Formation C++ Ubisoft - Module 3

Romain Arcila[1,2]
Charles de Rousiers[1]

29 mars 2009

[1] Inria Grenoble
[2] Liris -CNRS Lyon

Plan

Séance du jour

Les Designs Paterns

Plan

Design Pattern

- Principe de programmation robuste, idée, voire bloc de code couramment utilisé.
- Regrouper par famille.
- Chaque famille et chaque design pattern sert à résoudre un problème particulier.
- Pattern principaux présentés dans le livre de Gamma et al. : Design Pattern (Gang of Four).
- Pattern plus récent : MVC, Signal/Slot...

Design Pattern - Réponse à tout ?

Réponse à tout ?
Non

Design Pattern - Réponse à tout ?

- Chaque pattern répond à un problème donné.
- Un pattern mal utilisé pose plus de problème.
- Probléme de conception

Design Pattern - Conception

- Conception : réfléchir au problème.
- Identifier le cas échéant le pattern associé.
- Exprimer le problème sous forme de pattern

Problème

- Utiliser un pattern quand ce n'est pas nécessaire.
- Essayer de "patterniser" : anti-pattern.
- Barrière pattern/idioms.

Famille de design pattern

- **Création** : instanciation d'objets.
- **Structure** : composition d'objet.
- **Comportement** : communication entre les objets.

Plan

Création

Design pattern permettant d'instancier des objets :

- Singleton
- Factory method
- Prototype
- Abstract Factory
- Builder

Creational Pattern

Abstraction de l'instanciation.

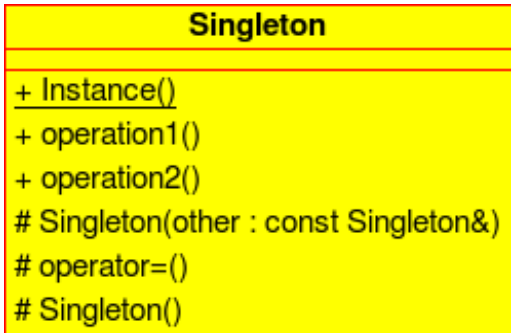
- Nécessaire lorsqu'un logiciel devient complexe.
- Cacher les classes réellement instanciée.
- Cacher les processus de créations.

Singleton - Principe

- Assurer qu'une seule instance existe (variante : maximum X instance).
- Facile à implémenter hormis pour la gestion multi-thread.
- Plusieurs implémentations existe.
- $\neq$ Classe avec variables statiques publiques : Multithread.

En C++, préférer le singleton aux variables (membres et globales) statiques. (pour plusieurs raisons)

Singleton - Schema



Exemple

Gestionnaire de configuration :

- Un seul gestionnaire.
- Pas de lecture et modification simultanée (mutex)

Code : Configuration

```
class Configuration {
  // Singleton part
  mutable static map<string,string> registry;
 public:
  void setValue(const string& key, const string& value);
  string& getValue(const string& key) const;
  void dump(const char* path);
  void read(const char* path);
};
```

Implémentation Interface

Implémentation Interface

```
class C {
  protected:
   C(ARGS) {...}
   C(const C& other); // no implementation needed
   C& operator=(const C& other); // no implementation needed
  public:
   static C& Instance(ARGS); // static method
   // C methods
};
```

Implémentation I

Implémentation Interface

```
C::C(ARGS) {...}

C& C::Instance(ARGS) {
  static C instance(ARGS);
  return instance;
}
```

Implémentation I

Implémentation Interface

```
namespace {
  C *instance=NULL; // note pointer
  void clear() {
    delete instance;
  }
}

C::C(ARGS) {...}

C& C::Instance(ARGS) {
  if (!instance) {
    atexit(clear);
    instance=new C(ARGS);
  }
  return *instance;
}
```

Singleton - Implémentation C++

Code : Singleton

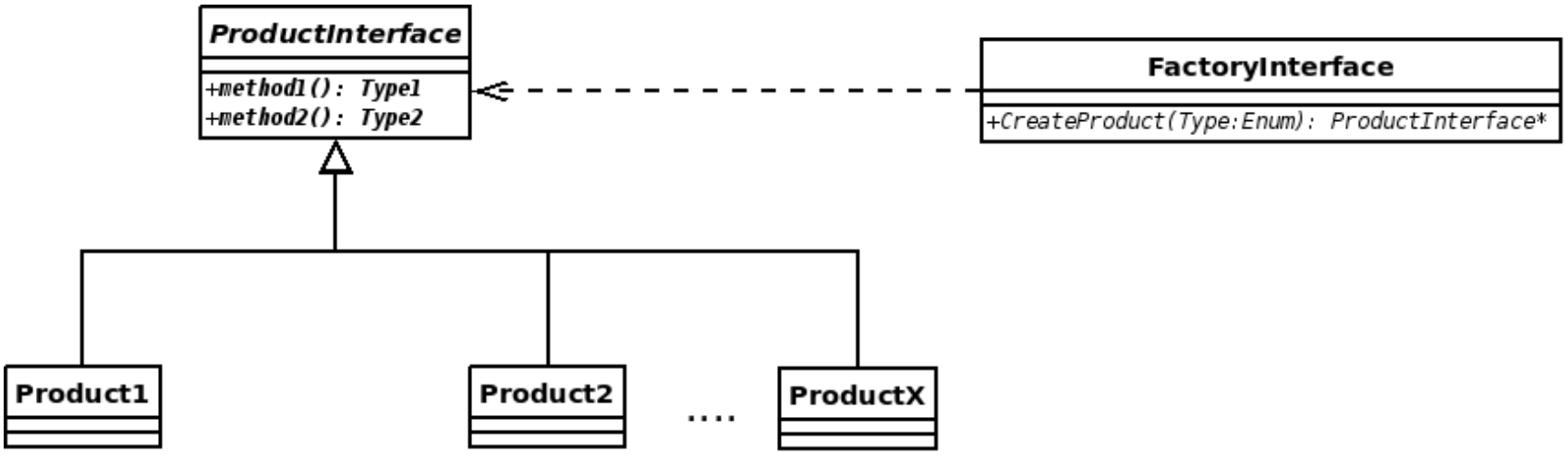
```
template<typename T>
class SingletonMixin {
  public:
    static T& Instance() {
      _mutex.lock();
      static T _instance;
      _mutex.unlock();
      return _instance;
    }
  private:
// Admit that we have a mutex class
    static Mutex _mutex;
};

template<class T>
class NonCopyable {
  typedef NonCopyable<T> Self
  protected:
   NonCopyable<T>() {}
   NonCopyable<T>(const Self&);
   NonCopyable<T>& operator=(const Self&);
};
```

```
template<class T>
Mutex SingletonMixin<T>::_mutex;

class Unique: public SingletonMixin<Unique>,
              private NonCopyable<Unique> {

    friend class SingletonMixin<Unique>;
  protected:
    Unique() {...}

  public:
    void print() const {
      cout << this << endl;
    }
};

Unique& unique=Unique::Instance();
Unique& unique2=Unique::Instance();
unique.print();
unique2.print();
```

Factory

- Concentrer la création d'objet.
- Souvent une classe ou fonction (singleton) responsable.
- La factory décide de l'objet à instancier.

Factory - Schema



Exemple

Exemple : Factory

```
struct Primitive { virtual void intersect(...)=0; };
struct Cube: public Primitive { void intersect(...) {cout << "Cube" << endl;} };
struct Sphere: public Primitive { void intersect(...) {cout << "Sphere" << endl;} };
struct Mesh: public Primitive { void intersect(...) {cout << "Mesh" << endl;} };

struct Factory {
  enum PrimitiveID {cube, sphere, mesh};
  static Primitive* Instanciate(const PrimitiveID& primitive) {
    switch (primitive) {
    case cube:
      return new Cube();
    case sphere:
      return new Sphere();
    case mesh:
      return new Mesh();
    default :
      assert(false );
    }
  }
};

int main() {
  shared_ptr<Primitive> primitive(Factory::Instanciate(Factory::cube));
  primitive->intersect();
}
```

Implémentation Standard

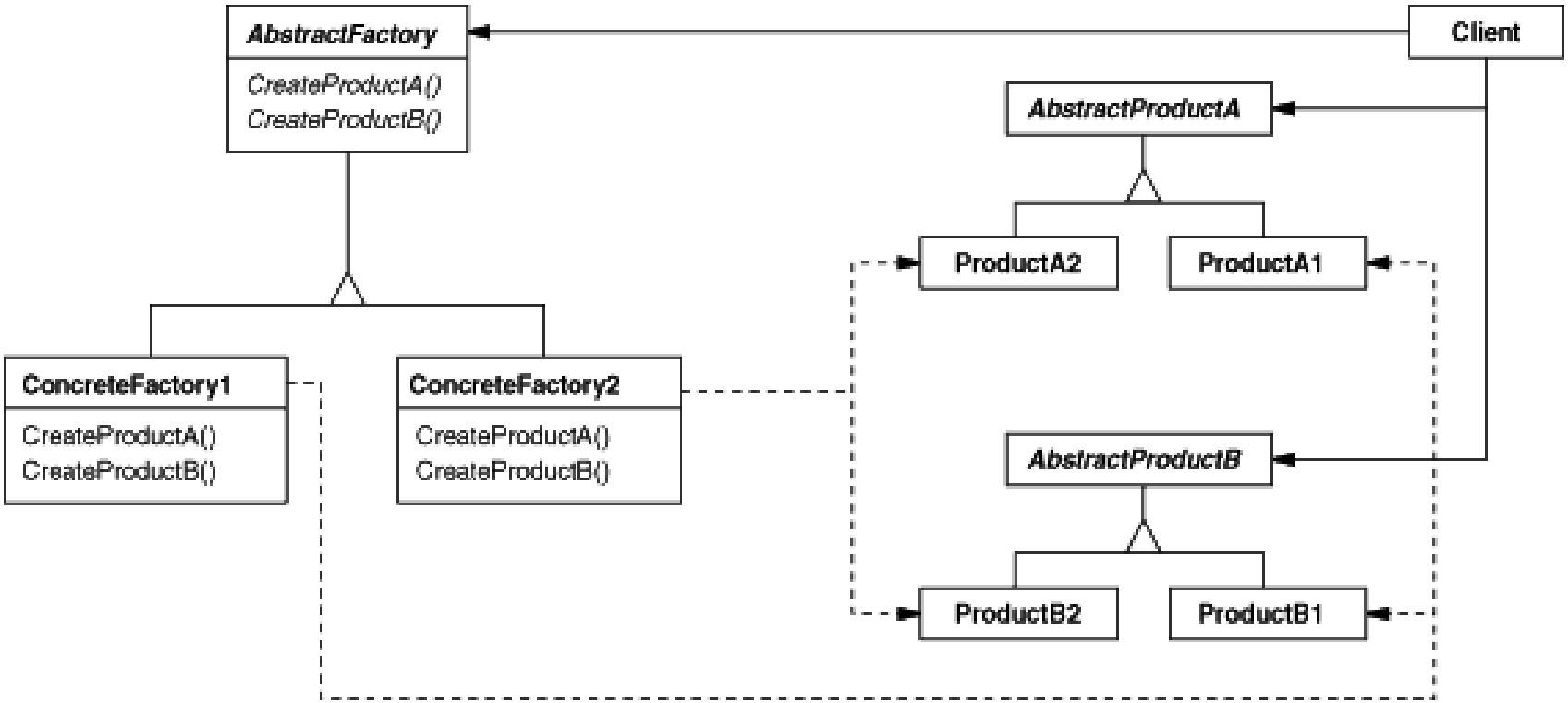
Implémentation

```
class Mother;
class D1: public Mother;
class D2: public Mother;
...
class MotherFactory {
  enum Type {d1,d2...}
   static Mother* Instanciate(const Type& t)) {
     switch(t){
       case d1:
         return new D1;
       case d2:
         return new D2;
        ...
     }
   }
};
```

Abstract Factory

Fournir une interface pour créer des familler d'objets ou d'objet dépendant sans spécifier leur classes concrètes.

Abstract Factory - Schema



Exemple

Toolkit Graphics themable :

- Plusieurs widgets : fenetre, ascenseur..
- Plusieurs themes : osx, win, kde...

(Creer et) Manipuler les widgets sans se soucier de leur theme (et de leur type).

Abstract Factory - Implémentation

Abstract Factory

```
struct Window { };
struct WinWindow: public Window { };
struct OSXWindow: public Window { };
```

```
struct Slider {};
struct WinSlider: public Slider { };
struct OSXSlider: public Slider { };
```

```
struct AbstractFactory {
  virtual Window* createWindow() const =0;
  virtual Slider* createSlider() const =0;
};

struct WinFactory: public AbstractFactory {
  Window* createWindow() const {return new WinWindow(); }
  Slider* createSlider() const {return new WinSlider(); }
};
struct OSXFactory: public AbstractFactory {
  Window* createWindow() const {return new OSXWindow(); }
  Slider* createSlider() const {return new OSXSlider(); }
};
```

Abstract Factory - Implémentation

Abstract Factory

```
void UseFactory(const AbstractFactory* fact) {
  Window *win=fact->createWindow();
  Slider *sl=fact->createSlider(win);
  // draw slider and window
}

string theme="OSX";
AbstractFactory* f;
if (theme=="OSX")
  f=new OSXFactory;
else
  f=new WinFactory;

UseFactory(f);
```

Implémentation

Abstract Factory

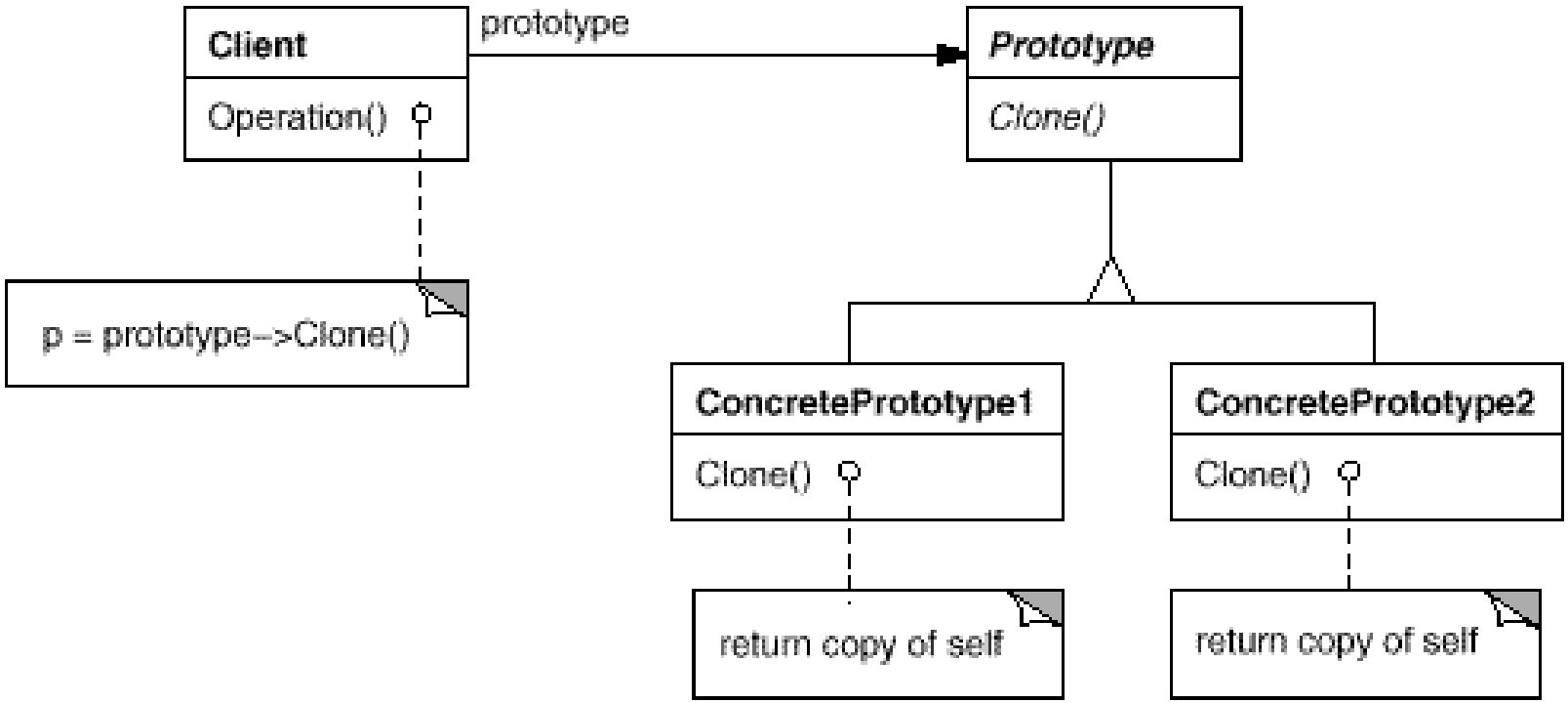
```
class A {};
class A1: public A {};
class A2: public A {};
class B {};
class B1: public B {};
class B2: public B {};
...
struct AbsFact {
  virtual A* createA()=0;
  virtual B* createB()=0;
  ...
};
struct Fact1: public AbsFact {
  A* createA() { return new A1(); }
  B* createB() { return new B1(); }
};
struct Fact2: public AbsFact {
  A* createA() { return new A2(); }
  B* createB() { return new B2(); }
};
...
```

Prototype

Instancier un objet a partir d'un autre objet :

- Proche de la factory method.
- Copier un objet sans savoir son type.
- A utiliser lorsque créer un objet coute cher (connection) : dupliquer puis modifier.
- Objet ajouter à l'execution.

Prototype - Schema



Prototype - Implémentation

Prototype

```
class Object { virtual Object *clone()=0 const; };
class Car: public Object { Object* clone() const { return new(*this);} };
class Bus: public Object { Object* clone() const { return new(*this);} };

struct Prototype {
  enum Vehicule {CAR, BUS};
  Prototype() {
    dict[CAR]= new Car;
    dict[BUS]= new Bus;
  }
  ~Prototype {
    delete dict[CAR];
    delete dict[BUS];
  }
  Object* Instanciate(Vehicule v) { return dict[v]->clone(); }
 private :
  map<Vehicule, Object*> dict;
};
```

Prototype - Implémentation

Abstract Factory

```
class Object { virtual Object *clone()=0 const; };
class A: public Object { Object* clone() const { return new(*this);} };
class B: public Object { Object* clone() const { return new(*this);} };
...
struct Prototype {
  enum ENUM {EA, EB...};
  Prototype() {
    dict[EA]= new A;
    dict[EB]= new B;
    ..
  }
  ~Prototype {
    delete dict[EA];
    delete dict[EB];
    ...
  }
  Object* Instanciate(ENUM e) {
    return dict[e]->clone();
  }
 private :
   map<ENUM, Object*> dict;
};
```

Clone - Schema

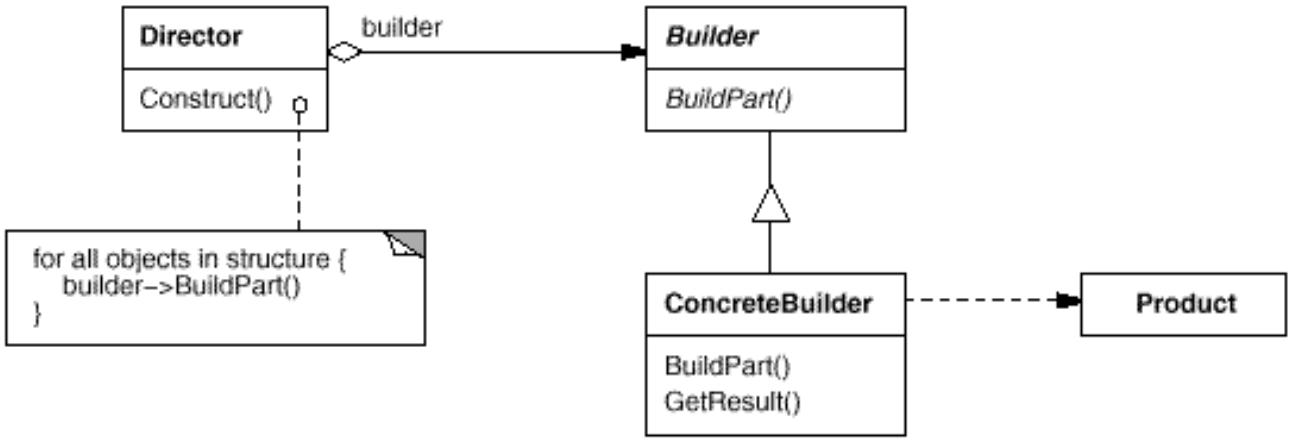
Repose sur l'idiom clone.

Builder

Séparer la création de l'objet de sa création :

- Permet de construire plus simplement un objet complexe en procédent par étape.
- Permet de spécifier plusieurs constructions.
- Permet de constuire de la même manière des implémentations différentes.

Builder - Schema



Builder - Exemple

Builder Factory

```
struct Scene {
  void addPrimitive(Primitive* p);
  void addLight(Light* l);
};

class SceneBuilder {
  protected:
    Scene* sc;
    string path;
  public:
    SceneBuilder(const string& p): path(p), sc(new Scene) {}
     virtual void buildPrimitive()=0;
     virtual void buildLight()=0;
    Scene* getScene() { return sc; }
};

struct SceneBuilderXML: public SceneBuilder {
  SceneBuilderXML(const string& p): SceneBuilder(p) { /* open XML file and parse */ }
   virtual void buildPrimitive() { Primitive *p=PrimFromXML(); sc->addPrimitive(p); }
   virtual void buildLight() { Light *l=LightFromXML(); sc->addLight(p); }
};

// same for other formats.
```

Builder - Exemple

Builder Factory

```
class Director {
    SceneBuilder* builder;
  public:
    void setBuilder(SceneBuilder* b) {builder=b;}
    Scene* getScene() {return builder->getScene();}
    void construct() {
      builder->buildPrimitive();
      builder->buildLight();
      builder->buildTexture();
    }
};

Director d;
switch (path.ext) {
  case XML:
    d.setBuilder(new SceneBuilderXML(path));
    break;
  case F1:
    d.setBuilder(new SceneBuilderF1(path));
    break;
  ...
}
d.construct();
renderScene(d.getScene());
```

Builder - Exemple

Builder Factory

```
struct ComplexProduct {
  void addPart1(Part1* p);
  void addPartX(Part2* l);
};
struct ProductBuilder {
    ComplexProduct* p;
    ProductBuilder(): p(new ComplexProduct) {}
     virtual  void buildP1()=0;
     virtual  void buildPX()=0;
    ComplexProduct* getProduct() { return p; }
};
struct PBuilder1: public ProductBuilder {
   virtual  void buildP1() {// build P1; add it to p;}
   virtual  void buildPX() {// build PX; add it to p;}
};
class Director {
    ProductBuilder* builder;
  public:
    void setBuilder(ProductBuilder* b) {builder=b;}
    Product getProduct() {return builder->p;}
    void construct() {
      builder->buildP1();
      builder->buildPX();
    }
};
```

Plan

Structure

- adapter
- bridge
- composite
- decorator
- facade
- flyweight
- proxy

Structural Pattern

- Composition pour former des classes complexes à partir de classes simples.

Mais avant... Délégation

- Délégation : méthode très utilisée.
- Consiste à relayer une métode à une classe interne.
- Pas vraiment un pattern

Mais avant... Délégation

Code : Delegation

```
struct A {
  void print() {cout << "A::print" << endl;}
};

class B {
   A* a;
 public :
   B(A* _a): a(_a) {}
   void print() { a->print(); }
   void affiche() { a->print(); }
};
```

Adapter

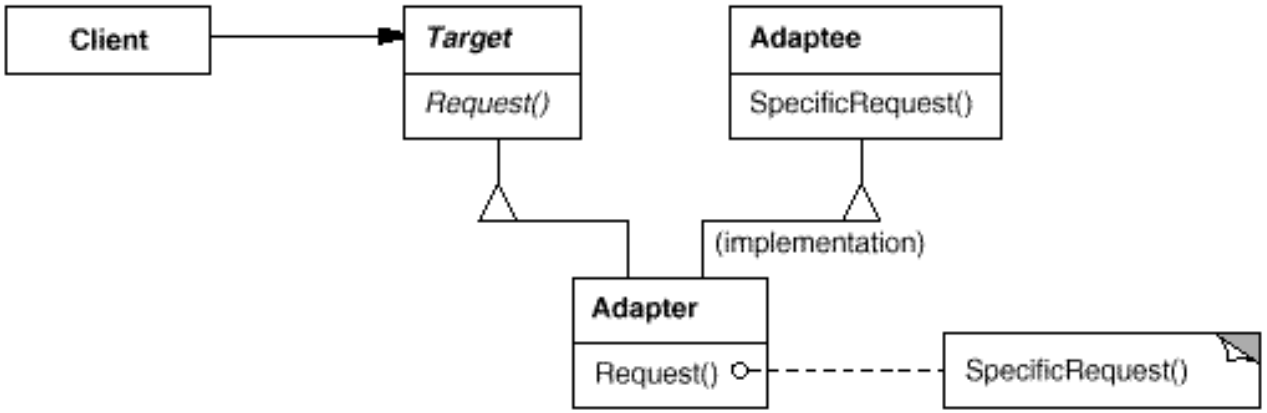
Principe :

- Une interface existance, fonctionnalité manquante.
- Fonctionnalité existance ailleurs avec une interface différente

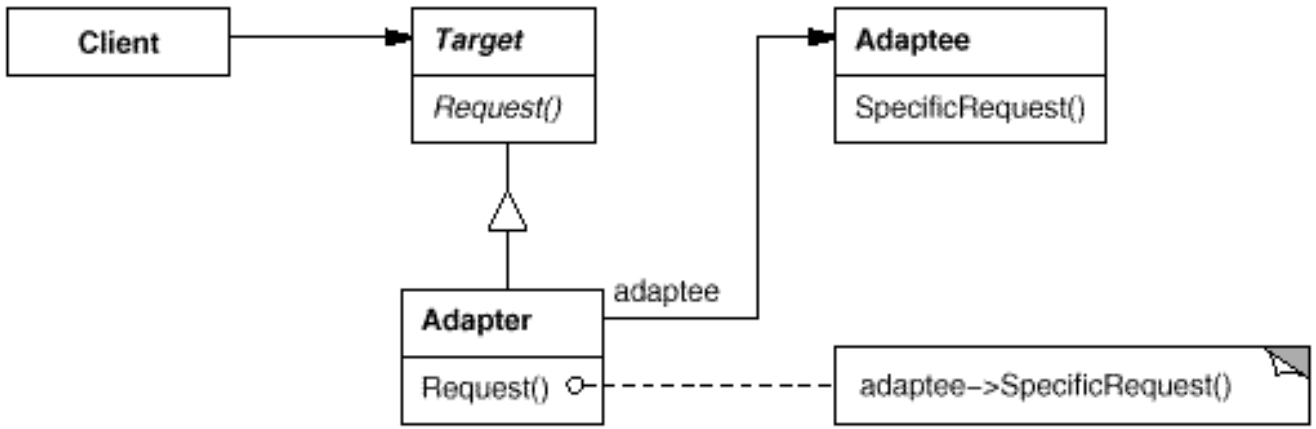
$\Rightarrow$ Adapter la fonctionnalité à l'interface par :

- héritage multiple
- composition

Adapter - Schema



Adapter - Schema



Adapter - Exemple

Adapter - Exemple

```
class Primitive { // interface
  virtual void intersection(const Ray& ray, Intersection& intersection);
  virtual RGB color() const;
  virtual bool planar() const;
};

class Cube { // fonctionnalite manquante
  virtual CIntersect intersect(const Ray& ray) {...}
  virtual RGB color() {...}
};
```

Adapter - implémentation - Heritage Multiple

Adapter Implementation

```
class AdapterCube: public Primitive, private Cube {
  void intersection(const Ray& ray, Intersection& inter) {
    Cintersect cinter =intersect(ray);
    inter=cinter // Suppose inter.operator=(cinter)
  }
  RGB color () const { return Cube::color(); }
  bool planar() const { return false ; }
};
```

Adapter - implémentation - Composition

Adapter Implementation

```
class AdapterCube: public Primitive {
  void intersection(const Ray& ray, Intersection& inter) {
    Cintersect cinter =_cube.intersect(ray);
    inter=cinter // Suppose inter.operator=(cinter)
  }
  RGB color () const { return _cube.color(); }
  bool planar() const { return false ; }
  private :
  Cube _cube;
};
```

Adapter - implémentation

Adapter Heritage multiple

```
class Adapter: public Interface, private Adaptee {
  void m1(...) {
    // redirect call from interace to implementation: Adaptee::m1(...)
  }
};
```

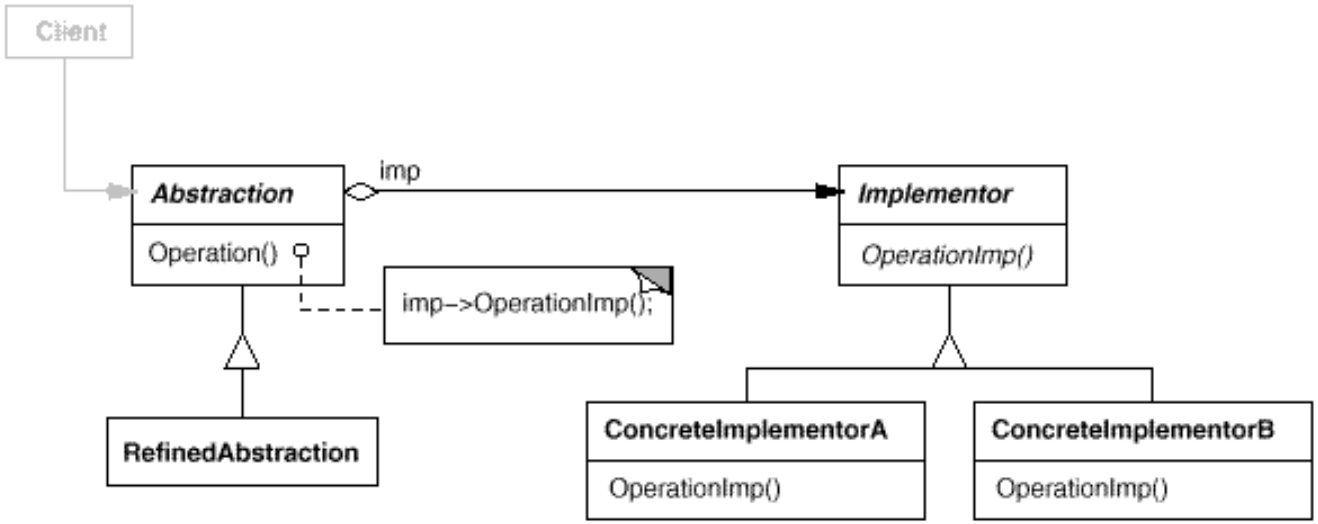
Adapter Composition

```
class Adapter: public Interface {
  void m1(...) {
    // redirect call from interace to implementation: _ada.m1()
  }
  private:
    Adaptee _ada;
};
```

Bridge

- Decoupler l'interface et son implémentation : permet de faire varier les deux indépendamment.
- Alternative héritage pour implémentation.
- Permet de changer le comportement à l'execution.

Bridge - Schema



Bridge - Implementation

Bride Implementation

```
class RenderingContext {
  virtual ~RenderingContext() {}
  virtual void RenderLine(const vector<Point>& line)=0;
  virtual void RenderPoint(const vector<Point>& Point)=0;
};
//-------------------------------------------------------//
class GLContext: public RenderingContext { // Render RealTime with OpenGL
  virtual void RenderLine(const vector<Point>& line) {} // call GL
  virtual void RenderPoint(const vector<Point>& Point) {}
};

class PNGContext: public RenderingContext { // Render in a PNG image
  virtual void RenderLine(const vector<Point>& line) {} // call PNG
  virtual void RenderPoint(const vector<Point>& Point) {}
};
```

Bridge - Implementation

Bride Implementation

```
//-------------------------------------//
class Primitive {
  public:
  Primitive(RenderingContext* rendering): _rendering(rendering){}
  void render(const RenderType& rt)=0;
  protected:
  RenderingContext* _rendering;
};
class Square:public Primitive {
  Cube(const Point& ul, int length, RenderingContext* rendering):
       Primitive(rendering), _ul(ul), _length(length) {
    // precompute point and store in _pt
  }
  void render(const RenderType& rt) {
    switch(rt) {
      case Line: _rendering.RenderLine(_pt); break;
      case Point: _rendering.RenderPoint(_pt); break;
    }
  }
  private:
  Point _ul;
  int length;
  vector<Point> _pt;
}
```

Bridge - Implementation

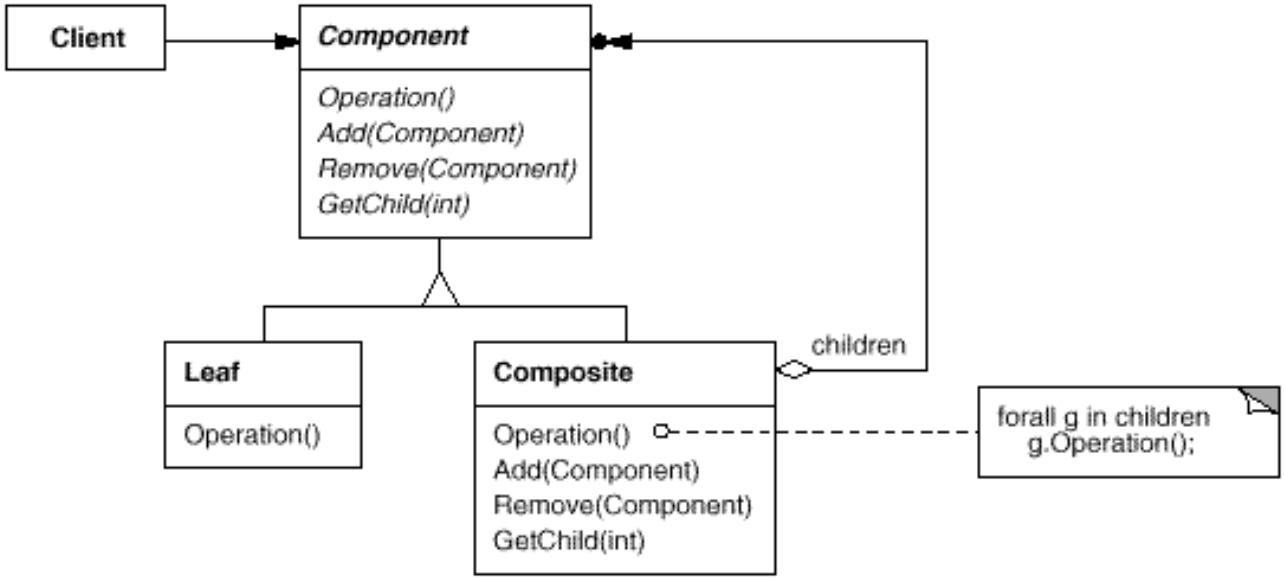
Bride Implementation

```
class Impl { virtual void Implementation(...)=0; ...};
class Impl1: virtual Impl { virtual void Implementation(...); ...};
class Impl2: virtual Impl { virtual void Implementation(...); ...};
...
class Abstr {
  Impl* impl;
  void function() { impl->Implementation(...); }
   ...
}
Class RefinedAbstr : public Abstr {
   ...
}
```

Composite

- Manipuler plusieurs objets de la même manière qu'on manipule un.
- Permet de créer une hiérarchie manipulable comme un seul objet.

Composite - Schema



Composite - Implementation

Composite - Implementation

```
struct Shape {
  virtual ~Shape() {};
  virtual void intersection(const Ray& ray) {}
};

struct KDTree: public Shape {
  virtual void intersection(const Ray& ray) {
    for (vector<Shape*>::iterator iter=store.begin(); iter!=store.end(); ++iter)
      iter->intersection(ray);
  }
  virtual void add(Shape* agregate) {store.add(agregate);}
  virtual void remove(Shape* agregate) { store.remove(agregate)}
 private :
  vector<Shape*> store; // store the primitive
};

struct Cube: public Shape {
  virtual void intersection(const Ray& ray) {} // compute intersection
};
```

Composite - Implementation

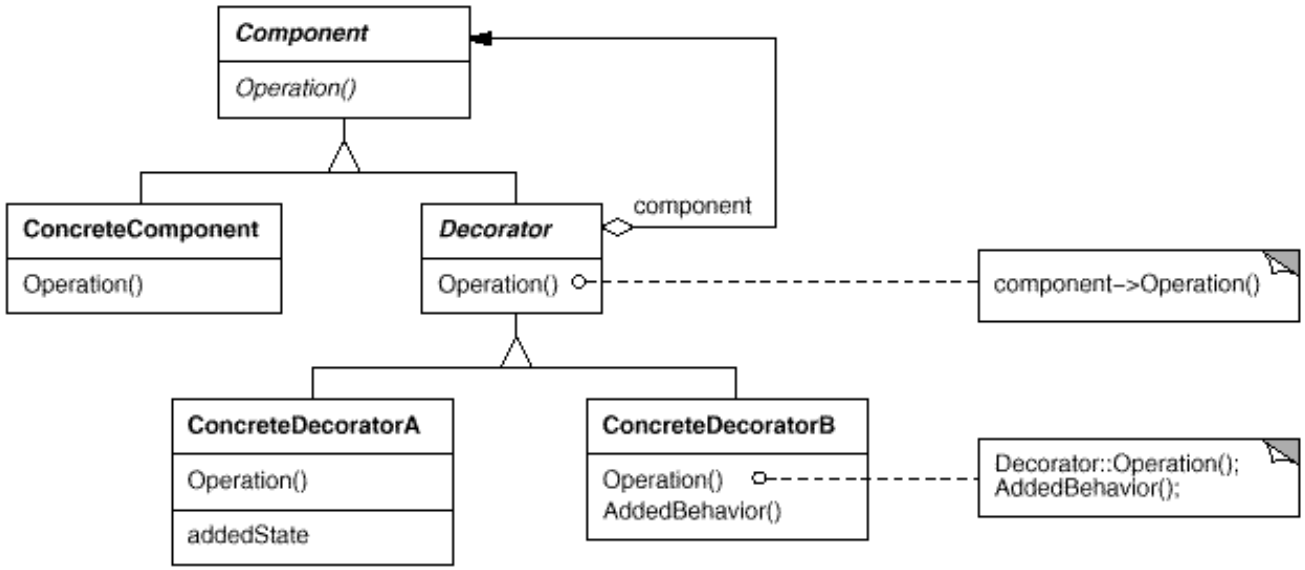
Composite - Implementation

```
class Component {
  virtual void operation1() {}
  ...
  virtual void add(Component *){}
  virtual void remove(Component *){}
  virtual Component* getChild() {return NULL;}
};

class Leaf: public Component {
  void operation1() {...}
  ...
};

class Aggregate: public Component {
  virtual void operation1() {}
  ...
  virtual void add(Component *){ add to Agregate}
  virtual void remove(Component *){ remove to Agregate}
  virtual Component* getChild() {return child;}
};
```

Decorator

- Ajouter/Retirer des fonctionnalités aux objets à l'exécution.
- Alternative héritage interface.

Decorator - Schema



Decorator - Implementation

Code : Decorator

```
class Shape; class Square; class Circle;

struct TranslatedShape: public shape {
  TranslatedShape(Shape* _shape, vec3 _translation): shape(_shape), tr(translation) {}

  void intersection(const Ray& ray, Intersection& intersection) {
    // apply translation to the ray
    shape->intersection(ray, Intersection);
    // translate the intersection
  }
 private :
    Shape* shape;
    vec3 tr;
};
class RotatedShape: public Shape {...}

Shape* Cube= RotatedShape(new TranslatedShape(new Cube(...)));
// a transformed cube
```

Decorator - Implementation

Code : Decorator

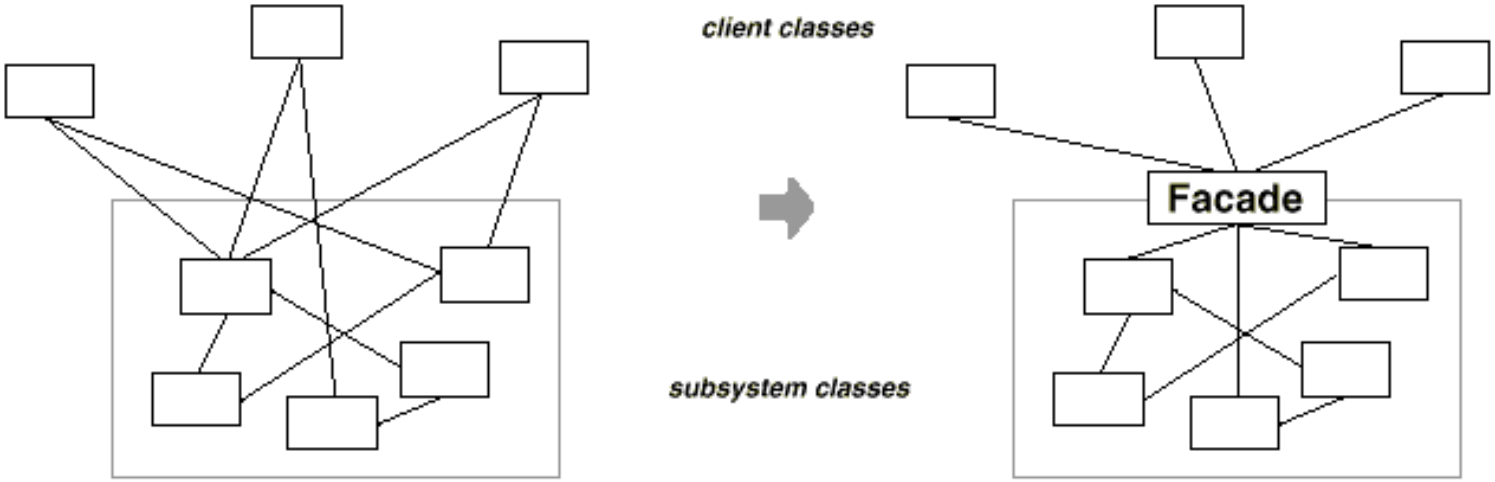
```
class Interface {virtual void m1(); ...};
class Implementation1: public Interface {...};
class Implementation2: public Interface {...};

struct Decorator1: public Interface {
   Decorator(Interface* if...):_if(if){}
   void m1() { // apply the decorator part and call _if->m1
  private :
    Interface _if;
};
 ....

Interface* Cube= Decorator1(new Decorator2(new Implementation()));
```

Facade

Permet de masquer la complexité d'une API : pour accomplir une action on doit utiliser de nombreuses classe avec des interactions complexes.

On utilise donc une classe intermédaire pour masquer la compléxité.

Facade - Schema



Facade - Implementation

Facade : Exemple

```
struct CPU {
  void freeze() { ... }
  void jump(long position) { ... }
  void execute() { ... }
};

struct Memory {
  void load(long position, char* data) {...}
};

struct HardDrive { };

struct Computer {
  void startComputer() {
    cpu.freeze();
    memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
    cpu.jump(BOOT_ADDRESS);
    cpu.execute();
  }
};

void main() {
  Computer facade = new Computer();
  facade.startComputer();
}
```

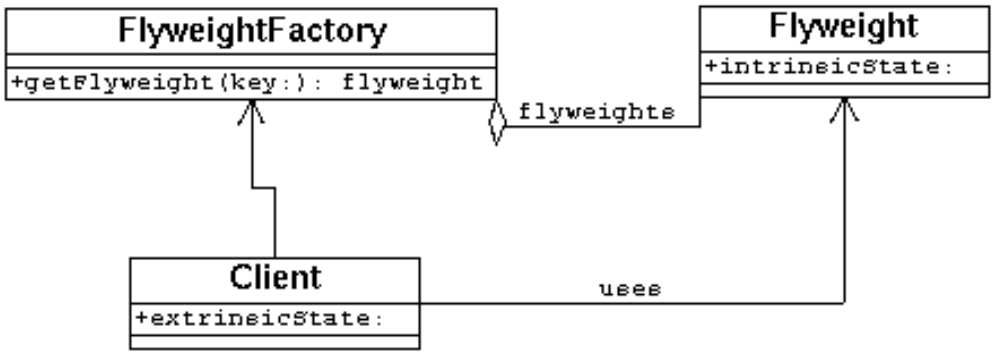
Facade - Implementation

Impossible à généralisé

Flyweight

Eviter la duplication d'objet : De nombreux objets instancier dont de nombreux doublons.

FlyWeight - Schema



Flyweight - Exemple

Flyweight : Exemple

```
class Circle {
  int _radius;
  int _center;
public:
  Circle(int center, int radius): _radius(radius), _center(center) {}
  static int hkey(int radius, int center) { return radius*1000+center; }
};

class FlyWeigthCircle {
  static map<int, Circle*> hmap;
public:
  static Circle* GetCircle(int radius, int center) {
    Circle* c=hmap[Circle::hkey(radius,center)];
    if (!c)
      hmap[Circle::hkey(radius,center)]=new Circle(radius,center);
    return hmap[Circle::hkey(radius,center)];
  }
  static int size() { return hmap.size(); }
};

for (int j=0; j < 100; ++j)
  for (int i=0; i< 1000; ++i)
    Circle* c=FlyWeigthCircle::GetCircle(i,i);

  cout << FlyWeigthCircle::size() << endl;
```

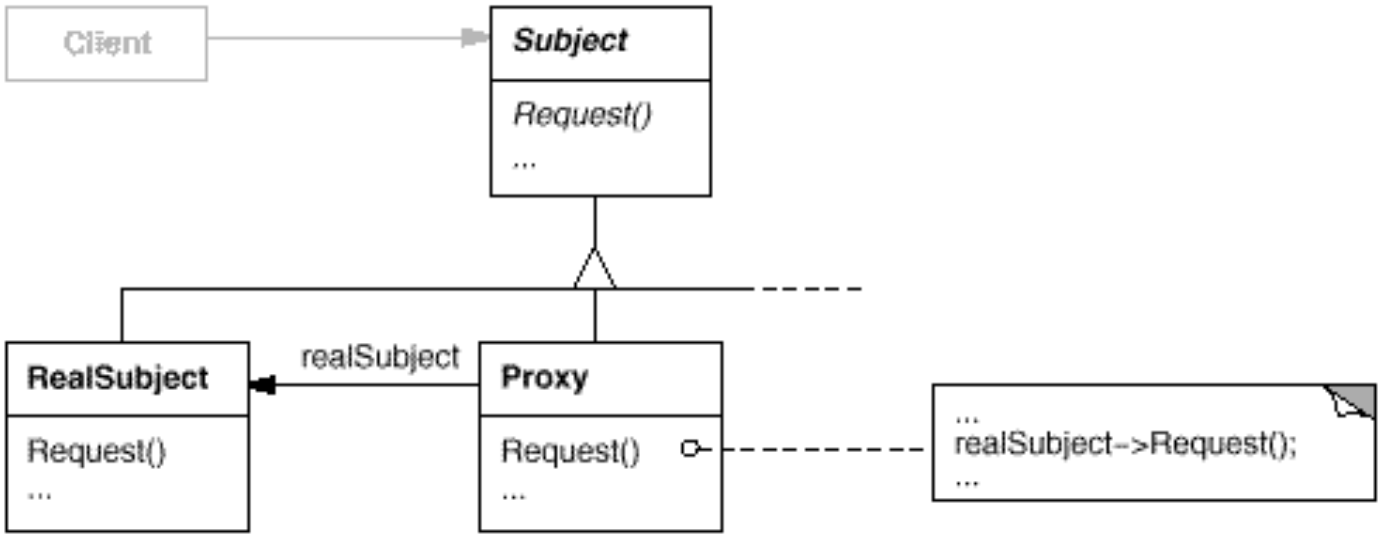
FlyWeight - Implementation

Assez dur, voir boost : :flyweight

Proxy

Un objet avec la même interface, servant à repousser une action

Proxy - Schema



Proxy - Implementation

Proxy : Implementation

```
struct Image {
  public:
   virtual void display()=0;
   virtual ~Image();
};

struct ImageFromDisk {
  ImageFromDisk(const string &path) { img=load(path);}
  void display() { /*display the image */}
 private:
  unsigned char* load(const string& path);
  unsigned char* img;
};

struct ProxyImg {
  ProxyImg(const string& path): _path(path), image(0){}
  void display() {
    if (!img)
      img=new ImageFromDisk(_path);
    img->display();
  }
 private:
  Image* image;
  string _path;
};
```

```
Image i1=new ProxyImg("img1"); //not loaded
Image i2=new ProxyImg("img2"); //not loaded
i1->display() //load
i2->display() // loaded
i1->display() // already loaded
```

Proxy - Implementation

Souvent similaire chose que decorator.

Plan

Behavior

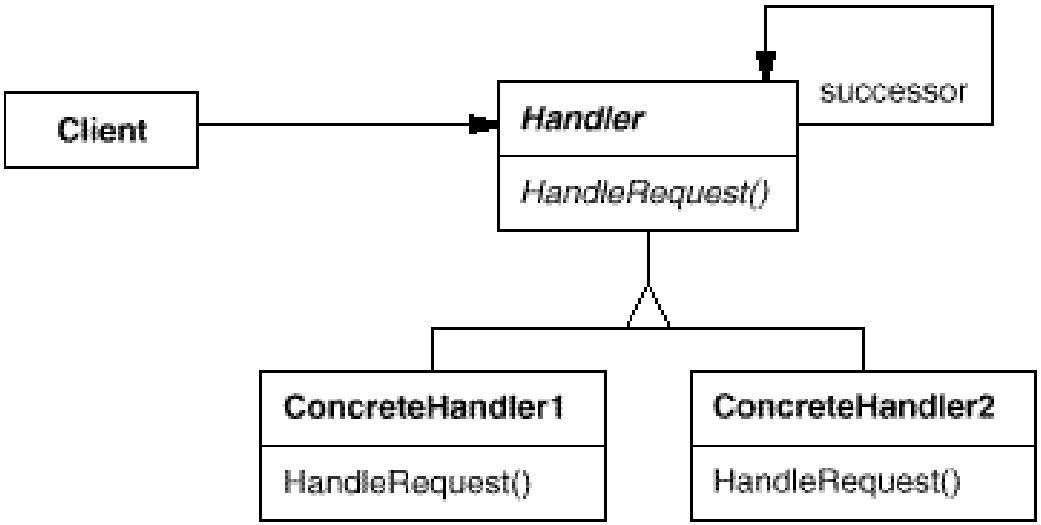
Communication entre les objets

Chain of Responsability

Chaine d'action :

- le 1er element de la chaine essaie d'effectuer une opération
- S'il y arrive : fin
- Sinon : le 2ème maillon essaye...

Chain of Responsability - Schema



Chain of Responsability - Implementation

Chain of Responsability : Decorator

```
struct Logger {
 Logger(Logger* log=0): next(log) {}
  virtual ~Logger() { delete next; next=0; }
 void setNext(Logger* _next) {next=_next;}
 void log(const string& protocol,
          const string& message) {
    if (!handle(protocol, message) && next)
       next->log(protocol, message);
 }
  virtual bool handle(const string& protocol,
                      const string& message) =0;
 Logger* next;
};

struct DebugLogger : public Logger {
  DebugLogger(Logger*log=0): Logger(log) {}
  bool handle(const string& protocol,
              const string& message) {
     if (protocol =="Debug") {
       cerr << "!DEBUG!" << message;
       return true;
     }
     return false;
  }
};
```

```
struct MailLogger : public Logger {
  MailLogger(Logger*log=0): Logger(log) {}

  bool handle(const string& protocol,
              const string& message) {
    if (protocol=="Mail") {
      cerr << "!MAIL!" << message;
       return true;
    }
    return false;
  }
};

struct StdLogger : public Logger {
  StdLogger(Logger*log=0): Logger(log) {}

  bool handle(const string& protocol,
  const string& message) {
    cerr << "!Standard!" << message;
    return true;
  }
};
```

Chain of Responsability - Implementation

Chain of Responsability : Decorator

```
Logger *logger=new MailLogger(
new DebugLogger(
new StandardLogger()));

logger->log("Mail","Test");
logger->log("Debug","Test");
logger->log("Standard","Test");
logger->log("Other","Test");
```

Chain of responsablity

```
template<class T>
struct ChainOfResponsability {
  ChainOfResponsability(T* n=0):next(n) {}
  template<class U, class V>
  void execute(const U& arg, const V& arg2) { if (!handle(arg,arg2) && next) next->execute(arg,arg2); }
  void setNext(T* _n){next=_n;}

   virtual ~ChainOfResponsability() {
     if (next){
       delete next; next=0;
     }
   }
protected:
  template<class U, class V>
  bool handle(const U& arg, const V& arg2) {return static_cast<T&>(*this).handle(arg,arg2);}
  T* next;
};

struct Logger: public ChainOfResponsability<Logger> {
  Logger(Logger* l=0): ChainOfResponsability<Logger>(l) {}
   virtual bool handle(const string& protocol,const string& message)=0;
};
```

Chain of responsablity

```
struct DebugLogger : public Logger {
  DebugLogger(Logger* l=0): Logger(l) {}
  bool handle(const string& protocol,const string& message) {
    if (protocol=="Debug") {
      cout << "DEBUG!" << " " << message << endl;
      return true;
    }
    return false;
  }
};


struct MailLogger : public Logger {
  MailLogger(Logger* l=0): Logger(l) {}
   virtual bool handle(const string& protocol,const string& message) {
    if (protocol=="Mail") {
      cout << "MAIL!" << " " << message << endl;
      return true;
    }
    return false;
  }
};
```

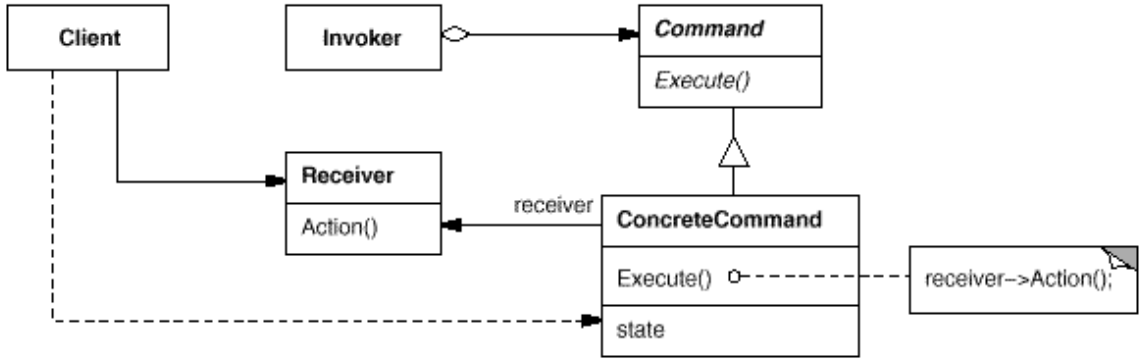
Chain of responsablity

```
struct StdLogger : public Logger {
  StdLogger(Logger* l=0): Logger(l) {}
   virtual bool handle(const string& protocol,const string& message) {
     cout << "NORMAL!" << " " << message << endl;
     return true;
   }
};


int
main() {

  MailLogger *logger=new MailLogger(new DebugLogger(new StdLogger()));

  logger->execute("Mail","test");
  logger->execute("Debug","test2");
  logger->execute("Standard","test3");
  logger->execute("Other", "test4");

   delete logger;
   return 0;
}
```

Command

- Definir une interface d'appel pour executer des commandes.
- Très utilisé pour gérer les undo/redos
- Permet de définir une séquence d'actions.

Command - Schema



Command

```
struct Command {
  virtual void execute()=0;
  virtual void undo()=0;
  virtual bool undoable()=0;
  virtual ~Command(){}
};

struct addCube: public Command {
  int id,x,y,z; Scene scene;
  virtual execute() {id=scene.addCube(x,y,z)}
  virtual undo() {scene.removePrimitive(id);}
  bool undoable() {return true;}
  void set{X,Y,Z,Scene}(...) {}
};

struct addLight: public Command {
  int id,x,y,z; Scene scene;
  virtual execute() {id=scene.addLight(x,y,z)}
  virtual undo() {scene.removePrimitive(id);}
  bool undoable() {return true;}
  void set{X,Y,Z,Scene}(...) {}
};

struct Render: public Command {
  virtual execute() {id=scene.render()}
  virtual undo() {}
  bool undoable() {return false;}
  void set{X,Y,Z,Scene}(...) {}
};
```

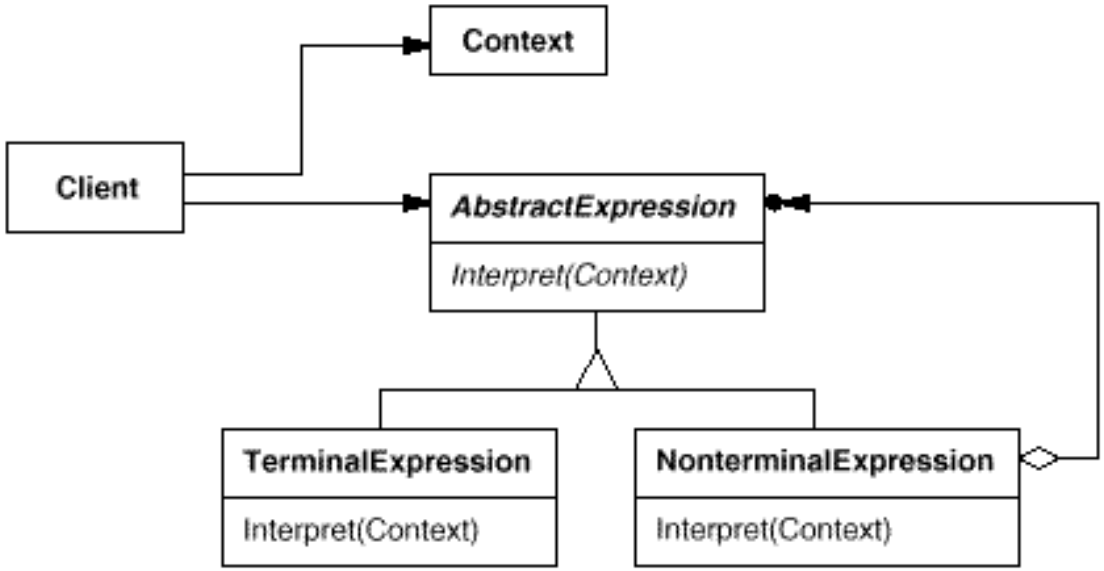
Command

```
class CommandGroupe: public Command {
  void addCommand(Command* cmd){_cmd.push_back(cmd);}
  void undo() {
    for (unsigned int i=0; i< _cmd.size(); ++i)
    _cmd[i]->undo()
  }
}
void execute() {
  for (unsigned int i=0; i< _cmd.size(); ++i)
  _cmd[i]->execute()
}
};

CommandGroupe actions;
action.push_back(new AddCube);
action.push_back(new AddLight);
action.push_back(new Render);
action.execute();
```

Interpreter

Evaluer des expressions à la volée

Interpreter - Schema



Interpreter exemple

Interpreter

```
struct Expression { virtual void interpret(stack<int>& s)=0; };

struct TerminalExpression_Number: public Expression {
  int number;
  TerminalExpression_Number(int n): number(n){ }
  void interpret(stack<int>& s) { s.push(number); }
};

struct TerminalExpression_Plus: public Expression {
  void interpret(stack<int>& s) {
    int top1=s.top(); s.pop();
    int top2=s.top(); s.pop();
    s.push(top1+top2);
  }
};

struct TerminalExpression_Minus :public Expression {
  void interpret(stack<int>& s) {
    int top1=s.top(); s.pop();
    int top2=s.top(); s.pop();
    s.push(top2-top1);
  }
};
```

Interpreter exemple

Interpreter

```
class Parser {
  list<Expression*> parseTree;
public:
  Parser(const string& s) {
    vector<string> tokens=split(s);
    for (unsigned int i=0; i < tokens.size(); ++i) {
      if      (tokens[i]=="+") parseTree.push_back(new TerminalExpression_Plus() );
      else if (tokens[i]=="-") parseTree.push_back(new TerminalExpression_Minus() );
      else parseTree.push_back( new TerminalExpression_Number(intFromString(tokens[i])));
    }
  }

  int evaluate() {
    stack<int> context;
    for (list<Expression*>::iterator iter=parseTree.begin(); iter != parseTree.end(); ++iter)
      (*iter)->interpret(context);
    return context.top();
  }
};

int main() {
  string expression = "42 4 2 - +";
  Parser p(expression);
  cout << "" << expression << "' equals " << p.evaluate()<<endl;
}
```

Iterator

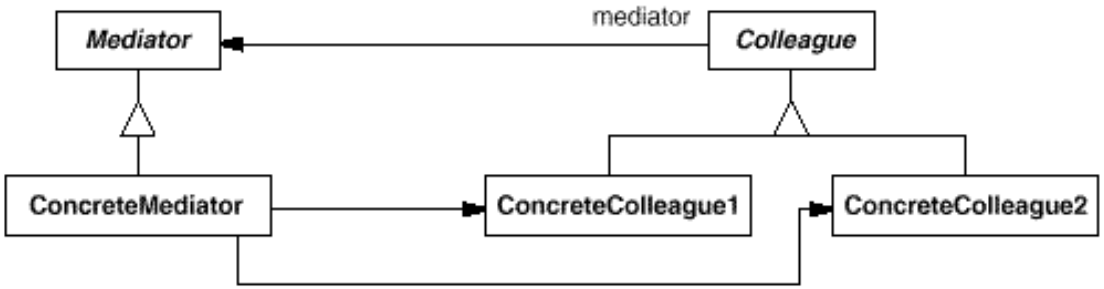
Deja Connu : parcourir une collection

Mediator

Une classe intermediaire entre plusieurs objets.

- Les objets discutent par l'intermediare de cette classe.
- Permet de controler les messages ;

Mediator - Schema



Mediator

Mediator - Implementation

```
struct Mediator {
  virtual void send(const string& message, Colleague* colleague)=0;
};

struct Colleague {
 Colleague(Mediator* m): mediator(m) { }
  virtual void send(string message)=0;
  virtual void notify(string message)=0;
protected:
 Mediator* mediator;
};

class ConcreteMediator : public Mediator {
 Colleague* colleague1;
 Colleague* colleague2;
public :
 void setColleague1(Colleague* c) { colleague1 = c; }
 void setColleague2(Colleague* c) { colleague2 = c; }
 void send(const string& message, Colleague* colleague) {
   if (colleague == colleague1)
     colleague2->notify(message);
   else
     colleague1->notify(message);
 }
};
```

Mediator

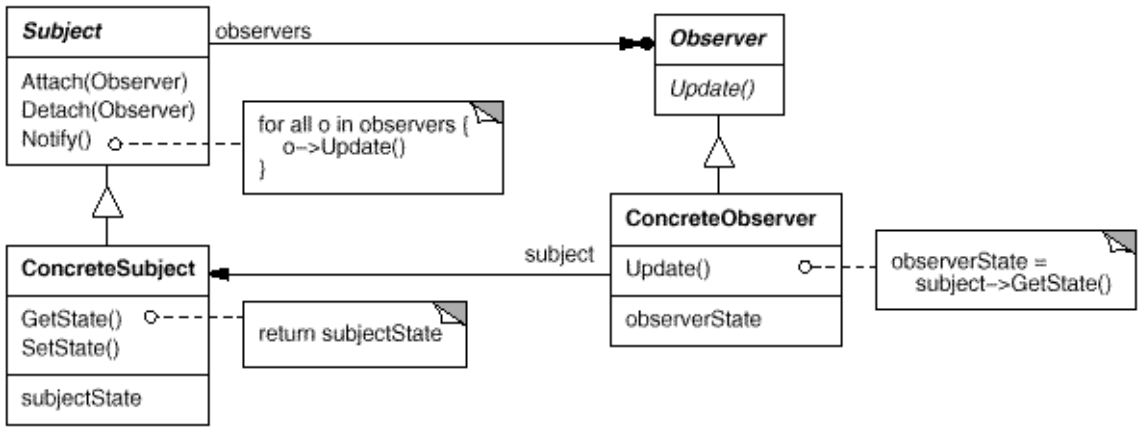
Mediator - Implementation

```
struct ConcreteColleague1 : public Colleague {
  ConcreteColleague1(Mediator* mediator): Colleague(mediator) { }
   virtual void send(string message) { mediator->send(message, this); }
   virtual void notify(string message) { cout << "Colleague1 gets message: " << message << endl; }
};

struct ConcreteColleague2 : public Colleague {
  ConcreteColleague2(Mediator* mediator): Colleague(mediator) { }
   virtual void send(string message) { mediator->send(message, this); }
   virtual void notify(string message) { cout << "Colleague2  gets message: " << message << endl; }
};

int main() {
  ConcreteMediator *m=new ConcreteMediator();
  ConcreteColleague1* c1 = new ConcreteColleague1(m);
  ConcreteColleague2* c2 = new ConcreteColleague2(m);

  m->setColleague1(c1);
  m->setColleague2(c2);

  c1->send("How are you?");
  c2->send("Fine, thanks");

   return 0;
}
```

Observer

Permet de notifier les objets de la modification d'un autre objet.

Observer - Schema



Observer - Implementation

Observer - Implementation

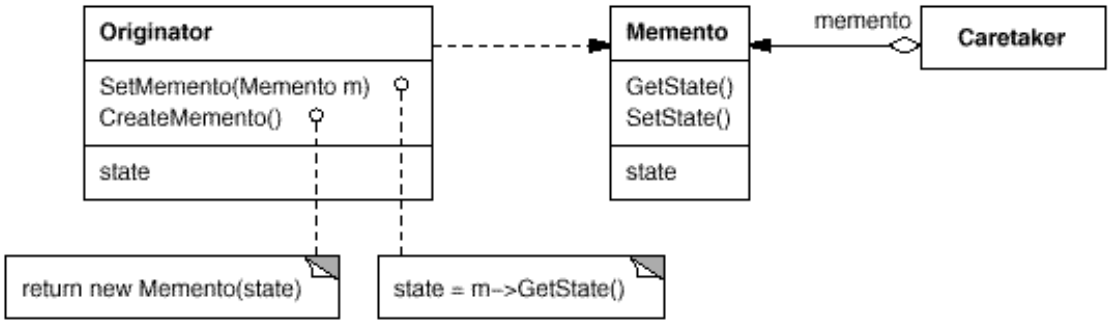
```
struct Observer {
  virtual void update(const int& u) =0;
  virtual ~Observer() {}
};

struct Observable {
  void addObserver(Observer* obs) {
    observers.push_back(obs);
  }
  void notify(const int& u) {
    list<Observer*>::iterator i;
    for (i=observers.begin(); i!= observers.end(); ++i)
      (*iter)->update(u);
  }

  void removeObserver(Observer* obs) { /*...*/ }
  protected:
   list<Observer*> observers;
};

struct TextRenderer: public Observer {
  void update(const int& data) {
    cerr << '\r' << data ;
  }
};
```

```
struct GraphicalRenderer: public Observer {
  void update(const int& data) {
    cerr << "update_progress_bar" << endl;
  }
};

struct ObservableRenderer: public Observable {
  void Render() {
    for (int i =0; i< 100; ++i) {
      //Render line
      notify(i);
      sleep(1);
    }
  }
};

GraphicalRenderer *gr=new GraphicalRenderer;
TextRenderer *tr=new TextRenderer;
ObservableRenderer obsr;
obsr.addObserver(grend);
obsr.addObserver(trenderer);
obsr.Render();
```

Memento

Restaurer l'état d'un objet.

Memento - Schema



Memento - Exemple

Memento - Exemple

```
class Memento {
  string state;
public:
  Memento(const string& stateToSave):state(stateToSave) { }
   string getSavedState() { return state; }
};

class Originator {
  string state;
public:
  void set(const string& s) {
    state=s;
    cout << "Originator: Setting state to " << state << endl;
  }

  Memento* saveToMemento() {
    cout << "Originator: Saving to Memento." << endl;
    return new Memento(state);
  }

  void restoreFromMemento(Memento* m) {
    state = m->getSavedState();
    cout << "Originator: State after restoring from Memento: " << state <<endl;
  }
};
```

Memento - Exemple

Memento - Exemple

```
class Caretaker {
  vector<Memento*> savedStates;
public:
  void addMemento(Memento* m) { savedStates.push_back(m); }
   Memento* getMemento(int index) { return savedStates[index]; }
};

int
main() {

  Caretaker* caretaker = new Caretaker();

  Originator* originator = new Originator();
  originator->set("State1");
  originator->set("State2");
  caretaker->addMemento(originator->saveToMemento());
  originator->set("State3");
  caretaker->addMemento(originator->saveToMemento());
  originator->set("State4");

  originator->restoreFromMemento(caretaker->getMemento(1));
  return 0;
}
```

Memento - Generalisation

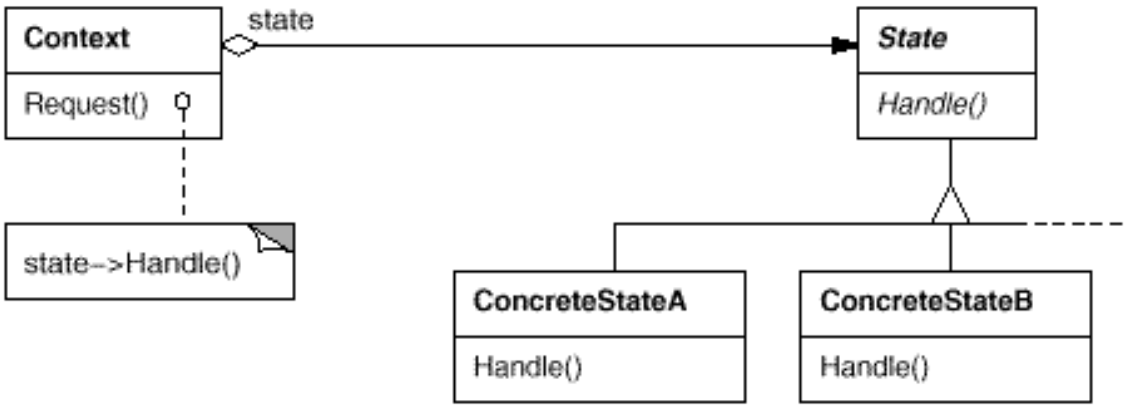
Memento - Generalisation

```
template<class T>
class Memento {
  T state;
public:
  Memento(const T& stateToSave):state(stateToSave) { }
  T getSavedState() { return state; }
};

class Originator {
  string state;
public:
  void set(const string& s) { state=s; }
  Memento<string>* saveToMemento() { return new Memento<string>(state); }
   void restoreFromMemento(Memento<string>* m) { state = m->getSavedState(); }
};

template<class T>
class Caretaker {
  vector<Memento<T>*> savedStates;
public:
  void addMemento(Memento<T>* m) { savedStates.push_back(m); }
  Memento<T>* getMemento(int index) { return savedStates[index]; }
};
```

State

Changer un objet et donc le comportement a l'utilisation.
Remplace les switch.

State - Schema



State - Exemple

State - Exemple

```
class State {
public:
  State(): current_position(0) {}
  void displayText() {
    for (BufferIterator iter=buffer.begin(); iter!= buffer.end(); ++iter)
      cout << *iter << " ";
  }
   virtual void keystrokes(const vector<string>& str)=0;
  void step_back() { current_position--;}
protected:
  vector<string> buffer;
  typedef vector<string>::iterator BufferIterator;
  typedef vector<string>::const_iterator ConstBufferIterator;
  int current_position;
};

struct Insert: public State {
   virtual void keystrokes(const vector<string>& str) {
    BufferIterator iter=buffer.begin();
    iter+=current_position;
    buffer.insert(iter, str.begin(), str.end());
    current_position+=str.size();
  }
};
```

State - Exemple

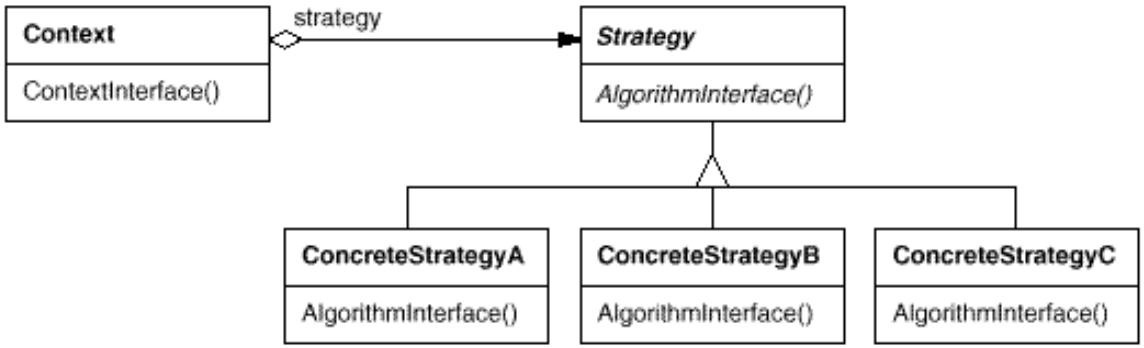
State - Exemple

```
struct Replace: public State {
  virtual void keystrokes(const vector<string>& str) {
    if (current_position+str.size() > buffer.size())
      buffer.resize(current_position+str.size());
    BufferIterator iter=buffer.begin();
    iter+=current_position;
    for (ConstBufferIterator i2=str.begin(); i2 != str.end(); ++i2, ++iter)
      *iter=*i2;
    current_position+=str.size();
  }
};

class TextEditor {
  State* s;
public:
  TextEditor(): s(new Insert) {}
  void setState(State* state) {s=state;}
  void key(const std::vector<string>& ss) {s->keystrokes(ss);}
};
```

Strategy

Changer un objet et donc le comportement a l'utilisation.
La difference avec le state pattern : Le comportment est le meme lorsque l'on change de stratégie (même résultat) mais l'algorithme est différent.

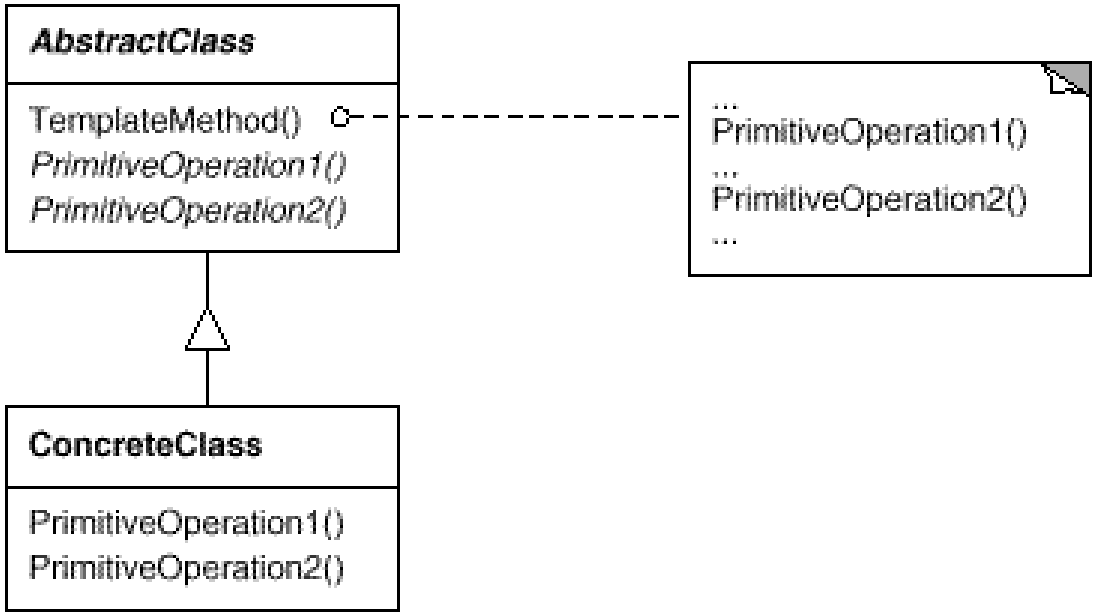
Strategy - Schema



Template

Garantir un cheminement.

Template - Schema



Template Exemple

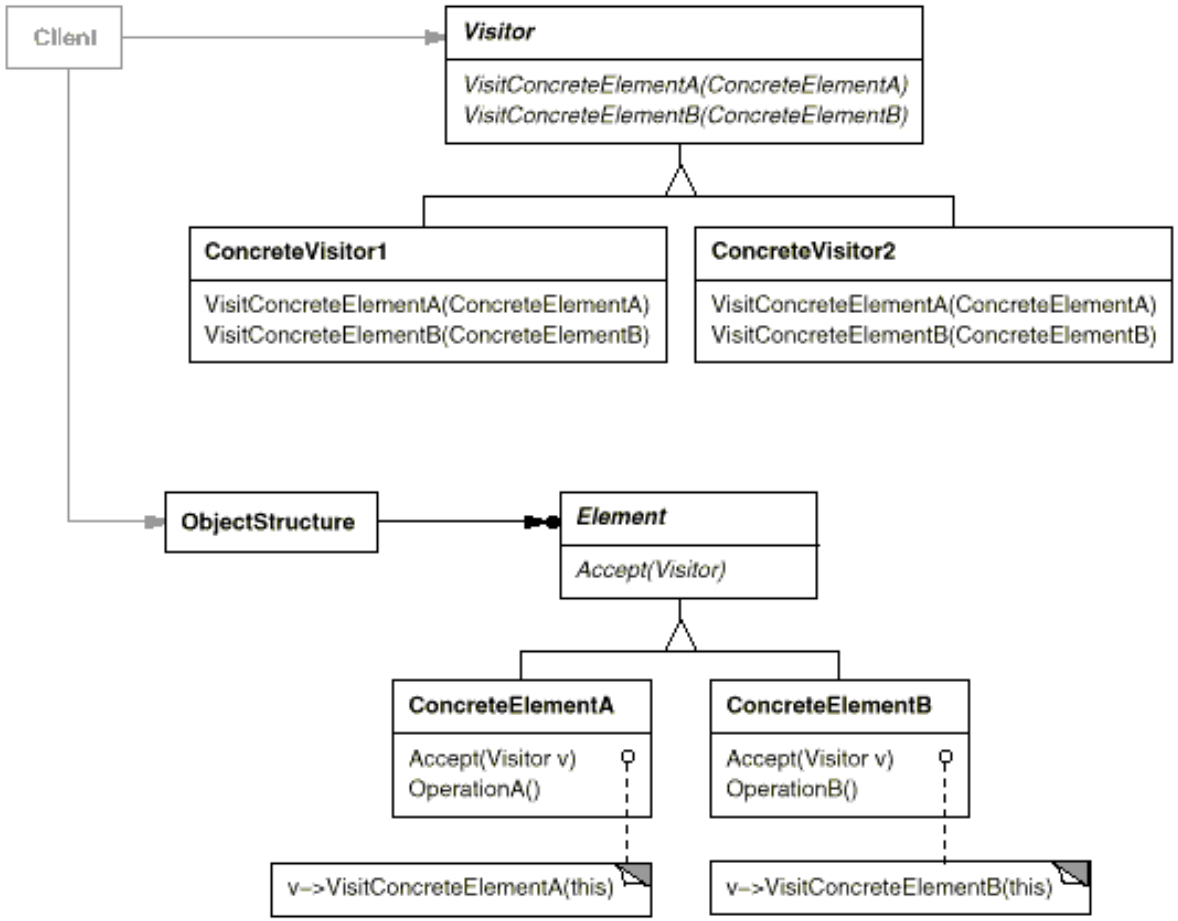
Template - Exemple

```
struct Mesh { void display() { std::cout << "display_Mesh" << std::endl; }};

struct Render {
  void draw() {
    preDisplay();
    mainDisplay();
    postDisplay();
  }
   virtual  void preDisplay() {std::cout << "Standart_preDisplay" << std::endl;}
  void mainDisplay() { mesh.display();}
   virtual  void postDisplay() {std::cout <<"Standart_preDisplay" << std::endl;}
  Mesh mesh;
};

struct InformationRender: public Render {void postDisplay() { cout << "Post_Display_overload" << endl;}};

struct SegmentationRender: public Render { void preDisplay() {cout << "Pre_Display_overload" << endl;}};

  Render *r1 = new Render();
  Render *r2 = new InformationRender(m);
  Render *r3 = new SegmentationRender(m);
  r1->draw();
  r2->draw();
  r3->draw();
```

Visitor

Definir de nouvelles opérations sans modifier une classe.

Visitor - Schema



Visitor Exemple

Visitor - Exemple

```
class Engine;
class Body;
class Car;

struct CarElementVisitor {
  virtual void visit(Engine& engine) const = 0;
  virtual void visit(Body& body) const = 0;
  virtual void visitCar(Car& car) const = 0;
  virtual ~CarElementVisitor() {}
};

struct CarElement {
  virtual void accept(const CarElementVisitor& visitor) = 0;
  virtual ~CarElement() {}
};

struct Engine : public CarElement {
  void accept(const CarElementVisitor& visitor) { visitor.visit(*this); }
};

struct Body : public CarElement {
  void accept(const CarElementVisitor& visitor) { visitor.visit(*this); }
};
```

Visitor Exemple

Visitor - Exemple

```
struct Car {
  vector<CarElement*>& getElements() { return elements_; }
  Car() {
    elements_.push_back( new Body() );
    elements_.push_back( new Engine() );
  }
  ~Car()  { delete elements[0]; delete elements[1]; }
private :
  vector<CarElement*> elements_;
};

struct CarElementPrintVisitor : public CarElementVisitor {
  void visit(Engine& engine) const { cout << "Visiting engine" << endl; }
  void visit(Body& body) const { cout << "Visiting body" << endl; }

  void visitCar(Car& car) const {
    cout << endl << "Visiting car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin(); it != elems.end(); ++it ) {
      (*it)->accept(*this);
    }
    cout << "Visited car" << endl;
  }
};
```

Visitor Exemple

Visitor - Exemple

```
int main() {
  Car car;
  CarElementPrintVisitor printVisitor;

  printVisitor.visitCar(car);

  return 0;
}
```

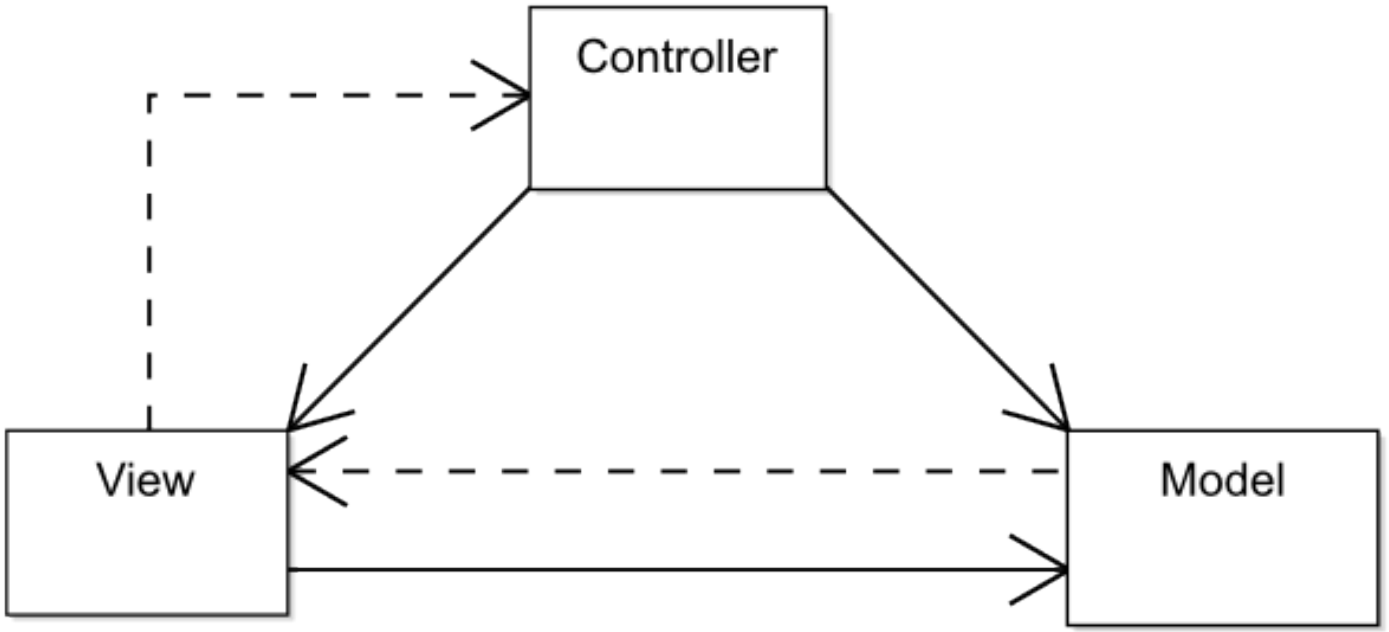
Plan

MVC

Model View Controller

- Principe Séparer les données de leurs visuation.
- Très utilisé pour les interfaces graphiques.

MVC - Schema



Pour Aller plus loin

Signal-Slot

- Communication entre Elements.
- Hautement Découplé.
- Seuls des méthodes sont liées.
- Implémentation : Qt Trolltech (préprocesseur moc)
- Implémentation : Template.

Pour Aller plus loin

- connection de signaux sur des slots (ou sur des signaux)
- implémentation non triviale
- utilisation très simple.

Qt

Preprocesseur spécial.

```
connect(Obj1, SIGNAL(Sig1(int)), Obj2, SLOT(Slot1(int)));  // Qt Way
```

Principe : Lorsque le signal Sig1 est émis il est transmis à tous les slots connectés, quelques soient le type de l'objet recepteur.
Conséquence : On connecte des objets, on émet le signal sans savoir qui le recevra et le traitement effectué par les récepteur.

Version template : Non présenté ici, utilisation différente mais simple.