1 Introduction aux diagrammes de classes UML

1.1 Définition et objectif des diagrammes de classes

Definition 1.1. Les diagrammes de classes UML (Unified Modeling Language) sont des représentations graphiques de la structure statique d'un système. Ils décrivent les types d'objets présents dans le système, leurs attributs, leurs opérations et les relations qui existent entre eux.

L'objectif principal d'un diagramme de classes est de modéliser la structure interne d'un logiciel. Ils sont largement utilisés dans les phases de conception et d'analyse du développement logiciel pour visualiser et comprendre l'architecture du système.

1.2 Rôle dans le processus de développement logiciel

Dans un processus de développement logiciel en V, les diagrammes de classes jouent un rôle crucial à différents niveaux :

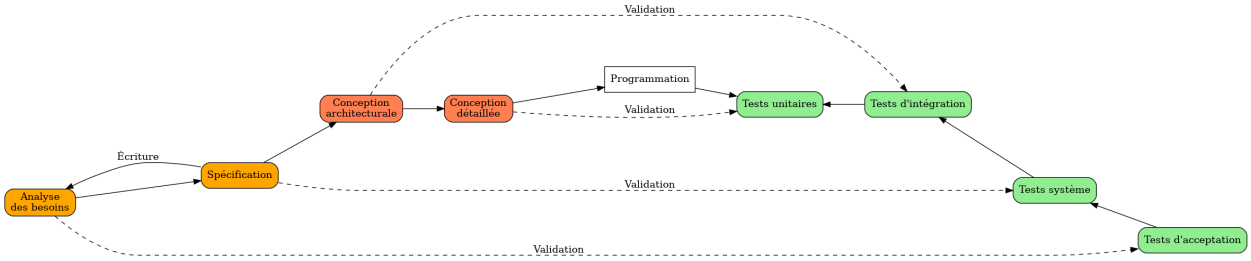

Figure 1: Processus de développement en V

Au début du processus, pendant la phase d'**analyse**, les diagrammes de classes permettent de modéliser le domaine du problème et d'identifier les concepts clés. Ils aident à comprendre *"ce que doit faire le système ?"* en représentant les entités du monde réel et leurs relations.

Durant la phase de **conception**, les diagrammes de classes sont utilisés pour définir l'architecture du logiciel et *"comment va-t-il le réaliser ?"* Ils précisent la structure interne du système, les classes de logiciels, leurs attributs, leurs méthodes et les relations entre ces classes.

Tout au long du cycle de développement, les diagrammes de classes servent de support à la communication entre les différents acteurs (analystes, concepteurs, développeurs) et de référence pour la documentation du système. Ils sont essentiels pour la **validation et la vérification**, s'assurant que *"fait-il ce qui était demandé et le fait-il correctement ?"* en confrontant le modèle aux besoins initiaux.

1.3 Diagrammes de classes vs diagrammes d'objets

Remark 1.2. Il est important de distinguer les diagrammes de classes des diagrammes d'objets.

- **Diagramme de classes** : Représente le **schéma**, le plan de construction. Il décrit la structure statique du logiciel, c'est-à-dire les classes, leurs attributs et opérations, et les relations entre ces classes. Le diagramme de classes est **abstrait** et général.
- **Diagramme d'objets** : Représente une **instance** du schéma à un moment donné de l'exécution du logiciel. Il montre des objets concrets, leurs états (valeurs des attributs) et les liens entre ces objets. Le diagramme d'objets est **concret** et spécifique à un instant T. Il évolue avec l'exécution du logiciel.

Par exemple, un diagramme de classes peut définir la classe `Compte` avec les attributs `numéro`, `devise`, `solde` et les opérations `déposer()`, `retirer()`, `solde()`. Un diagramme d'objets, lui, montrera des instances de `Compte` comme `MonLivretA : Compte` avec `numéro = 123456`, `devise = EUR`, `solde = 3509,43`.

2 Concepts fondamentaux

2.1 Classes

2.1.1 Définition d'une classe

Definition 2.1. Une **classe** est un regroupement d'objets de même nature, c'est-à-dire qui partagent les mêmes **attributs** et les mêmes **opérations**. Elle sert de **modèle** ou de **type** pour créer des objets. On peut voir la classe comme un moule et les objets comme les instances créées à partir de ce moule.

2.1.2 Composants d'une classe : Attributs et Opérations

Une classe est définie par deux types de composants :

- **Attributs** : Ce sont les caractéristiques ou propriétés d'une classe. Un attribut décrit une information que chaque objet de la classe possède. Chaque attribut a un nom et un type. Pour chaque objet de la classe, un attribut a une **valeur**.
- **Opérations** : Ce sont les actions ou services que les objets d'une classe peuvent réaliser ou que l'on peut leur demander de réaliser. Une opération est une fonction ou méthode applicable aux objets de la classe.

Dans un diagramme de classes, une classe est représentée par un rectangle divisé en trois parties (compartiments) :

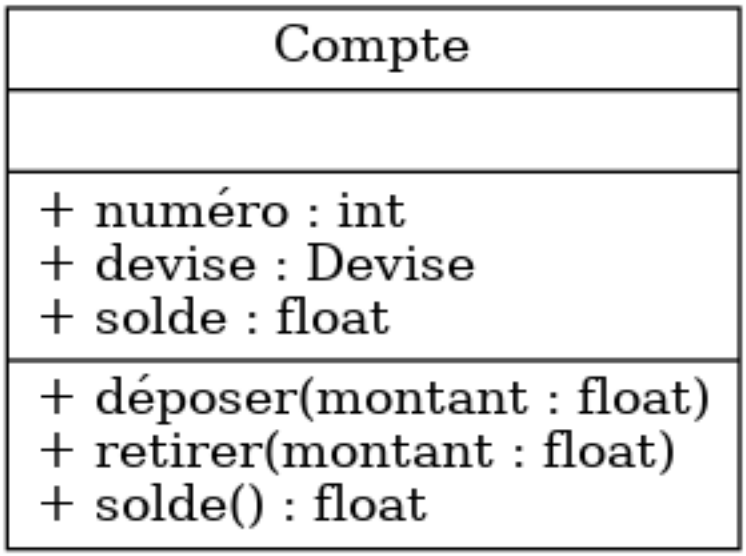

Figure 2: Représentation graphique de la classe `Compte`

- Le compartiment supérieur contient le **nom de la classe** (ici, `Compte`).
- Le compartiment du milieu liste les **attributs** avec leur type (par exemple, `numéro : int`, `devise : Devise`, `solde : float`).
- Le compartiment inférieur liste les **opérations** (ou méthodes) avec leur signature (par exemple, `déposer(montant : float)`, `retirer(montant : float)`, `solde() : float`).

2.1.3 Type des attributs

Le type d'un attribut définit le genre de valeurs qu'il peut prendre. Les types d'attributs peuvent être :

- **Types simples** : Types de données de base comme `int` (entier), `float` (nombre à virgule flottante), `boolean` (booléen), `string` (chaîne de caractères).
- **Types primitifs** : Types prédéfinis plus complexes, comme `Date`.
- **Types énumérés** : Types définissant un ensemble fini de valeurs nommées. Par exemple, un type énuméré `Devise` pourrait avoir comme valeurs : `EUR`, `GBP`, `USD`, `CHF`.

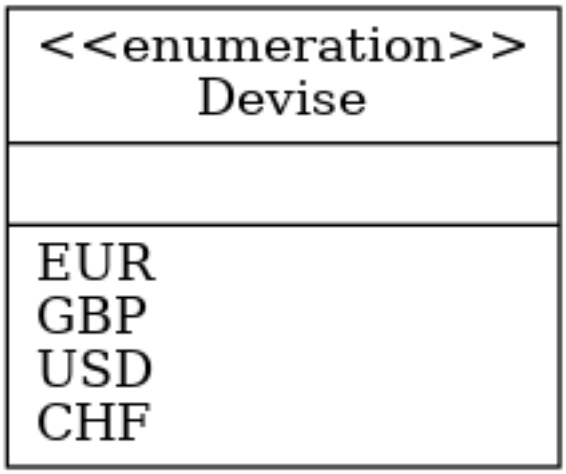

Figure 3: Représentation graphique de l'énumération `Devise`

Remark 2.2. Il est important de noter qu'un type énuméré n'est **pas une classe**. C'est un ensemble de valeurs possibles pour un attribut.

2.1.4 Classe vs Objet (Instance)

La classe est le concept, l'objet est la réalisation concrète de ce concept. Un **objet** est une **instance** d'une classe. Cela signifie qu'un objet est créé à partir du modèle défini par la classe. Pour une classe donnée, on peut créer de nombreux objets (instances). Chaque objet aura sa propre identité et son propre état (valeurs de ses attributs), mais tous partageront le même comportement (mêmes opérations définies dans la classe).

2.2 Objets

2.2.1 Définition d'un objet

Definition 2.3. Un **objet** est une entité concrète ou abstraite du domaine d'application que l'on cherche à modéliser. Il représente une occurrence particulière d'une classe.

2.2.2 Identité, état et comportement d'un objet

Chaque objet est caractérisé par :

- **Identité** : Chaque objet possède une identité unique qui le distingue des autres objets, même s'ils ont le même état. En termes techniques, l'identité peut être vue comme son adresse mémoire.

- **État** : L'état d'un objet est défini par les valeurs de ses attributs à un moment donné. L'état peut changer au cours du temps.
- **Comportement** : Le comportement d'un objet est déterminé par les opérations (méthodes) qu'il peut effectuer ou que l'on peut lui demander d'effectuer. Le comportement est défini par la classe de l'objet.

Dans un diagramme d'objets, un objet est représenté par un rectangle, similaire à la classe, mais avec les spécifications suivantes :

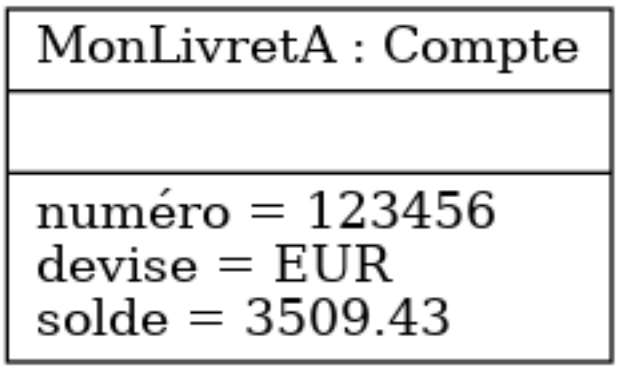

Figure 4: Représentation graphique de l'objet `MonLivretA`

- Le compartiment supérieur contient le **nom de l'objet** suivi de deux points et du **nom de sa classe** (ici, `MonLivretA : Compte`).
- Le compartiment du milieu liste les **attributs** et leur **valeur** pour cet objet spécifique (par exemple, `numéro = 123456`, `devise = EUR`, `solde = 3509,43`). Les opérations ne sont généralement pas représentées dans les diagrammes d'objets, car elles sont définies par la classe.

Remark 2.4. Des objets différents (ayant des identités différentes) peuvent avoir le même état (mêmes valeurs d'attributs). Par exemple, on peut avoir deux objets `Jean1 : Personne` et `Jean2 : Personne` avec les mêmes valeurs pour les attributs `nom`, `prénom` et `naissance`, mais ils restent deux objets distincts en mémoire.

3 Relations entre classes

Les relations entre classes permettent de modéliser les liens et les interactions entre les différents types d'objets d'un système. La relation la plus fondamentale est l'**association**.

3.1 Associations

3.1.1 Définition d'une association binaire

Definition 3.1. Une **association** binaire est une relation entre deux classes. Elle représente un lien sémantique entre les instances de ces classes. Une association permet de naviguer d'un objet d'une classe vers les objets associés d'une autre classe.

3.1.2 Multiplicité des associations

La **multiplicité** d'une association indique le nombre d'objets d'une classe qui peuvent être associés à un objet de l'autre classe. La multiplicité est spécifiée à chaque extrémité de l'association. Les multiplicités courantes sont :

- **1** : Exactement un.
- : Zéro ou plusieurs (plusieurs).
- **0..1** : Zéro ou un (au plus un).
- **1..** : Un ou plusieurs (au moins un).
- **n..m** : Entre n et m (n et m étant des entiers).

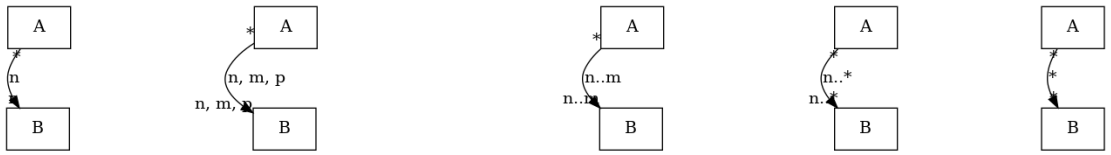

Figure 5: Exemples de multiplicités

Remark 3.2. En pratique, les multiplicités les plus fréquemment utilisées sont :

- **1** : Exactement 1.
- **0..1** : Au plus 1 (zéro ou un).
- **1..** : Au moins 1 (un ou plusieurs).
- : Zéro ou plusieurs (plusieurs).

3.1.3 Rôles des associations

Chaque extrémité d'une association peut avoir un **rôle**. Le rôle est un nom qui qualifie la façon dont une classe participe à l'association. Le rôle est utile pour préciser le sens de la navigation et pour accéder aux objets liés par l'association.

Par exemple, dans une association entre `Personne` et `Compte`, on peut nommer les rôles `sesPropriétaires` pour l'extrémité côté `Personne` et `sesComptes` pour l'extrémité côté `Compte`.

3.1.4 Nom des associations

Une association peut avoir un **nom**, qui est un verbe ou une courte phrase décrivant la nature de la relation. Le nom de l'association est optionnel. Par exemple, l'association entre `Personne` et `Compte` pourrait être nommée `possède`.

3.1.5 Navigabilité des associations

La **navigabilité** d'une association indique la direction dans laquelle on peut naviguer entre les objets associés. Par défaut, les associations sont **navigables dans les deux sens** (bidirectionnelles). On peut restreindre la navigabilité à un seul sens en ajoutant une flèche à l'extrémité navigable de l'association. Une association navigable de A vers B signifie que depuis un objet de la classe A, on peut accéder aux objets associés de la classe B, mais pas l'inverse.

3.2 Associations particulières

3.2.1 Association réflexive

Example 3.3. Une **association réflexive** est une association où la même classe participe aux deux extrémités de la relation. Cela permet de modéliser des relations entre les instances d'une même classe.

Par exemple, dans la classe `Personne`, on peut définir une association réflexive `parenté` pour représenter les relations parents-enfants entre personnes.

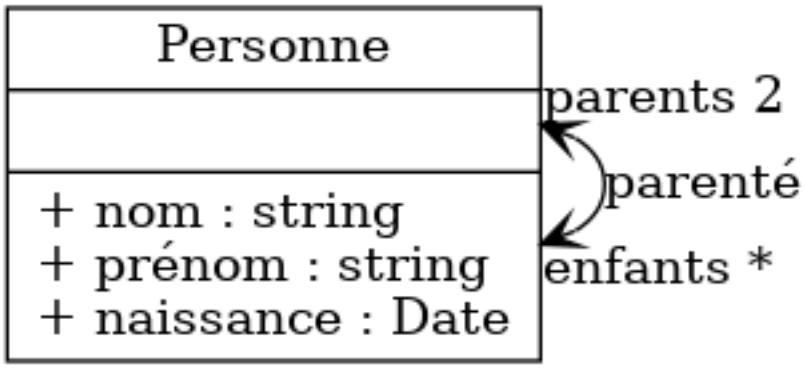

Figure 6: Exemple d'association réflexive : Relation de parenté entre personnes

3.2.2 Associations multiples

Example 3.4. Il est possible d'avoir **plusieurs associations** entre deux mêmes classes. Chaque association représente une relation sémantique différente entre les objets des classes concernées.

Par exemple, entre les classes `Personne` et `Appartement`, on peut avoir deux associations : `loue` et `propose`. La première représente les appartements que la personne loue, et la seconde ceux qu'elle propose à la location.

3.2.3 Classe-association

Example 3.5. Une **classe-association** est une association qui est elle-même une classe. On utilise une classe-association lorsqu'une association a besoin de porter des attributs qui lui sont propres, ou de participer à d'autres associations. Elle permet de paramétrer une association avec des informations supplémentaires.

Par exemple, la relation `travaille pour` entre `Personne` et `Entreprise` peut être représentée par une classe-association `Emploi`. Cette classe `Emploi` peut avoir des attributs comme `intitulé`, `date de début`, `date de fin`.

3.2.4 Association n-aire

Example 3.6. Une **association n-aire** est une association qui relie **plus de deux classes**. L'association ternaire (n=3) est un cas particulier d'association n-aire fréquemment rencontré. Ces associations sont utilisées pour modéliser des relations complexes impliquant plusieurs entités.

Par exemple, une association ternaire pourrait relier les classes `Enseignant`, `Étudiant` et `Cours` pour représenter l'inscription des étudiants à des cours dispensés par des enseignants.

Remark 3.7. La multiplicité d'une association n-aire est définie différemment des associations binaires. Pour chaque combinaison de (n-1) objets participants à l'association, la multiplicité contraint le nombre

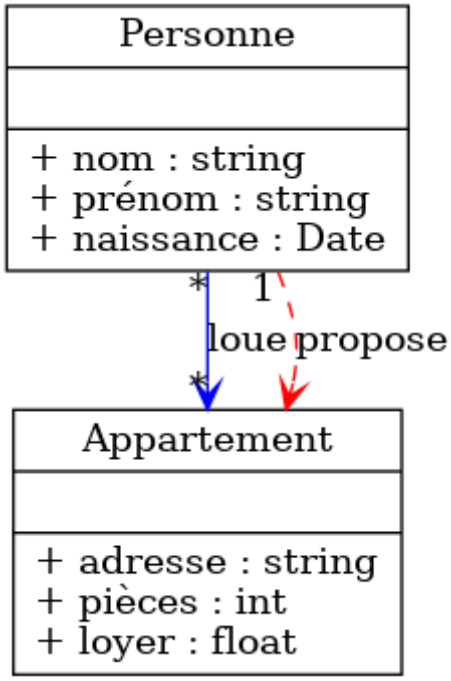

Figure 7: Exemple d'associations multiples entre `Personne` et `Appartement`

d'objets de la classe restante qui peuvent être associés.

3.3 Agrégation et Composition

L'agrégation et la composition sont des formes particulières d'association qui expriment des relations de type **composant-composite**. Elles indiquent qu'une classe (le composite) est constituée d'autres classes (les composants).

3.3.1 Agrégation faible (Agrégation par référence)

Example 3.8. L'**agrégation faible** représente une relation de type "has-a" (a un). Le composite contient une **référence** vers ses composants. Les composants peuvent exister indépendamment du composite et peuvent être partagés entre plusieurs composites. Le cycle de vie du composite et de ses composants est indépendant. On utilise un losange **vide** du côté du composite pour représenter l'agrégation faible.

Par exemple, une `Liste de lecture` (composite) est composée de plusieurs `Morceaux` (composants). Un morceau peut appartenir à plusieurs listes de lecture. Si on supprime une liste de lecture, les morceaux qui la composent continuent d'exister.

3.3.2 Composition (Agrégation forte ou Composition par valeur)

Example 3.9. La **composition** est une forme plus forte d'agrégation qui représente une relation de type "part-of" (fait partie de). Le composite **contient** physiquement ses composants. Les composants sont **exclusivement** liés au composite et leur cycle de vie est dépendant de celui du composite. Si le composite est détruit, ses composants sont également détruits. On utilise un losange **plein** du côté du composite pour représenter la composition.

Par exemple, un `Album` (composite) est composé de plusieurs `Morceaux` (composants). Un morceau

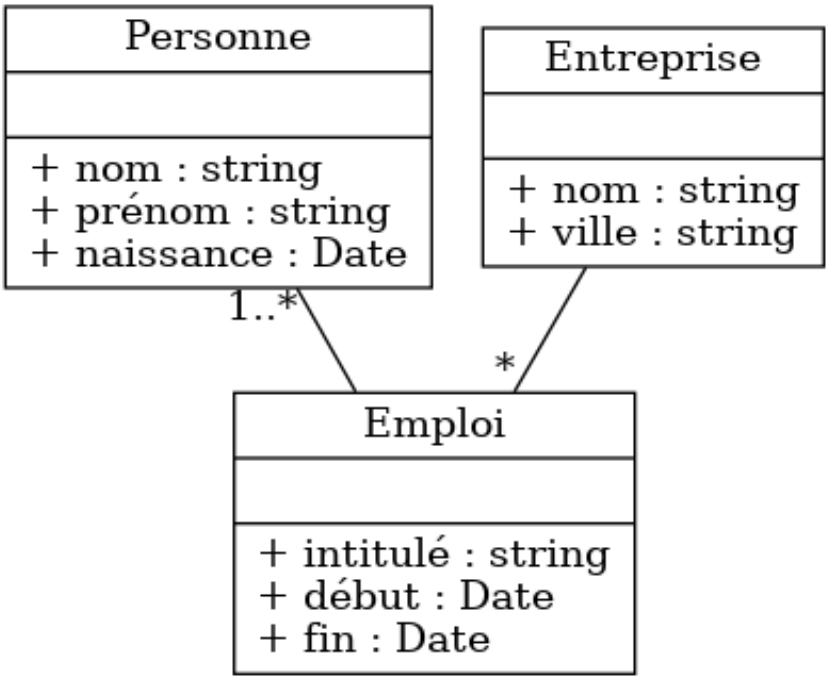

Figure 8: Exemple de classe-association : `Emploi` entre `Personne` et `Entreprise`

appartient à un et un seul album. Si on supprime un album, les morceaux qui le composent sont également supprimés.

3.3.3 Différence entre Agrégation et Composition

Remark 3.10. La principale différence entre agrégation faible et composition réside dans la **dépendance du cycle de vie** des composants par rapport au composite et dans le **partage** potentiel des composants. La composition implique une dépendance forte et une exclusivité, tandis que l'agrégation faible implique une indépendance et un partage possible. Graphiquement, la distinction se fait par le losange (vide pour l'agrégation, plein pour la composition).

4 Généralisation et Héritage

La généralisation et l'héritage sont des mécanismes fondamentaux de la modélisation orientée objet qui permettent d'organiser les classes en **hiérarchies** et de factoriser les attributs et opérations communs.

4.1 Hiérarchie de classes : Super-classes et Sous-classes

Remark 4.1. L'héritage permet de définir une classe (appelée **sous-classe** ou classe dérivée) à partir d'une autre classe existante (appelée **super-classe** ou classe de base). La sous-classe hérite des attributs et des opérations de la super-classe. On organise ainsi les classes en une **hiérarchie** ou **arborescence** de classes.

La **généralisation** est le processus d'abstraction qui consiste à identifier les caractéristiques communes à plusieurs classes et à les regrouper dans une super-classe. La **spécialisation** est le processus inverse, qui consiste à raffiner une classe générale (super-classe) en classes plus spécifiques (sous-classes).

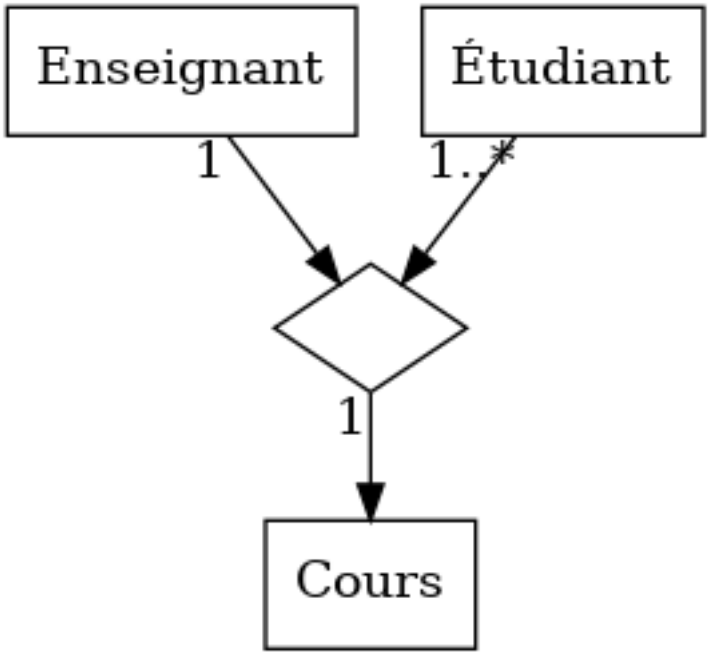

Figure 9: Exemple d'association ternaire entre `Enseignant`, `Étudiant` et `Cours`

Par exemple, on peut généraliser les classes `CompteCourant` et `CompteEpargne` en une super-classe `Compte`, qui représente les caractéristiques communes à tous les types de comptes bancaires (numéro, devise, solde, opérations déposer, retirer, solde). `CompteCourant` et `CompteEpargne` deviennent alors des sous-classes de `Compte`, héritant de ses attributs et opérations, et ajoutant leurs propres spécificités (découvert autorisé, frais de découvert pour `CompteCourant`, plafond, taux pour `CompteEpargne`).

Dans un diagramme de classes, l'héritage est représenté par une flèche à tête triangulaire vide pointant de la sous-classe vers la super-classe.

4.2 Héritage des attributs et des opérations

Lorsqu'une classe B hérite d'une classe A, elle hérite de tous les attributs et opérations définis dans la classe A. Les attributs et opérations hérités sont disponibles dans la sous-classe comme s'ils étaient directement définis dans celle-ci. La sous-classe peut également ajouter ses propres attributs et opérations spécifiques, et peut redéfinir (surcharger) les opérations héritées pour adapter leur comportement.

L'héritage favorise la réutilisation du code et la cohérence dans la modélisation. Il permet de créer des hiérarchies de classes bien structurées et de gérer la complexité des systèmes objets.

4.3 Classes abstraites

Definition 4.2. Une **classe abstraite** est une classe qui ne peut pas être instanciée directement. Elle sert uniquement de **base** pour définir des classes héritées (sous-classes concrètes). Une classe abstraite contient généralement des opérations abstraites, c'est-à-dire des opérations qui sont déclarées mais non implémentées dans la classe abstraite elle-même. Les sous-classes concrètes doivent implémenter ces opérations abstraites.

Une classe abstraite est utilisée pour modéliser des concepts généraux qui n'ont pas de réalisation concrète en soi, mais qui servent de modèle commun pour des concepts plus spécifiques.

Dans un diagramme de classes, une classe abstraite est généralement notée de deux manières :

- En mettant le **nom de la classe en italique**.

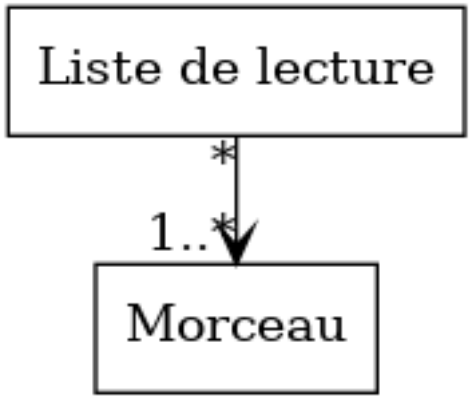

Figure 10: Exemple d'agrégation faible : `Liste de lecture` et `Morceau`

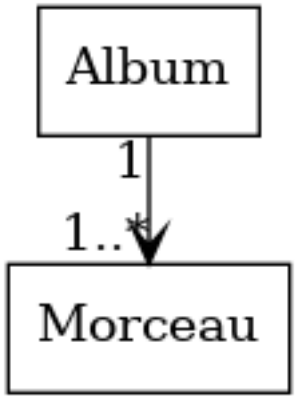

Figure 11: Exemple de composition : `Album` et `Morceau`

- En utilisant le **stéréotype abstract** au-dessus du nom de la classe.

Par exemple, la classe `Compte` pourrait être définie comme une classe abstraite, car un compte bancaire générique n'existe pas en soi. Seuls les types spécifiques de comptes (compte courant, compte épargne, etc.) sont concrets.

5 Exemple complet : Modélisation d'une bibliothèque

Pour illustrer l'ensemble des concepts présentés, nous allons développer pas à pas un modèle UML de diagramme de classes pour un système de gestion de bibliothèque.

5.1 Présentation du cas d'étude

On souhaite développer un système qui gère les emprunts et les retours de documents dans une bibliothèque.

La bibliothèque gère deux types de ressources : des **livres** et des **revues**. Un livre est caractérisé par son titre, son auteur et son code ISBN. Un numéro de revue est caractérisé par le titre de la revue, un numéro de volume et sa date de parution. Chaque **exemplaire** d'une ressource est identifié par un code barre.

Pour emprunter un ouvrage, un **utilisateur** doit être enregistré. Il s'enregistre auprès du bibliothécaire en donnant son nom et verse une caution. Chaque ouvrage a une caution associée. Un utilisateur ne peut emprunter un ouvrage que si la caution qui lui reste sur son compte est supérieure à la caution de l'ouvrage. La durée de l'emprunt est fixée à 15 jours. On ne peut pas emprunter plus d'un exemplaire d'une même ressource, ni emprunter une nouvelle ressource si on est en retard pour rendre une ressource.

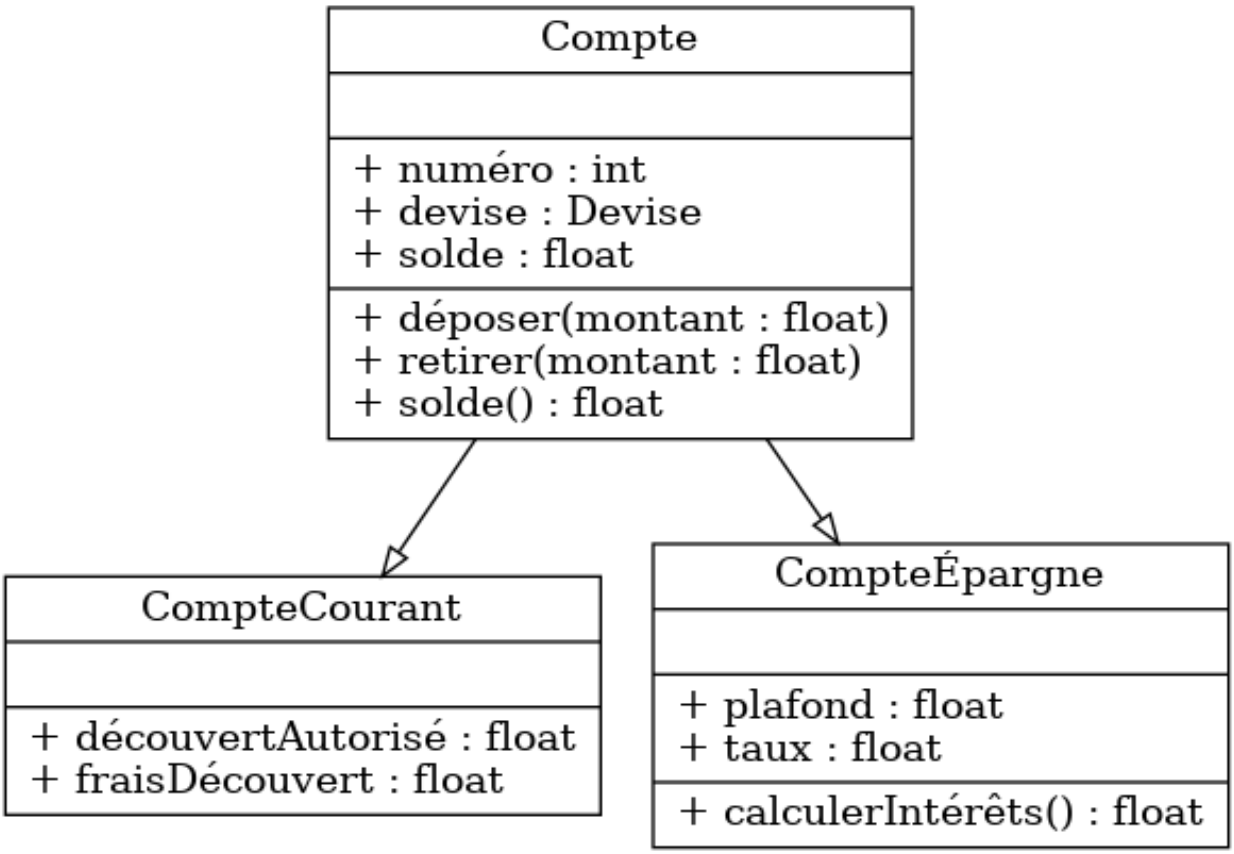

Figure 12: Exemple de hiérarchie de classes : `Compte`, `CompteCourant` et `CompteEpargne`

L'**emplacement** de stockage d'un ouvrage dans la bibliothèque est représenté par un numéro de travée, un numéro d'étagère dans la travée, et un niveau. Différentes ressources peuvent être rangées au même emplacement, mais tous les exemplaires d'une même ressource sont stockés au même endroit.

5.2 Diagrammes de classes successifs

5.2.1 Version 1 : Classes et Attributs

Dans une première version, on identifie les classes principales et leurs attributs, sans se préoccuper des relations.

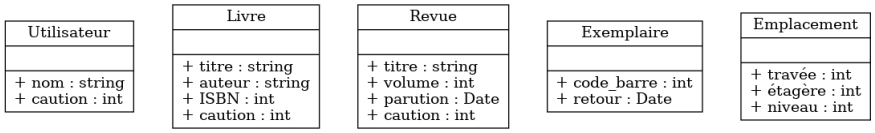

Figure 14: Diagramme de classes - Version 1 : Classes et Attributs

5.2.2 Version 2 : Associations et Multiplicités

On ajoute maintenant les associations entre les classes et on précise les multiplicités.

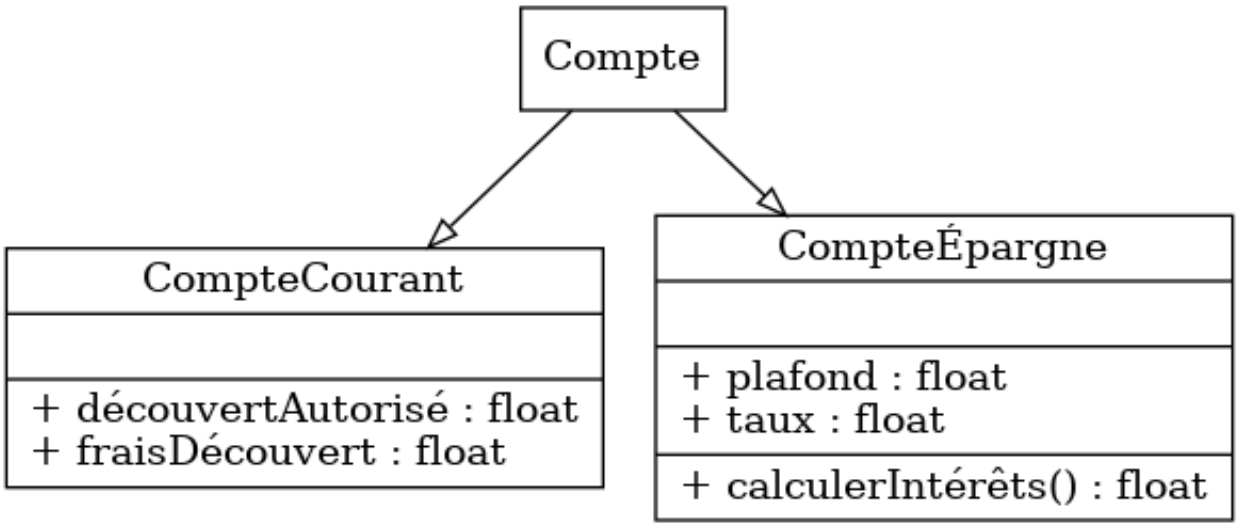

Figure 13: Exemple de classe abstraite : *Compte*

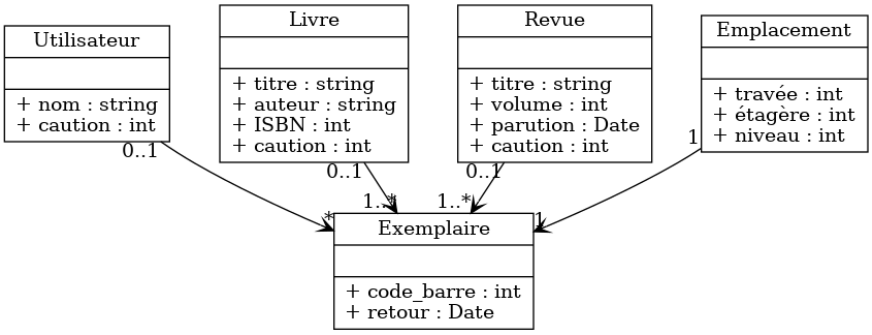

Figure 15: Diagramme de classes - Version 2 : Associations et Multiplicités

5.2.3 Version 3 : Introduction de la classe-association Emprunt

Pour gérer la date de retour d'un exemplaire lors d'un emprunt, on introduit une classe-association `Emprunt`.

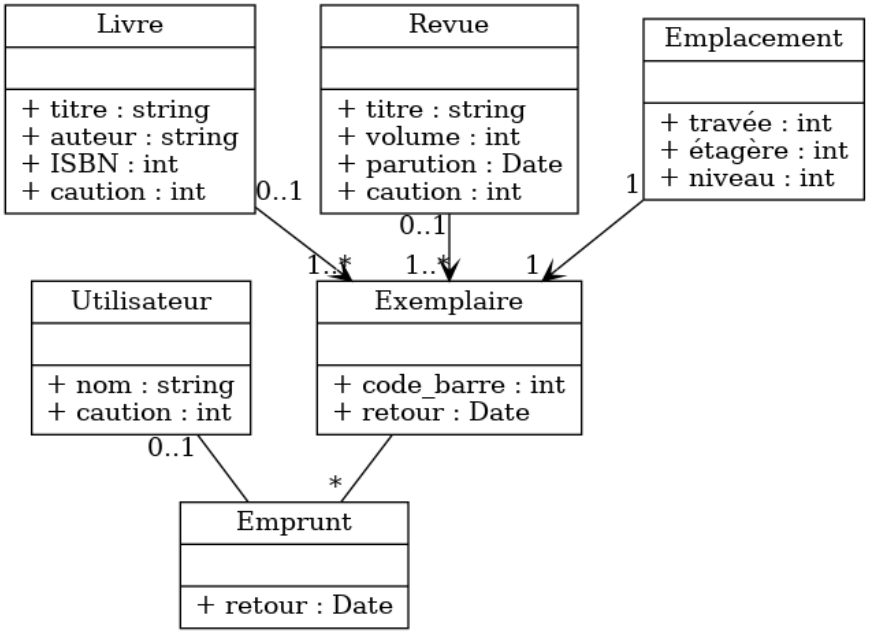

Figure 16: Diagramme de classes - Version 3 : Classe-association `Emprunt`

5.2.4 Version 4 : Généralisation avec la classe abstraite Ressource

On remarque que `Livre` et `Revue` partagent des attributs communs (titre, caution). On peut les généraliser en une classe abstraite `Ressource`.

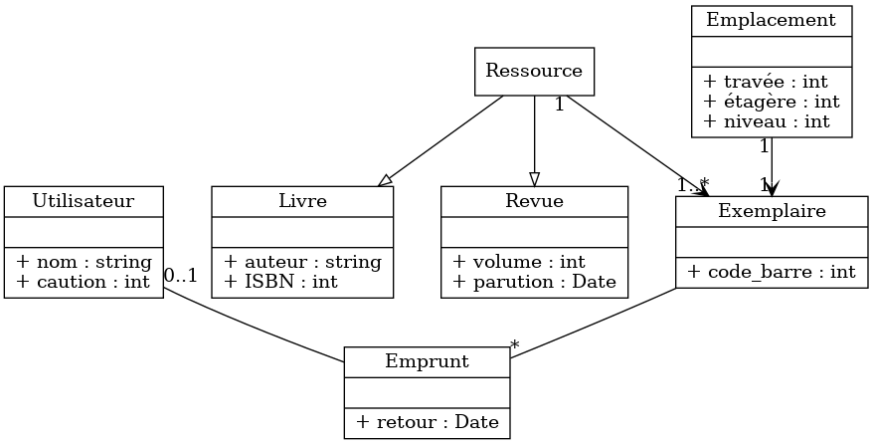

Figure 17: Diagramme de classes - Version 4 : Généralisation avec *Ressource*

5.2.5 Version 5 : Ajout de la composition entre Ressource et Exemplaire

On considère que la suppression d'une ressource (Livre ou Revue) doit entraîner la suppression de ses exemplaires. On utilise donc la composition entre `Ressource` et `Exemplaire`.

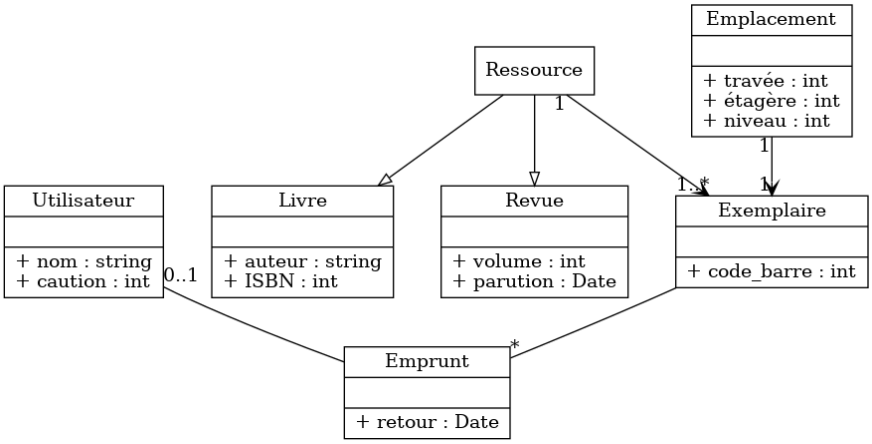

Figure 18: Diagramme de classes - Version 5 : Composition entre `Ressource` et `Exemplaire`

5.3 Exemples de diagrammes d'objets correspondants

Pour chaque version du diagramme de classes, on peut créer des diagrammes d'objets pour illustrer des instances concrètes. Voici des exemples de diagrammes d'objets correspondant aux versions présentées.

5.3.1 Diagramme d'objets pour la version 2

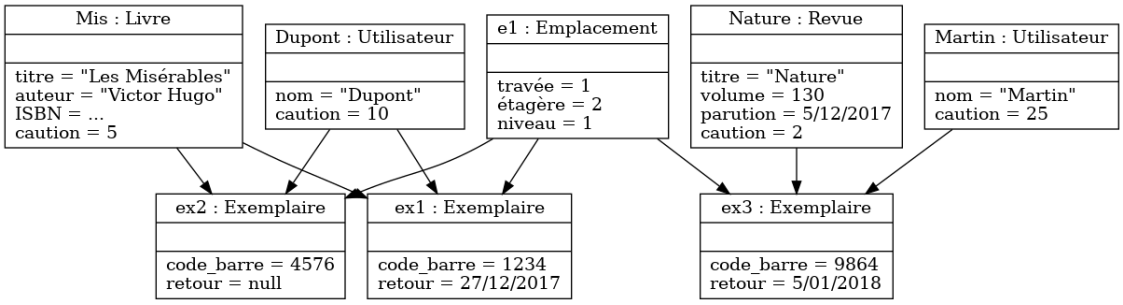

Figure 19: Diagramme d'objets pour la version 2 du diagramme de classes

5.3.2 Diagramme d'objets pour la version 3

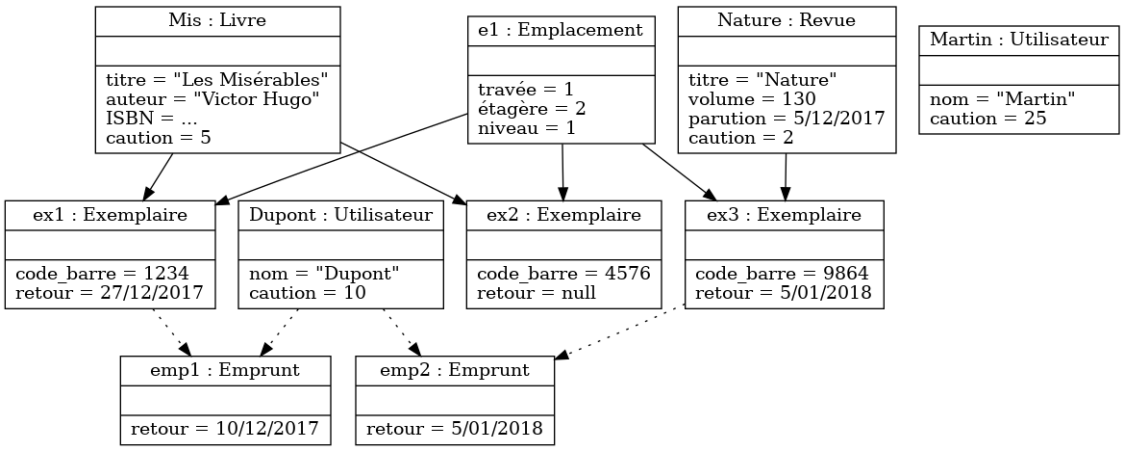

Figure 20: Diagramme d'objets pour la version 3 du diagramme de classes

6 Contraintes

Les diagrammes de classes UML permettent de représenter des contraintes simples graphiquement, notamment à travers les multiplicités. Cependant, pour des contraintes plus complexes, il est nécessaire d'utiliser d'autres mécanismes.

Remark 6.1. Une contrainte importante en modélisation UML est de **ne pas utiliser d'attribut dont le type est une classe du diagramme**. En effet, une relation entre classes doit être modélisée par une **association**, et non par un attribut.

Par exemple, dans une première modélisation, on pourrait être tenté d'ajouter un attribut `propriétaire : Personne` à la classe `Compte`. Cependant, cela viole la règle de modélisation. La relation entre `Compte` et `Personne` (un compte a un propriétaire qui est une personne) doit être modélisée par une association, comme illustré ci-dessous.

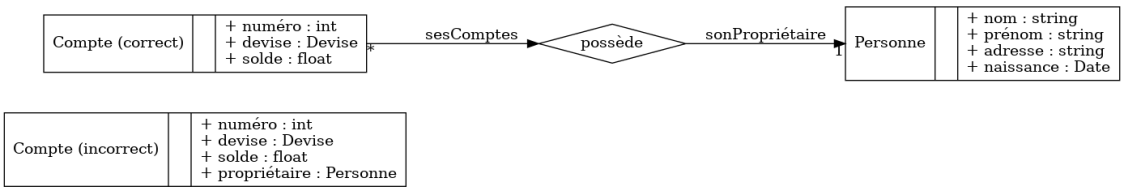

Figure 21: Contrainte : Pas d'attribut dont le type est une classe (comparaison incorrect vs correct)

Remark 6.2. Utiliser une association permet de préciser la multiplicité de la relation (un compte a un et un seul propriétaire, une personne peut avoir zéro ou plusieurs comptes), de nommer les rôles (sonPropriétaire, sesComptes) et potentiellement d'ajouter d'autres caractéristiques à la relation (navigabilité, agrégation, etc.). En résumé, les associations sont plus riches et plus expressives que les simples attributs pour représenter les relations entre objets.