CC BY-NC-SA

BTS SIO 2020

Modélisation détaillée avec UML

Rédigé par

David ROUMANET
Professeur BTS SIO

Changement

Date	Révision
Août 2020	Création
21/10/2021	Ajout dépendances

(cc) BY-NC-SA

Sommaire

A Modéliser avec UML....1
A.1 Problématique....1
A.2 Les détails des diagrammes de classes....1
A.3 Objectifs du cours....1
B Diagrammes de classes détaillés....2
B.1 Description du cas....2
B.2 Les classes....2
B.3 Les relations....3
B.3.1 Les multiplicités....4
B.3.1.a Catégories des multiplicités....4
B.3.1.b Multiplicité de un à N....5
Base de données....5
Codage POO....5
B.3.1.c Multiplicité un à un....6
Base de données....6
Codage POO....6
B.3.1.d Multiplicité de N à N....7
Base de données....7
Codage POO....7
B.3.2 Les dépendances....8
Codage POO....8
B.3.3 La navigabilité....9
B.3.4 L'arité....10
B.4 Diagramme de classes POO....11
B.4.1 Symboles des attributs et méthodes....11
B.4.1.a Les portées (niveaux d'accessibilité)....11
B.4.1.b Les types....11
B.4.1.c Les méthodes....11
B.4.2 Symboles des classes....12
B.4.2.a Les interfaces....12
B.4.2.b Les abstractions....12
C Annexes....14
C.1 Sources complémentaires....14

(cc) BY-NC-SA

A Modéliser avec UML

A.1 Problématique

Nous avons vu que le langage UML permet de modéliser de nombreuses vues d'un projet et nous avons étudié les bases des diagrammes d'utilisation, diagrammes de séquences et diagrammes de classes.

En BTS SIO, le diagramme de classe revêt une importance particulière et il existe de nombreuses manières de traduire un cahier des charges. Pour un néophyte, connaître les symboles peut suffire mais comprendre un diagramme nécessite plus de capacités :

- celle de connaître les symboles et leur interprétation
- celle de choisir un symbole pour représenter le bon concept
- celle de discerner la solution la moins mauvaise

Comment modéliser sans se tromper ?

A.2 Les détails des diagrammes de classes

La modélisation existant depuis longtemps, un certain nombre de règles et de modèles existent déjà : votre travail sera de reconnaître ces cas et de savoir quand les utiliser. Ce cours est donc orienté sur les règles de représentation des diagrammes de classes.

A.3 Objectifs du cours

Les objectifs de ce cours – classés selon la taxonomie de Bloom – sont :

Mémoriser	Connaître le vocabulaire et les symboles
Comprendre	Choisir les symboles pour représenter une situation
Appliquer	Mettre en œuvre les diagrammes sur des cas concrets
Analyser	Analyser un cahier des charges
Évaluer	–
Créer	–

B Diagrammes de classes détaillés

Nous avons la capacité à représenter les différentes fonctions décrites dans un cahier des charges, par l'utilisation de différents diagrammes UML.

Le développement orienté objet implique cependant une analyse orientée objet. Nous allons étudier les différentes associations, les conséquences du choix des multiplicités, ce qu'est l'arité, la navigabilité.

Il est important de respecter le formalisme, pour être le plus précis et complet possible. La modélisation nécessite d'utiliser un vocabulaire ne prêtant pas à confusion : elle vise à écarter les erreurs d'interprétation avec le maître d'œuvre (le client).

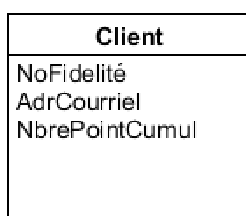
Le prochain paragraphe porte donc sur la description d'une relation simple, et les informations qu'elle porte. L'analyse faite au fur-et-à-mesure sert d'exemple de raisonnement. Il s'agit d'une démarche itérative, dans laquelle nous essayerons d'améliorer la modélisation dans le temps.

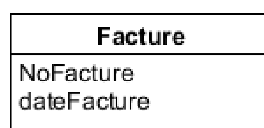
B.1 Description du cas

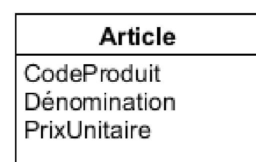
Un vendeur Internet propose des **articles** a ses **clients** français. Pour les clients fidèles, ils peuvent disposer d'un compte permettant de cumuler des points : pour cela, ils doivent juste laisser leurs **coordonnées** (adresse de courriel et adresse postale). À chaque facture, leur compte sera provisionné d'un nombre de points calculé sur la base du total de la **facture**. Des promotions (points doublés) seront proposées en fonction des villes qui auront la participation la plus forte.

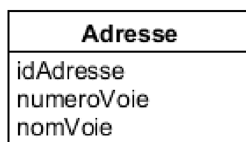
B.2 Les classes

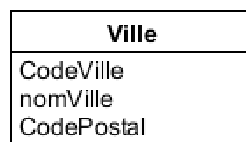
Il apparaît rapidement que client, article et facture sont des données utiles dans un magasin. Nous pouvons donc créer un diagramme de classe avec quelques données :

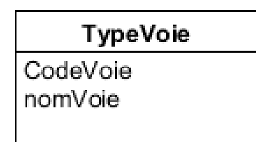












Vous constatez immédiatement qu'il y a plus de classes que le cahier des charges n'en décrit : pourquoi ?

 BY-NC-SA

La réponse est simple : notre application utilisera une base de données pour stocker les transactions dans le temps. Nous devons imaginer que le nombre de client peut croître de manière forte. Enregistrer l'adresse complète du client dans une chaîne de caractère est la solution qui apparaît facilement mais qui pose des problèmes.

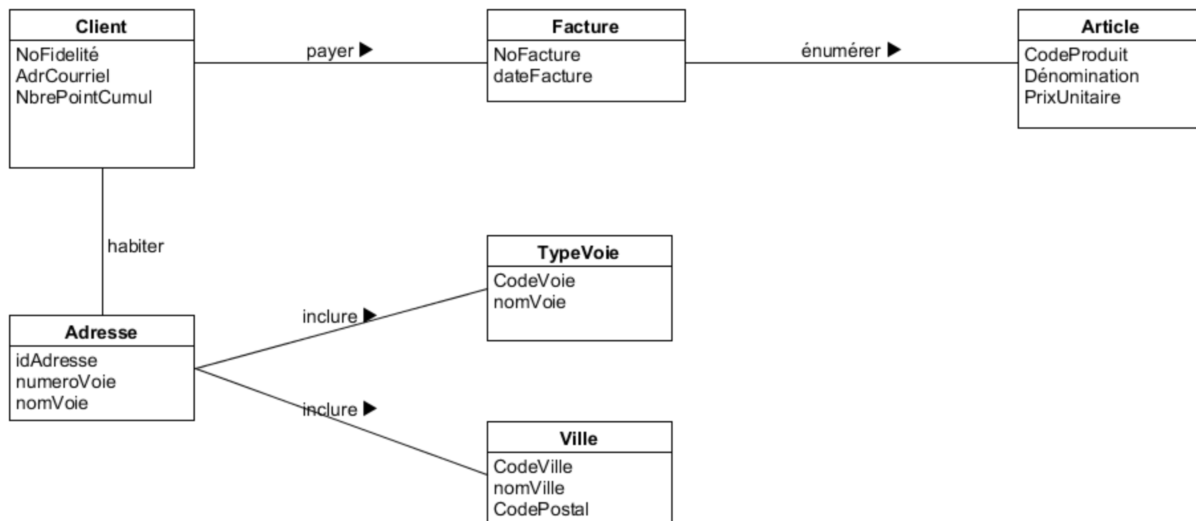
- Comment rechercher la ville qui a le plus de transaction ? Il faut un champ dédié.
- Quel encombrement est généré par une ville ayant un grand nombre de caractères (par exemple, Saint-Remy-en-Bouzemont-Saint-Genest-et-Isson) dans chaque utilisateur, plutôt qu'un numéro d'index sur 4 octets ? Un champ dédié économise de la place.
- Quel est le risque pour l'ensemble des clients d'écrire le nom d'une commune de plusieurs manières, rendant la recherche plus difficile (oubli des tirets, majuscules et minuscules, faute de frappe) ? Une liste de ville permet de réduire les risques d'erreurs, surtout si elle est pré-remplie.

Le code postal n'est pas unique à une ville :
– Grenoble dispose de deux codes postaux : 38000 et 38100
– Un même code postal peut représenter plusieurs communes

Ainsi, les classes sont généralement plus nombreuses que celles citées dans un énoncé ou cahier des charges, dans une recherche d'amélioration de performances. La connaissance approfondie du fonctionnement des bases de données est un avantage non négligeable.

B.3 Les relations

Placer les relations, c'est décider du lien qui relie deux classes ensemble. Voici une proposition :



Ici, le lien permet de retrouver un objet par rapport à un autre : un client habite à une adresse. Une adresse inclut un type de voie et une ville. Un client paye une facture qui contient des articles.

Avant de valider ces liens, il faut pratiquer une deuxième itération avec les multiplicités…

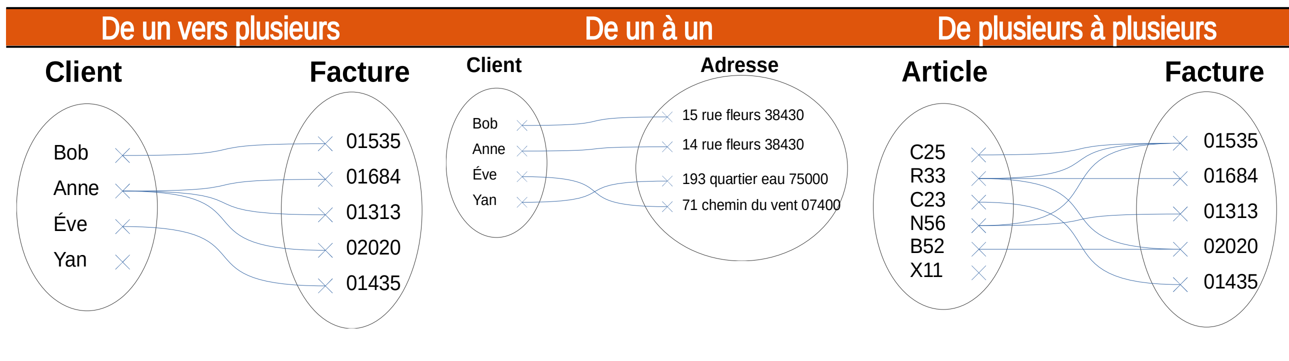
B.3.1 Les multiplicités

Le choix des multiplicités permet de préparer les contraintes de développement et de création de la base de données. Il suffit de se poser la question du nombre pour chaque relation, et ce, dans les deux sens de la relation.

B.3.1.a Catégories des multiplicités

Il faut penser chaque élément d'une entité comme ayant des caractéristiques réelles : un client à un nom, ce client peut avoir combien d'éléments dans la classe à laquelle il est lié ?

On trouvera 3 catégories de relations :



Pour chaque type de relation, il faut se poser la question du nombre minimal :

- Il peut être à zéro (nul) car existant mais non-attribué ou en cours d'attribution
- Il doit être à un car il n'existe pas seul (une facture doit forcément avoir un client)

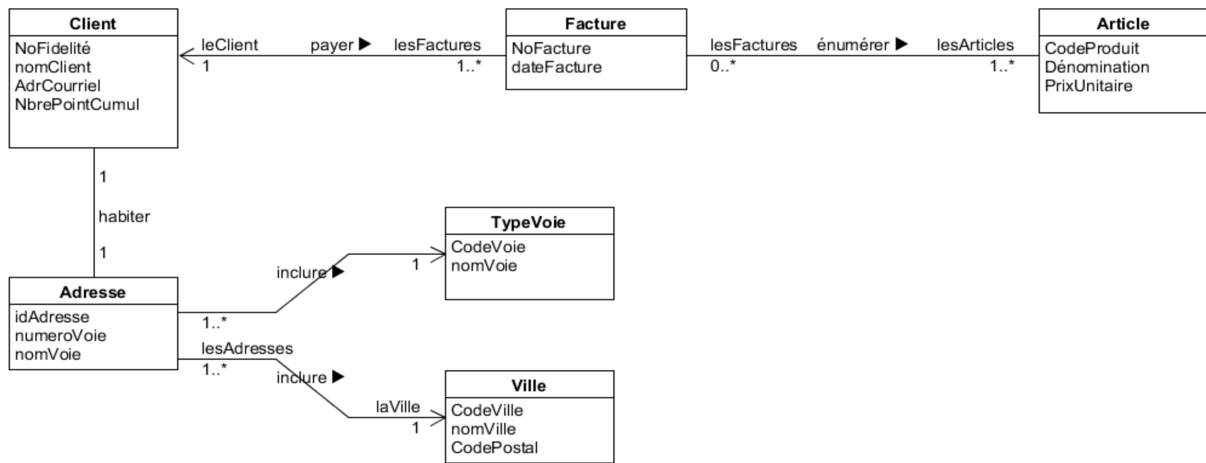
En fonction de la catégorie d'une relation, le traitement de modélisation pourra changer.

Exemple pour l'adresse et la ville :

- une adresse peut-elle avoir plusieurs villes ?
- Une ville peut-elle avoir plusieurs adresses ?

Ce travail va permettre de mettre en évidence de nouveaux besoins…

(cc) BY-NC-SA



En UML, la multiplicité se note X..Y (exemple : 1..*, 0..1, 0..5) tandis qu'en MCD la cardinalité s'écrit X,Y !

B.3.1.b Multiplicité de un à N

Dans le cas de notre vendeur Internet, un client peut avoir plusieurs factures.

Base de données

Dans une base de données, La création des tables implique que la table Facture contienne comme clé étrangère, la clé primaire du client.

Table **Facture**

NoFacture	dateFacture	#Client
010568	2020-07-18 08:52:26	227
010569	2020-07-18 10:55:11	101
010570	2020-08-01 09:13:21	227

Codage POO

Côté Facture, il suffit de mémoriser un seul objet de la classe Client :

```
Class Facture {
   Client leClient = new Client();
   int NoFacture;
}
```

En revanche, un client pouvant avoir plusieurs factures, stockera celles-ci dans une collection :

```
Class Client {
   list <Facture> lesFactures = new list<Facture>();
   String nomClient;
}
```

(cc) BY-NC-SA

B.3.1.c Multiplicité un à un

Un client ne peut habiter qu'une seule adresse : cela paraît logique… mais s'il déménage, devons-nous écraser l'adresse dans la base ? Si oui, la multiplicité reste bonne ; sinon, il faut modifier ma multiplicité ! C'est au maître d'œuvre de trancher cette question, pas au développeur.

Base de données

Dans la préparation au modèle physique, on envisage de placer la clé primaire de Client en clé étrangère d'adresse. L'inverse est également possible (clé primaire d'adresse dans la table client) mais il ne faut pas faire les deux en même temps.

Table **Adresse**

IdAdresse	NumeroVoie	NomVoie	#idTypeVoie	#idVille	#Client
115551	25	Des peupliers	2	11	227
115552	19	De l'Orme	1	11	511
115553	3	George pompidou	3	25	684

La table Adresse sera également enrichie des clés étrangères de TypeVoie et Ville.

Codage POO

Côté Client comme côté Adresse, il suffit de mémoriser un seul objet de la classe Client :

```
Class Adresse {
    Client leClient = new Client();
}
```

ou

```
Class Client {
    Adresse lAdresse = new Adresse();
}
```

B.3.1.d Multiplicité de N à N

Une facture peut contenir au moins un article et un article peut avoir plusieurs factures : nous considérons que l'article est un type d'article qui n'est pas unique (l'article «1 litre de Lait de marque VacheMeuh» existe en plusieurs exemplaires).

Base de données

Dans ce cas, il y aura création d'une table supplémentaire qui aura pour clés étrangères, la clé primaire de **Facture** et la clé primaire de **Article**. Appelons-la, table Énumérer

Table **Enumerer**

#idArticle	**#idFacture**	quantité
C27R	227	3
C21D	101	2
C27R	247	6
C05A	227	1

À noter, la clé primaire de cette table est alors constituée des deux clés étrangères.

Codage POO

Il est simplement nécessaire de respecter les données de la base, en créant la classe Énumérer.

Une erreur serait de rédiger le code suivant, dans lequel, rien ne relie les factures aux articles, et inversement :

```
Class Enumerer {
    list <Facture> lesFactures = new list<Facture>();
    list <Article> lesArticles = new list<Articles>();
    int quantite;
}
```

Il est préférable d'agir comme pour deux relations 1 vers N :

```
Class Article {
   int idArticle;
   list <Facture> lesFactures = new list<Facture>();
}
```

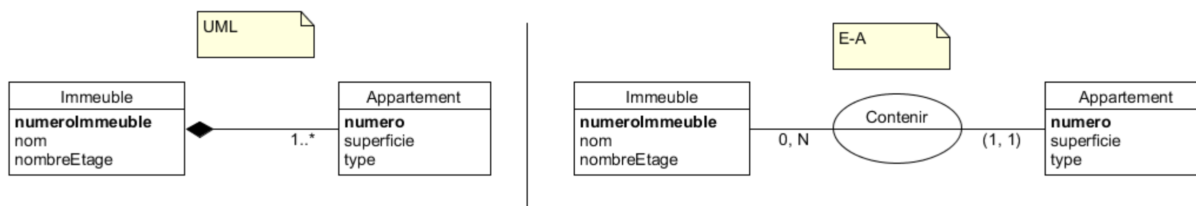
et

```
Class Facture {
   int idFacture;
   list <Article> lesArticles = new list<Articles>();
}
```

CC BY-NC-SA

B.3.2 Les dépendances

Il s'agit d'une association particulière, dans laquelle une classe dépend d'une autre. Par exemple, un appartement dépend de l'existence de l'immeuble, car si on détruit l'immeuble, l'appartement n'existe plus. Cette association est appelée composition.



Codage POO

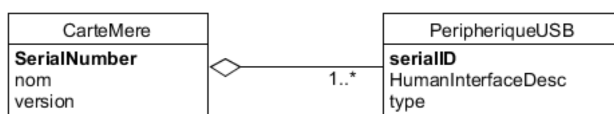
L'instanciation de la classe Appartement doit se faire à l'intérieur de la classe Immeuble.

```
Class Immeuble {
    list <Appartement> lesAppartements = new list<Appartement>();
}
```

À noter : en modélisation Entité-Association, on dirait qu'il y a une entité dépendante. Le numéro de l'appartement n'est unique, qu'au sein de l'immeuble. L'écriture en schéma relationnel serait :

```
Immeuble(numeroImmeuble, nom, nombreEtage)
Appartement(#NumeroImmeuble, Numero, superficie, type)
```

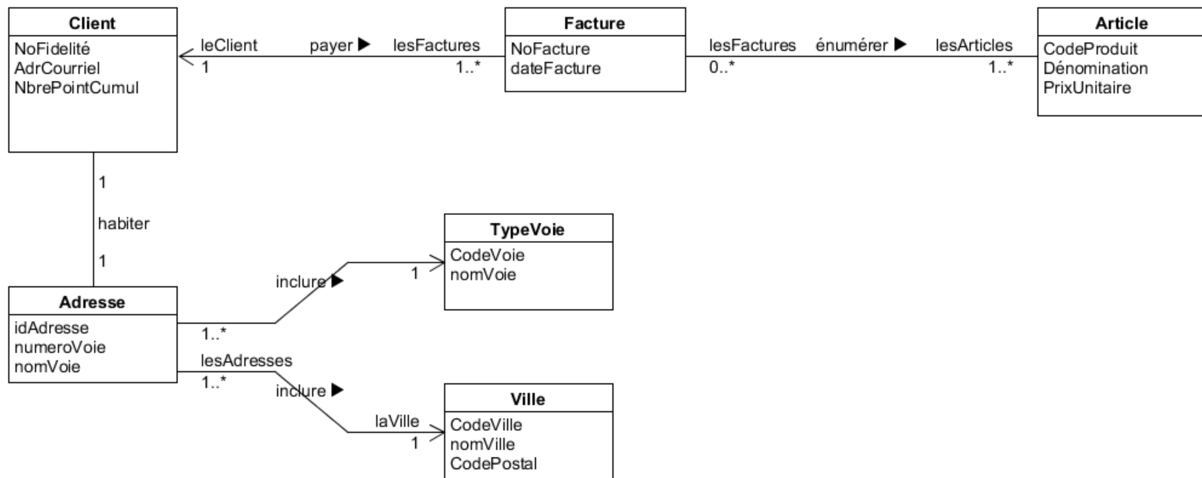
Une autre dépendance moins forte en UML s'appelle l'agrégation : un PC est constitué de composants, mais si on supprime le PC, on peut conserver les composants (pour les réutiliser). La schématisation UML est légèrement différente, et évidemment, l'instanciation ne se fait pas dans la classe principale.



B.3.3 La navigabilité

Il n'y a pas toujours d'intérêt à ce que deux classes se connaissent l'une et l'autre.

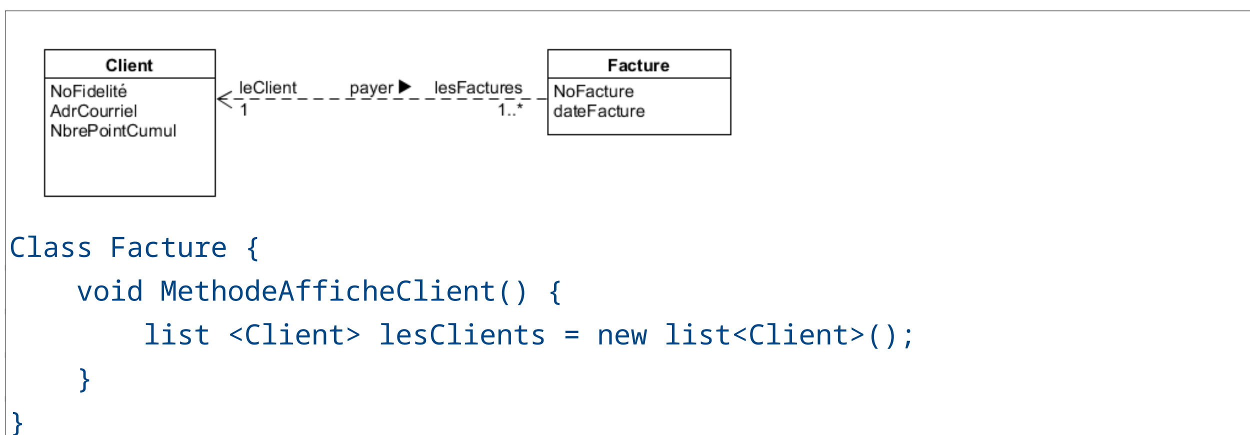
Dans notre exemple, bien que le service soit orienté vers le client, il n'est pas utile que le client connaisse toutes ses factures. Une simple requête sur la table Facture, avec une recherche sur l'attribut idClient (clé étrangère) permettra de toutes les retrouver. Les navigabilités peuvent être modifiées comme suit :



Cela ne change rien pour la création de la base de données, mais apporte une information complémentaire au développeur : la classe Client ne comportera pas de liste de factures.

La relation par défaut (un simple trait sans flèche) signifie que les deux classes se connaissent mutuellement. L'ajout d'une flèche d'un côté ou de l'autre pointe vers la classe qui sera connue. Il est inutile de placer une flèche des deux côtés, cela n'a pas de sens.

Il existe une particularité de représentation pour les relations faibles, qui sont dessinées en pointillées : cela signifie que ce n'est pas la classe qui connaît l'autre mais une méthode de la classe. La déclaration sera à l'intérieur de cette méthode. Les attributs à l'intérieur d'un bloc, sont détruits à la fin de la méthode. Exemple :




```
Class Facture {
    void MethodeAfficheClient() {
        list <Client> lesClients = new list<Client>();
    }
}
```

(cc) BY-NC-SA

B.3.4 L'arité

L'arité est un terme dont la définition se rapproche de « nombre d'arguments que prend une fonction ».

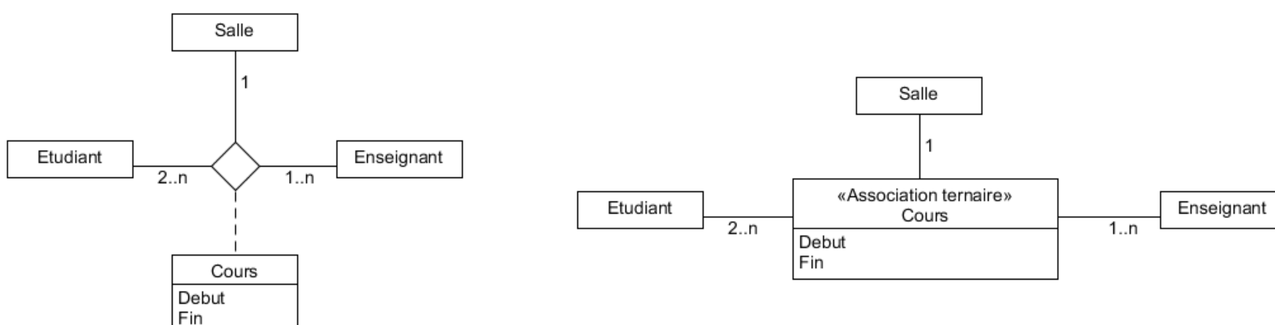
Dans la modélisation, l'arité correspond au nombre de classe que nécessite une relation. La plupart du temps, l'arité est binaire (2 classes) mais… il existe quelques cas (très prisés lors des épreuves de BTS SIO) ou l'arité est ternaire.

La recommandation des professionnels, est de tout faire pour ramener le cas à une arité binaire (les contraintes et les performances des bases de données rendent ce genre de relations délicates à gérer).

Voici cependant un exemple d'arité ternaire :

- un étudiant peut avoir cours dans plusieurs salles (mais pas en même temps)
- un enseignant peut avoir cours dans plusieurs salles (mais pas en même temps)
- un étudiant peut avoir plusieurs enseignants
- un enseignant peut avoir plusieurs étudiants
- un cours définit où se trouvent des étudiants à une horaire donnée, ainsi que les professeurs concernés
- un cours n'est disponible que dans une seule salle

Voici la modélisation UML :



à gauche, l'association ternaire voit également une classe-association Cours :

une classe-association contient des attributs qui ne peuvent apparaître dans aucune des autres classes de l'association.

Une association ternaire est donc une association qui nécessite 3 classes pour délimiter un ensemble cohérent.

Exemple : sur un marché, on trouve des fruits, des prix et des marchands : avec seulement le prix et le fruit, on ne peut pas retrouver le ou les marchands.

B.4 Diagramme de classes POO

La modélisation précédente permet encore de travailler au niveau de la base de données et de la préparation au codage. Cependant, pour représenter le code à écrire, il faut ajouter une étape qui intègre…

- les portées et les types des attributs ainsi que les méthodes
- les autres informations sur les classes elles-mêmes (abstraites, interfaces, etc.)

B.4.1 Symboles des attributs et méthodes

B.4.1.a Les portées (niveaux d'accessibilité)

Les portées correspondent à la visibilité des attributs d'une classe par les autres classes :

- Portée privée : symbolisée par un signe moins, elle indique que l'attribut n'est visible que par la classe propriétaire. Impossible de modifier l'attribut, sans passer par une méthode publique.
- Portée protégée : symbolisée par un signe dièse, elle indique que l'attribut n'est visible que par la classe propriétaire et ses classes dérivées.
- Portée publique : symbolisée par un signe plus, elle indique que l'attribut est visible ET modifiable directement par les classes qui utilisent cette classe. C'est extrêmement rare pour un attribut, mais très fréquent pour une méthode.

B.4.1.b Les types

Les types correspondent à ceux disponibles dans le langage de programmation. Un type date ne sera pas forcément compatible d'un langage à un autre (représentation différente en mémoire), mais permet au développeur de respecter le diagramme en utilisant le type spécifié.

L'important, est ici de réfléchir aux besoins de l'application, tout en essayant de trouver les passerelles entre le langage de programmation et la base de données.

B.4.1.c Les méthodes

Dans cette partie, il faut idéalement rédiger toutes les méthodes présentes dans la classe… cependant, il est fréquent que les mutateurs (setters) et accesseurs (getters) ne le soient pas. Par convention, il est possible d'utiliser des stéréotypes :

TypeVoie
- CodeVoie : string - nomVoie : string
+ getCodeVoie() : string + setCodeVoie(code : string) + getNomVoie() : string + setNomVoie(code : string) + vérifierTypeVoie() : string

TypeVoie
«get/set» - CodeVoie : string «get/set» - nomVoie : string
+ vérifierTypeVoie() : string

(cc) BY-NC-SA

B.4.2 Symboles des classes

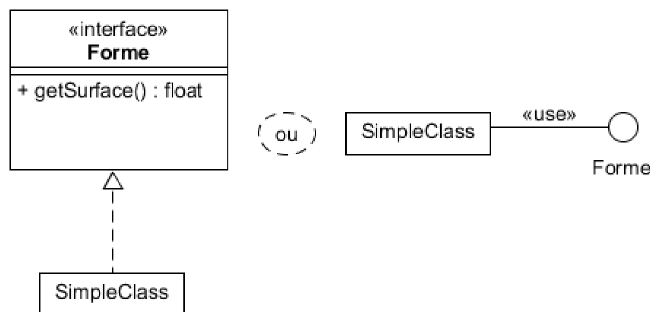
Les classes ont différentes représentations selon si elles sont abstraites ou s'il s'agit d'interfaces.

Cependant, on trouve ces représentations pour des relations de généralisation (héritage).

B.4.2.a Les interfaces

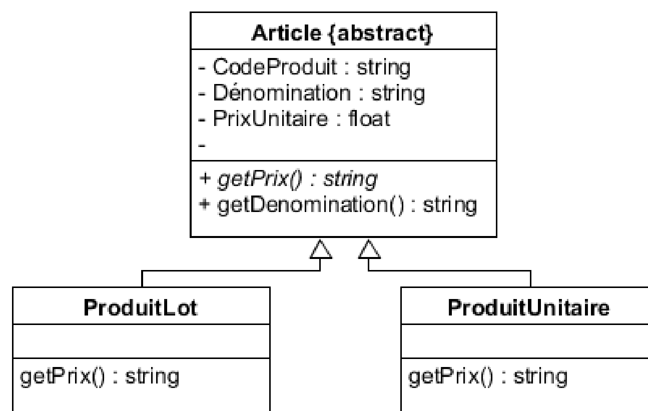
Il s'agit d'une classe qu'on ne peut pas instancier, elle représente un modèle : une sorte de contrat que le développeur doit respecter ! On ne peut qu'hériter de cette classe.

Une interface impose le formalisme des méthodes, mais propose en général un fonctionnement normalisé et reconnu par les classes qui connaissent cette interface. On peut utiliser le cercle, comme symbole d'interface, ou bien écrire le stéréotype << Interface >> avant son nom.



B.4.2.b Les abstractions

Moins contractuelle que l'interface, la classe abstraite contient une ou plusieurs méthodes que le développeur devra définir, s'il veut hériter de cette classe. Les méthodes à définir sont en italique.



Différence entre classe abstraite et interface

La différence en termes de développement entre une classe abstraite et une interface semble ténue. La tentation serait forte d'imaginer qu'une interface est une classe abstraite sans aucune méthode définie.

La réalité est que dans la plupart des langages, il n'est possible d'hériter que d'une seule classe, alors qu'il est possible d'implémenter plusieurs interfaces.

(cc) BY-NC-SA

C Annexes

C.1 Sources complémentaires

Vidéos – Cours UML (Emds) : https://www.youtube.com/watch?v=dJd6azZr9Kg&list=PLRR7wjtXb1cBQCE8ddM0B1D9DFj-WL3BX

Vidéo – Cours MCD (Emds) : https://www.youtube.com/watch?v=VFHVNA8xgK0

Support web – Laurent Audibert : https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes

Support web – interface versus classe abstraite : https://waytolearnx.com/2019/04/difference-entre-une-interface-et-une-classe-abstraite-en-java.html

MCD et équivalence UML / E-A : https://merise.developpez.com/faq/?page=MCD