

Feuille de travaux dirigés n° 10

Chapitre 10 - Éléments de Conception à objets



Dans cette série d'exercices, nous nous focalisons sur le développement de petits exemples, de la conception objet à l'implantation dans un langage à objets. Dans un premier temps, nous rappelons les éléments de conception détaillée abordés dans les TD des chapitres 2 et 3.

1 Rappels

1.1 De la modélisation à la programmation à Objets

Dans cette section, nous nous intéressons à un problème spécifique de conception détaillée : la représentation des associations UML en programmation. On ne prend en compte ici que les associations de base : pas de contrainte d'ordre, pas de qualifieur, pas de propriétés, pas de classe-association. Ce travail est revu de manière plus globale dans le chapitre 10, à travers un exemple dans la fiche *fiche2_nim.pdf* que nous vous invitons à consulter. Nous l'illustrons par les cas généraux de la figure 1.

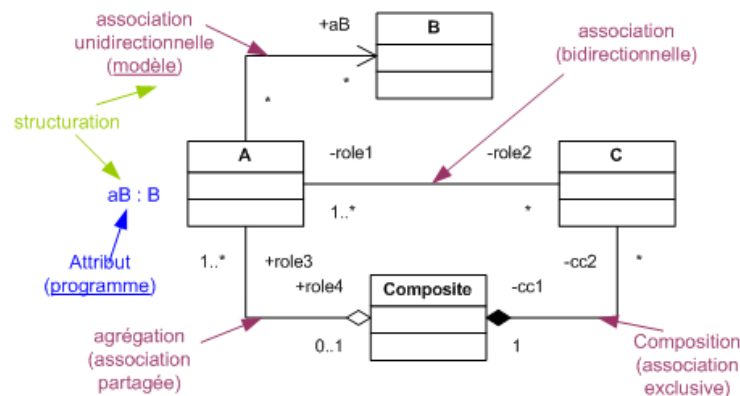


Figure 1 : Variantes d'une association UML

En programmation à objets, une association s'implante habituellement avec des pointeurs ou des références. On pose les règles de base suivantes :

1. Une association unidirectionnelle est représentée par un attribut (en utilisant le nom du rôle) dans la classe d'où part la flèche.
2. Une association bidirectionnelle sera représentée par deux associations unidirectionnelles.
3. L'agrégation est toujours traitée comme une association simple, en programmation (quel que soit le sens de navigation).
4. La composition est traitée *ici* comme une association simple (quel que soit le sens de navigation). En fait des règles plus strictes s'appliquent, et le codage varie d'un langage à l'autre.

unidirectionnel	bidirectionnel
<pre>class A { public B* aB; ... } class B { ... }</pre>	<pre>class A { private C* role2; ... } class C { private A* role1; ... }</pre>

Le signe * signifie que le type va dépendre de la cardinalité de l'extrémité d'association considéré.

1. Par défaut, avec la cardinalité 0..*, il s'agit d'un ensemble, conformément à OCL¹. En C++ on utilise des `templates`, en Smalltalk des collections (hétérogènes par défaut). En Java, on distinguera deux traductions, selon la version de Java utilisée :

- (a) avant Java 5.0, la généricité n'est pas autorisée, on utilise des tableaux.

```
class Composite {
  private C[] cc2;
  ...
}
```

Cela pose des

- (b) après Java 5.0 la généricité est autorisée, on préconise les collections génériques² de Java. Le type `Set` est une interface, une classe qui l'implémente est `HashSet`.

```
class Composite {
  private HashSet<C> cc2;
  ...
}
```

Cette méthode est applicable quelle que soit la cardinalité. Il faut noter que le cas (a) implique de nombreux `transtypage` (cast, coercition) ce qui alourdit notablement le code.

2. On peut simplifier dans deux situations :

- (a) Cardinalité minimale et maximale de 1 : un attribut typé par la classe.

```
class C {
  private Composite cc1;
  ...
}
```

- (b) Cardinalité minimale de 0 et maximale de 1 : un attribut typé par la classe en acceptant la valeur `null` qui correspond au cas 0. On le note par un commentaire car l'union de types³ n'est pas autorisée en Java ou C++; elle est implicite en Smalltalk.

```
class A {
  public Composite role4; // CompositeOrNull
  ...
}
```

2 De la modélisation à la programmation à Objets

Dans cette section, nous nous intéressons à un problème spécifique de conception détaillée de la relation de spécialisation : la représentation des spécialisations UML en relation d'héritage des langages de programmation.

Les problèmes associés sont

1. traitement de l'héritage multiple dans les langages qui n'autorisent que l'héritage simple.
 - En C++ ou Eiffel, l'héritage multiple est autorisé. On gère les conflits par des redéfinitions ou des renommages.

1. Avec une contrainte d'ordre `ordered`, on utilisera une séquence (ou liste). Pour une association qualifiée, on utilisera un dictionnaire (`HashMap` en Java).

2. voir les détails dans le chapitre 6.

3. Comme en CAML.

- En héritage simple, on choisit un axe principal de spécialisation puis
 - en Java, on définit des interfaces pour l'héritage secondaires et on les implante : on cumule les avantages du sous-typage sans l'inconvénient de gestion de l'héritage multiple.
 - En Smalltalk, on doit faire du copier coller !
- 2. Redéfinition éventuelle / renommage de propriétés.

3 Exemples à lire

Lire les fiches

- Exemple du jeu de Nim : `fiche1_nim.pdf`
- Tennis : `fiche10_tennis.pdf`

Exercice 10.1 (Exemples)

Pour les deux exemples,

- Tester les applications.
- Vérifier la concordance entre le programme

Exercice 10.2 (Conception détaillée, Programmation)

Reprendre l'exemple du tennis,

- Concevoir en détail le développement de la gestion de tennis en Java.
- Programmer l'application en Java.
- Vérifier la concordance entre le programme et sa conception.

4 Cas d'étude à traiter

Pour chaque cas proposé on pourra

1. Concevoir sous forme d'un diagramme UML les classes de ce système.
2. Concevoir les classes en fonction du langage de programmation choisi (Java, C++, Eiffel, Smalltalk).
3. Programmer l'application dans le langage cible.

Exercice 10.3 (Hôpital)

Reprendre l'énoncé de l'exercice 1.4 **Hôpital** de la section 1 du TD n° 1.

Exercice 10.4 (Compte)

Reprendre l'énoncé de l'exercice 1.5 **Compte** de la section 1 du TD n° 1.

Exercice 10.5 (Feux)

Reprendre l'énoncé de l'exercice 1.6 **Feux** de la section 1 du TD n° 1.

Exercice 10.6 (Géométrie)

Reprendre l'énoncé de l'exercice 1.2 de **Géométrie** de la section 1 du TD n° 1.