ÉTS
Le génie pour l'industrie

LOG100 / GTI100
Programmation en génie logiciel et des TI
Automne 2024

Cours 6 : Principes de la Conception Orientée Objet (COO)

Chargé de cours: Anes Abdennebi

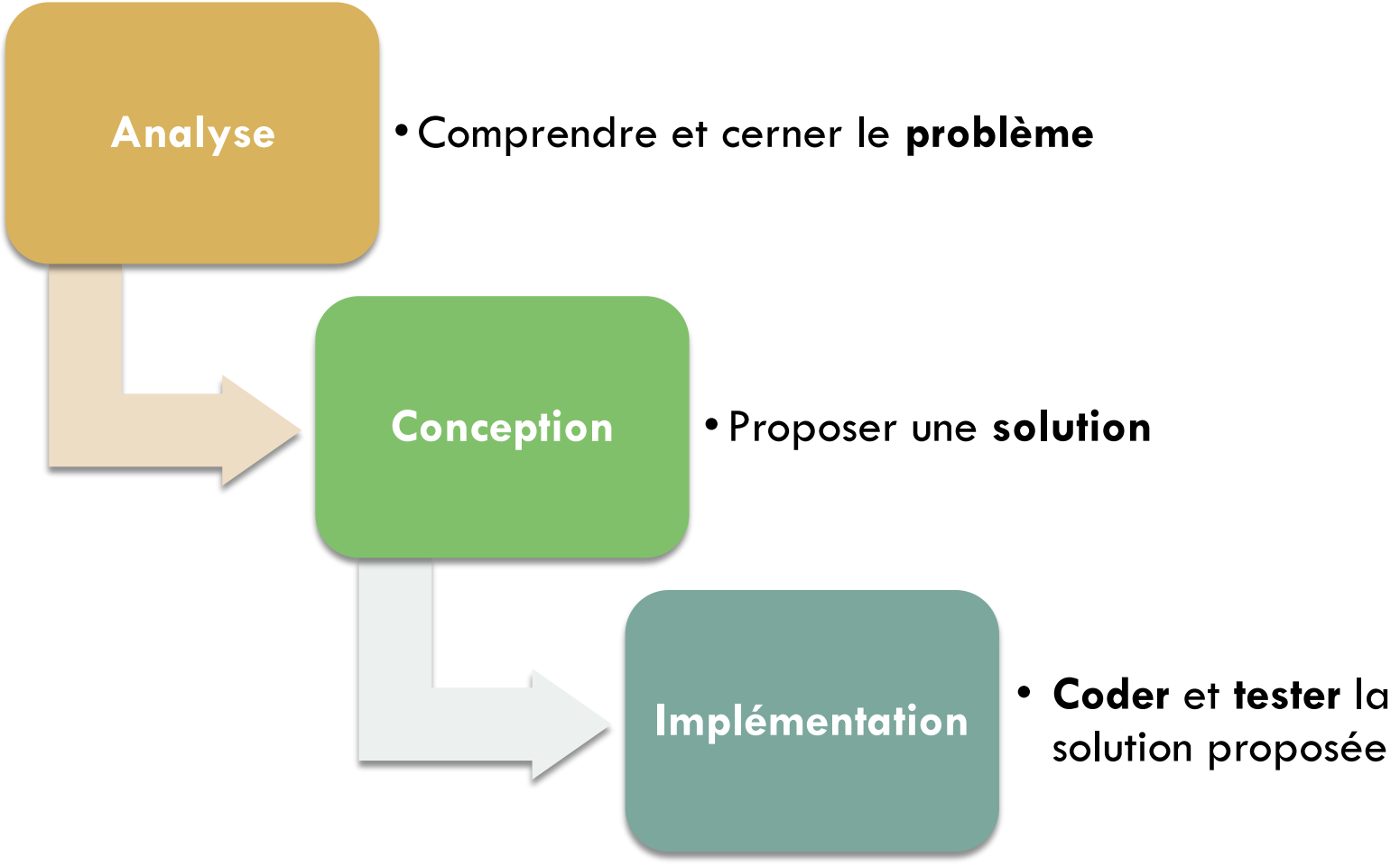
Crédits à: Ali Ouni, PhD

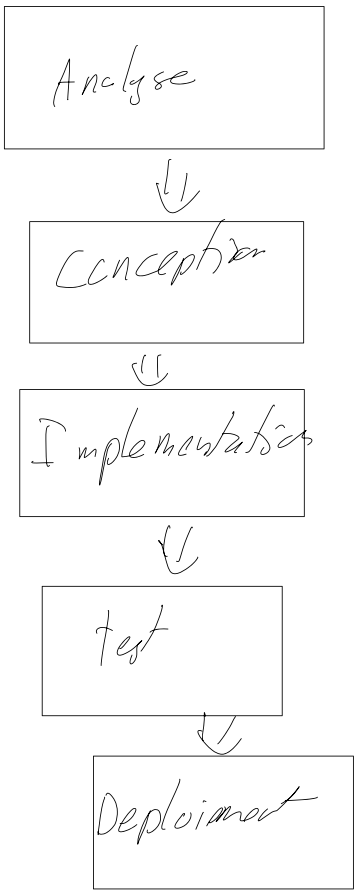
Plan

- Cycle de développement
- Identification des classes et des responsabilités
- Relations entre classes
- Où et comment chercher les classes, leurs responsabilités et leurs interactions?
- Bonnes pratiques et qualité d'une interface

Cycle de développement

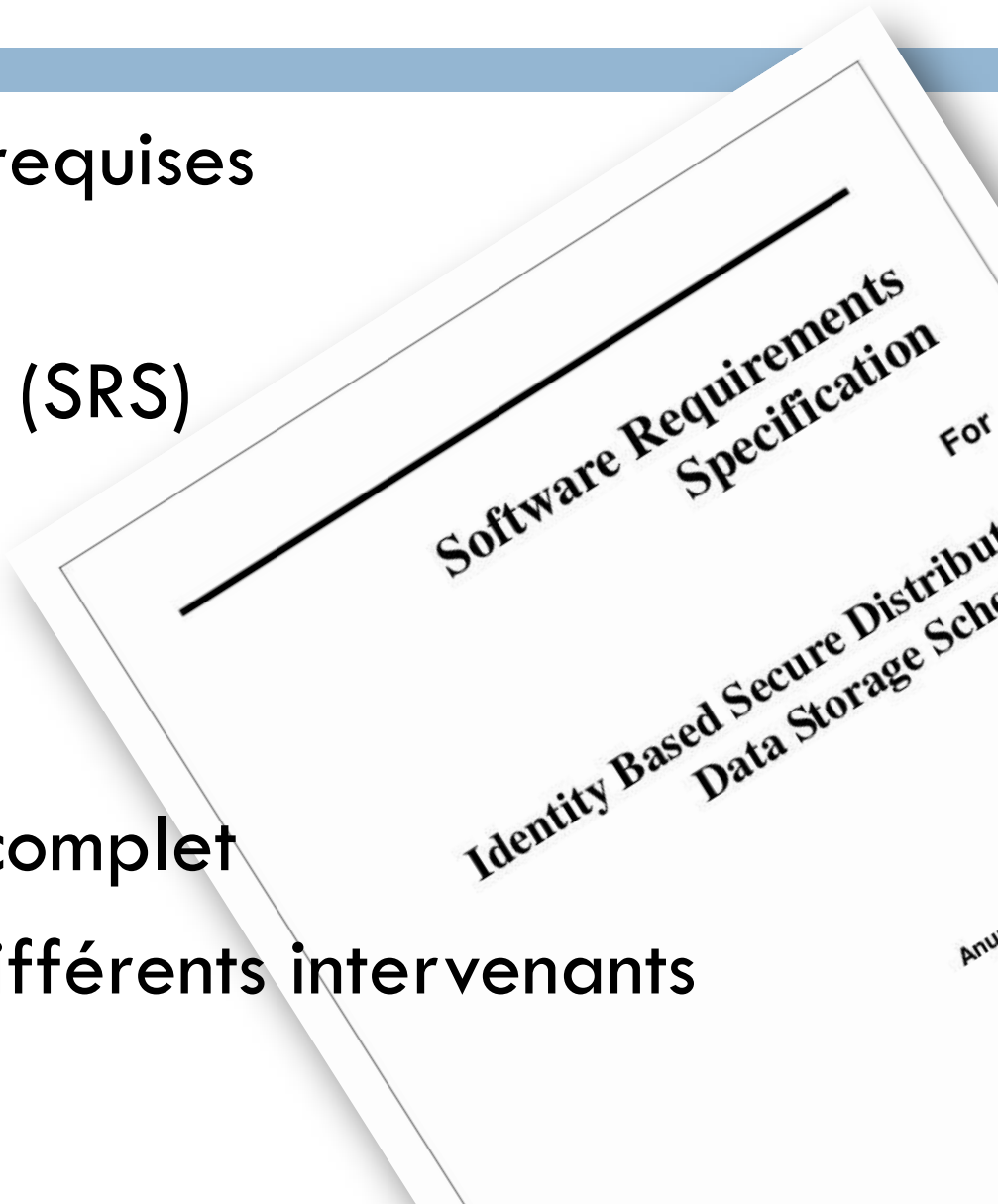
- Diviser le travail en **étapes** et **organiser** ces étapes




Analyse
Conception
Implementation
test
Deploiement

Phase d'analyse

- Déterminer les exigences requises
  - *Que doit faire* le système?
- Document de spécification (SRS)
  - Exigences fonctionnelles
  - Exigences de qualité
  - Contraintes
- Non ambigu, cohérent et complet
- Compréhensible par les différents intervenants



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.)

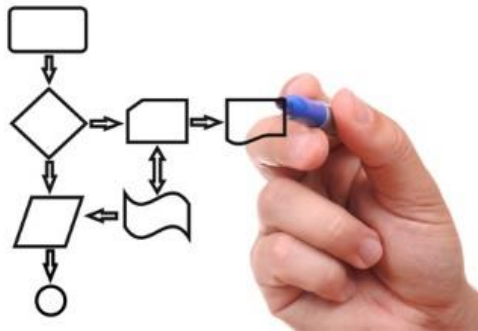
Phase de conception

- Objectifs de la conception orientée objet
  - Identifier les **classes**
  - Identifier les **responsabilités** de ces classes
  - Identifier les **relations** entre ces classes

On parle d'un processus de découverte itératif

Phase de conception

- **Documenter** la conception
  - Un plan qui facilite l'implémentation et la maintenance du logiciel
- Plusieurs **artéfacts** de conception
  - Diagrammes de classes
  - Diagrammes de collaborations
  - Diagrammes d'états
  - Description textuelle
  - …



Phase d'implémentation

- Choix d'un **langage de programmation**
- **Coder** les classes, les méthodes, etc.
- Faire des **tests** à différents niveaux
  - *Unitaire* – test d'une classe / d'un composant
  - *D'intégration* – vérifier, de façon **incrémentale**, l'interface et l'interaction entre deux composants différents
  - *Système* – l'ensemble du système par rapport aux exigences
- **Déployer!**

Itératif

Plan

- Cycle de développement
- Identification des classes et des responsabilités
- Relations entre classes
- Où et comment chercher les classes, leurs responsabilités et leurs interactions?
- Bonnes pratiques et qualité d'une interface

Objet : exemple

- Un système de gestion de messagerie vocale
  - La boîte de messagerie peut être **vide** ou **pleine**
  - On peut **ajouter**, **récupérer** et **supprimer** un message de la boîte de messagerie
  - *Si la boîte est vide* lorsque son propriétaire prend ses messages, elle va annoncer « aucun message »
  - Tout nouveau message sera rejeté lorsque la boite est pleine

Identifier les classes

- Chercher les **noms** dans la **spécification** du programme à réaliser
  - Exemples dans le système de messagerie vocale :
    - *Message*, *Mailbox*, *Extension*, etc.
- Attention, les noms identifiés ne correspondent pas forcément aux classes
  - Certains peuvent correspondre à des attributs
- Conseils :
  - Une classe doit avoir un nom **singulier**
  - Le nom doit être **significatif** et pas générique

Identifier les classes

- D'autres classes peuvent être nécessaires :
  - L'utilisateur d'une boite de messagerie désire traiter les messages dans un ordre FIFO
  - Nécessité d'introduire une classe *FileMessages*
  - Le comportement de *FileMessages* est celui d'une file
  - L'implémentation de *FileMessages* n'a aucun intérêt au stade de conception

Identifier les classes

- Des catégories de classes pour vous aider
  - Tangibles (?) : concepts **visibles** du domaine analysé
  - Agents : représentent des **opérations**
  - 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

Identifier les responsabilités

- **Chercher les verbes** dans la spécification du programme à réaliser
- Les responsabilités donnent un sens à l'existence de la classe
- Les responsabilités de *FileMessages* :
  - Ajouter un message à la fin de la file
  - Supprimer un message du début de la file
  - Vérifier si la file est vide ou non

Identifier les responsabilités

- Une responsabilité appartient à une seule classe

- Posez-vous les questions suivantes :
  - **Comment** un objet d'une classe peut réaliser cette responsabilité?
  - La classe a-t-elle la **connaissance nécessaire** pour réaliser l'opération?

Plan

- Cycle de développement
- Identification des classes et des responsabilités
- Relations entre classes
- Où et comment chercher les classes, leurs responsabilités et leurs interactions?
- Bonnes pratiques et qualité d'une interface

Relations entre classes

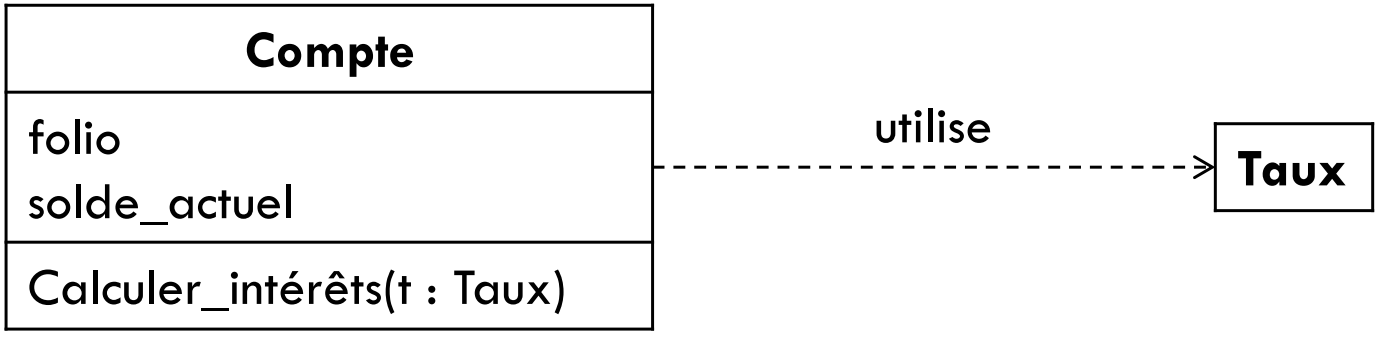
- **Dépendance** : « utilise », « connaît »
- **Association** : « contient », « a »
- **Héritage** : « est »

Dépendance entre classes

Une classe A dépend d'une classe B si A manipule des objets de B

C'est une relation d'utilisation : il existe une dépendance ponctuelle dans le temps

Exemple : une méthode d'une classe reçoit en paramètre les objets d'une autre classe



Dépendance entre classes

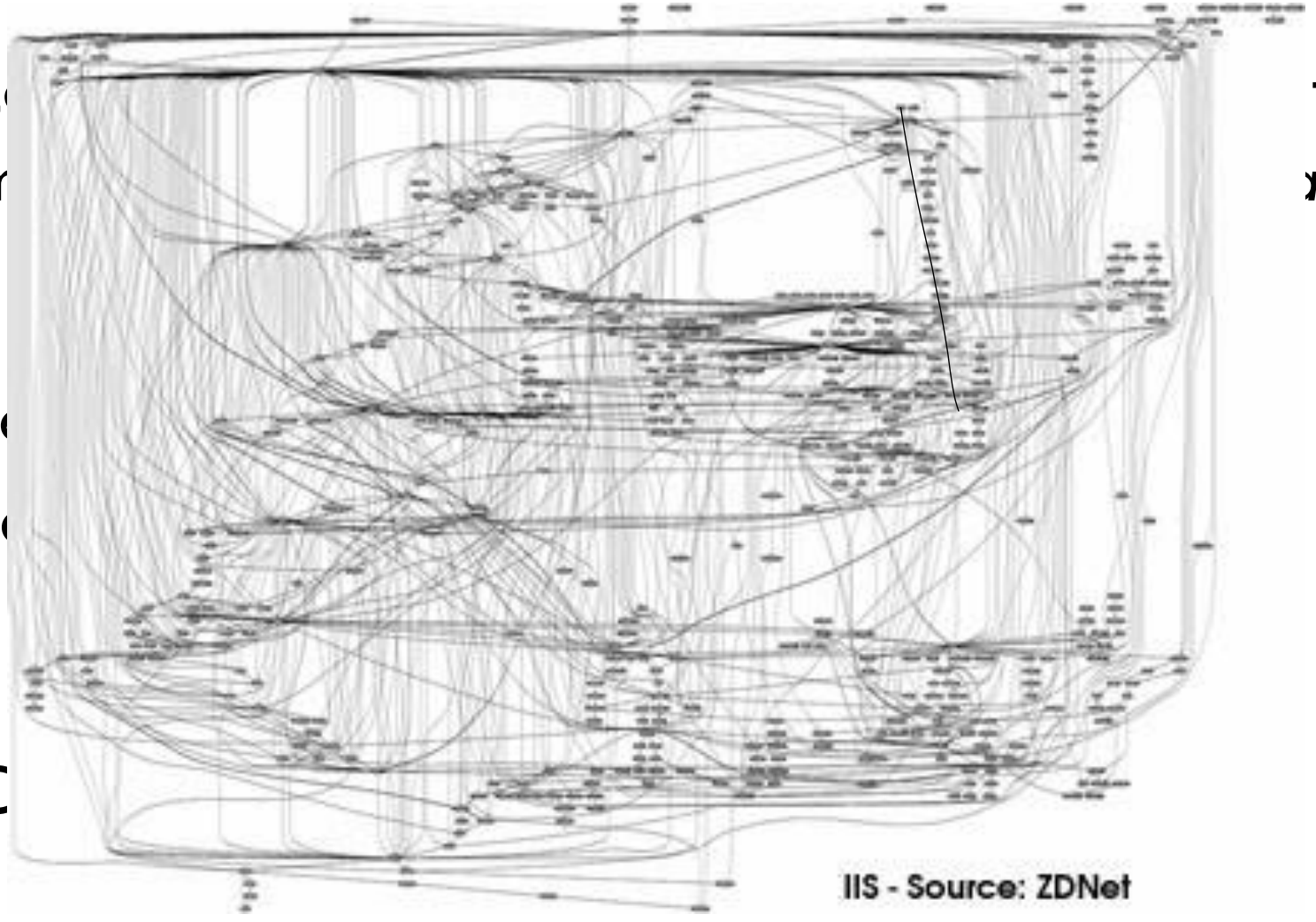
- Les règles de bonne conception stipulent qu'il faut minimiser la **dépendance** entre classes (*couplage*)

Pourquoi?

- Les classes sont plus faciles à **comprendre**
- La **maintenance** est plus facile
  - Limiter l'effet des changements d'une classe sur les autres classes du système
- Cela favorise la **réutilisation** des classes

Dépendance entre classes

- L faut
  m ge)
  - L
  - L les autres
  - C



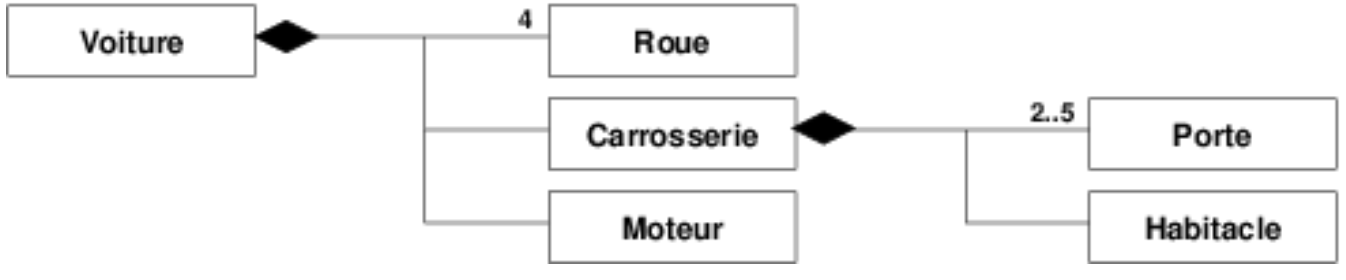
Association

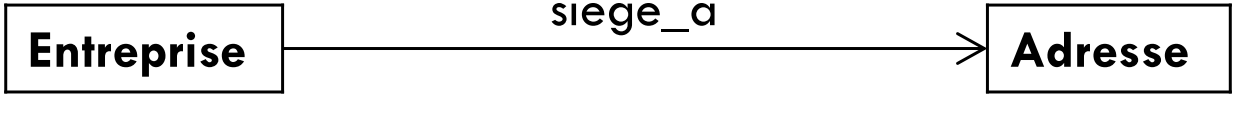
- Une classe A est associée à classe B si un objet de A **contient un ou plusieurs objets** de B
  - Exemple : *Company* a au moins un *Employee*

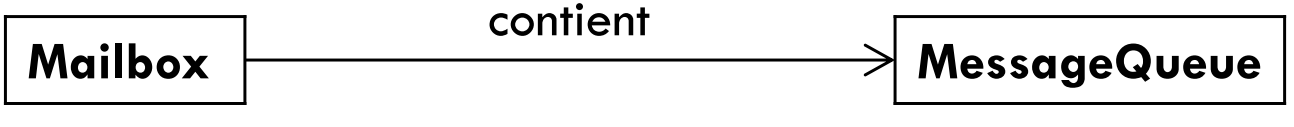
- Multiplicité : spécifie le nombre d'objets pouvant participer à cette dépendance
  - Exemples : 1 ; 0..1 ; 1..n ; 0..n

Association

- C'est un cas particulier de dépendance entre classes
- Peut-être une agrégation ou une composition







Association : exemple

- Un objet de type *FileMessage* contient 0 à N objets de type *Message*
- L'agrégation est implémentée par des **variables d'instances**

```
public class FileMessage {
  private ArrayList<Message> elements;
  //…
}
```

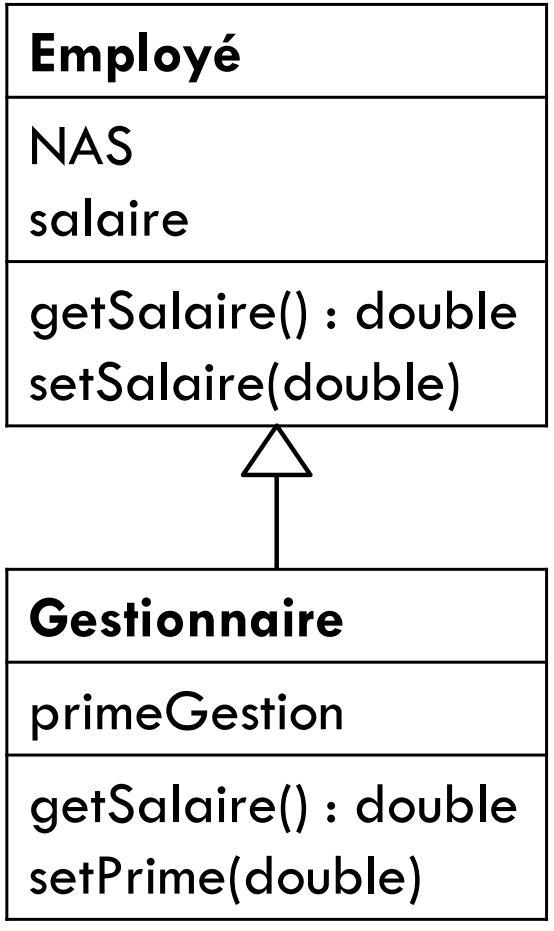
Héritage

- Une classe A hérite de la classe B si :
  - B est une **généralisation** de A
  - A est une **spécialisation** de B
    - A est un cas particulier de B
- Classe plus générale = *super-classe*
- Classe plus spécialisée = *sous-classe*

Héritage

- Une super-classe peut être substituée par sa sous-classe
  - La sous-classe supporte toutes les méthodes définies par la super-classe
    - L'implémentation d'une méthode peut être différente
- Une sous-classe étend sa classe parente en ayant, possiblement, des responsabilités et des états additionnels

Héritage : exemple



Plan

- Cycle de développement
- Identification des classes et des responsabilités
- Relations entre classes
- Où et comment chercher les classes, leurs responsabilités et leurs interactions?
- Bonnes pratiques et qualité d'une interface

Identifier les classes, etc.

- Où chercher?
  - Dans les **documents de spécification** produits lors de la phase d'analyse
    - Les exigences fonctionnelles sont décrites par un ensemble de **cas d'utilisation**
- Comment procéder?
  - Utiliser des **outils de conception**
    - Les fiches **CRC** (Classe, Responsabilité, Collaborateur)

Cas d'utilisation

- Principe introduit par Jacobson en 1986
- Le cas d'utilisation décrit une **manière d'utiliser** le système pour atteindre un objectif
- Il est décrit par une **séquence d'actions**
  - *Action* : interaction entre un acteur et un système informatique

- Un cas d'utilisation vise à produire un **résultat** ayant une valeur ajoutée pour l'un des acteurs

Cas d'utilisation

- Un cas d'utilisation peut contenir plusieurs **scénarios**
  - Un scénario **principal** qui décrit le plus courant
    - Scénario complet et réussi
  - Des scénarios appelés **variations**
    - Situations particulières
    - Scénario d'échec

Un cas d'utilisation peut être décrit selon différents formats

Exemple de cas d'utilisation

- **Laisser un message**

1. L'appelant compose le numéro principal du système de messagerie
2. Le système répond avec le message d'invitation :

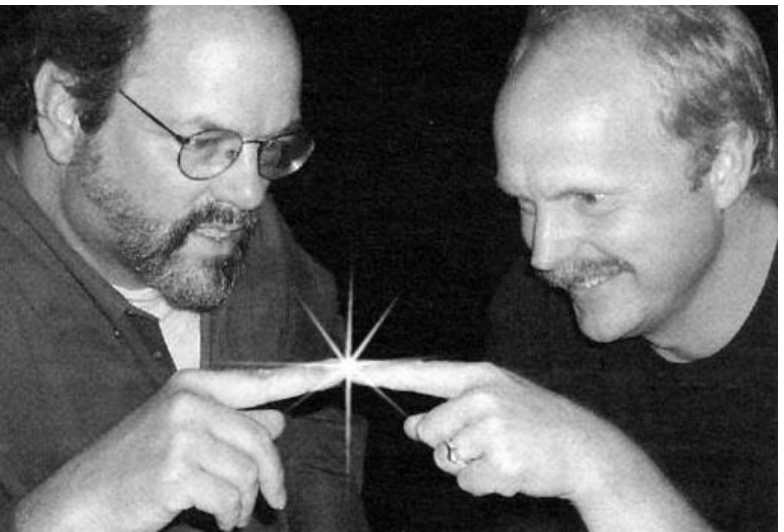
   « *Entrez le numéro de boîte aux lettres suivi de #* »
3. L'usager entre le numéro d'extension
4. Le système dit : « *Vous avez atteint la boîte aux lettres xxxx. Veuillez laisser un message après le bip* »
5. L'appelant dit son message
6. L'appelant raccroche
7. Le système met le message dans la boîte vocale désignée

Exemple de cas d'utilisation

- **Variation 1**
  - 1.1. À l'étape 3, l'usager entre un numéro d'extension invalide
  - 1.2. Le système de messagerie vocale dit : « *You have entered an invalid mailbox number* »
  - 1.3. Continuer avec l'étape 2
- **Variation 2**
  - 2.1. Après l'étape 4, l'appelant raccroche au lieu de laisser un message
  - 2.3. Le système de messagerie vocale écarte le message vide

Fiches CRC

- Développé par Beck et Cunningham en 1989
- CRC = **Classes, Responsabilités, Collaborations**
- Une fiche CRC permet de lister une classe, ses responsabilités et ses collaborateurs



Nom de la classe	
Responsabilités	Collaborateurs

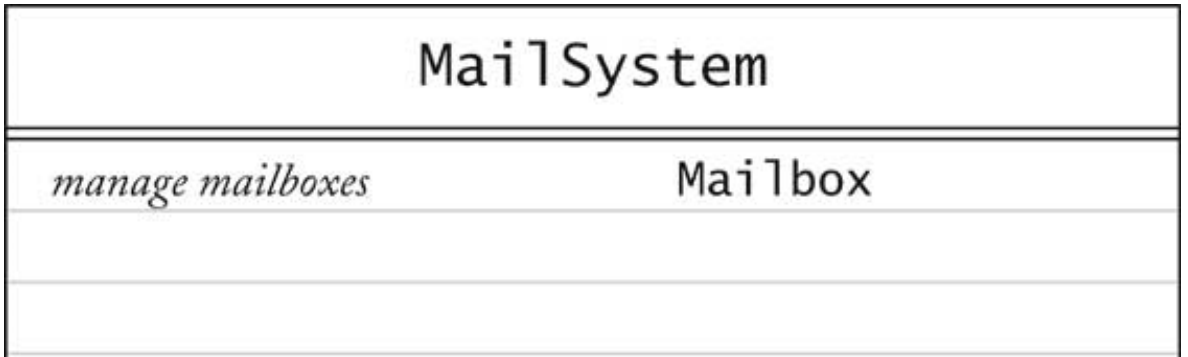
Fiches CRC

- Règles d'une bonne fiche :
  - Une responsabilité devrait être de **haut niveau**
    - Une responsabilité peut être réalisée par plusieurs méthodes
  - Nombre **limité** de responsabilités
    - Une à trois responsabilités par fiche

Mailbox	
manage passcode	MessageQueue
manage greeting	
manage new and saved messages	

Fiches CRC

- Les CRCs
  - Une façon intuitive de parcourir les cas d'utilisation
  - Permettent de répondre de façon collaborative à des questions de conception
- Dans l'étape 3 du cas d'utilisation « Laisser un message », l'appelant entre un numéro d'extension
- Un objet du système de messagerie vocale doit repérer la boîte vocale. Lequel?
- La classe *MailSystem* connaît toutes les boites de messagerie



Plan

- Cycle de développement
- Identification des classes et des responsabilités
- Relations entre classes
- Où et comment chercher les classes, leurs responsabilités et leurs interactions?
- Bonnes pratiques et qualité d'une interface

Encapsulation

Effets de bord d'une méthode :

- Tout changement d'état observable lorsque la méthode est appelée
  - Effet secondaire (*side effect*)
  - Un mutateur a l'effet de changer son paramètre **implicite**

Encapsulation

Effets de bord d'une méthode :

- Une méthode peut avoir d'autres effets
  - Modifier un paramètre **explicite**
  - Modifier un objet **statique** accessible
- Évitez ces effets secondaires
  - L'utilisateur ne s'attend pas à un changement des paramètres explicites
- Bon exemple : pas d'effet au-delà du changement du paramètre implicite

```
a.addAll(b); // change a mais pas b
```

Encapsulation

Paramètre explicite

```
01  public class Parameters {
02
03      public static void main(String[] args) {
04          // TODO Auto-generated method stub
05          JamesSquare james = new JamesSquare();
06          james.getArea(20);
07      }
08
09  }
10
11
12  class JamesSquare {
13      public int getArea(int side){
14          return side*side;
15      }
16  }
```

`side` est un paramètre explicite

Paramètre implicite

```
01  public class Parameters {
02
03      public static void main(String[] args) {
04
05          String str1="James";
06          str1.length(); //5
07
08          String str2="Bun";
09          str2.length(); //3
10      }
11  }
```

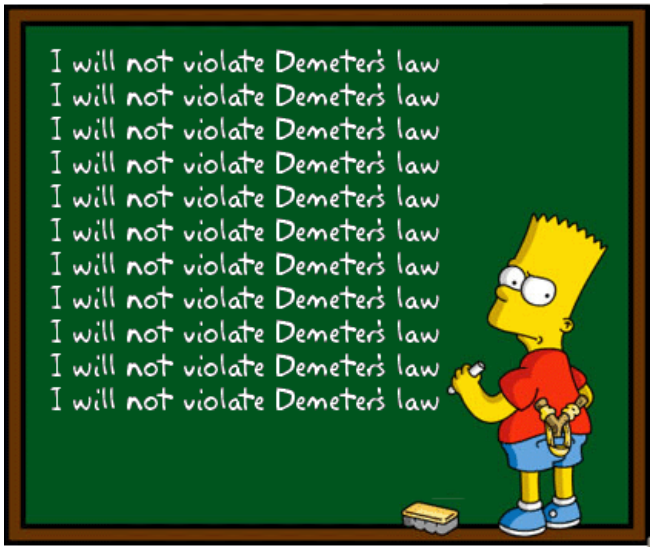
`str1` et `str2` sont des paramètres implicites

Encapsulation

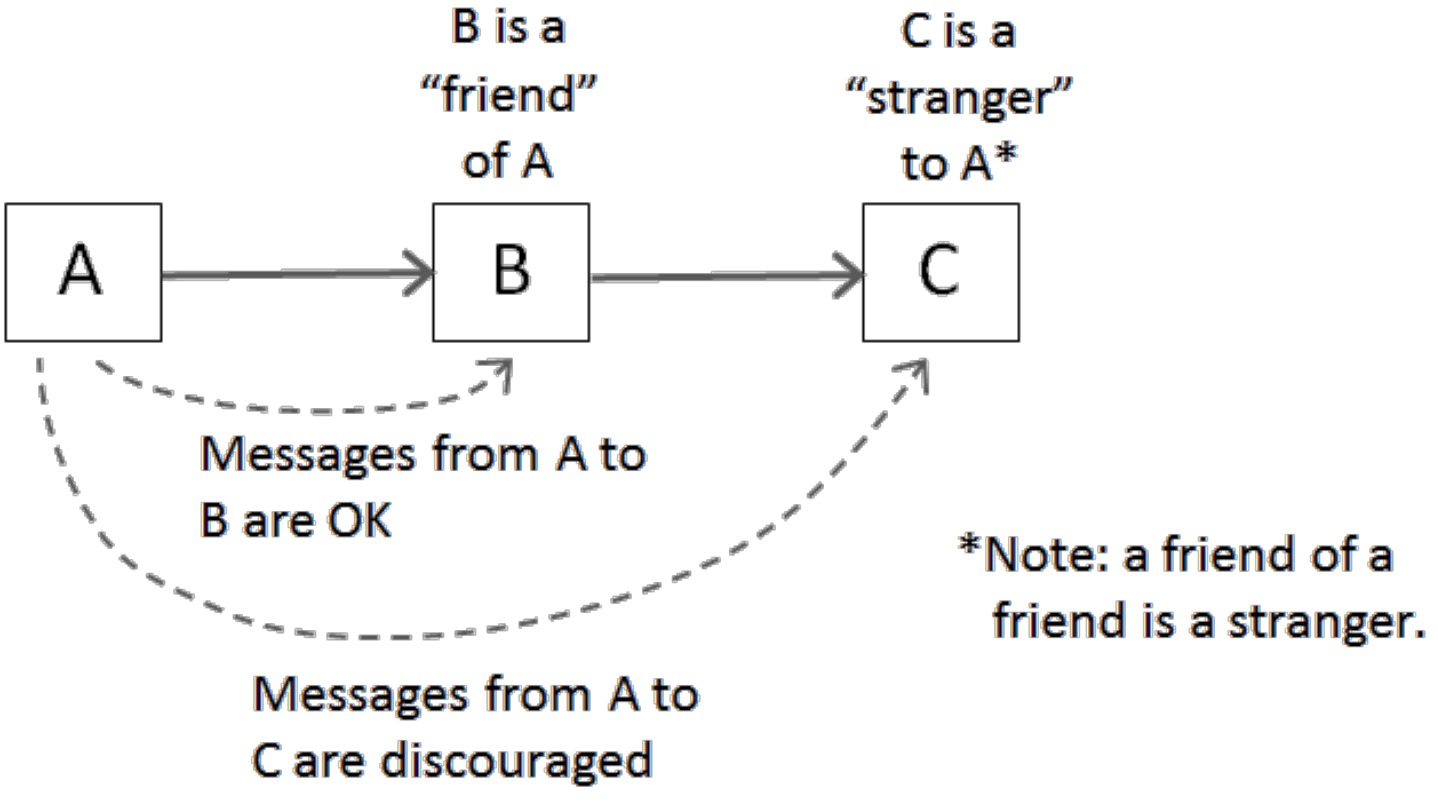
- **Loi de Déméter** (principe de connaissance minimale)
  - Connaissance **limitée** des autres classes
    - Communiquer avec ses relations **directes** (classes voisines)
    - Ne pas dépendre de leur implémentation
    - Éviter de récupérer un objet dans un autre objet
    - Un objet est le **seul** à connaître sa structure interne
  - Une méthode devrait *utiliser* uniquement :
    - ses paramètres (implicite **et** explicites)
    - les variables de sa classe
    - des objets construits dans la méthode

Encapsulation

- Attention aux *getters* systématiques
  - Une classe n'est pas qu'une structure de données
  - Éviter d'utiliser un objet retourné par un appel de méthode (à moins qu'il ne soit **créé** par cette méthode)
  - La classe doit prendre toute la responsabilité d'interaction avec cet objet
- Avantages & inconvénients
  - Moins de dépendance, plus d'adaptabilité
  - Maintenance plus simple
  - Davantage de code, surcharge



Encapsulation



Source: https://betterprogramming.pub/demeters-law-don-t-talk-to-strangers-87bb4af11694

Encapsulation (Loi de Déméter)

- Exemple:

```
class Employee {

    private Department department = new Deparment();

    public Department getDepartment() {
        return department;
    }
}

class Manager {
    public void approveExpense(Expenses expenses) {
        System.out.println("Total amounts approved" + expenses.total())
    }
}
```

```
class Department {
    private Manager manager = new Manager();

    public Manager getManager() {
        return manager;
    }
}
```

Encapsulation (Loi de Déméter)

- Exemple:

```
class Expenses {

    private double total;
    private double tax;

    public Expenses(double total, double tax) {
        this.total = total;
        this.tax = tax;
    }

    public double total() {
        return total + tax;
    }
}
Expenses expenses = new Expenses(100, 10);
Employee employee = new Employee();
employee.getDepartment().getManager().approveExpense(expenses);
```

Source: https://www.baeldung.com/java-demeter-law#:~:text=The%20Law%20of%20Demeter%20is,interact%20with%20its%20immediate%20dependencies.

Qualité de l'interface d'une classe

- Deux vues sur la conception et l'implémentation d'une classe
  - Objectifs du programmeur qui ***conçoit*** la classe
    - Facilité de codage
    - Algorithmes efficaces (et une conception efficace)
  - Objectifs du programmeur qui ***utilise*** la classe
    - Comprendre et utiliser les opérations de la classe sans avoir à connaître les détails d'implémentation
    - Avoir accès à une interface simple mais qui couvre complètement ses besoins

Qualité de l'interface d'une classe

- Critères de qualité de l'interface d'une classe (5C)
  - Cohésion
  - Complétude
  - Convenance
  - Clarté
  - Cohérence

- Activité d'ingénierie : faire des compromis
  - Clarté versus convenance et complétude, etc.
  - Pas de consensus général sur la notion de « bonne conception »
    - Ça dépend des objectifs

Qualité de l'interface d'une classe

- Cohésion
  - Une classe est une **abstraction** d'un seul concept
  - Les méthodes de la classe doivent être reliées à une seule abstraction, sinon il faut plusieurs classes
  - Exemple de classe non cohésive :

```
public class Mailbox {
  public void addMessage(Message aMessage) { ... }
  public Message getCurrentMessage() { ... }
  public Message removeCurrentMessage() { ... }
  public void processCommand(String command) { ... }
  // ...
}
```

Qualité de l'interface d'une classe

- Complétude
  - Supporter toutes les opérations qui font partie de l'abstraction représentée par la classe
  - Exemple *potentiellement* mauvais
    - L'utilisateur de la classe `Date` aimerait savoir combien de millisecondes se sont écoulées entre deux objets de type `Date`

```
Date start = new Date();
Date stop = new Date();
```

      - La classe `Date` n'offre pas d'opération de ce genre
      - Est-ce vraiment la responsabilité de la classe `Date`?
        - `Date` offre d'autres méthodes (`before`, `after`, `getTime`)

```
long difference = stop.getTime() - start.getTime();
```

Qualité de l'interface d'une classe

- Convenance (ou commodité)
  - Une bonne interface ne doit pas juste supporter toutes les tâches, elle doit aussi rendre simples les tâches communes
  - Mauvais exemple :
    - Lire une ligne de texte à partir de `System.in` avant Java 5.0

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

    - `System.out` a une méthode `println`, pourquoi `System.in` n'a pas de méthode `readLine`?
    - Le problème a été réglé avec la classe `Scanner` dans Java 5.0

Qualité de l'interface d'une classe

Clarté

- L'interface d'une classe doit être **claire** et sans confusion
- Mauvais exemple :

```
LinkedList<String> maListe = new LinkedList<String>();
maListe.add("A");
maListe.add("B");
maListe.add("C");
```

  - Itérer sur la liste :

```
ListIterator iterator = maListe.listIterator();
while (iterator.hasNext())
  System.out.println(iterator.next());
```

  - **Position** d'un itérateur : entre deux éléments, comme la barre verticale indiquant la position du curseur dans un éditeur de texte

Qualité de l'interface d'une classe

- Clarté (suite)
  - `add` ajoute à gauche de l'itérateur

```
ListIterator iterator = maListe.listIterator(); // |ABC
iterator.next(); // A|BC
iterator.add("X"); // AX|BC
```

  - La méthode `remove` n'est pas intuitive
    - Pour supprimer les deux premiers éléments, on ne peut pas juste utiliser « espace arrière »
      - remove ne supprime pas les éléments à gauche de l'itérateur
      - « *Supprime de la liste le **dernier élément** qui a été **retourné** par next ou previous* »
      - Seulement une fois, et pas après add

Qualité de l'interface d'une classe

- Cohérence
  - Les méthodes d'une classe devraient avoir une cohérence entre elles en termes de :
    - noms
    - paramètres
    - valeurs de retour
    - comportement
  - Mauvais exemple : classe `GregorianCalendar`

```
new GregorianCalendar(year, month - 1, day)
```

    - Pourquoi les mois sont-ils numérotés de 0 à 11 alors que les jours sont bien numérotés de 1 à 31?

Qualité de l'interface d'une classe

- Cohérence (suite)
  - Mauvais exemple : classe `String`
    - `equals` et `equalsIgnoreCase`
    - `compareTo` et `compareToIgnoreCase`
    - Mais ce n'est pas la même démarche pour la méthode `regionMatches`

```
boolean regionMatches(int toffset, String other, int ooffset, int len)
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
```

    - Pourquoi pas `regionMatchesIgnoreCase`?