Les associations

Introduction

Au cours d'UML, vous avez vu qu'il existait plusieurs types d'association. Le but de ce chapitre est de montrer une façon de les implémenter. Les deux problèmes qui se posent pour l'implémentation sont la gestion de la multiplicité (unaire ou multiple) et la navigation (uni ou bi directionnalité) de l'association.

Dans un premier temps, on va voir l'implémentation de tout ce qui se rapporte à la multiplicité en étant indépendant de la navigation. Ensuite, on verra comment écrire les méthodes dont l'implémentation dépendra de la navigation de l'association. Pour ce faire, on partira d'un exemple de gestion des commandes de clients.

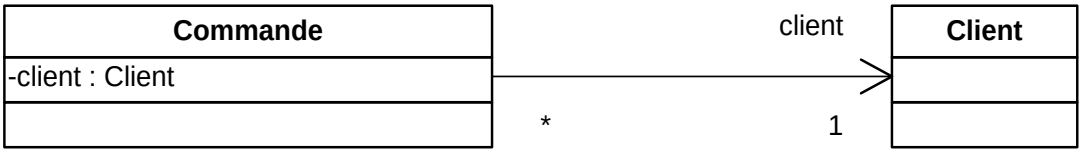
Rappelons que la **navigation** représente la **responsabilité** de l'association :

- les deux classes sont responsables de l'association, alors l'association est **bidirectionnelle**.
- une seule classe est responsable de l'association, alors l'association est **unidirectionnelle**.

Rôle unaire (0-1)

Interface

Considérons le diagramme de classes[1] suivant :



Le rôle unaire est celui joué par le `Client` dans l'association présentée ici.

Voici les méthodes à implémenter dans la classe `Commande` pour gérer l'association sont :

- `boolean enregistrerClient(Client client);`
- `boolean supprimerClient();`
- `Client getClient();`
- `boolean maximumClientAtteint();`
- `boolean minimumClientGaranti();`

Parmi ces méthodes, seule l'implémentation des deux premières dépend de la navigation.

Gestion de la multiplicité

Dans l'exemple, la multiplicité minimum et maximum du `Client` est de 1.

Lorsque la multiplicité maximale est de 1, il suffit de conserver une seule référence de `Client` dans `Commande`.

Lorsque la multiplicité minimum est de 1, cela signifie que l'association doit exister. Pour gérer cette multiplicité minimum de 1 (obligation d'attribut), il y plusieurs solutions possibles :

- créer un `Client` dans le constructeur de `Commande` mais cela nécessite des paramètres supplémentaires (BOF !). De plus, ce n'est pas toujours possible et logique...
- passer un paramètre `Client` à ce constructeur. De nouveau, ce n'est pas toujours possible.
- ne rien passer au constructeur et laisser la multiplicité minimum dans un état incorrect qui sera signalé à l'appel de `getClient`.

Il est parfois obligatoire d'adopter la troisième solution. Dans ce cas, les deux dernières méthodes de l'interface permettent de gérer cette multiplicité :

- `boolean maximumClientAtteint()` pour la gestion de la multiplicité maximum
- `boolean minimumClientGaranti()` pour la gestion de la multiplicité minimum

[1] Constatons que les diagrammes de classes de ce présent document sont des diagrammes d'implémentation qui montrent comment les associations sont implémentées.

Implémentation

Pour l'implémentation, on vérifie la multiplicité maximum lors de l'enregistrement. On empêche ainsi d'enregistrer un client s'il y en a déjà un. Par contre, on ne vérifie pas la multiplicité minimum lors de la suppression. Par conséquent, si on désire changer de client, il faudra d'abord supprimer le client actuel avant d'enregistrer le nouveau.

Si la multiplicité minimum n'est pas respectée lors de l'appel du getter, on le signalera en lançant une exception.

multiplicité minimum est de 1

```
public boolean minimumClientGaranti(){
	return this.client != null;
}
public Client getClient() throws
MinimumMultiplicityException {
	if (! minimumClientGaranti())
		throw new
	MinimumMultiplicityException();
	return client;
}
public boolean maximumClientAtteint() {
	return this.client != null;
}
```

multiplicité minimum est de 0

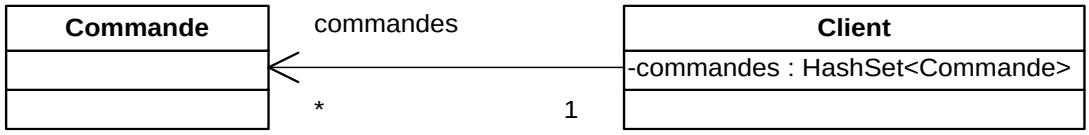
```
public boolean minimumClientGaranti() {
	return true;
}

public Client getClient() {
	return client;
}
public boolean maximumClientAtteint() {
	return this.client != null;
}
```

Rôle multiple (*)

Interface

Maintenant, le magasin désire pouvoir retrouver toutes les commandes d'un client. Ici, c'est le client qui va garder une « liste » de commandes. Dans l'exemple ci-dessous, on a choisi de garder cette liste dans un `HashSet`.



Voici la liste des méthodes qu'on implémente dans `Client` :

- `public boolean ajouterCommande(Commande c);`
- `public boolean supprimerCommande(Commande c);`
- `public boolean contientCommande(Commande c);`
- `public Iterator<Commande> commandes() throws …;`
- `public int nombreDeCommandes();`
- `public boolean maximumCommandesAtteint();`
- `public boolean minimumCommandesGaranti();`

Seule l'implémentation des deux premières méthodes dépend de la navigation.

Gestion de la multiplicité

S'il n'y a pas de limite maximale pour le nombre de commandes, il n'y a pas de problème. Mais si le nombre maximum de commandes est limité à une certaine valeur MAX, on doit tester cela à l'ajout. Plus précisément, on refuse l'ajout si le maximum autorisé est atteint.
Concernant la multiplicité minimum, si on autorise 0, il n'y a pas de problème. Par contre, si le nombre minimum de commandes est limité à une certaine valeur MIN (!=0), on accepte que cette multiplicité reste dans un état incorrect qui sera signalé lors d'appel de la méthode renvoyant un `Iterator`.

Implémentation

Pour l'implémentation, on refuse l'ajout si le maximum autorisé est atteint. Par contre, on ne teste pas le minimum lors de la suppression c.à.d. qu'on accepte de supprimer même si cela peut mener à un état incorrect. Si le maximum est atteint, on devra donc faire une suppression avant de pouvoir en rajouter un autre.

Exemple :

- Implémentation de `contientCommande` :

```
public boolean contientCommande(Commande c) {
      return commandes.contains(c);
}
```

- Implémentation de l'`Iterator` si la multiplicité minimum est de MIN (!=0) :

```
public Iterator<Commande> commandes() throws
                      MinimumMultiplicityException  {
      if (! minimumCommandesGaranti())
            throw new MinimumMultiplicityException();
      return Collections.unmodifiableSet(commandes).iterator();
}
```

- Implémentation de l'`Iterator` si la multiplicité minimum est de 0 :

```
public Iterator<Commande> commandes(){
      return Collections.unmodifiableSet(commandes).iterator();
}
```

On appelle la méthode `iterator()` sur `Collections.unmodifiableSet(commandes)` pour interdire la suppression (`remove()`) via l'itérateur.

Si l'association est bidirectionnelle, on est obligé de faire comme cela afin de conserver la cohérence des données. Par contre, si l'association est unidirectionnelle, on doit décider si on accepte ou non la suppression via l'itérateur. Si on l'accepte, on appellera la méthode `iterator()` directement sur le HashSet `commandes` (`return commandes.iterator();`).

- Implémentation de `nombreDeCommandes` :

```
public int nombreDeCommandes(){
      return commandes.size();
}
```

- Implémentation de `maximumCommandesAtteint` si la multiplicité maximum est de MAX :

```
public boolean maximumCommandesAtteint() {
      return commandes.size() >= MAX;
}
```

- Implémentation de `maximumCommandesAtteint` si la multiplicité maximum est * :

```
public boolean maximumCommandesAtteint() {
      return false;
}
```

- Implémentation de `minimumCommandesGaranti` si la multiplicité minimum est de MIN (!= 0)

```
public boolean minimumCommandesGaranti() {
      return commandes.size() >= MIN;
}
```

- Implémentation de `minimumCommandesGaranti` si la multiplicité minimum est de 0 :

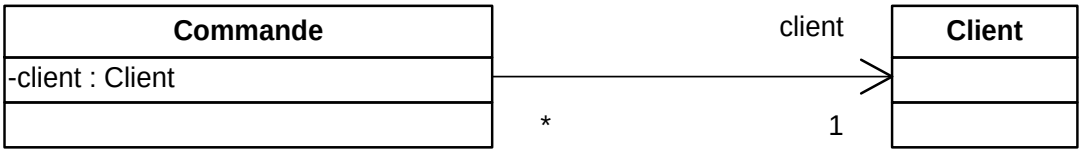
```
public boolean minimumCommandesGaranti() {
      return true;
}
```

Association unidirectionnelle

On va maintenant donner l'implémentation des méthodes qui dépendent de la directionnalité dans le cas où l'association est unidirectionnelle.

Rôle unaire

On est dans le cas où la responsabilité de l'association est donnée à la commande. Plus précisément, la commande garde le client mais le client, lui, ne garde aucune référence à ses commandes



- Implémenter `enregistrerClient` :

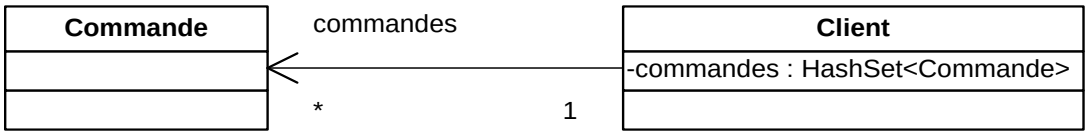
```
public boolean enregistrerClient(Client client) throws
                          ArgumentInvalideException {
      // tester la validité du paramètre si nécessaire
      if (client == null)
            throw new ArgumentInvalideException("…");
      // tester s'il n'y a rien à faire
      if (this.client == client) return false;
      // tester si le maximum est atteint
      if (maximumClientAtteint()) return false;
      // faire le travail
      this.client = client;
      return true;
}
```

- Implémenter `supprimerClient` :

```
public boolean supprimerClient() {
      // tester s'il n'y a rien à faire
```

```
      if (this.client == null) return false;
      // faire le travail
      this.client = null;
      return true;
}
```

Rôle multiple



Ici, on suppose que c'est le client qui est responsable de l'association. C'est donc lui qui garde la référence vers ses commandes tandis que les commandes ne gardent aucune référence au client.

- Implémentation d'`ajouterCommande` :

```
public boolean ajouterCommande(Commande c){
      // tester la validité du paramètre si nécessaire ...
      // tester s'il n'y a rien à faire
      if (this.contientCommande(c)) return false;
      // tester si le maximum est atteint
      if (this.maximumCommandesAtteint())
            return false;
      // faire le travail
      commandes.add(c);
      return true;
}
```

- Implémentation de `supprimerCommande` :

```
public boolean supprimerCommande(Commande c) {
      // tester le paramètre ...
      // tester s'il n'y a rien à faire
      if (! this.contientCommande(c))
            return false;
       // faire le travail
       commandes.remove(c);
       return true;
      }
```

Généralisation

Quand on implémente une association unidirectionnelle, on peut généraliser la démarche à suivre pour effectuer les ajouts et les suppressions :

- Ajout (ou enregistrement) :
    1. vérifier la validité du paramètre : s'il n'est pas valide on lance une exception ;

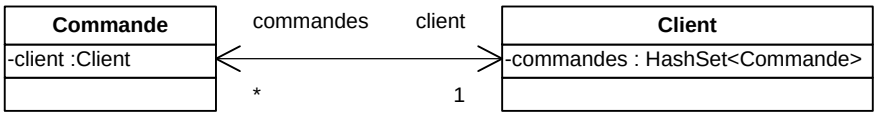
2. voir s'il y a réellement quelque chose à faire : si le travail a déjà été fait, on arrête la méthode ;
3. voir si on peut l'ajouter : si le maximum autorisé a déjà été atteint ou si on ne peut pas effectuer l'ajout à cause d'une autre contrainte liée au contexte (par exemple, on ne peut pas ajouter une commande au client s'il a une commande en retard de payement), on arrête la méthode ;
4. Effectuer l'Ajout (ou l'enregistrement).

- Suppression :
  1. S'il y a un paramètre, tester sa validité : s'il n'est pas valide on lance une exception ;
  2. Voir s'il y a réellement quelque chose à faire : s'il n'y a rien à faire, on arrête la méthode ;
  3. Effectuer la suppression

Association bidirectionnelle

Problématique

Ici, on est dans le cas où la commande garde une référence à son client et où le client garde la référence de ses commandes. Il faut évidemment que les rôles soient inverses l'un de l'autre. Cela veut dire que si une commande référence un client, il faut aussi que ce client référencie cette commande (et réciproquement). C'est au programmeur de faire en sorte que la cohérence des données soit maintenue.



On pourrait être tenté d'implémenter cela comme suit :

- Dans `Commande` :

```
public boolean enregistrerClient(Client client) {
       this.client = client;
       client.ajouterCommande(this); // pour maintenir la cohérence
      return true;
}
```

- Dans `Client` :

```
public boolean ajouterCommande(Commande c) {
      this.commandes.add(c);
      c.enregistrerClient(this); // pour maintenir la cohérence
      return true;
 }
```

Mais cela engendrait une boucle infinie → A NE PAS FAIRE !

Ci-dessous, nous allons montrer comment implémenter cela en évitant la boucle infinie.

Associations bidirectionnelles One to Many

Voici comment implémenter les méthodes dans la classe Commande :

- Implémentation d'`enregistrerClient` :

```
   public boolean enregistrerClient(Client client) {
// tester le paramètre ...
Util.checkObject(client);
// tester s'il n'y a rien à faire
if (this.client == client) return false;
// tester si le maximum est atteint de ce coté-ci
if (maximumClientAtteint()) return false;
// vérifier que la commande pourra, si besoin, être ajoutée au client
if (client.maximumCommandesAtteint()&&!client.contientCommande(this))
      return false;
// faire le travail
this.client = client;
// demander de faire le travail de l'autre coté
client.ajouterCommande(this);
return true;
   }
```

En terminant prématurément s'il n'y a rien à faire, on évite la boucle infinie

- Implémentation de `supprimerClient` :

```
   public boolean supprimerClient() {
// tester s'il n'y a rien à faire
if (this.client == null) return false;
// faire le travail de ce coté-ci
Client ex = this.client
this.client = null;
// demander de faire le travail de l'autre coté
ex.supprimerCommande(this);
return true;
   }
```

Associations bidirectionnelles Many to One

Voici comment implémenter les méthodes dans la classe Client :

- Implémentation d'`ajouterCommande` :

```
   public boolean ajouterCommande(Commande c) {

// tester le paramètre ...
```

```
Util.checkObject(c);
// tester s'il n'y a rien à faire
if (this.contientCommande(c)) return false;
// tester si le maximum est atteint de ce coté
if (this.maximumCommandesAtteint()) return false;
// vérifier qu'on peut si besoin enregistrer ce client à la commande
try {
     if (c.maximumClientAtteint() && c.getClient() != this)
          return false;
} catch (MinimumMultiplicityException e) {
     // l'exception ne devrait pas être lancée
     throw new InternalError();
}
// faire le travail
commandes.add(c) ;
// demander de faire le travail de l'autre coté
c.enregistrerClient(this);
return true;
   }
```

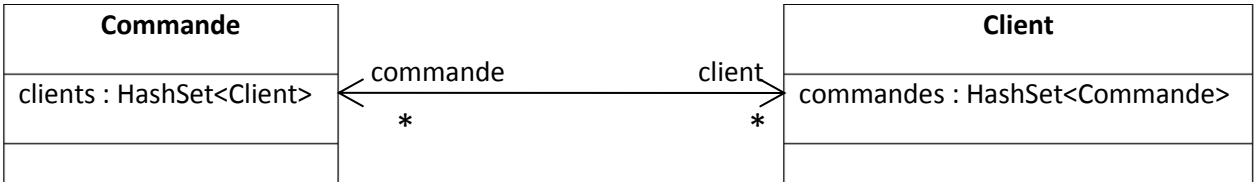
- Implémentation de `supprimerCommande` :

```
public boolean supprimerCommande(Commande c) {

// tester le paramètre ...
// tester s'il n'y a rien à faire
if (! this.contientCommande(c))
     return false;
// faire le travail
commandes.remove(c);
// demander de faire le travail de l'autre coté
c.supprimerClient();
return true;
   }
```

Associations bidirectionnelles Many to Many

On suppose maintenant qu'il peut y avoir plusieurs clients pour une commande. De plus, les commandes gardent les références de leurs clients et les clients gardent les références de leurs commandes



Comme la commande peut avoir plusieurs clients, on a une nouvelle interface du coté `Commande` :

- `public boolean ajouterClient(Client c);`
- `public boolean supprimerClient(Client c);`
- `public boolean contientClient(Client c);`
- `public Iterator<Client> clients() throws …;`
- `public int nombreDeClients();`
- `public boolean maximumClientsAtteint();`
- `public boolean minimumClientsGaranti();`

- Implémentation d'`ajouterCommande` :

```
public boolean ajouterCommande(Commande c) {

// tester le paramètre …
// tester s'il n'y a rien à faire
if (this.contientCommande(c)) return false;
// tester si le maximum est atteint de ce coté
if (this.maximumCommandesAtteint())
      return false;
// vérifier qu'on peut, si besoin, ajouter le client à la
//commande
if (c.maximumClientsAtteint() && !c.contientClient(this))
     return false;
// faire le travail
commandes.add(c);
// demander de faire le travail de l'autre coté
c.ajouterClient(this);
return true;
}
```

- Implémentation de `supprimerCommande` :

```
public boolean supprimerCommande(Commande c) {

// tester le paramètre …
// tester s'il n'y a rien à faire
if (! this.contientCommande(c))
     return false;
// faire le travail
commandes.remove(c);
// demander de faire le travail de l'autre coté
c.supprimerClient(this);
return true;
}
```

Généralisation

On peut généraliser la démarche suivie pour les associations unidirectionnelles pour effectuer les ajouts et les suppressions lorsque l'association est bidirectionnelle :

- Ajout (ou enregistrement) :
    1. vérifier la validité du paramètre : s'il n'est pas valide on lance une exception ;
    2. voir s'il y a réellement quelque chose à faire : si le travail a déjà été fait, on arrête la méthode ;
    3. Voir si on peut l'ajouter : si le maximum autorisé a déjà été atteint ou si une autre contrainte empêche l'ajout, on arrête la méthode ;
    4. Vérifier qu'on pourra maintenir la cohérence des données c.à.d. vérifier que, si l'objet courant n'a pas encore été ajouté (enregistré) de l'autre côté, on pourra ensuite l'ajouter (l'enregistrer).
    5. Effectuer l'Ajout (ou l'enregistrement).
    6. Demander à l'autre côté d'effectuer son travail.

Remarque :

Selon le contexte, il pourrait y avoir des contraintes supplémentaires qui interdisent d'effectuer l'ajout (l'enregistrement). Il faut évidemment tester toutes les contraintes avant de faire le travail.

- Suppression :
    1. S'il y a un paramètre, tester sa validité : s'il n'est pas valide on lance une exception ;
    2. Voir s'il y a réellement quelque chose à faire : s'il n'y a rien à faire, on arrête la méthode ;
    3. Effectuer la suppression
    4. Demander à l'autre côté d'effectuer son travail.

Simplification des associations

Il est parfois possible de simplifier l'implémentation des associations. C'est notamment le cas si l'association est immuable. Par exemple, si une commande ne peut pas changer de client et que le client est connu dès la création de la commande, on peut alors passer le client en paramètre au constructeur et ne pas implémenter `enregistrerClient`, `supprimerClient`, ni `maximumClientAtteint`.

Si de plus la multiplicité est 1 (ou un nombre fixe), on refusera un paramètre `null` dans le constructeur et il ne faudra pas implémenter la méthode `minimumClientGaranti`.

L'idée consiste à se simplifier la vie face à ces associations qui deviennent parfois terriblement fastidieuses à implémenter en utilisant la manière vue précédemment.

Implémenter les associations par délégation

Les classes du domaine ne vont plus implémenter directement les associations, celles-ci seront prises en charge par des classes package (visibilité restreinte au package). Les classes du domaine n'auront donc plus dans leur état les attributs qui matérialisent l'association et elles délègueront la gestion de ceux-ci à des classes qui maintiendront les associations.

Concrètement, si une classe `Client` et `Commande` possèdent une association bidirectionnelle telle qu'une personne possède plusieurs commandes et une commande est celle d'une seule personne. Dans la manière vue précédemment, cette association se traduit par la présence d'un attribut `Client` dans `Commande` et d'un attribut `List<Commande>` dans `Client`. Dans notre version du jour, aucun attribut ne figurera directement dans les classes du domaine.

Nous allons implémenter une classe `ClientCommande` qui contiendra une `Map<Client,List<Commande>>`. Cette classe sera de **visibilité package** et implémentera le **pattern singleton**.

Les classes `Client` et `Commande` ne possèderont plus d'attributs `List<Commande>` et `Personne` mais seulement l'instance de `ClientCommande`. Toutes les méthodes liées à l'association seront présentes mais délègueront le travail à cette instance.

Par exemple, la méthode `ajouterCommande(Commande commande)` dans `Client` délègue le travail à l'instance de `ClientCommande` (appelons-là `clientCommande` pour l'exemple) :

```
clientCommande.ajouterCommande(commande, this).
```

La méthode équivalente dans `Commande`, `enregistrerClient(Client client)` délègue également le travail à l'instance de `ClientCommande` de façon similaire :

```
clientCommande.ajouterCommande(this, client).
```

L'intérêt vient du fait que pour une association bidirectionnelle, il ne faut pas deux méthodes d'ajout et deux méthodes de suppression : plus besoin d'`enregistrerClient` ou de `supprimerClient` dans `ClientCommande`.