ÉTS
Le génie pour l'industrie

LOG121
Conception orientée objet

Cycle de développement logiciel – Conception –Concepts orientés objets

Enseignante: Souad Hadjres

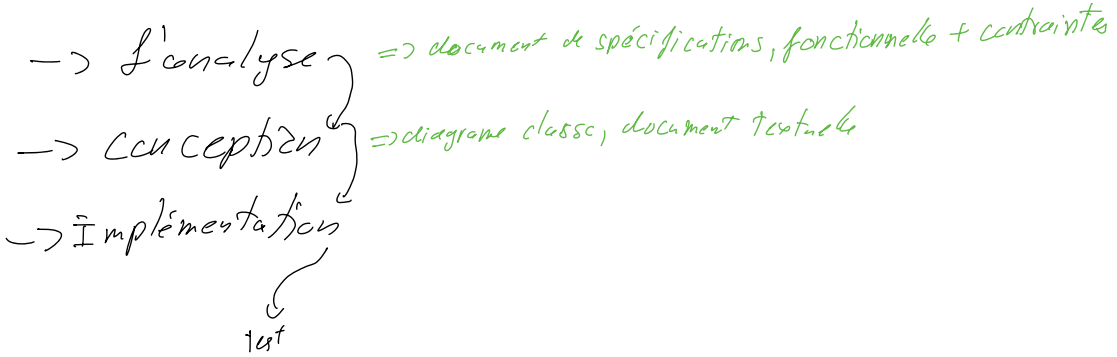
Plan

- Processus de développement du logiciel
- Phase de conception
- Concepts de l'orienté objet

Processus de développement logiciel

- Qu'est-ce que c'est?
- Pour quel besoin?

Ensemble d'étape qui permette de créer un logiciel



c'est ce que le client veut

Processus de développement logiciel

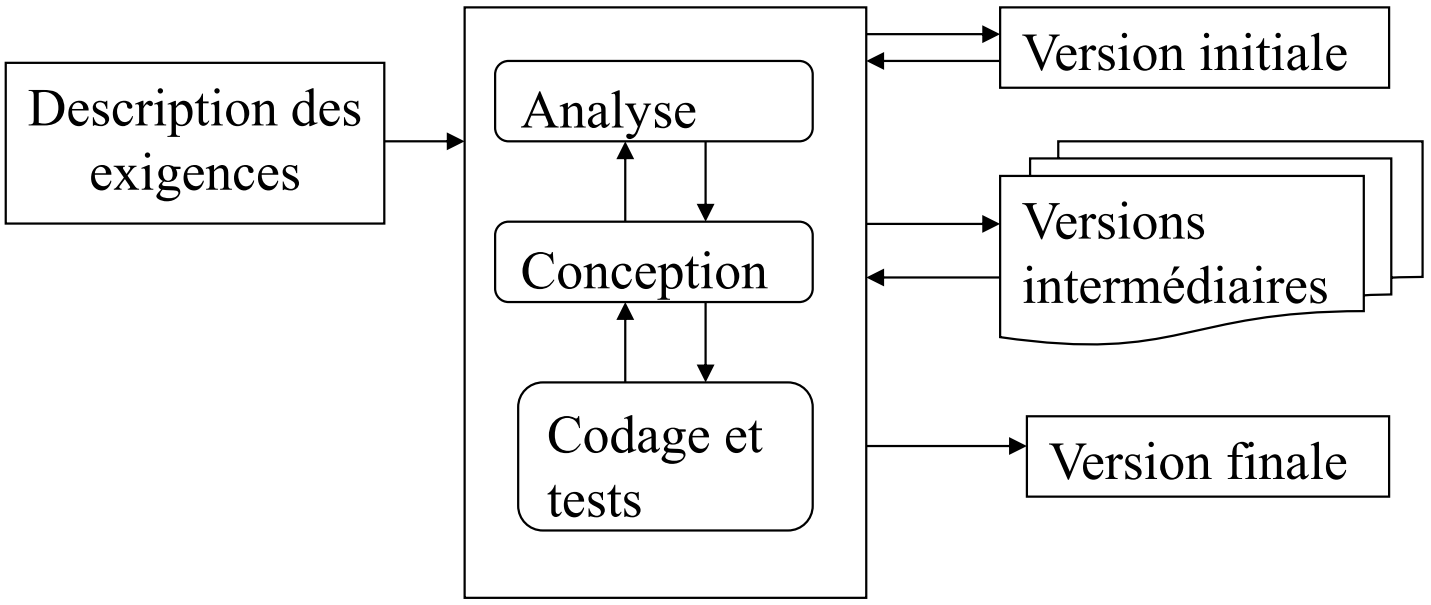
- Ensemble des étapes menant à la mise en œuvre d'un logiciel
  - Chaque étape est bien définie
    - Une étape a des intrants
    - Une étape a des livrables
- Différents types de processus
  - Linéaire
  - En cascades
  - Itératif
  - Unifié
  - ....

Processus de développement logiciel

- Répond aux besoins de gestion et de planification
  - Un budget limité
  - Des échéanciers à respecter
  - Plusieurs intervenants
- Permet une visibilité du progrès réalisé
  - La visibilité dépend du type de processus
- Permet un meilleur contrôle de la qualité du produit final

Processus de développement logiciel

- Cycle de développement itératif et incrémental



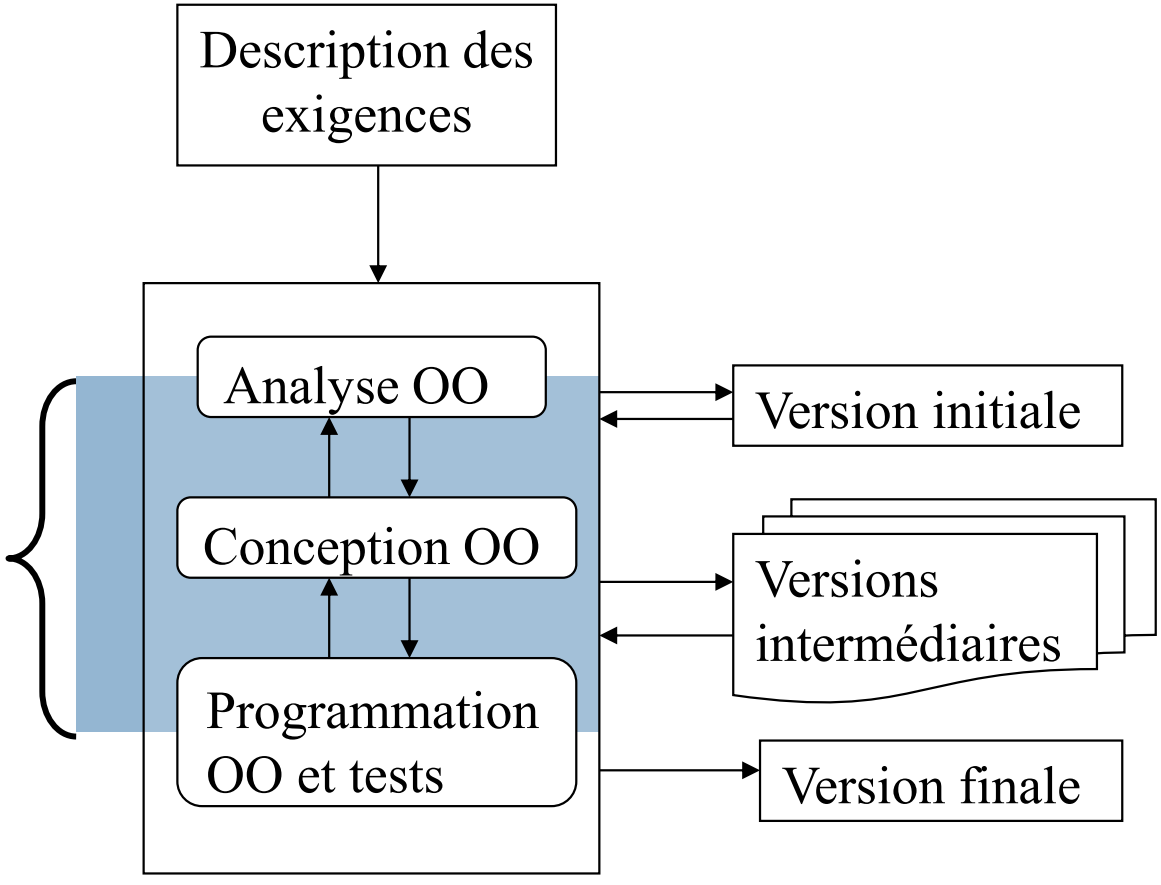
Modèle adapté de (Sommerville I., Software Engineering, Addison Wesley, 2001).

Processus de développement logiciel

- Cycle de développement itératif et incrémental
  - Itération
    - mini-projet à résultat testé et exécutable
    - feed-back rapide et minimisation des risques
  - Incréments
    - le système croît après chaque itération
    - convergence vers le produit final

Processus de développement logiciel

- LOG121: conception orientée objet (OO)
  - Concepts orientés objet
  - Patrons de conception
  - Java
  - UML



Plan

- Processus de développement du logiciel
- Phase de conception
- Concepts de l'orienté objet

Phase de conception

- Intrants: spécification des exigences
  - Exigences de fonction
  - Exigences de qualité
    - Temps de réponse, capacité en termes de nombre d'utilisateurs, disponibilité, utilisabilité, adaptabilité, etc.
  - Contraintes
    - Techniques
    - Règlements/Standards
    - Etc.

Phase de conception

- Élaborer les différentes parties du système et leurs interactions
  - Conception architecturale: partitionner le système en sous-systèmes / modules / composants
  - Conception détaillée: définir le contenu des modules identifiés (classes, collaborations, comportements, etc.).
- Résultats (Output)
  - Un plan qui facilite l'implémentation et la maintenance du logiciel
  - Artefacts de conception
    - Diagrammes de classes
    - Diagrammes de collaborations
    - Diagrammes d'états
    - Description textuelle, etc.

Plan

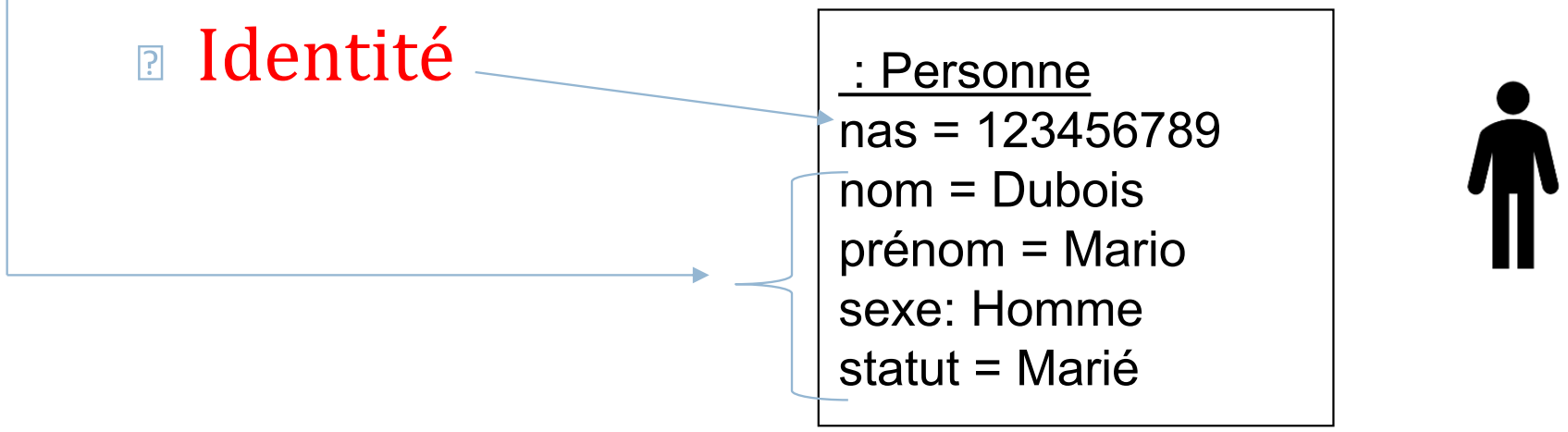
- Processus de développement du logiciel
- Phase de conception
- Concepts de l'orienté objet

Conception orientée objet

- Objectifs
  - Identifier les classes
  - Identifier les responsabilités de ces classes c- méthode
  - Identifier les relations entre ces classes
- Le processus est itératif
  - On parle d'objectifs et non pas d'étapes

Objet

- Un objet est défini par
  - État
  - Comportement
  - Identité



- Quels comportements Mario peut-il avoir?

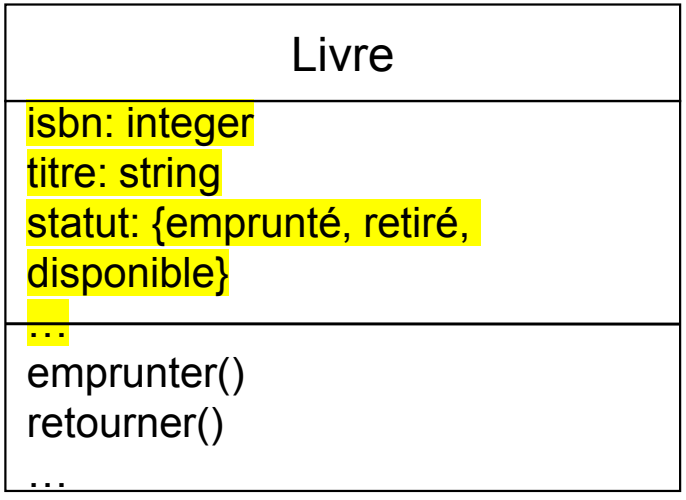
Classe

- Une classe spécifie le comportement et les états possibles d'un ensemble d'objets de même type

Personne
nas: integer nom: string prénom: string statut: {marié, célibataire, divorcé} …
seMarier() divorcer() …

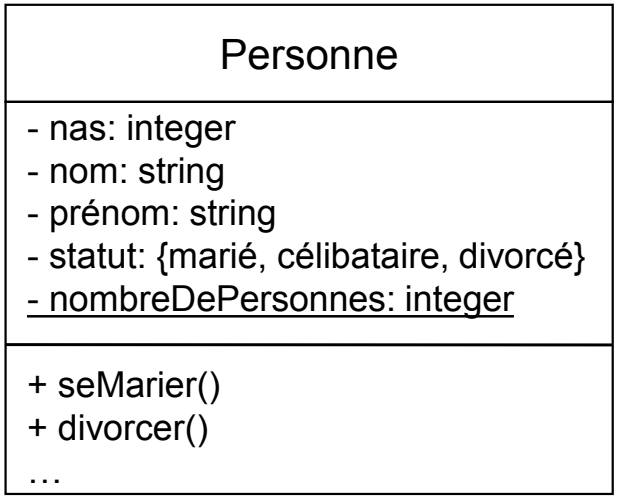
Classe

- Attribut
  - Propriété qui caractérise les objets de la classe
  - Les valeurs des attributs d'un objet définissent l'état de l'objet
    - Cet état change au cours de l'exécution de l'application



Classe

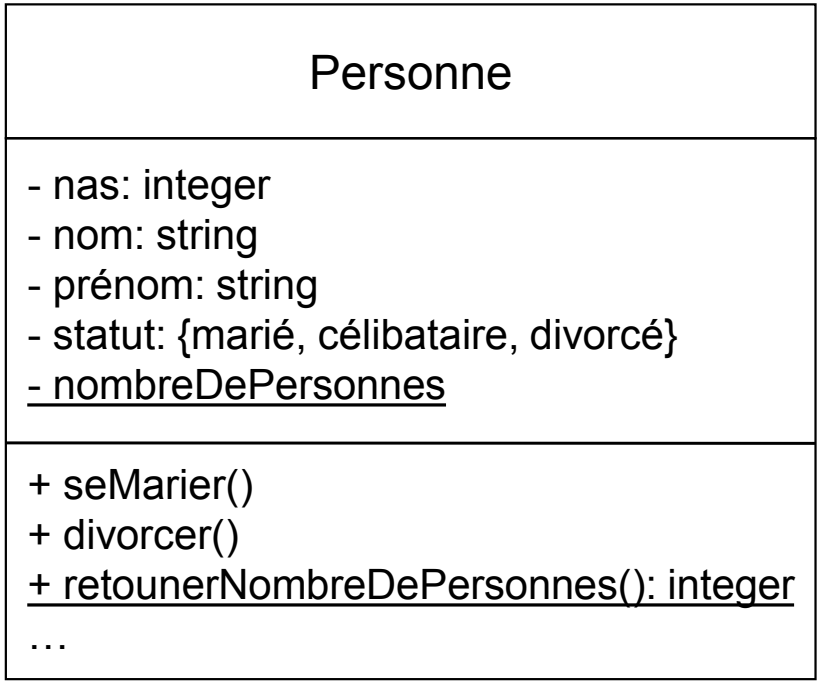
- Attribut statique versus d'instance
  - Statique: partagé par tous les objets instances de la classe
    - Exemple: nombreDePersonnes de la classe Personne
  - D'instance: attribut propre à chaque objet
- Visibilité de l'attribut
  - Publique (+), protégé (#) ou privé (-)
  - Accès aux attributs d'une classe par d'autres classes



Modificateur	Classes dans le même Package	Ses sous-classes	Le reste des classes
public	oui	oui	oui
protected	oui	oui	non
private	non	non	non

Classe

- Méthode statique versus d'instance
  - Statique: n'opère pas sur un objet
    - Exemple: retournerNombreDePersonnes() de la classe Personne
    - La classe Math offre plusieurs méthodes de calcul qui sont statiques
  - D'instance: doit être appelée sur un objet
    - Exemples: seMarier(), divorcer()
- Visibilité de la méthode
  - Publique (+), protégé (#) ou privé (-)



Classe


```
01: /**
02:    A class for producing simple greetings.
03: */
04:
05: public class Greeter
06: {
07:    /**
08:       Constructs a Greeter object that can greet a person or
09:       entity.
10:       @param aName the name of the person or entity who should
11:       be addressed in the greetings.
12:    */
13:    public Greeter(String aName)
14:    {
15:       name = aName;
16:    }
17:
18:    /**
19:       Greet with a "Hello" message.
20:       @return a message containing "Hello" and the name of
21:       the greeted person or entity.
22:    */
23:    public String sayHello()
24:    {
25:       return "Hello, " + name + "!";
26:    }
27:
28:    private String name;
29: }
```

```
1: public class GreeterTester
2: {
3:    public static void main(String[] args)
4:    {
5:       Greeter worldGreeter = new Greeter("World");
6:       String greeting = worldGreeter.sayHello();
7:       System.out.println(greeting);
8:    }
9: }
```

Comment trouver les classes?

- Comment procéder pour trouver les classes dans des énoncés?
  - On utilise une méthode empirique qui consiste à chercher les noms.
  - On utilise aussi des catégories de classes:
    - Tangibles: concepts visibles du domaine analysé
    - Agents: représentent des opérations (ex: Parser, Printer, etc.)
    - Événements et transactions: activités que l'on veut manipuler et dont on veut garder trace
    - Usagers/rôles: utilisateurs du système
    - Systèmes: sous-systèmes ou le système intégral
    - Interfaces de systèmes: interfaces avec d'autres systèmes
    - Classes de base: comprennent les types de base

Comment assigner des responsabilités aux classes?

- Les responsabilités donnent un sens à l'existence de la classe
- Chercher les verbes dans l'énoncé du logiciel à réaliser
  - Un verbe décrit généralement une opération qu'un objet doit réaliser
- Une opération est la responsabilité d'une seule classe

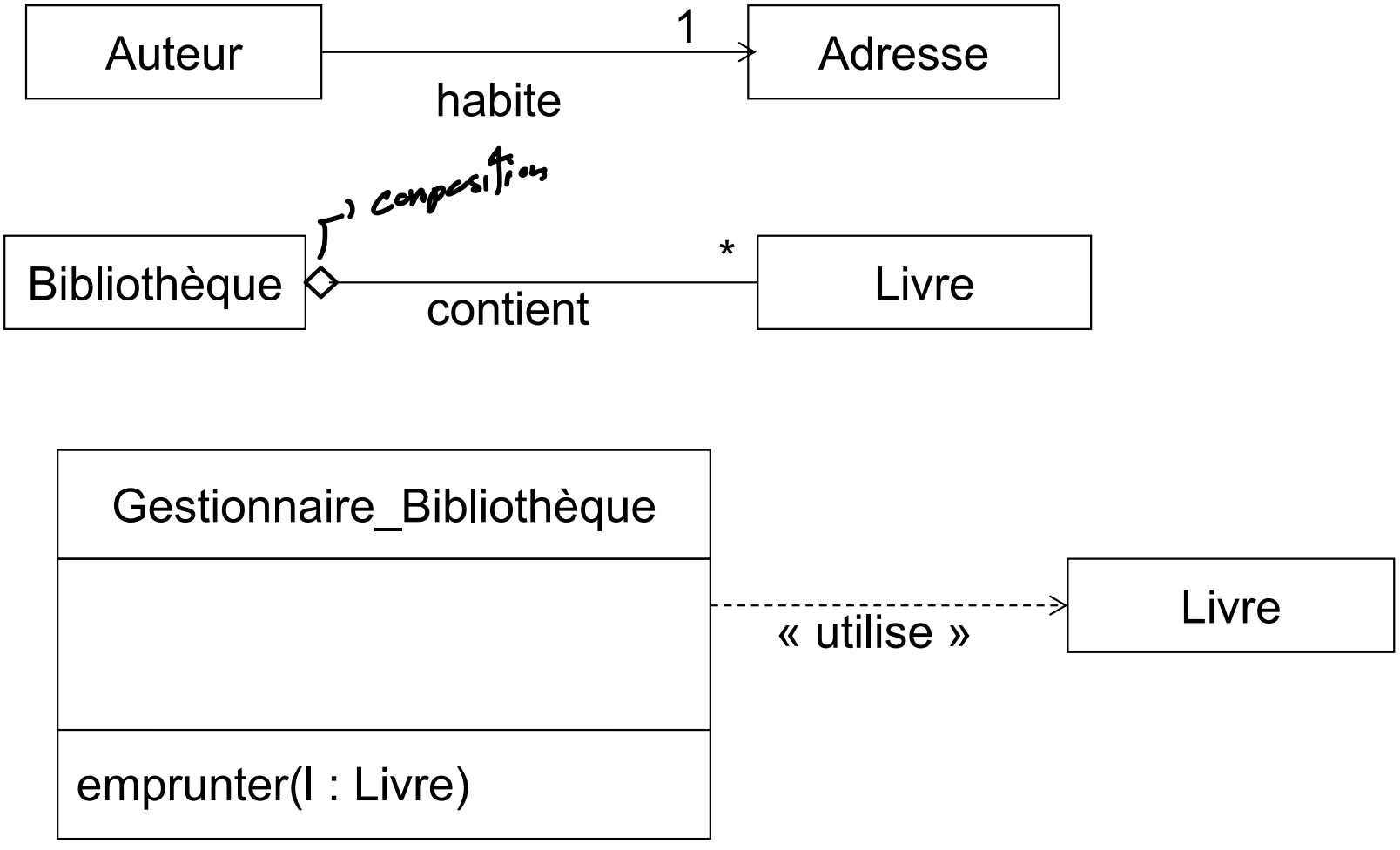
Relations entre classes

- À chaque famille de liens entre objets correspond une relation entre les classes de ces mêmes objets.
- Les objets sont les instances de classes, les liens entre objets sont des instances des relations entre classes

Relations entre classes

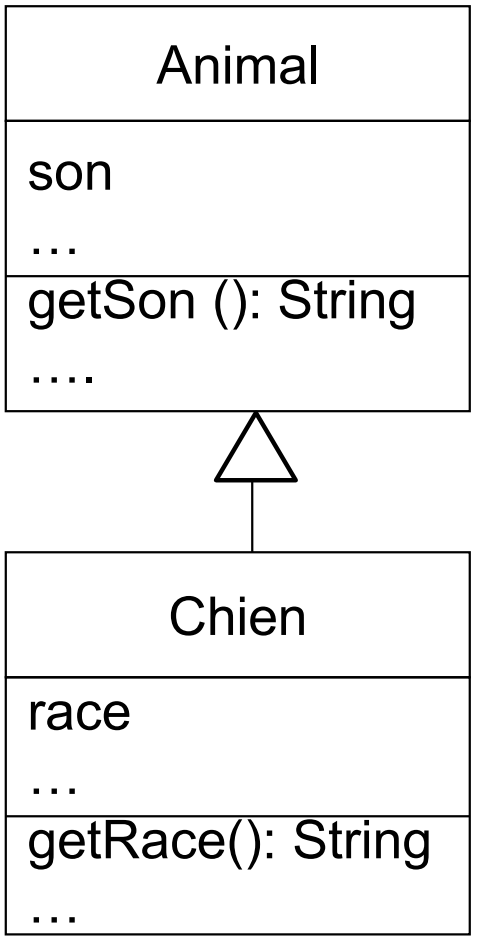
- Dépendance: « connaît », « utilise »
  - Association: « connaît »
    - Dépendance explicite durable dans le temps
    - Cas particulier: Agrégation (« contient », « a »)
  - Utilisation: « utilise »
    - Dépendance ponctuelle dans le temps
      - A lieu à un moment donné de l'existence des objets dépendants
        - Exemple: ObjetB passé en paramètre à une méthode d'un ObjetA. ObjetA établit une relation avec ObjetB juste quand cette méthode est appelée.

Relations entre classes



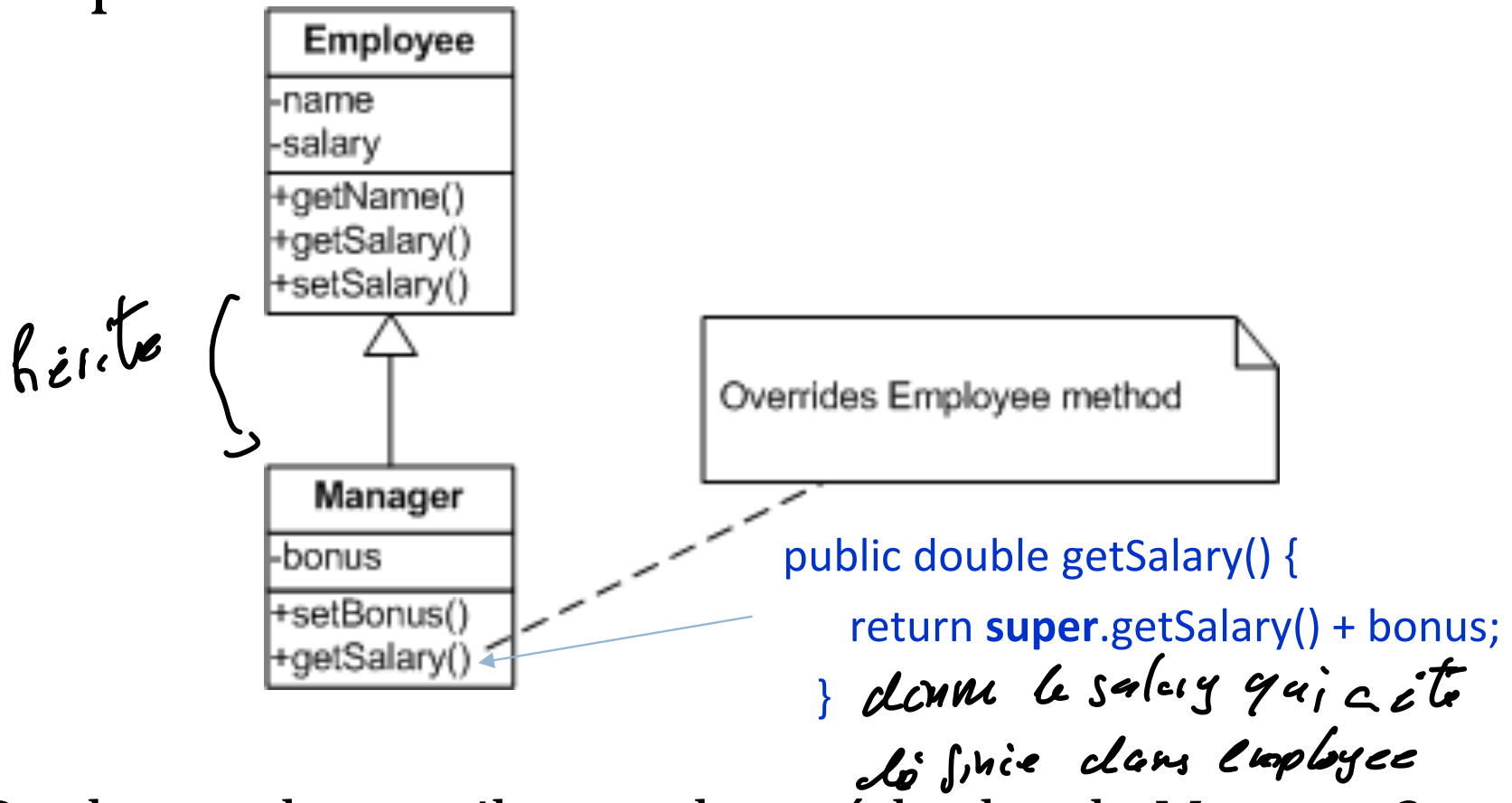
Héritage

- Permet la réutilisation des états et du comportement d'une classe générale par une classe plus spécialisée
  - La classe générale définit un ensemble de propriétés communes à des classes plus spécialisées
  - La classe plus spécialisée peut définir des propriétés additionnelles qui lui sont propres
- Relation d'héritage: « est »
  - Une sous-classe est un cas particulier de la superclasse



Héritage

- Exemple

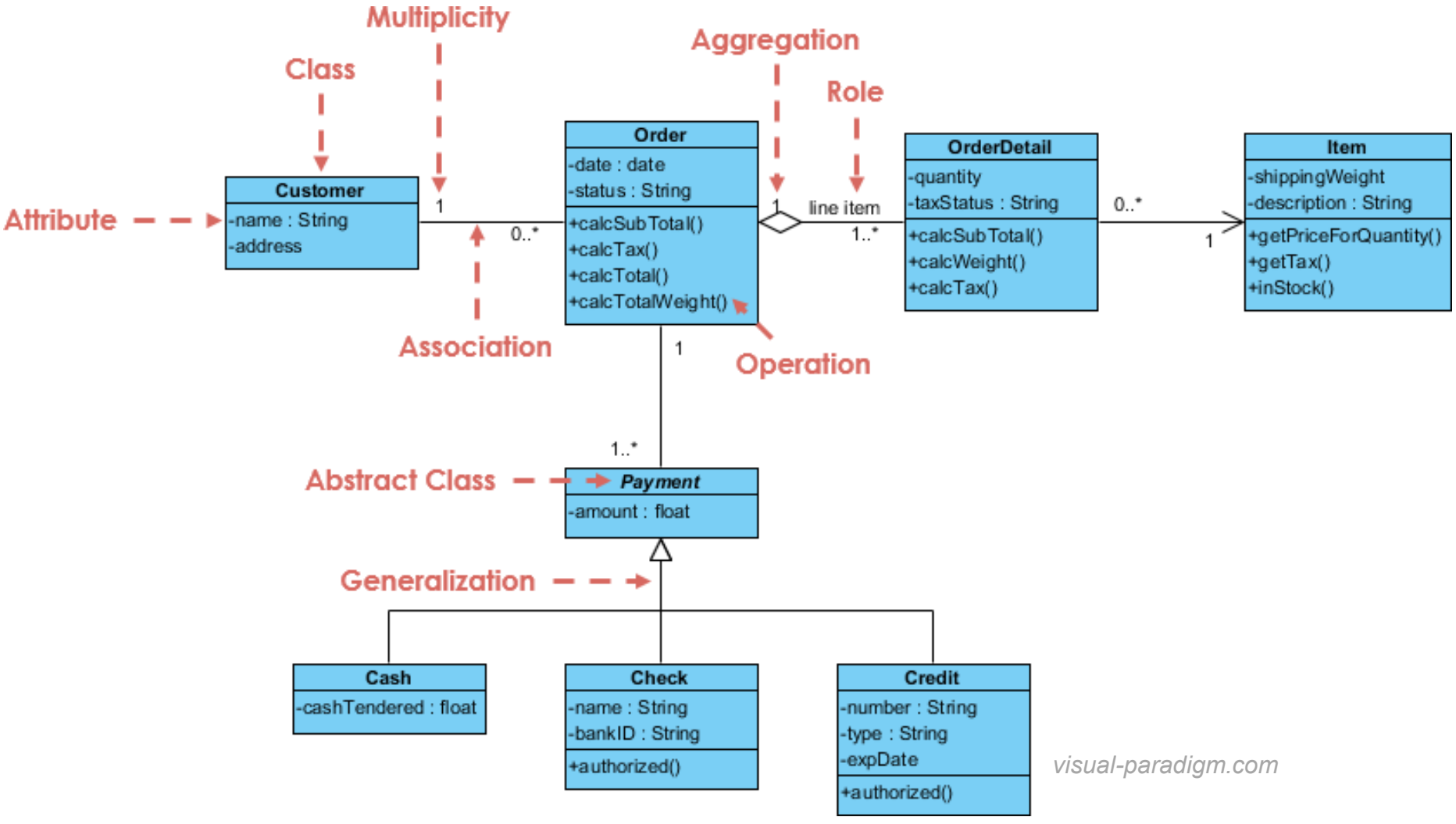


- Quels sont les attributs et les méthodes de Manager?

Héritage

- Vos exemples?
- Peut-on étendre une classe déclarée finale? non
- Une sous-classe peut-elle redéfinir une méthode déclarée finale dans la superclasse? non

Relations entre classes



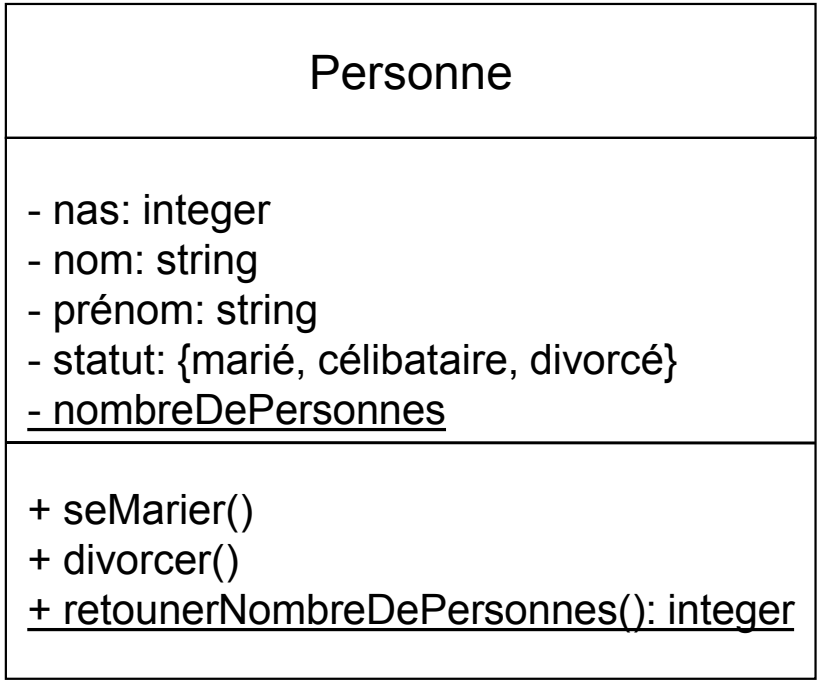
Exemples de classes et d'interactions entre-elles

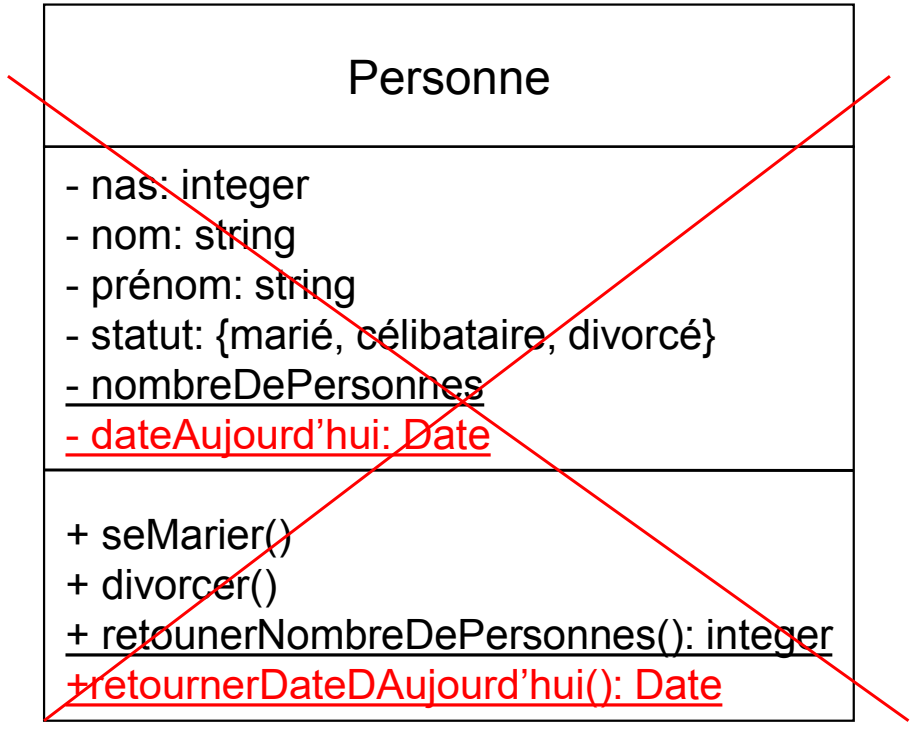
Couplage

- Qu'est-ce que c'est? Toutes les dépendances entre les classes
- Pourquoi faut-il le minimiser? si le couplage est fort difficile à modifier et réutiliser
  - Couplage: mesure de dépendances entre classes
  - Un couplage minimisé facilite
    - La compréhension des classes
    - La maintenance
      - Limiter l'effet des changements d'une classe sur les autres classes du système
    - La réutilisation des classes

Couplage et cohésion ?

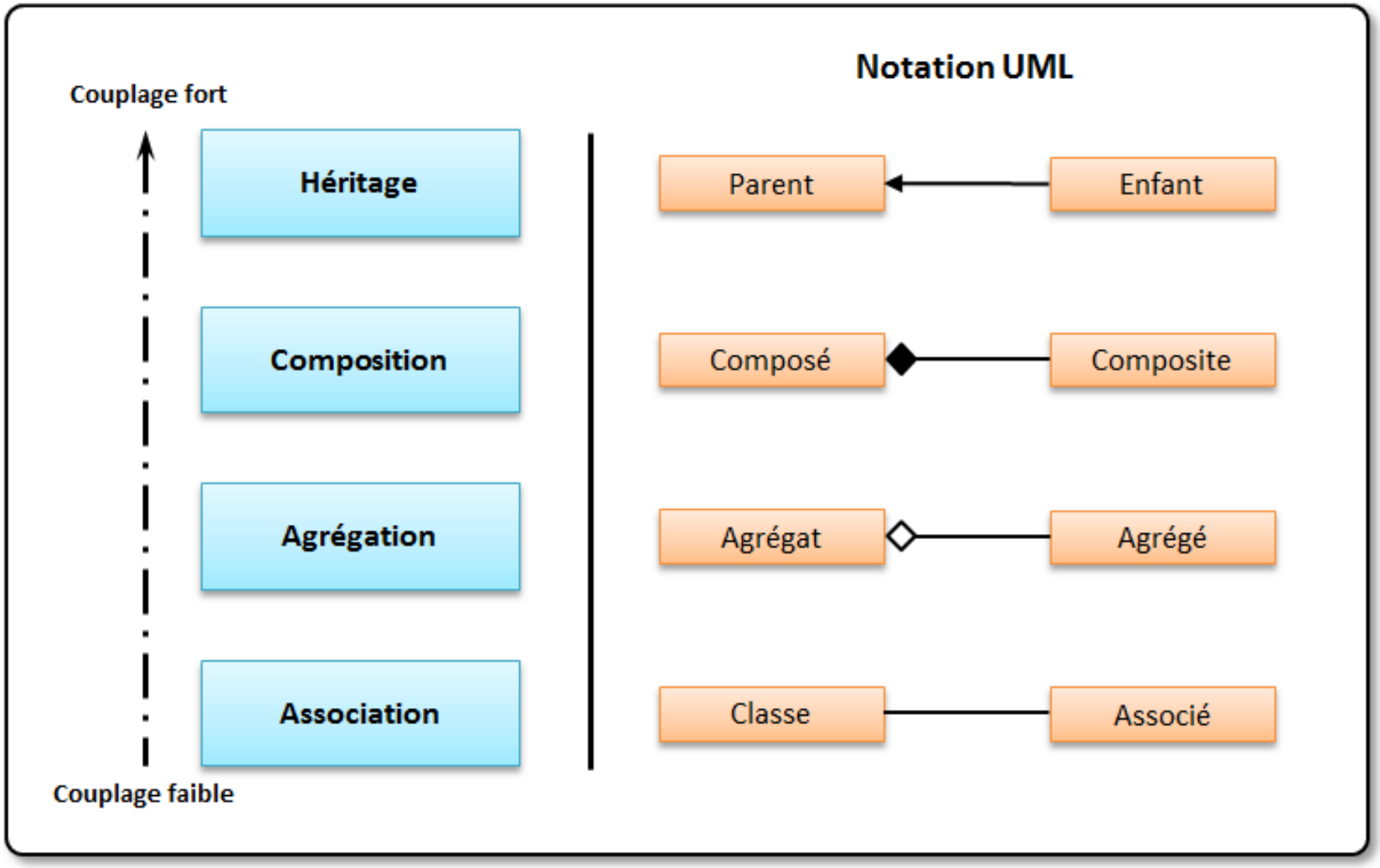
- Cohésion: Qu'est-ce que c'est?
  - Cohésion: Une classe est une abstraction d'un seul concept





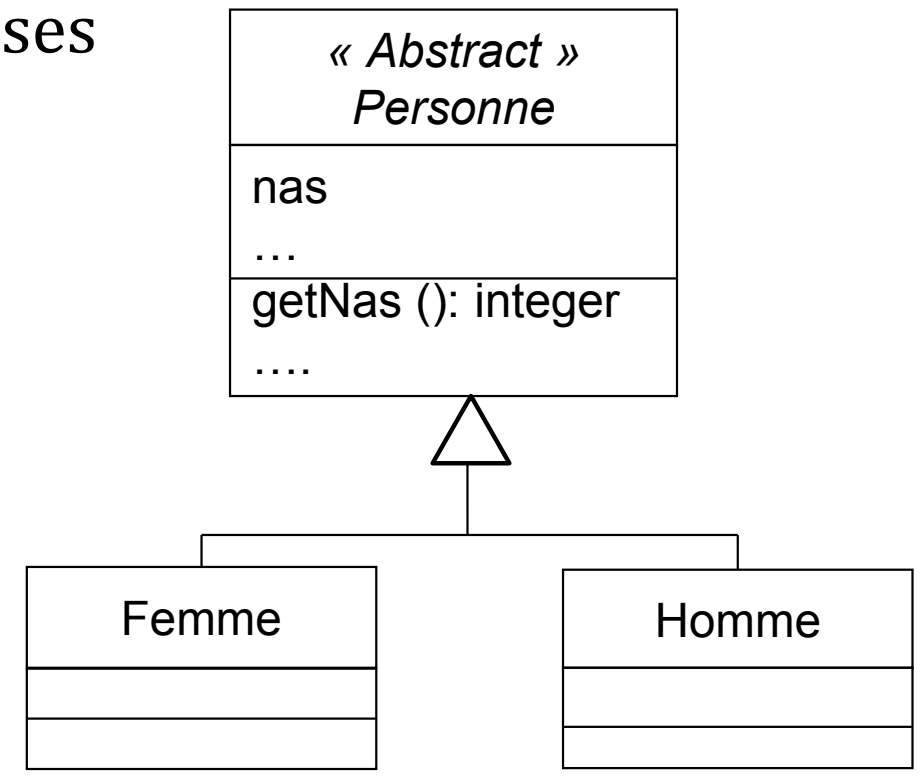
  - Des classes plus cohésives permettent d'optimiser le couplage

Couplages (de faible à fort)



Classe Abstraite

- Qu'est-ce qu'une classe abstraite?
  - C'est une classe qu'on ne peut pas instancier
  - Elle décrit des concepts abstraits
  - Une méthode abstraite de la classe abstraite doit être implémentée par les sous-classes



Interface

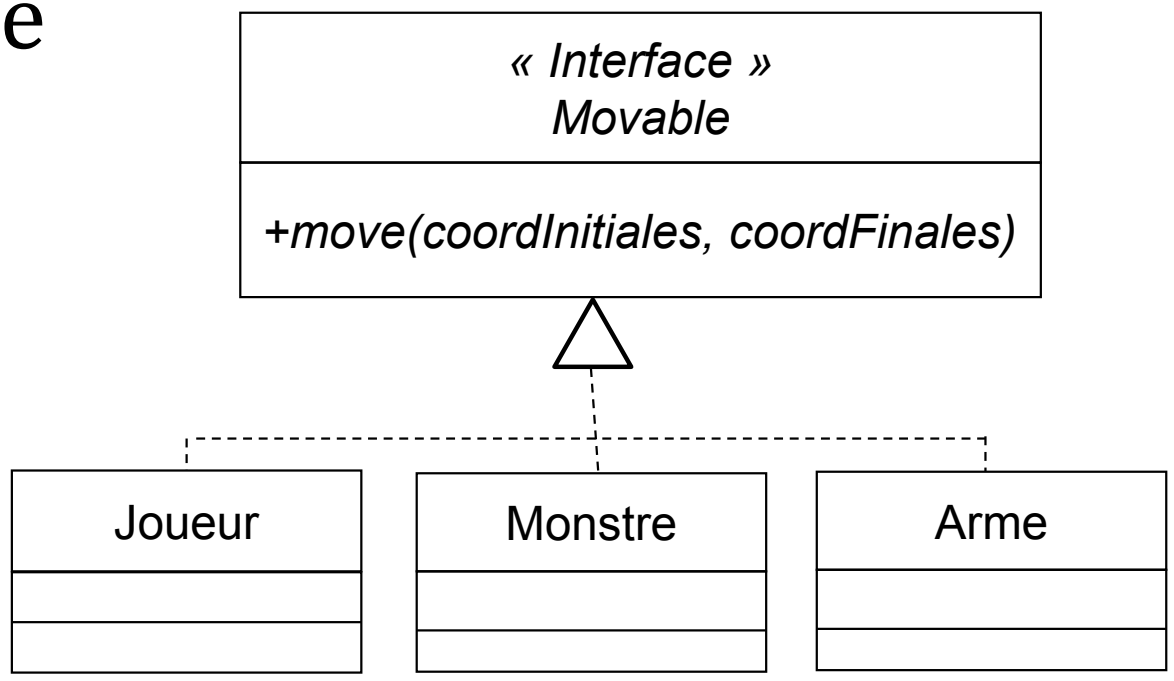
- Une interface est un contrat
  - Elle ne peut être instanciée
  - Elle spécifie un ensemble de méthodes sans les implémenter (java 8 permet l'implémentation des méthodes)
  - La classe implémentant l'interface doit implémenter toutes les méthodes définies (et non implémentées) par cette interface

Interface

- Qu'est ce qu'on peut déclarer dans une interface?
  - Constantes: attributs qui sont publiques, statiques et finaux.
  - Méthodes:
    - une méthode est définie par sa signature (son nom, ses paramètres et leurs types).
    - Les méthodes de l'interface sont publiques
  - Types: classes ou interfaces

Interface

- Exemple



- Plusieurs exemples dans Java
  - Serializable, Comparable, Scrollable, etc.

Classe Abstraite versus Interface

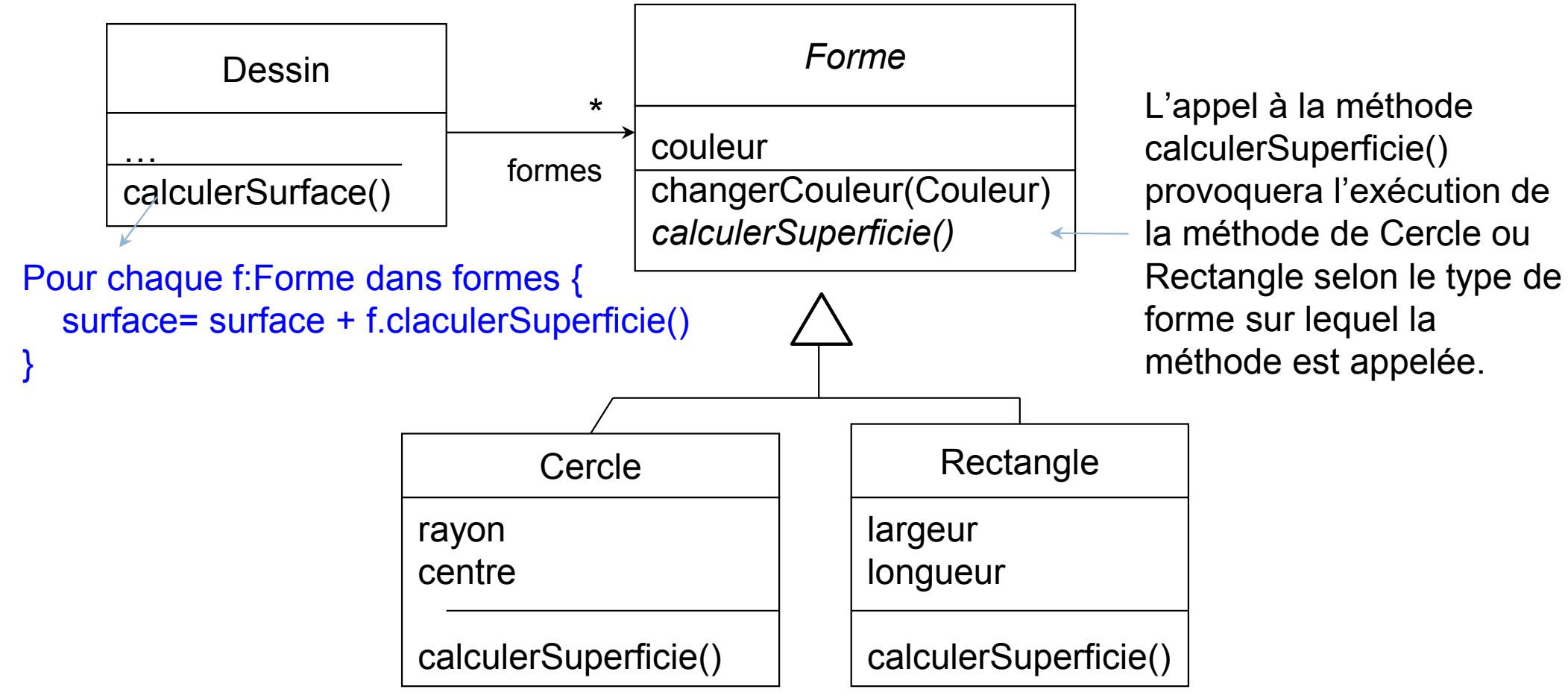
- Attributs
  - Une classe abstraite peut avoir des attributs
  - Une interface peut définir des constantes uniquement
- Méthodes
  - Une classe abstraite peut implémenter des méthodes
  - Une interface peut seulement déclarer des méthodes
- Dans Java, une classe peut étendre une seule autre classe mais elle peut implémenter plusieurs interfaces

Polymorphisme

- « Poly-morphisme »: prend plusieurs formes
  - Lié aux notions d'héritage et d'interface
    - Les sous-classes redéfinissent certaines méthodes de la superclasse pour implémenter leur comportement spécifique
    - Une classe implémente toutes les méthodes définies par l'interface qu'elle implémente
  - Une classe cliente peut appeler les méthodes définies par l'interface ou la superclasse
  - La méthode exécutée dépendra du type de l'objet recevant l'appel
    - L'interpréteur Java cherche d'abord quel est le type de l'objet et ensuite il appelle la méthode de ce type là

Polymorphisme (suite)

- Exemple



Polymorphisme (suite)

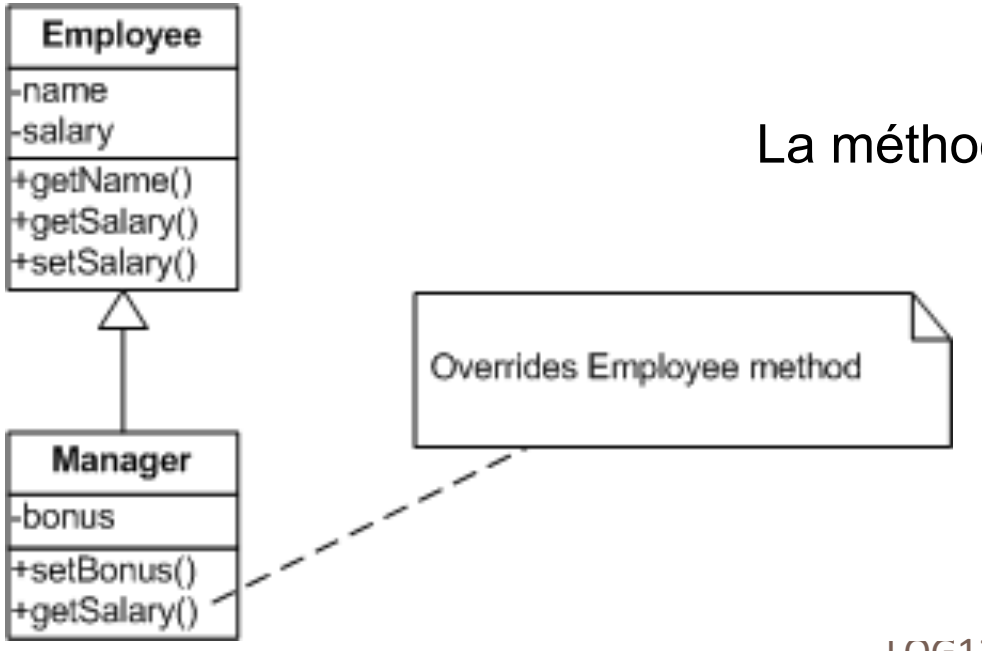
- Avantages
  - Réduction du couplage
    - La classe cliente connaît uniquement la superclasse d'une hiérarchie ou l'interface implémentée par des classes
  - Extensibilité
    - On peut facilement ajouter de nouvelles classes

Principe de substitution

- « On peut utiliser une sous classe partout où une superclasse est attendue » Barbara Liskov

```
Employee e;
...
e = new Manager("Bernie Smith");
System.out.println("name=" + e.getName());
System.out.println("salary=" + e.getSalary());
```

La méthode de quelle classe est exécutée ici?



Principe de substitution

- Une méthode peut être vue comme un contrat
  - Elle a une signature qu'il faut respecter
    - La signature inclut les paramètres et les exceptions levées
  - Précondition: une condition qui doit être satisfaite avant que la méthode ne soit appelée
  - Post-condition: une condition qui doit être satisfaite après que la méthode ne soit exécutée

Principe de substitution

- Impact du principe de substitution sur les méthodes redéfinies
  - La précondition de la méthode redéfinie **ne peut être plus forte** que celle de la méthode de la superclasse
    - Exemple d'illustration

```
public class Employee {
    /**
      Sets the employee salary to a given value.
      @param aSalary the new salary
      @precondition aSalary > 0
    */
    public void setSalary(double aSalary) { ... }
}
```

      - Peut-on redéfinir Manager.setSalary avec une pré-condition salary > 100000?
      - Non – cette condition pourrait être violée

```
Manager m = new Manager();
Employee e = m;
e.setSalary(50000);
```

Principe de substitution

- Impact du principe de substitution sur les méthodes redéfinies (suite)
  - La post-condition de la méthode redéfinie est aussi forte ou plus forte que celle de la superclasse
    - Exemple
      - Si Employee.setSalary a pour post-condition de ne pas diminuer le salaire alors Manager.setSalary doit respecter cette post-condition
  - La méthode redéfinie ne peut pas être moins accessible (visible) que la méthode de la superclasse
  - La méthode redéfinie ne peut pas lever plus d'exceptions que la méthode de la superclasse

Encapsulation

- Objectif: restreindre l'étendue des modifications que pourrait subir une classe à cause de la modification apportée à une autre classe
- Comment: Objet = boite noire
  - Cacher les détails d'implémentation aux classes utilisant l'objet
  - L'interaction avec l'objet se fait à travers une interface publique

Encapsulation

- Règles pour mettre en œuvre l'encapsulation
  - Les variables d'instance doivent être privées
    - Fournir des méthodes d'accès ou accesseurs (getters) et des mutateurs (setters)
    - Pas besoin de fournir un mutateur pour chaque variable d'instance
  - Favoriser la conception de classes immuables
    - Classe immuable = classe sans mutateurs
    - Les références aux objets immuables peuvent être partagées
  - Séparer les accesseurs et les mutateurs
  - Minimiser les effets secondaires au-delà du paramètre implicite d'une méthode
    - objet1.method1(les paramètres explicites) : objet1 est le paramètre implicite de méthod1
  - Loi de Demeter
    - Une classe ne doit pas retourner une référence à un objet qui fait partie de son implémentation. Elle doit prendre toute la responsabilité d'interaction avec cet objet.

Les qualités de l'interface d'une classe

- Cohésion
  - Une classe est une abstraction d'un seul concept
- Complétude
  - Supporter toutes les opérations qui font partie de l'abstraction représentée par la classe
- Convenance (commodité)
  - L'interface rend simple les tâches usuelles
- Clarté
  - L'interface est claire et n'introduit aucune confusion
- Cohérence
  - Adopter une façon uniforme dans la définition des noms, des paramètres et du comportement des méthodes