Un Modèle de transformation des patrons de conception de l'Orienté Objet vers l'Orienté Aspect

Mohamed Lamine Berkane[1], Mahmoud Boufaida[2]
Laboratoire LIRE, Université Mentouri de Constantine.
[1] bmlamine3@yahoo.fr, [2]mboufaida@umc.edu.dz

Résumé. Depuis l'émergence du concept des patrons de conception, plusieurs chercheurs se sont intéressés à offrir de l'aide aux concepteurs pour faciliter la mise en œuvre des patrons de conception. Cette dernière est très utilisée dans l'approche par Objets. Cependant l'approche par Objets pose plusieurs problèmes et limites, principalement liés à la dispersion et à l'enchevêtrement du code de leurs instances dans l'implémentation des applications. Par ailleurs, l'approche basée sur des Aspects permet de nouvelles solutions pour ces patrons contribuant à garder la visibilité et l'isolement de l'instance de chaque patron dans le code des applications. Ces deux caractéristiques permettent de pallier les problèmes d'utilisation et d'améliorer la réutilisation. Nous nous intéressons dans cet article à la mise en œuvre des patrons de conception basée sur l'approche par Aspects. Pour cela, nous présentons un modèle de transformation, où nous considérons une instance du patron dans le modèle Orienté-Objet comme une instance du problème qui sera résolue par une instance de solution dans le modèle Orienté-Aspect, et une transformation qui traduit des instances du problème en solution indiquer par le patron. Afin de valider notre modèle de transformation, nous exprimons le modèle de transformation par le framework EMF.

Mots-clés: Patrons de conception, paradigme Orienté-Aspect, modèle de transformation, EMF.

1 Introduction

Les patrons de conception constituent un moyen efficace pour la conception, la composition de plusieurs types de composants réutilisables, et pour le développement de grands systèmes complexes. Ils permettent d'améliorer la qualité et la compréhension des programmes, facilitent leur évolution et augmentent notamment leur réutilisation [12].

Les approches de la mise en œuvre des patrons de conception dans le cadre Objet sont distinguées en trois catégories. Dans la première catégorie appelée approche descendante, le patron de conception est vu comme un ensemble de classes paramétrées, reliées entre elles, qu'il s'agit d'instancier pour un domaine d'application particulier [9]. La deuxième catégorie appelée approche ascendante consiste à refactoriser un modèle existant pour en améliorer la qualité à l'aide d'un patron de conception. Dans ce cas, le point de départ est un modèle du niveau conception de l'application à modéliser, qui comporterait un fragment qui poserait

problème au niveau de la conception. Il s'agit alors de transformer ce fragment selon les prescriptions recommandées par le patron de conception approprié [9]. La dernière catégorie illustre les patrons par la spécification de la structure du problème résolu par le patron. Cette spécification est basée sur une représentation explicite et du problème résolu par le patron et la solution proposée par ce dernier. Cette dernière approche est vue aussi comme un cas particulier de l'approche ascendante.

Par ailleurs, la programmation orientée aspect a émergé avec comme objectif principal l'amélioration de la modularité des applications afin de faciliter leur évolution et leur réutilisation [15]. L'utilisation des patrons de conception dans une approche strictement Objet pose cependant plusieurs problèmes et limites qui sont principalement liés à la dispersion et à l'enchevêtrement du code de leurs instances dans l'implémentation des applications. Pour cela, l'approche basée sur les Aspects permet de nouvelles solutions pour ces patrons contribuant à garder la visibilité et l'isolement de l'imitation de chaque patron dans le code des applications, afin de pallier à leurs problèmes d'utilisation et d'améliorer leur traçabilité et leur réutilisation [13, 14].

Dans ce travail, nous proposons un modèle de transformation pour la mise en œuvre des patrons de conception. Cette approche est inscrite dans l'approche ascendante où, le modèle existant comporterait un fragment qui poserait problème sera considéré par une instance d'un patron dans le modèle Objet, et la solution proposée par ce patron sera résolu par une instance de solution dans le modèle Aspect.

La représentation des modèles du problème, et de ses solution sont de la forme de méta-modèles UML, permet d'effectuer l'opération de la mise en œuvre du patron comme une transformation du méta-modèle du problème (méta-modèle Objet) en le méta-modèle de la solution (méta-modèle Aspect).

Dans la prochaine section, nous décrivons les motivations de notre approche. La section 3 présente notre modèle de transformation. L'implantation de ce modèle est décrite dans la section 4. Nous présentons les travaux reliés dans la section 5, avant de conclure dans la section 6.

2. Motivations

Depuis l'émergence du concept des patrons de conception, plusieurs chercheurs se sont intéressés à offrir de l'aide aux concepteurs pour faciliter l'application de patrons de conception, dont [1, 6, 7, 10, 16, 19].

Par exemple, l'approche de Mili et al., [16] représente le problème par trois taches : La certification de la pertinence d'un patron à une situation donnée sans avoir un modèle du problème, puis la compréhension du patron doit illustrer l'« avant » et l' « après » application du patron, et enfin l'application du patron à un fragment de modèle d'analyse n'est rien d'autre que l'application de transformations à ce fragment. Ces transformations peuvent être encodées, d'une façon générique, comme des transformations du modèle du problème en un modèle de solution.

Cette approche décrit la représentation et la mise en œuvre des patrons qui prennent en compte la représentation du problème pour les trois tâches.

Notre approche est inspirée du travail de Mili et al., [16], sauf qu'un fragment qui poserait problème sera considéré par une instance d'un patron dans le modèle Objet, et la solution proposée par ce patron sera résolue par une instance de solution dans le modèle Aspect.

3. Une approche pour la transformation des patrons de conception

Dans cette section nous présentons la représentation du modèle Object, ainsi celle du modèle Aspect. Nous concluons cette section par la représentation du modèle de transformation.

3.1. Représentation des modèles et classification des patrons de conception

Avant d'entamer le principe de transformation, nous allons voir les différentes représentations des modèles. Le patron de conception est vu comme un triplet (MO, MA, T) où :

- MO (Modèle Objet) : est une caractérisation du problème résolu par le patron sous la forme d'un méta-modèle dont les instances sont représentées par le modèle Objet.
- MA (Modèle Aspect) : est la structure proposée par le patron pour résoudre les problèmes de l'Objet. Cette représentation est aussi représentée sous la forme d'un méta-modèle.
- Un modèle de transformation exprimant la mise en œuvre du patron consiste à transformer des instances de l'orientées objet vers des instances orientées aspect.

Nous avons classifié les patrons de conception en deux classes, une classe pour les patrons pouvant être transformés, et la deuxième classe pour les patrons exclus de la transformation [3, 4]. Les raisons d'écarter ces patrons sont présentées comme suit :

- Le patron 'Façade', ne sera pas considéré par la transformation, parce qu'il a une implémentation orientée aspect strictement identique à l'implémentation orientée objet [13, 14, 17].
- Le patron 'Singleton', ne sera pas également considéré par la transformation, (malgré que ce patron montre des avantages pour sa solution aspect) parce que l'implémentation aspect de ce patron, peut avoir des effets de bord nuisibles [17].
- Les patrons 'Template Methode', 'Bridge' 'Abstract Factory', 'Factory Method' et 'Builder', ont peu de bénéfice pour la solution aspect [14].
- Les patrons 'State' et 'Interpreter' parce qu'ils définissent des rôles de définition [14]. Ces rôles constituent la préoccupation principale de la classe. Cette dernière change d'une application à une autre.

Le tableau 1 résume la nouvelle classification :

Classes	**Patrons**
Patrons exclus de la transformation	Facade, Singleton, Template Methode, Bridge, Abstract Factory, Factory Methode, Builder, Stat, Interpreter.
Patrons pouvant être transformés	Adapter, Composite, Iterator, Prototype, Proxy, Visitor, Chain of Responsibility, Command, Decorator, Flyweight, Mediator, Memento, Observer, Strategy.

Tableau 1. Nouvelle classification des patrons de conception.

Notre étude se base sur le patron Observer, parce qu'il a beaucoup d'avantages pour la solution basée sur l'aspect à savoir : meilleurs localisation, réutilisation, composition, et adaptabilité. [13, 14].

3.2 Modèle Objet (Méta-Modèle Objet)

Les instances Objets sont des modèles de niveau conception. Pour décrire les classes du modèle Objet, nous allons définir un méta-modèle Objet, i.e. un modèle dont les instances seront des modèles tels que l'exemple de la Figure 1. Nous représentons le méta-modèle Objet dans la figure 2.

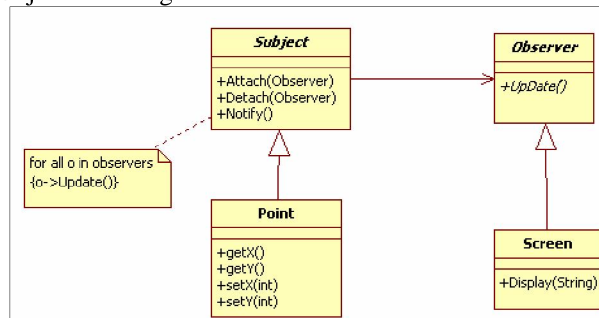

Fig. 1. Exemple Objet pour le patron Observer.

Les classes « Subject » et « Observer » sont des méta-classes dans le sens que leurs instances sont des classes. Les associations « inherits_from » représentent les relations d'héritage qui doivent relier les instances des classes correspondantes, par exemple le « ConcreteSubject » (point) « inherits_from » le « Subject » Subject, de même, le « ConcreteObserver » (screen) « inherits_from » « Observer » Obsever.

Nous utilisons l'association « has method » pour présenter les méthodes d'une classe. Dans notre méta-modèle Objet, la méta-classe Subject contient trois méthodes : Attach, Detach, et Notify.

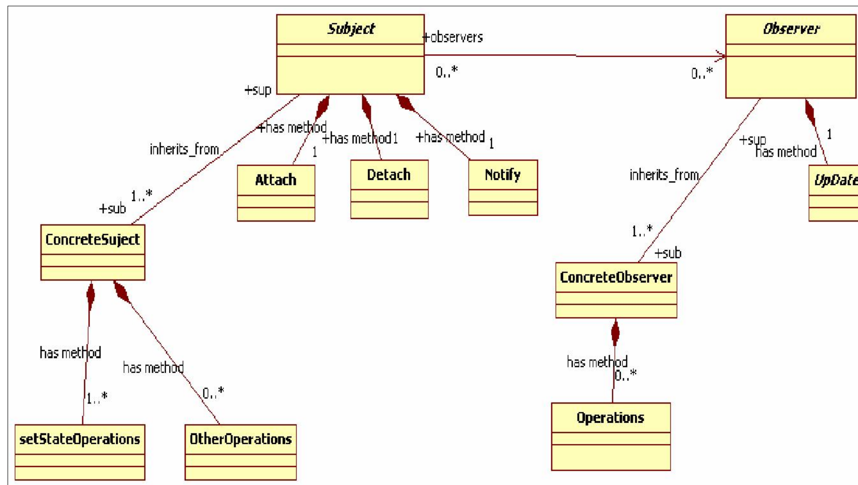

Fig. 2. Méta-Modèle Objet du patron Observer.

La méta-classe « ConcreteSubject » contient deux types de méthodes :

- Des méthodes déclenchant la mise à jour sur les classes Observers, telle que : setX et setY dans la figure 1. Ces méthodes sont définies dans le méta-modèle par la meta-classe (setStateOperations) ;
- Des simples méthodes liées uniquement au contexte du Subject telle que : getX et getY dans la figure 1. Ces méthodes sont définies dans le méta-modèle par la méta-classe (OtherOperations).

La méta-classe Observer contient uniquement la méthode abstraite UpDate.

Enfin, la méta-classe ConcreteObserver contient des simples méthodes.

En principe, il faudra représenter les paramètres et les types de retour des opérations pour pouvoir les transformer pour obtenir la solution Aspect. Nous nous passerons de ces détails dans le méta-modèle.

3.3 Modèle Aspect (Méta-Modèle Aspect)

Nous avons utilisé le même principe pour la représentation du méta-modèle Aspect. Pour définir les concepts de l'Aspect tel qu'Aspect, Pointcut, Advice et Declare Parents, nous utilisons les stéréotypes dans les méta-classes.

Les instances Aspect sont des modèles de niveau conception. Pour décrire les classes et les aspects du modèle Aspect, nous allons définir un méta-modèle Aspect, i.e. un modèle dont les instances seront des modèles tels que l'exemple de la figure 3. Nous représentons le méta-modèle Aspect dans la figure 4.

Les aspects « ObserverProtocol » et « SubjectObserver » sont des méta-aspects dans le sens que leurs instances sont des aspects. Les associations « inherits_from » représentent les relations d'héritage qui doivent relier les instances des aspects correspondantes, par exemple le « ObserverProtocol » (ObserverProtocol) « inherits_from » le « SubjectObserver » (CoordinateObserver).

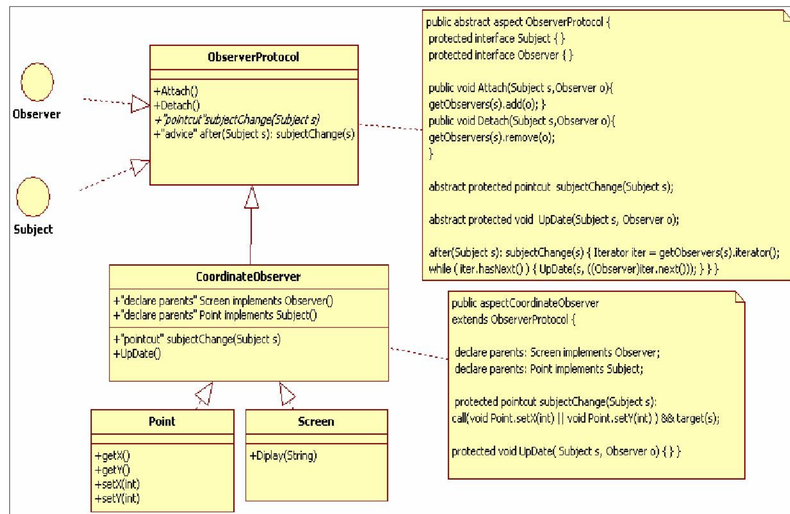

Fig. 3. Exemple Aspect pour le patron Observer.

L'association « inherits_from » décrit le lien d'héritage entre les classes, ainsi entre les aspects. Le méta-modèle Objet (figure 2) décrit le lien d'héritage entre les classes, mais dans le méta-modèle Aspect (figure 4) décrit ce lien entre les aspects. Nous utilisons aussi les associations « has method », « has pointcut », « has advice », et « has declare parents » pour présenter respectivement les méthodes d'une classe et/ou un aspect, le pointcut, le code Advice, et la déclaration inter-type d'un aspect.

La relation entre un pointcut et un code Advice est définie par l'association « linked to ». La déclaration inter-type est une relation nécessitant une interface, ainsi qu'une classe implémentant ce dernier. Pour cela, deux associations sont définies : « interface » décrit l'interface, et « impl » décrit la classe implémentant l'interface.

Le méta-aspect « ObserverProtocol » est un aspect abstrait qui contient l'essence du patron Observer. Il implémente les fonctions « Attach » et « Detach » du patron Observer, ainsi il définit un Advice qui sera exécuté après le déclenchement d'une opération du ConcretSubject (opération déclenchant la mise à jour). Le déclenchement de ces opérations est perçu par un pointcut, et la mise à jour est faite par la méthode UpDate. Ces derniers sont définis dans l'aspect abstrait « ObserverProtocol » d'une manière abstraite. Afin de concrétiser le pointcut définit par l'aspect abstrait, ainsi la méthode UpDate, il sera nécessaire de définir un méta-aspect concret « SubjectObserver ». Ce dernier définit aussi les méta-classes jouent le rôle Subject, ainsi les méta-classes jouent le rôle Observer par la déclaration inter-type. Les interfaces (méta-classes) « Subject » et « Observer » sont des interfaces vides. Les méta-classes ConcreteObserver et ConcreteSubject sont des classes qui ont le même principe du méta-modèle Objet. La représentation des aspects dans notre travail est faite selon la représentation de Suzuki [20]. Dans cette représentation, une relation de réalisation connectant une classe (ou une interface) à un aspect signifie que cet aspect entrecroisé (crosscut) cette classe (ou cette interface).

Fig. 4. Méta-Modèle Aspect du patron Observer.

3.4 Modèle de transformation

L'application d'un patron consiste à transformer une instance de l'Objet en une instance en Aspect. Nous pouvons donc représenter la transformation par une correspondance entre les éléments du méta-modèle Objet et ceux du méta-modèle Aspect. Ainsi, partant d'un modèle à l'entrée auquel nous voulons appliquer un patron, nous commençons d'abord par identifier les éléments du modèle qui jouent les rôles décrits par le méta-modèle Objet. Pour le cas du patron « Observer », nous cherchons, par exemple, à identifier les classes qui jouent les rôles de Subject et Observer dans le modèle à l'entrée. Ceci nous amène à avoir un modèle de l'entrer identique au modèle Objet. Ce dernier est transformé en une instance du méta-modèle Aspect grâce à la représentation de la transformation. Nous considérons dans cet article que le modèle de l'entrée correspond au modèle Objet.

La figure 5 montre le modèle de représentation des correspondances entre le méta-modèle Objet et le méta-modèle Aspect. Toutes les correspondances spécifient un élément (ou plusieurs) source, un élément destination (au plus), et une description de la transformation (ElementMapping). Une correspondance qui n'a pas d'élément source indique un élément fourni par la solution. Une correspondance qui n'indique pas un élément destination indique un élément qui disparaît dans la transformation. La correspondance entre méta-modèle Objet et méta-modèle Aspect est représentée par une instance de ModelMapping. La classe Element représente les éléments du modèle à l'entrée (une classe, une méthode, une association,etc.). Elle correspond au type ModelElement du MOF. La notion d'AspectClassMapping indique la correspondance entre une classe et un aspect.

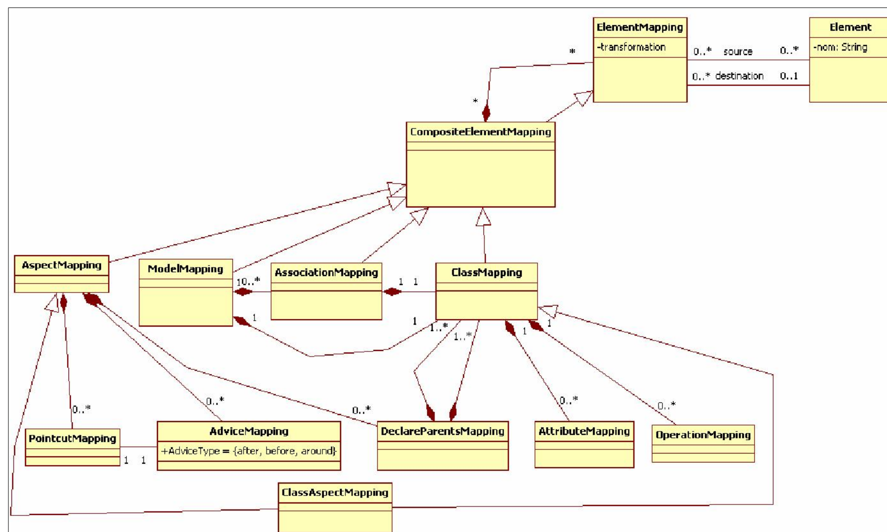

Fig. 5. Modèle des correspondances entre le modèle Objet et le modèle Aspect.

La transformation inhérente à un patron est spécifiée de façon déclarative par la correspondance entre les éléments du modèle Objet et ceux du modèle Aspect associés au patron considéré. Nous avons représenté cette transformation sous forme d'un modèle de transformation. La figure 6 montre un extrait du modèle de transformations du patron «Observer».

Par exemple, dans cet extrait, l'aspect-classe Subject&Observer2ObserverProtocol instance d'AspectClassMapping qui est une entité du méta-modèle des transformations associe les classes (interfaces) Observer et Subject du modèle Objet (Figure 2) à l'aspect ObserverProtocol du modèle Aspect (Figure 3). Elle spécifie aussi comment certaines propriétés de l'aspect ObserverProtocol sont déduites des classes (interfaces) Observer et Subject. Les détails de cette transformation est spécifie textuellement dans [14].

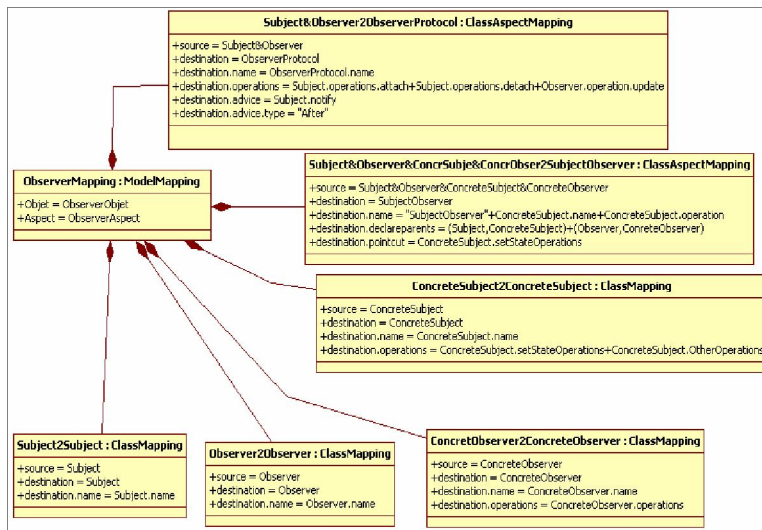

Fig. 6. Modèle de transformation du patron Observer.

4. Implémentation

Nous avons implémenté notre modèle de transformation au sein de l'environnement Eclipse, et plus particulièrement, en utilisant le cadre d'application Eclipse Modeling Framework (EMF) [21]. Le choix du EMF est influencé par son ouverture au domaine public et son support de la visualisation graphique du modèle de transformation.

4.1. Framework EMF

Le framework EMF [21] est un cadre d'application développé en Java destiné à l'environnement de développement Eclipse. De nos jours, un développeur va typiquement manipuler des fichiers Java (Annotated Java), des modèles UML (diagrammes de classes) exportés de divers outils de modélisation UML, et des fichiers XML. Le framework EMF a été développé dans le but d'offrir un cadre unifié pour la manipulation de ces trois types de fichiers. Dans notre cas, nous avons choisi Java (Annotated Java) pour décrire le modèle de transformation. Ce choix est justifié par la simplicité d'utilisation de ce dernier.

4.2 Implémentation de la transformation

Nous avons implémenté le modèle de transformation représenté par la figure 6 aussi comme un méta-modèle, avec sa propre extension de la classe EPackage.

La réalisation de cette transformation est effectuée par la création de 12 classes et interfaces en Annotated Java, décrites dans le tableau suivant (Tableau 2).

Classes/interfaces	**Objectifs**
interface ObserverTrans	contient tous les éléments utilisés pour la transformation : classes, interfaces, opérations, pointcut,etc.
interface SubjectOO	décrit l'interface Subject en O.O
classe SubjectOperationsPattOO	décrit les opérations du patron Observer pour l'interface Subject : Attach, Detach, et Notify.
interface ObserverOO	décrit l'interface Observer en O.O.
classe ObserverOperationsPattOO	décrit les opérations du patron Observer pour l'interface Observer : UpDate.
interface ConcreteSubjectOO	décrit la classe ConcreteSubject en O.O
interface setStateCSOO	décrit les méthodes de la classe ConcreteSubject déclenchant la mise à jour sur les classes ConcreteObservers.
interface OtherOperationsCSOO	décrit uniquement des simples méthodes liées uniquement au contexte du Subject.
interface ObserverProtOA	décrit le protocole du patron Observer (aspect abstrait).
interface SubjectChangeOA	décrit l'aspect concret.
interface SubjectChangePCOA	décrit le pointcut de l'aspect concret SubjectChange. Il est lié uniquement aux opérations setStateCSOO.
interface SubjectChangeDPsubjOA	décrit la déclaration inter-type entre les classes ConcreteSubject et l'interface Subject.
interface SubjectChangeDPobsOA	décrit la déclaration inter-type entre les classes ConcreteObserver et l'interface Observer.

Tableau 2. Classes et Interfaces du modéle de transformation.

Après la génération du **Model Code**, **Edit Code**, et enfin **Editor Code**, nous obtenons notre modèle de transformation via un éditeur graphique basée sur le package EMF.EDIT.

Rappelons que notre modèle de transformation effectue une correspondance entre les éléments Objets et les éléments Aspect.

Afin de valider notre modèle de transformation, nous allons utiliser l'exemple de la figure 1 comme un cas d'étude.

La figure 7 spécifie un pointcut (Subject Change PCOA) de l'aspect SubjectChangOA, par exemple les méthodes « setX » et « setY » sont considérées comme des points de jonction de ce pointcut. Ces méthodes représentent les méthodes déclenchant la mise à jour sur les classes Observers dans le modèle Objet.

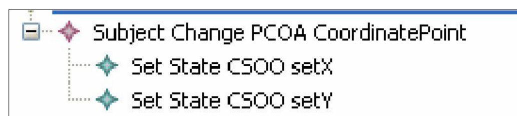

Fig. 7. Pointcut de l'aspect concret « SubjectChangeOA» dans le framework EMF.

5. Travaux reliés

Plusieurs travaux se sont intéressés à la représentation et la mise en œuvre des patrons de conception. Nous pouvons distinguer trois catégories d'approches. La première catégorie appelée approche descendante, l'objectif de cette catégorie est d'instancier les patrons pour un domaine d'application particulier. Cette instanciation vient de compléter un modèle de conception partiel construit manuellement. Elle représente les patrons soit par méta-modèle [1, 8] soit par des modèles [10, 11, 18]. Un cas particulier de cette catégorie est la génération de code par instanciation [3, 6].

La deuxième catégorie appelée approche ascendante, l'objectif de cette catégorie est de réaliser la réingénierie de modèles existants pour les rendre conformes à un patron [2, 19]. Cette catégorie se concentre sur l'aspect mise en œuvre des patrons. Elles fournissent une représentation explicite de la transformation inhérente à ce dernier.

Enfin, la dernière catégorie illustre les patrons par la spécification de la structure du problème résolu par le patron. Cette spécification est basée sur une représentation explicite et du problème résolu par le patron et de la solution proposée par ce dernier. Cette représentation, sous la forme de méta-modèles UML, permet d'opérationnaliser la mise en œuvre du patron comme une transformation du méta-modèle du problème en le méta-modèle de la solution. Ainsi, pour appliquer un patron de conception à un modèle UML sous considération, il faut chercher d'abord une occurrence du méta-modèle du problème dans le modèle UML [5, 9, 16].

Notre approche est inspirée de cette dernière catégorie où, le modèle existant qui comporterait un fragment qui poserait problème sera considéré par une instance d'un patron dans le modèle Objet, et la solution proposée par ce patron sera résolu par une instance de solution dans le modèle Aspect.

6. Conclusion

Nous avons proposé dans cet article un modèle de transformation pour la représentation et la mise en œuvre de patrons de conception dans l'approche par Aspect. Cette approche vise à fournir aux développeurs un reposoir de modèles réutilisables spécifiant les patrons dans le modèle Objet, ainsi dans le modèle Aspect. L'élément fondamental de notre modèle de transformation est la représentation explicite du modèle Aspect, une représentation qui permet de mieux comprendre les patrons et d'automatiser leur application. En effet, cela nous a permis de spécifier un patron comme un artefact réutilisable caractérisé par un modèle Objet (contexte d'utilisation), un modèle Aspect et un modèle de transformation décrivant la transformation inhérente à son application.

References

1. Albin-Amiot, H., Guéhéneuc, Y.G.: Meta-modeling Design Patterns: application to pattern detection and code synthesis", In proc. of ECOOP Work. on Aut. OO Sof. Dev. Met., (2001).
2. Alencar, P.S.C., Cowan, D.D., Dong, J., Lucena, C.J.P.:A transformational Process-Based Formal Approach to Object-Oriented Design. In : Formal Methods Europe FME'97, (1997).
3. Berkane, M.L., Boufaida, M. : Une Approche pour la génération du code aspect basée sur les patterns. In Journées Scientifiques sur l'Informatique et ses Applications, Guelma (2009).
4. Berkane, M.L., Boufaida, M.: A process to reverse engineering based on aspect-oriented implementation of design patterns. In: 9th International Arab Conference on Information Technology. ACIT'2008, Tunisia. (2008).
5. Berkane, M.L., et Boufaida, M.: Un processus de transformation pour la mise en oeuvre des patterns basée sur l'approche par aspects. In Séminaire National en Informatique Biskra SNIB'08, p253-259, Biskra. (2008).
6. Budinsky, F.J. Finnie, M.A. Vlissides, J.M. Yu, P.S.: Automatic Code Generation from Design Patterns. In IBM Systems Journal, vol. 35, n° 2, pp. 151-171, (1996).
7. Eden, A.H., Gil, J., Hirshfeld Y., Yehudai A.: Towards a mathematical foundation for design patterns. In Technical report, department of information technology, Uppsala University, (1999).
8. Elaasar, M., Briand, L., Labiche, Y. : A Metamodeling Approach to Pattern Specification and Detection. In Proc. of ACM/IEEE Inter. Conf. on MDE (MoDELS 2006), Italy, (2006).
9. El Boussaidi, G., Mili, H. : Une approche à base de règles pour la mise en oeuvre des patrons de conception. In International Symposium on Programming and Systems, Algiers 2007.
10. Florijn, G., Meijers, M., van-Winsen, P. : Tool support for object-oriented patterns. In Lecture Notes in Computer Science, vol. 1241, pp. 472-495, (1997).
11. Fontoura, M., Lucena, C. : Extending UML to Improve the Representation of Design Patterns. In Journal of OO Programming, vol. 13, n° 11, (2001).
12. Gamma, E., Helm, R., Johnson, R., Vlissides J.: Design Patterns, Elements of reusable Object-Oriented Software. In Addison-Wesley Publishing Company, (1995).
13. Hachani, O. : Patrons de conception a base d'aspects pour l'ingénierie des systèmes d'information par réutilisation. Thèse de doctorat en Juillet 2006. (2006).
14 Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In Proceedings of OOPSLA 2002, ACM SIGPLAN Notices, Vol. 37, n° 11, p. 161-173, (2002).
15. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J. : Aspect-Oriented Programming. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS vol.1241, Springer-Verlag, Juin 1997. (1997).
16. Mili, H., El Boussaidi, G., Salah, A. : Représentation et mise en oeuvre des patrons de conception par représentation explicite des problèmes. In Langages et modèles à objets (LMO), Suisse, Mars 2005. (2005).
17. Pawlak, R., Retaillé, R., Seinturier, L. : Programmation orienté aspect pour Java/J2EE. (2004).
18. Sanada, Y., Adams, R. : Representing Design Patterns and Frameworks in UML, Towards a Comprehensive Approach. In ", Journal of Object Technology, vol. 1, n° 2, pp.143-154, 2002. (2002).
19. Sunyé, G., Le Guennec, A., Jézéquel, J.M. : Design pattern application in UML. In Proc. of the 14th Object Oriented Programming European Conference, pp. 44-62, 2000. (2002).
20. Suzuki, J., Yamamoto, Y.: Extending UML with Aspects: Aspect Support in the Design Phase. In ECOOP'99 Workshop on Aspect-Oriented Programming, (1999).
21. Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/