CONCEPTION des SYSTÈMES d'INFORMATION

UML

Cours n°2 - 1

LES DIAGRAMMES ORGANIQUES « OBJET »

CLASSES, OBJET, SEQUENCE, COLLABORATION

Epitech 3

Bertrand LIAUDET

SOMMAIRE

Première édition : automne 2007
Deuxième édition : novembre 2008
Troisième édition : octobre 2009

LES DIAGRAMMES ORGANIQUES « OBJET »
CLASSES, OBJETS, SEQUENCE, COLLABORATION

Il est facile de décrire la méthode encore que son application exige à coup sûr savoir et pratique.

	Point de vue	Diagramme UML	
ANALYSE FONCTIONNELLE	Statique – non objet	Cas d'utilisation	
	Dynamique – non objet	Séquence	
		Activités	
ANALYSE DES DONNEES	Statique – non objet	MEA équivalent Classes	**DONNEES**
	Statique – objet	Classes-métier	
ANALYSE ORGANIQUE	Statique – objet	Classes	
		Objets	
	Dynamique - objet	Séquence	
		Collaboration	
	Dynamique Plutôt non objet	Etats-transitions	
		Activités	

La programmation fonctionnelle suit l'analyse fonctionnelle qui est calquée sur l'analyse externe du problème. C'est une analyse descendante.

La programmation objet est moins intuitive que la programmation fonctionnelle. La programmation objet s'intéresse au système en tant qu'ensemble d'objets en interaction. C'est une analyse systémique.

Le diagramme de classes est le diagramme objet le plus important de la modélisation orientée objet. Il contient les classes avec leurs méthodes. Mais **il n'indique pas comment utiliser ces méthodes**. C'est une description statique du système.

Les diagrammes d'interactions (**diagrammes de collaboration et de séquence**), modélisent l'aspect dynamique.

Les **diagrammes d'objets** modélise des états du système à un instant donné. Leur usage est secondaire.

1. L'approche objet - introduction

La programmation objet est basée sur 5 concepts fondateurs :

1. **Objet**
2. **Message**
3. **Classe**
4. **Héritage**
5. **Polymorphisme**

Les chapitres suivants présentent ces 5 concepts.

L'Objet

Présentation

Objet = une identité + un état + un comportement

On dit aussi :

Objet = données (état) + méthodes (comportement, rôles, responsabilités)

D'un point de vue abstrait, un objet informatique est une représentation d'un objet (une réalité) du monde extérieur. Cette représentation est caractérisée par des valeurs et des rôles à jouer.

D'un point de vue informatique, un objet informatique est une variable avec un ou plusieurs champs qui seront manipulés (en lecture ou en écriture) par les fonctions associées à l'objet (les méthodes). Cette variable est aussi associée à des fonctions de plus haut niveau : les responsabilités ou rôles.

Etat

Valeurs instantanées des attributs (des données) d'un objet.

Certaines parties de l'état peuvent évoluer au cours du temps.

D'autres parties de l'état peuvent être constantes.

Comportement

Le comportement regroupe les **méthodes** (ou **compétences** ou **responsabilités** ou **rôles**) d'un objet.

Les méthodes sont des fonctions qui permettent d'accéder aux valeurs des attributs d'un objet mais aussi des fonctions de plus haut niveau (responsabilités et rôles).

Ces méthodes sont déclenchées par des stimulations externes : des messages envoyés par d'autres objets (c'est-à-dire des appels de méthodes).

Identité

Chaque objet possède une identité attribuée de manière implicite à la création de l'objet et jamais modifiée (c'est son adresse en mémoire).

On peut donner un attribut clé à l'objet, qu'on appelle « clé naturelle » (par exemple, un numéro de sécurité sociale). Il s'agit toutefois d'un artifice de réalisation. Cette clé appartient à l'état de l'objet. **Le concept d'identité est indépendant du concept d'état.**

Les objets sont différenciés par leurs noms (comme un nom de variable).

Toutefois, il est parfois difficile de nommer tous les objets : on peut donc les nommer du nom de leur classe, avec « : » devant.

Syntaxe UML

Les objets sont soulignés et placés dans un rectangle. Le nom de l'objet commence par une minuscule. Le nom de la classe commence par un majuscule.

Objets nommés :

Objets sans noms :

Encapsulation

Pour accéder, en consultation ou en modification, à l'état d'un objet (à ses données), il faut **passer par ses fonctions** (ses méthodes). Il faut que ces fonctions soient appelées par d'autres fonctions qui peuvent être des méthodes du même objet ou d'un autre.

Quand une méthode d'un objet 2 est appelée par une méthode d'un objet 1, **on dit que l'objet 1 a envoyé un message à l'objet 2. Le message, c'est la méthode de l'objet 2.**

Donc, de façon générale, pour accéder à l'état d'un objet, il faut lui envoyer un message.

Les méthodes sont l'interface obligatoire d'accès aux données d'un objet.

Persistance des objets

Un objet persistant sauvegarde son état dans un système de stockage permanent, de sorte qu'il est possible d'arrêter le processus qui l'a créé sans perdre l'information représentée par l'objet. C'est la passivation de l'objet.

L'activation de l'objet consiste à reconstruire l'objet dans l'état dans lequel on l'avait sauvegarder.

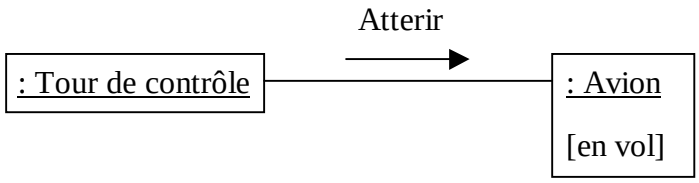
Par défaut, les objets ne sont pas persistants.

Transmission des objets

La transmission des objets consiste à faire passer un objet, par un moyen de communication quelconque, d'un système à un autre.

Scénario de communication

L'objet révèle son vrai rôle et sa vraie responsabilité lorsque, par l'intermédiaire de l'envoi de messages, il s'insère dans un scénario de communication (c'est-à-dire un cas d'utilisation concret du système).



L'objet « Avion » a dans ses méthodes la fonction « Atterrir ».

L'objet « Tour de contrôle » a dans ses données un objet avion ou un pointeur sur un objet avion. Une méthode de l'objet « Tour de contrôle » peut donc envoyer un message à l'objet « Avion », c'est-à-dire faire appel à la fonction « Atterrir ». Autrement dit, envoyer un message, c'est appeler une fonction : un message, c'est un ordre impératif !

Communication entre objets

Un système informatique peut être vu comme une société d'objets qui communiquent entre eux pour réaliser les fonctionnalités de l'application.

Le comportement global d'une application repose sur la communication entre les objets qui la composent.

De ce fait, l'étude des relations entre les objets du domaine est de première importance dans la modélisation objet.

De plus, la première communication étant la communication entre les utilisateurs (acteurs externes) et le système, on peut aussi partir de l'analyse fonctionnelle pour trouver les classes.

Collaboration et communication

On peut parler indifféremment de communication ou de collaboration entre objet.

Cependant, on parle plutôt de collaboration quand on décrit les communications nécessaires pour réaliser une fonctionnalité. La collaboration est plutôt finalisée.

On parle par contre indifféremment de communication pour une communication atomique ou une collaboration.

3 catégories d'objet en fonction de leur mode communication

Un objet peut être :
- émetteur de message sans jamais en recevoir,
- destinataire de message sans jamais en émettre,
- émetteur et destinataire de message.

- **Les acteurs : client, thread**

  Ce sont des objets à l'origine d'une interaction. Ce sont des objets actifs. Il possède un « fil d'exécution » : un thread. Ils passent la main aux autres objets. On peut les appeler « client ».

- **Les serveurs**

  Ce sont des objets qui ne sont jamais à l'origine d'une interaction. Ils sont toujours destinataire des messages. Ce sont des objets passifs.

- **Les agents**

  Ce sont des objets qui cumulent les caractéristiques des acteurs et des serveurs. Ils peuvent interagir avec les autres objets à tout moment, de leur propre initiative ou suite à une sollicitation externe.

  Les agents permettent le mécanisme de délégation. Un client peut communiquer avec un serveur qu'il ne connaît pas via un agent. Les agents découplent les acteurs des serveurs en introduisant une indirection dans le mécanisme de propagation des messages.

Le Message

Principes

- L'unité de communication entre les objets est le message.
- L'envoi de message s'apparente à un appel de fonction.
- Cependant, il n'y a pas de correspondance statique prédéfinie entre le nom de l'appel et le code effectivement exécuté. En effet, l'appel de fonction de la programmation procédurale correspond à un branchement sur la fonction à partir d'une table d'adresses. Le mécanisme d'envoi de message consiste à remonter à la classe pour trouver la fonction et à remonter la hiérarchie des classes jusqu'à trouver la fonction correspondante tant au niveau du nom qu'au niveau des paramètres (polymorphisme). Si aucune fonction n'est trouvée, alors il y aura un message d'erreur.
- Le message acquiert toute sa force d'intégration lorsqu'il est associé au polymorphisme et à la liaison dynamique.

Synchronisation des messages

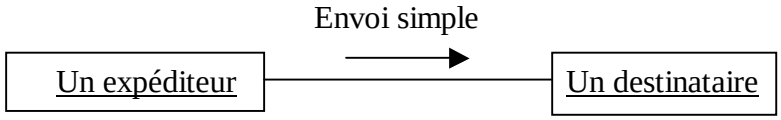
La synchronisation précise la nature de la communication et les règles qui régissent le passage des messages.

Message synchrone

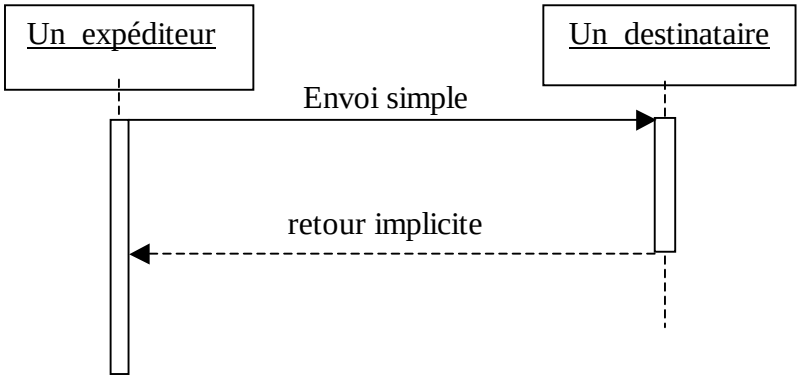
Une fois le message envoyé, l'expéditeur est bloqué jusqu'à ce que le destinataire accepte le message.

Un appel de procédure est un message synchrone.

- ***Diagramme de collaboration correspondant :***



- ***Diagramme de séquence correspondant :***

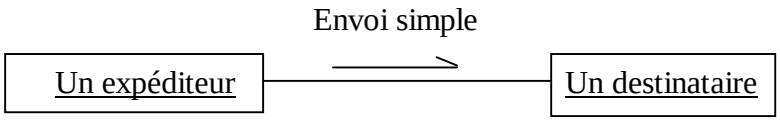


Message asynchrone

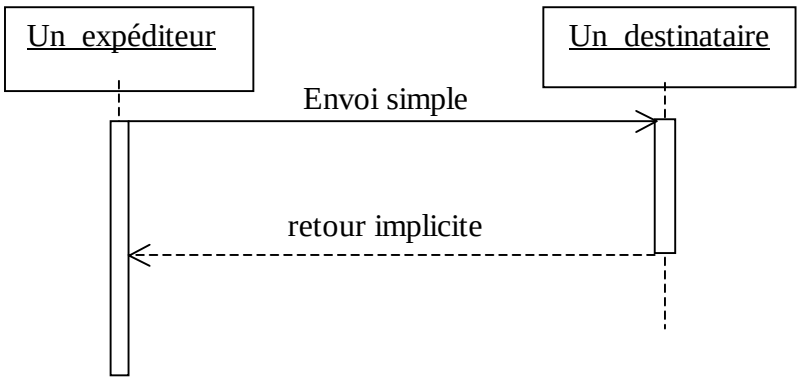
Il n'interrompt pas l'exécution de l'expéditeur.

L'expéditeur envoie le message sans savoir quand ni même si le message sera traité par le destinataire.

- ***Diagramme de collaboration correspondant :***



- ***Diagramme de séquence correspondant :***

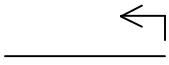


Demi flèche ou flèche simple peuvent représenter les envois asynchrones.

Message dérobant

Un message dérobant déclenche une opération seulement si le destinataire s'est préalablement mis en attente du message.

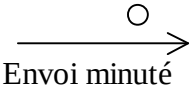
Dans le cas d'un message synchrone, l'expéditeur accepte d'attendre ; dans le cas d'un message dérobant, le destinataire accepte d'attendre.



Envoi dérobant

Message minuté

Un message minuté bloque l'expéditeur pendant un temps donné, en attendant la prise en compte de l'envoi par le destinataire, ou la durée spécifiée dans le message.



Envoi minuté

La Classe

Classe, objet, instance

Une classe est la description d'un ensemble d'objets ayant les mêmes méthodes et les mêmes types de données.

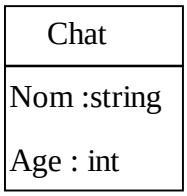
La classe peut être vue comme une extension de la notion de type.

L'objet est la réalisation concrète de la classe : un objet est une instance d'une classe.

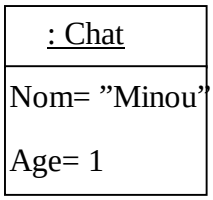
Les objets apparaissent alors comme des variables de type classe.

Les attributs d'une classe

Ils sont définis par un nom, un type et éventuellement une valeur initiale. En général leur valeur est variable pour chaque instance de la classe (chaque objet). Toutefois, il existe des attributs constants au niveau de la classe : leur valeur sera la même pour chaque instance de la classe.



Classe Chat



Un objet Chat

Les principales catégories de méthodes

- **Les constructeurs :** pour créer les objets (création).
- **Les destructeurs :** pour détruire les objet (destruction).
- **Les sélecteurs :** pour renvoyer tout ou partie de l'état d'un objet (consultation).
- **Les modificateurs :** pour modifier tout ou partie de l'état d'un objet (modification).
- **Les itérateurs :** pour consulter le contenu d'une structure de données qui contient plusieurs objets.
- **Les responsabilités ou rôles :** ces méthodes correspondent aux fonction de haut niveau permettant la réalisation des fonctionnalités du système. En phase de conception, on s'intéresse surtout à ces méthodes.

L'Héritage

Héritage et représentation ensembliste

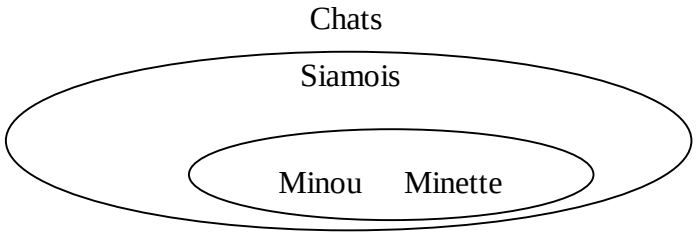
La représentation ensembliste permet de concevoir aisément la notion héritage : il suffit de concevoir l'inclusion d'un ensemble dans un autre.

L'ensemble inférieur hérite des propriétés et des associations de l'ensemble supérieur : l'espèce hérite des attributs du genre.

Un ensemble fait hériter ses attributs à tous les ensembles de niveaux inférieurs qu'il contient. Le genre fait hériter à toutes ses espèces.

Cette organisation forme une hiérarchie, c'est la hiérarchie de spécialisation/généralisation, encore appelée hiérarchie « is-a », « est-un ».

- ***Formalisme ensembliste***



Chaque élément d'un ensemble est aussi élément des ensembles qui le contiennent : « Minou », c'est mon chat siamois : il appartient à l'ensemble des Siamois, mais aussi à l'ensemble des Chats. Minette c'est le chat siamois de la voisine.

Vocabulaire ensembliste

- ***Classe : ensemble, entité, table***

On peut considérer les classes comme des ensembles. Les attributs de la classe sont les attributs de l'ensemble. Chaque objet est un élément de l'ensemble.

La notion rejoint celle d'entité et de table.

- ***Objet : élément, tuple***

L'objet correspond à un élément d'un ensemble.

La notion rejoint celle de tuple (ligne de la table).

- ***Sous-ensemble : espèce, classe-enfant, classe-mère***

L'espèce est un sous-ensemble de l'ensemble dont on parle : le siamois est une espèce de chat. Quand on parle d'un sous-ensemble, on dit aussi : « classe-enfant ». Quand on parle de l'ensemble incluant, on dit aussi : « classe-mère ».

- ***Sur-ensemble : genre, classe-mère***

  Le genre est un « sur-ensemble » de l'ensemble dont on parle : le chat est le genre du siamois. Quand on parle d'un « sur-ensemble », on dit aussi : « classe-parent » ou « classe-mère ».

Abstrait / Concret – Abstraction - Abstraire

- ***L'abstraction, c'est la classe !***

  Tous les noms d'ensemble sont abstraits : ce sont des abstractions. Une classe est donc toujours abstraite d'un certain point de vue : ce qui est concret, c'est l'objet instance de la classe.

  Les seules choses concrètes, ce sont les éléments de l'ensemble, c'est-à-dire les objets (Minou, mon chat siamois).

  Toutefois, UML définit particulièrement la classe abstraite, par opposition aux autres : une classe abstraite est une classe pour laquelle il n'y a pas d'instanciations d'objet. Elle sert uniquement à porter des spécifications en tant que classe-mère pour une classe « concrète ».

- ***Abstraire***

  Abstraire, c'est remonter du concret à l'abstrait, donc des objets à la classe qui les englobe. Par exemple de Minou et Minette au Siamois, ou de Minou et Minette au Chat.

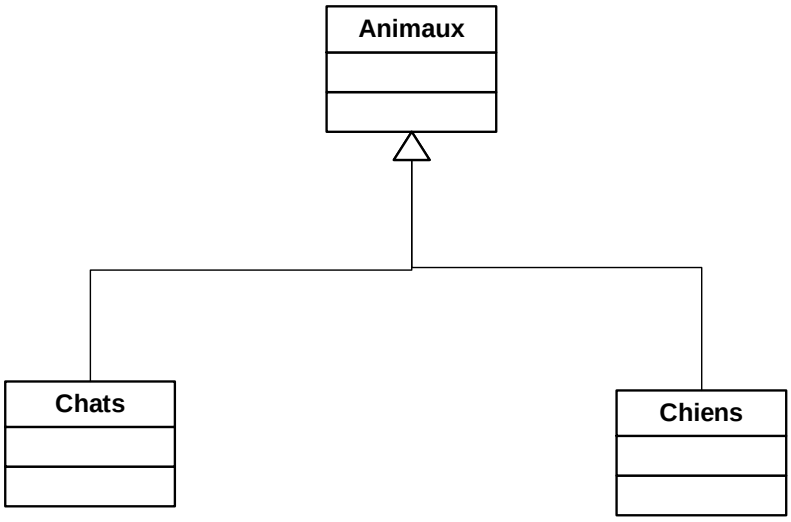
  Abstraire, c'est aussi remonter de l'espèce au genre, c'est-à-dire d'une classe-enfant à une classe-parent. Par exemple du Siamois et de l'Angora au Chat.

  Abstraire consiste à trouver des attributs communs à plusieurs ensembles ou à plusieurs choses concrètes pour définir un ensemble qui portera ces attributs et qui inclura les ensembles ou les choses concrètes en question.

L'héritage en programmation objet

L'héritage est une technique qui permet de construire une classe (espèce) en lui faisant hériter des attributs et des méthodes d'autres classes (genre, classe-parent, classe-mère, super-classe).

L'héritage est utilisé pour construire de nouvelles classes, pour classer les objets et pour la conception abstraite (avec le polymorphisme).



Principe de substitution

On peut substituer n'importe quel objet d'une super-classe par n'importe quel objet d'une sous-classe. La spécialisation ajoute des attributs et des opérations mais ni n'en détruit, ni n'en modifie.

Attention à l'héritage des associations !

Les associations sont héritées mais pas complètement ! ! !

Le plus souvent, les contraintes des associations ne sont pas transmises automatiquement de la super-classe vers la sous-classe. Les contraintes sont le plus souvent traduites par un bout de code implanté dans la réalisation d'une opération. Comme les langages permettent la redéfinition des opérations dans les sous-classes, les programmeurs peuvent involontairement introduire des incohérences entre la spécification d'une super-classe et sa réalisation dans une des sous-classes.

L'abstraction dans la programmation objet : un nouveau paradigme de programmation

L'abstraction consiste à regrouper les éléments qui se ressemblent et à distinguer des structures de plus haut niveau d'abstraction, débarrassées de détails inutiles.

La classe décrit le domaine de définition d'un ensemble d'objets.

Les généralités sont contenues dans la classe.

Les particularités sont contenues dans les objets.

Avec les langages-objet, le programmeur peut construire une représentation informatique des abstractions de haut niveau correspondant aux usages mêmes de l'application, sans traduction vers des concepts de plus bas niveau, comme les variables, les types abstraits de données et les fonctions des langages non objet.

Le Polymorphisme

- Le polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes (l'eau peut être à l'état solide, liquide ou gazeux).
- Le polymorphisme désigne le principe qui fait qu'un nom d'objet peut désigner des objets différents (des instances de classes différentes) issus d'une même arborescence.
- Le polymorphisme désigne surtout le polymorphisme d'opération : la possibilité de déclencher des opérations différentes en réponse à un même message.
- Le polymorphisme permet de manipuler des objets sans en connaître précisément le type.
- Le polymorphisme est lié à la notion d'héritage. On peut manipuler des objets dont le type est abstrait au niveau de la classe générale (abstraite). Le type deviendra concret quand l'objet deviendra concret et portera le type de sa classe spécialisée.
- Le polymorphisme n'influence pas l'analyse organique, mais il en dépend. Il ne faut pas penser l'analyse en terme de polymorphisme, mais en terme d'abstraction. L'analyse en terme d'abstraction rend possible le polymorphisme.
- Les bénéfices du polymorphisme sont un plus grand découplage entre les objets : ils sont avant tout récoltés durant la maintenance.

2. Le diagramme de classes

Notions générales sur les classes et formalisme UML

Définition

Une classe est une description abstraite d'un ensemble d'objets qui partagent les mêmes propriétés (attributs et associations) et les mêmes comportements (mêmes opérations, c'est-à-dire les mêmes en-têtes de méthodes).

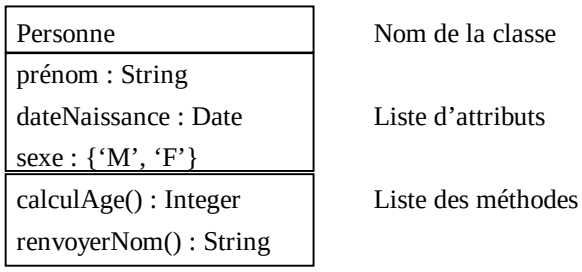
Instance

Un objet est une instance d'une classe.

L'instance est une notion générale :

- un lien est une instance d'une association ;
- une variable peut être considérée une instance d'un type.

Représentation UML



Le nom de la classe doit être significatif. Il commence par une majuscule.

Le nom de la classe peut être préfixé par son ou ses paquetages d'appartenance.

Si personne est dans le paquetage A, lui même dans le paquetage B, on écrira

B ::A ::Personne

Encapsulation

L'occultation des détails de réalisation est appelée : encapsulation.

L'encapsulation présente un double avantage :

- Les données encapsulées dans les objets sont protégées.
- Les utilisateurs d'une abstraction ne dépendent pas de sa réalisation, mais seulement de sa spécification, ce qui réduit le couplage dans les modèles.

Le degré d'encapsulation peut être paramétré : c'est la notion de visibilité.

Visibilité des attributs et des méthodes

Par défaut, les attributs d'une classe sont « private » : les attributs d'un objet sont encapsulées dans l'objet. Les attributs ne sont manipulables que par les méthodes de l'objet.

Par défaut, les méthodes d'une classe sont « public » : les méthodes d'un objet sont accessibles par tous les « clients » de l'objet.

UML définit quatre niveaux de visibilité

La visibilité des attributs et des méthodes est précisée par des mot-clés ou des symboles.

Symbole	Mot-clé	Signification
+	Public	Visible **partout**
		Visible **dans tout le paquetage** où la classe est définie
#	Protected	Visible dans la classe, **dans ses sous-classes** et par les amis.
-	Private	Visible uniquement **dans la classe** et par les amis.

Attribut de Classe

Certains attributs ont une valeur identique pour tous les objets de la classe. Ce sont des sortes de constantes définies au niveau de la classe. Ce sont les attributs de classe.

En UML, ces attributs sont listés avec les autres, mais ils sont soulignés.

Symbole	Mot-clé	Signification
Souligné		Valeur identique pour tous les objets de la classe

Attribut dérivé

Certains attributs peuvent être calculés à partir d'autres attributs de la classe.

En UML, ces attributs sont listés avec les autres, mais ils sont précédés d'un « / ».

Symbole	Mot-clé	Signification
/		Attribut dont la valeur est calculée à partir de celle d'autres attributs

Méthode de Classe

Quand une méthode ne porte pas sur les attributs d'objet de la classe mais uniquement sur des attributs de classe ou sur des valeurs constantes, cette méthode est dite méthode de classe.

Une méthode de classe peut être utilisée sans avoir à instancier d'objet. On peut se contenter d'un objet déclaré pour y accéder.

En UML, ces méthodes sont listées avec les autres, mais elles sont soulignées.

Compartiment des responsabilités

Le compartiment des « responsabilités » liste l'ensemble des tâches que la classe doit réaliser. C'est une façon de lister les méthodes dont on n'a pas encore défini l'entête. Quand la conception sera achevée, ce compartiment devra disparaître.

Compartiment des exceptions

Le compartiment des « exceptions » liste les situations exceptionnelles devant être gérées par la classe. Là encore, quand la conception sera achevée, ce compartiment devra disparaître.

Associations entre classes

Comme dans le modèle entité-association (MEA, MCD MERISE), UML permet d'établir des relations (des associations) entre les classes.

Il y a trois types d'associations entre classes :

- Les associations simples
- Les associations d'agrégation de composition
- Les associations d'héritage

Les deux premières expriment des relations entre les objets de deux classes (ou plus).

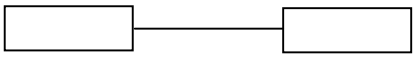
L'association d'héritage permettent de définir des sous-ensembles dans un ensemble ou inversement de définir un ensemble à partir de sous-ensembles. Elle exprime une relation entre les classes et pas entre les objets.

Les associations simples

Association

Une association représente une relation structurelle entre des classes d'objets.

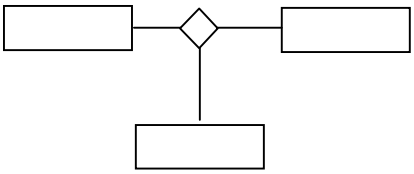
Une ligne entre deux classes représente une association.



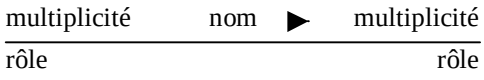
Association binaire (arité = 2)

La plupart des associations sont binaires : elles ne réunissent que deux classes.

Association ternaire et plus (arité >2)



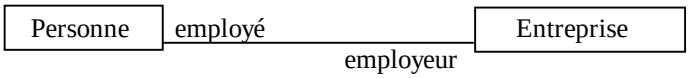
L'association et ses différents ornements



Nom et sens de lecture des associations

L'association peut avoir un nom. Le nom explicite le lien entre les deux classes. C'est souvent un verbe. Le sens de lecture ► permet de préciser dans quel sens il faut lire le nom.

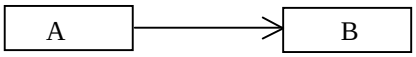
Rôles des extrémités des associations



Le rôle est un pseudo-attribut. On peut préciser sa visibilité (+, -, #)

La plupart du temps, le nommage des rôles est limité en fonction de la navigabilité.

Navigabilité des associations



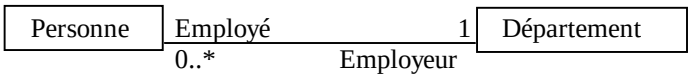
L'association n'est navigable que dans le sens de A vers B.

Une association navigable uniquement dans un sens peut être vue comme une demi-association.

Multiplicité (cardinalité) des associations

Chaque extrémité de l'association peut porter une indication de multiplicité qui montre combien d'objets de la classe considérée peuvent être liés à un objet de l'autre classe.

1	Un et un seul
0..1	Zéro ou un
N	N (entier naturel qui peut être précisé)
M .. N	De M à N (entiers naturels qui peuvent être précisés)
*	De zéro à plusieurs
0..*	De zéro à plusieurs
1..*	De un à plusieurs

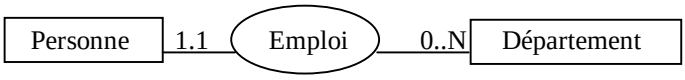


Chaque personne est employée dans un département et un seul.

Les départements sont employeur de 0 ou plusieurs personnes.

Attention :

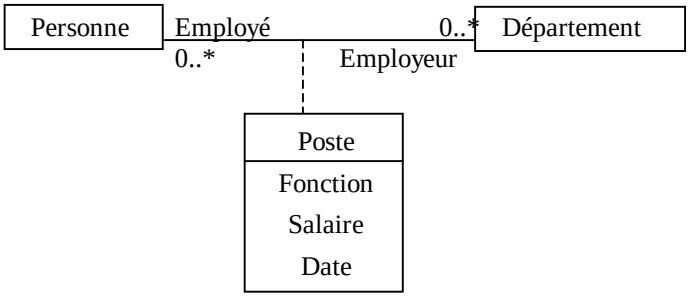
Les cardinalités sont mises à l'inverse du modèle entité-association (MEA) :



Classe-Association

Une association peut avoir ses propriétés qui ne sont disponibles dans aucune des classes qu'elle relie.

Pour cela, on relie une classe à une association :



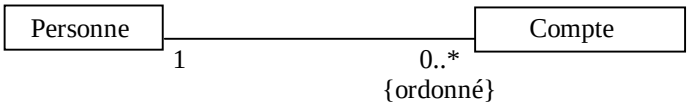
La classe-association correspond à l'association non-hiérarchique avec attributs du MEA .

Contraintes sur les associations

Toutes sortes de contraintes peuvent être définies sur les associations.

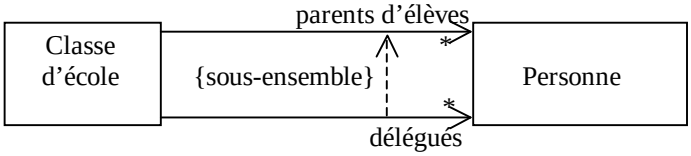
Certaines contraintes s'appliquent à une seule association :

- {ordonné} : précise qu'une collection (0..*) doit être ordonnée.



Certaines contraintes s'appliquent à plusieurs associations :

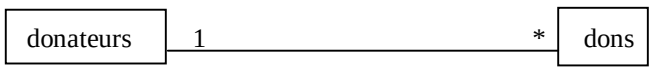
- {sous ensemble} : précise qu'une collection est incluse dans une autre collection
- {ou exclusif} : précise pour un objet donné qu'une association et une seule est possible parmi les associations contraintes par le ou exclusif.



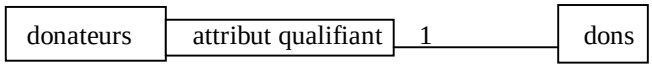
Qualification des associations (restriction)

La qualification des associations consiste à réduire le nombre d'occurrences d'une association.

Elle ne s'applique qu'à des associations dont la multiplicité est supérieure à 1 (sinon, le nombre d'occurrences ne peut pas être réduit).



Dans ce cas, la qualification ne peut s'appliquer qu'aux donateurs (un donateur peut faire plusieurs dons).

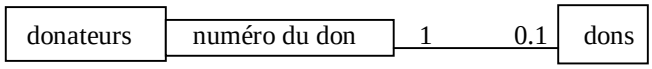


L'attribut qualifiant peut être extrait de la classe des dons.

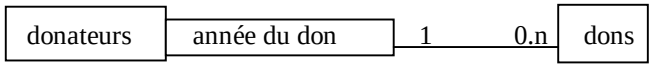
La qualification restreint la relation aux seuls couples concernés par un qualifieur.

La qualification des associations est une forme d'association contrainte.

- ***Exemples***



La classe « donateurs » contient un objet « dons », instancié ou pas, et filtré par numéro de don.

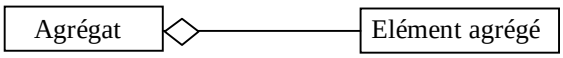


La classe « donateurs » contient une collection d'objets « dons » filtrés par année de don.

Agrégation et composition

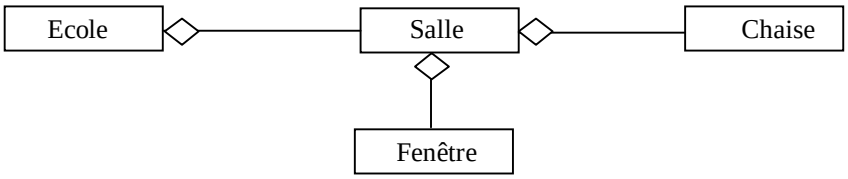
Agrégation

L'agrégation est une association non symétrique dans laquelle une des extrémités, l'agrégat, joue un rôle prédominant par rapport à l'autre extrémité, l'élément agrégé.



Quelle que soit l'arité de l'association, il ne peut y avoir qu'un seul agrégat.

L'agrégation représente en général une relation d'inclusion structurelle ou comportementale.



Une école est composée de plusieurs salles, qui elles-mêmes sont composées de plusieurs fenêtres et plusieurs chaises.

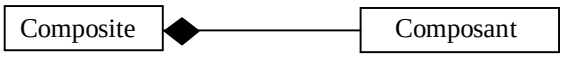
En général, les agrégations sont navigables uniquement dans le sens Agrégat vers Eléments agrégés.

En général, les agrégations sont des associations 1..*.

Quand on ne précise pas la cardinalité d'une agréation, c'est qu'elle est de cardinalité : 1..*.

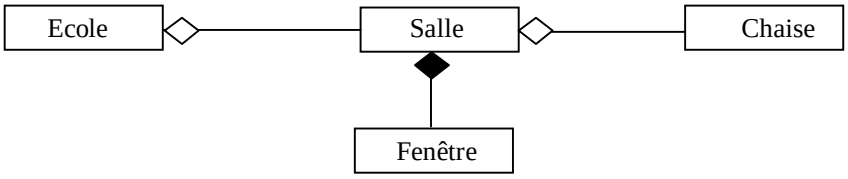
Composition

La composition est un cas particulier d'agrégation.



La composition implique une coïncidence des durées de vie des composants et du composite : la destruction du composite implique la destruction de tous ses composants.

La composition implique aussi une contrainte sur les valeurs de multiplicité du côté de l'agrégat : elle ne peut être que 0 ou 1 : un composant ne peut pas être partageable.



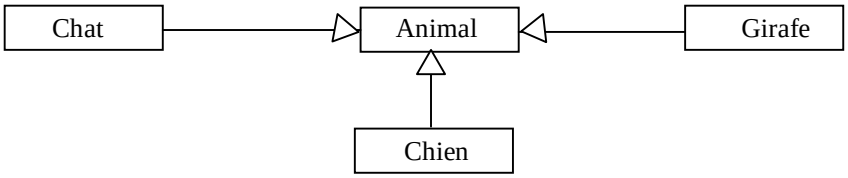
Si on supprime la salle, les fenêtres sont aussi supprimées. Ce qui n'est pas le cas des chaises.

Généralisation

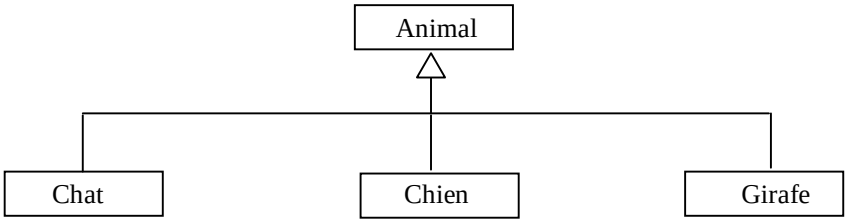
Le terme généralisation désigne une relation de classification entre un élément plus général (le genre) et un élément plus spécifique (l'espèce). La classe Chat est une espèce de la classe Animal.

On parle de classe spécifique, ou sous-classe, ou classe enfant.

Et de classe générale, ou sur-classe, ou super classe, ou classe parent, ou classe mère.



Ou encore :



Distinction entre généralisation et composition

Un chat est un animal : c'est une généralisation. La classe chat est une partie de la classe animal. Chaque objet « chat-x » appartient à la classe « chat » et à la classe « animal ».

Un chat a deux oreilles : c'est une composition. L'objet « oreille » est une partie de l'objet « chat ». Pour tout objet « chat », il existe deux objets « oreille ».

Héritage

La classe spécifique contient des attributs et des méthodes qui lui sont propres.

La classe spécifique hérite de tous les attributs, méthodes et associations de la classe générale, sauf pour ceux qui sont privées.

Une classe spécifique peut redéfinir une ou plusieurs méthodes. C'est le principe de la surcharge des opérations. Un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.

Un objet spécifique peut être utilisé partout où un objet général est attendu : partout où on attend un animal, on peut utiliser un chat.

Généralisation multiple

Une sous-classe peut avoir plusieurs sur-classes

Les relation de dépendance

Présentation

Les relations de dépendances sont utilisées quand il existe une relation sémantique entre plusieurs éléments qui n'est pas de nature structurelle (association, composition, agrégation ou héritage.

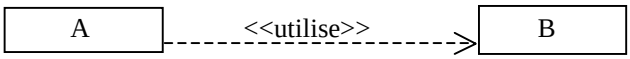
Type de dépendance

Type de dépendance	Stéréotype	Signification
Permission	<<Ami>>	La source a accès à la destination, quelle que soit la visibilité.
Abstration	**<<Réalise>>** <<Raffine>> <<Trace>>	Un même concept à des niveaux d'abstraction différents. Réalisation d'une spécification (opération). Hiérarchie sémantique (l'analyse raffine la conception) Historique des constructions de différents modèles.
Utilisation	**<<Utilise>>** <<Appelle>> <<Crée>> <<Instancie>>	La source requiert la cible pour son bon fonctionnement. Une opération de la source invoque une opération de la cible. La source crée une instance de la cible. Idem que <<Crée>>.
Liaison	<<Lie>> <<Dérive>>	Liaison entre une classe paramétrée et une classe paramétrable Elément calculé à partir d'autres.

Principales dépendances

<< Utilise >> et << Réalise >> sont les principales dépendances.

Représentation UML



Remarque sur la dépendance d'utilisation

Dans la déclaration d'une classe, on peut faire référence à une autre classe de 3 manières :

- En déclarant un attribut
- En déclarant un paramètre formel
- En déclarant une variable locale

Dans chacun de ces cas, la classe d'origine utilise la classe à laquelle elle fait référence pour une déclaration.

Il y a toujours dépendance quand il y a navigabilité.

Le principe général de la modélisation est de minimiser au maximum des dépendances, quelles qu'elles soient.

Les classes abstraites

Définition

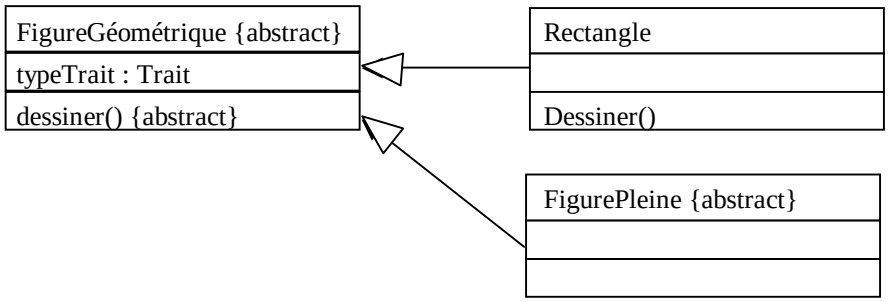
Une classe abstraite est une classe qui ne s'instancie pas directement, par opposition à une classe concrète : c'est une classe qui ne contient pas d'objets.

Les classes abstraites sont en général des super-classes qui contiennent des sous-classes qui, elles, contiendront des objets.

Ces classes abstraites servent surtout pour la classification et le polymorphisme.

En phase de modélisation, une classe est aussi dite abstraite quand elle, ou une de ses classes parents, définit au moins une méthode abstraite (et donc qu'elle ne peut pas, au moins provisoirement, être instanciée).

Formalisme



La classe « figure géométrique » est abstraite parce quelle contient une méthode abstraite, c'est-à-dire une méthode dont on connaît l'entête mais pas le corps.

La classe « Figure pleine » est abstraite parce qu'elle hérite de la méthode abstraite « dessiner » qu'elle ne redéfinit pas comme la classe « Rectangle ».

Méthode et opération

La spécification d'une méthode correspond à son en-tête : on l'appelle aussi : **l'opération**.

L'implémentation de la méthode (l'algorithme), c'est ce qu'on appelle la méthode.

Méthode abstraite

Une méthode est dite abstraite quand on connaît son en-tête mais pas la manière dont elle peut être réalisée.

Les classes « interface » ou « boundary »

Présentation

Une classe « interface » décrit le comportement visible d'une classe autrement dit la communication entre la classe et son environnement, que ce soit un utilisateur ou une autre classe.

Les classes interfaces modélisent les interfaces du système.

L'interface est la partie visible d'une classe ou d'un package. Elle est parfois synonyme de spécifications, ou vue externe, ou vue publique.

Caractéristiques techniques

Une interface est une classe avec le stéréotype « interface ».

- Elle contient un ensemble d'opérations qui caractérisent le comportement d'un ou plusieurs éléments, ces opérations ayant une visibilité « public ».
- Elle ne contient pas d'attributs.
- Elle est abstraite : aucun objet (instance) ne la réalise. Elle ne contient que des opérations et pas de méthodes.
- Les opérations d'une interface sont réalisées par la ou les classes qui la réalisent.
- Une classe qui réalise une interface réalise toute les opérations de l'interface.
- Une classe peut réaliser plusieurs interfaces.
- Une interface peut être réalisée par plusieurs classes.

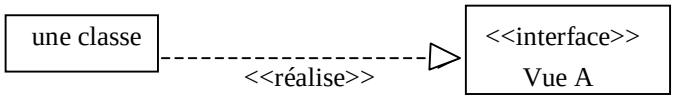
Représentation UML des interfaces

Le symbole ——○ montre l'existence d'une interface, mais ne précise pas les opérations.

La flèche --------▷ signifie que la source réalise la destination. La flèche triangulaire est de type « héritage ».

<<interface>> est le nom d'un stéréotype

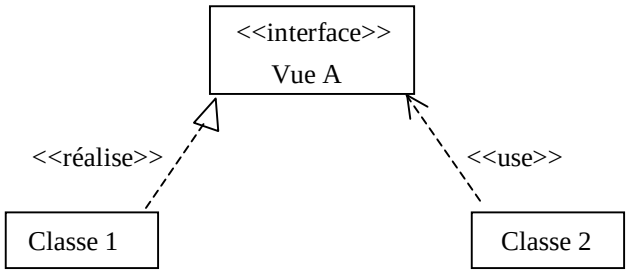
On peut préciser le stéréotype du lien : « réalise » (ce n'est pas obligé).



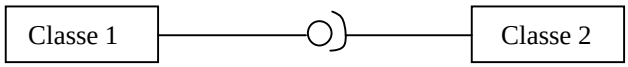
Utilisation des interfaces

Une classe peut utiliser tout ou partie des opérations proposées par une interface. Cette classe est alors dépendante de l'interface.

La relation d'utilisation <<use>> signifie que la source requiert la cible pour son bon fonctionnement.



ou encore :



Méthode de d'analyse des interfaces

Pour trouver les interfaces du système, il faut **examiner toutes les paires : acteurs physiques – scénario.** On a intérêt à créer dans un premier temps au moins une classe interface par cas d'utilisation.

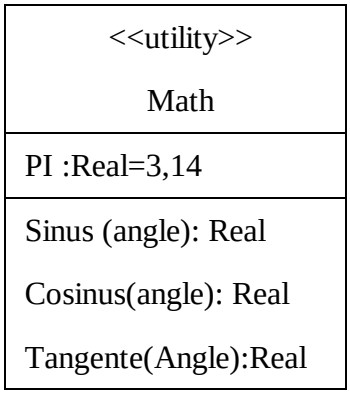
Les interfaces se trouvent à un haut niveau d'abstraction. On commence par renseigner les besoins en interfaces utilisateur sans les implémenter. Ces classes seront affinées au fur et à mesure.

On a intérêt à regrouper les interfaces dans un paquetage à part.

En utilisant des interfaces plutôt que des classes, on sépare les traitements de leur interface. Ainsi, on facilité l'évolution des applications et leur adaptations à différents environnement d'interface.

Quelques classes particulières

Les classes de variables et d'opérations globales : stéréotype <<utility>>



Les classes actives

Par défaut, les classes sont passives.

Une classe active possède son propre flot d'exécution. Les classes actives peuvent être des <<processus>> ou des <<thread >>.

Un **processus** est un flot de contrôle lourd qui s'exécute dans un espace d'adressage indépendant.

Un **thread** est un flot de contrôle léger qui s'exécute à l'intérieur d'un processus.

Une classe active choisit un ou plusieurs flots de contrôle.

- ***Représentation UML***

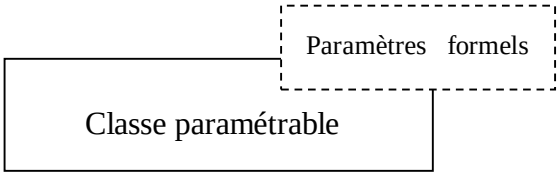
Classe et objet actif sont représentés par un **cadre plus épais** ou avec un trait double.

Les classes paramétrables : template

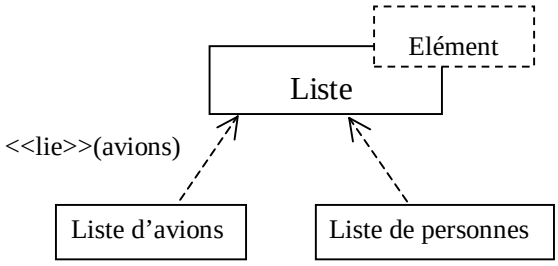
Une classe paramétrable est un modèle de classe.

Une classe paramétrable a des paramètres formels. Chaque paramètre possède un nom, un type et une valeur par défaut optionnelle.

Une classe paramétrable est liée à des paramètres effectifs pour devenir une classe paramétrée qui pourra être instanciée.

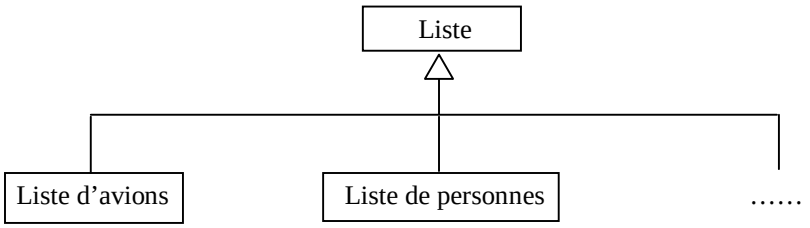


➢ ***Exemple***



Ce type de classe apparaît rarement au début de la modélisation.

A noter que les classes paramétrables peuvent être remplacées, de façon moins élégante, par des généralisations :

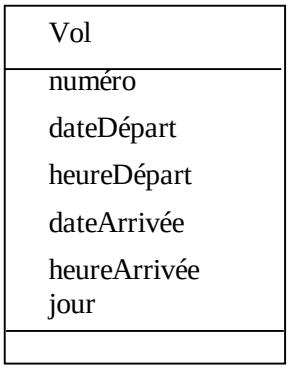


Amélioration de l'analyse des classes : notion de métaclasse

Une métaclasse est une classe (2) d'une classe (1). La métaclasse est telle qu'un objet de cette classe contient des données et des opérations qui caractérisent plusieurs objets de la classe dont elle est issue : classe (1).

Exemple :

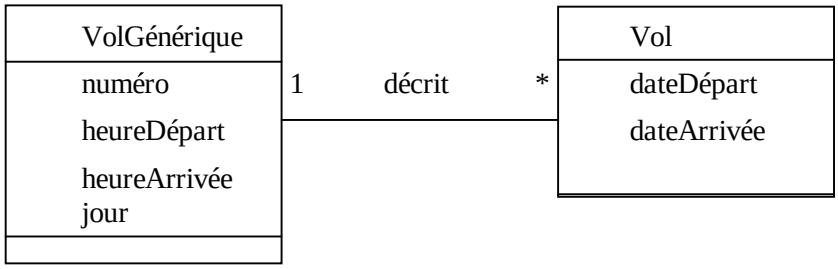
Soit la classe des vols d'une compagnie aérienne. Cette classe contient les attributs suivants :



Le numéro du vol caractérise tous les vols qui ont lieu le même jour de la semaine à la même heure.

En réalité, cette classe modélise deux classes : la métaclasse des « genres de vols » - VolGénérique - et la classe des vols concrets.

La classe des vols concrets ne contient que les attributs : dateDépart et dateArrivée.

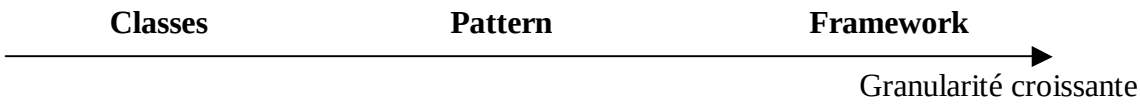


Notion de pattern

Présentation

Les patterns sont des micro-architectures finies.

Comme les frameworks (architectures semi-finies) ils permettent d'élever la granularité de la modélisation, et d'éviter de réinventer la roue à chaque projet !



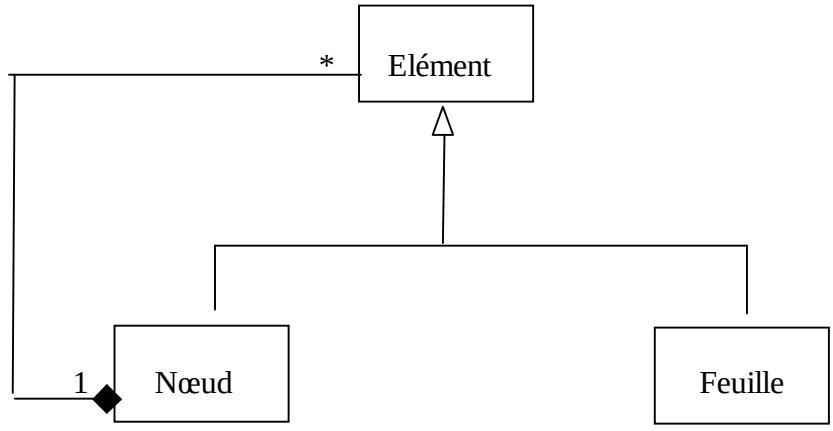
Caractéristiques des patterns

- Les patterns synthétisent l'élaboration itérative de solutions éprouvées, qui ont évolué au cours du temps pour fournir des structures toujours plus flexibles et plus facilement réutilisables.
- Les patterns forment un vocabulaire de très haut niveau et très expressif. Les informaticiens habitués aux patterns identifient, nomment et parlent des problèmes et des solutions en utilisant ce vocabulaire.
- Les patterns sont indépendants de tout langage de programmation, ce qui leur confère une grande généricité. En contrepartie, il n'y a pas de réutilisation de code et un effort de programmation est requis pour les mettre en œuvre.

Les différents types de patterns

- **Les patterns de conception (design patterns)** : ce sont les plus populaires car ce sont les patterns des informaticiens.
- **Les patterns d'analyse ou patterns métier** : les deux notions sont équivalentes. Ils dépendent d'un secteur d'activité particulier.
- **Les patterns d'architecture :** ils décrivent la structure des architectures logicielles, comme la structuration en couches.
- **Les patterns organisationnels** : ils fournissent des solutions à des problèmes organisationnels, telle l'organisation des activités de développement logiciel.
- **Les patterns d'implémentation :** ils expliquent comment exploiter un langage de programmation pour résoudre certains problèmes typiques.
- **Les patterns pédagogiques :** ils décrivent des solutions récurrentes dans le domaine de l'apprentissage.

Exemple de design pattern : Le pattern « composite »



Ce pattern est décrit dans « Design Patterns : Elements of Reusable Object-Oriented Software, E. Gamma et al., 1995, Addison-Wesley.

Il fournit une solution pour modéliser les arbres.

Construction d'un diagramme de classes : 3 grands types de classe

3 points de vue guident la modélisation du diagramme des classes :

- Le point de vue statique
- Le point de vue fonctionnel (au sens de l'analyse fonctionnelle)
- Le point de vue dynamique

Le point de vue statique : les classes « entité » ou « métier »

- Il met l'accent sur les concepts du domaine et les associations qui les relient.
- Les classes correspondantes sont parfois stéréotypées par le nom « entité » ou « entity »: c'est le point de vue qui se rapproche le plus du MCD (ou de la modélisation des données persistantes).

Méthode de construction du diagramme des classes « entité » :

- Analyse du type de celle faite pour un MCD. Il s'agit de trouver les classes (entités), les attributs et les associations à partir de la description « métier » du domaine.
- En même temps ou à la suite, ajouter les méthodes (les responsabilités au sens large) en s'appuyant sur l'analyse fonctionnelle des cas d'utilisation.
- En même temps ou à la suite, organiser et simplifier le diagramme en utilisant l'héritage.
- En même temps ou à la suite, ajouter les classes « interface » et « control »
- Itérer et affiner le modèle

Le point de vue fonctionnel : les classes « interface »

- Il met l'accent sur les interfaces : interfaces utilisateur ou interfaces avec d'autres systèmes.
- Les classes correspondantes sont stéréotypées par le nom « interface » ou « boundary ».
- Les classes « interface » constituent la partie du système qui dépend de l'environnement.

Méthode pour trouver les classes « interface » :

- Examiner les paires acteur - scénario (occurrence de cas d'utilisation).
- Les classes correspondent en gros à une étape de l'interaction avec l'utilisateur.
- Ne pas hésiter à travailler à un haut niveau d'abstraction : ces classes seront affinées pendant la conception.

Le point de vue dynamique : les classes « control »

- Il met l'accent sur l'implémentation.
- Les classes correspondantes sont stéréotypées par le nom « control ».
- Les classes « control » modélisent le séquençage comportemental (une séquence est une suite ordonnée d'opérations ; le séquençage est la détermination de la séquence).

Méthode pour trouver les classes « control » :

- Ajouter une classe « control » pour chaque paire acteur / cas d'utilisation.
- Les classes « control » définies par le couple acteur/cas d'utilisation ne sont qu'un point de départ : au cours de l'avancement de l'analyse, elle pourront être éliminées, éclatées ou fusionnées.

Attention !!!

Les classes « interface » et « control » ne doivent pas conduire à dissocier le comportement et les données :

- Elles ne doivent pas conduire à revenir à la programmation procédurale !
- Elles ne doivent pas appauvrir le diagramme des classes « entité ».

Le point de vue essentiel reste le point de vue des classes « entité ». Les deux autres points de vue servent :

- A rendre plus concrète la modélisation
- A partir d'un niveau d'abstraction élevé
- A affiner le diagramme des classes « entité ».

Documentation des méthodes

Dans le diagramme de classe, il faudra préciser pour chaque méthode (ou au moins pour chaque opérations des interfaces) ce qu'on précise classiquement pour chaque fonction :

- Nom de la fonction
- Liste des paramètres : nom de la variable, type, mode de passage (entrée, sortie, entrée-sortie), signification (usage).
- Liste des attributs de classe utilisées par la méthode : nom de l'attribut, mode de passage
- But de la méthode

Une zone de commentaires permet d'ajouter ces informations dans les ateliers de génie logiciel (avec power AMC par exemple). Les commentaires se retrouvent ensuite dans le code généré automatiquement.

3. Les diagrammes de séquence

Présentation

Les diagrammes d'interactions permettent de montrer les échanges de messages entre objet (les interactions) pour une méthode.

On les utilise en général pour les méthodes d'interface.

Diagramme de séquence et diagramme de collaboration sont deux instances possibles pour un diagramme d'interaction ;

Diagramme de séquence	**Diagramme de collaboration**
Représentation orientée client	Représentation orientée développeur
Représentation graphique d'un scénario	
Montre des **interactions entre des objets** dans un enchaînement temporel	
Montre les **messages échangés** entre les objets pour réaliser la fonctionnalité du scénario	
Peuvent être transformés l'un par l'autre	

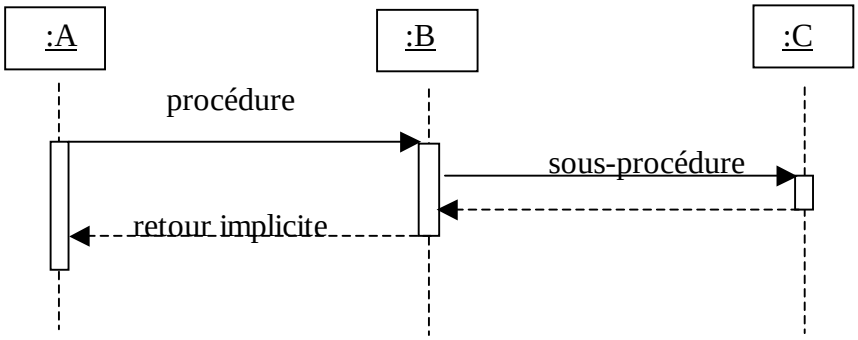
Les types de messages

→ **Message synchrone :** une fois le message envoyé, l'expéditeur est bloqué jusqu'à ce que le destinataire accepte le message.

→ **Message asynchrone :** le message envoyé n'interrompt pas l'exécution de l'expéditeur.

→ Autre représentation d'un message asynchrone

Représentation d'un appel de procédure emboîtée

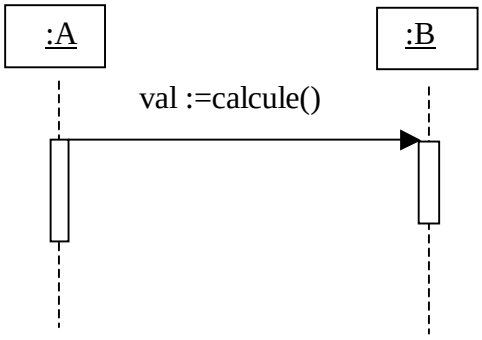


:A est un objet

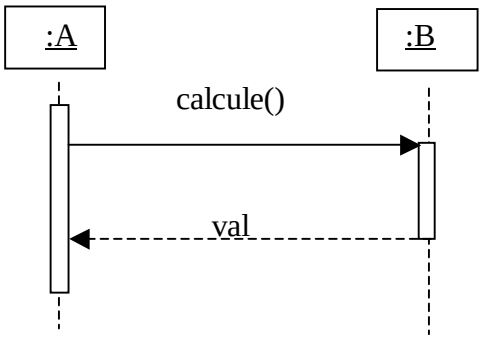
La ligne verticale en pointillé est appelée : ligne de vie.

Le rectangle sur la ligne de vie représente la période d'activité de l'objet.

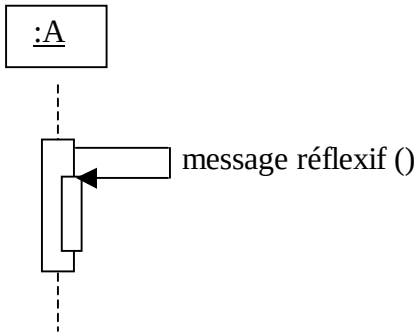
Représentation du paramètre de retour



Ou encore :

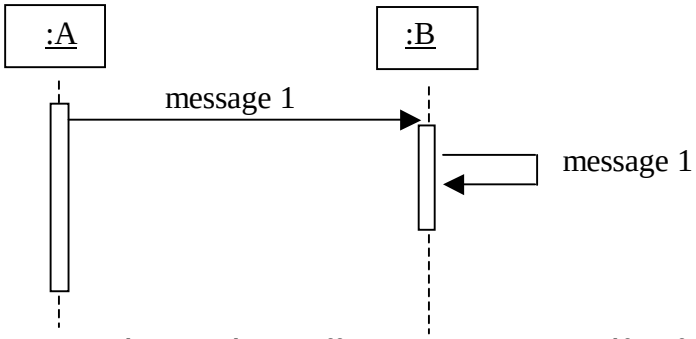


Envoi d'un message réflexif



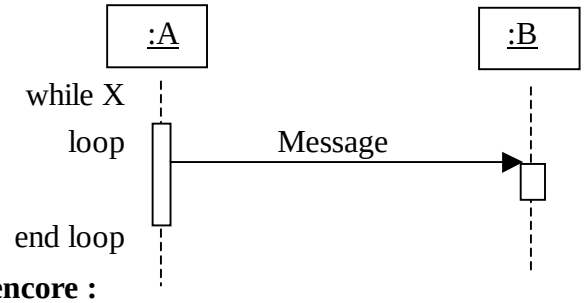
Un message réflexif est un message envoyé d'un objet vers lui-même. Autrement dit, une méthode d'un objet fait appel à une autre méthode du même objet.

Représentation de la récursivité

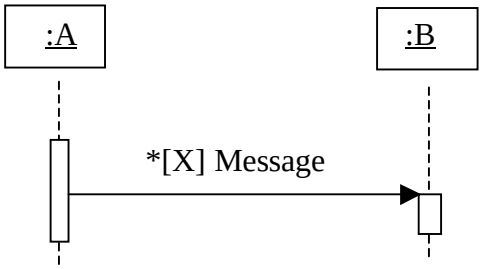


Pour représenter la récursivité, il suffit que le message réflexif ait le même nom que celui d'origine (message 1 dans l'exemple).

Représentation des boucles



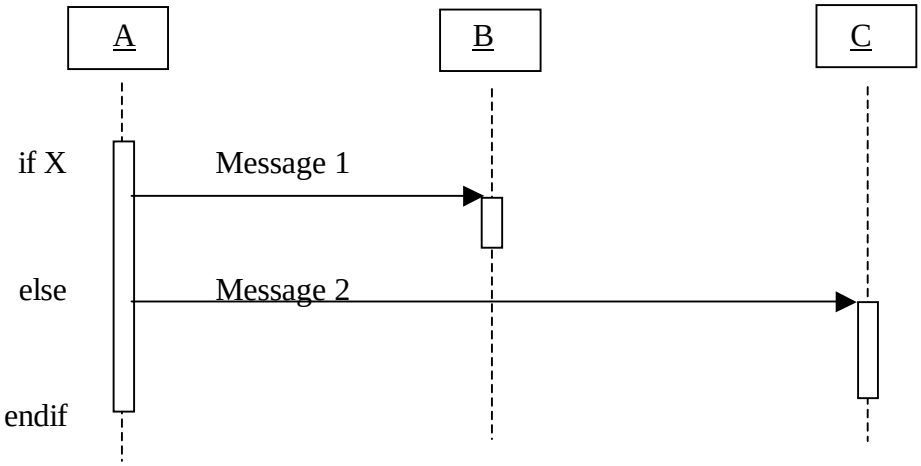
Ou encore :



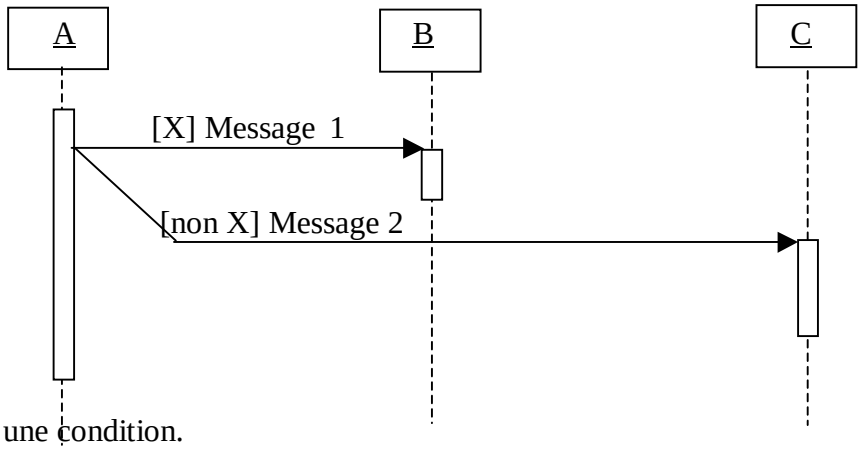
La boucle est symbolisée par le "*" place devant la condition entre crochets.

Le « X » est une condition de boucle « tant que ». Pour une boucle « pour tous », on écrira *[ ] ou *[tous].

Représentation des tests

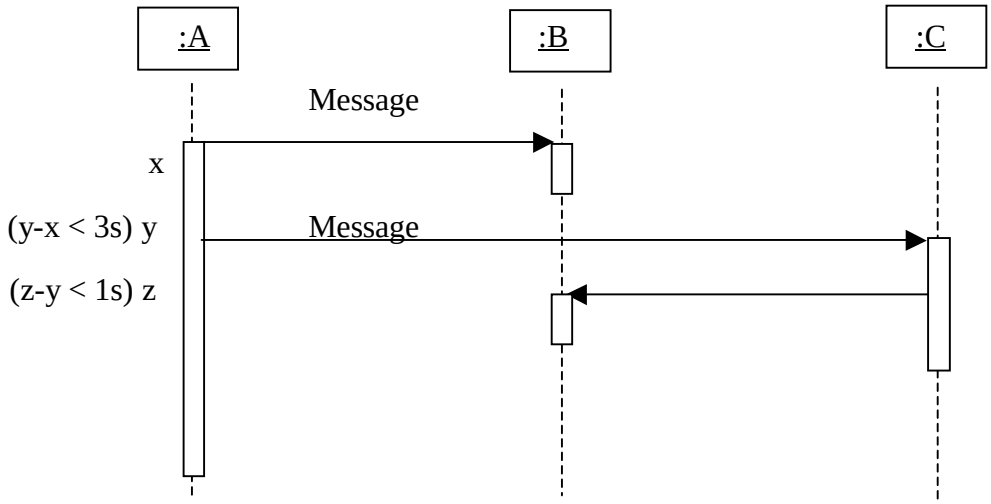


Ou encore :

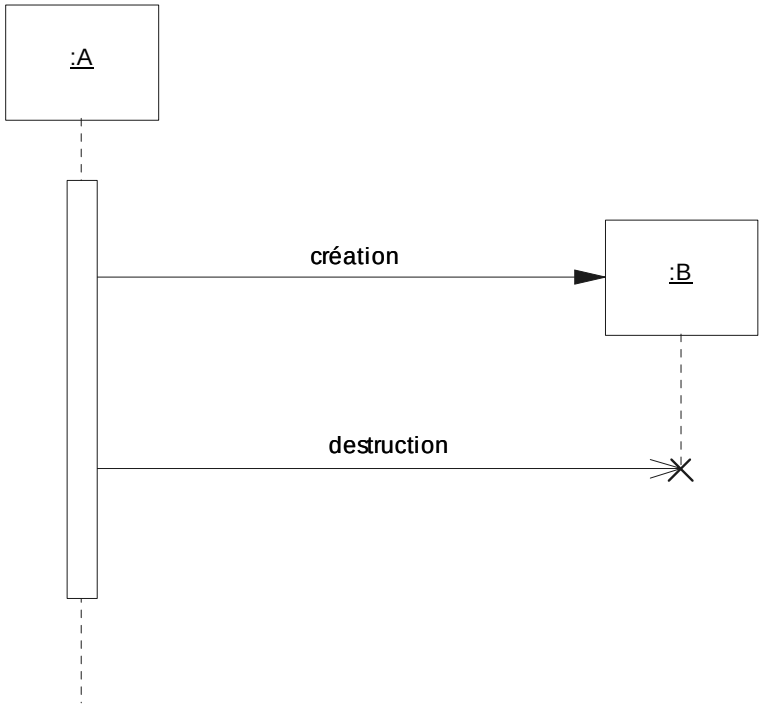


Le « X » est une condition.

Représentation de contraintes temporelles

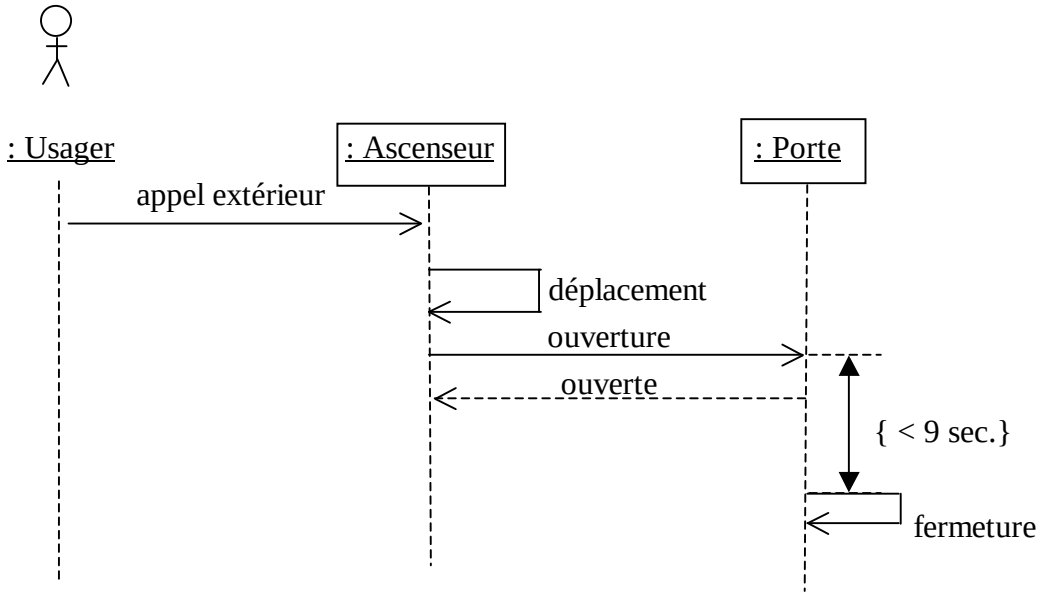


Représentation de la création et de la destruction d'un objet



Exemple de diagramme de séquence

L'ascenceur



Le diagramme de séquence d'un scénario fait apparaître les objets du système.

On peut s'aider des diagrammes de séquence pour construire le diagramme des classes.

4. Les diagrammes de collaboration

Présentation

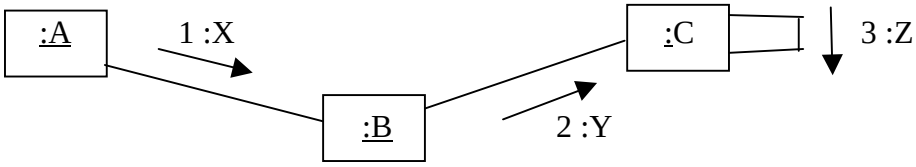
Les diagrammes d'interactions permettent de montrer les échanges de messages entre objet (les interactions) pour une méthode.

On les utilise en général pour les méthodes d'interface.

Diagramme de séquence et diagramme de collaboration sont deux instances possibles pour un diagramme d'interaction ;

Diagramme de séquence	**Diagramme de collaboration**
Représentation orientée client	Représentation orientée développeur
Représentation graphique d'un scénario	
Montre des **interactions entre des objets** dans un enchaînement temporel	
Montre les **messages échangés** entre les objets pour réaliser la fonctionnalité du scénario	
Peuvent être transformés l'un par l'autre	

Exemple



Le scénario débute par un objet A qui envoie un message X à un objet B, puis l'objet B envoie un message Y à un objet C, et enfin C s'envoie un message Z.

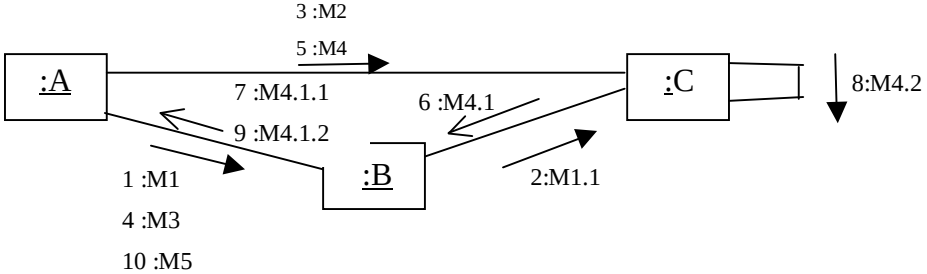
Avantages

Les diagrammes de collaboration sont particulièrement indiqués pur la phase exploratoire qui correspond à la recherche des objets et des classes.

Inconvénients

Seule une petite collaboration est représentable.

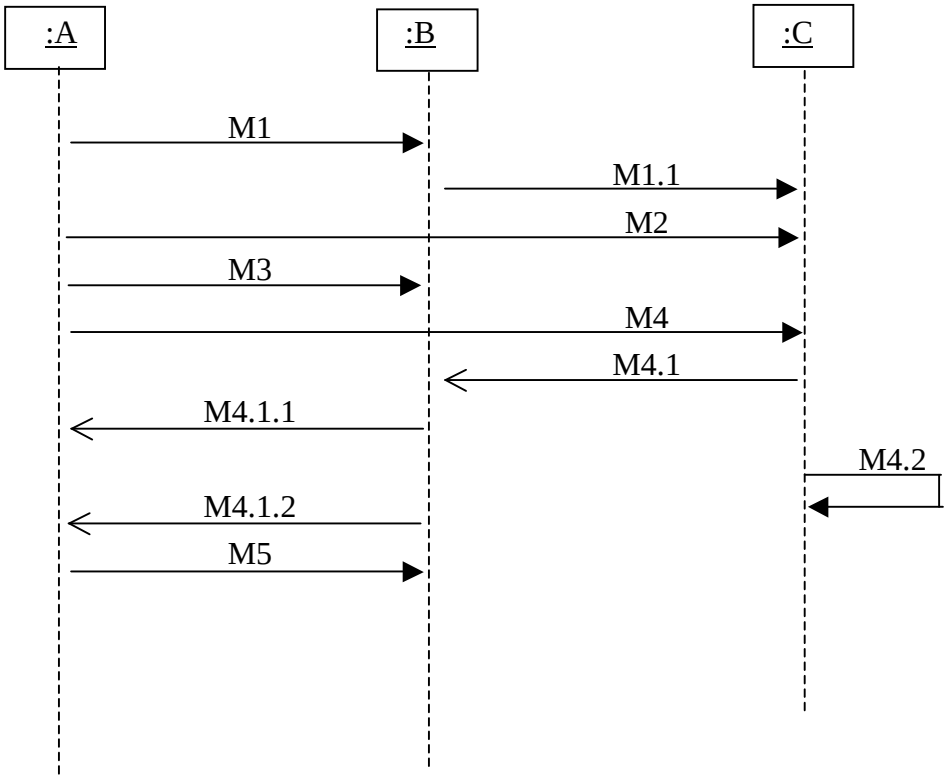
S'il y a beaucoup de messages échangés, le diagramme est peu lisible.



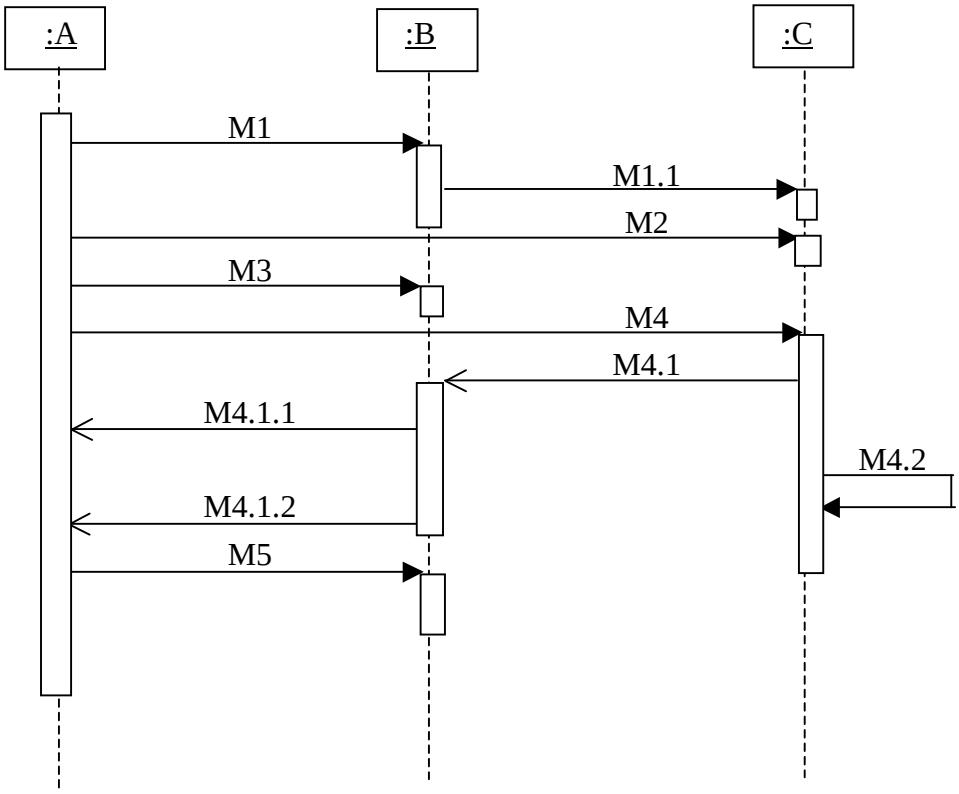
Remplacement par un diagramme de séquence

Si le diagramme de collaboration est trop dense, il vaut mieux utiliser un diagramme de séquence.

Le passage d'un diagramme de collaboration à un diagramme de séquence est automatique.



On peut aussi représenter les périodes d'activité des objets :



5. Les diagramme d'objets

Présentation

Le diagramme d'objets montre **les objets et leurs liens à un moment** de l'exécution du programme.

C'est **un instantané, une photo**, d'un sous-ensemble d'objets d'un système à un instant de la vie du système.

Il permet de rendre plus concrètes et plus claires certaines parties du diagramme de classe.

C'est un diagramme orienté développeur.

Représentation UML

La représentation s'apparente à celle d'un diagramme de classes.

On représente les objets et pas les classes, donc pas les liens d'héritage.

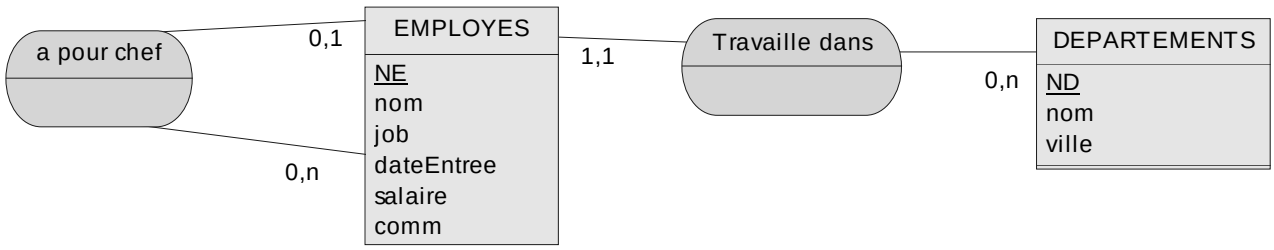
On représente les liens du diagramme de classes entre les objets tels qu'il sont représentés dans un diagramme de classes.

ANNEXES

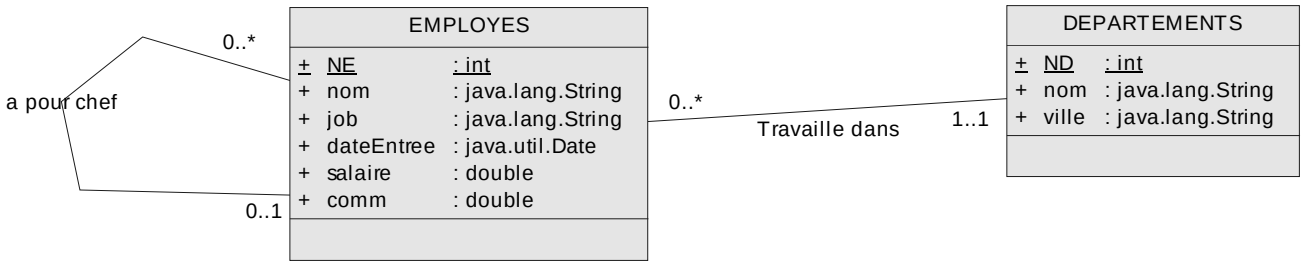
1. Traduction de MEA en UML

Les employés et les départements

MEA



UML



Explications

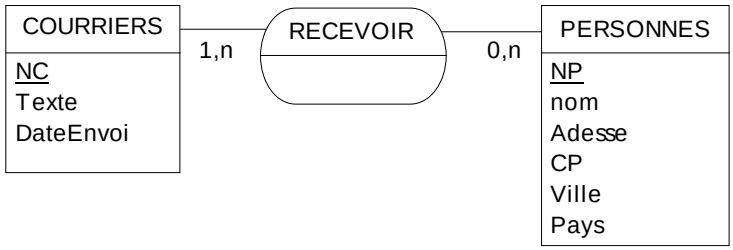
- Tous les attributs sont « + » c'est-à-dire public. En réalité, ils devraient passer « - » , c'est-à-dire privés. Il faut donc faire attention aux résultats des traductions automatiques !
- La notion de clé primaire n'a pas de signification dans un diagramme de classe. En effet, tout objet (instance d'une classe) est caractérisé par ses attributs, ses méthodes et son identifiant qui est son adresse. Cependant, on peut préciser la notion d'identifiant primaire pour les attributs.
- Les cardinalités des associations UML peuvent reprendre le même formalisme que dans le MEA : 0.1, 1.1, 0.N, 1.N.

- La position des cardinalités est inversée par rapport au formalisme MEA : un employé travaille dans 1 et 1 seul département. La cardinalité 1.1 est du coté du département.

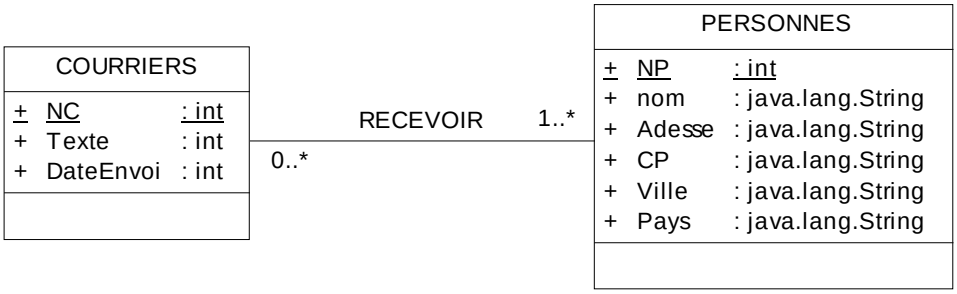
- Les associations UML sont orientées : il peut y avoir des flèches dans un sens ou un autre. Cette notion n'ayant pas de sens dans le MEA, l'association sera rendu navigable dans les deux sens, ce qui conduit à l'élimination des flèches.

Les courriers : association non hiérarchique sans attributs

MEA



UML



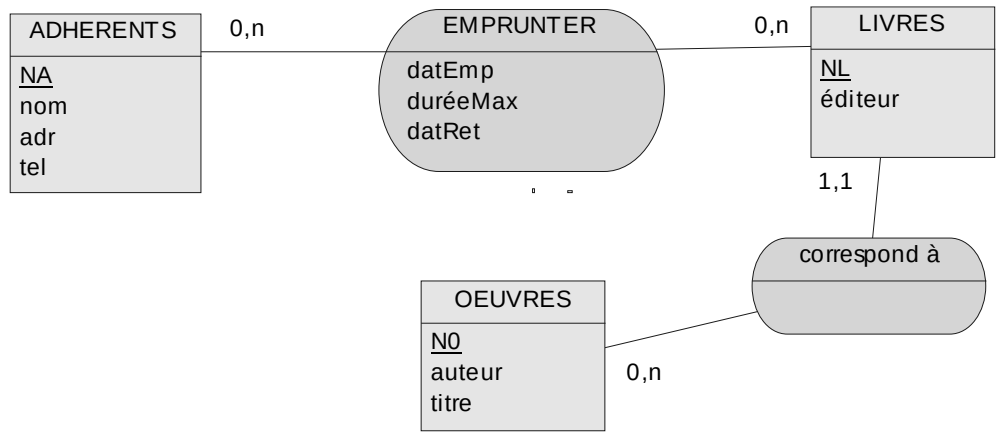
Explications

- Tous les attributs sont « + » c'est-à-dire public. En réalité, ils devraient passer « - » , c'est-à-dire privés. Il faut donc faire attention aux résultats des traductions automatiques !

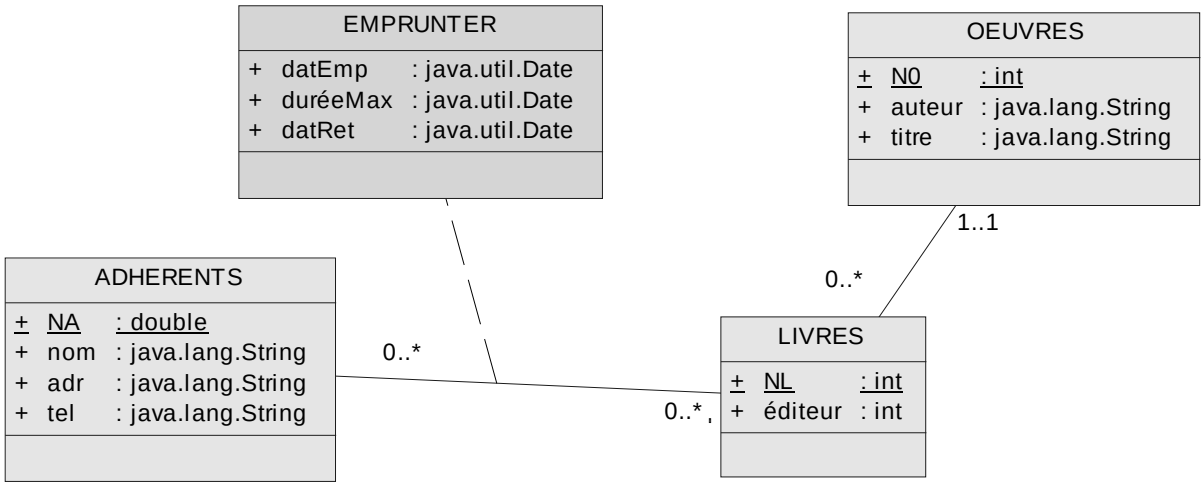
- Les associations non hiérarchiques sans attributs du MEA sont transformées en association dans le modèle de BD UML. La position des cardinalités est inversée : un personne peut recevoir 0 ou N courriers (ça peut être 0 si on ne lui à jamais écrit). Un courrier est envoyé à 1 ou N personnes. Au moins 1 car il n'y a pas de courriers qui ne soit pas envoyé.

La bibliothèque : association non hiérarchique avec attributs et classe-association

MEA



UML

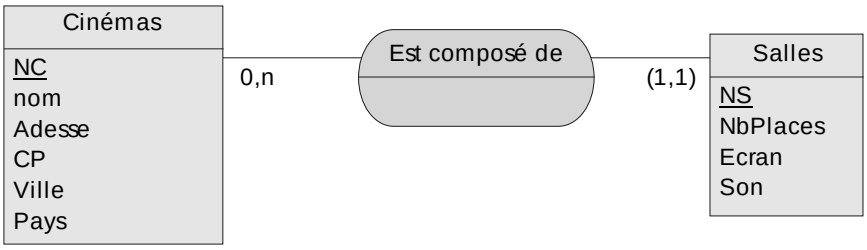


Explications

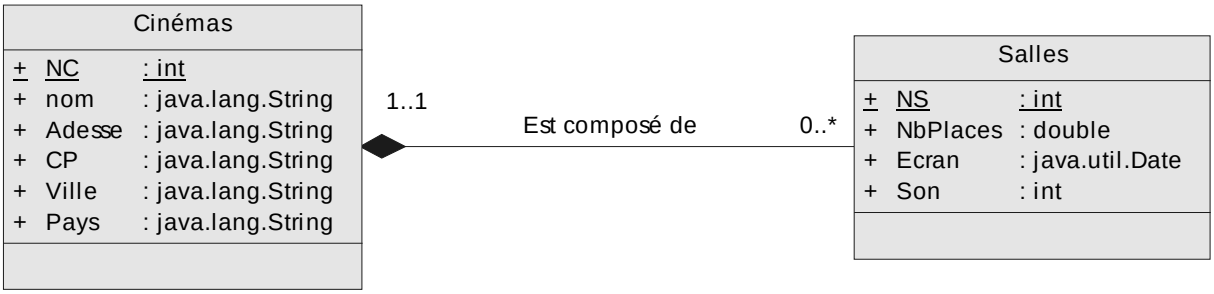
- Tous les attributs sont « + » c'est-à-dire public. En réalité, ils devraient passer « - » , c'est-à-dire privés. Il faut donc faire attention aux résultats des traductions automatiques !
- En UML, on ne peut pas mettre d'attributs sur les associations. Les associations non hiérarchiques avec attributs du MEA donnent lieu dans le modèle de BD UML à des classes-associations.
- Une classe-association est une association classique à laquelle est raccrochée une classe dont les attributs proviennent de l'association non hiérarchique du MEA.

Les cinémas : identifiant relatif et composition

MEA



UML

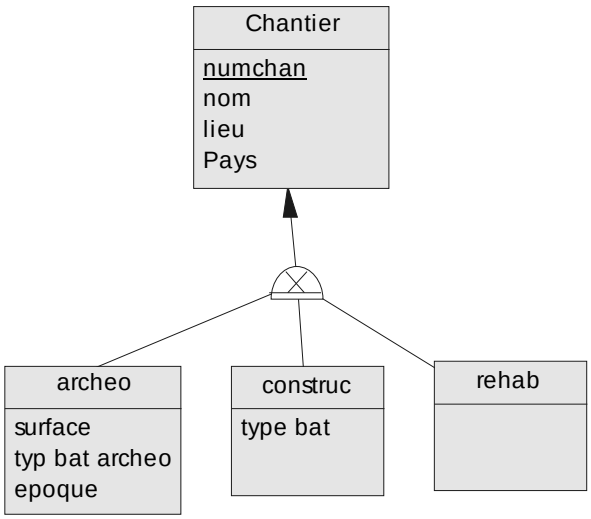


Explications

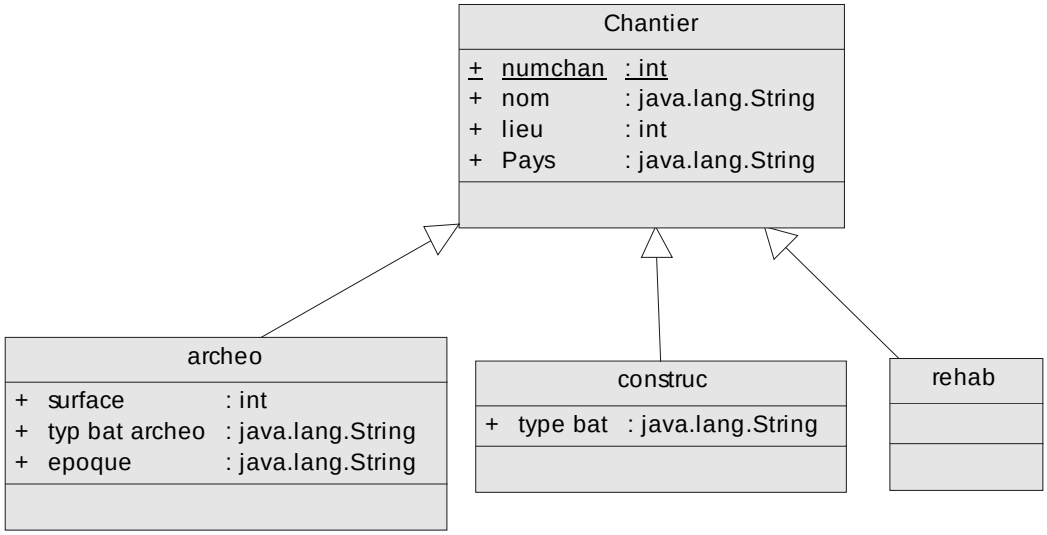
- Tous les attributs sont « + » c'est-à-dire public. En réalité, ils devraient passer « - » , c'est-à-dire privés. Il faut donc faire attention aux résultats des traductions automatiques !
- L'identifiant relatif du MEA est transformé, dans le modèle de BD UML, en une association de composition : losange plein (et pas creux !) du coté du composé.
- La composition signifie que si on supprime de cinéma, alors on supprime aussi les salles.

Les chantiers : héritage

MEA



UML



Explications

- Tous les attributs sont « + » c'est-à-dire public. En réalité, ils devraient passer « - » , c'est-à-dire privés. Il faut donc faire attention aux résultats des traductions automatiques !
- L'héritage dans les MEA se traduit par un héritage dans le modèle de BD UML.
- Dans notre exemple, l'héritage est de type XT, ou + (X souligné dans notre formalisme) : ce qui veut dire qu'il y a couverture et disjonction : un chantier ne peut être que d'une seule espèce (disjonction, X) et il est forcément d'une espèce donnée (couverture, T).
- Les caractéristiques de disjonction, X, et de couverture, T, ne sont pas représentées dans le modèle de BD UML.

2. Génération de C++

Classe et association

Association 1 - *


```
class B;
class A
{
private:
  int a;
  B** vers_b;
};
```

```
class A;
class B
{
private:
  int b;
  A* vers_a;
};
```

Association 0.1 - 10


```
class B;
class A
{
private:
  int a;
  B* vers_b[10];
};
```

```
class A;
class B
{
private:
  int b;
  A* vers_a;
};
```

Association mono-navigable


```
class B;
class A
{
private:
  int a;
  B* vers_b[10];
};
```

```
class B
{
private:
  int b;
};
```

Classe et agrégation


```
class B;
class A
{
private:
  int a;
  B* vers_b[10];
};
```

On retrouve la même chose qu'avec une simple association.

Classe et composition


```
class B;
class A
{
private:
   int a;
   B* vers_b;
};
```

On retrouve la même chose qu'avec une simple association.

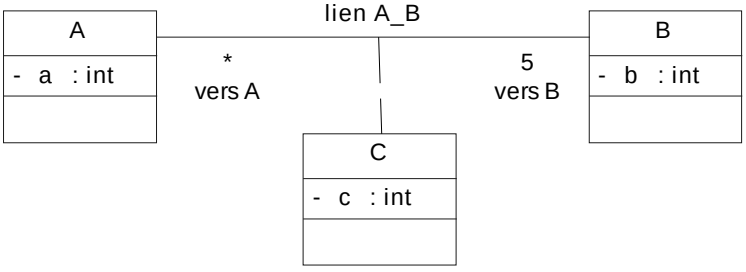
Héritage


```
#include <B.h>
class A : public B
{
private:
   int a;
};
```

```
class B
{
private:
   int b;
};
```

On retrouve la même chose qu'avec une simple association.

Classe association



```
class C;
class A
{
private:
   int a;
  C* lien_A_B;
};
```

```
class C;
class A
{
private:
   int a;
   C* lien_A_B;
};
```

```
class A;
class B;
class C
{
private:
   int c;
   B* vers B[5];
   A** vers A;
};
```

A noter que la production de code par Power AMC est fausse : la classe C devrait pointer que 1 A et 1 B et les classes A et B vers plusieurs C.