Développement Orienté Objets

Décrire des classes en UML et en Kotlin

Arnaud Lanoix Brauer
Arnaud.Lanoix@univ-nantes.fr

IUT de Nantes
Départemement informatique

Sommaire

UML = Unified Modeling Language

- UML est un langage de modélisation visuel à base de pictogrammes permettant de représenter l'ensemble des facettes d'un projet (informatique).
- UML définit un ensemble de notations graphiques permettant de décrire tous les concepts utiles lors des différentes étapes d'analyse et de conception (orientée objet).
- `https://www.uml.org/`

Orthographe des pictogrammes

UML est un langage comme le français : il y a donc une "orthographe" et une "grammaire".
On n'utilise pas un □ à la place d'un ◊ ou une ⇾ à la place d'une ⇢

UML = Unified Modeling Language

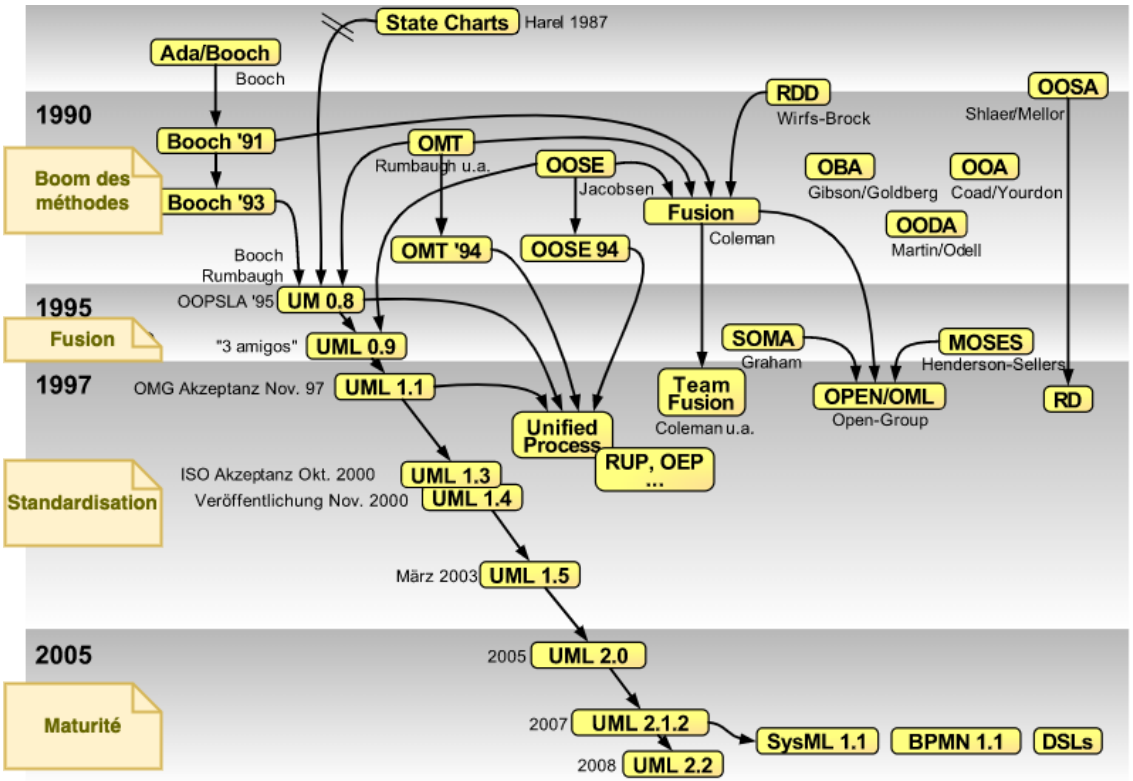
- UML est un langage de modélisation visuel à base de pictogrammes permettant de représenter l'ensemble des facettes d'un projet (informatique).
- UML définit un ensemble de notations graphiques permettant de décrire tous les concepts utiles lors des différentes étapes d'analyse et de conception (orientée objet).
- `https://www.uml.org/`

Orthographe des pictogrammes

UML est un langage comme le français : il y a donc une "orthographe" et une "grammaire".
On n'utilise pas un $\square$ à la place d'un $\Diamond$ ou une $\rightarrow\!\!\triangleright$ à la place d'une $\dashrightarrow$

(bref) historique d'UML

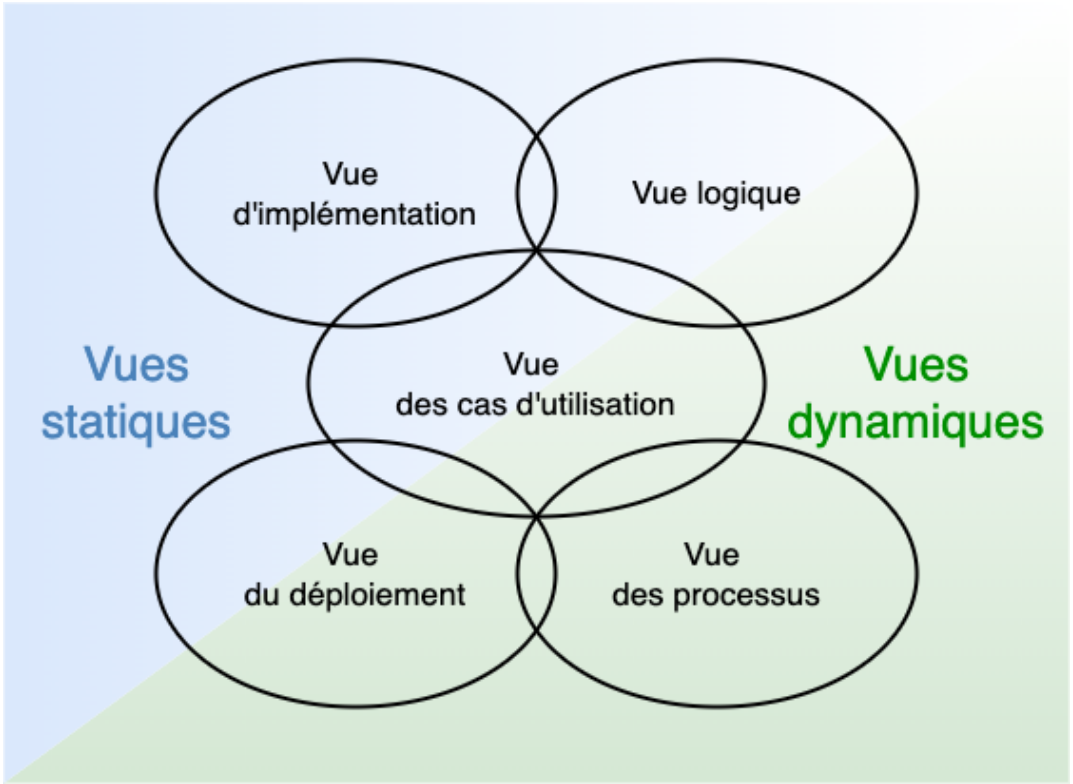
- élaboré par *Rumbaugh*, *Booch* et *Jacobsen* (entre autre) à partir de 1995
- = fusion de différentes méthodes : OMT, OOSE, statechart, etc.
- Normalisé par l'OMG (Object Management Group) [1]
- Première version officielle 1.1 en 1997
- Actuellement, version 2.5.1, datant de 2017



1. `https://www.omg.org/`

Facettes d'un système vs. UML

Un système (informatique) est trop complexe pour être décrit par un seul diagramme : il y a différentes facettes qui représentent chacune des aspects différents du système (statique, dynamique, temporel, etc.)



Des diagrammes spécifiques permettent de décrire tous ces aspects.

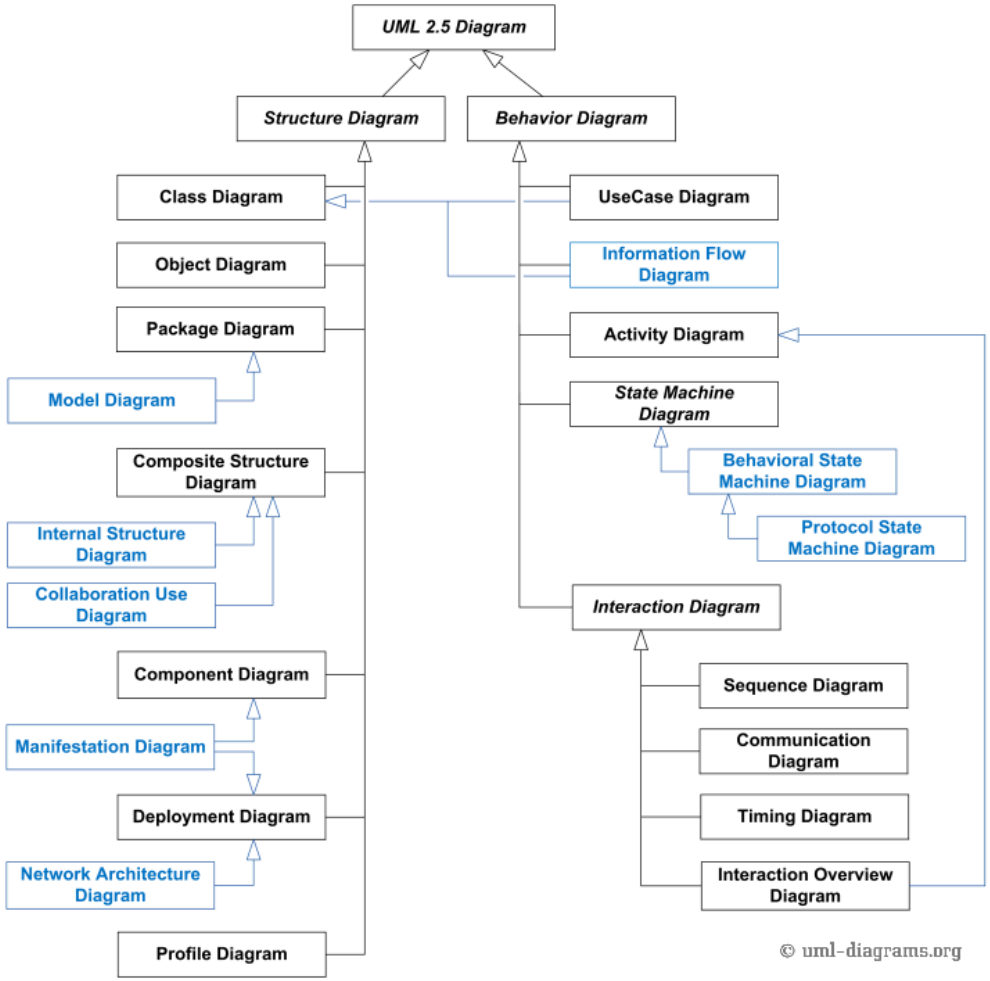
Diagrammes en UML 2.5

Diagrammes de structure (statiques)

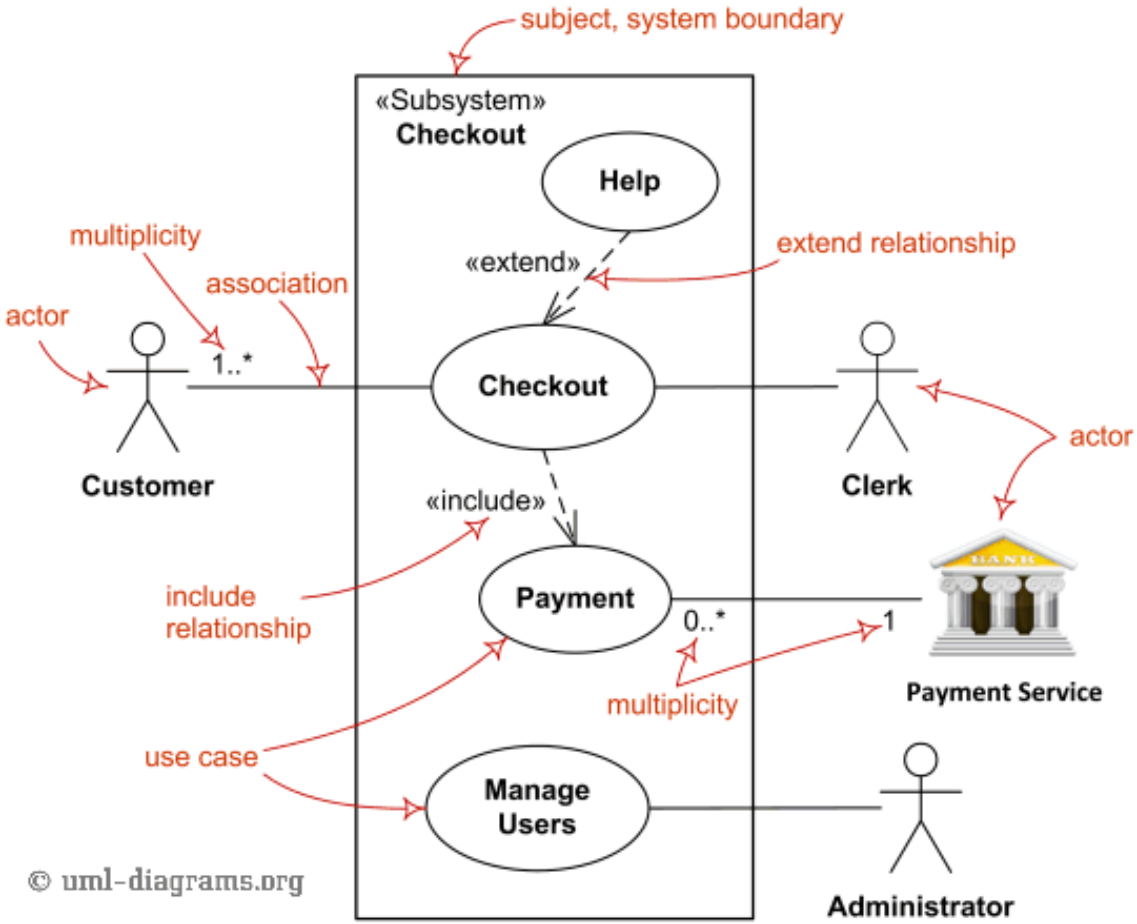
- **Diagramme de classes**
- Diagramme d'objets
- Diagramme de paquets
- Diagramme de structure composite
- Diagramme de composants
- Diagramme de déploiement
- Diagramme de profils

Diagrammes de comportement (dynamiques)

- Diagramme des cas d'utilisation
- Diagamme d'activité
- Diagramme états-transitions
- *Diagrammes d'interaction*
  - **Diagramme de séquence**
  - Diagramme de communication
  - Diagramme de temps
  - Diagramme global d'interaction

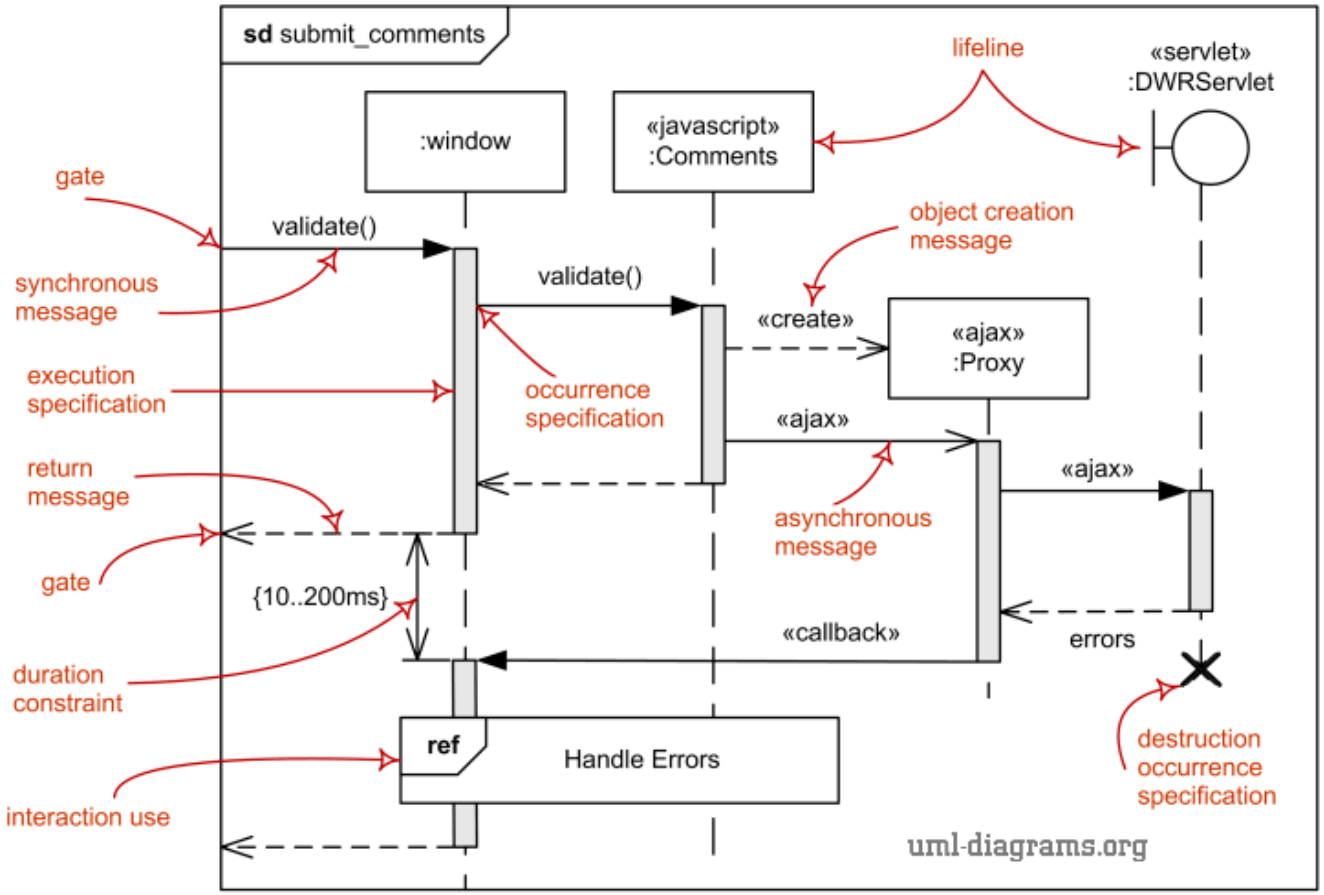


Exemple de diagramme de cas d'utilisation

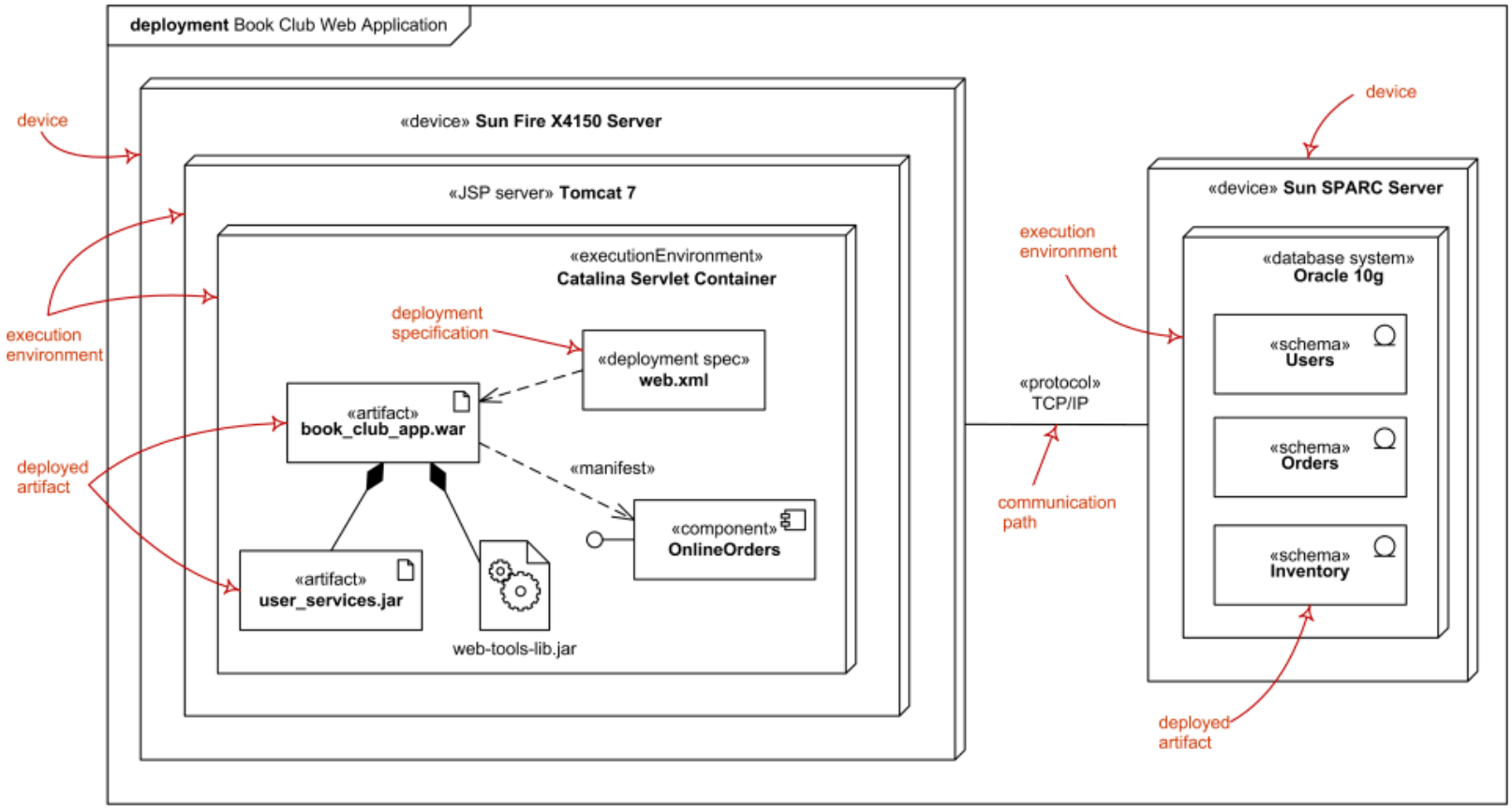


détaillé dans le module GPO2 par Jean-Marie Mottu

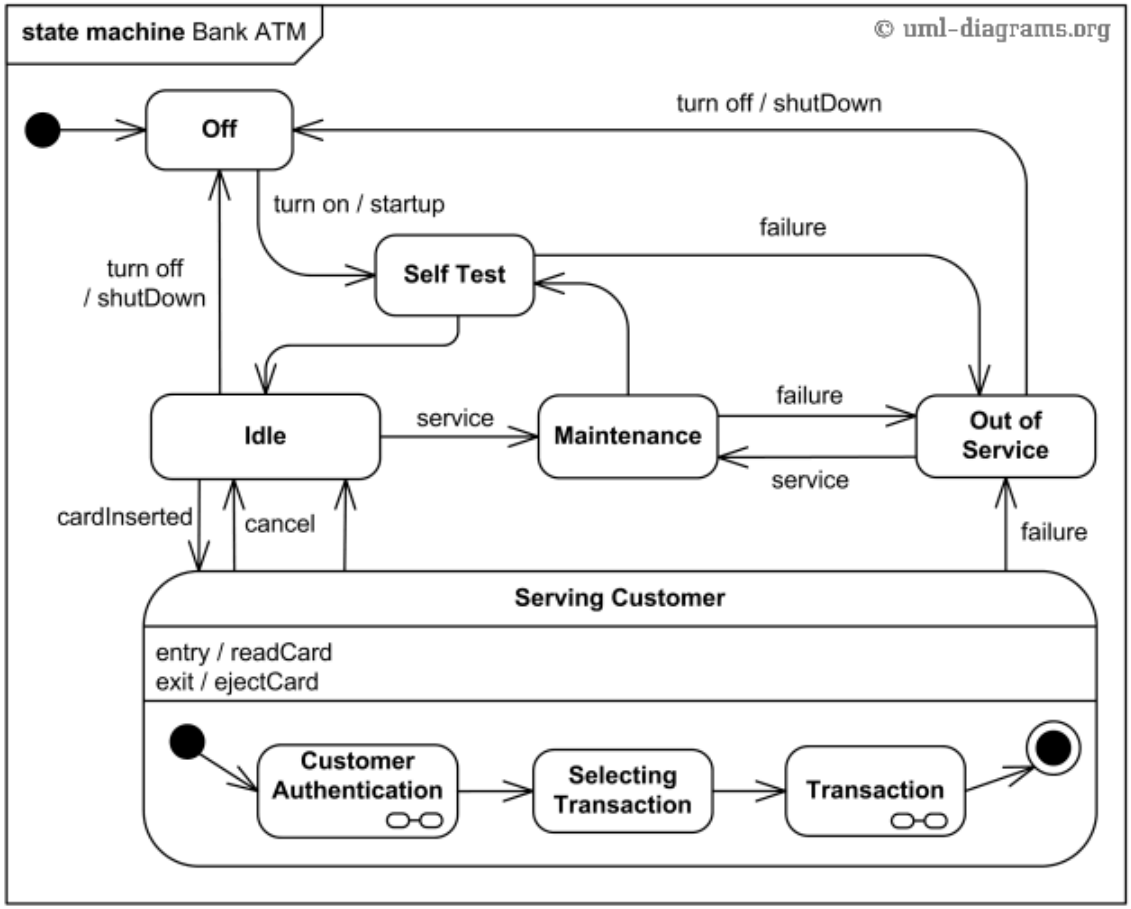
Exemple de diagramme de séquence



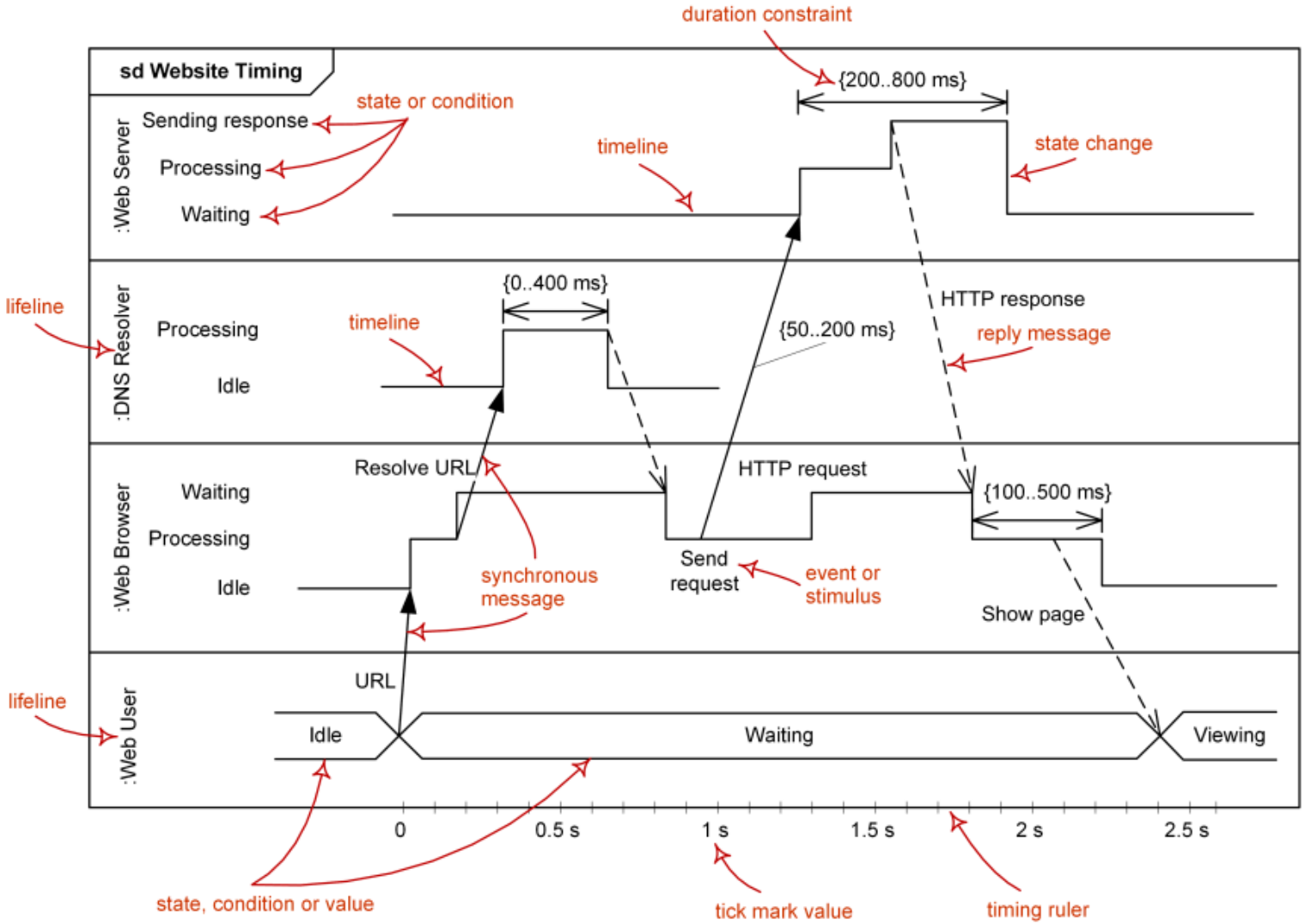
Exemple de diagramme de déploiement



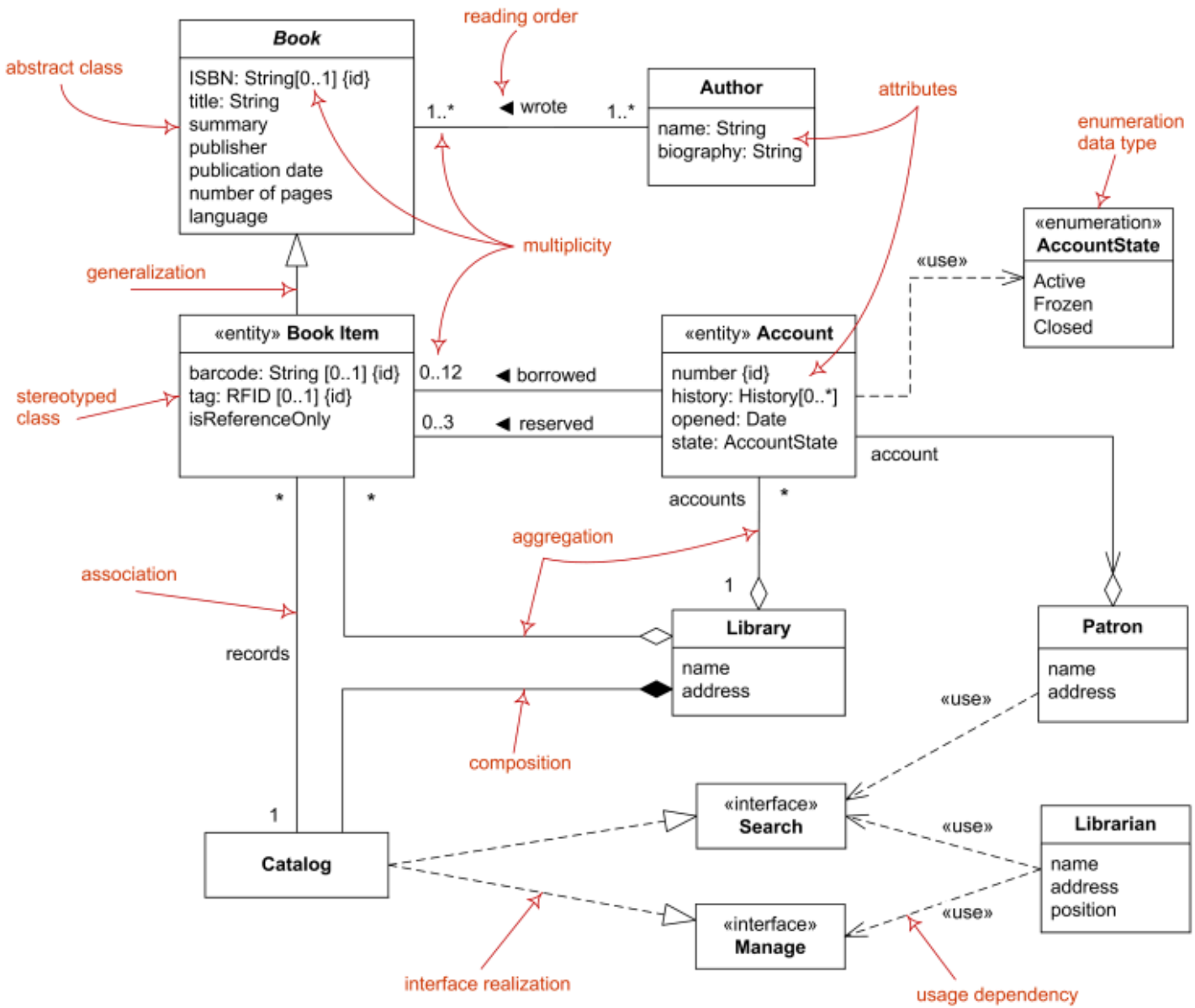
Exemple de diagramme états-transitions



Exemple de diagramme de temps



Exemple de diagramme de classes



Sommaire

Rappel : qu'est-ce qu'une classe ?

= sorte de "moule" [2] pour définir des objets, qui précise

- les propriétés qui définissent la structure interne des objets (= les attributs)
- Les interactions qu'offrent les objets, les comportements possible pour les objets (= les méthodes)
- Les (éventuels) liens d'héritage
- ...

La classe **Citoyen**

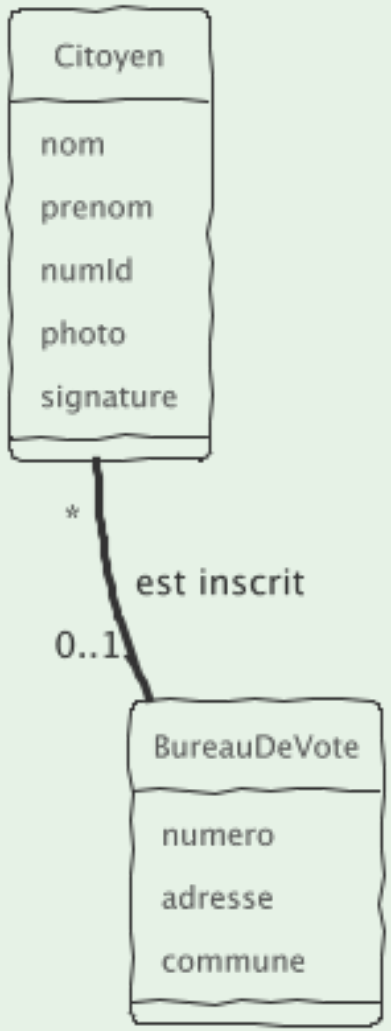
- nom, prénom, date de naissance,
- numéro carte d'identité,
- photo,
- signature,
- ...

2. "patron", "plan"", "schéma de construction", ...

Diagramme de classes UML

- Diagramme utilisé pour représenter graphiquement les données métier d'un système ainsi que leurs relations
- en termes de classes et d'associations
- Une classe décrit des caractéristiques de base liées ensemble sémantiquement (=attributs) ET des interactions possibles (= méthodes)
- Un objet est une instance d'une classe
- Une association décrit une relation sémantique unissant des instances des classes

Exemple Citoyen : un premier diagramme de classes



- 1 classe `Citoyen` qui regroupe les caractéristiques nom, prénom, numId, photo, ... définissant un citoyen
- 1 classe `BureauDeVote` qui regroupe les caractéristiques numero, adresse, commune, ... définissant un bureau de vote
- 1 association `est inscrit` qui caractérise une relation entre un citoyen et un bureau de vote
  elle indique qu'un citoyen donné est inscrit dans un bureau de vote donné et qu'un bureau de vote donné est le lieu d'inscription de plusieurs citoyens donnés

2 niveaux de diagramme de classes

- **niveau *Analyse* :** on précise les entités métiers et leurs relations
  quelques méthodes-clefs pertinentes, certains types, etc.
  But : capturer le domaine métier
- **niveau *Conception* :** on *prépare* l'implémentation
  - on précise tous les types de tous les attributs,
  - on précise toutes les méthodes,
  - certaines classes peuvent disparaitre,
  - des classes peuvent apparaître (des classes "outils")
  - ...

  But : obtenir un "plan" précis à implémenter

2 niveaux de diagramme de classes

- **niveau *Analyse* :** on précise les entités métiers et leurs relations
  quelques méthodes-clefs pertinentes, certains types, etc.
  But : capturer le domaine métier **de quoi on va parler ?**
- **niveau *Conception* :** on *prépare* l'implémentation
  - on précise tous les types de tous les attributs,
  - on précise toutes les méthodes,
  - certaines classes peuvent disparaitre,
  - des classes peuvent apparaître (des classes "outils")
  - ...

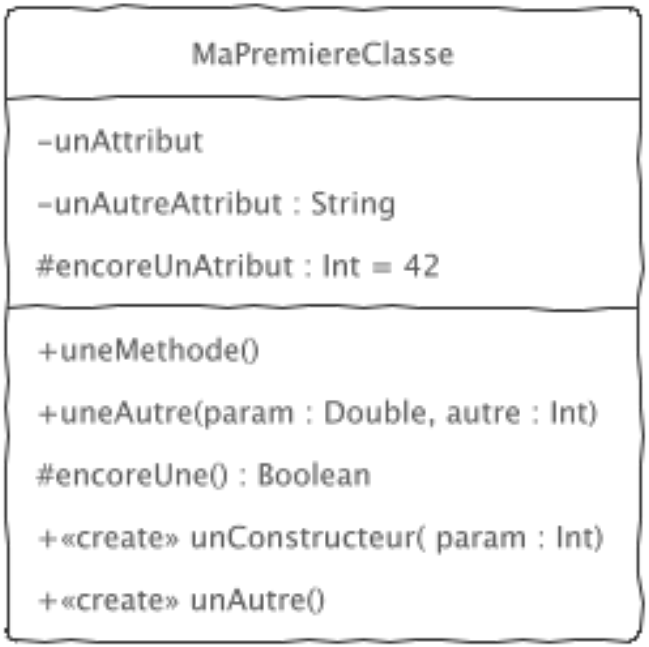
  But : obtenir un "plan" précis à implémenter

2 niveaux de diagramme de classes

- **niveau *Analyse* :** on précise les entités métiers et leurs relations
  quelques méthodes-clefs pertinentes, certains types, etc.
  But : capturer le domaine métier **de quoi on va parler ?**
- **niveau *Conception* :** on *prépare* l'implémentation
  - on précise tous les types de tous les attributs,
  - on précise toutes les méthodes, **comment on va**
  - certaines classes peuvent disparaitre, **le réaliser ?**
  - des classes peuvent apparaître (des classes "outils")
  - ...

  But : obtenir un "plan" précis à implémenter

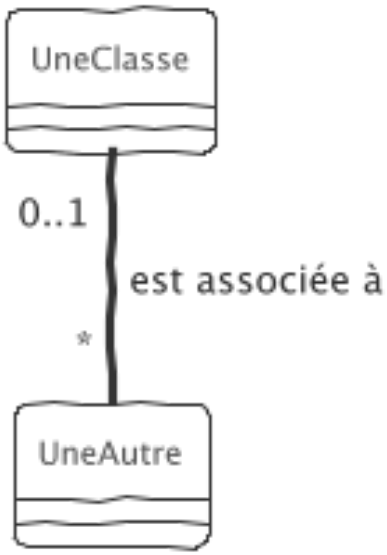
Décrire une classe en UML



- Un nom de classe
  - = forme nominale
- Une liste d'attributs
  - typés ou non
  - avec une initialisation possible
- Une liste de méthodes
  - des paramètres sont possibles
  - des types de retour
  - les constructeurs sont à préciser par `<<create>>`
- Les visibilités (des attributs/des méthodes)
  - L'élément est-il accessible par d'autres ou interne à la classe ?)

+	publique
–	privée
#	protégée

Décrire une association en UML



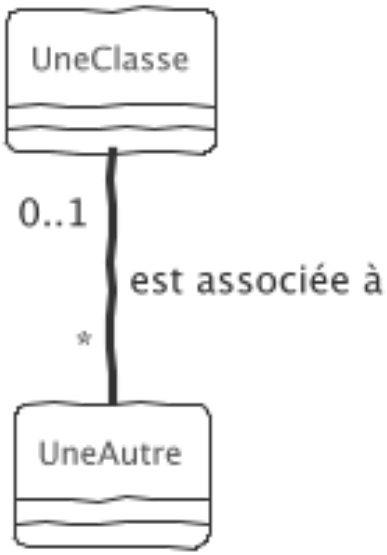
- Un nom d'association
  - = forme verbale exprimant la sémantique de la relation
- Des cardinalités
  - = nombre d'instances en jeu dans la relation

`1`	un et un seul
`0..1`	zéro ou un
`*`	plusieurs
`0..*`	de 0 à plusieurs
`1..*`	de 1 à plusieurs
`N..M`	entre N et M

Attention

Nom d'association **et** cardinalités sont **obligatoires**

Décrire une association en UML



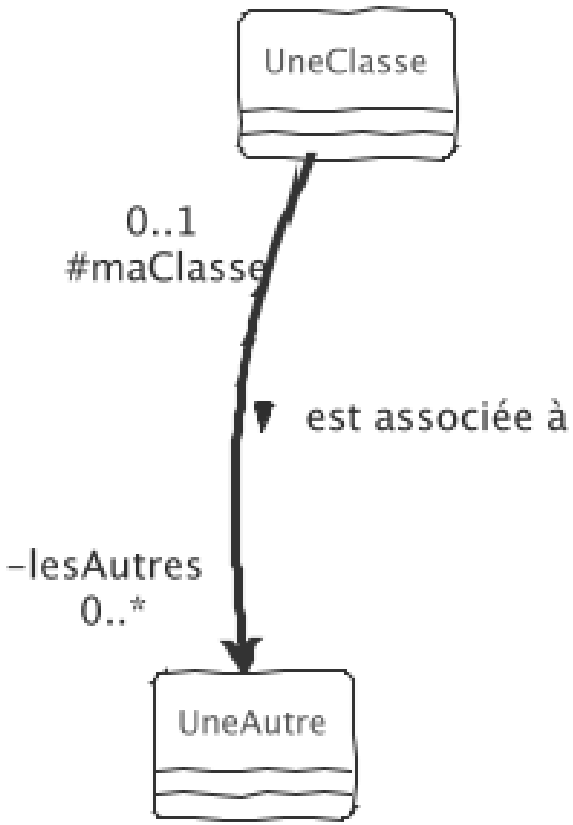
- Un nom d'association
  - = forme verbale exprimant la sémantique de la relation
- Des cardinalités
  - = nombre d'instances en jeu dans la relation

`1`	un et un seul
`0..1`	zéro ou un
`*`	plusieurs
`0..*`	de 0 à plusieurs
`1..*`	de 1 à plusieurs
`N..M`	entre N et M

Attention

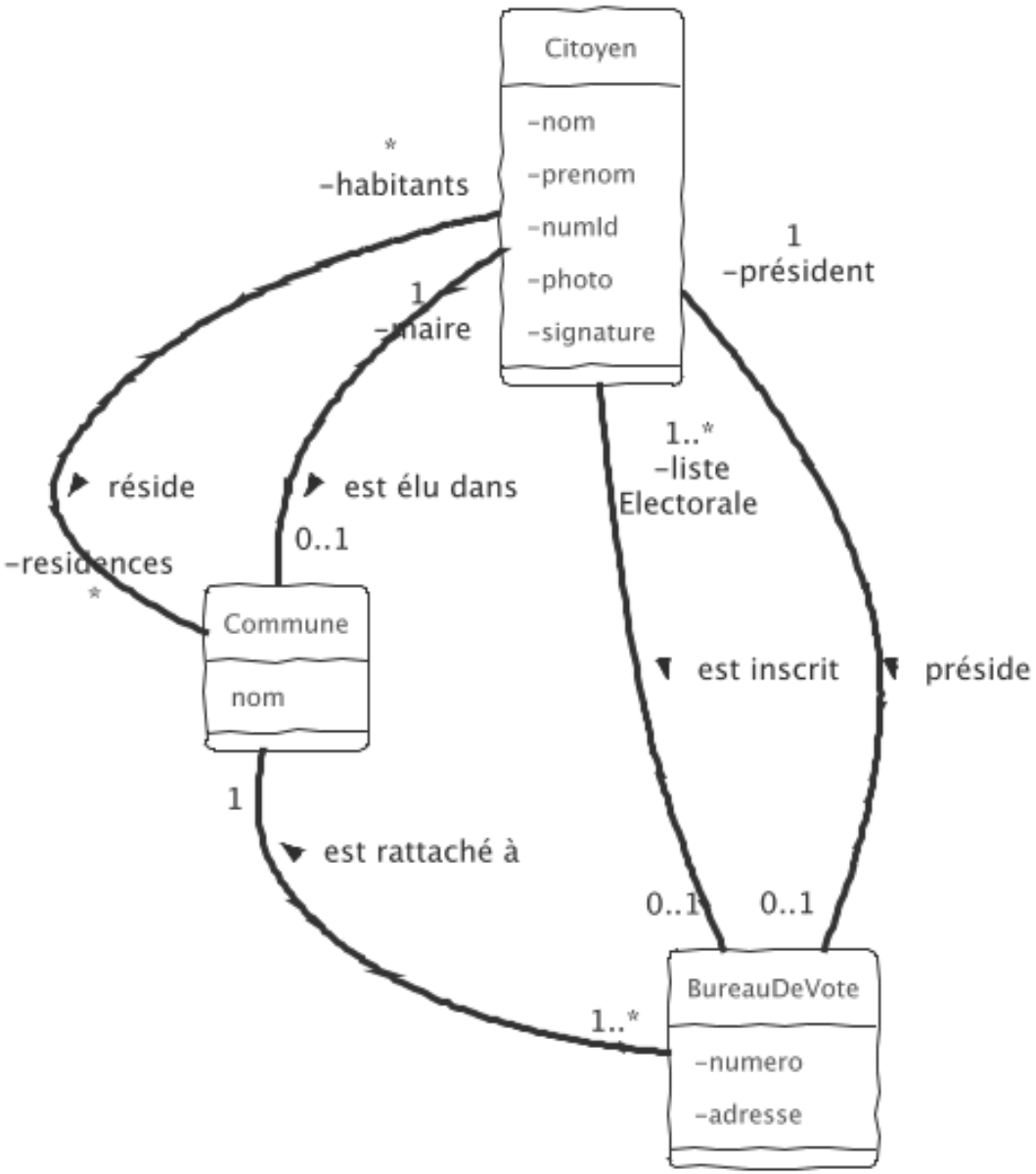
Nom d'association **et** cardinalités sont **obligatoires**

Décrire une association en UML : précisions possible



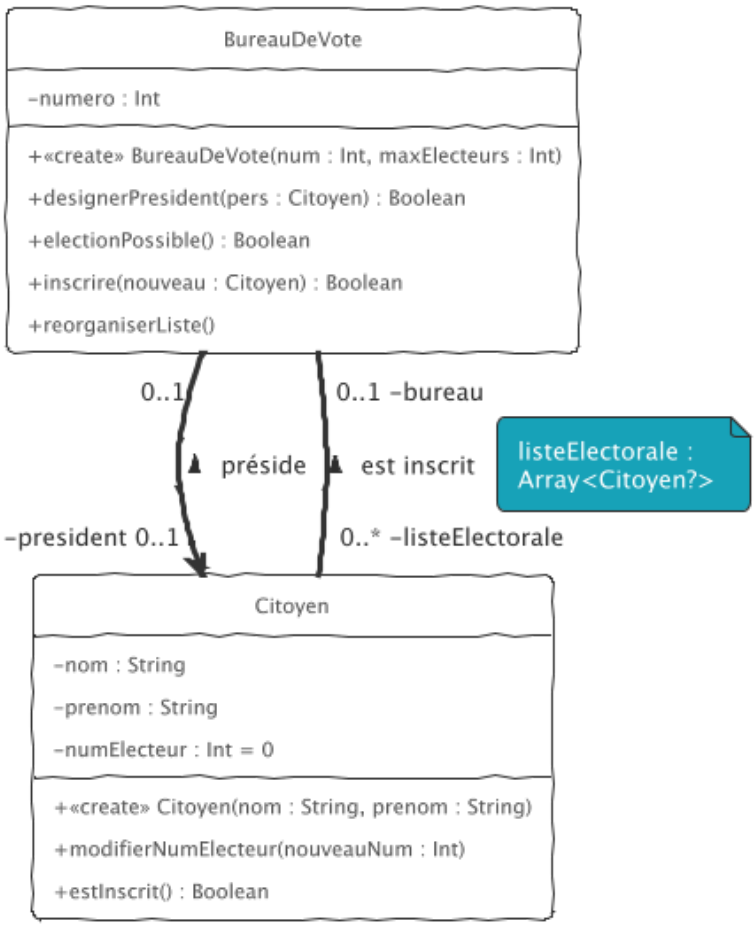
- un > à coté du nom de l'association peut donner le sens de lecture
- le rôle joué par une classe dans l'autre classe peut être précisé
  - la visiblité peut aussi être précisée
- La flèche sur l'association indique la navigabilité, c-à-d une association unidirectionnelle
  - sans navigabilité, l'association est bidirectionnelle (par défaut)

Exemple Citoyen : plusieurs associations



- Deux classes peuvent être reliées par plusieurs associations
- Les rôles précisent la sémantique de l'association

Exemple Citoyen : conception détaillée



- On précise les types des attributs
  - cela dépend du langage cible
- On ajoute des méthodes
  - On ajoute également des constructeurs
- On précise les navigabilités des associations
  - préférez des associations unidirectionnelles
- On précise comment seront implémentées les cardinalités $*$
  - cela dépend du langage cible
- ...

Aller plus loin

On reviendra plus tard sur

- des précisions sur les associations
  - aggrégation / composition
  - contraintes entre association
  - classes-association
- **les relations d'héritage**
  - classes abstraites et généralisation
  - interfaces et réalisation
- les attributs / les méthodes de classes
- les relations de dépendances
- les diagrammes d'objets / de paquets

Sommaire

Déclarer une classe en Kotlin

```kotlin
class Chien {
  var nom :String = ""
  var age : Int = 0          // en mois
  var race : String = ""
  var poids : Double = 0.0 // en kg

  fun aboyer() {
    println("$nom dit : ouaf !!!")
  }

  fun renommer(nouveauNom : String) {
    nom  = nouveauNom
  }

  // @param distance en m
  fun courir(distance : Int) {
    // le chien perd 1 g / km
    poids -= (distance / 1000.0) / 1000
  }

  fun ageEnAnnee() = age / 12.0
}
```

- Une classe est déclarée grâce au mot-clef `class`
- Les attributs sont déclarés comme des variables internes à la classe
  - Les attributs doivent être initialisés
- Les méthodes sont déclarées comme des fonctions internes à la classe
  - On peut consulter ou modifier les valeurs des attributs via les méthodes

Utiliser une classe en Kotlin

```
fun main() {
  var rogue = Chien()
  rogue.nom = "Rogue"
  rogue.age = 15
  rogue.race = "Berger Australien"
  rogue.poids = 30.2

  var potter = Chien()
  potter.nom = "Other"
  potter.age = 40
  potter.race = "Beauceron"
  potter.poids = 39.4

  rogue.aboyer()
  potter.renommer("Potter")
  println("${potter.nom}:
        ${potter.poids} kg")
  potter.courir(400_000)
  println("${potter.nom}:
        ${potter.poids} kg")
  var age = rogue.ageEnAnnee()
  println("${rogue.nom}: $age ans")
}
```

- On instancie des variables de type `Chien`
- On utilise la notation `.` pour
  - accéder aux attributs d'un objet
  - appeler des méthodes sur un objet

Attention

ce n'est pas une bonne idée d'initialiser les attributs un par un, comme ici :

- risque d'oublier certains attributs
- problème en cas de nouvel attribut

Il faut passer par un **constructeur**

Utiliser une classe en Kotlin

```
fun main() {
  var rogue = Chien()
  rogue.nom = "Rogue"
  rogue.age = 15
  rogue.race = "Berger Australien"
  rogue.poids = 30.2

  var potter = Chien()
  potter.nom = "Other"
  potter.age = 40
  potter.race = "Beauceron"
  potter.poids = 39.4

  rogue.aboyer()
  potter.renommer("Potter")
  println("${potter.nom}:
        ${potter.poids} kg")
  potter.courir(400_000)
  println("${potter.nom}:
        ${potter.poids} kg")
  var age = rogue.ageEnAnnee()
  println("${rogue.nom}: $age ans")
}
```

- On instancie des variables de type `Chien`
- On utilise la notation `.` pour
  - accéder aux attributs d'un objet
  - appeler des méthodes sur un objet

Attention

ce n'est pas une bonne idée d'initialiser les attributs un par un, comme ici :

- risque d'oublier certains attributs
- problème en cas de nouvel attribut

Il faut passer par un **constructeur**

Constructeur en Kotlin

```
class Chien (monNom : String, race : String = "inconnue", poids : Double) {
  var nom :String
  var age : Int = 1       // en nb de mois
  val race : String
  var poids : Double  // en kg

  init {
      nom = monNom
      this.race = race
      this.poids = if (poids > 0.0) poids else 0.1
  }
```

- les paramètres du constructeur sont déclarés après le nom de la classe
- On peut définir des valeurs par défaut pour les paramètres
- tous les attributs ne sont pas forcément présents comme paramètres
- les paramètres du constructeur servent à initialiser les attributs dans la clause `init {...}`
- tous les attributs doivent être initialisés
- si un attribut et un paramètre portent le même nom, on distingue l'attribut grâce à `this.`
  - de préférence, soyez uniforme, pas comme ici

Constructeur en Kotlin

```
class Chien (monNom : String, race : String = "inconnue", poids : Double) {
  var nom :String
  var age : Int = 1       // en nb de mois
  val race : String
  var poids : Double  // en kg

  init {
      nom = monNom
      this.race = race
      this.poids = if (poids > 0.0) poids else 0.1
  }
```

- les paramètres du constructeur sont déclarés après le nom de la classe
- On peut définir des valeurs par défaut pour les paramètres
- tous les attributs ne sont pas forcément présents comme paramètres
- les paramètres du constructeur servent à initialiser les attributs dans la clause `init {...}`
- tous les attributs doivent être initialisés
- si un attribut et un paramètre portent le même nom, on distingue l'attribut grâce à `this.`
  - de préférence, soyez uniforme, pas comme ici

Constructeur en Kotlin : utilisation

```
var rogue = Chien("Rogue", "Berger Australien", 30.2)
var potter = Chien("Other", "Beauceron", 39.4)
var soukie = Chien("Soukie", 4.7) // race non precisee => valeur par defaut

rogue.aboyer()
potter.renommer("Potter")
soukie.aboyer()
```

Les objets sont instanciés par l'appel du constructeur.

Attributs déclarés immuables (`val`)

Cela peut avoir beaucoup de sens de déclarer certains attributs immuables afin d'exprimer qu'une fois initialisés, ils ne peuvent plus être modifiés

```
class Chien(nom : String, age :Int = 1,
            race : String = "inconnue", poids : Double) {
  var nom :String
  var age : Int        // en nb de mois
  val race : String
  var poids : Double   // en kg

  init{
      this.nom = nom
      this.age = age
      this.race = race
      this.poids = if (poids > 0.0) poids else 0.1
  }
```

Ici l'attribut `race`, pour un chien donné ne doit plus pouvoir être changé : cela voudrait dire qu'on change la race d'un chien, ce qui est impossible

Visibilités en Kotlin

Par défaut, la visibilité des attributs et des méthodes est `public`

Attributs et méthodes peuvent être déclarés `private` afin de restreindre leur accès direct

- Certains attributs doivent être (en partie) cachés pour contrôler les modifications possibles de la classe : principes d'abstraction et d'encapsulation
- On peut être plus fin en interdisant uniquement l'accès en lecture ou en écriture
  - ajouter `private get` interdit la consultation de l'attribut
  - ajouter `private set` interdit la modification de l'attribut
- Certaines méthodes sont simplement utilitaires et ne doivent pas pouvoir être appelées depuis une autre classe

Attention

Les attributs peuvent toujours être consultés/modifiés via des méthodes.

Visibilités en Kotlin

Par défaut, la visibilité des attributs et des méthodes est `public`

Attributs et méthodes peuvent être déclarés `private` afin de restreindre leur accès direct

- Certains attributs doivent être (en partie) cachés pour contrôler les modifications possibles de la classe : principes d'abstraction et d'encapsulation
- On peut être plus fin en interdisant uniquement l'accès en lecture ou en écriture
  - ajouter `private get` interdit la consultation de l'attribut
  - ajouter `private set` interdit la modification de l'attribut
- Certaines méthodes sont simplement utilitaires et ne doivent pas pouvoir être appelées depuis une autre classe

Attention

Les attributs peuvent toujours être consultés/modifiés via des méthodes.

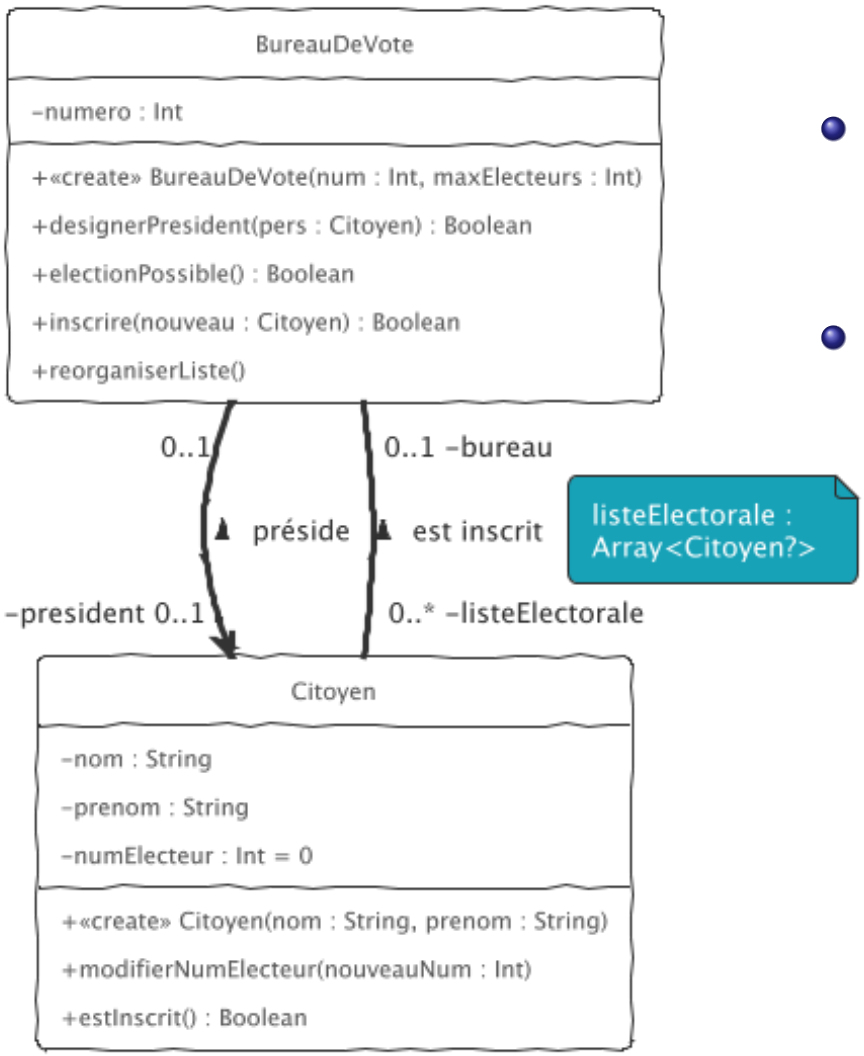
Visibilités en Kotlin : exemple

```
class Chien (...) {
  var nom :String
  private var age : Int
  val race : String
  var poids : Double
    private set

  fun courir(dist : Int) {
    poids -= poidsEnMoins(dist)
  }

  private fun poidsEnMoins(d : Int)
    = (d / 1000.0) / 1000
}
```

```
println("${rogue.nom}")
rogue.nom = "Severus"
println("${rogue.age}")      // error
  // it is private in 'Chien'
rogue.age = 10                // error
  // it is private in 'Chien'
println("${rogue.race}")
rogue.race = "Serpentard"     // error
  // val cannot be reassigned
println("${rogue.poids}")
rogue.poids = 42.0            // error
  // the setter is private in 'Chien'
rogue.courir(100)
rogue.poidsEnMoins(100)       // error
  // it is private in 'Chien'
```

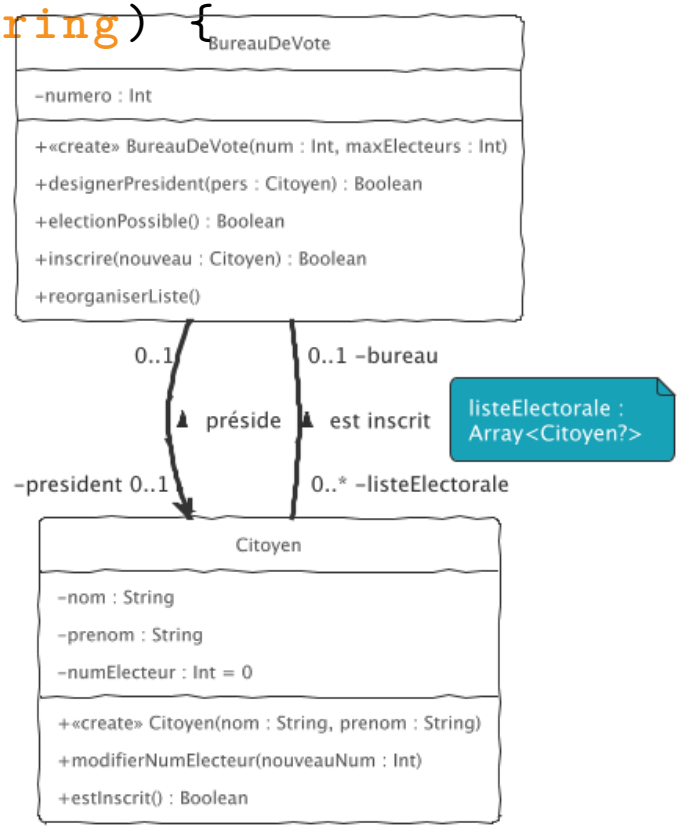
- L'attribut `nom` est `public` (par défaut) : accessible en lecture/écriture
- L'attribut `age` est `private` : aucun accès possible
- L'attribut `race` est `public`, mais immuable : accessible en lecture
- L'attribut `poids` est restreint en écriture : accessible en lecture
- La fonction `courir()` est `public` (par défaut) : accessible
- La fonction `poidsEnMoins()` est `private` : aucun accès possible

Exemple Citoyen : implémentation en Kotlin

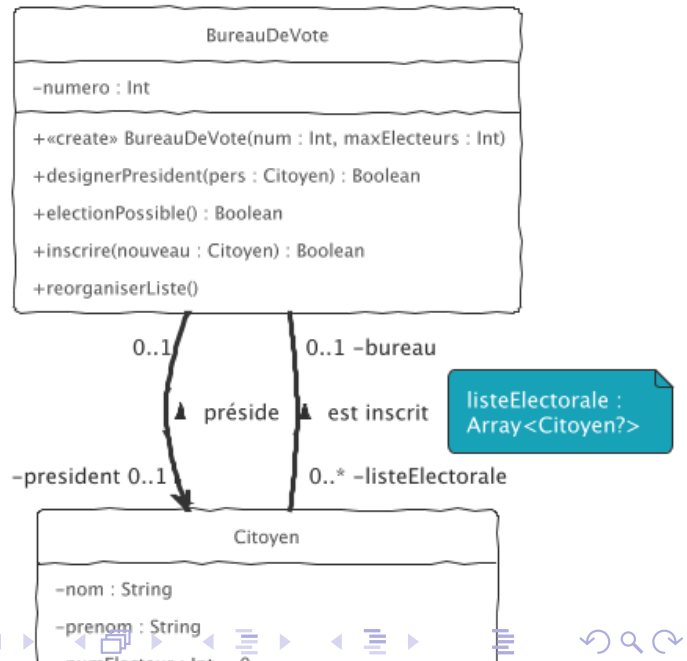


- On a déjà précisé beaucoup d'éléments :
  - types des attributs, méthodes, constructeurs, etc.
- **Comment implémenter les associations ?**
  - Une association unidirectionnelle devient un attribut dans la classe "source"
  - Une association bidirectionnelle devient deux attributs, un de chaque coté de l'association
  - Les rôles deviennent les noms des attributs à ajouter
  - (ajouter des méthodes pour mettre à jour les nouveaux attributs)

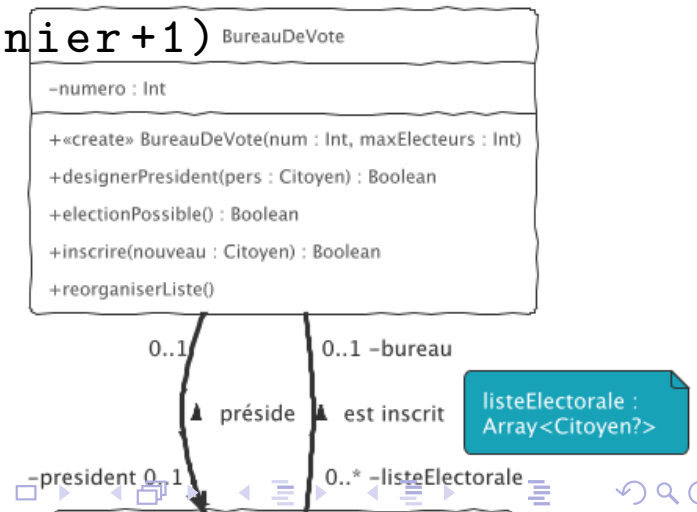
```
class Citoyen (nom : String, prenom : String) {

    private var nom : String
    private var prenom : String
    private var numElecteur : Int
    private var bureau : BureauDeVote?

    init {
        this.nom = nom
        this.prenom = prenom
        this.numElecteur = 0
        this.bureau = null
    }

    fun modifierNumElecteur(nouveauNum : Int) {
        numElecteur = nouveauNum
    }

    fun modifierBureauDeVote(nouveauBureau : BureauDeVote) {
        bureau = nouveauBureau
    }

    fun estInscrit() = (bureau != null)
```



```
class BureauDeVote (num : Int, maxElecteurs : Int) {

    private val numero : Int
    private var president : Citoyen?
    private val listeElectorale : Array<Citoyen?>

    init {
      numero   = num
      president = null
      listeElectorale = arrayOfNulls<Citoyen>(maxElecteurs)
    }

    fun designerPresident(pers : Citoyen) =
      if (pers.estInscrit()) {
        president = pers
        true
      }
      else false

    fun electionPossible() =
      (president != null
       && listeElectorale.isNotEmpty())
```



```
private fun dernierePosition() : Int {
    for (i in 0 until listeElectorale.size) {
        if (listeElectorale[i] == null)
        return i
    }
    return listeElectorale.size
}

fun inscrire(nouveau : Citoyen) : Boolean {
    if (nouveau.estInscrit())
        return false // nouveau deja inscrit
    val dernier = dernierePosition()
    if (listeElectorale.size == dernier)
        return false // listeElectorale pleine
    listeElectorale.set(dernier, nouveau)
    nouveau.modifierBureauDeVote(this)
    nouveau.modifierNumElecteur(dernier+1)
    return true
}

fun reorganiserListe() {
    // TODO
}
```



Résumé UML → Kotlin

UML tous les attributs doivent être typés

UML il doit y avoir un constructeur par classe

Kotlin les associations deviennent des attributs

- association bidirectionnelle : 2 attributs
- association unidirectionnelle : un attribut dans la classe source
- les rôles nomment les attributs

Kotlin une cardinalité `0..1` donne un type nullable : `X?`

Kotlin une cardinalité `0..*` ou `1..*` donne un type `Array<X?>`
(ou une autre collection)

Kotlin les visibilités doivent être respectées

Kotlin On respecte **scrupuleusement** tous les noms donnés, à la lettre près

Résumé UML → Kotlin

UML tous les attributs doivent être typés

UML il doit y avoir un constructeur par classe

Kotlin les associations deviennent des attributs

- association bidirectionnelle : 2 attributs
- association unidirectionnelle : un attribut dans la classe source
- les rôles nomment les attributs

Kotlin une cardinalité `0..1` donne un type nullable : `X?`

Kotlin une cardinalité `0..*` ou `1..*` donne un type `Array<X?>`
(ou une autre collection)

Kotlin les visibilités doivent être respectées

Kotlin On respecte **scrupuleusement** tous les noms donnés, à la lettre près

Aller plus loin

On reviendra plus tard sur

- des précisions sur les constructeurs
  - écriture simplifiée
  - constructeurs secondaires
- **Héritage**
  - classes abstraites et généralisation
  - interfaces et réalisation
- redéfinition du `toString()`
- redéfinition des `get()` / `set()`
- redéfinition des opérateurs `+`, ...
- les attributs / les méthodes de classes