LE LANGAGE C++
MASTER 1
LES PATTERNS

Jean-Baptiste.Yunes@univ-paris-diderot.fr
U.F.R. d'Informatique
Université Paris Diderot - Paris 7

2019–2020

PRESENTATION

PRÉSENTATION

- design pattern / motif conceptuel / patrons de conception / motif de conception
- ils désignent des cas récurrents et identifiés d'architecture et de conception de logiciels orientés objets
- ils décrivent aussi les solutions standards ou générales au problème
- les cas les plus généraux sont référencés dans un ouvrage considéré comme majeur : Design Patterns, de Gamma *et al.* (appelé aussi le Gang of Four - GoF / Bande des Quatre)
- le premier Design Pattern connu est le MVC Modèle-Vue-Contrôleur et qui permet d'obtenir un couplage faible entre le cœur fonctionnel d'une application et son interface utilisateur

PRÉSENTATION

- les motifs conceptuels ont été classés en 3 catégories principales :
  - création/construction
    - ces motifs se préoccupent de régler des problèmes liés à la création ou l'instanciation des objets
      - Abstract Factory,
      - Builder,
      - Factory Method,
      - Prototype,
      - Singleton

PRÉSENTATION

- structure
  - ces motifs s'occupent des questions relatives au découplage des concepts et à l'organisation structurelle des concepts manipulés par un logiciel.
    - Adapter,
    - Bridge,
    - Composite,
    - Decorator,
    - Facade,
    - Flyweight,
    - Proxy

PRÉSENTATION

- comportement
    - ces motifs ont pour sujet principal l'organisation de la collaboration des objets.
        - Callback,
        - Chain of Responsability,
        - Command,
        - Interpreter,
        - Iterator,
        - Mediator,
        - Memento,
        - Observer,
        - State,
        - Strategy,
        - Template Method,
        - Visitor

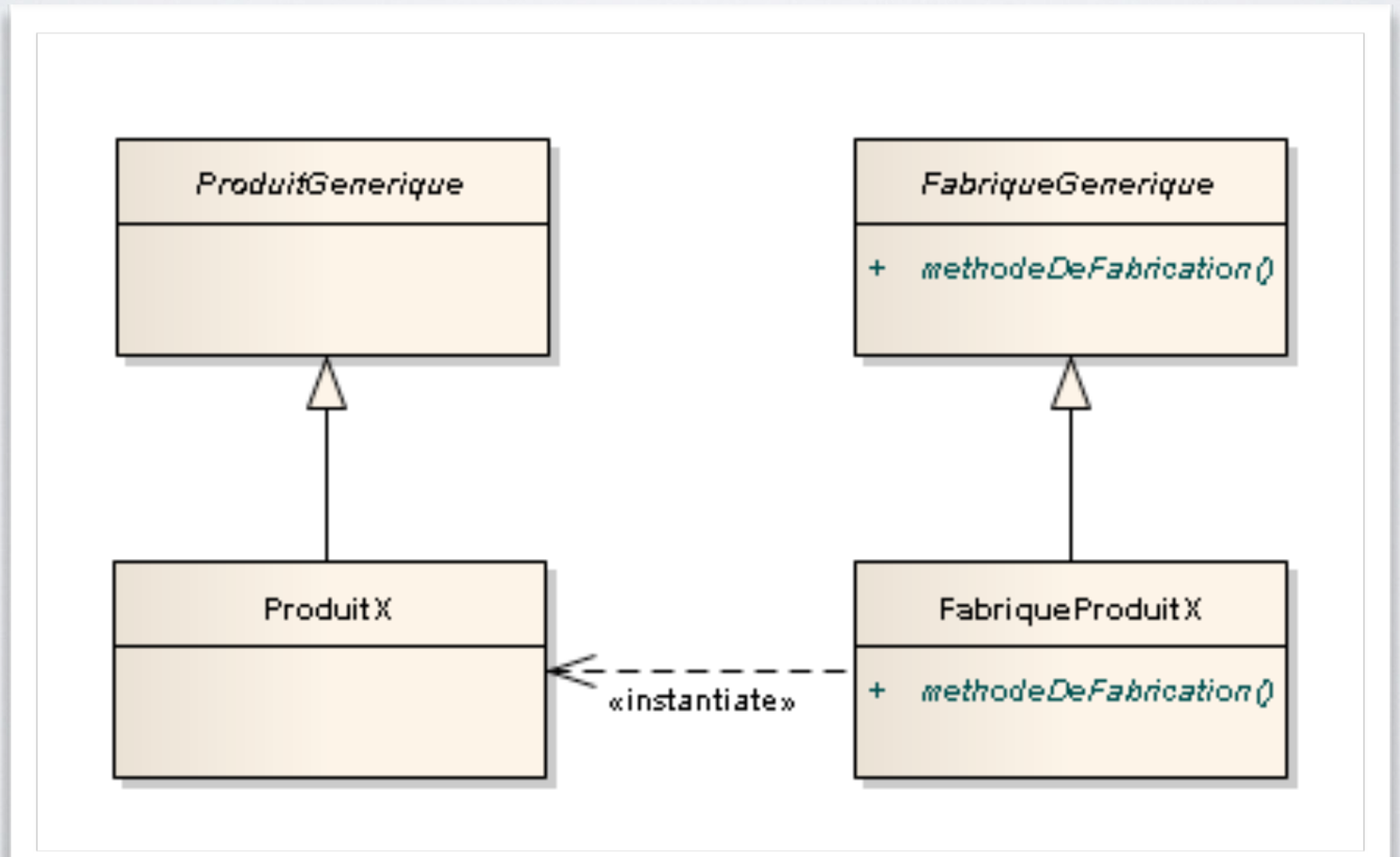
CREATION

FACTORY METHOD

- fabrique / factory method
- ce motif résout la question de la dépendance d'un code vis-à-vis de la classe d'instanciation d'un objet
- l'existence d'une classe abstraite, redéfinie en différentes classes concrètes est un trait qui doit faire se poser la question de l'usage de ce motif
    - en effet, la classe abstraite permet de manipuler de façon abstraite les objets, mais leur création/instanciation nécessite l'usage explicite de la classe (c'est nécessaire)
        - *i*.e. il reste encore les `new`

FACTORY METHOD

- le pattern « méthode de fabrication » est une version simplifiée de « la fabrique abstraite »



FACTORY METHOD

```
class Voiture {
public:
  virtual string getMarque() const = 0;
  friend ostream &operator<<(ostream &o,const Voiture &v) {
    return o << v.getMarque();
  }
};

class Renault : public Voiture {
public:
  string getMarque() const { return "Renault"; }
};

class Fiat : public Voiture {
public:
  string getMarque() const { return "Fiat"; }
};
```

FACTORY METHOD

```
//  cette fonction est indépendante du type concret
void f(const Voiture &v) {
  cout << "Voiture " << v << endl;
}

//  cette partie du code est dépendante du type concret!
int main() {
  const Voiture &v = Renault();
  f(v);
  return 0;
}
```

FACTORY METHOD

```
class Voiture {
public:
  virtual string getMarque() const = 0;
  friend ostream &operator<<(ostream &o, const Voiture &v) {
    return o << v.getMarque();
  }
  //  méthode statique pour créer un objet
  static Voiture *createVoiture(string o);
};

Voiture *Voiture::createVoiture(string origine) {
  if (origine=="fr") return new Renault();
  if (origine=="it") return new Fiat();
  return nullptr;
}
```

FACTORY METHOD

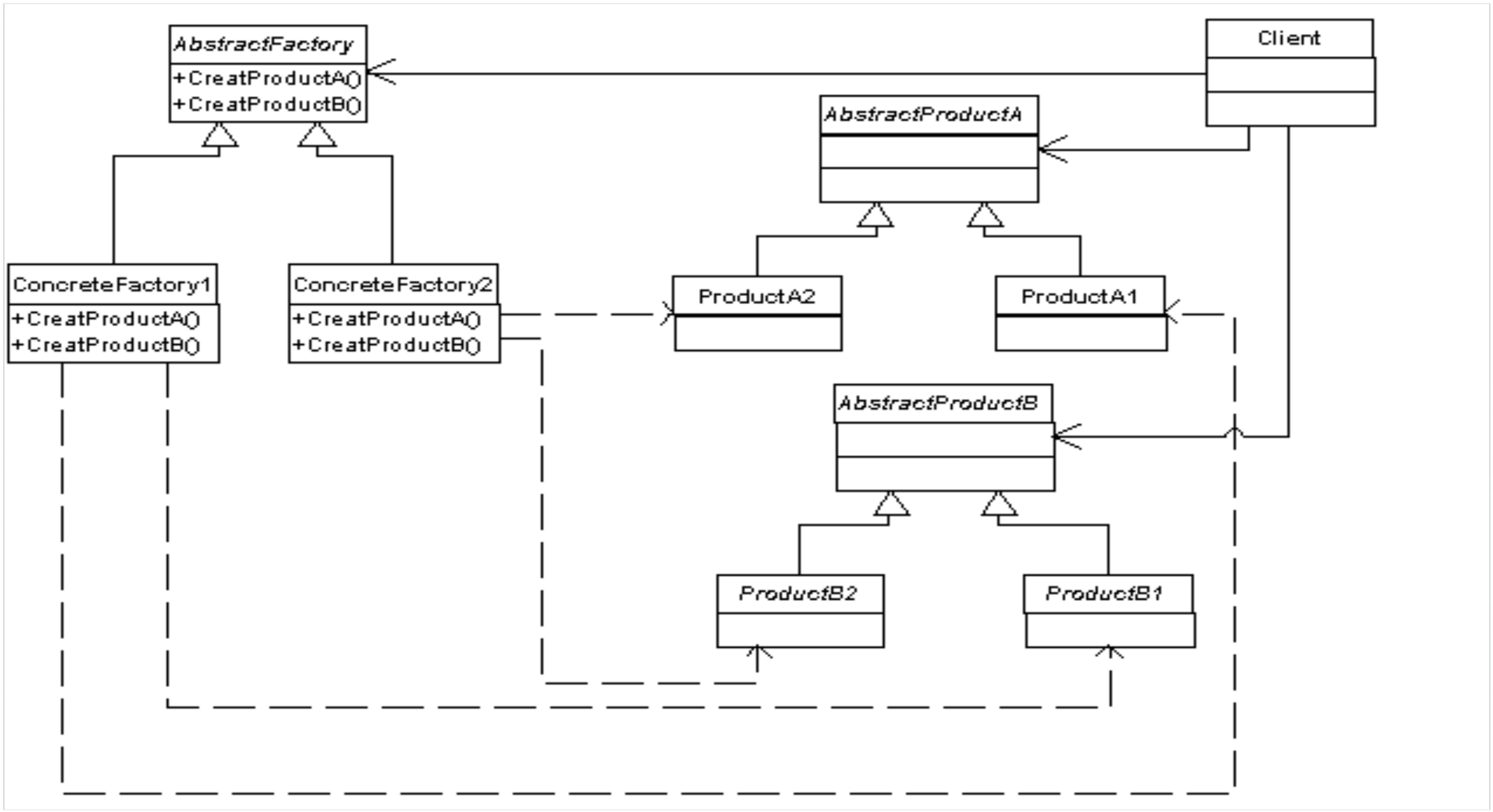
- la méthode `createVoiture` est une méthode de fabrication
  - elle reçoit en paramètre les critères qui permettront au concepteur des objets `Voiture` d'opérer le choix du type concret
  - dans le code client, les types concrets n'apparaissent plus. Le code client est devenu entièrement indépendant des types concrets

```
//  cette fois le type concret n'apparaît pas!
int main() {
  Voiture *v = Voiture::createVoiture("fr");
  f(*v);
  return 0;
}
```

ABSTRACT FACTORY

- la fabrique abstraite / l'usine abstraite / abstract factory permet d'obtenir la construction d'une famille uniforme d'objets
- permet de séparer les détails d'implémentation d'une famille d'objet de leur usage abstrait ou générique

ABSTRACT FACTORY



ABSTRACT FACTORY

```
class Voiture {
public:
  virtual string getModele() const = 0;
  friend ostream &operator<<(ostream &o,const Voiture &v){
    return o << v.getModele();
  }
};

class Kangoo : public Voiture {
public:
  string getModele() const { return "Kangoo"; }
};

class CinqueCento : public Voiture {
public:
  string getModele() const { return "500"; }
};
```

ABSTRACT FACTORY

```
class Camionnette {
public:
  virtual string getModele() const = 0;
  friend ostream &operator<<(ostream &o,const Camionnette &v) {
    return o << v.getModele();
  }
};

class Trafic : public Camionnette {
public:
  string getModele() const { return "Trafic"; }
};
class Ducato : public Camionnette {
public:
  string getModele() const { return "Ducato"; }
};
```

ABSTRACT FACTORY

```
class Fabrique {
public:
  virtual Voiture      *createVoiture()      = 0;
  virtual Camionnette *createCamionnette() = 0;
};

class FabriqueFrancaise : public Fabrique {
public:
  Voiture      *createVoiture()      { return new Kangoo(); }
  Camionnette *createCamionnette() { return new Trafic(); }
};

class FabriqueItalienne : public Fabrique {
public:
  Voiture      *createVoiture()      { return new CinqueCento(); }
  Camionnette *createCamionnette() { return new Ducato(); }
};
```

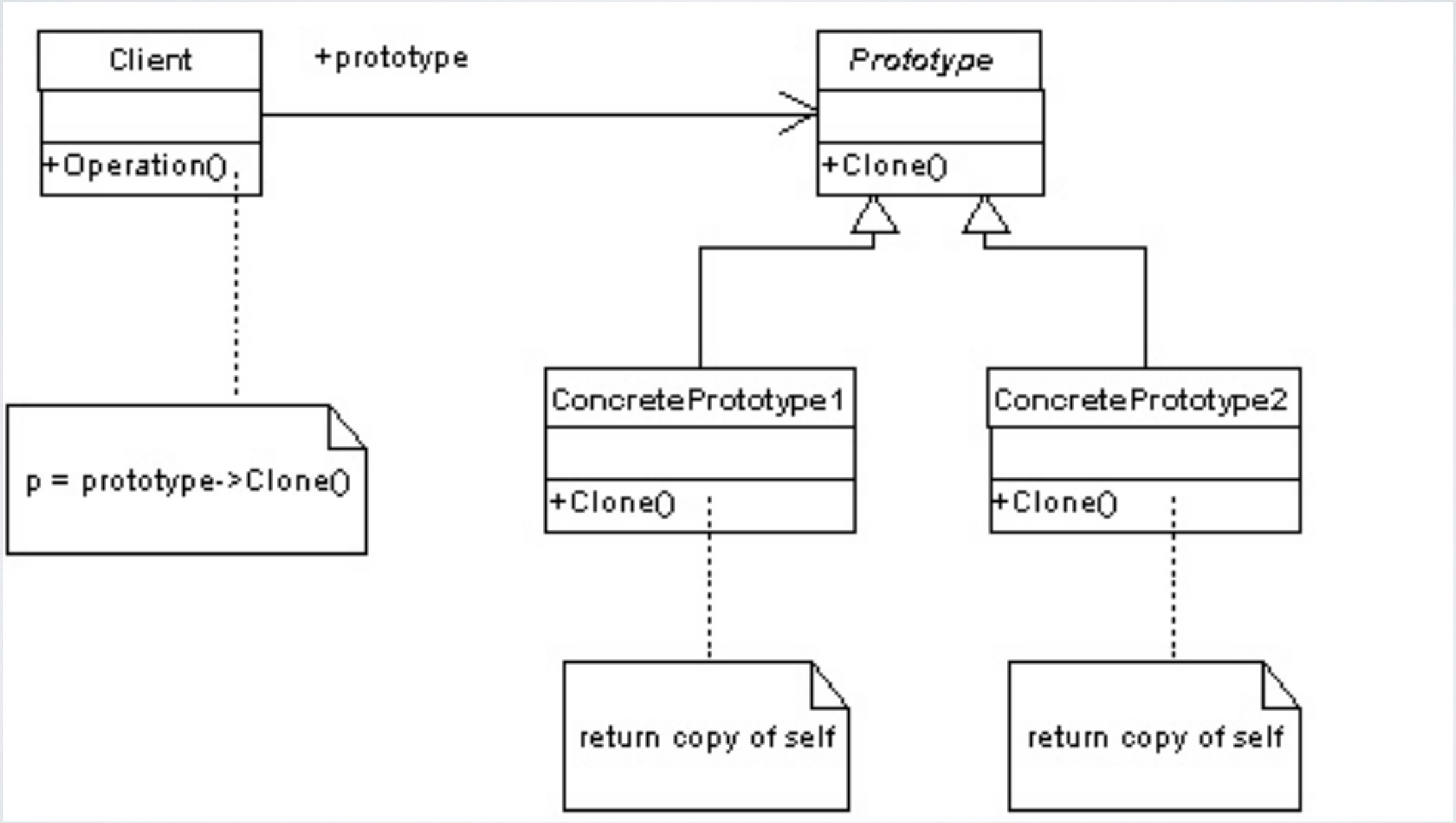
ABSTRACT FACTORY

```
void application(Fabrique &f) {
  Voiture     *v = f.createVoiture();
  Camionnette *c = f.createCamionnette();
  cout << *v << endl;
  cout << *c << endl;
}

int main() {
  FabriqueItalienne fi;
  application(fi);
  FabriqueFrancaise fr;
  application(fr);
  return 0;
}
```

PROTOTYPE

- ce pattern est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer (cloner) un objet existant et à modifier la copie ensuite :
  - la modification de la copie peut intervenir à deux endroits différents :
    - dans la méthode de clonage (donc au moment du clonage)
    - dans le code client (donc après le clonage)
- La méthode de construction est habituellement appelée `clone()`

PROTOTYPE


Client
+prototype
Prototype
+Operation()
+Clone()
p = prototype->Clone()
ConcretePrototype1
+Clone()
return copy of self
ConcretePrototype2
+Clone()
return copy of self

PROTOTYPE

```
class Animal {
public:
  virtual Animal *clone() =0;
  virtual void print()    =0;
};
class Mouton : public Animal {
public:
  virtual Animal *clone() {
    return new Mouton(*this);
}
  virtual void print() {
    cout << "Béééééé" << endl;
  }
};
```

PROTOTYPE

```
class Souris : public Animal {
public:
  virtual Animal *clone() {
    return new Souris(*this);
  }
  virtual void print() {
    cout << "Hiiiiii" << endl;
  }
};
```

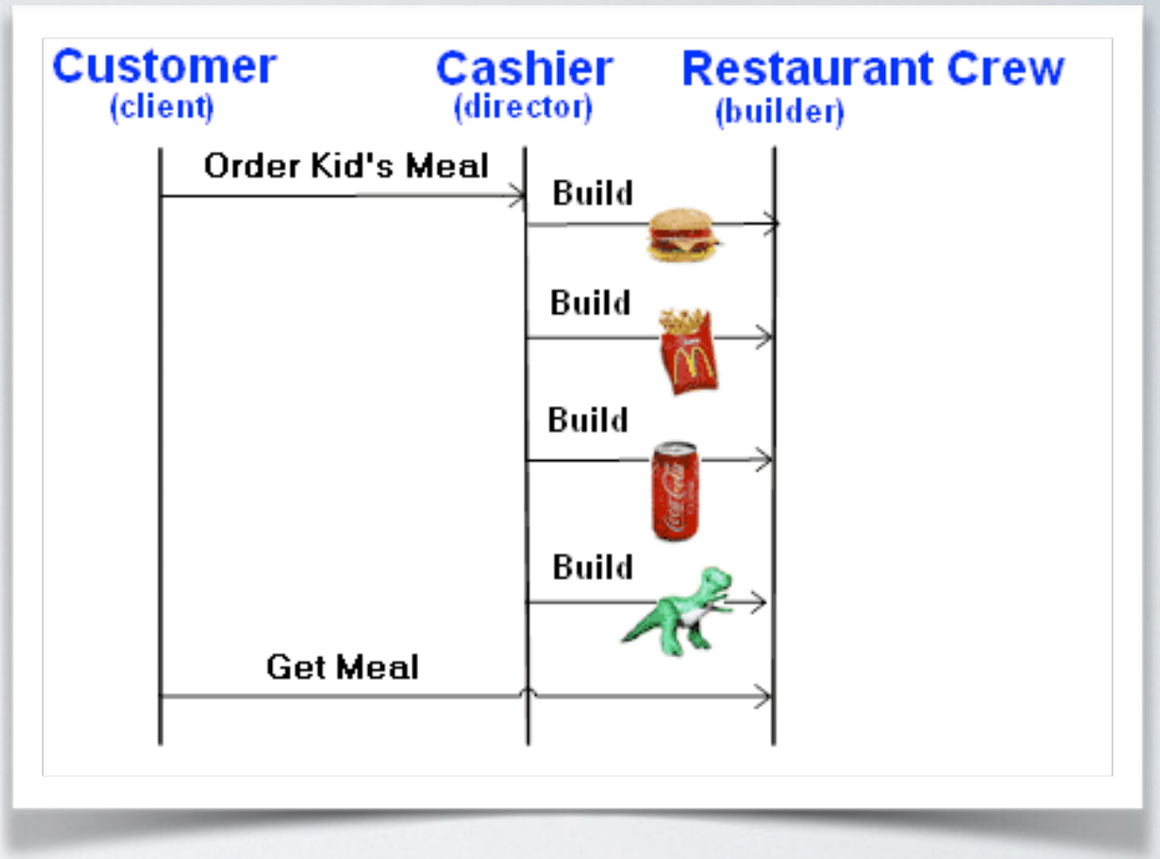
PROTOTYPE

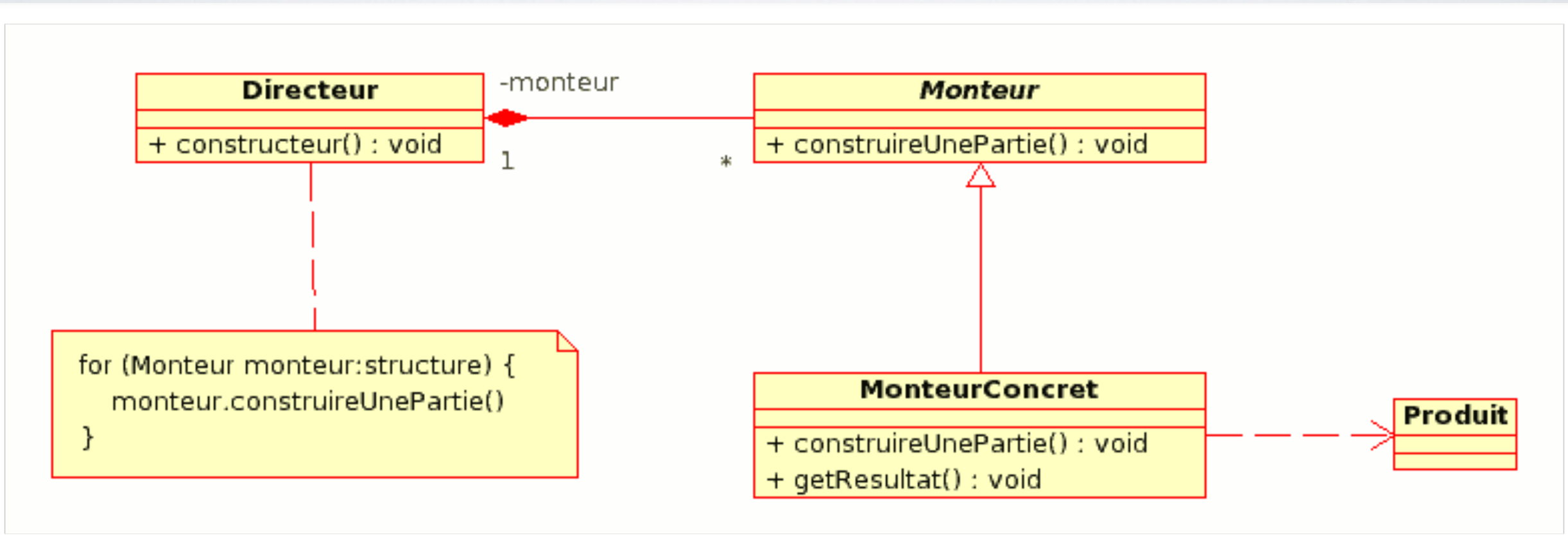
```
Animal *cloningMachine(Animal *a) {
   return a->clone();
}


int main() {
   Mouton m;
   cloningMachine(&m)->print();
}
```

BUILDER

- monteur est un pattern utilisé pour décrire la construction **incrémentale** d'un objet :
  - la construction elle-même est décrite abstraitement, *i*.e. le procédé de fabrication est prédéfini
  - le monteur (objet responsable de réaliser la construction) implémente les méthodes de construction utiles, *i*.e. la représentation finale des composants

BUILDER


Customer
(client)
Cashier
(director)
Restaurant Crew
(builder)
Order Kid's Meal
Build
Build
Build
Build
Get Meal


Directeur
+ constructeur() : void
-monteur
1
*
Monteur
+ construireUnePartie() : void
for (Monteur monteur:structure) {
monteur.construireUnePartie()
}
MonteurConcret
+ construireUnePartie() : void
+ getResultat() : void
Produit

BUILDER

```
class Voiture {
private:
  string portiere;
  string siege;
public:
  void setPortiere(string p) { portiere = p; }
  void setSiege(string s) { siege = s; }
  friend ostream &operator<<(ostream &o,Voiture *v) {
    o << "Portiere en " << v->portiere;
    return o << " et siege en " << v->siege;
  }
};
```

BUILDER

```
class MonteurVoiture {
protected:
   Voiture *v;

public:
   void createVoiture()  { v = new Voiture(); }
   Voiture *getVoiture() { return v; }
   virtual void addPortiere() = 0;
   virtual void addSiege()    = 0;
};
```

BUILDER

```
class MonteurVoitureLuxe : public MonteurVoiture {
public:
  void addPortiere() { v->setPortiere("acier blindé"); }
  void addSiege()    { v->setSiege("cuir"); }
};

class MonteurVoitureOrdinaire : public MonteurVoiture {
public:
  void addPortiere() { v->setPortiere("tôle"); }
  void addSiege()    { v->setSiege("tissu"); }
};
```

BUILDER

```
class Atelier {
private:
  MonteurVoiture *monteur;
public:
  Atelier(MonteurVoiture *m) { monteur = m; }
  Voiture *doTheJob() {
    monteur->createVoiture();
    monteur->addPortiere();
    monteur->addSiege();
    return monteur->getVoiture();
  }
};
```

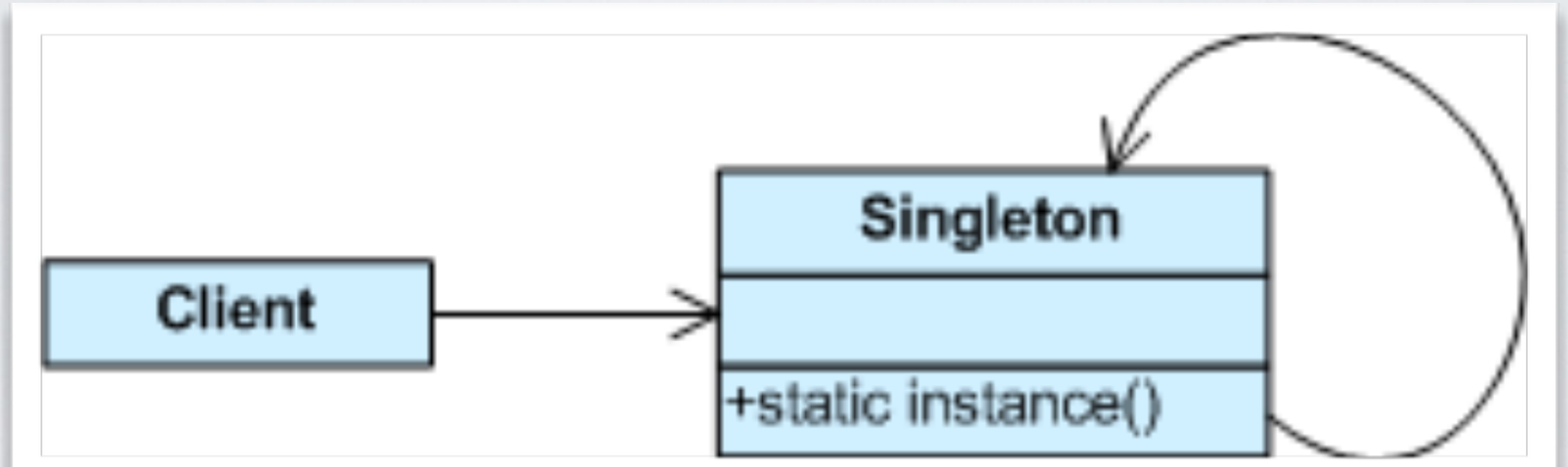
BUILDER

```
int main() {
  MonteurVoitureLuxe m;
  Atelier a(&m); // embauché!
  cout << a.doTheJob() << endl;
  return 0;
}
```

SINGLETON

- si l'on considère qu'une classe est un ensemble et qu'un objet de la classe est un élément de cet ensemble alors le pattern singleton permet d'obtenir un ensemble réduit à un seul élément :
  - tout instanciation renverra systématiquement le même unique objet

SINGLETON


Client
Singleton
+static instance()

GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource

SINGLETON

```
class Logger {
private:
  static Logger theUniqueLogger;
  Logger() {}                          //  rend impossible l'instanciation
  Logger(Logger &l) {}                 //  rend impossible la copie
  void operator=(Logger &l) {} //  rend impossible l'affectation
  ~Logger() {}                         //  rend impossible la destruction
public:
  void log(string message) { cerr << message << endl; }
  static Logger *getLogger() { return &theUniqueLogger; }
};
```

SINGLETON

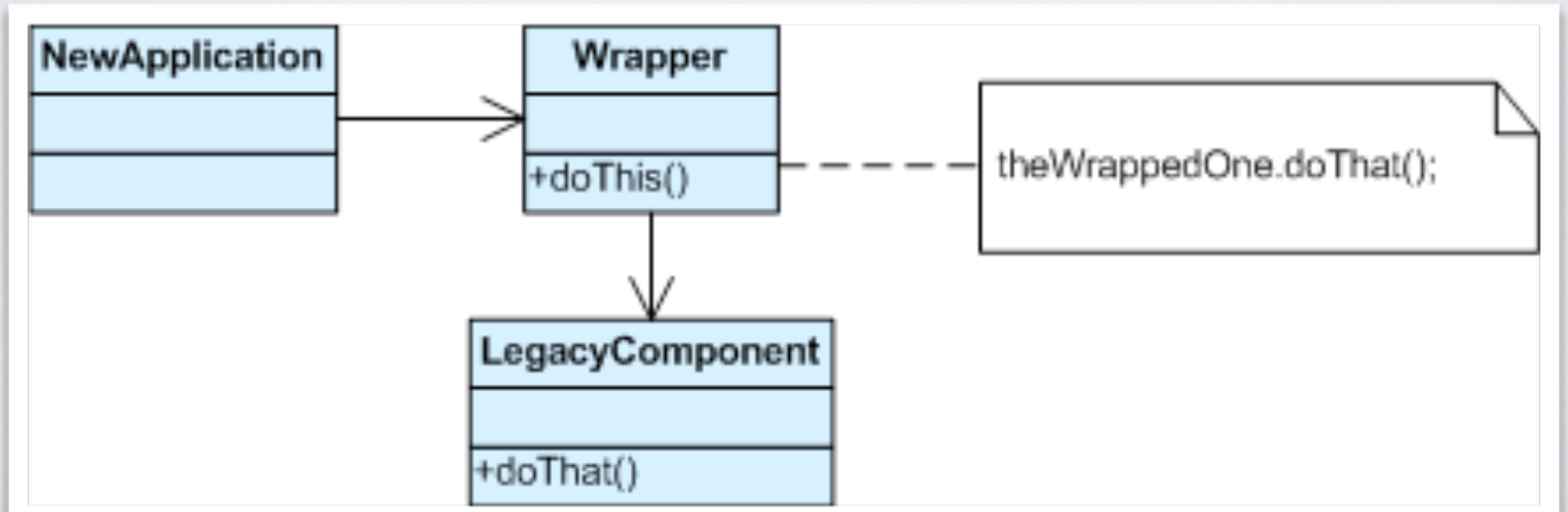
```
Logger Logger::theUniqueLogger; // instanciation
void f() {
  Logger *l = Logger::getLogger();
  l->log("je suis dans f");
}
void g() {
  Logger *l = Logger::getLogger();
  l->log("je suis dans g");
}
int main() {
  Logger *l = Logger::getLogger();
   l->log("démarrage");
   f(); g();
  return 0;
}
```

STRUCTURE

ADAPTER

- le pattern adaptateur/adapter permet de convertir l'interface d'un objet en une autre interface (transtypage)
- pour des raisons de compatibilité (reprendre une ancienne classe mais l'adapter à une nouvelle interface de manipulation)
- pour une question de réutilisabilité (récupérer un objet dans un autre cadre que pour lequel il a été défini)

ADAPTER


NewApplication
Wrapper
+doThis()
theWrappedOne.doThat();
LegacyComponent
+doThat()

ADAPTER

```
class Four {
public:
  virtual void ouvrirPorte() {}
  virtual void fermerPorte() {}
  virtual void thermostat(int v) {}
};
```

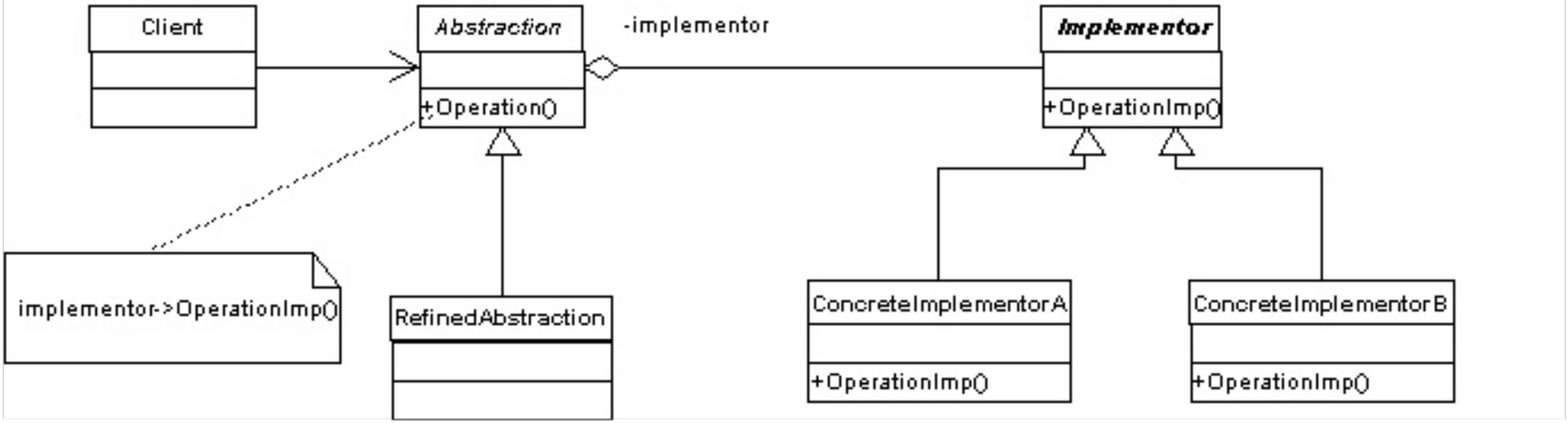
```
class Chauffage{
public:
  virtual void allumer()=0;
  virtual void chaud()=0;
  virtual void tresChaud()=0;
  virtual void eteindre()=0;
};
```

```
class FourRadiateurAdaptateur : public Chauffage,
                                private Four {
  void allumer()   { ouvrirPorte(); }
  void chaud()     { thermostat(5); }
  void tresChaud() { thermostat(10); }
  void eteindre()  { thermostat(0); fermerPorte(); }
};
```

BRIDGE

- le pattern pont est un adaptateur dynamique
  - l'abstraction et l'implémentation sont découplées

BRIDGE


Client
Abstraction
-implementor
Implementor
+Operation()
+OperationImp()
implementor->OperationImp()
RefinedAbstraction
ConcreteImplementorA
+OperationImp()
ConcreteImplementorB
+OperationImp()

BRIDGE

```
class Drawer {
public:
  virtual void drawRectangle(int x, int y, int l, int h) = 0;
};
```

BRIDGE

```
class Drawer1 : public Drawer {
public:
  void drawRectangle(int x,int y,int l,int h) {
    line(x,y,x+l,y); line(x,y+h,x+l,y+h);
    line(x,y,x,y+h); line(x+l,y,x+l,y+h);
  }
};

class Drawer2 : public Drawer {
  void drawRectangle(int x,int y,int l,int h) {
    line(x,y,x+l,y); line(x+l,y,x+l,y+h);
    line(x+l,y+h,x,y+h); line(x,y+h,x,y);
  }
};
```

BRIDGE

```
class Forme {
protected:
  Drawer *d;
  Forme(Drawer *d) { this->d = d; }
public:
  virtual void draw() = 0;
};
```

BRIDGE

```
class Rectangle : public Forme {
private: int x, y, l, h;
public:
  Rectangle(int x,int y,int l,int h,Drawer *d) :
            x(x), y(y), l(l), h(h), Forme(d) {}
  void draw() { d->drawRectangle(x,y,l,h); }
};

class RectangleBis : public Forme {
private: int x, y, l, h;
public:
  RectangleBis(int x,int y,int l,int h) :
       x(x), y(y), l(l), h(h), Forme(new Drawer2()) {}
  void draw() { d->drawRectangle(x,y,l,h); }
};
```

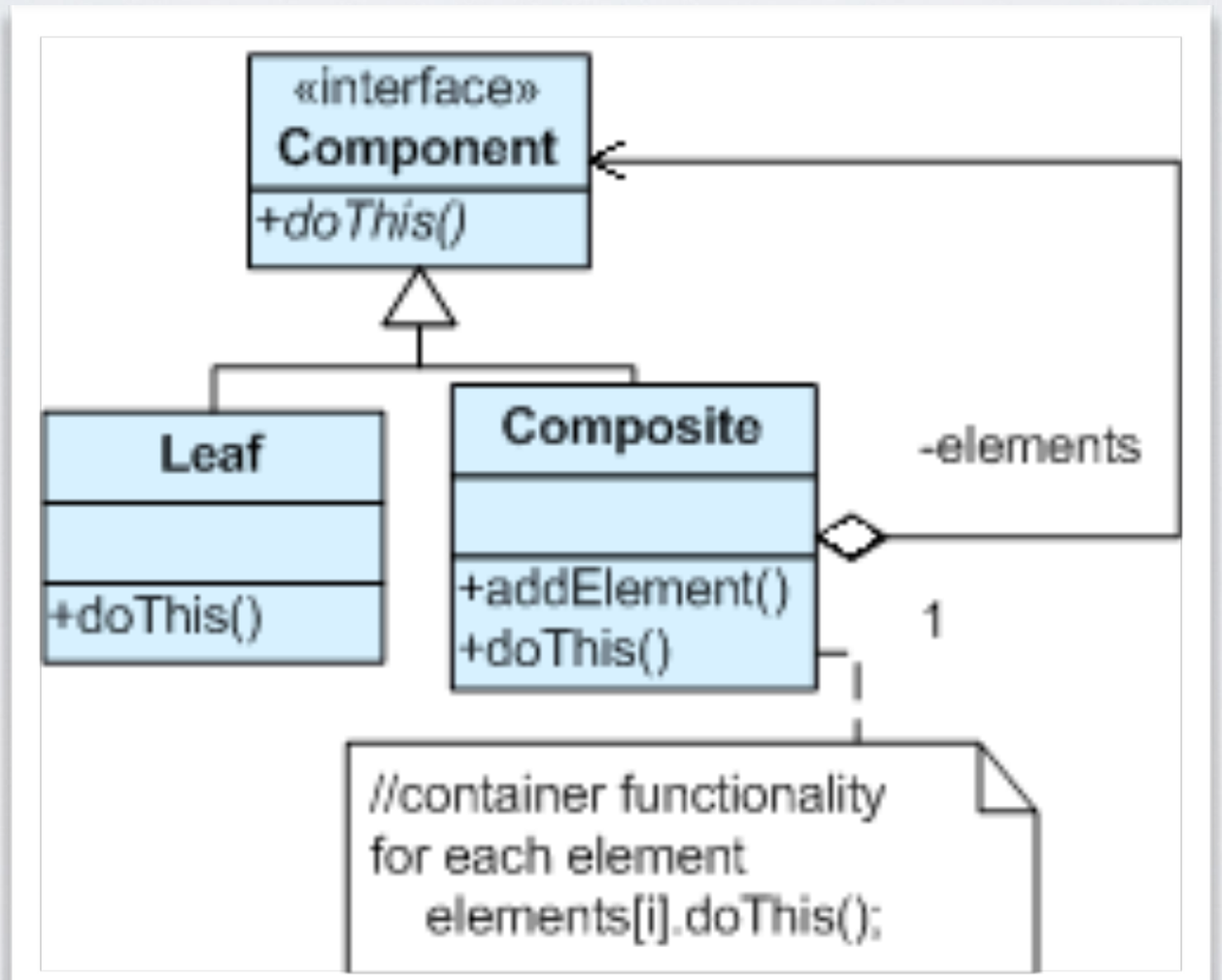
BRIDGE

```
int main() {
   Forme *s[2];
   s[0] = new Rectangle(0,0,1,1,new Drawer1());
   s[1] = new RectangleBis(1,1,2,2);
   for (int i=0; i<2; i++) s[i]->draw();
   delete s[0];
   delete s[1];
}
```

COMPOSITE

- est un pattern permettant de représenter des arborescences et d'y réaliser des traitements uniformes
  - les composants des interfaces graphiques sont typiques de ce pattern

COMPOSITE


«interface»
Component
+doThis()
Leaf
+doThis()
Composite
+addElement()
+doThis()
-elements
1
//container functionality
for each element
elements[i].doThis();

COMPOSITE

```
class Composant {
public:
  virtual void doit() = 0;
};

class Bouton : public Composant {
  string name;
 public:
  Bouton(string n) : name(n) {}
  void doit() { cout << '[' << name << ']'; }
};
```

COMPOSITE

```
class Container : public Composant {
  vector<Composant *> entries;
public:
  void add(Composant *c) { entries.push_back(c); }
  void doit() {
    cout << "(";
    for_each(entries.begin(), entries.end(),
             mem_fun(&Composant::doit));
    cout << ")";
  }
};
```

COMPOSITE

```
int main() {
  Container c1, c2, c3;
  Bouton b1("b1"), b2("b2"), b3("b3");
  c1.add(&b1);
  c1.add(&c2);
  c2.add(&b2);
  c2.add(&b3);
  c2.add(&c3);
  c1.doit();
  cout << endl;
  return 0;
}
```

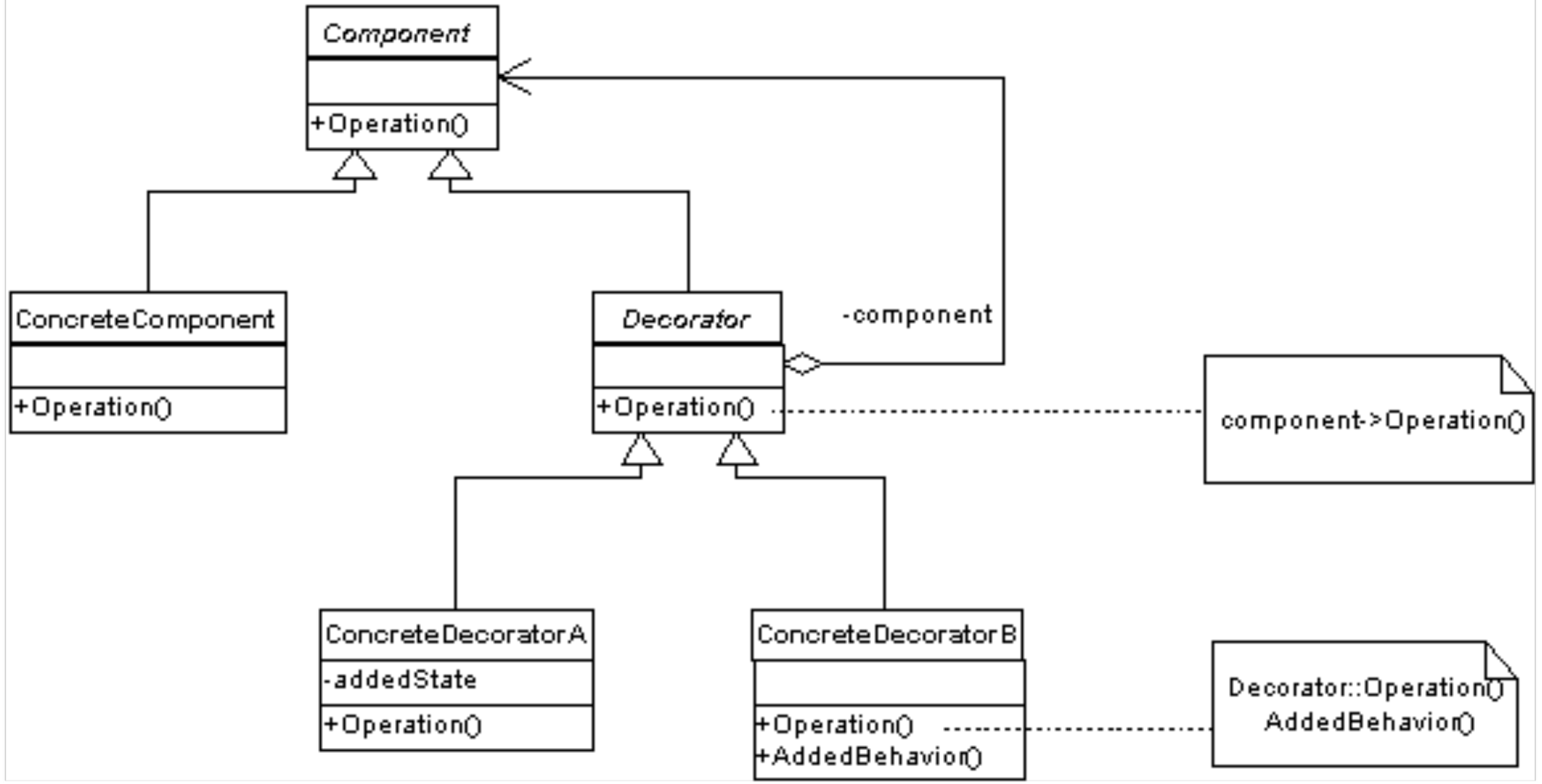
COMPOSITE

```
[Ciboulette:] ./composite
([b1]([b2][b3]()))
[Ciboulette:]
```

DECORATOR

- le pattern décorateur permet d'embellir fonctionnellement des objets existants. Cette alternative à la dérivation (qui n'est pas toujours possible) est plus flexible
  - on rencontre fréquemment ce pattern dans les hiérarchies d'entrées/sorties
  - ou les interfaces graphiques

DECORATOR


Component
+Operation()
ConcreteComponent
+Operation()
Decorator
-component
+Operation()
component->Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
Decorator::Operation()
AddedBehavior()

DECORATOR

```
class Boisson {
public:
  virtual void print() = 0;
};

class Eau : public Boisson {
public:
  void print() { cout << "eau"; };
};

class Lait : public Boisson {
public:
  void print() { cout << "lait"; };
};
```

DECORATOR

```
class Additif : public Boisson {
  Boisson *b;
public:
  Additif(Boisson *b) : b(b) {}
  virtual void print() {
    if (b) { b->print(); cout << '+'; }
  }
};
```

DECORATOR

```
class Sucre : public Additif {
public:
  Sucre(Boisson *b=0) : Additif(b) {}
  void print() { Additif::print(); cout << "sucre"; }
};

class Glacon : public Additif {
public:
  Glacon(Boisson *b=0) : Additif(b) {}
  void print() { Additif::print(); cout << "glacon"; }
};

class CafeSoluble : public Additif {
public:
  CafeSoluble(Boisson *b=0) : Additif(b) {}
  void print() { Additif::print(); cout << "cafesoluble"; }
};
```

DECORATOR

```
int main() {
  Boisson *b = new CafeSoluble(new Sucre(new Eau));
  b->print();
  cout << endl;
  Boisson *b2 = new CafeSoluble(new Sucre(
                                  new Glacon(new Lait)));
  b2->print();
  cout << endl;
  return 0;
}
```

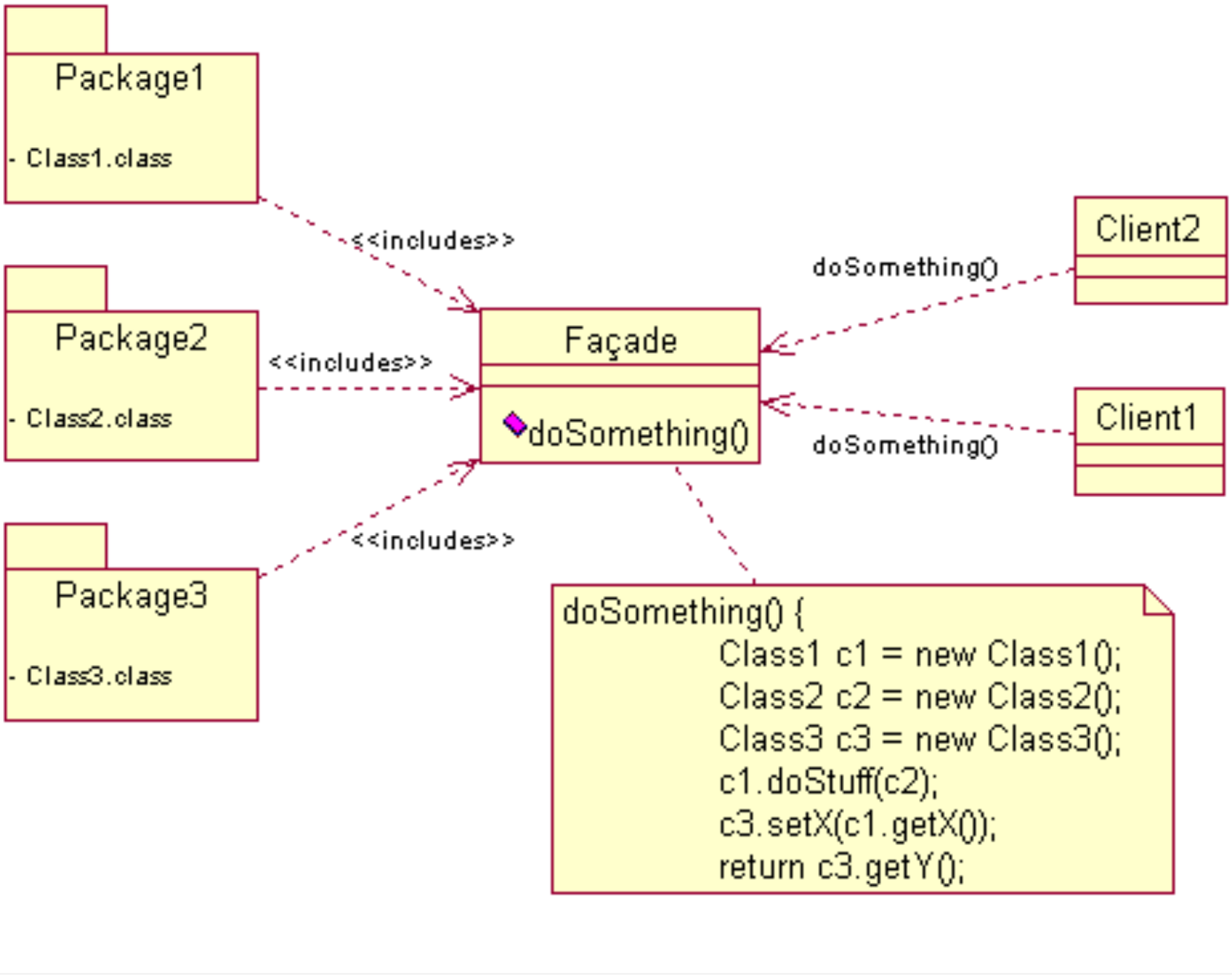
DECORATOR

```
[Ciboulette:] ./decorateur
eau+sucre+cafesoluble
lait+glacon+sucre+cafesoluble
[Ciboulette:]
```

FACADE

- permet de simplifier l'utilisation d'un sous-système complexe
  - parce que le sous-système contient de très/trop nombreux composants
  - parce que l'interface est très/trop compliquée et/ou mal-conçue
- attention facade ne doit pas masquer le sous-système, il doit juste offrir une simplification

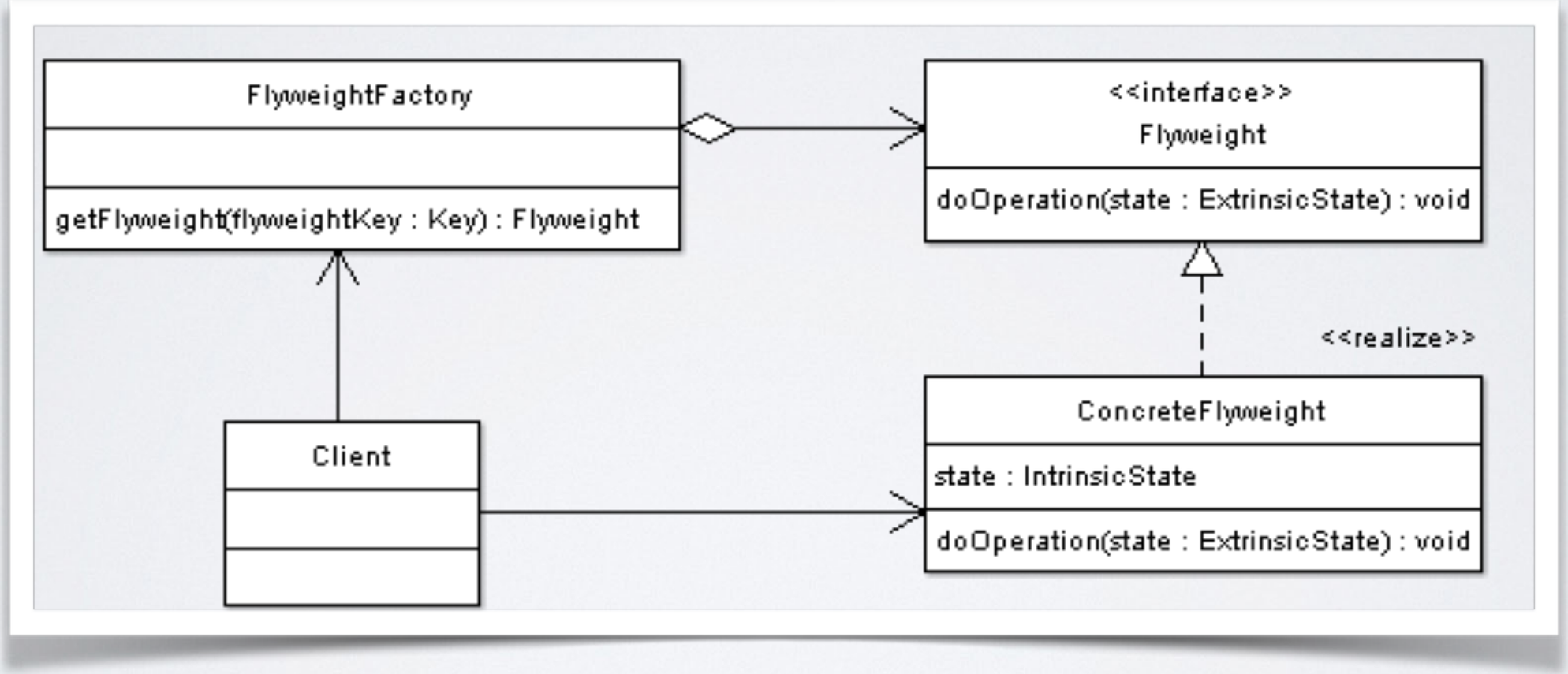
FACADE


Package1
- Class1.class
<<includes>>
Client2
doSomething()
Package2
<<includes>>
Façade
- Class2.class
doSomething()
doSomething()
Client1
<<includes>>
Package3
- Class3.class
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2);
c3.setX(c1.getX());
return c3.getY();

FLYWEIGHT

- poids-mouche est un pattern chargé d'éviter la création de trop nombreuses instances de petites classes lorsque la granularité conceptuelle le nécessite
- l'exemple typique est celui du concept de caractère+attributs dans un traitement de texte
- l'idée est
  - d'externaliser certaines données, ex. attributs
  - appeler une méthode prenant en paramètre les attributs (internalisation dynamique)

FLYWEIGHT


FlyweightFactory
getFlyweight(flyweightKey : Key) : Flyweight
<<interface>>
Flyweight
doOperation(state : ExtrinsicState) : void
<<realize>>
Client
ConcreteFlyweight
state : IntrinsicState
doOperation(state : ExtrinsicState) : void

FLYWEIGHT

```
class ImagePion {
public:
  virtual void drawAt(int,int)=0;
};

class _ImagePion : public ImagePion {
private:
  string name; // lots of datas inside...
public:
  _ImagePion(string name) : name(name) {
    cout << "création _ImagePion " << name << endl;
  }
  void setName(string n) { name = n; }
  void drawAt(int x,int y) {
    cout << name << " at " << x << ',' << y << endl;
  }
};
```

FLYWEIGHT

```
class FlyWeightPourJeu {
private:
  static ImagePion *blanc, *noir;
public:
  static ImagePion *getImagePion(string n) {
    if (n=="blanc") {
      if (blanc==0) blanc = new _ImagePion("blanc");
      return blanc;
    }
    if (n=="noir") {
      if (noir==0) noir = new _ImagePion("noir");
      return noir;
    }
    return 0;
  }
};
```

FLYWEIGHT

```
ImagePion *FlyWeightPourJeuDeDame::blanc=0;
ImagePion *FlyWeightPourJeuDeDame::noir=0;


class Pion {
protected:
   int x, y;
public:
   Pion(int x,int y) : x(x), y(y) {}
   virtual void draw()=0;
};
```

FLYWEIGHT

```
class PionBlanc : public Pion {
public:
  PionBlanc(int x,int y) : Pion(x,y) { cout << "création pion blanc" << endl; }
  void draw() {
    ImagePion *ip = FlyWeightPourJeuDeDame::getImagePion("blanc");
    ip->drawAt(x,y);
  }
};

class PionNoir : public Pion {
public:
  PionNoir(int x,int y) : Pion(x,y) { cout << "création pion noir" << endl; }
  void draw() {
    ImagePion *ip = FlyWeightPourJeuDeDame::getImagePion("noir");
    ip->drawAt(x,y);
  }
};
```

FLYWEIGHT

```
int main(int argc,char *argv[]) {
  Pion *p[6] = { new PionBlanc(1,1),
                 new PionNoir(1,2),
                 new PionBlanc(1,3),
                 new PionNoir(1,2),
                 new PionBlanc(3,1),
                 new PionBlanc(4,2) };
  cout << "avant draws" << endl;
  for (int i=0; i<6; i++) p[i]->draw();
   return 0;
}
```

FLYWEIGHT

```
[Ciboulette:] ./flyweight
création pion blanc
création pion noir
création pion blanc
création pion noir
création pion blanc
création pion blanc
création pion blanc
avant draws
création _ImagePion blanc
blanc at 1,1
création _ImagePion noir
noir at 1,2
blanc at 1,3
noir at 1,2
blanc at 3,1
blanc at 4,2
[Ciboulette:]
```

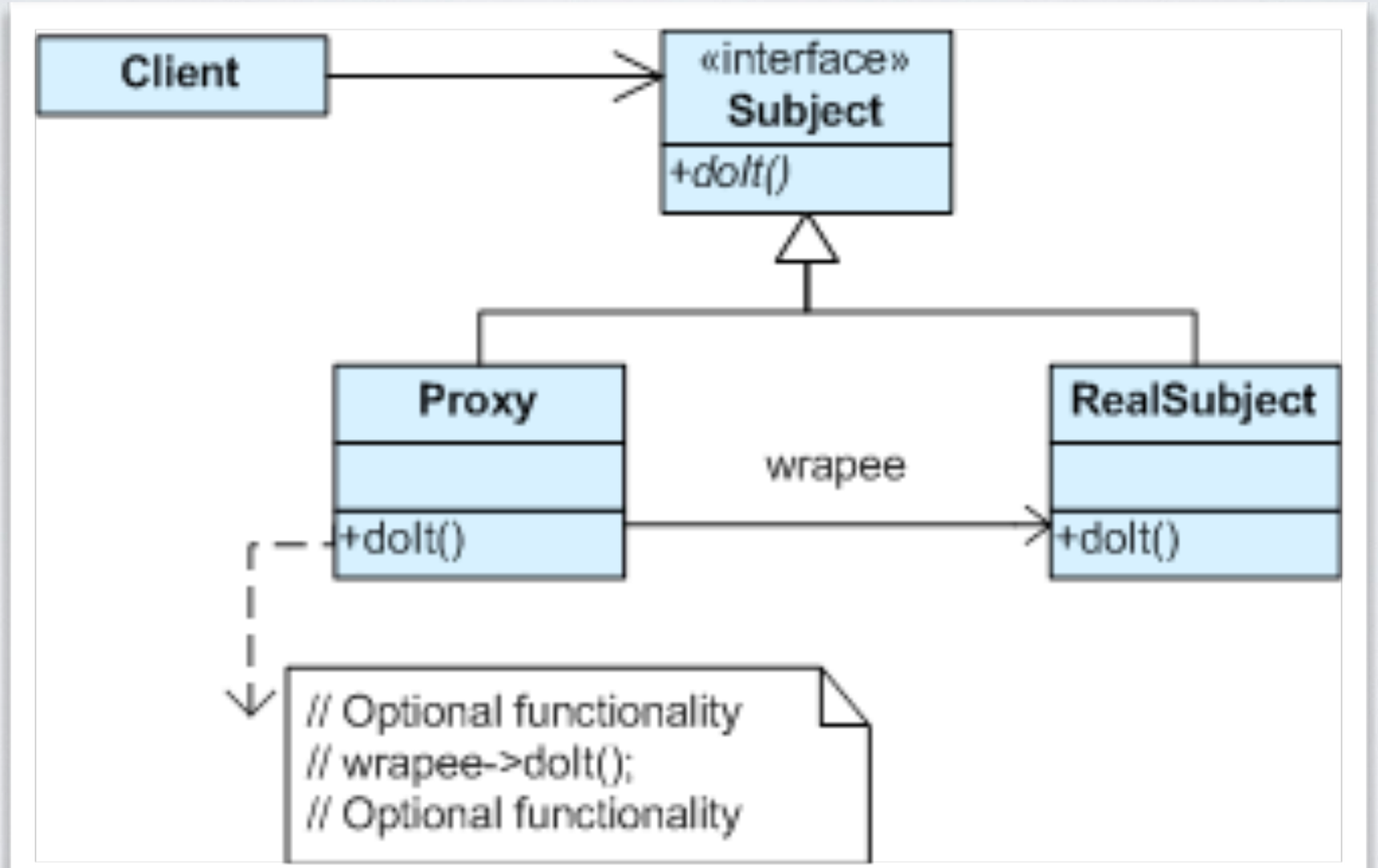
FLYWEIGHT

- la principale difficulté de flyweight est de nécessiter de repenser le concept concerné en éclatant les données en :
  - les données internes et qui identifieront les instances
  - les données externes et qui seront utilisées pour utiliser les instances
    - les méthodes permettant de récupérer des instances recevront ces données en paramètre

PROXY

- délégation / mandataire / proxy est un pattern permettant de manipuler à l'identique un objet caché à travers un représentant
- les quatre cas d'utilisation courant sont :
- le Remote Proxy qui permet d'accéder à un objet distant comme s'il était localement présent (stub)
- le Virtual Proxy qui permet de retarder les opérations coûteuses jusqu'au moment où il est nécessaire de les effectuer (deferred image loading)
- le Smart Reference Proxy (smart pointer)
- le Access Proxy qui fournit un contrôle d'accès

PROXY


Client
«interface»
Subject
+doIt()
Proxy
+doIt()
wrapee
RealSubject
+doIt()
// Optional functionality
// wrapee->doIt();
// Optional functionality

PROXY

```
class Canape {};

class Vendeur {
public:
  virtual int getCanape(Canape *&v)=0;
};

class Honnete : public Vendeur {
public:
  int getCanape(Canape *&v) { v = new Canape; return 1750; }
};

class Escroc : public Vendeur {
public:
  int getCanape(Canape *&v) { v = new Canape; return 19750; }
};
```

PROXY

```
class Mandataire : public Vendeur {
private:
  Vendeur *vendeur;
public:
  Mandataire() {
    // Au début donner confiance...
    vendeur = new Honnete;
  }
  int getCanape(Canape *&v) {
    int prix = vendeur->getCanape(v);
    // C'est bon, ils ont confiance, je vais les escroquer!
    delete vendeur;
    vendeur = new Escroc;
    return prix;
  }
  ~Mandataire() { delete vendeur; }
};
```

PROXY

```
void acheteDeuxCanapes(Vendeur &v) {
  Canape *c;
  int prix = v.getCanape(c);
  cout << "J'ai eu un canapé neuf pour " << prix << endl;
  if (prix>5000)
    cout << "Je me suis fait escroquer!" << endl;
  delete c;
  prix = v.getCanape(c);
  cout << "J'ai eu un canapé neuf pour " << prix << endl;
  if (prix>5000)
    cout << "Je me suis fait escroquer!" << endl;
  delete c;
}
```

PROXY

```
int main() {
   Honnete Jean;
   Escroc Baptiste;
   Mandataire Yunes;
   acheteDeuxCanapes(Jean);
   acheteDeuxCanapes(Baptiste);
   acheteDeuxCanapes(Yunes);
   return 0;
}
```

PROXY

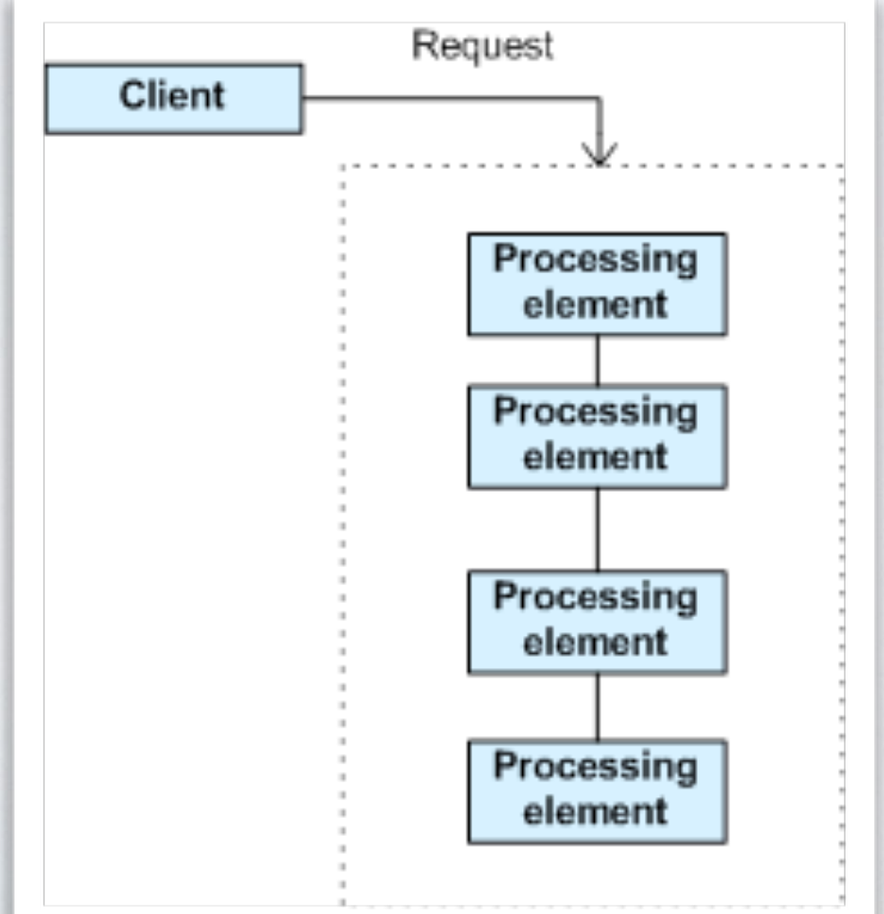
```
[Ciboulette:] ./proxy
J'ai eu un canapé neuf pour 1750
J'ai eu un canapé neuf pour 1750
J'ai eu un canapé neuf pour 19750
Je me suis fait escroquer!
J'ai eu un canapé neuf pour 19750
Je me suis fait escroquer!
J'ai eu un canapé neuf pour 1750
J'ai eu un canapé neuf pour 19750
Je me suis fait escroquer!
[Ciboulette:]
```

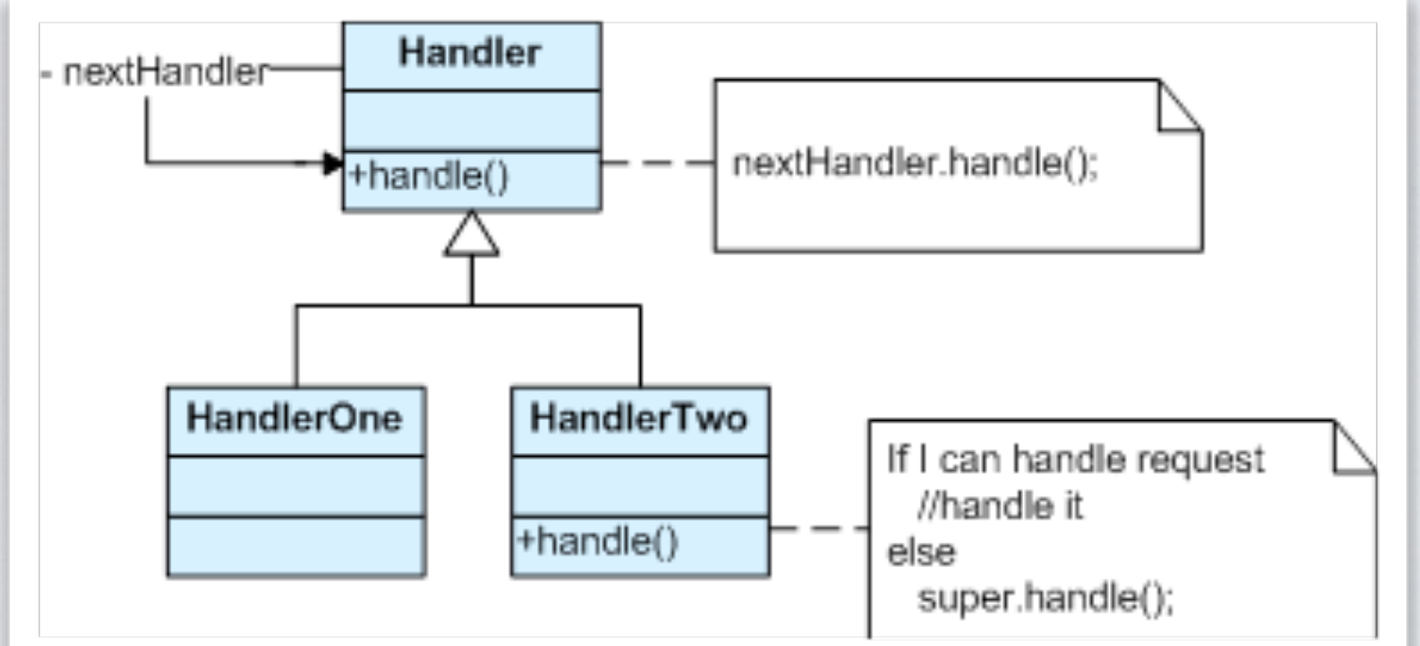
COMPORTEMENT

CHAIN OF RESPONSIBILITY

- le pattern chaîne de responsabilité est utilisé pour permettre à un nombre d'objet quelconque de réaliser un traitement
  - chaque objet capable de réaliser le traitement s'inscrit dans la chaîne
  - la chaîne est parcourue jusqu'à trouver quelqu'un qui accepte de prendre en charge le traitement

CHAIN OF RESPONSIBILITY





CHAIN OF RESPONSIBILITY

```
class Process {
  Process *next;
protected:
  Process() : next(0) {}
public:
  void add(Process *p) {
    if (next) next->add(p);
    else next = p;
  }
  virtual void doIt() {
    if (next) next->doIt();
    else cout << "Tout le monde s'en fiche. Traitement par défaut..." << endl;
  }
};
```

CHAIN OF RESPONSIBILITY

```
class RealProcess : public Process {
public:
  void doIt() {
    if (rand()%100<75) {
      cout << "non ";
      Process::doIt();
    }
    else
      cout << "Ok c'est bon je m'en occupe" << endl;
  }
};
```

CHAIN OF RESPONSIBILITY

```
class Chaine {
  Process *racine;
public:
  Chaine(Process &p) : racine(&p) {}
  Chaine &add(Process &p) {
    racine->add(&p);
    return *this;
}
  void traiteRequete() {
    if (racine) racine->doIt();
    else
      cout << "Personne pour s'en occuper" << endl;
  }
};
```

CHAIN OF RESPONSIBILITY

```
int main() {
  srand(time(NULL));
  RealProcess p1, p2, p3, p4, p5, p6;
  Chaine c(p1);
  c.add(p2).add(p3);
  c.traiteRequete();
  c.traiteRequete();
  c.add(p4).add(p5).add(p6);
  c.traiteRequete();
  return 0;
}
```

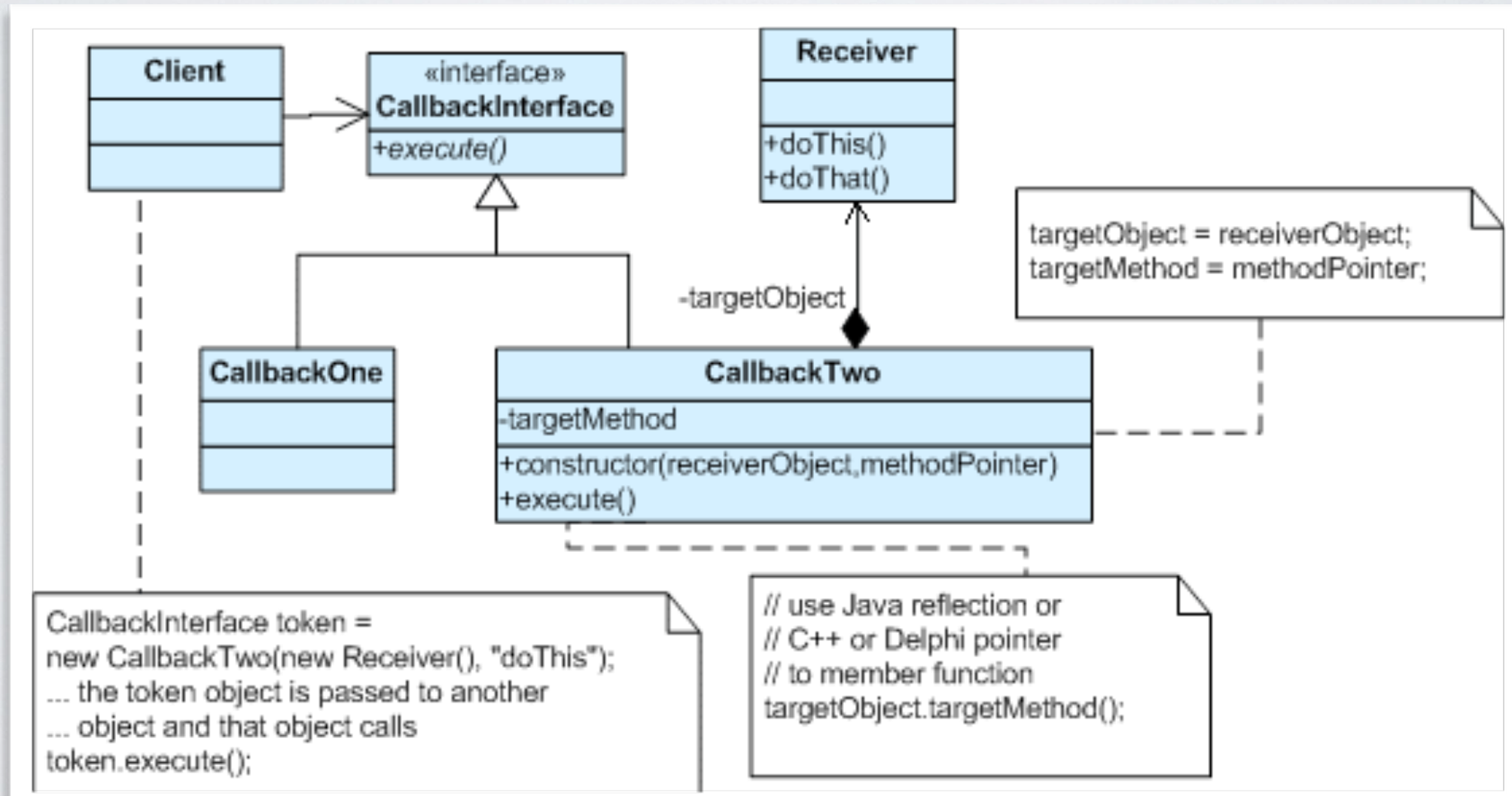
CHAIN OF RESPONSIBILITY

```
[Ciboulette:] ./responsability
Ok c'est bon je m'en occupe
non Ok c'est bon je m'en occupe
Ok c'est bon je m'en occupe
[Ciboulette:] ./responsability
non non non Tout le monde s'en fiche. Traitement par défaut...
Ok c'est bon je m'en occupe
non non Ok c'est bon je m'en occupe
[Ciboulette:]
```

COMMAND

- le pattern commande réifie les appels de méthodes en créant des messages (objets)
  - typique des interfaces graphiques car ce pattern est une version objet des callbacks

COMMAND


Client
«interface»
CallbackInterface
+execute()
Receiver
+doThis()
+doThat()
targetObject = receiverObject;
targetMethod = methodPointer;
-targetObject
CallbackOne
CallbackTwo
-targetMethod
+constructor(receiverObject,methodPointer)
+execute()
CallbackInterface token =
new CallbackTwo(new Receiver(), "doThis");
... the token object is passed to another
... object and that object calls
token.execute();
// use Java reflection or
// C++ or Delphi pointer
// to member function
targetObject.targetMethod();

COMMAND

```
class Commande {
public:
  virtual void faitQuelqueChose()=0;
};
```

COMMAND

```
class RepareLeMoteur : public Commande {
public:
  void faitQuelqueChose() {
    cout << "Je répare le moteur" << endl;
  }
};

class DeboucherLaBaignoire : public Commande {
public:
  void faitQuelqueChose() {
    cout << "Je débouche la baignoire" << endl;
  }
};
```

COMMAND

```
class Reveil {
private:
  vector<Commande *> commandes;
public:
  void enregistre(Commande &c) {
    commandes.push_back(&c);
  }
  void doIt() {
    for (int i=0; i<commandes.size(); i++) {
      commandes[i]->faitQuelqueChose();
    }
  }
};
```

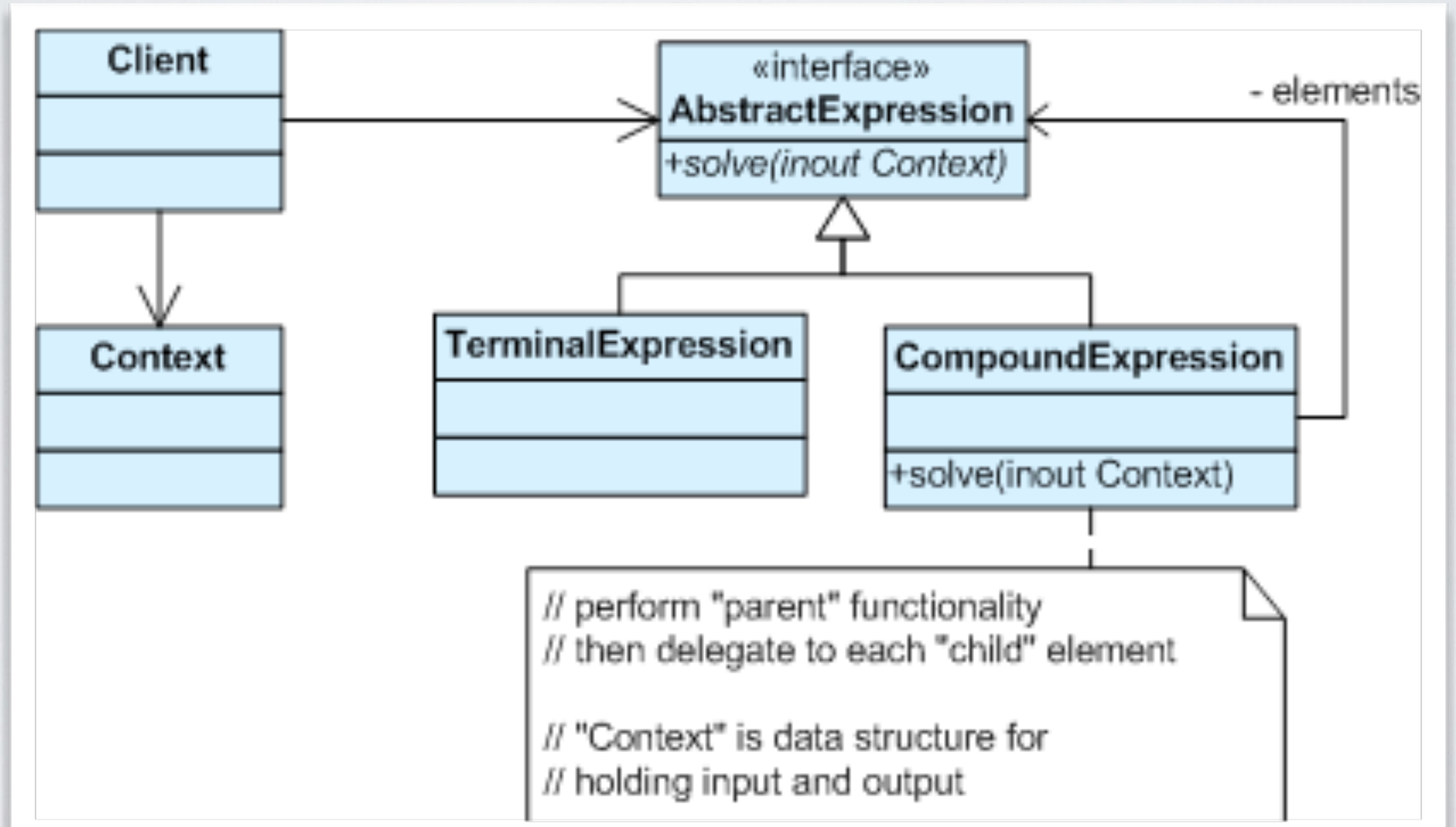
COMMAND

```
int main() {
  Reveil reveil;
  DeboucherLaBaignoire d;
  RepareLeMoteur r;
  reveil.enregistre(r);
  reveil.enregistre(d);
  //  long after
  reveil.doIt();
}
```

INTERPRETER

- interpréteur permet d'intégrer un langage à interpréter
  - il facilite l'écriture d'expressions dans le langage du domaine

INTERPRETER



INTERPRETER

```
class ExpressionAbstraite {
public:
  virtual int compute(Contexte &) = 0;
};

class Constante : public ExpressionAbstraite {
  int valeur;
public:
  Constante(int v) : valeur(v) {}
  int compute(Contexte &) { return valeur; }
};

class Variable : public ExpressionAbstraite {
  string id;
public:
  Variable(string id) : id(id) {}
  int compute(Contexte &c) { return c[id]; }
};
```

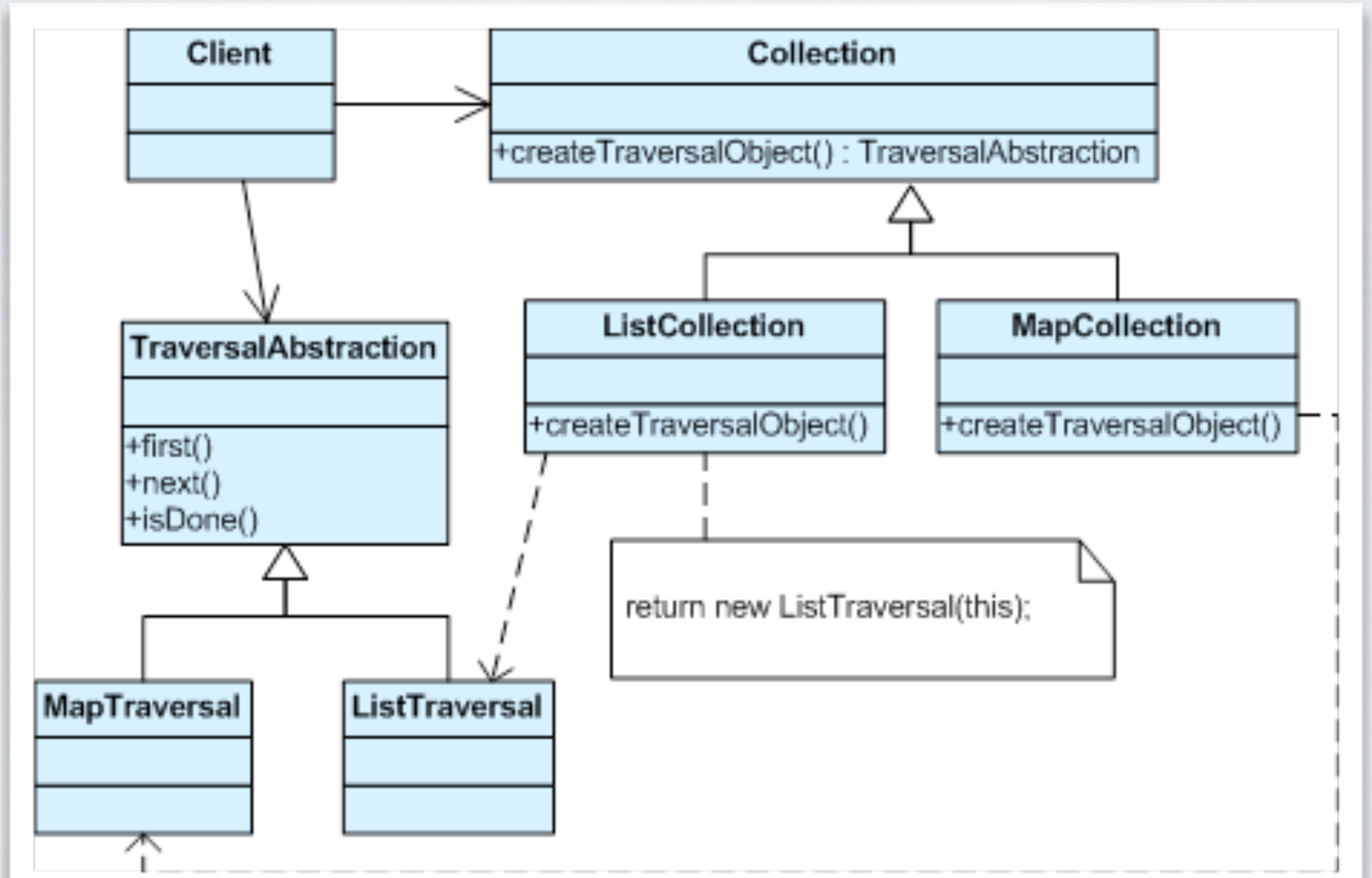
INTERPRETER

```
class Addition : public ExpressionAbstraite {
  ExpressionAbstraite *e1, *e2;
public:
  Addition(ExpressionAbstraite &e1,ExpressionAbstraite &e2) : e1(&e1), e2(&e2) {}
  Addition(ExpressionAbstraite *e1,ExpressionAbstraite *e2) : e1(e1), e2(e2) {}
  int compute(Contexte &c) { return e1->compute(c) + e2->compute(c); }
};

int main() {
  Constante quatre(4), sept(7);
  Variable v1("a"), v2("b");
  Contexte c;
  c["a"] = 2; c["b"] = 12;
  ExpressionAbstraite *a1 = new Addition(quatre,v2);
  ExpressionAbstraite *a2 = new Addition(v1,sept);
  ExpressionAbstraite *e = new Addition(a1,a2);
  cout << e->compute(c) << endl;
  c["a"] = 20; c["b"] = 121;
  cout << e->compute(c) << endl;
  delete a1; delete a2; delete e;
```

ITERATOR

- itérateur est un pattern suffisamment connu pour ne pas insister ???
- il permet d'abstraite le parcours d'une collection
- c'est un « pointeur » ou « indice » généralisé

ITERATOR


Client
Collection
+createTraversalObject() : TraversalAbstraction
TraversalAbstraction
+first()
+next()
+isDone()
ListCollection
+createTraversalObject()
MapCollection
+createTraversalObject()
return new ListTraversal(this);
MapTraversal
ListTraversal

ITERATOR

```
class Iterateur {
public:
   virtual int nextElement()=0;
   virtual bool aLaFin()=0;
};
```

ITERATOR

```
class Collection {
private:
  int valeurs[10];
  class IterateurInverse : public Iterateur {
  private:
    Collection *collection;
    int i;
  public:
    IterateurInverse(Collection *c) : collection(c), i(9) {}
    int nextElement() { return collection->valeurs[i--]; }
    bool aLaFin() { return i==-1; }
  };
public:
  Collection() {
    for (int i=0; i<10; i++) valeurs[i] = i+10;
  }
  Iterateur *getIterateur() {
    return new IterateurInverse(this);
  }
};
```

ITERATOR

```
int main() {
  Collection uneCollection;
  Iterateur *i = uneCollection.getIterateur();
  while (!i->aLaFin()) {
    cout << i->nextElement() << endl;
  }
  delete i;
  return 0;
}
```

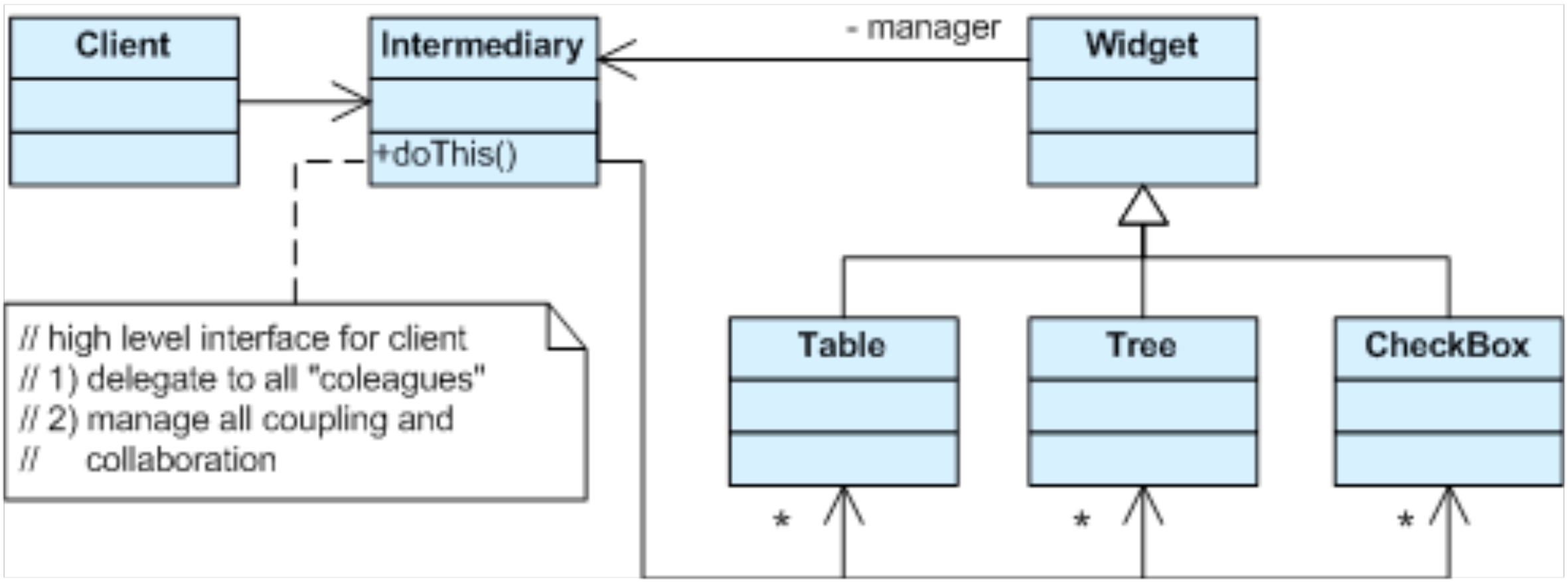
ITERATOR

```
[Ciboulette:] ./iterator
19
18
17
16
15
14
13
12
11
10
[Ciboulette:]
```

MEDIATOR

- le médiateur est un pattern qui permet de simplifier la gestion de la communication entre objets dans le cas d'un ensemble important d'objet et lorsque la maintenance de l'état du système nécessite des interactions compliquées

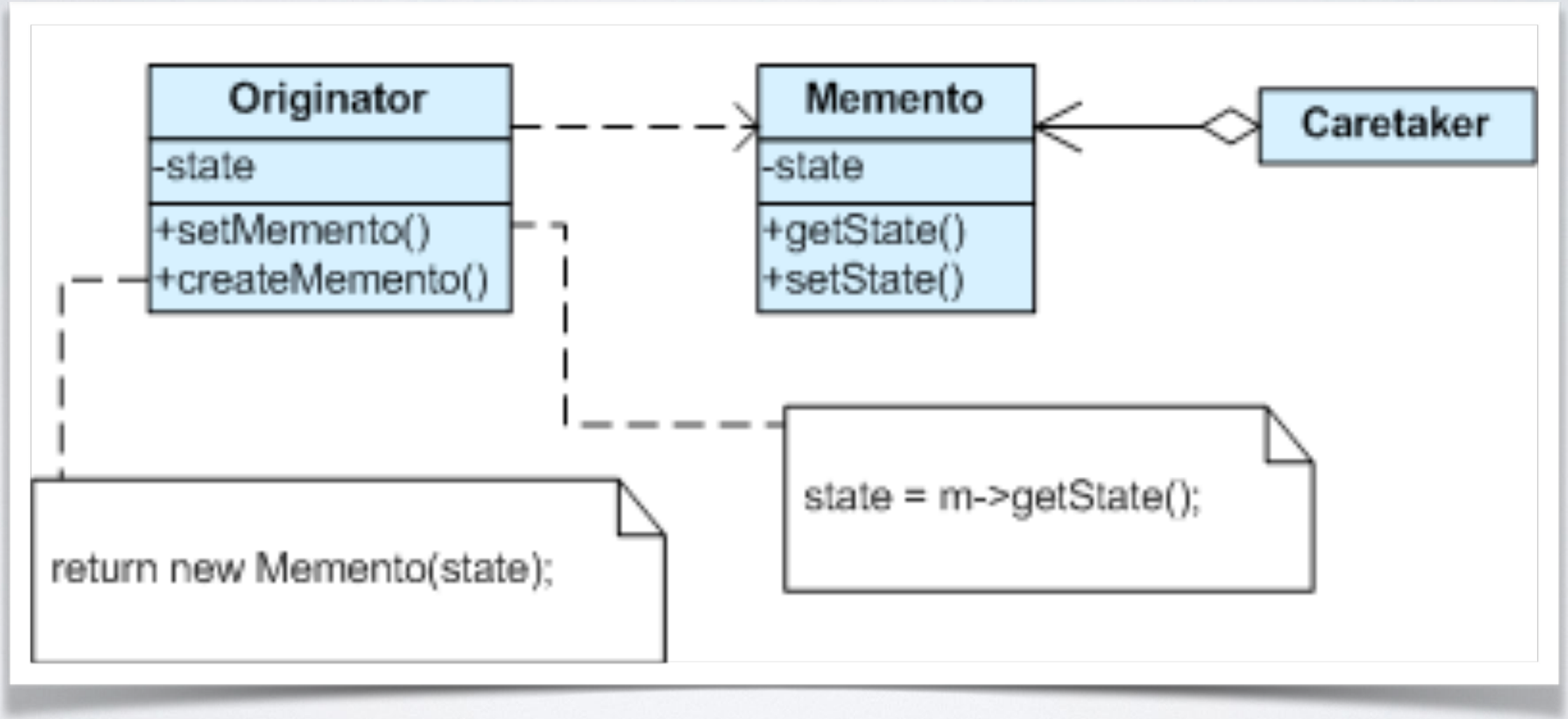
MEDIATOR


Client
Intermediary
+doThis()
- manager
Widget
// high level interface for client
// 1) delegate to all "coleagues"
// 2) manage all coupling and
//   collaboration
Table
Tree
CheckBox
*
*
*

MEMENTO

- mémento est un pattern dont l'objet principal est la conservation d'un état cohérent auquel on pourra revenir (rollback, undo, etc)

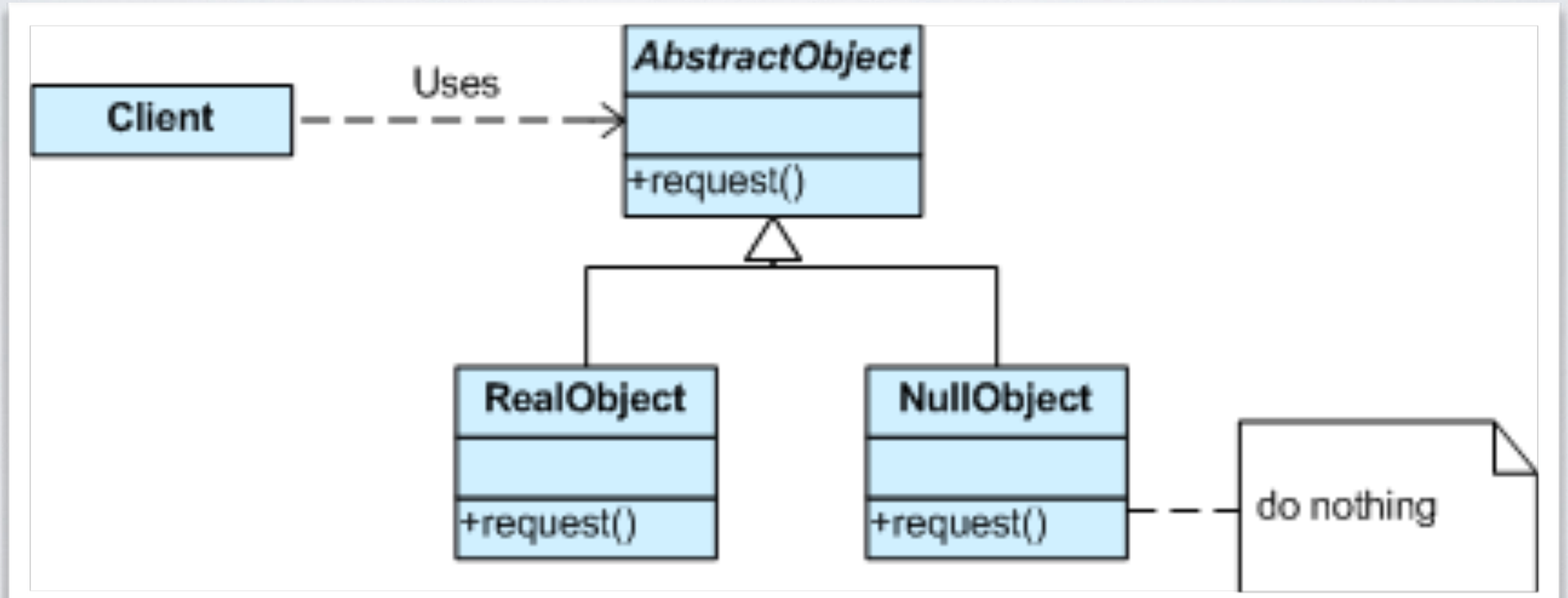
MEMENTO


Originator
-state
+setMemento()
+createMemento()
Memento
-state
+getState()
+setState()
Caretaker
return new Memento(state);
state = m->getState();

NULL OBJECT

- l'objet vide ou Null Object est un classique du prototypage. Il permet de tester la validité de la structure et des interactions

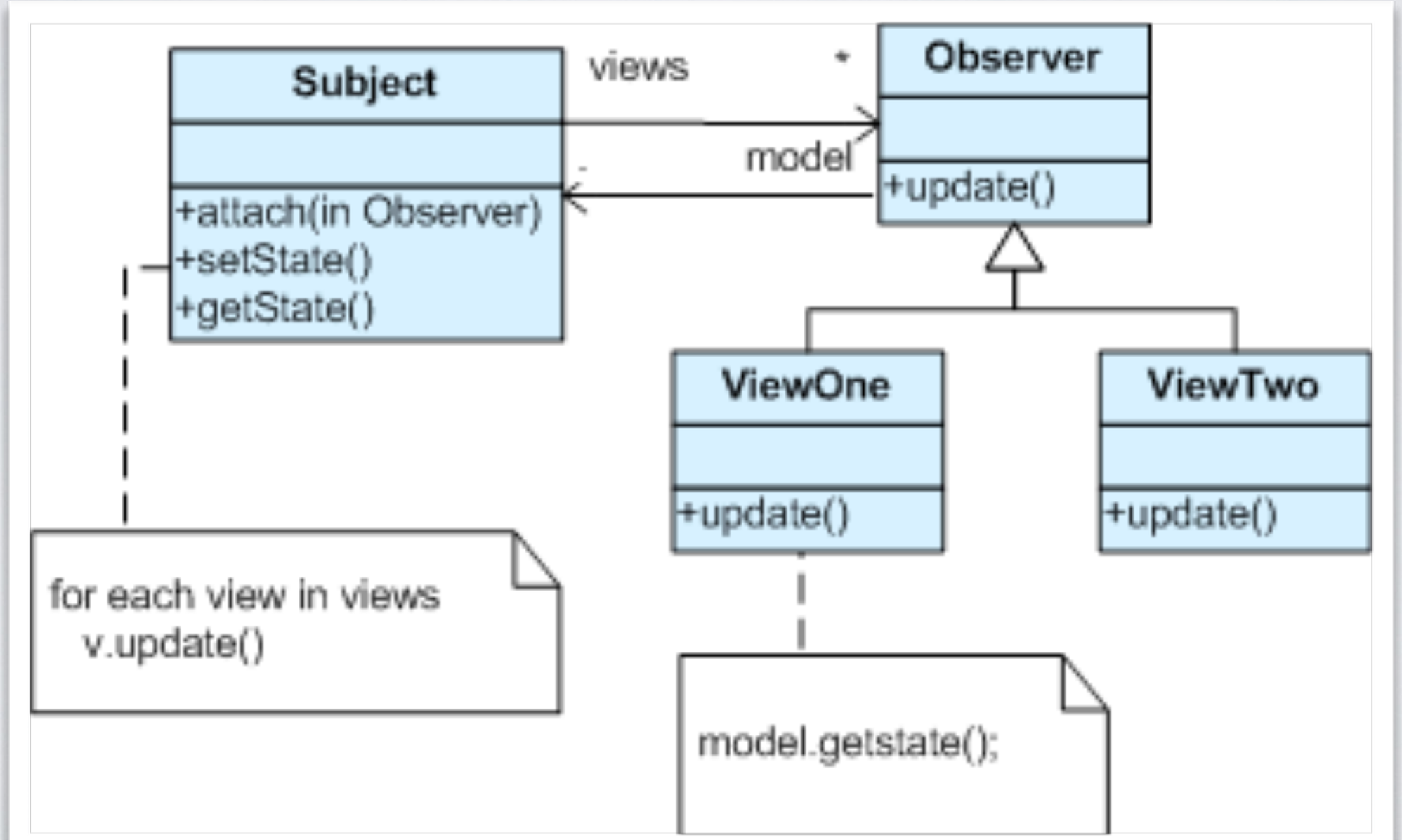
NULL OBJECT


Client
Uses
AbstractObject
+request()
RealObject
+request()
NullObject
+request()
do nothing

OBSERVER

- observateur est un pattern permettant à un modèle de prévenir les objets intéressés qu'il a changé d'état
- ce pattern utilise un mécanisme de notification pour renverser la situation
  - les observateurs s'enregistrent auprès de l'observable
  - l'observable notifie d'un changement les observateurs enregistrés lorsqu'il le juge nécessaire

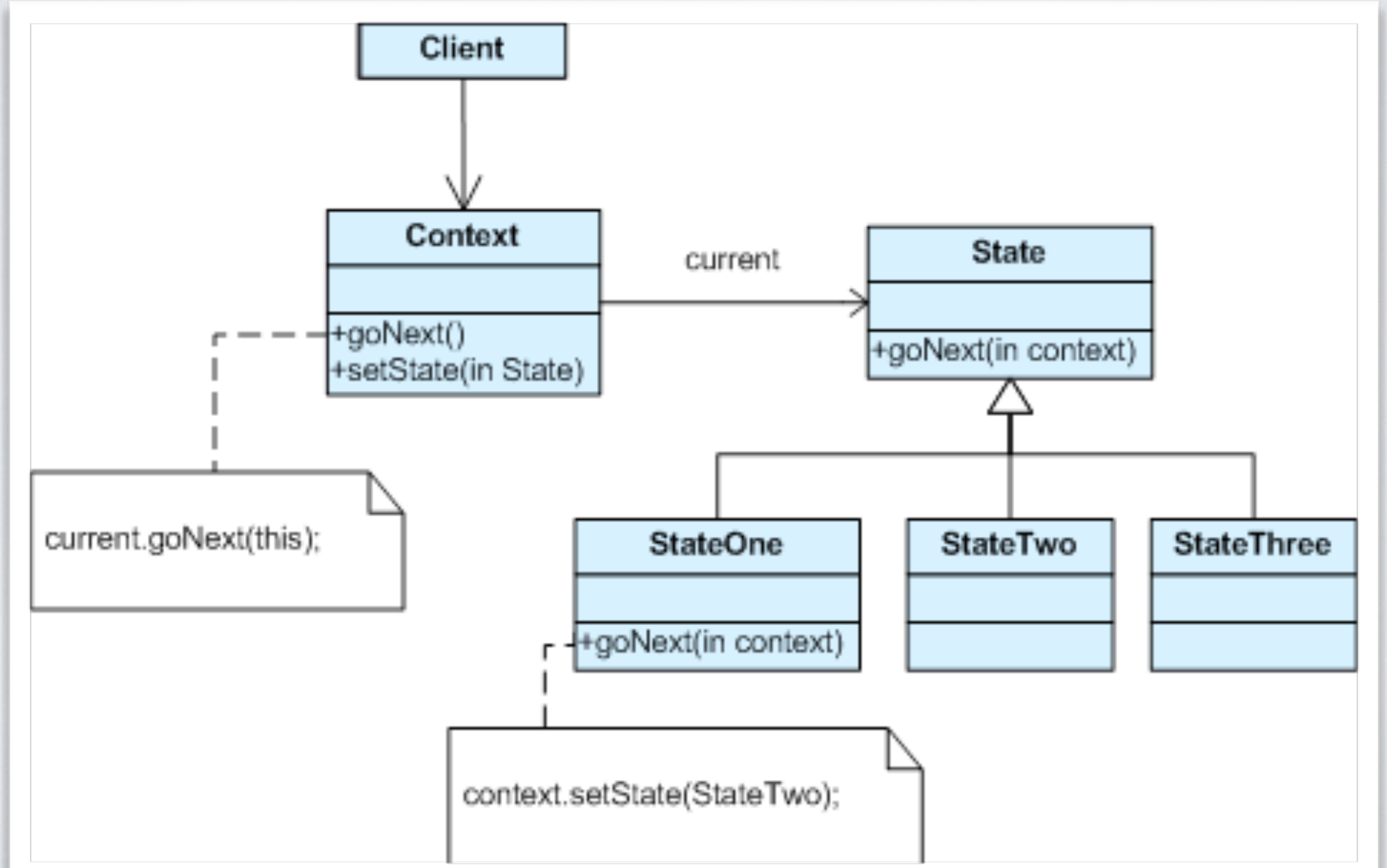
OBSERVER


Subject
views
*
Observer
model
+update()
+attach(in Observer)
+setState()
+getState()
ViewOne
+update()
ViewTwo
+update()
for each view in views
v.update()
model.getstate();

STATE

- le pattern état permet d'obtenir un comportement dynamique flexible pour un objet donné
  - il permet par exemple d'obtenir une mutation comportementale

STATE


Client
Context
+goNext()
+setState(in State)
current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);

STATE

```
class Etat {
public:
   virtual void parite()=0;
   virtual Etat *next()=0;
};
```

STATE

```
class Pair : public Etat {
public:
  void parite() { cout << "pair" << endl; }
  Etat *next();
};

class Impair : public Etat {
public:
  void parite() { cout << "impair" << endl; };
  Etat *next() { new Pair; }
};

// forward problem
Etat *Pair::next() { return new Impair; }
```

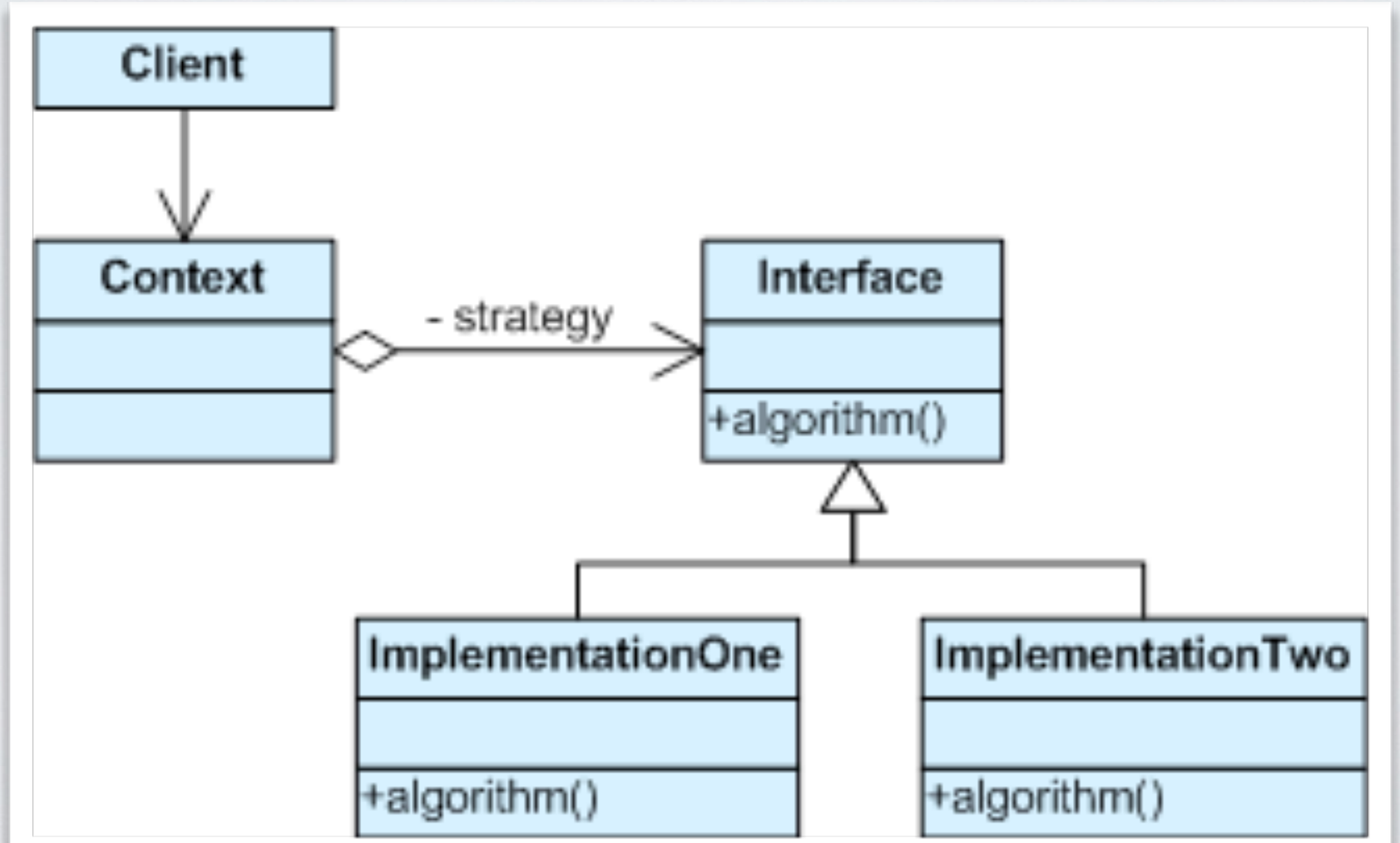
STATE

```
class Automate {
private:
  Etat *e;
public:
  Automate() { e = new Pair; }
  void doIt() { e->parite(); e = e->next(); }
};

int main() {
  Automate a;
  for (int i=0; i<12; i++) {
    a.doIt();
  }
  return 0;
}
```

STRATEGY

- stratégie est un pattern permettant de fixer un comportement à un objet donné
- cette fixation n'est pas nécessairement liée à l'état de l'objet lui-même comme dans state mais est plus libre
- de plus cette fixation est en générale irrémédiable

STRATEGY


Client
Context
- strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

STRATEGY

```
class Comportement {
public:
  virtual void reaction()=0;
};
```

STRATEGY

```
class Agressif : public Comportement {
public:
  void reaction() {
    cout << "Dégage, espèce d'abruti!" << endl;
  }
};

class Calme : public Comportement {
public:
  void reaction() {
    cout << "Bonjour ami, bienvenue..." << endl;
  }
};
```

STRATEGY

```
class DeNirosStyle : public Comportement {
public:
  void reaction() {
    cout << "You talkin' to me?" << endl;
  }
};

class Bavard : public Comportement {
public:
  void reaction() {
    cout << "Bla bla bla bla bla bla bla bla..." << endl;
  }
};
```

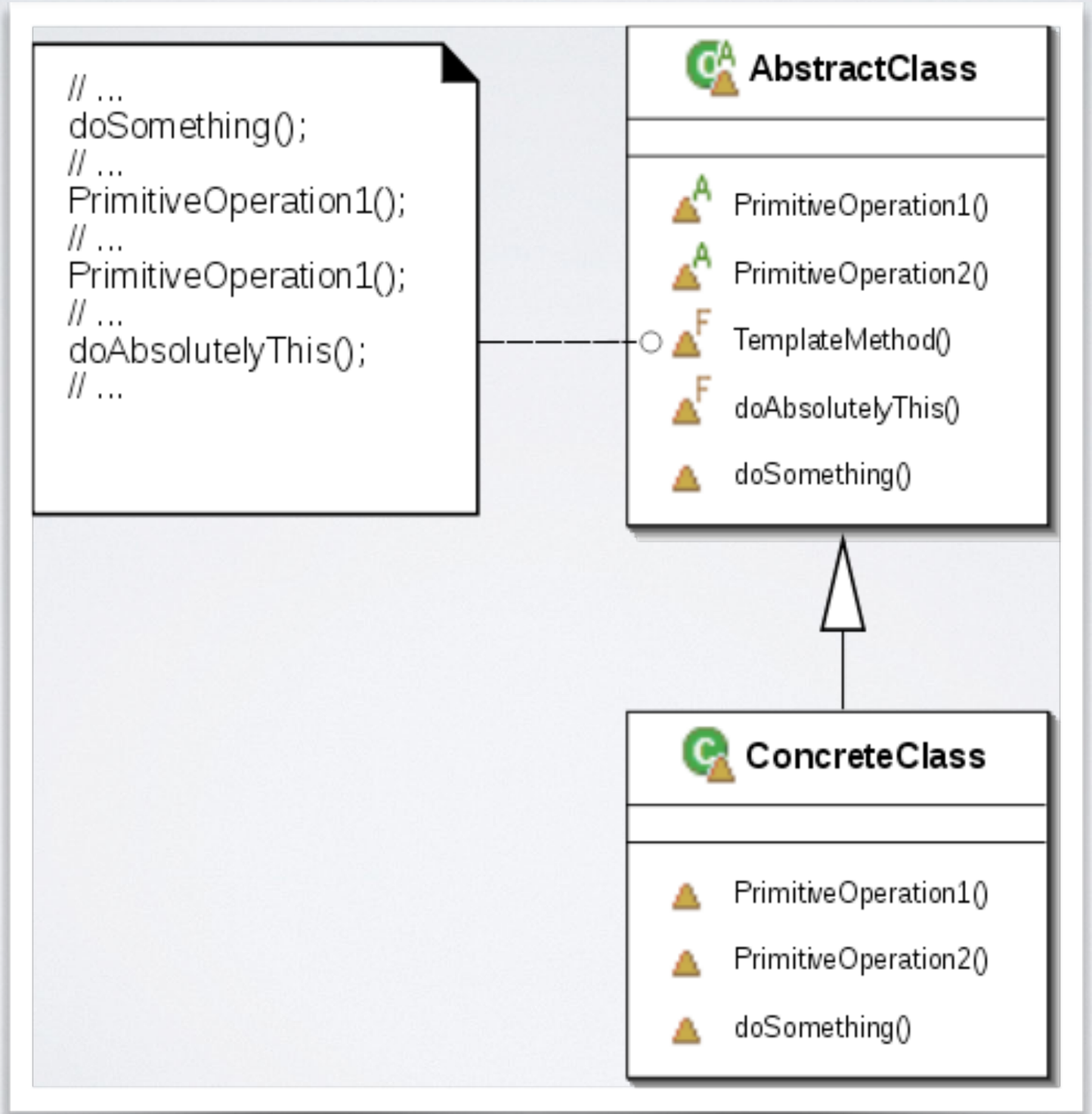
STRATEGY

```
class Humain {
  Comportement *c;
public:
  void setComportement(Comportement *c) {
    this->c = c;
  }
  Humain() { setComportement(new Calme()); }
  void bonjour() { c->reaction(); }
};

int main() {
  Humain h;
  h.bonjour();
  h.setComportement(new Bavard());
  h.bonjour();
}
```

TEMPLATE METHOD

- patron de méthode est un pattern de généricité ne faisant pas appel aux templates du langage
- une classe partiellement abstraite implémente un algorithme dont certains aspects ne seront définis que dans des sous-classes adéquates

TEMPLATE METHOD



TEMPLATE METHOD

```
class Voyage {
public:
  virtual void lundi()    =0;
  virtual void mardi()    =0;
  virtual void mercredi()=0;
  virtual void jeudi()    =0;
  virtual void vendredi()=0;
  void organise() {
    lundi(); mardi(); mercredi();
    jeudi(); vendredi();
  }
};
```

TEMPLATE METHOD

```
class VoyageDAffaire : public Voyage {
public:
    virtual void lundi() {
        cout << "rencontrer Alain Carrefour" << endl;
    };
    virtual void mardi() {
        cout << "rencontrer Georges Auchan" << endl;
    };
    virtual void mercredi() {
        cout << "rencontrer Robert Continent" << endl;
    };
    virtual void jeudi() {
        cout << "rencontrer Charles Bricorama" << endl;
    };
    virtual void vendredi() {
        cout << "rencontrer Patrick SuperU" << endl;
    };
};
```

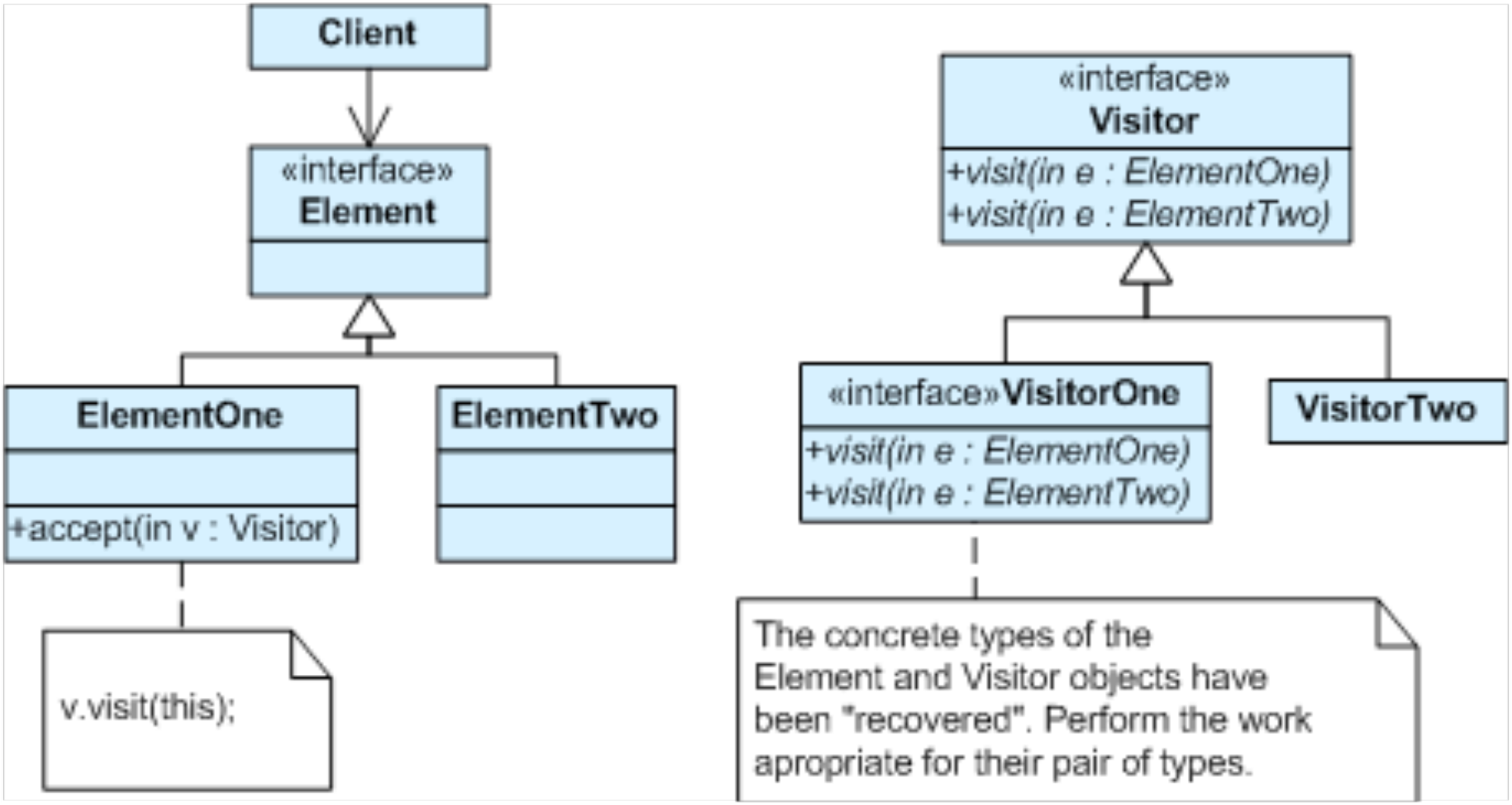
TEMPLATE METHOD

```
class VoyageDAgrement : public Voyage {
public:
    virtual void lundi() {
        cout << "visiter tour eiffel" << endl;
    };
    virtual void mardi() {
        cout << "visiter montmartre" << endl;
    };
    virtual void mercredi() {
        cout << "visiter moulin rouge" << endl;
    };
    virtual void jeudi() {
        cout << "visiter notre-dame" << endl;
    };
    virtual void vendredi() {
        cout << "visiter le louvre" << endl;
    };
};
```

VISITOR

- visiteur est un pattern important qui permet d'obtenir un comportement sur des objets sans modifier l'objet lui-même
  - la technique est aussi connue sous le nom de double dispatch (envoi croisé)

VISITOR


Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
v.visit(this);
ElementTwo
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface»VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
VisitorTwo
The concrete types of the
Element and Visitor objects have
been "recovered". Perform the work
apropriate for their pair of types.

VISITOR

```
class Visitor; // forward...

class Couleur {
public:
  virtual void accepte(Visitor &v) = 0;
};

class Vert: public Couleur {
public:
  void accepte(Visitor &v);
  string vert() { return "Vert"; }
};
```

VISITOR

```
class Rouge: public Couleur {
public:
  void accepte(Visitor &v);
  string rouge() { return "Rouge"; }
};

class Bleu: public Couleur {
public:
  void accepte(Visitor &v);
  string bleu() { return "Bleu"; }
};
```

VISITOR

```
class Visitor {
public:
  virtual void visite(Vert *e)  = 0;
  virtual void visite(Rouge *e) = 0;
  virtual void visite(Bleu *e)  = 0;
};

//  double dispatch...
void Vert::accepte(Visitor &v) { v.visite(this); }
void Rouge::accepte(Visitor &v){ v.visite(this); }
void Bleu::accepte(Visitor &v) { v.visite(this); }
```

VISITOR

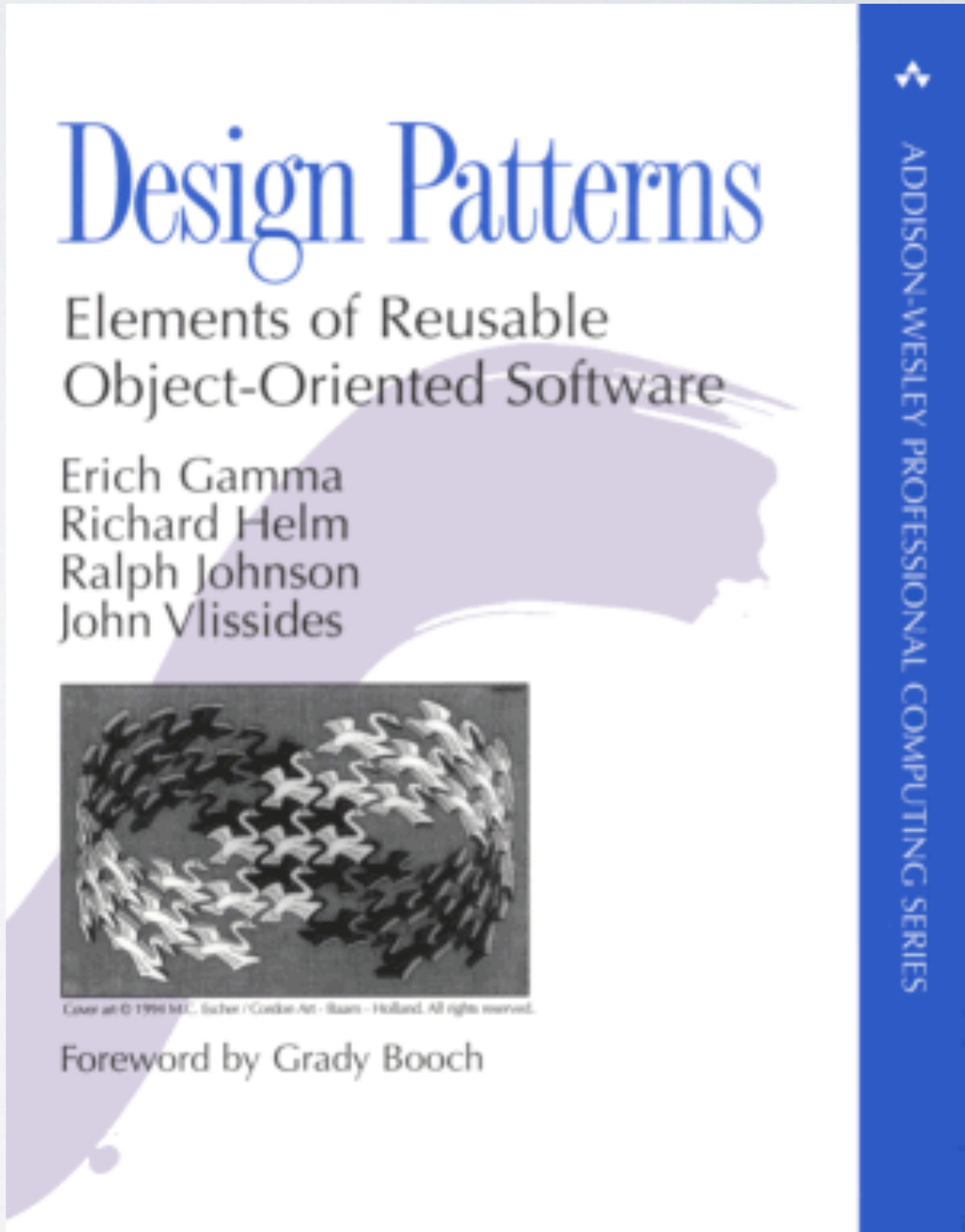
```
class Compteur : public Visitor {
  int rouge, vert, bleu;
public:
  Compteur() : rouge(0), vert(0), bleu(0) {}
  void visite(Vert *e)  { vert++; }
  void visite(Rouge *e) { rouge++; }
  void visite(Bleu *e)  { bleu++; }
  void print() {
    cout << rouge << " rouges, "
         << bleu << " bleus, "
         << vert << " verts" << endl;
  }
};
```

VISITOR

```
class Idiot : public Visitor {
  void visite(Vert *e) {
    cout << "salut toi le " + e->vert() << endl;
  }
  void visite(Rouge *e) {
    cout << "salut toi le " + e->rouge() << endl;
  }
  void visite(Bleu *e) {
    cout << "salut toi le " + e->bleu() << endl;
  }
};
```

VISITOR

```
int main() {
  Couleur *list[] = { new Vert, new Rouge,
                      new Bleu, new Rouge,
                      new Rouge };
  Compteur cpt;
  Idiot idiot;
  for (int i = 0; i < 5; i++) list[i]->accepte(cpt);
  cpt.print();
  for (int i = 0; i < 5; i++) list[i]->accepte(idiot);
  return 0;
}
```

BIBLIOGRAPHIE

- *Design Patterns.*
  Catalogue des modèles de conception réutilisables
  - auteurs : E. Gamma, R. Helm, R. Johnson, J. Vlissides
  - éditeur : Vuibert