Développement du modèle statique

Objectifs du chapitre

Ce chapitre va nous permettre d'illustrer les principales constructions du diagramme de classes UML durant l'étape d'analyse.

Le diagramme de classes a toujours été le diagramme le plus important dans toutes les méthodes orientées objet. C'est également celui qui contient la plus grande gamme de notations et de variantes. UML a réussi à unifier le vocabulaire et les concepts sans perdre la richesse et les apports des différentes méthodes existantes.

Quand intervient le développement du modèle statique ?

Le développement du modèle statique constitue la deuxième activité de l'étape d'analyse. Elle se situe sur la branche gauche du cycle en Y et succède au découpage en catégories. Les diagrammes de classes établis sommairement dans les DCP (diagrammes des classes participantes du chapitre 4), puis réorganisés lors du découpage en catégories (chapitre 6), vont être détaillés, complétés, et optimisés.

Il s'agit d'une activité itérative, fortement couplée avec la modélisation dynamique, décrite au chapitre suivant. Pour les besoins du livre, nous avons présenté ces deux activités de façon séquentielle, mais dans la réalité elles sont effectuées quasiment en parallèle.

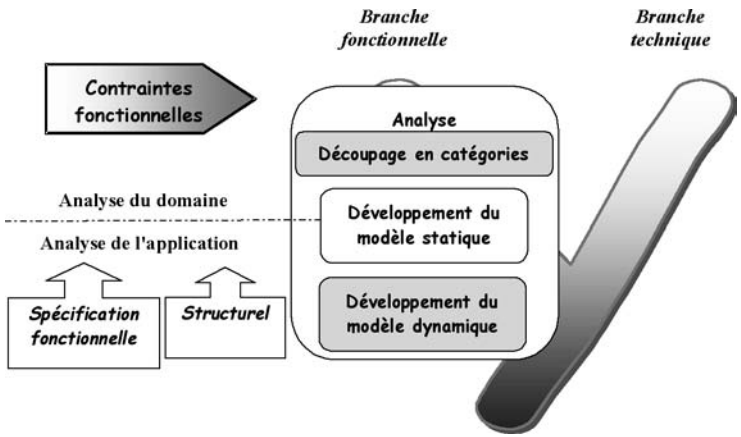

Figure 7-1 : Situation du développement du modèle statique dans le 2TUP

Éléments mis en jeu

- Classe, responsabilité,
- Association, multiplicité, agrégation, composition,
- Attribut, attribut dérivé, attribut de classe,
- Classe d'association, qualificatif,
- Opération, opération de classe,
- Classification, généralisation, spécialisation,
- Classe abstraite, principe de substitution, généralisation multiple,
- Contrainte.

Affiner les classes

Les classes identifiées lors de l'étude des cas d'utilisation (chapitre 4), puis réparties dans les catégories (chapitre 6), sont simplement des classes candidates pour l'analyse objet. Il convient désormais de les examiner de manière détaillée, d'en éliminer certaines, ou au contraire d'en ajouter d'autres. Cette activité de validation est itérative ; l'affinement des associations, ainsi que l'ajout des attributs et des opérations, vont nous fournir de précieuses informations.

On peut cependant dès à présent répertorier quelques principes généraux pour éliminer les classes qui n'en sont pas :

- classes redondantes : elles représentent le même concept ;

Exemple SIVEx : si plusieurs analystes avaient travaillé en parallèle, on aurait pu trouver les classes redondantes *Vehicule* et *Camion*, *EnCoursDeCmd* et *SuiviCommande*, *IncidentMission* et *Alarme*, etc.

- classes vagues : elles ne correspondent pas à des concepts que l'on peut exprimer par des classes. Ainsi, des termes tels que « organisation des réseaux » ou « configuration des étapes » sont trop généraux et ne sont pas suffisamment précis pour justifier la création d'une classe ;
- classes à la place d'attribut : elles expriment des concepts quantifiables ;

  Exemple SIVEx : pas de classe *Poids*, ce n'est qu'un attribut de *Colis*.
- classes à la place d'un rôle : elles expriment un rôle dans une association particulière ;

  Exemple SIVEx : dans la catégorie *Ressources*, la notion d'agence principale est représentée par un rôle d'association entre les classes *ZoneRedistribution* et *Agence*, et non par une classe à part entière.
- classes représentant des acteurs : souvent utiles, mais uniquement lorsque le système gère des informations sur l'acteur ;

  Exemple SIVEx : dans la catégorie *Ressources*, on ne trouve pas la classe *Repartiteur*, contrairement aux autres acteurs. En effet, elle semble inutile, alors qu'*OperateurQuai* est utilisée dans *Colis*, *Receptionniste* dans *Commandes*, et *Chauffeur* dans *Missions*.
- classes de conception : elles introduisent trop tôt des choix de réalisation. Ainsi, le concept « fichier client » n'a pas de sens dans le métier, bien qu'intégré au jargon des utilisateurs ;
- classes représentant des groupes d'objets : elles sont inutiles, car implicites dans les multiplicités des associations, et font souvent référence à des choix de conception.

  Exemple SIVEx : on ne trouve pas dans le modèle d'analyse de classe *ListeCommandes* ou *ListeColis*. Ces groupes d'objets sont implicites dans les associations entre *Mission* et *Commande*, et entre *Commande* et *Colis*.

De la même façon, on peut indiquer quelques principes afin d'optimiser le modèle structurel. Cela conduit tantôt à ajouter des classes manquantes, tantôt à subdiviser une classe existante en plusieurs :

- une classe ne doit pas avoir trop de responsabilités, il en résultera sinon un nombre élevé d'associations, d'attributs ou d'opérations. Il est donc préférable de la découper en ensembles plus petits et homogènes en termes de responsabilités et de cycles de vie ;
- ne pas confondre objet physique et objet logique : autrement dit une entité et sa description.

Ces deux principes sont illustrés dans l'étude de cas ci-après.

ÉTUDE DE CAS : SÉPARATION DE RESPONSABILITÉS

Dans la catégorie *Commandes*, nous avons isolé une classe *EnCoursDeCmd* à laquelle la classe *Commande* délègue le suivi réel des dates d'enlèvement et de livraison, ainsi que des dates de départ et d'arrivée dans les agences. Nous sommes amenés de la même façon à séparer le *SuiviMission* de la *Mission*, pour traiter tous les aspects liés au suivi en temps réel des événements.

Nous avons identifié au chapitre 4 (voir la figure 4-16) l'association décrit entre *Commande* et *Colis*. En fait, une commande contient initialement la description de chacun des colis prévus, puis au cours d'une mission, chaque colis réel est identifié et rattaché à une commande. Il faut donc distinguer deux classes différentes : *DescriptionColis* et *Colis*, qui ne comportent ni les mêmes responsabilités, ni les mêmes états.

Notez également que les multiplicités des associations ont été affinées, pour que les cas dégradés, tels que la possibilité d'égarer des colis, soient pris en compte.

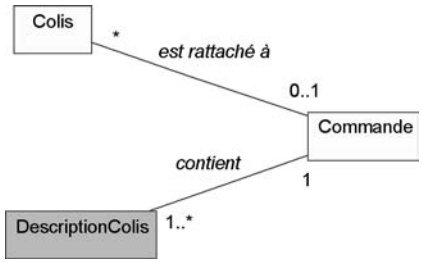

Figure 7-2 : Modification de la relation entre Commande et Colis

Affiner les associations

À l'instar des classes, les associations identifiées jusqu'à présent ne forment qu'une ébauche de structure statique. Il convient à présent de les valider, les préciser, en éliminer, et en ajouter. Là encore, il s'agit d'une activité itérative, qui sera complétée grâce à l'identification des attributs.

N'oublions pas qu'en analyse, les associations représentent des relations conceptuelles entre les classes. On peut également dire qu'elles impliquent des responsabilités en termes de *navigation*. La navigation dans un modèle statique représente la capacité à obtenir des informations en parcourant les associations entre les classes. L'exemple de la figure 7-2 indique que l'on peut demander à une *Commande* quelles sont les *DescriptionColis* qui lui sont rattachées et réciproquement à toute *DescriptionColis* à quelle *Commande* elle est rattachée. On peut donc considérer les associations comme porteuses d'une partie fondamentale de la structure statique des classes, en ce sens qu'il est de la nature d'une *Commande* d'être reliée à des *DescriptionColis*.

En revanche, ces responsabilités ne préjugent pas de la structure des classes en conception. C'est en effet durant l'étape de conception détaillée (voir chapitre 11), qu'on effectue les choix d'implémentation du modèle structurel, avec des objectifs d'optimisation et de modularité.

Voici deux principes généraux qui permettent d'éliminer les associations incorrectes ou inutiles :

- associations non structurelles : elles expriment des relations dynamiques, c'est-à-dire des liens instantanés entre objets. Les liens structurels se caractérisent par une certaine durée et une certaine stabilité ;
- associations redondantes : elles peuvent être retrouvées par navigation grâce aux associations existantes.

ÉTUDE DE CAS : ASSOCIATIONS À ÉLIMINER

Il n'incombe pas à une *Commande* de savoir quel *Repartiteur* est en train de la sélectionner pour l'affecter à une *Mission*. Il s'agit d'une relation purement dynamique entre un acteur *Repartiteur* et un objet *Commande*. L'association ne doit donc pas figurer dans le modèle statique, mais être exprimée par une construction dynamique, comme l'envoi de messages.

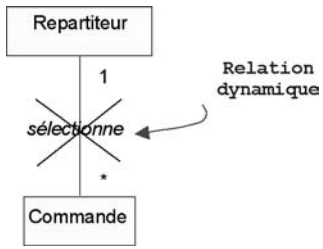

Figure 7-3 : Association erronée entre Commande et Repartiteur

Un *EnCoursDeCmd* concerne une *Commande*, une *Commande* concerne un *Client*. Il serait inutile d'ajouter une association entre *EnCoursDeCmd* et *Client* pour préciser qu'un *EnCoursDeCmd* concerne un et un seul *Client*. En effet, implicitement, un parcours enchaînant plusieurs associations combine les multiplicités successives, en multipliant respectivement entre elles les bornes minimales et les bornes maximales. En voici un exemple :

- de *EnCoursDeCmd* à *Client* en passant par *Commande*, la multiplicité s'obtient par : (1..1) x (1..1) = 1..1 ;
- dans l'autre sens, de *Client* à *EnCoursDeCmd* en passant par *Commande*, la multiplicité s'obtient par : (0..*) x (0..1) = 0..*.

L'association *concerne* est donc totalement redondante avec les deux autres associations et peut être supprimée sans conséquence.

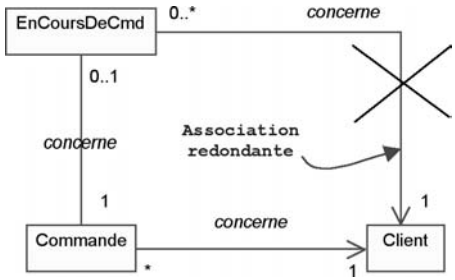

Figure 7-4 : Association redondante entre EnCoursDeCmd et Client

INFLUENCES DES MULTIPLICITÉS SUR LES ASSOCIATIONS

La multiplicité exprimée sur les associations doit être vraie à tous les moments du cycle de vie des instances. Elle induit donc des contraintes sur le processus d'instanciation des classes. Dans l'exemple de la figure 7-4, on ne peut pas instancier *d'EnCoursDeCmd* sans lui assigner une instance de *Commande*. En revanche, pour instancier une *Commande*, il n'est évidemment pas obligatoire de lui associer dès le départ un *EnCoursDeCmd*... La différence entre « 0..1 » et « 1 » est par conséquent plus forte qu'il n'y paraît.

De même, la sémantique des classes associées et de l'association elle-même peut subtilement influencer les multiplicités. Une illustration en est donnée par les trois exemples suivants, tous corrects[1], mais ayant des significations différentes.

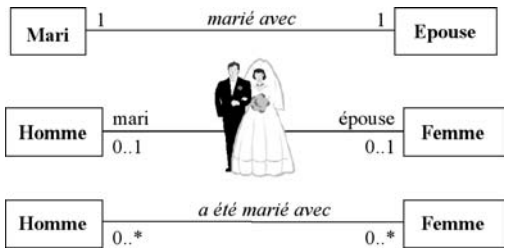

Figure 7-5 : Exemples de multiplicités différentes pour des associations voisines

Un *Mari* est forcément marié à une et une seule *Epouse*, dans le contexte de la loi française actuelle. En revanche, si les concepts plus généraux d'*Homme* et de *Femme* nous intéressent indépendamment des liens du mariage, l'association devient optionnelle ; *Mari* et *Epouse* deviennent alors des rôles. Vous remarquerez au passage combien la différence entre classe et rôle est ténue, dans la mesure où elle dépend fortement du contexte du problème. Si enfin, on veut conserver l'historique des liens de mariage, la multiplicité devient non bornée.

1.Attention, précisons que ces diagrammes sont corrects dans le contexte de la loi française en ce début d'année 2007 ! Ils excluent en effet la polygamie ainsi que le mariage homosexuel…

Affiner les associations consiste également à utiliser deux notions complémentaires fournies par UML : l'agrégation et la composition.

Une association entre deux classes représente par défaut une relation structurelle symétrique entre entités de même importance. Mais UML fournit deux notions qui permettent d'affiner la définition conceptuelle d'une association. Il s'agit de l'agrégation et de la composition qui ajoutent à l'association le sens d'une relation d'éléments à ensemble.

Si l'une des classes joue le rôle d'ensemble composé d'instances de l'autre classe, utilisez l'agrégation. Une agrégation n'est plus sémantiquement symétrique, puisqu'elle privilégie l'une des deux classes en l'élevant au rang de conteneur. L'agrégation garde cependant les propriétés d'une association et n'influe ni sur l'expression des multiplicités, ni sur la navigabilité, ni sur le cycle de vie des instances reliées. Par conséquent, il est possible de partager l'agrégation : une partie peut appartenir simultanément à plusieurs agrégats.

La composition est en revanche une variante d'agrégation qui influe sur la structure des instances qu'elle relie. Avec une composition, nous introduisons ainsi les deux caractéristiques suivantes :

- la composition n'est pas partageable : un objet ne peut appartenir qu'à un seul composite à la fois ;
- le cycle de vie des parties est fortement lié à celui du composite : la destruction du composite entraîne en particulier la destruction de ses parties.

ÉTUDE DE CAS : AGRÉGATION ET COMPOSITION

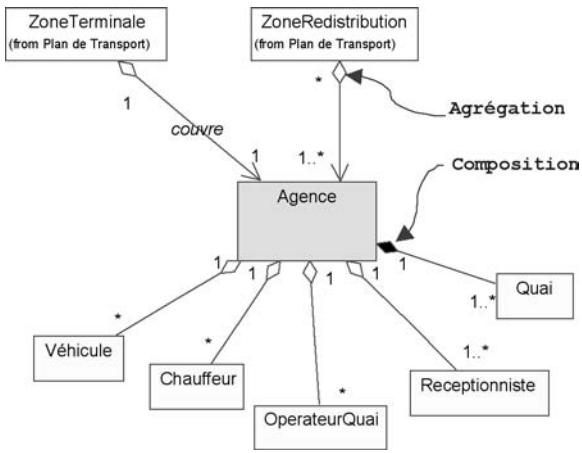

Figure 7-6 : Exemples d'agrégation et de composition autour de la classe Agence

Dans cet exemple, toutes les associations ont la sémantique de l'agrégation. Mais vérifient-elles en plus les critères de la composition ? Le premier critère (non partageable) est bien vérifié par toutes les agrégations sauf celle entre *ZoneRedistribution* et *Agence*. En revanche, seule l'agrégation entre *Agence* et *Quai* correspond au critère d'imbrication du cycle de vie entre composite et composants. Par exemple, la suppression d'une *Agence* n'entraîne pas celle de ses véhicules et chauffeurs, qui vont être réaffectés à une autre agence.

Si l'on reprend l'exemple de la figure 7-2, on peut également distinguer une composition et une agrégation, en fonction du critère qui consiste à lier les cycles de vie des parties à leur ensemble : la suppression d'une commande implique celle de ses descriptions de colis.

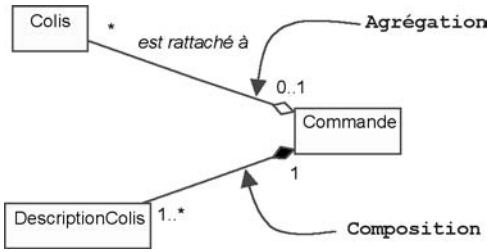

Figure 7-7 : Exemples d'agrégation et de composition dans la catégorie Commandes

Affiner les associations, c'est aussi identifier leurs règles de gestion. UML propose un certain nombre de *propriétés* standard applicables aux associations :

- on peut spécifier que les objets à une extrémité de l'association doivent être ordonnés avec la propriété `{ordered}`;
- on peut également préciser qu'un lien ne peut plus être modifié ni détruit avec la propriété `{frozen}`. Dans le même esprit, la propriété `{addOnly}` signifie que de nouveaux liens peuvent être ajoutés depuis un objet de l'autre côté de l'association, mais non supprimés[1].

Il est à noter que la propriété `{ordered}` n'induit pas la façon dont les objets vont être ordonnés : numéro, ordre alphabétique, etc. Il s'agit en général d'un choix de conception.

1. Même si les contraintes prédéfinies {frozen} et {addOnly} semblent avoir disparu du standard UML 2, nous continuerons à les utiliser pour leur valeur ajoutée.

ÉTUDE DE CAS : PROPRIÉTÉS DES ASSOCIATIONS DES CATÉGORIES MISSIONS ET COMPTABILITÉ

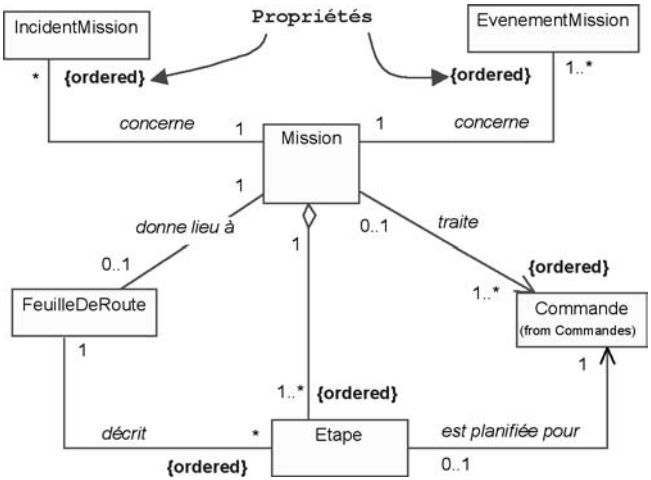

Figure 7-8 : Associations ordonnées dans la catégorie Missions

Les associations entre *IncidentMission*, *EvenementMission* et *Mission* ne sont pas seulement ordonnées. En effet, les liens ne pouvant plus être modifiés ni détruits du côté *Mission*, ils s'ajoutent obligatoirement de l'autre côté.

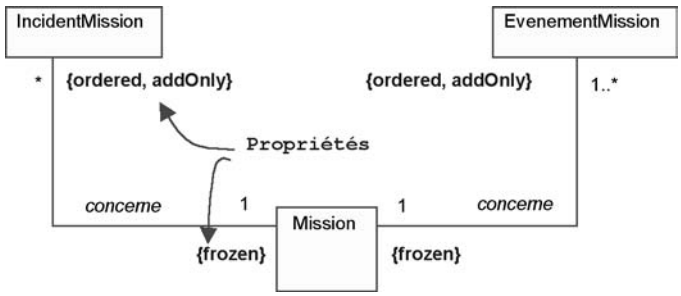

Figure 7-9 : Autres propriétés des associations de la catégorie Missions

Dans la catégorie *Comptabilité*, une réflexion plus poussée sur les multiplicités, les propriétés et la navigabilité permet d'éliminer une association redondante et supprime de la sorte la dépendance entre les catégories *Comptabilité* et *Clients*.

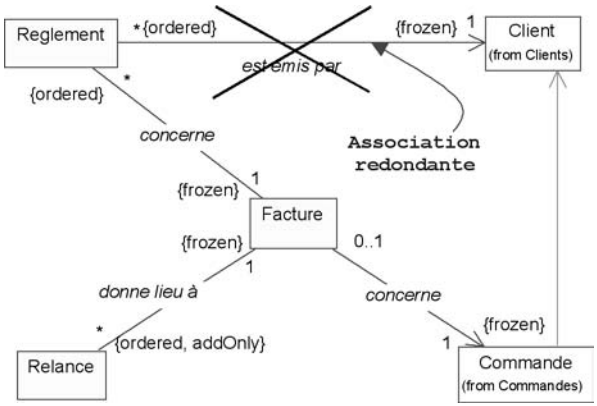

Figure 7-10 : Propriétés des associations de la catégorie Comptabilité

Ajouter les attributs

Un *attribut* est une propriété nommée d'une classe qui décrit un domaine de valeurs possibles partagé par tous les objets de la classe. À tout instant, chaque objet d'une classe porte une valeur spécifique pour chaque attribut de sa classe. Dans un modèle d'analyse, vous conserverez uniquement comme attributs les propriétés simples des classes que le système doit mémoriser et utiliser.

Exemples SIVEx :

- un *Vehicule* a un n° d'immatriculation et un kilométrage ;
- un *Chauffeur* et un *Client* ont un nom ;
- une *Commande* a une référence, un coût estimé et un type de service ;
- un *Colis* a un poids.

DIFFÉRENCE ENTRE CLASSE ET ATTRIBUT

En analyse, un concept est une entité utilisée par l'expert du domaine dans le cadre de son métier ou de l'application. Tout concept doit être modélisé par une classe car il est implicitement porteur de propriétés, assume des responsabilités fonctionnelles dans le système et parcourt éventuellement des états différents.

L'attribut n'est qu'une des propriétés d'une classe se caractérisant par une quantité mesurable, comme le poids du *Colis*, ou la possibilité d'être valorisé par une ou plusieurs données, par exemple le nom d'un *Client*.

Notez encore qu'un attribut peut être valorisé par une structure de données ou bien qu'il peut être multiple au sein d'une classe. Contrairement à ce que nous rencontrons souvent dans les projets, ces caractéristiques ne transforment pas pour autant un attribut en classe. Dans l'exemple ci-dessous, l'analyste comprend qu'un *Point* n'est pas manipulé par l'utilisateur et que l'on peut valoriser un *Point* par la donnée de deux coordonnées dans son modèle. Il transforme donc la classe *Point* en attribut multiple dans les différentes classes géométriques. Remarquez comment le diagramme s'en trouve allégé.

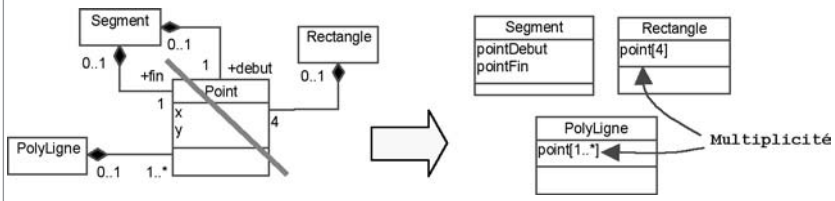

Figure 7-11 : Attribut ou classe ?

Nous allons illustrer la différence entre attribut et classe sur l'étude de cas, ainsi que trois erreurs fréquentes de modélisation liées au concept d'attribut.

ÉTUDE DE CAS : ATTRIBUTS ERRONÉS OU REDONDANTS

Le *Compte* d'un *Client* n'est pas un simple nombre, c'est un concept à part entière comportant lui-même plusieurs attributs ; il peut être bloqué ou débloqué, ce qui implique des états et des responsabilités.

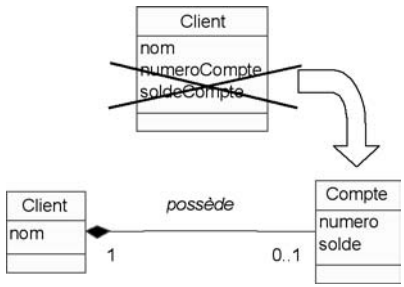

Figure 7-12 : Exemple d'attribut abusif

Un autre défaut fréquent consiste à décrire correctement l'association entre les deux classes, mais à ajouter tout de même un attribut redondant dans l'une des classes. Par exemple, la classe *Commande* n'a pas besoin d'un attribut *nomClient*, puisque chaque *Commande* concerne un *Client* qui a déjà un nom.

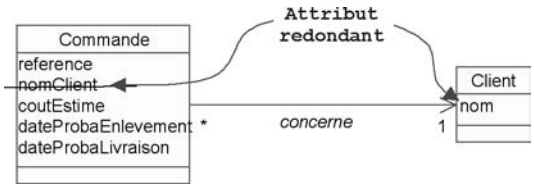

Figure 7-13 : Attribut redondant avec une association

Le cas suivant est un peu plus complexe, car la correspondance n'est pas directe. Cependant, là encore, l'attribut est inutile en analyse, car il peut être déduit de la navigation de l'association. Le nombre de colis d'une *Commande* s'obtient simplement en comptant le nombre d'objets *DescriptionColis* liés à l'objet *Commande* correspondant.

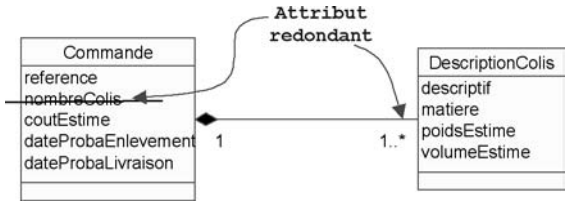

Figure 7-14 : Attribut inutile car implicite d'après l'association

Le cas contraire peut également arriver : une des classes candidates s'avère représenter un « type simple », et non un concept du domaine. C'est le cas dans la catégorie Réseau, où la classe *Commune* est superflue. En effet, elle ne sert qu'à stocker une structure de données représentant la position d'un *Site*. Comme dans l'exemple de la figure 7-11, on peut simplifier le modèle en la transformant en attribut. Notez aussi la multiplicité et la navigabilité de l'association : le fait que *Commune* ne soit pas tenue de connaître ses *Sites* est un argument qui joue en faveur de sa suppression.

Figure 7-15 : Remplacement d'une classe candidate par un attribut

DÉNOMINATION DES ASSOCIATIONS PAR LES RÔLES

La différence subtile entre les concepts d'attribut et de classe se retrouve également dans la notion de *rôle* d'une extrémité d'association.

Étude

Revenons sur les deux façons possibles de nommer les associations. La première consiste à utiliser un *groupe verbal*, afin que l'association puisse être lue comme une phrase. C'est ainsi que procédaient les méthodes traditionnelles de modélisation de données. Cette méthode présente un inconvénient : le sens de lecture du verbe n'est pas forcément le sens naturel (de gauche à droite ou de haut en bas) après plusieurs remaniements de diagramme. Cependant, en analyse, c'est souvent la technique la plus simple à mettre en œuvre.

La deuxième consiste à nommer le rôle que joue chacune des classes dans son association avec l'autre. Ce rôle représente une sorte d'attribut complexe de la classe distante, il est donc décrit par un *substantif*. L'inconvénient de la méthode précédente est ainsi résolu, puisque les noms de rôles restent bien « accrochés » à leur extrémité d'association, même si on les déplace graphiquement. De plus, le substantif a de grandes chances d'être utilisé lors de la génération de code (voir le chapitre 11 sur la conception détaillée).

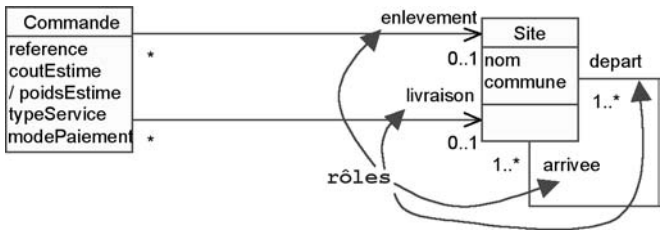

Figure 7-16 : Exemples de rôles d'association

Il est à noter que les noms de rôles sont obligatoires pour les associations qui bouclent sur une classe, comme celle de la classe *Site* sur la figure précédente

Toutefois, il faut rester pragmatique : ne nommez les associations que si le nom apporte une information significative pour le lecteur. Inutile de préciser *est relié à*, *est associé à* ou de recopier le nom de la classe comme nom de rôle ! Certaines associations sont évidentes ; par ailleurs n'oubliez pas que l'agrégation et la composition signifient déjà quelque chose par elles-mêmes.

Conseil

DISTINGUEZ LES ATTRIBUTS DÉRIVÉS !

Un attribut *dérivé* est un attribut intéressant pour l'analyste, mais redondant, car sa valeur peut être déduite d'autres informations disponibles pour la classe concernée. UML permet à la fois de le citer en tant qu'attribut, d'indiquer au lecteur du modèle son caractère redondant grâce au « / » et enfin d'exprimer la contrainte associée.

ÉTUDE DE CAS : ATTRIBUTS DÉRIVÉS

Le cas le plus simple est celui d'un attribut dérivé d'un autre attribut de la même classe. Par exemple, l'âge d'un *Chauffeur* est dérivé de sa date de naissance.

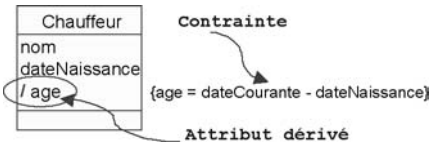

Figure 7-17 : Attribut dérivé et contrainte

En analyse, un attribut dérivé indique seulement une contrainte entre deux propriétés, un invariant, comme le montre l'exemple précédent. Il ne précise pas encore ce qui doit être calculé par rapport à ce qui doit être stocké : ce sera un choix de conception.

Un attribut dérivé peut aussi être déduit de façon plus complexe. Par exemple, le poids estimé d'une *Commande* est égal à la somme des poids estimés de ses descriptions de colis.

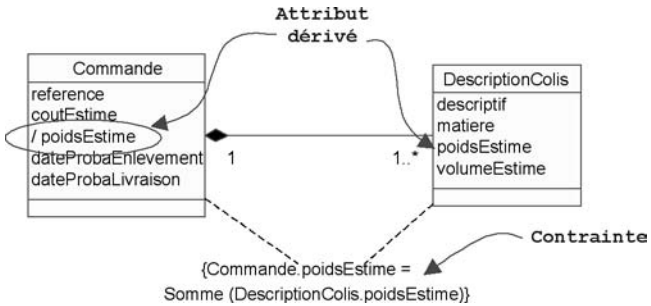

Figure 7-18 : Autre exemple d'attribut dérivé


Conseil

DISTINGUEZ LES ATTRIBUTS DE CLASSE !

Par défaut, un attribut a une portée d'instance : chaque objet de la classe possède sa propre valeur pour la propriété. Dans certains cas plus rares, l'attribut peut avoir une portée de classe : il existe alors une seule valeur commune de la propriété pour toutes les instances de la classe. On parle dans ce cas d'attribut *de classe*[1], et on le souligne pour le distinguer des attributs d'instance.

1.La plupart des outils de modélisation proposent plutôt une case à cocher « static », d'après le mot-clé correspondant en Java, C++ ou C#.

ÉTUDE DE CAS : ATTRIBUT DE CLASSE

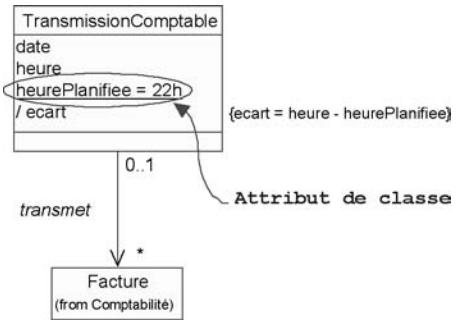

Figure 7-19 : Attribut de classe

Les factures sont transmises en fin de journée au module CO de SAP : la classe *Transmission-Comptable* comporte deux attributs principaux : une date et une heure de transmission effective. On souhaite suivre les écarts de transmission réelle par rapport à l'heure planifiée, soit 22 h tous les soirs. Il suffit pour cela d'ajouter un attribut de classe *heurePlanifiee*, dont la valeur est identique pour tous les objets de la classe, et un attribut dérivé *ecart*.


Ne pas faire

N'UTILISEZ PAS LA NOTATION COMPLÈTE DE L'ATTRIBUT EN ANALYSE !

Dans les diagrammes précédents, nous avons simplement indiqué le nom des attributs. C'est généralement suffisant pour le lecteur d'un modèle d'analyse. Il sait parfaitement ce qu'est une date, un *poidsEstime*, ou un *coutEstime*.

La syntaxe complète d'un attribut en UML est évidemment plus complexe :

```
[visibilité] nom [multiplicité] [: type] [= val_init]
[{propriété}]
```

Les déclarations optionnelles (entre « [ ] ») sont utiles en conception ; certaines sont même nécessaires. En revanche, en analyse, nous vous conseillons de ne les employer qu'avec parcimonie, le risque étant de faire prématurément des choix de conception injustifiés. En analyse, il faut rarement anticiper le type, la valeur initiale, ou la visibilité. Les seules déclarations intéressantes sont les suivantes :

- la multiplicité permet de simplifier certains diagrammes, en exprimant de façon condensée des structures de données, comme nous l'avons vu à la figure 7-11 ;
- la valeur initiale est surtout utile pour les attributs de classe (comme dans l'exemple de la *TransmissionComptable* : heurePlanifiee = 22h) ;

- la propriété `{frozen}` permet d'indiquer un attribut dont la valeur ne peut plus changer une fois que l'objet a été créé.
  La propriété permet d'indiquer un attribut dont la valeur ne peut pas être modifiée par les objets clients.

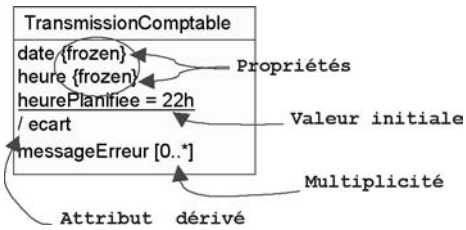

Figure 7-20 : Utilisation avancée de la syntaxe de l'attribut en analyse

Nous venons de voir comment affiner les classes, affiner les associations et ajouter les attributs. Il arrive que l'on ait besoin d'effectuer les trois activités à la fois : ajouter une classe pour décrire des attributs portés par une association.

En effet, une association entre deux classes peut elle-même comporter des attributs. L'exemple type est celui de la relation « employeur/employé » entre *Société* et *Personne*. Où placer les propriétés de salaire, ancienneté, etc., si les multiplicités sont « 0..* » des deux côtés de l'association ? Il faut valoriser ces attributs pour chaque couple d'instances (Société-Personne), et pas simplement pour un objet.

La solution en UML consiste à modéliser cette association comme une classe *Emploi* qui contient les propriétés de l'association. Il existe alors une instance d'*Emploi* pour chaque lien entre objets des deux classes. La classe *Emploi* est appelée *classe d'association* ; il s'agit d'un élément de modélisation UML qui est à la fois une classe et une association. Elle peut donc à son tour établir des relations avec d'autres classes, comme *Emploi* avec *ConventionCollective*.

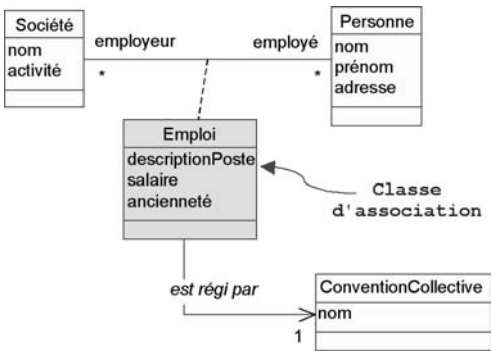

Figure 7-21 : Exemple de classe d'association

La figure suivante présente une instanciation possible du diagramme précédent, dans lequel les objets E1 et E2 portent des valeurs différentes pour certains attributs. Nous ne les avons pas indiqués pour des raisons évidentes de confidentialité :-).

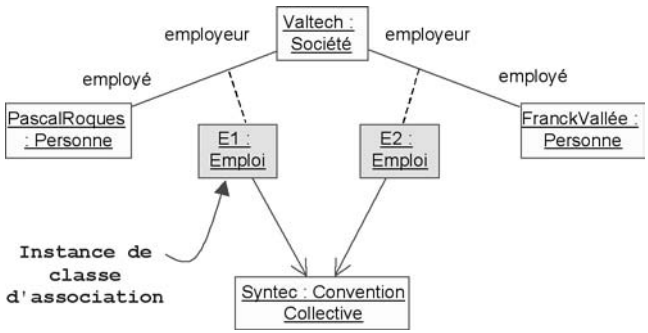

Figure 7-22 : Exemples d'objets d'association

ÉTUDE DE CAS : CLASSE D'ASSOCIATION *PARCOURS*

Dans la catégorie *Réseau*, chaque *Parcours* décrit en réalité une relation entre deux *Sites* (il en va de même pour les *Connexions* entre *Agences*). Le modèle se trouve amélioré si l'on transforme *Parcours* et *Connexion* en classes d'associations.

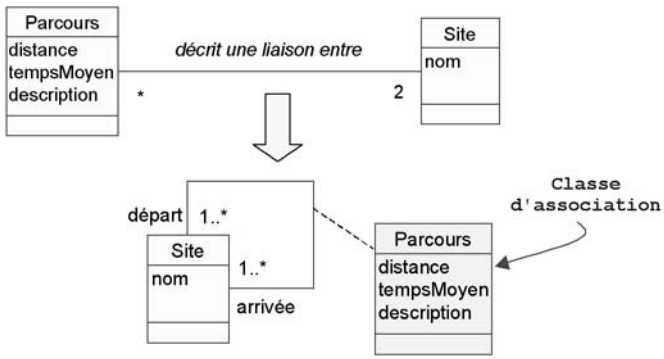

Figure 7-23 : Exemple de classe d'association dans SIVEx

Notez que l'existence d'un objet *Parcours* est obligatoirement subordonnée au lien entre deux objets *Site*. Cette relation de dominance entre *Site* et *Parcours* est plus clairement exprimée par la deuxième solution : celle où *Parcours* devient une classe d'association.

Une association ne peut comprendre des propriétés qu'en se transformant en classe d'association. En conséquence, le nom de la classe d'association est également celui de l'association. Vous pouvez néanmoins nommer également les rôles portés par les extrémités de cette association, comme à la figure 7-23.

Il peut arriver que vous vouliez décrire les mêmes propriétés pour plusieurs associations. Dans ce cas, vous ne pouvez pas réutiliser une classe d'association en l'attachant à d'autres associations, puisqu'elle est l'association elle-même. Une solution peut consister à définir une super-classe (généralement abstraite) dont hériteront les différentes classes d'association, comme illustré à la figure ci-après.

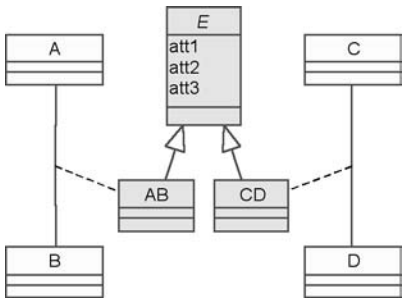

Figure 7-24 : Super-classe d'association

Parfois, un ou plusieurs attributs initialement affectés à une classe ne servent qu'à préciser la désignation d'objets par le biais d'une association.

Définition

QU'EST-CE QU'UN QUALIFICATIF ?

Un *qualificatif*[1] est un attribut d'association dont les valeurs partitionnent la liste des objets reliés par le biais d'une association. En d'autres termes, la connaissance d'un objet et d'une valeur de qualificatif permet de retrouver un ou plusieurs objets liés à l'autre bout de l'association concernée.

Le qualificatif affine donc l'accès à une instance par navigation sur une association. Il ne s'applique évidemment qu'à une multiplicité supérieure à « 1 » puisqu'on ne peut partitionner une liste composée d'un seul élément.

1.Ou « qualifieur » ? Il s'agit en effet d'une traduction du terme anglais qualifier …

La définition du qualificatif n'est pas aisée à saisir sans exemple, nous en développons ci-après différentes utilisations.

Reprenons par exemple la relation « employé–employeur » de la figure 7-21. Où placer l'attribut *matricule* ? À première vue, il s'agit d'une propriété de la classe *Personne*. Mais en fait, une personne possède un matricule pour chacun

de ses employeurs. Il s'agit donc plutôt d'un attribut de l'association, que nous pouvons placer dans la classe *Emploi*. Poussons l'analyse encore un peu plus loin : à quoi sert l'attribut *matricule* sinon à référencer un employé au sein de son employeur ? Il s'agit d'un identifiant relatif, dont la valeur permet d'accéder à une instance particulière de *Personne*, pour une *Société* donnée. C'est exactement la notion de qualificatif en UML. Il se représente graphiquement comme indiqué à la figure ci-après, qui récapitule les étapes de la transformation d'un attribut en attribut d'association, puis en qualificatif, avec la réduction de multiplicité qui s'ensuit.

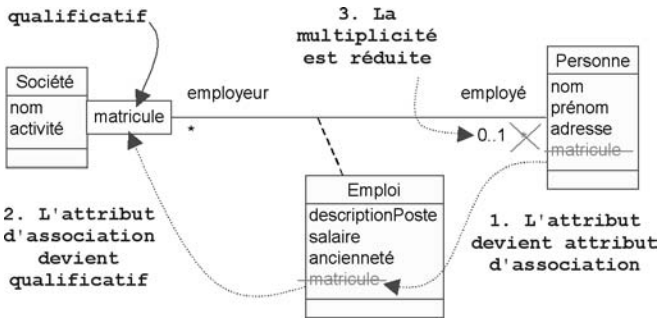

Figure 7-25 : Exemple de qualificatif remplaçant un attribut

Un objet *Société* doté d'une valeur particulière du qualificatif *matricule* a accès à un objet *Personne* au plus, car la multiplicité a été réduite à « 0..1 ». Il peut y avoir des numéros de matricule inutilisés, d'où la limite inférieure à 0.

ÉTUDE DE CAS : QUALIFICATIFS

Une *Agence* contient des *Quais* et chaque *Quai* y est référencé par un numéro. Ce numéro représente un identifiant relatif à une *Agence* particulière, et non un identifiant absolu du *Quai*. Dans un sens, numéro n'est pas seulement un attribut de l'association, puisqu'il permet d'identifier de façon unique un *Quai* par rapport à une *Agence*.

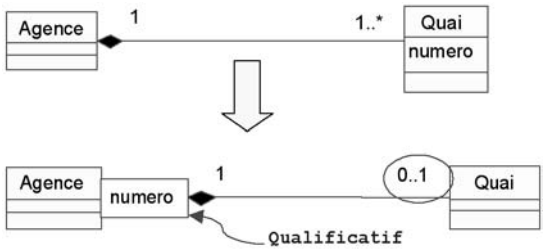

Figure 7-26 : Exemple de qualificatif dans la catégorie Ressources

Notez bien la modification de la multiplicité de l'autre côté du qualificatif. Pour une instance d'Agence et une valeur de numero, il existe au plus un Quai. Autrement dit, dans une agence donnée, deux quais ne peuvent porter le même numéro.

Attention cependant, car même si c'est le cas le plus fréquent, un qualificatif ne réduit pas systématiquement la multiplicité de « 0..* » à « 1 » (ou « 0..1 »). Supposons que chaque *Agence* possède plusieurs parcs de véhicules. Si l'on utilise le numéro de parc comme qualificatif, la multiplicité du côté de la classe *Vehicule* reste « 0..* ».

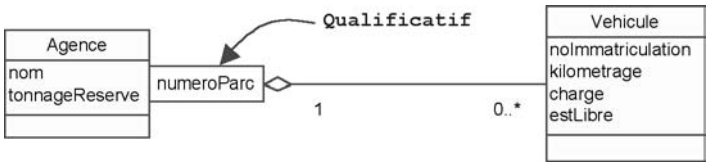

Figure 7-27 : Exemple de qualificatif ne réduisant pas la multiplicité

Ajouter les opérations (optionnel)

Une *opération* représente un service, un traitement qui peut être demandé à n'importe quel objet de la classe. Une opération est l'abstraction de ce que vous pouvez réaliser sur un objet, et elle est partagée par tous les objets de la classe.

Souvent, invoquer une opération sur un objet modifie son état ou les valeurs de certains attributs, ou encore l'existence de certains liens avec d'autres objets.

En analyse, il est possible d'identifier certaines opérations par analyse textuelle du cahier des charges et des fiches de description des cas d'utilisation. Il faut chercher des verbes d'action, comme « envoyer », « valider », les verbes d'état comme « appartient à » ou « concerne », se traduisant plutôt par des associations.

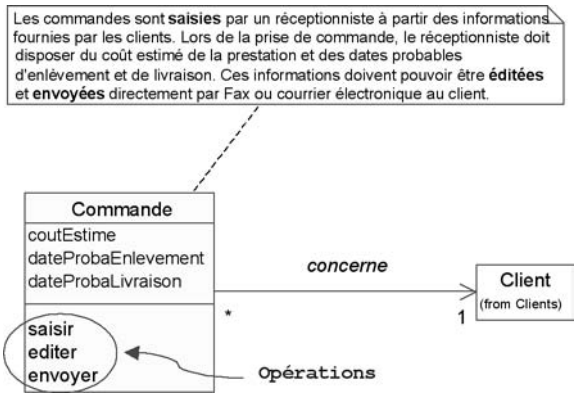

Figure 7-28 : Exemples d'opérations identifiées par analyse textuelle

Soyez toutefois vigilants dans la mesure où toutes les opérations identifiées de cette façon ne sont pas pertinentes. Certains verbes ne traduisent pas des opérations métier, mais plutôt des traitements de l'IHM comme les opérations *saisir* et *editer* de l'exemple précédent, qui ne seront pas conservées. D'autres verbes représentent des opérations implicites en analyse, qui ne seront donc pas indiquées sur le diagramme de classes.

Au prochain chapitre, nous verrons que la meilleure façon d'identifier les opérations consiste à étudier la dynamique de l'application : les interactions entre les objets et les états des classes en fournissent la matière première.


Ne pas faire

NE RÉPERTORIEZ PAS LES OPÉRATIONS IMPLICITES EN ANALYSE !

Pour ne pas surcharger les diagrammes de classes, il est inutile de recenser certaines opérations implicites pour toutes les classes, comme :

- la création et la destruction d'instances : constructeur et destructeur en programmation objet ;
- la manipulation des attributs, à savoir lecture et modification : accesseurs en programmation objet ;
- la création et la destruction de liens, implicites d'après les associations, avec leurs multiplicités et leurs propriétés éventuelles ;
- les parcours et les recherches sur les associations.

Notez que cette recommandation s'applique souvent tout aussi bien en conception, puisque la plupart des outils UML du marché sont capables d'ajouter automatiquement les opérations citées précédemment.

De même, les opérations « non métier » liées en particulier à l'IHM ou au stockage physique seront ajoutées ultérieurement. Il n'est donc pas étonnant qu'une classe d'analyse ait rarement plus de quatre ou cinq opérations. Toutes les autres opérations, implicites ou non métier, viendront s'ajouter en conception[1].


Ne pas faire

N'UTILISEZ PAS LA NOTATION COMPLÈTE

Comme pour les attributs, le nom des opérations est suffisant pour le lecteur d'un modèle d'analyse. Il comprend ce que signifient *envoyer*, *bloquer*, *valider*. Au mieux, il se référera au modèle dynamique pour saisir le contexte dans lequel l'opération est effectuée (voir chapitre 8).

1. Depuis plusieurs années, la tendance consiste à reporter l'identification des opérations dans les classes à l'étape de conception. De nombreux auteurs recommandent soit de ne lister en analyse que des « responsabilités » (et pas de figer l'interface de la classe) soit carrément de ne pas du tout s'occuper des opérations ! Voilà la raison pour laquelle nous avons indiqué en début de chapitre que l'ajout des opérations dans le modèle d'analyse est optionnel.


Ne pas faire

La syntaxe complète d'une opération en UML est évidemment plus complexe :

```
[visibilité] nom [(liste_param)] [: type_retour] [{propriété}]
```

Mais il est clair qu'en analyse, le nom de l'opération et un commentaire textuel suffisent. Toutes les autres informations ne seront utiles qu'en conception.

Comme pour les attributs, une opération a une portée d'instance par défaut : elle s'applique à un objet de la classe. Certaines opérations ont une portée de classe, on parle alors d'opération de classe. Les deux exemples les plus courants d'opérations de classe sont :

- les opérations qui permettent de créer de nouvelles instances ;
- les opérations qui manipulent les attributs de classe.

Or, d'après ce que nous avons dit précédemment, ces types d'opérations sont implicites en analyse. On peut donc en conclure que les opérations de classe sont très rarement utiles en phase d'analyse, en tout cas nettement moins que les attributs de classe.

ÉTUDE DE CAS : OPÉRATIONS DES CLASSES DE LA CATÉGORIE *CLIENTS*

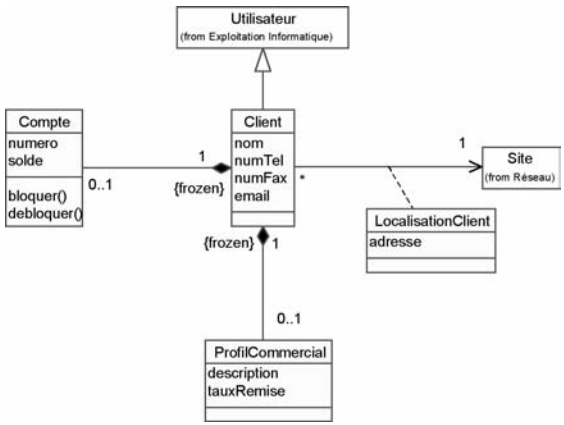

Figure 7-29 : Opérations des classes de la catégorie Clients

Notez les deux seules opérations vraiment « métier » de la classe *Compte*. Toutes les autres seraient des opérations implicites de manipulation d'attributs.

Ces opérations font penser à des changements d'état. Le modélisateur doit donc considérer l'opportunité de réaliser un diagramme d'états pour la classe Compte (voir chapitre suivant).

Optimiser avec la généralisation

À ce stade, il s'agit de découvrir les classes possédant des caractéristiques communes : attributs, associations, opérations. Les propriétés communes seront rassemblées dans une super-classe, et les propriétés spécifiques resteront dans les sous-classes.

ÉTUDE DE CAS : GÉNÉRALISATION DE LA CLASSE *EVENEMENT MISSION*

Reprenons l'exemple du suivi de mission (figure 7-8). Si nous ajoutons les attributs et les opérations et que nous extrayons la classe *SuiviMission* de la classe *Mission* existante (pour bien séparer les responsabilités), nous obtenons le diagramme suivant :

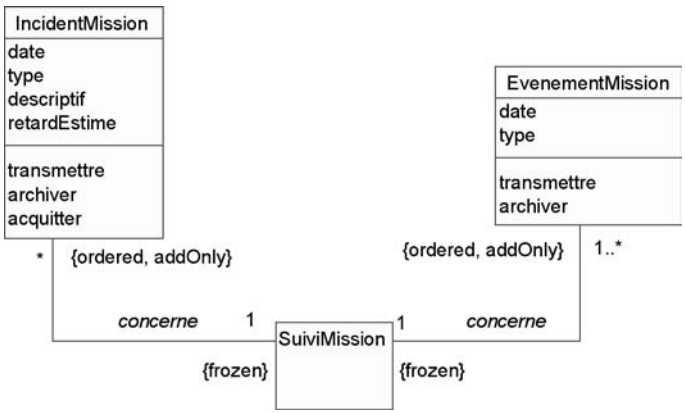

Figure 7-30 : IncidentMission et EvenementMission

Il est clair que nous pouvons simplifier ce schéma en considérant *IncidentMission* comme une sous-classe d'*EvenementMission*. En effet, toutes les propriétés d'*EvenementMission* se retrouvent dans *IncidentMission*, et sémantiquement un *IncidentMission* est bien une sorte d'*EvenementMission*. On obtient alors :

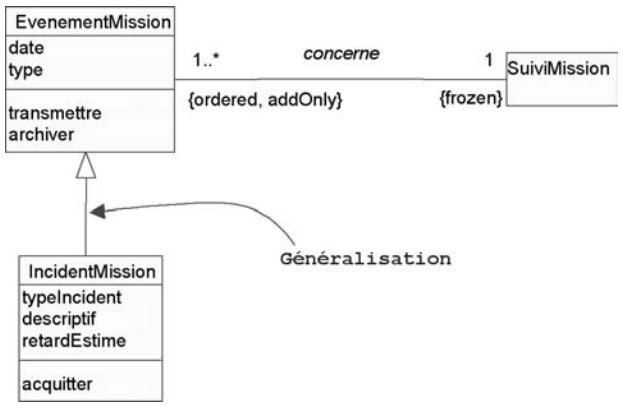

Figure 7-31 : EvenementMission comme généralisation d'IncidentMission

On peut même affiner plus avant en distinguant deux types d'incidents : les incidents de trajet (panne du véhicule, etc.) et les incidents d'étape (client absent, livraison refusée, etc.).

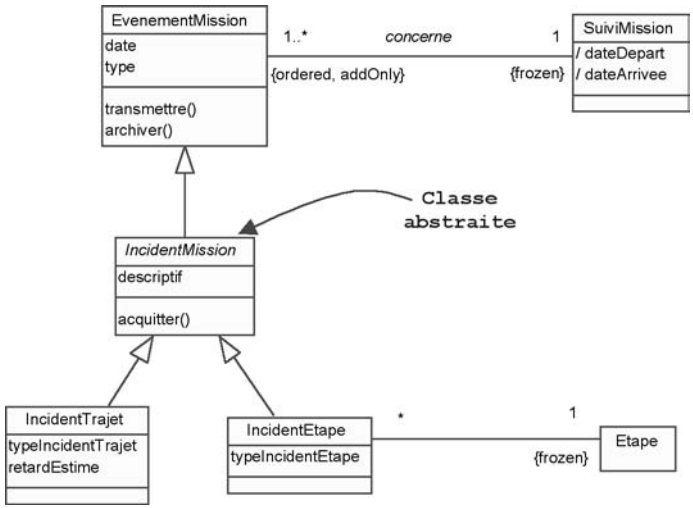

Figure 7-32 : Arbre de généralisation d'EvenementMission

Dans le diagramme précédent, vous remarquez que la classe *IncidentMission* est en italique. En UML, cela signifie que cette classe est abstraite, c'est-à-dire qu'on ne peut pas l'instancier directement. Autrement dit, pour instancier la super-classe abstraite *IncidentMission*, il faut obligatoirement instancier une de ses sous-classes, qui sont pour leur part concrètes.

Une super-classe n'est pas forcément abstraite. Ainsi, la classe *EvenementMission* ne l'est pas, car il peut en exister des instances directes. Sa sous-classe *IncidentMission* est en revanche abstraite, ce qui montre que la super-classe d'une classe abstraite peut être concrète !

Nous avons dit qu'IncidentMission peut être considérée comme une sous-classe d'*EvenementMission*, puisque toutes les propriétés d'*EvenementMission* valent également pour *IncidentMission*, et qu'elle a des propriétés spécifiques supplémentaires.

Une autre façon de l'exprimer consiste à utiliser le principe de substitution de Liskov. Partout où nous parlons d'*EvenementMission* (la super-classe), nous devons pouvoir lui substituer n'importe laquelle de ses sous-classes, sans que cela cause le moindre problème. Si nous écrivons la phrase : « Un *EvenementMission* concerne une et une seule *Mission*, il est daté et peut être archivé », nous pouvons remplacer *EvenementMission* par *IncidentMission*, *IncidentTrajet*, ou *IncidentEtape*, et la phrase doit rester vraie.

Ne pas faire

N'ABUSEZ PAS DE LA GÉNÉRALISATION/SPÉCIALISATION !

On s'aperçoit parfois, après avoir ajouté les attributs et les opérations, que certaines sous-classes ont été distinguées à tort. En effet, elles ont les mêmes propriétés et les mêmes comportements et peuvent être facilement fusionnées.

ÉTUDE DE CAS : AFFINEMENT DE GÉNÉRALISATIONS

Prenons les différentes sous-classes d'*Utilisateur*, que nous retrouvons dans plusieurs catégories. Si *Chauffeur* et *Client* ont bien des attributs et des associations propres, on peut se passer de *Receptionniste* et *OperateurQuai*.

C'est le cas également pour les sous-classes *MissionEnlevement* et *MissionLivraison*, qui ne présentent aucune différence fondamentale. Il est donc souhaitable de les fusionner en une seule classe *MissionDeTournee*, en ajoutant simplement un attribut nature pour pouvoir les distinguer. Nous conservons en revanche l'autre sous-classe *MissionTraction*, car elle possède une association supplémentaire (avec *Agence*), ainsi que des règles de gestion différentes.

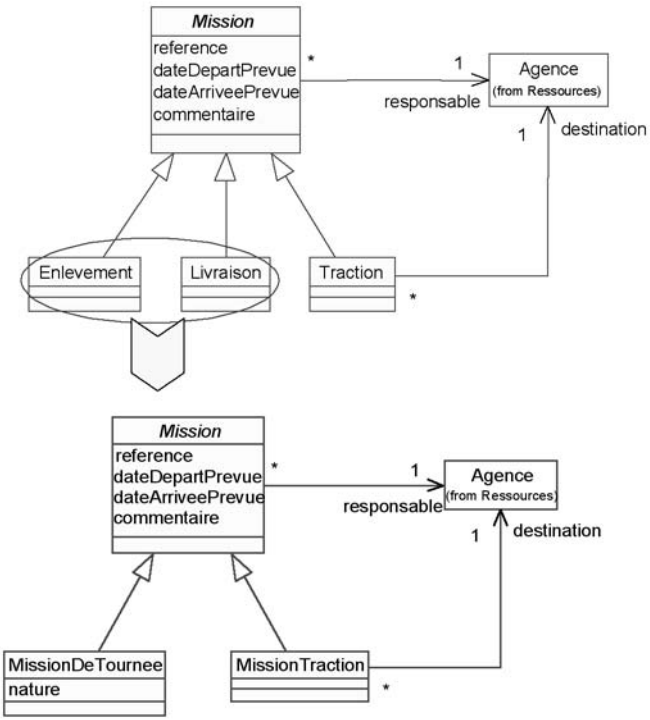

Figure 7-33 : Simplification de l'arbre de généralisation de Mission

Notez que la classe *Mission* est maintenant considérée comme une classe abstraite.

Pour la catégorie *Ressources*, nous avons séparé le modèle statique sur deux diagrammes de classes :

- un diagramme général sur les agences,
- puis le détail concernant les véhicules et les chauffeurs.

Nous n'avons pas encore envisagé le fait qu'une ressource d'une agence peut être provisoirement mise à disposition d'une autre agence. Il en résulte des associations supplémentaires par-

tant d'Agence, et arrivant à *Vehicule* et *Chauffeur*. Pour factoriser toutes ces associations identiques, nous avons introduit une super-classe abstraite *Ressource*.

Notez également la classe d'association *MiseADisposition*. Les dates de mise à disposition des ressources d'une agence à une autre sont typiquement valorisées pour chaque couple (*Agence*, *Ressource*).

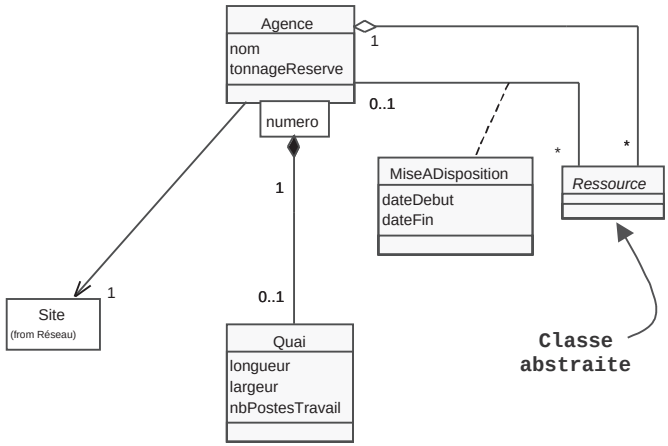

Figure 7-34 : Ressources : diagramme général de l'agence

La figure suivante présente le détail concernant les chauffeurs et les véhicules. On peut remarquer l'héritage multiple de la classe *Chauffeur*. En effet, tout chauffeur est un Utilisateur, mais aussi une *Ressource*. Dès lors que le principe de substitution est respecté, il n'y a pas de raison d'interdire la généralisation multiple en analyse.

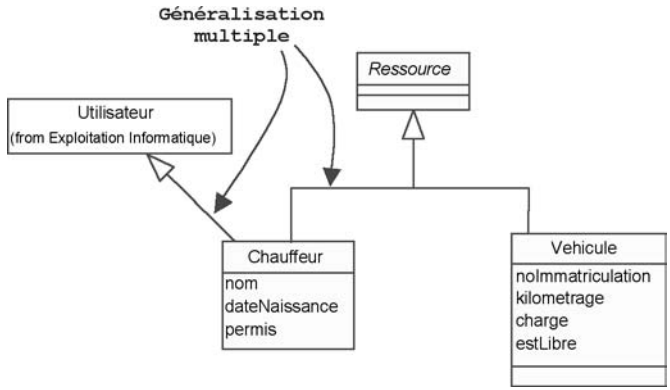

Figure 7-35 : Généralisation multiple de la classe Chauffeur

Encore un petit effort !

Le modèle statique doit rendre compte au final de toutes les règles structurelles. Il sera certes complété par le modèle dynamique, mais doit être exhaustif pour tout ce qui concerne les relations statiques et le contenu des classes.

Il faut également s'assurer de son homogénéité, de sa cohérence, et de sa modularité. À cet effet, il est nécessaire de vérifier si les responsabilités attribuées aux classes sont cohérentes, homogènes et pas trop nombreuses. Il arrive fréquemment que l'on puisse améliorer encore le modèle en introduisant des classes supplémentaires appelées métaclasses, comme nous allons l'illustrer avec l'étude de cas.

ÉTUDE DE CAS : AJOUT DE LA MÉTACLASSE *TYPE VEHICULE*

Reprenons la liste des responsabilités de la classe Vehicule (cf. chapitre 4). Nous pouvons la compléter par la gestion du numéro d'immatriculation et du kilométrage, ainsi que la notion de qualification des chauffeurs suivant le type de véhicule. La traduction de chacune de ces responsabilités, sous forme d'attributs ou d'associations, est détaillée dans le schéma suivant :

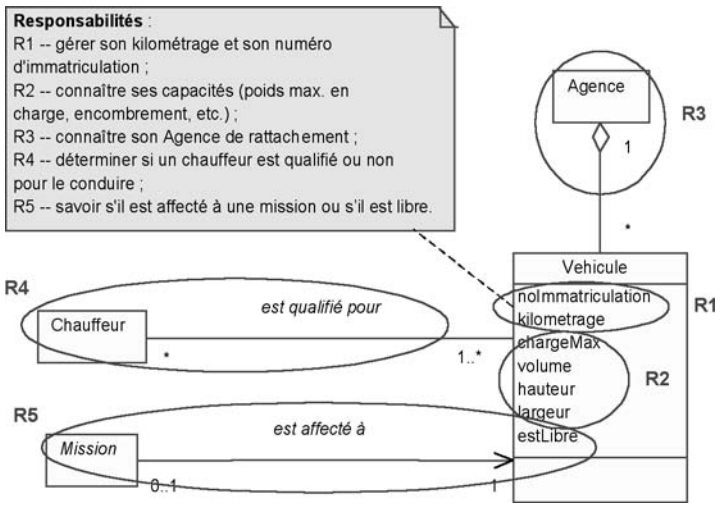

Figure 7-36 : Traduction des responsabilités de la classe Vehicule

Si l'on regarde plus attentivement, on s'aperçoit que les responsabilités R2 et R4 ne dépendent pas de chaque instance de Vehicule. En effet, les capacités d'un véhicule dépendent exclusivement de son type. De même, un chauffeur est qualifié pour un type de véhicule, et non pour un véhicule particulier. La bonne solution de modélisation consiste à isoler cette notion de type de

celle de véhicule, afin de mieux répartir les responsabilités. À cet effet, nous allons créer une nouvelle classe, appelée TypeVehicule, et modifier la répartition des attributs et associations de la façon suivante :

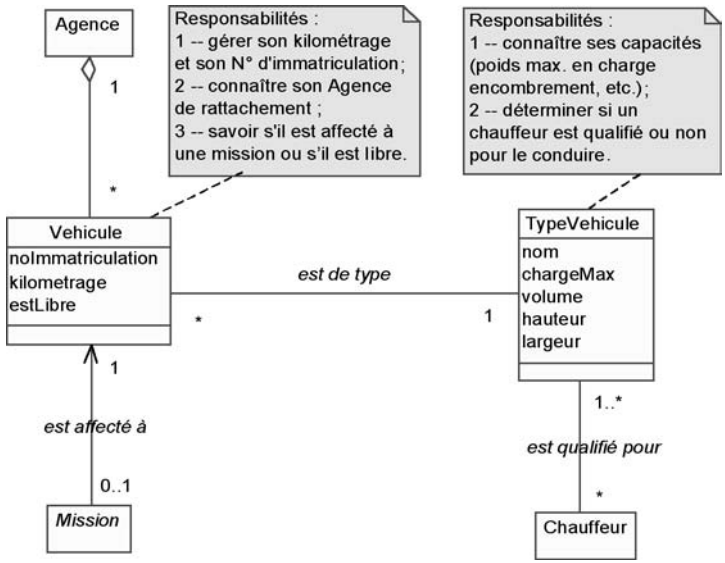

Figure 7-37 : Séparation des responsabilités en ajoutant TypeVehicule

Vous pouvez noter que cette manière de procéder est réutilisable quel que soit le contexte[1]. On identifie une classe *XX* qui possède de nombreuses responsabilités. Certaines ne sont pas propres à chaque instance. On ajoute alors la classe *TypeXX* ou *ModeleXX* et on répartit les attributs et associations sur les deux classes. On termine en reliant *XX* et *TypeXX* par une association « * - 1 », souvent unidirectionnelle vers *TypeXX*. La classe *TypeXX* est qualifiée de métaclasse, car elle contient des informations qui décrivent la classe *XX*.

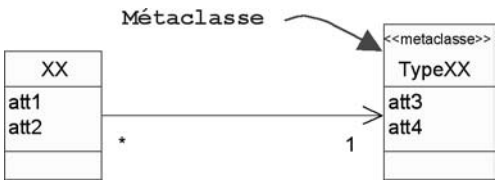

Figure 7-38 : Schéma générique réutilisable (pattern d'analyse).

1. On peut parler dans ce cas de pattern d'analyse (*analysis pattern*).

On nomme classiquement *délégation* la technique consistant pour un objet à déléguer une partie de ses responsabilités à un autre objet lié. Dernier avantage de cette solution : elle permet de lier de nombreuses instances de *XX* à la même instance de *TypeXX*, au lieu de répéter les valeurs communes des attributs *att3* et *att4*.

AJOUTEZ DES CONTRAINTES !

Nous avons précédemment évoqué des propriétés prédéfinies en UML concernant les attributs et les associations. Il est parfois utile d'exprimer des contraintes plus sophistiquées entre plusieurs éléments de modélisation. C'est le cas pour les attributs dérivés ; cela peut également concerner les associations. On peut ainsi indiquer des contraintes d'inclusion ou d'exclusion entre associations, des contraintes de navigation ou des contraintes de dérivation.

Attention cependant : on peut, si on abuse des contraintes, être tenté de combler des lacunes de modélisation par ce biais.

ÉTUDE DE CAS : AJOUT DE CONTRAINTES SUR LES ATTRIBUTS ET LES ASSOCIATIONS

Voici un exemple de contrainte d'inclusion entre associations :

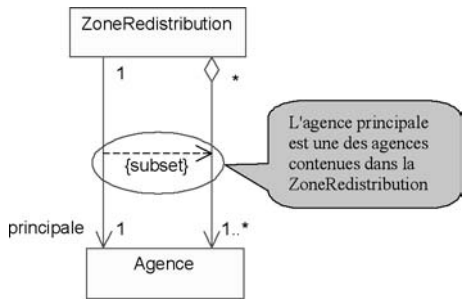

Figure 7-39 : Exemple de contrainte d'inclusion entre associations

Nous avons déjà signalé un certain nombre de contraintes sur les attributs, telles que celles qui expriment la dérivation des attributs dérivés, comme indiqué aux figures 7-17, 7-18, et 7-19.

Le schéma suivant traduit un autre exemple de contrainte entre attributs.

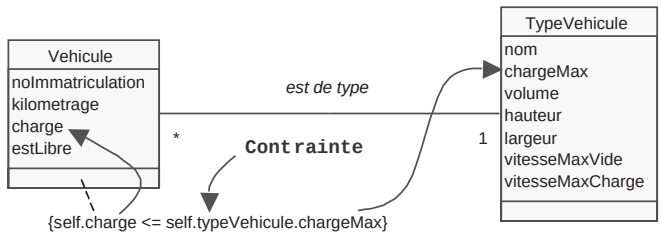

Figure 7-40 : Exemple de contrainte entre attributs

Remarquez la notation pointée qui est utilisée dans la contrainte de la figure précédente. En fait, elle fait partie d'une syntaxe plus étendue appelée OCL et proposée par UML. Le langage OCL (Object Constraint Language) fait en effet partie prenante d'UML et permet d'exprimer des contraintes sous forme d'expressions booléennes qui doivent être vérifiées par le modèle. Mais son utilisation n'est absolument pas imposée, et suivant l'objectif du modèle, vous pouvez exprimer vos contraintes en texte libre, ou plus formellement en OCL.

OCL est un langage à expressions, sans effet de bord sur le système modélisé. Sans entrer dans les détails, l'exemple suivant montre l'utilisation du langage OCL. La contrainte que nous voulons exprimer formellement est la suivante : un colis est dans l'état nominal s'il n'a pas donné lieu à la moindre anomalie.

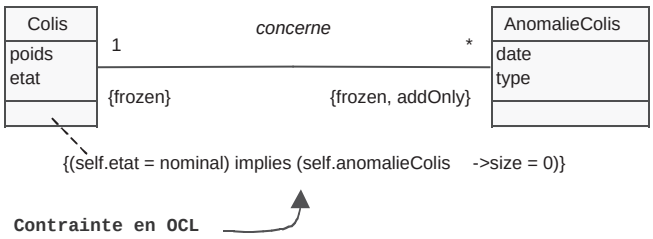

Figure 7-41 : Exemple de contrainte exprimée en OCL

OCL définit plusieurs types de collections d'objets, ainsi que de nombreuses opérations de manipulation de ces collections. Set est un ensemble au sens mathématique. L'expression `self.anomalieColis` retourne l'ensemble des objets AnomalieColis liés à l'objet colis concerné. L'opération prédéfinie size permet de tester si l'ensemble est vide.

Phases de réalisation du modèle statique d'analyse

Le développement du modèle statique constitue la deuxième étape de la phase d'analyse. Les diagrammes de classes préliminaires obtenus lors du découpage en catégories sont détaillés, complétés et optimisés.

Il faut d'abord valider les classes et les associations candidates. Ce travail d'affinement est itératif : il ne faut pas hésiter à modifier, ajouter ou même supprimer des classes et des associations, en se fondant sur la définition de leurs responsabilités. Sur les associations retenues, on précise les multiplicités, les propriétés, les contraintes, et l'on distingue les agrégations et les compositions. On ajoute ensuite les attributs, en distinguant les attributs dérivés et les attributs de classe. Cela conduit fréquemment à identifier des classes d'association, et des qualificatifs. Une première recherche d'opérations peut alors être effectuée ; elle sera complétée lors de l'analyse dynamique. On optimise ensuite les diagrammes de classes en introduisant des super-classes par généralisation, tout en respectant le principe de substitution.

La démarche mise en œuvre dans ce chapitre est synthétisée par la figure suivante :

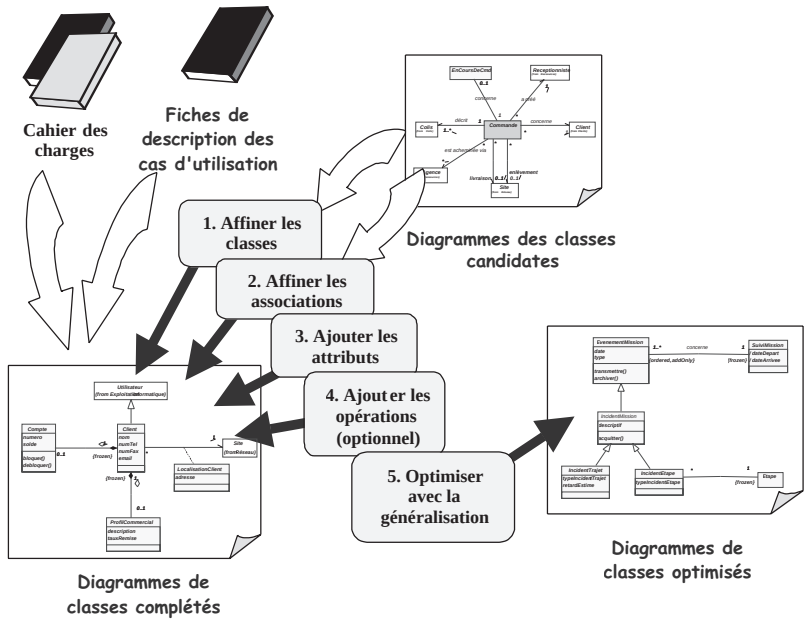

Figure 7-42 : Démarche d'élaboration du modèle statique