# Module Programmation Objet

## Chapitre 4 - Typage et Héritage Multiple

Pascal André, Gilles Ardourel

Université de Nantes

2010

Projet DVD Miage



# Refactoring

Coder vite sans se soucier de demain c'est prendre un risque
Métaphore du prêt :

# Refactoring

Coder vite sans se soucier de demain c'est prendre un risque
Métaphore du prêt :

- On veut le résultat *tout de suite*

# Refactoring

Coder vite sans se soucier de demain c'est prendre un risque
Métaphore du prêt :

- On veut le résultat *tout de suite*
- On paye les intérêts en maintenance

# Refactoring

Coder vite sans se soucier de demain c'est prendre un risque
Métaphore du prêt :

- On veut le résultat *tout de suite*
- On paye les intérêts en maintenance
- En cas de surendettement, on fait faillite

# Refactoring

Coder vite sans se soucier de demain c'est prendre un risque
Métaphore du prêt :

- On veut le résultat *tout de suite*
- On paye les intérêts en maintenance
- En cas de surendettement, on fait faillite

$\Rightarrow$ On renégocie son prêt, on fait des remboursements anticipés quand on le peut (*après une release*).

# Refactoring : types

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

# Refactoring : types

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- lorsque l'on veut qu'une méthode renvoie plusieurs valeurs

# Refactoring : types

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- lorsqu'on utilise des paramètres "en groupe"

# Refactoring : types

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- lorsqu'on utilise des paramètres "en groupe"
- lorsqu'une méthode devrait être appliquée à plusieurs types d'objets : un supertype commun

# Refactoring : types

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- lorsqu'on utilise des paramètres "en groupe"
- lorsqu'une méthode devrait être appliquée à plusieurs types d'objets : un supertype commun
- lorsqu'on peut factoriser du code

# Refactoring : méthodes

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.
Extraire la partie générique des opérations

# Refactoring : méthodes

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.
Extraire la partie générique des opérations

- trop longues

# Refactoring : méthodes

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.
Extraire la partie générique des opérations

- trop longues
- intérieur des boucles

# Refactoring : méthodes

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.
Extraire la partie générique des opérations

- trop longues
- intérieur des boucles
- créations d'objets

# Refactoring : méthodes

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.
Extraire la partie générique des opérations

- trop longues
- intérieur des boucles
- créations d'objets

Procédure incrémentale : *cent fois sur le métier. . .*
Si toutes les méthodes ne sont pas implantées, la classe est abstraite.

# Classe abstraite

La généralisation ou la factorisation peut faire apparaître des classes abstraites.
Une classe abstraite ne peut avoir d'instances. Car

- trop générale (animal, nombre, comparable)
- ou possédant des méthodes non exprimables *abstraites*

Toutes ses méthodes ne sont pas forcément abstraites.
Ex : La méthode `supérieur` de la classe `Comparable` peut se définir à partir des méthodes abstraites `inférieur` et `égal`, qui ne peuvent être définies que dans les sous-classes de `Comparable`.
Une classe héritant d'une classe abstraite doit

- soit donner un corps à toutes les méthodes abstraites dont elle hérite,
- soit être abstraite elle-même.

# Caractéristiques

Une classe abstraite:

- définit une interface commune à un ensemble de sous-classes, sans donner son implémentation .
- spécifie des attributs communs à ses sous-classes.
- spécifie des méthodes abstraites et assure que ses sous-classes (concrètes) implémenteront ces méthodes.
- spécifie des méthodes concrètes, qui implémentent partiellement son comportement.

# Motivation

Utilisation des classes abstraites:

- Lorsque l'on souhaite utiliser une super-classe, sans permettre que celle-ci soit instanciable.
- Par exemple, la classe **Humain** et ses sous-classes **Homme** et **Femme**.
- **Humain** définit les propriétés communes de ses sous-classes, mais ne peut pas être instancié.

# Motivation

Utilisation des classes abstraites:

- Lorsque l'on souhaite définir une interface commune et forcer les sous-classes à l'implémenter.
- Par exemple, la classe **Forme** et ses sous-classes **Cercle** et **Rectangle**.
- **Forme** peut déclarer la méthode **aire()**, mais ne peut pas l'implémenter. L'implémentation sera réalisée par ses sous classes.

# Syntaxe

## C++

- En C++, une classe abstraite est une classe qui spécifie au moins une méthode abstraite, appelée *virtuelle pure*.

```
class <cname> {
public:
  virtual <type> <mname>(<plist>) = 0;
}

class DrawingElement {
public:
  virtual void display () = 0;
  virtual void translate () = 0;
}
```

# Syntaxe

## Java

- En Java, le mot-clé **abstract** est utilisé pour déclarer une classe abstraite.
- Le même mot-clé est utilisé pour déclarer des méthodes abstraites.

```
abstract class <cname> {
< visibility > abstract <type> <mname>(<plist>);
}

abstract class DrawingElement {
  public abstract void display ();
  public abstract void translate ();
}
```

# Syntaxe

## Eiffel

- En Eiffel, le mot-clé **deferred** est utilisé pour déclarer une classe abstraite.
- Le même mot-clé est utilisé à la place du corps dans les méthodes abstraites.

```
deferred class DrawingElement
feature
  display is deferred ;
  translate is deferred ;
}
```

# Attributs

- Une classe abstraite peut avoir des attributs, qui seront hérités par ses sous-classes.

# Méthodes

- Une classe abstraite peut contenir des méthodes normales et des méthodes abstraites.
- Les méthodes normales sont héritées par les sous-classes, qui peuvent les redéfinir.
- Les méthodes abstraites doivent être implémentées par les sous-classes.
- Si une sous-classe n'implémente pas une méthode abstraite, elle sera elle aussi une classe abstraite.

# Constructeurs

- Une classe abstraite peut définir des constructeurs, qui seront utilisés par ses sous-classes.

# Héritage multiple

En C++ et en Eiffel, une classe peut avoir plusieurs super-classes

Avantage : plus de code hérité (en faisant attention à la spécialisation)

Inconvénients : conflits potentiels

- conflit de nom (département géographique ou administratif)

# Héritage multiple

En C++ et en Eiffel, une classe peut avoir plusieurs super-classes

Avantage : plus de code hérité (en faisant attention à la spécialisation)

Inconvénients : conflits potentiels

- conflit de nom (département géographique ou administratif)
  $\Rightarrow$ renommage en amont
  - suppose l'accès au code
  - impact sur les clients

# Héritage multiple

En C++ et en Eiffel, une classe peut avoir plusieurs super-classes

Avantage : plus de code hérité (en faisant attention à la spécialisation)

Inconvénients : conflits potentiels

- conflit de nom (département géographique ou administratif)
  $\Rightarrow$ renommage en amont
  - suppose l'accès au code
  - impact sur les clients
- conflit de valeur (comment choisir la bonne)

# Héritage multiple

En C++ et en Eiffel, une classe peut avoir plusieurs super-classes

Avantage : plus de code hérité (en faisant attention à la spécialisation)

Inconvénients : conflits potentiels

- conflit de nom (département géographique ou administratif)
  ⇒ renommage en amont
  - suppose l'accès au code
  - impact sur les clients
- conflit de valeur (comment choisir la bonne)
  ⇒ désambiguisation
  - désignation explicite (C++) ("casse" la liaison tardive et le principe de découplage)
    grâce à l'opérateur de résolution de portée : A::prop et B::prop
  - renommage local des méthodes en conflit et sélection de celle retenue pour la liaison tardive (Eiffel).

# Héritage multiple

En C++ et en Eiffel, une classe peut avoir plusieurs super-classes

Avantage : plus de code hérité (en faisant attention à la spécialisation)

Inconvénients : conflits potentiels

- conflit de nom (département géographique ou administratif)
  ⇒ renommage en amont
  - suppose l'accès au code
  - impact sur les clients
- conflit de valeur (comment choisir la bonne)
  ⇒ désambiguisation
  - désignation explicite (C++) ("casse" la liaison tardive et le principe de découplage)
    grâce à l'opérateur de résolution de portée : A::prop et B::prop
  - renommage local des méthodes en conflit et sélection de celle retenue pour la liaison tardive (Eiffel).

La solution de Java : interdiction de l'héritage multiple

# Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code.
En Java, on peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types: les interfaces.

# Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code.
En Java, on peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types: les interfaces.

- pas d'attributs

# Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code.
En Java, on peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types: les interfaces.

- pas d'attributs
- pas de constructeurs

# Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code.
En Java, on peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types: les interfaces.

- pas d'attributs
- pas de constructeurs
- seulement des méthodes publiques

# Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.
Ex: `public class Etudiant extends Personne implements Localise`

# Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.
Ex: `public class Etudiant extends Personne implements Localise`
La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

# Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.
Ex: `public class Etudiant extends Personne implements Localise`
La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- soit en définissant la méthode

# Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.
Ex: `public class Etudiant extends Personne implements Localise`
La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- soit en définissant la méthode
- soit en en héritant

# Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.
Ex: `public class Etudiant extends Personne implements Localise`
La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- soit en définissant la méthode
- soit en en héritant

$\Rightarrow$ Toute classe possède les types de ses superclasses.

# Extensibilité

## *Extensibilité*

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.
Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés.

# Extensibilité

## *Extensibilité*

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.
Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés. Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

# Extensibilité

## *Extensibilité*

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.
Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés. Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- récupération de code pas forcément utile

# Extensibilité

## *Extensibilité*

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.
Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés. Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- récupération de code pas forcément utile
- si héritage simple, la meilleure superclasse n'est pas forcément celle utilisée

# Extensibilité

## *Extensibilité*

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.
Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés. Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- récupération de code pas forcément utile
- si héritage simple, la meilleure superclasse n'est pas forcément celle utilisée

A contrario, l'implémentation ne coûte presque rien. Il est facile de créer des objets qui sont du type d'une interface

# Utilisation d'interfaces

La décision d'utiliser des interfaces est soit prévisionnelle, soit un rafinement.

# Utilisation d'interfaces

La décision d'utiliser des interfaces est soit prévisionnelle, soit un rafinement.

1. création d'interface

   regroupant des méthodes fréquemment rencontrées et utilisées de concert

# Utilisation d'interfaces

La décision d'utiliser des interfaces est soit prévisionnelle, soit un rafinement.

1. création d'interface

   regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

# Utilisation d'interfaces

La décision d'utiliser des interfaces est soit prévisionnelle, soit un rafinement.

1. création d'interface

   regroupant des méthodes fréquemment rencontrées et utilisées de concert
2. déclaration d'implémentation dans les classes qui possèdent les méthodes
3. utilisation de l'interface plutôt que les classes pour typer

# Utilisation d'interfaces

La décision d'utiliser des interfaces est soit prévisionnelle, soit un rafinement.

1. création d'interface

   regroupant des méthodes fréquemment rencontrées et utilisées de concert
2. déclaration d'implémentation dans les classes qui possèdent les méthodes
3. utilisation de l'interface plutôt que les classes pour typer $\Rightarrow$ clients plus flexibles

# Utilisation d'interfaces

La décision d'utiliser des interfaces est soit prévisionnelle, soit un rafinement.

1. création d'interface

   regroupant des méthodes fréquemment rencontrées et utilisées de concert
2. déclaration d'implémentation dans les classes qui possèdent les méthodes
3. utilisation de l'interface plutôt que les classes pour typer $\Rightarrow$ clients plus flexibles
4. créer une classe abstraite implémentant l'interface

# Utilisation d'interfaces

La décision d'utiliser des interfaces est soit prévisionnelle, soit un rafinement.

1. création d'interface

   regroupant des méthodes fréquemment rencontrées et utilisées de concert
2. déclaration d'implémentation dans les classes qui possèdent les méthodes
3. utilisation de l'interface plutôt que les classes pour typer $\Rightarrow$ clients plus flexibles
4. créer une classe abstraite implémentant l'interface $\Rightarrow$ en prévision d'une factorisation

# Exemple: collections

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

# Exemple: iterators

L'interface Iterator permet de parcourir les éléments d'une collection sans se soucier du type de la collection

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```

Utilisation :

```
static void filter(Collection collect) {
    for (Iterator i = collect.iterator(); i.hasNext(); )
        if (!cond(i.next())) i.remove();
}
```

Une version améliorée sera présentée dans le chapitre 5.

# conclusion

Pour produire un code maintenable (correction et évolution)

- Décomposez, factorisez et rendez le code générique
- Programmez vers les interfaces
- Retravaillez votre code régulièrement