

LANGUAGE OBJ. AV.(C++) MASTER 1

Yan Jurski

U.F.R. d'Informatique
Université de Paris Cité

"DESIGN PATTERNS"

Introduction :

- des figures de style ?
- une taxonomie ?
(classification de pbs fréquents)

design pattern / motif conceptuel / patrons
de conception / motif de conception

Ils désignent des cas récurrents, identifiés,
d'architecture et de conception logicielles.

Ils décrivent des solutions "standards"

Exemple : Modèle-Vue-Contrôleur est un pattern, qui vise à obtenir un
couplage faible entre le cœur fonctionnel d'une application et son
interface utilisateur

« Le Bourgeois gentilhomme » (Molière)

Mr JOURDAIN : [...] il faut que je vous fasse une confidence. Je suis amoureux d'une personne de grande qualité, **je souhaiterais que vous m'aidassiez à écrire quelque chose** dans un petit billet que je veux laisser tomber à ses pieds.

MAÎTRE DE PHILO. : Sans doute. Sont-ce des vers que vous lui voulez écrire ?

Mr JOURDAIN : Non, non, point de vers.

MAÎTRE DE PHILO. : Vous ne voulez que de la prose ?

Mr JOURDAIN : Non, je ne veux **ni prose ni vers**.

MAÎTRE DE PHILO.: Il faut bien que ce soit l'un, ou l'autre.

Mr JOURDAIN : Pourquoi ?

MAÎTRE DE PHILO. : Par la raison, Monsieur, qu'**il n'y a pour s'exprimer que la prose, ou les vers**.

Mr JOURDAIN: Il n'y a que la prose ou les vers ?

MAÎTRE DE PHILO. : Monsieur : tout ce qui n'est point prose est vers ; et tout ce qui n'est point vers est prose.

Mr JOURDAIN: Et comme l'on parle qu'est-ce que c'est donc que cela ?

MAÎTRE DE PHILO. : De la prose.

Mr JOURDAIN : Quoi ! quand je dis: « Nicole, apportez-moi mes pantoufles, et me donnez mon bonnet de nuit », c'est de la prose ?

MAÎTRE DE PHILO. : Oui, Monsieur.

Mr JOURDAIN : Par ma foi ! **il y a plus de quarante ans que je dis de la prose sans que j'en sache rien, et je vous suis le plus obligé du monde de m'avoir appris cela.**

MONSIEUR JOURDAIN : Je voudrais donc lui mettre un billet (...) mais **je voudrais que cela fût mis d'une manière galante, que cela fût tourné gentiment.**

MAÎTRE DE PHILO. : Mettez que (... blah blah blah ...)

MONSIEUR JOURDAIN : Non, non, non, je ne veux point tout cela ; je ne veux que ce que je vous ai dit ...

MAÎTRE DE PHILO. : Il faut bien étendre un peu la chose.

MONSIEUR JOURDAIN : Non, vous dis-je, **je ne veux que ces seules paroles-là dans le billet ; mais tournées à la mode ; bien arrangées comme il faut. Je vous prie de me dire un peu, pour voir, les diverses manières dont on les peut mettre.**

MAÎTRE DE PHILO. : On les peut mettre premièrement comme vous avez dit : « bla bla bla ... » Ou bien : « bli bli bli ». Ou bien : « bla bli bla ». Ou bien : (etc ..)

MONSIEUR JOURDAIN : **Mais de toutes ces façons-là, laquelle est la meilleure ?**

MAÎTRE DE PHILO. : **Celle que vous avez dite** : « Belle Marquise, vos beaux yeux me font mourir d'amour ».

MONSIEUR JOURDAIN : **Cependant je n'ai point étudié, et j'ai fait cela tout du premier coup.** Je vous remercie de tout mon cœur, et vous prie de venir demain de bonne heure.

MAÎTRE DE PHILOSOPHIE : Je n'y manquerai pas. (Il sort)

Les motifs conceptuels ont été classés
en 3 catégories principales :

- création/construction
- structure
- comportement

Création / construction :

(problèmes liés à la création ou l'instanciation des objets)

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structure :

(Organisation structurelle, identification de concepts, découplage)

- Adapter,
- Bridge,
- Composite,
- Decorator,
- Facade,
- Flyweight,
- Proxy

Comportement :

(Organisation de la collaboration des objets)

- Callback,
- Chain of Responsibility,
- Command,
- Interpreter,
- Iterator,
- Mediator,
- Memento,
- Observer,
- State,
- Strategy,
- Template Method,
- Visitor

PATTERNS DE CRÉATION

le couple `new` + constructeur, qui endosserait seul le rôle de la création d'objet est un peu court pour contrôler suffisamment finement la subtilité de ce qu'on peut attendre.

`private`, `protected`, `public` viennent déjà le moduler.

Le constructeur de copie le complète.

On va travailler à mettre en place quelques concepts plus "haut niveau".

MÉTHODE DE FABRICATION ("FACTORY METHOD")

L'existence d'une classe abstraite, ayant plusieurs classes filles concrètes doit faire se poser la question de l'usage de ce motif : la classe abstraite permet de manipuler de façon abstraite les objets, mais leur création/instanciation nécessite l'usage explicite de l'une ou l'autre des classes filles.

Imposer à un utilisateur de la classe abstraite l'usage de new pour ces classes filles n'est pas toujours le plus pratique.

MÉTHODE DE FABRICATION ("FACTORY METHOD")

```
class Voiture {  
public:  
    virtual string getModele() const = 0;  
};
```

(Sans Factory Method)

```
class Clio : public Voiture {  
public:  
    string getModele() const;  
};
```

```
class Panda : public Voiture {  
public:  
    string getModele() const;  
};
```

```
string Clio::getModele() const { return "Clio"; }
```

```
string Panda::getModele() const { return "Panda"; }
```

ici l'utilisateur aura toute liberté de créer Clio ou Panda

MÉTHODE DE FABRICATION ("FACTORY METHOD")

```
class Voiture {  
public:  
    virtual string getModele() const = 0;  
    static Voiture *createVoiture(string origine);  
};
```

```
#include "Clio.hpp"  
#include "Panda.hpp"  
Voiture *Voiture::createVoiture(string origine) {  
    if (origine=="fr") return new Clio();  
    if (origine=="it") return new Panda();  
    return nullptr;  
}
```

(Avec Factory Method)

l'idée est différente : un critère de choix

MÉTHODE DE FABRICATION ("FACTORY METHOD")

Les types concrets n'apparaissent plus chez le client.

```
#include "Voiture.hpp"
int main() {
    Voiture *v = Voiture::createVoiture("fr");
    f(*v);
    return 0;
}
```

On peut aller plus loin et interdire au code client la possibilité de passer par `new Clio()` qui est ici encore autorisé

MÉTHODE DE FABRICATION ("FACTORY METHOD")

On interdit au code client de pouvoir passer par new :

```
class Voiture {  
public:  
    virtual string getModele() const = 0;  
    static Voiture *createVoiture(string origine);  
};
```

```
class Clio : public Voiture {  
private : Clio();  
public:  
    string getModele() const ;  
    friend Voiture* Voiture::createVoiture(string);  
};
```

```
class Panda : public Voiture {  
private : Panda();  
public:  
    string getModele() const ;  
    friend Voiture* Voiture::createVoiture(string);  
};
```


MÉTHODE DE FABRICATION ("FACTORY METHOD")

Remarques :

- le pattern illustré avec cet exemple pourrait moralement s'associer au pattern singleton (plus loin) car le fait que la méthode soit statique revient à ce qu'une factory soit unique.

MÉTHODE DE FABRICATION ("FACTORY METHOD")

Autre exemple pour illustrer l'usage de ce pattern :

les nombres complexes qui habituellement ont deux représentations : polaire ou cartésienne, de signature identique (deux doubles).

MÉTHODE DE FABRICATION ("FACTORY METHOD")

```
int main() {  
    Complex c = Complex::fromPolar(1, 3.14);  
    Complex autre_c = Complex::fromCartesian(-1, 0);  
}
```

Plutôt qu'un paramètre pour distinguer la nature des arguments, on peut introduire deux factory methods

MÉTHODE DE FABRICATION ("FACTORY METHOD")

```
class Complex {
public :
    static Complex fromCartesian(double real, double imag) {
        return Complex(real, imag);
    }
    static Complex fromPolar(double rho, double theta) {
        return Complex(rho * cos(theta), rho * sin(theta));
    }
private :
    double x,y;
    Complex(double a, double b):x{a},y{b} {}
};
```

```
int main() {
    Complex c = Complex::fromPolar(1, 3.14);
    Complex d = Complex::fromCartesian(-1, 0);
}
```

Dans cet exemple on a préféré n'avoir qu'une classe Complex (et pas Polar, Cartesian filles). On présente ainsi une solution différente de celle des voitures.

MÉTHODE DE FABRICATION ("FACTORY METHOD")

```
class Complex {  
    public :  
        static Complex fromCartesian(double real, double imag) {  
            return Complex(real, imag);  
        }  
        static Complex fromPolar(double rho, double theta) {  
            return Complex(rho * cos(theta), rho * sin(theta));  
        }  
    private :  
        double x, y;  
        Complex(double a, double b) : x{a}, y{b} {}  
};
```

On aurait pu n'utiliser qu'une factory méthode, avec un champs `char rep`,
ou introduire `rep` dans le constructeur,
mais cela obligerait à envisager les cas d'erreurs sur `rep`

MÉTHODE DE FABRICATION ("FACTORY METHOD")

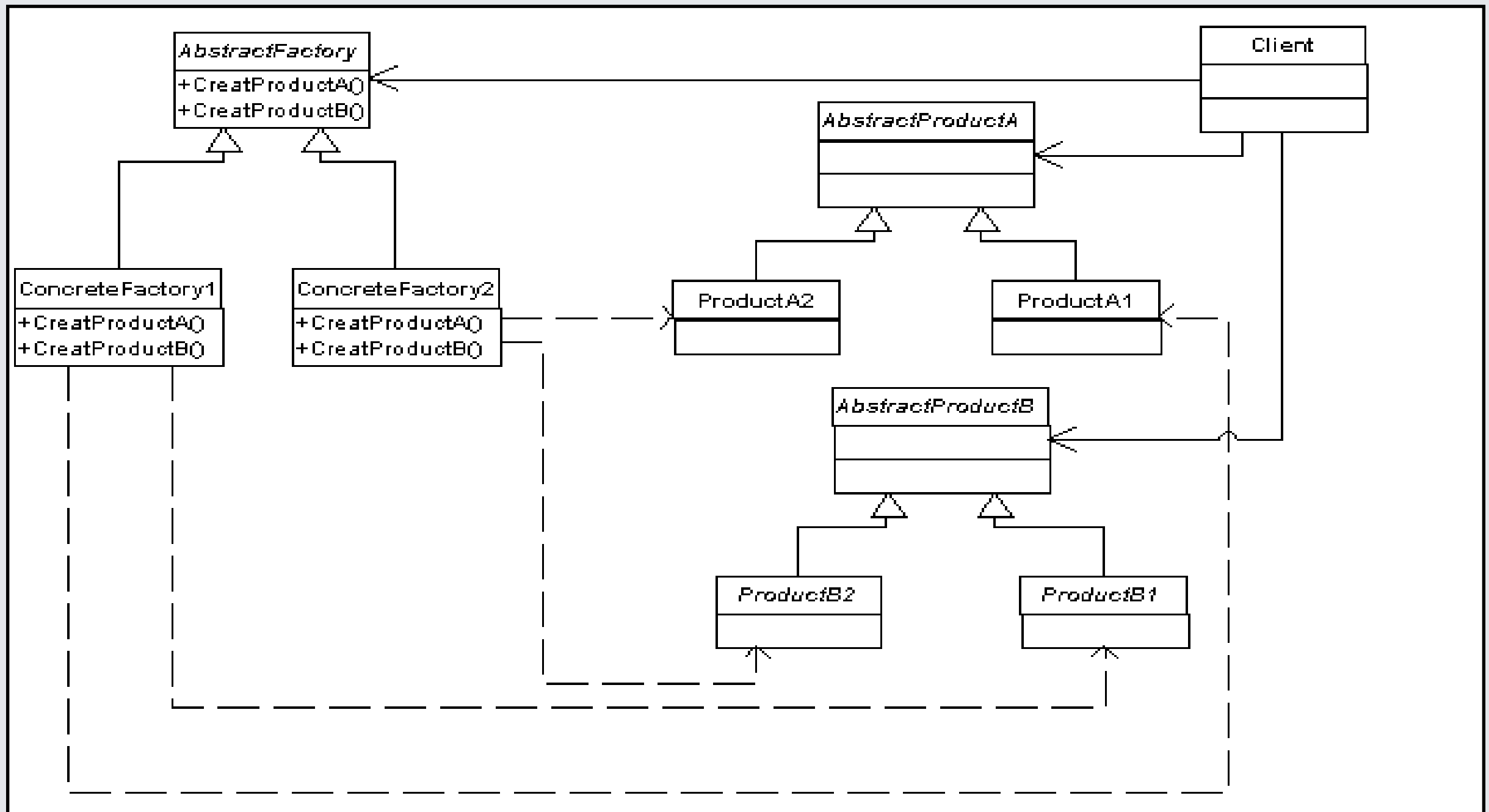
le pattern « méthode de fabrication » que l'on vient d'illustrer est en quelque sorte un problème simplifié de « la fabrique abstraite » que nous décrivons maintenant.

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

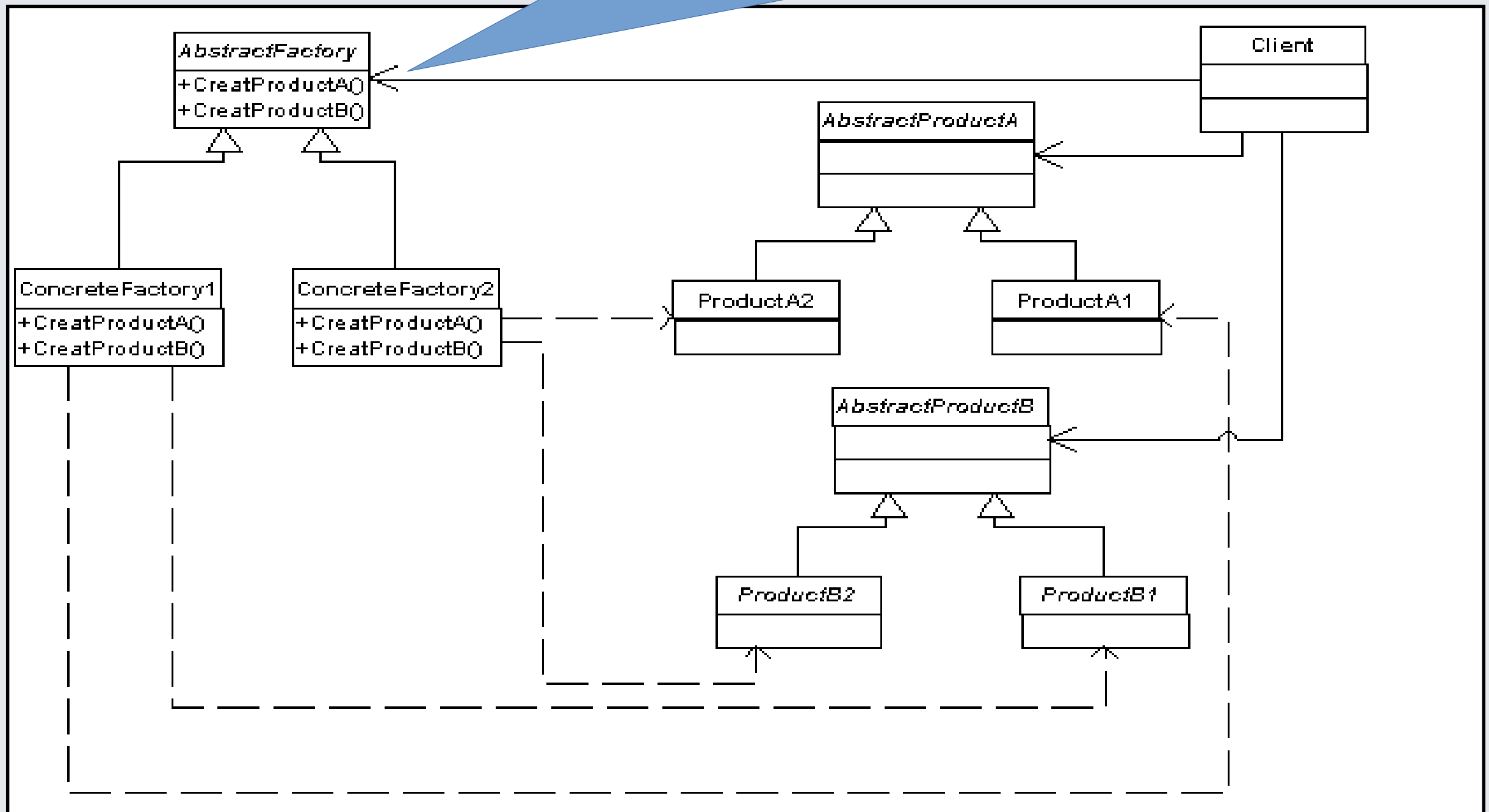
ABSTRACT FACTORY



ABSTRACT FACTORY

Un client est associé à (et connaît) :

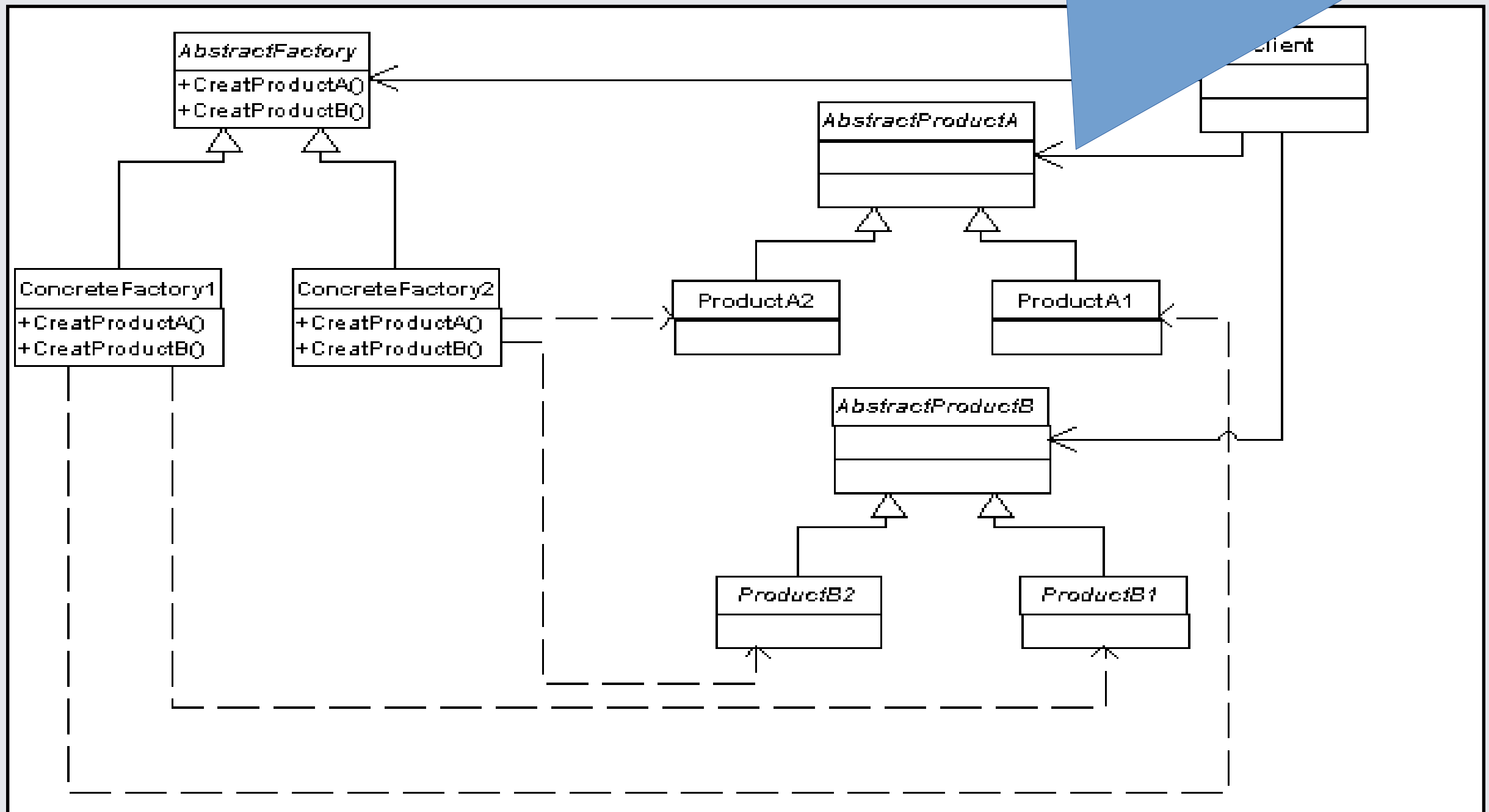
- Une/des abstractFactory
- Des ProduitsAbstraits A ou B



ABSTRACT FACTORY

Un client est associé à (et connaît) :

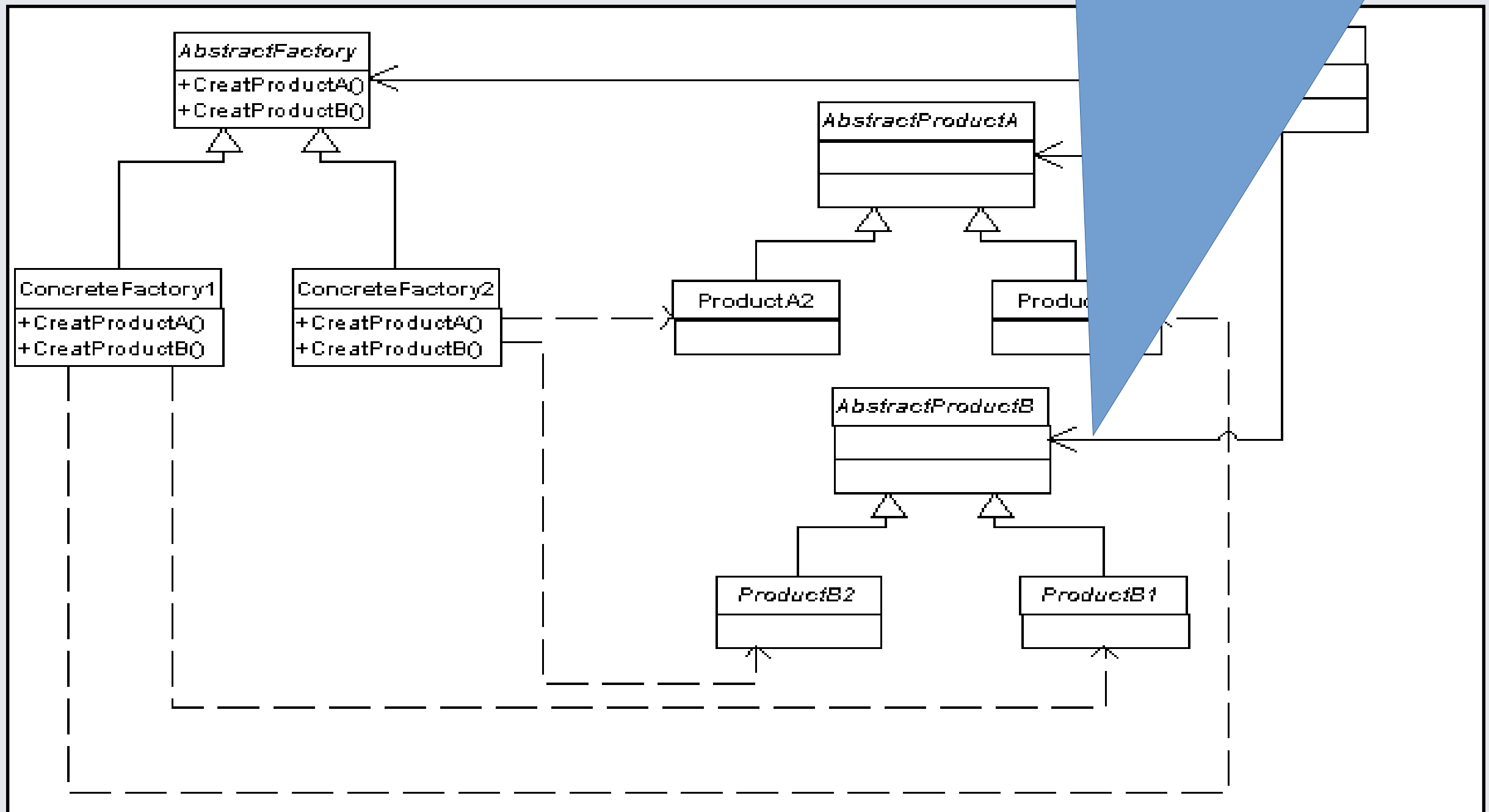
- Une/des abstractFactory
- Des ProduitsAbstraits A ou B



ABSTRACT FACTORY

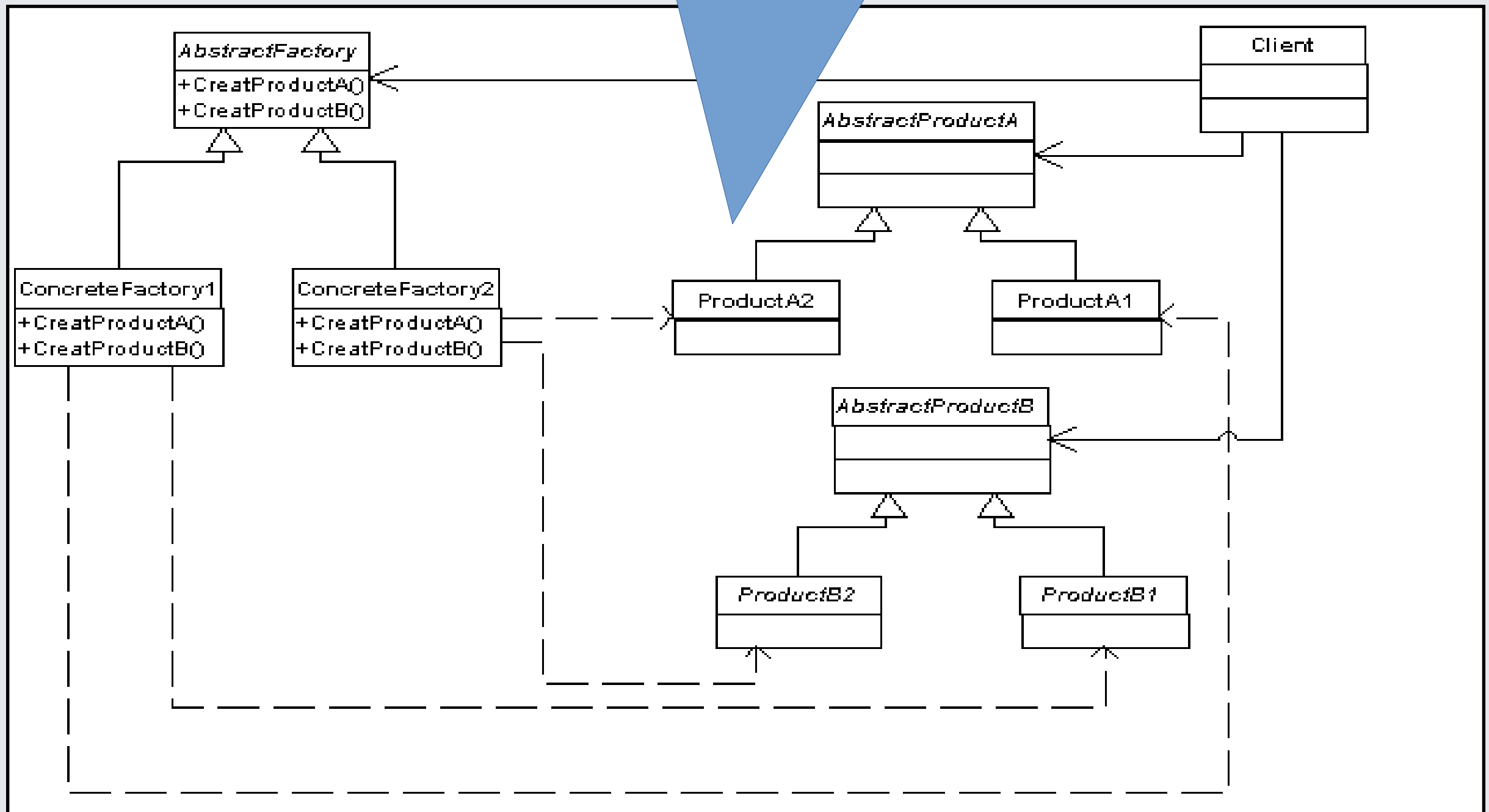
Un client est associé à (et connaît) :

- Une/des abstractFactory
- Des ProduitsAbstraits A ou B



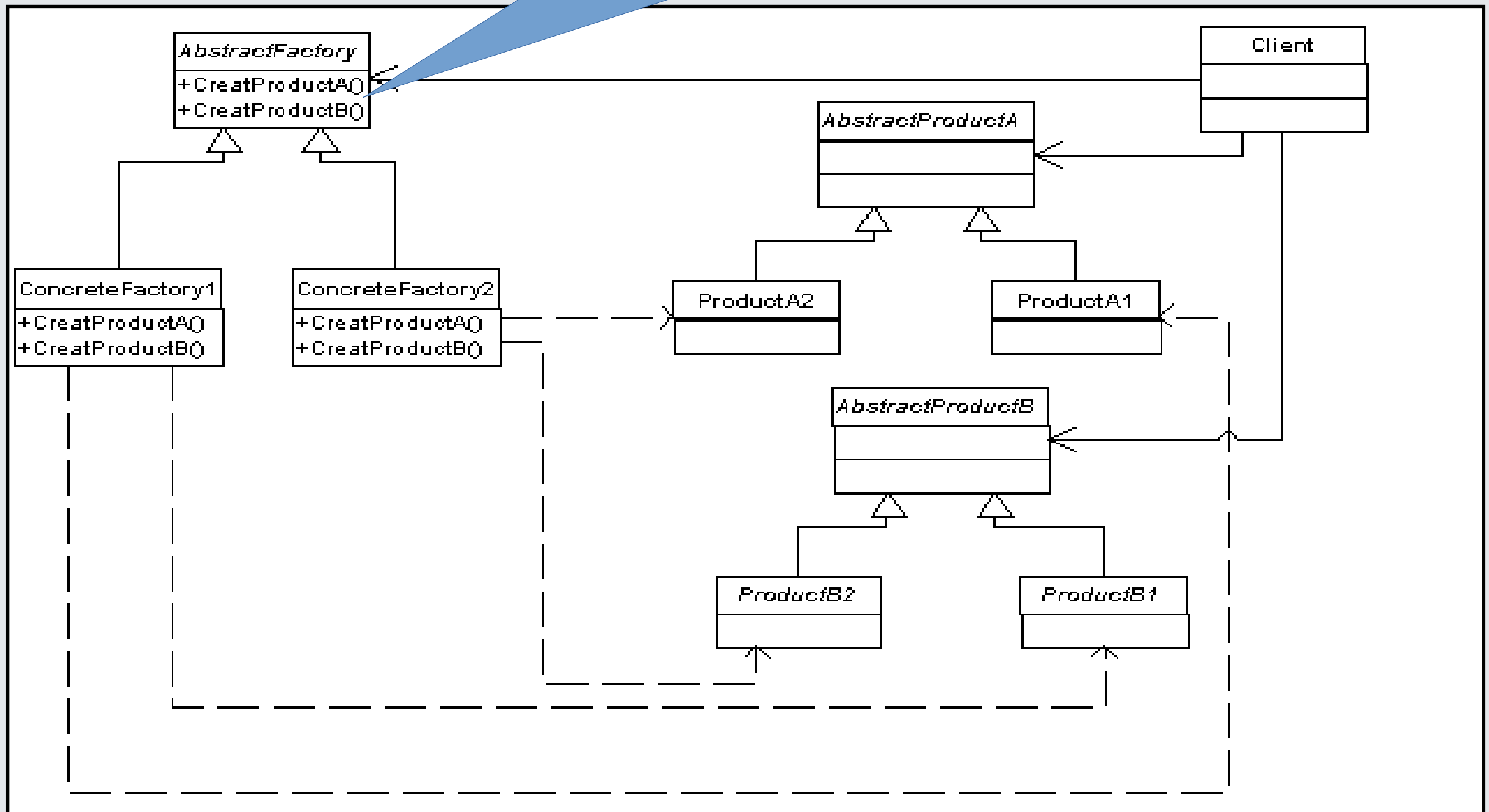
ABSTRACT FACTORY

Les ProduitsAbstrait ont des implémentations concrètes filles



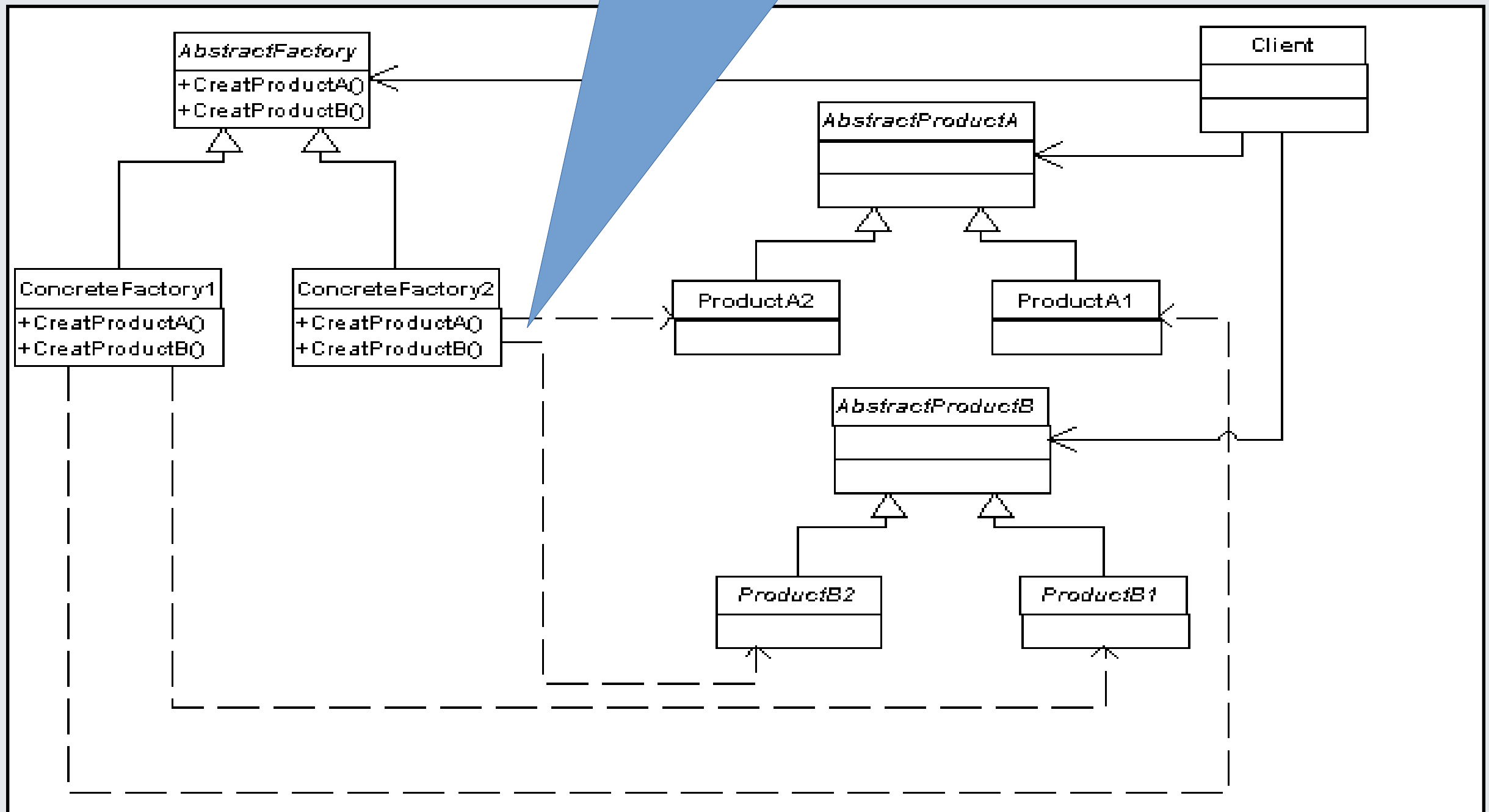
ABSTRACT FACTORY

Une abstract factory doit être capable de fournir des A ou des B



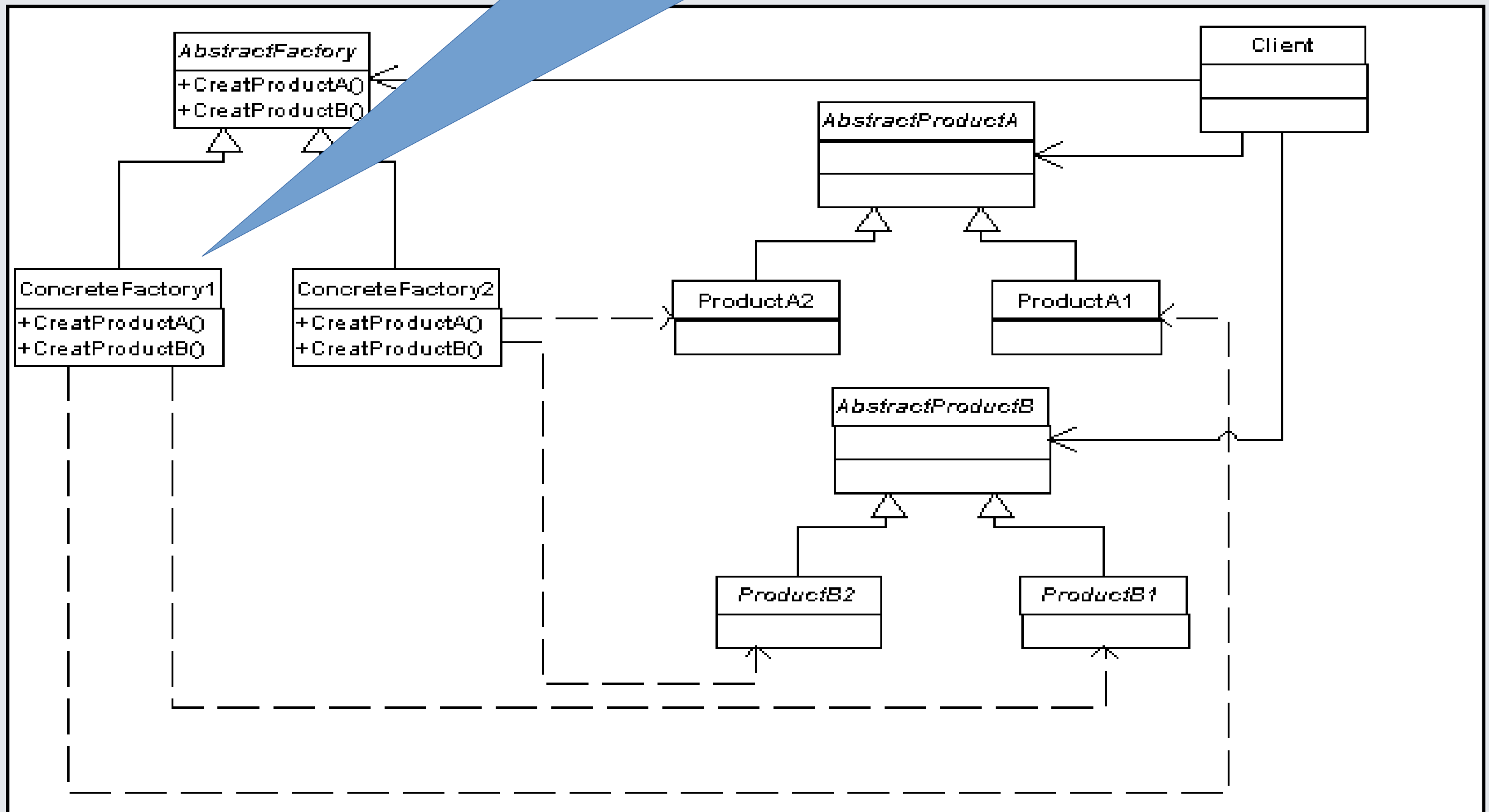
ABSTRACT FACTORY

Cette factory concrète utilise des A2 et des B2



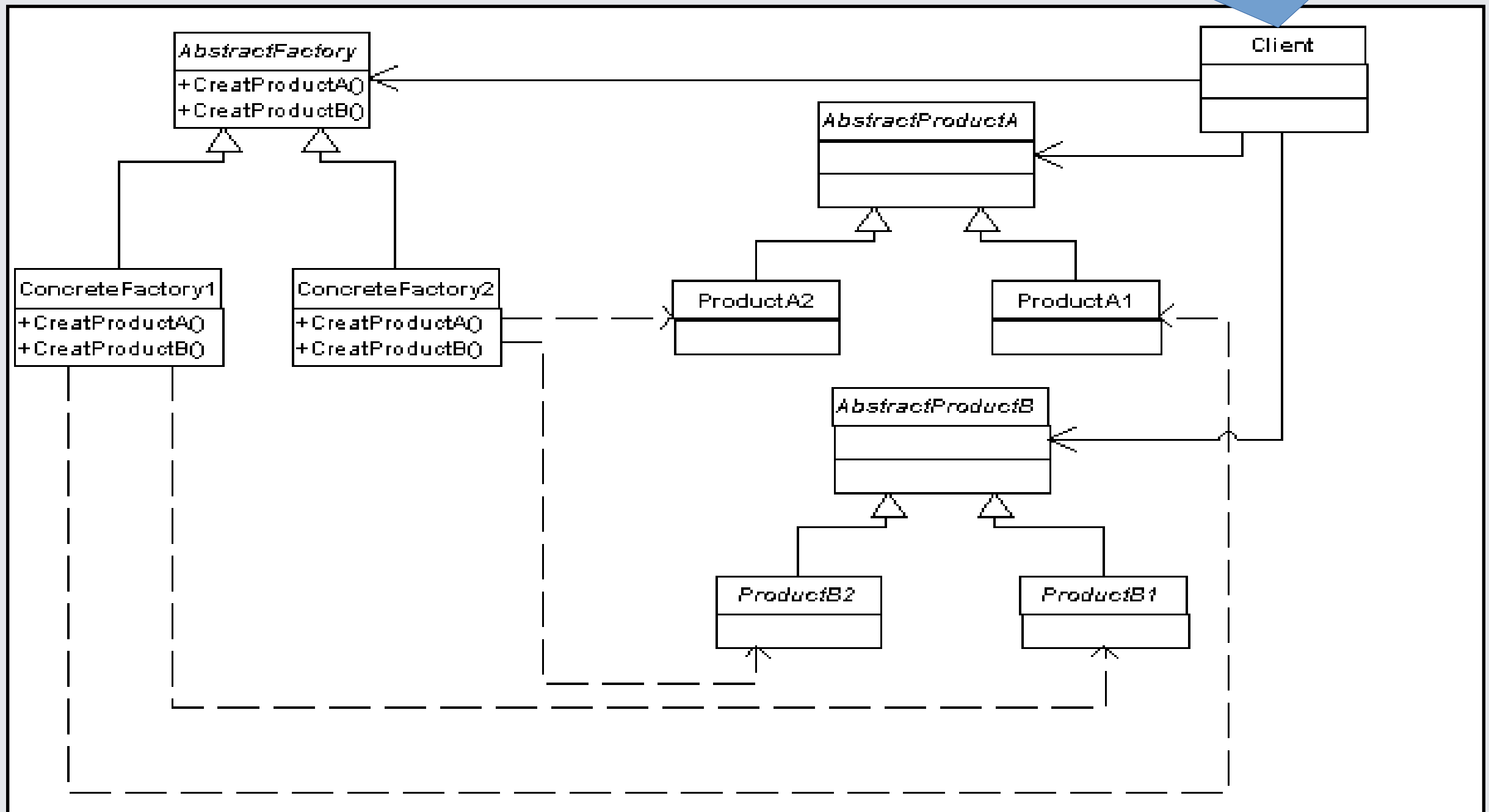
ABSTRACT FACTORY

Celle là utilise des A1 et des B1



ABSTRACT FACTORY

Du point de vue du client tout est abstrait : Factory ou Product



ABSTRACT FACTORY

Exemple :

```
class Voiture { // AbstractProduit A
public:
    virtual string getModele() const = 0;
};
```

```
class Clio : public Voiture {
public:
    string getModele() const ;
};
```

```
class Panda : public Voiture {
public:
    string getModele() const ;
};
```

ABSTRACT FACTORY

Exemple :

```
class Camionnette { // AbstractProduit B
public:
    virtual double getVolume() const = 0;
};
```

```
class Trafic : public Camionnette {
public:
    double getVolume() const { return 8.9; }
};
```

```
class Scudo : public Camionnette {
public:
    double getVolume() const { return 6.6; }
};
```

ABSTRACT FACTORY

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

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

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

ABSTRACT FACTORY

```
class Client {
private :
    AbstractFactory & f;
    Voiture *v;
    Camionnette *c;
public :
    Client (AbstractFactory &x) : f{x} , v{f.createVoiture()}
                                , c{f.createCamionnette()} {}
}
int main() {
    FabriqueFrancaise fr;
    FabriqueItalienne it;
    Client c1{fr};
    Client c2{it}
}
```

Du côté du client les manipulations sont abstraites.
C'était l'objectif recherché.

ABSTRACT FACTORY

Remarquez qu'on a les motifs :

- AbstractProductA avec ProductA1 et ProductA2
- idem pour AbstractProductB
- idem pour AbstractFactory

Et on a vu juste auparavant que c'était exactement dans ce cadre qu'on pouvait envisager method factory (trois fois)...

Faut-il introduire une méthode statique dans les classes mères pour construire les classes concrètes ... ?

Comprenez que ces patterns sont un peu modulables... pas forcément systématiques.

Dans le Bourgeois-Gentilhomme, Mr Jourdain, cède au "mamamouchisme"
(action de s'élever à de prétendues dignités)

Autre exemple : pour le MVC on tolère assez bien que le rôle du "contrôleur" se dilue un peu dans le modèle ou dans la vue.

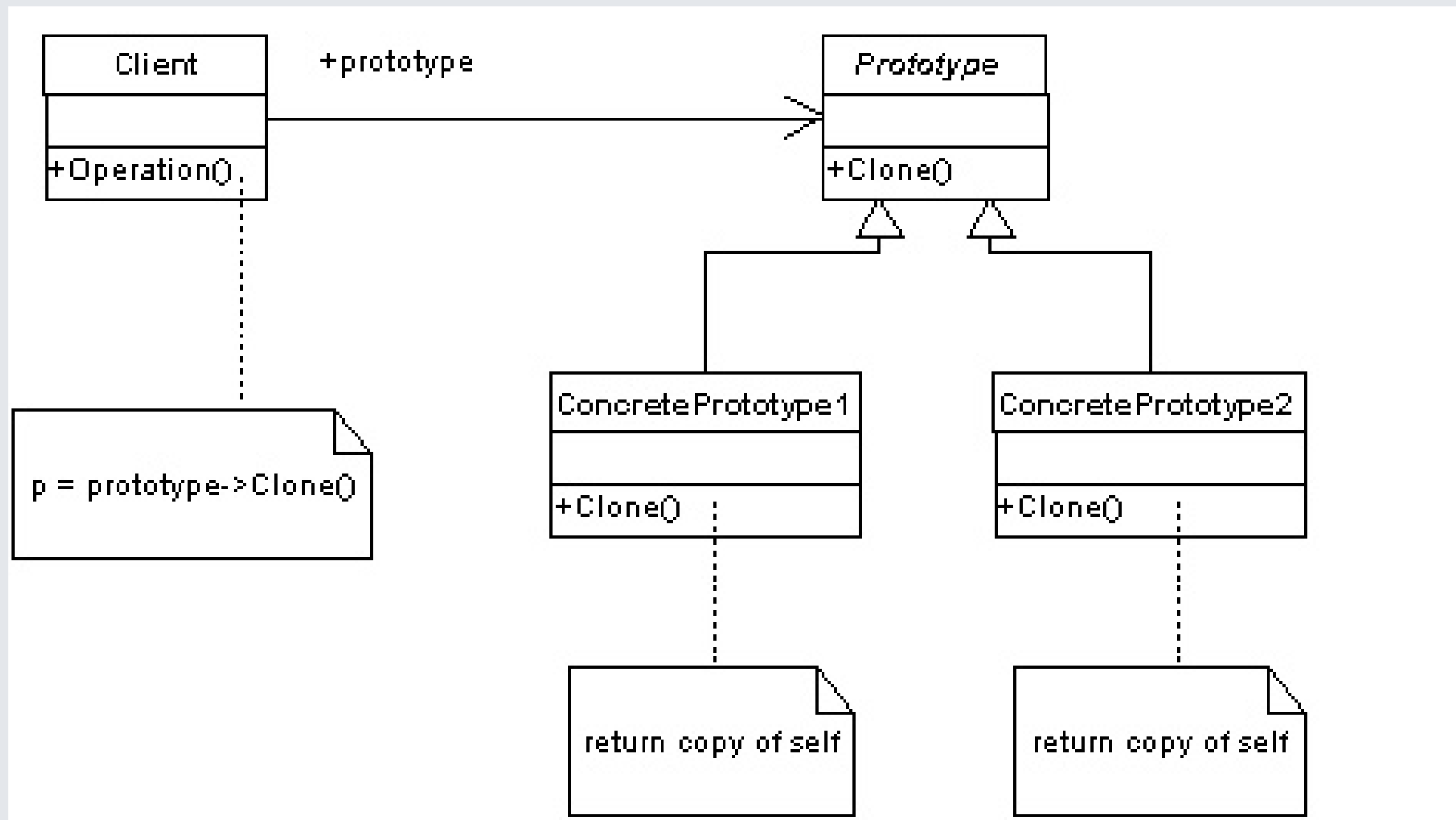
Passons au pattern PROTOTYPE

Il est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer (cloner) un objet existant puis de modifier la copie ensuite.

La méthode de construction est habituellement appelée `clone()`

PROTOTYPE

Dans le contexte où un client est associé à un (ou des) objets abstraits implémentant le pattern prototype



PROTOTYPE

Remarque :

Le constructeur de copie n'est pas tout à fait un pattern "prototype" car il n'est pas virtual : avec lui le type exact est connu à l'appel et attendu en retour.

On aimerait ici que la copie se fasse au plus près.

PROTOTYPE

Rappel : ce pattern est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer un objet existant et de modifier la copie ensuite.

PROTOTYPE

Rappel : ce pattern est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer un objet existant et de modifier la copie ensuite.

Exemple 1 :

Une grille de sudoku est une sorte de "carré magique". Les contraintes associées à sa construction sont fortes.

8	1	3	9	2	5	7	4	6
9	5	6	8	4	7	3	1	2
4	7	2	3	6	1	8	9	5
6	2	4	7	1	9	5	3	8
7	9	5	6	3	8	4	2	1
3	8	1	4	5	2	9	6	7
2	3	8	1	7	4	6	5	9
5	4	9	2	8	6	1	7	3
1	6	7	5	9	3	2	8	4

PROTOTYPE

Rappel : ce pattern est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer un objet existant et de modifier la copie ensuite.

Exemple 1 :

Pour proposer un nouveau jeu, (ayant une solution) on peut partir d'une grille complète et :

- opérer des symétries
- des rotations
- des permutation ex : $7 \leftrightarrow 9$
- des effacements

8	1	3	9	2	5	7	4	6
9	5	6	8	4	7	3	1	2
4	7	2	3	6	1	8	9	5
6	2	4	7	1	9	5	3	8
7	9	5	6	3	8	4	2	1
3	8	1	4	5	2	9	6	7
2	3	8	1	7	4	6	5	9
5	4	9	2	8	6	1	7	3
1	6	7	5	9	3	2	8	4

PROTOTYPE

Rappel : ce pattern est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer un objet existant et de modifier la copie ensuite.

Exemple 1 :

Pour proposer un nouveau jeu, (ayant une solution) on peut partir d'une grille complète et :

- opérer des symétries
- des rotations
- des permutation ex : $7 \leftrightarrow 9$
- des effacements

8	1	3	9	2	5	7	4	6
9	5	6	8	4	7	3	1	2
4	7	2	3	6	1	8	9	5
6	2	4	7	1	9	5	3	8
7	9	5	6	3	8	4	2	1
3	8	1	4	5	2	9	6	7
2	3	8	1	7	4	6	5	9
5	4	9	2	8	6	1	7	3
1	6	7	5	9	3	2	8	4

Intérêt : zéro tests !

PROTOTYPE

Rappel : ce pattern est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer un objet existant et de modifier la copie ensuite.

Exemple 2 :

Un dictionnaire n'est pas qu'un ensemble de mot. Il est adossé à des outils de recherche (hashtable par ex) construits au fur et à mesure, et stockés. Reconstruire un dictionnaire est donc coûteux !

PROTOTYPE

Rappel : ce pattern est utilisé lorsque le coût de fabrication d'un objet est lourd et qu'il est plus facile de dupliquer un objet existant et de modifier la copie ensuite.

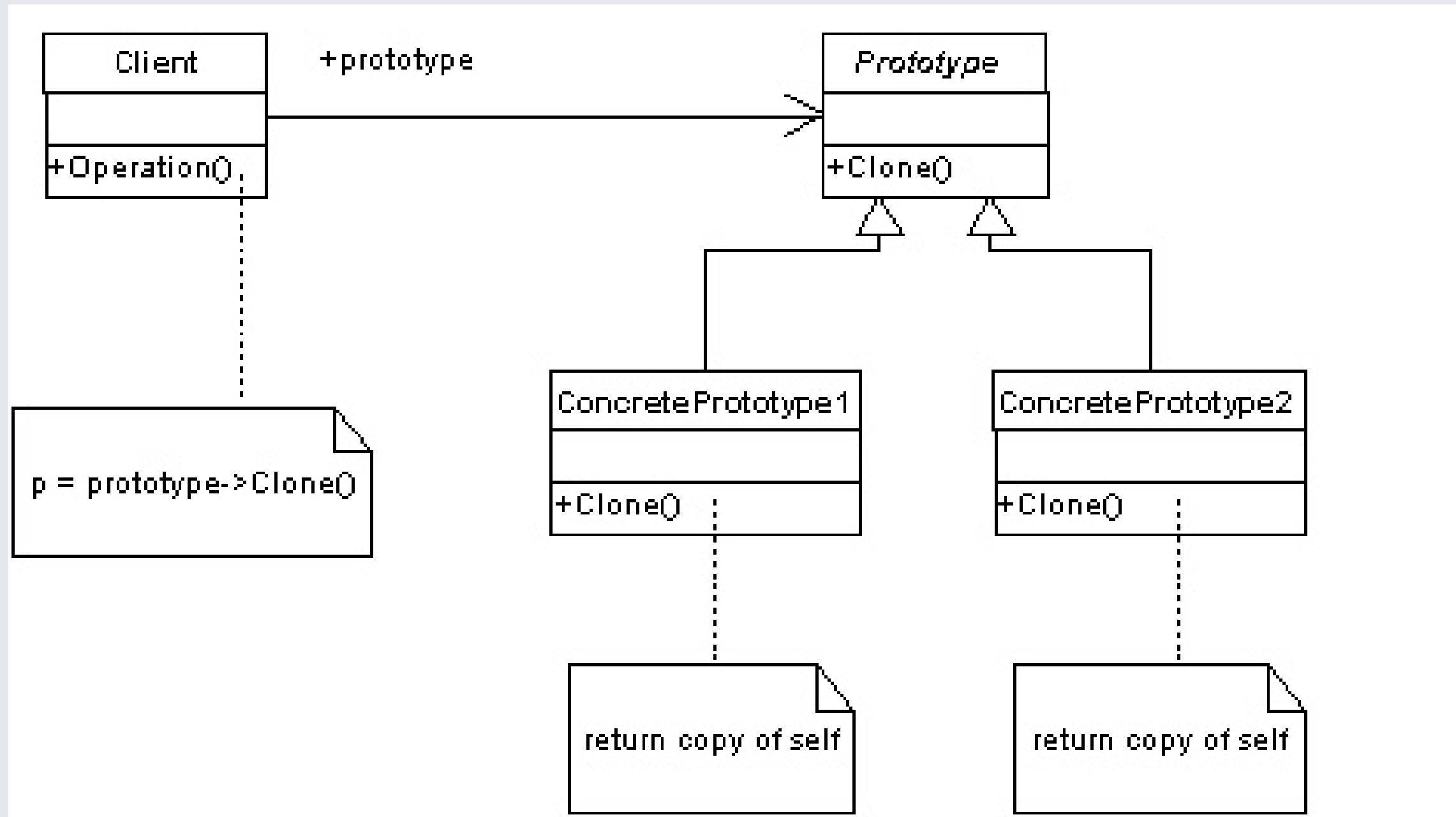
Exemple 2 :

Un dictionnaire n'est pas qu'un ensemble de mot. Il est adossé à des outils de recherche (hashtable par ex) construite au fur et à mesure, et stockée. Reconstruire un dictionnaire est donc coûteux !

On peut penser à deux types concrets :

DicoHashSet, DicoTreeSet

PROTOTYPE



PROTOTYPE

```
class Dico { // AbstractPrototype
public:
    virtual Dico* clone() const = 0;
    virtual void add(string) = 0;
    virtual void remove(string) = 0;
};
```

```
class DicoHash : public Dico {
private:
    unordered_set <string> words;
public:
    void add(string);
    void remove(string);
    DicoHash* clone() const;
};
```

```
DicoHash* DicoHash::clone() const {return new DicoHash(*this); }
void DicoHash:: add(string x){ words.insert(x); }
void DicoHash:: remove(string x){ words.erase(x); }
```


PROTOTYPE

```
class Dico { // AbstractPrototype
public:
    virtual Dico* clone() const = 0;
    virtual void add(string) = 0;
    virtual void remove(string) = 0;
};
```

On raffine
le type retour

```
class DicoHash : public Dico {
private:
    unordered_set <string> words;
public:
    void add(string);
    void remove(string);
    DicoHash* clone() const;
};
```

```
DicoHash* DicoHash::clone() const {return new DicoHash(*this); }
void DicoHash:: add(string x){ words.insert(x); }
void DicoHash:: remove(string x){ words.erase(x); }
```

PROTOTYPE

```
class Dico { // AbstractPrototype
public:
    virtual Dico* clone() const = 0;
    virtual void add(string) = 0;
    virtual void remove(string) = 0;
};
```

On raffine
le type retour

```
class DicoHash : public Dico {
private:
    unordered_set <string> words;
public:
    void add(string);
    void remove(string);
    DicoHash* clone() const;
};
```

Le constructeur de
copie par défaut
fait l'affaire

```
DicoHash* DicoHash::clone() const {return new DicoHash(*this); }
void DicoHash:: add(string x){ words.insert(x); }
void DicoHash:: remove(string x){ words.erase(x); }
```

PROTOTYPE

```
class Dico { // AbstractPrototype
public:
    virtual Dico* clone() const = 0;
    virtual void add(string) = 0;
    virtual void remove(string) = 0;
};
```

```
class DicoTree : public Dico {
private:
    set<string> words;
public:
    void add(string);
    void remove(string);
    DicoTree* clone() const;
};
```

```
DicoTree* DicoTree::clone() const {return new DicoTree(*this);}
void DicoTree:: add(string x){ words.insert(x); }
void DicoTree:: remove(string x){ words.erase(x); }
```

PROTOTYPE

Exemple :

```
int main() {  
    Dico &d1 {Dico::Larousse;}  
    Dico &d2 {Dico::Robert;}  
    Dico *myWoke {d1.clone();}  
    Dico *yourWoke {d2.clone();}  
    myWoke -> remove("auteur");  
    yourWoke -> remove("auteur");  
    myWoke.add("auteur.trice");  
    yourWoke -> add("auteur.e");  
}
```

On suppose disposer
de ces variables
statiques

PROTOTYPE

Exemple :

```
int main() {  
    Dico &d1 {Dico::Larousse;}  
    Dico &d2 {Dico::Robert;}  
    Dico *myWoke {d1.clone();}  
    Dico *yourWoke {d2.clone();}  
    myWoke -> remove("auteur");  
    yourWoke -> remove("auteur");  
    myWoke.add("auteur.trice");  
    yourWoke -> add("auteur.e");  
}
```

deux clonages
pas cher

PROTOTYPE

Exemple :

```
int main() {  
    Dico &d1 {Dico::Larousse;}  
    Dico &d2 {Dico::Robert;}  
    Dico *myWoke {d1.clone();}  
    Dico *yourWoke {d2.clone();}  
    myWoke -> remove("auteur");  
    yourWoke -> remove ("auteur");  
    myWoke.add("auteur.trice");  
    yourWoke -> add("auteur.e");  
}
```

deux clonages
pas cher

des modifications
des prototypes

PROTOTYPE

Exemple :

```
int main() {  
    Dico &d1 {Dico::Larousse;}  
    Dico &d2 {Dico::Robert;}  
    Dico *myWoke {d1.clone();}  
    Dico *yourWoke {d2.clone();}  
    myWoke -> remove("auteur");  
    yourWoke -> remove ("auteur");  
    myWoke.add("auteur.trice");  
    yourWoke -> add("auteur.e");  
}
```

Rq : ici on pourra reconnaître un pattern flyweight

deux clonages
pas cher

des modifications
des prototypes

PROTOTYPE

Exemple :

```
int main() {  
    Dico &d1 {Dico::Larousse;}  
    Dico &d2 {Dico::Robert;}  
    Dico *myWoke {d1.clone();}  
    Dico *yourWoke {d2.clone();}  
    myWoke -> remove("auteur");  
    yourWoke -> remove ("auteur");  
    myWoke.add("auteur.trice");  
    yourWoke -> add("auteur.e");  
}
```

Rq : ici on pourra reconnaître un pattern flyweight

deux clonages
pas cher

des modifications
des prototypes

autre exemple : réseau de neurone (apprentissage long) etc...

Passons au pattern **BUILDER**

Il est utilisé pour décrire la construction incrémentale d'un objet.

Le monteur (un objet responsable de la construction) implémente des méthodes de réalisation partielles.

Il se charge, à la fin, de la synthèse, en assemblant les composants précédemment réalisés.

Ce pattern est évoqué dans le TP noté 2023 (classe Proto-Run)

BUILDER

Utilisé pour une construction incrémentale.

Exemple :

Dans un fastfood, un client fait le choix d'un type de menu (enfant ou adulte), et le soumet à un caissier.

Celui ci lance la construction des produits associés, avant de produire le repas attendu, via le builder du menu.

```
int main() { //client
    Caissier c;
    HappyMealBuilder b1;
    AdultMealBuilder b2;
    c.createMeal(b1);
    c.createMeal(b2);
}
```

BUILDER

Utilisé pour une construction incrémentale.

Exemple :

Dans un fastfood, un client fait le choix d'un type de menu (enfant ou adulte), et le soumet à un caissier.

Celui ci lance la construction des produits associés, avant de produire le repas. attend le build du builder du menu

```
int main() { //client
    Caissier c;
    HappyMealBuilder b1;
    AdultMealBuilder b2;
    c.createMeal(b1);
    c.createMeal(b2);
}
```

```
class Caissier {
public :
    Meal* createMeal(MealBuilder &b) {
        b.buildBoisson();
        b.buildPlat();
        b.buildDessert();
        return b.build();
    }
};
```

```

class Builder { // abstrait
protected :
    Boisson *_boisson;
    Plat *_plat;
    Dessert *_dessert;
private :
    virtual void buildBoisson()=0;
    virtual void buildPlat()=0;
    virtual void buildDessert()=0;
    Meal* build() {
        if (consistant())
            return new Meal(_boisson,_plat,_dessert);
        else ...
    }
    friend class Caissier ;
};

```

```

int main() { //client
    Caissier c;
    HappyMealBuilder b1;
    AdultMealBuilder b2;
    c.createMeal(b1);
    c.createMeal(b2);
}

```

```

class Caissier {
public :
    Meal* createMeal(MealBuilder &b) {
        b.buildBoisson();
        b.buildPlat();
        b.buildDessert();
        return b.build();
    }
};

```

struction

e de menu

iés,

```
class AdultMealBuilder : public Builder {
public :
    void buildBoisson() {_boisson=new Beer();}
    ... etc ...
};
```

```
private :
    virtual void buildBoisson()=0;
    virtual void buildPlat()=0;
    virtual void buildDessert()=0;
    Meal* build() {
        if (consistant())
            return new Meal(_boisson,_plat,_dessert);
        else ...
    }
    friend class Caissier ;
};
```

```
int main() { //client
    Caissier c;
    HappyMealBuilder b1;
    AdultMealBuilder b2;
    c.createMeal(b1);
    c.createMeal(b2);
}
```

```
class Caissier {
public :
    Meal* createMeal(MealBuilder &b) {
        b.buildBoisson();
        b.buildPlat();
        b.buildDessert();
        return b.build();
    }
};
```

```

class AdultMealBuilder : public Builder {
public :
    void buildBoisson() {_boisson=new Beer();}
    ... etc ...
};

```

```

private :
    virtual void buildBoisson()=0;
    virtual void buildPlat()=0;
    virtual void buildDessert()=0;
    Meal* build() {
        if (consistant())
            return new Meal( _boisson, _plat, _dessert);
        else
            return 0;
    }
friend class Caissier;
};

```

```

int main()
{
    Caissier c;
    HappyMealBuilder b1;
    AdultMealBuilder b2;

    c.createMeal(b1);
    c.createMeal(b2);
}

```

```

class Meal {
public :
    Boisson * const b;
    Plat * const p;
    Dessert * const d;
private :
    Meal(Boisson, Plat, Dessert);
friend Meal* Builder::build();
};

```

```

    return b.build();
}
};

```

ction

e de menu

&b) {

SINGLETON

Le pattern singleton permet de s'assurer qu'une classe est un ensemble réduit à un seul élément

(rencontré au tp noté 2022)

SINGLETON

```
class Single {  
private:  
    static Single theUnique;  
    Single() {...} //avec private on bloque les créations  
    Single(Single &l) = delete // et les copies  
    ~Single() {...} // et la destruction  
public:  
    static Single *getIt() { return &theUnique; }  
};
```

s'il est unique et déterminé dès le départ, il est assez normal que le constructeur soit sans argument.

SINGLETON

```
class Single {
private:
    static Single * theUnique;
    Single(int param) {...}
    Single(Single &l) = delete // et les copies
    ~Single() {...}           // et la destruction
public:
    static Single *getIt() {
        if (theUnique==nullptr) theUnique=new Single(rand()) ;
        return theUnique;
    }
};
Single * Single::theUnique{nullptr};
```

Si vous tolérez qu'il soit indéterminé tant qu'il n'a pas été utile...

Ce n'est pas tout à fait/strictement un singleton, mais une bonne introduction à flyweight

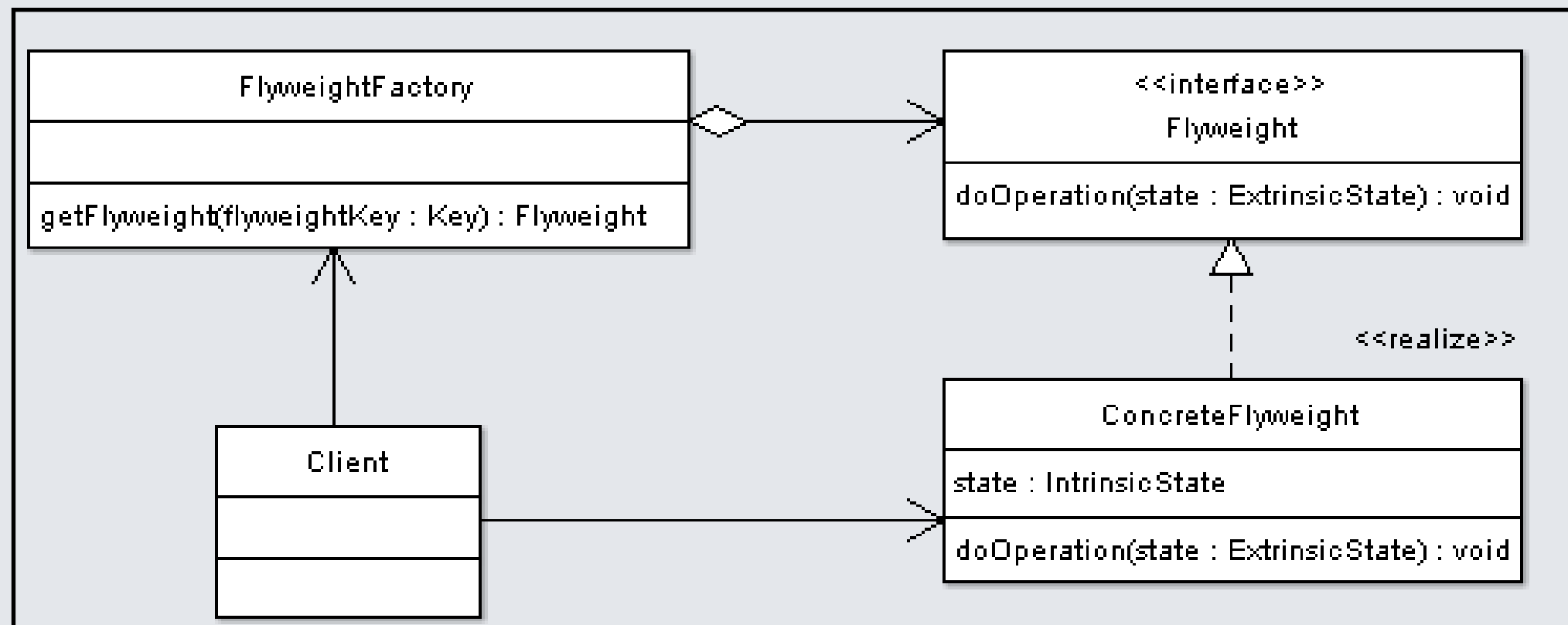
PATTERNS DE STRUCTURE

FLYWEIGHT

Ce pattern poids-mouche est chargé d'éviter la création de trop nombreuses instances identiques.

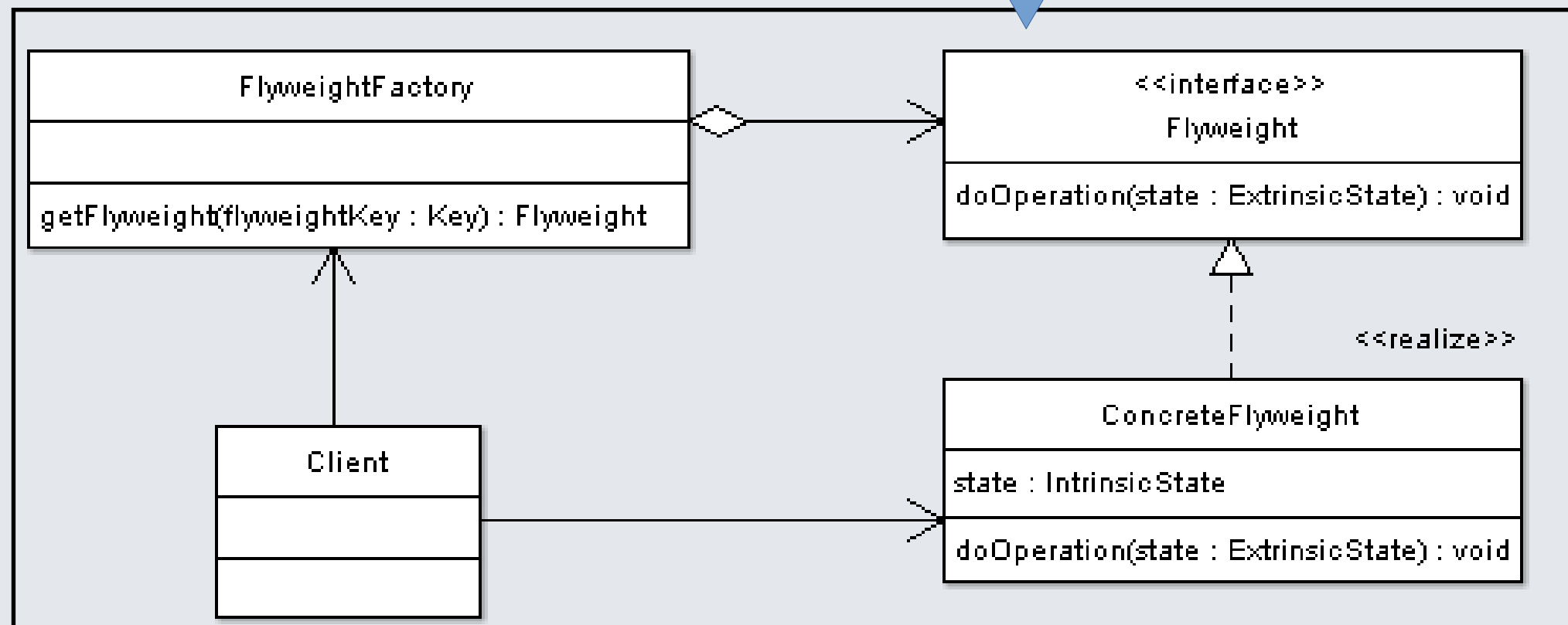
L'idée est d'externaliser certaines données, en utilisant une technique proche de ce qui est fait avec Singleton.

FLYWEIGHT



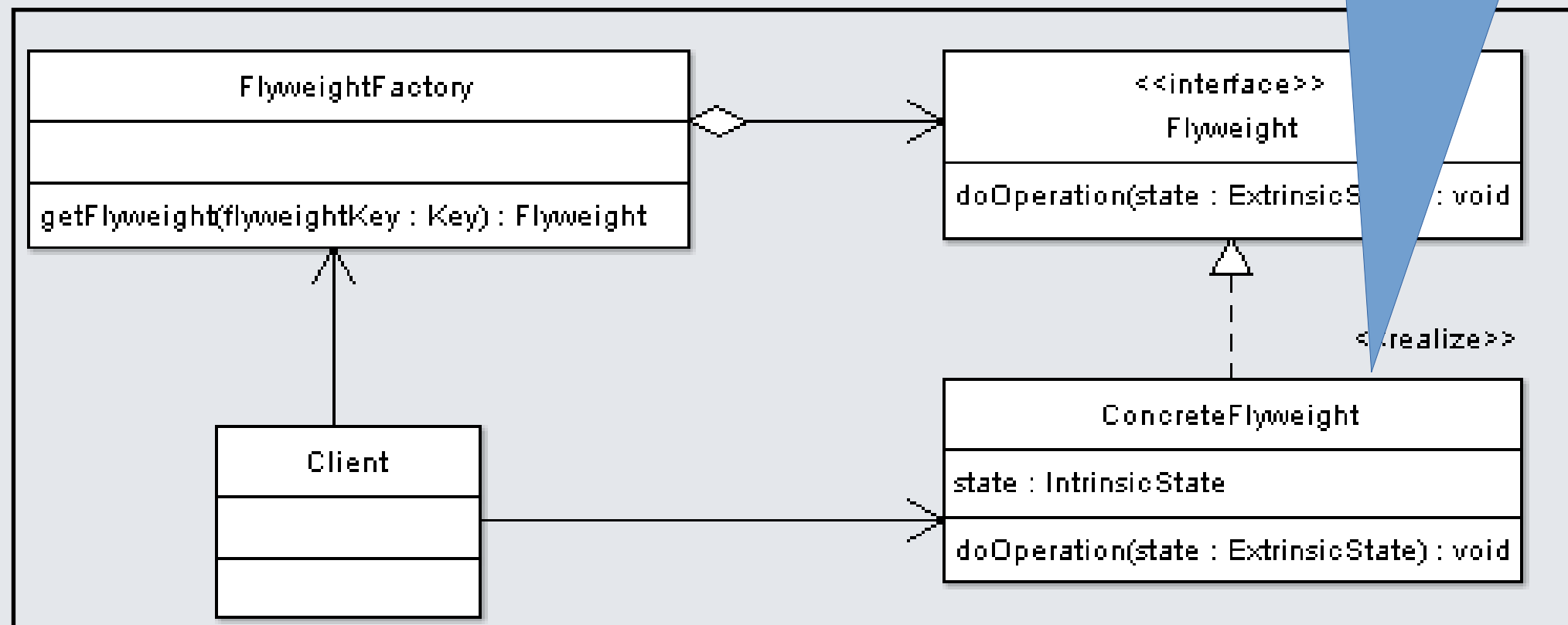
FLYWEIGHT

image abstraite d'un pion
dans un jeu de dames,
avec une méthode de dessin



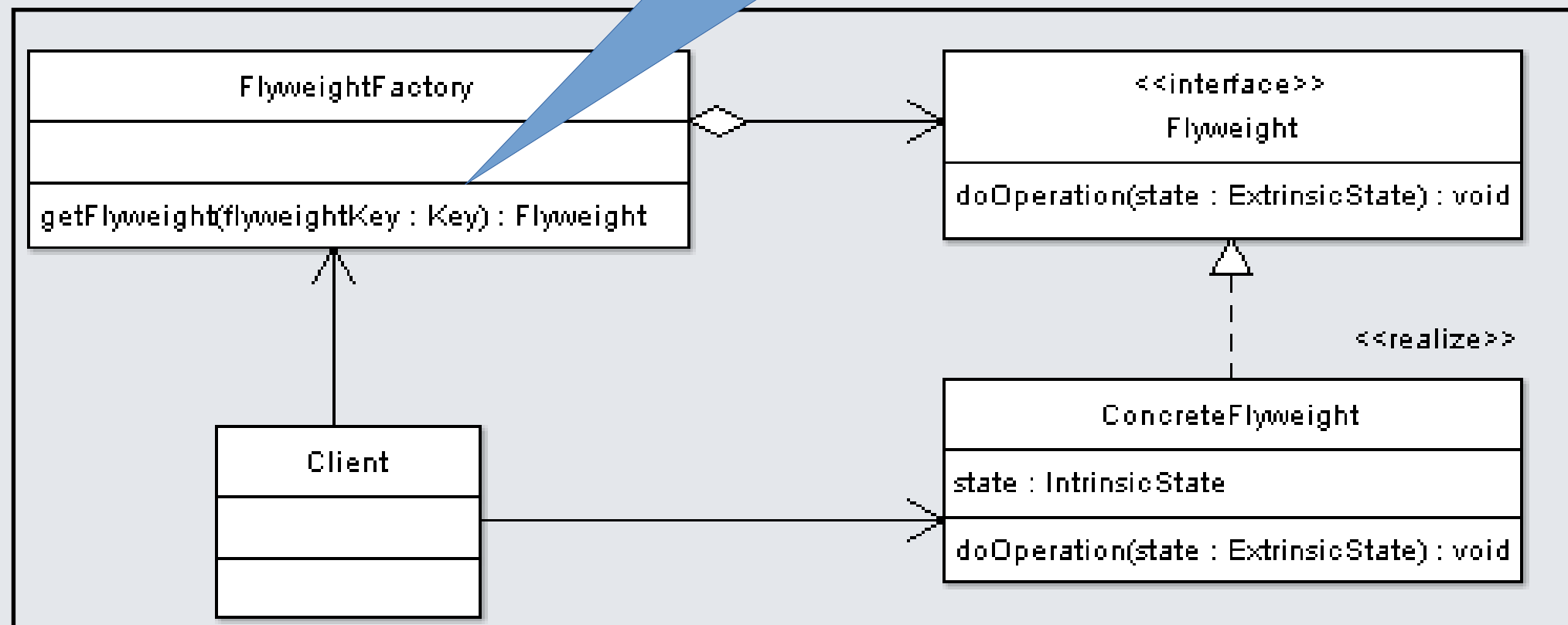
FLYWEIGHT

image concrète d'un pion
(gif, jpeg ...)



FLYWEIGHT

récupération des images
pour des pions blancs/noirs



FLYWEIGHT

La factory agrège des unités de poids mouche

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

```
ImagePion *ImagePionFactory::blanc=nullptr;
ImagePion *ImagePionFactory::noir =nullptr;
```


FLYWEIGHT

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

initialisées à nullptr

```
ImagePion *ImagePionFactory::blanc=nullptr;  
ImagePion *ImagePionFactory::noir =nullptr;
```

FLYWEIGHT

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

construites lorsque nécessaires

```
ImagePion *ImagePionFactory::blanc=nullptr;  
ImagePion *ImagePionFactory::noir =nullptr;
```

FLYWEIGHT

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

```
class _ImagePion : public ImagePion { //Flyweight Concret
private:
    // beaucoup de données internes, lourd ...
public:
    void drawAt(int x,int y) {
        // travail sur les datas ...
    }
};
```

FLYWEIGHT

```
class Pion { // un client
protected:
    int x, y;
    string couleur;
public:
    Pion(string c, int x,int y) : couleur{c}, x{x}, y{y} {}
    void draw() {
        ImagePion *ip = ImagePionFactory::getImagePion("blanc");
        ip->drawAt(x,y);
    }
};
```

FLYWEIGHT

```
int main() { // ailleurs
    Pion *p[6] = { new Pion("Blanc",1,1),
                  new Pion("Noir",1,2),
                  new Pion("Blanc",1,3),
                  new Pion("Noir",1,2),
                  new Pion("Blanc",3,1),
                  new Pion("Blanc",4,2)
                };
    for (int i=0; i<6; i++) p[i]->draw();
}
```

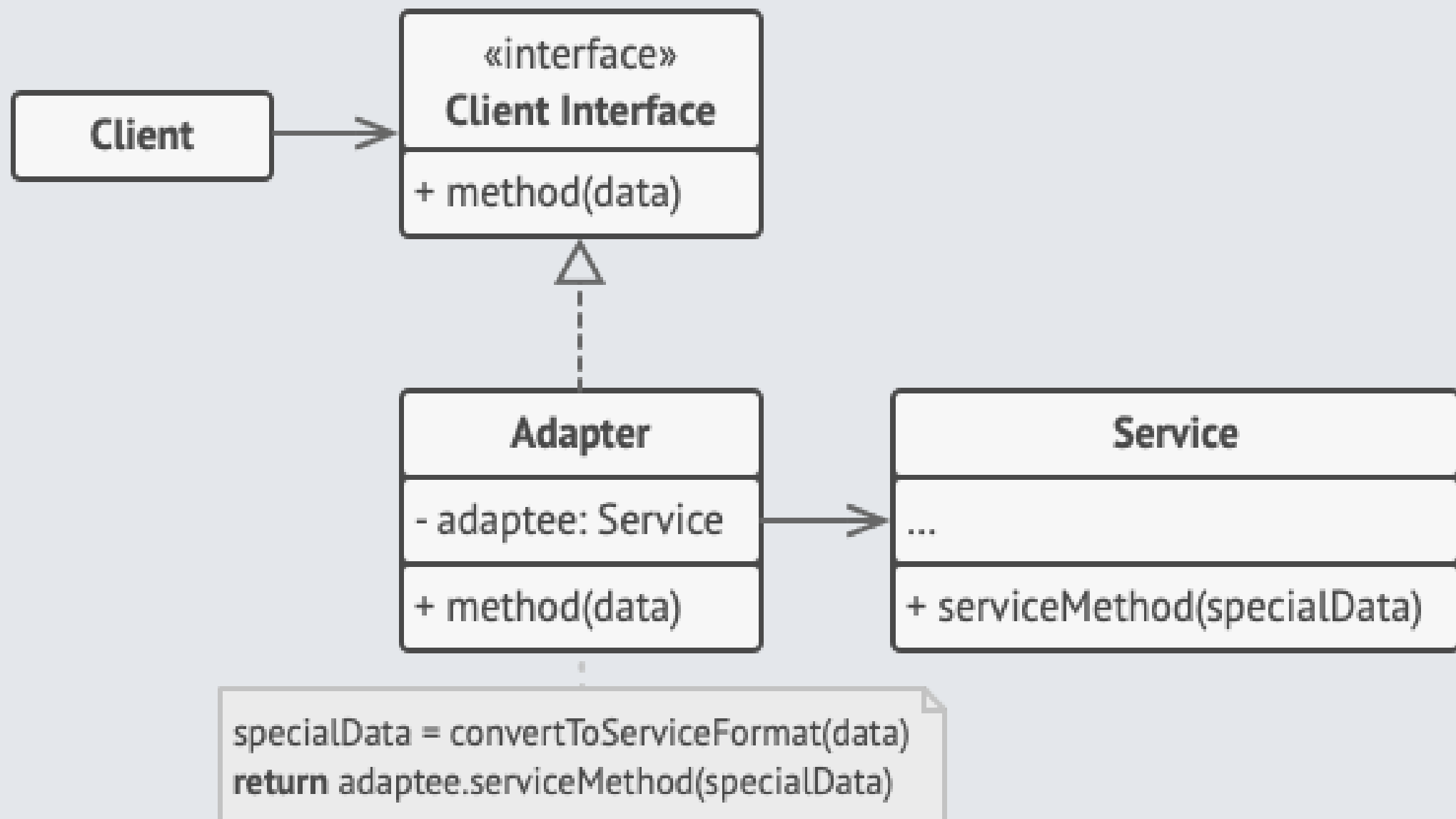
**n pions mais
seulement 2 images**

"Ce pattern était chargé d'éviter la création de trop nombreuses instances identiques.

L'idée était d'externaliser certaines données, en utilisant une technique proche de ce qui est fait avec Singleton."

ADAPTER

Le pattern adaptateur/adapter permet de convertir l'interface d'un objet en une autre interface.



Exemple : on peut utiliser un Four en tant que Chauffage, modulo l'écriture d'un Adaptateur

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

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

Exemple : on peut utiliser un Four en tant que Chauffage, modulo l'écriture d'un Adaptateur

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

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

c'est une réinterprétation rapide acceptable du diagramme

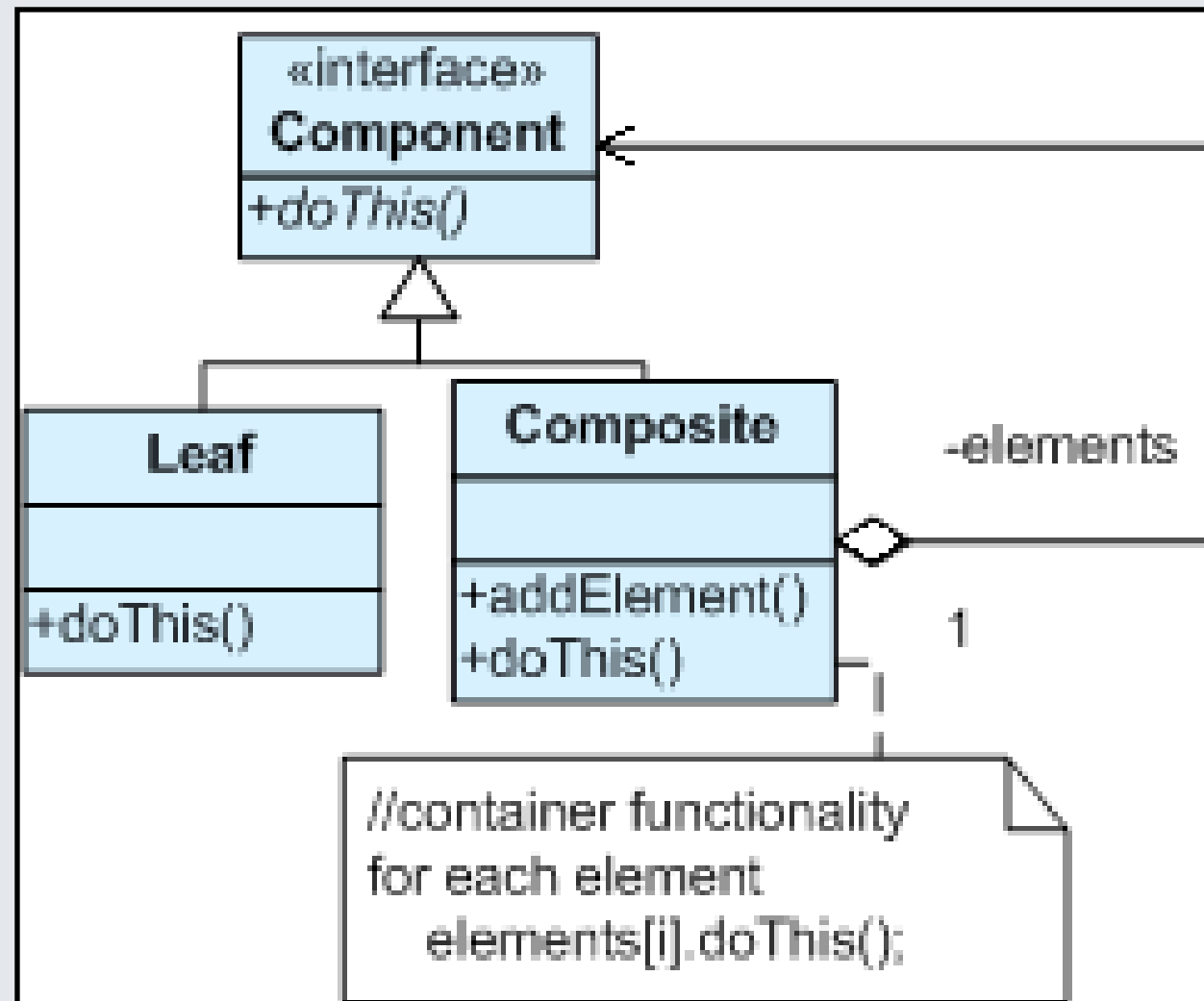
```
class MonAdaptateur : public Chauffage, private Four {  
    void allumer()      { ouvrirPorte(); chaud(); }  
    void chaud()        { thermostat(5); }  
    void tresChaud()    { thermostat(10); }  
    void eteindre()     { thermostat(0); fermerPorte(); }  
};
```


COMPOSITE

est un pattern permettant de représenter des "arborescences" ou "des composés" et d'y réaliser des traitements uniformes.

Ex : les composants des interfaces graphiques, imbriqués les uns dans les autres pour des raisons de présentation, sont typiques de ce pattern. Un évènement "survol de la souris" impacte potentiellement tous les composants imbriqués concernés.

COMPOSITE



COMPOSITE

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

```
class Leaf : public Composant {  
public:  
    void do_this() { ... }  
};
```

```
class ContainerComposite : public Composant {  
    vector<Composant *> all;  
public:  
    void add(Composant *c) { all.push_back(c); }  
    void do_this() {  
        ...  
        for (Composant * c:all) c -> do_it();  
    }  
};
```

COMPOSITE

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

Tentative d'application
triviale à du graphisme

```
class Bouton : public Composant {  
public:  
    string name;  
    Bouton(string n): name{n}{}  
    void do_this() { cout << "(" << name << ")"; }  
};
```

```
class Container : public Composant {  
    vector<Composant *> all;  
public:  
    void add(Composant *c) { all.push_back(c); }  
    void do_this() {  
        cout << "(";  
        for (Composant * c:all) c -> do_it();  
        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.do_this();  
}
```

((b1) ((b2) (b3) ()))

DECORATOR

ce pattern permet d'embellir fonctionnellement des objets existants.

C'est aussi une alternative à la dérivation.

DECORATOR

ce pattern permet d'embellir fonctionnellement des objets existants.

C'est aussi une alternative à la dérivation.

Dans l'exemple traité ici, on va s'intéresser à des sources de données (fichier, etc) qui peuvent être compressées ou encodées.

Dans une approche purement héritage on aurait :

Fichier, FichierEncodé, FichierCompressé, FichierEncodéEtCompressé

DECORATOR

ce pattern permet d'embellir fonctionnellement des objets existants.

C'est aussi une alternative à la dérivation.

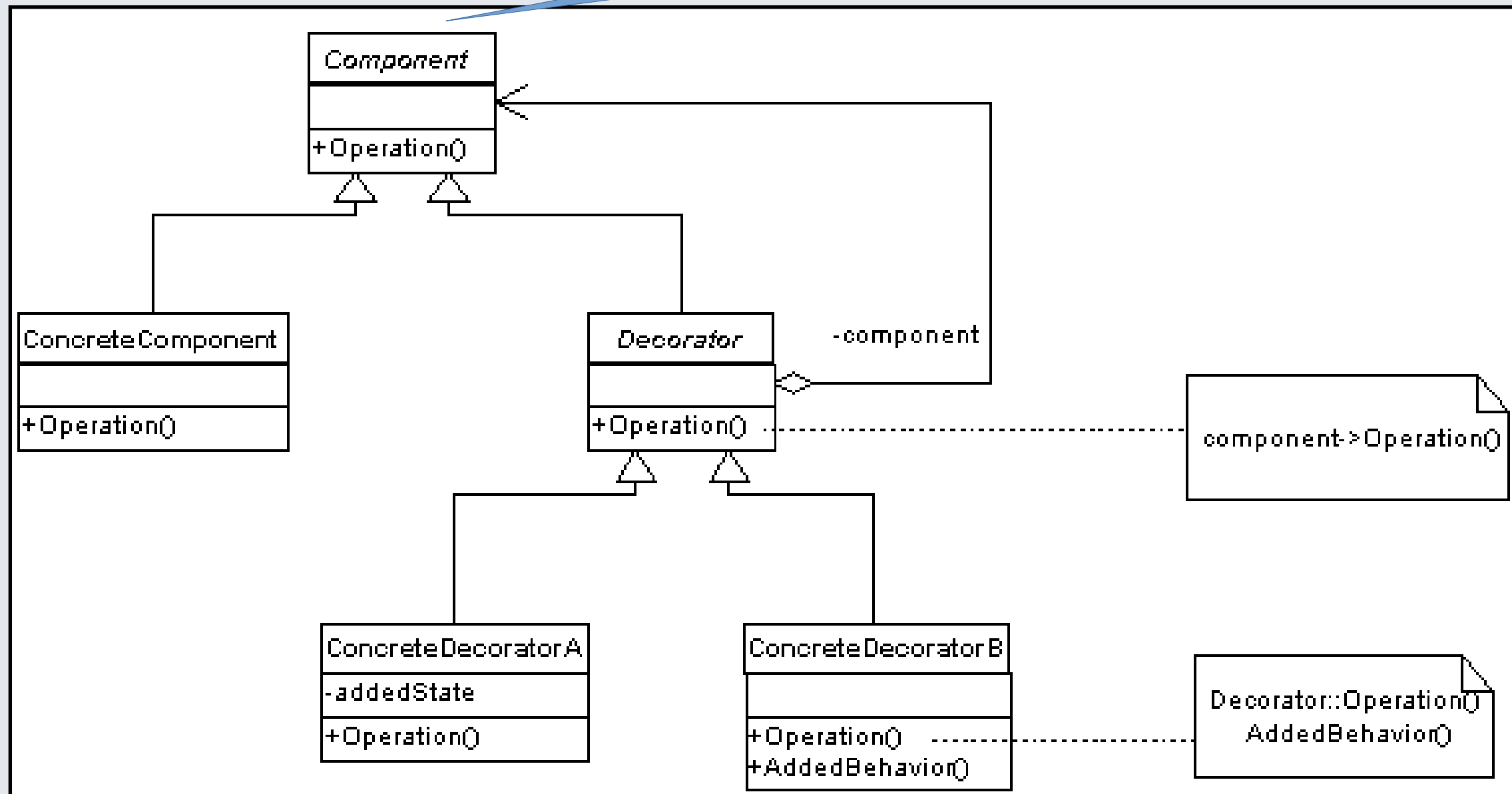
Dans l'exemple traité ici, on va s'intéresser à des sources de données (fichier, etc) qui peuvent être compressées ou encodées.

Dans une approche purement héritage on aurait :
Fichier, FichierEncodé, FichierCompressé, FichierEncodéEtCompressé

Si on avait BD en source de donnée (en plus de Fichier) il faudrait :
BD, BDEncodé, BDCompressé, BDEncodéEtCompressé

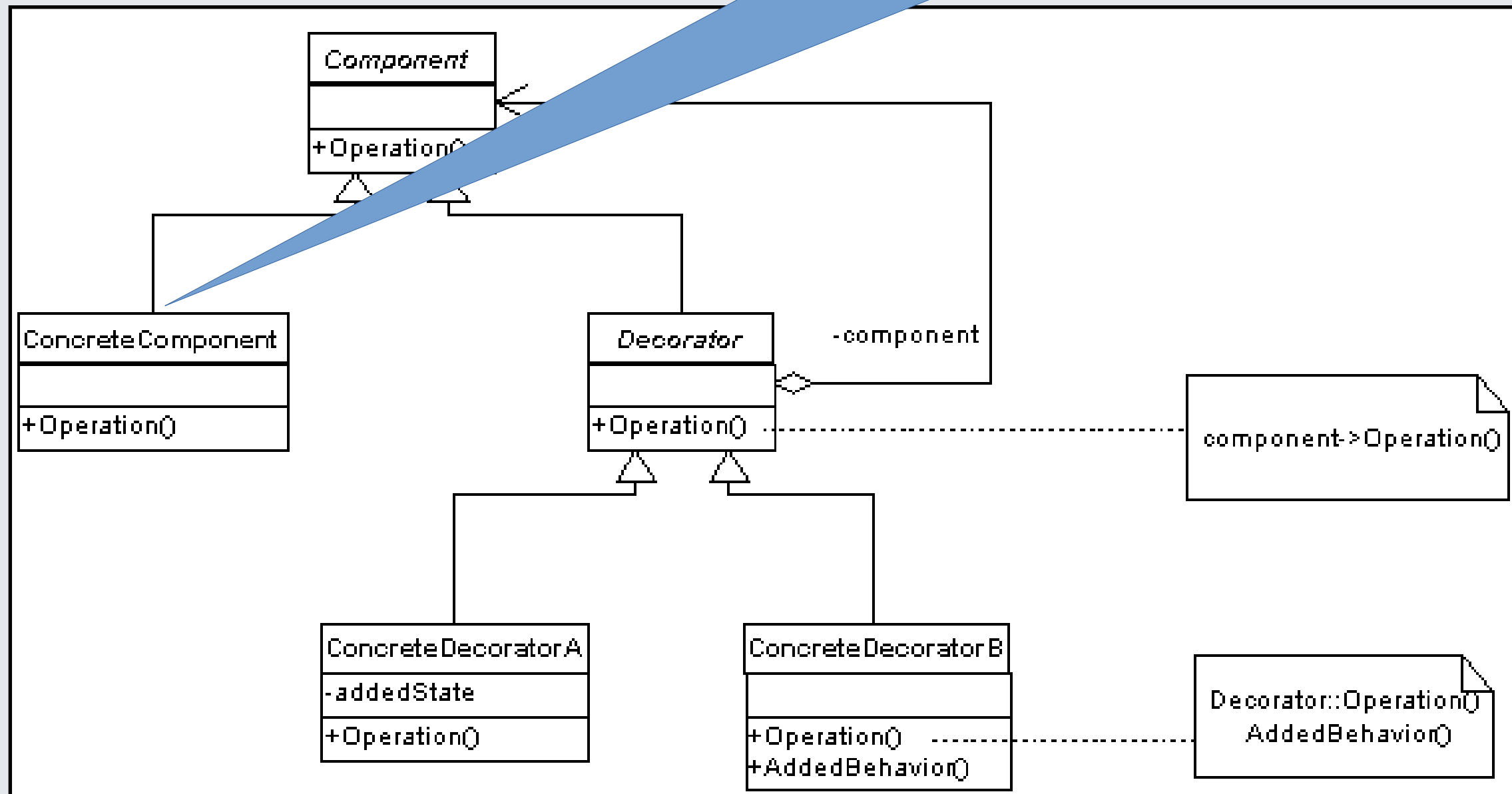
DECORATOR

Dans notre exemple, les objets manipulés seront des DataSources



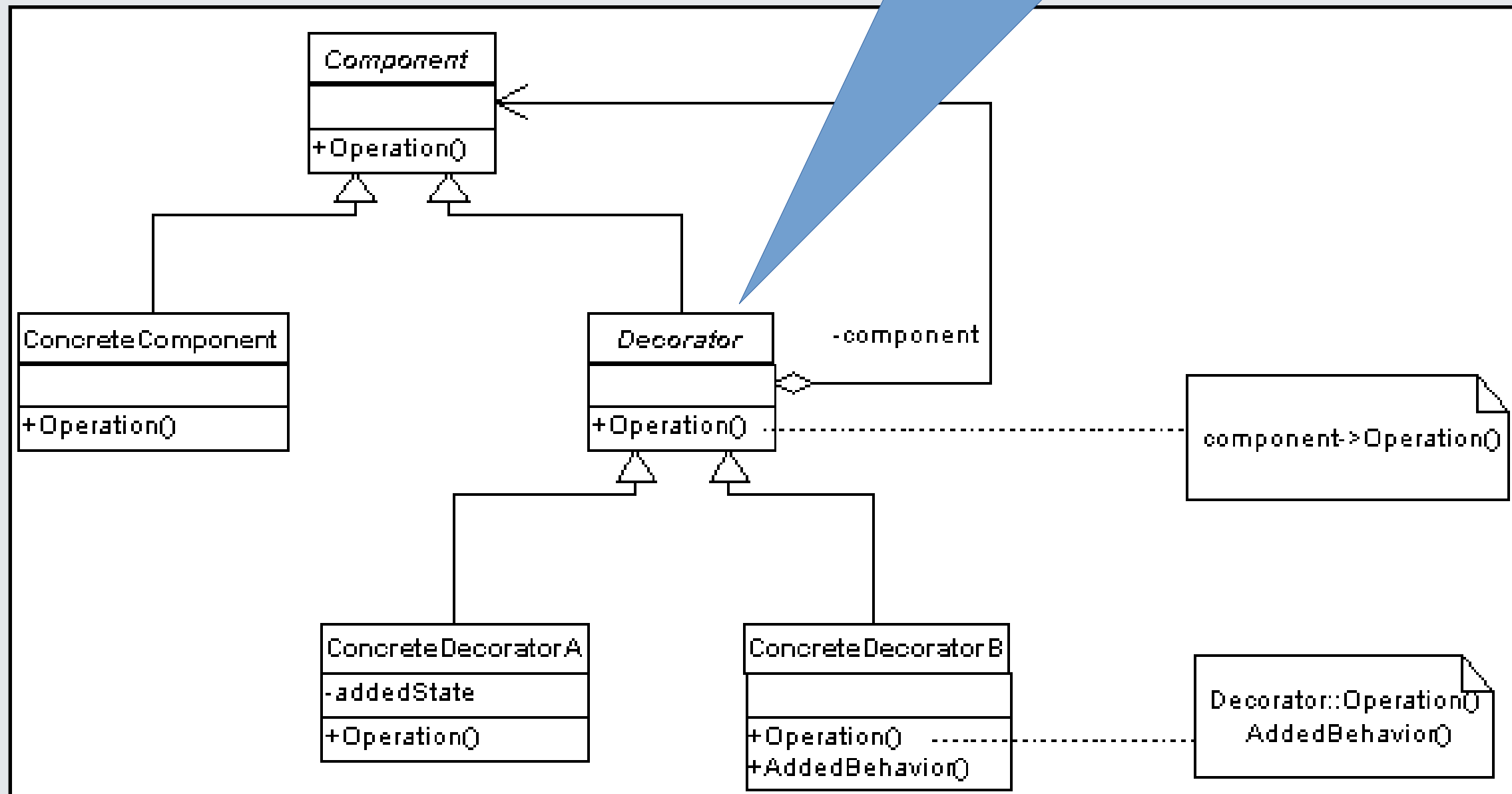
DECORATOR

par exemple, de base ce
seront des FileSource, des
StringSource ...



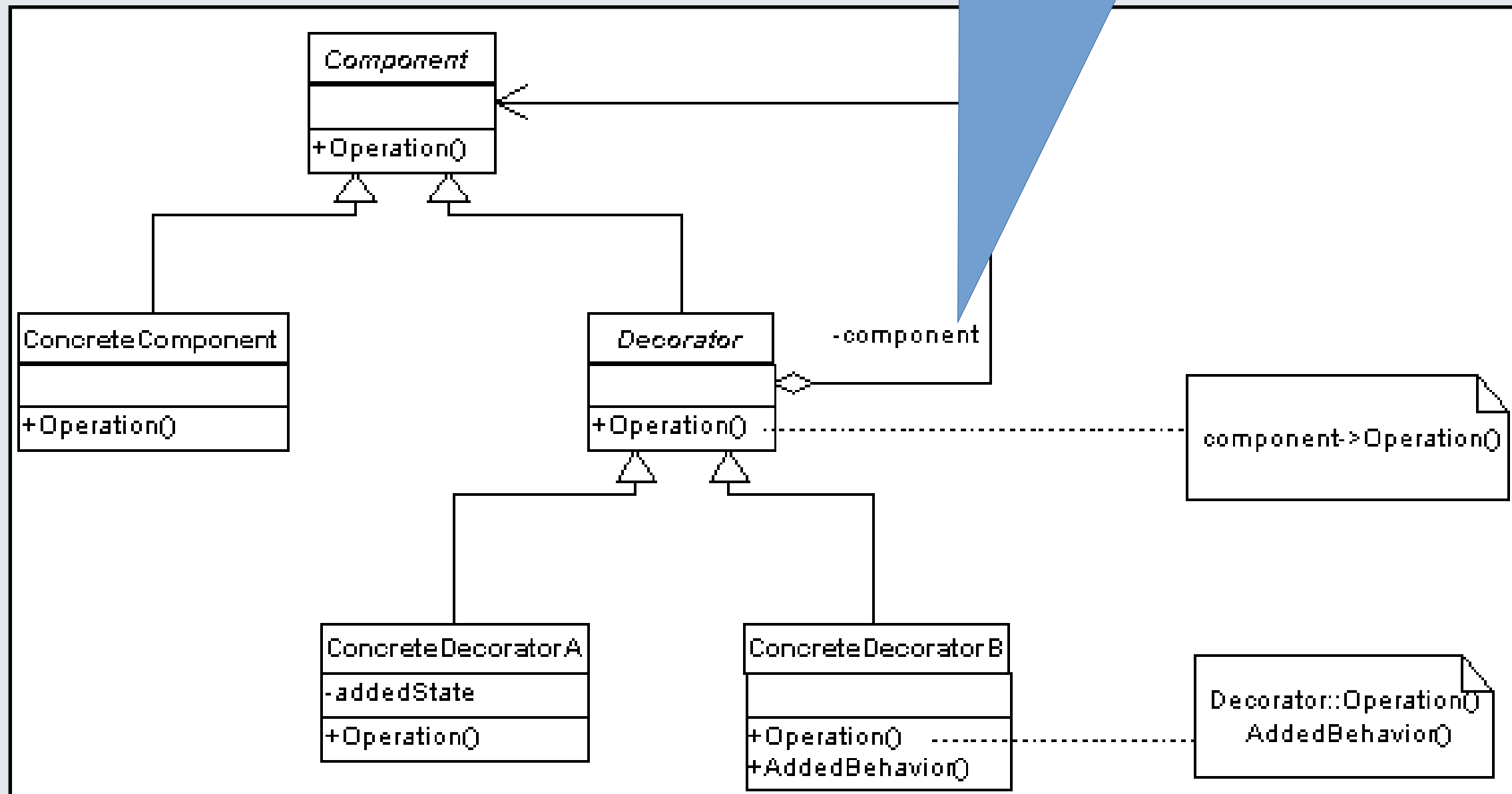
DECORATOR

Le Decorator est lui aussi une forme de DataSource



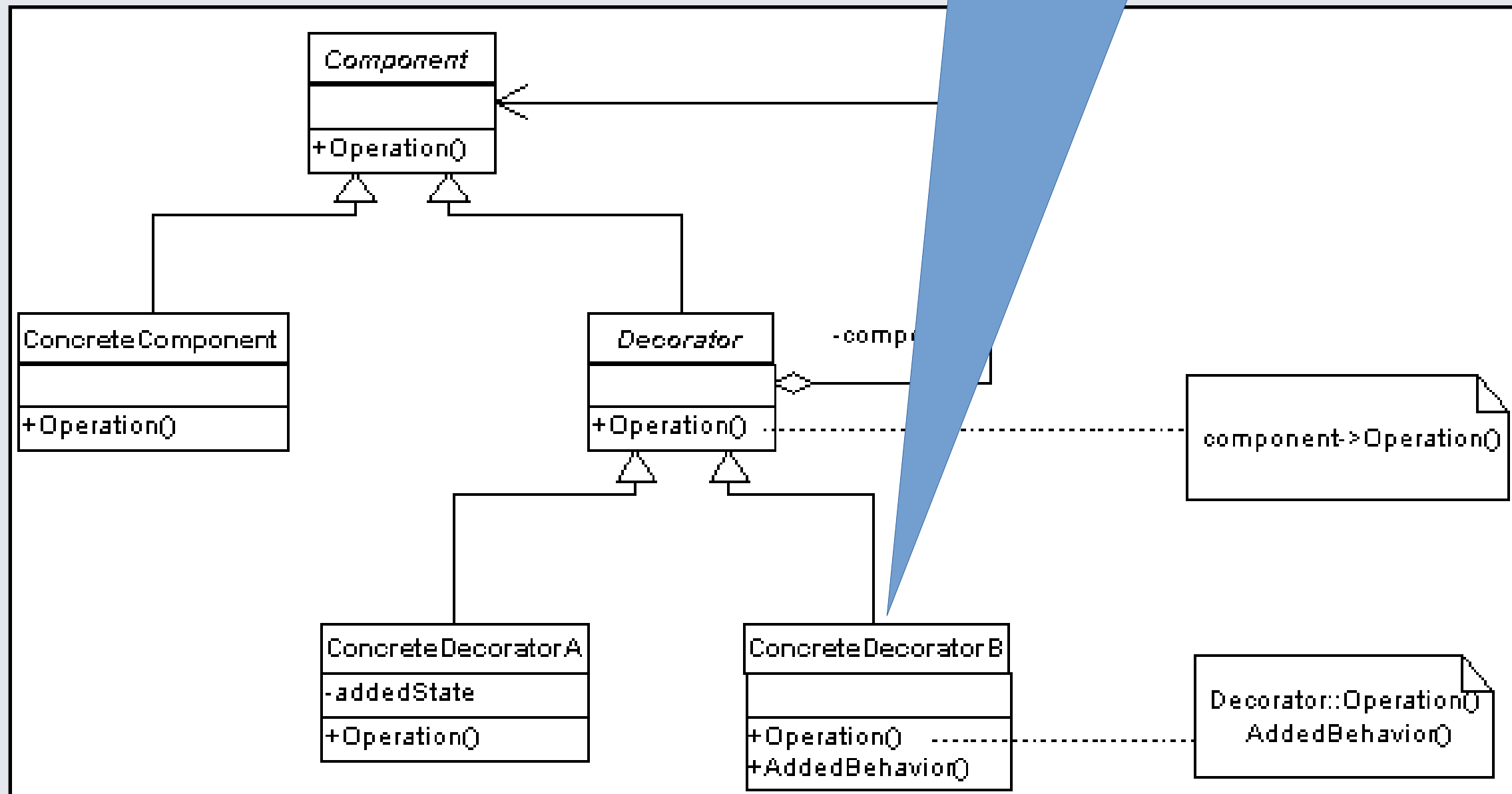
DECORATOR

construit au dessus d'une
autre DataSource



DECORATOR

Compression et Encodage
seront des Décorations



DECORATOR

(notre exemple)

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

la classe abstraite

et deux composants concrets

```
class StringSource : public DataSource {  
    string contenu;  
public:  
    StringSource(string s):contenu{s}{}  
    string read() {return contenu;}  
};
```

```
class FileSource : public DataSource {  
    string path;  
public:  
    FileSource(string p):path{p}{}  
    string read() { return "lecture from file"; }  
};
```

DECORATOR

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

sera utile aux
sous-classes

```
class Decorator : public DataSource{  
protected :  
    DataSource * source;  
public :  
    Decorator(DataSource *x):source{x}{}  
};
```

DECORATOR

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

```
class Decorator : public DataSource{  
protected :  
    DataSource * source;  
public :  
    Decorator(DataSource *x):source{x}{}  
};
```

```
class Encode : public Decorator {  
private :  
    string codage(string );  
public:  
    Encode(DataSource *x);  
    string read();  
};
```


DECORATOR

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

```
class Decorator : public DataSource{  
    protected :  
        DataSource * source;  
    public :  
        Decorator(DataSource *x):source{x}{}  
};
```

```
class Encode : public Decorator {  
    private :  
        string codage(string );  
    public:  
        Encode(DataSource *x);  
        string read();  
};
```

```
string Encode::codage(string s) {  
    string rep;  
    for (char c:s) rep+=(c+1); // mieux faire avec qq chose %26  
    return rep;  
}  
  
Encode::Encode(DataSource *x) : Decorator{x}{}  
  
string Encode::read() { return codage(source->read()); }
```

DECORATOR

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

```
class Decorator : public DataSource{  
protected :  
    DataSource * source;  
public :  
    Decorator(DataSource *x):source{x}{}  
};
```

```
class Encode : public Decorator {  
private :  
    string codage(string );  
public:  
    Encode(DataSource *x);  
    string read();  
};
```

```
int main() {  
    StringSource a{"aaaxxx"};  
    DataSource *s=new Encode(&a);  
    cout << s->read();  
    cout << a.read();  
}
```

bbbyyy
aaaxxx

```
string Encode::codage(string s) {  
    string rep;  
    for (char c:s) rep+=(c+1); // mieux faire avec qq chose %26  
    return rep;  
}  
  
Encode::Encode(DataSource *x) : Decorator{x}{}  
  
string Encode::read() { return codage(source->read()); }
```

DECORATOR

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

```
class Decorator : public DataSource{  
protected :  
    DataSource * source;  
public :  
    Decorator(DataSource *x):source{x}{}  
};
```

```
class Compress:public Decorator {  
private :  
    string compression(string s);  
public:  
    Compress(DataSource *);  
    string read();  
};
```

DECORATOR

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

```
class Decorator : public DataSource{  
protected :  
    DataSource * source;  
public :  
    Decorator(DataSource *x):source{x}{}  
};
```

```
class Compress:public Decorator {  
private :  
    string compression(string s);  
public:  
    Compress(DataSource *);  
    string read();  
};
```

```
string Compress::compression(string s) { return "["+s+"]"; }  
  
Compress::Compress(DataSource *x) : Decorator{x}{}  
  
string Compress::read() {  
    return compress(source->readAll());  
}
```

DECORATOR

```
class DataSource {  
public:  
    virtual string read()=0;  
};
```

```
class Decorator : public DataSource{  
protected :  
    DataSource * source;  
public :  
    Decorator(DataSource *x):source{x}{}  
};
```

```
class Compress:public DataSource{  
private :  
    string compression;  
public:  
    Compress(DataSource *s):source{s}{}  
    string read();  
};  
  
int main() {  
    StringSource a{"aaaxxx"};  
    DataSource* s=new Encode(new Compress( & a));  
    cout << s->read();  
}
```

[bbbyyy]

```
string Compress::compression(string s) { return "["+s+"]"; }  
  
Compress::Compress(DataSource *x) : Decorator{x}{}  
  
string Compress::read() {  
    return compress(source->readAll());  
}
```

DECORATOR

Remarque :

dans un soucis de clarté de la présentation, nous n'avons pas tenu compte des destructeurs.

```
int main() {  
    StringSource a;  
    DataSource* s=new Encode(new Compress( & a));  
    ...  
}
```

Ici les objets créés ne sont pas libérés.

Il faudrait faire un choix : le décorateur est-il oui ou non responsable de la durée de vie de sa source. Puis écrire les destructeurs...

Exercice :

Au moment où nous avons parlé de la redéfinition des opérateurs, nous avons envisagé une opération du style :

gateau=oeuf+lait+farine

Qu'aurait on tendance à utiliser pour l'accepter ?

Un decorateur ? Un builder ? Autre chose ?

COMPORTEMENT

ITERATOR

Vous avez donc à présent bien intégré ce pattern sans peut être tout à fait vous êtes rendu compte qu'au départ c'en était un ...

Mr JOURDAIN : Quoi ! quand je dis: « Nicole, apportez-moi mes pantoufles, et me donnez mon bonnet de nuit », c'est de la prose ?

MAÎTRE DE PHILO. : Oui, Monsieur.

Mr JOURDAIN : Par ma foi ! **il y a plus de quarante ans que je dis de la prose sans que j'en sache rien, et je vous suis le plus obligé du monde de m'avoir appris cela.**

....

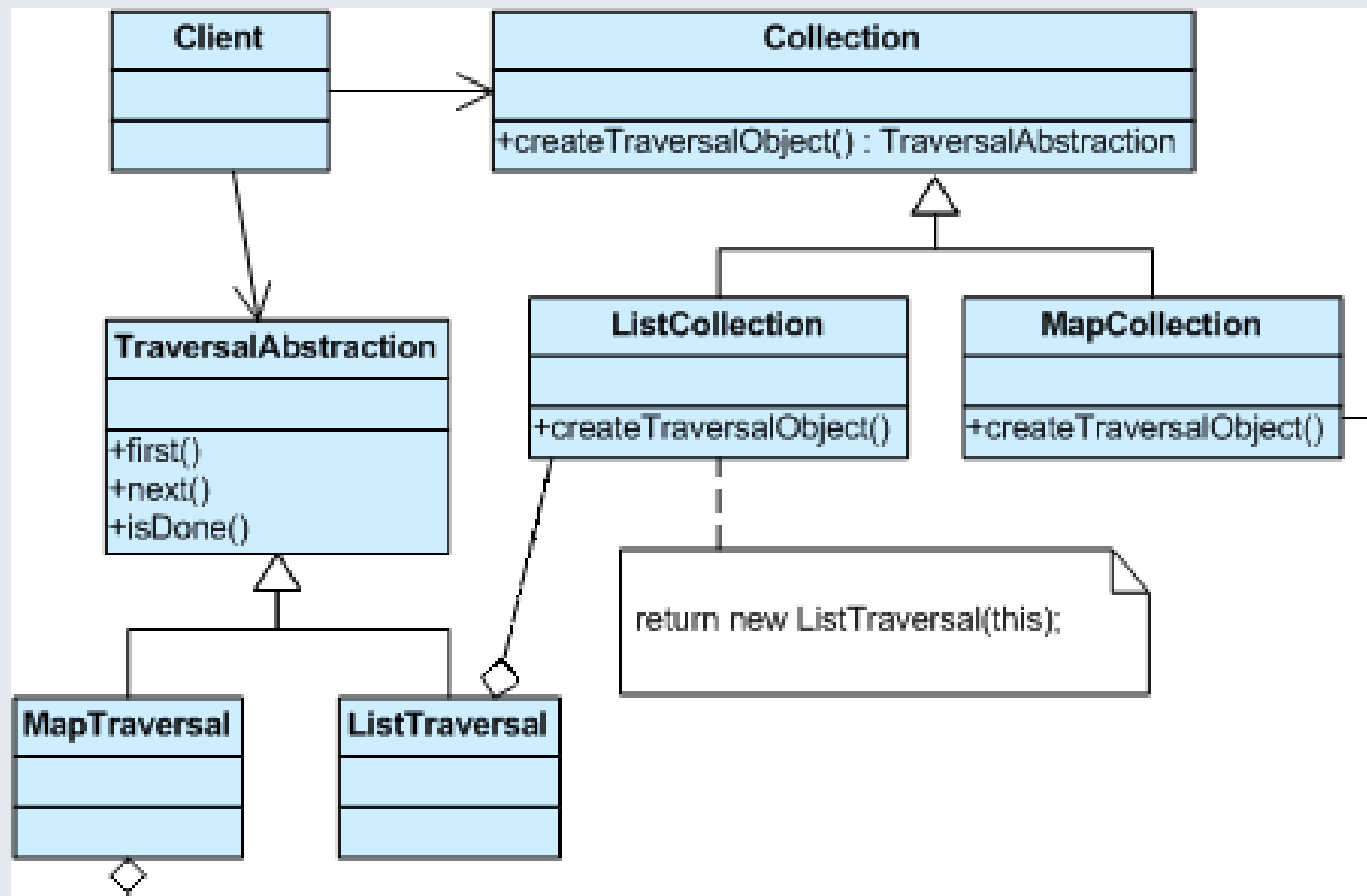
MAÎTRE DE PHILO. : On les peut mettre premièrement comme vous avez dit : « bla bla bla ...» Ou bien : « bli bli bli». Ou bien : « bla bli bla». Ou bien : (etc ..)

MONSIEUR JOURDAIN : **Mais de toutes ces façons-là, laquelle est la meilleure ?**

MAÎTRE DE PHILO. : **Celle que vous avez dite** : « Belle Marquise, vos beaux yeux me font mourir d'amour ».

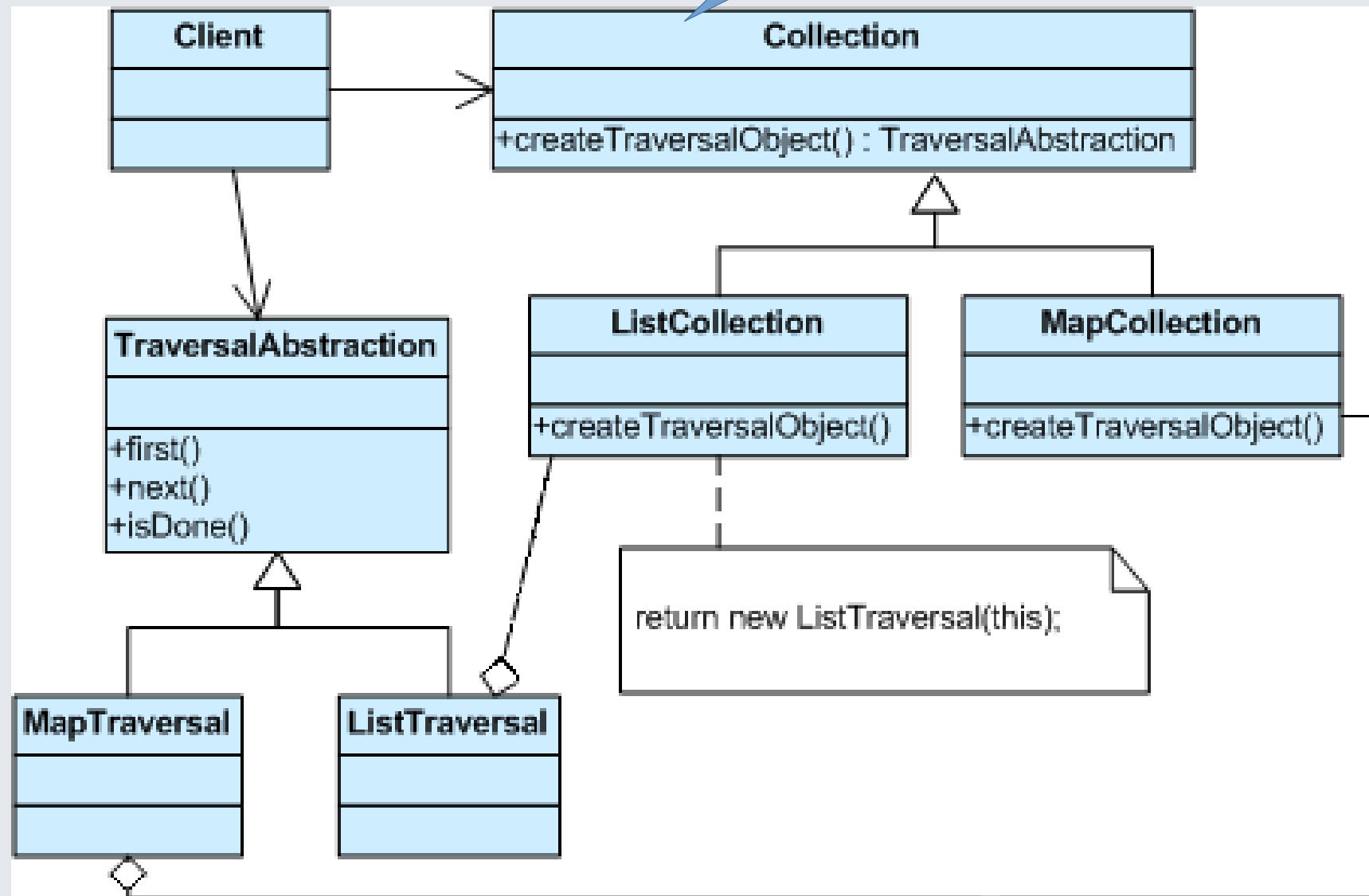
MONSIEUR JOURDAIN : **Cependant je n'ai point étudié, et j'ai fait cela tout du premier coup.**

ITERATOR



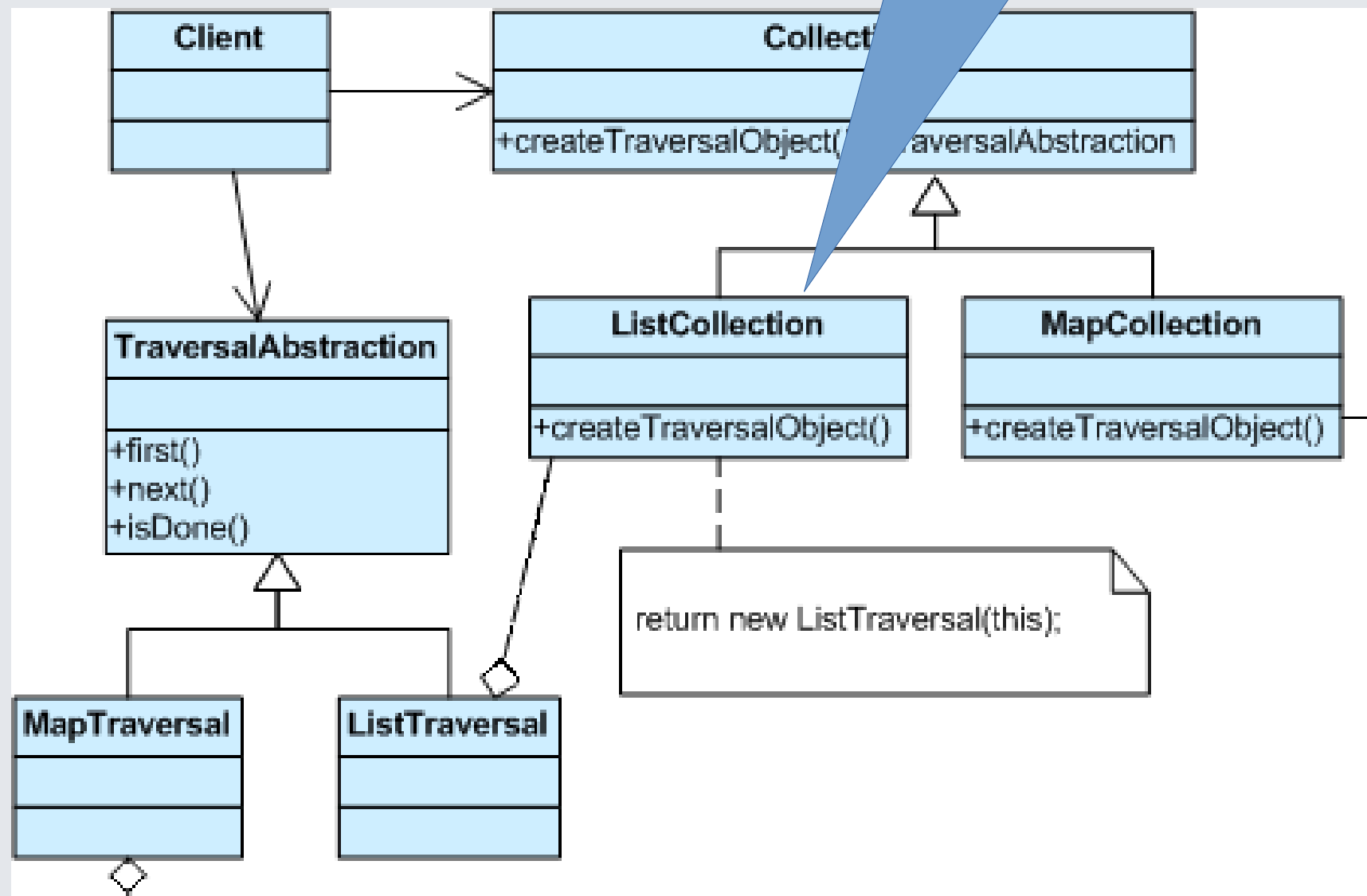
ITERATOR

les itérateurs sont associés à un travail sur des ensembles (Collection)



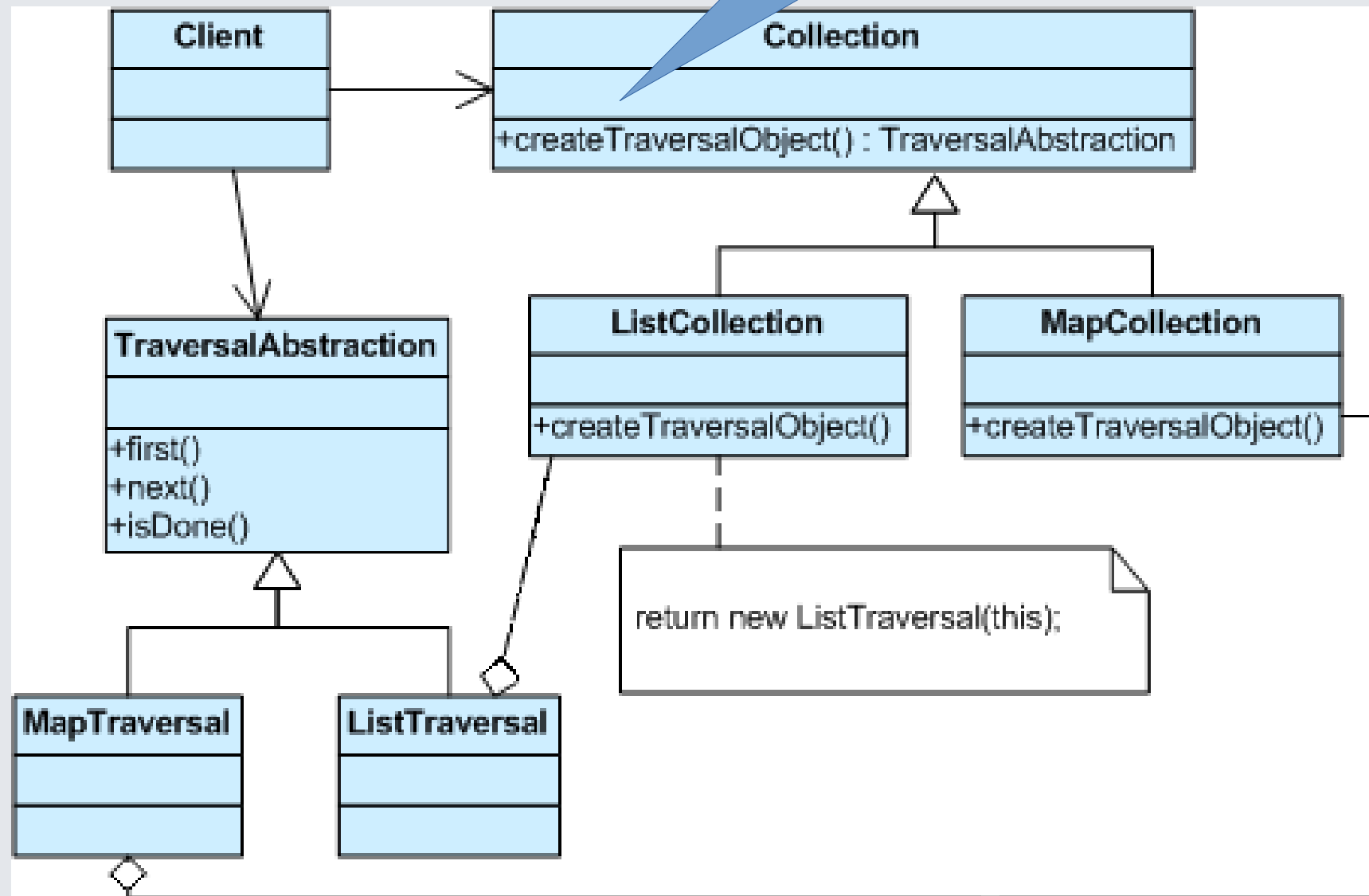
ITERATOR

par exemple des listes, des arbres, etc



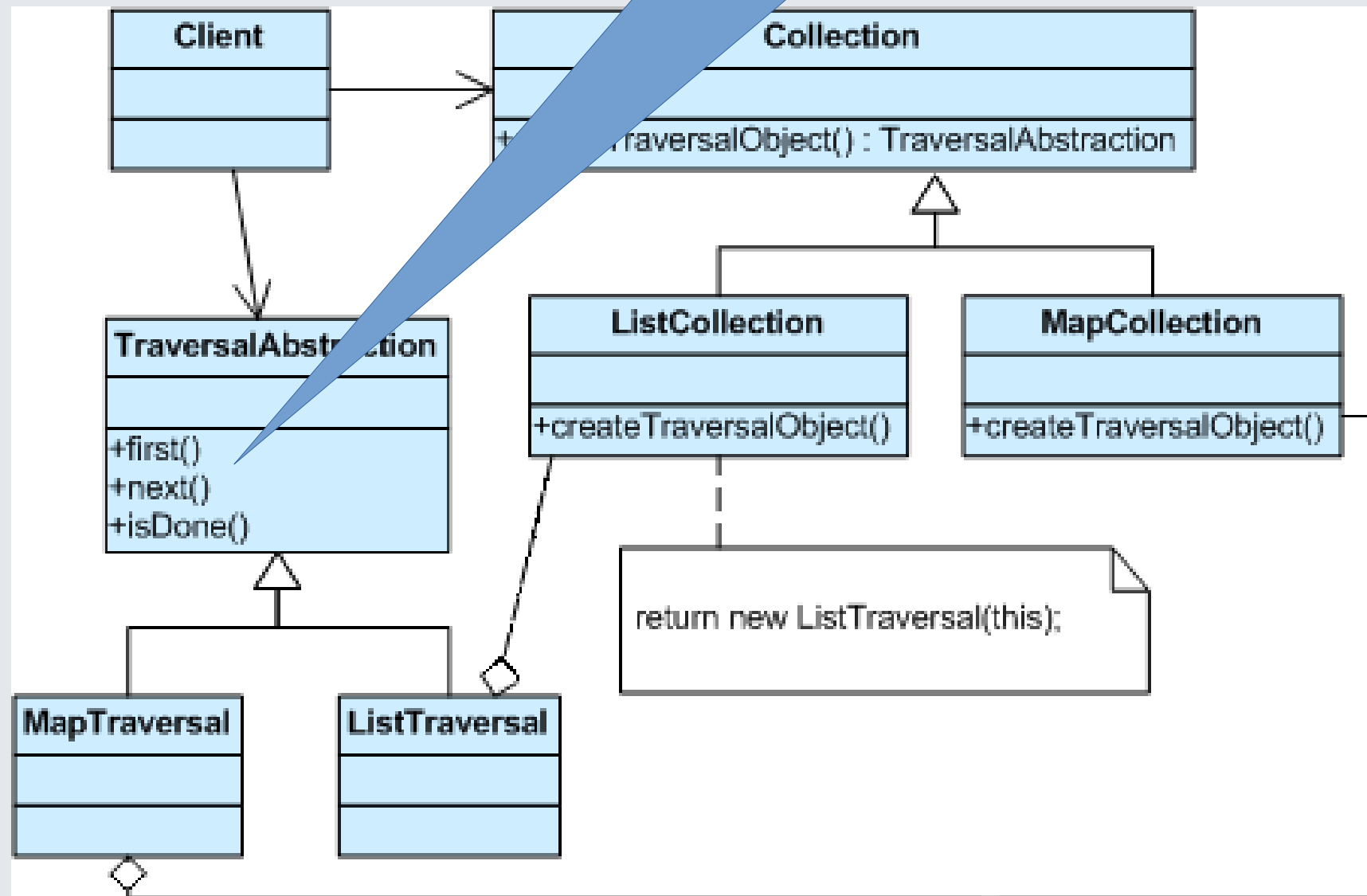
ITERATOR

C'est la Collection qui va délivrer un objet traversant (c.à d. un itérateur)



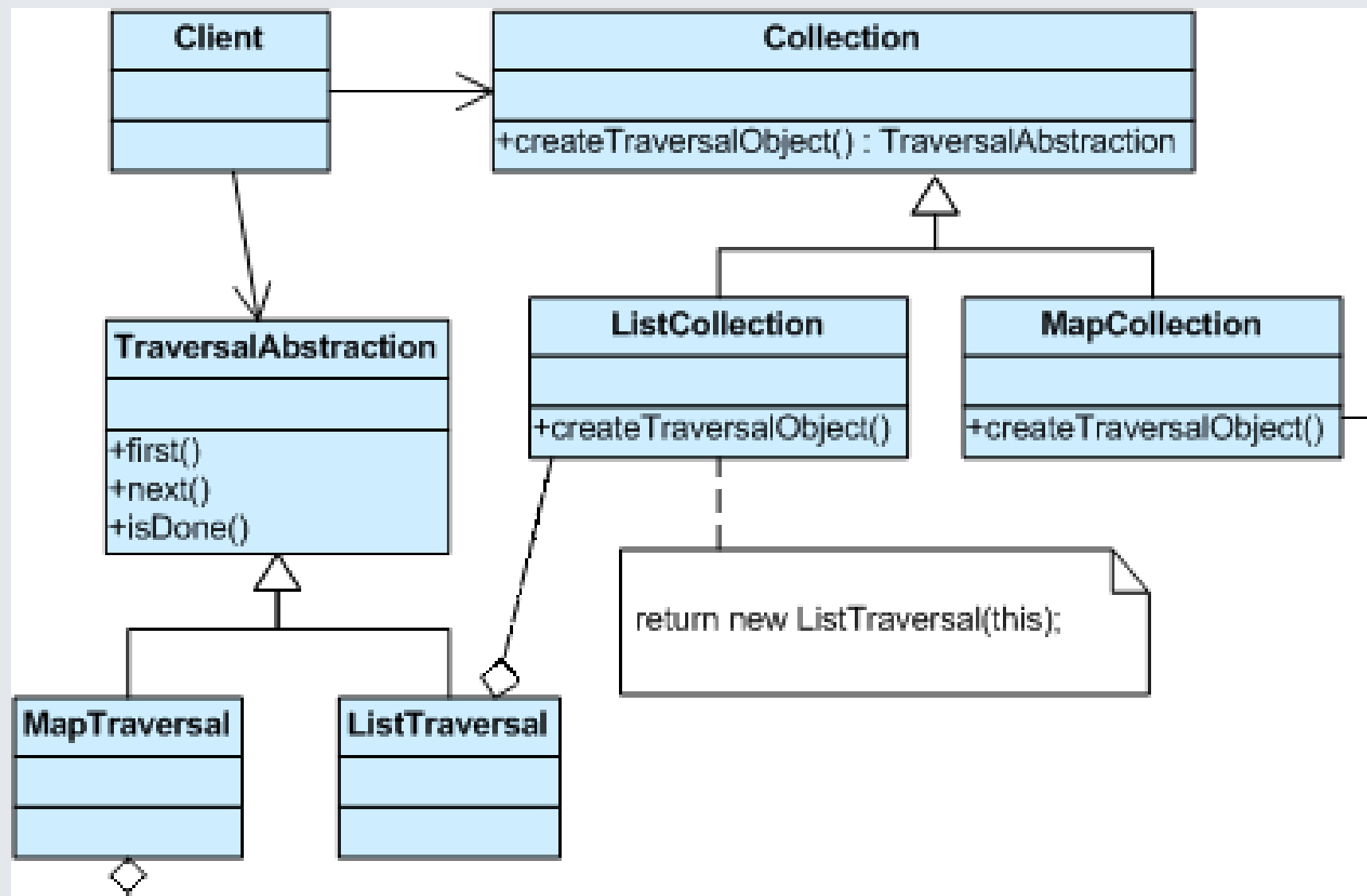
ITERATOR

Voici les méthodes associées



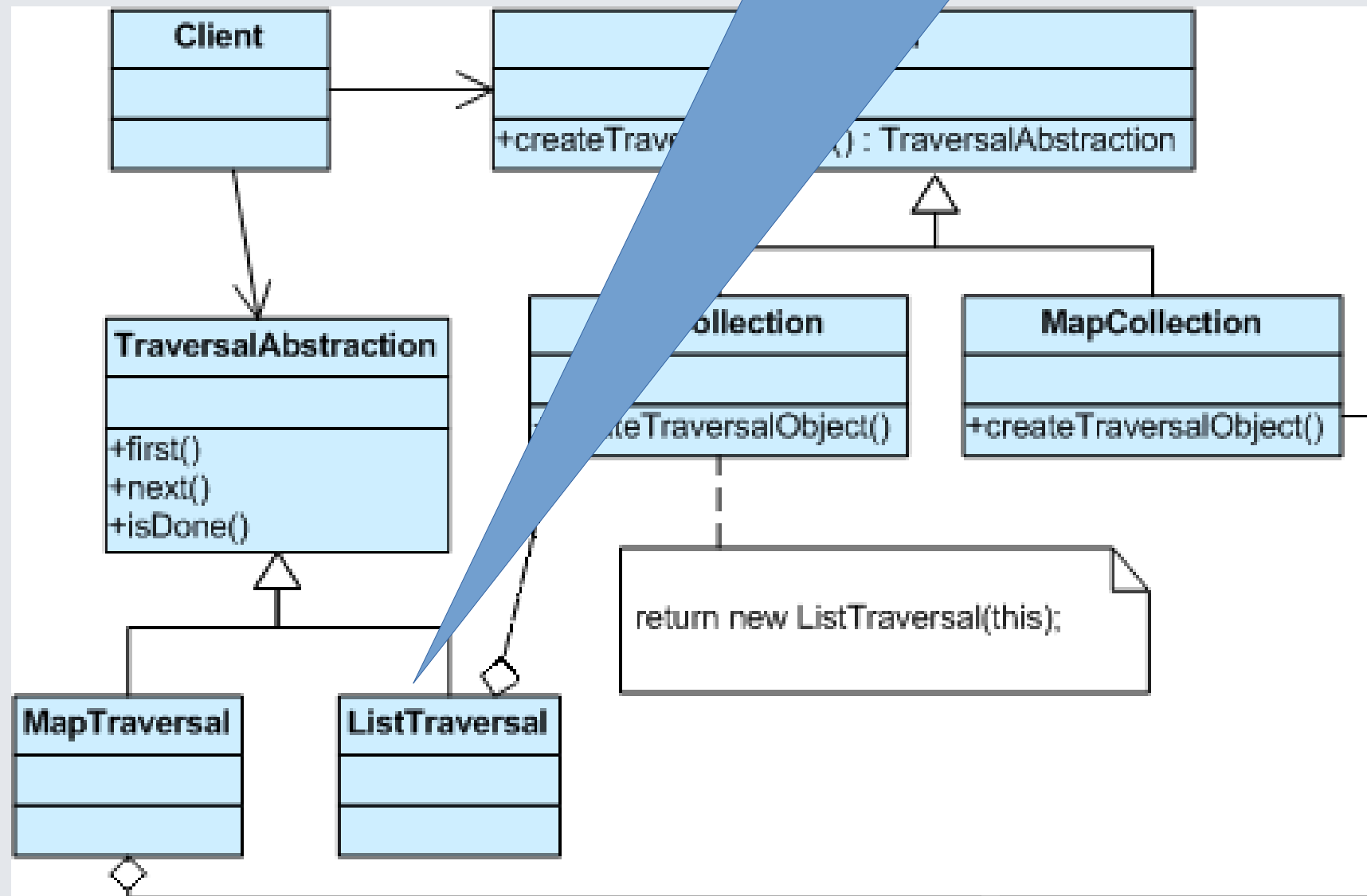
ITERATOR

Le client, lui, travaillera sur des abstractions



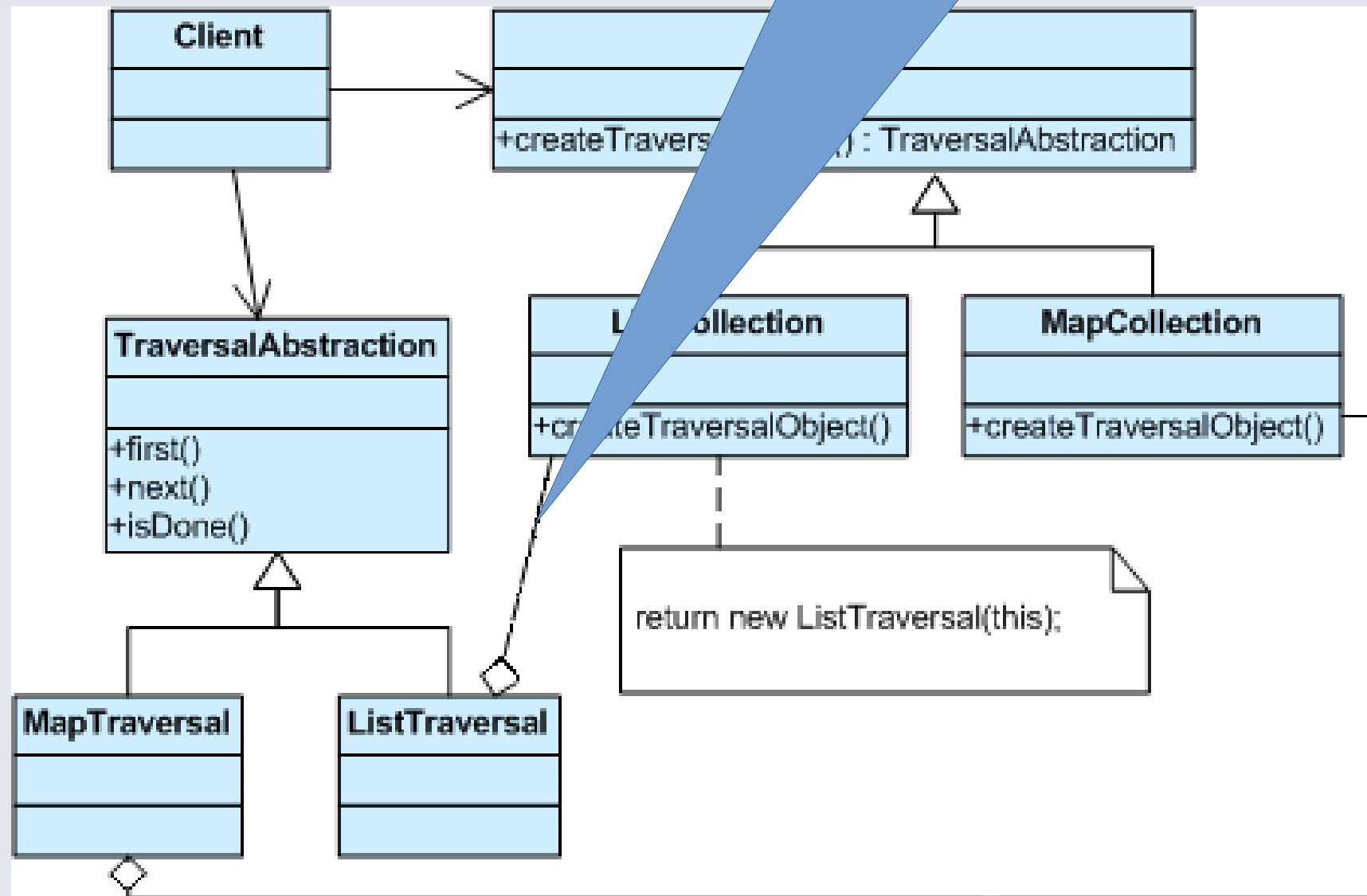
ITERATOR

Il y aura des itérateurs dédiés
aux types concrets des Ensembles



ITERATOR

L'itérateur sait quelle est la collection qu'il parcourt

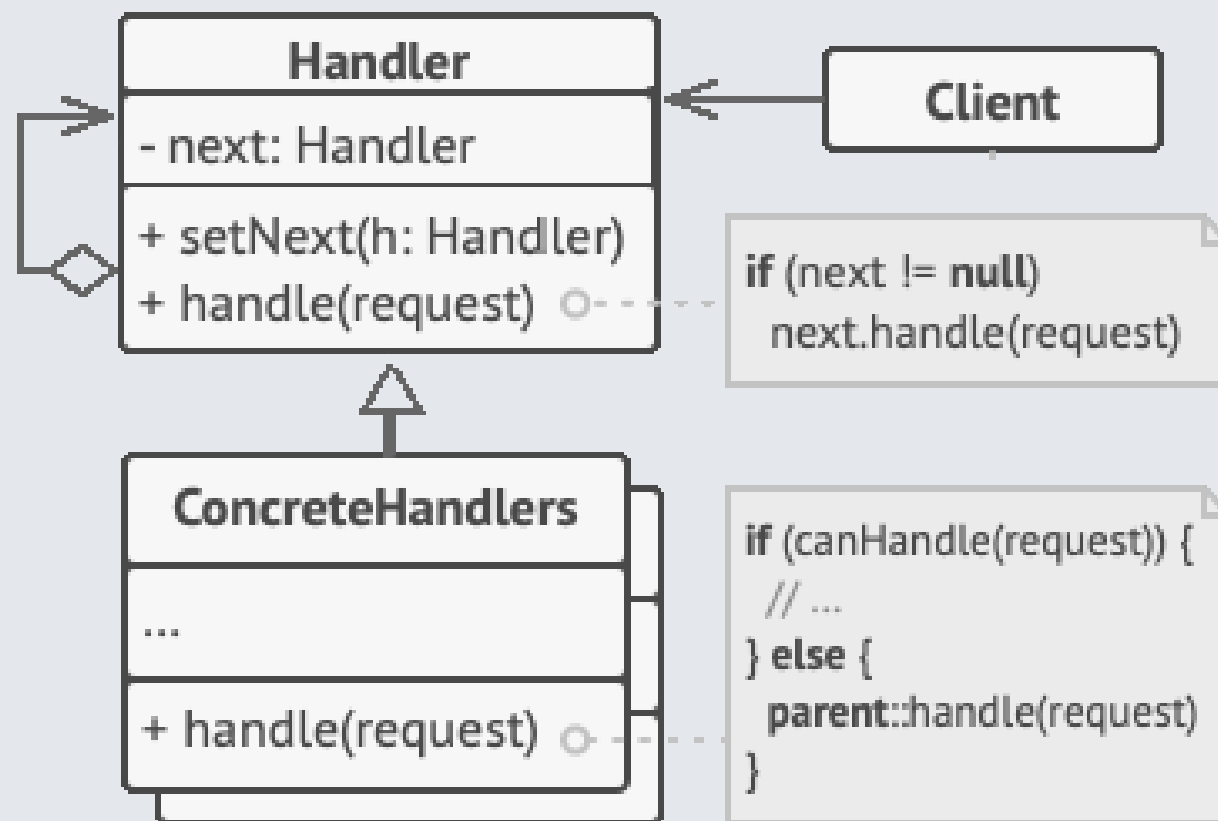


CHAIN OF RESPONSABILITY

le pattern chaîne de responsabilité est utilisé pour faire faire un traitement à un objet qui lui même peut posséder des sous-traitants.

Les objets capables de réaliser le traitement s'inscrivent dans une chaîne, elle est parcourue jusqu'à trouver quelqu'un qui accepte d'assumer le traitement demandé.

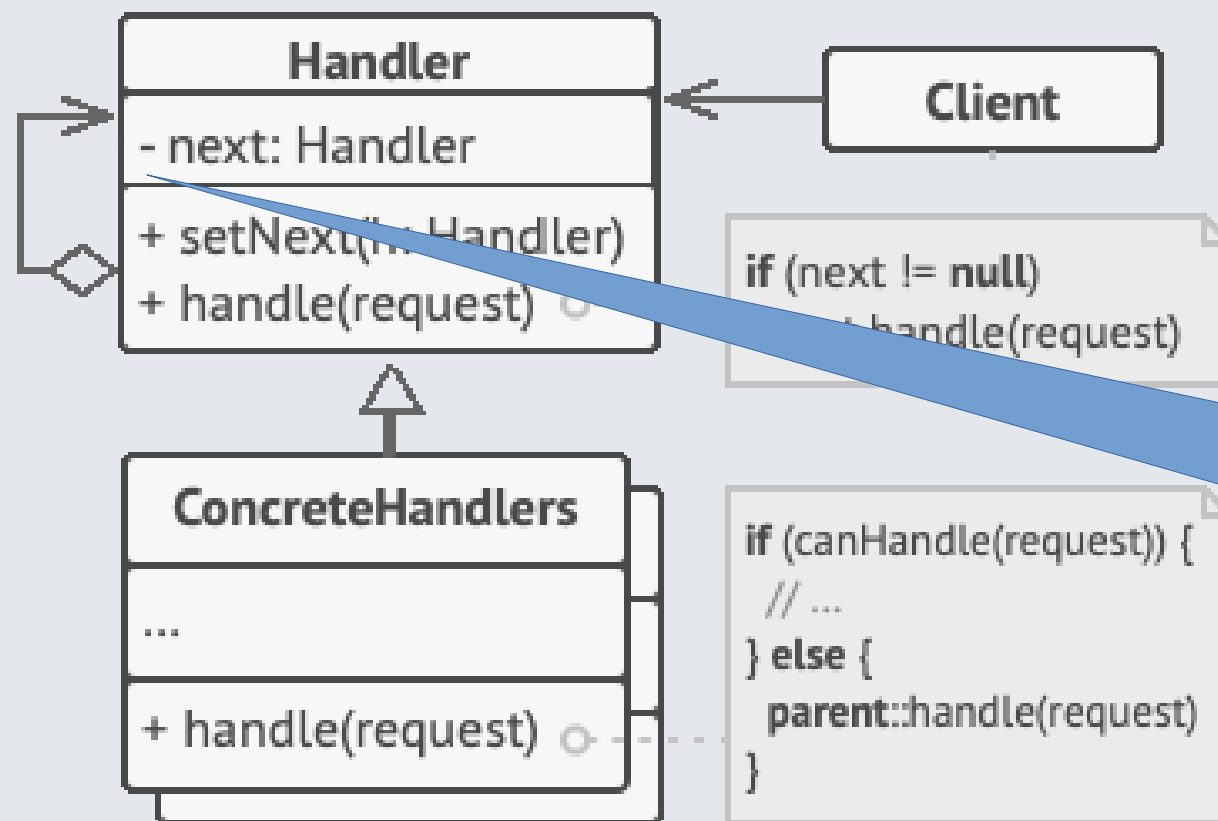
CHAIN OF RESPONSABILITY



Handler se traduit par
« gestionnaire » (de la chaîne)

handle par « gère »

CHAIN OF RESPONSABILITY

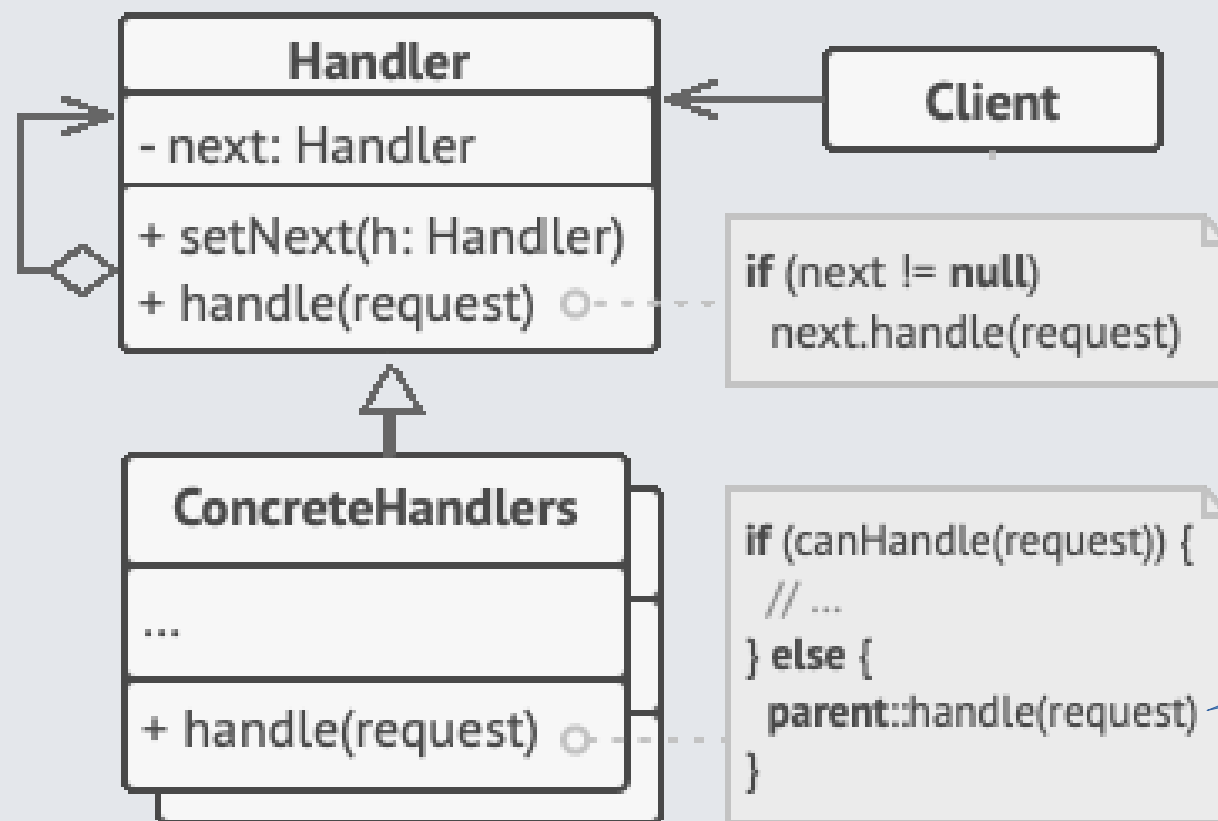


Handler se traduit par
« gestionnaire » (de la chaîne)

handle se traduit par « gère »

next est private :
les gestionnaires ne
connaîtront pas leurs
collègues

CHAIN OF RESPONSABILITY

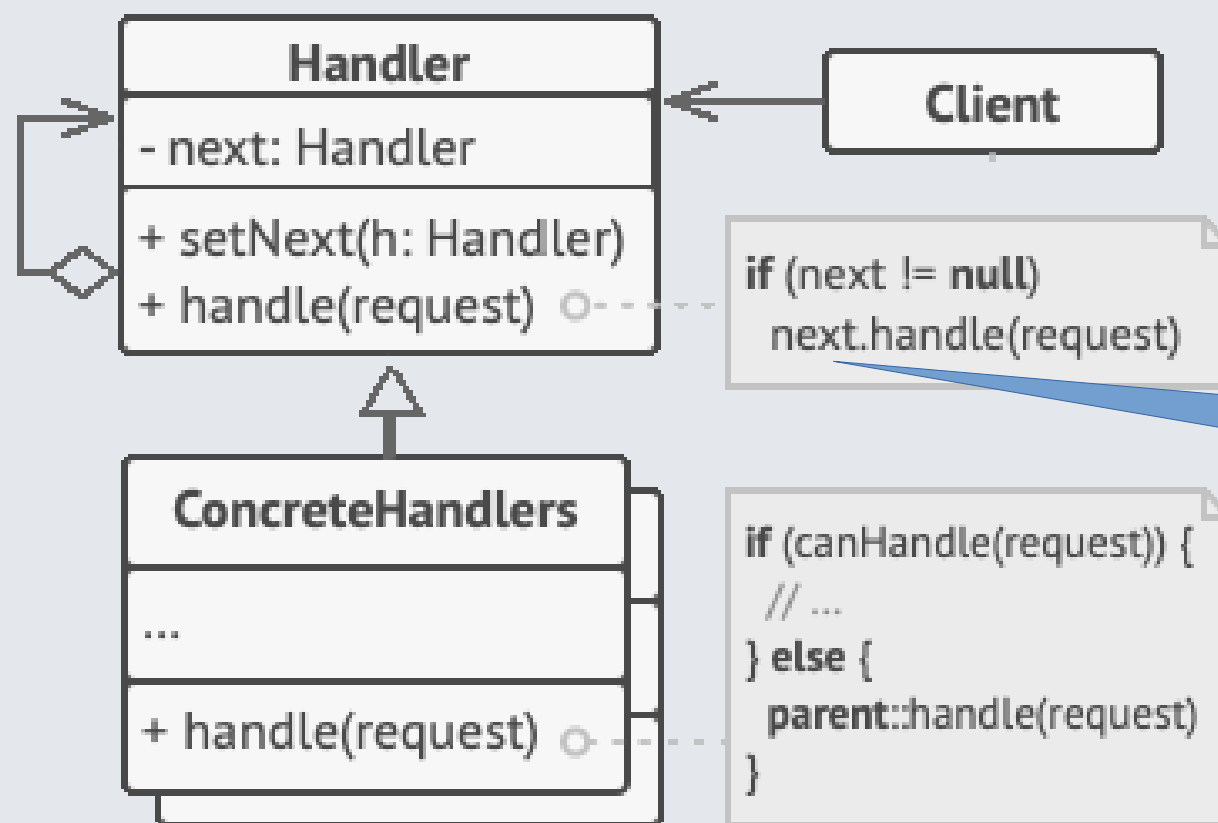


Handler se traduit par
« gestionnaire » (de la chaîne)

handle se traduit par « gère »

passse au suivant en
revenant au parent

CHAIN OF RESPONSABILITY



Handler se traduit par
« gestionnaire » (de la chaîne)

handle se traduit par « gère »

qui regarde s'il y
en a d'autres

CHAIN OF RESPONSABILITY

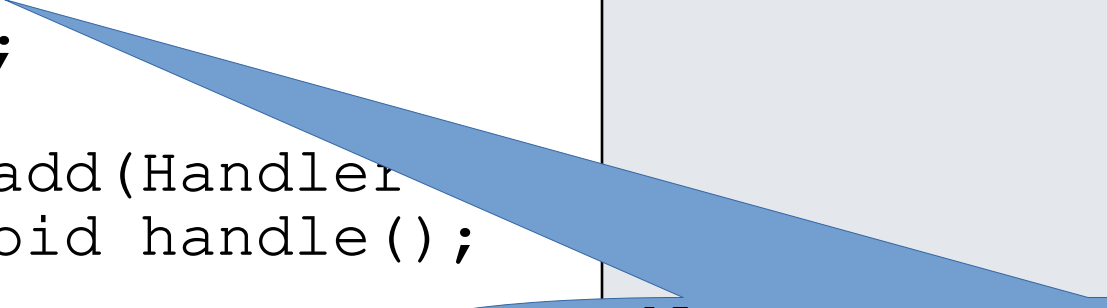
```
class Handler {  
    private :  
        Handler *next;  
    protected:  
        Handler();  
    public:  
        Handler& add(Handler &);  
        virtual void handle();  
};
```



privé

CHAIN OF RESPONSABILITY

```
class Handler {  
    private :  
        Handler *next;  
    protected:  
        Handler();  
    public:  
        Handler& add(Handler);  
        virtual void handle();  
};
```



Handler reste abstrait
(non constructible publiquement)

CHAIN OF RESPONSABILITY

```
class Handler {  
    private :  
        Handler *next;  
    protected:  
        Handler();  
    public:  
        Handler& add(Handler &);  
        virtual void handle();  
};
```

légère variante
meilleur choix que setNext
non virtual

CHAIN OF RESPONSABILITY

```
class Handler {  
    private :  
        Handler *next;  
    protected:  
        Handler();  
    public:  
        Handler& add(Handler &);  
        virtual void handle();  
};
```

virtual, mais non pure

sans arguments : ici on néglige
la requête

CHAIN OF RESPONSABILITY

```
class Handler {  
    private :  
        Handler *next;  
    protected:  
        Handler();  
    public:  
        Handler& add(Handler &);  
        virtual void handle();  
};
```

```
Handler::Handler() : next{nullptr} {}  
Handler& Handler::add(Handler &p) {  
    if (next!=nullptr) next->add(p);  
    else next = &p;  
    return *this; // pour pouvoir enchaîner les add  
}  
virtual void Handler::handle() {  
    if (next!=nullptr) next->handle();  
    else cout << "désolé, personne n'a fait le job ..." ;  
}
```

CHAIN OF RESPONSABILITY

```
class ConcreteHandler : public Handler {
public:
    // le constructeur par défaut est ok
    void handle();
};
```

```
void ConcreteHandler::handle() {
    if (rand()%100<75) {
        cout << "pas moi ";
        Handler::handle();
    }
    else
        cout << "je m'en occupe";
}
```

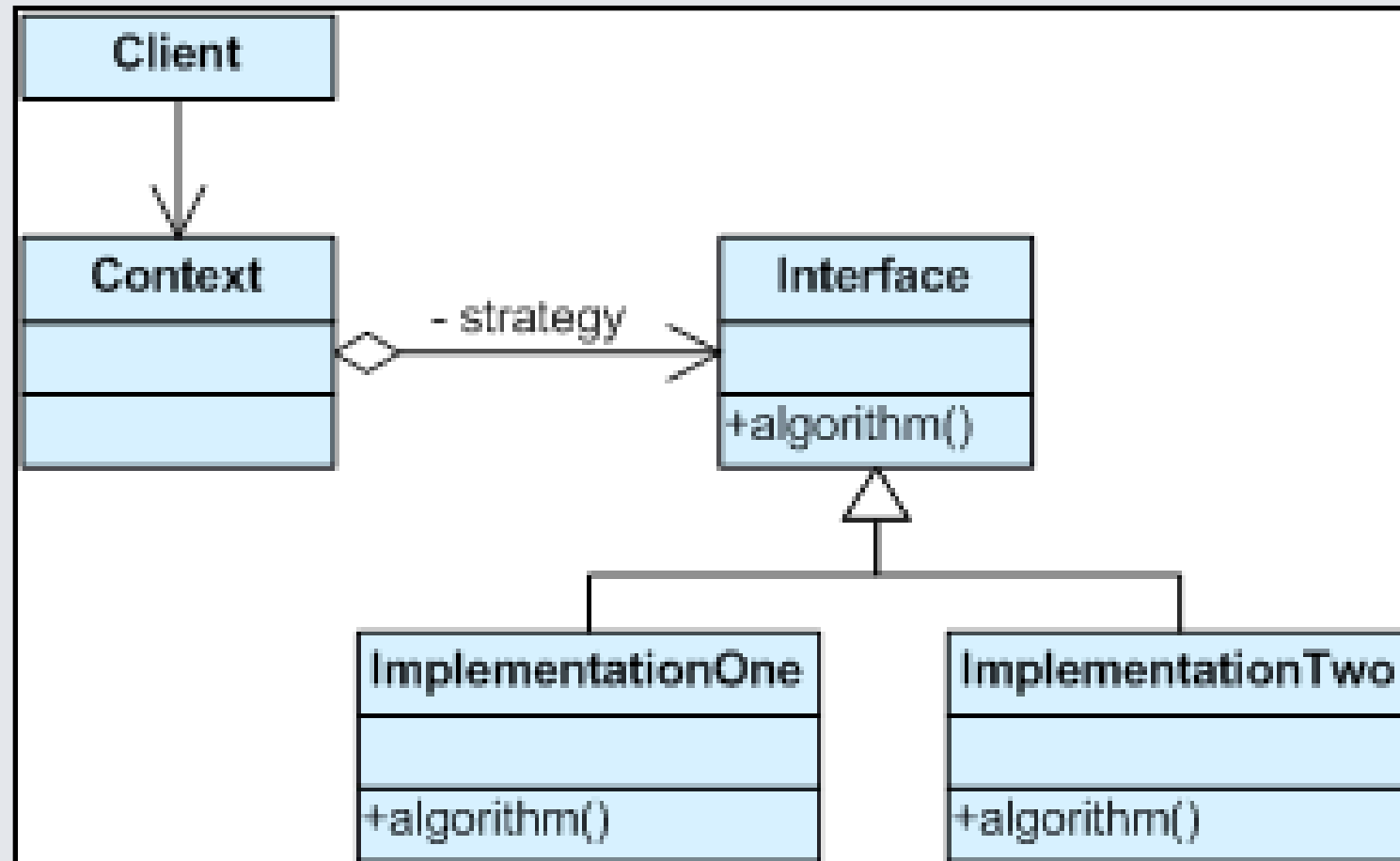
```
int main() {
    ConcreteHandler p1, p2, p3;
    Handler &c{p1}; // on choisi la tête de liste
    c.add(p2).add(p3);
    c.handle();
}
```

STRATEGY

c'est un pattern qui permet d'attribuer un comportement à un objet.

on peut également aborder ce même problème avec un autre pattern (state), et éventuellement les combiner tous les deux ...

STRATEGY

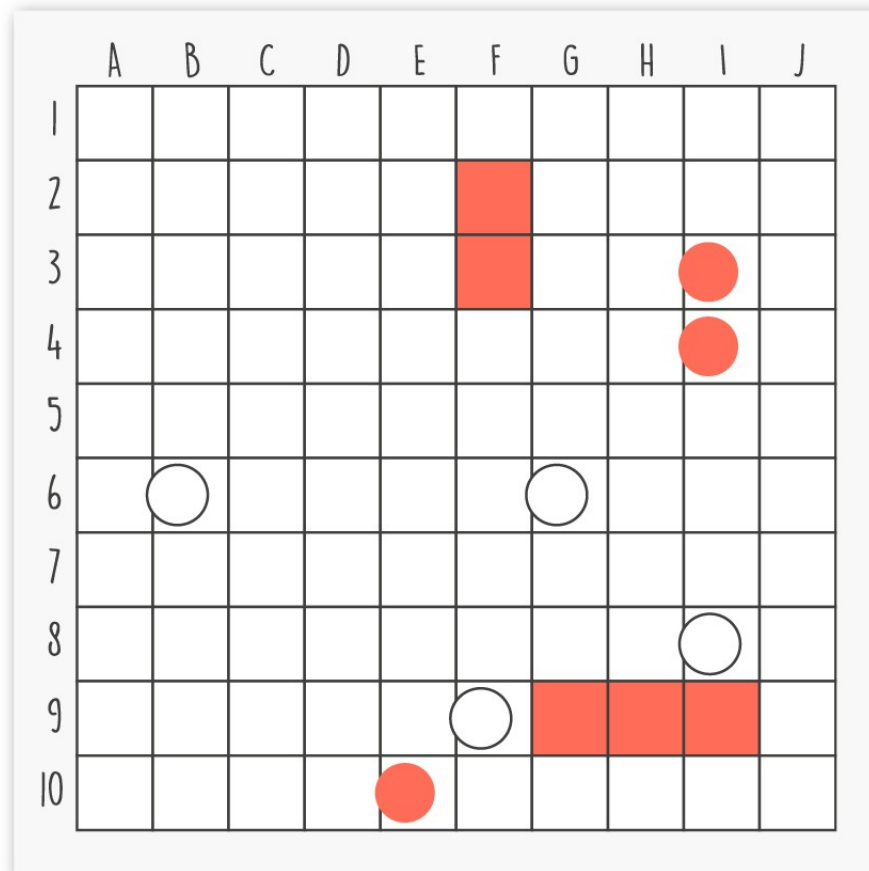


dans context on peut envisager de changer de « stratégie »
c'est à dire d'algorithme de décision.

Traitions un exemple

STRATEGY

LA BATAILLE NAVALE EXEMPLE D'UNE GRILLE D'ATTAQUE



TIR RATÉ



BATEAU ENNEMI TOUCHÉ



BATEAU ENNEMI COULÉ

*Je Suis
Animateur.fr*

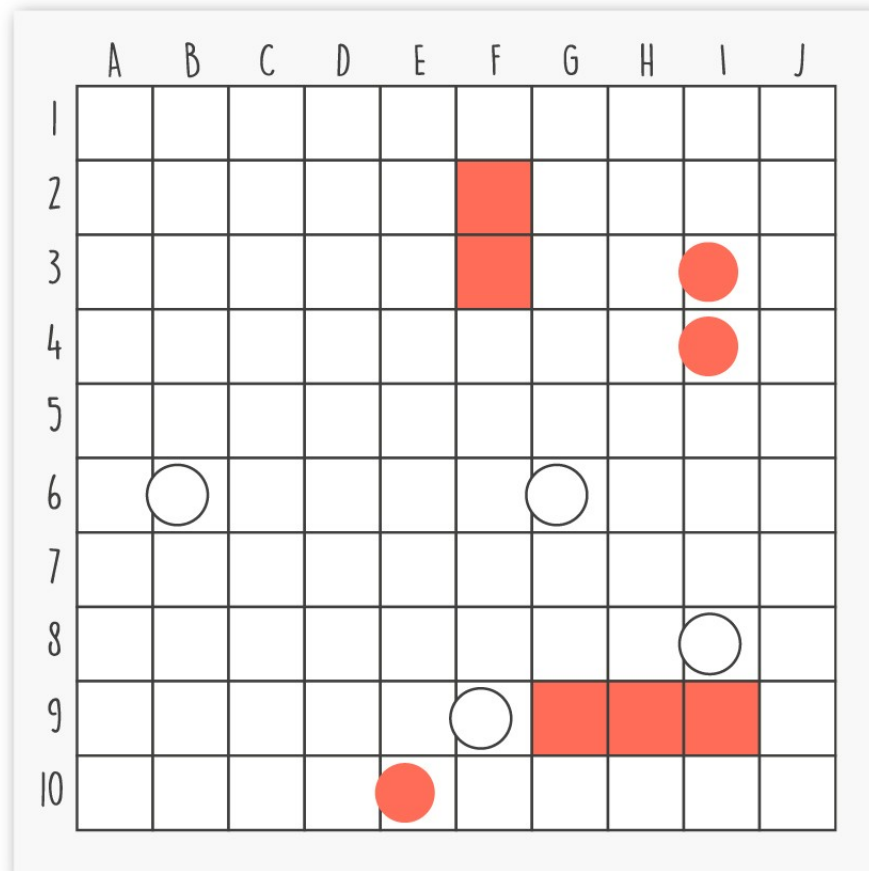
TOUS DROITS RÉSERVÉS ©

```
class Strategy {  
public:  
    virtual Coup choix(Grille &)=0;  
};
```


STRATEGY

```
Coup Strategy_0::choix(Grille &g) {  
    // n'importe quoi au hasard  
}
```

LA BATAILLE NAVALE EXEMPLE D'UNE GRILLE D'ATTAQUE



- TIR RATÉ
- BATEAU ENNEMI TOUCHÉ
- BATEAU ENNEMI COULÉ

*Je Suis
Animateur.fr*

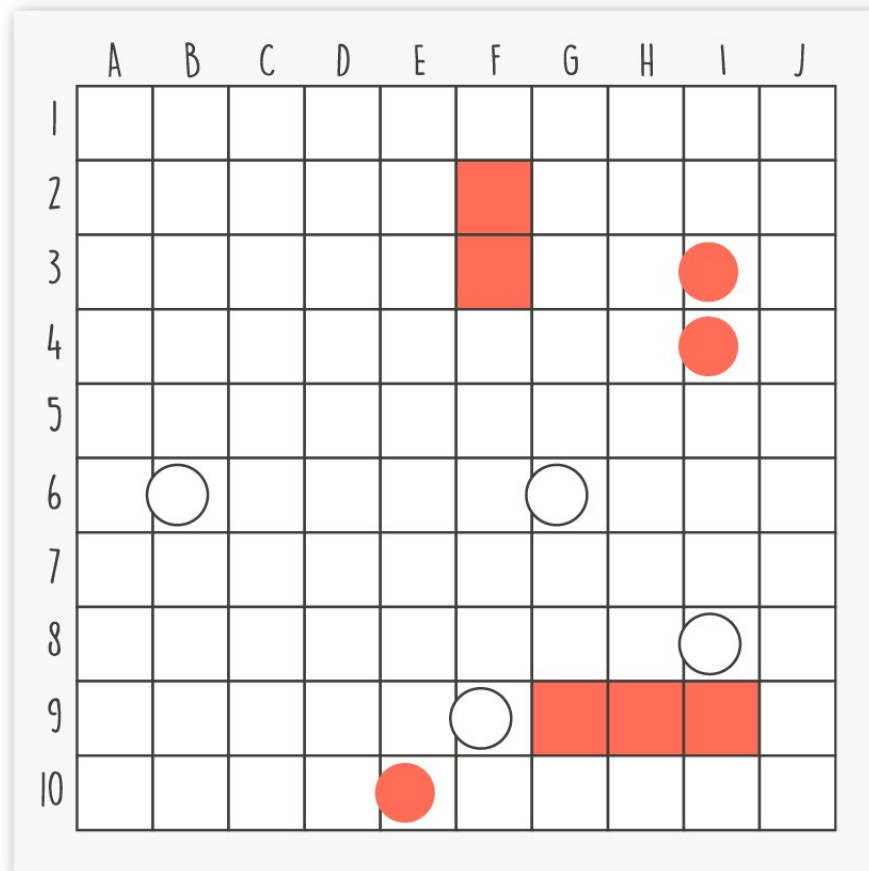
```
class Strategy {  
public:  
    virtual Coup choix(Grille &)=0;  
};
```

STRATEGY

```
Coup Strategy_0::choix(Grille &g) {  
    // n'importe quoi au hasard  
}
```

LA BATAILLE NAVALE EXEMPLE D'UNE GRILLE DE JEU

```
Coup Strategy_1::choix(Grille &g) {  
    // au hasard à un endroit jamais vu  
}
```



TIR RATÉ



BATEAU ENNEMI TOUCHÉ



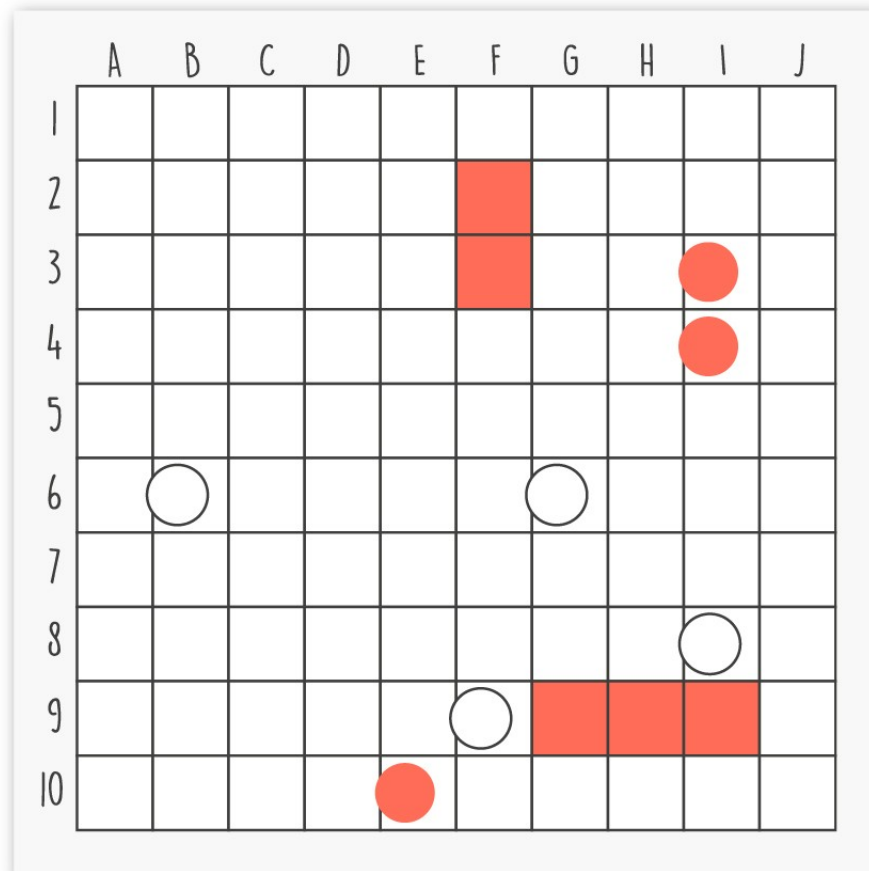
BATEAU ENNEMI COULÉ

*Je Suis
Animateur.fr*

```
class Strategy {  
public:  
    virtual Coup choix(Grille &)=0;  
};
```

STRATEGY

LA BATAILLE NAVALE EXEMPLE D'UNE GRILLE D



```
Coup Strategy_0::choix(Grille &g) {  
    // n'importe quoi au hasard  
}
```

```
Coup Strategy_1::choix(Grille &g) {  
    // au hasard à un endroit jamais vu  
}
```

```
Coup Strategy_2::choix(Grille &g) {  
    // une case sur 2 à partir de A,1  
}
```

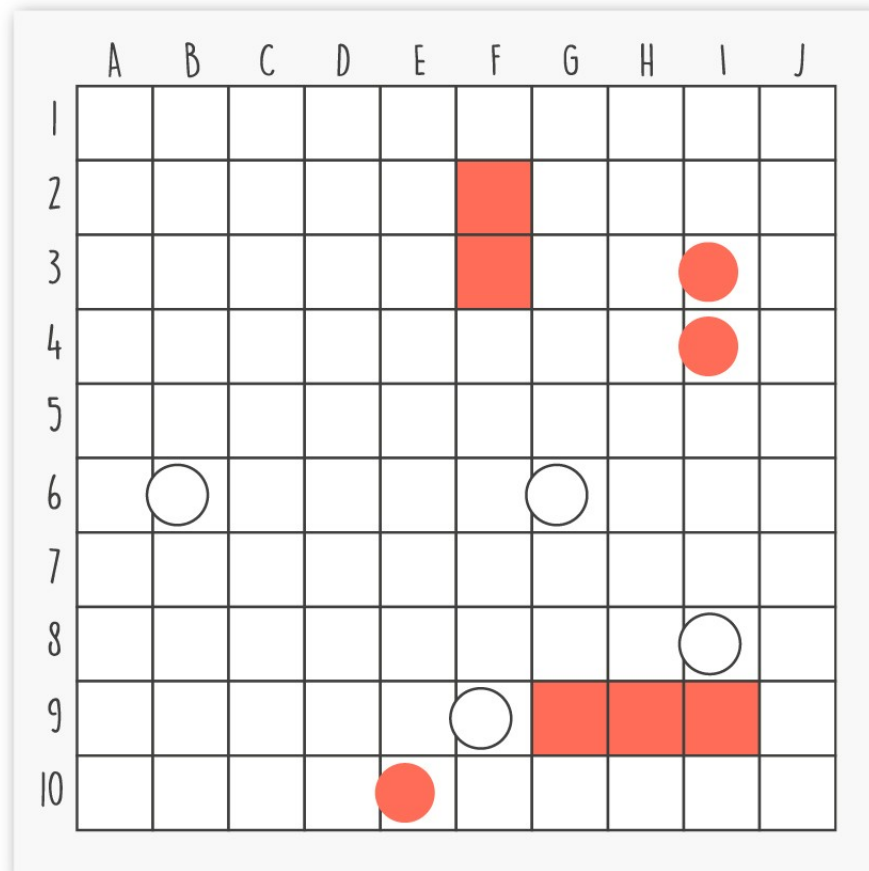
 BATEAU ENNEMI COULÉ

*Je Suis
Animateur.fr*

```
class Strategy {  
public:  
    virtual Coup choix(Grille &)=0;  
};
```

STRATEGY

LA BATAILLE NAVALE EXEMPLE D'UNE GRILLE DE JEU



TOUS DROITS RÉSERVÉS ©

```
Coup Strategy_0::choix(Grille &g){  
    // n'importe quoi au hasard  
}
```

```
Coup Strategy_1::choix(Grille &g){  
    // au hasard à un endroit jamais vu  
}
```

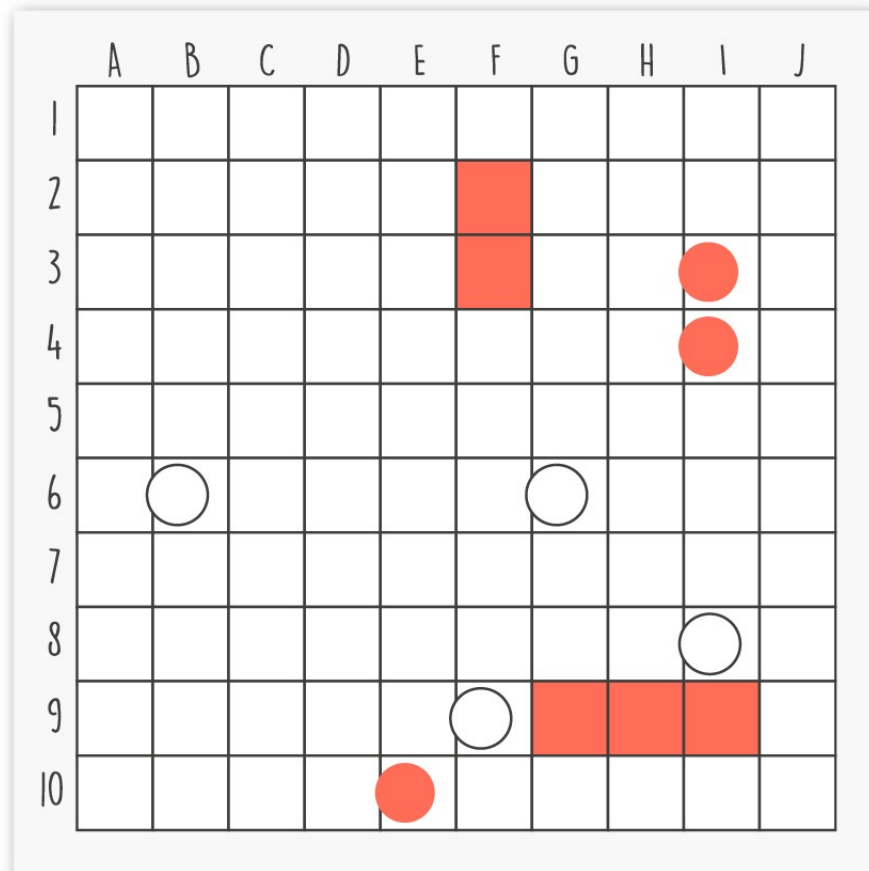
```
Coup Strategy_2::choix(Grille &g){  
    // une case sur 2 à partir de A,1  
}
```

```
Coup Strategy_3::choix(Grille &g){  
    // Tracer des lignes entre les coups  
    // précédent.  
    // Choisir alors la zone de plus  
    // grande surface  
    // Tirer proche de son milieu  
}
```

```
class Strategy {  
public:  
    virtual Coup choix(Grille &)=0;  
};
```

STRATEGY

LA BATAILLE NAVALE EXEMPLE D'UNE GRILLE DE



```
class Strategy {  
public:  
    virtual Coup choix(Grille  
};
```

```
Coup Strategy_0::choix(Grille &g) {  
    // n'importe quoi au hasard  
}
```

```
Coup Strategy_1::choix(Grille &g) {  
    // au hasard à un endroit jamais vu  
}
```

```
Coup Strategy_2::choix(Grille &g) {  
    // une case sur 2 à partir de A,1  
}
```

```
Coup Strategy_3::choix(Grille &g) {  
    // Tracer des lignes entre les coups  
    // précédent.  
    // Choisir alors la zone de plus  
    // grande surface  
    // Tirer proche de son milieu  
}
```

```
Coup Strategy_4::choix(Grille &g) {  
    // Tirer dans le voisinage d'un bateau  
    // touché s'il y en a  
    // utiliser Strategy_x sinon  
    // ... un peu « à la » proxy ?  
    // ou « à la » chain of responsibility  
}
```

STRATEGY

Un context peut fixer ou changer la stratégie :

- choix initial du niveau du joueur
- réaction : on peut commencer par stratégie1 puis passer à stratégie4 si on a eu un tir intéressant, et revenir à une autre stratégie. (On voit ici un mélange avec le pattern Etat)

