Introduction aux méthodes Orientées Objets

Première partie

Modélisation avec UML 2.0

Programmation orientée objet en C++

Pré-requis:

maitrise des bases algorithmiques (cf. 1ier cycle),
maitrise du C (variables, fonctions, pointeurs, structures)

Thomas Grenier

Bibliographie

- ***https://cpp.developpez.com/livres/***

- ***Le langage C++***, Bjarne Stroustrup, PEARSON (France), 2003
- ***Introduction à UML 2.0***, Miles & Hamilton, O'Reilly, 2006
- ***UML 2.0 en concentré***, Pilone & Pitman, O'Reilly, 2006

- Wikipedia
  - http://fr.wikipedia.org/wiki/Programmation_orientée_objet
  - http://fr.wikipedia.org/wiki/Unified_Modeling_Language
  - http://fr.wikipedia.org/wiki/Chronologie_des_langages_de_programmation
- Web divers
  - http://www.langpop.com/ (language popularity)
  - http://www.tiobe.com/ (language index)
  - http://www.nawouak.net/?cat=informatics+lang=fr (informatique et C++)
  - **https://en.cppreference.com/w/ (reference du C++)**

- Logiciels:
  - UMLDesigner: http://www.umldesigner.org
  - Diagrams.net https://app.diagrams.net/ (https://github.com/jgraph/drawio-desktop)
  - IDE C++ : **QtCreator**, Microsoft Visual Studio Code, Code::Blocks, Eclipse, …

Introduction

- Illustration du concept objet sur un exemple issu des mathématiques

- Brève histoire de la POO
  Origine du C++
- Modélisation Orienté Objet
  Présentation de UML

Socrative 846835

Introduction, concept d'objet

mathématique | *implémentation*

- Les nombres complexes

soit $z_1, z_2 \in \mathbb{C}$

$z_i = a_i + jb_i$

2 données regroupées

Par exemple : $z_1 = 1 + j$

$z_2 = 2 - 4j$

z_1 et z_2 sont indépendants l'un de l'autre

z_1 et z_2 appartiennent au même ensemble (complexe)

L'intérêt des complexes : c'est un corps commutatif (+,x)

$z_3 = z_1 + z_2 = 3 - 3j$

$z_4 = z_1 \times z_2 = 6 - 2j$

➔ Association d'opérations propres aux données

en OO:

- z_1, z_2, z_3 et z_4 sont des **objets**
- $\mathbb{C}$ est une **classe**

Structures du C

Méthode Orientée objet

Introduction POO

- La Programmation Orientée Objet (POO): origine de la démarche « *Orienté Objet* » (OO)

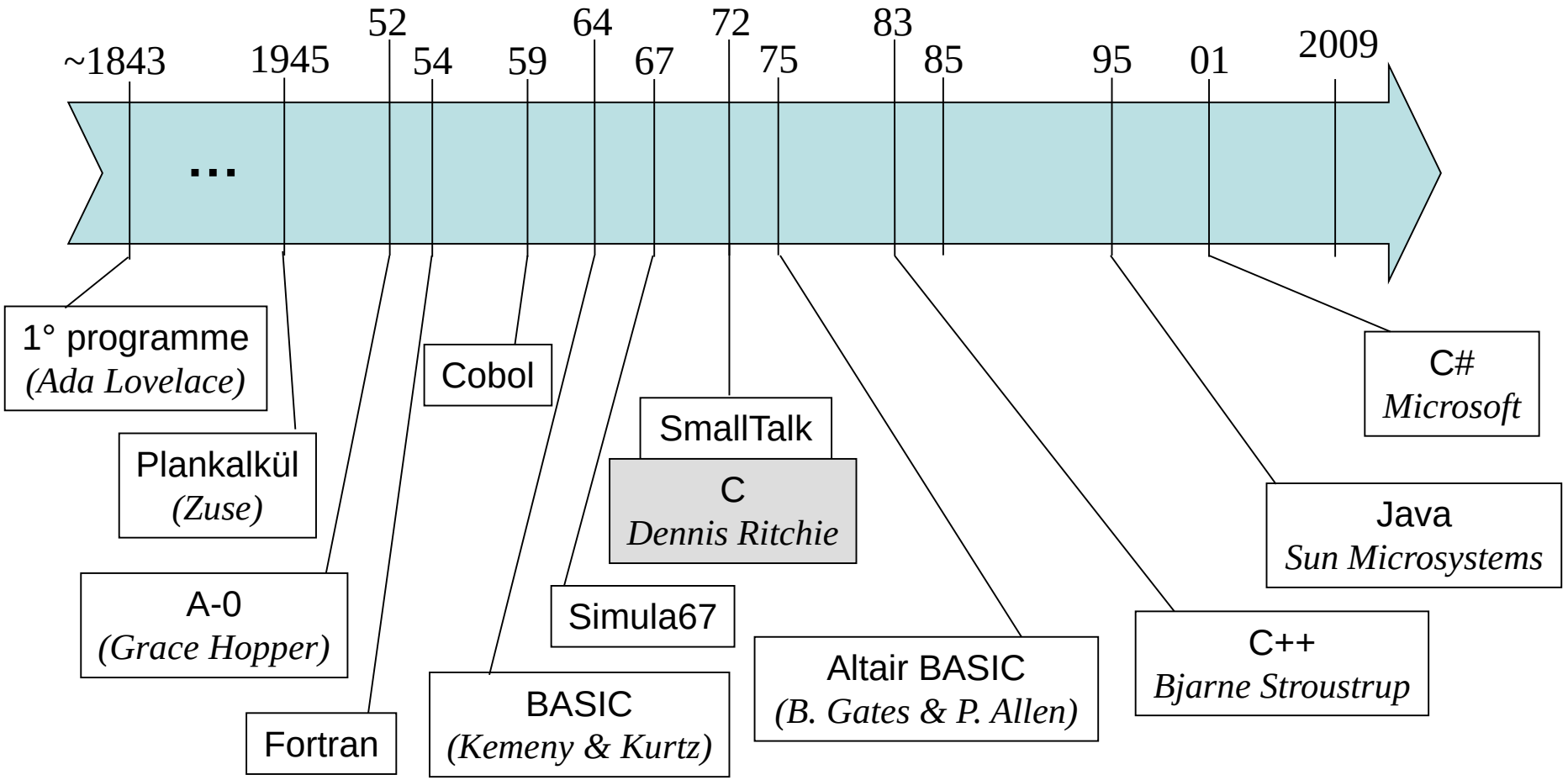
(OO)
\ ~ /

Date du premier programme (langage) informatique ?

Ada Byron, Comtesse de Lovelace
(Ada Lovelace)
1815-1852

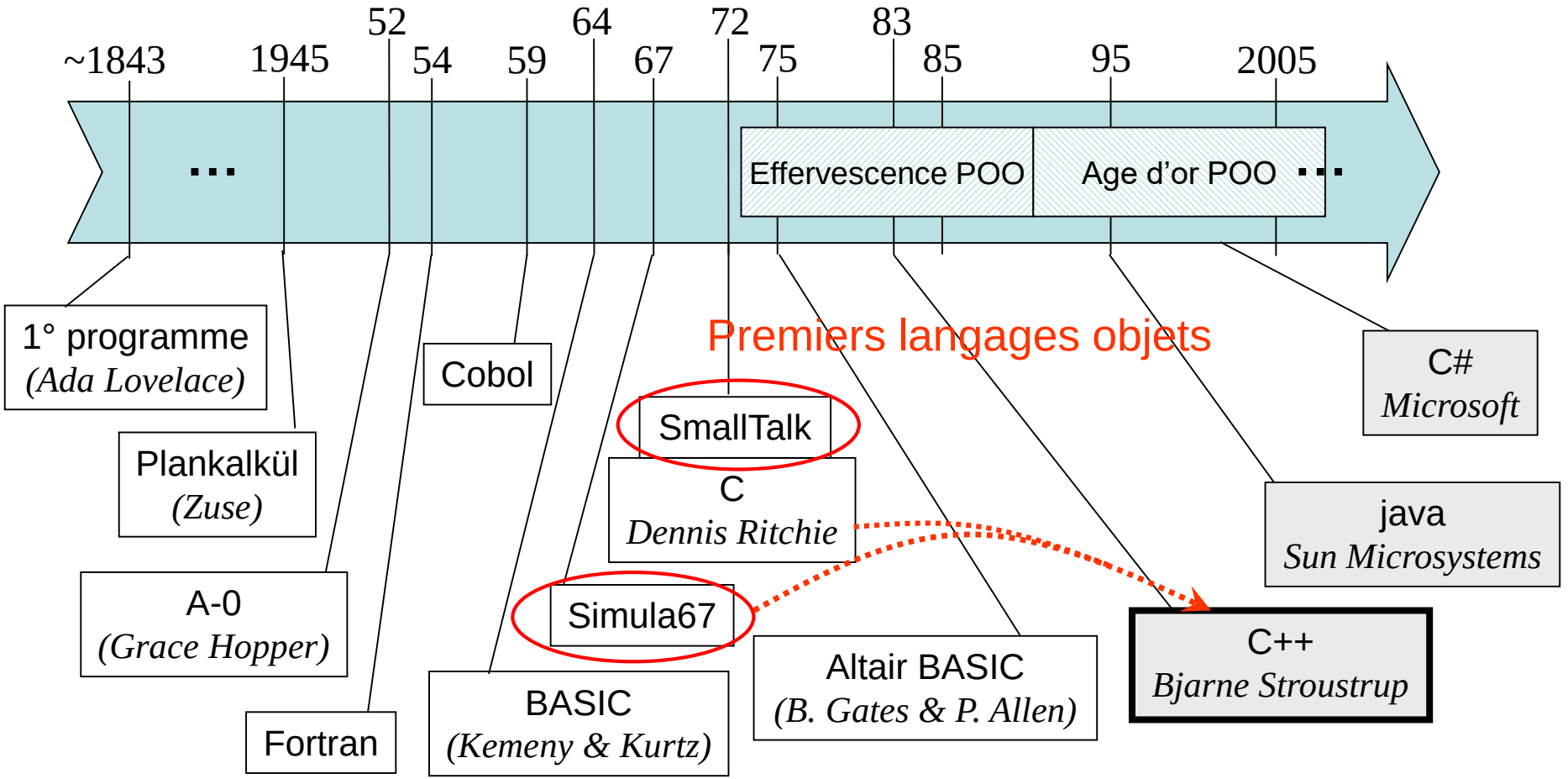
Introduction, POO

- Langages informatiques et OO:



Introduction, POO

- Langages informatiques et OO:



POO, origine de l'effervescence OO

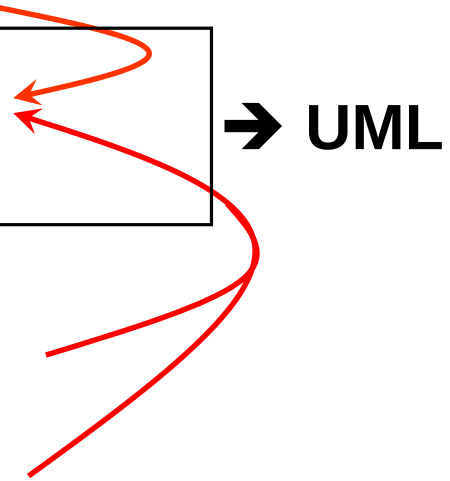
- L'OO en informatique aujourd'hui?
  - Quasi-totalité des applications programmées en OO (POO)
  - Quasi tous les langages intègrent la notion d'objet (même fortran et cobol!)

> L'OO permet de répondre aux exigences actuelles des applications informatiques (type PC):

- Applications conviviales et graphiques (mode « fenêtre »)
- Applications complexes
  - Quantités des exigences client
  - développements, effectifs, temps
- Réutilisation du code
  - Investissement sur le développement
- Maintenance du code
  - Nombreux aller-retour entre validation et conception

Introduction

- Méthodes orientées objet :
  - Analyse (AOO)
  - Modélisation (MOO)
  - Conception (COO)
  - Bases de données (SGBDOO)
  - Programmation (POO)
  - …



➔ **Méthodologie Orientée Objet**

Design Pattern (+20 patterns : Factory, Decorator, Facade, …)

Model Driven Architecture (MDA), …

Exemple : génération automatique de code à partir d'une modélisation UML;

Introduction, UML

- UML : Unified Modeling Language
  - Langage de modélisation

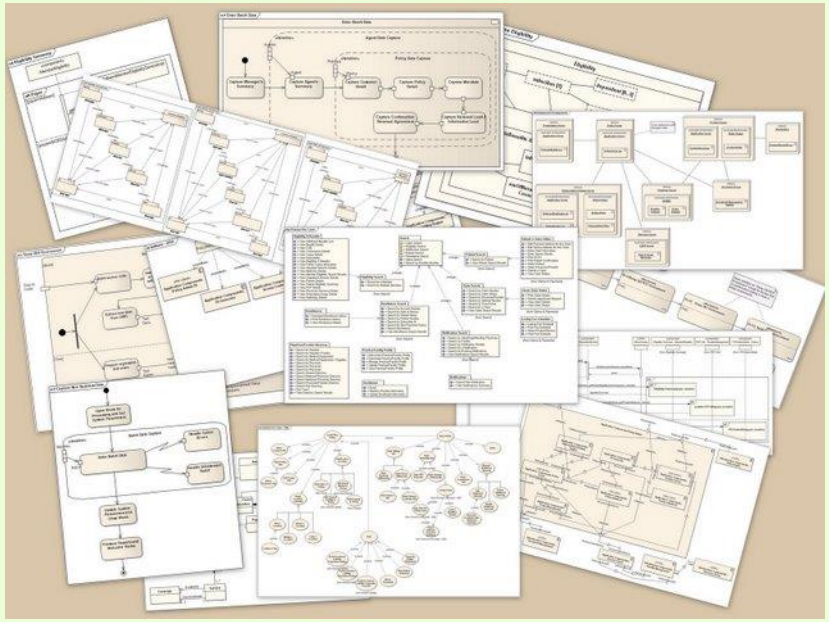
    Permet de représenter un système de manière abstraite
  - Langage formel

    Concis, précis et simple
  - Langage graphique

    Basé sur des diagrammes (13 pour UML 2.0)
  - Standard industriel

    Object Management Group, 1997
  - Initié par

    Grady Booch, James Rumbaugh et Ivar Jacobson



Introduction, UML

- UML, outils d'excellence pour
  - Concevoir des logiciels informatiques
  - Communiquer sur des processus logiciels ou d'entreprise
  - Présenter, documenter un système à différents niveaux de détails
- UML, généralisation aux domaines
  - Secteur de la banque et de l'investissement
  - Santé
  - Défense
  - Informatique distribuée, Systèmes embarqués
  - Systèmes complexes (pluri techniques, multi physique)
  - Secteur de la vente et de l'approvisionnement

Introduction, UML

- « Niveaux » de détails d'UML
  - UML comme une esquisse

    (Re)-présenter les points clés, faire l'analyse, mettre en place les premières grandes idées
  - UML comme un plan

    Spécifier en détail un système (diagrammes)

    Souvent associé à des systèmes logiciels et implique une retro-ingénierie pour que le modèle reste synchronisé avec le code
  - Langage de programmation

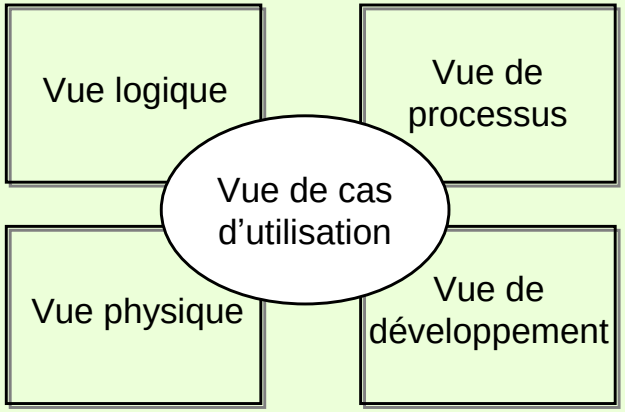
    Passer directement modèle UML → code exécutable

    Tous les aspects du système sont modélisés…

    {UML 2.0 est exécutable}

Introduction, UML

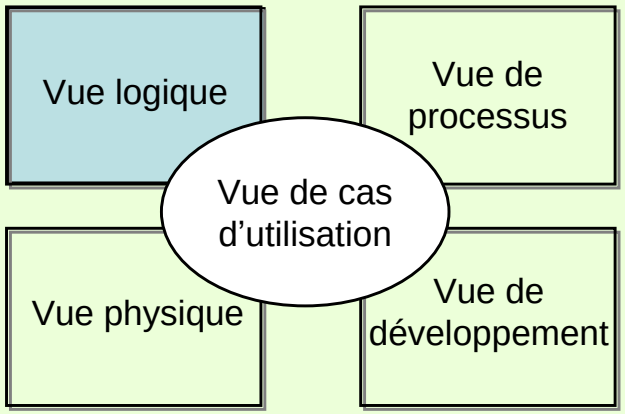
- Modèle de vue
  - organisation de l'analyse OO et des 13 diagrammes UML



Modèle de vue 4+1 de P. Kruchten

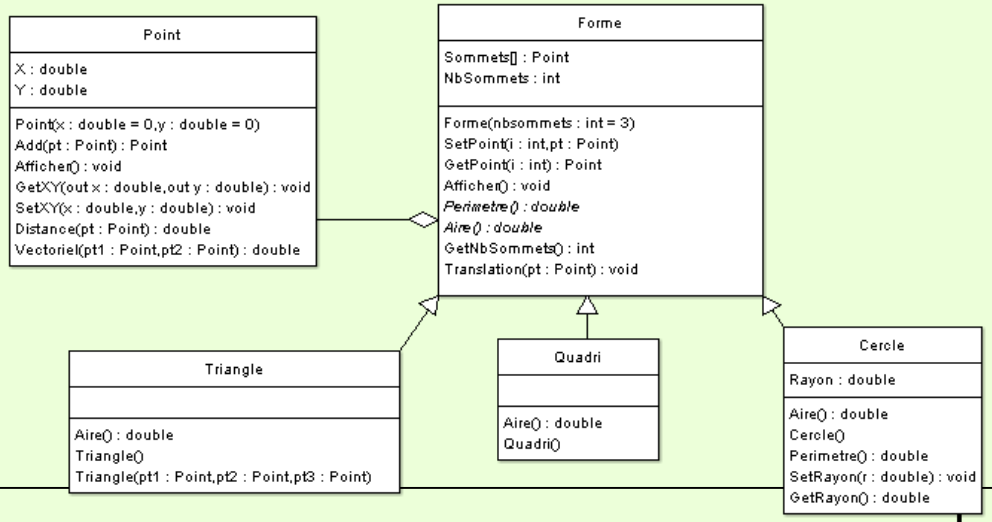
Introduction, UML

- Modèle de vue: Vue logique



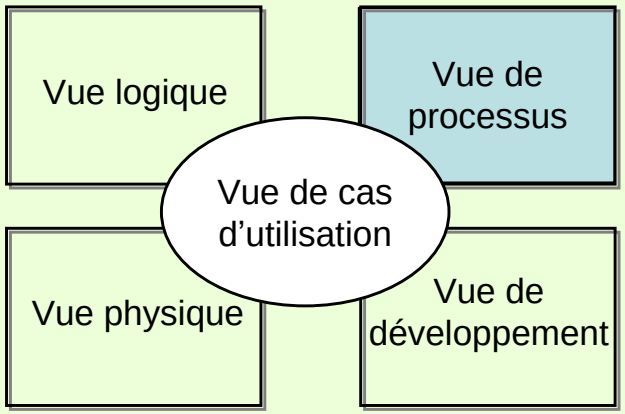
• Donne les descriptions abstraites des parties d'un système
• Sert à modéliser les composants d'un système et leurs interactions

Diagrammes associés:
- **de classes**,
- d'objets,
- de machines d'état
- d'interactions



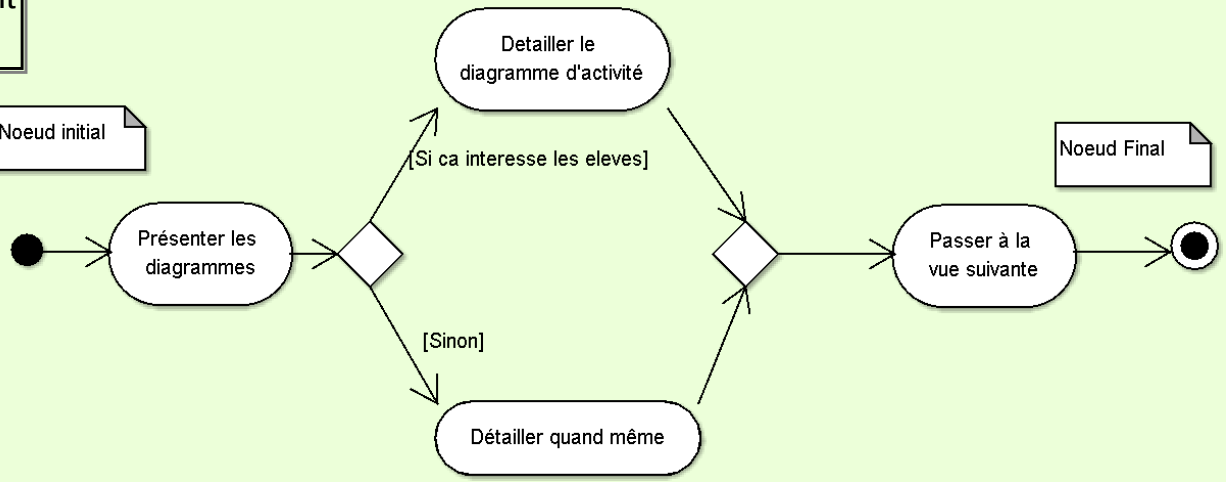
Introduction, UML

- Modèle de vue: Vue de processus



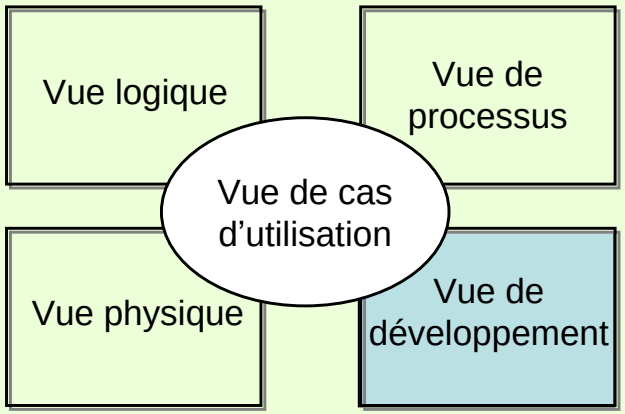
- Décrit les processus du système
- Utile pour la visualisation de l'activité du système

Diagrammes associés:
diagrammes d'activité



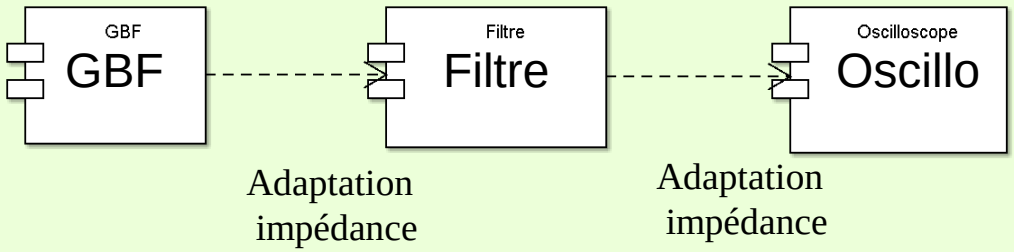
Introduction, UML

- Modèle de vue: Vue de développement



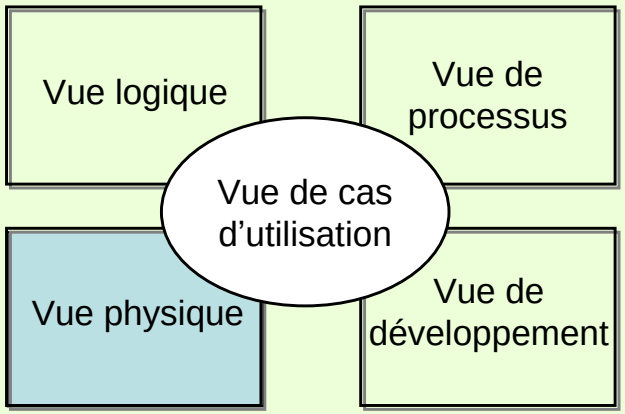
• Décrit l'organisation en modules et composants des parties du système
• Utile pour gérer les différentes couches de l'architecture du système

Diagrammes associés:
- paquetages
- **composants**



Introduction, UML

- Modèle de vue: Vue physique

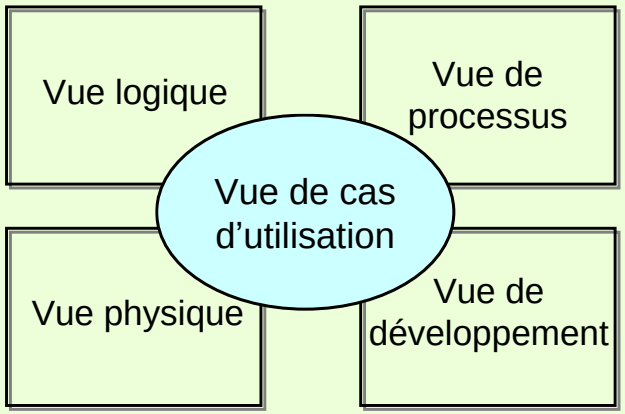


• Décrit comment la conception du système (les 3 autres vue) est mise en œuvre par un ensemble d'entités réelles

Diagramme associé:
diagramme de déploiement

Introduction, UML

- Modèle de vue: Vue de cas d'utilisation



- Décrit la fonctionnalité du système en cours de modélisation, point de vue du monde extérieur
- Nécessaire pour décrire ce que le système est supposé faire
- Toutes les autres vues s'appuient sur celle-ci

Diagrammes associés:
- **cas d'utilisation**
- des descriptions
- interactions globales

Étudiants dans un amphi

Introduction, résumé

- Illustration du concept objet avec les maths

  Encapsuler données et fonctions spécifiques aux données

- Langage de POO utilisé dans le cours: C++

  Basé sur le C et Simula67 (et Algol68), écrit par Bjarne Stroustrup (1980 « C with Class »; puis 1983: C++)

- Modélisation Orienté Objet : UML

  ➔ Diagramme de classes

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet
III. Encapsulation, concept de classe
IV. Héritage (généralisation)
V. Polymorphisme
VI. Généricité (modèles de classe)

Plan

I. Premier exemple

Des fonctions aux classes… *et UML*

II. Définitions de l'Orienté Objet

III. Encapsulation, concept de classe

IV. Héritage (généralisation)

V. Polymorphisme

VI. Généricité (modèles de classe)

Premier exemple

- Nombres complexes
  - En C, écrire une **procédure** permettant de faire la somme de 2 nombres complexes

```
void Somme(double a1, double b1,
       double a2, double b2, double *a, double *b)
{
  *a = a1 + a2;
  *b = b1 + b2;
}
```

$z_1 = 1 + j$

$z_2 = 2 - 4j$

```
int main(void)
{
  double re,im;
  Somme( 1, 1, 2, -4, &re, &im);
  printf(« %lf +j %lf », re, im);
return 0;
}
```

Premier exemple

- Nombres complexes
  - En C++, écrire une **procédure** permettant de faire la somme de 2 nombres complexes

```
void Somme(double a1, double b1,
       double a2, double b2, double *a, double *b)
{
  *a = a1 + a2;
  *b = b1 + b2;
}
```

$z_1 = 1 + j$

$z_2 = 2 - 4j$

```
int main(void)
{
  double re,im;
  Somme( 1, 1, 2, -4, &re, &im);
  cout<< re << "+j" << im << endl;
return 0;
}
```

Premier exemple

- Nombres complexes, avec structure (du C)
  - Ecrire une **procédure** permettant de faire la somme de 2 nombres complexes

```
typedef struct
{ double Reel;
  double Imag;
} Complexe;
```

⇨ `Complexe` est maintenant un nouveau type

```
void Somme(Complexe z1, Complexe z2, Complexe *z3)
{
  z3->Reel = z1.Reel + z2.Reel;
  z3->Imag = z1.Imag + z2.Imag;
}
```

```
void main()
{ Complexe z1, z2, z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
Somme( z1, z2, &z3);
}
```

Premier exemple

- Nombres complexes, fonction avec structure
  - Ecrire une **fonction** permettant de faire la somme de 2 nombres complexes

```
typedef struct
{ double Reel;
  double Imag;
} Complexe;
```

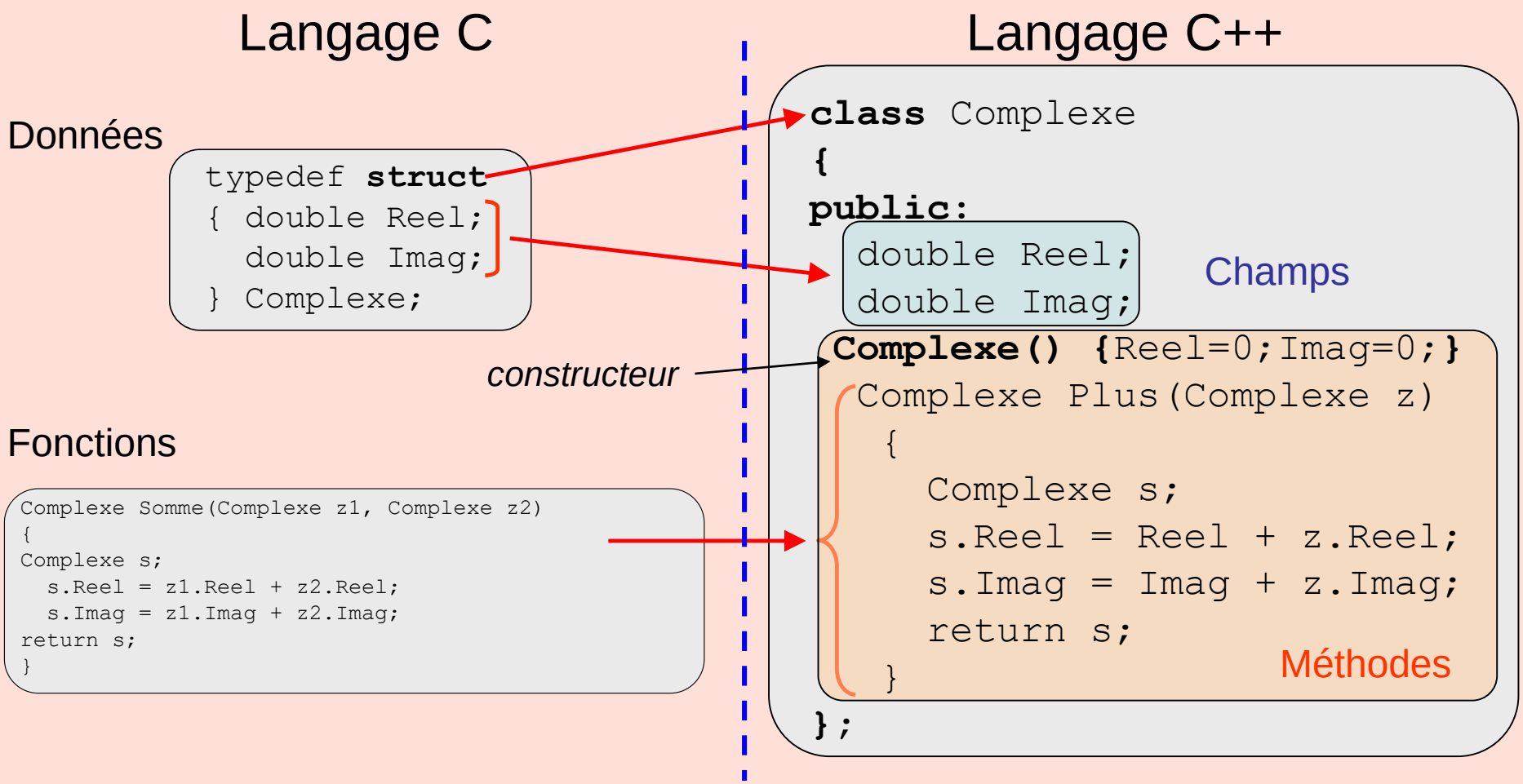
⇨ `Complexe` est maintenant un nouveau type

```
Complexe Somme(Complexe z1, Complexe z2)
{
Complexe s;
  s.Reel = z1.Reel + z2.Reel;
  s.Imag = z1.Imag + z2.Imag;
return s;
}
```

```
void main()
{ Complexe z1, z2, z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = Somme( z1, z2);
}
```

Premier exemple

- Nombres complexes, du C au C++



Premier exemple

- Nombres complexes, C++

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

```
#include <iostream>
using namespace std;
…
…
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

Objets

Premier exemple

- Explications de la fonction `main` C++

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

1) 3 objets Complexe (z1, z2, z3) sont créés
un constructeur de la classe est exécuté

En mémoire :

z1	z2	z3
Reel = 0	Reel = 0	Reel = 0
Imag = 0	Imag = 0	Imag = 0

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Premier exemple

- Explications de la fonction `main` C++

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1) 3 objets Complexe (z1, z2, z3) sont créés
   un constructeur de la classe est exécuté

2) On modifie les champs de z1 et z2
   Accès direct aux valeurs (`public`…)

En mémoire :

z1	z2	z3
Reel = 1	Reel = 2	Reel = 0
Imag = 1	Imag = -4	Imag = 0

Premier exemple

- Explications de la fonction `main` C++

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1) 3 objets Complexe (z1, z2, z3) sont créés
   Un constructeur de la classe est exécuté

2) On modifie les champs de z1 et z2
   Accès direct aux valeurs (`public`…)

3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

En mémoire :

z1	z2	z3
Reel = 1	Reel = 2	Reel = 3
Imag = 1	Imag = -4	Imag = -3

Premier exemple

- Explications de la fonction `main` C++

  3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

```
z3 = z1.Plus(z2);
```

- En mémoire, à l'appel de la fonction *Plus*

```
z3 = z1.Plus(z2);
```

z1	z2	z3
Reel = 1 Imag = 1	Reel = 2 Imag = -4	Reel = 0 Imag = 0

- En mémoire, **dans** la fonction *Plus*

z	s
Reel = 2 Imag = -4	Reel = 0 Imag = 0

Premier exemple

- Explications de la fonction `main` C++

  3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

- En mémoire, à l'appel de la fonction *Plus*

`z3 = z1.Plus(z2);`

z1	z2	z3
Reel = 1	Reel = 2	Reel = 0
Imag = 1	Imag = -4	Imag = 0

- En mémoire, **dans** la fonction *Plus*

z	s
Reel = 2	Reel = **3**
Imag = -4	Imag = **-3**

Premier exemple

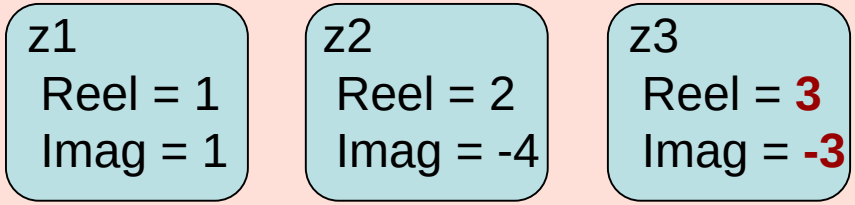
- Explications de la fonction `main` C++

  3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3

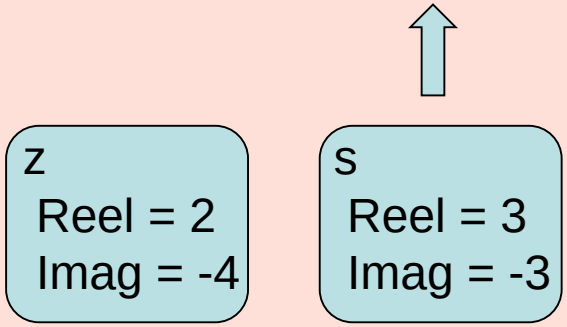
```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

- En mémoire, à l'appel de la fonction *Plus*

```
z3 = z1.Plus(z2);
```



- En mémoire, **dans** la fonction *Plus*



Premier exemple

- Explications de la fonction `main` C++

```
int main(void)
{
  Complexe z1,z2,z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = z1.Plus(z2);
  cout << z3.Reel<< endl;
  cout << z3.Imag << "j \n";
 return 0;
}
```

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;};
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

1) 3 objets Complexe (z1, z2, z3) sont créés
   Un constructeur de la classe est exécuté
2) On modifie les champs de z1 et z2
   Accès direct aux valeurs (`public`…)
3) La fonction `Plus` de z1 est exécutée avec z2 comme paramètre, le résultat est copié dans z3
4) Affichage du résultat:

```
3
-3j
```

En mémoire :

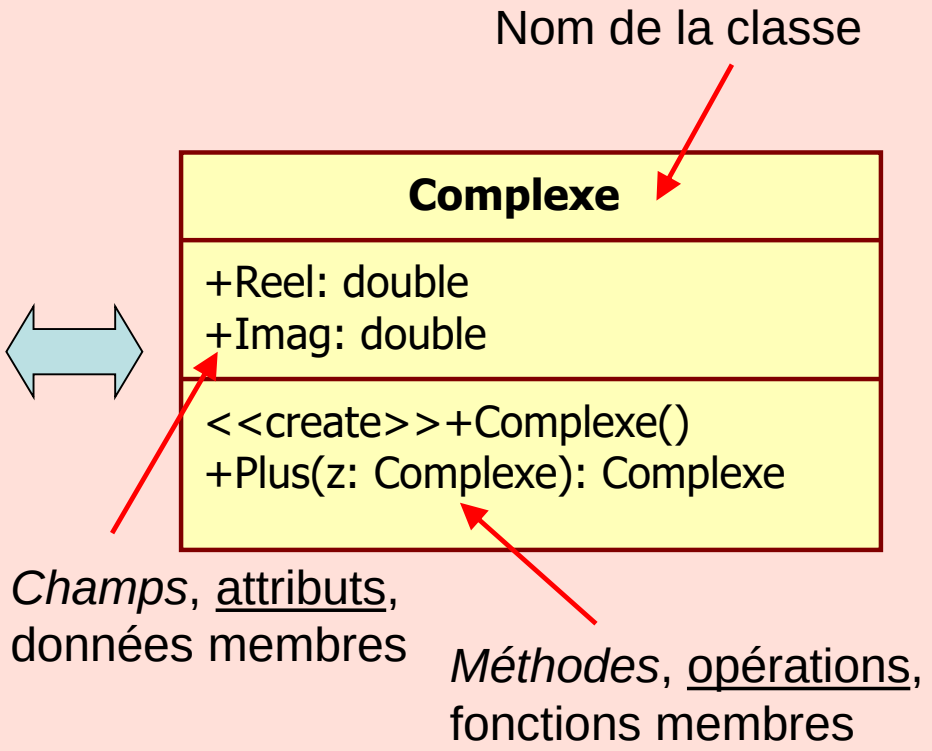
z1	z2	z3
Reel = 1	Reel = 2	Reel = 3
Imag = 1	Imag = -4	Imag = -3

Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implémentation



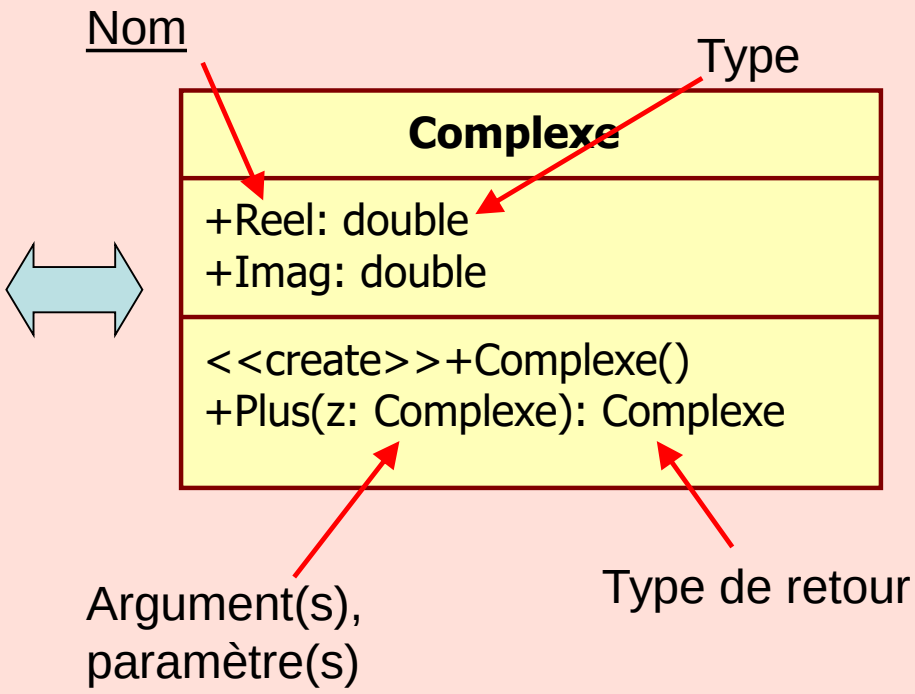
Modélisation

Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
   double Reel;
   double Imag;
   Complexe(){Reel=0;Imag=0;}
   Complexe Plus(Complexe z)
    {
      Complexe s;
      s.Reel = Reel + z.Reel;
      s.Imag = Imag + z.Imag;
      return s;
    }
};
```

Implémentation



Modélisation

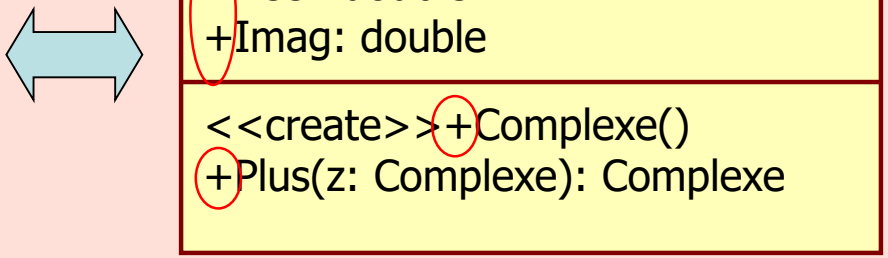
Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
   double Reel;
   double Imag;
   Complexe(){Reel=0;Imag=0;}
   Complexe Plus(Complexe z)
    {
      Complexe s;
      s.Reel = Reel + z.Reel;
      s.Imag = Imag + z.Imag;
      return s;
    }
};
```

Implémentation

Visibilité, *accessibilité*



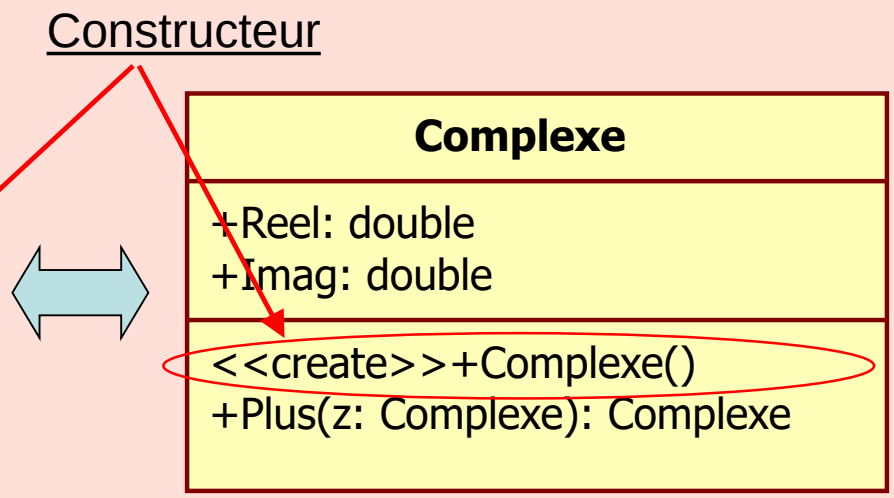
Modélisation

Premier exemple

- Nombres complexes, C++ et UML

```
class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe(){Reel=0;Imag=0;}
  Complexe Plus(Complexe z)
   {
     Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
     return s;
   }
};
```

Implémentation



Modélisation

Premier exemple, exo 1

- Etendre la classe Complexe aux fonctionnalités:
  - Conjuguer() : retourner le complexe conjugué
  - Moins(…): soustraction de deux complexes
  - MultiplierPar(…) : produit de deux complexes
  - DiviserPar(…): division de deux complexes
  - Retourner le module du complexe
  - Retourner l'argument du complexe (en radians)

Premier exemple, exo 2

- Modéliser et concevoir un *PorteMonnaie* permettant de gérer une somme d'argent définie au départ. On pourra:
  - ***Ajouter*** de l'argent
  - ***Enlever*** de l'argent, à condition qu'il en reste suffisamment (signaler ce problème à l'utilisateur)
  - ***Afficher*** la somme restant dans le porte monnaie

→ Il faudra interdire l'accès direct à l'attribut gérant la somme d'argent (être obliger de passer par les opérations *Ajouter* et *Enlever* pour le modifier)

Premier exemple, exo 4

- Modéliser une basse (guitare). *(pas plus d'informations…)*

→ La modélisation dépend des besoins du système!
→ Il n'y a donc pas de modélisation unique d'un même concept abstrait

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet … Deuxième partie…

III. Encapsulation, concept de classe

IV. Héritage (généralisation)

V. Polymorphisme

VI. Généricité (modèles de classe)