Objet et développement d'applications
Cours 1 - POO et UML

Patrons de conception pour POO

Florian Richoux

2014-2015

Sur quoi porte ce cours ?

Patrons de conception (design patterns)

Pattern : vient des années d'expérience de la communauté des programmeurs.

Solution efficace pour résoudre un problème précis dans un contexte donné.

Sur quoi porte ce cours ?

Efficace ?

Ici, code "efficace" = code facilement...

- extensible / modifiable,
- lisible / compréhensible,
- communicable (vocabulaire commun aux développeurs)

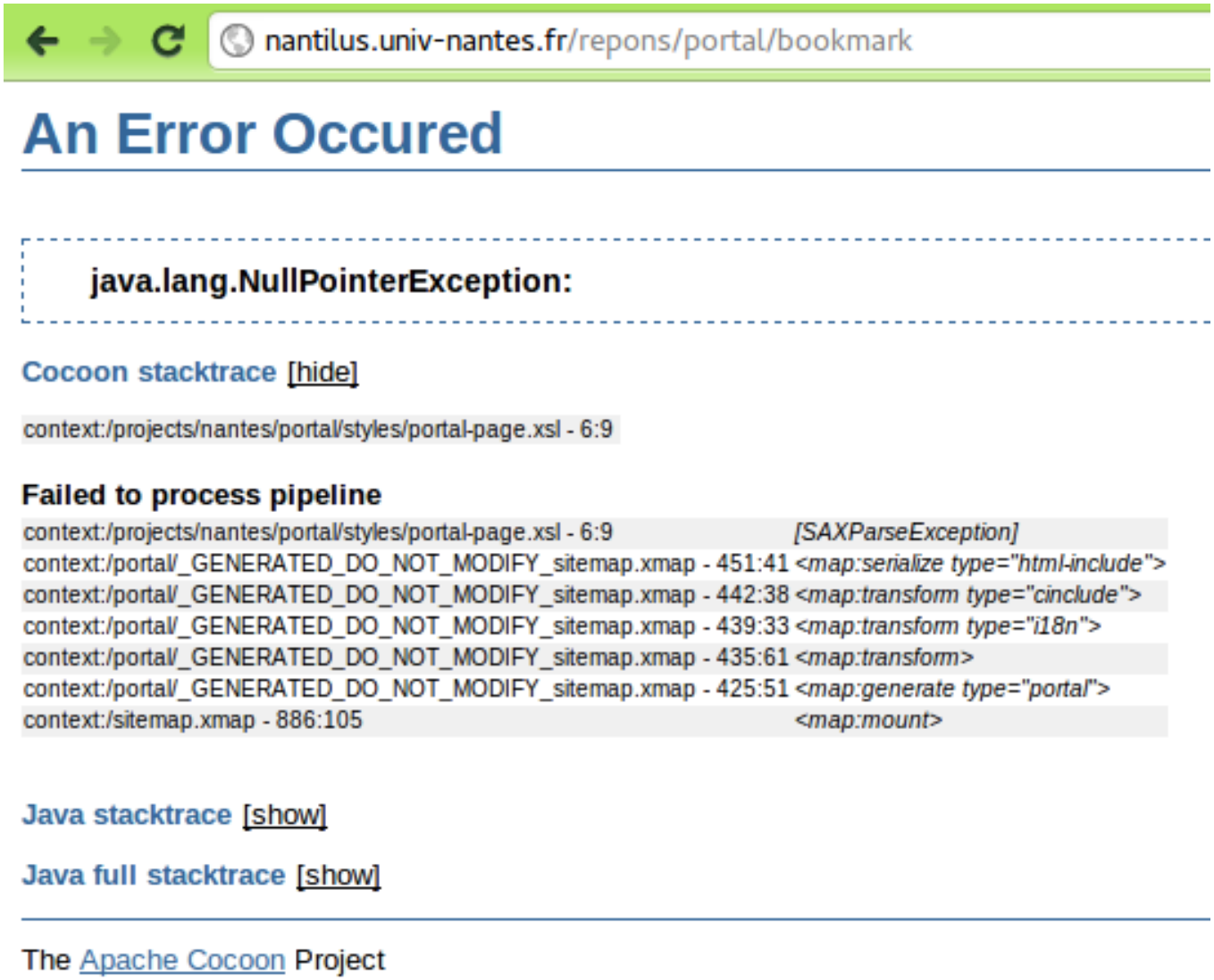
Bref, du code facilement maintenable.

Maintenance

Aujourd'hui, écrire un programme, c'est :

- 10-30% de développement,
- 70-90% de maintenance (post-codage).

Avoir un code robuste



Avoir un code robuste

Petit sondage

Et répondez honnêtement svp

Cours

1^{er} cours

Rappels sur la POO et UML (diagramme de classe).

Autres cours

Objectif : 1 voire 2 nouveaux patterns par cours ($\sim$9 après 9 cours).
Exemples en Java.

Support de cours

Parfait pour débuter

Design Patterns : Tête la première, E. et E. Freeman, ed. O'Reilly.

Plus tard, quand vous serez grand...

Design Patterns, E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka "The Gang of Four").

Apprendre/approfondir Java

Programmer en Java, Claude Delannoy, ed. Eyrolles.

Apprendre/approfondir C++

Programmer en C++, Claude Delannoy, ed. Eyrolles.

Besoin d'aide ? Des questions ?

E-mail

florian.richoux@univ-nantes.fr

LINA, bureau 214, 2^{eme} étage

← Nord

Plan d'aujourd'hui

Plan

I Rappels de la POO en Java

II Rappel ou introduction aux diagrammes de classe d'UML

Plan

Rappels de la POO en Java

Ville

```
class Ville
{
  public Ville(double lat, double lon)
  {
    this.lat = lat;
    this.lon = lon;
  }

  public void info()
  {
    System.out.println("Latitude=" + lat +
    ",␣longitude=" + lon);
  }

  private double lat, lon;
}
```

Ville

```
class Ville                                   Nom de classe
{
  public Ville(double lat, double lon)
  {
    this.lat = lat;
    this.lon = lon;
  }                                           Constructeurs

  public void info()
  {
    System.out.println("Latitude=" + lat +
    ",␣longitude=" + lon);
  }                                           Méthodes

  private double lat, lon;
}                                             Attributs
```

Les permissions

Les 4 types de permissions en Java

- Publique

```
public int nombre;
```

- Protégée

```
protected int nombre;
```

- Privée

```
private int nombre;
```

- "rien"

```
int nombre;
```

Les permissions

Modifier	Class	Package	Subclass	World
public	✔	✔	✔	✔
protected	✔	✔	✔	✘
no modifier	✔	✔	✘	✘
private	✔	✘	✘	✘

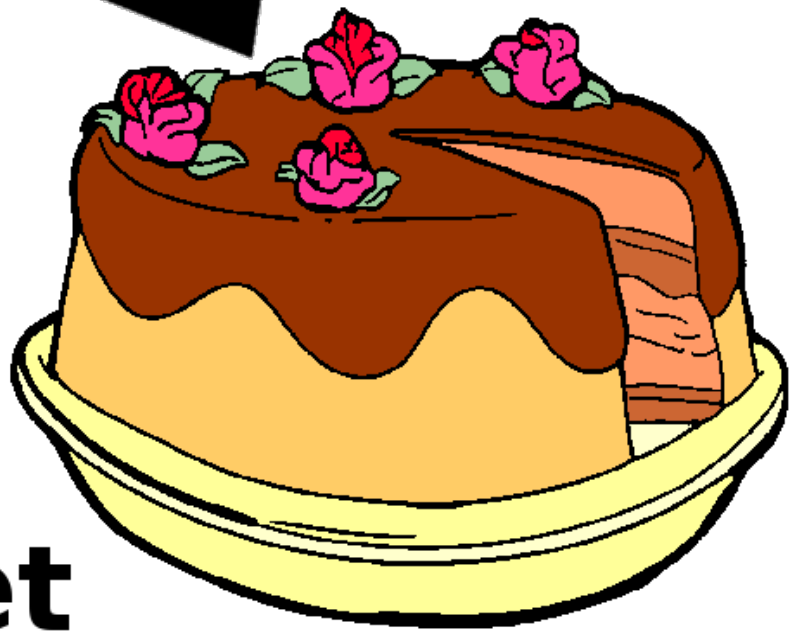
Objets de type Ville

```
class Programme
{
  public static void main(String[] args)
  {
    Ville paris = new Ville(48.8, 2.3);
    Ville nantes = new Ville(47.2, -1.5);
  }
}
```

Les objets

Instanciation

Classe



Objet

Définir sans instancier

Les interfaces

```
interface Animal
{
  void manger(); // méthodes sans implémentation,
                 // toujours public !
  final boolean CARNIVORE = true; // attributs constants
                                  // uniquement
}
```

Une implémentation d'Animal

```
class Chien implements Animal
{
  // On doit implémenter toutes les méthodes
  // de l'interface Animal.
  public void manger()
  {
    System.out.println("Mange de la viande");
  }
}
```

L'héritage

Classe-mère (ou classe de base, ou super-classe)

```
class Personne
{
  public String nom;
  public int numeroSecu;
}
```

Classes-filles (ou classes dérivées)

```
class Etudiant extends Personne
{
  public int numeroEtudiant;
}

class Enseignant extends Personne
{
  public int numeroBureau;
}
```

On hérite des attributs...

```
class Programme
{
  public static void main(String[] args)
  {
    Personne per = new Personne();
    Etudiant etu = new Etudiant();
    Enseignant ens = new Enseignant();

    per.numeroSecu = 123456789;
    etu.nom = "Alice";
    etu.numeroEtudiant = 987654321;
    ens.nom = "Bob";
    ens.numeroBureau = 42;
  }
}
```

Les classes dérivées héritent des attributs, mais aussi des méthodes.

Ville

```
class Ville
{
  protected double lat, lon;

  public Ville(double lat, double lon)
  {
    this.lat = lat;
    this.lon = lon;
  }

  public void info()
  {
    System.out.println("Latitude=" + lat +
    ", longitude=" + lon);
  }
}
```

Capitale

```
class Capitale extends Ville
{
  private String pays_;

  public Capitale(double lat, double lon, String pays)
  {
    super(lat, lon);
    pays_ = pays;
  }

  public void info()
  {
    super.info();
    System.out.println("Pays : " + pays_);
  }
}
```

Compatibilité par affectation

Exemple d'une ville

```
Ville v;
v = new Ville(50.6, 4.6);
```

Compatibilité par affectation

```
Ville v;
v = new Capitale(50.8, 4.3, "Belgique"); // Ok
```

Compatibilité par affectation

Compatibilité ascendante seulement !

```
Ville v = new Ville(50.6, 4.6);
Capitale c = new Capitale(50.8, 4.3, "Belgique");

v = c; // Ok : une capitale est aussi une ville

_____________________

c = v; // Erreur : Ville ne descend pas de Capitale

_____________________

c = (Capitale) v; // Ok à la compilation, mais
                  // à l'exécution v devra référer
                  // un objet de type Capitale,
                  // sinon ClassCastException.
```

Liaison dynamique

Liaison dynamique des méthodes

```
Ville tabVilles[] = new Ville[4];

tabVilles[0] = new Ville(50.6, 4.6);
tabVilles[1] = new Capitale(50.8, 4.3, "Belgique");
tabVilles[2] = new Ville(47.9, 1.9);
tabVilles[3] = new Capitale(35.6, 139.7, "Japon");

for (int i = 0; i < tabVilles.length; i++)
{
  tabVilles[i].info();
}
```

Pour les familiers de C++

En Java, *toutes* les méthodes sont *virtuelles* !

Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

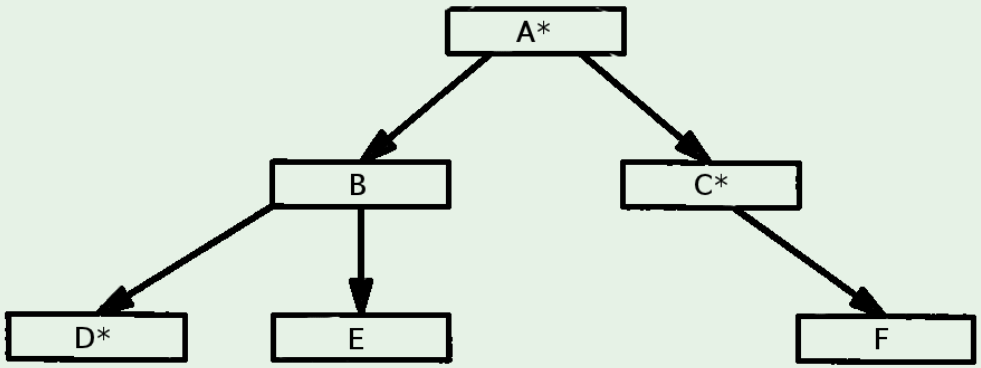
Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

Exemple



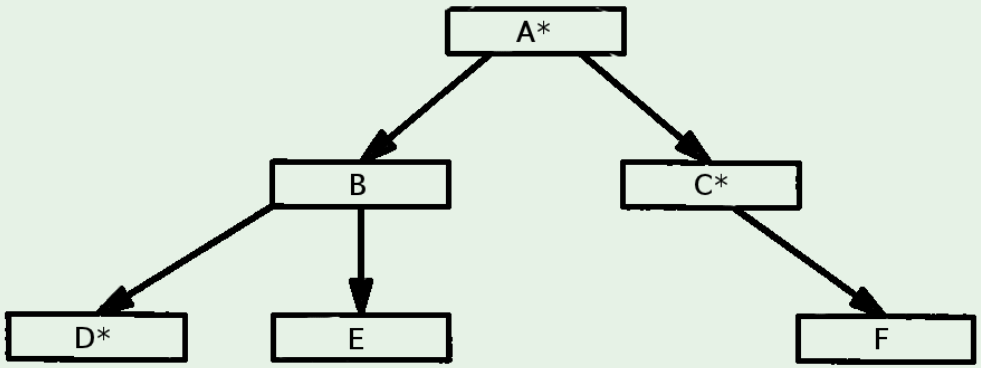
Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

Exemple



- a.m() m de A

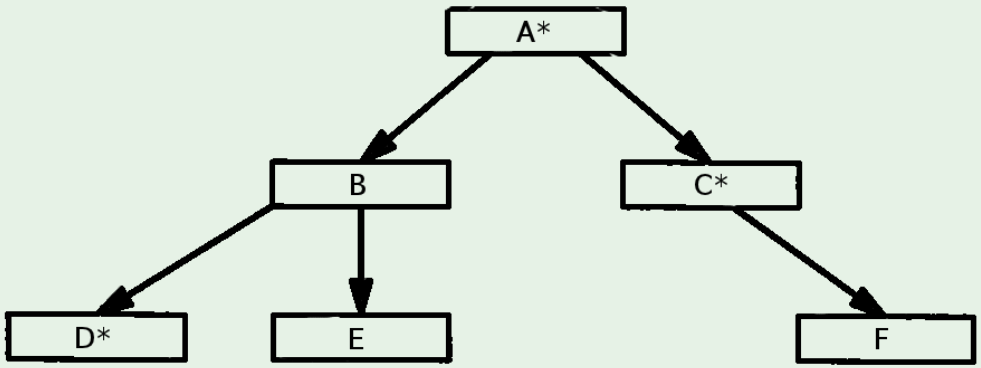
Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

Exemple



- a.m() m de A
- b.m() m de A

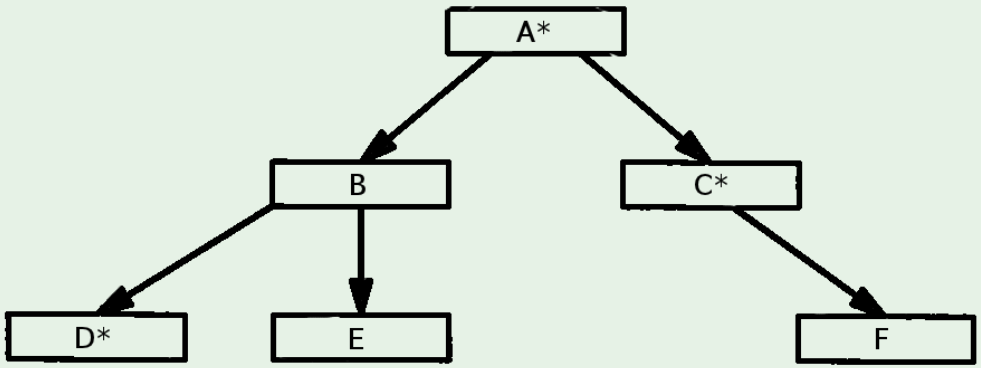
Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

Exemple



- a.m() m de A
- b.m() m de A
- c.m() m de C

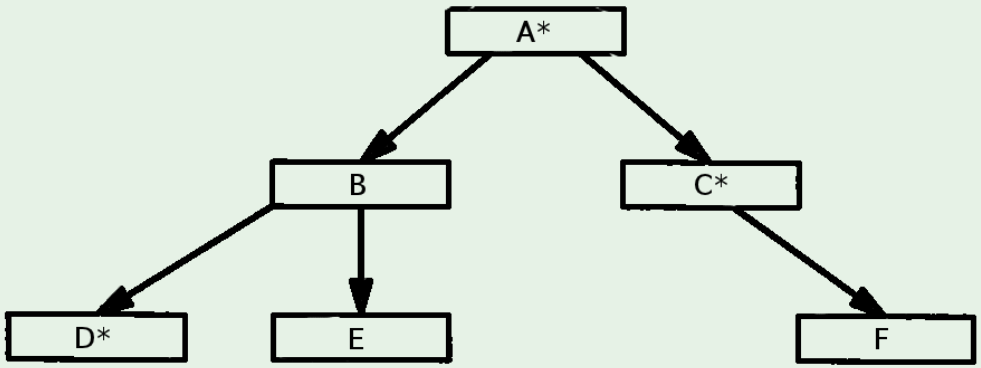
Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

Exemple



- a.m() m de A
- b.m() m de A
- c.m() m de C
- d.m() m de D

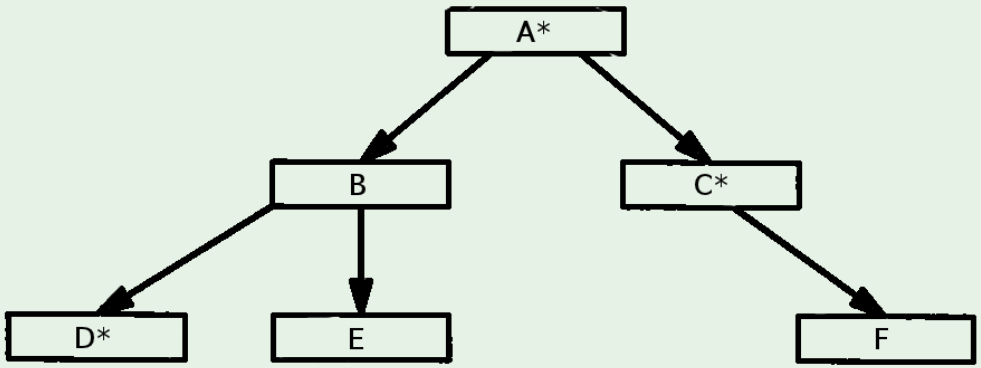
Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

Exemple



- a.m() m de A
- b.m() m de A
- c.m() m de C
- d.m() m de D
- e.m() m de A

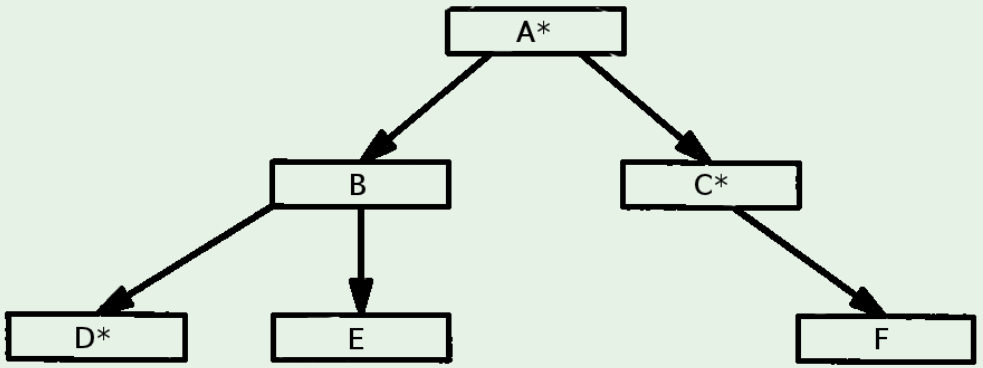
Liaison dynamique

Règle pour la liaison dynamique

On appelle une méthode `m` à partir d'une variable `v` (instruction `v.m()`).

- Si `m` est (re)définie dans la classe de l'objet référencé par `v`, on exécute cette version de `m`.
- Sinon, on remonte dans les classes-mères jusqu'à trouver une définition de `m`.

Exemple



- a.m() m de A
- b.m() m de A
- c.m() m de C
- d.m() m de D
- e.m() m de A
- f.m() m de C

Liaison dynamique et compilation

```
class A
{
  public void f (double x) {...}
  ...
}
class B extends A
{
  public void f (double x) {...} // redéfinition
  public void f (int n) {...}  // surdéfinition
  ...
}
A a = new A(...);
B b = new B(...);
int n;

a.f(n); // appelle f (double) de A
b.f(n); // appelle f (int) de B
a = b;
a.f(n);
```

Liaison dynamique et compilation

```
class A
{
  public void f (double x) {...}
  ...
}
class B extends A
{
  public void f (double x) {...} // redéfinition
  public void f (int n) {...}  // surdéfinition
  ...
}
A a = new A(...);
B b = new B(...);
int n;

a.f(n); // appelle f (double) de A
b.f(n); // appelle f (int) de B
a = b;
a.f(n); // appelle f (double) de B
```

Limites du polymorphisme

```
class Ville
{
   ...
   public boolean identique(Ville x)
   {
      return (lat == x.lat && lon == x.lon);
   }
}
class Capitale extends Ville
{
   ...
   public boolean identique(Capitale x)
   {
      return (super.identique(x) &&
              pays_.equals(x.pays_));
   }
}

Ville v1 = new Capitale(48.8, 2.3, "France");
Ville v2 = new Capitale(48.8, 2.3, "Royaume-Uni");
v1.identique(v2); // Retourne VRAI !
```

UML

Unified Modeling Language : modélisation graphique de projet logiciel.

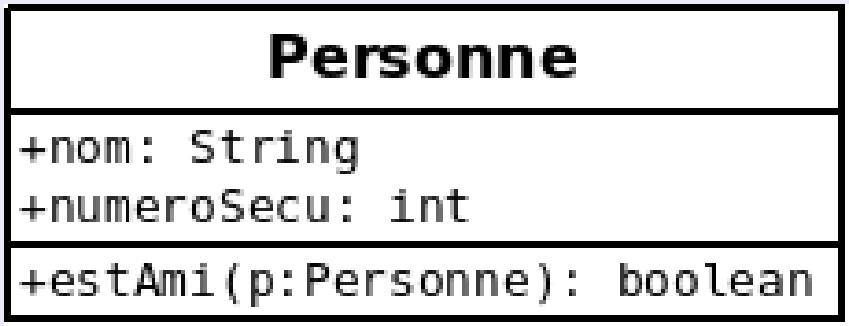
En clair

Schémas définissant et expliquant les composants d'un futur programme.

Pour ce cours

Besoin de ne connaître qu'une chose : le diagramme de classe !

Diagramme de classe



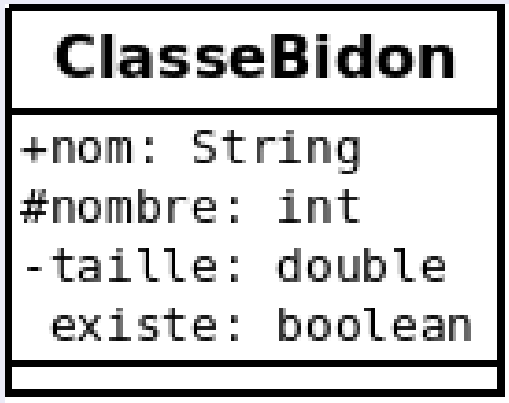
En Java

```java
class Personne
{
  // Attributs
  public String nom;
  public int numeroSecu;

  // Méthodes
  public boolean estAmi(Personne p) = {...};
}
```

Les permissions

Les 4 permissions



En Java

```
class ClasseBidon
{
  public String nom;
  protected int nombre;
  private double taille;
  boolean existe;
}
```

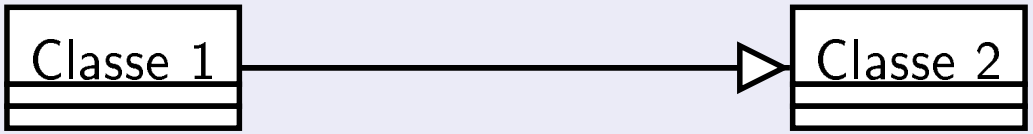
Relations

Pour ce cours : 4 types de relations

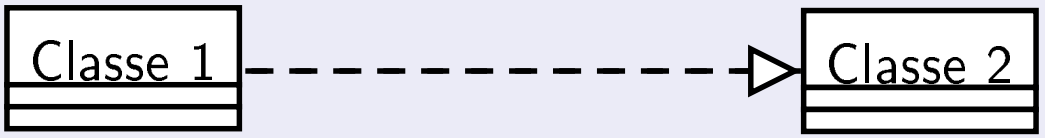
- Association

- Agrégation (Composition)

- Spécialisation (Héritage)

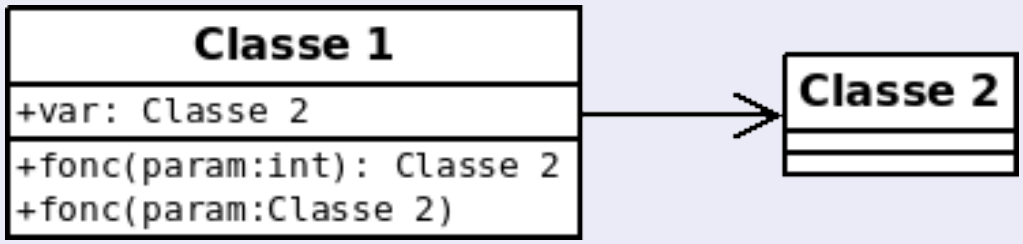


- Réalisation (Implémentation)



Association

Association



En Java

```
class Classe_1
{
  public Classe_2 var;
  public Classe_2 fonc(int param);
  public void fonc(Classe_2 param);
}

class Classe_2
{
  ...
}
```

Agrégation (Composition)

En Java

```
class Roue
{
  public int taille;
}

class Voiture
{
  public Roue roues[] = new Roue[4];
}
```

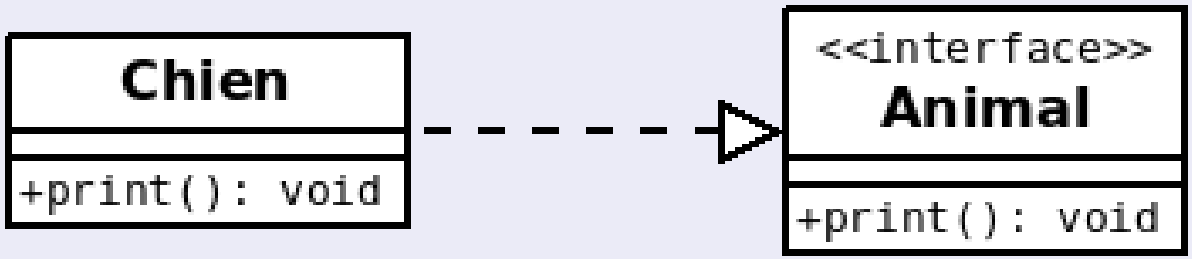
Spécialisation (Héritage)

En Java

```
class Personne
{
  public String nom;
  public int numeroSecu;
}

class Etudiant extends Personne
{
  public int numeroEtudiant;
}
```

Réalisation (Implémentation)



En Java

```
interface Animal
{
  public void print();
}

class Chien implements Animal
{
  public void print()
  {
    System.out.println("Waf waf !");
  }
}
```

Déroulement de ce cours

Concernant l'évaluation

3 notes

Contrôle écrit (25%) + Projet (25%) + Examen (50%)

Contrôle écrit

- Quelque part début novembre, à la place d'un TD.
- À connaître : tout depuis le début jusqu'à la veille du CC.
- Supports de cours / TD / TP autorisés. Tout le reste interdit.

Projet

- Binômes fixés dans deux semaines, avec une idée de projet (je ferai passer une feuille le cours prochain).
- Rapport de 10 pages max à me rendre dans mon casier le jeudi 27 novembre.
- Présentation du projet début décembre.

Concernant le projet

Projet

- Groupes en binômes.

Concernant le projet

Projet

- Groupes en binômes.
- 3 patterns d'au moins deux familles différentes à utiliser, au choix. Si vous utilisez un ou plusieurs patterns qui ne seront pas vu en cours, m'envoyer un email **avant** de vous lancer dans la programmation.

Concernant le projet

Projet

- Groupes en binômes.
- 3 patterns d'au moins deux familles différentes à utiliser, au choix. Si vous utilisez un ou plusieurs patterns qui ne seront pas vu en cours, m'envoyer un email **avant** de vous lancer dans la programmation.
- Liste des patterns qu'on devrait voir en détail : Strategy, State, Observer, Decorator, Adapter, ~~Façade~~, Factory Method, Abstract Factory, ~~Singleton~~.

Concernant le projet

Projet

- Groupes en binômes.
- 3 patterns d'au moins deux familles différentes à utiliser, au choix. Si vous utilisez un ou plusieurs patterns qui ne seront pas vu en cours, m'envoyer un email **avant** de vous lancer dans la programmation.
- Liste des patterns qu'on devrait voir en détail : Strategy, State, Observer, Decorator, Adapter, ~~Façade~~, Factory Method, Abstract Factory, ~~Singleton~~.
- Liste des patterns qu'on devrait voir rapidement : Command, Chain of Responsability, Visitor, Composite, Bridge, Proxy, Builder, Object Pool.

Concernant le projet

Projet

- Groupes en binômes.
- 3 patterns d'au moins deux familles différentes à utiliser, au choix. Si vous utilisez un ou plusieurs patterns qui ne seront pas vu en cours, m'envoyer un email **avant** de vous lancer dans la programmation.
- Liste des patterns qu'on devrait voir en détail : Strategy, State, Observer, Decorator, Adapter, ~~Façade~~, Factory Method, Abstract Factory, ~~Singleton~~.
- Liste des patterns qu'on devrait voir rapidement : Command, Chain of Responsability, Visitor, Composite, Bridge, Proxy, Builder, Object Pool.
- Vos programmes doivent être écrit **en C++** et doivent tourner dans les salles TP sous Linux.

Concernant le projet

Projet

- Groupes en binômes.
- 3 patterns d'au moins deux familles différentes à utiliser, au choix. Si vous utilisez un ou plusieurs patterns qui ne seront pas vu en cours, m'envoyer un email **avant** de vous lancer dans la programmation.
- Liste des patterns qu'on devrait voir en détail : Strategy, State, Observer, Decorator, Adapter, ~~Façade~~, Factory Method, Abstract Factory, ~~Singleton~~.
- Liste des patterns qu'on devrait voir rapidement : Command, Chain of Responsability, Visitor, Composite, Bridge, Proxy, Builder, Object Pool.
- Vos programmes doivent être écrit **en C++** et doivent tourner dans les salles TP sous Linux.
- **Votre programme doit fonctionner ; à vous de doser la difficulté !**