

Approches Objet de la Programmation

～ Agrégation, Composition, Héritage ～

Didier Verna
EPITA / LRDE

didier@lrde.epita.fr



[lrde/~didier](#)



[@didierverna](#)



[didier.verna](#)



[google+](#)



[in/didierverna](#)

Plan

Relations entre Classes

- Agrégation
- Composition
- Héritage

Caractéristiques de l'Héritage

- Hiérarchies de Classes
- Politiques d'Instanciation
- Accessibilité

Problèmes liés à l'Héritage

- Héritage et Instanciation
- Ambivalence de l'Héritage
- Héritage Multiple



Plan

Relations entre Classes

Agrégation

Composition

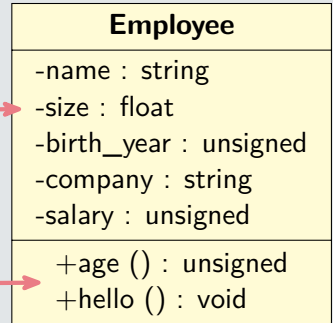
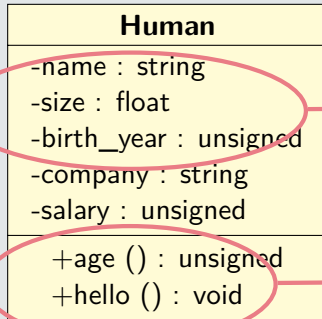
Héritage

Caractéristiques de l'Héritage

Problèmes liés à l'Héritage

Copier / Coller

Exemple UML



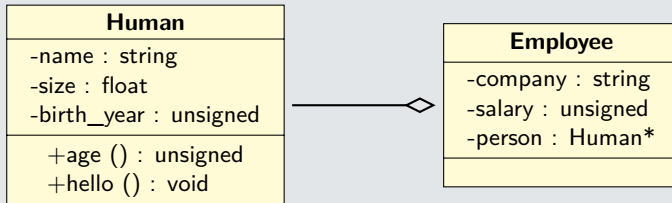
► Très mauvaise approche !



Agrégation

- ▶ Une forme d'inclusion
 - ▶ **Agrégat** : maintient de références/pointeurs vers des objets *agrégés*

Exemple UML

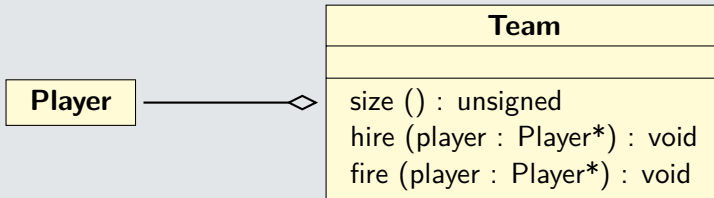


- ▶ Pas très adapté dans ce cas
 - ▶ Relation « ensemble / parties » (transitive)
 - ▶ L'agrégé survit à son agrégat

Agrégation

- ▶ Une forme d'inclusion
 - ▶ **Agrégat** : maintient de références/pointeurs vers des objets *agrégés*

Meilleur Exemple UML



Agrégation

C++

```
class Player {};
```

```
using player_set = std::unordered_set<const Player*>;
```

```
class Team
```

```
{
```

```
public:
```

```
    player_set::size_type size () const;
```

```
    void hire (const Player& player);
```

```
    void fire (const Player& player);
```

```
private:
```

```
    player_set members;
```

```
};
```

```
player_set::size_type Team::size () const { return members.size (); }
```

```
void Team::hire (const Player& player) { members.insert (&player); }
```

```
void Team::fire (const Player& player) { members.erase (&player); }
```

Agrégation

► Une forme d'inclusion

- Agrégat : maintient de références / pointeurs vers des objets agrégés

Java

```
M public class Team
{
    public static class Player {}

    public Team () { members = new HashSet<Player> (); }

    public int size () { return members.size (); }

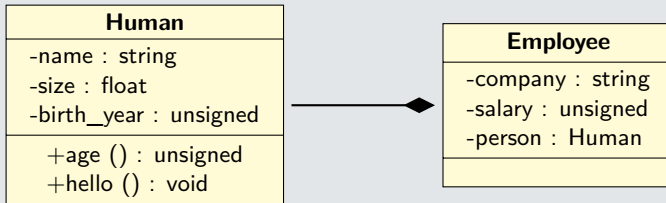
    public void hire (Player player) { members.add (player); }
    public void fire (Player player) { members.remove (player); }

    private HashSet<Player> members;
}
```


Composition (Agrégation Composite)

- ▶ Une forme d'inclusion plus stricte
 - ▶ L'agrégé ne survit pas à son (unique) agrégat

Exemple UML



- ▶ Toujours pas très adapté dans ce cas
 - ▶ L'accès à la partie Human est (toujours) indirect

Composition (Agrégation Composite)

- ▶ Une forme d'inclusion plus stricte
 - ▶ L'agrégé ne survit pas à son (unique) agrégat

Meilleur Exemple UML



Composition (Agrégation Composite)

- ▶ Une forme d'inclusion plus stricte
 - ▶ L'agrégé ne survit pas à son (unique) agrégat

Meilleur Exemple

C++

```
class Head {};  
class Arm {};  
class Leg {};  
  
class Body  
{  
private:  
    Head head;  
    Arm arms[2];  
    Leg legs[2];  
};
```

Composition (Agrégation Composite)

- ▶ Une forme d'inc
- ▶ L'agrégé ne

Java

```
public class Body
{
    public static class Head {}
    public static class Arm {}
    public static class Leg {}

    public Body ()
    {
        head = new Head ();
        arms = new Arm[2];
        legs = new Leg[2];
    }

    private Head head;
    private Arm[] arms;
    private Leg[] legs;
}
```

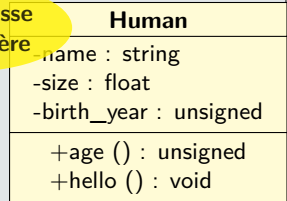
Meilleur Exemple

Héritage / Dérivation

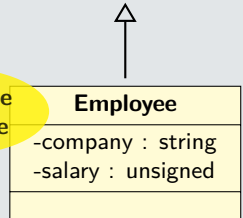
Exemple UML

- ▶ Une forme d'inclusion encore plus structurée
 - ▶ Contenu agrégé directement incorporé à la classe
- ▶ Solution la plus adaptée ici
 - ▶ Pas de risque lié à la duplication manuelle
 - ▶ Pas d'objet intermédiaire (agrégat)
 - ▶ Le contenu de la classe Human fait *implicitement* partie de la classe Employee (sauf les constructeurs / destructeurs)
- ▶ La classe Employee « hérite » ou « dérive » de la classe Human

Super-classe
Classe mère



Sous-classe
Classe fille



Héritage / Dérivation

C++

```
class Employee : public Human
{
public:
    Employee (const std::string& name, float size, unsigned birth_year,
              const std::string& company, unsigned salary);
    ~Employee ();

private:
    std::string company_;
    unsigned salary_;
};

Employee::Employee (const std::string& name, float size, unsigned birth_year,
                    const std::string& company, unsigned salary)
    : Human (name, size, birth_year), company_ (company), salary_ (salary)
{}

Employee::~~Employee ()
{}

```

Héritage / Dérivation

Exemple UML

Java

```
public class Employee extends Human
{
    public Employee (String _name, float _size, int _birthYear,
                     String _company, int _salary)
    {
        super (_name, _size, _birthYear);
        company = _company;
        salary = _salary;
    }

    public void finalize () throws Throwable
    {
        try { ... } finally { super.finalize (); }
    }

    private String company;
    private int salary;
}
```

Plan

Relations entre Classes

Caractéristiques de l'Héritage

- Hiérarchies de Classes

- Politiques d'Instanciation

- Accessibilité

Problèmes liés à l'Héritage

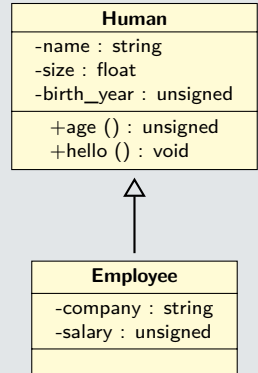
Caractéristiques de l'Héritage

- ▶ **Agrégation** : relation de type « a un »
 - ▶ Une équipe *a des* joueur, un corps *a une* tête *etc.*
- ▶ **Héritage** : relation de type « est un »
 - ▶ Un employé *est un* humain

Conséquence fondamentale :

- ▶ L'héritage *ressemble* à du sous-typage
 - ▶ Tout employé peut être vu comme un simple humain
 - ▶ La classe réelle d'un objet n'est plus nécessairement connue à la compilation
 - ▶ Mais *ressemblance seulement* !
- Cf. le principe de substitution de Liskov [Liskov, 1988]

Exemple UML



Caractéristiques de l'Héritage

► Agrégation : relation de type « a un »

Exemple IIMI

C++

```
auto h = Human { "Alain Térieur", 1.80, 1970 };
h.census ();
h.hello ();
birth_control (h);
std::cout << std::endl;

auto e = Employee { "Alex Térieur", 1.78, 1975, "EPITA", 2400 };
e.census ();
e.hello ();
birth_control (e);
std::cout << std::endl;

Human* incognito = new Employee ("Vladimir Guez", 1.85, 1980, "Ionis", 2500);
incognito->census ();
incognito->hello ();
birth_control (*incognito);
std::cout << std::endl;
```

Caractéristiques de l'Héritage

- ▶ **Agrégation** : relation de type « a un »
 - ▶ Une équipe *a des* joueur, un corps *a une*

Exemple UML

Java

```
Human h = new Human ("Alain Térieur", 1.80f, 1970);
h.census ();
h.hello ();
System.out.println ();

Employee e = new Employee ("Alex Térieur", 1.78f, 1975, "EPITA", 2400);
e.census ();
e.hello ();
System.out.println ();

Human incognito = new Employee ("Vladimir Guez", 1.85f, 1980, "Ionis", 2500);
incognito.census ();
incognito.hello ();
System.out.println ();
```

Cf. le principe de substitution de Liskov [Liskov, 1988]



Caractéristiques de l'Héritage

- ▶ **Agrégation** : relation de type « a un »
 - ▶ Une équipe *a des* joueur, un corps *a une* tête *etc.*

Exemple UML

Human
-name : string

C++

```
const Human& unpredictable ()
{
    static const auto h
        = Human { "Corinne Titgoutte", 1.68, 1985 };
    static const auto e
        = Employee { "Justine Titgoutte", 1.83, 1990, "EPITA", 2600 };

    return (rand () % 2) ? e : h;
}

const Human& dont_know = unpredictable ();
```

- ▶ **Mais ressemblance seulement !**

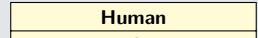
Cf. le principe de substitution de Liskov [Liskov, 1988]



Caractéristiques de l'Héritage

- ▶ **Agrégation** : relation de type « a un »
 - ▶ Une équipe *a des* joueur, un corps *a une* tête etc.

Exemple UML



Java

```

public static final class Unpredictable
{
    public static final Human get () { return random.nextBoolean () ? e : h; }

    private static final Random random = new Random ();
    private static final Human h
        = new Human ("Corinne Titgoutte", 1.68f, 1985);
    private static final Employee e
        = new Employee ("Justine Titgoutte", 1.83f, 1990, "EPITA", 2600);
}
    
```

Human dontKnow = Unpredictable.get ();

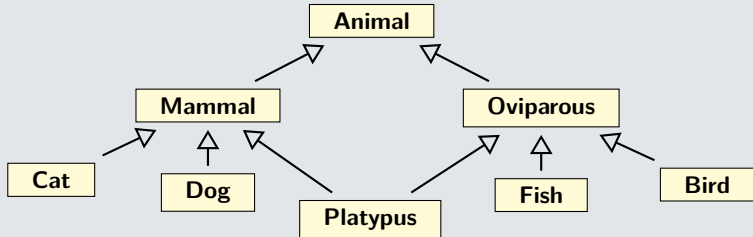
▶ Mais ressemblance seulement !

Cf. le principe de substitution de Liskov [Liskov, 1988]



Hiérarchies de Classes

Exemple UML



- L'héritage est une relation transitive
- Héritage multiple (pas toujours disponible) : plusieurs super-classes
- **Hiérarchie de classes** : arbre (ou graphe) orienté d'héritage



Hiérarchies de Classes

Exemple UML

C++

```
class Animal {};
```

```
class Mammal : public Animal {};
```

```
class Cat : public Mammal {};
```

```
class Dog : public Mammal {};
```

```
class Oviparous : public Animal {};
```

```
class Bird : public Oviparous {};
```

```
class Fish : public Oviparous {};
```

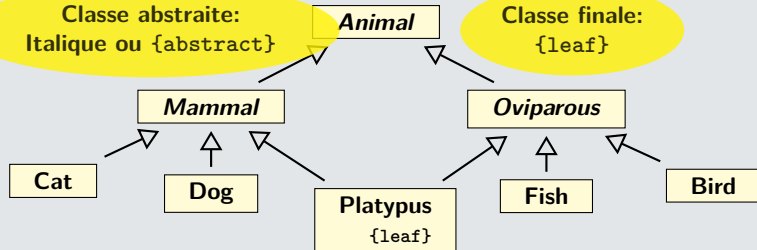
```
class Platypus : public Mammal, public Oviparous {};
```

- ▶ L
- ▶ H
- ▶ Hierarchie de classes : arbre (ou graphe) orienté d'héritage



Politiques d'Instanciation

Exemple UML



- Classe abstraite : non instanciable
- Classe finale : non dérivable



Politiques d'Instanciation

C++

```
class Animal
{
public:
    // Make this class abstract.
    virtual void speak () const = 0;
};

// Also abstract.
class Mammal : public Animal {};
class Cat : public Mammal
{
public:
    // Not abstract anymore.
    void speak () const override { std::cout << "Meow! Meow!\n"; };
};
```

Politiques d'Instanciation

Exemple UML

Classe abstraite:
Italique ou {abstract}

Animal

Classe finale:
{leaf}

Java

```
public abstract class Animal
{
    public static abstract class Mammal extends Animal {}
    public static class Cat extends Mammal
    {
        void speak () { System.out.println ("Meow! Meow!"); }
    }
}
```

► Classe finale : non dérivable



Politiques d'Instanciation

Exemple UML

Classe abstraite:
Italique ou {abstract}

Animal

Classe finale:
{leaf}



C++

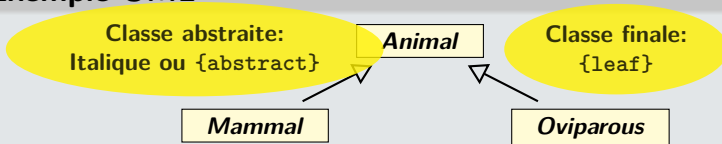
```

// Final class (C++11).
class Platypus final : public Mammal, public Oviparous
{
public:
    // Not abstract anymore.
    void speak () const override { std::cout << "Platty! Platty!\n"; };
};
  
```

- Classe finale : non dérivable

Politiques d'Instanciation

Exemple UML



Java

```

public static final class Platypus extends Animal
{
    void speak () { System.out.println ("Platty! Platty!"); }
}
  
```

- ▶ Classe abstraite : non instanciable
- ▶ Classe finale : non dérivable



Accessibilité et Héritage

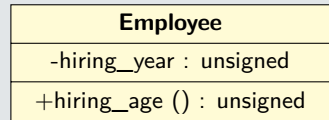
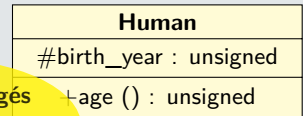
► Attribut / Méthode

Protégé(e) :

accès restreint à la sous-hiérarchie
de la classe

Champs protégés
précédés par un
"#"

Exemple UML



Accessibilité et Héritage

C++

```
class Human
{
protected:
    const unsigned birth_year_;
};

class Employee : public Human
{
public:
    unsigned hiring_age () const;

private:
    unsigned hiring_year_;
};

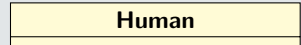
unsigned Employee::hiring_age () const
{
    return hiring_year_ - birth_year_;
}
```

Accessibilité et Héritage

► Attribut / Méthode Protégé(e) :

accès restreint à la sous-hiérarchie

Exemple UML



Java

```

public class Human
{
    protected final int birthYear;
}

public class Employee extends Human
{
    public int hiringAge () { return hiringYear - birthYear; }
    private int hiringYear;
}
  
```

Plan

Relations entre Classes

Caractéristiques de l'Héritage

Problèmes liés à l'Héritage

Héritage et Instanciation

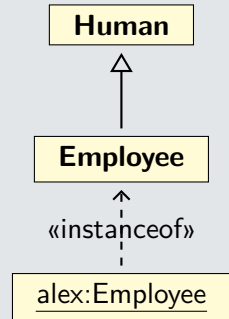
Ambivalence de l'Héritage

Héritage Multiple

Héritage et Instanciation

- ▶ Manipuler la relation « est un » avec précaution
- ▶ Un employé est un (type d')humain
- ▶ Alex est un employé (en particulier)

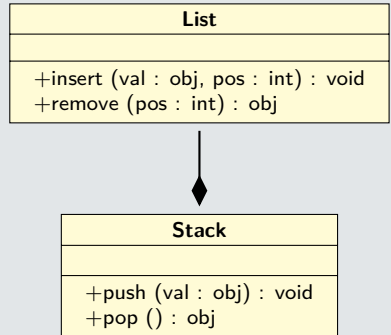
Exemple UML



Ambivalence de l'Héritage

- ▶ Problèmes :
 1. Exposition de l'implémentation
 2. Héritage de l'interface de liste
- ▶ Deux effets du sous-classage :
 1. Héritage d'implémentation
ré-utilisabilité du code
 2. Héritage d'interface
sémantique, sous-typage
- ▶ L'héritage d'implémentation entraîne celui de l'interface
- ▶ Préférer la composition à l'héritage
Une pile n'est pas une liste

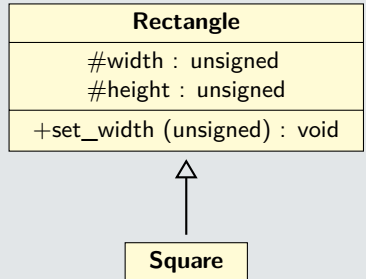
Exemple UML



Héritage par Restriction

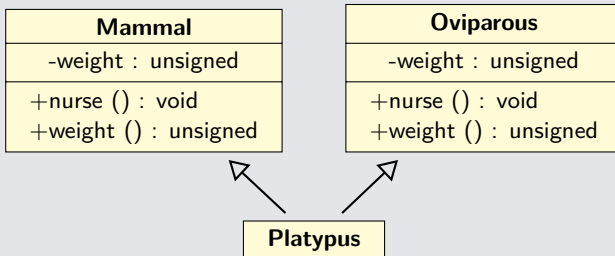
- ▶ Le problème cercle/ellipse (carré/rectangle)
- ▶ Un carré est un rectangle...
- ▶ ...mais avec des contraintes statiques...
- ▶ ...et dynamiques
- ▶ Programmation Différentielle :
 - ▶ Hériter de manière additive et non restrictive
 - ▶ Problème surtout lié à la mutation
- ▶ Cf. le principe de substitution de Liskov [Liskov, 1988]

Exemple UML



Ambiguïtés de l'Héritage Multiple

Exemple UML



- ▶ Quelle(s) méthode(s) choisir (`nurse`) ?
- ▶ Pourquoi y en aurait-il plusieurs (`weight`) ?
- ▶ Remarques tout aussi valables pour les attributs

Ambiguïtés de l'Héritage Multiple

Ex C++

```
class Mammal
{
public:
    void nurse () const { std::cout << "I milk my offsprings.\n"; }
};
```

```
class Oviparous
{
public:
    void nurse () const { std::cout << "I brood my eggs.\n"; }
};
```

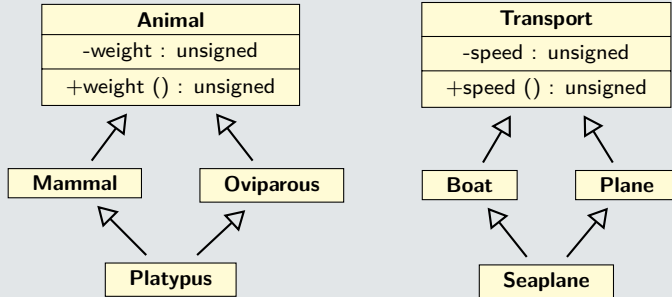
```
class Platypus final : public Mammal, public Oviparous {};

Platypus platty;
platty.Oviparous::nurse ();
platty.Mammal::nurse ();
```



Héritage en Diamant / Losange

Exemple UML



- Combien de copies de la classe de base veut-on ?
- Pourquoi raisonner au niveau classe ?

Héritage en Diamant / Losange

Ex C++

```
class Animal
{
public:
    Animal (unsigned weight) : weight_ (weight) {};

private:
    unsigned weight_;
};

class Mammal      : public virtual Animal {};
class Oviparous   : public virtual Animal {};

class Platypus final : public Mammal, public Oviparous
{
public:
    Platypus (unsigned weight) : Animal (weight) {};
};
```



C++

Ex

```
class Transport
{
public:
    Transport (unsigned speed) : speed_ (speed) {};
private:
    unsigned speed_;
};
class Boat : public Transport
{
public:
    Boat (unsigned speed) : Transport (speed) {};
};
class Plane : public Transport
{
public:
    Plane (unsigned speed) : Transport (speed) {};
};
class Seaplane : public Boat, public Plane
{
public:
    Seaplane (unsigned boat_speed, unsigned plane_speed)
        : Boat (boat_speed), Plane (plane_speed)
    {};
};
```


Plan

Bibliographie

Bibliographie



Barbara Liskov.

Data Abstraction and Hierarchy.

OOPSLA'87 Keynote, 1988.