Ingénierie des Systèmes d'Information

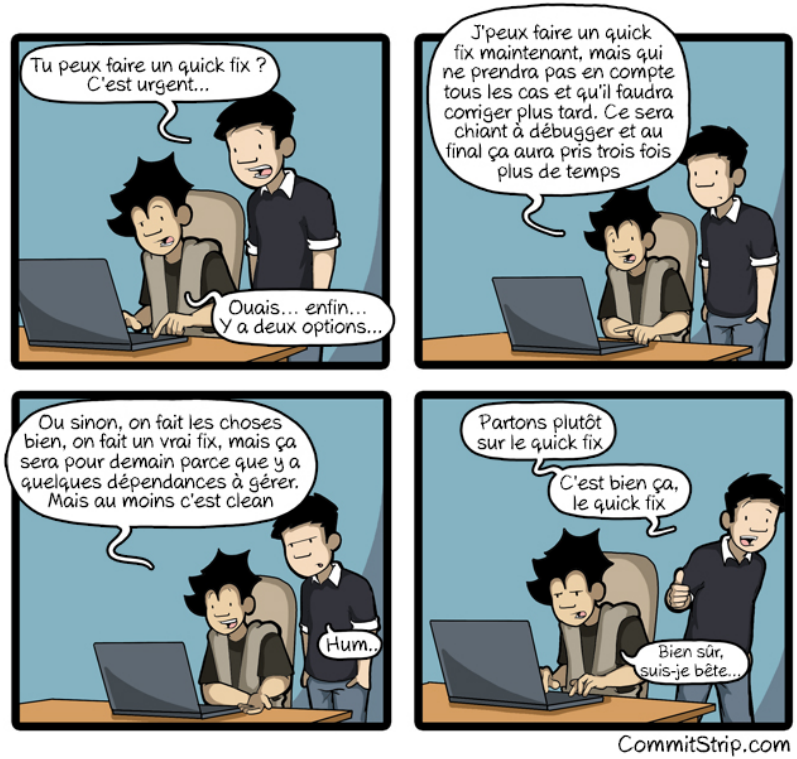
Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2019 - 2020

Objectifs

Objectifs

- Maîtriser différents aspects du développement d'un logiciel
- Validation des acquis en POO, UML et programmation Java
- Compréhension des principes avancés de conception orientée objet
- Compréhension de l'intérêt des design pattern informatiques
- Mise en pratique des design pattern (UML et Java)
- Compréhension de l'intérêt des tests unitaire et des tests fonctionnels



Programme

- Rappels UML
- Mécanismes de base et principes avancées de Conception Orientée Objet
- Patrons de conception
- Tests logiciels, Programmation fonctionnelle

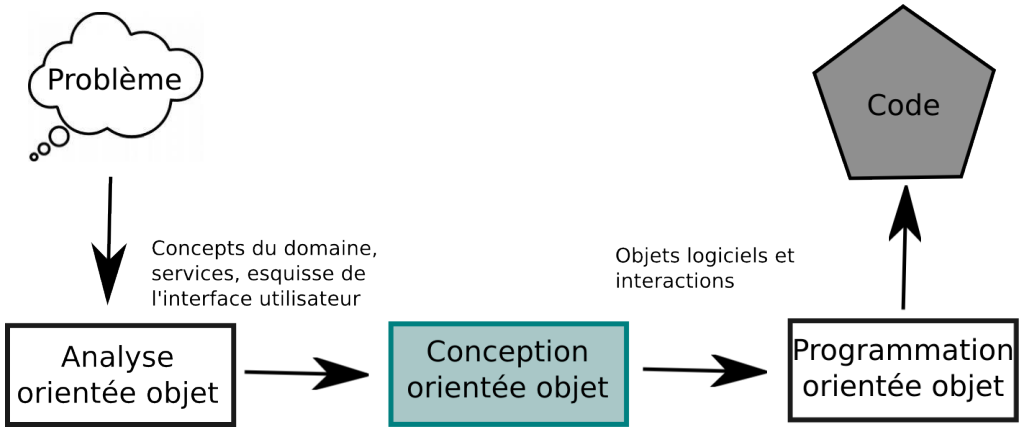
- liens vers cours, énoncés TDs : `https://projet.liris.cnrs.fr/sycosma/wiki/doku.php?id=isi3apprentis`
- Évaluation : notes de TPs

1. Concepts fondamentaux et Principes avancés de Conception Orientée Objet

Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2017 - 2018

Conception logicielle



Objectifs du cours

- Rappel des concepts UML relatifs à la vue structurelle
  - Préciser les différences sémantiques entre le langage UML (diagramme de classes) et les langages de POO (Java, C++)
  - Nécessaire pour *reverse engineering*
- Présenter les principes avancés de conception orientée objet
- Rappel sur les notions de couplage et dépendance

Plan

Qu'est-ce qu'UML ?

- UML = Unified Modeling Language
- UML = **Langage** universel pour la modélisation objet ... pas une méthode

Différence Langage – Méthode

- Langage de modélisation = notations, grammaire, sémantique
- Méthode = comment utiliser le langage de modélisation (recueil des besoins, analyse, conception, mise en oeuvre, validation, ...)

- UML est une unification de quasiment tous les langages de modélisation d'applications orientées objet.
- UML n'est pas un langage de programmation, n'est pas un processus de développement
- UML est indépendant d'un langage de programmation
- UML est une **norme** maintenue par l'OMG
  `http://www.omg.org/uml`

Classes et objets

- On définit des classes = regroupement d'objets similaires (appelés instances)
  - **Abstraction** : factorisation des caractéristiques communes à un ensemble d'objets similaires
  - Une classe décrit une infinité d'instances
- On instancie des objets à partir des classes :
  - Objet = etat + comportement + identité
  - Attributs, méthodes, référence

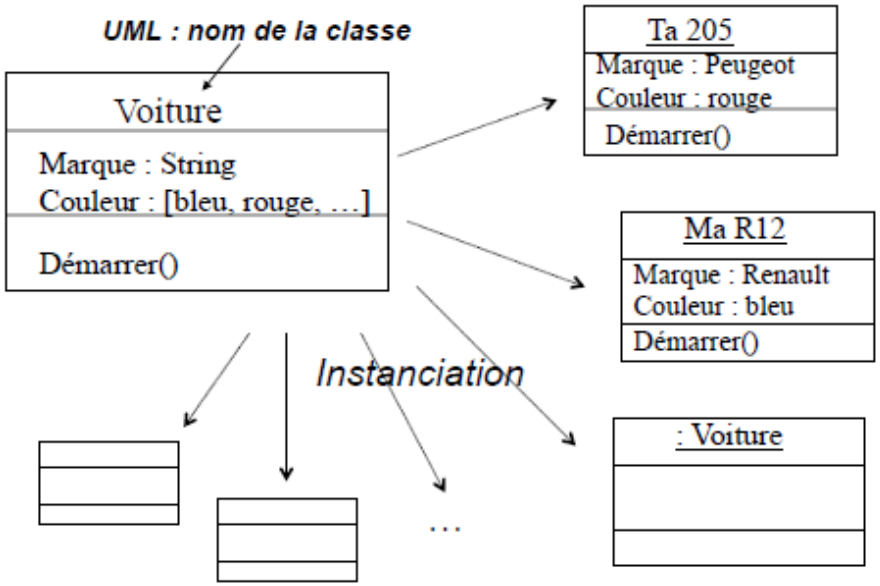
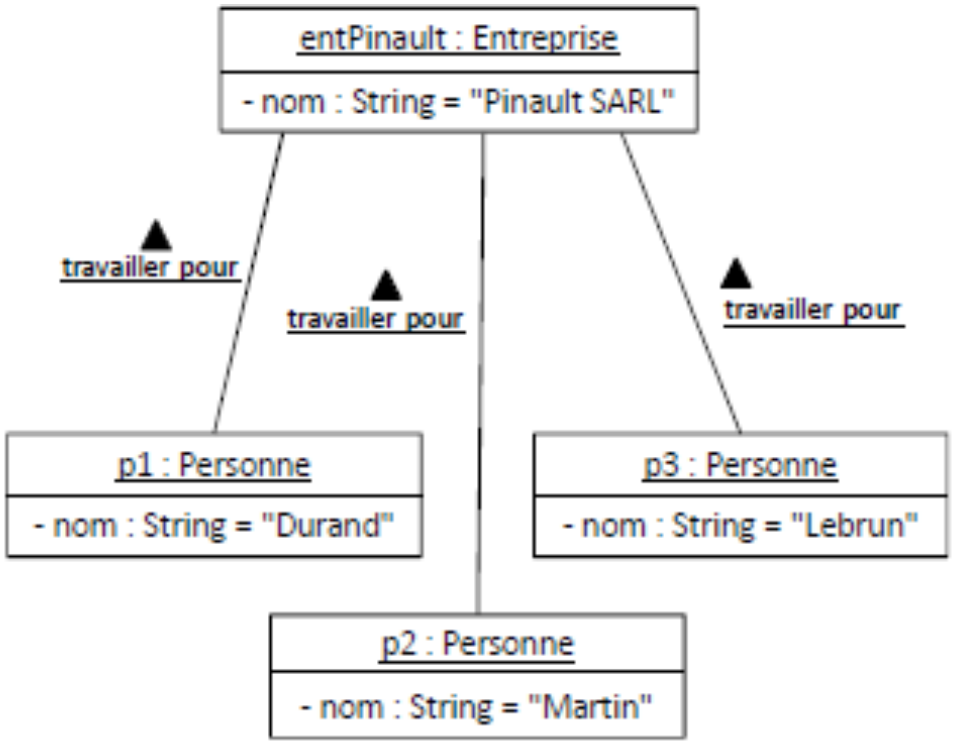


Diagramme d'objets

Objectifs

- Représente les objets/**instances** et leurs liens **à un instant donné**

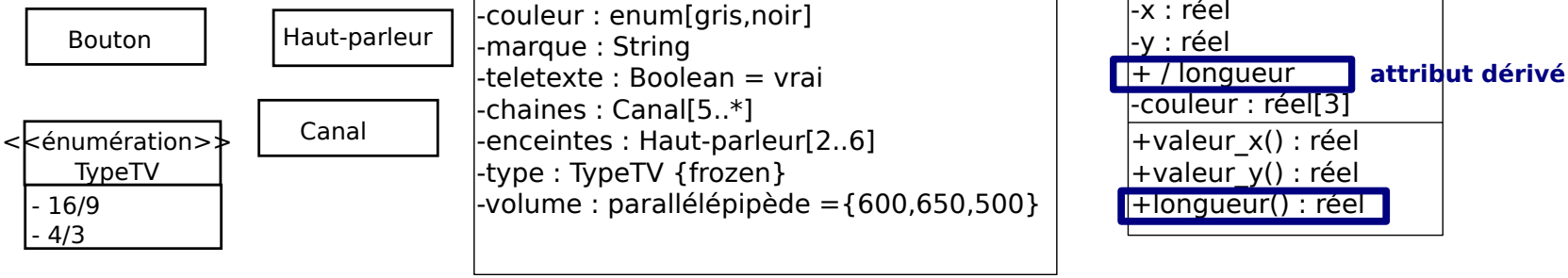


Représentation UML des attributs

- Format de description d'un attribut :

visibilité nom : type [multiplicité] = valeur_initiale {propriétés}

- public + privé - protégé # paquetage ~
- facultatif mais impératif pour l'implémentation
- facultatif ex. couleurs : Saturation[3] points: Points[2..*]
- facultatif
- facultatif ex. {frozen} mise à jour interdite



Représentation UML des attributs

- Format de description d'un attribut :

visibilité nom : type [multiplicité] = valeur_initiale {propriétés}

public +
privé -
protégé #
paquetage ~

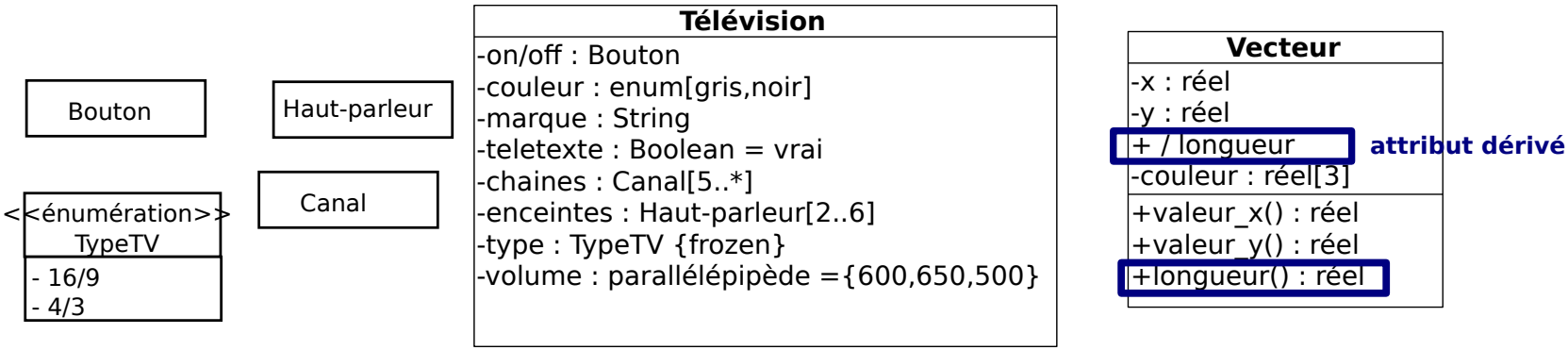
facultatif
mais impératif pour
l'implémentation

facultatif
ex.
couleurs : Saturation[3]
points: Points[2..*]

facultatif

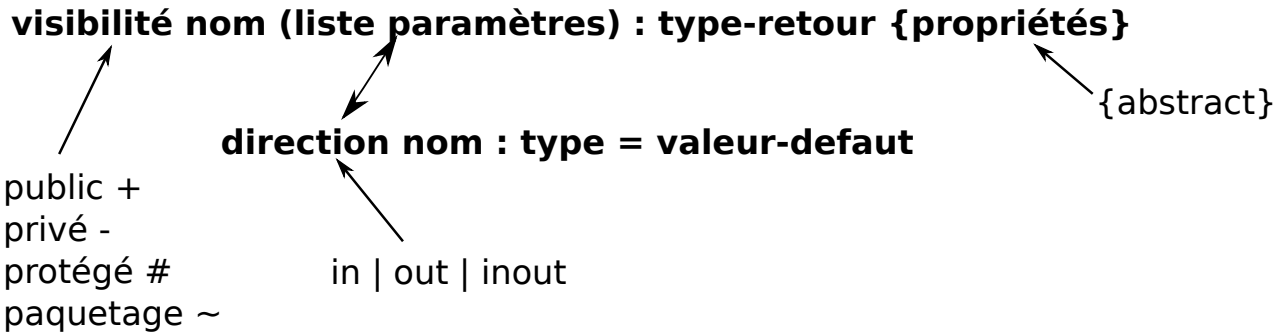
facultatif
ex.
{frozen} mise à jour interdite

- Attributs de classe (statiques) soulignés
- Attributs dérivés (calculés) précédés de " /"
- Enumération : stéréotype ≪enumeration≫

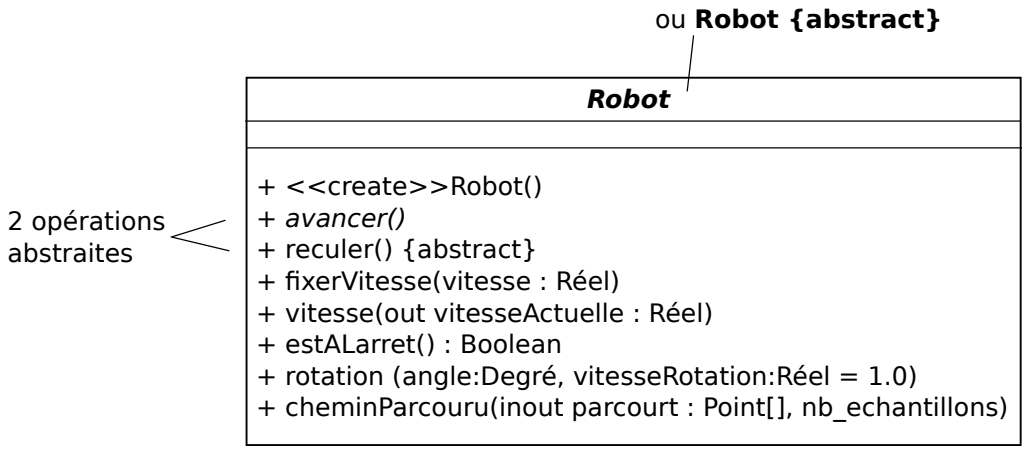


Représentation UML des opérations

- Format de description d'une opération :



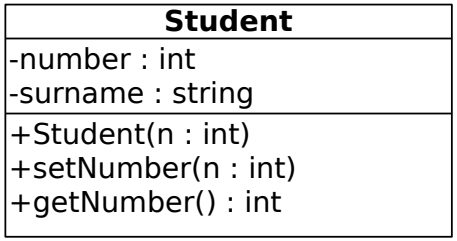
- *opérations abstraites* (non implémentées) / opérations de classe (statiques)
- Propriétés : {abstract}, {query} (l'opération n'altère pas l'état de l'instance concernée), pré- et post-conditions, description du contenu (commentaires, OCL)
- Stéréotypes d'opérations : constructeur ≪create≫, destructeur ≪destroy≫



Les classes : Implémentation Java

Student.java

```
...
public class Student {
  // Attributs
  private int number;
  private String surname;
  ...
  // Methodes
  public Student(int _n)
      {
    number = _n;
    surname = "STUDENT";
  }
  public void setNumber(
      int n) {
    number = n;
  }
  public int getNumber()
      {
    return number;
}
...
};
```

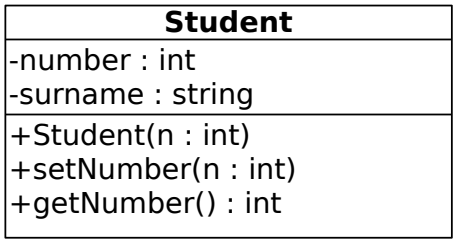


- Mode d'accès (`public`, `private`, `protected`), également appelé **visibilité**, spécifié avant chaque attributs ou méthodes
- Définition des méthodes (écriture du corps des fonctions) dans la définition de la classe, dans un unique fichier `.java`
- Obligatoirement un fichier .java par classe

Les classes : Implémentation C++

Student.h

```
class Student
{ // Attributs
  private:
    int number;
    std::string surname;
    ...
  // Methodes
  public:
   Student(int _n) {
     number = _n;
     surname = "STUDENT";
   }
   void SetNumber(int n);
   int GetNumber() const;
    ...
};
```



Student.cpp

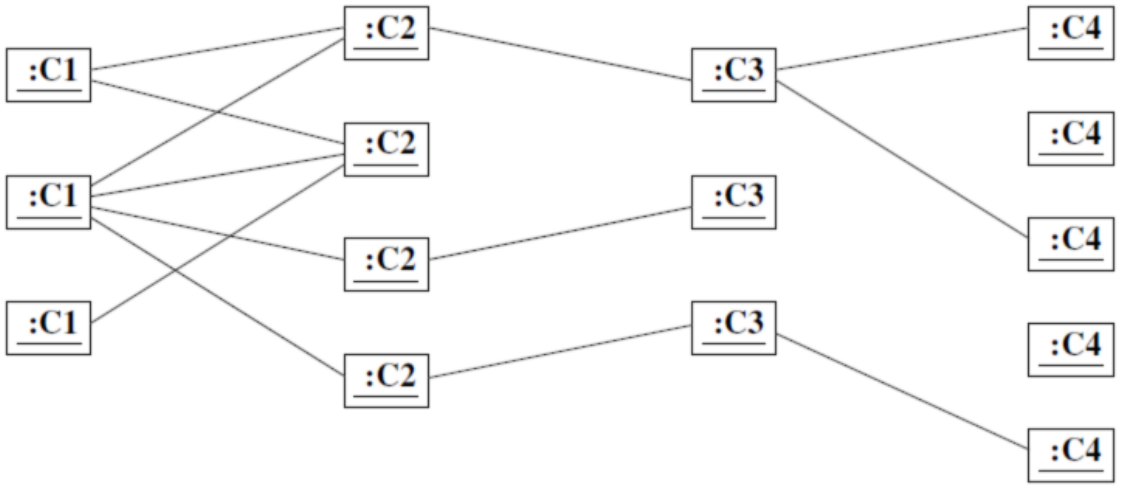
```
#include "Student.h"
void Student::SetNumber(
     int n)
{    number = n;  }
int Student::GetNumber()
     const
{    return number;  }
```

- Mode d'accès (`public`, `private`, `protected`) spécifié "par lot"
- Définition des méthodes (écriture du corps des fonctions)
  - Dans la définition de classe (.h) : méthodes inline (plutôt pour les petites fonctions)
  - En dehors de la définition de la classe (.cpp) : le plus courant
- Autant de classes que l'on veut par fichier .h

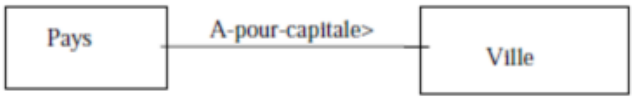
Associations entre classes

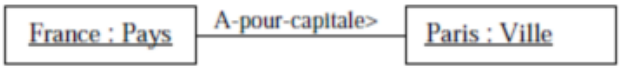
- Relation entre deux classes (binaire) ou plus (n-aire).
- Décrit les connexions structurelles entre les instances des classes associées
- Abstraction des relations définies par les liens entre objets

Liens entre objets :



Associations entre classes d'objets :

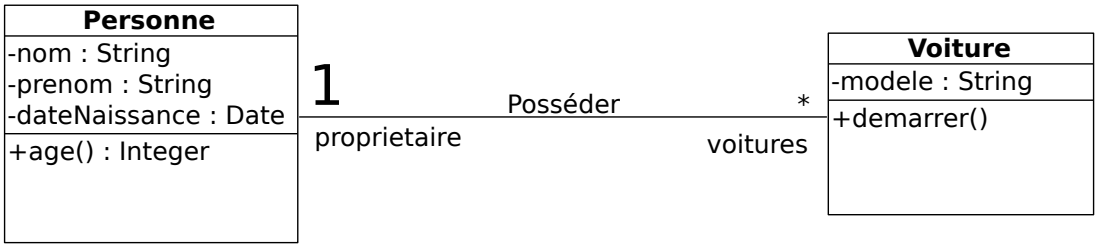


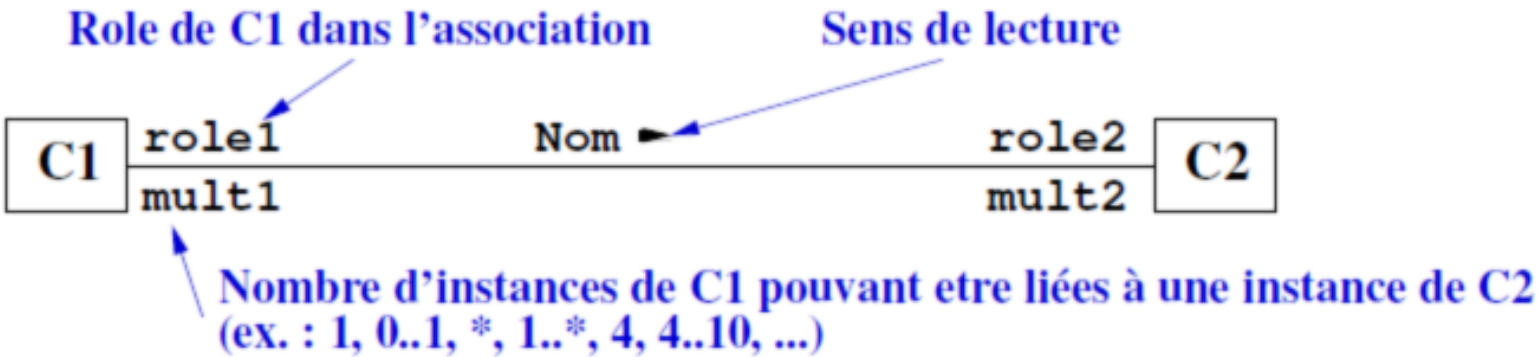


Associations entre classes

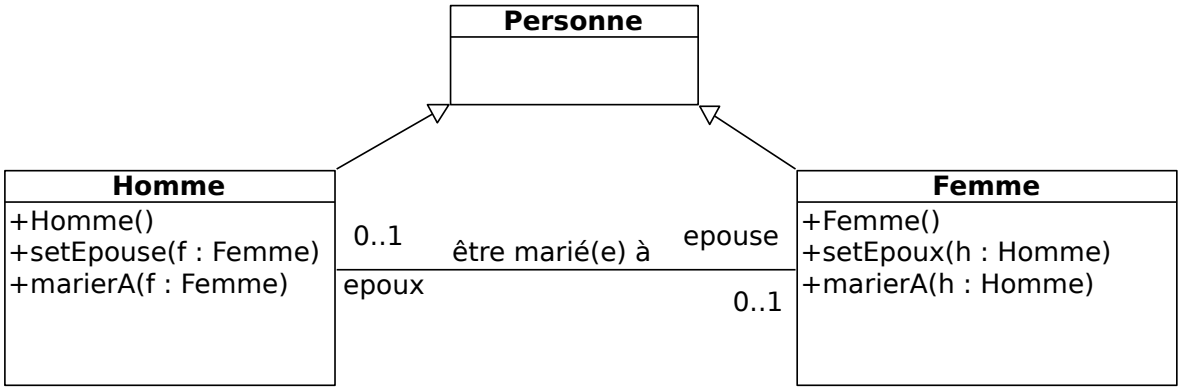
Notion d'associations

- Une association de la classe A vers B est équivalente à un attribut de type B dans la classe A





Association binaire



Homme.java

```
public class Home extends
      Personne {
  private Femme epouse;
  public Homme() {
// Celibataire par defaut
   epouse = null; }
//mutateur
  public void setEpouse(Femme f)
 {  epouse = f;}
  public void marierA(Femme f)
{   epouse = f;
   f.setEpoux(this); }
 };
```

Femme.java

```
public class Femme extends
     Personne {
private Homme epoux;
public Femme() {
 // Celibataire par defaut
 epoux = null;  }
//mutateur
public void setEpoux(Homme h)
 { epoux = h; }
public void marierA(Homme h)
{epoux = h;
h.setEpouse(this);}
};
```

Association avec sens de navigation

Diagramme UML correspondant ?

Polygone.java

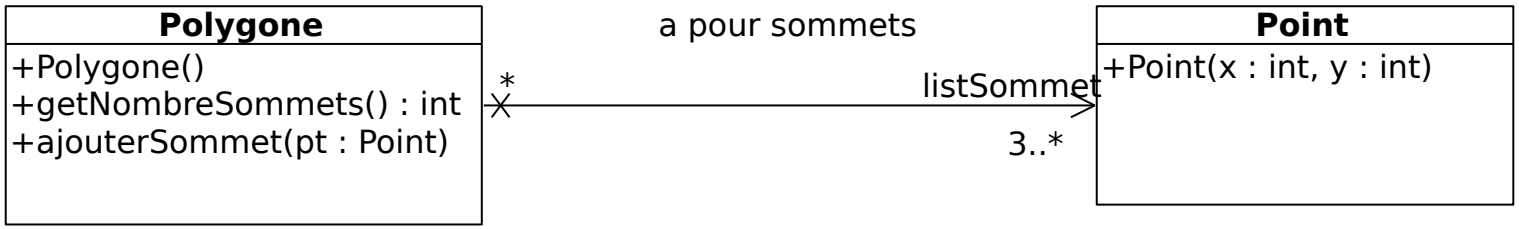
```
public class Polygone {
 protected ArrayList<Point> listSommet;
 public Polygone() {listSommet = new ArrayList<Point>();}
 public int getNombreSommets() {return listSommet.size();}
 public void ajouterSommet(Point pt) {listSommet.add(pt);}

};
```

Point.java

```
public class Point {
 public int x, y;
 ...
 public Point(int _x, int _y) {x=_x; y=_y;}
};
```

Association avec sens de navigation



Polygone.java

```
public class Polygone {
 protected ArrayList<Point> listSommet;
 public Polygone() {listSommet = new ArrayList<Point>();}
 public int getNombreSommets() {return listSommet.size();}
 public void ajouterSommet(Point pt) {listSommet.add(pt);}

};
```

Point.java

```
public class Point {
 public int x, y;
 ...
 public Point(int _x, int _y) {x=_x; y=_y;}
};
```

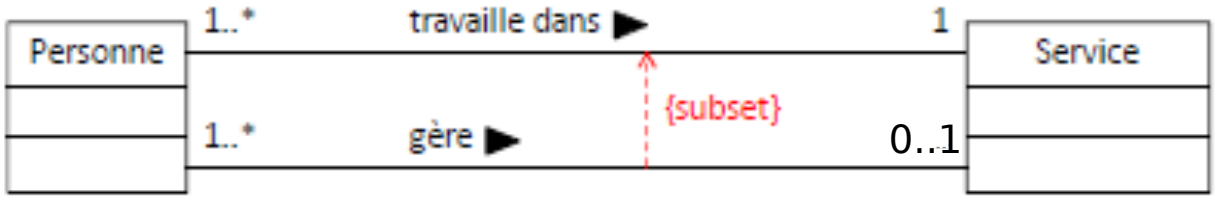
Contraintes sur les associations

- Exemple 1 : *les sommets dans un polygone sont ordonnés.*

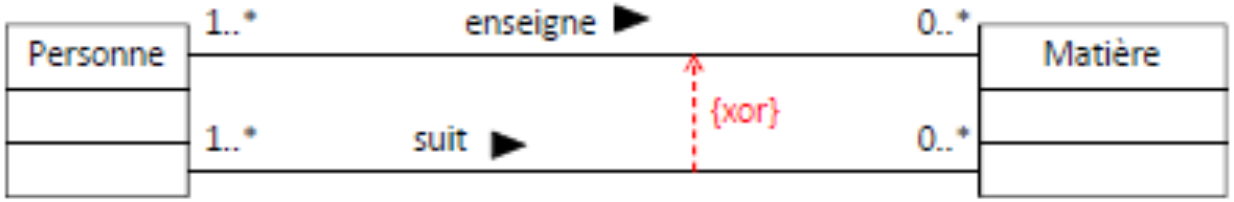
- La notion d'ordre est déjà présente dans la plupart des collections d'objet (`List` en Java ou std : :list en C++)

Contraintes sur les associations

- Contrainte d'inclusion : une personne travaille dans un service, et elle peut, en plus, gérer ce service. Plusieurs employés peuvent cogérer un service



- Contrainte d'exclusivité : une personne (sans distinction de classe entre étudiant et enseignant) est associée à une matière, soit parce qu'elle l'enseigne, soit parce qu'elle la suit (mais jamais les deux simultanément)

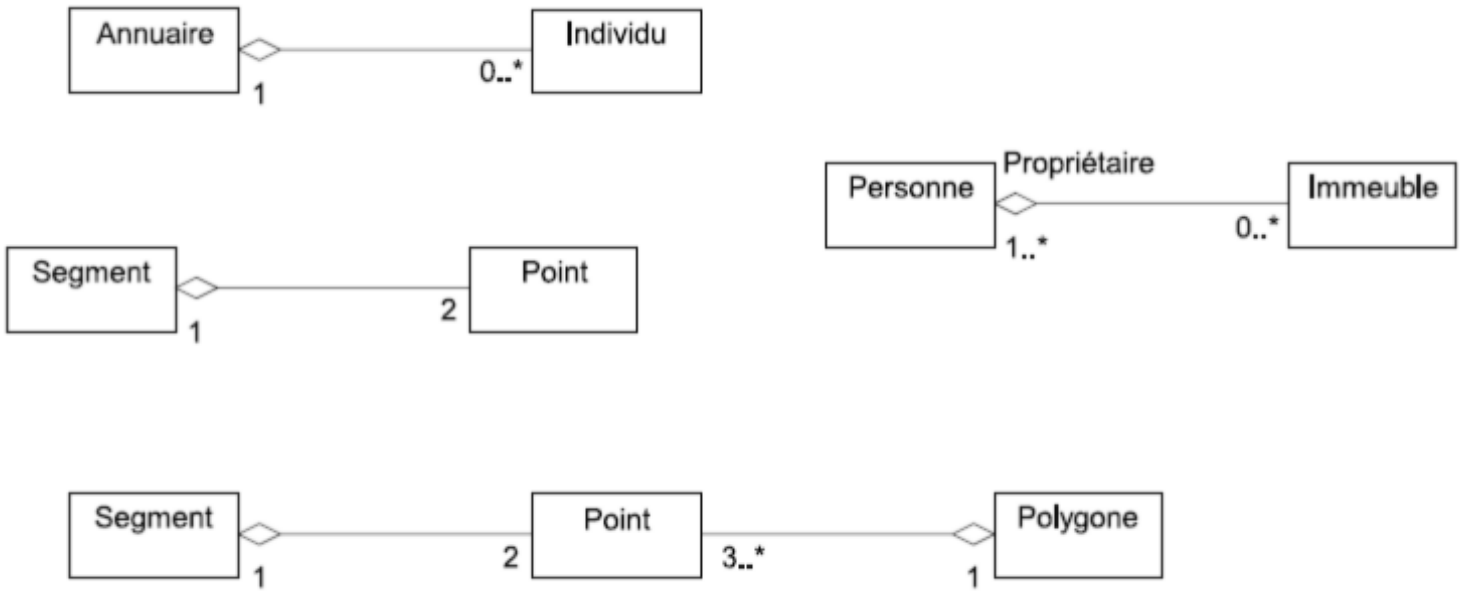


- Ces contraintes ne peuvent pas être représentée par le type de collection. Elles doivent êtres maintenues lors de la création des liens.

Associations particulières : agrégation

Agrégation

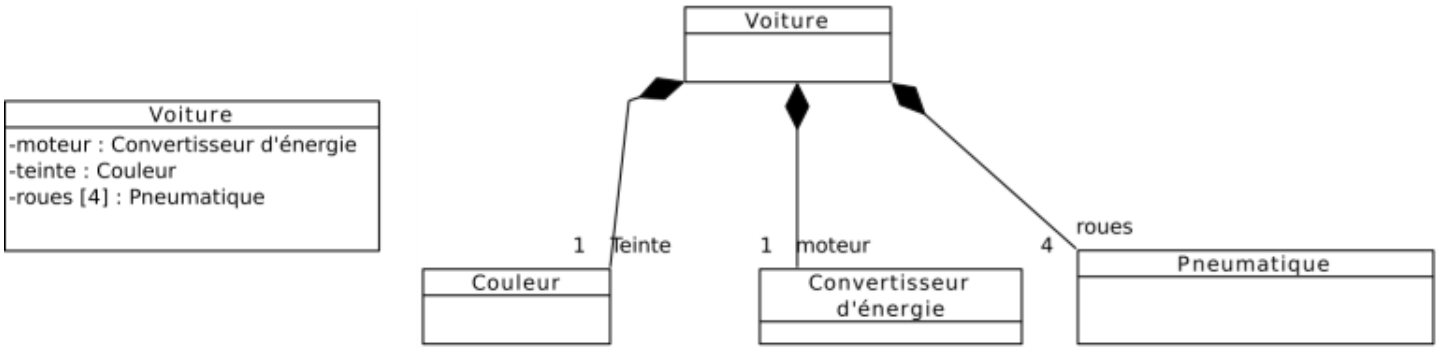
- Modéliser regroupement de parties dans un tout
- Association non-symétrique
  - Relation de dominance et de subordination
  - Une classe **fait partie** d'une autre classe
  - Une action sur une classe implique une action sur une autre classe
- Une classe peut appartenir à plusieurs agrégats



Associations particulières : composition

Composition

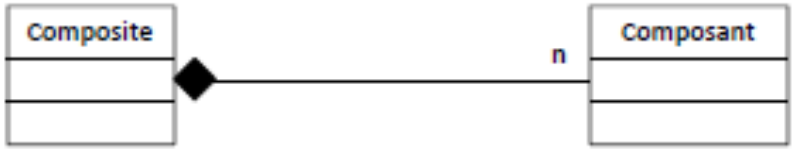
- Agrégation forte
- Contenance structurelle : Création/Copie/Destruction du composite → Création/Copie/Destruction de ses composants
- Un composant appartient à au plus un composite (mult. côté conteneur max 1)



Agrégation et composition

- Agrégation : Implémentation similaire à une association simple
- Composition :
  - Inclusion structurelle d'un composant dans un composite
  - La vie du Composant est liée à celle du Composite : la création (resp. destruction) du Composite entraîne la création (resp. destruction) de ses Composants
  - La composition peut être implémentée par l'ajout d'un attribut de classe Composant dans la classe Composite et instanciation du Composant dans la classe Composite.

Agrégation et composition



Composant.java

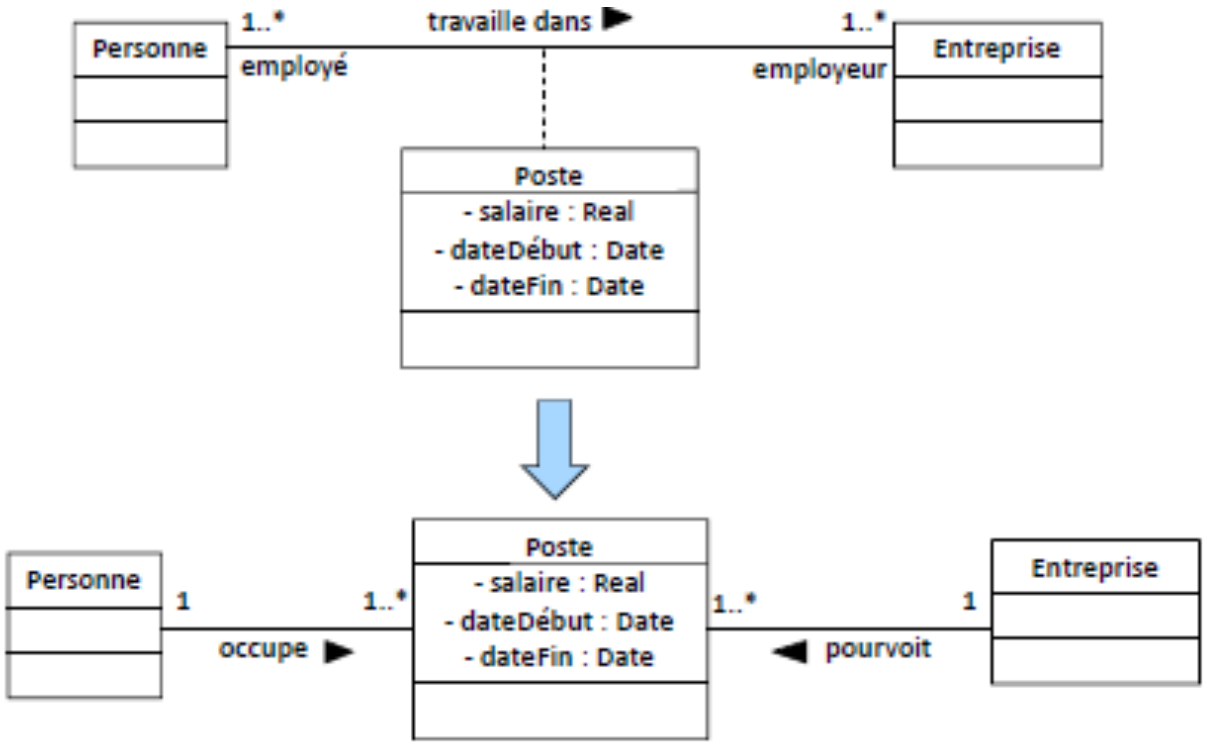
```
public class Composant {
  private Composite comp;
  public Composant() {comp = null;}
  void modifierComposite(Composite c) {comp = c;}
};
```

Composite.java

```
public class Composite {
  private Composant tabComposants[];
  public Composite() {
    tabComposants = new tabComposants[n];
    for (int i=0; i<n; i++) {
      tabComposants[i] = new Composant();
      tabComposants[i].modifierComposite(this);
    }
  }
...
}
```

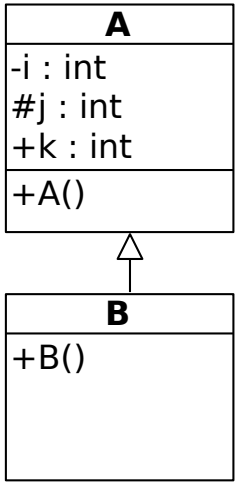
Classes-associations

- Une association peut être raffinée et avoir ses propres propriétés, qui ne sont disponibles dans aucune des classes qu'elle lie.
- Une association peut être représentée par une classe pour ajouter attributs et opérations à des associations
- Exemple : *une personne à travaillé dans des entreprises, à des périodes données et avec un certain salaire.*

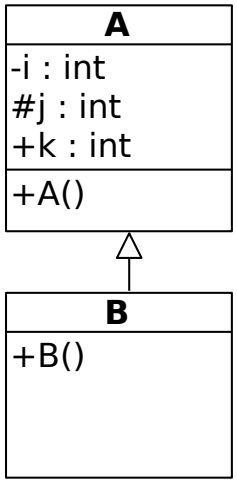


Héritage

- Concept fondamental : transmission des caractéristiques d'une classe vers une sous-classe
- Objectifs :
  - **spécialisation** d'une classe existante
  - factoriser des propriétés et comportements communs à plusieurs classes (**généralisation**)
  - hériter des attributs et des méthodes de sa super-classe
  - éviter la duplication et encourager la réutilisation
- Moyens :
  - relation de généralisation/spécialisation en UML
  - héritage en POO



Héritage simple : Implémentation



A.java

```
public class A {
  private int i;
  protected int j;
  public int k;
  public A() {
    i = 0;
    j = 0;
    k = 0;
  }
};
```

B.java

```
public class B extends
      A {
  public B() {
    i = 0; //???
    j = 0; //???
    k = 0; //???
  }
};
```

A.h

```
class A
{ private: int i;
  protected: int j;
  public: int k;
   A(){
    i = 0;
    j = 0;
    k = 0;
   } };
```

B.h

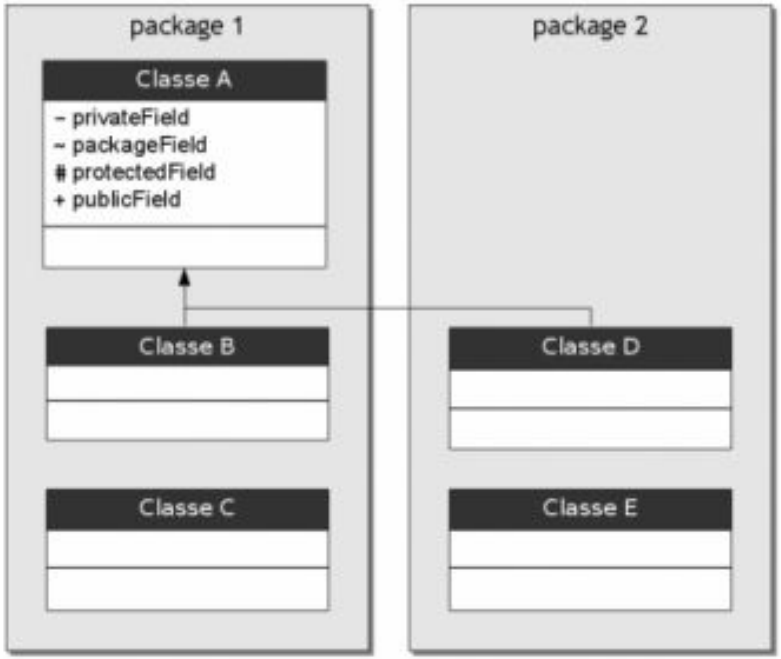
```
#include "A.hpp"
class B: public A
{
  public:
   B() {
    i = 0; // ???
    j = 0; //???
    k = 0; // ???
    }
};
```

Visibilité en Java

Quelles classes ont accès aux attributs protected de la classe A ?

Visibilité en Java

Les modificateurs de visibilité en Java



Visibilité des champs :

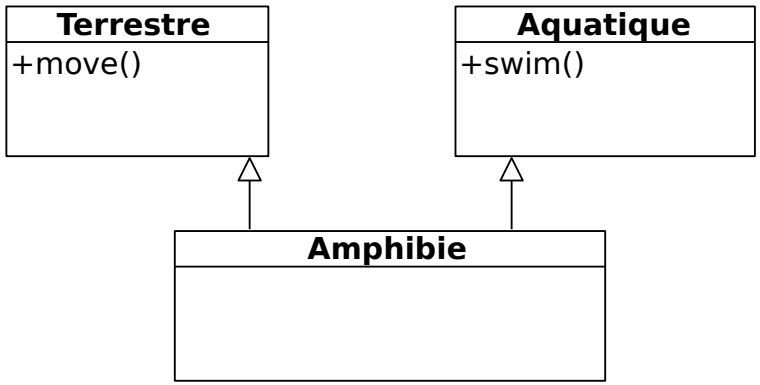
	Classe A	Classe B	Classe C	Classe D	Classe E
privateField	V				
packageField	V	V	V		
protectedField	V	V	V	V	
publicField	V	V	V	V	V

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Java Access to Members of a Class

Héritage multiple



- Héritage multiple autorisé en UML et en C++, pour bénéficier des opérations de plusieurs classes mères

Terrestre.h

```
class Terrestre {
  ...
 public:
   void move ();
}
```

Aquatique.h

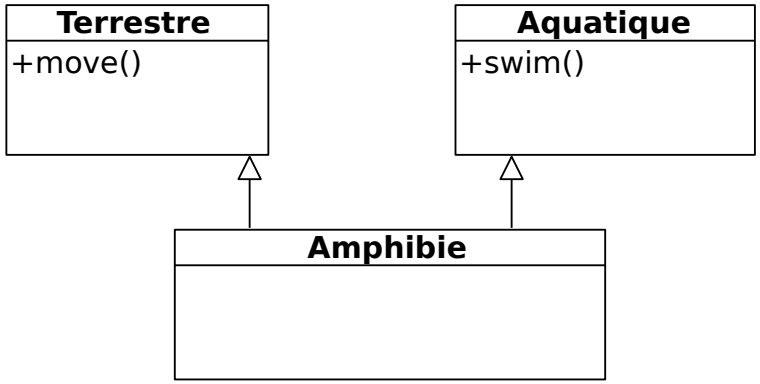
```
class Aquatique {
  ...
 public:
  void swim ();
 }
```

Amphibie.h

```
class Amphibie : public Terrestre, public Aquatique {...}
```

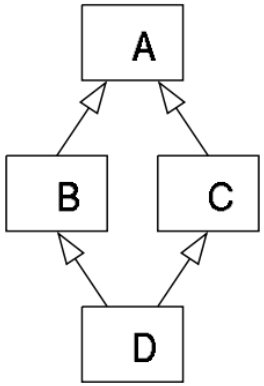
- `Amphibie` dispose à la fois des méthodes `move()` et `swim()`, qu'elle peut redéfinir

Héritage multiple : implémentation en C++



- La gestion des membres homonymes hérités des classes parentes
  - Si la classe `Amphibie` hérite à la fois des classes `Terrestre` et `Aquatique`. Il peut y avoir des problèmes de gestion des homonymes si :
    - `Terrestre` et `Aquatique` ont des attributs dont le nom est identique
    - `Terrestre` et `Aquatique` ont des méthodes dont le nom est identique
  - $\rightarrow$ on utilise le nom de la classe mère en préfixe afin de spécifier l'origine du membre requis (`Terrestre::attr` et `Aquatique::attr`)

Héritage multiple : implémentation en C++



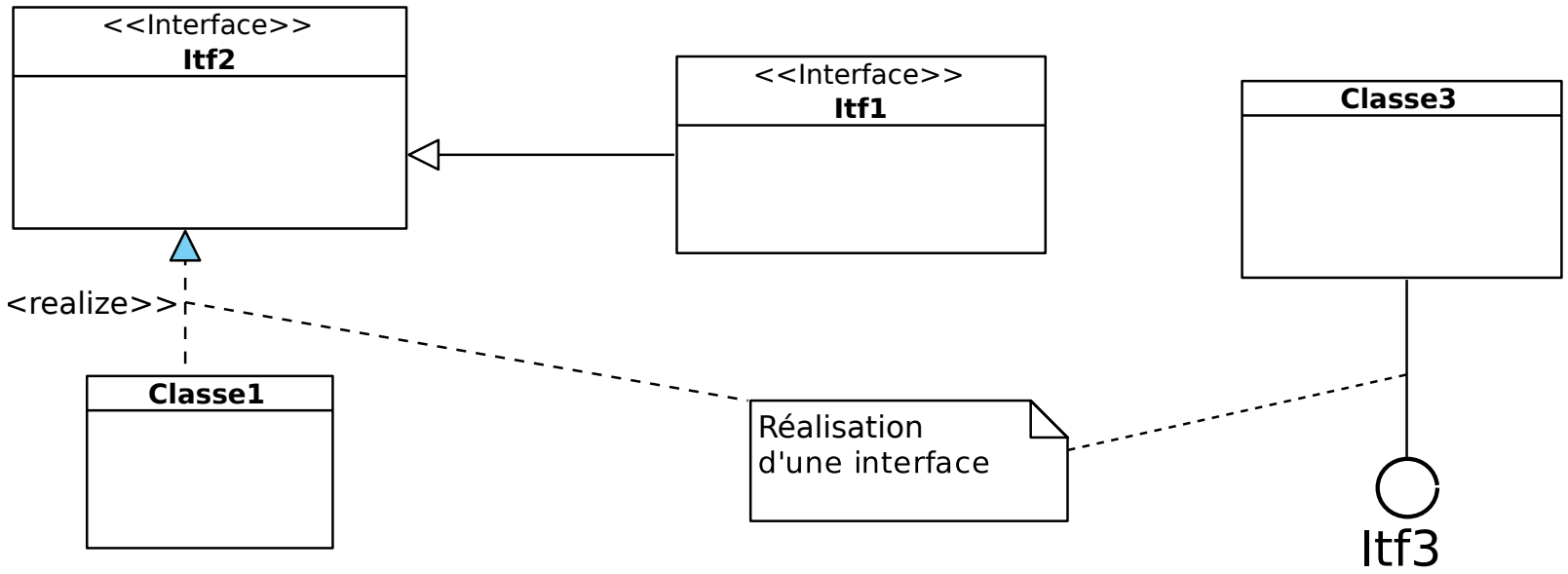
- La gestion de l'héritage à répétition
  - Les attributs de la classe A sont présents en double exemplaire dans la classe D, une fois à cause de l'héritage venant de B, une autre fois à cause de l'héritage en provenance de C
  - → Faire un héritage en mode `virtual` pour préciser de n'incorporer qu'une seule fois les membres de la classe A dans D
  - ```
    class B : public virtual A {...};
    class C : public virtual A {...};
    class D : public B, public C {...};
    ```
    ```

Interface

Qu'est-ce qu'une interface ?

Classe sans attributs dont toutes les opérations sont abstraites (classe abstraite pure)

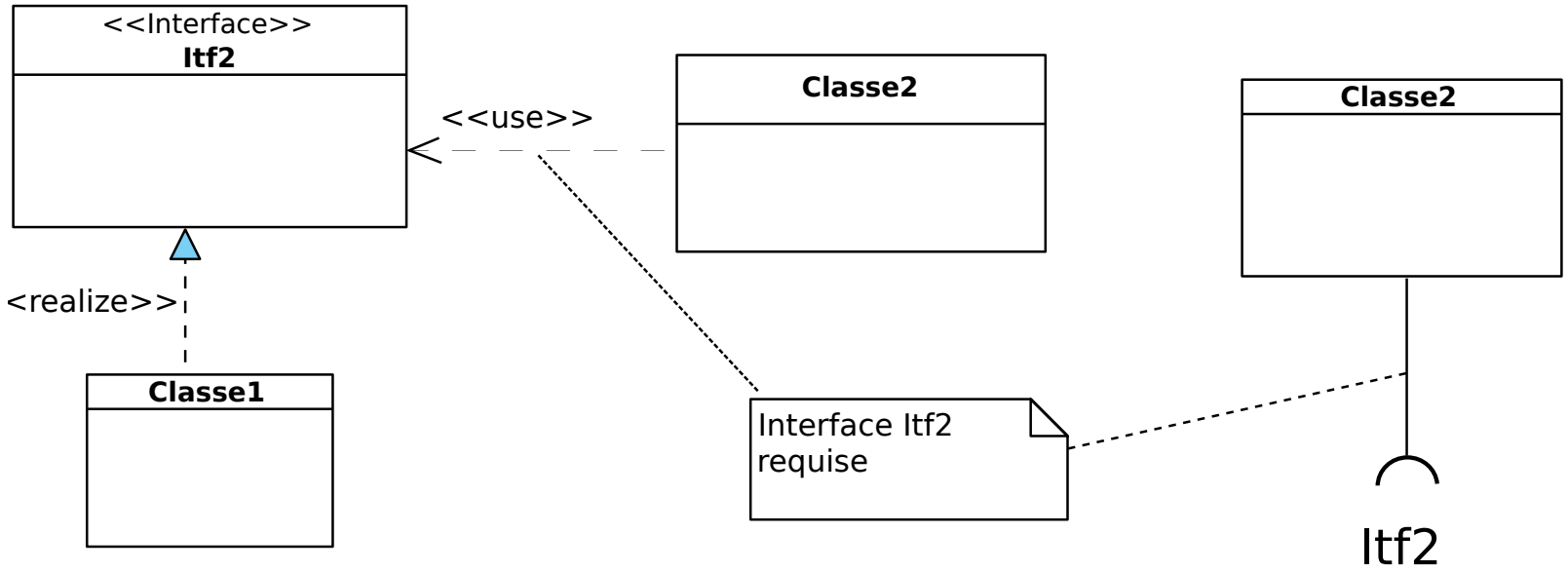
- Liste de services, savoir faire
- Ne peut pas être instanciée
- Doit être réalisée (implémentée) par des classes non abstraites
- Peut hériter d'une autre interface



Stéréotype `<<realize>>` facultatif.

Interface

Une classe peut aussi simplement dépendre d'une interface (interface requise).



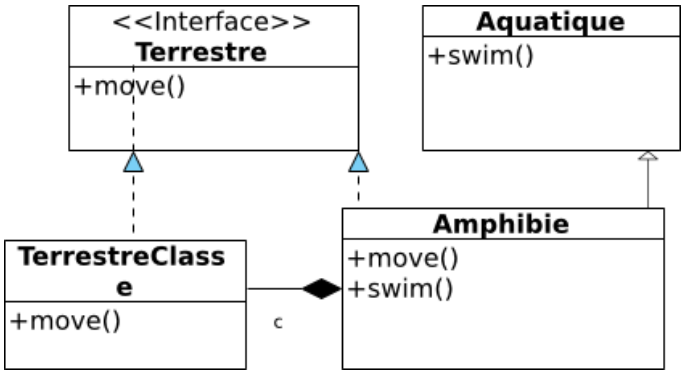
Réalisation

```
public class Classe1
    implements Itf2 {
  ...
}
```

Utilisation

```
public class Classe2 {
   public Classe2(Itf2 iface)
        { // public constructor
     ...
  }
```

Héritage multiple : implémentation en Java



- Héritage multiple impossible en Java : utilisation d'**interfaces** et **délégation**

Terrestre.java

```
public interface Terrestre {  public void move (); };
```

TerrestreClasse.java

```
public class TerrestreClasse implements Terrestre {
//TerrestreClasse doit definir la methode move()
  public void move () {...};    }
```

Amphibie.java

```
    public class Amphibie extends Aquatique implements Terrestre {
     private TerrestreClasse c;     ...
     public void move() {c.move();}   }
```

Conception logicielle

Comment concevoir des logiciels maintenables et réutilisables ?

Maitriser la nature et le nombre des dépendances :

- Respecter les concepts fondamentaux de la conception orientée objet
- Suivre les principes avancés de conception orientée objet
- Appliquer des patrons de conception (*design patterns*) qui répondent concrètement à des problèmes récurrents
- Suivre des patrons d'architecture