UML

Modélisation avec UML

Rédigé par

David ROUMANET
Professeur BTS SIO

Changement

Date	Révision
Août 2020	Création

Sommaire

A Modéliser avec UML

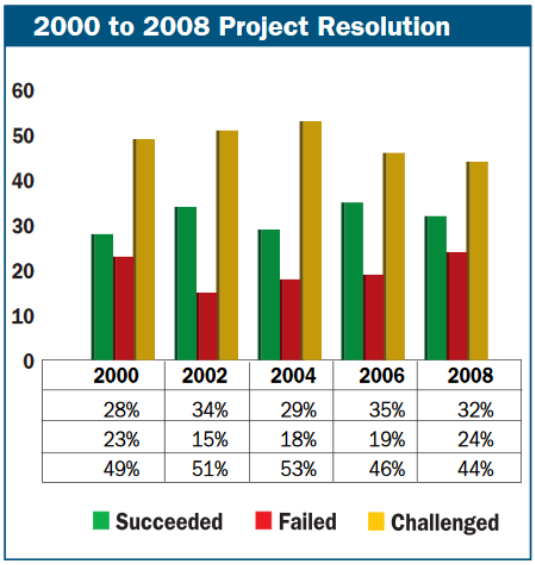
A.1 Problématique

Nous savons que la modélisation est nécessaire pour créer une application manipulant des données. En utilisant la programmation orientée objet, il devient nécessaire de considérer la modélisation utilisant des objets, comme MEA ou UML.

Savoir écouter un client et transformer ses besoins en fonctionnalités dans une application, ne peuvent se faire sans un certain formalisme.

La preuve, 83 % des projets informatiques sont abandonnés, entraînant pertes de temps et d'argent et mécontentement des clients. Cette statistique provient du groupe Standish[1] en 1994. Chaque année le rapport est actualisé et on constate que seulement un tiers des projets est un succès.

Jusqu'où modéliser pour coder une application orientée objet ? Par quoi commencer ?



A.2 Les diagrammes UML

En tant qu'informaticien, vous vous devez de savoir utiliser les diagrammes UML, pour permettre à vos projets de réussir. L'ordre d'utilisation et la bonne utilisation vous évitera parfois des heures de travail inutiles.

A.3 Objectifs du cours

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

Mémoriser	Savoir lire les symboles des diagrammes UML
Comprendre	Utiliser correctement chaque diagramme (utilisation, séquence, classe)
Appliquer	Mettre en œuvre les diagrammes sur des cas concrets
Analyser	Analyser un cahier des charges
Évaluer	–
Créer	–

1 Standish Group : https://standishgroup.com/

B UML et ses diagrammes

Nous avons vu dans le cours précédent, les différentes modélisations et particulièrement, la méthode Merise et son formalisme MEA.

UML n'a pas de méthodologie à proprement parler, toutefois, les diagrammes permettent de matérialiser une certaine chronologie. D'abord il y a les deux ensembles :

- Les diagrammes de structures (statiques) sont proches de la réalisation technique.
- Les diagrammes de comportements[2] (dynamiques) décrivent le fonctionnement.

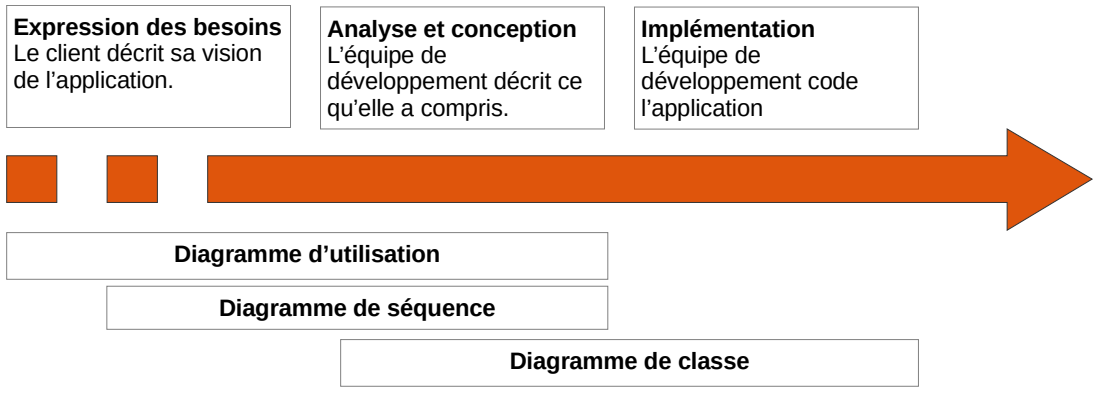
Nous allons utiliser les diagrammes suivants

- Comportement
  - Les diagrammes de cas d'utilisation (permettent de régir les interactions avec les acteurs)
  - Les diagrammes de séquences (permettent de visualiser les enchaînements d'actions)
- Structure
  - Les diagrammes de classes (permettent de conceptualiser et modéliser les données et les classes)

B.1 Démarche

Lors de l'expression des besoins, le client doit décrire le fonctionnement de l'application, ce qu'elle permet de faire : c'est donc un ou plusieurs diagrammes de comportement qui doivent être utilisés.

Lorsque toutes les fonctionnalités ont été décrites, il faut analyser et concevoir la structure des informations. Les diagrammes de structure s'imposent naturellement.



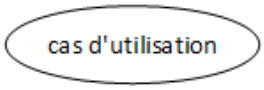
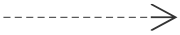
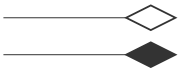
Au milieu des diagrammes, on trouvera également la création des scénarios et le maquettage des interfaces. UML n'ayant pas de méthodologie, il convient parfois d'adapter les autres méthodes[3].

2 De l'anglais, "behavior"
3 Exemple, RUP ou 2RTUP ou Agile : http://formation-uml.fr/et-apres/methode/

B.2 Les diagrammes d'utilisation (use case)

B.2.1 Symboles

La lecture d'un diagramme UML s'appuie sur quelques éléments graphiques. Tous les symboles n'apparaissent pas dans tous les diagrammes, voici les symboles pour les diagrammes d'utilisation :

symboles	signification
	**Acteur** : c'est le rôle qui pourra interagir avec le système. On ne dessine qu'un seul rôle, même s'il représente plusieurs personnes *Ex : rôle = client de banque*
	**Cas d'utilisation** : c'est une opération simple qui nécessitera un développement dans le système. Les acteurs sont liés à ces cas. *Ex : cas d'utilisation = retirer de l'argent*
	Association : c'est le lien entre un acteur et un cas d'utilisation. Il y a autant de liens que de cas d'utilisation, on ne peut créer une association directe avec le système. *Ex : rôle "client de banque" associé avec "retirer de l'argent"*
	**Généralisation** : cette association permet de montrer une généralisation (un héritage) vers l'élément ayant plus d'actions. La flèche est creuse. *Ex : rôle "client banque" et une généralisation d'un simple "porteur CB"*
	**Étendre** ou **inclure** : cette association indique si un élément dépend d'un autre. La pointe de la flèche est vide (ce n'est pas un triangle) et la flèche est en pointillé. On écrit dessus <<extend>> ou <<include>>
	**Agrégation** : indique si un élément est constitué de plusieurs autres (démontables) **Composition** : indique si un élément est composé de plusieurs autres (soudés)
	**Système** : le système contient les cas d'utilisation. C'est lui qui sera réalisé par les développeurs. Les acteurs sont forcément à l'extérieur du système.

B.2.2 Formalisme

Le diagramme d'utilisation met en scène les acteurs de l'application et leur relation avec le système, c'est-à-dire, préciser les actions possibles.

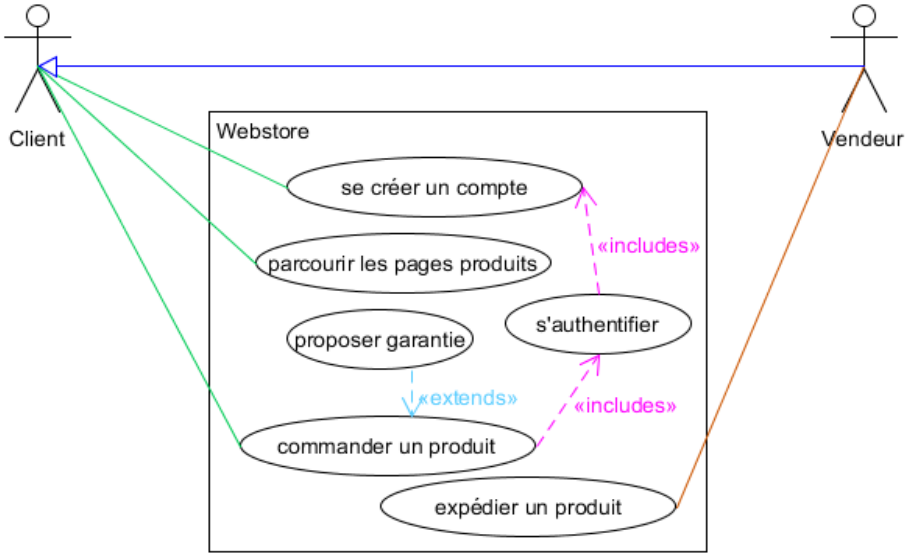
Dans cette étape, le client raconte les différents scénarios qu'un utilisateur peut effectuer, il n'y a pas d'indication sur les données, mais seulement sur les actions. Il est possible de relier les actions entre elles, par les mots clés « inclure » et « étendre ».

En revanche, il n'y a pas de notions temporelles, et les actions ne doivent pas s'enchaîner les unes aux autres.

D'autre part, les acteurs représentent des rôles (le rôle « client » signifie n'importe quelle personne cliente) tandis qu'il est possible d'avoir une généralisation entre deux rôles (un vendeur peut effectuer les mêmes actions qu'un client).

B.2.3 Exemple

Voici un exemple simple de diagramme d'utilisation.



Un client peut (lignes vertes) :

- se créer un compte
- parcourir les pages produits
- commander un produit ; dans ce cas…
  - il doit obligatoirement s'authentifier ; s'il n'a pas de compte il doit..
    - se créer un compte

Un vendeur peut faire la même chose qu'un client (généralisation en bleue) et (ligne orange) :

- expédier un produit

Le diagramme d'utilisation doit être exhaustif sur les actions, mais il ne doit pas contenir de détails précis (sur l'interface graphique, sur la charte de couleur, sur les différents scénarios…), ni être classé de manière chronologique.

La différence entre une relation obligatoire (include) et une extension (extends) tient compte du sens des flèches :

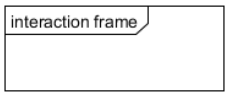
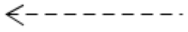
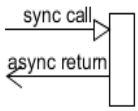
- dans la relation obligatoire, la flèche part d'une action et pointe vers l'action nécessaire à la réalisation de la première action.
- Dans la relation d'extension, il s'agit d'un service (une action) qui va faire l'action initiale : il s'agit d'une option (par exemple, proposer une garantie étendue à un produit).

C Les diagrammes de séquences

Le diagramme de séquence permet de valider les étapes utiles pour la réalisation d'un cas d'utilisation. Il indique l'ordre des opérations et des réponses.

C.1.1 Symboles

Les diagrammes de séquences peuvent reprendre le symbole de l'acteur mais généralement, ce sont d'autres symboles qui sont utilisées.

symboles	signification
	**Trame d'interaction** : c'est le cadre qui contient le cas d'utilisation à produire. Peut également servir à l'intérieur du cas d'utilisation pour différencier un ensemble d'actions (**par**allèle, **alt**ernative, **loop** = boucle, **assert**ion…)
	**Acteur** ou **Instance** : ce sont les différents éléments d'interactions. Pour les instances, il est possible de les créer (une flèche pointe alors le rectangle à la hauteur de la création dans le processus) ou de les détruire (une croix est placée à la fin du trait vertical).
	Message synchrone (nécessite un délai contrôlé)
	Message asynchrone (la plupart des actions d'un utilisateur)
	Message réponse, création d'objet
	Les rectangles représentent les barres d'activation (ou focus de contrôle). Il s'agit de représenter une durée non négligeable pour un processus (un ensemble d'opérations dans le même bloc de temps). *Ex : communication à un client avec un délai avant action.*

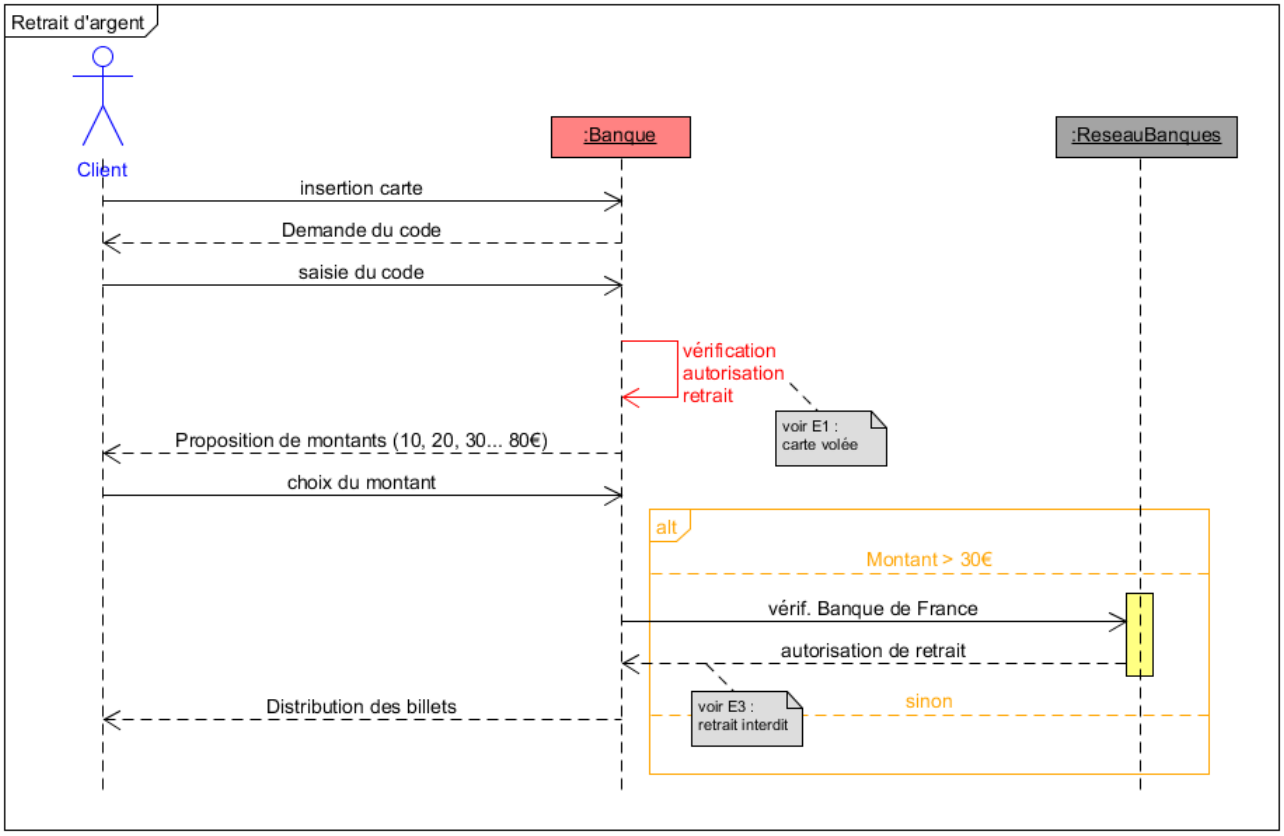
Ce diagramme est beaucoup plus précis, car il décrit les étapes de réalisation d'une action. De plus, on peut y faire apparaître les conditions de réalisation. Un diagramme de séquence peut renvoyer vers un autre diagramme de séquence.

C.1.2 Formalisme

Ce diagramme sert à visualiser les interactions du système étudié. L'axe vertical représente le temps, permettant ainsi de dérouler séquentiellement les actions lors d'un cas d'utilisation.

C.1.3 Exemple

L'exemple ci-dessous représente le cas d'utilisation du retrait d'argent dans un guichet automatique, avec les étapes internes et les différentes instances.



Le graphique se lit de haut en bas pour le temps, et de gauche à droite pour les interactions avec les acteurs.

Entre le choix du montant et la demande du code de carte bleue au client, un processus particulier vérifie que le distributeur pourra honorer la demande.

Enfin, une trame d'interaction est ajoutée en fonction du montant : deux scénarios y sont présents, selon la condition de montant supérieur à 30 €.

Il faut associer des actions et des réactions (réponses) pour chaque étape. Dans l'exemple ci-dessus, le client initie le dialogue et à la fin, reçoit de l'argent.
Si l'autorisation n'est pas acceptée, il faut consulter le diagramme E3 de retrait interdit.

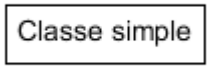
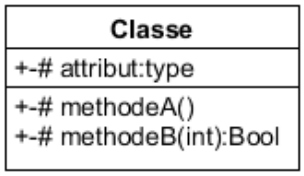
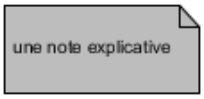
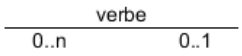
D Les diagrammes de classes

Ces diagrammes sont les plus connus des développeurs, car ils représentent facilement des classes d'objets : il existe même des outils capables de générer un code en fonction du diagramme.

Cette phase de structure est utilisé pour représenter les données et (en programmation) leurs traitements. Il faut donc prévoir toutes les relations entre les classes, les portées, les types et les méthodes.

D.1.1 Symboles

Les symboles peuvent sembler moins nombreux mais en réalité, ils ont plus de variations.

symboles	signification
	**Classe simple** : il s'agit de montrer l'existence d'une classe mais sans la décrire. Un nom de classe commence toujours par une majuscule.
	**Classe** : c'est le schéma traditionnel. • La partie 1 contient le nom de la classe. Peut-être suivi de { propr.} pour indiquer une propriété. Ex : {Abstraite} • La partie 2 contient les attributs. Les symboles + (public), - (privé) et # (protégé) représentent la portée des attributs. • La partie 3 contient les méthodes avec les mêmes symboles.
	**Note** : ce symbole permet d'ajouter des commentaires sur le schéma UML.
	**Association** : il en existe de nombreuses (flèches pleines, losanges, flèches creuses, etc.) mais les associations déterminent les liens des classes entre elles. En UML, les associations peuvent recevoir une notion de multiplicité (les nombres de part et d'autres des associations).

On considère que les associations sont de plusieurs types :

Association simple (connaît...)	**Héritage** (de la classe pointée)	**Agrégation** (peut appartenir)	**Composition** (au sens compose...)
			
Ce sont les lignes les plus fréquentes.	La flèche vide indique la classe mère ("hérite de...") ce n'est pas une association	Le losange est collé à la classe qui utilise les autres classes. Une [voiture]<>-----[moteur] montre que le moteur compose la voiture	

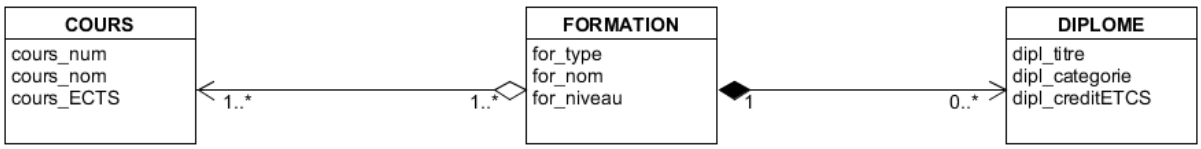
Outre les symboles, une écriture en italique de la classe peut signifier que celle-ci est abstraite (ou bien être précédé du stéréotype <<abstract>>).

De même un attribut souligné signifie un attribut de classe (soit statique, soit global).

D.1.2 Agrégation

Les agrégations sont de deux types :

- l'**agrégation** (une formation scolaire est composée de cours). L'élément composé peut être supprimé, sans supprimer la classe qui le compose (on peut supprimer un objet de la classe FORMATION sans supprimer les objets de la classe COURS)
- la **composition** est une agrégation forte (un diplôme dépend d'une formation). Dans ce cas, la suppression de la formation supprime le diplôme correspondant (l'objet DIPLOME n'aurait alors aucun sens).



D.1.3 Formalisme

Le formalisme du diagramme de classe dépend de son utilisation : pour représenter les informations d'une base de données ou bien pour coder une application.

D.1.3.a Modélisation d'une base de données

Une base de données n'utilise que les attributs et les relations (MCD) et éventuellement leur type (MPD).

Un diagramme de classe pour une base de données ne devrait pas contenir de flèche d'héritage, de navigabilité, de portée ou de méthodes.

En ce sens, le formalisme UML n'apporte pas plus d'information que le Modèle Entité-Association (MEA).

D.1.3.b Modélisation d'un code pour une application

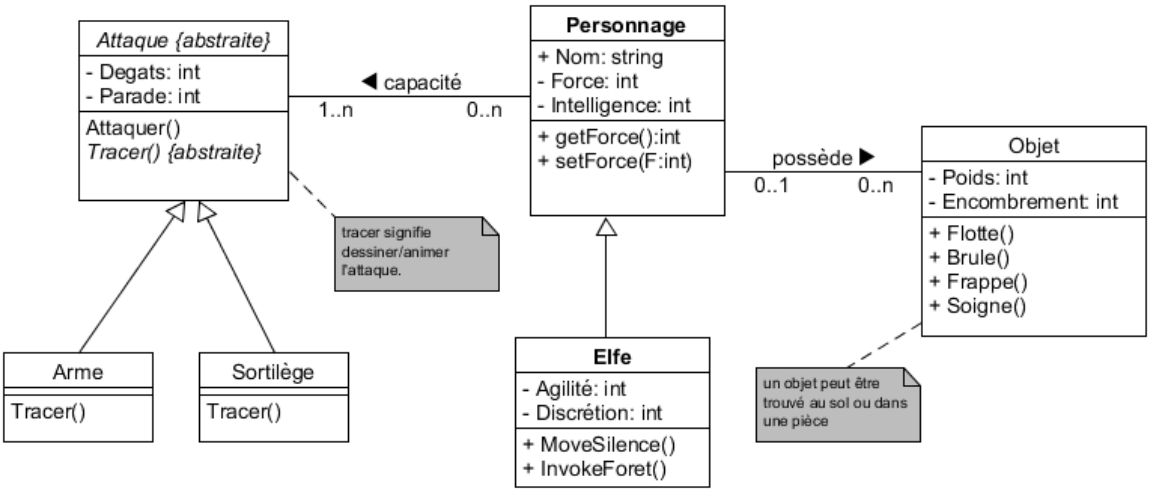
Dans le cas d'une modélisation ayant pour objectif la génération d'un code, le diagramme de classes peut contenir de nombreuses informations complémentaires :

- Attributs
  - Type (tous les types connus par le langage, int, bool, float, date, etc. Y compris une classe)
  - Portée (le signe détermine la portée : Privée (-), protégée (#) ou publique (+))
- Relations
  - Navigabilité (*voir chapitre suivant*)
- Traitements
  - Méthode (formalisme décrivant le nom et les attributs utilisés par ces méthodes)

Cette modélisation est abordée et expliquée au prochain chapitre.

D.1.4 Exemple

L'exemple suivant formalise une application pour un jeu de rôle (JDR).



Ce diagramme peut aussi servir dans la création de la base de données associée : il y aura une table **Personnage**, une table **Objet** et une table **Attaque** (car Arme et Sortilège qui hérite d'attaque, ne modifient pas le nombre d'attributs).

La table **Elfe** pose le problème des évolutions futures : existera-t-il d'autres catégories d'êtres ? Vaut-il mieux mettre les attributs Agilité et Discrétion dans la table Personnage (et les mettre à 0 s'ils ne sont pas utilisés) ?

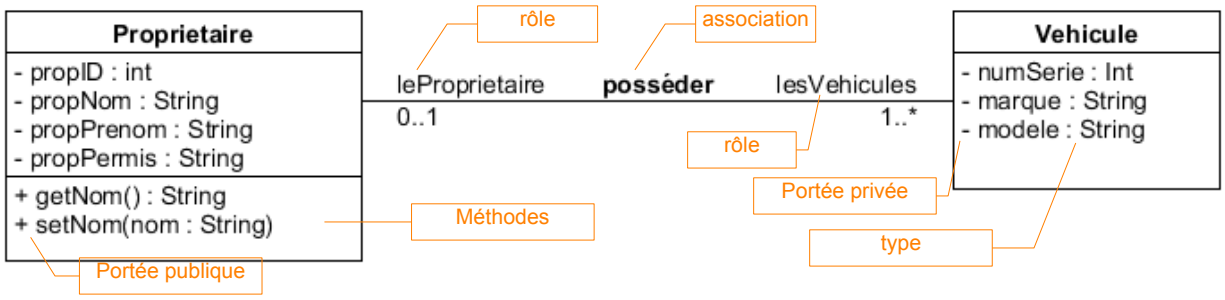
Enfin, les relations multiples (x..n) nécessitent une table, il faudra donc une table **Posséder** et **Capacité**.

E Codage et diagramme de classes

Voici quelques éléments pour comprendre l'intérêt du formalisme UML en programmation.

E.1.1 Multiplicités dans un code

Les multiplicités représentent soit un attribut, soit une liste d'attribut. Le diagramme de classe suivant est parfaitement programmable dans la plupart des langages :



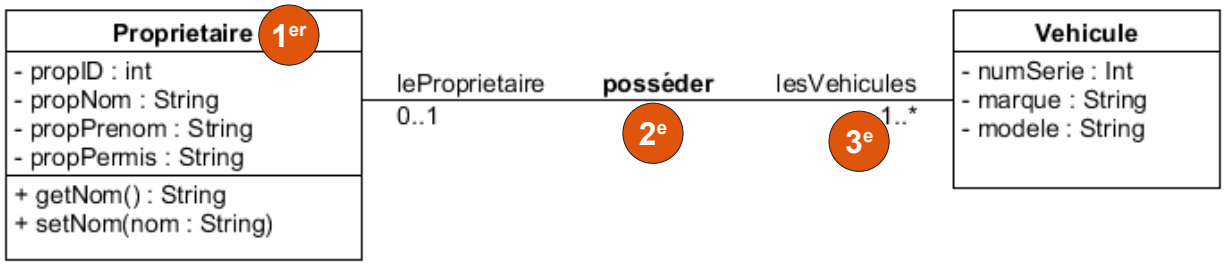
Le schéma fait maintenant apparaître **les portées** (un symbole moins signifie "privé", un symbole plus signifie "public"), **les types** mais aussi **les méthodes**.

Le point important, est l'apparition des **rôles** "leProprietaire" et "lesVehicules" qui permettent de nommer les attributs qui apparaîtront dans la classe. Un rôle contenant un pluriel (les ...) prépare une **liste** (ou collection).

Enfin, la relation n'a pas été fléchée, ce qui implique que le véhicule connaît son propriétaire, tandis que le propriétaire connaît… la liste de ses véhicules.

Pour rappel, le sens de lecture des multiplicités d'UML se lit comme suit :

Un propriétaire possède un ou plusieurs véhicules.



À l'inverse, on peut dire :

Un véhicule est possédé par zéro (encore au magasin) ou un propriétaire

Le code ci-après représente cette classe.

E.1.2 Code (en C#)

Le code peut parfois être créé par des outils spécialisés ou des extensions. Il ressemblera à ceci :

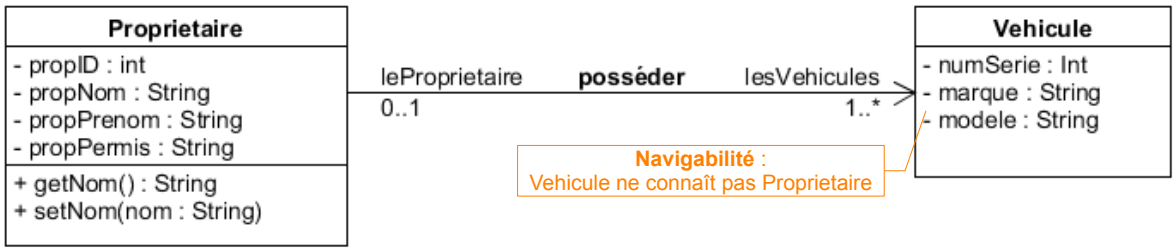
```
public Class Proprietaire {
    private int propID;
    private String propNom;
    private String propPrenom;
    private String propPermis;
    private List<Vehicule> lesVehicules = new List<Vehicule>();

    public String getNom() {
        return this.propNom;
    }
    public void setNom(String nom) {
        this.propNom = nom;
    }
}

public Class Vehicule {
    private int numSerie;
    private String marque;
    private String modele;
    private Proprietaire leProprietaire;
}
```


Il est important de voir que le rôle devient un attribut dans les classes :

- un rôle ayant une multiplicité de 0 à 1 sera un objet
- un rôle ayant une multiplicité supérieure à 1 sera une collection d'objets

E.1.3 Navigabilité dans un code

En modifiant légèrement le schéma, on va modifier le code correspondant : la relation peut être fléchée pour indiquer qu'elle n'est pas bidirectionnelle.



Dans ce cas, la relation indique que les propriétaires connaissent leurs véhicules et que **les véhicules ne connaissent pas le propriétaire**.

Le code change légèrement :

```
public Class Proprietaire {
    private int propID;
    private String propNom;
    private String propPrenom;
    private String propPermis;
    private List<Vehicule> lesVehicules = new List<Vehicule>();

    public String getNom() {
        return this.propNom;
    }
    public void setNom(String nom) {
        this.propNom = nom;
    }
}

public Class Vehicule {
    private int numSerie;
    private String marque;
    private String modele;
    private Proprietaire leProprietaire;
}
```

Navigabilité :
Vehicule ne connaît pas Proprietaire

En effet, en supprimant l'attribut "leProprietaire" dans la classe "Vehicule", il devient impossible de retrouver qui possède l'objet.

E.1.4 Liaison faible et code

Pour terminer les différences sur les variations de la relation, le schéma suivant aura une signification différente du cas précédent :

Ici, les pointillés indiquent une liaison dite **faible** ! Elle n'est pas permanente, et indique donc que le propriétaire n'aura pas constamment la liste de ces véhicules.

Le code pourrait ressembler à ceci (observez l'emplacement de la création de la liste).

```
public Class Proprietaire {
    private int propID;
    private String propNom;
    private String propPrenom;
    private String propPermis;
    public String getNom() {
        return this.propNom;
    }
    public void setNom(String nom) {
        this.propNom = nom;
    }
    public Vehicule getMesVehicules() {
        private List<Vehicule> lesVehicules = new List<Vehicule>();
        ...
        return lesVehicules;
    }
}

public Class Vehicule {
    private int numSerie;
    private String marque;
    private String modele;
}
```

Ici, la liste est créée à l'intérieur d'une méthode, ce qui signifie qu'à la fin de cette méthode, la liste sera détruite.

F Annexes

F.1 Sources complémentaires

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