Démarche de développement orienté modèles : de la vérification de modèles à l'outillage de la démarche

Leblanc Hervé, Millan Thierry, Ober Ileana

IRIT - Université Paul Sabatier

118, route de Narbonne – 31062 Toulouse Cedex 4

Tel. - Fax. : (33-0)5 61 55 67 82

{leblanc, millan, ober}@irit.fr

Résumé

Ce papier présente l'outillage d'une démarche de développement basée sur les modèles, ayant comme support le standard UML. L'intérêt de cet outillage est de s'assurer que chaque acteur suit le bon déroulement de la démarche à travers un diagnostique établi à différents points de contrôle. Notre approche vise un des aspects oubliés de l'outillage : le contrôle du suivi de la méthodologie elle-même. Pour être outillé, cette démarche doit s'inscrire dans un cadre caractérisé par des propriétés que nous identifierons. Nous présentons dans ce papier la démarche NEPTUNE ainsi que le processus amenant à son outillage. L'instrumentation consiste alors à expliciter le cycle de développement en termes d'activités et de résultats attendus, en termes de modèles, sur lesquelles nous définissons des règles de bonne formation. Nous présentons ensuite des exemples de règles instrumentant les activités d'analyse et de conception ainsi que leur mise en œuvre en OCL. Cette expérience nous a permis d'affiner la démarche NEPTUNE, choisie pour ces travaux, tout en lui définissant une sémantique opératoire.

MOTS-CLES : processus de développement, validation statique de modèles, UML, OCL.

1. Démarche de développement

L'application d'une démarche de développement, tout au long du cycle de vie des logiciels, favorise *à priori* la qualité des résultats attendus en termes de modèles et de codes, et permet à tout instant la maîtrise du cycle de vie: de l'analyse des besoins à la maintenance applicative et ce, en fonction de contraintes liées à l'environnement et au contexte applicatif. Une démarche de développement se caractérise par l'identification et l'enchaînement des principales étapes spécifiant les activités en cours, les acteurs concernés et la validité des résultats intermédiaires. Elle résulte de multiples retours d'expériences et d'un savoir-faire intégrant les différents points de vue, la répartition des responsabilités ainsi que l'organisation des activités de plus en plus nombreuses nécessaires à la mise en production des logiciels.

Dans les années 1990-2000, un grand nombre de démarches de développement à objets ont

émergé tel que HOOD [20] et Fusion [5]. Avec la complexité croissante des systèmes à réaliser, est apparue dans l'approche objet, la nécessité d'augmenter le niveau d'abstraction à l'aide de modèles. La notation standardisée UML a de fait unifié tous les points de vue sur le développement à objets, mais a aussi introduit des aspects essentiels manquant au « tout objet », tels les cas d'utilisation pour l'ingénierie des besoins, les diagrammes d'activités pour les flux de documents et le *business process*. De fait, de nouvelles démarches s'appuyant sur la richesse de la notation composent la « nouvelle vague » des démarches de développement orientés modèles : Catalysis [7], UP [8] et ses nombreuses variations basées sur des ateliers de génie logiciel tels que RUP [13] pour Rational Rose.

Cependant, ces démarches ont été souvent accueillies avec scepticisme [25] pour différentes raisons liées pour l'essentiel au manque d'expérience de leurs utilisateurs potentiels jugeant la définition et l'apport des méthodologies trop vagues et insuffisamment explicites. Mal formulé et difficilement transmissible, un tel savoir-faire demeure insuffisamment exploité au regard notamment de la nature itérative et incrémentale des cycles de développement actuels, de plus en plus courts, et qui conduisent à réviser le résultat des phases précédentes.

Il existe plusieurs réponses à la crise que traverse les démarches de développement :

- Soit considérer le code comme le résultat principal d'un projet, et les modèles comme support de communication entre les différents programmeurs concepteurs du groupe de projet (documentation succincte des modèles à jour et documentation permanente du code), comme préconisé par les méthodes agiles tel XP [1].
- Soit considérer les modèles comme des entités de première classe et se diriger vers une programmation basé sur les modèles [2], [12], [23].

Dans le cadre du projet de recherche NEPTUNE[1] [6], appuyés par des retours industriels, entre ces deux visions « extrêmes », nous avons choisi de préciser et d'outiller les démarches de développement existantes afin de les rendre opérationnelles. C'est pourquoi, nous estimons nécessaire qu'une démarche de développement soit précisée d'une manière « pseudo-formelle » à l'aide d'un langage de description uniforme des étapes à suivre en termes d'objectifs, d'interactions, de résultats nécessaires et attendus sous forme de modèles conformes aux préconisations de la démarche et à partir desquels des points de contrôle peuvent être mis en place. Notre approche vise alors un des aspects oubliés de l'outillage : le contrôle du suivi de la démarche elle-même.

Pour que ce contrôle soit efficace et pertinent, des règles de bonne formation doivent viser à la fois le type et le contenu des résultats produits à chaque étape de la démarche. De telles règles affinent et complètent la définition de la démarche tout en offrant un support à son outillage.

Avant la présentation proprement dite de notre travail sur le contrôle et le suivi d'une démarche de développement, nous donnons, à la section 2, le cadre pour son instrumentation. Nous continuons, en présentant à la section 3 la méthodologie que nous avons décidée d'outiller. La section 4 présente des exemples significatifs de règles d'outillage, au niveau de l'analyse, puis au niveau de la conception. La section 5 dresse un bilan de nos travaux et propose des perspectives pour le futur.

2. Outillage d'une démarche

Dans cette section, nous proposons de définir un cadre pour outiller une démarche de développement au travers de trois propriétés :

- **développement à base de modèle :** au cours de la démarche il faut pouvoir identifier des

[1] http://neptune.irit.fr

points de contrôle et des règles de bonne formation par rapport à l'étape en cours. Pour cela il est nécessaire d'axer la démarche *sur la production de modèles* décrivant les différents artéfacts du développement (exigences, architecture du système à modéliser, spécification du comportement, éléments de conception plus proches de la mise en œuvre, scénarios de test, etc.). L'information caractérisant ces artéfacts évolue tout au long du développement et peut se retrouver dans un ou plusieurs *modèles* selon le point de vue considéré.

- **représentation des modèles en tant que données structurées :** de plus, il faut disposer *d'une représentation des modèles en tant que données structurées*. Ceci est nécessaire pour automatiser le traitement et gérer les modèles obtenus pendant le développement. Idéalement, les différentes facettes d'une modélisation doivent pouvoir se décrire dans le cadre d'un même modèle unificateur, afin de réduire les risques d'incohérence liés à l'utilisation séparée de ces facettes.
- **raisonnement introspectif** : enfin, formuler des règles de cohérence pour contrôler les points d'encrage entre les différentes étapes du développement nécessite de pouvoir *raisonner de manière introspective* [19] sur les modèles.

Axée sur l'utilisation d'UML, notre démarche implique l'utilisation de modèles permettant d'exprimer différents types d'information (exigences, modélisation de la structure et du comportement, etc.).

Pour UML, nous disposons d'une représentation de la structure des modèles, conforme au méta-modèle standard. Il est à souligner que dans notre approche, la nature de la relation de *conformité* [3][10] entre le modèle et le méta-modèle n'est pas essentielle, dans la mesure où l'on peut représenter et exploiter l'information relative à un modèle. Cependant, pour des raisons pragmatiques liées aux technologies employées, nous utilisons des modèles qui sont des *instances* de méta-modèle.

Le dernier critère que nous avons identifié, vise le raisonnement introspectif sur des modèles. Même si l'introspection et la réflexivité sont quasi-absentes en UML, le fait même de pouvoir utiliser le méta-modèle du langage et d'accéder aux méta-informations, nous offre l'expressivité nécessaire à la formulation des règles de cohérence. La norme UML inclut le langage fonctionnel OCL basé sur la logique du premier ordre, permettant de définir des contraintes générales sur des modèles, des gardes de transition, des pré et post conditions d'opérations et des invariants de classes. De plus, OCL, permettant la navigation entre les différents artefacts d'une modélisation, dispose en outre de primitives ensemblistes et d'itérateurs sur des collections. Le méta-modèle UML étant décrit par des structures de classes [18], OCL est utilisé comme langage de spécification de contraintes sur le méta-modèle. Ainsi, des règles de bonne formation (*wellformedness rules*) peuvent être adjointes au méta-modèle UML pour décrire la validité des modèles.

Dans ce cadre, une *méthodologie* est la description d'une démarche en explicitant les différentes étapes qui la composent et en spécifiant par un ensemble de règles les résultats attendus, exprimés en général sous la forme de modèles. Nous appelons *instrumentation* le suivi automatisé d'une démarche qui consiste donc à :

a. Expliciter et spécifier chacune des étapes de la démarche envisagée en fournissant ses objectifs et les résultats attendus en termes de modèles.

b. Formuler, en langage naturel, les règles de bonne formation des modèles en sortie de chaque étape ainsi que les règles de cohérence entre les modèles issus des étapes précédentes.

c. Traduire en langage OCL, chaque fois que c'est possible, les règles agissant sur les données du méta-modèle UML en :

    - mettant en correspondance le vocabulaire méthodologique avec les éléments de modélisation du méta-modèle,

- trouvant et délimitant le fragment de méta-modèle enveloppant les éléments de modélisation intervenant dans la règle,
- codant la règle en utilisant les liens de navigation à disposition.

d. Valider l'ensemble des modèles données à l'aide d'un interprète OCL en

- activant un point de contrôle à chaque fin d'étape,
- fournissant un diagnostic à la demande.

. L'ensemble des règles et points de contrôle proposés pour l'outillage d'une démarche méthodologique est une manière de définir formellement sa *sémantique opératoire*.

A noter que l'utilisation du langage OCL pour formuler les règles proposées dans la démarche n'est pas le seul choix possible. Nous aurions pu décrire ces règles en utilisant un autre formalisme. Le choix d'OCL nous a semblé le plus naturel, car OCL est partie intégrante de la norme UML.

En principe, on pourrait instrumenter toute démarche de développement ayant les propriétés mentionnées en début de cette section. Nous faisons référence dans la suite de cette présentation à une démarche précise, que nous détaillons à la section suivante. Cette démarche, ayant comme support UML, satisfait ces trois propriétés.

3. Explicitation et spécification des activités d'analyse et conception de la démarche NEPTUNE

La démarche NEPTUNE est issue d'expertises académiques et industrielles. Cette démarche suit un processus de développement itératif et incrémental.

3.1 Organisation des activités

Comme toute démarche, NEPTUNE préconise que les membres de l'équipe participant à l'analyse, à la conception et la réalisation d'un logiciel suivent un certain nombre de phases, appelées tâches (*workdefinition*) dans la terminologie du méta-modèle standardisé par l'OMG, permettant de décrire tout processus de développement logiciel SPEM [19]. Chaque tâche regroupe un certain nombre d'activités. Les différentes activités peuvent s'enchaîner itérativement à l'intérieur de chacune des tâches. La Figure 1 présente une vision synthétique de la démarche : les disciplines au sens SPEM sont structurées suivant trois « axes » verticaux correspondant au développement, au déploiement et à l'expertise, les tâches sont structurées suivant trois « axes » horizontaux correspondant à l'analyse des besoins, la conception de l'architecture et la conception par objets. La Figure 2 précise, dans la notation SPEM, les différents « *travailleurs* » intervenants dans chacune des activités ainsi que la phase concernée.

3.1.1 Activités de développement (axe central)

L'axe central définit les activités de développement proprement dit, ordonnées chronologiquement. Cet axe correspond aux activités « classiques » d'analyse et de conception préconisées par la démarche RUP.

3.1.2 Activités de déploiement

L'analyse et la conception d'un logiciel devraient normalement aboutir à une solution indépendante des plates-formes d'exécution. Cependant, pour faciliter le déploiement et l'exploitation du logiciel, il est important de tenir compte des contraintes techniques au plus tôt [4].

En effet, l'avènement des architectures 2 tiers, 3 tiers, n tiers et des technologies nombreuses associées, fait que le choix des architectures physiques et logiques du système à étudier influe de manière forte le processus de développement. D'où l'apparition de processus en Y menant conjointement les aspects fonctionnels et techniques, tel qu'on le retrouve dans l'approche MDA et dans la démarche 2TUP[21]. Pour notre part, les activités de déploiement (axe de droite de la Figure 1) spécifient ces choix et contraintes en montrant leurs interactions avec les activités de développement. A titre d'exemple, l'identification des composants logiciels (tâche *conception de l'architecture*) doit tenir compte de leur futur déploiement en termes *de processus et composants physiques* (activité β).

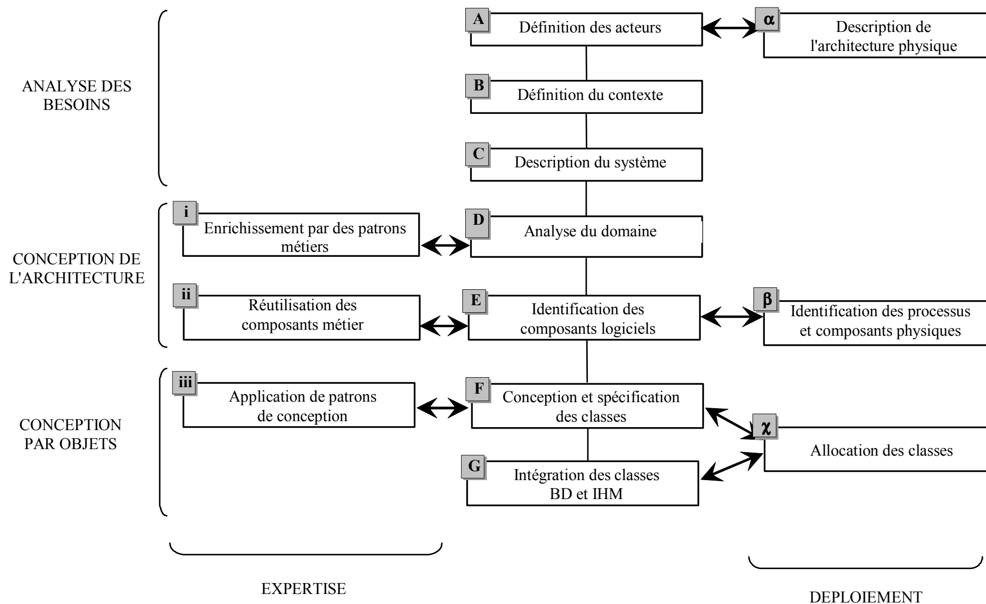

Figure 1 : La démarche NEPTUNE

3.1.3 Activités d'expertise

La réalisation d'une activité du processus pose parfois aux développeurs des problèmes récurrents d'analyse et de conception, dont certains éléments de solution sont déjà décrits de façon générique dans la littérature [11]. Cette connaissance relève d'experts d'un domaine métier ou technologique que l'on sollicite pour leur savoir-faire. Orthogonalement à l'axe centrale et à l'axe de déploiement, s'adjoint la possibilité de prendre en compte, par les activités de l'axe d'expertise, des solutions déjà éprouvées qui devront s'adapter harmonieusement et s'intégrer logiquement dans la démarche méthodologique. Il en est ainsi, par exemple à la Figure 1 de l'enrichissement d'un domaine d'analyse par des patrons métier ou de l'intégration de patrons de conception à un schéma de classes.

3.2 Description des tâches

La démarche NEPTUNE, instance du RUP pour l'analyse et la conception, couvre pour l'essentiel sa phase d'*élaboration*. La Figure 2 présente dans le standard SPEM l'architecture de la démarche qui s'articule autour de trois *tâches* (*workdefinition*) déjà introduites et présentées à la Figure 1.

Ces tâches, qui sont *l'analyse des besoins, la conception de l'architecture* et la *conception par objets,* sont composées de différentes activités et sont réalisées par des *acteurs*.

Les trois autres phases du RUP (*inception, construction* et *transition*) ne sont pas entièrement couvertes par la démarche NEPTUNE. Nous considérons également que la conception d'une

application est un tout et qu'il est nécessaire de gérer en parallèle et de manière synchronisée les aspects architecturaux et les aspects métiers. A noter aussi que la démarche NEPTUNE, fortement centrée sur les modèles, ne peut être assimilée aux démarches agiles de type XP qui se focalisent sur des cycles itératifs très courts mettant en œuvre et validant des fonctionnalités choisies au fur et à mesure du développement par le client.

3.2.1 Tâche analyse des besoins

La démarche NEPTUNE préconise de commencer le développement en identifiant le système à réaliser, de manière à fixer rigoureusement le cadre de l'application. La tâche d'analyse des besoins a pour but de recenser les acteurs et les entités passives qui communiquent avec le système (activité A de la Figure 1), d'identifier les échanges entre ces acteurs et le système (activité B), et de décrire les fonctionnalités que le système devra réaliser (activité C). Parallèlement, il convient de prendre en compte les caractéristiques techniques des éléments matériels qui supporteront l'application afin de s'assurer de la faisabilité du déploiement (activité α).

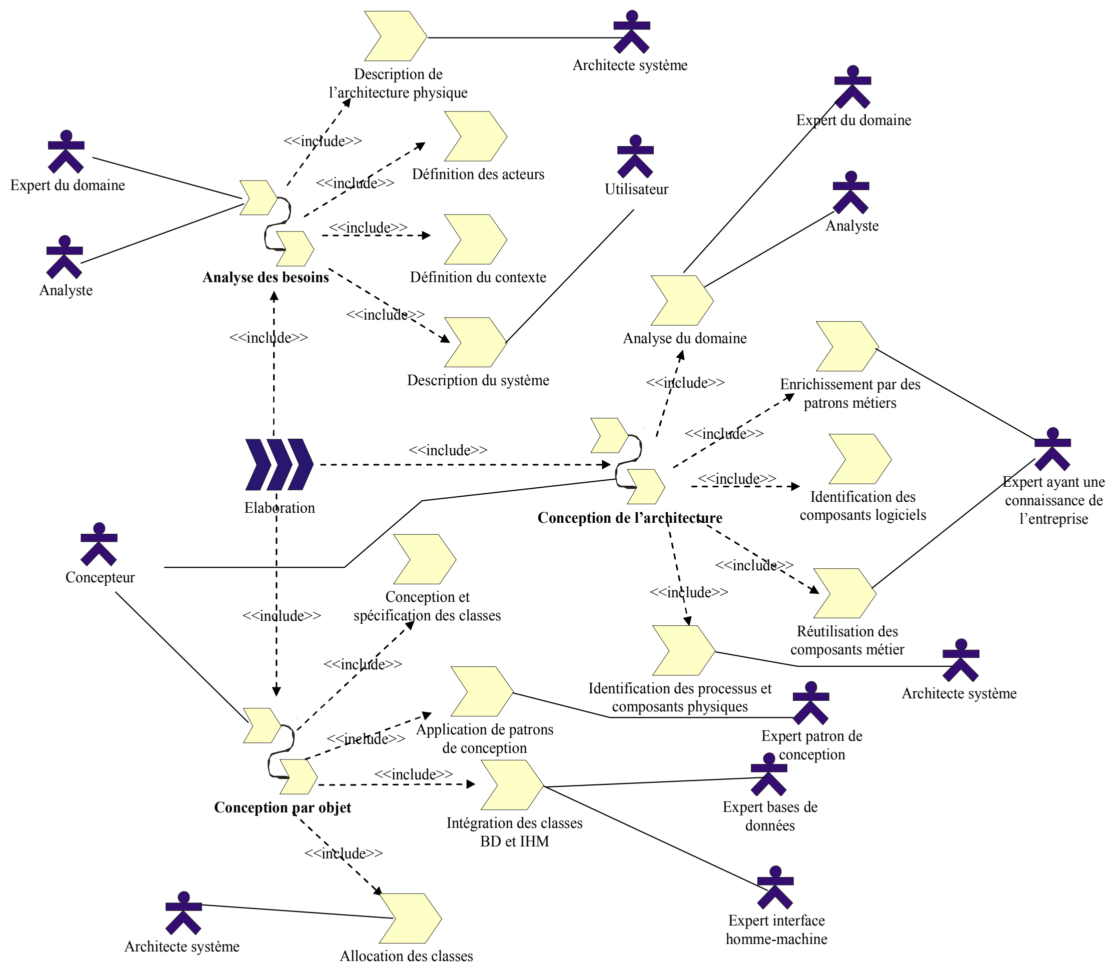

Figure 1 : Vue générale de la démarche NEPTUNE en SPEM

3.2.2 Tâche conception de l'architecture

Le processus de la démarche NEPTUNE se poursuit par la conception de l'architecture logicielle en termes de composants. Cette tâche consiste à identifier tout d'abord l'ensemble des objets métier et leurs relations qui contribuent à la réalisation des fonctionnalités du système (activité D), puis les composants logiciels gérant les objets identifiés précédemment (activité E). L'identification des composants logiciels lors de la tâche de la conception de l'architecture peut être influencées par la prise en compte des exigences et des besoins non fonctionnels (activité β). Un

expert pourra apporter son savoir-faire au niveau de la conception de l'architecture en termes de patrons métier (activité i) et de composants métier prêts à l'emploi (activité ii) vis-à-vis du système à réaliser.

3.2.3 Tâche conception par objets

La tâche de conception par objets de la démarche NEPTUNE est une projection dans le monde des objets de la réalisation de chacun des composants logiciels mis en évidence par l'architecture définie précédemment. Cette conception détaillée fera apparaître de nouvelles classes (activité F) nécessaires à la mise en œuvre du composant, donnant lieu le plus souvent à des objets typiquement informatiques. Les modèles de conception réutilisables, préférables à une mise en œuvre ad hoc, pourront à ce stade être exploités en tant que solutions éprouvées (activité iii). Il s'agira également d'identifier et de découpler du cœur de l'application les classes chargées des interfaces graphiques, et d'en faire de même pour les classes persistantes d'accès à la base de données (activité G). Le déploiement vers le matériel nécessite aussi d'associer le code de l'application aux composants physiques et aux processus (activité γ).

3.3 Description des activités de l'axe central

Dans ce qui suit, nous précisons les différentes activités de l'axe central de la démarche présentée à la Figure 1.

3.3.1 Tâche analyse des besoins

Cette tâche a pour objectif de définir un modèle métier réutilisable qui soit compatible avec les besoins.

L'activité « définition des acteurs » (activité A) distingue les acteurs actifs qui sont en interaction directe avec le système et qui déclenchent une de ses fonctionnalités, des entités passives qui produisent ou consomment des données utiles à la mise en œuvre d'une fonctionnalité. Le résultat de cette activité produit une classification de l'ensemble des acteurs, au sens UML.

L'activité « définition du contexte » (activité B) décrit et restitue le contexte passif général du système dans son environnement externe. Il s'agit de décrire les relations d'échange entre les entités passives et le système. Il en résulte un ensemble de diagrammes de collaboration et de séquence matérialisant les échanges de données.

L'activité « description du système » (activité C) détermine les cas d'utilisation et pour chacun d'entre eux l'ensemble des scénarios possibles. Elle produit le diagramme général des cas d'utilisation, la description textuelle de chaque cas d'utilisation, le diagramme général des flots de données et pour chaque scénario son diagramme de séquence.

3.3.2 Tâche conception de l'architecture

Cette tâche a pour objectif de définir d'une part l'architecture globale de l'application à base de composants et d'autre part de définir l'architecture boîte blanche de chacun des composants de l'architecture globale.

L'activité « analyse du domaine » (activité D) met en évidence le passage de l'aspect fonctionnel à l'aspect objet, en identifiant les objets métier de l'application. Elle produit un diagramme de classes et définit les scénarios secondaires, raffinements des scénarios de l'activité précédente en introduisant les objets métier. Le diagramme de classes du domaine découle des entités mises en évidence lors de la réalisation des cas d'utilisation, des entités passives et des flots de données. Il peut d'autre part être enrichi par l'application d'un patron métier.

La conception préliminaire se poursuit par l'activité « identification des composants

logiciels » (activité E) qui produit une architecture logicielle conforme aux besoins. A partir du diagramme de classes métier et des scénarios secondaires on finalise la décomposition en veillant à réduire les interactions entre composants. Une fois un composant identifié, on se focalise sur ses points d'interactions avec l'environnement pour définir ses interfaces, ce qui conduit à une architecture boîte noire du contexte dans lequel le composant est utilisé. L'identification des composants s'établit par itérations successives chaque fois que l'équipe de conception considère que le composant est structurellement et fonctionnellement complexe, donnant lieu à un niveau supplémentaire d'imbrication des composants. Ce processus de décomposition se poursuit jusqu'à ce qu'un composant soit constitué uniquement de composants boîtes noires (réutilisés) et/ou de classes.

3.3.3 Tâche conception par objets

Cette tâche a pour objectif de spécifier complètement les classes nécessaires à chacun des composants.

La conception détaillée spécifiée par l'activité « conception et spécification des classes » (activité F) a pour objectif de décrire l'architecture boîte blanche de chacun des composants de l'activité précédente. La vue boîte blanche de chaque composant est ainsi stabilisée. Ces raffinements doivent laisser invariantes les interfaces de chaque composant. Il est aussi recommandé de spécifier le comportement de chacune des classes et chacune des opérations.

Au cours de la conception, l'activité « intégration des classes BD et IHM » (activité G) devra assurer le couplage des classes dont les objets sont persistants avec les classes de l'application ; il en sera de même pour découpler l'interface graphique de l'ensemble du système proprement dit.

4. Formulation et mise en oeuvre de règles d'analyse et conception de la démarche NEPTUNE

Après avoir présenté un certain nombre de règles correspondant aux différentes activités de l'analyse et conception, sont précisés et discutés la mise en œuvre de deux règles issue de la tâche d'analyse puis de deux règles ayant toute leur importance lors de la conception. Nous utilisons OCL pour coder les règles, restreignant ainsi l'ensemble des instances valides des modèles vis-à-vis de la démarche.

4.1 Les règles d'analyse et conception

A toute activité correspond un ensemble de règles portant sur les résultats attendus en termes de modèles. Parmi ces règles, on distingue les règles intra-activités qui ne concernent que les modèles produits par cette activité des règles inter-activités liant des modèles issus de plusieurs activités. Un critère orthogonal concerne la portée de la règle : une règle peut concerner les éléments de modélisation d'un ou plusieurs diagrammes UML.

A - Définition des acteurs	*intra*	*inter*
Un acteur actif ne spécialise qu'un autre acteur actif	✓	
Une entité passive ne spécialise qu'une autre entité passive	✓	
α - Description de l'architecture physique		
Le graphe des connexions des unités physique est connexe	✓	
Tout acteur actif et entité passive sont associés à un nœud physique		✓

B - Définition du contexte		
L'entité stéréotypée « système à étudier » apparaît dans toute collaboration	✓	
Toute entité passive d'une collaboration communique directement avec le système à étudier	✓	
Aucun acteur actif n'apparaît dans une collaboration	✓	
Toute entité passive apparaît dans au moins une collaboration		✓
C - Description du système		
Tout cas d'utilisation est déclenché par un acteur actif		✓
Tout acteur actif doit être connecté à au moins un cas d'utilisation		✓
Tout cas d'utilisation est illustré par au moins un scénario	✓	
Tout acteur actif apparaît dans au moins un scénario		✓

Tableau 1 : Exemples de règles d'analyse NEPTUNE

D - Analyse du domaine	***intra***	***inter***
Tout objet d'une collaboration est instance d'une classe	✓	
La première interaction d'un scénario émane d'un acteur actif		✓
L'ensemble des messages échangés entre les acteurs et le système à étudier est invariant	✓	
A tout envoi de message entre objets correspond un lien entre classes	✓	
E - Identification des composants logiciels		
l'héritage multiple ne génère pas de conflit de nom	✓	
L'accès au code d'une classe à l'intérieur d'un composant n'exige pas la navigation de plus de n niveaux définis par le nombre de composants emboîtés auquel on ajoute la profondeur de l'arbre d'héritage du dernier composant ouvert	✓	
β - Identification des processus et des composants physiques		
Toute unité logique doit être associée à un nœud physique		✓
iii – Application de patrons de conception		
Toute classe concrète implémente directement ou par héritage une interface	✓	
δ - Allocation des classes		
Lorsque deux nœuds physiques communiquent, il doit exister sur chacun d'eux une classe qui soit en association		✓
Lorsque deux classes sont en association et qu'elles appartiennent à deux nœuds physiques différents alors ces deux nœuds sont en communication		✓

Tableau 2 : Exemples de règles de conception NEPTUNE

Dans notre approche, la définition des règles est fortement liée à la démarche envisagée, ce qui permet d'instrumenter et d'outiller les différentes activités qui la composent, et d'offrir ainsi une assistance au développement sous la forme d'un guide opératoire fournissant un diagnostique de la modélisation par application des règles. D'autres approches cherchent à produire un ensemble exhaustif de règles de bonne formation des modèles, indépendamment de tout contexte [24], ce qui revient à privilégier une certaine sémantique « universelle » des modèles UML.

4.2 Mise en œuvre de règles d'analyse

La tâche d'analyse semble être une phase privilégiée pour guider le processus de développement à l'aide de règles OCL, exprimant ainsi des contraintes fortes sur les modèles attendus en fin d'activité

4.2.1 Tout acteur actif doit être connecté à au moins un cas d'utilisation

Cette règle est de type inter-activité. L'activité « description du système » consiste à déterminer ses fonctionnalités par un diagramme général de cas d'utilisation. Si une fonctionnalité n'a de sens que si elle répond à un besoin utilisateur, un acteur n'a d'intérêt, dans une analyse, que s'il interagit directement avec le système. Il est donc nécessaire de vérifier le bien fondé de tout acteur en sortie de cette activité.

Le fragment de méta-modèle proposé à la Figure 3 montre les éléments de modélisation concernés par cette règle.

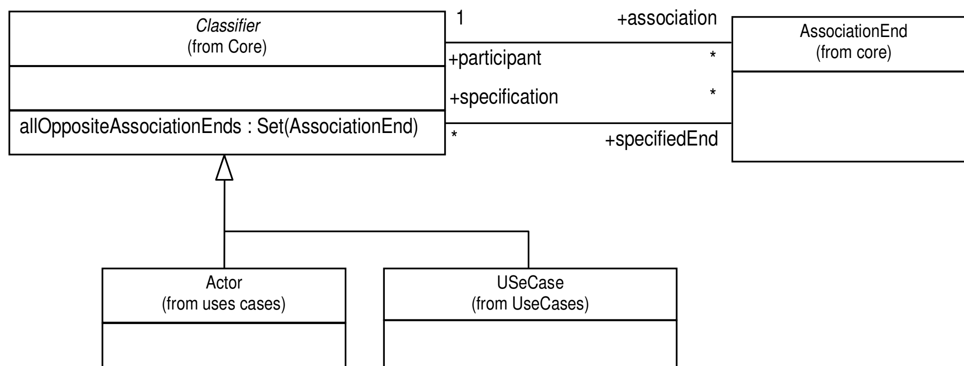

Figure 3 : Le fragment du méta-modèle connectant acteurs et cas d'utilisations

Cette règle peut donc s'énoncer de la manière suivante en fonction de la structuration des données de ce fragment de méta modèle : pour tout acteur (1), vérifier qu'il existe dans l'ensemble des entités connectées à l'acteur (2) une entité de type cas d'utilisation (3).

D'où la règle OCL suivante :

```
Context Behavioral_Elements::Use_Cases::Actor                          (1)
inv actorConnectionToUseCase:
  self.allOppositeAssociationEnds.                                     (2)
    Participant->exists(c : Classifier | c.oclIsKindOf(UseCase))       (3)
```

4.2.2 Tout acteur actif apparaît dans au moins un scénario

Cette règle est de type inter-activité. Si dans l'activité « description du système », le diagramme des cas d'utilisation illustre le rôle de chaque acteur dans le système, cette vision n'est qu'une vision macroscopique. Elle ne présente pas la description précise des interactions entre le système et chaque acteur à travers chacune des fonctionnalités. La démarche impose que soit réalisé chaque scénario sous forme d'une collaboration, ceux-ci devant servir de base à la rédaction ultérieure des plans de tests. Comme tout cas d'utilisation est illustré par au moins un scénario, il apparaît nécessaire de compléter la règle précédente en exigeant que tout acteur actif apparaisse dans au moins un scénario.

Le fragment de méta-modèle proposé à la figure 4 montre les éléments de modélisation concernés par cette règle.

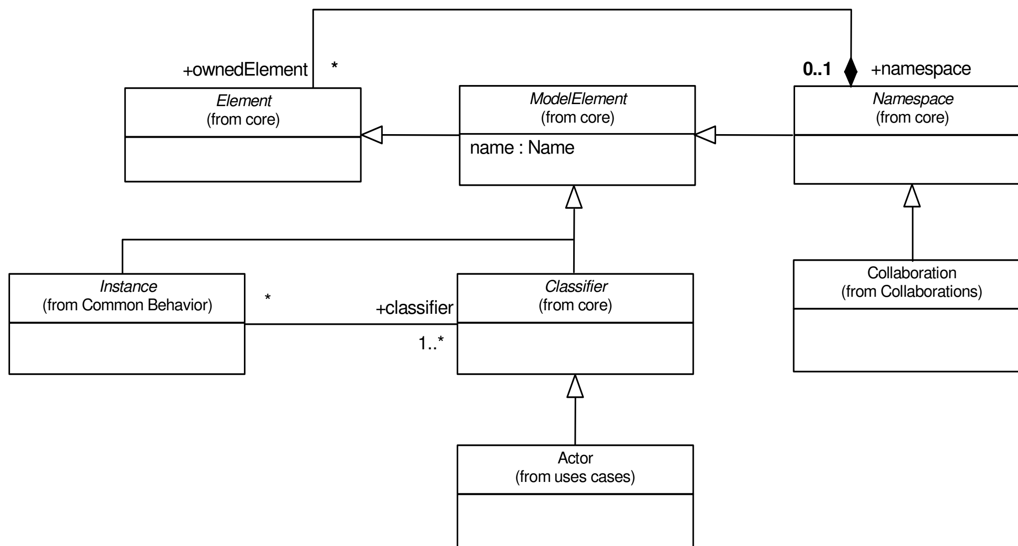

Figure 4 : Le fragment du méta-modèle connectant acteurs et collaborations

Cette règle peut donc s'énoncer de la manière suivante : pour tout acteur (1), vérifier qu'une de ses instances (3) soit incluse (4) dans une collaboration (2).

D'où la règle OCL suivante :

```
context Behavioral_Elements::Use_Cases::Actor                              (1)
 inv actorPresentDansCollaboration:
 Collaboration.allInstances.ownedElement->                                 (2)
   select(me:ModelElement | me.oclIsKindOf(Instance)).oclAsType(Instance). (3)
      classifier->includes(self)                                           (4)
```

4.3 Mise en œuvre de règles de conception

Lors de la conception, nous identifions trois catégories de règles : les règles spécifiques à un contexte de développement, les règles génériques dépendantes de la démarche et enfin les règles dites incitatives permettant de donner des indications aux concepteurs afin de faciliter ou d'encourager l'enrichissement de modèles. Il existe par ailleurs des règles de « bonne formation de modèles étendues » et des règles « paramétrables » traduisant des métriques sur les modèles. Par exemple, les règles suivantes couvrent les types de règles énoncés :

- *l'héritage multiple ne génère pas de conflit de nom* est une règle spécifique et de bonne formation ;
- *toute classe concrète implémente directement ou par héritage une interface* est une règle incitative, de bonne formation qui englobe les interfaces comme paramètres d'une conception ;
- *l'accès au code d'une classe à l'intérieur d'un composant n'exige pas la navigation de plus de n niveaux définis par le nombre de composants emboîtés auquel on ajoute la profondeur de l'arbre d'héritage du dernier composant ouvert* est une règle générique qui introduit une métrique sur la structuration des composants.

Dans ce qui suit, nous explicitons les deux premières règles.

4.3.1 L'héritage multiple ne génère pas de conflit de nom

Dans le cadre d'un projet spécifique, un langage à objets acceptant l'héritage multiple est utilisé. Le but ici n'est pas d'interdire purement et simplement ce trait du langage, mais plutôt d'en limiter l'usage à une sémantique définie et non ambiguë. Même si ce type de conflit peut être résolu par le concepteur de la classe concernée, la complexité des futures évolutions s'en trouve accrue. Si par ailleurs, les attributs doivent être uniquement de visibilité privé, le conflit de nom sur les attributs ainsi que le problème de la gestion de l'héritage répété se trouvent résolus. D'autre part, le conflit de nom peut être autorisé s'il n'est pas la source d'un conflit de valeur potentiel, c'est le cas lorsque la forme d'héritage multiple proposé par le langage Java est utilisée.

Le fragment de méta-modèle proposé à la Figure 5 montre les éléments de modélisation concernés par cette règle.

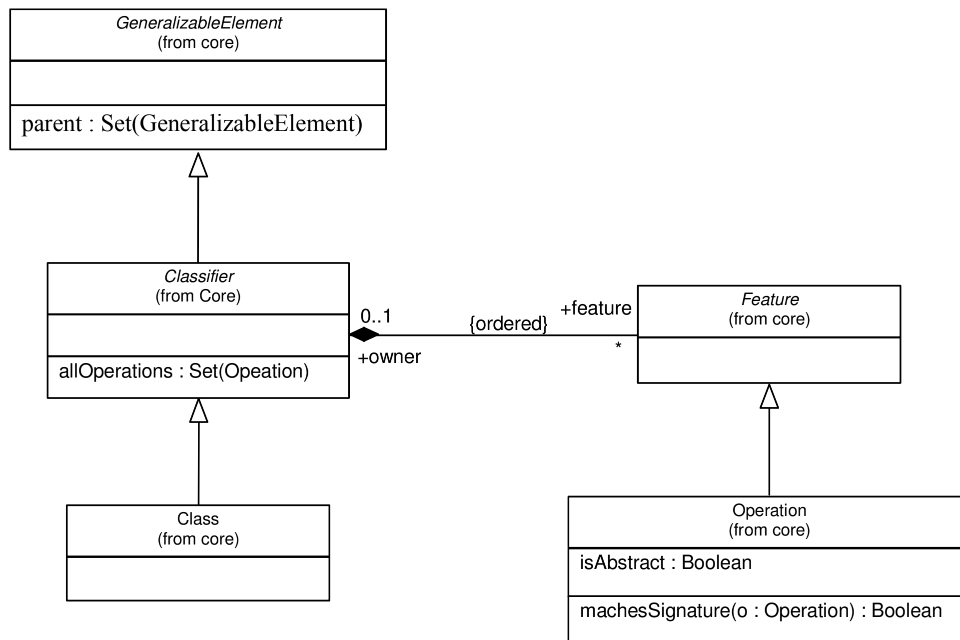

Figure 5 : Fragment du méta-modèle nécessaire à l'expression de la règle de l'héritage multiple

Cette règle peut donc s'énoncer de la manière suivante : pour toute classe d'un modèle (1) ayant plus d'une super-classe (2), alors (3) pour tout couple de super-classes distinctes, il faut vérifier que (4) toute opération concrète de l'une des classes du couple (5) n'a pas même signature (7) que l'une des opérations concrètes de l'autre (6). La sémantique des signatures équivalentes est donnée par la méthode *matchesSignature* qui prend en considération le nom de l'opération et la liste des types d'arguments, définie par le standard UML. Pour moins de flexibilité vis-à-vis de la conformité des signatures, l'utilisation de la méthode *hasSameSignature* peut s'envisager.

D'où la règle OCL suivante :

```
context Foundation::Core::Class                                          (1)
inv utilisationHeritageMultiple:
self.parent->select(ge:GeneralizableElement|ge.oclIsKindOf(Class)->size>1    (2)
  implies                                                                (3)
self.parent->
   select(ge:GeneralizableElement|ge.oclIsKindOf(Class)).oclAsType(Class)->
      forAll(c, c1 : Class | c<>c1 implies                               (4)
      c.allOperations->select(op : Operation | not(op.isAbstract))->     (5)
        forAll(op : Operation | c1.allOperations->
          select(op1 : Operation | not(op1.isAbstract))->                (6)
            forAll(op1 : Operation | not(op1.matchesSignature(op)))))    (7)
```

4.3.2 Toute classe concrète implémente directement ou indirectement une interface

Erich Gamma, l'un des fondateurs des patrons de conception, préconise de programmer pour une interface et non pour un développement [11]. L'idée d'une telle règle incitative, qui pourrait préparer à l'activité *application de patrons de conception* (activité iii de la Figure 1) de la démarche NEPTUNE, est de promouvoir une conception réutilisable. Les règles incitatives n'ont pas un caractère impératif, mais peuvent par exemple être activées pour une revue de code ou une séance de *refactoring*. Le but de cette règle est alors de s'assurer que toute classe concrète concoure à au moins une fonctionnalité interne ou externe au composant. Toutes les classes ne sont pas en relation directe avec une interface, mais les interfaces exhibées se doivent d'avoir une valeur ajoutée dans le sens ou elles proposent un comportement commun à un ensemble de classes.

Le fragment de méta-modèle proposé à la Figure 9 montre les éléments de modélisation concernés par cette règle.

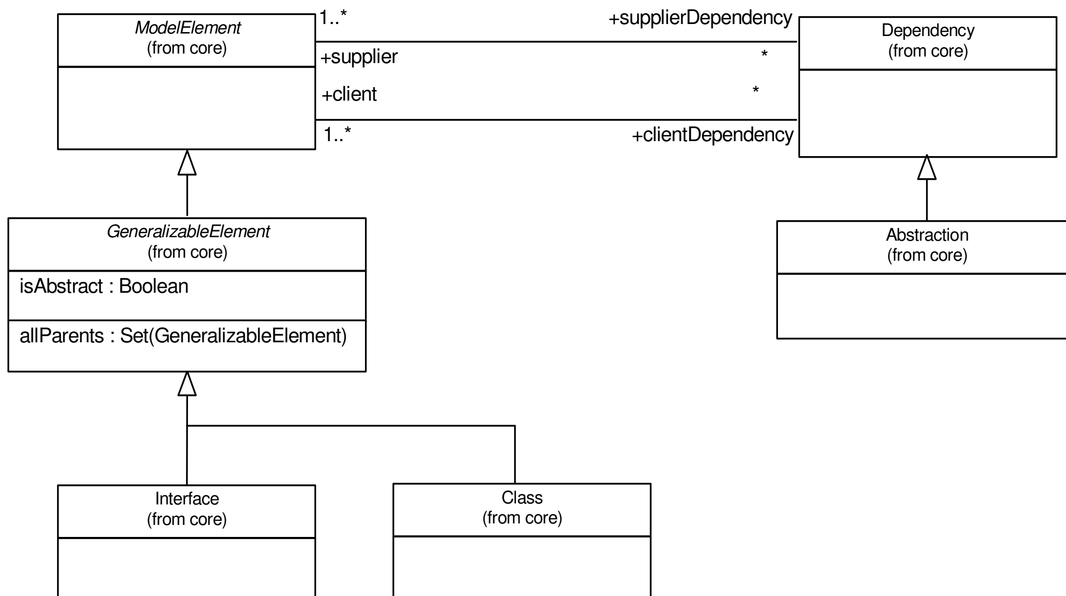

Figure 6 : Fragment du méta-modèle nécessaire à l'expression de la règle programmer pour une interface

Cette règle peut maintenant s'écrire : pour toute classe concrète d'un modèle (1, 2), il existe dans l'ensemble des ancêtres de la classe, elle-même incluse (3), au moins une relation de dépendance de type *Abstraction* (4) avec une interface (5).

D'où la règle OCL suivante :

```
context Foundation::Core::Class                                      (1)
inv programmerPourUneInterface :
  not(self.isAbstract) implies                                       (2)
  Bag{self}->union(self.allParents->select(ge:GeneralizableElement|
      ge.oclIsKindOf(Class)).oclAsType(Class)).                      (3)
        clientDependency->select(d:Dependency| d.oclIsKindOf(Abstraction)).  (4)
           supplier->exists(me:ModelElement| me.oclIsKindOf(Interface))     (5)
```

5. Conclusion

Dans ce papier, nous avons montré comment spécifier une démarche de développement orienté modèles (visibilité interne) tout en organisant son outillage (visibilité externe). Ce qui assure un suivi du bon déroulement de la démarche considérée avec un diagnostique à chaque point de

contrôle donnant un fil conducteur commun à tous les acteurs de la démarche. L'outillage de la démarche NEPTUNE a été validé sur des modèles conséquents issus du milieu industriel [6]. Il a permis de fédérer les équipes de développement qui ont pu s'approprier la démarche et détecter des erreurs de conception non vus par les différentes relectures des modèles, ce qui a diminué d'à peu près 10% le temps de validation de la modélisation. La suite logique de nos travaux consiste à appliquer notre processus « méta-méthodologique » à d'autres démarches de développement orientées modèles et respectant les critères énoncés à la section outillage.

En édictant et codant un ensemble de règles OCL déduit de l'étude de la démarche NEPTUNE, un certain nombre de questions nous sont apparues importantes à appréhender :

- Les mêmes modèles traversent les différentes phases du cycle de vie du logiciel, et se composent des mêmes artefacts de développement vus selon plusieurs niveaux d'abstractions obtenus par affinages successifs. Cela nuit à la gestion de l'historique de la modélisation et à l'écriture de certaines règles méthodologiques de type inter-activité. Quel serait alors un système efficace pouvant gérer la traçabilité des éléments de modélisation et qui serait compatible avec la norme ?
- Le codage des règles OCL reste une affaire de spécialistes. En effet, bien que le langage OCL soit de haut niveau, trouver le fragment de méta-modèle, les liens de navigation ainsi que les opérations additionnelles nécessaire, demande une expertise des méta-modèles UML, toujours en cours d'évolution. Une fois la représentation UML de chaque élément de modélisation de la démarche fixée, ne pourrait-on pas concevoir un générateur de requêtes OCL à partir des règles édictées dans un langage de plus haut niveau ?

Nos travaux portent aussi sur la définition et l'outillage des activités d'« expertise » de la démarche NEPTUNE. Nous pensons favoriser l'application de patrons de conception par un ensemble de règles incitatives. En ce qui concerne l'injection de patrons de conception dans une conception existante ainsi que la complétion d'un patron métier par les spécificités des modèles d'analyse du domaine étudié, nous comptons utiliser le futur langage normalisé permettant des transformations sur les modèles UML. Aussi, nous pensons doter OCL d'un ensemble d'opérations additionnelles de haut niveau applicables à des données fortement structurées, tels que des arbres et des graphes.

6. Bibliographie

[1] K. Beck. Extreme Programming Explained – Embrace Change Addison-Wesley, 2000

[2] J. Bézivin, M. Blay, M. Bouzeghoub, J. Estublier, J.M. Favre. Rapport de Synthèse de l'Action CNRS sur le MDA (Model Driven Architecture) Chapitre principal du rapport de l'Action Spécifique CNRS sur le MDA, Décembre 2004

[3] M. Blay, P.Franchi. Rapport de Synthèse de l'Action CNRS sur le MDA (Model Driven Architecture) Chapitre « Espace technologique » langages du rapport de l'Action Spécifique CNRS sur le MDA, Décembre 2004

[4] J.-M. Bruel, G. Georg, H. Hussmann, I. Ober, C. Pohl, J. Whittle, S. Zschaler. Models for Non-functional Aspects of Component-Based Software (NfC'04), UML 2004 Modeling Languages and Applications, UML Satellite Activities, LNCS 3297: pp 62-66

[5] D. Coleman. Object-Oriented Development: The Fusion Method, Prentice Hall, 1993

[6] J.-C. Cruellas, J.-P.Bodeveix, T. Millan, A. Canals. The NEPTUNE Technology to Verify and to Document Software Components in Business Component-Based Software Engineering, ed. Franck Barbier - Kluwer Academic Publishers 2003, chapter 6, pages 101-118

[7] D.-F. D'Souza, A.-C. Wills. Objects, Components, and Frameworks with UML : The

Catalysis Approach Addison-Wesley, 1998

[8] I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process, Addison-Wesley, 1999

[9] Y. Jin, R. Esser, W.-J.Jörn. A method for describing the syntax and semantics of UML statecharts, in Software and Systems Modeling, Issue: Volume 3, Number 2. Springer-Verlag Heidelberg, May 2004, pages 150-163

[10] J.M. Favre. Towards a Basic Theory to model Driven Engineering Workshop on Software Model Engineering à UML 2004, Lisbonne, Octobre 2004

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns Catalogue de modèles de conception réutilisables, Vuibert, 1999

[12] A. Kleppe, J. Warmer, W. Bast. MDA explained - the Model Driven Architecture : Practice and Promise - Addison-Wesley, 2003

[13] P. Kroll, P. Kruchten. Guide pratique du RUP, CampusPress, 2003

[14] T. Mens, K. Czarnecki, P. Van Gorp. A Taxonomy of Model Transformations, in Proceedings of Dagstuhl 04101 Language Engineering for Model-Driven Software Development, Jean Bezivin et Reiko Heckel (eds), 2005

[15] NEPTUNE Consortium. Report of the Experimentation Stage, NEPTUNE Report, January 2003, http://neptune.irit.fr/Public/AnglaisV2/InformationDownload.html

[16] OMG. Response to the UML 2.0 OCL RFP (ad/2000-09-03). http://www.omg.org/docs/ad/02-05-09.pdf. pp. 2.1-2.25. Version 1.5, June 2002

[17] OMG. Final MetaObjectFacility (MOF) 1.4 specification : change-barred / annotated version, August 2001, http://www.omg.org/docs/ptc/01-08-22.pdf

[18] OMG. Unified Modeling Language Specification (draft) Version 1.4 draft, February 2001, http://cgi.omg.org/cgi-bin/doc?ad/01-02-14

[19] OMG. Software Process Engineering Metamodel Specification Version 1.0, November 2002, formal/02-11-14

[20] P.-J. Robinson. Hierarchical Object-Oriented Design Prentice Hall, London; 1992

[21] P. Roques, F. Vallée. UML 2 en action, Editions Eyrolles, 2004

[22] J.-M. Sobel, D.-P. Friedman. An Introduction to Reflection-Oriented Programming. Reflection'96 San Francisco, CA, 1996

[23] Softeam. Profils UML et langage J : Contrôlez totalement le développement d'applications avec UML - white paper - 1999 - http://www.softeam.fr/pdf/fr/uml_profiles.pdf

[24] J.-L. Sourrouille, G. Caplat. A Pragmatic View about Consistency Checking of UML Model, Workshop on Consistency Problems in UML-Based Software Development, 2003, pp43-50

[25] K. E. Wiegers. Creating a Software Engineering Culture Dorset House Publishing Company, 1996