

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

Yan Jurski

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...

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

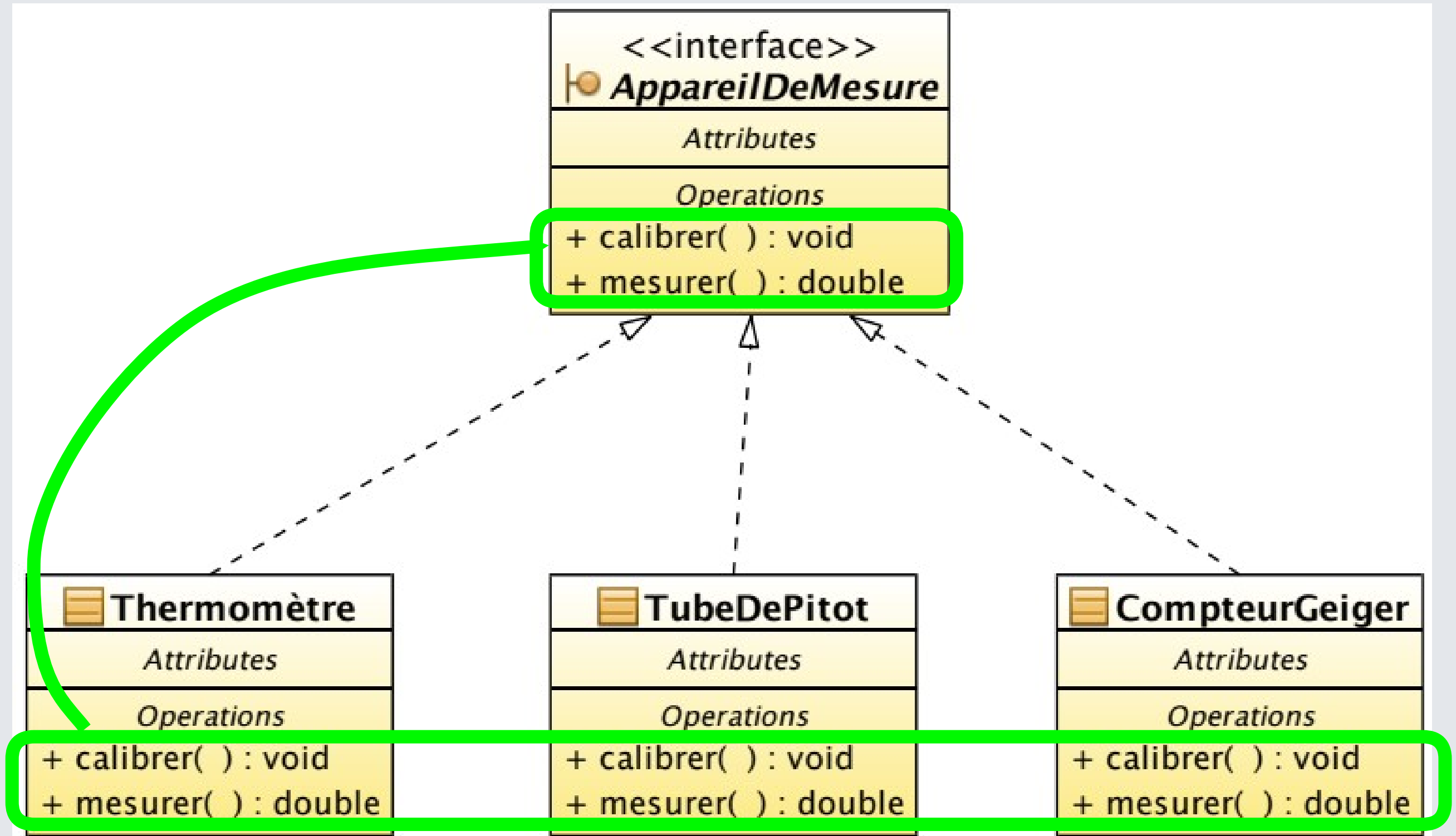
 TubeDePitot
<i>Attributes</i>
<i>Operations</i> + calibrer() : void + mesurer() : double

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

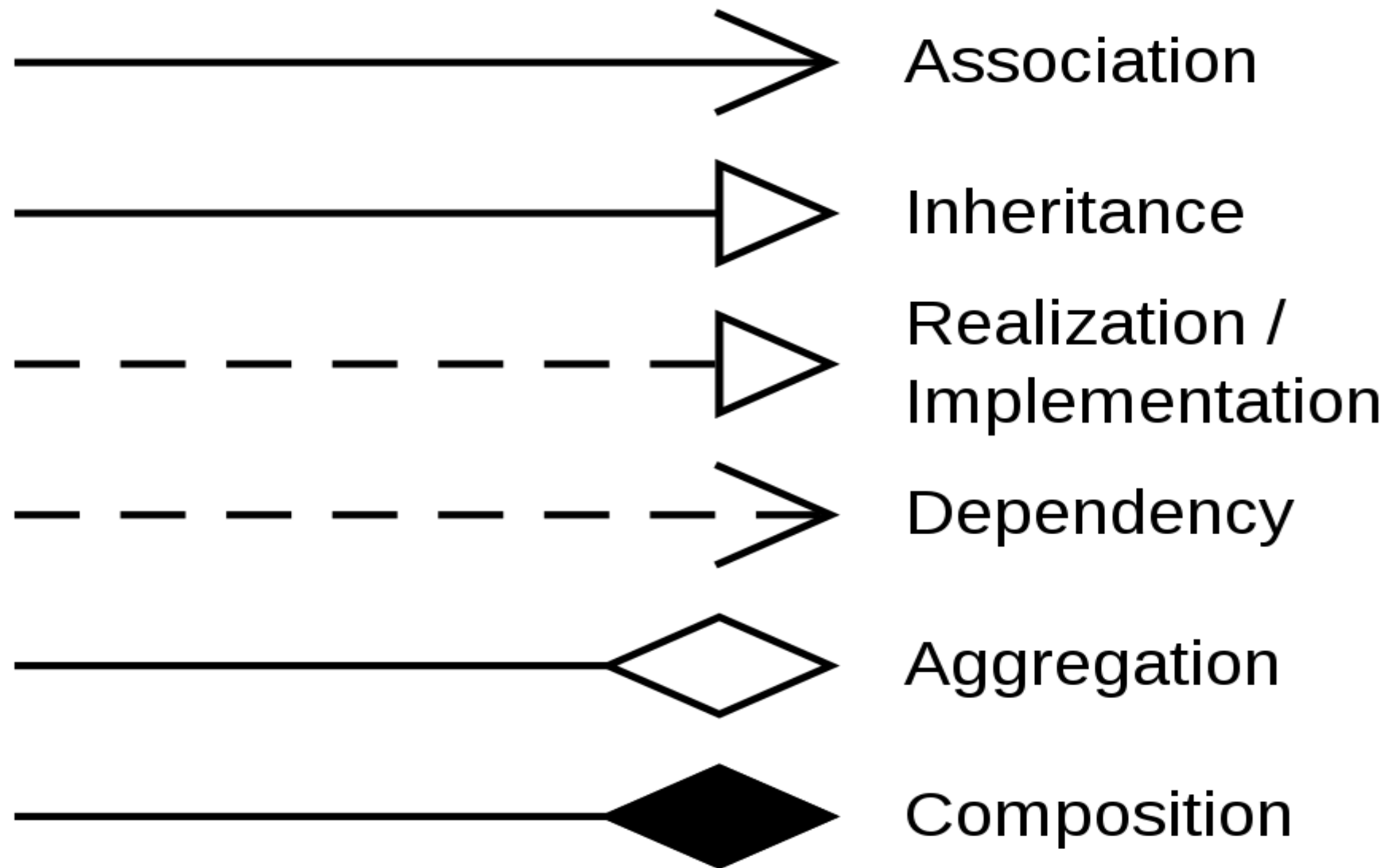
Des classes concrètes identifiées...

 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

on parle de **virtuelle pure**

```
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

```
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 */ }  
};
```

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.

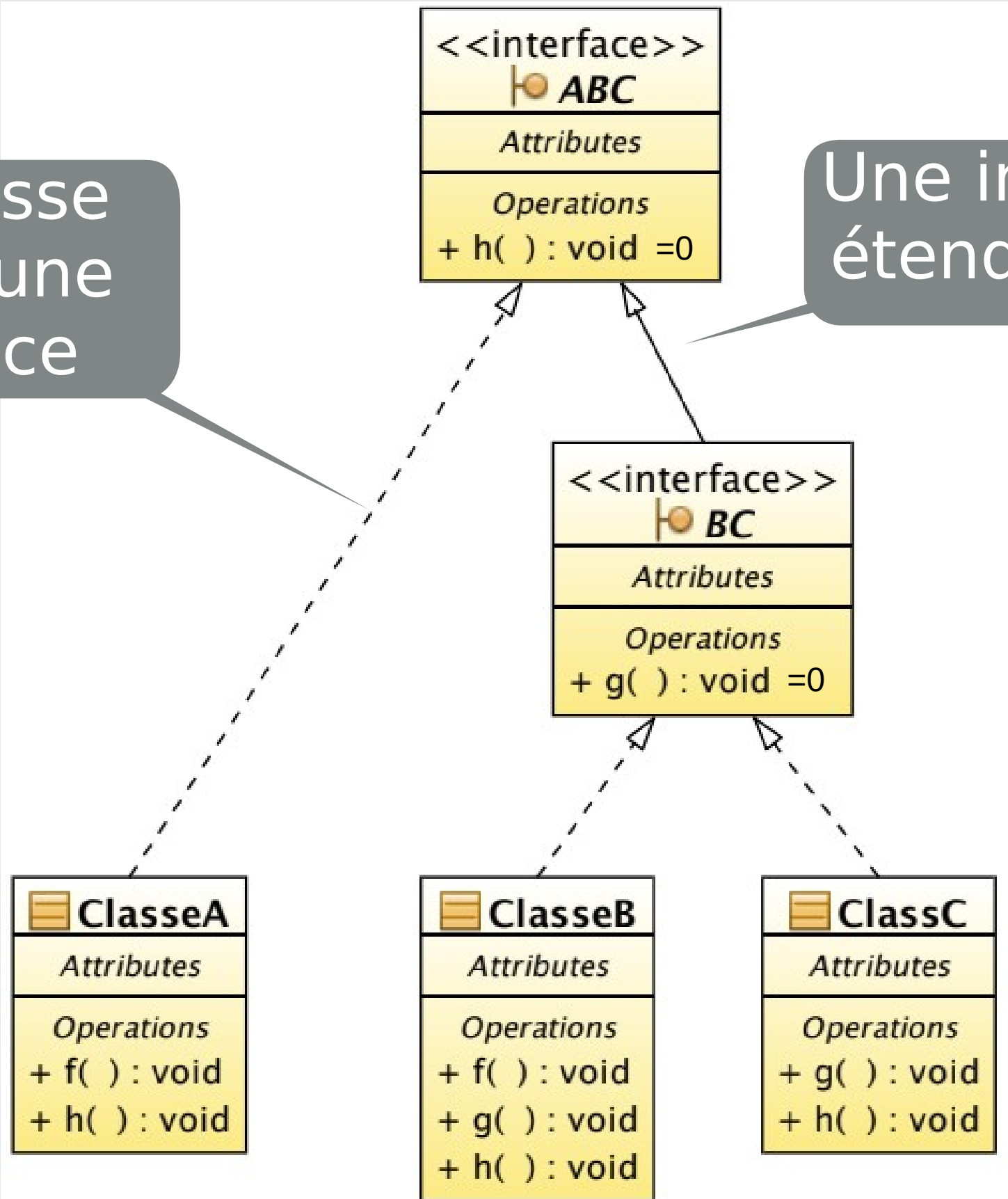
 ClasseA
<i>Attributes</i>
<i>Operations</i> + f() : void + h() : void

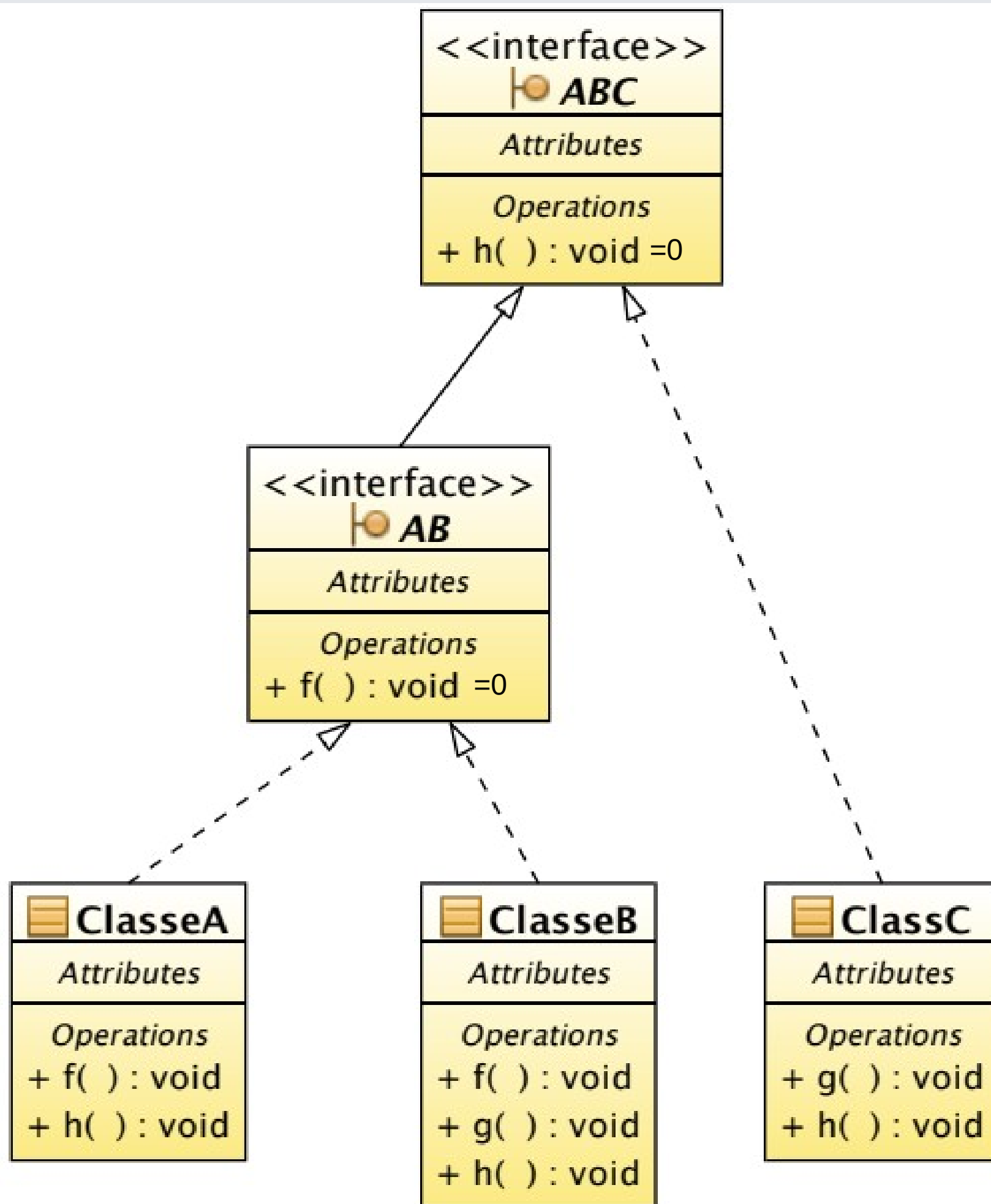
 ClasseB
<i>Attributes</i>
<i>Operations</i> + f() : void + g() : void + h() : void

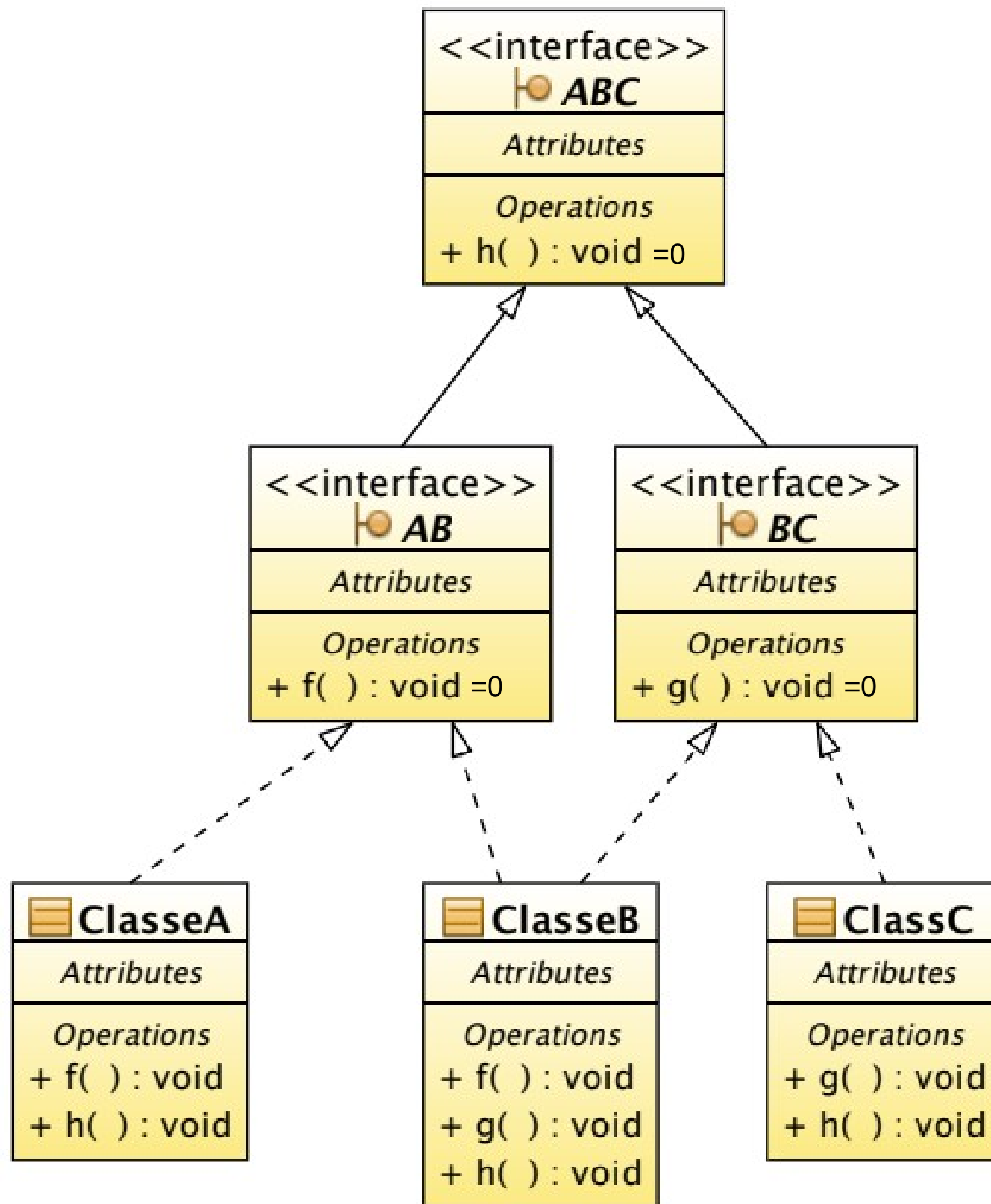
 ClassC
<i>Attributes</i>
<i>Operations</i> + g() : void + h() : void

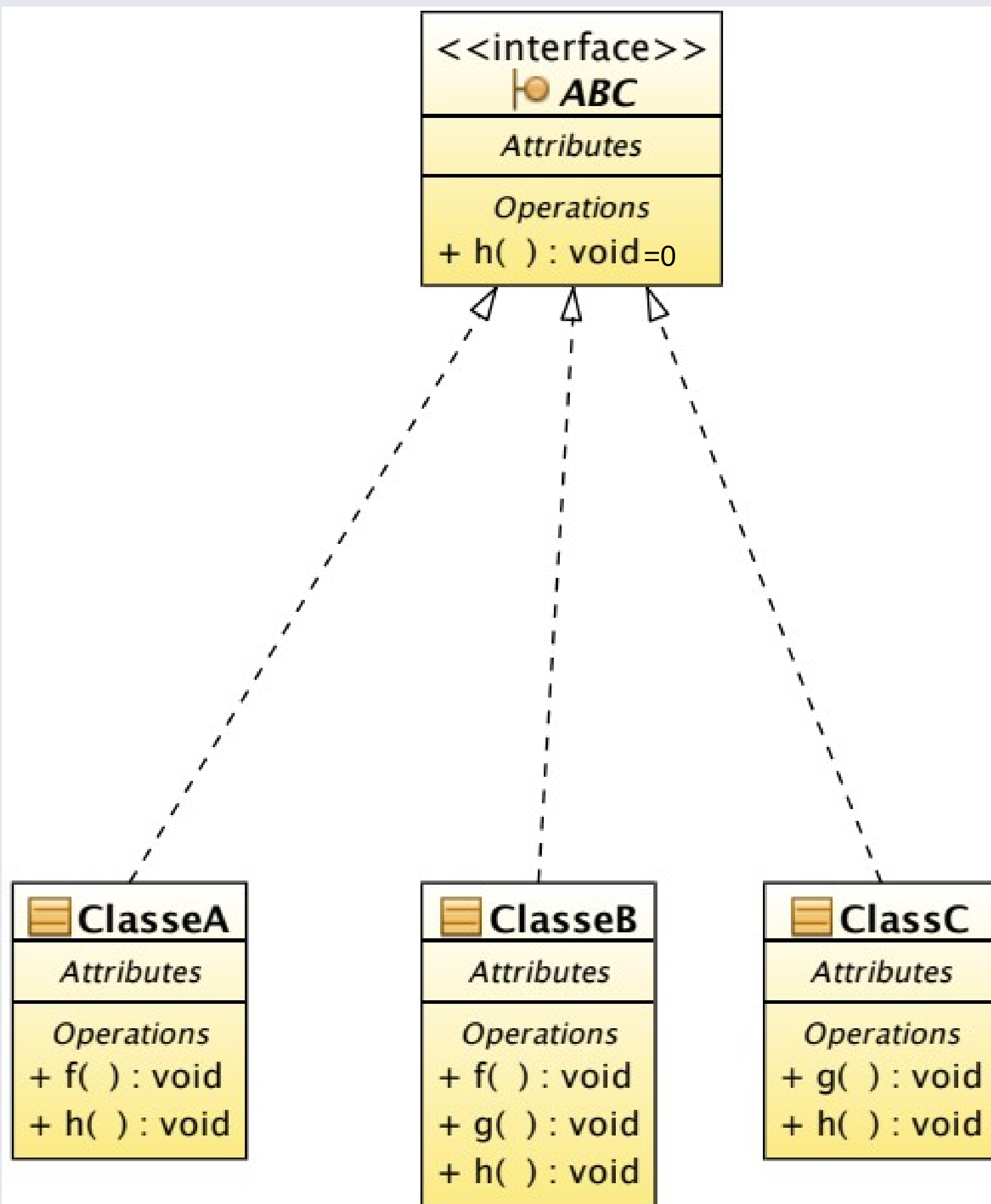
Une classe réalise une interface

Une interface en étend une autre









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

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 {  
    public :  
        A (const A&)=delete;  
};
```

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

Les méthodes deleted

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

On pouvait avoir un résultat quasiment similaire en utilisant « private »

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

Les méthodes deleted

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

On pouvait avoir un résultat quasiment similaire en utilisant « private »

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

Avec une petite nuance (copie possible en interne)

Les méthodes deleted

On utilise essentiellement `=delete` pour empêcher le compilateur d'ajouter des méthodes implicites : copie, affectation (et casting).

Les méthodes deleted

On utilise essentiellement `=delete` pour empêcher le compilateur d'ajouter des méthodes implicites : copie, affectation (et casting).

Naturellement `=delete` ne se combine pas avec `=0...`

Pour info, il existe aussi un `=default` ...

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

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

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 (s'il a prévu d'essayer qq chose et d'envisager l'échec)

```
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

```
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)

Il y a une subtilité si TypeErreur est une référence ou pas.

Avec une référence : compatible avec l'héritage

Sans référence : compatible avec l'héritage si la copie est permise


```
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)

Il y a une subtilité si TypeErreur est une référence ou pas.

Avec une référence : compatible avec l'héritage

Sans référence : compatible avec l'héritage si la copie est permise

complément un peu tricky : la règle de matching sur les arguments des fonctions, et le matching des exceptions n'est pas tout à fait la même .On pourrait s'attendre, si on dispose d'un constructeur de A à partir d'un B, à ce qu'un B puisse être rattrapé par un A. Ce n'est pas le cas, seul le sous typage est pris en compte. De toutes façons: utilisez les références !

Dans cet exemple, si un test est vérifié on choisi de provoquer (lancer) une erreur

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

mais l'allocation mémoire
n'est pas rendue ...

```
int main () {  
    try { f(); }  
    catch (...) { cout << "catch" ; };  
    return EXIT_SUCCESS;  
};
```

Dans cet exemple, si un test est vérifié on choisi de provoquer (lancer) une erreur

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

mais l'allocation mémoire
n'est pas rendue ...

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

Dans cette version l'objet x est détruit
avant l'affichage de « catch »

```
int main () {  
    try { f(); }  
    catch (...) { cout << "catch" ; };  
    return EXIT_SUCCESS;  
};
```

Dans cet exemple, si un test est vérifié on choisi de provoquer (lancer) une erreur

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

mais l'allocation mémoire
n'est pas rendue ...

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

Dans cette version l'objet x est détruit
avant l'affichage de « catch »

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

Ici, le concepteur l'a bien prévue

```
int main () {  
    try { f(); }  
    catch (...) { cout << "catch" ; };  
    return EXIT_SUCCESS;  
};
```


Dans cet exemple, si un test est vérifié on choisi de provoquer (lancer) une erreur

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

mais l'allocation mémoire
n'est pas rendue ...

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

Dans cette version l'objet x est détruit
avant l'affichage de « catch »

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

Ici, le concepteur l'a bien prévue

```
int main () {  
    try { f(); }  
    catch (...) { cout << "catch" ; };  
    return EXIT_SUCCESS;  
};
```

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

Que se passe t'il lorsque l'échec a lieu à la construction ?

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

```
int main () {  
    try { B b; } catch (...) { cout << "catch" ; }  
}
```

Que se passe t'il lorsque l'échec a lieu à la construction ?

B
catch

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

il n'y a pas d'appel au destructeur de B.
(l'objet n'a pas existé)

```
int main () {  
    try { B b; } catch (...) { cout << "catch" ; }  
}
```

Autre exemple

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

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

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

```
int main () {  
    try { C c; } catch (...) {cout << "catch" ; }  
}
```



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

Autre exemple

```
A
B
mort A
catch
```

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

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

pas d'appel au destructeur de B (ni de C) (ils n'ont pas existé) mais « a » détruit

```
int main () {
    try { C c; } catch (...) {cout << "catch" ;}
}
```

Que se passe t'il lorsque l'échec a lieu à la destruction ?

Dans ce cas la situation est mauvaise ...

Que se passe t'il lorsque l'échec a lieu à la destruction ?

Dans ce cas la situation est mauvaise ...

Le mécanisme est inadapté.

On rappelle qu'une erreur lorsqu'elle est lancée :

- remonte les appels jusqu'au catch adapté
- détruit les objets définis dans les blocs parcourus

L'idée est de conserver et de traiter l'erreur dans le catch.

Si les destructions lancent d'autres erreurs (en plus), que faire ?

Laquelle conserver ?

Exemple d'échec à la destruction

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

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

```
int main () {  
    try { f(); }  
    catch (...) { cout << "catch"; }  
};
```

La 1ère erreur lancée est de type string,
elle enclenche en remontant la destruction de « a » qui lance un int
Le catch devrait rattraper n'importe quoi (...) mais ...

Exemple d'échec à la destruction

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

```
A  
mort A  
terminate called after  
throwing an instance of  
'int'
```

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

```
int main () {  
    try { f(); }  
    catch (...) { cout << "catch"; }  
};
```

La 1ère erreur lancée est de type string,
elle enclenche en remontant la destruction de « a » qui lance un int
Le catch devrait rattraper n'importe quoi (...) mais ...

Exemple d'échec à la destruction

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

```
A  
mort A  
terminate called after  
throwing an instance of  
'int'
```

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

```
int main () {  
    try { f(); }  
    catch (...) { cout << "catch"; }  
};
```

La 1ère erreur lancée est de type string,
elle enclenche en remontant la destruction de « a » qui lance un int
Le catch devrait rattraper n'importe quoi (...) mais ...

Il y a ici une concurrence entre l'erreur 1 et "Erreur Voulue".

Il est difficile d'établir une règle indiscutable pour décider laquelle gérer.

Le mécanisme est inadapté, alors il a été décidé qu'une exception dans un destructeur sera **irrattrapable**

```
class A {};
```

En particulier ...

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



```
class A {};
```

En particulier ...

```
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 devait préciser quelles exceptions risquent de lancer les méthodes.

Le programmeur est alors syntaxiquement contraint :

il doit soit les rattraper, soit les relancer.

Le mécanisme correspondant en c++ est **deprecated**

(ca a été essayé, jugé inefficace/coûteux, et abandonné)

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

Les classes internes

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

spoiler :

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

B est utilisable ici
puisque'elle est
déclarée publique

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

sa définition s'écrit
dans le .cpp de A

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

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();  
}
```

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

```
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();  
}
```

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

g() B

```
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();
}
```

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";
};
```

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();
}
```

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";
    n++;
};
```

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

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

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

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

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

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 Makefile ajoutez les options de compilation :

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

et aucun problème avec vsodium

Installation :

sous windows + codeblocks ...

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

Privilégiez un travail sous unix.

Avec mac ça s'est bien passé également (retour chargés de TD)

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'
        ... // mise en place des objets graph.
        app.display(); // Affichage effectif
    } // fenêtrée fermée
    return EXIT_SUCCESS;
}
```

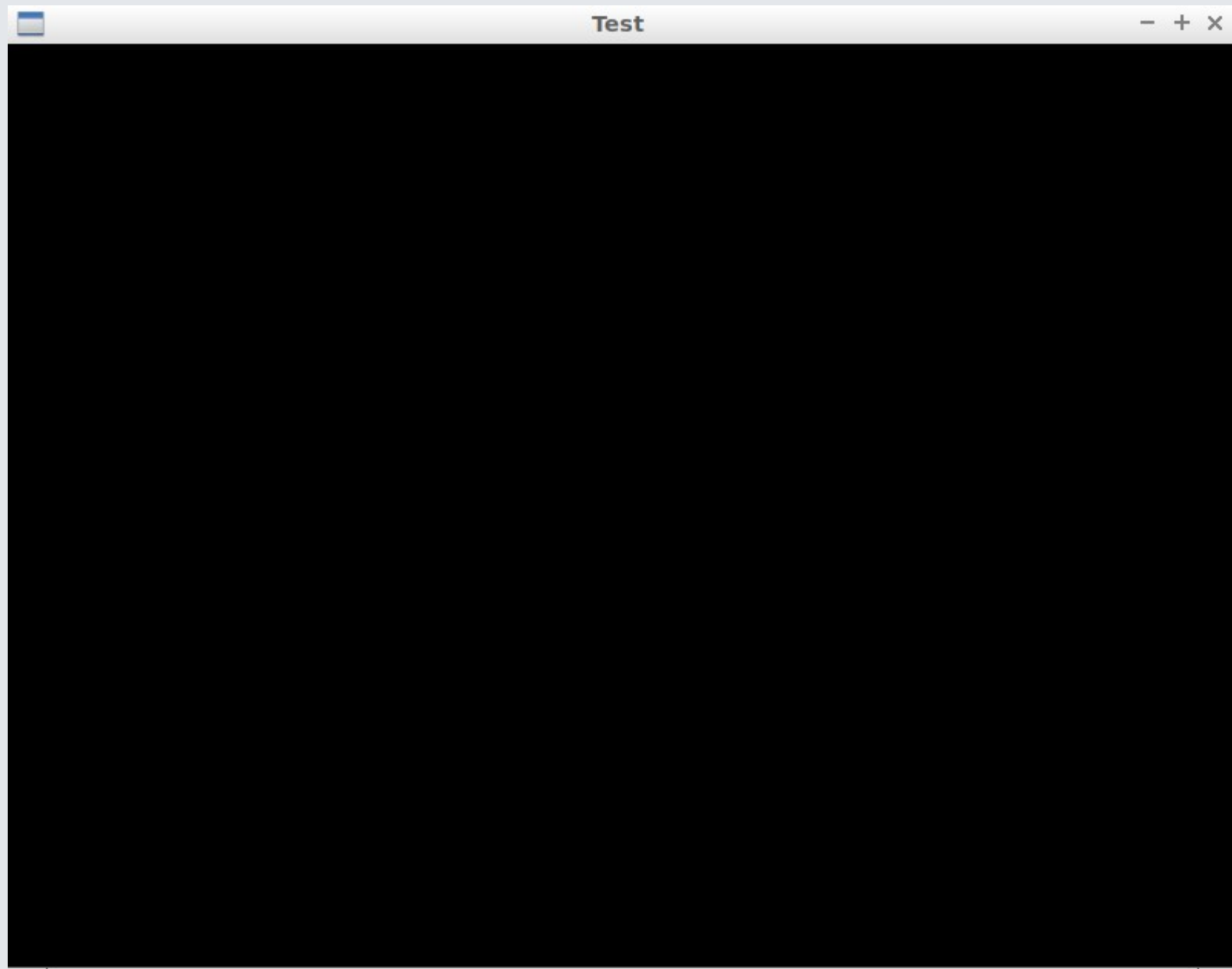
Une fenêtre

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();
```



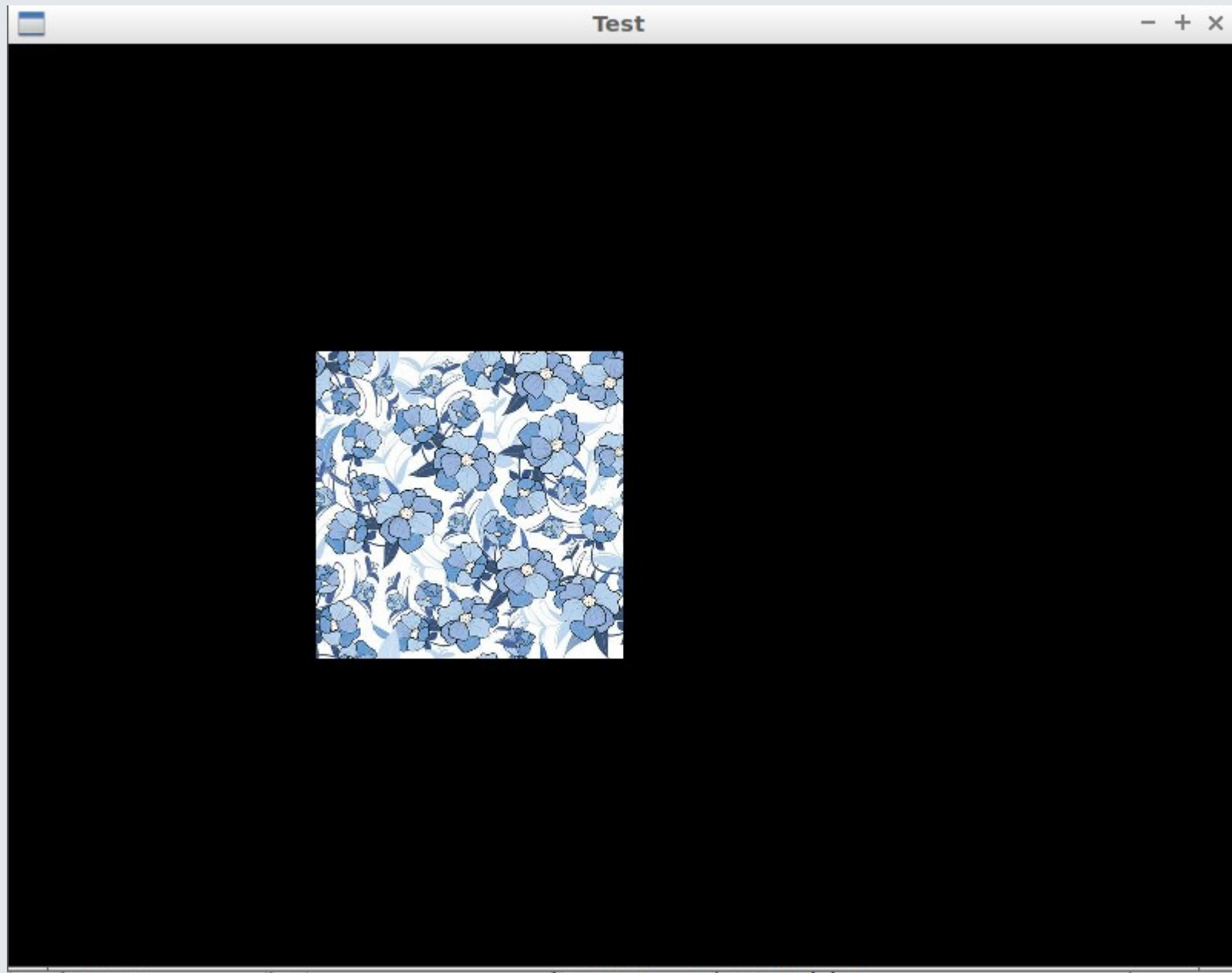

Ajout d'un élément graphique :

```
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

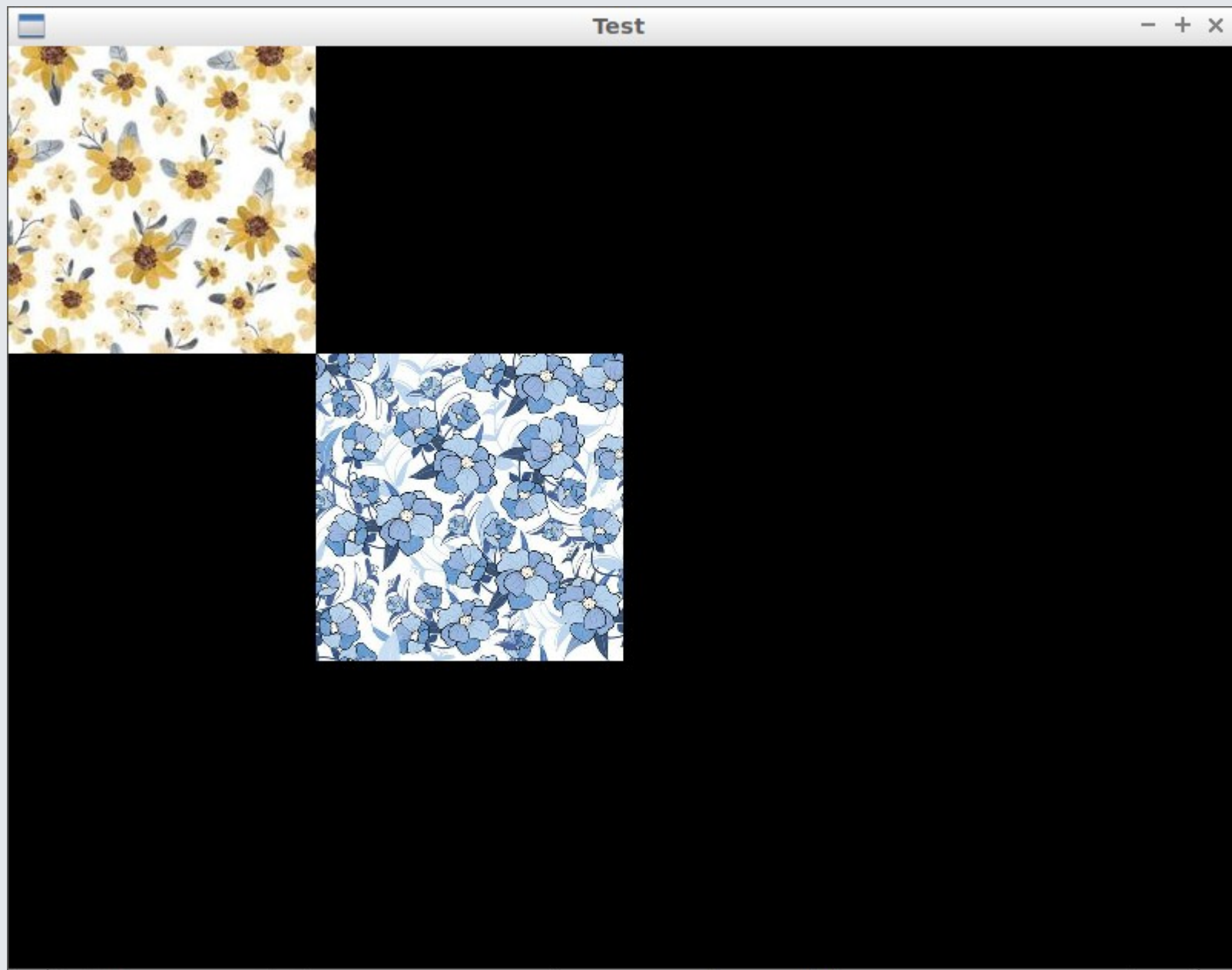
```
app.clear();  
app.draw(tuile);  
app.display();
```



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(Keyl  
    current -> move(1, 0);
```

le fait d'avoir
appuyé

le fait d'être
appuyée

...

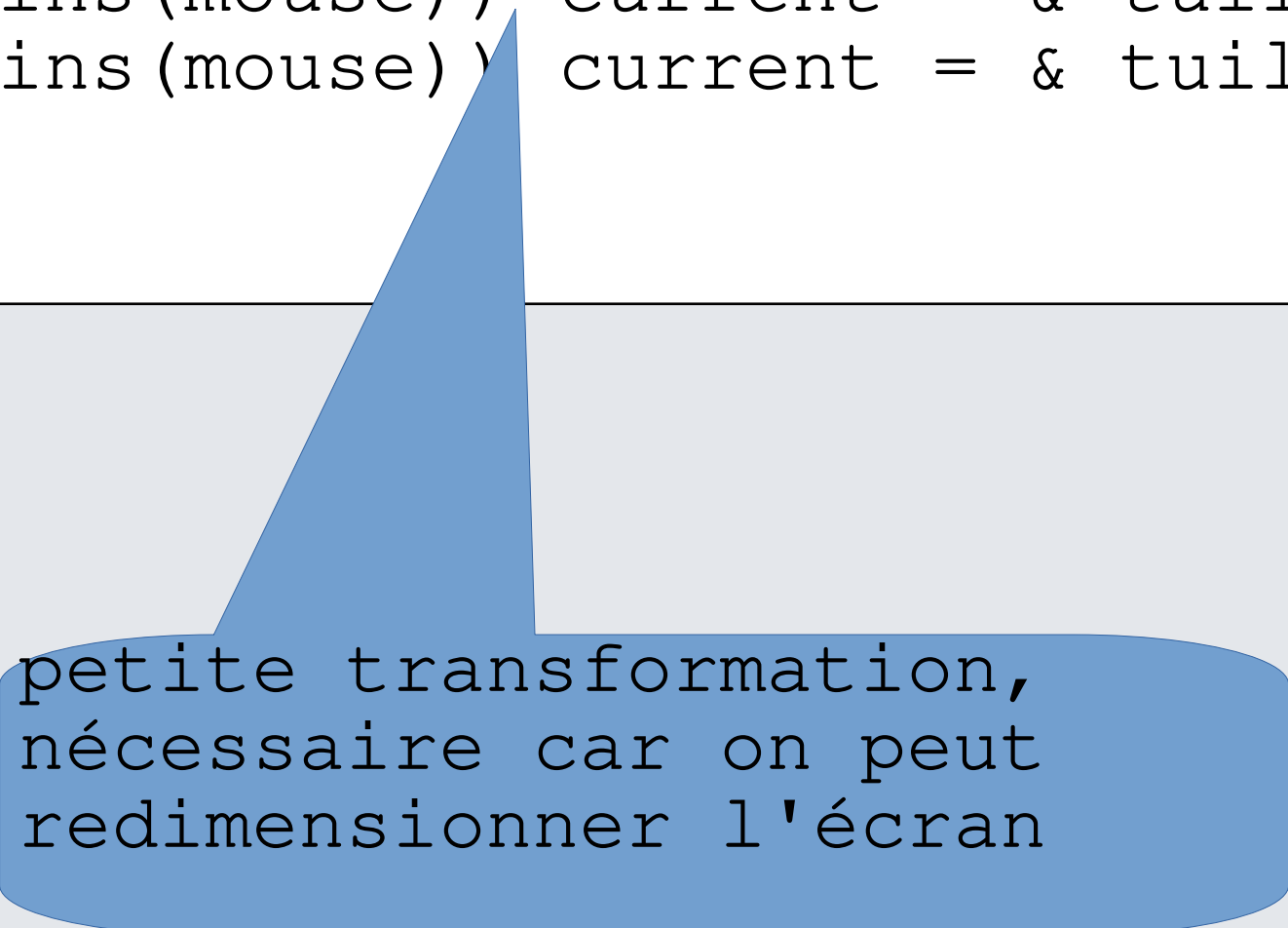
}

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>

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

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'ambigüité

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>

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**

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";  
}
```

```
#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";  
}
```

```
#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";  
}
```

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);
};
```

Premier jet : 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);
};
```

très 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);  
};
```

invariant

ce getter laisse
l'objet invariant

const et
référence

Les sommets sont étiquetés, et on prévoit qu'ils
seront parcouru : 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 auprès du GC

```
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 ...
arcs ?

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 ici
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 ici
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();  
    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;  
}
```

Petites discussions en plus

Probablement qu'il faut neutraliser delete
(ou le rendre compatible avec le GC)