Alan SHALLOWAY
James TROTT

Design patterns par la pratique

ISBN : 2-212-11139-8

EYROLLES

7

Le pattern Adaptateur

Sommaire

Nous allons poursuivre notre étude des design patterns par la présentation du pattern Adaptateur qui est utilisé en combinaison avec de nombreux autres patterns.

Ce chapitre abordera les thèmes suivants :

- *description du pattern Adaptateur, de son utilisation et de son implémentation ;*
- *caractéristiques principales du pattern Adaptateur ;*
- *illustration du polymorphisme par le pattern Adaptateur ;*
- *utilisation du langage UML à différents niveaux de détail ;*
- *remarques d'ordre pratique sur le pattern Adaptateur, et notamment comparaison avec le pattern Façade ;*
- *rôle du pattern Adaptateur dans notre exemple de CFAO.*

L'exemple figurant dans le corps du chapitre est en Java. Vous trouverez son équivalent en C++ à la fin de ce chapitre.

Présentation du pattern Adaptateur

La bande des quatre indique que le pattern Adaptateur :

« Convertit l'interface d'une classe en une autre conforme à l'attente du client. L'Adaptateur permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles. »[1]

Concrètement, ce pattern nous permet de créer une interface pour un objet effectuant les actions dont nous avons besoin, mais n'utilisant pas l'interface qui nous convient.

Le pattern Adaptateur en contexte

L'objet client ignore les détails

L'exemple suivant vous permettra de mieux comprendre le rôle du pattern Adaptateur. Supposons que vous deviez respecter les spécifications suivantes :

- créer des classes pour des points, des lignes et des carrés ayant un comportement « afficher » ;
- les objets clients n'ont pas besoin de savoir s'ils ont un point, une ligne ou un carré, mais uniquement qu'ils possèdent l'une de ces formes.

En d'autres termes, il s'agit d'inclure ces formes dans un concept de niveau supérieur que nous appellerons « forme affichable ».

Vous avez probablement déjà rencontré des situations similaires, par exemple :

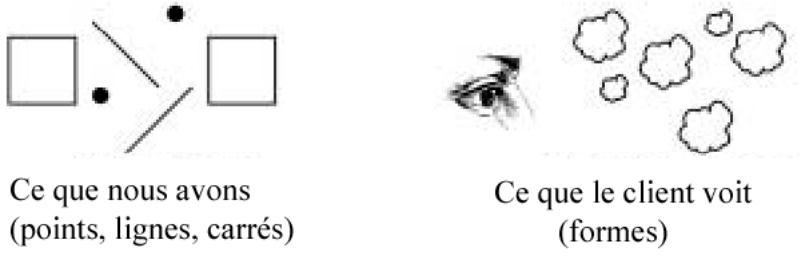
- vous aviez besoin d'utiliser une sous-routine ou une méthode écrite par quelqu'un d'autre parce qu'elle contenait des fonctions qui vous intéressaient ;
- vous ne pouviez pas intégrer une routine directement dans votre programme ;
- l'interface ou la méthode d'appel du code ne correspondait pas exactement à celle dont les objets associés ont besoin.

Pour en revenir à notre exemple, bien que le système soit conçu pour avoir des points, des lignes et des carrés, les objets clients doivent penser qu'ils ont simplement des formes pour les raisons suivantes :

- cela leur permet de traiter tous ces objets de la même façon, sans s'arrêter sur leurs différences ;
- cela vous permettra d'ajouter ultérieurement d'autres types de formes sans modifier les clients (voir la figure 7-1).

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass.:Addison-Wesley, 1995, page 139 (page 163 dans l'édition française).

Figure 7-1
Les objets ne sont que des formes



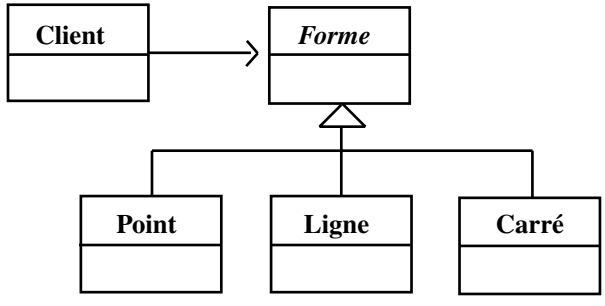
Polymorphisme et classes dérivées

Pour ce type de conception, nous utiliserons le polymorphisme, c'est-à-dire que nous aurons plusieurs objets dans le système, mais les clients correspondants agiront envers eux d'une seule et même façon.

Dans le cas présent, l'objet client indique simplement au point, à la ligne ou au carré qu'il doit effectuer une action, par exemple s'afficher ou annuler son affichage. Chaque point, ligne ou carré doit ensuite savoir comment se comporter en fonction de son type.

Pour cela, nous devons créer une classe `Forme` et en dériver les classes des points, des lignes et des carrés, comme illustré dans la figure 7-2.

Figure 7-2
Classes dérivées de la classe `Forme` *:* `Point`*,* `Ligne` *et* `Carre`



Tous les diagrammes de classes de ce livre utilisent la notation UML (langage de modélisation unifié). Reportez-vous au chapitre 2 pour plus d'informations sur ce sujet.

Définition de l'interface et implémentation des classes dérivées

Tout d'abord, nous devons spécifier le comportement des formes. Pour cela, nous définissons une interface dans la classe `Forme`, puis nous implémentons ce comportement dans chaque classe dérivée.

La classe `Forme` aura les comportements suivants :

- définir un emplacement pour la forme ;
- obtenir un emplacement pour la forme ;
- afficher une forme ;
- remplir une forme ;
- définir la couleur d'une forme ;
- annuler l'affichage d'une forme.

Ces méthodes sont illustrées dans la figure 7-3 suivante :

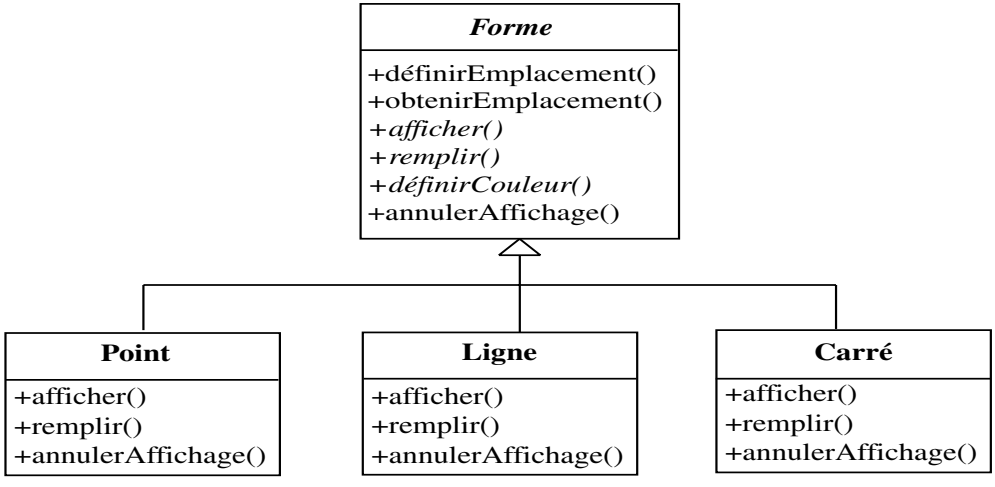


Figure 7-3
Classes `Point`, `Lignes` *et* `Carre` *avec leurs méthodes*

Supposons maintenant que vous deviez implémenter un nouveau type de forme tel que le cercle (n'oubliez pas que les spécifications changent sans cesse !). Vous créez alors une classe `Cercle` qui implémente la forme correspondante et qui est dérivée de la classe `Forme` de façon à bénéficier du polymorphisme.

Vous devez ensuite écrire les méthodes `afficher`, `remplir` et `annulerAffichage` de la classe `Cercle`. Supposons maintenant qu'il existe déjà une classe CercleXX et que vous décidiez d'utiliser les méthodes qu'elle contient :

- `afficherCercle`
- `remplirCercle`

- `annulerAffichageCercle`

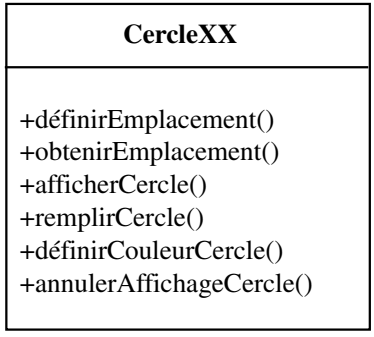


Figure 7-4
Classe `CercleXX` *existante*

Dans le cas présent, vous ne pouvez pas utiliser `CercleXX` directement si vous souhaitez conserver le polymorphisme de `Forme` pour les deux raisons suivantes :

- les noms des méthodes et les listes de paramètres diffèrent de ceux de la classe `Forme` ;
- la nouvelle classe doit être dérivée de `Forme`.

À ce stade, vous vous demandez donc probablement comment utiliser le code existant sans modifier toutes les méthodes de `CercleXX`, ce qui risquerait d'entraîner des effets secondaires imprévus. La solution est simple : il suffit de créer une nouvelle classe qui sera dérivée de `Forme` et implémentera son interface mais en évitant de réécrire l'implémentation du cercle fournie dans `CercleXX`, comme illustré dans la figure 7-5.

Pour résumer :

- la classe `Cercle` est dérivée de `Forme` ;
- elle contient `CercleXX` ;
- elle passe les requêtes effectuées à l'objet `Cercle` *via* l'objet `CercleXX`.

Le losange situé sur la droite du schéma, entre `Cercle` et `CercleXX`, indique que cette première classe contient la seconde. Lorsqu'un objet `Cercle` est instancié, il doit également instancier un objet `CercleXX` correspondant. Si tout ce que l'objet `Cercle` fait est systématiquement transmis à l'objet `CercleXX` et que ce dernier possède toutes les fonctionnalités nécessaires à l'objet `Cercle`, l'objet `CercleXX` peut effectuer les actions demandées.

Ce processus d'encapsulation serait traduit de la façon suivante dans le code :

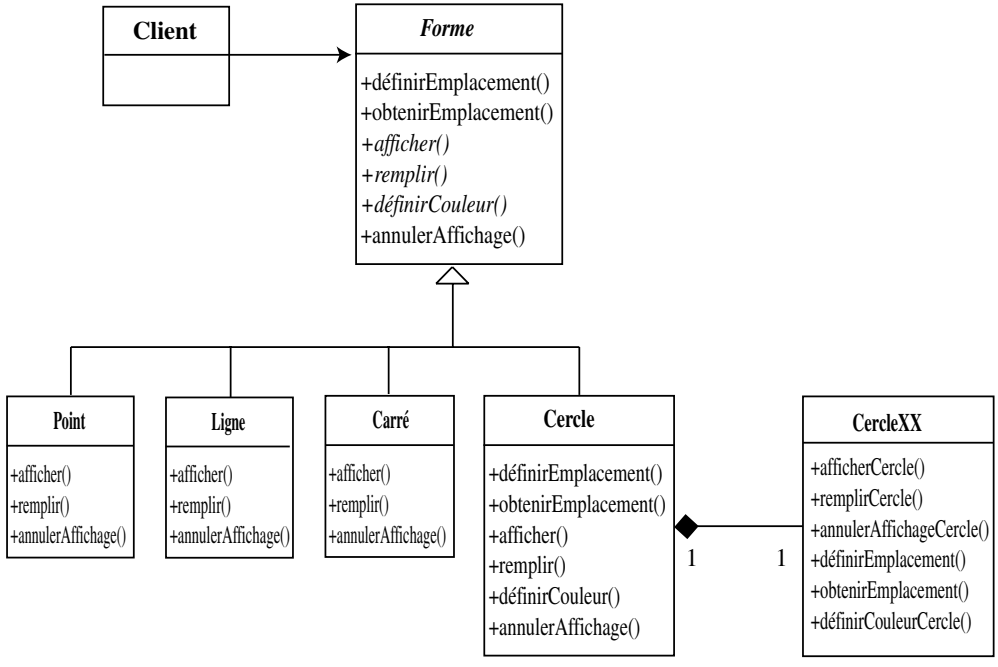


Figure 7-5
Le pattern Adaptateur : `Cercle` *encapsule* `CercleXX`

Listing 7-1 – Extrait de code Java : implémenter le pattern Adaptateur

```
class Cercle extends Forme {
   ...
   private CercleXX pcx;
   ...
   public Cercle () {
     pcx= new CercleXX();
   }

   void public afficher() {
      pcx.afficherCercle();
   }
}
```

Comme vous avez pu le constater, le pattern Adaptateur est surtout utile dans un contexte de polymorphisme. Nous verrons d'ailleurs plus tard qu'il sert également lorsque d'autres design patterns ont recours au polymorphisme.

Le pattern Adaptateur : caractéristiques

Objectif	Faire correspondre à une interface donnée un objet existant que vous ne contrôlez pas.
Problème	Un système a les bonnes données et les bonnes méthodes, mais la mauvaise interface. Généralement utilisé lorsque vous devez créer des dérivées d'une classe abstraite en cours de définition ou déjà définie.
Solution	L'adaptateur fournit un encapsuleur avec l'interface voulue.
Participants et collaborateurs	La classe `Adaptateur` adapte l'interface à la `classe Adaptee` pour qu'elle corresponde à celle de la `Cible` de l'`Adaptateur` (c'est-à-dire la classe à partir de laquelle elle est dérivée). Cela permet au `Client` d'utiliser la `classe Adaptee` comme s'il s'agissait d'un type de `Cible`.
Conséquences	Grâce au pattern Adaptateur, des objets existants peuvent être intégrés à de nouvelles structures de classes sans être limités par leur interface.
Implémentation	Intégrer la classe existante dans une autre classe. La classe qui encapsule doit être compatible avec l'interface voulue et appeler les méthodes de la classe encapsulée.
Référence Gof	Pages 139 à 150 (Pages 163 à 176 dans l'édition française).

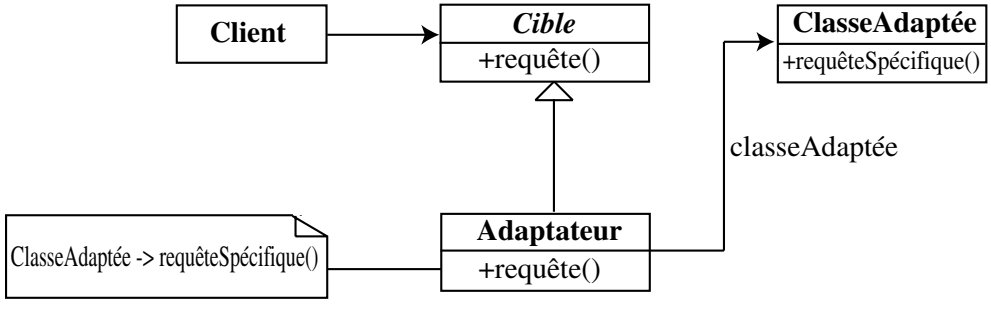


Figure 7-6
Vue standard simplifiée du pattern Adaptateur

Le pattern Adaptateur : notes pratiques

Au-delà de l'encapsulation

Vous vous retrouverez souvent dans le genre de situation que nous venons de voir, mais l'objet adapté n'aura pas nécessairement toutes les fonctions dont vous aurez besoin. Le cas échéant, vous pourrez tout de même avoir recours au pattern Adaptateur, qui ne sera pas parfait, mais grâce auquel :

- les fonctions implémentées dans la classe existante pourront être adaptées ;
- les fonctions absentes pourront être implémentées dans l'objet encapsuleur.

L'interface des classes existantes n'est plus une contrainte

Grâce au pattern Adaptateur, vous n'avez plus à vous préoccuper des interfaces des classes existantes lors de la conception. Si vous disposez d'une classe avec les fonctions dont vous avez besoin, tout au moins sur le plan conceptuel, vous pouvez utiliser le pattern Adaptateur afin de lui donner l'interface qui vous convient.

Ce concept sera d'autant plus important après l'étude d'autres patterns.

Adaptateur d'objet et Adaptateur de classe

Il existe deux types de pattern Adaptateur :

- Adaptateur d'objet – Il s'agit du pattern que nous venons d'utiliser. Il s'appuie sur un objet encapsuleur qui contient un autre objet adapté.
- Adaptateur de classe – Ce pattern est implémenté avec l'héritage multiple.

Le choix de l'un ou l'autre de ces patterns s'imposera surtout au moment de l'implémentation, en fonction des éléments impliqués.

Si vous souhaitez plus d'informations sur ce choix, reportez-vous aux pages 142 à 144 du livre de la bande des quatre (166 à 169 dans l'édition française).

Comparaison entre le pattern Adaptateur et le pattern Façade

De prime abord, ces deux patterns semblent similaires : ils impliquent tous les deux une ou plusieurs classes existantes n'ayant pas l'interface voulue et la création d'un objet avec cette interface souhaitée, comme illustré dans la figure 7-7.

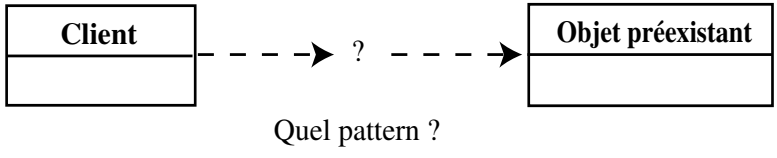


Figure 7-7
Un objet client utilisant un objet préexistant avec la mauvaise interface

Cependant, même si tous deux sont des encapsuleurs, leur méthode d'encapsulation diffère, comme le démontre le tableau 7-1 qui vous révèle les propriétés de ces patterns.

Tableau 7-1 – Comparaison entre le pattern Façade et le pattern Adaptateur

	Façade	Adaptateur
Existe-t-il des classes préexistantes ?	Oui	Oui
Une interface spécifique doit-elle être utilisée pour la conception	Non	Oui

Tableau 7-1 – Comparaison entre le pattern Façade et le pattern Adaptateur (*suite*)

	Façade	Adaptateur
Le polymorphisme est-il nécessaire ?	Non	Probablement
Une interface plus simple est-elle nécessaire ?	Oui	Non

Ce tableau nous permet de tirer les conclusions suivantes :

- les deux patterns utilisent des classes préexistantes ;
- contrairement au pattern Adaptateur, le pattern Façade n'implique pas la prise en compte d'une interface spécifique lors de la conception ;
- le polymorphisme n'intervient pas dans la façade, alors qu'il est utile dans l'adaptateur, sauf si vous devez simplement concevoir une API spécifique ;
- le pattern Façade a pour but de simplifier l'interface tandis que l'adaptateur cherche d'abord à concevoir une solution pour une interface existante.

Certains programmeurs ont tendance à conclure que le pattern Façade diffère de l'adaptateur parce qu'il masque plusieurs classes, contre une seule pour l'adaptateur. Cette conclusion se vérifie souvent, même si elle ne dépend pas du pattern lui-même. En outre, il peut arriver qu'une façade soit utilisée pour un objet excessivement complexe, tandis qu'un adaptateur peut encapsuler plusieurs petits objets implémentant la fonction voulue.

En bref, nous pouvons dire qu'une façade simplifie une interface alors qu'un adaptateur la convertit en une interface préexistante.

Rôle du pattern Adaptateur dans notre exemple de CFAO

Dans le cadre de notre problème de CFAO (voir le chapitre 3), les motifs du système V2 sont représentés par des objets `MotifGOO`. Malheureusement, étant donné que ces objets ont été conçus par un autre programmeur, ils ne disposent pas d'une interface appropriée et il est impossible de les dériver de la classe `Motif`.

Dans ce cas, nous n'avons même pas la possibilité d'écrire de nouvelles classes pour implémenter cette fonction. Nous devons communiquer avec les objets `MotifGOO`. Le pattern Adaptateur est parfait pour nous aider à résoudre ce problème.

Exemple de code en C++

Listing 7-2 – Extrait de code en C++ : implémenter le pattern Adaptateur

```
class Cercle : public Forme {
   .  .  .
  private:
    CercleXX *pcx;
   .  .  .
}
Cercle::Cercle () {
   .  .  .
   pcx= new CercleXX();
}
void Cercle::afficher () {
   pcx->afficherCercle();
}
```