ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2017-2018

Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. Intro, concepts, 1 exemple		*1. Organisation objet des données*
2. Modélisation objet / UML		*2. Diagrammes de classe UML*
3. C++ pratique 1		*3. C++ pratique, E/S, string, vector*
4. C++ pratique 2		*4. C++ pratique, type &, surcharge*
5. Classes & C++ : bases		*5. Classes & collec. objets en C++*
6. Classes & C++ : compléments		*6. UML et C++, associations*
7. Conteneurs & C++ : la STL		*7. Gestion de collections complexes*
8. Héritage / polymorphisme		*8. Collections polymorphes*
9. Abstraction / design patterns		*9. Framework, exemples 2 patterns*
10. Exceptions, flots, fichiers ...		*10. Flots / parsing / fichiers / except.*
11. Templates côté développeur		*11. Suivi de projet*
12. Synthèse, complément, révision		*12. Soutenance de **projet** ...*

COURS 1

A) Présentation C++ / contexte
B) Programmation Orientée Objet
C) Du C au C++ sur un exemple

COURS 1

A) Présentation C++ / contexte
B) Programmation Orientée Objet
C) Du C au C++ sur un exemple

Présentation C++ / contexte

- *Développement de grosses applications*
  → *Besoin de sécurité, ré-utilisabilité, grosses équipes*
- *Les **types de données structurées** se retrouvent au centre de la conception : prog. « **orientée objet** »*
- *Des langages font le choix de la rupture*
  → *« orienté objet » comme approche exclusive*
- *Le C++ fait un choix dans la continuité du C...*
  - *Compatible (presque) avec le code source C*
  - *Compilé, fortement typé, optimisations poussées*
  - *L'orienté objet vient « en plus »*
  - ***Approche maximaliste** : ++ de concepts !*

Présentation C++ / contexte

- *Initié dans les années 1980 …*

Bjarne Stroustrup, initiateur et grand gourou du C++

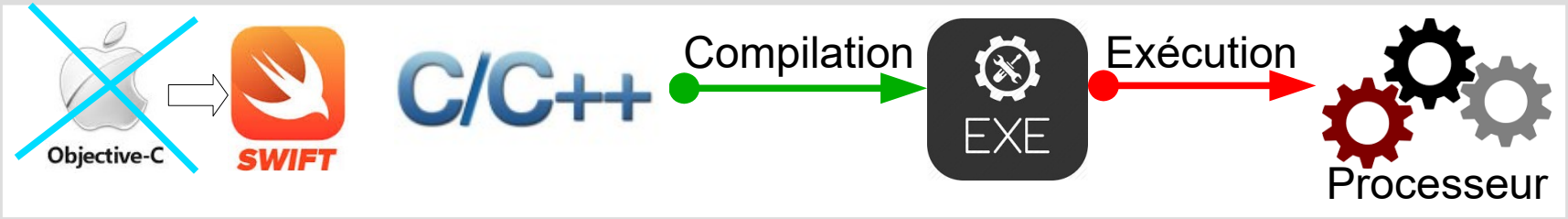
Présentation C++ / contexte

- *Le C++ ne cesse d'évoluer*
  - *Constructions natives du langage (primitives)*
  - *Bibliothèques (boites à outils)*
  - *Outils et écosystème (compilateurs, IDEs …)*
  - *Bonnes pratiques (expérience, expertise)*
- *Les versions successives sont compatibles mais attention les pratiques évoluent …*
- *C++98 1ère version normalisée du langage*
- ***C++11** évolution majeure, C++ « moderne »*
- *C++14 C++17 C++20 (à venir)*

Présentation C++ / contexte

3 schémas d'exécution de langages

- *Les langages **compilés** produisent des **exécutables** directement exploitables par le(s) processeur(s)*
  Performances optimales (si bien utilisés !)



Présentation C++ / contexte

3 schémas d'exécution de langages

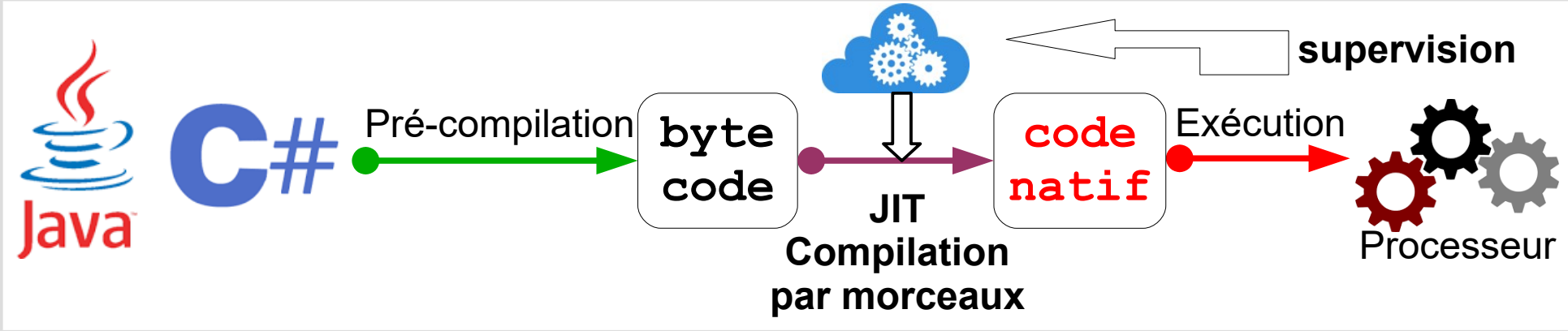
- *Les langages **interprétés** exécutés indirectement par une « **machine virtuelle** » qui est un exécutable (le plus souvent codé en C/C++) Plus souples et confortables mais moins performants (plus lents/lourds)*

Présentation C++ / contexte

3 schémas d'exécution de langages

- *Les langages **à bytecode** et **compilation JIT** (Just In Time) qui sont des langages dits «**managés**» sont pré-compilés en un pseudo langage machine (le bytecode) puis une machine virtuelle traduit ce bytecode en code natif (exécutable) à la volée.*

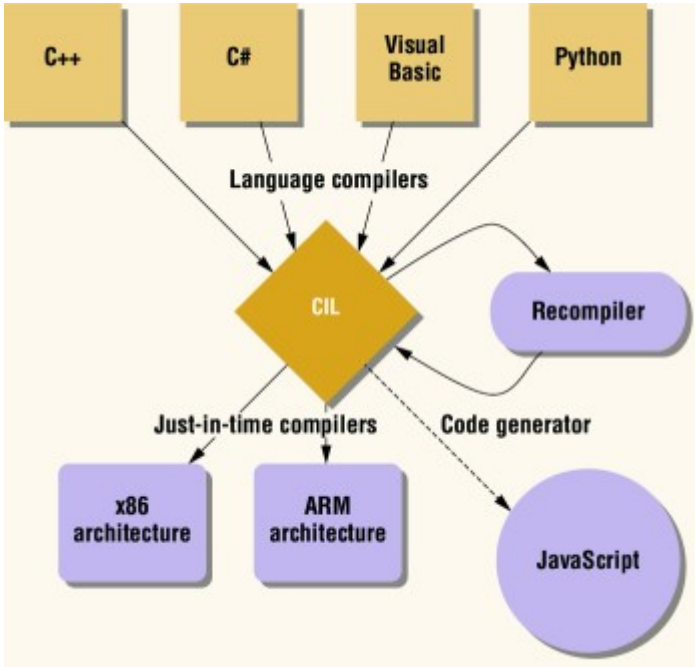
  La machine virtuelle (ou « runtime ») n'exécute plus par procuration, elle supervise et optimise la traduction du bytecode en code natif exécuté par le processeur



Présentation C++ / contexte

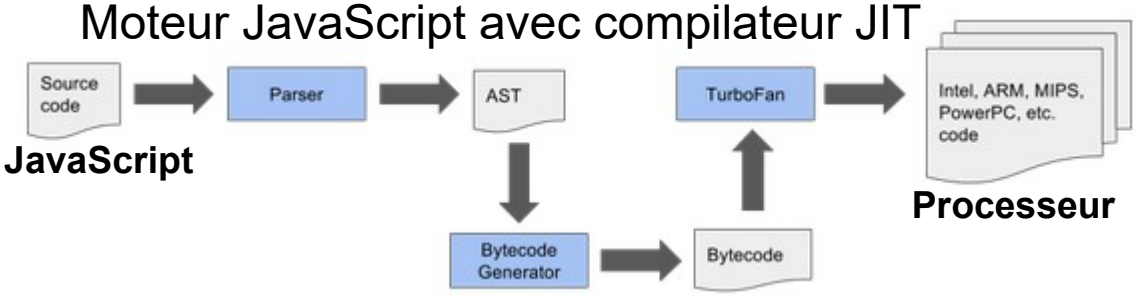
3 schémas d'exécution de langages

- *Distinction traditionnelle compilé / interprété brouillée*
- *Le Java a commencé comme un bytecode interprété*
- *Des langages historiquement interprétés comme Python et Javascript se retrouvent aussi JIT compilés*



Microsoft propose un bytecode intermédiaire CIL *Common Intermediate Language* et architecture la plateforme .NET pour être « language agnostic »

Chrome V8 (Chrome, Opera, Node.js ...)
Moteur JavaScript avec compilateur JIT



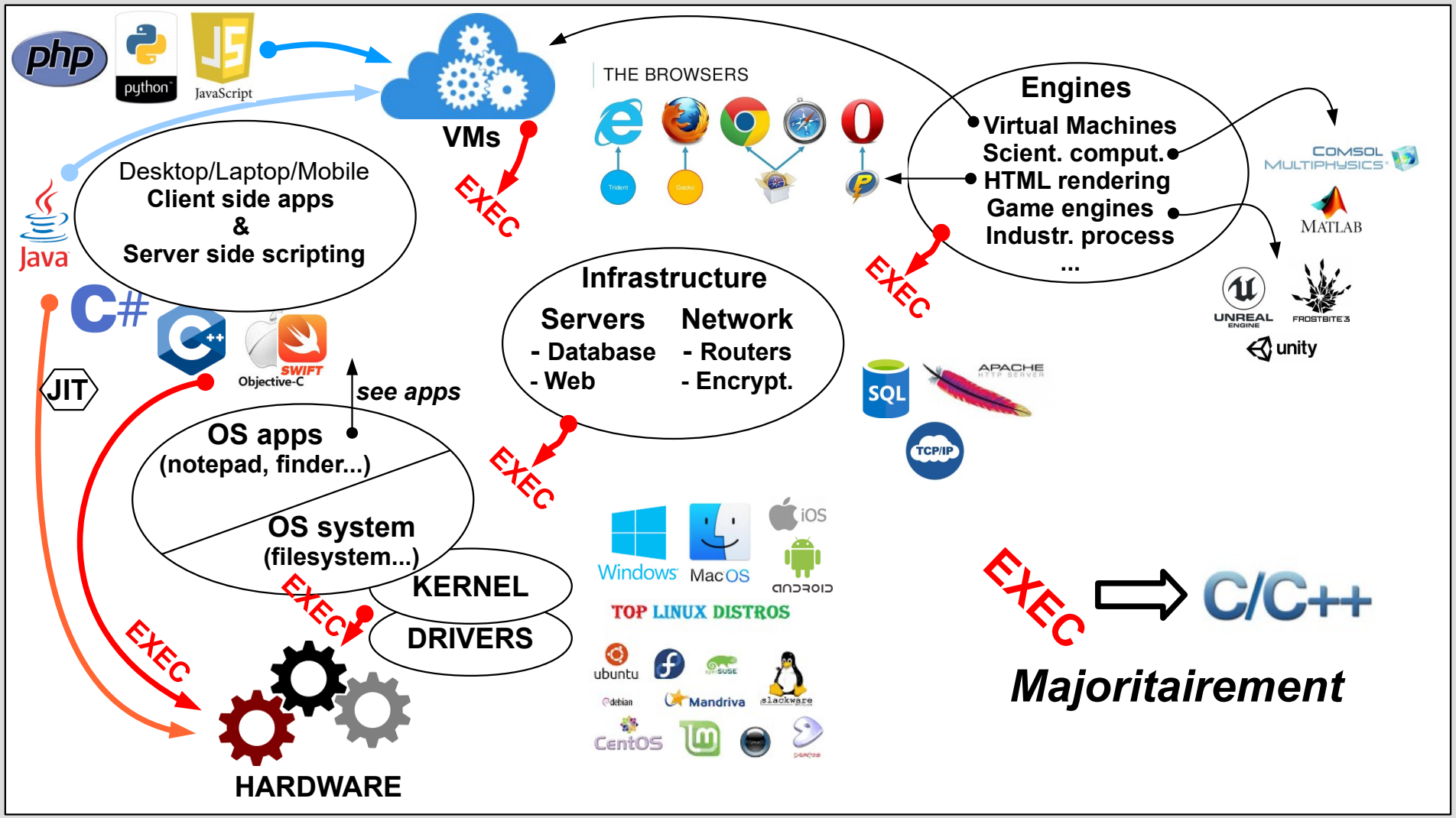
Présentation C++ / contexte

Trop compliqué ? *A retenir :*

- *Les langages « modernes » faciles à programmer sont des langages interprétés et/ou « managés » : ils ont besoin d'une machinerie auxiliaire au runtime*
- *Le C/C++ (et objective C, remplacé par Swift, Apple-centric) sont compilés « à l'ancienne » ce qui offre la meilleure performance pour l'exploitation des ressources*
- *L'absence d'intermédiaire et de supervision lors de l'exécution d'un code compilé natif implique que le développeur C/C++ gère lui même finement les ressources, en particulier la mémoire allouée (dur!)*
- ***Tous ces langages (sauf le C) sont « orienté objet »***

Présentation C++ / contexte

Un vaste écosystème



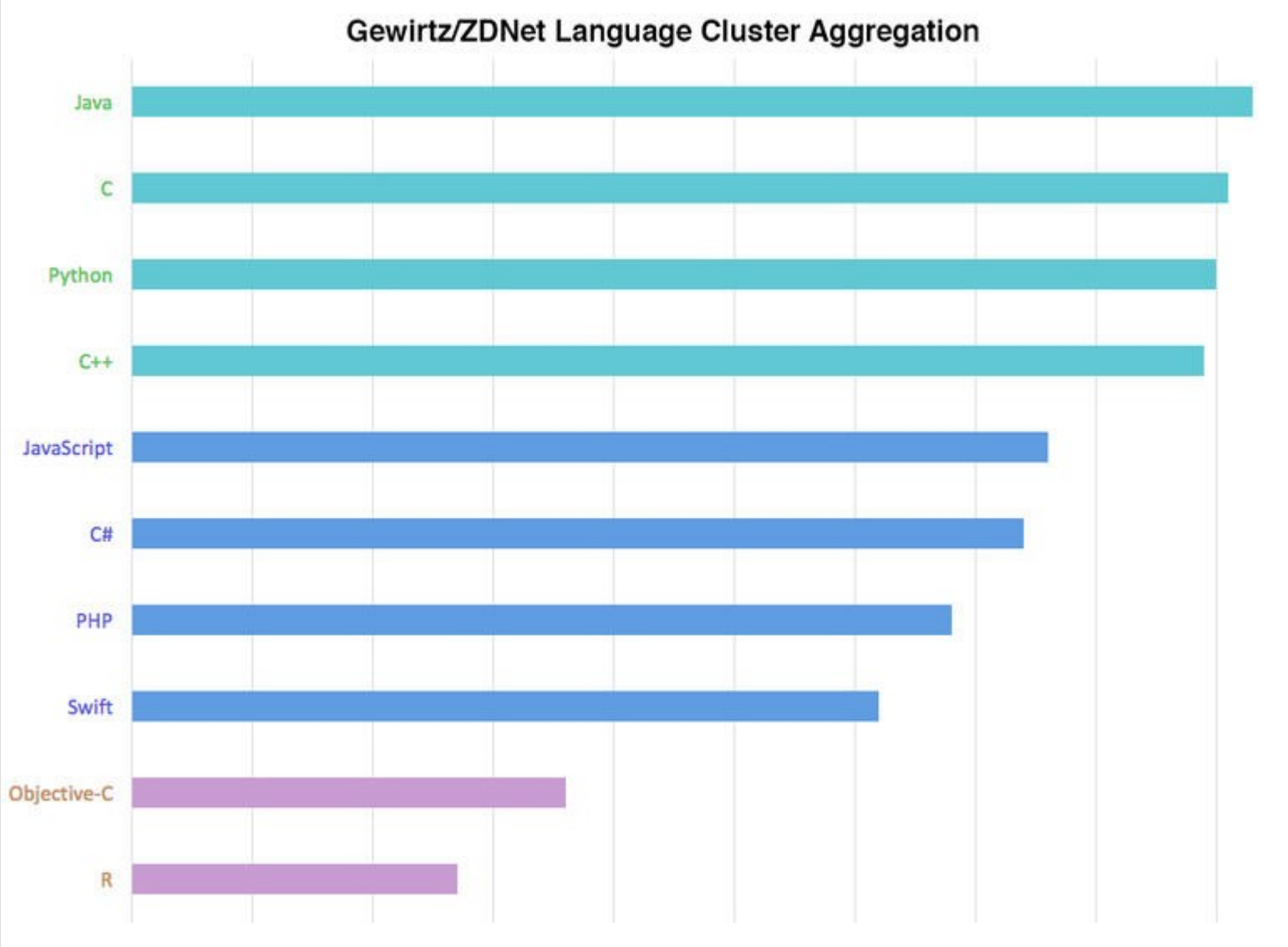
Présentation C++ / contexte

Ça veut dire que C/C++ recrute plus ?

- *Pas forcément ! Une majorité des cycles processeurs exécutent du C/C++ compilé (c'est le « carburant »)*
- *Mais la « couche applicative » recrute plus en total (il y a plus d'emplois de chauffeurs que de mécanos)*
- *Typiquement il y a plus de lignes de code **appelant** (client) que de code **appelé** (bas niveau / biblio.) Faciliter le travail client est une des raison de l'objet...*

Présentation C++ / contexte

Popularité 2017 (source)



COURS 1

A) Présentation C++ / contexte
B) Programmation Orientée Objet
C) Du C au C++ sur un exemple

Programmation Orientée Objet

Prenons de la hauteur

- *Une majorité de ces langages de programmation industrielle ont une syntaxe + ou - dérivée du C (C++, objective C, Java, JavaScript, C#, PHP)*
- *Il y a des if/else des for des while des blocs { } etc...*
- *Avec quelques variantes ce sont des C orientés objet !*
- Au fait qu'est-ce que c'est que cette histoire d'objets ?

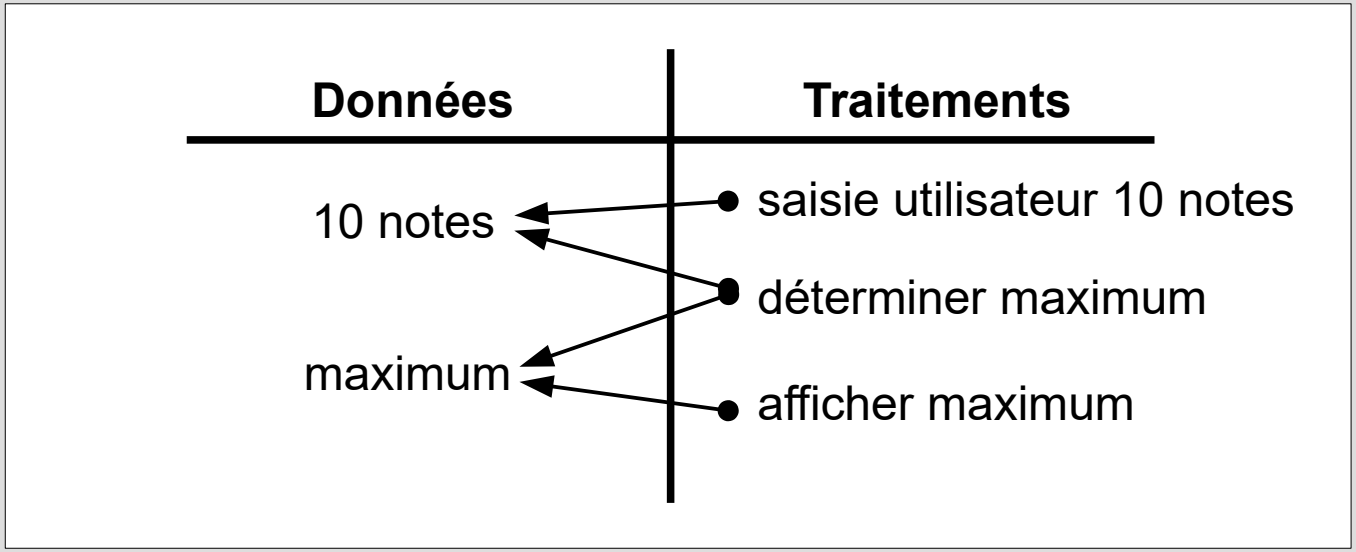
Programmation Orientée Objet

*On programme pour un **CDC***

- *Dans l'industrie logicielle on ne développe pas ni pour le fun ni pour faire plaisir à Stroustrup*
- *On développe pour fournir des solutions logicielles qui correspondent à des demandes / besoins / buts*
- *Ces buts sont spécifiés par un Cahier Des Charges précisant les objectifs, le périmètre, les fonctionnalités*
- *Partant d'un CDC comment arriver à la solution de manière sûre et efficace ? Il faut un plan !*

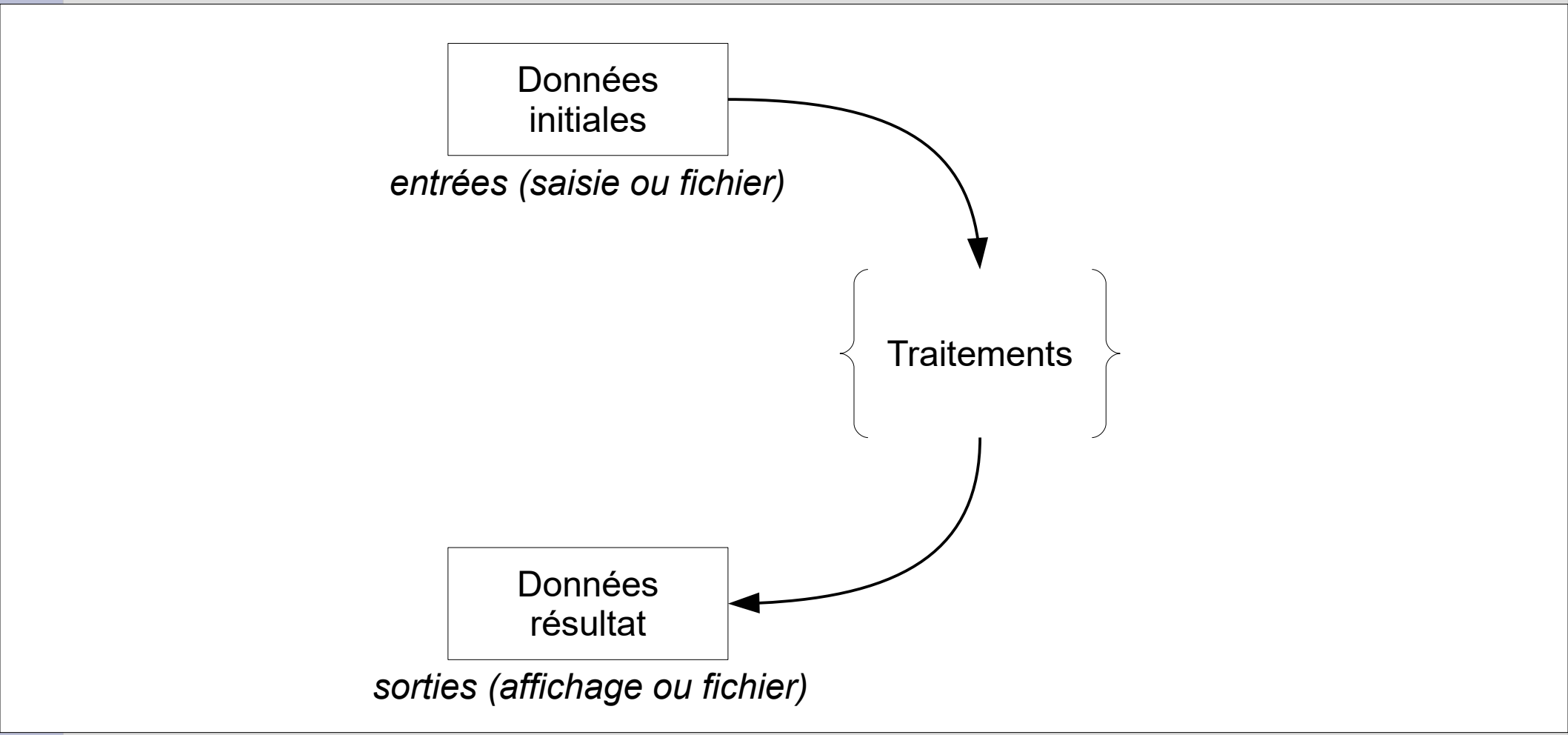
Programmation Orientée Objet

- Toutes les **méthodes de conception** distinguent
  - *Traitements* : les actions, ordres du programme
  - *Données* : ce qui est transformé par les actions, nombres ou symboles représentant des informations réelles ou virtuelles
- Une phase analyse/conception articule ce binôme



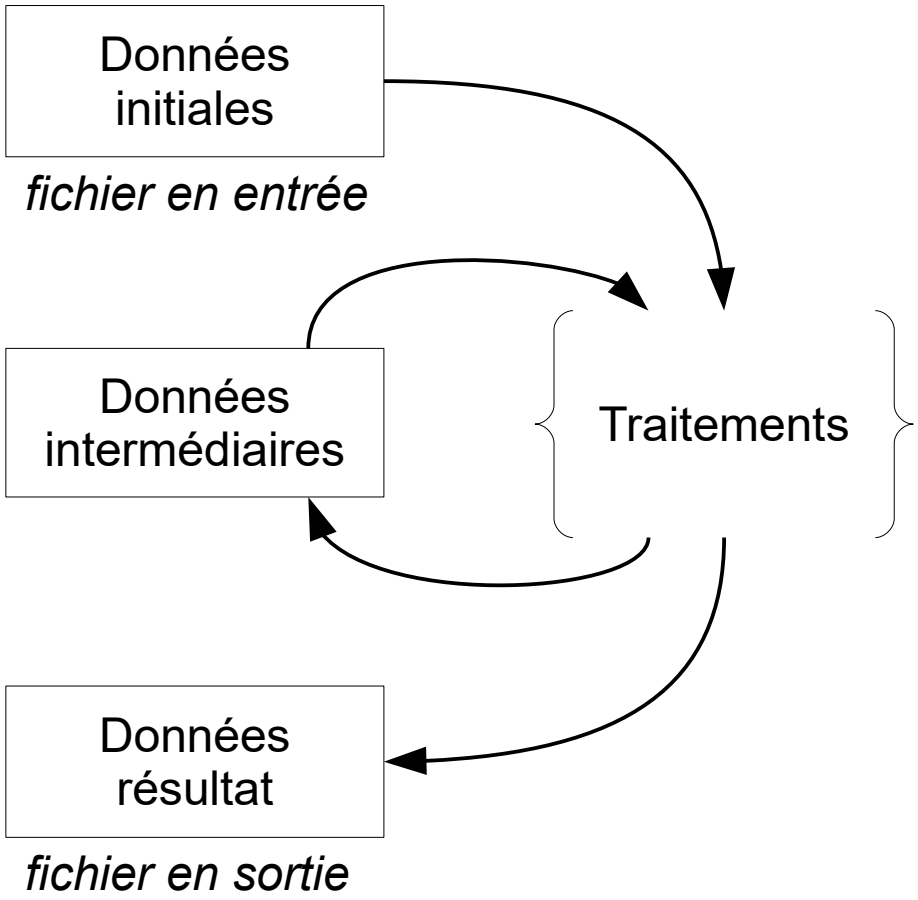
Programmation Orientée Objet

Programmes simples, le traitement est « au centre »



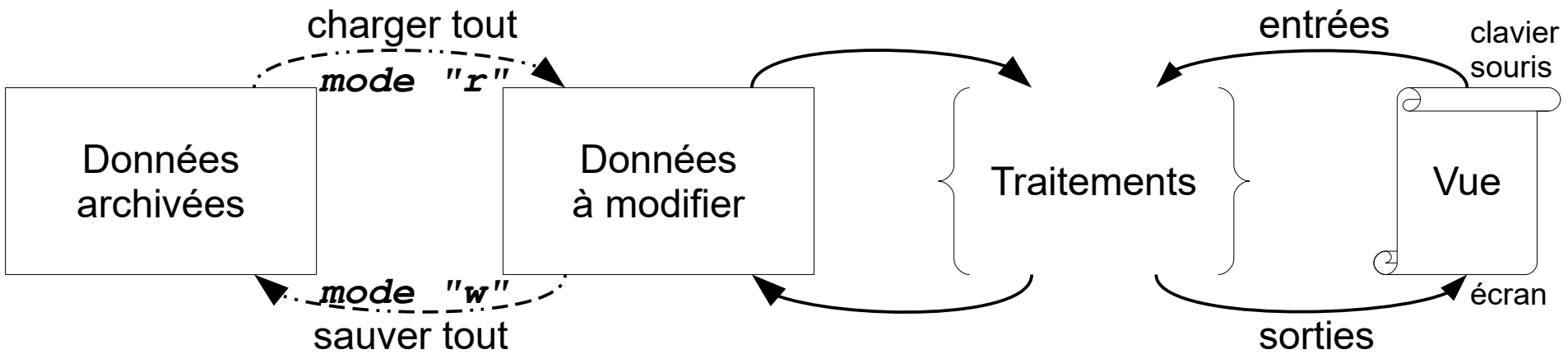
Programmation Orientée Objet

Algorithmes complexes : données intermédiaires



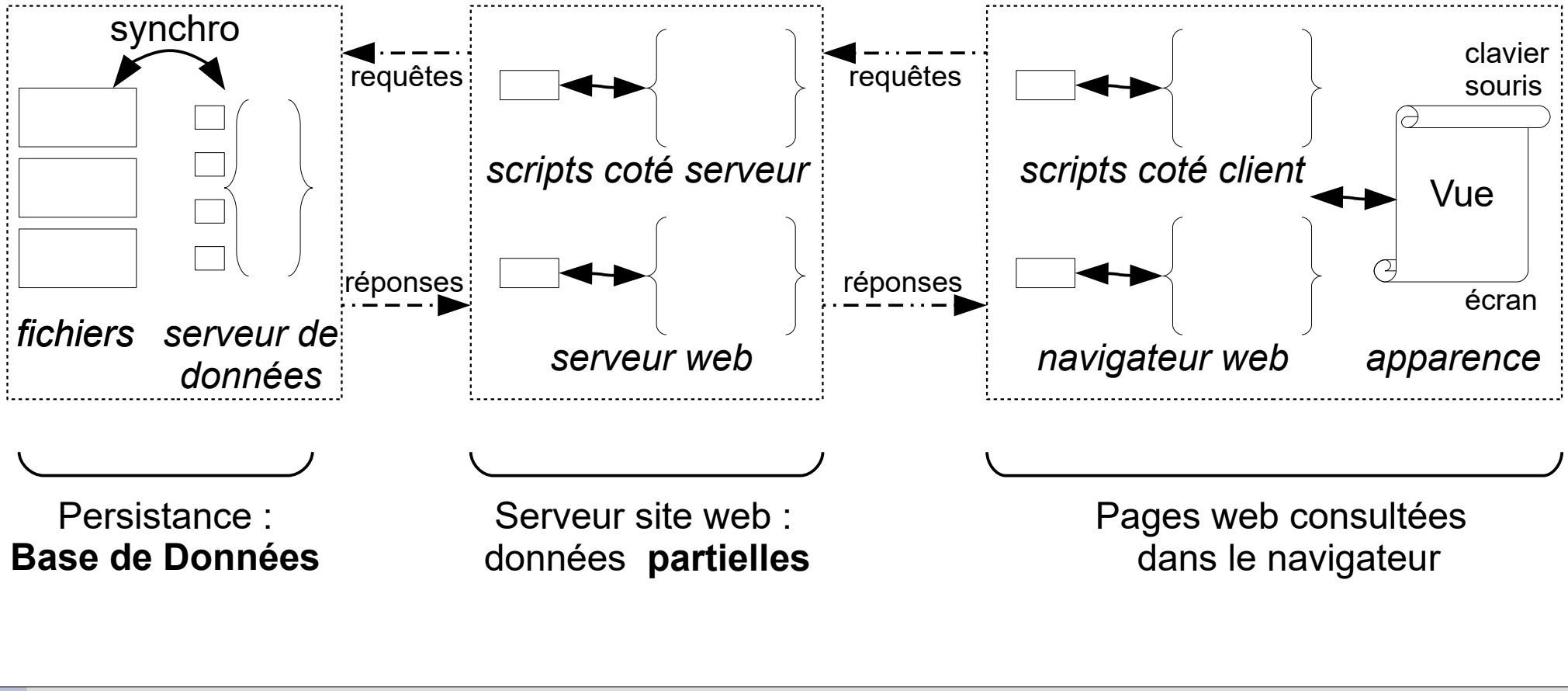
Programmation Orientée Objet

Modèle "application" : on travaille sur un document



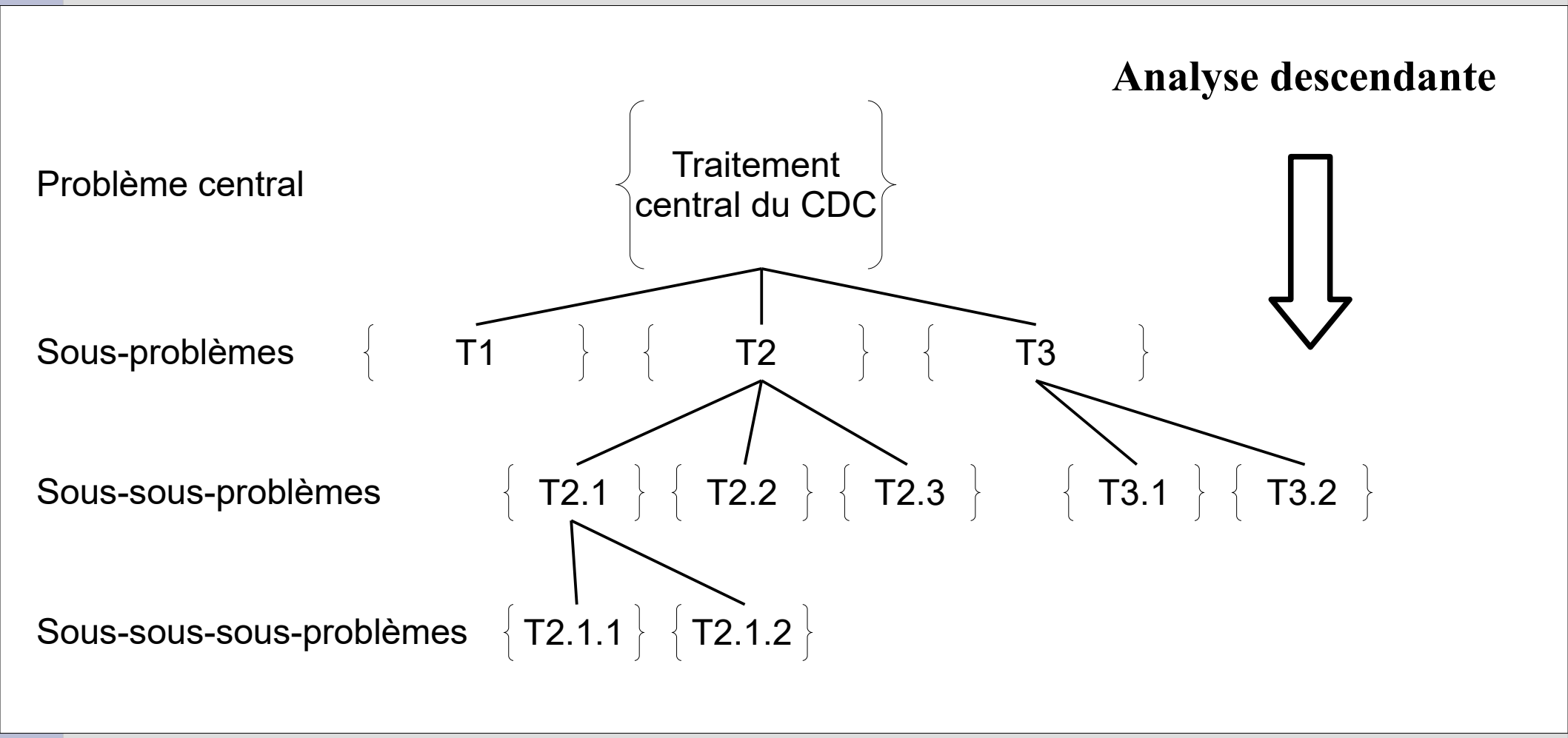
Programmation Orientée Objet

Architecture Client/Serveur : site web dynamique

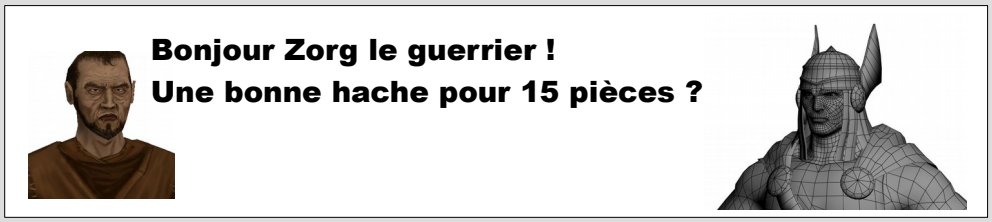


Programmation Orientée Objet

Ça se complique ! On peut toujours décomposer un problème de **traitement** en sous-problèmes...



Programmation Orientée Objet



Informations présentées à l'utilisateur
Messages / Images / Animations / Sons ...

Et décomposer les **données** complexes en données élémentaires

```
joueur → "Luc"
perso → "Zorg"
classe → 2
fortune → 12
```

association

0	"magicien"
1	"guérisseur"
2	"guerrier"
...	

-2.5214	3.2178	6.5789
5.7894	3.9000	2.1036
-3.2181	-4.7411	3.7877
1.6546	9.7865	6.5414
...		

Types composés
Tableaux / Structures

```
'L' 'u' …          12   -2427 …          -2.521423   7.354846e3 …
```

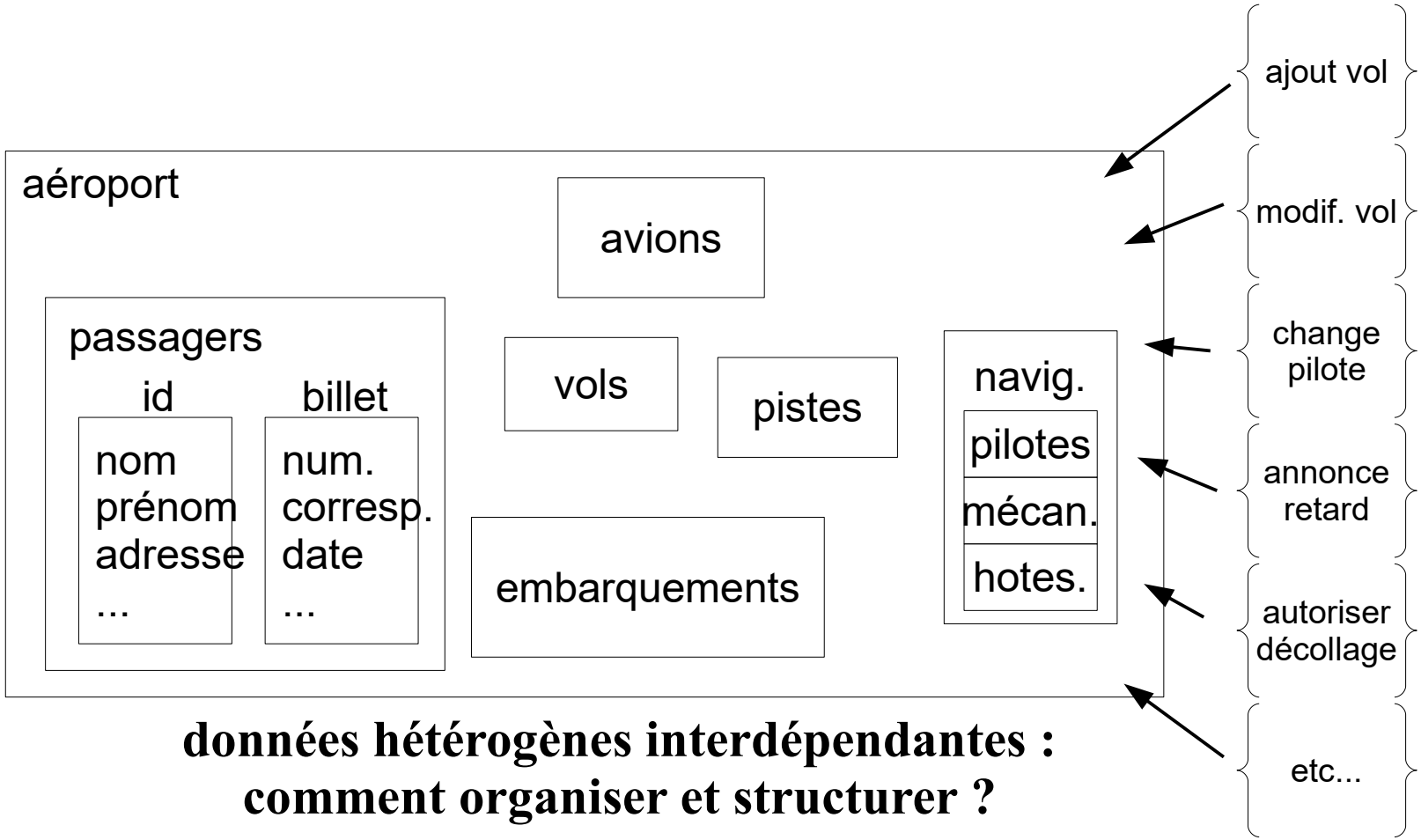
Types scalaires fondamentaux
Caractères / Entiers / Flottants / Pointeurs

```
… 01101110 11110100 00000001 01000000 10111000 10000010 10010100 …
```

Représentation binaire (niveau machine)

Programmation Orientée Objet

Mais on arrive aux problématiques d'organisation et d'assemblage d'ensembles "hétérogènes" de données

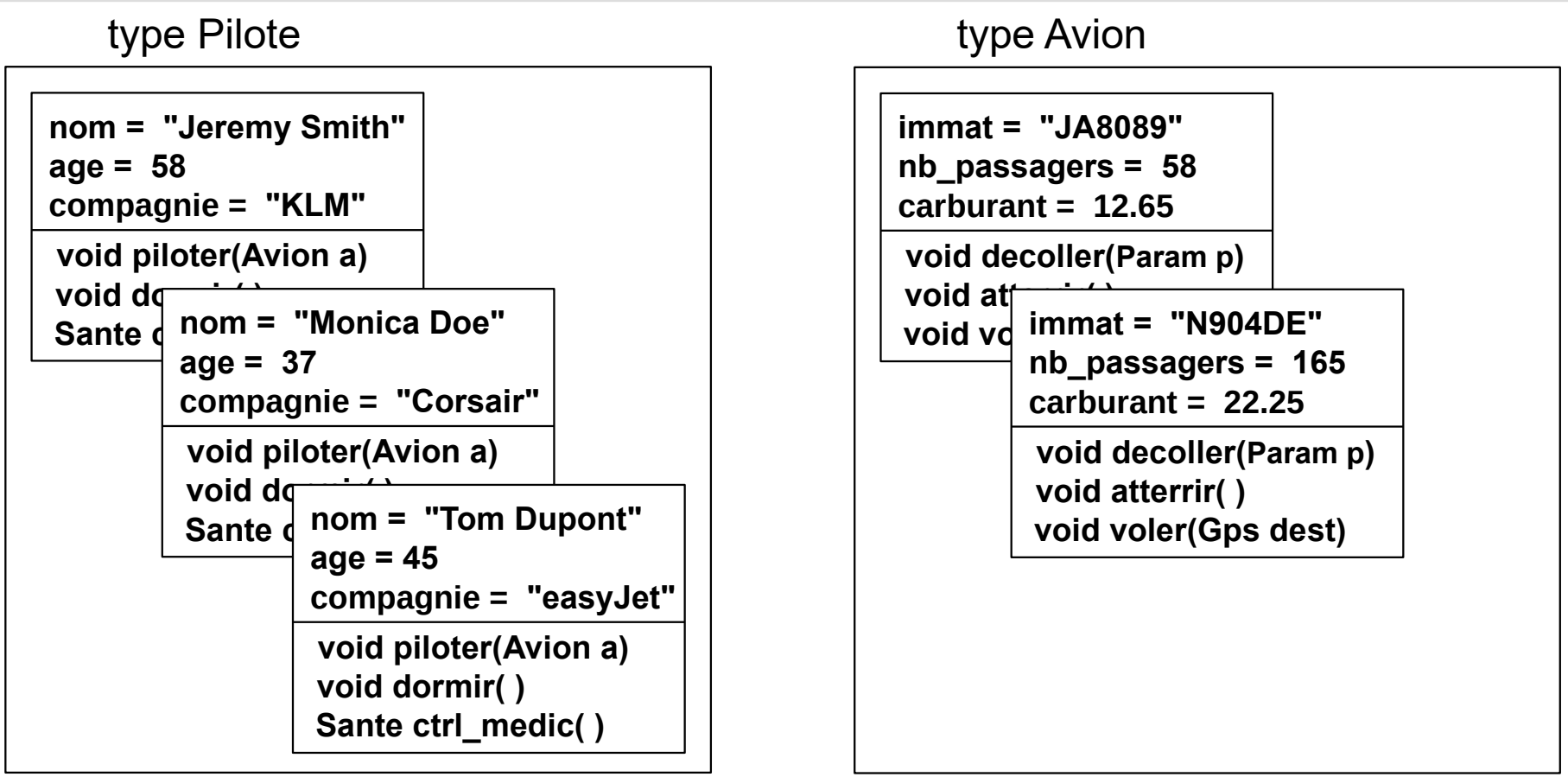


Programmation Orientée Objet

Comment articuler la relation étroite entre :

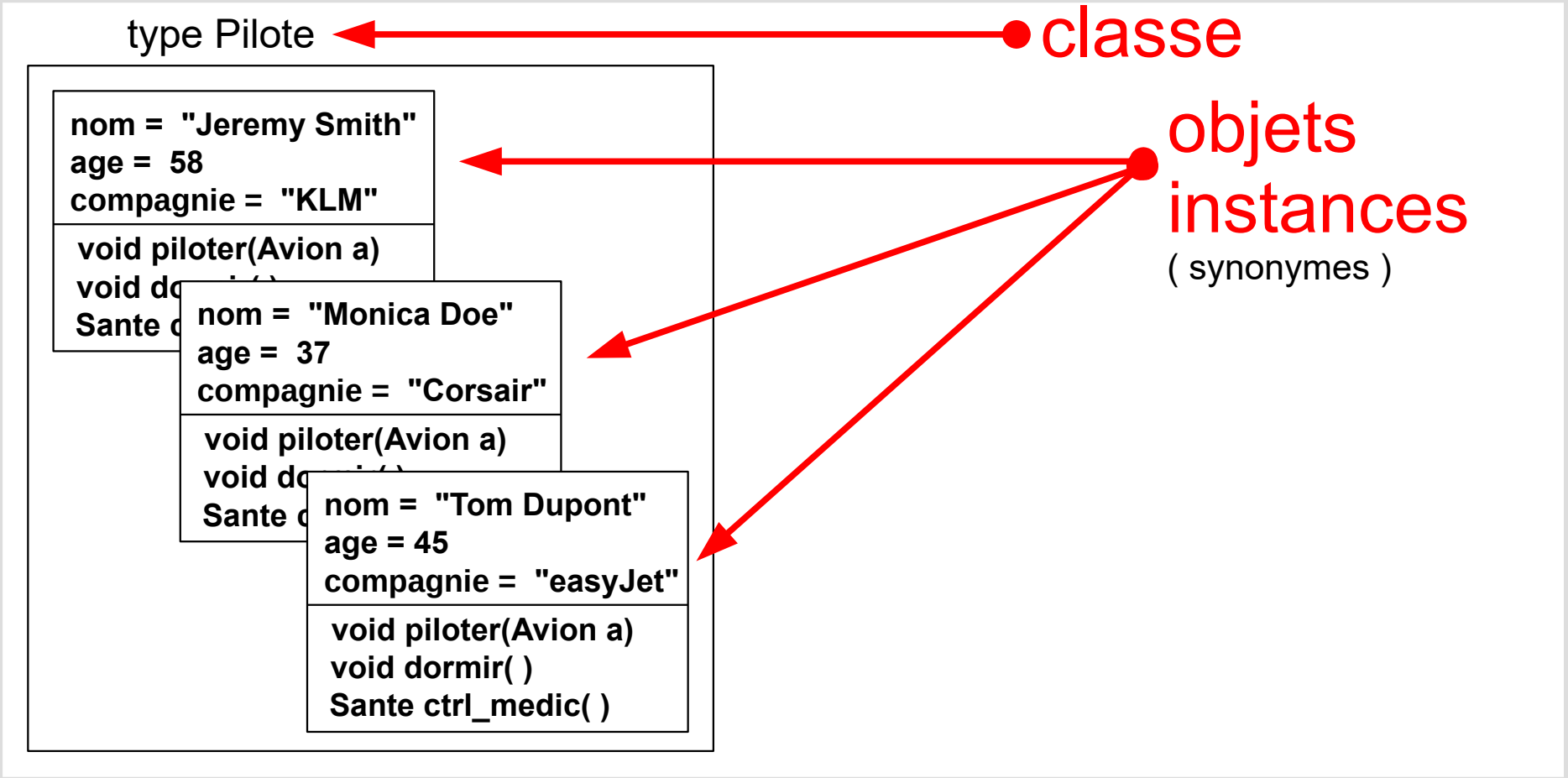
*→ les différents **types** de blocs de données*

→ les traitements qui leur sont associés



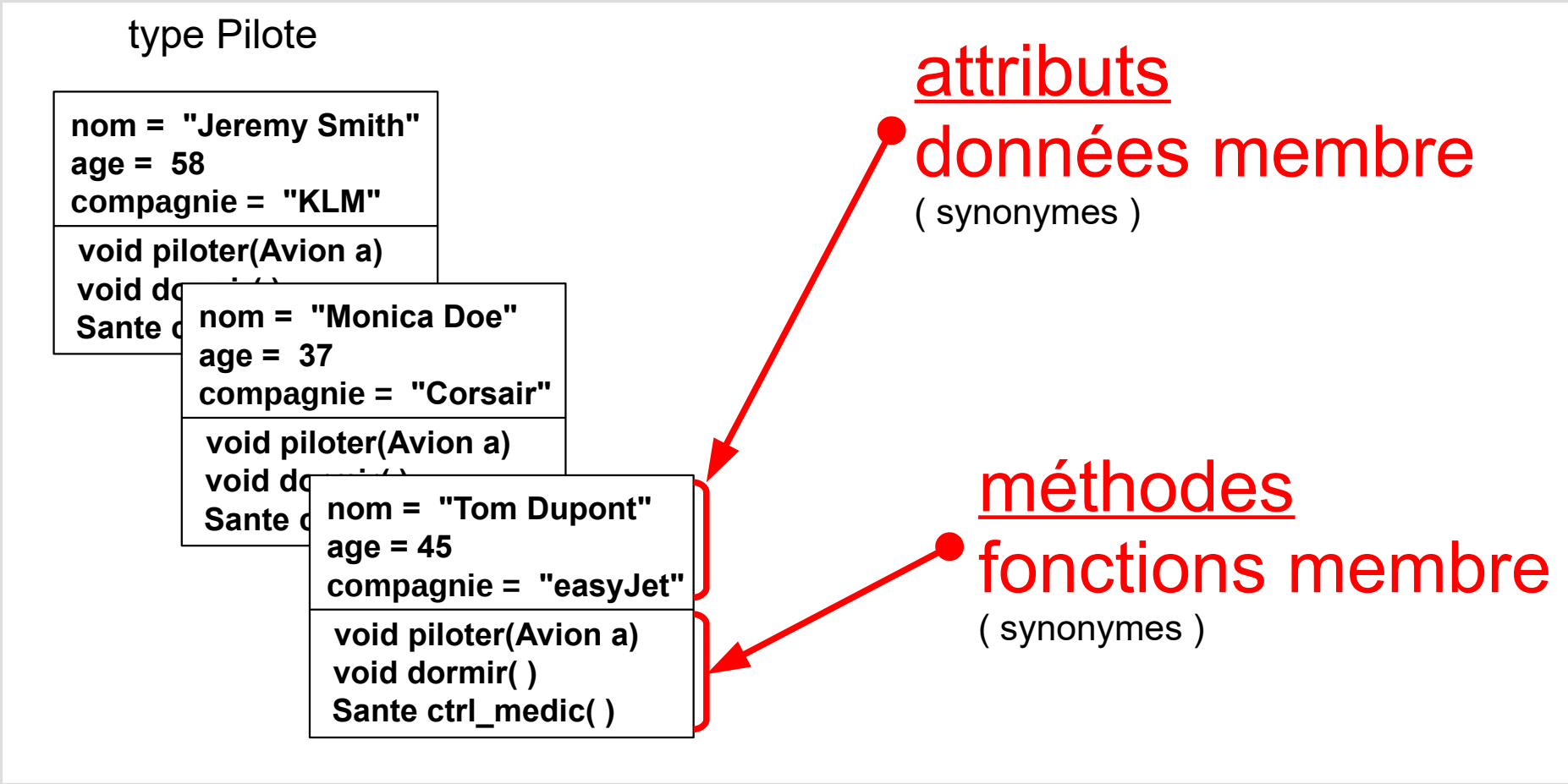
Programmation Orientée Objet

- *Un type structuré définit une **classe***
- *Les entités concrètes de cette classes sont des **objets** ou **instances** de la classe*



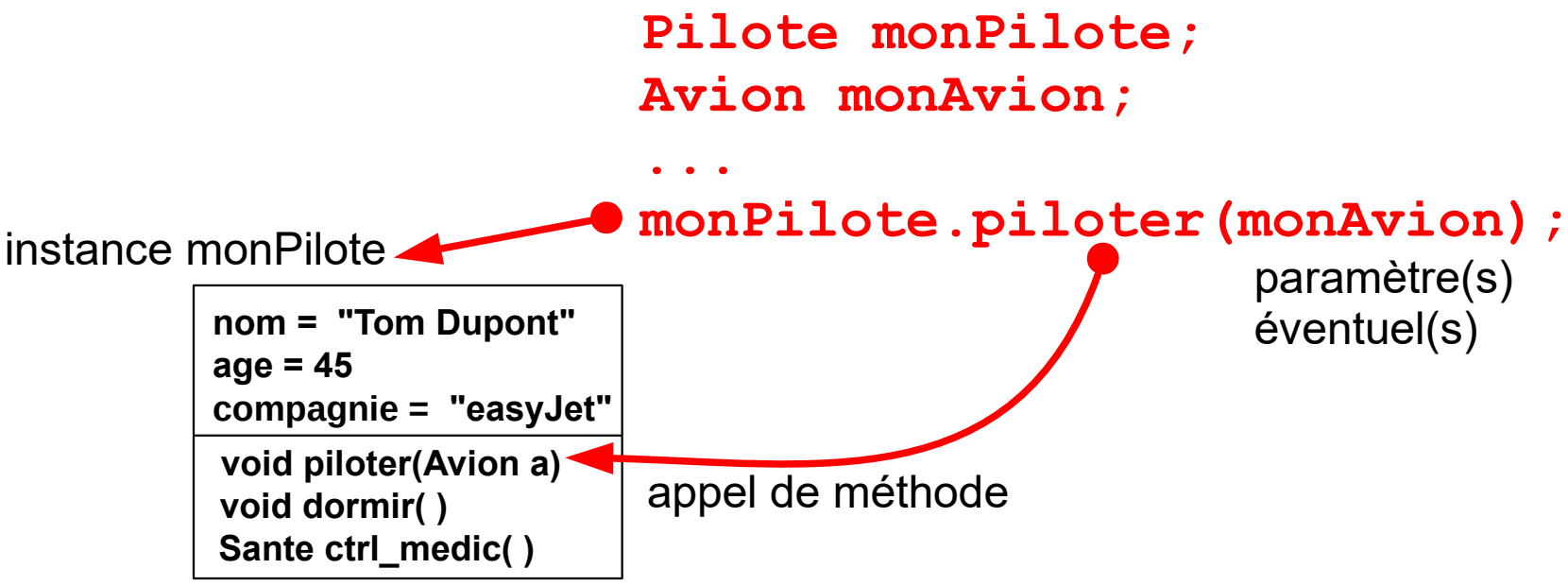
Programmation Orientée Objet

- *Les instances d'une même classe ont*
  - → *même structure de **données** (mais valeurs spécifiques)*
  - → *même ensemble de **fonctions** possibles*



Programmation Orientée Objet

- *Le déclenchement d'un traitement passe par l'**appel** d'une méthode en partant d'une **instance spécifique***
- *L'action est centrée sur les valeurs spécifiques de cette instance (cet objet spécifique agit)*



Programmation Orientée Objet

<u>Une méthode de conception « orientée objet »</u>

à partir de l'analyse du CDC et des exemples de cas d'usages

→ *repérer des exemples d'instances*
Noms propres, entités concrètes

→ *identifier les classes (les catégories d'entités)*
Noms communs, groupes d'entités concrètes homogènes

→ *leurs attributs (ce qui qualifie ces entités)*
Adjectifs, quantités, énumérations de valeurs possibles

→ *leurs méthodes (fonctions / traitements / actions)*
Verbes, formes verbales

Programmation Orientée Objet

<u>Une méthode de conception « orientée objet »</u>

*On s'efforcera d'**encapsuler** les attributs*

→ *Le code client devra pouvoir **utiliser** les objets sans accéder directement aux données membres...*

→ *Les méthodes constituent l'**interface** càd. la façon normale d'utiliser les objets*

*L'objectif est de découpler le code client (**appelant**) des "détails" internes (**appelé**)*

Programmation Orientée Objet

Une méthode de conception « orientée objet »

Exemple :

*Un objet (de la classe) **Avion** comporte de nombreuses données membre techniques*
(poids embarqué, carburant aile gauche, pression hydraulique frein droit…)

*Un objet (de la classe) **TourDeControle** n'a pas vocation à mettre son nez dans tous ces détails, elle va interagir avec un objet avion avec un **appel** à la **méthode***

```
bool estPretAuDecollage()
```

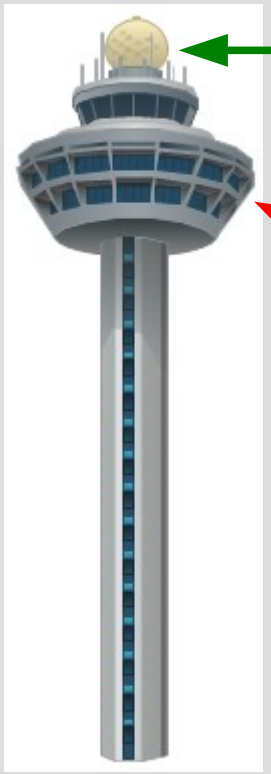
qui retourne un indicateur booléen Oui/Non à partir des données internes à l'objet.

Programmation Orientée Objet

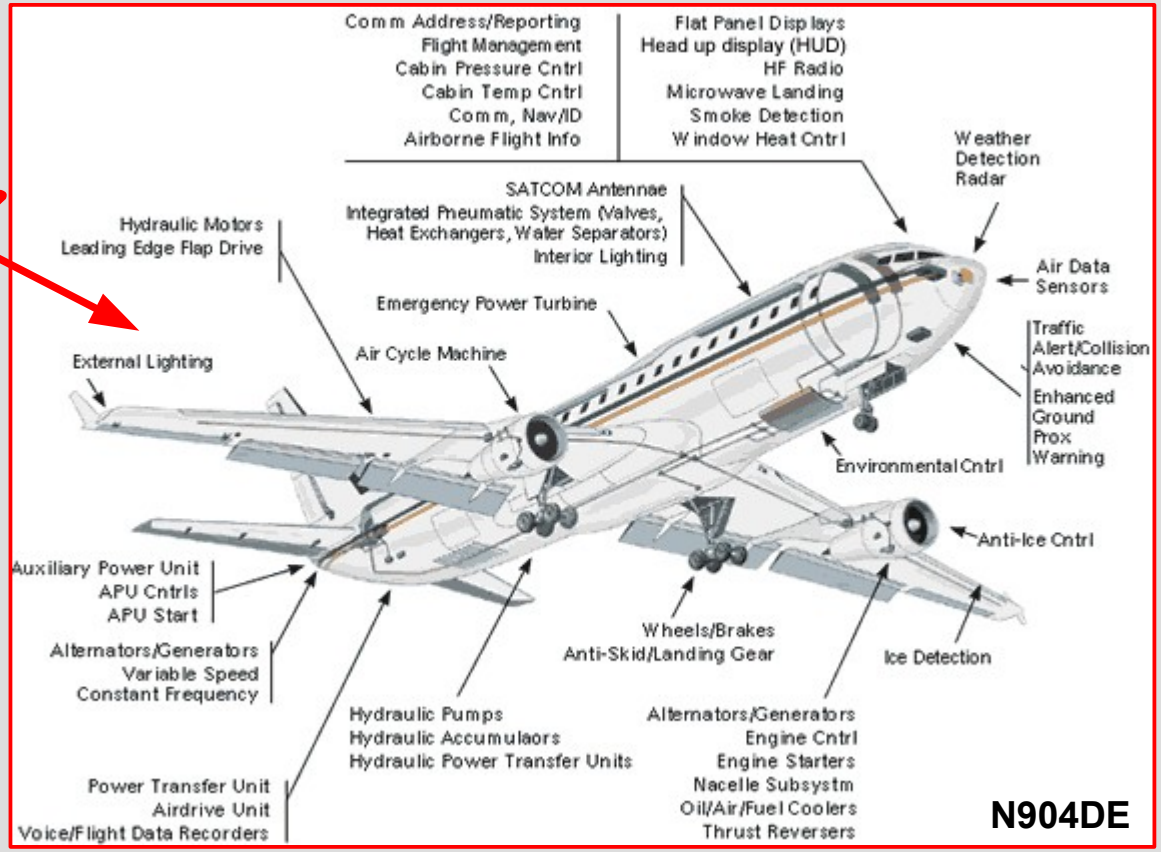
Une méthode de conception « orientée objet »

Interface

```
bool estPretAuDecollage()
void autoriserDecollage(Piste p)
XYZ  getPosition()
```



ORLY-TDC3



Implémentation

Programmation Orientée Objet

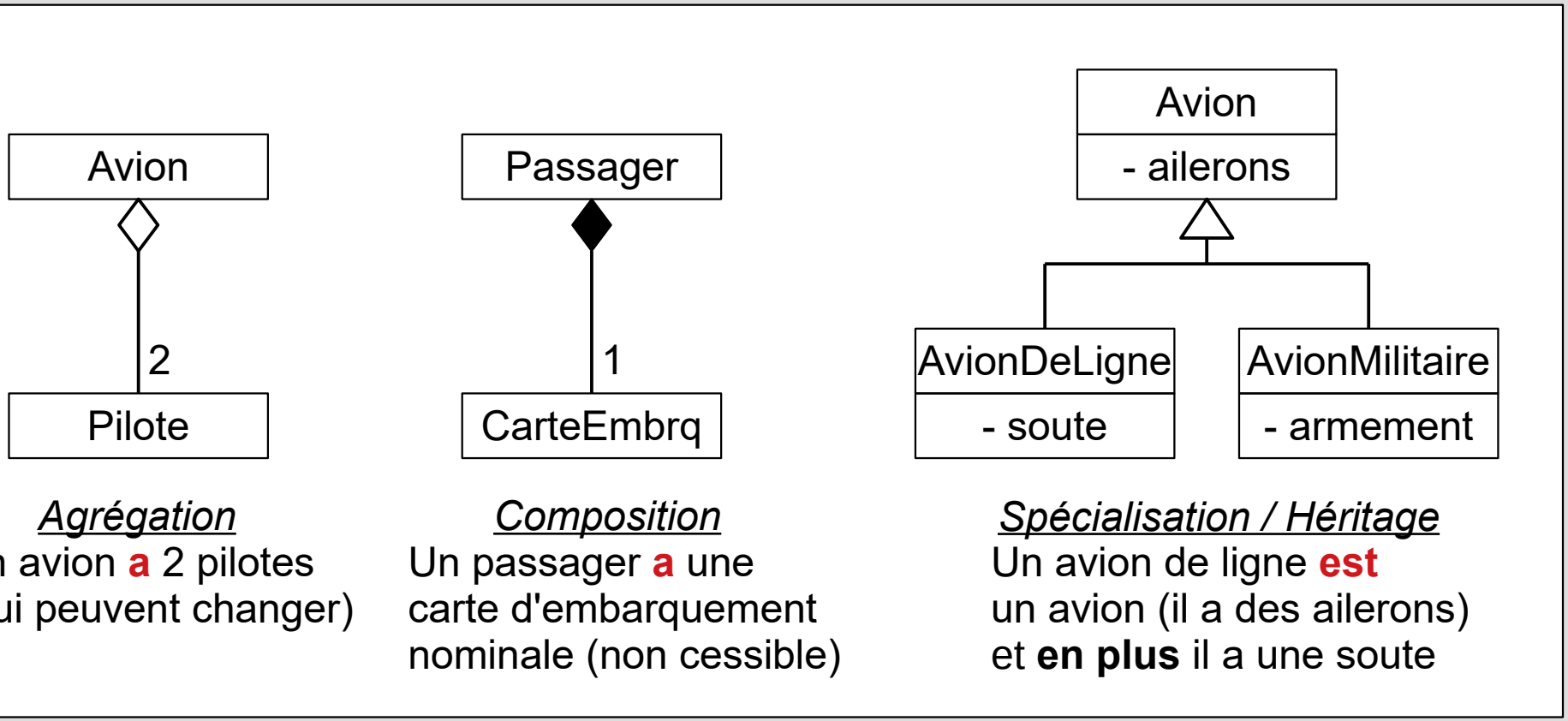
Une méthode de conception « orientée objet »

- *L'objet lui même est le mieux placé pour gérer ses propres données et proposer à l'utilisateur des possibilités claires et circonscrites*
- *L'interface à vocation à rester **stable***
- *L'implémentation peut **évoluer** sans casser le protocole d'utilisation de l'objet (le mode d'emploi reste le même) et donc sans casser le code utilisateur (code client)*

Programmation Orientée Objet

Une méthode de conception « orientée objet »

*Ensuite/conjointement on définira les **relations** entre les (objets des différentes) classes.*



Agrégation
Un avion **a** 2 pilotes
(qui peuvent changer)

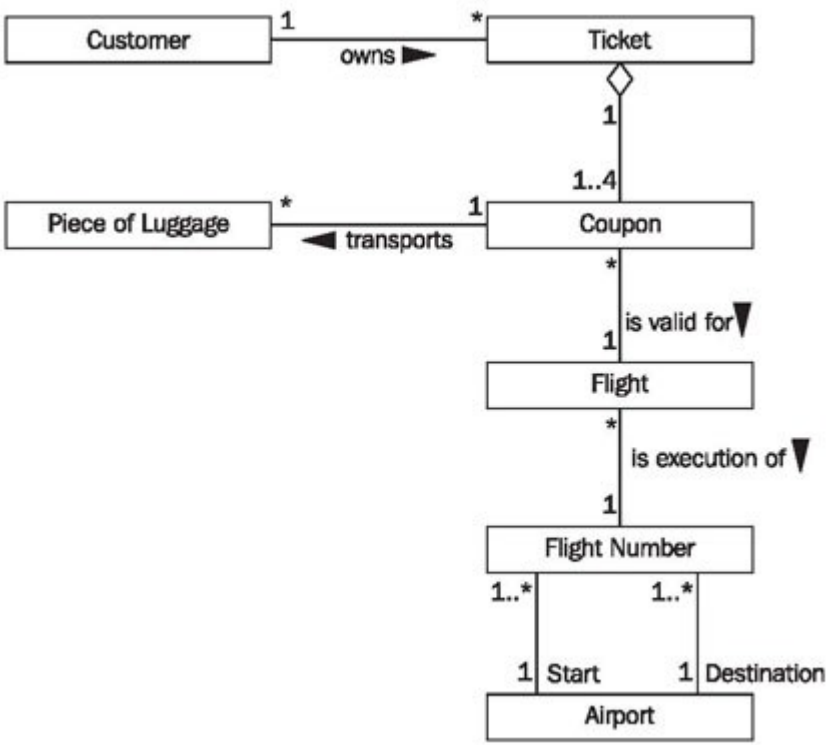
Composition
Un passager **a** une
carte d'embarquement
nominale (non cessible)

Spécialisation / Héritage
Un avion de ligne **est**
un avion (il a des ailerons)
et **en plus** il a une soute

Programmation Orientée Objet

Une méthode de conception « orientée objet »

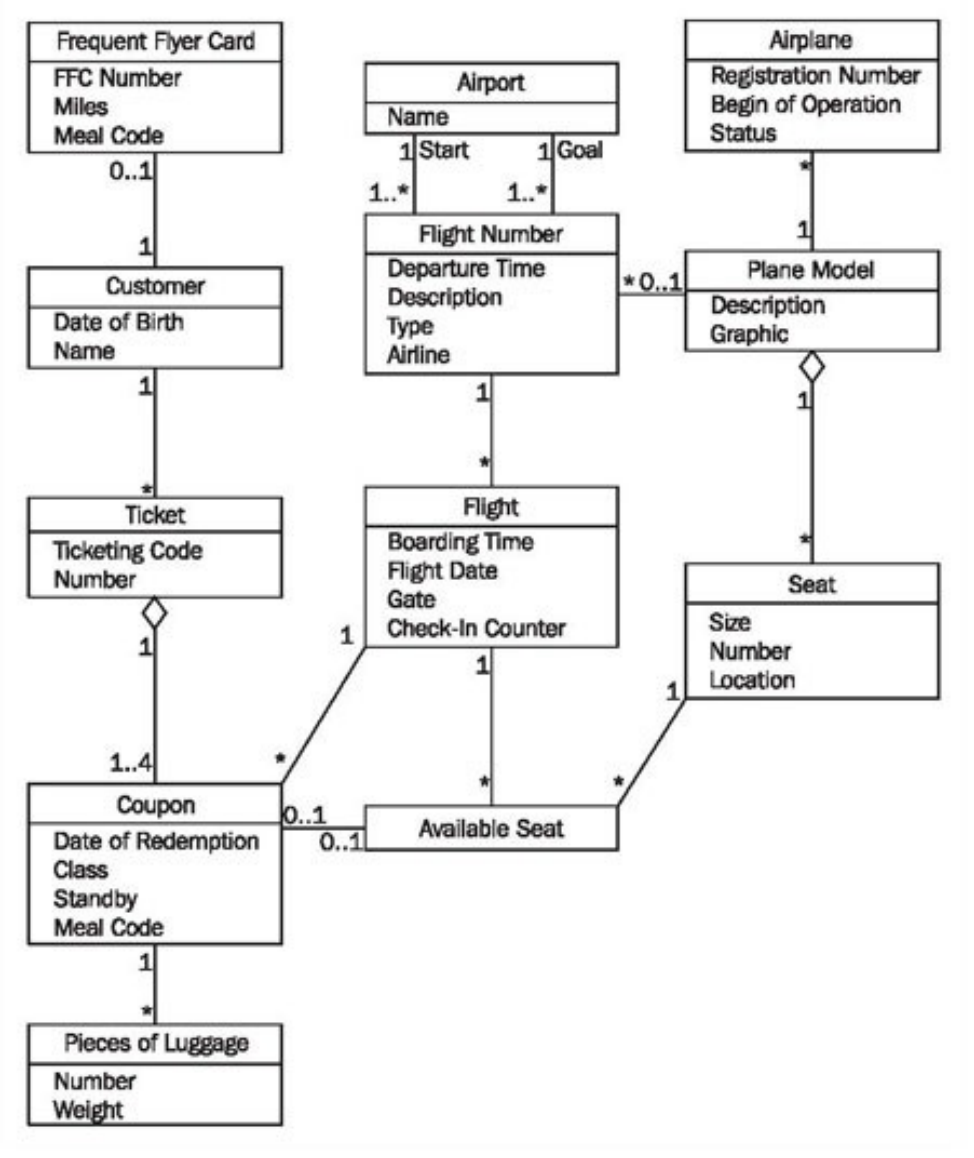
*Cette étude aboutit à la mise en place d'un **modèle objet** du projet avec **diagramme(s) de classes**. (notation normalisée **UML** → cours 2)*



source

Programmation Orientée Objet

Une méthode de conception « orientée objet »



*On ne connaîtra pas tout de suite tous les aspects C++ pour **implémenter** des diagrammes complexes*

diagramme de classes d'un système d'information **"service passagers"**

source

Programmation Orientée Objet

Une méthode de conception « orientée objet »

- *Toute cette phase de conception / architecture logicielle se fait indépendamment du(des) langage(s) d'implémentation (codage)*
- ***Pour programmer en C++ il faut programmer orienté objet donc connaître ces méthodes de conception en amont du code***
- *En tant que programmeurs de « C avec structs » vous avez déjà pris un bon départ :*
  *la **class** du C++ dérive directement de*
  *la **struct** du C !*

COURS 1

A) Présentation C++ / contexte
B) Programmation Orientée Objet
C) Du C au C++ sur un exemple

Du C au C++ sur un exemple

Un problème classique : gestion de collection d'entités homogènes → ajouter / voir / modifier

```
0 comptes :

0 : quitter
1 : ajouter un compte
2 : ajouter un compte avec cadeau
3 : crediter un compte
4 : debiter un compte

choix menu : 1↵
titulaire : Lucien↵
Creation du compte Lucien
```

Du C au C++ sur un exemple

Un problème classique : gestion de collection d'entités homogènes → ajouter / voir / modifier

```
1 comptes :
 1 Titulaire Lucien     Solde 0.00

0 : quitter
1 : ajouter un compte
2 : ajouter un compte avec cadeau
3 : crediter un compte
4 : debiter un compte

choix menu : 2↵
titulaire : Alexia↵
montant du cadeau : 45↵
Creation du compte Alexia
```

Du C au C++ sur un exemple

Un problème classique : gestion de collection d'entités homogènes → ajouter / voir / modifier

```
2 comptes :
 1 Titulaire Lucien     Solde 0.00
 2 Titulaire Alexia     Solde 45.00

0 : quitter
1 : ajouter un compte
2 : ajouter un compte avec cadeau
3 : crediter un compte
4 : debiter un compte

choix menu : 3↵
compte numero : 1↵
montant a crediter : 120.50↵
```

Du C au C++ sur un exemple

Un problème classique : gestion de collection d'entités homogènes → ajouter / voir / modifier

```
2 comptes :
 1 Titulaire Lucien     Solde 120.50
 2 Titulaire Alexia     Solde 45.00

0 : quitter
1 : ajouter un compte
2 : ajouter un compte avec cadeau
3 : crediter un compte
4 : debiter un compte

choix menu : 4↵
compte numero : 2↵
montant a debiter : 50↵
provisions insuffisantes Alexia !
```

Du C au C++ sur un exemple

Un problème classique : gestion de collection d'entités homogènes → ajouter / voir / modifier

```
2 comptes :
 1 Titulaire Lucien     Solde 120.50
 2 Titulaire Alexia     Solde 45.00

0 : quitter
1 : ajouter un compte
2 : ajouter un compte avec cadeau
3 : crediter un compte
4 : debiter un compte

choix menu : 4↵
compte numero : 2↵
montant a debiter : 40↵
```

Du C au C++ sur un exemple

Un problème classique : gestion de collection d'entités homogènes → ajouter / voir / modifier

```
2 comptes :
 1 Titulaire Lucien     Solde 120.50
 2 Titulaire Alexia     Solde 5.00

0 : quitter
1 : ajouter un compte
2 : ajouter un compte avec cadeau
3 : crediter un compte
4 : debiter un compte

choix menu : 0↵
Liberation du compte Lucien
Liberation du compte Alexia

Process returned 0 (0x0)
```

Du C au C++ sur un exemple

En termes de conception orientée objets
on identifie immédiatement une classe centrale :
*la **classe** Compte, avec 2 **attributs***

- *titulaire : chaîne de caractères*
- *solde : une valeur flottante*

*et 5 **méthodes***

- *Créer un compte avec solde initial paramétrable*
- *Libérer (la mémoire d') un compte*
- *Afficher un compte (pour l'affichage de la liste)*
- *Créditer un compte avec crédit en paramètre*
- *Débiter un compte avec débit en paramètre*

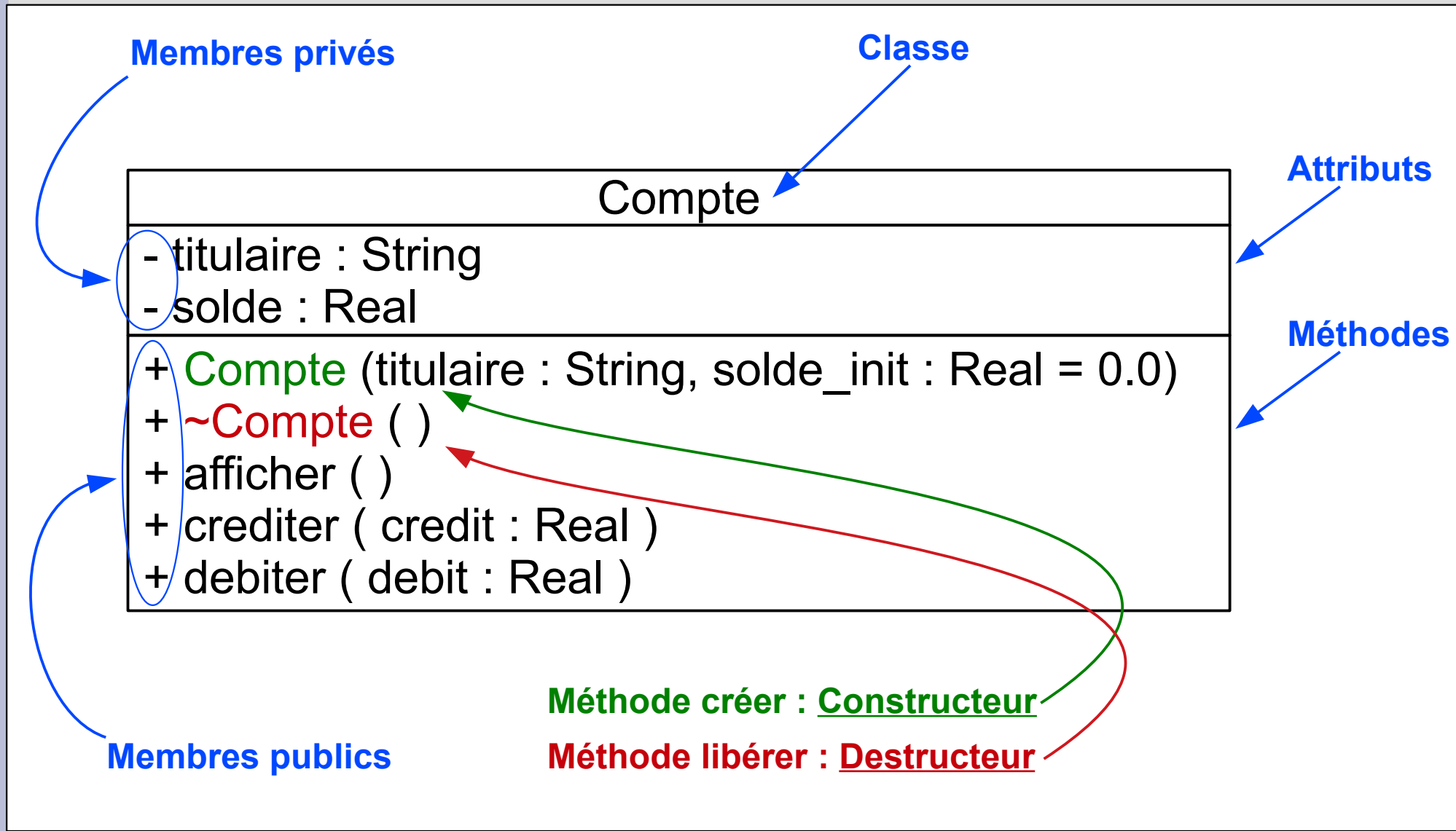
Du C au C++ sur un exemple

*La **classe** Compte en notation UML normalisée*

Compte
- titulaire : String - solde : Real
+ Compte (titulaire : String, solde_init : Real = 0.0) + ~Compte ( ) + afficher ( ) + crediter (credit : Real) + debiter (debit : Real)

Du C au C++ sur un exemple

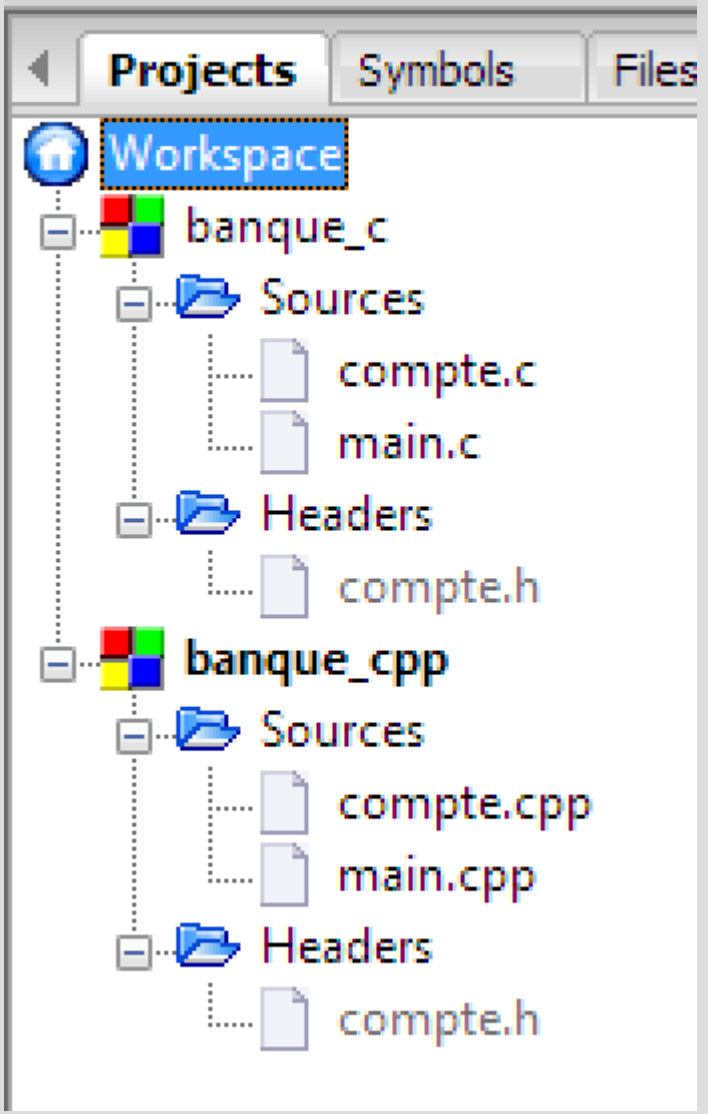
*La **classe** Compte en notation UML normalisée*



Du C au C++ sur un exemple

En C comme en C++

- *~ même nbr de lignes 150*
- *même IDE Code::Blocks*
- *même compilateur GCC*
- *même distinction fichiers d'en-tête (.h) et fichiers d'implémentation (.c → .cpp)*
- *même découpage de projet*



Voyons ++ en détail
ce qui change ...

Du C au C++ sur un exemple

*En **C** d'un côté une struct, de l'autre des sous-programmes qui reçoivent cette struct en param.*

compte.h

```
/// Définition d'un type "compte en banque"
typedef struct compte
{
    char *titulaire;    // Nom du titulaire
    float solde;        // Montant actuel
}
t_compte;

/// Déclaration des traitements associés au type
t_compte * compteCreer(char *titu);
t_compte * compteCreerAvecSolde(char *titu, float solde_init);
void compteLiberer(t_compte * compte);
void compteAfficher(t_compte * compte);
void compteCrediter(t_compte * compte, float credit);
int compteDebiter(t_compte * compte, float debit);
```

Noter la redondance : tous les sous-progs associés au type t_compte prennent un même 1er paramètre

Du C au C++ sur un exemple

*En **C++** la classe groupe les données (attributs) et traitements (méthodes) d'un même type d'objets*

compte.h

Seules les méthodes de l'objet ont accès aux données internes déclarées « private »

```
/// Définition d'un type "compte en banque"
class Compte
{
    /// Attributs (données associées à un objet)
    private :
        std::string m_titulaire;    // Nom du titulaire
        float m_solde;              // Montant actuel


    /// Méthodes (déclarations des traitements associés)
    public :
        Compte(std::string _titulaire, float _solde_init=0.0f);
        ~Compte();
        void afficher() const;
        void crediter(float _credit);
        void debiter(float _debit);
        std::string getTitulaire() const;
};
```

Du C au C++ sur un exemple

*En **C++** les méthodes (sous-progs. associés à une classe) reçoivent automatiquement l'objet*

compte.h

```
/// Définition d'un type "compte en banque"
class Compte
{
    /// Attributs (données associées à un objet)
    private :
        std::string m_titulaire;    // Nom du titulaire
        float m_solde;              // Montant actuel


    /// Méthodes (déclarations des traitements associés)
    public :
Constructeur→ Compte(std::string _titulaire, float _solde_init=0.0f);
 Destructeur→ ~Compte();
        void afficher() const;
        void crediter(float _credit);
        void debiter(float _debit);
        std::string getTitulaire() const;
};
```

Valeur par défaut d'un paramètre

L'objet de type Compte (cible du traitement) n'est plus mentionné explicitement : il est transmis **implicitement** à la méthode

Accesseur en lecture, oublié dans l'analyse UML mais nécessaire pour permettre au code client d'afficher le titulaire lors d'un débit à découvert...

Du C au C++ sur un exemple

*En **C++** on ne plaisante pas avec les types mais on a des commodités : enfin des chaînes pratiques*

compte.h

```
/// Définition d'un type "compte en banque"
class Compte
{
    /// Attributs (données associées à un objet)
    private :
        std::string m_titulaire;    // Nom du titulaire
        float m_solde;              // Montant actuel


    /// Méthodes (déclarations des traitements associés)
    public :
        Compte(std::string _titulaire, float _solde_init=0.0f);
        ~Compte();
        void afficher() const;
        void crediter(float _credit);
        void debiter(float _debit);
        std::string getTitulaire() const;
};
```

Le type string de la bibliothèque standard gère tout seul des chaînes de taille variable !

float ≠ double

Ni un affichage ni la récupération du nom d'un titulaire ne doivent modifier l'objet sur lequel porte l'opération : on déclare que l'objet y restera constant

On peut retourner une chaîne aussi simplement qu'un vulgaire int ! *(sémantique par valeur)*

Du C au C++ sur un exemple

*En **C** le fichier .c donne l'**implémentation** des sous-programmes déclarés dans l'**interface** .h*

compte.c

```c
#include "compte.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/// Définition des traitements associés au type

t_compte * compteCreer(char *titu)
{ ... }
t_compte * compteCreerAvecSolde(char *titu, float solde_init)
{ ... }
void compteLiberer(t_compte * compte)
{ ... }
void compteAfficher(t_compte * compte)
{ ... }
void compteCrediter(t_compte * compte, float credit)
{ ... }
int compteDebiter(t_compte * compte, float debit)
{ ... }
```

Du C au C++ sur un exemple

*En **C++** le fichier .cpp donne l'**implémentation** des méthodes de classe déclarées dans l'**interface** .h*

compte.cpp

```
#include "compte.h"

#include <iostream>
#include <string>
#include <stdexcept>

/// Méthodes (définitions des traitements associés)

Compte::Compte(std::string _titulaire, float _solde_init)
{ ... }
Compte::~Compte()
{ ... }
void Compte::afficher() const
{ ... }
void Compte::crediter(float _credit)
{ ... }
void Compte::debiter(float _debit)
{ ... }
std::string Compte::getTitulaire() const
{ ... }
```

Opérateur de résolution de portée :
on parle de la classe string de la bibliothèque standard

Opérateur de résolution de portée :
on parle bien de la méthode afficher de la classe Compte

Du C au C++ sur un exemple

*En **C** il faut allouer explicitement les structs qui doivent « survivre » à l'appel d'un sous-prog.*

compte.c

```
/// Constructeurs d'objet de type compte
t_compte * compteCreer(char *titu)
{
    return compteCreerAvecSolde(titu, 0);
}

t_compte * compteCreerAvecSolde(char *titu, float solde_init)
{
    // Pointeur sur et allocation d'un nouveau compte
    t_compte * compte;
    compte = (t_compte *)malloc( 1*sizeof(t_compte) );

    // Initialisation des données
    compte->titulaire = (char *)malloc( (strlen(titu)+1) * sizeof(char) );
    strcpy(compte->titulaire, titu);
    compte->solde = solde_init;

    // Retour à l'appelant du compte alloué & initialisé
    return compte;
}
```

Du C au C++ sur un exemple

*En **C++** la méthode constructeur ne gère pas explicitement l'allocation de son propre espace*

compte.cpp

```
/// Constructeur d'objet de type compte (avec ou sans solde_init)
Compte::Compte(std::string _titulaire, float _solde_init)
{
    m_titulaire = _titulaire;
    m_solde = _solde_init;
}
```

Pas d'allocation explicite de l'objet ici (voir appelant...)
Pas d'allocation explicite des attributs
qui ont une « sémantique par valeur »

compte.h

```
class Compte
{
    private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string _titulaire, float _solde_init=0.0f);
        ...
```

Du C au C++ sur un exemple

*En **C++** dans la méthode d'un objet on accède aux attributs de celui-ci sans l'expliciter*

compte.cpp

```
/// Constructeur d'objet de type compte (avec ou sans solde_init)
Compte::Compte(std::string _titulaire, float _solde_init)
{
    m_titulaire = _titulaire;
    m_solde = _solde_init;
}
```

copie de la chaîne en paramètre dans l'attribut
(les chaînes strings s'utilisent comme des scalaires !)

on écrit directement l'attribut de l'objet (l'objet est implicite)

pour éviter les confusion on peut par convention
préfixer les **données membre** par **m_**
préfixer les **paramètres** par **_**

compte.h

```
class Compte
{
    private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string _titulaire, float _solde_init=0.0f);
        ...
```

Du C au C++ sur un exemple

*En **C** ce qui a été alloué explicitement dans le constructeur doit être libéré explicitement*

compte.c

```
/// Destructeur d'objet de type compte
void compteLiberer(t_compte * compte)
{
    // Le champ titulaire a été alloué -> libération
    free(compte->titulaire);

    // L'objet lui même a été alloué -> libération
    free(compte);
}
```

Du C au C++ sur un exemple

*En **C++** aussi ! Mais souvent le constructeur ne fait aucune allocation explicite (pas besoin) ...*

compte.cpp

```
/// Destructeur d'objet de type compte
Compte::~Compte()
{
    // Rien à faire ici
    // ( car aucune allocation explicite dans le constructeur )
}
```

Du C au C++ sur un exemple

*En **C** les entrées/sorties consoles utilisent des fonctions format-typées printf et scanf de stdio.h*

compte.c

```
/// Opération d'affichage d'un objet de type compte
void compteAfficher(t_compte * compte)
{
    printf("Titulaire %s \tSolde %.02f\n",
               compte->titulaire, compte->solde);
}
```

Du C au C++ sur un exemple

*En **C++** les entrées/sorties consoles utilisent des flots chaînés std::cin et std::cout de iostream*

compte.cpp

```
/// Opération d'affichage d'un objet de type compte
void Compte::afficher() const
{
    std::cout << "Titulaire " << m_titulaire
              << " \tSolde " << m_solde << std::endl;
}
```

Du C au C++ sur un exemple

*En **C** les conditions d'erreur sont souvent retournées à l'appelant par valeur spéciale*

compte.c

```
/// Opération de créditer un objet de type compte
void compteCrediter(t_compte * compte, float credit)
{
    compte->solde += credit;
}

/// Opération de débiter un objet de type compte
// Valeur de retour == 0 indique un compte pas approvisionné
// ( dans ce cas le compte n'est pas débité ... )
int compteDebiter(t_compte * compte, float debit)
{
    if ( compte->solde - debit < 0.0 )
        return 0;

    compte->solde -= debit;
    return 1;
}
```

Du C au C++ sur un exemple

*En **C++** les conditions d'erreurs passent par un nouveau mécanisme, les **exceptions** ...*

compte.cpp

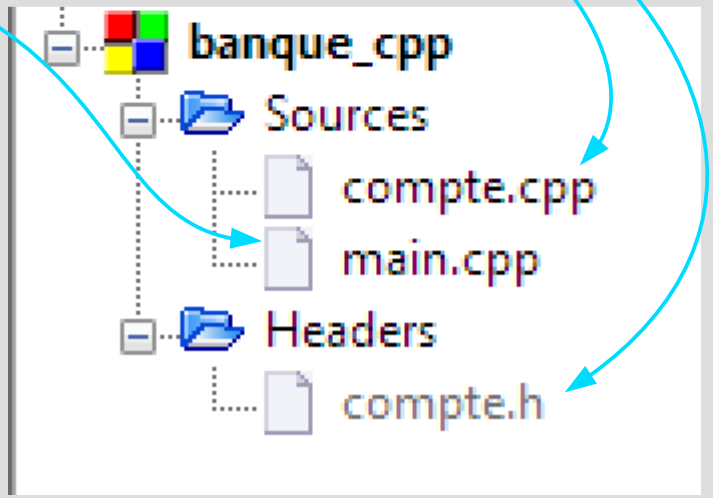
```
// Opération de créditer un objet de type compte
void Compte::crediter(float _credit)
{
    m_solde += _credit;
}

// Opération de débiter un objet de type compte
// Exception invalid_argument si le compte n'est pas approvisionné
// ( dans ce cas le compte n'est pas débité ... )
void Compte::debiter(float _debit)
{
    if ( m_solde - _debit < 0.0f )
        throw std::invalid_argument("provisions insuffisantes");

    m_solde -= _debit;
}
```

Anomalie : on **interrompt** l'exécution ici, on reprend au niveau de l'appelant, ou de l'appelant de l'appelant... jusqu'à trouver un niveau qui **déclare** savoir s'occuper de ce problème en ayant précisé un bloc **try** / **catch**

Du C au C++ sur un exemple

En « C objet » comme en C++
on distingue

- *l'interface d'une classe et l'implémentation d'une classe constituent le code **utilisé***
- *le code **utilisateur** du type ou code client ou appelant*



le développeur d'une classe doit faciliter le travail du développeur client de la classe : interface claire, stable, documentée, bien séparée de l'implémentation

Du C au C++ sur un exemple

*En **C** **code client** du main, ici on choisit d'utiliser un tableau de pointeurs sur structs*

main.c

```
// Utilisation de la "bibliothèque" gestion de compte
#include "compte.h"
#include <stdio.h>
...
/// Gestion de quelques comptes (moins de 50)
int main()
{
    /// La collection des (pointeurs sur) comptes
    /// Au démarrage il y a 0 compte
    /// Il y en aura 50 au plus
    t_compte * comptes[50] = {NULL};
    int nbComptes = 0;

    /// Variables auxiliaires (saisies...)
    int choix;
    char nom[100];
    float montant;
    int id;
    int debitOk;
```

alternatives en C :
- malloc/realloc
- liste chaînée

Du C au C++ sur un exemple

*En **C++** **code client** du main, ici on peut utiliser un conteneur standard : un **vecteur** de pointeurs...*

main.cpp

```
// Utilisation de la "bibliothèque" gestion de compte
#include "compte.h"

#include <iostream>
#include <string>
#include <vector>
...
/// Gestion comptes (quantité non limitée)
int main()
{
    /// La collection des (pointeurs sur) comptes
    /// Au démarrage il y a 0 compte
    /// vector est comme un tableau mais extensible...
    std::vector<Compte*> comptes;

    /// Variables auxiliaires (saisies...)
    int choix;
    std::string nom;
    float montant;
    size_t id;
```

Le vecteur reçoit le type « pointeur sur Compte » en paramètre, on utilise un **template** ...

Du C au C++ sur un exemple

*En **C** à chaque passage en paramètre de la collection il faut envoyer tableau et nombre d'élém.*

main.c

```
do  /// Boucle interactive de menu
{
    afficherComptes(comptes, nbComptes);
    afficherMenu();

    saisirEntierBorne("choix menu", &choix, 0, 4);

    switch(choix)
    {
        case 0:
            break;
        ...
    }
}
while (choix != 0);

libererComptes(comptes, nbComptes);

return 0;
}
```

on borne des entrées
entiers, flottants...
un sous-prog. par type

valeur paramètre modifiée par l'appel :
passage **par adresse** (syntaxe spécifique)

Du C au C++ sur un exemple

*En **C++** le passage en paramètre de la collection est référencé par le vecteur qui encapsule tout*

main.cpp

```
do  /// Boucle interactive de menu
{
    afficherComptes(comptes);
    afficherMenu();

    saisirBorne("choix menu", choix, 0, 4);

    switch(choix)
    {
        case 0:
            break;
        ...
    }
}
while (choix != 0);

libererComptes(comptes);

return 0;
}
```

déduction auto du type à borner un seul template

valeur paramètre modifiée par l'appel : passage **par référence** (syntaxe spécifique seulement pour le code appelé, rien dans le code appelant)

Du C au C++ sur un exemple

*En **C** l'ajout d'un élément à la collection est simple si on ne gère pas la quantité limitée [50] !*

main.c

```
// Ajouter un compte (! pas de gestion 50 comptes max !)
case 1:
    saisirMotBorne("titulaire", nom, 'A', 'Z');
    comptes[nbComptes++] = compteCreer(nom);
    break;

// Ajouter un compte avec cadeau (! idem cas précédent !)
case 2:
    saisirMotBorne("titulaire", nom, 'A', 'Z');
    saisirFlottantBorne("montant du cadeau", &montant, 0.10, 55.90);
    comptes[nbComptes++] = compteCreerAvecSolde(nom, montant);
    break;
```

Du C au C++ sur un exemple

*En **C++** l'ajout d'un élément à la collection est simple et la quantité « illimitée » (mémoire vive...)*

main.cpp

```
// Ajouter un compte (pas de limite nombre comptes)
case 1:
    saisirBorne<std::string>("titulaire", nom, "A", "ZZZ" );
    comptes.push_back( new Compte(nom) );
    break;

// Ajouter un compte avec cadeau (pas de limite nombre comptes)
case 2:
    saisirBorne<std::string>("titulaire", nom, "A", "ZZZ" );
    saisirBorne("montant du cadeau", montant, 0.10f, 55.90f);
    comptes.push_back( new Compte(nom, montant) );
    break;
```

l'allocation dynamique de l'objet
ne se fait pas dans le constructeur
mais au niveau de l'appelant
l'opérateur **new** remplace la fonction **malloc**

Du C au C++ sur un exemple

*En **C** l'appel à un traitement de l'objet passe l'objet en paramètre*

main.c

```
// Créditer un compte
case 3:
    saisirEntierBorne("compte numero", &id, 1, nbComptes);
    id--;
    saisirFlottantBorne("montant a crediter", &montant, 0, FLT_MAX);
    compteCrediter(comptes[id], montant);
    break;
```

Du C au C++ sur un exemple

*En **C++** l'appel à un traitement de l'objet part de l'objet, l'objet n'est pas dans les paramètres*

main.cpp

le vecteur est un objet, on récupère le nombre d'élément qu'il contient en utilisant sa méthode size()

```
// Créditer un compte
case 3:
    saisirBorne("compte numero", id, 1u, comptes.size());
    --id;
    saisirBorne("montant a crediter", montant, 0.0f);
    comptes[id]->crediter(montant);
    break;
```

on part de l'objet pour un appel de méthode

compte[id] est l'adresse d'un Compte donc on utilise -> au lieu de .
(même règle que accès champs struct)

on accède au i^{ème} élément d'un vecteur comme pour un tableau usuel

coquetterie sans grande importance
on préférera --compteur à compteur--
idem pour les incréments : ++i plutôt que i++

Du C au C++ sur un exemple

*En **C** la gestion d'une anomalie dans le sous-prog appelé passe par le contrôle d'un code retour...*

main.c

```
// Débiter un compte
case 4:
    saisirEntierBorne("compte numero", &id, 1, nbComptes);
    id--;
    saisirFlottantBorne("montant a debiter", &montant, 0, FLT_MAX);
    debitOk = compteDebiter(comptes[id], montant);
    if (!debitOk)
        printf("provisions insuffisantes %s !\n",
                                        comptes[id]->titulaire);
    break;
```

En C on s'autorise généralement à accéder directement aux valeurs des attributs d'un « objet » au niveau du code client...

Du C au C++ sur un exemple

*En **C++** l'appelé n'a pas besoin d'utiliser le canal return pour signaler un problème : exceptions !*

main.cpp

on essaye d'exécuter une séquence dans le bloc try, il peut y avoir un problème signalé par un appelé, l'appelé d'un appelé etc... avec un **throw** (voir slide 62)

```
// Débiter un compte
case 4:
    try
    {
        saisirBorne("compte numero", id, 1u, comptes.size());
        --id;
        saisirBorne("montant a debiter", montant, 0.0f);
        comptes[id]->debiter(montant);
        std::cout << "debit ok" << std::endl;
    }
    catch ( const std::invalid_argument& e )
    {
        std::cout << e.what() << " "
                  << comptes[id]->getTitulaire() << " !\n";
    }
    break;
```

on rattrape ici l'exécution si il y a eu une anomalie lors de l'exécution du bloc try

Du C au C++ sur un exemple

*En **C++** le respect du principe d'encapsulation est imposé par les attributs en private ...*

main.cpp

```
// Accesseur en lecture de l'attribut titulaire
std::string Compte::getTitulaire() const
{
    return m_titulaire;
}
```

compte.cpp

le **code client** ne peut pas accéder directement aux attribut, il doit utiliser un **accesseur** public !

```
catch ( const std::invalid_argument& e )
{
    std::cout << e.what() << " "
              << comptes[id]->getTitulaire() << " !\n";
}
break;
```

Du C au C++ sur un exemple

*En **C++** comme en C il est préférable d'anticiper les anomalies **avant** l'appel quand c'est possible !*

L'utilisation du mécanisme d'exception du code précédent était illustrative, mais il est déconseillé d'utiliser des exceptions pour gérer des cas de « business logic ». L'approche suivante est préférable :

main.cpp

```
case 4:
    saisirBorne("compte numero", id, 1u, comptes.size());
    --id;
    saisirBorne("montant a debiter", montant, 0.0f);
    if ( comptes[id]->debitable(montant) )
    {
        comptes[id]->debiter(montant);
        std::cout << "debit ok" << std::endl;
    }
    else
        std::cout << "provisions insuffisantes "
                  << comptes[id]->getTitulaire() << " !\n";
    break;
```

Attention cette solution n'est pas « thread safe » si l'objet est utilisé par plusieurs process son état peut changer entre la vérification et l'opération

méthode avec valeur de retour booléenne : l'objet est le mieux placé pour savoir si une opération le concernant est possible

Du C au C++ sur un exemple

*En **C** une procédure auxiliaire du main.c*

main.c

```
void afficherMenu()
{
    printf("0 : quitter\n");
    printf("1 : ajouter un compte\n");
    printf("2 : ajouter un compte avec cadeau\n");
    printf("3 : crediter un compte\n");
    printf("4 : debiter un compte\n");
    printf("\n");
}
```

Du C au C++ sur un exemple

*En **C++** une procédure auxiliaire du main.cpp*

main.cpp

```
void afficherMenu()
{
    std::cout << "0 : quitter" << std::endl;
    std::cout << "1 : ajouter un compte" << std::endl;
    std::cout << "2 : ajouter un compte avec cadeau" << std::endl;
    std::cout << "3 : crediter un compte" << std::endl;
    std::cout << "4 : debiter un compte" << std::endl;
    std::cout << std::endl;
}
```

end of line : équivalent à `"\n"`

Du C au C++ sur un exemple

*En **C** le parcours d'une collection dans un tableau*

main.c

```
void afficherComptes(t_compte * comptes[50], int nbComptes)
{
    int i;

    printf("\n\n%d comptes :\n", nbComptes);
    for (i=0; i<nbComptes; i++)
    {
        printf("%2d ", i+1);
        compteAfficher(comptes[i]);
    }
    printf("\n");
}
```

Du C au C++ sur un exemple

En C++ le parcours d'une collection dans un vecteur

main.cpp

passage par référence : on travaillera ici avec le vecteur de l'appelant et non pas une copie

```
void afficherComptes(const std::vector<Compte*>& comptes)
{
    std::cout << "\n\n" << comptes.size() << " comptes :\n";
    for (size_t i=0; i<comptes.size(); i++)
    {
        std::cout << i+1 << " ";
        comptes[i]->afficher();
    }
    std::cout << std::endl;
}
```

le type `size_t` est un « entier non signé »
on l'utilisera à la place de `int` pour les variables comparées à une size() (compteurs …)

hey ! On déclare le compteur dans la boucle for !

Du C au C++ sur un exemple

*En **C** le parcours d'une collection dans un tableau pour libérer les objets*

main.c

```
void libererComptes(t_compte * comptes[50], int nbComptes)
{
    int i;

    for (i=0; i<nbComptes; i++)
        compteLiberer(comptes[i]);
}
```

Du C au C++ sur un exemple

*En **C++** le parcours d'une collection dans un vecteur pour libérer les objets*

main.cpp

```
void libererComptes(std::vector<Compte*>& comptes)
{
    for (size_t i=0; i<comptes.size(); ++i)
        delete comptes[i];
}
```

en C++ l'opérateur **delete**
remplace la fonction **free**

la libération appelle implicitement
la méthode destructeur ~Compte

Du C au C++ sur un exemple

*En **C** 3 codes presque identiques avec des types distincts nécessitent 3 sous-progs différents*

main.c

```
void saisirEntierBorne(char *message, int *pe, int min, int max)
{
    printf("%s : ", message);
    scanf("%d", pe);
    while (*pe<min || *pe>max)
    {
        printf("Saisie incorrecte, recommencer : ");
        scanf("%d", pe);
    }
}
void saisirFlottantBorne(char *message, float *pf, float min, float max)
{
    printf("%s : ", message);
    scanf("%f", pf);
    while (*pf<min || *pf>max)
    {
        printf("Saisie incorrecte, recommencer : ");
        scanf("%f", pf);
    }
}
void saisirMotBorne(char *message, char *pc, char min, char max)
{
    printf("%s : ", message);
    scanf("%s", pc);
    while (*pc<min || *pc>max)
    {
        printf("Saisie incorrecte, recommencer : ");
        scanf("%s", pc);
    }
}
```

*Le C permettrait d'éviter cette répétition mais en utilisant des **macros** :*

- *code peu lisible*
- *piégeux*
- *typage non strict*

Du C au C++ sur un exemple

*En **C++** le mécanisme de **templates** permet de « paramétrer en fonction du type »*

main.cpp

```
template<typename T>
void saisirBorne(std::string message, T& res, T min, T max)
{
    std::cout << message << " : " ;
    std::cin >> res;
    while (res<min || res>max)
    {
        std::cout << "Saisie incorrecte, recommencer : " ;
        std::cin >> res;
    }
}
```

Du C au C++ sur un exemple

*En **C++** les **templates** permettent la programmation **générique** : même algo. indépendamment du type*

main.cpp

Prototypage du template avec valeur par défaut au dernier paramètre

```
template<typename T>
void saisirBorne(std::string message, T& res, T min,
                 T max=std::numeric_limits<T>::max());
```

C'est un « paradigme » de programmation qui fait partie des points forts du C++ mais qui est assez éloigné de ce qu'on connaissait :
on utilisera rapidement des templates en code client mais l'implémentation des templates sera vu à la fin

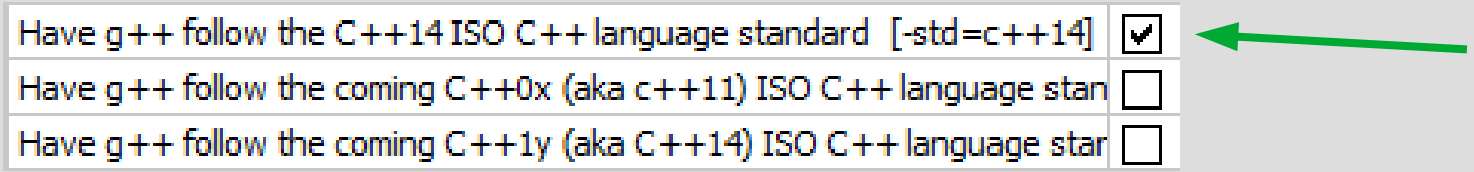
Du C au C++ sur un exemple

Les codes C et C++ de ce chapitre sont disponibles intégralement en 2 projets Code::Blocks

banque_c version C
banque_cpp version C++

https://fercoq.bitbucket.io/cpp/cours/cours1/banque.zip

Pour compiler les exemples de code sous Code::Blocks configurer C++14 dans menu déroulant → Settings → Compiler...



Du C au C++ sur un exemple

Ça fait beaucoup trop pour un 1er cours !

- *Pas d'inquiétude, il s'agissait d'un **survol** de 2 monuments à la fois :*
  - *conception objet / UML*
  - *C++*
- *Tous les concepts présentés seront détaillés lors des prochains cours et pratiqués en TD/TP*
- *Vos professeurs se feront un plaisir de répondre aux questions qui ne manqueront pas de se poser*