

# LE LANGAGE C++

## MASTER 1

### TYPES ET SOUS-TYPES

*Factorisation :  $16x^2+20x = 4.(4x+5)$*

Jean-Baptiste.Yunes@u-paris.fr  
U.F.R. d'Informatique  
Université de Paris

11/2021

# LA FACTORISATION CONCEPTUELLE

La factorisation conceptuelle conduit à l'apparition d'abstractions, *i.e.* de types abstraits, *i.e.* d'interfaces

- Elle consiste à réunir en une même unité d'encapsulation des actions communes
- La **factorisation conceptuelle** s'appelle la **généralisation**

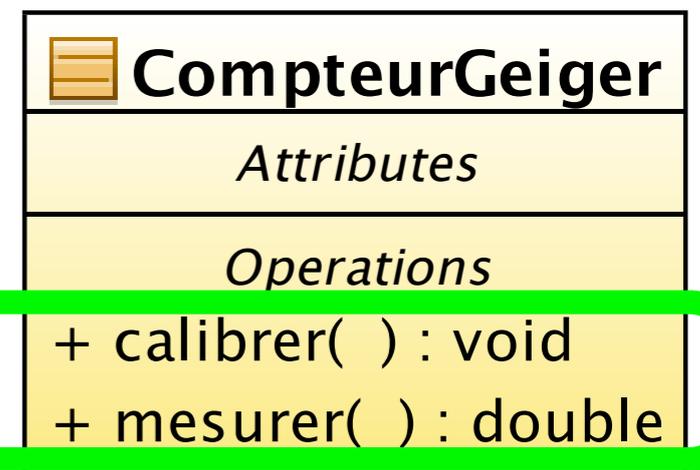
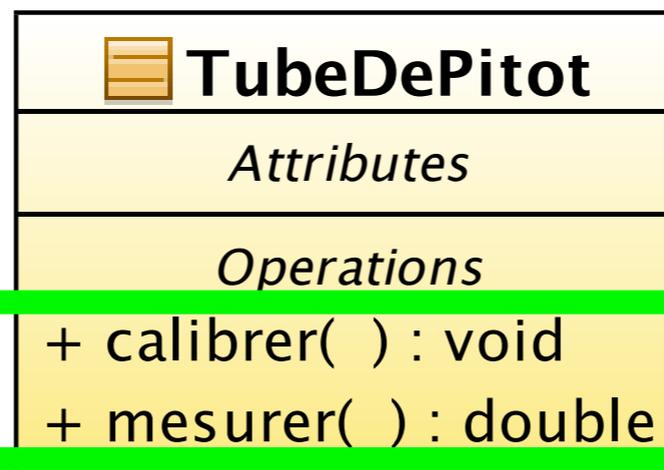
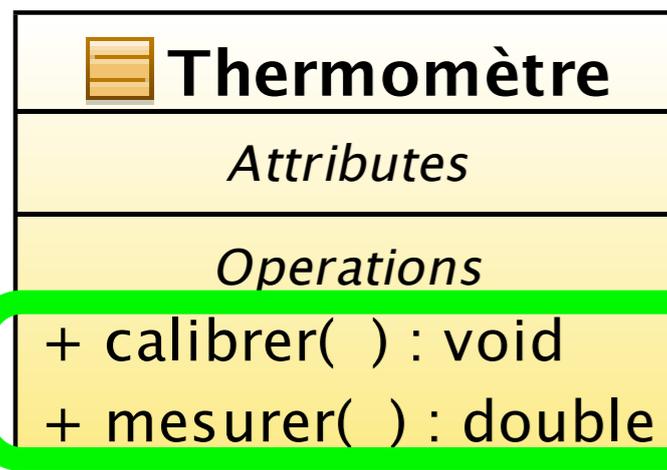
- Des classes concrètes identifiées...

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

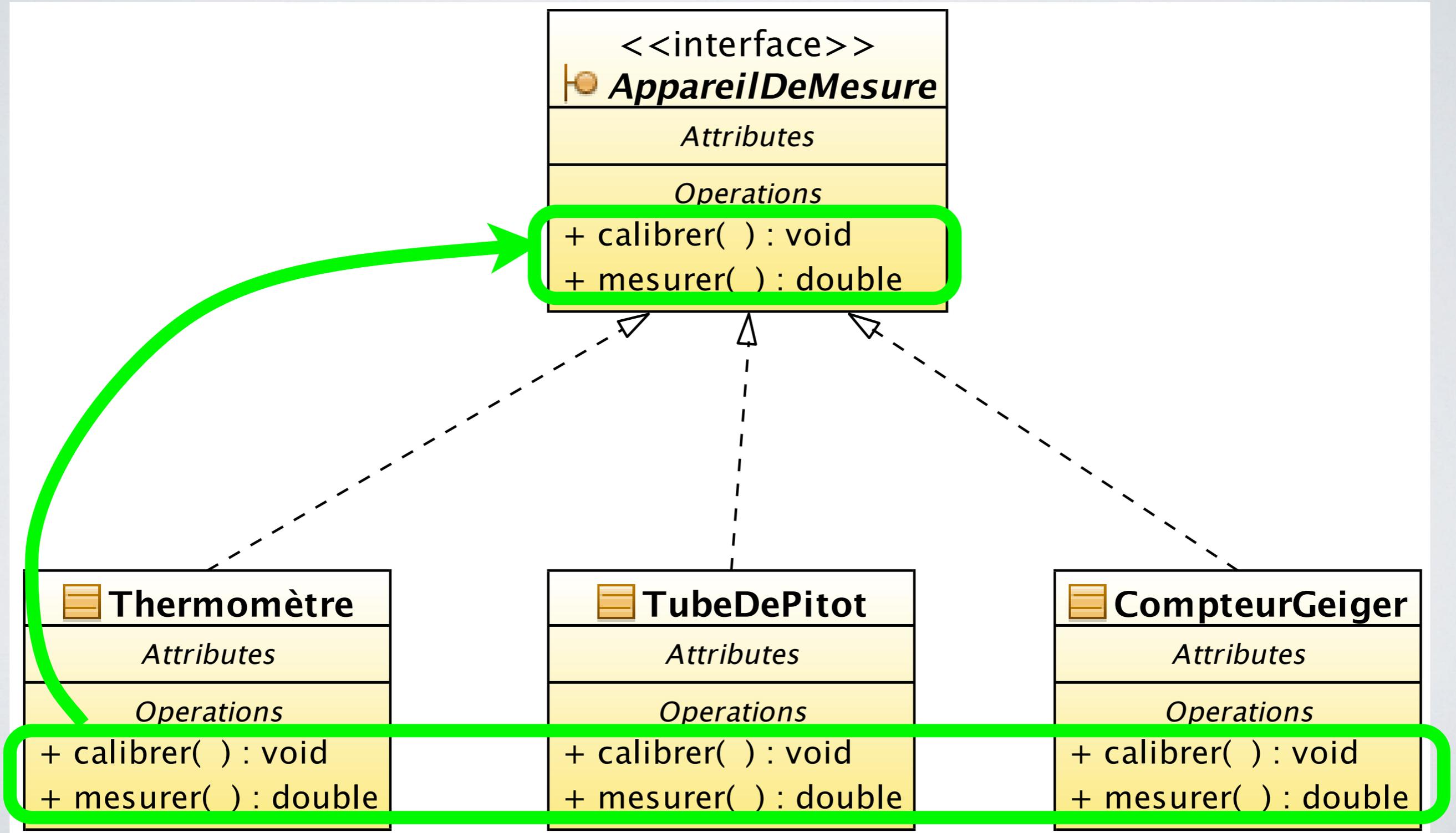
 <b>TubeDePitot</b>
<i>Attributes</i>
<i>Operations</i> + calibrer( ) : void + mesurer( ) : double

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

- Des classes concrètes identifiées...



- On « remonte » le facteur commun



- En C++ cela s'écrit

```
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 */ }  
};
```

- `virtual type identificateur(type identificateur, ...) = 0`
- est la déclaration d'une fonction membre d'instance **virtuelle pure** (abstraite)
- qui ne peut être définie à ce niveau **conceptuel** (enfin c'est plus subtil en réalité)
- donc
  - la classe n'est pas instanciable
  - la fonction devra être (re)définie

- Une classe contenant une fonction virtuelle pure est une classe abstraite
- La classe n'est pas instanciable mais on peut déclarer :
  - une référence de ce type
  - un pointeur de ce type

```
void faitFonctionner(AppareilDeMesure &a) {
    a.mesurer();
}

void faitFonctionner(AppareilDeMesure *pa) {
    pa->mesurer();
}

int main() {
    Thermomètre t;
    faitFonctionner(t);
    faitFonctionner(&t);
    SondePitot s;
    faitFonctionner(s);
    faitFonctionner(&s);
    return 0;
}
```

```
[yunes] ./main
Thermo
Thermo
Pitot
Pitot
[yunes]
```

Attention la factorisation n'est pas toujours simple

- Elle n'est en général **pas unique**
- Attention à factoriser en conservant du sens, ce qui est loin d'être toujours évident
- Il n'y a **pas une bonne solution**

Contrairement à la spécialisation, elle conduit à fabriquer des sur-types (est donc essentielle à la conception)

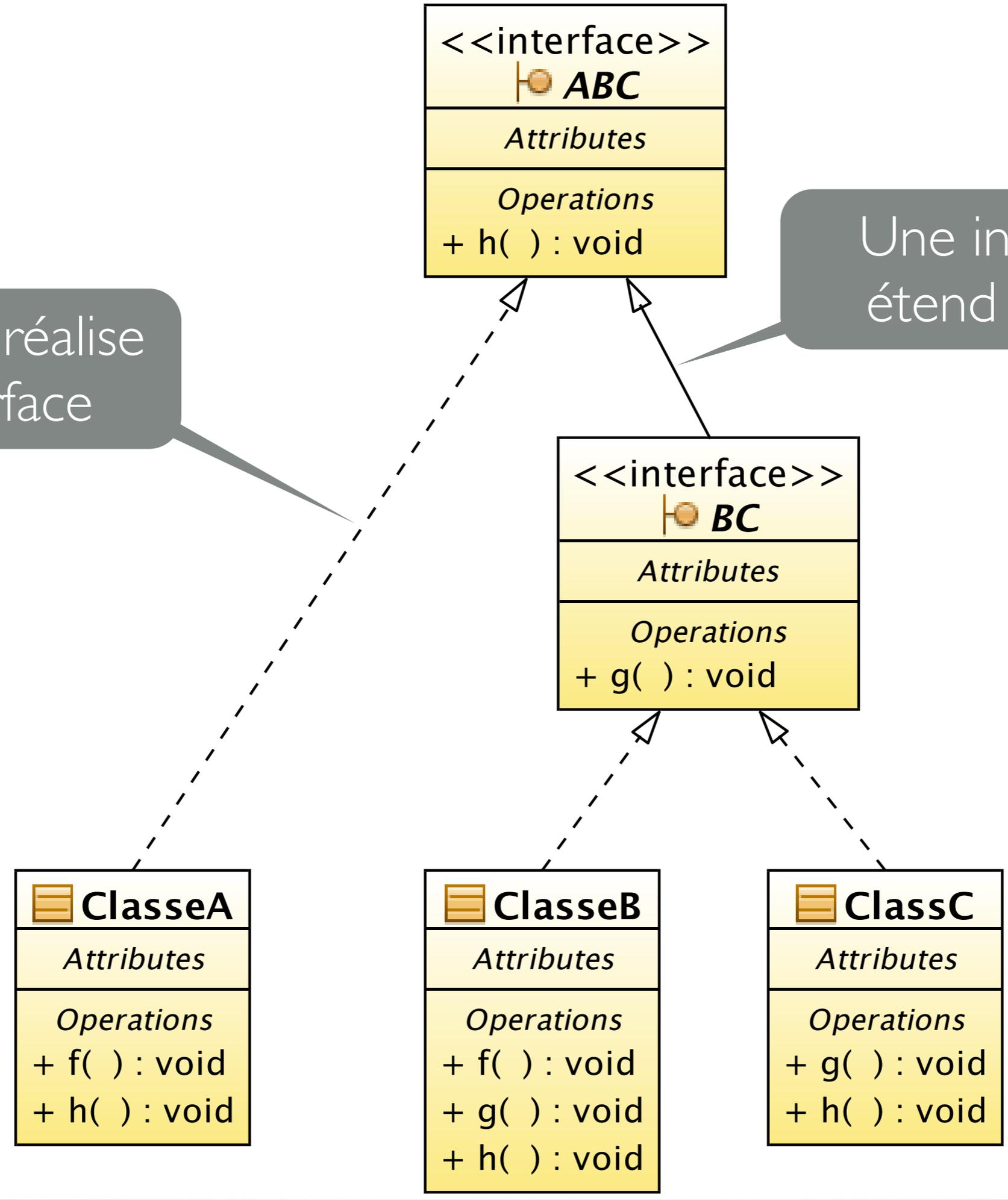
 <b>ClasseA</b>
<i>Attributes</i>
<i>Operations</i> + f( ) : void + h( ) : void

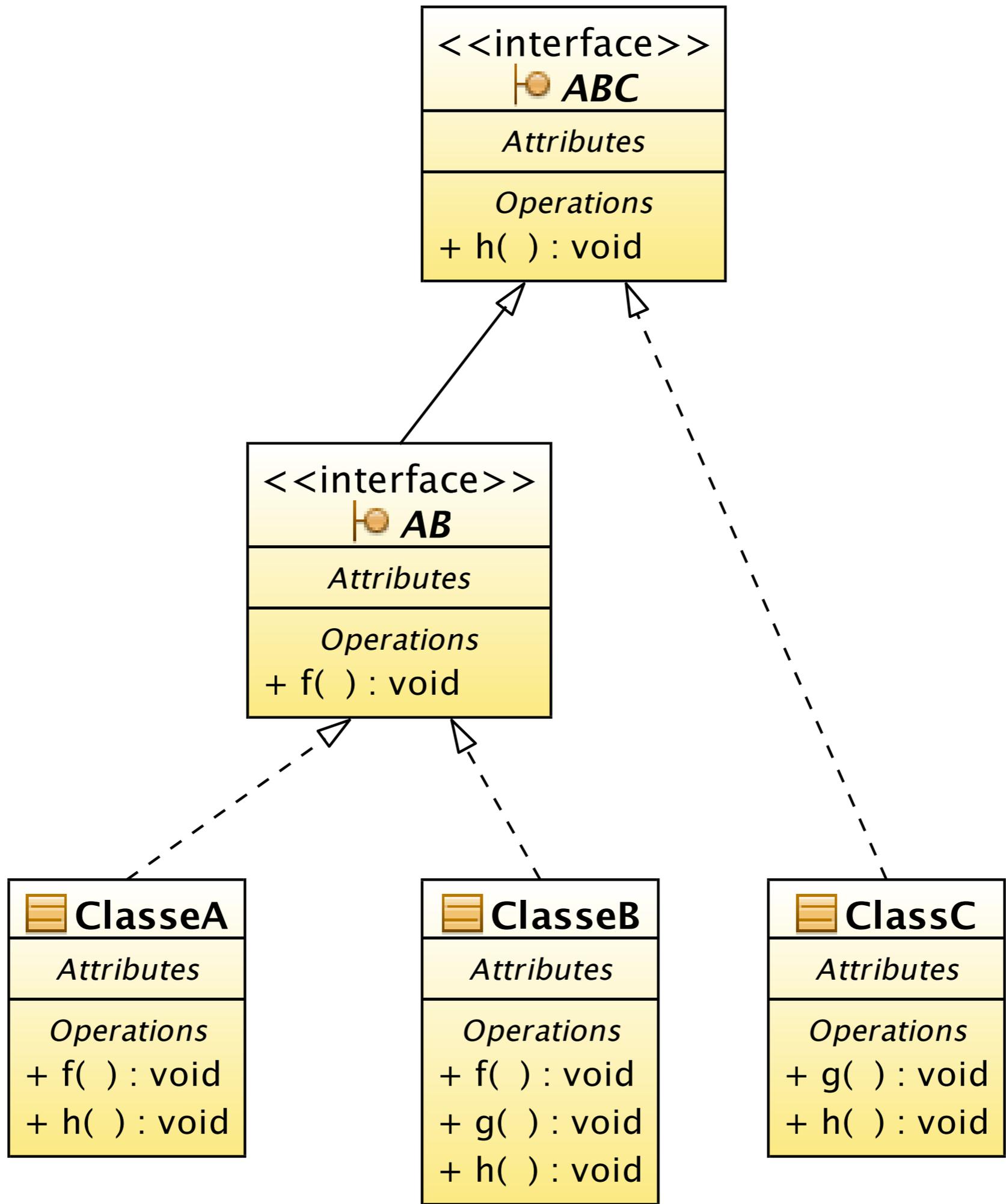
 <b>ClasseB</b>
<i>Attributes</i>
<i>Operations</i> + f( ) : void + g( ) : void + h( ) : void

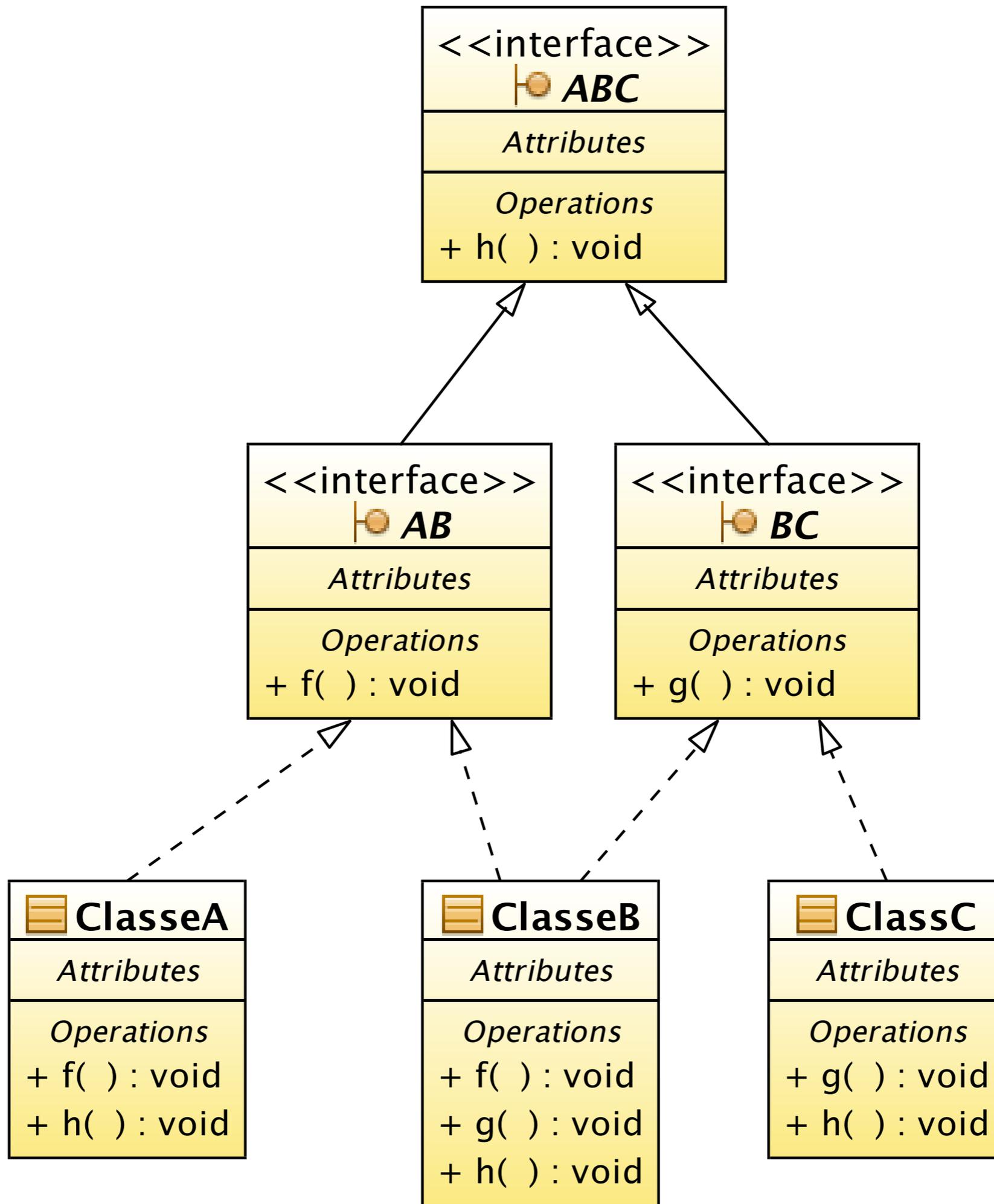
 <b>ClassC</b>
<i>Attributes</i>
<i>Operations</i> + g( ) : void + h( ) : void

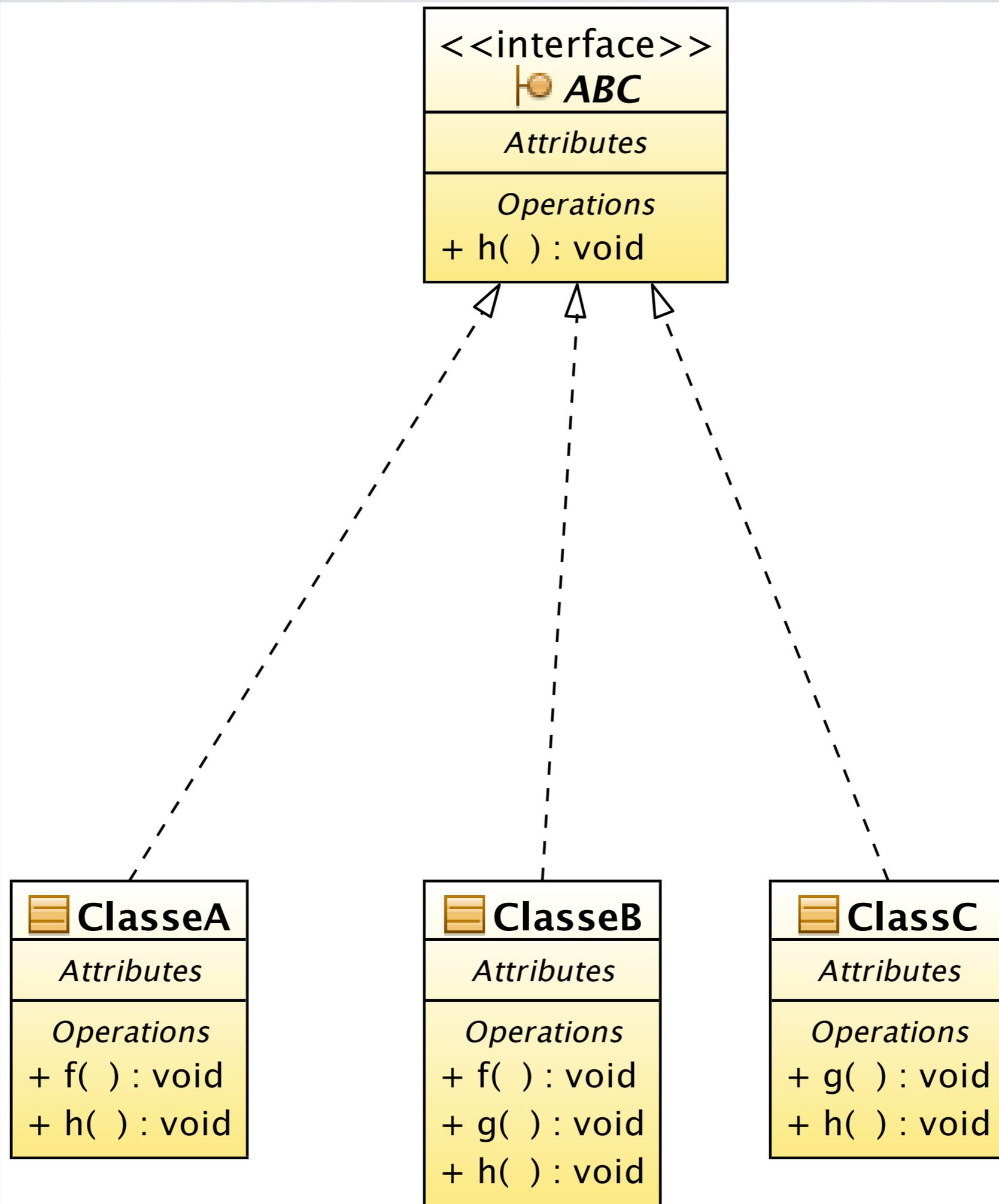
Une classe réalise une interface

Une interface en étend une autre





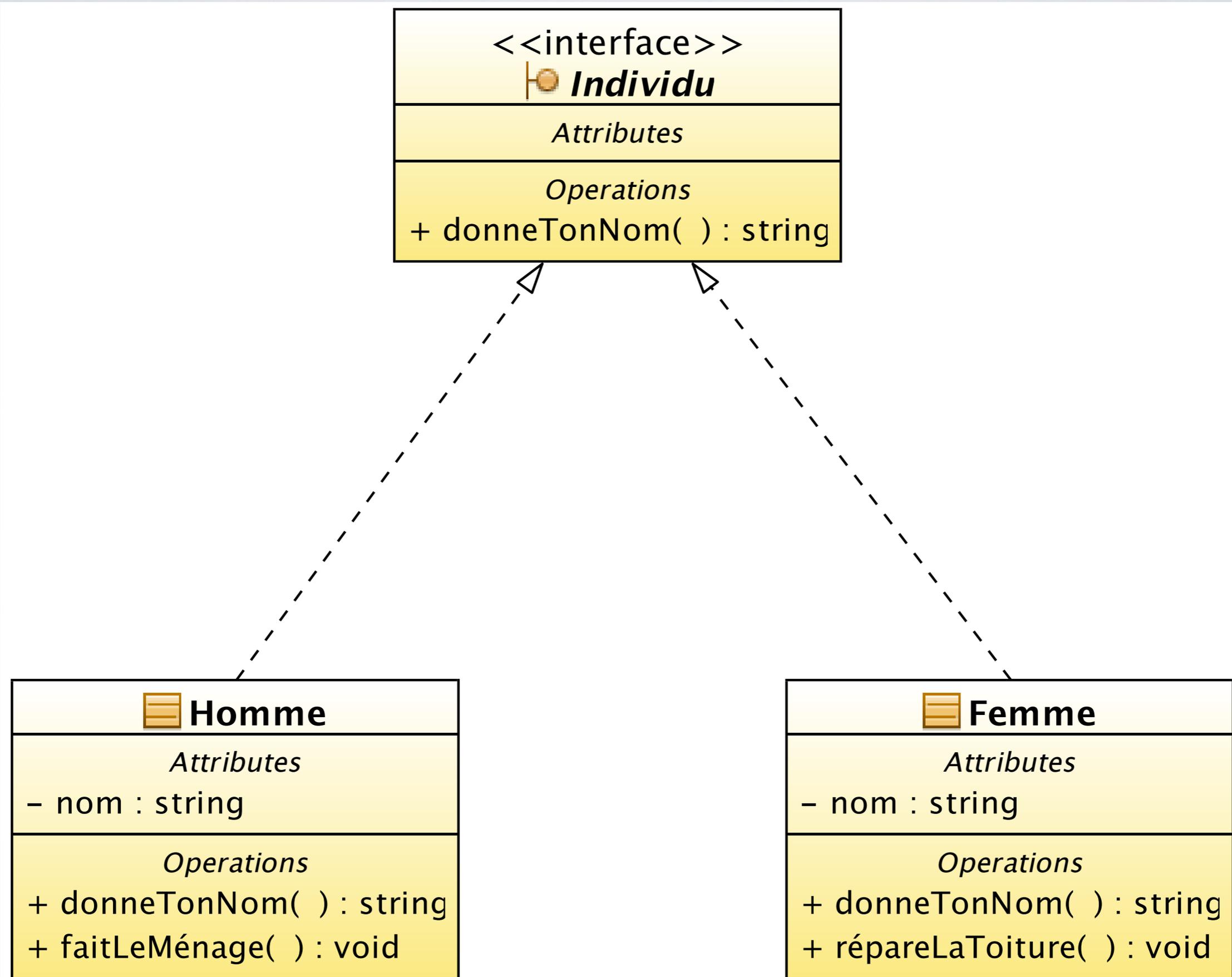




# LA FACTORISATION D'IMPLÉMENTATION

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



```
class Individu {
    public:
        virtual string donneTonNom()=0;
};
```

```
class _Individu : public Individu {
    private:
        string nom;
    public:
        virtual string donneTonNom() { return nom; }
};
```

```
class Homme : public _Individu {
    public:
        void faitLeMénage() { ... }
};
```

```
class Femme : public _Individu {
    public:
        void répareLeToit() { ... }
};
```

Factorisation de réalisation :  
i.e. `donneTonNom()`  
fonctionne à l'identique dans  
les deux classes

- à la réalisation

```
class Individu {
    public:
        virtual string donneTonNom()=0;
        virtual void prononceTonNom()=0;
};

class _Individu : public Individu {
    private:
        string nom;
    protected:
        virtual string donneTonNom() { return nom; }
};

class Homme : public _Individu {
    public:
        virtual void prononceTonNom() {
            SynthétiseurVocal::setMode(GRAVE);
            SynthétiseurVocal::synthétise(donneTonNom());
        }
};

class Femme : public _Individu {
    public:
        virtual void prononceTonNom() {
            SynthétiseurVocal::setMode(AIGU);
            SynthétiseurVocal::synthétise(donneTonNom());
        }
};
```

- à la réalisation

```
class Individu {
    public:
        virtual string donneTonNom()=0;
        virtual void prononceTonNom()=0;
};

class _Individu : public Individu {
    private:
        string nom;
    public:
        virtual string donneTonNom() { return nom; }
        virtual void prononceTonNom() { synthétiseurVocal.synthétise(donneTonNom()); }
};

class Homme : public _Individu {
    public:
        virtual void prononceTonNom() {
            synthétiseurVocal.setMode(GRAVE);
            _Individu::prononceTonNom();
        }
};

class Femme : public _Individu {
    public:
        virtual void prononceTonNom() {
            synthétiseurVocal.setMode(AIGU);
            _Individu::prononceTonNom();
        }
};
```

Factorisation partielle  
du comportement des  
méthodes concrètes...

# • à la réalisation

```
class Individu {
    public:
        virtual string donneTonNom()=0;
        virtual void prononceTonNom()=0;
};

class _Individu : public Individu {
    private:
        string nom;
    protected:
        void prononceTonNom(mode m) {
            synthétiseurVocal.setMode(m);
            synthétiseurVocal.synthétise(donneTonNom());
        }
    public:
        virtual string donneTonNom() { return nom; }
};

class Homme : public _Individu {
    public:
        virtual void prononceTonNom() {
            prononceTonNom(GRAVE);
        }
};

class Femme : public _Individu {
    public:
        virtual void prononceTonNom() {
            prononceTonNom(AIGU);
        }
};
```

## Attention :

- les classes intermédiaires ne doivent jamais être employées comme type (elles ne sont là que pour faciliter la réalisation des types concrets)...
- inconvénient : pour comprendre comment fonctionne un type concret donné, il faut lire le code de toutes les réalisations partielles faites...
- la conception de telles classes doit être pensée avant l'écriture du code (attribut `virtual`)

**NOTE SUR LA  
DESTRUCTION DES OBJETS**

Attention, quoique bizarre :

- les destructeurs des classes **doivent** **être** qualifiés de `virtual`
- ceci afin que le bon destructeur soit appelé en cas de polymorphisme

```

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]

```

```

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]

```

# CONVERSIONS POLYMORPHES

## Conversions :

- conversion vers le haut (*upcast*)
- objet (object slicing - découpage)
- pointeur
- référence
- conversion vers le bas (*downcast*)
- interdite sur les objets
- à contrôler sur les pointeurs
- à contrôler sur les références

L'opérateur `dynamic_cast<type>` permet **si *type* est polymorphe** (*i.e.* possède une méthode virtuelle) :

- d'obtenir une conversion vers un sous-type (*downcast*)
- un pointeur nul (0) si la conversion n'est pas correcte
- une exception (`bad_cast`) si la référence n'est pas correcte

## Downcast via des pointeurs...

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

class B : public A {
public:
    virtual void f() { cout << "B::f()" << endl; }
};

void f(A *a) {
    B *p = dynamic_cast<B *>(a);
    cout << p << endl;
}

int main() {
    B b;
    f(&b); // ca va marcher
    A a;
    f(&a); // ca va rater...
    return 0;
}
```

```
[yunes] ./main
0x7fff5fbff680
0
[yunes]
```

## Downcast via des références...

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

class B : public A {
public:
    virtual void f() { cout << "B::f()" << endl; }
};

void f(A &a) {
    B &b = dynamic_cast<B &>(a);
    b.f();
}

int main() {
    B b;
    f(b); // ca va marcher
    A a;
    f(a); // ca va rater...
    return 0;
}
```

```
[yunes] ./main
B::f()
terminate called after throwing an
instance of 'std::bad_cast'
    what():  std::bad_cast
Abort
[yunes]
```

- d'autres usages sont possibles avec l'héritage multiple (bientôt...)

# RTTI

## RunTime Type Identification



Rongeurs de Taille (Très) Inhabituelle, The Princess Bride, Rob Reiner

- Il est possible d'obtenir des informations sur le type des objets si nécessaire
- Note : sauf dans de très rares cas, le polymorphisme doit être employé, et la connaissance du type réel des objets doit être ignorée
- usage : débogage...

- L'opérateur `typeid(type)` ou `typeid(expression)` renvoie la référence sur un objet de type `type_info` représentant :
  - le *type* donné
  - le type de l'*expression* donnée

- Le type `type_info` :
  - deux `type_info` peuvent être comparés avec `==` ou `!=`
  - on peut récupérer le nom du type par emploi de la méthode `const char *name() const;`
  - ce type n'est pas instanciable ni copiable
- Pour être utile, les types considérés doivent être polymorphes

**En conclusion à n'employer que pour déboguer...**