Associations et collections

Associations UML complexes (composition, classes d'associations, associations qualifiées, associations n-aires), et leurs implémentations en Java

Modélisation et Programmation par Objets 1 – cours 4

Nous avons vu les principaux éléments des associations binaires. Nous allons ici aborder des éléments qui permettent de modéliser plus finement certaines situations, discuter du cas particulier des associations réflexives, puis aborder les associations n-aires et les classes d'association. Nous donnerons en parallèle des éléments sur la façon d'implémenter ces différents éléments en Java.

1 Précisions de modélisation sur les associations

Nous avons vu jusqu'ici les associations binaires, à gros grain. De nombreuses informations additionnelles à celles que nous avons vues peuvent être ajoutées. Nous en abordons ici certaines.

1.1 Composition et agrégation

L'agrégation et la composition permettent de préciser qu'une association dénote une relation tout-partie. La relation tout-partie est fréquemment rencontrée[1] : dans la vie courante (un squelette se compose d'os, le squelette est le tout, les os sont les parties ; le corps humain contient un squelette, le corps est le tout, le squelette est une partie, etc ...) mais aussi en informatique (une page web contient des éléments graphiques, la page web est le tout, les éléments graphiques les parties ; un graphe contient des nœuds, le graphe est le tout et les nœuds des parties, etc ...).

La relation tout-partie peut être simple, et on parle alors d'agrégation (le tout est l'agrégat, et les parties les agrégées). Cette relation est symbolisée par un losange creux placé côté agrégat (côté tout).

La relation tout-partie peut aussi être plus forte en incluant une notion d'exclusivité : chaque partie ne peut appartenir qu'à un seul tout. De plus le tout a alors la responsabilité de l'existence et du stockage des parties. Cela implique usuellement que la destruction du tout entraîne la destruction des parties. Toutefois rien n'empêche d'enlever une partie du tout avant sa destruction comme cela est signalé dans la norme. La composition se dénote par un losange plein placé côté composite (côté tout). La cardinalité côté composite est au plus 1.

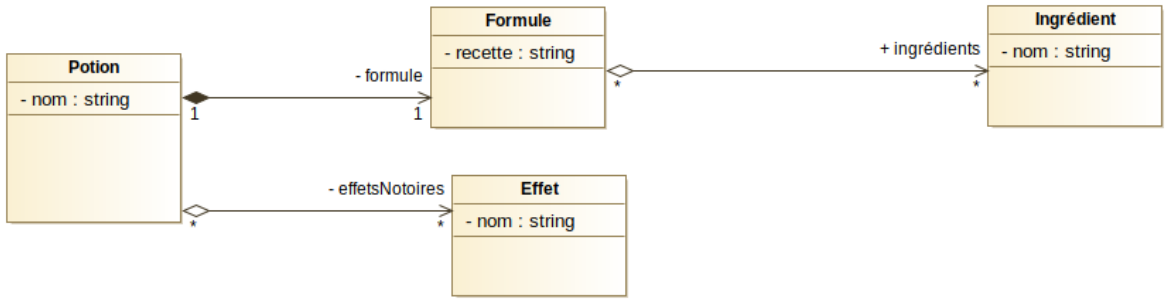

Figure 1 – Agrégation et composition

Nous illustrons ici les 2 concepts à la figure 1, dans laquelle nous nous intéressons aux différentes potions existant dans notre magnifique jeu. Une potion se compose d'une formule. Une formule ne correspond qu'à une seule potion, et la destruction d'un objet Potion entraîne la destruction de sa formule. Une Potion est un agrégat d'effets (les effets induits par l'absorption de la potion). Un même effet (comme ralentir l'activité, ou diminuer la durée de vie du consommateur) peut être commun à plusieurs potions.

La différence entre une association simple et une agrégation est assez ténue, et pour le moins floue. Certains y voient une différence ontologique, où l'agrégat est un objet de premier ordre et les agrégés des objets d'ordre inférieur, tandis que dans une association simple les 2 extrémités sont d'un même niveau conceptuel. La définition de l'agrégation est peu précise, James Rumbaugh, l'un des concepteur initiaux d'UML a d'ailleurs écrit : "Think of it as a modeling placebo".

La composition et l'agrégation ne concernent que les associations binaires, et ne sont donc pas applicables aux associations n-aires que nous verrons plus loin.

1. et pour briller en société, utilisez le terme de méronymie

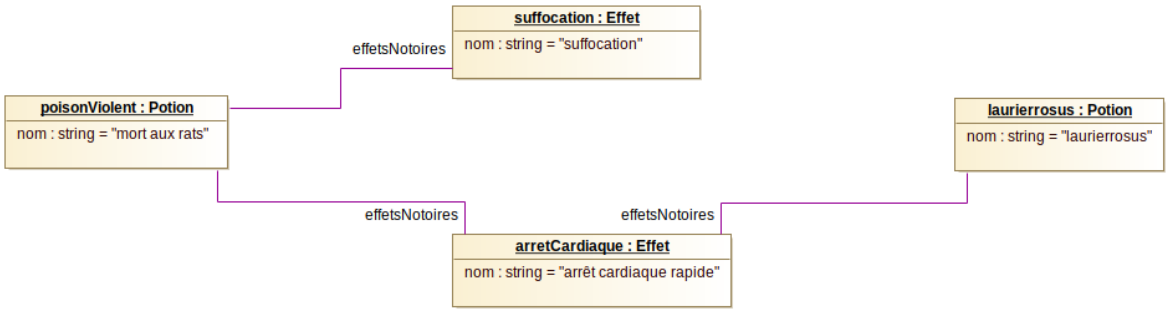

FIGURE 2 – Agrégation : effet partagé !

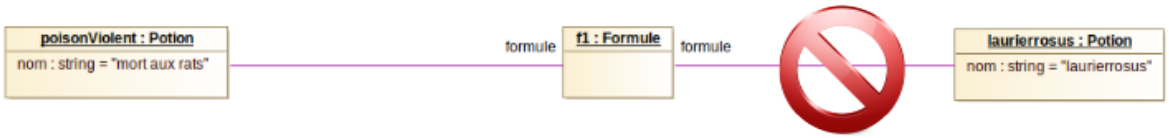

FIGURE 3 – Composition : partage de formule interdit !

Du point de vue de l'implémentation, la présence d'une agrégation n'a pas d'impact. La présence d'une composition, peut avoir un impact, mais qui est délicat à mettre en place. Le premier élément concerne l'exclusivité, et le second la destruction des composants en cas de destruction des composites. Dans notre cas, une même formule ne peut pas définir deux potions différentes, et la destruction d'une potion doit impliquer la destruction de sa formule. Les deux points sont compliqués à mettre en place, du fait que les objets sont transmis par référence. On peut imaginer construire l'objet Formule à l'intérieur du constructeur de Potion (ou via toute autre méthode de la classe Potion si la cardinalité côté Formule passait de 1 à 0..1), ce qui en fait alors un composant dont on est sûr qu'il ne compose pas une autre potion. Par ailleurs, on doit alors s'interdire de placer un constructeur paramétré par une formule et/ou un accesseur en écriture pour la formule. On garantit alors que la référence à la formule détenue par la potion n'est partagée par aucune autre potion. Mais il faut alors s'interdire de diffuser cette référence à l'extérieur de la classe Potion pour satisfaire le deuxième point : la destruction de la potion doit entraîner celle de la formule. Si la référence de formule a été diffusée, elle a pu être stockée par ailleurs, et donc persister à un déréférencement de la potion.

Programme 1 : *Implémentation possible d'une composition, extraits de code pour le modèle de la figure 1*

```
public class Potion {
        private String nom;
        private Formule formule;
        private ArrayList<Effet> effetsNotoires=new ArrayList<>();
        public Potion(String nom, String recette, Ingredient... ing) {
                this.nom=nom;
                this.formule=new Formule(recette, ing);
        }
        ...
}
public class Formule {
        private ArrayList<Ingredient> ingrédients;
        private String recette;
        public Formule (String recette, Ingredient... ing) {
                ingrédients=new ArrayList<>();
                Collections.addAll(ingrédients, ing);
                this.recette=recette;
        }
        ...
}
public class Ingredient {
        private String nom;
        public Ingredient(String nom) {this.nom = nom;}
        ...
```

```
}
public class Effet {...}
```

Nous notons au passage que l'utilisation de constructeurs à nombres variables de paramètres (ce qui n'était pas strictement nécessaire). Les paramètres de taille variable (varargs) ont été introduits en Java 1.5. Ils se signalent syntaxiquement par des points de suspension suivant le type du paramètre ; il ne peut y avoir qu'un paramètre de taille variable par méthode, et c'est obligatoirement le dernier. Concrètement, java les collecte dans un tableau.

1.2 Associations dérivées

Tout comme les attributs dérivés, les associations dérivées sont calculables, en général à partir d'autres associations. Cela signifie que les liens induits par l'association peuvent se déduire de liens induits par d'autres associations. Dans l'exemple de la figure 4, on a repris les associations "se compose de" de Clan vers Personnage et "possède" de Personnage vers Arme. L'association "peut mobiliser" de Clan vers Arme indique les armes qu'un clan peut mobiliser : ce sont toutes les armes stockées par tous les membres du clan. Cette association est donc dérivée (ce qui est indiqué par un /). Elle se calcule en utilisant les 2 autres associations du diagramme. L'extrémité armesMobilisables peut se "calculer" en allant chercher tous les membres du clan puis pour chacun collecter les armes stockées (se qui se traduirait en OCL [2] par l'expression : `Context Clan inv: Clan.armesMobilisables=membres.armesStockées`).

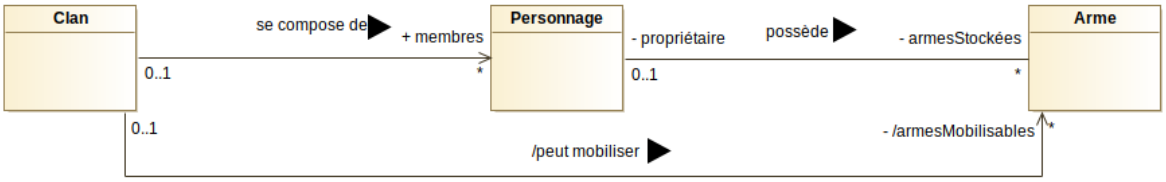

FIGURE 4 – Associations dérivées

Nous abordons également ici l'union derivée (derived union) : on peut spécifier qu'une extrémité d'association est dérivée à partir de l'union d'autres extrémités d'associations. Pour cela :

— on indique à l'extrémité dérivée (de rôle `r`) la contrainte `{union}`
— on indique à chacune des extrémités qui entrent dans le cadre de l'union : `{subsets r}`.

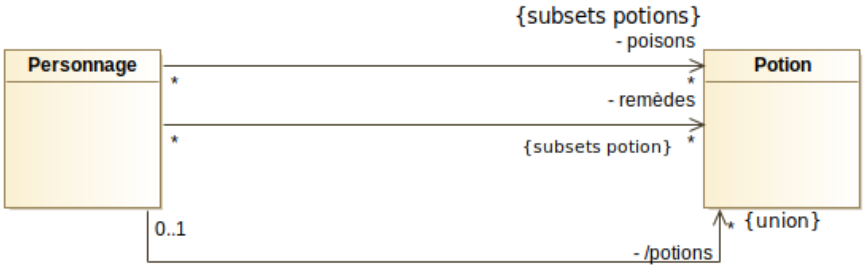

FIGURE 5 – Union dérivée

L'union dérivée est illustrée à la figure 5. Dans ce diagramme, on représente le fait qu'un personnage stocke des potions. Il a été fait le choix de séparer les potions en deux : les poisons d'une part et les emèdes d'autre part, c'est le sens des 2 associations d'extrémités `poisons` et `remèdes`. Pour manipuler de manière homogène toutes les potions, on introduit l'association dérivée d'extrémité `potions`, qui est l'union de `poisons` et `remèdes`. Dit autrement (et c'est comme cela que c'est spécifié en UML) : `poisons` est un sous-ensemble de `potions` et donc `subsets potions`, de manière similaire `remèdes` est un sous-ensemble de `potions` et donc `subsets potions` ; et `potions` est dérivée de l'union de ses sous-ensembles.

Comme pour l'implémentation des attributs dérivés, l'implémentation des associations dérivées présente une alternative : stocker ou calculer. L'association est calculable, toutefois, elle peut aussi être stockée. Si l'on choisit de stocker l'information, il faut veiller à gérer la redondance qui est alors introduite. En contrepartie, l'information sera immédiatement mobilisable, sans calcul préalable. Si l'on choisit de calculer l'information, on implémentera l'extrémité d'association calculable et navigable par une méthode réalisant le calcul dans la classe opposée à l'extrémité d'association. L'information ne sera pas directement mobilisable puisqu'elle nécessitera un calcul, mais en contre-partie il n'y a pas de redondance d'information.

2. OCL (Object Constraint Language) est un langage d'expression de contraintes créé par l'OMG et utilisé entre autres pour préciser des modèles UML

Par exemple, si on reprend l'union dérivée de la figure 5, on peut soit calculer l'union par une méthode (comme au code 2), soit stocker l'union dans une troisième ArrayList, et en gérant la redondance induite, par exemple en veillant à ce que l'ajout (resp. kle retrait) d'un poison ou d'un remède déclenche l'ajout (resp. le retrait) dans la liste union, et en n'ajoutant rien directement dans la liste union (pas d'accesseur d'ajout par exemple), comme illustré au code 3 (les retraits ne sont pas illustrés).

Programme 2 : *Implémentation d'une association dérivée, version "calcul"*

```
public class Personnage {
        ...
        private ArrayList<Potion> poisons=new ArrayList<>();
        private ArrayList<Potion> remèdes=new ArrayList<>();
        ...

        private ArrayList<Potion> getPotions(){
                ArrayList<Potion> res=new ArrayList(poisons);
                res.addAll(remèdes);
                return res;
        }
        ...
```

Programme 3 : *Implémentation d'une association dérivée, version "stockage"*

```
public class Personnage {
        ...

        private ArrayList<Potion> poisons=new ArrayList<>();
        private ArrayList<Potion> remèdes=new ArrayList<>();
        private ArrayList<Potion> potions=new ArrayList<>();

        ...

        public void ajoutPoison(Potion p) {
                // il faudrait sans doute vérifier que p est un poison ...
                poisons.add(p);
                potions.add(p);
        }

        public void ajoutRemède(Potion p) {
                // il faudrait sans doute vérifier que p est un remède ...
                poisons.add(p);
                potions.add(p);
        }
        ...
}
```

Les deux solutions ont donc des avantages et des inconvénients, le choix entre les deux se fait selon la nature des données et leur usage, et la complexité de la règle de dérivation.

1.3 Autres contraintes placées aux extrémités d'association

Des contraintes peuvent être spécifiées aux extrémités des associations, notamment quand les cardinalités maximales sont supérieures strictement à 1. Nous en citerons deux ici :

— `{ordered}` : indique que les éléments sont ordonnés
— `{unique}` : indique que les éléments ne contiennent pas de doublons

Pour prendre en compte ces contraintes lors de la programmation, on choisit une structure de données permettant de les satisfaire pour l'implémentation de l'extrémité d'association contrainte. Par exemple, les ArrayList permettent de maintenir un ordre parmi les éléments, et de garantir l'absence de doublons à condition de contrôler l'ajout d'élément. On peut également utiliser pour des éléments sans doublons la classe TreeSet, qui nécessite cependant que ses éléments soient explicitement comparables, ce qui fait appel en Java à la notion d'interface que nous n'avons pas encore vue.

Comme nous l'avons déjà rencontré en abordant les unions dérivées, on peut ajouter sur une extrémité d'association la contrainte `{subsets x}` pour indiquer que cette extrémité est un sous-ensemble de x, où x est le nom de rôle d'une autre

extrémité d'association. Par exemple, nous aurions pu utiliser cette contrainte pour indiquer que l'arme saisie par un personnage faisait partie de ses armes stockées, comme illustré à la figure 6.

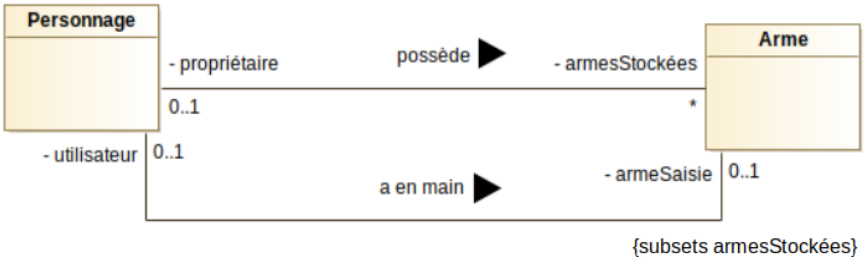

FIGURE 6 – Extrémité "sous-ensemble" d'une autre

La prise en compte d'une contrainte de sous-ensemble lors de l'implémentation se réalise en général par des contrôles sur les accès à l'association sous-ensemble : on s'assure de n'y affecter que des éléments du sur-ensemble, ou bien on y ajoute automatiquement l'élément dans le sur-ensemble.

2 Associations qualifiées et implémentation à base de dictionnaires

2.1 Associations qualifiées

Les associations binaires peuvent présenter à une de leurs extrémités un qualificateur (ou plusieurs), on parle alors d'associations qualifiées. Le qualificateur se matérialise par un attribut placé dans un rectangle et intercalé entre la classe et l'association, comme illustré à la figure 7. Le qualificateur, placé à côté d'une classe A, permet, pour une instance de A donnée, de partitionner [3] les instances associées à l'autre extrémité de l'association. Chaque élément de partage ainsi créé est identifié par une valeur du qualificateur. La taille des éléments de partage est dimensionnée par les cardinalités placées à l'autre extrémité de l'association. Dans l'exemple de la figure 7, à partir d'une instance de A et d'une valeur pour q, on identifie 0 ou 1 instances de B. Le ou les

FIGURE 7 – Association qualifiée

qualificateurs peuvent ainsi être vus comme des clefs d'accès, qui sélectionnent des sous-ensembles d'instances.

Nous allons illustrer les associations qualifiées en étudiant un plateau de jeu, qui, entre autres, gère les personnages présents sur le plateau. Sur le plateau, il y a donc plusieurs (*) personnages. Chaque personnage a un pouvoir (qui est une énumération, comme vu précédemment). Cela est représenté à la figure 8. Du point de vue des instances, cela signifie que pour chaque plateau, l'association identifie plusieurs personnages. Nous prenons à la figure 9 un exemple de plateau "Larzac" sur lequel se trouvent

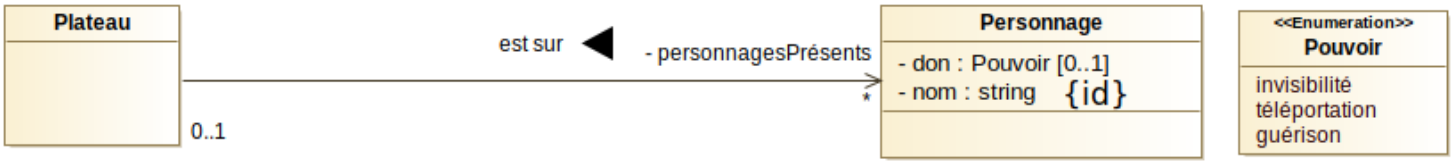

FIGURE 8 – Association non qualifiée entre Plateau et Personnage

4 personnages : Ronflex, Popoki, Gandalf et Mario. Si l'on ajoute le qualificateur `nom:String`, comme illustré à la figure 10, on sélectionne avec un plateau et un nom zéro ou un (0..1) personnages : les noms sont supposés ici être des identifiants des personnages [4], donc il n'y a pas plus d'un personnage portant un nom donné dans cette association. Si le nom qualificateur ne correspond à aucun personnage dans l'association, alors ce nom sélectionnera zéro personnage. Si on regarde au niveau instances, on a par exemple :

— à partir du plateau "Larzac" et du nom "Popoki le chat" on sélectionne l'unique objet popoki
— à partir du plateau "Larzac" et du nom "Ronflex" on sélectionne l'unique objet ronflex
— à partir du plateau "Larzac" et du nom "Attila" on ne sélectionne aucun objet.

3. Le terme "partitionner" est ici à prendre au sens commun, et non au sens mathématique
4. Nous avons à cet effet rajouté le petit `{id}` lors de la déclaration de l'attribut `nom`

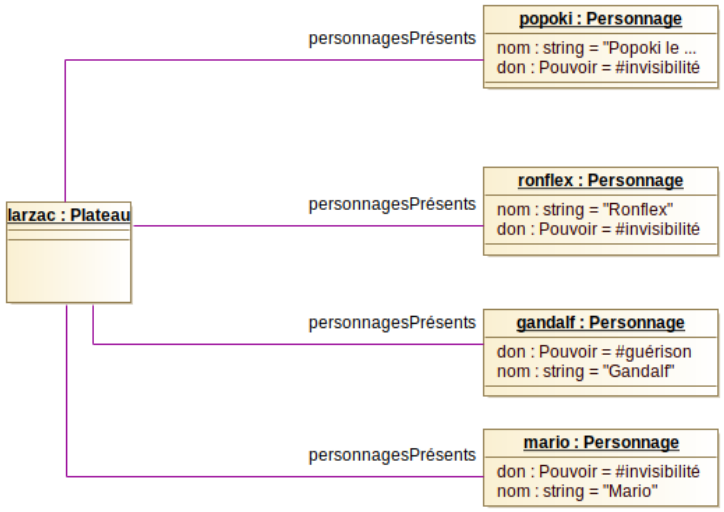

FIGURE 9 – Liens entre un plateau et 4 personnages

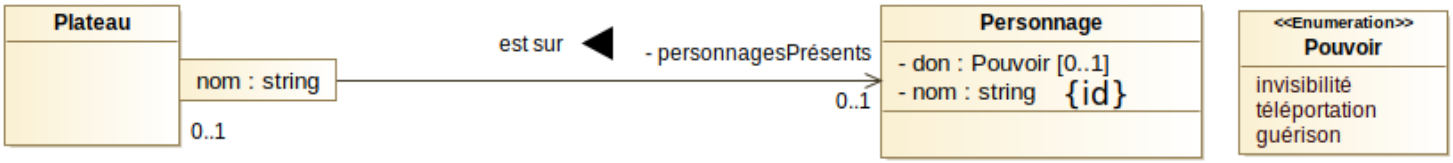

FIGURE 10 – Association qualifiée par le nom, entre Plateau et Personnage

Si par contre on rajoute le qualificateur `don:Pouvoir`, comme illustré à la figure 11, on sélectionne pour un plateau et un pouvoir plusieurs personnages (*) : les dons ne sont pas des identifiants, et donc plusieurs personnages peuvent avoir le même don (comme par exemple ici popoki, ronflex et mario ont le même don d'invisibilité). Si on regarde au niveau instances, on a par exemple :

— à partir du plateau "Larzac" et du don `#invisibilité` on sélectionne les objets popoki, ronflex et mario.
— à partir du plateau "Larzac" et du don `#guérison` on sélectionne l'objet gandalf.
— à partir du plateau "Larzac" et du don `#téléportation` on ne sélectionne aucun objet (dans ce plateau, aucun personnage n'a ce pouvoir).

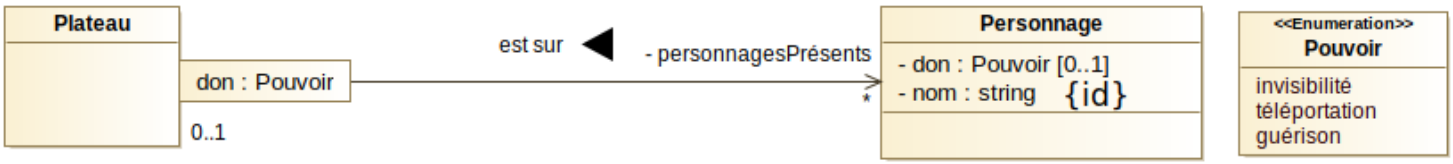

FIGURE 11 – Association qualifiée par le don, entre Plateau et Personnage

2.2 Implémentation à base de dictionnaires

Pour implémenter des associations qualifiées (ou plus exactement l'extrémité opposée au qualificateur, car l'autre extrémité ne soulève aucun point particulier), on utilise en général des dictionnaires (aussi appelés tableaux associatifs, ou encore tables d'association). Nous en donnons ici les grands principes d'utilisation en Java, en illustrant avec la classe HashMap.

Un dictionnaire est un type de données qui permet d'associer un ensemble de valeurs à un ensemble de clefs. Si on prend l'exemple non informatique d'un dictionnaire (vous savez, le gros livre ...), les clefs sont les mots présents dans le dictionnaire, et les valeurs sont les définitions. D'un point de vue logique, la notion de dictionnaire est une extension des tableaux, pour lesquels l'index (la clef) n'est pas nécessairement un entier.

Les opérations classiques sur un dictionnaire sont :

— ajout : on ajoute une nouvelle entrée dans le dictionnaire, c'est-à-dire une clef et la valeur correspondante.
— modification : on modifie la valeur associée à une clef existante.

— recherche : on cherche (et retourne) la valeur correspondant à une clef.
— suppression : on supprime une clef (et donc la valeur correspondante).

Si l'on reprend l'exemple des personnages présents sur un plateau, et indexé par leur nom, on peut implémenter les personnages présents avec un dictionnaire, indexé par le nom. Une vue logique d'un tel dictionnaire est donné à la figure 12.

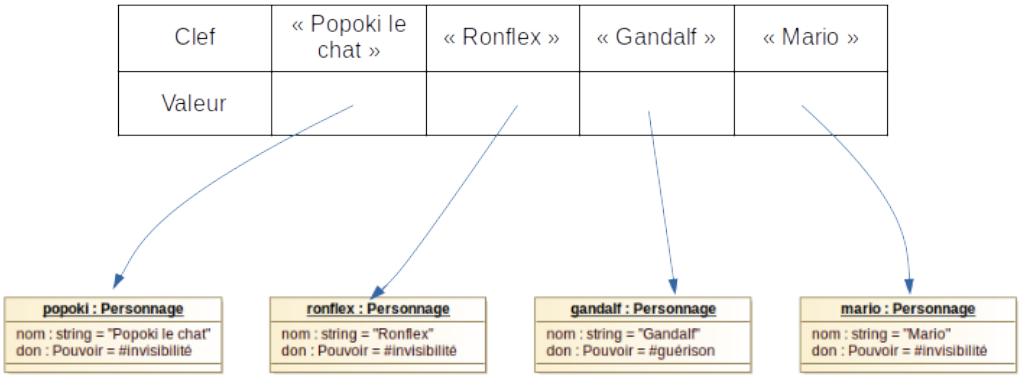

FIGURE 12 – Dictionnaires : vue logique

L'accès à une valeur à partir d'une clef est en général implémenté au sein d'un dictionnaire de manière à être rapide. Nous ne verrons pas ici les mécanismes mis en œuvre pour cela.

La classe HashMap de l'API Java est une implémentation de dictionnaire. Le terme "Hash" provient du mécanisme de hachage utilisé pour l'implémentation du dictionnaire, mais nous ne l'aborderons pas ici. La classe HashMap est une classe générique, paramétrée par les types des éléments qu'elle va manipuler : le type des clefs, usuellement dénoté K (pour Key) et le type des valeurs, usuellement dénoté V.

Pour déclarer un objet de type HashMap, on doit donc donner les types effectifs pour les clefs et les valeurs, ainsi qu'illustré au code 4.

Programme 4 : *Utilisation des HashMap : déclaration*

```
public class Plateau {
        private HashMap<String, Personnage> personnagesPrésents=new HashMap<>();
        ...
}
```

Pour ajouter un élément à une hashmap, on utilise la méthode put, et on doit donc fournir une clef et une valeur. Pour ajouter popoki on utilise par exemple : `personnagesPrésents.put("Popoki le chat", popoki);`. Un exemple de méthode d'ajout de personnage présent est illustrée au code 5 : on place une nouvelle entrée en choisissant pour clef le nom du personnage et pour valeur le personnage.

Programme 5 : *Utilisation des HashMap : ajout*

```
public class Plateau {
        private HashMap<String, Personnage> personnagesPrésents=new HashMap<>();

        public void ajoutPersonnagePrésent(Personnage p) {
                personnagesPrésents.put(p.getNom(),p);
        }
        ...
}
```

Il est à noter que dans cet exemple, s'il existait déjà une entrée pour la clef (c'est-à-dire un personnage de même nom dans la table), celle-ci aurait été silencieusement écrasée.

Pour tester l'existence d'une clef dans une hashmap, on utilise la méthode containsKey, ainsi qu'illustré au code 6 (les homonymes n'existant pas théoriquement, nous nous contentons ici d'afficher un message indiquant que le personnage existe déjà).

Programme 6 : *Utilisation des HashMap : existence d'une clef*

```
public class Plateau {
        private HashMap<String, Personnage> personnagesPrésents=new HashMap<>();
```

```
        public void ajoutPersonnagePrésent(Personnage p) {
                if (personnagesPrésents.containsKey(p.getNom())) {
                        System.out.println("personnage déjà présent");
                }
                personnagesPrésents.put(p.getNom(),p);
        }
        ...
}
```

Pour obtenir une valeur à partir d'une clef, on utilise la méthode `get`, comme illustré au code 7. Si la clef n'existe pas dans la table, la valeur null est retournée.

Programme 7 : *Utilisation des HashMap : obtention d'une valeur*

```
public class Plateau {
        private HashMap<String, Personnage> personnagesPrésents=new HashMap<>();

        public Personnage getPersonnagePrésent(String nom) {
                return personnagesPrésents.get(nom);
        }
        ...
}
```

Pour supprimer une clef de la table, on utilise la méthode `remove`, comme illustré au code 8. Si la clef n'existe pas dans la table, la méthode retourne faux (et vrai sinon).

Programme 8 : *Utilisation des HashMap : suppression d'une clef*

```
public class Plateau {
        private HashMap<String, Personnage> personnagesPrésents=new HashMap<>();

        public Personnage removePersonnagePrésent(String nom) {
                return personnagesPrésents.remove(nom);
        }
        ...
}
```

Pour parcourir une table, on peut le faire clef par clef ou valeur par valeur. Pour obtenir l'ensemble des clefs, on utilise la méthode `keySet` tandis que pour obtenir les valeurs, on utilise la méthode `values`, comme illustré à la figure 9. On peut également obtenir toutes les entrées du dictionnaire, mais nous ne l'aborderons pas ici (notion de classe/interface imbriquée non vue dans ce cours).

Programme 9 : *Utilisation des HashMap : parcours*

```
public class Plateau {
        private HashMap<String, Personnage> personnagesPrésents=new HashMap<>();

        public void afficherLesNomsDesPrésents() {
                for (String n:personnagesPrésents.keySet()) {
                        System.out.println(n);
                }
        }

        public void afficherLesPrésents() {
                for (Personnage p:personnagesPrésents.values()) {
                        System.out.println(p);
                }
        }
        ...
}
```

3 Associations réflexives

Certaines associations binaires ont leur deux extrémités reliées à la même classe. On parle alors d'association réflexive. Ce sont toutefois des associations normales, qui s'implémentent normalement. Il faut bien prendre soin au nommage des rôles dans ces associations réflexives, surtout dans le cas où les deux extrémités ne jouent pas le même rôle. Une instance peut être associée avec elle-même via une association réflexive, si l'on souhaite exclure ce cas de figure, il faut ajouter une contrainte le spécifiant, ce que nous n'aborderons pas ici.

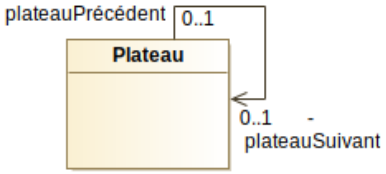

FIGURE 13 – Association réflexive

4 Classes d'associations

Les classes d'association interviennent quand on souhaite modéliser des éléments (attributs, opérations) qui concernent une association.

Par exemple, on s'intéresse à l'association de la figure 14 qui représente les denrées possédées par un personnage.

FIGURE 14 – Personnage et Denrée

Via cette association, on va pouvoir exprimer des liens indiquant par exemple que popoki a des croquettes, et que ronflex et gandalf ont de l'orge comme illustré à la figure 15.

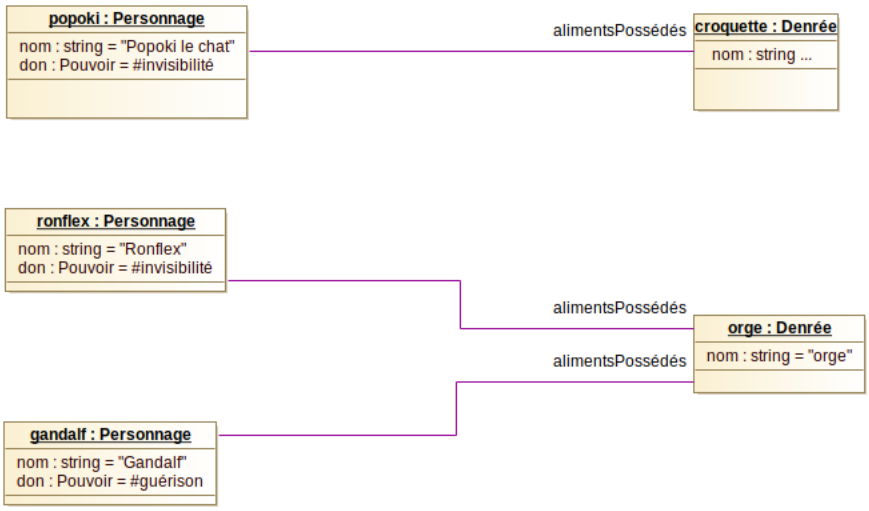

FIGURE 15 – Personnages et Denrées (instances)

Mais où ajouter la quantité de denrée possédée ou encore la date d'obtention ? Ces informations ne peuvent être ajoutées ni dans la classe Personnage ni dans la classe Denrée, où elles n'ont pas de sens. En fait, ces informations n'ont de sens que dans le contexte de l'association entre Personnage et Denrée. Comme illustré à la figure 16, nous introduisons donc une classe ItemAlimentaire, avec pour attribut une quantité et une date d'obtention (ici un entier pour simplifier), cette classe est reliée à l'association : c'est une classe d'association.

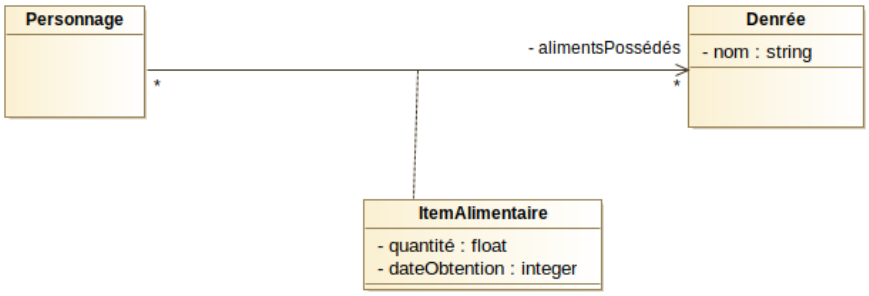

FIGURE 16 – Classe d'association

Du point de vue des instances, cela signifie que sur chaque lien entre Personnage et Denrée, il existera une instance d'ItemAlimentaire qui portera les valeurs de quantité et de date d'obtention, ainsi qu'illustré à la figure 17.

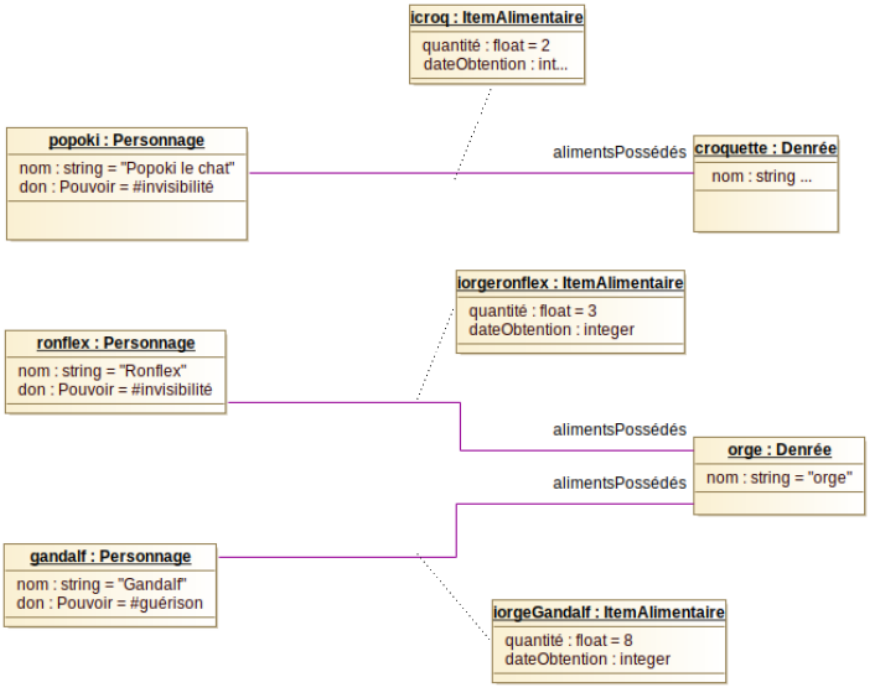

FIGURE 17 – Classe d'association : une instanciation

Du point de vue de l'implémentation, on commence en général par se ramener à une situation sans classe d'association, en fonction des cardinalités. Dans le cas de cardinalités *-* comme celui que nous avons ici, on procède en général comme à la figure 18, ce qui donne finalement au niveau instances celui de la figure 19.

Dans le cas de cardinalités avec borne supérieure 1, on peut procéder de la même manière, ou bien décider de ranger les attributs dans la classe du côté de cette cardinalité 1.

5 Associations n-aires

Les associations représentent le cas général des associations : elles peuvent lier non pas seulement 2 classes comme les associations binaires mais n classes, avec n arbitrairement grand. En pratique, n n'est jamais si grand que cela, et les associations n-aires sont très peu usitées. Nous les abordons rapidement ici, en suivant un exemple d'association quaternaire.

Nos personnages peuvent cultiver des végétaux, sur des parcelles, ce qui permet de produire des denrées. Une modélisation possible est donnée à la figure 20. L'association n-aire est matérialisée par un losange, d'où sortent les 4 extrémités d'association (qui ont comme pour les associations binaires une cardinalité et un nom de rôle). Ici la culture est réalisée par exactement un personnage, pour cultiver exactement un végétal, sur 1 ou plusieurs lieux (les parcelles cultivées) pour produire 0 ou 1 denrée.

Pour implémenter les associations n-aires, il est naturel de les réifier, c'est-à-dire ici de les transformer en classe. Ainsi, l'association culture peut se transformer en une classe Culture.

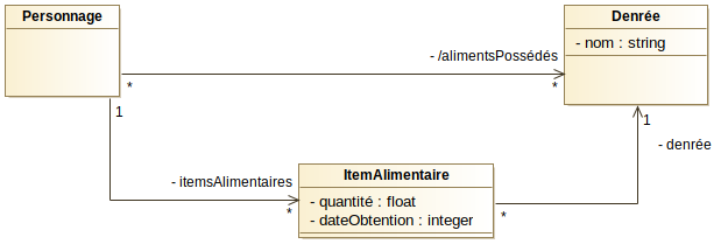

FIGURE 18 – Vers une implémentation de la classe d'association de la figure 16

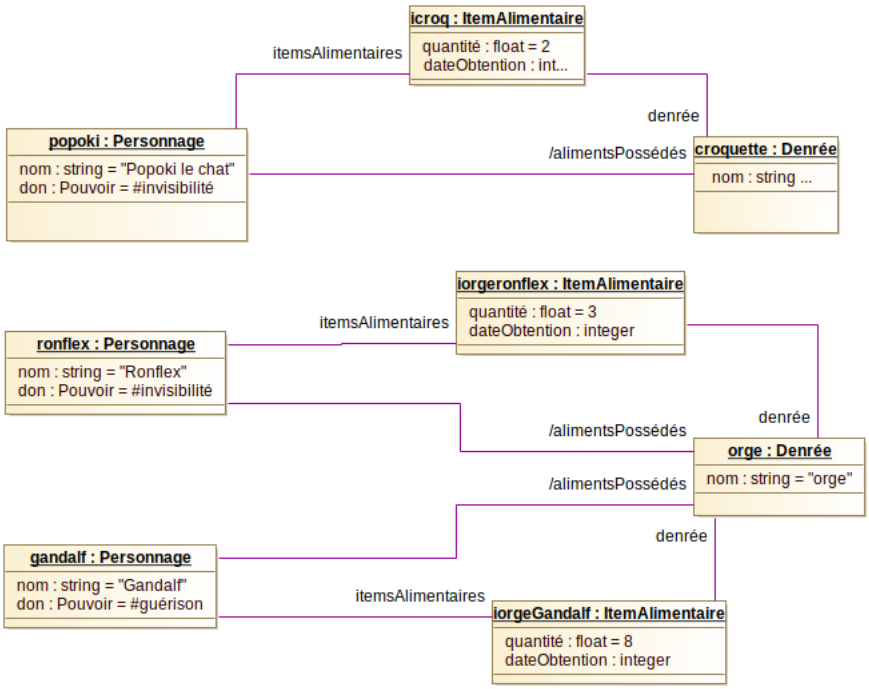

FIGURE 19 – Vers une implémentation de la classe d'association de la figure 16 : une instanciation

6 Conclusion

Nous avons vu comment préciser (dans certains cas) les associations pour prendre en compte différents éléments : certaines peuvent se dériver d'autres, certaines extrémités d'associations sont ordonnées, certaines associations reflètent de relations tout-partie, etc. Nous avons également vu le concept de classe d'association qui permet de faire porter des attributs et des méthodes sur les associations, et les associations qualifiées avec des éléments d'implémentation par des HashMap en Java. Nous avons évoqué les associations n-aires, liant n classes.

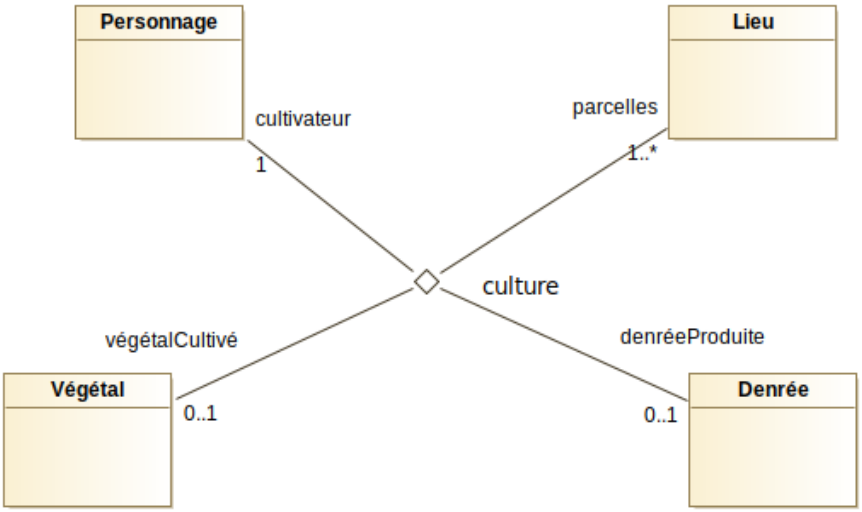

FIGURE 20 – ssociation n-aire