4. Généricité avancée

4.1 Introduction aux modèles de conception

Les modèles de conception (design patterns en anglais) sont des abstractions qui aident à structurer les conceptions et les structures des systèmes. Bien qu'ils ne soient pas nouveaux, Ils ont pris de l'importance avec la publication en 1994 du fameux livre "Elements of Reusable Object-Oriented Software" connu sous le nom de "Gang of Four" du nombre de ces auteurs : Erich Gamma, Richard Helm , Ralph Johnson et John Vlissides. L'ouvrage identifie et décrit 23 modèles architecturaux de conception.

Un modèle est un moyen de décrire une solution ou une approche re-productible d'un problème de conception courant (ce n'est pas un algorithme fixe) c'est-à-dire une manière courante de résoudre un problème générique et récurrent. Selon le problème de conception qu'ils traitent, les 23 patrons de conception (GOF) sont classés en trois catégories à savoir :

1. **Modèles de création :** ce sont des modèles qui traitent des mécanismes de création d'objets ; essayant de créer des objets d'une manière adaptée à chaque situation. La forme de base de la création d'objets peut entraîner des problèmes de conception ou augmenter la complexité. Les modèles de création résolvent ces problèmes en contrôlant d'une manière ou d'une autre le processus de création d'objets.
2. **Modèles de structure :**le rôle principal de ce type de modèles est de découpler les interfaces et les implémentations des classes et des objets par une description de la manière dont les objets sont assemblés.
3. **Modèles de comportement :** Ces modèles portent sur la description des algorithmes de comportements entre les différents objets et des formes de communication entre eux.

Les modèles se trouvent couramment dans les langages de programmation orientés objet comme C++ ou Java. Ils peuvent être considérés comme un modèle pour résoudre un problème qui se produit dans de nombreuses situations ou applications différentes. Chaque modèle de conception identifie le problème et/ou le besoin ainsi que la solution abstraite à ce problème. Dans ce qui suit une description détaillée ainsi que des exemples de quelques modèles sont donnés.

4.2 Modèles de création

4.2.1 Builder

— **Problème :** nous voulons construire un objet complexe sans passer par un constructeur complexe (trop d'arguments par exemple).
— **Solution :** définition d'un objet intermédiaire dont les fonctions membres crée l'objet souhaité partie par partie avant que l'objet ne soit disponible pour l'utilisateur.
— **Exemple :**

```
1 #include <iostream>
2 #include <string>
3
4 /* Les parties d'une voiture */
5 class Wheel
6 {
7     public:
8         int size;
9 };
```

```
10 class Engine
11 {
12     public:
13         int horsepower;
14 };
15 class Body
16 {
17     public:
18         std::string shape;
19 };
20 /* Produit final -- Une voiture */
21 class Car
22 {
23     public:
24         Wheel*   wheels[4];
25         Engine*  engine;
26         Body* body;
27 
28         void specifications()
29         {
30             std::cout << "body:" << body->shape << std::endl;
31             std::cout << "engine horsepower:" << engine->horsepower <<
       std::endl;
32             std::cout << "tire size:" << wheels[0]->size << "'" << std
       ::endl;
33         }
34 };
35 /* Builder est une classe abstraite responsable de la construction des
       parties des la voiture */
36 class Builder
37 {
38     public:
39         virtual Wheel* getWheel() = 0;
40         virtual Engine* getEngine() = 0;
41         virtual Body* getBody() = 0;
42 };
43 /* La classe Director est responsable de tout le processus de
       construction */
44 class Director
45 {
46     Builder* builder;
47 
48     public:
49         void setBuilder(Builder* newBuilder)
50         {
51             builder = newBuilder;
52         }
53 
54         Car* getCar()
55         {
56             Car* car = new Car();
57 
58             car->body = builder->getBody();
59 
60             car->engine = builder->getEngine();
61 
```

```
62              car->wheels[0] = builder->getWheel();
63              car->wheels[1] = builder->getWheel();
64              car->wheels[2] = builder->getWheel();
65              car->wheels[3] = builder->getWheel();
66
67              return car;
68          }
69  };
70  /* Un builder concret (non abstrait) pour véhicule Jeep */
71  class JeepBuilder : public Builder
72  {
73      public:
74          Wheel* getWheel()
75          {
76              Wheel* wheel = new Wheel();
77              wheel->size = 22;
78              return wheel;
79          }
80
81          Engine* getEngine()
82          {
83              Engine* engine = new Engine();
84              engine->horsepower = 400;
85              return engine;
86          }
87
88          Body* getBody()
89          {
90              Body* body = new Body();
91              body->shape = "SUV";
92          }
93  };
94  /* Un builder concret (non abstrait) pour véhicule Nissan */
95  class NissanBuilder : public Builder
96  {
97      public:
98          Wheel* getWheel()
99          {
100             Wheel* wheel = new Wheel();
101             wheel->size = 16;
102             return wheel;
103         }
104
105         Engine* getEngine()
106         {
107             Engine* engine = new Engine();
108             engine->horsepower = 85;
109             return engine;
110         }
111
112         Body* getBody()
113         {
114             Body* body = new Body();
115             body->shape = "hatchback";
116         }
117 };
```

```
118 /* Exemple d'utilisation */
119 int main()
120 {
121     Car* car; // Final product
122     /* L'objet director qui contrôle la construction */
123     Director director;
124     /* Des Builders  Concrets */
125     JeepBuilder jeepBuilder;
126     NissanBuilder nissanBuilder;
127     /* Construire un  Jeep */
128     std::cout << "Jeep" << std::endl;
129     director.setBuilder(&jeepBuilder); // using JeepBuilder instance
130     car = director.getCar();
131     car->specifications();
132     std::cout << std::endl;
133     /* Construire une Nissan */
134     std::cout << "Nissan" << std::endl;
135     director.setBuilder(&nissanBuilder); // using NissanBuilder
        instance
136     car = director.getCar();
137     car->specifications();
138     return 0;
139 }
```

4.2.2 Singleton

— **Problème :** on veut s'assurer qu'une classe ne produise qu'une seule instance unique et offre un point d'accès globale à cette instance pour tous ses utilisateurs. Cette situation se présente lorsqu'on a besoin que d'un seul objet qui doit contrôler certaines ressources (serveur unique, sécurité d'accès, etc.).

— **Solution :** La solution peut se résumé en :

1. Définir un attribut statique privé dans la classe "Singleton".
2. Définir une fonction accesseur statique publique dans la classe.
3. procéder à une unique création d'instance à la première demande au niveau de la fonction accesseur.
4. Définir tous les constructeurs comme étant protégés ou privés (les cacher). Les clients ne peuvent utiliser la classe Singleton que pour obtenir une référence de l'instance unique.

— **Exemple :**

```
1 La première étape est de rendre le constructeur de la classe Singleton
     privé:
2 \begin{lstlisting}
3 class MyClass {
4 private:
5     MyClass(){}
6 };
```

Nous avons besoin maintenant d'une instance de cette classe pour pouvoir l'utiliser. Une méthode membre statique doit être utilisée du fait que la classe ne peut pas être

instanciée.

```
#include <iostream>
using namespace std;
class Singleton
{
public:
    static Singleton *getInstance(); // retourne une référence vers l'
    instance
 private:
    Singleton(){}  // constructeur privé
    static Singleton* instance; // instance privée
};

Singleton* Singleton::instance = 0;
Singleton* Singleton::getInstance()
{
    if(!instance) {
        instance = new Singleton();
        cout << "getInstance(): First instance\n";
        return instance;
    }
    else {
        cout << "getInstance(): previous instance\n";
        return instance;
    }
}

int main()
{
    Singleton *s1 = Singleton::getInstance();
    Singleton *s2 = Singleton::getInstance();
    return 0;
}
```

Le résultat de l'exécution du code précédent :

```
getInstance(): First instance
getInstance(): previous instance
```

Afin d'éviter l'utilisation du constructeur par copie et obtenir éventuellement une deuxième copie de l'instance du Singleton par affectation, on déclare un constructeur par copie privé et on surcharge l'opérateur d'affectation toujours dans la partie privée de la classe. Le nouveau code du Singleton sera donc :

```
#include <iostream>
class Singleton
{
public:
    static Singleton& getInstance();

```

```
7  private:
8      Singleton() {std::cout << "Ctor\n";};
9      ~Singleton() {std::cout << "Dtor\n";};
10     Singleton(const Singleton&);
11     const Singleton& operator=(const Singleton&);
12 };
13 Singleton& Singleton::getInstance()
14 {
15     static Singleton instance;
16     return instance;
17 }
18 int main()
19 {
20     Singleton &s1; = Singleton::getInstance();
21     Singleton &s2; = Singleton::getInstance();
22     return 0;
23 }
```

Un autre problème dans la classe Singleton peut surgir en cas de son utilisation par plusieurs threads concurrents, la concurrence peut aboutir à la création de plusieurs instances du Singleton. Pour en remédier, il suffit de rendre la méthode statique de renvoi de l'instance indivisible (section critique). Le code suivant utilise un verrou pour protéger la méthode getInstance() :

```
1  Singleton& Singleton::getInstance()
2  {
3      Mutex mutex;
4      ScopedLock(&mutex;);  // pour libérer le verrou à la sortie
5      static Singleton instance;
6      return instance;
7  }
```

4.2.3 Factory Method

— **Problème :** nous voulons décider, au moment de l'exécution, quel objet doit être créer en fonction d'une configuration ou d'un paramètre de l'application. Lorsque nous écrivons le code, nous ne savons pas quelle classe doit être instancier.

— **Solution :**définition d'une interface (méthode abstraite pure) pour créer un objet, en laissant les sous-classes décider de quelle classe instancier l'objet en question. La méthode Factory permet à une classe de reporter l'instanciation au niveau des sous-classes.

— **Exemple :**

```
1  // Un exemple du modèle de conception "Factory method"
2  #include <iostream>
3  using namespace std;
4  enum VehicleType {
5      VT_TwoWheeler, VT_ThreeWheeler, VT_FourWheeler
6  };
```

```
7  // classes à instancier
8  class Vehicle {
9  public:
10     virtual void printVehicle() = 0;
11     static Vehicle* Create(VehicleType type);
12     virtual ~Vehicle(){}
13 };
14 class TwoWheeler : public Vehicle {
15 public:
16     void printVehicle() {
17         cout << "I am two wheeler" << endl;
18     }
19 };
20 class ThreeWheeler : public Vehicle {
21 public:
22     void printVehicle() {
23         cout << "I am three wheeler" << endl;
24     }
25 };
26 class FourWheeler : public Vehicle {
27     public:
28     void printVehicle() {
29         cout << "I am four wheeler" << endl;
30     }
31 };
32
33 // Ici la méthode Factory pour créer des objets de différents types.
34 // Les changements sont requises uniquement dans cette classe pour
35 // créer de nouveaux types d'objets
36 Vehicle* Vehicle::Create(VehicleType type) {
37     if (type == VT_TwoWheeler)
38         return new TwoWheeler();
39     else if (type == VT_ThreeWheeler)
40         return new ThreeWheeler();
41     else if (type == VT_FourWheeler)
42         return new FourWheeler();
43     else return NULL;
44 }
45 // Classe qui utilise la factory method
46 class Client {
47 public:
48     // Le client ne crée pas directement d'objets
49     // il passe le type d'objet voulu à la factory method "Create()"
50     Client()
51     {
52         VehicleType vtype = VT_ThreeWheeler;
53         pVehicle = Vehicle::Create(vtype);
54     }
55     ~Client() {
56         if (pVehicle) {
57             delete pVehicle;
58             pVehicle = NULL;
59         }
60     }
61     Vehicle* getVehicle() {
62         return pVehicle;
```

```
63      }
64  private:
65      Vehicle *pVehicle;
66  };
67  // Programme de test
68  int main() {
69      Client *pClient = new Client();
70      Vehicle * pVehicle = pClient->getVehicle();
71      pVehicle->printVehicle();
72      delete pClient;
73      return 0;
74  }
```

4.2.4 Abstract Factory

— **Problème :** idem que celui de la "Factory Method" mais avec plusieurs familles de classes produits.

— **Solution :** mettre en oeuvre plusieurs classes avec une "Factory Method" héritée à partir d'une classe abstraite commune (Abstract Factory).

4.2.5 Prototype

— **Problème :** lorsque la création d'un objet est coûteuse et complexe et on veut créer rapidement une copie de test à partir d'un modèle dit prototype dont la création est rapide et moins coûteuse.

— **Solution :** déclarer une classe abstraite de base qui spécifie une méthode pure virtuelle clone(). Toute classe qui a besoin d'une capacité de ce "constructeur polymorphe" dérive elle-même de la classe de base abstraite et implémente l'opération clone(). Le clonage se fait sur mesure à des fins de test (prototypage).

4.3 Modèles de structure

4.3.1 Adapter

— **Problème :** faire coopérer des classes dont les interfaces (méthodes) sont incompatibles.

— **Solution :** convertir l'interface d'utilisation (méthodes publiques) de la classe à adapter à une autre interface attendue par ses utilisateurs.

— **Exemple :**

```
1  #include <iostream>
2  // L'interface désirée (Cible)
3  class Rectangle
4  {
5    public:
6      virtual void dessiner() = 0;
7  };
8
9  // Ancien Composant non adapté (L'Adaptee)
10 class AncienRectangle
11 {
```

```
12    public:
13      AncienRectangle(int x1, int y1, int x2, int y2) {
14          x1_ = x1;
15          y1_ = y1;
16          x2_ = x2;
17          y2_ = y2;
18          std::cout << "AncienRectangle(x1,y1,x2,y2)\n";
19      }
20      void ancienDessiner() {
21          std::cout << "AncienRectangle:  oldDraw(). \n";
22      }
23    private:
24      int x1_;
25      int y1_;
26      int x2_;
27      int y2_;
28  };
29
30  // L'interface d'adaptation (Adaptateur)
31  class RectangleAdapter: public Rectangle, private AncienRectangle
32  {
33    public:
34      RectangleAdapter(int x, int y, int w, int h):AncienRectangle(x, y,
       x + w, y + h) {
35            std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
36        }
37
38      void dessiner() {
39          std::cout << "RectangleAdapter: dessiner().\n";
40          ancienDessiner();
41      }
42  };
43
44  int main()
45  {
46    int x = 20, y = 50, w = 300, h = 200;
47    Rectangle *r = new RectangleAdapter(x,y,w,h);
48    r->dessiner();
49  }
```

4.3.2 Bridge

— **Problème :** on veut séparer l'interface d'un objet de son implémentation pour que les deux varient indépendamment.

— **Solution :** utiliser les mêmes signatures des méthodes d'utilisation (ponts) dans les classes concrètes afin d'uniformiser l'utilisation des différents objets implémentation.

— **Exemple :**

```
1  #include <iostream>
2  using namespace std;
3  // Les différentes Implementions
4  class DrawingImplementor { // classe abstraite racine des implé
```

```
    mentations
public:
    virtual void drawSquare(double) = 0;
     virtual ~DrawingImplementor() {
    }
};
 // Implémentation Concrète A
class DrawingImplementorA: public DrawingImplementor {
public:
    DrawingImplementorA() {
    }
     virtual ~DrawingImplementorA() {
    }
 // exemple : dessiner avec un pinceau
    void drawSquare(double side) {
        cout << "\nImplementorA.square avec face = "<< side << endl;
    }
};

// Implémentation concrète B
class DrawingImplementorB: public DrawingImplementor {
public:
    DrawingImplementorB() {
    }

    virtual ~DrawingImplementorB() {
    }

    // exemple: dessiner avec un crayon
    void drawSquare(double side) {
        cout << "\nImplementorB.square avec face = "<< side << endl;
    }
};

// Classe abstraite de formes
class Shape {
public:
    virtual void draw()= 0; // low-level
    virtual void resize(double pct) = 0; // high-level
    virtual ~Shape() {
    }
};

// Classe concrète Square
class Square: public Shape {
public:
    Square(double s, DrawingImplementor& Implementor) :
        side(s), drawingImplementor(Implementor) {
    }

    virtual ~Square() {
    }

    // low-level i.e. Implementation specifique
    void draw() {
        drawingImplementor.drawSquare(side);
```

```
60      }
61      // high-level i.e. Abstraction specifique
62      void resize(double pct) {
63          side *= pct;
64      }
65  private:
66      double side;
67      DrawingImplementor& drawingImplementor;
68  };
69
70  int main(int argc, char* argv[]) {
71      DrawingImplementorA ImplementorA;
72      DrawingImplementorB ImplementorB;
73
74      Square sqA(1, ImplementorA);
75      Square sqB(2, ImplementorB);
76
77      Shape* shapes[2];
78      shapes[0] = &sqA;
79      shapes[1] = &sqB;
80
81      shapes[0]->resize(10);
82      shapes[0]->draw();
83      shapes[1]->resize(10);
84      shapes[1]->draw();
85
86      return 0;
87  }
```

4.3.3 Composite

— **Problème :** composer, par agrégation, plusieurs objets hiérarchiques d'une manière uniforme.
— **Solution :** mettre en œuvre une classe abstraite de tous les composants (objets simples appelés feuilles et composés) avec une interface pour le comportement par défaut. Les Feuilles qui sont des objets n'ayant pas de sous-éléments doivent implémenter le comportement par défaut et le Composite représente un objet pouvant avoir des sous-éléments et stocke des composants enfants afin d'y accéder (ajout, suppression et accès aux feuilles). Il doit implémenter un comportement en utilisant ses enfants (Feuilles).

4.3.4 Decorator

— **Problème :** attacher dynamiquement des responsabilités (comportements) additionnelles à un objet.
— **Solution :** utiliser l'agrégation pour rattacher dynamiquement un objet décorateur à un autre objet qui peut exécuter une tâche supplémentaire par le biais du décorateur rattaché, le décorateur peut être détacher de l'objet concerné.

4.3.5 Facade

— **Problème :** cacher la complexité d'un système en offrant au client une interface uniforme d'utilisation indépendante du sous-système adjacent utilisé.
— **Solution :** l'objet "Facade" doit définir une interface de haut niveau qui facilite l'utilisation du sous-système. Par exemple, faire en sorte qu'une méthode de classe exécute un processus complexe en appelant plusieurs autres classes.

4.3.6 Flyweight

— **Problème :** lorsque un grand nombre d'objets sont définis dans un système et ont une bonne partie commune, on veut optimiser leur utilisation (principalement l'occupation en mémoire).
— **Solution :** déplacer les propriétés communes à ces objets vers une structure de données externe et fournir à chaque objet le lien vers cette structure de données.

4.3.7 Proxy

— **Problème :** on veut cacher un objet sensible par un autre objet substitut (le substitut est appelé proxy) pour contrôler l'accès à l'objet vulnérable ou complexe.
— **Solution :** mettre on oeuvre un objet substitut qui contrôle l'appel des méthodes de l'objet qu'on veut cacher. Le proxy gérera les requêtes des utilisateurs en utilisant l'objet caché de manière adéquate.

4.4 Modèles de comportement

4.4.1 Chain of Responsibility

— **Problème :** on veut éviter de coupler l'expéditeur d'une requête avec son destinataire en donnant à plusieurs objets la possibilité de traiter la demande.
— **Solution :** enchaîner les objets récepteurs et transmettre les requêtes le long de la chaîne jusqu'à ce qu'un objet adéquat la traite.
— **Exemple :**

```
#include <iostream>
using namespace std;
class Handler {
    protected:
        Handler *next;
    public:
        Handler() {
            next = NULL;
        }
        virtual ~Handler() { }
        virtual void request(int value) = 0;
        void setNextHandler(Handler *nextInLine) {
            next = nextInLine;
        }
};
class SpecialHandler : public Handler {
    private:
        int myLimit;
        int myId;
```

```
20      public:
21          SpecialHandler(int limit, int id) {
22              myLimit = limit;
23              myId = id;
24          }
25          ~SpecialHandler() { }
26          void request(int value) {
27              if(value < myLimit) {
28                  cout << "Handler " << myId << " handled the request
    with a limit of " << myLimit << endl;
29              } else if(next != NULL) {
30                  next->request(value);
31              } else {
32                  cout << "Sorry, I am the last handler (" << myId << ")
    and I can't handle the request." << endl;
33              }
34          }
35  };
36  int main () {
37      Handler *h1 = new SpecialHandler(10, 1);
38      Handler *h2 = new SpecialHandler(20, 2);
39      Handler *h3 = new SpecialHandler(30, 3);
40      h1->setNextHandler(h2);
41      h2->setNextHandler(h3);
42      h1->request(18);
43      h1->request(40);
44      delete h1;
45      delete h2;
46      delete h3;
47      return 0;
48  }
```

4.4.2 Observer

— **Problème :** à un ou plusieurs endroits de l'application, nous voulons qu'un ensemble d'objets, appelés observateurs et qui se sont abonnés au préalable à un évènement, soient informés du changement d'état d'un objet, appelé l'observé, qui corresponds à cet évènement. Il devrait également y avoir un moyen de se désabonner de l'évènement pour un observateur.

— **Solution :** mettre en œuvre une interface comportant une méthode abstraite de traitement de l'évènement, les observateurs doivent implémenter cette interface. D'autre part, l'observé doit mettre à la disposition des observateurs les méthodes d'abonnement et de désabonnement à l'évènement. Au changement d'état nécessitant une notification, l'observé itère sur tous les observateurs abonnés en invoquant leur méthode de traitement implémentée.

— **Exemple :**

```
1  #include <iostream>
2  #include <list>
3  #include <string>
4  // Interface à implémenter par les observateurs
5  class IObserver {
```

```
6   public:
7     virtual ~IObserver(){};
8     virtual void Update(const std::string &message_from_subject) = 0;
9   };
10
11  // Interface de l'observé ==> Observateur abstrait
12  class ISubject {
13   private:
14    std::list<IObserver *> list_observer_;
15    std::string message_;
16   public:
17    virtual ~ISubject(){};
18    virtual void Attach(IObserver *observer) = 0;
19    virtual void Detach(IObserver *observer) = 0;
20    virtual void Notify() = 0;
21  };
22  // L'observé ==> Subject
23  class Subject : public ISubject {
24   public:
25    virtual ~Subject() {
26      std::cout << "Goodbye, I was the Subject.\n";
27    }
28   // méthode pour abonner un observateur
29    void Attach(IObserver *observer) override {
30      list_observer_.push_back(observer); // mettre l'observateur dans
       une liste
31    }
32    // méthode de désabonnement
33    void Detach(IObserver *observer) override {
34      list_observer_.remove(observer); // retirer l'observateur de la
       liste
35    }
36    // méthode de notification de l'évènement
37    void Notify() override {
38      std::list<IObserver *>::iterator iterator = list_observer_.begin();
39      HowManyObserver();
40      while (iterator != list_observer_.end()) {
41        (*iterator)->Update(message_);
42        ++iterator;
43      }
44    }
45    void CreateMessage(std::string message = "Empty") {
46      this->message_ = message;
47      Notify();
48    }
49    void HowManyObserver() {
50      std::cout << "There are " << list_observer_.size() << " observers
       in the list.\n";
51    }
52    void stateChange() {
53      this->message_ = "change message message";
54      Notify();
55      std::cout << "I'm about to do some thing important\n";
56    }
57  };
58  // Un obeservateur Concret
```

```
59 class Observer : public IObserver {
60  public:
61   Observer(Subject &subject) : subject_(subject) {
62     this->subject_.Attach(this);
63     std::cout << "Hi, I'm the Observer \"" << ++Observer::
     static_number_ << "\".\n";
64     this->number_ = Observer::static_number_;
65   }
66   virtual ~Observer() {
67     std::cout << "Goodbye, I was the Observer \"" << this->number_ << "
     \".\n";
68   }
69   void Update(const std::string &message_from_subject) override {
70     message_from_subject_ = message_from_subject;
71     PrintInfo();
72   }
73   void RemoveMeFromTheList() {
74     subject_.Detach(this);
75     std::cout << "Observer \"" << number_ << "\" removed from the list
     .\n";
76   }
77   void PrintInfo() {
78     std::cout << "Observer \"" << this->number_ << "\": a new message
     is available --> " << this->message_from_subject_ << "\n";
79   }
80  private:
81   std::string message_from_subject_;
82   Subject &subject_;
83   static int static_number_;
84   int number_;
85 };
86 int Observer::static_number_ = 0;// initialisation d'un membre de donné
     es statique
87 void ClientCode() {
88   Subject *subject = new Subject;
89   Observer *observer1 = new Observer(*subject);
90   Observer *observer2 = new Observer(*subject);
91   Observer *observer3 = new Observer(*subject);
92   Observer *observer4;
93   Observer *observer5;
94   subject->CreateMessage("Hello World! :D");
95   observer3->RemoveMeFromTheList();
96   subject->CreateMessage("The weather is hot today! :p");
97   observer4 = new Observer(*subject);
98   observer2->RemoveMeFromTheList();
99   observer5 = new Observer(*subject);
100
101  subject->CreateMessage("My new car is great! ;)");
102  observer5->RemoveMeFromTheList();
103
104  observer4->RemoveMeFromTheList();
105  observer1->RemoveMeFromTheList();
106
107  delete observer5;
108  delete observer4;
109  delete observer3;
```

```
110     delete observer2;
111     delete observer1;
112     delete subject;
113 }
114
115 int main() {
116     ClientCode();
117     return 0;
118 }
```

4.4.3 Command

— **Problème :** dissocie l'expéditeur d'un service de son destinataire en encapsulant la demande en tant qu'un objet utilisable à tout moment comme une commande permanente.

— **Solution :** mettre en œuvre un mécanisme orienté objet similaire aux méthodes de rappel (callback) en utilisant des références sur les objets qui représentent les différentes commandes. Ces références seront stockées au niveau de l'objet receveur qui peut les déclencher à tout moment (via une méthode uniformisée).

4.4.4 Interpreter

— **Problème :** il s'agit d'une classe de problèmes qui se produisent de manière répétée dans un domaine bien défini et bien cerné. Si le domaine est caractérisé par une "langue", alors les problèmes pourraient être facilement résolus avec un "moteur" d'interprétation de ladite langue.

— **Solution :** représenter la grammaire du langage du domaine sous forme hiérarchique d'objets.

4.4.5 Iterator

— **Problème :** on veut parcourir des listes ou des collections d'objets mais de façon abstraite (sans se préoccuper du détail des objets parcourus). Ainsi, on peut définir des algorithmes uniformes applicables sur tout type de listes ou de collections.

— **Solution :** encapsulation des éléments de la collection par un niveau d'abstraction (Iterator). L'accès aux éléments se fait exclusivement par le biais des itérateurs. En pratique, Chaque classe Collection stocke un objet de la classe Iterator adéquate dans la hiérarchie des itérateurs. A l'instanciation d'un objet collection, son itérateur est instancié et renvoyé aux utilisateurs au besoin.

4.4.6 Mediator

— **Problème :** on veut concevoir des composants réutilisables, sans être confronter au phénomène de "code spaghetti" (essayer de récupérer une seule portion entraîne un "bloc tout ou rien").

— **Solution :** définition d'un objet qui encapsule la manière dont un ensemble d'objets interagissent. L'objet "Mediator" favorise le découplage en empêchant les objets de se référer explicitement les uns aux autres et il permet de faire varier leur interaction indépendamment. Pratiquement, le "Mediator" maintient une liste dynamique de

pairs d'objets qui devront interagir, de cette manière les objets seront faiblement couplés et peuvent être réutilisés avec une certaine aisance.

4.4.7 Memento

— **Problème :** on veut, à un instant donné, capturer et externaliser l'état interne d'un objet afin que l'objet puisse revenir ultérieurement à cet état (sauvegarde d'état), et ce sans violer le principe d'encapsulation.
— **Solution :** création d'une classe "Memento" (avec la structure adéquate d'état à sauvegarder), déclarer la classe d'objets à sauvegarder comme amie dans le Memento et enfin, Déléguer l'initiation de la sauvegarde et la restauration à un objet tiers (Un gardien du Memento). A la demande de ce dernier, l'objet à sauvegarder crée un objet mémento et copie son état dans ce mémento.

4.4.8 State

— **Problème :** on veut que le changement d'état d'un objet donné se fait indépendamment de l'objet lui même, Ainsi le changement de comportement se fait sans couplage fort avec l'état de l'objet.
— **Solution :** exploitation du polymorphisme ; concevoir une classe abstraite d'état, identifier tous les états possibles de l'objet en question et concevoir les classes dérivées adéquates (qui héritent toutes de la classe abstraite d'état). L'objet ne fait que stocker une référence de type état, l'état réel de l'objet à l'exécution est pointé par cette référence.

4.4.9 Strategy

— **Problème :** pour des besoins de flexibilité, on veut définir une famille d'algorithmes (méthodes), les stocker sous forme d'objets indépendants et les rendre interchangeables entre eux. Les objets peuvent les utiliser d'une manière transparente et uniforme.
— **Solution :** identifier un algorithme (c'est-à-dire un comportement) auquel l'utilisateur préférerait accéder via un "point flexible", spécifier la signature de cet algorithme dans une interface (ou classe abstraite) et Encapsuler le détail des différentes implémentations alternatives dans des classes dérivées. Les utilisateurs de l'algorithme manipulent ces différentes implémentations par le biais de l'interface (pour l'uniformité de l'accès).

4.4.10 Template Method

— **Problème :** on a un squelette d'un algorithme et on veut travailler avec plusieurs versions de cet algorithme où chaque version définie réalise une sous-opérations de l'algorithme et laisse les autres sous forme générique.
— **Solution :** exploiter le polymorphisme et la généricité ; déclarer une méthode générique dans une classe abstraite de base et réaliser des spécialisation partielles dans des classes dérivées représentant différente implémentations de la "template method"

4.4.11 Visitor

— **Problème :** on a une structure comportant plusieurs objets hétérogènes agrégés et on veut effectuer une opération sur ces objets sans "polluer" la classe de la structure par cette opération.

— **Solution :** Utiliser un objet (Visitor) comportant l'opération en question, ajouter à la classe de la structure une méthode d'acceptation qui reçoit l'objet visiteur en paramètre et invoque l'opération (du visiteur) sur la structure (éventuellement référencée par "this" ou "*this").

4.5 Conclusion

Les design patterns sont des recettes à utiliser dans les applications afin de mettre en œuvre des solutions rapides et surtout de qualité respectant les principes de l'approche orientée objet et aboutissant à des codes clairs, extensibles et réutilisables.