CHAPITRE 12

Héritage et sémantiques des méthodes

POLYTECHNIQUE MONTRÉAL

Définition de la relation de généralisation

Une généralisation est une relation taxonomique (de classement) entre un élément plus général et un élément plus spécifique.

L'élément plus spécifique est complètement compatible avec l'élément plus général (il en a toutes les propriétés, membres et relations) et peut contenir des informations additionnelles.

C'est le principe de substitution de Liskov,

- Le 'L' des principes SOLID.
- Un objet de la classe de base peut être remplacé par un objet de la sous-classe sans casser l'application.

Relation de généralisation

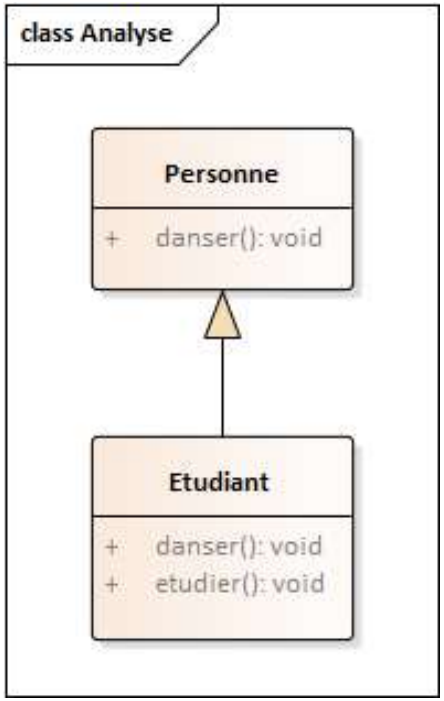
Une relation de généralisation doit modéliser une relation du type «est un».

Le C++ implante ce type de comportement.

```
class Personne {
  void danser( void );   // tout le monde peut danser
};
class Etudiant : public Personne {
  void etudier( void ); // seules les étudiants étudient
};

Personne p; Etudiant e;

p.danser();   // Ok
e.danser();   // Ok
e.etudier();  // Ok
p.etudier();  // Erreur !
```



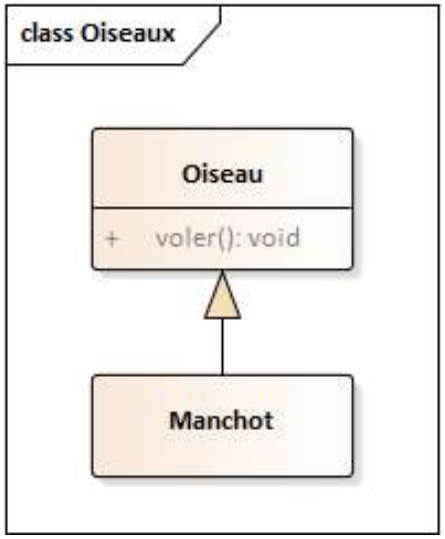
Relation de généralisation

L'héritage publique et l'intuition. En langage courant on dira:

- Les oiseaux volent,
- Les manchots sont des oiseaux.

Traduit en classes on a:

```
class Oiseau {
public:
  virtual void voler();
};
class Manchot : public Oiseau {
...
};
```



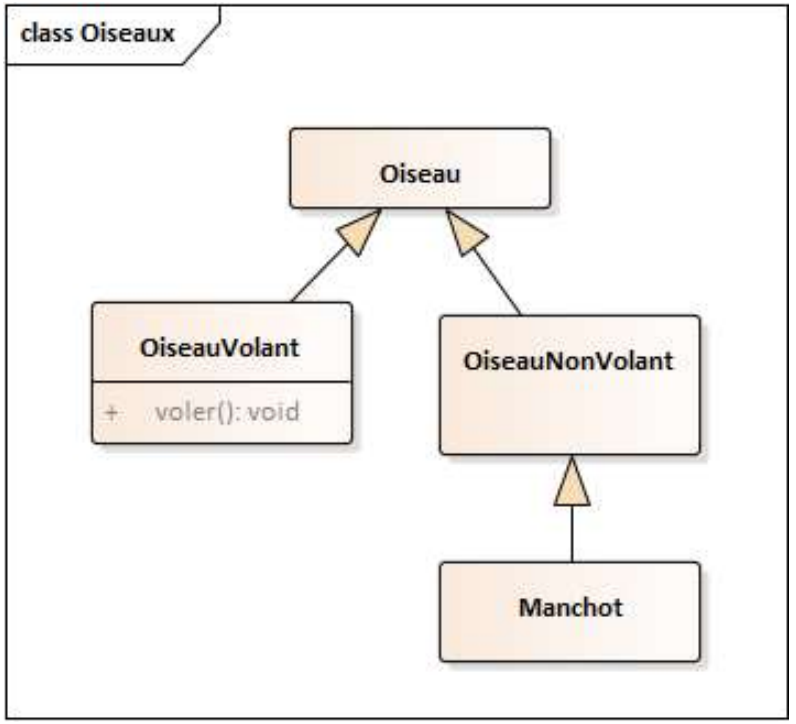
ATTENTION: Les manchots ne volent pas

POLYTECHNIQUE MONTRÉAL

Relation de généralisation

Il faut être plus précis:

```
class Oiseau { ... }; // pas de méthode voler()
class OiseauVolant : public Oiseau {
public:
  virtual void voler();
};
class OiseauNonVolant : public Oiseau {
...
};
class Manchot : public OiseauNonVolant {
...
};
```



Relation de généralisation

Est-ce que la seconde hiérarchie de classes est vraiment meilleure ?

- Pour un client de la classe Oiseau qui ne se soucie pas de faire voler les oiseaux, les deux conceptions sont équivalentes,
  - Il n'y a pas un seul bon design qui satisfait toutes les applications, même dans un domaine de problème précis,
- Pour beaucoup d'applications, le raffinement de la hiérarchie en quatre classes est trop complexe, et sera plus difficile à comprendre et à entretenir.

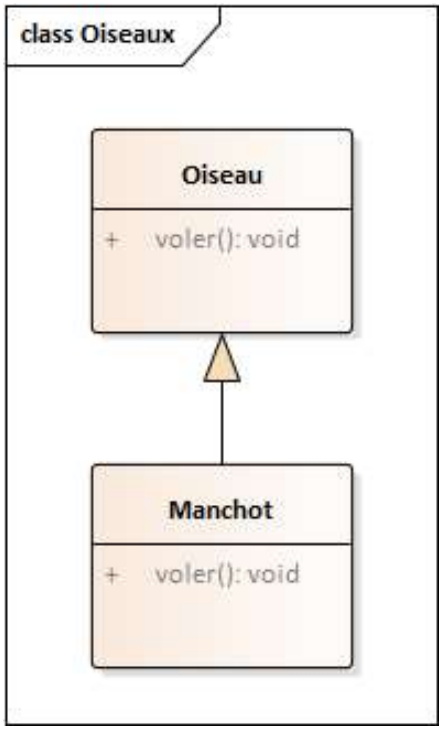
POLYTECHNIQUE MONTRÉAL

Relation de généralisation

Une autre approche de solution au problème:

```
void erreur(const char* msg);    //définie ailleurs

class Oiseau {
public:
  virtual void voler();
  ...
};
class Manchot : public Oiseau {
public:
  virtual void voler()
  { erreur("Les manchots ne volent pas"); };
  ...
};
```



Relation de généralisation

Erreurs de compilation vs. erreurs à l'exécution:

Il y a une différence importante entre les deux approches,

- La seconde approche (propre aux langages interprétés comme Smalltalk) ne dit pas que les manchots ne peuvent pas voler,
  - Elle dit: « les manchots peuvent voler, mais c'est une erreur pour eux d'essayer de le faire »
- L'approche utilisant quatre classes dit « les manchots ne peuvent pas voler, point à la ligne »,
  - C'est le compilateur qui va s'assurer que les manchots ne volent pas.

Relation de généralisation

Erreurs de compilation vs. erreurs à l'exécution:

La détection des erreurs à la compilation est généralement supérieure à la détection des erreurs lors de l'exécution.

- Simplifie la vérification des programmes,
- N'implique aucun coût de gestion durant l'exécution pour la détection d'erreurs.

POLYTECHNIQUE MONTRÉAL

Relation de généralisation

Multiplication des caractéristiques: que faire si l'on doit tenir compte de nombreuses caractéristiques sans liens entre-elles ?

Une hiérarchie d'oiseaux:

- Volant vs. Non volant
- Granivore vs. Insectivore
- Migrateur vs. Sédentaire
- Etc.

La multiplication des caractéristiques va entraîner une hiérarchie de classes très profonde.

POLYTECHNIQUE MONTRÉAL

Relation de généralisation

Deux solutions possibles:

1. Remplacer l'héritage par des agrégations. La classe englobante doit publier une interface qui rend accessibles les caractéristiques.
2. Utiliser des interfaces et de l'héritage multiple. Support variable selon les langages de programmation.

Une classe doit se concentrer sur un petit ensemble de responsabilités essentielles et déléguer les autres responsabilités.

- Le 'S' des principes SOLID.

POLYTECHNIQUE MONTRÉAL

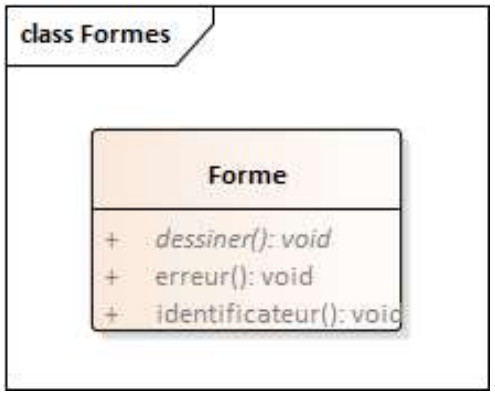
Héritage de l'**interface** vs. Héritage de l'**implémentation**

- **Héritage de l'interface**: correspond à la déclaration des méthodes,
  - Mène à la réutilisation de la conception,
- **Héritage de l'implémentation**: correspond à la définition des méthodes,
  - Mène à la réutilisation du code,

POLYTECHNIQUE MONTRÉAL

Héritage de l'**interface** vs Héritage de l'**implémentation**

```
class Forme {
 public:
   virtual void dessiner() const = 0;
   virtual void erreur( const char* msg );
   int identificateur() const;
   ...
 };
```



Trois types de méthodes :

- dessiner() est une méthode virtuelle pure,
- erreur() est une méthode virtuelle (ordinaire, non pure),
- identificateur() est une méthode non virtuelle.

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles pures

Une méthode virtuelle pure spécifie l'héritage de l'**interface** seulement:

- Raisonnable pour **`Forme::dessiner`** – comment peut-on écrire du code pour dessiner une forme sans la connaître ?
- Doit être interprété par les concepteurs de sous-classes comme: « Vous devez fournir une méthode **`dessiner`**, mais je ne sais pas comment vous allez l'implémenter ».
- Permet aux concepteurs des classes de base de spécifier la fonctionnalité obligatoire des sous-classes sans spécifier comment cette fonctionnalité doit être rendue disponible.

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles ordinaire (non pure)

Une méthode virtuelle ordinaire spécifie l'héritage de l'interface plus l'héritage d'une implémentation par défaut

- La méthode **`Forme::erreur`** fournit un mécanisme de traitement des erreurs par défaut,
- Les sous-classes peuvent remplacer le comportement de défaut si elles le veulent,
- Doit être interprété par les concepteurs de sous-classes comme: «Vous devez fournir une méthode **`erreur`**, mais vous n'êtes pas obligé de l'écrire vous-même si celle qui vous est fournie vous convient».

Sémantique des opérations: méthodes virtuelles

Le couplage entre l'interface obligatoire et l'implémentation par défaut peut-être dangereux:

```
class Aeroport { ... };
class Avion
{
public:
  virtual void atteindre( const Aeroport& dest );
  ...
};
```

Supposons qu'il n'y a que deux sortes d'avions, et que les deux atteignent une destination de la même façon.

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles

On peut définir la méthode atteindre directement au niveau de la classe Avion:

```
void Avion::atteindre( const Aeroport& dest )
{
  // Méthode de défaut pour atteindre un aéroport
};

// Le RJ ne redéfinit pas la méthode atteindre
class RegionalJet : public Avion { ... };

// Le Global non plus
class GlobalExpress : public Avion { ... };
```

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles

On peut utiliser la méthode **`atteindre`** de la même façon quelque soit le type d'avion:

```
void piloter( Avion& un_avion, const Aeroport& dest )
{
  // Utiliser la méthode atteindre vers un aéroport
  un_avion.atteindre( dest );
};
```

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles

Que se passe-t-il si un modèle d'avion complètement différent est ajouté au système ?

```
class CL_415 : public Avion {
  /* pas de définition pour atteindre */
};
```

Mais un CL_415 ne se pilote pas du tout comme un avion de ligne!

```
Aeroport roberval;
Avion* au_feu = new CL_415;
piloter(*au_feu, roberval );
```

Va-t-il se rendre ?

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles ordinaires

Une meilleure approche consiste à fournir un comportement par défaut aux sous-classes, en les forçant explicitement à l'utiliser:

```
class Avion {
public:
  virtual void atteindre( const Aeroport& dest ) = 0;
  ...
protected:
  void atteindreDefaut( const Aeroport& dest );
};

void Avion::atteindreDefaut( const Aeroport& dest )
{
 // Méthode par défaut pour atteindre un aéroport
};
```

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles

Les sous-classes sont obligées de fournir explicitement une implantation pour la méthode **`atteindre`**, mais peuvent tout de même utiliser la méthode de défaut :

```
class RegionalJet : public Avion {
  virtual void atteindre( const Aeroport& dest )
  { atteindreDefaut(dest); };
};

class GlobalExpress : public Avion {
  virtual void atteindre( const Aeroport& dest )
  { atteindreDefaut(dest); };
};
```

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes virtuelles

Le concepteur de la classe CL_415 est maintenant forcé de se demander si l'implantation par défaut est utilisable :

```
class CL_415 : public Avion
{
  virtual void atteindre( const Aeroport& dest )
  {
    /* devrait y penser ... */
  };
};
```

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: fonctions virtuelles « impures »

Les méthodes qui fournissent l'implantation par défaut:

- Ce sont des détails d'implantation, elles devraient donc être **protected**.
- Elles ne devraient **jamais** être surchargées, elles ne sont donc pas virtuelles:
  - Les rendre virtuelles reproduit le problème que l'on essayait de corriger: que se passe-t-il si quelqu'un oublie de les redéfinir alors qu'on devrait ?

Éviter la redéfinition d'une méthode virtuelle

Dans plusieurs langages, il est possible d'empêcher la surcharge de méthodes virtuelles dans des sous-classes:

- En C++ 2011, une méthode virtuelle peut être déclarée `final`:

```
class Base {
public:
    virtual bool resultatGaranti() final { return true; }
};

class Enfant : public Base {
public:
    bool resultatGaranti() { return false; } /* tente de
                                        changer le resultat */
};
```

  - Produit une erreur de compilation
- C'est aussi possible en Java avec le mot-clé `final`
- En C#, on utilise le mot-clé `sealed`

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes non virtuelles

Les méthodes non virtuelles spécifient l'héritage d'une interface et l'héritage d'une implémentation obligatoire:

- Ce sont des fonctions qui sont invariantes en cours de généralisation,
- La méthode **`identificateur()`** doit être implémentée de la même façon pour toutes les formes,
- Doit être interprété par les concepteurs de sous-classes comme: « Vous devez fournir une fonction **`identificateur()`**, et vous devez utiliser celle qui est fournie ».

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes non virtuelles

Que se passe-t-il si on surcharge une méthode non virtuelle ?

```
class A
{
public:
  void uneMethode();
};

class B : public A
{
public:
  void uneMethode();
};

B x;          // x est un objet de type B
A* pA = &x; // récupérer un pointeur sur x
B* pB = &x; // récupérer un autre pointeur sur x
```

Sémantique des opérations: méthodes non virtuelles

On s'attendrait à ce que les deux énoncés suivants donnent le même résultat:

```
pa->uneMethode();
pb->uneMethode();
```

Mais ce ne sera pas le cas.

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: méthodes non virtuelles

Les appels aux fonctions non-virtuelles sont résolus au moment de la compilation:

- Le compilateur détermine quelle fonction appeler en se basant sur le type du pointeur à l'objet, et non sur le type de l'objet lui-même,
- Si la classe B définit sa propre version de la méthode uneMethode qui est non virtuelle dans la classe de base A, alors, bien que pa et pb pointent sur le même objet,

```
pa->uneMethode(); // appelle A::uneMethode()
pb->uneMethode(); // appelle B::uneMethode()
```

- Cette règle s'applique aussi aux références

IL NE FAUT JAMAIS SURCHARGER UNE MÉTHODE NON-VIRTUELLE

Sémantique des opérations: retour sur les méthodes virtuelles

Que se passe-t-il si on change les méthodes pour les rendre virtuelles ?

```
class A
{
public:
  virtual void uneMethode();
};

class B : public A
{
public:
  virtual void uneMethode() override;
};

B x;          // x est un objet de type B
A* pA = &x; // récupérer un pointeur sur x
B* pB = &x; // récupérer un autre pointeur sur x
```

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: retour sur les méthodes virtuelles

On s'attendrait à ce que les deux énoncés suivants donnent le même résultat:

```
pa->uneMethode();
pb->uneMethode();
```

Et ce sera le cas!

POLYTECHNIQUE MONTRÉAL

Sémantique des opérations: retour sur les méthodes virtuelles

Les appels aux méthodes virtuelles sont résolus au moment de l'exécution:

- Le compilateur détermine quelle fonction appeler en se basant sur le pointeur ***vptr*** contenu dans l'objet, qui dépend du type de l'objet lui-même,
- Si la classe B définit sa propre version de la méthode uneMethode qui est virtuelle dans la classe de base A, alors, pa et pb pointent sur le même objet, donc le vptr est le même et sert à déterminer la fonction à appeler

```
pa->uneMethode(); // appelle B::uneMethode()
pb->uneMethode(); // appelle B::uneMethode()
```

- Cette règle s'applique aussi aux références

UNE MÉTHODE QUI DOIT POUVOIR ÊTRE REDÉFINIE DANS LES SOUS-CLASSES DOIT ÊTRE DÉCLARÉE VIRTUELLE DANS LA CLASSE DE BASE

Héritage et destructeurs virtuels

Les destructeurs dans les classes de base devraient toujours être déclarés virtuels.

Voyons ce qui arrive dans le cas contraire:

```
class CibleEnnemie
{
private:
   static unsigned nbCibles; // compte les cibles

public:
   CibleEnnemie() { ++nbCibles; };
   ~CibleEnnemie() { --nbCibles; };

   static unsigned getNombreDeCibles()
   { return nbCibles };
};
```

POLYTECHNIQUE MONTRÉAL

Héritage et destructeurs virtuels

Définition d'une classe dérivée de CibleEnnemie:

```
class CharDAssaut : public CibleEnnemie
{
private:
    static unsigned nbChars; // compte les chars

public:
    CharDAssaut() : CibleEnnemie()
    { ++nbChars; };

    ~CharDAssaut()
    { --nbChars; };

    static unsigned getNombreDeChars()
    { return nbChars };
};
```

POLYTECHNIQUE MONTRÉAL

Héritage et destructeurs virtuels

Supposons maintenant l'utilisation des classes CibleEnnemie et CharDAssaut suivante:

```
CibleEnnemie* c = new CharDAssaut();
...
delete c;
```

Quel destructeur sera appelé ?

Héritage et destructeurs virtuels

- Étant donné que le destructeur de la classe `CibleEnnemie` n'est pas virtuel, le compilateur se base sur le type du pointeur `c` plutôt que le type de l'objet pointé par `c`.
  - Seul le destructeur de `CibleEnnemie` sera appelé.
  - Le compteur `nbChars` ne sera plus à jour.
  - Ça fait un bogue difficile à trouver !
- **Les destructeurs de classes de base doivent toujours être virtuels.**

POLYTECHNIQUE MONTRÉAL

Héritage et destructeurs virtuels

```
class CibleEnnemie {
public:
    ...
    CibleEnnemie() { ++nbCibles; };
    virtual ~CibleEnnemie() { --nbCibles; };
    ...
};

Class CharDAssaut : public CibleEnnemie {
    ...
};


CibleEnnemie* cible = new CharDAssaut;
...
delete cible; // Maintenant, les deux destructeurs
              // seront appelés
```

POLYTECHNIQUE MONTRÉAL

Héritage et destructeurs virtuels

Pourquoi le C++ ne déclare-t'il pas implicitement tous les destructeurs virtuels ?

```
class Point {
 private:
    short int x, y;
 public:
     Point( short int xc, short int yc);
     ~Point();
 };
```

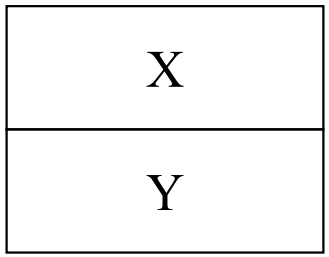
Tel que déclaré ici, un point entre dans un registre de 32bits,

Un point peut être passé tel quel à d'autres langages comme C ou FORTRAN.

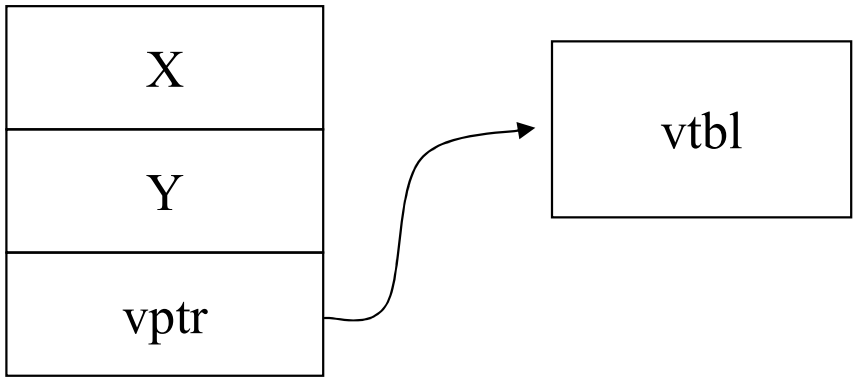
POLYTECHNIQUE MONTRÉAL

Héritage et destructeurs virtuels

Si la classe Point contient une fonction virtuelle, la disposition des objets Point en mémoire change:



Sans fonctions virtuelles



Avec fonctions virtuelles

Taille d'un objet Point sans fonction virtuelle: 32 bits

Taille d'un objet Point avec fonction virtuelle: 64 bits

➔ L'objet vient de doubler de taille et n'est plus compatible avec le C ou le FORTRAN.

POLYTECHNIQUE MONTRÉAL

Héritage et destructeurs virtuels

Le destructeur devrait être virtuel si:

- La classe est prévue pour être utilisée comme classe de base,
- La classe contient d'autre fonctions virtuelles.

Le destructeur ne devrait pas être déclaré virtuel si:

- La classe n'est pas prévue pour être utilisée comme classe de base (c'est une façon de communiquer cette idée aux autres développeurs),
- Si la structure de données doit être compatible avec d'autres langages.