



ISIE4 – LOO C++

Spécialisation & Généralisation
(Factorisation)

Au menu...

- Spécialisation
- Factorisation d'implémentation
- Factorisation conceptuelle
 - Généralisation
 - Interfaces



La spécialisation...

- Rappel - une classe définit un type de données, et est
 - Une unité d'encapsulation
 - Un modèle de construction, une description concrète, une implémentation
- L'héritage permet de personnaliser une classe existante pour
 - Lui ajouter des fonctionnalités
 - Modifier le comportement d'actions déjà existantes
 - Sans modifier la classe de base
 - Construction d'un nouveau type (sous-classe)



Un exemple pour

- La syntaxe
- La redéfinition de méthodes
 - Simple et avec réutilisation
- La gestion des constructions
 - Et des destructions



Redéfinition \neq Surcharge

■ Redéfinition

- Même signature dans la classe fille que dans la classe mère
- Attention : la redéfinition masque les définitions

■ Surcharge

- Signatures différentes
- Dans la même classe, ou pas
 - Une méthode de la classe mère peut être surchargée dans la classe mère et/ou dans la classe fille

Protection & contrôle d'accès

- Rappels...
- private:
 - visible ***uniquement*** depuis les fonctions membres de ***la classe***
- public:
 - visible depuis n'importe quelle partie du code
- Être dans une fonction membre d'une sous-classe ne signifie pas être dans la classe de base

Un peu restrictif non ?

- Oui, d'où un 3eme domaine : protected
- Un membre protégé est:
 - de l'extérieur (i.e. ni dans la classe ni dans une sous-classe) considéré inaccessible
 - de l'intérieur (i.e. dans la classe ou dans n'importe quelle sous-classe) considéré comme accessible



Résumé

■ Pour la dérivation publique

Protection dans la classe de base	Accessibilité dans une fonction membre de la classe dérivée	Accessibilité dans une fonction amie de la sous-classe	Accessibilité par utilisation de la classe dérivée (hors cas précédents)	Protection vis-à-vis d'une nouvelle dérivation
private	non	non	non	private
protected	oui	oui	non	protected
public	oui	oui	oui	public

Points d'attention

- Attention à ne pas utiliser la dérivation pour violer l'encapsulation
 - en faisant réapparaître publiquement des choses qui devaient être cachées...
- Attention donc en faisant apparaître des membres avec le qualificatif `protected`...
 - il ne faut jamais oublier que l'on peut dériver une classe...
 - Arrivée du qualificatif « final » à partir de C++11
 - Interdit la construction d'une sous-classe
- Les violations sont forcément conscientes... C'est déjà une chose...



Compatibilité de types

- Si spécialisation = ajout structurel
 - On doit pouvoir « supprimer » des choses ajoutées sans perdre la cohérences des définitions
- Donc...
 - On peut toujours convertir une instance d'une classe donnée en un objet de sa super-classe
 - On peut toujours convertir un pointeur vers une instance d'une classe donnée en un pointeur vers un objet de sa super-classe



Rapide exemple...

dérivation publique

```
class DeBase {
    ...
};

class SousClasse : public DeBase {
    ...
};

class BienEnDessous : public SousClasse {
    ...
};

int main() {
    BienEnDessous unObjet;
    DeBase unObjetDeBase = unObjet;
    unObjet = unObjetDeBase; // strictly forbidden...

    BienEnDessous *p = &unObjet;
    DeBase *p2;
    p2 = p;
    p = p2; // formellement interdit
}
```

Et côté méthodes appelées...

```
class DeBase {
public:
    void f() { cout << "DeBase" << endl; }
};

class SousClasse : public DeBase {
public:
    void f() { cout << "SousClasse" << endl; }
};

int main() {
    SousClasse unObjet;
    unObjet.f();
    DeBase instanceDeBase = unObjet;
    instanceDeBase.f();

    SousClasse *pSousClasse = &unObjet;
    pSousClasse->f();
    DeBase *pBase = pSousClasse;
    pBase->f();
    return 0;
}
```

```
[Gibi:~]% ./main
SousClasse
DeBase
SousClasse
DeBase
[Gibi:~]%
```

Early binding

- En résumé :

- la méthode qui sera appelée à l'exécution est déterminée statiquement (à la compilation)
 - ***Early binding***
- la méthode appelée est celle apparaissant ***dans le type*** de la référence ou du pointeur, ou à défaut une méthode héritée (attention au masquage - redéfinition vs. surcharge)

Late binding

- Le late binding permet de provoquer à l'exécution la recherche de la méthode appropriée
 - la méthode appelée sera celle apparaissant ***dans l'objet*** désigné par la référence ou par le pointeur, ou à défaut une méthode héritée (attention au masquage - redéfinition vs. surcharge)
 - Polymorphisme par sous-typage
- Mot clé « virtual »
 - On parle de méthodes virtuelles



Le voir pour y croire...

- La liaison tardive.

la méthode est éligible à la liaison tardive (dynamique)

```
class DeBase {
public:
    virtual void f() { cout << "DeBase" << endl; }
};
class SousClasse : public DeBase {
public:
    virtual void f() { cout << "SousClasse" << endl; }
};
void uneFonction(DeBase &o) { o.f(); }
int main() {
    SousClasse unObjet;
    unObjet.f();
    DeBase instanceDeBase = unObjet;
    instanceDeBase.f();
    SousClasse *pSousClasse = &unObjet;
    pSousClasse->f();
    DeBase *pBase = pSousClasse;
    pBase->f();
    uneFonction(unObjet);
    return 0;
}
```

```
[Gibi:~]% ./main
SousClasse
DeBase
SousClasse
SousClasse
SousClasse
[Gibi:~]%
```

Le passage par référence permet la liaison tardive

Classe polymorphe

- Une classe dont la définition contient au moins une méthode à liaison tardive est appelée ***classe polymorphe***
 - Polymorphisme par sous-typage



Un point sur la construction...

- Qu'advient-il des constructions par copie en liaison avec l'héritage...
 - De façon surprenante (?), lorsqu'une classe dérivée définit un constructeur par copie, C++ n'appelle pas automatiquement le constructeur de copie de la classe de base (même s'il existe)...
 - Si rien n'est indiqué, un appel au constructeur sans paramètre de la classe de base est tenté...

Illustration

```
class A {
    public:
        A() { cout << "A()" << endl; }
        A(const A &a) { cout << "A(const A&" << endl; }
};

class B : public A {
    public:
        B() { cout << "B()" << endl; }
        B(const B &b) { cout << "B(const &B)" << endl; }
};

void f(B unObjet)
{

}

int main()
{
    B b;
    cout << "Appel a f(B)" << endl;
    f(b);
    return 0;
}
```

rien
d'indiqué

```
[Gibi:~]% ./main
A()
B()
Appel a f(B)
A()
B(const &B)
[Gibi:~]%
```

Appel explicite de ctor

```
class A {
public:
    A() { cout << "A()" << endl; }
    A(const A &a) { cout << "A(const A&)" << endl; }
};

class B : public A {
public:
    B() { cout << "B()" << endl; }
    B(const B &b) : A(b) { cout << "B(const &B)" << endl; }
};

void f(B b) {

int main()
{
    B b;
    cout << "Appel a f(B)" << endl;
    f(b);
    return 0;
}
```

explicite...

```
[Gibi:~]% ./main
A()
B()
Appel a f(B)
A(const A&)
B(const &B)
[Gibi:~]%
```

...et sur la destruction

- Où l'on répond enfin à la question « Pourquoi les dtor doivent-ils être qualifiés de virtual ? »
 - Afin que ce soit toujours le bon destructeur qui soit appelé en cas de polymorphisme

Illustration

```
class Individu {
public:
    virtual string donneTonNom()=0; // à implémenter quelque part
    ~Individu() { cout << "~Individu()" << endl; };
};
class Femme : public Individu {
public:
    Femme(string nom) : Individu(nom) {};
    ~Femme() { cout << "~Femme(" << donneTonNom() << ")" << endl; }
};

void libere(Individu *pi) {
    delete pi;
}

int main()
{
    Femme f("Georgette");
    Femme *pf = new Femme("Pascale");
    libere(pf);
    return 0;
}
```

```
[yunes] ./main
~Individu()
~Femme(Georgette)
~Individu()
[yunes]
```

Illustration

```
class Individu {
public:
    virtual string donneTonNom() = 0; // à implémenter quelque part
    virtual ~Individu() { cout << "~Individu()" << endl; };
};
class Femme : public Individu {
public:
    Femme(string nom) : _Individu(nom) {};
    virtual ~Femme() { cout << "~Femme(" << donneTonNom() << ")" << endl; }
};

void libere(Individu *pi) {
    delete pi;
}

int main()
{
    Femme f("Georgette");
    Femme *pf = new Femme("Pascale");
    libere(pf);
    return 0;
}
```

```
[yunes] ./main
~Femme(Pascale)
~Individu()
~Femme(Georgette)
~Individu()
[yunes]
```


A oublier !

- Modification d'accessibilité dans le sens souhaité possible
 - Dangereux et irresponsable ☺
- Héritages non publics
 - Privés et protégés
 - Ne sont pas des sous-typages
 - Peuvent être utilisés pour faire de la composition
 - Oui, mais non !

Je ne vous montre pas ce tableau qui est partout...

Statut du membre hérité dans la classe dérivée		Droit d'accès du membre dans la classe de base		
Héritage		private	protected	public
	private	private	private	private
	protected	private	protected	protected
	public	private	protected	public

Vous ne l'avez pas vu !

En résumé...

- La spécialisation permet de créer un ou plusieurs sous-types exploitant une relation de type « est une sorte de »
- Test : « is a kind of » Lien de spécialisation a priori
 - A compléter par un second test pour valider la relation
 - « Y a-t-il au moins une propriété/méthode qui ne s'applique pas à la classe dérivée ? »
- Mais, est-ce que ça peut « fonctionner à l'envers » ?
 - Oui, on parle de factorisation / généralisation

Factorisation d'implémentation

- La factorisation d'implémentation/de réalisation conduit à la construction de classes incomplètes, de ***réalisations partielles***
 - Elle consiste à réunir en une même unité d'encapsulation des actions (avec une partie de leur implémentation) et des attributs communs
- C'est l'***héritage comme moyen de réutiliser*** une implémentation

Pour faire simple


- Après analyse (voire après définition parfois) :
 - On identifie pour plusieurs classes suffisamment de points communs pour qu'elles puissent être considérées comme descendantes d'une classe mère
 - Inexistante pour le moment
- Il est alors possible de créer cette classe mère et y « déplacer » les éléments communs
 - Avec mise en place ou non de polymorphisme
 - Méthodes virtuelles


Factorisation conceptuelle


- aka ***généralisation***
- Conduit à l'apparition d'abstractions (types abstraits)
 - Notion ***d'interface***
- Consiste (principalement) à réunir ***au moment de l'analyse conceptuelle*** des actions communes dans une même unité d'encapsulation

Point de départ

- Classes concrètes identifiées
 - Pas nécessairement définies
 - C'est d'ailleurs mieux si elles ne le sont pas...


 Thermomètre
<i>Attributes</i>
<i>Operations</i> + calibrer() : void + mesurer() : double


 TubeDePitot
<i>Attributes</i>
<i>Operations</i> + calibrer() : void + mesurer() : double


 CompteurGeiger
<i>Attributes</i>
<i>Operations</i> + calibrer() : void + mesurer() : double

Point de départ

- Classes concrètes identifiées
 - Pas nécessairement définies
 - C'est d'ailleurs mieux si elles ne le sont pas...

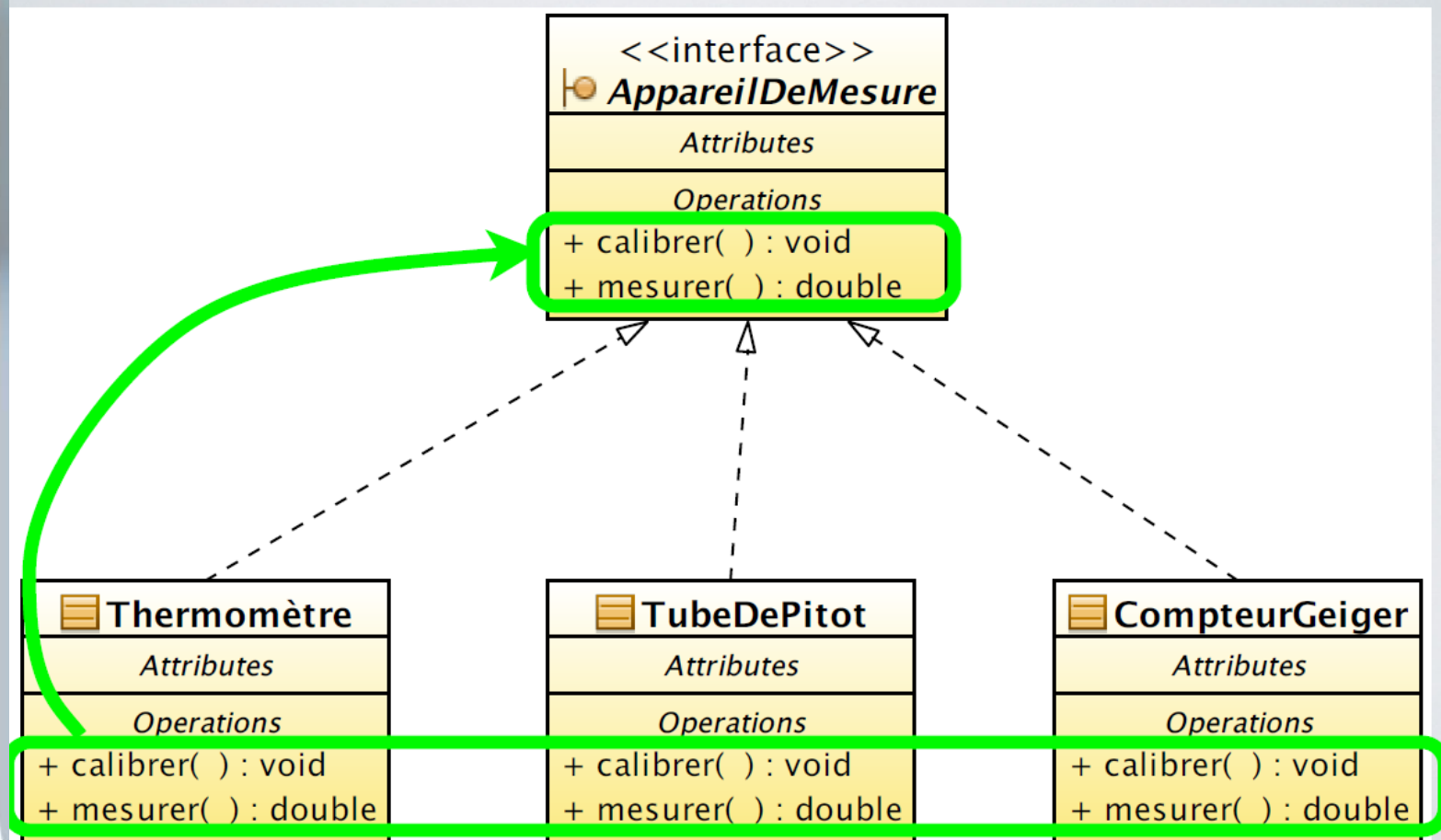
 Thermomètre
<i>Attributes</i>
<i>Operations</i>
+ calibrer() : void + mesurer() : double

 TubeDePitot
<i>Attributes</i>
<i>Operations</i>
+ calibrer() : void + mesurer() : double

 CompteurGeiger
<i>Attributes</i>
<i>Operations</i>
+ calibrer() : void + mesurer() : double

Point d'arrivée

- On « remonte » le facteur commun



En C++

```
class AppareilDeMesure {  
    public:  
        virtual void calibrer() = 0; // équiv. C++ du abstract de Java  
        virtual double mesurer()= 0;  
};
```

```
class CompteurGeiger : public AppareilDeMesure {  
    public:  
        virtual void calibrer() { /* remettre à zéro */ }  
        virtual double mesurer() { /* compter des particules */ }  
};
```

```
class TubeDePitot : public AppareilDeMesure {  
    public:  
        virtual void calibrer() { /* remettre à zéro */ }  
        virtual double mesurer() { /* soustraire des pressions */ }  
};
```

```
class Thermomètre : public AppareilDeMesure {  
    public:  
        virtual void calibrer() { /* laisser refroidir */ }  
        virtual double mesurer() { /* attendre la stabilisation */ }  
};
```



Prenons le temps de poser les choses...

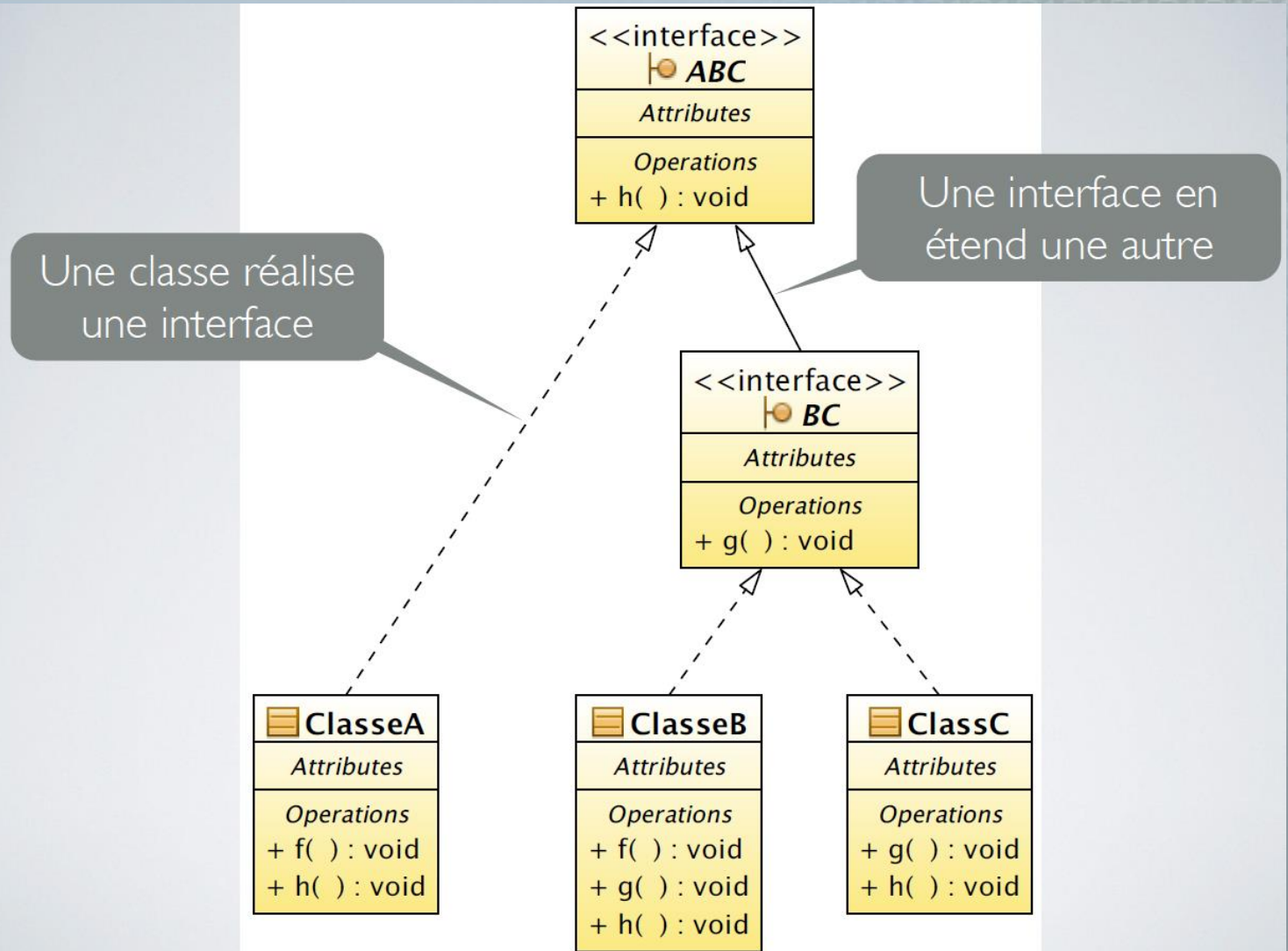
- virtual type Id (...) = ***0***
- Est la déclaration d'une fonction membre d'instance virtuelle pure (abstraite)
 - Qui ne peut être définie au niveau conceptuel
- Donc
 - La classe n'est pas instanciable
 - La méthode devra être redéfinie
- Remarque :
 - Même si la classe n'est pas instanciable on peut déclarer un pointeur ou une référence de ce type



Points d'attention

- La factorisation n'est pas toujours simple
- Non unique en général
 - Attention à garder du sens
 - Pas une seule bonne solution
- La factorisation est essentielle à la conception
 - Conduit à créer des « sur-types »

Petit point de sémantique



C'est fini pour ça !

- La suite sur ce thème :
 - Du modèle à la classe
 - Du diagramme de classes au codage en C++
- Mais tout de suite, la gestion des erreurs en C++
 - Et en plus, ça met en œuvre la spécialisation !

