  
 public String getNom( ) {  
 return nom;  
 }  
}  
  
class PointObserver implements ***Observer*** {  
 public void update( Observable obs, Object o ) { // ici o est null  
 if ( obs instanceof Point )  
 System.**out**.println( "Nouvelle abscisse " + ( ( Point ) obs ).getX( ) + " dans le point de nom " + ( ( Point ) obs ).getNom( ) );  
 }  
  
}  
  
public class TestObservateur {  
 public static void main( String args[] ) {

PointObserver obs = new PointObserver ( );  
  
 Point p1 = new Point( "A", 3, 5 );  
 Point p2 = new Point( "B", 2, 2 );  
  
 p1.deplace( 3, 9 ); // ici, on n'est pas prevenu  
 p1.addObserver( obs ); // obs observe maintenant p1  
 p1.deplace( 2, 8 ); // ici, on est prevenu pour p1  
  
 p2.deplace( 3, 2 ); // mais pas pour p2  
 p2.addObserver( obs ); // obs observe maintenant p1 et p2  
 p1.deplace( 1, 8 ); // ici, on est prevenu pour p1  
 p2.deplace( 2, 2 ); // et pour p2  
 p2.deplace( 0, 4 ); // ici, l'abscissse n'a pas change  
 }  
}

Il arrive fréquemment qu’on ait besoin qu’un ou plusieurs objets soient « prévenus » du changement d’état d’un autre objet. C’est ce qui se passe dans la gestion d’un événement (clic souris, frappe clavier...) dans un environnement graphique.

Le pattern Observer permet de mettre en place une telle dépendance en distinguant :

•   les objets observateurs ;

•   les objets observables (plus précisément, susceptibles d’être observés), capables d’enregistrer, à leur demande, des références d’objets observateurs et de les prévenir d’un changement d’état.

Contrairement aux précédents, ce pattern peut être mis en œuvre à l’aide d’outils fournis par Java.

Un même observateur pourra observer plusieurs objets, comme dans notre exemple où obs observait les objets p1 et p2, tous deux de type Point. Ces objets observés pourront éventuellement être de classes différentes. Réciproquement, un même objet peut être observé par plusieurs observateurs, éventuellement de classes différentes, pour peu que ces dernières dérivent de Observer.

On dit que ce pattern introduit un couplage faible entre l’observable et l’observé. L’observable ne connaît pas les classes concrètes de ses observateurs ; il sait simplement qu’elles implémentent l’interface Observer.

Ce pattern souffre d’une limitation importante : la classe des objets observés doit dériver de Observable, ce qui lui exclut de dériver d’autre chose. Avec une implémentation de son crû, on pourra prévoir une interface là où les outils Java imposaient une classe abstraite.

**Pattern comportemental : Observer (implémentation custom)**

class Pixel {  
 private int x, y;  
  
 public Pixel( int x, int y ) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public void deplace( int dx, int dy ) {  
 x += dx;  
 y += dy;  
 }  
  
 public int getX( ) {  
 return x;  
 }  
}  
  
class Point extends Pixel implements ***AbscisseObservable*** {  
 private ArrayList< ObservateurDAbscisses > listeObs;  
 private String nom;  
  
 public Point( String nom, int x, int y ) {  
 super( x, y );  
 this.nom = nom;  
 listeObs = new ArrayList< ObservateurDAbscisses >( );  
 }  
  
 public void deplace( int dx, int dy ) {  
 super.deplace( dx, dy );  
 if ( dx != 0 ) prevenir( );  
 }  
  
 public String getNom( ) {  
 return nom;  
 }  
  
 public void enregistre( ObservateurDAbscisses obs ) {  
 listeObs.add( obs );  
 }  
  
 public void prevenir( ) {  
 for ( ObservateurDAbscisses obs : listeObs ) {  
 obs.actualise( this );  
 }  
 }  
}  
  
interface ***AbscisseObservable*** {  
  
 public void enregistre( ObservateurDAbscisses obs );  
 public void prevenir( );  
  
}  
  
abstract class ObservateurDAbscisses {  
 public abstract void actualise( ***AbscisseObservable*** p );  
  
}  
  
class ObservateurDePoints extends ObservateurDAbscisses {  
 public void actualise( ***AbscisseObservable*** obj ) // impossibe d’utiliser Point ici  
 {  
 if ( obj instanceof Point )  
 System.**out**.println( "Nouvelle abscisse " + ( ( Point ) obj ).getX( )  
 + " du point de nom " + ( ( Point ) obj ).getNom( ) );  
 }  
  
}  
  
public class TestObservateurInt {  
 public static void main( String args[] ) {  
 ObservateurDePoints of = new ObservateurDePoints( );  
   
 Point p1 = new Point( "A", 3, 5 );  
 Point p2 = new Point( "B", 2, 2 );  
   
 p1.deplace( 3, 9 ); // ici, on n'est pas prevenu  
 p1.enregistre( of ); // of observe maintenant p1  
 p1.deplace( 2, 8 ); // ici, on est prevenu pour p1  
   
 p2.deplace( 3, 2 ); // mais pas pour p2  
 p2.enregistre( of ); // of observe maintenant p1 et p2  
 p1.deplace( 1, 8 ); // ici, on est prevenu pour p1  
 p2.deplace( 2, 2 ); // et pour p2  
 }  
}

Ici, donc, pour appliquer notre pattern Observer, nous sommes obligés de faire implémenter à Point une interface que nous nommerons AbscisseObservable. Par ailleurs, la classe concrète d’observateurs, nommée ObservateurDePoints dérivera d’une classe abstraite ObservateurDAbscisses.

D’une manière générale, pour implémenter ce pattern Observer, on utilisera :

•   une classe abstraite (ou une interface) jouant le rôle de Observable, dont dériveront (qu’implémenteront) les classes concrètes des objets observés ; on y trouvera une méthode d’enregistrement d’un observateur (jouant le rôle de addObservers) ainsi que la méthode de notification d’un observateur (jouant le rôle de notifyObservers). Nous les nommerons ici respectivement enregistre et prevenir ;

•   une classe abstraite (ou une interface) jouant le rôle de Observable, dont dériveront (qu’implémenteront) les classes concrètes d’observateurs ; on y trouvera une méthode nommée ici actualise, jouant le rôle de update.