ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2018-2019

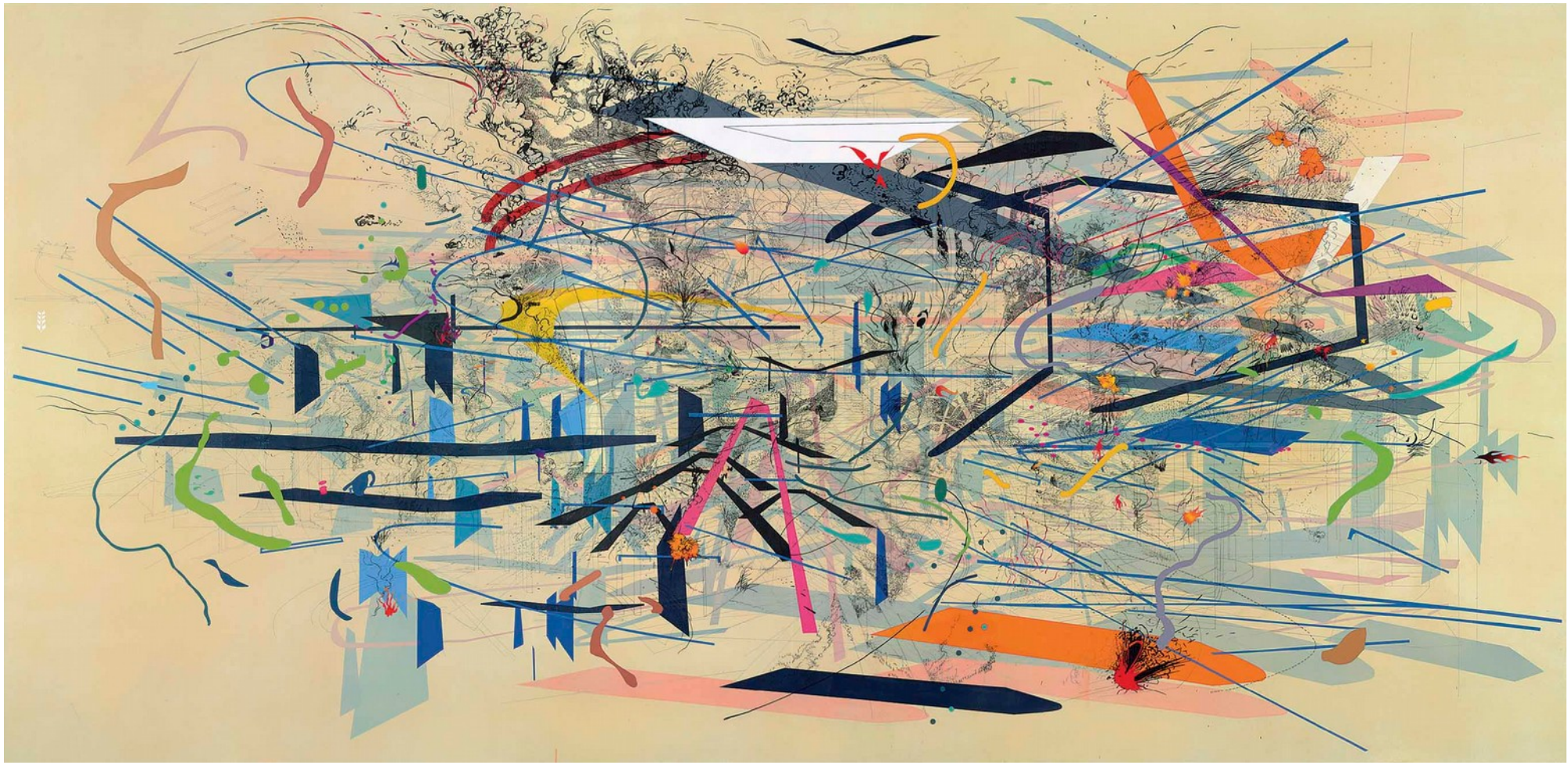
Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. Abstraction / design patterns		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. *Templates côté développeur*		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

Abstraction / design patterns



Retopistics: A Renegade Excavation, Julie Mehretu

COURS 9

A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

COURS 9

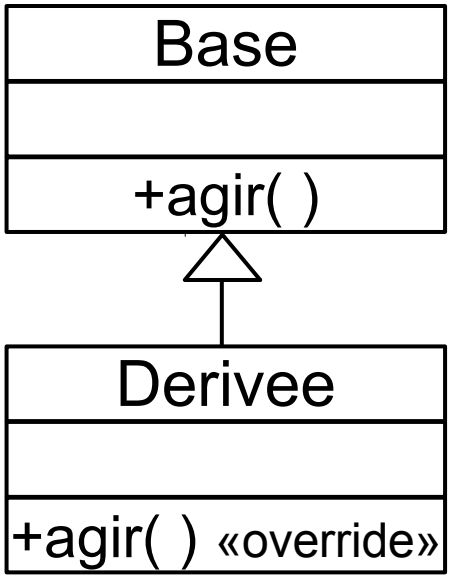
A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

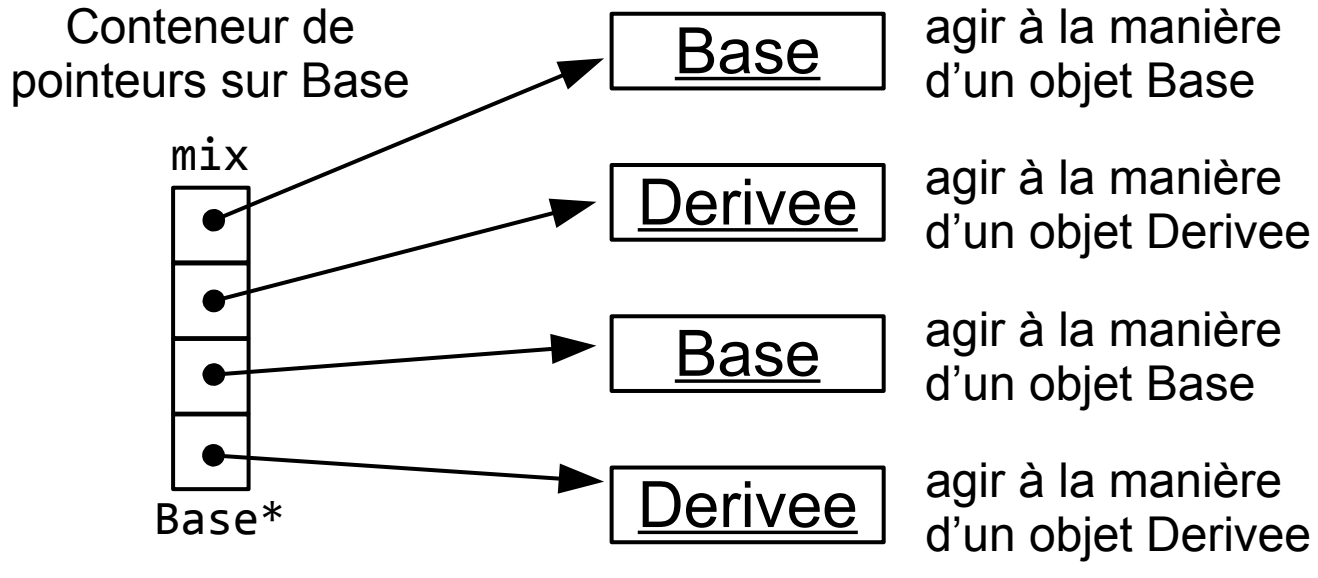
Classes abstraites / interfaces

Composition suprématiste, Kasimir Malevitch

Classes abstraites / interfaces

- *Le **polymorphisme** (Rappel du dernier cours)*
  - *Héritage : classe de Base, classe Dérivée*
  - *Dérivée redéfinit méthode(s) classe Base*
  - *Objets manipulés par pointeurs/référence Base*
  - *Méthodes classe Base déclarées* **`virtual`**





Classes abstraites / interfaces

base.h

```
class Base
{
    public :
        virtual ~Base() = default;
        virtual void agir();
};
```

base.cpp

```
void Base::agir()
{
    std::cout << "J'agis d'une certaine maniere\n";
}
```

derivee.h

```
class Derivee : public Base
{
    public :
        virtual void agir();
};
```

derivee.cpp

```
void Derivee::agir()
{
    std::cout << "J'agis d'une autre maniere\n";
}
```

Classes abstraites / interfaces

base.h

```
class Base
{
    public :
        virtual ~Base() = default;
        virtual void agir();
};
```

Obligatoire : destructeur virtuel

Méthode à redéfinir : virtuelle

base.cpp

```
void Base::agir()
{
    std::cout << "J'agis d'une certaine maniere\n";
}
```

derivee.h

```
class Derivee : public Base
{
    public :
        virtual void agir();
};
```

Méthode redéfinie !

derivee.cpp

```
void Derivee::agir()
{
    std::cout << "J'agis d'une autre maniere\n";
}
```

Classes abstraites / interfaces

- *Le **polymorphisme** permet de traiter "à égalité" tous les objets d'une **hiérarchie d'héritage** avec une classe de base en commun, tout en conservant les spécificité de chacun...*

main.cpp

```
int main()
{
    std::list<Base*> mix;

    mix.push_back(new Base);
    mix.push_back(new Derivee);
    mix.push_back(new Base);
    mix.push_back(new Derivee);

    for (auto ptobj : mix)
        ptobj->agir();

    for (auto ptobj : mix)
        delete ptobj;

    return 0;
}
```

action polymorphe

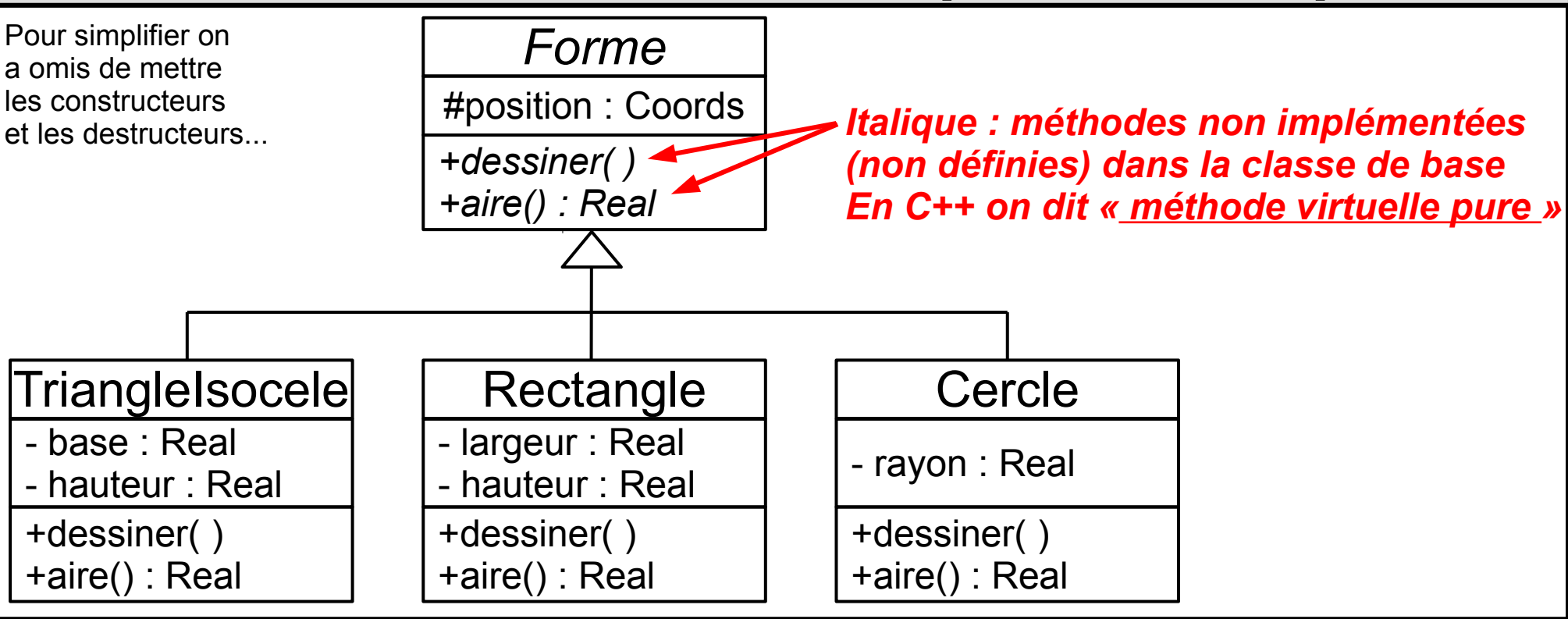
```
J'agis d'une certaine maniere
J'agis d'une autre maniere
J'agis d'une certaine maniere
J'agis d'une autre maniere
```

Ici destruction polymorphe (appel au bon destructeur)

Classes abstraites / interfaces

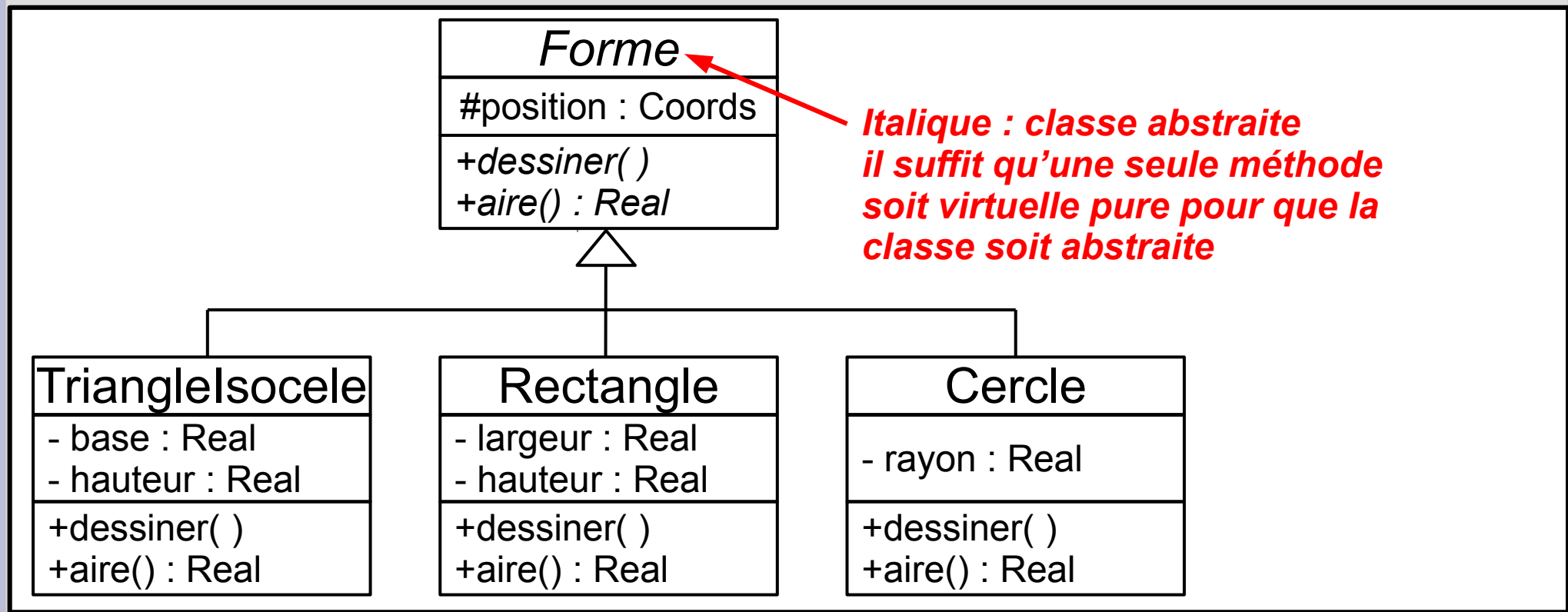
- *Souvent le polymorphisme des classes dérivées est l'objectif principal de l'utilisation de l'héritage*
- *Dans ce cas implémenter certaines/toutes les méthodes de la classe de base peut ne pas avoir de sens : **on ne les implémentera pas !***

Pour simplifier on
a omis de mettre
les constructeurs
et les destructeurs...



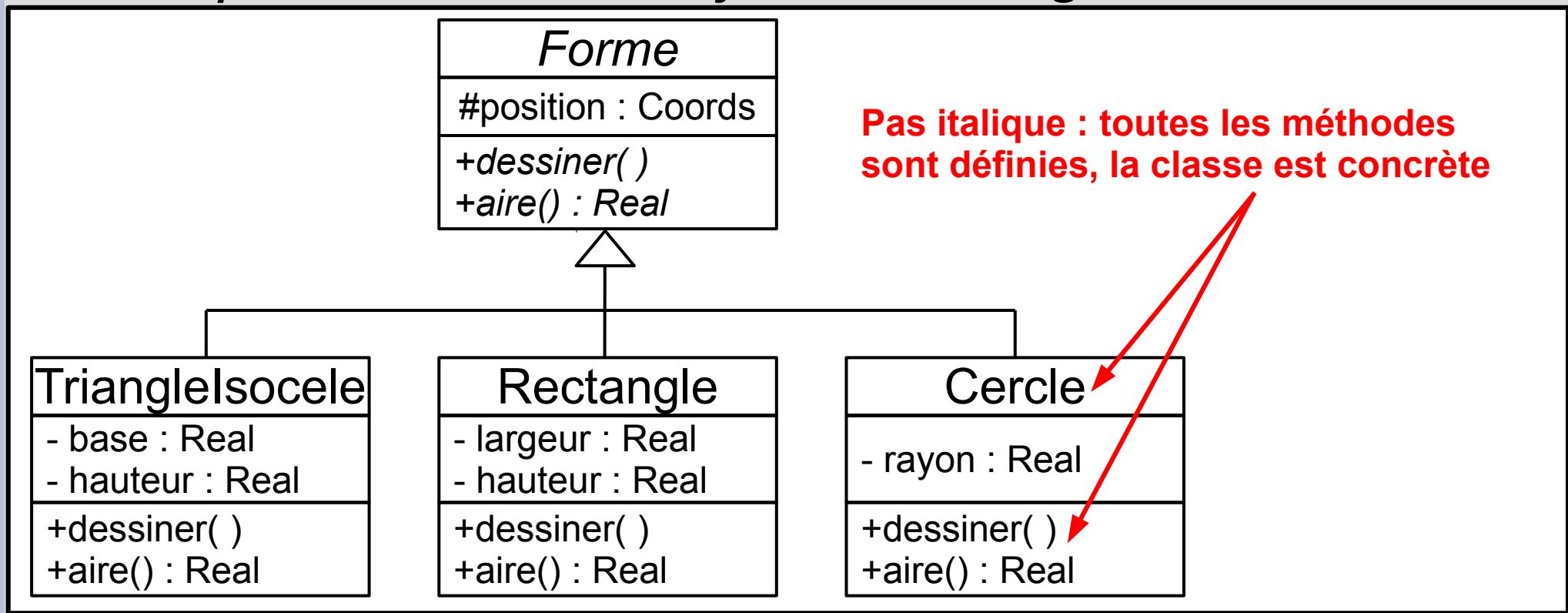
Classes abstraites / interfaces

- *Certaines méthodes de la classe de base ne sont pas définies : celle-ci est **non instanciable***
- *On dit que la classe est **abstraite** : aucun objet de type « Forme » n'est possible*



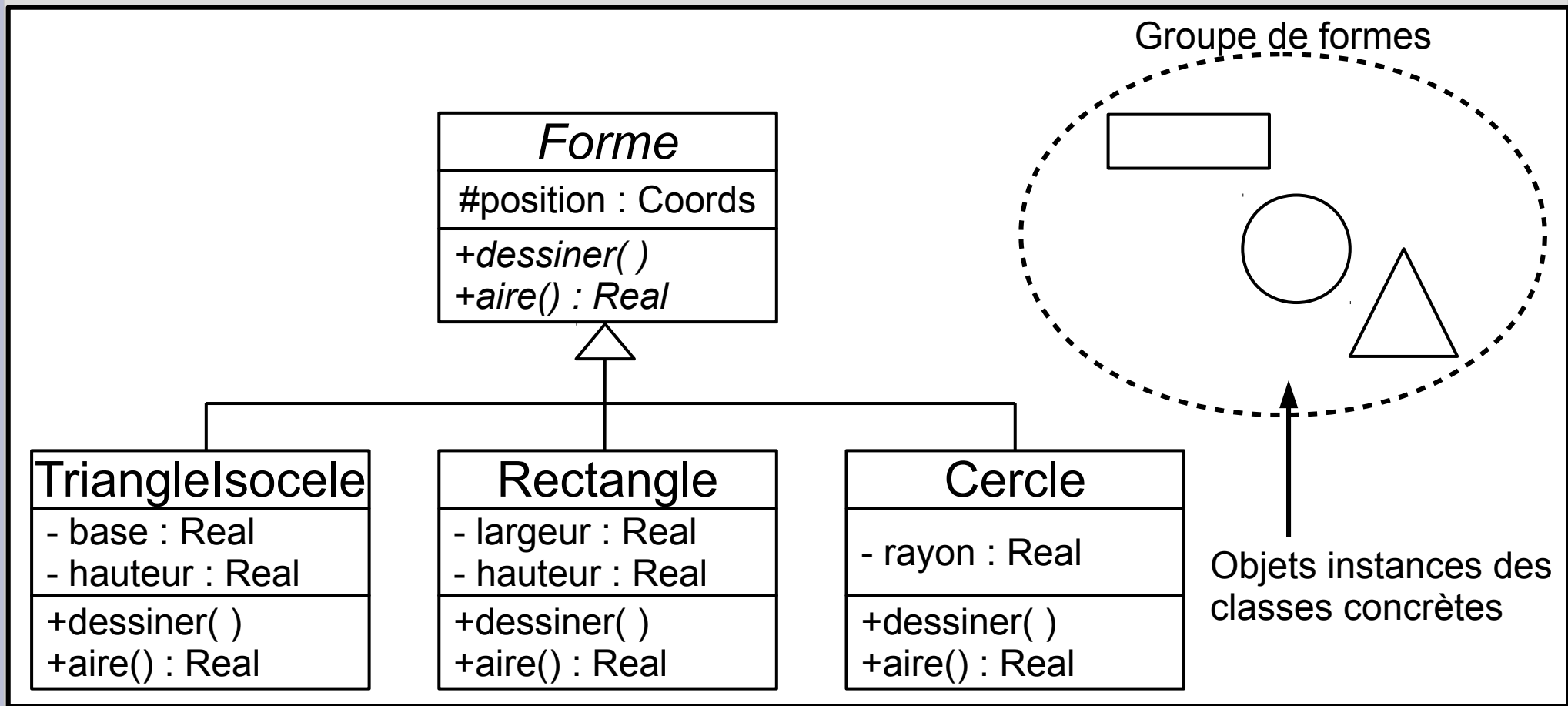
Classes abstraites / interfaces

- ***Toutes*** *les méthodes virtuelles pures de la classe de base abstraite doivent être définies par une classe dérivée pour l'instancier...*
- *On dit que la classe dérivée est* ***concrète*** *: on peut avoir des objets Rectangle, Cercle...*



Classes abstraites / interfaces

- *De nombreuses utilisations, la plus évidente : grouper des objets « similaires en principe » mais différents en pratique (traitements spécifiques pour des opérations communes)*



Classes abstraites / interfaces

- *Les méthodes **virtuelles pures** s'indiquent avec **=0** en fin de déclaration de la méthode.*
- *Le classe Forme est non instanciable mais elle peut fournir un constructeur (pour les dérivées)*

forme.h

```
class Forme
{
    public :
        Forme(Coords position);
        virtual ~Forme() = default;
        virtual void dessiner(Svgfile& svgout) = 0;
        virtual double aire() = 0;

    protected :
        Coords m_position;
};
```

forme.cpp

```
Forme::Forme(Coords position)
    : m_position{position}
{ }
```

Classes abstraites / interfaces

- *Les méthodes **virtuelles pures** s'indiquent avec **=0** en fin de déclaration de la méthode.*
- *Le classe Forme est non instanciable mais elle peut fournir un constructeur (pour les dérivées)*

forme.h

```
class Forme
{
    public
        Forme(Coords position);
        virtual ~Forme() = default;
        virtual void dessiner(Svgfile& svgout) = 0;
        virtual double aire() = 0;

    protected :
        Coords m_position;
};
```

Un construteur n'est jamais virtuel

Polymorphisme => destructeur virtuel (si possible default)

Ces 2 méthodes sont virtuelles pures.
Au moins une méthode est virtuelle pure donc la classe est automatiquement abstraite

forme.cpp

```
Forme::Forme(Coords position)
    : m_position{position}
{ }
```

Classes abstraites / interfaces

- Une classe concrète hérite de la classe abstraite
- Elle définit toutes les méthodes virtuelles de la classe de abstraite de base
- Elle peut ajouter des attributs, un constructeur...

rectangle.h

```
class Rectangle : public Forme
{
    public :
        Rectangle(Coords position, double largeur, double hauteur);
        virtual void dessiner(Svgfile& svgout);
        virtual double aire();

    private :
        double m_largeur;
        double m_hauteur;
};
```

```
Rectangle::Rectangle(Coords position, double largeur, double hauteur)
    : Forme{position}, m_largeur{largeur}, m_hauteur{hauteur}
{ }
```

rectangle.cpp

Classes abstraites / interfaces

- Une classe concrète hérite de la classe abstraite
- Elle définit toutes les méthodes virtuelles de la classe de abstraite de base
- Elle peut ajouter des attributs, un constructeur...

rectangle.h

```
class Rectangle : public Forme
{
    public :
        Rectangle(Coords position, double largeur, double hauteur);
        virtual void dessiner(Svgfile& svgout);
        virtual double aire();

    private :
        double m_largeur;
        double m_hauteur;
};
```

Facultatif (ces méthodes sont de toute façon virtuelles)

Si possible pas de destructeur (destructeur implicite ok)

rectangle.cpp

```
Rectangle::Rectangle(Coords position, double largeur, double hauteur)
    : Forme{position}, m_largeur{largeur}, m_hauteur{hauteur}
{ }
```

Appel au constructeur de la classe de base (classe de base non instanciable directement)

Classes abstraites / interfaces

- Une classe concrète hérite de la classe abstraite
- Elle définit toutes les méthodes virtuelles de la classe de abstraite de base
- Elle peut ajouter des attributs, un constructeur...
- Tout ceci est **spécifique** à cette classe concrète

rectangle.cpp

```
void Rectangle::dessiner(Svgfile& svgout)
{
    svgout.addRect(m_position.getX()-m_largeur/2,
                   m_position.getY()-m_hauteur/2,
                   m_position.getX()+m_largeur/2,
                   m_position.getY()+m_hauteur/2);
}

double Rectangle::aire()
{
    return m_largeur * m_hauteur;
}
```

Classes abstraites / interfaces

- ***Utiliser les classe concrètes sans les connaître !***

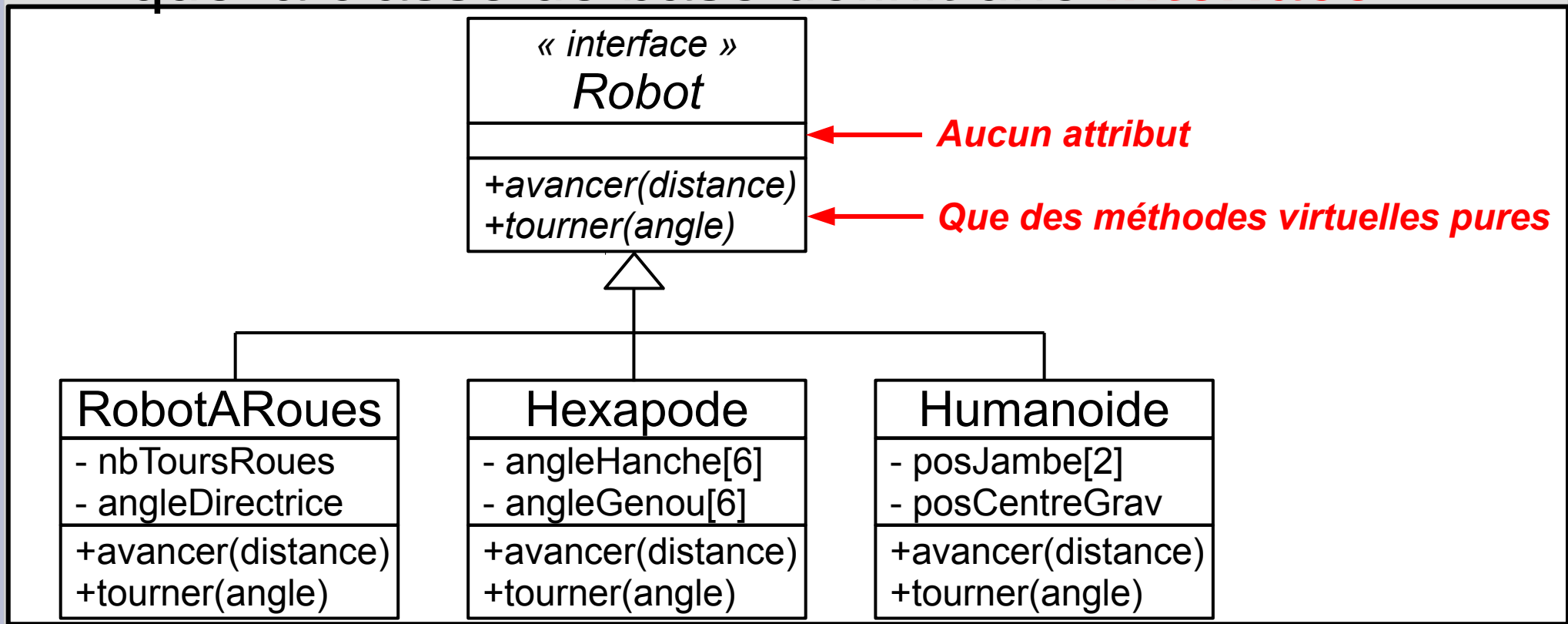
main.cpp

```
double aireTotale(std::vector<Forme*> formes)
{
    double total = 0;
    for (auto ptforme : formes)
        total += ptforme->aire();
    return total;
}

int main()
{
    std::vector<Forme*> groupe;

    groupe.push_back(new Rectangle{{100,100}, 150, 50});
    groupe.push_back(new Circle{{200,200}, 50});
    groupe.push_back(new TriangleIsocele{{300,300}, 100, 100});

    for (auto ptforme : groupe)
        ptforme->dessiner(svgout);

    std::cout << aireTotale(groupe) << std::endl;

    for (auto ptforme : groupe)
        delete ptforme;
```

Code polymorphe

Classes abstraites / interfaces

- Il y a souvent intérêt à abstraire complètement la classe de base : elle ne propose **plus aucune méthode concrète ni aucun attribut !**
- On dit que la classe est **abstraite pure** ou que la classe de base définit une **interface**



Classes abstraites / interfaces

Classe interface — ***robot.h***

Pas de constructeur. Pas d'attribut. Que des méthodes virtuelles pures (sauf le destructeur)

```
class Robot
{
    public :
        virtual ~Robot() = default;
        virtual void avancer(double distance) = 0;
        virtual void tourner(double angle) = 0;
};
```

Classe concrète implémente l'interface — ***robot_a_roues.h***

```
class RobotARoues : public Robot
{
    public :
        RobotARoues();
        virtual void avancer(double distance);
        virtual void tourner(double angle);

    private :
        int m_nbToursRoues;
        double m_angleDirectrice;
};
```

robot_a_roues.cpp

```
RobotARoues::RobotARoues() : m_nbToursRoues{0}, m_angleDirectrice{0} { }

void RobotARoues::avancer(double distance) {
    m_nbToursRoues += distance/3.14;
}
void RobotARoues::tourner(double angle) {
    m_angleDirectrice = angle;
}
```

COURS 9

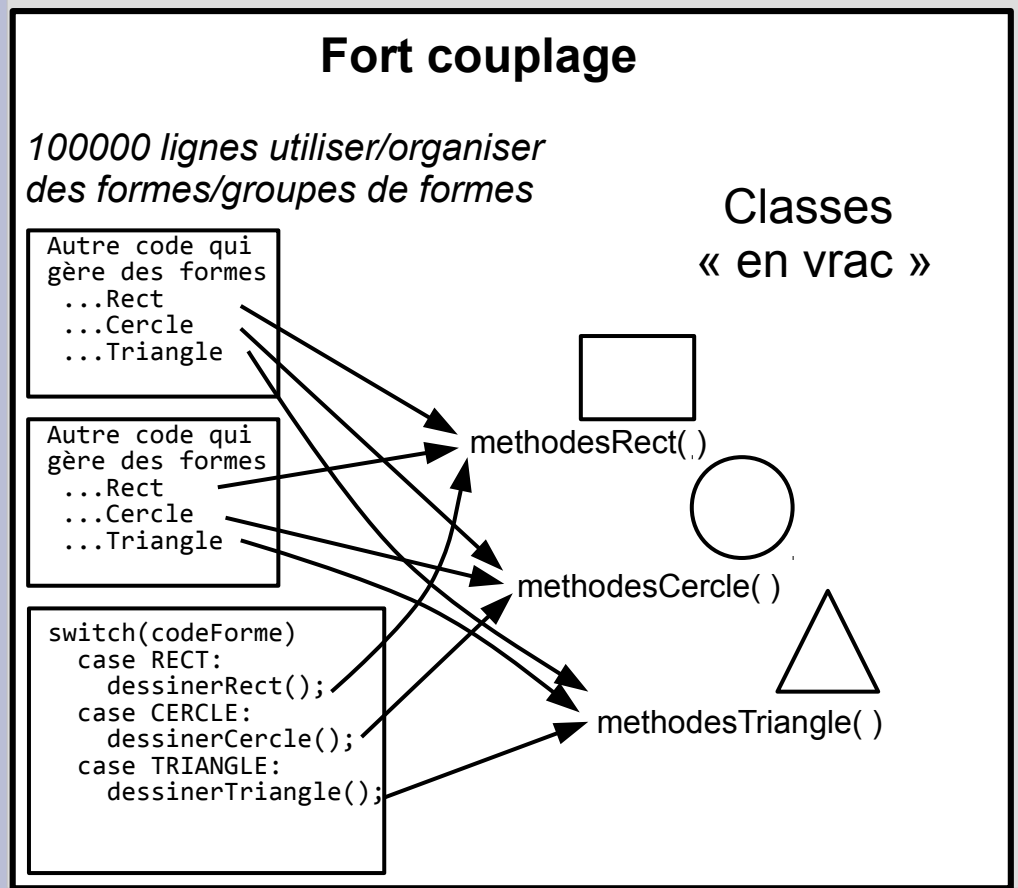
A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

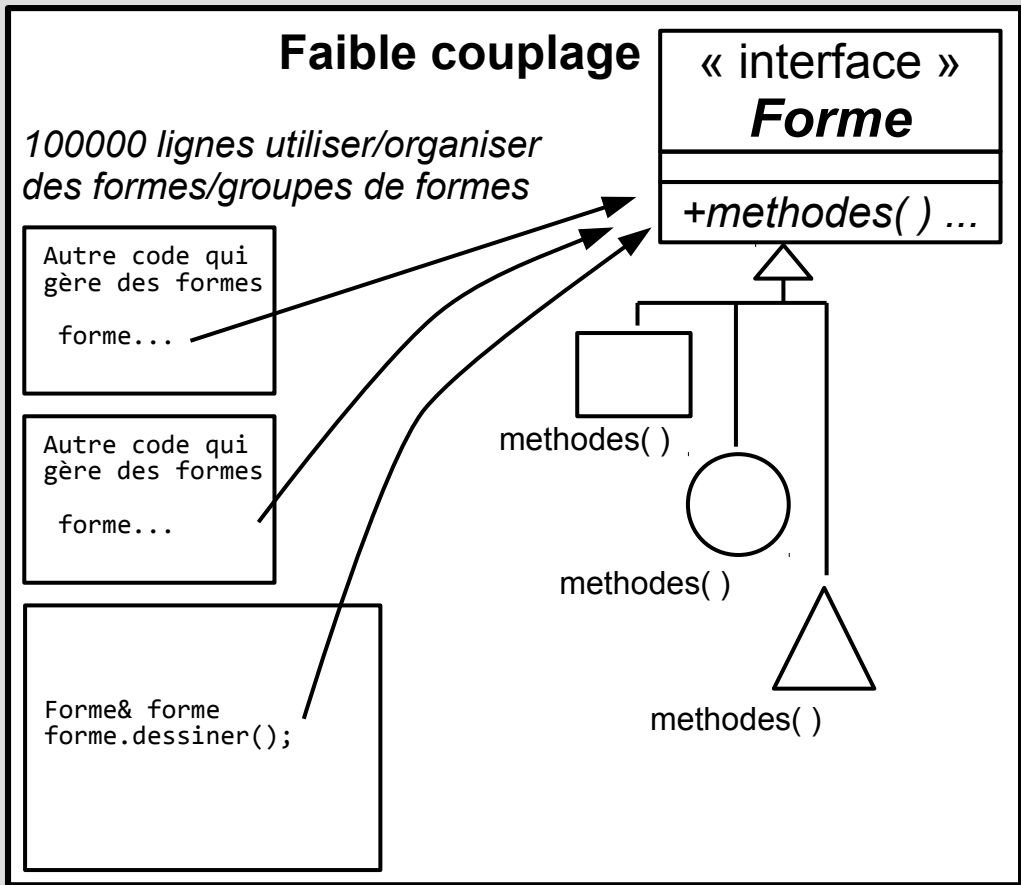
Couplage / inversion du contrôle

Blue Beams, Judith Godwin

Couplage / inversion du contrôle

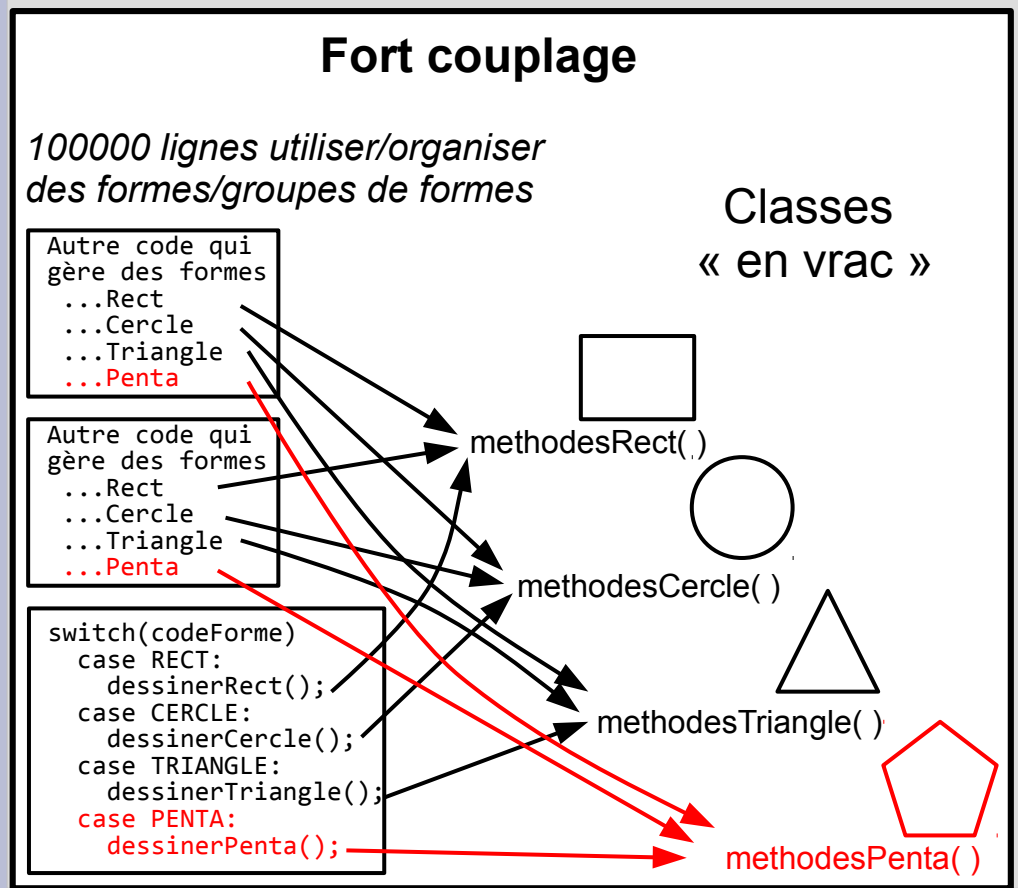
- L'utilisation de classes abstraites ou **interfaces** va plus loin que les conteneurs polymorphes...
- Elle permet de diminuer le **couplage** entre composants appelants et composants appelés

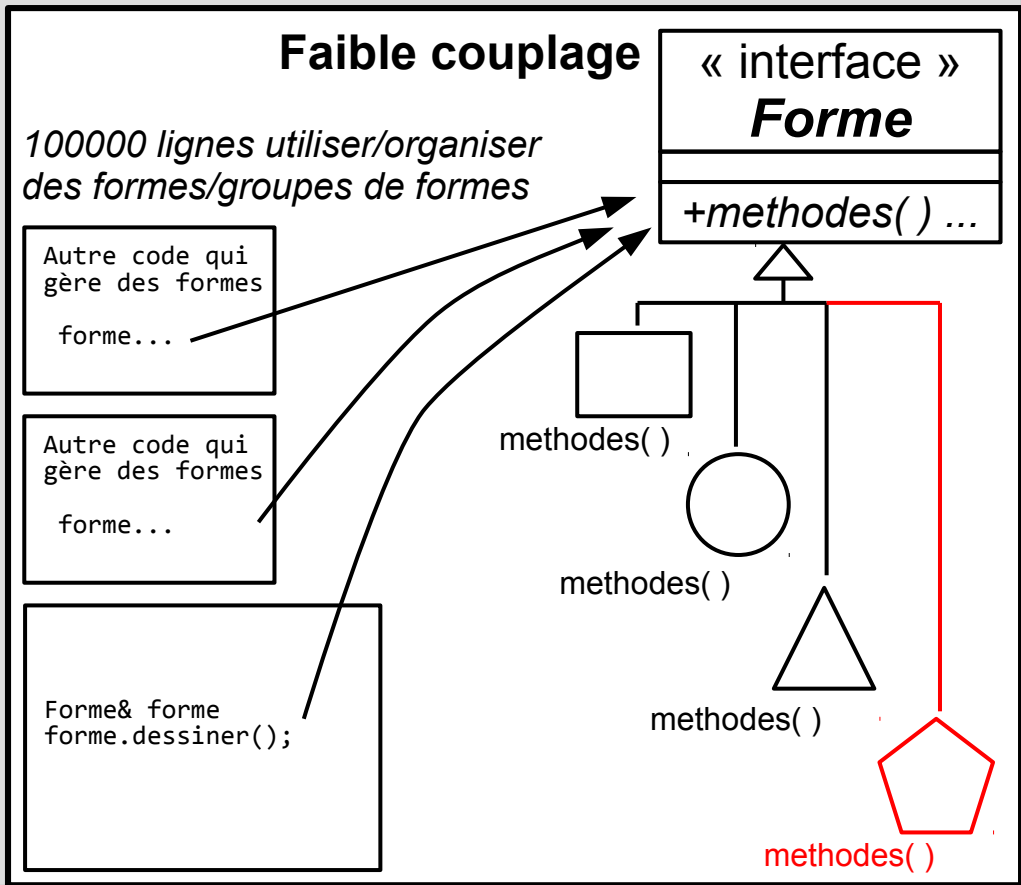




Couplage / inversion du contrôle

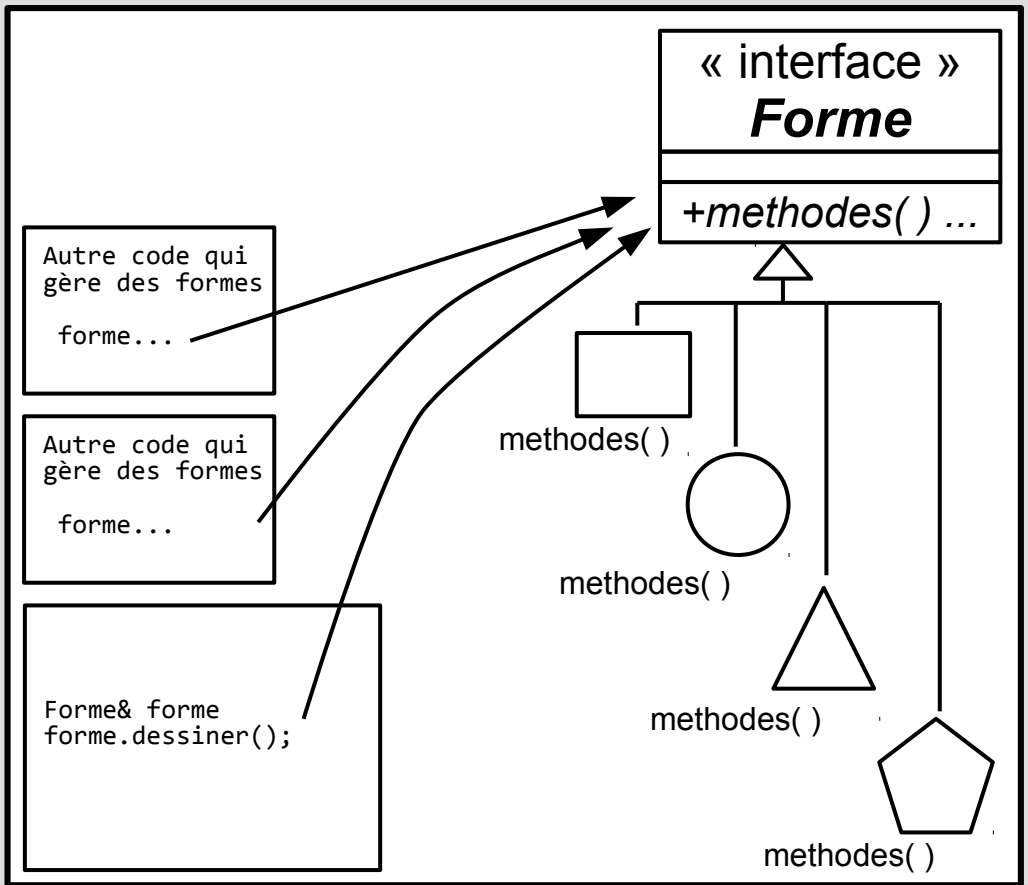
- Un couplage fort oblige à reprendre le code utilisateur quand les classes sont **étendues**
- ➔ Avec une interface, ajouter une nouvelle classe nécessite juste de définir ses méthodes !





Couplage / inversion du contrôle

- L'interface constitue en quelque sorte un « guichet unique » pour toutes les démarches concernant une catégorie de classes !
- ➔ L'interface doit être **stable**, c'est une articulation

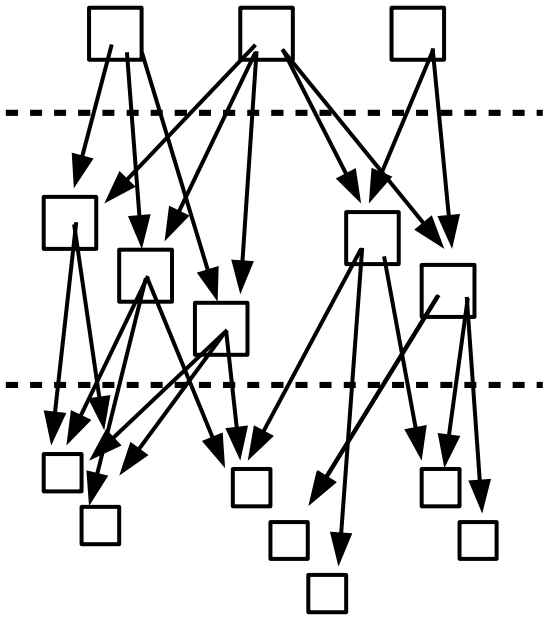


Couplage / inversion du contrôle

- Principe de conception : *dependency inversion*
- ➔ *Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau.*
  Tous doivent dépendre des abstractions.

Depend upon abstractions. Do not depend upon concrete classes.

Fort couplage

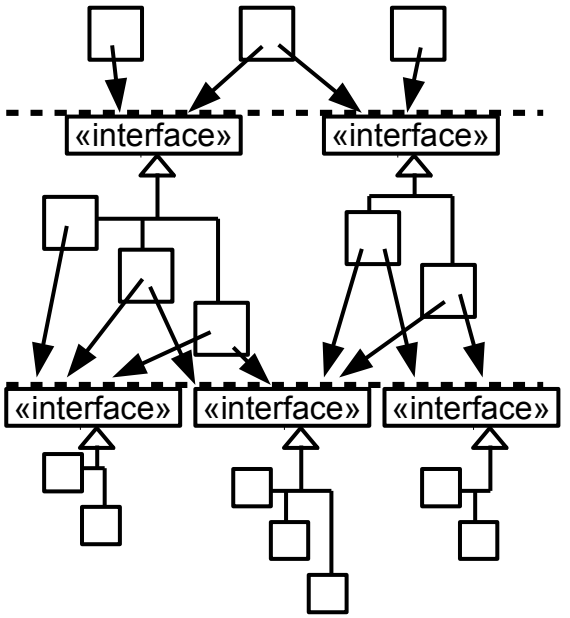


Classes de niveau application
Types Document, Jeu, Systeme...

Composants intermédiaires
Types entités, composites...

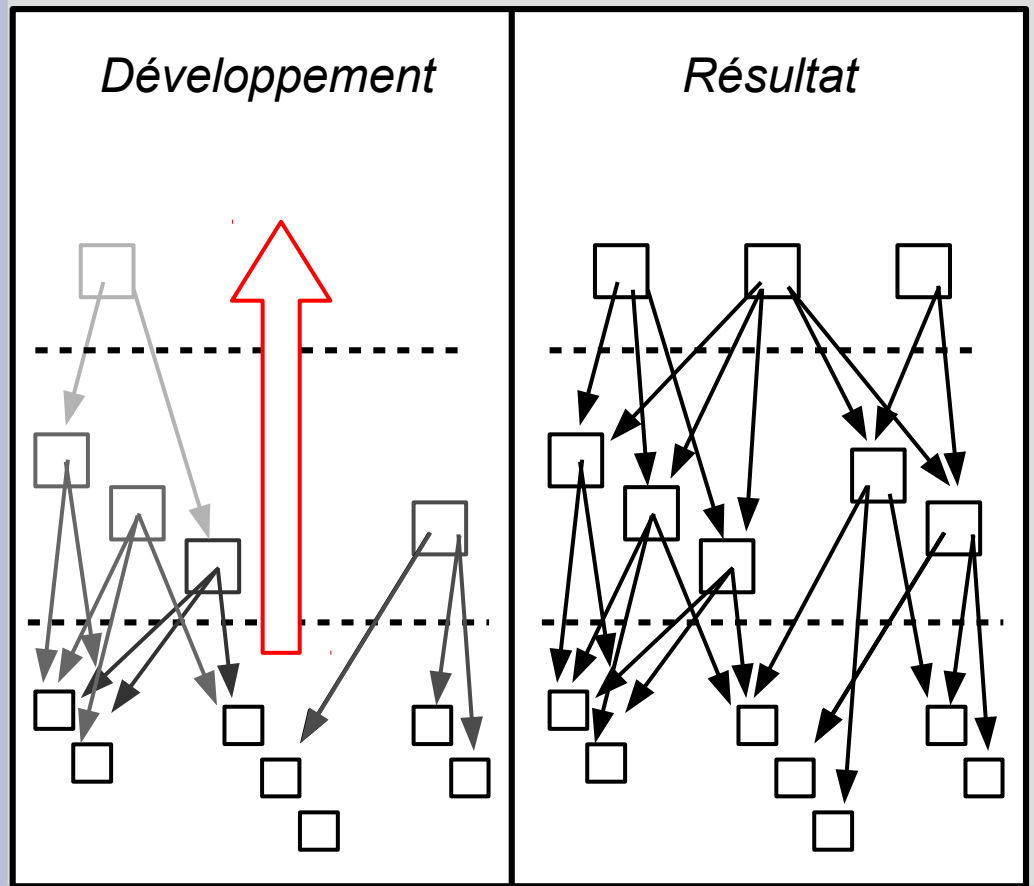
Composants de bas niveau,
Types utilitaires, types valeurs...

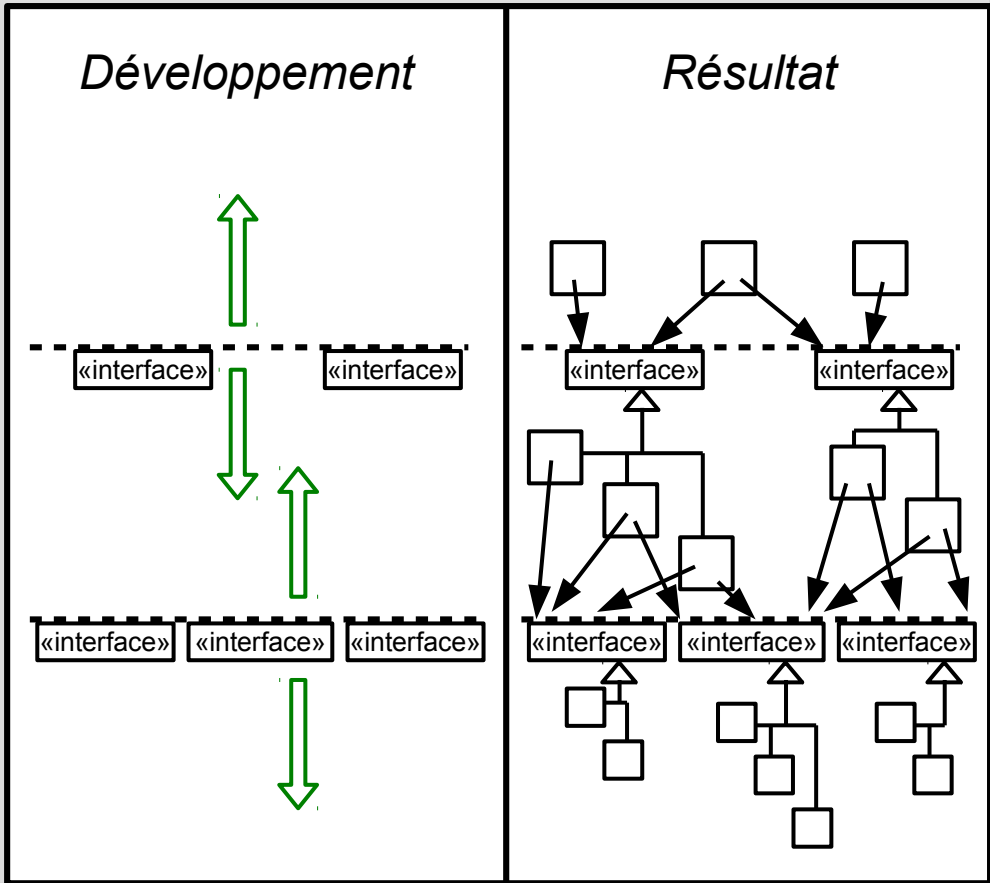
Faible couplage



Couplage / inversion du contrôle

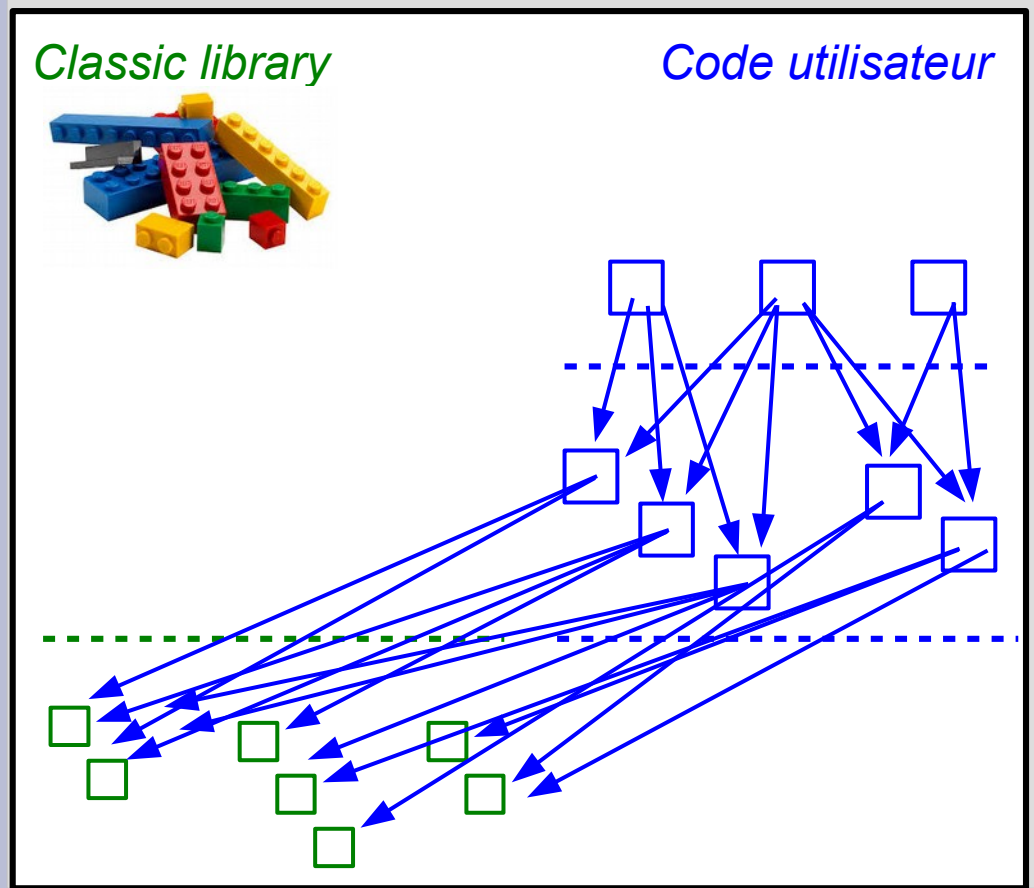
- Plutôt que coder l'application « de bas en haut » on code l'application de l'abstrait vers le concret
- ➔ *Par exemple un Maillage pourrait être indifférent au fait de manipuler des Sommets en 2D ou 3D*

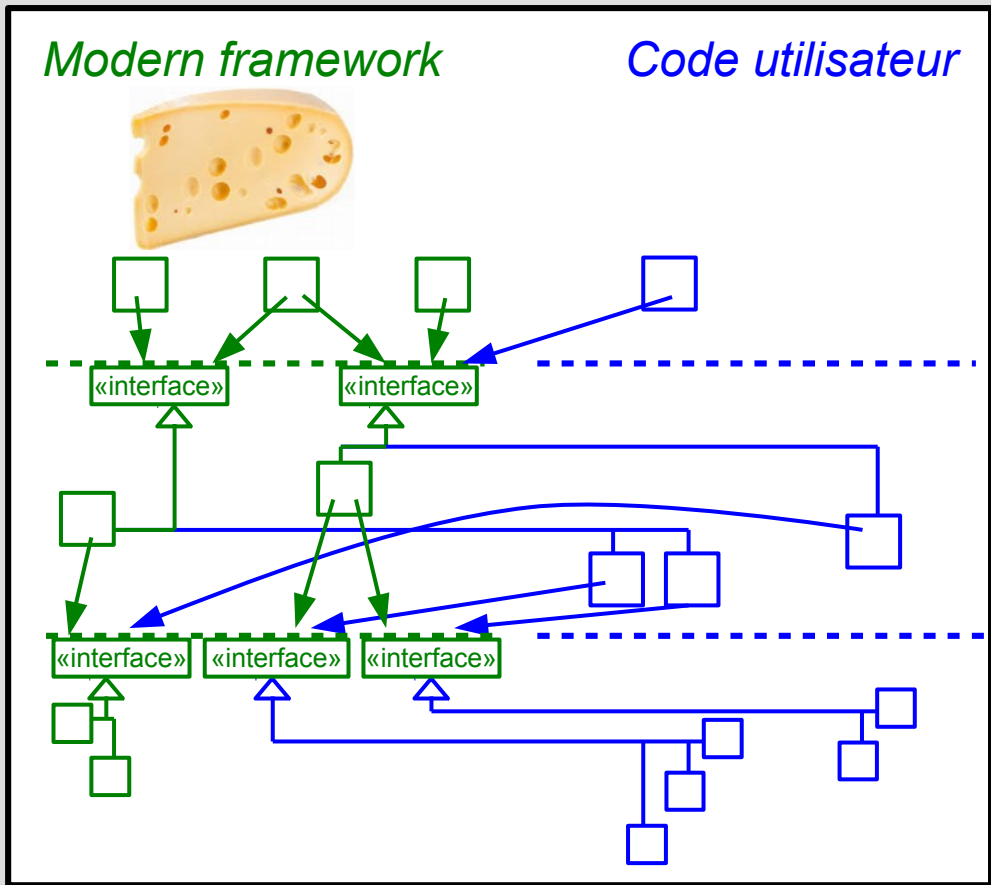




Couplage / inversion du contrôle

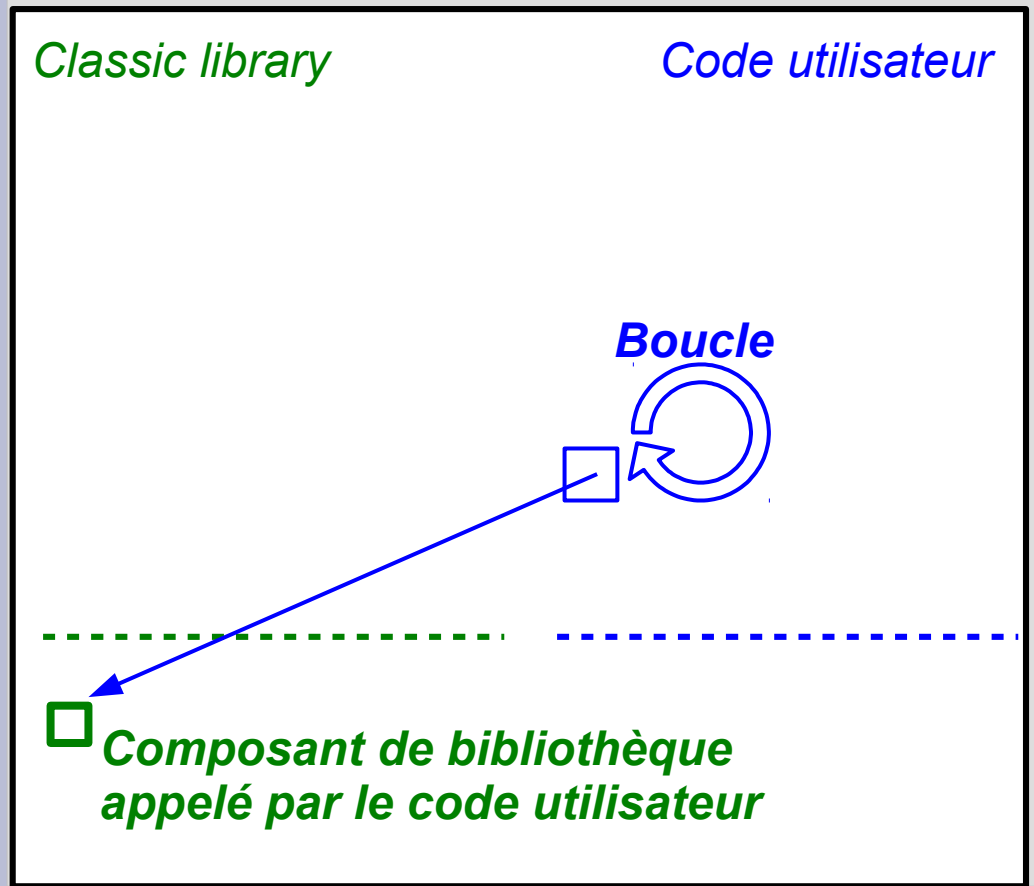
- Ce qui donne la forme **moderne** de bibliothèque
- ➔ Le ***framework*** par opposition à la ***library*** est une base de code qui propose avant tout des abstractions et qui **contrôle le code utilisateur**

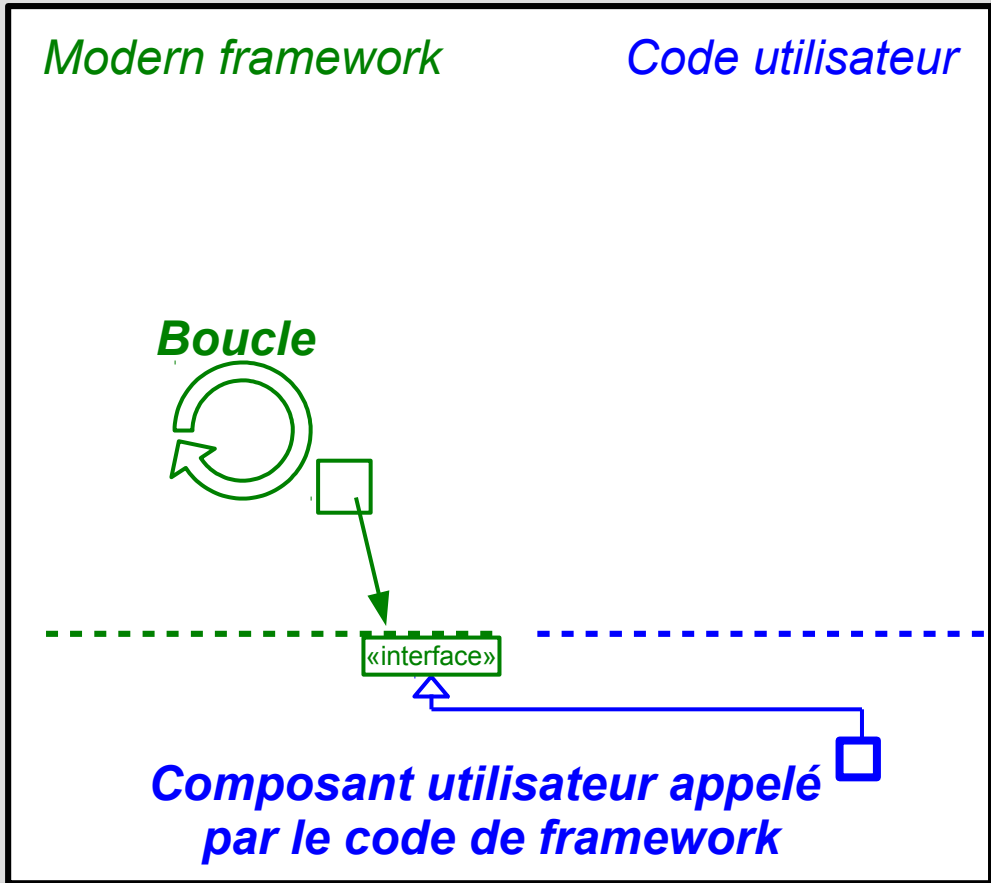




Couplage / inversion du contrôle

- Inversion de contrôle (*inversion of control*) :
  « *le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework* »





Couplage / inversion du contrôle

- Inversion de contrôle (*inversion of control*)
- Différentes techniques « objets » pour réaliser l'inversion de contrôle. Hériter d'une interface du framework est une des façons...

```
/// Classe interface (Abstraite pure)
class Fonction
{
  public :
    virtual double evaluer(double x)=0;
};

/// Intégration méthode du point milieu
double integrer(Fonction& f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f.evaluer(x) * pas;
  return somme;
}
```

```
/// Code utilisateur
class Fracrat : public Fonction
{
  public :
    double evaluer(double x);
};

double Fracrat::evaluer(double x)
{
  return 1/(1+x*x);
}

int main()
{
  Fracrat fr;
  std::cout<<4.0*integrer(fr,
                0, 1,
                0.001) << std::endl;
```

```
3.14159
```

Couplage / inversion du contrôle

- Inversion de contrôle (*inversion of control*)
- Différentes techniques « objets » pour réaliser l'inversion de contrôle. Hériter d'une interface du framework est une des façons...

```
/// Classe interface (Abstraite pure)
class Fonction
{
  public :
    virtual double evaluer(double x)=0;
};

/// Intégration méthode du point milieu
double integrer(Fonction& f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f.evaluer(x) * pas;
  return somme;
}
```

Polymorphisme...

Appel par l'interface

```
/// Code utilisateur
class Fracrat : public Fonction
{Classe concrète hérite interface
  public :
    double evaluer(double x);
};

double Fracrat::evaluer(double x)
{
  return 1/(1+x*x);
}

int main()
{
  Fracrat fr;
  std::cout<<4.0*integrer(fr,
                0, 1,
                0.001) << std::endl;
```

Implémentation !

3.14159

Couplage / inversion du contrôle

$$\pi = 4\left(\arctan 1 - \arctan 0\right) = 4\left[\arctan x\right]_0^1$$

$$= 4\int_0^1 \frac{1}{1+x^2} \approx 4 \sum_{x=0.0005}^{0.9995\ step\ 0.001} 0.001\,\frac{1}{1+x^2}$$

```
/// Classe interface (Abstraite pure)
class Fonction
{
  public :
    virtual double evaluer(double x)=0;
};

/// Intégration méthode du point milieu
double integrer(Fonction& f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f.evaluer(x) * pas;
  return somme;
}
```

```
/// Code utilisateur
class Fracrat : public Fonction
{
  public :
    double evaluer(double x);
};

double Fracrat::evaluer(double x)
{
  return 1/(1+x*x);
}

int main()
{
  Fracrat fr;
  std::cout<<4.0*integrer(fr,
                0, 1,
                0.001) << std::endl;
```

```
3.14159
```

COURS 9

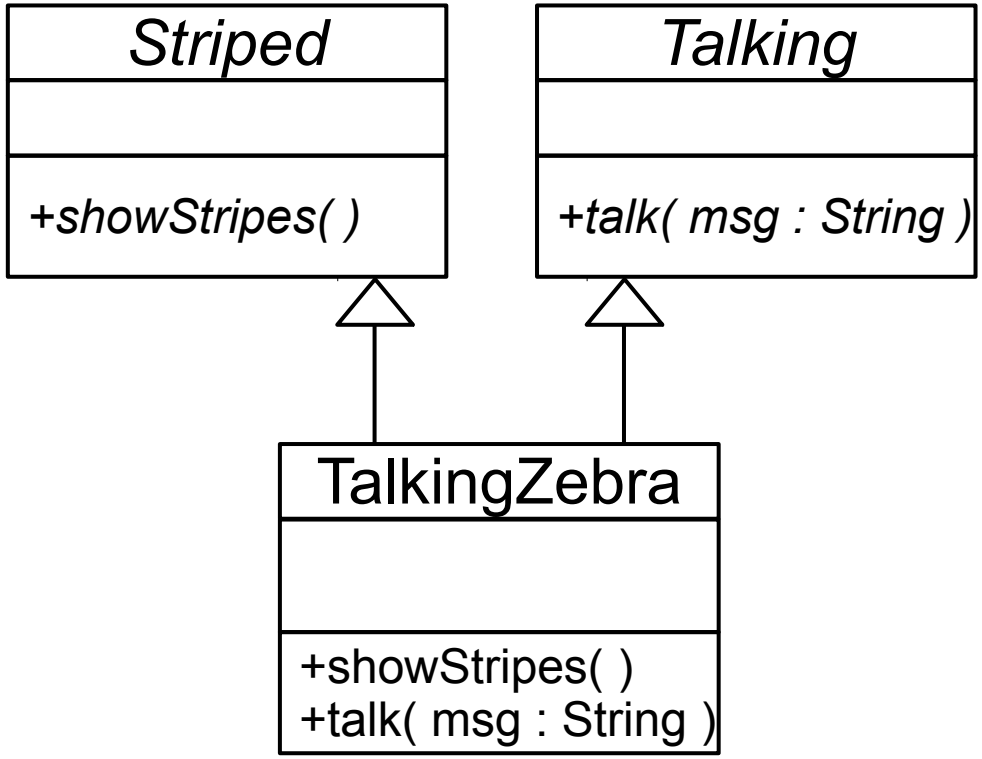
A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

Héritage multiple

African sonata, Vladimir Kush

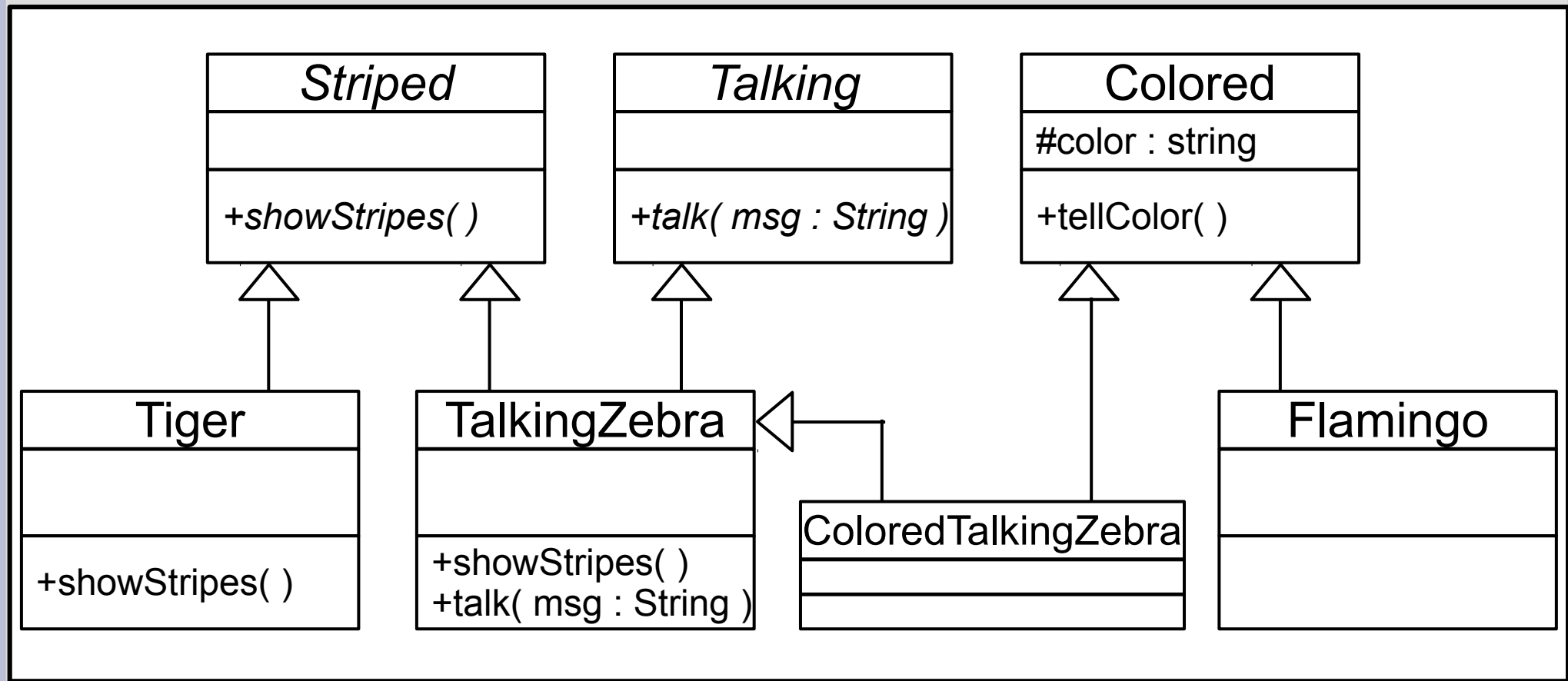
Héritage multiple

- L'héritage simple implique pour une classe dérivée d'avoir une seule classe de base
- L'héritage multiple implique pour une classe dérivée d'avoir plusieurs classes de base



Héritage multiple

- L'héritage multiple est souvent utile pour hériter des **capacités** de classes abstraites/interfaces, mais pas exclusivement (ici Colored est concrète)
- Héritage simple et multiple sont compatibles...



Héritage multiple

- Les classes de base de l'exemple...

striped.h

```
class Striped
{
    public :
        virtual void showStripes() = 0;
};
```

talking.h

```
class Talking
{
    public :
        virtual void talk(std::string msg) = 0;
};
```

colored.h

```
class Colored
{
    public :
        Colored(std::string color);
        virtual void tellColor();

    protected :
        std::string m_color;
};
```

colored.cpp

```
Colored::Colored(std::string color)
    : m_color{color}  { }

void Colored::tellColor() {
    std::cout << m_color << std::endl;
}
```

Héritage multiple

- À la déclaration de la classe dérivée, les classes de base sont indiquées **dans une liste**
- ➔ *Cette hiérarchie fera l'objet d'un exo TD/TP 9 ...*

talking_zebra.h

Liste des classes de base

```
class TalkingZebra : public Striped, public Talking
{
    virtual void showStripes();
    virtual void talk(std::string msg);
};
```

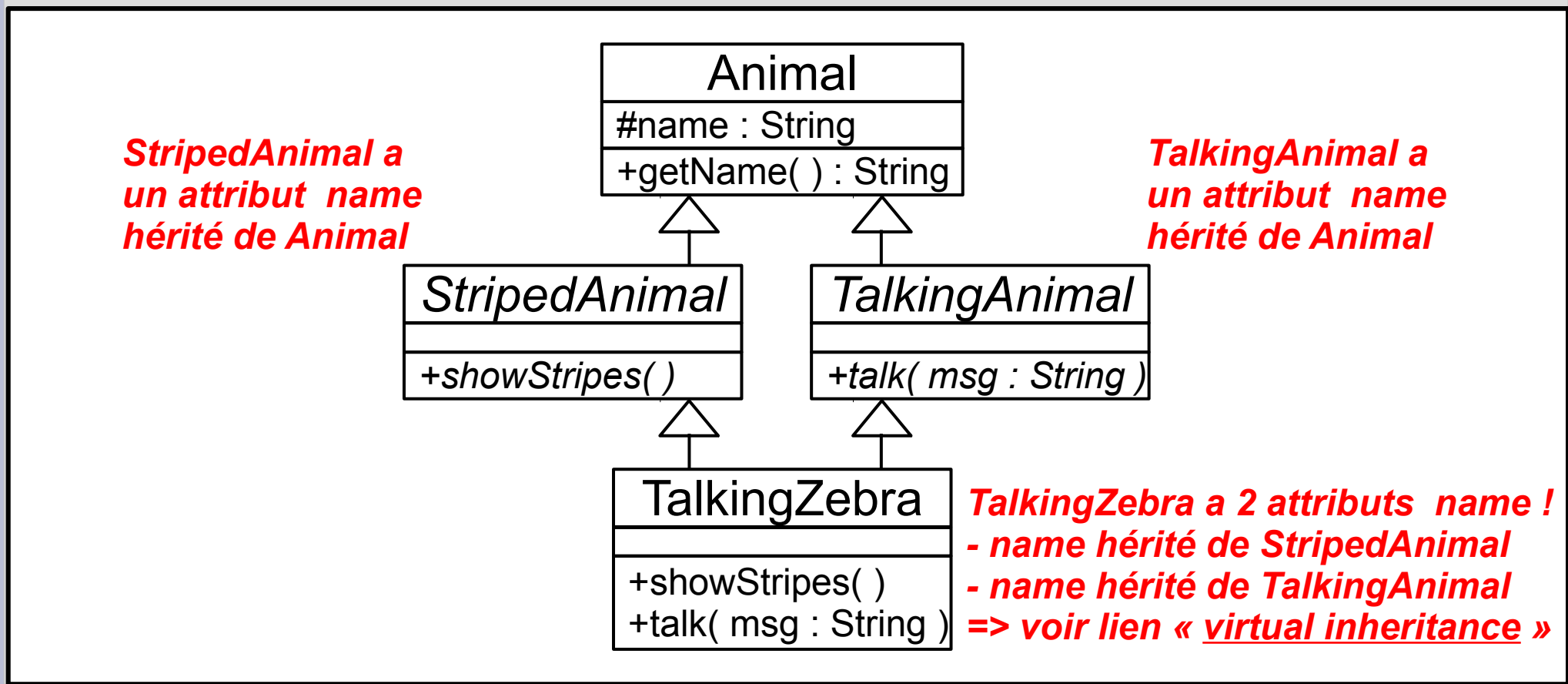
talking_zebra.cpp

```
void TalkingZebra::showStripes()
{
    std::cout << "|||||||" << std::endl;
}

void TalkingZebra::talk(std::string msg)
{
    std::cout << "I zay " << msg << std::endl;
}
```

toutes les méthodes virtuelles pures de toutes les classes de base doivent être définies pour que la classe dérivée puisse être instanciée

Héritage multiple

- Problèmes en cas d'**héritage en diamant** quand une même classe est « héritée plusieurs fois »
- Il faut utiliser l'héritage virtuel *virtual inheritance*
  Ou éviter ces situations (*delegation pattern...*)



COURS 9

A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

Design patterns

Study for "Composition VII", Wassily Kandinsky

Design patterns

- Un ***design pattern*** (patron de conception) est
  - « *un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel* »
  - « *Il décrit une solution standard, utilisable dans la conception de différents logiciels* »
  - « *Un patron de conception est issu de l'expérience des concepteurs de logiciels* »
  - « … *le patron de conception décrit les grandes lignes d'une solution, qui peuvent ensuite être modifiées et adaptées selon les besoins* »

Design patterns

- Une analogie en programmation procédurale :
  - Pour parcourir une matrice on va utiliser **une double boucle imbriquée**
  - Pour « blinder » une saisie on va utiliser **faire saisie tant que saisie pas correcte**
  - Pour faire un menu application en console on va **faire**
    saisie choix,
    selon choix appeler sous-prog. associé
    tant que choix différent de quitter
- Ces algorithmes sont des sortes de design patterns, des « recettes qui marchent »

Design patterns

- Les design patterns sont donc un peu comme des algorithmes mais au niveau "orienté objet"
- « *Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts* »
- Rangés en 3 catégories :

➔ créateurs : ils définissent comment faire l'instanciation et la configuration des classes et des objets

➔ structuraux : ils définissent comment organiser les classes d'un programme dans une structure plus large

➔ comportementaux : ils définissent comment organiser les objets pour que ceux-ci collaborent et expliquent le fonctionnement des algorithmes impliqués

Design patterns

Simple survol : tout ceci n'est pas au programme !

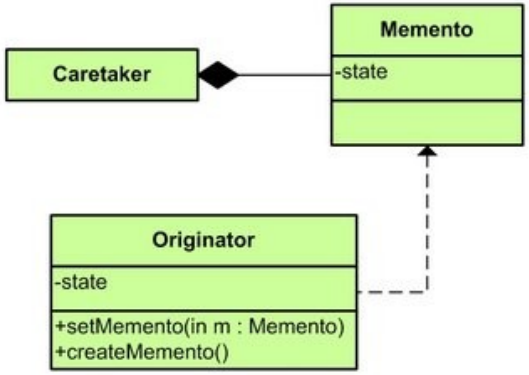
Les 23 patterns classiques

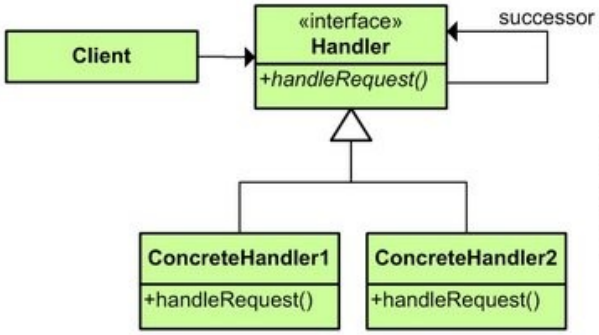
C Abstract Factory	S Facade	S Proxy
S Adapter	C Factory Method	B Observer
S Bridge	S Flyweight	C Singleton
C Builder	B Interpreter	B State
B Chain of Responsibility	B Iterator	B Strategy
B Command	B Mediator	B Template Method
S Composite	B Memento	B Visitor
S Decorator	C Prototype	

Memento

Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.





Chain of Responsibility

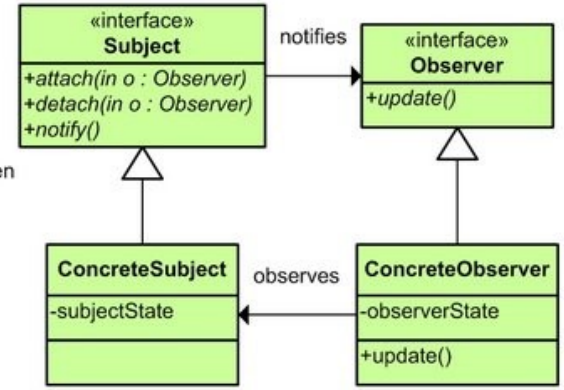
Type: Behavioral

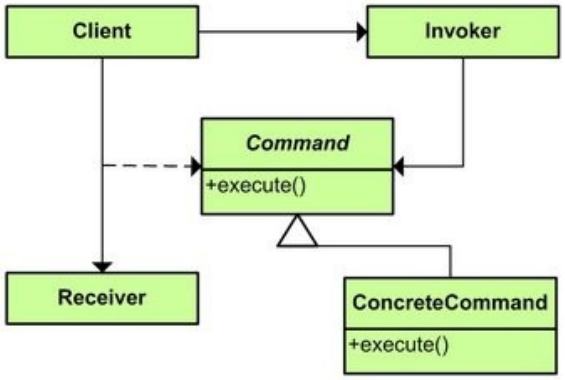
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.





Command

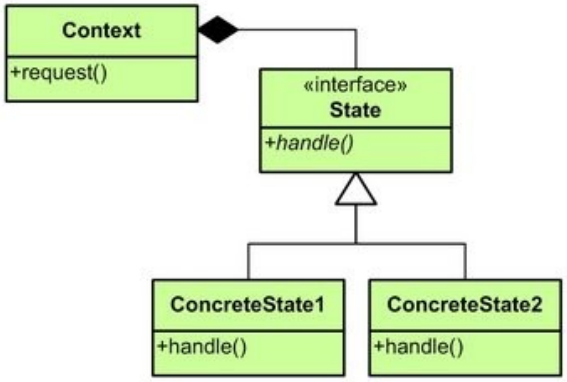
Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

State

Type: Behavioral

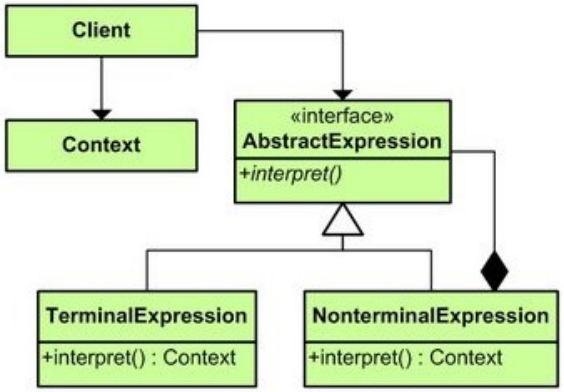
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Design patterns

Les 23 patterns classiques

Simple survol : tout ceci n'est pas au programme !



Interpreter

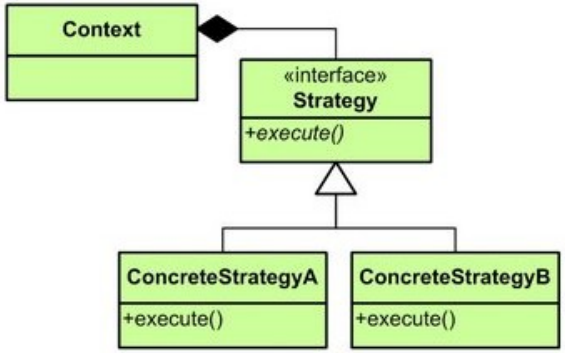
Type: Behavioral

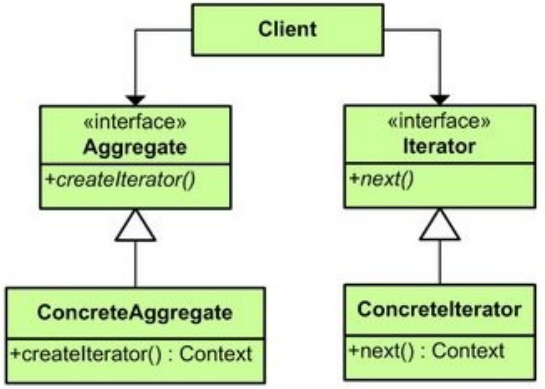
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.





Iterator

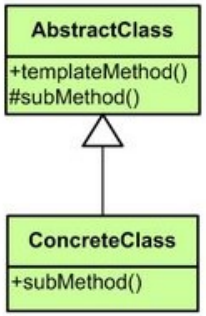
Type: Behavioral

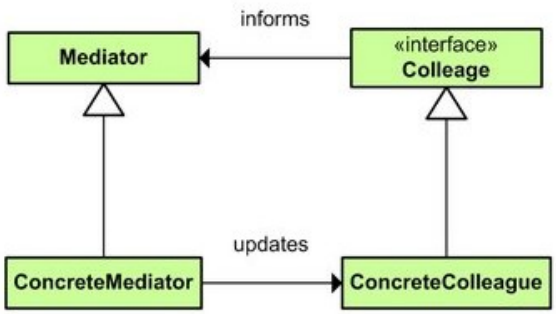
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Template Method

Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.





Mediator

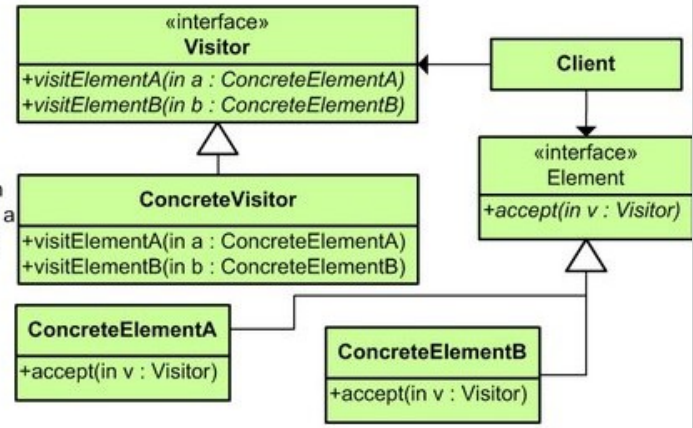
Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

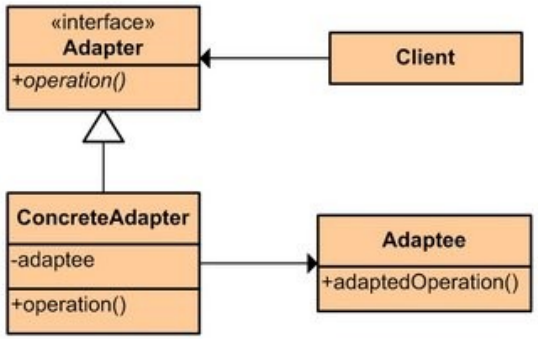


Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

Design patterns

Les 23 patterns classiques

Simple survol : tout ceci n'est pas au programme !



Adapter

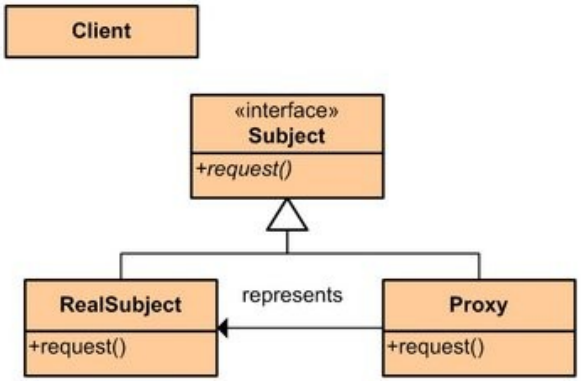
Type: Structural

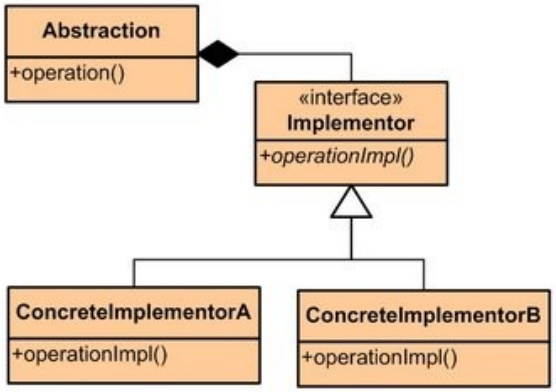
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.





Bridge

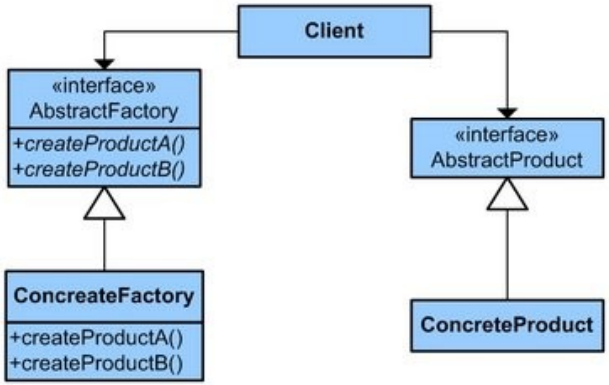
Type: Structural

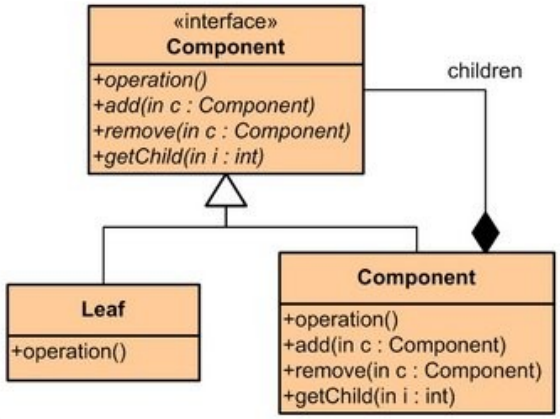
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.





Composite

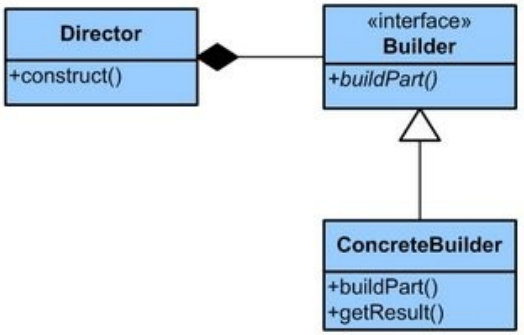
Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Builder

Type: Creational

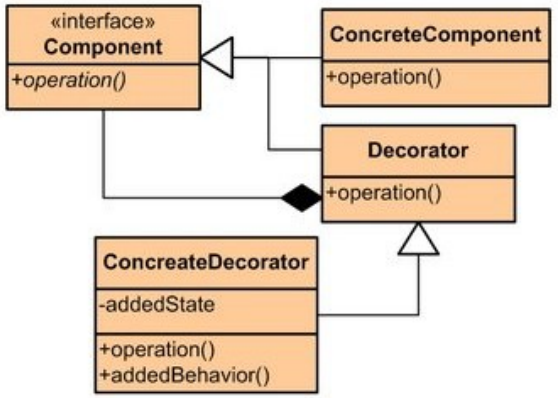
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Design patterns

Simple survol : tout ceci n'est pas au programme !

Les 23 patterns classiques



Decorator

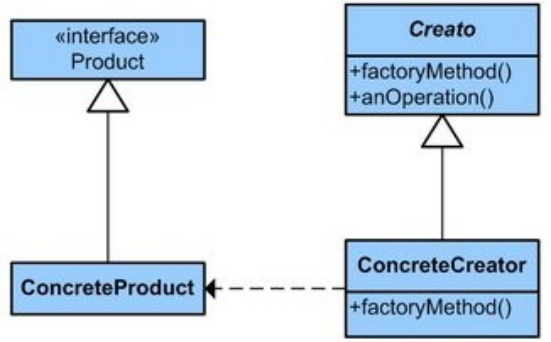
Type: Structural

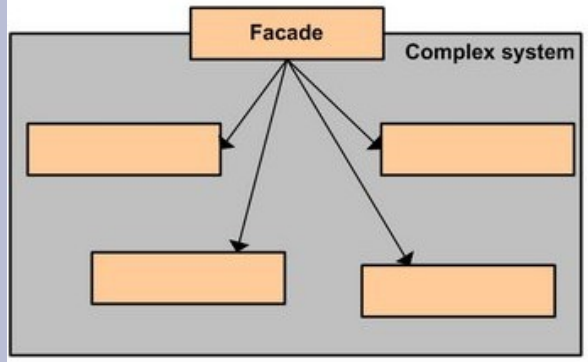
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.





Facade

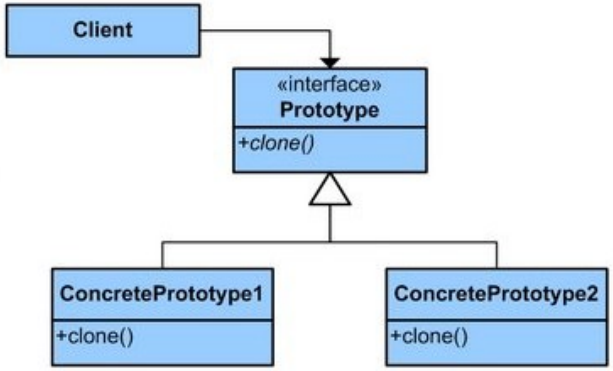
Type: Structural

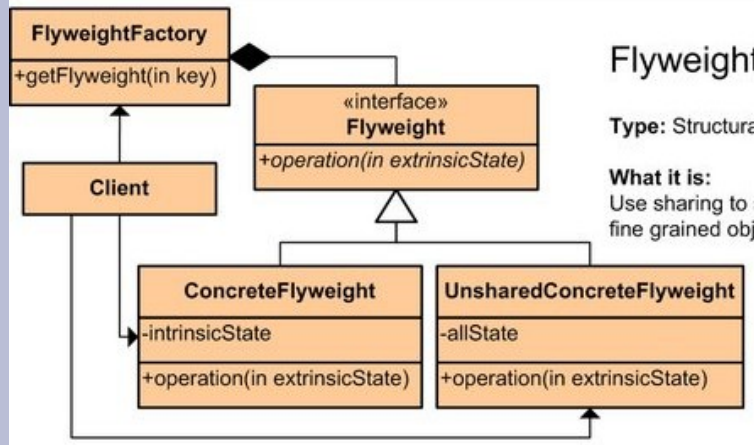
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.





Flyweight

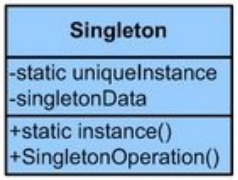
Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements o Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.

COURS 9

A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

Delegation pattern

Roue de bicyclette, Marcel Duchamp

Delegation pattern

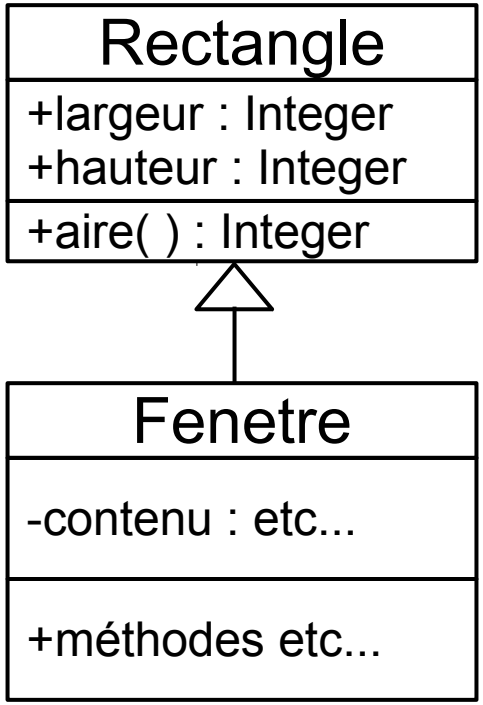
- Le **pattern délégation** correspond à l'usage de la composition comme **alternative à l'héritage**
- Exemple, on a déjà une classe Rectangle, on aimerait **réutiliser ses méthodes** pour une classe Fenetre. Mais l'héritage pose problème...

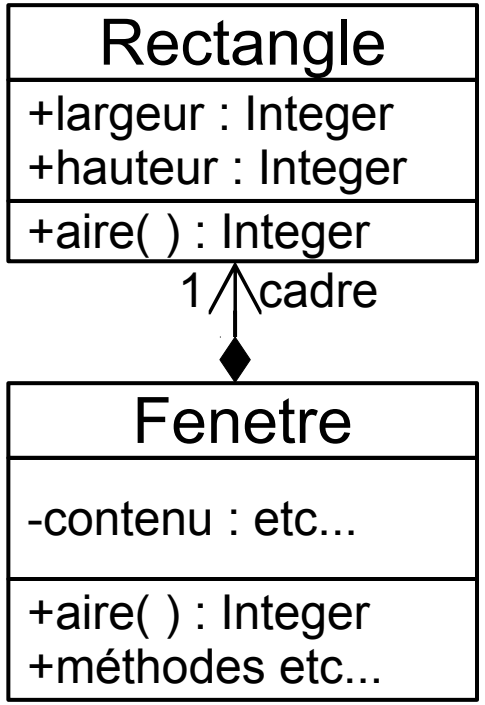
Héritage Problèmes :

conceptuellement une Fenetre n'est pas un Rectangle

Rectangle type valeur
Fenetre type entité

2 raisons de ne pas vouloir faire un héritage





Composition OK :

Fenetre va recevoir sa « Rectanglitude » d'un objet composant de type Rectangle

La méthode aire de Fenetre déléguera le travail à la méthode aire de Rectangle

Delegation pattern

- la composition comme **alternative à l'héritage**

```
struct Rectangle
{
    int m_largeur, m_hauteur;
    int aire();
};
```

```
Rectangle::aire()
{
    return m_largeur * m_hauteur;
}
```

```
class Fenetre
{
    public :
       Fenetre(Rectangle cadre);
       int aire();
       /// + méthodes ...

    private :
       Rectangle m_cadre;
       /// + m_contenu ...
};
```

```
Fenetre::Fenetre(Rectangle cadre)
    : m_cadre{cadre}
{ }

Fenetre::aire()
{
    /// Délégation !
    return m_cadre.aire();
}
```

```
int main()
{
    Fenetre maFenetre{ {10, 5} };

    std::cout << maFenetre.aire()
              << std::endl;

    return 0;
}
```

50

Delegation pattern

- Attention aux raccourcis ! On ne dit **pas** que la composition peut toujours remplacer l'héritage !
- Les défauts de l'héritage sont un couplage fort entre classe et des contraintes sémantiques
- La délégation nécessite du code de "plomberie" (*forwarder* les appels de méthodes au délégué)
- On a déjà vu un exemple de délégation en TP avec la relation Sommet / Coords :
  On ne peut pas dire "*1 Sommet **est** 1 Coords*"
  On peut dire "*1 Sommet **a** 1 Coords*"
- Vous avez sûrement remarqué qu'il fallait *forwarder* (déléguer) pas mal de méthodes de Sommet à Coords

Delegation pattern

- La « délégation » au sens large indiquée précédemment correspond au *forwarding*
- En toute rigueur le pattern Délégation implique un passage en paramètre au composant appelé du contexte *this* de l'appelant : le composant fait **comme si** il était une classe de base.
- *Ce pattern précis fera l'objet d'un exo TD/TP 9*

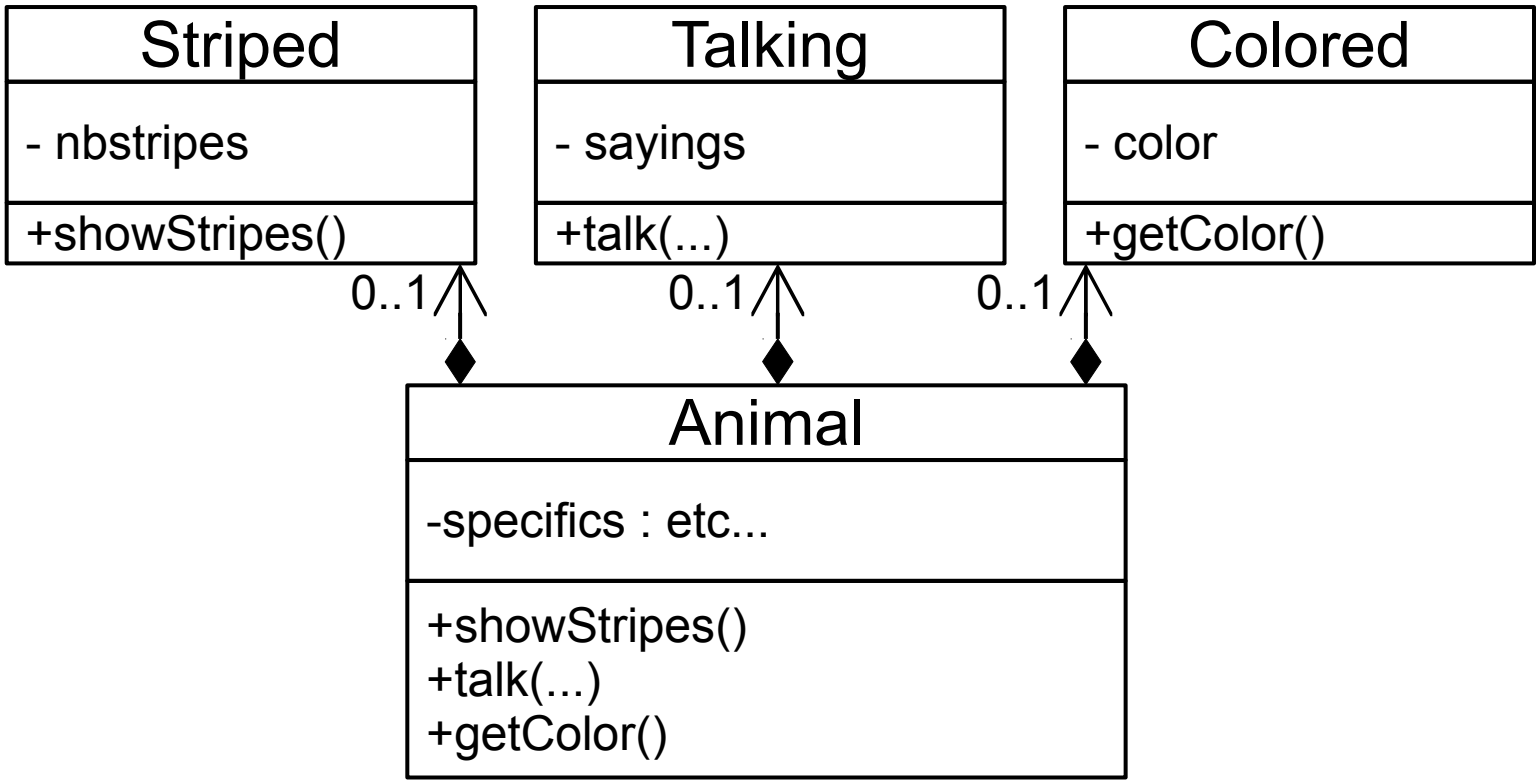
Delegation pattern

- En général la délégation ou le *forwarding* permet de gérer des **combinatoires** d'aptitudes, souvent trop lourds avec l'héritage multiple

	Striped	Talking	Colored
Dog			
Flamingo			✔
Pigeon		✔	
Parrot		✔	✔
Tiger	✔		
Dinosaur	✔		✔
TalkingZebra	✔	✔	
Marty	✔	✔	✔

Delegation pattern

- En général la délégation ou le *forwarding* permet de gérer des **combinatoires** d'aptitudes, souvent trop lourds avec l'héritage multiple



COURS 9

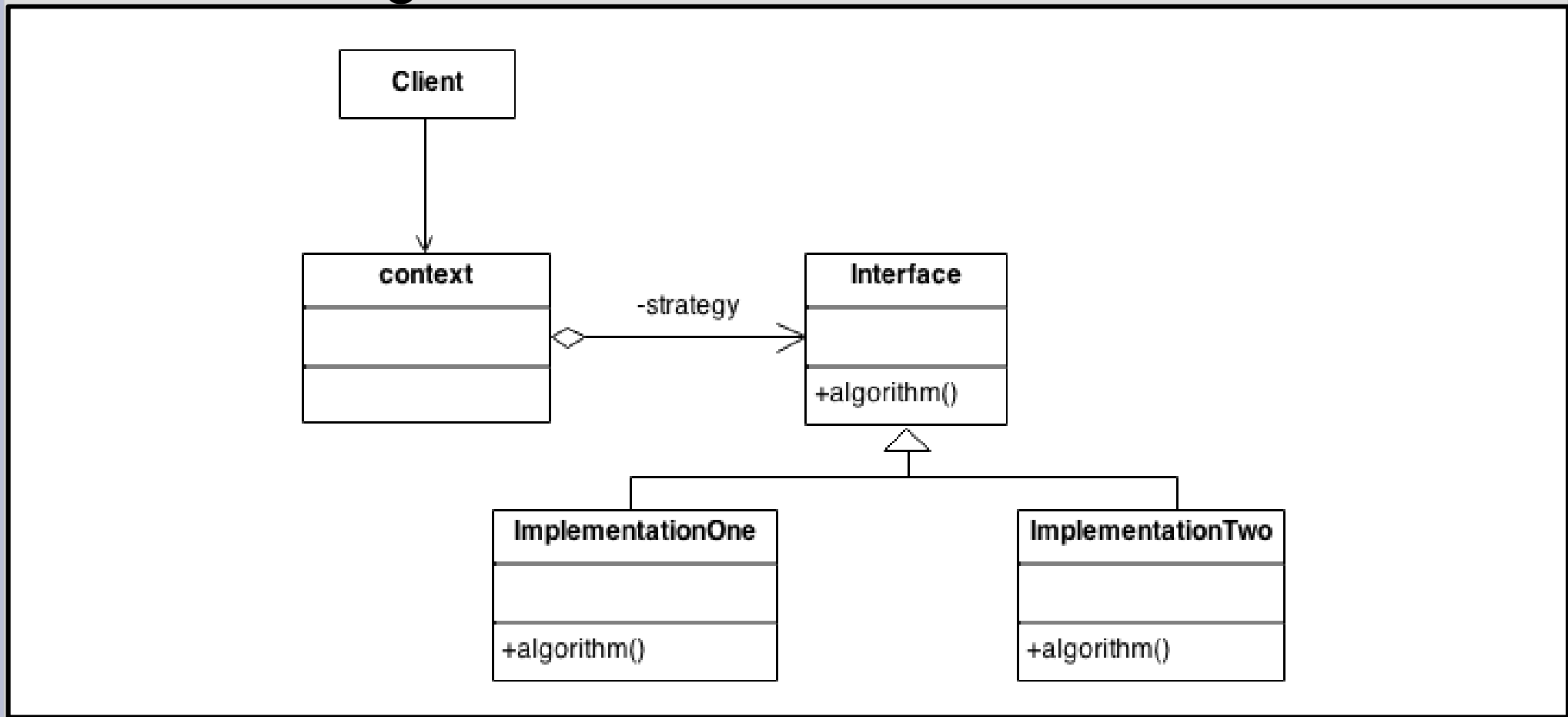
A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

Strategy pattern

Watchdog II, Nam June Paik

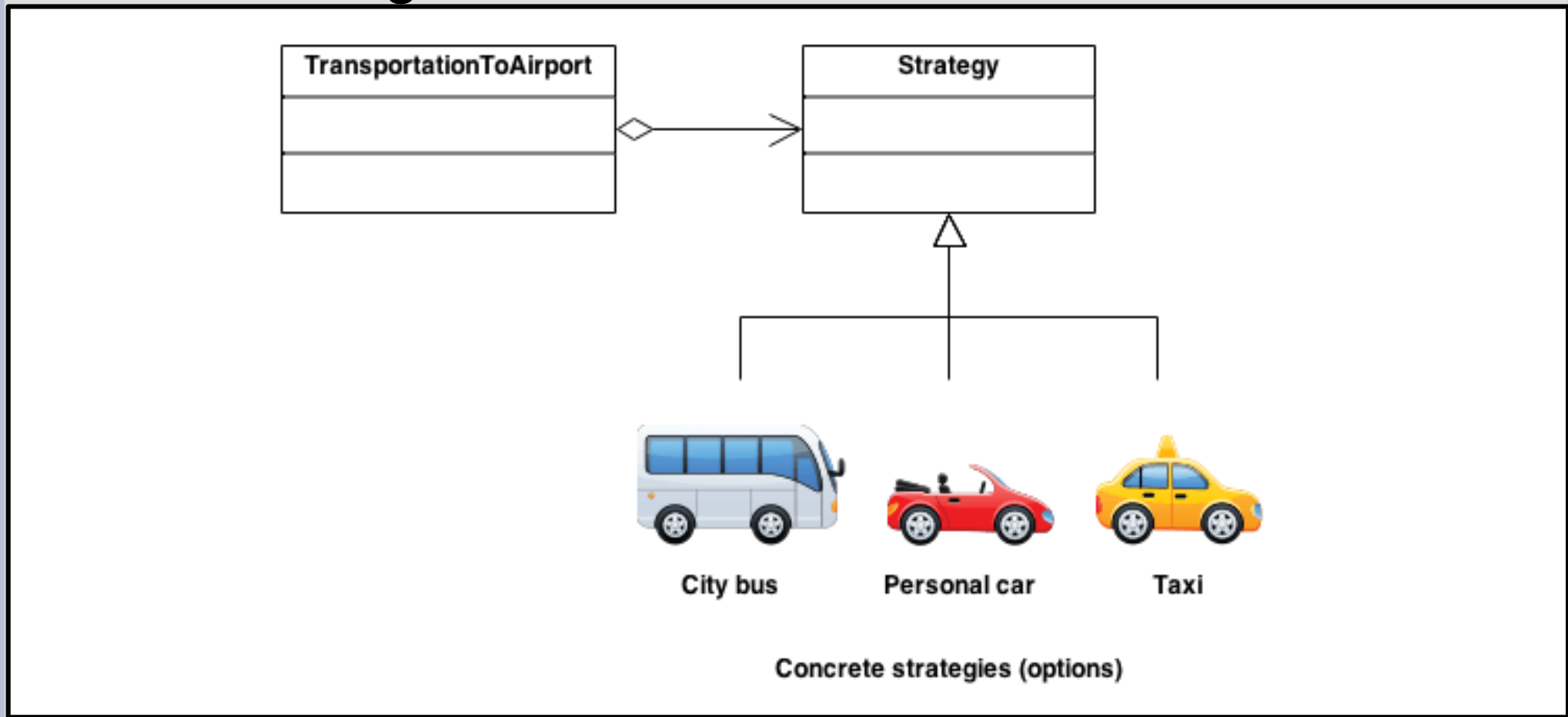
Strategy pattern

- Le *pattern strategy* est un usage particulier de délégation : on délègue à une classe qui hérite d'une interface. Ceci permet de choisir/changer de stratégie même en cours d'utilisation...



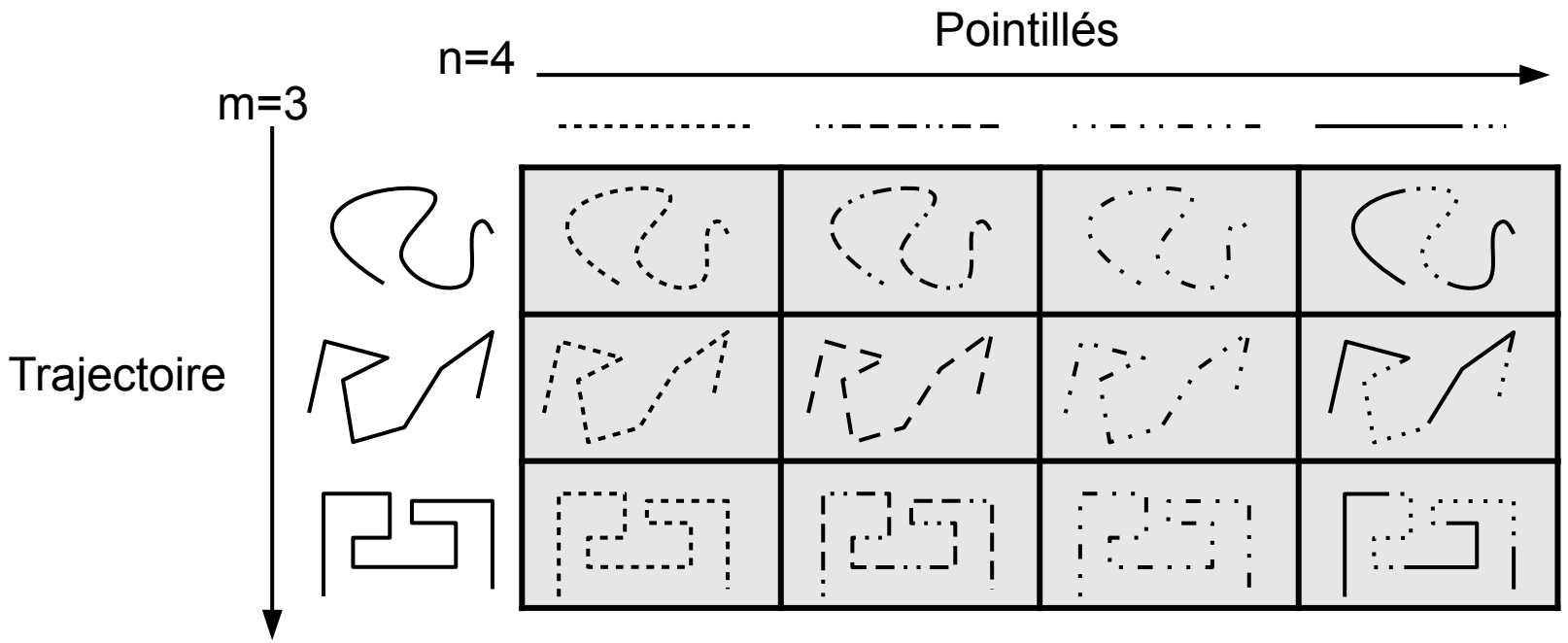
Strategy pattern

- Le *pattern strategy* est un usage particulier de délégation : on délègue à une classe qui hérite d'une interface. Ceci permet de choisir/changer de stratégie même en cours d'utilisation...



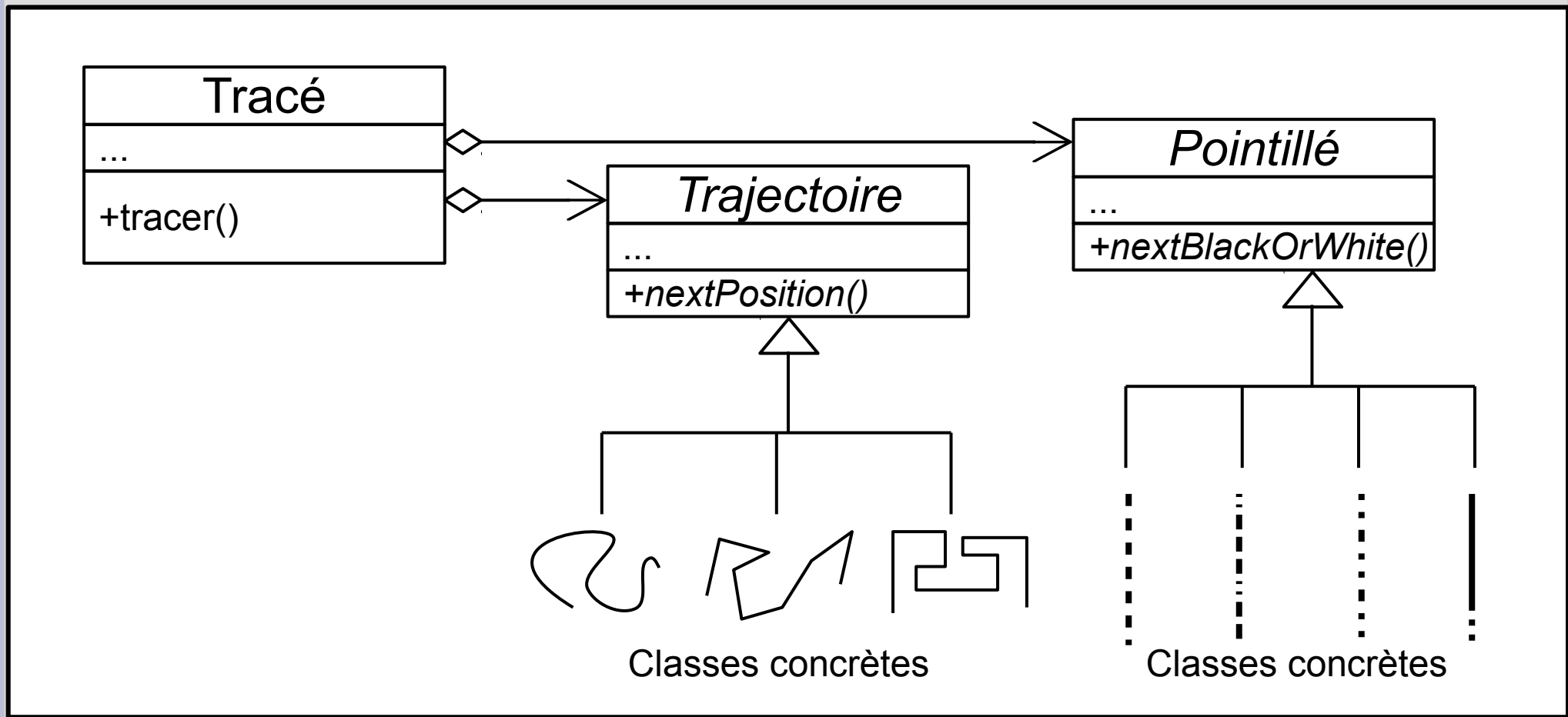
Strategy pattern

- Là encore on dispose d'une technique puissante pour gérer des **combinatoires** sans multiplier les codes croisés (m+n au lieu de m×n)



Strategy pattern

- Là encore on dispose d'une technique puissante pour gérer des **combinatoires** sans multiplier les codes croisés (m+n au lieu de m×n)



COURS 9

A) Classes abstraites / interfaces
B) Couplage / inversion du contrôle
C) Héritage multiple
D) Design patterns
E) Delegation pattern
F) Strategy pattern
G) Composite pattern

Composite pattern

Orphism, Sonia & Robert Delaunay

Composite pattern

- Le pattern composite permet de gérer des objets composites arborescents où des composants sont élémentaires (feuilles) et d'autres sont des groupes de composants (composites)

Diagramme d'objets

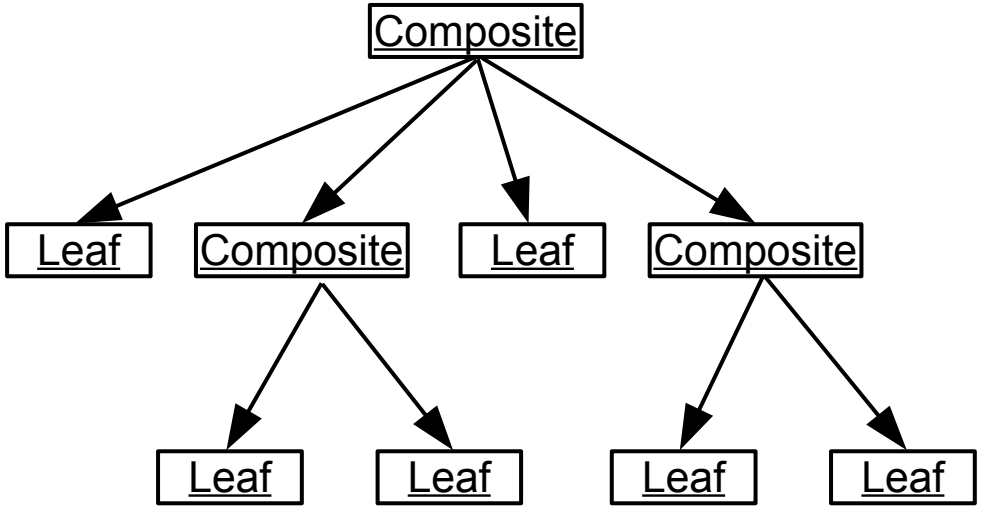
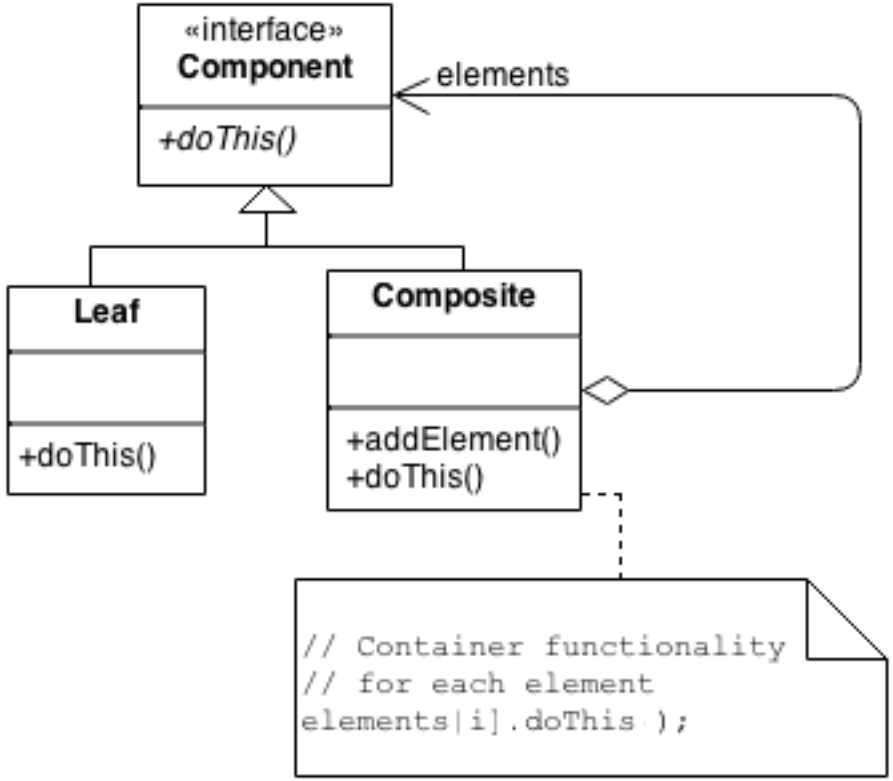
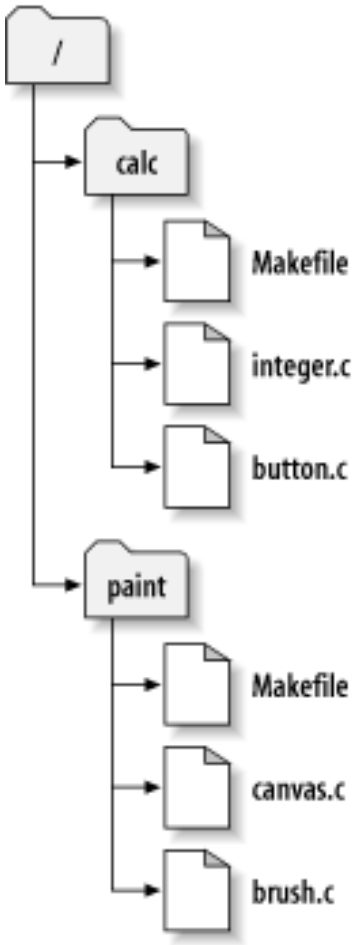


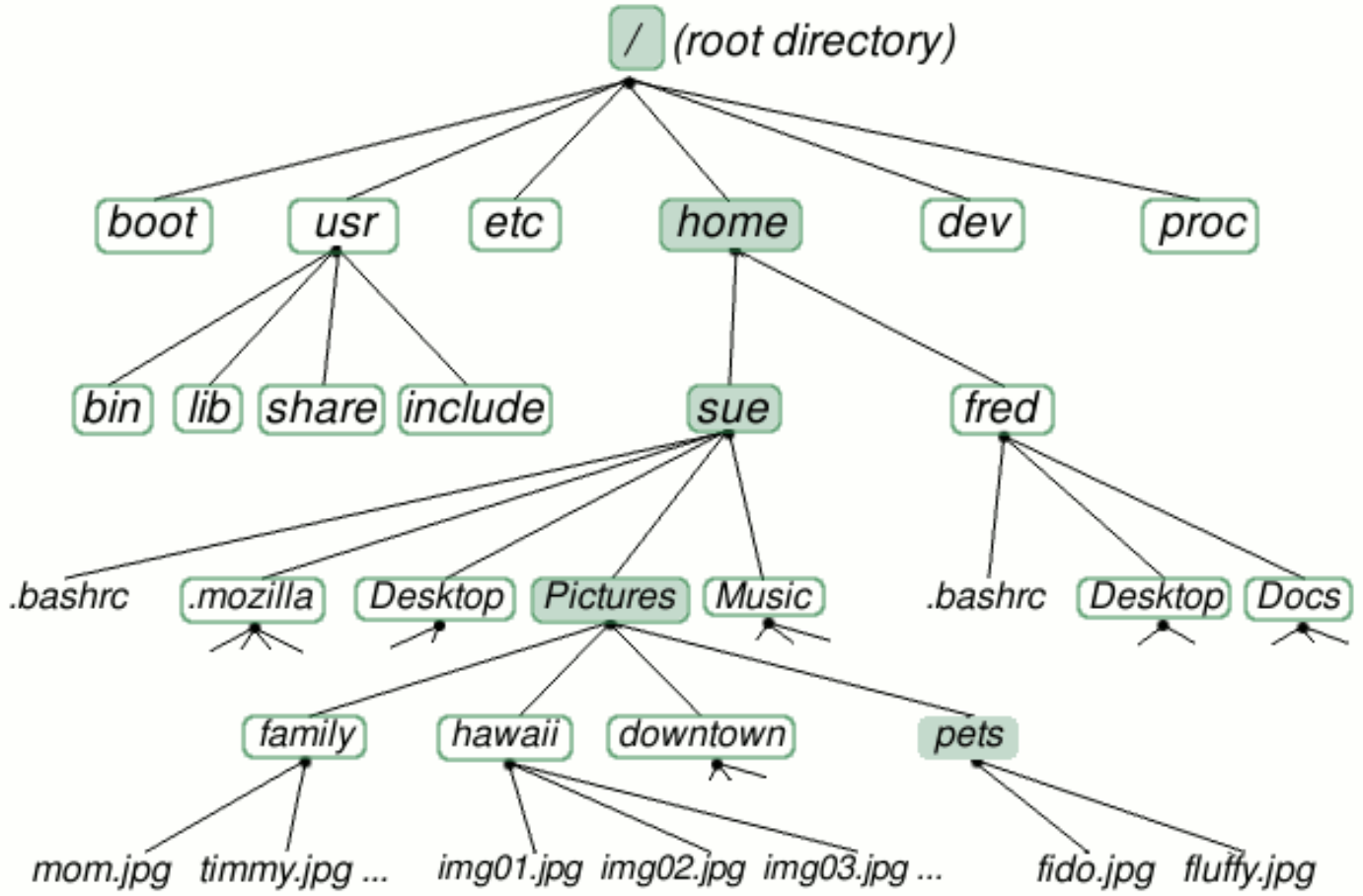
Diagramme de classes



Composite pattern

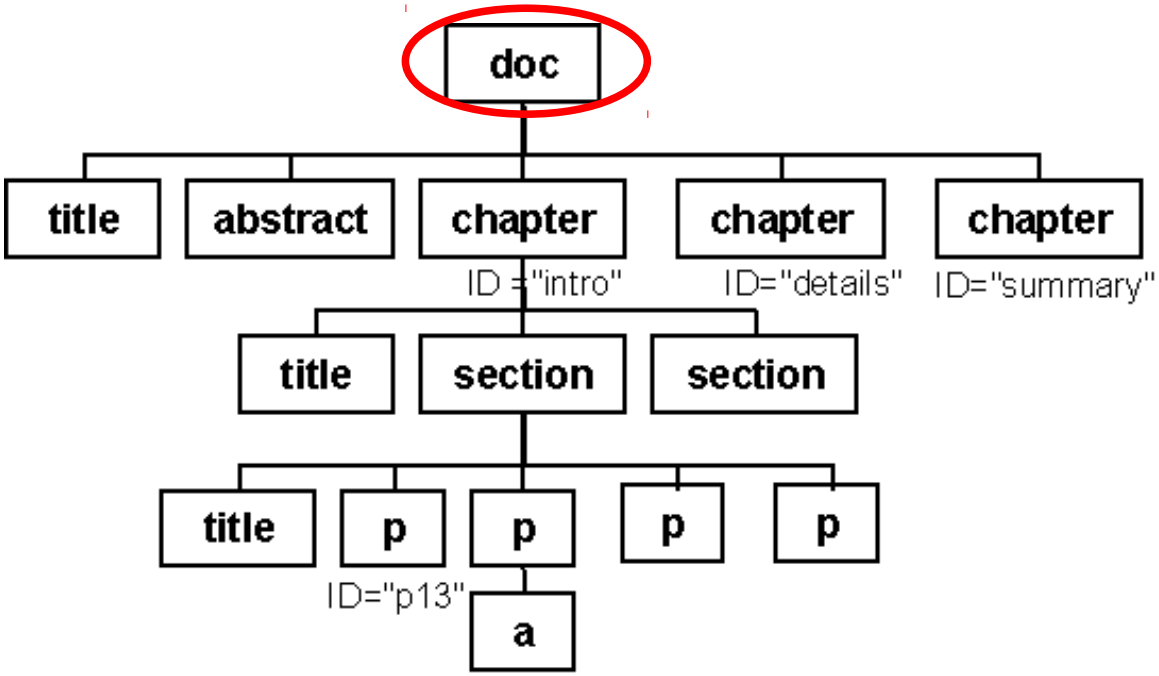
- **Logiques de conteneurs imbriqués**
  exemple : systèmes de fichiers

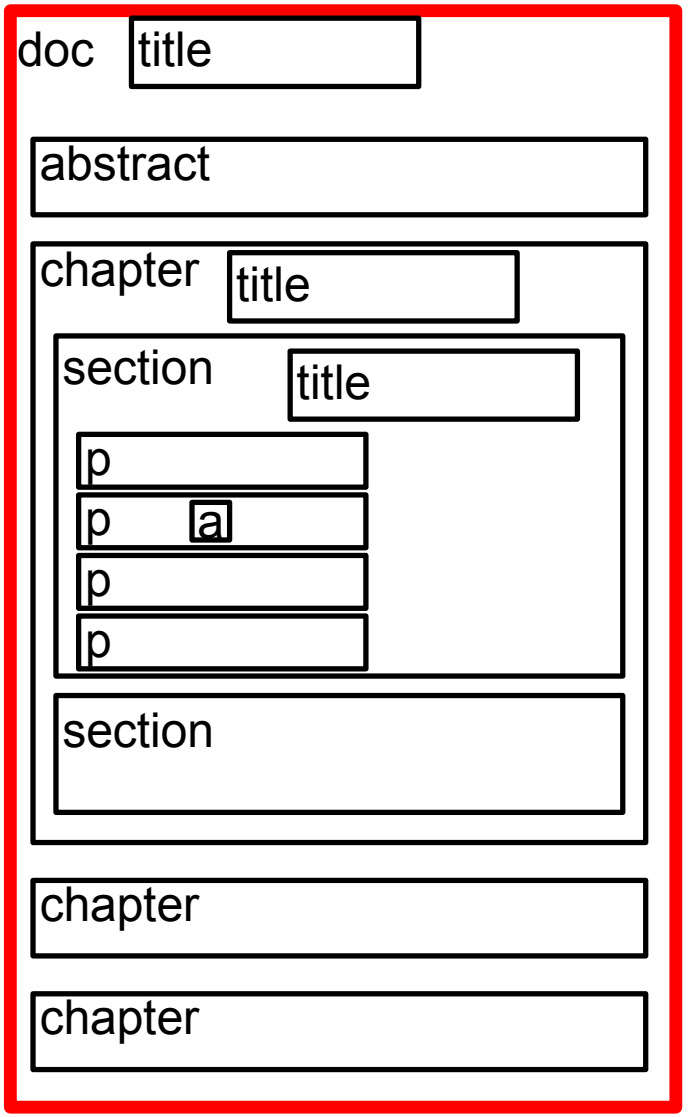




Composite pattern

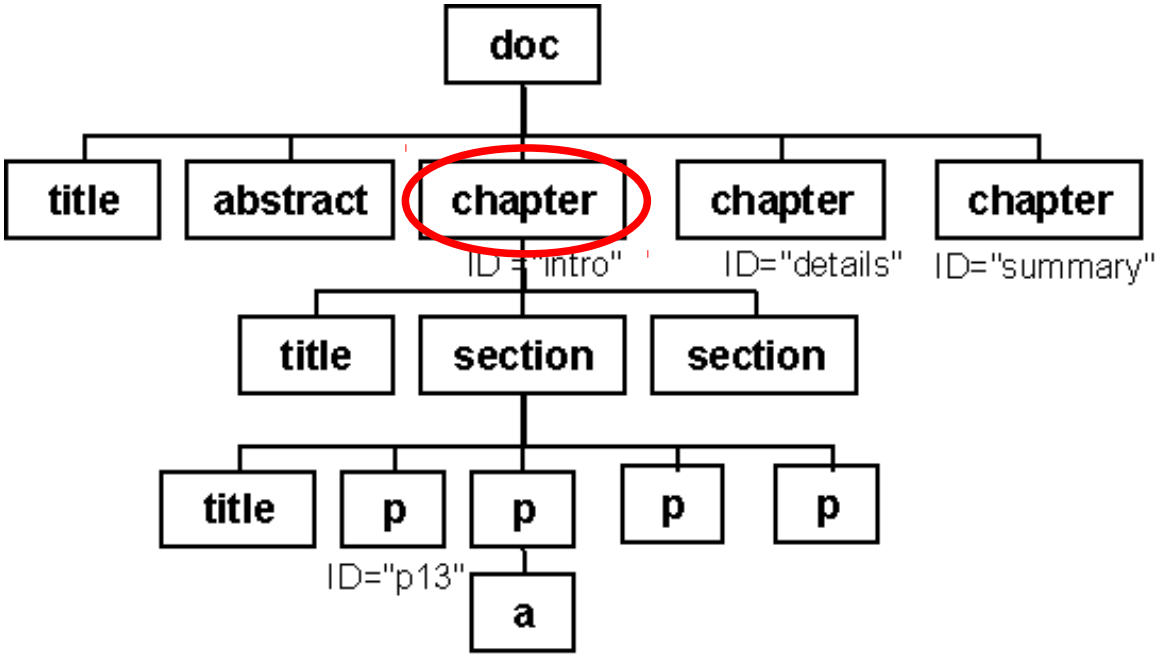
- **Logiques de conteneurs imbriqués**
  exemple : documents structurés

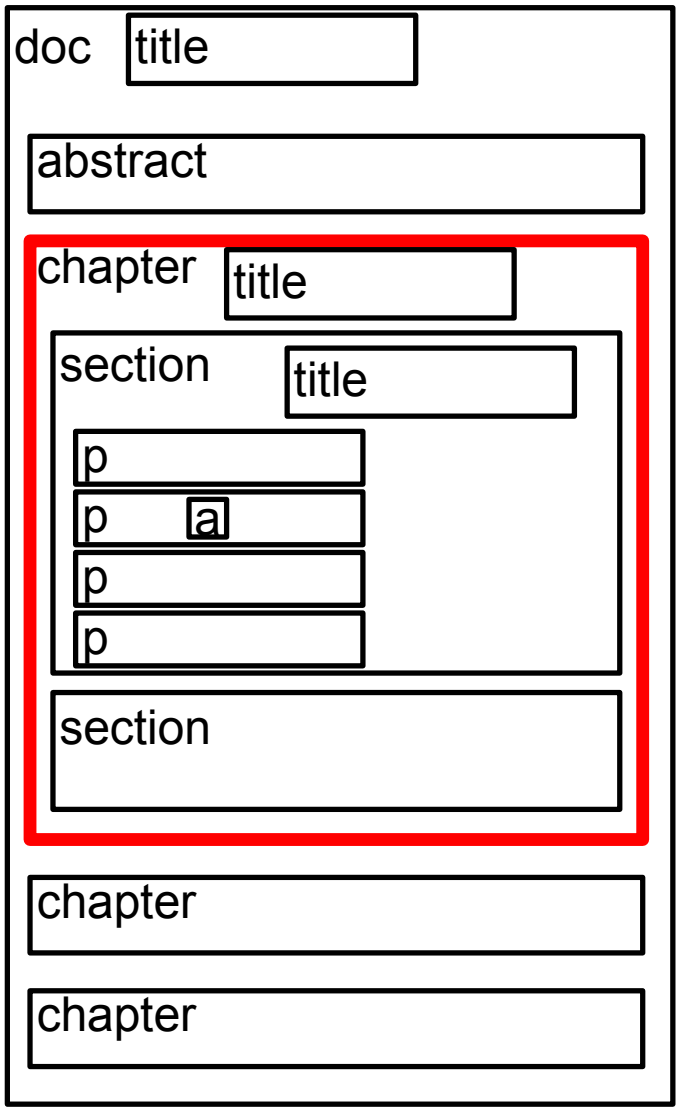




Composite pattern

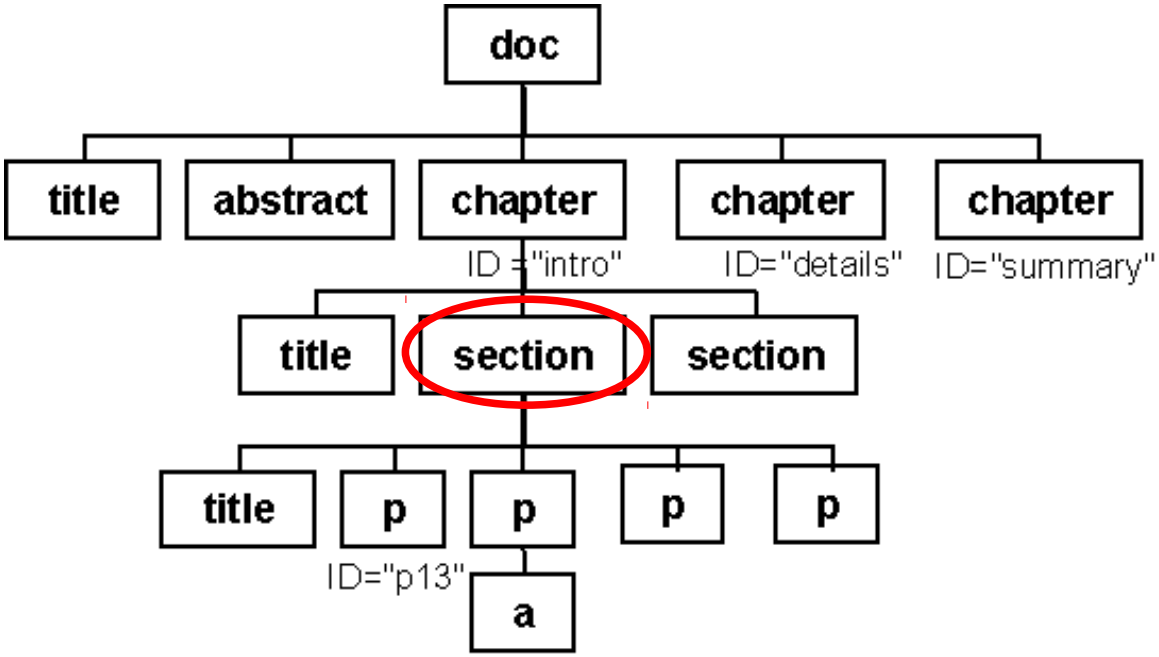
- **Logiques de conteneurs imbriqués**
  exemple : documents structurés

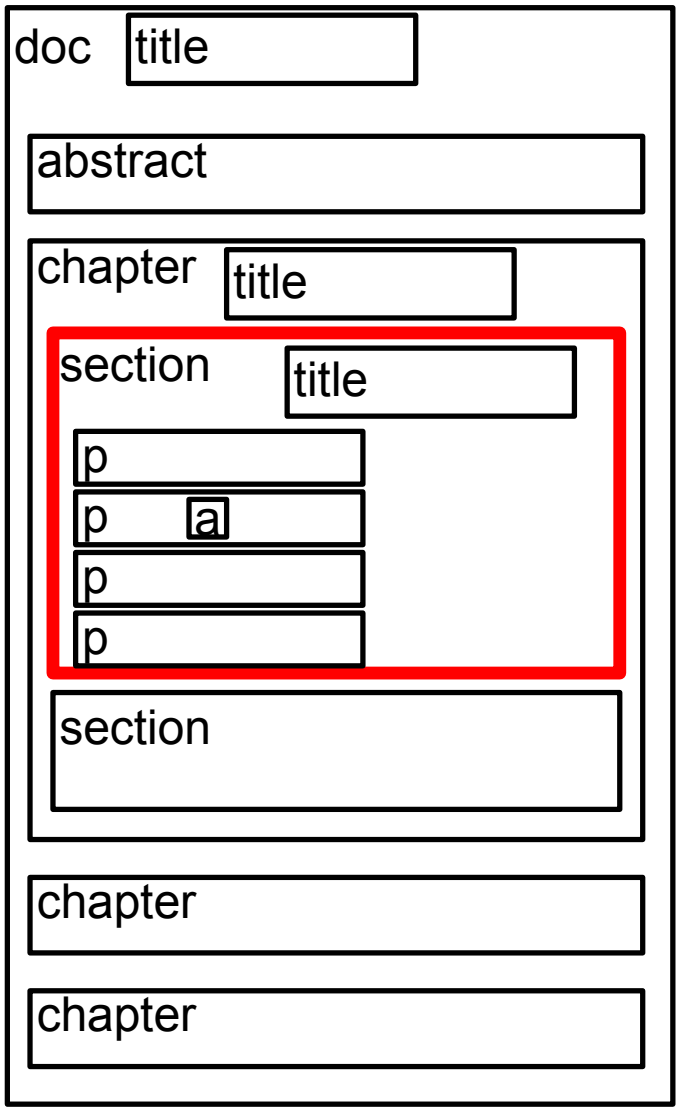




Composite pattern

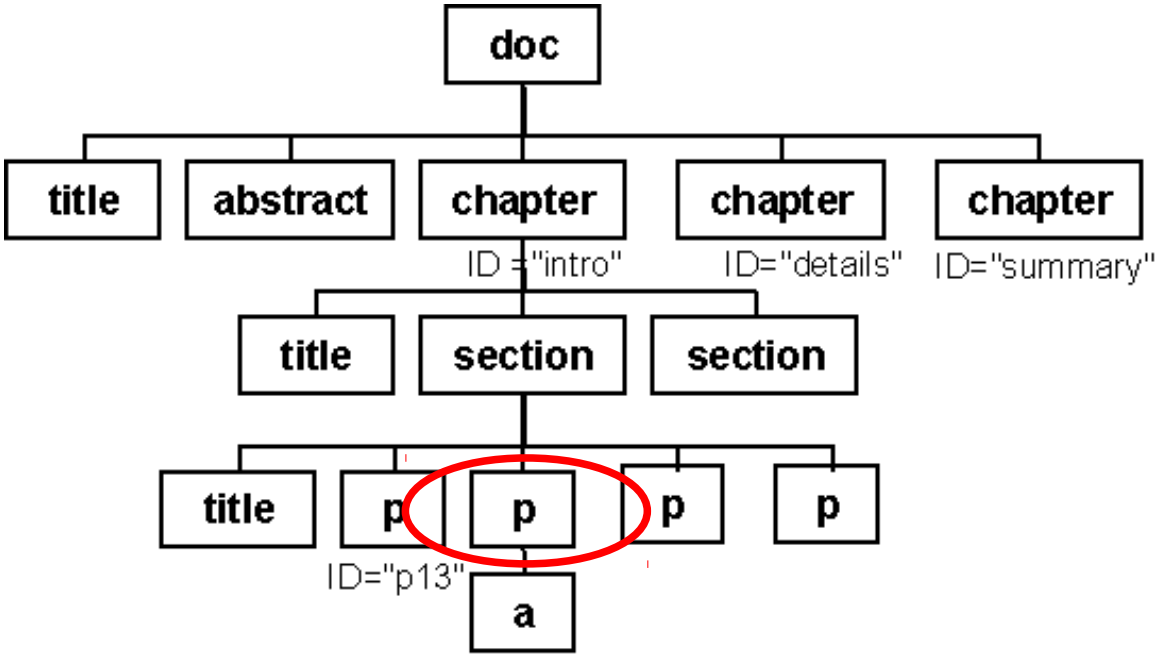
- **Logiques de conteneurs imbriqués**
  exemple : documents structurés

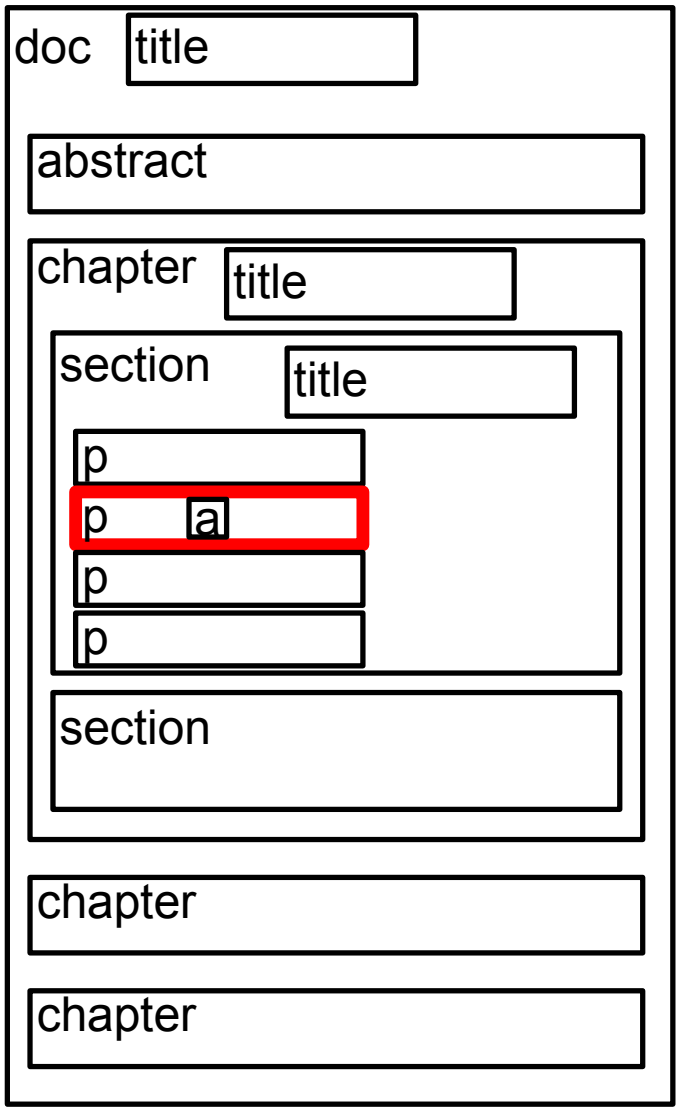




Composite pattern

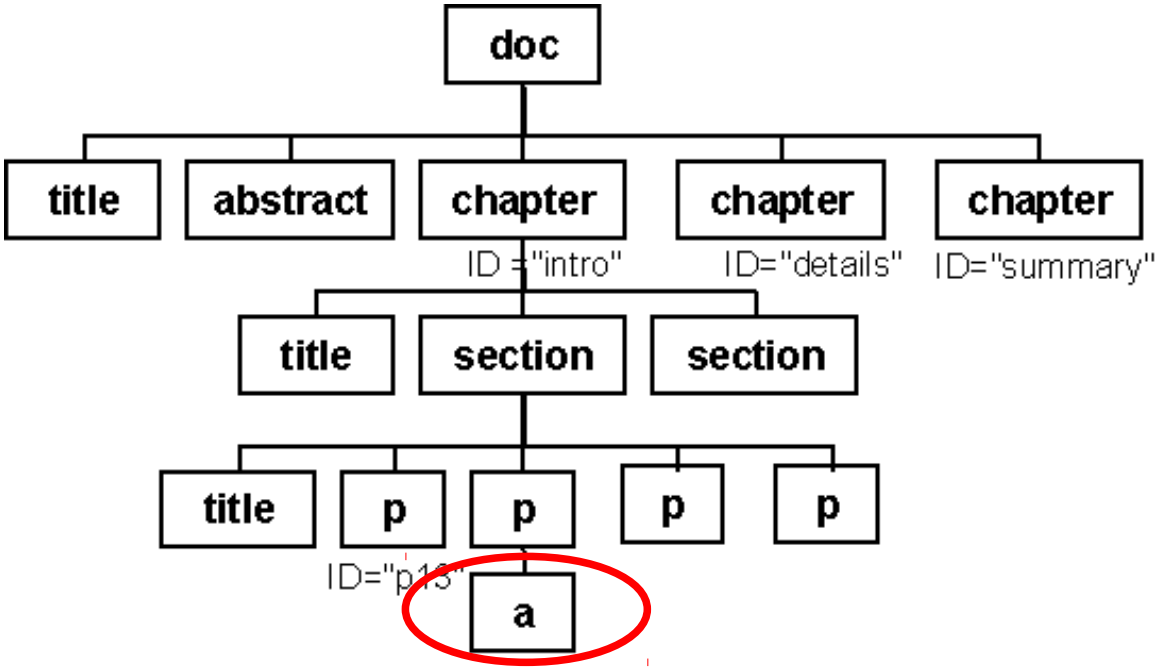
- **Logiques de conteneurs imbriqués**
  exemple : documents structurés

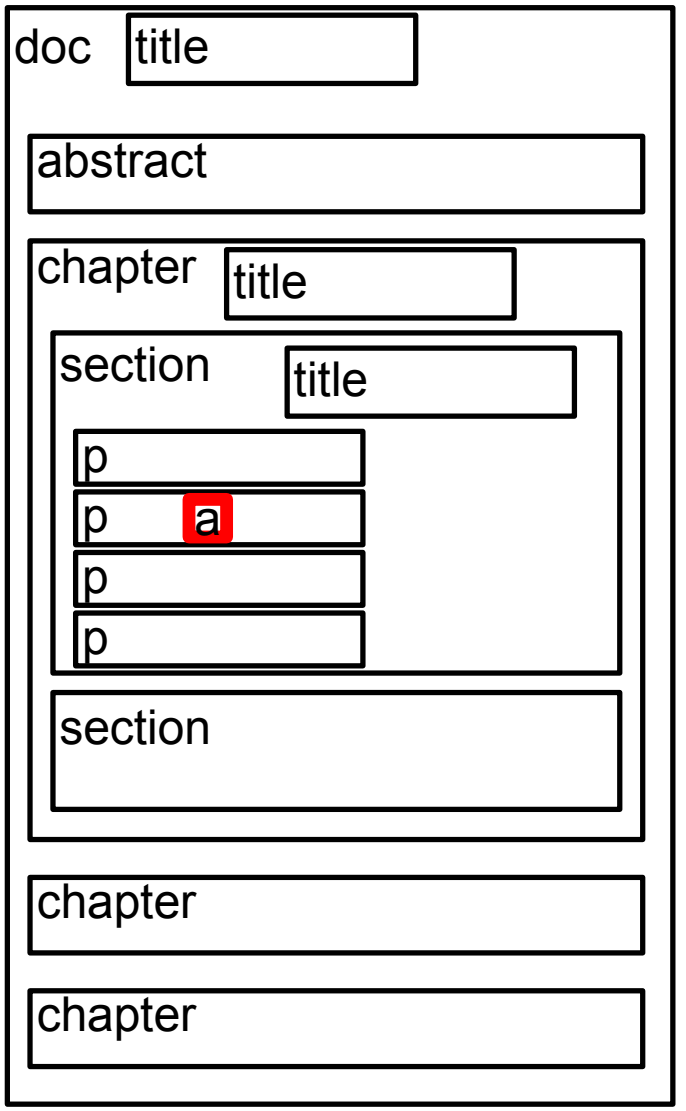




Composite pattern

- **Logiques de conteneurs imbriqués**
  exemple : documents structurés





Composite pattern

- Le pattern composite sera certainement très **utile au projet** : c'est la façon idéal d'organiser des groupes d'éléments graphiques et de gérer à égalité les groupes et les primitives

- *Ce pattern fera l'objet d'un exo TD/TP 9*