36

Introduction aux Design Patterns

Au fil des chapitres précédents, tout en présentant les caractéristiques détaillées du langage C++, nous avons montré comment utiliser à bon escient les fondements de la P.O.O. que sont l'encapsulation, l'héritage, la composition, le polymorphisme et les classes abstraites. Néanmoins, lors du développement de grosses applications, des problèmes dits de « conception », risquent d'apparaître. Par exemple, il faudra trouver des réponses à des questions telles que :

- Comment choisir les bonnes classes ?
- Comment gérer les relations entre les différentes classes, les faire coopérer tout en les gardant suffisamment autonomes pour être réutilisables ?
- Comment faire face à l'évolution des besoins des utilisateurs du code, sans remettre en cause l'existant ?

Le développeur, pour affiner son art, doit alors compléter sa panoplie par des principes plus orientés vers la conception. Parmi ceux-ci, les *Design Patterns*[1] occupent une place de choix. A priori, il semble facile de fournir une définition formelle d'un pattern telle que « manière de résoudre, de façon indépendante d'un langage, à l'aide d'une organisation appropriée de classes, un problème qui se pose de façon récurente ». Il est tout aussi facile de fournir un diagramme UML (*Unified Modeling Langage*) qui décrit l'articulation des classes concernées. Mais l'expérience montre que ce n'est que par la réflexion et l'utilisation répétée d'un pattern que le concepteur en appréhende la nature véritable, les subtilités qu'il contient, les

1. En Français, on trouvera : motifs de conception, patrons de conception. Exceptionnellement, ici, nous conserverons la terminologie anglaise, largement utilisée dans la littérature en Français.

variantes auxquelles il peut conduire et la manière dont il est nuancé par son application dans un langage donné.

Nous vous proposons d'étudier ici les plus répandus de ces patterns, en vous fournissant des exemples de mise en œuvre en C++. Si ces derniers, de par leur simplicité, ne justifient pas toujours le recours à un pattern, ils n'en permettent pas moins de mettre en évidence d'une manière concise à la fois la structure du pattern et son implémentation en C++.

1 Généralités

1.1 Historique

Bien que l'on ait tendance à associer la notion de pattern à la programmation, le concept est apparu dans le domaine de l'architecture, notamment lors de la parution en 1977 de l'ouvrage *A Pattern Language*[2]. Ce n'est qu'ultérieurement qu'il a été repris par les développeurs, donnant lieu à la publication de nombreux ouvrages. Le premier en date, qui reste d'ailleurs le plus connu est l'ouvrage *Design Patterns*[3], paru en 1995, dont les quatre auteurs sont surnommés *gang of four* (bande des quatre, abrégé souvent en *gof*) dans la littérature. Il décrit 23 patterns d'intérêt général. D'autres patterns ont été proposés par la suite, parfois dans des domaines spécialisés.

On a tendance à classer les patterns en catégories, la classification la plus répandue étant celle du *gof* :

- patterns de création : ils permettent d'isoler la création des objets de leur utilisation ;
- patterns de structure : ils permettent d'assembler des classes ou des objets individuels en des structures plus complexes ;
- patterns de comportement : ils portent sur l'utilisation d'algorithmes et sur l'affectation de responsabilités aux objets.

Il ne faut toutefois pas s'attacher trop formellement à ces classifications qui risquent parfois de cacher la véritable nature du pattern. De plus, certains auteurs utilisent des classifications différentes. Vous pourrez trouver des patterns d'opérations, d'extensions, de responsabilité, d'interface. Parfois, un même pattern pourrait être classé dans deux catégories différentes.

2. Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

1.2 Patterns et P.O.O.

Un pattern est indépendant d'un langage, mais il utilise les notions fondamentales de la P.O.O. que sont les classes, les objets, l'encapsulation, l'héritage et le polymorphisme.

De plus, de nombreux patterns (dont ceux du *gof*) reposent sur les notions de classe abstraite, non instanciable, utilisée pour donner naissance, par héritage, à une hiérarchie de classes instanciables, dites concrètes. Contrairement à ce qui pourrait se produire avec d'autres langages (par exemple Java), la mise en œuvre d'une telle classe abstraite ne posera guère de problème en C++, hormis le fait que pour être abstraite, une classe doit comporter au moins une méthode virtuelle pure. Les rares fois où cela ne sera pas le cas, on aura alors affaire à une classe (concrète), potentiellement instanciable...

Une autre notion qui apparaît souvent dans les patterns est celle d'interface qui, rappelons-le, correspond à l'ensemble des en-têtes des méthodes publiques. Contrairement à Java, C++ ne permet pas d'exprimer directement qu'une classe implémente uen interface. On peut cependant « simuler » la situation en recourant à une classe abstraite ne contenant que des méthodes publiques virtuelles pures.

Dans les explications relatives à un pattern, on parlera souvent de *client*. Il s'agit en fait du code écrit pour utiliser un code existant appliquant le pattern (dans nos exemples, ce code sera simplement la méthode *main*). À ce propos, notez que, comme nous l'avons fait la plupart du temps dans le reste de l'ouvrage par souci de simplicité et de lisibilité, nous placerons toujours l'ensemble du code dans un seul fichier. Ce ne sera jamais le cas en pratique, ne serait-ce que parce que le code du client sera obligatoirement distinct du reste ; de plus, il faudra bien lui fournir les fichiers en-têtes relatifs aux classes qu'on lui transmettre par ailleurs sous forme de module objet.

1.3 Patterns et C.O.O.

On peut dire que tous les patterns utilisent la mise en œuvre, éprouvée au fil du temps par une communauté de développeurs, de tout ou partie de ce que nous nommerons « principes élémentaires de Conception Orientée Objet ». Parmi ces principes, on peut notamment citer :

- concevoir pour des abstractions (classes abstraites)[4], non pour des classes concrètes : grâce au polymorphisme, les classes concrètes peuvent ainsi évoluer, sans que le client ne soit concerné ;
- trouver les variations et les encapsuler dans des classes prévues à cet effet ; il peut s'agir des variations qui apparaissent lors de la conception initiale, mais aussi des demandes de modifications qu'on est prêt à accepter ultérieurement de la part du client ;

4. Dans d'autres langages comme Java, on peut aussi concevoir pour des interfaces, grâce à la notion de polymorphisme d'interface.

- savoir dans certains cas utiliser la composition plutôt que l'héritage ; nous verrons que la composition offre des possibilités de modification dynamique (pendant l'exécution) que n'apporte pas l'héritage ;
- limiter le « couplage » entre les objets, c'est-à-dire minimiser la connaissance qu'un objet doit avoir d'un autre objet pour pouvoir l'utiliser.

Précisément, l'étude de patterns va vous permettre de mieux appréhender de telles notions et d'en percevoir l'intérêt.

1.4 Patterns et pointeurs intelligents

La plupart des design pattern sont basés sur le polymorphisme, de sorte qu'ils nécessitent l'emploi de pointeurs ou, plus rarement, de références (celles-ci souffrant des limitations exposées au paragraphe 4.2 du chapitre 22). Dans le cas des pointeurs, se pose alors le problème de savoir s'il faut ou non recourir aux pointeurs intelligents.

Tout d'abord, il est nécessaire de distinguer le code du pattern lui-même du code du client. Dans le code du pattern, il faut distinguer trois situations :

- le code utilise des pointeurs en interne qui n'ont donc pas à être connus du client : aucune contrainte particulière ne pèse alors sur leur nature et le pattern est libre de les implanter comme il l'entend ; notez que ce cas ne se présente pas dans les patterns examinés ici ;
- le code fournit des pointeurs sur des objets qu'il a lui-même créés ; là encore, il peut recourir indifféremment à des pointeurs nus ou des pointeurs intelligents (généralement, il s'agira de pointeurs de type *unique_ptr*, compte tenu des propriétés des fonctions dites de « fabrique », exposées au paragraphe 4.6 du chapitre 10) ; mais, le client doit avoir connaissance de l'implémentation utilisée ;
- le code utilise des adresses d'objet fournies par le client ; là encore, il peut théoriquement recourir à des pointeurs nus ou des pointeurs intelligents (en général, *shared_ptr*, pour éviter que le client ne perde la propirété de l'objet concerné).Toutefois, on n'oubliera pas que C++ permet de faire cohabiter des objets automatiques et des objets dynamiques et que seuls les derniers sont susceptibles d'être référencés par des pointeurs intelligents.

En définitive, on voit qu'il existe beaucoup de possibilités. On notera que les propriétés relatives des pointeurs nus et des pointeurs intelligents font qu'il est plus facile à un client de s'adapter à un code de pattern utilisant des pointeurs natifs que l'inverse. Ici, nous avons choisi d'écrire des patterns n'imposant pas de contraintes particulières aux clients, c'est-à-dire employant des pointeurs nus. Néanmoins, il nous arrivera de fournir une seconde version plus contraignante recourant à des pointeurs intelligents.

En revanche, pour ce qui est du code du client, nous proposerons souvent deux versions, l'une avec pointeurs natifs, l'autre avec pointeurs intelligents (en l'occurence ici *unique_ptr*). Cela constitue d'ailleurs un bon exercice d'application des propriétés de ces pointeurs (récapitulées au paragraphe 7 du chapitre 23).

2 Les patterns de construction

Pour instancier un objet, il faut utiliser un contructeur, donc en connaître le type et disposer des informations nécessaires (paramètres du constructeur). Mais il est toujours possible de déléguer la création d'un objet à une autre classe, laquelle pourra décider d'utiliser le constructeur voulu et, éventuellement, se charger d'obtenir les informations nécessaires.

D'une manière générale, les patterns de construction abstraient le processus d'instanciation d'un objet en permettant au client d'ignorer le type exact de l'objet instancié, lui permettant d'écrire un code plus souple puisque indépendant des classes concrètes.

Ici, nous étudierons les deux principaux patterns de construction que sont *Factory Method* (nommé parfois « constructeur virtuel ») et *Abstract Factory*[5].

2.1 Le pattern Factory Method (Fabrique)

2.1.1 Premier exemple

Supposons que l'on souhaite :

- disposer d'un ensemble de classes servant à manipuler des formes représentant des « logos » et possédant toutes une méthode d'affichage *affiche* ;
- pouvoir créer et utiliser un tel logo, sans avoir besoin d'en connaître la classe exacte.

Ici, nous nous limiterons à deux classes de logos nommées *LogoCercle* et *LogoRectangle* et, pour l'instant, nous ferons en sorte que la classe utilisée soit choisie au hasard.

Pour appliquer le pattern *Factory Method*, nous devons tout d'abord faire dériver nos deux classes logo d'une classe abstraite que nous nommerons *Logo*. Nous la munirons d'un destructeur virtuel, afin que les destructeurs des classes dérivées soient bien virtuels (revoyez éventuellement le paragraphe 5.6 du chapitre 22).

```
class Logo
{ public :
    virtual void affiche () = 0 ;
    virtual ~Logo() {} ;   // ou mieux, depuis C++11 : virtual ~Logo() = default ;
} ;
class LogoCercle : public Logo
{ public :
    void affiche () { cout << "Logo circulaire" << endl ; }
} ;
class LogoRectangle : public Logo
{ public :
    void affiche () { cout << "Logo rectangle" << endl ; }
} ;
```

5. Pour les noms de patterns, nous conserverons également la terminologie anglaise, tout en fournissant une possiblité de traduction en français.

Puis nous créons une classe *FabriqueLogoHasard* dotée d'une méthode *getLogo* chargée de fournir au hasard un objet logo, suivant ce schéma :

```
class FabriqueLogoHasard
{ public :
    Logo *getLogo()
    { if (.....) return new LogoCercle () ;
           else return new LogoRectangle () ;
    }
} ;
```

On voit que cette méthode peut fournir un objet de type logo dont on sait simplement qu'il dérive de *Logo*. Voici un exemple complet dans lequel le code du client est décliné en deux versions : pointeurs natifs, pointeurs intelligents :

```
// FactoryMethod1
#include <iostream>
#include <cstdlib>
#include <memory>     // pour unique_ptr
using namespace std ;
class Logo
{ public :
   virtual void affiche () = 0 ;
   virtual ~Logo() {} ;  // ou mieux, depuis C++11 : virtual ~Logo() = default ;
} ;
class LogoCercle : public Logo
{ public :
    void affiche () { cout << "Logo circulaire" << endl ; }
} ;
class LogoRectangle : public Logo
{ public :
    void affiche () { cout << "Logo rectangle" << endl ; }
} ;
class FabriqueLogoHasard
{ public :
    Logo *getLogo()
    { int n = rand () ;
      if (n < RAND_MAX/2) return new LogoCercle () ;
                    else return new LogoRectangle () ;
    }
} ;

int main ()    /*************** Version pointeurs natifs *********************/
{ FabriqueLogoHasard fab  ;
    for (int i = 0 ; i<4 ; i++)
    { Logo *l = fab.getLogo() ;   // auto (C++11) utilisable ici car l est recree
                                  // a chaque tour de boucle
      l->affiche() ;
      delete l ;
    }
}
```

```
int main ()    /*************** Version pointeurs intelligents ***************/
{ FabriqueLogoHasard fab  ;
    for (int i = 0 ; i<4 ; i++)
    { unique_ptr<Logo> l (fab.getLogo()) ;   // ici, avec auto, le code fonctionne
                                             // mais l est de type Logo *
      l->affiche() ;
    }
}
```

```
Logo circulaire
Logo rectangle
Logo circulaire
Logo rectangle
```

Exemple d'utilisation du pattern Factory Method *(1)*

On voit que le pattern *Factory Method* nous a permis de dissocier la création des objets de leur utilisation. Le client (ici *main*) peut utiliser des objets logos, sans en connaître le type concret. Il sait simplement que ce type dérive du type abstrait *Logo*. On peut dire qu'on a « encapsulé » la création des objets et que le client se contente de programmer pour des classes abstraites, non pour des objets concrets.

Si l'on souhaite ajouter de nouvelles classes logos, il suffira d'adapter la fabrique en conséquence, sans que le client n'ait à modifier son code.

Notez qu'ici, c'est la fabrique qui décide du logo à créer. Cela n'est pas imposé par le pattern. Nous y reviendrons un peu plus loin.

Remarque

À titre indicatif, voici comment nous pourrions utiliser des pointeurs intelligents dans le pattern lui-même :

```
// FactoryMethod1
#include <iostream>
#include <cstdlib>
#include <memory>     // pour unique_ptr
using namespace std ;
class Logo
{ public :
    virtual void affiche () = 0 ;
    virtual ~Logo() {} ;   // ou, avec C++11 : virtual ~Logo() = default ;
} ;
class LogoCercle : public Logo
{ public :
    void affiche () { cout << "Logo circulaire" << endl ; }
} ;
class LogoRectangle : public Logo
{ public :
    void affiche () { cout << "Logo rectangle" << endl ; }
} ;
```

```
class FabriqueLogoHasard
{ public :
    unique_ptr<Logo> getLogo()
    { int n = rand () ;
      if (n < RAND_MAX/2) return make_unique<LogoCercle> () ;
                    else return make_unique<LogoRectangle> () ;
    }
} ;
int main ()    /*************** Version pointeurs intelligents ***************/
{ FabriqueLogoHasard fab  ;
    for (int i = 0 ; i<4 ; i++)
    { auto l { fab.getLogo()} ;
      l->affiche() ;
    }
}
```

2.1.2 Deuxième exemple

D'une manière générale, le pattern *Factory Method* prévoit qu'il puisse exister plusieurs fabriques qui vont alors dériver d'une même classe abstraite. Nous vous proposons un nouvel exemple exploitant cette possibilité pour créer :

- une fabrique de logos choisis au hasard (la même que dans l'exemple précédent) ;
- une fabrique de logos choisis de façon alternée, nommée *FabriqueLogoAlternes*.

On utilise ici une variable statique booléenne (*indic*) qu'on fait basculer entre vrai et faux, à chaque création d'un nouveau logo.

```
// FactoryMethod2
#include <iostream>
#include <cstdlib>
#include <memory>    // pour unique_ptr
using namespace std ;
class Logo
{ public :
    virtual void affiche () = 0 ;
    virtual ~Logo() {} ;   // ou, avec C++11 : virtual ~Logo() = default ;
} ;
class LogoCercle : public Logo
{ public :
    void affiche () { cout << "Logo circulaire" << endl ; }
} ;
class LogoRectangle : public Logo
{ public :
    void affiche () { cout << "Logo rectangle" << endl  ; }
} ;
class FabriqueLogo
{ public :
    virtual Logo *getLogo() = 0 ;
    virtual ~FabriqueLogo() { }
} ;
```

```
class FabriqueLogoHasard : public FabriqueLogo
{ public :
    Logo *getLogo()
    { int n = rand () ;
      if (n < RAND_MAX/2) return new LogoCercle () ;
                    else return new LogoRectangle () ;
    }
} ;
class FabriqueLogoAlternes : public FabriqueLogo
{ public :
    Logo *getLogo ()
    {  if (indic) { indic = false ; return new LogoCercle() ; }
             else { indic = true ;  return new LogoRectangle () ; }
    }
  private :
    static bool indic ;
} ;
bool FabriqueLogoAlternes::indic = false ;    // initialisation membre statique
```

```
int main ()    /*************** Version pointeurs natifs *********************/
{ FabriqueLogo *fab  ;   // auto pas utilisable ici car fab change de type ensuite
  fab = new FabriqueLogoHasard () ;  // fab dynamique pour le polymorphisme
  cout << "--- avec Fabrique au hasard" << endl ;
  for (int i = 0 ; i<3 ; i++)
  { Logo *l = fab->getLogo() ; l->affiche(); delete l ; }
  delete fab ;
  fab = new FabriqueLogoAlternes () ;
  cout << "--- avec Fabrique alternee" << endl ;
  for (int i = 0 ; i<3 ; i++)
  { Logo *l = fab->getLogo() ; l->affiche(); delete l ; }
  delete fab ;
}
```

```
int main ()    /*************** Version pointeurs intelligents ***************/
{ unique_ptr<FabriqueLogo> fab (new FabriqueLogoHasard()) ;
  cout << "--- avec Fabrique au hasard" << endl ;
  for (int i = 0 ; i<3 ; i++)
  { unique_ptr<Logo> l(fab->getLogo()) ; l->affiche() ;
  }
  fab = unique_ptr<FabriqueLogo> (new FabriqueLogoAlternes()) ;
    // il n'est pas necessaire de faire  (voir &7 chap 23) :
    // fab = move (unique_ptr<FabriqueLogo> (new FabriqueLogoAlternes())) ;
  cout << "--- avec Fabrique alternee" << endl ;
  for (int i = 0 ; i<3 ; i++)
  { unique_ptr<Logo> l (fab->getLogo()) ; l->affiche() ;
  }
}
```

```
--- avec Fabrique au hasard
Logo circulaire
```

```
Logo rectangle
Logo circulaire
--- avec Fabrique alternee
Logo rectangle
Logo circulaire
Logo circulaire
```

2.1.3 Discussion

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. Dans nos exemples, le client se contentait de laisser la fabrique décider. Mais on pourrait aussi rencontrer des « fabriques paramétrées », dans lesquelles le client fourni un paramètre à la fabrique indiquant le type de produit à instancier. On notera bien que cette possibilité introduit généralement des instructions conditionnelles dans la fabrique, telles que :

```
if (....) return new LogoCercle ()
If (....) return new LogoRectangle ()
```

Mais, la gestion de la modification de logos reste transparente au client. Toutefois, en cas d'introduction de nouveaux logos, il doit simplement être prévenu de l'existence de nouvelles valeurs pour le paramètre de choix du logo.

On peut utiliser ce pattern en prévoyant une fabrique concrète par produit. Dans ce cas, choisir la fabrique revient à choisir le produit. Mais le client n'a toujours pas besoin de connaître la classe concrète réellement instanciée par la fabrique. Dans ce cas, il peut être intéressant de recourrir aux possiblités de programmation générique (patrons de classes) en créant une version générique de la fabrique.

2.2 Le pattern Abstract Factory (Fabrique Abstraite)

2.2.1 Présentation

Ce pattern intervient lorsque l'on a affaire à plusieurs familles d'objets et qu'il est nécessaire, à un moment donné, d'utiliser les objets d'une seule famille. Par exemple, supposons que, à l'image de ce qui se produit dans les « modèles d'aspect » des interfaces graphiques, l'on dispose de deux façons, nommées *style A* et *style B* de représenter des composants graphiques. Ici, pour simplifier, nous nous limiterons à des boutons radios et des cases à cocher.

Le pattern *Abstract Factory* va nous permettre de dissocier la création des objets concrets de leur utilisation. Pour l'exploiter, il faut tout d'abord que les classes concrètes représentant les deux styles de boutons radios dérivent d'une classe abstraite commune. Il en ira de même pour les cases à cocher.

Pour les boutons radio, nous procéderons ainsi, en supposant leurs classes dotées d'une seule méthode (nommée *type*) identifiant la nature de l'objet (ici, bouton radio) et son style (ici A ou B) :

```
class BoutonRadio   // classe abstraite pour les boutons radio
{ public :
    virtual string type ()= 0 ;
    virtual ~BoutonRadio() { } // ou, avec C++11 : virtual ~BoutonRadio() = default ;
} ;
class BoutonRadioA : public BoutonRadio
{ public :
    string type () { return "Bouton radio Style A" ; }
} ;
class BoutonRadioB : public BoutonRadio
{ public :
    string type () { return "Bouton radio style B" ; }
} ;
```

Nous procédons de façon semblable pour les cases à cocher, en remarquant que le pattern n'impose pas qu'elles dérivent de la même classe abstraite que les boutons radio ; ici, nous doterons également leurs classes d'une unique méthode nommée *identification*, identifiant là aussi le type d'objet (case à cocher) et son style :

```
class CaseCocher   // classe abstraite pour les cases a cocher
{ public :
    virtual string identification () = 0 ;
    virtual ~CaseCocher() { } // ou, avec C++11 : virtual ~CaseCocher() = default ;
} ;
class CaseCocherA : public CaseCocher
{ public :
    string identification () { return "Case a cocher style A" ; }
} ;
class CaseCocherB : public CaseCocher
{ public :
    string identification () { return "Case a cocher style B" ; }
} ;
```

Puis nous créons deux classes concrètes de fabrique, l'une pour les composants de style A, l'autre pour les composants de style B en les faisant dériver d'une même classe abstraite :

```
class FabriqueAbstraite   // classe abstraite pour les fabriques
{ public :
    virtual BoutonRadio *creerBoutonRadio () = 0 ;
    virtual CaseCocher  *creerCaseCocher () = 0 ;
    virtual ~FabriqueAbstraite() { }    // ou =default depuis C++11
} ;
class FabriqueStyleA : public FabriqueAbstraite
{ public :
    BoutonRadio *creerBoutonRadio ()  { return new BoutonRadioA ()  ; }
    CaseCocher  *creerCaseCocher ()   { return new CaseCocherA () ; }
} ;
class FabriqueStyleB : public FabriqueAbstraite
{ public :
    BoutonRadio *creerBoutonRadio ()  { return new BoutonRadioB () ; }
    CaseCocher  *creerCaseCocher ()   { return new CaseCocherB () ; }
} ;
```

Voici un exemple complet dans lequel nous nous contentons d'afficher un message au moment de la création d'un composant (en utilisant son unique méthode *type* ou *identification*) :

```
// FabriqueAbstraite
#include <iostream>
#include <memory>     // pour unique_ptr
using namespace std ;
class BoutonRadio     // classe abstraite pour les boutons radio
{ public :
    virtual string type ()= 0 ;
    virtual ~BoutonRadio() { } // ou, avec C++11 : virtual ~BoutonRadio()=default ;
} ;
class BoutonRadioA : public BoutonRadio
{ public :
    string type () { return "Bouton radio Style A" ; }
} ;
class BoutonRadioB : public BoutonRadio
{ public :
    string type () { return "Bouton radio style B" ; }
} ;
class CaseCocher   // classe abstraite pour les cases a cocher
{ public :
    virtual string identification () = 0 ;
    virtual ~CaseCocher() { } // ou, avec C++11 : virtual ~CaseCocher() = default ;
} ;
class CaseCocherA : public CaseCocher
{ public :
    string identification () { return "Case a cocher style A" ; }
} ;
class CaseCocherB : public CaseCocher
{ public :
    string identification () { return "Case a cocher style B" ; }
} ;
class FabriqueAbstraite   // classe abstraite pour les fabriques
{ public :
    virtual BoutonRadio *creerBoutonRadio () = 0 ;
    virtual CaseCocher  *creerCaseCocher () = 0 ;
    virtual ~FabriqueAbstraite() { }   // ou, avec C++11 = default ;
} ;
class FabriqueStyleA : public FabriqueAbstraite
{ public :
    BoutonRadio *creerBoutonRadio ()  { return new BoutonRadioA ()  ; }
    CaseCocher  *creerCaseCocher ()   { return new CaseCocherA () ; }
} ;
class FabriqueStyleB : public FabriqueAbstraite
{ public :
    BoutonRadio *creerBoutonRadio ()  { return new BoutonRadioB () ; }
    CaseCocher  *creerCaseCocher ()   { return new CaseCocherB () ; }
} ;
```

```
int main ()    /*************** Version pointeurs natifs *********************/
{ auto *f  = new FabriqueStyleA() ;    // auto possible ici car f
                                       // ne change pas de type
  auto br1 = f->creerBoutonRadio () ; cout << br1->type () << endl ;
  auto cc  = f->creerCaseCocher ()  ; cout << cc->identification () << endl ;
  auto br2 = f->creerBoutonRadio () ; cout << br2->type () << endl ;
  delete br1 ; delete cc ; delete br2 ; delete f ;
}

int main ()    /*************** Version pointeurs intelligents ***************/
{ unique_ptr<FabriqueAbstraite> f (new FabriqueStyleA()) ;
  unique_ptr<BoutonRadio> br1(f->creerBoutonRadio ()) ;
  cout << br1->type () << endl ;
  unique_ptr<CaseCocher> cc(f->creerCaseCocher ())  ;
  cout << cc->identification () << endl ;
  unique_ptr<BoutonRadio> br2(f->creerBoutonRadio ()) ;
  cout << br2->type () << endl ;
}

Bouton radio Style A
Case a cocher style A
Bouton radio Style A
```

Exemple d'utilisation du pattern Abstract Factory

Ici encore, 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 classes abstraites, non pour des objets concrets.

En outre, ici, un simple changement de fabrique change le style de toute une famille d'objets.

Si l'on souhaite ajouter un nouveau style, il suffit de créer une nouvelle classe de fabrique dérivée de *FabriqueAstraite* et de créer les nouveaux composants voulus.

2.2.2 Discussion

On notera bien que, dans ce pattern, il n'existe de lien de parenté qu'entre deux classes de composants de même nature et de style différent. Il n'existe aucun lien entre différents composants d'un même style. C'est d'ailleurs pour mettre cet aspect en évidence que nous avons choisi d'attribuer des noms différents (*type* et *identification*) à des méthodes jouant un rôle semblable[6]. Les classes de composants peuvent donc avoir des interfaces (au sens général)

6. En toute rigueur, il n'était même pas nécessaire que boutons radio et cases à cocher disposent de méthodes semblables...

totalement différentes. Ici, le client sait s'il manipule un bouton radio ou une case à cocher ; en revanche, il n'a pas à en connaître le style. Mais, il reste possible que les composants d'une même famille appartiennent à une même hiérachie ; dans ce cas, il faudra simplement recourrir à l'héritage multiple.

Dans notre exemple, le choix de la fabrique, c'est-à-dire d'un style, est effectué par le client (*main*). Il pourrait également dépendre de paramètres externes au code, comme un « fichier de configuration ». Dans ce cas, plutôt que de laisser le client s'en occuper, on pourrait confier le choix à une méthode de classe (attribut *static*) *getFabrique* de la classe abstraite.

Enfin, notons que ce pattern présente une contrainte importante au niveau de l'ajout de nouveaux éléments à une famille. Ainsi, dans notre exemple, si l'on souhaite prendre en compte un nouveau type de composant (par exemple un bouton), il faudra intervenir à la fois sur la fabrique abstraite et sur toutes les fabriques concrètes.

3 Les patterns de structure

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.

Ici, nous étudierons deux de ces patterns : *Composite* et *Decorator*.

3.1 Le pattern Composite

3.1.1 Présentation

Supposons que dans un système de représentation graphique, nous souhaitions pouvoir gérer différentes formes, en imposant les contraintes suivantes :

- on peut créer des groupes d'objets regroupant plusieurs formes ;
- les groupes peuvent contenir, non seulement des formes, mais aussi d'autres groupes (la notion de groupe est donc récursive) ;
- les formes et les groupes doivent pouvoir être traités de la même manière, sans qu'il ne soit nécessaire d'en connaître la véritable nature.

Pour simplifier l'exposé, nous supposerons que nous ne disposons que de deux formes *Cercle* et *Rectangle* et que les méthodes applicables aux groupes et aux formes se limitent à la seule méthode *affiche*.

L'application du pattern *Composite* nous offre une solution satisfaisant aux contraintes imposées. Pour ce faire, nous créons une classe de base abstraite (*Composant*) dont dériveront à la fois les formes (*Cercle* et *Rectangle*) et les groupes (classe *Groupe*). On y trouvera tout naturellement la méthode *affiche*, mais également au moins une méthode (nommée ici *ajoute*)

permettant d'ajouter un élément (forme ou groupe) à un groupe et dont l'en-tête sera de la forme :

```
void ajoute (Composant *)
```

Voici une première ébauche de cette classe *Composant* :

```
class Composant
{ public :
    virtual void ajoute (Composant *c) {}  // par défaut, ne fait rien
    virtual void affiche () = 0 ;  // a redefinir dans chaque classe concrete
    .....
} ;
```

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. Ici, nous pouvons régler le problème en dotant cette méthode d'un corps vide dans la classe *Composant*. Ainsi, nous n'aurons pas besoin de la redéfinir dans les classes de formes et son éventuel appel par une forme sera simplement sans effet.

Nous définissons ensuite les classes représentant les formes (ici *Cercle* et *Rectangle*) et les groupes (*Groupe*), en les dérivant de la classe *Composant.* Par exemple, pour *Cercle* :

```
class Cercle : public Composant
{ public :
    virtual void affiche () { .....}
    .....
} ;
```

Le schéma sera comparable pour *Rectangle* et pour *Groupe*. Mais, on voit que la méthode *affiche* de *Groupe* devra en fait appeler la méthode *affiche* de tous ses composants (du moins c'est là une requête raisonnable si l'on souhaite que l'affichage d'un groupe entraîne bien l'affichage de chacun de ses constituants). Dans ces conditions, il est nécessaire qu'un groupe conserve les références de ses composants ; nous utiliserons à cet effet un objet de type *vector <Composant *>*, dans lequel la méthode *ajoute* viendra introduire un pointeur sur le nouveau composant ajouté au groupe. Voici une première ébauche de notre classe *Groupe* :

```
class Groupe : public Composant
{ public :
    virtual void ajoute (Composant *c) { listeComposants.push_back (c) ; }
    virtual void affiche ()
    { .....  // affichage infos du groupe lui-meme
      // puis affichage des infos pour chaque composant du groupe
      for (int i = 0 ; i<listeComposants.size() ; i++)
      { listeComposants[i]->affiche() ; }
    }
  private :
    vector<Composant *> listeComposants ;
} ;
```

Voici en définitive un exemple complet dans lequel nous avons prévu de repérer chaque composant par un nom fourni en argument à son constructeur, une méthode *getString* permettant

de le récupérer. Quant à la méthode *affiche*, elle se contente d'afficher le nom et la nature du composant, ainsi que le nom et la nature de chacun des éventuels composants qu'il contient, s'il s'agit d'un groupe. Notez qu'ici, nous avons volontairement mélangé des composants automatiques et des composants dynamiques.

```
// Composite
#include <iostream>
#include <vector>
#include <memory>     // pour unique_ptr
using namespace std ;
class Composant
{ public :
   virtual void ajoute (Composant *) {}  // par defaut, ne fait rien
   virtual void affiche () = 0 ;  // a redefinir dans chaque classe concrete
   Composant (string nom_c) : nom (nom_c) { }
   virtual string getNom () { return nom ; }
   virtual ~Composant () {}  // ou, avec C++11 virtual ~Composant () = default ;
  private :
   string nom ;
} ;
class Cercle : public Composant
{ public :
   Cercle (string nom_c) : Composant (nom_c) { }
   virtual void affiche () { cout << "Cercle " << getNom() << endl ; }
} ;
class Rectangle : public Composant
{ public :
   Rectangle (string nom_c) : Composant (nom_c) { }
   virtual void affiche () { cout << "Rectangle " << getNom() << endl; }
} ;
class Groupe : public Composant
{ public :
  Groupe (string nom_c): Composant (nom_c), listeComposants(vector<Composant *>())
    { }
  virtual void ajoute (Composant *c) { listeComposants.push_back (c) ; }
  virtual void affiche ()
  { cout << "---- Groupe " << getNom() << " contenant : " << endl ;
    for (auto & compo : listeComposants)  compo->affiche() ;
    cout << "-------- fin groupe " << getNom() << endl ;
  }
  private :
    vector<Composant *> listeComposants ;
} ;

int main ()    /*************** Version pointeurs natifs *********************/
{ Cercle *c1 = new Cercle ("C1") ; Cercle *c2 = new Cercle ("C2") ;
   Rectangle r1 ("R1") ;   // objet automatique
   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 () ;
```

```
   delete c1 ; delete c2 ;
   delete ga ; delete gb ;
}
```

```
int main () /**** Version pointeurs intelligents (pour les objets dynamiques ****/
{  unique_ptr<Cercle> c1 (new Cercle ("C1")) ;
   unique_ptr<Cercle> c2  (new Cercle ("C2")) ;
   Rectangle r1 ("R1") ;   // objet automatique
   c1->affiche () ;
   unique_ptr<Groupe> ga (new Groupe ("GA")) ;
   ga->ajoute(c1.get()) ; ga->ajoute(&r1) ; ga->affiche () ;
   unique_ptr<Groupe> gb (new Groupe ("GB")) ;
   gb->ajoute(ga.get()) ; gb->ajoute(c2.get()) ;  gb->affiche () ;
}
```

```
Cercle C1
---- Groupe GA contenant :
Cercle C1
Rectangle R1
-------- fin groupe GA
---- Groupe GB contenant :
---- Groupe GA contenant :
Cercle C1
Rectangle R1
-------- fin groupe GA
Cercle C2
-------- fin groupe GB
```

Exemple d'application du pattern Composite

Remarques

1 En pratique, en plus de la méthode *ajoute*, les groupes disposeront généralement d'une méthode de suppression d'un composant.

2 Si, au lieu de conserver simplement les pointeurs sur les composants, nous avions prévu de conserver les composants eux-mêmes, par exemple dans un conteneur de type *vector <Composant>*, chaque ajout au conteneur aurait entraîné une copie du composant concerné. Il aurait fallu alors définir clairement par quelle méthode cette copie devait être détruite.

3.1.2 Discussion

Ici, chaque groupe maintient l'adresse des objets qu'il contient. Réciproquement, on pourrait envisager que chaque composant comporte une adresse de l'éventuel groupe dans lequel il est contenu. Cela pourrait faciliter le travail de certaines méthodes.

Sur le plan du vocabulaire, vous rencontrerez souvent le terme de « feuille » pour décrire les composants simples (nos formes) et de « nœud » pour les groupes. Cette terminologie rejoint

celle utilisée dans les structures arborescentes. Cela pourrait laisser penser que les structures créées par ce pattern sont obligatoirement des arbres, c'est-à-dire qu'un élément quelconque (feuille ou nœud) yest toujours contenu dans au plus un autre élément. En fait, rien dans le pattern n'interdit d'introduire plusieurs fois un même composant dans deux groupes différents. C'est ce qui se produirait dans notre exemple, en faisant simplement :

```
gb->ajoute(c2) ; ga->ajoute(c2) ;  // ici c2 est à la fois dans ga et dans gb
```

Dans la plupart des cas, cette situation pourra s'avérer gênante (supposez que vous souhaitiez compter le nombre de composants simples présents à un moment donné !). Il faudra alors prendre les précautions voulues pour éviter qu'elle ne se présente.

Comme on l'a vu dans l'exemple, ce pattern amène à prévoir dans la classe abstraite *Composant* des méthodes (ici *ajoute*) qui peuvent ne pas avoir de sens pour les composants simples. Nous avons réglé le problème en en fournissant une version par défaut ne faisant rien. On pourrait envisager de ne placer ces méthodes que dans les groupes. Dans ce cas, le client devra tester la nature d'un composant pour savoir s'il peut lui appliquer la méthode *ajoute*. On peut lui simplifier un peu la tâche en dotant tous les composants d'une méthode *isGroupe* renvoyant *false* par défaut et *true* dans le cas d'un groupe.

On peut dire que ce pattern utilise à la fois :

- l'héritage et le polymorphisme pour traiter de façon unique tous les composants (simples ou groupes) ;
- la composition pour créer une structure récursive et modifiable pendant l'exécution.

3.2 Le pattern Decorator (Décorateur)

3.2.1 Présentation

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

Supposons que l'on dispose d'une classe *Point*, munie d'une méthode *affiche* et que l'on souhaite pouvoir lui ajouter, sans avoir à la modifier, des fonctionnalités comme la gestion d'une couleur ou d'une forme (par exemple une croix au lieu d'un simple point). On veut également pouvoir combiner ces fonctionnalités, pour obtenir, par exemple, un point possédant à la fois une couleur et une forme. Ici, par souci de simplicité, nous nous contenterons d'afficher une information concernant cette couleur et/ou cette forme (comme nous le faisons pour les coordonnées des points).

Certes, on peut toujours définir des classes dérivées de *Point*, par exemple *PointColore* et *PointForme*. Mais il faudra aussi prévoir *PointColoreForme*. En cas d'ajout d'une nouvelle fonctionnalité, il faudra prévoir autant de nouvelles classes dérivées qu'il en existe déjà et la situation deviendra explosive avec l'accroissement du nombre des fonctionnalités (*n.(n+1)/2* classes pour *n* fonctionnalités).

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.

Pour mettre en œuvre ce pattern, nous allons utiliser le schéma suivant :

1. Définir une classe abstraite (nommée *Affichable*) regroupant les fonctionnalités communes à la classe *Point* et aux décorateurs. Ici, elle se limitera à la méthode *affiche* :

```
class Affichable
{ public :
    virtual void affiche () = 0 ;
} ;
```

2. Faire dériver la classe *Point* de cette classe abstraite :

```
classe Point : public Affichable
```

3. Créer une classe décorateur pour chaque fonctionnalité à ajouter, suivant ce schéma (ici pour la fonctionnalité de couleur) :

```
class Coloration : public Affichable
{ public :
    void affiche ()
    { p->affiche() ;    // affiche d'abord l'objet a decorer
       .....            // puis ce qui est lie a la couleur
    }
  private :
    Affichable *p ;     // adresse de l'objet a decorer
    .....
} ;
```

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

   La méthode *affiche* du décorateur réalise à la fois le travail à effectuer sur l'objet à décorer (*p->affiche()*), qu'elle complète par ce qui est spécifique à la fonctionnalité ajoutée par le décorateur.

Voici un exemple de programme complet :

```
// Decorator
#include <iostream>
#include <memory>      // pour unique_ptr
using namespace std ;
class Affichable
{ public :
    virtual void affiche () = 0 ;
    virtual ~Affichable() { }
} ;
```

```
class Point : public Affichable
{ public :
    Point (int abs, int ord) : x(abs), y(ord) { }
    void affiche ()
    { cout << "coordonnees = " << x << " " << y << endl ; }
  private :
    int x, y ;
} ;

class Coloration : public Affichable
{ public :
    Coloration (Affichable *pt, int cc) : p(pt), c(cc) { }
    void affiche ()
    { p->affiche(); cout << "+++ couleur " << c << endl  ;
    }
  private :
    Affichable *p ;
    int c ;
} ;
class Forme : public Affichable
{ public :
    Forme (Affichable *pt, char fc) : p(pt), f(fc) { }
    void affiche ()
    { p->affiche() ; cout << "+++ forme " << f << endl ; }
  private :
     Affichable *p ;
     char f ;
} ;
```

```
int main ()    /*************** Version pointeurs natifs *********************/
{  auto adp1 = new Point (6, 5) ;
   Affichable *pc = new Coloration (adp1 , 10) ;  // auto serait utilisable
      // ici, car pc ne change pas de type, mais il n'en ira pas toujours ainsi
   pc->affiche() ;
   auto adp2 = new Point (1, 4) ;
   Affichable *pf = new Forme (adp2, '*') ; pf->affiche () ;  // meme remarque
   Affichable *pcf = new Forme (pc, '+') ;  pcf->affiche () ; // meme remarque
   delete adp1 ; delete adp2 ;  delete pc ; delete pf ; delete pcf ;
}
```

```
int main ()    /*************** Version pointeurs intelligents ***************/
{  unique_ptr<Point> adp1 (new Point (6, 5)) ;  // avec auto, adp1 serait Point *
   unique_ptr<Affichable> pc (new Coloration (adp1.get(), 10)) ; // auto utilisable
    // ici, car pc ne change pas de type, mais il n'en ira pas toujours ainsi
   pc->affiche() ;
   unique_ptr<Point> adp2 (new Point (1, 4)) ;  // avec auto, adp2 serait Point *
   unique_ptr<Affichable> pf (new Forme (adp2.get(), '*')) ;// auto utilisable
    // ici, car pf ne change pas de type, mais il n'en ira pas toujours ainsi
   pf->affiche () ;
   unique_ptr<Affichable> pcf (new Forme (pc.get(), '+')) ; // auto utilisable
```

```
    // ici, car pcf ne change pas de type, mais il n'en ira pas toujours ainsi
   pcf->affiche () ;
}
```

```
coordonnees = 6 5
+++ couleur 10
coordonnees = 1 4
+++ forme *
coordonnees = 6 5
+++ couleur 10
+++ forme +
```

Exemple d'utilisation du pattern Decorator

On notera bien que nous avons pu appliquer un décorateur, non seulement à un point, mais aussi à un point décoré.

Remarques

1 Rien n'empêche d'appliquer plusieurs fois un même décorateur, comme dans cet exemple où nous ajoutons deux formes au même point :

```
Affichable *pBizare ;
pBizare = new Forme (new Forme (new Coloration (new Point(5, 5), 12), 'X'), '+') ;
```

L'instruction :

```
pbizare->affiche () ;
```

afficherait alors :

```
Coordonnees : 5 5
+++ couleur 12
+++ forme X
+++ forme +
```

La signification, voire la légitimité, d'une telle opération dépendra alors de l'application concernée.

2 Au lieu de :

```
Point * adp = new Point (6, 5) ;
Affichable *pc = new Coloration (adp , 10) ;
```

Nous aurions pu utiliser directement :

```
Affichable *pc = new Coloration (new Point(6, 5), 10) ;
```

Mais, il ne serait alors plus possible de détruire le point (*Point(6, 5)*) fourni au constructeur de *Coloration*, ce qui conduirait à une fuite de mémoire.

3 En pratique, il pourra être nécessaire de doter la classe d'un constructeur par recopie et d'un opérateur d'affectation, ou d'en interdire l'utilisation.

3.2.2 Discussion

Grâce à la souplesse offerte par la composition, le pattern *Decorator* permet de décorer des objets de façon individuelle, et non nécessairement tous les objets d'une même classe. Dans notre exemple, nous avons ajouté une couleur à un objet de type *Point*, et non à tous les objets de type *Point*, comme le ferait l'héritage.

Il permet d'ajouter autant de décorateurs que voulus à un objet donné. Ceux-ci peuvent se composer à volonté puisqu'un objet décoré dérive de la même classe abstraite qu'un objet à décorer. Un même décorateur peut être appliqué plusieurs fois à un même objet, pour peu que cela ait un sens dans le problème concerné.

Il est facile de mettre en place une récursivité des appels de certaines méthodes (comme nous l'avons fait pour *affiche*). Un exemple typique est celui où l'on dispose d'un produit de base (classe à décorer), possédant un certain prix, que l'on agrémente avec des compléments divers (décorateurs), possédant chacun, eux aussi, un prix : la récursivité du calcul du prix total est alors facile à mettre en œuvre.

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.

Les décorateurs sont souvent appliqués à leur construction (comme dans notre exemple), mais cela n'est pas imposé par le pattern. On pourait très bien doter les décorateurs de méthodes telles que *setCouleur* (pour *Coloration*) et *setForme* (pour *Forme*) que l'on pourrait appeler dynamiquement sur un objet implémentant *Affichable*.

Ce pattern demande généralement que, dès la conception d'une classe, on ait prévu qu'elle était susceptible de disposer de décorateurs, afin d'extraire dans une classe abstraite[7] (*Affichable*) les fonctionnalités communes aux objets à décorer et aux décorateurs. Il faut bien prendre soin de garder cette classe la plus concise possible et, en tout cas, d'y prévoir le moins de champs données possibles.

4 Les patterns comportementaux

On trouve dans cette rubrique des patterns concernés par les algorithmes et la répartitition des responsabilités entre différents objets. Quelques-uns sont des patterns de classe (patron de méthode et interpréteur), c'est-à-dire qu'ils utilisent l'héritage pour répartir le comportement entre les classes. La plupart sont des patterns d'objet, c'est-à-dire qu'ils utilisent la composition. Nous étudierons ici trois patterns : *Strategy*, *Template Method* et *Observer*.

7. Ne confondez pas cette classe avec l'éventuelle classe abstraite, mère des décorateurs (ici *Decorateur*).

4.1 Le pattern Strategy (Stratégie)

4.1.1 Présentation

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 ; le client en ignore ainsi les détails internes et n'en connaît que l'interface.

Supposez que nous disposions d'une classe *Point*, dotée d'une méthode *affiche* dont on souhaite pouvoir faire varier la manière dont elle affiche les informations coordonnées :

- soit sous une forme abrégée :

```
4 9
```

- soit sous une forme longue :

```
abscisse = 4, ordonnee = 9
```

Le pattern *Strategy* va nous permettre de découpler la classe *Point* de l'algorithme d'affichage. Pour ce faire, nous devons faire de chaque algorithme une classe (réduite ici à une seule méthode que nous nommerons *presente*), dérivant d'une classe abstraite commune (nommée ici *ModeAffichage*), suivant ce schéma :

```
class ModeAffichage
{ public :
    virtual void presente (int x, int y) = 0 ;
} ;
class AffichageCourt : public ModeAffichage
{ public :
    void presente (int x, int y)  { ..... }
} ;
class AffichageLong : public ModeAffichage
{ public :
    void presente (int x, int y) { ..... }
} ;
```

On voit alors que le choix d'une stratégie d'affichage peut se faire en créant un objet de l'un des deux types *AffichageCourt* ou *AffichageLong*. Ici, nous choisissons de communiquer la stratégie choisie au moment de la construction d'un point, en en faisant un paramètre de son constructeur. Il suffit alors que la méthode *affiche* de la classe *Point* utilise la stratégie d'affichage suivant ce schéma :

```
class Point
{ public :
    Point (int x, int y, ModeAffichage *mode) { ..... }
     .....
  private :
    ModeAffichage *mode ;   // strategie d'affichage
} ;
```

Voici un exemple de programme complet où nous faisons intervenir à la fois un point de classe automatique et un point dynamique :

```
// Strategy
#include <iostream>
#include <memory>         // pour unique_ptr
using namespace std ;
class ModeAffichage
{ public :
    virtual void presente (int x, int y) = 0 ;
    virtual ~ModeAffichage () { }    // ou, avec C++11  = default
} ;
class AffichageCourt : public ModeAffichage
{ public :
    void presente (int x, int y)
    { cout << x << " " << y << endl ; }
} ;
class AffichageLong : public ModeAffichage
{ public :
    void presente (int x, int y)
    { cout << "abscisse = " << x << " ordonnee = " << y << endl ; }
} ;
class Point
{ public :
    Point (int abs, int ord, ModeAffichage *amode) : x(abs), y(ord), mode(amode)  { }
    void affiche () { mode->presente(x, y) ; }
  private :
    int x, y ;
    ModeAffichage *mode ;   // strategie d'affichage
} ;
int main ()    /*************** Version pointeurs natifs *********************/
{ ModeAffichage *court = new AffichageCourt () ;
  Point *p1 = new Point (2, 9, court ) ; p1->affiche () ;
  Point p2 (4, 7, new AffichageLong()) ; p2.affiche () ;
  delete p1 ;
}
```

```
int main () /**** Version pointeurs natifs (pour les objets dynamiques) *****/
{ unique_ptr<ModeAffichage> court (new AffichageCourt ()) ;
  unique_ptr<Point> p1 (new Point (2, 9, court.get())) ; p1->affiche () ;
  Point p2 (4, 7, new AffichageLong()) ; p2.affiche () ;
}
```

```
2 9
abscisse = 4 ordonnee = 7
```

Exemple d'utilisation du pattern Strategy

Ici, nous avons attribué une stratégie d'affichage à un point au moment de sa construction. Il ne s'agit bien sûr pas d'une obligation. On pourrait éventuellement la définir par une méthode, voire la modifier au fil de l'exécution.

4.1.2 Discussion

Comme nous l'avons dit en introduction, ce pattern peut être utilisé pour gérer différentes variantes d'un algorithme.

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.

Toutefois, généralement, le client doit être au courant de l'existence de différents algorithmes et il doit disposer des connaissances nécessaires pour pouvoir effectuer un choix. Une exception a lieu si la stratégie est définie selon des critères « externes », par exemple un fichier de configuration.

Enfin, on notera qu'il est généralement nécessaire que le client communique certaines informations à la stratégie (valeurs de *x* et *y* dans notre exemple). Les choses peuvent être plus ou moins complexes selon l'importance de ces informations. On peut éventuellement transmettre systématiquement tout l'objet, pour peu qu'il dispose des méthodes d'accès nécessaires. On peut aussi créer un objet regroupant les seules informations voulues, ce qui demande des connaissances supplémentaires de la part du client.

4.2 Le pattern Template Method (Patron de méthode)

4.2.1 Présentation

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. Supposons par exemple que l'on souhaite définir des classes de manipulations de formes (ici, réduites à *Point* et *Cercle*) dans lesquelles une méthode *affiche* fournit, dans cet ordre :

- le type de la forme ;
- ses coordonnées ;
- éventuellement des informations complémentaires.

Nous avons là un squelette bien défini, dans lequel seule la seconde partie est indépendante de la forme concernée. Le pattern *Template Method* va nous permettre de placer ce qui est invariant dans une classe de base, en laissant le soin aux classes dérivées de fournir les variantes voulues. Pour ce faire, nous définirons une classe abstraite nommée *Forme* contenant notamment le squelette de la méthode *affiche*, par exemple :

```
class Forme
{ public :
    void affiche ()   // ici, non virtuelle
    { afficheNature () ;
      cout << "-- Coordonnees = " << x << " " << y << endl ;
      afficheAutresInfos () ;
    }
     .....
} ;
```

Les méthodes *afficheNature et afficheAutresInfos* seront déclarées abstraites dans la classe *Forme* et redéfinies dans les classes *Point* et *Cercle*, dérivées de *Forme*. Eventuellement, si cela a un sens, on pourra en définir certaines dans la classe *Forme*.

Voici un exemple de programme complet qui, ici encore, mélange des objets dynamiques et des objets automatiques :

```
// Template Method
#include <iostream>
#include <memory>
using namespace std ;
class Forme
{ public :
    void affiche ()   // ici, non virtuelle
    { afficheNature () ;
      cout << "-- Coordonnees = " << x << " " << y << endl ;
      afficheAutresInfos () ;
    }
    virtual void afficheNature () = 0 ;     // a redefinir obligatoirement
    virtual void afficheAutresInfos () { }  // version par defaut si pas redefinie
    Forme (int abs, int ord) : x(abs), y(ord) {}
    virtual ~Forme() { }                // ou, depuis C++11 = default
  protected :
    int x, y ;        // pour eviter pb acces dans classes derivees
} ;
class Point : public Forme
{ public :
    Point (int abs, int ord) : Forme (abs, ord) { }
    virtual void afficheNature () { cout << "Je suis un Point" << endl ; }
    // ici, on ne redefinit pas afficheAutresInfos
} ;
class Cercle : public Forme
{ public :
    Cercle (int abs, int ord, double ray) : Forme (abs, ord) , r(ray) { }
    virtual void afficheNature () { cout << "Je suis un Cercle" << endl ; }
    virtual void afficheAutresInfos ()  { cout << "-- Rayon = " << r << endl ; }
  private :
    double r ;
} ;
```

```
int main ()    /*************** Version pointeurs natifs *********************/
{ Forme *p = new Point(2, 5) ;  p->affiche () ;  // auto inutilisable ici
  p = new Cercle (1, 2, 5.25) ; p->affiche () ;
  Cercle c (3, 8, 4.5) ; c.affiche () ;
  delete p ;
}
```

```
int main () /*** Version pointeurs intelligents (pour les objets dynamiques) ****/
{ unique_ptr<Forme> p (new Point(2, 5)) ;  p->affiche () ;  // auto inutilisable ici
  p = unique_ptr<Forme> (new Cercle (1, 2, 5.25)) ; p->affiche () ;
  Cercle c (3, 8, 4.5) ; c.affiche () ;
}
```

```
Je suis un Point
-- Coordonnees = 2 5
Je suis un Cercle
-- Coordonnees = 1 2
-- Rayon = 5.25
Je suis un Cercle
-- Coordonnees = 3 8
-- Rayon = 4.5
```

Exemple d'application du pattern Template Méthod

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 prévue abstraite pour imposer sa redéfinition par les classes dérivées.

4.2.2 Discussion

D'une manière générale, ce pattern de classe permet de factoriser les comportements d'un algorithme communs à différentes classes d'une hiérarchie, ces dernières se contentant d'implémenter les parties variables.

Dans les méthodes appelées par l'algorithme, on distingue généralement :

- celles, comme *afficheNature*, qui doivent absolument être redéfinies par les classes concrètes. On les nomme parfois « méthodes de rappel » (ou *hook* en anglais) ;
- celles, comme *afficheAutresInfos*, dont la redéfinition est facultative et qui disposent d'une définition par défaut.

On dit parfois que ce pattern applique le principe dit d'Hollywood, à savoir que la classe de base « dit » à ses classes dérivées : « ne nous appelez pas, nous vous appelerons ».

4.3 Le pattern Observer (Observateur)

4.3.1 Présentation

Il arrive fréquemment que l'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 adresses d'objets observateurs et de les prévenir d'un changement d'état.

La démarche la plus générale consiste à définir :

- une classe abstraite servant de base aux classes des objets observateurs ; on y trouvera une méthode abstraite (nommée ici *actualise*) qui sera appelée par les objets observés en cas de changement d'état ;
- une classe abstraite servant de base aux classes des objets observés ; on y trouvera au minimum :
  - une méthode (nommée ici *enregistre*), permettant à l'objet observé d'enregistrer un objet observateur ; notez qu'un même objet pourra enregistrer différents observateurs ;
  - une méthode (nommée ici *prevenir*) qui devra appeler la méthode *actualise* de chacun des observateurs enregistrés.

Supposez que nous souhaitions pouvoir observer les modifications d'abscisses d'objets tels que des points, des cercles... Nos deux classes abstraites, que nous nommerons ici *ObservateurDAbscisses* et *AbscisseObservable* se présenteront alors suivant ce canevas :

```
class ObservateurDAbscisses
{ public :
    virtual void actualise (.....) = 0 ;   // a redefinir par chaque observateur
} ;

class AbscisseObservable
{ public :
    virtual void enregistre (ObservateurDAbscisses *obs)
    { .....     // enregistre l'adresse d'un observateur    }
    virtual void prevenir()
    { ..... }   // appelle actualise de chaque utilisateur enregistre
    .....
} ;
```

Si nous souhaitons que des objets de type *Point* soient observables, leur classe se présentera ainsi :

```
class Point : public AbscisseObservable
{ // dans les méthodes modifiant l'abscisse du point, on trouvera
  //   if (.....) prevenir() ;
  .....
} ;
```

Enfin, un programme utilisant ces fonctionnalités se présentera ainsi :

```
int main ()
{ ObservateurA obsA ;
  Point *p1 = .....
  Point *p2 = .....
    .....
  p1->enregistre (&obsA) ;  // obsA observe maintenant p1
      // si son abscisse change, la méthode actualise de obsA sera appelée
    .....
  p2->enregistre (&obsA) ;  // obsA observe maintenant p1 et p2
    .....
}
```

Il nous faut maintenant compléter ces canevas, en choisissant :

- la manière d'enregistrer les observateurs ; ici, nous utiliserons simplement un objet de type *vector <ObservateurDAbscisses *>* ;
- l'information que l'on souhaite que la méthode *actualise* fournisse à chaque observateur. Ici, nous choisirons de transmettre l'adresse de l'objet observé. Dans ces conditions, on voit que si l'on souhaite que l'observateur puisse obtenir des informations plus précises sur le changement d'état de l'objet observé, il faudra que ce dernier dispose des méthodes d'accès voulues. Ici, nous prévoirons une méthode *getX* fournissant la nouvelle abscisse et une méthode *getNom* fournissant le nom de l'objet concerné (ce nom sera fourni ici à la construction).

Nous vous proposons de définir deux classes d'observateurs :

- *ObservateurA*, un peu rudimentaire, qui se contentera simplement d'imprimer la nouvelle abscisse ;
- *ObservateurB* qui affichera en outre le nom de l'objet concerné.

Nous définirons également deux classes (dérivées de *AbscisseObservable*) d'objets observables, à savoir *Point* et *Cercle*. Par souci de simplicité, elles ne comporteront qu'une seule méthode (*deplace*), susceptible de modifier l'abscisse.

Voici un exemple de programme complet faisaint intervenir à la fois des points automatiques et des points dynamiques. Nous fournissons ici une version utilisant des pointeurs intelligents pour les objets automatiques.

```
// Observateur
#include <iostream>
#include <vector>
#include <string>
#include <memory>      // pour unique_ptr
using namespace std ;
class AbscisseObservable ;   // declaration anticipee
class ObservateurDAbscisses
{ public :
    virtual void actualise (AbscisseObservable *p) = 0 ;
    virtual ~ObservateurDAbscisses () {} ;
} ;
class AbscisseObservable
{ public :
    AbscisseObservable () : observateurs (vector<ObservateurDAbscisses *>()) {}
    virtual void enregistre (ObservateurDAbscisses *obs)
    { observateurs.push_back (obs) ;
    }
    virtual void prevenir()
    { for (unsigned int i=0 ; i<observateurs.size() ; i++)
        observateurs[i]->actualise(this) ;
    }
    virtual int getX () = 0 ;
    virtual string getNom () = 0 ;
    virtual ~AbscisseObservable () {}
  private :
    vector <ObservateurDAbscisses *> observateurs  ;
} ;
class Point : public AbscisseObservable
{ public :
    Point (string nomc, int abs, int ord) : nom(nomc), x(abs), y(ord) { }
    void deplace (int dx, int dy)
    { x += dx ; y += dy ;
      if (dx != 0) prevenir() ;
    }
    int getX () { return x ; }
    string getNom () { return nom ; }
  private :
    string nom ;
    int x, y ;
} ;
class Cercle : public AbscisseObservable
{ public :
    Cercle (string nomc, int abs, int ord, float ray)
           : nom(nomc) , x(abs), y(ord) ,rayon (ray) {}
    void deplace (int dx, int dy)
    { x += dx ; y += dy ;
      if (dx != 0) prevenir() ;
    }
```

```
    int getX () { return x ; }
    string getNom () { return nom ; }
  private :
    string nom ;
    int x, y ;
    float rayon ;
 } ;
class ObservateurA : public ObservateurDAbscisses  // affiche nouvelle abscisse
{ public :
   void actualise (AbscisseObservable *objet)
   { cout << "Nouvelle abscisse " << objet->getX() << endl ;
   }
} ;
class ObservateurB : public ObservateurDAbscisses  // affiche abscisse et nom
{ public :
   void actualise (AbscisseObservable *objet)
   { cout << "Nouvelle abscisse " << objet->getX()
          << " dans objet de nom " << objet->getNom() << endl ;
   }
} ;
```

```
int main ()    /*************** Version pointeurs natifs *********************/
{ ObservateurA obsA ; ObservateurB obsB ;
  Point *p1 = new Point("A", 3, 5) ;
  Point p2("B", 2, 2) ;
  Cercle *c = new Cercle ("C", 3, 8, 2.5f) ;
  p1->deplace (3, 9) ;      // ici, on n'est pas prevenu
  p1->enregistre (&obsA) ;  // obsA observe maintenant p1
  p1->deplace (2, 8) ;      // ici, obsA previent pour p1
  p2.deplace (3, 2) ;       //   mais pas pour p2
  p2.enregistre (&obsA) ;   // obsA observe maintenant p1 et p2
  p1->deplace (1, 8) ;      // ici, on est prevenu par obsA pour p1
  p2.deplace (2, 2) ;       //   et pour p2
  p1->enregistre (&obsB) ;  // p1 est maintenant obsrve par obsA et obsB
  p1->deplace (5, 2) ;      // on est prevenu pour p1 par obsA et obsB
  c->enregistre (&obsB) ;   // c est maintenant observe par obsB
  c->deplace (2, 2) ;       // on est prevenu par c par obsB
  delete p1 ; delete c ;
}
```

```
int main ()    /*************** Version pointeurs intelligents **************/
{ ObservateurA obsA ; ObservateurB obsB ;
  unique_ptr<Point> p1 (new Point("A", 3, 5)) ;
  Point p2 ("B", 2, 2) ;
  unique_ptr<Cercle> c (new Cercle ("C", 3, 8, 2.5f)) ;
  p1->deplace (3, 9) ;      // ici, on n'est pas prevenu
  p1->enregistre (&obsA) ;  // obsA observe maintenant p1
  p1->deplace (2, 8) ;      // ici, obsA previent pour p1
  p2.deplace (3, 2) ;       //   mais pas pour p2
  p2.enregistre (&obsA) ;   // obsA observe maintenant p1 et p2
  p1->deplace (1, 8) ;      // ici, on est prevenu par obsA pour p1
```

```
  p2.deplace (2, 2) ;       //   et pour p2
  p1->enregistre (&obsB) ;  // p1 est maintenant obsrve par obsA et obsB
  p1->deplace (5, 2) ;      // on est prevenu pour p1 par obsA et obsB
  c->enregistre (&obsB) ;   // c est maintenant observe par obsB
  c->deplace (2, 2) ;       // on est prevenu par c par obsB
}
```

```
Nouvelle abscisse 8
Nouvelle abscisse 9
Nouvelle abscisse 7
Nouvelle abscisse 14
Nouvelle abscisse 14 dans objet de nom A
Nouvelle abscisse 5 dans objet de nom C
```

Exemple d'application du pattern Observer

Remarque

Ici encore, nous avons doté nos classes *ObservateurDAbscisses* et *AbscisseObservable* de destructeurs virtuels de corps vide. Cela laisse la liberté aux classes dérivées de définir ou non un destructeur.

4.3.2 Discussion

Le canevas proposé par ce pattern est extrêmement général. Dans sa version la plus simple, il pourra servir à définir un seul observateur d'un seul objet. Mais, un même observateur pourra observer plusieurs objets (ici *obsA* observait les objets *p1* et *p2*, tous deux de type *Point*). Comme nous l'avons vu, ces objets observés pourront éventuellement être de classes différentes (*obsB* observait à la fois un point *p1* et un cercle *c*).

Réciproquement, un même objet peut être observé par plusieurs observateurs, éventuellement de classes différentes (*c* était observé par *obsA* et *obsB*). Toutefois, on notera que, dans ce cas, tous les observateurs voient leur méthode *actualise* appelée de la même manière, avec les mêmes informations.

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 dérivent d'une classe abstraite donnée (ici *AbscisseObservable*). Il lui faut quand même décider de ce qui constitue un changement d'état pour ses observateurs : dans notre exemple, nous avons volontairement prévu de ne prendre en compte que les modifications d'abscisse d'un objet.

En revanche, on notera bien qu'on ne peut observer que des objets qui ont prévu de l'être (on ne peut pas espionner un objet, à son insu !).

D'une manière générale, l'observateur dispose de deux méthodes pour obtenir des informations concernant les changements d'état des objets observés, nommées souvent protocole *push* ou protocole *pull* :

1 Soit, comme nous l'avons fait ici, il va rechercher lui-même les informations qui l'intéressent (*pull*) ; il est alors nécessaire qu'il reçoive l'adresse de l'objet concerné et que ce dernier dispose des méthodes d'accès nécessaires ;

2 Soit l'observable fournit l'information intéressant l'observateur (*push*) ; dans ce cas, le couplage est plus important qu'avec la méthode précédente, puisque l'observable doit savoir ce que les observateurs attendent de lui. Cette méthode peut s'avérer délicate si un même objet observable dispose d'observateurs de classes différentes ; par exemple, ici, il nous aurait fallu transmettre systématiquement l'abscisse et le nom à *actualise*, alors que l'une des deux classes d'observateurs n'avait pas besoin de la deuxième information.