Analyse pour le BTS IG/SIO

Alexandre Meslé

3 février 2020

Table des matières

Chapitre 1

Merise

1.1 Introduction

1.1.1 Pourquoi l'analyse ?

Ce cours portera essentiellement sur la branche de l'analyse permettant la modélisation de bases de données relationnelles. Les SGBDR (serveurs de bases de données relationnelles) sont des logiciels stockant des données dans des tables.

Les tables sont des tableaux à deux dimensions. Les données sont réparties dans ces tables d'une façon permettant de modéliser des situations plus ou moins complexes. L'analyse est une phase se trouvant en amont, et permettant de déterminer de quelle façon agencer les tables pour que les données représentent au mieux la réalité.

De façon très générale, analyser signifie "comprendre un objet en le décomposant en ses constituants". Ce qui se transpose aux bases de données en décomposant une situation réelle en données, tout en observant de quelle façon elles sont assemblées pour les modéliser au mieux à l'aide de tables.

1.1.2 Comment ça marche ?

Il existe plusieurs méthodes. La plus connue est la méthode Merise. Il s'agit d'un ensemble de techniques mathématiques ayant pour but de modéliser une situation à l'aide de shémas. Les trois premières étapes sont :

— **DDD**, le dictionnaire des données est l'ensemble des données qui seront représentées dans la base.
— **DF**, les dépendances fonctionnelles sont des relations entre les données.
— **MCD**, le modéle conceptuel des données, ou modèle entités-associations est un graphe faisant la synthèse de la situation. Le MCD est un modèle très puissant, il est dense et exprime une très grande quantité d'informations.

Une fois ces trois étapes achevées, il existe une technique permettant d'en déduire le **MPD** (modèle physique des données). Le MPD donne directemment l'architecture de la base de données.

1.1.3 Logiciels

Merise

Le logiciel **JMerise** permet de représenter des MCDs et de les exporter vers une base de données. Je vous conseille d'apprendre à vous en servir.

Les shéma dans ce cours sont réalisés à l'aide de Mocodo.

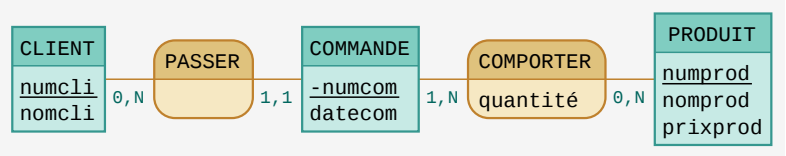


FIGURE 1.1 – Exemple Mocodo

UML

Le plugin pour eclipse objectaid génère un diagramme UML à partir d'un projet Java. Pour inclure des illustrations, UmlGraph présente l'avantage de générer de nombreux formats à partir d'un fichier de classes java et de gérer lui même le placement.

L'impossibilité de faire des classes-associations ou des relations ternaires a nécessité l'emploi de Tikz-Uml pour la rédaction de ce document.

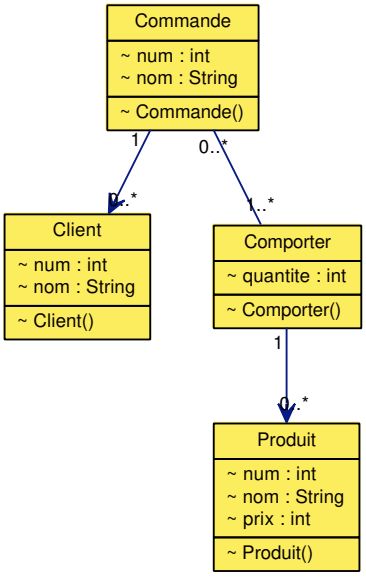


FIGURE 1.2 – Exemple UmlGraph

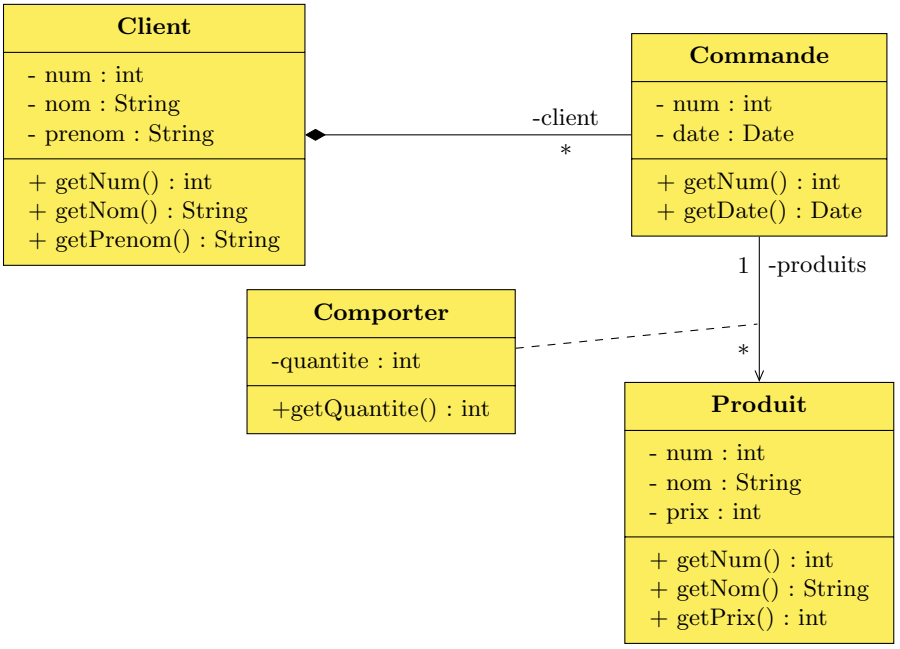


FIGURE 1.3 – Exemple Tikz-Uml

1.2 Dictionnaire des données

1.2.1 Critères de sélection des données

Les données doivent être :
— Sans redondance
— Atomiques

1.2.2 Données à ajouter au dictionnaire

Il est usuel d'ajouter au dictionnaire des données, numériques, appelées **identifiants**. Par exemple si vous devez modéliser une liste de clients, il serait malvenu de ne disposer que de leurs noms et prénoms. En effet, tout doublon dans ces données serait une source d'erreurs. Il convient donc d'ajouter un numéro de client (`numCli` par exemple) pour que chaque client soit identifié de façon unique.

Exercice 1 - Secrétariat pédagogique

Constituer le dictionnaire des données permettant de modéliser un secrétariat pédagogique A.1.

Exercice 2 - Chaîne d'approvisionnement

Constituer le dictionnaire des données permettant de modéliser une chaîne d'approvisionnement : A.2.

1.3 Dépendances fonctionnelles

1.3.1 Définition

Deux données A et B sont en dépendance fonctionnelle si la connaissance d'une valeur de A détermine la connaissance d'au plus une valeur de B.

Par exemple, la connaissance d'un numéro de sécurité sociale détermine un seul nom de famille, celui du titulaire de ce numéro. Par contre, un prénom ne détermine rien, car plusieurs personnes peuvent avoir le même prénom. On peut représenter cette DF de la façon suivante :

$$numSecu \rightarrow nomPers$$

Il est aussi possible que la donnée A soit composée de plusieurs données. Par exemple, si l'on souhaite connaître la note obtenue par un étudiant à un examen, il est nécessaire de connaître le numéro de l'étudiant, le numéro du module, et la session qu'il passait. Ce qui se représente :

$$numEtudiant, numModule, numSession \rightarrow valeurNote$$

1.3.2 Quelques règles

Identifiants

Si on a une dépendance $A \rightarrow B$, A est nécessairement un identifiant. En effet, toute donnée n'étant pas un identifiant est ambiguë, par conséquent, il est impossible de s'en servir pour établir des règles de dépendance.

Arcs de transitivité

Si on a $A \rightarrow B \rightarrow C$, il est inutile d'indiquer sur le diagramme que $A \rightarrow C$, cette relation n'apporte aucune information supplémentaire.

1.3.3 Dépendances fonctionnelles faibles

Une dépendance fonctionnelle de A vers B est dite **faible** si la connaissance d'une valeur de A permet de déterminer 0 ou 1 valeur de B. Dans ce cas on représente la flèche en pointillés.

Une DF "classique", par opposition à une DF faible, est dite **forte**.

Exercice 1 - Secrétariat pédagogique

Représentez le graphe des dépendances fonctionnelles associées au secrétariat pédagogique : A.1.

Exercice 2 - Chaîne d'approvisionnement

Construire le graphe des DF modélisant une chaîne d'approvisionnement : A.2.

1.4 Modèle conceptuel des données

Pour le moment, en analyse, nous avons vu comment décomposer une situation réelle en données, et nous avons vu aussi comment représenter des liaisons entre ces données. Cependant, le graphe des dépendances fonctionnelles est incomplet dans le sens où il ne permet pas de représenter suffisamment de liaisons pour représenter fidèlement la réalité.

Le MCD, dit aussi **Modèle Conceptuel des Données**, est une représentation graphique davantage complète que le graphe des dépendances fonctionnelles. L'élaboration d'un MCD nécessite plus d'observations de la situation que le graphe des DF. Le modèle est par conséquent suffisamment complet pour qu'il soit possible de déduire la répartition des données dans les tables sans observation supplémentaire de la situation. Cette étape est donc la dernière étape épineuse d'une analyse.

Commençons par examiner un exemple.

1.4.1 Exemple introductif

Le bureau des étudiants souhaite lorsqu'il planifie des soirées, lister les étudiants qui ont réservé des places pour gérer la billetterie. Nous utiliserons les données suivantes :

— numetud
— nometud
— prenometud
— mailetud
— numsoiree
— nomsoiree
— datesoiree

Il apparaît clairement que :

— $numetud \rightarrow nometud$
— $numetud \rightarrow prenometud$
— $numetud \rightarrow mailetud$
— $numsoiree \rightarrow nomsoiree$
— $numsoiree \rightarrow datesoiree$

On observe que les dates des soirées ne peuvent pas servir d'identifiant parce que plusieurs soirées peuvent être programmées le même soir par le BDE. On remarque aussi que les données *numetud*, *nometud*, *prenomnometud* et *mailetud* sont liées, parce que trois d'entre elles dépendent de *numetud*. De même, on peut regrouper *numsoiree*, *nomsoiree* et *datesoiree*.

A titre de parenthèse, rappelons que $numetud \not\rightarrow numsoiree$ parce qu'un étudiant peut avoir réservé des places pour plusieurs soirées (ou aucune). De façon analogue, on a $numsoiree \not\rightarrow numetud$, parce qu'il peut y avoir plusieurs étudiants dans une soirée, tout comme il peut n'y en avoir aucun. On conclura donc que le graphe des DF ne permet pas de représenter la façon complète la billetterie, vu qu'il ne permet pas de mettre en correspondance les étudiants et les soirées.

1.4.2 Les entités

Il est usuel, en analyse, de regrouper dans des **entités** des données liées à leur identifiant par une relation de dépendance fonctionnelle. On représente cela ainsi :

Les boîtes dans lesquelles les données sont regroupées sont des entités, les identifants doivent être soulignés, et toutes les données doivent être en dépendance fonctionnelle avec l'identifiant.

La clé dessinée à côté de l'identifiant est ajoutée par Open Model Sphere, vous n'avez pas besoin de la dessiner. Par rapport à la question que nous nous posions, et qui était comment représenter graphiquement les correspondances entre les soirées et les étudiants, nous n'avons toujours pas répondu.

1.4.3 Les Associations

Les associations sont des correspondances entre les entités, on les représente ainsi :

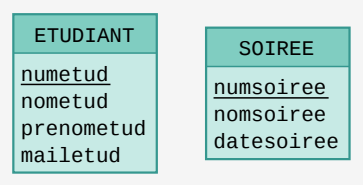


FIGURE 1.4 – Exemple entités

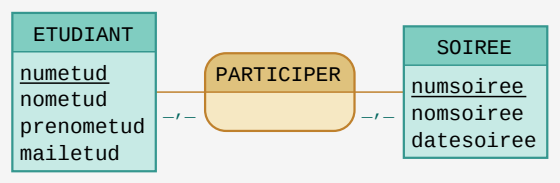


FIGURE 1.5 – Exemple associations

Le fait que Etudiant soit associé à Soirée signifie que la base de données devra permettre étant donné un *étudiant* particulier, de savoir à quelle(s) *soirée(s)* il a *participé*, et vice-versa, à partir d'une *soirée* donnée, il devra être possible de déterminer quels *étudiants* ont *participé*.

Il est usuel de choisir des verbes pour les nommer, pour la simple raison que la plupart du temps, les entités représentent des objets (au sens très large du terme), et les associations des actions impliquant ces objets.

1.4.4 Les Cardinalités

Etendons notre modèle : supposons que le BDE souhaite aussi gérer les lieux réservés pour les soirées. Il serait envisageable d'agrandir notre MCD de la sorte :

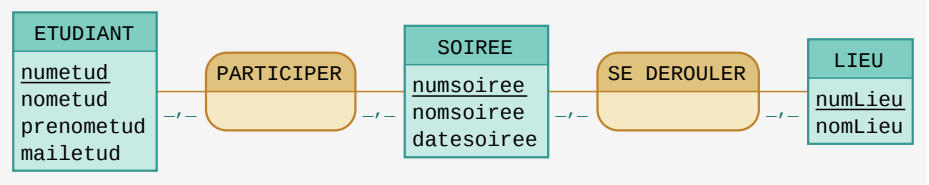


FIGURE 1.6 – Exemple incomplet...

Le problème qui se pose est que ce modèle est incomplet... En effet la relation `Se Dérouler` n'est du tout de la même nature que la relation `Participer`... Pourquoi ? Parce que pour une soirée on ne réserve qu'une salle à la fois ! On s'en convainc en remarquant qu'il y a une dépendance fonctionnelle entre *numsoiree* et *numlieu*.

Etant donnée une soirée particulière, on lui associe un et un seul lieu, alors qu'à un lieu correspond un nombre de soirées inconnu à l'avance. On complète le modèle en ajoutant ce que l'on appelle des **cardinalités** :

Le $0,n$ entre `Etudiant` et `Participer` signifie qu'à un étudiant sera associé un nombre de soirées pouvant varier de 0 à plusieurs. Le n signifie que le nombre de soirées peut être aussi grand que l'on veut. Le $0,1$ séparant `Soirée` de `Se Dérouler` signifie qu'à une soirée est associé 0 ou 1 salle. A votre avis, que signifient les autres cardinalités ?

1.4.5 Associations et attributs

Supposons que les étudiants aient le droit de venir avec des personnes extérieures, c'est-à-dire qu'ils aient le droit de réserver plusieurs places. Où mettre la donnée `nbPlaces` ? On ne peut pas la mettre dans `Etudiant`, ni dans `Soirée`,

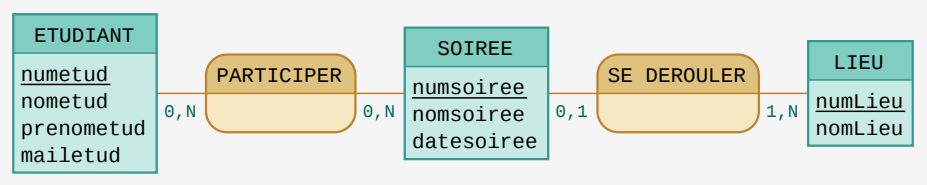


FIGURE 1.7 – Exemple cardinalités

parce que le nombre de place réservées par un étudiant à une soirée dépend à la fois de la soirée et de l'étudiant. La solution est la suivante :

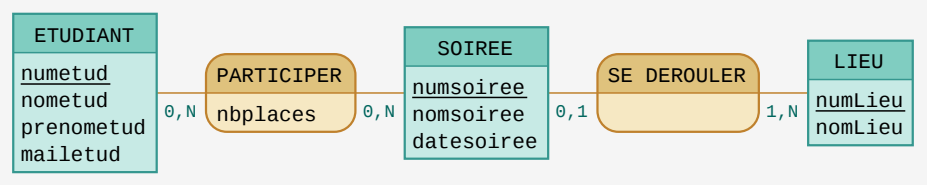


FIGURE 1.8 – Exemple attribut

Cet attribut de l'association `Participer` signifie qu'à chaque fois d'un étudiant annoncera sa participation pour une soirée, il devra préciser combien de places il souhaite réserver.

Exercice 1 - Chaîne d'approvisionnement

Construire un MCD modélisant la chaîne d'approvisionnement : A.2.

1.4.6 Associations complexes

Il est possible d'exprimer avec des associations des situations relativement élaborées. Des bases données complexes comme par exemple celles que l'on peut trouver dans des réseaux sociaux se modélisent de façon surprenante.

Réflexives

Si l'on souhaite par exemple représenter une liste d'internautes pouvant étant être amis entre eux. L'association `Etre ami` met donc un `Internaute` avec **un autre** `Internaute`. Il faut donc pour représenter cela utiliser une association réflexive.

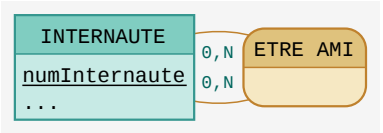


FIGURE 1.9 – Exemple réflexive

Multiples

Un autre cas qui pourrait se présenter est celui où il existe de multiples associations entre deux entités. Par exemple dans le cas d'un groupe (ou forum), il existe pour chaque groupe un internaute particulier qui en est le créateur. Cela ne doit pas empêcher d'autres internautes de s'inscrire sur le groupe. On représente cette situation avec deux associations différentes entre ces deux entités.

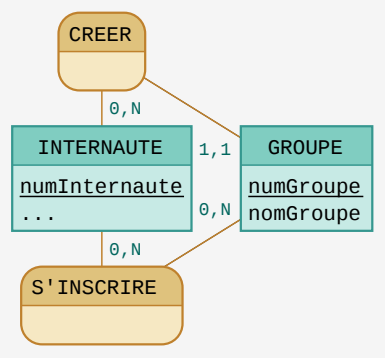


FIGURE 1.10 – Exemple multiple

Ternaires

Il est aussi fréquent que plus deux associations soient impliquées dans une association. Si par exemple, on souhaite garder en mémoire des produits achetés par des clients sur un site, l'association mettant en relation le produit, la personne qui l'achète, et la date d'achat devra donc relier trois entités.

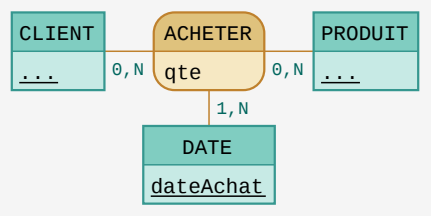


FIGURE 1.11 – Exemple ternaire

On remarque au passage que la quantité dépend à la fois de l'internaute, du produit et de la date.

Entité faible

Si on considère qu'une photo appartient à une et une seule personne, et qu'elle ne pourra jamais changer de propriétaire, on utilise une entité faible comme dans l'exemple ci-dessous.

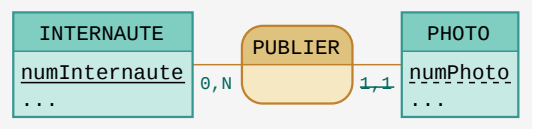


FIGURE 1.12 – Exemple entité faible

On remarque que l'entité faible est notée en soulignant l'identifiant en pointillés.
Il est aussi possible d'indiquer la présence d'une entité faible en mettant les cardinalités $(1,1)$ entre parenthèses.

1.4.7 Exercices

Exercice 2 - Secrétariat pédagogique

Représentez un MCD associé au secrétariat pédagogique (A.1).

Exercice 3 - Arbre généalogique

Représentez un arbre généalogique (A.3) avec un MCD.

Exercice 4 - CMS

Réalisez un MCD permettant de représenter un CMS (A.4).

Exercice 5 - Forum

Réalisez un MCD permettant de représenter un Forum (A.6).

Exercice 6 - Covoiturage

Réalisez un MCD permettant de modéliser un site de covoiturage (A.7).

1.5 Modèle physique des données

1.5.1 Introduction

Le MPD (Modèle physique des données) est la dernière étape de l'analyse. Le MPD n'est autre qu'une liste de tables avec pour chacune d'elle les colonnes faisant partie de cette table. Il s'obtient par calcul à partir du MCD.

1.5.2 Formalisme

Chaque nom de table est suivi d'une liste de colonnes entre parenthèses.
— Les clés primaires sont soulignées
— les clés étrangères sont précédées par un dièse.

1.5.3 Calcul du MPD

Entités

Une entité devient une table, tous ses attributs deviennent des colonnes et son identifiant devient la clé primaire.

Associations Plusieurs à plusieurs

Une association aux cardinalités de type $\ldots, n$ des deux cotés (donc de type plusieurs à plusieurs), devient aussi une table :
— Les attributs de cette association deviennent des colonnes.
— Les clés primaires des tables se trouvant de part et d'autres de l'association sont importées sous forme de clés étrangères.
— La concaténation de ces clés étrangères devient la clé primaire.

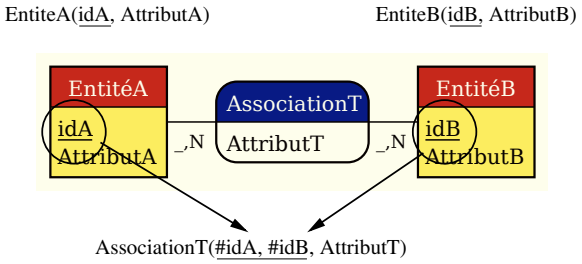


FIGURE 1.13 – Relations plusieurs à plusieurs

Dans l'exemple ci-dessus, *EntiteA* et *EntiteB* sont reliées par *AssociationT*, leurs clés primaires pA et pB sont dupliquées dans la table *AssociationT* pour devenir des clés étrangères. Le couple $(p1, pB)$ devient alors clé primaire de *AssociationT*.

Associations Un à plusieurs

Une association aux cardinalités de type $\ldots, 1$ d'un côté et $\ldots, n$ de l'autre, (donc de type un à plusieurs), disparaît :
— La clé primaire de la table se trouvant du côté *plusieurs* est dupliquée du côté *un*. Elle devient une clé étrangère.
— Les attributs de cette association sont reportés dans la table se trouvant du côté *un*.
Dans cet exemple, *AssociationT* disparaît. La clé primaire de *EntiteB*, se trouvant du côté plusieurs, est reportée dans la table *EntiteA* sous forme de clé étrangère. *AttributT* est lui aussi reporté dans la table *EntiteA*.

Exercice 1 - Livraisons

Construire le MPD pour le cas Livraisons.

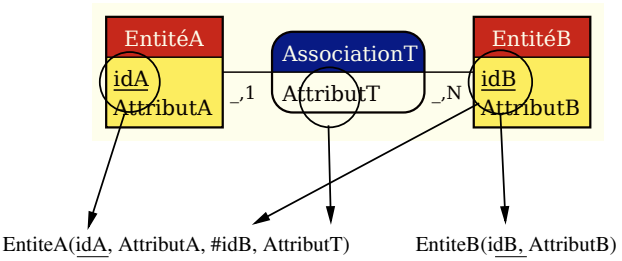


FIGURE 1.14 – Relations un à plusieurs

Exercice 2 - Secrétariat pédagogique

Construire le MPD pour le cas Secrétariat pédagogique.

Exercice 3 - Arbre généalogique

Construire le MPD pour le cas arbre généalogique.

Exercice 4 - CMS

Construire le MPD pour le cas CMS.

1.6 Exercices Récapitulatifs

Exercice 1 - Bibliothèque

Nous souhaitons gérer une bibliothèque décrite en annexe (A.5).

1. Réaliser une analyse complète : DDD, DF, MCD.
2. Déterminer le MPD et écrire le script de création de tables.
3. Insérer des données cohérentes dans vos tables.
4. Mettre en place des vues permettant de d'obtenir les informations suivantes :
   (a) Abonnés dont l'abonnement n'est plus à jour.
   (b) Ouvrages en circulation.
   (c) Exemplaires empruntés devant déjà être revenus, et les abonnés correspondants.
   (d) Ouvrages dont il existe des exemplaires disponibles pour le prêt (empruntables et non en circulation).
   (e) Ouvrages dont il n'existe plus d'exemplaire.
   (f) Ouvrages dont tous les exemplaires sont en circulation.
   (g) Nombre d'emprunts par ouvrage.
   (h) Abonnés pouvant emprunter (abonnement à jour et limite non dépassée).
   (i) Nombre d'ouvrages en circulation par abonné.
   (j) D'autres requêtes qui vous sembleraient intéressantes...

Chapitre 2

UML

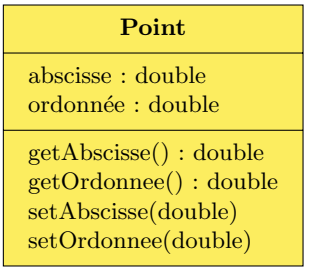
2.1 Introduction au UML

UML est un ensemble de représentations graphiques permettant de modéliser des applications utilisant des langages à objet. Dans ce cours nous nous concentrerons sur les **diagrammes de classes**. Le diagramme de classe est l'adaptation du MCD aux classes, c'est-à-dire une représentation permettant de rapidement comprendre l'architecture d'une application. Son emploi est incontournable dès que l'on aborde des projets d'une certaine ampleur.

Pour lire ce document, il est nécessaire d'avoir quelques bases de Merise (première partie de ce cours) et de connaître les grands principes de la programmation objet (classe, instances, héritage, interfaces, etc.).

2.1.1 Classes

Une **classe** est représentée par une boîte contenant trois sections :
— L'**entête** contient le nom de la classe et certaines informations complémentaires (abstraite, interface, etc.).
— Les **attributs** sont les variables de classe (**static**) ainsi que les variables d'instance (non **static**).
— Les **méthodes** contiennent les sous-programmes d'instance et de classe.

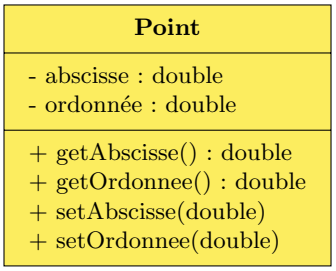


L'exemple ci-dessus nous montre une classe `Point` disposant de deux attributs `abscisse` et `ordonnée`, tous deux de type `double`. Les quatre méthodes (les `getter` et les `setter`) se trouvent dans la troisième partie de la boîte.

Visibilité

La **visibilité** d'un identificateur est notée en le faisant précéder d'un symbole `+` (**public**) ou `-` (**private**), `#` (**protected**) ou `~` (**package**).

Un attribut est :
— **public** s'il est possible de l'utiliser depuis n'importe où dans le code.
— **privé** s'il n'est visible que depuis l'intérieur de la classe. On omet souvent les attributs privé dans le diagramme de classe.
— **protected** s'il n'est visible que depuis les sous-classes. Attention, en Java un attribut protected n'est accessible que sur l'objet **this**, vous ne pourrez pas vous en servir sur un autre objet de la même classe.
— **package**, la visibilité par défaut, s'il n'est visible que dans le package.



Par exemple dans la classe `Point` ci-avant `abscisse` et `ordonnée` sont privés tandis que les quatre méthodes sont publiques.

Attention : Avec le logiciel que j'utilise, les attributs et méthodes statiques sont soulignés.

2.1.2 Relations

Lorsque deux classes sont reliées par une flèche, cela signifie que chaque instance de la classe se trouvant à l'extrémité initiale de la classe contient une référence vers une (ou plusieurs) instances de la classe se trouvant à l'extrémité finale de la flèche. Donc si flèche part de la classe A vers la classe B, on dit qu'il y a une **relation** de A vers B.

Cet exemple modélise l'inscription d'étudiants à des soirées. Un étudiant sera en correspondance avec les soirées auxquelles il est inscrit.

Attribut

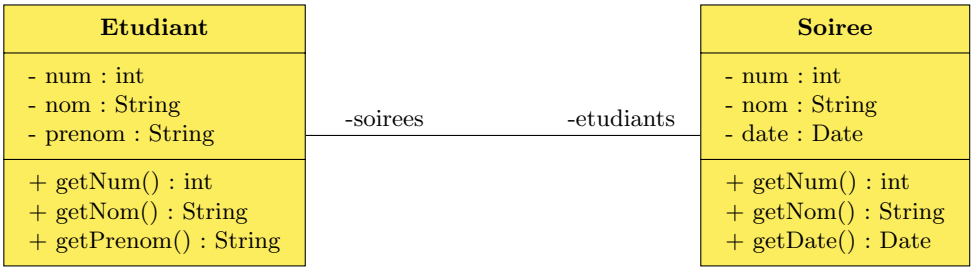
L'attribut de la classe située à l'extrémité initiale de la flèche servant à représenter la relation est très souvent précisé à côté de la flèche afin d'indiquer comment cette relation est implémentée. Pour éviter les redondances, on ne la précise pas avec les attributs de la classe.

Par exemple, mémoriser pour un étudiant l'ensemble des soirées auxquelles il est inscrit nécessite une variable (non-scalaire) appelée ici `soirées`.

Sens de navigation

Vous noterez que les sens des flèches indiquent un sens de navigation entre les objets. Si l'on a $A \longrightarrow B$, cela signifie qu'il est possible depuis un objet A de déterminer directement l'objet B qui est en relation avec. Vous noterez que la navigation dans l'autre sens ne sera pas facilité par cette spécification.

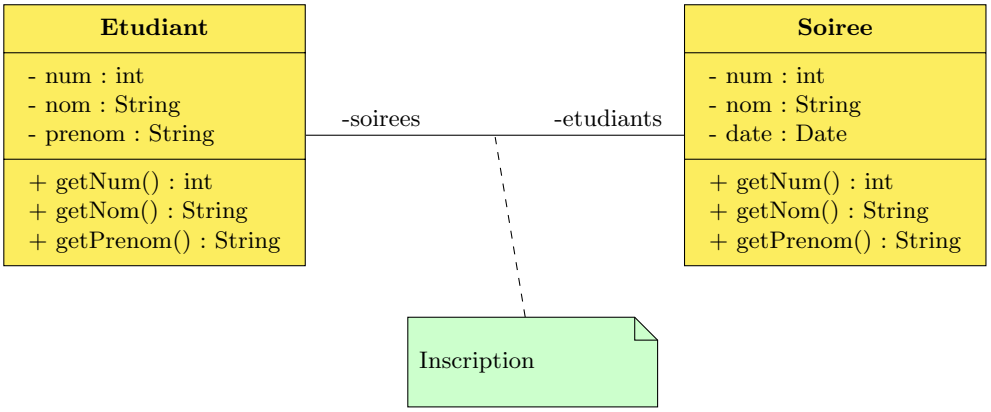
L'absence de flèche signifie que la navigation peut s'effectuer dans les deux sens. Cela peut être tout à fait pratique pour exploiter les classes, mais ce type de référence croisée peut être assez difficile à programmer.



La relation est ici bidirectionnelle, il est possible à partir d'un étudiant de retrouver les soirées et à partir des soirées de retrouver un étudiant.

Label

Une relation peut être accompagnée d'un label étiquetant la nature de la relation, ce qui est apprécié lorsque cette dernière n'est pas évidente.

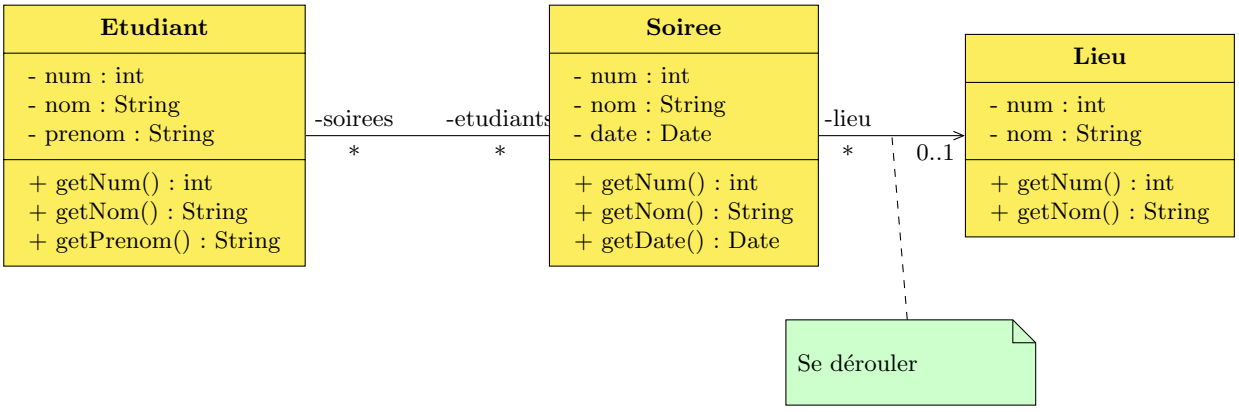


Multiplicité

Les valeurs aux extrémités des flèches sont l'analogue des cardinalités en Merise. Sauf qu'elles sont **à l'envers** !
Vous remarquerez aussi que les notations peuvent être très hétérogènes :

— Certaines sont de la forme $inf..sup$, où inf est le plus petit nombre d'instances référencées, et sup le nombre maximal. Une valeur arbitrairement grande est notée $*$.

— Tandis que d'autres n'indiquent qu'une seule valeur. 1 s'il y a un objet référencé, $*$ s'il peut y en avoir 0, 1 ou plusieurs.



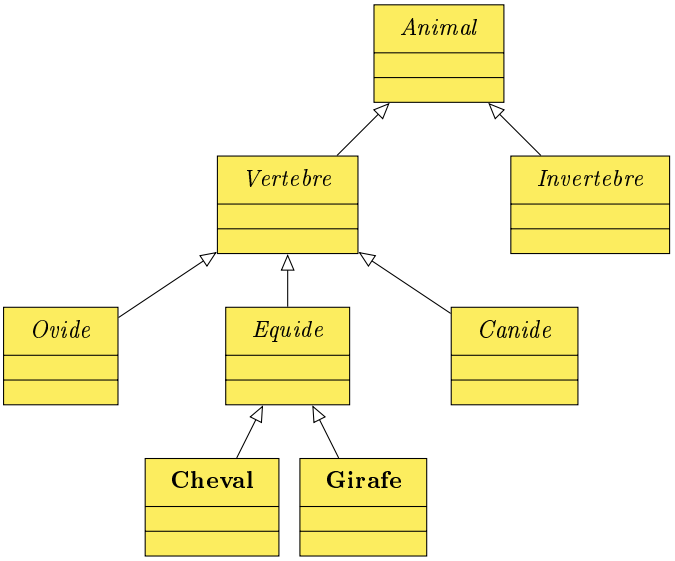
Un étudiant peut s'inscrire à 0, une ou plusieurs soirées, tout comme à une soirée peut s'inscrire un nombre arbitraire d'étudiants. Par contre, une soirée ne peut se dérouler que dans un seul lieu, Le 0 signifiant qu'on se garde

le droit de représenter des soirées dont le lieu n'a pas encore été déterminé. On notera que plusieurs soirées peuvent se dérouler dans le même lieu.

2.1.3 Héritage

L'**héritage** est noté avec une flèche (quelques fois en pointillées) terminée par un triangle vide. La classe mère peut être une classe conventionnelle, une classe abstraite, ou encore une interface. On précise dans ce cas la nature de la classe mère afin d'éviter toute confusion. On prête aussi attention au fait que dans certains langages à objets, en particulier Java, l'héritage multiple est interdit.

Le cas particulier d'héritage qu'est l'implémentation se représente avec des flèches en pointillés.



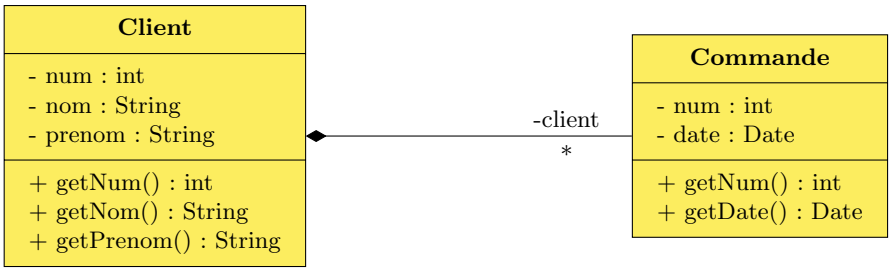
Un objet de type `Girafe` sera aussi de type `Equidé`, de type `Vertébré` et enfin de type `Animal`.

Attention : Avec le logiciel que j'utilise, les classes et méthodes abstraites sont représentés en italique.

2.1.4 Relations spécifiques

Composition

La **composition** est la notation utilisée pour transposer les entités faibles ou mondes objets. Une classe A est agrégée à une classe B si toute instance de A est reliée à une (et exactement une) instance de B, et ne peut donc pas exister sans cette instance de B. La composition implique qu'une instance de A sera toujours en relation avec la même la instance de B. On la note avec un losange noir du côté de la classe propriétaire.



Une commande sans client n'existe pas, la destruction du client entraîne la destruction de toutes ses commandes. Une fois crée, une commande sera toujours associée au même client.

Agrégation

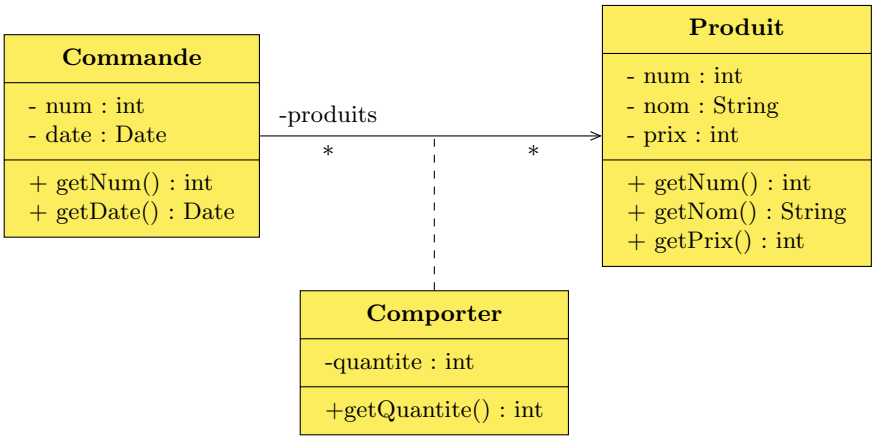
Une classe A est l'**agrégation** d'une classe B lorsqu'une instance de A est une collection d'instances de B. On la représente avec un losange blanc du côté de la classe contenant la collection.

La différence avec une relation est simplement conceptuelle vu que dans les langages à objets, l'agrégation s'implémente de la même façon que si elle avait été représentée avec une relation plusieurs à plusieurs.

Une liste d'invités n'est qu'une classe pour regrouper des invités. Mais la disparition de la liste n'entraîne pas nécessairement la disparition des invités. C'est pour cela que dans le cas présent une agrégation à été préférée à la composition.

Classe d'association

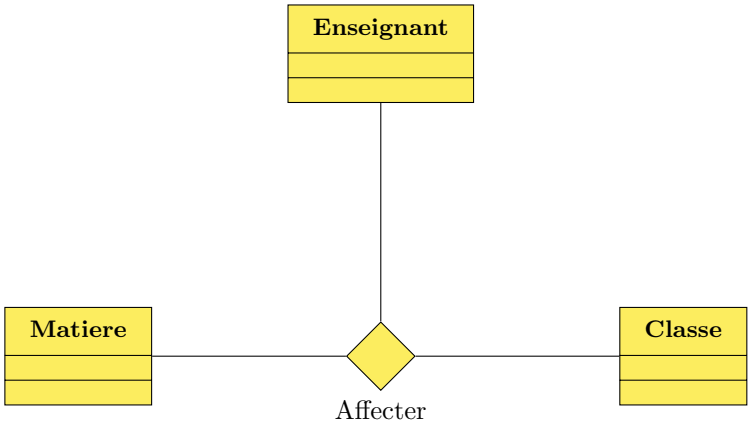
Dans le cas de relation plusieurs à plusieurs, il est fréquent que des valeurs accompagnent la relation. Dans les langages à objets, on n'a d'autres choix que de créer une classe spécialement pour ces valeurs. A chaque couple d'instances impliquées dans la relation correspond donc une instance de la **classe d'association**. La classe d'association est reliée avec des traits en pointillés à la relation.



Le fait qu'un produit figure sur une commande ne suffit pas à complètement caractériser leur relation. Il est nécessaire de disposer de la quantité de produit figurant sur cette commande. Comme cette quantité dépend à la fois du produit et de la commande, celle-ci ne peut figurer que dans une classe d'association.

Relations ternaires

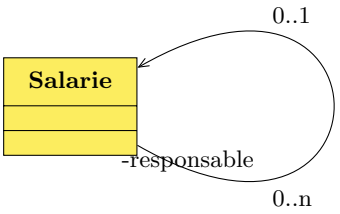
Une **relation ternaire** est un ensemble de triplets mettant en relation trois objets. On l'écrit avec un losange relié aux classes de ces trois objets.



Une affectation concerne un enseignant, une matière et une classe. Ce qui signifie par exemple que chaque enseignant sera en correspondance avec des couples (matière/classe).

Relations réflexives

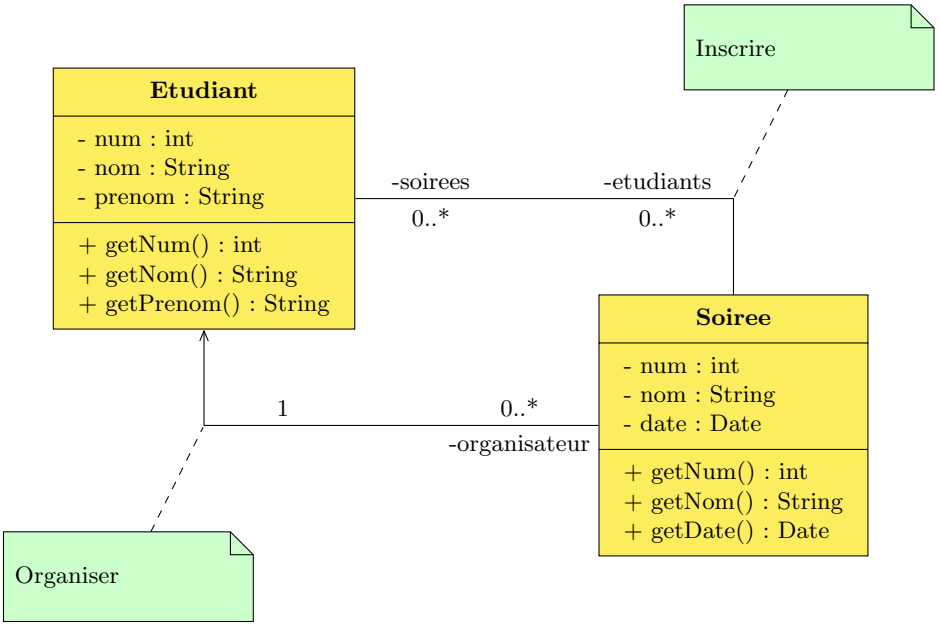
Lorsque chaque instance d'une classe contient une référence vers un objet du même type, on a afaire à une **relation reflexive**.



Chaque salarié possède un champ `responsable` référençant l'autre salarié qui est son supérieur hiérarchique.

Relations multiples

Deux classes peuvent être reliées par plusieurs relations différentes.



Un étudiant peut aussi bien avoir organisé une soirée que s'y être inscrit (voire les deux). Les deux relations n'ont pas la même nature et il serait une erreur de les fusionner.

2.1.5 Exercices

Exercice 1 - Livraisons

Construire un diagramme UML pour le cas Livraisons (vous utiliserez le MCD fait lors des cours précédents).

Exercice 2 - Arbre généalogique

Nous souhaitons mettre au point un logiciel permettant de représenter un arbre généalogique. C'est à dire listage des personnes, mariages et filiation. Nous tiendrons compte le fait que les parents peuvent être inconnus. Par contre, nous ne gérerons que les mariages hétérosexuels et nous ferons abstractions de certains pratiques en vogue comme la transexualité (si vous êtes sages nous ferons un TP là-dessus plus tard dans l'année). Représentez cette situation avec un MCD.

Exercice 3 - Bibliothèque

Nous souhaitons gérer une bibliothèque simple. Nous recenserons une liste d'ouvrages, avec pour chacun le titre et l'auteur (éventuellement plusieurs). On tiendra compte du fait que des ouvrages peuvent exister en plusieurs exemplaires, certains pouvant ne pas être disponibles pour le prêt (consultables uniquement sur place, ou détruits). Une liste d'adhérents devra être tenue à jour, les adhésions devant se renouveler une fois par an. Il devra être possible de vérifier qu'une adhésion est à jour (c'est-à-dire qu'elle a été renouvelée il y a moins d'un an), mais il ne sera pas nécessaire de connaître l'historique des renouvellements. Les adhérents peuvent emprunter jusqu'à 5 livres simultanément et chaque livre emprunté doit être retourné dans les deux semaines. On devra conserver un historique permettant de savoir quel abonné a emprunté quels livres, les dates d'emprunts et de retours.

Exercice 4 - Comptes en banque

Nous souhaitons gérer des comptes en banque. Une liste de comptes, avec numéro et libellé doit être tenue à jour. On souhaitera connaître pour chacun d'eux l'identité du propriétaire du compte (sachant qu'un compte peut dans certains cas appartenir à plusieurs personnes). Pour chaque compte, on conservera l'historique des modifications (virement, retrait, dépôt). On considérera comme un virement tout opération impliquant deux comptes et comme des retraits (ou dépôts) toute opération n'impliquant qu'un seul compte.

Exercice 5 - Inscriptions sportives

Une ligue sportive souhaite informatiser les inscriptions à des compétitions. Chaque compétition porte un nom et une date de clôture des inscriptions. Selon les compétitions, les candidats peuvent se présenter seul ou en équipe, sachant que le détail des équipes (liste des candidats et entraîneur) devra aussi être géré.

Exercice 6 - CMS

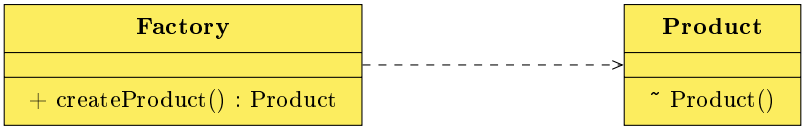
Nous souhaitons gérer un CMS (content management system). Un CMS est un logiciel permettant de gérer le contenu d'un ou plusieurs sites web. Chaque site porte un nom, est caractérisé par une URL, et est découpé en catégories, imbricables les unes dans les autres. Des utilisateurs sont répertoriés dans le CMS, et chacun peut avoir le droit (ou non) d'écrire dans un site web. Chaque utilisateur doit avoir la possibilité de publier des articles (ou des brèves) dans une catégorie d'un site web (pourvu qu'il dispose de droits suffisants dessus). Une brève est composée d'un titre et d'un contenu textuel. Un article, en plus de son titre et de son texte d'introduction, est constitué de chapitres portant chacun un nom et contenant un texte. Il va de soi qu'on doit avoir la possibilité pour chaque site de conserver l'historique de quel article (ou brève) a été publié par quel utilisateur. Réalisez un diagramme de classes modélisant cette situation.

2.2 Design patterns *(En construction)*

Les design Patterns sont des modèles permettant de résoudre de façon élégante des problèmes rencontrés fréquemment en programmation objet. Nous en étudierons quelques uns, je vous laisserai le soin de consulter de la documentation sur Internet pour vous familiariser avec les autres.

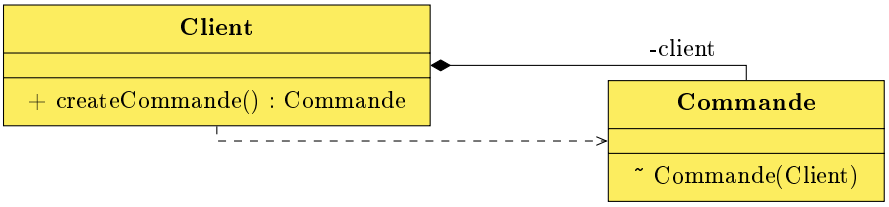
2.2.1 Factory

Lorsque la création d'un objet doit être contrôlé, on met l'opérateur **new** en **private** et on en confie l'appel à une méthode généralement **static** appelée une **factory**.



Exemple

Un client possède des commandes, mais une commande n'existe pas sans client. Lorsqu'une commande est crée sans qu'un client non **null** soit spécifié, lever une exception peut toujours être une solution, mais elle présente l'inconvénient d'alourdir le code. Une solution assez élégante consiste à réduire la visibilité du constructeur de commande et à créer une méthode non statique `createCommande` dans client. Cette fonction se charge de créer une commande dont le client est celui à partir duquel la fonction a été appelée.

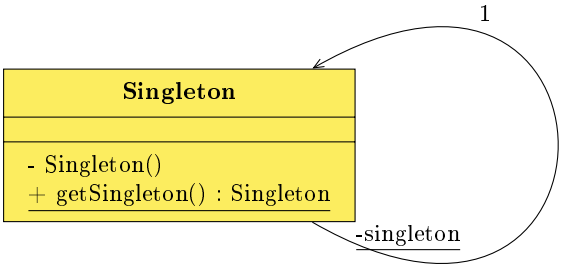


Exercice 1 - Clients et commandes

Créez une classe `Commande` et une classe `Client`. Vous réduirez la visibilité du constructeur de `Commande` et écrirez une méthode `createCommande`() dans la classe `Client`. *Vous pourrez utiliser une inner class.*

2.2.2 Singleton

Le **singleton** est une classe dont il ne peut exister qu'une seule instance. Il se code en utilisant une factory.



Exemple

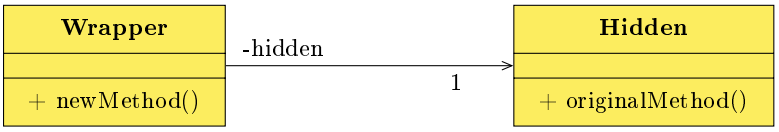
Lorsqu'un programme se connecte à une base de données, le fait qu'il dispose de plusieurs ouvertures simultanées vers la même base peut être une source de bugs très difficiles à trouver. Le singleton empêche l'utilisateur de la classe de créer directement une connexion et contrôle le nombre de connexions.

Exercice 2 - Singleton

Créez une classe `Singleton` vide.

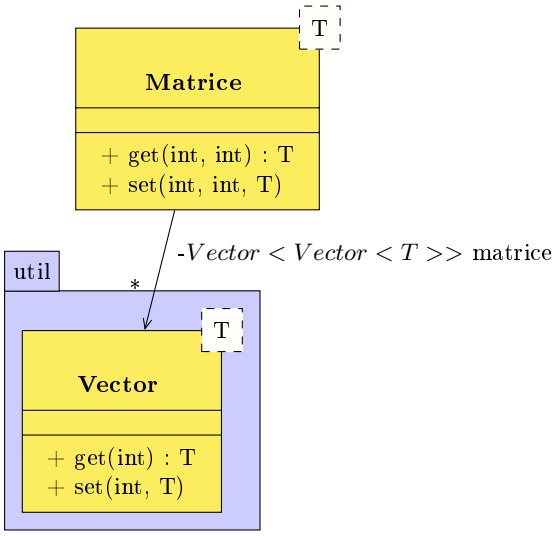
2.2.3 Wrapper

Le **wrapper** est une classe masquant une autre classe. Son but est de d'adapter d'utilisation d'une classe à un besoin, ou plus simplement de cacher la classe que vous avez choisi d'utiliser.



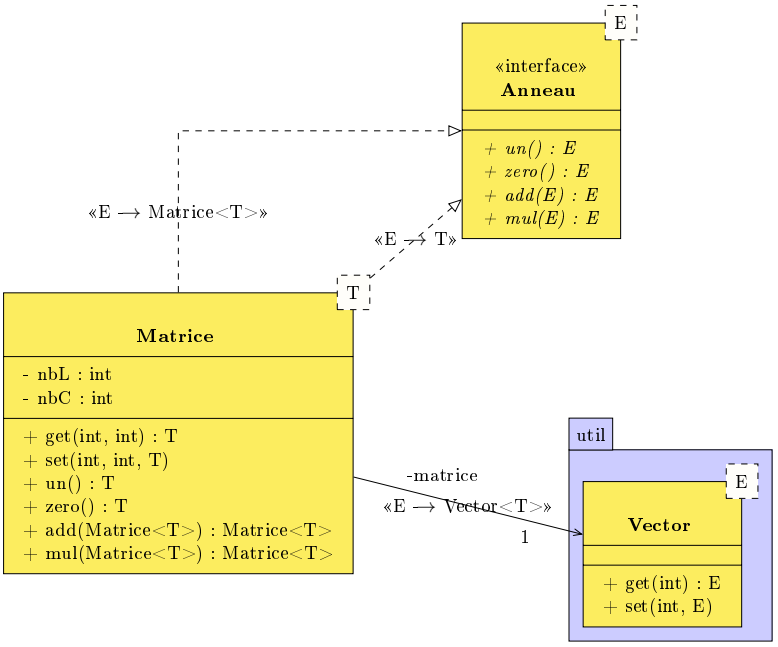
Exemple

Une matrice s'utilise avec deux indices, les fonctions permettant de manipuler des collections avec plusieurs indices sont peu commodes et source d'erreurs.



Exercice 3 - Integer trafiqué

Écrire une classe permettant de wrapper un Integer dans une implémentation de l'interface `Anneau<E>` présentée dans le diagramme ci-dessous.



Exercice 4 - Matrice

Écrire une classe permettant de gérer une matrice de type `T` à deux indices. Vous implémenterez les fonctions de somme et de produit et ferez en sorte que l'on puisse faire des matrices de matrices. Le diagramme de classes correspondant est présenté ci-dessus.

2.2.4 Adapter

L'**adapteur** est l'implémentation - quelques fois vide - la plus simple que l'on peut faire d'une interface. L'avantage qu'elle présente est qu'elle évite au programmeur d'écrire un grand nombre de méthodes vides ou contenant du code répétitif pour implémenter l'interface. L'inconvénient est que comme l'héritage multiple est interdit dans beaucoup de langages, une classe ne peut hériter que d'un adapteur à la fois.

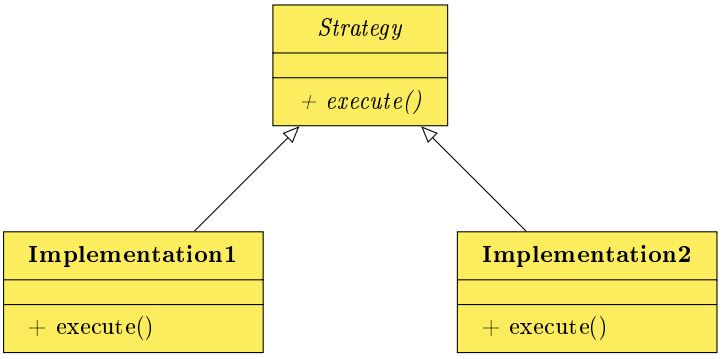
Attention, dans beaucoup de documentations, l'adapteur est assimilé à un wrapper (ce qui si vous y réflechissez bien, est tout à fait pertinent).

Exemple

Les interfaces comme `MouseMotionListener` contiennent beaucoup de méthodes et il est quelques fois pénible d'avoir plein de méthodes vides dans les classes d'implémentation. La classe abstraite `MouseMotionAdapter` contient une implémentation vide de `MouseMotionListener` et héritant de cette classe il est possible de ne redéfinir que les méthodes dont on a besoin.

2.2.5 Strategy

Lorsque l'on souhaite disposer de plusieurs algorithmes dans un projet et choisir lequel utiliser lors de l'exécution, ou lors de différentes étapes du projet, les diverses modification à opérer peuvent s'avérer particulièrement laides. Le pattern **Strategy** consiste à définir une classe mère représentant l'opération à effectuer et d'implémenter algorithme dans des classes filles.



Exemple

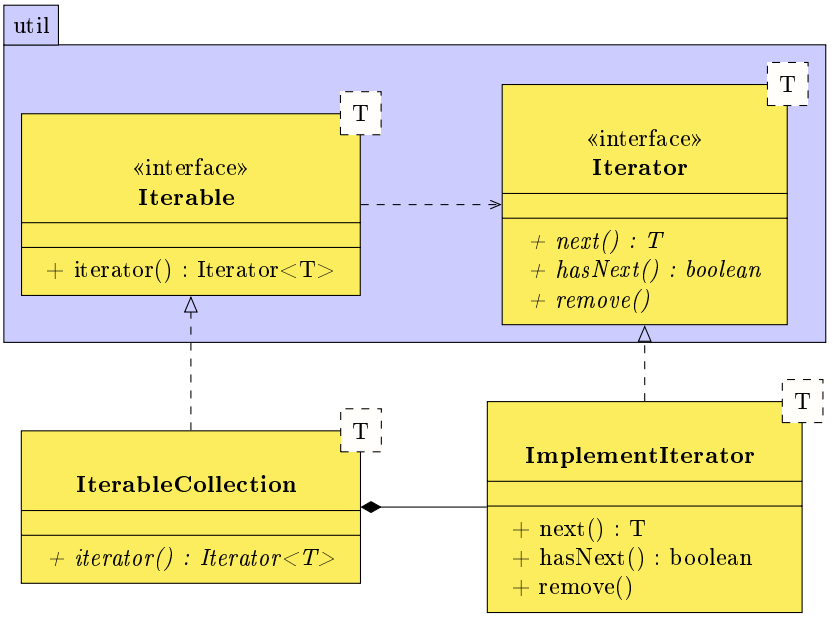
Si par exemple, vous souhaitez accéder à des données et que selon la situation le mode d'accès aux données peut changer (fichier, SGBD, accès réseau, etc.). Une façon élégante de procéder est de créer une classe abstraite chargée d'accéder aux données, et de créer une sous-classe par mode d'accès.

Exercice 5 - Carré

Créer une classe chargée de calculer le carré d'un nombre n. Vous implémenterez deux méthodes :
— Calculer directement n^2
— Utiliser le fait que le carré de n est la somme des n premiers nombres impairs.
Vous utiliserez une factory pour permettre à l'utilisateur de choisir la méthode de calcul.

2.2.6 Iterator

Les type abstraits de données comme les `Set`, `Map` ou encore `Tree` ne dispose pas les éléments dans une ordre aussi clair qu'un tableau ou une liste. **itérateur** est un moyen d'exploiter des collections avec le même code, en les parcourant avec une boucle **for**.

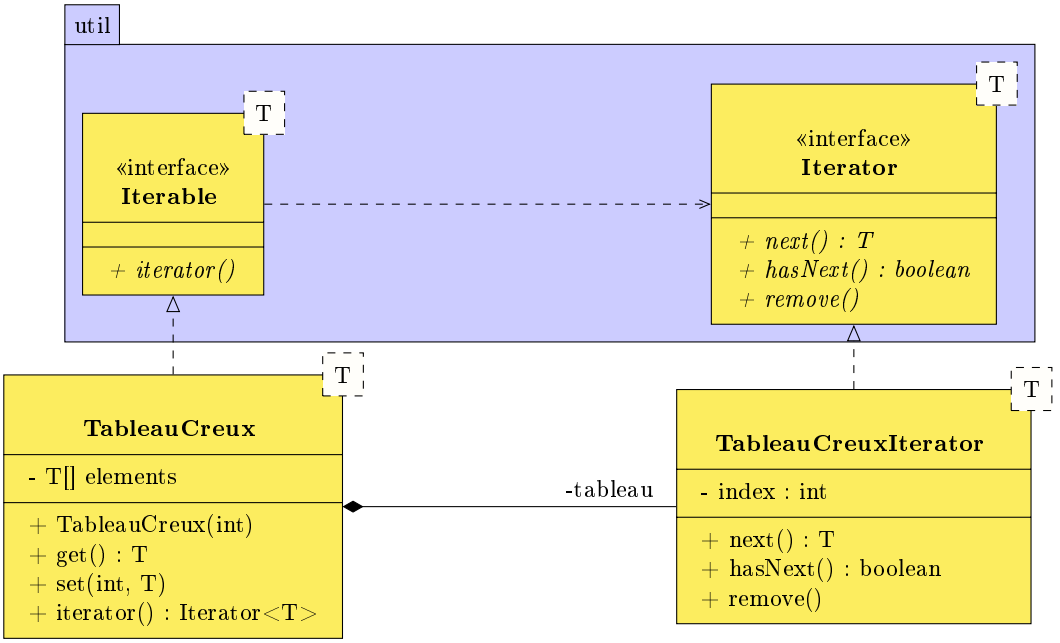


Exemple

En java, toute collection héritant de l'interface `Iterable<T>` peut se parcourir avec une boucle de la forme **for** (`T element : collection`)/*...*/.

Exercice 6 - Tableau creux

Créez un wrapper implémentant `Iterable<T>` et encapsulant un tablau. Votre itérateur retournera tous les éléments non **null** du tableau. *Dans une classe paramétrée en java, on ne peut pas instancier un tableau T[], vous êtes obligé de remplacer le tableau par une collection.*

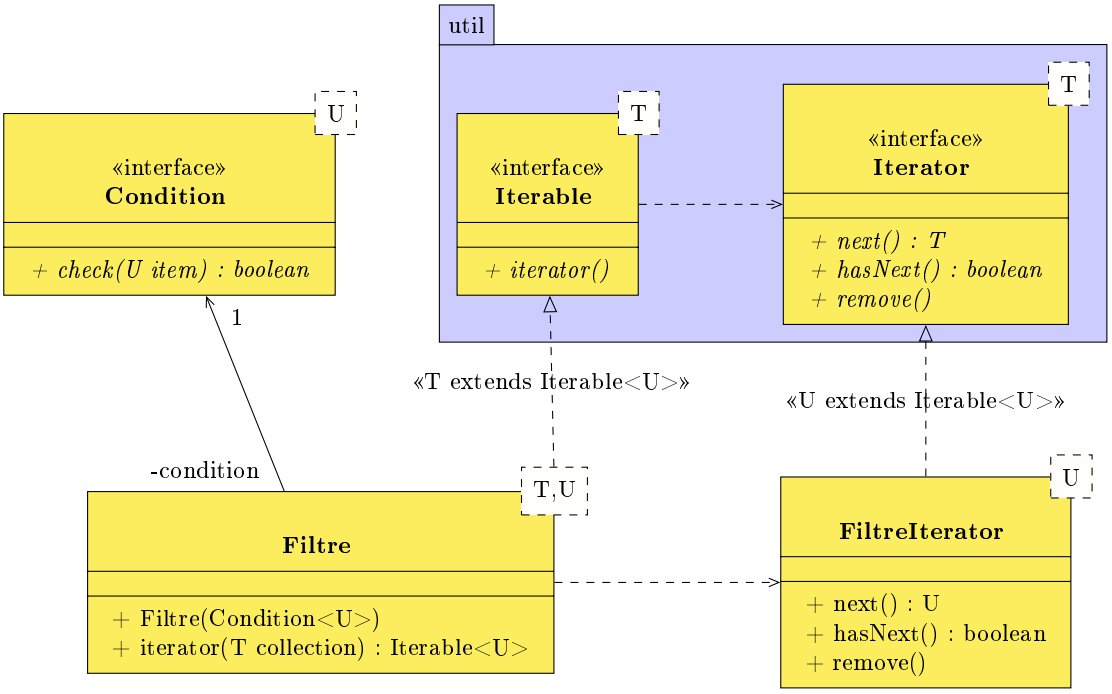


Exercice 7 - Matrice itérable

Rendre itérable la matrice de l'exercice sur les wrappers. *Ne réinventez pas la roue, utilisez l'itérateur fourni avec le type `Vector<E>`.*

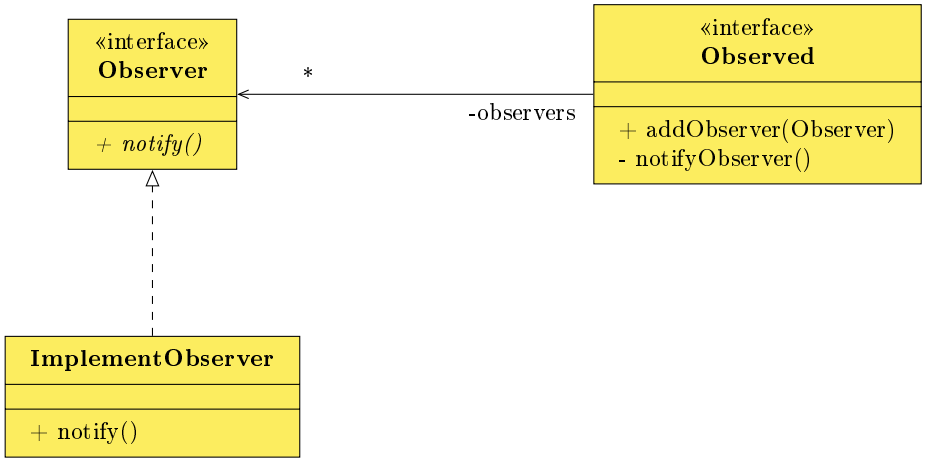
Exercice 8 - Filtre

Créez une interface `Condition<U>` contenant une méthode **boolean** `check(U item)`. Créez une classe filtre `Filtre<T, U>` contenant un `Condition<U>` et permettant de filtrer une sous-classe `T` de `Iterable<U>`. Créez une méthode `filtre(Collection<T> condition)` retournant un itérable contenant tous les éléments de la `collection` qui vérifient `check`.



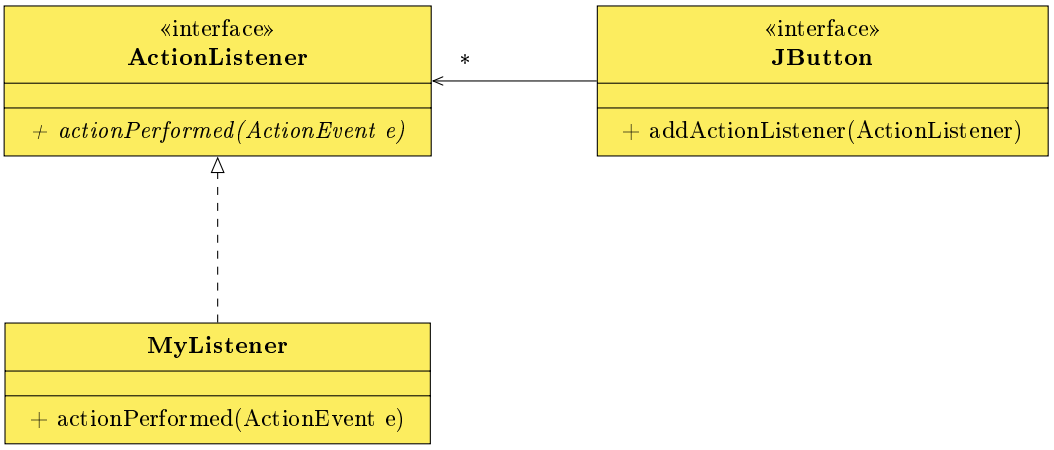
2.2.7 Observer

Le modèle en couche est souvent problématique lorsque deux classes dans des couches différentes ont besoin d'interagir. L'**Observer** permet d'éviter les références croisées et de simplifier considérablement le code. Un observer est un objet surveillant un autre objet, il contient une méthode appelée automatiquement lorsqu'une modification intervient sur l'objet surveillé.



Exemple

Dans les interfaces graphiques en Java, il est possible d'associer aux objets graphiques des `ActionListener`. L'interface `ActionListener` contient une seule méthode, `actionPerformed` qui est appelée automatiquement lorsqu'un utilisateur clique sur un objet graphique. De cette façon, une classe de la bibliothèque standard de java peut, sans que vous ayez à la modifier, exécuter du code que vous avez écrit.



Exercice 9 - Surveillance

Reprenez le tableau creux et créez un observateur qui va afficher toutes les modifications survenant dans le tableau.

2.2.8 Proxy

Le proxy cache un objet qui est généralement difficile d'accès (via une connexion réseau, une lecture sur un fichier, une base de données, etc.), et a pour but de masquer la complexité de cet accès, ainsi que d'appliquer un **lazy loading**. On parle de lazy loading si un objet est lu que lorsque quelqu'un le demande.

L'autre avantage du proxy est qu'il permet via des interfaces de changer la méthode de chargement (réseau, base de donnée, etc.).

Exemple

Si l'accès à un objet nécessite une connexion réseau qui peut prendre du temps, le proxy va le charger une seule fois, et seulement au moment où il sera demandé.

Exercice 10 - Somme de un à n

Écrivez une classe qui calcule la somme des nombres de 1 à n avec une boucle. Le calcul ne sera fait que la première fois que la fonction sera appelée et le résultat sera stocké dans un champ. Les fois suivantes, le résultat ne sera pas recalculé.

2.2.9 Decorator

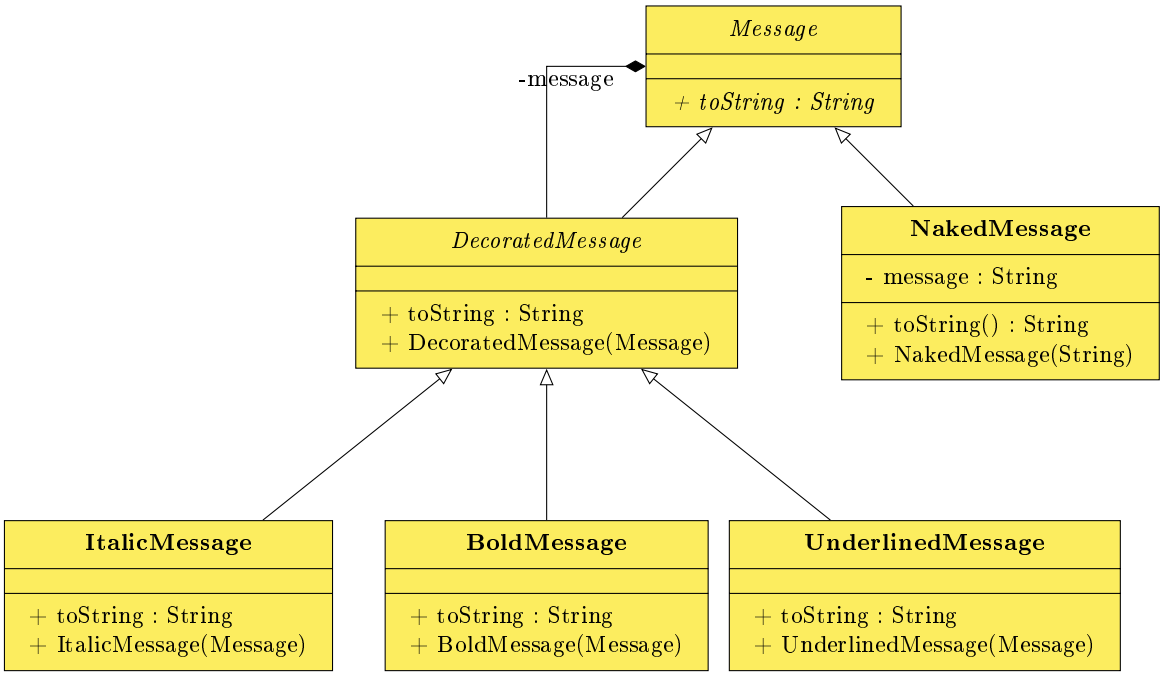
Le **decorator** est un moyen élégant d'éviter l'héritage multiple sans pour autant dupliquer de code. Rappelons que l'héritage est en général déconseillé (risque de conflit entre les noms de méthodes), et dans certains langages (Java) interdit.

Un décorateur est un wrapper qui contient un objet dont le comportement est modifié, la puissance de ce pattern vient du fait qu'un décorateur peut contenir un autre décorateur, il est ainsi possible de placer les décorateurs en cascade.

L'avantage du décorateur est qu'il permet de créer des intersections entre sous-classes et de remplacer l'héritage multiple. Les inconvénients sont de taille : le fait qu'une liste chaînée de décorateurs précède un objet est d'une part une source de bugs pour les programmeurs inexpérimentés, et par ailleurs le compilateur ne peut plus contrôler le type des objets de façon aussi précise (nous verrons pourquoi dans les exercices).

Exemple

Si l'on souhaite représenter des messages affichables en gras, en italique ou en souligné, trois sous-classes ainsi que les multiples combinaisons permettant de recouper ces sous-classes sont nécessaires. En utilisant trois wrappers on obtient la possibilité de les combiner.



Exercice 11 - Messages

Programmez l'exemple précédent.

Exercice 12 - Choix de dessert

Implémentez la facturation d'un dessert. Il est possible de choisir des boules de glaces (vanille, fraise et café), à 1 euro chacune, sachant qu'on peut mettre dans la même coupe des parfums différents. L'utilisateur peut aussi ajouter de la chantilly (pour 0.5) et le nappage (sauce caramel ou chocolat) est à 0.75 euros.

Annexe A

Énonces des cas étudiés

A.1 Secrétariat pédagogique

Nous souhaitons gérer un secrétariat pédagogique. Nous recensons les étudiants en utilisant leurs noms, prénoms et adresse. Des formations sont organisées en modules, qui eux-mêmes sont répartis sur des semestres. Un étudiant ne peut être inscrit que dans une formation à la fois, et un module est affecté à un seul semestre dans une formation. On tiendra compte du fait qu'un étudiant peut redoubler ou suspendre sa formation en prenant des congés.

A.2 Chaîne d'approvisionnement

Nous souhaitons gérer une chaîne d'approvisionnement. Nous tenons une liste de produits et une liste de fournisseurs. Certains produits peuvent être proposés par plusieurs fournisseurs, à des prix qui peuvent être différents. Nous garderons l'historique des livraisons en mémoire, avec pour chacune d'entre elle, le fournisseur qui l'a effectué, la date de livraison ainsi que le détail des produits livrés.

A.3 Arbre Généalogique

Nous souhaitons mettre au point un logiciel permettant de représenter un arbre généalogique. C'est à dire listage des personnes, mariages et filiation. Nous tiendrons compte du fait que les parents peuvent être inconnus. Par contre, nous ne gérerons que les mariages hétérosexuels et nous ferons abstraction de certains pratiques en vogue comme la transexualité (si vous êtes sages nous ferons un TP là-dessus plus tard dans l'année).

A.4 CMS

Nous voulons gérer un CMS (Content Management System). Un CMS est un logiciel permettant de gérer le contenu d'un ou plusieurs sites web. Chaque site porte un nom, est caractérisé par une URL, et est découpé en catégories, imbricables les unes dans les autres. Des utilisateurs sont répertoriés dans le CMS, et chacun peut avoir le droit (ou non) d'écrire dans un site web. Chaque utilisateur doit avoir la possibilité de publier des articles dans une catégorie d'un site web (pourvu qu'il dispose de droits suffisants sur le site). Un article, en plus de son titre et de son texte d'introduction, est constitué de chapitres portant chacun un nom et contenant un texte. Il va de soi qu'on doit avoir la possibilité pour chaque site de conserver l'historique de quel article a été publié par quel utilisateur.

A.5 Bibliothèque

Nous souhaitons gérer une bibliothèque simple. La base de données devra recenser une liste d'ouvrages, avec pour chacun le titre et l'auteur (éventuellement plusieurs). Vous tiendrez compte du fait que des ouvrages peuvent exister en plusieurs exemplaires, certains pouvant ne pas être disponibles pour le prêt (consultables uniquement sur place, ou détruits). Une liste d'adhérents devra être tenue à jour, les adhésions devant se renouveler une fois par an. Il devra

être possible de vérifier qu'une adhésion est à jour (c'est-à-dire qu'elle a été renouvelée il y a moins d'un an), mais il ne sera pas nécessaire de connaître l'historique des renouvellements. Les adhérents peuvent emprunter jusqu'à 5 livres simultanément et chaque livre emprunté doit être retourné dans les deux semaines. On devra conserver un historique permettant de savoir quel abonné a emprunté quels livres, les dates d'emprunts et de retours.

A.6 Forum

Nous souhaitons gérer un forum. Un forum est constitué de catégories, une catégorie pouvant elle-même être contenue dans une autre catégorie. Chaque catégorie peut contenir des conversations, elles-même constituées de messages, un message pouvant etre la réponse à un autre message. Les utilisateurs doivent s'inscrire pour publier, et certains d'entre eux peuvent être nommés modérateurs par les administrateurs.

Les utilisateurs disposent aussi d'une messagerie privée, permettant de faire des conversations à deux. Les utilisateurs peuvent aussi contacter les modérateurs en messagerie privée, tous les modérateurs peuvent alors répondre dans la conversation.

Les utilisateurs peuvent signaler des contenus inappropriés aux modérateurs. Il sera important de savoir, en plus du message qui a été signalé, quel est l'utilisateur qui a effectué le signalement.

Il faut que les utilisateurs connectés puissent voir quels sont les messages qu'ils ont lu et ceux qu'ils n'ont pas lu. Le fait de savoir quels messages n'ont pas été lu permettra d'envoyer les notifications aux utilisateurs.

A.7 Covoiturage

Nous allons étudier la modélisation d'une plate-forme de co-voiturage.

Des utilisateurs peuvent être :

— des conducteurs
— des passagers
— des modérateurs de la société de co-voiturage.
— ou encore prendre plusieurs rôles à la fois...

On mémorisera leur nom, prénom, mail, téléphone, coordonnées bancaires.

Les utilisateurs peuvent en tant que conducteurs s'inscrire et donner des informations sur leurs véhicules (marque, modèle, immatriculation).

Les conducteurs peuvent créer des trajets (point et date de départ, d'arrivée, véhicule, nombre de places disponibles, étapes).

Les points de départ, d'arrivée, et les étapes, sont :

— soit des points covoiturage prédéfinis,
— soit des adresses utilisant des villes prédéfinies.

Le conducteur doit indiquer pour chaque étape :

— l'heure à laquelle il doit passer
— le coût de prise en charge d'un passager

Les utilisateurs, en tant que passagers peuvent :

— rechercher des trajets,
— contacter le conducteur avec une messagerie interne,
— faire une réservation sur un trajet ou une partie du trajet.
— annuler leur réservation
— valider leur trajet, ce qui déclenche le paiement.

Une fois un trajet validé, les passagers peuvent noter le conducteur et le conducteur peut noter ses passagers. Une note est constituée :

— d'un nombre d'étoiles (1 à 5)
— d'un commentaire

Les commentaires doivent être validés par un modérateur. Un usager ne peut pas voir un commentaire le concernant tant qu'il n'a pas lui-même rédigé un commentaire. Si au bout de 15 jours il n'a pas répondu, alors le commentaire est publié et l'usager concerné ne peut pas répondre.

La note moyenne d'un utilisateur est toujours publiée sur son profil.