C++ Avancé (4° Partie)

Table des matières

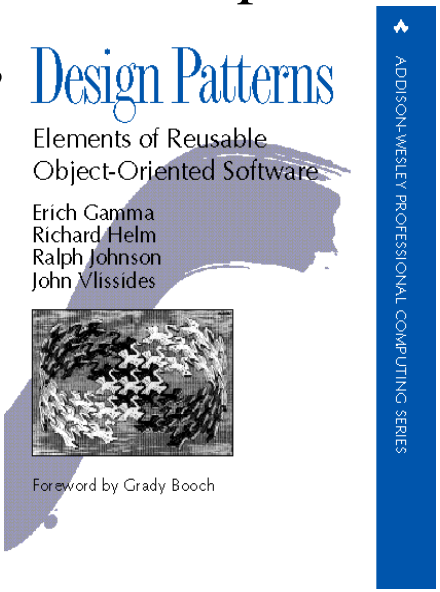
Formalisés dans le livre du « Gang of Four » (GoF, Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) intitulé « Design Patterns -- Elements of Reusable Object-Oriented Software » en 1995, les patrons de conception tirent leur origine des travaux de l'architecte Christopher Alexander dans les années 70.

Design Pattern

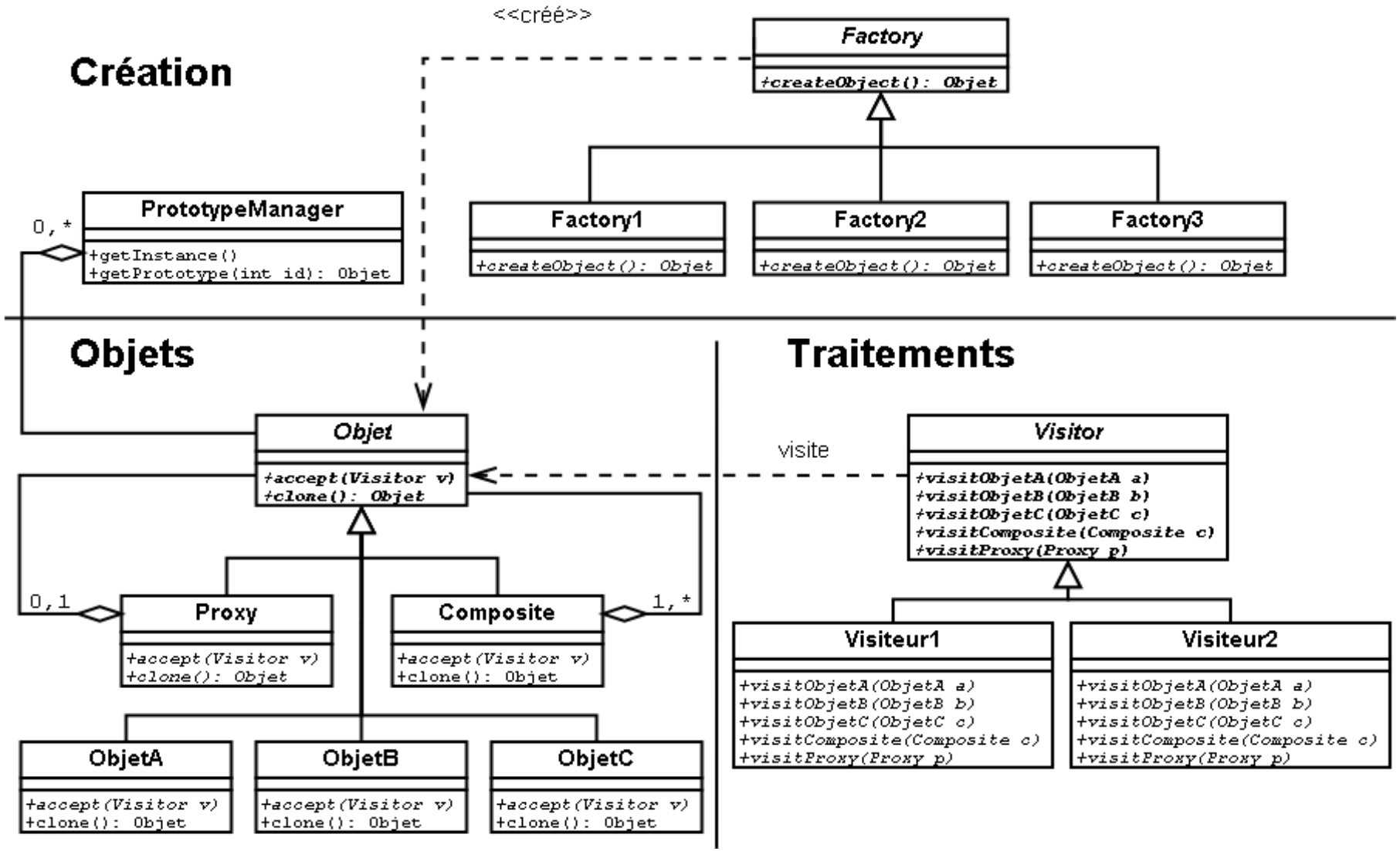
- Dans le monde de l'orienté-objet, les design patterns se présentent comme un catalogue de méthodes de résolution de problèmes récurrents.
- Pattern GoF : le concept de design patterns a été développé dans un ouvrage de référence publié en 1995 par quatre auteurs, le "Gang of Four" : Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. Il y identifiait 23 motifs de conceptions, chacun offrant une solution à un problème récurrent de la conception orientée-objet.
- Un pattern ou motif de conception est un document qui décrit une solution générale à un problème qui revient souvent.

Classification des patterns

- Création : *Abstract Factory, Builder, Factory Method, Prototype, Singleton*
- Structure : *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy*
- Comportement : *Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor*
- Autres : d'autres motifs et catégories ont été découverts après, notamment le motif MVC (Modèle Vue Contrôleur)



Séparation Création/Objets/Traitements



Description d'un pattern

- Nom et classification : par exemple observateur/comportement
- Intention : description générale et succincte
- Alias : autres noms connus pour le pattern
- Motivation : exemples qui montrent pourquoi on a besoin du pattern
- Indications d'utilisation : une liste des situations qui justifient l'utilisation du pattern
- Structure : un diagramme de classe
- Constituants : explication des différentes classes qui interviennent dans la structure du pattern
- Implémentation : les principes, pièges, astuces, techniques pour implanter le pattern dans un langage objet donné
- Utilisations remarquables : des programmes réels dans lesquels on trouve le pattern
- Limites : les limites concernant l'utilisation du pattern

Le Pattern Observateur

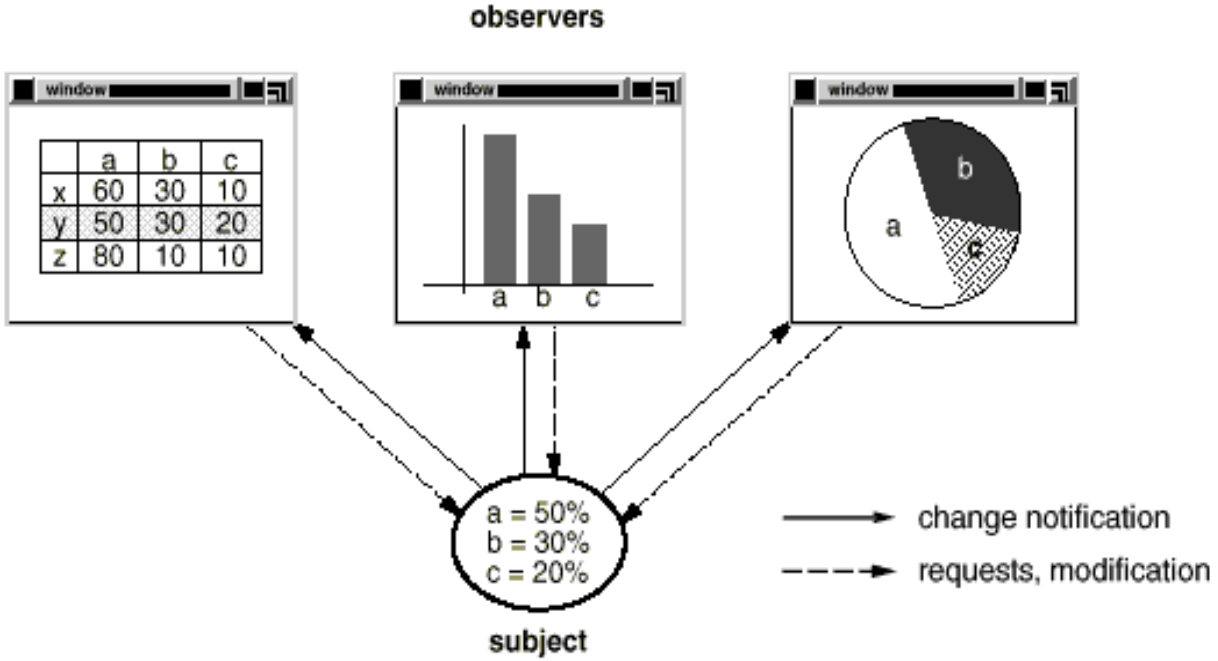
- Intention : définir une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.

- Les objets de base de ce modèle sont le **sujet** (à observer) et le (ou les) **observateur(s)**. Un sujet peut avoir un nombre quelconque d'observateurs sous sa dépendance.

- *Remarque : dans une architecture où beaucoup d'objets issus de classes différentes coopèrent il est très important que les classes soient indépendantes, pour maintenir et réutiliser le code.*

Pattern Observateur : Motivation

- Chaque fois que le sujet subit une modification de son état, il le notifie à ses observateurs (diffusion)
- En réponse, chaque observateur interrogera le sujet sur son état et réalisera son action
- Les observateur doivent souscrire pour recevoir les notifications (souscription).

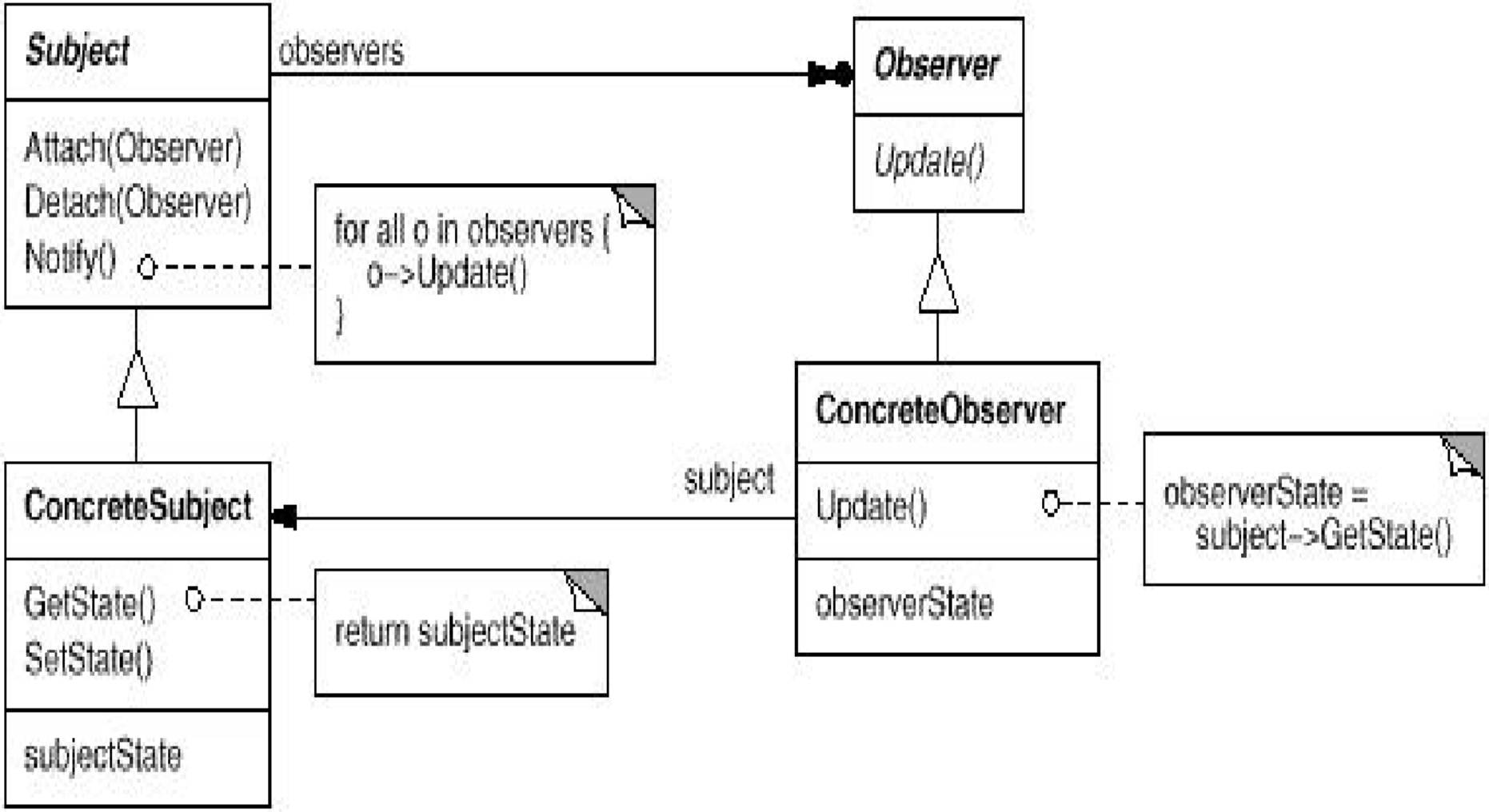


Pattern Observateur : Besoins

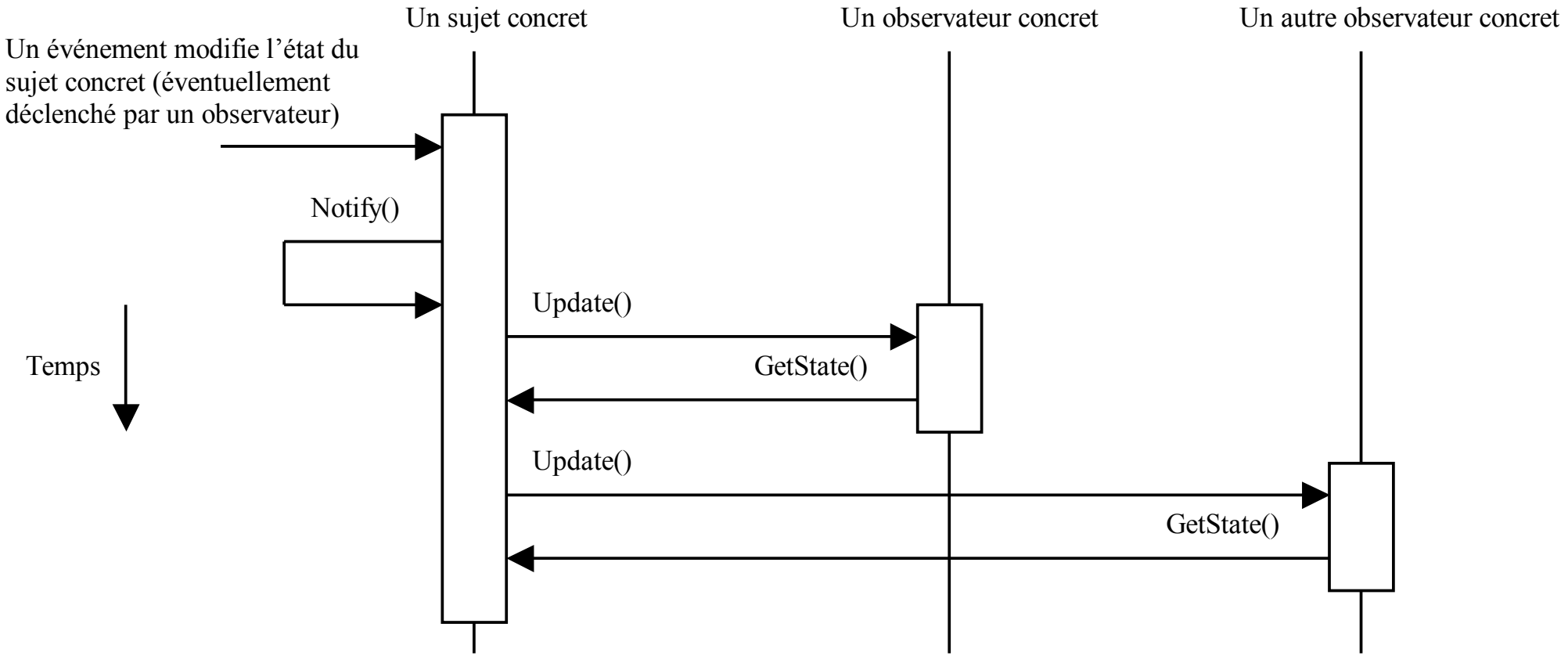
- *Quand l'appliquer :*

- Lorsqu'une abstraction possède deux aspects dont l'un dépend de l'autre : l'encapsulation de ces aspects dans des objets séparés permet de les varier et de les réutiliser indépendamment.
- Lorsqu'une modification d'un objet exige la modification des autres, et que l'on ne sait pas a priori combien d'objets devront être modifiés.
- Lorsqu'un objet devrait être capable d'informer les autres objets sans faire d'hypothèses sur ce que sont ces objets, on ne veut pas que les objets soient étroitement liés.

STRUCTURE : DIAGRAMME DE CLASSE



Pattern Observateur : Collaboration



L'implémentation (1/6)

- Une classe de base abstraite, CObservateur, dont dériveront les observateurs. Cette classe présente une méthode virtuelle pure Update(Sujet) qui est appelée lorsque le sujet change.

```
class CObservateur
{
public:
 // la classe étant polymorphe, le destructeur doit être virtuel
 virtual ~CObservateur(){}
 virtual void Update(CSujet * pSujet)=0;
};
```

L'implémentation (2/6)

- La classe de base, CSujet, dont dériveront les différentes classes de sujet à observer, maintient une liste d'observateurs. Les méthodes Attach() et Detach() permettent d'abonner ou de désabonner un observateur. La méthode Notify() est appelée lorsque l'état du sujet a changé et avertit les observateurs du changement en appelant la méthode CObservateur::Update(this) de chaque observateur.

```
typedef list<CObservateur *> TListObs;
class CSujet
{public:
   virtual ~CSujet(){}
   virtual void Attach(CObservateur *);
   virtual void Detach(CObservateur *);
   virtual void Notify();
 protected:
   CSujet(){} // empêche cette classe de base d'être instanciée autrement que par une classe dérivée
 private:
   TListObs m_ListObs; };
```

L'implémentation (3/6)


```
void CSujet::Attach(CObservateur * pObserver)
{ m_ListObs.push_front(pObserver); }

void CSujet::Detach(CObservateur * pObserver)
{  TListObs::iterator it = find(m_ListObs.begin(),
m_ListObs.end(), pObserver );
   if( it != m_ListObs.end() )
   m_ListObs.erase(it);
}

void CSujet::Notify()
{  TListObs::iterator it;
   for(it = m_ListObs.begin();it != m_ListObs.end();it++)
      (*it)->Update(this);
}
```

L'implémentation (4/6)

- Une classe concrète CSujetConcret qui dérive de CSujet et qui présente des méthodes pour récupérer l'état du sujet concret.

```
class CSujetConcret : public CSujet
{
public:
   CSujetConcret():m_Val(0) { }
   void do() { Notify(); }
   long getEtat() { return m_Val; }
private:
   long m_Val;
};
```

L'implémentation (5/6)

- Une classe concrète CObservateurConcret par type d'observateur, qui dérive de CObservateur. Cette classe abonne l'observateur concret au sujet concret dans le constructeur et le désabonne dans le destructeur. De plus elle implémente la méthode Update(Sujet) qui récupère l'état du sujet concret.

```
class CObservateurConcret : public CObservateur
{ public:
   CObservateurConcret(CSujetConcret * pSujetConcret);
   ~CObservateurConcret();
   virtual void Update(CSujet * pSujet);
  private:
   CSujetConcret * m_pSujetConcret;
};
```

L'implémentation (6/6)


```
CObservateurConcret::CObservateurConcret(CSujetConcret *pSujetConcret
):m_pSujetConcret( pSujetConcret )
{
   m_pSujetConcret->Attach(this);
}

CObservateurConcret::~CObservateurConcret()
{
   m_pSujetConcret->Detach(this);
}

void CObservateurConcret::Update(CSujet * pSujet)
{  if(pSujet == m_pSujetConcret)
   {
      // TODO
      // m_pSujetConcret->getEtat();
   }
}
```

Les conséquences

- Un sujet sait seulement qu'il a une liste d'observateur qui présente la fonction Update(). Le sujet n'a aucune connaissance des classes concrètes d'observateurs.
- Le sujet n'a pas à se préoccuper de quel observateur a besoin de quelle notification. Le sujet est responsable seulement d'appeler la fonction Notify() à chaque changement. Libre aux observateurs de tenir compte ou non du changement.
- Néanmoins on peut notez les faiblesses suivantes :
  - Les observateurs n'ayant aucune connaissances mutuelles, un changement peut provoquer une "lourde" cascade d'update.
  - A moins de raffiner le code, l'observateur ne sait pas exactement la nature de la modification du sujet.

Le "Tirer/Pousser"

- Il existe deux manières de récupérer l'information dans la méthode Update :
  - Une première consiste à "tirer" l'information de l'objet en appelant une méthode qui va nous donner l'information.
  - La deuxième consiste au contraire à "pousser". C'est l'objet observé qui va pousser l'information jusqu'à l'observateur.
- On adopte souvent la méthode « tirer » qui est plus simple et permet plus de modularité.

Bibliographie

- Cours et tutoriels : http://conception.developpez.com/cours/#dp
- Présentation : http://fr.wikipedia.org/wiki/Patron_de_conception
- Design Patterns, Elements of Reusable Object-Oriented Software de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Ed. ADDISON-WES-LEY - ISBN: 0-201-63361-2