Table des légendes

Table des légendes

Introduction générale

*"It's a book of **design patterns** that describes simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time. Hence they aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form."*

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995)

Contexte et Problématique

Le génie logiciel est un domaine qui regroupe l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi [JOfficiel87]. De nouvelles avancées émergent dans ce domaine, notamment l'ingénierie dirigée par les modèles qui tend à accorder une place prépondérante et systématique aux modèles. Modéliser un système consiste à représenter de manière simplifiée un aspect de la réalité pour un objectif donné. Loin de se réduire à l'expression d'une solution à un niveau d'abstraction plus élevé que le code, la modélisation peut être vue comme la séparation des différents besoins fonctionnels et non fonctionnels issus des exigences. Couplée à une approche par objets, la modélisation d'un logiciel permet d'atteindre les concepts de modularisation, de lisibilité, de compréhension, de réutilisation et d'extensibilité. C'est ainsi que pour donner aux modèles un caractère plus consistant et plus évaluable, les processus dirigés par les modèles se doivent d'être capables de prendre en compte le savoir-faire d'experts, généralement exprimé en termes de *patron*. Ces patrons ont été éprouvés et validés par une communauté pour résoudre des problèmes spécifiques et représentent un consensus de la solution la plus efficace. Utiliser le patron adéquat, lors d'une analyse ou lors d'une conception, est le gage de réutiliser la pratique considérée comme la plus efficace.

Un concepteur désireux d'utiliser un patron de conception dans son modèle a à sa disposition un catalogue (le GoF), qui lui présente un ensemble de patrons de conception pour résoudre certains types de problèmes. Cependant, utiliser ce catalogue n'est pas toujours une chose aisée pour un concepteur. En effet, il lui est nécessaire de trouver quel patron est le plus adapté à son problème et comment bien l'utiliser dans son modèle. Tout d'abord, pour choisir dans le GoF quel patron concerne le problème qu'il est en train de concevoir, le concepteur doit éliminer le contexte du problème, afin de le comparer à ceux présentés dans le GoF. En effet, dans ce catalogue, les problèmes sont génériques de manière à pouvoir s'adapter à n'importe quel contexte. Après avoir identifié le patron adéquat, le concepteur doit s'assurer qu'il a intégré correctement le patron dans son modèle, afin que cette intégration n'ait pas d'éventuelles conséquences négatives. En effet, si l'adaptation du patron au contexte du problème est mal effectuée, le modèle peut s'en trouver dégradé. Il est donc très important pour un concepteur de s'assurer qu'il a utilisé le patron comme il a été prévu ou qu'il gère les problèmes engendrés par l'emploi d'une forme

dégradée du patron. Ceci n'est pas une chose simple, surtout pour un concepteur utilisant pour la première fois un patron. Malgré les détails d'utilisation, la structure et les conseils inscrits avec chaque patron dans le GoF, leur utilisation est toujours subjective au contexte du problème et à l'expérience propre du concepteur.

Il existe de nombreux travaux visant à aider le concepteur dans le choix de son patron, tant au niveau de l'amélioration de la classification et de la description d'un patron qu'au niveau de méthodes permettant de vérifier qu'un patron a été correctement intégré à la modélisation. Ainsi, durant sa conception, un concepteur se trouve assisté lorsqu'il désire réutiliser le savoir-faire exprimé par les patrons.

Dans cette thèse, nous proposons une approche complémentaire visant à favoriser l'utilisation des patrons de conception dans les modèles, a posteriori, sans imposer au concepteur de chercher lesquels sont utiles à son problème et en lui expliquant en quoi l'utilisation de tel ou tel patron est plus pertinente pour son modèle.

Contribution

L'objectif de nos travaux consiste à offrir au concepteur, par l'exécution d'une activité de revue de conception, un moyen de vérifier, à l'issue de sa conception, si des défauts de conception pourraient être corrigés en utilisant des patrons de conception. Le cas échéant, nous lui permettons, après lui avoir fourni un ensemble d'explications, de transformer automatiquement son modèle pour y injecter les patrons concernés.

Pour identifier les zones d'un modèle où un patron de conception pourrait être utilisé, nous avons défini le concept de *patron abîmé* et mis en œuvre une méthode de détection structurelle. Un patron abîmé s'apparente à un patron de conception, dans le sens où il décrit une solution générique à un problème, mais avec une architecture différente de celle du patron conduisant à de mauvaises pratiques de conception. Notre technique de détection consiste à rechercher dans un modèle des fragments correspondant aux adaptations possibles de patrons abîmés dans un modèle.

Nous avons défini une ontologie que nous utilisons pour établir un dialogue avec le concepteur du modèle. L'ontologie relie chaque patron abîmé à son intention ainsi qu'aux propriétés qu'il dégrade. Ainsi, nous pouvons vérifier, directement auprès du concepteur, quelle intention exprime le fragment identifié, et nous pouvons lui présenter les avantages du remplacement du fragment par le patron de conception adéquat.

Le remplacement d'un fragment par le patron abîmé adapté au contexte du modèle est effectué par un ensemble d'opérations élémentaires de transformation. Ces commandes sont construites en comparant le patron de conception au patron abîmé. Elles sont incluses dans le patron abîmé et ne dépendent pas du contexte des modèles analysés.

Notre activité de revue de conception a été outillée pour pouvoir être intégrée dans un processus de développement. Cette activité permet, à partir d'un modèle en entrée, de détecter des fragments caractéristiques, de les présenter au concepteur et de les remplacer, avec l'approbation du concepteur, par des concrétisations de patrons de conception. Chaque cycle détection, explications et restructuration est exécuté tant qu'un patron abîmé est détecté. Chaque restructuration d'un patron abîmé provoque une nouvelle détection pour tenir compte de modifications effectuées sur le modèle par ladite restructuration. Cette activité permet ainsi de concrétiser l'intégralité de notre approche.

Structuration du mémoire

Le premier chapitre de ce mémoire se consacre à la description de notre problématique et à la définition des concepts mis en œuvre pour sa résolution. Nous y définissons ainsi le concept de patron abîmé, représentant une mauvaise pratique de conception invalidant une partie des points forts d'un patron de conception.

Le deuxième chapitre se concentre sur la méthode nous ayant permis de collecter des patrons abîmés. Ces derniers constituent un « catalogue de patrons abîmés », complétant le GoF. Nous détaillons ainsi les patrons abîmés que nous avons pu déduire et qui servent de base à notre processus de détection et de remplacement de défauts de conception.

Le troisième chapitre s'intéresse à notre technique de détection nous permettant d'identifier des fragments caractéristiques sans connaître a priori leurs formes. Grâce à un procédé de filtrages successifs, incluant des informations spécifiques telles que relations interdites entre les classes, participant de référence et points d'extension, nous sommes à même de détecter des fragments de modèles remplaçables par des patrons de conception.

Le quatrième chapitre présente la concrétisation de la revue de conception avec un outil nous permettant d'enchaîner détections, explications et transformations de modèles. Les requêtes utilisées pour détecter les fragments sont automatiquement générées à partir des patrons abîmés modélisés à l'aide d'un profil dédié. Le dialogue avec le concepteur s'appuie sur une ontologie permettant de récupérer aisément points forts et intention d'un patron abîmé.

1

Les patrons abîmés : contexte

Afin de garantir l'utilisation de bonnes pratiques d'analyse et de conception et une maintenance plus aisée des logiciels, les analystes et les concepteurs disposent entre autres de *patrons*. Ces patrons capitalisent l'expérience de ceux qui se sont déjà confrontés à certains types de problèmes et qui ont essayé, au fur et à mesure des tentatives, de les résoudre au mieux. Un patron est un consensus sur la solution la plus efficace pour résoudre un problème donné [Baroni03]. Utiliser un patron est le gage de réutiliser la solution la plus adéquate pour son problème et ainsi, garantir une qualité consensuelle à son analyse ou à sa conception.

Pour assister les concepteurs, le catalogue des patrons de conception du Gang of Four (GoF) [Gamma95] fournit un ensemble de solutions. Si un concepteur a le réflexe d'avoir recours au GoF lors de sa conception, nous considérons qu'il s'assure d'utiliser la meilleure microarchitecture de classes pour résoudre ses problèmes. Cependant, si des erreurs persistent, ou si le concepteur n'a pas l'habitude d'avoir recours aux patrons de conception, des défauts de conception peuvent perdurer dans les modèles. Pour limiter ou éviter ce risque, des aides à l'utilisation des patrons ont été conçues [Borne99] [ElBoussaidi04]. Les patrons ont été classifiés et décrits de plusieurs manières différentes pour aider à leur sélection [Albin-Amiot01_a] [Albin-Amiot01_b] [Baroni03] [Conte01] [Dietrich05] [Dong07] [Guennec00] [Kampffmeyer07] [Mak04], et des méthodes permettant la vérification de la bonne intégration d'un patron ont été établies [Eden97] [ElBoussaidi08] [France03] [Mili05] [O'Cinnéide99].

Pour un même problème de conception, nous considérons que plusieurs solutions différentes existent : celle reconnue comme étant la plus performante et la plus efficace, c'est-à-dire, utilisant correctement le patron de conception adéquat, puis les autres, certainement moins performantes et moins efficaces. Nous nous proposons de détecter et de corriger ces « autres » solutions par une activité de revue de conception outillée. Cette dernière a pour objectif d'inspecter les modèles à la recherche de fragments caractéristiques de mauvaises pratiques de conception et de les remplacer, après communication auprès du concepteur, par des patrons de conception. Nous considérons alors que nous recherchons des « patrons abîmés » dans des modèles. Un patron abîmé dégrade les qualités intrinsèques d'un patron de conception et a une microarchitecture telle qu'elle soit remplaçable par un patron de conception.

Nous présentons dans la première section de ce chapitre le concept de patron. Nous montrons qu'ils sont apparus dans le domaine de l'architecture et qu'ils ont été adaptés à de nombreux domaines, dont la conception de logiciels. Ils constituent un véritable moyen de transfert du savoir-faire de concepteurs expérimentés. En deuxième section, nous mettons en avant les difficultés de sélection et d'utilisation des patrons de conception. Nous y présentons certaines solutions actuelles, comme l'amélioration de la classification des patrons, et nous introduisons notre solution, consistant à rechercher des défauts de conception pour les remplacer par des patrons de conception. La troisième section de ce chapitre introduit les concepts clefs de notre approche : les patrons abîmés, les fragments

alternatifs et les procédés de contextualisation et de décontextualisation. Nous décrivons ainsi les fondements de notre activité de revue de conception, permettant de détecter, d'expliquer et de transformer des fragments de modèles particuliers en patrons de conception. Enfin, nous terminons en illustrant le fonctionnement de l'activité sur un exemple concret.

I. Les patrons

C'est un architecte du nom de Christopher Alexander qui a, le premier, étudié les patrons dans le bâtiment et les collectivités. Il a constaté qu'au fil des siècles, les grands bâtisseurs n'ont pas suivi de modèles préétablis avec des règles rigoureuses pour ériger leurs édifices, mais que les architectures avaient été adaptées les unes après les autres à leur environnement. Il a également remarqué, malgré leurs adaptations aux différentes situations, une certaine récurrence dans les solutions de conception considérées comme efficaces. Il a alors mis au point un langage de patrons permettant à des non-architectes de construire eux-mêmes leur propre édifice. Ce langage est structuré en un ensemble de patrons dont chacun permet de résoudre un problème. Selon lui, chaque patron concerne un problème qui se manifeste constamment dans notre environnement et décrit le cœur de la solution de ce problème, d'une façon telle qu'il est possible de réutiliser plusieurs fois cette solution sans jamais le faire deux fois de la même manière [Alexander77]. Le tableau 1.1 présente un des patrons d'Alexander.

Patron	Domaine du couple
Problème	Comment éviter que la présence d'enfants dans une famille ne détruise la proximité et l'intimité nécessaire à un homme et une femme ?
Solution	Construire une zone spéciale dans la maison distincte des zones communes et des chambres des enfants, où l'homme et la femme peuvent se retrouver en privé. Donner à cette zone un accès rapide à la chambre des enfants, mais, en conservant à tout prix, une séparation distincte.

Tableau 1.1 : Le patron *Domaine du couple* d'Alexander

Ce patron est intéressant de par la relative simplicité du problème qu'il résout. La première partie de la solution semble particulièrement évidente, puisqu'il suffit de créer une zone d'intimité séparée de la zone d'évolution des enfants pour résoudre le problème. Cependant, c'est la deuxième partie qui apporte toute la force à ce patron. Pour conserver une « contrainte » de surveillance permanente des enfants, le patron suggère de placer la zone d'intimité à proximité de la zone des enfants. Ainsi, la surveillance peut être maintenue tout en conservant un minimum d'intimité.

Dans cette section, nous présentons dans une première partie les patrons tels que le génie logiciel les utilise. Nous montrons que les patrons d'Alexander ont été déclinés suivant les différentes étapes du cycle de vie d'un logiciel, chaque type de patron étant décrit avec une base commune. Cette base permet de définir le contexte et la manière d'utiliser un patron. Nous nous intéressons dans une deuxième partie à une catégorie particulière de

patrons : les patrons de conception. Ils ciblent essentiellement des problèmes pouvant apparaître lors de la phase de conception des logiciels. En utilisant le catalogue du « Gang of Four » (GoF) comme référence, nous décrivons les trois types de patrons de conception existants, en montrant le type de problème qu'ils sont le plus à même de résoudre.

I.1. Patrons logiciels

La première adaptation à la conception et à la programmation par objets, du langage de patrons d'Alexander, a été présentée à la conférence OOPSLA de 1987 par Kent Beck et Ward Cunningham [Beck87]. Depuis, de nombreuses classifications de patrons ont été déclinées : c'est ainsi que l'on parle aujourd'hui de « patrons architecturaux » [Buschmann96], de « patrons d'analyse » [Fowler97] et de « patrons de conception » [Gamma95], ciblant une ou plusieurs étapes du cycle de vie d'un logiciel. Il existe également des « patrons de patrons » [Larman00], des « patrons composites » [Riehle97_b] et bien d'autres [Ambler98], [Booch97], [Coad95], [Coplien96], dont les spécificités concernent d'autres aspects de la construction d'un logiciel.

Quel que soit son domaine d'application, un patron est au minimum décrit grâce à trois éléments essentiels, un *nom*, une *intention* et une *solution*. Tout d'abord, le *nom* est choisi de manière à décrire en un ou deux mots un type de problème ou une solution. Le *nom* est souvent utilisé pour identifier le patron parmi d'autres. Son choix est donc très important et nécessite une précision suffisante pour pouvoir le discerner et le reconnaître. L'*intention* est présentée de manière à décrire un problème récurrent que le patron de conception est à même de résoudre. Enfin, la *solution* met en avant la manière de résoudre le problème en donnant les procédures à suivre ou les éléments à utiliser.

Les patrons architecturaux [Buschmann96] 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 eux. L'exemple le plus connu est le patron *Modèle-Vue-Contrôleur* (MVC), introduit par les concepteurs de SmallTalk [Krasner88] et décrit dans le tableau 1.2.

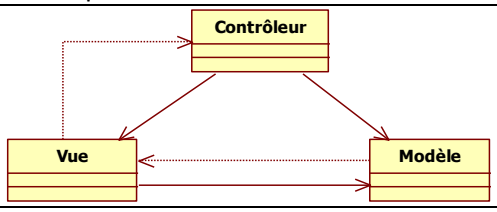
Nom	Modèle-Vue-Contrôleur
Intention	Assure l'indépendance du modèle par rapport à l'interface. Les objets sont séparés en trois composants : le modèle, la vue et le contrôleur. Le modèle contient la logique et l'état de l'application. La vue représente l'interface utilisateur. Le contrôleur réagit aux requêtes de l'utilisateur en effectuant les actions nécessaires sur le modèle.
Solution	

Tableau 1.2 : Le patron architectural *MVC*

Choisis en amont de la phase de conception d'un logiciel, les patrons architecturaux permettent de fixer au plus tôt la structure globale du système. Ils décrivent les règles d'organisation et de fonctionnement, les stratégies de haut niveau, les propriétés et les mécanismes globaux d'une application, permettant ainsi de créer des architectures extensibles et interopérables.

Les patrons d'analyse [Fowler97] traitent de problèmes qui apparaissent lors de l'analyse des exigences des systèmes. Ils aident le développeur dans la construction de modèles devant représenter au mieux les besoins utilisateurs du logiciel. Ils identifient des problèmes répétitifs dans l'expression des exigences des applications de différents domaines et transforment ces besoins en des modèles réutilisables. Le tableau 1.3 présente la description du patron d'analyse nommé *Événement* [Fowler96].

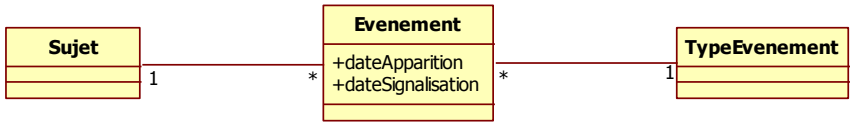
Nom	Événement
Intention	Permet la conservation d'une trace des événements qui peuvent changer le système ou avoir des conséquences sur le domaine.
Solution	

Tableau 1.3 : Le patron d'analyse *Événement*

Il existe d'autres catégories de patrons comme les « patrons de patrons » qui définissent des patrons utiles à la création de patrons [Larman00], ou les « patrons composites » qui sont en fait des assemblages de patrons de conception pour en produire de nouveaux [Riehle97_b]. Un exemple de patron composite, est le patron *Bureaucracy* [Riehle97_a] décrit dans le tableau 1.4.

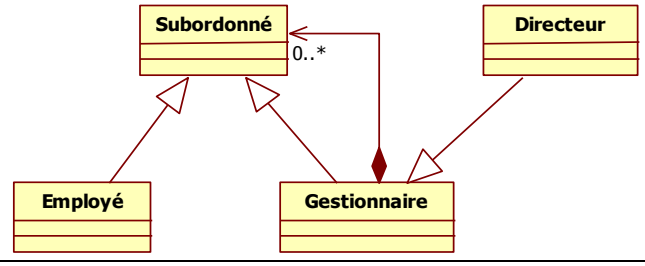
Nom	Bureaucracy
Intention	Permet la construction de structures hiérarchiques autosuffisantes qui sont capables de maintenir leur consistance interne et d'accepter les interactions avec les clients à n'importe quel niveau de hiérarchie.
Solution	

Tableau 1.4 : Le patron composite *Bureaucracy*

Nous pouvons remarquer, dans le cas de ce patron, que la description peut ne pas être suffisante pour maîtriser son utilisation, ce patron étant la combinaison des patrons de conception *Observateur*, *Chaîne de responsabilités* et *Composite*. En effet, à l'exception du patron *Composite*, les autres patrons composés n'apparaissent pas explicitement dans la structure de la solution. Cependant, nombre de patrons ont un exemple d'utilisation permettant de se familiariser avec leur fonctionnement et leur intérêt. Dans l'exemple présenté, le patron représente une hiérarchie d'objets, avec des responsabilités différentes

mais hiérarchiques (Employé, Gestionnaire, 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 étant assurée par les patrons *Chaîne de responsabilités* et *Observateur*.

Finalement, et quel que soit leur domaine, les patrons constituent un véritable moyen pour transférer des connaissances, en permettant l'échange d'un savoir-faire de concepteurs expérimentés, ayant rencontré et résolu plusieurs fois les mêmes problèmes. Ils offrent un gain de temps et une efficacité, tout en réduisant les indéterminations inhérentes à l'élaboration d'un système de qualité [Albin-Amiot01_b]. Les patrons facilitent également la compréhension des systèmes en permettant la structuration et la description contrôlée des systèmes à un haut niveau d'abstraction [Bouhours09]. Les patrons capturent donc les propriétés essentielles d'une architecture et facilitent ainsi la communication autour d'abstractions, de concepts et de techniques qu'ils décrivent dans le but de faciliter la compréhension et la maintenance des systèmes [Bouhours06_b].

I.2. Patrons de conception

Les patrons de conception ciblent essentiellement des problèmes pouvant apparaître lors de la phase de conception des logiciels à objets. Ils permettent de limiter considérablement le temps nécessaire à la résolution des problèmes et d'améliorer la documentation et la maintenance d'un système existant. Les patrons de conception donnent un nom, isolent et identifient les principes fondamentaux d'une structure générale, pour en faire un moyen utile à l'élaboration d'une conception orientée objets réutilisables [Gamma95]. Ils déterminent les classes et les instances intervenantes, leurs responsabilités et leur coopération.

Chaque patron de conception décrit un problème de conception, les circonstances où il s'applique, les effets résultants de sa mise en œuvre, ainsi que les compromis sous-entendus. Les plus répandus ont été proposés et classés par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides [Gamma95] nommés pour le reste de ce mémoire « le GoF » (Gang of Four). Ils ont regroupé, dans un catalogue, vingt-trois patrons de conception, correspondant à vingt-trois problèmes de conception récurrents.

Les patrons du GoF sont organisés en trois catégories. La première concerne les patrons créateurs, listés dans le tableau 1.5, qui proposent des solutions pour gérer l'instanciation d'objets complexes. Les patrons structurels sont listés dans le tableau 1.6. Ils proposent des agencements particuliers de classes et d'objets pour réaliser des structures de modèles plus complexes. Enfin, les patrons comportementaux, listés dans les tableaux 1.7 et 1.8, traitent de la répartition des algorithmes entre plusieurs classes et simplifient les responsabilités attribuées à chaque objet.

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

Tableau 1.5 : Les patrons créateurs et leur intention [Gamma95]

À titre d'exemple, la structure d'un des patrons créateurs, le patron *Singleton* est présenté dans la figure 1.1.

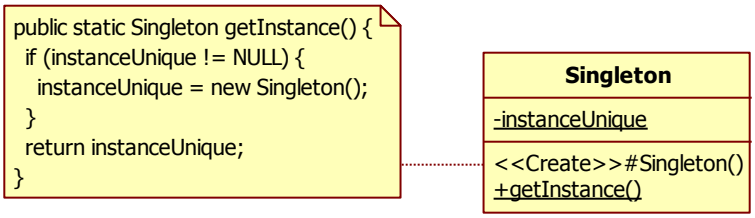


Figure 1.1 : La structure du patron *Singleton*

Le patron *Singleton* permet d'assurer qu'une seule et même instance d'un objet donné est présente pendant toute l'exécution de l'application. Pour obtenir cette unicité, il est nécessaire de masquer tous les constructeurs de la classe, de manière à garantir que la classe est seule maître de la création de ses objets, et de publier une méthode retournant l'instance de la classe. Cette méthode instancie elle-même la classe si ce n'est pas déjà le cas, sinon, retourne l'instance qu'elle aura mémorisée lors du premier appel de cette méthode. Ce patron est très utile pour maintenir une liste de données commune à tout le système, ou pour servir de point d'accès à des sous-systèmes.

Patrons structurels	
Adaptateur	Convertit l'interface d'une classe existante en une interface conforme à l'attente de l'utilisateur. Permet à des classes de travailler ensemble malgré leur incompatibilité d'interface.
Composite	Organise les objets en structure arborescente représentant une hiérarchie de composition. Permet un traitement uniforme des objets individuels et des objets composés.
Décorateur	Attache des responsabilités supplémentaires à un objet de façon dynamique. Offre une solution alternative à la dérivation de classes pour l'extension de fonctionnalités.
Façade	Fournit une interface unifiée à un ensemble d'interfaces d'un sous-système. Définit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.
Poids mouche	Supporte de manière efficace un grand nombre d'instances de granularité fine.
Pont	Découple une abstraction de son implémentation, afin que les deux puissent être modifiées indépendamment.
Procuration	Permet de remplacer temporairement un objet par un autre, pour en contrôler l'accès.

Tableau 1.6 : Les patrons structurels et leur intention [Gamma95]

En raison de sa structure originale incluant le patron *Composite* et à titre d'exemple pour les patrons structurels, la structure du patron *Décorateur* est présentée dans la figure 1.2.

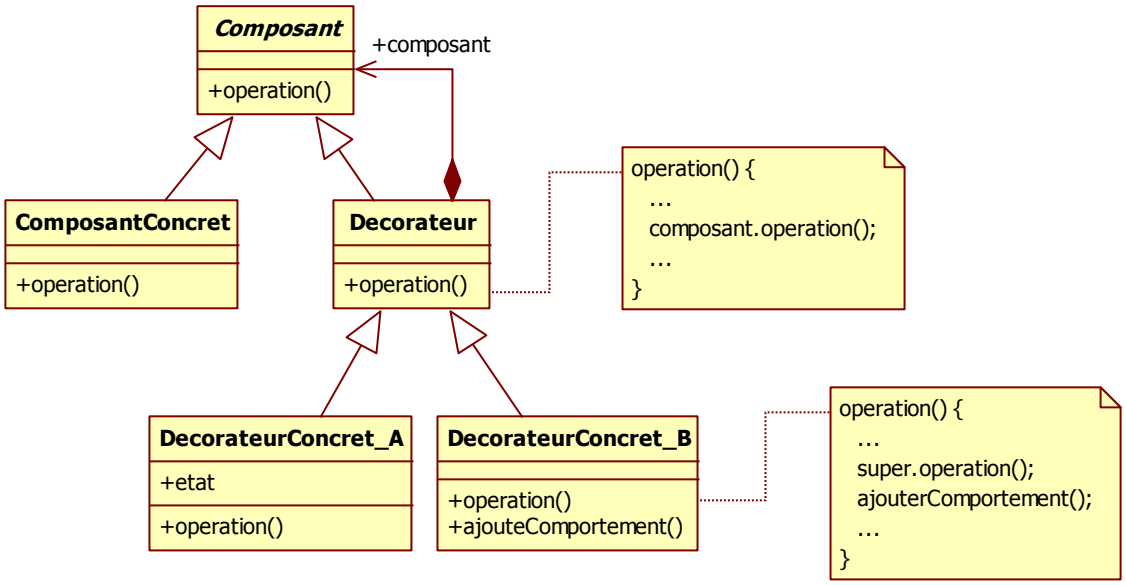


Figure 1.2 : La structure du patron *Décorateur*

Le patron *Décorateur* permet l'ajout dynamique de comportements à un objet. Il propose une structure permettant la décoration multiple des objets en un nombre minimal de classes décorateurs. Pour l'utiliser, il est nécessaire de définir une classe racine qui représentera l'intégralité des décorations possibles (classe *Décorateur*), et une classe qui factorisera les responsabilités des objets pouvant être décorés (classe *Composant*). Il est intéressant de remarquer que les objets décorateurs sont également des objets à décorer, ce qui permet à un objet décorateur d'être décoré d'un autre objet décorateur.

Patrons comportementaux	
Chaîne de responsabilités	Permet de découpler l'expéditeur d'une requête de 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.
Commande	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.
État	Permet à un objet de modifier son comportement lorsque son état interne change.
Interprète	Pour un langage donné, définit une représentation objet de la grammaire utilisable par un interprète pour analyser des phrases du langage.
Itérateur	Fournit un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation sous-jacente.
Médiateur	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 explicitement les uns aux autres.
Mémento	Sans violer l'encapsulation, 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.

Tableau 1.7 : Les patrons comportementaux et leur intention (partie 1) [Gamma95]

Patrons comportementaux	
Observateur	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.
Patron de méthode	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.
Stratégie	Définit une famille d'algorithmes, les encapsule et les rend interchangeables. Une stratégie permet de modifier un algorithme indépendamment de ses clients.
Visiteur	Représente une opération à effectuer sur les éléments d'une structure d'objets. Le visiteur permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère.

Tableau 1.8 : Les patrons comportementaux et leur intention (partie 2) [Gamma95]

Il est intéressant de remarquer que la description de l'intention des patrons comportementaux est plus complexe que pour les autres catégories. En effet, comme nous allons le voir pour le patron *Commande*, dont la structure est présentée dans la figure 1.3, en plus des aspects structurels, ces patrons organisent l'échange des messages entre les classes.

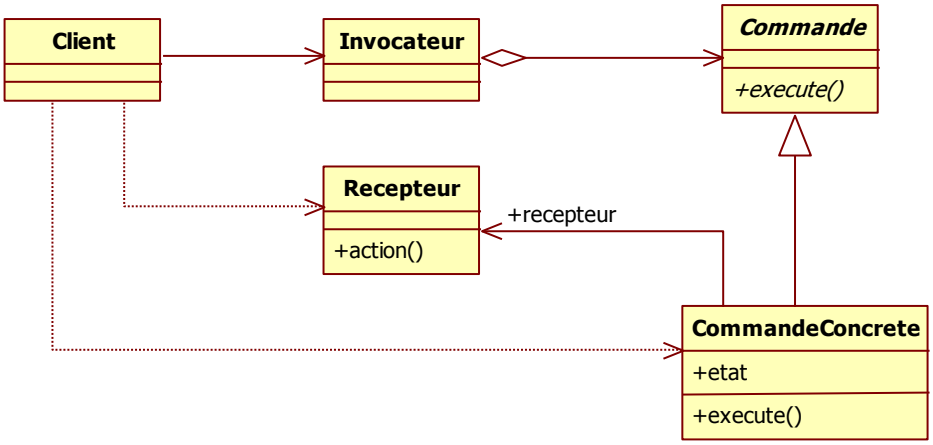


Figure 1.3 : La structure du patron *Commande*

Le patron *Commande* permet de découpler le code de l'invocateur d'une action, du code de l'action elle-même. Son fonctionnement repose sur les objets commande (classes *Commande et CommandeConcrete*). Un objet commande permet de communiquer une action à effectuer, ainsi que les arguments requis, à un récepteur qui exécutera l'action de la commande. Ainsi, lors d'une modification d'une commande, il n'est pas nécessaire de répercuter la modification sur tous les récepteurs. L'invocateur a la possibilité de constituer une liste de commandes pour les exécuter en temps différé, ainsi que pour annuler leurs effets.

Malgré les détails fournis dans le GoF, il n'est pas toujours aisé pour un concepteur de profiter pleinement de l'utilisation des patrons de conception. En effet, pour utiliser un patron, il est nécessaire de trouver lequel correspond le mieux au problème posé et comment le concrétiser sur le contexte de son problème. C'est pourquoi des travaux ont cherché à favoriser l'utilisation des patrons de conception.

II. Favoriser l'utilisation des patrons de conception

Un concepteur désireux d'utiliser un patron de conception doit identifier quel type de problème il souhaite résoudre, et reconnaître ce type de problème dans le GoF. Or, les problèmes du GoF sont génériques de manière à pouvoir s'adapter à n'importe quel contexte de problème. Ainsi, la principale difficulté consiste à s'abstraire du contexte du problème en cours de résolution, pour pouvoir le comparer aux problèmes présentés dans le GoF.

Après avoir identifié le patron adéquat, le concepteur doit s'assurer qu'il a intégré correctement le patron dans son modèle, afin que cette intégration n'ait pas de conséquences négatives. En effet, si l'adaptation du patron au contexte du problème est mal effectuée, le modèle peut s'en trouver dégradé [Huston01]. Il est donc très important pour un concepteur de s'assurer qu'il a utilisé le patron comme il a été prévu. Ceci n'est pas une chose simple, surtout pour un concepteur utilisant pour la première fois un patron. Malgré les détails d'utilisation, la structure et les conseils inscrits avec chaque patron dans le GoF, leur utilisation est toujours sujette au contexte du problème et à l'expérience propre du concepteur.

Pour illustrer cette problématique, imaginons qu'un concepteur cherche à résoudre le problème « *permettre à l'utilisateur d'effectuer un undo/redo* ». Lors de sa recherche dans le GoF, il identifie deux patrons de conception dont l'intention semble correspondre. Ces deux patrons sont le *Mémento* et la *Commande*, dont leur intention respective est « *saisir et transmettre à l'extérieur d'un objet son état interne dans le but de pouvoir le restaurer ultérieurement dans cet état* » et « *encapsuler une requête comme un objet, autorisant ainsi leur exécution sur des clients, ainsi que leur réversion* ». Si le concepteur ne maîtrise pas suffisamment les patrons de conception, il ne saura pas lequel choisir. Le *Mémento* est capable de restaurer l'état d'un objet et la *Commande* permet de mémoriser des opérations pour éventuellement les annuler. La nuance entre les deux est donc subtile. De plus, pour doter une application d'un système de « undo/redo », il est souvent nécessaire de mémoriser l'état d'un objet ainsi que les commandes exécutées. Dans ce cas, l'utilisation combinée des deux patrons s'avère nécessaire : l'un pour profiter de sa capacité à sauvegarder des états et l'autre pour sauvegarder des actions.

Il est donc nécessaire d'aider les concepteurs à utiliser les patrons de conception, de manière à ce qu'ils puissent choisir facilement quel patron utiliser et comprendre comment ils doivent l'adapter au contexte de leur problème. Pour ce faire, des travaux améliorant la classification et la description des patrons existent, ainsi que des méthodes pour vérifier la bonne intégration d'un patron. Cependant, ces travaux ne permettent pas de suggérer, a posteriori, l'intégration de patrons de conception dans un modèle.

En première partie de cette section, nous présentons un formalisme de représentation des patrons, ainsi qu'une méthode de description de leur intention, tous deux destinés à faciliter leur sélection et leur utilisation. Dans une deuxième partie, nous nous intéressons à une technique de métamodélisation permettant d'aider le concepteur, durant sa conception, à reconnaître quand il est nécessaire d'utiliser un patron, et quels en seront les effets sur le modèle. En troisième partie, nous abordons une autre technique de métamodélisation consistant à concevoir des règles métamodélisant le processus d'intégration des patrons de conception, afin de permettre au concepteur de vérifier si les opérations qu'il effectue pour utiliser le patron sont conformes à ce qui a été prévu par le patron. Nous terminons cette section par une quatrième partie décrivant notre contribution pour favoriser l'utilisation des patrons de conception. Grâce à une activité de revue de conception, nous offrons au concepteur un moyen de vérifier, à l'issue de sa conception et de manière semi-automatique, si des patrons pourraient y être injectés.

II.1. Améliorer la classification

En partant du constat que chaque type de patron a son formalisme propre, il est possible d'unifier la représentation des patrons afin de faciliter leur sélection et leur utilisation [Conte01]. De plus, en exprimant une sémantique commune aux formalismes des différents types de patron, il devient possible d'expliciter l'interface de sélection des patrons et de proposer des relations interpatrons permettant d'améliorer l'organisation des catalogues. Pour ce faire, le formalisme utilisé est P-Sigma. Il permet de décrire tout type de patron selon trois rubriques supplémentaires aux rubriques du GoF :

- « Interface » permet de décrire le patron de manière textuelle et formelle selon cinq critères (Identifiant, Classification, Contexte, Problème, Force). Cette rubrique offre au concepteur la possibilité de comparer le patron au type de problème qu'il souhaite résoudre. L'*identifiant* définit le couple problème/solution à partir duquel le patron pourra être référencé. La *classification* définit la fonction du patron par un ensemble de mots-clés du domaine. Le *contexte* décrit textuellement ou formellement la précondition pour l'application du patron. Le *problème* définit ce que résout le patron, et la *force* définit les bonnes pratiques inhérentes à l'utilisation du patron.
- « Réalisation » formalise la démarche à suivre pour utiliser correctement la solution proposée par le patron. De plus, cette rubrique donne des exemples d'utilisation du patron, ainsi que les conséquences de son application.
- « Relation » regroupe les patrons compatibles avec le patron concerné, ou les patrons pouvant être utilisés pour le même type de problème dans des cas précis.

Un patron exprimé en P-Sigma est donc décrit par des critères de comparaison comme des mots-clefs ou de bonnes pratiques de conception. Il permet également de fournir une solution au problème posé ainsi que la démarche à suivre pour obtenir cette solution sur le contexte du problème du concepteur. Le patron est ainsi doté d'un guide méthodologique assistant le concepteur pour adapter le patron. Ainsi décrit, le concepteur détient des éléments supplémentaires pour l'aider à choisir et à appliquer un patron de conception. Ce formalisme constitue un apport majeur à l'aide au choix du patron de conception adéquat.

Une des caractéristiques essentielles des patrons de conception est également leur intention. Cette intention est la description de ce que le patron cherche à résoudre. Elle peut être assimilée au problème, mais avec un niveau de granularité plus fine. Ainsi, l'intention du patron peut se décomposer en plusieurs éléments [verbe – complément] décrivant les concepts communs concernés par le patron [Kampffmeyer07]. C'est ainsi que les éléments de l'intention du patron *Composite* sont « *composer des objets* », « *emboîter des objets* » et « *construire des structures arborescentes* ». En liant les patrons aux éléments formels d'intention et en proposant au concepteur de décrire leurs problèmes en fonction de ces éléments, le choix des patrons s'en retrouve facilité.

II.2. Augmenter la précision de la description

Par delà la classification, il est possible que les patrons de conception soient peu ou mal utilisés parce que leur description n'est pas suffisamment précise [Mili05]. En effet, pour pouvoir utiliser correctement un patron de conception, il est nécessaire d'être capable de reconnaître quand il faut l'utiliser, de comprendre ses effets sur le modèle et d'adapter le patron à ses propres besoins, sans altérer ses fondements [Mili05]. En caractérisant formellement le problème résolu par le patron, en le représentant par exemple sous forme de métamodèle, le choix du concepteur peut s'en retrouver facilité. En effet, le fait que le problème soit représenté sous forme de métamodèle permet au concepteur de vérifier si le problème qu'il souhaite concevoir est conforme à ce métamodèle. Les métamodèles permettent de définir un patron comme des triplets (MP, MS, T), où MP représente le métamodèle du problème résolu par le patron, MS le métamodèle de la solution pour résoudre le problème, et T l'opération de transformation qui permet de transformer MP en MS. Ainsi, lorsque le concepteur découvre un fragment de son modèle conforme au métamodèle du problème résolu par le patron, il n'a plus qu'à appliquer la règle de transformation pour modifier le fragment afin qu'il corresponde au métamodèle de la solution [ElBoussaidi08].

II.3. Valider l'intégration

La principale difficulté, une fois le patron choisi, est de s'assurer que son adaptation et son intégration dans le modèle n'ont pas provoqué l'apparition de défaut de conception. Cependant, en ajoutant à la description des patrons des règles de transformation, il est possible de décrire formellement quelles modifications il est nécessaire d'effectuer, dans le modèle, pour intégrer le patron [Ammour06] [ElBoussaidi08] [France03]. Ces règles métamodélisent le processus d'intégration en permettant ainsi au concepteur de vérifier si les opérations qu'il effectue pour utiliser le patron sont conformes à ce qui a été prévu par le patron. De plus, un métamodèle présentant la solution résolvant le problème du patron permet également de vérifier si le modèle obtenu, après intégration, est conforme au métamodèle de la solution. Finalement, la principale différence avec la description classique des patrons se situe dans la description de ses caractéristiques par des métamodèles. Comme les auteurs pensent qu'il est possible de vérifier si un modèle est conforme à un métamodèle, il est possible de valider que le modèle issu de l'adaptation du patron de conception soit conforme au métamodèle du patron.

De manière générale, il s'avère que de plus en plus de concepteurs sont amenés à travailler sur des modèles existants ou rétroconçus, dans le but de les étendre à de nouvelles fonctionnalités [Sunyé01]. Ce genre d'extension est souvent précédé d'étapes de restructuration dont le but est d'améliorer la qualité des modèles avant d'en modifier le comportement. À ce moment-là, il ne s'agit plus seulement d'adapter et d'intégrer un patron dans un modèle, mais de restructurer un modèle existant pour y adjoindre un patron de conception [Sunyé01]. Or, comme tout processus de restructuration impose que le comportement original du modèle ne soit pas modifié lors de son exécution, l'injection d'un patron de conception est problématique. En effet, bien que certains patrons de conception se concentrent uniquement sur la structure d'un modèle, il en est d'autres qui s'attachent à imposer une cinématique d'échange de messages. Ainsi, l'injection de ces patrons modifie les aspects comportementaux du modèle d'origine et ne peut donc pas être effectuée par restructuration. Cependant, en autorisant la modification de certaines propriétés comportementales prédéfinies explicitement dans la description d'un patron, il est possible de valider qu'un patron de conception ait été correctement intégré, en vérifiant que seules les propriétés prévues aient été modifiées [Sunyé01] [France03].

Tous les travaux présentés permettent d'assister le concepteur lorsqu'il souhaite intégrer les patrons dans ses modèles. Cependant, lorsque la conception est terminée, aucune technique présentée ne permet d'identifier des défauts de conception qui pourraient être corrigés en utilisant les patrons de conception adéquats.

II.4. Recherche de défauts de conception dans un modèle

Nous proposons une approche différente qui consiste à favoriser l'utilisation des patrons de conception dans les modèles, sans imposer au concepteur de chercher lesquels sont utiles pour ses problèmes et en lui justifiant en quoi l'utilisation de tel ou tel patron est plus pertinente pour son modèle. Ainsi, nous offrons au concepteur un moyen de vérifier, à l'issue de sa conception, si des défauts de conception pourraient être corrigés dans son modèle en utilisant des patrons de conception. Le cas échéant, nous lui permettons de transformer automatiquement son modèle pour y injecter les patrons concernés. À l'issue de la vérification et de la transformation, nous garantissons donc au concepteur que son modèle utilise, de manière adéquate, certains patrons de conception.

Pour ce faire, nous avons outillé une activité dite de revue de conception, destinée à vérifier si des patrons pourraient être utilisés dans le modèle. Comme n'importe quel processus nécessite un contrôle pour être efficace, et comme n'importe quel produit de ce processus peut être inspecté [Fagan02], nous assimilons notre activité à un processus d'inspection de modèles. Cette activité de revue de conception constitue notre contribution à l'utilisation des patrons de conception. Fondée sur un catalogue de mauvaises pratiques de conception, pouvant être mis à jour par une communauté, elle permet de cibler les carences en patrons de conception dans un modèle.

Le catalogue de mauvaises pratiques de conception utilisé par l'activité est un regroupement de ce que nous avons appelé « des patrons abîmés » [Bouhours07_b] [Bouhours07_c]. Ces derniers dégradent les qualités intrinsèques des patrons de conception et sont ainsi remplaçables par les patrons de conception correspondants. Ils représentent l'élément central de nos travaux en nous permettant d'identifier et d'expliquer les défauts de conception constatés et corrigés par l'utilisation de patrons de conception.

III. Les patrons abîmés

Identifier dans un modèle des fragments puis les remplacer par des patrons de conception nécessite de pouvoir cibler des agencements de classes. Nous cherchons donc des fragments représentant de mauvaises pratiques de conception que des patrons de conception sont capables de corriger. De même qu'un patron de conception représente la solution idéale à un type de problème, nous nommons « patrons abîmés », l'ensemble des solutions moins efficaces pour résoudre le même problème [Bouhours06_b] [Bouhours09].

Dans cette section, nous détaillons, dans une première partie, nos hypothèses de travail. Notre axiome de base est qu'un patron de conception est la meilleure microarchitecture réutilisable pour un type de problème donné. En deuxième partie, nous définissons les concepts que nous avons utilisés pour concrétiser notre approche. Nous mettons notamment en relation le concept de patron de conception avec celui de patron abîmé, par la « manière » dont ils résolvent des problèmes de conception. Afin d'expliciter chaque définition, nous illustrons, dans une troisième partie, les concepts présentés. Pour ce faire, nous utilisons un problème de composition d'objets, le patron *Composite* et une solution que nous qualifions d'alternative parce qu'elle résout, d'une autre « manière », ledit problème. Nous terminons enfin cette section en reliant les concepts existants de « bad smells » et d'« antipatterns » avec notre concept de patron abîmé.

III.1. Hypothèses de travail

Puisqu'un patron de conception a été approuvé, éprouvé et validé par une communauté d'experts du domaine, nous estimons qu'il fournit une solution optimale à un problème donné. Ce problème est présenté sous forme générique et adaptable à n'importe quel contexte d'utilisation. Ainsi, le patron de conception est une entité réutilisable et adaptable à un contexte de problème. De plus, comme il est le garant d'une facilité de développement et d'un gain de temps durant la conception, grâce aux bonnes pratiques de conception qu'il apporte, nous considérons qu'il représente le meilleur agencement de classes et de messages, pour résoudre un type de problème.

Axiome 1 : *« Un patron de conception » est la meilleure microarchitecture réutilisable pour un type de problème donné.*

Par microarchitecture, nous regroupons l'agencement des classes, la répartition des attributs et des méthodes et la structure des échanges de messages entre les classes.

Corollaire 1 : *Pour chaque problème de conception soluble par un patron de conception, « la meilleure solution » est l'adaptation d'un patron de conception au contexte du problème.*

Un patron de conception étant présenté de manière générique pour s'appliquer à n'importe quel contexte d'un problème, nous définissons le procédé permettant d'utiliser le patron de conception dans un contexte précis.

Définition 1 : *« Une contextualisation » est le procédé consistant à adapter un patron de conception au contexte particulier d'un problème.*

Définition 2 : *« Une décontextualisation » est le procédé inverse de la contextualisation.*

Pour contextualiser un patron sur un problème, il est nécessaire que le problème soit conforme au type de problème soluble par le patron. Ce procédé permet, à partir d'un patron de conception, de produire une solution à chaque contexte d'un type de problème donné. La figure 1.4 illustre le résultat de l'exécution de ce procédé et inversement.

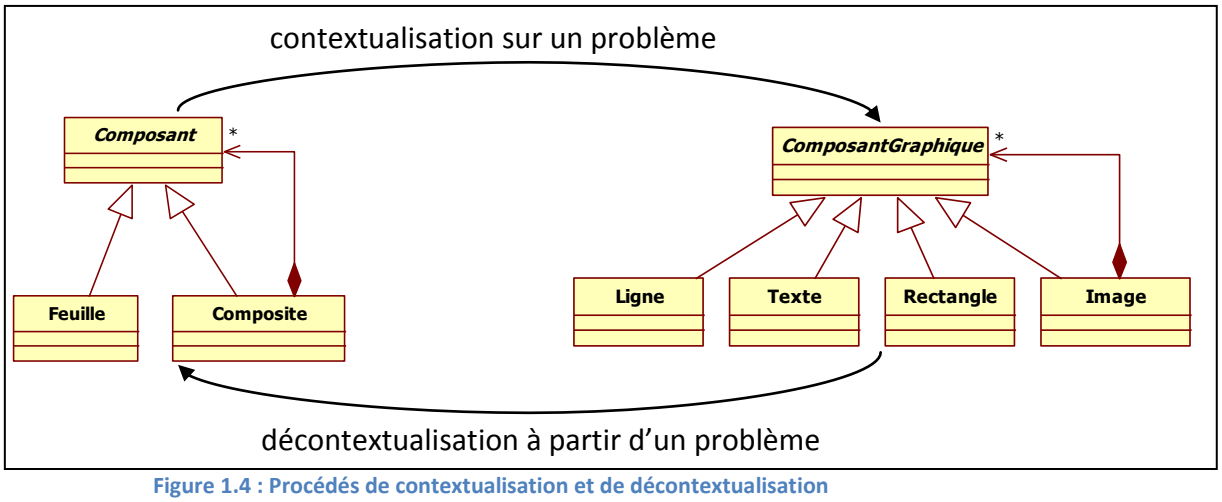


Figure 1.4 : Procédés de contextualisation et de décontextualisation

Dans ce mémoire, lorsque nous parlerons de « problème donné », de « problème » ou de « problème de conception », nous sous-entendrons qu'un contexte a été appliqué à un type de problème. Ainsi, nous pouvons dire qu'un problème est issu de la contextualisation d'un type de problème. Un patron de conception permet donc de résoudre un « type de problème », alors qu'un modèle utilisant un patron de conception résout un « problème ».

III.2. Définitions

Dans les définitions suivantes, nous admettons qu'un problème donné est soluble par la contextualisation d'un patron de conception.

> **Définition 3 :** *« Une solution alternative » est une solution valide pour un problème donné, mais avec une architecture différente de celle de la meilleure solution.*

Ainsi, les exigences du problème sont respectées par la solution alternative, mais les relations entre les classes sont différentes de celles du patron, et/ou il n'y a pas la totalité des participants du patron. Si le concepteur s'est trouvé confronté à un problème de conception qu'il aurait pu résoudre avec un patron de conception, mais qu'il ne l'a pas utilisé, il l'aura résolu avec une solution alternative.

Nous pouvons en déduire le corollaire suivant :

> **Corollaire 2 :** *Une solution alternative est une solution inadéquate à un problème donné et est remplaçable par la contextualisation du patron de conception correspondant au type de problème considéré.*

Puisqu'une solution alternative est valide pour un contexte donné, il est possible de la décontextualiser pour obtenir un modèle générique permettant de résoudre, de manière inadéquate, un certain type de problème.

> **Définition 4 :** *« Un patron abîmé » est la décontextualisation d'une solution alternative, de la même manière qu'un patron de conception est la décontextualisation de la meilleure solution. Un patron abîmé est relié à un et un seul patron de conception.*

Un patron abîmé, auquel nous avons donné un nom dépendant de son défaut de conception principal, est comparable à un patron de conception, puisqu'il est réutilisable pour produire des modèles qui résolvent des problèmes. Nous avons donc pour un type de problème, un patron de conception permettant de construire la meilleure solution, et un ensemble de patrons abîmés permettant de produire des solutions inadéquates.

Par comparaison, nous pourrions dire que les contextualisations de patrons abîmés sont équivalentes à des contextualisations incomplètes ou défaillantes de patrons de conception. Cependant, alors qu'un patron abîmé représente un ensemble de défauts de conception caractérisés et corrigibles, un patron de conception mal contextualisé n'a pas une forme connue à l'avance et ne représente pas forcément un défaut de conception.

Structurellement, un patron abîmé est représenté avec le même niveau de granularité qu'un patron de conception, ce qui permet de le décrire de la même manière qu'un patron de conception.

Définition 5 : *« Les particularités remarquables » d'un patron abîmé sont ses caractéristiques architecturales significatives exprimées en UML (associations, généralisations et lien de composition, mais sans tenir compte ni des attributs ni des méthodes).*

Les particularités remarquables nous permettent de décrire structurellement un patron abîmé. Elles sont décomposées en deux sous-ensembles :

- *Les particularités locales,* qui décrivent chaque classe individuellement. Puisque les caractéristiques architecturales significatives qui nous intéressent sont les relations entre les classes, les particularités locales décrivent une classe avec ses relations entrantes et sortantes, sans spécifier les classes sources ou de destination.
- *Les particularités globales,* qui caractérisent les relations entre les classes d'un patron abîmé. Il s'agit donc de la description de la manière dont une classe est reliée aux autres classes du patron.

Nous utilisons les particularités remarquables pour détecter des contextualisations de patrons abîmés dans les modèles, et cette séparation local/global nous permet différents niveaux de filtrage des classes. Les particularités locales nous permettent d'éliminer toutes les classes n'ayant pas « localement » les bonnes relations. Les particularités globales nous permettent de comparer des structures de graphes. Ceci permet d'éviter de vérifier toutes les relations entre les classes, puisque seules les relations des classes structurellement comparables aux particularités locales sont vérifiées. Grâce aux particularités remarquables, nous pouvons identifier dans un modèle des fragments structurellement comparables aux patrons abîmés.

Définition 6 : *« Un fragment alternatif » est un fragment de modèle tel que ses particularités remarquables correspondent aux particularités remarquables d'un patron abîmé et dont l'intention est conforme au patron de conception correspondant.*

Cette correspondance indique qu'il y a une concordance structurelle entre un fragment alternatif et un patron abîmé. Ainsi, lorsqu'un fragment alternatif est détecté, cela signifie qu'une contextualisation du patron abîmé a été détectée, et donc, que ce fragment est remplaçable par une contextualisation du patron de conception correspondant. La méthode de détection structurelle que nous avons conçue et mise en œuvre est décrite plus amplement dans le chapitre 3 de cette thèse.

Avant de qualifier un fragment de modèle de fragment alternatif, il est nécessaire de s'assurer que l'intention du fragment est bien conforme à l'intention du patron abîmé. En effet, la concordance structurelle ne garantit aucunement que l'intention du fragment soit conforme à celle du patron alternatif. Cette conformité d'intention n'est pas détectée par les particularités structurelles et doit donc être vérifiée en complément de cette détection. Chaque patron de conception ayant une intention, les patrons abîmés correspondants

possèdent la même. Ainsi, pour un fragment identifié en concordance structurelle avec un patron abîmé, l'intention du patron correspondant est présentée au concepteur qui devra confirmer sa conformité.

Nous avons choisi le terme « abîmé » pour décrire ce nouveau type de patron, parce qu'il correspond à une dégradation des qualités intrinsèques des patrons de conception et qu'ils sont ainsi remplaçables par les patrons de conception correspondants. Les différences structurelles entre un patron de conception et un patron abîmé provoquent une dégradation de l'efficacité à résoudre un type de problème de manière adéquate.

Définition 7 : *« Les points forts » d'un patron de conception expriment les critères d'architecture et les facteurs de qualité logicielle apportés par son utilisation. Ces critères sont partiellement déduits de la section conséquence du catalogue du GoF et des défauts de conception constatés lors de l'utilisation des patrons abîmés. Ils explicitent en quoi un patron de conception est la meilleure solution pour un type de problème.*

Les solutions alternatives sont en fait plus ou moins efficaces pour résoudre un problème. Il est possible de quantifier un degré de dégradation en considérant la valuation des points forts d'un patron. Comme les points forts du patron caractérisent l'efficacité et la qualité de la solution, nous pouvons dire que le remplacement d'un fragment alternatif par un fragment optimal, issu de la contextualisation d'un patron de conception, corrige les défauts de conception engendrés par l'utilisation du patron abîmé.

Définition 8 : *« Un fragment optimal » est le résultat du remplacement d'un fragment alternatif par la contextualisation du patron de conception correspondant.*

Tous les concepts qui viennent d'être présentés sont mis en relation dans la figure 1.5.

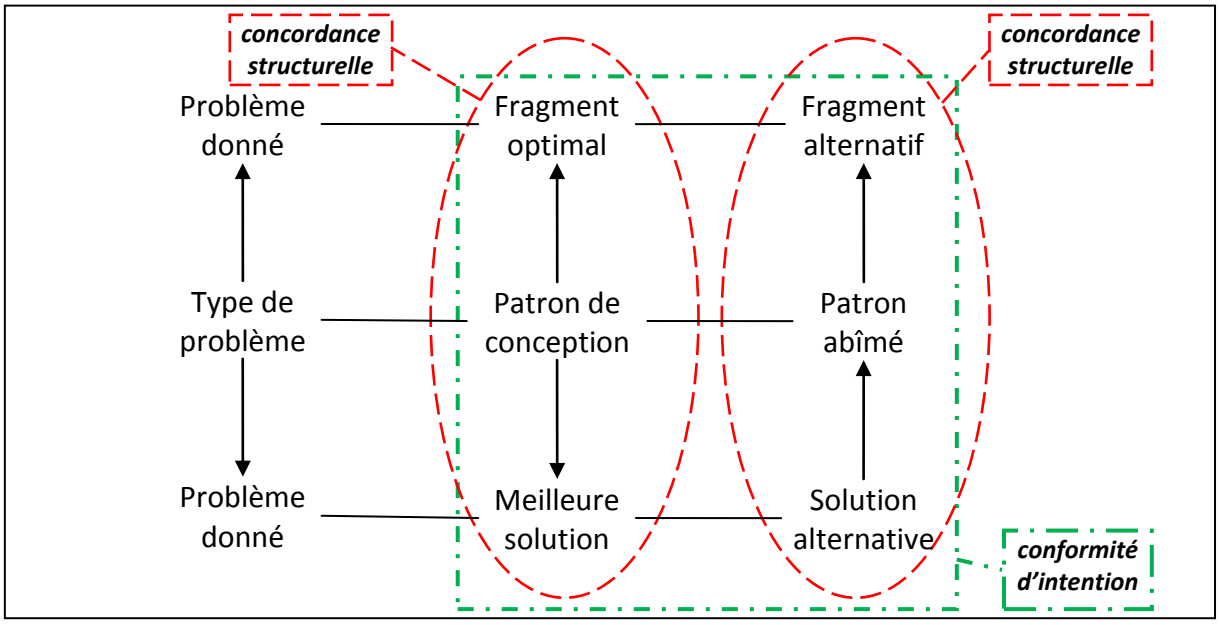


Figure 1.5 : Synopsis des concepts

Sur ce schéma, les flèches représentent les procédés de contextualisation et de décontextualisation. Ainsi, un *fragment optimal* est issu de la contextualisation d'un *patron de conception* pour un *problème donné*. À l'inverse, les *patrons abîmés* sont issus de la décontextualisation de *solutions alternatives* à un problème donné. C'est en effet en collectant des solutions alternatives que nous avons conçu notre catalogue de patrons abîmés. La méthodologie utilisée est présentée dans le chapitre 2 de cette thèse.

Le carré représente une conformité d'intention entre chacun des concepts qu'il englobe. Ainsi, une *meilleure solution*, une *solution alternative*, un *patron de conception*, un *patron abîmé*, un *fragment optimal* et un *fragment alternatif* ont la même intention. Un *patron abîmé* a la même intention qu'un *patron de conception*, ce qui est également vrai pour les fragments issus de leur contextualisation.

Enfin, les deux ellipses mettent en avant la concordance structurelle entre les concepts. Par exemple, la relation entre un *patron abîmé*, *un fragment alternatif* et *une solution alternative* se caractérise par leur structure équivalente, issue du procédé de contextualisation.

III.3. Illustration des concepts

Nous illustrons nos concepts par le patron de conception *Composite*, décrit par la figure 1.6. Nous avons délibérément choisi de ne représenter les patrons de conception que par des diagrammes de classes, inspirés de la section *structure* du GoF, avec uniquement les participants du patron en tant que classes et leurs relations structurelles (associations et liens d'héritage). Nous avons donc omis les méthodes de chacun des participants du patron lorsqu'elles étaient renseignées dans le GoF. Nous justifions cette limite à l'architecture des modèles, car la prise en compte des méthodes aurait nécessité l'usage d'un oracle pour interpréter leur sémantique. De plus, en ne prenant pas les méthodes en compte, nous limitons le nombre de variantes possibles pour un patron à un problème donné. Dans ce contexte, notre approche consiste à identifier les concordances structurelles en utilisant des comparaisons, sans se préoccuper des détails d'implémentation qui peuvent complexifier l'identification des contextualisations des patrons. Ainsi, nous sommes capables d'identifier des fragments alternatifs avec un minimum de données. Notre méthode de détection est détaillée dans le chapitre 3.

L'intention du patron *Composite* est de « c*omposer les objets en des structures arborescentes pour représenter des hiérarchies composant/composé, ce qui permet au client de traiter de la même et unique façon les objets individuels ainsi que leurs combinaisons* » [Gamma95]. En suivant nos hypothèses de travail, ce patron est la meilleure solution pour résoudre des problèmes de composition d'objets, de construction de structures arborescentes ou d'emboîtement d'objets [Kampffmeyer07].

Le patron *Composite* introduit trois participants : un *Composant* abstrait, un *Composite* et une *Feuille*. Le *composant* abstrait définit une interface commune aux objets composés et à la gestion de la composition, et offre un point d'accès unique au client. Cette entité permet la factorisation de la composition sur les composites et sur les feuilles. Le participant *Composite* gère la relation de composition et délègue les opérations récursivement le long de la structure arborescente. Les feuilles, comme leur nom l'indique, représentent les éléments terminaux de la structure arborescente.

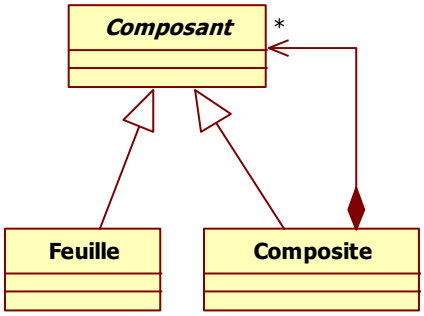


Figure 1.6 : Structure du patron de conception *Composite*

Considérons maintenant le problème présenté dans l'énoncé 1.1, inspiré du GoF.

Concevoir un système permettant de dessiner des images graphiques : une image graphique est composée de lignes, de rectangles, de textes et d'images. Une image peut elle-même être composée d'autres images, de lignes, de rectangles et de textes.

Énoncé 1.1 : Un problème issu du GoF

Cet énoncé sous-entend que le type de ce problème concerne une composition hiérarchique d'objets, la hiérarchie s'articulant autour du concept d'image. Pour contextualiser le patron *Composite* sur ce problème, nous devons identifier les éléments du problème ayant les mêmes responsabilités que chacun des participants du patron. Le concept d'*Image* a les mêmes responsabilités que le participant *Composite*. Les classes *Ligne*, *Texte* et *Rectangle* constituent les éléments terminaux de la hiérarchie et ont donc les mêmes responsabilités que le participant *Feuille*. Enfin, nous pouvons considérer qu'*ImageGraphique* constitue l'élément générique de la hiérarchie de composition, ce qui le rapproche des responsabilités du *Composant*. Nous obtenons ainsi une contextualisation du patron *Composite*, et, en accord avec nos hypothèses, nous pouvons dire que la figure 1.7 représente la meilleure solution pour le problème présenté ci-dessus.

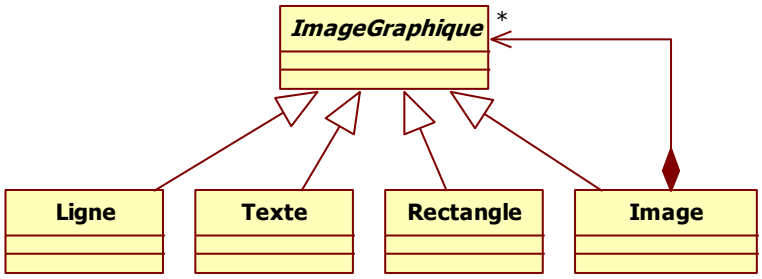


Figure 1.7 : Contextualisation du patron *Composite* sur l'exemple

La figure 1.8 présente une solution alternative au problème précédent. Dans cette solution, nous pouvons identifier qu'une image est composée d'autres images qui peuvent être composées de lignes, de textes et de rectangles. Les besoins du problème sont donc respectés vis-à-vis de la composition d'objets. La classe *ImageGraphique* est utilisée pour supporter la factorisation des protocoles et pour être le point d'accès unique au client. Cependant, le fait que les classes *Ligne*, *Rectangle* et *Texte* soient attachées à *Image* entraîne des modifications de code si de nouvelles classes sont ajoutées, avec les responsabilités de *Feuille* ou de *Composite*. Ainsi, si une nouvelle classe *Cercle* est ajoutée comme *Feuille*, la classe *Image* devra gérer cette nouvelle référence vers la nouvelle classe, ce qui entraînera une modification du code de la classe *Image*. Ceci n'est pas le cas en utilisant le patron de conception présenté à la figure 1.6.

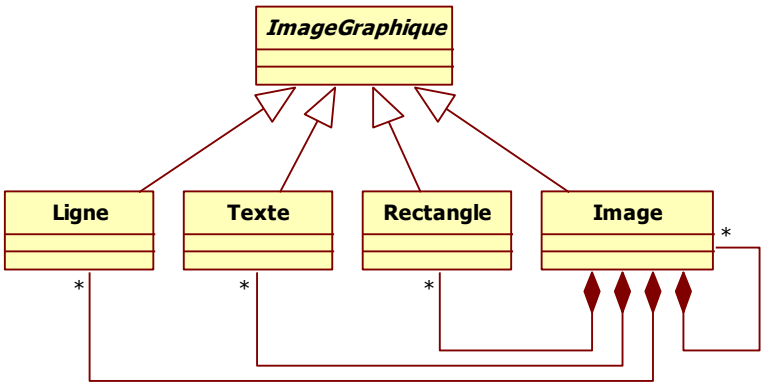


Figure 1.8 : Une solution alternative

Pour qu'une solution alternative soit détectable dans un modèle quel que soit le contexte du problème, il est nécessaire de la décontextualiser. Cette décontextualisation nous permet d'obtenir un patron abîmé « générique », capable d'être adapté à n'importe quel contexte de problème. Cette généricité nous permet de considérer un patron abîmé comme une base génératrice de fragments alternatifs.

Le procédé de décontextualisation d'une solution alternative nécessite d'identifier les participants du patron, puis d'effectuer une « réduction » permettant de ne conserver qu'une classe par participant du patron. Cependant, certaines solutions alternatives n'utilisent pas la totalité des participants du patron de conception, ce qui implique que certaines des classes ont les responsabilités de plusieurs participants. Dans ce cas, nous ne conservons que les participants présents dans la solution, ce qui produit des patrons abîmés avec moins de participants que le patron qu'ils dégradent.

La première étape de ce procédé de décontextualisation consiste à marquer chaque classe de la solution alternative avec le nom d'un des participants du patron ayant les mêmes responsabilités. *ImageGraphique*, classe abstraite, offre une interface commune à toutes les autres classes et un point d'accès unique pour la classe client. Elle a donc les responsabilités du participant *Composant*. La classe *Image* gère la composition et représente, par son lien récursif, l'articulation de l'arborescence ; elle a donc les responsabilités de *Composite*. Enfin les classes *Ligne*, *Texte* et *Rectangle* sont clairement les éléments terminaux de l'arborescence et ont donc les mêmes responsabilités que le participant *Feuille*.

Ce marquage des classes d'une solution alternative est effectué manuellement, puisqu'il nécessite une analyse de la sémantique des classes. De plus, dans cet exemple, même si on pourrait imaginer un marquage automatique en fonction de la similarité structurelle entre la meilleure et la solution alternative, toutes les solutions alternatives n'ont pas une architecture aussi proche de la meilleure solution. Certaines ont des classes en plus ou en moins, ou des architectures avec peu de points communs, ce qui empêche toute automatisation. Le résultat, résumé par la figure 1.9, montre le marquage des classes de la solution alternative par les participants du patron *Composite*.

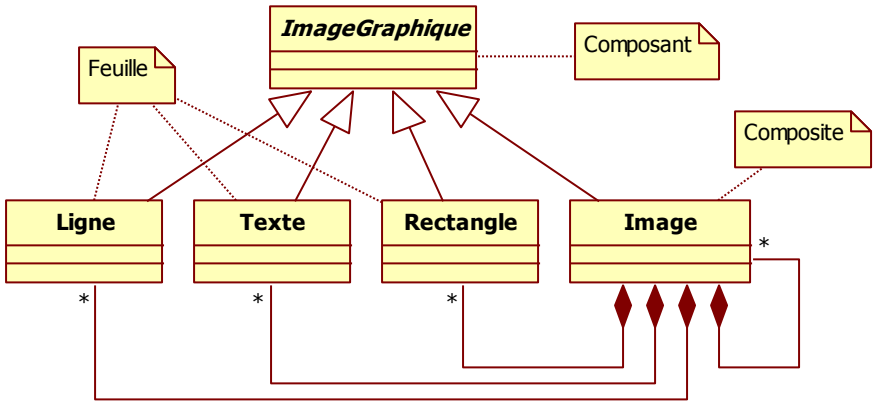


Figure 1.9 : La solution alternative après l'étape de marquage

Après le marquage, la seconde étape de la décontextualisation consiste à conserver, une seule fois, chacun des participants rattachés de la même manière que dans la solution alternative. Cette réduction peut être complexe sur certains participants lorsque plusieurs classes sont indissociables en raison de leur collaboration commune sur les responsabilités du participant concerné. Cette étape de décontextualisation est donc un processus mental d'assimilation des responsabilités vis-à-vis des classes de la solution alternative.

Dans le cas de la solution présentée dans la figure 1.9, nous déduisons un modèle avec trois classes *Composant*, *Composite* et *Feuille*, remplaçant respectivement la classe *ImageGraphique*, la classe *Image* et une des classes *Texte*, *Ligne* ou *Rectangle*. Nous obtenons alors un patron abîmé du *Composite* où la composition est développée sur la classe *Composite*. La figure 1.10 résume cette étape du processus de décontextualisation et présente un patron abîmé pour le patron de conception *Composite,* nommé *développement de la composition sur <<Composite>>*.

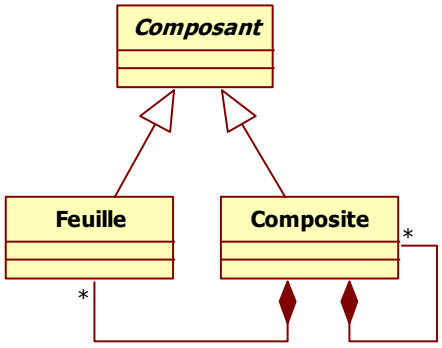


Figure 1.10 : Un patron abîmé du *Composite*

À partir de plusieurs solutions alternatives d'un même type de problème, nous avons obtenu un ensemble de patrons abîmés pour le patron de conception résolvant ce type de problème. Pour classifier les patrons abîmés les uns par rapport aux autres, nous avons quantifié leur degré de dégradation grâce aux points forts du patron de conception concerné. En effet, chaque patron abîmé ne profite que d'une partie des points forts du patron. C'est ce qui explique sa dégradation.

Pour le patron *Composite*, la factorisation maximale de la composition et la standardisation du protocole, grâce aux liens d'héritage, nous permet de dire que les points forts du patron sont « *découplage et extensibilité* » et « *protocole uniforme* ». Comme la composition du patron abîmé de la figure 1.10 se traduit par un lien réflexif et un développement sur toutes les classes feuilles, un défaut de conception apparaît, conséquence de la dégradation du point fort « *découplage et extensibilité* ». La factorisation n'est pas maximale et le fort couplage entre *Feuille* et *Composite* impose des modifications de code, dès l'ajout d'une classe avec l'une ou l'autre responsabilité. Cependant, comme il y a toujours des liens d'héritage, le patron abîmé ne dégrade pas le point fort « *protocole uniforme* ».

Cette caractérisation des patrons abîmés nous permet de présenter au concepteur l'avantage qu'il aurait à remplacer le fragment détecté par le patron de conception correspondant. Nous lui montrons ainsi quels points forts dégradés par le patron abîmé seront corrigés par l'injection du patron de conception, ce qui lui permet également de prendre conscience des défauts de conception présents dans son modèle.

Le tableau 1.9 résume la dégradation des points forts par le patron abîmé de la figure 1.10. Les points forts du patron *Composite* dégradés par le patron abîmé y sont décrits précédés du symbole ✘, à l'inverse des points forts conservés qui sont précédés de ✔.

Découplage et Extensibilité	
✘	Factorisation maximale de la composition.
✘	L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code.
✘	L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code.
Protocole uniforme	
✔	Protocole uniforme sur les opérations des objets composés.
✔	Protocole uniforme sur la gestion de la composition.
✔	Point d'accès unique pour la classe client.

Tableau 1.9 : Dégradation des points forts du patron *Composite* par l'un de ses patrons abîmés

Pour permettre l'identification des contextualisations d'un patron abîmé, il est nécessaire de matérialiser ses particularités structurelles, en analysant structurellement chaque participant du patron abîmé. L'analyse du patron abîmé de la figure 1.10 nous montre que le participant *Composite* a au moins deux relations de composition, une pour *Feuille* et une récursive. Concernant le *Composant*, il est caractérisé par deux liens d'héritage pour *Feuille* et pour *Composite*. Enfin, le participant *Feuille* est agrégé à *Composite*. Le tableau 1.10 présente les particularités structurelles du patron abîmé présenté dans la figure 1.10.

Participant de référence	Composite	
Particularités locales	Composite	Classe avec au moins deux compositions (0..*) dont une récursive et un lien d'héritage.
	Feuille	Classe avec au moins une composition (0..*) et un lien d'héritage.
	Composant	Classe avec au moins deux filles.
Particularités globales	Feuille	Sous-classe de *Composant* rattachée au participant de référence par une composition (0..*).
	Composant	Superclasse du participant de référence et de *Feuille*.

Tableau 1.10 : Les particularités structurelles d'un des patrons abîmés du *Composite*

Nous pouvons noter que le participant *Composite* est marqué en tant que participant de référence et qu'il n'a pas de particularités globales. La spécificité du participant de référence est qu'il est unique et obligatoire à chaque contextualisation du patron abîmé. Il n'est alors pas positionné globalement par rapport aux autres participants. Nous détaillons ce participant particulier dans le chapitre 3 de cette thèse.

Pour constituer une base de patrons abîmés, nous avons collecté, à l'aide d'expérimentations, un ensemble de solutions alternatives auprès de jeunes concepteurs n'ayant pas ou peu d'expérience sur les patrons de conception. Nous leur avons demandé de concevoir des solutions à des problèmes dont nous savions que la meilleure solution était la contextualisation d'un patron de conception. Ces concepteurs n'ayant pas le réflexe d'utiliser les patrons, ils ont cherché des solutions aux problèmes selon leur propre expérience et ont ainsi produit, pour la plupart, des solutions alternatives. Nous détaillons ces expérimentations et les patrons abîmés déduits dans le chapitre 2 de cette thèse.

III.4. Patrons abîmés, bad smells et antipatterns

Nous positionnons maintenant notre concept de « patron abîmé » par rapport aux termes d'ingénierie logicielle consacrés aux patrons. La détection de fragments alternatifs dans un modèle peut évoquer les « bad smells » et l'explication de leurs défauts de conception par rapport aux patrons de conception peut faire penser aux « antipatterns ».

III.4.1. Les bad smells

Ce sont Kent Beck et Martin Fowler qui ont introduit le terme « bad smells » [Fowler99]. Ces bad smells sont un ensemble d'indices dans le code d'un programme suggérant de mauvaises pratiques de conception, ce qui permet d'identifier quelles sont les parties du code à restructurer pour corriger les problèmes, et quelles procédures doivent être suivies pour effectuer cette restructuration. Par exemple, la duplication de code dans un programme est un bad smell qui peut être corrigé par la technique de restructuration « extract method » [Fowler99]. Cette dernière consiste à ajouter une méthode dans une classe afin qu'elle factorise les parties de code concernées.

Un fragment alternatif dans un modèle indique où se situe une malfaçon pouvant engendrer des effets indésirables sur le modèle et cible une zone du modèle qu'il faudrait restructurer. Alors que les bad smells ont été définis pour cibler des morceaux de code, les fragments alternatifs ciblent des fragments de modèle. Ainsi, par analogie, identifier des fragments alternatifs peut être assimilé à une recherche de bad smells dans des modèles. Un fragment alternatif étant issu de la contextualisation d'un patron abîmé, nous considérons que les patrons abîmés peuvent être utilisés comme base génératrice de bad smells dans des modèles.

III.4.2. Les antipatterns

Il existe deux manières de définir un antipattern. Alors qu'un patron de conception présente la meilleure solution à suivre pour résoudre un problème, l'antipattern présente une leçon apprise. Il décrit les effets résultant de mauvaises pratiques de conception et donne la procédure à suivre pour tendre vers une meilleure qualité logicielle. Un antipattern permet alors de vérifier ou de surveiller des mauvaises pratiques [Dodani06]. Un antipattern peut également représenter de bonnes pratiques de conception, mais qui utilisées de manière excessive produisent, au final, des conséquences plus néfastes que les résultats escomptés [Brown98]. Dans tous les cas, un antipattern propose un ensemble d'opérations pour corriger une mauvaise solution constatée. Un antipattern est décrit par les explications sur la nature des défauts constatés et par un processus de restructuration qui explique comment passer de la mauvaise solution à une bonne solution. À titre d'exemple, citons l'antipattern « attente active » qui consiste, en programmation concurrente, à tester une condition jusqu'à ce qu'elle soit vérifiée. Cet antipattern peut être corrigé en utilisant des événements ou des signaux.

Nous considérons que les patrons abîmés sont des antipatterns, mais avec une précision plus fine. Le patron abîmé ne donne pas d'information permettant d'éviter la mauvaise solution. De par la description fine de la mauvaise solution, le patron abîmé peut être détectable automatiquement, ce qui n'est pas le cas, ni le but, des antipatterns. Un patron abîmé permet de vérifier que des « mauvaises manières de faire » n'ont pas déjà été utilisées, et il est directement lié à un patron de conception. La suite d'opérations de restructuration utiles pour le substituer est de même beaucoup plus précise qu'une restructuration d'antipattern.

IV. Une activité de revue de conception

Pour concrétiser les concepts présentés à la section précédente et afin de pouvoir l'intégrer dans un processus de développement, nous avons conçu et implémenté une activité de revue de conception, au même titre qu'il existe des activités de revue de code [Fagan02]. Il s'agit de détecter, de manière semi-automatique, des contextualisations de patrons abîmés, et si nécessaire, de transformer les modèles en y injectant les patrons de conception adéquats. Notre activité se décompose en trois étapes : la détection de fragments alternatifs, la communication avec le concepteur pour vérifier l'intention des fragments et lui présenter les avantages de la substitution, puis les transformations de modèles pour y intégrer les patrons de conception [Bouhours07_a] [Bouhours08].

Dans cette section, nous présentons dans une première partie chaque étape de notre activité de revue de conception. Nous montrons que les résultats de chacune d'entre elles sont utilisés dans la suivante, ce qui les rend indissociables et complémentaires. En deuxième partie, nous illustrons le fonctionnement de l'activité sur un cas d'école. À partir d'un modèle conçu en utilisant de bonnes pratiques de conception, notre activité nous a permis de détecter et de corriger un fragment alternatif du patron *Composite*.

IV.1. Présentation

Pour rechercher des fragments alternatifs, l'activité utilise un catalogue de patrons abîmés présenté dans le chapitre 2. Elle est capable d'identifier dans un modèle toutes les contextualisations possibles de chaque patron abîmé du catalogue, d'après leurs particularités structurelles. Le modèle à analyser est fourni à l'activité au format XMI [OMG07] et est chargé en mémoire grâce à la plate-forme Neptune [Millan08]. Des requêtes OCL [OMG06], déduites automatiquement des particularités structurelles de chaque patron abîmé, sont fournies à Neptune pour identifier structurellement un ensemble de fragments de modèles correspondant aux contextualisations de patrons abîmés. La méthode de détection utilisée est présentée dans le chapitre 3 de cette thèse, et la manière dont les requêtes sont générées dans le chapitre 4.

À l'issue de la détection, les fragments identifiés ne sont pas encore considérés comme alternatifs car nous ignorons leur intention. En effet, la structure seule d'un fragment de modèle ne fournit aucune certitude sur les responsabilités de chacune des classes et seul le concepteur est capable de fournir ces informations. C'est pourquoi la deuxième étape consiste à vérifier auprès du concepteur quelle est l'intention des fragments précédemment identifiés. Pour ce faire, nous utilisons une ontologie définie en OWL [MCGuinness04], présentée dans le chapitre 4, contenant les informations relatives aux intentions des patrons de conception, ainsi que les points forts dégradés par les patrons abîmés. Chaque fragment étant associé à un patron abîmé, lui-même associé à un patron de conception, nous pouvons facilement vérifier que le fragment a la même intention que le patron de conception. Le concepteur peut alors confirmer ou infirmer l'intention de chacun des fragments identifiés à

la première étape. En cas de confirmation, les fragments prennent le statut de fragments alternatifs, sinon, ils sont éliminés de la liste des fragments potentiels. Le concepteur est ensuite informé de l'avantage qu'il aurait à remplacer ces fragments alternatifs par les patrons adéquats. Les fragments alternatifs choisis interviennent alors dans la troisième étape de l'activité.

Les restructurations de modèle sont effectuées automatiquement par un outil que nous avons développé. Chaque classe des fragments alternatifs détectés est marquée, en fonction de sa responsabilité, par le nom d'un des participants du patron abîmé correspondant. Grâce à ce marquage, l'injection du patron en est simplifiée.

La figure 1.11 représente notre activité de conception, étape par étape.

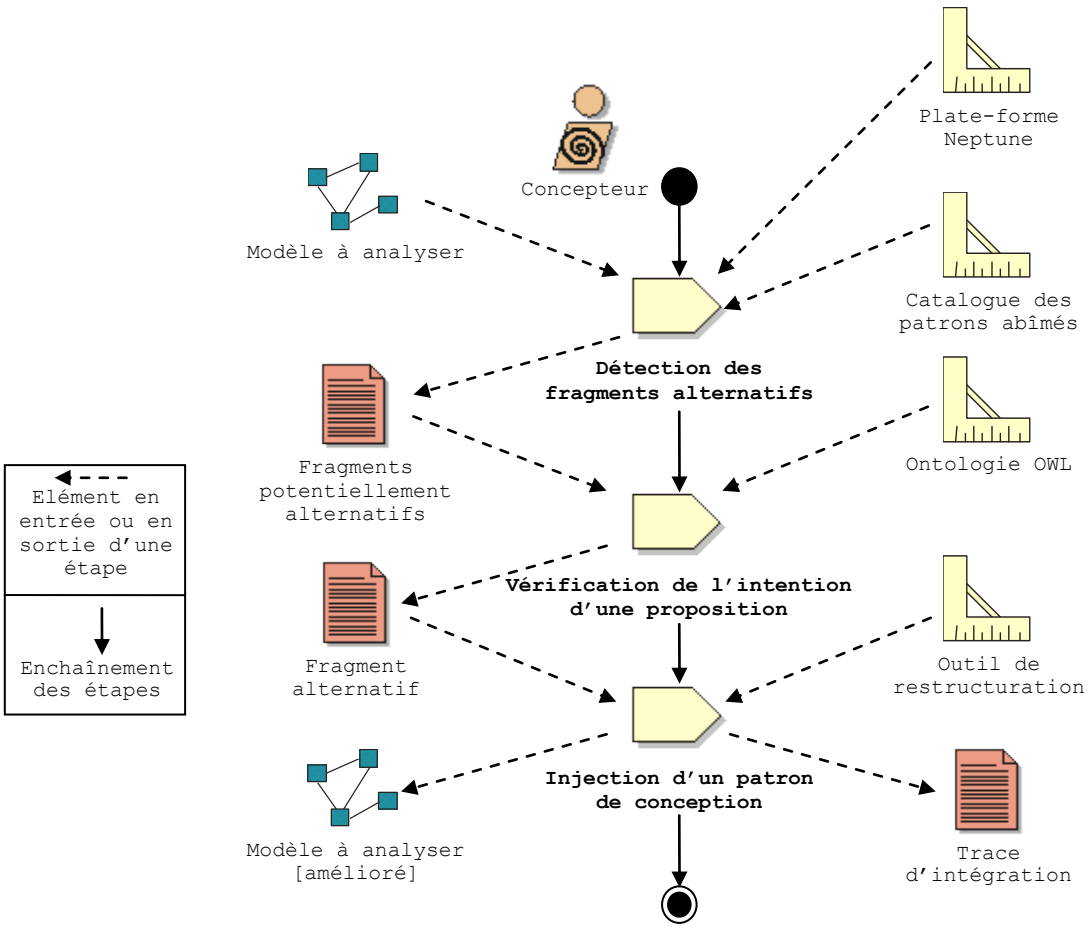


Figure 1.11 : L'activité de revue de conception

IV.2. Illustration

Pour illustrer le fonctionnement de l'activité, nous l'exécutons sur un modèle concret. Ce modèle est extrait d'un sujet de TD de programmation Java destiné à des étudiants en troisième année d'informatique. L'objectif de ce TD était de programmer un système de gestion de fichiers, à partir d'un modèle fourni dans l'énoncé.

IV.2.1. Le modèle à analyser

La figure 1.12 présente le modèle de l'énoncé du TD et constitue le modèle en entrée de notre activité.

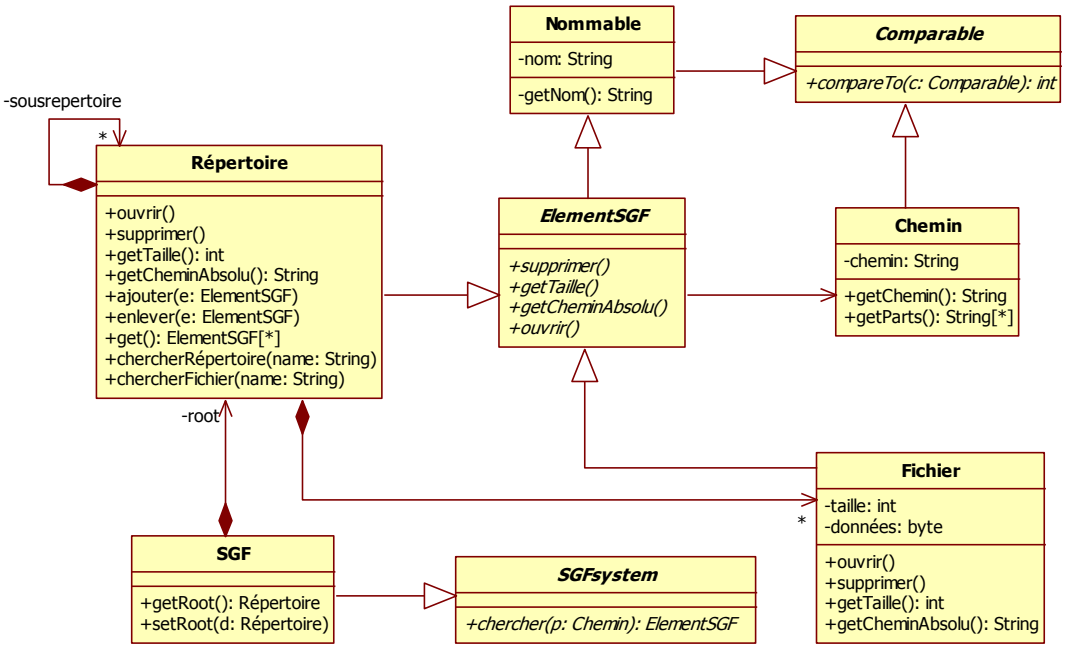


Figure 1.12 : Le modèle à analyser

Dans un premier temps, nous pouvons dire que ce diagramme de classes UML représente une architecture basique d'un système de gestion de fichiers. Les auteurs de ce modèle ont veillé à ce que les bonnes pratiques de conception par objets soient respectées :

- *L'héritage de protocole entre les classes.* La classe *ElementSGF* encapsule un protocole uniforme dont profitent les classes *Répertoire* et *Fichier* grâce à leurs relations d'héritage. Nous pouvons également remarquer que toutes les classes concrètes sont dérivées directement ou indirectement d'une classe abstraite. Cette règle renforce l'héritage d'interfaces.
- *Utilisation de la délégation*. La classe *Répertoire* gère des références, sur elle-même mais aussi sur la classe *Fichier*, tout en déléguant des actions le long de la hiérarchie de composition. Par exemple, la méthode *getTaille()* est déléguée à tous les fichiers et aux sous-répertoires qui délèguent à leurs fichiers et ainsi de suite.

A priori, un effort de bonne conception a été effectué lors de la modélisation, produisant une conception de bonne qualité. Cependant, ce modèle présente quelques défauts de conception. Bien qu'il y ait un protocole uniforme offert par la classe *ElementSGF*, la gestion des liens de composition vers les autres types de données présents dans la hiérarchie de composants est dupliquée. En effet, la classe *Répertoire* gère indépendamment les liens sur *Fichiers* et ceux sur elle-même.

Il suffit de considérer les deux scénarios d'évolution suivants pour mettre en doute notre premier avis sur la qualité de la conception :

- Le premier est l'ajout de nouveaux types terminaux dans la structure arborescente, comme des liens symboliques du type de ceux que l'on trouve dans les systèmes de fichiers UNIX. Cette évolution nécessite la gestion de nouveaux types de liens vers la classe *Répertoire* et donc impose des modifications et des duplications de code.
- Le second est l'ajout de nouveaux types non terminaux dans la structure arborescente, comme des fichiers archives. Nous pouvons dire qu'une archive a les mêmes fonctionnalités qu'un répertoire et qu'elle contient des répertoires et des fichiers. Cette évolution impose un lien réflexif sur la nouvelle classe *Archive* et une duplication de tous les liens de composition. De plus, les répertoires peuvent contenir des archives, ce qui impose également des modifications de code dans la classe *Répertoire*.

Ces deux scénarios mettent en avant un problème de découplage (chaque classe conteneur gère une partie de la composition) et une limite à l'extension (chaque modification nécessite des modifications du code existant). Ce modèle peut être donc amélioré grâce à l'injection du patron *Composite*, connu pour être efficace quant à la composition récursive et uniforme d'objets.

IV.2.2. Revue de conception

Pour pouvoir exécuter l'activité, nous avons développé le logiciel Triton, dont une capture d'écran est présentée à la figure 1.13. Il accède à la totalité du catalogue des patrons abîmés et utilise la plate-forme Neptune pour effectuer la recherche. De plus amples détails sur cet outil sont fournis dans le chapitre 4 de cette thèse.

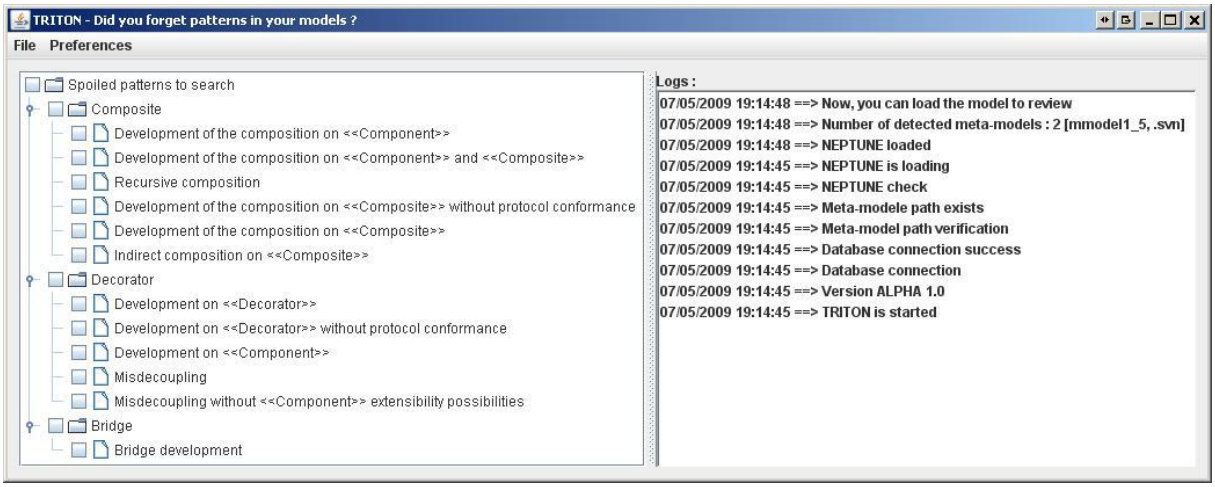


Figure 1.13 : Triton - l'outil d'exécution de l'activité

La première étape de l'activité consiste à rechercher des fragments, dans le modèle analysé, qui correspondent structurellement à des contextualisations possibles de patrons abîmés. Après le chargement du modèle à analyser dans Triton, les requêtes OCL déduites des particularités structurelles de chaque patron abîmé sont exécutées sur le modèle, selon la sélection effectuée par le concepteur dans la fenêtre principale de Triton. Ce dernier présente alors le résultat de l'exécution des requêtes sous forme de tableau, comme le montre la figure 1.14, pour notre exemple.

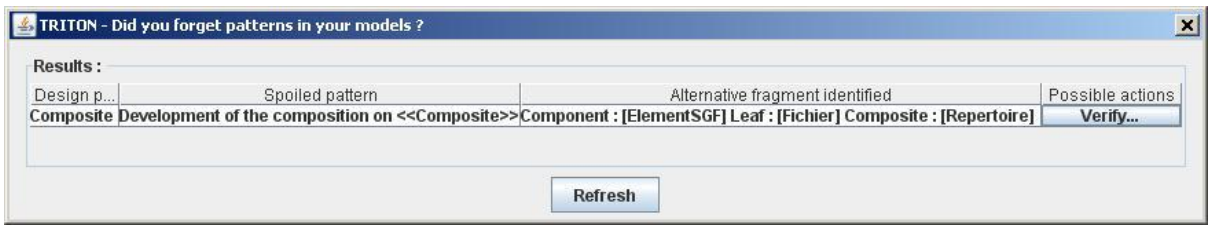


Figure 1.14 : Le résultat de la détection

Ce tableau présente les fragments détectés dans le modèle, le patron abîmé correspondant et le patron de conception rattaché. Dans le cas de notre modèle, Triton a identifié le fragment {*ElementSGF*, *Fichier*, *Répertoire*}, illustré dans la figure 1.15.

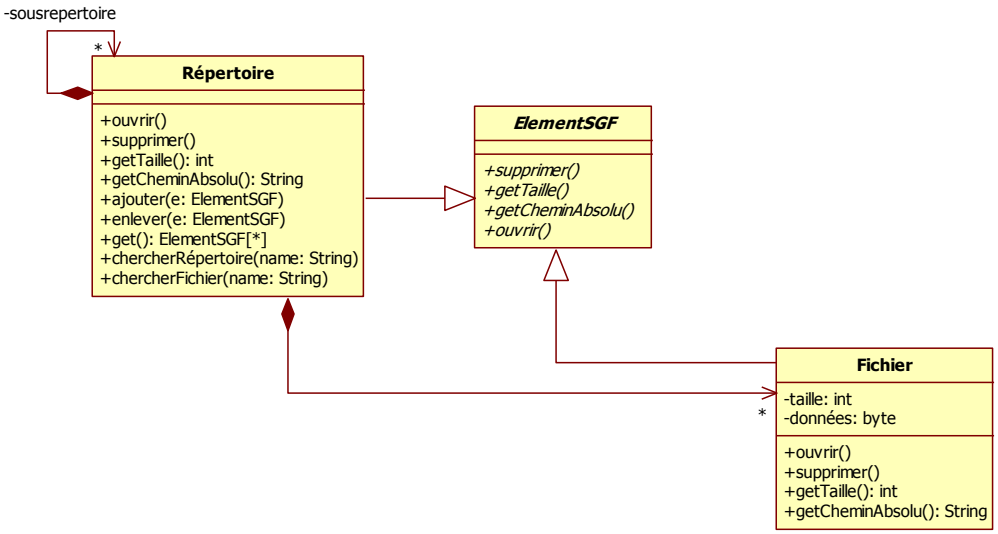


Figure 1.15 : Le fragment identifié dans le modèle

Le tableau 1.11 rappelle les particularités structurelles du patron abîmé du fragment identifié.

Participant de référence	Composite	
Particularités locales	Composite	Classe avec au moins deux compositions (0..*) dont une récursive et un lien d'héritage.
	Feuille	Classe avec au moins une composition (0..*) et un lien d'héritage.
	Composant	Classe avec au moins deux filles.
Particularités globales	Feuille	Sous-classe de *Composant* rattachée au participant de référence par une composition (0..*).
	Composant	Superclasse du participant de référence et de *Feuille*.

Tableau 1.11 : Les particularités structurelles du patron abîmé identifié

Nous pouvons remarquer que les particularités structurelles de chaque classe correspondent à celles des participants du patron abîmé. Par exemple, la classe *ElementSGF*, identifiée comme ayant les responsabilités de *Composant*, a bien au moins deux filles et est superclasse de *Répertoire*, identifiée comme *Composite*. Dans cet exemple, il n'y a qu'une classe pour chaque participant, mais le fait que les particularités locales intègrent le terme « au moins », permet de détecter des fragments ayant plusieurs classes avec les mêmes responsabilités.

À l'issue de la détection, le concepteur peut vérifier plus en détail chaque fragment identifié et ainsi passer à l'étape suivante : la vérification de l'intention et la présentation des avantages de la substitution. La figure 1.16 présente la boîte de dialogue émise par Triton pour vérifier l'intention auprès du concepteur du fragment identifié.

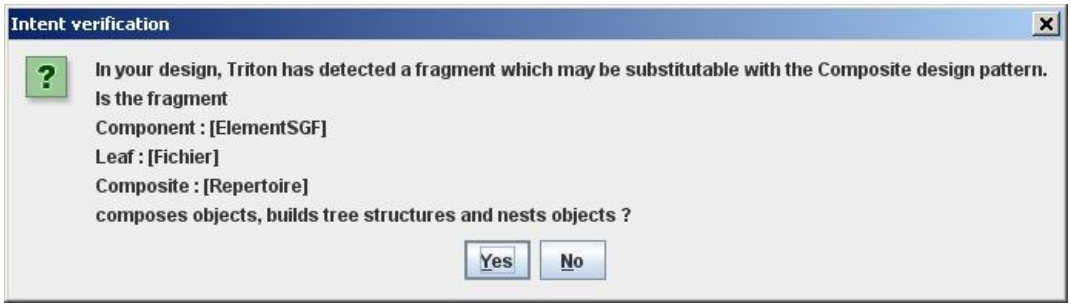


Figure 1.16 : Vérification de l'intention

Triton demande alors au concepteur de vérifier si l'intention du fragment correspond à l'intention du patron de conception. Dans notre cas, étant donné que c'est un fragment correspondant au patron abîmé du *Composite* qui a été identifié, c'est l'intention du patron *Composite* qui est présentée. Si le concepteur valide la conformité d'intention, Triton présente les points forts du patron dont le modèle profitera après l'injection du patron. Dans notre exemple, nous pouvons dire que notre fragment a bien l'intention d'emboîter et de composer hiérarchiquement des objets. Ainsi, puisque nous acceptons l'intention, Triton affiche la boîte de dialogue illustrée par la figure 1.17.

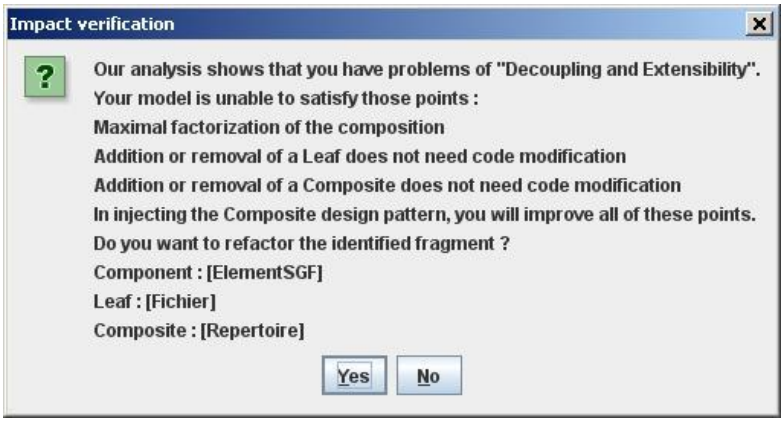


Figure 1.17 : Présentation des avantages de l'injection

L'affichage des points forts dégradés par le patron abîmé détecté permet au concepteur de prendre conscience des avantages de l'utilisation du patron de conception adéquat. Pour notre exemple, en injectant le patron de conception, le concepteur gagne en découplage et en extensibilité, ce qui correspond aux défauts que nous avions identifiés lors de l'analyse précédente. Ainsi, le concepteur a tout intérêt à remplacer le fragment par le patron *Composite*. Pour notre exemple, nous acceptons la transformation, ce qui permet à Triton d'effectuer la transformation du modèle en mémoire.

Après la transformation, le concepteur est invité à exécuter à nouveau la détection afin de vérifier si d'autres fragments sont apparus, ou ont disparu, dans le cas où plusieurs fragments auraient été identifiés lors de la première analyse. Finalement, lorsque le concepteur estime que son modèle est dans une qualité suffisante, ou si Triton n'identifie plus de fragment, un système de sérialisation de modèles permet la génération d'un nouveau fichier XMI contenant le modèle transformé.

IV.2.3. Le modèle amélioré

En fin de revue, le modèle présenté figure 1.12 est transformé pour que le patron de conception *Composite* soit injecté. Le résultat est présenté à la figure 1.18.

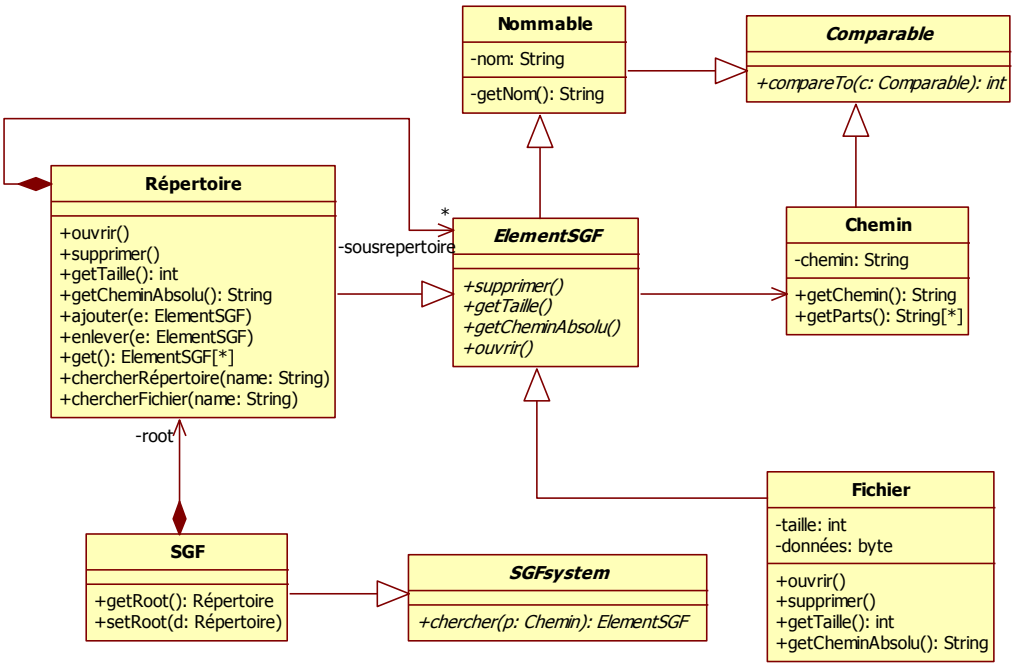


Figure 1.18 : Le modèle du système de gestion de fichiers après intégration du patron *Composite*

La transformation effectuée entre les figures 1.12 et 1.18 est schématisée dans la figure 1.19.

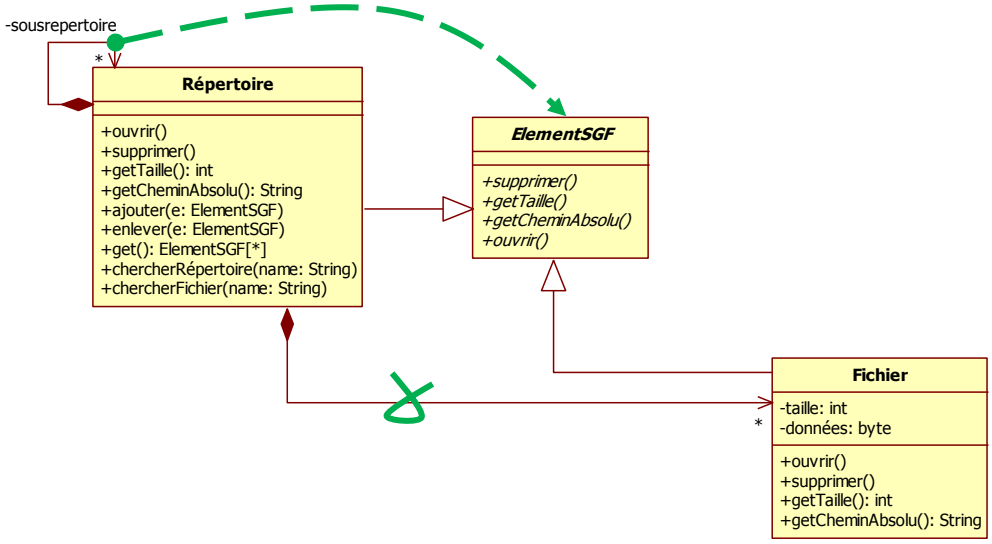


Figure 1.19 : Transformation effectuée sur le modèle du système de gestion de fichiers

Nous pouvons voir que la transformation a provoqué la factorisation de la composition en supprimant les compositions développées. Les conséquences de cette transformation se retrouvent dans la simplification du codage de la gestion de la structure arborescente et par le fait que les deux scénarios d'évolution ne nécessitent plus de modification du code existant. À la fin de l'activité, nous pouvons donc considérer que le modèle a été amélioré.

V. Conclusion

L'utilisation adéquate des patrons de conception dans un modèle est la garantie d'utiliser, pour un type de problème, une solution approuvée et validée par une communauté d'experts. De ce fait, nous considérons cette solution comme la meilleure pour résoudre ce type de problème. Cependant, le choix du patron correspondant au problème en cours de conception ainsi que son utilisation ne sont pas aisés. En effet, étant donné que les problèmes solubles par les patrons sont présentés de manière générique, il est nécessaire de décontextualiser le problème en cours de conception pour pouvoir le comparer à celui du patron, puis de contextualiser la solution proposée par le patron pour l'adapter à son propre contexte. Une mauvaise décontextualisation du problème peut provoquer le choix d'un patron non adapté, et une mauvaise contextualisation peut dégrader la qualité apportée par le patron.

Pour affranchir le concepteur de ces problèmes, des travaux existent visant à améliorer la classification et la description des patrons de conception, et définissant des techniques de vérification de leur bonne intégration. Cependant, ces travaux partent du principe que le concepteur choisit d'exploiter a priori les patrons de conception. Nous proposons une approche différente, dans le sens où nous laissons le concepteur modéliser son application sans se préoccuper de l'optimisation de son architecture. Ainsi, nous avons conçu et implémenté une activité de revue de conception, intégrable à un processus de développement, qui vérifie et corrige, avec l'aide du concepteur, les modèles de conception.

Nous avons ainsi défini le concept de « patron abîmé », qui représente une manière de résoudre un type de problème, différemment de la solution consensuelle du patron de conception. En identifiant ces mauvaises pratiques, nous ciblons des fragments de modèle remplaçables par des patrons de conception, sous réserve d'une conformité d'intention avec les patrons concernés. Un patron abîmé se caractérise par la dégradation des critères d'architecture et des facteurs de qualité logicielle du patron de conception générant la meilleure solution. Nous avons appelé ces critères et ces facteurs les « points forts » du patron de conception notamment parce qu'ils explicitent en quoi un patron de conception est la meilleure solution pour un type de problème. La dégradation des points forts par les patrons abîmés nous permet donc d'expliquer au concepteur en quoi le remplacement des fragments identifiés par des patrons de conception est plus intéressant en termes de critères architecturaux et de facteurs de qualité logicielle. Nous favorisons ainsi l'utilisation des patrons de conception et expliquons au concepteur en quoi l'utilisation de ces patrons est plus avantageuse dans son modèle.

2

Création d’une base de patrons abîmés

Le concept central de notre approche est le concept de patron abîmé. Notre technique de détection et notre activité de revue de conception reposent sur une base de patrons abîmés. Ils doivent constituer des alternatives valides aux patrons de conception, c'est-à-dire résoudre le même problème, mais d'une manière différente et par hypothèse moins optimale. Nous avons donc élaboré une méthode pour collecter un ensemble de solutions alternatives à des problèmes dont les solutions optimales reviennent à utiliser des patrons de conception [Bouhours07_b] [Bouhours07_c].

L'inexpérience de jeunes concepteurs nous a permis de constituer un catalogue de patrons abîmés, déduits d'un ensemble de solutions alternatives à des problèmes orientés sur l'intention même des patrons de conception. Nous avons ainsi obtenu des patrons abîmés pour trois des sept patrons structurels, et nous détenons un ensemble de solutions alternatives pour les patrons comportementaux.

Dans ce chapitre, nous présentons, dans une première section, les expérimentations que nous avons réalisées pour constituer notre base de patrons abîmés. Nous avons ainsi collecté des solutions alternatives à des problèmes admettant, en tant que meilleure solution, une contextualisation d'un patron de conception. Dans une deuxième section, nous nous intéressons aux résultats obtenus, en détaillant notamment les solutions alternatives obtenues pour le patron *Composite*. Nous terminons ce chapitre en abordant le problème de la décontextualisation des patrons comportementaux. Nous montrons que le procédé de décontextualisation des solutions alternatives ne peut pas toujours s'utiliser de la même manière que pour les patrons structurels.

I. Recherche de solutions alternatives

Il existe deux manières de constituer une base de patrons abîmés. La première est d'analyser les patrons de conception et d'effectuer des transformations sur leur structure afin de les dénaturer. Ainsi, en modifiant les relations entre les classes, que ce soit leur destination (de telle classe vers telle autre) ou leur type (association, héritage…), les responsabilités des classes sont transformées et ainsi la manière dont le patron résout le problème. Les modifications effectuées doivent cependant maintenir un minimum de cohérence, de manière à ce que les modèles obtenus restent valides. Il s'agit donc de modifications sans altération de l'intention. Si cette solution permet de lister de façon exhaustive l'ensemble des patrons abîmés possibles, elle risque de provoquer une explosion combinatoire, dont beaucoup 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 proposer à des concepteurs de concevoir un ensemble de solutions résolvant un problème soluble par un patron de conception, sans exploiter ledit patron. Pour ce faire, il est nécessaire de faire appel à des concepteurs inexpérimentés pour collecter un maximum de solutions alternatives. Ainsi, ces concepteurs n'ayant pas le réflexe

d'exploiter le savoir-faire existant, leur démarche n'est influencée que par leur propre expérience et ils n'utilisent pas, pour une majeure partie des cas, de bonnes pratiques de conception. En conclusion, nous considérons qu'un concepteur inexpérimenté aura statistiquement moins de chance de produire la meilleure solution à un problème qu'un concepteur expert du domaine.

Finalement, afin d'obtenir les patrons abîmés les plus pertinents, nous avons choisi d'utiliser la seconde possibilité. Ainsi, et plus schématiquement, nous avons demandé à des concepteurs novices de réinventer les patrons de conception.

I.1. Élaboration des expérimentations

Au cours de leur cursus, les étudiants en informatique découvrent en premier lieu les techniques de conception, puis de plus en plus de formations intègrent dans un second temps les patrons de conception. En général, en amont de cet enseignement, les étudiants produisent des modèles résolvant des problèmes, sans utiliser les patrons de conception. C'est à ce stade de leur cursus que nous avons demandé à des étudiants de résoudre des problèmes de conception. Ils ont ainsi produit des modèles d'après leur propre expérience et souvent entachés de défauts de conception. De plus, par delà nos besoins de patrons abîmés, ces expérimentations ont permis aux étudiants de mettre en exergue l'intérêt d'utiliser les patrons, ce qui constitue un apport pédagogique non négligeable. En effet, pendant leur formation sur les patrons, nous les avons confrontés à leurs modèles, mettant ainsi en avant les défauts de conception corrigés par les patrons de conception.

Réparties sur une durée de trois ans, nos expérimentations ont donc visé des étudiants en troisième et en cinquième année d'études en informatique. Chaque expérimentation se présentait sous la forme d'un travail personnel composé d'une dizaine d'exercices. Chaque exercice soulevait un problème de conception soluble par l'utilisation d'un patron de conception. Nous avons travaillé l'énoncé de chaque problème de manière à ce que les solutions correspondent directement à l'utilisation du patron de conception. Nous avons ainsi limité au maximum le nombre de classes non significatives afin que les étudiants ne se dispersent pas dans des conceptions trop complexes. Pour cela, nous nous sommes inspirés de la « *section motivation* » des patrons du GoF [Gamma95], ou, lorsqu'ils ne nous convenaient pas, nous avons élaboré nos propres problèmes de conception. De manière générale, cette « *section* » présente un exemple de problème soluble par le patron de conception, ainsi que sa solution sous forme de diagramme de classes, de séquence ou d'objets. Cet exemple a pour but d'aider à comprendre, sur un cas concret, le patron et ce qu'il apporte. Pour certains patrons, nous avons directement utilisé le problème tel qu'il était présenté, pour d'autres, nous l'avons modifié ou simplifié de manière à ce qu'il s'adapte au mieux aux besoins de nos expérimentations.

Nos premières expérimentations ont concerné essentiellement les patrons structurels. Le public visé était des étudiants en L3 et M2 de l'IUP *Ingénierie des Systèmes Informatiques* (ISI), puis en M2 Recherche *Sûreté du Logiciel et Calcul à Hautes Performances*. Les résultats obtenus ont été suffisants pour déduire des patrons abîmés structurels. Pour les expérimentations suivantes, nous nous sommes concentrés sur les patrons comportementaux, et ce, pour un public similaire. C'est pour ces dernières expérimentations que nous avons imposé, dans l'énoncé distribué aux étudiants, la création de diagrammes de séquence permettant d'illustrer la communication entre les objets. En l'état actuel de nos travaux, nous n'avons pas encore déduit de patrons abîmés de ces expérimentations, car le procédé de décontextualisation était de fait plus subtil que pour les patrons structurels. Ce point est abordé dans la section III de ce chapitre. De plus, pour chacune des expérimentations réalisées, nous avons intégré un ou deux problèmes solubles par des patrons créateurs, mais sans procéder, pour le moment, à l'analyse des résultats obtenus.

Les énoncés 2.1, 2.2, 2.3 et 2.4 présentent le sujet d'une des expérimentations sur les patrons comportementaux. L'intégralité des sujets des expérimentations est disponible en annexe 1.

Trouver des solutions à des problèmes de conception

Ce document propose un ensemble d'exercices de modélisation à objets. Vous devez produire un diagramme de classes UML **ainsi qu'un diagramme de séquence ou de collaboration** illustrant chaque exercice posé. Chaque diagramme doit contenir suffisamment d'informations pour démontrer que le problème est résolu (attributs, méthodes, relations, stéréotypes).

Le but de ces exercices est que vous suiviez un raisonnement qui vous est propre. Ces modélisations peuvent s'envisager de plusieurs manières différentes. Ne cherchez pas de solutions communes avec vos camarades, ni de solutions sur Internet ou dans des ouvrages de conception.

Certaines modélisations sont présentées avec des évolutions probables. Vos conceptions doivent être structurées de manière à ce que ces évolutions soient facilement intégrables. **Faites apparaître ces évolutions dans vos diagrammes**.

Les diagrammes de classes et de séquence doivent être rendus sur papier.

Exercice 1

Modéliser des classes comparables.

Soit les classes Nombre et Chaîne de caractères, ayant, entre autres, les fonctions de comparaisons suivantes : inferieur, inferieurOuEgal, superieur, superieurOuEgal, et egal, en sachant que d'autres classes peuvent avoir ces mêmes fonctions de comparaison. Vous devez veiller à ce que les protocoles soient uniformes et que le minimum de fonctionnalités soit implémenté dans les classes ayant des instances comparables.

Énoncé 2.1 : Un des questionnaires distribués aux étudiants (partie 1)

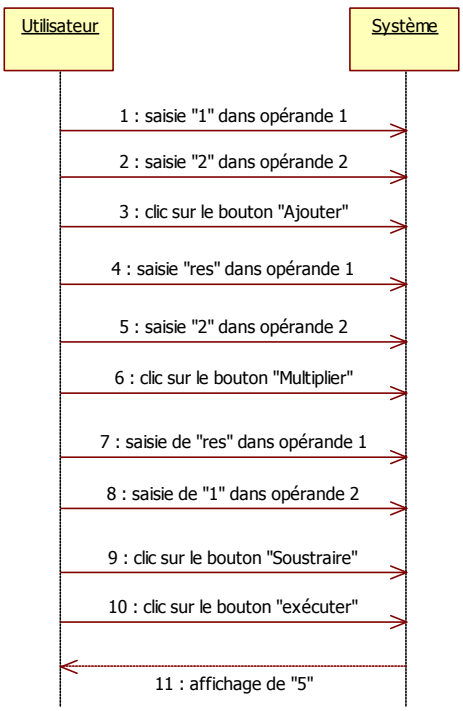
Exercice 2

Modéliser un gestionnaire d'aide d'une application Java.

Un gestionnaire d'aide permet d'afficher un message d'aide en fonction de l'objet sur lequel le client a cliqué. Par exemple, le « ? » situé quelquefois à côté du menu contextuel d'une fenêtre Windows permet d'afficher l'aide en fonction du bouton ou de la zone sur laquelle on clique. Si le bouton sur lequel on clique ne contient pas d'aide, c'est la zone contenante qui affiche son aide, et ainsi de suite. Si aucun objet ne contient d'aide, au final, le gestionnaire affiche « Pas d'aide disponible pour cette zone ». Instanciez votre diagramme de classes pour un diagramme de séquence sur l'exemple d'une fenêtre d'impression. Cette fenêtre (JDialog) est constituée d'un texte explicatif (JLabel), et d'un container (JPanel). Ce dernier contient un bouton Imprimer (JButton) et un bouton Annuler (JButton). Le bouton Imprimer contient l'aide « Lance l'impression du document ». Le bouton Annuler, le texte ainsi que la fenêtre ne contiennent pas d'aide. Enfin, le container contient l'aide « Cliquez sur l'un des boutons ». Dans le diagramme de séquence, faites apparaître les scénarii : « L'utilisateur demande l'aide du bouton Imprimer », « L'utilisateur demande l'aide du bouton Annuler », et « L'utilisateur demande l'aide du texte ».

Exercice 3

Modéliser un didacticiel pour apprendre à programmer une calculette.



Cette calculette exécute les quatre opérations arithmétiques de base. Le but de ce didacticiel est de permettre de saisir un ensemble d'opérations à effectuer séquentiellement. Le didacticiel présente un bouton par opération arithmétique, et deux champs de saisie pour les opérandes. Après chaque clic sur un bouton d'une opération, l'utilisateur a alors le choix de recommencer ou d'exécuter la suite d'opérations pour obtenir le résultat. Le diagramme de séquence ci-contre présente les interactions entre l'utilisateur et le didacticiel.

Il est probable que ce didacticiel évolue afin de permettre à l'utilisateur de supprimer la dernière opération de la liste et de prendre en compte l'opération de modulo.

Vous pouvez utiliser le scénario présenté dans le diagramme de séquence système pour illustrer votre diagramme de séquence détaillé.

Exercice 4

Modéliser l'affichage sur l'écran d'une calculatrice.

Cette calculatrice permet la saisie d'expressions, qui peuvent être « des nombres » ou « des opérations binaires » (addition, soustraction et multiplication). La calculatrice propose deux modes d'affichage : « infixés » (i.e 3_+_3) ou « préfixés » (i.e +_3_3). Après avoir saisi ses expressions, l'utilisateur peut choisir le mode d'affichage, et passer de l'un à l'autre autant de fois qu'il le désire. Il est probable que le système évolue pour que le mode d'affichage « suffixé » (i.e 3_3_+) soit ajouté.

Énoncé 2.2 : Un des questionnaires distribués aux étudiants (partie 2)

Exercice 5

Modéliser le fonctionnement d'une boîte de dialogue.

Cette boîte de dialogue contient une liste de noms et un champ de saisie en lecture seule. Lorsque l'utilisateur clique sur un nom de la liste, il apparaît automatiquement dans le champ de saisie. Tant que l'utilisateur n'a pas cliqué sur un nom de la liste (donc tant que le champ de saisie est vide), le bouton de validation de la boîte de dialogue est désactivé. Attention, **ne modélisez pas la boîte de dialogue, mais uniquement son fonctionnement, en y intégrant son affichage**, et veillez à ce que l'interconnexion entre les différents objets de la boîte de dialogue soit minimale.

Exercice 6

Modéliser un système permettant d'afficher des fenêtres vides (sans bouton, sans menu…).

Une fenêtre peut avoir plusieurs styles différents, dépendants de la plate-forme d'utilisation. On considère qu'il existe deux plates-formes, 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 spécialiser ces fenêtres en fenêtres applicatives (capables de gérer des applications) et en fenêtres iconisées (avec une icône).

Exercice 7

Modéliser un logiciel de « veille » pour les traders en bourse.

Chaque fois qu'une transaction boursière a lieu, une vente est émise et est ajoutée à un pool de ventes. Les traders doivent être mis immédiatement au courant. Ils ont à leur disposition, un jeu d'applets activables à tout instant permettant :

- soit de surveiller en temps réel le nombre de ventes effectuées pendant une journée boursière,
- soit de surveiller en temps réel le total des ventes en actions effectuées pendant une journée boursière,
- soit de surveiller en temps réel le total des nombres d'actions vendues pendant une journée boursière.

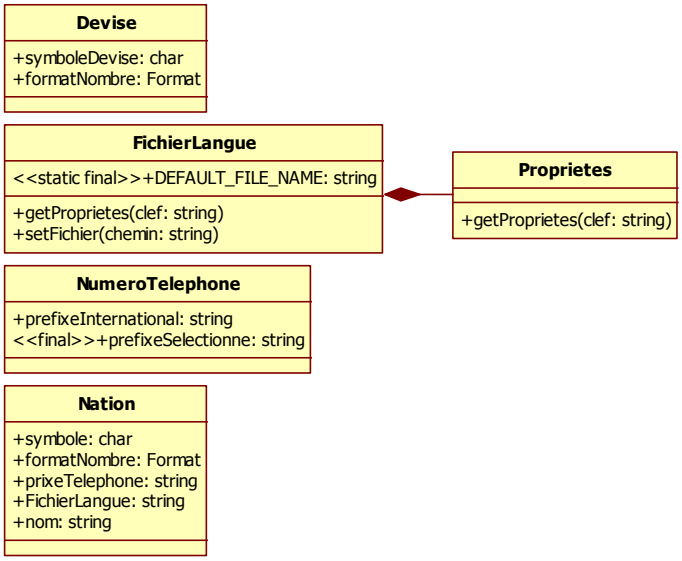
Ce logiciel devra évoluer afin de prendre en compte un pool d'achats (de la même façon que les ventes). D'autre part, une demande des traders est de pouvoir surveiller les ventes et les achats d'un titre particulier à la demande.

Exercice 8

Compléter le diagramme de classes.

Vous devez permettre à tout programme client du sous-système fourni à la page suivante de mettre à jour, en fonction d'une nation choisie, le symbole courant, le formatage des nombres, le nom du fichier des propriétés, ainsi que le préfixe international par défautd'un numéro de téléphone. Dans le diagramme de séquence, mettez en évidence les interactions de la méthode setNation(String name) [appelée par un client] qui permet de récupérer ces paramètres.

Énoncé 2.3 : Un des questionnaires distribués aux étudiants (partie 3)



Exercice 9

Modéliser l'utilisation d'un formulaire de saisie.

Ce formulaire est destiné à recueillir des informations sur des utilisateurs. Une liste de champs (de type String) est gérée par l'administrateur, de manière à en ajouter ou en supprimer en fonction de besoins statistiques. À chaque demande d'affichage, le formulaire est construit dynamiquement en fonction de cette liste. N'oubliez pas de mémoriser les informations des utilisateurs recueillies par le formulaire. Il n'est pas nécessaire de fournir un diagramme de séquence pour cet exercice.

Énoncé 2.4 : Un des questionnaires distribués aux étudiants (partie 4)

Pour cette expérimentation, les meilleures solutions à chacun de ces problèmes sont respectivement l'utilisation des patrons : *Patron de méthode*, *Chaîne de responsabilités*, *Commande*, *Visiteur*, *Médiateur*, *Pont*, *Observateur*, *Façade* et *Singleton*. Destinée à des étudiants de L3 de l'IUP ISI de Toulouse et à des étudiants de la licence professionnelle *Systèmes Informatiques et Logiciels, spécialité Qualité du Logiciel* de l'IUT A de Toulouse, cette expérimentation est principalement axée sur les patrons comportementaux. Lors des premières expérimentations, nous avons cependant intégré quelques patrons structurels dont les solutions obtenues ne nous semblaient pas encore satisfaisantes.

Au final, sur les trois années, nous avons couvert les sept patrons structurels, les onze patrons comportementaux et trois des patrons créateurs. Nous avons ainsi obtenu mille trois cents modèles qu'il a été nécessaire d'analyser afin d'éliminer les conceptions erronées et les doublons. L'analyse des solutions alternatives des patrons structuraux nous a permis de déduire les premiers patrons abîmés. L'analyse des solutions alternatives des patrons comportementaux n'a pas encore abouti à l'identification de patrons abîmés de par le traitement particulier qu'ils nécessitent. Le tableau 2.1 présente les statistiques issues de nos expérimentations.

Niveau d'étude	Patrons concernés	Étudiants participants	Nombre de conceptions obtenues	Conceptions erronées	Conceptions utilisant le patron de conception	Solutions alternatives	Patrons abîmés déduits
M2P	Adaptateur, Pont, Composite, Décorateur, Façade, Poids mouche, Procuration, Fabrique abstraite, Monteur	16	144	61 (42%)	17 (12%)	66 (46%)	15
L3	Adaptateur, Pont, Composite, Décorateur, Façade, Poids mouche, Procuration, Fabrique abstraite, Monteur	30	370	176 (47%)	24 (7%)	170 (46%)	11
M2R	Adaptateur, Pont, Composite, Décorateur, Façade, Poids mouche, Procuration, Fabrique abstraite, Monteur	1	9	4 (44%)	4 (44%)	1 (11%)	1
LP	Adaptateur, Pont, Composite, Décorateur, Façade, Poids mouche, Procuration, Fabrique abstraite, Monteur	9	81	18 (22%)	25 (31%)	38 (47%)	13
L3	Médiateur, Observateur, Singleton, Façade, Interprète, Itérateur, Mémento, État, Stratégie, Commande	28	280	143 (51%)	100 (36%)	37 (13%)	N/A
M2P	Médiateur, Observateur, Singleton, Façade, Interprète, Itérateur, Mémento, État, Stratégie, Commande	5	50	18 (36%)	22 (44%)	10 (20%)	N/A
L3	Patron de méthode, Chaîne de responsabilité, Commande, Visiteur, Médiateur, Pont, Observateur, Façade, Singleton	29	261	162 (62%)	71 (27%)	28 (11%)	N/A
LP	Patron de méthode, Chaîne de responsabilité, Commande, Visiteur, Médiateur, Pont, Observateur, Façade, Singleton	18	162	65 (40%)	62 (38%)	35 (22%)	N/A
	Total :	**136**	**1357**	**647**	**325**	**385**	**16 distincts**

Tableau 2.1 : Statistiques issues de nos expérimentations

Chaque ligne de ce tableau représente une expérimentation sur un ensemble d'étudiants d'un même niveau d'études. Nous pouvons remarquer que le nombre de patrons abîmés déduits est faible par rapport aux solutions alternatives, ce qui montre le nombre de conceptions similaires. De plus, il est intéressant de souligner qu'un nombre non négligeable d'étudiants a réussi à concevoir directement les meilleures solutions à certains problèmes. Enfin, la colonne des patrons abîmés pour les expérimentations sur les patrons comportementaux n'est pas encore renseignée. Notre procédé de décontextualisation est, à ce jour, uniquement centré sur l'aspect structurel des patrons. Nous donnons cependant nos premières analyses sur les solutions alternatives aux patrons comportementaux à la section III de ce chapitre et quelques éléments nous permettant d'étendre le procédé de décontextualisation aux aspects comportementaux d'un patron.

I.2. Limites de notre approche par expérimentations

Notre méthode de collecte des patrons abîmés présente, dans sa forme actuelle, deux limites. La première concerne la collecte par expérimentation, la deuxième l'analyse manuelle des solutions alternatives.

- Collecter des solutions alternatives auprès de concepteurs n'ayant pas une expérience affirmée des patrons de conception correspond à une démarche qui nous permet d'exploiter un grand nombre de solutions. Cependant, la participation d'étudiants suivant un même cursus se traduit par l'obtention de résultats très proches lorsque les problèmes se complexifient, et le manque de compétences en modélisation par objets se traduit par une difficulté à résoudre les problèmes. Les étudiants d'une même promotion effectuent les mêmes erreurs. Ayant eu la même formation théorique et technique, les mêmes défauts de conception se retrouvent dans leurs modèles, limitant ainsi le nombre de solutions alternatives différentes.
- Pour constituer notre base de patrons abîmés, nous avons analysé manuellement chaque solution proposée. Une telle analyse reste manuelle, car il semble difficile d'automatiser l'examen d'un modèle à partir d'un simple diagramme de classes. Pour les patrons structurels, l'effort n'est pas très conséquent puisque seule la structure de la solution est significative, à l'inverse des patrons comportementaux qui mettent en jeu la cinématique des échanges de messages entre les objets.

Afin d'éviter la multiplication des doublons et d'augmenter la diversité des alternatives proposées, il convient d'engager une expérimentation à plus large échelle en touchant des concepteurs de tout horizon. L'utilisation d'un site web collaboratif de partage de problèmes et de solutions alternatives permettrait d'identifier ainsi les patrons abîmés les plus fréquents, et plus largement, les mauvaises pratiques de conception. De plus, ce site web permettrait l'émergence d'une communauté d'experts ouvrant une zone de partage de « mauvaises pratiques ». Nous avons donc créé un site web collaboratif donnant accès à l'ensemble des patrons abîmés du catalogue. Ce site présente chaque patron de conception du GoF avec sa liste des patrons abîmés. Chaque patron abîmé y est décrit avec ses particularités structurelles et une justification de sa dégradation. Le site prévoit également une zone de dépôt de problèmes et de solutions alternatives. Chaque dépôt effectué est soumis à un comité examinant sa validité et son intérêt en tant que patron abîmé. Ainsi, ce site offre la possibilité de créer une communauté d'experts pour l'amélioration de modèles par utilisation des patrons de conception. Ce site, à l'état de prototype, est accessible sur [GOPROD08] où le terme « modèle alternatif », utilisé au début de notre étude, correspond au terme « patron abîmé » présenté dans cette thèse.

II. Analyse des résultats

De l'ensemble des solutions proposées par les étudiants pour les patrons structurels, nous avons sélectionné seize solutions alternatives à trois patrons structurels. Nous avons attribué à chaque solution alternative un identifiant et un nom décrivant en quelques mots les défauts de conception constatés. Comme nous ne nous intéressons, pour l'instant, qu'à l'architecture des modèles, nous avons éliminé les méthodes et les attributs de chacune des solutions présentées. Nous partons du postulat que les attributs et les méthodes indispensables à l'exécution du modèle sont présents dans les classes concernées.

Dans la première partie de cette section, nous analysons, dans un souci de complétude, l'intégralité des solutions alternatives au problème du *Composite*. Nous mettons en avant les différences entre chacune des solutions, ainsi que leurs défauts de conception. Nous détaillons, en deuxième partie, les cas particuliers des autres patrons structurels. Nous montrons par exemple un patron abîmé indétectable structurellement. Dans la dernière partie, nous donnons l'intégralité des patrons abîmés structurels que nous avons déduits de nos expérimentations, présentant ainsi l'état actuel de notre catalogue.

II.1. Analyse complète des solutions alternatives au *Composite*

Pour le patron *Composite*, dont l'intention est de « *composer et d'emboîter les objets ainsi que de construire des structures arborescentes* » [Kampffmeyer07], nous avons simplement posé le problème de composition récursive décrit dans l'énoncé 2.5.

Modéliser un système permettant de dessiner un graphique, sachant qu'un graphique est composé de lignes, de rectangles, de textes et d'images. Une image peut-être elle-même composée d'autres images, de lignes, de rectangles et de textes.

Énoncé 2.5 : Problème soluble par le patron *Composite*

La solution optimale à ce problème est le patron *Composite*. Ce dernier et sa contextualisation sont présentés dans la figure 2.1.

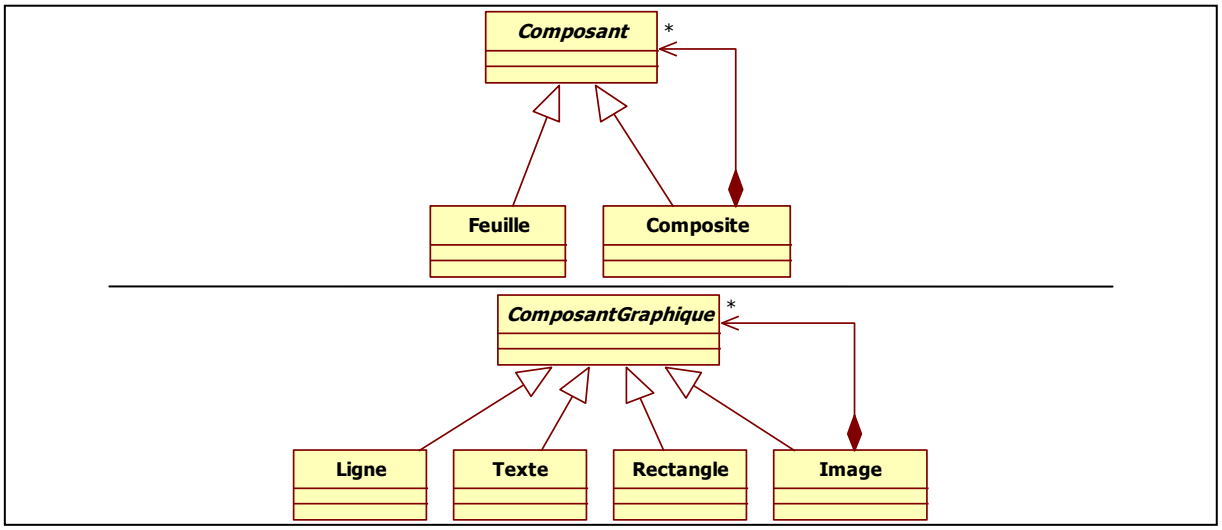


Figure 2.1 : Le patron *Composite* et sa contextualisation sur le problème posé

Ligne, *Texte* et *Rectangle* ne peuvent, d'après le problème, qu'être des composants de base. Ce sont donc les éléments terminaux de la composition hiérarchique. Ils possèdent les responsabilités du participant *Feuille*. La classe *Image* représente la structure composée récursivement et a ainsi les responsabilités du participant *Composite*. Enfin, nous avons ajouté une classe *ComposantGraphique* qui factorise la composition, sert de point d'entrée à un éventuel client et uniformise les protocoles de tous les objets inclus dans la hiérarchie de la composition. Nous pouvons dire que les points forts de ce patron sont le *Découplage*, l'*Extensibilité* et l'*Uniformité des protocoles*.

Parmi les étudiants, 11% ont résolu le problème en utilisant le patron. La plupart étaient des étudiants en M2. Il semblerait qu'en cherchant à utiliser un maximum de bonnes pratiques de conception, les étudiants ont réinventé le patron *Composite*. Parmi toutes les solutions valides, nous avons pu déduire six solutions alternatives structurellement différentes du *Composite*.

La première est présentée dans la figure 2.2. Parfois, dans les solutions alternatives présentées dans cette section, les noms des classes sont différents de la meilleure solution, car il s'agit de noms donnés par les étudiants. Nous les avons conservés dans les solutions alternatives car ils reflètent les responsabilités sous-entendues.

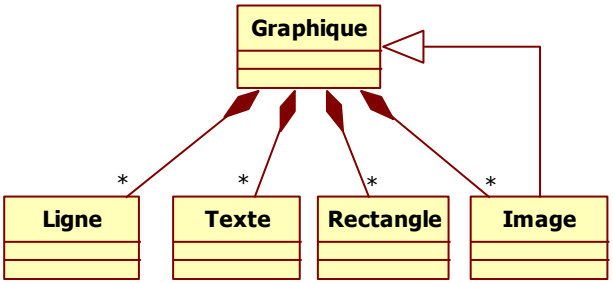


Figure 2.2 : Solution alternative 1 = Développement de la composition sur <<*Composant*>>

La composition des classes est exprimée, dans la solution alternative 1, par l'utilisation de la relation de composition sur la classe *Graphique* couplée à la relation d'héritage. Cette solution est valide, même si la structure impose des duplications de code pour la classe *Graphique*. En effet, tous les liens de composition sont mémorisés dans cette classe, qui devra ainsi gérer la totalité de la composition hiérarchique. Il est possible de remarquer dans le tableau 2.2 que l'allocation de toutes les responsabilités de la composition à la classe *Graphique* invalide le point fort « découplage et extensibilité ».

1. Découplage et Extensibilité	
✗	1.1. Factorisation maximale de la composition
✗	1.2. L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code
✗	1.3. L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code
2. Uniformité des protocoles	
✗	2.1. Protocole uniforme sur les opérations des objets composés
✓	2.2. Protocole uniforme sur la gestion de la composition
✓	2.3. Accès unique pour la classe client

Tableau 2.2 : Les points forts du patron *Composite* perturbés par la solution alternative 1

La solution alternative 2 est présentée dans la figure 2.3.

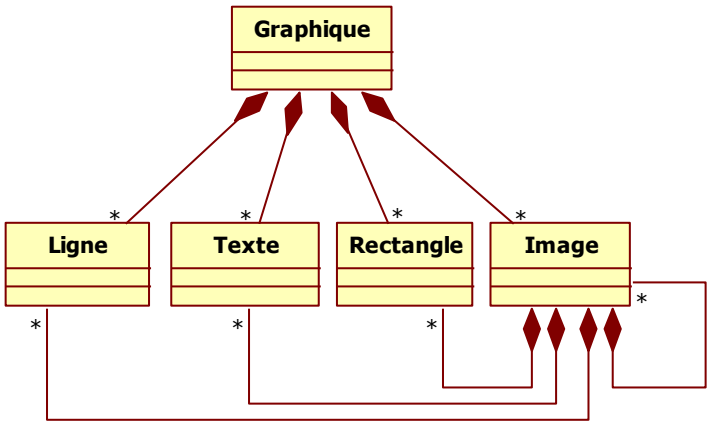


Figure 2.3 : Solution alternative 2 = Développement de la composition sur <<*Composant*>> et sur <<*Composite*>>

Les étudiants qui ont proposé cette solution ont effectué une modélisation « mot à mot » du problème posé. La classe *Graphique* est composée de toutes les autres, et ce schéma est répété pour la classe *Image*, s'incluant de plus elle-même. Cette solution est donc valide puisqu'elle respecte les données du problème. Son principal défaut, caractérisé par l'absence de lien d'héritage, est la multiplication maximale des liens de composition ce qui provoque, au moment du codage, des duplications de code et de liens. En analysant au tableau 2.3, les points forts du patron dégradés par cette solution, on constate cependant que le point d'accès unique pour la classe client est conservé.

1. Découplage et Extensibilité	
✗	1.1. Factorisation maximale de la composition
✗	1.2. L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code
✗	1.3. L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code
2. Uniformité des protocoles	
✗	2.1. Protocole uniforme sur les opérations des objets composés
✗	2.2. Protocole uniforme sur la gestion de la composition
✓	2.3. Accès unique pour la classe client

Tableau 2.3 : Les points forts du patron *Composite* perturbés par la solution alternative 2

La solution alternative 3 est présentée dans la figure 2.4.

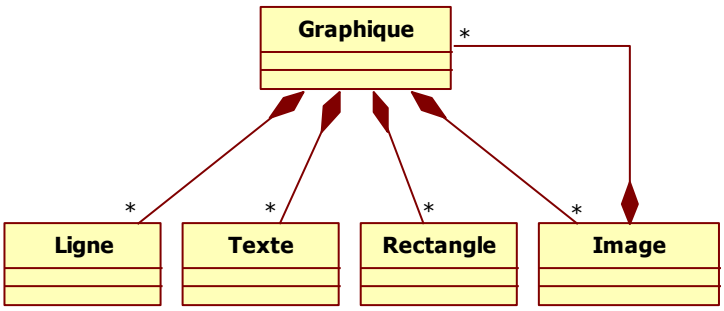


Figure 2.4 : Solution alternative 3 = Composition récursive

Sur cette solution, comme le montre le tableau 2.4, l'intégralité des points forts du patron sont perturbés, puisqu'il y a perte du point d'accès unique pour la classe client par rapport à la solution 2. Cette solution reste néanmoins valide, puisque la composition récursive est possible et que les classes *Feuilles* constituent des objets terminaux.

1. Découplage et Extensibilité	
✗	1.1. Factorisation maximale de la composition
✗	1.2. L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code
✗	1.3. L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code
2. Uniformité des protocoles	
✗	2.1. Protocole uniforme sur les opérations des objets composés
✗	2.2. Protocole uniforme sur la gestion de la composition
✗	2.3. Accès unique pour la classe client

Tableau 2.4 : Les points forts du patron *Composite* perturbés par la solution alternative 3

Le fort couplage des classes induit par les liens de composition implique une duplication de code, a priori plus faible que celle observée pour la solution alternative 2. En effet, même s'il n'y a pas d'utilisation de lien d'héritage, le fait qu'*Image* soit composée de *Graphique* provoque une sorte de factorisation des compositions entre *Image* et les autres classes *Ligne*, *Texte* et *Rectangle*. Cette factorisation réduit le nombre de listes d'objets à mémoriser.

La solution alternative 4 est présentée dans la figure 2.5.

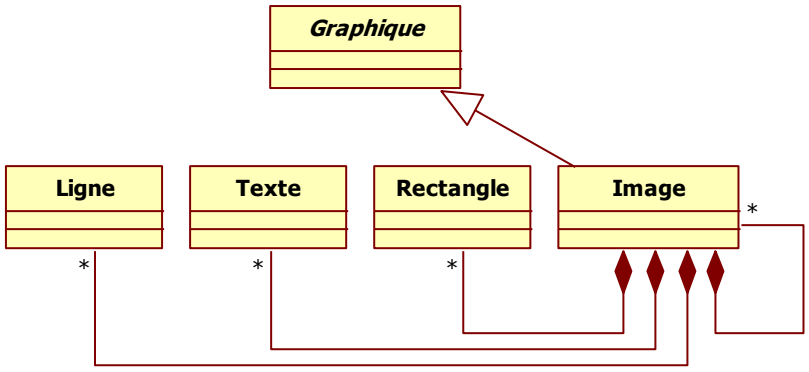


Figure 2.5 : Solution alternative 4 = Développement de la composition sur <<*Composite*>> sans conformité de protocole

Cette quatrième solution alternative met en évidence, comme précédemment, un grand nombre de liens de composition, mais exploite un lien d'héritage pour uniformiser le protocole de la gestion de la composition. Ainsi, c'est la classe *Image* qui gère l'intégralité de la composition. L'absence d'héritage entre *Ligne*, *Texte*, *Rectangle* et *Graphique* est dommageable, car cela induit à nouveau une forte duplication de code liée à un usage excessif de la délégation ainsi qu'une limite à l'extensibilité du modèle. Le tableau 2.5 illustre ce phénomène qui résulte du seul lien d'héritage identifié.

1. Découplage et Extensibilité	
✗	1.1. Factorisation maximale de la composition
✗	1.2. L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code
✗	1.3. L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code
2. Uniformité des protocoles	
✗	2.1. Protocole uniforme sur les opérations des objets composés
✓	2.2. Protocole uniforme sur la gestion de la composition
✓	2.3. Accès unique pour la classe client

Tableau 2.5 : Les points forts du patron *Composite* perturbés par la solution alternative 4

La solution alternative 5 est présentée dans la figure 2.6.

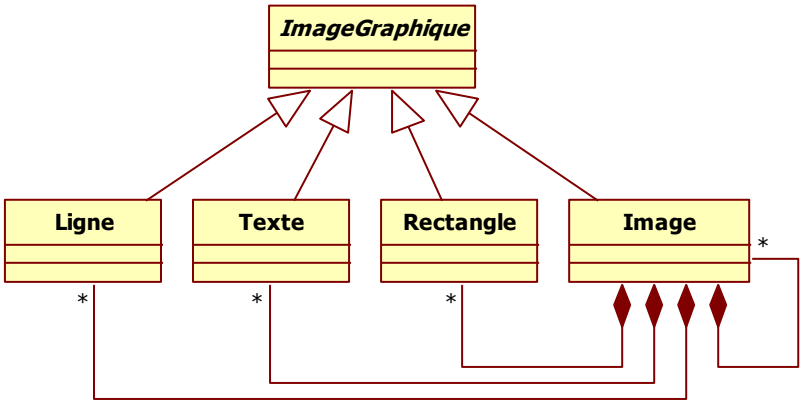


Figure 2.6 : Solution alternative 5 = Développement de la composition sur <<*Composite*>>

Cette solution est très proche de la solution 4 en corrigeant le manque de factorisation du protocole. Maintenant que toutes les classes héritent de *Graphique*, l'uniformité du protocole est acquise, comme le montre le tableau 2.6. Cependant, l'utilisation des liens d'héritage n'est pas maximale puisque les liens de composition n'ont pas été factorisés.

1. Découplage et Extensibilité	
✗	1.1. Factorisation maximale de la composition
✗	1.2. L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code
✗	1.3. L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code
2. Uniformité des protocoles	
✓	2.1. Protocole uniforme sur les opérations des objets composés
✓	2.2. Protocole uniforme sur la gestion de la composition
✓	2.3. Accès unique pour la classe client

Tableau 2.6 : Les points forts du patron *Composite* perturbés par la solution alternative 5

La solution alternative 6 est présentée dans la figure 2.7.

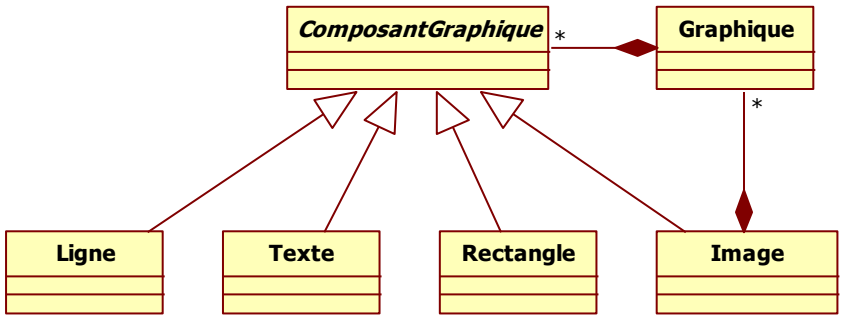


Figure 2.7 : Solution alternative 6 = Composition indirecte sur <<*Composite*>>

La solution alternative 6 est probablement la plus originale, car leurs auteurs ont rajouté une classe dont les responsabilités restent obscures, comme s'ils avaient eu besoin de réifier la composition. En effet, en l'absence de méthode, il est difficile d'entrevoir quelles sont les responsabilités de cette classe dans la hiérarchie de composition. Les compositions sont fonctionnelles et un effort de factorisation est même constaté. Si nous supprimons cette classe du modèle et que nous relions *Image* à *ComposantGraphique*, nous obtenons la contextualisation du patron *Composite*. Pourtant, l'impact de cette nouvelle classe sur les points forts du patron n'est pas négligeable, comme illustré dans le tableau 2.7.

1. Découplage et Extensibilité	
✗	1.1. Factorisation maximale de la composition
✓	1.2. L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code
✓	1.3. L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code
2. Uniformité des protocoles	
✓	2.1. Protocole uniforme sur les opérations des objets composés
✗	2.2. Protocole uniforme sur la gestion de la composition
✗	2.3. Accès unique pour la classe client

Tableau 2.7 : Les points forts du patron *Composite* perturbés par la solution alternative 6

Les alternatives au patron *Composite* ont mis en exergue des idées intéressantes pour la composition des objets. Les défauts les plus fréquents sont un manque de factorisation des relations de composition et l'absence d'interface commune que doit proposer le *Composant*. Dans l'ensemble des expérimentations, cet énoncé est celui qui a généré le moins de conceptions erronées chez les étudiants. La majorité des solutions analysées se sont réparties assez équitablement entre les cinq premières solutions alternatives.

II.2. Autres expérimentations pour les patrons structurels

A l'instar du patron *Composite*, nous avons obtenu un ensemble non négligeable de solutions alternatives intéressantes pour les patrons *Décorateur* et *Pont*. Cependant, quelques solutions alternatives à ces patrons constituent des cas particuliers que nous détaillons maintenant.

II.2.1. Une solution alternative non détectable structurellement

Pour le patron *Décorateur*, nous avons considéré l'énoncé 2.6 suivant.

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.

Énoncé 2.6 : Problème soluble pas le patron Décorateur

Ce problème cible bien l'intention du patron qui consiste à « *composer des objets et à contrôler dynamiquement de nouvelles fonctionnalités* » [Kampffmeyer07]. La dernière phrase de l'énoncé a été ajoutée pour inciter les étudiants à réfléchir aux meilleures manières de concevoir un modèle facilement extensible. La solution optimale à ce problème est présentée en figure 2.8 avec la structure du patron *Décorateur*.

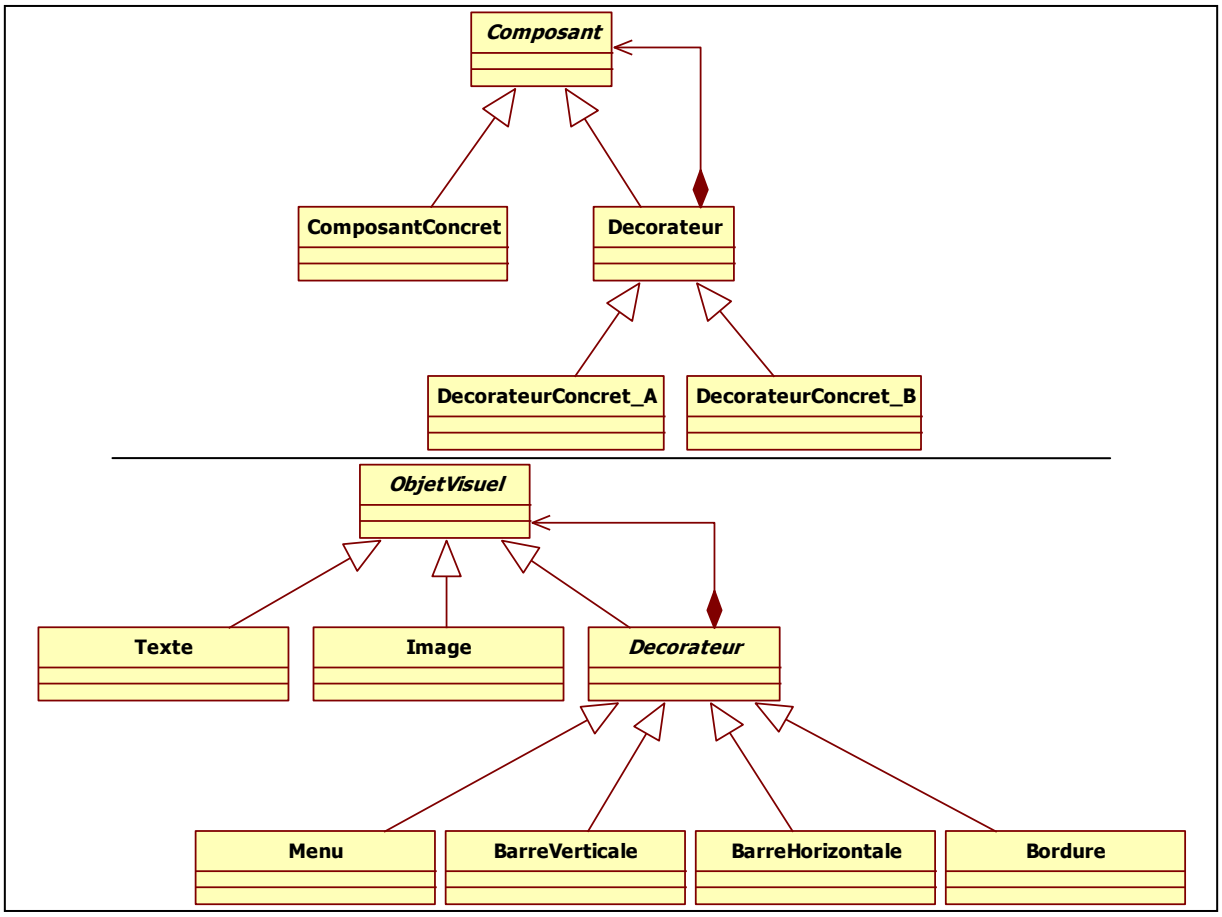


Figure 2.8 : Le patron *Décorateur* et sa contextualisation sur le problème posé

En toute logique, les classes *Texte* et *Image* constituent les objets à décorer et ont donc les responsabilités du participant *ComposantConcret*. Nous avons défini la superclasse *Decorateur* pour factoriser les protocoles de décoration et faciliter l'ajout d'un nouveau *DecorateurConcret*. Grâce à la relation de composition entre *Decorateur* et *Composant*, la décoration à la volée est possible, ce qui permet de décorer en une seule instruction un objet avec plusieurs décorateurs, même s'ils n'ont aucune connaissance les uns des autres, comme le montre le code source 2.1.

```
new BarreVerticale(new Menu(new Bordure(new Texte())))
```

Code source 2.1 : Décoration à la volée

La factorisation des protocoles de chaque classe et la structure composite pour la décoration nous permettent d'affirmer que les points forts du patron sont l'*Extensibilité*, le D*écouplage entre les objets décorateurs et les objets à décorer* et la *Gestion de la décoration lors de l'exécution* avec le minimum de classes *Décorateur*.

Pour résoudre ce problème, aucun étudiant n'a mis en évidence le patron *Décorateur*, même si beaucoup s'en sont approchés. Ce patron est difficile à appréhender structurellement, car sa force se trouve en partie dans la communication implicite entre décorateurs et objets à décorer. Nous avons tout de même pu isoler pour ce problème sept patrons abîmés différents, dont un a des particularités structurelles empêchant de le détecter. La solution alternative ayant permis de le générer est présentée dans la figure 2.9.

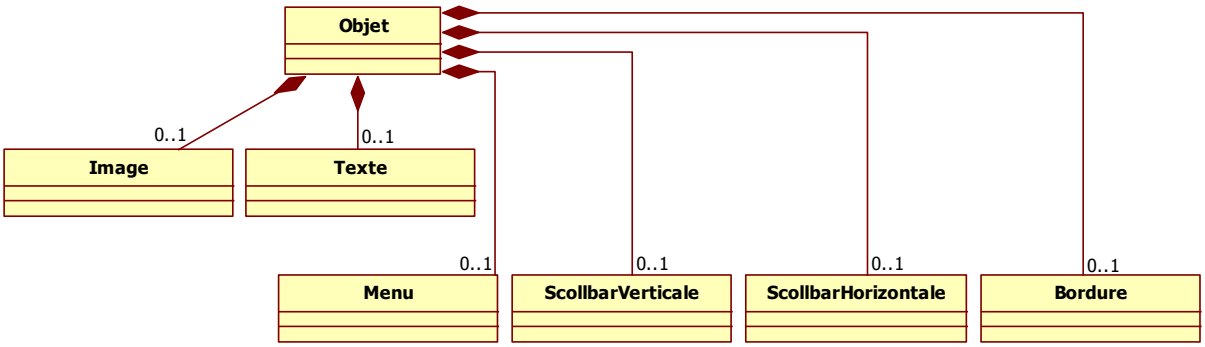


Figure 2.9 : Solution alternative 4 = Développement maximal sur <<*Composant*>>

Dans cette solution, la classe *Objet* mémorise l'intégralité des objets à décorer et des décorateurs, ce qui lui confère une trop grande responsabilité, par rapport à l'exigence d'extensibilité. Cette solution reste cependant valide, car il est possible d'activer depuis la classe *Objet* un objet à décorer, soit *Image*, soit *Texte*, puis les décorations nécessaires. Le tableau 2.8 montre que ce modèle ne perturbe pas tous les points forts du patron, dès lors que le nombre de classes *Decorateur* reste minimal, et qu'un découplage entre les objets à décorer et les décorateurs est mis en place.

1. Extensibilité	
✗	1.1. L'ajout ou la suppression d'un *Décorateur* ne nécessite pas de modification de code
✗	1.2. L'ajout ou la suppression d'un objet à décorer ne nécessite par de modification de code
2. Découplage entre les objets décorateurs et les objets à décorer	
✓	2.1. Nombre minimum de classes *Décorateur*
✗	2.2. Factorisation maximale entre les décorateurs et les objets à décorer
3. Gestion de la décoration lors de l'exécution	
✓	3.1. Les objets à décorer n'ont aucune connaissance des décorateurs
✗	3.2. Un décorateur peut être décoré par un autre décorateur

Tableau 2.8 : Les points forts du patron *Décorateur* perturbés par la solution alternative 4

Il n'est cependant pas possible de détecter cette solution alternative en utilisant la démarche que nous présentons dans le chapitre 3. Nous utilisons les particularités structurelles locales et globales de chaque participant du patron abîmé pour détecter les fragments alternatifs concernés. Dans cette solution, deux des trois participants : *ComposantConcret* et *DecorateurConcret* ont leurs particularités locales identiques. Ainsi, le patron abîmé peut se résumer structurellement par « *trois classes agrégées* », ce qui peut être potentiellement le cas de beaucoup de classes d'un modèle.

De manière générale pour le problème du *Décorateur*, la décoration a été modélisée en utilisant des liens de composition. À l'exception de la décoration à la volée, les solutions alternatives proposées respectent les données de l'énoncé. Ainsi, nous avons considéré que leur intention coïncidait avec l'intention du patron *Décorateur*.

II.2.2. Des solutions alternatives déjà présentées dans le GoF

L'intention du patron *Pont* est de « *découpler l'implémentation, l'abstraction et les interfaces* » [Kampffmeyer07] dans l'objectif de pouvoir modifier indépendamment chaque objet découplé. Nous avons alors posé un problème orienté découplage présenté dans l'énoncé 2.7.

Modéliser un système permettant d'afficher à l'écran des fenêtres vides, des fenêtres applicatives et des fenêtres avec des icônes. Une fenêtre peut avoir des styles différents dépendants de la plate-forme utilisée. Nous considérons deux plates-formes : XWindow et PresentationManager. Le code du client doit être écrit indépendamment et sans aucune connaissance de la future plate-forme d'exécution.

Énoncé 2.7 : Problème soluble par le patron *Pont*

La solution optimale de ce problème est présentée dans la figure 2.10.

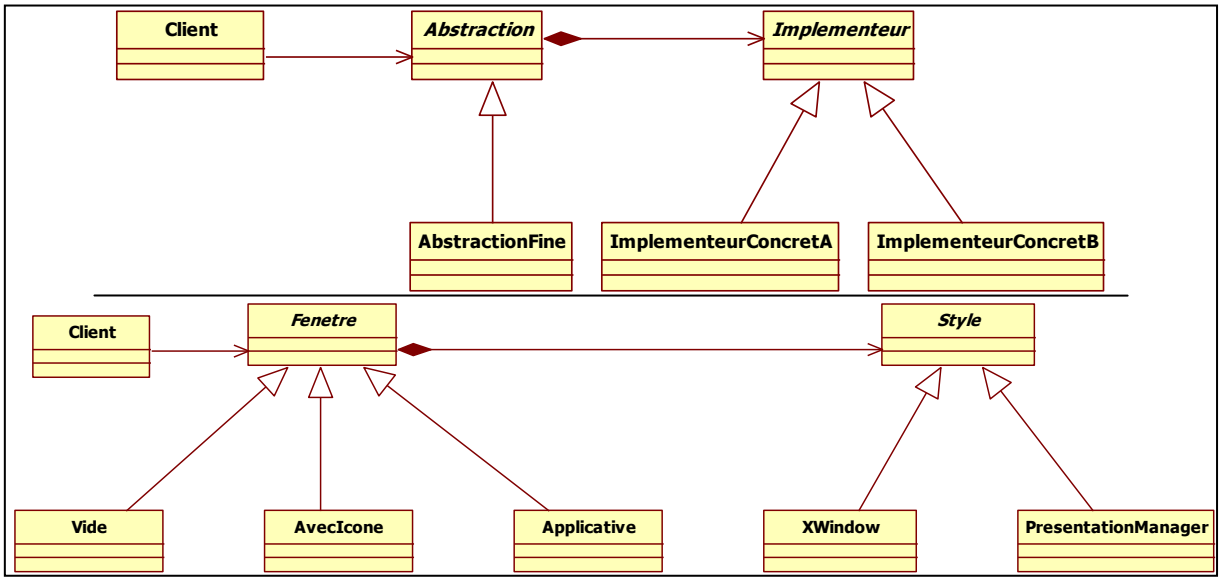


Figure 2.10 : Le patron *Pont* et sa contextualisation sur le problème posé

Le problème demande de modéliser un système d'affichage de fenêtres dont le style dépend de la plate-forme utilisée. Spécialiser chaque type de fenêtre pour chaque plate-forme disponible impose des limites d'extensibilité et de codage. La séparation des fenêtres, qui savent placer leurs composants (icône, bouton, menu…), des différentes plates-formes, qui dessinent un composant à leur manière, permet d'éviter ce problème. Ainsi, l'ajout d'une plate-forme ne nécessite aucune modification de code des fenêtres, et inversement pour les fenêtres. Cette *extensibilité* et ce fort *découplage* constituent les points forts de ce patron.

Ce problème a posé des difficultés aux étudiants, car nous avons recensé beaucoup de modèles erronés. Nous avons cependant pu extraire trois patrons abîmés dont deux sont fortement dégradés, cités en tant que tels dans le GoF. Les figures 2.11 et 2.12 présentent les deux solutions alternatives concernées.

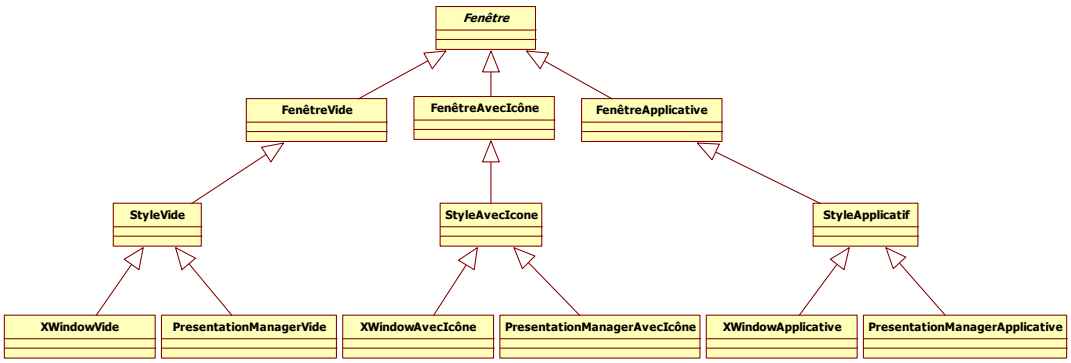


Figure 2.11 : Solution alternative 2 = Développement complet sur <<*Implémenteur*>>

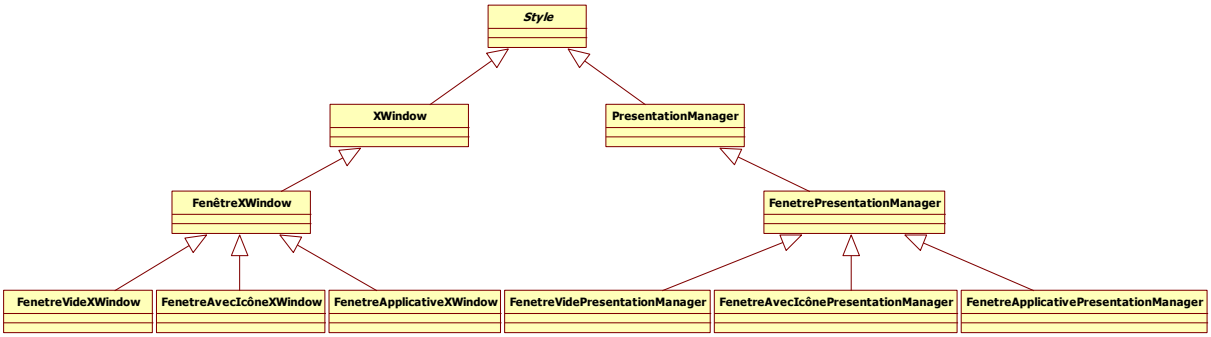


Figure 2.12 : Solution alternative 3 = Développement complet sur <<*Abstraction*>>

Ces deux solutions alternatives, que nous considérons comme identiques même si les critères de classement sont inversés, constituent les plus mauvaises solutions possible. Le GoF exploite d'ailleurs ces exemples pour montrer en quoi le patron est pertinent. Cependant, une partie non négligeable d'étudiants a eu recours à ce type de modèle pour résoudre le problème. L'utilisation exclusive de liens d'héritage ne favorise pas le découplage. Certes, l'ajout d'une nouvelle fenêtre ou d'une nouvelle plate-forme impose la création de nombreuses nouvelles classes, mais il n'y a aucune conséquence sur le code existant. Les points forts dégradés par ces solutions sont présentés dans le tableau 2.9.

1. Extensibilité	
✔	1.1. L'ajout ou la suppression d'un *Implémenteur concret* ne nécessite pas de modification de code
✔	1.2. L'ajout ou la suppression d'une *Abstraction fine* ne nécessite pas de modification de code
2. Découplage entre abstraction et implémentation	
✘	2.1. Nombre minimum de classes *Implémenteur concret*
✘	2.2. Factorisation maximale du lien entre abstraction et implémentation

Tableau 2.9 : Les points forts du patron *Pont* perturbés par les solutions alternatives 2 et 3

Ces deux solutions alternatives ne sont pas détectables structurellement, car les particularités locales et globales de chaque participant sont strictement identiques. Pour détecter ce type de fragment, il faudrait disposer d'un système capable d'estimer la profondeur d'un arbre d'héritage. Cependant, un arbre d'héritage à quatre niveaux, comme c'est le cas pour ces modèles, ne constitue en rien quelque chose d'exceptionnel.

II.2.3. Les patrons *Adaptateur*, *Façade*, *Poids-mouche* et *Procuration*

Lors de nos expérimentations sur les patrons structurels, nous avons posé des problèmes pour chacun des sept patrons. Cependant, nous n'avons pu déduire de patrons abîmés que pour *Composite*, *Décorateur* et *Pont*.

Pour les patrons *Adaptateur* et *Procuration*, nous n'avons obtenu aucune solution alternative, car soit les solutions utilisaient le patron de conception soit étaient erronées. Il semblerait que les étudiants aient utilisé le patron de conception sans le vouloir, étant donné sa simplicité structurelle.

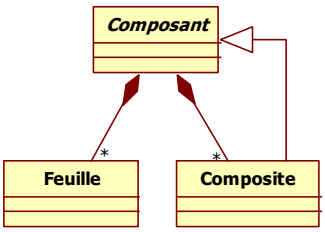
Pour le patron *Façade*, notre approche structurelle n'est pas suffisante. En effet, c'est au niveau des paquetages que la structure du *Façade* prend tout son sens. Des métriques sur les associations entrantes ou sortantes entre classes ou entre paquetages sont indispensables. En effet, ce patron est pertinent pour uniformiser et limiter la communication d'un système.

Enfin, pour le patron *Poids-mouche*, nous n'avons obtenu aucune solution valide, probablement en raison de la complexité du type de problème concerné. De plus, ce patron n'a pas vraiment pour objectif de résoudre un problème de conception. Il s'agit plutôt d'optimiser en mémoire un ensemble d'objets.

II.3. Patrons abîmés déduits

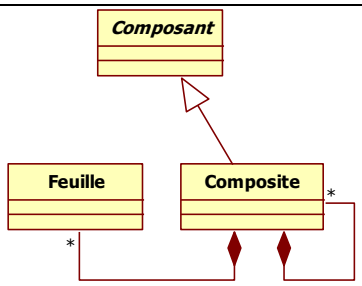
Pour chacune des solutions alternatives identifiées comme valides, nous pouvons exécuter le procédé de décontextualisation, pour en déduire les patrons abîmés associés. Pour mémoire, ce procédé consiste à marquer chaque classe de la solution alternative, avec le nom d'un des participants du patron ayant les mêmes responsabilités, puis à ne conserver, qu'une seule fois, chacun des participants rattachés de la même manière que dans la solution alternative. Nous obtenons ainsi un patron abîmé avec le même niveau d'abstraction qu'un patron de conception.

Chaque patron abîmé, présenté dans les tableaux 2.10, 2.11 et 2.12 avec la dégradation des points forts qu'il provoque, a été décontextualisé des solutions alternatives issues de nos expérimentations.



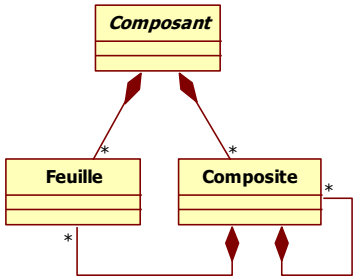
Développement de la composition sur <<Composant>>

1.1	✗	2.1	✗
1.2	✗	2.2	✓
1.3	✗	2.3	✓



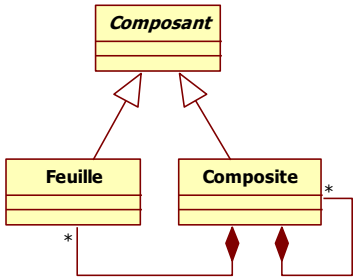
Développement de la composition sur <<Composite>> sans conformité de protocole

1.1	✗	2.1	✗
1.2	✗	2.2	✓
1.3	✗	2.3	✓



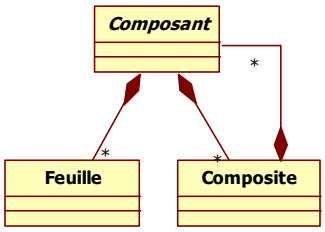
Développement de la composition sur <<Composant>> et sur <<Composite>>

1.1	✗	2.1	✗
1.2	✗	2.2	✗
1.3	✗	2.3	✓



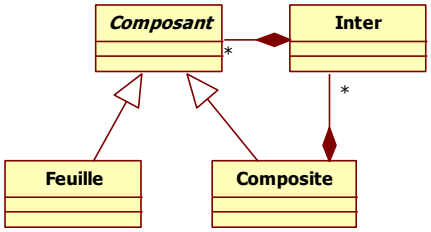
Développement de la composition sur <<Composite>>

1.1	✗	2.1	✓
1.2	✗	2.2	✓
1.3	✗	2.3	✓



Composition récursive

1.1	✗	2.1	✗
1.2	✗	2.2	✗
1.3	✗	2.3	✗



Composition indirecte sur <<Composite>>

1.1	✗	2.1	✓
1.2	✓	2.2	✗
1.3	✓	2.3	✗

Tableau 2.10 : Les patrons abîmés du *Composite*

Des sept patrons abîmés du patron *Décorateur*, seul celui nommé « *Développement maximal sur <<Composant>>* » ne pourra pas être détectable de par son uniformité structurelle.

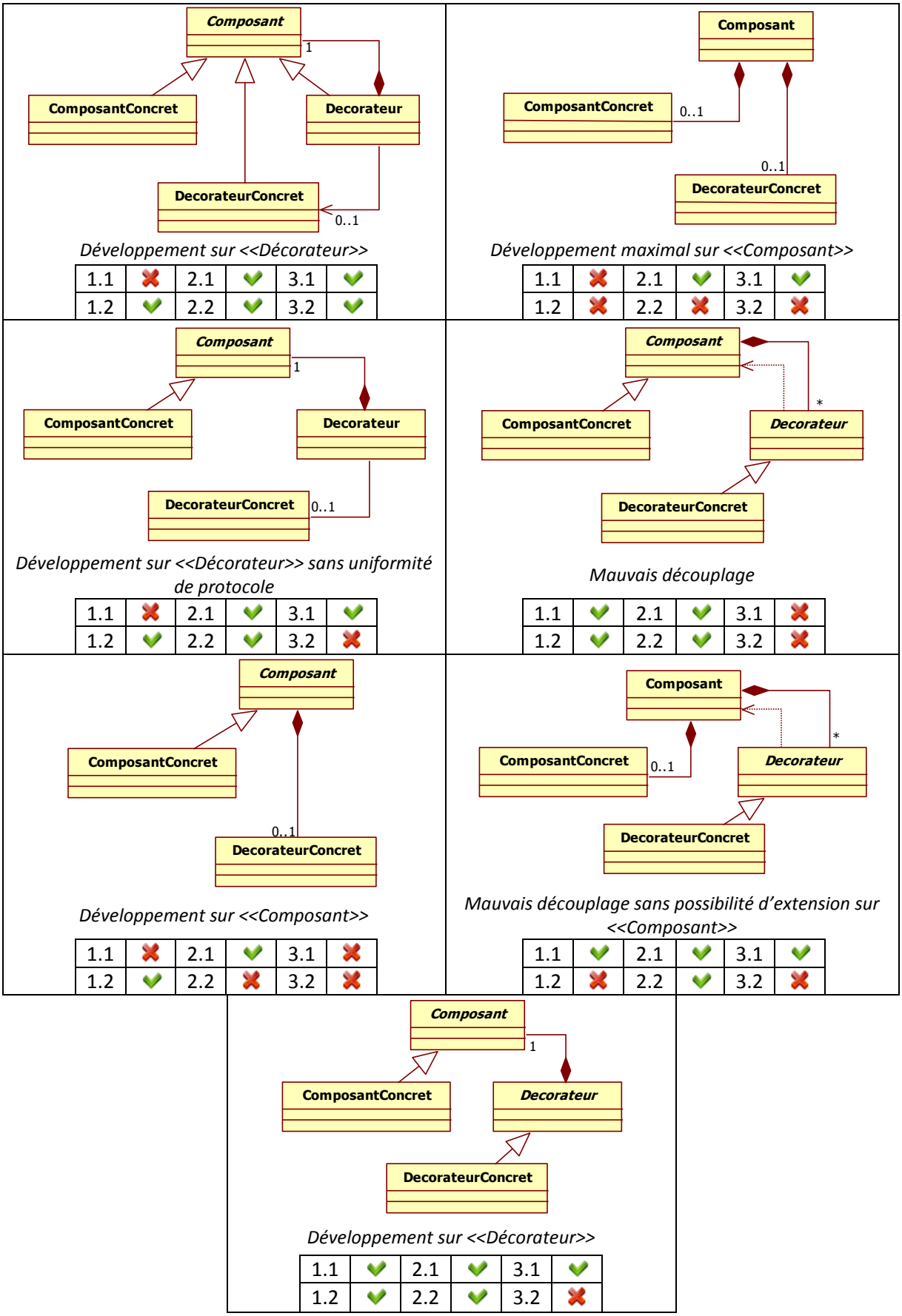


Tableau 2.11 : Les patrons abîmés du *Décorateur*

Sur les trois patrons abîmés du *Pont* déduits, seul le premier est détectable, car il est possible de différencier structurellement chaque participant.

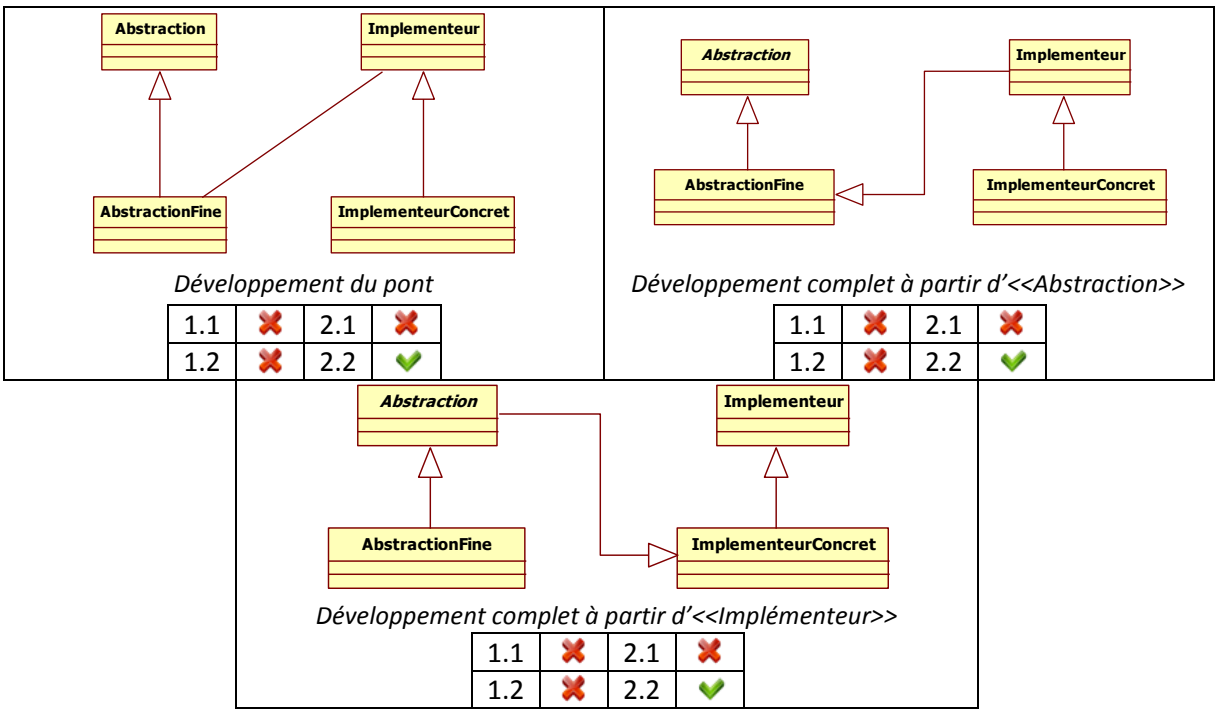


Tableau 2.12 : Les patrons abîmés du *Pont*

Finalement, nous pouvons dire que nous sommes en mesure de détecter des fragments alternatifs correspondant à des contextualisations de treize patrons abîmés structurels. À l'heure actuelle, l'analyse des solutions alternatives pour les autres patrons structurels n'a pas permis de déduire de patrons abîmés correspondants.

III. Problèmes de décontextualisation

Afin de prendre en compte le caractère dynamique des patrons comportementaux, nous avons imposé, lors de nos expérimentations, l'élaboration de diagrammes de séquence illustrant l'échange de messages entre les objets du diagramme de classes fourni. Nous souhaitions ainsi étendre nos concepts à la gestion du comportement, sans pour autant analyser la sémantique des méthodes des classes.

Dans cette section, nous nous intéressons aux décontextualisations des solutions alternatives aux patrons comportementaux et composites [Riehle97_b]. Pour les patrons comportementaux, nous montrons que plus d'une classe des solutions alternatives peut partager les responsabilités d'un même participant et que certaines solutions alternatives sont trop particulières pour identifier les responsabilités de leurs classes. Dans une deuxième partie, nous abordons le cas des patrons composites pour lesquels les solutions alternatives ne permettent pas de déduire les patrons abîmés concernés.

III.1. Les patrons comportementaux

Lors de notre analyse des expérimentations sur les patrons comportementaux, nous avons identifié cent dix solutions alternatives. D'après leurs diagrammes de classes et de séquence, chacune de ces solutions résout le problème posé. Cependant, il ne nous a pas été possible d'appliquer le procédé de décontextualisation de la même manière que pour les patrons structurels.

Nous pouvons illustrer ce phénomène grâce à un problème dont la meilleure solution consiste à contextualiser le patron de conception *Chaîne de responsabilités*. L'énoncé 2.8 présente ce problème.

Modéliser un gestionnaire d'aide d'une application Java. Un gestionnaire d'aide permet d'afficher un message d'aide en fonction de l'objet sur lequel le client a cliqué. Par exemple, le « ? » situé quelquefois à côté du menu contextuel d'une fenêtre Windows permet d'afficher l'aide en fonction du bouton ou de la zone sur laquelle on clique. Si le bouton sur lequel on clique ne contient pas d'aide, c'est la zone contenant qui affiche son aide, et ainsi de suite. Si aucun objet ne contient d'aide, au final, le gestionnaire affiche « Pas d'aide disponible pour cette zone ». Instanciez votre diagramme de classes par un diagramme de séquence sur l'exemple d'une fenêtre d'impression. Cette fenêtre (JDialog) est constituée d'un texte explicatif (JLabel) et d'un container (JPanel). Ce dernier contient un bouton Imprimer (JButton) et un bouton Annuler (JButton). Le bouton Imprimer contient l'aide « Lance l'impression du document ». Le bouton Annuler, le texte ainsi que la fenêtre ne contiennent pas d'aide. Enfin, le conteneur contient l'aide « Cliquez sur l'un des boutons ». Dans le diagramme de séquence, faites apparaître les scénarii : « L'utilisateur demande l'aide du bouton Imprimer », « L'utilisateur demande l'aide du bouton Annuler » et « L'utilisateur demande l'aide du texte ».

Énoncé 2.8 : Problème soluble par le patron *Chaîne de responsabilités*

Ce problème a pour intention de *chaîner les objets en découplant l'émetteur et le receveur*, d'*autoriser les variations de protocoles* et de *favoriser la distribution des comportements* [Kampffmeyer07]. La solution optimale est l'utilisation du patron *Chaîne de responsabilités* présentée dans les figures 2.13 et 2.14.

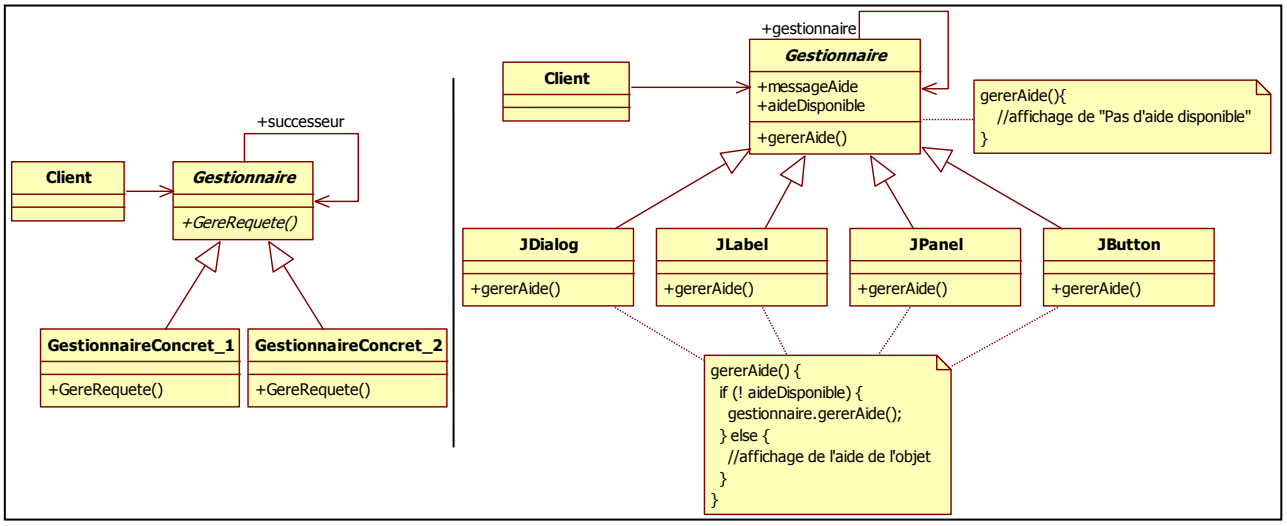


Figure 2.13 : Le patron *Chaîne de responsabilités* avec sa contextualisation pour le problème posé

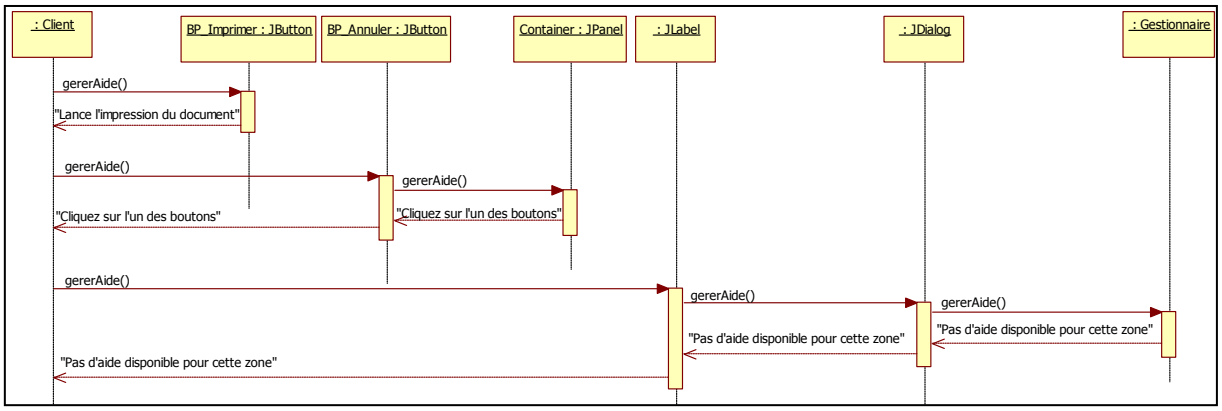


Figure 2.14 : Le diagramme de séquence de la contextualisation du patron *Chaîne de responsabilités*

Pour résoudre ce problème, il est nécessaire de mettre en place un chaînage entre les objets graphiques, en utilisant une association réflexive sur le gestionnaire d'aide, et d'en faire hériter tous les objets potentiellement receveurs de ce type de requête. Cette construction permet d'éviter le couplage de l'émetteur d'une requête à ses récepteurs, en donnant à plus d'un objet la possibilité d'entreprendre la requête. Les récepteurs sont chaînés entre eux et peuvent faire transiter la requête tout au long de la chaîne, jusqu'à ce qu'un objet la traite [Gamma95]. Ce chaînage est mis en évidence dans le diagramme de séquence par le biais des imbrications d'envoi de messages aux objets de la chaîne. Arrivé au niveau le plus bas, un message générique est envoyé si nécessaire. Ce message remonte la chaîne jusqu'à l'émetteur. Une des solutions alternatives identifiées est présentée dans la figure 2.15.

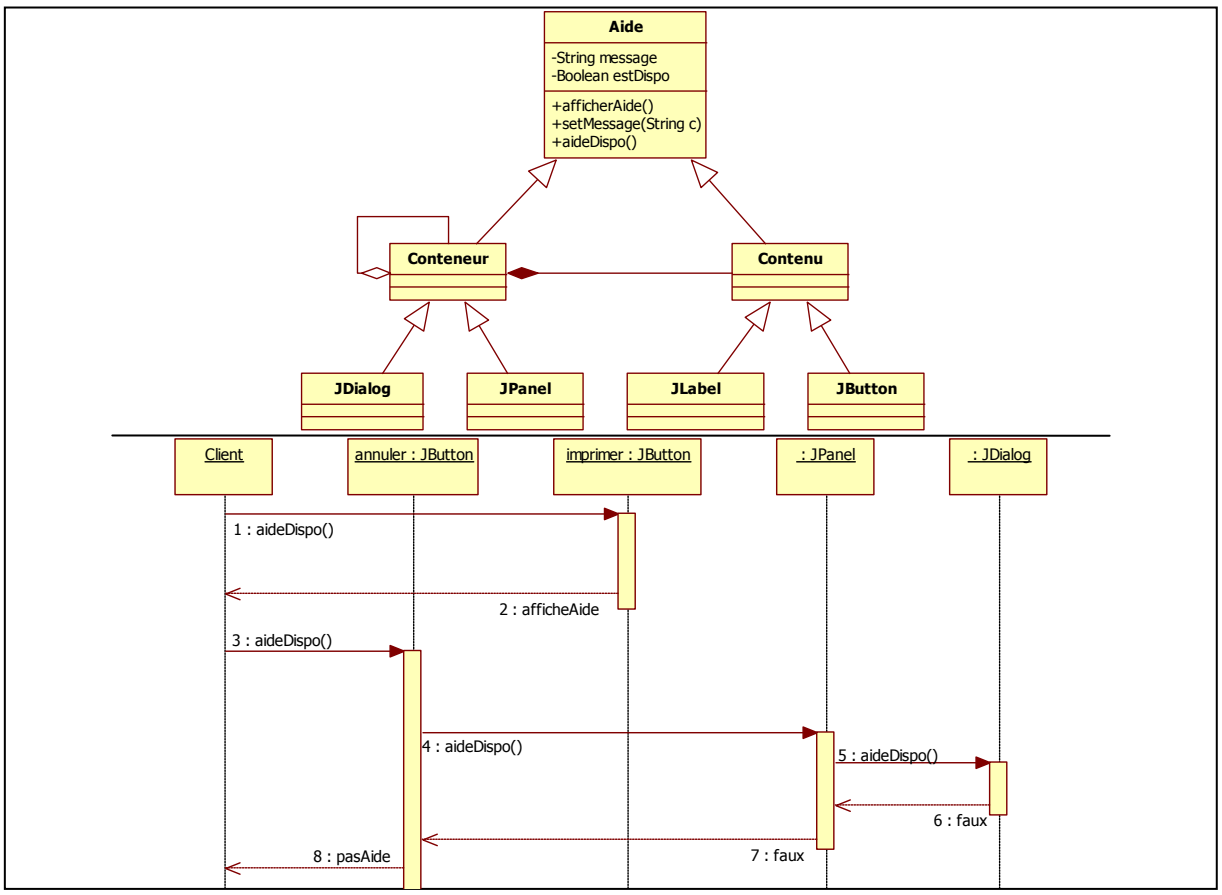


Figure 2.15 : Une solution alternative au problème de la *Chaîne de responsabilités*

Cette solution alternative respecte le chaînage des messages, comme l'illustre le diagramme de séquence. Lorsqu'une demande d'aide est activée, l'objet concerné a la possibilité soit de répondre, soit de la communiquer à un autre objet. Il est intéressant de remarquer que la structure du diagramme de classes est différente de celle du patron de conception, ce qui n'est pas le cas du diagramme de séquence.

Du point de vue du procédé de décontextualisation, il est complexe de déduire de cette solution alternative, un patron abîmé. Les classes *Aide*, *Conteneur* et *Contenu* partagent les mêmes responsabilités que le participant *Gestionnaire*, d'après l'analyse du diagramme de séquence. Concernant les échanges de messages, il n'y a pas de distinction entre *Conteneur* et *Contenu*. Il est ainsi intéressant de remarquer que si nous envisageons de décontextualiser cette solution en partageant les responsabilités, nous obtenons un patron abîmé avec trois classes pour un même participant, avec des particularités structurelles différentes. La figure 2.16 illustre le patron abîmé de la chaîne de responsabilités obtenu par notre procédé de décontextualisation actuel, si nous augmentons le nombre de participants.

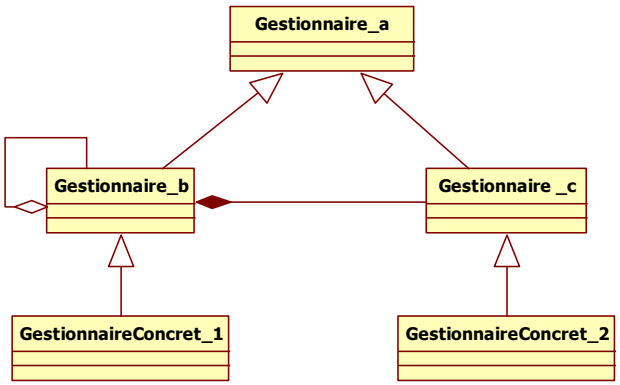


Figure 2.16 : Suggestion de patron abîmé de la *Chaîne de responsabilités*

Nous avons été également confrontés à d'autres problèmes de décontextualisation, notamment pour le patron *Médiateur*, pour lequel il ne nous a pas été possible d'identifier, dans la solution alternative, les responsabilités de chaque classe. L'énoncé 2.9 présente le problème dont la meilleure solution est la contextualisation du patron *Médiateur*.

Modéliser le fonctionnement d'une boîte de dialogue. Cette boîte de dialogue contient une liste de noms et un champ de saisie en lecture seule. Lorsque l'utilisateur clique sur un nom de la liste, il apparaît automatiquement dans le champ de saisie. Tant que l'utilisateur n'a pas cliqué sur un nom de la liste (donc tant que le champ de saisie est vide), le bouton de validation de la boîte de dialogue est désactivé. Attention, ***ne modélisez pas la boîte de dialogue, mais uniquement son fonctionnement, en y intégrant son affichage****, et veillez à ce que l'interconnexion entre les différents objets de la boîte de dialogue soit minimale.*

Énoncé 2.9 : Problème soluble par le patron *Médiateur*

Ce problème a pour intention de *masquer une complexité intrinsèque*, de *contrôler les interactions*, de *distribuer le comportement* et de *découpler les opérations* [Kampffmeyer07]. La solution optimale est l'utilisation du patron *Médiateur* présenté dans la figure 2.17.

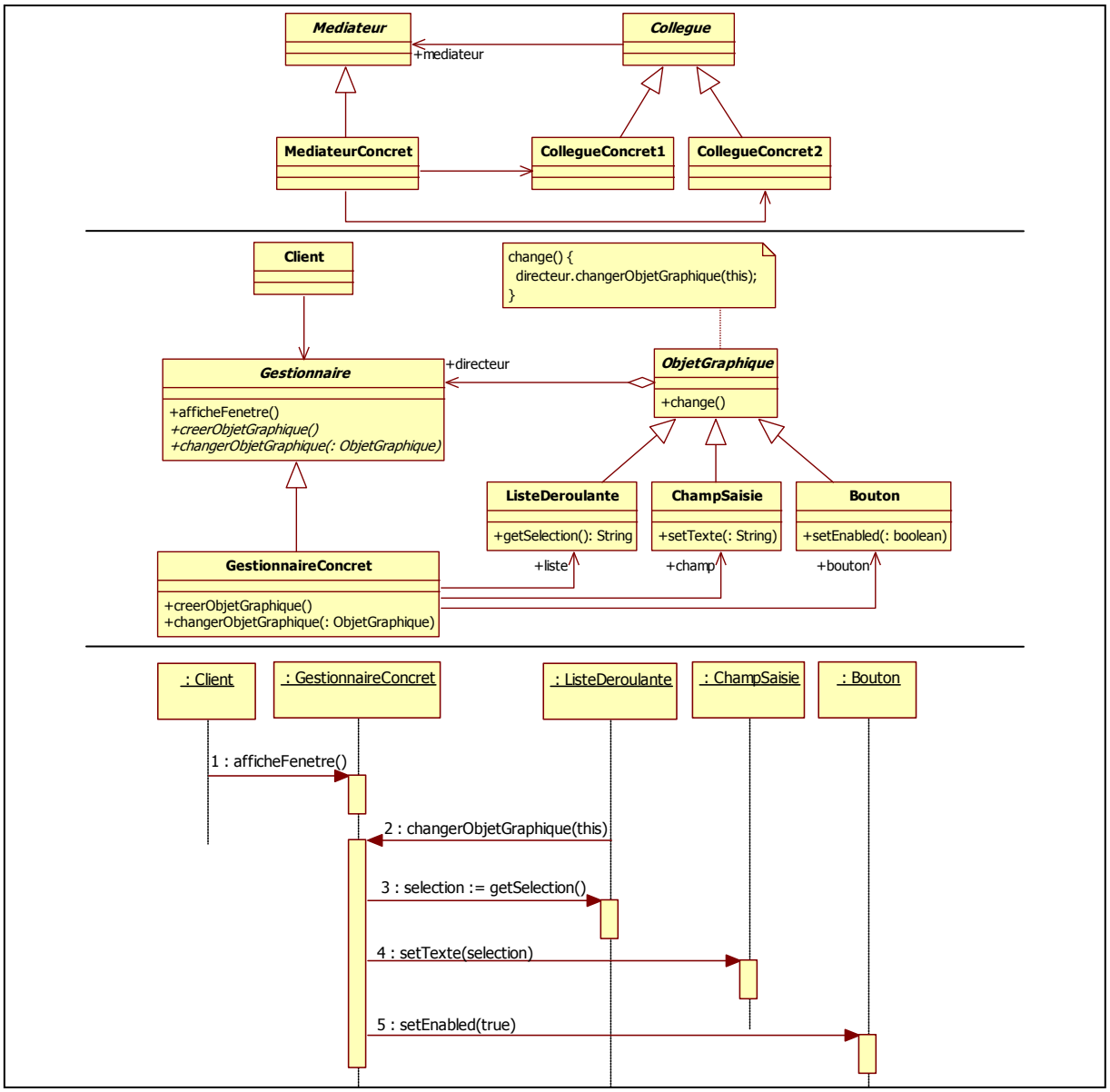


Figure 2.17 : Le patron *Médiateur* avec sa contextualisation pour le problème posé

Pour résoudre ce problème, il est nécessaire d'identifier la forte relation entre les objets pour la collaboration considérée. La solution consiste donc à centraliser les interconnexions vers un seul et même objet chargé de distribuer les messages entre les objets concernés. Cette centralisation est visible sur le diagramme de séquence où la liste déroulante prévient le gestionnaire qui se charge de transmettre le message aux autres objets de la collaboration.

Une solution alternative est présentée dans la figure 2.18.

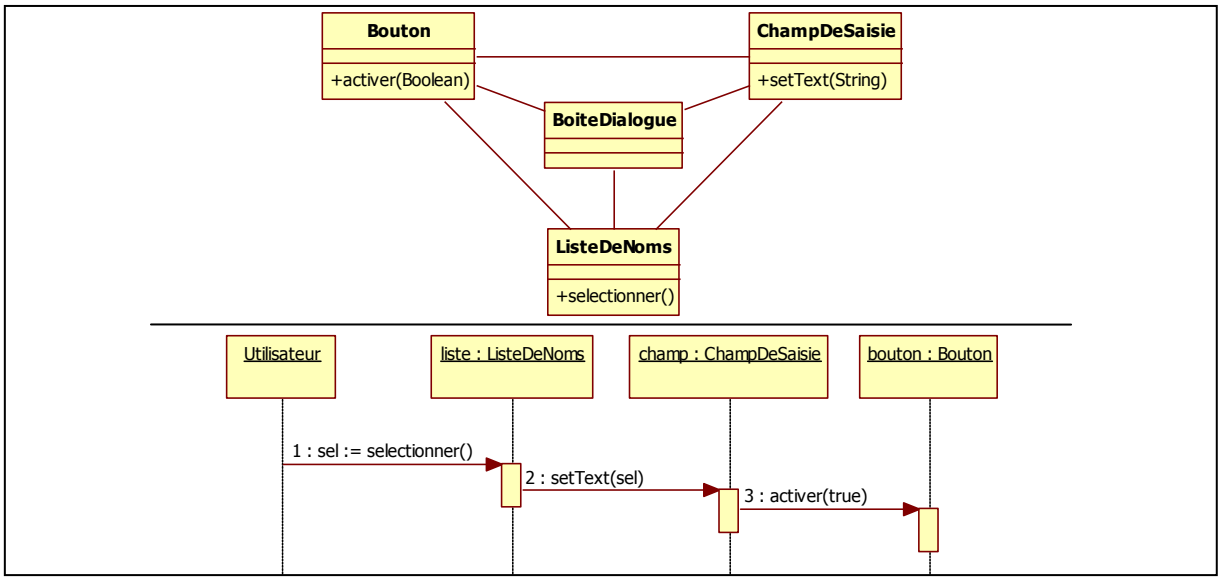


Figure 2.18 : Une solution alternative pour le problème du *Médiateur*

Cette solution ne respecte pas la faible interconnexion des objets imposés par l'énoncé, mais elle est tout de même valide. Cette solution est la plus mauvaise solution possible, et peut être exploitée pour montrer en quoi le patron est pertinent. Chaque classe a connaissance des autres classes du système et est donc à même de communiquer de manière individuelle avec elles.

À l'inverse des solutions alternatives 2 et 3 du patron *Pont*, qui constituent elles-aussi les plus mauvaises solutions pour leur patron, nous ne sommes pas en mesure, pour cette solution, d'identifier avec précision les responsabilités des classes. D'après le nom des classes, il semblerait que la classe *BoîteDialogue* puisse avoir les responsabilités du participant *Médiateur*, et c'est probablement ce qui a gêné le concepteur, puisqu'il n'a pas su l'utiliser dans le diagramme de séquence. Cette classe semble lui avoir été imposée par l'énoncé, sans qu'il ne puisse lui attribuer une fonctionnalité dans l'échange des messages.

Nous pouvons imaginer une décontextualisation de cette solution d'après le nommage des classes, mais sa faiblesse structurelle empêche sa détection avec ses particularités structurelles. Notre proposition de décontextualisation est présentée dans la figure 2.19.

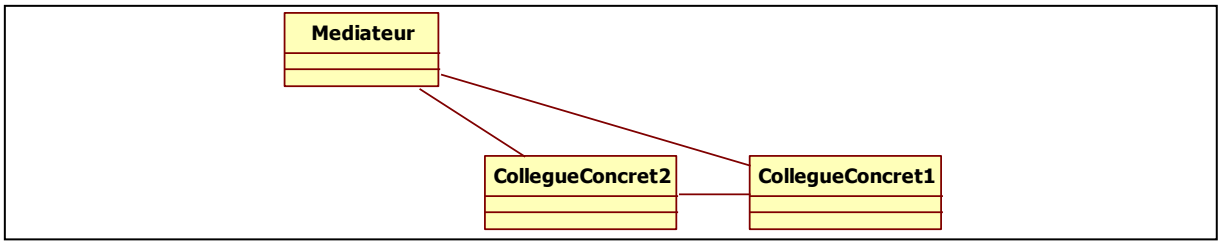


Figure 2.19 : Suggestion de patron abîmé du *Médiateur*

Il est intéressant de remarquer que les expérimentations sur les patrons comportementaux ont mis en avant la non-complétude de notre procédé de décontextualisation conçu à partir des patrons structurels. Bien que le diagramme de séquence nous permette d'analyser plus finement les solutions, l'adaptation du procédé de décontextualisation s'avère nécessaire à l'ajout de patrons abîmés comportementaux dans notre catalogue.

III.2. Les patrons composites

Pour certains contextes, l'utilisation conjointe de plusieurs patrons est nécessaire, complexifiant d'autant notre tâche. En effet, dans ces cas-là, il s'agit de décontextualiser des solutions alternatives faisant intervenir plusieurs patrons de conception de manière plus ou moins dégradés.

Prenons le cas d'un énoncé ayant pour but initial de trouver des solutions alternatives au patron de conception *Visiteur*, dont une contextualisation est la meilleure solution pour le problème de l'énoncé 2.10.

Modéliser l'affichage à l'écran d'une calculatrice. La calculatrice permet la saisie d'expressions, qui peuvent être « des nombres » ou « des opérations binaires » (addition, soustraction et multiplication). La calculatrice propose deux modes d'affichage : « infixé » (i.e. 3_+_3) ou « préfixé » (i.e. +_3_3). Après avoir saisi ses expressions, l'utilisateur peut choisir le mode d'affichage, et passer de l'un à l'autre autant de fois qu'il le désire. Il est probable que le système évolue pour que le mode d'affichage « suffixé » (i.e. 3_3_+) soit ajouté.

Énoncé 2.10 : Problème soluble pas le patron *Visiteur*

Ce problème est soluble avec le patron *Visiteur*. L'intention de ce patron est de *distribuer le comportement*, de *découpler les opérations tout en leur laissant une variabilité*, de permettre l'*accumulation des états* des objets et d'imposer une *dépendance algorithmique* [Kampffmeyer07]. La solution optimale au problème posé est présentée dans les figures 2.20 et 2.21.

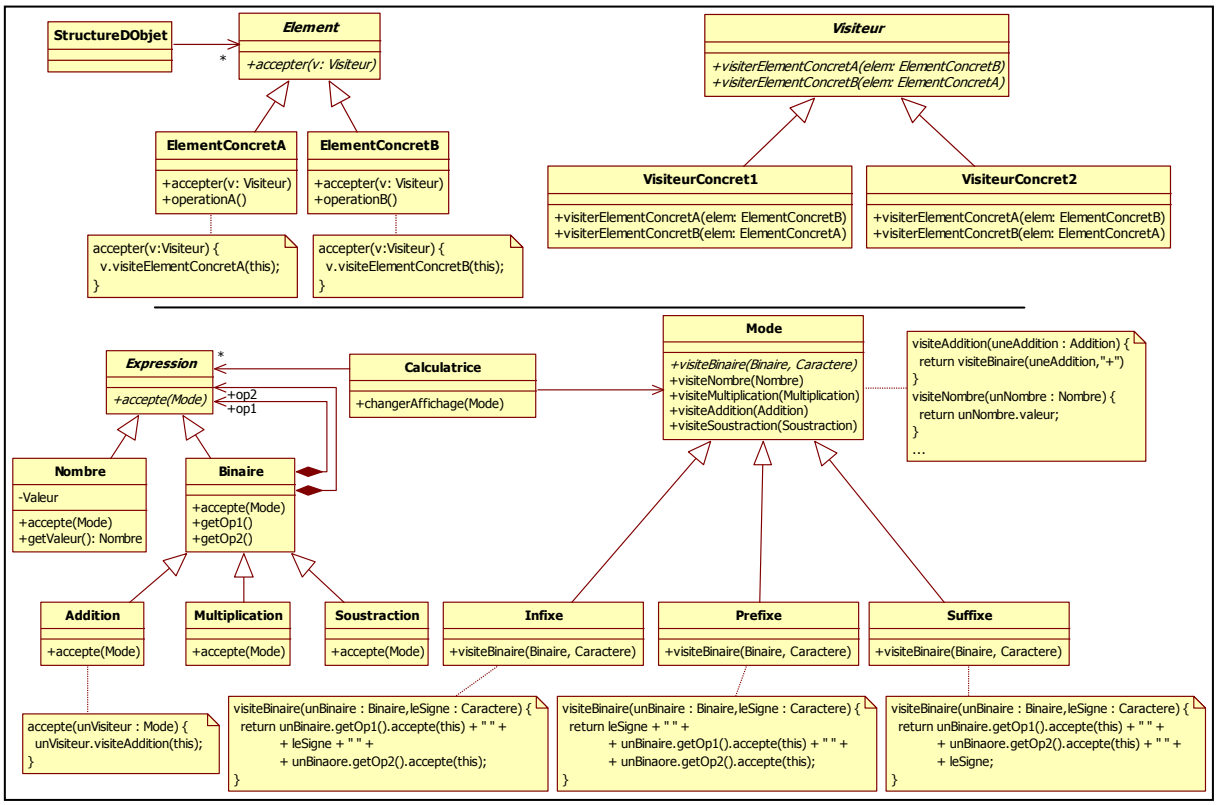


Figure 2.20 : Le patron *Visiteur* et sa contextualisation sur le problème posé

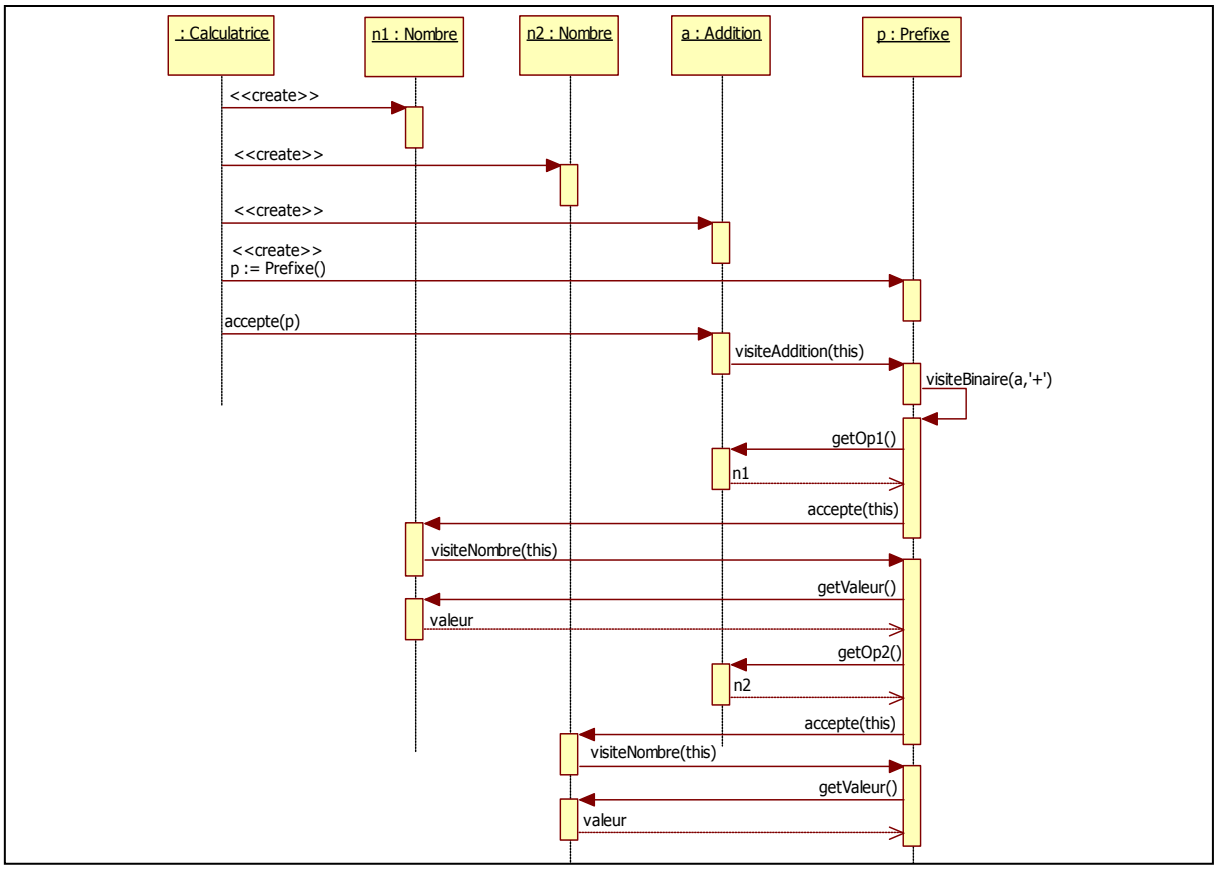


Figure 2.21 : Le diagramme de séquence de la contextualisation du patron *Visiteur*

Avec le patron *Visiteur*, les classes gérant les modes d'affichage sont séparées des classes des opérations de la calculatrice. Chaque mode d'affichage est encapsulé dans une sous-classe de *Mode* distincte d'une sous-classe d'*Expression* chargée d'un calcul spécifique. Cela permet au client de changer de stratégie d'affichage facilement et de simplifier les responsabilités des classes gérant les expressions. Le diagramme de séquence illustre cette alternance entre les classes gérant le mode d'affichage et les classes gérant les expressions. Il est intéressant de remarquer que la meilleure solution à ce problème s'appuie sur la structure d'un patron *Interprète*, et utilise un *Patron de Méthode* pour gérer le code commun aux différents types de visite. Alors que l'*Interprète* est fortement lié à l'évaluation d'expressions de la calculatrice, et donc propre à ce problème, le *Patron de méthode* semble plus difficilement dissociable du *Visiteur*, d'autant qu'il est explicitement couplé au *Visiteur* dans le GoF. Nous pouvons donc nous demander lequel de ces trois patrons sera abîmé dans les solutions alternatives.

La figure 2.22 illustre une solution alternative.

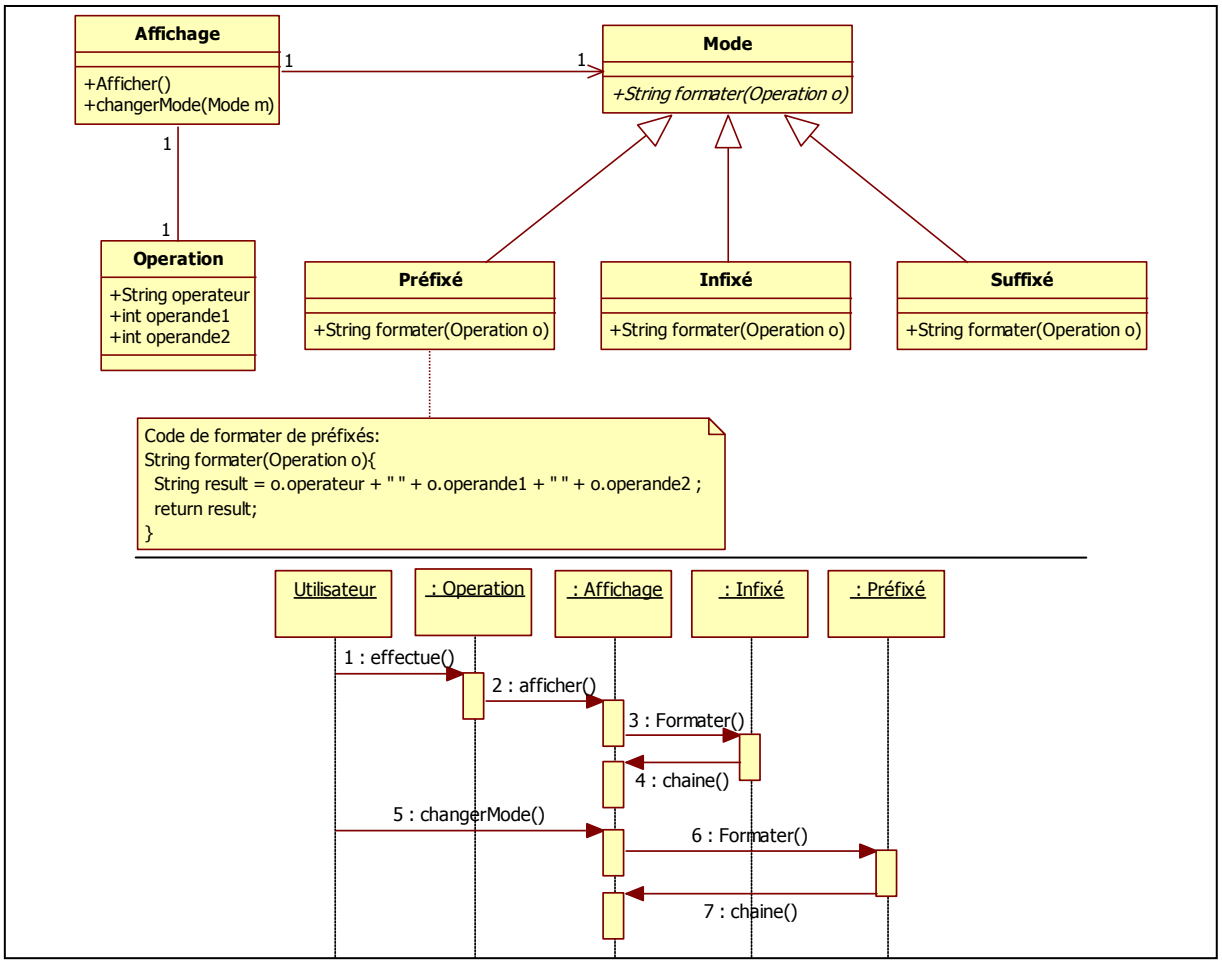


Figure 2.22 : Une solution alternative au problème du *Visiteur*

Nous retrouvons dans cette solution alternative l'utilisation du *Patron de méthode*. Cependant, ni le *Visiteur*, ni l'*Interprète* ne sont présents. De plus, cette solution alternative est particulièrement intéressante car le découplage entre les modes d'affichage et les opérations est respecté, mais le diagramme de séquence l'illustre différemment du patron de conception. Un objet de type *Operation* est transmis en paramètre aux méthodes de la classe *Mode* pour que l'affichage soit spécialisé en fonction du mode. Il en résulte que les protocoles de la classe *Operation* doivent être connus des classes *Mode*. Si un nouveau type d'opération est ajouté, avec par exemple un seul opérande, il est nécessaire de s'assurer de la conformité de ce nouveau protocole. Sans le diagramme de séquence, il aurait été difficile d'entrevoir le fonctionnement de cette solution.

La décontextualisation de cette solution devient problématique dès lors que l'on considère les trois patrons en présence. Il est difficile de considérer que la classe *Operation* détient à elle-seule l'intégralité des responsabilités de chaque participant du patron *Interprète*. De plus, nous ne pouvons pas associer le participant *Visiteur* à la classe *Mode* qui n'a pas réellement l'intention de visiter les autres classes. Dans ce cas de figure, un participant du patron abîmé peut avoir plusieurs responsabilités mais chacun d'entre eux fait référence à un participant d'un des patrons du composite. Notre procédé de décontextualisation est donc là aussi trop rigide. Il semblerait qu'une solution soit de considérer chaque appariement composite et de le traiter comme un patron de conception ayant l'union des participants de chacun des patrons de la composition.

IV. Conclusion

Nous avons présenté dans ce chapitre la méthode et les résultats de notre collecte de solutions alternatives à des problèmes de conception, nous permettant de déduire un ensemble de patrons abîmés. Elle nous garantit que tous les patrons abîmés obtenus ont été déduits de solutions valides dont les défauts de conception ont été identifiés. Cette collecte s'est effectuée directement auprès de concepteurs n'ayant aucune expérience sur l'utilisation des patrons de conception. Ils ont résolu les problèmes posés selon leur propre expérience, et dans la majorité des cas, en utilisant une autre manière de faire qu'avec un patron de conception. Grâce à cette collecte, nous détenons maintenant un catalogue de patrons abîmés au même titre que le GoF pour les patrons de conception. Nous réunissons ainsi un ensemble de mauvaises pratiques de conception pouvant compléter le GoF.

Afin de classifier chaque patron abîmé, nous avons mesuré leur degré de dégradation en utilisant les points forts des patrons de conception. De plus, cette classification par défauts de conception nous apporte une base explicative de l'intérêt d'utiliser des patrons de conception. En présentant au concepteur les points forts dégradés par les patrons abîmés, nous justifions en quoi leur remplacement par un patron de conception est judicieux.

De manière à envisager une collecte à plus grande échelle, nous avons entrepris la création d'un site web collaboratif. Ce dernier est destiné à créer une communauté, consacrée à l'identification à plus grande échelle de mauvaises pratiques de conception, qui sera à même de collecter et d'analyser les solutions alternatives. Nous pourrons alors viser un consensus sur les patrons abîmés au même titre que pour les patrons de conception.

3

Détection et transformation de fragments alternatifs dans un modèle

L'objectif de nos travaux est de vérifier, dans un modèle, la présence de défauts de conception caractéristiques, et le cas échéant, de les corriger en utilisant des patrons de conception. Pour ce faire, nous travaillons avec une base de patrons abîmés qui représentent des mauvaises solutions pour résoudre un problème, par opposition aux patrons de conception. L'idée est donc de détecter dans un modèle les patrons abîmés, sous la forme de fragments du modèle où le concepteur aurait pu mieux faire s'il avait contextualisé un patron de conception. En procédant au remplacement de tous les fragments caractéristiques de mauvaises pratiques de conception, nous permettons l'utilisation des points forts des patrons de conception. Nous encourageons ainsi la réutilisation du savoir-faire des experts du domaine pour certains problèmes de conception [Bouhours06_a].

De nombreux travaux existent visant à détecter des fragments significatifs dans un modèle. Dans la majeure partie des cas, ces travaux recherchent les patrons de conception, dans le but d'aider à la redocumentation ou à la maintenance des modèles. Ces travaux se basent sur des techniques de détection visant à identifier l'utilisation des patrons de conception, quel que soit le contexte du problème, même si leur forme est incomplète. Bien que nous ne recherchions pas des patrons de conception, mais des patrons abîmés, les techniques de détection de patrons de conception peuvent s'apparenter à notre problématique, puisque cela revient à identifier des zones caractéristiques dans un modèle. Cependant, nous avons choisi de ne pas utiliser les méthodes existantes, car nous souhaitons que notre détection puisse être réglable par le concepteur et qu'elle fonctionne en n'utilisant que des éléments de niveau modèle, sans aucune information issue du code.

Pour ce chapitre, dans une première section, nous nous intéressons à certaines techniques de détection de patrons de conception à des fins de redocumentation et d'identification de problèmes de conception solubles par l'utilisation de patrons de conception. Nous effectuons ainsi un tour d'horizon des problématiques inhérentes à la détection de fragments particuliers dans les modèles, en les comparant aux spécificités de notre approche. En deuxième section, nous présentons la technique de détection que nous avons conçue, fondée sur une identification par concordance de particularités structurelles. Elle fonctionne grâce à un procédé de filtrages successifs, incluant des informations spécifiques telles que les relations interdites entre les classes. Dans une dernière section, nous validons l'approche en présentant les résultats des tests unitaires et des tests aux limites effectués. Nous montrons ainsi que notre approche est opérationnelle et qu'elle est capable de détecter des fragments alternatifs remplaçables par des patrons de conception.

I. Techniques de détection de fragments

Identifier dans un modèle des fragments particuliers peut avoir plusieurs objectifs. Divers travaux visent à identifier des fragments représentant des contextualisations de patrons de conception correctes, incorrectes ou incomplètes, afin d'aider à la compréhension de conceptions existantes et de fournir une base pour d'éventuelles améliorations [Tsantalis06]. En ce qui nous concerne, nous souhaitons rechercher des fragments correspondant structurellement à des contextualisations de patrons abîmés, afin de les remplacer par des patrons de conception. Cependant, quel que soit le but de la détection et la sémantique donnée aux fragments identifiés, un nombre non négligeable de problèmes se pose.

Tout d'abord, pour identifier des fragments caractéristiques, il est nécessaire de parcourir le modèle, en utilisant des techniques souvent empruntées à la théorie des graphes. De plus, il est nécessaire de veiller à ce que le temps d'exécution et la complexité de l'algorithme soient adaptés à des modèles conséquents. D'autre part, il n'est pas possible d'exécuter un algorithme de recherche sans connaître à l'avance la forme du fragment recherché, ou du moins à quoi il doit ressembler. Cette approximation de la forme recherchée est très problématique, car elle introduit des incertitudes dans la recherche. Dans le cas des patrons de conception, le concepteur ayant adapté le patron à son problème, il est nécessaire que les méthodes de détection soient capables de n'omettre aucune forme [Wenzel05_a].

Ainsi, nous avons cherché à établir une méthode de détection utilisable sans qu'il soit nécessaire d'effectuer un prétraitement sur le modèle à analyser et sans utiliser d'informations issues du code. De plus, la détection doit s'effectuer par étapes successives afin de réduire au plus tôt le nombre d'éléments à analyser, et donc minimiser son temps d'exécution. Le nombre de fragments détectés doit être limité sans pour autant négliger de contextualisations possibles, et le consensus du patron doit être respecté.

Dans cette section, nous présentons des techniques existantes de détection de fragments dans des modèles. Nous montrons tout d'abord qu'il est possible de considérer, en l'état, un modèle UML sous forme d'un graphe, ce qui résume la problématique de recherche de fragments, à un problème de recherche de sous-graphes. En deuxième partie, nous décrivons une adaptation de l'approche de pattern matching approchée à des mécanismes de comparaison de similarités afin d'identifier des patrons de conception pour redocumenter des modèles. Nous présentons ensuite un outil qui concrétise la détection de patrons de conception en utilisant des mécanismes d'évaluation floue. Dans une dernière partie, nous détaillons une approche consistant à métamodéliser les problèmes solubles par les patrons de conception. Ainsi, lorsque le concepteur rencontre, dans son modèle, des fragments conformes à ces métamodèles, il est capable d'identifier quel patron résout au mieux son problème.

I.1. Modèle UML et graphes

Dans le contexte de nos travaux, modèle, patron de conception et patron abîmé sont, tous trois, décrits par la notation UML 1.5 [OMG03]. Cette dernière nous a été imposée par la plate-forme que nous utilisons pour exécuter les requêtes de détection. Dans le but de permettre l'échange de modèles et la collaboration avec des outils de modélisation, le format XMI [OMG07] peut être utilisé étant donné que c'est un format normalisé. Cependant, même si chaque outil de modélisation utilise à sa manière ce format, créant des problèmes d'interopérabilité, nous l'avons tout de même choisi, afin de regrouper sous un même format modèle à analyser, patron de conception et patron abîmé. Le format XMI permet une représentation XML [Bray06] des différents diagrammes UML existants, chaque balise XML reprenant un élément du métamodèle utilisé par le modèle UML.

Ce format étant fondé directement sur le métamodèle, nous considérons qu'un modèle, un patron de conception et un patron abîmé sont exprimés en fonction des métaclasses qu'ils utilisent. Ainsi, de par la forme et les caractéristiques du métamodèle, nous considérons qu'il est possible de représenter les modèles par des graphes orientés. Ces graphes UML préservent la structure des modèles, car ils sont composés de sommets typés représentant à la fois les classes et les relations entre les classes. Les arcs, quant à eux, servent à indiquer le sens de la relation. Dans notre cas, comme nous ne nous intéressons qu'aux classes, qu'aux associations et qu'aux généralisations, nous avons une représentation centrée sur la structure de nos modèles et de nos patrons. La figure 3.1 est un exemple de représentation d'un patron abîmé sous forme de graphe.

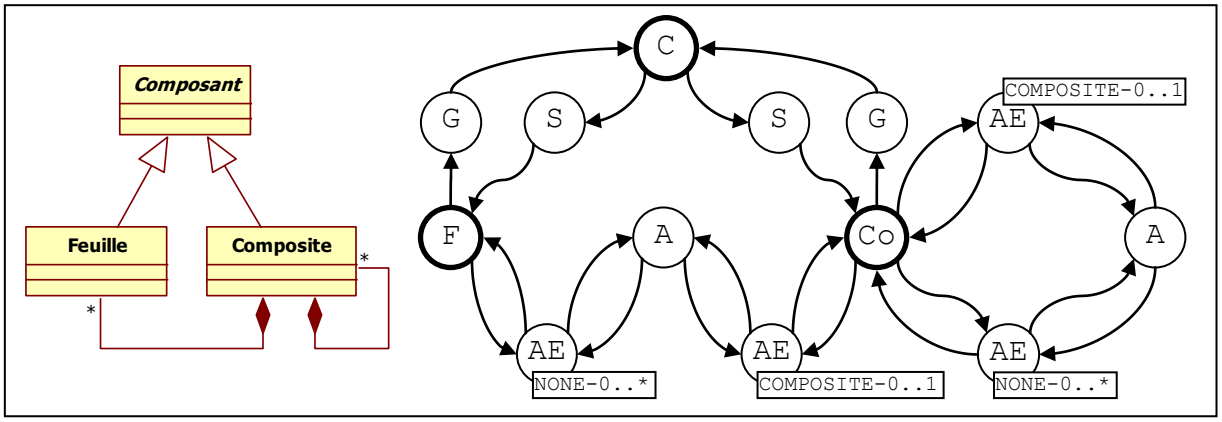


Figure 3.1 : Un patron abîmé et sa représentation sous forme de graphe

Dans cet exemple, le patron abîmé est un graphe orienté simple, avec les sommets `C`, `F` et `Co`, respectivement *Composant*, *Feuille* et *Composite*, puis les ensembles de sommets {`A`}, {`AE`}, {`G`} et {`S`}, respectivement *Association*, *AssociationEnd*, *Generalization* et *Specialization* du métamodèle UML.

Comme nous considérons nos modèles UML comme des graphes, nous pouvons reformuler notre problématique de recherche de contextualisation de patrons abîmés dans un modèle à un problème d'identification de sous-graphes dans un graphe. Il existe deux principales approches de recherche dans des graphes. La première, le pattern matching

exact, consiste à retrouver exactement un sous-graphe donné [Ullmann76], la seconde, le pattern matching approché, consiste à identifier les sous-graphes ressemblant plus ou moins à un graphe donné [Bengoetxea02].

I.1.1. Pattern matching exact

Les problèmes de pattern matching exact consistent à retrouver dans un graphe les sous-graphes identiques à un graphe modèle. Pour ce faire, il existe des algorithmes permettant d'identifier dans un graphe tous les sous-graphes isomorphes à un graphe donné. Par exemple, considérons un graphe A dans lequel la recherche doit être effectuée, et un graphe B représentant le graphe recherché. Ces algorithmes comparent le graphe B à toutes les combinaisons de sous-graphes possibles de A, jusqu'à trouver une correspondance parfaite entre la combinaison trouvée et le graphe B. Ainsi, l'application de tels algorithmes requiert l'examen de tous les sous-graphes possibles qui ont le même nombre de sommets et d'arrêtes que le graphe recherché, ainsi que la même configuration, ce qui rend ce problème NP-complet [Ullmann76].

Pour notre problématique, nous pouvons dire que nous ne nous inscrivons pas dans les problèmes de pattern matching exact. En effet, il est important de rappeler qu'un patron abîmé est une base génératrice d'une famille de contextualisations possibles. Ainsi, nous ne recherchons pas exactement le patron abîmé, mais une de ses contextualisations, dont on ne connaît pas la forme par avance. Le pattern matching exact de graphes ne concerne donc pas notre problème d'autant que nous ne pouvons pas envisager de rechercher toutes les formes possibles de contextualisations dans un graphe.

I.1.2. Pattern matching approché

Cette approche, concernant l'identification de sous-graphes ressemblant au graphe recherché, est très utile lorsqu'un isomorphisme entre deux graphes n'a pas pu être trouvé. Les algorithmes de pattern matching approché permettent de trouver la meilleure correspondance entre deux graphes non isomorphes. Par exemple, certains algorithmes calculent la distance entre deux graphes, cette distance étant souvent exprimée en nombre de modifications qu'il est nécessaire d'effectuer pour transformer un graphe vers le graphe comparé [Tsantalis06]. Ainsi, si un sous-graphe ne nécessite que deux opérations de transformation pour être isomorphe au graphe recherché, il est sélectionné prioritairement à un sous-graphe dont le nombre de modifications serait plus élevé. Vis-à-vis d'un contexte de détection de patrons de conception, ces algorithmes reviennent à rechercher les fragments de modèle ressemblant au patron recherché.

Dans notre contexte de recherche de patrons abîmés, ce type d'algorithme est plus intéressant, car il nous permet de détecter les fragments structurellement proches du patron abîmé recherché. Cependant, cela n'est pas suffisant, car nous ne recherchons pas les fragments les plus ressemblants au patron abîmé, mais les fragments ressemblant exactement à une contextualisation d'un patron abîmé. Un patron abîmé est défini de la même manière qu'un patron de conception, avec, en supplément, un ensemble de points

d'extension et de liens interdits. Les points d'extension nous permettent de définir à l'avance toutes les formes possibles, en indiquant par exemple quels sommets du graphe peuvent se répéter dans la contextualisation. Les liens interdits nous permettent également de préciser si des formes sont incompatibles avec le patron abîmé.

Nous présentons maintenant trois techniques de détection utilisées dans un contexte proche de notre problématique. Les deux premières concernent l'identification de patrons de conception à des fins de redocumentation, et la troisième l'identification de problèmes de conception solubles par l'utilisation de patrons de conception.

I.2. Identification par comparaison de similarités

Il est possible, moyennant un traitement particulier, de représenter les modèles UML et par extension les patrons de conception, sous forme de matrices carrées, notamment grâce aux relations binaires entre les classes [Tsantalis06]. Dans un contexte d'identification de patrons de conception, il est possible d'adapter les algorithmes de pattern matching approché pour définir un algorithme de comparaison de similarités, utilisant les matrices pour calculer efficacement les approximations [Tsantalis06].

I.2.1. Modèle UML et représentation matricielle

Une autre manière de représenter un modèle UML est de considérer en l'état sa forme visuelle comme un multigraphe. À la différence d'une représentation XMI, chaque classe du modèle est représentée par un sommet et les relations entre classes par des arcs. Chaque type de relation est alors représenté par un graphe différent. Afin de manipuler de manière plus aisée chacun de ces graphes, il est possible de les mapper dans des matrices carrées. Grâce aux règles de manipulation des matrices, cette représentation des modèles apporte un formalisme permettant de construire un système de détection efficace. Le patron *Décorateur*, illustré dans la figure 3.2 en UML, est présenté dans la figure 3.3 sous forme de matrices.

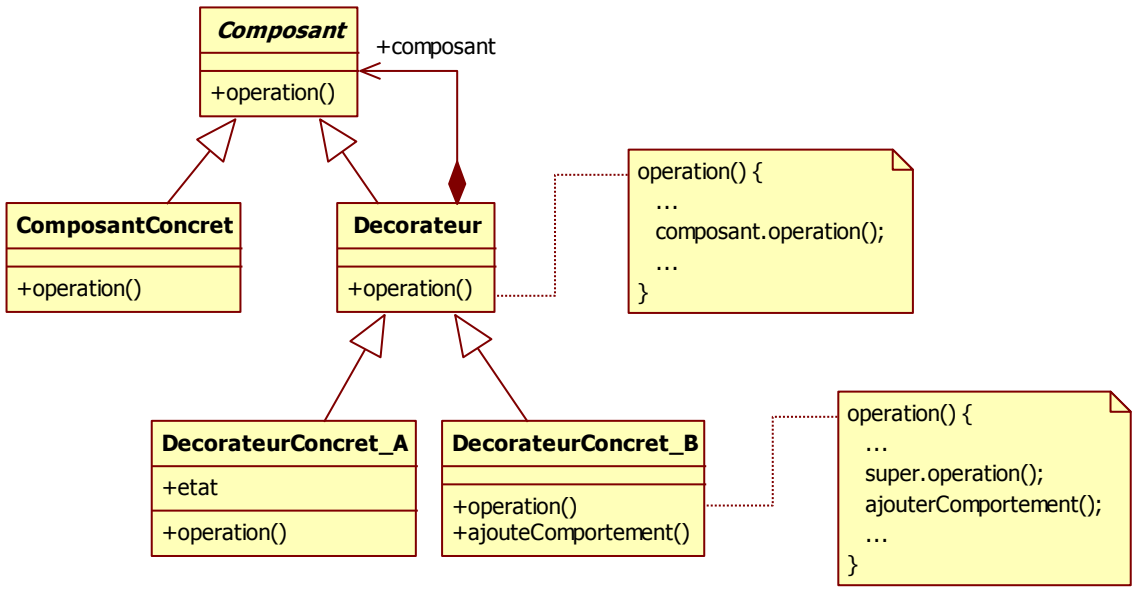


Figure 3.2 : Le patron *Décorateur*

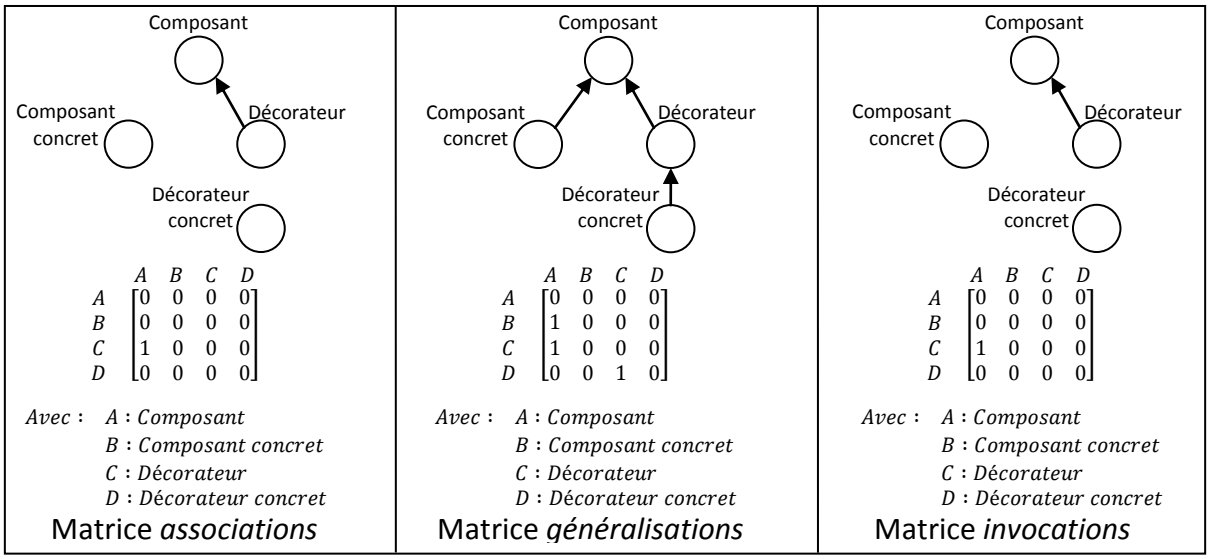


Figure 3.3 : Le patron *Décorateur* sous forme de graphes et de matrices [Tsantalis06]

Chaque matrice représente une caractéristique du patron. Par exemple, dans la matrice *associations*, la valeur 1 située en C-A indique qu'il y a un arc de C vers A et donc qu'une association relie C (*Décorateur*) et A (*Composant*). De même pour la matrice *généralisations* qui indique que B (*Composant concret*) et C (*Décorateur*) héritent de A (*Composant*).

En exploitant ce mode de représentation, un concepteur désireux de rechercher la présence d'un patron de conception a la possibilité de spécialiser lui-même quelles informations il recherche. En effet, il peut n'utiliser que les matrices représentant la structure du patron, ou y adjoindre des matrices comportementales, comme c'est le cas de la matrice *invocations* présentée dans la figure 3.3 qui traduit l'invocation d'une méthode (*opération*) par une méthode similaire (avec la même signature, ici, *opération* dans la classe *Composant*).

I.2.2. Comparaison des similarités de graphes

Cette méthode de détection se caractérise par un algorithme de comparaison de similarités. Inspiré des travaux de [Blondel04] qui a proposé un algorithme pour calculer les similarités entre les sommets de deux graphes différents, cet algorithme génère, à partir des matrices représentant les patrons et les modèles, une matrice d'adjacence représentant la proximité des deux graphes. Grâce à cette technique, il est possible de détecter des patrons implémentés différemment de la manière préconisée, avec de meilleurs résultats qu'avec les approches de pattern matching approché [Tsantalis06]. La matrice générée par l'algorithme exprime la proximité de chacune des valeurs par un rapport à un. Plus le résultat est proche de un, plus les graphes comparés sont proches selon la matrice concernée. Ainsi, pour un même graphe, il suffit d'établir la moyenne pondérée des matrices résultats pour chacune des matrices significatives en entrée (associations, généralisations, méthodes…). La pondération est fixée par le concepteur en fonction de l'importance qu'il souhaite donner à une matrice ou à une autre. Le résultat de cette moyenne représente donc la « ressemblance » d'un fragment par rapport à un patron de conception [Tsantalis06].

I.2.3. Mise en relation avec notre problématique

Le fait qu'il soit nécessaire d'effectuer un prétraitement pour convertir le modèle en matrices, d'autant que le modèle est issu d'une rétroconception de code, constitue la limite principale de cette approche. Ce prétraitement peut prendre un temps conséquent lorsque les programmes sont volumineux.

Par-delà le fait que nous ne pouvons renseigner que des matrices représentant la structure du modèle, puisque nous souhaitons proposer une méthode orientée modèle sans information issue du code, il nous faut pouvoir modifier la conception sans avoir à effectuer le prétraitement après chaque transformation. En effet, chaque fragment que nous détectons peut être potentiellement remplacé par la contextualisation d'un patron de conception. Après chaque restructuration, il nous est nécessaire d'exécuter à nouveau la détection afin de prendre en compte les modifications issues des transformations. Cela nous permet ainsi de vérifier s'il ne reste pas d'autres fragments ou si la substitution n'a pas provoqué l'apparition d'un autre fragment alternatif par effet de bord sur le modèle. Sur une base de patrons abîmés conséquente, sachant que plusieurs fragments peuvent être identifiés pour un même patron abîmé, le nombre de réexécutions du prétraitement serait beaucoup trop conséquent pour être envisagé dans un processus de conception que nous voulons interactif.

I.3. Détection par évaluation floue de modèles UML

Afin d'offrir aux concepteurs et aux développeurs un outil capable de générer automatiquement du code Java à partir d'un modèle UML et d'impacter les modifications du code sur ce même modèle, l'application Fujaba (From UML to Java And Back Again) a été développée [FUJABA05]. En supplément de cet outil, un composant de détection automatique de patrons de conception a été ajouté. Compatible avec la rétroconception de code, ce composant utilise l'intégralité des données des modèles pour effectuer la détection, en considérant les modèles sous forme de graphes de syntaxe abstraite (Abstract Syntax Graph : ASG) [Niere02]. Ce mode de représentation permet de décrire formellement un modèle en éliminant la plupart des variantes syntaxiques et des problèmes de mise en forme.

Les patrons de conception recherchés sont décrits sous la forme d'un ensemble de sous-patrons, eux-mêmes considérés comme des règles de transformation de graphe. Un sous-patron représente un élément fondamental à la constitution de tout modèle, et peut être comparé à un élément du métamodèle. Ainsi, un patron de conception est décrit par un ensemble de règles de transformation permettant d'annoter des fragments de l'ASG. La figure 3.4, représente un extrait d'un ASG qui a été annoté par l'exécution de la règle de transformation du sous-patron *Generalization*. Il est possible de dire que le sous-patron *Generalization* a été identifié dans cet ASG. Les trois sommets et deux arcs sur le côté droit de la figure représentent l'extrait de l'ASG montrant l'héritage entre deux classes. La règle de transformation a annoté cette structure en tant que *Generalization* en associant la classe mère et la fille aux participants du sous-patron.

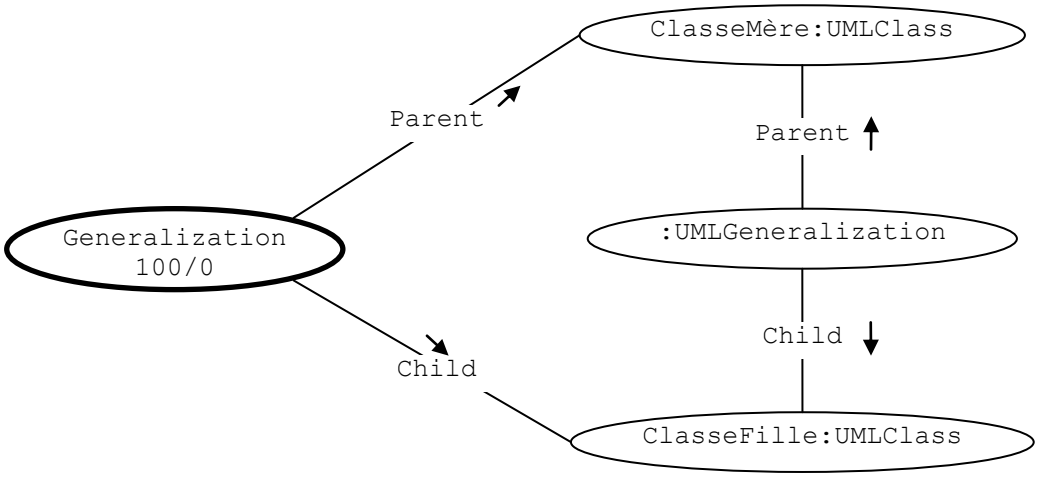


Figure 3.4 : Extrait d'un ASG annoté par le sous-patron Generalization.

Le nouveau sommet *Generalization* est tagué par deux pourcentages, le premier indiquant la fiabilité de la détection, et le deuxième le seuil minimal à atteindre pour considérer la détection comme fiable. Ce seuil est particulièrement utile lorsque plusieurs sous-patrons sont regroupés pour former une structure de plus haut niveau.

De manière générale, pour détecter les patrons, toutes les règles de transformation des sous-patrons les plus fondamentaux sont appliquées sur l'ASG. Après cette première passe, l'algorithme tente d'analyser les nouveaux sommets apparus en appliquant les règles de transformation des sous-patrons de niveau supérieur. Il est ainsi possible de cibler les patrons de conception du modèle en identifiant les sous-patrons qui le constituent. Cependant, bien que certains sous-patrons soient génériques au point de reconnaître, par exemple, toutes les manières possibles d'affecter une valeur à un attribut dans une méthode d'une classe, l'algorithme de détection de Fujaba ne permet pas de détecter des patrons dont la forme n'est pas réellement l'assemblage de sous-patrons prévus [Wenzel05_a].

Afin de remédier à cette limite, il est possible d'utiliser des mécanismes d'évaluation floue pour détecter structurellement des fragments dans des modèles [Wenzel05_a], en modifiant notamment la représentation du modèle et des patrons recherchés. Appliqués à des patrons de conception structuraux dans des modèles UML, ces mécanismes permettent de détecter des patrons utilisés différemment de ce qui est préconisé, ainsi que des patrons incomplets.

I.3.1. Définition structurelle des patrons

Pour pouvoir détecter les patrons directement dans les modèles UML, il est nécessaire de les décrire en utilisant une combinaison d'éléments appelés « rôles », au sens UML. Un rôle correspond au nom d'une métaclasse, ou par extension, le type d'un élément du modèle. Ainsi, chaque métaclasse d'un métamodèle représente un rôle particulier. Il est possible de décrire structurellement les patrons en utilisant le rôle de ses éléments (*Classifier*, *Association*…), couplé à des contraintes OCL [OMG06], afin de pouvoir décrire la complexité des arrangements de certains patrons et de préciser l'organisation des éléments les uns par rapport aux autres. Ces contraintes renforcent les propriétés des éléments inclus

dans les rôles, par exemple les attributs d'une métaclasse (navigabilité, visibilité, type d'agrégation…) et font référence aux autres rôles du patron. Ainsi, un patron est décrit par un ensemble de rôles le définissant structurellement, couplé à un ensemble de contraintes OCL. Cette description peut être vue comme l'ensemble des responsabilités structurelles du patron. La figure 3.5 illustre cette représentation des patrons par les rôles.

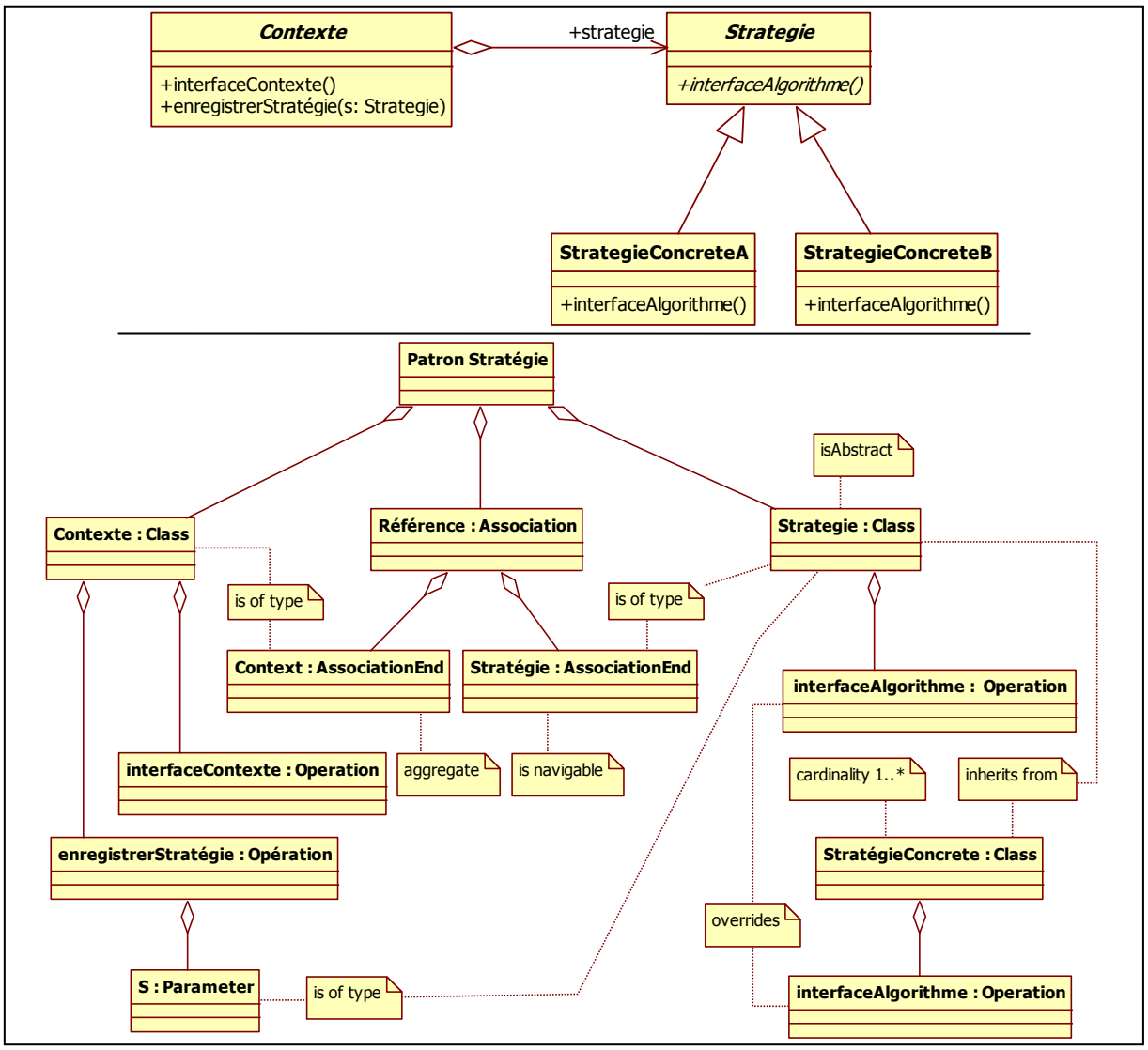


Figure 3.5 : Le patron *Stratégie* en UML et sa représentation par rôles [Wenzel05_a]

Il est possible de remarquer sur cette figure que chaque rôle caractérise une responsabilité du patron. Les instructions marquées dans les notes UML représentent les contraintes OCL.

I.3.2. Détection semi-automatique

En utilisant ce mode de représentation, il est possible de détecter des patrons de conception de la même manière qu'un « casting de théâtre » [Wenzel05_a]. Chaque élément du modèle est candidat à un des rôles du patron. Cela peut être une classe, une association, ou tout élément du métamodèle associé au modèle. La détection revient ainsi à assigner, à certains éléments du modèle, un rôle dans le patron recherché. Chaque

association « *élément-rôle* » est quantifiée par une valeur représentant à quel point l'élément peut jouer le rôle concerné. Ainsi, un élément est quantifié à 0% s'il n'a aucun rapport avec le rôle, ou à 100% s'il correspond parfaitement. Pour obtenir 100%, un élément doit être de même type que le rôle et répondre positivement à chacune des contraintes décrites dans le patron. Finalement, c'est le développeur qui décide quels sont les éléments les plus adéquats pour les patrons recherchés.

La méthode de détection se décompose en trois étapes :

1. Les rôles sont classifiés hiérarchiquement pour améliorer le « casting ». L'ensemble des rôles du patron est représenté sous forme de graphe, les sommets étant distingués en fonction du rôle qu'ils représentent. Le graphe est ensuite trié en fonction du type de chaque sommet, et des relations entre les sommets. L'idée est d'obtenir une hiérarchie avec, au niveau le plus bas, les rôles ayant le moins de dépendances (*Attribute*, *Method*, *Parameter*…) et, au niveau le plus haut, les plus dépendants (*Classifier*, *Association*).
2. La détection s'effectue alors niveau par niveau, en commençant par les rôles les moins dépendants. Les contraintes OCL de chaque rôle sont confrontées à chaque *Classifier*, *Association* ou tout autre élément du métamodèle. Si les contraintes d'un rôle ne sont pas validées sur un élément, le score de *0%* est attribué à cet élément. Si elles répondent toutes *vrai*, c'est *100%* qui est attribué. C'est ce pourcentage qui représente la possibilité qu'a l'élément de pouvoir jouer ce rôle. Tous les éléments ayant *0%* ne sont pas considérés comme candidats.
3. Une fois que tous les éléments du métamodèle ont été analysés, une deuxième phase démarre afin de vérifier à nouveau chaque candidat potentiel, mais cette fois-ci en redescendant la hiérarchie. Il est ainsi possible de vérifier si les éléments candidats ont bien les bonnes dépendances avec les autres candidats.

À l'issue de la détection, les candidats sont présentés au concepteur qui peut choisir de les conserver ou non. Implémentée dans un composant de Fujaba [FUJABA05] [Wenzel05_b], cette technique de détection cible des fragments « ressemblant » à des contextualisations de patrons de conception. La figure 3.6 montre le résultat de la recherche du patron *Stratégie* présenté figure 3.5.

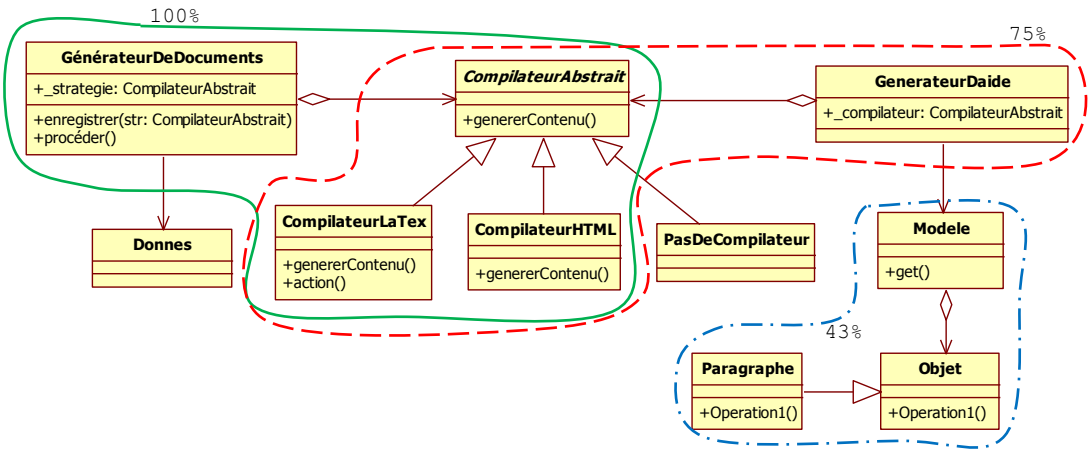


Figure 3.6 : Les fragments du patron *Stratégie* détectés dans un modèle [Wenzel05_b]

Trois fragments ont été identifiés dans ce modèle. Le premier, quantifié à 100%, correspond exactement à une contextualisation du patron *Stratégie*. Le second est quantifié à 75% car la classe *GenerateurDaide* ne détient pas de méthode. Enfin, le troisième n'est que quantifié de 43% car la classe *Objet* n'est pas abstraite et la classe *Modele* n'a pas la méthode *enregistrerStrategie* déclarée dans le patron.

I.3.3. Mise en relation avec notre problématique

Les principales limites de cette méthode concernent le temps d'exécution de la détection pour de gros modèles et le nombre de fragments identifiés. L'intégralité des métaclasses est passée en revue, ce qui est conséquent pour un modèle industriel. De plus, vérifier toutes les métaclasses n'est pas judicieux, car la plupart ne sont pas significatives dans la structure d'un patron. Les auteurs de cette méthode reconnaissent volontiers cette limite, mais jugent qu'un modèle conçu par un humain est de taille raisonnable pour ne pas être problématique [Wenzel05_a]. Cependant, issus de l'assemblage de plusieurs sous-modèles, certains modèles industriels peuvent être, selon nous, beaucoup plus conséquents que ce qu'un être humain est capable de gérer.

Pour la détection des patrons abîmés, cette technique pourrait être utilisable si sa contrainte de temps d'exécution n'était pas aussi forte, et si elle ne retournait pas autant de fragments différents. Comme il y a plusieurs patrons abîmés pour un seul patron de conception, le nombre de fragments à présenter au concepteur est trop important même pour un petit modèle, introduisant de fait un temps d'exécution de l'analyse prohibitif.

I.4. Détection par propagation de contraintes

En ne représentant plus le patron de conception à rechercher, mais en métamodélisant le problème qu'il résout, il est possible de détecter, en utilisant un procédé basé sur les CSP (Constraint Satisfaction Problem), des fragments de modèles remplaçables par des patrons de conception [ElBoussaidi08]. Il ne s'agit plus cette fois de redocumentation, mais vraiment d'aide à l'utilisation des patrons de conception. L'idée est de détecter, dans un modèle, un fragment conforme au métamodèle du problème d'un patron, et ainsi, de remplacer ce fragment par la solution proposée par le patron. Cette technique de détection reformule le problème d'homomorphisme de graphes proposé par [Rudolf00].

I.4.1. Homomorphisme de graphes par CSP

Les algorithmes d'homomorphisme de graphes, connus pour être NP-complets, peuvent être reformulés et résolus en utilisant les CSP [ElBoussaidi08]. Un CSP est défini par un ensemble fini de variables sous un domaine de définition connu, et par un ensemble fini de contraintes spécifiant comment des valeurs peuvent être affectées aux variables [Bacchus98]. Les CSP sont utilisés généralement pour résoudre des recherches complètes avec plusieurs niveaux de retour en arrière. Quand une variable est affectée à une valeur, toutes les contraintes de cette variable sont examinées, ce qui peut provoquer une propagation de contraintes vers les autres variables.

Pour construire un CSP, il est nécessaire de travailler avec deux graphes, celui à rechercher (le graphe source) et celui dans lequel s'effectue la recherche (le graphe destination) [Rudolf00]. Chaque sommet et chaque arc du graphe source sont associés à une variable distincte. Le domaine de définition des variables des sommets et des arcs correspond respectivement à l'ensemble des sommets et des arcs du graphe destination. La construction des contraintes s'effectue ensuite sur les paramètres à comparer pour valider la recherche.

I.4.2. Représentation des patrons de conception par triplets

Selon [Mili05], il est possible de définir les patrons de conception comme des triplets (MP, MS, T), où MP représente le problème résolu par le patron, MS est la solution pour résoudre le problème, et T l'opération de transformation qui permet de transformer MP en MS. En procédant ainsi, ce n'est plus vraiment le patron qui est représenté, mais le problème qu'il résout avec l'opération qui permet de le transformer en solution. MP et MS sont respectivement les métamodèles du problème et de la solution. Lorsqu'un concepteur découvre un fragment de son modèle conforme au métamodèle du problème résolu par le patron, il n'a plus qu'à appliquer la règle de transformation pour modifier le fragment. La figure 3.7 présente le métamodèle du problème résolu par le patron de conception *Visiteur*.

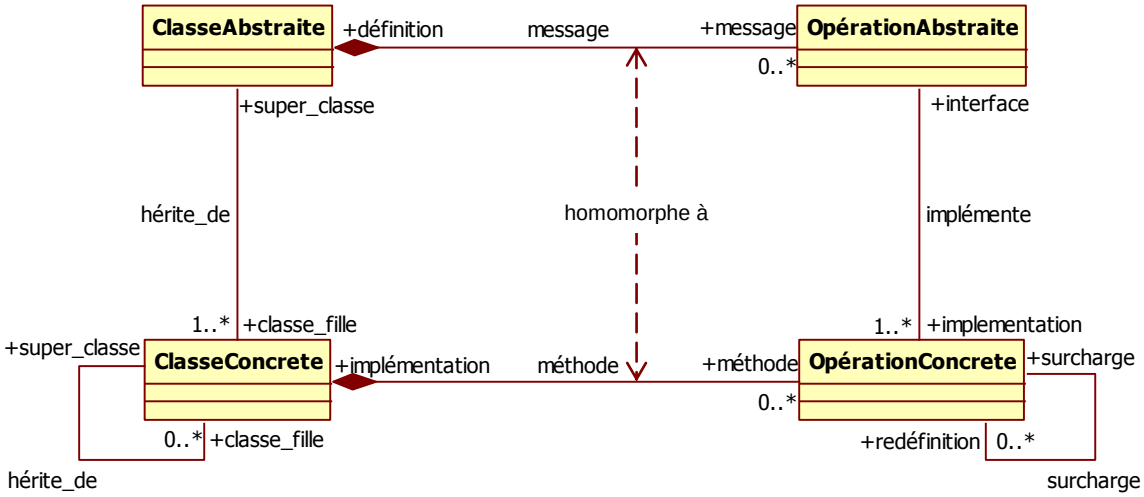


Figure 3.7 : Le métamodèle du problème résolu par le patron de conception *Visiteur* [ElBoussaidi08]

Ce métamodèle montre une hiérarchie de classes qui implémente le même comportement. La racine de la hiérarchie est représentée par la métaclasse *ClasseAbstraite* qui implémente un certain nombre d'opérations abstraites, par le biais du lien *message*. Elle a également un ensemble de classes concrètes qui implémentent toutes les opérations abstraites, par le lien *méthode*. Le fait que toutes les classes concrètes aient une opération concrète pour chaque opération abstraite est représenté par le lien *homomorphe à* [ElBoussaidi08].

I.4.3. Recherche de fragments conformes aux métamodèles de problèmes

Pour rechercher les fragments conformes au métamodèle par CSP, il est nécessaire de définir variables, domaines de définition et contraintes d'après la représentation des patrons. Pour un patron donné, les variables sont extraites de toutes les informations issues du métamodèle (*Classifier*, *Association*, *Attribute*…). Dans le cas de l'exemple présenté dans la figure 3.7, les variables sont : *var_ClasseAbstraite*, *var_ClasseConcrete*, *var_hérite_de*, *var_surcharge*… Leur domaine de définition est l'élément du métamodèle du modèle à analyser. Par exemple, pour une variable de type *Classifier*, comme *var_ClasseConcrete*, le domaine de définition est l'ensemble des *Classifier* du modèle à analyser. En procédant ainsi, la couverture du modèle est totale, et tous les cas sont envisagés. Enfin, les contraintes sont définies par les attributs des métaclasses du métamodèle du patron de conception. Ainsi, chaque contrainte définit les particularités de chaque élément, et donc de chaque variable. Lors de la recherche, chaque élément du modèle à analyser est soumis aux contraintes des variables du patron.

I.4.4. Mise en relation avec notre problématique

Deux limites se détachent de cette approche par CSP. Tout d'abord, le fonctionnement de la méthode est fortement lié à la représentation des problèmes solubles par les patrons. Un problème mal métamodélisé provoque la détection de fragments n'ayant aucun lien avec le patron concerné. Cette représentation par métamodèle doit donc être très précise et obligatoirement manuelle. En effet, il n'est pas possible d'automatiser cette représentation puisqu'à l'exception de la structure, aucune information existante n'est disponible dans le GoF pour concevoir un tel métamodèle. De plus, le fait que cette représentation soit manuelle ajoute un côté subjectif au patron, provoquant la perte du consensus communautaire du patron.

La deuxième limite concerne également le métamodèle, mais sur le fait qu'il ne représente qu'une seule manière de concevoir le problème à résoudre. Il est ainsi nécessaire que le fragment implémente le problème conformément au métamodèle sans aucune différence. Par exemple, si un concepteur a implémenté incomplètement un patron, le fragment n'est pas détectable, car non conforme au métamodèle.

Vis-à-vis de notre problématique, cette approche paraît peu compatible, car nous désirons identifier les mauvaises manières de résoudre un problème, en étant capable de s'adapter aux compétences du concepteur. Ainsi, notre méthode doit être capable de détecter si le concepteur n'a pas utilisé le patron, s'il l'a mal utilisé, ou s'il a utilisé une autre manière de faire. Nous ne pouvons donc pas utiliser ce système de représentation et de détection des patrons pour nos patrons abîmés.

I.5. Synthèse vis-à-vis de notre problématique

Nous venons de décrire trois techniques de détection de fragments de modèles, choisies parmi les travaux existants dans le domaine [Albin-Amiot01_a] [Antoniol01] [Balanyi03] [Bergenti00] [Brown96] [Costagliola05] [Gueheneuc04] [Heuzeroth03] [Wendehals03]. Le tableau 3.1 reprend les spécificités de notre problématique et les mets en relation avec les travaux présentés dans cette section.

	Tsantalis	Wenzel	ElBoussaidi
Ne pas effectuer de prétraitement sur le modèle à analyser		OK	
Ne pas utiliser d'informations issues du code			
Effectuer la détection par étapes successives		OK	OK
Limiter le temps d'exécution			OK
Limiter la multiplication des fragments identifiés			OK
Ne pas dégrader le consensus du patron	OK	OK	
Détecter toutes les contextualisations possibles	OK	OK	

Tableau 3.1 : Mise en relation des spécificités de notre problématique

Étant donnée la taille de notre base de patrons abîmés, nous devons garantir une technique de détection rapide, même sur de grands modèles. Il n'est pas envisageable de tester toutes les métaclasses du modèle à analyser, et donc, il est nécessaire d'effectuer un filtrage pour limiter rapidement le nombre de comparaisons. Dans la même idée, le nombre de patrons abîmés et le fait que cette base ne soit pas figée, nous imposent d'être capables de travailler directement et sans prétraitement, avec les patrons exprimés sous forme de modèles. Enfin, notre approche se situant au niveau modèle, nous devons être capables de détecter les patrons abîmés sans utiliser d'informations issues du code. Ainsi, aucun contenu de méthode ou élément d'exécution ne pourra être utilisé, à l'exception des diagrammes dynamiques prévus à cet effet. De plus, un patron abîmé constituant, vis-à-vis de la détection, une base génératrice d'une famille de contextualisations possibles, nous devons être à même d'identifier tous les fragments inclus d'une même famille. Pour satisfaire toutes ces contraintes, nous avons défini notre propre technique de conception, basée sur une analyse de concordances structurelles.

II. Concordances structurelles

Un patron abîmé est doté d'un ensemble de particularités structurelles remarquables permettant de le décrire structurellement, et ainsi de reconnaître sa contextualisation dans un modèle. Découpées en deux sous-ensembles, les particularités locales et les particularités globales sont à la base de la méthode de détection que nous utilisons pour identifier des fragments alternatifs dans un modèle.

Pour mémoire, nous considérons que le modèle dans lequel nous effectuons la recherche, ainsi que les patrons de conception et abîmés sont décrits en UML 1.5 au format XMI. Ainsi, nous considérons que nos modèles sont assimilables à un graphe représentant le métamodèle UML. La figure 3.8 présente un extrait du métamodèle.

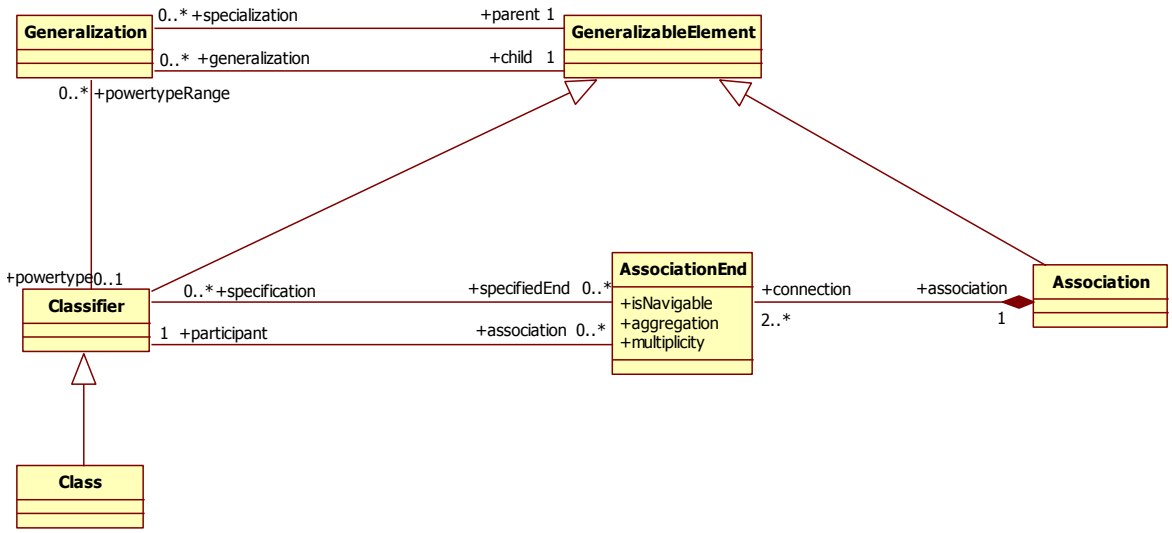


Figure 3.8 : Extrait du métamodèle UML 1.5 ciblé autour de *Classifier*

Nous utilisons ces métaclasses pour construire les graphes représentant nos concepts. Ainsi, la figure 3.9 illustre les deux représentations pour un patron abîmé, en UML et sous forme de graphe XMI.

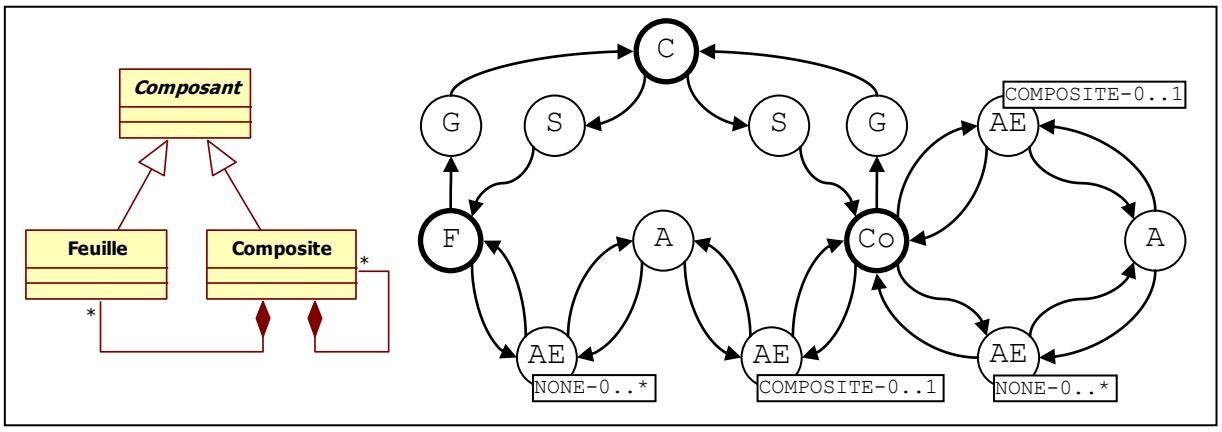


Figure 3.9 : Un patron abîmé et sa représentation sous forme de graphe

Les sommets sont séparés en deux sous-ensembles distincts : $\boldsymbol{Vc}$, contenant l'ensemble des classes du modèle, et $\boldsymbol{Vm}$, contenant l'ensemble des éléments du métamodèle permettant de relier les classes entre-elles (les métaclasses *Association*, *AssociationEnd*, *Generalization* et son inverse, *Specialization*). La présence des sommets {AE} nous est imposée par le métamodèle. En effet, nous pouvons remarquer dans la figure 3.8 la métaclasse *AssociationEnd* séparant les *Classifier* des *Association*. Les sommets {AE} sont tagués par des attributs de la métaclasse *AssociationEnd*, afin de caractériser l'extrémité des associations, et de fait, les relations entre classes. Par exemple, pour la classe *Composite* appartenant à $\boldsymbol{Vc}$, les seuls sommets adjacents appartenant à $\boldsymbol{Vm}$ sont {AE}, {G} et {S} en excluant l'ensemble {A} qui n'est accessible que depuis {AE}.

Dans cette section, nous détaillons notre méthode de détection. Nous abordons tout d'abord la théorie de notre méthode de détection par concordances structurelles, puis le concept de participant de référence qui nous permet, lors de la détection, de différencier des fragments représentant un même patron abîmé. Nous décrivons ensuite les

informations supplémentaires que nous avons ajoutées aux patrons abîmés, afin d'affiner la détection en décrivant des relations ou des particularités interdites dans les fragments. En suivant, nous présentons l'algorithme de détection. Nous montrons que son découpage en trois phases permet de filtrer progressivement les résultats pour ne conserver que des fragments alternatifs. Nous présentons ensuite notre technique de transformation des fragments identifiés, permettant de les remplacer par les patrons de conception adéquats. Cette technique est directement déduite des particularités de notre propre méthode de détection. Dans une dernière partie, nous justifions le choix d'OCL pour le codage des requêtes, ainsi que la plate-forme utilisée en support de la détection.

II.1. Concordance des particularités structurelles

Les particularités structurelles d'un patron abîmé nous permettent de détecter des fragments alternatifs dans des modèles. Rapportées aux graphes, elles nous permettent de détecter des familles de sous-graphes, car elles décrivent, de manière précise, les patrons abîmés ainsi que les fragments alternatifs qu'ils peuvent générer. Notre méthode de détection utilise le caractère local et global des particularités structurelles pour vérifier la concordance structurelle des fragments alternatifs avec les patrons abîmés.

La définition 3.1 définit de manière formelle un patron abîmé. Comme vu précédemment, il s'agit d'un graphe orienté constitué de deux ensembles de sommets $\boldsymbol{Vc}$ et $\boldsymbol{Vm}$, représentant respectivement les participants du patron abîmé et les instances de métaclasses décrivant les relations entre les participants.

Chaque patron abîmé est doté d'un unique participant de référence que nous notons *reference_sp*. Il représente un sommet particulier de $\boldsymbol{Vc}_{sp}$ que nous détaillons dans la partie II.2. Ce sommet est choisi par un oracle en fonction de sa complexité structurelle et de ses responsabilités vis-à-vis du problème à résoudre.

$$
\begin{aligned}
\forall sp \in \{Patrons\ abimés\} : sp = graphe\ orienté\left(\boldsymbol{Vc}_{sp}, \boldsymbol{Vm}_{sp}, \boldsymbol{E}_{sp}\right) \text{ et } reference\_sp\ /\ & \\
\boldsymbol{Vc}_{sp} = \{Participants\ du\ patron\ abimé\} & \\
\quad \text{tel que x} \in \boldsymbol{Vc}_{sp} \Rightarrow \text{x } instance\_de \text{ Classifier} & \\
\wedge\ \boldsymbol{Vm}_{sp} = \{Liens\ entre\ les\ participants\} & \\
\quad \text{tel que x} \in \boldsymbol{Vm}_{sp} \Rightarrow \text{x } instance\_de & \\
\quad\quad \text{AssociationEnd} \vee \text{Association} & \\
\quad\quad \vee \text{Generalization} \vee \text{Specialization} & \\
\wedge\ \boldsymbol{E}_{sp} = \{Arcs\} & \\
\wedge\ \forall c \in \boldsymbol{Vc}_{sp}, adjacent(c) \subseteq \boldsymbol{Vm}_{sp} & \\
\wedge\ reference\_sp \in \boldsymbol{Vc}_{sp} &
\end{aligned}
$$

Définition 3.1 : Un patron abîmé, du point de vue de la détection

De la même manière que pour un patron abîmé, nous pouvons définir formellement le modèle à analyser, comme le montre la définition 3.2.

$$\forall m \in \{Modèles\ à\ analyser\} : m = graphe\ orienté(\boldsymbol{Vc}_m, \boldsymbol{Vm}_m, \boldsymbol{E}_m)\ /$$
$$\boldsymbol{Vc}_m = \{Classes\ du\ modèle\}$$
$$\text{tel que x} \in \boldsymbol{Vc}_m \Rightarrow \text{x } instance\_de \text{ Classifier}$$
$$\wedge\ \boldsymbol{Vm}_m = \{Liens\ entre\ les\ classes\}$$
$$\text{tel que x} \in \boldsymbol{Vm}_m \Rightarrow \text{x } instance\_de$$
$$\text{AssociationEnd} \vee \text{Association}$$
$$\vee\ \text{Generalization} \vee \text{Specialization}$$
$$\wedge\ \boldsymbol{E}_m = \{Arcs\}$$
$$\wedge\ \forall c \in \boldsymbol{Vc}_m, adjacent(c) \subseteq \boldsymbol{Vm}_m$$

Définition 3.2 : Le modèle à analyser, du point de vue de la détection

Nous sommes donc en présence de deux graphes orientés dont nous recherchons certaines combinaisons d'occurrences du premier dans le deuxième. Afin d'éviter une explosion combinatoire des possibilités de recherche, et donc pour limiter la complexité du problème, nous procédons à un premier filtrage de l'ensemble des sommets ayant les particularités structurelles locales adéquates.

La première étape consiste à rechercher tous les sommets du graphe $\boldsymbol{m}$ conformément au prédicat 3.1 *local_CPR*. Ce prédicat, signifiant « Concordance des particularités remarquables locales », permet de vérifier si un sommet $\boldsymbol{c}$ appartenant au graphe $\boldsymbol{m}$ a, au moins, les mêmes sommets adjacents qu'un sommet $\boldsymbol{p}$ du graphe $\boldsymbol{sp}$. Ainsi, si $\boldsymbol{c}$ est *local_CPR* avec $\boldsymbol{p}$, la classe correspondant au sommet $\boldsymbol{c}$ a, au minimum, les mêmes particularités structurelles locales que le participant du patron abîmé correspondant au sommet $\boldsymbol{p}$. La comparaison des sommets adjacents est effectuée avec une relation d'équivalence comparant le type et les attributs des sommets adjacents, comme le montre la définition 3.3, moyennant un relâchement de certaines contraintes défini dans la partie II.3 de ce chapitre.

$$\forall (e1, e2) \in \boldsymbol{Vm}_m \times \boldsymbol{Vm}_{sp} : e1 \Leftrightarrow e2$$
$$\Leftrightarrow type(e1) = type(e2)$$
$$\wedge\ \forall\ attr \in \{\text{attributs de } e1 \text{ ou } e2\} :\ valuation(attr_{e1}) = valuation(attr_{e2})$$

Définition 3.3 : Relation d'équivalence entre deux sommets de Vm appartenant à des graphes distincts

Par extension, il vient la définition 3.4.

$$\forall Ei \subseteq \boldsymbol{Vm}_m, \forall Ej \subseteq \boldsymbol{Vm}_{sp} : Ei \supseteq Ej \ \Leftrightarrow\ \forall ej \in Ej, \exists ei \in Ei\ /\ ej \Leftrightarrow ei$$

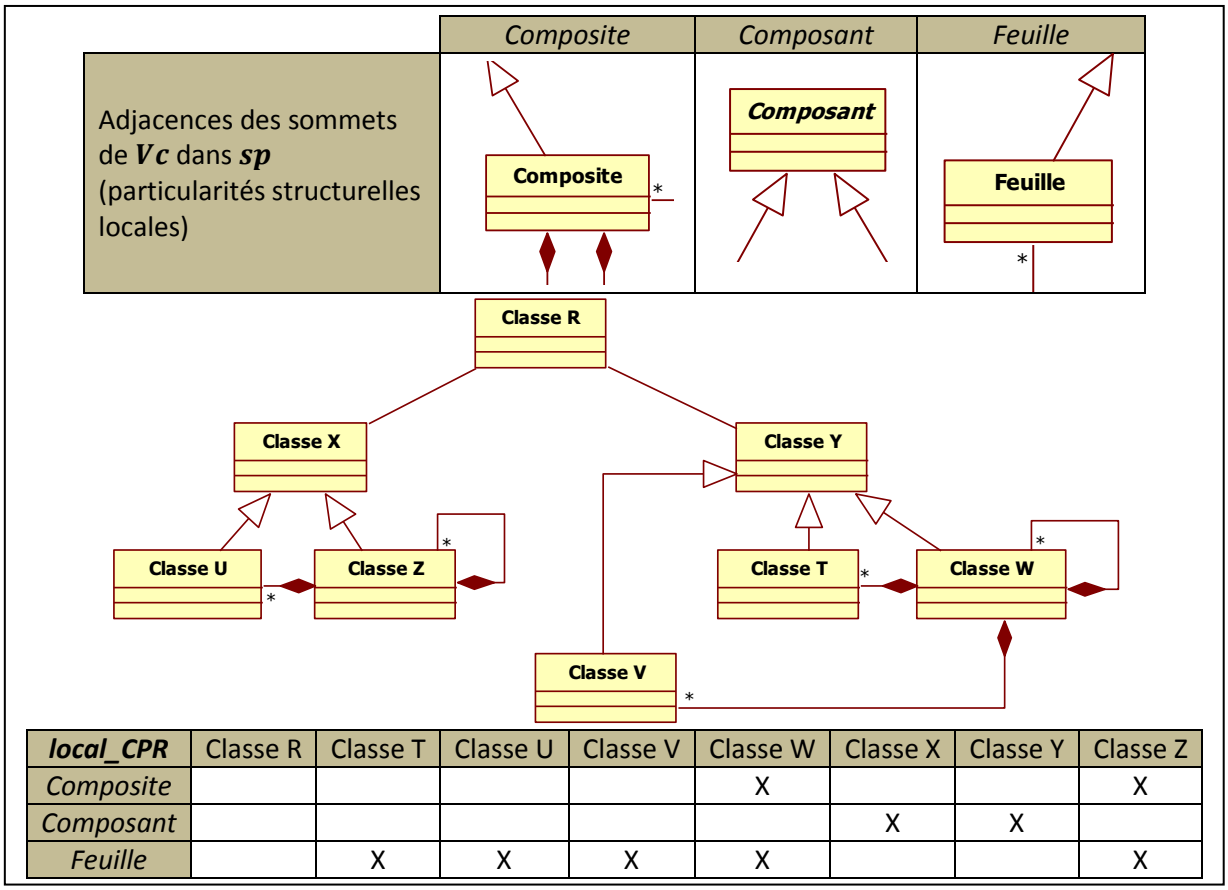
Définition 3.4 : Relation d'inclusion entre deux sous-ensembles de Vm inclus dans des graphes distincts

Comme les sommets adjacents à un sommet de $\boldsymbol{Vc}$ appartiennent à $\boldsymbol{Vm}$ et comme tous les sommets $\boldsymbol{Vm}$ sont fortement typés, les sommets de $\boldsymbol{Vc}$ peuvent être filtrés grâce à leurs particularités structurelles locales, comme le montre le prédicat 3.1.

$$\forall c \in \boldsymbol{Vc}_m, \forall p \in \boldsymbol{Vc}_{sp} : local\_CPR(c, p) \Rightarrow adjacent(c) \supseteq adjacent(p)$$

Prédicat 3.1 : Concordance des particularités structurelles locales

Le prédicat *local_CPR* est illustré dans la figure 3.10 en utilisant une représentation UML pour plus de lisibilité.



local_CPR	Classe R	Classe T	Classe U	Classe V	Classe W	Classe X	Classe Y	Classe Z
Composite					X			X
Composant						X	X	
Feuille		X	X	X	X			X

Figure 3.10 : Illustration du prédicat *local_CPR* sur un exemple

Les classes marquées dans le tableau du bas de la figure valident le prédicat *local_CPR* avec le participant correspondant. Il est possible de remarquer que les classes ont les mêmes sommets adjacents que leurs participants, à l'exception des classes *W*, *X*, *Y* et *Z* qui en ont en plus. Par exemple, nous pouvons constater que la classe *W* a trois sommets `AE` avec l'attribut `COMPOSITE-0..1`. C'est en partie grâce au fait qu'un sommet de $\boldsymbol{Vc}$ puisse avoir plus d'adjacences, que nous pouvons détecter toutes les différentes contextualisations possibles de patrons abîmés. De plus, nous pouvons remarquer que *Classe W* et *Classe Z* valident le prédicat sur deux participants différents du patron, *Feuille* et *Composite*. Les particularités structurelles locales de *Feuille* sont effectivement incluses dans celles de *Composite*. Sans les particularités structurelles globales, nous ne pouvons pas encore différencier les classes *Composite* des classes *Feuille*.

Après avoir comparé tous les sommets de $\boldsymbol{Vc}_{sp}$ à tous ceux de $\boldsymbol{Vc}_{m}$, c'est-à-dire tous les participants du patron abîmé avec toutes les classes du modèle à analyser, nous obtenons un ensemble de sommets ayant leurs sommets adjacents au moins identiques à ceux des participants du patron abîmé. Ce premier prédicat sert de filtre sur l'ensemble des sommets de $\boldsymbol{m}$.

Le prédicat 3.2 *global_CPR* permet de vérifier la concordance des particularités structurelles globales, c'est-à-dire si un sous-graphe $\boldsymbol{sf}$ de $\boldsymbol{m}$ est isomorphe à $\boldsymbol{sp}$.

$$\forall sf, \forall sp : global\_CPR(sf, sp) \Leftrightarrow \exists g : isomorphisme\ entre\ sf\ et\ sp$$

Prédicat 3.2 : Concordance des particularités structurelles globales

Un sous-fragment *global_CPR* à $\boldsymbol{sp}$ a, par définition, le même nombre de sommets que le patron abîmé. Bien que la contextualisation du patron abîmé provoque la multiplication de certains sommets, toutes les combinaisons, telles que chaque classe représente un participant distinct, restent isomorphes au patron abîmé. La figure 3.11 illustre cet isomorphisme des sous-fragments du modèle identifiés dans la figure 3.10.

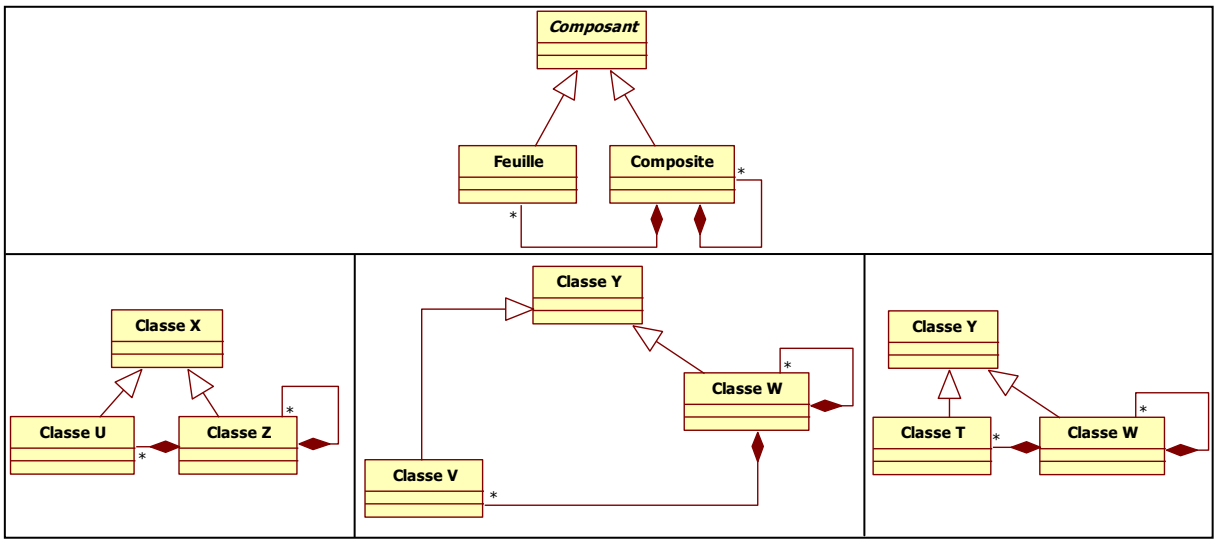


Figure 3.11 : Isomorphisme des sous-fragments

Dans le modèle de la figure 3.10, d'après les sommets identifiés comme étant *local_CPR* avec les sommets du patron abîmé, nous ne pouvons former que trois sous-fragments conformes à *global_CPR*. Par exemple, la combinaison *Classe U*, *Classe W* et *Classe Y*, n'est pas un sous-fragment, car même s'il y a le même nombre de sommets que dans le patron abîmé et que chaque sommet est *local_CPR* avec un sommet différent du patron abîmé, il n'y a pas d'isomorphisme entre cette combinaison et le patron abîmé.

Ainsi, le prédicat *global_CPR* nous permet d'éliminer *Classe W* et *Classe Z* des responsabilités de *Feuille*, puisqu'il n'est pas possible de former une combinaison de classes conforme au prédicat *global_CPR* avec l'une de ces classes aux responsabilités de *Feuille*.

Reste maintenant à former les fragments alternatifs, c'est-à-dire à coupler les sous-fragments partageant des mêmes sommets. Un fragment alternatif $\boldsymbol{af}$ est un sous-graphe de $\boldsymbol{m}$ incluant au moins un sous-fragment isomorphe au patron abîmé et tel que tout graphe induit par une combinaison de sommets référençant une fois et une seule chaque participant du patron abîmé reste isomorphe au patron abîmé. De plus, un seul sommet de $\boldsymbol{af}$, que nous nommons *classe_reference,* est *local_CPR* avec *reference_sp*, le sommet de référence du patron abîmé.

Le prédicat 3.3 *frag_alternatif* permet ainsi de vérifier si un fragment $\boldsymbol{af}$ est bien un ensemble de sous-fragments, chacun isomorphe à $\boldsymbol{sp}$.

$$\forall af \subseteq \boldsymbol{m}, \forall sp : frag\_alternatif(af, sp)$$
$$\Leftrightarrow \forall sf \subseteq af : |sf| = |sp|$$
$$\wedge \forall c_1, c_2 \in \boldsymbol{Vc}_{sf}, participant(c_1) \neq participant(c_2) \Rightarrow global\_CPR(sf, sp)$$

Prédicat 3.3 : Identification d'un fragment alternatif

L'application 3.1 *participant* associe à chaque sommet du fragment $\boldsymbol{af}$, un sommet du patron abîmé tel qu'ils soient en concordance structurelle locale et qu'ils soient reliés de la même manière au participant de référence.

application $participant$: $\boldsymbol{Vc}_{af} \rightarrow \boldsymbol{Vc}_{sp}$
$c \mapsto c'$
$participant(c) = c'$
si $local\_CPR(c, \text{reference\_sp})$ alors
$c' = \text{reference\_sp}$
et c' est la classe de référence
sinon
$local\_CPR(c, c')$
$\wedge\ global\_CPR(sg(c, \text{classe\_reference}), sg(c', \text{reference\_sp}))$
où sg correspond au graphe induit par l'ensemble des sommets considérés

Application 3.1 : Association à chaque classe du fragment d'un participant du patron abîmé

En procédant ainsi, nous pouvons former des fragments représentant toutes les contextualisations possibles du patron abîmé. En effet, même s'il n'est pas possible d'anticiper quelle forme a un fragment alternatif, il est certain que quelle que soit sa forme, il est composé de sous-fragments isomorphes au patron abîmé, puisque tous les fragments alternatifs possibles auront toujours leurs classes reliées de la même manière aux participants respectifs du patron abîmé.

La figure 3.12 présente deux fragments alternatifs composés de sous-fragments isomorphes à un patron abîmé.

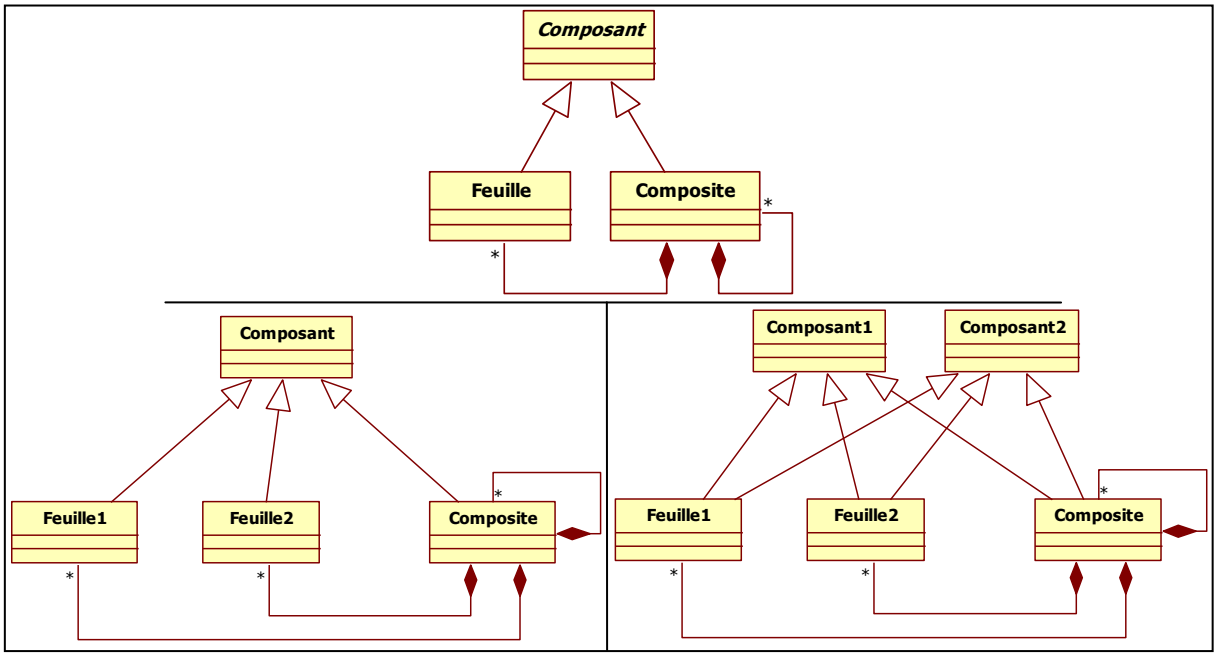


Figure 3.12 : Deux fragments alternatifs composés de sous-fragments isomorphes au patron abîmé

Les sous-fragments du fragment alternatif de gauche sont {*Composite*, *Composant*, *Feuille1*} et {*Composite*, *Composant*, *Feuille2*}. Ceux du fragment de droite sont {*Composite*, *Composant1*, *Feuille1*}, {*Composite*, *Composant2*, *Feuille1*}, {*Composite*, *Composant1*, *Feuille2*} et {*Composite*, *Composant2*, *Feuille2*}. Il est donc possible de remarquer que les deux fragments présentés sont deux contextualisations différentes, mais reconnues comme étant des fragments alternatifs.

Un cas particulier est présenté dans la figure 3.13.

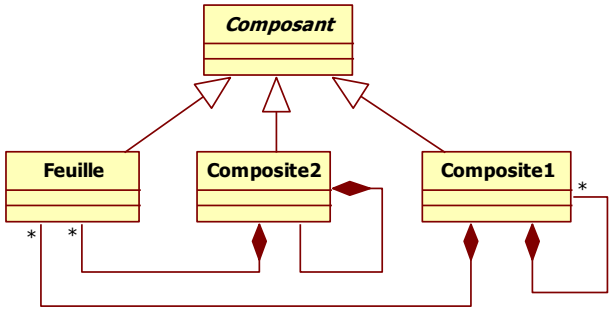


Figure 3.13 : Cas particulier de fragment alternatif

Les deux sous-fragments de ce modèle sont isomorphes au patron abîmé : {*Composite1*, *Composant*, *Feuille*}, {*Composite2*, *Composant*, *Feuille*}. Cependant, si nous analysons ce cas, nous pouvons nous demander si, hormis sa structure, il constitue un véritable fragment alternatif. En effet, les deux classes *Composite* soulèvent ici une question d'intention du fragment. Pour que chacune puisse gérer idéalement la composition, elles doivent avoir connaissance de toutes les classes composables. Dans ce cas, il s'agit de chacune des feuilles et également, de chacun des composites. D'après le patron abîmé non contextualisé, une classe *Composite* doit se composer d'elle-même, donc, d'un *Composite*, et par extension de tous les autres composites du modèle. Ainsi, il devrait y avoir, dans le modèle, une double relation de composition entre les classes *Composite1* et *Composite2*. Nous proposons donc que la figure 3.13 présente deux fragments alternatifs distincts {*Composite1*, *Composant*, *Feuille*} et {*Composite2*, *Composant*, *Feuille*}. Ainsi, nous n'autorisons pas un fragment à avoir deux classes ayant les responsabilités de *Composite*. Nous avons nommé cette particularité supplémentaire, le « participant de référence », nécessaire à la représentation des résultats de la détection.

II.2. Participant de référence

Nous allons voir sur quelques exemples simples l'importance relative des différents participants d'un patron abîmé. Ce qui fait qu'un patron de conception est la meilleure solution réside, en ce qui nous concerne, dans son organisation structurelle, support obligatoire à toute collaboration entre objets. Il en est de même pour les patrons abîmés : leur faiblesse est intimement liée à l'organisation des participants entre eux. Cependant, tous les participants ne jouent pas le même rôle dans la faiblesse du patron abîmé. Prenons l'exemple du patron abîmé « développement de la composition sur *Composite* », représenté dans la figure 3.9. Nous l'avons nommé ainsi, car tous les éléments permettant la

composition hiérarchique des objets sont exprimés sur le participant *Composite*. Prenons en considération le modèle UML présenté dans la figure 3.14, et essayons de répondre à la question : ce modèle, issu de la dénaturation d'un patron abîmé, peut-il toujours être considéré comme un patron abîmé du *Composite* ?

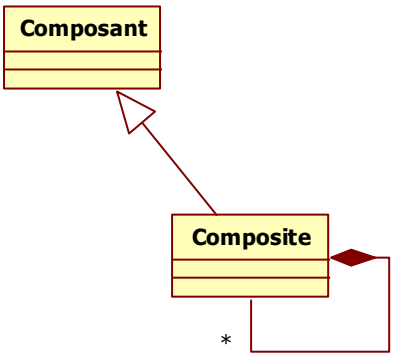


Figure 3.14 : Un patron abîmé du *Composite* sans *Feuille*

La dénaturation a consisté à supprimer toutes les occurrences d'un participant du patron, les *Feuilles*. Cette absence n'est pas forcément gênante, car même si nous perdons la possibilité d'ajouter des éléments terminaux à l'arbre hiérarchique de composition, nous ne perdons pas l'intention du patron.

La figure 3.15 illustre une autre dénaturation du patron abîmé.

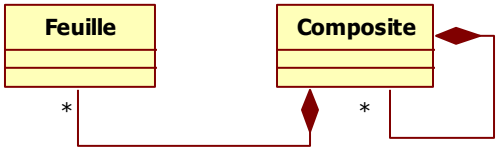


Figure 3.15 : Un patron abîmé du *Composite* sans *Composant*

Cette fois, la dénaturation a consisté à supprimer le participant *Composant* du patron abîmé. Contrairement au cas précédent, la dégradation du patron est plus importante, notamment parce qu'elle invalide tous les points forts du patron correspondant. En effet, nous perdons totalement l'uniformité du protocole qu'il soit de comportement ou de composition. Cependant, malgré sa forte dénaturation et la lourdeur du code engendré, ce modèle conserve l'intention du patron de conception.

La figure 3.16 illustre un dernier cas de figure en supprimant le dernier participant du patron.

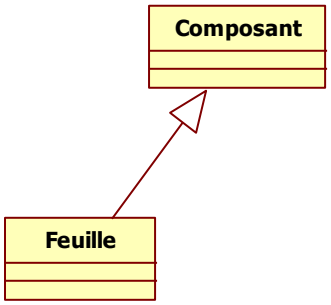


Figure 3.16 : Un patron abîmé du *Composite* sans *Composite*

Cette fois-ci, la dénaturation du patron abîmé invalide l'intention du patron, car plus aucune composition n'est possible. Cette conséquence s'explique par le fait que c'est le participant *Composite* qui gère totalement les responsabilités de la composition des objets, intention première du patron. Ce participant joue donc un rôle prépondérant dans le patron abîmé. C'est lui que nous appelons le participant de référence et qui nous permet de fixer dans le patron abîmé quel participant ne doit pas être multiplié. Ainsi, lors de la détection, nous créons autant de fragments alternatifs qu'il y a de classes ayant les mêmes particularités structurelles que le participant de référence.

II.3. Relations autorisées, interdites ou facultatives

Nous l'avons vu précédemment, notre méthode de détection intègre la possibilité de détecter plusieurs formes possibles de fragments alternatifs, en se basant sur la notion de participant de référence. Dès lors, toute classe n'ayant pas les responsabilités du participant de référence peut se retrouver multipliée dans le fragment alternatif, alors que celles marquées référence seront séparées dans des fragments différents. Mais que se passe-t-il si une classe du fragment est reliée de plus d'une autre façon que prévue à l'une des autres classes du fragment ? La figure 3.17 illustre ce cas sur deux fragments, l'un avec une relation réflexive sur *Composant* et l'autre avec un lien d'héritage entre *Composite* et *Feuille.*

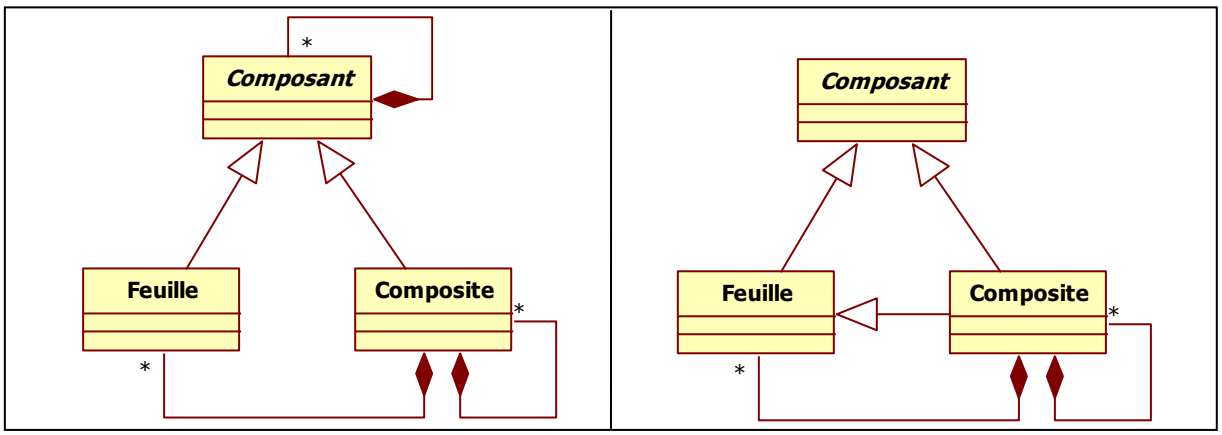


Figure 3.17 : Deux fragments de modèle avec une relation supplémentaire par rapport au patron abîmé

Si nous appliquons la méthode présentée précédemment, les deux fragments sont reconnus comme alternatifs, car ils respectent bien les particularités structurelles du patron abîmé correspondant.

Le fragment de gauche a bien la même intention que le patron abîmé. En effet, même si le *Composant* a la possibilité de se composer lui-même, et donc, par héritage, la *Feuille* également, il n'y a, a priori, aucune conséquence sur la capacité du fragment à composer des objets. C'est la résultante d'un fragment plus dégradé que ce que prévoit le patron abîmé et qui justifie d'autant plus le remplacement par le patron de conception.

Pour le fragment de droite, « la donne est différente », car ce lien d'héritage supplémentaire modifie les responsabilités des classes concernées. Un *Composite* héritant d'une *Feuille* n'a pas de sens dans la composition hiérarchique d'objets, puisqu'un objet non terminal ne peut pas être un objet terminal spécialisé. Le fragment a sûrement un sens dans le modèle d'où il a été extrait, mais isolé ainsi, il ne peut pas avoir l'intention de composer des objets. Ainsi, nous avons choisi de rendre discriminant tout fragment ayant des liens invalidant la structure mise en œuvre pour supporter l'intention d'un patron, afin de limiter le nombre de fragments potentiellement alternatifs. Nous appuyons ce choix sur le fait que l'activité peut produire beaucoup de fragments potentiellement alternatifs dont le concepteur devra vérifier l'intention un par un. Nous essayons donc, en choisissant ce mode de fonctionnement, de limiter au maximum le nombre de fragments présentés au concepteur.

Pour reconnaître quelles relations du fragment alternatif sont discriminantes, nous avons documenté le patron abîmé avec des informations indiquant quels liens sont facultatifs, obligatoires ou interdits. Rapportées au graphe, nous complétons la description du graphe représentant le patron abîmé avec un graphe des relations interdites, assimilable aux « Negative Application Conditions » (NAC) des grammaires de graphes [Habel96]. Elles représentent quels sont les arcs interdits sur un graphe. Nous les avons adaptées pour représenter des chemins interdits. La figure 3.18 illustre un graphe des relations interdites du patron abîmé que nous avons utilisé précédemment. Les relations interdites présentées en pointillés mettent bien en valeur le fait que nous n'acceptons aucune association, quel que soit le type des *AssociationEnd*, entre *Composite*, *Composant* et *Feuille*, ni aucun lien d'héritage entre *Composite* et *Feuille*.

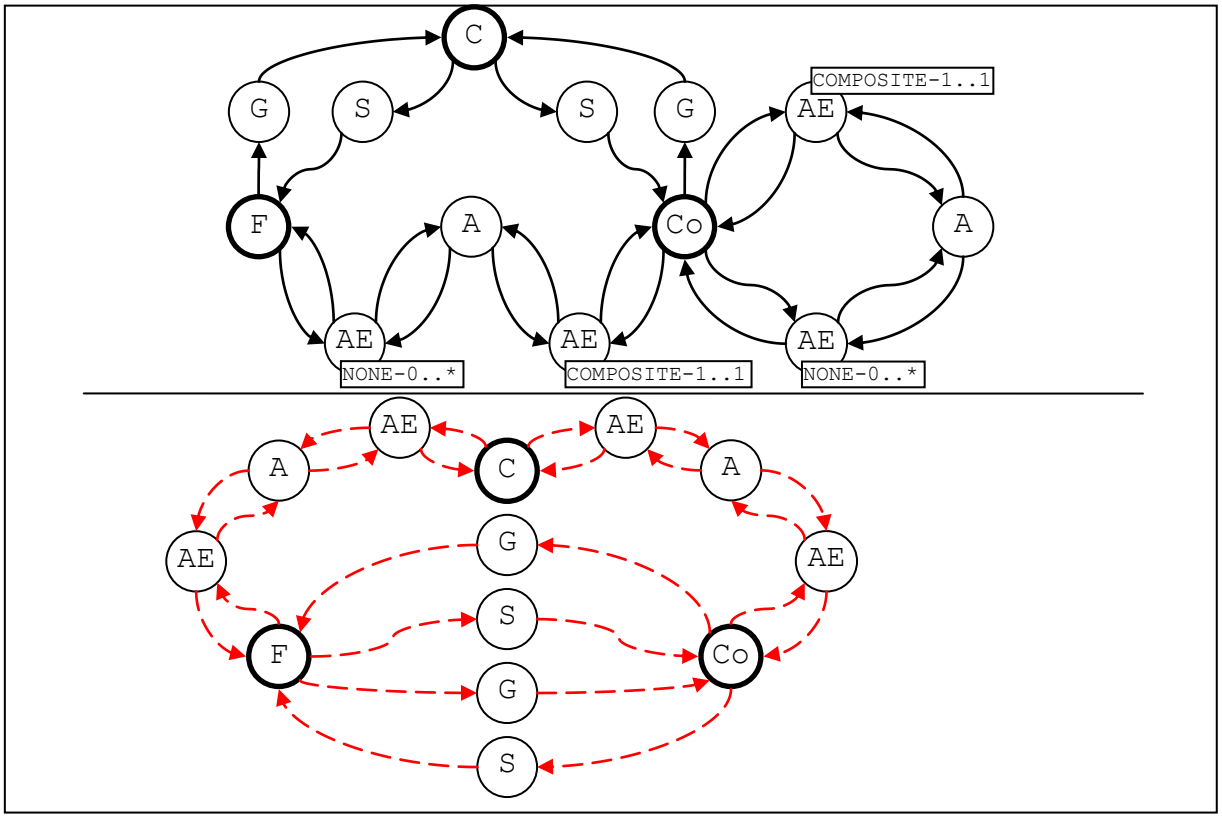


Figure 3.18 : Le graphe du patron abîmé et le graphe des relations interdites associé

Ce graphe représente alors les liens qui invalident la détection du fragment. Pour résumer, si un lien apparaît dans le graphe du patron abîmé, il doit obligatoirement être présent dans le fragment ; s'il apparaît dans le graphe des relations interdites, il doit absolument être absent du fragment. Si un lien est présent dans le fragment, mais absent des deux autres graphes, il est considéré comme facultatif et neutre pour la détection.

Grâce à ce complément, nous pouvons maintenant, en plus de détecter toutes les formes possibles de contextualisation du patron abîmé, refuser certaines formes que nous supposons mauvaises pour l'intention du fragment. Nous espérons ainsi limiter le nombre de fragments présentés au concepteur lors de notre revue de conception.

Afin d'implémenter notre méthode de détection, nous avons défini un algorithme s'inspirant des prédicats cités précédemment.

II.4. Algorithme de détection

Cet algorithme se décompose en trois étapes. À l'issue de chaque étape, un ensemble de classes caractéristiques est sélectionné de l'ensemble issu de l'étape précédente. Ainsi, chaque étape réduit le nombre de classes à traiter jusqu'à ne conserver que les classes appartenant à des fragments alternatifs.

Pour présenter le déroulement de cet algorithme, nous illustrons chaque étape sur l'exemple présenté dans la figure 3.19, où toutes les contextualisations du patron abîmé sont recherchées dans le modèle.

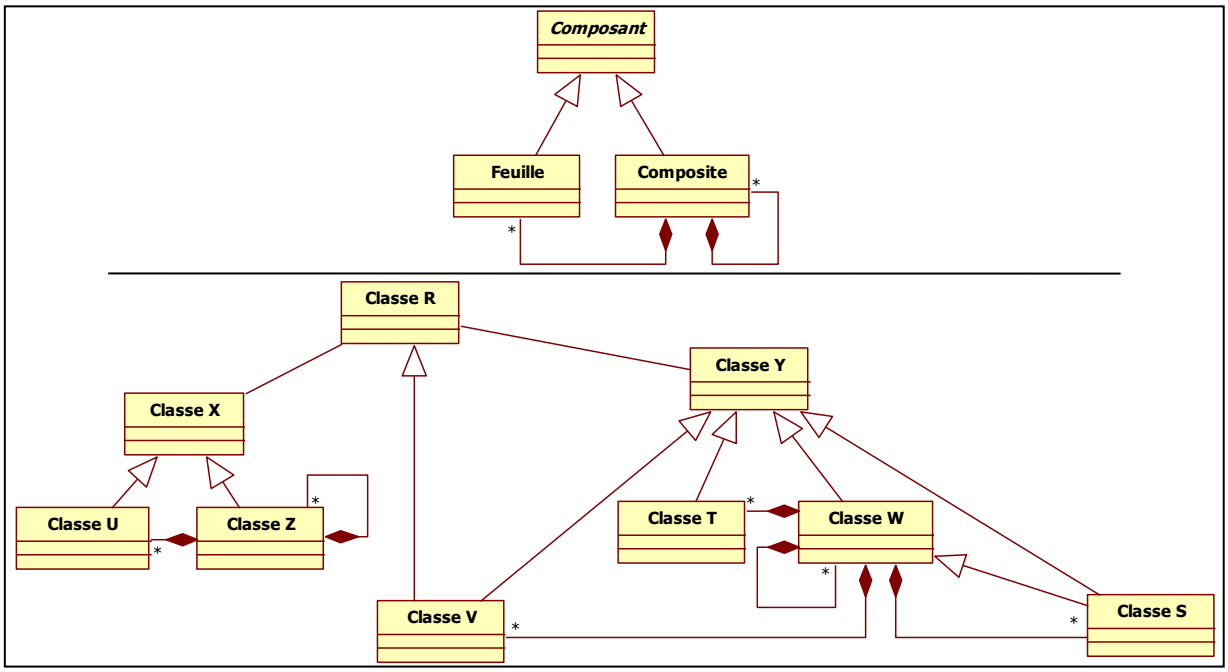


Figure 3.19 : Un patron abîmé et un modèle à analyser

La première étape de l'algorithme consiste à sélectionner l'ensemble des classes vérifiant le prédicat *local_CPR* pour chacun des participants du patron abîmé. Ainsi, chaque classe du modèle est analysée pour vérifier si ses particularités structurelles locales sont conformes aux particularités structurelles locales d'au moins un des participants du patron abîmé. Cette étape construit donc un ensemble de classes pour chaque participant du patron abîmé. Dans notre exemple, cette étape retourne trois ensembles de classes. Le tableau 3.2 présente le résultat de l'algorithme pour l'exemple précédent.

Participant	Classes ayant leurs responsabilités locales concordantes avec le participant
Composant	Classe X, Classe Y
Composite	Classe Z, Classe W
Feuille	Classe U, Classe Z, Classe V, Classe T, Classe W, Classe S

Tableau 3.2 : Résultat de la première étape de l'algorithme de détection

La deuxième étape récupère le résultat de l'étape précédente et cherche à regrouper les classes par fragment alternatif distinct, un par classe de référence, ici *Composite*. Pour mémoire, le participant de référence doit être unique dans un fragment. Ainsi, en vérifiant par quelles relations chaque classe est rattachée au participant de référence, l'algorithme « démêle » les classes sélectionnées précédemment afin de former un ensemble d'ensembles de classes. Chaque ensemble contient une seule classe candidate aux responsabilités du participant de référence et, pour chacun des autres participants, un ensemble de classes reliées à la classe référence par les mêmes relations que dans le patron abîmé.

Dans notre exemple, deux ensembles sont donc créés par cette étape puisqu'il y a deux classes candidates aux responsabilités de *Composite*, la *Classe Z* et la *Classe W*. Lors du démêlage, la *Classe U* et la *Classe X* sont associées au fragment ayant pour référence *Classe Z*. Pour le fragment ayant pour référence *Classe W*, ce sont *Classe Y*, *Classe V*, *Classe T* et *Classe S* qui lui sont associées. En ce qui concerne *Classe Z* et *Classe W*, nous pouvons remarquer dans le tableau 3.2 qu'elles sont également candidates aux responsabilités de *Feuille*, mais qu'elles ne le sont plus dans le tableau 3.3 qui présente le résultat du démêlage. Elles ont été toutes les deux éliminées des ensembles de *Feuille*. Cette élimination à l'étape de démêlage s'explique par le fait que ces deux classes ne sont pas reliées correctement au participant de référence. Dans le cas où une classe ne serait pas reliée correctement à un participant non-référence, le démêlage ne l'aurait pas éliminée, et il aurait fallu attendre l'étape de filtrage des particularités globales.

Fragment	Participant	Classes ayant les mêmes responsabilités locales
1	Composant	Classe X
	Composite	Classe Z
	Feuille	Classe U
2	Composant	Classe Y
	Composite	Classe W
	Feuille	Classe V, Classe T, Classe S

Tableau 3.3 : Résultat de la deuxième étape de l'algorithme de détection

La troisième étape de l'algorithme vérifie si les fragments issus de l'étape précédente sont bien conformes au prédicat *frag_alternatif*. Pour cela, l'algorithme vérifie, pour chacune des classes du fragment, s'il est possible d'atteindre les autres classes avec les mêmes relations que pour les participants correspondants du fragment. Si une classe n'est pas reliée de la même manière, c'est-à-dire qu'elle n'a pas les mêmes particularités structurelles globales que le participant correspondant dans le patron, elle est éliminée du fragment. De plus, si l'algorithme identifie qu'en plus des liens autorisés, une classe est reliée à une autre par un lien supplémentaire, il vérifie si ce lien n'est pas interdit. Le cas échéant, il élimine la classe concernée.

Dans notre exemple, *Classe S* est la fille de *Classe W* qui a les responsabilités de *Composite*. Il est explicitement décrit dans le patron abîmé qu'il ne doit y avoir aucune relation d'héritage entre une *Feuille* et un *Composite*. Cette classe est donc éliminée du fragment. Finalement, la troisième étape fournit les deux fragments alternatifs de notre exemple, comme illustré dans le tableau 3.4.

Fragment	Participant	Classes ayant les mêmes responsabilités locales
1	Composant	Classe X
	Composite	Classe Z
	Feuille	Classe U
2	Composant	Classe Y
	Composite	Classe W
	Feuille	Classe V, Classe T

Tableau 3.4 : Résultat de la troisième étape de l'algorithme de détection

Nous pouvons remarquer que le lien d'héritage entre *Classe V* et *Classe R* n'interfère pas dans l'identification des fragments, puisque *Classe R* ne s'est vue attribuée aucune responsabilité. La figure 3.20 met en exergue les deux fragments identifiés.

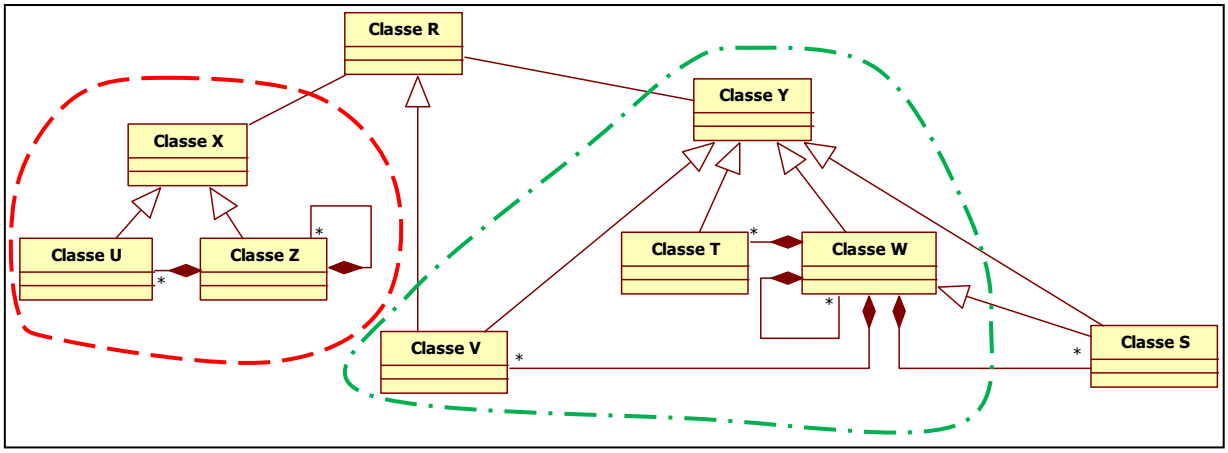


Figure 3.20 : Deux fragments alternatifs identifiés dans le modèle à analyser

En conclusion, notre algorithme permet de détecter de manière détaillée les contextualisations des patrons abîmés. Ces derniers constituant une base génératrice de formes à identifier, nous caractérisons exactement ce qui est recherché, moyennant points d'extensions, relâchement de contraintes et relations interdites. L'utilisation d'OCL nous permet d'éviter de changer d'espace technologique, et ainsi, de ne pas imposer au concepteur de prétraitement sur les modèles avant de lancer l'activité.

II.5. Conséquences de la détection sur la transformation

Les fragments identifiés lors de la détection sont un ensemble de classes ayant chacune une relation directe avec l'un des participants du patron abîmé. Ainsi, nous pouvons dire que chaque classe est marquée par le participant ayant les mêmes responsabilités. De son côté, chaque participant du patron abîmé fait référence à un des participants du patron de conception. Nous sommes donc à même d'identifier aisément les différences structurelles entre les participants d'un patron de conception et ceux de ses patrons abîmés. En analysant ces différences structurelles, nous pouvons déduire les opérations à effectuer sur chaque participant du patron abîmé pour en modifier sa structure de manière à le transformer en patron de conception. La figure 3.21 met en relation un patron de conception et un de ses patrons abîmés.

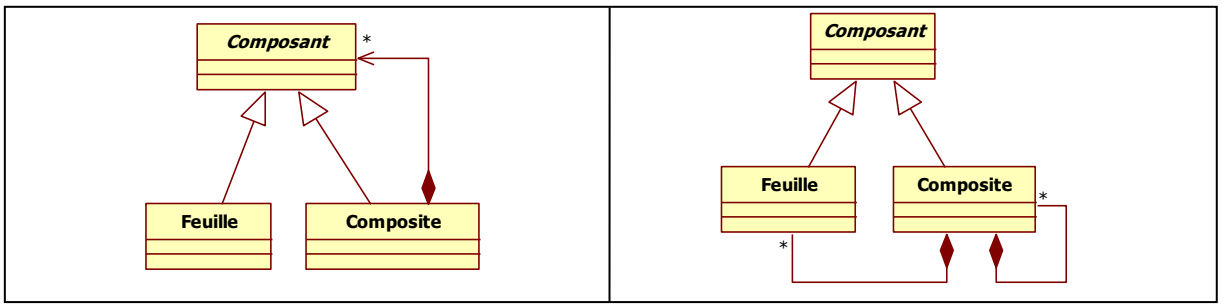


Figure 3.21 : Confrontation du patron *Composite* avec l'un de ses patrons abîmés

D'après l'illustration, nous pouvons dire simplement que pour transformer le patron abîmé à droite en patron de conception à gauche, il suffit de supprimer l'association entre *Composite* et *Feuille*, l'association réflexive sur *Composite* et d'ajouter une relation de composition entre *Composite* et *Composant*. Au niveau du fragment identifié, qui correspond donc à la contextualisation d'un patron abîmé, les instructions précédentes sont applicables en l'état. De manière ensembliste, il est tout à fait envisageable de supprimer, par exemple, les associations entre toutes les classes marquées par les participants *Composite* et *Feuille*.

Grâce à notre technique de détection capable de mapper chaque classe d'un fragment avec les responsabilités des participants du patron abîmé, nous permettons une transformation structurelle du fragment, et ainsi l'injection du patron de conception. Étant limités, pour l'instant, au niveau structurel, nous n'avons pas la possibilité d'impacter les modifications structurelles effectuées sur les méthodes ou les attributs des classes. Nous laissons à la charge du concepteur le soin d'effectuer ces adaptations à l'issue de chaque transformation. De plus amples informations seront fournies dans le chapitre 4.

II.6. Mise en œuvre

Nous avons choisi d'utiliser OCL pour encoder nos requêtes de détection. OCL est un langage de contraintes utilisé pour ajouter de la sémantique aux modèles UML [OMG06]. Il permet de définir des invariants sur des éléments du modèle que le système devra respecter. Ces contraintes n'ont aucun effet de bord sur le modèle et peuvent être utilisées

sur des métamodèles pour définir, en autres, des règles de bonne formation des modèles. Il est ainsi possible d'interdire, par exemple, des cycles dans les liens d'héritage. Les contraintes OCL sont donc des expressions booléennes caractérisant l'ensemble des instances valides exprimées dans un modèle ou un métamodèle.

La plate-forme Neptune a été développée par notre équipe dans le cadre d'un projet européen de même nom [Millan08]. L'interprète OCL qu'elle propose, conforme à la norme OCL 2.0 [OMG06], met en œuvre deux extensions d'OCL [Millan09]. La première permet l'écriture de contraintes ayant un effet de bord sur les modèles afin de pouvoir les transformer, la deuxième concerne les requêtes qui peuvent retourner un résultat de n'importe quel type du métamodèle, ce qui introduit la notion de vue d'un modèle.

Afin de pouvoir profiter des capacités de Neptune, nous avons décidé de détecter les fragments alternatifs en utilisant OCL. Nous souhaitions ainsi décrire des requêtes sous une forme compréhensible par un concepteur, lui permettant éventuellement de les adapter à ses besoins. La deuxième extension de l'interprète de Neptune nous est très utile dans le contexte de notre activité. Nous utilisons la capacité des requêtes à retourner un résultat de n'importe quel type du métamodèle, pour effectuer notre recherche de fragment alternatif dans un modèle. Nous avons associé à chaque patron abîmé une requête OCL retournant un ensemble d'ensembles de classes, chaque ensemble représentant un fragment alternatif composé d'un ensemble de participants. En utilisant ce type de requête sur la plate-forme, nous profitons d'un outil efficace capable de parcourir un modèle en fonction de son métamodèle.

En ce qui concerne l'injection du patron, nous n'avons pas utilisé directement l'extension d'OCL liée aux transformations. Nous avons préféré transformer le modèle directement en mémoire, nous distinguant ainsi de la détection. Ce choix est motivé par le fait que Neptune n'effectue pas de transformation sur place. Sur un gros modèle, ce mode de transformation provoque donc la recopie de toutes les métaclasses. Cette contrainte ne pose pas de problème pour une transformation occasionnelle, mais notre activité pouvant être amenée à transformer plusieurs fois le modèle, le temps d'exécution et la consommation mémoire sont bien trop importants sur des modèles industriels. En transformant en mémoire comme nous le faisons actuellement, nous évitons un grand nombre de recopies.

De plus amples détails seront fournis dans le chapitre 4 de cette thèse.

III. Validation

Pour valider notre algorithme de détection et de substitution de patrons abîmés, nous avons réalisé un jeu de tests, décomposé en deux parties. Notre objectif était de valider notre approche de détection et de transformation des fragments sous la forme la plus simple possible, puis sous une forme prédéfinie. Un fragment avec une forme la plus simple possible est strictement identique aux patrons abîmés sans contextualisation. En ce qui concerne les formes prédéfinies, les fragments contenaient des multiplications, des liens interdits et des attributs facultatifs, afin de constituer des cas de tests aux limites de notre approche. Pour réaliser ces tests, nous avons utilisé notre outil permettant l'exécution de l'activité, ainsi que les requêtes OCL de détection. Cet outil, que nous avons nommé Triton, et les requêtes OCL utilisées sont détaillés dans le chapitre 4 de cette thèse.

Nous avons découpé cette section en deux parties, correspondant chacune à deux types de tests que nous avons effectués : les tests unitaires et les tests aux limites. Dans chacune des parties, nous présentons le mode opératoire suivi pour réaliser les tests, et les résultats obtenus. Durant toute cette section, afin d'améliorer la lisibilité des statistiques, nous avons associé un identifiant à chaque patron abîmé. Le tableau 3.5 présente la relation entre ces identifiants et les patrons abîmés.

Identifiant	Patron de conception	Patron abîmé
C_SP1	Composite	Développement de la composition sur <<Composant>>
C_SP2	Composite	Développement de la composition sur <<Composant>> et sur <<Composite>>
C_SP3	Composite	Composition récursive
C_SP4	Composite	Développement de la composition sur <<Composite>> sans conformité de protocole
C_SP5	Composite	Développement de la composition sur <<Composite>>
C_SP6	Composite	Composition indirecte de la composition sur <<Composite>>
D_SP1	Décorateur	Développement sur <<Décorateur>>
D_SP2	Décorateur	Développement sur <<Décorateur>> sans conformité de protocole
D_SP3	Décorateur	Développement sur <<Composant>>
D_SP5	Décorateur	Mauvais découplage
D_SP6	Décorateur	Mauvais découplage sans possibilité d'extension sur <<Composant>>
D_SP7	Décorateur	Non conformité entre décorateurs et objets à décorer
P_SP1	Pont	Développement du pont

Tableau 3.5 : Table de correspondance des symboles des patrons abîmés

Les patrons abîmés D_SP4, P_SP2 et P_SP3 n'ont pas été intégrés à ce tableau car ils ne sont pas détectables en l'état actuel de notre approche, comme indiqué au chapitre 2.

III.1. Tests unitaires

Nous avons exécuté chacune des requêtes générées sur l'ensemble des patrons abîmés, en utilisant des modèles indépendants, composés uniquement des participants d'un patron abîmé et en respectant son architecture. Ainsi, ces modèles sont strictement identiques aux patrons abîmés sans contextualisation.

III.1.1. Mode opératoire

Chaque classe des modèles testés est nommée par le nom du participant qu'elle représente, ce qui permet de vérifier très rapidement si la détection est correcte. En effet, si le fragment est détecté, cela signifie que la requête est capable de détecter au minimum un fragment strictement identique au patron abîmé. De par le nommage des classes, un fragment correctement identifié doit avoir une correspondance entre le nom de ses classes et le nom des participants candidats. La figure 3.22 illustre le patron abîmé utilisé pour générer une requête de détection, et le modèle contenant le fragment minimal.

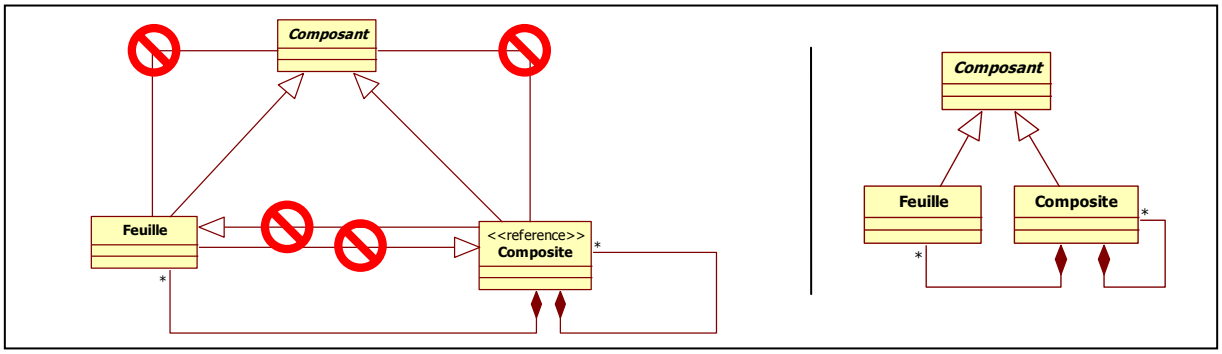


Figure 3.22 : Un patron abîmé et un fragment minimal

Le patron abîmé est présenté avec ses liens interdits, car ils ont bien sûr été utilisés pour générer les requêtes. Il est très visible que le fragment du modèle testé a exactement la même architecture que le patron abîmé.

Pour vérifier si les requêtes de transformation sont au moins capables de transformer un fragment identique au patron abîmé non contextualisé en patron de conception non contextualisé, nous avons continué le test jusqu'à la dernière étape de l'activité. Nous avons donc comparé tous les modèles transformés afin de vérifier s'ils contenaient les mêmes modèles pour un même patron de conception. Par exemple, pour le patron *Composite*, nous avions six modèles différents, et nous devions obtenir après transformation, six modèles contenant chacun le même modèle représentant le patron *Composite*.

III.1.2. Résultats globaux

Nous avons donc exécuté Triton avec treize modèles correspondant aux treize patrons abîmés de notre base. Ainsi, nous avons pu vérifier si chaque requête était capable de retrouver son propre patron abîmé, mais nous avons également vérifié si les autres requêtes détectaient les autres patrons abîmés, étant donnée la proximité structurelle de certains patrons abîmés. Les résultats généraux de ces tests sont présentés dans le tableau 3.6.

	Analyses effectuées avec des modèles contenant individuellement un patron abîmé												
	C_SP1	C_SP2	C_SP3	C_SP4	C_SP5	C_SP6	D_SP1	D_SP2	D_SP3	D_SP5	D_SP6	D_SP7	P_SP1
req C_SP1	D T												
req C_SP2		D T											
req C_SP3		I	D T	I	I								
req C_SP4				D T	I								
req C_SP5				I	D T								
req C_SP6						D T							
req D_SP1							D T						
req D_SP2								D T				I	I
req D_SP3	I								D T	I		I	
req D_SP5	I								I	D T		I	
req D_SP6	I	I^2	I	I	I		I				D T		
req D_SP7	I								I	I		D T	
req P_SP1	I					I	I	I	I	I^2	I	I^2	D T

D : fragment alternatif détecté
I : fragment alternatif détecté de manière incomplète
T : fragment alternatif transformé

Tableau 3.6 : Confrontation des requêtes aux patrons abîmés

Ce tableau contient les résultats des treize modèles analysés par les treize requêtes de détection. Une colonne représente la confrontation d'un modèle à l'ensemble des requêtes, et une ligne la confrontation d'une requête à l'ensemble des modèles.

Le tableau contient trois types de résultats. Chaque type est représenté par une lettre. Ainsi, « D » indique qu'une requête a détecté complètement un fragment : chaque classe est identifiée comme ayant les mêmes caractéristiques structurelles que le participant du même nom. La lettre « T » indique que la transformation a été effectuée sans erreur, c'est-à-dire que le modèle à l'issue de Triton contient la structure du patron de conception.

La lettre « I » indique qu'un fragment a été identifié, mais de manière incomplète. Dans ce cas, certains participants n'ont pas pu être mis en relation avec une classe du modèle. Par exemple, pour un patron abîmé du *Composite*, Triton aurait identifié une classe *Composant*, une classe *Feuille*, mais pas de classe *Composite*. Nous avons paramétré Triton de manière à ce qu'il n'élimine pas ces fragments incomplets et qu'il les présente au concepteur. Bien sûr, nous ne pouvons pas transformer ces fragments, mais nous indiquons au concepteur qu'il peut y avoir un problème particulier.

Enfin, lorsqu'un exposant apparaît à côté de « I », cela signifie que plusieurs fragments ont été identifiés pour le même patron abîmé. Une classe est proposée pour une responsabilité dans un fragment et pour une différente dans un autre fragment.

III.1.3. Analyse des résultats

Il y a deux manières d'analyser le tableau 3.6, par delà le fait que l'intégralité des requêtes détecte au moins le patron abîmé qui a servi à la générer : soit en s'intéressant au nombre de fragments incomplets détectés par les requêtes, soit en vérifiant pourquoi un patron abîmé peut être détecté plusieurs fois de manière incomplète. Les deux manières sont corrélées, car les résultats dépendent, dans les deux cas, des particularités structurelles

de chaque participant. En effet, la requête du patron abîmé n°1 du *Pont* (P_SP1) détecte de manière incomplète la totalité des patrons abîmés du *Décorateur* et une partie des patrons abîmés du *Composite*. Cela est dû au fait que l'intersection des particularités structurelles des participants des patrons détectés de manière incomplète et de ceux du patron recherché par la requête est non vide.

Il est également possible de remarquer que quatre requêtes différentes détectent de manière incomplète le patron abîmé n°7 du *Décorateur* (D_SP7). La raison en est la même que pour le cas précédent. Tout ceci montre que les particularités structurelles d'un participant ne sont pas suffisantes pour décrire correctement un patron abîmé. C'est l'ensemble des particularités de tous les participants d'un patron abîmé qui permet de le décrire fidèlement. Pour visualiser ce phénomène, analysons les particularités structurelles du patron abîmé n°1 du *Pont* illustré dans la figure 3.23 et dans le tableau 3.7.

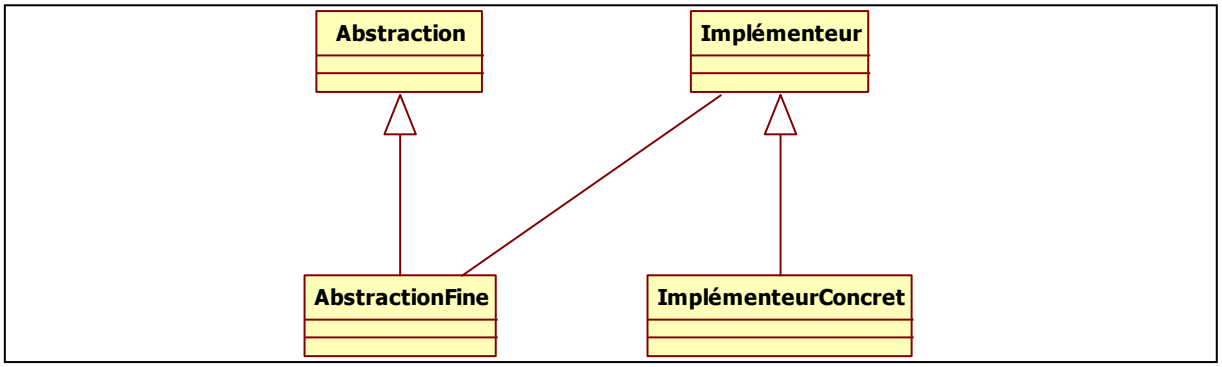


Figure 3.23 : Le patron abîmé n°1 du *Pont*

Participant de référence	Implémenteur	
Particularités locales	Implémenteur	Classe avec au moins une fille et une association.
	Abstraction	Classe avec au moins une fille.
	Abstraction fine	Classe avec au moins un lien d'héritage et une association.
	Implémenteur concret	Classe avec au moins un lien d'héritage.
Particularités globales	Abstraction	Super-classe d'*AbstractionFine*.
	Abstraction fine	Sous-classe d'*Abstraction* rattachée au participant de référence.
	Implémenteur concret	Sous-classe du participant de référence.

Tableau 3.7 : Particularités structurelles du patron abîmé n°1 du *Pont*

Pour mémoire, rappelons que le participant de référence, qui représente la classe ayant un rôle prépondérant dans le patron, n'a pas de particularité structurelle globale. Ainsi, seule la concordance des particularités locales est suffisante pour qu'une classe soit candidate aux responsabilités du participant de référence. Dans notre cas, toutes les *classes* « *avec au moins une fille et une association* » sont détectées.

De plus, comme l'association n'a pas d'attribut de restriction (navigabilité, agrégation, multiplicité), n'importe quelle association est validée par le détecteur. Une fois le participant de référence détecté, les autres participants peuvent apparaître dans le fragment si leur relation avec le participant de référence n'est pas complexe et qu'ils n'ont pas de relation avec les autres participants. Dans notre cas, le participant *Implémenteur concret* n'est pas relié à un autre participant, et la relation avec le participant de référence n'est qu'un simple lien d'héritage. À l'inverse, le participant *Abstraction Fine* est relié à un autre participant, ce qui limite sa probabilité d'être détecté. Ainsi, ce patron abîmé est détecté de manière incomplète à chaque fois qu'une classe est associée à une autre classe et hérite d'une troisième.

Prenons un modèle contenant le patron abîmé n°7 du *Décorateur*. Ce patron abîmé est présenté dans la figure 3.24 avec deux fragments incomplets détectés par la requête *req P_SP1*.

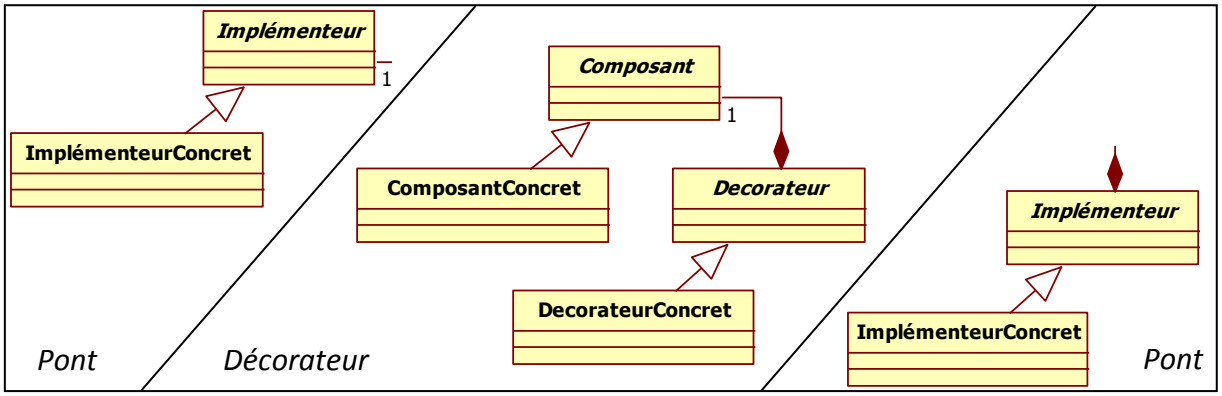


Figure 3.24 : Le patron abîmé n°7 du *Décorateur* et deux fragments incomplets du *Pont*

La structure du patron abîmé n°7 du *Décorateur* intègre bien une classe « *avec au moins une fille et une association* ». En effet, même si l'association entre *Décorateur* et *Composant* est une composition, la requête du patron abîmé n°1 du *Pont* recherche toutes les classes avec au moins une fille et une association, quelles que soient ses particularités. Le fait que ce soit une *agrégation* n'est pas vérifié par la requête. Ainsi, la classe *Composant* a les mêmes particularités structurelles que le participant *Implémenteur* du pont. La classe est donc sélectionnée. Et comme la requête recherche les filles de l'implémenteur, la classe *DécorateurConcret* est assignée aux responsabilités de *ImplémenteurConcret*. Nous avons donc ici un fragment incomplet. De plus, la structure est identique aux classes *Composant* et *ComposantConcret*. Le participant *Implémenteur* étant le participant de référence, la requête retourne deux fragments. Ainsi, lorsque la requête du patron abîmé n°1 du *Pont* est exécutée sur le patron abîmé n°7 du *Décorateur*, elle retourne deux fragments incomplets présentés dans le tableau 3.8.

Fragment	Participant	Classes candidates
P_SP1 - 1	*Implémenteur*	Composant
	Implémenteur concret	Composant concret
	Abstraction	
	Abstraction fine	
P_SP2 - 2	*Implémenteur*	Décorateur
	Implémenteur concret	Décorateur concret
	Abstraction	
	Abstraction fine	

Tableau 3.8 : Résultat de l'exécution de la requête P_SP1 sur le patron abîmé D_SP7

La détection de fragments incomplets d'une requête n'est pas la résultante d'un défaut, mais d'une zone du modèle ayant une structure proche de celle recherchée. Le fait qu'une requête ramène plus ou moins de fragments incomplets est donc directement lié à la complexité des particularités structurelles qu'elle recherche. À l'issue de ce jeu de tests, nous concluons que toutes les requêtes que nous avons générées sont capables de détecter au moins le fragment correspondant exactement au patron abîmé correspondant.

III.2. Tests aux limites

Afin de vérifier les requêtes de détection et de transformation sur toutes les formes possibles de contextualisation des patrons abîmés, nous les avons confrontées à un ensemble de modèles particuliers. Nous n'avons pas cherché à donner une sémantique à ces modèles, mais nous avons mis à l'épreuve les requêtes sur les liens interdits ou les multiplications.

III.2.1. Mode opératoire

Nous avons produit un ensemble de modèles structurellement identiques à chaque patron de conception, en y intégrant des multiplications de participants, ainsi que des liens interdits et facultatifs. Théoriquement, cet ensemble de tests devrait être infini, puisqu'il existe un nombre infini de multiplications possibles de participants et tout autant de liens facultatifs. Cependant, de par sa spécificité, une requête qui détecte un fragment avec un participant multiplié une fois est capable de le détecter avec un participant multiplié *n* fois. En effet, la multiplication est détectée grâce à des comparateurs numériques (« *toutes les classes avec au moins une agrégation* » devient « *aggregation >= 1* »). Pour les liens facultatifs, le fait que les requêtes fonctionnent par filtrage, c'est-à-dire qu'elles vérifient que les classes aient au moins les liens obligatoires, mais aucun lien interdit, permet d'autoriser toutes les formes possibles de liens facultatifs.

De la même manière, les requêtes de transformation fonctionnant sur des ensembles de classes candidates, la modification est effective qu'il y ait une ou plusieurs classes dans les ensembles. De plus, comme un fragment contient au moins tous les liens obligatoires, Triton ne modifie aucun lien supplémentaire entre les classes du fragment identifié. Il en est de même pour les relations liant les classes appartenant au fragment et les classes extérieures au fragment.

Nous avons donc restreint notre jeu de tests à un seul type de cas particulier pour chaque patron abîmé. Nous avons testé les requêtes sur des modèles ayant une multiplication significative, un lien ou un attribut facultatif, un ou plusieurs liens interdits et des agencements de patrons particuliers.

III.2.2. Analyse de modèles sélectionnés

Chacun des cas présentés dans cette partie a été sélectionné parce qu'il présente une particularité permettant de mettre en valeur un paramètre de notre approche.

III.2.2.1. Relations entre deux participants multipliés

Notre premier modèle particulier, présenté dans la figure 3.25, contient le patron abîmé n°5 du *Composite* mentionné à la figure 3.22, avec deux classes *Feuille* et deux classes *Composite*.

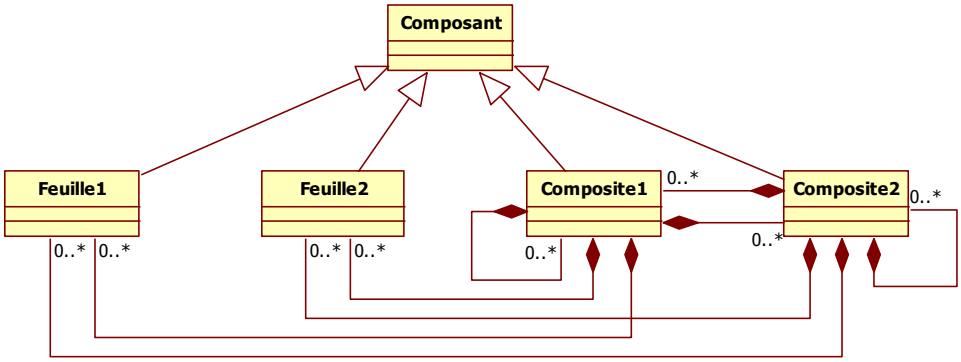


Figure 3.25 : Cas particulier n°1 du C_SP5

Le participant de référence de ce patron abîmé est le *Composite*. Ce modèle ne contient pas de lien interdit. Les deux compositions entre les classes *Composite* ont été ajoutées pour donner une intention de composition récursive au modèle. Pour que chacun des composites puisse gérer idéalement la composition, il doit avoir connaissance de toutes les classes composables. Dans ce cas, il s'agit de chacune des feuilles et également, de chacun des composites. D'après le patron abîmé non contextualisé, une classe *Composite* doit se composer d'elle-même, donc, d'un *Composite*, et par extension de tous les composites du modèle. Ainsi, Triton doit, à l'issue de l'analyse, présenter deux fragments alternatifs possibles contenant chacun une des deux classes *Composite1* ou *Composite2*. Le résultat de la recherche est présenté dans le tableau 3.9.

Fragment	Participant	Classes candidates
C_SP5 - 1	*Composite*	Composite1
	Composant	Composant
	Feuille	Feuille1, Feuille2, Composite2
C_SP5 - 2	*Composite*	Composite2
	Composant	Composant
	Feuille	Feuille1, Feuille2, Composite1

Tableau 3.9 : Résultat de la détection du cas particulier n°1 du C_SP5

La particularité de ce modèle se situe dans les classes candidates aux responsabilités de *Feuille*. En effet, la classe composite qui n'a pas les responsabilités de *Composite* devient une feuille d'après le résultat de la requête.

Prenons pour exemple, le fragment 1. La classe *Composite1* a les responsabilités de *Composite*. Ainsi, *Composite2* ne peut pas avoir les mêmes responsabilités puisque ce participant est référence. Les particularités structurelles d'une *Feuille*, sont pour ce patron abîmé « *toutes les classes filles de Composant et agrégés à Composite* ». La classe *Composite2* est bien fille de *Composant* et agrégée à *Composite1*. Elle peut donc avoir les responsabilités de *Feuille*, puisque rien n'interdit une *Feuille* de se composer récursivement, ni d'être composée d'autres feuilles.

Cependant, cette dernière affirmation peut être sujette à discussion. Dans une architecture représentant un arbre de composition hiérarchique, un élément terminal est en bout d'arbre et n'est donc nœud d'aucun sous arbre. Pour éviter ce cas, il suffit d'exprimer, dans le patron abîmé, grâce au profil, qu'une *Feuille* ne peut pas se composer elle-même. En ajoutant cette contrainte et en exécutant à nouveau le test, la classe *Composite* non sélectionnée aux responsabilités de *Composite* est éliminée du fragment, comme le montre le tableau 3.10.

Fragment	Participant	Classes candidates
C_SP5 - 1	*Composite*	Composite1
	Composant	Composant
	Feuille	Feuille1, Feuille2
C_SP5 - 2	*Composite*	Composite2
	Composant	Composant
	Feuille	Feuille1, Feuille2

Tableau 3.10 : Résultat de la détection du cas particulier n°1 du C_SP5 après adaptation du C_SP5

Les résultats obtenus sont maintenant en accord avec les responsabilités de chaque classe.

III.2.2.2.Ordre d'exécution des transformations

Le modèle illustré dans la figure 3.26 contient une contextualisation du patron abîmé C_SP5, avec un lien interdit entre *Feuille3* et *Composite*.

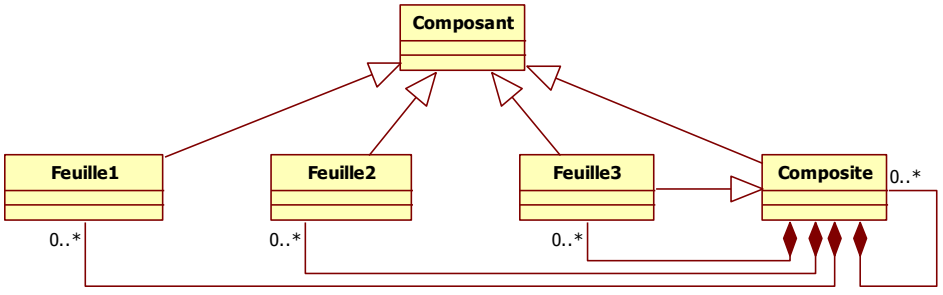


Figure 3.26 : Cas particulier n° 2 du C_SP5

Le résultat de la recherche sur ce modèle est présenté dans le tableau 3.11.

Fragment	Participant	Classes candidates
C_SP5 - 1	*Composite*	Composite
	Composant	Composant
	Feuille	Feuille1, Feuille2
C_SP1 - 2	*Composite*	Feuille3
	Composant	Composite
	Feuille	Feuille1, Feuille2

Tableau 3.11 : Résultat de la détection du cas particulier n°2 du C_SP5

Triton a détecté dans ce modèle, deux fragments pour deux patrons abîmés différents. Le premier est celui que nous attendions, un fragment du C_SP5 n'incluant pas la classe *Feuille3* puisqu'il a un lien interdit, et le second est un fragment du C_SP1 n'incluant pas la classe *Composant*. Le patron abîmé C_SP1 est présenté dans la figure 3.27.

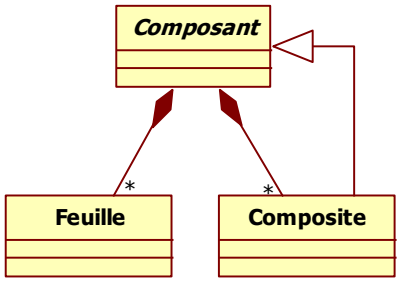


Figure 3.27 : Le patron abîmé n°1 du *Composite* (C_SP1)

L'ajout du lien interdit a provoqué l'apparition de deux fragments imbriqués. Les conséquences principales de cette imbrication apparaissent lors de la transformation. En effet, le remplacement de l'un des fragments provoque la disparition de l'autre. L'importance de la validation de l'intention par le concepteur prend ici toute son importance.

Pour approfondir l'illustration de l'ordre d'exécution des transformations, reprenons l'exemple du paragraphe précédent présenté à la figure 3.25. Nous choisissons de remplacer le fragment 1 du tableau 3.10 par la contextualisation du patron de conception *Composite*. Nous pouvons remarquer dans le tableau 3.9 que la classe *Composite2* est hors fragment et n'est donc pas concernée par la transformation. Ainsi, après transformation du fragment 1, nous obtenons la figure 3.28.

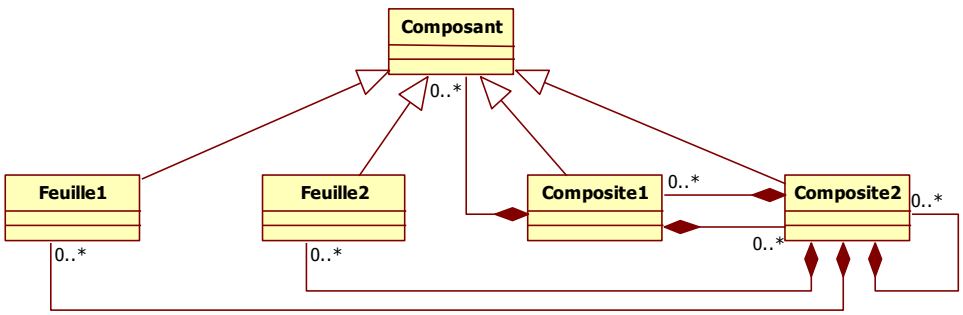


Figure 3.28 : Cas particulier n°1 du C_SP5 après la transformation du fragment 1

Le patron a été correctement injecté sur la base de la classe *Composite1*. Tous les liens de composition vers les feuilles ont été supprimés, de même que la composition récursive, et un nouveau lien a été ajouté entre *Composite1* et *Composant*. Il est très visible que la classe *Composite2* a été ignorée. À l'issue de cette transformation, Triton invite le concepteur à relancer la détection, et le résultat présenté dans le tableau 3.12 est obtenu.

Fragment	Participant	Classes candidates
C_SP5 - 1	*Composite*	Composite2
	Composant	Composant
	Feuille	Feuille1, Feuille2

Tableau 3.12 : Résultat de la détection du cas particulier n°1 du C_SP5 après correction du C_SP5

L'ancien fragment est à nouveau détecté, mais cette fois en ignorant la classe *Composite1* qui ne peut être ni un *Composite* ni une *Feuille* puisqu'elle est reliée par un lien interdit à *Composant*. Après exécution de la transformation, le concepteur obtient le modèle présenté dans la figure 3.29.

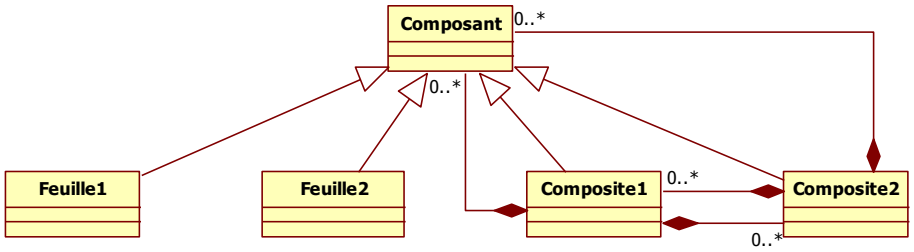


Figure 3.29 : Cas particulier n°1 du C_SP5 après la transformation des deux fragments

Le modèle obtenu à l'issue de la transformation montre un point intéressant. Les deux fragments ont été corrigés et transformés, et nous obtenons un patron *Composite* respectant la structure du GoF.

Il est donc important que les transformations soient effectuées dans un ordre précis, et que chaque transformation soit suivie d'une nouvelle détection. Dans notre premier exemple, la transformation d'un fragment a provoqué la disparition d'un autre fragment, dans le deuxième, l'ordre des transformations n'a pas d'importance, mais la succession des détections a permis l'apparition d'un nouveau fragment.

Pour aller plus loin, nous avons décidé de nous attarder sur le cas particulier présenté à la figure 3.29. En effet, nous considérons que pour être efficace, il faudrait factoriser au maximum les relations de composition au niveau des classes *Composite1* et *Composite2*. Certes, le GoF ne donnant aucune instruction sur la manière de gérer la multiplication d'un des participants, ce modèle est structurellement en accord avec le GoF. Cependant, il est clair que les deux liens de composition entre les deux *Composite* sont redondants avec la factorisation du lien sur *Composant*. A priori, il serait correct de les supprimer pour obtenir le modèle présenté dans la figure 3.30.

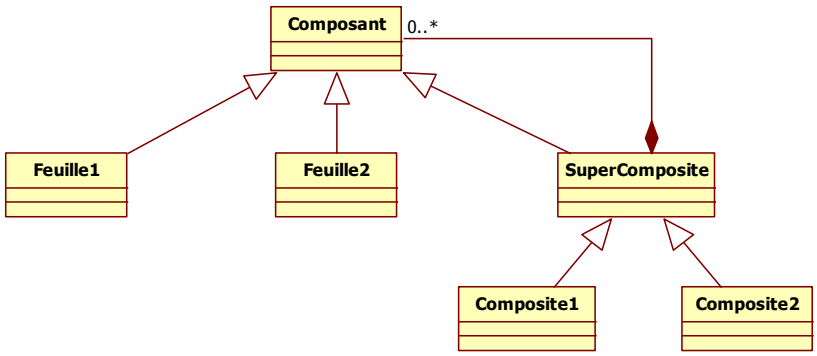


Figure 3.30 : Cas particulier n°1 du C_SP5 après une transformation supplémentaire

Nous considérons que cette configuration constitue la réelle contextualisation du patron *Composite*. Il est intéressant de remarquer qu'il est nécessaire dans ce cas d'effectuer deux transformations successives pour aboutir à un nouveau fragment. Ce résultat nous conforte dans l'idée que nous avons à inviter le concepteur à relancer la détection après chaque transformation pour parer à toute éventualité.

III.2.2.3.Gestion ensembliste des liens interdits

Nous présentons le modèle de la figure 3.31, pour mettre en avant une limite de notre approche qui concerne la détection par ensemble.

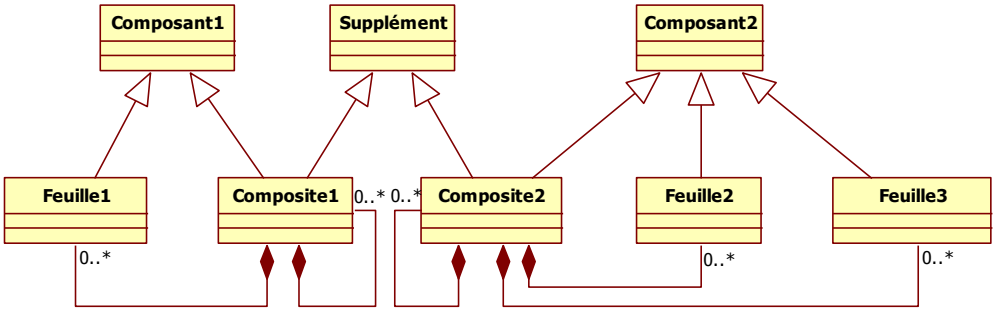


Figure 3.31 : Cas particulier n°3 du C_SP5

Pour ce modèle, nous avons simplement couplé deux fragments du C_SP5 par une classe *Supplément*. Le résultat de la détection, présenté dans le tableau 3.13 correspond à nos attentes.

Fragment	Participant	Classes candidates
C_SP5 - 1	*Composite*	Composite1
	Composant	Composant1
	Feuille	Feuille1
C_SP5 - 2	*Composite*	Composite2
	Composant	Composant2
	Feuille	Feuille2, Feuille3

Tableau 3.13 : Résultat de la détection du cas particulier n°3 du C_SP5

Triton a correctement détecté dans ce modèle les deux fragments du C_SP5. Les classes correspondent aux participants associés, et les deux fragments sont correctement démêlés. Ainsi, chaque classe est correctement reliée à son participant de référence.

Cependant, d'après les particularités structurelles de ce modèle, nous aurions dû obtenir les résultats présentés dans le tableau 3.14, où deux fragments supplémentaires incompatibles avec les deux premiers apparaissent. Ces fragments concernent le patron abîmé n°4 du *Composite*, présenté dans la figure 3.32

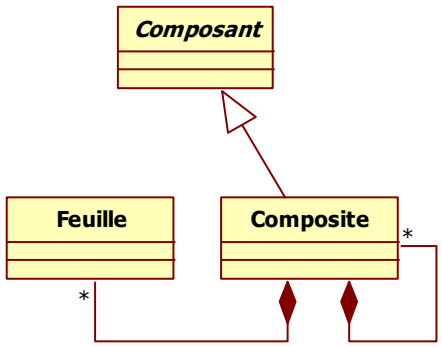


Figure 3.32 : Le patron abîmé n°4 du *Composite* (C_SP4)

Fragment	Participant	Classes candidates
C_SP5 - 1	*Composite*	Composite1
	Composant	Composant1
	Feuille	Feuille1
C_SP5 - 2	*Composite*	Composite2
	Composant	Composant2
	Feuille	Feuille2, Feuille3
C_SP4 - 3	*Composite*	Composite1
	Composant	Supplément
	Feuille	Feuille1
C_SP4 - 4	*Composite*	Composite2
	Composant	Supplément
	Feuille	Feuille2, Feuille3

Tableau 3.14 : Résultat théorique de la détection du cas particulier n°3 du C_SP5

Les particularités structurelles du modèle étant compatibles, Triton a bien détecté les fragments 3 et 4, mais de manière incomplète. Il n'a pas trouvé de classes candidates aux responsabilités de *Feuille*. Ceci s'explique par le fait qu'il est interdit pour une *Feuille* du C_SP4 d'hériter de *Composant*. Or, les classes *Composant1*, *Composant2* et *Supplément* ont toutes les trois des particularités structurelles compatibles avec le participant *Composant*. Comme il existe, par exemple, un lien interdit entre *Feuille1* et *Composant1*, les deux classes sont éliminées de leurs ensembles respectifs. Il en est de même pour *Feuille2* et *Feuille3* avec *Composant2*. Ainsi, au final, il ne reste plus que la classe *Supplément* dans l'ensemble des candidats à *Composant*, et plus aucune dans l'ensemble des feuilles.

Comme nous travaillons sur des ensembles avec OCL, nous n'avons pas la possibilité, sans augmenter la complexité de la requête, de traiter le contenu des ensembles de participants un par un. À partir du moment où une relation est interdite, c'est l'ensemble des classes candidates qui est éliminé.

IV. Conclusion

De nombreuses méthodes de détection existent pour détecter des patrons de conception dans des modèles. L'objectif est d'aider les concepteurs à comprendre et à maintenir des modèles, en identifiant des fragments significatifs caractérisés par des patrons. Même si nous recherchons des fragments significatifs dans des modèles, la finalité de notre étude n'est pas la même. Notre objectif est l'amélioration des modèles en y détectant des défauts de conception, et en les corrigeant. De plus, comme la base des patrons abîmés n'est pas figée, et que d'autres patrons peuvent apparaître dès qu'une solution alternative est identifiée, la méthode que nous devons utiliser doit permettre de rajouter un objet à détecter facilement. Enfin, comme nous cherchons à améliorer les modèles avant qu'ils ne soient implémentés, nous ne pouvons pas nous appuyer sur une méthode utilisant des informations issues des codes sources.

Nous avons défini et analysé dans ce chapitre une méthode de détection en deux étapes principales opérant uniquement au niveau modèle. La première étape consiste à rechercher toutes les classes étant localement identiques aux participants du patron, la deuxième vérifiant que ces classes sont correctement reliées entre-elles. Les patrons abîmés et les modèles à analyser sont comparés sous forme de graphes, composés de sommets représentant les classes, et de sommets représentant tout type de relations entre-elles. Chaque sommet étant typé par un élément du métamodèle, nous avons défini une relation d'équivalence nous permettant de les comparer. Grâce à cette technique, tous les fragments du modèle analysé, structurellement comparables aux patrons abîmés, sont identifiés. Chaque fragment est composé d'un ensemble de classes dont les responsabilités ont été associées aux responsabilités des participants du patron abîmé. Cet appariement nous a permis de définir une méthode de transformation structurelle simple et efficace, qui permet de remplacer tout fragment alternatif identifié par la contextualisation adéquate du patron de conception.

4

Concrétisation de l’approche

L'objectif de nos travaux est de détecter, dans des modèles UML, des fragments caractéristiques de mauvaises pratiques de conception. Après vérification et confirmation auprès du concepteur, nous visons à remplacer automatiquement ces fragments par les pratiques de conception adéquates. Afin d'encapsuler nos différents concepts dans un cadre méthodologique approprié, nous avons défini une activité de revue de conception itérative en trois étapes. La première détecte les fragments alternatifs d'un modèle, la deuxième présente ces fragments au concepteur pour qu'il vérifie leur intention et qu'il prenne conscience des défauts engendrés, puis la dernière étape transforme le modèle analysé pour substituer le patron de conception adéquat au fragment validé par le concepteur. A l'issue de la transformation, une nouvelle itération est proposée au concepteur. La figure 4.1 illustre une itération de cette activité dont les concepts ont été présentés dans le chapitre 1 [Bouhours07_a] [Bouhours08].

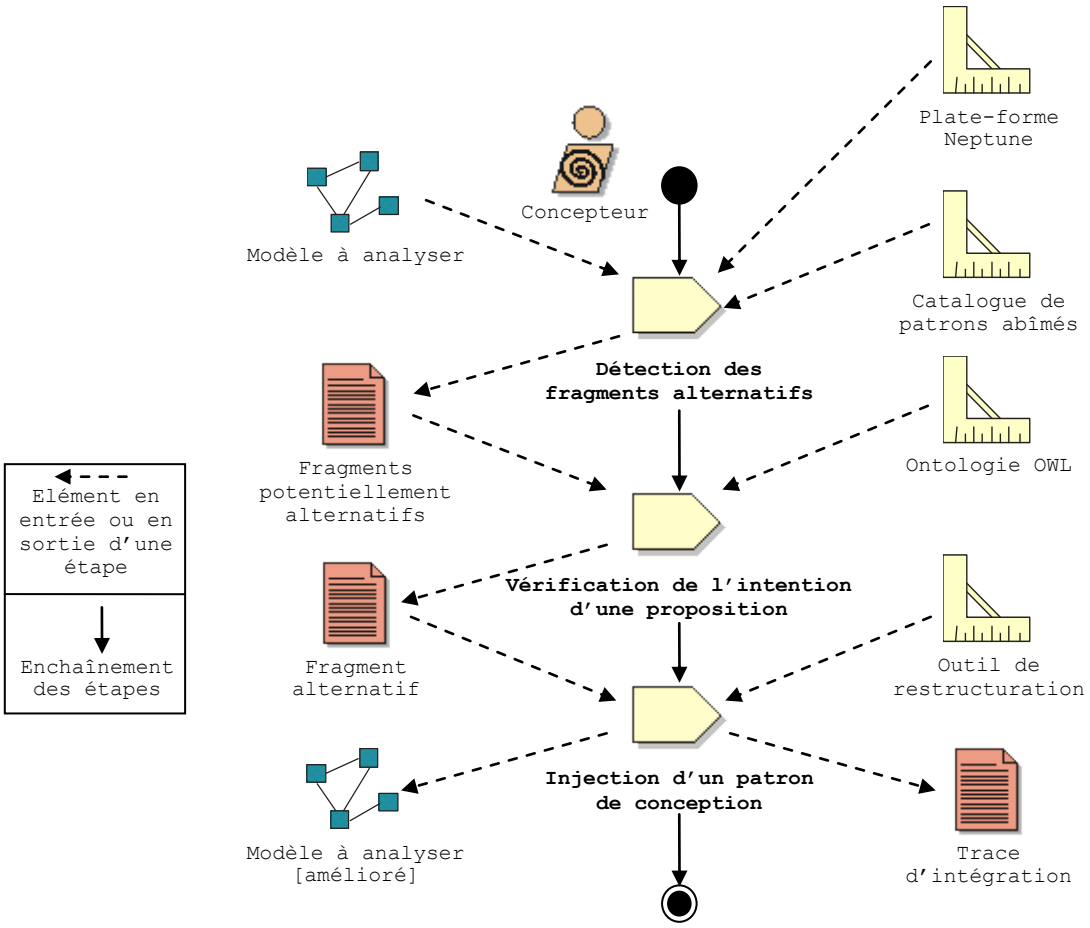


Figure 4.1 : Une itération de notre activité de revue de conception

Chaque fragment détecté représente une contextualisation possible d'un des patrons abîmés de notre catalogue. Ainsi, afin d'automatiser la détection de toutes les contextualisations de tous les patrons abîmés, nous avons doté ces derniers d'une requête capable de les identifier dans un modèle. Nous avons mis en relation nos concepts d'intention et de points forts avec nos concepts de patrons abîmés au sein d'une ontologie OWL, afin de communiquer avec le concepteur, une fois la détection terminée. Ce dernier a alors la possibilité de valider l'intention des fragments identifiés, afin que nous les remplacions par la contextualisation du patron de conception adéquat. Pour ce faire, et de manière à être compatible avec toutes les contextualisations possibles, nous générons automatiquement un ensemble d'opérations de transformation à partir des différences structurelles existantes entre un patron abîmé et son patron de conception.

Nous débutons ce chapitre par la présentation du générateur de requêtes. Grâce à un système générique, les requêtes de détection et de transformation de n'importe quel patron abîmé sont générées automatiquement. Ainsi, la complexité de ces requêtes ne constitue pas un frein à la détection et à la transformation des patrons abîmés. Nous détaillons ensuite l'ontologie que nous avons utilisée pour mettre en relation patrons abîmés, points forts et intentions. Nous montrons que cette ontologie nous permet de construire les questions posées au concepteur à l'issue de la détection. Nous terminons ce chapitre en analysant notre approche sur des modèles industriels. Ces tests ont été réalisés grâce à Triton, l'outil que nous avons développé pour supporter l'intégralité de notre activité de revue de conception.

I. Génération des requêtes

Le cœur de notre activité étant fondé sur la détection et la transformation de patrons abîmés, nous avons associé à chaque patron abîmé une requête OCL de détection et un ensemble d'opérations de transformation. La requête de détection est capable de retrouver toutes les contextualisations possibles du patron abîmé auquel elle est attachée. Grâce aux particularités de la plate-forme Neptune [Millan08] [Millan09], que nous utilisons en support de notre activité, nous pouvons proposer des requêtes OCL capables de retourner un résultat de n'importe quel type du métamodèle. Ainsi, en exécutant les requêtes de chaque patron abîmé sur un modèle, nous identifions les fragments alternatifs que nous pouvons remplacer par les patrons de conception correspondants. L'originalité de notre approche réside dans un outil que nous avons développé et qui permet de générer automatiquement ces requêtes à partir de la description détaillée du patron abîmé, c'est-à-dire en incluant des informations (liens interdits, autorisés, facultatifs…) issues d'un profil que nous avons défini.

Dans cette section, nous débutons en présentant le profil UML que nous avons utilisé pour décrire plus finement les patrons abîmés. En effet, c'est grâce à un profil dédié que nous avons pu décrire les liens interdits et les éléments facultatifs d'un patron abîmé. Ce profil est utilisé par notre générateur de requêtes de détection que nous décrivons dans un deuxième temps. Nous montrons comment il procède pour construire les requêtes quel que soit le patron abîmé fourni. Nous présentons ensuite la manière dont le générateur déduit les opérations permettant de remplacer un fragment alternatif par le patron de conception adéquat, ainsi que l'algorithme sous-jacent aux opérations de transformation.

I.1. Description des liens autorisés, interdits et facultatifs

De manière à ce que le générateur puisse produire des requêtes conformes à notre méthode de détection, il est nécessaire de prendre en compte le patron abîmé, avec le participant de référence, ainsi que ses liens interdits. Pour ce faire, nous avons ajouté directement au modèle UML du patron abîmé, les informations traduisant les liens interdits sous forme de profil UML. Un profil UML est un mécanisme qui permet d'étendre le métamodèle UML afin d'y ajouter des éléments spécifiques à un domaine. Nous avons donc choisi d'utiliser un profil UML pour affiner la sémantique de chaque élément structurel du patron abîmé.

Comme précisé dans le chapitre précédent, les relations obligatoires et interdites sont exprimées directement dans le patron abîmé. À l'inverse, les relations facultatives sont caractérisées par toutes les relations non renseignées dans le patron abîmé. Cependant, profitant des possibilités offertes par l'utilisation d'un profil, nous avons également ajouté des informations exprimant l'état facultatif de certains éléments. Par exemple, le fait qu'une relation de composition soit mononavigable ou binavigable peut, dans certains cas, présenter une contrainte forte dans la détection : le fragment pourrait être valide que la relation soit mono ou bi navigable. Dans ces cas, la navigabilité de la relation devient facultative. Or, il n'est pas possible, en UML, de ne pas renseigner cet attribut dans la métaclasse *AssociationEnd*, empêchant ainsi d'appliquer la règle qu'une relation facultative doit être omise du patron abîmé. Nous avons donc défini des stéréotypes supplémentaires dans le profil pour exprimer le caractère facultatif de certains éléments. Le tableau 4.1 résume les quatre stéréotypes qui nous servent à définir les relations interdites, les attributs facultatifs, et quel participant du patron abîmé doit être considérée comme participant de référence.

Stéréotype	Métaclasse	Définition
reference	Classifier	Cette classe représente le participant de référence. Celui-ci ne pourra être présent qu'une seule fois dans le fragment détecté. Ce stéréotype est obligatoire dans la description du patron abîmé.
*[nom d'un attribut]*_nevermind	Toutes	*[nom d'un attribut]* doit être remplacé par le nom d'un attribut de la métaclasse concernée. Dans ce cas, cet attribut est considéré comme facultatif lors de la détection. Ainsi, le fait qu'il soit présent ou non dans le fragment n'affectera pas sa détection.
role_never	Generalization	La généralisation concernée par ce stéréotype est interdite. Ainsi, lors de la détection, si cette généralisation est présente, la classe fille est éliminée de l'ensemble des candidates. La classe mère n'est pas concernée puisqu'elle est reliée à la classe fille par la métaclasse *Specialization* qui n'est pas stéréotypée.
role_never	AssociationEnd	L'*Association* attachée à l'*AssociationEnd* stéréotypée est interdite. Les deux *AssociationEnd* de l'*Association* doivent être stéréotypées avec le même stéréotype. Ainsi, lors de la détection, si une association est présente (quels que soient les attributs des *AssociationEnd*), les classes rattachées seront considérées comme non-candidates.

Tableau 4.1 : Les stéréotypes applicables à un patron abîmé

Le stéréotype *reference* doit être présent une et une seule fois dans un patron abîmé. Sa présence est recherchée par le générateur et utilisée pour construire les requêtes. De plus, étant donné que les fragments ne doivent contenir qu'une seule classe ayant les responsabilités du participant de référence, sa présence explicite dans le patron abîmé est utilisée pour construire la requête de manière à gérer l'unicité du participant dans le fragment.

Le stéréotype *[nom d'un attribut]_nevermind* ajoute de la souplesse à la détection, en précisant quel attribut de la métaclasse est facultatif. De plus, grâce à sa généricité, il peut être appliqué à n'importe quelle métaclasse. Par exemple, pour préciser que la navigabilité d'une relation est facultative, il suffit de stéréotyper la métaclasse *AssociationEnd* avec le stéréotype *isNavigable_nevermind*. Dès lors, la requête générée n'intègrera aucune contrainte sur l'attribut *isNavigable* de cette *AssociationEnd*. Ce stéréotype assouplit ainsi les contraintes de détection. Il est possible, en changeant le nom de l'attribut, de le mentionner plusieurs fois sur une même *AssociationEnd*, et d'utiliser le mot clef « *all* » pour que tous les attributs de la métaclasse soient considérés comme facultatifs. Ceci est très utile pour traduire les contraintes de rattachement d'une classe comme par exemple dans l'énoncé « cette classe doit être rattachée à une autre, mais peu importe comment ».

Le stéréotype *role_never* représente l'interdiction d'un lien entre classes. Dans le cas d'un lien de *Generalization*, c'est la fille qui est concernée par l'interdiction. Ainsi, il est par exemple interdit qu'une classe A hérite d'une classe B, mais pas que B hérite de A. En effet, dans le métamodèle UML, une classe fille est rattachée à sa mère par une métaclasse *Generalization*. À l'inverse, une classe mère est rattachée à sa fille par une métaclasse *Specialization* qui n'est pas concernée par ce stéréotype. La représentation d'un modèle par un graphe met en avant ce phénomène, comme l'illustre la figure 4.2.

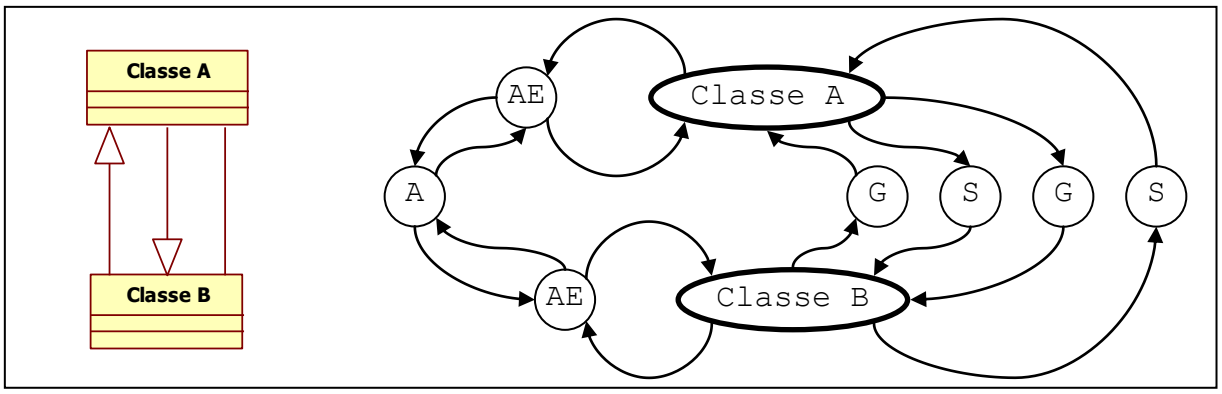


Figure 4.2 : Les différentes relations entre classes sous forme de graphe

Pour relier deux classes par une association, le même ensemble de sommets peut être utilisé (`AE`, `A`, `AE` sur la figure). En revanche, pour exprimer le fait que *Classe A* hérite de *Classe B*, le couple de sommets `G` et `S` n'est pas le même que pour exprimer que *Classe B* hérite de *Classe A*.

Ainsi, pour exprimer qu'aucune des deux classes ne doit hériter de l'autre, il est nécessaire de stéréotyper les métaclasses *Generalization* dans les deux sens. Cependant, si un sens de généralisation est *autorisé*, l'opposé est automatiquement interdit, de par les règles de bonne formation d'UML. Dans le cas d'une association, il est simplement nécessaire de stéréotyper chaque *AssociationEnd* de l'association concernée.

La figure 4.3 montre le patron abîmé « développement de la composition sur *Composite* », annoté par les informations issues du profil.

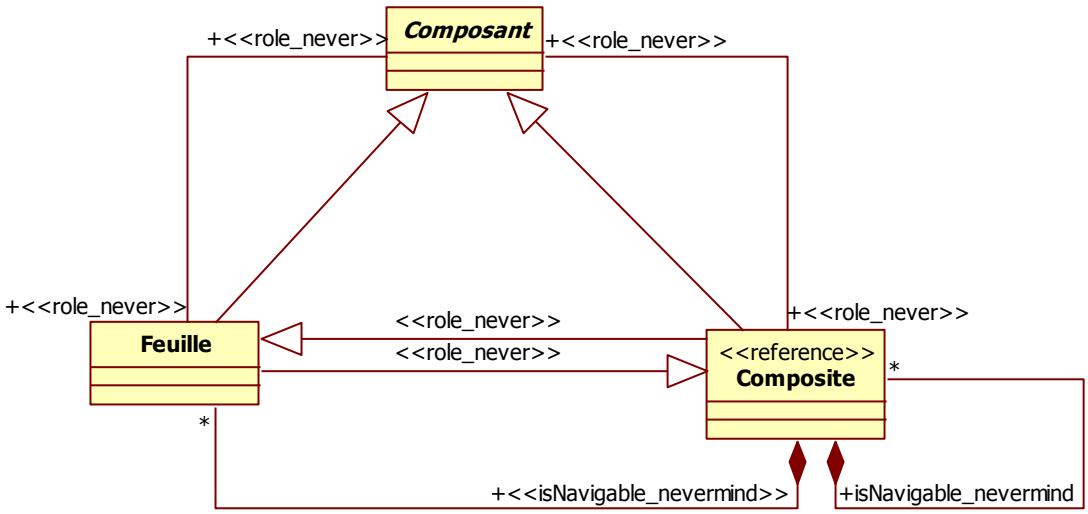


Figure 4.3 : Le patron abîmé "développement de la composition sur <<Composite>>", annoté par le profil

Grâce au profil, il est possible de remarquer que les relations réflexives sur *Composant* et sur *Feuille* sont facultatives puisque non représentées. De même, la binavigabilité des relations de composition est facultative, mais cette fois-ci, exprimée dans le patron abîmé par le stéréotype correspondant.

Avec cette représentation du patron abîmé, nous disposons de toutes les informations nécessaires à la détection. Le générateur peut alors exploiter directement un tel modèle annoté et intégrer dans les requêtes les contraintes exprimées par le profil.

I.2. Génération automatique de requêtes

L'axe central du générateur est de pouvoir générer des requêtes depuis n'importe quel modèle UML à partir du moment où une classe est marquée comme *référence*. Notre idée a donc été de développer un outil recherchant le participant de référence comme point de départ de la génération et naviguant dans le modèle en suivant tous les liens existants. Ce générateur est donc capable de construire des requêtes OCL permettant de détecter tout assemblage de classes, qu'il représente ou non un patron abîmé. Par extension, nous pouvons ainsi générer des requêtes permettant de détecter des patrons de conception, ou tout autre modèle caractéristique.

La figure 4.4 présente un extrait du modèle UML du générateur.

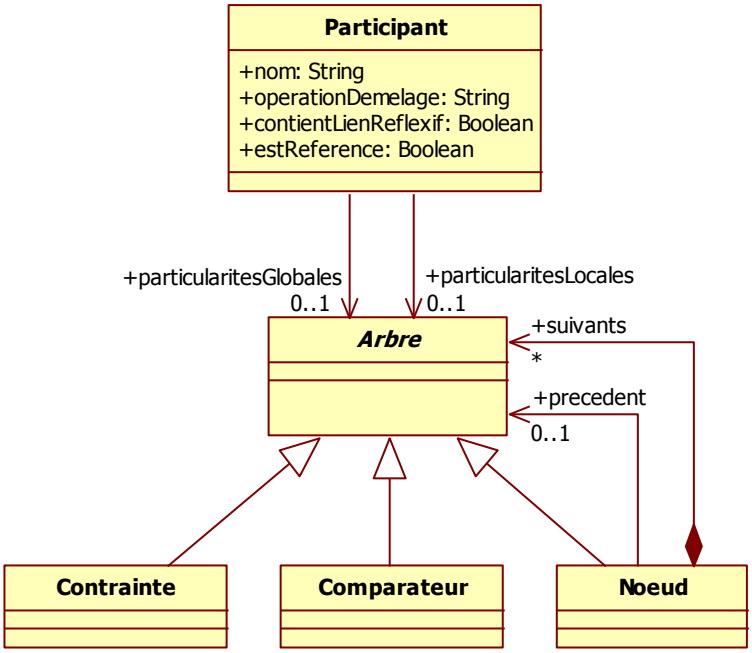


Figure 4.4 : Extrait du modèle UML du générateur

Nous pouvons remarquer que le générateur utilise le patron *Composite* pour construire l'arbre servant à mémoriser les particularités structurelles du patron abîmé. La base même du générateur est fondée sur un analyseur de modèle non représenté dans la figure 4.4. Cet analyseur construit le modèle en mémoire conformément à son métamodèle. Suivre les liens entre les classes du modèle s'effectue alors simplement en suivant les métaclasses représentant les relations entre classes (*Association*, *Connection*, *Generalization*…).

I.2.1. Construction d'une requête de détection

La première étape du générateur consiste à identifier tous les participants présents dans le patron abîmé. L'analyseur ayant représenté le modèle en mémoire sous forme de listes de métaclasses, identifier l'ensemble des participants revient à travailler sur la liste représentant la métaclasse *Classifier*. Le générateur cherche ensuite à identifier le participant de référence puis les particularités locales de chaque participant. Enfin, après avoir analysé comment les participants non-références sont rattachés au participant de référence, il identifie leurs particularités globales.

I.2.1.1. Recherche du participant de référence

L'identification du participant de référence dès le départ est importante, car le participant de référence n'a pas de particularités globales. De plus, comme il est seul dans le fragment détecté, l'identification des particularités globales des autres participants tient compte de lui. En effet, vérifier l'existence d'un chemin entre un participant non-référence et un participant de référence est simple, puisqu'il s'agit de suivre les métaclasses entre deux classes données. Pour un chemin entre deux participants non-références, il est nécessaire de suivre tous les chemins entre une classe et un ensemble d'autres classes. En effet, dans le fragment détecté, les classes non-références peuvent être plusieurs à se

partager les mêmes responsabilités au sein du même fragment. La requête devra donc intégrer des instructions comme « A est fille de B » si B est référence, et « A est fille de toutes les classes ayant les responsabilités de B » si B est non-référence.

I.2.1.2. Recherche des particularités locales

Rappelons que les particularités locales permettent de décrire structurellement un participant individuellement, c'est-à-dire ses relations entrantes et sortantes, sans spécifier la classe source ou destination. Ainsi, chaque participant du patron abîmé étant décrit avec ses particularités structurelles locales, une première étape de filtrage est possible lors de la détection pour limiter le nombre de classes à tester dans les étapes suivantes.

Étant donné que les particularités locales de chaque participant ne font pas intervenir les autres participants, le générateur peut les déduire en les analysant séparément. Pour chaque participant, le générateur interprète chacun des rôles UML qui entrent dans notre domaine de définition. Au regard des critères de performances de la détection, nous avons volontairement limité le domaine de définition à certains rôles uniquement, tout en laissant la possibilité d'en ajouter d'autres si nécessaire. L'état actuel de cette liste est résumé dans les tableaux 4.2 et 4.3.

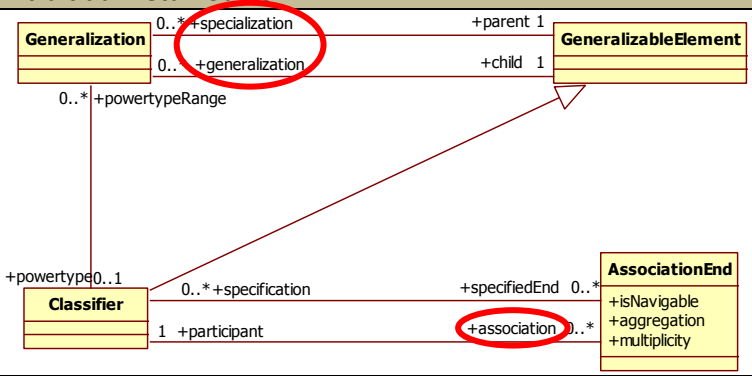
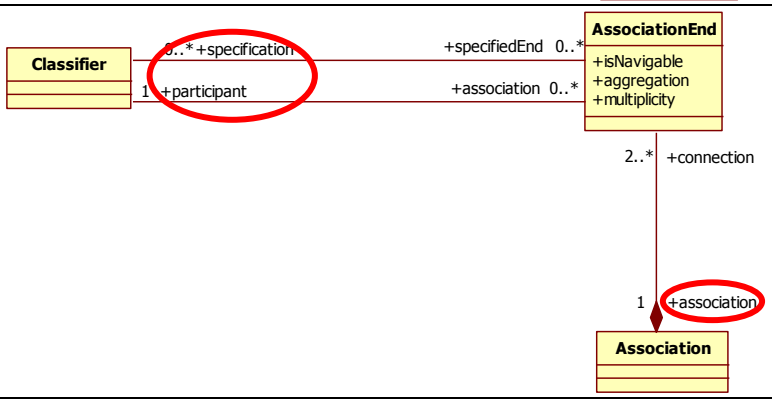
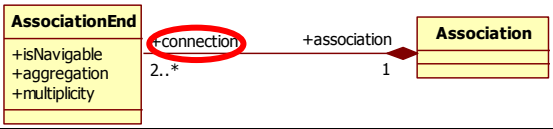
Métaclasse	Rôle	Extrait du métamodèle
Classifier	association generalization specialization	
AssociationEnd	participant association specification	
Association	connection	

Tableau 4.2 : Liste des rôles suivis par le générateur (partie 1)

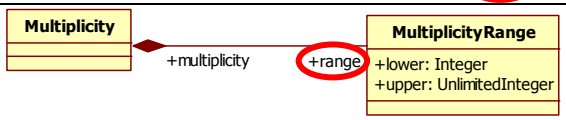
Métaclasse	Rôle	Extrait du métamodèle
Generalization	parent child	
Multiplicity	range	

Tableau 4.3 : Liste des rôles suivis par le générateur (partie 2)

Pour chaque participant, le générateur liste donc les rôles et les utilise pour se déplacer dans le métamodèle en suivant itérativement chacun d'entre eux. Les participants étant des *Classifier*, les rôles concernés sont donc *association*, *generalization* et *specialization*, ce qui permet d'atteindre, en continuant le parcours, les métaclasses *AssociationEnd* et *Generalization*. Étant donné qu'il s'agit de déduire des particularités locales, le générateur ne suit pas les rôles des métaclasses qu'il vient d'atteindre. Le parcours se limite ainsi aux sommets adjacents à l'élément considéré.

Le générateur construit alors l'arbre représentant le parcours qu'il vient d'effectuer. Pour chaque attribut de la métaclasse qu'il atteint, il crée les feuilles de l'arbre, de type *Contrainte*. Cette classe permet, lors de la génération de la requête, de construire une comparaison typée des attributs des métaclasses. En effet, pour comparer, par exemple, si l'attribut *aggregation* de *AssociationEnd* est *AGGREGATION*, *COMPOSITION* ou *NONE*, il est nécessaire d'utiliser le chemin absolu du type à comparer. Dans notre cas, il s'agit de « `Foundation::DataTypes::AggregationKind ::COMPOSITE` ». Dans le cas où le type de l'attribut à comparer est une autre métaclasse, par exemple l'attribut *multiplicity,* de type *Multiplicity,* le générateur reprend le parcours du métamodèle : il suit le rôle *range* (voir tableau 4.3) pour atteindre *MultiplicityRange*. Cette métaclasse n'ayant plus de rôle, le générateur instancie des classes *Contraintes* pour les attributs *lower* et *upper*.

Une fois toutes les feuilles créées, le générateur remonte le long des rôles qu'il a déjà parcourus. Ce parcours ayant été effectué par récursivité, à chaque remontée d'un niveau, le générateur instancie un *Nœud* de l'arbre contenant le nom du rôle parcouru. Lorsque le générateur rejoint le participant de départ, et que tous les rôles de ce participant ont été analysés, l'arbre représente les particularités locales du participant.

La figure 4.5 donne un exemple de l'arbre des particularités locales d'un participant. Il est possible d'y remarquer qu'il y a bien un niveau supplémentaire de rôles UML pour atteindre les attributs de *MultiplicityRange*. Dans le cas de *Composant*, le générateur n'a pas distingué les deux relations de spécialisation, car, localement, les deux rôles sont identiques. Nous verrons lors de la construction de l'arbre des particularités globales qu'il y a une distinction. Le cas est identique pour les deux associations de *Composite*. Pour ce faire, le générateur effectue normalement son parcours du métamodèle, puis, une fois terminé, vérifie si certaines branches de l'arbre sont identiques. Le cas échéant, il incrémente le champ *multiplicateur* de leur racine commune, et supprime l'un des deux. En procédant de la sorte, la requête générée permet la détection de deux spécialisations.

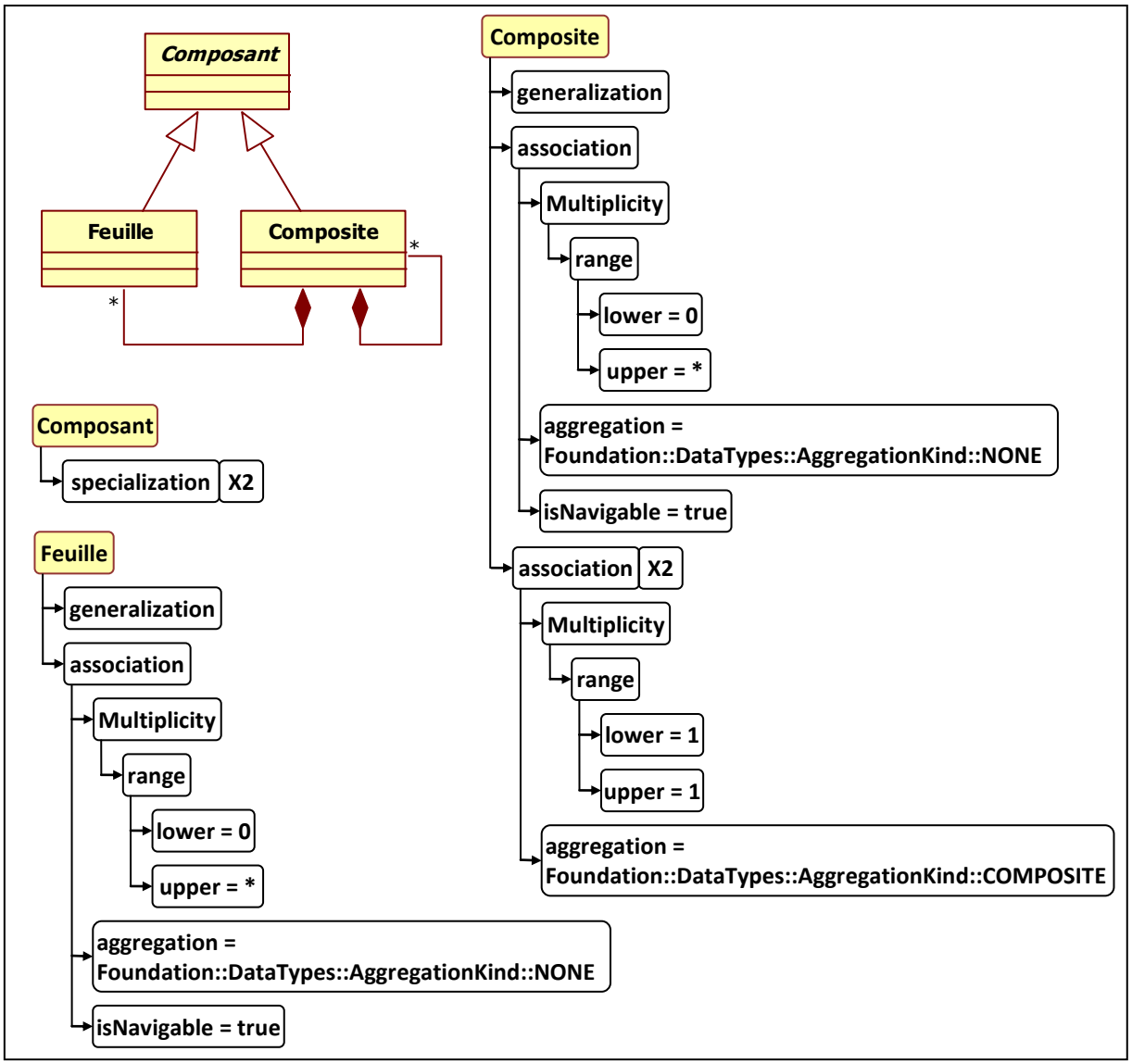


Figure 4.5 : L'arbre des particularités locales des participants du patron abîmé

Afin de ne pas suivre un lien interdit, le générateur vérifie si un stéréotype est présent sur chaque métaclasse qu'il atteint. Si un stéréotype indique que le lien est interdit, le générateur le mémorise et désactive la création des nœuds de l'arbre lors de la remontée du métamodèle. Pour les attributs facultatifs utilisant le stéréotype *[nom d'un attribut]_nevermind*, le générateur détecte le stéréotype sur la métaclasse concernée, élimine le « *_nevermind* », et compare chaque attribut de la métaclasse avant d'instancier la classe *Contrainte*. Le cas se présente pour le patron abîmé utilisé en exemple. À la figure 4.3, la navigabilité des deux associations partant de *Composite* étant facultative, une des branches *association* de l'arbre de *Composite* n'intègre pas la contrainte *isNavigable*. Ainsi, lors de la détection, quelle que soit la valeur de cet attribut, la classe candidate est sélectionnée.

I.2.1.3. Le cas particulier des relations réflexives

Les relations réflexives, comme c'est le cas pour le participant *Composite* du patron abîmé utilisé dans nos exemples, sont gérées comme des particularités locales. En effet, ce type de relation ne fait intervenir aucune autre classe et concerne localement la classe. Cependant, le générateur ne peut pas les déduire lors de la construction de l'arbre des particularités locales puisqu'il ne considère qu'un seul niveau de métaclasses. Il constate qu'il y a deux relations sortantes, mais ne détecte pas qu'elles représentent la même relation. Certes, la déduction des particularités globales permettrait de détecter ces relations, mais cela pose problème quand le participant concerné est le participant de référence. Comme ce dernier n'a pas de particularités globales, il ne serait pas possible de détecter les relations réflexives. Notre exemple montre précisément ce cas en ayant une relation réflexive sur le participant de référence *Composite*. Pour parer à ce problème, après avoir détecté les particularités locales, le générateur recherche toutes les relations réflexives de chaque participant, et intègre leur présence lors de la génération.

I.2.1.4. Construction d'une opération de démêlage

Lors de la première étape de détection, toutes les classes ayant les mêmes particularités structurelles locales que les participants du patron abîmé sont identifiées. Il en résulte plusieurs ensembles de classes correspondant à chacun des participants du patron abîmé. Pour distinguer de ces ensembles les différents fragments, il est nécessaire de rechercher à quelle classe de référence chaque classe non-référence est rattachée. Nous appelons cette étape le « démêlage » des fragments.

Pour permettre ce « démêlage », le générateur visite tous les rôles de chaque participant non-référence et essaye d'atteindre le *Classifier* représentant le participant de référence. Chaque participant étant attaché d'une manière ou d'une autre au participant de référence, le générateur l'atteint toujours. Le générateur construit alors directement une chaîne de caractères représentant les rôles qu'il vient de parcourir. Cette chaîne n'est pas préformée sous forme d'arbre comme pour les particularités locales, car il s'agit de vérifier, en appliquant la chaîne générée à un participant, s'il est possible d'atteindre le participant de référence de la même manière que dans le patron abîmé. Par exemple, la chaîne de démêlage est « depuis A pour aller à B, il faut suivre le rôle *Generalization* », tandis que la requête des particularités locales est « chercher toutes les classes ayant une *Generalization* ». Le démêlage n'est pas donc effectué par un algorithme itératif.

I.2.1.5. Recherche des particularités globales

La dernière étape de déduction du générateur consiste à extraire les particularités globales de chaque participant non-référence. Pour ce faire, le générateur effectue le même traitement que pour les particularités locales, sauf qu'il ne s'arrête pas à un seul niveau de métaclasse. Il examine tous les rôles de chaque participant, puis instancie un arbre des particularités globales. Hormis la profondeur, la différence avec les particularités locales se

situe dans le type des feuilles de l'arbre. En plus des *Contraintes*, le générateur instancie des classes *Comparateur*. Alors que les classes *Contraintes* sont utiles pour vérifier si la manière dont une classe est attachée à une autre est identique au patron abîmé, les classes *Comparateur* permettent de vérifier si la classe sélectionnée est rattachée aux bonnes classes du fragment. Par exemple, dans « [la classe A doit être rattachée à la classe C] [par une généralisation] », la première partie est issue d'une classe *Comparateur*, et la deuxième d'une *Contrainte*.

La particularité du type *Comparateur* réside surtout dans la différentiation des comparaisons si les classes concernées sont références ou non. En effet, pour vérifier qu'une classe est rattachée à un participant de référence, il suffit de vérifier que la classe rattachée est bien candidate aux responsabilités du rôle de référence. Dans ce cas, cette classe étant seule dans le fragment, il s'agit d'une comparaison de « un vers un ». Dans le cas des participants non-références, il peut y avoir plusieurs classes candidates dans le fragment. La comparaison devient alors « un vers plusieurs ». Il faut comparer la classe rattachée à toutes les classes candidates aux responsabilités du participant non-référence. C'est donc la classe *Comparateur* qui génère la comparaison appropriée dans la requête en fonction du participant concerné par la comparaison.

L'interprétation des liens interdits est effectuée à cette étape d'une tout autre manière que pour les particularités locales, où il s'agissait de ne pas tenir compte des rôles parcourus pour atteindre une classe. Cette fois, il s'agit d'interdire explicitement dans la requête le chemin d'accès à cette classe. Pour ce faire, le générateur utilise l'attribut *multiplicateur* qu'il positionne à zéro lorsqu'il parcourt un lien interdit. Ainsi, lors de la détection, la requête filtre les classes dont le nombre de ces rôles n'est pas nul. En ce qui concerne les stéréotypes exprimant le caractère facultatif de certains attributs, la génération est gérée de la même manière que pour les particularités locales, c'est-à-dire en éliminant de l'arbre l'ensemble des *Contraintes* concernées par les stéréotypes.

I.2.1.6. Assemblage de la requête de détection

Une fois que le générateur a identifié les particularités globales du patron et construit l'arbre associé, l'assemblage de la requête à proprement parler peut débuter. Cette requête regroupe plusieurs sous-requêtes élémentaires issues de la déduction (particularités locales, globales, démêlage…), ce qui permet leur exécution en une fois, chaque sous-requête utilisant le résultat de la précédente.

La structure de la requête est la même pour tous les patrons abîmés, les sous-requêtes sont appelées de la même manière quel que soit leur nombre (dépendant du nombre de participants) et leur taille. Finalement, la requête contient trois sous-requêtes, en ne comptant pas les instructions spécifiques à son exécution, comme illustré dans le code source 4.1. Dans ce template, les parties en gras sont des instructions pour le générateur, les parties entre {} sont remplacées par leurs valeurs associées et les autres instructions représentent le point commun entre chaque requête de chaque patron abîmé.

```
{entêtes}

pour chaque participant du patron abîmé, dupliquer
|   fonction step1_{nomParticipant}_{i} retourne set(classes) :
|   |  {requête des particularités locales}
|   fin fonction
fin pour

pour chaque participant non-référence du patron abîmé, dupliquer
|   fonction step2_{nomParticipant}_{i}(set(set(classes))) retourne set(set(classes)) :
|   |  {opération de démêlage}
|   fin fonction
|
|   fonction step3_{nomParticipant}_{i}(set(set(set(classes)))) retourne set(set(set(classes))) :
|   |  {requête des particularités globales}
|   fin fonction
fin pour

fonction local retourne set(set(classes)) :
| construit set(
|   |  pour chaque participant du patron abîmé, dupliquer
|   |  |  step1_{nomParticipant}_{i}
|   |  fin pour
| )
fin fonction

fonction démêlage retourne set(set(set(classes))) :
| construit set(
|   |  pour chaque participant non-référence du patron abîmé, dupliquer
|   |  |  step2_{nomParticipant}_{i}(local)
|   |  fin pour
| )
fin fonction

fonction global retourne set(set(set(classes))) :
  |  pour chaque participant non-référence du patron abîmé, dupliquer
  |  |  step3_{nomParticipant}_{i}(démêlage)
  |  fin pour
fin fonction

exécuter global
```

Code source 4.1 : Le template des requêtes de détection

La première sous-requête représente la recherche des particularités locales. Pour la générer, le générateur utilise une fonction d'aplatissement de l'arbre des particularités locales. En partant des feuilles de l'arbre, cette fonction construit la chaîne de caractères représentant la sous-requête OCL en remontant les branches de l'arbre vers la racine. Cette sous-requête permet la détection des classes ayant les mêmes responsabilités que chaque participant du patron abîmé. Le résultat de l'exécution de cette sous-requête est un ensemble d'ensembles de classes. Chaque ensemble contient autant d'ensembles que de participants dans le patron abîmé, et chacun de ces sous-ensembles contient les classes ayant les mêmes particularités structurelles locales que les participants correspondants.

La deuxième sous-requête concerne le « démêlage » des fragments et est donc construite directement à partir de la chaîne de démêlage. Cette sous-requête utilise les ensembles obtenus par l'étape précédente et effectue le démêlage en utilisant le participant de référence du patron abîmé. Chaque ensemble de classes est donc filtré pour qu'il ne reste que les classes correctement rattachées au participant de référence. La sous-requête provoque la génération d'ensembles de fragments potentiellement alternatifs, chaque fragment étant représenté par un ensemble d'ensembles de classes pour chaque participant du patron.

La dernière sous-requête, qui concerne les particularités globales, est construite de la même manière que celle des particularités locales, c'est-à-dire en aplatissant l'arbre représentatif. Cependant, afin de pouvoir intégrer dans la sous-requête la comparaison avec les ensembles obtenus dans les sous-requêtes précédentes, le générateur référence ces ensembles directement dans les classes *Comparateur* avant de lancer l'aplatissement. À l'issue de cette sous-requête, chaque classe du fragment potentiellement alternatif a été vérifiée et est conforme aux particularités globales attendues.

I.3. Génération et exécution des opérations de transformation

Le générateur que nous avons développé est également capable de générer les requêtes permettant la transformation du fragment détecté en patron abîmé contextualisé sur le modèle analysé. À partir du patron de conception et d'un patron abîmé, il génère des opérations permettant de transformer le patron abîmé en patron de conception. Ces opérations se résumant à des ordres à exécuter sur des ensembles de classes, elles sont compatibles avec n'importe quelle contextualisation du patron abîmé.

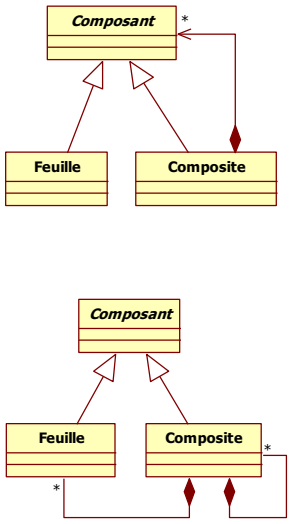
I.3.1. Différenciation du patron abîmé par rapport au patron de conception

Pour générer les requêtes de transformation, le générateur analyse le patron de conception et le patron abîmé, en utilisant le même analyseur que celui des requêtes de détection. Il constitue ensuite deux listes contenant respectivement l'ensemble des participants du patron et l'ensemble des participants du patron abîmé. Ces deux listes n'ont pas obligatoirement la même taille puisqu'un patron abîmé peut avoir des participants en plus ou en moins par rapport au patron de conception.

À partir de ces listes, le générateur parcourt chacun des deux patrons en utilisant le même système de génération que celui des requêtes de détection. Pour chacun des participants, il se déplace dans le métamodèle jusqu'à ce que la couverture des relations soit complète, afin de construire une chaîne de caractères représentant le lien entre deux participants, de la même manière que pour la chaîne de démêlage. Le code source 4.2 présente la chaîne générée pour le patron *Composite* et l'un de ses patrons abîmés.

```
MODEL_1_DESIGN_PATTERN {
   Composant={
      Foundation.Core.Association=[Composite],
      Foundation.Core.Generalization=[Feuille, Composite]
   },
   Feuille={
   },
   Composite={
   }
}

MODEL_2_SPOILED_PATTERN {
   Composant={
      Foundation.Core.Generalization=[Composite, Feuille]
   },
   Feuille={
      Foundation.Core.Association=[Composite]
   },
   Composite={
      Foundation.Core.Association=[Composite]
   }
}
```



Code source 4.2 : La chaîne de description du patron *Composite* et d'un de ses patrons abîmés

La structure de la chaîne se compose du nom de chaque participant suivi du nom de la métaclasse représentant le lien et termine sur le ou les noms des participants attachés. Il est possible de remarquer que les métaclasses *AssociationEnd* des associations n'apparaissent pas dans les chaînes. Ce choix est volontaire, car la transformation est capable de gérer, pour les commandes d'ajout ou de suppression d'une *Association*, les *AssociationEnd* impliquées. Nous pouvons également remarquer que la relation entre deux participants n'apparaît qu'une seule fois dans la chaîne, et donc, dans un seul sens. Par exemple, dans la chaîne du patron de conception, *Composant* est associé à *Composite*, mais *Composite* n'est pas associé à *Composant*. Il en est de même pour l'héritage : *Composant* est associé par un héritage à *Feuille* et à *Composite*, mais on ne sait pas dans quel sens. Cette étape ayant pour simple objectif de référencer les liens des deux patrons, l'orientation des relations n'a pas d'importance.

I.3.2. Construction des opérations de transformation

Une fois les deux chaînes construites, le générateur les compare en cherchant ce qu'il est nécessaire d'ajouter ou de supprimer dans le patron abîmé pour qu'il se transforme en patron de conception. Le générateur effectue donc un traitement itératif sur la chaîne du patron abîmé et sur la chaîne du patron de conception pour mettre en avant les différences. Dans notre exemple, en prenant le participant *Composant* du patron abîmé, on remarque qu'il manque une association. L'héritage est correct. Pour *Feuille* il y a une association en trop, de même que pour *Composite*. La comparaison se résume donc à ajouter une *Association* entre *Composite* et *Composant*, et à supprimer les autres. Le problème dans ce cas est de savoir si les *Generalization* sont bien dans le même sens, et quel est le type d'*Association* qu'il faut ajouter. Le générateur utilise les adresses des métaclasses, stockées précédemment, pour comparer le sens des *Generalization* (en interrogeant les rôles *child* et *parent*) et pour déterminer les caractéristiques des *AssociationEnd* liées à l'*Association* à rajouter. Une fois ces comparaisons terminées, le générateur génère les commandes de transformations. Celles de notre exemple sont présentées dans le code source 4.3.

```
TEMPLATE COMMAND : [COMMAND, METACLASS, FROM, TO, HOWfrom, HOWto]
FROM SPOILED PATTERN #5
TO DESIGN PATTERN COMPOSITE
[
   [
      ADD,
      Foundation.Core.Association,
      Component,
      Composite,
      {aggregation=NONE, upper=-1, lower=0, isNavigable=true},
      {aggregation=COMPOSITE, upper=1, lower=1, isNavigable=true}
   ],
   [
      DELETE,
      Foundation.Core.Association,
      Leaf,
      Composite,
      {aggregation=NONE, upper=-1, lower=0, isNavigable=true},
      {aggregation=COMPOSITE, upper=1, lower=1, isNavigable=true}
   ],
   [
      DELETE,
      Foundation.Core.Association,
      Composite,
      Composite,
      {aggregation=NONE, upper=-1, lower=0, isNavigable=true }
      {aggregation=COMPOSITE, upper=1, lower=1, isNavigable=true}}
   ]
]
```

Code source 4.3 : Une requête de transformation

Chaque commande est constituée de six champs distincts. Chaque champ représente respectivement le *nom de l'action* à exécuter sur le fragment (*COMMAND* qui peut être *ADD* ou *DELETE*), la *métaclasse* concernée par l'action (*METACLASS*), le participant représentant la *source* de la relation (*FROM*), le participant représentant la *cible* de la relation (*TO*), et les *caractéristiques* des métaclasses représentant la relation, du côté source (*HOWfrom*) et du côté cible (*HOWto*). Ces deux derniers champs sont optionnels, car il se peut qu'ils ne contiennent aucune information supplémentaire nécessaire, comme par exemple, lorsque la métaclasse est *Generalization. Dans ce cas,* le sens de la relation est exprimé par la position des champs trois *FROM* et quatre *TO de* la requête. Les commandes intègrent ainsi toutes les informations d'ordre et les caractéristiques des relations. Le deuxième champ des commandes peut être n'importe quelle classe du métamodèle. Cette généricité permet à la fois d'ajouter ou de supprimer n'importe quel type de relation.

I.3.3. Exécution des opérations de transformation

Le fait que les commandes citent uniquement le nom du participant à transformer permet, lors de la transformation, de travailler directement sur les ensembles sélectionnés. En effet, il suffit, par exemple, d'ajouter une *Association* entre toutes les classes du fragment marquées *Component* et *Composite*. Cette particularité nous permet d'avoir des requêtes totalement génériques sur toutes les contextualisations possibles du patron abîmé. Si le fragment a été détecté, alors toutes les classes sont marquées et sont donc concernées par la requête de transformation, et ce, quelle que soit la forme du fragment. Si le fragment a un lien en plus que nous n'avions pas prévu, cela signifie qu'il est facultatif dans le fragment et dans ce cas, il n'est pas concerné par la transformation. Il est difficilement envisageable que ce lien supplémentaire dénature la transformation (en provoquant par exemple l'apparition d'un lien contradictoire après la transformation), puisque nous interdisons explicitement

dans la description du patron abîmé, tout lien pouvant potentiellement le dégrader ou dégrader le patron. De plus, pour parer à toute éventualité et pour offrir au concepteur le contrôle total de son modèle, la transformation est réversible. La figure 4.6 présente le modèle du transformateur.

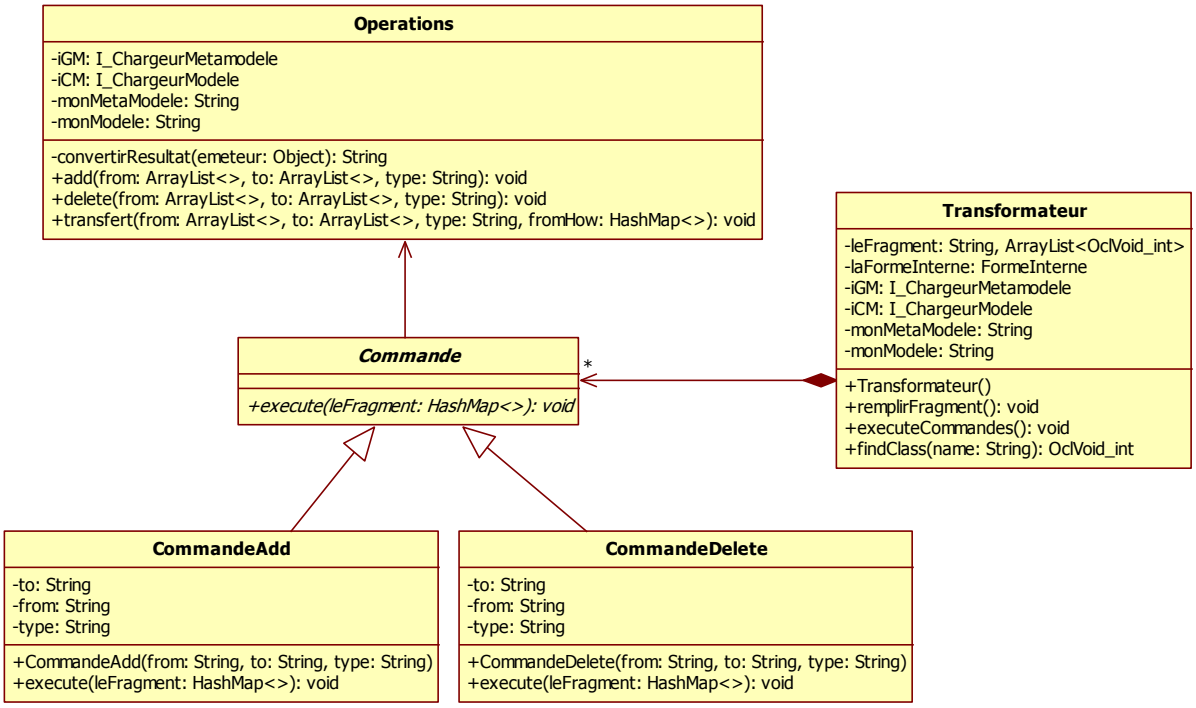


Figure 4.6 : Le modèle du transformateur de Triton

Il est possible de remarquer que le transformateur est conçu en utilisant le patron de conception *Commande*, ce qui permet de spécialiser l'exécution de chaque type de commande, et de les annuler. Chaque commande travaille avec un fragment exprimé sous forme de liste d'associations *[Clef, Valeur]*, la *clef* étant le participant du patron, la *valeur* étant la liste des classes du modèle ayant les mêmes responsabilités que le participant. C'est à chaque élément de cette liste que la commande est appliquée. Le transformateur est capable de gérer les rôles de chaque métaclasse modifiée.

Il est important de remarquer que la transformation reste structurelle. Aucune intervention n'est effectuée sur les méthodes ou sur les attributs à l'issue de la transformation. Nous laissons donc à la charge du concepteur, une fois la transformation effectuée, l'adaptation des méthodes et des attributs à la nouvelle architecture de classes. Nous considérons que la personne la plus à même pour modifier le modèle reste le concepteur. En effet, alors que nous connaissons, grâce aux points forts des patrons, l'impact sur le modèle des relations entre les classes, nous ne pouvons pas, en l'état actuel de nos travaux, anticiper les responsabilités des différentes méthodes et attributs. Cela nous empêche donc d'envisager leur modification automatique. Cependant, nous considérons que cela ne constitue pas une limite à notre approche, puisque nous parvenons à attirer l'attention du concepteur sur des fragments de son modèle où il peut intervenir.

Notre implémentation nous permet de parcourir efficacement les modèles à la recherche de fragments alternatifs. Nous abordons ce point dans la section III de ce chapitre. L'utilisation d'OCL, imposée en partie par la plate-forme Neptune, nous a permis d'effectuer une recherche directement dans les modèles, sans imposer de prétraitement au concepteur. Cependant, de par sa finalité originelle, OCL a apporté une limite à notre détection.

I.4. Limite de l'utilisation d'OCL

Au commencement de notre étude, nous décrivions structurellement les patrons abîmés en une seule expression : il n'y avait pas de différenciation entre particularités locales et globales. Ainsi, nous recherchions toutes les classes du modèle « structurellement ressemblantes » au participant de référence, et depuis chacune d'elle, nous tentions d'atteindre les autres participants du patron abîmé, en s'assurant que les liens suivis étaient identiques à ceux du patron. De plus, nous souhaitions que le concepteur ait la possibilité d'adapter la requête de détection aux particularités de son modèle. Comme les requêtes étaient écrites à la main, et non générées automatiquement, elles étaient facilement compréhensibles par un concepteur maîtrisant OCL.

Le code source 4.4 présente l'une de ces requêtes, où il est possible de remarquer les trois sous-requêtes correspondant à chaque participant du patron. Le premier, *Composite*, est le participant de référence et n'est décrit que par ses associations. Le deuxième est *Composant* où pour chaque *Composite* trouvé, la requête cherche toutes les classes parentes. Pour *Feuille*, le troisième, la requête cherche toutes les classes associées et filles de *Composant,* pour chaque *Composite* trouvé.

```
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
|  |  )
|  )

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
|  |  |  and c.association.connection->select(
|  |  |  |  aggregation=1
|  |  |  ).participant.parent.includesAll(composant)
|  |  )
|  )

in
    composant->fail('COMPOSANTS : ')
    composite->fail('COMPOSITES : ')
    feuille->fail('FEUILLES : ')
```

Code source 4.4 : Première génération de requête de détection

Au fil de notre étude, nous avons amélioré notre requête, de manière à ce qu'elle intègre nos nouveaux concepts, entre autres les particularités structurelles locales et globales, ainsi que les relations interdites. En ajoutant tous ces paramètres, les requêtes sont devenues difficiles à écrire manuellement. Nous avons alors développé le générateur, produisant des requêtes précises, mais très difficiles à relire. Ces requêtes de cent vingt lignes sont composées de trois sous-requêtes pour un participant (*particularités locales*, *démêlage*, *particularités globales*), dont la troisième est particulièrement difficile à appréhender pour un concepteur, même s'il maîtrise OCL, notamment à cause des relations interdites décrites conjointement aux particularités globales. En guise d'exemple, une de nos dernières requêtes est présentée en annexe 2.

Pour mémoire, qu'il n'est pas possible de connaître de manière déterministe l'*AssociationEnd* suivante d'un lien *Classifier AssociationEnd Association*, comme illustré dans la figure 4.7.

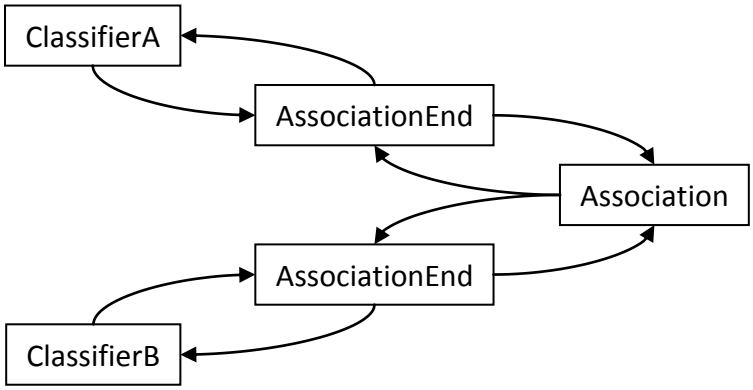


Figure 4.7 : Le problème de la réification de la métaclasse *Association* lors du parcours du métamodèle

Ce schéma représente les chemins possibles pour se déplacer dans le métamodèle. Si l'on veut partir du *ClassifierA* pour aller au *ClassifierB*, on remarque que lorsque l'on atteint la métaclasse *Association*, la suite du parcours n'est pas déterministe. Pour continuer vers la bonne *AssociationEnd*, il suffirait d'effectuer un marquage de l'*AssociationEnd* précédente. Ce marquage est impossible en OCL notamment parce que les requêtes OCL ne provoquent pas d'effets de bord sur les modèles. Ainsi, nous avons dû travailler les requêtes pour remplacer ce marquage. Actuellement, après un passage par la métaclasse *Association*, nous suivons les deux liens *AssociationEnd*, et nous comparons le *Classifier* attaché. Ceci complexifie d'autant la requête qu'il devient difficile de la modifier manuellement.

En l'état actuel de nos travaux, notre contrainte d'origine visant à permettre au concepteur de modifier, à son gré, la requête n'est plus applicable. Il nous est donc nécessaire de trouver un équilibre entre la lisibilité des requêtes, leur génération automatique, et le fait que les problèmes de morphisme de graphes sont complexes et dépassent la portée du langage OCL. Une solution pourrait consister à ne plus utiliser OCL et d'exploiter directement la forme interne des modèles de la plate-forme Neptune. En procédant de la sorte, le marquage des chemins parcourus serait possible réduisant ainsi le nombre de tests à effectuer lors de la détection. Cette diminution des tests réduira le temps d'exécution, mais aussi le temps d'interprétation des résultats.

D'après notre activité, entre la détection et la transformation des fragments, le concepteur intervient pour donner son avis sur l'intention des fragments détectés. La détection ayant été structurelle, la vérification de l'intention est indispensable. Afin d'interroger le concepteur sur l'intention des fragments, et pour lui montrer en quoi la transformation de ce fragment améliorerait son modèle, nous avons utilisé une ontologie nous permettant de structurer chacun de nos concepts les uns par rapport aux autres. Cette mise en relation nous permet de construire des questions en langage naturel en fonction des éléments issus de la détection.

II. Explications au concepteur

Vérifier auprès du concepteur l'intention des fragments détectés, puis lui expliquer ce que la transformation apporterait à son modèle, nécessite de pouvoir accéder à l'intention des patrons de conception, ainsi qu'à leurs points forts. En effet, l'intention d'un patron abîmé et donc de sa contextualisation est la même que celle du patron de conception correspondant. De plus, les défauts de conception induits par un patron abîmé et corrigés par le patron de conception sont caractérisés par les perturbations causées aux points forts par le patron abîmé.

En étendant une ontologie destinée à formaliser l'intention des patrons de conception dans le but d'aider à leur réutilisation, nous avons pu structurer nos concepts indispensables à l'établissement d'un dialogue avec le concepteur. En effet, les ontologies sont utilisées pour capturer la connaissance en décrivant les concepts du domaine ainsi que leurs relations en fournissant un vocabulaire formel. L'intérêt principal de leur développement concerne le partage, la réutilisation, l'analyse et la séparation de la connaissance du domaine de la connaissance opérationnelle [Noy01].

Dans cette section, nous nous attachons à présenter les objectifs de l'ontologie qui nous a servi de support de travail. Cette ontologie est utilisée dans la même problématique globale que nos travaux : favoriser l'utilisation des patrons de conception. Destinée à formaliser leur intention afin d'aider le concepteur à choisir le patron le plus adéquat à son problème, cette ontologie s'inscrit en partie dans notre intention de communiquer avec le concepteur. En l'interrogeant en partie différemment, puis en ajoutant nos concepts de points forts et de patron abîmés, nous montrons en deuxième partie comment nous utilisons cette ontologie dans notre activité. Nous mettons également en valeur le fait que cette ontologie est notre base génératrice de questions destinées au concepteur, lors de l'exécution de la revue de conception.

II.1. Design Pattern Intent Ontology (DPIO)

De nombreux travaux existent pour affiner la représentation des patrons de conception afin d'aider à leur réutilisation. La plupart de ces travaux nécessitent une description formelle des patrons de conception. Dietrich et al. [Dietrich05] ont utilisé OWL (*Ontology Web Language*) [McGuinness04] pour décrire formellement la structure des patrons de conception. Cependant, ils n'ont pas intégré de détails sur l'intention ou l'applicabilité des patrons, à l'inverse des travaux de [Kampffmeyer07], qui ont utilisé une ontologie OWL pour relier des concepts connexes à chaque patron de conception, permettant notamment d'en faire émerger l'intention. Cette ontologie qu'ils ont appelée DPIO est une base de connaissances extensible de patrons de conception classés selon leur intention. Son objectif est d'aider les concepteurs à choisir les patrons les plus adaptés à leurs problèmes.

II.1.1. Ontology Web Language (OWL)

OWL est un langage de balises sémantiques pour l'édition et le partage des ontologies sur le Web. Le langage OWL a été conçu pour être utilisé par les applications ayant besoin de comprendre l'information disponible sur le Web. OWL permet de décrire des classes et des propriétés grâce à des constructeurs (par analogie avec la programmation par objets), et à des concepts de *classe*, de *ressource*, de *littéral* et de *propriétés des sous-classes*, de *sous-propriétés*, de *champs de valeurs* et de *domaines d'application*.

Une ontologie OWL se détaille en « *individus* », « *propriétés* » et « *classes* ». Les *individus* représentent les objets du domaine concerné. Les *propriétés* sont les relations entre deux éléments (concepts ou individus) et un schéma de données XML. Les *propriétés* peuvent être limitées à une seule valeur, être transitives ou symétriques. Les *classes* sont une représentation concrète des concepts, interprétées comme des ensembles d'individus, sachant qu'un individu peut appartenir à une classe ou à plusieurs. Les *classes* sont hiérarchisables comme en UML par des relations « parent-enfant ». Pour résumer, les classes sont interprétées comme des ensembles d'objets qui représentent les individus du domaine. Les propriétés sont des relations binaires qui lient les individus, et sont interprétées comme des ensembles de tuples.

II.1.2. Description de l'ontologie DPIO

La structure de DPIO est présentée en figure 4.8. La notation utilisée est UML par souci de lisibilité. Les classes et les associations UML symbolisent respectivement les classes OWL et les propriétés des objets OWL.

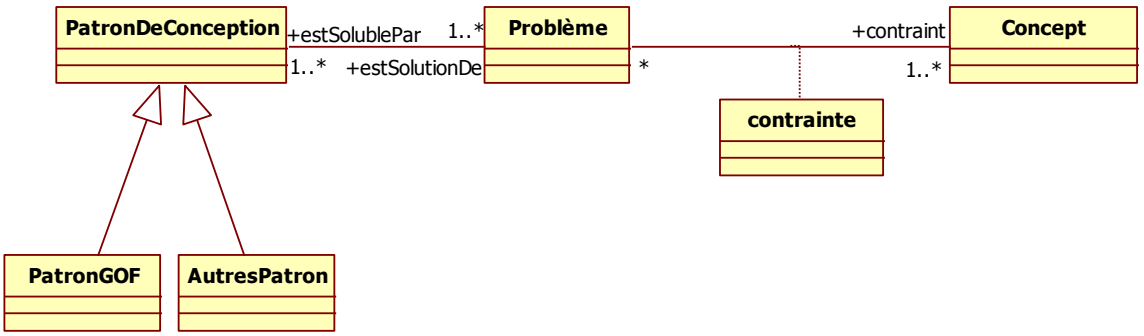


Figure 4.8 : L'ontologie DPIO [Kampffmeyer07]

Chaque *PatronDeConception* est solution d'un ou plusieurs *Problèmes* qui sont eux-mêmes solubles par un ou plusieurs *PatronsDeConception*. Un *Problème* est un concept sur lequel on a appliqué une contrainte sous forme d'une propriété OWL. Les vingt-trois patrons du GoF ont été intégrés dans cette ontologie sous la classe *PatronGOF*. La représentation du patron *Composite* dans l'ontologie est illustrée dans la figure 4.9.

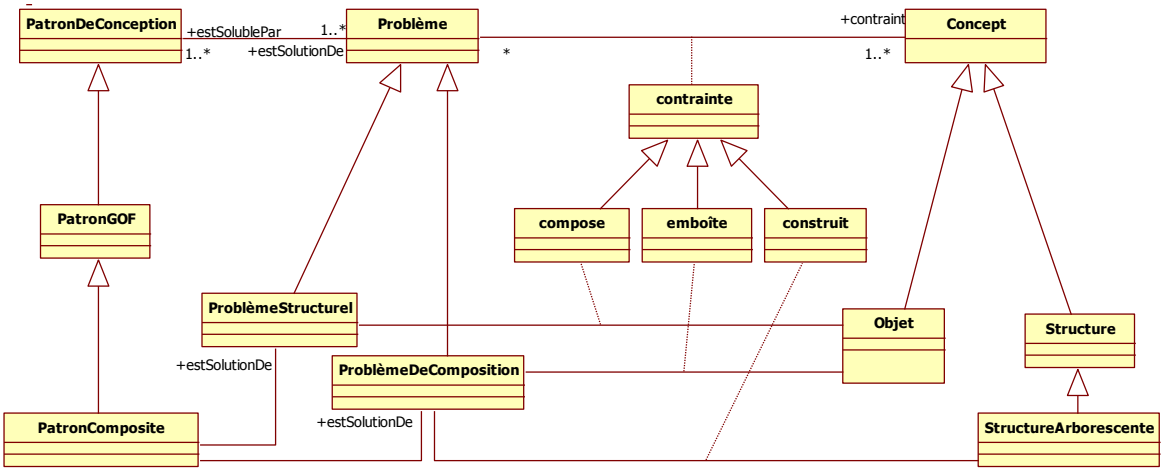


Figure 4.9 : La représentation du patron *Composite* dans l'ontologie DPIO [Kampffmeyer07]

Nous pouvons remarquer que chaque classe de l'ontologie est spécialisée par un concept concernant le patron *Composite*. Dans une ontologie, il n'y a pas de notion d'instanciation de classe. Ainsi, la classe *PatronComposite* est un *PatronGOF* qui est un *PatronDeConception*. Le *PatronComposite* est une solution pour des *ProblèmesStructurels* et des *ProblèmesDeComposition*. D'après l'ontologie, un *ProblèmeStructurel* consiste à « composer des objets », et un *ProblèmeDeComposition* à « emboîter des objets » et à « construire des structures arborescentes ». Ainsi, l'ontologie indique que le patron *Composite* a pour intention : « composer et emboîter les objets ainsi que construire des structures arborescentes ». Le code source 4.5 présente le même patron dans l'ontologie, mais en OWL.

```
Class: PatronComposite
  SubClassOf: PatronGOF
  and estSolutionDe some ProblèmeStructurel
  and estSolutionDe some ProblèmeDeComposition

Class: ProblèmeStructurel
  SubClassOf: Problème
  and compose some Objet

Class: ProblèmeDeComposition
  SubClassOf: Problème
  and emboîte some Objet
  and construit some StructureArborescente

Class: Objet
  SubClassOf: Concept

Class: StructureArborescente
  SubClassOf: Structure

Class: Structure
  SubClassOf: Concept

ObjectProperty: compose
  SubPropertyOf: contrainte

ObjectProperty: emboîte
  SubPropertyOf: contrainte

ObjectProperty: construit
  SubPropertyOf: contrainte
```

Code source 4.5 : Le patron *Composite* et sa représentation dans l'ontologie DPIO en OWL

Un important travail de validation a été effectué par les auteurs [Kampffmeyer07], afin de garantir une bonne formalisation de l'intention de chaque patron. En effet, dans le GoF, l'intention du patron *Composite* est définie ainsi : « *Le modèle Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon les objets individuels et leur combinaison* ».

II.1.3. Utilisation de l'ontologie DPIO

L'ontologie DPIO [Kampffmeyer07] offre une aide aux concepteurs pour choisir les patrons de conception dont ils ont besoin pour résoudre les problèmes qu'ils rencontrent. Afin d'utiliser l'ontologie sur des cas concrets d'aide à l'utilisation des patrons de conception, un interprète de requêtes permet de retrouver tous les patrons reliés aux contraintes et aux concepts fournis par le concepteur. En effet, la recherche des patrons de conception par leur outil s'effectue à partir de couples [contrainte – concept]. Cet interprète permet de répondre à des questions comme : « *Quel patron de conception est une solution au problème de composition des objets ?* » à partir d'instructions fournies par le concepteur sous la forme de : « la solution [contraint][quelque chose] », comme illustré dans la figure 4.10.

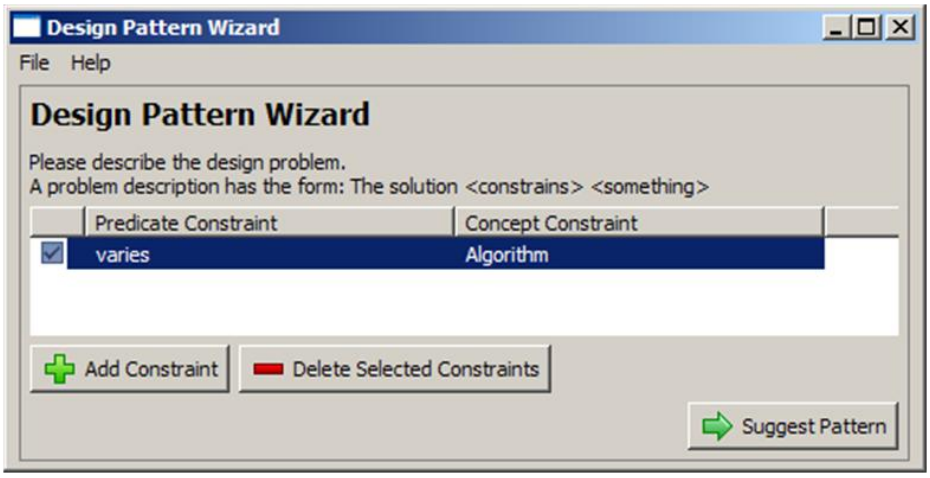


Figure 4.10 : L'outil d'aide au choix des patrons de conception utilisant l'ontologie DPIO

Les concepteurs doivent donc indiquer la contrainte et le concept inhérents à la solution de leur problème de conception. Dans l'exemple présenté, pour résoudre le problème du concepteur, la solution doit permettre une variation des algorithmes. Une fois la saisie de toutes les contraintes effectuée, l'outil parcourt l'ontologie pour proposer la liste des patrons de conception répondant à ces contraintes. La figure 4.11 illustre cette liste.

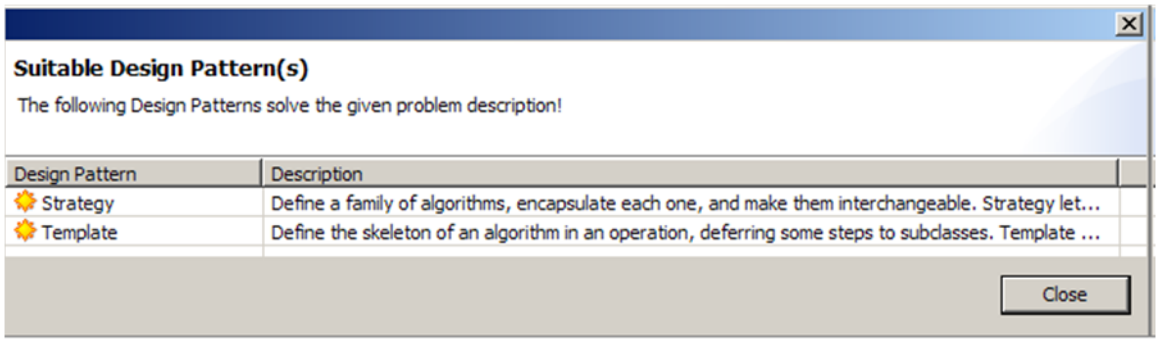


Figure 4.11 : Les patrons proposés par l'outil d'aide au choix des patrons de conception

À la contrainte « variation des algorithmes », l'outil a ainsi correctement identifié deux patrons de conception, *Stratégie* et *TemplateMethod*.

La lecture des données d'une ontologie OWL pouvant être effectuée de manière binavigable [McGuinness04], nous pouvons, à partir de la donnée « patron de conception », atteindre l'intention, de la même manière que l'outil présenté permet de connaître les patrons de conception concernés par la donnée « intention ».

II.2. Extension de l'ontologie DPIO

Dans nos travaux, nous identifions des fragments alternatifs dans un modèle. Ces fragments sont des contextualisations de patrons abîmés, eux-mêmes rattachés à des patrons de conception. Ainsi, à l'issue de la détection, l'identification du « patron de conception » nous permet, en utilisant des requêtes adéquates, d'utiliser l'ontologie pour en retrouver l'intention. En présentant cette intention au concepteur, il peut vérifier si le fragment identifié a bien la même intention que le patron de conception. De plus, afin de présenter au concepteur en quoi la substitution est avantageuse pour son modèle, nous

pouvons ajouter à l'ontologie les points forts perturbés par les patrons abîmés. Ainsi, en plus de générer les questions de vérification de l'intention de fragments, l'ontologie nous permet de générer des messages informatifs relatifs aux conséquences de la substitution [Harb08] [Harb09].

II.2.1. Conception

Pour déterminer la portée d'une ontologie, il est nécessaire d'identifier des « questions de compétences » auxquelles l'ontologie devra être en mesure de répondre [Noy01]. Ainsi, dans notre cas, les questions sont :

1. Quel patron de conception peut remplacer le fragment identifié comme contextualisation d'un patron abîmé ?
2. Quelle est l'intention d'un patron de conception donné ?
3. Quels sont les points forts dégradés par le patron abîmé ?

La première question permet d'identifier le patron de conception lié au fragment détecté, la deuxième sert à construire le dialogue avec le concepteur afin de vérifier l'intention. La troisième question est utile pour présenter au concepteur les défauts de conception visés par le fragment qui seront corrigés par l'injection du patron. Alors que la réponse à la première question nécessite de relier le concept de patron abîmé au concept de patron de conception déjà présent dans l'ontologie, la deuxième question ne nécessite aucune modification. Le concept de point fort nous étant spécifique, son ajout est indispensable dans l'ontologie, en le reliant au concept de patron abîmé. La nouvelle ontologie est présentée, au format UML, dans la figure 4.12, où l'extension est mise en valeur.

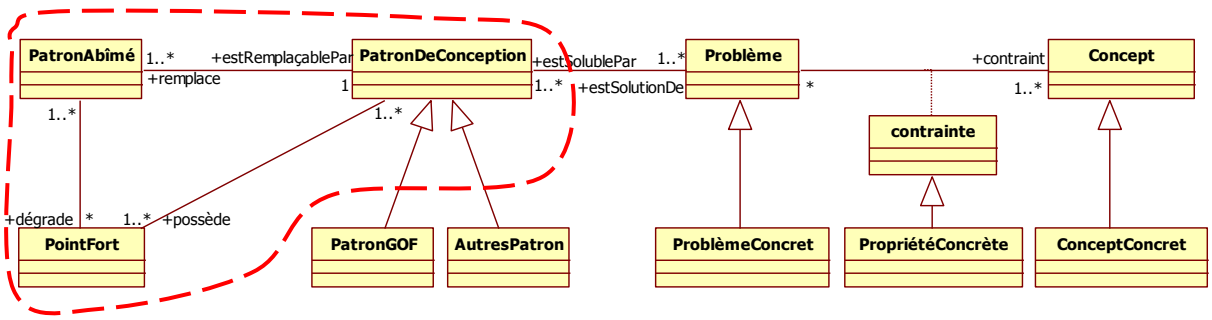


Figure 4.12 : L'extension de l'ontologie DPIO

Nous avons donc ajouté la classe *PatronAbîmé* en la reliant à *PatronDeConception* par une relation de remplacement. L'ajout de cette classe nous offre un point d'accès à l'ontologie. La recherche d'intention suit alors le nouveau lien et les anciens liens existants vers *Problème*, *contrainte* et *Concept*.

La classe *PointFort* a été reliée avec le *PatronAbîmé* et le *PatronDeConception*. En effet, un patron de conception possède des points forts qui justifient en quoi le patron est la meilleure solution pour un problème donné. De son côté, un patron abîmé dégrade certains points forts du patron auquel il est rattaché. Le lien *possède* relie tous les points forts du

patron, alors que le lien *dégrade* ne relie que les points forts dégradés par le patron abîmé. Ainsi, pour montrer au concepteur en quoi l'injection du patron est pertinente, il suffit de suivre le lien *dégrade* pour lister les points forts améliorés.

II.2.2. Utilisation

Ainsi enrichie, la place du patron *Composite* dans l'ontologie est illustrée dans la figure 4.13. Pour des soucis de lisibilité, la partie de l'ontologie concernant l'intention a été tronquée.

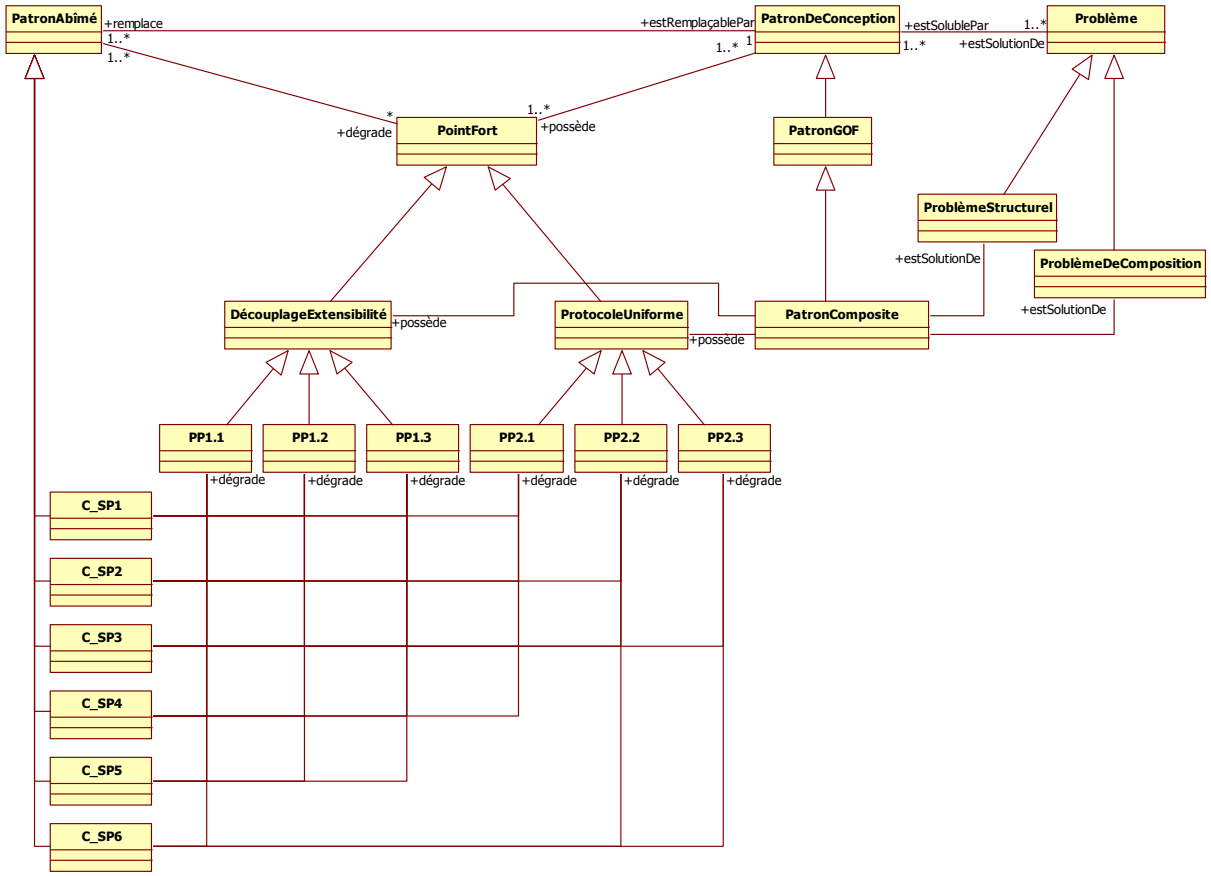


Figure 4.13 : Le patron *Composite* dans l'ontologie étendue

Les noms des *PointForts* et des *PatronAbîmés* sont référencés par le tableau 4.4.

Symboles	Nom complet
PP1.1	Factorisation maximale de la composition.
PP1.2	L'ajout ou la suppression d'une *Feuille* ne nécessite pas de modification de code.
PP1.3	L'ajout ou la suppression d'un *Composite* ne nécessite pas de modification de code.
PP2.1	Protocole uniforme sur les opérations des objets composés.
PP2.2	Protocole uniforme sur la gestion de la composition.
PP2.3	Point d'accès unique pour la classe client.
C_SP1	Développement de la composition sur <<*Composant*>>.
C_SP2	Développement de la composition sur <<*Composant*>> et sur <<*Composite*>>.
C_SP3	Composition récursive.
C_SP4	Développement de la composition sur <<*Composite*>> sans conformité de protocole.
C_SP5	Développement de la composition sur <<*Composite*>>.
C_SP6	Composition indirecte sur <<*Composite*>>.

Tableau 4.4 : Détail des symboles de la figure 4.13

Cette ontologie étendue nous permet de construire les questions à poser aux concepteurs. Grâce à des requêtes SPARQL permettant de se déplacer dans une ontologie de la même manière que dans une base de données SQL [Prud'hommeaux08], nous pouvons connaître l'intention d'un fragment détecté en utilisant simplement le nom du patron abîmé permettant d'engendrer ce fragment. De la même manière, en utilisant le nom du patron abîmé détecté, nous pouvons lister les points forts perturbés. Grâce à des « templates de question » permettant l'adaptation à plusieurs langues par exemple, nous communiquons avec le concepteur dans un langage naturel.

La troisième étape de l'activité est donc outillée pour communiquer avec le concepteur et pour lui expliquer l'intérêt des corrections effectuées. En associant l'ontologie au générateur de requêtes de détection et d'opérations de transformation, chaque étape de l'activité a été concrétisée. Afin de valider de la manière la plus précise possible, et pour proposer un déploiement en situation réelle de notre approche, nous avons développé un outil permettant d'exécuter automatiquement notre activité de revue de conception.

III. Validation sur des cas concrets

Afin d'automatiser notre activité de revue de conception, nous avons développé un outil nommé *Triton*. Alors qu'en mythologie grecque, Triton est la divinité marine descendant du Dieu Triton, représentée avec un corps d'homme barbu et une queue de poisson, tirant le char des dieux de la mer, en astronomie, Triton est le plus gros et le premier satellite de Neptune [Planete09]. C'est pour cette dernière définition que nous avons nommé ainsi notre outil. En effet, pour être opérationnel, Triton s'appuie sur la plate-forme Neptune [Millan08] et son interprète de requêtes OCL [Millan09]. Afin de travailler sur des modèles décrits dans des fichiers XMI, Triton utilise l'analyseur XMI de Neptune, en plus d'utiliser l'interprète OCL de la plate-forme.

Pour valider Triton, et par la même occasion notre approche de détection et de substitution de patrons abîmés, nous avons réalisé un jeu de tests dont l'objectif était d'éprouver notre approche sur des cas concrets, c'est-à-dire sur des modèles industriels, pour vérifier si des fragments étaient détectables en conditions réelles.

Dans cette section, nous présentons dans un premier temps le mode opératoire que nous avons suivi pour réaliser les tests. Nous montrons qu'en raison de difficultés à trouver des modèles industriels pertinents, nous avons été contraints de travailler en rétroconception. Nous terminons cette section en analysant les résultats issus de l'exécution de Triton sur les modèles rétroconçus. Afin d'améliorer la lisibilité des tableaux statistiques, nous identifions chaque patron abîmé par un symbole. Le tableau 4.5 présente la relation entre ces symboles et les patrons abîmés.

Symbole utilisé	Patron de conception	Nom du patron abîmé
C_SP1	Composite	Développement de la composition sur <<Composant>>
C_SP2	Composite	Développement de la composition sur <<Composant>> et sur <<Composite>>
C_SP3	Composite	Composition récursive
C_SP4	Composite	Développement de la composition sur <<Composite>> sans conformité de protocole
C_SP5	Composite	Développement de la composition sur <<Composite>>
C_SP6	Composite	Composition indirecte de la composition sur <<Composite>>
D_SP1	Décorateur	Développement sur <<Décorateur>>
D_SP2	Décorateur	Développement sur <<Décorateur>> sans conformité de protocole
D_SP3	Décorateur	Développement sur <<Composant>>
D_SP5	Décorateur	Mauvais découplage
D_SP6	Décorateur	Mauvais découplage sans possibilité d'extension sur <<Composant>>
D_SP7	Décorateur	Non conformité entre décorateurs et objets à décorer
P_SP1	Pont	Développement du pont

Tableau 4.5 : Table de correspondance des symboles des patrons abîmés

Les patrons abîmés D_SP4, P_SP2 et P_SP3 n'ont pas été intégrés à ce tableau car ils ne sont pas détectables en l'état actuel de notre approche.

III.1. Mode opératoire

Notre objectif étant de nous confronter à des modèles industriels, nous avons cherché des modèles à analyser sur des projets existants. Par delà les problèmes imposés par la compatibilité des fichiers XMI générés, et à l'exception de deux modèles, nous n'avons pas trouvé de projets industriels avec des modèles exploitables. Deux modèles n'étant pas suffisants pour véritablement éprouver notre approche, nous nous sommes résolus à rétroconcevoir des projets libres.

Pour ce faire, nous avons utilisé le module de rétroconception Java d'ArgoUML, ce dernier étant capable de générer du XMI lisible par Neptune. Le problème lors d'une rétroconception de code concerne les associations entre classes. Il est très difficile pour un logiciel de rétroconception de connaître quelle variable doit être considérée comme attribut, association, composition… Le module d'ArgoUML permet cependant d'imposer que tous les attributs soient transformés en associations, mais sans pouvoir distinguer ni leur navigabilité ni leur agrégation. Ainsi, en raison de cette contrainte, bon nombre de patrons abîmés sont devenus indétectables puisque certaines de leurs particularités structurelles intègrent des contraintes de navigabilité et d'agrégation.

De manière à ce que nous puissions tout de même effectuer des détections sur des modèles rétroconçus, nous avons pris le parti de modifier, pour ce jeu de test, les particularités structurelles de certains patrons. Ainsi, en utilisant le profil, nous avons rendu facultatifs les attributs de navigabilité et d'agrégation de tous nos patrons abîmés. Cette modification n'altère pas vraiment les particularités des patrons abîmés, puisque nous relâchons simplement une contrainte.

III.2.Analyse et résultats globaux

Les résultats de nos analyses sont présentés dans le tableau 4.6.

Modèle	ArgoUML	JHotDraw	JRefactory	Neptune	X1	X2
Nombre d'instances de métaclasses	79808	316977	672662	28011	230999	798539
Nombre de classes	2476	1749	1843	904	201	202
Nombre d'associations	1971	388	1498	433	6	6
Nombre de généralisations	1459	330	993	334	193	193
Temps d'analyse en minutes	28	7	4	14	0,01	0,01
Nombre de fragments identifiés	41	10	28	5	0	0
Nombre de fragments incomplets	2431	72	220	168	0	0
Nombre de fragments alternatifs	0	0	0	1 P_SP1	0	0
Nombre de mauvaises intentions	41	10	28	7	0	0

Tableau 4.6 : Résultat des analyses des modèles industriels

Il est important de préciser que la ligne indiquant les temps d'exécution n'est qu'indicative puisque Neptune intègre un système de cache qui accélère la détection au fur et à mesure de l'exécution des requêtes. Cependant, l'effet filtrant des requêtes se retrouve dans ces temps d'exécution, puisque des modèles avec très peu d'*Association*, et donc très peu de fragments candidats, ont été analysés en des temps inférieurs à la minute.

Nous n'avons pas pu identifier de manière significative des fragments alternatifs, signes de mauvaises pratiques de conception. Cependant, le fait que ces modèles aient été rétroconçus a altéré leur pertinence. Le nombre de fragments incomplets pour chacun de ces modèles met en évidence leur faiblesse structurelle. En ce qui concerne les modèles X1 et X2, il est possible de remarquer le nombre d'associations est beaucoup trop faible pour être significatif. Il est particulièrement surprenant que des entreprises élaborent un modèle de conception avec seulement six associations pour deux cents classes. Ces deux modèles n'ont donc présenté aucun intérêt en l'état. Après avoir analysé les fragments complets, nous n'avons trouvé qu'un seul fragment que nous considérons comme réellement alternatif. Tous les autres avaient une intention différente de celle du patron.

À l'issue de nos tests sur des modèles industriels, nous pouvons conclure que Triton est capable d'analyser en détail et de présenter des résultats cohérents dans des délais plus que raisonnables, qu'il s'agisse de petits modèles ou de plus gros dépassant les sept cent mille métaclasses. En effet, grâce à notre approche par filtrages successifs, le temps de détection n'est pas directement lié au nombre d'instances de métaclasses, mais au nombre de fragments potentiellement identifiables. Ainsi, nous pouvons dire que nous proposons une activité de revue de conception efficace, visant à corriger, par l'utilisation de patrons, des mauvaises pratiques de conception.

IV. Conclusion

En réunissant sur un même modèle l'intégralité des informations nécessaires à la détection d'un patron abîmé, nous avons pu développer un générateur de requêtes OCL. En considérant le patron abîmé comme un graphe, le générateur analyse tous les chemins possibles entre chaque sommet et les convertit en représentation arborescente dont chaque nœud représente une métaclasse du patron abîmé. Ainsi, chaque arbre décrit les métaclasses à traverser pour se déplacer d'un participant à un autre. Le patron abîmé étant décrit avec le profil UML précisant les liens interdits, le générateur mémorise au travers des arbres, les chemins autorisés et interdits. Cette transformation du modèle en arbre nous permet de générer des sous-requêtes permettant, lors de l'exécution de l'activité, d'identifier, dans le modèle à analyser, si des arbres similaires sont présents. Grâce à un algorithme de couplage de ces sous-requêtes, le résultat de chacune d'entre elles provoque l'identification d'un fragment.

Ce générateur nous permet ainsi de construire des requêtes de détection particulièrement complexes, intégrant la totalité de nos concepts de particularités structurelles locales et globales, ainsi que de liens autorisés, facultatifs et interdits. En complément, le générateur est également capable de construire les opérations d'injection de patron de conception. En comparant structurellement un patron abîmé à son patron de conception, le générateur met en évidence leurs différences structurelles, qu'il intègre dans les opérations de transformation. Ainsi, la transformation d'un fragment alternatif en contextualisation d'un patron de conception se résume à la correction des différences structurelles.

Intégrées dans Triton, notre outil permettant l'exécution de la revue de conception, ces requêtes concrétisent la première et la dernière étape de l'activité. En ce qui concerne la deuxième étape, consistant à communiquer avec le concepteur, nous avons utilisé une ontologie existante orientée sur l'intention des patrons de conception. Nous l'avons utilisée et étendue pour qu'elle nous permette de construire des questions à poser au concepteur. Ces questions vérifient que l'intention des fragments identifiés est conforme à celle du patron abîmé et présente au concepteur les avantages d'un remplacement de ces fragments par les patrons de conception adéquats. Nous avons ainsi étendu l'ontologie existante en y ajoutant le concept de points forts perturbés par le patron abîmé. Nous pouvons donc montrer au concepteur quels sont les points forts dont il peut bénéficier dans son modèle.

Conclusion

Synthèse

Identifier dans un modèle, a posteriori, des défauts de conception issus d'un manque d'utilisation de patrons de conception constitue le principal intérêt de notre approche. Nous avons alors développé une technique qui permet d'assister le concepteur dans l'utilisation des patrons et ce même s'il n'a pas eu le réflexe de les exploiter pendant sa conception. Pour ce faire, nous recherchons des défauts de conception, par le biais de patrons abîmés. Grâce à des requêtes de détection générées automatiquement à partir de la donnée d'un patron abîmé, nous sommes à même de cibler des fragments de modèle. Ces requêtes suivent un algorithme de détection basé sur des concordances structurelles. Les fragments ainsi détectés peuvent être, après un dialogue avec le concepteur, remplacés par les contextualisations des patrons de conception adéquats.

Un patron abîmé a le même niveau de généricité qu'un patron de conception et représente donc une famille génératrice de solutions alternatives à un type de problème dont un patron de conception propose la meilleure solution. Ces solutions alternatives sont structurellement différentes et présentent moins d'avantages, en termes de bonnes pratiques, que la solution optimale préconisée par le patron. Nous avons caractérisé les solutions alternatives par l'ensemble des points forts qu'elles dégradent. Les points forts d'un patron de conception expriment les critères d'architecture et les facteurs de qualité logicielle apportés par son utilisation. Ces critères explicitent en quoi un patron de conception constitue la meilleure solution pour un type de problème, et nous permettent de classifier les patrons abîmés selon leur degré de dégradation.

L'intégralité des patrons abîmés que nous avons identifiés sont issus de la décontextualisation de solutions alternatives collectées auprès de concepteurs particuliers. Ces derniers ont été choisis pour leurs connaissances académiques en modélisation par objets, n'incluant pas, au temps de la collecte, les patrons de conception. Nous avons ainsi obtenu un large éventail de solutions alternatives. Nous disposons alors d'une base de patrons abîmés pouvant être utilisée en complément du GoF, constituant ainsi un catalogue de mauvaises pratiques de conception.

Afin d'identifier les fragments de modèles caractéristiques de ces mauvaises pratiques de conception, nous avons développé un algorithme capable de détecter toutes les contextualisations de patrons abîmés. Cet algorithme recherche exactement un patron abîmé en autorisant des classes supplémentaires structurellement concordantes à l'architecture du patron abîmé. Grâce à des filtrages successifs, cet algorithme est rapide et efficace. De plus il est capable de gérer les relations interdites et facultatives entre les classes des fragments. En utilisant un profil UML, nous avons complété la description des patrons abîmés afin de générer de manière automatique les requêtes de détection. Ainsi, tout nouveau patron abîmé ajouté à notre base devient immédiatement détectable.

Nous avons également conçu et outillé une activité de revue de conception, au même titre qu'il existe des activités de revue de code. Cette activité analyse le modèle d'un concepteur, cherche les fragments caractéristiques, et le cas échant, les transforme pour y intégrer un patron de conception. De par le dialogue que notre outil établit avec le concepteur, ce dernier peut gérer de manière semi-automatique le déroulement de l'activité tout en ayant conscience des qualités supplémentaires apportées à son modèle. Notre outil peut être déployé dans tout processus où les modèles jouent un rôle prépondérant.

Perspectives de recherche

Notre procédé de décontextualisation est intimement lié à notre méthode de détection. Chaque classe d'une solution alternative est associée à un participant du patron, ce qui permet à la détection d'être efficace. Afin d'étendre notre base de patrons abîmés à d'autres types de patron, il est nécessaire de reconsidérer ce procédé afin de gérer plus finement le partage des responsabilités. Ainsi, une classe pourrait correspondre à plusieurs participants et inversement. En procédant de la sorte nous serions à même de décontextualiser des solutions alternatives aux patrons comportementaux et composites. De fait, nos patrons sont décrits avec un seul participant par responsabilité. Nous ne décrivons pas comment, dans un fragment, deux participants ayant les mêmes responsabilités doivent être interconnectés. En ajoutant une description supplémentaire à chacun des participants, nous pourrions envisager de décrire les liens obligatoires, interdits et facultatifs entre deux participants identiques. Cette extension de la description des patrons abîmés complexifie la génération automatique des requêtes de détection et de transformation.

Cette complexité peut être réduite en remettant en cause le choix d'OCL, initialement établi afin de permettre à un concepteur d'adapter la requête à ses propres besoins. Il est donc nécessaire de modifier notre implémentation de la détection, pour obtenir un équilibre entre la lisibilité des requêtes et leur génération automatique. L'accès à la forme interne des métamodèles et des modèles par la plate-forme Neptune devrait permettre d'atteindre raisonnablement cet objectif.

Enfin, le passage à l'échelle de notre activité de revue de conception constitue une perspective nécessaire. Pour parer les difficultés à obtenir des modèles industriels pertinents, il serait intéressant de proposer notre outil à des équipes de développement. De même, le déploiement de notre site web collaboratif nous permettra la constitution d'une communauté. Nous serons ainsi à même de proposer un processus de collecte à grande échelle permettant l'établissement d'un consensus sur les patrons abîmés au même titre que pour les patrons de conception.

Bibliographie

[Albin-Amiot01_a] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, N. Jussien, "**Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together**", dans : *proceedings of the 16th conference on Automated Software Engineering (ASE),* IEEE Computer Society Press, pages 166-173, 2001.

[Albin-Amiot01_b] H. Albin-Amiot, Y.-G. Guéhéneuc, "**Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis**", dans : *proceedings of the 1st ECOOP workshop on Automating Object-Oriented Software Development Methods*, University of Twente, 2001.

[Alexander77] C. Alexander, S. Ishikawa, M. Silverstein, "**A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)**", Oxford University Press, 1977.

[Ambler98] S.W. Ambler, "**Process Patterns: Building Large-Scale Systems Using Object Technology**", Cambridge University Press, 1998.

[Ammour06] S. Ammour, "**Support des patrons de conception dans les outils UML**", thèse, supervision Mikal ZIANE, soutenue le 11/10/2006.

[Antoniol01] G Antoniol, G. Casazza, M.D. Penta, R. Fiutem, "**Object-oriented design patterns recovery**", dans : *Journal of Systems and Software*, Elsevier Science Inc., volume 59, numéro 2, pages 181-196, 2001.

[Bacchus98] F. Bacchus, P Van Beek, "**On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems**", dans : *proceedings of the 15th National Conference on Artificial Intelligence (AAAI) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI)*, AAAI Press, pages 311-318, 1998.

[Balanyi03] Z. Balanyi, R. Ferenc, "**Mining Design Patterns from C++ Source Code**", dans : *procedings of the 19th IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, pages 305-314, 2003.

[Baroni03] A.L. Baroni, Y.-G. Guéhéneuc, H. Albin-Amiot, "**Design patterns formalization**", rapport de recherche, Département d'informatique, École des Mines de Nantes, numéro 03/03/INFO, 2003.

[Beck87] K. Beck, W. Cunningham, "**Using Pattern Languages for Object Oriented Programs**", rapport de recherche, Computer Research Laboratory, Tektronix, numéro CR-87-43, note : *submitted to the OOPSLA'87 workshop on the Specification and Design for Object-oriented Programming*, 1987.

[Bengoetxea02] E. Bengoetxea, "**Inexact Graph Matching Using Estimation of Distriburion Algorithms**", thèse, Ecole Nationale Supérieure des Télécommunications, France, 2002.

[Bergenti00] F. Bergenti, A. Poggi, "**Improving UML Designs Using Automatic Design Pattern Detection**", dans : *proceedings of 12th International Conference on Software Engineering and Knowledge Engineering,* pages 336-343, 2000.

[Blondel04] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, P. Van Dooren, "**A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching**", dans : *review Society for Industrial and Applied Mathematics (SIAM)*, Society for Industrial and Applied Mathematics, volume 46, numéro 4, pages 647-666, 2004.

[Booch97] G. Booch, "**Ingénierie du logiciel avec ADA: de la conception à la realisation**", InterEditions, 1997.

[Borne99] I. Borne, N. Revault, "**Comparaison d'outils de mise en oeuvre de Design Patterns**", dans : *L'Objet*, volume 5, numéro 2, pages 243-266, 1999.

[Bouhours06_a] C. Bouhours, H. Leblanc, C. Percebois, "**Structural Variants Detection for Design Pattern Instantiation**", dans : *International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE)*, IEEE Computer Society, on line, 2006.

[Bouhours06_b] C. Bouhours, H. Leblanc, "**Des méta-modèles au banc d'essai des patrons de conception (Ré)utilisation et Intégration**", dans : *actes des Journées sur l'Ingénierie Dirigée par les Modèles*, Model Driven Engineering (MDE), pages 239-244, 2006.

[Bouhours07_a] C. Bouhours, H. Leblanc, C. Percebois, "**Alternative Models for a Design Review Activity**", dans : *proceedings of the 2nd workshop on Quality in Modeling (QM) in conjonction with ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer, pages 65-79, 2007.

[Bouhours07_b] C. Bouhours, H. Leblanc, C. Percebois, "**Towards a knowledge base to improve reusability of design pattern**", dans : *proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT)*, Institute for Systems and Technologies for Information, Control and Communication, volume 2, pages 421-424, 2007.

[Bouhours07_c] C. Bouhours, H. Leblanc, C. Percebois, "**Alternative Models for Structural Design Patterns**", rapport de recherche, IRIT, numéro IRIT/RR--2007-1--FR, 2007.

[Bouhours08] C. Bouhours, "**Une activité de revue de design pour améliorer les modèles à objets**", dans : *1er Workshop Paterns & Architectures*, conférencier invité, 2008.

[Bouhours09] C. Bouhours, H. Leblanc, C. Percebois, "**Bad smells in design and design patterns**", dans : *Journal of Object Technology*, ETH Swiss Federal Institute of Technology, volume 8, numéro 3, pages 43-63, 2009.

[Bray06] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, Yergeau, François, "**Extensible Markup Language (XML) 1.0**", Fourth Edition, 2006.

[Brown96] K. Brown, "**Design reverse-engineering and automated design-pattern detection in Smalltalk**", rapport de recherche, North Carolina State University, numéro TR-96-07, 1996.

[Brown98] W.J. Brown, R. C. Malveau, T.J. Mowbray, "**AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**", Wiley, 1998.

[Buschmann96] F. Buschmann, R. Meunier, H. Rohnert,P. Sommerlad, M. Stal, "**Pattern-Oriented Software Architecture**", John Wiley & Sons, 1996.

[Chikofsky90] E.J. Chikofsky, J.H. Cross, "**Reverse Engineering and Design Recovery: A Taxonomy**", dans : *IEEE Software*, IEEE Computer Society Press, volume 7, numéro 1, pages 13-17, 1990.

[Coad95] P. Coad, "**Object Models – strategies, patterns and applications**", Prentice Hall, 1995.

[Conte01] A. Conte, M. Fredj, J.-P. Giraudin, D. Rieu, "**P-Sigma : un formalisme pour une représentation unifiée de patrons**", dans : *actes du 19ième congrès Informatique des Organisations et Systèmes d'Information et de Décision (INFORSID)*, pages 67-86, 2001.

[Coplien96] J.O. Coplien, "**Patterns, the Patterns White Paper**", 1996.

[Costagliola05] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, M. Risi, "**Design Pattern Recovery by Visual Language Parsing**", dans : *proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society, pages 102-111, 2005.

[Dietrich05] J. Dietrich, C. Elgar, "**A Formal Description of Design Patterns Using OWL**", dans : *proceedings of the 16th Australian Software Engineering Conference*, IEEE Computer Society, pages 243-250, 2005.

[Dodani06] M. Dodani, "**Patterns of Anti-Patterns ?**", dans : *Journal of Object Technology*, volume 5, numéro 5, pages 29-33, 2006.

[Dong07] J. Dong, Y. Zhao, "**Classification of Design Pattern Traits**", dans : *proceedings of the 19th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 473-477, 2007.

[Eden97] A.H. Eden, A. Yehudai, J. Gil, "**Precise specification and automatic application of design patterns**", dans : *proceedings of the 12th international conference on Automated Software Engineering (ASE)*, IEEE Computer Society, pages 143-152, 1997.

[ElBoussaidi04] G. El Boussaidi, H. Mili, "**Les patrons de conception : représentation et mise en oeuvre**", rapport de recherche, LATECE, mars 2004.

[ElBoussaidi08] G. El-Boussaidi, H. Mili, "**Detecting Patterns of Poor Design Solutions Using Constraint Propagation**", dans : *proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag, volume 5301, pages 189-203, 2008.

[Fagan02] M. Fagan, "**Design and code inspections to reduce errors in program development**", dans : *Software pioneers: contributions to software engineering*, Springer-Verlag New York, pages 575-607, 2002.

[Fowler96] M. Fowler, "**Accounting Patterns**", dans : *Analysis Patterns: Reusable Object Models*, Addison-Wesley Object Technology Series, 1996.

[Fowler97] M. Fowler, "**Analysis patterns: reusable objects models**", Addison Wesley Longman Publishing Co, 1997.

[Fowler99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "**Refactoring: Improving the Design of Existing Code**", Addison-Wesley Professional, 1999.

[France03] R. France, S. Ghosh, E. Song, D.-K. Kim, "**A Metamodeling Approach to Pattern-Based Model Refactoring**", dans : *IEEE Software*, IEEE Computer Society Press, volume 20, numéro 5, pages 52-58, 2003.

[FUJABA05] FUJABA, From UML to Java and Back Again, http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/index.html, 2005.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "**Design Patterns: Elements of Reusable Object-Oriented Software**", Addison Wesley Professional, 1995.

[GOPROD08] http://www.irit.fr/recherches/DCL/MACAO/GOPROD/, 2008.

[Gueheneuc04] Y.-G. Gueheneuc, H. Sahraoui, F Zaidi, "**Fingerprinting Design Patterns**", dans : *proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society, pages 172-181, 2004.

[Guennec00] A.L. Guennec, G. Sunyé, J.-M. Jézéquel, "**Precise Modeling of Design Patterns**", dans : *proceedings of 3rd International Conference on the Unified Modeling Language (UML)*, Springer Verlag, pages 482-496, 2000.

[Habel96] A. Habel, R. Heckel, et G. Taentzer, "**Graph grammars with negative application conditions**", dans : *Fundamenta Informaticae Journal*, IOS Press, volume 26, numéro 3-4, pages 287-313, 1996.

[Harb08] D. Harb, C. Bouhours, H. Leblanc, "**Using an ontology to suggest software design patterns integration**", dans : *proceedings of the 1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE) on International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, CEUR Workshop Proceedings, volume 395, on line, 2008.

[Harb09] D. Harb, C. Bouhours, H. Leblanc, "**Using an Ontology to Suggest Design Patterns Integration**", best paper award, dans : *Workshops and Symposia at MoDELS*, Springer-Verlag, volume 5421, numéro 5421, pages 318-331, 2009.

[Heuzeroth03] D. Heuzeroth, T. Holl, G. Högström, W. Löwe, "**Automatic Design Pattern Detection**", dans : *proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC)*, IEEE Computer Society, pages 94-104, 2003.

[Huston01] B. Huston, "**The effects of design pattern application on metric scores**", dans : *Journal of Systems and Software*, Elsevier Science Inc., volume 58, numéro 3, pages 261-269, 2001.

[JOfficiel87] Journal officiel français du 19 février 1984.

[Kampffmeyer07] H. Kampffmeyer, S. Zschaler, "**Finding the Pattern You Need: The Design Pattern Intent Ontology.**", dans : *proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Springer, volume 4735, pages 211-225, 2007.

[Krasner88] G. E. Krasner, S. T. Pope, "**A cookbook for using the model-view controller user interface paradigm in Smalltalk-80**", dans : *Journal of Object-Oriented Programming (JOOP)*, SIGS Publications, volume 1, numéro 3, pages 26-49, 1988.

[Larman00] G. Larman, "**Applying UML on Patterns**", Prentice Hall, seconde édition, 2000.

[Mak04] J.K.H. Mak, C.S.T. Choy, D.P.K. Lun, "**Precise Modeling of Design Patterns in UML**", dans : *proceedings of the 26th International Conference on Software Engineering (ICSE),* IEEE Computer Society, pages 252-261, 2004.

[McGuinness04] D.L. McGuinness, F.V. Harmelen, "**OWL Web Ontology Language Overview**", http://www.w3c.org/TR/owl-features/, 2004.

[Mili05] H. Mili, G. El-Boussaidi, "**Representing and Applying Design Patterns: What Is the Problem?**", dans : *proceedings of the 8th international conference on Model Driven Engineering Languages and Systems (MoDELS),* pages 186-200, 2005.

[Millan08] T. Millan, L. Sabatier, P. Bazex, C. Percebois, "**NEPTUNE II Une plate-forme pour la vérification et la transformation de modèles**", dans : *journal Génie Logiciel,* GL & IS, volume 85, pages 30-34, 2008.

[Millan09] T. Millan, L. Sabatier, T. T. Le Thi, P. Bazex, C. Percebois C, "**An OCL extension for checking and transforming UML Models**", dans : *proceedings of the 8th International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS),* WSEAS Press, pages 144-150, 2009.

[Niere02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, J. Welsh, "**Towards pattern-based design recovery**", dans : *proceedings of the 24th International Conference on Software Engineering (ICSE),* ACM Press, pages 338-348, 2002.

[Noy01] N. F. Noy, D. L. McGuinness, "**Ontology Development 101: A Guide to Creating Your First Ontology**", rapport de recherche, Stanford University, numéro KSL-01-05, 2001.

[O'Cinnéide99] M. O'Cinnéide, P. Nixon, "**A Methodology for the Automated Introduction of Design Patterns**", dans : *proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM),* IEEE Computer Society, pages 463-473, 1999.

[OMG03] Object Management Group., "Unified Modeling Language", http://www.omg.org/spec/UML/1.5/PDF/index.htm, 2003.

[OMG06] Object Management Group., "Object Constraint Language", http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, 2006.

[OMG07] Object Management Group., "XML Metadata Interchange", http://www.omg.org/technology/xml/index.htm, 2007.

[Planete09] Planète Astronomie, http://www.planete-astronomie.com/, 2009.

[Prud'hommeaux08] E. Prud'hommeaux, A. Seaborne, "**SPARQL Query Language for RDF**", rapport de recherche, World Wide Web Consortium, 2008.

[Riehle97_a] D. Riehle, "**Bureaucracy**", dans : *Pattern languages of program design 3,* Addison-Wesley Longman, pages 163-185, 1997.

[Riehle97_b] D. Riehle, "**Composite design patterns**", dans : *proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA),* pages 218-228, 1997.

[Rudolf00] M. Rudolf, "**Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching**", dans : *selected papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT),* Springer-Verlag, pages 238-251, 2000.

[Sunyé01] G. Sunyé, D. Pollet, Y. L. Traon, J.M. Jézéquel, "**Refactoring UML models**", dans : *proceedings of the 4th International Conference on the Unified Modeling Language (UML),* Springer, pages 134-148, 2001.

[Tsantalis06] N. Tsantalis, S. T. Halkidis, "**Design Pattern Detection Using Similarity Scoring**", dans : *IEEE Transactions on Software Engineering,* IEEE Press, volume 32, numéro 11, pages 896-909, 2006.

[Ullmann76] J. R. Ullmann, "**An Algorithm for Subgraph Isomorphism**", dans : *journal of the ACM (JACM),* ACM, volume 23, numéro 1, pages 31-42, 1976.

[Wendehals03] L. Wendehals, "**Improving design pattern instance recognition by dynamic analysis**", dans : *proceedings of the workshop on Dynamic Analysis on the International Conference on Software Engineering (ICSE),* 2003.

[Wenzel05_a] S. Wenzel, "**Automatic detection of incomplete instances of structural patterns in UML class diagrams**", dans : *Nordic Journal of Computing,* Publishing Association Nordic Journal of Computing, volume 12, numéro 4, pages 379-394, 2005.

[Wenzel05_b] S. Wenzel, "**Detection of Incomplete Patterns Using FUJABA Principles**", dans : *proceedings of the 3rd International Fujaba Days 2005 : MDD in Practice,* pages 33-40, 2005.

Annexes

1

Sujets des expérimentations

I. Expérimentation de février 2006

IUP ISI Février 2006

Modélisation de problèmes courants avec les design patterns

Ce document propose un ensemble d'exercices de modélisation à objets. Vous devez produire un diagramme de classe UML illustrant les exercices posés. Chaque diagramme doit contenir suffisamment d'informations pour démontrer que le problème est résolu (attributs, méthodes, relations, stéréotypes…), et éventuellement être complété par d'autres diagrammes (séquences, collaborations…).

Le but de ces exercices est que vous suiviez un raisonnement qui vous est propre, ces modélisations peuvent s'envisager de plusieurs manières différentes. Ne cherchez pas de solution commune avec vos camarades, ni de solutions sur Internet ou dans des ouvrages de conception.

Certaines modélisations sont présentées avec des évolutions probables. Vos conceptions doivent être structurées de manière à ce que ces évolutions soient facilement intégrables. Faites apparaître ces évolutions dans vos diagrammes.

Exercice 1

Modéliser un éditeur de dessin.

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.

Exercice 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 à 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.

Exercice 3

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.

Exercice 4

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.

Exercice 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 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.

Exercice 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 le style de mise en forme de celui-ci. 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 à un texte, un autre jeu de caractères.

Exercice 7

Modéliser le chargement et l'affichage d'un éditeur de documents.
Un document est composé de textes et d'images. La contrainte de cet éditeur est 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.

Pour aller plus loin … Exercice 8

Modéliser un jeu de labyrinthe.
Le jeu du labyrinthe consiste à parcourir un dédale à la recherche de la sortie. Un dédale est un ensemble de salles, chaque salle ayant 4 murs et une ou plusieurs portes, une porte reliant forcément deux salles entre elles. Il existe deux types de dédales, un dédale simple (avec salles, murs et portes) et un dédale piégé (avec salles explosives, portes fermées et murs). Au lancement du jeu, un tirage aléatoire décide quel type de dédale va être utilisé. Il est probable que le jeu évolue pour que l'on puisse utiliser un dédale enchanté (avec salles paradisiaques, murs décorés et portes).

II. Expérimentation de novembre 2006

M2 SLCP – module CAL Février 2006

Trouver des solutions à des problèmes de conception récurents

Ce document propose un ensemble d'exercices de modélisation à objets. Vous devez produire un diagramme de classe UML illustrant les exercices posés. Chaque diagramme doit contenir suffisamment d'informations pour démontrer que le problème est résolu (attributs, méthodes, relations, stéréotypes…), et éventuellement être complété par d'autres diagrammes (séquences, collaborations…).

Le but de ces exercices est que vous suiviez un raisonnement qui vous est propre, ces modélisations peuvent s'envisager de plusieurs manières différentes. Ne cherchez pas de solution commune avec vos camarades, ni de solutions sur Internet ou dans des ouvrages de conception.

Certaines modélisations sont présentées avec des évolutions probables. Vos conceptions doivent être structurées de manière à ce que ces évolutions soient facilement intégrables. Faites apparaître ces évolutions dans vos diagrammes.

Exercice 1

Modéliser un éditeur de dessin.

Un dessin est composé de formes graphiques (lignes, rectangles et rosaces), placées à des positions précises. Chaque forme graphique doit être modélisée par une classe mettant à disposition une méthode *dessiner() : void*. Une rosace est une forme complexe dessinée par une classe boîte-noire* fournie. Cette classe effectue ce dessin en mémoire, et y donne accès grâce à une méthode *getRosace() : int* qui retourne l'adresse du dessin. Il est probable que le système évolue pour que l'on puisse dessiner des cercles.

* dont on ne dispose pas du code, mais dont on connaît l'interface. Cette classe est donc non modifiable et ne peut pas hériter d'une autre classe.

Exercice 2

Modéliser un système permettant d'afficher des fenêtres vides (sans bouton, sans menu…) à l'écran.

Une fenêtre peut avoir plusieurs styles différents dépendant de la plate-forme d'utilisation. On considère qu'il existe deux plates-formes, XWindow et PresentationManager. Le code client doit pouvoir être écrit indépendamment et sans connaissance à priori de la future plate-forme d'exécution. Il est probable que le système évolue pour que l'on puisse spécialiser ces fenêtres en fenêtres applicatives (capables de gérer des applications) et en fenêtres iconisées (avec une icône).

Exercice 3

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.

Exercice 4

Modéliser un système permettant d'afficher des objets visuels à l'écran.
Un objet visuel peut être un texte ou une image. En cas de besoin, le système doit permettre d'ajouter à ces objets une barre de défilement verticale, une barre de défilement horizontale, et une bordure (tous ces ajouts sont cumulables). Il est probable que le système évolue pour que l'on puisse ajouter aux objets visuels une barre de menu.

Exercice 5

Modéliser la structure du 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 graphique. Le contenu du moteur de jeu est de moindre importance.

Exercice 6

Modéliser la fonction d'affichage d'un éditeur de documents.
Un document est composé de lignes, de colonnes, et de caractères. L'intersection d'une ligne et d'une colonne donne l'emplacement où doit s'afficher le caractère ainsi que le style de mise en forme de celui-ci. Il est important d'éviter la prolifération d'objets ayant le même état, ainsi, il ne sera pas acceptable d'avoir autant d'instances que de caractères dans le document. Il est probable que le système évolue pour que l'on puisse ajouter à un texte, un autre jeu de caractères (par exemple, les caractères arabes).

Exercice 7

Modéliser le chargement et l'affichage d'un éditeur de documents.
Un document est composé de textes et d'images. La contrainte de cet éditeur est 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.

Pour aller plus loin … Exercice 8

Modéliser un générateur de labyrinthe.
Le jeu du labyrinthe consiste à parcourir un dédale à la recherche de la sortie. Un dédale est un ensemble de salles, chaque salle ayant 4 murs et une ou plusieurs portes, une porte reliant forcément deux salles entre elles. Il existe deux types de dédales, un dédale simple (avec salles, murs et portes) et un dédale piégé (avec salles explosives, portes fermées et murs). Au lancement du jeu, un tirage aléatoire décide quel type de dédale va être utilisé. Il est probable que le jeu évolue pour que l'on puisse utiliser un dédale enchanté (avec salles paradisiaques, murs décorés et portes).

III. Expérimentation de décembre 2006

Licence Pro SIL - option qualité du logiciel Décembre 2006

Trouver des solutions à des problèmes de conception récurrents

Ce document propose un ensemble d'exercices de modélisation à objets. Vous devez produire un diagramme de classe UML illustrant les exercices posés. Chaque diagramme doit contenir suffisamment d'informations pour démontrer que le problème est résolu (attributs, méthodes, relations, stéréotypes…), et éventuellement être complété par d'autres diagrammes (séquences, collaborations…).

Le but de ces exercices est que vous suiviez un raisonnement qui vous est propre, ces modélisations peuvent s'envisager de plusieurs manières différentes. Ne cherchez pas de solution commune avec vos camarades, ni de solutions sur Internet ou dans des ouvrages de conception.

Certaines modélisations sont présentées avec des évolutions probables. Vos conceptions doivent être structurées de manière à ce que ces évolutions soient facilement intégrables. Faites apparaître ces évolutions dans vos diagrammes.

Exercice 1

Modéliser un éditeur de dessin.

Un dessin est composé de formes graphiques (lignes, rectangles et rosaces), placées à des coordonnées X et Y. Chaque forme graphique doit être modélisée par une classe mettant à disposition une méthode *dessiner() : void*. Une rosace est une forme complexe dessinée par une classe boîte-noire* fournie. Cette classe effectue ce dessin en mémoire, et y donne accès grâce à une méthode *getRosace() : int* qui retourne l'adresse du dessin. Il est probable que le système évolue pour que l'on puisse dessiner des cercles.

* dont on ne dispose pas du code, mais dont on connaît l'interface. Cette classe est donc non modifiable et ne peut pas hériter d'une autre classe.

Exercice 2

Modéliser un système permettant d'afficher des fenêtres vides (sans bouton, sans menu…) à l'écran.

Une fenêtre peut avoir plusieurs styles différents dépendant de la plate-forme d'utilisation. On considère qu'il existe deux plates-formes, XWindow et PresentationManager. Le code client doit pouvoir être écrit indépendamment et sans connaissance à priori de la future plate-forme d'exécution. Il est probable que le système évolue pour que l'on puisse spécialiser ces fenêtres en fenêtres applicatives (capables de gérer des applications) et en fenêtres iconisées (avec une icône).

Exercice 3

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. Chacune de ces formes graphiques est positionnée à des coordonnées X et Y.

Exercice 4

Modéliser un système permettant d'afficher des objets visuels à l'écran.

Un objet visuel peut être composé d'un texte ou d'une image. En cas de besoin, le système doit permettre d'ajouter à ces objets une barre de défilement verticale, une barre de défilement horizontale, et une bordure (tous ces ajouts sont cumulables). Le choix de ces ajouts peut-être effectué à l'exécution. Ces ajouts nécessitent une connaissance sur les objets visuels. Il est probable que le système évolue pour que l'on puisse ajouter aux objets visuels une barre de menu.

Exercice 5

Modéliser la communication d'un ensemble d'applications avec l'environnement d'exécution.
Une application peut avoir besoin de connaître, à tout instant, des propriétés de l'environnement. Si l'une des applications veut modifier une de ces propriétés, il faut être sûr qu'aucune autre application ne peut accéder à l'environnement à ce moment là.

Exercice 6

Modéliser la structure du 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 graphique. Le contenu du moteur de jeu est de moindre importance.

Exercice 7

Modéliser la fonction d'affichage d'un éditeur de documents.
Un document est composé de lignes, de colonnes, et de caractères. L'intersection d'une ligne et d'une colonne donne l'emplacement où doit s'afficher le caractère ainsi que le style de mise en forme de celui-ci. Il est important d'éviter la prolifération d'objets ayant le même état, ainsi, il ne sera pas acceptable d'avoir autant d'instances que de caractères dans le document. Ainsi, pour le jeu de caractères ASCII, on aura au maximum 128 instances (une par caractère). Il est probable que le système évolue pour que l'on puisse ajouter à un texte, un autre jeu de caractères (par exemple, les caractères arabes).

Exercice 8

Modéliser le chargement et l'affichage d'un éditeur de documents.
Un document est composé de textes et d'images. La contrainte de cet éditeur est 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.

Exercice 9

Modéliser les communications d'un avion à l'approche d'un aéroport.
Lorsqu'un avion est en approche de l'aéroport, il doit signaler à tous les autres avions qui sont autour de lui qu'il a l'intention de se poser, et attendre leurs confirmations avant d'effectuer la manœuvre. C'est la tour de contrôle de l'aéroport qui garantit la régulation du trafic aérien, en s'assurant qu'il n'y a pas de conflit de trajectoire ou de destination entre plusieurs avions.
En plus du diagramme de classes, représenter par une collaboration (diagramme de collaboration ou diagramme d'objets et de séquence) l'atterrissage d'un avion parmi deux voulant atterrir et un voulant décoller).

Exercice 10

Modéliser un afficheur graphique de données.
L'afficheur se base sur un ensemble de données pour dessiner les graphiques correspondants. Il affiche deux types de diagrammes, un camembert et un histogramme. Toute modification sur les données doit être automatiquement répercutée sur les diagrammes. Il est probable que le système évolue pour que les données soient en plus présentées dans un tableau.

Pour aller plus loin … Problème facultatif

Modéliser un générateur de labyrinthe.
Le jeu du labyrinthe consiste à parcourir un dédale à la recherche de la sortie. Un dédale est un ensemble de salles, chaque salle ayant 4 murs et une ou plusieurs portes, une porte reliant forcément deux salles entre elles. Il existe deux types de dédales, un dédale simple (avec salles, murs et portes) et un dédale piégé (avec salles explosives, portes fermées et murs). Au lancement du jeu, un tirage aléatoire décide quel type de dédale va être utilisé. Il est probable que le jeu évolue pour que l'on puisse utiliser un dédale enchanté (avec salles paradisiaques, murs décorés et portes).

IV. Expérimentation de janvier 2007

M2 - IUP ISI Janvier 2007

Trouver des solutions à des problèmes de conception

Ce document propose un ensemble d'exercices de modélisation à objets. Vous devez produire un diagramme de classe UML illustrant les exercices posés. Chaque diagramme doit contenir suffisamment d'informations pour démontrer que le problème est résolu (attributs, méthodes, relations, stéréotypes…), et éventuellement être complété par d'autres diagrammes (séquences, collaborations…).

Le but de ces exercices est que vous suiviez un raisonnement qui vous est propre, ces modélisations peuvent s'envisager de plusieurs manières différentes. Ne cherchez pas de solution commune avec vos camarades, ni de solutions sur Internet ou dans des ouvrages de conception.

Certaines modélisations sont présentées avec des évolutions probables. Vos conceptions doivent être structurées de manière à ce que ces évolutions soient facilement intégrables. Faites apparaître ces évolutions dans vos diagrammes.

Exercice 1

Modéliser les communications d'un avion à l'approche d'un aéroport.
Lorsqu'un avion est en approche de l'aéroport, il doit signaler à tous les autres avions qui sont autour de lui qu'il a l'intention de se poser, et attendre leur confirmation à tous avant d'effectuer la manœuvre. C'est la tour de contrôle de l'aéroport qui garantit la régulation du trafic aérien, en s'assurant qu'il n'y a pas de conflit de trajectoire ou de destination entre plusieurs avions.
En plus du diagramme de classes, représenter par une collaboration (diagramme de collaboration ou diagramme d'objets et de séquence) l'atterrissage d'un avion parmi deux voulant atterrir et un voulant décoller).

Exercice 2

Modéliser un afficheur graphique de données.
L'afficheur se base sur un ensemble de données pour dessiner les graphiques correspondants. Il affiche deux types de diagrammes, un camembert et un histogramme. Toute modification sur les données doit être automatiquement répercutée sur les diagrammes. Il est probable que le système évolue pour que les données soient en plus présentées dans un tableau.

Exercice 3

Modéliser la communication d'un ensemble d'applications avec l'environnement d'exécution.
Une application peut avoir besoin de connaître, à tout instant, des propriétés de l'environnement. Si l'une des applications veut modifier une de ces propriétés, il faut être sûr qu'aucune autre application ne peut accéder à l'environnement à ce moment là.

Exercice 4

Modéliser la structure du 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 graphique. Le contenu du moteur de jeu est de moindre importance.

Exercice 5

Modéliser une grammaire syntaxique.

Votre modèle doit permettre de représenter la grammaire suivante, les côtes ne faisant pas partie de la grammaire (Les littéraux peuvent être considérés comme des objets de type String) :

```
expression ::= littéral | alternative | séquence | répétition | '('expression')'
alternative ::= expression '|' expression
séquence ::= expression '&' expression
répétition ::= expression '*'
littéral ::= 'a'|'b'|'c'|…{'a'|'b'|'c'|…}*
```

En plus du diagramme de classes, montrer par un diagramme d'objet que votre modèle peut instancier : `bonjour & (madame | monsieur)*`

Exercice 6

Modéliser un ensemble hifi-vidéo

L'ensemble hifi-vidéo est composé d'une télévision et d'une chaîne HIFI séparée. Les deux appareils sont totalement indépendants, mais ils ont été fabriqués par le même constructeur, et répondent à des protocoles communs. Un utilisateur peut intervenir à distance sur ces appareils grâce à une télécommande (une par appareil) fabriquée par le même constructeur. La télécommande de la télévision permet uniquement de changer de chaîne (n chaînes mémorisées à un instant t), et celle de la chaîne HIFI permet uniquement de changer de station (10 stations mémorisées).

Exercice 7

Modéliser un éditeur de classes UML

Une classe est un objet graphique qui a une position (coordonnées X et Y) sur un diagramme. Tous les objets graphiques du diagramme sont déplaçables à la souris. L'éditeur a une fonctionnalité Undo/Redo qui permet d'annuler ou de reproduire un déplacement.

Exercice 8

Modéliser le fonctionnement d'une vidéothèque

La vidéothèque met à disposition de ses clients des DVD selon trois catégories : *Enfant*, *Normal* et *Nouveauté*. Un DVD est dans la catégorie *Nouveauté* pendant quelques semaines, puis passe dans l'une des autres catégories. Le prix des DVD dépend de la catégorie. Il est probable que le système évolue pour que la catégorie *Horreur* soit ajoutée.

Exercice 9

Modéliser le fonctionnement d'une vidéothèque (suite)

Lorsque le gérant fait ses comptes à la fin du mois, il souhaite récupérer les informations des clients qui ont acheté un DVD pendant le mois. Au choix, il peut afficher à l'écran ces informations au format texte ou HTML. On veillera à limiter au maximum les duplications de code. Il est probable que le système évolue pour que l'affichage soit aussi possible en XML.

Il n'est pas utile de récupérer le modèle de l'exercice précédent.

Exercice 10

Modéliser une application

Deux équipes de développement travaillent en parallèle sur cette application. L'une travaille sur le fonctionnel, l'autre sur l'interface graphique. Nous ne nous intéressons ici qu'aux menus. Modéliser la gestion des menus de cette application, de manière à ce que le travail des deux équipes reste indépendant. Les concepteurs de la partie fonctionnelle doivent implémenter les fonctionnalités des menus de la manière la plus simple possible.

V. Expérimentation de février 2007

L3 - IUP ISI Février 2007

Trouver des solutions à des problèmes de conception

Ce document propose un ensemble d'exercices de modélisation à objets. Vous devez produire un diagramme de classe UML illustrant les exercices posés. Chaque diagramme doit contenir suffisamment d'informations pour démontrer que le problème est résolu (attributs, méthodes, relations, stéréotypes…), et éventuellement être complété par d'autres diagrammes (séquences, collaborations…).

Le but de ces exercices est que vous suiviez un raisonnement qui vous est propre, ces modélisations peuvent s'envisager de plusieurs manières différentes. Ne cherchez pas de solution commune avec vos camarades, ni de solutions sur Internet ou dans des ouvrages de conception.

Certaines modélisations sont présentées avec des évolutions probables. Vos conceptions doivent être structurées de manière à ce que ces évolutions soient facilement intégrables. Faites apparaître ces évolutions dans vos diagrammes.

Exercice 1

Modéliser les communications d'un avion à l'approche d'un aéroport.
Lorsqu'un avion est en approche de l'aéroport, il doit signaler à tous les autres avions qui sont autour de lui qu'il a l'intention de se poser, et attendre leur confirmation à tous avant d'effectuer la manœuvre. C'est la tour de contrôle de l'aéroport qui garantit la régulation du trafic aérien, en s'assurant qu'il n'y a pas de conflit de trajectoire ou de destination entre plusieurs avions.
En plus du diagramme de classes, représenter par une collaboration (diagramme de collaboration ou diagramme d'objets et de séquence) l'atterrissage d'un avion parmi deux voulant atterrir et un voulant décoller).

Exercice 2

Modéliser un afficheur graphique de données.
L'afficheur se base sur un ensemble de données pour dessiner les graphiques correspondants. Il affiche deux types de diagrammes, un camembert et un histogramme. Toute modification sur les données doit être automatiquement répercutée sur les diagrammes. Il est probable que le système évolue pour que les données soient en plus présentées dans un tableau.

Exercice 3

Modéliser la communication d'un ensemble d'applications avec l'environnement d'exécution.
Une application peut avoir besoin de connaître, à tout instant, des propriétés de l'environnement. Si l'une des applications veut modifier une de ces propriétés, il faut être sûr qu'aucune autre application ne peut accéder à l'environnement à ce moment là.

Exercice 4

Modéliser la structure du 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 graphique. Le contenu du moteur de jeu est de moindre importance.

Exercice 5

Modéliser une grammaire syntaxique.

Votre modèle doit permettre de représenter la grammaire suivante, les côtes ne faisant pas partie de la grammaire (Les littéraux peuvent être considérés comme des objets de type String) :

```
expression ::= littéral | alternative | séquence | répétition |
'('expression')'
alternative ::= expression '|' expression
séquence ::= expression '&' expression
répétition ::= expression '*'
littéral ::= 'a'|'b'|'c'|…{'a'|'b'|'c'|…}*
```

En plus du diagramme de classes, montrer par un diagramme d'objet que votre modèle peut instancier : `bonjour & (madame | monsieur)*`

Exercice 6

Modéliser un ensemble hifi-vidéo

L'ensemble hifi-vidéo est composé d'une télévision et d'une chaîne HIFI séparée. Les deux appareils sont totalement indépendants, mais ils ont été fabriqués par le même constructeur, et répondent à des protocoles communs. Un utilisateur peut intervenir à distance sur ces appareils grâce à une télécommande (une par appareil) fabriquée par le même constructeur. La télécommande de la télévision permet uniquement de changer de chaîne (n chaînes mémorisées à un instant t), et celle de la chaîne HIFI permet uniquement de changer de station (10 stations mémorisées).

Exercice 7

Modéliser un éditeur de classes UML

Une classe est un objet graphique qui a une position (coordonnées X et Y) sur un diagramme. Tous les objets graphiques du diagramme sont déplaçables à la souris. L'éditeur a une fonctionnalité Undo/Redo qui permet d'annuler ou de reproduire un déplacement.

Exercice 8

Modéliser le fonctionnement d'une vidéothèque

La vidéothèque met à disposition de ses clients des DVD selon trois catégories : *Enfant*, *Normal* et *Nouveauté*. Un DVD est dans la catégorie *Nouveauté* pendant quelques semaines, puis passe dans l'une des autres catégories. Le prix des DVD dépend de la catégorie. Il est probable que le système évolue pour que la catégorie *Horreur* soit ajoutée.

Exercice 9

Modéliser le fonctionnement d'une vidéothèque (suite)

Lorsque le gérant fait ses comptes à la fin du mois, il souhaite récupérer les informations des clients qui ont acheté un DVD pendant le mois. Au choix, il peut afficher à l'écran ces informations au format texte ou HTML. On veillera à limiter au maximum les duplications de code. Il est probable que le système évolue pour que l'affichage soit aussi possible en XML.

Il n'est pas utile de récupérer le modèle de l'exercice précédent.

Exercice 10

Modéliser une application

Deux équipes de développement travaillent en parallèle sur cette application. L'une travaille sur le fonctionnel, l'autre sur l'interface graphique. Nous ne nous intéressons ici qu'aux menus. Modéliser la gestion des menus de cette application, de manière à ce que le travail des deux équipes reste indépendant. Les concepteurs de la partie fonctionnelle doivent implémenter les fonctionnalités des menus de la manière la plus simple possible.

VI. Expérimentation de janvier 2008

L3 - IUP ISI et LP SIL Janvier 2008

Trouver des solutions à des problèmes de conception

Ce document propose un ensemble d'exercices de modélisation à objets. Vous devez produire un diagramme de classes UML **ainsi qu'un diagramme de séquence ou de collaboration** illustrant chaque exercice posé. Chaque diagramme doit contenir suffisamment d'informations pour démontrer que le problème est résolu (attributs, méthodes, relations, stéréotypes).

Le but de ces exercices est que vous suiviez un raisonnement qui vous est propre. Ces modélisations peuvent s'envisager de plusieurs manières différentes. Ne cherchez pas de solutions communes avec vos camarades, ni de solutions sur Internet ou dans des ouvrages de conception.

Certaines modélisations sont présentées avec des évolutions probables. Vos conceptions doivent être structurées de manière à ce que ces évolutions soient facilement intégrables. **Faites apparaître ces évolutions dans vos diagrammes**.

Les diagrammes de classes et de séquence doivent être rendus sur papier.

Exercice 1

Modéliser des classes comparables.

Soit les classes Nombre et Chaîne de caractères, ayant, entre autres, les fonctions de comparaisons suivantes : inferieur, inferieurOuEgal, superieur, superieurOuEgal, et egal, en sachant que d'autres classes peuvent avoir ces mêmes fonctions de comparaison. Vous devez veillez à ce que les protocoles soient uniformes et que le minimum de fonctionnalités soit implémenté dans les classes ayant des instances comparables.

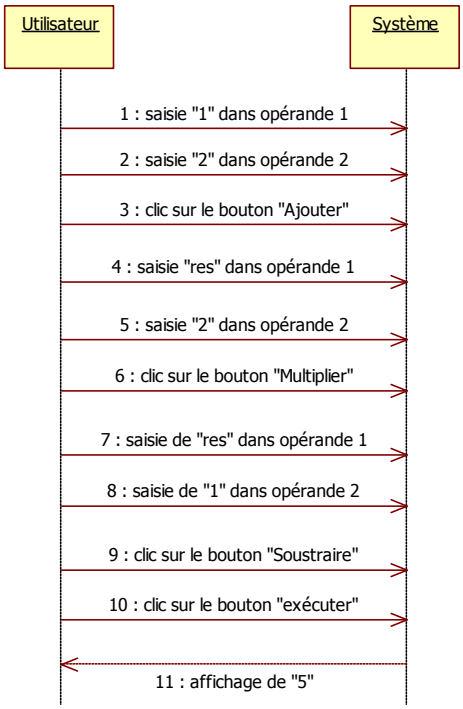
Exercice 2

Modéliser un gestionnaire d'aide d'une application Java.

Un gestionnaire d'aide permet d'afficher un message d'aide en fonction de l'objet sur lequel le client a cliqué. Par exemple, le « ? » situé quelquefois à côté du menu contextuel d'une fenêtre Windows permet d'afficher l'aide en fonction du bouton ou de la zone sur laquelle on clique. Si le bouton sur lequel on clique ne contient pas d'aide, c'est la zone contenante qui affiche son aide, et ainsi de suite. Si aucun objet ne contient d'aide, au final, le gestionnaire affiche « Pas d'aide disponible pour cette zone ». Instanciez votre diagramme de classes dans un diagramme de séquence sur l'exemple d'une fenêtre d'impression. Cette fenêtre (JDialog) est constituée d'un texte explicatif (JLabel), et d'un container (JPanel). Ce dernier contient un bouton Imprimer (JButton) et un bouton Annuler (JButton). Le bouton Imprimer contient l'aide « Lance l'impression du document ». Le bouton Annuler, le texte ainsi que la fenêtre ne contiennent pas d'aide. Enfin, le container contient l'aide « Cliquez sur l'un des boutons ». Dans le diagramme de séquence, faites apparaître les scénarii : « L'utilisateur demande l'aide du bouton Imprimer », « L'utilisateur demande l'aide du bouton Annuler », et « L'utilisateur demande l'aide du texte ».

Exercice 3

Modéliser un didacticiel pour apprendre à programmer une calculette.



Cette calculette exécute les quatre opérations arithmétiques de base. Le but de ce didacticiel est de permettre de saisir un ensemble d'opérations à effectuer séquentiellement. Le didacticiel présente un bouton par opération arithmétiques, et deux champs de saisie pour les opérandes. Après chaque clic sur un bouton d'une opération, l'utilisateur a alors le choix de recommencer ou d'exécuter la suite d'opérations pour obtenir le résultat. Le diagramme de séquence ci-contre présente les interactions entre l'utilisateur et le didacticiel.
Il est probable que ce didacticiel évolue afin de permettre à l'utilisateur de supprimer la dernière opération de la liste et de prendre en compte l'opération de modulo.

Vous pouvez utiliser le scénario présenté dans le diagramme de séquence système pour illustrer votre diagramme de séquence détaillé.

Exercice 4

Modéliser l'affichage sur l'écran d'une calculatrice

Cette calculatrice permet la saisie d'expressions, qui peuvent être « des nombres » ou « des opérations binaires » (addition, soustraction et multiplication). La calculatrice propose deux modes d'affichage : « infixés » (i.e 3_+_3) ou « préfixés » (i.e +_3_3). Après avoir saisi ses expressions, l'utilisateur peut choisir le mode d'affichage, et passer de l'un à l'autre autant de fois qu'il le désire. Il est probable que le système évolue pour que le mode d'affichage « suffixé » (i.e 3_3_+) soit ajouté.

Exercice 5

Modéliser le fonctionnement d'une boîte de dialogue.

Cette boîte de dialogue contient une liste de noms et un champ de saisie en lecture seule. Lorsque l'utilisateur clique sur un nom de la liste, il apparaît automatiquement dans le champ de saisie. Tant que l'utilisateur n'a pas cliqué sur un nom de la liste (donc tant que le champ de saisie est vide), le bouton de validation de la boîte de dialogue est désactivé. Attention, **ne modélisez pas la boîte de dialogue, mais uniquement son fonctionnement, en y intégrant son affichage**, et veillez à ce que l'interconnexion entre les différents objets de la boîte de dialogue soit minimale.

Exercice 6

Modéliser un système permettant d'afficher des fenêtres vides (sans bouton, sans menu…).

Une fenêtre peut avoir plusieurs styles différents, dépendants de la plate-forme d'utilisation. On considère qu'il existe deux plates-formes, 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 spécialiser ces fenêtres en fenêtres applicatives (capables de gérer des applications) et en fenêtres iconisées (avec une icône).

Exercice 7

Modéliser un logiciel de « veille » pour les traders en bourse.

Chaque fois qu'une transaction boursière a lieu, une vente est émise et est ajoutée à un pool de vente. Les traders doivent être mis immédiatement au courant. Ils ont à leur disposition, un jeu d'applets activables à tout instant permettant :

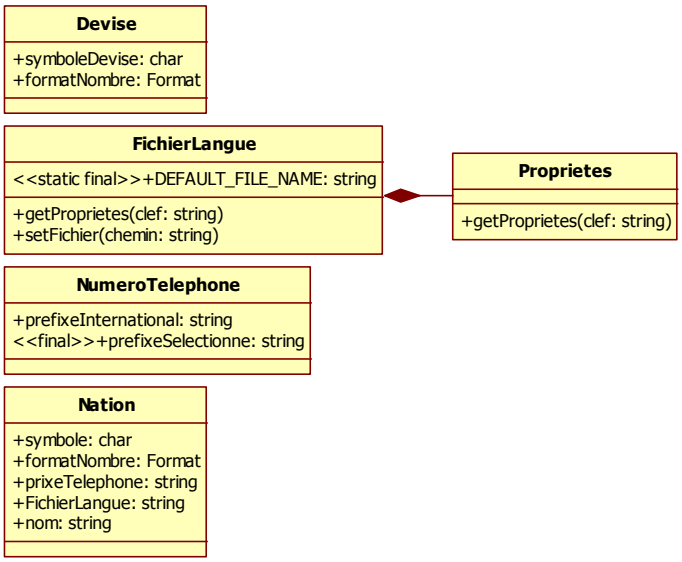
- soit de surveiller en temps réel le nombre de ventes effectuées pendant une journée boursière,
- soit de surveiller en temps réel le total des ventes en actions effectuées pendant une journée boursière,
- soit de surveiller en temps réel le total des nombres d'actions vendues pendant une journée boursière.

Ce logiciel devra évoluer afin de prendre en compte un pool d'achats (de la même façon que les ventes). D'autre part, une demande des traders est de pouvoir surveiller les ventes et les achats d'un titre particulier à la demande.

Exercice 8

Compléter le diagramme de classes.

Vous devez permettre à tout programme client du sous-système fourni ci-dessous de mettre à jour, en fonction d'une nation choisie, le symbole courant, le formatage des nombres, le nom du fichier des propriétés, ainsi que le préfixe international par défaut, pour tout numéro de téléphone. Dans le diagramme de séquence, mettez en évidence les interactions de la méthode setNation(String name) [appelée par un client] qui permet de récupérer ces paramètres.



Exercice 9

Modéliser l'utilisation d'un formulaire de saisie.

Ce formulaire est destiné à recueillir des informations sur des utilisateurs. Une liste de champs (de type String) est gérée par l'administrateur, de manière à en ajouter ou en supprimer en fonction de besoins statistiques. À chaque demande d'affichage, le formulaire est construit dynamiquement en fonction de cette liste. N'oubliez pas de mémoriser les informations des utilisateurs recueillies par le formulaire. Il n'est pas nécessaire de fournir un diagramme de séquence pour cet exercice.

2

Requête de détection de dernière génération

I. Requête de détection du C_SP5

```
package Foundation::Core
context Element
def : isRecc(a : Association) : Boolean =
|  a.connection->flatten.participant->asSet->size=1
def : hasRecc(Ba : Bag(Association)) : Boolean =
|  Ba->select(
|  |  a:Association|isRecc(a)
|  )->size>0
def : modele : Bag(Class) =
|  Class::allInstances->asBag
def : req_Component_1(ens_modele : Bag(Class)) : Bag(Class) =
|  ens_modele->select(
|  |  specialization->size>=2
|  )
def : step2_req_Component_1(elem_req_Composite_3 : Class,Bag_req_Component_1 : Bag(Class)) :
Bag(Class) =
|  Bag_req_Component_1->select(
|  |  specialization->flatten.child->includes(elem_req_Composite_3)
|  )
def : step4_req_Component_1(elem_req_Composite_3 : Class,Bag_req_Component_1 : Bag(Class)) :
Bag(Class) =
|  Bag_req_Component_1->excluding(elem_req_Composite_3)
def : step5_req_Component_1(elem_req_Composite_3 : Class,Bag_req_Component_1 : Bag(Class),
Bag_req_Leaf_2 : Bag(Class)) : Bag(Class) =
|  Bag_req_Component_1->select(
|  |  association->select(
|  |  |  association->select(
|  |  |  |  connection->select(
|  |  |  |  |  participant->forAll(
|  |  |  |  |  |  elem | Bag_req_Leaf_2->includes(elem) and
|  |  |  |  |  |  Bag_req_Component_1->excludes(elem)
|  |  |  |  |  )
|  |  |  |  )->size=0
|  |  |  )->size=0
|  |  )->size=0 and
|  |  association->select(
|  |  |  association->select(
|  |  |  |  connection->select(
|  |  |  |  |  participant->includes(elem_req_Composite_3)
|  |  |  |  )->size=0
|  |  |  )->size=0
|  |  )->size=0 and
|  |  specialization->select(
|  |  |  child->includes(elem_req_Composite_3)
|  |  )->size>=1 and
|  |  specialization->select(
|  |  |  child->forAll(
|  |  |  |  elem | Bag_req_Leaf_2->includes(elem) and
|  |  |  |  Bag_req_Component_1->excludes(elem)
|  |  |  )
|  |  )->size>=1
|  )
def : req_Leaf_2(ens_modele : Bag(Class)) : Bag(Class) =
|  ens_modele->select(
|  |  association->select(
|  |  |  aggregation=Foundation::DataTypes::AggregationKind::NONE and
|  |  |  isNavigable=true and
|  |  |  multiplicity->select(
|  |  |  |  range->select(
|  |  |  |  |  lower=0 and upper=-1
|  |  |  |  )->size>=1
|  |  |  )->size>=1
|  |  )->size>=1 and
|  |  generalization->size>=1
|  )
```

```
def : step2_req_Leaf_2(elem_req_Composite_3 : Class,Bag_req_Leaf_2 : Bag(Class)) :
Bag(Class) =
|  Bag_req_Leaf_2->select(
|  |  association->select(
|  |  |  aggregation=Foundation::DataTypes::AggregationKind::NONE and
|  |  |  isNavigable=true and
|  |  |  multiplicity->select(
|  |  |  |  range->select(
|  |  |  |  |  lower=0 and upper=-1
|  |  |  |  )->size>=1
|  |  |  )->size>=1
|  |  )->flatten.association->flatten.connection->flatten->select(
|  |  |  aggregation=Foundation::DataTypes::AggregationKind::COMPOSITE and
|  |  |  multiplicity->select(
|  |  |  |  range->select(
|  |  |  |  |  lower=1 and upper=1
|  |  |  |  )->size>=1
|  |  |  )->size>=1
|  |  )->flatten.participant->includes(elem_req_Composite_3)
|  )
def : step4_req_Leaf_2(elem_req_Composite_3 : Class,Bag_req_Leaf_2 : Bag(Class)) :
Bag(Class) =
|  Bag_req_Leaf_2->excluding(elem_req_Composite_3)
def : step5_req_Leaf_2(elem_req_Composite_3 : Class,Bag_req_Component_1 : Bag(Class),
Bag_req_Leaf_2 : Bag(Class)) : Bag(Class) =
|  Bag_req_Leaf_2->select(
|  |  association->select(
|  |  |  association->select(
|  |  |  |  connection->select(
|  |  |  |  |  participant->includes(elem_req_Composite_3)
|  |  |  |  )->size>=1 and
|  |  |  |  connection->select(
|  |  |  |  |  aggregation=Foundation::DataTypes::AggregationKind::COMPOSITE and
|  |  |  |  |  multiplicity->select(
|  |  |  |  |  |  range->select(
|  |  |  |  |  |  |  lower=1 and upper=1
|  |  |  |  |  |  )->size>=1
|  |  |  |  |  )->size>=1
|  |  |  |  )->size>=1 and
|  |  |  |  connection->select(
|  |  |  |  |  aggregation=Foundation::DataTypes::AggregationKind::NONE and
|  |  |  |  |  isNavigable=true and
|  |  |  |  |  multiplicity->select(
|  |  |  |  |  |  range->select(
|  |  |  |  |  |  |  lower=0 and upper=-1
|  |  |  |  |  |  )->size>=1
|  |  |  |  |  )->size>=1
|  |  |  |  )->size>=1
|  |  |  )->size>=1
|  |  )->size>=1 and
|  |  association->select(
|  |  |  aggregation=Foundation::DataTypes::AggregationKind::NONE and isNavigable=true and
|  |  |  multiplicity->select(
|  |  |  |  range->select(
|  |  |  |  |  lower=0 and upper=-1
|  |  |  |  )->size>=1
|  |  |  )->size>=1
|  |  )->size>=1 and
|  |  association->select(
|  |  |  association->select(
|  |  |  |  connection->select(
|  |  |  |  |  participant->forAll(
|  |  |  |  |  |  elem | Bag_req_Component_1->includes(elem) and
|  |  |  |  |  |  Bag_req_Leaf_2->excludes(elem)
|  |  |  |  |  )
|  |  |  |  )->size=0
|  |  |  )->size=0
|  |  )->size=0 and
|  |  generalization->select(
|  |  |  parent->forAll(
|  |  |  |  elem | Bag_req_Component_1->includes(elem) and
|  |  |  |  Bag_req_Leaf_2->excludes(elem)
|  |  |  )
|  |  )->size>=1 and
|  |  generalization->select(
|  |  |  parent->includes(elem_req_Composite_3)
|  |  )->size=0 and
|  |  specialization->select(
|  |  |  child->includes(elem_req_Composite_3))->size=0)
```

```
|  |  )->size=0
|  )
def : req_Composite_3(ens_modele : Bag(Class)) : Bag(Class) =
|  ens_modele->select(
|  |  association->select(
|  |  |  aggregation=Foundation::DataTypes::AggregationKind::COMPOSITE and
|  |  |  multiplicity->select(
|  |  |  |  range->select(
|  |  |  |  |  lower=1 and upper=1
|  |  |  |  )->size>=1
|  |  |  )->size>=1
|  |  )->size>=2 and
|  |  association->select(
|  |  |  aggregation=Foundation::DataTypes::AggregationKind::NONE and
|  |  |  isNavigable=true and
|  |  |  multiplicity->select(
|  |  |  |  range->select(
|  |  |  |  |  lower=0 and upper=-1
|  |  |  |  )->size>=1
|  |  |  )->size>=1
|  |  )->size>=1 and
|  |  generalization->size>=1 and
|  |  hasRecc(association->flatten.association)
|  )



def : execution1(Bag_req_Composite_3 : Bag(Class),Bag_req_Component_1 : Bag(Class),
Bag_req_Leaf_2 : Bag(Class)) : Bag(Bag(Bag(Class))) =
|  Bag_req_Composite_3->iterate(
|  |  elem_req_Composite_3 : Class;res:Bag(Bag(Bag(Class))) =
|  |  Bag{} | res->including(
|  |  |  execution2(
|  |  |  |  elem_req_Composite_3,step4_req_Component_1(
|  |  |  |  |  elem_req_Composite_3,step2_req_Component_1(
|  |  |  |  |  |  elem_req_Composite_3,Bag_req_Component_1
|  |  |  |  |  )
|  |  |  |  ),step4_req_Leaf_2(
|  |  |  |  |  elem_req_Composite_3,step2_req_Leaf_2(
|  |  |  |  |  |  elem_req_Composite_3,Bag_req_Leaf_2
|  |  |  |  |  )
|  |  |  |  )
|  |  |  )
|  |  )
|  )
def : execution2(elem_req_Composite_3 : Class,Bag_req_Component_1 : Bag(Class),
Bag_req_Leaf_2 : Bag(Class)) : Bag(Bag(Class)) =
|  Bag{
|  |  Bag{elem_req_Composite_3},
|  |  step5_req_Component_1(
|  |  |  elem_req_Composite_3,Bag_req_Component_1,Bag_req_Leaf_2
|  |  ),
|  |  step5_req_Leaf_2(
|  |  |  elem_req_Composite_3,Bag_req_Component_1,Bag_req_Leaf_2
|  |  )
|  }
def : execution(ens_modele : Bag(Class)) : Bag(Bag(Bag(Class))) =
|  execution1(
|  |  req_Composite_3(ens_modele),
|  |  req_Component_1(ens_modele),
|  |  req_Leaf_2(ens_modele)
|  )
query : execution(modele)
endpackage
```