Object-Oriented Approaches to Programming

∼ Aggregation, Composition, Inheritance ∽

Didier Verna

EPITA / LRE

didier@lrde.epita.fr

lrde/~didier @didierverna didier.verna in/didierverna

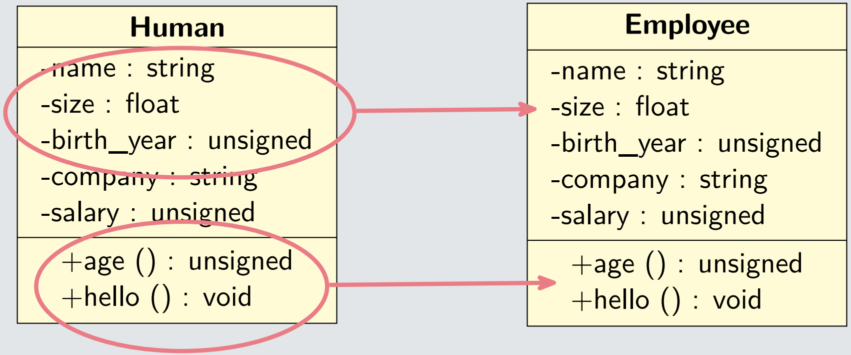
Build 2024-11-03 12:29:16+01:00

Outline

Plan

Copy / Paste

Example



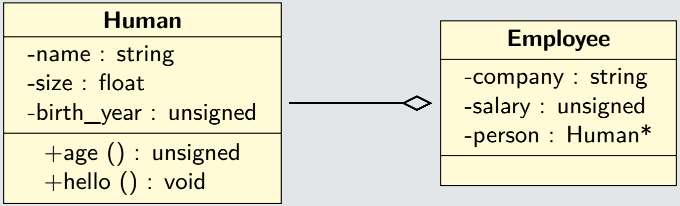
😠 Very bad approach!

Plan

Aggregation

- A kind of inclusion
  - **Aggregate**: maintenance of references / pointers to *aggregated* objets

Example



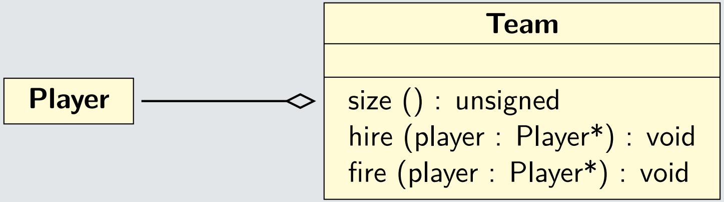
😕 Not very fit for this example

- Relation "set / elements" (transitive)
- The aggregated survives its aggregate

Aggregation

- A kind of inclusion
  - **Aggregate**: maintenance of references / pointers to *aggregated* objets

Better Example


Team
Player
size () : unsigned
hire (player : Player*) : void
fire (player : Player*) : void

Aggregation

- A kind c
  - Agg

Bet

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);  }
```

Aggregation

- A kind of inclusion
  - **Aggregate**: maintenance of references / pointers to *aggregated* objets

Bet

Java

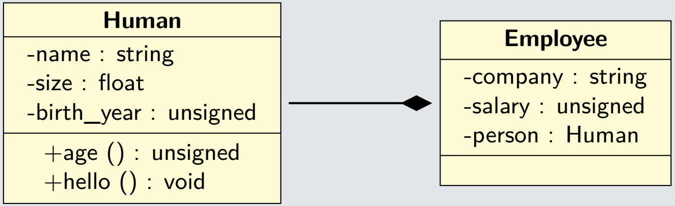
```
public class Player {}

public class Team
{
  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;
}
```

Plan

Composition (Composite Aggregation)

- A stricter form of inclusion
  - The aggregated does not survive its (unique) aggregate

Example

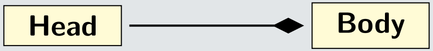


😕 Still not very fit for this example
  - Access to the `Human` part is (still) indirect

Composition (Composite Aggregation)

- A stricter form of inclusion
  - The aggregated does not survive its (unique) aggregate

Better Example



Composition (Composite Aggregation)

- A stricter form of inclusion
  - The aggregated does not survive its (unique) aggregate

Better Exampl

C++

```cpp
class Head {};
class Arm {};
class Leg {};

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

Composition (Composite Aggregation)

- A stricter form of inclusion
  - The aggregated do

Better Exampl

Java

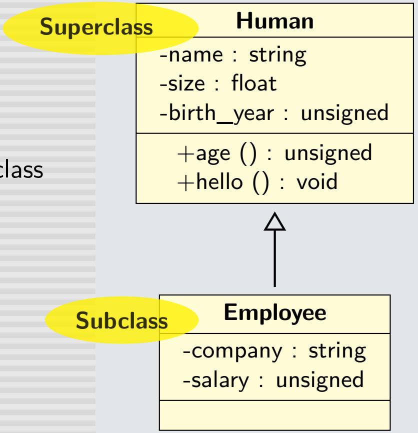
```java
public class Head {}
public class Arm {}
public class Leg {}

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

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

Plan

Inheritance / Derivation

- An *even stricter* form of inclusion
  - Aggregated contents directly incorporated into the class
- 🙂 Best solution here
  - No risk related to manual duplication
  - No intermediate object (aggregated)
  - The contents of class `Human` belongs *implicitly* to class `Employee` as well (except for constructors / destructors)
- Class `Employee` "inherits" or "derives" from class `Human`
- 😉 Not far from (automatic) cut'n paste

Example



Inheritance / Derivation

- An
  -
- Bes
  -
  -
  -
- Clas
  Hum
- Not

C++

```cpp
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 ()
{}
```

gned
ned
ng
ed

Inheritance / Derivation

- An *even stricter* form of inclusion
  - Aggregated contents directly incorporated into the

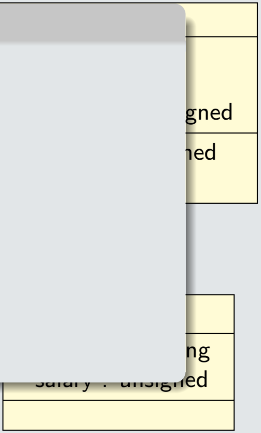
Bes

Clas

Hum

Not

Example



Java

```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;
  }

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

Build 2024-11-03 12:29:16+01:00

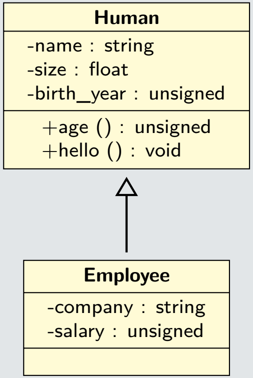
Plan

Characteristics of Inheritance

- **Aggregation**: "has a" relationship
  - A team *has a* player, a body *has a* head, *etc.*
- **Inheritance**: "is a" relationship
  - An employee *is a* human
- **Consequence**
  - Inheritance *looks like* sub-typing
    - Every employee can be seen as a simple human
    - The actual class of an object does not need to be known at compile-time any longer

⚠ But "looks like" only!
Cf. the Liskov substitution principle [Liskov, 1988]

Example



Build 2024-11-03 12:29:16+01:00

Characteristics of Inheritance

- Agg
  -
- Inh
  -
- Cor
  -

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;
```

ned
ed
g
d

Characteristics of Inheritance

- **Aggregation**: "has a" relationship
  -
- **Inh**
  -
- **Co**
  -

Example

ned

ed

g

d

Java

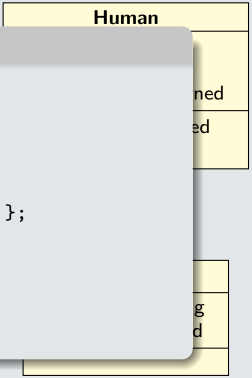
```java
Human h = new Human ("Alain Térieur", 1.80f, 1970);
h.census ();
h.hello ();
System.out.println ();

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 ();
```

Characteristics of Inheritance

- **Aggregation**: "has a" relationship
  - A team *has a* player, a body *has a* head, *etc.*
- **Inh**
- **Co**

⚠

Example



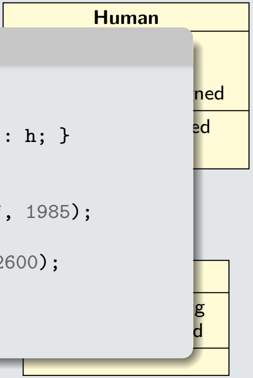
C++

```
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 ();
```

Characteristics of Inheritance

- **Aggregation**: "has a" relationship
  - A team *has a* player, a body *has a* head, *etc.*
- **Inh**
- **Cor**

Example



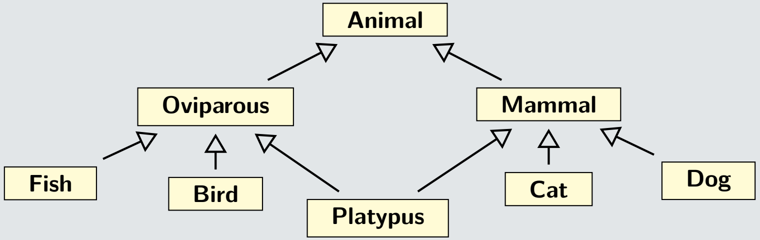
Java

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

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

Human dontKnow = Unpredictable.get ();
```

Plan

Class Hierarchies

Example



- Inheritance is a transitive relation
- Multiple inheritance (not always available): several super-classes
- Class hierarchy: oriented inheritance tree (or graph)

Class Hierarchies

Example

C++

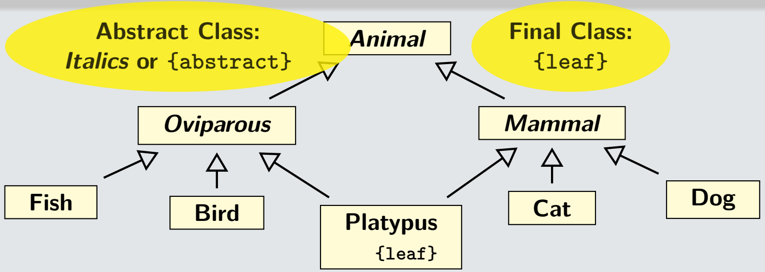
```
class Animal {};

class Oviparous : public Animal {};
class Bird : public Oviparous {};
class Fish : public Oviparous {};

class Mammal : public Animal {};
class Cat : public Mammal {};
class Dog : public Mammal {};

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

- Inheritance is
- Multiple inhe
- Class hierarcl

Plan

Instantiation Policies

Example



- **Abstract Class:** not instantiable
- **Final Class:** non derivable
  Additional technical benefits: safety, performance

Instantiation Policies

Exa

C++

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

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

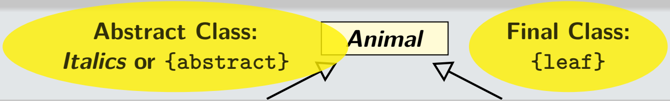
- **Abs**
- **Fin**

Adc

Build 2024-11-03 12:29:16+01:00

Instantiation Policies

Example



Java

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

- **Abs**
- **Final Class:** non derivable
  Additional technical benefits: safety, performance

Instantiation Policies

Example

Abstract Class:
Italics or {abstract}
Animal
Final Class:
{leaf}

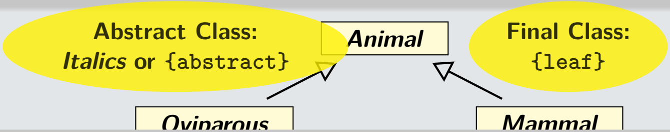
C++

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

- **Abs**
- **Final Class:** non derivable
  Additional technical benefits: safety, performance

Instantiation Policies

Example



Java

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

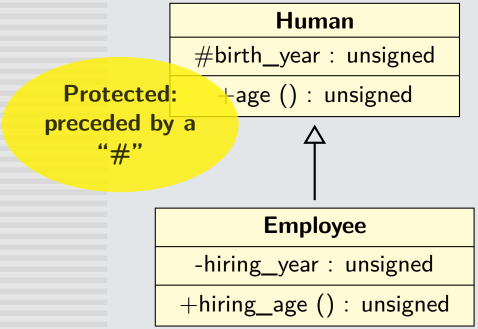
- **Abstract Class:** not instantiable
- **Final Class:** non derivable
  Additional technical benefits: safety, performance

Plan

Accessibility and Inheritance

- **Protected Attribute / Method:** acces restricted to the class sub-hierarchy
- **Remark:** Public interface + protected attributes = closest form to the principle of sub-typing

Example

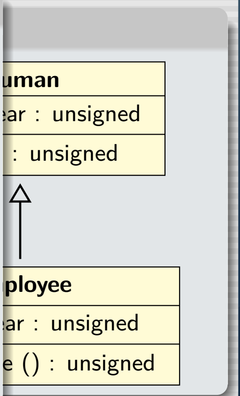


Accessibility and Inheritance

- **Protected A**
  acces restrict
- **Remark:** Pu
  attributes =
  sub-typing

C++

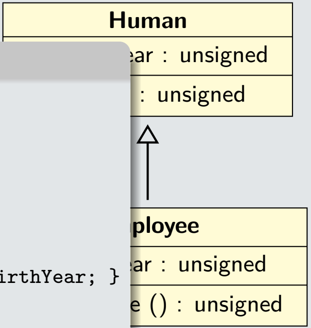
```
class Human
{
protected:
  const unsigned birth_year_;
};

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

private:
  unsigned hiring_year_;
};

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



Accessibility and Inheritance

- **Protected Attribute / Method:** acces restricted to the class sub-hierarchy
- **Remark:** Public interface + protected attributes = sub-typing

Example



Java

```
public class Human
{
  protected final int birthYear;
}

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

Build 2024-11-03 12:29:16+01:00

Plan

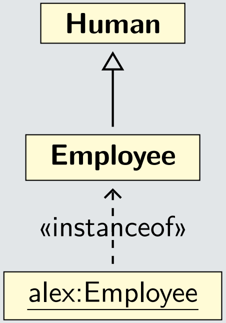
Plan

Inheritance and Instantiation

⚠ Manipulate the relation "is a" with caution

- An employee is a (kind of) human
- Alex is an employee (in particular)

Example



Plan

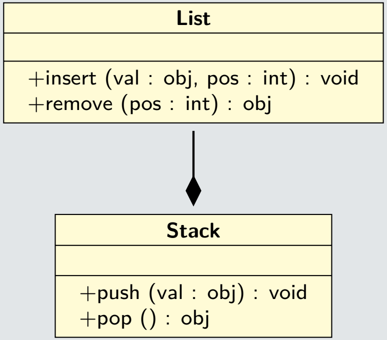
Ambivalence of Inheritance

- Problems:
  1. Exposition of the implementation
  2. Inheritance of the list's interface
- Two effects of sub-classing:
  1. Inheritance of implementation
     code reusability
  2. Inheritance of interface
     semantics, sub-typing

⚠ Implementation inheritance entails interface inheritance

- Favor composition over inheritance
  A stack is not a list

Example


List
+insert (val : obj, pos : int) : void
+remove (pos : int) : obj
Stack
+push (val : obj) : void
+pop () : obj

Remark: “Private” Inheritance

C++

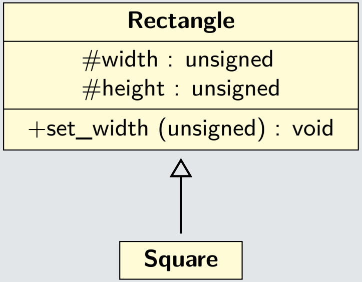
```
class List
{
public:
  void insert (obj val, int pos);
  obj remove (int pos);
};

class Stack : ~~public~~ private List
{
public:
  void push (obj val) { insert (val, 0); };
  obj pop () { return remove (0); }
};
```

- “Is implemented in terms of” relation
- Favor composition, still

Inheritance by Restriction

- The square rectangle (elliptic circle) problem
  - A square is a rectangle…
  - …although with static constraints…
  - …and dynamic ones
- Differential Programming:
  - Inherit in an additive (not restrictive) manner
  - Problem mostly related to mutation
- Cf. Liskov substitution principle [Liskov, 1988]

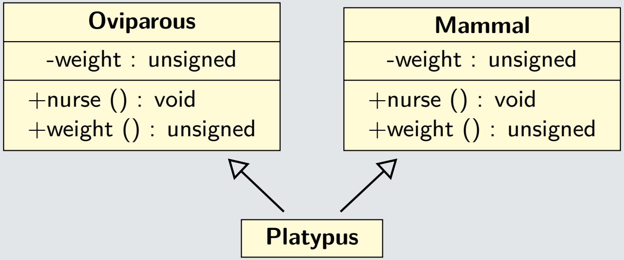
Example



Plan

Multiple Inheritance Ambiguities

Example



- Which method(s) to choose (`nurse`) ?
- Why would there be several (`weight`) ?
- Those remarks apply to attributes as well

Multiple Inheritance Ambiguities

Exa

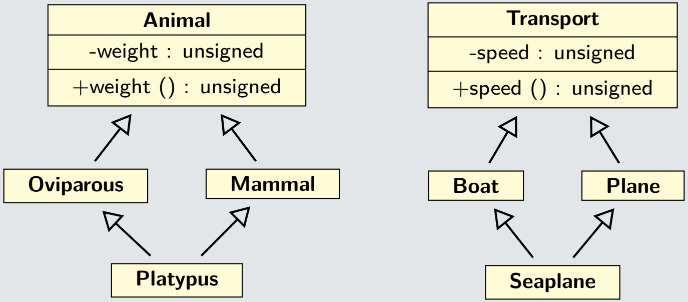
C++

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

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

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

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

- Which n
- Why wo
- Those re

Diamond Inheritance

Exemple



- How many copies of the base class do we want?
- Why reason at the class level?
- ⚠ Each language has its own position…

Diamond Inheritance

Exempl

“Shared” inheritance

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

private:
  unsigned weight_;
};

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

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

- How many c
- Why reason
- ⚠️ Each languag

Diamon

Exempl

- How many c
- Why reason
- ⚠ Each languag

“Repeated” inheritance

```
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

Bibliography

Barbara Liskov.
Data Abstraction and Hierarchy.
OOPSLA'87 Keynote, 1988.