



Mention SMIS
Master 2 de Recherche
Spécialité Sûreté du Logiciel et Calcul à Haute Performance



118, route de Narbonne
31062 Toulouse Cedex

Détection de particularités structurelles de modèles pour l'injection de patrons de conception

Cédric BOUHOURS

Sous la direction de :

Pr C. PERCEBOIS, Professeur, UPS

Dr H. LEBLANC, Maître de Conférences, UPS

Année 2005 - 2006

Mots-clef :

Patrons de conception, Particularités remarquables, Modèles alternatifs, Injection, Transformation

Résumé :

Les patrons de conception représentent un savoir-faire sous forme de microarchitecture de classes réutilisables. Ils proposent des solutions élégantes à des problèmes de modélisation donnés. Cependant, un nombre conséquent de modèles industriels en sont dépourvus. Nous pensons donc qu'un outil d'aide au *refactoring* de modèles, par l'injection de patrons de conception, permettrait de clarifier et d'optimiser ces modèles. Dans cette première étude, nous proposons un moyen de détecter des fragments de modèles, à partir desquels l'intégration de patrons est envisageable. D'une expérience menée avec des étudiants en cursus informatique, nous avons obtenu des modèles alternatifs aux patrons, desquels nous avons déduit un ensemble de particularités remarquables. La mise en œuvre de la détection de ces particularités, nous a permis de cibler des fragments de modèles.

Détection de particularités
structurelles de modèles pour
l'injection de patrons de conception

Modèles alternatifs.
Particularités remarquables.
Règles OCL.

Cédric BOUHOURS

Master 2 Recherche - Sûreté du Logiciel et Calcul à Hautes Performances

Sous la direction de :

Pr C. PERCEBOIS, Professeur, UPS

Dr H. LEBLANC, Maître de Conférences, UPS

REMERCIEMENTS

Je tiens sincèrement à remercier mon tuteur, **Dr. Hervé LEBLANC**, maître de conférences, pour m'avoir permis de réaliser ce stage particulièrement intéressant, et pour toute la confiance qu'il m'a accordée durant toute cette année.

Je remercie également **Pr. Christian PERCEBOIS**, professeur et responsable de l'équipe MACAO, pour tous les précieux conseils qu'il m'a prodigué pendant mon stage.

Je remercie ensuite **Dr. Thierry MILLAN**, maître de conférences, pour son aide et sa disponibilité vis-à-vis des problèmes techniques qui ont pu survenir pendant ce stage.

De manière générale, je remercie tous les membres de l'équipe pour l'accueil et la convivialité qu'ils m'ont témoignés tout au long de mon stage

SOMMAIRE

REMERCIEMENTS	3
ILLUSTRATIONS	5
POSITIONNEMENT DU PROBLÈME	6
1. ETAT DE L'ART	6
1.1) <i>Les patrons</i>	6
1.2) <i>Les patrons de conception</i>	7
1.3) <i>Travaux connexes</i>	9
2. PROBLEMATIQUE	10
2.1) <i>Contexte général de l'étude</i>	10
2.2) <i>Contexte particulier de l'étude</i>	11
CŒUR DE L'ÉTUDE	13
1. RECHERCHE DE MODELES ALTERNATIFS	13
1.1) <i>Une expérience</i>	13
1.2) <i>Problème posé aux étudiants</i>	13
1.3) <i>Sélection des modèles alternatifs</i>	15
2. DEDUCTION DES PARTICULARITES REMARQUABLES	15
2.1) <i>Problème soluble par le Composite</i>	17
2.2) <i>Problème soluble par le Pont</i>	22
2.3) <i>Problème soluble par le Décorateur</i>	25
2.4) <i>Problème soluble par l'Adaptateur</i>	28
2.5) <i>Problème soluble par la Façade</i>	30
2.6) <i>Problème soluble par le Poids mouche</i>	32
2.7) <i>Problème soluble par la Procuration</i>	33
3. MISE EN ŒUVRE	35
4. EXPERIMENTATION	38
4.1) <i>Expérimentation sur des modèles</i>	38
4.2) <i>Expérimentation sur des méta-modèles</i>	40
CONCLUSION	44
BIBLIOGRAPHIE	47
ANNEXES	49

ILLUSTRATIONS

FIGURE 1 : DIAGRAMME D'ACTIVITE : INTEGRATION DE PATRONS DE CONCEPTIONS	12
FIGURE 2 : LA DEMARCHE NEPTUNE	11
FIGURE 3 : MODELE DE REFERENCE DU PATRON COMPOSITE	17
FIGURE 4 : MODELE ALTERNATIF N°1 DU COMPOSITE	18
FIGURE 5 : MODELE ALTERNATIF N°2 DU COMPOSITE	19
FIGURE 6 : MODELE ALTERNATIF N°3 DU COMPOSITE	19
FIGURE 7 : MODELE ALTERNATIF N°4 DU COMPOSITE	20
FIGURE 8 : MODELE ALTERNATIF N°5 DU COMPOSITE	21
FIGURE 9 : ILLUSTRATION DE LA MULTIPLICATION DES CLASSES LORSQUE LE PATRON PONT N'EST PAS UTILISE	22
FIGURE 10 : MODELE DE REFERENCE DU PATRON PONT	23
FIGURE 11 : MODELE ALTERNATIF N°1 DU PONT	23
FIGURE 12 : MODELE ALTERNATIF N°2 DU PONT	24
FIGURE 13 : MODELE DE REFERENCE DU DECORATEUR	26
FIGURE 14 : MODELE ALTERNATIF N°1 DU DECORATEUR	26
FIGURE 15 : MODELE ALTERNATIF N°2 DU DECORATEUR	27
FIGURE 16 : MODELE DE REFERENCE DE L'ADAPTATEUR	28
FIGURE 17 : MODELE ALTERNATIF N°1 POUR L'ADAPTATEUR	29
FIGURE 18 : MODELE DE REFERENCE DE FAÇADE	30
FIGURE 20 : MODELE DE REFERENCE DE POIDS MOUCHE	33
FIGURE 21 : MODELE DE REFERENCE DE PROCURATION	34
FIGURE 22 : MODELE ALTERNATIF N°1 DU COMPOSITE	35
FIGURE 23 : EXTRAIT DU META-MODELE UML 1.4	36
FIGURE 24 : EXTRAIT DU MODELE "GENERATEUR DE META-MODELES" DE NEPTUNE 2	39
FIGURE 25 : FRAGMENT DU SPEM 1.1	41
FIGURE 26 : INTEGRATION DES PATRONS COMPOSITE ET DECORATEUR DANS LE SPEM 1.1	42
FIGURE 27 : INTEGRATION D'UN ADAPTATEUR DANS UN FRAGMENT DU NOYAU DE UML 1.4	43

1. ETAT DE L'ART

1.1) Les patrons

Par définition, un patron est un modèle. En couture, il est le modèle à suivre pour coudre un vêtement, en géométrie, il est le modèle permettant de plier une figure. Ainsi, à priori, dès qu'il est question de modèle, la notion de patron est présente. Un architecte du nom de Christopher Alexander a, le premier, étudié les modèles dans le bâtiment et les collectivités. Il a mis au point un langage de modélisation en disant que "chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière" [C. Alexander, 1977]. L'idée des patrons étant applicable à tout procédé de construction, Kent Beck et Ward Cunningham ont réutilisé cette idée dans le contexte des développements de systèmes à objets [K. Beck, *et al*, 1987].

Appelés également "motifs", ils représentent des modèles permettant de résoudre des problèmes de modélisation, à différentes échelles. En général, un patron possède quatre éléments essentiels :

- Un nom qui est un moyen de décrire en un ou deux mots un problème, ses solutions et leurs conséquences.
- Un problème qui décrit les situations où le modèle s'applique. Il expose le sujet à traiter et son contexte.
- Une solution qui décrit les éléments pour résoudre le problème, les relations entre eux, leurs parts dans la solution, et leur coopération.
- Des conséquences qui sont les effets résultants de la mise en œuvre du modèle et les compromis que cela entraîne.

En fonction du type de problème et de la granularité des modèles envisagés, il existe plusieurs catégories de patrons. Par exemple, les patrons *métiers* tendent à résoudre des problèmes organisationnels liés à un métier donné. Ils prennent en compte un ensemble de besoins informationnels associés d'une part aux différents processus industriels de conception, fabrication, production, qualité etc. et d'autre part à différents métiers et donc acteurs humains qui ont à coopérer dans la réalisation de ces processus. Les patrons *architecturaux* représentent des organisations structurelles fondamentales de systèmes informatiques complets. Ils fournissent un ensemble de sous-systèmes prédéfinis, spécifient leurs responsabilités et organisent les relations entre-deux. L'exemple le plus connu est le patron Modèle Vue Contrôleur, introduit par les concepteurs de *SmallTalk*. Les patrons *de conception* sont un recueil de l'expertise, en matière de conception, de logiciels orientés-objet. Chaque patron nomme, explique et évalue un concept important qui figure fréquemment dans les systèmes orientés-objet. Ils facilitent la réutilisation de solutions de conception et d'architectures efficaces. La représentation par des modèles de conception de techniques standards, rend celles-ci plus facilement accessibles aux développeurs de nouveaux systèmes. Les patrons de conception peuvent même améliorer la documentation et la maintenance d'un système existant

en donnant explicitement la spécification des relations entre classes et objets. Nous nous intéressons plus particulièrement aux patrons de conception parce qu'ils concernent directement notre étude.

1.2) Les patrons de conception

Un patron de conception donne un nom, isole et identifie les principes fondamentaux d'une structure générale, pour en faire un moyen utile à l'élaboration d'une conception orientée objet réutilisable. Le patron de conception détermine les classes et instances intervenantes, leurs rôles et leur coopération. Chaque patron décrit un problème de conception, les circonstances où il s'applique, ainsi que les effets résultants de sa mise en œuvre, et les compromis sous-entendus. Les patrons de conception les plus répandus ont été proposés et classés par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides [E. Gamma, *et al*, 1995] nommés pour le reste de ce rapport le GOF (Gang of Four). Ils ont classé les patrons de conception dans un catalogue, selon trois types.

1.2.1) Les patrons créateurs

Ils proposent des solutions pour gérer l'instanciation d'objets complexes (héritage, délégation...).

- *la Fabrique Abstraite* : Elle fournit une interface pour créer des familles d'objets apparentés ou dépendants, sans avoir à spécifier leurs classes concrètes.
- *le Monteur* : Il dissocie la construction de la représentation d'un objet complexe, de sorte que le même procédé de construction puisse engendrer des représentations différentes.
- *la Fabrication* : Elle définit une interface pour la création d'un objet, tout en laissant à ses sous-classes le choix de la classe à instancier.
- *le Prototype* : En utilisant une instance type, il crée de nouveaux objets par clonage.
- *le Singleton* : Il garantit qu'une classe n'a qu'une seule instance, et fournit à celle-ci un point d'accès de type global.

1.2.2) Les patrons structurels

Ils proposent des schémas de classes et d'objets pour réaliser des structures plus complexes.

- *l'Adaptateur* : Il convertit l'interface d'une classe existante en une interface conforme à l'attente de l'utilisateur. Il permet à des classes de travailler ensemble malgré leur incompatibilité d'interface.
- *le Pont* : Il découple une abstraction de son implémentation, afin que les deux puissent être modifiés indépendamment.

- *le Composite* : Il organise les objets en structure arborescente représentant une hiérarchie de composition. Il permet un traitement uniforme des objets individuels, et des objets composés.
- *le Décorateur* : Il attache des responsabilités supplémentaires à un objet de façon dynamique. Il offre une solution alternative à la dérivation de classes pour l'extension de fonctionnalités.
- *la Façade* : Elle fournit une interface unifiée pour un ensemble d'interfaces d'un sous-système. Elle définit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.
- *le Poids mouche* : Il supporte de manière efficace un grand nombre d'instances de granularité fine.
- *la Procuration* : Elle permet de remplacer temporairement un objet par un autre, pour en contrôler l'accès.

1.2.3) Les patrons comportementaux

Ils traitent du placement des algorithmes entre plusieurs classes et simplifient la dynamique des responsabilités entre les objets.

- *la Chaîne de responsabilité* : Elle permet de découpler l'expéditeur d'une requête à son destinataire, en donnant la possibilité à plusieurs objets de prendre en charge la requête. De ce fait, une chaîne d'objets récepteurs est créée pour faire transiter la requête jusqu'à ce qu'un objet soit capable de la prendre en charge.
- *la Commande* : Elle réifie une requête, ce qui permet de faire un paramétrage des clients avec différentes requêtes, files d'attentes, ou historique de requêtes, et d'assurer le traitement des opérations réversibles.
- *l'Interprète* : Pour un langage donné, il définit une représentation objet de la grammaire utilisable par un interprète pour analyser des phases du langage.
- *l'Itérateur* : Il fournit un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation sous-jacente.
- *le Médiateur* : Il définit un objet qui encapsule les modalités d'interaction de divers objets. Il factorise les couplages faibles, en dispensant les objets d'avoir à faire référence explicite les uns aux autres ; de plus, il permet de modifier une relation indépendamment des autres.
- *le Memento* : Sans violer l'encapsulation, il acquiert et délivre à l'extérieur une information sur l'état interne d'un objet, afin que celui-ci puisse être rétabli ultérieurement dans cet état.
- *l'Observateur* : Il définit une corrélation entre des objets de façon que lorsqu'un objet change d'état, tous ceux qui en dépendent, en soient notifiés et mis à jour automatiquement.
- *l'Etat* : Il permet à un objet de modifier son comportement lorsque son état interne change.
- *la Stratégie* : Elle définit une famille d'algorithmes, les encapsule, et les rend interchangeables. Une stratégie permet de modifier un algorithme indépendant de ses clients.
- *le Patron de méthode* : Il définit le squelette de l'algorithme d'une opération, en déléguant le traitement de certaines étapes à ses sous-classes. Le patron de

méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de l'algorithme.

- *le Visiteur* : Il représente une opération à effectuer sur les éléments d'une structure d'objet. Le visiteur permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère.

1.2.4) Les extensions du GOF

Les patrons de conception ne se cantonnent pas à ceux décrits par le GOF. Dirk Riehle a essayé, entre autre, de composer les patrons de conception entre eux pour obtenir des patrons dit "composites" [D. Riehle, 1997]. Un exemple de patron composite, est le patron *Bureaucracy* [D. Riehle, 1996]. Ce patron représente une hiérarchie d'objets, avec des rôles différents mais hiérarchiques (Employé, Manager, Subordonné, Directeur). Chaque niveau ayant en plus le rôle du niveau en dessous. Cette structure hiérarchique est définie grâce au patron de conception *Composite*, la communication entre les niveaux est assurées par les patrons *Chaîne de responsabilité* et *Observer*. La notion de niveau est, quant à elle, gérée par le patron *Manager*.

A l'inverse, les notions de GRASP patterns (General Responsibility Assignment Patterns), fondées sur des modèles d'affectation des responsabilités, proposées par Graig Larman [G. Larman, 2002], sont des microarchitectures de classes sur lesquels sont basés les patrons de conceptions. Elles représentent une aide didactique intéressante lors de la modélisation objet, et pourrait faire l'objet d'une mise en forme précédant l'intégration des patrons proprement dite.

1.3) Travaux connexes

L'efficacité des patrons n'est, à l'heure actuelle, plus à prouver. De nombreux travaux [C. Cauvet, *et al*, 2000], [B. Barthez, *et al*, 2000] ont déjà montré l'intérêt qu'il y avait à les réutiliser. Mais le problème persistant tient à la sélection des patrons à réutiliser. Même si les patrons de conception du GOF sont présentés avec une section "motivation" permettant de les choisir en fonction du problème qu'ils résolvent, d'autres patrons n'ont pas été classifiés. De plus, cette classification ne permet pas toujours de retrouver le patron à utiliser, notamment quand il s'agit de faire collaborer les patrons entre eux. Les travaux de [A. Conte, *et al*, mai 2001] consistent à améliorer cette classification, en proposant un nouveau formalisme de représentation des patrons. Ce formalisme, appelé P-Sigma, effectue un classement selon trois parties. Une partie "Interface", qui contient tous les éléments qui permettent la sélection d'un patron, une partie "Réalisation" qui exprime la solution d'un patron en terme de modèle et de solution, et une partie "Relation", qui permet d'organiser les relations entre patrons, donc d'organiser des catalogues de patrons. Même si des catalogues de patrons sont développés, le problème du choix restera présent, car il faudra effectuer une recherche dans des listes parfois longues de patrons, surtout en ce qui concerne les patrons métiers. Des outils existent pour aider à cette recherche, en classifiant à nouveau les patrons dans des rubriques représentées par des collections des mots ou de phrases. Sur la même idée, l'atelier AGAP [A. Conte, *et al*, 2001], destiné aux ingénieurs de patrons (ceux qui spécifient des catalogues de patrons),

permet de définir plusieurs systèmes de patrons en utilisant le même formalisme ou en réutilisant des rubriques de formalismes existants.

Aujourd'hui, malgré les efforts pour améliorer la classification des patrons de conception, et des outils d'aide à l'intégration de patrons dans des modèles, nous n'avons pas trouvé d'outil inspectant des modèles, et incitant, de manière la plus automatique possible, à l'utilisation de patrons.

2. PROBLEMATIQUE

2.1) Contexte général de l'étude

Les processus de développement orientés modèles sont conduits à réutiliser un savoir-faire d'experts exprimé le plus souvent en termes d'éléments de modélisation d'analyse ou de conception approuvés par une communauté. Il en est ainsi des patrons métiers ou des patrons de conception nécessaires à l'élaboration d'une architecture fondée sur les modèles permettant une factorisation maximale des artefacts de développement.

Les processus de développement orientés modèles se doivent d'être guidés par des transformations :

- Lors de la construction des modèles d'analyse propres à l'entreprise, il est envisageable d'intégrer de nouvelles fonctionnalités à un modèle par une transformation de type « greffe » permettant d'enrichir la portée du modèle initial par un patron métier.
- La transition de l'analyse à la conception est souvent délicate et cruciale car les adaptations nécessaires aux choix techniques et non fonctionnels perturbent le modèle métier qui tend à fusionner avec les paradigmes des plates-formes d'exécution et de déploiement envisagées. Les transformations envisagées sont alors l'« adaptation », puis l'« injection », de patrons de conception dans un modèle métier afin de faciliter les futures transformations de tissage et de fusion préconisées par les processus de développement MDE en Y.
- Ces transformations s'appuient sur un ensemble de transformations élémentaires automatisables permettant d'assister un expert métier ou un architecte logiciel lors de séances de *refactoring* de modèles.
- Des algorithmes de restructuration de modèles endogènes sont aussi envisageables.

Ce travail concerne donc la définition, la modélisation et l'outillage de processus. L'exploitation et l'intégration de patrons au sein des processus constituent un prolongement au projet NEPTUNE (Nice Environment with a Process and Tools using Norms - UML, XML and XMI - and Example). L'étude tend à répondre à l'exigence visant à maîtriser le processus de développement, comme proposé dans [H. Leblanc, *et al.*, 2005] et, d'une manière générale, rejoint les préoccupations de la communauté émergente IDM (Ingénierie Dirigée par les Modèles) qui vise à conférer aux modèles un caractère productif. La *Figure 1* présente

l'architecture générale de la démarche NEPTUNE. Lors des phases de conception de l'architecture et de conception par objets, dans le couloir d'activité relatif à l'expertise, les activités *i* et *iii* constituent notre contexte général d'étude. Les expérimentations s'appuieront sur des modèles UML vus comme des données, des requêtes ou contraintes OCL (Object Constraint Language) pour vérifier la pertinence et la qualité d'une transformation, et sur un langage de transformation de modèles conforme aux préconisations QVT (Queries / Views / Transformations) de l'OMG (Object Management Group).

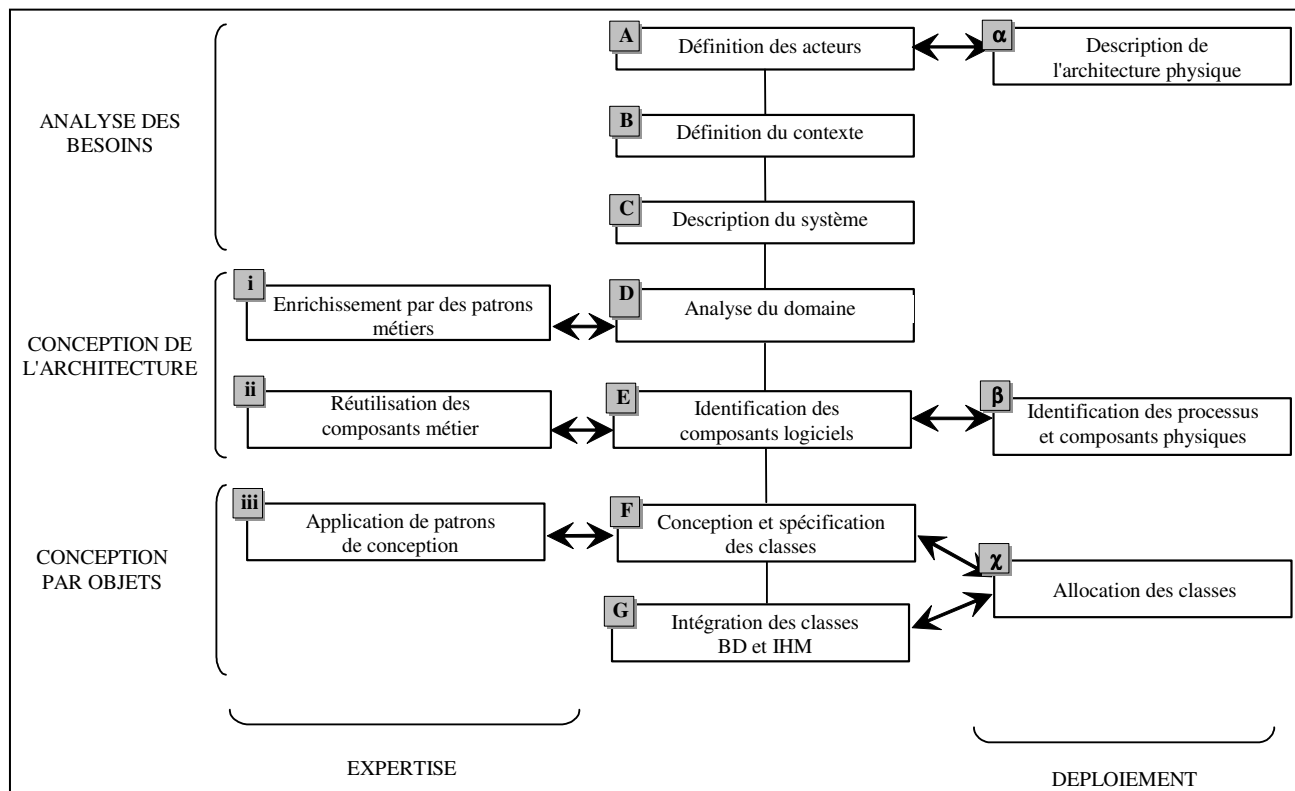


Figure 1 : La démarche NEPTUNE

Pour faciliter la tâche des concepteurs, il nous semble utile de proposer, en amont à toutes ces transformations, une méthode semi-automatique permettant de déterminer des fragments de modèles dans lesquels pourraient être injectés différents types de patrons. La méthode devra suggérer, de manière ciblée, un ensemble de patrons que le concepteur pourra choisir librement d'intégrer. L'intégration sera prise en charge par une transformation automatique. Nous allons nous attacher, dans une première étude, à la recherche des conditions d'application des patrons de conception du catalogue du GOF. Les patrons de conception ont l'avantage d'être connus, acceptés par la communauté, en nombre limité, indépendant de tout domaine applicatif et suffisamment concis.

2.2) Contexte particulier de l'étude

Dans ce contexte général, cette première étude consistait à formaliser, à l'aide d'un ensemble collaboratif de règles OCL, les éventuels points d'ancrage à l'injection d'un ou plusieurs patrons de conception dans une modélisation UML donnée. L'application de patron de conception se décompose en plusieurs étapes (cf. Figure 2). Cette étude se situe après l'étape de préparation des modèles, et ne prend pas en compte la transformation permettant

l'intégration de patrons. La finalité consiste à avoir à notre disposition un ensemble de règles OCL permettant une liste de patrons de conception à intégrer dans une modélisation.

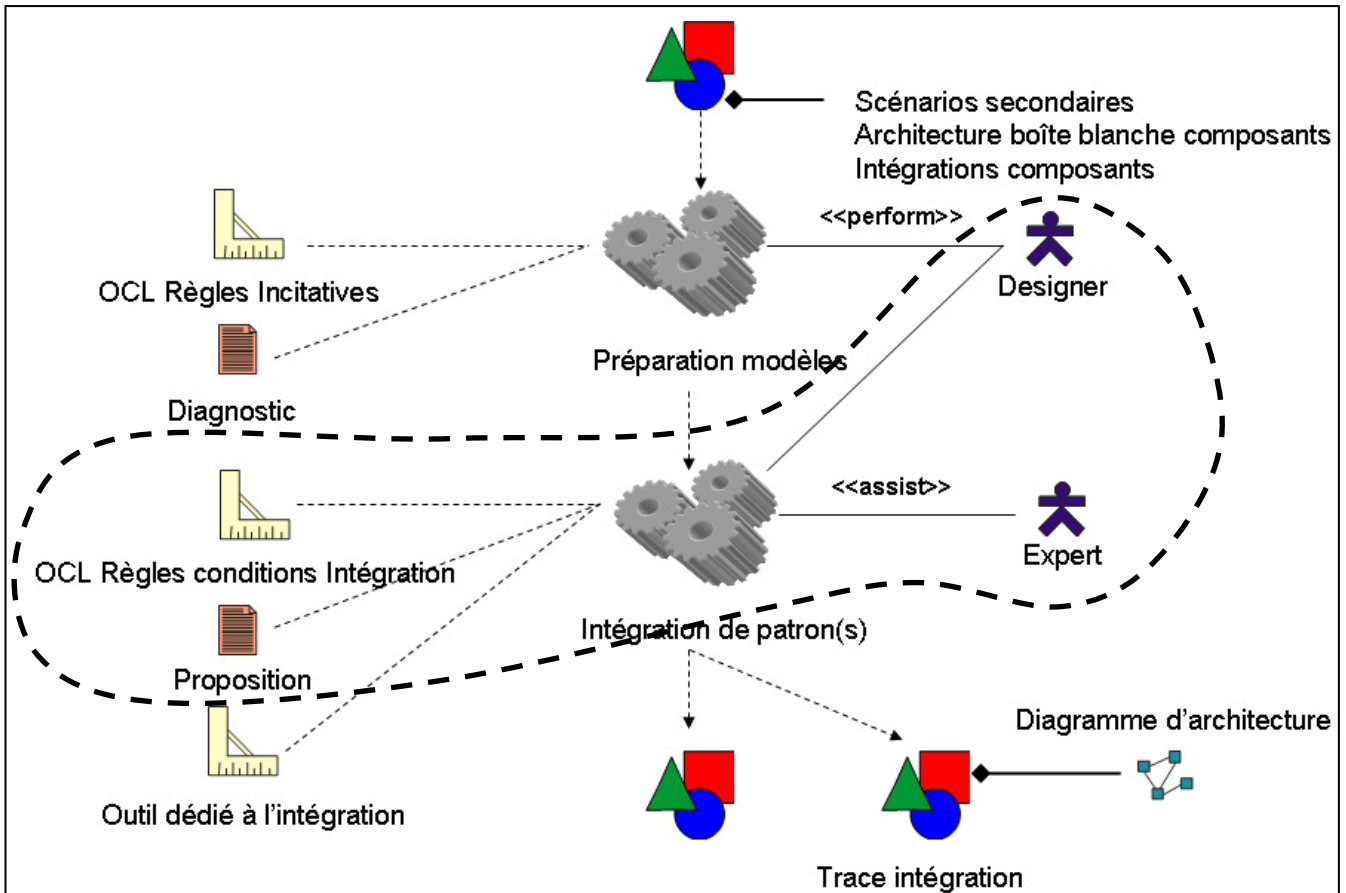


Figure 2 : Diagramme d'activité : Application de patrons de conceptions

Les patrons de conception représentent un savoir-faire sous forme de microarchitecture de classes réutilisables. Les patrons de conception comportementaux nécessitent souvent une vue dynamique du système à modéliser, or les modèles industriels à notre disposition sont uniquement constitués de diagrammes statiques. Les patrons créateurs étendent en général les possibilités des patrons déjà existants. Nous avons donc décidé de limiter les champs de l'étude aux patrons structurels. Ils permettent une conception simple en termes de structure (gestion des liens d'héritage et d'association) et élégante en termes de responsabilité entre classes.

Nous avons donc cherché à définir les conditions d'application des patrons de conception structurels. Pour cela, nous avons dû, pour chaque patron structurel identifié par le GOF, trouver un ensemble de particularités qui, une fois repérées dans un modèle, cibleraient le fragment de modèle à transformer par l'injection d'un patron structurel. La recherche de ces particularités doit permettre d'insérer des patrons de conception à bon escient et au bon endroit, dans des modèles UML.

Pour définir ces conditions d'application, nous avons commencé par chercher des modèles alternatifs à chaque patron. Un modèle alternatif à un patron est un modèle qui résout le même problème que le patron, mais dont la structure n'intègre pas la microarchitecture du patron. Il s'agit donc d'un modèle candidat à la substitution par un patron. Après avoir obtenu ces alternatives, nous avons cherché les particularités remarquables permettant de les détecter dans tout modèle. Ensuite, nous avons automatisé cette détection à l'aide de règles OCL supportées par la plate-forme NEPTUNE. Enfin, nous avons testé ces règles sur des modèles industriels et des méta-modèles standards.

1. RECHERCHE DE MODELES ALTERNATIFS

1.1) Une expérience

Il existe plusieurs possibilités pour identifier des modèles alternatifs.

La première est d'analyser la structure des patrons eux-mêmes, et d'effectuer des transformations sur leur structure : classes et collaborations afin de les dénaturer. Les modifications effectuées doivent cependant maintenir une cohérence minimum, de manière à ce que les modèles obtenus résolvent le même problème qu'avant la transformation. Il s'agit donc de modifications sans altération du comportement. Si cette solution peut permettre de lister de façon exhaustive l'ensemble des modèles alternatifs, elle risque de provoquer une explosion combinatoire des cas possibles, parmi lesquels beaucoup d'entre eux n'auraient qu'un intérêt limité. En effet, trop distants d'une conception classique ou trop artificiels, ils ne se retrouveraient pas dans des modèles courants.

La seconde possibilité est de travailler sur un ensemble de modèles existants, résolvant un problème soluble par l'emploi d'un patron de conception, sans utiliser ce patron. Pour obtenir cet ensemble de modèles, nous avons proposé à deux promotions d'élèves en IUP ISI (ingénierie des systèmes informatiques) de réaliser une expérience (38 élèves de L3 juste avant leur premier cours sur les patrons, et 28 élèves de M2, n'ayant pas eu, dans leur cursus, de cours sur les patrons). Ils ont eu à modéliser, dans la notation UML, une série de sept problèmes types, solubles avec les sept patrons structurels. Les étudiants allaient ainsi pouvoir modéliser les problèmes selon leur propre façon de penser. Pour que le résultat obtenu soit assimilable à une alternative valide aux patrons de conception, les problèmes posés devaient être très précis, et soluble en l'état par un patron structurel. Pour cela, nous avons choisi d'utiliser, lorsqu'ils nous semblaient pertinents, les exemples présentés dans la section motivations des patrons du GOF, desquels nous avons déduit des problèmes adaptés pour l'expérience. Chaque problème admettait ainsi une solution utilisant un patron structurel, mais les étudiants ont résolu ces problèmes sans connaissance à priori sur les patrons.

1.2) Problème posé aux étudiants

Le texte ci-dessous présente l'énoncé qui a été distribué aux étudiants. Ils avaient deux heures en travaux dirigés pour en faire le maximum, et ils pouvaient le terminer chez eux, s'ils le souhaitaient. Dans l'ensemble, la majorité des étudiants a modélisé tous les problèmes.

Ce document propose un ensemble d'exercices de modélisation de problèmes courants. Pour chacun de ces exercices, vous devez produire un diagramme de classe présentant une solution au problème posé. Chaque diagramme doit contenir suffisamment d'informations (attributs, méthodes, relations, stéréotypes...) pour pouvoir être interprété en l'état.

Ne cherchez pas de solutions sur Internet ou dans des ouvrages de conception, car le but de l'exercice est que vous suiviez un raisonnement qui vous est propre. Ne cherchez pas non plus à uniformiser vos diagrammes avec vos camarades, ces problèmes peuvent se résoudre de plusieurs manières différentes.

Certains problèmes ont des évolutions probables. Les diagrammes doivent être structurés de manière à ce que ces évolutions soient facilement intégrables. Faites apparaître ces évolutions dans les diagrammes ou dans un diagramme supplémentaire.

Problème 1 :

Modéliser un système permettant de dessiner un graphique : un graphique est composé de lignes, de rectangles, de textes et d'images. Une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.

Problème 2 :

Modéliser un système permettant d'afficher des fenêtres à l'écran : le style de ces fenêtres dépend de la plate-forme d'utilisation. Deux plates-formes sont considérées, XWindow et PresentationManager. Le code client doit pouvoir être écrit indépendamment et sans connaissance a priori de la future plate-forme d'exécution. Il est probable que le système évolue pour que l'on puisse en plus, spécialiser ces fenêtres en fenêtres applicatives et en fenêtres iconisées.

Problème 3 :

Modéliser un système permettant d'afficher des objets visuels à l'écran : un objet visuel peut être un texte ou une image. Le système doit permettre d'ajouter à ces objets une barre de défilement verticale, une barre de défilement horizontale, et une bordure. Il est probable que le système évolue pour que l'on puisse ajouter aux objets visuels une barre de menu.

Problème 4 :

Modéliser un éditeur de dessins : un dessin est composé de lignes, de rectangles et de raclettes, placés à des positions précises. Une raclette est une forme complexe qu'une classe boîte-noire dessine. Cette classe, qui est fournie, effectue ce dessin en mémoire, et le met à disposition grâce à une méthode getRaclette(). Il est probable que le système évolue pour que l'on puisse en plus, dessiner des cercles. * dont on ne dispose pas du code, mais dont on connaît l'interface.*

Problème 5 :

Modéliser le moteur d'un jeu de l'oie pour qu'il soit utilisable par une interface graphique : l'interface graphique aura uniquement besoin de connaître la référence à une partie ainsi qu'à un plateau de jeu mais ne se souciera pas des cases, du dé et des pions, qui sont gérés par le moteur. Cette modélisation doit faire ressortir essentiellement une structure facilitant l'utilisation du moteur par l'interface. Le contenu du moteur de jeu est de moindre importance.

Problème 6 :

Modéliser la fonction d'affichage d'un éditeur de textes : un document est composé de lignes, de colonnes, et de caractères. La valeur intrinsèque d'un caractère est unique et dépend du contexte d'utilisation. L'intersection d'une ligne et d'une colonne donne l'emplacement où doit s'afficher la représentation de la valeur d'un caractère ainsi que son style de mise en forme. Il est important

d'éviter la prolifération d'objets ayant le même état. Il est probable que le système évolue pour que l'on puisse ajouter un autre jeu de caractères à un texte.

Problème 7 :

Modéliser le chargement et l'affichage d'un éditeur de documents : un document est composé de textes et d'images. Les contraintes de cet éditeur sont que l'ouverture d'un document soit la plus rapide possible, quelle que soit la taille du document, et que cette taille soit visible dès l'ouverture. Au niveau physique, un document est stocké sur un disque, de même que les images. Le contenu, et donc la taille des images peuvent changer entre deux ouvertures du document. Il est probable que le système évolue pour que l'on puisse ajouter des fichiers vidéo.

1.3) Sélection des modèles alternatifs

Sur les trois cent modèles obtenus, nous en avons sélectionné quarante qui constituaient des alternatives valides aux patrons. Nous avons effectué cette sélection en retirant les modèles qui ne résolvaient pas le problème posé. Nous avons éliminé des quarante modélisations tous les doublons, pour en obtenir onze qui présentaient des différences structurelles significatives par rapport aux patrons. Nous avons considéré ces modèles comme valides car ils permettaient de résoudre le problème posé.

Chaque modélisation obtenue (entre deux et cinq par patron) constituait donc une alternative plausible à un patron. Il fallait maintenant que nous détections les particularités de chacune d'entre-elles pour pouvoir les repérer dans d'autres modélisations. Pour cela, nous avons analysé leurs particularités structurelles : sous-arbres équivalents dans des hiérarchies d'héritage, répartition quantitative des associations et des compositions...

2. DEDUCTION DES PARTICULARITES REMARQUABLES

Chaque patron de conception structurel se caractérise par un rôle attribué à chacune des classes qui le compose. Chacun de ces rôles a une particularité structurelle propre. Ainsi, le patron *Composite* a trois rôles :
- « Composant » joué par une seule classe ayant comme particularité structurelle, d'être racine du graphe d'héritage et agrégée à l'une de ses filles,
- « Composite » joué par la classe fille et agrégat du « Composant »
- « Feuille » joué par les autres classes filles du « Composant ».

Chaque modèle alternatif se caractérise de la même façon qu'un patron. A chaque rôle du modèle alternatif est associé un ensemble de particularités structurelles dites remarquables. Celles-ci permettent de repérer dans un modèle, les classes pouvant jouer les différents rôles du patron. En associant, à chaque classe des modèles alternatifs, les rôles des patrons correspondant, et en étudiant leur structure, nous avons pu déduire ces particularités.

Si nous prenons l'exemple du rôle « Composant » du patron *Composite*, certaines particularités remarquables déduites des différents modèles alternatifs sont :

- Peuvent porter le rôle de « Composant » :
- Une classe ayant au moins deux agrégations
 - Une classe ayant au moins deux agrégations sans aucune réflexive
 - Une classe ayant des enfants et étant agrégée

Pour chaque modèle alternatif, il y a donc un ensemble de particularités remarquables associées à chacun des rôles du patron.

Pour détecter dans un modèle ces particularités, nous avons établi une méthode de recherche qui repère une classe pouvant jouer un rôle de référence sur le patron à identifier, puis, en utilisant cette classe, tente d'affecter les autres rôles du patron aux autres classes du modèle. Par exemple, pour le patron *Composite*, nous avons cherché les classes pouvant jouer le rôle de « Composant », puis à partir de ces classes, nous avons tenté d'affecter les rôles de « Composite » et de « Feuille » aux autres classes liées et dont la disposition correspondait à la particularité remarquable que nous souhaitions repérer. Cette méthode nous a permis d'en déduire l'algorithme de recherche générique donné ci-dessous.

Entrée : modèle UML ET ensemble de particularités remarquables liées à chaque rôle du modèle alternatif.

Chercher les classes candidates du modèle satisfaisant les particularités remarquables du rôle de référence.

Pour chaque classe candidate Faire

Chercher les classes liées dans le modèle

Pour chaque classe liée

Si elle satisfait les particularités remarquables d'un autre rôle

Alors

affecter le rôle à cette classe

Fin Si

Fin Pour

Fin Pour

Sortie : ensembles de classes pouvant porter chaque rôle du modèle alternatif

Les particularités remarquables identifiées concernent uniquement la structure des classes. Pour le moment, ni l'interface ni la sémantique des classes n'est entrée en considération. En effet, pour intégrer des notions de sémantique, un oracle est nécessaire, ce qui dépassait le cadre de mon stage. Nous nous sommes donc bornés aux aspects structurels des modèles. Cette limitation impose, pour l'instant, un contrôle supplémentaire de la part des concepteurs. En effet, si nous prenons un modèle alternatif du *Composite* en exemple, les particularités remarquables sont centrées sur les agrégations des classes entre-elles. Si les classes agrégées n'ont pas la même interface, voire même sémantique, il n'y a pas lieu d'appliquer le patron *Composite*, vérification que doit effectuer le concepteur après la détection. De plus, dans les modèles à analyser, nous avons considéré les compositions et les agrégations de la même manière, en raison de la proximité de leur sémantique, et parce que nous ne cherchions qu'à détecter la notion de lien entre les objets, plutôt que la manière de les composer (particularité qui n'est plus gérée dans la nouvelle norme UML2).

Les parties suivantes présentent la solution de référence incluant le patron, et chaque solution alternative proposée par les étudiants. Nous avons rajouté sur les diagrammes des étudiants les rôles des patrons que nous cherchons à détecter. De plus, pour chacun des modèles alternatifs, les données de l'algorithme de détection sont précisées.

2.1) Problème soluble par le Composite

Le premier problème de l'énoncé était soluble directement avec le patron *Composite*. Ce paragraphe présente la solution utilisant le modèle de référence (cf. *Figure 3*), puis tous les modèles alternatifs que nous avons sélectionnés. Pour chaque modèle alternatif, nous ferons une analyse mettant en valeur les lacunes qui auraient été évitées en utilisant un patron.

2.1.1) Solution avec le patron

Modéliser un système permettant de dessiner un graphique : un graphique est composé de lignes, de rectangles, de textes et d'images. Une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.

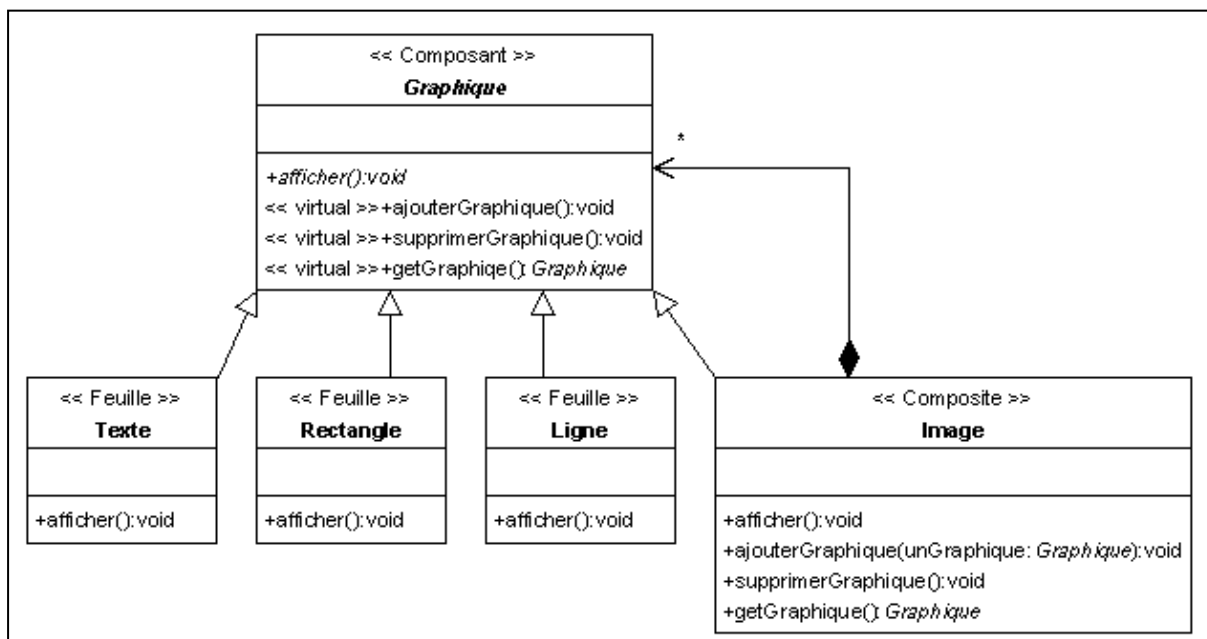


Figure 3 : Modèle de référence du patron Composite

Ce problème est typique du besoin de composer des objets complexes, de manière récursive. Ce genre de composition est extrêmement fréquent, et peut se retrouver dans beaucoup de modélisations, dès qu'il y a une notion de hiérarchie de composition entre les objets. Ici, les textes, rectangles et lignes sont les feuilles de l'arbre et les images en sont les nœuds, y compris la racine. La virtualité des méthodes permet au client de ne travailler qu'avec la classe *Graphique*.

2.1.2) Modèle alternatif n°1 : développement de la composition

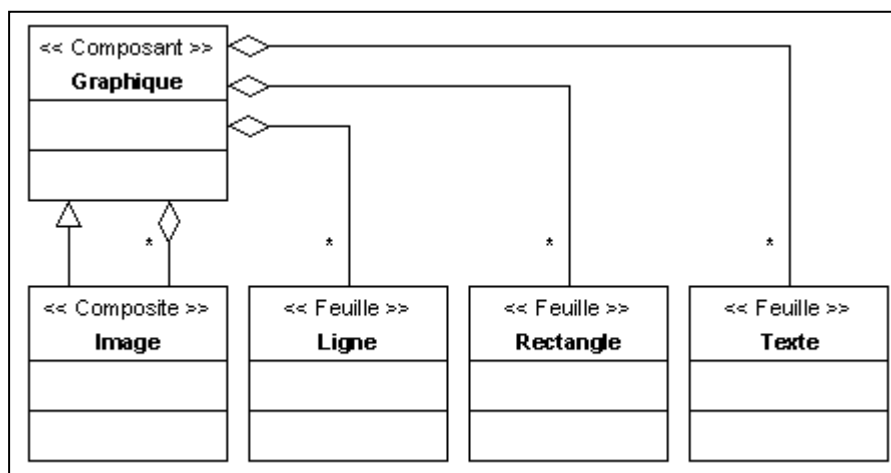


Figure 4 : Modèle alternatif n°1 du Composite

Dans ce cas, l'étudiant a exprimé la composition en utilisant directement la notion d'agrégation. La classe graphique joue un rôle de container d'objets, et on peut imaginer que ce container est utilisé directement par le client. De plus, la classe *Image* joue, elle aussi, un rôle de container, exprimé par la spécialisation de *Graphique*. Cette solution est valide, puisque la composition récursive est possible, même si au codage, la classe *Graphique* sera contrainte de stocker toutes les informations de composition. Ceci provoquera la multiplication des itérateurs sur tous les éléments de la hiérarchie. Si l'étudiant avait utilisé le patron de conception, *Graphique* aurait joué le rôle de « Composant », *Image* celui de « Composite » et *Ligne*, *Rectangle* et *Texte* le rôle de « Feuille ». L'utilisation du patron aurait ainsi évité la redondance des liens d'agrégation, simplifiant la structure et le codage.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Composant »

Particularité du rôle « Composant » : Classe ayant au moins deux agrégations

Particularité du rôle « Composite » : Classe étant à la fois fille et agrégée au « Composant »

Particularité du rôle « Feuille » : Classe agrégée au « Composant » et qui n'est pas fille

2.1.3) Modèle alternatif n°2 : développement complet de la composition

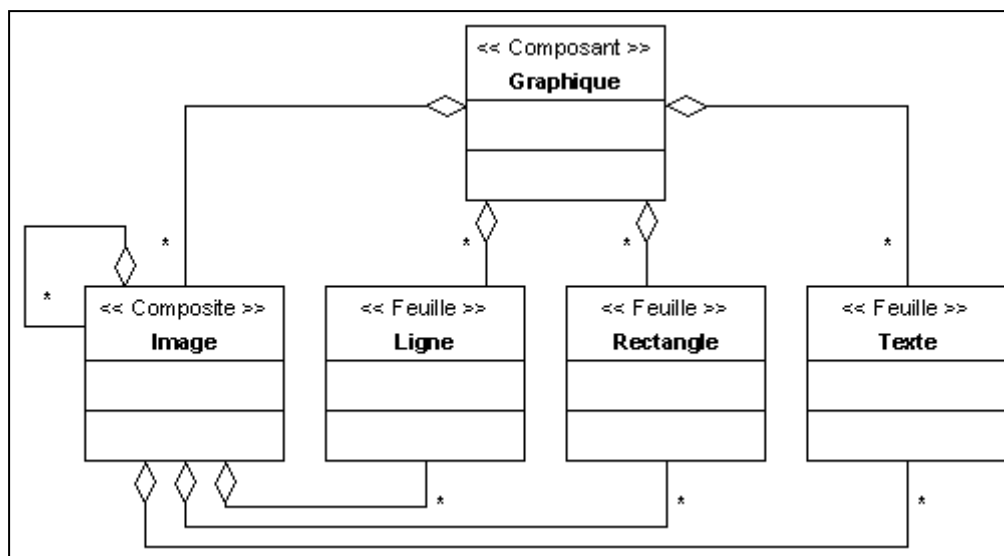


Figure 5 : Modèle alternatif n°2 du Composite

Cet étudiant a reproduit la structure du patron *Composite*, en oubliant toute notion de factorisation. Il apparaît clairement que *Graphique* contient toutes les autres classes, et *Image* fait de même, en se contenant elle-même. Cette solution est valide, même si le manque de factorisation sera à l'origine d'un code très lourd.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Composant »

Particularité du rôle « Composant » : Classe ayant au moins deux agrégations sans aucune agrégation réflexive

Particularité du rôle « Composite » : Classe agrégée au « Composant » et agrégation d'autres classes, y compris d'elle-même

Particularité du rôle « Feuille » : Classe agrégée au « Composant » et sans aucune agrégation réflexive

2.1.4) Modèle alternatif n°3 : la classe intermédiaire

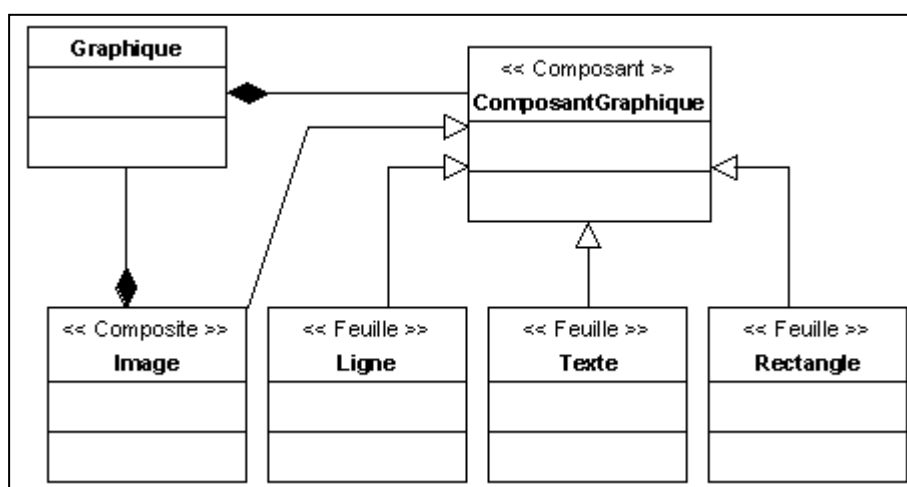


Figure 6 : Modèle alternatif n°3 du Composite

Ici, la proximité avec le patron *Composite* est surprenante, même si l'étudiant a complexifié la composition en ajoutant une classe intermédiaire. Les conséquences sur le codage sont minimales, même si la présence superflue de la classe *Graphique* risque de dupliquer le code de la classe *ComposantGraphique*.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Composant »

Particularité du rôle « Composant » : Classe ayant des enfants et étant agrégée

Particularité du rôle « Composite » : Classe étant à la fois fille du « Composant » et agrégation d'autres classes

Particularité du rôle « Feuille » : Classe fille du « Composant » et qui n'est pas agrégation d'autres classes

2.1.5) Modèle alternatif n°4 : développement de la composition sur le composite

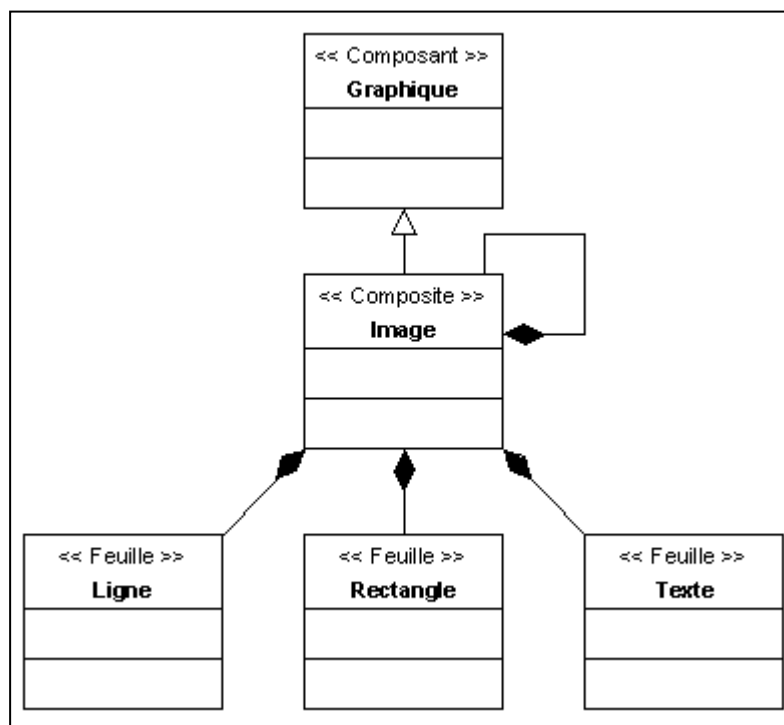


Figure 7 : Modèle alternatif n°4 du Composite

Cet étudiant a exprimé la composition telle qu'elle est décrite dans l'énoncé. On repère bien qu'une image est composée d'autres images qui peuvent être composées de lignes, de rectangles ou de textes. La généralisation de la classe *Image* n'est là que pour servir de point d'accès au client. Cependant, le fait que les classes *Ligne*, *Rectangle* et *Texte* n'héritent pas de *Graphique* provoquera des duplications de code.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Composite »

Particularité du rôle « Composite » : Classe ayant au moins deux agrégations, dont une au moins réflexive et ayant une super-classe

Particularité du rôle « Composant » : Classe mère du « Composite »

Particularité du rôle « Feuille » : Classe agrégée au « Composite »

2.1.6) Modèle alternatif n°5 : agrégation récursive

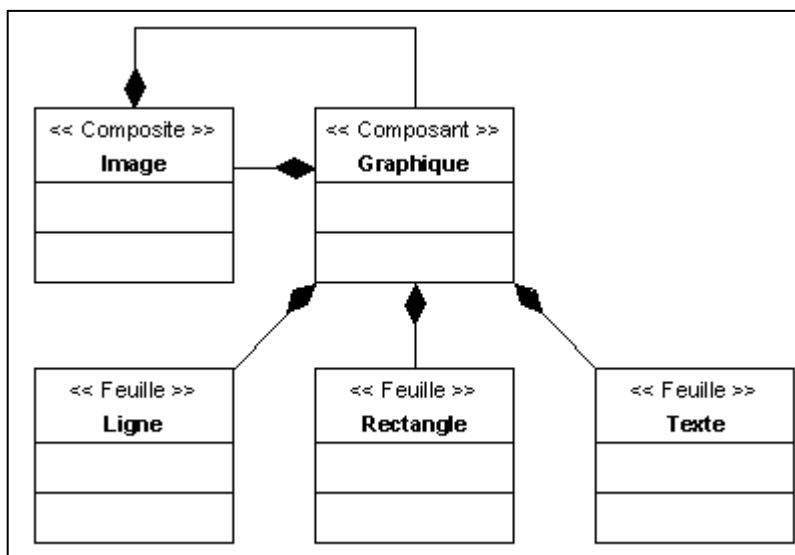


Figure 8 : Modèle alternatif n°5 du Composite

Ce dernier modèle ressemble fortement au deuxième, mais la classe *Image* n'est pas composée d'elle-même mais de la classe *Graphique*. Le code sera ainsi moins lourd que pour le modèle 2, mais là encore, le manque de généralisation se traduira par une duplication de code.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Compositant »

Particularité du rôle « Compositant » : Classe ayant au moins deux agrégations

Particularité du rôle « Composite » : Classe étant à la fois agrégées et agrégation du « Compositant »

Particularité du rôle « Feuille » : Classe agrégée au « Compositant » mais non agrégation du « Compositant »

2.1.7) Synthèse sur le patron Composite

Ce patron est celui qui a le plus de modèle alternatifs. Les étudiants ont eu des idées intéressantes pour composer les objets entre eux, les particularités les plus récurrentes restant logiquement les agrégations. L'erreur la plus fréquente est donc un manque de factorisation de la relation d'agrégation, ainsi que la perte de l'interface commune des « Compositant », les étudiants n'ayant pas suffisamment utilisé les liens d'héritage. Cela dit, il est probable que s'ils avaient eu à coder leur modélisation, la duplication du code les aurait incités à revenir sur leur modélisation et à factoriser davantage.

La diversité des modèles alternatifs laissent à penser que les détections du patron *Composite* vont se multiplier. Cependant, la sémantique risque de jouer beaucoup en notre défaveur. En effet, un ensemble d'agrégation dans un modèle n'a pas forcément le même sens sémantique que celui du modèle alternatif correspondant. A l'heure actuelle, le concepteur doit donc effectuer un contrôle minutieux des résultats de l'algorithme pour s'assurer que l'utilisation du patron *Composite* a un sens dans son modèle.

2.2) Problème soluble par le Pont

Le deuxième problème de l'énoncé était soluble directement avec le patron *Pont*. Ce paragraphe présente la solution utilisant le modèle de référence (cf. *Figure 10*), puis tous les modèles alternatifs que nous avons sélectionnés. Pour chaque modèle alternatif, nous ferons une analyse mettant en valeur les lacunes qui auraient été évitées en utilisant un patron.

2.2.1) Solution avec le patron

Voici à nouveau l'énoncé du problème donné aux étudiants :

Modéliser un système permettant d'afficher des fenêtres à l'écran : le style de ces fenêtres dépend de la plate-forme d'utilisation. Deux plates-formes sont considérées, XWindow et PresentationManager. Le code client doit pouvoir être écrit indépendamment et sans connaissance a priori de la future plate-forme d'exécution. Il est probable que le système évolue pour que l'on puisse en plus, spécialiser ces fenêtres en fenêtres applicatives et en fenêtres iconisées.

Ce problème met en avant le besoin de séparer l'abstraction de l'implémentation. Il faut modéliser un système d'affichage de fenêtre dont le style dépend de la plate-forme utilisée. Spécialiser chaque fenêtre par les plates-formes disponibles pose de gros problèmes lors des évolutions du système. En effet, si chaque type de fenêtre a une sous-classe par plate-forme, l'ajout d'un nouveau type impose l'ajout d'autant de sous-classes que de plate-forme. De même, l'ajout d'une nouvelle plate-forme implique l'ajout d'une nouvelle fille à chaque type de fenêtre. La Figure 9 illustre cette multiplication de classes. Le modèle de gauche spécialise la classe *Fenêtre* à chaque plate-forme (PM : Présentation Manager et X : XWindows). Le passage au modèle de droite correspond à l'ajout d'un nouveau type de fenêtre (Fenêtre avec une icône). Pour modéliser cette extension, il faut ajouter à la classe *FenetreIcône* une classe par plate-forme considérée.

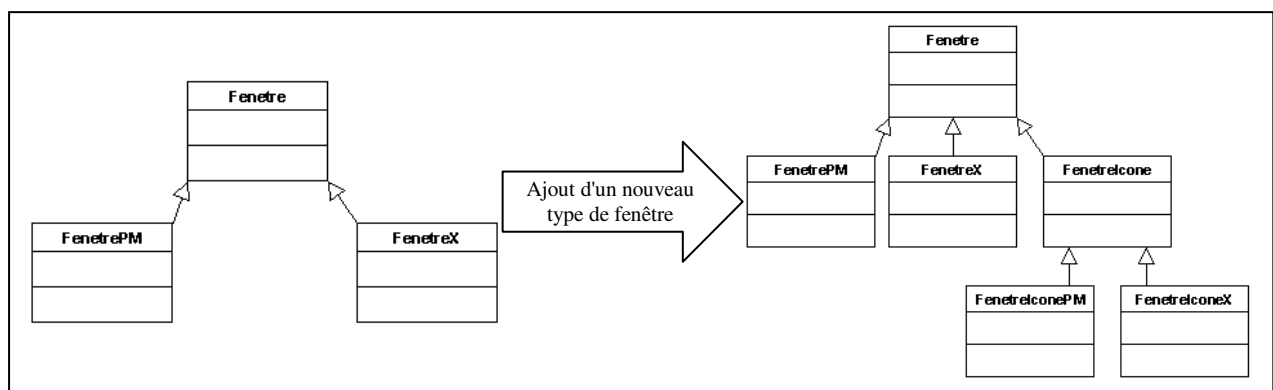


Figure 9 : Illustration de la multiplication des classes lorsque le patron Pont n'est pas utilisé

Pour éviter cette multiplication, la solution consiste à séparer l'abstraction de l'implémentation. Ainsi, d'un côté, sont placés les différents types de fenêtre qui savent placer leur composant (icône, bouton, menu...), et de l'autre, les différentes plates-formes, qui dessinent un composant à leur manière. Ainsi, l'ajout d'une plate-forme ne nécessite aucun changement du côté des fenêtres, et l'ajout d'une fenêtre, n'impose que l'ajout de l'implémentation nécessaire.

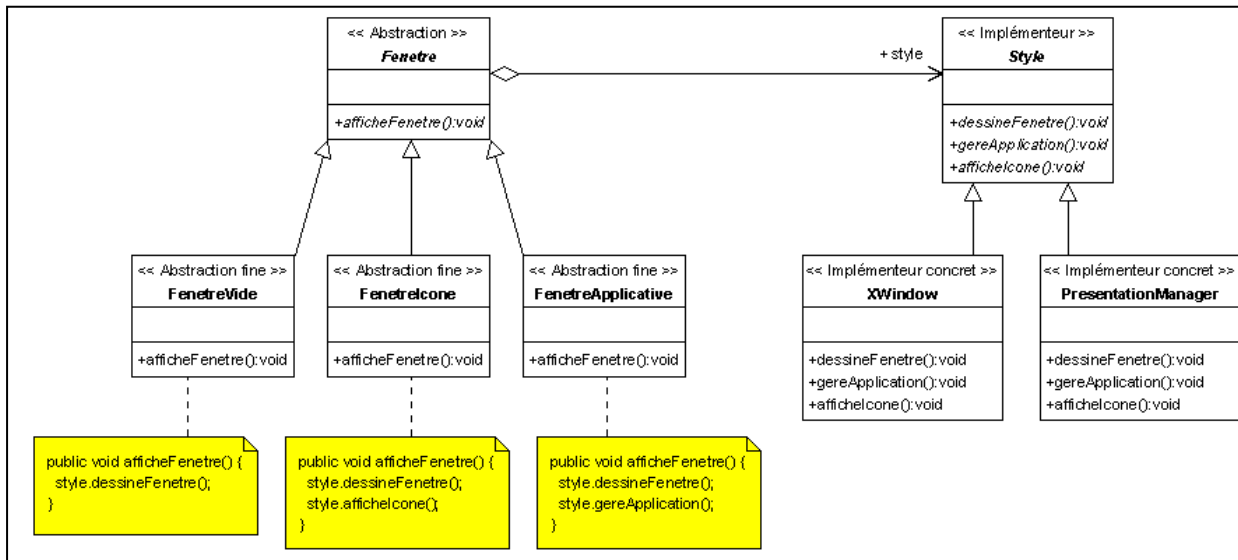


Figure 10 : Modèle de référence du patron Pont

2.2.2) Modèle alternatif n°1 : développement de la relation d'utilisation

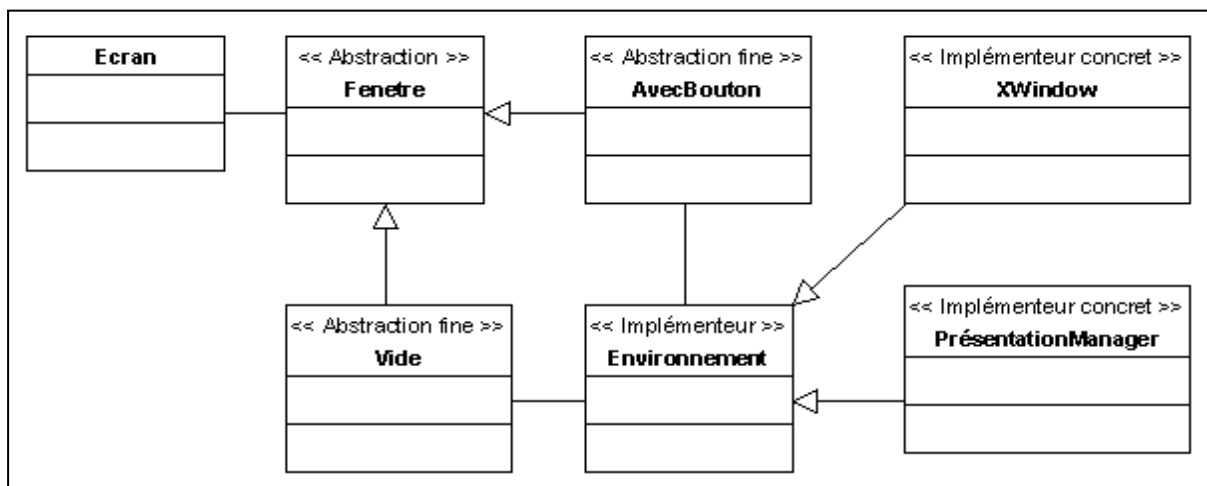


Figure 11 : Modèle alternatif n°1 du Pont

Ici, l'étudiant a bien séparé les types de fenêtre des plates-formes, mais il n'a pas factorisé les associations entre « Implémenteur » et « Abstraction fine ». Ainsi, l'ajout d'une plate-forme ne génère pas de problème particulier, en revanche, l'ajout d'un type de fenêtre supplémentaire créera un nouveau lien d'association vers la classe *Environnement*.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Abstraction fine »

Particularité du rôle « Abstraction fine » : Classes ayant le même parent et une extrémité d'association commune

Particularité du rôle « Abstraction » : Classe mère des « Abstraction fine »

Particularité du rôle « Implémenteur » : Classe associée aux « Abstraction fine »

Particularité du rôle « Implémenteur concret » : Classe fille de « Implémenteur »

2.2.3) Modèle alternatif n°2 : développement des « Implémenteurs concrets »

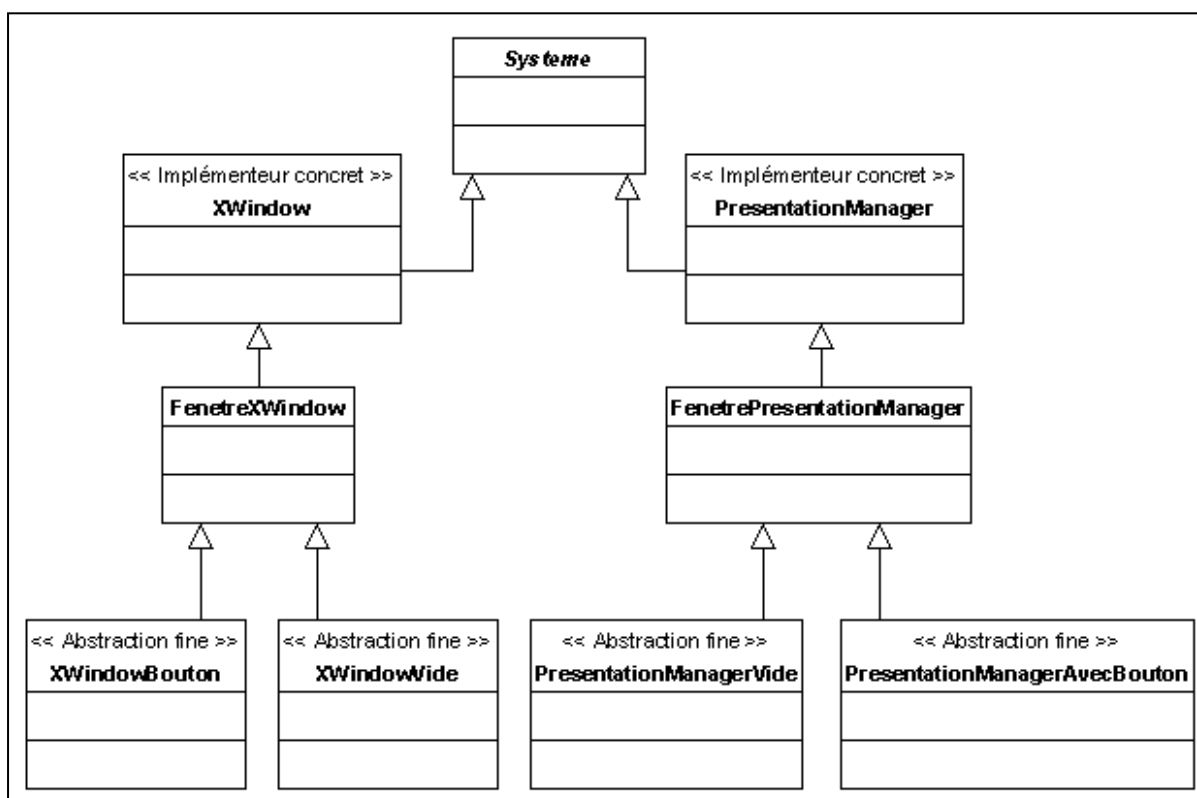


Figure 12 : Modèle alternatif n°2 du Pont

Cet exemple est le cas typique de mauvaise conception que la patron résoud, car il multiplie le nombre de classes « Abstraction fine » par le nombre de classes « Implémenteur concret », à l'origine de la création du patron *Pont*. Cependant, cette solution reste juste. Ici, il est très difficile d'établir les particularités remarquables liées à chaque rôle, car structurellement, il s'agit d'une hiérarchie d'héritage, un oracle étant nécessaire pour identifier chacun des rôles. Si les classes de chaque côté de la classe *Système* n'ont pas la même sémantique, il n'y a pas à appliquer le patron *Pont*.

De ce cas, nous n'avons donc pas pu tirer de particularités remarquables :

Rôle de départ : aucun

Particularité du rôle « Abstraction fine » : Indétectable ici

Particularité du rôle « Abstraction » : Indétectable ici

Particularité du rôle « Implémenteur » : Indétectable ici

Particularité du rôle « Implémenteur concret » : Indétectable ici

L'algorithme générique n'étant pas applicable, pour le moment, sur ce modèle alternatif, nous l'avons substitué par un algorithme de recherche de sous-arbres symétriques ayant une racine commune. Dans ce cas, faute d'oracle, l'analyse de la structure doit être impérativement validée au niveau sémantique par un concepteur.

2.2.4) Synthèse sur le patron Pont

Beaucoup d'étudiant ont modélisé leurs solutions de la même manière que le modèle alternatif n°1, ou alors ils n'ont pas fait apparaître les plates-formes comme des objets, mais simplement comme des paramètres. Ce dernier cas n'est pas exploitable pour le moment, car il implique un analyseur de code. Ceci est un véritable problème d'échelle industrielle, dépassant, pour le moment, la portée d'analyse des étudiants. Voilà donc ce qui explique le faible nombre de modèles alternatifs. De plus, l'alternative n°2 met en difficulté notre méthode de recherche générique, car nous ne travaillons, pour l'instant, qu'avec des particularités structurelles remarquables. La création de l'algorithme de recherche comparant les structures d'héritages permet tout de même de montrer que le modèle reste détectable, mais d'une autre manière.

A l'heure actuelle, la détection du patron *Pont* se fait donc de deux manière différents, mais dont le résultat reste le même, puisque les deux méthodes réussissent à affecter les rôles du patron aux classes des modèles alternatifs. Une limite reste présente pour la deuxième alternative. Sans oracle, la vérification de la détection par le concepteur est indispensable, pour garantir une sémantique cohérente.

2.3) Problème soluble par le Décorateur

Le troisième problème de l'énoncé était soluble directement avec le patron *Décorateur*. Ce paragraphe présente la solution utilisant le modèle de référence (cf. *Figure 13*), puis tous les modèles alternatifs que nous avons sélectionnés. Pour chaque modèle alternatif, nous ferons une analyse mettant en valeur les lacunes qui auraient été évitées en utilisant un patron.

2.3.1) Solution avec le patron

Voici l'énoncé du problème qui a été posé aux étudiants :

Modéliser un système permettant d'afficher des objets visuels à l'écran : un objet visuel peut être un texte ou une image. Le système doit permettre d'ajouter à ces objets une barre de défilement verticale, une barre de défilement horizontale, et une bordure. Il est probable que le système évolue pour que l'on puisse ajouter aux objets visuels une barre de menu.

Ce problème représente le besoin de composer des comportements entre eux, à l'exécution. Les textes ou les images constituent les feuilles, et les barres de défilement, les bordures et les barres de menu sont des nœuds. La particularité de ce problème est la notion de "décoration". Les nœuds ont pour but de décorer les feuilles, en ajoutant un comportement ou un état spécifique. Pour faciliter l'utilisation, la décoration se doit d'être composable (un texte peut avoir une barre de défilement, un menu et une bordure), sans multiplier les lignes de code, c'est-à-dire effectuer une décoration à la volée. Pour cela, il suffit de profiter de la structure du patron *Composite*. Par exemple, `new BarreVerticale(new BarreMenu(new Bordure(new Texte())))`, permet d'ajouter une bordure, une barre de menu et une barre de défilement à un texte en une seule ligne de code.

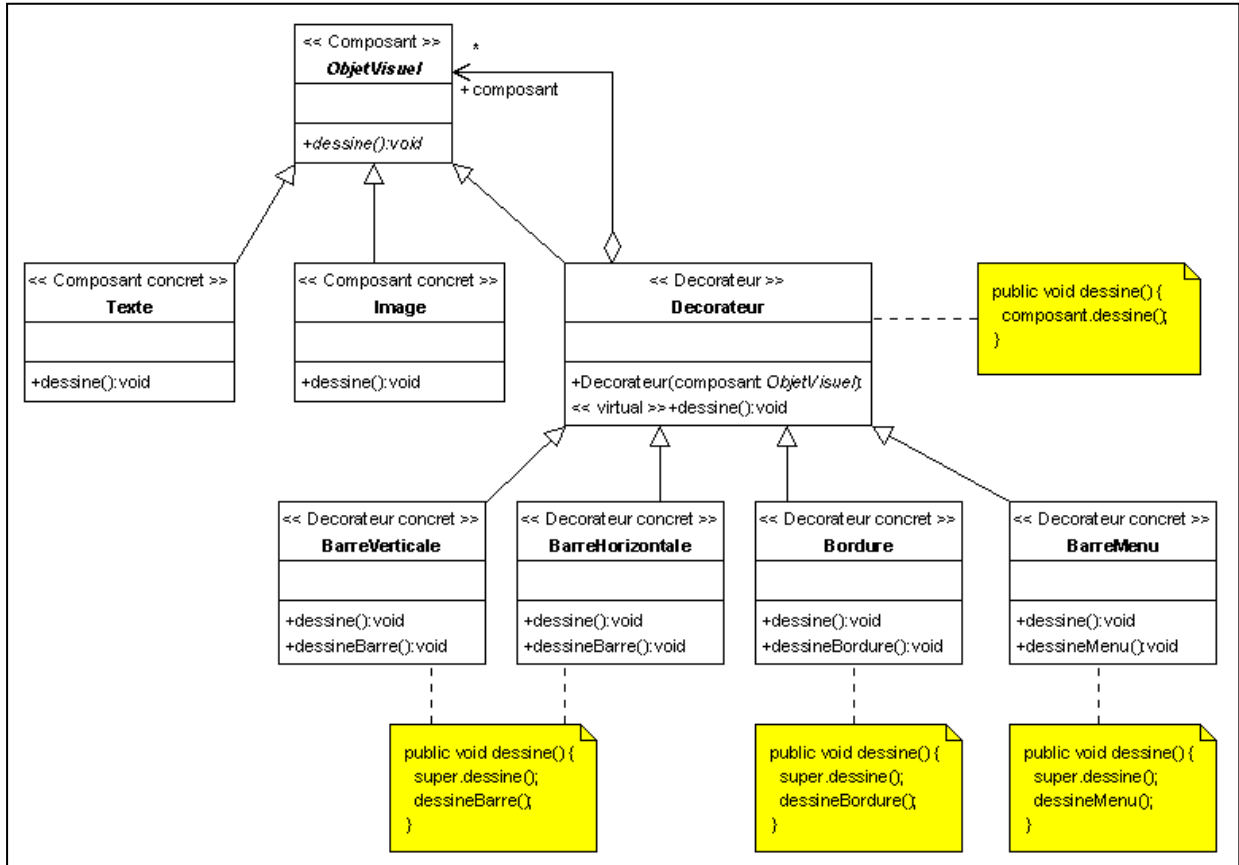


Figure 13 : Modèle de référence du Décorateur

2.3.2) Modèle alternatif n°1 : « Décorateurs » statiques greffés au « Composant »

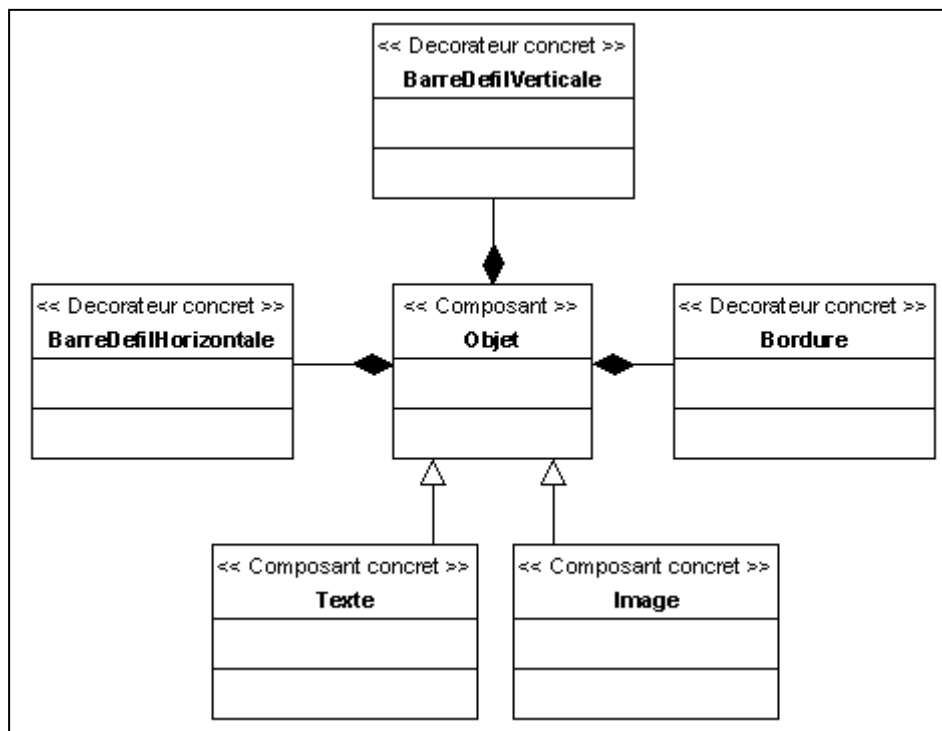


Figure 14 : Modèle alternatif n°1 du Décorateur

Dans ce cas, l'étudiant a exprimé en dur la décoration des différents objets graphiques sur la classe mère « Composit ». Si les comportements attendus sont a priori les mêmes, la structure de ce modèle ne permet pas la décoration à la volée, et nécessite autant de lignes de code que de composition de décorateurs, pour tout programme client.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Composit »

Particularité du rôle « Composit » : Classe étant à la fois agrégation et mère

Particularité du rôle « Composit concret » : Classe fille de « Composit »

Particularité du rôle « Décorateur » : Indétectable ici

Particularité du rôle « Décorateur concret » : Classe agrégée à « Composit »

Le « Décorateur » n'étant pas présent dans ce modèle alternatif, le concepteur devra combler ce manque en ajoutant une classe spécifique, mère des différents « Décorateur concret ». Le fait que la classe à ajouter soit simplement une abstraction, ne va pas poser de problème au concepteur. Il lui suffira de factoriser, si possible, les interfaces, pour en faire hériter toutes les classes identifiées comme « Décorateur concret ».

2.3.3) Modèle alternatif n°2 : « Décorateurs » statiques greffés au « Composite »

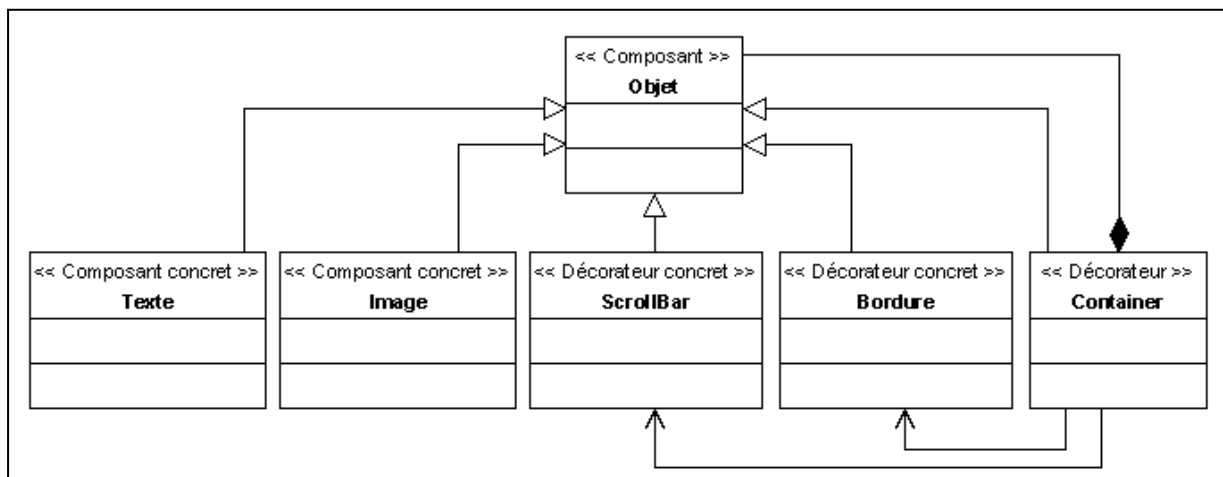


Figure 15 : Modèle alternatif n°2 du Décorateur

L'étudiant a reproduit ici le patron *Composite*, auquel il a ajouté des associations pour faciliter l'accès aux objets décorateurs. L'idée est intéressante, mais le nombre d'associations dépend du nombre d'objets décorateur, et comme pour le modèle précédent, la décoration à la volée n'est pas possible.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Composit »

Particularité du rôle « Composit » : Classe étant à la fois agrégée et mère

Particularité du rôle « Décorateur » : Classe à la fois fille et agrégation de « Composit »

Particularité du rôle « Décorateur concret » : Classe fille de « Composit » et associée au « Décorateur »

Particularité du rôle « Composit concret » : Classe fille de « Composit » qui n'est pas associé au « Décorateur »

2.3.4) Synthèse sur le patron Décorateur

La majorité des étudiants ont représenté la notion de décoration grâce à des agrégations. Hormis le problème de la décoration à la volée, ce système ne semble pas présenter de problème. Cependant, une question peut se poser pour savoir si syntaxiquement le modèle est correct. En effet, plusieurs classes agrégées à une autres peuvent-elles être instanciables toutes en même temps ? C'est-à-dire, en se fondant sur le modèle alternatif n°1, est-ce qu'un objet peut-avoir une bordure et une barre de défilement verticale en même temps ? Les multiplicités n'apparaissent pas sur le modèle, parce que l'étudiant ne les a pas mentionnées, mais peut-être que l'intégration des multiplicités dans la détection sera un passage obligé.

2.4) Problème soluble par l'Adaptateur

Le quatrième problème de l'énoncé était soluble directement avec le patron *Adaptateur*. Ce paragraphe présente la solution utilisant le modèle de référence (cf. Figure 16), puis le modèle alternatif que nous avons sélectionnés.

2.4.1) Solution avec le patron

Modéliser un éditeur de dessins : un dessin est composé de lignes, de rectangles et de raclettes, placés à des positions précises. Une raclette est une forme complexe qu'une classe boîte-noire dessine. Cette classe, qui est fournie, effectue ce dessin en mémoire, et le met à disposition grâce à une méthode getRaclette(). Il est probable que le système évolue pour que l'on puisse en plus, dessiner des cercles. * dont on ne dispose pas du code, mais dont on connaît l'interface.*

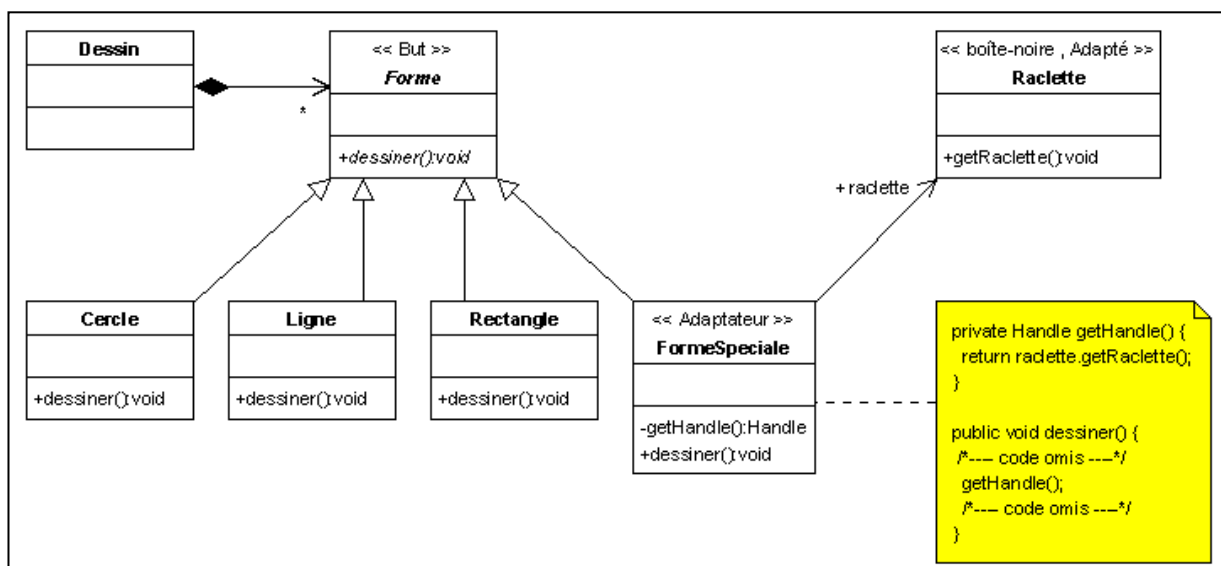


Figure 16 : Modèle de référence de l'Adaptateur

Ce problème présente le besoin d'adapter un développement existant à une nouvelle interface d'utilisation. L'éditeur de dessins peut dessiner différentes formes. Afin de faciliter l'ajout de nouvelles formes, une interface est définie : la classe abstraite *Forme*. Tout ce qui doit être dessiné dans l'éditeur doit hériter de cette classe. Ainsi, la classe *Dessin* ne manipule que des objets de type statique. L'énoncé impose d'utiliser une classe *Raclette*, mais son code étant inaccessible, puisque c'est une boîte noire, la solution consiste à utiliser une classe intermédiaire qui va adapter l'interface de *Raclette* à l'interface de *Forme*.

2.4.2) Modèle alternatif n°1 : adaptation par héritage multiple

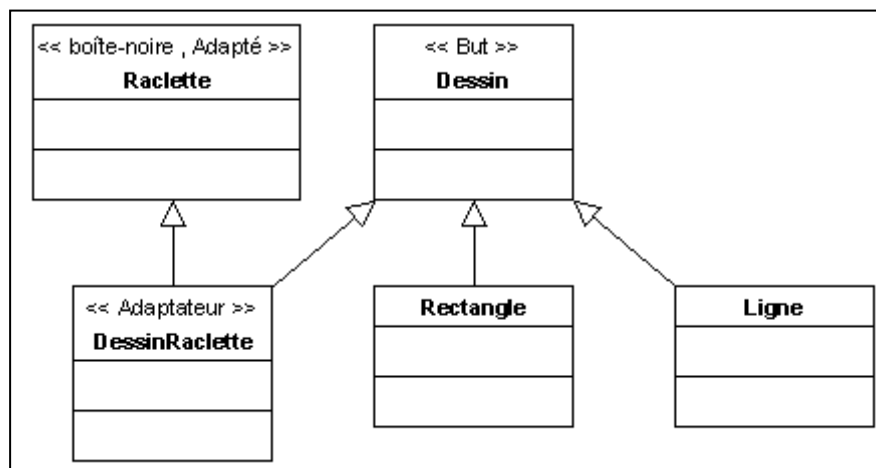


Figure 17 : Modèle alternatif n°1 pour l'Adaptateur

Les solutions proposées par les étudiants pour le patron *Adaptateur* n'ont pas été très convaincantes, car la notion de boîte noire n'a pas été comprise correctement. D'une part parce que certains étudiants l'ont faite hériter de la classe « But », et d'autre part parce que la notation UML ne prévoit rien pour spécifier cette contrainte, ce qui la rend impossible à détecter.

Seule cette solution s'est démarquée des autres. Elle correspond en fait à l'une des deux implémentations possibles du patron *Adaptateur* proposée par le GOF. L'héritage multiple étant source de conflits à gérer par le programmeur, les langages actuels n'encouragent pas son utilisation. Nous avons donc considéré ce cas comme un modèle alternatif à substituer à la solution utilisant la délégation.

De ce cas, nous avons tirés les particularités remarquables suivantes :

Rôle de départ : « Adaptateur »

Particularité du rôle « Adaptateur » : Classe ayant un héritage multiple

Particularité du rôle « Adapté » : Classes parentes de « Adaptateur » et ayant moins d'enfants que tous les autres parents

Particularité du rôle « But » : Classes parentes de « Adaptateur » et ayant plus d'enfants que tous les autres parents

Nous partons du fait que la classe mère ayant le moins d'enfants est la classe boîte noire à adapter. Ceci reste une heuristique structurale qui doit être soumise à la vérification du concepteur, puisque le stéréotype boîte noire n'existe pas dans la notation UML.

2.4.3) Synthèse sur le patron Adaptateur

Ce problème met en avant le fait que notre méthode de détection est fortement couplée à la notation UML. Tout manque dans la notation limite d'autant notre capacité de détection. Il sera a priori envisageable, voir indispensable, d'imposer aux concepteurs qui voudront utiliser notre système de détection, un ensemble de règles définissant une notation particulière. Ainsi, les détections pourront s'effectuer sur des stéréotypes prédéfinis, et par exemple, le fonctionnement de l'oracle pourra se fonder sur le nom des méthodes et des attributs

2.5) Problème soluble par la Façade

Le cinquième problème de l'énoncé était soluble directement avec le patron *Façade*. Ce paragraphe présente la solution utilisant le modèle de référence (cf. *Figure 18*), moyennant une petite adaptation.

2.5.1) Solution avec le patron

Modéliser le moteur d'un jeu de l'oie pour qu'il soit utilisable par une interface graphique : l'interface graphique aura uniquement besoin de connaître la référence à une partie ainsi qu'à un plateau de jeu mais ne se souciera pas des cases, du dé et des pions, qui sont gérés par le moteur. Cette modélisation doit faire ressortir essentiellement une structure facilitant l'utilisation du moteur par l'interface. Le contenu du moteur de jeu est de moindre importance.

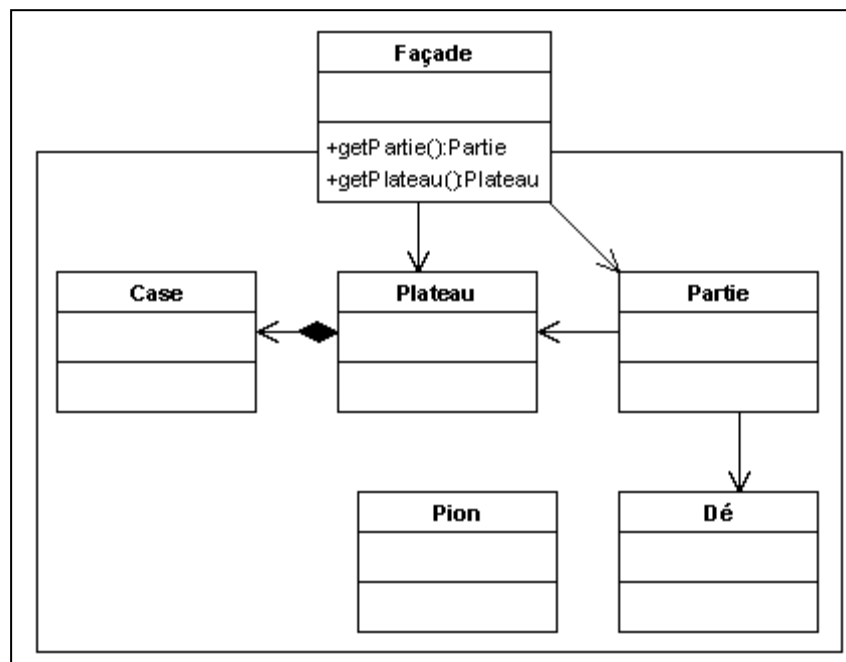


Figure 18 : Modèle de référence de Façade

La nécessité de créer un sous-système avec une interface d'utilisation unique est imposée par l'énoncé. Le moteur du jeu de l'oie constitue le sous-système qui doit être utilisé par une interface graphique. L'interface graphique devant avoir accès à la partie et au plateau du jeu, il apparaît nécessaire d'utiliser une classe *Façade* servant de point d'accès au sous-système.

2.5.2) Modèle alternatif n°1 : fort couplage entre paquetages

Le cas présenté ici n'est pas une solution d'étudiant car le problème posé n'était pas suffisamment clair pour obtenir des solutions exploitables, la notion de sous-système n'ayant pas de correspondance en UML. De plus, la solution de référence (cf. *Figure 18*) ne présente pas d'intérêt non plus, car c'est l'interaction avec d'autres sous-systèmes qui présente tous les éléments décisifs à l'utilisation du patron *Façade*.

Nous avons donc constitué manuellement cette solution type, faisant interagir des paquetages entre eux, mais sans aucun lien avec le problème posé.

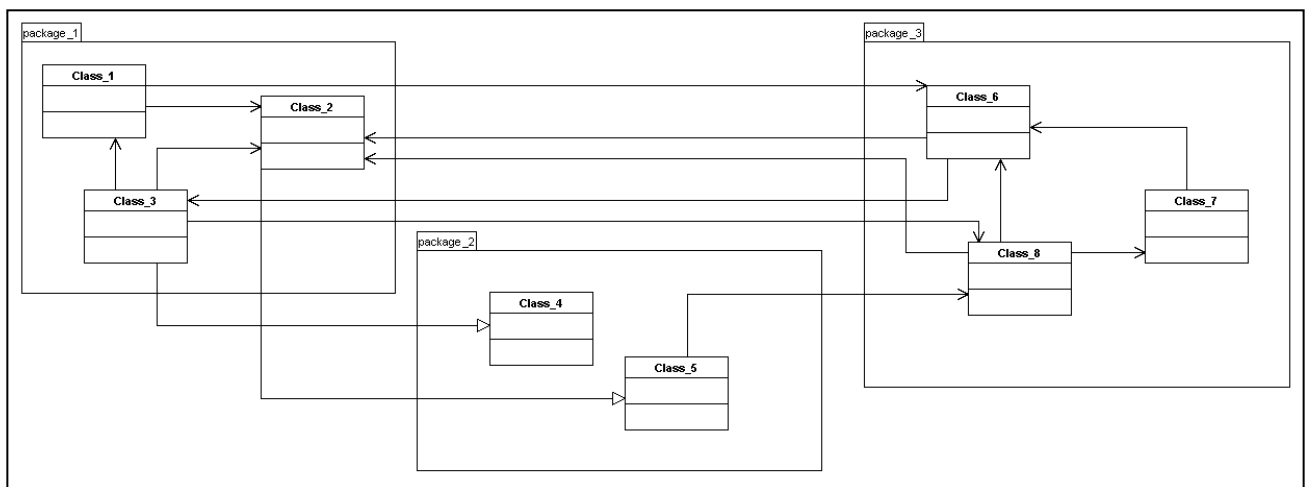


Figure 19 : Modèle alternatif n°1 de Façade

Façade est l'un des rares patrons de conception à ne pas avoir ses constituants décrits sous forme de rôle. En effet, il n'a qu'un unique constituant qui est la classe *Façade*. Elle permet de minimiser les liens de dépendance entre les différents sous-systèmes. Pour détecter l'utilisation éventuelle d'un patron *Façade*, l'utilisation de métriques sur les liens de dépendances entre classes semble être la solution.

Il faut tout d'abord opérer une sélection sur les paquetages ayant une forte complexité interne. La première de ces métriques consiste donc à déterminer la complexité interne d'un paquetage :

Complexité interne = nombre d'associations internes / nombre de classes internes
avec association interne : association reliant deux classes internes au paquetage

Si les classes internes sont faiblement couplées, le patron *Façade* ne va rien apporter, et même alourdir la conception. Une complexité supérieure à 2 semble refléter un fort couplage.

Sur cet ensemble de paquetages sélectionné, il faut opérer une nouvelle sélection en fonction de la complexité externe d'utilisation des classes du paquetage. La seconde de ces métriques consiste ainsi à déterminer la complexité externe d'un paquetage :

Complexité externe = nombre d'associations externes entrantes / nombre de classes externes liées
avec
association externe entrante : association permettant à une classe extérieure d'utiliser une classe interne au paquetage
classe externe liée : classe externe liée à au moins une classe interne du paquetage

Une complexité externe à 1 représente deux possibilités :

- soit une dépendance entre paquetages traduite par une architecture en couches
- soit l'utilisation d'une seule classe du paquetage, jouant de fait le rôle de façade.

Si la complexité externe est différente de 1, une façade simplifierait la gestion des dépendances.

2.5.3) Synthèse sur le patron Façade

De nouveau, notre méthode n'est pas utilisable, mais cela ne nous a pas empêché d'établir un autre moyen de détection. Pour la première fois, les particularités remarquables peuvent se résumer à de simples métriques. Evaluer les couplages entre des paquetages, les multiplicités entre des classes, le nombre d'associations pour une classes... semble être une solution à tous les modèles alternatifs ne présentant pas de particularités structurelles, mais plutôt des particularités quantitatives.

Nous avons donc là encore une nouvelle méthode, mais son fonctionnement est très proche de la méthode d'origine. En adaptant légèrement l'algorithme de détection générique, nous devrions pouvoir intégrer dans la notion de particularités remarquables, des métriques.

2.6) Problème soluble par le Poids mouche

Le sixième problème de l'énoncé était soluble directement avec le patron *Poids mouche* (cf. Figure 20).

La limitation du nombre d'instances est une contrainte non fonctionnelle sous-entendue dans l'énoncé. Le modèle du traitement de texte étant tout objet, il n'est pas envisageable d'avoir un objet par caractère du document, puisqu'on atteindrait très rapidement plusieurs milliers de petits objets en mémoire. Le moyen de contrôler la création de ces objets consiste à utiliser une fabrique. Cette fabrique aura pour but de créer de nouveaux objets si nécessaire, ou de réutiliser des objets existants quand cela est possible. Dans ce cas, on peut facilement imaginer que tous les caractères 'A' d'un document soient le même objet en mémoire.

Après analyse des solutions proposées par les étudiants, nous nous sommes rendu compte que le patron *Poids mouche* n'est pas détectable structurellement. La limitation du nombre d'objets est une information liée aux exigences que l'on ne peut repérer que si la fabrique de poids mouche est identifiée clairement.

De nouveau, une notation prédéfinie est indispensable pour ce genre de détection. En effet, si par avance, un stéréotype signifiant une limitation des instances est ajouté à la classe concernée, la détection sera possible. Cependant, si ce genre de stéréotype est la disposition du concepteur, pourquoi est-ce qu'il n'utiliserait pas d'office le patron dans sa modélisation. Ici, tout pousse donc à dire que si le concepteur n'utilise pas le patron *Poids mouche*, c'est qu'il ne faut pas l'utiliser, et qu'il n'est, par conséquent, pas nécessaire de le détecter.

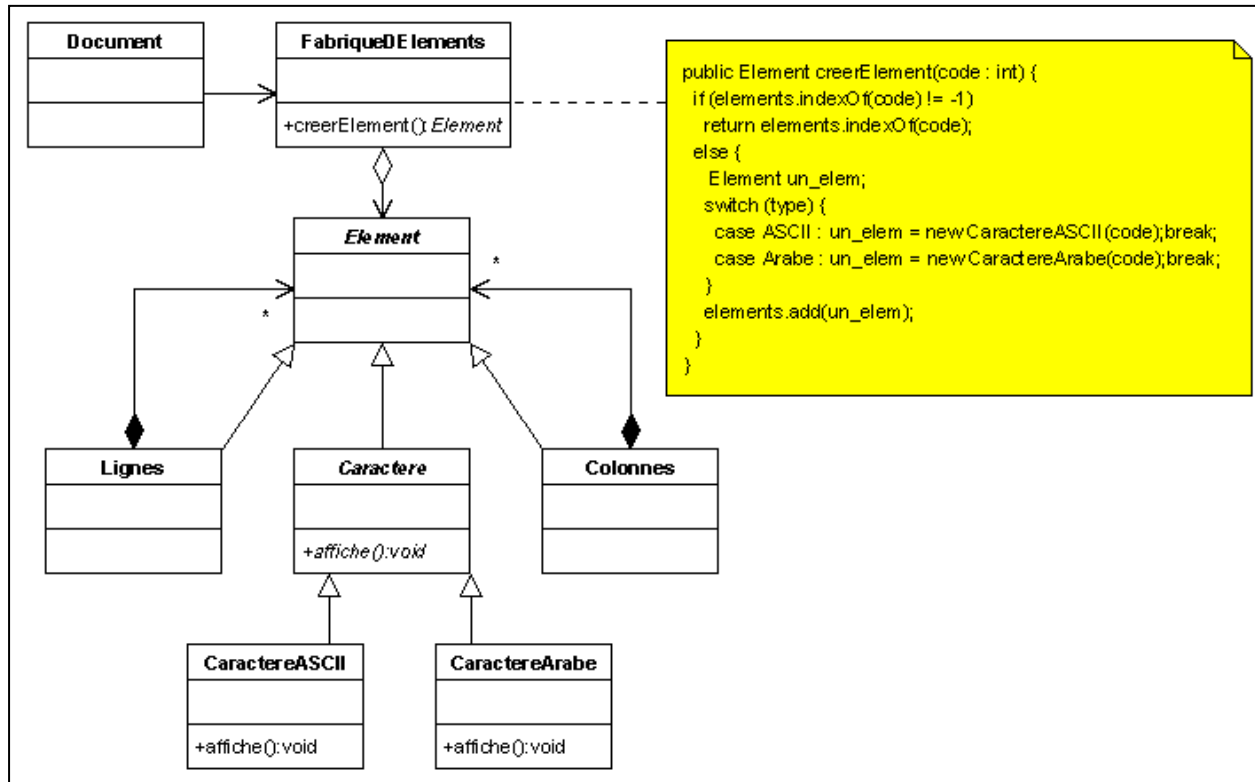


Figure 20 : Modèle de référence de Poids mouche

2.7) Problème soluble par la Procuration

Le septième problème de l'énoncé était soluble directement avec le patron *Poids mouche* (cf. Figure 21).

Le but du problème est d'afficher un document le plus rapidement possible, quelle que soit la taille des images, ou des fichiers vidéo. La solution consiste à ne pas afficher les images et les vidéos, tant qu'elles ne sont pas visibles à l'écran. Pour cela, il suffit de remplacer ces objets par d'autres qui ne vont pas s'occuper du contenu de l'image ou de la vidéo, mais qui seront capables de calculer leur taille. Alors qu'un objet *Image* charge les données en mémoire, un objet *ProxyImage* se contente de lire la taille du fichier sur le disque. Au lancement du document, toutes les images seront remplacées par un *ProxyImage* qui fournira la taille. Dès que l'image devra être affichée, le *ProxyImage* sera remplacé par l'image proprement dite.

Là encore, après analyse des solutions, nous nous sommes rendus compte que la Procuration n'est pas détectable structurellement. Ces contraintes non fonctionnelles liées aux performances de l'application sont fournies par les exigences, et ne sont repérables que si l'on a un moyen d'identifier une classe ayant le rôle de procuration.

De la même manière que pour le patron *Poids mouche*, l'ajout d'un stéréotype prédéfini semble être inutile, puisque si le concepteur représente un de ces classes comme proxy, il utilise de fait le patron. Certains étudiants ont nommés leur classe *Buffer*, ce qui laissait à penser qu'ils souhaitaient l'utiliser comme procuration. Leurs modèles étaient pour la plupart conformes au patron, sauf quelques uns qui ne factorisaient pas suffisamment. Ce manque de factorisation ne peut pas être considéré comme particularité remarquable de ce patron, mais plutôt comme passage obligé, précédant toute détection de tous les patrons.

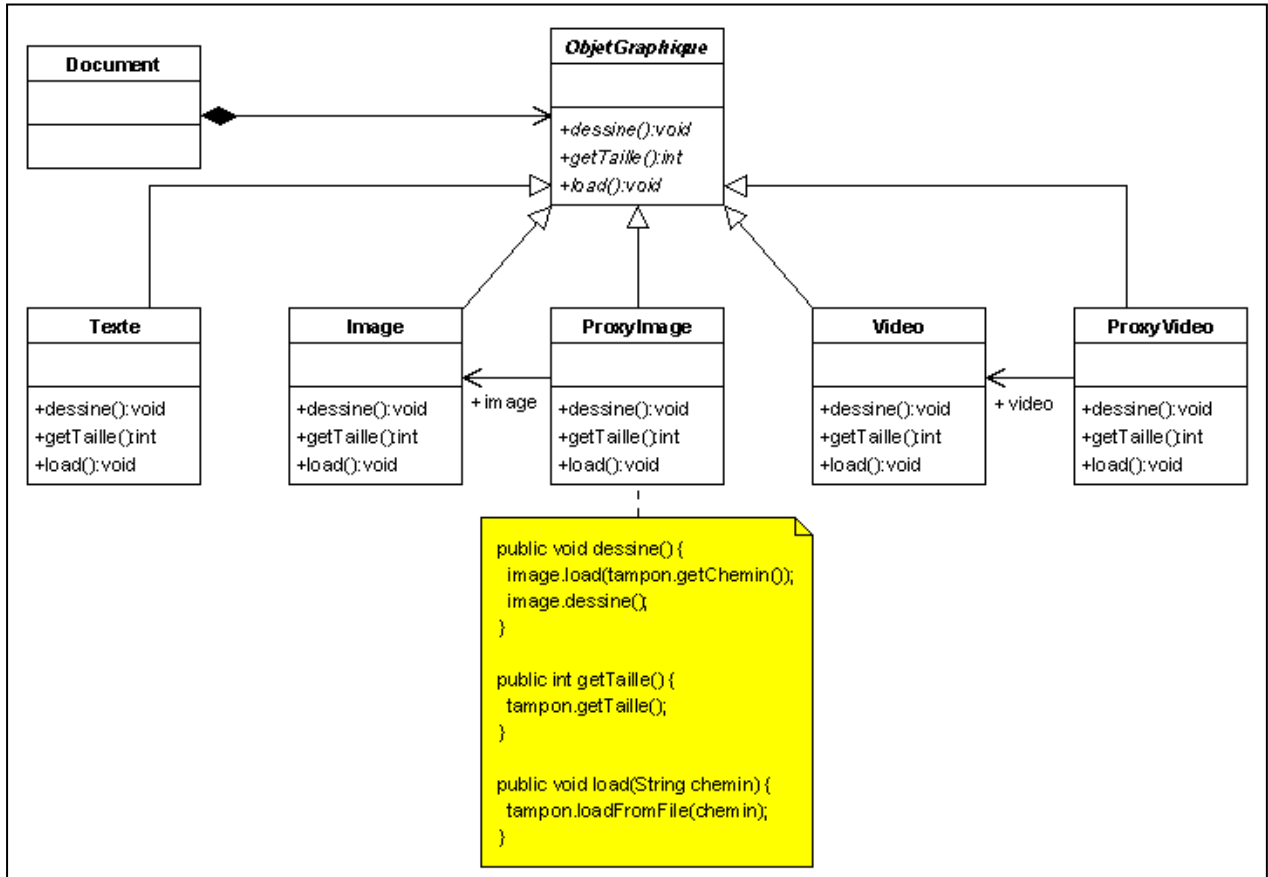


Figure 21 : Modèle de référence de Procuration

Les deux derniers patrons étudiés (cf. Figure 20 et Figure 21) augmentent la complexité structurelle des modèles, ce qui ne permet pas a priori la détection de propriétés structurelles remarquables. D'autre part, ils sont fortement liés à des exigences de temps d'exécution et de gestion de la mémoire. Ils résolvent de véritables problèmes de conception, mais avec une orientation système non fonctionnelle. Des éléments extérieurs doivent donc être ajoutés en plus des particularités remarquables, éléments prédéfinis et connus du concepteur.

3. MISE EN ŒUVRE

Afin d'expérimenter les algorithmes de détection sur des modèles, nous avons utilisé la plate-forme NEPTUNE, qui permet d'appliquer facilement des règles OCL sur des modèles XMI. Il a donc fallu coder chaque algorithme par plusieurs règles OCL (cf. ANNEXE 1). Normalement, une règle OCL permet d'ajouter des contraintes à un diagramme UML. Une règle renvoie donc un booléen qui indique si un diagramme respecte ou non la règle concernée. Dans notre cas, nous souhaitons que les règles OCL nous retournent des ensembles de classes correspondants aux différents rôles de chaque patron. Pour ce faire, nous avons écrit les règles OCL de manière à ce que leur valeur de retour soit toujours fausse. Lorsqu'une règle retourne faux, la plate-forme NEPTUNE donne le contexte de l'erreur, contexte qui est souvent l'ensemble des classes sur lequel était appliquée la règle. Profitant de cette particularité de la plate-forme, nous avons écrit les règles de manière à ce que le contexte de l'erreur corresponde à l'ensemble des classes qui m'intéressait.

Pour illustrer la méthode de construction, nous allons détailler la construction des règles du modèle alternatif n°1 du patron *Composite* :

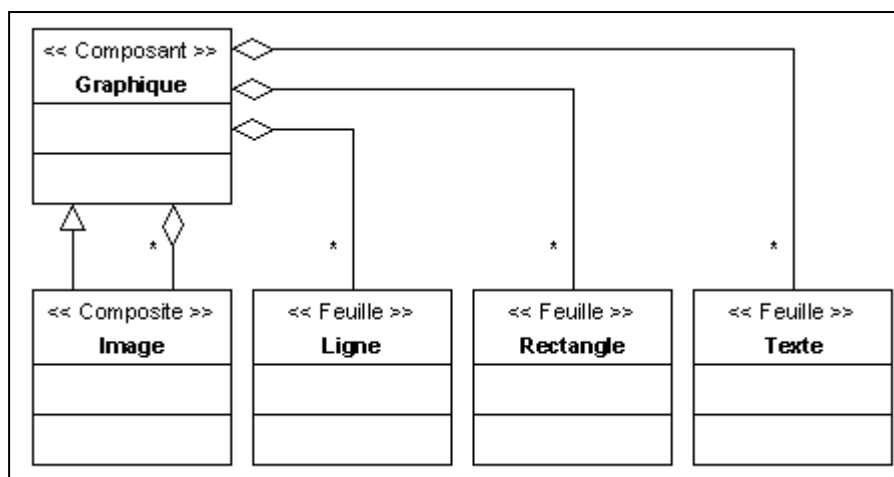


Figure 22 : Modèle alternatif n°1 du Composite

Voici les particularités remarquables identifiées :

Rôle de départ : « Compositant »

Particularité du rôle « Compositant » : Classe ayant au moins deux agrégations

Particularité du rôle « Composite » : Classe étant à la fois fille et agrégée au « Compositant »

Particularité du rôle « Feuille » : Classe agrégée au « Compositant » et qui n'est pas fille

Pour écrire une règle OCL, il faut naviguer dans le méta-modèle, à la recherche des méta-classes significatives. La Figure 23 présente un extrait du méta-modèle UML 1.4, qui permettra d'illustrer la construction des règles détectant le modèle alternatif.

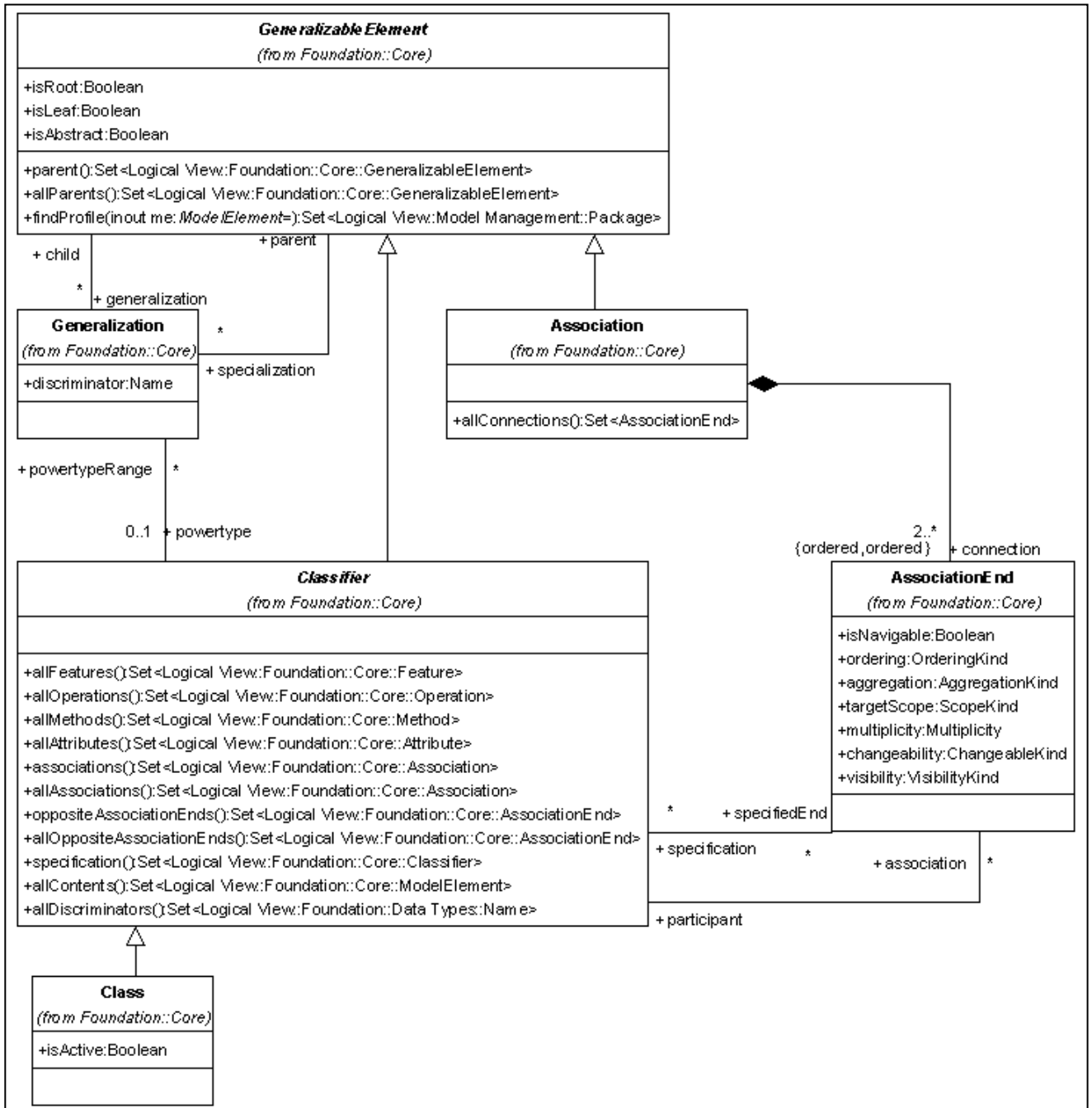


Figure 23 : Extrait du méta-modèle UML 1.4

Le rôle de référence de ce patron étant « Composant », nous recherchons tout d'abord les classes ayant les particularités structurelles de ce rôle.

Particularité du rôle « Composant » : Classe ayant au moins deux agrégations

Pour repérer toutes les classes ayant au moins deux agrégations, il nous a suffi de partir de toutes les instances de *Class*, et de remonter jusqu'à la méta-classe *AssociationEnd* qui représente les bouts d'associations accrochés à une classe. L'attribut *aggregation* contient 1 si la classe concernée est agrégée, 2 ou 3 si elle est composant ou agrégation. Il nous a suffi ensuite de quantifier le nombre de bouts d'associations. On obtient la règle suivante :

```
Class.allInstances->select(association->select(aggregation=2 or aggregation=3)->size > 1).oclAsType(Classifier)->asSet
```

Une règle OCK doit toujours retourner un booléen, donc cette règle n'est pas terminées, puisqu'elle retourne l'ensemble des classes ayant au moins deux agrégations. En forçant un retour faux en utilisant la commande *fail()*, NEPTUNE donnera le contexte de l'erreur qui sera l'ensemble des classes qui nous intéresse. La règle finale est donc :

```
Class.allInstances->select(association->select(aggregation=2 or aggregation=3)->
size > 1).oclAsType(Classifier)->asset->fail('Classes composant : ')
```

Cette règle retourne donc toutes les classes ayant au moins deux agrégations, ce qui correspond à la particularité remarquable du rôle « Composant ».

Particularité du rôle « Composite » : Classe étant à la fois fille et agrégée au « Composant »

Pour ce rôle, et en suivant notre méthode générique, nous avons réutilisé l'ensemble obtenu par la règle précédent. Pour chacun des classes identifiées comme « Composant », nous avons cherché les classes agrégées et vérifié si le parent de ces classes tait bien la classe « Composant » courante. Pour obtenir les agrégées, il a suffit de suivre le même lien que pour la règle précédente, et à partir des classes trouvées, de comparer leur parent à la classes « Composant », en utilisant le paramètre *parent* de la méta-classe *GénéralizableElement*. Enfin, il a fallu invalider le résultat pour obtenir l'ensemble. Voici la règle, qui fait appel à la règle précédente, nommé *composant*. La partie en gras constitue la recherche de la particularité remarquable, le reste servant à itérer sur els classes « Composant ».

```
composant -> iterate(c : Classifier;res : Bag(Set(Classifier)) = Bag{} | res->
including(c.association->select(aggregation=2 or aggregation=3).association.
connection->select(aggregation=1).participant-> select(parent = Set{c}).
oclAsType(Classifier)->asSet))->fail('Classes composite : ')
```

Cette règle retourne donc toutes les classes étant à la fois filles et agrégées aux classes « Composant », ce qui correspond à la particularité remarquable du rôle « Composite ».

Particularité du rôle « Feuille » : Classe agrégée au « Composant » et qui n'est pas fille

Là encore pour ce rôle, une simple navigation dans le méta-modèle jusqu'à la méta-classe *AssociationEnd* permet d'obtenir les classes agrégées au « Composant ». Pour éliminer les filles, il nous a suffit d'éliminer de cet ensemble les classes déjà identifiées en temps de « Composite ». Voici la règle obtenue qui fait appel aux deux règles précédents. La partie en grand constitue la détection.

```
composant -> iterate(c: Classifier;res : Bag(Set(Classifier)) = Bag{} | res->
including(c.association->select(aggregation=2 or aggregation=3).association.
connection->select(aggregation=1).participant.oclAsType(Classifier)->asSet
-
composite(c))
```

Le résultat des trois règles se présente donc sous la forme de plusieurs ensembles de classes, chacun correspondant à un rôle du patron.

Selon la même méthode, nous avons écrit les règles de chaque rôle de chaque modèle alternatif. NEPTUNE travaillant avec des modèles XMI, nous avons cherché des modèles et des méta-modèles sur lesquels nous allions pouvoir appliquer toutes ces règles et vérifier que le résultat obtenu pouvait être remplacé par un patron de conception structurel.

4. EXPERIMENTATION

Ce paragraphe synthétise les expérimentations que nous avons réalisées. Dans un premier temps nous avons appliqué les règles sur des modèles, puis sur des méta-modèles, ce qui nous a permis d'écrire un article pour le workshop sur les patrons de méta-modèles des journées IDM 06. De plus, dans cet article, nous avons essayé de trouver des patrons de conception candidats pour être patrons de méta-modèles (*cf. ANNEXE 2*).

4.1) Expérimentation sur des modèles

4.1.1) Difficultés rencontrées

L'équipe n'ayant pas à ce jour de contrats industriels, nous avons eu certaines difficultés à trouver des modèles industriels UML conséquents. Nous avons tout de même à notre disposition, le modèle complet de la plate-forme NEPTUNE 1 au format XMI. Hélas, ce modèle ayant été conçu avec l'outil Rose, et soi-disant exporté au format XMI, s'est avéré illisible par d'autres éditeurs UML et par la plate-forme NEPTUNE. Après quelques investigations, nous nous sommes rendu compte que le problème ne concernait pas un défaut de génération, mais bel et bien un manque de compatibilité entre différents outils. En effet, deux fichiers XMI, générés avec deux outils différents, à partir de deux modèles identiques, sont différents. De plus, la première version de la plate-forme est dépendante d'un format XMI, c'est à dire que sa représentation interne est associée de manière figée au méta-modèle XMI 1.3, ce qui n'a pas facilité nos recherches.

Finalement, nous nous sommes tournés vers un extrait du modèle statique de la future plate-forme NEPTUNE 2 (*cf. Figure 24*), concernant le composant générateur de méta-modèles. Nous avons pu appliquer sur ce modèle nos règles, puisque nous disposions d'un format compatible.

4.1.2) Deux propositions

L'application des règles sur le modèle a permis de détecter un certain nombre de modèles alternatifs desquels nous avons extrait deux des plus pertinents.

4.1.2.1) Détection d'un Composite

Le premier est le modèle alternatif n°4 du patron *Composite*. Voici le résultat de l'application des règles :

Classe « Composite » : EntiteLogique
Classe « Composant » : EntiteJava
Classe « Feuille » : MethodeJava

En effet, les classes détectées correspondent aux particularités remarquables de chaque rôle de ce modèle alternatif.

EntiteLogique « Composite » : Classe ayant au moins deux agrégations, dont une au moins réflexive et ayant une super-classe

EntiteJava « Composant » : Classe mère du « Composite »

MethodeJava « Feuille » : Classe agrégée au « Composite »

La classe *EntiteJava* nous semble être une bonne candidate au rôle « Composant », car elle est racine du graphe d'héritage. A ce titre, elle propose un ensemble d'opérations minimales à travers desquelles toute *EntiteJava* va pouvoir être manipulée.

La classe *EntiteLogique* nous semble être aussi une bonne candidate au rôle de « Composite », car elle est munie d'un agrégat récursif dont le nom des extrémités (*entiteConteneur* – *entiteContenu*) se rapproche d'une sémantique d'arbre de décomposition.

Enfin, la classe *MethodeJava* semble correspondre elle-aussi au rôle de « Feuille », car elle n'est pas, à priori, décomposable en entité plus basique.

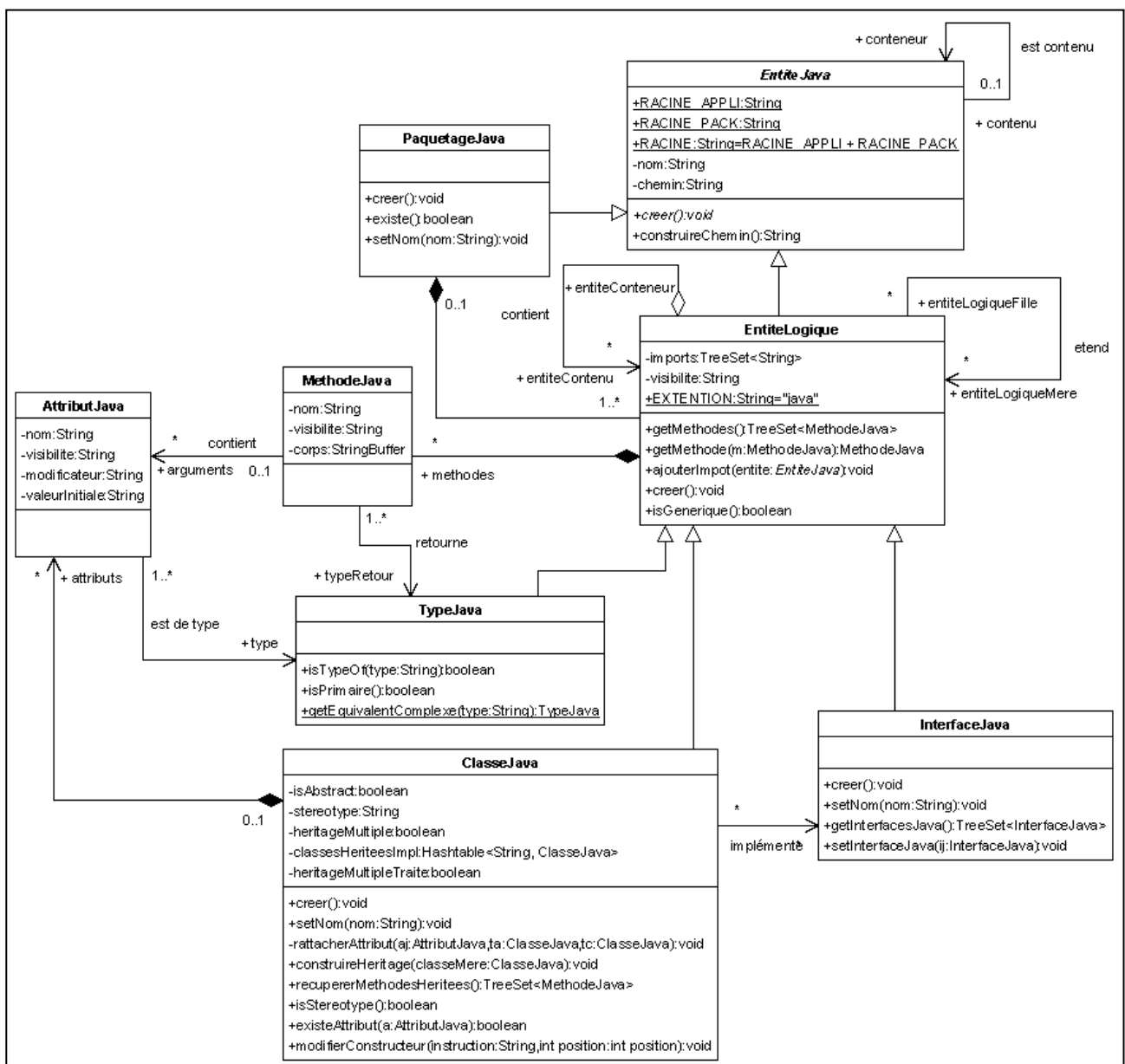


Figure 24 : Extrait du modèle "générateur de méta-modèles" de NEPTUNE 2

Si l'on considère ce fragment comme point de départ du patron *Composite*, nous pouvons étendre sa portée en ajoutant la classe *AttributJava* en temps que « Feuille », et les classes *ClassesJava* et *InterfaceJava* en tant que spécialisation du « Composite » *EntiteLogique*. Toute *EntiteLogique* se composerait alors d'*EntiteJava* avec une contrainte spéciale portant sur la classe *AttributJava*, un *AttributJava* ne pouvant être composé que dans une *ClasseJava*.

4.1.2.2) Détection d'un Décorateur

Le modèle alternatif n°1 du patron *Décorateur* a aussi été détecté. Voici le résultat des règles :

Classe « Composant » : *EntiteLogique* (Classe étant à la fois agrégation et mère)
Classe « Composant concret » : *ClasseJava*, *InterfaceJava*, *TypeJava* (Classes filles de « Composant »)

Classe « Décorateur concret » : *MethodeJava* (Classe agrégée à « Composant »)

A la vue de cette proposition, un expert en modélisation objets ne validerait certainement pas l'intégration de ce patron. D'une part le « Décorateur concret » *MethodeJava* est unique, ce qui n'incite pas à la décoration à la volée avec d'autres décorateurs. D'autre part, les classes *ClasseJava*, *InterfaceJava* et *TypeJava*, dans l'optique d'une génération, n'ont pas besoin d'être décorées par le comportement d'une *MethodeJava*. Pour aller plus loin, une analyse dynamique des messages transmis nous permettrait de savoir s'il y a ajout d'un comportement spécifique de la part de *MethodeJava*.

4.1.3) Synthèse

Ce qui nous semble intéressant de signaler sur ces deux exemples, est le chevauchement des deux fragments ciblés par les règles. Bien que les patrons *Composite* et *Décorateur* peuvent s'assembler (un *Décorateur* peut-être fusionné avec un *Composite* dégénéré), les classes ciblés par les règles les rendent incompatibles. En effet, il y aurait fusion entre les « Décorateur concret » et les « Feuilles » du *Composite*, ce qui contredit leur intégration commune. Un choix doit donc être fait en fonction de leur pertinence dans les modèles. Dans notre exemple, l'injection du patron *Composite* se justifie plus, d'autant que le *Décorateur*, même s'il avait été détecté seul, n'aurait pas d'intérêt dans ce fragment de modèle.

De manière plus générale, si les classes ciblées se chevauchent entre plusieurs patrons, soit nous sommes en présence d'un patron composite [D. Riehle, 1997], soit nous sommes dans le cas de figure précédent.

4.2) Expérimentation sur des méta-modèles

Les méta-modèles UML étant instances du MOF, ils sont décrits par des diagrammes de classes, ce qui nous a permis d'y appliquer nos règles OCL. Nous les avons considérés alors comme des modèles aptes à l'intégration de patrons de conception. De ce fait, certains d'entre eux pourraient être considérés comme des patrons de méta-modèle.

Sur les sept patrons structurels du GOF, nous en avons éliminé quatre :

- Façade, car les liens de dépendances entre les paquetages des méta-modèles de l'OMG sont en général matérialisés par des liens d'héritages, suggérant une architecture logicielle en couches.

- Pont, car les liaisons entre abstraction et implémentation n'ont pas à être définies à ce niveau de modélisation.

- Procuration et Poids mouche, car ils ne sont pas détectables structurellement. Les problèmes qu'ils résolvent émanent d'exigences non fonctionnelles.

L'application des règles concernant les trois patrons restants Composite, Décorateur et Adaptateur, nous a permis de cibler des fragments de méta-modèles à partir desquels nous proposons la discussion suivante. Notre expérimentation a porté sur le paquetage core de UML1.4 et sur le méta-modèle SPEM, pour éviter l'explosion combinatoire des possibilités d'intégration de patrons, et pour faciliter la vérification de notre hypothèse.

4.2.1) Composite et Décorateur intégrés dans le SPEM

Le SPEM (Software Process Engineering Metamodel) est un méta-modèle spécifique aux processus de développement proposé par l'OMG (OMG, 2005). Le noyau se compose en particulier des classes WorkDefinition (découpage du processus en activité élémentaire, itération, phase et cycle de vie), ProcessRole (entité reliant des compétences requises à des activités) et WorkProduct (paramètre en entrée-sortie des activités). La Figure 25 montre le fragment ciblé par nos règles. Elles font apparaître deux patrons de conception susceptibles d'être intégrés : le patron Composite (relation réflexive subwork sur la super classe WorkDefinition) et le patron Décorateur sur le même ensemble de classes.

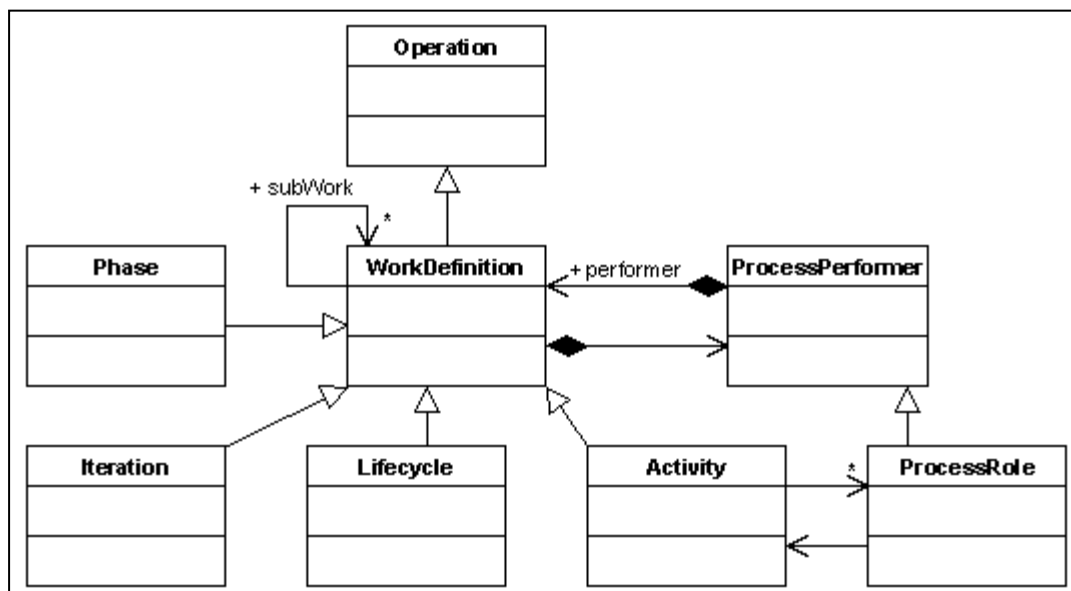


Figure 25 : Fragment du SPEM 1.1

Une analyse plus fine nous permet de constater que la classe WorkDefinition joue plusieurs rôles : les rôles « Composite » et « Composant » du patron Composite, et le rôle « Décorateur » du patron Décorateur. La classe Activity ne pouvant se décomposer qu'en travaux atomiques nommés Step, elle joue nécessairement le rôle « Feuille » du patron Composite. Les classes Lifecycle, Phase et Iteration peuvent jouer le rôle « Décorateurs concrets » : Phase et Iteration ajoutent les notions de temps et de buts à atteindre, Lifecycle

spécialise la relation subWork aux instances de Phase et relie celles-ci aux processus et composants de processus considérés comme des référentiels d'activités.

La Figure 26 montre le fragment du méta-modèle SPEM, enrichie d'une intégration explicite des deux patrons Composite et Décorateur. En intégrant la classe WorkComponent, jouant explicitement le rôle de « Composant », nous simplifions la définition de la classe WorkDefinition jouant uniquement les rôles « Composite » et « Décorateur ».

L'intégration de ces deux patrons de conception a deux effets : d'une part elle clarifie les possibilités de chacune des classes de ce méta-modèle, en leur attribuant des rôles précis. D'autre part, elle permet de préciser la sémantique du méta-modèle sans ajouter de *well formedness rules* supplémentaires ou de descriptions en langage naturel. Par exemple, le rôle « Feuille » de la classe Activity permet de se passer des précisions suivantes (OMG, §7-3, p7-5, 2005) : « *Although this is not explicitly prohibited, an Activity does not normally use the subWork structure inherited from WorkDefinition; instead decomposition within Activity is done using Steps* ». En revanche, des contraintes de composabilité entre les différents décorateurs concrets restent nécessaires.

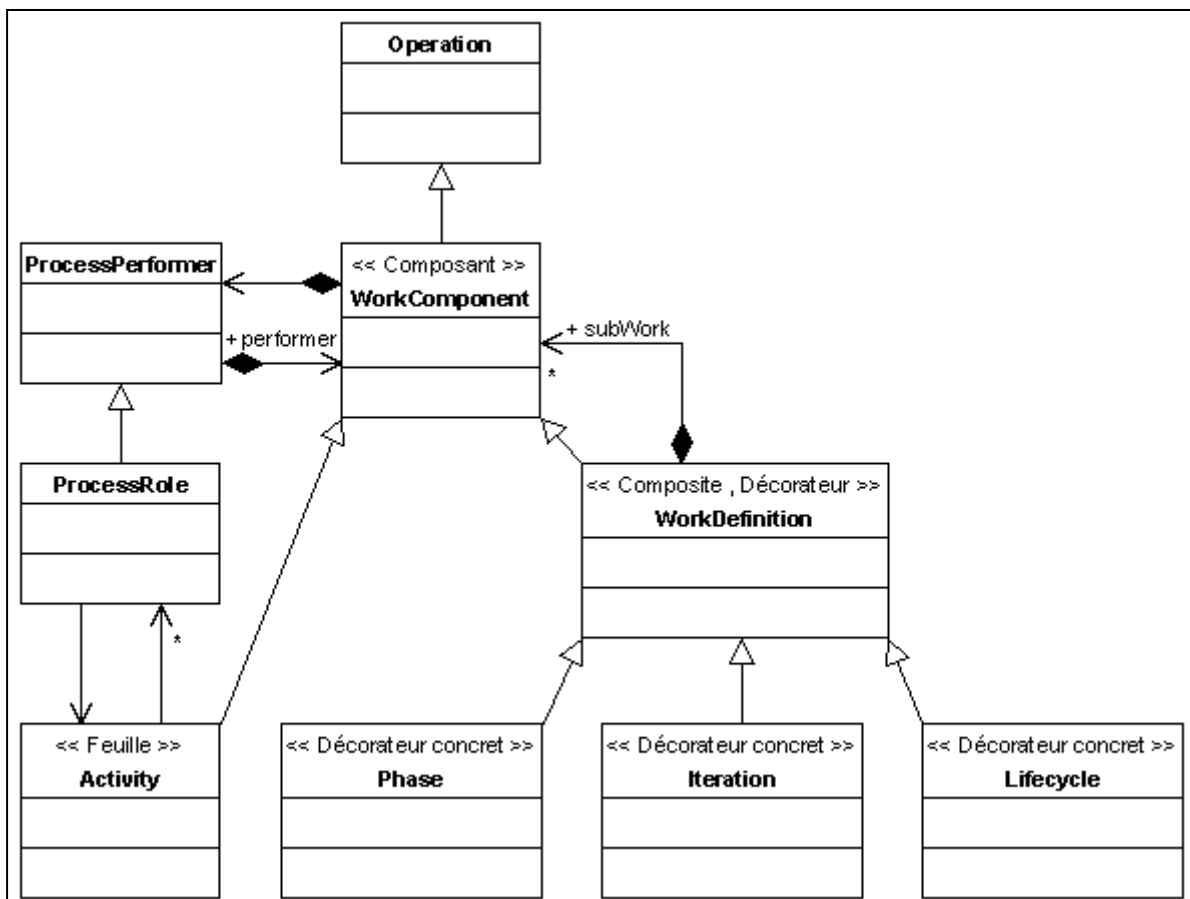


Figure 26 : Intégration des patrons Composite et Décorateur dans le SPEM 1.1

4.2.2) Composite et Adaptateur pour simplification du paquetage core de UML 1.4

Lors de l'application des règles OCL sur le noyau du méta-modèle UML 1.4, nous obtenons un patron Composite sur les classes Namespace (rôle « Composite ») et ModelElement (rôle « Composant »). A priori, la classe GeneralizableElement est « Feuille », mais pour des raisons conceptuelles, les classes Classifier et Package sont en même temps

« Feuille » (hérite de GeneralizableElement) et « Composite » (hérite de Namespace). Il existe donc une double structure d'héritage multiple et répété intégrée dans un patron Composite.

Nous proposons alors une simplification de conception en intégrant le patron Adaptateur, chargé d'implémenter l'interface de GeneralizableElement pour l'ensemble de ses anciennes sous classes, comme cela est suggéré à la Figure 27.

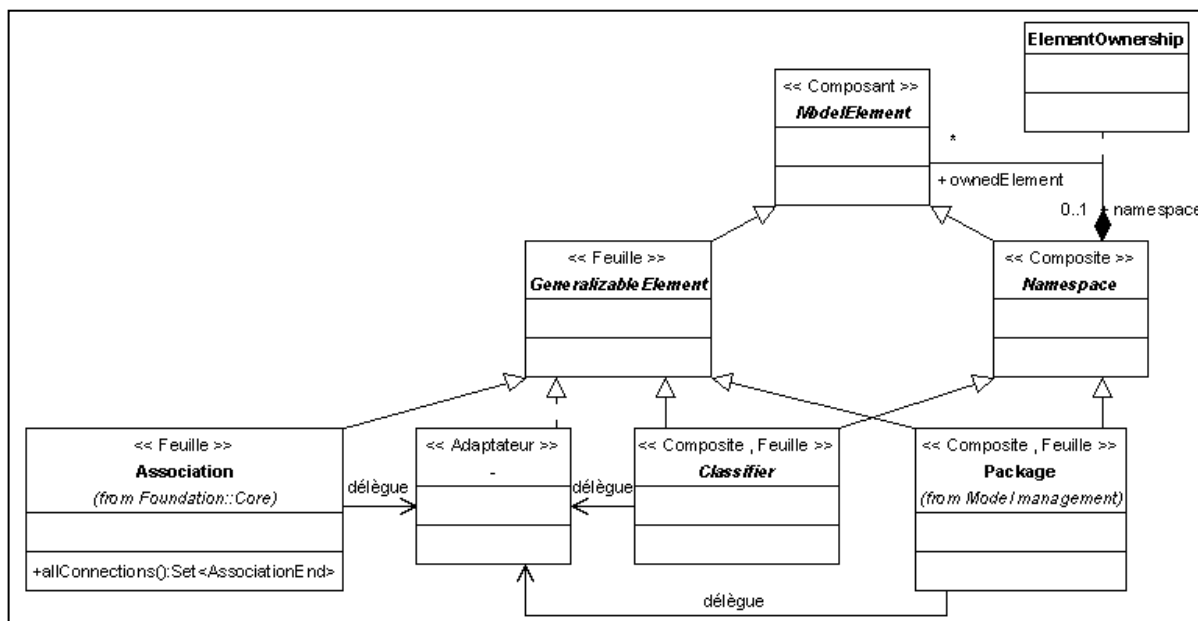


Figure 27 : Intégration d'un Adaptateur dans un fragment du noyau de UML 1.4

4.2.3) Synthèse

Cette détection, prévue au départ pour des modèles UML, nous a permis de détecter des patrons dit de conception sur des méta-modèles de l'OMG. Nous considérons que les patrons Composite et Décorateur sont des candidats au statut de patrons de méta-modèle.

Afin d'étendre la discussion, il nous semblerait intéressant d'étudier le statut des autres patrons de conception. Si nous éliminons les patrons créateurs, chargés de gérer l'instanciation d'objets complexes (agrégats, frameworks), il subsiste certains patrons comportementaux candidats : le patron Interprète, servant de support au méta-modèle dédié aux langages spécifiques de domaines (DSL), le patron Chaîne de responsabilité, chargé d'explicitier les liens de navigation au travers des classes du méta-modèle et le patron Médiateur simplifiant la navigation dans les modèles. De plus, le patron structurel Pont peut être réutilisé pour la séparation propre entre les éléments de modélisation et les représentations graphiques associées.

CONCLUSION

Dans cette étude, nous avons montré qu'il est possible d'injecter dans des modèles des patrons de conception. Nous avons restreint notre champ d'étude aux patrons de conceptions structurels que nous détectons grâce à des modèles alternatifs. Avant de penser à l'expérimentation, nous avons étudiés les patrons créateurs, en essayant de trouver des particularités structurelles à partir d'un problème fixé. Voici un résumé de notre investigation sur les patrons créateurs :

Nous avons travaillé sur un problème de construction d'un labyrinthe : un labyrinthe était un dédale composé d'un ensemble de lieux, chaque lieu pouvant être une salle, une porte ou un mur :

- *Fabrique abstraite* : Ce patron a été le plus facile à utiliser pour ce problème, la notion de groupe de produit pouvant être considéré comme une condition d'application (dans le problème, il s'agirait de Salle, Porte et Mur). Le problème est venu de la détection des produits. En effet, comment repérer, dans un diagramme de classes, qu'une certaine classe peut être considérée comme groupe de produit ? Il est possible d'imaginer comme point de départ que des classes héritant d'une super classe abstraite peuvent constituer un groupe de produit.

- *Monteur* : Très convainquant, il semble que c'est la meilleure solution pour ce genre de problème. La notion de construction séparée apporte un grand intérêt à la conception. En revanche, le repérage semble compliqué puisqu'à priori, rien n'impose le recours à ce patron, son utilisation n'apportant qu'une souplesse de conception.

- *Fabrication* : La ressemblance structurelle avec la *Fabrique abstraite* est surprenante. De fait, la différenciation de ces deux patrons va probablement constituer un problème, même si l'utilisation de *Patron de méthode* constitue la différence des deux patrons. Le repérage de *Patron de méthode* devrait permettre la greffe de *Fabrique abstraite*.

- *Prototype* : Nous ne sommes pas parvenus à utiliser le patron *Prototype* dans cette conception sans le greffer à *Fabrique abstraite*. Cette greffe, associée à la proximité structurelle de *Fabrication* et de *Patron de méthode*, pourrait laisser présumer que d'autres sous-ensembles. Ainsi, *Fabrique abstraite*, *Patron de méthode* et *Prototype* peuvent constituer une sorte de famille de patrons, qui faciliterait le repérage mutuel.

- *Singleton* : Bien qu'il n'avait à priori pas sa place dans ce problème, nous avons tout de même réussi à appliquer ce patron en suivant les directives du GOF, mais sans en être totalement convaincus.

Le fait que nous ayons réussi à appliquer les patrons créateur sur le même problème laisse présager qu'il sera difficile d'obtenir des modèles alternatifs conséquents en faisant le même type d'expérience. La différenciation de ces patrons étant essentiellement liée à la manière dont la création des objets doit être effectuée, l'énoncé des problèmes devrait contenir des exigences non fonctionnelles, qu'il sera difficile d'intégrer en particularités remarquables (comme pour les patrons *Procuration* et *Poids mouche*). De plus, une analyse fine de la dynamique du système devra être envisagée pour repérer l'adéquation entre les classes créatrices d'instances et les classes utilisatrices de ces mêmes instances.

En ce qui concerne les patrons comportementaux, là en revanche, il doit être possible, en effectuant le même type d'expérience, de déduire des particularités remarquables de modèles alternatifs dynamiques. Une fois ce travail effectué, il devrait être envisageable de déduire les patrons restant, en utilisant par exemple les relations proposés par le GOF (cf Figure 28).

- les patrons composites sont des bons candidats pour résoudre les situations où plusieurs patrons se chevaucheraient.

- plus en amont, en phase d'analyse, une étude sur les patrons métier devrait être entreprise.

Pour chacun des patrons étudié, va correspondre un ensemble non négligeable de modèles alternatifs, multipliant le nombre de règles OCL à écrire. En utilisant les travaux de [G. Sunyé, *et al.*, 2000], le développement d'un outil de construction de règles automatiques, à partir des particularités remarquables, s'avèrera sûrement indispensable.

BIBLIOGRAPHIE

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1995.

Neptune, Nice Environment with a Process and Tools using Norms - UML, XML and XMI - and Example, [w] <http://neptune.irit.fr>, 2003.

Object Management Group, Software Process Engineering Metamodel Specification, version 1.1 formal/05-01-06, 2005.

Riehle D. « Composite Design Patterns », *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA '97*. ACM Press, 1997. Page 218-228.

Agnès Conte, M. Fredj, J.P. Giraudin, D. Rieu, **P-Sigma : un formalisme pour une représentation unifiée de patrons**, Inforsid'01, Genève, Mai 2001.

Dirk Riehle. "Bureaucracy--A Composite Pattern." EuroPLoP '96, *preliminary conference proceedings*. Kloster Irsee, 87660 Irsee, Germany, 1996. Paper 16, 11 pages.

Dirk Riehle, D. Composite design patterns. In ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, pages 218--228, 1997.

Agnès Conte, Ibtissem Hassine, Jean-Pierre Giraudin, Dominique Rieu: AGAP : un Atelier de Gestion et d'Application de Patrons. INFORSID 2001: 142-159

Bruno Barthez, Agnès Front-Conte, Dominique Rieu: Intégration d'imitations de patrons pour la spécification de cadrage. INFORSID 2000: 399-415

Corine Cauvet, Lilia Gzara, Philippe Ramadour, Dominique Rieu: Ingénierie des systèmes d'information produit: une approche méthodologique centrée réutilisation de patrons. L'OBJET 6(4): (2000)

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1995.

Borne I. et Revault N. Comparaison d'outils de mise en œuvre de design patterns. L'Objet, 5(2) : 243-266, 1999.

Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel, (1977), *A Pattern Language : Towns, Buildings, Construction*,

Kent Ward Cunningham, Beck, Apple Computer, Inc., Using Pattern Languages for Object-Oriented Programs,

OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming., 1987.

Jan Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2) :18-32, 1998.

Albin-Amiot H, Cointe P., Guéhéneuc Y. et Jussien N. *Instanciating and Detecting Design Pattern : Putting Bits and Pieces Together*. Proceedings of ASE'2001.

Eden A, Gil J. et Yehudai A. Automating the application of Design Patterns. *Journal of Object Oriented Programming*, May 1997.

Krämer C. et Prechelt L. Design Recovery by Automated for Structural Design Patterns in Object Oriented Software. In *Working Conference on Reverse Engineering (WCRE 96)*. IEEE CS Press, 1996.

[Hervé Leblanc](#), [Thierry Millan](#), [Ileana Ober](#). *Démarche de développement orienté modèles : de la vérification de modèles à l'outillage de la démarche*. Dans : *Ingénierie Dirigée par les Modèles*, Paris, 30 juin 1 juillet 2005. Sébastien Gérard, Jean-Marie Favre, Pierre-Alain Müller, Xavier Blanc (Eds.), CEA List

Graig Larman, *Applying UML on Patterns*, Prentice Hall, 2000, seconde édition

G. Sunyé, A.L. Guennec, J-M Jézéquel, "Precise Modeling of Design Patterns", In Proceedings of UML 2000, volume 1939 of LNCS, pages 482-496. Springer Verlag, 2000

ANNEXES

ANNEXE 1 REGLES OCL	I
ANNEXE 2 ARTICLE POUR LE WORKSHOP SUR LES PATRONS DE META-MODELES	XIII

ANNEXE 1

Règles OCL

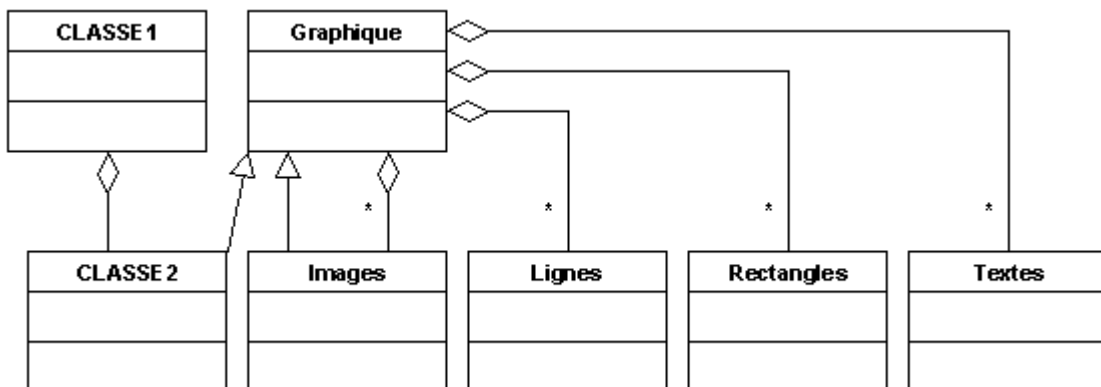
COMPOSITE, cas 1

```
inv composant :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 1).oclAsType
    (Classifier)->asSet
  in
    composant -> fail('COMPOSANTS : ')
inv composite :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 1).oclAsType
    (Classifier)->asSet
  let composite : Bag(Set(Classifier)) = composant -> iterate(c : Classifier;
    res : Bag(Set(Classifier)) = Bag{} | res->including(c.association ->
    select(aggregation=2 or aggregation=3).association.connection ->
    select(aggregation=1).participant-> select(parent = Set{c}).oclAsType
    (Classifier)->asSet))
  in
    composite -> fail('COMPOSITES : ')
inv feuille :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 1).oclAsType
    (Classifier)->asSet
  let composite(composant_courant : Classifier) : Set(Classifier) =
    composant_courant.association->select(aggregation=2 or aggregation=3)
    .association.connection -> select(aggregation=1).participant ->
    select(parent = Set{composant_courant}).oclAsType(Classifier)->asSet
  let feuille : Bag(Set(Classifier)) = composant -> iterate(c: Classifier;
    res : Bag(Set(Classifier)) = Bag{} | res->including(c.association ->
    select(aggregation=2 or aggregation=3).association.connection ->
    select(aggregation=1).participant.oclAsType(Classifier)->asSet
    -
    composite(c)))
  in
    feuille -> fail('FEUILLES : ')
```

COMPOSANTS : Set{Class(Graphique)}

COMPOSITES : Bag{Set{Class(Images)}}

FEUILLES : Bag{Set{Class(Rectangles),Class(Textes),Class(Lignes)}}



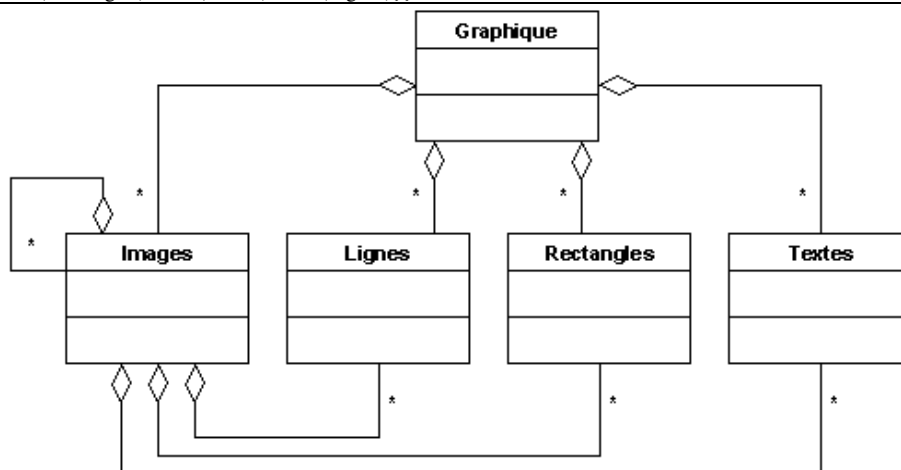
COMPOSITE, cas 2

```
inv composant :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 1) -> select
    (associations->select(connection -> select(aggregation=1).participant
    = connection->select(aggregation=2 or aggregation=3).participant) ->
    isEmpty()).oclAsType(Classifier)->asSet
  in
    composant -> fail('COMPOSANTS : ')
inv composite :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 1) -> select
    (associations->select(connection -> select(aggregation=1).participant
    = connection->select(aggregation=2 or aggregation=3).participant) ->
    isEmpty()).oclAsType(Classifier)->asSet
  let composite : Bag(Set(Classifier)) = composant -> iterate(c : Classifier;
    res: Bag(Set(Classifier)) = Bag{} | res->including(c.association ->
    select(aggregation=2 or aggregation=3).association.connection ->
    select(aggregation=1).participant.association->select(aggregation=2 or
    aggregation=3) -> select(association.connection -> select(aggregation
    = 1).participant = association .connection -> select(aggregation=2 or
    aggregation=3).participant).participant.oclAsType(Classifier)->asSet))
  in
    composite -> fail('COMPOSITES : ')
inv feuille :
  let composant : Set(Classifier) = Class.allInstances->select(association ->
    select(aggregation=2 or aggregation=3)->size > 1) -> select
    (associations -> select(connection -> select(aggregation=1).
    participant = connection->select(aggregation=2 or aggregation
    =3).participant)->isEmpty()).oclAsType(Classifier)->asSet
  let composite(composant_courant : Classifier) : Set(Classifier) =
    composant_courant.association->select(aggregation=2 or aggregation=3.
    association.connection -> select(aggregation=1).participant.
    association->select(aggregation=2 or aggregation=3) -> select
    (association. connection -> select(aggregation=1).participant =
    association.connection-> select(aggregation=2 or aggregation=3).
    participant).participant.oclAsType(Classifier)->asSet
  let feuille : Bag(Set(Classifier)) = composant -> iterate(c : Classifier;
    res : Bag(Set(Classifier)) = Bag{} | res->including(c.association ->
    select(aggregation=2 or aggregation=3).association.connection ->
    select(aggregation=1).participant.oclAsType(Classifier)->asSet
    -
    composite(c))
  in
    feuille -> fail('FEUILLES : ')
```

COMPOSANTS : Set{Class(Graphique)}

COMPOSITES : Bag{Set{Class(Images)}}

FEUILLES : Bag{Set{Class(Rectangles),Class(Textes),Class(Lignes)}}



COMPOSITE, cas 3

```

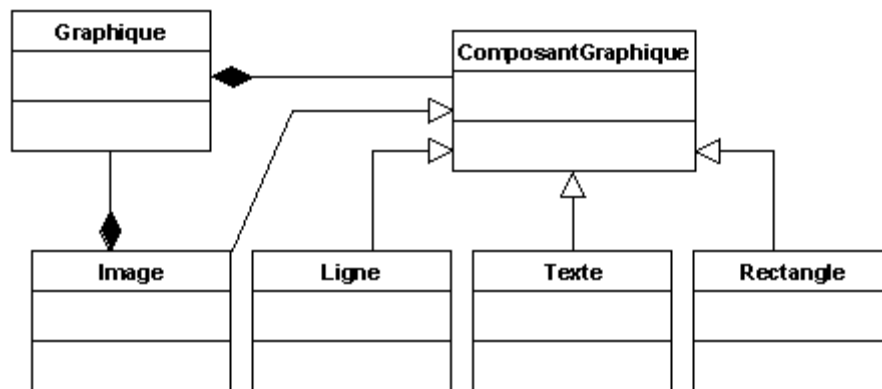
inv composant :
  let composant : Set(Classifier) = Class.allInstances.association->select
    (aggregation=2 or aggregation=3).participant.associations.connection->
    select (aggregation=1).participant->asset->select (specialization.child
    -> notEmpty).oclAsType(Classifier)->asSet
  in
    composant -> fail('COMPOSANT : ')
inv composite :
  let composant : Set(Classifier) = Class.allInstances.association->select
    (aggregation=2 or aggregation=3).participant.associations.connection->
    select (aggregation=1).participant->asSet->select (specialization.child
    -> notEmpty).oclAsType(Classifier)->asSet
  let composite : Bag(Set(Classifier)) = composant -> iterate(c : Classifier;
  res : Bag(Set(Classifier)) = Bag{} | res->including(c.specialization.
  child. oclAsType(Classifier).association -> select (aggregation=2 or
  aggregation=3).participant -> asSet -> select (Class.allInstances.
  association -> select (aggregation=2 or aggregation=3).participant.
  associations.connection -> select (aggregation=1).participant->asSet ->
  includesAll(associations.connection -> select (aggregation=1).
  participant)).oclAsType(Classifier)->asSet)
  in
    composite -> fail('COMPOSITE : ')
inv feuille :
  let composant : Set(Classifier) = Class.allInstances.association->select
    (aggregation=2 or aggregation=3).participant.associations.connection->
    select (aggregation=1).participant->asSet->select (specialization.child
    -> notEmpty).oclAsType(Classifier)->asSet
  let composite(composant_courant : Classifier) : Set(Classifier) =
    composant_courant.specialization.child.oclAsType(Classifier).
    association-> select (aggregation=2 or aggregation=3).participant ->
    asSet->select (Class.allInstances.association -> select (aggregation=2
    or aggregation=3).participant.associations.connection -> select (
    aggregation=1).participant->asSet -> includesAll(associations.
    connection->select (aggregation=1).participant)).oclAsType(Classifier)
    ->asSet
  let feuille : Bag(Set(Classifier)) = composant -> iterate(c : Classifier;
  res : Bag(Set(Classifier)) = Bag{} | res->including(c.specialization.
  child. oclAsType(Classifier)->asSet
  -
  composite(c))
  in
    feuille -> fail('FEUILLES : ')

```

COMPOSANT : Set{Class(ComposantGraphique)}

COMPOSITE : Bag{Set{Class(Image)}}

FEUILLES : Bag{Set{Class(Ligne),Class(Texte),Class(Rectangle)}}



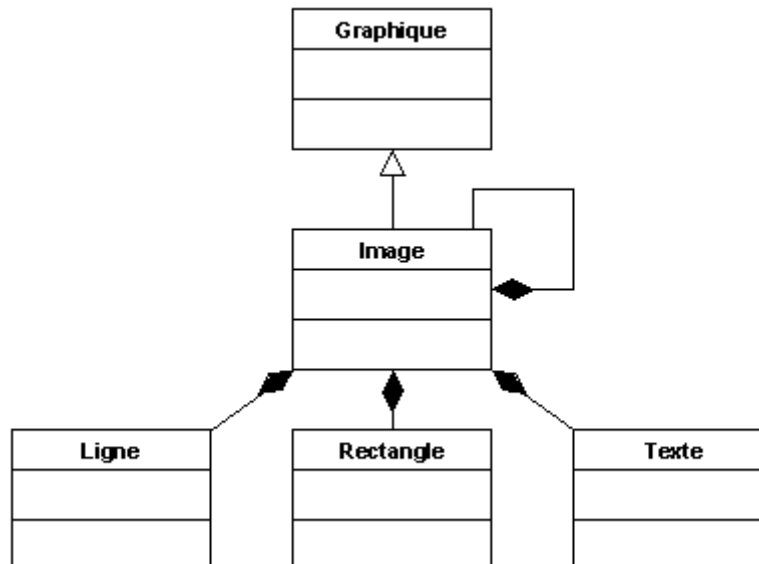
COMPOSITE, cas 4

```
inv composant :
  let composite : Set(Classifier) = Class.allInstances.association->select
    (aggregation=2 or aggregation=3).participant.associations.connection->
    select(aggregation=1).participant->asSet->intersection(Class.
    allInstances.association -> select(aggregation=2 or aggregation=3).
    participant).oclAsType(Classifier)->asSet
  let composant : Bag(Set(Classifier)) = composite -> iterate(c : Classifier;
  res: Bag(Set(Classifier)) = Bag{} | res->including(c.parent.oclAsType
  (Classifier)->asSet))
  in
    composant -> fail('COMPOSANTS : ')
inv composite :
  let composite : Set(Classifier) = Class.allInstances.association->select
    (aggregation=2 or aggregation=3).participant.associations.connection->
    select(aggregation=1).participant->asSet->intersection(Class.
    allInstances.association -> select(aggregation=2 or aggregation=3).
    participant).oclAsType(Classifier)->asSet
  in
    composite -> fail('COMPOSITES : ')
inv feuilles :
  let composite : Set(Classifier) = Class.allInstances.association->select
    (aggregation=2 or aggregation=3).participant.associations.connection->
    select(aggregation=1).participant->asSet->intersection(Class.
    allInstances.association -> select(aggregation=2 or aggregation=3).
    participant).oclAsType(Classifier)->asSet
  let feuille : Bag(Set(Classifier)) = composite -> iterate(c : Classifier;
  res : Bag(Set(Classifier)) = Bag{} | res->including(c.associations.
  connection->select(aggregation=1).participant->asSet - Set{c}.
  oclAsType(Classifier)->asSet))
  in
    feuille -> fail('FEUILLES : ')
```

COMPOSANTS : Bag{Set{Class(Graphique)}}

COMPOSITES : Set{Class(Image)}

FEUILLES : Bag{Set{Class(Ligne),Class(Texte),Class(Rectangle)}}



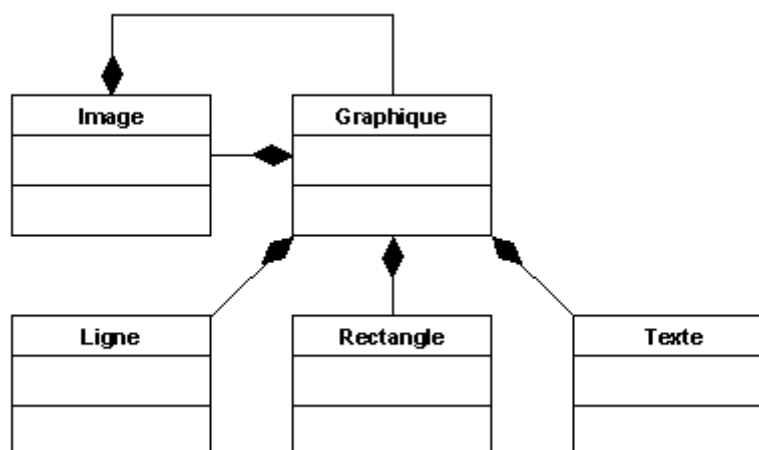
COMPOSITE, cas 5

```
inv composant :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 2).oclAsType
    (Classifier)->asSet
  in
    composant -> fail('COMPOSANTS : ')
inv composite :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 2).oclAsType
    (Classifier)->asSet
  let composite : Bag(Set(Classifier)) = composant -> iterate(c : Classifier;
    res: Bag(Set(Classifier)) = Bag{} | res->including(c.association->
    select (aggregation=2 or aggregation=3).association.connection ->
    select (aggregation=1).participant -> select(association -> select
    (aggregation=2 or aggregation=3).association.connection -> select
    (aggregation=1).participant -> includesAll(Set{c})).oclAsType
    (Classifier)->asSet))
  in
    composite -> fail('COMPOSITES : ')
inv feuille :
  let composant : Set(Classifier) = Class.allInstances->select(association->
    select(aggregation=2 or aggregation=3)->size > 2).oclAsType
    (Classifier)->asSet
  let composite(composant_courant : Classifier) : Set(Classifier) =
    composant_courant.association->select(aggregation=2 or aggregation=3).
    association.connection -> select(aggregation=1).participant -> select
    (association -> select(aggregation=2 or aggregation=3).
    association.connection -> select(aggregation=1).participant->
    includesAll(Set{composant_courant})).oclAsType(Classifier)->asSet
  let feuille : Bag(Set(Classifier)) = composant -> iterate(c : Classifier;
    res : Bag(Set(Classifier)) = Bag{} | res->including((c.association->
    select(aggregation=2 or aggregation=3).association.connection->select
    (aggregation=1).participant->asSet-composite(c)).oclAsType(Classifier)
    ->asSet))
  in
    feuille -> fail('FEUILLES : ')
```

COMPOSANTS : Set{Class(Graphique)}

COMPOSITES : Bag{Set{Class(Image)}}

FEUILLES : Bag{Set{Class(Ligne),Class(Rectangle),Class(Texte)}}



BRIDGE, cas 1

```

inv detect_bridge :
  let classes_plusieurs_assoc : Set(Classifier) = Class.allInstances->select
    (associations->size >= 2)
  let detect_bridge : Bag(Integer) = classes_plusieurs_assoc->iterate(c :
    Classifier;res:Bag(Integer) = Bag{} | res->including((c.associations.
    connection.participant->asSet - Set{c}).parent->asSet->size))
  in
    detect_bridge -> fail('DETECTION SI 1 : ')
inv abstraction_fine :
  let classes_plusieurs_assoc : Set(Classifier) = Class.allInstances->select
    (associations->size >= 2)
  let abstraction_fine : Bag(Set(Classifier)) = classes_plusieurs_assoc->
    iterate(c : Classifier;res:Bag(Set(Classifier)) = Bag{} |
    res->including((c.associations.connection.participant->asSet - Set{c})
    .oclAsType(Classifier)->asSet))
  in
    abstraction_fine -> fail('ABSTRACTION FINE : ')
inv abstraction :
  let classes_plusieurs_assoc : Set(Classifier) = Class.allInstances->select
    (associations->size >= 2)
  let abstraction : Bag(Set(Classifier)) = classes_plusieurs_assoc->iterate(
    c : Classifier;res:Bag(Set(Classifier)) = Bag{} | res->including(
    (c.associations.connection.participant->asSet - Set{c}).parent.
    oclAsType(Classifier)->asSet))
  in
    abstraction -> fail('ABSTRACTION : ')
inv implementeur :
  let classes_plusieurs_assoc : Set(Classifier) = Class.allInstances->select
    (associations->size >= 2)
  let implementeur : Bag(Set(Classifier)) = classes_plusieurs_assoc->iterate(
    c : Classifier;res:Bag(Set(Classifier)) = Bag{} | res->including(
    (Set{c} - (c.associations.connection.participant->asSet - Set{c}).
    parent.oclAsType(Classifier)->asSet).oclAsType(Classifier)->asSet))
  in
    implementeur -> fail('IMPLEMENTEUR : ')
inv implementeur_concret :
  let classes_plusieurs_assoc : Set(Classifier) = Class.allInstances->select
    (associations->size >= 2)
  let implementeur(courant : Classifier) : Set(Classifier) = (Set{courant} -
    (courant.associations.connection.participant->asSet - Set{courant}).
    parent.oclAsType(Classifier)->asSet).oclAsType(Classifier)->asSet
  let implementeur_concret : Bag(Set(Classifier)) = classes_plusieurs_assoc->
    iterate(c : Classifier;res:Bag(Set(Classifier)) = Bag{} | res->
    including(implementeur(c).specialization.child.oclAsType(Classifier)->
    asSet))
  in
    implementeur_concret -> fail('IMPLEMENTEUR_CONCRET : ')

```

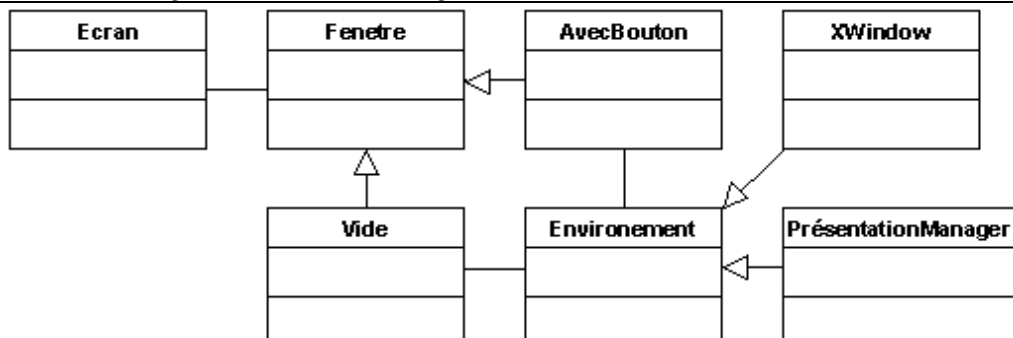
DETECTION SI 1 : Bag{1}

ABSTRACTION FINE : Bag{Set{Class(Vide),Class(AvecBouton)}}

ABSTRACTION : Bag{Set{Class(Fenetre)}}

IMPLEMENTEUR : Bag{Set{Class(Environnement)}}

IMPLEMENTEUR_CONCRET : Bag{Set{Class(PrésentationManager),Class(XWindow)}}



BRIDGE, cas 2

```

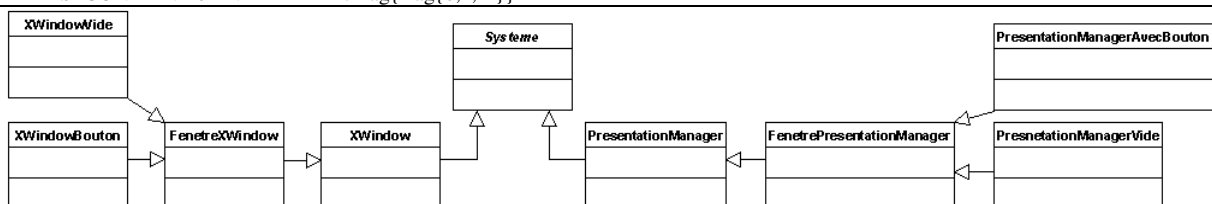
inv racine :
  let racine : Set(Classifier) = Class.allInstances->select(isAbstract).
    oclAsType(Classifier)->asSet
  in
    racine->fail('RACINES :')
inv premier_niveau :
  let racine : Set(Classifier) = Class.allInstances->select(isAbstract).
    oclAsType(Classifier)->asSet
  let premier_niveau : Bag(Set(Classifier)) = racine->iterate(c:Classifier;
    res : Bag(Set(Classifier)) = Bag{} | if (c.specialization.child->size
    > 1) then
    res->including(c.specialization.child.oclAsType(Classifier)->asSet)
    else
    res->including(Set{})
    endif)
  in
    premier_niveau->fail('PREMIER NIVEAU :')
inv symetrie :
  let nombre_filles(depart : Set(Classifier)) : Bag(Integer) = depart->iterate
    (c : GeneralizableElement; res : Bag(Integer) = Bag{} | res->including
    (c.specialization.child->size))
  let filles_identiques(depart : Set(Classifier)) : Integer =
    if (depart->asSet->size = 1 and depart->asSet->includes(0)) then
      0
    else
      depart->asSet->size
    endif
  let descente_filles(depart : Set(Classifier), valeur_prec : Integer,
    res : Bag(Integer)) : Bag(Integer) =
    if (filles_identiques(nombre_filles(depart)) = 0) then
      res->including(valeur_prec)
    else
      if (filles_identiques(nombre_filles(depart)) > 1) then
        res->including(filles_identiques(nombre_filles(depart)))
      else
        descente_filles(depart.specialization.child.oclAsType
          (Classifier)->asSet, filles_identiques(nombre_filles
            (depart)), res)->including(valeur_prec)
      endif
    endif
  let racine : Set(Classifier) = Class.allInstances->select(isAbstract).
    oclAsType(Classifier)->asSet
  let premier_niveau(racine : Classifier) : Set(Classifier) =
    if (racine.specialization.child->size > 1) then
      racine.specialization.child.oclAsType(Classifier)->asSet
    else
      Set{}
    endif
  let symetrie : Bag(Bag(Integer)) = racine->iterate(c:Classifier; res :
    Bag(Bag(Integer)) = Bag{} | res->including(descente_filles
    (premier_niveau(c), -1, Bag{})))
  in
    symetrie->fail('SYMETRIE SI CONTIENT 0 EN PREMIER : ')

```

RACINES :Set{Class(Systeme)}

PREMIER NIVEAU :Bag{Set{Class(XWindow),Class(PresentationManager)}}

SYMETRIE SI CONTIENT 0 EN PREMIER : Bag{Bag{0,1,-1}}



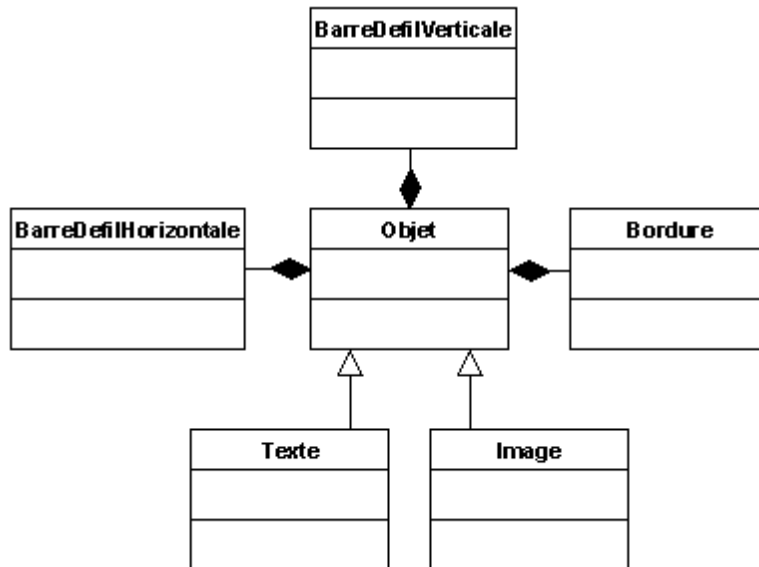
DECORATOR, cas 1

```
inv composant :
  let composant : Set(Classifier) = Class.allInstances->select(c |
    c.specialization.child->notEmpty and c.association->select(
      aggregation=2 or aggregation=3)->size>1).oclAsType(Classifier)->asSet
  in
    composant->fail('COMPOSANTS : ')
inv composant_concret :
  let composant : Set(Classifier) = Class.allInstances->select(c |
    c.specialization.child->notEmpty and c.association->select(
      aggregation=2 or aggregation=3)->size>1).oclAsType(Classifier)->asSet
  let composant_concret : Bag(Set(Classifier)) = composant->iterate
    (c : Classifier; res : Bag(Set(Classifier)) = Bag{} | res->including
      (c.specialization.child.oclAsType(Classifier)->asSet))
  in
    composant_concret->fail('COMPOSANT CONCRET : ')
inv decorateur_concret :
  let composant : Set(Classifier) = Class.allInstances->select(c |
    c.specialization.child->notEmpty and c.association->select(
      aggregation=2 or aggregation=3)->size>1).oclAsType(Classifier)->asSet
  let decorateur_concret : Bag(Set(Classifier)) = composant->iterate
    (c:Classifier; res : Bag(Set(Classifier)) = Bag{} | res->including
      (c.associations.connection->select(aggregation = 1).participant.
        oclAsType(Classifier)->asSet))
  in
    decorateur_concret->fail('DECORATEUR CONCRET : ')
```

COMPOSANTS : Set{Class(Objet)}

COMPOSANT CONCRET : Bag{Set{Class(Texte),Class(Image)}}

DECORATEUR CONCRET : Bag{Set{Class(BarreDefilHorizontale),Class(Bordure),Class(BarreDefilVerticale)}}



DECORATOR, cas 2

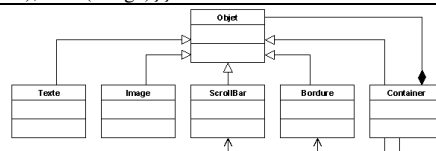
```
inv composant :
  let composant : Set(Classifier) = Class.allInstances->select(c | c.
    specialization.child->notEmpty and c.specialization.child->includesAll
    (c.associations.connection->select(aggregation = 2 or aggregation = 3)
    .participant)).oclAsType(Classifier)->asSet
  in
    composant->fail('COMPOSANTS : ')
inv decorateur :
  let composant : Set(Classifier) = Class.allInstances->select(c | c.
    specialization.child->notEmpty and c.specialization.child->includesAll
    (c.associations.connection->select(aggregation = 2 or aggregation = 3)
    .participant)).oclAsType(Classifier)->asSet
  let decorateur : Bag(Set(Classifier)) = composant->iterate(c:Classifier;
    res : Bag(Set(Classifier)) = Bag{} | res->including(c.associations.
    connection->select(aggregation = 2 or aggregation = 3).participant->
    select(t | c.specialization.child->includes(t)).oclAsType(Classifier)
    ->asSet))
  in
    decorateur->fail('DECORATEURS : ')
inv decorateur_concret :
  let composant : Set(Classifier) = Class.allInstances->select(c | c.
    specialization.child->notEmpty and c.specialization.child->includesAll
    (c.associations.connection->select(aggregation = 2 or aggregation = 3)
    .participant)).oclAsType(Classifier)->asSet
  let decorateur(depart : Classifier) : Set(Classifier) = depart.associations.
    connection->select(aggregation = 2 or aggregation = 3).participant->
    select(t | depart.specialization.child->includes(t)).oclAsType
    (Classifier)->asSet
  let decorateur_concret : Bag(Set(Classifier)) = composant->iterate(c:
    Classifier;res : Bag(Set(Classifier)) = Bag{} | res->including
    (decorateur(c).associations->select(c.specialization.child->
    includesAll(connection.participant)).connection.participant.oclAsType
    (Classifier)->asSet - decorateur(c))
  in
    decorateur_concret->fail('DECORATEURS CONCRETS : ')
inv composant_concret :
  let composant : Set(Classifier) = Class.allInstances->select(c | c.
    specialization.child->notEmpty and c.specialization.child->includesAll
    (c.associations.connection->select(aggregation = 2 or aggregation = 3)
    .participant)).oclAsType(Classifier)->asSet
  let decorateur(depart : Classifier) : Set(Classifier) = depart.associations.
    connection->select(aggregation = 2 or aggregation = 3).participant->
    select(t | depart.specialization.child->includes(t)).oclAsType
    (Classifier)->asSet
  let decorateur_concret(depart : Classifier) : Set(Classifier) = decorateur
    (depart).associations->select(depart.specialization.child->
    includesAll(connection.participant)).connection.participant.oclAsType
    (Classifier)->asSet - decorateur(depart)
  let composant_concret : Bag(Set(Classifier)) = composant->iterate
    (c:Classifier;res : Bag(Set(Classifier)) = Bag{} | res->including(c.
    specialization.child.oclAsType(Classifier)->asSet -
    decorateur_concret(c) - decorateur(c))
  in
    composant_concret->fail('COMPOSANTS CONCRETS : ')
```

COMPOSANTS : Set{Class(Objet)}

DECORATEURS : Bag{Set{Class(Container)}}

DECORATEURS CONCRETS : Bag{Set{Class(ScrollBar),Class(Bordure)}}

COMPOSANTS CONCRETS : Bag{Set{Class(Texte),Class(Image)}}



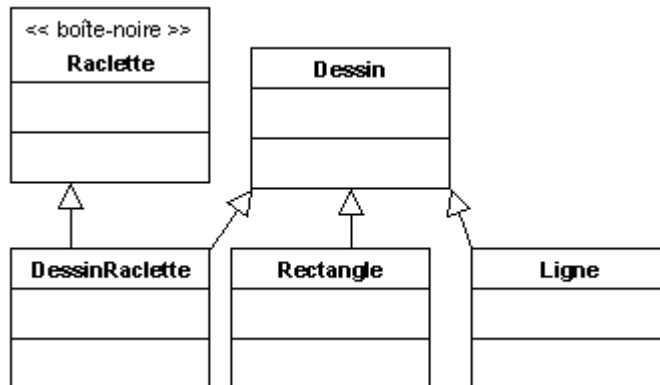
ADAPTER, cas 1

```
inv adapter :
  let adapter : Set(Classifier) = Class.allInstances->select(parent->size > 1)
    .oclAsType(Classifier)->asSet
  in
    adapter->fail('ADAPTATEURS : ')
inv adapte :
  let adapter : Set(Classifier) = Class.allInstances->select(parent->size > 1)
    .oclAsType(Classifier)->asSet
  let adapte : Bag(Set(Classifier)) = adapter->iterate(c : Classifier; res :
    Bag(Set(Classifier)) = Bag{} | res->including(c.parent.oclAsType
    (Classifier)->asSet))
  in
    adapte->fail('ADAPTES : ')
inv nombre :
  let adapter : Set(Classifier) = Class.allInstances->select(parent->size > 1)
    .oclAsType(Classifier)->asSet
  let adapte : Bag(Set(Classifier)) = adapter->iterate(c : Classifier;
    res : Bag(Set(Classifier)) = Bag{} | res->including(c.parent.oclAsType
    (Classifier)->asSet))
  let nombre : Bag(Bag(Integer)) = adapte->iterate(c : Set(Classifier);
    res : Bag(Bag(Integer)) = Bag{} | res->including(c->iterate(e :
    Classifier; nbd : Bag(Integer) = Bag{} | nbd->including(e.
    specialization.child->size))))
  in
    nombre->fail('ADAPTE A LE MOINS : ')
```

ADAPTATEURS : Set{Class(DessinRaclette)}

ADAPTES : Bag{Set{Class(Raclette),Class(Dessin)}}

ADAPTE A LE MOINS : Bag{Bag{1,3}}



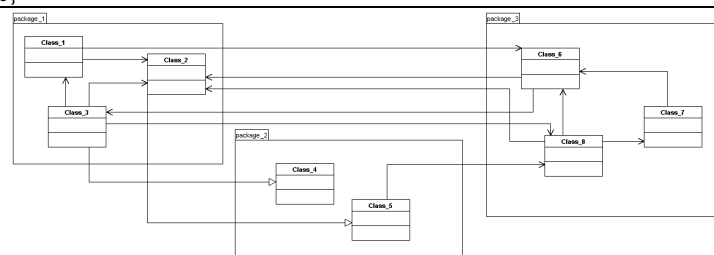
FACADE, cas 1

```
inv paquetage :
  let paquetages : Set(Package) = Package.allInstances->select(oclIsTypeOf
    (Package))
  let paquetages_concernes : Set(Package) = paquetages->iterate(p : Package;
    res : Set(Package) = Set{} | if (p.allContents->select(oclIsTypeOf
    (Class))->size/2 <= p.allContents->select(oclIsTypeOf(Class)).
    oclAsType(Class).associations->select(a | (p.allContents->select(
    oclIsTypeOf(Class)).oclAsType(Classifier)->asSet->includesAll
    (a.connection.participant->asSet)))->asSet->size) then
      res->including(p)
    else
      res
    endif)
  in
    paquetages_concernes->fail('PAQUETAGES CONCERNES :')

inv facade :
  let paquetages : Set(Package) = Package.allInstances->select(oclIsTypeOf
    (Package))
  let paquetages_concernes : Set(Package) = paquetages->iterate(p : Package;
    res : Set(Package) = Set{} | if (p.allContents->select(oclIsTypeOf
    (Class))->size/2 <= p.allContents->select(oclIsTypeOf(Class)).
    oclAsType(Class).associations->select(a | (p.allContents->select(
    oclIsTypeOf(Class)).oclAsType(Classifier)->asSet->includesAll
    (a.connection.participant->asSet)))->asSet->size) then
      res->including(p)
    else
      res
    endif)
  let classifieurs(paquetage_en_cours : Package) : Set(Classifier) =
    paquetage_en_cours.allContents->select(oclIsTypeOf(Class)).oclAsType
    (Classifier)->asSet
  let classifieurs_lies_ext(paquetage_en_cours : Package) : Set(Classifier) =
    classifieurs(paquetage_en_cours)->select(associations->select(a | not
    (paquetage_en_cours.allContents->select(oclIsTypeOf(Class)).oclAsType
    (Classifier)->asSet->includesAll(a.connection->select(not isNavigable)
    .participant->asSet)))->notEmpty)
  let associations_ext(paquetage_en_cours : Package) : Set(Association) =
    classifieurs(paquetage_en_cours).associations->select(a | not
    (paquetage_en_cours.allContents->select(oclIsTypeOf(Class)).oclAsType
    (Classifier)->asSet->includesAll(a.connection->select(not isNavigable)
    .participant->asSet)))->asSet
  let classifieurs_exterieurs(paquetage_en_cours : Package) : Set(Classifier) =
    associations_ext(paquetage_en_cours).connection->select(not
    isNavigable).participant->asSet
  let need_facade(paquetage_en_cours : Package) : Boolean =
    classifieurs_exterieurs(paquetage_en_cours)->size < associations_ext
    (paquetage_en_cours)->size
  let executeur : Bag(Boolean) = paquetages_concernes->iterate(p : Package;
    res : Bag(Boolean) = Bag{} | res->including(need_facade(p)))
  in
    executeur->fail('BESOIN FACADE :')
```

PAQUETAGES CONCERNES :Set{Package(package_1),Package(package_3)}

BESOIN FACADE :Bag{true,false}



ANNEXE 2
Article pour le WorkShop sur les
patrons de méta-modèles

Des méta-modèles au banc d'essai des patrons de conception

(Ré)utilisation et Intégration

Cédric Bouhours, Hervé Leblanc

*IRIT – équipe MACAO (Modèles, Aspects et Composants pour des Architectures à Objets)
Université Paul Sabatier
118 Route de Narbonne
F-31062 TOULOUSE CEDEX 9
{bouhours,leblanc}@irit.fr*

RÉSUMÉ. Les méta-modèles jouent un double rôle. D'une part, ils structurent les modèles et ont un rôle conceptuel, et d'autre part, ils servent d'accès à l'information via des langages de requêtes et de transformations. Dans ce contexte, nous relatons une expérience faite sur deux méta-modèles, SPEM et le paquetage core de UML1.4, en y intégrant des patrons de conception, devenant de fait, candidats au statut de patron de méta-modèle.

ABSTRACT. Meta-models act a dual role. First, they structure models and have a conceptual role, on the other hand, they are used to access information via query languages and transformations. So, we report an experiment on two meta-models, SPEM and core of UML1.4, which integrate design patterns, becoming indeed, pretender to status of meta-model pattern.

MOTS-CLÉS: patrons de conception, méta-modèles, SPEM, UML 1.4.

KEYWORDS: design patterns, meta-model, SPEM, UML 1.4.

1. Un ensemble de règles OCL dédié aux patrons structurels

Les patrons de conception, tout comme les autres types de motifs, représentent un savoir-faire sous forme de microarchitecture de classes réutilisables. Les patrons de conception structurels permettent une conception simple en termes de structure (gestion des liens d'héritage et d'association) et élégante en termes de responsabilité entre classes. Afin de favoriser leur utilisation dans des conceptions existantes, nous nous sommes attachés à déterminer les conditions d'applications de chacun des patrons structurels proposés par (Gamma *et al.*, 1995). Pour cela, nous avons du trouver un ensemble de particularités structurelles qui, une fois repérées dans un modèle, ciblerait le fragment de modèle à transformer par l'injection d'un patron.

Un modèle alternatif à un patron est un modèle qui résout le même problème que le patron, mais dont la structure n'intègre pas la microarchitecture du patron. Il s'agit donc d'un modèle candidat à la substitution par un patron.

Pour obtenir ces alternatives, une expérience a été réalisée sur deux promotions d'élèves en cursus informatique, qui n'avaient à priori aucune connaissance sur ces patrons. Ils devaient modéliser, dans la notation UML, une série de sept problèmes solubles avec les sept patrons structurels. Sur les trois cent modèles obtenus, nous en avons sélectionné quarante qui constituaient des alternatives aux patrons, et après avoir retiré tous les doublons, il en est resté onze qui présentaient des différences structurelles significatives. Chaque modélisation obtenue (entre deux et cinq par patron) constituait donc une alternative plausible à un patron.

Chaque patron de conception structurel se caractérise par un rôle attribué à chacune des classes qui le compose, chaque rôle ayant une particularité structurelle propre. Chaque modèle alternatif se caractérise de la même façon qu'un patron. A chaque rôle du modèle alternatif est associé un ensemble de particularités structurelles dites remarquables. Celles-ci permettent de repérer dans un modèle, les classes pouvant jouer les différents rôles du patron. En associant, à chaque classe des modèles alternatifs, les rôles des patrons correspondant, et en étudiant leur structure (sous-arbres équivalents dans des hiérarchies d'héritages, répartition quantitative des associations et des compositions...), nous avons pu déduire ces particularités. Pour chaque modèle alternatif, il y a donc un ensemble de particularités remarquables associées à chacun des rôles du patron. Ces particularités sont pour le moment uniquement structurelles, la notion d'équivalence des interfaces des classes n'étant pas pour le moment dans la portée de ces travaux (Bouhours, 2006).

Ensuite, nous avons établi une méthode générique de recherche qui repère, dans le modèle à analyser, les classes pouvant jouer le rôle de référence sur le patron à identifier. Puis, en utilisant ces classes, la méthode tente d'affecter les autres rôles du patron aux autres classes du modèle liées à celles-ci. Enfin, nous avons automatisé cette détection à l'aide de règles OCL supportées par la plate-forme Neptune (Neptune, 2003).

2. Expérimentation sur des méta-modèles

Les méta-modèles UML étant instances du MOF, ils sont décrits par des diagrammes de classes, ce qui nous a permis d'y appliquer nos règles OCL. Nous les avons considérés alors comme des modèles aptes à l'intégration de patrons de conception. De ce fait, certains d'entre eux pourraient être considérés comme des patrons de méta-modèle.

D'après la spécification du langage OCL, une règle permet d'ajouter des contraintes à un diagramme UML, et de fait, d'affiner la sémantique de modèles ou de méta-modèles, comme proposé par *the precise UML group* quant aux méta-

modèles UML. Pour notre part, nous avons détourné OCL de son but initial pour rechercher des motifs structurels sur des modèles, en naviguant sur les liens du méta-modèle. Normalement, une règle renvoie un booléen qui indique si un diagramme respecte ou non la règle concernée. Dans notre cas, nous souhaitons que les règles OCL retournent des collections de classes correspondant aux différents rôles de chaque patron. La plate-forme Neptune ayant la particularité de retourner le contexte d'une erreur (ensemble des éléments du méta-modèle sur lequel était appliquée la règle), nous avons codé nos règles OCL de manière à ce qu'elles retournent toujours faux, et que le contexte de l'erreur corresponde à l'ensemble des classes qui nous intéressait.

Le résultat se présente sous la forme de plusieurs collections de classes, chacune correspondant à un rôle du patron. Par exemple, le résultat de l'application des règles concernant le patron *Composite* se présente sous la forme de trois ensembles. Ils contiennent respectivement les classes pouvant jouer les rôles de « Composant », « Composite » et « Feuille ».

Sur les sept patrons structurels du GOF, nous en avons éliminé quatre :

- Façade, car les liens de dépendances entre les paquetages des méta-modèles de l'OMG sont en général matérialisés par des liens d'héritages, suggérant une architecture logicielle en couches.

- Pont, car les liaisons entre abstraction et implémentation n'ont pas à être définies à ce niveau de modélisation.

- Procuration et Poids mouche, car ils ne sont pas détectables structurellement. Les problèmes qu'ils résolvent émanent d'exigences non fonctionnelles.

L'application des règles concernant les trois patrons restants Composite, Décorateur et Adaptateur, nous a permis de cibler des fragments de méta-modèles à partir desquels nous proposons la discussion suivante. Notre expérimentation a porté sur le paquetage core de UML1.4 et sur le méta-modèle SPEM, pour éviter l'explosion combinatoire des possibilités d'intégration de patrons, et pour faciliter la vérification de notre hypothèse.

3. Composite et Décorateur intégrés dans le SPEM

Le SPEM (Software Process Engineering Metamodel) est un méta-modèle spécifique aux processus de développement proposé par l'OMG (OMG, 2005). Le noyau se compose en particulier des classes WorkDefinition (découpage du processus en activité élémentaire, itération, phase et cycle de vie), ProcessRole (entité reliant des compétences requises à des activités) et WorkProduct (paramètre en entrée-sortie des activités). La figure 1 montre le fragment ciblé par nos règles. Elles font apparaître deux patrons de conception susceptibles d'être intégrés : le patron Composite (relation réflexive subwork sur la super classe WorkDefinition) et le patron Décorateur sur le même ensemble de classes.

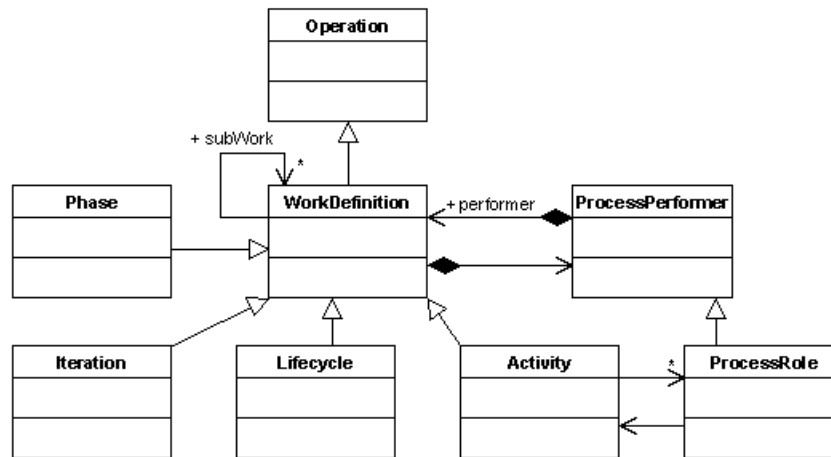


Figure 1. Fragment du SPEM 1.1

Une analyse plus fine nous permet de constater que la classe WorkDefinition joue plusieurs rôles : les rôles « Composite » et « Composant » du patron Composite, et le rôle « Décorateur » du patron Décorateur. La classe Activity ne pouvant se décomposer qu'en travaux atomiques nommés Step, elle joue nécessairement le rôle « Feuille » du patron Composite. Les classes Lifecycle, Phase et Iteration peuvent jouer le rôle « Décorateurs concrets » : Phase et Iteration ajoutent les notions de temps et de buts à atteindre, Lifecycle spécialise la relation subWork aux instances de Phase et relie celles-ci aux processus et composants de processus considérés comme des référentiels d'activités.

La figure 2 montre le fragment du méta-modèle SPEM, enrichie d'une intégration explicite des deux patrons Composite et Décorateur. En intégrant la classe WorkComponent, jouant explicitement le rôle de « Composant », nous simplifions la définition de la classe WorkDefinition jouant uniquement les rôles « Composite » et « Décorateur ».

L'intégration de ces deux patrons de conception a deux effets : d'une part elle clarifie les possibilités de chacune des classes de ce méta-modèle, en leur attribuant des rôles précis. D'autre part, elle permet de préciser la sémantique du méta-modèle sans ajouter de *well formedness rules* supplémentaires ou de descriptions en langage naturel. Par exemple, le rôle « Feuille » de la classe Activity permet de se passer des précisions suivantes (OMG, §7-3, p7-5, 2005) : « *Although this is not explicitly prohibited, an Activity does not normally use the subWork structure inherited from WorkDefinition; instead decomposition within Activity is done using Steps* ». En revanche, des contraintes de composabilité entre les différents décorateurs concrets restent nécessaires.

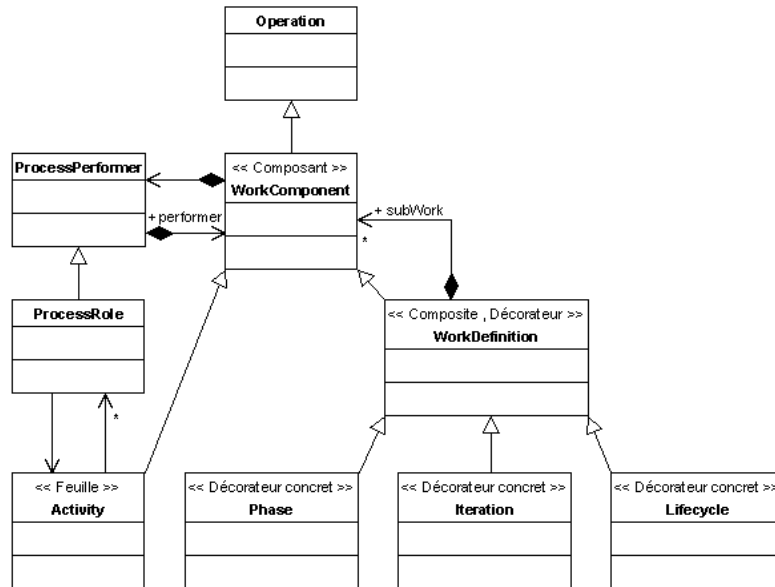


Figure 2. Intégration des patrons Composite et Décorateur dans le SPEM 1.1

4. Composite et Adaptateur pour simplification du paquetage core de UML 1.4

Lors de l'application des règles OCL sur le noyau du méta-modèle UML 1.4, nous obtenons un patron Composite sur les classes Namespace (rôle « Composite ») et ModelElement (rôle « Composant »). A priori, la classe GeneralizableElement est « Feuille », mais pour des raisons conceptuelles, les classes Classifier et Package sont en même temps « Feuille » (hérite de GeneralizableElement) et « Composite » (hérite de Namespace). Il existe donc une double structure d'héritage multiple et répété intégrée dans un patron Composite.

Nous proposons alors une simplification de conception en intégrant le patron Adaptateur, chargé d'implémenter l'interface de GeneralizableElement pour l'ensemble de ses anciennes sous classes, comme cela est suggéré à la figure 3.

5. Conclusion

Cette détection, prévue au départ pour des modèles UML, nous a permis de détecter des patrons dit de conception sur des méta-modèles de l'OMG. Nous considérons que les patrons Composite et Décorateur sont des candidats au statut de patrons de méta-modèle.

Afin d'étendre la discussion, il nous semblerait intéressant d'étudier le statut des autres patrons de conception. Si nous éliminons les patrons créateurs, chargés de gérer l'instanciation d'objets complexes (agrégats, frameworks), il subsiste certains patrons comportementaux candidats : le patron Interprète, servant de support au méta-modèle dédié aux langages spécifiques de domaines (DSL), le patron Chaîne de responsabilité, chargé d'explicitier les liens de navigation au travers des classes du méta-modèle et le patron Médiateur simplifiant la navigation dans les modèles. De plus, le patron structurel Pont peut être réutilisé pour la séparation propre entre les éléments de modélisation et les représentations graphiques associées. Pour terminer, deux questions nous semblent intéressantes à débattre. Les patrons métier sont-ils candidats à être des patrons de méta-modèles dédiés ? Les patrons de méta-modèles peuvent-ils être structurés à l'image des patrons composite (Riehle, 1997) ?

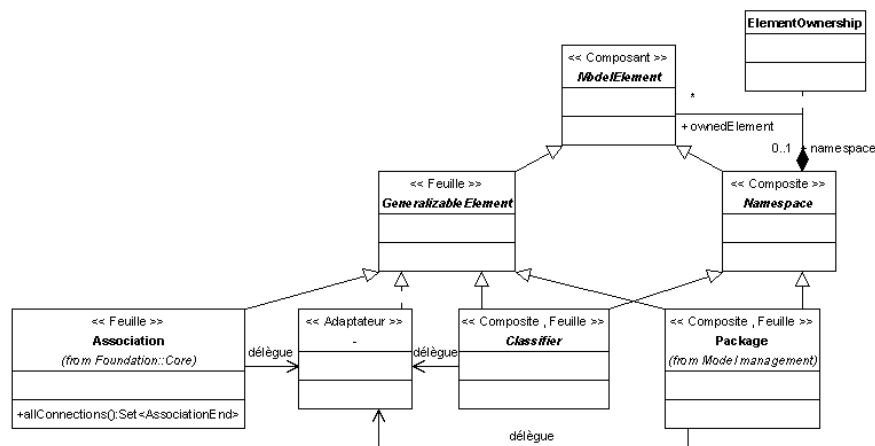


Figure 3. Intégration d'un Adaptateur dans un fragment du noyau de UML 1.4

6. Bibliographie

- Bouhours C., Détection de particularités structurelles de modèles incitant à l'injection de patrons de conception, Master de recherche, Université Paul Sabatier, Toulouse, 2006.
- Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1995.
- Neptune, Nice Environment with a Process and Tools using Norms - UML, XML and XMI - and Example. [w] <http://neptune.irit.fr>, 2003.
- Object Management Group, Software Process Engineering Metamodel Specification, version 1.1 formal/05-01-06, 2005.
- Riehle D. « Composite Design Patterns », *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA '97*. ACM Press, 1997. Page 218-228.