Types Abstraits & Bases de la POO
IV - Bases de la Programmation Objet : UML

- - - - - - - - - -
1 – Introduction
2 – Principes de base des langages objet
3 – Le modèle UML : présentation
4 – Modèle UML : le diagramme de classes
- - - - - - - - - -

ENSMA A3-S5 - période A

2021-2022

M. Richard
richardm@ensma.fr

Sommaire

I - Introduction

↪ **La programmation que vous connaissez :**
On réalise ici une décomposition Top-down ; La fonction principale est décomposée en plusieurs sous fonctions, elles-même pouvant être à leur tour décomposées.

La décomposition en ensemble de fonctions pose souvent un certain nombre de problèmes :

- dans un système réel, il n'y a pas qu'une seule fonction importante ...
- réutilisabilité assez mauvaise
- extensibilité assez délicate ...
- très dépendante de la structure de données ...

↪ Définitions

La *programmation Orientée Objet (POO)* est un *paradigme* de programmation informatique consistant à centrer l'approche de modélisation (et d'implémentation par conséquence) sur les données et non sur les fonctionnalités (comme en programmation procédurale).

Ainsi, un programme objet peut-être vu comme :

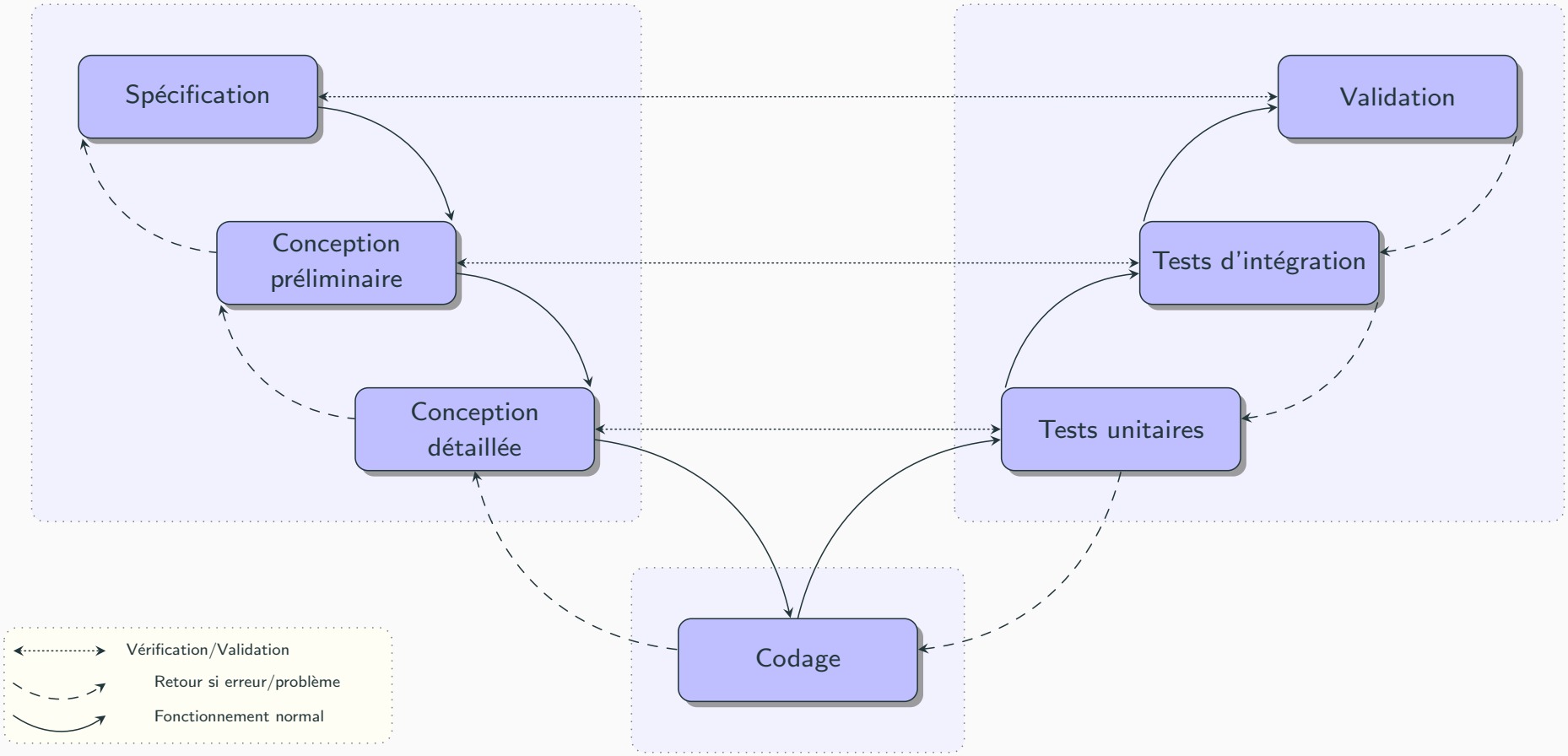
- un ensemble de briques, ou *objets*, chacune d'elles représentant un concept, et interagissant avec les autres, permettant ainsi de réaliser les fonctionnalités attendues.

La *conception orientée objet (CPO)* d'un logiciel est une méthode de conception fondant l'architecture de celui-ci sur des modules déduits des types des objets manipulés et non pas des fonctionnalités que le système doit assurer.

Le langage de modélisation le plus utilisé dans ce contexte est *UML (Unified Modeling Language)*

↪ CPO & Cycle de développement

Le paradigme objet se prête mieux à la mise en place et au respect du cycle en V :



↪ Paradigme objet VS procédural

- Paradigme objet :
  - Abstraction, encapsulation, réutilisation
  - Facilite la conception et la maintenance
  - Conception/réalisation "par composants"
  - Très adapté aux systèmes événementiels
  - Peu adapté dès lors que la complexité des données manipulées est faible

- Paradigme procédural :
  - Modules, variables, fonctions
  - Conception descendante
  - Chaque fonction résout une partie du problème
  - Peu adapté aux systèmes événementiels
  - Très adapté lorsque les fonctionnalités manipulent des données très simples (calcul, ...)

↪ Quel modèle pour un objet ?

Rappelez vous de la composition de la définition d'un type de données abstrait ; un TDA est défini par :

- un type, i.e. une structure de données
- un ensemble de méthodes
- un ensemble d'axiomes
- un ensemble de préconditions

↪ La classe : définition

On peut alors définir la notion de *classe* par :

> Une classe est un type de données abstrait (TDA) avec une implémentation *totale ou partielle*.

- l'implémentation est en générale toujours partielle . . .

> Un logiciel orienté objet est une collection structurée d'implémentation (partielle ou totale) de TDA, i.e. de classes.

II - Principes de base des langages objet

1 – Classe & objet
2 – Encapsulation
3 – Communication
4 – Instanciation

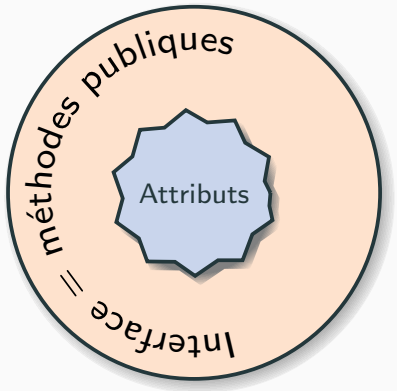
↪ Définitions

- *Une classe* est la modélisation statique d'une "partie des données"
  - ensemble d'*attributs* modélisant sa structure (i.e. structure de données)
  - ensemble de *méthodes* permettant d'agir sur les attributs
- *Un objet* est une entité identifiable manipulée par l'application ayant comme *modèle une classe*
  - on parle d'*instance d'une classe*
  - identité unique permettant de distinguer un objet d'un autre
- Un objet possède un *comportement* ; défini par les réactions aux différents *messages* qu'il reçoit de l'extérieur
  - dicté par un ensemble de fonctions et procédures portant le nom de *méthodes*
  - les réactions d'un objet sont dépendantes de l'*état* dans lequel il se trouve
- *l'état d'un objet* est défini par les valeurs de variables, portant le nom d'attributs, décrivant l'objet

> Un objet est l'instance d'une classe qui se définit par le regroupement d'un ensemble de données (attributs) et d'un ensemble d'algorithmes (méthodes)

↪ Définition

- l'accès aux attributs d'un objet ne doit se faire que via les méthodes proposées par l'objet
  - les attributs sont donc privés
- les méthodes déclarées publiques définissent l'interface de l'objet

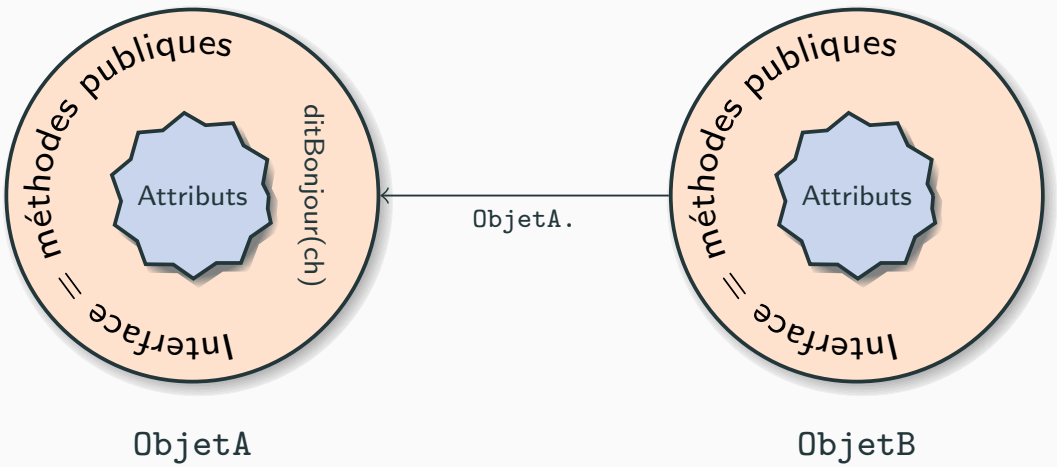


↪ Intérêts

- une modification structurelle interne à un objet n'a aucune conséquence sur les objets utilisant celui-ci
- modification de la valeur des attributs contrôlée par les accesseurs

↪ Définition

- communication entre objet par *message*
  - i.e. appel d'une méthode
- les méthodes d'une classe définissent l'ensemble des messages qu'un objet peut recevoir
- *syntaxe d'un message* :
  - récepteur du message (objet concerné)
  - nom du message (i.e. de la méthode)
  - paramètres nécessaires éventuels nécessaires au message
  - Ex : `ObjetA.DitBonjour(''aux options 3'');`



↪ Définition

- Un objet est le résultat de l'instanciation d'une classe
- On parle d'instance
- Attention, une instanciation est différente d'une déclaration . . .
- Utilise la notion de constructeur portant généralement le nom de la classe généralement
- Schéma d'un objet définissant totalement cet objet :
  - attributs,
  - méthodes,
  - ... etc.
- Exemple :

Figure 1 – Instanciation

III - Le modèle UML : présentation

1 – Constat
2 – Pourquoi un modèle
3 – Points de vue
4 – Représentation graphique
5 – Éléments de modélisation communs

↪ Besoin de modéliser . . .

Un logiciel peut avoir une durée de vie très longue :

- fonctionnalités devenant plus nombreuses et complexes,
- supports matériel et architectures d'exécution qui évoluent, . . .

La lecture du code source dans un langage de programmation ne suffit plus à contenir toutes les informations nécessaires à la bonne compréhension de la façon dont il est construit

Nécessaire d'utiliser des modèles intermédiaires de plus haut niveau afin :

- de le comprendre
- de le valider

Modèles intermédiaires sont utiles dans les différentes phases du cycle en V :

- analyse
- conception
- implémentation

↪ Besoin de modèle...

Ces modèles permettent :

- de représenter de façon non ambiguë le problème à des niveaux d'abstraction de plus en plus raffinés
- d'échanger au sein des équipes de développement
- de refléter/représenter tous les choix de conception à tout niveau
- la traçabilité des informations concernant la réalisation de l'application

↪ Modèle VS méthode...

- il ne faut pas confondre méthode et modèle
- un modèle est un langage permettant d'exprimer un certain nombre de propriétés
- des méthodologies adaptées sont développées pour utiliser au mieux un modèle

UML organise son modèle de spécification autour de plusieurs points de vue permettant de comprendre le système et de le décrire dans tous ses aspects.
On distingue :

- les aspects "statiques", qui permettent de décrire :
  - le point de vue fonctionnel : que fait le système, quelles sont ses fonctions ?
  - le point de vue structurel des objets du domaine (aspect ≪ structure de données ≫) visé par le système au travers du modèle de classes : quels sont les objets manipulés par le système, comment sont-ils organisés, quels sont leurs relations mutuelles ?
- des aspects "dynamiques" du système visant :
  - le point de vue du comportement externe du système (comment se passent les interactions entre les différents utilisateurs et les objets du système, de façon à couvrir l'exécution des fonctions du système) : comment se sert-on du système ?
  - le point de vue de la dynamique du système et des objets qui le composent, permettant de décrire l'activité du système au cours du temps, dans ses aspects ≪ contrôle ≫ et son évolution en cours d'exécution sur l'apparition d'évènements externes ou internes : comment réagit le système ?

↪ Les différents diagrammes

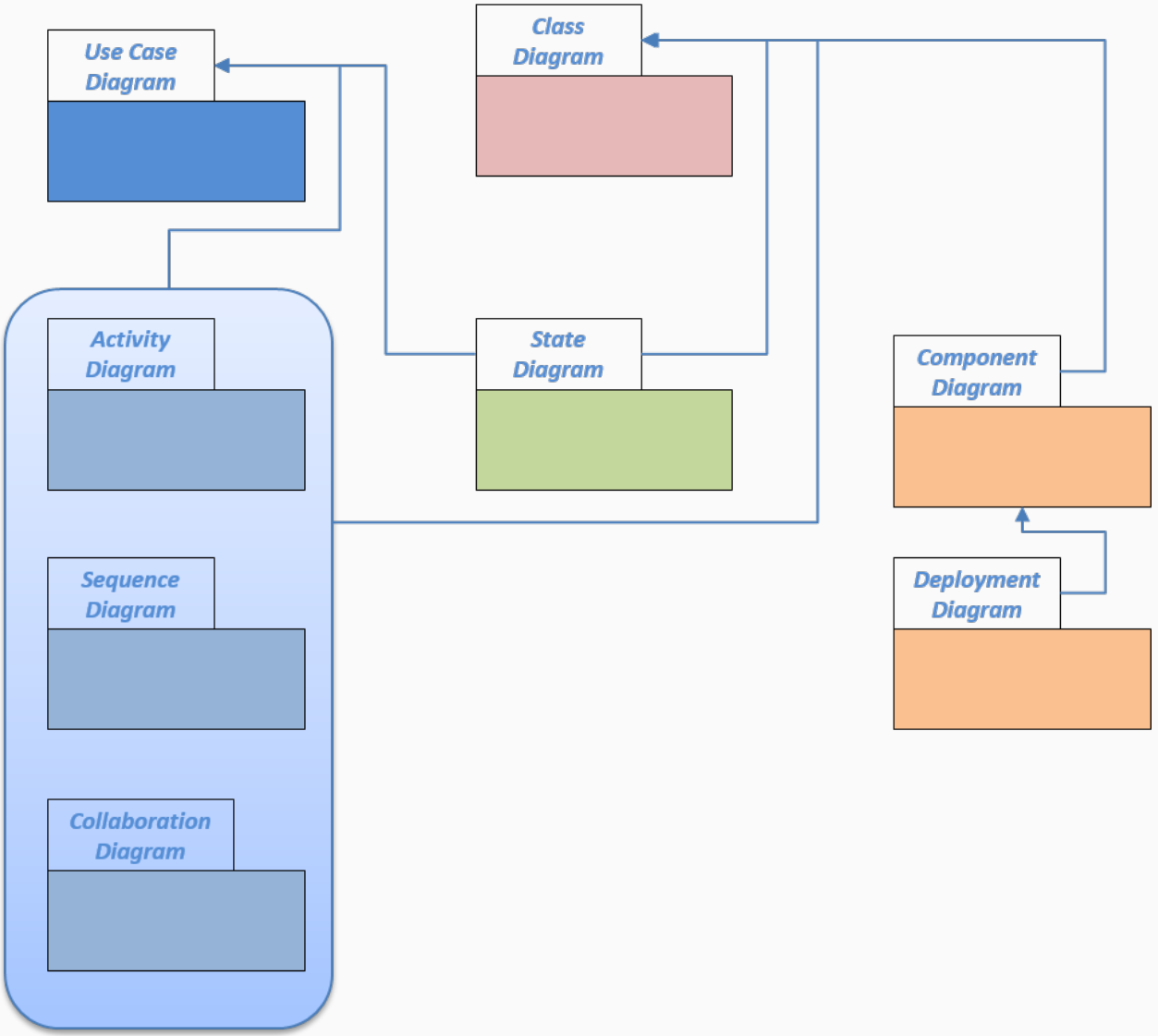
UML est un langage à représentation graphique. Une conception UML est donc constituée d'un ensemble de diagrammes :

- pouvant être nombreux
- liés entre eux
- intégrant des éléments formels (i.e. informatiquement interprétables)
- intégrant des éléments informels (notes, commentaires, ...)

Les principaux diagrammes sont :

- Use Case Diagram : modélise les objectifs du système
- Class Diagram : modèle structurel de classe
- State Diagram : modèle dynamique (états/transitions)
- Activity Diagram, Sequence Diagram et Collaboration Diagram : modélisent les interactions du système
- Component Diagram : modélise les composants logiciel du système (Architecture)
- Deployment Diagram : modélise l'architecture physique du système.
- ...

↪ Relations entre les différents diagrammes

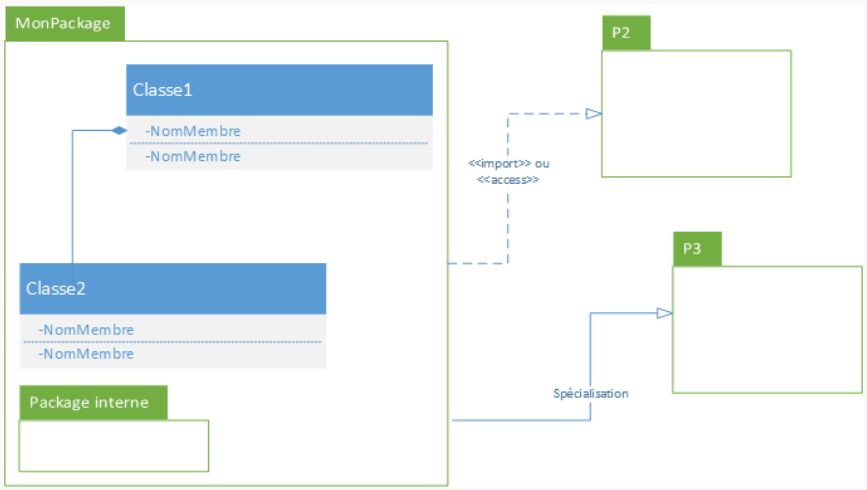


UML utilise de nombreux éléments de modélisation pour décrire les différents points de vue dans le développement logiciel. On distingue :

- les éléments de modélisation propres à chaque type de diagramme
- les éléments de description communs que l'on peut retrouver dans tous

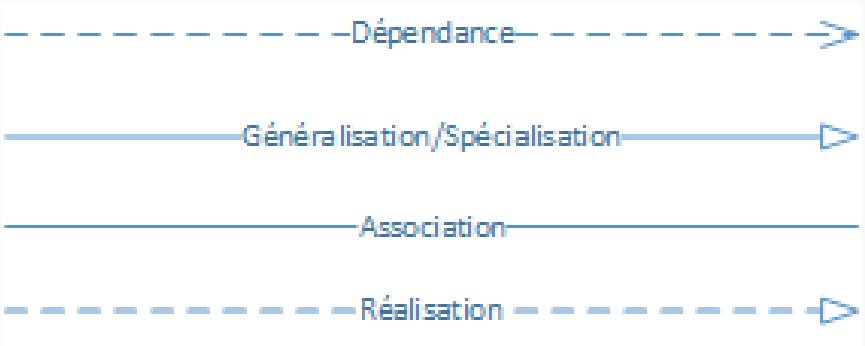
Ces éléments de description communs peuvent être catégorisés :

- organisation modulaire de l'architecture : le package qui constitue un moyen d'organisation de la modélisation
  - Un package définit un espace de noms,
  - un package peut contenir d'autres packages,
  - il peut ≪ *importer* ≫ les éléments déclarés "public" d'un autre package ou simplement y faire référence (≪ *access* ≫),
  - un package peut spécialiser un autre package

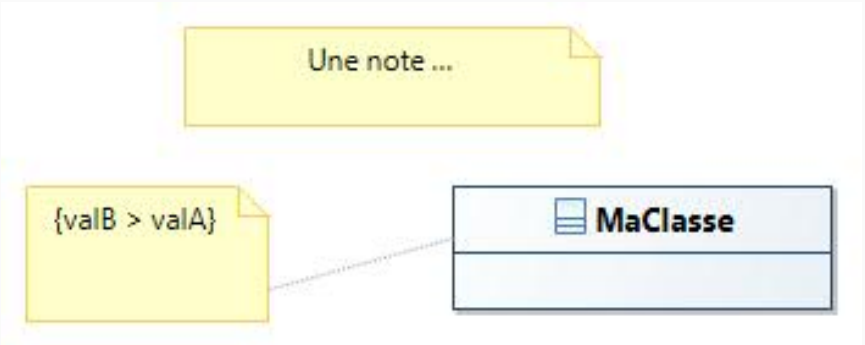


↪ Les relations

- Relations entre éléments de modélisation :
  - Ces relations ont une représentation générique : leur sens précis dépend des annotations qui l'accompagnent et des éléments de modélisation en jeu.

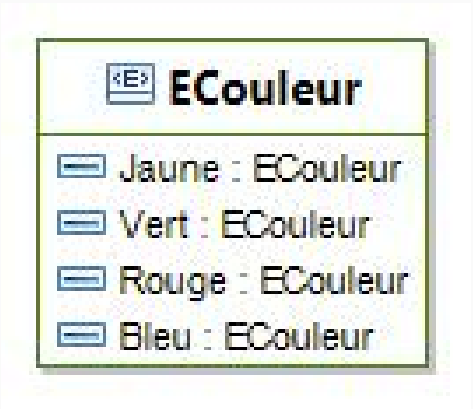


- Notes et Contraintes sur un élément de modélisation
  - Permet de documenter précisément les éléments d'un modèle.



↪ Extension

- Mécanisme d'extension du méta-modèle :
  - Stéréotypes : Annotations qui représentent une spécialisation d'un concept du méta-modèle indiquant une sémantique particulière par rapport à la sémantique standard ; par exemple les énumérations



  - Tagged Values : Extension de propriétés affectées à un concept du méta-modèle
  - Contraintes et langage de contrainte : OCL : Permet d'exprimer formellement les contraintes

IV - Modèle UML : le diagramme de classes

Objectifs :

- Modéliser le domaine d'un système, c'est-à-dire les classes (et leurs relations mutuelles) manipulées par le système : vue *statique* du système.

Concepts utilisés :

- les packages
- les classes, énumérations, interfaces, ...
- les associations

Etapes de construction :

1. Identifier les classes,
2. Identifier les attributs des classes et les liens entre ces classes,
3. En déduire les classes et les associations,
4. Identifier les fonctions du système et les traduire en opérations de classe,
5. Factoriser, simplifier les classes,
6. Vérifier, itérer et affiner ...
7. Grouper les éléments en sous-systèmes

↪ Diagramme de classe VS Objet

Diagramme de classes :

- Représentation de la structure interne du logiciel
- Utilisé surtout en conception mais peut être utilisé en analyse

Diagramme d'objets :

- Représentation de l'état du logiciel (objets + relations)
- Diagramme évoluant avec l'exécution du logiciel
  - création et suppression d'objets
  - modification de l'état des objets (valeurs des attributs)
  - modification des relations entre objets

Définition :

- Classe : Regroupement d'objets de même nature (mêmes attributs + mêmes opérations)

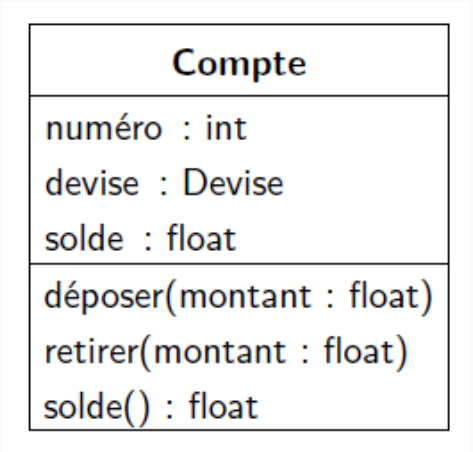
Le concept central de classe est définie par :

- la structure (la liste des attributs) d'une classe
- les opérations applicables aux instances de cette classe
  - Attention : une opération est implémentée par une ou plusieurs méthodes

Attributs :

- Propriété partagée par tous les objets de la classe
- Associe à chaque objet une valeur
- Type associé simple (int, bool...), primitif (Date) ou énuméré

Représentation graphique :

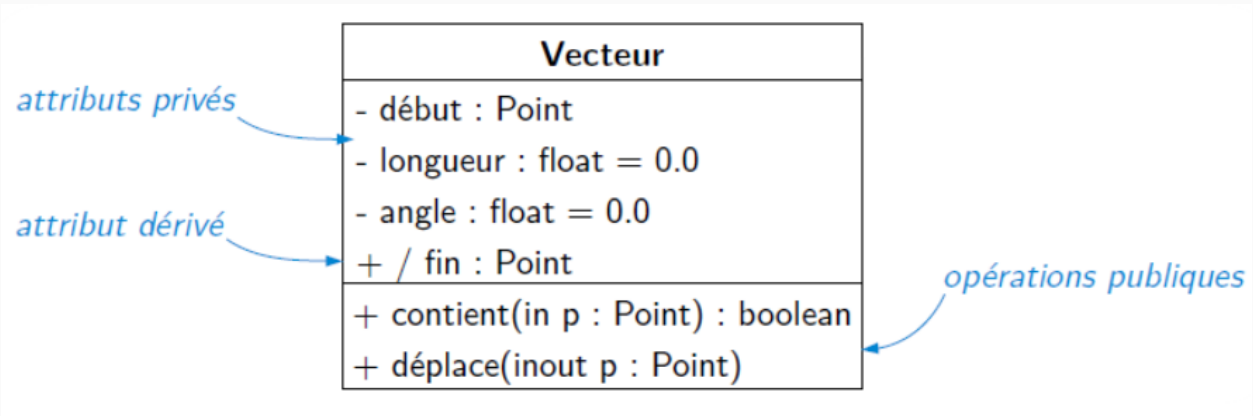


Notations avancées : Attributs

- `[visibilité] [dérivé] nomAttribut [: type] [= valeur par défaut]`
- Visibilité :
  - + : public, accessible à toutes les classes
  - # : protégé, accessible uniquement aux sous-classes
  - - : privé, inaccessible à tout objet hors de la classe
  - Remarque : il n'existe pas de visibilité par défaut
- Dérivé :
  - / : un attribut dérivée peut-être calculé à tout moment à partir des autres attributs de la classe

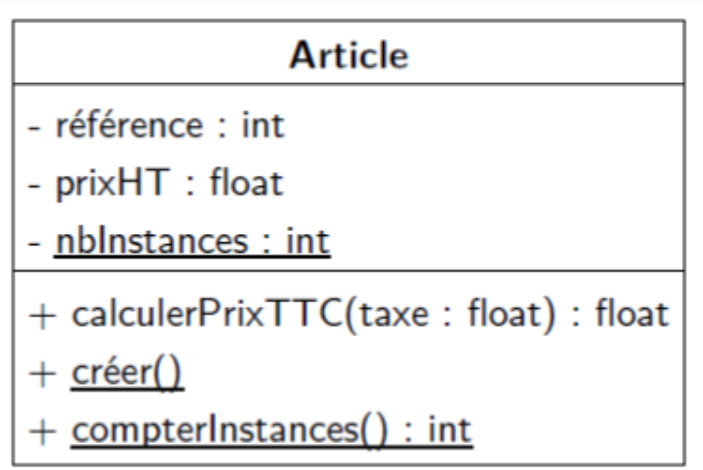
Notation avancées : Opérations

- `[visibilité] nomOpération [(liste Paramètres)] [: typeRetour]`
- `Paramètre ::= [direction] nom : type [= valeur par défaut]`
- `Direction ::= in | out | inout (par défaut : in)`



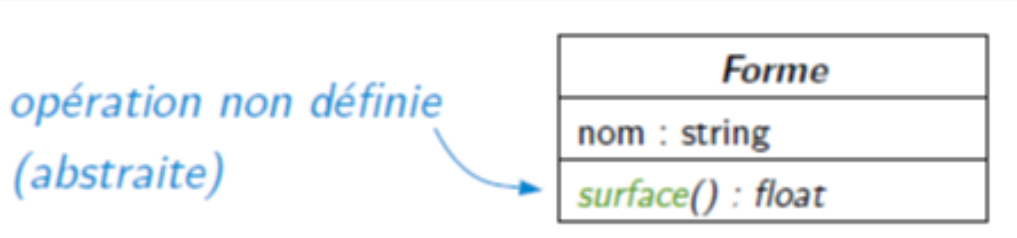
Notations avancées : Attributs et opérations de classe

- Attributs de classe : valeurs communes à toutes les instances
- Opérations de classe : opérations sur la classe elle-même
- Soulignés dans la classe



Classe abstraite :

- Classe sans instance
- certaines opérations ne peuvent être définies à ce niveau
- Opération non définie en italique
- Nom de la classe en italique (ou stéréotype ≪ abstract ≫)



Définition : Association

- une association entre classe est en général une relation binaire

Définition : Rôle

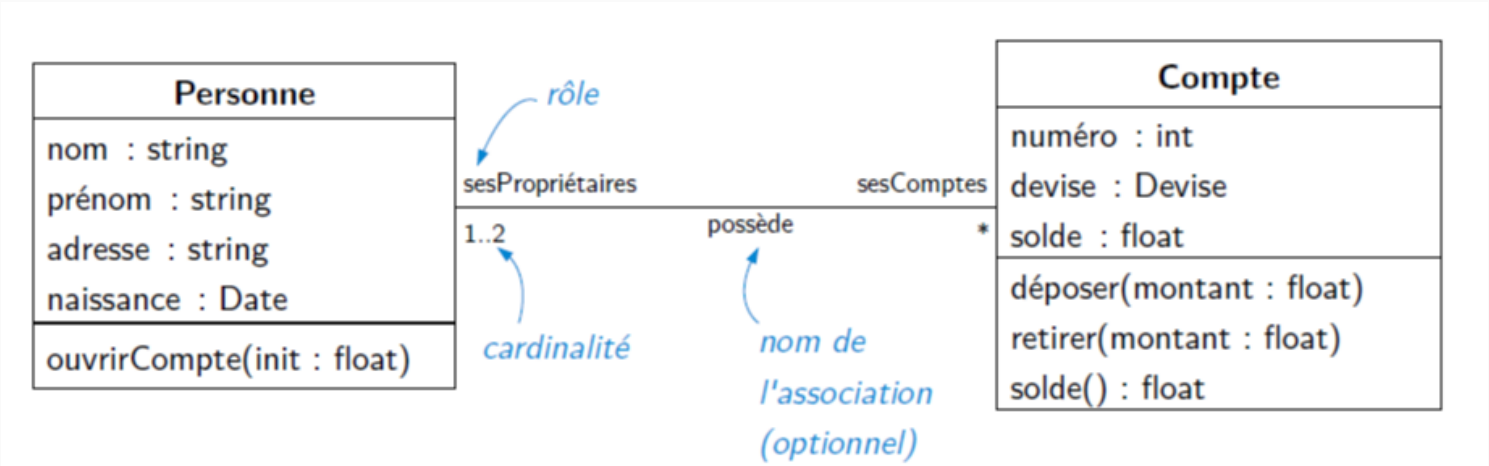
- Nomme l'extrémité d'une association,
- permet d'accéder aux objets liés par l'association à un objet donné

Définition : Cardinalité

- Contraint le nombre d'objets liés par l'association

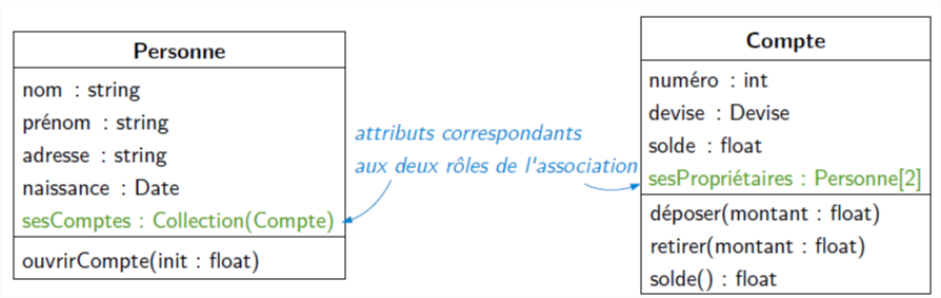
Utilisation :

- l'utilisation d'une association définit donc deux attributs implicites :
  - explicitée comme attributs en phase de conception détaillée,
  - qui porteront les noms des rôles de l'association



Exemple d'un diagramme en conception

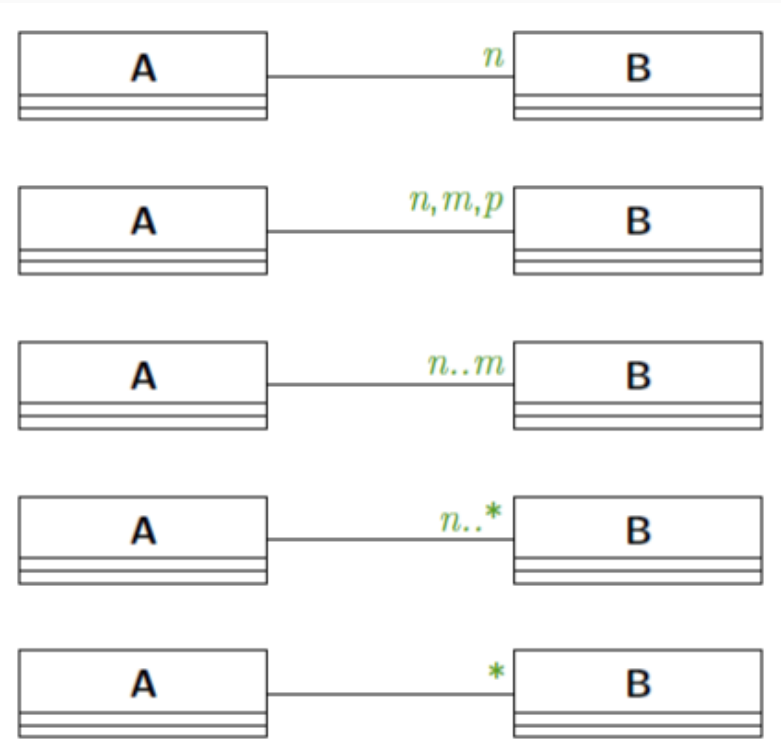
Exemple d'un diagramme en conception détaillée



Exemple d'un diagramme faux

Cardinalités :

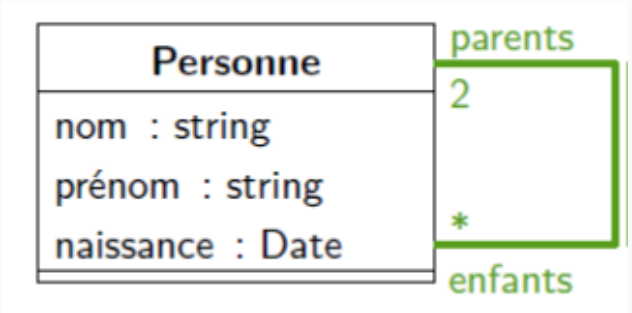
- permet de définir le nombre d'instances de classe B associées à une classe A
  - n : exactement n
  - n, m, p : n ou m ou p
  - $n..m$: entre n et m
  - $n..*$: au moins n
  - $*$: 0 ou plus



Définition :

- Permet de définir une relation d'une classe vers elle-même

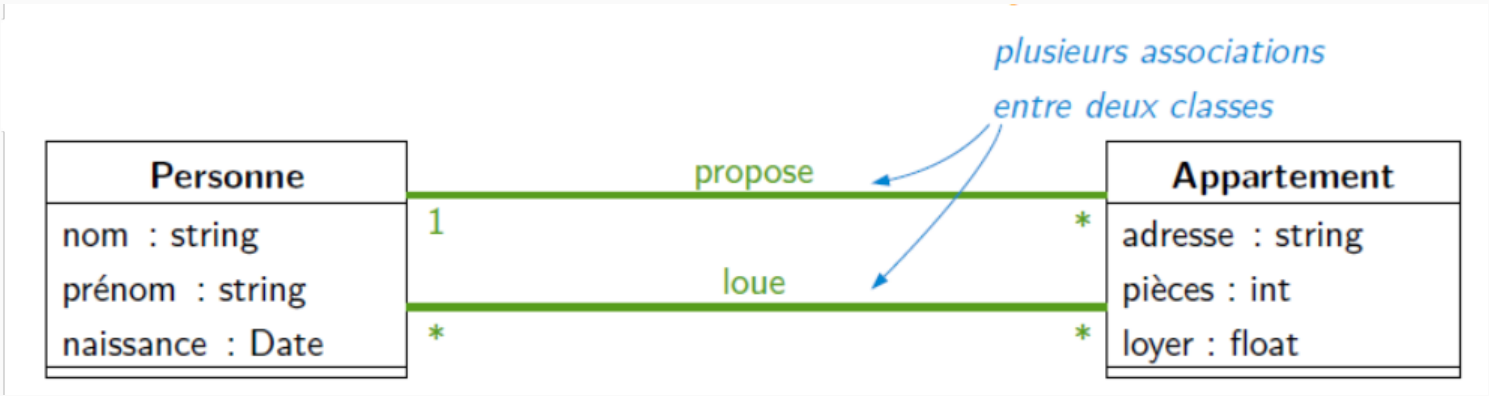
Exemple :



Définition :

- permet de définir plusieurs associations différentes entre deux classes

Exemple :



Définition :

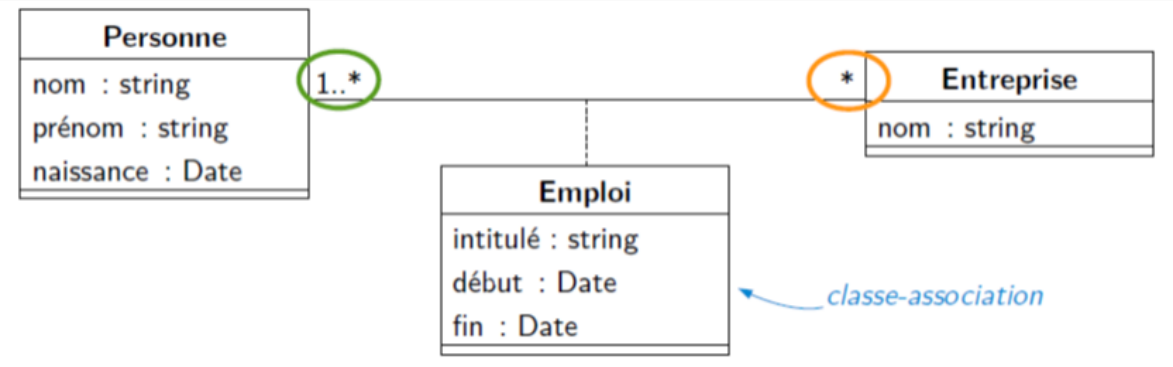
- permet de définir l'orientation d'une association
  - l'objectif est de restreindre l'accessibilité des instances mises en œuvre dans une association
- par défaut, i.e. sans flèche, l'association est navigable dans les deux sens

Exemple :

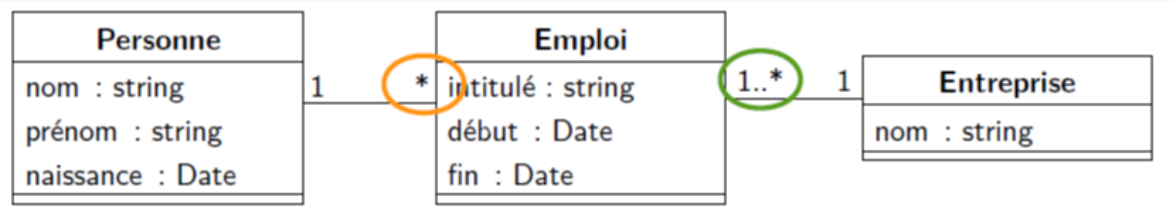
Définition :

- Permet de paramétrer une association à l'aide d'une autre classe

Exemple :



Équivalence :

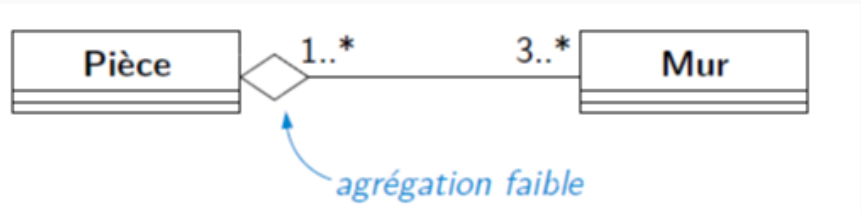


Définition :

- Association particulière entre classes
  - Dissymétrique : une classe prédominante sur l'autre
  - Relation de type composant-composite
- Deux types d'agrégation :
  - Agrégation faible
  - Composition

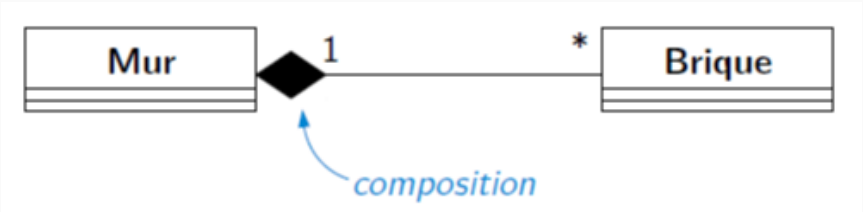
Agrégation faible :

- porte aussi le nom d'agrégation par référence
- le composite fait référence à ses composants
- La création ou destruction du composite est indépendante de la création ou destruction de ses composants
- Un objet peut faire partie de plusieurs composites à la fois
- Exemple :
  - Une pièce est composée de murs
  - Un mur peut être commun à plusieurs pièces



Composition

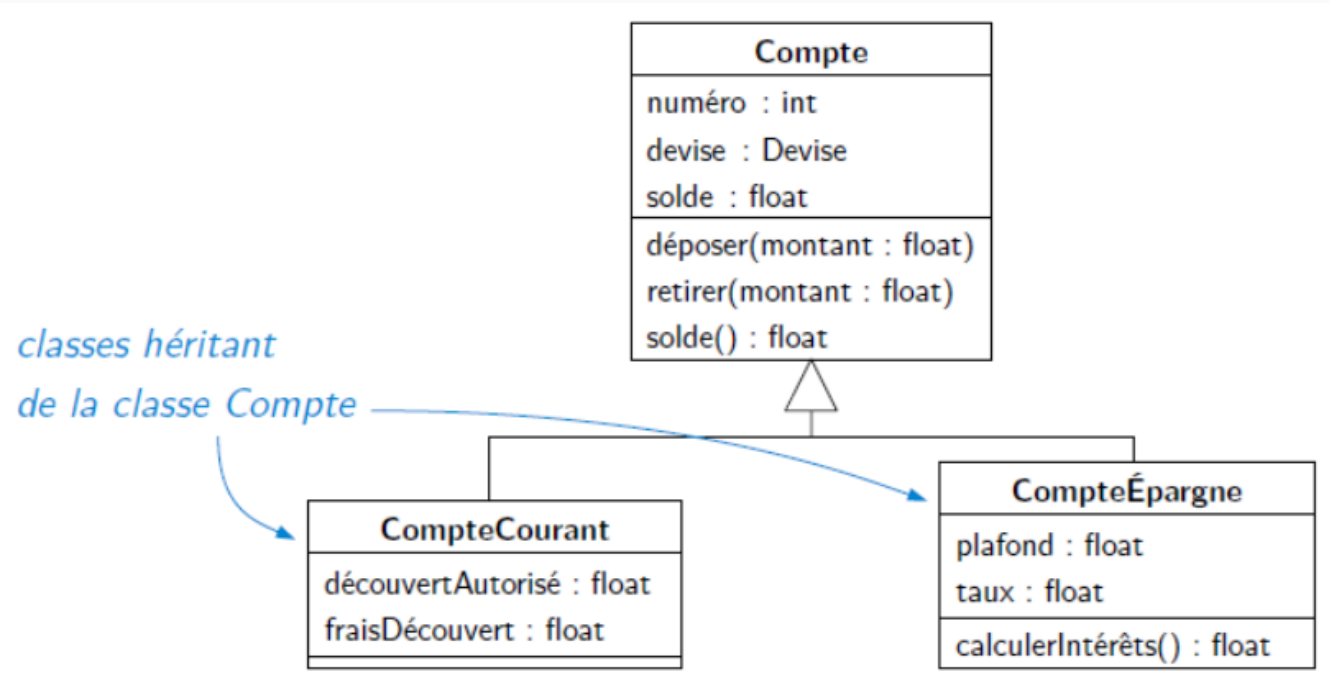
- porte aussi le nom d'agrégation par valeur
- le composite contient ses composants
- la création ou destruction du composite entraîne la création ou destruction de ses composants
- un objet ne fait partie que d'un composite à la fois
- Exemple :
  - Un mur est composé de briques
  - Une brique n'appartient qu'à un mur



Définition :

- Construction d'une classe à partir d'une classe plus haute dans la hiérarchie
  - partage des attributs, opérations, contraintes ...

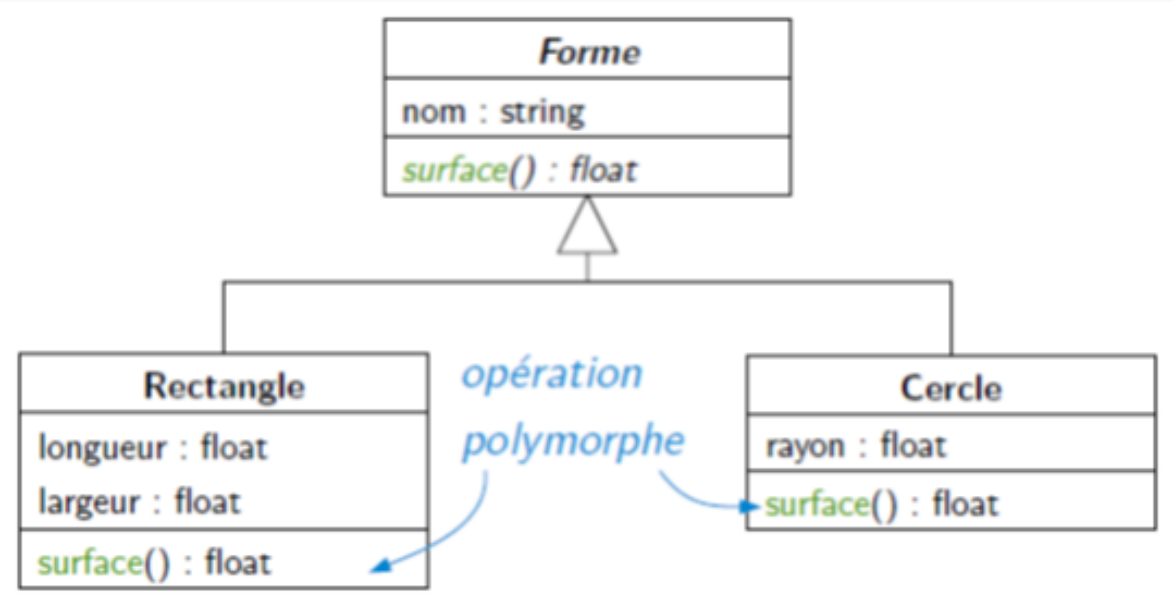
Exemple :



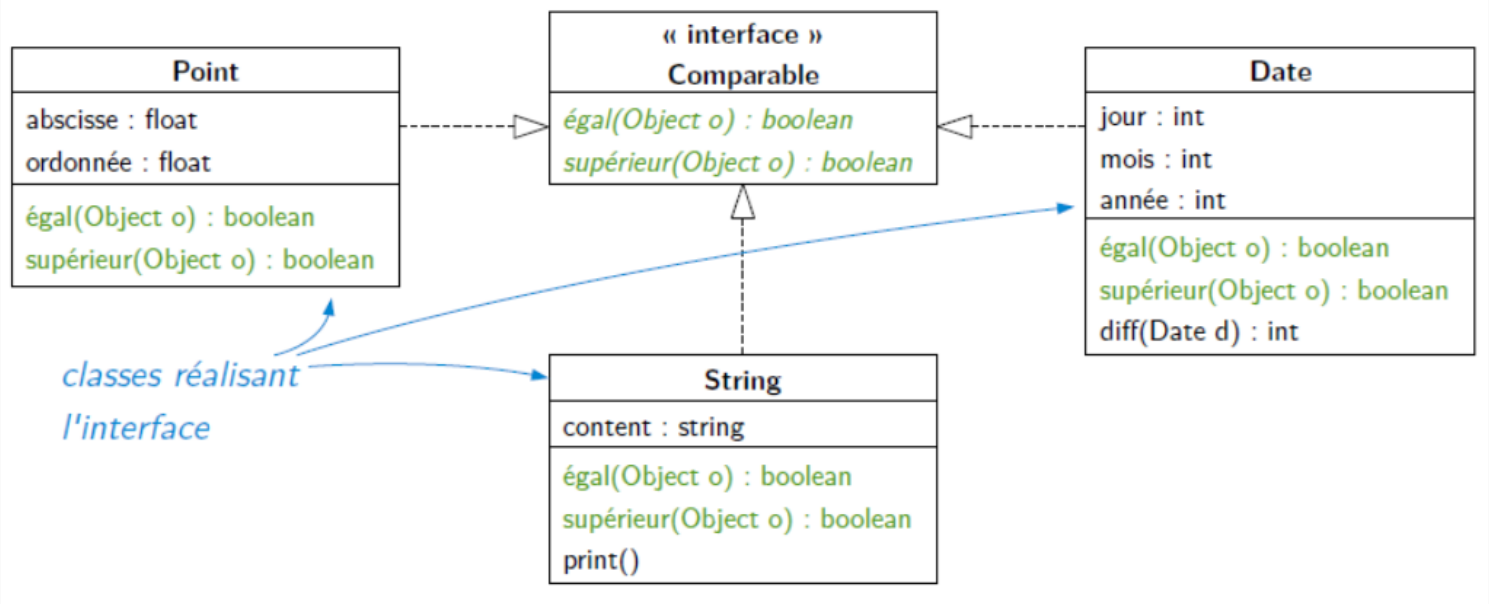
Définition :

- Opération définie dans différentes sous-classes mais opération spécifique à la sous-classe

Exemple :



Exemple :


« interface »
Comparable
égal(Object o) : boolean
supérieur(Object o) : boolean
Point
abscisse : float
ordonnée : float
égal(Object o) : boolean
supérieur(Object o) : boolean
Date
jour : int
mois : int
année : int
égal(Object o) : boolean
supérieur(Object o) : boolean
diff(Date d) : int
String
content : string
égal(Object o) : boolean
supérieur(Object o) : boolean
print()
classes réalisant
l'interface