Associations et collections

Associations UML binaires simples et leurs implémentations en Java

Modélisation et Programmation par Objets 1 – cours 3

Les principales approches de modélisation par objets, dont UML est la plus répandue actuellement, permettent de concevoir des relations entre les concepts (essentiellement les classes). En UML ces relations s'appellent des associations. Nous verrons ici des premiers éléments sur les associations, en nous limitant aux associations binaires. Nous verrons ensuite comment ces associations peuvent être traduites dans un langage de programmation tel que Java.

1 Associations binaires entre classes, liens entre objets

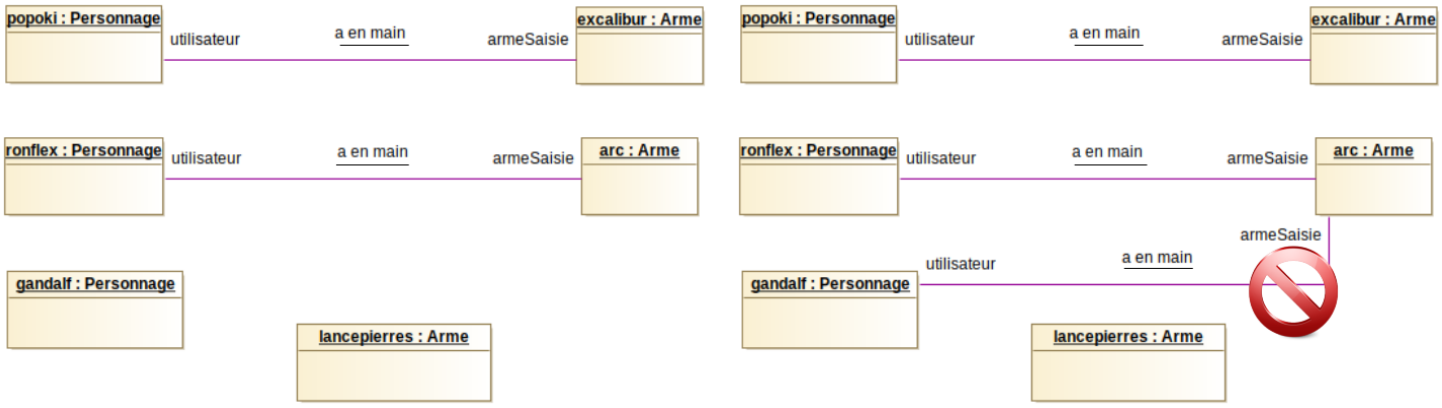
1.1 Associations binaires et liens

Dans un système à objets, les objets sont rarement "isolés" : ils se "connaissent" pour pouvoir s'envoyer des messages, et réaliser une tâche. Au niveau des classes, ces relations s'appellent des associations, au niveau des objets (des instances) ces relations s'appellent des liens.

Commençons par un exemple simple, issu des cours précédents. Un personnage peut prendre une arme en main. Pour être un peu plus précis, cela signifie que chaque personnage a en main une arme ou aucune arme (donc 0 ou 1 arme). Réciproquement, une arme peut donc être prise en main par au plus un personnage. La classe Personnage et la classe Arme sont alors liées par une association (matérialisée par un trait) reflétant cette relation, que l'on peut nommer "a en main" comme illustré à la figure 1. En général, on place une petite flèche triangulaire pour indiquer dans quel sens se lit le nom de l'association : doit-on lire qu'un personnage a une arme en main ou qu'une arme a en main un personnage ? On dit ici que l'association est binaire : elle associe 2 classes.

Figure 1 – Association binaire

Une fois cette association modélisée, des instances de personnages et d'armes peuvent être liées par cette association. Un diagramme d'objets peut alors illustrer ces liens. Par exemple, dans la figure 2a, popoki a une arme en main, ronflex aussi mais pas gandalf. Certaines armes sont saisies par des personnages, d'autres pas.



(a) Liens

(b) Liens, avec un lien incorrect

Figure 2 – Liens pour l'association de la figure 1

On notera que les liens d'un diagramme d'objets font apparaître le nom de l'association en le soulignant, et les noms de rôles non soulignés.

Dans un diagramme d'objets, les cardinalités modélisées par le diagramme de classe doivent être respectées : une arme ne peut pas être saisie par plus d'un personnage, un personnage ne peut pas saisir plus d'une arme. Ainsi, le diagramme d'objets de la figure 2b présente un lien incorrect : l'arc ne peut pas avoir ronflex et gandalf comme propriétaire.

1.2 Extrémités d'associations, rôles et cardinalités

Comme nous l'avons vu dans l'exemple précédent, plusieurs informations sont apportées sur une association. Au centre, se place le nom de l'association, et à chacune des deux extrémités on précise plusieurs informations : le nom de rôle, et la cardinalité. Le nom de rôle placé à une extrémité reflète le rôle que jouent dans cette association les instances de la classe placée à cette extrémité. Par exemple, dans l'exemple de la figure 3, une association modélise la relation "possède" entre la classe Personnage et la classe Arme (un personnage possède plusieurs armes). Un personnage joue le rôle de propriétaire dans l'association, tandis qu'une arme joue le rôle d'armesStockées. On place également aux extrémités d'association des indications sur le nombre minimal et maximal d'instances impliquées dans l'association. Un personnage peut posséder plusieurs armes, on place donc côté Personnage la cardinalité * (qui sous-entend 0..*). Une arme ne peut être possédée que par au plus un personnage, on place donc la cardinalité 0..1. Il est à noter que l'on a tendance à mettre le nom de rôle au pluriel quand la multiplicité (cardinalité) indique plusieurs éléments jouant le même rôle. La cardinalité * côté Arme signifie qu'un même personnage peut posséder plusieurs armes. Par

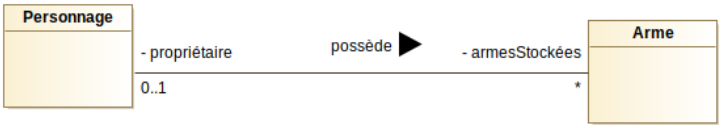


FIGURE 3 – Association binaire

exemple, à la figure 4, popoki possède excalibur et le lance-pierres : popoki a donc 2 armes, ce qui est conforme à la cardinalité *. Les cardinalités s'expriment en général sous la forme $i..j$ où i est la borne inférieure (le nombre minimal d'instances impliquées)

FIGURE 4 – Diagrammes d'instances pour le diagramme de classes de la figure 4

et j la borne supérieure (le nombre maximal d'instances impliquées). Quand il n'y a pas de borne supérieure connue, on utilise le symbole *. La notation $0..*$ peut se noter par le raccourci $*$.

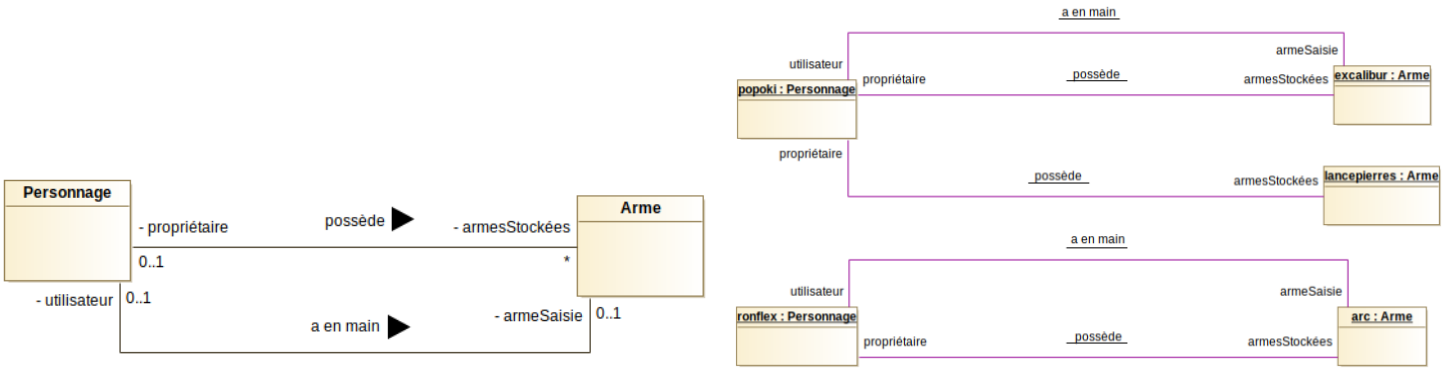
Bien évidemment, deux mêmes classes peuvent être liées par plusieurs associations. Si nous reprenons nos 2 classes Personnage et Arme, nous y avons placé deux associations : "a en main" et "possède", comme montré sur le diagramme de la figure 5a. L'une des associations permet de déterminer quelle arme est dans la main de quel personnage, l'autre permet de déterminer quelles armes sont possédées par quels personnages. Cela est illustré par le diagramme d'objets de la figure 5b.

Dans le cas de ces 2 associations, il est raisonnable de penser que, comme c'est le cas pour le diagramme d'objets de la figure 5b, l'arme qui est dans la main d'un personnage est également possédée par ce même personnage. On voit que cela n'aurait pas beaucoup de sens d'imaginer que l'arc soit dans la main de ronflex mais possédé par popoki. Toutefois, rien ne reflète dans notre diagramme de classes cette contraintes, qu'il faudrait écrire de manière externe, pour indiquer que pour tout personnage, son arme saisie est inclue dans ses armes stockées. On l'écrit ici dans un commentaire, il faudrait en faire un invariant de classe à respecter dans l'implémentation.

Dans le cas le plus général, il n'y a pas de relation particulière entre les instances liées par deux associations distinctes. Par exemple, on pourrait imaginer que les personnages puissent créer puis vendre (ou donner !) des armes. Dans ce cas, on pourrait comme illustré à la figure 6a, modéliser deux associations "possède" et "a fabriqué" entre les classes Personnage et Arme. On voit au diagramme d'instances de la figure 6b, qu'une arme peut avoir été créée par un personnage mais être possédée par un autre.

1.3 Navigabilité

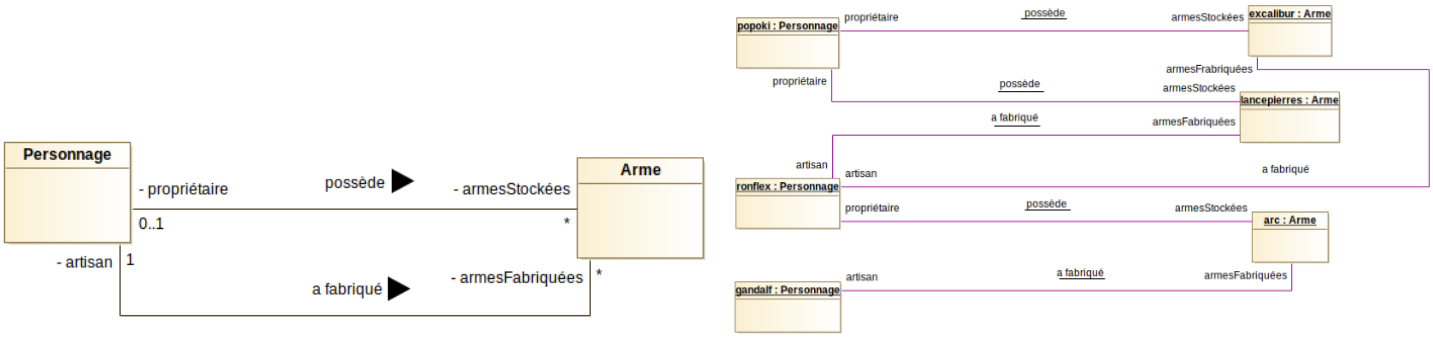
La navigabilité indique dans quel ou quels sens l'association va être naviguée, en d'autres termes : quel groupe d'instances doit avoir accès à l'autre groupe d'instances au travers de cette association. Elle s'exprime en général à l'aide de flèches simples



(a) Deux associations

(b) Des liens correspondants

FIGURE 5 – Associations "possède" et "a en main"



(a) Deux associations

(b) Des liens correspondants

FIGURE 6 – Associations "possède" et "a fabriqué"

placées à l'une, l'autre ou les deux extrémités de l'association. Par exemple, si l'on souhaite que chaque personnage ait accès aux armes qu'il possède, mais réciproquement qu'une arme n'ait pas accès à son propriétaire, on place une flèche du côté de la classe Arme, comme indiqué à la figure 7a. Si l'on souhaite que chaque personnage ait accès aux armes qu'il possède, et également qu'une arme ait accès à son propriétaire, on place une flèche à chaque extrémité, comme indiqué à la figure 7b, on parle alors d'association bi-directionnelle.

(a) Navigabilité des personnage vers les armes possédées

(b) Navigabilité bi-directionnelle

FIGURE 7 – Navigabilité

Par défaut, si aucune indication de navigabilité n'est indiquée, cela peut vouloir dire soit que l'association n'est pas navigable du tout (mais alors elle perd en utilité) soit qu'elle est navigable dans les deux sens. Plus pragmatiquement, cela signifie en général que les choix sur la navigabilité n'ont pas encore été faits. Les versions récentes d'UML introduisent une notation supplémentaire : une petite croix placée à une extrémité d'association indique explicitement que cette extrémité n'est pas navigable. Nous n'utiliserons pas cette notation ici.

1.4 Visibilité

Comme pour les attributs et les méthodes, les extrémités d'associations ont une visibilité qui indique quels éléments peuvent y accéder. La notation est comme pour les attributs. Une extrémité d'association privée pour armesStockées indique que seules les instances de Personnage auront un accès sur les armes stockées. On préfère en général une visibilité privée ou protégée, comme pour les attributs.

1.5 Résumé

Parmi toutes les informations que l'on peut placer sur une association, aucune n'est à proprement parler obligatoire. En général, on y place :

— les cardinalités,

— autant de noms que nécessaire à la compréhension de la sémantique de l'association : parfois un nom de rôle suffit par exemple. Il est souvent difficile de trouver des noms d'associations judicieux, ce sont souvent des groupes verbaux, et la langue française possède beaucoup moins de verbes que de noms,

— la navigabilité quand elle est décidée.

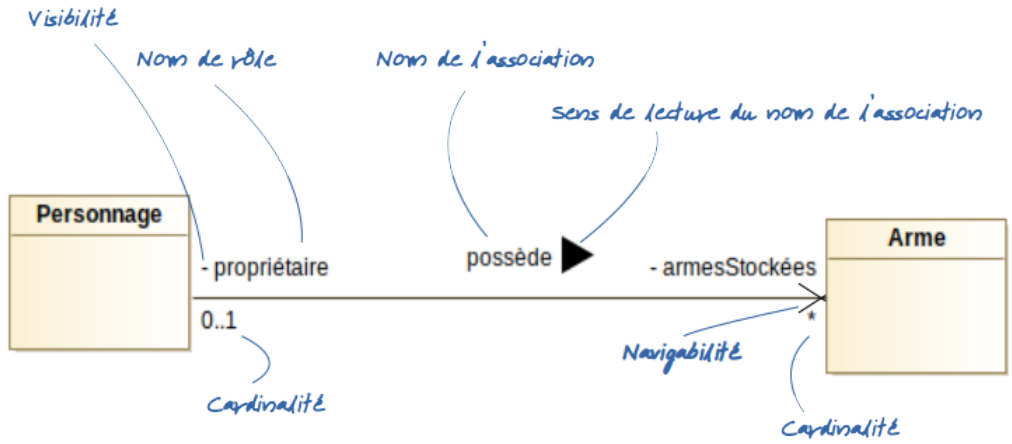
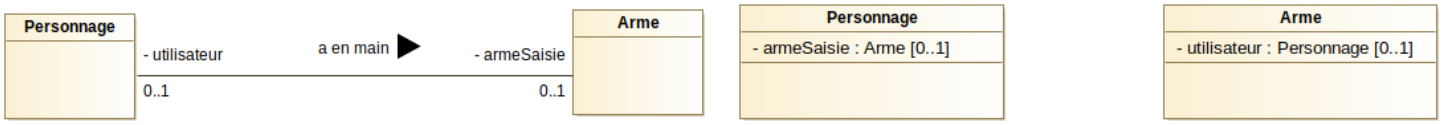


FIGURE 8 – Principaux éléments d'une association binaire

2 Associations binaires et attributs

Les extrémités d'associations et les attributs sont des notions très proches, et on peut légitimement se demander pourquoi introduire la notion d'association, plutôt que d'utiliser une paire d'attributs. Si nous reprenons notre exemple de personnage qui a en main une arme, on pourrait envisager une modélisation avec deux attributs, ainsi qu'illustré à la figure 9b.



(a) Version avec association

(b) Version avec paire d'attributs

FIGURE 9 – Associations vs paire d'attributs

Les associations sont à privilégier pour plusieurs raisons :

— Graphiquement, l'association relie les deux classes. Ainsi, visuellement, les deux classes sont mises en relation, tandis que la relation n'est pas immédiatement capturée quand on regarde des classes avec des attributs typés par d'autres classes.

— Quand on définit une paire d'attributs, les attributs évoluent en indépendance l'un de l'autre : la valeur de l'un n'influence pas la valeur de l'autre. Au contraire, avec une association, on relie deux extrémités d'associations, et on spécifie qu'elles sont réciproques l'une de l'autre. Ainsi, la version de la figure 9a indique qu'étant donné un personnage p, son armeSaisie a pour utilisateur p. Dans la version de la figure 9b, rien n'empêche un personnage p d'avoir son armeSaisie qui a pour utilisateur un autre utilisateur pbis différent de p. Autrement dit ronflex a en main excalibur, mais excalibur est dans la main de popoki ...

— les associations, comme nous le verrons par la suite, peuvent être augmentées d'informations supplémentaires encore plus difficilement intégrables dans des attributs.

Dans la suite de ce cours, nous utiliserons la convention usuelle suivant laquelle les attributs sont uniquement de type simple (entiers, flottants, booléens, ...). La conséquence logique est que l'on ne définira jamais un attribut de type complexe, on préférera alors utiliser une association.

3 Des associations binaires à leur implémentation en Java

La notion d'association n'existe pas en Java, elle n'existe d'ailleurs dans aucun langage de programmation généraliste. Traduire une association en Java doit donc se ramener aux concepts principaux de classes, attributs et méthodes.

Pour traduire une association en Java, on regarde tout d'abord la navigabilité : seules les extrémités navigables vont nécessiter une traduction directe. Nous étudions ici le cas le plus courant, dans lequel chaque extrémité navigable est traduite par un attribut placé dans la classe opposée à la flèche de navigation. Le cas moins courant de traduction au travers d'une nouvelle classe réifiant l'association sera étudié ultérieurement, lorsque nous aborderons les classes d'association.

3.1 Traduction d'une extrémité d'association navigable

Une extrémité d'association navigable peut se traduire par un attribut placé dans la classe opposée à l'extrémité, et typé par la classe contre la flèche de navigation. Par exemple, reprenons l'association entre les personnages et les armes saisies par les personnages, navigable de Personnage vers Arme. Cette association exprime le fait que l'on souhaite que la classe Personne puisse accéder à l'arme saisie. On vient donc naturellement placer un attribut stockant l'arme saisie dans la classe Personnage. On utilise le nom de rôle de l'extrémité comme nom d'attribut (avec d'éventuelles variations), on garde la visibilité, et la cardinalité, comme nous le détaillerons plus tard. Cela est illustré à la figure 10. On remarque que le nom de l'association n'est pas utilisé ici.

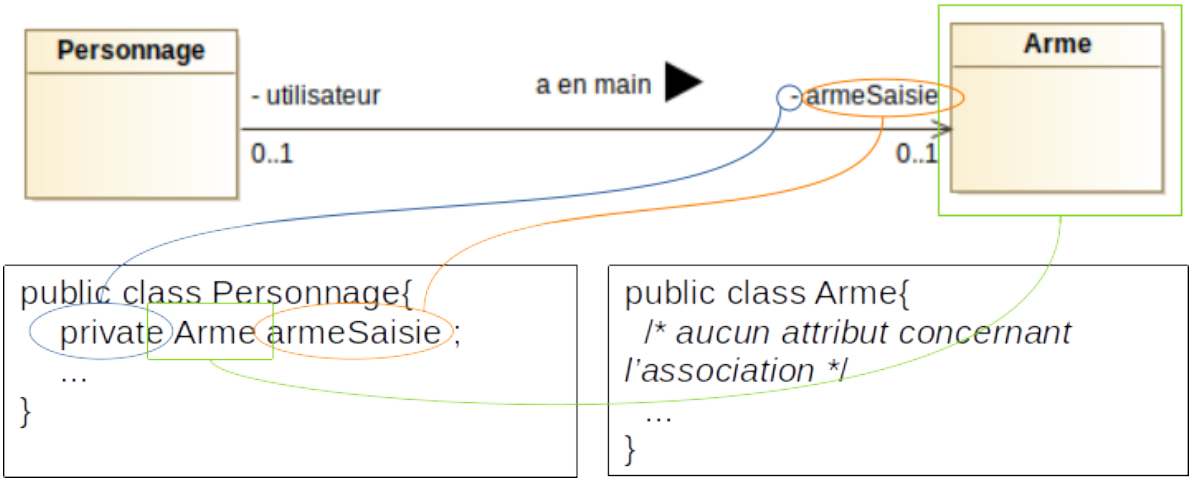


FIGURE 10 – Extrémité d'association navigable de cardinalité ≤ 1 et traduction Java

Cette traduction de l'extrémité d'association est ici satisfaisante du fait de la cardinalité 0..1 de l'extrémité d'association. Dans le cas où la cardinalité est par exemple * comme pour l'extrémité armesStockées de l'association "possède", on doit stocker dans l'attribut traduisant l'extrémité d'association non pas une instance d'Arme, mais plusieurs. On choisit alors une structure de données qui permet de stocker plusieurs armes, par exemple un tableau d'instances d'Arme, comme illustré à la figure 11.

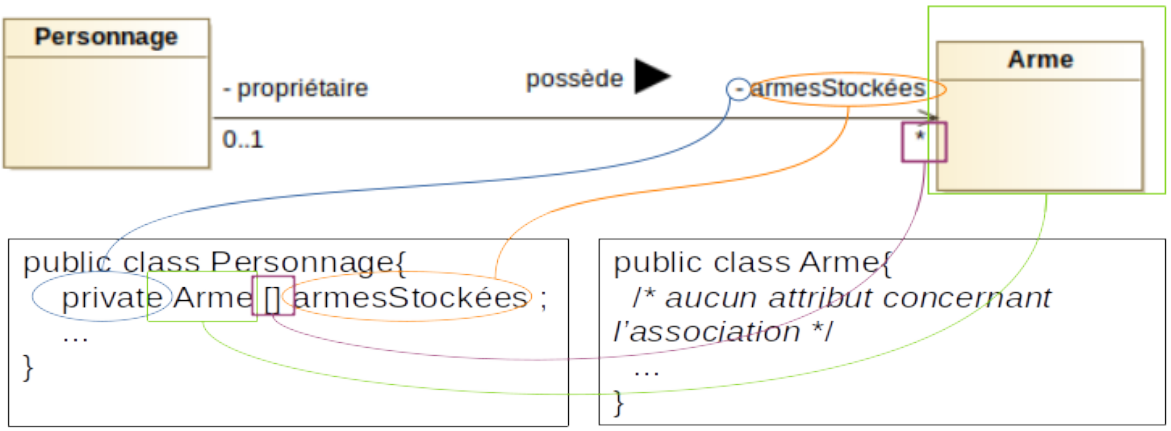


FIGURE 11 – Extrémité d'association navigable de cardinalité $*$ et traduction Java

Les tableaux Java ne sont pas toujours la structure la plus adaptée, surtout si on ne connaît pas le nombre maximal d'instances à stocker (ce qui est typiquement le cas lors d'une cardinalité *). Nous verrons à la section suivante quelques classes de l'API Java permettant de stocker plusieurs instances dans des collections avec des caractéristiques à la fois proches des tableaux et des listes. Dans les cas où les cardinalités sont "précises", par exemple 0..5 ou 2..3, on avise au cas par cas selon plusieurs critères : toutes

les instances jouent-elles le même rôle ? (auquel cas plusieurs associations auraient possiblement été une meilleure modélisation) Y a t-il beaucoup d'instances à stocker ? Y a t-il un intérêt à créer une structure de données dédiée ?

Pour conclure, dans le cas d'une association entre A et B, pour traduire une extrémité d'association accolée à une classe B, on étudie la cardinalité :

- 0..1 : on place un attribut de type B dans la classe opposée (A)
- 1 : on place un attribut de type B dans la classe opposée (A) et on veille à l'initialisation de cet attribut par un objet non null dès la construction d'un objet de type A.
- $i..j$ avec $j > 1$: on place un attribut permettant de stocker plusieurs instances de B dans la classe A. On choisit la structure de données adéquate en fonction de l'utilisation que l'on aura de l'attribut. En fonction de la cardinalité minimale modélisée i, on ajoute une ou plusieurs instances de B dans cette structure de données dès la construction d'un objet de type A.

3.2 Maintien des contraintes de cohérence induites par l'association

Les concepts d'association (UML) et d'attribut (UML ou Java) étant différents, une association n'est pas traduite intégralement par un ou plusieurs attributs.

3.2.1 Cas des associations bidirectionnelles

Dans le cas des associations bidirectionnelles, en suivant ce que nous avons vu jusqu'ici, la traduction en Java consiste en une paire d'attributs. Or nous avons vu qu'une paire d'attributs pouvait mener à des incohérences qui sont interdites par l'association : c'était par exemple le cas de ronflex qui avait en main une arme qui était tenue en main par popoki. La cohérence doit être maintenue en Java par le biais du contrôle des accès en écriture aux valeurs des attributs : ces accès doivent garantir la cohérence.

3.2.2 Respect des cardinalités

Dans le cas d'associations navigables dans un seul sens, seul le sens de navigation est traduit. Toutefois, il faut que l'implémentation fasse en sorte de maintenir la cardinalité de l'extrémité non navigable de l'association. Par exemple, pour l'arme prise en main, même si on ne stocke pas dans un attribut le personnage qui a l'arme en main, on doit garantir la cardinalité si elle est spécifiée, c'est-à-dire dans notre cas garantir qu'une même arme ne peut pas être dans la main de plusieurs personnages différents.

4 Collections d'objets en Java : tableaux et listes tabulaires

4.1 Brefs rappels sur les tableaux

Nous avons déjà vu les tableaux. Un tableau permet de ranger un nombre pré-déterminé k d'éléments (de type simple ou pas) dans une zone mémoire indexée par des entiers entre 0 et $k-1$. La capacité d'un tableau est non modifiable une fois le tableau créé. Quand nous avons utilisé les tableaux pour l'implémentation de clans, nous avons rencontré des lourdeurs d'utilisation :

- il a fallu fixer a priori une taille de clan arbitraire, et donc redimensionner le tableau quand il devient trop petit,
- lors de l'éjection d'un membre, soit il y avait un "trou" (une case null) dans le tableau, ce qui n'est pas pratique ensuite, soit il fallait déplacer vers la gauche tous les personnages.

4.2 Les listes tabulaires de l'API Java

L'API de Java propose (entre autres) de nombreuses classes permettant de représenter des collections d'objets (ensembles, piles, files, ...) et notamment des listes tabulaires, qui ont certaines des caractéristiques intéressantes des tableaux (accès direct à un élément indexé par un entier) ainsi que certaines de celles des listes (taille maximale non définie a priori, maintien de tous les éléments en début de liste (sans "trous")). Les listes tabulaires sont de plus munies de méthodes en permettant une manipulation aisée : recherche d'élément, suppression d'élément, etc.

Nous nous concentrons ici sur les ArrayList de cette API. Ces listes se basent sur une structure (cachée à l'utilisateur) de tableau, et en fournit une surcouche en rendant l'utilisation facile (par exemple : le redimensionnement quand le tableau qui sous-tend la liste est plein).

La classe ArrayList se trouve dans le paquetage java.util, donc pour l'utiliser, il faut soit préfixer le nom de la classe par le nom du package (`java.util.ArrayList`) soit l'importer par la ligne :

```
import java.util.ArrayList;
```

4.2.1 Une collection générique

Comme toutes les collections de l'API Java, les ArrayList sont implémentées par une classe dite *générique*. Cela signifie que la définition de la classe ArrayList mentionne le type des éléments qui y sont stockés, sous forme d'un paramètre : le paramètre de généricité. Ainsi, ce qui est décrit n'est pas la classe ArrayList, mais la classe ArrayList<E> (liste tabulaires d'éléments de

type E), où E est ici le paramètre générique. Pour utiliser une ArrayList, il faut donc donner une valeur à ce paramètre : on lui donne pour valeur un type objet existant, comme String, Personnage, Arme, etc. Attention, on ne peut pas directement utiliser de type primitif (comme int ou boolean). Si on peut disposer d'une ArrayList d'un type primitif, on doit passer par le type enveloppe de ce type primitif (par exemple Integer ou Boolean).

4.2.2 Déclaration et création de liste

Avant d'être utilisées, les ArrayList doivent être au préalable déclarées et créées. Nous allons ici reprendre l'exemple des clans de personnages que nous avions vus en utilisant des tableaux.

Programme 1 : *Déclaration et création de la liste des membres dans la classe Clan*

```
public class Clan {
        private String nom;
        ...
        protected ArrayList<Personnage> membres; // si initialisation ici rajouter : =new
        ↪  ArrayList<Personnage>()

        public Clan(String nom) {
                this.nom = nom;
                membres=new ArrayList<Personnage>(); // construction de la liste des membres, pouvant
                ↪  alternativement être réalisée lors de la déclaration
        }
        ...
}
```

4.2.3 Illustration de quelques opérations utiles

La classe ArrayList s'utilise comme une classe normale, c'est-à-dire en invoquant les méthodes de la classe ArrayList sur l'instance d'ArrayList, afin de réaliser différentes actions comme ajouter un élément, en retirer un, connaître la taille de la liste, etc. Nous allons illustrer dans la suite certaines des méthodes les plus utilisées de la classe ArrayList<E> :

- `void add(E obj)`. Ajoute l'objet obj à la fin de la liste (à droite).
- `boolean contains(E obj)`. Retourne vrai si et seulement si obj est dans la liste.
- `E get(int index)`. Retourne l'objet placé en position index dans la liste.
- `boolean isEmpty()`. Retourne vrai si et seulement si la liste n'a aucun élément (est vide).
- `int size()`. Retourne la taille de la liste, c'est-à-dire le nombre d'éléments qu'elle contient.
- `E remove(int index)`. Supprime l'objet en position index et le retourne.
- `boolean remove(Object o)`. Supprime la première occurrence de o rencontrée (laisse la liste inchangée si l'objet o n'est rencontré, et retourne alors faux).

Les codes donnés ci-dessous sont supposés placés dans la classe Clan.

Ajout d'un élément La méthode add permet d'ajouter un élément (passé en paramètre) à la liste. Cet élément est alors ajouté à la fin de la liste. Ainsi, la méthode ajoutMembre de la classe Clan peut se limiter à l'appel de cette méthode, comme illustré ci-dessous. Toutefois, il faut être conscient de deux points :

- la méthode add ne vérifie pas si l'élément à ajouter fait doublon (une liste n'est pas un ensemble),
- la méthode add ajoute systématiquement l'objet reçu en paramètre, même s'il est null.

Programme 2 : *Illustration de la méthode add*

```
public void ajoutMembre(Personnage p) {
        membres.add(p); // il faudrait vérifier que p n'est pas déjà dans le clan ...
}
```

Test d'appartenance d'un élément Pour éviter les doublons dans le clan, nous allons vérifier, avant d'ajouter un personnage, qu'il n'y est pas déjà. Nous en profitons pour nous assurer également qu'il n'est pas null.

Programme 3 : *Illustration de la méthode contains*

```
public void ajoutMembre(Personnage p) {
        if (!membres.contains(p)&&p!=null) {
```

```
                membres.add(p);
        }
}
```

Notons que la méthode contains indique si l'élément spécifié se trouve dans la liste, au sens de la comparaison d'objets avec la méthode equals. Ici, nous manipulons des listes de personnages, donc le test d'égalité equals sera celui de la classe Personnage. Si une méthode equals est définie dans la classe Personnage, c'est cette méthode qui sera appelée, sinon c'est la méthode equals de Object qui sera utilisée (rappelons que cette méthode se ramène à un == sauf pour le cas d'instances null).

Accès au énième élément La méthode get permet un accès au énième [1] Nous illustrons l'utilisation de cette méthode avec la méthode qui permet d'éjecter (aléatoirement) un membre du clan, qu'elle retourne. Cette méthode choisit aléatoirement un entier i correspondant à un index dans la liste des membres, récupère le personnage en position i, l'enlève et le retourne.

Enlever le énième élément La méthode remove est surchargée. L'un des surcharges permet d'enlever le énième élément de la liste. Nous illustrons également l'utilisation de cette méthode avec la méthode qui permet d'éjecter (aléatoirement) un membre du clan. Après avoir choisi aléatoirement un entier i correspondant à un index dans la liste des membres et stocké le personnage correspondant (avant de l'enlever !), elle enlève le membre en position i en appelant la méthode remove. Un décalage à gauche des éléments à droite de l'élément retiré (les éléments d'index supérieur à i) est effectué pour ne pas "laisser de trou".

Programme 4 : *Illustration des méthodes get et remove(int i)*

```
public Personnage ejecterUnMembre() {
        Personnage resultat=null;
        int taille=taille();
        if (taille()!=0) {
                Random rand=new Random(); // java.util.Random
                int i=rand.nextInt(taille);
                resultat=membres.get(i);
                membres.remove(i);
        }
        return resultat;
}
```

Enlever un élément particulier L'autre surcharge de la méthode remove prend en paramètre l'élément à retirer. Cette méthode cherche l'élément à retirer dans la liste (en faisant des comparaisons avec des equals). Lorsqu'une instance est trouvée (notons qu'en cas de doublons, il pourrait y en avoir plusieurs), elle est retirée. S'il y avait plusieurs autres instances de l'objet à retirer, elles sont laissées dans la liste. Si aucune instance n'est trouvée, la méthode retourne faux (et vrai sinon). Comme pour l'autre surcharge, un décalage à gauche des éléments à droite de l'élément retiré est effectué.

Programme 5 : *Illustration des méthodes get et remove(E obj)*

```
public Personnage ejecterUnMembre() {
        Personnage resultat=null;
        int taille=taille();
        if (taille()!=0) {
                Random rand=new Random(); // java.util.Random
                int i=rand.nextInt(taille);
                resultat=membres.get(i);
                membres.remove(resultat);
        }
        return resultat;
}
```

Dans le cas précis de la méthode ejecterUnMembre, il est plus judicieux d'utiliser la surcharge qui enlève le i-ème élément plutôt que celle qui enlève un élément donné, cela évite une recherche de l'élément avant la suppression ...

Accès à la taille de la liste La taille d'une liste est son nombre d'éléments, et est donné par la méthode size.

1. Saviez-vous qu'on pouvait écrire énième, $n^{ième}$ ou encore n-ième ? Mais qu'on ne devait pas écrire $2^{ème}$ mais 2^{e} ? Bizarre ...

Programme 6 : *Illustration de la méthode size*

```
public int taille() {
        return membres.size();
}
```

Test de vacuité Les test de vacuité permet de déterminer si la liste est vide. Il est réalisé par la méthode isEmpty. Dans l'exemple ci-dessous, nous remplaçons le test de nullité de la taille par l'appel à la méthode isEmpty.

Programme 7 : *Illustration de la méthode isEmpty*

```
public Personnage ejecterUnMembre() {
        Personnage resultat=null;
        int taille=taille();
        if (!membres.isEmpty()) {
                Random rand=new Random(); // java.util.Random
                int i=rand.nextInt(taille);
                resultat=membres.get(i);
                membres.remove(resultat);// ou membres.remove(i);
        }
        return resultat;
}
```

4.2.4 Parcourir des collections

Les collections et les tableaux se parcourent classiquement :
— à l'aide d'itérateurs, mais nous ne les aborderons pas dans ce cours
— à l'aide de boucles (boucles for ou while)
— moins naturellement à l'aide de méthodes récursives.

Notons que depuis la version 1.5 de Java, une boucle "for each" a été introduite, qui permet de parcourir une collection ou un tableau sans indice de boucle. Ainsi, le code suivant permet d'afficher le nom de tous les membres du clan.

Programme 8 : *Illustration de la boucle "for each"*

```
public void afficherNomsMembres() {
        for (Personnage p:membres) {
                System.out.println(p.getNom());
        }
}
```

Cette boucle se lit : "pour chaque personnage p trouvé dans la collection membres, faire". La variable p va successivement référencer chacun des éléments de la liste membres.

4.2.5 Quelques précisions de taille ...

On peut constater la présence d'un constructeur paramétré par un entier dans la classe ArrayList. Ce constructeur permet de créer une liste dont la taille de la structure (interne) de tableau est prise en paramètre. Toutefois, il ne s'agit en aucun cas d'une taille maximale pour la liste : si cette taille est atteinte et dépassée, la structure interne est redimensionnée (remplacée par une structure plus grande) de manière transparente (modulo le surcoût induit).

Il n'est pas possible de fixer une taille maximale à la liste.

Concernant la taille de la structure interne utilisée, l'utilisateur de la liste ne la maîtrise pas, ou peu : la seule façon de l'influencer est lors de la construction (comme vu précédemment) et éventuellement via la méthode ensureCapacity. La taille de la structure interne est appelée capacity, on peut s'assurer (par appel à cette méthode) que la capacité en terme de nombre d'éléments pouvant être accueillis a bien une certaine valeur passée en paramètre. L'appel à cette méthode peut s'avérer utile dans le cas où un grand nombre d'éléments va être ajouté à la liste, pour éviter de multiples redimensionnements.

Enfin, toujours à propos de taille de la liste, nous avons dit que la taille de la liste était le nombre d'éléments qui y étaient présents. Ce nombre inclut les éléments null ; aussi si on a explicitement ajouté null à la liste, la taille est augmentée de 1 suite à cette opération puisque l'élément null est effectivement ajouté à la liste.

4.3 Quels accesseurs pour les collections ?

Nous avons vu que les associations pouvaient se traduire en Java avec des attributs typés par des collections d'objets. Faut-il faire des accesseurs "classiques" sur ces attributs ? Par exemple, dans l'exemple des clans, faut-il ajouter les accesseurs `public ArrayList<Personnage> getMembres(){return membres;}` et `public void setMembres(ArrayList<Personnage> membres){this.membres=membres;}` ?

Concernant l'accesseur en écriture, on peut constater qu'il ne joue pas le rôle que l'on attend d'un accesseur en écriture, qui doit en effet contrôler la validité des valeurs reçues en paramètre. Dans le cas des clans, la liste des membres reçue est construite extérieurement à la classe Clan, et peut par exemple contenir des doublons, ce qui n'est pas souhaité ici. On évitera donc ce genre d'accesseur en écriture, et on se contente en général de méthode d'ajout et de suppression d'éléments. L'accesseur en lecture n'est guère plus satisfaisant : il permet effectivement de retourner vers l'appelant une référence vers la liste ; l'appelant aura par conséquent tout le loisir de faire ce qu'il souhaite avec la liste (enlever ou ajouter des éléments par exemple) de manière parfaitement non contrôlée par la classe Clan (on pensera bien que l'on retourne lors d'un `return membres;` non pas une copie de la liste mais une référence vers la liste). Si vraiment un tel accesseur est intéressant, on veillera à retourner plutôt une version immuable (non modifiable) de la liste, en utilisant la méthode de classe Collections.unmodifiableList qui retourne une version non modifiable de la liste (en fait qui encapsule la liste dans un objet qui va en bloquer les accès en écriture).

Programme 9 : *Un accesseur en lecture pour une liste, qui retourne une version immuable de la liste*

```
public List<Personnage> getMembres(){
        return Collections.unmodifiableList(membres);
}
```

Notez que l'appelant continuera de voir les modifications qui seront faites ultérieurement sur la liste, mais ne pourra pas directement y faire de modifications.