Compléments C++

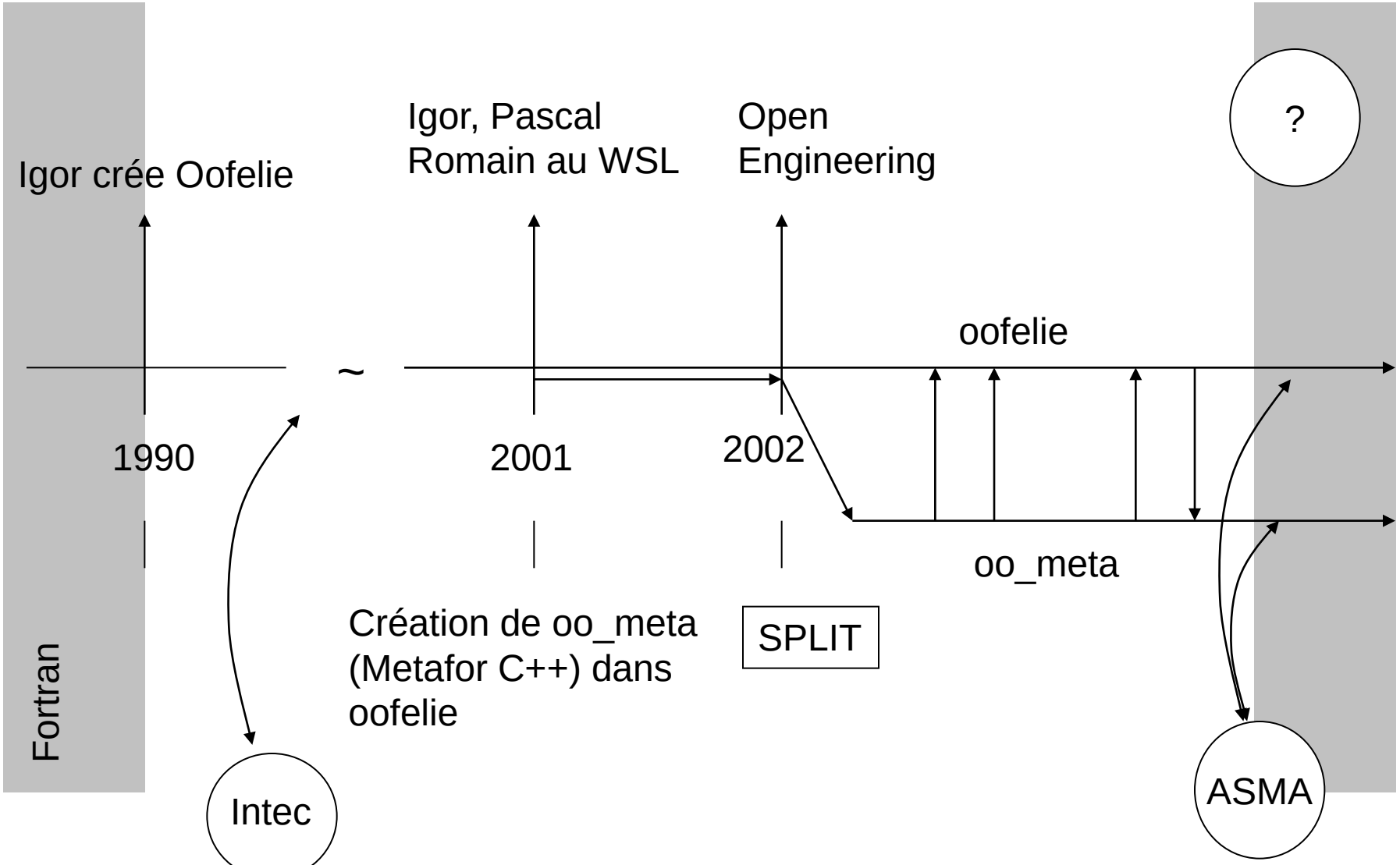
Romain BOMAN

novembre 2003

Contenu

Contexte

Contexte


Igor crée Oofelie
Igor, Pascal
Romain au WSL
Open
Engineering
?
oofelie
~
1990
2001
2002
oo_meta
Création de oo_meta
(Metafor C++) dans
oofelie
SPLIT
Fortran
Intec
ASMA

Contexte

Constat à l'ULg (hors cadre Oofelie)

- Pas de vision long terme
- Informatique != science
- Vision individualiste

Constat à l'ULg (cadre Oofelie)

- Règles de programmation non respectées
- Peu de communication sur stratégie long terme
- Bluff (monumental)

Mon but :

aider les gens à programmer correctement dans Oofelie et échanger des idées

Il faut compléter la présentation d'Igor (qui ne tient pas compte du contexte)

- Vision plus **orientée objet**
- Lier les concepts étudiés à Oofelie
- Vision moins « guru » (utilité UML – assembleur pour débutant?)
- Insister sur les règles
- Valoriser Oofelie malgré ses défauts en proposant des solutions
- Dénoncer le bluff malgré la censure

Introduction

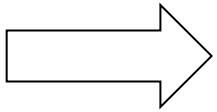
Introduction

C++ en temps que langage « orienté objet » (OO)

Les problèmes :

1. ~~Le C++~~ est lié au C (non OO)
2. Le C++ ne dispose pas de gestion avancée (OO) de mémoire (cfr. Java, Python)
3. Le C++ peut être (très) lent

Oofelie / Metafor



3 Règles de base

Vu le succès d'Oofelie et Metafor, il n'est plus aujourd'hui nécessaire de nier ces vérités (les accros du Fortran sont – enfin – convaincus)

Des solutions / méthodes sont exposées dans cette présentation

Introduction

Problèmes C++ & Solutions C++

Problème #1 Lié au C → Effort personnel

```
// Mon premier programme C++

#include <iostream>

void main()
{
     std::cout << "Hello!\n";
     std::cout << "Good bye!\n";
}
```

C'est du Fortran traduit en grammaire C++

```
#include <iostream>

class World
{
public:
    World ()  { std::cout << "Hello!\n"; }
    ~World () { std::cout << "Good bye!\n"; }
};

World TheWorld;

void main() {}
```

C'est de l'orienté objet traduit en grammaire C++

A LIRE ! (c'est gratuit) →

Industrial Strength Programming – C++ In Action
Bartosz Milewski (Microsoft)
http://www.relisoft.com/book/index.htm

Introduction

Problèmes C++ & Solutions C++

- Le C++ aide (et seulement aide) à faire du code OO.
- Inversément, il est possible de faire du OO en C, voire en Fortran !

Mauvaise nouvelle :

- de nombreuses parties d'Oofelie (et de Metafor!) sont du Fortran déguisé en C++

Bonne nouvelle :

- c'est tout à fait normal !
- Il est impossible de faire de l'orienté objet du premier coup (c'est incrémental)

C'est la raison d'être du **refactoring**

Introduction

Problèmes C++ & Solutions C++

Discours dangereux : « Do not make each bit an object » :

- Un programme séquentiel est une petite série de grande routines.
- Un programme OO est une grande série de petite routines.

> Un programmeur qui vient du Fortran ou du C va toujours faire de trop gros objets !

- On n'enrichit jamais une classe (sauf exception)
- On la découpe (systématiquement, aveuglement) pour :
  - séparer les concepts (plus clair pour moi et les autres)
  - minimiser les dépendances (librairies)
  - réutiliser (à mort le code dupliqué!)

> REGLE 1 : Je découpe mon code

Introduction

Problèmes C++ & Solutions C++

Problème #2 « Gestion des ressources » (de la mémoire)

Constat	Solution proposées par le C++
Idée : « Le C++ est un casse-tête pour la gestion de la mémoire » C'est VRAI : la preuve = Oofelie sous Purify*!	• Ne pas utiliser de pointeurs • Références • Null Objects • Factory Objects • Compteurs de références • Smart pointers (auto_ptr<>) • STL
Des solutions existent MAIS	Utilisation **malheureusement facultative**

Philosophie : Chaque objet acquiert des ressources et les libère

Règle 2 : je libère la mémoire que j'utilise (question de civisme)

* Purify = analyseur de mémoire de Rational

Introduction

Problèmes C++ & Solutions

Problème #3 Lenteur

Constat	Solution proposées par le C++
Idée : « Le C++ est général donc lent » C'est VRAI : la preuve = Oofelie sous Quantify*! (facteur 300 observé sur Metafor)	• Profiling (cfr. State) • Inlining • const • STL • Lecture du source (Material::get()) • Changer de compilateur

Des solutions existent MAIS Demandent du temps

Philosophie OO: Pour gagner du temps, il faut d'abord en perdre

Règle 3 : j'optimise mon code (lorsque ça marche)

* Quantify = analyseur de mémoire de Rational

Rappels brefs

Leçon #1 – Notions de base

- Rappel du vocabulaire

Leçon #2 – UML & Design Patterns

- Rappel de ce qui est utile à un débutant
- Application à Oofelie

Rappels

Leçon #1 – Notions de base

Vocabulaire de base de l’orienté objet en général – du C++ en particulier

- Objet / classe / instanciation
- Encapsulation des données (attributes, properties (set/get), public/protected/private)
- Abstraction (fonctions virtuelles**, interface**)
- Polymorphisme (héritage (simple, multiple, virtuel))
- Composition, Agrégation (idem par refs)
- Constructeur/Construction par copie/Destructeur
- Surdéfinition de fonctions
- static
- templates
- const

Ce sont les outils pour faire de l’OOP

Rappels

Leçon #2 – UML

- UML = Unified Modeling Language
- Conventions pour **diagrammes de classes**, d'objets, de séquence, d'états...
- Débutant : savoir lire un diagramme (rien de plus).
- Peut être généré automatiquement (Rational, Doxygen, Toegether, ...) !
- Utile aux personnes qui communiquent leurs idées.

Mémo	
héritage	= flèche vers la classe mère
aggrégation	= flèche avec losange à la base
association	= flèche simple

! Le UML est aussi un outil pour « faire sérieux » !

Rappels

Leçon #2 – Design Patterns

- Modèles à avoir sous la main (Voir mon site web – voir VTK)
- Peu utilisé dans Oofelie

➢ Création

- Factory method (`NumberedObject::getClone()`)*
- Abstract Factory, Builder, etc.

➢ Structure

- Adapter (`NurbsCurve`)*
- Composite (`PhySet`)
- Proxy (`FastPhySet`)
- Facade, Bridge, ...

➢ Comportement

- Chain of Responsibility (`PhySet::get_properties()`)
- Observer (`CurveOnFile`)*
- Command, Iterator, State, Strategy, Template Method,...

* Classes Metafor

Compléments

Complément :
Gestion des erreurs

- Erreurs « normales » du programme ↔ Exceptions
- Erreurs « fatales » / debug ↔ Assertions

Gestion des erreurs

Utilisation des exceptions (1)

Exception = événement qui ne se produit pas souvent ~ erreur.

Cas concret : le Jacobien négatif

- Etape #1 : Créer une (hiérarchie de) classe(s) définissant une (des) exception(s)

```
class OException
{
public :
        virtual void print() = 0;
};

class NegativeJacobian : public OException
{
        const Element &element;
        int            gaussPointNo;
public :
        NegativeJacobian(const Element &el, int n) :
            element(el), gaussPointNo(n) {}
        virtual void print()
        {
                cout << “Negative Jacobian: element " << element;
                cout << " GaussPoint No " << gaussPointNo << " !” << endl;
        }
};
```

Gestion des erreurs

Utilisation des exceptions (2)

- Lancer l'exception lors d'une erreur :

```
void
MyElement::fill_int_forces()
{
        // ... loop over Gauss Points
        double detJ = computeJacobian(gaussPointNo) ;
        if(detJ <= 0) throw NegativeJacobian(*this, gaussPointNo);
        // ...
}
```

- « Attraper » l'exception et gérer la situation :

```
void
VectorStr::int_forces()
{
        try
        {
                // ... loop over elements
                element.fill_int_forces();
                // ...
        }
        catch( OException &e )
        {
            e.print();
        }
}
```

Gestion des erreurs

Utilisation des assertions

- Macro
- Ajout automatique de code de vérification en mode « debug »
- Utile pour les phases de développement
- Ne pénalise pas le code optimisé.
- Convient uniquement pour les « erreurs fatales »
- Nécessite `#include <assert.h>`

➢ Exemple simple

```
int
myRatio(int a, int b)
{
        // cette function ne marche que pour b!=0
        assert(b!=0);
        return a/b;
}
```

➢ Exemple Oofelie

```
Domain *domain = (Domain*)get_properties(DOMAIN_PO);
assert(domain!=NULL);
```

Complément :
comprendre le code « guru »

Les choses à éviter

Pour faire du OO en C++

- Ce qu'il faut éviter en C (`goto`)
- Ce qu'on fait en C :
  - L'arithmétique de pointeurs (cfr. `locel--` dans `Element`)
  - Les structures (cfr. `typedef struct`)
  - Les conversions « utilisateur » (non encapsulées)
  - Les conversions implicites via les types de base (int et constructeurs associés)
- Les variables statiques

Variables statiques

1. Problème d'initialisation

   • Ordre aléatoire d'initialisation (appels au constructeur) si plusieurs fichiers .obj
   • Voir l'exemple de refactoring

2. Problème de parallélisation

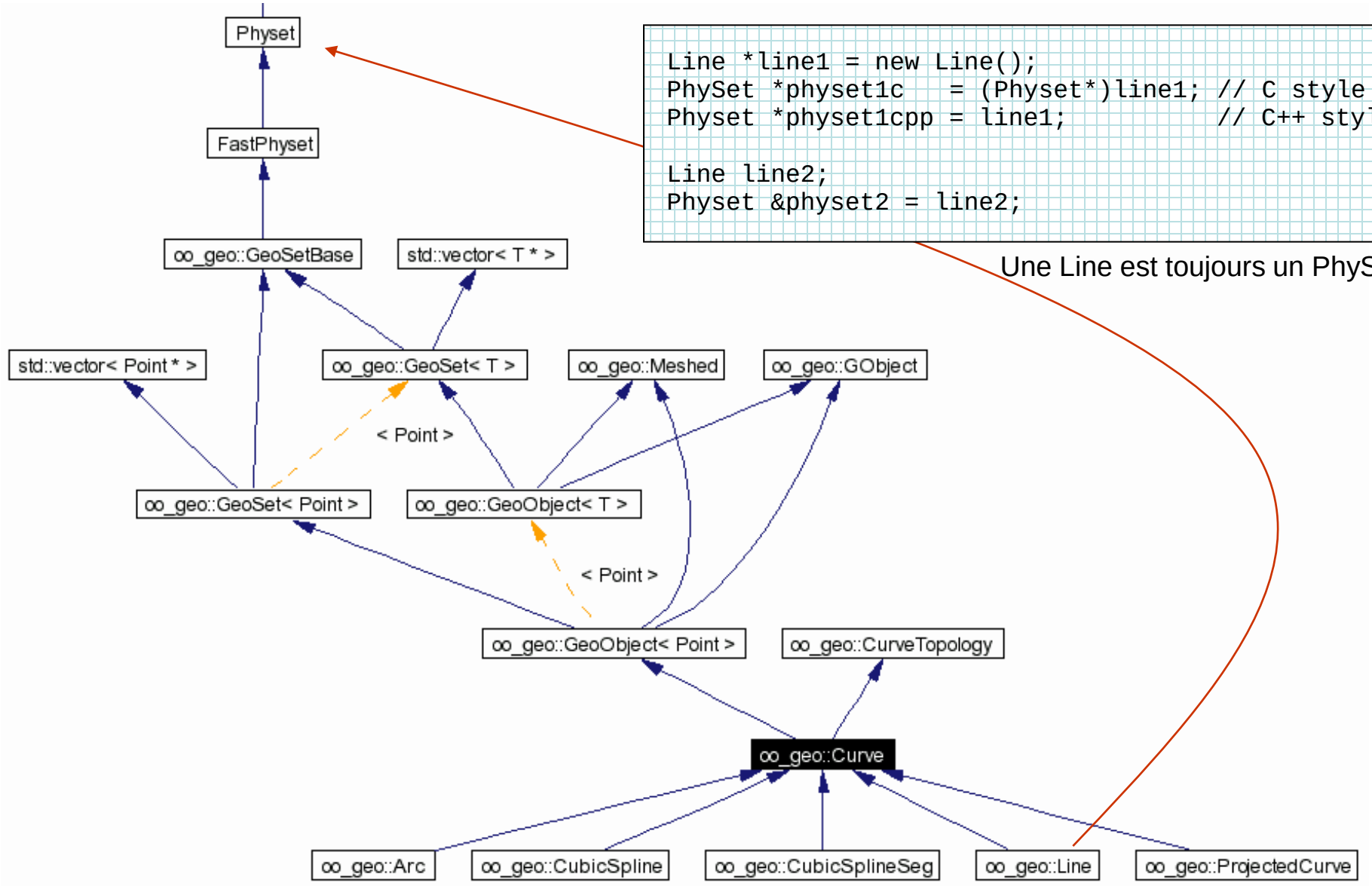
   • Plusieurs threads → mécanisme sophistiqué d'accès aux variables partagées

Solutions

- Ne pas utiliser sauf si vraiment nécessaire
- Singletons
- Création d'objets non initialisés ou pointeurs & appel explicite et ordonné à des fonctions d'initialisation au début du programme.

Opérations de cast

Upcast – Opération sure & implicite




```
Line *line1 = new Line();
PhySet *physet1c   = (Physet*)line1; // C style
Physet *physet1cpp = line1;          // C++ style

Line line2;
Physet &physet2 = line2;
```

Une Line est toujours un PhySet

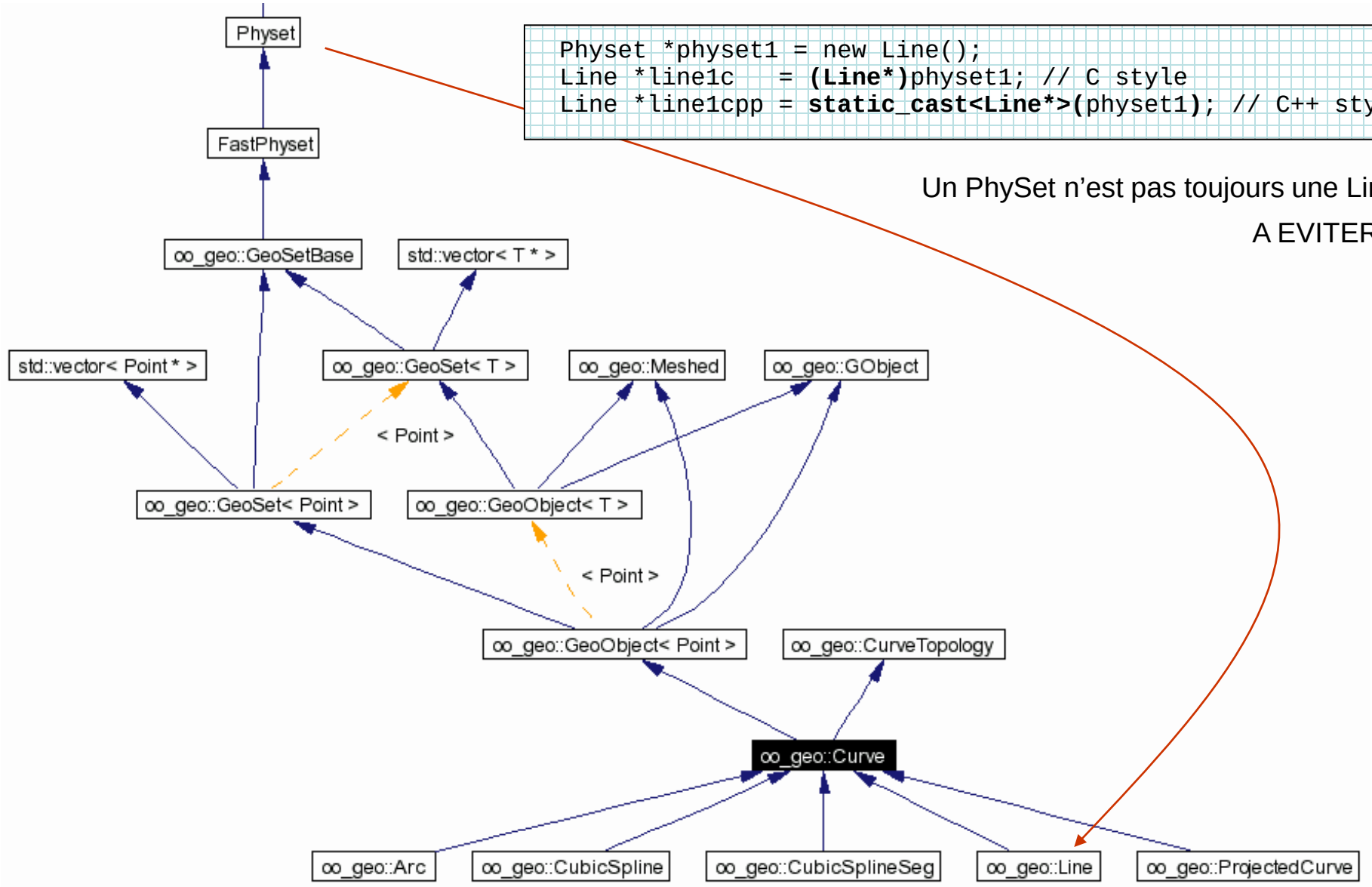
Opérations de cast

Downcast – Opération risquée & explicite

```
Physset *physet1 = new Line();
Line *line1c   = (Line*)physet1; // C style
Line *line1cpp = static_cast<Line*>(physet1); // C++ style
```

Un PhySet n'est pas toujours une Line

A EVITER !

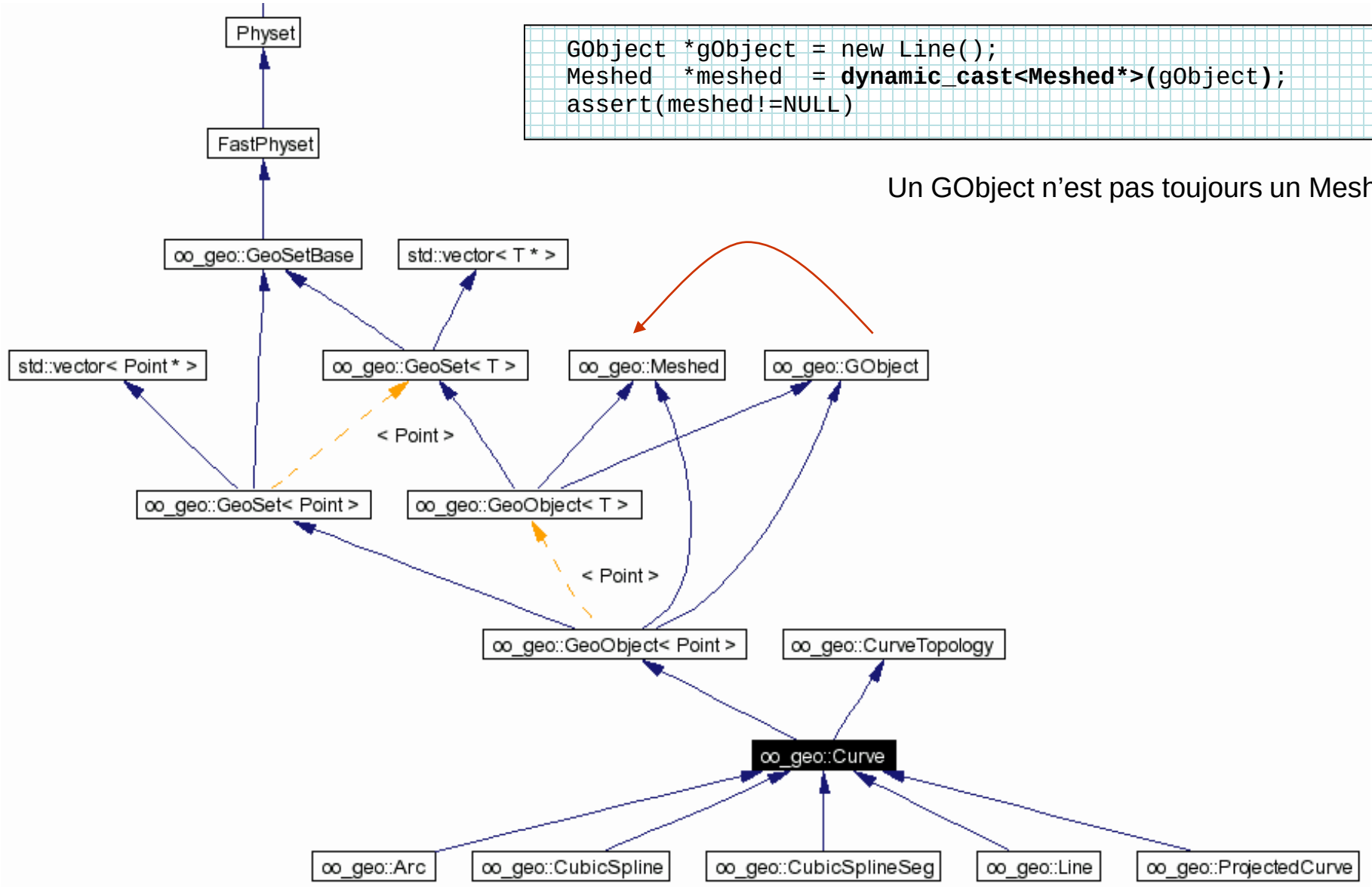


Opérations de cast

Dynamic cast – Opération risquée MAIS contôlée

```
GObject *gObject = new Line();
Meshed  *meshed  = dynamic_cast<Meshed*>(gObject);
assert(meshed!=NULL)
```

Un GObject n'est pas toujours un Meshed



Complément : Utiliser la STL

STL

Standard Template Library

Idées :

- Unifier les structures de données courantes => norme ANSI
- Utiliser des Templates pour garantir un typage fort

Contenu

- Conteneurs (vecteurs, listes, maps, queues, ...)
- Strings
- Algorithmes
- Gestion mémoire (smart pointers)
- IO Streams

Remarques

- Standard relativement nouveau (surtout pour les streams)
- Optimisé !
- Namespace std

STL

Pré requis - Namespaces

Permet de définir des « packages »

```
namespace myPackage
{
        int  data;
        class MyClass
        {
        public:
                MyClass();
                // ...
        };
        void myFunction();
};
```

La STL est dans le namespace **std**

Utilisation (2 manières)

```
void main()
{
        myPackage::data = 3 ;
        myPackage::myClass object;
        //...
}
```

```
using namespace myPackage;

void main()
{
        data = 3 ;
        myClass object;
        //...
}
```

STL

Utilisation des conteneurs

Exemple : template<class T> vector

sans STL

```
// initialisation
int mySetSize = 3;
double *mySet = new double[mySetSize];
mySet[0] = 1.0;
mySet[1] = 2.0;
mySet[2] = 3.0;

// ajout d'un element
double *mySetTmp = new double[mySetSize+1];
for(int i=0; i<mySetSize; ++i)
        mySetTmp[i] = mySet[i];
mySetTmp[mySetSize++] = 4.0;
delete[] mySet;
mySet = mySetTmp;

// utilisation
for(int j=0; j<mySetSize; ++j)
    cout << mySet[j]

// fin du programme
delete[] mySet;
```

avec STL

```
// initialisation

std::vector<double> mySet(3);
mySet[0] = 1.0;
mySet[1] = 2.0;
mySet[2] = 3.0;

// ajout d'un element
mySet.push_back(4.0);




// utilisation
for(int j=0; j<mySet.size(); ++j)
    cout << mySet[j]

// fin du programme
```

STL

Utilisation des conteneurs

Avantages

- Taille du code minimale
- Pas de pointeur visible
- Gestion de mémoire transparente et implicite (cfr. appel au destructeur)
- Pas besoin de stocker la taille du tableau
- Se comporte comme un « double * »
- Nombreuses fonctions (at(), resize(), reserve(), empty(), capacity(), ...)
- Facilement interchangeable avec un autre conteneur (vector ↔ list)
- Algorithmes standard dans `<algorithm>`
- Evolution future garantie (optimisations, etc)

Inconvénients

- Aucun aujourd'hui (la norme est respectée partout)

STL

itérateurs

But: Rendre les algorithmes indépendants du conteneur

Exemple:

Code original

```
typedef std::vector<double>::iterator mySetIterator

for(mySetIterator it=mySet.begin(); it!=mySet.end() ; ++it)
    cout << *it << endl;
```

Si `mySet` devient une **`std::list<double>`**

```
typedef std::list<double>::iterator mySetIterator

for(mySetIterator it=mySet.begin(); it!=mySet.end() ; ++it)
    cout << *it << endl;
```

STL

conteneurs - séquences

Vecteurs : **`std::vector<T>`**

- Optimisé pour l'accès aléatoire : operator[]
- Comparable à T *

Listes : **`std::list<T>`**

- Optimisé pour l'ajout/suppression d'élements
- Opérations spéciales :

  insert(p,x) insert(p, first, last)
  erase(p) erase(first,last)
  clear()
  sort(), unique()
  merge(l2)
  push_front(), pop_front()

STL

conteneurs - séquences

File d'attente à double accès : **`std::deque<T>`**

- Optimisé pour l'accès aux éléments extrêmes

Pile : **`std::stack<T>`**

- Optimisé pour push() = push_back() et pop() = pop_back()

Queue : **`std::queue<T>`**

- Optimisé pour push_back() = push() et pop_front() = pop()

STL
conteneurs associatifs

Maps: **`std::map<K,V>`**

- Optimisé pour la recherche d'élements
- Opérations spéciales :

  find(k), operator[](k)

- Exemple :

```
std::map<std::string,double> bookPrices;
myMap[“Design Pattern”] = 45;
myMap[“Refactoring”]    = 52;
```

Maps à entrées multiples: **`std::multimap<K,V>`**

- Permet plusieurs valeurs V pour la même clef K

Ensembles : **`std::set<K>`**, **`std::multiset<K>`**

- Ce sont des maps avec uniquement des clefs K

STL

algorithmes

Nombreux algorithmes disponibles dans `<algorithm>`

- for_each() : exécution sur tous les éléments
- find() find_if() : recherche
- count() : compte des éléments
- copy() : copie
- replace(), replace_if() : remplace des éléments
- rotate() : fait tourner des éléments
- sort() : trie une séquence
- reverse() : inverse l'ordre de la séquence
- min(), max() ...
- ...

Pour en savoir plus → Stroustrup

STL

algorithmes

utilisation simple

```
list<int>::iterator p = find(li.begin(), li.end(), 42);
if(p!=li.end()) { ... }
```

utilisation avancée (objets fonctions)

```
class Object;

class PrintElement
{
public:
        void operator()(const Object& obj) { cout << obj << endl; )
};

std::list<Object> li;
for_each(li.begin(), li.end(), PrintElement());
```

STL

autres fonctions de la librairie

Chaînes de caractères : `std::string`

- Déclarées dans `<string>`
- Remplace avantageusement « char * »

```
std::string s1 = "Hello";
std::string s2 = "world";
std::string s3 = s1 + "," + s2 + "!\n";

s3.replace(0,5, "Goodbye");

if( s3.substr(2,5) == "ll")
{
      cout << s3.c_str() << endl;
}
```

IO – Streams : `std::ostream`, `std::istream`

- Attention ! ne pas confondre
  - les anciennes streams `<iostream.h>`
  - les nouvelles streams `<iostream>` définies dans le namespace std
- Oofelie => anciennes streams
- Partie la moins portable de la STL !

Complément:
Eviter les pointeurs

Eviter les pointeurs

```
class MyObject
{
        int userNo;
public:
        int getNo() const {return userNo;}
};
```

Objet numéroté

```
class MySet
{
        int size;
        MyObject **arrayOfObjects;
public:
        MyObject *operator[](int internalNo);
        MyObject *getObjectByUserNo(int userNo);
        int       searchSimilarObjectInternalNo(MyObject const*object);
        // ...
};
```

Ensemble d'objets numérotés

Règle:

Un pointeur est utile lorsqu'une référence est insuffisante

- Allocation de mémoire
- Changement de valeur

Eviter les pointeurs

pointeur inutile

```
MyObject *
MySet::operator[](int internalNo)
{
        assert(internalNo>=0 && internalNo<size);
        return arrayOfObjects[internalNo];
}

MyObject *
MySet::getObjectByUserNo(int userNo)
{
        int i;
        for(i=0; i<size; ++i)
                if(arrayOfObjects[i]->getNo()==userNo)
                        return arrayOfObjects[i];
        return NULL;
}

int
MySet::searchSimilarObjectInternalNo(MyObject const* object)
{
        assert(object);
        for(i=0; i<size; ++i)
                if(*arrayOfObjects[i]==*object)
                        return i;
        return -1;
}
```

pointeur inutile

Eviter les pointeurs

On remplace des pointeurs par des références

```
MyObject &
MySet::operator[](int internalNo)
{
        assert(internalNo>=0 && internalNo<size);
        return *arrayOfObjects[internalNo];
}

MyObject &
MySet::getObjectByUserNo(int userNo)
{
        int i;
        for(i=0; i<size; ++i)
                if(arrayOfObjects[i]->getNo()==userNo)
                        return arrayOfObjects[i];
        return MyObject::invalid();
}

int
MySet::searchSimilarObjectInternalNo(MyObject const& object)
{
        assert(object);
        int i;
        for(i=0; i<size; ++i)
                if(*arrayOfObjects[i]==object)
                        return i;
        return -1;
}
```

```
MyObject &
MyObject::invalid()
{
        static obj = MyObject(0);
        return obj;
}
```

- **Intérêt #1: l'objet nul est typé**

  Moins de risques d'erreurs

- **Intérêt #2 : inutile de tester la validité**

  Même si MyObject::invalid est passé, le code n'explosera pas.

- **Intérêt #3 : gestion mémoire**

  La question de « désallouer la référence » ne se pose pas (sauf si Object &ref= *new Object())

  L'origine des objets est mieux contrôlée.

Complément:
Typage fort du C++

Typage fort du C++

```
class MyObject
{
        int userNo;
public:
        int getNo() const {return userNo;}
};
```

mauvais choix : on risque de faire de l'arithmétique sur ce numéro ou d'indicer un tableau avec.

Nouvelle classe

```
class Number
{
        int userNo;
public:
        explicit Number(int no) : userNo(no) {}
        bool operator==(Number const& n) { return n.userNo==userNo;}
};
```

```
class MyObject
{
        Number userNo;
public:
        Number getNo() const {return userNo;}
};
```

- **Intérêt #1 :**

  constructeur explicite:
  int i=2;
  Number n(3); // autorisé
  n = i; // interdit
  array[n]; // interdit !

- **Intérêt #2 :**

  opérations limitées:
  n1==n2; // autorisé
  n3=n1+n2; // interdit !
  n1++; // interdit;

Typage fort du C++

Même chose pour le numéro interne ...

```
MyObject &
MySet::operator[](InternalNo internalNo)
{
        assert(internalNo.isValid() && internalNo<size);
        return *arrayOfObjects[(int)internalNo];
}

MyObject &
MySet::getObjectByUserNo(Number userNo)
{
        int i;
        for(i=0; i<size; ++i)
                if(arrayOfObjects[i]->getNo()==userNo)
                        return arrayOfObjects[i];
        return MyObject::invalid();
}

InternalNo
MySet::searchSimilarObjectInternalNo(MyObject const& object)
{
        int
        for(i=0; i<size; ++i)
                if(*arrayOfObjects[i]==object)
                        return InternalNo(i);
        return InternalNo::invalid();
}
```

```
class InternalNo
{
        int intNo;
public:
        explicit InternalNo (int no);
        static InternalNo &invalid();
        bool isValid()
        {
            return (intNo>=0);
        }
        void operator++();
        // ...
};
```

En interne, on utilise toujours un int

But :
identifier rapidement où l'utilisateur manipule des numéros internes en recherchant InternalNo dans le source.

Typage fort du C++

Encore plus loin...

```
class InternalNo
{
        const &MySet set;
        int intNo;
public:
        explicit InternalNo (MySet &s, int no) : s(set), intNo (no) {}
        static InternalNo &invalid();
        bool isValid()
        {
            return (intNo>=0 && intNo<set.size());
        }
        void operator++();
        // ...
};
```

Complément: Application STL

Application STL

```
class MySet
{
        int size;
        MyObject **arrayOfObjects;
public:
        MySet()
        {
            arrayOfObject=NULL; size=0;
        }
        ~MySet()
        {
             if(arrayOfObjects)
             {
                  for(int i=0; i<size; ++i)
                        delete arrayOfObjects[i];
                  delete []arrayOfObjects;
             }
        }
        // ...
};
```

doivent rester synchronisées

cette classe gère tous les objets

source d'erreur fréquente

```
class MySet
{
        std::vector<MyObject*> arrayOfObjects;
public:
        ~MySet()
        {
             if(arrayOfObjects.size())
                  for(int i=0; i<size; ++i)
                        delete arrayOfObjects[i];
        }
        // ...
};
```

constructeur inutile – destructeur simplifié

pas tjs possible

alternative à envisager :

```
class MySet
{
        std::vector<MyObject> arrayOfObjects;
public:
        // ...
};
```

plus de constructeur ni de destructeur

Refactoring

Refactoring

- Qu’est-ce que c’est?
- A quoi ça sert?

Refactoring

introduction

Refactoring = nettoyage d'un code dans le but de:

- permettre de nouveaux développements
- introduire des Design Patterns
- rendre le code « Orienté Objet »
- être lisible par d'autres développeurs

Refactoring = liste de « bad smells » pour identifier les endroits critiques
justifications des « bad smells »

Refactoring = liste de « recettes » systématiques pour modifier le code

Basé sur le travail de Martin Fowler
Refactoring – improving the Design of Existing Code
http://www.refactoring.com/

Refactoring

introduction

Hypothèses: (eXtreme Programming)

- Impossibilité de prévoir l'évolution d'un code!
- Phase d'analyse préliminaire indispensable mais toujours incomplète.

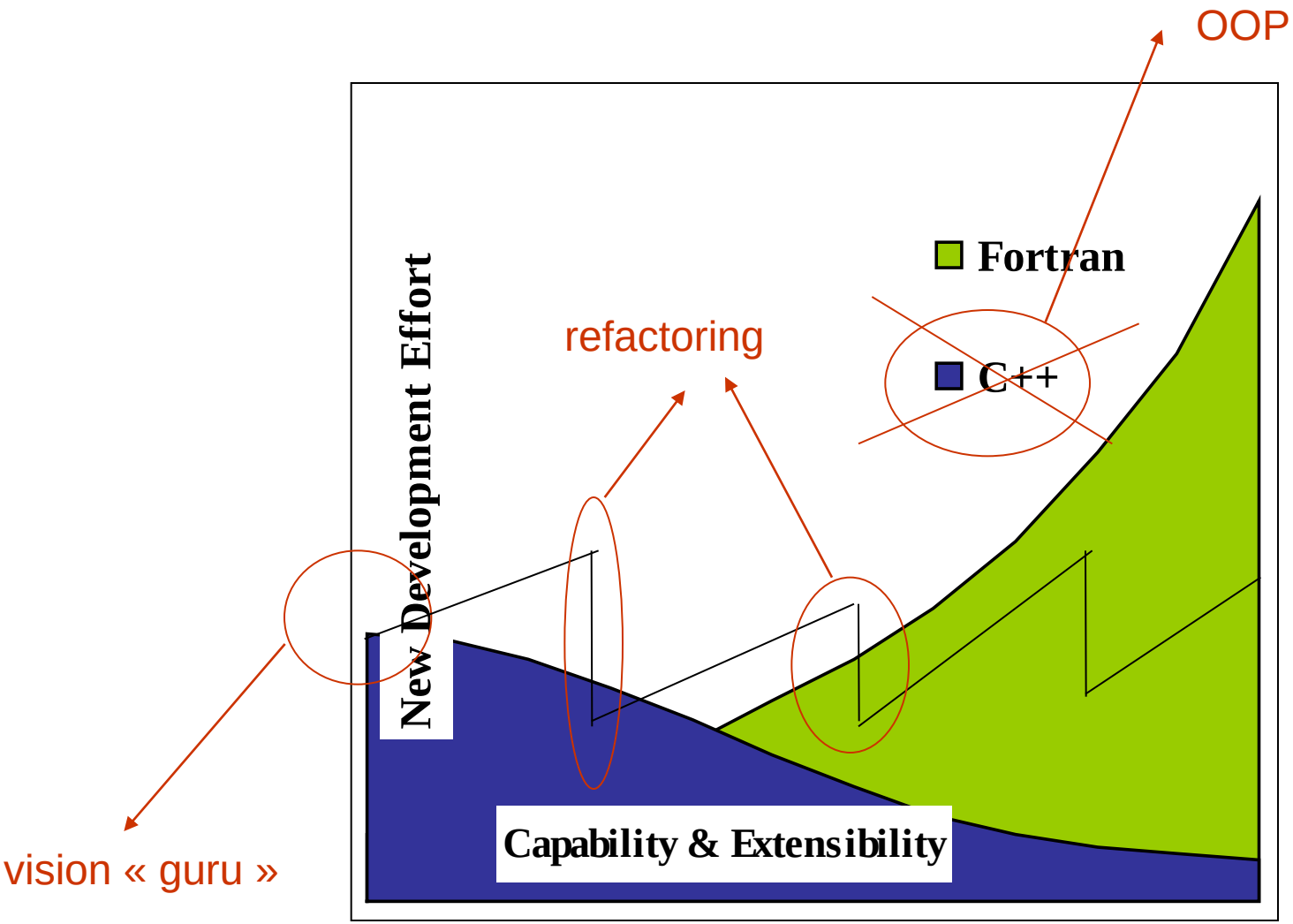
⟶ Inutile d'élaborer des structures complexes pour qq chose de simple qui n'évoluera peut-être pas (généralité spéculative).

Procédure répétitive et continue en 2 temps:

1. Modification de la structure du code pour accueillir la nouveauté **à fonctionnalité constante (refactoring)**
2. Ajout de la nouvelle fonctionnalité

Refactoring

Introduction – graphique revisité



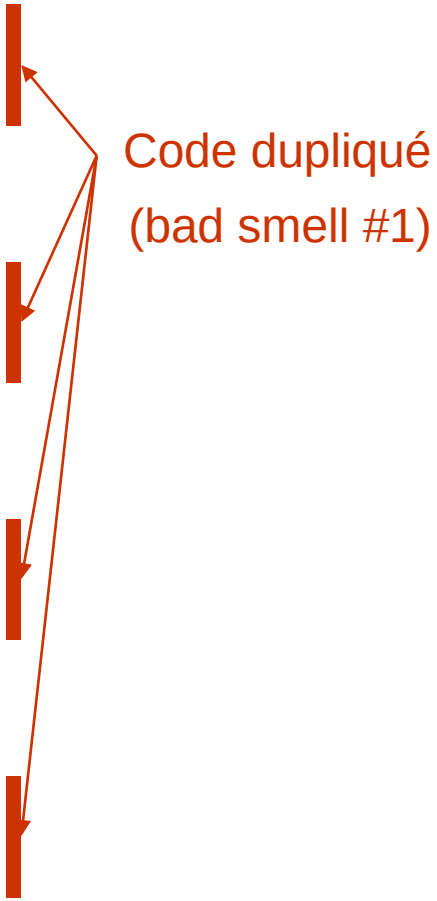
Refactoring

- Par l’exemple (simple)

Refactoring

exemple simple

```
double
Element::getEnergy (int which_one, Lock &L)
{

    State St; St[TO]=((Analysis *) get_properties (DOMAIN_PO)->get_pere ())->ref_val ();
    Propelem *prprties = (Propelem *) get_properties (PROPELEM_PO);
    double coef = prprties->exist(EXISTENCE) ? prprties->get(EXISTENCE,St) : 1;

    // ...
}

void
Element::fill_elemmatr (TypeMatStr & type, SymMatrix & k,Lock & type_ddl)
{
    State St; St[TO]=((Analysis *) get_properties (DOMAIN_PO)->get_pere ())->ref_val ();
    Propelem *prprties = (Propelem *) get_properties (PROPELEM_PO);
    double coef = prprties->exist(EXISTENCE) ? prprties->get(EXISTENCE,St) : 1;

    // ...
}

void
Element::fill_elemmatr (TypeMatStr & type, Matrix & k, Lock & type_ddl)
{
    State St; St[TO]=((Analysis *) get_properties (DOMAIN_PO)->get_pere ())->ref_val ();
    Propelem *prprties = (Propelem *) get_properties (PROPELEM_PO);
    double coef = prprties->exist(EXISTENCE) ? prprties->get(EXISTENCE,St) : 1;

    // ...
}

void
Element::fill_elemvect (int icoef, Vector & coef, Vector & type, Vector & v, Lock &type_ddl)
{
    State St; St[TO]=((Analysis *) get_properties (DOMAIN_PO)->get_pere ())->ref_val ();
    Propelem *prprties = (Propelem *) get_properties (PROPELEM_PO);
    double dcoef = prprties->exist(EXISTENCE) ? prprties->get(EXISTENCE,St) : 1;

    // ...
}
```



Refactoring

exemple simple

Quels sont les problèmes?

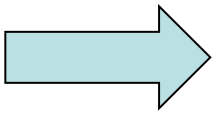
- Le code est dupliqué !
- Le code dupliqué n’est pas tout à fait identique !

Pourquoi l’auteur a-t-il fait ça?

- Pour gagner 10 minutes (« je te fais ça en 2 minutes »)
- Il faut introduire une fonction membre, modifier le .h et recompiler toutes les classes dérivées
- Egoïsme

Conséquences?

- Les différentes copies peuvent (VONT!) évoluer différemment ! (cfr. bug TO)
- Le code est alourdi et difficile à lire.



Des heures vont être perdues par les autres développeurs !

Refactoring

exemple simple – solution par le refactoring

bad smell : **Code dupliqué**

solution associée **: « Extract method »**

procédure:

1. création de la méthode
2. recherche des variables locales utilisées => paramètres
3. recherche des variables temporaires => redéfinir
4. Vérification de la compilation
5. Remplacement du code dupliqué
6. Compilation et test

notes:

- La recette peut paraître inutilement laborieuse dans ce cas simple.
- simple « recette » à suivre presque les yeux fermés

Refactoring

exemple simple

```
double
Element::getEnergy (int which_one, Lock &L)
{
    double coef = computeVanishingCoefficient();
    // ...
}

void
Element::fill_elemmatr (TypeMatStr & type, SymMatrix & k,Lock & type_ddl)
{
    double coef = computeVanishingCoefficient();
    // ...
}

void
Element::fill_elemmatr (TypeMatStr & type, Matrix & k, Lock & type_ddl)
{
    double coef = computeVanishingCoefficient();
    // ...
}

void
Element::fill_elemvect (int icoef, Vector & coef, Vector & type, Vector & v, Lock &type_ddl)
{
    double dcoef = computeVanishingCoefficient();
    // ...
}

double
Element::computeVanishingCoefficient()
{
    State St; St[TO]=((Analysis *) get_properties (DOMAIN_PO)->get_pere ())->ref_val ();
    Propelem *prprties = (Propelem *) get_properties (PROPELEM_PO);
    return (prprties->exist(EXISTENCE) ? prprties->get(EXISTENCE,St) : 1);
}
```

choix d'un nom qui rend tout commentaire superflu !

Refactoring

- Par l’exemple (moins simple)

Refactoring

exemple Oofelie - Application à ValidCodeList

Contexte

- Nouvelle gestion matériaux/éléments
- Classe instanciée **`ValidCodeList<MaterialProperty>`**
- Permet de garder en mémoire une liste de propriétés.
- Permet de vérifier si une propriété est définie pour un matériau donné

```
class MaterialProperty
{
    int myId;
    char *name;
    char *sentence;
    const char *myType_char;
    //...
};
```

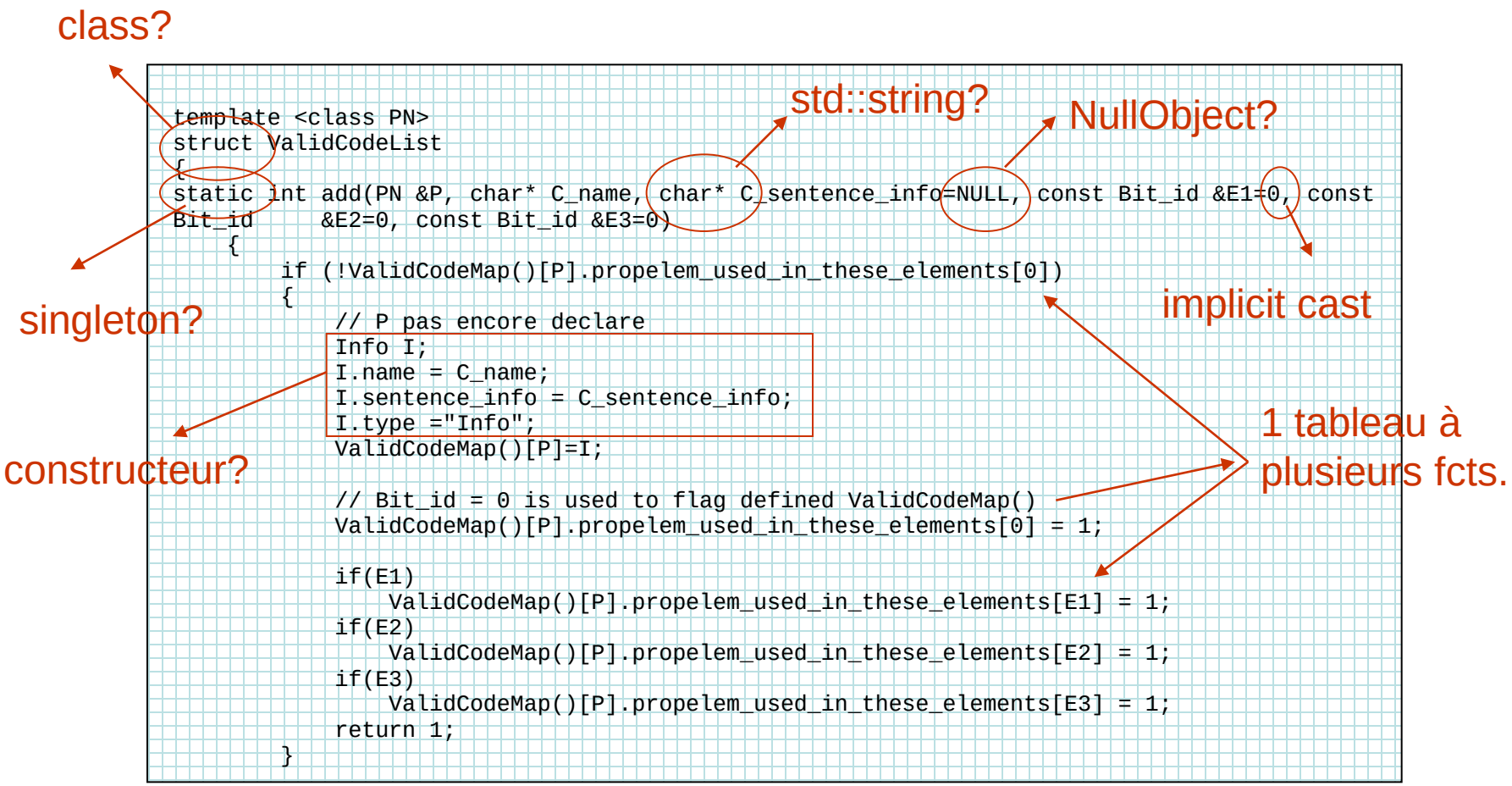
Exemple plus complexe pour

- prouver que identifier des « bad smells » != comprendre le code
- insister sur l'intérêt du refactoring
- montrer la démarche réelle de développement de OE

(qui est la bonne! – il ne manque que l'étape de refactoring)

Refactoring

exemple Oofelie




```
template <class PN>
struct ValidCodeList
{
static int add(PN &P, char* C_name, char* C_sentence_info=NULL, const Bit_id &E1=0, const
Bit_id     &E2=0, const Bit_id &E3=0)
    {
        if (!ValidCodeMap()[P].propelem_used_in_these_elements[0])
        {
            // P pas encore declare
            Info I;
            I.name = C_name;
            I.sentence_info = C_sentence_info;
            I.type ="Info";
            ValidCodeMap()[P]=I;

            // Bit_id = 0 is used to flag defined ValidCodeMap()
            ValidCodeMap()[P].propelem_used_in_these_elements[0] = 1;

            if(E1)
                ValidCodeMap()[P].propelem_used_in_these_elements[E1] = 1;
            if(E2)
                ValidCodeMap()[P].propelem_used_in_these_elements[E2] = 1;
            if(E3)
                ValidCodeMap()[P].propelem_used_in_these_elements[E3] = 1;
            return 1;
        }
```

implicit cast (en int puis bool) ! P est une classe !

```
else
    if(P)
        // Ce P est deja declare
        {
            if(C_name)
            {
                char *h = ValidCodeMap()[P].name;
                if(!h)
                {
                    ValidCodeMap()[P].name = C_name;
                    ValidCodeMap()[P].sentence_info = C_sentence_info;
                }
                else
                    if(strcmp(C_name, h))
                    {
                        // Les noms sont differents => on change la valeur de P
                        P = ValidCodeMap().size();
                        ValidCodeMap()[P].name = C_name;
                        ValidCodeMap()[P].sentence_info = C_sentence_info;
                        //ValidCodeMap()[P].type ="Type undefined";
                    }
                    ValidCodeMap()[P].propelem_used_in_these_elements[0] = 1;
                    if(E1)
                        ValidCodeMap()[P].propelem_used_in_these_elements[E1] = 1;
                    if(E2)
                        ValidCodeMap()[P].propelem_used_in_these_elements[E2] = 1;
                    if(E3)
                        ValidCodeMap()[P].propelem_used_in_these_elements[E3] = 1;
                    return 1;
            }
            else
                cerr << "** Warning: Prop.h: P("<<P<<") not defined !"<<endl;
        }
        else // P =0
```

rules !

cast implicite
P(int)

code dupliqué

mauvaise indentation => illisible

```
}
else // P =0
    if(C_name)
     if(strcmp(ValidCodeMap()[P].name ,C_name))
{
    PN PfromName = fromName(C_name);
    if(PfromName)
        P = PfromName;
    else
    {
        P = ValidCodeMap().size();
      if(!P) P = ValidCodeMap().size(); // to avoid declaration on 0
        ValidCodeMap()[P].name = C_name;
    }
    if (!ValidCodeMap()[P].sentence_info)
        ValidCodeMap()[P].sentence_info = C_sentence_info;
    ValidCodeMap()[P].propelem_used_in_these_elements[0] = 1;
    if(E1)
        ValidCodeMap()[P].propelem_used_in_these_elements[E1] = 1;
    if(E2)
        ValidCodeMap()[P].propelem_used_in_these_elements[E2] = 1;
    if(E3)
        ValidCodeMap()[P].propelem_used_in_these_elements[E3] = 1;
    return 1;
}
return 0;
}
```

commentaire « guru »

?

code dupliqué (3x)
et non finalisé!

« écrivez votre idée sur papier en UML qu'ils disaient ... »

Refactoring

exemple Oofelie - Application à ValidCodeList

Idée générale dans ce cas:

Rendre la structure lisible par un débutant.

Procédure incrémentale :

- La structure va s'éclaircir d'elle même au cours des itérations.

Procédure systématique :

- Le gros du travail peut être réalisé par un débutant en C++ !

Attention :

•Dans ce qui suit, nous faisons plusieurs opérations à la fois (le travail est trop gros pour être détaillé étape par étape).

•Chaque modification du code entraîne compilation (au minimum) et tests!

Refactoring

exemple Oofelie - Application à ValidCodeList

Début :

Bit_id = 0	Bad smell #6	: changements chirurgicaux
Char *	Bad smell #9	: obsession des types primitifs
Struct	Bad smell #17	: intimité inappropriée

+ rules

```
template <class PN>
struct ValidCodeList
{
static int add(PN &P, char* C_name, char* C_sentence_info=NULL, const Bit_id &E1=0, const
Bit_id     &E2=0, const Bit_id &E3=0)
```

```
template <class PN>
class ValidCodeList
{
public:
        static int add(PN &P,
                       std::string &C_name,
                       std::string &C_sentence_info = myNullString,
                       const Bit_id &E1 = NOID,
                       const Bit_id &E2 = NOID,
                       const Bit_id &E3 = NOID)
```

Refactoring

exemple Oofelie - Application à ValidCodeList

```
Info I;
I.name = C_name;
I.sentence_info = C_sentence_info;
I.type ="Info";
ValidCodeMap()[P]=I;
```

Bad smell #17 : intimité inappropriée
Bad smell #20 : classes « données »

Le code est déplacé dans
le (nouveau) constructeur de Info

```
ValidCodeMap()[P] = Info(C_name,C_sentence_info,"Info");
```

```
Info::Info(const std::string &C_name,
           const std::string &C_sentence_info,
           const std::string &C_type) : name(C_name),
                                        sentence_info(C_sentence_info),
                                        type(C_type)
{
}
```

Refactoring

Bad smell #1 : code dupliqué
Bad smell #22 : commentaires

```
// Bit_id = 0 is used to flag defined ValidCodeMap()
ValidCodeMap()[P].propelem_used_in_these_elements[0] = 1;

if(E1)
    ValidCodeMap()[P].propelem_used_in_these_elements[E1] = 1;
if(E2)
    ValidCodeMap()[P].propelem_used_in_these_elements[E2] = 1;
if(E3)
    ValidCodeMap()[P].propelem_used_in_these_elements[E3] = 1;
```

```
ValidCodeMap[P].setUsedForTheseElements(E1,E2,E3);
```

```
Info::setUsedForTheseElements(const Bit_id &E1, const Bit_id &E2, const Bit_id &E3)
{
        // Bit_id = 0 is used to flag defined ValidCodeMap()
        propelem_used_in_these_elements[0] = 1;

        if(E1)
                propelem_used_in_these_elements[E1] = 1;
        if(E2)
                propelem_used_in_these_elements[E2] = 1;
        if(E3)
                propelem_used_in_these_elements[E3] = 1;
}
```

variable séparée

enum comme index!

Refactoring

exemple Oofelie - Application à ValidCodeList

Corps de la fonction Bad smell #2 : fonction longue

```
template <class PN>
class ValidCodeList
{
public:
static int add( ... )
{
        if (ValidCodeMap()[P].isNotUsed())
        {
            ValidCodeMap()[P] = Info(C_name,C_sentence,"Info");
            ValidCodeMap()[P].setUsedForTheseElements(E1,E2,E3);
            return 1;
        }
        else
            if(P.isAValidProperty() )
            {
                fillMissingInfoForThisProperty(P, C_name, C_sentence_info, E1, E2, E3);
            }
            else
                tryToSetPropertyUsingNameString(P, C_name, C_sentence_info, E1, E2, E3);

      return 0;
}
```

et ainsi de suite...

Refactoring

- Liste des « bad smells » avec justification
- Application à Oofelie et Metafor

Refactoring

Bad Smells – comment identifier du code pourri?

Type	Description	exemples dans Oofelie
Code dupliqué	Il faut absolument lutter contre le code dupliqué. On arrive inévitablement à une situation où une partie du code dupliqué évolue et les copies non. Faire du couper coller devrait donc être interdit	Partie mécanique dans meta_qs.e et metat_iso.e. Les éléments thermiques, thermique second degré.
Fonctions longues	Les fonctions longues sont typiques des langages de programmation structurés tels que le C ou le Fortran. Un programme orienté objet et une grande série de petites fonctions.	Très courant dans Oofelie. Le problème est généralisé. Voir par exemple les algorithmes (meta_qs), la création de la topologie, les tests de l'élément de contact, le formalisme ALE, etc, etc.
Grandes classes	Tout comme les fonctions, les classes doivent avoir une action limitée pour qu'on puisse facilement cerner les éventuels problèmes	L'élément, l'ALE, Face et Volume (qui contiennent la description géométrique, les mailleurs et les test d'appartenance, les projections, etc etc)
Longues listes de paramètres	C'est à éviter parce que ce n'est pas lisible et c'est la source de nombreuses erreurs. C'est souvent le signe qu'une classe fait le boulot d'une autre qui n'existe pas encore.	Routines de projection dans la géométrie
Changements divergents	Lorsque ajouter deux fonctionnalités différentes revient à modifier la même classe	La classe Metafor, où tout est mélangé.
Changements chirurgicaux	Lorsque ajouter une fonctionnalité nécessite de nombreux petits changements partout	Tozmesh, Tomatlab, les smooth_fields
Envie de fonctionnalités	Lorsqu'une classe s'intéresse plus à manipuler les données d'une autre que les siennes	la classe Metafor
Ensemble de données	Ce sont des données qui sont ensembles dans une classe parmi d'autres mais qui sont liées entre elles. Il faut généralement les lier plus fortement via une classe.	Classe Metafor (ou certaines données concernant le fac cotoient les mde et des paramètres de visualisation), VizWin, AleMethod, l'élément, etc.

Type	Description	exemples dans Oofelie
Obsession des types primitifs	Utilisation abusive des types primitifs (int, double) au lieu de petites classes spécifiques. Ces petites classes permettent de bénéficier des vérifications de type du compilateur et sont plus claires à la lecture du code.	mde, mdr, smooth_type, ... int * au lieu de std::vector<int>
Switch	L'utilisation des switch est typique d'une programmation structurée. Les switch doivent être remplacés par du polymorphisme lorsque c'est possible. Cela permet de ne pas devoir modifier tous les switchs lorsqu'on ajoute une fonctionalité	L'interpréteur d'Oofelie (id.cpp)
Hiérarchie de classes parallèles	Une modification dans une classe entraine des modifiactions dans d'autres. Il vaut mieux donc faire deux hiérachies séparées qui se référencient l'une l'autre	Ajouter le pilotage en force nécessite de modifier tous les éléments de contact.
Classe paresseuse	C'est une classe qui ne fait presque plus rien	+/- OK
Généralité spéculative	C'est une classe ou une structure qui est beaucoup trop générale pour ce qui existe à l'heure actuelle dans le code. Ca embrouille inutilement le code et la majorité de la structure est vide et donc inutile.	Les matériaux d'Igor
Variable membre temporaire	Ce sont des variables membres qui sont parfois utilisées, parfois pas. Ce n'est pas clair du tout pour le lecteur du code	
Chaines de messages	Lorsqu'un objet demande un accès à un autre, puis un autre pour avoir accès à une information. Dès qu'on change un objet intermédiaire, tous les appels doivent être changés.	Dans la géométrie, on accède à beaucoup d'objets de cette manière.
Objet intermédiaire	C'est une classe qui ne fait que transmettre des ordres à une autre. Elle peut donc être souvent supprimée.	+/- OK
Intimité inappropriée	Lorsque deux classes sont trop visibles l'une de l'autre. Il faut absolument privatiser toutes les variables membres d'une classe. Cela permet de modifier très facilement la structure interne d'une classe sans toucher à son interface.	Toutes les classes de Metafor
Classes similaires avec interfaces différentes	Les interfaces de deux classes faisant la même chose sont différentes. Il faut les unifier.	+/- OK

Type	Description	exemples dans Oofelie
Bibliothèques incomplètes	Lorsque les classes des bibliothèques utilisées sont incomplètes, il existe des moyens d'ajouter les fonctionnalités voulues	La bibliothèque Oofelie (Physet, Element, Material, etc)
Classes "données"	Les classes de données ont tendance à devenir très vite des "structures C" où tout est public. Ces classes ont besoin d'un refactoring pour être orientées objet.	Les points de Gauss GPState, certaines structures de pointeurs dans l'élément et les matériaux.
Interface refusée	Une classe dérivée n'utilise pas les fonctions dont elle hérite. Il se peut donc qu'elle ne soit pas à sa place ou que la hierachie de classes soit mauvaise.	+/- OK
Commentaires	Trop de commentaires est mauvais: il faut que le code puisse se lire sans commentaires en utilisant des fonctions courtes et des noms de fonctions et de variables très explicites. Bien sûr le pire est de ne pas avoir de commentaires sur du code pourri.	Tout le code

Refactoring

- Liste des « recettes »
- Application à Oofelie et Metafor

Refactoring

méthodes

- Méthodes de composition
- Déplacement de fonctionnalités entre objets
- Organisation des données
- Simplification d'expressions conditionnelles
- Simplification des appels de fonctions
- Gestion de la généralisation

Refactoring

Méthodes de composition

Type	Description	Oofelie
Extraction de méthode	Méthode de base: faire une fonction membre d'une suite de lignes de code qui font une action spécifique. Il faut veiller à bien nommer la fonction résulante et faire très attention aux variables locales.	Doit être effectué sur tout le code pour supprimer une bonne partie du code dupliqué.
Fonction inline	L'inverse de "Extraction de méthode"	Sans objet pour l'instant.
Temp inline	Suppression des variables temporaires. Ces variables empêchent très souvent de faire d'autres étapes de refactoring.	Peut se faire sans problème dans les routines où on ne fait pas de calculs (où les performances ne seront pas dégradées)
Remplacer un temp avec une demande	Au lieu d'utiliser une variable temp, on utilise une fonction qui renvoie la valeur voulue.	Certaines optimisation dans la fonction doivent être faites pour éviter de recalculer tout le temps la même chose (comme nous l'avons fait dans les shortcuts de la géométrie).
Introduction d'une variable d'explication	Nommer des variables temporaires qui regroupent (par exemple) des opérations booléennes avant un if. Il faut alors donner un nom très explicite à la variable temp.	A faire partout
Division de variables temporaires	Une variables temporaire, déclarée au début de la routine est utilisée plusieurs fois pour des raisons différentes.	C'était une pratique très courante en Fortran mais beaucoup moins en C++ si on applique la règle de déclarer ses variables le plus tard possible.
Suppression des modifications de paramètres	Les paramètres reçus par une fonction ne doivent pas être modifiés. cela entraîne rapidement des erreurs. Il faut donc d'abord assigner la valeur du paramètre à une variable temporaire et modifier cette dernière.	+/- OK
Remplacer une méthode par un "objet méthode"	Extraire une fonction longue qui possède beaucoup de paramètres dans un objet et créer un objet si on veut utiliser la méthode.	les mailleurs, les projections, les tests d'appartenance, etc.
Substitution d'algorithme	Remplacer un algorithme compliqué par un simple	utilisation des std:: et algorithmes associés (find, sort, ...).

Refactoring

Méthodes de déplacement de fonctionnalités entre objets

Type	Description	Oofelie
Déplacement de méthode	Déplacer une méthode d'une classe vers une autre	A utiliser lors de la création de nouvelles classes
Déplacement de variable	Déplacer une variable membre d'une classe vers une autre	A utiliser lors de la création de nouvelles classes
Extraction de classes	Lorsque une classe fait deux choses, on peut diviser la classe en deux classes (en les liant éventuellement l'une à l'autre via des références).	classes Metafor, AleMethod, certains objets géométriques
Inlining d'une classe	On inline les fonctions de la classe manuellement parce que la classe ne fait pas grand chose.	Pas utile aujourd'hui
Cacher le délégué	Ajouter des fonctions pour cacher les objets intermédiaires nécessaires pour obtenir une info.	Pour l'instant, rien est caché, gros boulot.
Suppression d'une classe intermédiaire	On supprime une classe qui ne fait qu'en appeler une autre.	Pas utile. On a plutôt trop peu de classes
Introduction d'une méthode étrangère	Ajout d'une fonctionalité à une classe dont on a pas accès au source en définissant une fonction statique recevant l'objet à étendre.	Utilisé déjà dans AleMethod par exemple pour Elemset qu'on ne peut pas modifier
Introduction d'extension locale	Création d'une sous classe ou d'un wrapper vers une classe dont on veut étendre les fonctions et dont on n'a pas le source.	Par exemple, la classe FastPhyset

Refactoring

Méthodes d’organisation de données

Type	Description	Oofelie
Encapsulation des données privées	Ajouter les set et get pour accéder en variables membres privées.	On n'en est pas encore la: la plupart des variables sont publiques
Remplacement de données par des objets	Création de petits objets caractérisant des données	Ca doit être fait dans les grandes classes
Changement de valeurs par références	Utilise des références au lieu de copies (gros objets ou objets modifiables)	On a plutôt tendance à mettre des références partout même lorsqu'il ne faut pas
Changement de références par des valeurs	Utilise des copies au lieu de références (petits objets). Permet une meilleure encapsulation des données.	C'est une étape indispensable pour encapsuler les données correctement.
Remplacement d'un tableau par un objet	Remplace les tableaux. Cela permet de cacher les allocations et la gestion du tableau dans une classe.	remplacer les tableaux, listes par des std::
Duplication de données observées	Duplique certaines données pour permettre de séparer des classes qui ne peuvent pas se connaître. La synchronisation peut se faire par le design pattern : observer	BWin devrait utiliser ça.
Changement d'association unidirect. en bidirect.	Permet que deux classes se connaissent l'une l'autre en utilisant deux références et un mécanisme de communication entre les classes.	Je pense qu'on a plutot trop de liens que trop peu
Changement d'association bidirect. en unidirect.	Supprime les liens inutiles entre les classes.	Utile lorsque trop de liens sont présents et sont difficilement mis à jour

Type	Description	Oofelie
Remplacer les nombre magiques par des constantes	Remplacer les paramètres (pi, sqrt(3/2), ...) par des constantes statiques.	Remplacer les define par des const
Encapsulation de variables	Rendre une variable membre privée et ajouter les set et get pour accéder en variables membres.	Très important.
Encapsulation de collections	Rend les containers privés et ajoute des add(), remove() pour cacher l'implémentation.	Un pas de plus vers du vrai orienté objet
Remplacer des "records" par une classe de données	Remplace les structures ou groupes de données par des classes.	C'est très important (GPState, Pr_struct, etc). Ces structures, initialement petites sont devenues de véritables objets.
Remplace du code de type par une classe	Remplace des valeurs numériques ou enums par de vraies objets. Cela permet de bénéficier de tous les avantages des objets et de la vérification de types.	Réflechir pour Mde, Mdr, etc.
Remplace du code de type par une sous classe	Idem en utilisant une hiérachie de classes.	Moins utile dans notre cas.
Remplace du code de type par un state / strategy	Utilise une hiérachie de classe indépendante et assigne à l'objet un objet de cette hiérachie qui caractérise son type. L'avantage de cette méthode est que le type peut changer au cours du temps.	
Remplacer une sous classe par une variable "type"	Supprime les sous classes inutiles en ajoutant un type à la classe de base (utile si les sous classes ne font presque rien ou retournent des constantes)	

Refactoring

Méthodes de simplification d’expressions conditionnelles

Type	Description	Oofelie
Décomposition de conditions	Utilise l'extraction de méthodes pour rendre les actions soumises à conditions plus lisibles. Le but est de séparer la logique des actions.	Travail global sur le code
Consolidation des conditions	Regroupe les conditions qui peuvent l'être (des if successifs, etc).	
Consolidation du code dupliqué	On sort le code dupliqué (dans le if et le else par exemple) de la condition.	
Suppression de flag de contrôle	Utilisation des break et continue au lieu de if(sortir)	
Remplacement des conditions imbriquées par des conditions successives	Décompose les conditions et remplace les conditions imbriquées complexes en une série de conditions successives.	
Remplacement des conditions par le polymorphisme	Crée une hiérarchie de classes pour profiter des fonctiuons virtuelles et supprimer les if	
Introduction d'objets nuls	Remplace des actions conditionnelles complexes par l'exécution d'action inconditionnelle sur une liste comportant des objets nuls.	
Intraduction d'assertion	Utilisation d'assert lorsqu'une hypothèse est faite.	

Refactoring

Méthodes de simplification des appels de fonctions

Type	Description	Oofelie
Renommer les fonctions	Donne un nom plus explicite à une méthode	
Ajoute un paramètre	Ajoute un paramètre dont la fonction à besoin	
Supprime un paramètre	Supprime un paramètre dont la fonction n'a plus besoin.	
Sépare demande et modification	Veille à ce qu'une fonction qui retourne un paramètre ne modifie pas l'objet et inversément (la fonction est soit de type Query, soit Modifier)	
Paramétrisation d'une méthode	Fusionne plusieurs méthodes qui font la même chose à peut de chose près en ajoutant un ou des paramètres	
Remplacer un paramètre par des fonctions explicites	Divise une méthode qui fait des actions différentes en fonction de la valeur d'un paramètre	
Préservation de l'objet entier	Passer l'objet entier à une méthode au lieu de lui passer des morceaux.	
Remplacer un paramètre par une méthode	Il ne faut pas passer de paramètres à une fonction si celle-ci peut les récupérer autrement.	
Introduction d'un "objet paramètre"	Groupe de nombreux paramètres en un seul objet avant l'appel de la fonction	
Suppression de la methode "set"	Il faut définir une "setting method" uniquement pour les valeurs qui vont être modifiées. Il faut donc supprimer celles relatives à des constantes	
Cacher une méthode	Rendre privée les méthodes non utilisées par les autres classes	
Remplacer le constructeur par un "Factory method"	Permet de créer des objets complexes (sous classes) dans la classe de base.	
Encapsulation des Downcasts	Il faut que les fonctions retournent des types les plus spécialisés possibles.	
Remplacer les codes d'erreur par des exceptions	Le nom est explicite. Ca permet de séparer le code qui gère l'erreur du code d'exécution normal.	
Remplacer les exceptions par des tests	Permet de supprimer les exceptions qui ne sont pas des événements exceptionels	

Refactoring

Méthodes de gestion de la généralisation

Type	Description	Oofelie
Remonter / Descendre une variable	Lorsqu'une variable est déclarée dans toutes les sous classes d'une classe, on peut déplacer celle-ci dans la classe mère	
Remonter / Descendre une fonction membre	Idem	
Remonter / Descendre le constructeur	Idem	
Extraction d'une sous classe	Utilisé si certaines instances de la classe n'utilisent pas toutes les fonctionnalités de la classe.	
Extraction d'une classe mère	Si des classes ont des fonctionnalités communes, on crée une classe mère de celles-ci	
Extraction d'une interface	Crée une interface pour un ensemble de classes	
Fusion de hiérarchie	Si une classe et une sous classe sont plus ou moins identiques, on peut les fusionner	
Création d'une méthode template	Lorsque deux sous classes ont une méthode similaire qui suit un même schéma, il est intéressant de la programmer dans la classe de base et appeler des fonctions virtuelles pour les parties différentes	
Remplacer l'héritage par de la délégation	Si une classe n'utilise pas toute l'interface de sa classe mère, il peut être utile de supprimer l'héritage et d'encapsuler la classe mère dans l'objet.	

Conclusions

Conclusions

- C++ n’implique pas l’orienté objet
- Ecrire un code orienté objet est incrémental
- Utiliser au maximum la STL
- Ecrire un programme en commun (2 auteurs ou +) => refactoring

Un bon programme C++ est un programme lisible par un débutant