

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

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

# Plan de la séance :

- Les classes abstraites
- Les méthodes "deleted"
- Les exceptions
- Les classes internes
- SFML (une librairie graphique)
- les énumérations
- correction du TP noté 2022


# Retour sur l'héritage


Point de vue :


factorisation conceptuelle / généralisation

Cela consiste à faire apparaître des abstractions (des types abstraits, des interfaces) par exemple lors d'une refonte du code




Des classes concrètes identifiées...

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

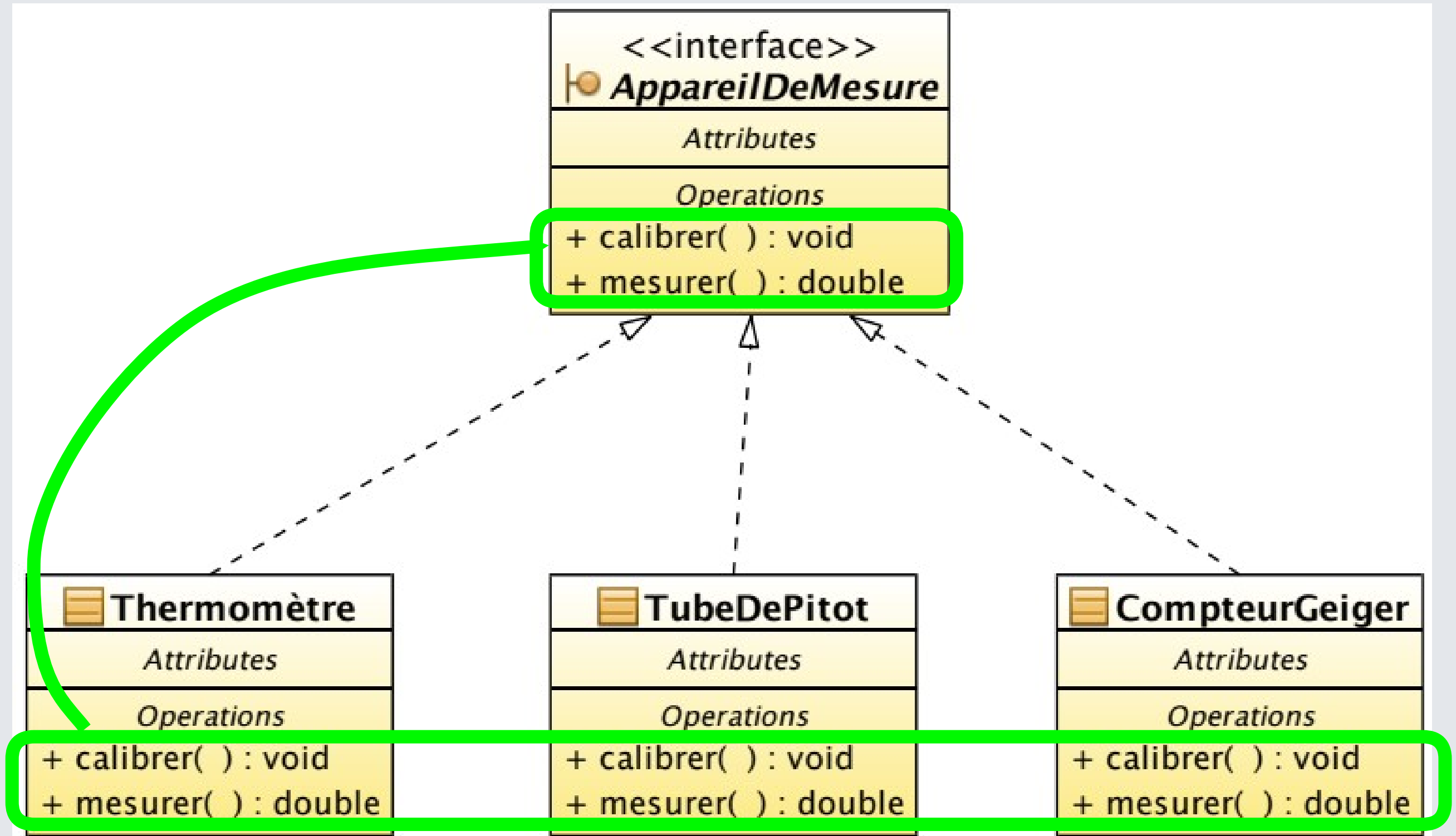
 <b>TubeDePitot</b>
<i>Attributes</i>
<i>Operations</i> + calibrer( ) : void + mesurer( ) : double

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

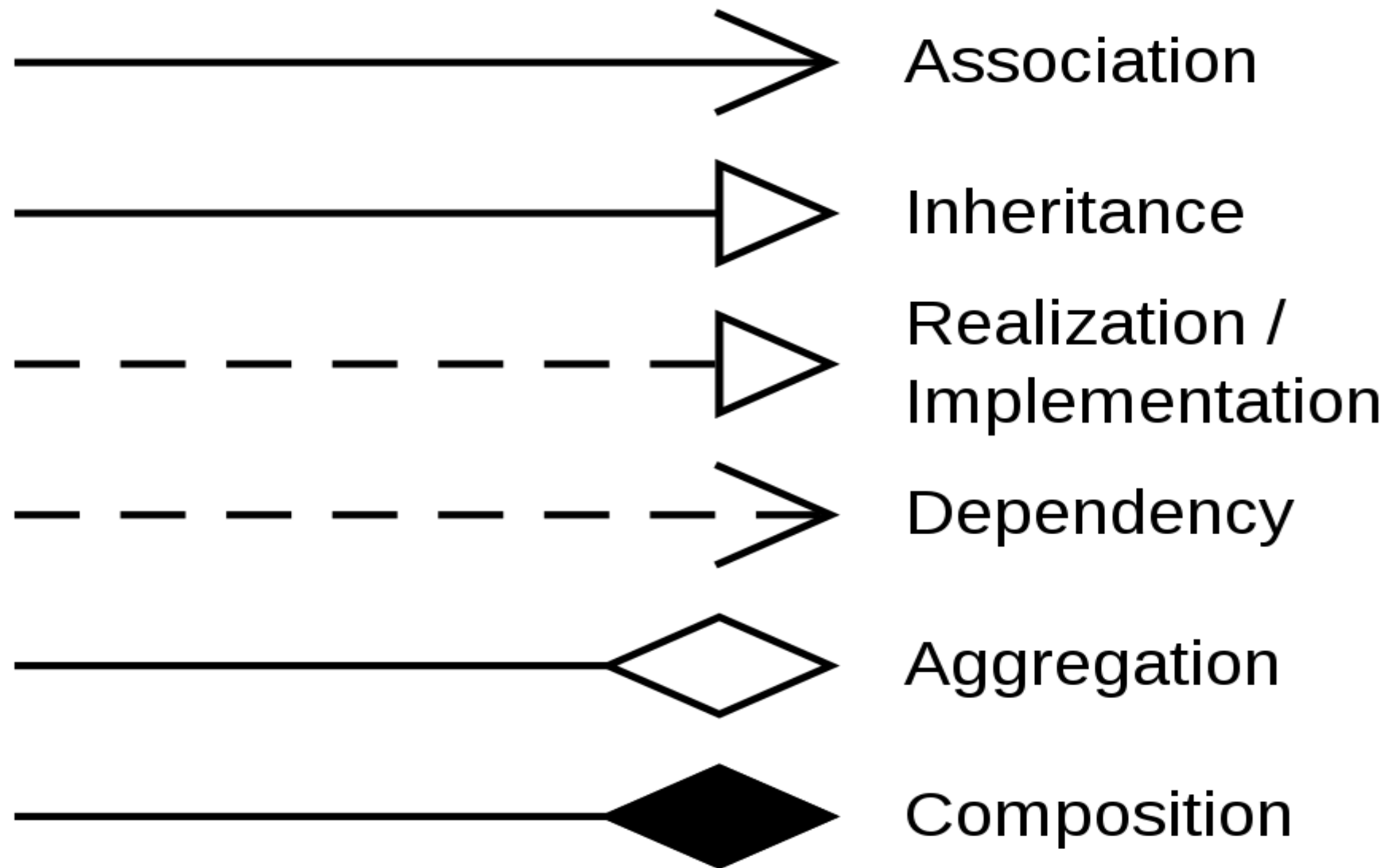
Des classes concrètes identifiées...

 Thermomètre	 TubeDePitot	 CompteurGeiger
Attributes	Attributes	Attributes
Operations	Operations	Operations
+ calibrer( ) : void + mesurer( ) : double	+ calibrer( ) : void + mesurer( ) : double	+ calibrer( ) : void + mesurer( ) : double

# Les facteurs communs peuvent définir une interface



# Rappel/complément : les symboles UML



# En C++ cela s'écrit

ce =0 est l'équivalent d'abstract en java

```
class AppareilDeMesure {  
    public:  
        virtual void calibrer() = 0; //  
        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 */ }  
};
```



# En C++ cela s'écrit

ce =0 est l'équivalent d'abstract en java

```
class AppareilDeMesure {  
    public:  
        virtual void calibrer() = 0; //  
        virtual double mesurer() = 0;  
};
```

virtual est évident  
puisque la liaison  
tardive est souhaitée

```
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 */ }  
};
```

# En C++ cela s'écrit

ce =0 est l'équivalent  
d'abstract en java  
on parle de **virtuelle pure**

```
class AppareilDeMesure {  
public:  
    virtual void calibrer() = 0; //  
    virtual double mesurer()= 0;  
};
```

virtual est évident  
puisque la liaison  
tardive est souhaitée

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

les autres virtual  
peuvent être implicites

```
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 */ }  
};
```

Une classe qui n'est que déclarative de virtuelle pures s'appelle une **interface**


Une classe contenant au moins une fonction virtuelle pure est une **classe abstraite**


Elles ne sont pas instanciable directement, mais on peut déclarer une référence de ce type, ou un pointeur vers ce type


```
int main() {  
    Thermomètre t;  
    AppareilDeMesure &x{t}, *px{&t}  
    x.mesurer();  
    px -> mesurer();  
}
```

La factorisation conduit à fabriquer des sur-types : elle est essentielle à la conception

Elle n'est en général pas unique.

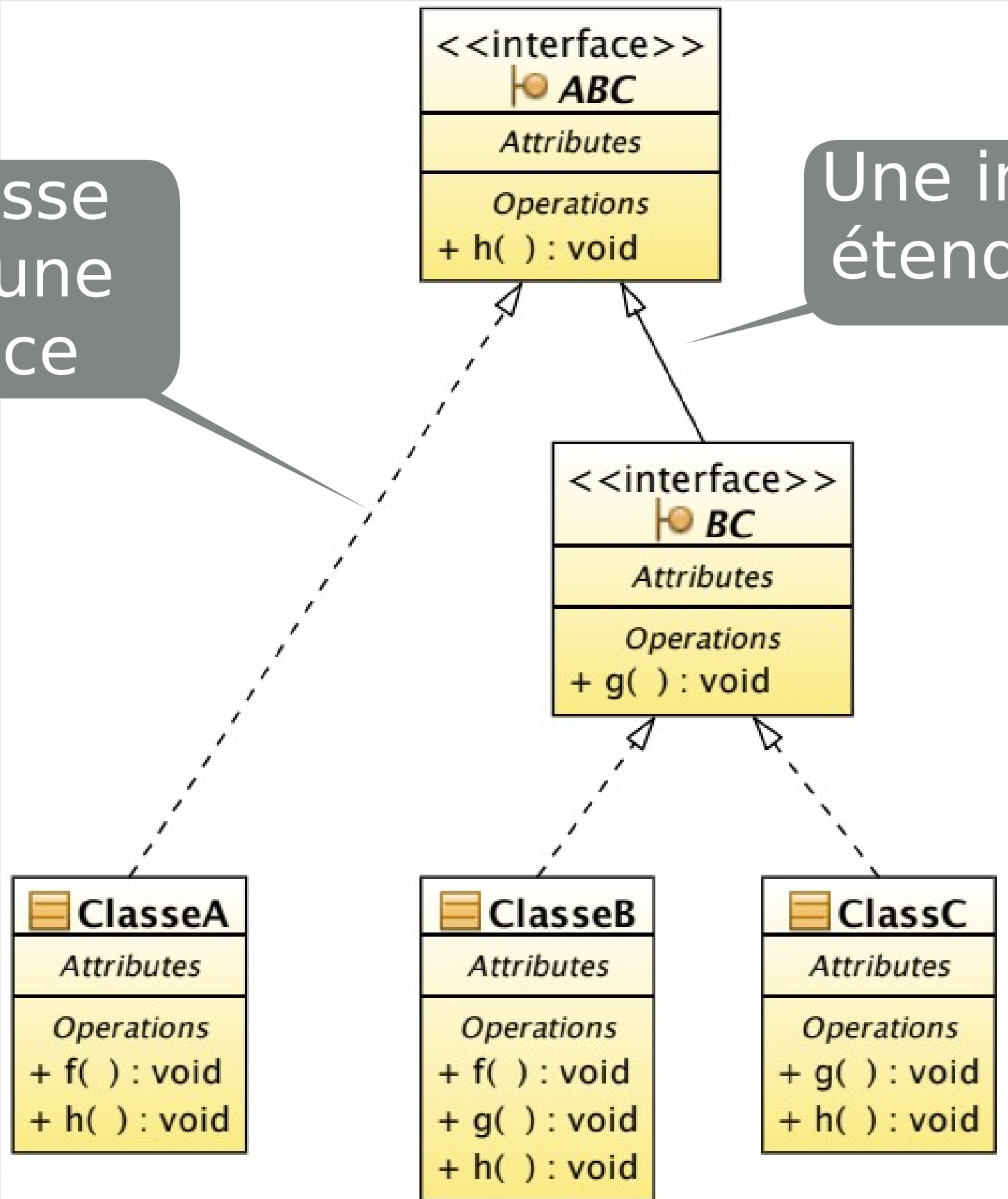
 <b>ClasseA</b>
<i>Attributes</i>
<i>Operations</i> + f( ) : void + h( ) : void

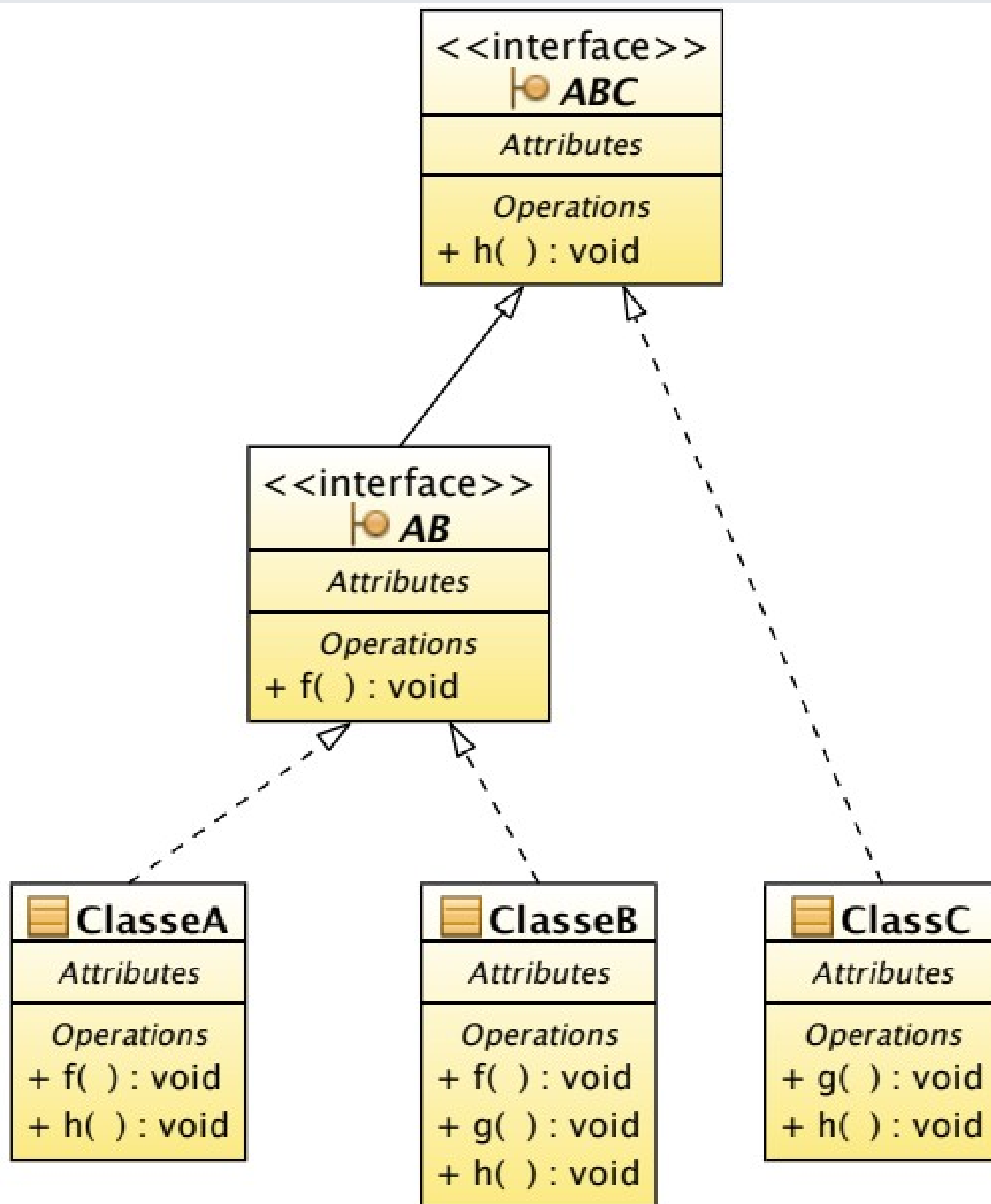
 <b>ClasseB</b>
<i>Attributes</i>
<i>Operations</i> + f( ) : void + g( ) : void + h( ) : void

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

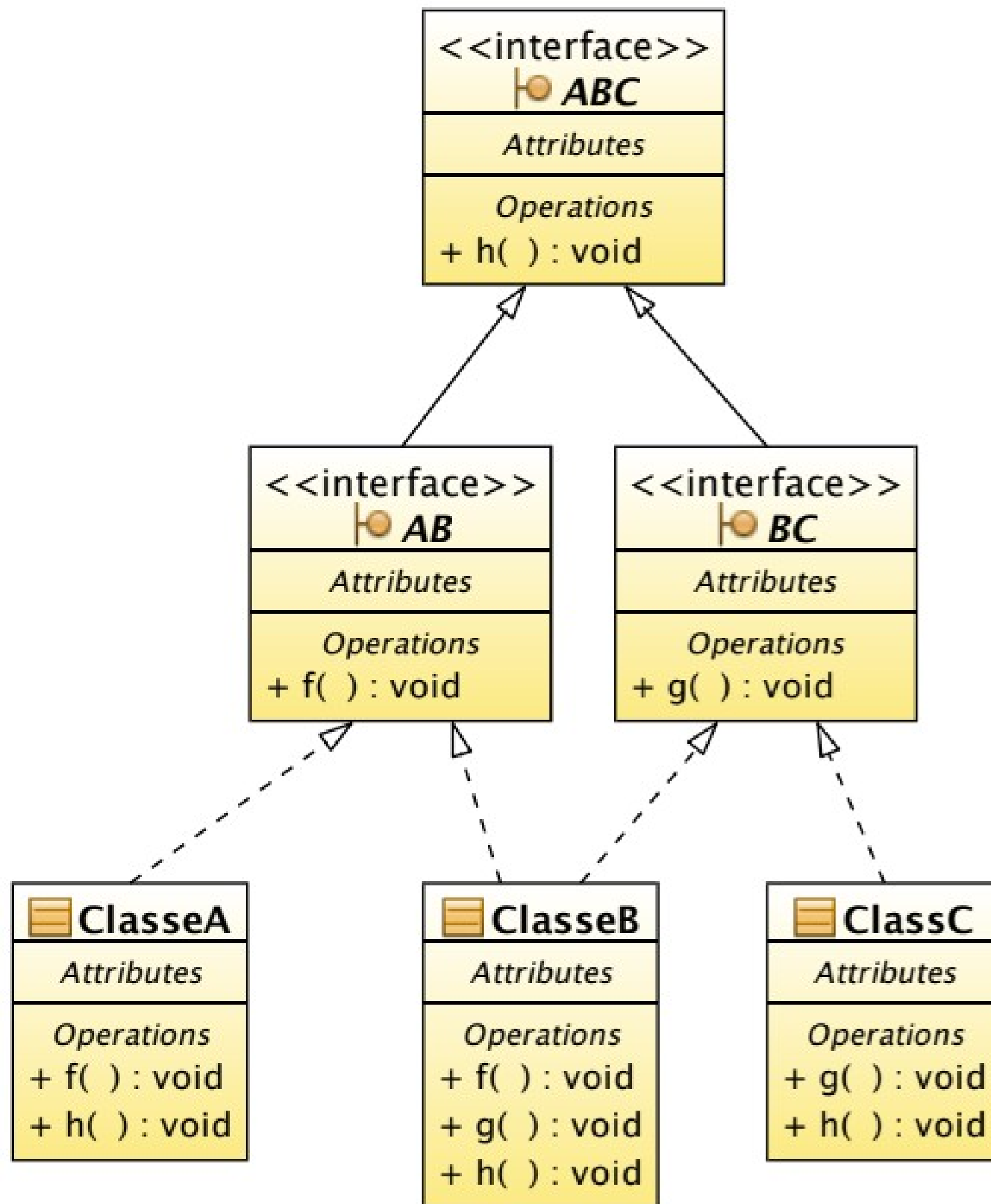
Une classe réalise une interface

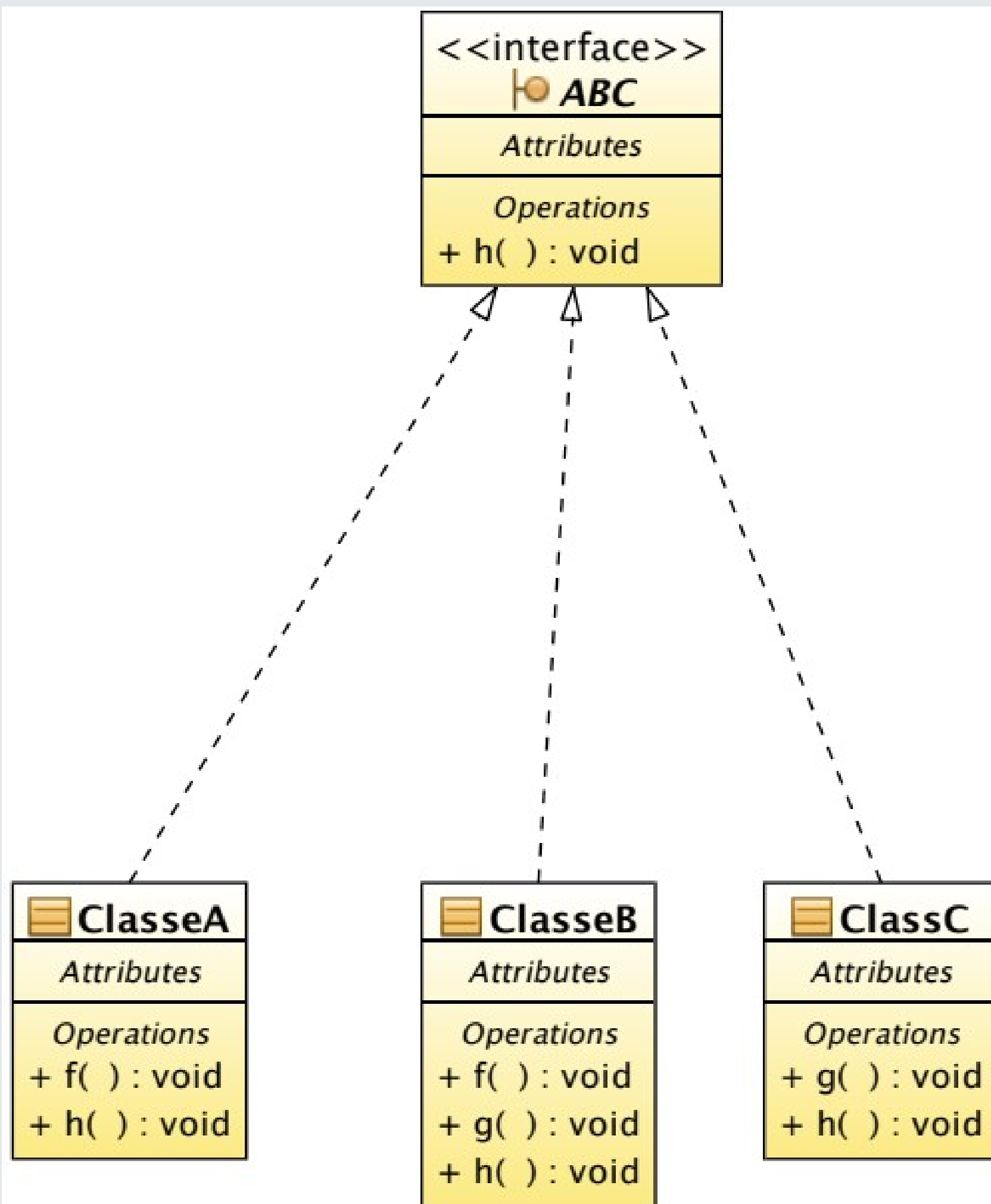
Une interface en étend une autre











On peut factoriser dans des classes incomplètes (c.a.d abstraites) des réalisations partielles, et des attributs communs ... mais cela nous mène à l'héritage multiple (qu'on verra à la rentrée)

# Les méthodes deleted

pour "interdire" l'usage de méthodes, on peut penser à les déclarer privées.

Mais ce n'est pas tout à fait satisfaisant, car parfois, même en interne on peut vouloir que des méthodes n'aient pas d'existence.

C'est possible en les déclarant avec  
=delete

# Les méthodes deleted

```
Class A {  
    A (const A&)=delete;  
};
```

```
int main() {  
    A a;  
    A b{a}; // impossible car deleted  
}
```

# Les méthodes deleted

```
Class A {  
    private : A (const A&) ;  
};
```

```
int main() {  
    A a;  
    A b{a}; // impossible car private  
}
```

on pouvait avoir quasiment le même résultat avec private

# Les méthodes deleted

```
Class A {  
    private : A (const A &);  
    void f();  
};
```

```
A::A(const A& x) {}  
void A::f() {  
    A y{*this};    // copie possible qd même  
}
```

on pouvait avoir quasiment le même résultat avec private, avec une petite nuance

# **Les méthodes deleted**

les usages pour `=delete` sont essentiellement pour supprimer des méthodes implicitement ajoutées par le compilateur : copie, affectation (et casting). Pensez-y dans le TP noté.



# **Les méthodes deleted**

les usages pour `=delete` sont essentiellement pour supprimer des méthodes implicitement ajoutées par le compilateur : copie, affectation (et casting). Pensez-y dans le TP noté.

Naturellement `=delete` ne se combine pas avec `=0`. Pour info, il existe aussi un `=default` ...

**Quelques mots sur les exceptions :**

# Quelques mots sur les exceptions :

Idée : séparer dans le code

- les instructions qui représentent la partie *fonctionnellement intéressante* du programme
- les instructions qui servent à traiter/corriger les erreurs rencontrées et qui empêchent de continuer normalement

# Les exceptions dans le monde réel :

// planning de votre vie quotidienne

blah

blah

blah

blah

blah

blah

blah

blah

blah

blah

blah

blah

## **Consignes en cas d'accident**

tout arrêter,

préserver ce qui peut l'être

réparer les dégâts

reprendre à ce point là

C'est un mécanisme de contrôle du flux d'exécution

- soit une fonction réalise correctement son travail : elle **termine** et **renvoie** une valeur
- soit quelque chose l'en empêche : on **sort précipitamment** de la fonction **en identifiant une erreur**



Il y a donc deux mécanismes d'exécution :

- le flux normal
- le flux de récupération des erreurs

et un moyen de basculer de l'un à l'autre :

- lancement d'une exception
- capture d'une exception

# Une exception

- représente une erreur détectée
- exprime la fiche d'incident qui contient les informations nécessaire pour envisager de *réparer*

Le déclenchement s'effectue par :

`throw qq chose;`

En c++ tout peut être utilisé comme exception. En général on construit un objet pour l'occasion.

`throw UneClasse{...};`



Au lancement de l'exception,  
l'exécution normale s'interrompt et  
une recherche de reprise sur erreur  
est effectuée :

il y a remontée de la pile des appels  
en détruisant les objets temporaires  
jusqu'à trouver une reprise sur  
erreur adéquate ou atteindre `main()`

Un code faisant appel à des fonctions pouvant lever des exceptions peut :

**les ignorer** s'il n'est pas en situation de pouvoir les corriger

**les corriger** (sachant qu'il a essayé)

```
try {  
    bloc d'instructions dans lequel  
    des exceptions peuvent se produire  
}  
catch (TypeErreur1 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur produite de TypeErreur1  
}  
catch (TypeErreur2 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur de TypeErreur2  
}  
catch (...) {  
    bloc d'instructions de traitement  
    des erreurs des Types non cités  
}  
// suite du programme
```

(...) est la syntaxe  
pour capturer les  
autres sortes  
d'exceptions

```
try {  
    bloc d'instructions dans lequel  
    des exceptions peuvent se produire  
}  
catch (TypeErreur1 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur produite de TypeErreur1  
}  
catch (TypeErreur2 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur de TypeErreur2  
}  
catch (...) {  
    bloc d'instructions de traitement  
    des erreurs des Types non cités  
}  
// suite du programme
```

(...) est la syntaxe  
pour capturer les  
autres sortes  
d'exceptions

Un catch au plus est choisi (selon l'ordre de leur écriture )

Si le traitement est complet, l'exécution reprend son cours normal à la 1ère instruction qui suit tous ces catches

En cas de throw dans le traitement, il ne concerne plus ce bloc :  
il remonte dans la pile des appels...



# Qqs petites choses

```
try { // ...  
} catch (A erreur) { // ... }
```

ici une copie est faite

```
try { // ...  
} catch (A & erreur) { // ... }
```

# Qqs petites choses

```
try { // ...  
} catch (A erreur) { // ... }
```

ici une copie est faite

```
try { // ...  
} catch (A & erreur) { // ... }
```

```
try { // ...  
} catch (...) { throw; }
```

throw seul : relance  
l'exception telle quelle

```
void f() {  
    A *x {new A{}};  
    if (test()) throw Erreur{"voulue"};  
    delete x;  
}
```

si erreur : l'allocation  
mémoire n'est pas rendue

si erreur : l'allocation  
mémoire n'est pas rendue

```
void f() {  
    A *x {new A{}};  
    if (test()) throw Erreur{"voulue"};  
    delete x;  
}
```

```
void f() {  
    A x {};  
    if (test()) throw Erreur{"voulue"};  
}
```

```
void f() {  
    A *x {new A()};  
    if (test()) {  
        delete x;  
        throw Erreur{"voulue"};  
    }  
    delete x;  
}
```



si erreur : l'allocation  
mémoire n'est pas rendue

```
void f() {  
    A *x {new A{}};  
    if (test()) throw Erreur{"voulue"};  
    delete x;  
}
```

```
void f() {  
    A x {};  
    if (test()) throw Erreur{"voulue"};  
}
```

```
void f() {  
    A *x {new A()};  
    if (test()) {  
        delete x;  
        throw Erreur{"voulue"};  
    }  
    delete x;  
}
```

l'équivalent du finally de java n'existe pas en c++

```
class A {  
    public :  
        A() {cout << " A";}  
        ~A() {cout << "mort A";}  
};
```

Echec à la construction ?

```
class B {  
    public:  
        B() { cout << " B"; throw 1; }  
        ~B() {cout << "mort B";}  
};
```

```
class C {  
    A a;  
    B b;  
    C() : a{}, b{} {cout << "C";}  
    ~C() { cout << "mort C";}  
};
```

```
int main() {  
    C c;  
};
```

```
class A {
public:
    A() {cout << " A";}
    ~A() {cout << "mort A";}
};
```

Echec à la construction ?

```
A
B
mort de A
```

```
class B {
public:
    B() { cout << " B"; throw 1; }
    ~B() {cout << "mort B";}
};
```

il n'y a pas d'appel  
au destructeur de B  
(ni de C)

```
class C {
    A a;
    B b;
    C(): a{}, b{} {cout << "C";}
    ~C() { cout << "mort C";}
};
```

```
int main() {
    C c;
};
```

Echec à la destruction ?

```
class A {  
    public :  
        A() {cout << " A";}  
        ~A() {cout << "mort A";}  
};
```

```
class B {  
    public:  
        B() { cout << " B";}  
        ~B() {cout << "mort B"; throw 1;}  
};
```

```
class C {  
    A a;  
    B b;  
    C() : a{}, b{} {cout << "C";}  
    ~C() { cout << "mort C";}  
};
```

ici

```
int main() {  
    C c;  
};
```

ici

```
class A {  
    public :  
    A () { cout << "A" ; }  
    ~A () { cout << "mort A" ; }  
};
```

Echec à la destruction ?

Très mauvaise situation ...

Imaginez qu'une erreur 'x' était déjà en traitement

En remontant (pour gérer l'erreur 'x') des objets sont détruits. Si une erreur y survient, il y a une concurrence entre 'x' et 'y'. Laquelle gérer ? Le mécanisme est inadapté, alors :

une exception dans un destructeur est **irrattrapable**

```
class B {  
    public:  
    B () { cout << "B" ; }  
    ~B () { cout << "mort B" ; throw 1 ; }  
};
```

```
class C {  
    A a;  
    B b;  
    C () : a {}, b {} { cout << "C" ; }  
    ~C () { cout << "mort C" ; }  
};
```

```
int main () {  
    C c;  
};
```

```
class A {};
```

Echec à la destruction ?

```
int main() {  
    A *a{new A{}};  
    delete a;  
    try{  
        delete a;  
    }catch (...) {  
        cout << "et là ?" ;  
    }  
};
```

non ! C'est **irratrapable aussi**



Pour finir sur les exceptions :  
en java on peut préciser quelles sont les exceptions que risquent de lancer les méthodes.

Le programmeur est alors contraint : il doit soit les rattraper, soit les relancer.

Le mécanisme correspondant en c++ est **deprecated** (essayé, jugé inefficace/coûteux, abandonné)

# Les classes internes



# Les classes internes

```
class Externe {  
    // ...  
    class Interne {  
        // ...  
    }  
    // ...  
};
```

spoil :

Notion (plus faible) **non** équivalente à celle en java.

```
class A{  
public :  
    class B{  
        public :  
            void g();  
    };  
    B b;  
};
```

on peut déclarer une  
classe dans une  
autre

```
int main() {  
    A::B x;  
    x.g();  
    return 0;  
}
```

```
void A::B::g() {  
    cout << "g() B" << endl;  
};
```

```
class A{  
public :  
    class B{  
        public :  
            void g();  
    };  
    B b;  
};
```

on peut déclarer une  
classe dans une  
autre

```
int main() {  
    A::B x;  
    x.g();  
    return 0;  
}
```

sa définition s'écrit  
dans le .cpp de A

```
void A::B::g() {  
    cout << "g() B" << endl;  
};
```

```
class A{  
public :  
    class B{  
        public :  
            void g();  
    };  
    B b;  
};
```

on peut déclarer une  
classe dans une  
autre

B est utilisable ici  
puisque'elle est  
déclarée publique

sa définition s'écrit  
dans le .cpp de A

```
void A::B::g() {  
    cout << "g() B" << endl;  
};
```

```
int main() {  
    A::B x;  
    x.g();  
    return 0;  
}
```

```
class A{  
public :  
    class B{  
        public :  
            void g();  
    };  
    B b;  
};
```

on peut déclarer une  
classe dans une  
autre

B est utilisable ici  
puisque'elle est  
déclarée publique

sa définition s'écrit  
dans le .cpp de A

```
void A::B::g() {  
    cout << "g() B" << endl;  
};
```

```
int main() {  
    A::B x;  
    x.g();  
    return 0;  
}
```

g() B

```
class A{  
private :  
    class B{  
        public :  
            void g();  
    };  
    B b;  
};
```

**error:**  
**'class A::B' is private**  
**within this context**

```
int main() {  
    A::B x;  
    x.g();  
    return 0;  
}
```

```
void A::B::g() {  
    cout << "g() B" << endl;  
};
```

```
class A{  
public :  
    class B{  
        public :  
            void g();  
    };  
    B b;  
};
```

Remarquez qu'on a pu définir x sans avoir d'instance de la classe A.

```
int main() {  
    A::B x;  
    x.g();  
    return 0;  
}
```

```
void A::B::g() {  
    cout << "g() B" << endl;  
};
```

g() B

```
class A{  
public :  
    class B{  
        public :  
            void g();  
    };  
    B b;  
};
```

Remarquez qu'on a pu définir c sans avoir d'instance de la classe A.

```
int main() {  
    A::B c;  
    c.g();  
    return 0;  
}
```

En c++ une classe interne est nécessairement considérée statiquement.

Il n'y a pas d'autre notion (de classe membre) donc pas de raison de faire apparaître "static"

```
void A::B::g() {  
    cout << "g() B" << endl;  
};
```

g() B



```
class A{  
public :  
    int n;  
    class B{  
        public :  
            void g() ;  
    } ;  
    B b ;  
} ;
```

Remarquez qu'on a pu définir x sans avoir d'instance de la classe A.

```
int main() {  
    A::B x ;  
    x.g() ;  
    return 0 ;  
}
```

En c++ une classe interne est nécessairement considérée statiquement.

Il n'y a pas d'autre notion (de classe membre) donc pas de raison de faire apparaître "static"

```
void A::B::g() {  
    cout << "g() B" << endl ;  
    n++ ;  
} ;
```

error: invalid use of non-static data member 'A::n'

```
class A{  
private :  
    int x;  
    class B{  
        public :  
            void g(A& );  
    };  
    B b;  
};
```

```
void A::B::g(A &a) {  
    cout << "g() B" << endl;  
    a.x++;  
};
```

Les privilèges accordés à B  
sont ceux d'une classe  
amie de A

```
class A{  
private :  
    class B{  
        private :  
            int x;  
    };  
    void f();  
};
```

Mais la classe englobante  
n'a pas de privilège particulier

```
void A::f() {  
    B b;  
    b.x++;  
};
```

error: 'int A::B::x' is private within this context

**La librairie SFML :**

« Simple and Fast Multimedia Library »

(permettra d'avoir un projet présentable)

# Installation :

sous linux (5 min) simplement :

```
sudo apt-get install libsfml-dev
```

puis dans votre make ajoutez les options de compilation :

```
g++ main.cpp -lsfml-graphics -lsfml-window -lsfml-system
```

# Installation :

sous windows + codeblocks ...

j'ai dû tout désinstaller et réinstaller  
car les binaires devaient correspondre  
parfaitement à une version du  
compilateur... (~2h avec les tutos)

Privilégiez un travail sous unix

Le tutorial est très bien fait :

<https://www.sfml-dev.org/learn-fr.php>

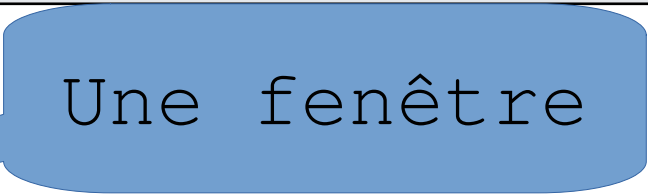


# Exemple de démonstration :

```
#include <SFML/Graphics.hpp>
using namespace sf;
int main() {
    RenderWindow app(VideoMode(800, 600, 32), "Test ");
    while (app.isOpen()) {
        Event event;
        while (app.pollEvent(event)) {
            switch (event.type) {
                case Event::Closed:
                    app.close(); break;
                default: break;
            }
        }
        app.clear(); // vide l'écran
        ...         // mise en place des objets graph.
        app.display(); // Affichage effectif
    } // fenêtre fermée
    return EXIT_SUCCESS;
}
```

# Exemple de démonstration :

```
#include <SFML/Graphics.hpp>
using namespace sf;
int main() {
    RenderWindow app(VideoMode(800, 600, 32), "Test ");
    while (app.isOpen()) {
        Event event;
        while (app.pollEvent(event)) {
            switch (event.type) {
                case Event::Closed:
                    app.close(); break;
                default: break;
            }
        }
        app.clear(); // vide l'écran
        ... // mise en place des objets graph.
        app.display(); // Affichage effectif
    } // fenêtre fermée
    return EXIT_SUCCESS;
}
```



Une fenêtre

# Exemple de démonstration :

```
#include <SFML/Graphics.hpp>
using namespace sf;
int main() {
    RenderWindow app(VideoMode(800, 600, 32), "Test ");
    while (app.isOpen()) {
        Event event;
        while (app.pollEvent(event)) {
            switch (event.type) {
                case Event::Closed:
                    app.close(); break;
                default: break;
            }
        }
        app.clear(); // vide l'écran
        ... // mise en place des objets graph.
        app.display(); // Affichage effectif
    } // fenêtre fermée
    return EXIT_SUCCESS;
}
```

Une fenêtre

Boucle "infinie"

# Exemple de démonstration :

```
#include <SFML/Graphics.hpp>
using namespace sf;
int main() {
    RenderWindow app(VideoMode(800, 600, 32), "Test ");
    while (app.isOpen()) {
        Event event;
        while (app.pollEvent(event)) {
            switch (event.type) {
                case Event::Closed:
                    app.close(); break;
                default: break;
            }
        }
        app.clear(); // vide l'écran
        ... // mise en place des objets graph.
        app.display(); // Affichage effectif
    } // fenêtre fermée
    return EXIT_SUCCESS;
}
```

Une fenêtre

Boucle "infinie"

gestions des événements

# Exemple de démonstration :

```
#include <SFML/Graphics.hpp>
using namespace sf;
int main() {
    RenderWindow app(VideoMode(800, 600, 32), "Test ");
    while (app.isOpen()) {
        Event event;
        while (app.pollEvent(event)) {
            switch (event.type) {
                case Event::Closed:
                    app.close(); break;
                default: break;
            }
        }
        app.clear(); // vide l'écran
        ... // mise en place des objets graph.
        app.display(); // Affichage effectif
    } // fenêtre fermée
    return EXIT_SUCCESS;
}
```

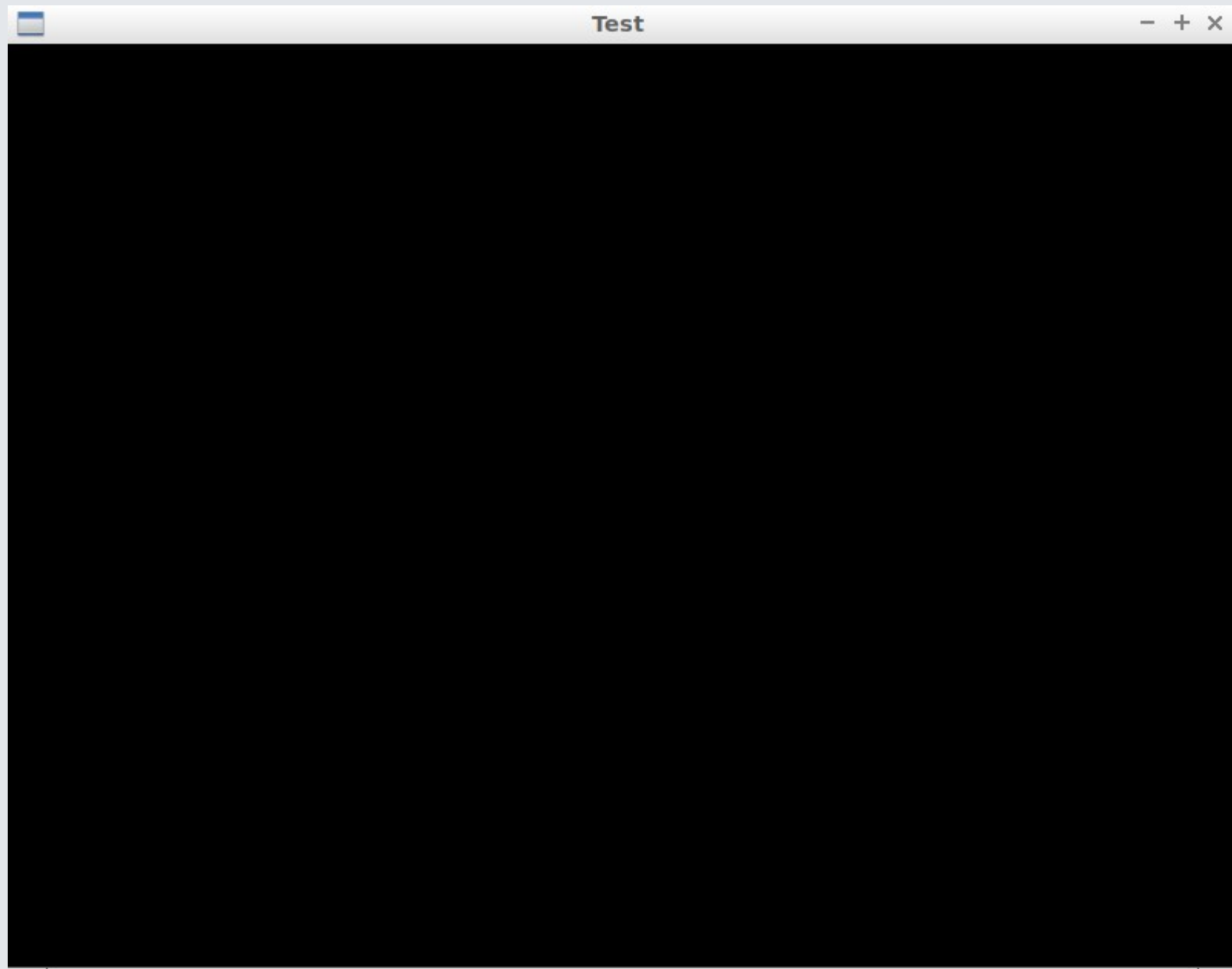
Une fenêtre

Boucle "infinie"

gestions des événements

pollEvent prend une référence : event pourra être modifié  
On traite les evts avant de passer à la maj

mise à jour du contenu puis affichage



# Hello World ...

```
Font font;  
font.loadFromFile("./Agatha.ttf");  
Text text;  
text.setFont(font);  
text.setString("    Hello world");  
text.setCharacterSize(100);  
text.setFillColor(Color::Red);
```

```
app.clear();  
app.draw(text);  
app.display();
```





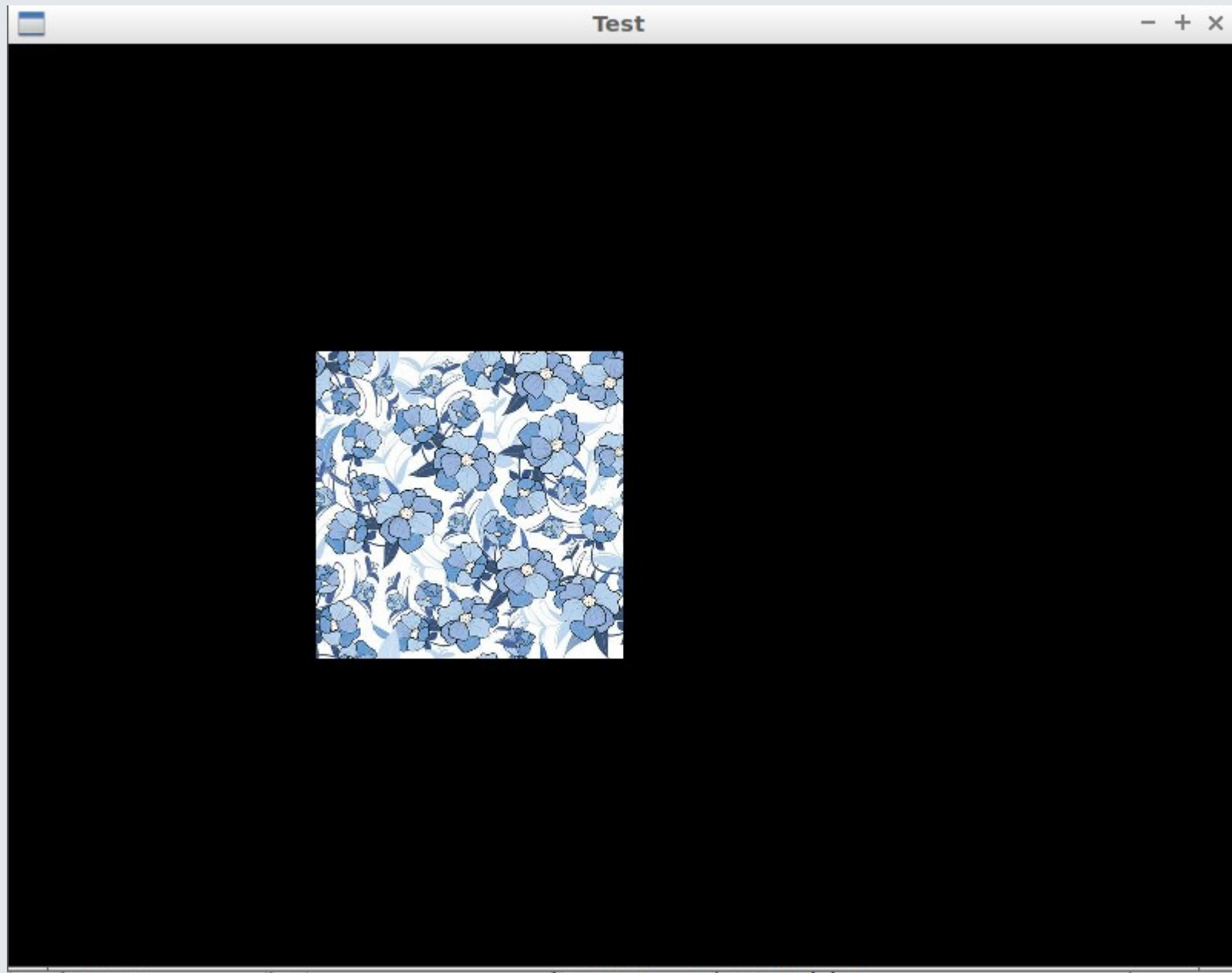
# Ajout d'un élément graphique :

```
Texture texture;  
texture.loadFromFile("./fleurs.png");  
Sprite tuile;  
tuile.setTexture(texture);  
tuile.setScale(Vector2f(0.5, 0.5)); // reduction moitié  
tuile.move(Vector2f(200, 200)); // placement
```

Un sprite est un objet dessinable

On lui associe une image/texture

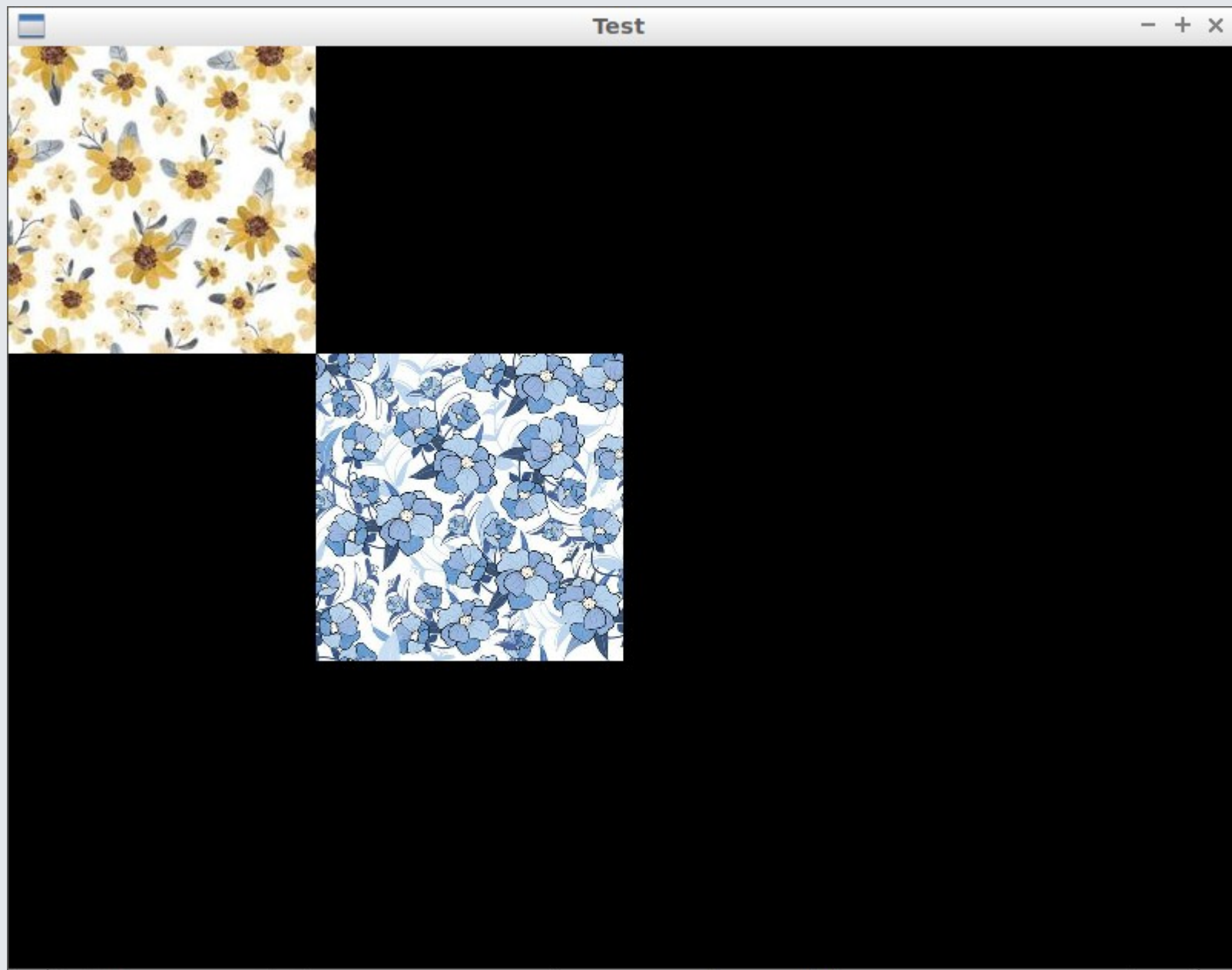
```
app.clear();  
app.draw(tuile);  
app.display();
```



# Ajout d'un second élément :

```
Texture tex;  
tex.loadFromFile("./fleur2.png");  
Sprite tuile2;  
tuile2.setTexture(tex);
```

```
app.clear();  
app.draw(tuile);  
app.draw(tuile2);  
app.display();
```



# Un peu de mouvement :

```
Sprite * current =& tuile; // c'est la bleue
```

```
if (Keyboard::isKeyPressed(Keyboard::Right))  
    current -> move(1, 0);  
if (Keyboard::isKeyPressed(Keyboard::Left))  
    current -> move(-1, 0);  
if (Keyboard::isKeyPressed(Keyboard::Up))  
    current -> move(0, -1);  
if (Keyboard::isKeyPressed(Keyboard::Down))  
    current -> move(0, 1);
```

TEST SUR LA CONSOLE

# Rq sur la gestion des événements

...

```
while (app.pollEvent(event)) {  
    switch (event.type) {  
        case Event::Closed:  
            app.close(); break;  
        case Event::KeyPressed:  
            if (event.key.code==Keyboard::Return)  
                cout << "salut" << endl;  
            break;  
        default: break;  
    }  
}  
if (Keyboard::isKeyPressed(Keyboard::Right))  
    current -> move(1, 0);
```

...

}



# Rq sur la gestion des événements

...

```
while (app.pollEvent(event)) {  
    switch (event.type) {  
        case Event::Closed:  
            app.close(); break;  
        case Event::KeyPressed:  
            if (event.key.code==Keyboard::Return)  
                cout << "salut" << endl;  
            break;  
        default: break;  
    }  
    if (Keyboard::isKeyPressed(Keyboard::Right))  
        current -> move(1, 0);  
}
```

le fait d'avoir  
appuyé

le fait d'être  
appuyée

...

}

TEST SUR LA CONSOLE

# Sélection d'un des sprites à la souris :

```
// On stocke les bornes du sprite  
FloatRect bounds_t1 = tuile.getGlobalBounds();  
FloatRect bounds_t2 = tuile2.getGlobalBounds();
```

```
if (Mouse::isButtonPressed(Mouse::Left)) {  
    Vector2f mouse =  
        app.mapPixelToCoords(Mouse::getPosition(app));  
    if (bounds_t1.contains(mouse)) current = & tuile;  
    if (bounds_t2.contains(mouse)) current = & tuile2;  
}
```

petite transformation,  
nécessaire car on peut  
redimensionner l'écran

TEST SUR LA CONSOLE

Pour tout le reste, voyez le tutorial

<https://www.sfml-dev.org/learn-fr.php>

# qqs mots sur les énumérations

On les a déjà rencontrées dans SFML

```
// ...  
text.setFillColor(Color::Red) ;  
  
// ...  
if (Keyboard::isKeyPressed(Keyboard::Right) )
```

# qqs mots sur les énumérations

le plus simple :

```
enum Color {  
    Red,  
    Orange,  
    Blue  
};
```

```
int main() {  
    Color x{Red};  
    switch(x) {  
        case Orange :  
            f(); break;  
        case Red :  
            g(); break;  
        default : break;  
    }  
}
```

Rq : les constantes ainsi introduites font partie du domaine de nom

# qqs mots sur les énumérations

Si on essaye de faire du rétro-ingéniering et de deviner comment ça a pu être écrit ...

```
// ...  
text.setFillColor(Color::Red) ;  
  
// ...  
if (Keyboard::isKeyPressed(Keyboard::Right) )
```



# qqs mots sur les énumérations

c'est codable avec une enum simple, mais alors on perdra la designation du contexte

```
// ...  
text.setFillColor(Color::Red) ;  
  
// ...  
if (Keyboard::isKeyPressed(Keyboard::Right) )
```

# qqs mots sur les énumérations

c'est codable avec une enum simple, mais alors on perdra la designation du contexte

```
// ...  
text.setFillColor(Red) ;  
  
// ...  
if (Keyboard::isKeyPressed(Keyboard::Right) )
```

```
enum Color {  
    Red,  
    Orange,  
    Blue  
};
```

# qqs mots sur les énumérations

```
// ...  
text.setFillColor(Red) ;  
  
// ...  
if (Keyboard::isKeyPressed(Keyboard::Right) )  
    move(Direction::Right)
```

Ici pour Right il y a 2 concepts, donc on ne peut pas introduire avec enum le "mot" Right dans le namespace sans créer d'ambiguïté

# qqs mots sur les énumérations

```
if (Keyboard::isKeyPressed (Keyboard::Right) )  
    move (Direction::Right)
```

```
enum class Direction {  
    Right,  
    Left  
};  
  
enum class Keyboard {  
    Right,  
    Up  
};
```

Les enum class  
permettent de préserver  
la définition de portée

# qqs mots sur les énumérations

```
if (Keyboard::isKeyPressed (Keyboard::Right) )  
    move (Direction::Right)
```

```
enum class Direction {  
    Right,  
    Left  
};  
  
enum class Keyboard {  
    Right,  
    Up  
};
```

Les enum class permettent de préserver la définition de portée

Mais les enum class ne sont pas de vraies classes, on ne peut pas les enrichir d'attributs ou de méthodes, etc ...

# qqs mots sur les énumérations

```
if (Keyboard::isKeyPressed(Keyboard::Right))  
    move(Direction::Right)
```

```
enum class Direction {  
    Right,  
    Left  
};
```

```
enum class Keyboard {  
    Right,  
    Up  
};
```

Les enum class permettent de préserver la portée

Mais les enum class ne sont pas de vraies classes, on ne peut pas les enrichir d'attributs ou de méthodes, etc ...

Or, dans cet exemple Keyboard ne peut pas être une simple enum class, à cause de cette méthode disponible

# qqs mots sur les énumérations

```
if (Keyboard::isKeyPressed(Keyboard::Right))  
    move(Direction::Right)
```

```
enum class Direction {  
    Right,  
    Left  
};  
  
class Keyboard {  
public :  
    enum Key{Right, up};  
    static bool isKeyPressed(Key);  
};
```

# qqs mots sur les énumérations

```
if (Keyboard::isKeyPressed (Keyboard::Right) )  
    move (Direction::Right)
```

```
enum class Direction {  
    Right,  
    Left  
};
```

```
class Keyboard {  
    public :  
        enum Key {Right, up};  
        static bool isKeyPressed (Key) ;  
};
```

Remarquez que  
Keyboard::Key::Right  
n'est pas nécessaire  
  
car Key est une enum  
simple dans Keyboard



# qqs mots sur les énumérations

```
if (Keyboard::isKeyPressed (Keyboard::Right) )  
    move (Direction::Right)
```

Rq : je n'ai pas trouvé de syntaxe pour importer localement avec l'équivalent de using l'une ou l'autre des constantes en laissant le choix au programmeur.

Dites moi si vous le voyez qq part, mais je doute que ce soit fait vu les discussions en cours dans :  
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1099r5.html>

# Correction du TP noté 2022

L'occasion de revenir sur les erreurs :

- initialisations
- commentaires
- règles du makefile
- mauvaise gestion des copies,  
ou affectation
- tests non significatifs

# Correction du TP noté 2022

Objet :

- un travail sur les multigraphes, pondérés, avec un calcul d'arbre couvrant minimal (kruskal)
- la gestion de la construction/destruction des sommets/arcs/graphes devait être centralisée via un GarbageCollector

# Nature des GarbageCollector ?

On peut remarquer qu'il serait contre productif d'imaginer avoir plusieurs GC : idéalement cette classe ne contient qu'un seul élément.  
C'est un ensemble singleton.

A la limite, il préexiste au main(), et se détruit tout seul quand le main() termine.

La manipulation de cet objet doit être la plus invisible possible.

# GC en singleton

```
class Gc final{  
    private:  
        static Gc instance,  
        Gc();  
        ~Gc();  
};
```

```
Gc Gc::instance;  
Gc::Gc() {  
    cout << "création GC";  
}  
Gc::~~Gc() {  
    cout << "destruction GC";  
}
```

```
#include "Gc.hpp"  
int main() {}  
    cout << "main()";  
};
```



prudent + économie de place :  
~Gc n'a pas à être virtual

# GC en singleton

```
class Gc final{  
    private:  
        static Gc instance,  
        Gc();  
        ~Gc();  
};
```

un élément Gc particulier  
est rattaché à la classe

```
Gc Gc::instance;  
Gc::Gc() {  
    cout << "création GC";  
}  
Gc::~~Gc() {  
    cout << "destruction GC";  
}
```

prudent + économie de place :  
~Gc n'a pas à être virtual

```
#include "Gc.hpp"  
int main() {}  
    cout << "main()";  
};
```

# GC en singleton

```
class Gc final{  
    private:  
        static Gc instance,  
        Gc();  
        ~Gc();  
};
```

un élément Gc particulier  
est rattaché à la classe

les constructeurs et  
destructeurs sont cachés

```
Gc Gc::instance;  
Gc::Gc() {  
    cout << "création GC";  
}  
Gc::~~Gc() {  
    cout << "destruction GC";  
}
```

prudent + économie de place :  
~Gc n'a pas à être virtual

```
#include "Gc.hpp"  
int main() {}  
    cout << "main()";  
};
```

# GC en singleton

```
class Gc final{
private:
    static Gc instance,
    Gc();
    ~Gc();
};
```

un élément Gc particulier  
est rattaché à la classe

les constructeurs et  
destructeurs sont cachés

```
Gc Gc::instance;
Gc::Gc() {
    cout << "création GC";
}
Gc::~~Gc() {
    cout << "destruction GC";
}
```

prudent + économie de place :  
~Gc n'a pas à être virtual

l'édition de lien  
intervient avant le main.  
L'instance préexiste au  
main()

```
#include "Gc.hpp"
int main() {}
    cout << "main() ";
};
```

```
creation GC
main()
destruction GC
```



# Discussions sur l'interface de GC

```
class Gc final{
private:
    static Gc instance;
    Gc();
    ~Gc();
    list<Arete*> list_a;
    list<Sommet*> list_s;
public:
    static void add(Arete* a);
    static void add(Sommet* s);
    static void remove(Arete* a);
    static void remove(Sommet* s);
};
```

Cet objet sera informé des  
construction/destruction, et en fera la gestion

# Discussions sur l'interface de GC

```
class Gc final{
private:
    static Gc instance;
    Gc();
    ~Gc();
    list<Arete*> list_a;
    list<Sommet*> list_s;
public:
    static void add(Arete* a);
    static void add(Sommet* s);
    static void remove(Arete* a);
    static void remove(Sommet* s);
};
```

nécessaire pour les mises à jour

# Discussions sur l'interface de GC

```
class Gc final{
private:
    static Gc instance;
    Gc();
    ~Gc();
    list<Arete*> list_a;
    list<Sommet*> list_s;
public:
    static void add(Arete* a);
    static void add(Sommet* s);
    static void remove(Arete* a);
    static void remove(Sommet* s);
};
```

trop permissif...

# Discussions sur l'interface de GC

```
class Gc final{
private:
    static Gc instance;
    Gc();
    ~Gc();
    list<Arete*> list_a;
    list<Sommet*> list_s;

    static void add(Arete* a);
    static void add(Sommet* s);
    static void remove(Arete* a);
    static void remove(Sommet* s);

    friend class Arete;
    friend class Sommet;
};
```

ne faisons confiance  
qu'à ces deux là

# Discussions sur l'interface de GC

```
class Gc final{
private:
    static Gc instance;
    Gc();
    ~Gc();
    list<Arete*> list_a;
    list<Sommet*> list_s;

    static void add(Arete* a);
    static void add(Sommet* s);
    static void remove(Arete* a);
    static void remove(Sommet* s);

    friend class Arete;
    friend class Sommet;
};

Gc::Gc() : list_a{}, list_s{}
{}

void Gc::add(Arete* a) {
    instance.list_a.push_front(a);
}

void Gc::remove(Arete* a) {
    instance.list_a.remove(a);
}

Gc::~~Gc() {
    cout << "a faire plus tard";
    // gérer la destruction
    // de ceux qui restent
}
```

# Les sommets

```
class Sommet {  
private:  
    const string etiquette;  
    int marquage; // pour union  
  
public:  
    Sommet(string s);  
    Sommet(Sommet const& s);  
    virtual ~Sommet();  
    const string & get_label() const;  
  
friend  
    ostream& operator<<(ostream& out, const Sommet& x);  
};
```

The diagram consists of three blue callout boxes with pointers directed at specific parts of the C++ code. The first box, labeled 'invariant', points to the 'const' keyword in the 'etiquette' member declaration. The second box, labeled 'ce getter laisse l'objet invariant', points to the 'const' keyword in the 'get\_label()' method signature. The third box, labeled 'const et référence', points to the 'const Sommet&' parameter in the overloaded operator '<<' function.

Les sommets sont étiquetés, et on prévoit qu'ils seront parcourus : on aura besoin de les marquer

# Les sommets

liste d'initialisation

```
Sommet::Sommet(string s)
: etiquette{s}, marquage{0} {
    Gc::add(this);
}
```

A sa création, un objet se déclare au GC, via une méthode de classe

```
Sommet::Sommet(Sommet const& s) :
etiquette{ s.etiquette }, marquage{0} {
    Gc::add(this);
}
```

idem

```
Sommet::~~Sommet() {
    Gc::remove(this);
}
```

A sa destruction, un objet se retire.  
... discussion ...

# Destructions par ou en dehors du GC ?

```
int main() {  
    Sommet* a {new Sommet("a")};  
    Sommet b {"b"};  
    { // dummy bloc  
        Sommet b2{b};  
    } // fin de bloc  
}
```



# Destructions par ou en dehors du GC ?

```
int main() {  
    Sommet* a {new Sommet("a")};  
    Sommet b {"b"};  
    { // dummy bloc  
        Sommet b2{b};  
    } // fin de bloc  
}
```

ce new est  
clairement à la  
charge du GC

# Destructions par ou en dehors du GC ?

```
int main() {  
    Sommet* a {new Sommet("a")};  
    Sommet b {"b"};  
    { // dummy bloc  
        Sommet b2{b};  
    } // fin de bloc  
}
```

ce new est  
clairement à la  
charge du GC

b2 est détruit tout  
seul à la fin du  
bloc.  
Hors GC...

# Destructions par ou en dehors du GC ?

```
int main() {  
    Sommet* a {new Sommet("a")};  
    Sommet b {"b"};  
    { // dummy bloc  
        Sommet b2{b};  
    } // fin de bloc  
}
```

ce new est  
clairement à la  
charge du GC

b2 est détruit tout  
seul à la fin du  
bloc.  
Hors GC...

b aussi, à la fin du  
bloc main()  
Hors GC...

```
Sommet::~~Sommet() {  
    Gc::remove(this);  
}
```

# Destructions par ou en dehors du GC ?

```
int main() {  
    Sommet* a {new Sommet("a")};  
    Sommet b {"b"};  
    { // dummy bloc  
        Sommet b2{b};  
    } // fin de bloc  
}
```

ce new est  
clairement à la  
charge du GC

b2 est détruit tout  
seul à la fin du  
bloc.  
Hors GC...

b aussi, à la fin du  
bloc main()  
Hors GC...

```
Sommet::~~Sommet() {  
    Gc::remove(this);  
}
```

la destruction du  
singleton de GC  
intervient ici

# Destructions par ou en dehors du GC ?

```
void f(Sommet x) {...}  
  
int main() {  
    Sommet b {"b"};  
    f(b);  
}
```

Rappel : on a ce cas de figure de bloc de manière non "dummy" lors de l'appel de fonction

# Le destructeur de GC

On peut penser à qq chose comme :

```
Gc::~~Gc() {  
    for (Sommet * x : list_s) delete x;  
}
```

# Le destructeur de GC

On peut penser à qq chose comme :

```
Gc::~~Gc() {  
    for (Sommet * x : list_s) delete x;  
}
```

Mais cela plante ...

# Le destructeur de GC

On peut penser à qq chose comme :

```
Gc::~Gc() {  
    for (Sommet * x : list_s) delete x;  
}
```

Mais cela plante ...

En effet, delete ne fait pas que libérer de la mémoire,  
il fait appel à ~Sommet()

```
Sommet::~~Sommet() {  
    Gc::remove(this);  
}
```

Et la liste est modifiée pendant son parcours,  
les itérateurs sont perdus



# Le destructeur de GC

mais ici :

```
Gc::~Gc () {  
    while (!list_s.empty()) delete list_s.front();  
}
```

Le delete fait un remove dans la foulée

La liste est aussi modifiée, mais on n'utilise pas d'itérateurs

```
Sommet::~Sommet () {  
    Gc::remove(this);  
}
```

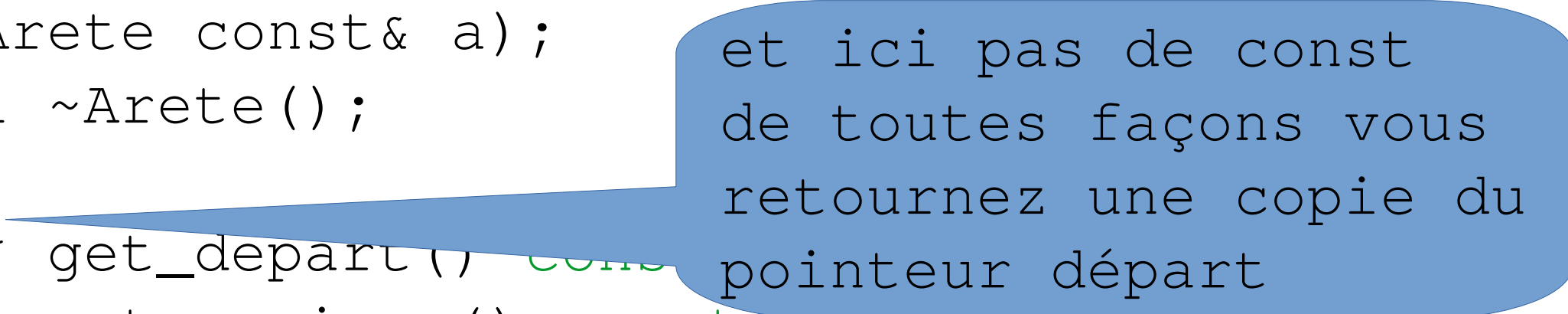
Il faut impérativement expliquer ce genre de "truc" !

# Arrêtes - Pas de difficultés

```
class Arete {  
    private:  
        Sommet*  const depart;  
        Sommet*  const arrivee;  
        int poids;  
    public:  
        Arete(string depart, string arrivee, int poids);  
        Arete(Sommet* depart, Sommet* arrivee, int poids);  
        Arete(Arete const& a);  
        virtual ~Arete();  
  
        Sommet*  get_depart() const;  
        Sommet*  get_arrivee() const;  
        int get_poids() const;  
  
    friend  
    ostream& operator<<(ostream& out, const Arete& a);  
};
```

# Arrêtes - Pas de difficultés

```
class Arete {  
    private:  
        Sommet* const depart;  
        Sommet* const arrivee;  
        int poids;  
    public:  
        Arete(string depart, string arrivee, int poids);  
        Arete(Sommet* depart, Sommet* arrivee, int poids);  
        Arete(Arete const& a);  
        virtual ~Arete();  
  
        Sommet* get_depart() const;  
        Sommet* get_arrivee() const;  
        int get_poids() const;  
  
    friend  
        ostream& operator<<(ostream& out, const Arete& a);  
};
```



et ici pas de const  
de toutes façons vous  
retournez une copie du  
pointeur départ

# Arrêtes - Pas de difficultés

```
class Arete {
private:
    Sommet* const depart;
    Sommet* const arrivee;
    int poids;
public:
    Arete(string depart, string arrivee, int poids);
    Arete(Sommet* depart, Sommet* arrivee, int poids);
    Arete(Arete const& a);
    virtual ~Arete();

    Sommet* get_depart() const;
    Sommet* get_arrivee() const;
    int get_poids() const;

friend
ostream& operator<<(ostream& out, const Arete& a);
};
```

un pointeur est un peu discutable.  
On autoriserait null ?

# Arrêtes - Pas de difficultés

```
class Arete {
private:
    Sommet* const depart;
    Sommet* const arrivee;
    int poids;
public:
    Arete(string depart, string arrivee, int poids);
    Arete(Sommet& depart, Sommet& arrivee, int poids);
    Arete(Arete const& a);
    virtual ~Arete();

    Sommet* get_depart() const;
    Sommet* get_arrivee() const;
    int get_poids() const;

friend
ostream& operator<<(ostream& out, const Arete& a);
};
```

préférez des références  
Qui ne sont jamais null !

# Arrêtes - Pas de difficultés

```
Arete::Arete(string depart, string arrivee, int poids)
: depart{ new Sommet(depart) },
  arrivee{ new Sommet(arrivee) },
  poids{ poids } {
    Gc::add(this); // sans s'occuper des sommets
}

Arete::~~Arete() {
    Gc::remove(this); // idem
}
```

# Les graphes - 1/6

```
class Graph {
public:
    Graph(vector<Sommet *> = vector<Sommet *> {},
          vector<Arete *> = vector<Arete *>{} );
    void ajoute_sommet(Sommet &);
    void ajoute_arete(Arete & x) ;
    void symetrise();
    Graph kruskal();
private:
    vector <Sommet *> vertices;
    vector <Arete *> edges;
};
```

# Les graphes - 2/6

```
#include <algorithm>

Graph::Graph(vector<Sommet *> s,
             vector<Arete *> a)
    : vertices{s}, edges{a} {} // on fait confiance ?

void Graph::ajoute_sommet(Sommet & x) {
    if( find(vertices.begin(), vertices.end(), &x)
        == vertices.end())
        vertices.push_back(&x);
}

void Graph::ajoute_arete(Arete & x) {
    if( find(edges.begin(), edges.end(), &x) ==
        edges.end()) {
        edges.push_back(&x);
        ajoute_sommet(*x.get_depart());
        ajoute_sommet(*x.get_arrivee());
    }
}
```



# Les graphes - 3/6

```
void Graph::symetrise() {
    vector<Arete*> newEdges;
    for (Arete * x : edges) {
        bool exist = false;
        for (Arete * y : edges)
            if (x->get_arrivee() == y->get_depart()
                && x->get_depart() == y->get_arrivee()
                && x->get_poids() == y->get_poids()) {
                exist = true;
                break;
            }
        // fin du for y
        if (!exist)
            newEdges.push_back(new Arete (* (x->get_arrivee()),
                                           * (x->get_depart()),
                                           x->get_poids()));

    } // fin du for x
    for (Arete * x : newEdges) edges.push_back(x);
}
```

# Les graphes - 4/6

```
Graph Graph::kruskal() {
    vector <Arete *> tree;

    this->symetrise();

    // sort edges (avec la biblio algorithm)
    sort(edges.begin(),
        edges.end(),
        [](Arete* a, Arete* b) {
            return a->get_poids() < b->get_poids();
        }
    );

    //creer ensemble()
    int id = 0;
    for(Sommet *x : vertices) x->mark(id++);

    ... à suivre ...
}
```

# Les graphes - 5/6

```
Graph Graph::kruskal() {
    ... suite ...
    for(Arete* x : edges)
        if( x->get_depart()->mark()
            !=
            x->get_arrivee()->mark() )
        {
            tree.push_back(x);
            //union
            int a = x->get_arrivee()->mark();
            int d = x->get_depart()->mark();
            for(Sommet *y : vertices)
                if( y->mark() == a) y->mark(d);
        }
    ... à suivre ...
}
```

# Les graphes - 6/6

```
Graph Graph::kruskal() {  
    ... suite ...  
    Graph rep;  
    for (Arete *x:tree) rep.ajoute_arete(*x);  
    rep.symetrise();  
    return rep;  
}
```