Programmation Orientée Objet : Introduction à la POO

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs de la leçon d'aujourd'hui

- Concepts fondamentaux
- Étude de cas

Organisation du travail (semestre)

		MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	22.02.24		0	Intro + compil. séparée	
2	29.02.24	1. Intro POO	0	Intro POO	
3	07.03.24	2. Constructeurs/Des	0	Constructeurs	
4	14.03.24	3. Surcharge des opé	0	Surcharge	
5	21.03.24	4. Héritage	0	Héritage	
6	28.03.24	5. Polymorphisme	0	Polymorphisme 1	
-	11.04.24		-	vacances Pâques	
7	04.04.24		1	Polymorphisme 2 / Collections hétérogènes	
8	18.04.24		-	-	Série notée
9	25.04.24	6. Héritage multiple	2	Héritage multiple	
10	02.05.24	(7. Etude de cas)	-	Templates	
12	16.05.24		-	(Ascension)	
11	09.05.24		-	Structure de données abstraites ; Bibliothèques	
13	23.05.24	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions	
14	30.05.24		-	-	Examen

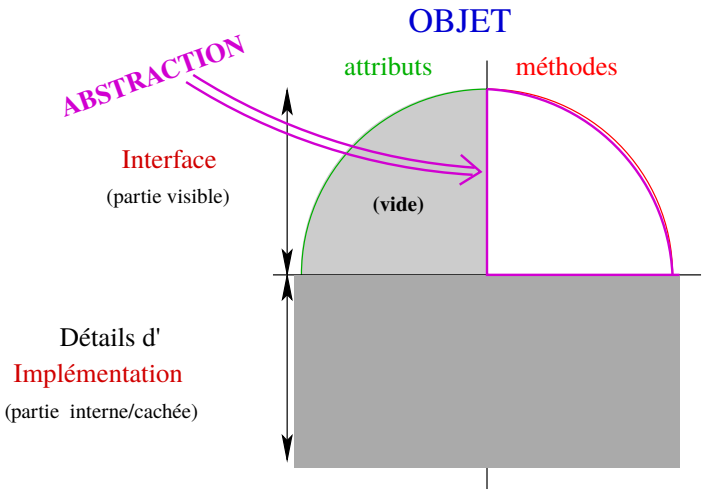
Concepts fondamentaux

- Encapsulation :
  regrouper données et traitements d'un même « concept »
- Abstraction :
  se focaliser sur ce qui est caractéristique de ce « concept »
  ☞ interface
  cacher les détails

Jean-Cédric Chappelier
& Jamila Sam

Encapsulation / Abstraction : Résumé

MIEUX :



Encapsulation / Abstraction : vues en C++

```
class Concept {
public:
  methodes importantes;

private:
  attributs;

  methodes secondaires;
};

Concept une_instance;
```


Jean-Cédric Chappelier
& Jamila Sam

Etude de cas

Comment représenter des nombres complexes ?

Représenter des nombres complexes ?

Premières idées (non POO) :

1. `typedef array<double, 2> Complexe;`
2. `struct Complexe { double x; double y; };`

et l'on déclarerait par exemple : `Complexe z;`

Pour l'affecter, avec le premier on écrirait :

```
z[0] = 1.0; z[1] = 2.0;
```

et avec le second :

```
z.x = 1.0; z.y = 2.0;
```

☞ Laquelle vous semble la plus claire/la plus parlante ?

Jean-Cédric Chappelier
& Jamila Sam

Représenter des nombres complexes ?

Mais pourquoi avoir défini les nombres complexes comme

```
struct Complexe { double x; double y; };
```

et non pas comme

```
struct Complexe { double rho; double theta; };
```

Qui « a raison » ? Laquelle est la meilleure ?

☞ Aucune, elles sont toutes les deux aussi **mauvaises** !

Jean-Cédric Chappelier
& Jamila Sam

Découplage des codes

Elles sont les deux mauvaises car le code utilisateur
(par exemple `z.x = 1.0`)
est directement dépendant du choix d'implémentation :

si l'on change la *représentation interne* des `Complexe`s
on est obligé de changer **tout** le code qui l'utilise :-(

☞ le *couplage* entre le code « producteur » et le code « utilisateur »
est *trop fort* !

Découpler ces codes, réduire les dépendances est la raison profonde
des principes d'encapsulation et d'abstraction en POO.

Représentation POO des nombres complexes

La première question à se poser est :
quelles sont les caractéristiques attendues des nombres complexes ?

- partie réelle (en lecture, en écriture ?), partie imaginaire
- module, argument
- conjugué
- addition
- ...

☞ pour garantir le *découplage* de code dont on parlait précédemment, **toutes** ces caractéristiques doivent être des *méthodes* (et ceci *indépendemment* du choix d'implémentation, c.-à-d. indépendemment des attributs choisis)

Jean-Cédric Chappelier
& Jamila Sam

Un exemple possible (1/3)

```
#include <iostream>
#include <cmath>  // pour abs, sqrt, atan, cos, sin et M_PI
using namespace std;

class Complexe {
public:

  // accesseurs
  double x()     const { return x_; }
  double y()     const { return y_; }
  double rho()   const { return sqrt(x_ * x_ + y_ * y_); }
  double theta() const {
    double const module(rho());
    double const precision(1e-15);
    if (abs(module) < precision)
      { return 0.0; }
    else if ((abs(y_) < precision) and (x_ < 0.0))
      { return M_PI; }
    else
      { return 2.0 * atan(y_ / (x_ + module )); }
  }
```

Un exemple possible (2/3)

```
// manipulateurs
void cartesiennes(double abscisse, double ordonnee)
  { x_ = abscisse ;  y_ = ordonnee ; }
void polaires(double module, double argument)
{ x_ = module * cos(argument) ;
  y_ = module * sin(argument) ;
}
void set_x(double abscisse)          //
  { cartesiennes(abscisse, y()); }  // Ces quatre là sont
void set_y(double ordonnee)          // TRÈS discutables !
  { cartesiennes(x(), ordonnee); }  //
void set_rho(double module)          // (et sont discutés
  { polaires(module, theta()); }    //  en cours)
void set_theta(double argument)      //
  { polaires(rho(), argument); }    //

// autres opérations
Complexe conjugue() const {
  // sera plus simple à écrire quand nous aurons les constructeurs
  Complexe c;
  c.cartesiennes(x_, -y_);
  return c;
}
```

Un exemple possible (3/3)

```
private:  // un choix d'implémentation parmi d'autres
  double x_;
  double y_;
};

// ================================================================
int main()
{
  // exemple d'utilisation
  Complexe a;
  a.cartesiennes( 1.0, 2.0);

  cout << a.x() << "+" << a.y() << "i = "
       << a.rho() << "e^(i*" << a.theta() << ")"
       << endl;

  return 0;
}
```


Jean-Cédric Chappelier
& Jamila Sam

Modularisation de l'exemple : `.h`

```
class Complexe {
public:
  // accesseurs
  double x()     const { return x_; }
  double y()     const { return y_; }
  double rho()   const;
  double theta() const;

  // manipulateurs
  void cartesiennes(double abscisse, double ordonnee);
  void polaires(double module, double argument);
  // ...

  // autres opérations
  Complexe conjugue() const;

private:
  // un choix d'implémentation parmi d'autres
  double x_;
  double y_;
};
```


Jean-Cédric Chappelier
& Jamila Sam

Modularisation de l'exemple : .cc

```
#include <cmath> // pour abs, sqrt, atan, cos, sin et M_PI
#include "complexe.h"
using namespace std;

double Complexe::rho() const { return sqrt(x_ * x_ + y_ * y_); }

double Complexe::theta() const {
    double const module(rho());
    double const precision(1e-15);
    if (abs(module) < precision)
      { return 0.0; }
    else if ((abs(y_) < precision) and (x_ < 0.0))
      { return M_PI; }
    else
      { return 2.0 * atan(y_ / (x_ + module )); }
  }

void Complexe::cartesiennes(double abscisse, double ordonnee)
{ x_ = abscisse ;  y_ = ordonnee ; }

void Complexe::polaires(double module, double argument)
{ x_ = module * cos(argument) ;
  y_ = module * sin(argument) ; }
```