Méthodes objets – bref historique

Le développement dans les années 80 du courant objet a conduit à l'apparition de méthodes permettant de couvrir la partie analyse et conception du cycle de développement du logiciel. Une cinquantaine de méthodes ont ainsi été proposées au début des années 90 mais, rapidement, seules trois ont émergé : OMT (Object Modeling Technique), de James Rumbaugh, BOOCH, de Grady Booch et OOSE (Object Oriented Software Engineering), de Ivar Jacobson. Les méthodes OMT et BOOCH ont été les plus diffusées en France. La méthode OOSE s'est imposée dans le monde objet pour la partie formalisation des besoins.

James Rumbaugh et Ivar Jacobson ont alors rejoint Grady Booch au sein de Rational Software, dans le but de fusionner leurs méthodes (en 1994). C'est ainsi qu'est né UML (Unified Modeling Language), dont la première version (norme UML 1.1) est publiée en 1997 par l'OMG (Object Management Group).

Cette étape a permis une convergence des notations dans les domaines de l'analyse et de la conception objet. D'autres versions suivront : UML 1.3 en 2000, UML 1.5 en 2003 et UML 2.0 en 2004…

UML propose un certain nombre de règles normalisées, d'écriture ou de représentation graphique, ainsi que des mécanismes ou des concepts communs applicables à l'ensemble des diagrammes (e.g. commentaires, contraintes, …). On cherche à modéliser un système utilisant les techniques orientées objet, depuis la conception jusqu'à la maintenance, à l'aide d'un langage abstrait, compréhensible par l'homme et la machine.

On utilise pour cela des diagrammes, qui correspondent à des modèles mettant en valeur des aspects différents (fonctionnels, statiques, dynamiques, organisationnels) du système. On pourra ainsi donner des descriptions de la structure interne du système (e.g. diagramme de classe) ou des descriptions de son environnement, selon différents points de vue (e.g. diagramme de cas d'utilisation, qui correspond à un point de vue « utilisateur »). Le langage UML va donc nous permettre de représenter, spécifier, construire et documenter les systèmes logiciels.

Une méthode de développement définit généralement à la fois un langage de modélisation et une démarche de conception (voir par exemple MERISE). A contrario, UML propose une notation, dont l'interprétation est définie par une norme, mais pas de méthodologie complète, permettant ainsi son adaptation à différents cas de figure (taille des applications, contexte de développement, etc.) et donc à différents types de processus de développement. Différentes processus complets de développement basés sur UML existent (e.g. RUP, Rational Unified Process, de Booch, Jacobson et Rumbaugh, ou MDA, Model Driven Architecture, proposée par l'OMG), mais ils ne font pas partie du standard UML.

UML s'intègre dans toutes les phases du cycle de vie du développement, de la collecte des besoins au déploiement. Il donne la possibilité d'utiliser les mêmes concepts et notations, sans nécessité de conversion, dans les différentes phases de développement. Il est de ce fait bien adapté au cycle de développement itératif (on précise à chaque itération les degrés d'abstraction) et incrémental (on divise le développement en étapes aboutissant chacune à la construction de tout ou partie du système).

UML repose sur la possibilité d'utiliser plusieurs diagrammes, notamment : le *diagramme de cas d'utilisation* (description des besoins des utilisateurs par rapport au système, i.e. fonctionnalités du système) – le *diagramme de séquence* (description des scénarios de chaque cas d'utilisation, mettant l'accent sur la chronologie des opérations) – le *diagramme de communication* (autre description des scénarios, mettant l'accent sur les objets et les messages échangés) – le *diagramme de classes* (description statique du système qui intègre données et traitements) – le *diagramme d'objets* (représentation des instances de classes permettant d'illustrer le diagramme de classe sur un exemple concret) – le *diagramme d'états-transitions* (description des états des objets en réaction aux événements) – le *diagramme d'activités* (spécification de traitements, proche de l'algorithmique) – le *diagramme de composants* (représentation des constituants logiciels du système) – le *diagramme de déploiement* (description de l'architecture technique du système), etc.

Diagramme de contexte statique

Le diagramme de contexte statique va permettre de délimiter le cadre de l'étude et de recenser tous les acteurs et/ou systèmes qui interagiront avec « notre » système.

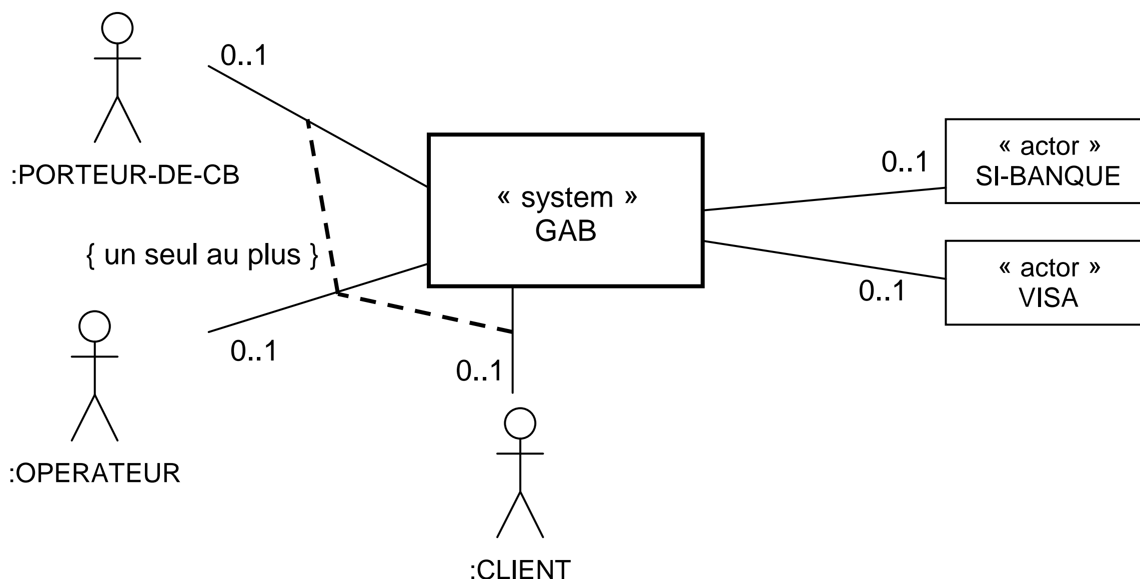
Un *acteur* représente un ensemble cohérent de rôles joués par des entités externes au système informatique : utilisateur humain, dispositif matériel ou autre système. Un acteur correspond souvent à un groupe de personnes ayant le même rôle (par exemple l'acteur *employé*). À l'inverse, une même personne physique peut correspondre à différents acteurs, selon le rôle que l'on considère (par exemple, M. X se connecte en tant que *employé* ou *administrateur* du système de gestion des absences).

Exemple[1]. Considérons un guichet automatique de banque (GAB) de la banque BB. Ce distributeur peut être utilisé par des porteurs de carte bancaire ou par des clients de la banque BB (à qui sont offertes des fonctionnalités supplémentaires). Périodiquement, des interventions d'un opérateur sont nécessaires, par exemple pour recharger le distributeur en billets ou effectuer un dépannage (pendant ce temps, le GAB est inutilisable par les autres usagers).

Le porteur de carte bancaire est naturellement un acteur. On a cependant intérêt à distinguer le *client de la banque* et le *porteur de CB* non client de la banque. Notre système va par ailleurs interagir avec d'autres systèmes : le *SI de la banque* (pour connaître par exemple le solde d'un compte) ou le *système d'autorisation VISA* (pour les autorisations de retrait).

Le diagramme de contexte va permettre de préciser, via les *multiplicités* des associations, le nombre d'acteurs pouvant intervenir *simultanément* sur notre système. Par ailleurs, nous rajoutons une *contrainte* de type XOR entre les trois acteurs humains car un seul d'entre eux peut intervenir à la fois.

Nous avons ainsi le diagramme suivant :



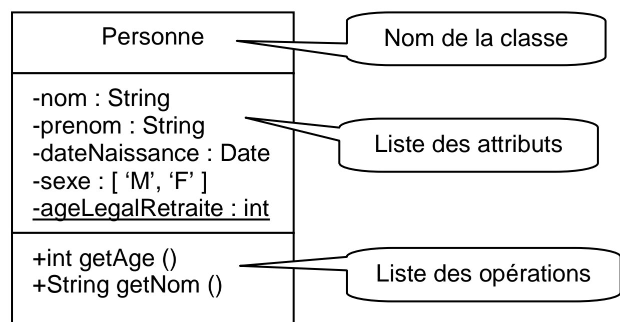
[1] d'après Pascal Roques, *UML 2 par la pratique*, Eyrolles, 6ème édition, 2008.

Objets, Classes, Attributs, Opérations

Une *classe* permet de regrouper des *objets* similaires, c'est-à-dire ayant les mêmes caractéristiques, tant au niveau de leur description que de leur comportement. Une classe correspond ainsi au regroupement des données et des traitements applicables aux objets qu'elle représente. Une classe est donc la description d'un ensemble d'objets ayant une sémantique, des attributs, des comportements (opérations) et des relations (associations) à d'autres classes en commun. Un objet d'une classe C est appelé une *instance* de la classe C.

Les classes peuvent être regroupées au sein de *paquetages* (et ce, récursivement). La classe « Toto » du paquetage « Inter » sera alors désignée par « Inter::Toto ».

En UML, une classe est représentée par un rectangle à trois compartiments (voir ci-contre).

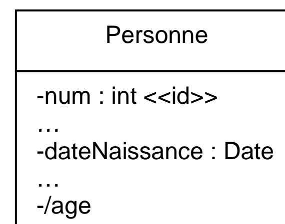


Notons que les noms de classes commencent par une majuscule, les noms d'attributs ou de méthodes par une minuscule (si « en plusieurs mots », les mots suivants sont accolés et débutent par une majuscule).

Suivant ce que l'on souhaite montrer, il est possible de ne faire apparaître que le premier ou les deux premiers compartiments d'une classe.

Les attributs de classe (ayant la même valeur pour tous les objets de la classe) et les opérations de classe doivent être soulignés (un attribut de classe correspond à un attribut de type *static* en C++).

Un objet d'une classe doit avoir une unique valeur pour chacun des attributs de la classe et pouvoir exécuter toutes les méthodes de la classe.



Un attribut *dérivé*, symbolisé par l'ajout d'un « / » devant son nom, peut être calculé à partir d'autres attributs de la classe. Il est possible de déclarer l'un des attributs comme *identifiant* des objets de la classe (symbolisé par <<id>>).

Il est possible de définir les droits d'accès aux différents éléments (attributs, opérations) d'une classe. En UML, quatre niveaux peuvent être définis : *public* (+) pour un élément visible partout, *private* (-) pour les éléments visibles uniquement dans la classe, *protected* (#) pour un élément visible uniquement dans la classe et ses descendantes (voir spécialisation/héritage), et « rien » pour un élément visible uniquement dans le paquetage où il est défini.

Généralement, les attributs sont privés et les opérations sont publiques. On parle alors *d'encapsulation* des données : les données sont protégées, accessibles uniquement à l'aide d'opérations spécifiques, les *accesseurs*. On peut ainsi modifier l'implémentation des données, ou des opérations, sans changer la façon d'accéder aux informations (*interface* offerte par les accesseurs).

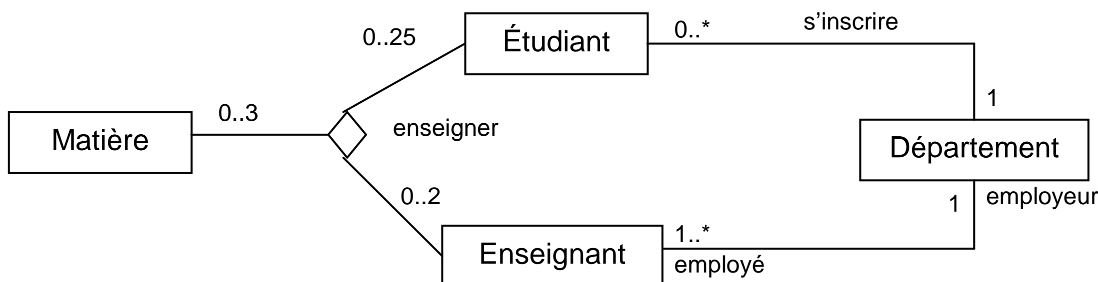
Au niveau conceptuel, on se contente souvent de donner le nom des opérations, parfois avec le type de la valeur retournée, sans préciser la manière de les implémenter. Il s'agit donc simplement d'une *spécification* de la méthode. En UML, la spécification d'une méthode est appelée *opération* et le terme *méthode* est réservé à son implémentation (une opération implémentée devient une méthode…). On appelle *signature* d'une opération (ou d'une méthode), la liste de ses arguments, avec leur type, et le type de la valeur retournée. Par exemple, *Int nombreDeCommandes (Client monClient, int année)*.

Associations

Une *association* représente une relation sémantique entre des objets de deux (ou plusieurs) classes. Elle se représente à l'aide d'un trait plein reliant les classes associées (et d'un losange pour les associations non binaires). Le nom de l'association est indiqué « au milieu » du trait (ou près du losange). Chaque extrémité de l'association correspond au *rôle* de la classe dans cette relation. En pratique, on donne un nom à une association, ou à un ou plusieurs de ses rôles, dans le but d'identifier clairement celle-ci.

Un rôle doit avoir une multiplicité, qui précise le nombre d'objets de la classe correspondante qui peuvent être associés à un unique objet de l'autre classe (attention à la différence avec le modèle entités-associations). La multiplicité est un ensemble non vide d'entiers positifs, à l'exclusion de l'ensemble {0}. Les principales sont : * ou *0..** (plusieurs), *n* ou *n..n* (exactement *n*), *n..** (au minimum *n*, par exemple *1..**) et *n..m* (entre *n* et *m*), où *n* et *m* doivent être des entiers.

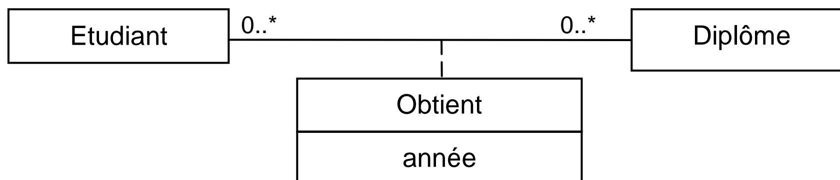
Considérons l'exemple suivant :



Dans cet exemple, un enseignant n'est employé que par un seul département, et chaque département emploie un ou plusieurs enseignants. Attention à la multiplicité des associations ternaires : si je fixe un étudiant (Jean) et un enseignant (Guy), le nombre de matières possibles pour lesquelles ils sont « ensemble » ne peut excéder 3. Un étudiant ne peut avoir plus de 2 enseignants pour une même matière (mais il peut n'en avoir aucun, car il ne suit pas toutes les matières).

Une association est généralement bidirectionnelle alors que son nom (qui est un verbe) induit une direction privilégiée… Cette direction peut être précisée à l'aide d'un « triangle plein » directionnel :

Une association peut avoir ses propres attributs. Comme dans le modèle objet seules les classes peuvent avoir des propriétés, une telle association est alors une *classe-association*, représentée ainsi :



Si les classes reliées par une classe-association ont un identifiant (par exemple « numEtudiant » et « numDiplome » dans l'exemple ci-dessus), les objets de la classe-association sont identifiés par le couple < numEtudiant, numDiplome >. Ainsi, on ne déclare jamais d'attribut identifiant dans une classe-association.

Agrégation, composition

L'agrégation et la *composition* sont des associations particulières, non symétriques, modélisant une relation de contenance. Les agrégations n'ont pas besoin d'être nommées : implicitement, elles signifient « contient », « est composé de »…

La composition est une agrégation *forte*, dans le sens où

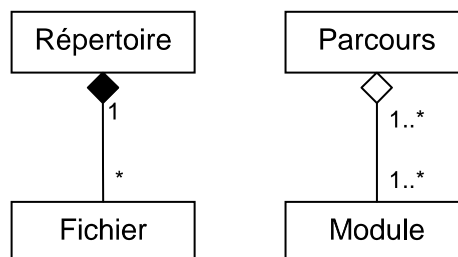
(1) un élément ne peut appartenir qu'à un seul agrégat (ou composite), et

(2) la destruction de l'agrégat entraîne la destruction de tous ses éléments.

Exemples :

- un répertoire contient des fichiers (composition),
- un parcours de formation comprend plusieurs modules (agrégation, un module peut appartenir à plusieurs parcours).

L'agrégation se note à l'aide d'un losange creux (du côté de l'agrégat), la composition d'un losange plein :

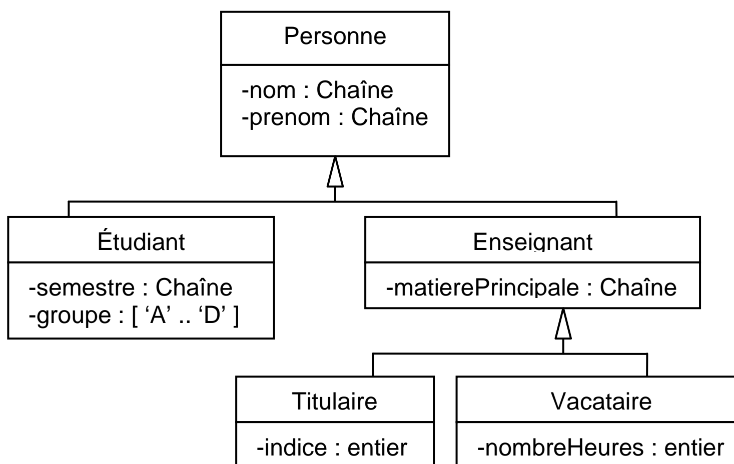


Remarque : Composition et agrégation ne sont que des associations particulières (et peuvent donc être représentées par de simples associations). Leur présence ne fait que renforcer le lien de « contenance » qu'elles induisent (plus lisible sur le schéma).

Spécialisation, généralisation, héritage

L'un des intérêts de l'approche objet est de pouvoir manipuler des concepts abstraits. Cela permet de simplifier la représentation et correspond de fait à la façon doit on voit « le monde », selon différents niveaux d'abstraction : la notion de « véhicule » par exemple est abstraite, il peut s'agir de « voitures », de « camions », etc. puis d'une « Peugeot 106 », puis de la « Peugeot 106 de mon cousin » qui est vraiment concrète… Parler de véhicule permet ainsi de *factoriser* un certain nombre de caractéristiques (vitesse, possibilité d'embarquer un certain nombre de passager, etc.).

Le concept d'*héritage* va permettre de mettre en œuvre ces différents niveaux d'abstraction. Une classe va pouvoir être découpée en sous-ensembles d'objets (*spécialisation*), chacun de ces sous-ensembles correspondant à une classe « moins abstraite ». La « sur-classe » (ou *classe parent*) est une *généralisation* de ses « sous-classes » (ou *classes enfants*) et ce processus peut être répété récursivement. Les sous-classes *héritent* des attributs et des opérations de leur sur-classe.

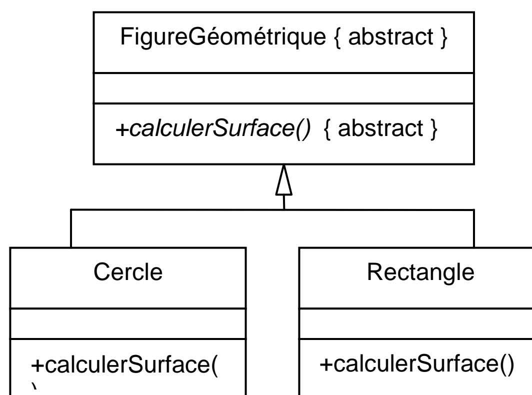


On parle de *polymorphisme* lorsqu'une même opération peut s'appliquer à des objets de différentes classes. Par exemple, l'opération *calculerSurface()* d'une figure géométrique dépend de la nature de la figure (rectangle, cercle, etc.).

Une opération est dite *abstraite* lorsqu'on connaît son entête mais pas la manière dont elle peut être réalisée (elle se situe au niveau d'une classe « trop abstraite »). Une classe est dite *abstraite*, si elle contient une opération abstraite, ou si elle descend d'une classe contenant une opération abstraite non encore réalisée.

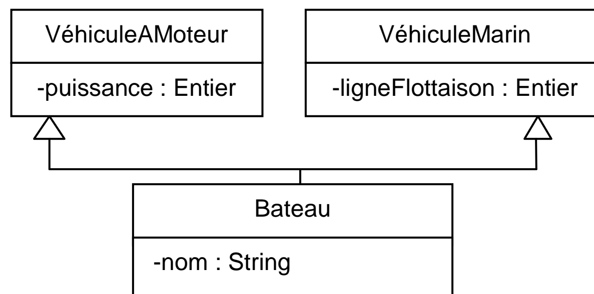
La classe abstraite impose à ses classes filles d'implémenter ses opérations abstraites.

On parle de *surcharge* lorsque qu'une classe fille implémente une opération également implémentée par la classe mère.



Une classe peut parfois être vue comme une spécialisation de plusieurs autres classes. Par exemple, un bateau est à la fois un véhicule à moteur et un véhicule marin.

Attention, si les deux classes parent ont un attribut commun, il est nécessaire de définir une priorité sur les deux généralisations, pour savoir duquel des deux attributs héritera la classe fille…



Navigabilité – Génération de code C++

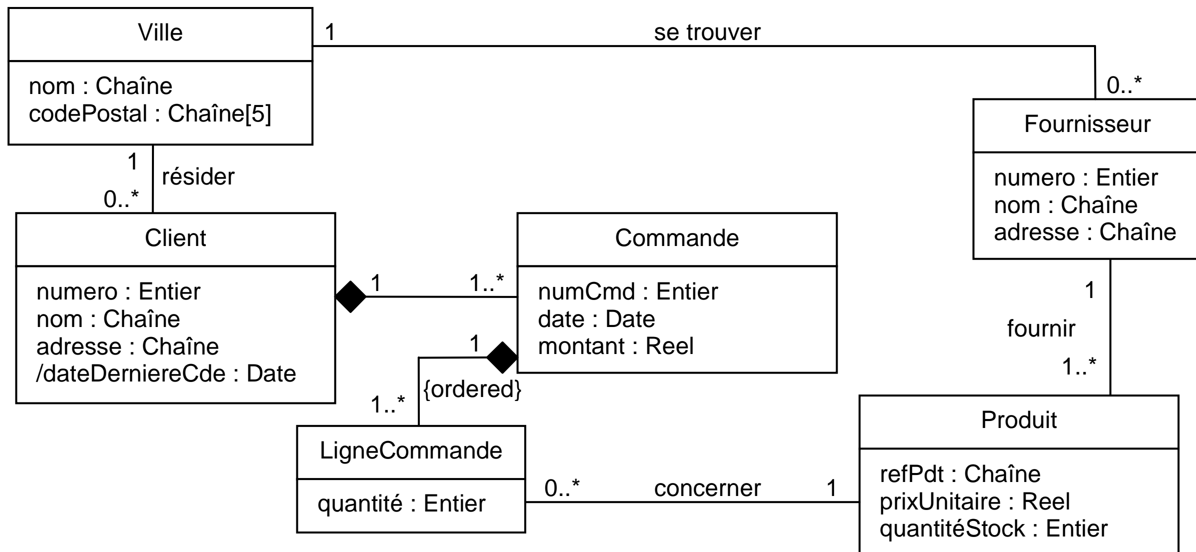
A FAIRE

Diagramme d'objets

Le diagramme d'objets permet d'illustrer un diagramme de classes sur une situation concrète. Ce diagramme permet donc de préciser l'organisation des données. D'une certaine façon, ce diagramme décrit un *instantané* (snapshot) possible du système.

Le diagramme d'objets est composé d'objets (plusieurs instances d'une même classe peuvent apparaître), éventuellement reliés entre eux par des associations, conformément au diagramme de classes qu'il illustre.

Un objet se représente graphiquement comme une classe avec deux compartiments : son nom, puis la liste de ses attributs qui peuvent être *valués*. Le nom d'un objet est souligné. Le nom d'un objet est donné par son identifiant, suivi du nom de sa classe (e.g. albert:Personne, ou 145:Personne). On peut également utiliser des objets *anonymes*, dont le nom est sans importance (e.g. :Personne). Voici par exemple un diagramme de classes, puis un diagramme d'objets l'illustrant.



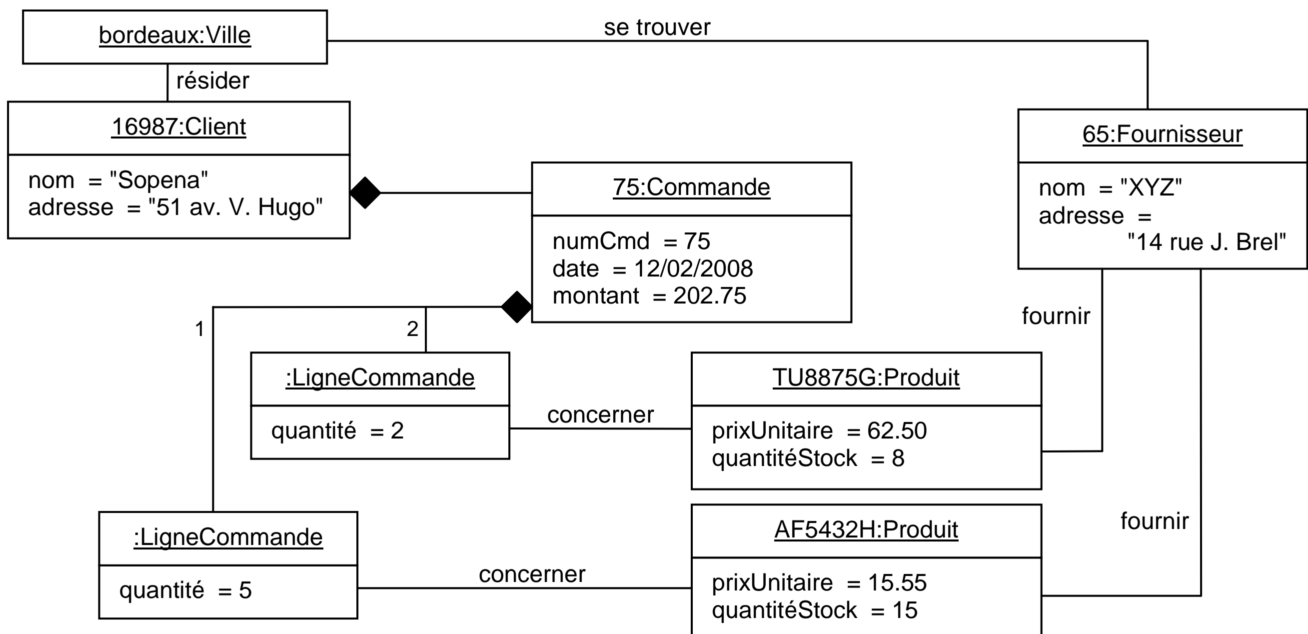
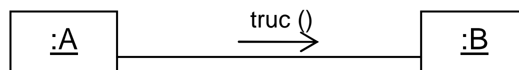


Diagramme de communication

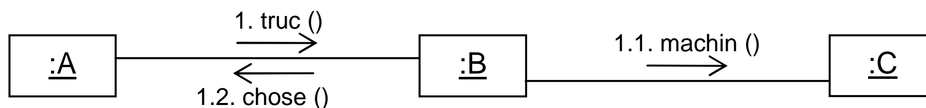
Un *diagramme de communication* est toujours associé à une *fonctionnalité particulière* du système. Ce diagramme permet de décrire le *comportement* des objets impliqués dans la réalisation d'une fonctionnalité donnée.

Les objets communiquent entre eux en s'échangeant des *messages*. Un objet :A envoie par exemple le message « truc () » à l'objet :B pour déclencher l'exécution de l'opération « truc () » (membre de la classe B). Cet échange se modélise ainsi :

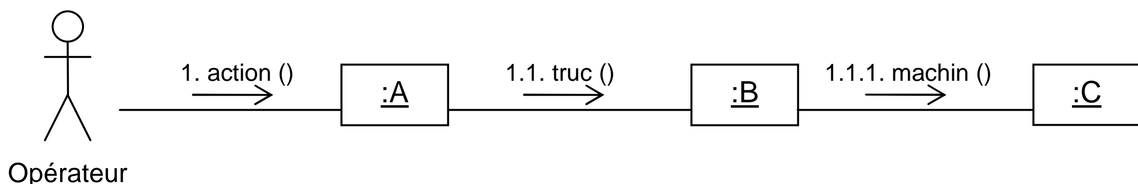


Les objets :A et :B sont des *instances* des classes A et B respectivement. Le lien qui les unit est une instance de l'association entre les classes A et B, et est appelé *connecteur* (il permet de « transporter » le message truc…). Un connecteur doit correspondre à une association entre les classes correspondantes dans le diagramme de classes.

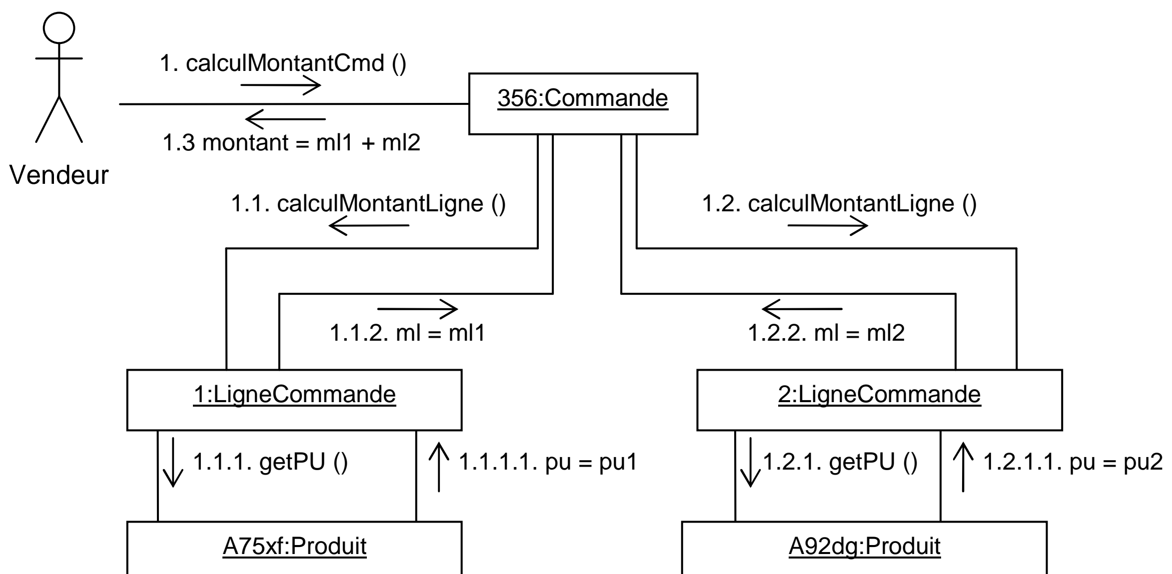
Lorsque plusieurs messages sont présents, on doit les numéroter afin de préciser leur ordonnancement dans le temps (la réception du message n° i déclenche l'émission des messages i.1, i.2, etc.) :



Un message initial, déclencheur, peut venir de l'extérieur (i.e. d'un acteur, via l'interface du système) :



Voici par exemple le diagramme de communication de la fonctionnalité « calculer montant commande » :



Lexique Français-Anglais

Français	Anglais
acteur	actor
activité	activity
cas d'utilisation	use case
composant	component
contrainte	constraint
diagramme d'activité	activity diagram
diagramme d'états-transitions	statechart diagram or state diagram
diagramme d'interaction	interaction diagram
diagramme des objets	object diagram
diagramme de cas d'utilisation	use case diagram
diagramme de classes	class diagram
diagramme de communication	communication diagram
diagramme de déploiement	deployment diagram
diagramme de paquetage	package diagram
diagramme de séquence	sequence diagram
enchaînement d'activités	activity edge
état	state
généralisation	generalisation
héritage	inheritance
instance	instance
interface	interface
lien (entre objets)	link (between objects)
méthode de classe	class method
objet	object
paquetage	package
processus	process
relation d'extension	extension relationship
relation d'inclusion	inclusion relationship
scénario	scenario
spécialisation	specialisation
stéréotype	stereotype
système	system