*Chapitre IX*

# Patrons de conception

## Sommaire

# Introduction

Les patrons de conception sont des modèles d'organisation de classes permettant d'obtenir certains comportements ou certaines propriétés souhaitées.

Ils constituent des modèles ou des exemples dont vous pouvez vous inspirer.

Ils sont divisés en trois catégories :

- les **patrons de créations** sont des modèles orientés vers la création ou la configuration d'objets.
- les **patrons de structure** sont des modèles qui minimisent les dépendances entre l'implémentation concréte et l'utilisation.
- les **patrons de comportement** sont des modèles pour les comportements entre classes ou les interactions entre classes.

# 1 Patrons de création

Les patrons de créations sont des modèles de création et de configuration d'objet.

Ils sont composés des modèles suivants :

- **singleton** : pour construire une classe dont il ne peut y avoir au plus qu'une seule instance.
- **prototype** : pour construire le clone d'un objet concret à partir de sa représentation abstraite. celui-ci est manipulé sous sa forme polymorphique.
- **fabrique** : pour utiliser une classe abstraite de construction (=fabrique) afin de créer une famille d'objets abstraits (une famille = partage la même interface).
- **fabrique abstraite** : même idée, mais cette fois la fabrique peut créer des familles d'objets abstraits ; chaque famille d'objets abstraits ayant les mêmes interfaces.
- **monteur** : pour construire des familles d'objets complexes associé à un type concret, chaque famille étant définie par la façon de construire concrètement un objet.
- **singleton** : pour définir une classe dont il ne peut exister qu'un seul objet de ce type (*i.e.* toutes les instances créées constituent un seul et même objet).

## 1.1 Fabrique (ou Factory)

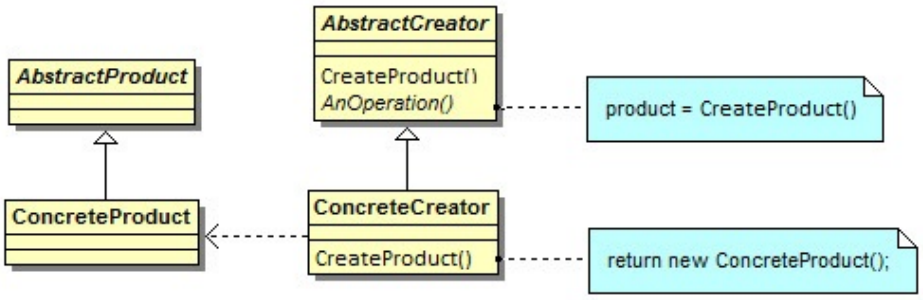
**But :** définit une interface pour créer un objet en laissant aux sous-classes quelles seront les classes instanciées.

**Motivation :** Une fabrique laisse l'instanciation d'une classe aux sous-classes.

**Applications :**

- une classe ne peut anticiper la classe des objets qu'elle doit créer.
- une classe souhaite que ses sous-classes spécifient les objets à créer.
- une classe délègue des responsabilités à une ou plusieurs de ses sous-classes.

**Diagramme UML :**




**Participants :**

- `Product` : définit l'interface des objets que la fabrique crée.
- `ConcreteProduct` : implémente l'interface du `Product`.
- `AbstractCreator` : déclare la méthode de construction qui retourne un objet de type Product.
  - ◇ peut également définir une implémentation du constructeur par défaut qui retourne un objet par défaut.
  - ◇ peut appeler la méthode de construction pour créer un objet `Product`.
- `ConcreteCreator` : surcharge de la méthode de construction et retourne une instance du produit concret.

**Notes :**

- L'`AbstractCreator` se base sur ses sous-classes pour définir les méthodes de construction tel qu'il retourne une instance du `ConcreteProduct` appropriée.
- Elimine le besoin d'attacher au code des classes spécifiques à une application. Le code manipule seulement l'interface du `Product`.

**Fonctionnement :**

- utilise BuildProduct pour créer le produit.
- utilise l'interface abstraite du Produit créé pour le manipuler.

**Implémentation :**

- l'AbstractCreator peut fournir une implémentation par défaut du Creator.
- l'AbstractCreator peut créer différents types de produit : le type est passé en paramètre à la création.
- il est possible d'utiliser des templates pour éviter de multiplier les sous-classe.
- idée : convention de nommage d'une méthode `Factory: Class* DoMakeClass(?)`.

**Exemple :**

```
// produit abstrait: interface virtuelle
class AbstractProduct { virtual void do_this() = 0; };
```

```
// implémentation concrète du produit
class ConcreteProduct: public AbstractProduct {
    void do_this() { ... }
};
```

```
// interface virtuelle de la fabrique
class AbstractCreator {
  virtual AbstractProduct* CreateProduct() = 0;
};
```

```
// implémentation concrète de la fabrique
class ConcreteCreator : public AbstractCreator {
public:
  AbstractProduct *CreateProduct() { ... }
};
```

```
// utilisation: création de la fabrique
AbstractCreator  *creator = new ConcreteCreator;
// utilisation de la fabrique pour créer un produit
AbstractProduct *product = creator->CreateProduct();
```

**Exemple :** implémentation utilisant un `AbstractCreator` basé sur un template.

```
// produit abstrait: interface virtuelle
class AbstractProduct { virtual void do_this() = 0; };
```

```
// sous-classe sous forme de template
template <class AbstractObject> class AbstractCreator {
  // implémentation du constructeur de la sous-classes template
  template <typename CreateProduct>
    AbstractObject *CreateProduct() {
      return new ConcreteProduct();
  }
};
```

```
// implémentation concrète du produit
class ConcreteProduct : public AbstractProduct {
  void do_this() { ... }
};
```

```
// instanciation d'AbstractFactory pour créer une fabrique
// d'AbstractProduct
AbstractCreator<AbstractProduct>  creator;
```

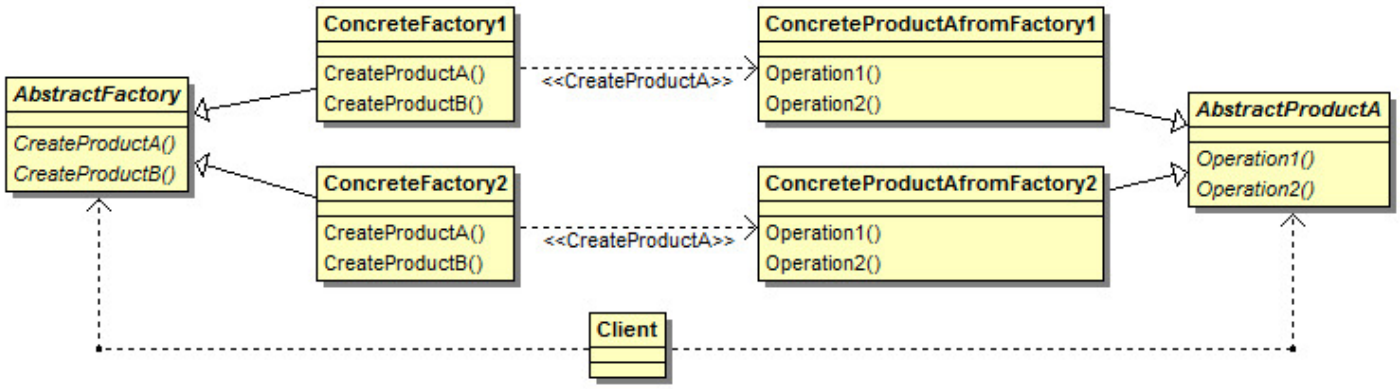
```
// instanciation du Create de la fabrique afin de créer un
// ConcreteProduct
AbstractProduct *p = creator.CreateProduct<ConcreteProduct>();
```

## 1.2 Fabrique abstraite (ou Abstract Factory)

**But :** fournit une interface pour créer des familles d'objets (dépendant ou ayant des liens entre eux) sans fournir leurs classes concrètes.

**Motivation :** disposer d'un constructeur de classes virtuelles produisant des objets concrets avec des appels normalisés, le constructeur virtuel s'instanciant en fonction du type d'objets concrets nécessaires.

**Diagramme UML :**




**Participants :**

- `AbstractFactory` : interface (=classe virtuelle) de création d'objets abstraits.
- `ConcreteFactory` : classe concrète qui implémente l'interface de l'`AbstractFactory` pour créer les Produits concrets.
- `AbstractProduct` : interface d'utilisation des Produits abstraits.
- `ConcreteProduct` : classe concrète qui implémente l'interface de l'`AbstractProduct` pour créer un Produit abstrait.
- `Client` : fait appel à l'`AbstractFactory` pour créer les Produits abstraits, et fait appel à l'interface des produits Abstrait pour accéder aux fonctionnalités implémentés par les Produits concrets.

**Remarques**

- Une seule instance de la `ConcreteFactory` est créée au run-time (= fabrique pour une famille particulière).
- Par contre, plusiers `AbstractFactory` peuvent être utilisée chacun faisant référence à une `ConcreteFactory` différentes (= fabriques pour plusieurs familles différentes).

**Conséquences**

- elle isole les classes concrètes (`Factory` et `Product`) à travers des interfaces abstraites.
- permet l'ajout simple de nouvelles implémentations concrètes.
- attention à n'utiliser ensemble que des `Products` issus de la même `Factory`.
- le support de nouveau produit peut être difficile s'il conduit à l'extension des interfaces (demande la mise à jour de tous les `Products`).

### Implémentations

- un seule instance d'une `Factory` nécessaire, donc le plus souvent implémentées comme `Singleton`.
- une seule `ConcreteFactory` par classe de `ConcreteProduct`.
- s'il y a beaucoup de `ConcreteFactory`, l'implémenter éventuellement comme un `Prototype`.

### Exemple :

```
// Interface implémentée par le produit abstrait A
class AbstractProductA {
public:
  virtual void DoThis(...) = 0;
  virtual void DoThat(...) = 0;
};
```

```
// Implémentation concrète du produit concret A
// pour la fabrique concrète 1
#ifdef __platform1__
class ConcreteProductAFrom1 : public AbstractProductA {
public:
  ConcreteProductAFrom1(...) { /* implementation */ };
  void DoThis(...) { /* implementation */ };
  void DoThat(...) { /* implementation */ };
};
#endif
```

```
// Fabrique abstraite
// Interface de constructions de produits abstraits
class AbstractFactory {
public:
  virtual AbstractProductA *BuildProductA(...) = 0;
  /* ici: autres constructions de produits abtraits */
};
```

```
// Implémentation de la fabrique concrète 1
#ifdef __platform1__
class ConcreteFactory1 : public AbstractFactory {
public:
    ConcreteFactory1(...) { /* implementation */ };
    // construction de l'objet abstrait A
    // dans ce cas: objet concret A de la fabrique 1
    virtual AbstractProductA *BuildProductA(...) {
        return new ConcreteProductAFrom1(...);
    };
    /* reste  de l'implementation */
};
#endif
```

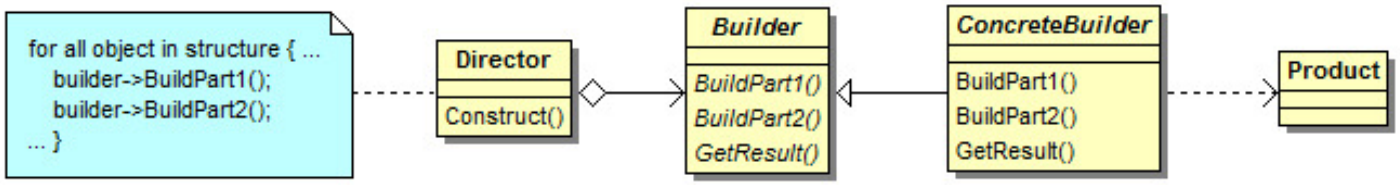
```
#ifdef __anotherplatform__
  // definition et implementation de:
  // - une autre ConcreteFactory
  // - des autres ConcreteProducts associés
#endif
```

```
// pointeur vers la fabrique abstraite
AbstractFactory        *pAF = NULL;

// quelque part dans le code: définition de la fabrique
#ifdef __platform1__
pAF = new ConcreteFactory1();
#endif

// construction d'un produit A à partir de la fabrique
AbstractProductA    *pPA = pAF->BuildProductA(...);

// utilisation du produit
pPA->DoThis(...);
pPA->DoThat(...);
```

## 1.3 Monteur (ou Builder)

**But :** séparer la construction d'un objet complexe de sa représentation afin que le même processus de construction puisse créer différentes représentations.

**Diagramme UML :**




**Conséquences :** permet de changer la représentation interne d'un produit.

- le Builder fournit au Director une interface abstraite pour construire le Product.
- l'interface permet au Builder de masquer la représentation et la structure interne du Product.
- l'objet créé reste associé au Builder concret qui l'a créé.

**Applications :**

- L'algorithme, pour créer un objet complexe, doit être indépendant des parties qui constituent l'objet et de la façon dont ils sont assemblés.
- Le processus de construction doit autoriser différentes représentations pour l'objet qui est construit.

**Participants :**

- `Builder` : interface abstraite pour créer les parties d'un objet concret.
- `ConcreteBuilder` :
  - ◇ construit et assemble les parties du produit en implémentant l'interface du Builder.
  - ◇ définit et garde la trace des représentations qu'il crée.
  - ◇ fournit un interface pour obtenir le produit.
- `Director` : construit l'interface en utilisant le Builder.
- `Product` : objet complexe en construction.

**Exemple :**

```
// object concret à construire
class ConcreteProduct {
public: // pour l'exemple
  PartA   *partA;
  PartB   *partB;
};
```

```
// constructeur abstrait de l'objet concret
class AbstractBuilder {
    // dernière représentation créée
    ConcreteProduct  *products;
public:
  virtual PartA* BuildPartA() = 0;
  virtual PartB* BuildPartB() = 0;
  ConcreteProduct *GetResult()  { return products; };
};
```

```
// constructeur concret: implémentation
class ConcreteBuilder : public AbstractBuilder {
public:
  PartA*  BuildPartA() {
    PartA *part = new PartA(/* args */);
    // suite construction
    return part;
  }
  PartB*  BuildPartB() {
    PartB *part = new PartB(/* args */);
    // suite construction
    return part;
  }
  ConcreteProduct *GetResult() { return product;  }
};
```

```
// constructeur concret: autres implémentations
...
```

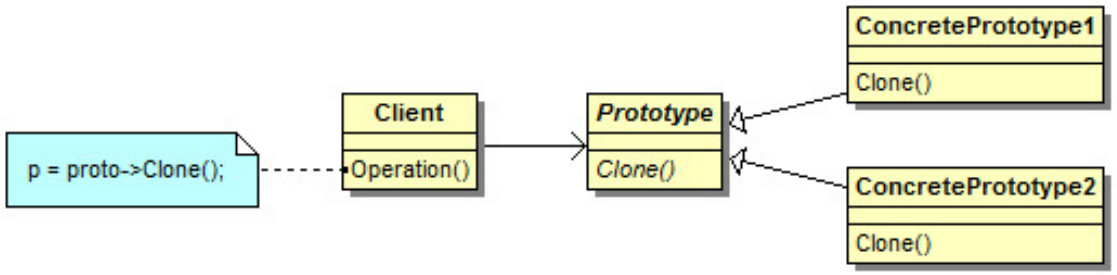
```
// directeur: constructeur de ConcreteObject à partir
// d'un constructeur abstrait
class Director {
  AbstractBuilder   *builder;
public:
  Director(AbstractBuilder *b) : builder(b) {}
  void Construct() {
    ConcreteProduct *product = new ConcreteProduct;
    product->partA = builder->BuildPartA();
    product->partB = builder->BuildPartB();
    builder->product = product;
  }
  ConcreteProduct *GetResult()
        { return builder->GetResult(); }
};
```

```
// exemple d'utilisation
ConcreteBuilder *aConcreteBuilder = new ConcreteBuilder1;
Director    director(aConcreteBuilder);
director.Construct();
ConcreteObject *object = director.GetResult();
// alternative pour obtenir le résultat
ConcreteObject *object = aConcreteBuilder->GetResult();
```

## 1.4 Prototype

**But :** création de nouveaux objets par duplication d'objets existants (= les prototypes) qui disposent d'une capacité de clonage.

**Diagramme UML :**




**Application :**

- quand le système doit être indépendant de la façon dont ses produits sont créés, composés et représentés (et)
- (ou) quand les classes sont spécifiques à l'exécution (changement dynamique)
- (ou) pour éviter la construction d'une hiérarchie de Factory en parallèle de la hiérarchie de Product.
- (ou) quand l'instance d'une classe peut avoir une parmi quelques combinaisons d'état. Il peut être alors plus pratique d'installer un petit nombre de prototypes et de les cloner, plutôt qu'instancier manuellement toutes les classes avec l'état approprié.

**Participants :**

- `Prototype` : interface permettant à un objet de se cloner.
- `ConcretePrototype` : implémentation de l'opération de clonage.
- `Client` : crée un nouvel objet qui demandant à un prototype de se cloner.

**Conséquences :**

- cache le produit concret au client, et donc réduit le nombre de nom que le client connait.
- permet de travailler avec des classes spécifiques à l'application sans modification.
- ajout/suppression de produit à l'exécution : incorporer le nouveau produit en enregistrant une instance du prototype au client.

**Implémentation :**

- utiliser un manager de prototype lorsque le nombre de prototypes n'est pas dixé, et garder un registre des prototypes disponibles (non géré par le client).
- l'implémentation du clonage peut être délicate (surtout lorsqu'il y a des références circulaires).
- problème sur le sens du mot clonage :

  - ◇ "shallow copy" = les instances des variables sont partagées entre le clone et sa copie (= par référence, par pointeur, ...).
  - ◇ "deep copy" = cloner les instances de toutes les variables.
- problème avec l'initialisation de l'état si l'interface de clonage est uniforme à travers tous les objets.

**Exemple :**

```
// objet abstrait
class Prototype {
  virtual Prototype* Clone() = 0;
};
```

```
// objet concret clonable 1
class ConcretePrototype1 : public Prototype {
  Prototype* Clone() {
    return this; // self
  }
};
```

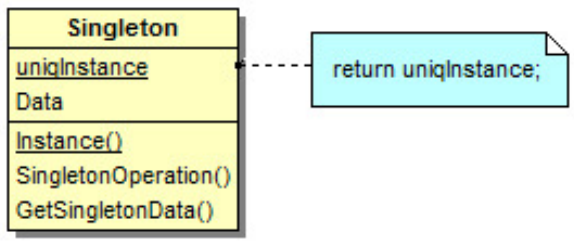
```
// objet concret clonable 2
class ConcretePrototype2 : public Prototype {
  Prototype* Clone() {
    return new ConcretePrototype2(*this); // copie
  }
};
```

```
// exemple d'application (client)
Prototype  *p1 = new ConcretePrototype2();
// clone d'un prototype abstrait
Prototype  *p2 = p1->Clone();
```

## 1.5 Singleton

**Intérêt :** s'assurer qu'une classe a au plus une instance et y fournit un point d'accès global.

**Diagramme UML :**




**Applications :**

- il doit y avoir exactement une instance et elle doit être accessible aux clients depuis un point d'accès connu.
- quand une instance unique doit être extensible à travers des sous-classes et que les clients doivent être capable d'utiliser l'instance étendue sans modifier leur code.

**Participants :**

- définit une opération d'instance et laisse le client accéder à l'instance unique. C'est une opération de classe (= un membre statique en C++).
- peut être responsable de la création de sa propre instance unique.

**Conséquences :**

- accès contrôlé à l'instance unique (car le singleton l'encapsule).
- espace de nommage réduit : amélioration par rapport aux variables globales.
- peut être dérivé à des sous-classes (partage naturel du singleton).
- permet un nombre variable d'instances.
- plus flexible qu'une opération de classe (membre statique). Permet le partage entre classe et le polymorphisme.

**Implémentation :**

1. assurer l'instance unique : cacher les opérations qui crées des instances derrière des opérations de classe (=membre statique et méthode d'une classe). Donc définition du constructeur par copie et operator= comme des opérations privées.
2. en C++, il est insuffisant de définir le singleton comme un objet global ou statique, et se reposer sur une initialisation automatique :
   - on ne peut garantir qu'une seule instance sera jamais déclarée.
   - l'instanciation d'un singleton à l'initialisation n'est pas nécessairement possible (éventuellement besoin de paramètres/objets créés à l'exécution).
   - il n'est pas possible de définir l'ordre des initialisations statiques. Donc, surtout pas de dépendances entre singletons subissant des initialisations statiques.

**Motivation :**

la seule façon de s'assurer qu'un objet n'a qu'une seule instance, c'est d'en rendre responsable la classe en interceptant les demandes de création des nouveaux objets. Elle donne aussi accès au singleton. Elle peut être en charge de la création.

**Problème de l'implémentation C++**

L'implémentation utilisant des membres statiques est celle qui s'impose.

Elle pose néanmoins des problèmes :

- il n'est pas possible de garantir qu'au moins une instance de l'objet static sera déclarée.
- le singleton n'a pas nécessairement assez d'information pour s'instancier à l'initialisation du code (ce que permettent les champs statiques).
- il n'est pas possible de définir l'ordre d'initialisation des objets globaux (dont statique). En conséquence, il n'est pas possible de créer des dépendances entre singletons (sous peine d'erreur).

**Exemple :**

```
class ConcreteObject { ... };
```

```
class Singleton {
public: using type = shared_ptr<ConcreteObject>;
private: static type d;
public:
  Singleton() {}
  static type getInstance() {
    if (!d) d.reset(new ConcreteObject);
    return d;
  }
  ~Singleton() {}
};
Singleton::type Singleton::d;
```

```
// client
Singleton::type  x = Singleton::getInstance();
```

**Attention :**

- l'implémentation ci-dessus n'est thread-safe (cf `reset`).
- il est très difficile de créer une classe Singleton qui va fonctionner dans tous les cas de figure (voir "Singleton Pattern : A review and analysis of existing C++ implementations" sur CodeProject ; noter que cette article est antérieur au $C_{11}^{++}$).

# 2 Patron de structure (Structural Patterns)

Les patrons de structure sont des modèles d'organisation de classes permettant de minimiser les dépendances entre l'implémentation (concrète) et l'utilisation (interface).

- **Adaptateur** : permet d'adapter une interface existante à une autre interface.
- **Pont** : utilise une interface abstraite à la place d'une implémentation spécifique, et ce qui permet de rendre indépendante l'interface abstraite utilisée de son implémentation concrète.
- **Objet composite** : permet de manipuler un objet composite avec la même interface que ses composants (= structure hiérarchique).
- **Décorateur** : attache dynamiquement de nouveaux comportements ou responsabilités à un objet (alternative à l'héritage pour la composition de fonctionnalités).
- **Façade** : occulte une conception ou un ensemble d'interfaces complexes d'un sous-système en fournissant une interface simple.
- **Poids-mouche** : si de nombreux petits objets similaires doivent être manipulés mais qu'il serait trop coûteux de les instancier tous, permet de partager leurs parties communes.
- **Proxy** : permet de substituer une classe (plus flexible) à une autre classe utilisant la même interface afin d'en contrôler l'accès.

## 2.1 Adaptateur (Adapter ou Wrapper)

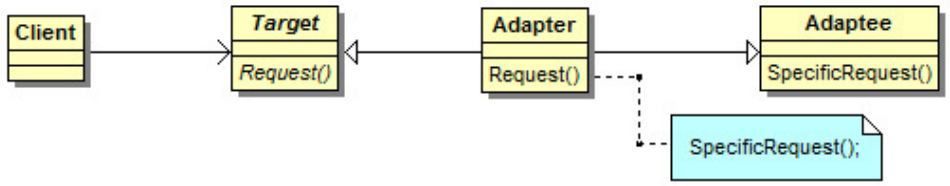
**Intérêt :**

- convertit l'interface d'une classe d'un objet existant en une autre interface attendue par le client.

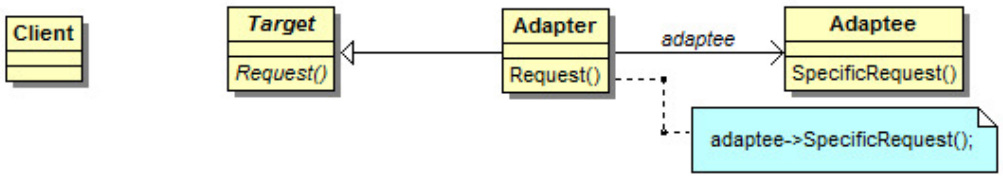
- permet aux classe de travailler ensemble malgré des interfaces initialement incompatible.

**Diagramme UML :**

Adapteur de classe




Adapteur d'objet




**Application :**

- utiliser une classe existante dont l'interface ne correspond pas à ce qui est attend.
- créer une classe réutilisable qui coopère avec plusieurs autres classes ayant des interfaces différentes.
- utilisation de plusieurs sous-classes lorsqu'il n'est pas pratique d'adapter toutes les interfaces. L'adaptateur d'objet peut adapter l'interface de sa classe parent.

**Participants :**

- `Target` : interface (virtuelle) utilisée par le client.
- `Client` : collabore avec les objets à travers de l'interface Target.
- `Adaptee` : interface existante qui a besoin d'une adaptation.
- `Adapter` : adapte l'interface de l'Adapter à l'interface Target.

**Conséquences :** 2 types d'Adapter

**Adaptateur de classe :**

- l'adaptation se fait en s'attachant à une classe Adapter concrète. Donc, l'adaptation de classe ne fonctionnera pas lorsque l'on veut adapter la classe et toutes ses sous-classe.
- permet à l'Adapter de modifier/surcharger certains comportements, puisque l'Adapter est une sous-classe de l'Adaptee.
- introduit seulement un objet, et aucune indirection pour obtenir l'Adaptee.

**Adaptateur d'objet :**

- permet à un seul Adapter de fonctionner pour tous les sous-classes de l'Adaptee. Des fonctionnalités supplémentaires peuvent être ajoutée à l'Adaptee.
- il est plus difficile de modifier/surcharger le comportement de l'Adaptee : nécessite de sous-classer l'Adaptee et que l'Adapter fasse référence à la sous-classe plutôt qu'à l'Adaptee.

On peut avoir besoin d'Adapter multi-voie si l'on a besoin d'adaptation différente du même objet pour différents clients.

**Implémentation :** L'Adapter hérite de façon publique de Target, et de façon privée de l'Adaptee.

**Exemple :**

```
// objet à adapter
class Adaptee {  void SpecificRequest() { ..; }  };
```

```
// interface seulement
class Target {  virtual void Request() = 0; }
```

```
class ClassAdapter : public Target, public Adaptee {
public: ClassAdapter() : Adaptee() {}
        void Request() { SpecificRequest(); }
};
```

```
class ObjectAdapter : public Target {
private:  Adaptee   *adaptee;
public:   ObjectAdapter() : adaptee(new Adaptee) {}
          void Request() { adaptee->SpecificRequest(); }
};
```

```
// client pour l'adapteur de classe
ClassAdapter   adapter;
adapter.Request();
```

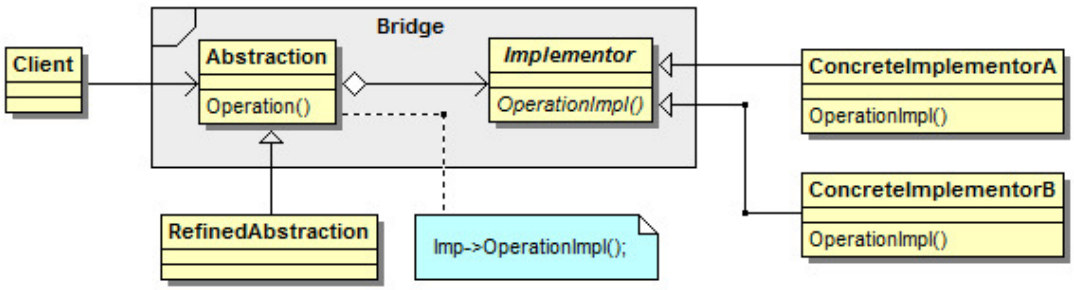
```
// client pour l'adapteur d'objet
ObjectAdapter  adapter;
adapter.Request();
```

## 2.2 Bridge (Handle/Body)

**Intérêt :** découple une abstraction de son implémentation, telle qu'elle puisse varier indépendamment.

- une abstraction peut avoir des implémentations différentes.
- rendre le code client indépendant de la plateforme.

**Diagramme UML :**




**Application :**

- permet un lien permanent entre une abstraction et son implémentation (choix de l'implémentation au runtime)
- l'abstraction et l'implémentation doivent être extensibles par sous-classes.
- les changements d'implémentation d'une abstraction ne doivent pas avoir d'impact sur le Client.
- cache complètement l'implémentation d'une abstraction au Client.
- partage une implémentation à travers des objets multiples, et ce fait doit être caché au Client.

**Participants :**

- `Abstraction` :
  - ◇ définit l'interface de l'abstraction, et les opérations de haut-niveau basée sur les primitive de l'Implementor.
  - ◇ maintient une référence à un objet de type Implementor.
- `RefinedAbstraction` : étend l'interface définie par l'abstraction.
- `Implementor` :
  - ◇ définit l'interface pour les classes implémentées.
  - ◇ ne définit que des opérations primitives.
- `ConcreteImplementor` : implémente l'interface Implementor et définit son implémentation concrète.

**Conséquences :**

1. Découplage de l'interface et de l'implémentation :
   - l'implémentation n'est plus attachée de façon permanente à l'interface.
   - l'implémentation d'une abstraction peut être configurée au run-time (voir chargée).
   - découple l'`Abstractor` de l'`Implementor` élimine les dépendance de l'implémentation à la compilation : encourage les couches et conduit à un système mieux structuré.
2. L'extension des hiérarchies de l'`Abstractor` et de l'`Implementor` peut se faire indépendamment.
3. Permet de cacher les détails de l'implémentation au client.

**Implémentation :** un seul `Implementor` : lorsqu'il n'y a qu'un seul `Implementor`, la classe abstraite `Implementor` n'est plus nécessaire (relation 1 :1 entre Abstraction et Implementor). Devient similaire à un Adapter.

**Exemple :**

```
// interface (abstraite) de l'implémentation
class Implementor {
    virtual void OperationImpl(Abstraction*) = 0;
};
```

```
// première implémentation possible
class ConcreteImplementorA : public Implementor {
  void OperationImpl(Abstraction *x) { ... }
};
// deuxième implémentation possible
class ConcreteImplementorB : public Implementor {
    void OperationImpl(Abstraction *x) { ... }
};
```

```
// abstraction de l'implémentation
class Abstraction {
private: Implementor   *impl = nullptr;
public:  Abstraction(Implementor *i) : impl(i) {}
         void Operation() { impl->OperationImpl(this); }
};
```

```
class RefinedAbstraction : public Abstraction {
  // extension implémentation (si nécessaire)
};
```

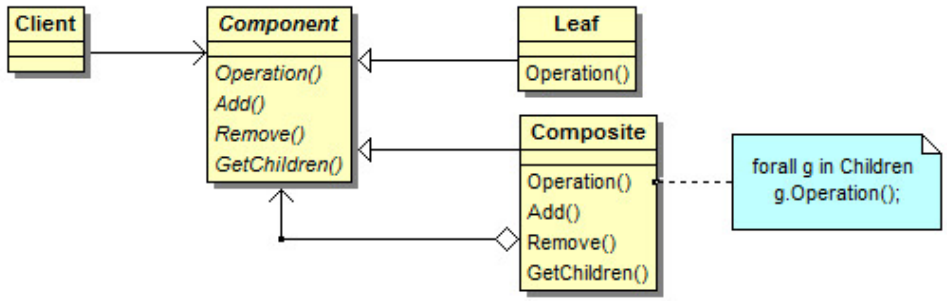
```
// client : choix de l'implémentation de l'abstraction
Abstraction   abstr(new ConcreteImplementorA);
// utilisation de l'abstraction de l'implémentation
abstr.Operation();
```

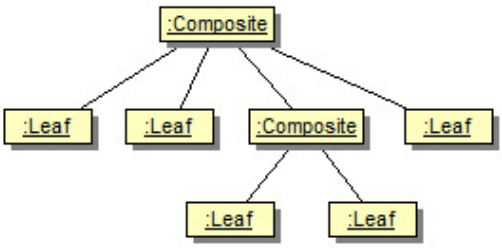
## 2.3 Objet composite (Composite)

**Intérêt :**

- compose des objets en une structure d'arbre pour représenter une partie ou la totalité d'une hiérarchie
- permet au client de considérer des objets individuels et les compositions d'objets uniformément.

**Diagramme UML :**




**Application :**

- permet de représenter une partie ou la totalité d'une hiérarchie d'objets.
- permet d'ignorer la différence entre une composition d'objets et objets individuels.

**Participants :**

- `Component` :
  - ◇ déclare l'interface des objets dans la composition.
  - ◇ implémente le comportement par défaut de l'interface commun à toutes les classes.
  - ◇ déclare une interface pour accéder et gérer les composants fils
  - ◇ (optionnel) définit une interface pour accéder au composant parent dans la structure récursive et l'implémente au besoin.

- `Leaf` :
  - ◇ représente un objet feuille (=sans fils) dans le Composite.
  - ◇ définit le comportement pour les objets primitifs dans le Composite.
- `Composite` :
  - ◇ définit le comportement des composants qui sont des fils.
  - ◇ stocke les composants fils.
  - ◇ implémente les opérations relatives aux fils dans l'interface Component.
- `Client` : manipule les objets de la composition à travers l'interface Component.

**Implémentation :**

- **Référence explicite au parent :** simplifie la traversée d'une structure composite (+ remontée, + suppression). Placer la référence au parent dans la classe Component.
- **Partage des composants :** difficile si un composant peut avoir plus d'un parent. Oblige le fils à stocker des parents multiples ce qui peut conduire à des ambigüités.
- **Minimisation de l'interface Component :** on intérêt à définir le plus d'opérations communes à Leaf/Composite dans Component (sinon la différence entre objet individuel/composition d'objet devient significative). L'interface Component peut fournir une implémentation par défaut, surchargée par Leaf/Composite.
- il est préférable de faire en sorte que Add/Remove échoue par défaut. Ainsi, si le Component ne peut pas avoir d'enfant, le cas est automatiquement géré (échec par défaut).
- Possibilité de mettre dans un cache les opérations les plus courantes sur les fils (traversée, recherche de valeur, ?). Charger un Composant invalide alors le cache de la partie de la hiérarchie concernée.
- **Destruction des Components :** le mieux et d'en rendre responsable le Composite. Exception : les objets (immutables) partagés.

**Exemple :**

```
// interface commune
class Component {
public: virtual void traverse() = 0;
};
```

```
class Leaf : public Component {
  LeafStorage    data;  // contenu des feuilles
public:
  // implémentation interface commune
  void traverse() { cout << data; }
};
```

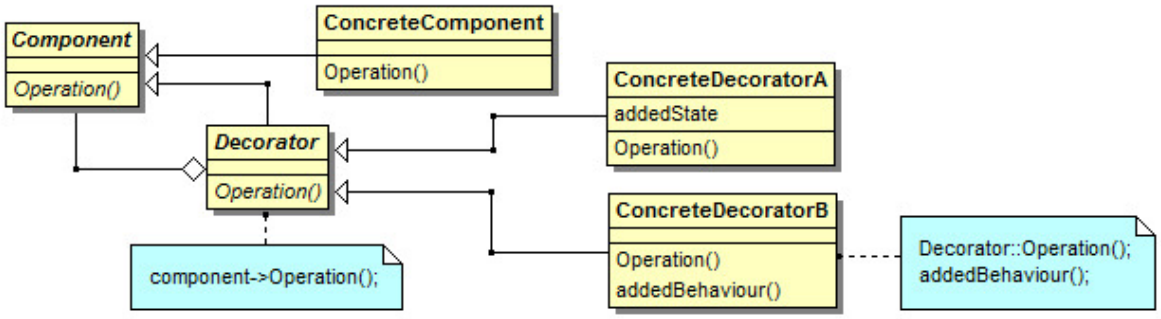
```
class Composite : public Component {
  vector<Component*>   children;  // contenu des noeuds
public:
  // spécialisation sur les noeuds
  void Add(Component *elt) { children.push_back(elt); }
  // implémentation interface commune
  void traverse()
        { for(Component *c : children) c->traverse(); }
};
```

## 2.4 Décorateur (Decorator)

**Intérêt :**

- attache dynamiquement des responsabilités supplémentaires à un objet.
- alternative flexible à la création de sous-classes pour étendre les fonctionnalités.

**Diagramme UML :**




**Application :**

- ajoute de nouvelles fonctionnalités à un objet individuel de manière dynamique et transparente (i.e. sans affecter les autres objets).
- ces fonctionnalités peuvent être retirées.
- quand une extension par subclassing n'est pas pratique :
  - ◇ un grand nombre d'extensions indépendantes sont possibles et provoqueraient une explosion combinatoire.
  - ◇ la définition de la classe est cachée ou n'est pas disponible.

**Participants :**

- `Component` : définit l'interface qui doit être ajoutée dynamiquement aux objets.
- `ConcreteComponent` : définit l'objet auquel l'interface (avec les nouvelles fonctionnalités) doit être attachée.
- `Decorator` : maintient une référence à un objet `Component` et définit une interface qui est conforme à l'interface du `Component`.
- `ConcreteDecorator` : ajoute les fonctionnalités à un Component.

**Conséquences :**

- plus flexible que l'héritage statique :
  - ◇ les fonctionnalités peuvent être ajoutées/supprimées à l'exécution simplement en les attachant/détachant.
  - ◇ fournir plusieurs Decorators pour un même Component permet de mélanger et d'ajuster les fonctionnalités.
  - ◇ la fonctionnalité peut être héritée plusieurs fois.
- permet de définir une classe simple et d'ajouter de manière incrémentale les fonctionnalités avec des Decorators. Evite à une application d'avoir des fonctionnalités qu'elle n'utilise pas (énorme classe qui fait tout < classe simple + Décorators).
- du point de vue de l'identité d'un objet, un Component décoré est différent du Component seul. Donc, ne pas se base sur l'identité d'un objet lorsqu'un Decorator est utilisé. à tendance à générer de nombreux petits objets qui se ressemblent beaucoup (les objets ne diffèrent que

par la façon dont ils sont connectés, mais pas par leurs classes ou la valeur de leurs variables).

**Implémentation :**

- l'interface du Decorator doit se plier à l'interface du Component qu'il décore. Le Concrete-Decorator dérive en général d'une classe commune.
- si l'on ajoute qu'une seule fonctionnalité, le Decorator abstrait n'est pas utile (fusion du Decorator et du ConcreteDecorator).
- pour obtenir une interface compatible, Components et Decorators doivent dériver d'une classe Component commune la plus légère possible (i.e. une interface et non des données).

**Exemple :**

```
// interface commune
class Component { virtual void Operation() = 0; };
```

```
// composant concret
class ConcreteComponentA : public Component {
  // état pour cette implémentation
public:
  // implémentation concrete de l'interface commune
  void Operation() { }
  // méthodes supplémentaires
};
```

```
class ConcreteComponentB; // même idée
```

```
//décorateur
class Decorator : public Component {
  // vers le composant
  Component   *comp;
public:
  Decorator(Component *c) : comp(c) {}
  // délégation au composant
  void Operation() { comp->Operation(); }
};
```

```
// décorateur concret
class ConcreteDecoratorA : public Decorator {
public:
  ConcreteDecoratorA(Component *w) : Decorator(w) {}
  void Operation() {
    Decorator::Operation();
    // complément du décorateur A
  }
};
```

```
class ConcreteDecoratorB; // même idée
class ConcreteDecoratorC; // même idée
```

```
// utilisation par le client
// composant A sans décoration
Component *comp1 = new ConcreteComponentA();
// composant A avec décoration C
Component *comp2
    = new ConcreteDecoratorC(new ConcreteComponentA());
// composant A avec décoration A,B et C
Component *comp3 = new ConcreteDecoratorA(
    new ConcreteDecoratorB(
      new ConcreteDecoratorC(
        new ConcreteComponentB())));
comp3->Operation();
```

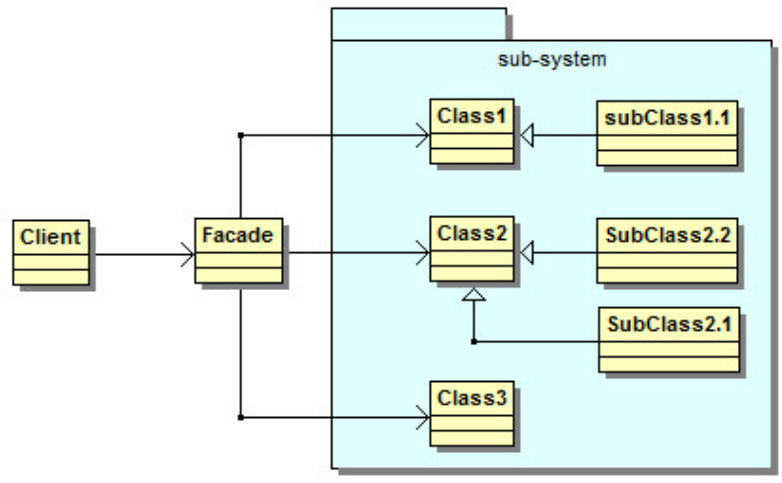
**Interprétation :** `Operation()` sur le `Component`

- `comp1` utilise l'implémentation concrète de `ConcreteComponentA`.
- `comp2` utilise l'implémentation concrète de `ConcreteComponentA`, à laquelle est ajoutée le complément défini dans le décorateur `ConcreteDecoratorC`.
- `comp3` utilise l'implémentation concrète de `ConcreteComponentB`, à laquelle sont ajoutés successivement les compléments défini dans les décorateur `ConcreteDecoratorA`, `ConcreteDecoratorB` et `ConcreteDecoratorC`.

## 2.5 Façade (Facade)

**Intérêt :** fournir une interface unifiée pour définir une interface dans un sous-système (interface de plus haut niveau rendant le sous-système plus facile à utiliser).

**Diagramme UML :**




**Application :**

- pour fournir une interface simple à un système complexe : rend le sous-système plus simple à utiliser et réutilisable. Peut être une vue par défaut suffisante pour la plupart des clients.
- pour réduire le nombre de dépendance entre un client et une abstraction. La façade découple le sous-système du client, le rendant plus indépendant et portable.

- pour structurer les sous-systèmes. Une façade est le point d'entrée de chaque sous-système. S'il y a des dépendances entre sous-systèmes, ils peuvent communiquer à travers leur façade.

**Participants :**

- `Façade` :
  - ◇ connait quelles classes du sous-système est responsable de la requête.
  - ◇ délègue les requêtes des clients aux objets appropriés du sous-système.
- `Subsystem classes` :
  - ◇ implémentes les fonctionnalités du sous-système.
  - ◇ traite le travail assigné par l'objet façade.
  - ◇ n'a pas connaissance de la façade (aucune référence).

**Collaboration :**

- Le client envoie les requêtes à la façade qui les fait suivre aux sous-systèmes appropriés.
- Si le sous-système effectue le travail, la façade peut avoir besoin de traduire son interface en l'interface du sous-système.
- Les clients qui utilisent la façade n'ont pas accès directement aux objets du sous-système.

**Conséquences :**

- protège le client des composants du sous-système, rendant ce dernier plus facile à utiliser.
- favorise le couplage faible entre le client et le sous-système. Le sous-système peut donc évoluer sans affecter le client.
  même remarque lorsque l'on utilise des façades entre sous-systèmes.
  aide à réduire les dépendances et simplifie les partages.
- n'empêche pas les clients d'utiliser directement le sous-système.

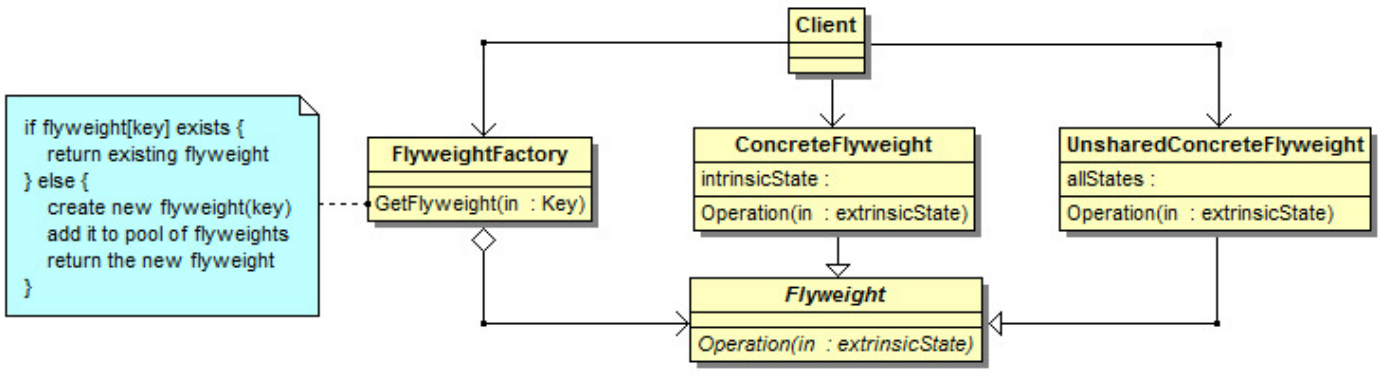
**Implémentation :**

- réduction du couplage client-sous système : faire de la façade une classe abstraite avec des sous-classes concrètes pour chacune des implémentations associées au sous-système. Le client communique avec le sous-système à travers l'interface abstraite de la Façade.
- alternative : configurer la façade avec différents objets du sous-système.

## 2.6 Poids-mouche (Flyweight)

**Intérêt :** pour partager efficacement un grand nombre d'objets à grain fin.

**Diagramme UML :**



**Application :** lorsque l'ensemble des conditions suivantes sont vérifiées :

- il y a un grand nombre d'objets.
- les coûts de stockage sont importants en raison de la grande quantité d'objets.
- la plupart des états des objets peuvent être extrinsèque (=externe)
- beaucoup de groupes d'objets peuvent être remplacés par relativement peu d'objets partagés une fois que l'état extrinsèque a été enlevé.
- l'application ne dépend pas de l'identité des objets (en raison du partage de l'identité par ce pattern).

**Participants :**

- `Flyweight` : déclare l'interface à travers laquelle les `Flyweight`s peuvent recevoir et agir sur les états extrinsèques.
- `ConcreteFlyweight` : implémente l'interface `Flyweight` et ajoute le stockage pour les états intrinsèques. Il doit être partageable. Chaque état stocké est intrinsèque et doit être indépendant du contexte de l'objet.
- `UnsharedConcreteFlyweight` : toutes les sous-classes du `Flyweight` n'ont pas besoin d'être partagée. L'interface `Flyweight` autorise le partage mais ne le rend pas obligatoire.
- `FlyweightFactory` : crée et gère les `Flyweight`s.
  s'assure que les `Flyweight`s sont partagés correctement. Quand un client effectue une requête de `FLyweight`, la `FlyweightFactory` fournit une instance existante ou en construit une si aucune n'existe.
- Client : maintient une référence au Flyweight ; calcule et stocke les états extrinsèques du `Flyweight`.

**Collaboration :**

- l'état qu'un Flyweight a besoin pour fonctionner doit être caractérisé soit comme extrinsèque ou intrinsèque.
  - ◇ état intrinsèque : stocké dans un ConcreteFlyweight.
  - ◇ état extrinsèque : stocké ou calculé par l'objet Client. Le client passe son état Flyweight lors des appels.
- Un client ne devrait pas instancier un ConcreteFlyweight directement. Il devrait obtenir les objets ConcreteFlyweight exclusivement depuis le FlyweightFactory pour assurer qu'ils sont partagés correctement.

**Conséquences :**

- les Flyweights peuvent introduire des coûts d'exécution associées aux transferts, à la recherche et/ou le calcul des états extrinsèques. Ces coûts sont compensés par le gain d'espace (partagé).
- le gain de stockage est fonction :
  - ◇ de la réduction du nombre d'instances dues au partage,
  - ◇ du volume d'état intrinsèque par objet,

  et dépend si les états extrinsèques sont stockés ou calculés.
- structure hiérarchique/graphe : les noeufs/feuilles ne peuvent pas stocker un pointeur vers son parent. Le pointeur parent est passé comme Flyweight comme une partie de son état extrinsèque.

**Implémentation :**

- élimination des états extrinsèques : idéalement, l'état extrinsèque peut être calculé à partie d'une structure séparée.
- gestion des objets partages : les objets étant partagés, ils ne doivent pas être instanciées directement.
  Le `FlyweightFactory` permet au client de trouver un `Flyweight` particulier, et utilise pour ce faire un stockage associatif.
- le partage implique aussi une forme de comptage de référence ou de garbage collector pour récupérer le stockage des `Flyweight`s non référencés.

**Exemple :**

```
class Flyweight {
public: virtual size_t getId() = 0;
        virtual void Operation() = 0;
};
```

```
class ConcreteFlyweight : public Flyweight {
private: size_t  key;
         // autres états internes
public: ConcreteFlyweight(size_t x) : key(x) {}
        size_t getId() { return key; }
        void Operation() { ... }
};
```

```
class UnsharedConcreteFlyweight : public Flyweight {
public: size_t getId() { return 0; }
        void Operation() { ... }
};
```

```
class FlyweightFactory {
private:
  // ensemble des objets concrets existants
  static vector<Flyweight*>   objects;
public:
  static Flyweight *getFlyweight(size_t key) {
    if (key == 0) return nullptr; // unshared Flyweight
    for(Flyweight *f : objects)
      if (f->getId() == key) return f;
    objects.push_back(new ConcreteFlyweight(key));
    return objects.back();
  }
};
```

**Remarque :** il serait préférable d'utiliser un `vector<shared_ptr<Flyweight> >` afin de garder trace des références acquises sur les `Flyweight`s :

- Le garbage collector devient alors intégré (l'objet alloué est détruit automatiquement lorsque le compteur de référence arrive à 0),
- On peut alors recycler les places libérées dans le vecteur si une nouvelle allocation est nécessaire.
- La clé ici est fournie, mais il s'agit plus naturellement de clé de Hashage.

**Exemple :** (suite)

```
// code du client
// utilisation du FlyweightFactory
// pour créer/récupérer le Flyweight
Flyweight  *f1 = FlyweightFactory::getFlyweight(1);
```

```
// extrinsic state pour appel à l'opération
int   state = 4;
f1->Operation(state);
```

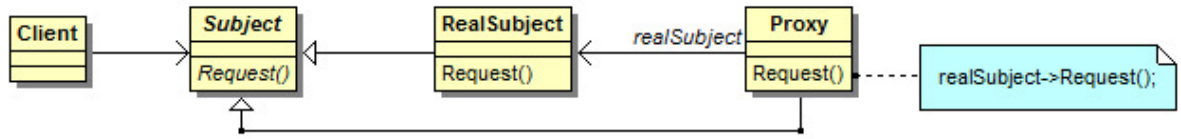
```
// création d'autres Flyweight
Flyweight  *f2 = FlyweightFactory::getFlyweight(13);
// ...
Flyweight  *fp = FlyweightFactory::getFlyweight(1);
// noter que ici: fp = f1
```

```
// Flyweight non partagé
Flyweight  *fu = new UnsharedConcreteFlyweight();
```

## 2.7 Proxy (Proxy ou Surrogate)

**Intérêt :** objet qui se substitue à un autre objet et qui en contrôle l'accès. L'objet qui effectue la substitution possède la même interface ce qui rend rend la substitution transparente.

**Diagramme UML :**




**Application :** utilisable à chaque fois qu'il y a besoin d'une référence à un objet plus flexible et sophistiqué qu'un simple pointeur.

- représentant local d'un objet dans un espace d'adressage différent.
- proxy virtuel qui crée des objets coûteux sur demande.
- proxy de protection d'accès à un objet original.
- références intelligentes comme remplacement d'un pointeur nu qui effectue des opérations supplémentaires lors de l'accès à un objet (comptage du nombre de références, chargement d'un objet persistant en mémoire la première fois que l'on y fait référence, vérification que l'objet réel est verrouillé avant d'y accéder pour s'assurer qu'aucun autre objet peut le changer.

**Participants :**

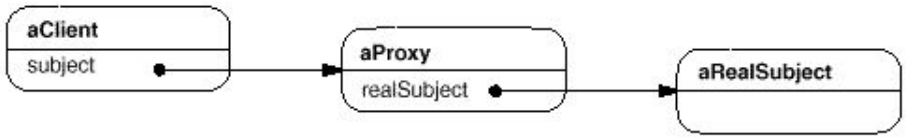
- `Proxy` :
  - ◇ maintient une référence qui permet au sujet réel d'accéder au `RealSubject`.
  - ◇ fournit une interface identique à celle du sujet afin que le Proxy puisse se substituer au `RealSubject`.
  - ◇ contrôle l'accès au `RealSubject` et peut être responsable de sa création et de sa destruction (voir les alternative ci-dessous).
- `Subject` : définit l'interface commune pour le `RealSubject` et le `Proxy` tel que le `Proxy` puisse être utilisé partout où le `RealSubject` est attendu.

- `RealSubject` : définit l'objet réel que le `Proxy` représente.

**Autres possibilités dépendant du type de `Proxy` :**

- **remote proxy** : encodage d'une requête et envoie au `RealSubject` dans un espace d'adressage différent.
- **virtual proxy** : cache d'information sur le `RealSubject` afin d'en différer l'accès.
- **protection proxy** : vérifie que l'appelant a les permissions d'accès requises pour effectuer la requête.

**Collaboration :** Le `Proxy` fait suivre les requêtes au `RealSubject` lorsque cela est approprié (dépendant du type de proxy).




**Conséquences :** le proxy introduit un niveau d'indirection au `RealSubject` lorsque l'on accède au sujet.

- un **remote proxy** peut cacher le fait qu'un objet réside dans un espace d'adressage différent.
- un **virtual proxy** peut effectuer des optimisations telles que créer des objets à la demande.
- un **protection proxy** ou une référence intelligente autorise des taches de nettoyage lors de l'accès à un objet.
- **copy-on-write** : sorte de proxy qui n'effectue la copie que si l'objet pointé est modifié (nécessite aussi un compteur de référence sur l'objet).

**Exemple :**

```
// interface du sujet
class Subject {
public: virtual int Request() = 0;
};
```

```
class RealSubject : public Subject {
private: int data;  // données internes
public: int Request() { return data; };
    RealSubject(int x) : data(x) {}
};
```

```
// proxy intermédiaire
class Proxy : public Subject {
private: RealSubject *subject;
public:
  Proxy(int x) : subject(new RealSubject(x)) {}
  int Request() {
        // contrôle d'accès
        return subject->Request();
    };
};
```

```
// code client: accès au sujet à travers le proxy
Subject  *ptr = new Proxy(2);
int x = ptr->Request();
```

# 3 Patrons de comportement (Behavioral Patterns)

Les patrons de comportement sont des modèles d'organisation de classes permettant de résoudre des problèmes liés aux comportements ou à l'interaction entre les classes.

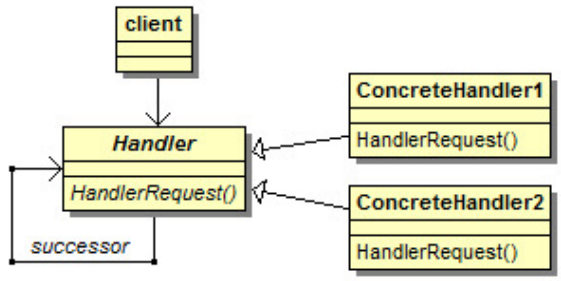
Les patrons de comportement sont les suivants :

- **Chaine de responsabilité** : construit une chaîne de traitements destinée à gérer une requête.
- **Commande** : encapsule l'invocation d'une suite de commandes.
- **Interpréteur** : représente la grammaire d'un langage spécialisé et interprète les expressions issues de ce langage.
- **Itérateur** : parcours séquentiel des éléments d'un objet agrégé sans exposer sa structure interne.
- **Médiateur** : utilise une classe (médiateur) comme intermédiaire de communication entre un groupe de classes afin de réduire leurs dépendances.
- **Mémento** : capture et externalise l'état interne d'un objet dans le but de pouvoir le sauvegarder et le restaurer.
- **Observateur** : définit une dépendance entre objets tels que lorsqu'un objet change d'état, tous ceux dont il dépend soient avertis et mis-à-jour.
- **État** : associe des comportements à l'état de manière à ce que lorsque l'état change, les comportements associés puissent changer.
- **Stratégie** : permet de changer de dynamiquement de stratégie (=d'algorithme).
- **Patron de méthode** : définit un modèle de méthode qui utilise des méthodes primitives abstraites pour son implémentation, permettant ainsi de modifier la méthode.
- **Visiteur** : découple classes et traitements, afin de pouvoir ajouter de nouveaux traitements utilisant les méthodes internes de la classe sans ajouter de nouvelles méthodes aux classes existantes.

## 3.1 Chaîne de responsabilité (Chain of responsability)

**Intérêt :** évite de coupler l'expéditeur d'une requête à son récepteur en donnant, à plus qu'un objet, une chance de traiter une requête. On enchaine les objets récepteurs et on passe la requête le long de la chaine jusqu'à ce qu'un objet le gère.

**Diagramme UML :**




**Application :**

- lorsque plus d'un objet peut traiter une requête, et que l'objet qui traite la requête (handler) n'est pas connu a priori. Le handler doit être choisi automatiquement.
- une requête doit être envoyée à un ensemble d'objet sans spécifier son récepteur.

- l'ensemble des objets qui peuvent gérer une requête doit être spécifié dynamiquement.

**Participants :**

- `Handler`
  - ◇ définit l'interface pour traiter les requêtes.
  - ◇ (optionnel) implémente le lien vers le successeur.
- `ConcreteHandler`
  - ◇ traite la requête dont il est responsable.
  - ◇ peut accéder à son successeur.
  - ◇ si le `ConcreteHandler` peut traiter la requête, il le fait. Sinon, il la fait suivre à son successeur.
- `Client` : initie la requête à un `ConcreteHandler` de la chaine.

**Collaboration :** Lorsqu'un client effectue un requête, la requête se propage à travers la chaine jusqu'à ce qu'un `ConcreteHandler` prend la responsabilité de la traiter.

**Conséquences :**

- couplage réduit : libère l'objet de connaitre quel autre objet gère la requête (seulement qu'elle sera traitée de façon appropriée). Le récepteur et l'émetteur n'ont aucune connaissance l'un de l'autre, et un objet dans la chaine n'a pas ç connaitre la structure de la chaine.
- flexibilité supplémentaire dans l'assignation des responsabilités aux objets : l'ajout et le changement de responsabilité peut se faire à l'exécution.
- la réception n'est pas garantie : puisque la requête n'a pas de récepteur explicite. La requête peut aboutir en fin de chaine sans avoir été traitée. Même chose si la chaine n'a pas été configurée correctement.

**Implémentation :**

- **implémentation de la chaine des successeurs :**
  il y a 2 méthodes :
  - ◇ définir de nouveaux liens (dans le `Handler`, mais le `ConcreteHandler` peut également le définir)
  - ◇ utiliser des liens existants (s'ils supportent la chaine existante).
- **connexion des successeurs :**
  s'il n'y a pas de référence préexistante pour définir la chaine, elles doivent être introduite manuellement. Le `Handler` est alors utilisé pour gérer les requêtes et manipuler la chaine des successeurs.
- **représenter les requêtes :** il peut être utile de définir une classe de requêtes afin de les standardiser et de les étendre.

**Exemple :**

```
class Handler {
private: Handler *successor;
public:
  virtual void HandleRequest(int i) {
    if (successor) successor->HandleRequest(i);
    else throw unhandled_request;
  };
  Handler() : successor(nullptr) {}
  void AddHandler(Handler *h)
      { h->successor = successor; successor = h; }
};
```

```
class ConcreteHandler1 : public Handler {
private: int mask;
public:
  void HandleRequest(int i) {
    if (mask & i) { /* handling */ };
    else Handler::HandleRequest(i); // passing
  }
  ConcreteHandler1(int m) : Handler(), mask(m) {}
};
```

```
class ConcreteHandler2; // similaire ConcreteHandler1
class ConcreteHandler3; // similaire ConcreteHandler1
```

```
// code client
// construction du handler
Handler *h = new ConcreteHandler1(1);
h->AddHandler(new ConcreteHandler2(4));
h->AddHandler(new ConcreteHandler3(2));
```

```
// Handling
h->HandleRequest(3);
// handler1: succès
```

```
h->HandleRequest(4);
// handler1: échec, handler3: succès
```

```
h->HandleRequest(8);
// handler1: échec, handler3: échec, handler2: échec, throw
```

**Notes sur l'implémentation ci-dessus :**

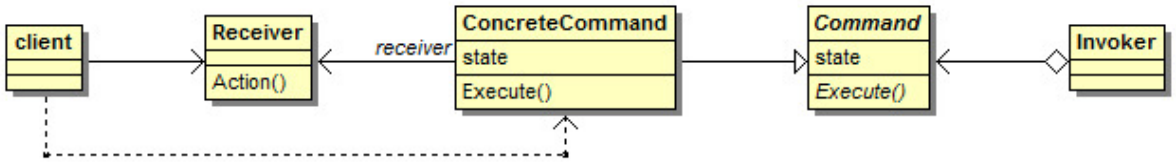
- l'insertion se fait en seconde position pour simplifier le code.
- la requête passée (entier) et le traitement (correspondant avec un masque binaire) sont évidemment uniquement à titre d'exemple.

## 3.2 Commande (Command)

**Intérêt :**

- encapsule une requête comme un objet, permettant de paramétrer le client avec des requêtes différentes, des files d'attente, des requêtes de log, ... et supporte l'annulation de commande.
- permet d'effectuer des requêtes à des objets sans rien connaître des opérations demandées ou du destinataire de la requête.

**Diagramme UML :**




**Application :**

- paramétre un objet par les actions à traiter = remplaçant orienté-objet des Callbacks.
- spécifier, placer dans la file d'attente, exécuter des requêtes à des instants différents (durée de vie indépendante de la requête qui l'a engendrée).
- supporte l'annulation : la commande d'exécution peut stocker des états pour rendre réversible les effets de la commande elle-même (traversée des listes d'exécution avant/arrière avec Execute/Unexecute).
- supporte le log des changements (possibilité de conserver un log persistant des changements). Application : récupération d'un crash en chargeant les commandes du disque et en les réexécutant.
- structurer un système autour d'opérations haut-niveaux construites sur des opérations primitives (transaction).

**Participants :**

- `Command` : déclare une interface pour exécuter une opération.
- `ConcreteCommand` :
  - ◇ définit le lien entre un `Receiver` et une action.
  - ◇ implémente Execute en invoquant les opérations correspondantes sur le `Receiver`.
- `Client` : crée une `ConcreteCommand` et l'affecte à son `Receiver`.
- `Invoker` : demande à la commande de traiter la requête.
- `Receiver` : sait comment traiter les opérations associées avec l'exécution d'une requête. Toute classe peut être utilisée comme `Receiver`.

**Collaboration :**

- le client crée une `ConcreteCommand` et spécifie son `Receiver`.
- l'`Invoker` stocke l'objet `ConcreteCommand`.
- l'`Invoker` envoie une requête en appellant `Execute` sur la commande. Si la commande est annulable, la `ConcreteCommand` stocke l'état courant avant d'appeler `Execute`.
- la `ConcreteCommand` invoque l'operation sur son `Receiver` pour exécuter la requête.

**Conséquences :**

- découple l'objet qui invoque de l'opération de celle qui sait comment l'effectuer.
- les commandes peuvent être manipulée et étendue comme tous les autres objets.
- les commandes peuvent être assemblées comme une commande composite.
- il est facile d'ajouter des commandes.

**Implémentation :**

- Intelligence des commandes :
  - ◇ définit un lien entre le Receiver et les actions qui traitent la requête.

  - ◇ implémente tout sans rien déléguer au Receiver (utile pour définir des commandes indépendantes des classes existantes quand aucun Receiver n'existe, ou quand la commande connaît explicitement son Receiver).
  - ◇ entre les 2 extrêmes précédents : la commande ayant assez de connaissance pour trouver dynamiquement son Receiver.
- Annulation des commandes : besoin de stocker l'objet Receiver, les arguments de l'opération, ...
- Utiliser les templates C++ : pour les commandes qui ne sont pas annulable et qui n'ont pas besoin d'argument, il est possible d'utiliser des templates pour éviter de créer une sous-classe de commande pour chaque type d'action et de Receiver.

**Exemple :**

```
// interface Command
class Command {
public: virtual void Execute() = 0;
};
```

```
class Receiver {
private:
    // stockage des commandes
  vector<Command*> commands;
public:
    // ajout d'une commande au receiver
  void AddCommand(Command *c) { commands.push_back(c); }
    // execution de toutes les commandes du receiver
  void Action() { for (Command *c : commands) c->Execute(); }
};
```

```
// Couple Invoker/ConcreteCommand
// Invoker cas 1: fonction sans argument
void FunctionalInvoker() { ... }
// ConcreteCommand d'appel de l'Invoker
class ConcreteCommand1 : public Command {
public:
  ConcreteCommand1() {};
  void Execute() { FunctionalInvoker(); }
};
```

```
// Couple Invoker/ConcreteCommand
// Invoker cas 2: fonction avec arguments
void FunctionalInvokerWithArgs(int args) { ... }
// ConcreteCommand d'appel de l'Invoker
class ConcreteCommand2 : public Command {
private: int arg;  // sauvegarde interne de l'argument
public:  ConcreteCommand2(int x) : arg(x) {}
  void Execute() { FunctionalInvokerWithArgs(arg); }
};
```

```
// Couple Invoker/ConcreteCommand
// Invoker cas 3: méthode d'un objet
class ClassInvoker {
private: int x;
public:  ClassInvoker(int v) : x(v) {}
         // méthode que l'on veut invoquer
       void incr() { x++; }
};
// ConcreteCommand d'appel de l'Invoker
class ConcreteCommand3 : public Command {
private: // référence interne vers l'objet
         ClassInvoker &obj;
public:  ConcreteCommand3(ClassInvoker &o) : obj(o) { ... }
         void Execute() { obj.incr(); }
};
```

```
// code client
ClassInvoker  object(4);
Receiver    receiver;
receiver.AddCommand(new ConcreteCommand1);
receiver.AddCommand(new ConcreteCommand2(4));
receiver.AddCommand(new ConcreteCommand3(object));
receiver.Action();
```

**Note :** noter que l'utilisation d'objet fonctionnel `std::function`, de `std::bind` et de `std::placeholder` permet l'utilisation d'une classe ConcreteCommand générique.

**Exemple :**

```
// ConcreteCommand compatible avec tout function
class ConcreteCommand : public Command {
private: function<void()>  fun;
public:  ConcreteCommand(function &&f) : fun(move::f) {}
         void Execute() { fun(); }
};
```

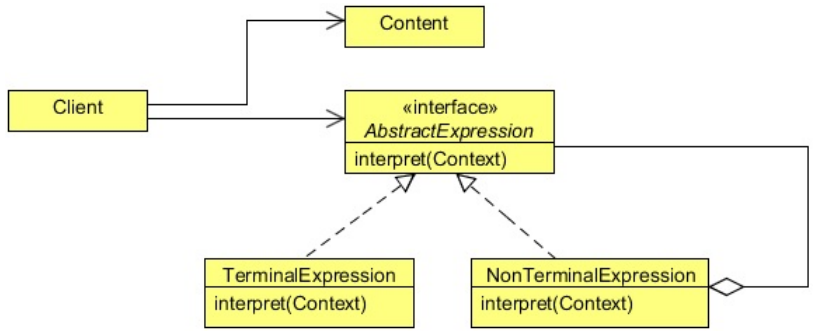
```
// code client équivalent au précédent
ClassInvoker  object(4);
Receiver    receiver2;
receiver.AddCommand( new ConcreteCommand(
    FunctionalInvoker));;
receiver.AddCommand( new ConcreteCommand(
    bind(FunctionalInvokerWithArgs,4)));
receiver.AddCommand( new ConcreteCommand(
    bind(&ClassInvoker::incr, ref(object))));
```

## 3.3 Interpréteur (Interpreter)

**Intérêt :**

A partir d'un langage, définir une représentation de sa grammaire et un interpréteur qui utilise la représentation pour interpréter des expressions du langage.

**Diagramme UML :**




**Application :**

- lorsqu'il y a un langage à interpréter, et que les expressions du langage peuvent être représentées sous forme d'un arbre syntaxique abstrait.
- la grammaire doit être simple (pour les grammaires complexes, un outil de génération d'analyseurs syntaxiques est préférable), et l'efficacité ne doit pas être critique.

**Participants :**

- `AbstractExpression` : déclare une opération abstraite `Interpret` commune à la totalité de l'arbre syntaxique abstrait.
- `TerminalExpression` : associé à tout symbole terminal d'une expression (une instance par symbole terminal). Implémente le `Interpret` associé au terminal.
- `NonterminalExpression` : (expression du type $R \rightarrow R_1 R_2 \ldots R_n$) une classe nécessaire par règle $R$ de la grammaire, où chaque $R_i$ est une `AbstractExpression`.
- `Context` : contient les informations globales pour l'`Interpreter`.
- `Client` : construit l'arbre syntaxique abstrait à partir d'une expression du langage. Puis, fait appel à `Interpret` dans le contexte souhaité pour l'expression.

**Collaboration :**

- Le `Client` construit construit l'arbre syntaxique abstrait à partir d'une expression du langage (assemblé à partir de `NonterminalExpression`.
- Puis le `Client` initialise le context et invoque l'opération d'interprétation.
- Chaque `NonterminalExpression` définit `Interpret` en terme d'`Interpret` sur chaque sous-expression (= appel récursif). Chaque `TerminalExpression` définit `Interpret` comme une interprétation de l'expression terminale.
- Sur chaque nœud, `Interpret` utilise le `Context` pour stocker et accèder ) l'état de l'interpréteur.

**Conséquences :**

- facilité de modifier la grammaire : utilise des classes pour représenter les règles de grammaires (peuvent être héritées pour changer ou étendre la grammaire)
- implémenter la grammaire est facile, mais les grammaires complexes sont dures à maintenir.
- facilité d'ajouter de nouvelle façon d'interpréter les expressions.

**Implémentation :**

- **création de l'arbre syntaxique abstrait :** le pattern n'explique pas comment le créer. Tous les choix sont possibles.

- **opération `Interpret` :** s'il est courant de changer d'`Interpreter`, le patron `Visitor` peut être utilisé.
- **partage de symboles terminaux** : les symboles terminaux peuvent être partagé avec le patron `FlyWeight`.

**Exemple :** langage = expression arithmétique avec des entiers positifs (sans priorité des opérateurs)

```
class Context {
private
    map<char,int>  vars;
public:
  void Assign(const char x, int v) { vars[x] = v; }
  int Value(const char x) const { return vars.at(x); }
};
```

```
class AbstractExpression {
public:
  virtual int interpreter(const Context &context) = 0;
  virtual ~AbstractExpression() {}
};
```

```
class TerminalValue : public AbstractExpression {
  int    val;
public:
  TerminalValue(const string &s) { val = stoi(s); }
  int Interpret(const Context &context) { return val; }
};
```

```
class TerminalVariable : public AbstractExpression {
  char   var;
public:
  TerminalVariable(const string &s) { var = s[0]; }
  int Interpret(const Context &context) {
    return context.Value(var);
  }
};
```

```
using Fun = int(*)(int, int);
Fun Add = [](int a, int b) { return a + b; };
Fun Min = [](int a, int b) { return a - b; };
Fun Mul = [](int a, int b) { return a * b; };
Fun Div = [](int a, int b) { return a / b; };
```

```
class NonTerminalOperator : public AbstractExpression {
private:  AbstractExpression *lh, *rh;
          Fun op;
protected:
  bool isValue(const string &s) { ... }
  bool isVariable(const string &s) { ... }
  bool isAbstractExpression(const string &s) { ... }
  AbstractExpression* NewAbstractExpression(const string &s) {
    if (isValue(s))         return new TerminalValue(s);
    else if (isVariable(s)) return new TerminalVariable(s);
    else if (isAbstractExpression(s))
            return new NonTerminalOperator(s);
    else return nullptr;
  }
public:
  NonTerminalOperator(const string &s); // voir plus loin
  int Interpret(const Context &context) {
    return op(lh->Interpret(context), rh->Interpret(context));
  }
};
```

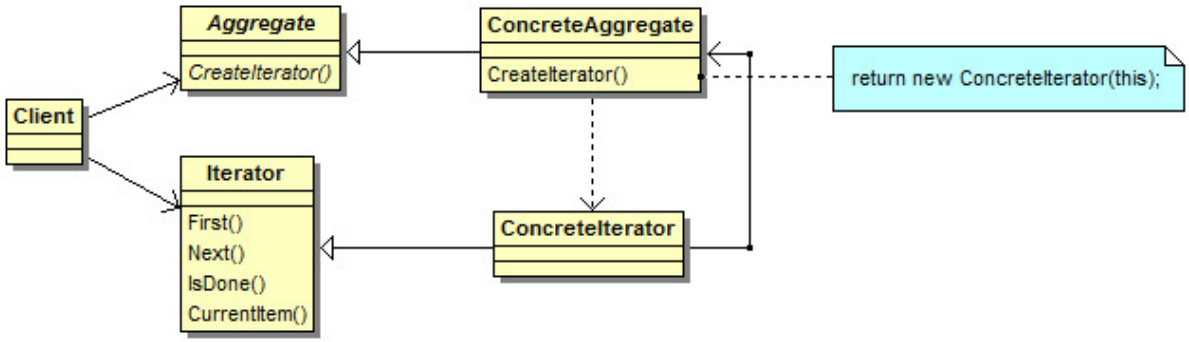
```
NonTerminalOperator::NonTerminalOperator(const string &s) {
  assert(isAbstractExpression(s));
  string  OpName("+-*/");
  Fun   OpFun[4] = { Add, Min, Mul, Div };
  for (size_t i = 0; i < s.size();++i) {
    auto it = find(OpName.begin(), OpName.end(), s[i]);
    if (it != OpName.end()) {
      lh = NewAbstractExpression(s.substr(0, i));
      rh = NewAbstractExpression(s.substr(i+1,s.size()-i));
      op = OpFun[distance(OpName.begin(), it)];
    }
  }
}
```

```
// utilisation
Context c;
c.Assign('x', 5);
AbstractExpression *s = new NonTerminalOperator("x+5*13");
int result = s->Interpret(c);
```

## 3.4 Itérateur (Iterator)

**Intérêt :** fournir un moyen d'accéder séquentiellement aux éléments d'un objet agrégé sans exposer sans représentation sous-jacente.

**Diagramme UML :**




**Application :**

- accéder à un objet agrégé sans exposer sa représentation interne.
- supporter les traversées multiples d'un objet agrégé.
- fournir une interface unifiée pour traverser différentes structures d'agrégats (i.e. itération polymorphique).

**Participants :**

- `Iterator` : définit une interface pour accéder aux éléments ou les traverser.
- `ConcreteIterator` :
  - ◇ implémente l'interface de l'`Iterator`.
  - ◇ garde la trace de la position courante dans la traversée de l'agrégat.
- `Aggregate` : définit l'interface pour créer un objet de type `Iterator`.
- `ConcreteAggregate` : implémente l'interface de création de l'`Iterator` afin de retourner une instance du ConcreteIterator correct.

**Conséquences :**

- support de variation dans la traversée de l'agrégat = changer l'instance du ConcreteIterator.
- simplifie l'interface de l'agrégat (ne contient pas les méthodes de traversée)/ plus d'une traversée peut être en cours dans un agrégat : l'Itarator garde la trace de son propre état de la traverse.

**Questions sur l'implémentation :**

- Qui contrôle l'Iterator : 2 approches
  **Iterator interne :** le client avance la traversée lui-même (appel à Next(), CurrentItem(), ...)
  **Iterator externe :** le client passe la fonction à appliquer sur chaque élément, et l'Iterator effectue la traversée en appliquant la fonction sur chaque élément.
- Où est défini l'algorithme de traversée ?
  **dans l'Iterator** lui-même
  **dans l'agrégat :** l'Iterator stocke alors seulement l'état de l'itération (= un curseur) :
  • l'appel aux méthodes de l'Iterator prend alors comme paramètre le curseur à modifier.
  • une seule méthode de traversée possible.

**Implémentation :**

- **Robustesse d'un `Iterator` :** il peut être dangereux de modifier un agrégat pendant qu'on le traverse : si un élément est ajouté ou retiré, un élément peut être manqué ou lu 2 fois. Solution simple (et couteuse) : associer une copie de l'agrégat à l'`Iterator`.

- **Opération supplémentaire pour un agrégat :** il est possible d'ajouter les opérations suivantes sur les agrégats ordonnées : Previous, SkipTo, ...
- **Iterator polymorphique en C++ :** nécessite que l'Iterator soit alloué dynamiquement par une Factory. Donc, ne les utiliser seulement lorsque le polymorphisme est obligatoire. Autre inconvénient : le client est responsable de leurs libérations (sauf à utiliser un Proxy).
- **Accès privilégié de l'`Iterator` à l'agrégat :** l'`Iterator` peut être vu comme une extension de l'agrégat (couplage étroit).
- **Iterator pour Composite :** un Iterator externe peut être difficile à implémenter sur des structures récursives agrégées comme un Composite.
- **Null Iterator :** `Iterator` dégénéré utile pour gérer les conditions aux bornes. Objet pour lequel : IsDone est toujours vrai (= la traversée est fini). Dans une structure récursive, un `Iterator` par nœud, un `NullIterator` par feuille.

**Exemple :**

```
// interface de l'agrégat
class Aggregate {
public: virtual Iterator* CreateIterator() = 0;
};
```

```
// agrégat concret
class ConcreteAggregate : public Aggregate {
private: size_t size;
         int  *data;
public:
  ConcreteAggregate(size_t n)
        : data(new int[n]), size(n) {}
  Iterator* CreateIterator()
        { return new ConcreteIterator(*this); }
  friend class ConcreteIterator;
};
```

```
class Iterator {
public: virtual void First() = 0;
        virtual void Next() = 0;
        virtual bool IsDone() = 0;
        virtual int CurrentItem() = 0;
};
```

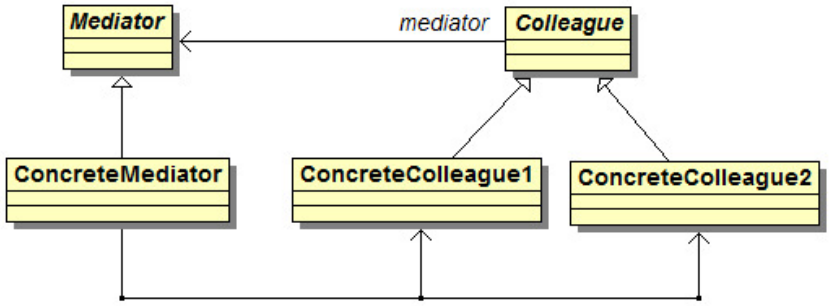
```
class ConcreteIterator : public Iterator {
private: int        *it;
       ConcreteAggregate  &ref;
public:
  ConcreteIterator(ConcreteAggregate &r) : ref(r), it(r.data) {}
  void First() { it = ref.data; }
  void Next() { it++; }
  bool IsDone() { return (it == ref.data + ref.size); }
  int CurrentItem() { return *it;  }
};
```

```
// code client
Aggregate    *tab = new ConcreteAggregate(10);
Iterator     *it = tab->CreateIterator();
for (it->First(); !it->IsDone(); it->Next()) {
    // utiliser it->CurrentItem() pour la valeur courante
}
```

## 3.5 Médiateur (Mediator)

**Intérêt :** définit un objet qui encapsule comment un ensemble d'objets interagit. Permet un couplage faible en empêchant les objets de faire références les uns aux autres explicitement et permet de modifier leurs interactions indépendamment.

**Diagramme UML :**




**Application :**

- lorsqu'un ensemble d'objets communiquent d'une façon complexe et bien définie. Les interdépendances résultantes sont non structurées et difficiles à comprendre.
- réutiliser un objet lorsqu'il fait référence et communique avec beaucoup d'autres objets.
- un comportement qui est distribué entre un ensemble de classes doit être modifiés.

**Participants :**

- `Mediator` : définit une interface pour communiquer avec les objets `Colleague`.
- `ConcreteMediator` :
  - ◇ implémente le comportement coopératif en coordonnant les objets `Colleague`.
  - ◇ connait et maintient ses `Colleague`s
- `Colleague` classes :
  - ◇ chaque classe `Colleague` connait son `Mediator`.
  - ◇ chaque `Colleague` communique avec son `Mediator` à chaque fois qu'il veut communiquer avec un autre `Colleague` (pas de communication directe entre `Colleague`).

**Collaboration :**

- les `Colleague`s envoient et reçoivent les requêtes au `Mediator`.
- le `Mediator` implémente le comportement coopératif en routant les requêtes entre les `Colleague`s appropriés.

**Conséquences :**

- **Limite le subclassing :** localise un comportement qui aurait sinon été distribué entre plusieurs objets. Le changement de comportement introduit du subclassing sur le `Mediator` seulement.

- **Découplage des `Colleagues` :** couplage faible entre `Colleague`s (permet aux `Colleague`s et au Mediator de varier indépendamment).
- **Simplifie les protocoles entre objets :** protocole $1 : N$ (Mediator :Object) au lieu de $N : N$ (Object :Object). Conséquence : plus simple à comprendre, à maintenir et à étendre.
- Abstraction de la façon dont les objets coopèrent : faire de la médiation un concept indépendant et l'encapsuler dans un objet permet de se concentrer sur les interactions plutôt que sur les comportements individuels (rend plus clair la façon dont les objets interagissent avec le système).
- **Centralisation du contrôle :** échange la complexité d'interaction par la complexité dans le `Mediator`. Comme le `Mediator` encapsule les protocoles, il est plus complexe qu'un `Colleague` individuel. Conséquence : le `Mediator` est un monolithe difficile à maintenir.

**Implémentation :**

- omettre la classe Mediator abstraite s'il n'y a qu'un seul `Mediator`. La classe `Mediator` abstraite permet aux `Colleague`s de communiquer avec différentes sous-classes du `Mediator`.
- **communication Colleague-Mediator :** une solution est d'utiliser le pattern Observer pour faire communiquer le Mediator (=Observer) avec les Colleagues (Subjects). Cer derniers envoient des notifications au Mediator en cas de changement.

**Exemple :**

```
class Colleague {
public: Mediator   *mediator;
     Colleague(Mediator *m, size_t v = 0): mediator(m)
           { m->Register(this, v); }
       // interface partagée entre collègues
     virtual void update() = 0;
     virtual void set(int) = 0;
     virtual int get() = 0;
};
```

- Idée de ce `Mediator`, permet des médiations entre collègues (`Set`/`Get`/`Update`) ou dans un groupe de collègue (`SetGroup`/`GetGroup`/`UpdateGroup`) pour le `ConcreteMediator`.
- La classe `ConcreteColleague1` utilise ce médiator pour communiquer entre collègues.
- La classe `ConcreteColleague2` utilise ce médiator pour communiquer dans un groupe de collègues.

```
class Mediator {
public:
  virtual void Register(Colleague*, size_t=0) = 0;
  virtual void UpdateAll() = 0;
  virtual void UpdateGroup(size_t group) = 0;
  virtual void Update(size_t id) = 0;
  virtual void SetGroup(size_t group, int val) = 0;
  virtual void Set(size_t id, int) = 0;
  virtual int GetGroup(size_t group) = 0;
  virtual int Get(size_t id) = 0;
  virtual void Status() = 0;
};
```

```
class ConcreteMediator : public Mediator {
protected: vector < pair<Colleague*, size_t> > objs;
public:
  void Register(Colleague *c, size_t g)
        { objs.push_back(make_pair(c,g)); }
  void UpdateAll() { for (auto &x : objs) x.first->update(); }
  void UpdateGroup(size_t group) {
      for (auto &x : objs)
        if (x.second & group) x.first->update();  }
  void Update(size_t id) { objs[id].first->update(); }
  void SetGroup(size_t group, int val) {
      for (auto &x : objs)
        if (x.second & group) x.first->set(val); }
  void Set(size_t id, int val) { objs[id].first->set(val); }
  int Get(size_t id) { return objs[id].first->get(); }
  int GetGroup(size_t group) {
    int val = 0;
    for (auto &x : objs)
            if (x.second & group) val += x.first->get();
    return val;
  }
  void Status() {
     for(auto &x : objs)
        cout << x.second << ":" << x.first->get() << endl;  }
};
```

```
// class collègue communiquant par groupe
// utilise seulement les fonctions
// de médiation de groupe du Mediator
class ConcreteColleague1 : public Colleague {
  int  x;
public:
  ConcreteColleague1(Mediator *m, size_t  g = 0) :
        Colleague(m,g), x(0) {}
  void update() { ... }
  void set(int u) { x = u;  ... }
  int get() { ... return x; }
  void UpdateGroup(size_t group)
        { mediator->UpdateGroup(group); }
  void UpdateAll() { mediator->UpdateAll();  }
  void SetGroup(size_t group, int val)
        { mediator->SetGroup(group, val); }
  int GetGroup(size_t group)
        { return mediator->GetGroup(group);  }
};
```

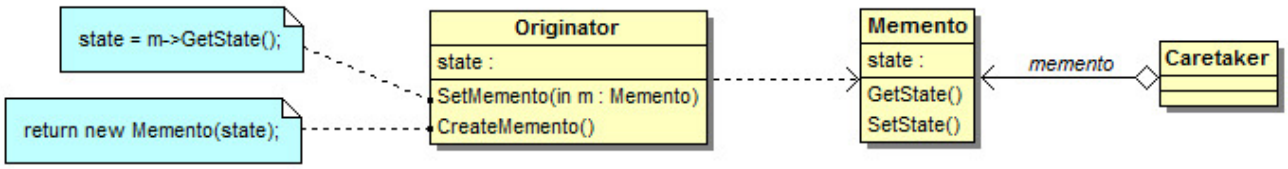
```
// class collègue communiquant individuellement
// utilise seulement les fonctions
// de médiation individuelle du Mediator
class ConcreteColleague2 : public Colleague {
  int  a,b;
public:
  ConcreteColleague2(Mediator *m, size_t g = 0) :
        Colleague(m,g), a(0), b(0) {}
  void update() { ... }
  void set(int u) { a = u / 2; b = u - a; ... }
  int get() { ... return a+b; }
  void UpdateAll() { mediator->UpdateAll(); }
  void Update(size_t id) { mediator->Update(id); }
  void Set(size_t id, int val) { mediator->Set(id, val); }
  int Get(size_t id) { return mediator->Get(id); }
};
```

```
Mediator  *med = new ConcreteMediator();
ConcreteColleague1 *c1 = new ConcreteColleague1(med,1);
ConcreteColleague2 *c2 = new ConcreteColleague2(med,2);
ConcreteColleague2 *c3 = new ConcreteColleague2(med,1);
c1->SetGroup(1,5);
med->Status();
```

## 3.6 Mémento (Memento)

**Intérêt :** sans viol de l'encapsulation, capture et externalise l'état interne d'un objet tel que son état puisse être retourné plus tard.

**Diagramme UML :**




**Application :**

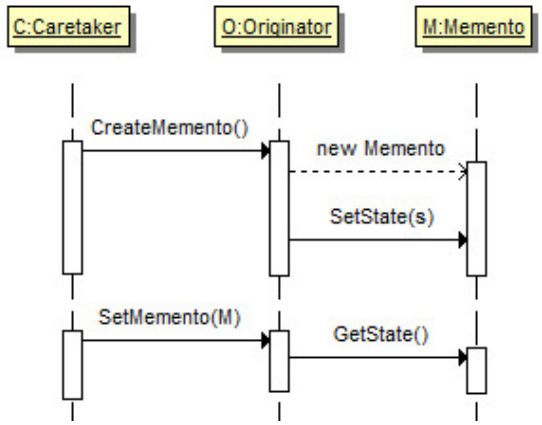
- une photo complète ou partielle de l'état d'un objet doit être sauvegardée tel que son état puisse être retourné.
- une interface directe pour obtenir l'état pourrait exposer les détails de l'implémentation ou briser l'encapsulation de l'objet.

**Participants :**

- `Memento` :
  - ◇ stocke (une partie de) l'état de l'objet d'origine `Originator`.
  - ◇ protège contre les accès par des objets autres que l'`Originator`. Le Memento a 2 interfaces :
    • **pour le `Caretaker` :** interface resserrée (passe seulement le `Memento` aux autres objets).
    • **pour l'`Originator` :** interface large (accès à toutes les données,et inversement pour le `Memento`)

- `Originator` :
  - crée un `Memento` contenant une photo de son état interne courant.
  - utilise le `Memento` pour restaurer un état interne.
- `Caretaker` :
  - responsable de la sauvegarde de `Memento`.
  - n'opère ou n'examine jamais le contenu du `Memento`.

**Collaboration :**




- le `Caretaker` C demande le `Memento` à l'`Originator`, le stocke pour un certain temps, et le retourne à l'`Originator`.
- un `Memento` est passif. Seul l'`Originator` qui a créé le `Memento` affectera et récupérera son état.

**Conséquences**

- préservation des bornes de l'encapsulation : le `Memento` évite l'exposition d'informations que seul l'`Originator` devrait gérer, mais qui ne devraient jamais être stockées à l'extérieur de l'`Originator`.
- peut être couteux (vitesse et mémoire) si l'on doit copier une grande quantité d'informations.

**Implémentation :**

- Gestion des 2 interfaces du Memento :
  - interface étroite (Originator) : faire de l'Originator un ami du Memento, et rendre l'interface large privée.
  - interface étroite (Caretaker et autres) : inteface publique du Memento.
- Stockage du changement incrémental : il est possible de ne sauveharder que les changement de l'état interne (undoable commands history dans une liste).

**Exemple :**

```
struct OriginatorState {
  // internal states
  int    x;
  OriginatorState(int a) : x(a) {};
  // constructeur par copie définie
  // assignation par copie définie
  friend ostream& operator<<
        (ostream &os, const OriginatorState& s) {
    return os << "{" << s.x << "}";
  }
};
```

```
// classe utilisée pour stocker la copie de l'état
class Memento {
private:
  OriginatorState   state;
public:
  static Memento* null;
  Memento(const OriginatorState& s) : state(s) {}
  OriginatorState getState() {
    return state;  // retour par copie
  }
  void setState(const OriginatorState& s) {
    state = s; // assignation par copie
  }
  friend ostream& operator<<(ostream &os, const Memento& s) {
    return os << s.state;
  }
};
Memento* Memento::null = nullptr;
```

```
// classe stockant initialement l'état
class Originator {
private:
  OriginatorState    state;
public:
  Originator(int u) : state(u) {}
  void changeState(int u) { state.x = u;  }
  Memento *CreateMemento() {
    return new Memento(state);
  }
  void setMemento(Memento *m) {
    state = m->getState();
  }
  void updateMemento(Memento *m) {
    m->setState(state);
  }
  friend ostream& operator<<(ostream &os, const Originator& s) {
    return os << s.state;
  }
};
```

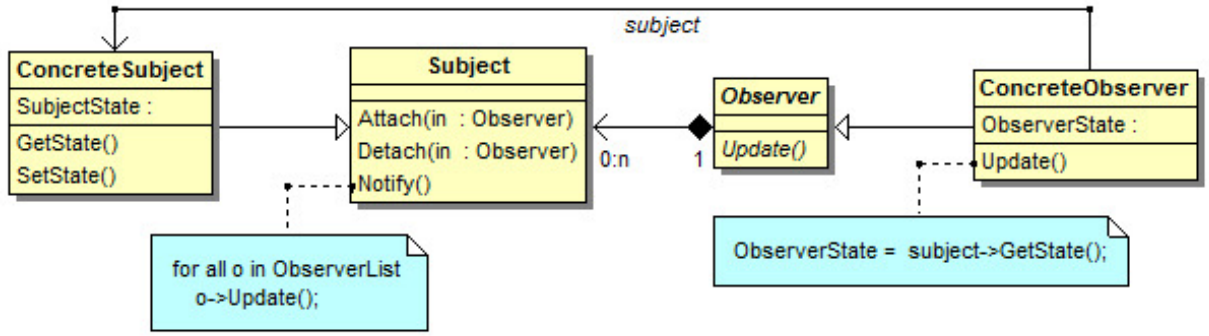
```
// classe permettant de sauvegarder ou restaurer des états
class Caretaker {
private:
  vector<pair<Originator*, Memento*>>  v;
public:
  void Register(Originator *o) {
    v.push_back(make_pair(o, o->CreateMemento() ));
  }
  void BackupStates() {
    for (auto &x : v) x.first->updateMemento(x.second);
  }
  void RestoreStates() {
    for (auto &x : v) x.first->setMemento(x.second);
  }
  friend ostream& operator<<(ostream &os, const Caretaker& s) {
    int id = 0;
    for (auto &x : s.v)
      os << ++id << *(x.first) << *(x.second) << ", ";
    return os << endl;
  }
};
```

```
// utilisation
Caretaker  backup;
backup.Register(&a);
backup.Register(&b);
backup.Register(&c);
cout << "initial states:" << endl << backup;
a.changeState(7);
c.changeState(10);
cout << "after changes" << endl << backup;
backup.RestoreStates();
cout << "after restore" << endl << backup;
b.changeState(6);
cout << "after changes" << endl << backup;
backup.BackupStates();
cout << "after backup" << endl << backup;
```

## 3.7 Observateur (Observer)

**Intérêt :** définir une dépendance entre objets tels que lorsqu'un objet change d'état, tout ceux dont il dépend sont avertis et mis-à-jour automatiquement. Permet de palier au problème de consistance de la partition d'une collection de classes coopérantes.

**Diagramme UML :**



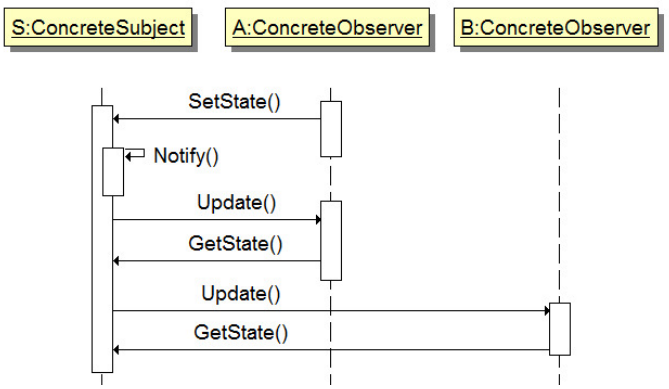

**Application :**

- lorsqu'une abstraction a 2 aspects coopérant l'un avec l'autre. Leur encapsulation permet de les faire varier et les réutiliser indépendamment.
- lorsque le changement d'un objet implique le changement d'un autre, et que l'on ne sait pas combien ont besoin d'être changés.
- lorsqu'un objet doit être capable de notifier d'autres objets sans faire d'hypothèse sur ce que sont ces objets (i.e. pas de couplage fort).

**Participants :**

- `Subject` :
  - connait ses `Observers`. Inversement, tout `Observer` peut observer un `Subject`.
  - fournit une interface pour attacher/détacher ses `Observers`.
- `Observer` : définit une interface de mise-à-jour pout les objets devant être notifiés des changements de sujet.
- `ConcreteSubject` :
  - stocke l'état d'intérêt des `ConcreteObservers`.
  - envoie les notifications à ses `Observers` quand son état change.
- `ConcreteObserver` :
  - maintient une référence à un `ConcreteSubject`.
  - stocke l'état avec lequel il doit rester consistent avec le sujet.
  - implémente l'interface de mise-à-jour de l'Observer pour maintenir son état consistant avec le sujet.

**Collaboration :**



- un `ConcreteSubject` change son état.
- Ce changement est suivi d'une notification indiquant à ses observateurs que son état a changé.
- Après avoir été informé du changement dans le `Subject` S, l' `Observer` A peut interroger S, et utiliser cette information pour réconcilier son état avec celui de S.

**Conséquences :**

- couplage abstrait entre Subject et Observer (à travers les classes et les couches logicielles.
- **broadcast :** tous les Observers intéressés inscrits au sujet sont avertis. Peu importe le nombre d'Observer.
- **mise-à-jour non souhaitée :** comme les Observers n'ont pas de lien entre eux, ils peuvent être aveugles du coût engendré par une mise à jour (update en cascade). Aggravé par le fait qu'un protocole simple ne donne pas de détail sur ce qui a changé.

**Implémentation :**

- **associer les Subjects aux Observers :** le plus simple est de stocker les références aux Observers dans le sujet.
- **Observer plus d'un Subject par un Observer :** étendre l'interface Update pour informer l'Observer quel Subject envoie la notification.
- **Qui déclenche la mise à jour ?** 2 possibilités :
  - ◇ placer dans SetState() du Subject l'appel à Notify() après le changement d'état.
    **avantage :** pas besoin d'appeler Notify()
    **inconvénient :** des opérations consécutives peuvent causer des mises-à-jour consécutives.
  - ◇ laisser au client l'appel au Notify().
    **avantage :** Notify() après une suite d'appel.
    **inconvénient :** oubli de l'appel par le client.
- **références aux sujets effacés** (dangling références) : attention aux Observers pointent vers des sujets effacés. Une solution est que le Subject informe ses Observers.
- attention à ce que le Subject soit consistant (et mis-à-jour) avant la notification.

**Exemple :**

```
class Subject {
  // ensemble indépendant de fonctionnalité
  list<Observer*>   observers;
    // couplés à travers une interface
public:
  // observer relation interface
  void Attach(Observer *obs) { observers.push_back(obs); }
  void Detach(Observer *obs) {
    auto pos = find(observers.cbegin(), observers.cend(), obs);
    if (pos != end(observers)) observers.erase(pos);
  }
  void Notify(){ for (auto &x : observers) x->Update(); }
  virtual int GetState() = 0;
  virtual void SetState(int) = 0;
};
```

```
class ConcreteSubject : public Subject {
private:
  int   state;
public:
  ConcreteSubject(int s) : state(s) {}
  int GetState() { return state; }
  void SetState(int u) {
    state = u;
    cout << "up-S:" << state << "␣";
    Notify();
  }
};
```

```
class Observer {
private:
  Subject *subject;
protected:
  Subject *GetSubject() { return subject; }
public:
  Observer(Subject *s) : subject(s) {
    subject->Attach(this);
  }
  virtual void Update() = 0;
  void SetState(int u) { subject->SetState(u); }
};
```

```
class ConcreteObserverA : public Observer {
private: int    x;
public:
  ConcreteObserverA(Subject *s) :
        Observer(s), x(s->GetState()) {}
  // concrete implementation of update
  void Update() {
    int u = GetSubject()->GetState();
    if (x > u) x = u;
  }
};
```

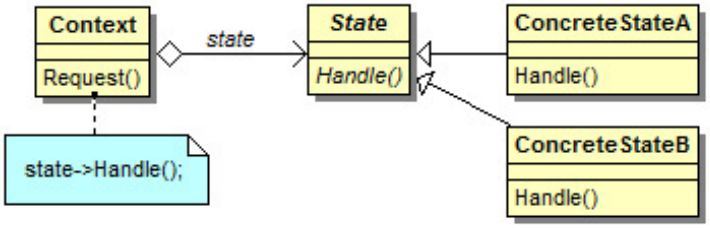
```
class ConcreteObserverB : public Observer {
private: int    x;
public:
  ConcreteObserverB(Subject *s) :
        Observer(s), x(s->GetState()) {}
    // concrete implementation of update
  void Update() {
    int u = GetSubject()->GetState();
    if (x < u) x = u;
  }
};
```

```
// s = sujet (valeur entière)
// o1 = observateur (plus petite valeur prise par s)
// o2 = observateur (plus grande valeur prise par s)
int main() {
  Subject *s = new ConcreteSubject(2);
  Observer *o1 = new ConcreteObserverA(s);
  Observer *o2 = new ConcreteObserverB(s);
  s->SetState(4);   cout << endl;
  o1->SetState(8);  cout << endl; // changement d'état par o1
  o2->SetState(1);  cout << endl; // changement d'état par o2
```

## 3.8 État (State)

**Intérêt :** permet à un objet de modifier son comportement quand un état interne change. L'objet semble changer de classe.

**Diagramme UML :**




**Application :**

- le comportement d'un objet dépend de son état, et doit changer son comportement à l'exécution en relation avec cet état.
- les opérations sur l'objet sont basées sur des conditionnelles dépendantes de son état : permet de placer chaque branche de la conditionnelle dans une classe (i.e. une classe par état).

**Participants :**

- `Context` :
  - définit l'interface exposée au client.
  - maintient une instance de la sous-classe `ConcreteState` qui définit l'état coutant.
- `State` : définit l'intégrale pour encapsuler le comportement associé avec un état particulier du contexte.
- `ConcreteState` : chaque sous-classe implémente un comportement associé avec un état du contexte.

**Collaboration :**

- le `Context` délègue les requêtes spécifiques à l'état au `ConcreteState` courant.
- le `Context` peut se passer en argument à l'objet `State` traitant la requête (permet l'accès au contexte si nécessaire).
- le `Context` est l'interface primaire du client qui configure le contexte avec les états.

**Conséquences :**

- localise les comportements spécifiques aux états et partitionne les comportements pour différents états. Le code spécifique à un état est dans une classe : permet d'ajouter facilement de

nouveaux états.
- rend les transitions d'état explicites : tend à éviter des états internes inconsistants.
- les objets `States` peuvent être partagés s'ils n'ont pas de variables d'instance (voir Fly-Weight).

**Implémentation :**

- ne spécifie pas qui définit les transitions d'état :
  - si les critères sont fixés, ils peuvent être implémentés dans le `Context`.
  - plus flexible et approprié : laisse les sous-classes `ConcreteState` spécifier l'état suivant et effectuer les transitions : requier l'ajout d'une interface au `Context` permettant au objet `State` de fixer explicitement l'état du `Context`.
- alternative basée sur des tables (cf p341)

**Exemple :**

```
class State {
public: virtual void HandleOpen(Context *s) = 0;
        virtual void HandleClose(Context *s) = 0;
        virtual bool HandleStatus() const = 0;
};
```

```
class ConcreteStateOpened : public State {
public: void HandleOpen(Context *s) { ... }
        void HandleClose(Context *s) {
           s->setState(new ConcreteStateClosed);
           delete this;
        }
        bool HandleStatus() const { return true; }
};
```

```
class ConcreteStateClosed : public State {
public: void HandleOpen(Context *s) {
            s->setState(new ConcreteStateOpened);
            delete this;
        }
        void HandleClose(Context *s) { ... }
        bool HandleStatus() const { return false; }
};
```

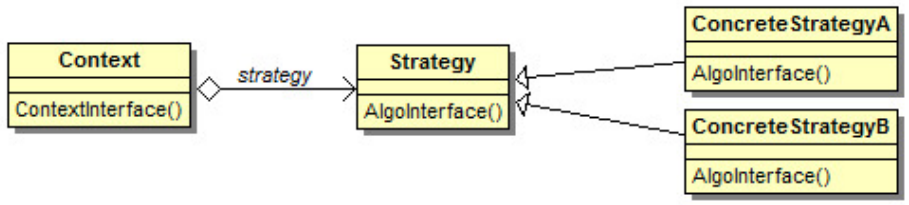
```
class Context {
private:
  State *state;
public:
  Context(bool v) {
    if (v) state = new ConcreteStateOpened;
    else new ConcreteStateClosed;
  }
  void setState(State *s) { state = s; }
  void RequestOpen() { state->HandleOpen(this); }
  void RequestClose() { state->HandleClose(this); }
  bool RequestStatus() const { return state->HandleStatus(); }
};
```

```
// utilisation
Context  c(true);
c.RequestOpen();
c.RequestClose();
```

## 3.9 Stratégie (Strategy)

`Intérêt:` définit une famille d'algorithmes, encapsulant chacun, et les rendant interchangeables. Permet à un algorithme de varier indépendamment des clients.

**Diagramme UML :**




**Applications :**

- plusieurs classes ne changent que par leurs comportements. Façon de configurer une classe avec un comportement parmi plusieurs.
- besoin de différentes variations d'un algorithme implémenté comme une hiérarchie de classe.
- évite d'exposer les structures de données spécifiques aux algorithmes.
- une classe définit beaucoup de comportements, et qu'ils apparaissent comme des conditionnelles multiples. Un comportement peut alors être implémenté comme une `Strategy`.

**Participants :**

- `Strategy` : déclare une interface commune à tous les algorithmes. Le `Context` utilise cette interface pour faire appel à l'algorithme défini par une `ConcreteStrategy`.
- `ConcreteStrategy` : implémente l'algorithme utilisant l'interface `Strategy`.
- `Context` est configuré avec un objet `ConcreteStrategy`.
  - maintient une référence à l'objet `Strategy`.
  - peut définir une interface qui permet d'accéder à ses données.

**Collaboration :**

- `Strategy` et `Context` interagissent pour implémenter l'algorithme choisit. Un context peut passer les données requises par les algorithmes à la `Strategy` quand l'algorithme est appelé.
- Alternativement, le `Context` peut se passer lui-même comme argument à la `Strategy`.
- Un `Context` fait suivre les requêtes depuis ses clients vers ses `Strategies`.

**Conséquences :**

- **Famille d'algorithmes :** la hiérarchie de `Strategy` définie une famille d'algorithmes ou de comportements. Elle aide à mettre en commun les fonctionnalités des algorithmes.
- **Alternative aux sous-classes :** l'encapsulation d'algorithme dans des classes de `Strategy` permet de varier l'algorithme indépendamment du `Context` (+ facile de permutter, à comprendre et à étendre.
- Elimine les conditionnelles de choix d'un algorithme.

- **Choix d'implémentation** : permet de fournir différentes implémentations du même comportement.
- **Surcharge de communication entre le `Strategy` et le `Context` :** l'interface `Strategy` est partagée par toutes les `ConcreteStrategies`, et peut n'avoir à utiliser aucune des données passées.
- **Augmente le nombre d'objets :** l'overhead peut être réduit en implémentant les `Strategies` comme des "stateless objects" que les `Contexts` peuvent partager.

**Implémentation :**

- Les interfaces `Strategy` et `Context` doivent donner à `ConcreteStrategy` un moyen efficace d'accéder aux données dont il a besoin d'après le `Context` et inversement :
  • le Context passe les données en paramètres à la `Strategy` (garde la `Strategy` et le `Context` découplés).
  ⇒ Le `Context` peut passer des données inutiles.
  • le Context se passe comme argument, et la `Strategy` les demande explicitement au `Context`. La `Strategy` peut stocker la référence à son `Context` (on ne passe alors plus rien).
- **`Strategy` comme template :** un template peut être utilisé pour configurer une classe avec une `Strategy`. Marche seulement si la `Strategy` peut être sélectionnée à la compilation et ne doit pas être changée.

```
template <class aStrategy> class Context {
  public: void Operation() { theStrategy.DoAlgo(); };
  private: aSrategy theStrategy;
};
```

**Exemple :**

```
class Strategy {
public: virtual void AlgoInterface() = 0;
};
```

```
class ConcreteStrategyA : public Strategy {
  void AlgoInterface() { ... }
};
```

```
class ConcreteStrategyB : public Strategy {
  void AlgoInterface() { ... }
};
```

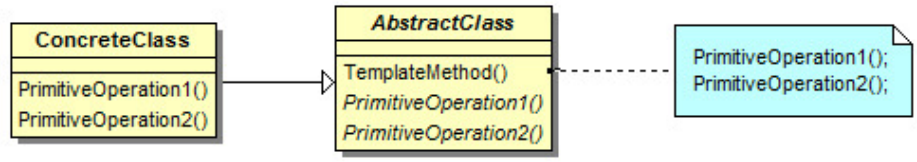
```
class Context {
private: Strategy *strategy;
public:
  Context(Strategy *s) : strategy(s) {}
  void SetStrategy(Strategy *s) { strategy = s; }
  void ContextInterface() { strategy->AlgoInterface(); }
};
```

```
// utilisation
Strategy  *strategyA = new ConcreteStrategyA,
          *strategyB = new ConcreteStrategyB;
Context   c(strategyA);
c.ContextInterface();
c.SetStrategy(strategyB);
c.ContextInterface();
```

## 3.10 Patron de méthode (Template Method)

**Intérêt :** définit le squelette d'un algorithme dans un algorithme dans une opération, repoussant certaines étapes à des sous-classes. Permet de redéfinir certaines étapes d'un algorithme sans changer la structure de l'algorithme.

**Diagramme UML :**




**Application :**

- pour implémenter des parties invariantes d'un algorithme une fois, et laisser aux sous-classes l'implémentation des comportements qui peuvent varier.
- quand le comportement commun aux sous-classes pourrait être regroupé et localisé dans une classe commune pour éviter la duplication de code.
- pour contrôler les extensions des sous-classes en définissant des opérations configurables à des points spécifiques.

**Participants :**

- AbstractClass :
  • définit des opérations primitives abstraites que les sous-classes concrètes redéfinissent pour implémenter les étapes d'un algorithme.
  • implémente une méthode template définissant le squelette d'un algorithme.
- ConcreteClass : implémente les opérations primitives pour effectuer les étapes de l'algorithme dans la sous-classe.

**Conséquences :**

- la Template Method est une technique fondamentale pour la réutilisation de code.
- il est important pour les Templates Methods de spécifier, parmi ses méthodes, lesquelles sont des hooks (= peuvent être surchargées) et lesquelles sont abstraites (= doivent être surchargées).
- exemples de façon de procéder :
  • extension de la classe parent : redéfinition + appel de l'opération parente.

```
void DerivedClass::Operation() {
  // comportement étendu de la classe dérivée
  ...
  // appel explicite de l'opération parente
  ParentClass::Operation();
}
```

  • la méthode dans la classe parent fait appel à des Hooks surchargées dans les classes dérivées.

```
void ParentClass::Operation() {
  // comportement du parent
  HookOperation();
}
// par défaut cette opération de fait rien dans la classe parent
void ParentClass::HookOperation() {};
// surcharge dans la classe fille
void DerivedClass::HookOperation() { ... };
```

**Implémentation :**

- utilisation du contrôle d'accès :
  • opérations primitives qu'une TemplateMethod appelle : protected memebers (= ne peuvent être appellées que par les TemplateMethods).
  • opérations primitives qui doivent être surchargées : déclarées comme pure virtual.
  • méthodes template : ne doivent pas être surchargées, donc déclarées comme non virtuelles.
- minimisation des opérations primitives : limiter le nombre de méthodes qui doivent être surchargées pour redéfinir un algorithme (raison : pénibilité et complexité).
- convention de nommage : identifier les opérations à surcharger en ajoutant un préfixe à leurs noms (exemple : `DoSomething(...)`);

**Exemple :**

```
class AbstractClass {
public: virtual int PrimitiveOperation1(int,int) = 0;
        virtual int PrimitiveOperation2(int,int) = 0;
        // méthode template à partir des operations primitives
        virtual int TemplateMethod(int x, int y) {
         int z = PrimitiveOperation1(x, y);
         return PrimitiveOperation2(x,z);
        }
};
```

```
// implémentation A des opérations primitives
class ConcreteClassA : public AbstractClass {
public: int PrimitiveOperation1(int x, int y) { return x | y; }
        int PrimitiveOperation2(int x, int y) { return x & y; }
};
```

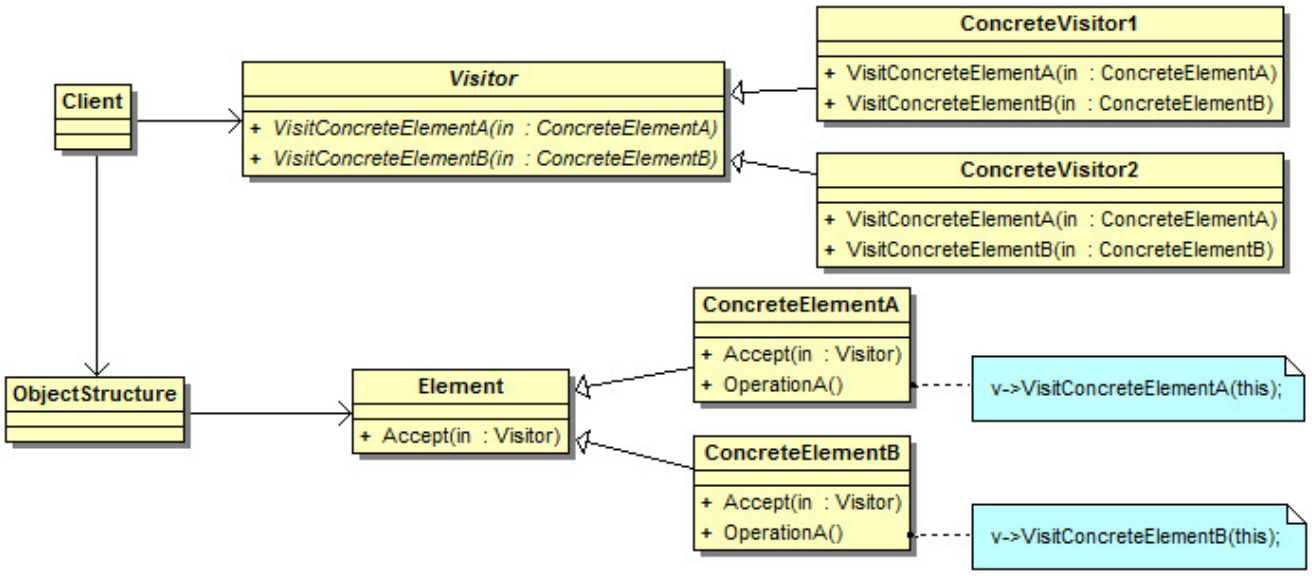
```
// implémentation B des opérations primitives
class ConcreteClassB : public AbstractClass {
public: int PrimitiveOperation1(int x, int y) { return x + y; }
        int PrimitiveOperation2(int x, int y) { return x * y; }
};
```

```
// utilisation
AbstractClass  *obj = new ConcreteClassA;
obj->TemplateMethod(2, 3);
```

## 3.11 Visiteur (Visitor)

**Intérêt :** représente une opération à effectuer sur des éléments de la structure d'un objet. Il permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels il agit.

**Diagramme UML :**




**Applications :**

- lorsque la structure d'un objet contient de nombreuses classes avec des interfaces différentes, et que l'on souhaite effectuer des opérations sur ces objets qui dépendent de leurs classes concrètes.
- des opérations nombreuses, distinctes et sans relations entre elles doivent être effectuées sur des objet dans une structure d'objets, et que l'on souhaite éviter de polluer ces classes avec ces opérations.
- les classes définissant la structure de l'objet ne doit pas changer, mais de nouvelles opérations doivent être définies sur la structure.

**Participants :**

- `Visitor` : déclare l'opération `Visit` sur chaque classes des éléments concrets dans la structure d'éléments. Le nom et la signature de l'opération identifie la classe que envoie la requête de visite au `Visitor`. Permet au `Visitor` de déterminer la classe concrète de l'élément à visiter, et d'y accéder directement au travers d'une interface particulière.
- `ConcreteVisitor` : implémente chaque opération déclarée par le `Visitor` (1 `ConcreteVisitor` = implémentation d'une seule opération). Chaque opération implémente un fragment de l'algorithme défini pour la classe correspondant à la structure. Le `ConcreteVisitor` fournit le contexte pour l'algorithme et stocke son état local. Cet état accumule souvent des résultats pendant la traversée de la structure.
- `Element` : définit une opération `Accept` qui prend un `Visitor` en argument.
- `ConcreteElement` : implémentation d'un élément.
- `ObjectStructure` : structure qui peut :
  - énumérer ses éléments,

- fournir une interface de haut-niveau qui permet au Visitor de visiter ses éléments.
- être un Composite ou une collection (tels qu'une liste ou un ensemble).

**Collaboration :**

- un client qui utilise un pattern `Visitor` doit créer un `ConcreteVisitor` pouis traverser la structure de l'objet, utilisant le `Visitor` pour visiter tous les éléments.
- quand un élément est visité, il appelle le `Visitor` correspondant à sa classe. L'élément se passe en argument de cette opération pour laisse le `Visitor` utiliser son état.

**Conséquences :**

- facilite l'ajout de nouvelles opérations (il suffit d'ajouter un nouveau `ConcreteVisitor`).
- un Visitor groupe l'ensemble des opérations élémentaire pour une opération (cf `VisitConcreteElement`).

**Implémentation :**

- **Double dispatch :** ajoute des opérations à des classes sans les modifier avec la technique du double dispatch.
  • **single dispatch** (C++) : 2 critères déterminent quelle opération satisfait une requête (le nom de la requête et le type du receiver).
  • **double dispatch :** dépend du type de la requête (le nom) + le type de 2 receivers.
  • opération Accept : double dispatch car dépend du type du `Visitor` et du receiver.
- Responsabilité de la traversée : 3 possibilités (dans la structure de l'objet, dans le Visitor, dans un itérator séparé)
  • dans la structure de l'objet : itération sur ses propres éléments avec appel de `Accept` sur chacun.
  • dans le `Visitor` : à l'avantage de permettre l'implémentation de traversée complexe, mais l'inconvénient de dupliquer le code de traverser sur chaque élément.
  • dans l'`Iterator` : dépend du type d'Iterator :
  ◇ **interne :** implémenté sur la structure d'objet, mais pas de double-dispatch (opération sur le Visitor avec l'élément en argument).
  ◇ **externe :** idem (mais opération sur l'élément avec le Visitor en argument).

**Exemple :**

```
class Element {
public: virtual void Accept(Visitor*) = 0;
};
```

```
// lumière
class ConcreteElementA : public Element {
private: ...
public: void Accept(Visitor *v) { v->visit(this); }
        void TurnOn() { ... }
        void TurnOff() { ... }
};
```

```
// porte
class ConcreteElementB : public Element {
private: ...
public: void Accept(Visitor *v) { v->visit(this); }
        void Open() { ... }
        void Close() { ... }
        void Lock() { ... }
        void Unlock() { ... }
};
```

```
class Visitor {
public: virtual void visit(ConcreteElementA*) = 0;
        virtual void visit(ConcreteElementB*) = 0;
};
```

```
class ConcreteVisitor1 : public Visitor {
  // closing all
  void visit(ConcreteElementA *a) { a->TurnOff(); }
  void visit(ConcreteElementB *b) { b->Close();  b->Lock(); }
};
```

```
class ConcreteVisitor2 : public Visitor {
  void visit(ConcreteElementA *a) { a->TurnOn();  }
  void visit(ConcreteElementB *b) { b->Unlock();  b->Open(); }
};
```

```
// utilisation
vector<Element*>  list = {
    new ConcreteElementA, new ConcreteElementB,
    new ConcreteElementB, new ConcreteElementA };
ConcreteVisitor1    VisitorOff;
ConcreteVisitor2    VisitorOn;
// Applying visitor 1 on Element list
for (auto &x : list) x->Accept(&VisitorOff);
// Applying visitor 2 on Element list
for (auto &x : list) x->Accept(&VisitorOn);
```

# Conclusion

Ce chapitre est important :

- les patrons de conceptions constituent une source d'inspiration sur différentes manières d'organiser des classes afin d'obtenir certains comportements souhaités.
- les différents patrons peuvent être combinés afin d'obtenir d'associer les différents comportements.
- il est souhaitable de savoir reconnaître les patrons lorsqu'on en croise un dans un code : permet une compréhension plus rapide du comportement attendu, et donc du fonctionnement de la classe.
- lorsque la virtualité n'est pas souhaitée, certains patrons peuvent être obtenus par l'utilisation de template C++ ("polymorphisme statique").
- lors des entretiens d'embauche, il est fréquent que certaines questions portent sur la connaissance de patrons de classe (singleton, fabrique abstraite, monteur, objet composite, proxy, patron de méthode, visiteur)