Chapitre 1

Méthode de développement du logiciel avec UML

Dans ce livre nous traitons à la fois de modélisation et de spécification avec UML pour développer du logiciel. Dans cette introduction, nous positionnons le cours et la contribution dans le large espace des méthodes de développement du logiciel à objets. Ce dernier reste le support d'implantation de la majorité des applications développées actuellement, y compris dans la vision service actuelle.

Après un rapide rappel de ce qu'est le développement du logiciel nous mettons en évidence les composantes d'une méthode de développement avec UML, notamment les concepts, la notation, le processus et les outils.

Les concepts et la notation UML sont examinés en section 2. Nous ne détaillons pas la *volumineuse* notation mais décorticons les principes majeurs. La structure des spécifications (les modèles) est relativement complexe, elle est fondée sur la notation d'architecture. Nous détaillons ce point dans la section 2.4.

Le processus met en évidence les activités de développement et leur enchaînement. À ce niveau, la pratique est loin d'être unifiée. Ici aussi un exposé exhaustif est rédhibitoire. Dans la section 3, nous présentons les principes, les grandes activités et les principales approches, en insistant notamment sur le RUP et le 2TUP. Puis nous croisons activités et notation dans la section 3.5 en indiquant quels diagrammes sont utilisés à quelle étape. En effet, l'utilisation des notations varie en fonction des étapes du processus de développement utilisé. Pour ce faire le processus est simplifié et reste commun à la plupart des méthodes à objets

La section 4 propose des pointeurs sur des outils autour d'UML (dessin, modélisation, vérification, génération de modèles ou de codes) et quelques références bibliographiques ou webographiques ciblées.

1 Le développement du logiciel

Développer c'est passer de l'idée au code (exprimer, programmer, vérifier) avec méthode (dans le bon ordre, éviter l'anarchie, travailler en groupe) et qualité (modèles corrects, fiables, évolutifs. . .- processus efficace, rentable...).

- Dans un développement individuel, l'analyse peut être légère ou dans la « tête », la programmation incrémentale et itérative sur le test comme dans les méthodes agiles ou XP (*eXtreme Programming*).
- Lorsque le développement grossit, on communique et collabore, on a donc besoin de mettre en place une organisation (organiser et répartir les tâches, définir les participants, gérer la communication et les normes d'échange, suivre l'avancement...).
- La troisième dimension prend en compte le client. Aux éléments précédents on ajoute une dimension *coût* (financier, temporel, humain...) qui fait aussi intervenir une analyse

de compétence et de performance (rentabilité). De nouvelles étapes sont nécessaires dans la gestion du projet (opportunité, faisabilité, risques, décisions) ainsi qu'une prise en compte du rôle du client, notamment pour la validation. De manière orthogonale, on doit prévoir dans les projets l'assurance qualité, l'automatisation... La qualité des résultats est liée d'une part à la satisfaction du client et d'autre part à la facilité à maintenir les applications. La qualité du processus se voit dans le respect des délais et des coûts et donc dans sa rentabilité.

– La quatrième dimension met en jeu le prestataire (SSII ou service interne) qui développe ou assure la maintenance corrective ou évolutive (TMA, tierce maintenance applicative). Ses préoccupations sont similaires à celles du client, mais du point de vue de la production : gestion des ressources humaines (carrières, formations...), gestion commerciale (rentabiliser les produits sur plusieurs clients), rentabilité en réutilisant tout ou partie de la la production, automatisation du codage...

Un *projet informatique* met donc en œuvre des aspects techniques (méthode de développement), organisationnels (gestion de projet) et méthodologiques (qualité). Il met à contribution différents acteurs avec des objectifs et des visions différentes. L'objectif majeur est de produire du logiciel de qualité, rationaliser le développement, rentabiliser les investissements. Le génie logiciel vise à répondre à ces besoins : mettre en œuvre des moyens pour réaliser du logiciel de qualité en respectant des contraintes de coûts. Le *génie logiciel* est l'art de construire INDUSTRIELLEMENT du logiciel en s'appuyant notamment sur des outils intégrés, de type AGL (Atelier de Génie Logiciel). Une bonne référence à ce sujet est [Pre10].

Nous classons les applications en trois catégories :

1. Les systèmes d'information (de gestion) sont caractérisés par un stockage important d'informations et de nombreux traitements simples en consultation et mise à jour de ces informations *e.g.* achat par Internet, gestion adminsitratives.
2. L'informatique temps réel (automatismes, contrôle de processus) se caractérise par une réactivité très forte du système d'information *e.g.* pilotage automatique d'avion, surveillance de centrale nucléaires. L'interface entre le système d'information et son objet naturel (le processus) prend ici une grande importance (capteurs, actionneures).
3. Le calcul scientifique se caractérise par des calculs complexes et nombreux *e.g.* simulation, météo, imagerie, web sémantique. Un critère important est l'efficacité des traitements.

Nous estimons que ces trois catégories représentent, assez grossièrement, les différents types de systèmes d'information que nous sommes susceptibles de modéliser ici.

Dans cet ouvrage, nous considérons principalement les aspects techniques du projet informatique, la méthode de développement. Ainsi que nous l'avions détaillé dans le chapitre 1 de [AV01a], une **méthode** est à la fois une philosophie dans l'approche des problèmes, une démarche ou un fil conducteur dans la résolution, des outils d'aide et enfin un formalisme ou des normes. Précisons ces volets dans le cas du développement à objet.

– La philosophie est celle des objets, à savoir un ensemble d'acteurs qui collaborent pour réaliser les fonctionalités du système. Le chapitre 6 du tome 2 [AV01b] détaille ces aspects de la conception à objets.
– Le formalisme est la norme UML, le standard adopté universellement. Nous en donnons les principes généraux dans la section 2. UML inclut des notations pour les trois catégories d'application que nous avons mis en évidence.
– Il n'y a pas de démarche explicite, mais des principes acceptés : développement est itératif et incrémental, basé sur les cas d'utilisation et l'architecture. Le processus unifié

est une proposition de Rational (IBM) qui illustre bien cette démarche [RJB99]. Nous détaillons les éléments d'un processus dans la section 3.
– Les outils couvrent une large gamme de produits, du logiciel de dessin comme Visio à l'AGL complet permettant le *round trip* (lien étroit entre le code et son modèle), le déploiement et la gestion de la qualité. Nous proposons un échantillon dans la section 4.

2 La notation UML

La notation UML inclut un grand nombre de concepts autour de l'objet mais aussi de l'analyse des besoins (acteurs, cas d'utilisation), de la conception du logiciel (composants, modules, processus) ou de l'implantation (nœuds, liaisons, déploiement). La raison est que cette notation est conçue pour décrire des modèles couvrant l'ensemble du cycle de développement. De plus certains concepts sont perçus à des niveaux d'abstraction différents. Par exemple, les opérations des objets sont décrites plus finement à la conception. UML vise à unifier les notations des nombreuses méthodes à objets. Elle est donc un compromis. Heureusement, les auteurs se sont entendus pour définir une notation de base, que chacun peut ensuite étendre par le mécanisme de stéréotypage. Par exemple, une classe abstraite est une classe affinée par le stéréotype `«abstract»`. Ainsi, une limite raisonnable est donnée au nombre de concepts.

Nous désignons par UML 1.X la famille de langage de première génération d'UML. Cette famille est caractérisée par une approche **unificatrice** de la notation, qui fait d'UML un langage universel. Dans cette vision, UML s'inspire de nombreuse références et regroupe un maximum de concept pour correspondre à tous les besoins. Les principales influences sont :

– Booch : Catégories et sous-systèmes
– Embley : Classes singletons et objets composites
– Fusion : Description des opérations, numérotation des messages
– Gamma, et al. : Frameworks, patterns, et notes
– Harel : Automates (Statecharts)
– Jacobson : Cas d'utilisation (use cases)
– Meyer : Pré- et post-conditions
– Odell : Classification dynamique, éclairage sur les événements
– OMT : Associations
– Shlaer-Mellor : Cycle de vie des objets
– Wirfs-Brock : Responsabilités (CRC)

On peut aussi affirmer que UML 1.X vise à satisfaire les utilisateurs (ceux qui développent avec UML). UML répond en ce sens à la cacophonie des méthodes et notations des années 1990.

Depuis, les principes fondateurs ont évolués, notamment sous l'impulsion des architectures applicatives émergentes (composants, Architectures de logiciels (ADL), services) mais aussi de l'approche l'*ingénierie des modèles* (IDM) prônée par l'OMG (*Object Management Group*) sous l'appellation *Model Driven Architecture* (MDA). La famille UML 2.X marque cette double rupture, dans laquelle UML devient un langage pivot pour une famille de langages spécifiques, les *Domain Specific Languages* (DSL). De ce fait, la sémantique est devenue moins contraignante pour épouser les sémantiques précises des DSL. UML 2.X n'est plus portée par les utilisateurs mais par les « fournisseurs » d'outils pour les modèles.

Présenter la notation UML est un exercice assez délicat car elle est complexe. Cette complexité trouve ses sources dans le nombre de concepts représentés, la variété d'utilisation de ces concepts et la confusion possible entre la notation et le modèle de la notation (le métamodèle) qui permet de l'exprimer.

UML 1.X disposait d'un guide de la notation ou d'un manuel de référence de la notation. Dans UML 2.X, la spécification de superstructure donne la vision pour l'utilisateur, mais c'est trop technique pour un manuel utilisateur tandis la spécification d'infrastructure, orientée vers l'architecture, donne bien l'idée d'une référence pour les fournisseurs d'outils.

Compte-tenu des remarques précédentes, nous n'avons pas adopté une présentation rigoureuse et complète de la méthode. Nous évitons de présenter l'ensemble des concepts puis les diagrammes dans lesquels on les retrouve. Nous évitons aussi d'utiliser le métamodèle comme support car il nous éloigne des concepts. Ces deux alternatives entraînent une description structurelle, longue et rébarbative, de la notation. Nous avons choisi de présenter les concepts dans leur contexte d'utilisation habituel.

Avertissement

Cet ouvrage n'est pas conçu comme un manuel de la notation : tous les concepts ne seront pas illustrés individuellement par des exemples. Nous avons choisi une approche plus globale, en commentant la notation sur des exemples plus « large ». Pour un exposé de la notation, consulter [Aud09, Fow04, PP05, Bal06]. Noter que les manuels de référence de l'OMG (infrastructure [Gro11b], superstructure [Gro11c]) sont orientés par l'approche MDA, c'est-à-dire plus pour les manipulateurs de modèles que pour les utilisateurs finaux de la notation, c'est pourquoi nous ne les conseillons pas ici.

Dans cette section, nous traçons les grandes lignes de la notation : les éléments généraux, les concepts fondamentaux, les outils de structuration et de présentation.

2.1 Les éléments généraux

Nous présentons dans cette section quelques éléments et mécanismes généraux de la notation qui nous semblent nécessaires à la compréhension de la suite du chapitre.

Terminologie

La notation UML inclut trois sortes de briques : les éléments, les relations et les diagrammes [RJB98]. Les éléments de modélisation représentent les concepts (au sens où nous l'avons entendu jusqu'ici) et des facilités de notation (paquetages, notes, contraintes). Les relations représentent des liens entre éléments de modélisation [1]. Elles sont détaillées dans la section 2.2. Un paquetage est un élément qui regroupe éléments et relations. La notion de diagramme ne fait pas partie des éléments de modélisation, contrairement au modèle qui est un paquetage. Un diagramme est une vue de l'utilisateur sur un modèle.

Stéréotype

La notion de **stéréotype** est un élément clé de l'extensibilité de la notation UML et de son adaptabilité aux AGL et aux méthodes propriétaires. Chaque concept de la notation peut être "spécialisé" par stéréotype. Le stéréotype est une annotation qui enrichit la description, mais n'intervient pas a priori dans les vérifications des modèle produits. Par exemple, une classe abstraite est un stéréotype de la classe. Les stéréotypes sont représentés par des doubles chevrons « ». Au fil des versions d'UML, certains stéréotypes ont été intégrés directement dans la notation. Par exemple, un nom de classe en italique indique que la classe est abstraite.

1. Bien que nous les présentions à part, les relations sont aussi des éléments de modélisation au sens du métamodèle d'UML.

Collaboration, interaction

La notion de collaboration a quelque peu perdu de la valeur au fil des évolutions d'UML. Une **collaboration** décrit une structure d'éléments qui collaborent selon des rôles déterminés. Cette terminologie provient de la méthode Fusion, une des sources d'inspiration d'UML. C'est une vision **statique** (structurelle) des liens entre objets. Il est d'usage de dire qu'une collaboration « implante » un cas d'utilisation.

Une **interaction** décrit un échange entre les éléments de la collaboration. C'est la vision **dynamique** (comportementale) des liens entre objets. On dira qu'une interaction se fait au travers d'envois de messages ou de signaux. Dans UML, deux points de vue équivalents représentent les interactions : les diagrammes de séquences qui insistent sur l'enchaînement et la causalité des interactions (vision temporelle) tandis que le diagramme de communications (ou de collaborations) qui met plus en évidence une vision spatiale de la collaboration. Le support de la collaboration est un réseau d'objets interconnectés (objets et liens).

> Dans UML 2.X, il y a clairement une dichotomie mais aussi une dualité entre la vision structurelle et la vision comportementale du système. Est-ce un retour aux sources des méthodes traditionnelles et à la vision Merise de la systémique ?

Paquetage

Un **paquetage** est un mécanisme qui permet de grouper des éléments. Il permet de structurer la spécification mais ne correspond pas à une abstraction d'un élément de conception comme le composant. Les paquetages sont surtout utilisés pour regrouper des classes. Un paquetage peut contenir d'autres paquetages. Le paquetage est représenté par un dossier. Des stéréotypes du paquetage le spécialisent en fonction du contexte. On parle de `catégorie` pour les paquetages de la vue logique et de `sous-systèmes` pour les paquetages d'implantation.

Comme pour les autres notations, plusieurs interprétations sont possibles. Dans une première interprétation, un paquetage est un module. Les relations entre paquetages sont alors des variantes de la relation de dépendance et on évite, autant que possible, les dépendances circulaires. Le critère de cohérence est soit une proximité logique (les cassettes et les exemplaires), soit de fonctionalité (objets interfaces, objets métiers)... Une autre interprétation est le découpage "physique" en sous-systèmes.

Système, sous-système

Un système est l'élément qu'on développe et pour lequel on construit des vues. Un sous-système est une partie d'un système. Chaque sous-système d'un système est représenté par un paquetage. La relation principale est l'agrégation mais la spécialisation est aussi autorisée. Le paquetage d'un sous-système comprend trois éléments : l'interface (opération), les éléments de spécification et les éléments de réalisation. Les modélisations faites pour un système peuvent l'être identiquement pour un sous-système. Autrement dit, le discours s'applique quelle que soit l'échelle.

Espace de nommage

Un *espace de nommage* permet de définir un contexte lexical pour les éléments de modélisation. De nombreux concepts sont des expaces de nommages : les paquetages bien sûr, mais aussi les classes, les opérations ou même les états. Un élément sera donc désigné par son nom et l'ensemble des espaces de nommages qui le contiennent hiérarchiquement. On utilise le séparateur `::` pour construire le nom complet, par exemple `pack::class:op(param)`.

2.2 Les relations dans le modèle à objets

Les relations permettent de structurer les éléments d'un modèle à objets. Ayant des objets et des classes, nous trouvons naturellement des relations entre instances (objets), des relations entre types (classes, *classifiers*) et des relations entre instances et types.

i) Relations entre objets et types

La relation entre un objet et sa classe est la relation d'**instanciation**. Un objet est instance d'une seule classe qui définit sa structure et son comportement.

ii) Relations entre objets

La base de l'interaction entre objets est l'envoi de message qui suppose un médium, une relation de l'objet client (émetteur) et l'objet serveur (receveur). Le client invoque un service du serveur, c'est une relation de **clientèle**, ou de **délégation**. Par exemple, on peut « demander » à une cassette quel est son film. C'est une relation fondamentale dans un modèle à objets, qui se représente en UML par un **lien** entre objets. Les liens sont abstraits au niveaux des classes par une **association** entre les classes de ces objets.

Un objet (le composé) peut inclure un autre objet (le composant). Cette relation de **structuration** est appelée **agrégation**, relation **tout-partie**. Elle induit une relation de clientèle de l'objet composé vers l'objet composant. L'agrégation est une relation dont les contours ne sont pas toujours bien définis [HSB99] car elle recouvre divers aspects :

- Visibilité. L'objet composant est encapsulé dans l'objet composé. De ce fait il doit suivre les règles de visibilité définies par celui-ci. En particulier, il se pose la question de savoir à quels objets le composant peut être associé : uniquement à des objets composants du même composé ou alors à n'importe quel objet du système ?
- Durée de vie. Quelles sont les règles de synchronisation entre la vie de l'objet composé et celle de ses composants ? Y-a-t'il suppression en cascade ?
- Evolution dynamique. L'objet composant a-t-il un flot de contrôle indépendant (concurrent) de l'objet composé ou son flot est-il assujetti à celui du composé ?

Si l'objet composant est fortement corrélé à l'objet composé (définition à fixer sur les critères ci-dessus) alors on parle de d'agrégation forte ou de **composition**.

Dans une modélisation à objets, les objets sont rarement représentés, on modélise plutôt leurs classes, les relations ci-dessus sont donc abstraites au niveau des classes. Illustrons les différences sur un exemple via la notation UML. Soient les trois relations de la figure 1.

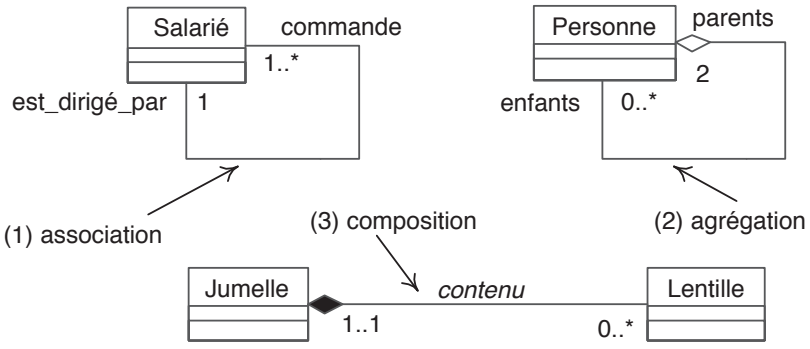

Figure 1 : *Association, agrégation, composition*

1. L'association symbolise des liens entre deux instances des classes associées. Un salarié est dirigé par un autre salarié. Le patron est dirigé par lui-même. Dans une association, un objet peut être lié à lui-même.

2. Une personne a deux parents mais ne peut être parent d'elle-même. L'agrégation est pertinente ici car c'est une association non réflexive (un objet n'est pas lié à lui-même) et les objets ont des lignes de vie propres (parents et enfants ont une vie indépendante !).
3. Les lentilles d'une jumelle font partie intégrante de la jumelle. Cette dépendance forte s'exprime en UML par une relation de composition. La composition est exclusive.

iii) Relations entre classes

Il existe deux types de relations entre classe : l'utilisation et l'héritage. Nous traitons aussi de la généricité.

Relation d'utilisation La relation d'**utilisation** exprime le fait que pour réaliser ses fonctionalités un objet utilise les services d'autres objets. Elle correspond à l'importation modulaire : un module utilise les services d'un autre module pour réaliser ses propres services. Cette structuration entre classes peut être affinée. Etudions quelques critères d'affinement, qui font parfois la différence entre deux modèles objets.

- Agrégation ou composition : ce point a été examiné dans le point ii).
- Sens de la relation : pour l'agrégation, la relation est orientée du composé vers le composant. Pour la relation de clientèle, plusieurs cas se présentent. Si toutes les instances d'une classe A sont clientes (au travers d'envois de messages) d'instances d'une classe B alors une relation d'utilisation existe entre la classe A et la classe B. Si inversement les instances de la classe B sont clientes des instances de la classe A, alors la relation d'utilisation est symétrique (ou bidirectionnelle). Par exemple connaissant la cassette, on peut demander quel est le film enregistré et connaissant le film, on peut demander quels sont les exemplaires en boutiques.
- Mémorisation : pour envoyer un message à un objet serveur, le client doit connaître l'objet serveur (ou son identité si le modèle dispose de cette notion). L'objet serveur peut être mémorisé soit au sein de l'objet client soit dans l'environnement (le système à objets). Il peut aussi être passé en paramètre d'une méthode.
- Multiplicité (cardinalité) : une instance d'une classe peut faire appel à une ou plusieurs instances différentes d'une autre classe. Ce critère a une influence sur les critères précédents. Par exemple, un même film peut être enregistré sur plusieurs cassettes.

Nerson distingue cinq types de relation d'utilisation [Ner92], issues des cardinalités du modèle à objets et de la dualité association/agrégation : du client vers le serveur (*utilise, a besoin de, a, consiste en*) et du serveur vers le client (*fournit*). Notons aussi qu'une distinction est parfois faite dans certaines méthodes entre les attributs typés par un type de base et les attributs typés par une classe.

En programmation à objets, la relation d'utilisation restreinte aux attributs de la structure est appelée *dépendance structurelle* (on suppose ici que le type de l'attribut est connu). La structure de l'objet (ses attributs) sert à mémoriser les valeurs simples (dont le type est un type de base), les objets composés et les objets serveurs. En C++, par exemple, une donnée membre (un attribut) a pour type un type C (type de base), une classe (objet composé) ou un pointeur vers une classe (objet serveur). Si la multiplicité est supérieure à 1, on utilise des collections d'objets ou des collections de pointeurs vers les objets. En analyse et conception à objets, les valeurs simples sont des attributs, les objets composants sont reliés par une relation d'agrégation et les objets serveurs sont reliés par une association.

La remarque précédente montre que l'implantation d'une modélisation à objets implique des choix de représentation dans le langage cible. En particulier, si la relation de clientèle est symétrique ou bidirectionnelle, plusieurs implantations sont possibles, qui nécessitent des pointeurs. Pour les mêmes raisons, définir quel est le receveur d'une méthode quelconque est un problème épineux. Par exemple, `emprunter(cassette, boutique, adhérent)` est-elle

une méthode de la classe adhérent, de la classe boutique ou de la classe cassette ou des trois ? Le critère de regroupement n'est pas uniquement la structure mais aussi le comportement.

La relation de **dépendance** est un cas particulier d'utilisation qui exprime que pour définir une classe on fait référence à d'autres classes ou à des interfaces.

Relation d'héritage L'innovation la plus importante du modèle à classes est l'**héritage** (relation de généralisation/spécialisation, sous-classe, sous-typage, raffinement). Voici une définition extraite de [Mey97].

Définition 1.1 (héritage) *L'**héritage** est un mécanisme permettant de définir une nouvelle classe (la sous-classe) à partir d'une classe existante (la super-classe) par extension ou restriction.*

L'extension se fait en rajoutant des méthodes dans le comportement ou des attributs dans la structure. Par exemple, les conditions d'emprunts des adhérents peuvent être différentes s'il s'agit d'employés du magasin ou d'autres personnes. La classe employé du magasin est une extension de la classe adhérent à laquelle on ajoute un compte boutique. La restriction consiste à réduire l'espace des valeurs définies par la classe en posant des contraintes (conditions logiques à vérifier) sur ces valeurs. Par exemple, supposons un attribut âge dans la classe adhérent et la règle suivante : les jeunes adhérents peuvent emprunter deux exemplaires de plus que les autres adhérents. La classe jeune adhérent est une restriction de la classe adhérent dont la contrainte porte sur l'âge. La restriction correspond plutôt au sous-typage, qui est souvent défini comme une inclusion ensembliste des valeurs de deux types. On retrouve donc le double aspect "classe = module" et "classe = type de donnée".

L'héritage induit le **polymorphisme** : une instance de la sous-classe est aussi une instance de la super-classe. Un jeune adhérent reste un adhérent. Cette instance peut donc utiliser sans les définir les méthodes de la super-classe.

L'héritage est à la fois un mécanisme d'inférence permettant de définir des hiérarchies de généralisation/spécialisation (taxonomie) et un mécanisme de construction incrémentale de classes par réutilisation de code. Ou bien, dit autrement, une sous-classe représente soit un sous-type soit une autre implantation du même type (soit les deux bien sûr). Certains modèles autorisent l'héritage multiple.

Définition 1.2 (généricité) *La **généricité** est un mécanisme permettant de paramétrer des types par d'autres types. L'actualisation (ou instanciation) de paramètres formels de types se fait en fournissant des types définis (des classes ou des types de base). Une classe générique acceptera donc en paramètre d'autres classes (et sous-classes).*

L'exemple classique est celui des collections d'éléments (ensemble d'entiers, de personnes...). Le type des éléments n'influe pas sur les opérations de la collection. Combiné avec l'héritage, la généricité offre un support puissant à la réutilisation.

> L'héritage et la généricité forment les deux cas de vrai polymorphisme (le même code est utilisé quels que soient les types). La coercition (cast, transtypage) et la surcharge (y compris les redéfinitions) sont des cas de faux polymorphisme puisqu'un code différent va être utilisé. Evidemment seul le vrai polymorphisme apporte la réutilisabilité du code.

Notation UML des relations

En UML, les relations s'appliquent aux éléments de modélisation. De ce fait, on peut dire que leur sémantique est surchargée. Toutes ces relations sont *binaires*. Elles peuvent être affinées par stéréotypage.