Processus IDM pour l'intégration des patrons de sécurité dans une application à base de composants

Rahma Bouaziz

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par

Université Toulouse 2 Le Mirail (UT2 Le Mirail)

Cotutelle internationale avec :

Présentée et soutenue par

Rahma Bouaziz

Le : 6 décembre 2013

Titre :

Processus IDM pour l'intégration des patrons de sécurité dans une application à base de composants

---

École doctorale et discipline ou spécialité :

Arts, Lettres, Langues, Philosophie, Communication (ALLPH@)

Unité de recherche :

MITT Institut de Recherche en Informatique de Toulouse

Directeur(s) de Thèse :

Bernard Coulette

Rapporteurs :

Ahmed HADJ KACEM

Mourad Chabane OUSSALAH

Autre(s) membre(s) du jury :

THÈSE

Manuscrit du 10 septembre 2013

U.F.R. MATHÉMATIQUES, INFORMATIQUE ET GESTION

Sera soutenue en vue de l'obtention du titre de

DOCTEUR DE L'UNIVERSITÉ DE TOULOUSE

Mention INFORMATIQUE

par

RAHMA BOUAZIZ

École doctorale : Mathématiques, Informatique et Télécommunications de Toulouse
Laboratoire d'accueil : Institut de Recherche en Informatique de Toulouse
Équipe d'accueil : Modèles, Aspects, Composants pour des Architectures à Objets

Processus IDM pour l'intégration des patrons de sécurité dans une application à base de composants

Ahmed HADJ KACEM	*Professeur, Université de Sfax, HdR*	(rapporteur)
Mourad Chabane OUSSALAH	*Professeur, Faculté des Sciences de Nantes*	(rapporteur)
Bernard COULETTE	*Professeur, Université de Toulouse II, IRIT*	(directeur de thèse)

Rahma Bouaziz

PROCESSUS IDM POUR L'INTÉGRATION DES PATRONS DE SÉCURITÉ DANS UNE APPLICATION À BASE DE COMPOSANTS

Directeur de thèse : Bernard Coulette, Professeur UTM, IRIT

Résumé

La sécurité est devenue un enjeu important dans le développement des systèmes logiciels actuels. La majorité des concepteurs de ces systèmes manquent d'expertise dans le domaine de la sécurité. Il s'avère donc important de les guider tout au long des différentes phases de développement logiciel dans le but de produire des systèmes plus sécurisés. Cela permettra de réduire le temps ainsi que les coûts de développement. Pour atteindre cet objectif, nous proposons d'appliquer l'expertise en matière de sécurité sous forme de patrons de sécurité lors de la phase de conception de logiciels. Un patron de sécurité intègre des solutions éprouvées et génériques proposées par des experts en sécurité. Cependant, les patrons de sécurité sont souvent négligés au niveau de la conception et ne constituent pas une solution intuitive qui peut être utilisée par les concepteurs de logiciels. Cela peut être le résultat de l'inadaptation de ces patrons au contexte des systèmes, la non-expertise des concepteurs dans le domaine de la sécurité ou encore l'absence d'un processus d'intégration de ces patrons dans les modèles à un haut niveau d'abstraction.

Afin de permettre aux concepteurs d'utiliser les solutions proposées par des patrons de sécurité, cette thèse propose une approche d'ingénierie dirigée par les modèles pour sécuriser des applications via l'intégration de patrons de sécurité. Nous avons choisi comme contexte d'application de notre approche, les applications à base de composants qui visent à faciliter le développement d'applications à partir de l'assemblage de briques logicielles préfabriquées appelées composants. Le processus proposé assure la séparation entre l'expertise du domaine d'application et l'expertise de sécurité, toutes les deux étant nécessaires pour construire une application sécurisée. La méthodologie proposée assure une intégration semi-automatique des patrons de sécurité dans le modèle initial. Cette intégration est réalisée tout d'abord lors de la modélisation de l'application à travers, dans un premier temps, l'élaboration de profils étendant les concepts du domaine avec les concepts de sécurité. Dans un second temps, l'intégration se fait à travers la définition de règles, qui une fois appliquées, génèrent une application sécurisée. Finalement, cette intégration est assurée aussi au niveau de la génération du code fonctionnel de l'application en intégrant le code non-fonctionnel relatif à la sécurité à travers l'utilisation des aspects. L'utilisation de l'approche orientée aspect garantit que l'application des patrons de sécurité est indépendante de toute application particulière. Le processus proposé est décrit avec le standard SPEM.

Ce travail a été concrétisé par un outil nommé SCRI-TOOL pour SeCurity patteRn Integration Tool. Cet outil permet aux développeurs non experts en sécurité d'intégrer les différentes propriétés de sécurité (intégrées dans les patrons) dans une application à base

de composants. Afin d'illustrer l'utilisation de SCRI-TOOL, nous proposons une étude de cas portant sur le domaine des systèmes de soins distribués. Le choix d'une telle étude de cas s'explique par l'importance des exigences en termes de sécurité requises pour le bon fonctionnement d'une telle application. En effet, vue le grand nombre d'acteurs pouvant interagir, la sécurité est une exigence critique dans de tels systèmes. Cette étude nous a permis de mettre en évidence l'importance de la gestion de la sécurité à un haut niveau d'abstraction et la façon d'appliquer la méthodologie proposée sur un cas réel.

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Toulouse II, 5 Allée Antonio Machado, 31058 TOULOUSE cedex 9

Rahma Bouaziz

AN MDE PROCESS FOR SECURITY PATTERN INTEGRATION IN COMPONENT BASED APPLICATION

Supervisor : Bernard Coulette, UTM Professor , IRIT

Abstract

Security has become an important challenge in current software and system development. Most of designers are experts in software development but not experts in security. It is important to guide them to apply security mechanisms in the early phases of software development to reduce time and cost of development. To reach this objective, we propose to apply security expertise as security patterns at software design phase. A security pattern is a well-understood solution to a recurring information security problem. So, security patterns encapsulate the knowledge accumulated by security experts to secure a software system. Although well documented, patterns are often neglected at the design level and do not constitute an intuitive solution that can be used by software designers. This can be the result of the maladjustment of those patterns to systems context, the inexpertness of designers with security solutions and the need of integration process to let designers apply those pattern ?s solutions in practical situations and to work with patterns at higher levels of abstraction. To enable designers to use solutions proposed by security patterns, this thesis proposes a model driven engineering approach to secure applications through the integration of security patterns. Component-based approach is a powerful means to develop and reuse complex systems. In this thesis, we take component based software systems as an application domain for our approach to facilitate the development of applications by assembling prefabricated software building blocks called components. The proposed process provides separation between domain expertise and application security expertise, both of which are needed to build a secure application. Our main goal is to provide a semi-automatic integrating of security patterns into component-based models, and producing an executable secure code. This integration is performed through a set of transformation rules. The result of this integration is a new model supporting security concepts. It is then automatically translated into aspect-oriented code related to security. These aspects are then woven in a modular way within the functional application code to enforce specified security properties. The use of aspect technology in the implementation phase guarantees that the application of security patterns is independent from any particular implementation. In order to provide a clear comprehension of the SCRIP process, we have described it using the standard SPEM . This work is implemented in a software tool called SCRI-TOOL (SeCurity patteRn Integration Tool). This tool allows not security experts developers to integrate different security properties throughout the development cycle of an component based application. To illustrate the use of SCRI-TOOL, we propose a case study regarding electronic healthcare systems. The choice of such a case study is motivated by the great attention archived for such systems from academia and industry and by the importance of security in such systems. Indeed, be-

cause of the large number of actors that can interact in such systems, security is a critical requirement. This case study will also allow us to illustrate the proposed methodology to highlight the importance of security management at a high level of abstraction. As results of the application of this process, we obtain a health care application completely secure and meeting the requirements of medical context.

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Toulouse II, 5 Allée Antonio Machado, 31058 TOULOUSE cedex 9

Table des matières

Liste des figures

Liste des tableaux

Introduction

Modèles à base composants, patrons de sécurité, ingénierie dirigée par les modèles et transformation de modèles sont les quatre notions au cœur de cette thèse. Cette introduction présente le contexte scientifique dans lequel se situe ce travail ainsi que la problématique dans laquelle il s'insère.

Contexte et Motivations

De nos jours, l'ingénierie logicielle ne cesse de faire face à des applications de plus en plus complexes qui doivent évoluer vite, à moindre frais ainsi que dans des courts délais. Et ce en répondant aux besoins croissant des utilisateurs et aux nombres grandissant des fonctionnalités à intégrer jusqu'à l'obtention du produit final. Pour aider les développeurs à suivre cette évolution, des nouvelles techniques de construction de logiciels destinées à faire face à ces contraintes ont vue le jour. C'est le cas de l'approche à composants, qui vise à faciliter le développement d'applications à partir de l'assemblage de briques logicielles préfabriquées appelées composants.

La définition du terme composant reprise dans cette thèse est celle proposée par Szyperski [Szyperski, 1998] en 1998, et qui reste la plus utilisée dans la littérature. D'après cette définition, un composant est une unité de composition ayant des interfaces spécifiées de façon contractuelle, possédant uniquement des dépendances de contexte explicites. Un composant peut être déployé de manière indépendante et est sujet à composition par des tiers.

Le développement des applications à base de composants favorise la réutilisation d'un même composant dans plusieurs applications. Afin d'assister les développeurs dans le processus de conception, plusieurs modèles à base de composants ont été proposés. Les plus connus en milieu industriel sont EJB (Entreprise Java Beans) [EJB] de Sun et CCM (Corba Component Model) [CCM] de l'OMG . De même, plusieurs modèles ont été proposés dans le milieu académique comme Sofa 2 [Bures *et al.*, 2006] ou Fractal [Bruneton *et al.*, 2006]. Ces modèles permettent la représentation des applications ou des systèmes comme un assemblage de plusieurs briques appelés composants.

Les systèmes modélisés à travers les composants doivent assurer un ensemble de fonctionnalités. Outre les fonctionnalités applicatives, un composant peut requérir des services non fonctionnels pour son exécution comme la sécurité, la performance, la disponibilité, la configuration ou la fiabilité.

Dans le cadre de ce travail, nous nous intéressons aux propriétés non-fonctionnelles, en particulier la sécurité. La sécurité représente l'une des propriétés non-fonctionnelle les plus importantes dans le cycle de développement logiciel. Une faille de sécurité peut avoir des conséquences lourdes et graves avec un impact direct sur l'environnement et/ou sur les utilisateurs [Laprie, 1996]. La prise en compte de cette propriété se révèle donc une nécessité pour le bon fonctionnement des applications et des systèmes à base de composants.

Pour faciliter la prise en compte des différents aspects relatifs à la sécurité, les patrons de sécurité [Schumacher *et al.*, 2006] ont été proposés comme solution. Les patrons de sécurité sont une adaptation des patrons de conception [Alexander *et al.*, 1977] [Gamma *et al.*, 1995] au domaine de la sécurité. Ils encapsulent les connaissances et les expériences accumulées par les experts afin de résoudre des problèmes de sécurité d'une façon structurée et réutilisable. Un patron de sécurité est défini comme une solution générique à un problème de sécurité particulier et récurrent qui se pose dans un contexte spécifique. Ainsi et à travers les patrons de sécurité, les concepteurs et développeurs non experts en matière de sécurité ont la possibilité de résoudre des problèmes de sécurité et cela en respectant les directives et les recommandations proposées par les patrons. La définition de nouveaux patrons de sécurité ainsi que leurs application lors du développement de systèmes logiciels sécurisés restent parmi les enjeux majeurs pour les chercheurs et les industrielles [Schumacher, 2003]. Dans ce cadre-là, peu de travaux se sont intéressés à l'utilisation des patrons de sécurité dans les architectures à base de composants. En effet nous constatons l'absence totale d'une démarche permettant l'intégration des solution proposées par les patrons lors d'un développement d'une application à base de composants.

Problématique et Contributions

L'intégration des patrons de sécurité dans un modèle à composants est donc la problématique majeure traitée de ce travail. En effet, plusieurs verrous restent à résoudre avant l'intégration effective et totale de ces solutions de sécurité dans un processus de développement logiciel. Dans la littérature, on constate l'existence de travaux qui proposent l'application des patrons de sécurité au niveau de la conception sans fournir les mécanismes de leur mise en œuvre [Mouratidis *et al.*, 2003] [Fuchs *et al.*, 2009]. On trouve aussi ceux qui proposent la mise en œuvre concrète au niveau code sans se soucier de leur intégration au niveau conceptuel [Horvath et Dörges, 2008b]. A notre connaissance, aucune technique d'intégration complète n'a été proposée. En plus, l'absence d'outils permettant la mise en œuvre de cette intégration représente aussi un frein devant leurs utilisations dans les modèles à composants.

La problématique majeure de cette thèse consiste à intégrer les solutions de sécurité proposées par les patrons dans les modèles à composants. Plusieurs travaux se sont intéressés à l'utilisation des patrons dans les modèles à objets, mais l'intégration de ces patrons dans les modèles à composants reste encore un challenge pour les développeurs vu la complexité des applications et la difficulté d'interpréter les solutions proposées par les patrons de sécurité.

L'intégration des patrons de sécurité dans un modèle à composants parait donc comme un défi pour les développeurs non experts en sécurité.

L'objectif de cette thèse est la proposition d'un processus complet basé sur une approche d'ingénierie dirigée par les modèles pour intégrer les solutions proposées par les patrons de sécurité dans le contexte des applications à base de composants. L'intérêt de notre approche est qu'elle permet une séparation claire entre les concepts relatifs aux composants logiciels et ceux liés à la sécurité. Notre approche peut être résumée en trois volets :

- Proposer un profil UML de sécurité permettant de regrouper les concepts de sécurité décrits par les patrons de sécurité ainsi que d'étendre le modèle de composants avec ces concepts.
- Définir les mécanismes (règles de transformations de modèles) permettant d'appliquer ces patrons dans un modèle à composants. C'est-à-dire de sécuriser une application à base de composants.
- Proposer un processus et un outillage permettant de guider les développeurs dans la phase d'intégration des solutions proposées par les patrons dans les modèles à base de composants.

Plus précisément, notre objectif est de permettre aux développeurs non experts en sécurité d'intégrer les propriétés de sécurité tout au long du cycle de développement d'une application à base de composants. Cette intégration est réalisée tout d'abord au niveau du modèle à composants. Afin d'assister les concepteurs tout au long de ce cycle d'intégration, nous proposons SCRIP (SeCurity PatteRn IntegratIon Process) un processus outillé permettant de faire cette intégration d'une façon semi-automatique. Cet outil permet en particulier la génération du code non- fonctionnel relatif à la sécurité de l'application.

Plan de la thèse

Outre cette introduction, ce manuscrit de thèse est composé de trois parties principales. **La première partie** dresse l'état de l'art du domaine. Cette première partie comporte quatre chapitres. Son objectif est de proposer une introduction aux domaines des composants, des patrons d'ingénierie et de l'ingénierie des modèles. Elle présente aussi les différentes approches et les travaux connexes existants dans la littérature.

- Le chapitre 1 présente la notion de composant ainsi que les motivations qui ont induit à son apparition. Ce chapitre présente un état de l'art sur les approches à composants ainsi que leurs supports des propriétés non-fonctionnelles.
- Le chapitre 2 s'intéresse à la présentation des patrons d'ingénierie et plus particulièrement les patrons de sécurité, les formalismes de leurs descriptions ainsi que leurs classifications.
- Le chapitre 3 introduit les principes généraux de l'IDM et présente un tour d'horizon sur les standards et technologies développés autour de l'IDM.
- Finalement, le chapitre 4 dresse un état des lieux des travaux visant à modéliser des systèmes sécurisés et les travaux qui s'intéressent à l'intégration et à l'instanciation des patrons d'ingénierie et des patrons de sécurité en particulier au sein d'un modèle à composants.

Une discussion et une conclusion sont proposées à la fin de cette première partie dans le but de se positionner par rapport à l'existant et de situer le travail proposé.

La deuxième partie de ce manuscrit présente notre proposition. Il s'agit d'un processus IDM pour l'intégration des patrons de sécurité dans un modèle à composants. Cette intégration est réalisée tout d'abord lors de la modélisation de l'application à travers, dans un premier temps, l'élaboration de profils étendant les concepts du domaine avec les concepts de sécurité. Dans un second temps, l'intégration se fait à travers la définition de règles, qui une fois appliquées, génèrent une application sécurisée. Finalement, cette intégration est assurée aussi au niveau de la génération du code fonctionnel de l'application en intégrant le code non-fonctionnel relatif à la sécurité à travers l'utilisation des aspects. Nous proposons un processus complet, allant de la conception à la génération du code. Le résultat de l'application de ce processus est un modèle de composants sécurisé. Dans un second temps nous proposons une description du process présenté en utilisant le standard SPEM (Software Process Engineering Meta-model.

Nous concluons cette partie par une discussion ainsi qu'une conclusion.

Afin d'assister les développeurs lors de l'utilisation de ce processus, **la troisième partie** de ce manuscrit, présente SCRI-TOOL (SeCurity PatteRn IntegratIon TOOL) un outil permettant l'exploitation du processus décrit précédemment.

– Le chapitre 6 porte sur l'implémentation et l'expérimentation de notre approche. Il décrit premièrement la réalisation de l'outil d'intégration SCRI-TOOL, qui a été implémenté en adaptant l'atelier Eclipse. En deuxième lieu, nous montrons l'utilisation de cet outil dans la l'intégration de la sécurité dans le cadre d'une étude de cas.

Une discussion des résultats obtenus est aussi présentée afin de mettre en avant les avantages de ce processus ainsi que ses limites et les améliorations à proposer.

La dernière partie du manuscrit est une conclusion générale qui dresse le bilan de cette thèse. Nous présentons les différents résultats obtenus ainsi que les contributions qui en résultent. Des perspectives à court et à moyen terme sont aussi présentées. Une liste des publication produites pendant cette thèse, ainsi qu'une liste des acronymes sont fournies à la fin de ce document.

Des annexes complètent ce document pour en illustrer certaines parties : l'annexe A décrit les sources ATL du module d'intégration des patrons ; l'annexe B présente le code du template de génération de code fonctionnel de l'application et finalement l'annexe C illustre le code de template de génération de code aspects généré pour les patrons de sécurité.

Première partie

État de l'art

1 Approche à composants

1.1 Introduction

Les composants logiciels constituent aujourd'hui une technologie très répandue dans l'industrie de développement du logiciel. L'approche à composants est apparue dans les années 1990 [Szyperski, 1998]. Le principe du paradigme composant est de diviser un système complexe en sous-systèmes de complexité moindre réutilisables dans d'autres systèmes. Cette approche repose sur le principe de l'assemblage de composants pour le développement desapplications. L'apparition de ce paradigme est dû d'un coté à la volonté de fournir des composants préfabriqués souvent appelés "composants sur étagères" dans le but de réduire le temps de développement d'applications et d'accroître la productivité, et d'un autre côté à la volonté de proposer des remèdes au approches antérieures, telle que l'approche orienté Objet [Wuyts et Ducasse, 2001].

Un modèle à composants définit les caractéristiques des composants, ainsi que leur assemblage. Il propose aussi un support d'exécution. Plusieurs modèles à composants ont été proposés et utilisés dans le monde industriel et académique. Dans cette section, nous présentons un ensemble représentatif de modèles à composants : CCM [CCM], EJB [EJB], UML 2.0 [UML, a], Fractal [Bruneton *et al.*, 2006],ect.

Cette section présente d'abord les concepts principaux de l'approche : composant, port, connecteur, etc. Puis, un bref aperçu de diverses implémentations de cette approche est présenté. Pour finir, la sous-section 1.5 dresse un bilan sur les différents modèles à composants, leurs différents concepts et la section 1.4 met l'accent sur leur support des propriétés non-fonctionnelles.

1.2 Concepts principaux

Dans cette section, nous présentons les concepts généraux de l'approche et des modèles à composants, indépendamment de la technologie utilisée.

1.2.1 Composant

Il n'existe pas une définition unique de ce qu'est un composant. Cependant, parmi le grand nombre de définitions proposées, nous retenons celle de Clemens Szyperski [Szy-

perski, 1998] qui reste la définition la plus utilisée et la plus adoptée dans la littérature : « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* »

Autrement dit, «*un composant logiciel est une unité binaire de composition ayant des inter-faces spécifiées de façon contractuelle, possédant uniquement des dépendances de contexte explicites. Un composant peut être déployé de manière indépendante et est sujet à composition par des tiers.*»

Un composant est caractérisé par une granularité. Il peut être aussi petit qu'une procédure simple, ou bien aussi large qu'une application entière. Deux types de composants existent :

- **Composant primitif**. Lorsqu'un composant représente une unité atomique non décomposable, il est nommé primitif ou de base. Généralement, il appartient au plus bas niveau de l'architecture d'un système et encapsule du code implantant les fonctionnalités métier tels que les structures de données et les fonctions mathématiques ;
- **Composant composite**. Un composant peut lui-même contenir un nombre fini d'autres composants interagissant entre eux, appelés sous-composants ou composants internes, permettant ainsi aux composants d'être emboîtés à un niveau quelconque. Dans ce cas, ce composant est dit composite. La notion de composant composite permet de manipuler des assemblages de composants formant une architecture logicielle, comme une seule unité réutilisable de modélisation ou de programmation. La notion de composant composite permet également de décrire une architecture à différents niveaux de granularité, ce qui facilite la lecture globale d'une architecture. La déclaration d'un composant composite consiste alors en la déclaration des instances de sous-composants primitifs et composites qui le composent et la description des interconnexions de ces instances. Un composant composite a ses propres interfaces. Il est alors nécessaire de lier ces interfaces à celles de ses sous-composants afin de pouvoir exécuter les services du composant composite.

Aussi selon le niveau d'abstractions, un composant peut être considéré selon deux vues principales :

- **vue boîte noire** (black box). La vue boîte noire d'un composant décrit les services fournis et requis par le composant et masque les détails de son implémentation. La Figure 1.1 illustre la vision boîte noire d'un composant. Les services (fournis et requis) d'un composant sont exposés par le biais de points d'interaction qui permettent de gérer les collaborations du composant avec son environnement et les autres composants. Suivant les approches, ces points d'interaction peuvent être de trois sortes : l'opération, l'interface ou bien le port. Dans la figure 1.1 les points d'interaction sont des interfaces ;

Figure 1.1 — Vue boîte noire du composant

- **vue boîte blanche** (white box). La figure 1.2 montre une vue boîte blanche du composant qui correspond à une vue sur son implémentation. Elle présente les composants

internes du composant composite. Cette représentation est connue par le nom *composition hiérarchique*. Un nombre important d'approches académiques comme UML, Sofa 2, Koala ou encore Fractal autorisent la composition hiérarchique.

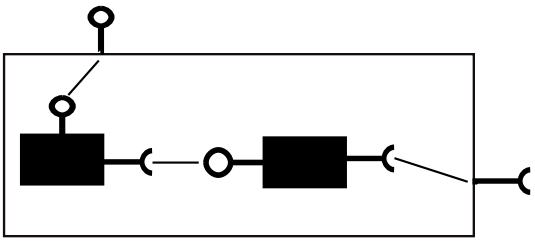


Figure 1.2 — Vue boîte blanche du composant

1.2.2 Interface

Les composants spécifient les fonctionnalités qu'ils offrent et celles qu'ils demandent à travers une ou plusieurs interfaces. Les interfaces logicielles définissent généralement l'ensemble des opérations qu'ils mettent en œuvre. L'interface est le type de point d'interaction le plus utilisé dans les approches à composants. Communément, une interface peut être de deux types (voir figure 1.3) :

- une interface fournie dans le cas où elle définit un service implémenté ou réalisé par le composant ;
- une interface requise dans le cas où elle spécifie un service invoqué par le composant.

Figure 1.3 — Représentation graphique des interfaces requises et fournies d'un composant

1.2.3 Port

Un composant possède des ports (voir figure figure 1.4) qui sont des points d'accès directs à des services offerts par d'autres composants. Chaque port d'un composant est caractérisé par un type donné (Type entier pour les propriétés, type d'événement, une direction telle que entrée, sortie ou entrée-sortie). Les ports possèdent aussi des sémantiques d'interaction telles que les invocations de méthodes synchrones ou asynchrones en point-à-point, un événement asynchrone comme dans le mode publish/subscribe, ou un flux de données en mode point-à-point.

1.2.4 Connecteur

Un connecteur est une entité qui assure les interactions entre deux composants par l'intermédiaire de leurs interfaces ou leurs ports. Les règles qui permettent la gestion du com-

Figure 1.4 — Représentation graphique d'un port d'un composant

portement de ces interactions sont définies par le connecteur. Un connecteur peut prendre des formes très variées selon les approches, comme par exemple une entité de première classe, elle-même considérée comme un composant et dotée aussi d'interfaces qui assurent le lien entre les interfaces métier qui participent à l'interaction. À l'inverse, un connecteur peut également représenter une simple liaison, spécifiant une dépendance entre composants par exemple, mais ne rajoutant aucun comportement supplémentaire.

1.2.5 Services non fonctionnels

Outre les services fonctionnels applicatifs (fournis et requis), un composant peut requérir des services non-fonctionnels pour son exécution. Il peut demander à exécuter une fonctionnalité du système d'exploitation (telle que l'accès à un périphérique, création de fichier,...etc) ou d'un intergiciel (middleware) comme accéder au service de gestion de transactions, de sécurité, de persistance, ou autre. Deux problèmes apparaissent lors de la gestion des services non-fonctionnels :

- Lorsque plusieurs propriétés non-fonctionnelles sont gérées, le code correspondant à leurs gestion peut devenir beaucoup plus important que celui qui implémente les fonctionnalités mêmes du composant.
- Les deux sortes de code peuvent se trouver mélangées dans l'implémentation du composant.

Pour résoudre cette problématique, on applique en général le principe de séparation des préoccupations, c'est-à-dire qu'on décrit de façon externe (hors du code) une partie des propriétés non-fonctionnelles des composants. Divers modèles à composants proposent des moyens permettant de gérer indépendamment les fonctionnalités initiales de composant et le code associé aux propriétés non-fonctionnelles. Dans la pratique, ceci est réalisé à travers l'utilisation d'intermédiaires entre les clients et les instances de composant, comme les conteneurs ou les contrôleurs (voir section 1.4).

1.2.6 Modèle à composants

Un modèle de composant définit toutes les caractéristiques et les contraintes que les composants et les frameworks d'application [1] des composants doivent satisfaire. Un modèle de composant est un ensemble de normes, de conventions et de règles que les composants doivent respecter. Il vise à fournir :

- des règles pour la spécification des propriétés des composants ;

1. Les frameworks peuvent être classés en deux catégories selon leur domaine d'utilisation : (i) les frameworks métier (framework verticaux) : ils encapsulent l'expertise d'un domaine particulier et (ii) les frameworks d'application (framework horizotaux) : ils encapsulent l'expertise technique applicable à plusieurs domaines.

– des règles et des mécanismes de composition des composants, y compris les règles de composition des propriétés.

Ces règles assurent que les composants développés par différents concepteurs et entreprises pourront être correctement installés dans des catalogues de composants, déployés dans les environnements d'exécutions et interagir au moment de l'exécution avec d'autres composants. En ce sens, un modèle de composant est la pierre angulaire de la normalisation pour le développement de logiciels à base de composants. Dans ce sens, Heineman et Councill [Heineman et Councill, 2001] proposent une définition du composant basée sur la définition d'un modèle de composants :

« A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard .»
«A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that con-form to the model.»

En appliquant ces concepts, l'ingénierie de logiciel à base de composants a prouvé son intérêt dans plusieurs domaines. Ce succès est mis en évidence par la prolifération des modèles de composants qui existent aujourd'hui.

1.3 Études des modèles à composants

Dans cette section, nous passons en revue quelques approches représentatives des modèles à base de composants, à savoir Corba Component Model (CCM) [CCM] de l'OMG, EJB [EJB] (Enterprise JavaBeans) de Sun, Fractal [Bruneton *et al.*, 2006] de France Télécom R& D au sein du consortium Object web [2] ,Sofa 2.0 [Bures *et al.*, 2006] développé par le groupe de recherche en systèmes distribué à l'université de Charles à Prague, Koala [van Ommering *et al.*, 2000] de Philips et enfin le modèle à composants UML2.0 [UML, a] de l'OMG (Object Management Group). Parmi les modèles à composants présentés dans cette section, nous détaillons le modèle adopté dans cette thèse : le modèle de composants UML 2.0. Ce choix sera justifié et motivé dans la section 1.2.3.

1.3.1 EJB

Le modèle "Enterprise Java Beans" (EJB) [EJB] a été proposé par Sun Microsystem en 1998. Ce modèle propose un ensemble de bibliothèques (classes) Java pour développer des applications distribuées à base de petits serveurs logiciels. Les spécifications EJB fournissent un cadre et des règles pour construire ce type d'applications. EJB fournit un environnement d'exécution stable et mature qui est utilisé dans de nombreuses applications d'entreprise. Comme il utilise un modèle de composant plat et donc ne supporte pas la composition, EJB n'a pas de problème avec l'ajout dynamique et la suppression de composants.

La figure 1.5 représente un modèle abstrait des EJB. L'interface distante de l'Entreprise JavaBean définit la vue cliente de l'EJB. En plus de cette interface métier, un composant EJB offre une interface pour accéder aux métadonnées de l'instance de composant et une

2. ObjectWeb Consortium, http ://www.objectweb.org/ 1999.

interface qui gère le cycle de vie d'un composant (maison ou home interface). L'interface de maison permet de créer (ou de rechercher dans le cas d'un composant persistant) et de détruire une instance de composant.

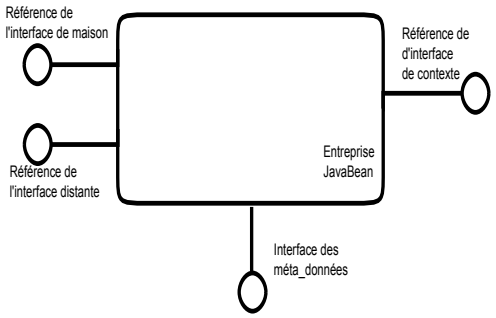


Figure 1.5 — Modèle abstrait des Enterprise Java Beans

L'avantage d'EJB est qu'il fournit un modèle et un environnement relativement simple qui a permis de lancer la technologie composant. C'est la première solution qui a séparé réellement la phase de codage, de déploiement et certains services non fonctionnels. Ainsi, on trouve dans ce modèle des descripteurs de déploiement séparés du code source Java.

Cette extraction de la description du déploiement est un premier pas vers la séparation des préoccupations. L'inconvénient du modèle EJB c'est qu'il est limité du point de vue conception. Une seule interface métier est fournie aux concepteurs. Par conséquent, il est impossible d'obtenir différentes vues de conceptions et donc d'implanter la séparation des préoccupations.

1.3.2 CCM

Le modèle à composants CORBA [CCM] nommé "Corba Component Model" est proposé par l'OMG Il est fortement inspiré du modèle EJB.. Cependant CCCM contient de nombreuses extensions qui le rendent plus abouti que EJB. Dans ce modèle, les composants CCM définissent les fonctions et services qui permettent aux développeurs d'applications de mettre en œuvre, gérer, configurer et déployer des composants.

Tel qu'illustré par la figure 1.6, un composant CCM est une entité composée de différents types de points de communication (multi-interfaces). La partie gauche de la figure présente les différentes interfaces fournies par un type de composant. Ces interfaces peuvent être de deux types : des facettes, qui sont des interfaces métiers synchrones offertes à l'environnement et des puits d'événements permettant la réception de messages asynchrones émis par d'autres composants.

La partie droite de la figure 1.6 présente les interfaces utilisées par un type de composant. De même que pour les interfaces fournies, deux types sont disponibles : les réceptacles sont des interfaces synchrones requises par le composant pour rendre son service et les sources d'événements permettent l'émission des messages asynchrones.

Le modèle CCM apporte de nombreuses améliorations en terme de qualité de développement par rapport au modèle EJB. Son principal avantage est de fournir plusieurs interfaces métiers. En plus, CCM offre une structuration claire des composants en définissant ce qui est offert à travers les Facettes, Sources, et Attributs, et ce qui est nécessaire au composant pour rendre ses services via les Réceptacles et les Puits.

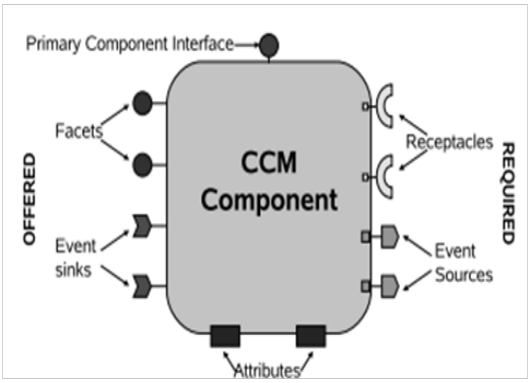


Figure 1.6 — Modèle abstrait d'un composant CCM (CORBA Component Model)

1.3.3 Koala

Le modèle Koala [van Ommering *et al.*, 2000] (C[K]omponent Organizer and Linking Assistant) a été proposé par Philips comme modèle à composants pour le développement de logiciels embarqués (pour les téléviseurs, set-top-boxes, etc.) Il utilise un modèle de composant hiérarchique. Les composants primitifs de Koala sont mis en œuvre comme un ensemble de fonctions C.

Les composants Koala sont des unités de conception, de développement et plus important encore de réutilisation. Même s'ils peuvent être très petits, ces composants nécessitent habituellement beaucoup d'efforts de développement. Un composant communique avec son environnement à travers des interfaces, comme le montre la figure 1.7.

Une interface Koala est un ensemble de fonctions sémantiquement liées. Un composant fournit une fonctionnalité à travers des interfaces et pour pouvoir assurer ces fonctionnalités le composant peut requérir aussi des fonctionnalités de son environnement à travers des interfaces. Dans ce modèle, les composants accèdent à toutes les fonctionnalités externes à travers des interfaces requises, même les services généraux tels que la gestion de la mémoire. Cette approche offre aux architectes une vision claire de l'utilisation des ressources du système.

1.3.4 Fractal

Fractal [Bruneton *et al.*, 2006] est un modèle à composants simple, flexible et extensible, développé par le consortium Object Web. Il définit un modèle à composants "générique" non orienté vers un domaine d'application particulier. Il peut être utilisé avec différents langages de programmation pour concevoir, mettre en œuvre, déployer et reconfigurer des systèmes.

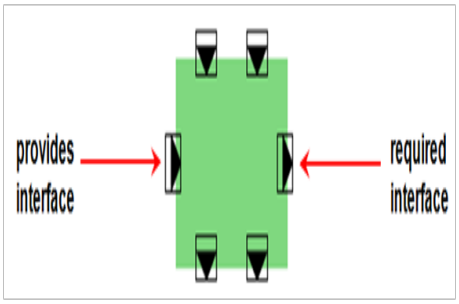


Figure 1.7 — Composant Koala

Le modèle à composants Fractal repose sur les concepts classiques de l'approche à composants : les composants sont des entités d'exécution qui sont conformes au modèle, les interfaces sont les seuls points d'interaction entre les composants. Les dépendances entre composants sont définies en termes de besoin (interface requise) et serveur (interfaces fournie) les liaisons de communication sont des canaux entre les interfaces de composants qui peuvent être primitifs, ou composites.

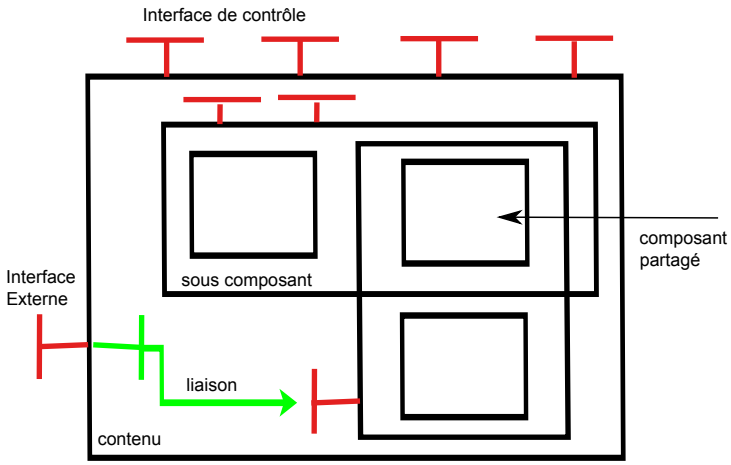


Figure 1.8 — Structure d'un composant Fractal

Un composant Fractal est généralement composé de deux parties, comme le montre la figure 1.8 :

- une membrane - qui possède des interfaces fonctionnelles et des interfaces permet-tant l'introspection et la configuration (dynamique) du composant.
- un contenu qui est constitué d'un ensemble de sous-composants.

Les interfaces d'une membrane sont soit externes, soit internes. Les interfaces externes sont accessibles à partir de l'extérieur du composant, alors que les interfaces internes sont accessibles uniquement par les sous-composants du composant. La membrane d'un composant est constituée d'un ensemble de contrôleurs. Chaque contrôleur a un rôle particulier comme le contrôle du comportement d'un composant et/ou de ses sous-composants. Un contrôleur peut, par exemple, permettre de suspendre/reprendre l'exécution d'un composant.

L'originalité du modèle Fractal réside dans le fait qu'il permet de construire des composants partagés. Un composant partagé est un composant qui est inclus dans plusieurs

composites. Le modèle Fractal fournit deux mécanismes permettant de définir l'architecture d'une application : l'imbrication (à l'aide des composants composites) et la liaison. La liaison est ce qui permet aux composants Fractal de communiquer. Fractal définit deux types de liaisons : primitive et composite. Les liaisons primitives sont établies entre une interface client et une et une interface serveur de deux composants résidant dans le même espace d'adressage. Les liaisons composites sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (ex. : stub, skeleton) reliés par des liaisons primitives.

Fractal est un modèle à composants généraliste. Il n'est pas dédié à un langage ou à un environnement d'exécution particulier contrairement aux modèles de composants industriels comme EJB et CCM. Fractal présente l'avantage d'être minimal et extensible et moins figé qu'EJB et CCM. La particularité du modèle Fractal par rapport aux modèles industriels c'est qu'il permet le partage de composants. Ainsi, un composant donné peut faire partie de plusieurs composants en même temps.

1.3.5 Sofa 2

Sofa 2 [Bures *et al.*, 2006] est un modèle à composants hiérarchique avec des composants primitifs ou composites. Contrairement au composant primitif, un composant composite peut contenir un ou plusieurs sous-composants.

La figure 1.9 représente une vue détaillée d'un composant Sofa 2. On peut distinguer trois éléments de base. Tout d'abord, le frame qui forme la frontière du composant. Deuxièmement, la partie de contrôle fournie par le Sofa 2 runtime. Et troisièmement, le contenu du composant qui peut être formé par une implémentation directe du code métier ou par d'autres composants. La communication entre les interfaces et le contenu est assurée par la partie de contrôle qui peut réagir à différents types d'événements.

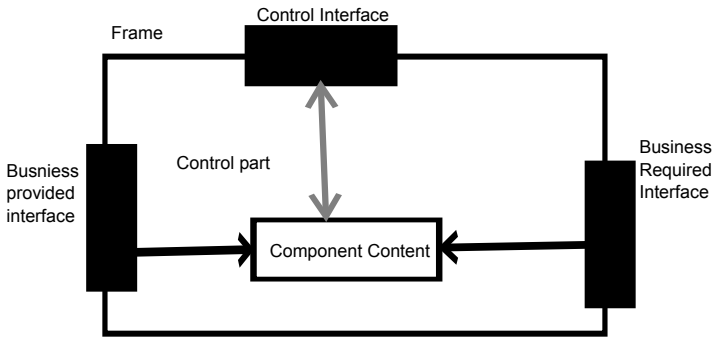


Figure 1.9 — Modèle d'un composant Sofa

Dans Sofa 2, les connecteurs sont de entités de première classe comme les composants. Chaque type de connecteur implémente une sémantique d'un type d'interaction spécifique. Les connecteur dans Sofa 2 sont aussi similaires au composant dans le sens ou ils sont spécifiés par frame de connecteur et de l'architecture du connecteur. Le frame du connecteur définit le type du connecteur en décrivant les services fournis par ce connecteur. L' architecture du connecteur décrit les connecteurs internes dans le cas ou le connecteur est un connecteur

composite. L'architecture décrit simplement le frame si le connecteur est primitif.

1.3.6 Modèle de composants UML 2.0

UML (UnifiedModelingLanguage) est un langage de modélisation graphique, semi-formel, largement répandu dans les deux mondes académique et industriel. Le métamodèle de composants proposé par UML 2.0 [UML, a] permet la spécification des composants, ainsi que l'architecture des systèmes à développer.

Dans sa version 2.0, adopté dans cette thèse, UML apporte plusieurs améliorations pour représenter les architectures logicielles en introduisant de nouveaux concepts : composant, connecteur, port, interface requise et interface fournie, etc.

UML2.0 spécifie un composant comme étant une unité modulaire, réutilisable, qui interagit avec son environnement par l'intermédiaire de points d'interaction appelés ports. Les ports sont typés par les interfaces : celles-ci contiennent un ensemble d'opérations et de contraintes ; les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Le comportement interne du composant ne doit être visible et accessible que par ses ports.

Il existe deux types de modélisation de composants dans UML2.0 : le composant basique et le composant composite (ou structure composite). La première catégorie défini le composant comme un élément exécutable du système. La deuxième catégorie étend la première en définissant le composant comme un ensemble cohérent de parties (appelées parts, chacune représentant une instance d'un autre composant).

La connexion entre les ports requis et les ports fournis se fait au moyen de connecteurs. Deux types de connecteurs existent : le connecteur d'assemblage et le connecteur de délégation. Un connecteur d'assemblage représente une connexion entre composant d'un même niveau hiérarchique, alors qu'un connecteur de délégation se trouve entre un port requis d'un composant et un autre port requis d'un de ses sous-composants, ou entre deux ports fournis. On peut voir un résumer de tout ceci dans la figure 1.10.

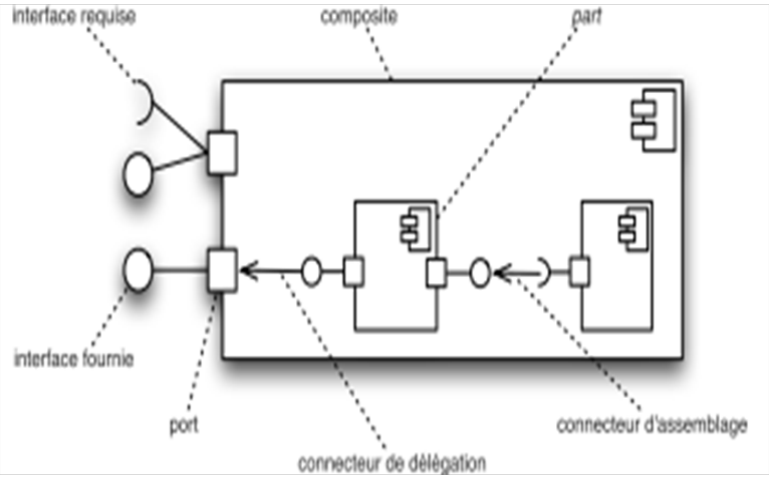


Figure 1.10 — Structure d'un composant UML 2.0

1.4 Support des Services non fonctionnels

Outre les services fonctionnels applicatifs (fournis et requis), un composant peut demander des services non fonctionnels pour son exécution. Par exemple, il peut demander l'exécution d'une fonctionnalité du système d'exploitation telle que l'accès à un périphérique ou la création d'un fichier. Il peut aussi demander l'exécution d'une fonctionnalité d'un intergiciel (middleware) comme accéder au service de gestion de transactions, de sécurité ou de persistance. Les fonctionnalités offertes par un système d'exploitation ou par un intergiciel sont dites *services non fonctionnels*. Un service non fonctionnel peut également désigner un service lié au cycle de vie des composants tel que les opérations de création, suppression, modification des composants ou leurs interconnexions. En effet, un composant est d'abord créé en instanciant un type défini de composant, connecté à d'autres instances de composant, puis activé. Durant son cycle de vie, une application peut être sujette à des reconfigurations éventuellement dynamiques, telles que désactiver des composants, supprimer une instance de composant, supprimer des liaisons entre composants, réaliser de nouvelles liaisons, etc.

Une partie des modèles à composants ont essayé de répondre au problème de la séparation des préoccupations, tels que EJB et CCM. Afin de permettre la gestion des services non fonctionnels indépendamment de la programmation des composants, deux concepts ont été utilisés dans les modèles à composants : le concept de conteneur et le concept de contrôleur.

1.4.1 Conteneur

Un conteneur est un environnement d'exécution qui contient un ensemble de composants. Le conteneur offre à ces composants un accès et une utilisation simplifiée des services non-fonctionnels.

Un conteneur gère les composants. Il est responsable de la création, la destruction et la réalisation des opérations de reconfiguration. Il permet également de faire appel aux fonctionnalités du système d'exploitation. De plus, comme un composant peut recevoir des requêtes de service ou faire appel à des services fonctionnels de composants appartenant à d'autres conteneurs, le conteneur s'occupe d'acheminer ces invocations depuis ou vers l'extérieur aux composants cibles. Cela dépend du modèle considéré, mais d'une façon générale, les conteneurs gèrent les aspects relatifs à la sécurité, la communication et les utilisateurs. Certains conteneurs gèrent aussi les files d'attente pour améliorer les performances de l'application.

La notion de conteneur a été notamment utilisée dans les modèles de composants EJB et CCM. Une notion plus large, le contrôleur, a été introduite dans le modèle à composants Fractal. L'intérêt du contrôleur est de permettre la personnalisation du contrôle offert aux composants.

1.4.2 Contrôleur

Un contrôleur est un méta-objet, faisant partie d'un composant et permettant de gérer les propriétés non-fonctionnelles requises par le composant. Un contrôleur a un rôle assez particulier, car il peut être chargé de fournir une représentation causalement connectée de la structure d'un composant (en termes de sous-composants). Il peut contrôler le comportement d'un composant et/ou de ses sous-composants, comme permettre de suspendre/reprendre l'exécution d'un composant. Il peut également connecter/déconnecter des sous-composants.

Par rapport au conteneur, un contrôleur ne fournit pas nécessairement un même flot d'exécution pour les composants qu'il contient (comme dans le cas où les composants sont distribués). Inversement, le conteneur ne constitue pas une partie d'un composant, comme c'est le cas pour un contrôleur. Le concept de contrôleur est plus large puisqu'il intègre la gestion du cycle de vie proposée par les modèles EJB et CCM, à l'intérieur d'un composant lui-même.

1.5 Synthèse et comparaison

L'approche à composants induit une méthodologie visant à faciliter le développement d'applications à partir de l'assemblage de briques logicielles préfabriquées appelées composants. Cette approche repose particulièrement sur une séparation entre l'étape de développement des différents composants et l'étape d'assemblage.

Un modèle à composants définit d'une manière rigoureuse la structure des composants qui le constituent et la façon d'assembler ces différents composants. Il permet ainsi de garantir un processus de développement propre à l'approche à composants. De plus, un modèle à composants est accompagné par un environnement d'exécution qui fournit un support aux applications pendant l'exécution. Les différents modèles de composants présentés dans ce

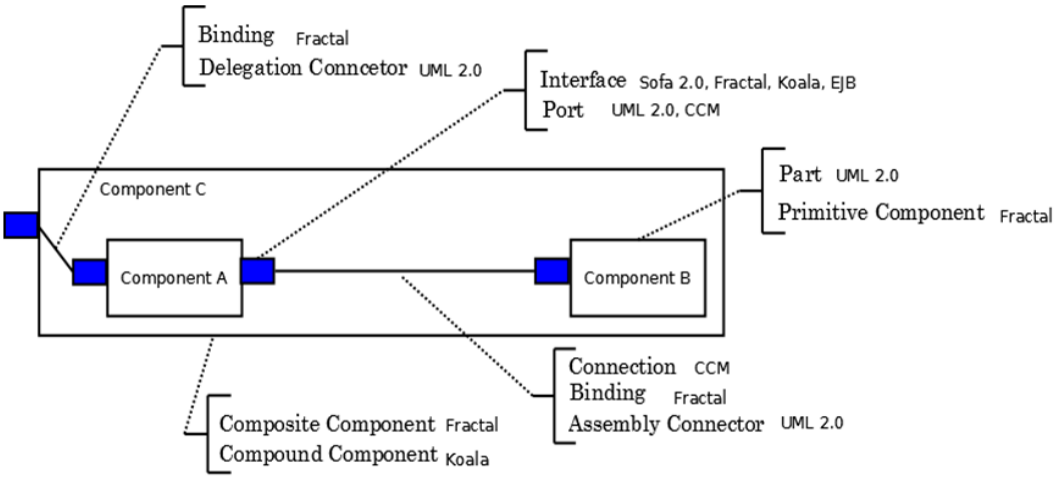


Figure 1.11 — Synthèse des concepts présents dans les modèles à composants (EJB, CCM, Fractal, Sofa 2, Koala et UML 2.0)

premier chapitre mettent l'accent sur un certain nombre de concepts. La figure 1.11 synthétise le support de ces différents concepts à travers les modèles à composants présentés dans cette première section.

A travers cette synthèse, nous pouvons voir comment les concepts présentés dans la première partie de ce chapitre sont implémentés. Ces concepts peuvent être partiellement présents dans les différents modèles à composants (par exemple la séparation entre la vue externe et la vue interne, ou l'existence d'interfaces de contrôle). Chaque technologie possède son propre modèle de composant et possède sa propre représentation de composant.

Cette synthèse nous permet aussi de conclure qu'il n'existe pas de modèle à composants "idéal". L'utilisation d'un modèle à composants particulier résulte donc d'un choix qui doit être guidé par les besoins applicatifs. Les applications de commerce électronique, par exemple, sont en général des applications à trois niveaux qui trouvent une bonne réponse dans les modèles EJB, alors que les applications de télécommunication sont quant à elles généralement réparties et requièrent des modèles de composants comme le CCM : un composant offre des services à n composants et utilise les services de m autres composants.

Afin de comparer les modèles à composants présentés auparavant, nous dressons un tableau récapitulatif des propriétés des modèles de composant présentés dans ce chapitre. La comparaison des modèles est réalisée selon plusieurs critères obtenue de la description des concepts réalisée dans la deuxième partie de ce chapitre. Ces critères concernant les propriétés générales, les propriétés de composant dont les propriétés non fonctionnelles et leur support. Le résultat de cette comparaison est résumé dans le tableau 1.1.

Tableau 1.1 — Comparaison entre les modèles à composants

	Propriétés générales			Propriétés de composant		
	Année	*Domaine d'application*	*Utili. indust.*	*Interface*	*Propriétés non Fonctionnelles*	*Support des Propriétés non Fonctionnelles*
EJB	1997	Applications diverses, réparties	Oui	Services métiers	Cycle de vie, métadonnées, sécurité, persistance, transaction, notification,...	Conteneur
CCM	2001	Applications réparties	Oui	Facette réceptacle puits source d'événement	Oui	Conteneur
Fractal	2002	Application diverses	Non	Interface serveur/ client, interface de contrôle	Diverses	Contrôleur
SOFA 2	1998	Application de composants distribués	Non	Interfaces fournies et requises	Oui	Conteneur
Koala	2000	Logiciels embarqués dans les produits TV	Oui	Interfaces fournies et requises	Sous formes de propriétés	-
UML 2.0	2002	Applications diverses	Oui	Interfaces fournies et requises		

1.6 Conclusion

Dans ce chapitre nous avons présenté d'une façon générale l'approche à composants en décrivant dans un premier temps les concepts principaux puis en détaillant quelques modèles académiques et industriels.

Dans un second temps, nous avons réalisé une comparaison entre les principaux modèles à composants existants.

A travers la comparaison des différents modèles à composants, nous pouvons conclure que les concepts décrits ne sont que partiellement présents (par exemple la séparation entre la vue externe et la vue interne, ou le concept de contrôleur ou conteneur). Un autre fait à noter est que certains modèles à composants sont très spécialisés par rapport à leur domaine d'application (EJB et CCM) tandis que d'autres modèles sont plus "génériques" (UML 2.0 et Fractal).

Nous choisissons dans la suite de cette thèse de travailler avec le modèle à composants proposé par UML 2.0. Ce choix est motivé par le fait que les outils de développement basés sur UML sont répandus. De plus, une grande communauté de concepteurs utilise UML 2.0 avec une large panoplie d'outils open source.

Notre objectif étant d'intégrer la sécurité dans les modèles à composant à travers les patrons de sécurité, dans le chapitre suivant, nous présentons de façon détaillée les patrons d'ingénierie et plus particulièrement les patrons de sécurité.

2 Patrons d'ingénierie et Sécurité

2.1 Introduction

La notion de patron a été proposée par l'architecte C. Alexander [Alexander *et al.*, 1977] [Alexander, 1979]. Dans une série de livres parue entre 1964 à 1979, Alexander a remarqué des ressemblances dans la conception des grands bâtiments, dues à leur environnement. Le résultat de cette réflexion a été la notion de patron.

Un patron est à la fois une description du contexte dans lequel il est applicable, le problème qu'il cherche à résoudre et la solution détaillée qui résout ce problème. L'utilisation des patrons d'ingénierie dans la conception de logiciels a été popularisée par le livre "Design Patterns", écrit par le Gang of Four [Gamma *et al.*, 1995].

Il est intéressant de noter que cette approche à base de patrons a changé radicalement l'approche de conception rationnelle qui mettait l'accent sur l'automatisation de la conception de la construction en utilisant des règles et des algorithmes [Coplien, 1998]. Selon lui, cette approche ne satisfait pas les exigences réelles des utilisateurs, par rapport à la conception en utilisant les patrons, qui progresse d'une manière évolutive et s'adapte au mieux aux besoins des utilisateurs.

Dans le domaine du logiciel, les utilisateurs (utilisateurs finaux, personnel de maintenance de logiciel, concepteurs, le développeur) sont intéressés par des qualités telles que la modularité, réutilisabilité, l'évolutivité, l'extensibilité, la sécurité, l'interopérabilité, la maintenabilité, coûts, etc.

Dans cette perspective, des patrons spécifiques à chaque domaine ont été proposés dans la littérature. En particulier, nous pouvons cité Yoder et al [Yoder et Barcalow, 1997] qui ont été les premier à présenter les patrons de sécurité en 1997. En effet, les patrons de sécurité regroupe la connaissance accumulée sur la sécurité. Ces derniers fournissent des lignes directrices pour la conception d'un système sécurisé. Les patrons de sécurité décrivent aussi un modèle générique précis pour un mécanisme de sécurité. Ils sont également utiles pour comprendre les systèmes complexes et à enseigner les concepts de sécurité.

Dans ce chapitre, nous introduisons les travaux ayant contribué à la genèse de la notion de patron d'ingénierie, et nous présentons d'une façon générale la notion de patron de sécurité ainsi que leur classification. Ensuite, nous décrivons en détail « Authorization Pattern » et « Session Pattern », deux patrons de sécurité, parmi d'autres, sur lesquels nous allons travailler dans la suite.

2.2 Les patrons d'ingénierie

Dans ce travail de thèse, nous nous basons sur la notion du patron comme définie par l'architecte C. Alexander [Alexander *et al.*, 1977]. Ce concept à fait l'objet d'une série de livre publiée entre 1964 et 1979.

L'idée du concept de patron consiste à capitaliser un problème récurrent d'un domaine et sa solution favorisant la réutilisation et l'adaptation de cette solution lors d'une nouvelle occurrence du problème. Il existe deux grandes familles de patrons :

- Les patrons produits expriment des savoirs, c'est-à-dire des fragments de spécification à adapter et à intégrer
- Les patrons processus expriment des savoir-faires, en particulier des fragments de démarche à suivre.

Il existe des formalismes de patrons combinant patrons produits et patrons processus afin de capitaliser tout ou partie de méthodes d'ingénierie [A. Conte, 2001]. Ce qui a donner naissance à la notion de patron d'ingénierie.

Les patrons d'ingénierie ont été introduits afin de capitaliser et de réutiliser des savoirs et des savoir-faire. Dans l'ingénierie logicielle, leurs usage est aujourd'hui reconnu, à tous les niveaux (analyse, conception, ...), comme un gage de qualité [Arnaud, 2008].

Pour ce travail, nous nous appuyons sur un système de patrons d'ingénierie bien connu, celui de [Schumacher *et al.*, 2006]. Notre but est de fournir mécanismes permettant l'intégration de ces patrons dans des modèles à base de composants.

Cette section introduit et définit le concept de patron d'ingénierie. Nous présentons tout d'abord les travaux ayant contribué à la genèse de la notion de patron.

2.2.1 Genèse des patrons d'ingénierie

L'architecte C. Alexander a remarqué avec ses collègues que certaines solutions de conception, dans le domaine de l'architecture des bâtiments, considérées comme efficaces et bien acceptées, étaient récurrentes et s'appliquaient dans différentes situations de construction. Cette idée a été concrétisée par la proposition de 253 patrons propres aux problèmes récurrents en architecture décrits d'une manière uniforme, pour des raison de clarté et de diffusion [Alexander *et al.*, 1977]. L'utilisation des patrons dans le domaine de l'informatique a vu le jour en 1987. Les premiers patrons ont été proposés par K. Beck et W. Cunninghan [Beck et Cunningham, 1987]. Il s'agissait d'une première adaptation du langage de patrons d'Alexander à la conception et à la programmation Objet. Un peu plus tard en 1991, un ensemble de patrons plus spécifiquement dédiés à la conception par objets a été élaboré par E. Gamma lors de ses travaux de thèse [Gamma, 1991]. Les patrons sont devenus populaires avec la parution d'un ouvrage intitulé "Design Patterns : Elements of Reusable Object-Oriente d Software" par le Gang of Four [Gamma *et al.*, 1995]. Depuis, de nombreux ouvrages et articles ont été publiés sur les patrons [Coad *et al.*, 1995], [Fowler, 1996], [Ambler, 1998b], [Larman, 2002], ect. La première conférence internationale dédiée à la notion de patron de conception date de 1994 (PLoP), la première conférence européenne s'est déroulée

en 1996 (EuroPLoP).

2.2.2 Définition des patrons d'ingénierie

Alexander affirme que : « un patron d'architecture, tout comme un patron de couture, capitalise un savoir-faire permettant de résoudre un problème récurrent du domaine (l'architecture ou la couture) ... chaque patron décrit à la fois un problème qui se produit très fréquemment dans votre environnement et l'architecture de la solution à ce problème ; de telle façon que vous puissiez utiliser cette solution des millions de fois sans jamais l'adapter deux fois de la même manière » [Alexander *et al.*, 1977]. Une définition similaire est proposée par Buschmann [Buschmann *et al.*, 1996] « Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts ». De manière générale, un patron constitue une base de savoir et de savoir-faire pour résoudre un problème récurrent dans un domaine particulier. La spécification de ces connaissances réutilisables [Hachan, 2006] :

- permet d'identifier le problème à résoudre par capitalisation et organisation de connaissances d'expériences,
- propose une solution possible, correcte, générale et consensuelle pour y répondre,
- et, offre les moyens d'adapter cette solution à un contexte spécifique.

2.2.3 Formalisme de description des patrons d'ingénierie

Les patrons sont décrits en langage naturel et le plus souvent enrichis par des schémas semi-formels, généralement des diagrammes UML [UML, a]. Les patrons sont aussi décrits à l'aide d'une structure fixe, qui améliore leurs lisibilité, que l'on nomme formalisme. Un formalisme est donc la structure (c'est-à-dire ensemble de rubriques) que l'on peut utiliser, pour décrire uniformément les spécifications de plusieurs patrons. Cette unification facilite la comparaison entre les patrons ainsi que la recherche d'information d'une manière systématique et améliore la lisibilité des patrons.

Les nombreux formalismes de description proposés dans la littérature sont essentiellement de deux types : les formalismes narratifs tels que ceux de C. Alexender [Alexander *et al.*, 1977], et les formalismes structurés tels que ceux de [Coad, 1992], [Gamma *et al.*, 1995], [Fowler, 1996], [Ambler, 1998a]. Par opposition aux formalismes narratifs par nature peu structurés et informels, le second type de formalisme offre une meilleure représentation des patrons. Ces formalismes diffèrent, le plus souvent, par le nombre et le degré de détail (plus ou moins élevé) de leurs rubriques ainsi que par le type des patrons qu'ils décrivent. Aussi il y a ceux qui accordent plus d'attention aux causes du problème alors que d'autres investissent plus dans la partie solution. Toutefois, tous les formalismes décrivent obligatoirement pour chaque patron, le triplet : contexte, problème, solution.

Voici un exemple de formalisme développé par Gamma et al. [Gamma *et al.*, 1995]. Il comporte 14 rubriques :

- **Nom.** Un label significatif qui traduit le principe du patron.
- **Classification.** Permet d'organiser les patrons selon deux critères. Le rôle qui traduit ce que fait le patron (création, structuration, comportement), et le domaine qui précise si le patron s'applique en général à une ou plusieurs classes ou objets.

- **Intention.** Précise le problème de conception soulevé par le patron.
- **Alias.** Donne d'autres noms connus pour le patron, s'il en existe.
- **Motivation.** Présente un scénario d'application du patron posant son problème dans un contexte particulier. Elle permet de mieux comprendre l'intérêt et la solution du patron.
- **Indications d'utilisation.** Reconnaît des situations dans lesquelles le patron peut être utilisé.
- **Structure.** Représente les classes participant dans la solution du patron et montre leurs interactions dans des diagrammes OMT.
- **Constituants.** Définit les classes et/ou objets participants ainsi que leurs responsabilités.
- **Collaboration.** Discute la manière dont les constituants collaborent pour assumer leurs responsabilités.
- **Conséquences.** Décrit comment le patron réalise ses objectifs et présente les résultats, compromis et impacts de l'application de la solution du patron.
- **Implantation.** Explique les directives et la technique d'implémentation du patron. Elle présente les pièges existants et propose des solutions types en fonction du langage utilisé, si elles existent.
- **Exemple de code.** Donne des parties de code (dans le langage C++) illustrant la solution du patron.
- **Utilisations remarquables.** Présente des imitations du patron considéré dans des applications connues (dans des frameworks, par exemple).
- **Patrons apparentés.** Référence d'autres patrons utilisant ou utilisés par celui-ci, leurs différences, la manière de les associer.

2.2.4 Intérêt des patrons d'ingénierie

En termes d'intérêts, les patrons d'ingénierie constituent un véritable moyen de transfert de connaissances. Ils permettent l'échange du savoir et du savoir-faire des développeurs expérimentés, ayant rencontré et résolu plusieurs fois les mêmes problèmes [Schmidt, 1995]. Les patrons offrent également un vocabulaire commun à l'équipe de développement [Fowler, 1996]. Ils donnent lieu à un un gain de temps et d'efficacité, tout en réduisant largement les tâtonnements fastidieux nécessaires à l'élaboration d' un système de qualité. De plus, tout en augmentant les facultés d'abstraction des développeurs, les patrons apportent également une meilleure compréhension des systèmes. Ils permettent, en effet, d'aborder des systèmes complexes de manière plus simple. Ils aident à mieux structurer les applications, guidant la perception du monde réel permettant d'en obtenir une description à un niveau d'abstraction élevé. Il devient ainsi facile, par exemple, de comprendre le fonctionnement d'un framework, sans qu'il soit nécessaire de se pencher sur les détails de son implémentation. Les patrons capturent donc les propriétés essentielles d'une architecture, tout en masquant les détails non pertinents. Ils facilitent, de la sorte, la communication autour d' abstractions, de concepts et de techniques que décrivent ou portent ces patrons, dans le but d'assurer une meilleure compréhension, évolution et maintenance des applications.

D'autant plus que la documentation de celles-ci se trouve enrichie et simplifiée, facili-

tant le transfert des spécifications, des modèles et des concepts, notamment avec le personnel non technique qui n'est pas apte à comprendre tous les détails d'implantation [Schmidt, 1995]. L'intérêt des patrons d'ingénierie réside dans le fait qu'ils apportent des solutions efficaces à divers problèmes d'analyse et de conception, permettant ainsi de créer des conceptions approuvées et réutilisables. Ils améliorent la documentation des solutions et par conséquent, elles deviennent plus facile à faire évoluer et à maintenir.

D'ailleurs, les nouveaux systèmes se servent désormais, de plus en plus, de ces patrons pour gérer certains services non fonctionnels (tels que la persistance, la distribution ou encore la sécurité), et offrir la possibilité d'adapter ces services à plusieurs environnements d'utilisation. L'objectif ici est d'améliorer la réutilisation des systèmes, tout en séparant la partie fonctionnelle de la partie non fonctionnelle. Dans la suite nous nous intéressons aux patrons de sécurité dans la mesure où notre objectif est de rajouter les aspects de sécurité à un modèle de composants (initialement non sécurisé).

2.3 Les patrons de sécurité

La sécurité est l'une des nombreuses propriétés non-fonctionnelles que les développeurs doivent assurer au cours de cycle de développement de logiciels. En fait, l'exigence de sécurité des systèmes actuels ne cesse d'augmenter. Plusieurs patrons de sécurité ont été proposés pour traiter de cette propriété [Fernández *et al.*, 1993], [Yoder et Barcalow, 1997], [Essmayr *et al.*, 1997], [Yoder et Barcalow, 2000], [Vivas *et al.*, 2003], [Schumacher *et al.*, 2006]. Dans cette section, nous commençons par définir le terme de sécurité. Puis nous introduisant les patrons de sécurité comme cas particulier des patrons d'ingénierie.

2.3.1 Définition de la sécurité

Dans le domaine de l'informatique, le terme "**sécurité**" est défini comme la combinaison de trois propriétés : la *confidentialité*, l'*intégrité* et la *disponibilité* de l'information [Laprie, 1996]. Notons que ces trois propriétés se rapportent à l'information, le terme d'information couvrant non seulement les données et les programmes, mais aussi les flux d'information, les traitements et la connaissance de l'existence de données, de programmes, de traitements, de communications, etc. Cette notion d'information va jusqu'à couvrir le système informatique lui-même, dont parfois l'existence doit être tenue secrète [Abou El Kalam].

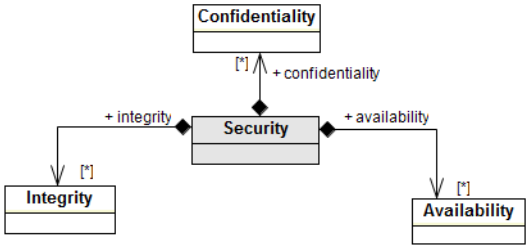


Figure 2.1 — Propriétés de la sécurité

Ces trois propriétés peuvent être définies comme suit (cf. figure 2.1) :

- **La confidentialité** est la propriété d'une information de ne pas être révélée à des utilisateurs non autorisés à la connaître. Ceci signifie que le système informatique doit :
  - empêcher les utilisateurs de lire une information confidentielle (sauf s'ils y sont autorisés)
- **L'intégrité** est la propriété d'une information de ne pas être altérée. Cela signifie que le système informatique doit :
  - empêcher une modification [1] indue de l'information, c'est-à-dire une modification par des utilisateurs non autorisés ou une modification incorrecte par des utilisateurs autorisés, et
  - faire en sorte qu'aucun utilisateur ne puisse empêcher la modification légitime de l'information. Par exemple, empêcher la mise à jour périodique d'un compteur de temps constituerait une atteinte à l'intégrité.
- **La disponibilité** est la propriété d'une information d'être accessible lorsqu'un utilisateur autorisé en a besoin. Cela signifie que le système informatique doit :
  - fournir l'accès à l'information pour que les utilisateurs autorisés puissent la lire ou la modifier, et
  - faire en sorte qu'aucun utilisateur ne puisse empêcher les utilisateurs autorisés d'accéder à l'information [Abou El Kalam].

Assurer la sécurité d'un système consiste à garantir le maintien de ces trois propriétés de sécurité. Ceci implique d'empêcher la réalisation d'opérations illégitimes contribuant à mettre en défaut ces propriétés, mais aussi de garantir la possibilité de réaliser des opérations légitimes dans le système. La description de ces différentes propriétés fait partie de la politique de sécurité du système. Assurer la sécurité du système, c'est donc assurer que les propriétés retenues sont vérifiées (et donc garantir la non-occurrence de défaillances vis-à-vis de ces propriétés).

2.3.2 Définition des patrons de sécurité

Un patron de sécurité intègre une partie des connaissances et de l'expérience des experts sur la sécurité d'une manière structurée. Il cible une variété de scénarios et de problèmes dans divers contextes. Les patrons de sécurité fournissent globalement les lignes directrices pour la conception et l'évaluation des systèmes sécurisés.

Les développeurs d'applications peuvent utiliser ces patrons, sans avoir l'expertise en sécurité et peuvent ainsi bénéficier de l'expérience d'experts en sécurité [Vivas *et al.*, 2003].

Ces patrons de sécurité sont également utiles pour comprendre les systèmes complexes et enseigner les concepts de sécurité. Dans [Schumacher *et al.*, 2006], un patron de sécurité est défini comme "*un patron qui décrit un problème particulier de sécurité récurrent qui se manifeste dans un contexte particulier et présente une solution éprouvée générique pour le résoudre*".

Yoder et Barcalow ont écrit le premier livre sur les patrons de sécurité [Yoder et Barcalow, 1997] [Yoder et Barcalow, 2000], même si plusieurs études avaient déjà mis au point des modèles orientés objet décrivant des mécanismes de sécurité [Fernández *et al.*, 1993] [Ess-

1. comprenant à la fois la création d'une nouvelle information, la mise à jour d'une information existante, et la destruction d'une information.

mayr *et al.*, 1997]. Schumacher et al.[Schumacher *et al.*, 2006] ont proposé un catalogue très complet de patrons de sécurité. Selon eux, " *Les patrons de sécurité répondent à l'ensemble de la sécurité dans la conception des systèmes, en utilisant les meilleures solutions pratiques pour montrer comment intégrer la sécurité dans le vaste processus d'ingénierie. Ces patrons sont essentiels pour les concepteurs des systèmes complexes qui veulent des solutions pratiques aux problèmes de sécurité classiques*".

2.3.3 Classification des patrons de sécurité

Dans cette section, nous présentons les différentes catégories de patrons de sécurité. Nous avons adopté la classification proposée dans le livre « Security Patterns : Integrating Security and Systems Engineering » [Schumacher *et al.*, 2006].

2.3.3.1 Identification et Authentification

Dans toute application Web, les utilisateurs sont besoin d'être identifiés et authentifiés. Le *I&A Requirements pattern* est le patron qui fournit des exigences générales communes pour tous les services d'authentification et d'identification sur le web. Il permet aussi de préciser si d'autres patrons d'identité et d'authentification sont nécessaires dans une application et dans un contexte spécifique. Les alternatives conceptuelles automatisées de l'I&A considèrent des techniques telles que les mots de passe ou la biométrie qui sont mis en œuvre dans le logiciel. De plus les I&A physique et procédurale constituent des choix dans certaines situations. Ceux-ci comprennent des mesures telles que la vérification de la carte d'identité d'une personne.

Le mot de passe avec l'identifiant constituent la méthode d'authentification la plus courante. Le *Password Authentication pattern* décrit comment concevoir, créer, gérer et utiliser des mots de passe en toute sécurité. Il permet également aux utilisateurs de choisir des mots de passe avec des niveaux de complexité variables et de gérer leurs mots de passe.

Le *Account Lockout pattern* protège les comptes d'utilisateurs des attaques automatisées visant à deviner un mot de passe ; ainsi, le système doit interdire par exemple toute nouvelle tentative après trois échecs.

Le *Biometrics Design Alternatives pattern* permet de sélectionner les options correctes pour les mécanismes biométriques pour satisfaire les exigences. Il existe de nombreux mécanismes disponibles, tels que la biométrie de reconnaissance faciale, l'empreinte digitale, la géométrie de la main, la vérification de la signature ou la vérification du locuteur.

2.3.3.2 Contrôle d'accès

Les patrons de sécurité dans la catégorie de contrôle d'accès représentent les politiques de sécurité de haut niveau.

Le *Authorization pattern* permet de décrire qui est autorisé à accéder à des ressources différentes et comment.

Le *Role-Based Access Control pattern* décrit comment attribuer les droits aux ressources en

fonction du rôle de l'utilisateur. De cette façon, les administrateurs doivent gérer un nombre raisonnable de droits pour des rôles différents et non pour chaque utilisateur séparément.

Dans certains environnements, les informations sensibles doivent être classées et protégées. Le *Multilevel Security pattern* décrit la façon de classer les informations sensibles et la façon de classer les données. Il décrit également la façon de classer les différentes unités organisationnelles et la façon de classer les utilisateurs. Les utilisateurs ne sont pas autorisés à violer les règles d'accès.

Le *Reference Monitor pattern* décrit comment utiliser le moniteur de référence pour faire respecter les droits d'accès aux ressources. Une entité demande des ressources du moniteur de référence qui vérifie l'autorisation.

Le *Role-Rights Definition pattern* prévoit une approche systématique pour mettre en œuvre la politique du moindre privilège en fonction des rôles des utilisateurs. Cette politique consiste à assigner seulement un minimum de droits.

2.3.3.3 Contrôle d'accès des systèmes

Outre les patrons proposés pour contrôler l'accès aux entités vulnérables dans un système défini, d'autres patrons de sécurité ont été proposés pour contrôler et sécuriser l'accès au systèmes en générale. Dans cette section nous nous intéressons à cette famille de patron.

Le service de contrôle d'accès est un service essentiel dans une application où l'accès permet d'accomplir une action qui est explicitement autorisée ou refusée. Le *Access Control Requirements pattern* fournit des exigences générales et décrit la fonctionnalité attendue et les propriétés du service dans le but de sécuriser le contrôle d'accès.

Le *Single Access Point pattern* offre un seul point d'accès au système. Ce patron simplifie la protection du système contre les abus. Le point d'accès vérifie le client puis il lui autorise ou refuse l'accès si nécessaire. Quand un point d'accès unique est en place, le Check Point pattern peut utiliser les moyens de I&A pour répondre aux tentatives d'effraction.

Le *Authenticated Session pattern* décrit l'utilisation d'une session d'authentification des utilisateurs. Le système crée une session authentifiée après la réussite de la connexion et permet à l'utilisateur, par exemple, d'accéder aux différentes pages d'un même site sans avoir besoin de s'authentifier lors de chaque demande de page.

Différents utilisateurs peuvent avoir des droits différents sur un même système. Il ya deux patrons pour mettre en œuvre les droits d'accès sur l'interface utilisateur. Tout d'abord, dans *Full Access withErrors pattern*, l'interface utilisateur expose toutes les fonctionnalités pour l'utilisateur mais le simple fait d'utiliser une fonctionnalité non autorisée induit une erreur. Deuxièmement, dans *Limited Access pattern*, l'interface utilisateur expose les fonctionnalités auxquelles l'utilisateur a accès donc l'interface utilisateur est différente pour les utilisateurs ayant des privilèges différents.

2.3.3.4 Comptabilité

Les personnes responsables du système ont besoin de connaître tous sur les événements de sécurité. La vérification de la sécurité et de l'accounting répond à ce besoin par le suivi des événements de sécurité. Le *Security Accounting Requirements pattern* fournit des exigences générales.

L'audit est un processus visant à analyser les journaux ou d'autres informations d'un événement. Le *Audit Requirements pattern* décrit les exigences et la conception d'audit permet de satisfaire ces exigences. De plus, la conception d'audit permet également d'identifier les besoins d'accès.

Les informations enregistrées peuvent être utilisées pour reconstituer les événements et pour analyser les problèmes. Le *Audit Trails & Logging Requirements pattern* définit les exigences pour effectuer ce type d'analyse. Ce patron permet de concevoir une solution de vérification et de mécanisme d'accès.

La détection d'intrusion automatise la détection des événements de sécurité dans les parties critiques du système ainsi que la réponse rapide aux événements de sécurité. Le *Intrusion Detection Requirements pattern* permet de capturer les exigences pour concevoir un service de détection d'intrusion à l'aide du patron de conception Intrusion Detection pattern.

Le *Non-repudiation pattern* crée des liens entre les événements et les acteurs. Son but est de fournir des listes gardant des traces sur qui a fait quoi pour que les acteurs ne puissent pas nier par exemple ce qu'ils ont fait.

2.3.3.5 Applications Internet sécurisées

Les informations sensibles peuvent être protégées à l'aide du *Information Obscurity pattern*. Habituellement, il est réalisé par le cryptage des données et l'obscurcissement des informations à propos de l'environnement. Le *Client Data Storage pattern* utilise le chiffrement pour stocker des données sur le client. Cela évite les problèmes où le client peut modifier, par exemple, les cookies, les champs masqués ou des paramètres d'URL. Le *Encrypted Storage pattern* fournit de l'aide pour crypter les données sur le disque du serveur. Le modèle de mise en œuvre Invisible cache les rouages internes d'une application ce qui rend plus difficile pour un attaquant de nuire le système.

Le *Secure Channels pattern* décrit comment protéger la communication à travers le réseau public. L'utilisation d'un canal sécurisé est essentielle lors du transfert des informations sensibles. Si les interactions avec l'utilisateur et le service sont sensibles, la détermination de l'identité exacte des parties est essentielle. Le *Known-Partners pattern* décrit comment les identités peuvent être identifiées de manière unique.

Dans les applications Web, le client peut entrer toutes les données qu'il veut. Certaines de ces données peuvent être dangereuses vulnérable pour le système et ne doivent pas être acceptées. Le *client Input Filters pattern* décrit quelles mesures doivent être prises pour protéger l'application.

En plus du pare-feu qui filtre les paquets, le serveur Web peut être protégé par le *Protection Reverse Proxy pattern*. Quand il y a de nombreuses applications provenant d'hôtes diffé-

rents qui composent le site Web, l'intégration du Reverse Proxy pattern permet de masquer la distribution physique des machines. Ainsi que le *Front Door pattern* fournit une connexion unique et une session unique pour plusieurs services.

2.3.4 Synthèse

Un tableau qui synthétise la classification des patrons de sécurité est présentée ci-dessous (tableau 2.1).

2.4 Conclusion

Un patron de sécurité intègre des solutions génériques et validées par des experts en sécurité qui peuvent être appliquées à des problèmes de sécurité dans différents contextes. Les patrons de sécurité sont un moyen pratique de créer des systèmes sécurisés et réutilisables. Pour atteindre cet objectif, ces patrons de sécurité sont proposés comme une bibliothèque dans un format permettant de capturer l'expertise en sécurité et de la communiquer aux novices. Même s'ils sont bien documentés, ces patrons sont souvent négligés au niveau de la conception et ne constituent pas une solution intuitive pour les concepteurs de logiciels. Cela peut être dû à plusieurs facteurs tels que (1) l'inadaptation de ces solutions au contexte des systèmes, (2) l'inexpérience du concepteur d'application avec ces solutions de sécurité ou (3) l'absence d'un processus d'intégration permettant aux concepteurs d'appliquer les solutions de ces patrons dans des situations pratiques et de travailler avec ces patrons à des niveaux d'abstraction plus élevés.

Dans ce chapitre nous avons introduit la notion de patron d'ingénierie et mis en évidence l'intérêt des patrons pour sécuriser les systèmes.

Pour inciter les concepteurs d'application à tirer profit des solutions de sécurité proposées par les patrons de sécurité, nous pensons qu'il faut leurs fournir des mécanismes d'intégration de ces patrons. Nous proposons dans le cadre de ce travail, un processus utilisant lors de la phase de conception les techniques de l'Ingénierie Dirigée par les Modèles, qui favorise l'utilisation systématique de modèles tout au long de cycle de développement de système, et dans la phase de génération de code, le paradigme de la programmation orienté aspects, qui favorise la séparation des préoccupation. Dans le chapitre suivant nous présentons ces deux paradigmes.

Tableau 2.1 — synthèse de la classification des patrons de sécurité

Identification et Authentification	**Contrôle d'accès**	**Contrôle d'accès des systèmes**	**Comptabilité**	**Applications Internet sécurisées**
I&A Requirements pattern	Authorization pattern	Access Control Requirements pattern	Security Accounting Requirements pattern	Information Obscurity pattern
Password Authentication pattern	Role-Based Access Control pattern	Single Access Point pattern	Audit Requirements pattern	Client Data Storage pattern
Account Lockout pattern	Multilevel Security pattern	Authenticated Session pattern	Audit Trails & Logging	Secure Channels pattern
Biometrics Design Alternatives pattern	Reference Monitor pattern	Authenticated Session pattern	Intrusion Detection Requirements pattern	Known-Partners pattern
-	Role-Rights Definition pattern	Full Access withErrors pattern	Non-repudiation pattern	client Input Filters pattern
-	-	Limited Access pattern	-	Protection Reverse Proxy pattern
-	-	-	-	Front Door pattern

3 Ingénierie dirigée par les modèles et Programmation Orientée Aspects

3.1 Introduction

Dans ce chapitre, nous présentons les deux principaux paradigmes utilisés dans le cadre de cette thèse. Nous nous intéressons donc dans cette partie à présenter le principe de l ?ingénierie dirigé par les modèles et le paradigme de la programmation orientée aspects.

Ce troisième chapitre est par conséquence divisé en deux principales sections. La première s'intéresse à l'ingénierie dirigé par les modèles ainsi que les définitions des concepts de base. Par la suite, nous décrivons les technologie et standards de l'OMG, développé autour de l'IDM, particulièrement l'approche MDA, la notion de transformation de modèles ainsi que les langages/outils associés. ainsi que le langage de modélisation UML [UML, a], les pro ?ls UML et le standard de modélisation de processus SPEM 2 [SPEM, 2008].

Dans la seconde section, nous présentons le paradigme de la programmation orienté aspects. nous présentons dans un premier temps les raisons qui nous ont poussés à utiliser ce mécanisme. Nous pré- sentons par la suite les concepts fondamentaux ainsi que le principe de décomposition et recomposition de la programmation par aspects, pour enchaîner sur la notion de langage de programmation par aspects, en particulier, le langage AspectJ.

3.2 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM) [France et Rumpe, 2007] [Bézivin, 2004a] [Jézéquel *et al.*, 2006], connue sous le terme MDE (Model-Driven Engineering) en anglais, est une forme d'ingénierie générative, par laquelle tout ou partie d'une application informatique est générée à partir de modèles. Le terme IDM a été proposé pour la première fois par [Kent, 2002].

L'IDM offre un cadre méthodologique et technologique qui permet d'unifier différentes façons de faire dans un processus homogène. L'IDM permet cette unification grâce à l'utilisation importante des modèles et des transformations entre les modèles. L'utilisation intensive de modèles permet un développement souple et itératif, grâce aux raffinements et

enrichissement par transformations successives.

L'IDM place les modèles au cœur du processus de développement des applications. Par rapport à l'approche traditionnelle dans laquelle l'artefact clé est le code lui-même et dans laquelle l'utilisation de modèle a tendance à être limitée à des fins de documentation, le développement orienté modèle peut alors être vu comme un ensemble de transformations de modèles partiellement ordonné, chaque transformation prenant des modèles en entrée et produisant des modèles en sortie, jusqu'à l'obtention du code exécutables.

L'objectif principal de l'ingénierie des modèles est d'améliorer la productivité du développement logiciel en s'appuyant sur l'utilisation de langages dédiés et de transformations de modèles. L'IDM repose sur les trois principes suivants : (i) la réutilisation des modèles, (ii) les modèles doivent être indépendants de toutes technologies (iii) la séparation des préoccupations métier et de la plateforme de mise en œuvre.

3.2.1 Définitions et principaux concepts

Modéliser le monde réel était et est toujours l'une des préoccupations principales de l'informatique. Cette modélisation consiste à créer un modèle informatique à partir du monde réel qu'on peut appeler aussi le domaine d'application. Cette section définit les concepts de base de l'ingénierie des modèles, à savoir la notion de modèle et de méta-modèle, et les relations qui existent entre ces concepts.

Dans [Bézivin et Gerbé, 2001], *un modèle est défini comme une représentation d'un (d'une partie) d'un système construit pour un objectif précis. Le modèle doit répondre aux questions que les utilisateurs se posent sur le système qu'il représente*.

Dans le contexte de l'IDM un modèle est défini dans [Kleppe *et al.*, 2003a] comme *une description d'un (d'une partie) d'un système dans un langage bien défini. Un langage bien défini est un langage qui a une syntaxe et une sémantique bien définie et qui est interprétable par un outil*.

On déduit de cette définition la première relation majeure de l'IDM, entre le modèle et le système qu'il représente, appelée **"représentationDe"** dans [Atkinson et Kühne, 2003] ,[Bézivin, 2004b] [Seidewitz, 2003].

Un méta-modèle est *un modèle qui définit le langage d'expression d'un modèle* [OMG, 2011], c.-à-d. le langage de modélisation. Dans l'IDM, on appelle méta-modèle toute représentation visant à définir les éléments que l'on va retrouver dans un modèle.

La notion de méta-modèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée **"conformeA"**.

Il faut noter que dans l'IDM, tout est modèle. Même un méta-modèle est un modèle, et doit donc être conforme à son méta-modèle. La relation "conformeA " pourrait donc s'appliquer un nombre indéfini de fois, chaque méta-modèle étant lui-même un modèle conforme à un méta-modèle supérieur. Autrement dit, un modèle est conforme à un méta-modèle qui est lui-même conforme à un méta-méta-modèle.

Ces deux relations permettent ainsi de bien distinguer le langage qui joue le rôle de système, du (ou des) méta-modèle(s) qui joue(nt) le rôle de modèle(s) de ce langage. La Figure 3.1 illustre ces notions et ces relations de base de l'IDM.

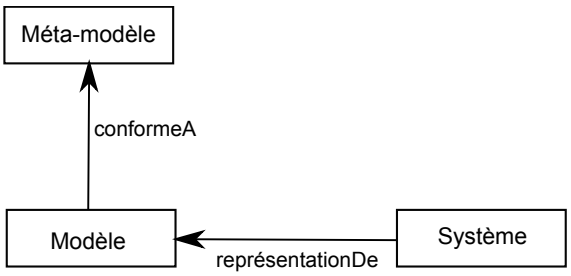


Figure 3.1 — Notions de base en ingénierie des modèles [Bézivin, 2004b]

3.2.2 Architecture de méta-modélisation

Dans le cadre de l'IDM, l'OMG a défini une architecture de méta-modélisation à quatre niveaux. L'architecture proposée est présentée par la figure 3.2. Dans cette architecture, les éléments de chaque couche sont décrits par les éléments de la couche en dessus. Cette architecture inclut le niveau du système réel à modéliser. Cette architecture est hiérarchisée en quatre niveaux. En partant du bas :

- **Le niveau M0** (ou instance) correspond au monde réel. Ce sont les informations réelles de l'utilisateur correspondant au système à modéliser. Ce sont les données que l'on désire modéliser.
- **Le niveau M1** (ou modèle) est composé de modèles. Il décrit les informations de M0. Un simple modèle UML fait partie de ce niveau. Les modèles de niveau M1 sont des instances de méta-modèle du niveau M2.
- **Le niveau M2** (ou méta-modèle), définit le langage de modélisation et la grammaire de représentation des modèles de niveau M1. Par exemple : le *méta-modèle UML* qui est décrit dans le standard UML et qui définit la structure interne des modèles UML et le *méta-modèle SPEM*, qui permet de définir des modèles de description de procédés appartiennent tous les deux à ce niveau. Il faut noter aussi que les profils UML, mécanisme d'extension du méta-modèle UML, font partie de ce niveau.
- **Le niveau M3** (ou méta-méta-modèle) est composé d'une unique entité qui s'appelle le MOF (Meta Object Facility) [OMG, 2011] (cf. section 3.2.3.1). Le MOF permet de décrire la structure des méta-modèles, d'étendre ou de modifier les méta-modèles existants. Le MOF est réflexif, il se décrit lui-même. Ce qui permet de dire que le niveau M3 est le dernier niveau de la hiérarchie.

Dans le cadre de cette thèse, on s'intéresse uniquement aux trois premiers niveaux : M0 pour décrire les systèmes qui s'exécutent, M1 pour spécifier les systèmes avec UML 2.0, et M2 pour spécifier le méta-modèle.

3.2.3 Les approches de l'OMG

L'OMG est un consortium regroupant des industriels, des fournisseurs d'outils, et des académiques dont l'objectif principal est de développer des standards pour normaliser les différentes approches d'un domaine mais aussi pour contrôler la prolifération des technologies. l'OMG définit plusieurs standards sur lesquels repose l'approche MDA. Nous pouvons citer : MOF [OMG, 2011],OCL [OCL], UML[UML, a], SPEM [SPEM, 2008], QVT [QVT, 2011],

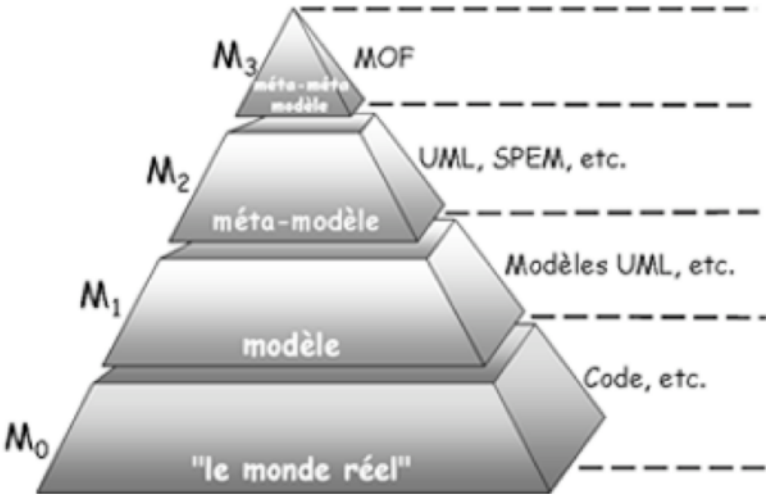


Figure 3.2 — La pyramide des niveaux de modélisation [Bézivin, 2003]

etc. La plupart des approches de mise en œuvre, aussi bien dans le monde industriel qu'académique, tentent de s'aligner sur ces standards.

La Figure 3.3 montre les couches de spécification du MDA. Dans le noyau se trouvent les standards de base (MOF, UML et CWM). La première couche représente les plates-formes supportées (Java, Corba, Web Service , ect.). La deuxième couche concerne les services système et enfin l'extérieur montre les domaines pour lesquels des composants métiers doivent être définis (Transport, Télécommunication, Finance, ...).

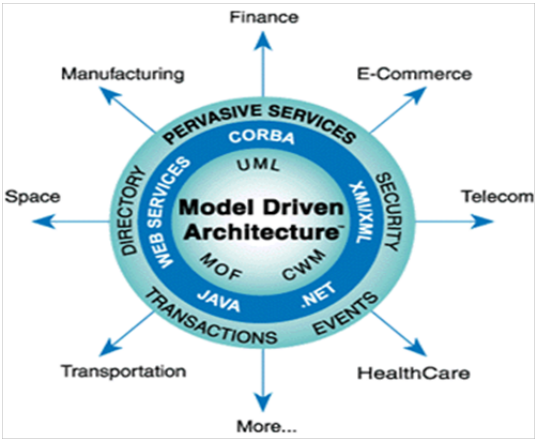


Figure 3.3 — Couches de spécifications du MDA [OMG, 2013]

3.2.3.1 MOF (Meta Object Facility)

Le MOF [OMG, 2011] est le langage standard de plus haut niveau d'abstraction dans une architecture à trois niveaux. Le standard MOF apporte le support de définition des formalismes de modélisation sous la forme de méta-modèles comme UML, CWM et XMI. Il définit les éléments essentiels, la syntaxe et la structure des méta-modèles utilisés pour construire des modèles. Dans ce travail, nous nous intéressons aux deux langages de niveau

2. Ces deux langages sont conformes au MOF. Le premier langage est UML 2.x qui permet de créer des modèles, éventuellement à travers une spécialisation, ou par extension, via les profils UML. Le deuxième langage est SPEM 2, qui permet permet de définir des modèles de description de procédés.

3.2.3.2 Unified Modeling Language (UML)

UML [UML, a] est un langage de modélisation graphique et semi-formel normalisé par l'OMG. UML, ne fournit pas une méthode en soi, mais plutôt une notation graphique, qui peut être utilisée dans plusieurs contextes pour décrire des systèmes logiciels. UML permet de décrire les aspects fonctionnels, structurels et comportementaux d'un système en définissant treize types de diagrammes.

Dans sa version 2.0, offre la possibilité définir la structure, les concepts et les éléments du modèle ainsi que des éléments de sémantique indépendamment de la plateforme technologique. Même si UML a été conçu pour pouvoir modéliser une grande variété de systèmes, il ne peut couvrir tous les domaines. C'est pour cela qu'ont été ajoutés des mécanismes d'extension permettant de personnaliser et d'étendre le langage UML à un domaine ou à un besoin particulier.

Le Profil UML

La notion de pro ?l UML a été introduite dans les premières versions de la spécification UML. L'OMG définit la notion de profil comme suit [UML, a] : «The Pro ?les package contains mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling). The pro ?les mechanism is consistent with the OMG Meta Object Facility (MOF) ».

Les profils UML permettent d'étendre le méta-modèle UML dans le cas où les développeurs ont du mal à exprimer des besoins spécifiques à leur domaine en se basant uniquement sur des éléments déjà existant dans UML. La notion de profil est non seulement un ensemble d'extensions qui permet à un utilisateur d'un domaine particulier (temps réel, BPM, etc.) d'entreprendre les activités liées à son domaine tout en utilisant UML, mai aussi, un ensemble d'extensions permettant d'adapter le méta-modèle UML à différentes plateformes techniques (J2EE, .Net, etc.).

Le profil contient des définitions de stéréotypes, des contraintes et des valeurs marquées sur les éléments du modèle. Les stéréotypes sont des annotations que l'on applique à des éléments de modélisation pour les spécialiser. Cependant, un stéréotype ne peut pas être instancié dans un modèle de l'utilisateur mais il peux être enrichie par un ensemble d'attributs (tagged values). Les valeurs marquées sont des paires "nom-valeur" qui ajoutent de l'information supplémentaire aux éléments stéréotypés. Ces informations sont ajoutées aux éléments lors de l'application d'un stéréotype à un élément.

3.2.3.3 Software System Process Engineering Metamodel (SPEM 2.0)

Une définition de ce standard, qui correspond à nos besoins est présentée dans [SPEM, 2008] : *The Software and Systems Process Engineering Meta-Model (SPEM) is a process engineering metamodel as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes.*

SPEM permet de décrire les processus de développement logiciel. Le but de SPEM est aussi de permettre la réutilisation des processus et de la documentation.

Le but du standard SPEM [SPEM, 2008] est de proposer des concepts partagés par la communauté du génie logiciel pour décrire les processus de développement logiciel. Le développement du méta-modèle SPEM a été motivé par :

- L'abondance de différents concepts de modélisation des processus décrits dans des formats différents, en utilisant différentes notations
- La violonté d'assurer la cohérence entre ces différentes approches.
  Ainsi le besoin de standardisation s'est posé et la norme SPEM a vu le jour.

La première version de la norme SPEM a été introduite par l'OMG en 2002. Cette première version était basé sur le standard UML 1.4. Des changements majeurs ont conduit à la version SPEM 2.0 [SPEM, 2008], qui est compatible avec UML 2. En raison de la conformité avec le standard UML, les diagrammes UML tels que les diagrammes d'activités ou diagrammes états peuvent être util pour visualiser des modèles du processus SPEM. SPEM 2.0 est décrit à la fois comme un mèta-modèle conforme à MOF et un profil UML. Comme profil UML, il bénéficie de l'outillage d'UML existant ainsi que des principaux diagrammes d'UML.

3.2.3.4 Architecture dirigée par les modèles (MDA)

L'architecture dirigée par les modèles ou MDA (Model Driven Architecture) [Soley, 2000] est une approche orientée modèle définie par l'Object Management Group (OMG) et rendue publique fin 2000. Cette approche spécifie trois niveaux d'abstraction (niveau métier, niveau indépendant de la plateforme et niveau dépendant de plateforme)pour la description d'une architecture d'un système et propose un processus de développement fondé sur ces trois niveaux et dirigé par les modèles.

L'idée de base du MDA est de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation sur une plate-forme cible donnée. Dans un processus de développement orienté MDA, tout est considéré comme modèle. Ainsi, MDA identifie quatre types de modèles : CIM, PIM, PDM et PSM.

- Le **CIM** (Computation Independent Model), appelé aussi modèle de domaine ou modèle métier, modélise les exigences du système. Son but est d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine particulier. MDA ne fait aucune préconisation quant au langage à utiliser pour décrire les CIM. Par exemple, dans UML les modèles des exigences sont décrit en utilisant les digrammes de cas d'utilisation ou encore avec l'utilisation de profils UML tels que SysML [SysML, 2012].

- Le **PIM** (Platform Independent Model) ou modèle d'analyse et de conception abstraite de l'application. Ce modèle décrit le système sans montrer les détails de son utilisation sur une plate-forme particulière. Dans MDA, il est possible d'élaborer plusieurs modèles PIM indépendants de la plate-forme cible. Les modèles PIM ne contiennent aucune information sur les plates-formes d'exécution. Un PIM doit être raffiné avec les détails d'une ou plusieurs architecture(s) particulière(s) pour produire un PSM.

- Le **PDM** (Platform Description Model) décrit la plate-forme sur laquelle le système va être exécuté (par exemple des modèles de composants à différents niveaux d'abstraction comme CCM ou EJB).

- Le **PSM** (Platform Specific Model) ou modèle spécifique des plates-formes d'exécution. PSM est le modèle produit par la transformation d'un PIM pour prendre en compte les informations techniques relatives à la plate-forme choisie. Le PSM peut être raffiné par transformations successives jusqu'à l'obtention d'un système exécutable.

La figure 3.4 donne une vue générale d'un processus MDA appelé couramment cycle de développement en Y en faisant apparaître les différents niveaux d'abstraction associés aux modèles.

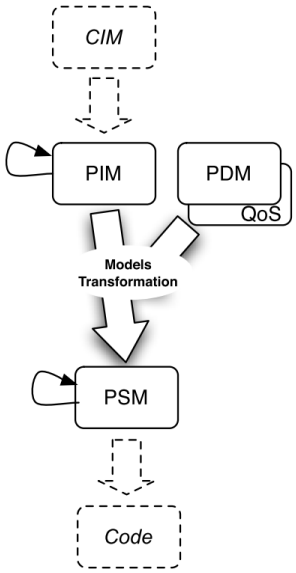


Figure 3.4 — Principes du processus MDA [Combemale, 2008]

3.2.4 Transformation de modèles

Les deux principaux artefacts de l'ingénierie des modèles sont la modélisation et les transformations de modèles. D'un point de vue général, on appelle une transformation de modèle tout processus dont les entrées et les sorties sont des modèles. La définition et l'automatisation des transformations a pour objectif de rendre les modèles plus opérationnels et d'augmenter la productivité du développement dans une approche IDM.

Cette section présente le principe général d'une transformation, les types de transformation de modèles ainsi que les principaux langages de transformation.

3.2.4.1 Principe général d'une transformation

Une transformation est la génération automatique d'un (de) modèle(s) cible(s) à partir d'un (de) modèle(s) source(s), suivant une définition de transformation. Une définition de transformation est un ensemble de règles de transformation qui décrivent comment un modèle décrit avec un langage source peut être transformé en un modèle décrit avec le langage cible [Bézivin, 2005]. Une règle de transformation décrit comment une ou plusieurs constructions du langage source peuvent être transformées en une ou plusieurs constructions du langage cible [Kleppe *et al.*, 2003a].

De plus, on peut modéliser la transformation elle-même en lui appliquant les principes de l'IDM, et donc de la considérer comme une application définie par un modèle de transformation (Mt) lui-même conforme à un métamodèle (MMt), ce dernier représentant la définition abstraite du langage de transformation [Bézivin, 2004b].

Comme illustré par la figure 3.5, une transformation de modèle définie par un modèle Mt est l'opération qui permet de transformer un modèle Ma (conforme à son méta-modèle MMa) en un modèle Mb (conforme à son méta-modèle MMb). D'une manière plus formelle, on peut définir cette opération par la fonction suivante :

Mb : MMb = Trans (Ma : MMa , Mt : MMt)

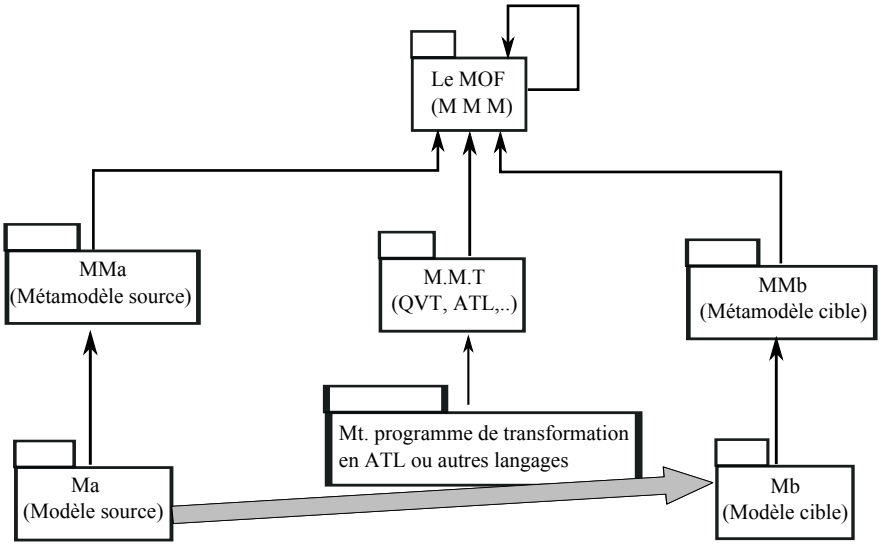


Figure 3.5 — Principe d'une transformation de modèles [Bézivin, 2004b]

3.2.4.2 Typologie des transformations

Les modèles sources et cibles sont décrits par des méta-modèles qui peuvent être identiques ou différents. Dans [Mens *et al.*, 2005], sont distinguées les transformations dites endogènes et exogènes combinées à des transformations dites verticales et horizontales.

La figure 3.6 donne des exemples illustrant l'orthogonalité de ces critères de classification. Une transformation est :

– **Endogène** : si les modèles source et cible sont conformes au même méta-modèle ;

- **Exogène** : si les modèles source et cible sont conformes à des méta-modèles différents ;
- **Horizontale** : si les méta-modèles résident au même niveau d'abstraction ;
- **Verticale** : si les méta-modèles résident à des niveaux d'abstraction différents.

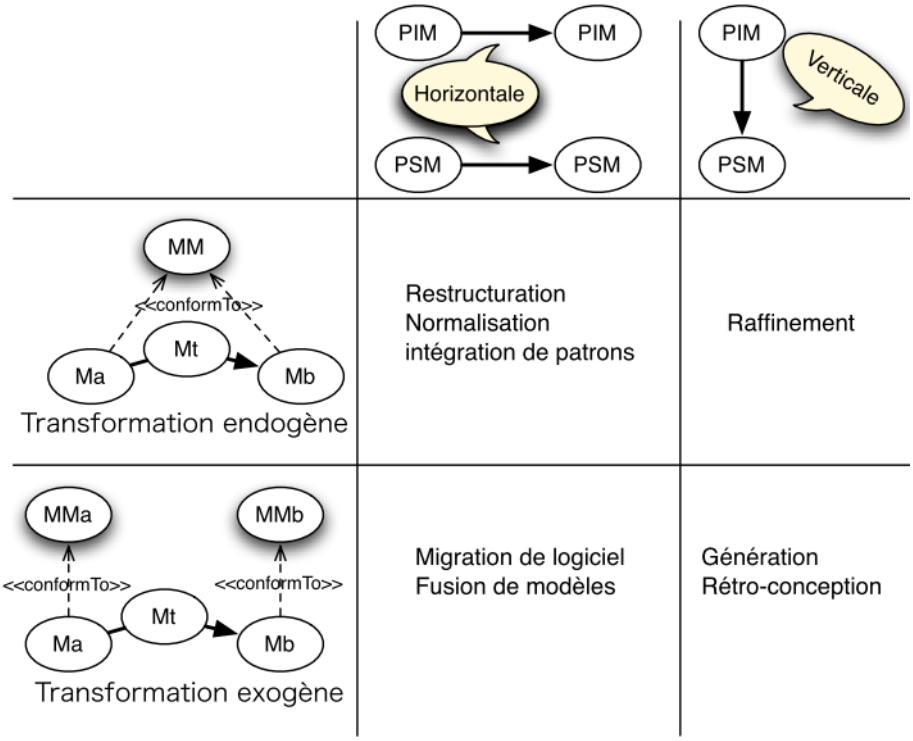


Figure 3.6 — Types de transformation et leurs principales utilisations [Combemale, 2008]

Dans la littérature [Mens *et al.*, 2005], [Czarnecki et Helsen, 2003] et aussi [Diaw, 2011] on trouve aussi une autre typologie des transformations caractérisée comme suit :

- **Une transformation simple** (1 vers 1) transforme un élément d'un modèle source en un élément d'un modèle cible. Un exemple typique est la transformation d'une classe UML en un document XML, qui définit sa structure, ou en une table de base de donnée relationnelle ;
- **Une transformation multiple** (M vers N) transforme un ou plusieurs éléments d'un modèle source en un ou plusieurs éléments d'un modèle cible. Comme exemples de transformations multiples nous pouvons citer les transformations de décomposition de modèles (1 vers N) et de fusion de modèles (N vers 1) ;
- **Une transformation de mise à jour** parfois appelée transformation sur place permet de modifier un modèle par ajout, modification ou suppression d'un de ses éléments. Dans ce type de transformation, le modèle source est mis à jour sans création explicite du modèle cible. La restructuration de modèles (Model Refactoring) qui permet de réorganiser les éléments d'un modèle afin d'améliorer sa structure ou sa lisibilité en est une illustration.

D'autre part, en se référant à la classification proposée dans [Czarnecki et Helsen, 2003] et présentée dans [Gilliers, 2005], basée sur la nature de l'artefact logiciel produit par la transformation et la technique utilisée par la mise en œuvre de celle-ci, deux grandes familles d'approches sont répertoriées : les approches de transformations «modèle vers modèle», et

les approches de transformations «modèle vers texte».

– **Modèle vers modèle**. La figure 3.7 présente les méthodes de transformation de type modèle vers modèle. Chaque modèle est exprimé sous la forme d'une instance de son méta-modèle à l'aide d'une structure de donnée appropriée. Les règles de transformation sont exprimées en fonction des entités et structures définies par ces derniers. Elles sont interprétées pour chaque modèle source par l'outil de transformation de modèle. Cette approche est caractérisée par le fait que le modèle source et le modèle produit sont exprimés de manière structurée en fonction de leur méta-modèle respectif. Le passage d'un modèle de type PIM à un PSM est l'exemple le plus classique de ce type de transformation.

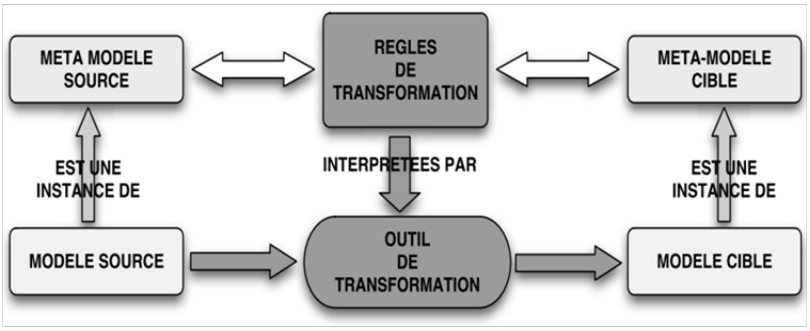


Figure 3.7 — Approche de transformation modèle vers modèle [Gilliers, 2005]

– **Modèle vers texte**. La figure 3.8 illustre l'utilisation de transformations de type modèle vers texte. Le modèle source est décrit en fonction de son méta-modèle, mais le modèle résultat est produit de manière non structurée sous forme de texte (code source, fichier XML, etc.). Les règles de transformation sont donc écrites en fonction du méta-modèle du modèle source, et de la syntaxe des fichiers produits. Le méta-modèle cible n'existe plus directement dans le processus de transformation. La génération de code peut être considérée comme un cas particulier de transformation de type modèle vers texte. Pour cela, plusieurs outils MDA open source et commerciaux basés sur des "templates" de transformation des modèles vers du code ont vu le jour. Dans cette catégorie on peut citer Acceleo [Acceleo] qui est intégré à la plate-forme Eclipse et au framework EMF ou encore AndroMDA [AndroMDA].

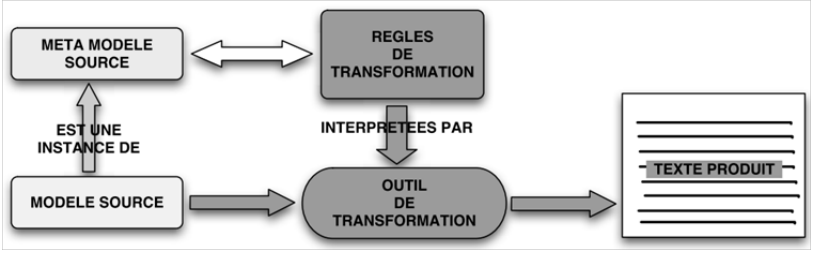


Figure 3.8 — Approche de transformation modèle vers texte[Gilliers, 2005]

3.2.4.3 Langages de transformations

La transformation de modèles est une opération centrale de l'Ingénierie Dirigée par les Modèles. C'est pour cette raison que plusieurs langages de transformation ont été définis. Généralement, les langages de transformation appliquent les techniques de l'IDM aux transformations elles-même. Le principe est d'offrir un méta-modèle permettant de construire des modèles de transformation.

Dans ce qui suit, nous présentons les langages QVT de l'OMG [QVT, 2011] et ATL du groupe ATLAS de l'INRIA-LINA [Jouault et Kurtev, 2006a] qui sont très utilisés dans la communauté de l'IDM et que nous jugeons représentatifs de cette catégorie de langages.

QVT Dans le cadre de l'approche MDA, l'OMG a introduit le langage QVT [QVT, 2011] (Querry, View, Transformation). *Query* est une requête qui prend en entrée un modèle et sélectionne des éléments spécifiques de ce modèle. *View* est un modèle qui dérive d'autres modèles. *Transformation* prend un modèle en entrée pour le modifier ou en créer un autre.

QVT est une spécification couplée à celle du MOF et comme son nom l'indique n'est plus dédiée à la spécification de modèles, mais à leur manipulation. Ce langage possède une nature hybride déclarative/impérative et implémente la transformation de modèle de différentes manières. Dans le cadre de MDA, ce langage est associé à ses outils tels que MOF/QVT. Ce langage est donc un nouveau paradigme qui offre des possibilités nouvelles en matière d'ingénierie logicielle.

Le standard QVT présente un caractère hybride en supportant trois langages de transformation (Figure IV-7). La partie déclarative de QVT est définie par deux langages de niveaux d'abstraction différents : *QVT-Relations* et *QVT-Core*.

Le langage *QVT-Relations* est un langage orienté utilisateur permettant de définir des transformations à un niveau d'abstraction élevé. Il a une syntaxe textuelle et graphique.

Le langage *QVT-Core* est un langage technique de bas niveau, défini par une syntaxe textuelle. Ce langage sert à spécifier la sémantique du langage QVT-Relations, donnée sous la forme d'une transformation *RelationsToCore* (Figure IV-7).

La composante impérative de QVT est supportée par le langage *Operational Mappings*. Ce langage étend les deux langages déclaratifs de QVT en ajoutant des constructions impératives (séquence, sélection, répétition, etc.) ainsi que des constructions OCL. Enfin, QVT propose un deuxième mécanisme d'extension nommé Boîte noire (*Black Box*) pour spécifier des transformations ; il s'agit d'invoquer des fonctionnalités de transformations implémentées dans un langage externe.

L'avantage de QVT est qu'il se base sur les standards existants à savoir le MOF et OCL. Cela permet de développer des outils simples pour un grand nombre d'utilisateurs ainsi que compatibles avec un grand nombre d'outils de modélisation déjà existants.

ATL Dans le but de proposer des outils se réclamant de l'architecture MDA publié en 2002 par l'OMG, le laboratoire de l'INRIA de Nantes a développé le langage de transformation de modèles ATL [Jouault et Kurtev, 2006a] (ATLAS Transformation Language) que nous

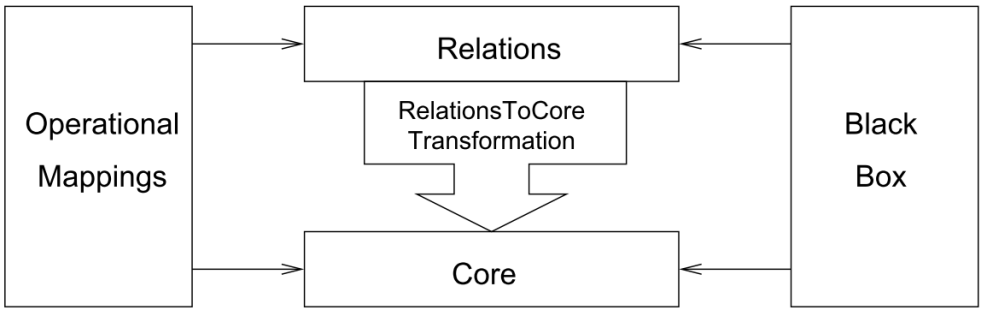


Figure 3.9 — Architecture du standard QVT [QVT, 2011]

utilisons dans la suite de cette thèse. Un programme ATL est composé de règles qui définissent comment les éléments du modèle source sont identifiés et parcourus pour créer les éléments du modèle cible (transformation de modèle à modèle (appelée Module)). Au delà des transformations de modèles classiques, ATL définit un modèle de requête supplémentaire qui permet de spécifier des requêtes sur les modèles (transformations de type modèle vers texte (appelée Query)).

Une des particularité du langage ATL, est son caractère hybride (déclaratif et impératif). La partie déclarative permet de faire correspondre directement un élément du méta-modèle source de la transformation avec un élément du méta-modèle cible de la transformation. L'exemple d'une transformation complètement déclarative en ATL est présenté ci-dessous.

Nous décrivons une règle nommé Sample, pour chaque élément InputM de type InputElement qu'elle identifie dans le modèle source, crée dans un modèle cible un élément OutputM de type OutputElement, et initialise la valeurs de l' attributs attributeA de OutputM avec la valeur de l' attributs attributeB de InputM.

```
1  module inModel2outModel;\\
2  create Out : outMetaModel from In : inMetaModel ;\\
3
4  rule Sample \\
5  \{
6   \hspace{1cm}from \\
7   \hspace{1.6cm} InputM: InputMetaModel!InputElement \\
8  \hspace{1cm} to \\
9  \hspace{1.6cm} OutputM : OutputMetaModel!OutputElement( \\
10   \hspace{1cm}attributeA <- InputM.attributeB, ) \\
11 \}
```

En ATL une transformation s'appelle module. Le mot-clé OUT montre le méta-modèle cible (ligne 2). Le mot clé IN montre le méta-modèle source (ligne 2). La règle (Sample) du fichier de transformation (inModel2outModel) est déclarative. et elle est définie comme suit :

- Sample est le nom de la règle de transformation.
- InputM est le nom de la variable qui dans le corps de la règle va représenter l'élément source identifié.
- OnputM est le nom de la variable qui dans le corps de la règle va représenter l'élément cible créé.

- InputMetaModel (resp. OutputMetaModel) est le méta-modèle auquel le modèle source (resp. le modèle cible) de la transformation est conforme.
- InputElement désigne la méta-classe des éléments du modèle source auxquels cette règle va s'appliquer.
- OutputElement désigne la méta-classe à partir de laquelle la règle va instancier les éléments cibles. Le point d'exclamation permet de spécifier à quel méta-modèle appartient une méta-classe en cas d'homonymie.
- attributeA est un attributs de la méta-classe OutputElement sa valeur est initialisée à l'aide de la valeur de l' attributs InputM.
- attributeB, de la méta-classe InputElement. l'utilisation du « . », issu de la spécification OCL et reprit par ATL, de "naviguer" dans les modèles.

La partie impérative d'ATL complète ces correspondances directes entre éléments. Elle comporte des déclarations conditionnelles (if, then, else...endif), des déclarations de variables (let VarName : varType = initialValue), des déclarations de boucle (while (condition) ?do), etc. Cette partie permet également de manipuler les éléments générés par les règles déclaratives (modification d'attributs, etc.).

Au niveau implémentation, un moteur d'exécution pour le langage ATL a été développé et est disponible à travers le projet M2M d'Eclipse. Ce moteur d'exécution est basé sur une machine virtuelle de transformation [Jouault et Kurtev, 2006a].

3.3 Programmation Orientée Aspects

La séparation et la représentation explicite et modulaire ainsi que l'adaptation des préoccupations transversales comme la sécurité, la synchronisation, la persistance ou la composition sont aujourd'hui au cœur de plusieurs travaux de recherche. Ces travaux ont comme objectifs de proposer des modèles permettant une meilleure séparation et ainsi qu'une composition de tout type de préoccupations. Ainsi de nouveaux modèles et langages de programmation associés sont actuellement proposés et présentés à la communauté. Nous nous intéressant dans ce cadre à la "programmation orientée aspect" (POA) qui est l'une des principales approches émergente à l'heure actuelle [Kiczales *et al.*, 1997] initiée par un groupe de recherche de Xerox Parc dirigé par Gregor Kiczales.

Ce paradigme repose sur une décomposition des programmes non seulement en unités modulaires représentant les préoccupations fonctionnelles de base, mais aussi en unités modulaires dédiées à la représentation des préoccupations transversales. Ils offrent de plus des solutions adéquates de composition de préoccupations (on parle aussi de tissage), afin de construire des systèmes efficaces.

Dans cette section, nous motivons et présentons tout d'abord le paradigme de la programmation par aspects. Puis nous présentons les raisons qui nous ont poussés à l'utilisation de cette approche dans le cadre de nos travaux.

3.3.1 Motivation

Dans les approches de développement traditionnelles (procédurale, fonctionnelle, objet ou composant) la structuration d'une application repose sur sa décomposition en unités fonctionnelles. Différents paradigmes et langages de programmation appropriés supportent l'implantation et la décomposition des systèmes en composants paramétrables, qui peuvent être appelés pour réaliser une fonction donnée. Or, il s'avère que beaucoup de systèmes ont besoin de fonctionnalités ou de propriétés systémiques qui ne correspondent pas forcément à ces modes de décomposition.

Par exemple, la sécurité, la synchronisation et la re-configuration sont toutes des propriétés transversales (appelées aussi propriétés non fonctionnelles) qui touchent à l'ensemble des composants fonctionnels de l'application. Ces propriétés peuvent être conçues et analysées séparément des fonctions de base de l'application.

Toutefois, leur implantation grâce à un langage à objets, par exemple, provoquerait une dispersion de leurs codes dans l'ensemble des composants fonctionnels constituant le système (voir figure 3.10).

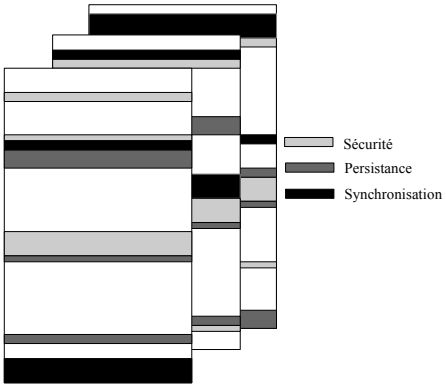


Figure 3.10 — Entrelacement des propriétés transversales d'une application dans l'approche Objet

Le code source de l'application devient alors difficilement lisible et instable car les instructions permettant la mise en œuvre de ces propriétés transversales est mêlé dans l'implantation des différentes classes affectées par ces propriétés, nuisant ainsi à la gestion de l'évolution et à la réutilisation du code de l'application.

3.3.2 Principes de la programmation par aspects

Afin de pallier les problèmes évoqués précédemment, la programmation par aspects propose de découpler les propriétés non fonctionnelles de la structure des modules de décomposition d'un système (composant, par exemple).

Dans le contexte où la partie fonctionnelle de l'application est fondée sur les composants, la programmation par aspects consiste à réaliser une application en deux temps. Une première étape consiste à décomposer l'application en composants et aspects. Les différents as-

pects et composants sont définis séparément les uns des autres. Leurs implantations doivent être découplées, de sorte qu'elles ne se référencent pas les unes les autres. Ensuite, l'application est produite par la recomposition, également désigné sous le nom de tissage, de l'ensemble des aspects et des composants ainsi définis (voir figure 3.11). Dans cette seconde étape, il faut spécifier les règles de recomposition.

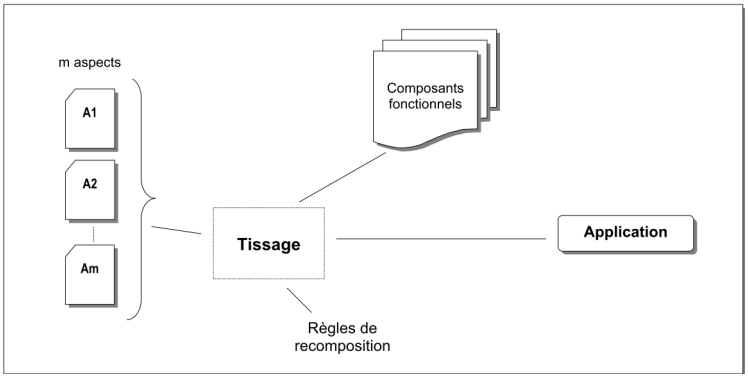


Figure 3.11 — Décomposition d'une application dans l'AOP [Hachani, 2002]

Cette recomposition est assuré par un mécanisme de composition : le tissage d'aspects (appelé en anglais, Aspect Weaving). Dans [Kiczales *et al.*, 1997], le tisseur d'aspects est représenté comme un outil qui reçoit en entrée les codes sources des composants et des aspects, et qui émet en sortie un code tissé représentant le code source de l'application (voir figure 3.12).

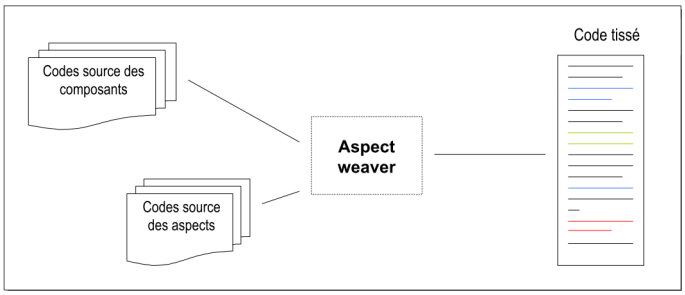


Figure 3.12 — Construction du code exécutable dans l'AOP [Hachani, 2002]

En pratique, la programmation orientée aspect définit les notions suivantes [Kiczales *et al.*, 1997] :

- **Les points de jonction ("joinpoint")**. Les points de jonction représentent des points, bien définis, dans le flot d'exécution des composants fonctionnels de l'application. Ce sont des éléments fondamentaux dans toute implantation par aspects. Ils sont relatifs à l'ensemble des points où les aspects interagissent avec les composants. Par exemple, l'invocation d'une méthode, l'accès à un attribut, l'accès à une instance d'une certaine classe.
- **Les coupes ("pointcut")**. Une coupe est un regroupement de points de jonctions. Un pointcut permet la définition des points de jonction à utiliser pour composer chaque aspect avec les composants.
- **Les codes advices ("advice")**. Les advices définissent des morceaux d'implantation,

attachés à un pointcut particulier. C'est le code qui définit le comportement de l'aspect. C'est le code que l'on veut intégrer à l'application. Mais il reste aussi à définir comment le code aspect va modifier le comportement du code applicatif. Ainsi, on précise si le code de l'aspect doit s'exécuter avant, après ou autour d'un point de jonction.

En conclusion, la programmation par aspects permet d'implémenter des préoccupations qui s'entrelacent avec le reste du système d'une manière modulaire. Ce paradigme permet de bénéficier des avantages de la modularité qui garantie un code plus simple, plus facile à développer et à maintenir et qui a un meilleur potentiel de réutilisation.

3.4 Conclusion

Dans ce chapitre, nous avons mis l'accent sur les concepts de base, les standards ainsi que les langages associés à l'ingénierie dirigée par les modèles. Le développement logiciel dirigé par les modèles a pour principal objectif de concevoir des applications en séparant les préoccupations et en plaçant les notions de modèles, méta-modèles et transformations de modèles au centre du processus de développement. L'IDM permet aussi de travailler à un niveau d'abstraction supérieur grâce au raisonnement autour des modèles. C'est pourquoi nous avons choisi d'utiliser cette approche et les technologie développés autour. Aussi dans ce même chapitre, nous avons montré que la complexité du code source d'un système résulte, principalement, de l'enchevêtrement et de la dispersion du code des propriétés non fonctionnelles d'une application, dans la description de ses composants. Une solution à ce problème consiste à séparer et à découpler la définition de ces propriétés, respectant ainsi le principe de séparation des préoccupations.

C'est sur ce principe que la programmation orientée aspect été proposée. Elle constitue un outil technique efficace pour la séparation des préoccupations. Différentes implantations existantes. Elles ont toutes des avantages et des inconvénients. Chacune répond à un besoin spécifique.

Dans notre contexte, nous allons définir des aspect relatives aux patrons de sécurité appliqués dans le modèle à composants. Cela garantit la séparation entre la partie fonctionnelle des composants et les aspects liés à la sécurité proposés par les patrons de sécurité.

Parmi les technologies et langages présentés dans ce chapitre nous avons choisi dans le cadre de cette thèse d'utiliser :

- **UML 2.0** pour modéliser les composants car UML s'est imposé comme la référence en modélisation et génie logiciel. Ce standard est très utilisé dans l'industrie du logiciel et de nombreux outils permettant de créer, éditer et manipuler des diagrammes UML existent actuellement sur le marché en open source et gratuit.
- **Les profils UML** comme mécanisme d'extension du méta-modèle UML. Ce mécanisme permet d'étendre UML pour supporter les concepts de sécurité.
- **ATL** comme langage de transformation de modèles. Nous fixons deux critères liés aux transformations de modèles que nous souhaitons établir dans le cadre de cette thèse : la simplicité d'établissement des règles et la possibilité de gérer des modèles et d'appliquer des profils UML. ATL semble répondre à ces deux critères. En plus d'être un outil libre, les règles ATL sont simples à mettre en œuvre. Son langage à base de

variables et d'affectation le rend accessible. De plus, ATL est considéré maintenant comme un standard de transformation dans Eclipse et est intégré depuis 2007 dans le projet UML2 Tools (un environnement Eclipse pour la gestion de modèles UML).

– **AspectJ** comme une implémentation du paradigme aspect. Pour le développement du prototype exposé au chapitre 6, nous avons utilisé AspectJ car il est totalement indépendant de l'environnement d'exécution. Il ne manipule pas la machine virtuelle Java ce qui le rend compatible de fait avec toutes les plates-formes composants.

Comme nous l'avons rapidement indiqué, nous nous intéressons dans cette thèse l'intégration de la sécurité dans les applications à base de composants à partir des patrons de sécurité. Nous présentons dans le chapitre suivant différent travaux qui ont étudié cette intégration.

4 Travaux connexes : Vers le développement de systèmes sécurisés

4.1 Introduction

Dans ce chapitre nous présentons dans un premier temps les principaux travaux qui ont essayé de prendre en considération la sécurité lors du processus de développement d'une application. Dans un second temps nous présentons les travaux qui proposent l'utilisation de patrons de sécurité. Nous concluons ce chapitre par une discussion générale qui mettra en évidence les limites des approches présentées.

4.2 L'ingénierie de la sécurité

Dans cette section, nous présentons les principaux travaux qui intègrent les aspects relatifs à la sécurité lors d'un processus de développement logiciel. Nous nous intéressons particulièrement aux travaux qui présentent une approche IDM. Le génie logiciel touche la spécification, la conception, le développement, l'implémentation, le test et l'évolution des systèmes logiciels. La sécurité dirigée par les modèles (Model-Driven Security (MDS)) [Basin *et al.*, 2006] est l'application des techniques du génie logiciel lors de la conception de systèmes sécurisés.

La sécurité dirigée par les modèles vise à concevoir, dans un premier temps, des systèmes sécurisés indépendamment de la plateforme utilisée [Basin *et al.*, 2006] et [Hafner et Breu, 2008]. La recherche dans le domaine des MDS est un sujet très actif qui vise la conception de systèmes sécurisé. Dans ce qui suit nous présentons d'une façon non exhaustive les principaux travaux de recherche qui proposent des solutions pour le développement d'applications sécurisées indépendamment de l'utilisation des patrons.

En 2002, David Bassin et al. [Lodderstedt *et al.*, 2002] introduisent pour la première fois le terme de «sécurité dirigée par les modèles » (Model-driven Security). Comme première application, ces mêmes auteurs ont proposé la génération des politiques de contrôle d'accès à partir des contraintes d'autorisation abstraites écrites en OCL (Object Constraint Langage). L'approche proposée est intéressante et flexible. En effet le DSL (Domain Specific Language)

proposé pour introduire la sécurité est appelé SecureUML. L'approche présentée dans SecureUML consiste à modéliser les politiques de contrôle d'accès pour montrer comment celles-ci peuvent être intégrées dans un processus de développement logiciel dirigé par les modèles. La solution proposée par SecureUML se base sur un modèle élargi de contrôle d'accès basé sur les rôles en utilisant le modèle RBAC (Role-Based Access Control) pour la spécification et l'application de la sécurité. SecureUML définit un vocabulaire pour annoter des modèles UML avec des informations pertinentes relatives aux contrôles d'accès. Dans cette approche, les auteurs proposent un processus dirigé par les modèles composé de deux phases qui sont (1) la définition des politiques de contrôle d'accès (en utilisant le contrôle d'accès basé sur les rôles (RBAC)) et (2) la transformation vers des configurations de descripteur de déploiement J2EE.

Le méta-modèle SecureUML est représenté par la figure 4.1. SecureUML est défini comme une extension du méta-modèle UML. Les concepts de RBAC (Role Based Access Control) sont représentés directement en tant que artefacts de méta-modèle. Les auteurs ont introduit neuf concepts dans le méta-modèle. Parmi ces concepts on trouve l'utilisateur, le rôle et l'autorisation ainsi que les relations entre eux. SecureUML est également représenté sous forme de profil UML avec les mécanismes d'extension en utilisant les stéréotypes.

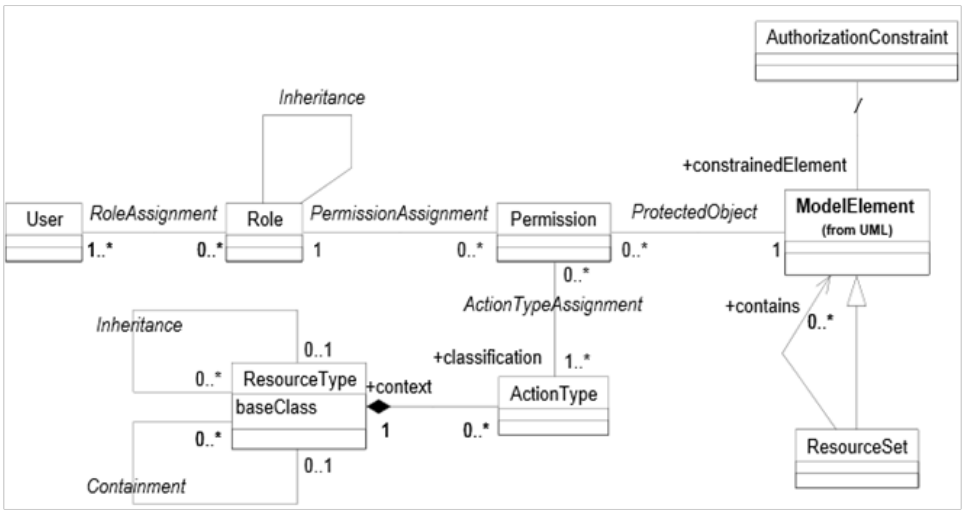


Figure 4.1 — Représentation du méta-modèle de secureUML comme décrit par Bassin et al [Lodderstedt *et al.*, 2002]. Il s'agit d'une extension du méta-modèle UML avec les concepts de RBAC

De leurs côté, Christian Wolter et al. [Wolter *et al.*, 2008], [Wolter *et al.*, 2009] définissent des politiques, de haut niveau, pour assurer des objectifs de sécurité comme l'authentification et la confidentialité dans les architectures orientées services. Les objectifs de sécurité ont été représentés avec des notations graphiques dans des modèles de processus métier dans le but de transformer les exigences de sécurité en politiques de sécurité concrètes. Comme résultats de ce premier travail, les auteurs ont proposé le modèle nommé "Security Policy Model"(figure 4.2). Ce modèle est capable d'assurer différentes exigences de sécurité comme l'authentification ou le contrôle d'accès.

Comme l'indique la figure 4.2, chaque objectif de sécurité est décrit par une contrainte

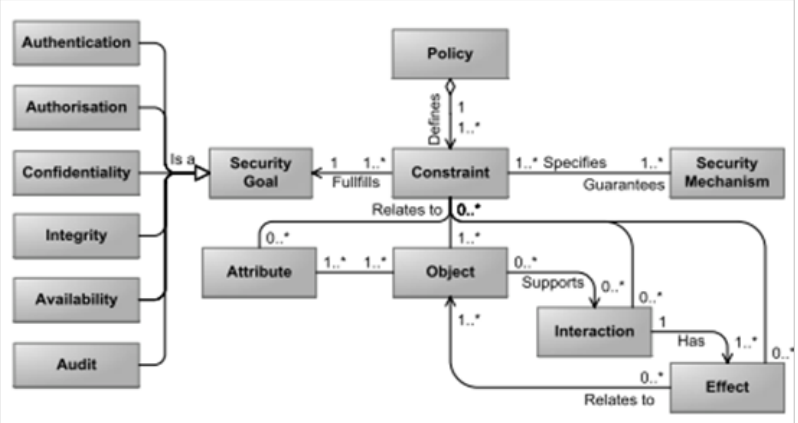


Figure 4.2 — Représentation du méta-modèle de secureUML comme décrit par Bassin et al [Lodderstedt *et al.*, 2002]. Il s'agit d'une extension du méta-modèle UML avec les concepts de RBAC

liée aux entités concernées. L'entité de base dans un modèle de politique de sécurité est un objet. Un objet est défini comme une entité capable de participer à une interaction avec d'autres objets. Cette interaction peut impliquer un ensemble d'informations qui sont échangée set elle aura toujours un effet, qui peut être composé à partir du changement d'état d'un objet ou d'informations dans un système. L'effet peut, mais ne doit pas nécessairement, être lié à l'objet qui a initié l'interaction. Par exemple, un objet pourrait être une application et un autre objet pourrait être un service pour stocker les données. Le processus d'accès à ce service serait l'interaction résultante dans laquelle les données sont stockées ou une information est renvoyée à l'application. Après avoir défini ce modèle de politique de sécurité ainsi que des modèles de contrainte de sécurité, comme le modèle de contraintes d'autorisation, les auteurs génèrent des politiques avec le standard "eXtensible Access Control Markup Language" (XACML). Ils produisent aussi une configuration de sécurité pour les services Web "Apache-axe2". Le modèle de politique de sécurité met en correspondance les besoins de sécurité avec les modèles de contraintes de sécurité.

Dans plusieurs de leurs travaux, Fumiko Satoh et al. [Satoh *et al.*, 2006],[Satoh *et al.*, 2008b], [Satoh *et al.*, 2008a], [Satoh et Yamaguchi, 2007] proposent des solutions génériques pour le développement d'applications sécurisées en se basant sur la sécurité dirigée par les modèles. Dans [Satoh *et al.*, 2006], ils introduisent des annotations de sécurité pour définir les exigences de sécurité dans les modèles du système logiciel. La politique de sécurité pour l'authentification est générée en respectant la norme "WS-Security Policy" en utilisant un modèle d'infrastructure de sécurité appelé " Security Infrastructure Model "(SIM). Ce modèle contient les mécanismes de sécurité pris en charge par la plate-forme cible. Pour gérer la transformation de modèles, ils utilisent des templates prédéfinis pour atteindre les objectifs de sécurités requis. En utilisant ces templates, ils sont en mesure de générer des politiques de sécurité exécutables pour chaque exigence de sécurité.

Les mêmes auteurs proposent une approche visant la génération des configurations de sécurité pour IBM WAS [Satoh et Yamaguchi, 2007]. Ce travail touche la mise en correspondance directe entre "IBM-WAS Deployment Descriptor (DD)" et WS-Security Policy. Les

auteurs proposent l'utilisation d'un modèle de mise en correspondance ou de transformation intermédiaire. Ce modèle permet à une politique de sécurité d'être transformée en une variété de configurations, y compris IBM WAS DD. Cette approche a été reprise dans [Satoh *et al.*, 2008b] et [Satoh *et al.*, 2008a], dans lesquels les mêmes auteurs ont pris pour cible la composition sécurisée du Service Component Architecture Composites (SCA) [OSOA] en utilisant SCA framework.

Parmi les travaux fondamentaux qui touchent le domaine de la sécurité dirigée par les modèles, on trouve aussi ceux de Jan Juerjens et. al. [Juerjens, 2002], [Juerjens, 2003], [Best *et al.*, 2007]. Ces auteurs proposent une extension du standard UML pour la validation des protocoles de sécurité appelés UMLsec [Juerjens, 2002]. La figure 4.3 présente une description détaillée des différents stéréotypes proposés. Étant donné qu'UMLsec est un profil UML, il définit la sécurité en utilisant des valeurs marquées et des contraintes associées aux éléments du méta-modèle. Un certain nombre de risques (delete, read, insert, access pour les liens) et de types d'éléments (un lien de type Internet supporte les risques delete, read, insert ; un lien de type encrypted ne supporte que le risque delete) sont prédéfinis. Il définit aussi des contraintes sur les éléments exprimant les règles de fonctionnement et la politique de sécurité, qui permettent l'évaluation des aspects sécurité d'une conception du système.

Stereotype	Base Class	Tags	Constraints	Description
Internet	link			Internet connection
encrypted	link			encrypted connection
LAN	link			LAN connection
secure links	subsystem		dependency security matched by links	enforces secure
secrecy	dependency			communication links assumes secrecy
secrecy dependency	subsystem		"xall", "send" respects data security	structural intercation data security
critical	object	secret		critical object
no down-flow	subsystem		provents down-flow provides secrecy	information flow
data security	subsystem			basic datasec requirements
fair exchange	package	start, stop	after start eventually reach stop	enforce fair exchange

Figure 4.3 — Un extrait du profil UMLsec proposé par Jan Juerjens et.al [Juerjens, 2002].

L'idée dans UMLsec est que les protocoles ou les mécanismes de sécurité ne doivent pas être insérés à l'aveuglette dans les systèmes, mais les aspects de la sécurité devraient être pris en compte dans le processus global de développement du système. Cela implique l'obligation de considérer la sécurité depuis la définition des exigences jusqu'à la mise en ?uvre (déploiement). Les auteurs prennent en compte les différents types de modèles UML à différents niveaux d'abstraction pour capturer les aspects de sécurité. L'idée est qu'un développeur d'applications, qui n'est pas un expert en sécurité pourrait utiliser ces modèles comme guide pour connaître les risques de sécurité et les vulnérabilités à différents stades de déve-

loppement de son système. Des exemples de diagrammes UML ont été illustrés dans leurs travaux comme le diagramme de classes UML (définition des niveaux de sécurité pour les attributs et les fonctions), le diagramme d'état transition (sécurité des flux d'information), le diagramme de séquence (sécurité des messages pour échanger des données) et le diagramme de déploiement (étiquettes de sécurité sur les composants de système physique). Cependant, les auteurs ne tiennent pas compte du raffinement successif des modèles ni de la génération du code.

De leur côté Michael Hafner et Ruth Breu [Hafner, 2006], [Hafner *et al.*, 2006] ont proposé un framework appelé SECTET pour l'ingénierie de la sécurité dirigée par les modèles. Afin d'utiliser ce Framework, ils ont proposé un langage de définition de politiques appelé SECTET-PL. Ce langage permet la définition des exigences de sécurité dans les modèles fonctionnels. Les politiques de contrôle d'accès abstraites sont décrites en SECTET-PL, et sont ensuite transformées en code XACML [M. Alam et Unterthiner, 2007], [Alam *et al.*, 2009]. Dans cette approche, semblable à l'approche de David Basin [Basin *et al.*, 2006], [Lodderstedt *et al.*, 2002], la politique abstraite est directement convertie en code, ce qui signifie que l'expert du domaine doit avoir une connaissance suffisante de la sécurité pour utiliser ce framework, dès les premières phases du développement du système, dans le modèle fonctionnel. Les principales contributions des travaux autour de SECTET sont : un langage spécifique au domaine appelé SECTET-DSL (utilisé pour modéliser des workflows inter-organisationnels), et le langage de définition de politique SECTET-PL [Hafner *et al.*, 2005a], [Hafner *et al.*, 2005b].

Pour Ulrich Lang et al. [Reznik *et al.*, 2007], [Lang et Schreiner, 2002], l'expert en sécurité doit définir la politique de sécurité de haut niveau et générer le code correspondant. Dans cet article, ces auteurs proposent un middleware spécifique appelé SecureMiddleware. Ce middleware étend le modèle de composant CORBA avec différentes propriétés non fonctionnelles comme la sécurité ou la reconfiguration. Les politiques de haut niveau sont spécifiées dans un langage de définition de la politique (Policy Definition Language (PDL)). L'approche par composants utilise des QoSEnablers, qui sont responsables d'assurer un type spécifique de Qualité de Services(QoS) tels que la sécurité. Les QoSEnabler renforcent les politiques de contrôle d'accès dans le middleware. Dans cette approche, la transformation ainsi que la mise à jour est effectuée par un OpenPMF (Policy Management Framework).Pour simplifier le développement d'applications à base de composants CORBA, un sous-ensemble d'UML appelé eUML et un profil de politique de sécurité appelé eUML-Sec ont été proposés. Le méta-modèle eUML contient deux packages principaux, Générique et Présentation, comme le montre la figure ci-dessous (figure 4.4). Le premier paquetage couvre la partie de modélisation générique (basé sur le méta-modèle UML 2.0) et le second comprend des concepts supplémentaires, qui définissent des informations de modélisation graphique d'un élément UML ainsi que le diagramme UML qui en résulte.

Cependant, cette approche s'adresse uniquement aux applications à base de composant CORBA. La généricité de l'approche proposé sur d'autres modèles de composants n'est pas explicitée. Le profil généré est également limité pour le modèle à composant CORBA. Ces différentes restrictions limitent la portée de cette démarche.

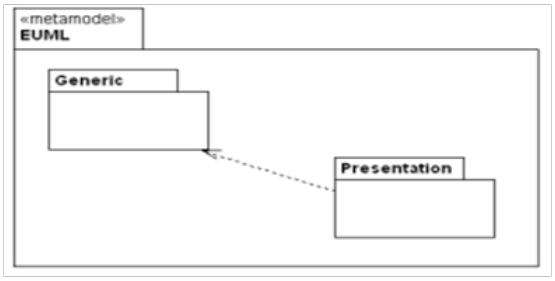


Figure 4.4 — Aperçu du métamodèle eUML proposé par Ulrich Lang et al. [Reznik *et al.*, 2007].

4.3 Intégration de la sécurité via les patrons

S'attaquer au développement de systèmes logiciels sécurisés reste un domaine de recherche actif dans l'ingénierie de la sécurité. Plusieurs méthodes ont été proposées pour aider les développeurs à prendre en considération les aspects relatifs à la sécurité dans le développement logiciels. Ces différentes méthodes peuvent être classées selon leurs niveaux de granularité, leurs flexibilités, les niveaux d'expertises requis pour les appliquer, leur coûts de déploiement, ou encore selon ce qu'ils soient formels ou informels [Georg *et al.*, 2002].

Comme décrit dans le chapitre 2, les patrons sont des solutions de sécurité bien pensées proposés par des experts en sécurité. Les développeurs d'applications peuvent utiliser les patrons de sécurité, sans avoir l'expertise en matière de sécurité et peuvent bénéficier de l'expérience d'experts en sécurité encapsulée dans ces patrons [Vivas *et al.*, 2003].

Comme présenté aussi dans la section précédente, l'intégration des aspects relatifs à la sécurité dans un processus de développement basé sur les modèles est parmi les préoccupations de plusieurs équipes de recherche et depuis pas mal de temps. Dans cette section nous nous intéressons en particulier aux travaux, peu nombreux, qui utilisent les patrons de sécurité comme mécanisme de cette intégration.

Dans le domaine de la sécurité logicielle, un certain nombre de patrons de sécurité et de catalogues de patrons liés à la sécurité existent (voir chapitre 2). Cependant, ces derniers fournissent peu voire pas d'informations sur la manière de les utiliser et le moment de leurs application dans le processus de développement logiciel/système.

Fernandez et al. [Fer, 2006], [E. B. Fernandez, 2007], proposent une méthodologie permettant l'intégration des patrons de sécurité pendant toutes les phases de développement d'un système. En effet, les auteurs proposent des recommandations permettant l'intégration de la sécurité dès l'étape de la définition des exigences. Ces directives touchent aussi la phase d'analyse, celle de la conception, l'implémentation, le test et finalement le déploiement du système. Cependant ce travail présentes des recommandations générales à suivre sans donner des détails de modélisation ou des solution concrètes.

D'autres approches existent permettant l'intégration de la sécurité à partir des patrons en utilisant le paradigme de la programmation orientée aspects. Il s'agit notamment des méthodologies de conception de sécurité orientée aspects, dans lesquelles les aspects de sé-

curité sont modélisés en utilisant des patrons pour être ensuite tissés dans les modèles fonctionnels [Georg *et al.*, 2002], [Ray *et al.*, 2004]. Cette méthodologie est assez intéressante car elle préserve une séparation entre les préoccupations fonctionnelles et non-fonctionnelles du système. Cependant, notre travail se rapporte à eux à l'égard de l'utilisation du développement orienté aspect pour la modélisation de la sécurité à partir des patrons.

Dans [Jurjens *et al.*, 2002] les auteurs expliquent comment utiliser les patrons de sécurité dans le cadre du développement des systèmes avec contraintes de sécurité. L'approche proposée intègre deux notions complémentaires. La première notion est le profil UMLsec, une extension d'UML pour le développement des systèmes avec contraintes de sécurité proposée par le même auteur. La deuxième notion est un outil industriel puissant, AUTO-FOCUS, utilisé pour analyser la spécification, pour générer du code ainsi que des séquences de test. L'approche présentée permet d'introduire des patrons de sécurité d'une manière méthodologique et formelle. Dans ce travail l'auteur affirme que les notions utilisées dans cette approche sont intuitives et selon lui, ce travail est une des étapes importantes vers une manière plus méthodologique de l'élaboration de systèmes sécurisés.

Gutiérrez et al. [Gutiérrez *et al.*, 2005] présentent un processus pour introduire la sécurité dans les services Web. Pour répondre aux exigences des systèmes en termes de sécurité, ce processus permet de produire une architecture de sécurité basée sur les services Web comprenant des mécanismes et des normes de sécurité. Les étapes principales de ce processus consistent à identifier les patrons de sécurités appropriés et à dériver des services de sécurité concrète à partir de ces patrons. Cependant la manière de sélectionner et d'intégrer ces patrons dans les services web reste non décrite et pas claire.

Dans les applications destinées à communiquer sur différents réseaux et afin d'assurer la sécurité dans les différentes communications à établir, G. Wimmel et A. Wisspeintner [Wimmel et Wipeintner, 2001] proposent l'utilisation de la notion de patron. Les auteurs proposent l'utilisation des patrons de sécurité de base telle que le cryptage ainsi que le mécanisme des stéréotypes UML pour définir les niveaux de sécurité des données. L'approche présentée se concentre spécifiquement sur la définition des niveaux de sécurité des données dans les modèles de système. L'inconvénient majeur de cette technique est qu'elle ne considère pas les différents niveaux d'abstraction et donc ne bénéficie pas des avantages de l'approche MDA.

Les travaux présentés par Horvath et al. [Horvath et Dörges, 2008a] tentent d'appliquer les réseaux de Petri lors de la modélisation des patrons de sécurité dans le contexte de l'IDM. L'idée principale est de «traduire», ou encore modéliser, un patron spécifique sous forme d'un réseau de Petri. L'utilisation du mécanisme de raffinage successif assure la génération du code des réseaux de Petri correspondant. Le code exécutable des réseaux de Petri est faisable grâce à la sémantique opérationnelle des réseaux de Petri. Dans ce travail les auteurs ont choisi comme contexte d'application les systèmes multi-agent en utilisant l'approche de génie logiciel orientée agent (Aose). Par contre, les auteurs ne donnent aucun détail sur la généricité de leur méthode dans le sens de son application dans d'autres systèmes distribués.

La figure 4.5 résume les étapes essentielles qui constituent cette méthodologie. Après la spécification du patron, ce dernier est d'abord «traduit» en utilisant les réseaux de Petri

tout en gardant le même niveau d'abstraction. Par la suite, le modèle est raffiné jusqu'à un niveau où il peut être exécuté.

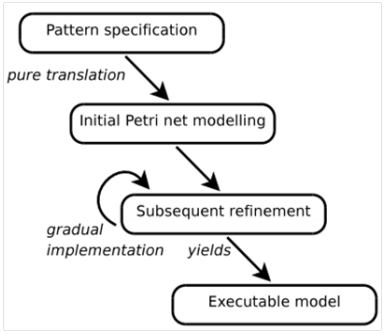


Figure 4.5 — La méthode proposée pour la modélisation des patrons avec les réseaux de Petri ([Horvath et Dörges, 2008a]).

Christian Wolter et al. en 2009 [Wolter *et al.*, 2009], étendent leurs travaux et plus particulièrement le modèle de politiques de sécurité présenté dans [Wolter *et al.*, 2008] par l'utilisation des patrons de sécurité. Le modèle étendu est présenté dans la figure 4.6. Comme le montre cette figure, les politiques sont interprétées par un module de sécurité qui prend en charge les patrons de sécurité spécifiques pour garantir les contraintes de sécurité déjà définies. Les patrons de sécurité sont utilisés pour mettre en correspondance les exigences de sécurité de haut niveau avec les mécanismes concrets qui appliquent les contraintes de sécurité mais sans préciser la façon ni la méthodologie adaptée pour le faire. D'autres questions telles que les critères de sélection des patrons, qui est responsable de cette sélection, la mise en ?uvre de ces patrons de sécurité n'ont pas été abordées.

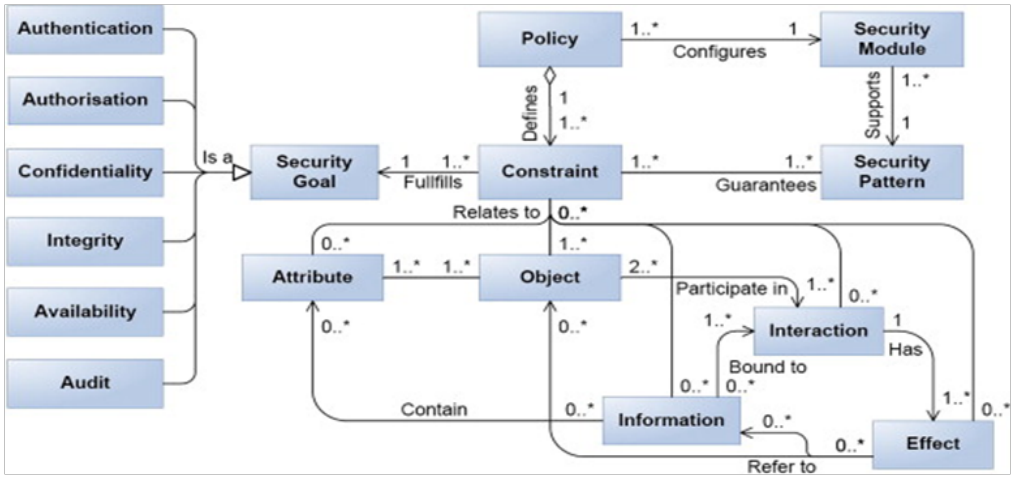


Figure 4.6 — Le modèle de politiques de sécurité étendu par la notion de patron de sécurité [Wolter *et al.*, 2009]. C'est une extension du modèle présenté par la figure 2 par la notion de patron de sécurité.

Abramov et al. [Abramov *et al.*, 2011], proposent une approche qui traite à la fois le niveau organisationnel et le niveau applicatif. En effet elle permet de proposer un schéma de

base de données conforme aux politiques de sécurité organisationnelles liées à l'autorisation. Premièrement au niveau organisationnel, les politiques de sécurité sont définies par des experts de sécurité et les experts du domaine, sous la forme de patrons de sécurité. Comme les patrons sont indépendants d'une application spécifique, ils peuvent être utilisés dans le développement de nombreuses applications. Puis au cours du développement de l'application, les concepteurs peuvent créer un modèle conceptuel de données tout en étant guidé par les patrons pour la mise en ?uvre des exigences de sécurité. Ce modèle est vérifié et comparé par rapport aux politiques de sécurité organisationnelles. Cependant, cette nouvelle approche est principalement dédiée au concepteur de base de données.

Le dernier travail investigué dans cette catégorie est le travail de Paul El Khoury [El-Khoury], [ElKhoury *et al.*, 2009b] dans sa thèse. Ces travaux porte sur la proposition d'un nouveau format de représentation des patrons de sécurité à été proposé. Ajoutant des niveau concepts comme les rôles. Aussi, l'utilisation du format XML pour la sauvegarde des patrons de sécurité dans une bibliothèque favorisant ainsi la réutilisation. la définition de la bibliothèque sous forme d'une ontologie est accompagné par un outils facilitant son alimentation. L'auteur aussi s'est intéressé à la composition des patrons de sécurité et la détection des conflits et des dépendances entre les patrons lors de leurs applications.

4.4 Discussion

Les approches présentées ci-dessus proposent toutes des solutions permettant de définir et de formaliser des solutions pour l'ingénierie de la sécurité. Certaines d'entre elles se distinguent par l'utilisation des patrons de sécurité favorisant la réutilisation des éléments de modélisation pour l'intégration de la sécurité, alors que d'autres se focalisent sur la définition des éléments génériques pour décrire les concepts de sécurité. Le but de cette synthèse est de définir dans un premier temps un ensemble de critères de comparaison pour toutes les approches présenté plus haut, et d'évaluer la pertinence de chaque solution par rapport à ces critères. Nous présentons dans la section suivante les critères de comparaison retenus en positionnant les approches étudiées par rapport à ces critères, puis nous présentons un tableau récapitulatif synthétisant les différentes approches étudiées.

4.4.1 Les critères de comparaison

Afin de comparer les différentes approches présentées ci-dessus, nous avons retenu un ensemble de critères permettant une caractérisation discriminante des approches :

- Degré d'automatisation : Il s'agit de la capacité de l'approche à sécuriser des modèles sans intervention humaine. En effet, notre objectif étant de proposer un processus automatique sans intervention externe
- Niveau de généricité et de réutilisation : il s'agit ici de considérer les aspects favorisant la restructuration et la réutilisation des éléments de modélisation et des mécanismes de composition proposés. Ce critère est analysé sur deux plans différents. Le premier plan permet d'étudier si l'approche propose des moyens pour définir une partie générique/variable dans les éléments de modélisation utilisés. Cette propriété favorise

la réutilisation et la structuration des systèmes. Le deuxième plan concerne les mécanismes offerts par une approche pour définir des opérateurs génériques pour introduire la sécurité de modèles.
- Propriétés de sécurité traitées : le terme sécurité comporte plusieurs exigence sous jacentes comme l'authentification et le contrôle d'accès. Ce critère est évidemment essentiel à considérer pour pouvoir identifier les approches couvrant le maximum d'exigences de sécurité.
- Outillage développé : ce critère permet d'identifier l'outil implémentant la solution proposée ainsi que son type : middleware, profil UML, modèle, framework. En effet plusieurs proposition n'ont pas atteint la phase de l'implémentation et se sont limité au niveau théorique.
- Support méthodologique : certaines approches proposent une méthodologie pour introduire la sécurité dans les systèmes ou les applications. Cette introduction peut être faite à travers les patrons de sécurité..
- Formalisation de l'approche : Ce critère caractérise l'existence d'une définition formelle du mécanisme d'introduction de la sécurité, dans le but de garantir des propriétés de sécurité sur tout le système et de valider, dans le cas d'application des patrons, l'application correcte des patrons de sécurité.

4.4.2 Comparaison des différentes approches

Cette section résume l'évaluation des différentes approches de composition étudiées selon les critères identifiés dans la section précédente. Le Tableau ? présente une synthèse des différentes approches par rapport au critères définit. .Comme nous pouvons le constater aucune des approches présentée ne satisfait l'ensemble des critères définit. Aussi , il y a très peu de travaux concernant l'intégration complète de la sécurité dans l'ingénierie des systèmes dès les premières phases du développement logiciel. Bien que plusieurs approches aient été proposées pour une intégration partielle de la sécurité, il n'existe actuellement aucune méthode générique pour aider les développeurs de systèmes critiques à introduire les aspects relatifs à la sécurité d'une façon claire et concise lors du développement d'applications ou de systèmes. Le manque de popularité de ces approches dans l'ingénierie de la sécurité pour le développement de systèmes logiciels est le résultat de : (i) les exigences de sécurité généralement difficiles à analyser et à modéliser et (ii) les développeurs manquent d'expertise dans le développement de logiciels sécurisés. Ces facteurs présentent des préoccupations supplémentaires lorsque l'on considère les exigences en termes de sécurité qui deviennent de plus en plus complexes et strictes dans divers domaines tels que le e-commerce, ou l'e-santé.

Les approches existantes ne couvrent pas l'ensemble du cycle de développement. Toutes ou presque se concentrent soit sur une étape particulière du cycle de développement (la conception, la mise en ?uvre, etc.) ou sur un aspect spécifique de la sécurité (le contrôle d'accès, l'authentification, etc.). En outre, les solutions existantes n'offrent généralement aucune indication sur la façon dont elles peuvent être intégrées dans des méthodes de développement des systèmes actuels. Les études empiriques faites par Tryfonas et al. [Tryfonas *et al.*, 2001] et Vaughn et al. [Vaughn *et al.*, 2002] confirment respectivement ce point de vue. En effet, il devient nécessaire de trouver une solution générique permettant l'intégration des

besoins de sécurité lors du développement des systèmes actuels en couvrant l'ensemble des phases du cycle de développement. C'est dans cet esprit que nous proposons une approche globale favorisant l'intégration de la sécurité via les patrons de sécurité lors du développement d'applications ou de systèmes critiques.

L'intégration des aspects relatifs à la sécurité dans le cycle de développement logiciel par le biais des patrons de sécurité est une approche assez nouvelle et innovante. Les travaux qui présentent cette technique sont encore rares et peu développés. Comme nous l'avons constaté à partir des études des travaux faites ci-dessus, les approches proposées sont sous forme de lignes directrices sans donner une méthodologie claire ou détaillée de la façon de faire cette intégration. D'autre part, les approches sont souvent bien spécifiques à un domaine d'application particulier et sont peu voir non génériques.

Toutes les approches présentées se sont intéressées à l'intégration des patrons dans des modèles objets, orienté service ou encore orientés agent. Nous constatons l'absence de travaux touchant les applications à base de composants. A notre connaissance, aucun travail n'a abordé l'intégration des partons de sécurité dans les modèles à composants. Cette problématique fera donc l'objet de ce travail de thèse.

Notre travail s'inscrit dans le cadre de l'ingénierie de la sécurité en utilisant des patrons de sécurité. Nous considérons que l'expert du domaine ne doit définir que les exigences de sécurité dans les modèles fonctionnels de haut niveau. Pour cela nous lui offrons la possibilité de sélectionner, d'appliquer les solutions proposées par les patrons de sécurité et enfin d'effectuer la génération de code à l'aide d'un outil dédié.

Tableau 4.1 — Tableau récapitulatif de l'évaluation des approches existantes

Approches	**Degré d'automatisation**	**Généricité et réutilisation**	**Propriétés de sécurité traitées**	**Support méthodologique**	**Formalisation**	**Outil support**
Ingénierie de la sécurité						
SecureUML	*	***	contrôle d'accès	Profil UML	non	oui
SecurityPolicyModel	*	**	Plusieurs	Modèle UML		
SIM	*	*	Authentification	Modèle UML		
UMLsec	*	***	Plusieurs	Profil UML		
SECTET	***	***	Plusieurs	Framework d'exécution		
eUML/eUMLsec	***	***	Plusieurs	Middleware CORBA	non	non
Intégration de la sécurité via les patrons						
[Fer, 2006]	-	***	Plusieurs	directive		
[Georg *et al.*, 2002] [Ray *et al.*, 2004]	*	***	Plusieurs	Programmation orienté aspects		
[Jurjens *et al.*, 2002]	-	**	Plusieurs	UMLsec/Autofocus		
[Gutiérrez *et al.*, 2005]	-	*	Plusieurs	Services web		
[Horvath et Dörges, 2008a]	*	***	Plusieurs	Processus, Réseau de Petri		
[Wolter *et al.*, 2009]	-	**	Plusieurs	Modèle UML		
[Abramov *et al.*, 2011]	***	*	Plusieurs	Modèle UML		
[ElKhoury]	**	***	Plusieurs		non	oui

Deuxième partie

Proposition

5 Intégration des patrons de sécurité dans les applications à base de composants

5.1 Introduction

Dans ce chapitre, nous détaillons l'approche proposée sous la forme d'une méthodologie complète permettant l'intégration des patrons dans les applications à base de composants. Comme présenté dans le chapitre précédent, différents travaux ont été menés afin d'intégrer la sécurité dans les systèmes ou les applications informatiques. Parmi les solutions proposées, quelques travaux ont déjà utilisés la notion de patron de sécurité, cependant, ces derniers se sont limitées à l'intégration des solutions proposées par les patrons dans des modèles orientés objets, orienté services ou encore à base d'agents. A notre connaissance aucun travail existant ne s'est intéressé aux modèles à bases de composants. Plus précisément, nous déplorons l'absence d'une méthodologie complète qui prend en charge l'intégration de ces solutions (patrons) durant toutes les phases du processus de développement d'une application. Dans ce contexte, l'absence d'un outil facilitant cette intégration est aussi un inconvénient majeur devant l'exploitation des solutions proposées. Pour pallier à ces manques, nous proposons, dans cette thèse, une approche globale permettant l'intégration des patrons de sécurité dans les architectures à base de composants sous la forme d'une démarche outillée couvrant tout le processus de développement logiciel allant de la phase de conception jusqu'à la génération de code. Dans un premier temps, nous détaillons la méthodologie proposée, puis et dans la troisième partie de ce manuscrit, nous présentons l'outil développé ainsi qu'une étude de cas complète.

5.2 Vue d'ensemble sur l'approche proposée

Dans ce travail, nous proposons une approche nommée SCRI-PRO pour SeCurity Pattern Integration apPROach. Cette approche assure l'intégration des patrons de sécurité dans des applications à base de composants. SCRI-PRO couvre toutes les phases d'un processus de développement d'un système. Ce processus touche la phase de conception, l'intégration des propriétés non-fonctionnelles (sécurité), et enfin, la mise en œuvre de ces propriétés et leurs intégrations dans le code fonctionnel de l'application.

SCRI-PRO combine à la fois les avantages de l'utilisation des patrons d'ingénierie (en particulier les patrons de sécurité), les points forts de l'ingénierie dirigée par les modèles ainsi que la souplesse de la programmation orientée aspect. Cela offre à l'utilisateur la possibilité d'intégrer les solutions proposées par les patrons d'une manière à la fois semi-automatique et modulaire.

Nous utilisons des patrons d'ingénierie pour spécifier différents types de propriétés non fonctionnelles. Le choix de ces patrons dépend du type de propriétés non fonctionnelles à mettre en œuvre ainsi que des exigences de l'application à concevoir. Par exemple, pour renforcer la gestion des droit d'accès dans une application distribuée, nous utilisons des patrons de sécurité et plus précisément ceux qui offrent des solutions pour la gestion des droits d'accès comme le patron RBAC [Schumacher *et al.*, 2006].

Lors du développement d'une application, permettre l'intégration des propriétés non fonctionnelles à un haut niveau à travers des patrons facilite la tâche du concepteur qui se concentre généralement sur la spécification des principales fonctionnalités de son application. En outre, grâce à ces patrons, les concepteurs même non experts auront la possibilité d'intégrer les différentes propriétés non fonctionnelles (comme la sécurité) grâce à la méthodologie proposée ainsi que les outils développés dans le cadre de ce travail.

L'approche proposée dans ce travail présente un processus complet pour intégrer d'une façon semi-automatique (guidé par les choix du concepteur) des patrons d'ingénierie dans le but de sécuriser une application à base de composants. Ce processus utilise les différents mécanismes offerts par l'ingénierie dirigée par les modèles (comme les profils UML) ainsi que les transformations des modèles. Nous proposons la génération automatique des squelettes du code des aspects qui traduit les solutions de sécurité appliqués à travers les patrons, au niveau code.

Tel que présenté au-dessus, nous avons choisi d'utiliser la programmation par aspect pour mettre en place les parties du code qui traitent les propriétés non-fonctionnelles. Ce choix s'explique par le fait que la Programmation Orientée Aspect (POA) a gagné en popularité en tant que paradigme de programmation qui supporte la modularité des préoccupations transversales comme la sécurité. Ce caractère transversal est une caractéristique que les aspects partagent avec les spécifications de propriétés non fonctionnelles.

Dans le cadre de ce manuscrit, nous allons illustrer cette approche en prenant la sécurité comme l'une des propriétés non fonctionnelle principales. Il est à noter que cette approche est illustrée dans cette thèse à travers l'utilisation des patrons de sécurité (comme exemple de patron d'ingénierie) dans le but de sécuriser un modèle à composants pris comme exemple de domaine d'application. Ce choix est motivé par l'absence de travaux d'intégration des patrons de sécurité dans les modèles à base de composants (voir discussion chapitre 4).

Nous adoptons une méthodologie basée sur le MDA. Nous proposons la démarche représentée par la figure 5.1 Cette dernière peut être lue :

Verticalement : les deux vues du MDA sont représentées. La vue indépendante de toutes plateformes et la vue spécifique à une plateforme. Nous pouvons voir aussi que les deux phases qui constituent la démarche proposée sont représentées : la phase de modélisation,

qui correspond aussi à la vue indépendante de plateforme de la MDA, et la phase d'implémentation, qui représente la vue spécifique à une plateforme.

La phase de modélisation consiste à : (1) Proposer le modèle à composants de l'application à réaliser (2) Annoter le modèle de l'application ainsi défini par les solutions proposées par les patrons de sécurité à travers, dans un premier temps, la définition des profils UML permettant d'étendre le méta-modèle de composants avec les concepts nécessaires pour la spécification des propriétés non-fonctionnelles (par exemple la sécurité). Dans un second temps, la spécification des règles d'intégration des solutions proposées par les patrons d'ingénierie. Ces règles une fois spécifiées peuvent être appliquées au modèle de l'application pour générer un modèle supportant les propriétés non-fonctionnelles.

La phase d'implémentation couvre la génération du code fonctionnel de l'application. Ce code ne contient que la logique métier de l'application et n'aborde pas les propriétés non fonctionnelles. Ces propriétés seront spécifiées à travers les templates aspects. Les aspects générés seront automatiquement intégrés, de façon modulaire, au code fonctionnel défini dans cette même phase.

Horizontalement, nous présentons la séparation entre les différentes préoccupations. Cette séparation est maintenue dans toutes les étapes de la démarche proposée. Notre démarche vise à découpler l'expertise du domaine d'application (préoccupations fonctionnelles) de l'expertise encapsulée par les patrons spécifiques comme les patrons de sécurité.

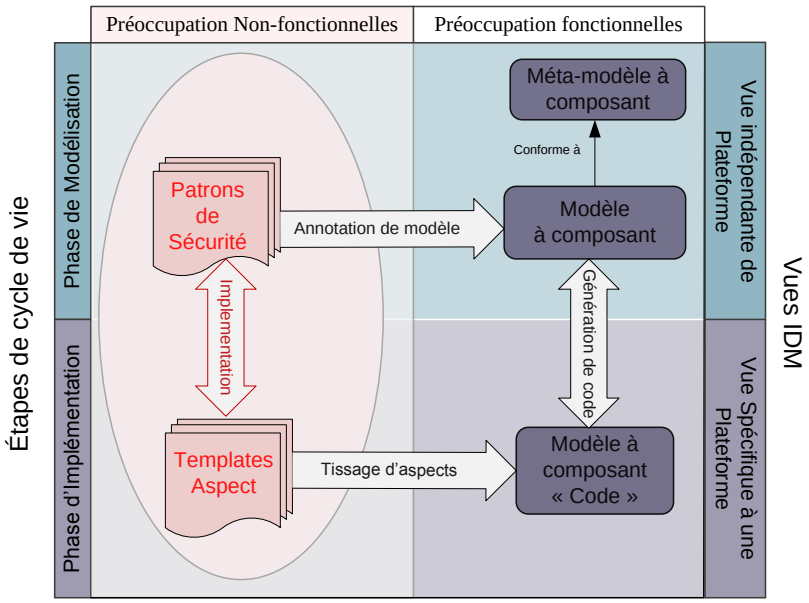


Figure 5.1 — Vue d'ensemble de l'approche SCRI-PRO

Les transformations entre les modèles sont représentées par des grandes flèches dans la figure 5.1. Ces transformations peuvent avoir deux types différents :

– Les patrons peuvent être intégrés dans un modèle à l'aide d'un profil UML qui étend le méta-modèle avec les concepts de sécurité. Puis dans un deuxième temps, cette intégration sera semi-automatique à travers l'exécution des règles de transformation. L'exécution de ces règles permet l'annotation du modèle à base de composants avec

les concepts du patron.
– Pour transformer un modèle d'une vue indépendante de la plateforme (PIM) en un modèle dépendant de la plateforme (PSM) (principe du MDA), nous utiliserons des templates aspect qui seront tissés au code fonctionnel.

La grande flèche (en rouge) entre les patrons de sécurité et les templates des aspects illustre l'utilisation du paradigme aspects dans l'implémentation des solutions proposées par les patrons de sécurité. Concrètement, lors de la sélection de patrons de sécurité (selon les exigences de sécurité pour une application particulière) au niveau abstrait (niveau PIM), les templates des aspects de sécurités correspondants sont générés.

Dans ce qui suit, nous décrivons en détail les différentes étapes qui constituent notre démarche. Comme indiqué précédemment, cette démarche n'est pas entièrement automatisée, en effet nous avons concentré nos recherches principalement sur l'automatisation des transformations permettant l'intégration des solutions de sécurité. De nombreux travaux ont déjà été réalisés entre la phase de modélisation et d'implémentation (de nombreux outils de génération de code existent déjà). Nous détaillons dans ce qui suit les différentes phases du processus SCRI-PRO.

5.3 Les phases de développement

La phase d'analyse et de définition des besoins, non spécifique aux applications à base de composants, n'est pas détaillée dans cette thèse. La figure 5.2 illustre les étapes nécessaires dans la phase de modélisation et la phase d'implémentation. Le tableau 5.1 présente les différents modèles utilisés dans chaque étape.

5.3.1 La phase de modélisation

A ce stade, un diagramme de composants décrit les composants et leurs interfaces (ports, connecteurs, opérations et messages) pour une application à sécuriser. Ce digramme est défini en utilisant le méta-modèle UML 2.0. Il constitue le modèle indépendant de la plateforme PIM (Platform Independent Model). Pour sécuriser son application, un architecte (pas nécessairement un expert en sécurité) peut choisir parmi les patrons de sécurité ceux nécessaires pour résoudre un problème de sécurité ou encore appliquer une politique de sécurité particulière.

Lors de cette application, le concepteur peut utiliser les informations textuelles présentes sur les patrons de sécurité, (conséquences, contexte, etc), pour prendre des décisions nécessaires. Les relations entre les patrons de sécurité peuvent également être utilisées pour suggérer des choix pour les patrons concernant un problème particulier. Une fois les patrons de sécurité choisis, ils doivent être inclus dans le PIM afin de produire un PIM sécurisé, noté secPIM. Cette transformation semi-automatique de PIM à secPIM est nommée PIM2secPIM et est définie dans la section 5.4.2. Elle nécessite la définition d'un profil UML qui étend le méta-modèle de composants avec des concepts de sécurité définis dans les patrons de sécurité. Cette transformation nécessite la définition de plusieurs règles d'application des patrons de sécurité choisis. La méthode d'élaboration de profil ainsi que celle des règles

d'intégration sont détaillées respectivement dans les sections 5.6 et 5.7.

5.3.2 La phase d'implémentation

Une transformation automatique intermédiaire, appelée PIM2PSM, consiste à produire le modèle spécifique à la plateforme PSM (Platform Specific Model) à partir du modèle PIM. Il s'agit d'une transformation MDA traditionnelle (n'est pas décrit dans le cadre de ce travail), nous utiliserons les travaux déjà proposés pour transformer un modèle à base de composants UML en un modèle spécifique à une technologie comme EJB, CCM, etc. Des templates aspects seront générés automatiquement à partir de modèle secPIM encapsulant les solutions de sécurité. Ces templates permettent de générer des aspects qui se tissent au code fonctionnel de l'application à travers une transformation PSM2secPSM. Cette transformation produit une version sécurisée en utilisant le tissage d'aspects.

Le tableau 5.1 synthétise les différentes transformations. Pour chacune d'entre elles nous présentons les entrées et les sorties respectifs ainsi que les outils potentiels qui peuvent servir lors de la réalisation de la transformation en question.

Tableau 5.1 — Synthèse des transformations

Nom de la Transformation	**Entrées**	**Sorties**	**Outils utilisés**	**Type**
PIM2secPIM	PIM : Modèle de l'application à base de composant conforme au méta-modèle de composant UML 2.0	SecPIM : Modèle de l'application à base de composant sécurisé	Règles de transformation en langage ATL	Semi-automatique
PIM2PSM	PIM : Modèle de l'application à base de composant conforme au méta-modèle de composant UML 2.0	PSM : Modèle de l'application à base de composant spécifique à une plateforme (EJB, CCM, ...)	Utilisation d'outils déjà existants, par exemple Acceleo [Acceleo]	Automatique
PSM2secPSM	PSM : Code fonctionnel de l'application à base de composants	SecPSM : Code de l'application à base de composants sécurisée	Tisseurs d'aspects , par exemple AspectJ [AspectJ]	Semi-automatique

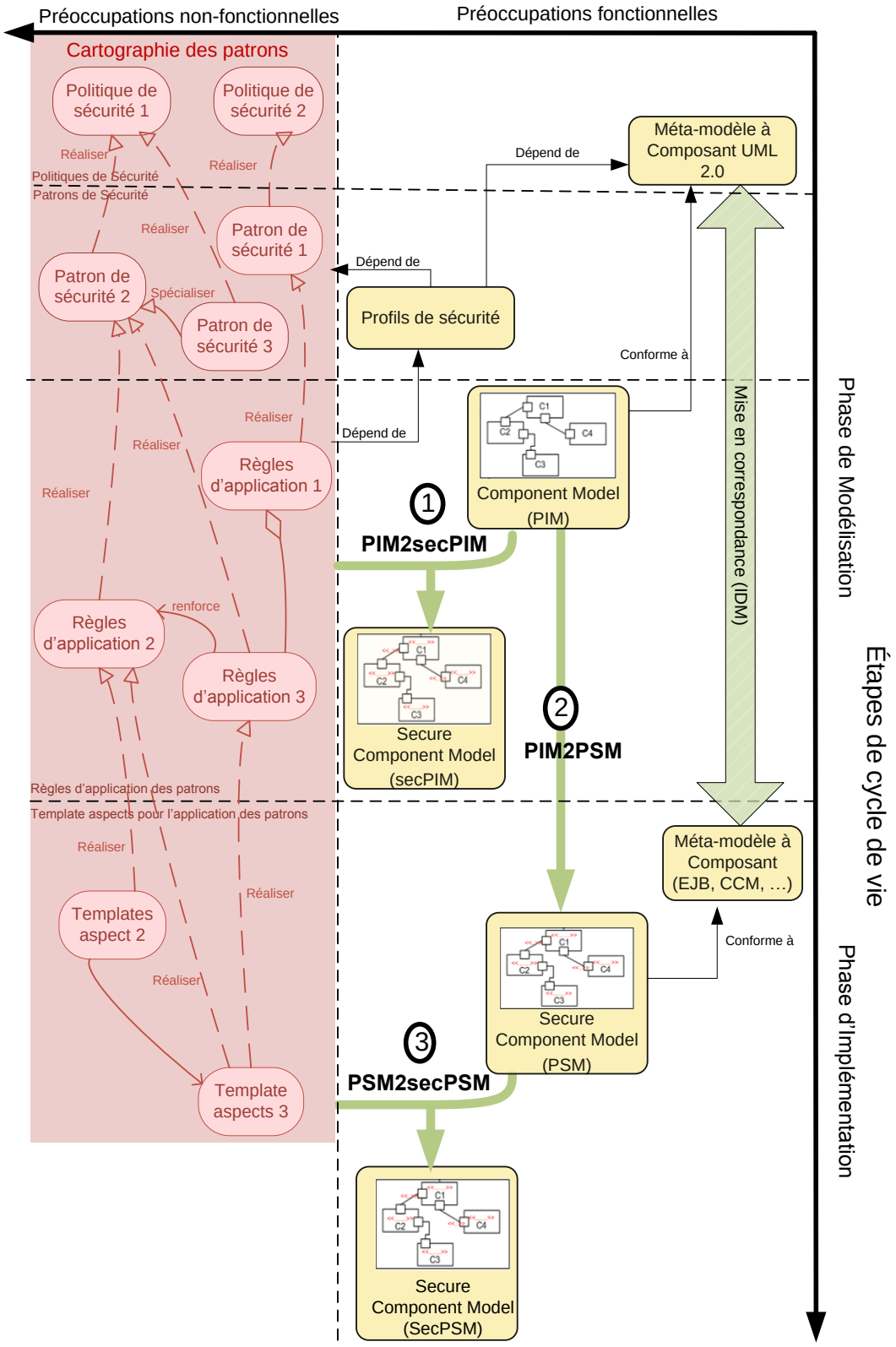


Figure 5.2 — Une démarche pour la sécurisation des applications à base de composants

5.4 Définitions des transformations de modèles du processus

Dans cette section, nous définissons les transformations qui permettent d'intégrer dynamiquement les patrons de sécurité à un modèle, produisant ainsi un modèle sécurisé. Nous définissons d'abord la transformation de PIM à PSM (PIM2PSM), puis nous décrivons la transformation des modèles non sécurisés à des modèles sécurisés (PIM2secPIM et PSM2secPSM).

5.4.1 Définition de la transformation PIM2PSM

Cette transformation est un cas classique de l'IDM. En effet nous allons, comme mentionné ci-dessous utiliser des travaux déjà existants pour le passage entre un modèle indépendant de la plateforme à un modèle spécifique à une plateforme. A partir des correspondances (règles) entre chaque méta-modèle (méta-modèle de composant UML et méta-modèle de composants spécifique à la plateforme telle que EJB, CCM), et un PIM, nous allons générer un PSM. En plus du modèle PSM produit, une correspondance entre le PIM et le PSM est générée, elle enregistre les relations entre les éléments des deux modèles et fournit ainsi la traçabilité à travers le processus. Pour ce faire nous pouvons aussi utiliser des travaux déjà existants pour générer un modèle EJB, par exemple, à partir d'un modèle à composant UML 2.0. En nous basant sur une technique semblable à celle utilisée dans [Ziadi *et al.*, 2002] qui consiste à produire des règles de mise en correspondance pour générer à partir d'un modèle à composants indépendant d'une plateforme, des squelettes de classes EJB. Ces règles permettent de générer que des EJB Session beans ce qui nous amène à compléter ce manque par l'utilisation du profil EJB proposé dans [UML, a]. Dans le but de générer un modèle spécifique à une plateforme, la transformation sera appliquée dans un seul sens. Ainsi, nous pouvons considérer un modèle source et le modèle cible :

- Le modèle source est le PIM du système, conforme au méta-modèle à composants UML 2.0
- Le modèle cible généré sera le PSM du système, conforme au méta-modèle de composants choisi (EJB, CCM, ...).

5.4.2 Définition de la transformation PIM2secPIM

Les patrons de sécurité sont essentiellement abstraits, car ils sont des solutions à des problèmes récurrents. Ainsi, nous incorporons les concepts des patrons de sécurité à des méta-modèles. En outre, nous devons produire, dans un premier temps, des profils UML (voir section 5.6 de ce même chapitre) permettant de mettre en relation les concepts du méta-modèle à base de composants et le modèle de patron de sécurité. Dans un second temps, nous produisons des règles d'applications de ces solutions (voir section 5.7). Enfin, ces règles doivent être appliquées en utilisant un langage impératif comme ATL (voir section ATL 3.2.4.3) ou n'importe quel langage de transformation de modèle afin d'annoter le modèle de l'application pour produire un modèle sécurisé (secPIM).

5.4.3 Définition de la transformation PSM2secPSM

Cette transformation est réalisée au niveau de l'implémentation. Elle consiste à produire un code sécurisé de l'application spécifique à une plateforme. Pour prendre en compte des avantages de la programmation orient aspects, nous proposons d'élaborer des templates aspects permettant l'intégration des solutions proposées par les patrons de sécurité. Ces templates seront utilisés pour générer des aspects spécifiques à une application. Cette transformation consiste alors à tisser le code des aspects générés au code fonctionnel de l'application résultant de la transformation PIM2PSM. Le produit de cette transformation est le modèle secPSM.

5.5 Une cartographie des patrons de sécurité

Dans cette section, nous présentons une cartographie de patrons de sécurité. Dans un premier temps, nous définissons la structure en niveaux de cette cartographie, puis le métamodèle permettant de représenter les patrons dans cette cartographie, enfin, la structure et l'utilisation de cette cartographie.

5.5.1 Introduction

Prendre des décisions en matière de sécurité est une tâche complexe qui ne peut être faite automatiquement. Dans la littérature, nous pouvons constater l'absence de règles systématiques aidant les concepteurs à sécuriser leurs applications ou systèmes. En revanche, les patrons de sécurité existent et constituent un ensemble de connaissances en sécurité. Ces dernières année et avec l'essor des patrons d'ingénierie en générale et les patrons de sécurité en particulier, nous pouvons constater le grand nombre de ces patrons proposés dans le monde académique et industriel. Bien que ces patrons présentent des solutions efficaces pour des problèmes spécifiques, le choix d'un patron par rapport aux autres reste un problème pour les développeurs non experts en sécurité. Il y a une prolifération des patrons de sécurité, il est donc de plus en plus difficile de choisir lesquels appliquer.

Dans cette section, nous proposons d'organiser les patrons de sécurité sous forme d'une cartographie. Celle-ci sera navigable, de tel sorte qu'il est plus facile pour un concepteur de prendre ces décisions en terme de sécurité. Autrement dis, selon un arbre de décision, l'utilisateur (développeur) est guider pour sélectionner le patron adéquat selon le problème rencontrer.

5.5.2 Relations entre les patrons de sécurité

Pendant le processus de développement du système, le problème est d'abord analysé puis la solution au problème est itérativement affinée. Au cours de chaque phase du cycle de vie du logiciel, la solution est considérée à un niveau d'abstraction différent. La sécurité est une exigence non-fonctionnelle qui doit être pensée à partir de la phase d'analyse des besoins. Dans ce contexte, les patrons de sécurité existent à différents niveaux d'abstraction. Les patrons de sécurité résidant dans la couche la plus élevée sont des patrons exprimant

des politiques, qui seront appliquées indépendamment de la technologie ou de la plateforme utilisée. Au niveau le plus bas, on trouve les patrons de sécurité qui implémentent ces politiques de sécurité. Puis les règles permettant d'implémenter ces patrons et finalement au dernier niveau les aspects implémenter ces solution de sécurité.

Nous décrivons ici les types de relations entre les patrons de sécurité qui apparaissent dans la cartographie proposée. Une première classification des relations entre les patrons de conception a été proposée par Zimmer en 1995 [Zimmer, 1994]. Dans le cadre des patrons de sécurité, une classification différente des patrons est donnée dans [Fernández *et al.*, 2008] et [Hafiz *et al.*, 2007]. Les relations entre les patrons peuvent exister. Le type de relation qui relie un ou plusieurs patron est une relation de réalisation. Nous avons identifié trois types de relations entre les patrons : la contenance, la spécialisation et la collaboration.

- La relation de contenance : On dit que le patron P1 contient le patron P2 lorsque le problème résolu par le patron P1 est plus large que celui résolu par le patron P2 et que le patron P1 utilise le patron P2 dans sa solution.
- La relation de spécialisation : On dit que le patron P1 spécialise le patron P2 si les deux patrons permettent la résolution du même problème, mais la solution proposée par le patron P1 est plus détaillée que celle de P2.
- La relation de collaboration : On dit que le patron P1 collabore avec le patron P2 lorsque le patron P1 et le patron P2 s'utilisent pour résoudre le même problème.

5.5.3 Un méta-modèle des patrons de sécurité

Dans le but de proposer un formalisme permettant de décrire les patrons de sécurité collectés suite à l'état de l'art présenté dans chapitre 2 et les relations entre eux, nous avons proposé le méta-modèle suivant (voir figure 5.3). Ce méta-modèle permet d'unifier la structure de ces patrons facilitant ainsi leurs classifications ainsi que leurs manipulations.

La structure d'un patron est constitué de :

- Un nom : identifie de manière unique un patron.
- La politique de sécurité à la quel il appartient.
- Description textuelle du patron :
  - Contexte : L'environnement dans lequel le patron peut être appliqué
  - Problème : Le problème que résout le patron.
  - Solution : une description de la façon dont le patron permet de résoudre le problème.
  - Conséquences : Décrit comment le patron réalise ses objectifs et présente les résultats.
  - Utilisations remarquables : exemples réels dans les quels ce patron a été observé
- Un ensemble de relations avec d'autres patrons, qui indiquent un ensemble de patrons liés.
- Un modèle UML qui est une description générative de la solution. Représente les classes participant dans la solution du patron et montre leurs interactions dans des diagrammes UML.

La figure suivante 5.3 décrit le méta-modèle de patrons de sécurité proposé :

Exemple d'instanciation de ce métamodèle : Afin d'illustrer l'instanciation de ce méta-

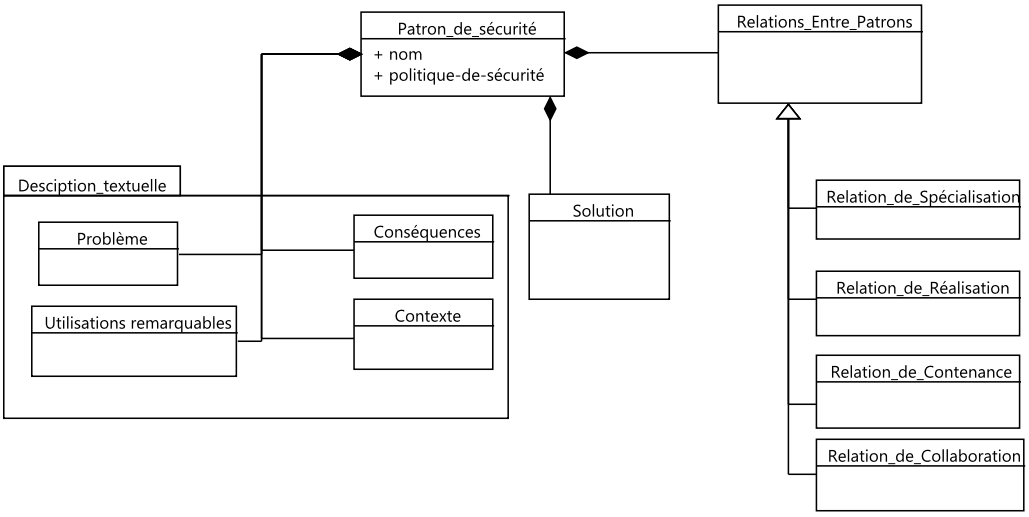


Figure 5.3 — Méta-modèle de patrons de sécurité

modèle de patron de sécurité nous présentons dans ce qui suit un exemple.

- Un nom : RBAC
- La politique de sécurité à la quel le patron appartient : Contrôle d'accès
- Description textuelle du patron :
  - Contexte : Tout environnement où nous avons besoin de contrôler l'accès aux ressources informatiques et où les utilisateurs peuvent être classés en fonction de leurs compétences ou de leurs tâches.
  - Problème : Dans les organisations, les utilisateurs ont des rôles différents qui exigent des compétences et des responsabilités, et ils devraient donc avoir des droits d'accès aux données, qui sont fondés sur leur rôle.
  - Conséquences : En introduisant la notion des rôles l'effort administrative est réduite, car il n'est pas nécessaire d'attribuer des droits individuellement. La structuration basée sur les rôles permet le traitement des groupes plus larges.
  - Utilisations remarquables : exemples réels dans les quels ce patron a été observé.
- Solution : le mécanisme RBAC [Ferraiolo et Kuhn, 1992] décrit pour chaque utilisateur les privilèges qu'ils peuvent acquérir en fonction de ses rôles ou des tâches qui lui sont accordées. Afin de soutenir le mécanisme RBAC lors de l'analyse et la conception d'applications, un patron correspondant a été développé [Schumacher *et al.*, 2006]. Le patron RBAC est présenté sur la figure 5.4. Les utilisateurs User sont assignés à des rôles, tandis que les rôles donnent les droits qui sont autorisés aux utilisateurs de ce rôle. Comme dans le patron d'autorisation, la classe d'association Right définit les types d'accès qu'un utilisateur dans un rôle est autorisé à appliquer sur la ProtectionObject. Une mise en œuvre correcte du patron RBAC assurera un contrôle d'accès efficace et sécurisé au ProtectionObject.
- Relation avec d'autres patrons :
  - Relation de spécialisation : -
  - Relation de contenance : Patron d'autorisation
  - Relation de Collaboration : Patron de session

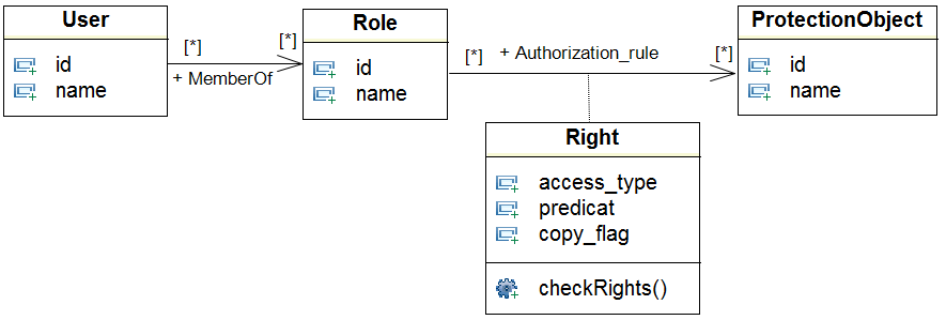


Figure 5.4 — Structure du patron RBAC

5.5.4 La cartographie

Dans ce travail, nous proposons une cartographie (voir figure 5.5) pour la structuration des patrons de sécurité. Toutefois, cette carte n'est pas exhaustive. De nouveaux patrons peuvent être proposés et intégrés à cette cartographie. En outre, tous les patrons utilisé et répertoriés dans cette cartographie ont été proposés dans la littérature par différents auteurs [Yoder et Barcalow, 1997], [Schumacher *et al.*, 2006] et dans différents domaines.

Dans notre cartographie, le plus haut niveau, celui des "politiques de sécurité", contient les politiques comme le contrôle d'accès, l'authentification, etc. Ces politiques définissent les besoins en termes de sécurité d'une application. La deuxième couche, moins abstraite, "Patrons de sécurité", contient des patrons qui décrivent les mécanismes de sécurité. Ils décrivent comment réaliser des politiques de sécurité décrites précédemment. Par exemple, le patron «Single access point» et le patron «Authenticator» de cette couche peuvent être utilisés en collaboration pour réaliser des politiques de «contrôle d'accès». D'autres couches peuvent être rajoutées pour implémenter ces patrons comme présenté par la figure 5.5. Une couches contenant les règles d'application des différents patrons de sécurité et une autre couche regroupant les templates des aspects permettant la génération de code aspects des solutions de sécurité choisis. Nous gardant la même sémantique des relations pour toutes les couches de cette cartographie. En d'autre terme, un ensemble de règles d'intégration dans patron X peuvent être renforcées par un ensemble d'autre règles d'intégration d'un patron Y.

Utilisation de la cartographie

Puisque les concepteurs doivent ajouter la sécurité à leurs modèles pendant toutes les phases de cycle de développement logiciel, ils doivent prendre des décisions en matière de sécurité. Dans notre approche, les décisions de sécurité sont en fait le choix des patrons de sécurité appropriés qui devraient être appliqués. Toutefois, en raison de la prolifération des patrons de sécurité, il peut être difficile de les sélectionner.

Un arbre de décision est un outil d'analyse décisionnelle qui utilise un graphe de nœuds pour modéliser tous les choix et leurs conséquences possibles. Mathématiquement, il n'a pas besoin d'être un arbre, mais au moins un graphe acyclique dirigé (c.à.d. graphe orienté qui ne possède pas de cycle). Notre cartographie peut être considérée comme une forêt d'arbres

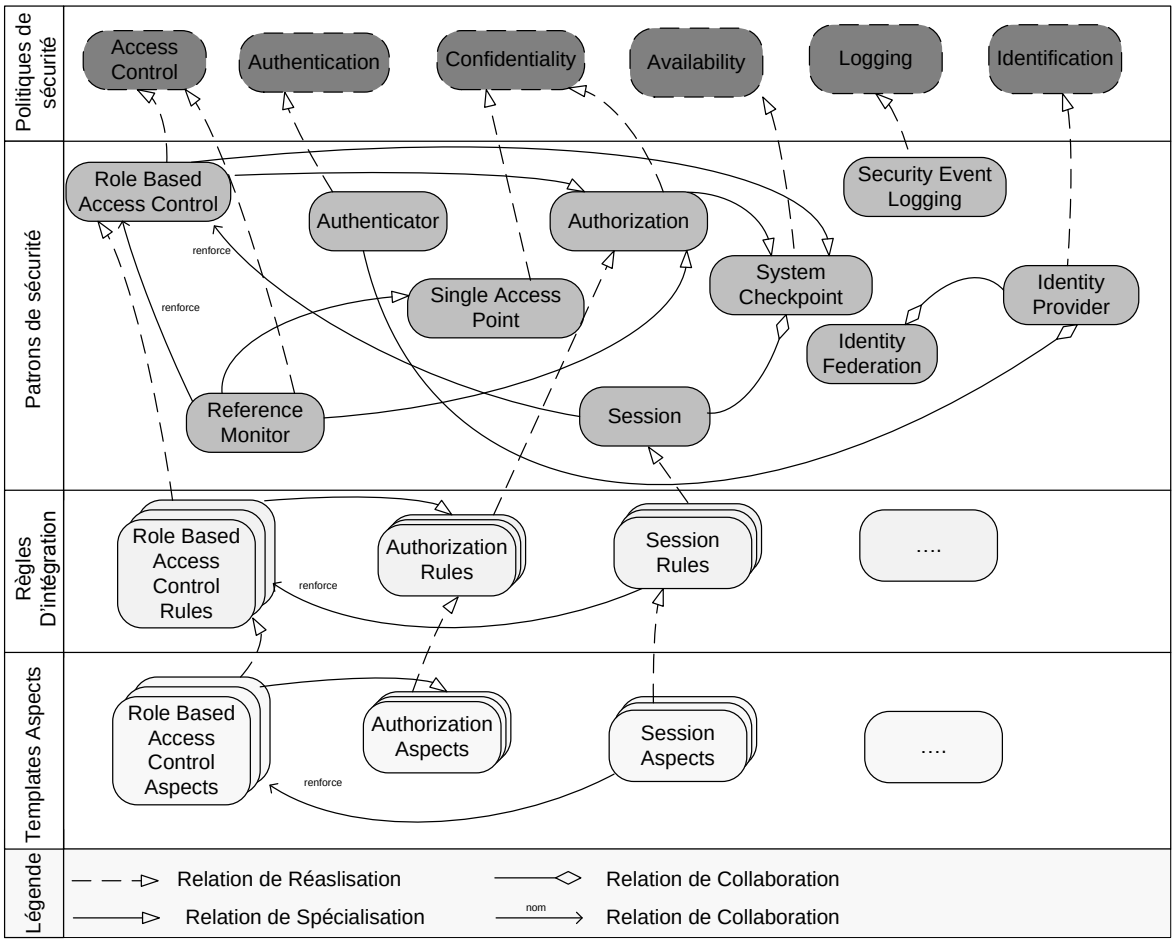


Figure 5.5 — Cartographie des patrons de sécurité

de décision utilisés pour choisir les patrons de sécurité appropriés (dans ce qui suit nous présentons un exemple d'arbre de décision) :

- Tous les nœuds dans le plus haut niveau représentent la décision suivante : "Comment pouvons-nous mettre en œuvre la politique de sécurité représentée par ce n ?ud ?"
- Les arcs des arbres de décision sont les relations entre les patrons de la cartographie.
- Les racines des arbres sont choisies en fonction des besoins de sécurité c.à.d. des politiques de sécurité à appliquer.

Les décisions en termes de sécurité sont finalement prises par des concepteurs (humains) à partir de la lecture des parties textuelles de la structure des patrons. En particulier, les sections «conséquences» de patrons candidats, car elles décrivent les résultats suite à l'application des patrons choisis.

En outre, l'utilisation de la cartographie permet également une bonne traçabilité des décisions tout au long de cycle de développement. Les décisions de sécurité sont enregistrées comme une séquence de nœuds (patrons de sécurité). Si le processus est itératif, un retour en arrière dans un même arbre est possible.

Exemple : *L'identification des politiques de sécurité et l'utilisation de la cartographie pour sélec-*

tionner les patrons de sécurité appropriés

Afin d'illustrer l'utilisation de la cartographie proposée, nous présentons ici un exemple d'application inspirée du système GPS. Pour simplifier la description, certaines fonctionnalités ne sont pas présentées.

Nous avons identifié des exigences de droits d'accès aux services offerts par les composants entre le segment utilisateur et le segment spatial du système GPS. Dans cet exemple, nous considérons principalement la gestion des droits d'accès aux différents services proposés par les opérateurs téléphoniques en particulier le téléchargement des cartes en temps réel ainsi que la gestion d'accès aux satellites sécurisés. Le système Basic GPS (voir Figure) fonctionne comme suit :

1. Le Terminal GPS reçoit en permanence le signal des Satellites ainsi que celui de Secure-Satellite à condition qu'il possède les droits d'accès nécessaire.
2. Le Terminal GPS envoie une requête de téléchargement de carte au Telephone Operator.
3. Le Telephone Operator permet à l'utilisateur, en fonction de ses droits d'accès, de télécharger la carte demandée.

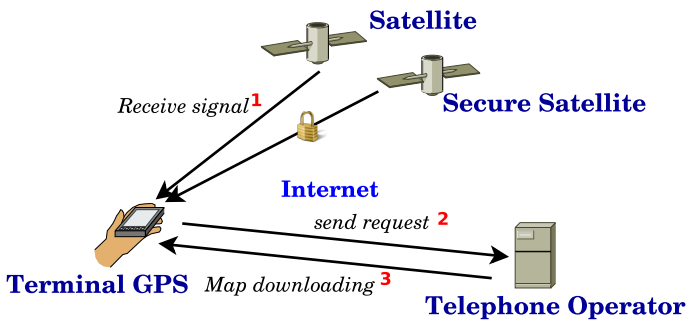


Figure 5.6 — Fonctionnement du système Basic GPS

Dans le cadre du système Basic GPS, nous avons identifié le cas d'utilisation : L'accès à des satellites sécurisés. Pour ce fait, nous avons identifié la politique de sécurité qui doit être appliqué : La politique de contrôle d'accès.

Par conséquent, nous ne considérons que les arbres de décision ayant la politique de contrôle d'accès comme racine. Comme illustrer dans la figure 5.5, nous pouvons voir que partant de cette racine et en suivant les liens de "réalisation", nous sélectionnons le patron "RBAC", le patron "Session", le patron "system check Point", le patron "Autorization" et le patron "référence monitor". Un profil UML de sécurité relatif à cette famille de patrons rependant à une même exigence ou politique de sécurité peut être élaboré. Le profil et ainsi élaboré et pendant la phase de modélisation, le concepteur peut choisir d'appliquer un ou plusieurs patron appartenant à cette même famille, par exemple, le patron "RBAC. Ainsi l'arbre nous guide vers la sélection des règles d'application du patron choisis. Après et dans la phase d'implémentation, l'arbre nous proposera aussi les templates aspects permettant de générer le code relatif à la génération de la solution de sécurité proposée par le patron en question, dans notre cas il s'agit du patron "RBAC".

Outils

La cartographie peut être implémentée à l'aide d'un outil informatique. Dans le cadre de ce travail, nous avons décrit comme un graphe des patrons de sécurité et nous avons défini un méta-modèle des patrons dans le langage UML. La réalisation et le développement de cet outil est parmi les travaux futurs que nous comptons réaliser rapidement.

En outre, les patrons peuvent être importés/exportés/édités en utilisant UML/-MOF/XMI, ce qui ajoute de la flexibilité dans le processus. L'outil peut facilement guider dans le choix des patrons de sécurité en naviguant dans la cartographie proposée et en suggérant les patrons accessibles et qui pourraient être appliquées.

5.6 Élaboration du profil UML

Dans la section précédente, nous avons proposé une cartographie des différents patrons de sécurité ainsi qu'un méta-modèle unifiant la description de ces patrons. La présente section propose une méthode permettant l'élaboration des profils de sécurité relatifs à chacune des politiques de sécurité présente dans la première couche de la cartographie des patrons.

5.6.1 Introduction

Notre approche repose sur l'intégration des patrons de sécurité à un haut niveau d'abstraction dans la conception de systèmes à base de composants. Nous proposons des étapes permettant d'intégrer la solution contenue dans ces patrons :

- La première étape consiste à produire le profil UML permettant d'adapter ces solutions au concept du domaine des composants.
- La deuxième étape consiste à élaborer des règles d'application permettant de semi-automatiser cette intégration.

Un profil de sécurité représente une solution spécifique à un domaine. Il embarque l'expertise en sécurité fournie par les patrons de sécurité appartenant à une même politique de sécurité (niveau politique de sécurité dans la cartographie des patrons), c.à.d. qu'un profil est élaboré pour chacune des politiques de sécurité définie pour sécuriser une application à base de composants. Ce qui implique la définition de plusieurs profils UML de sécurité dans le cas ou nous choisissant de garantir différentes exigences de sécurité et par conséquent satisfaire plusieurs politiques de sécurité. Ces profils permettent d'étendre le méta-modèle de composants par les concepts de sécurité des patrons de sécurité appartenant à une même arbre de décision.

5.6.2 Méthode de production du profil UML

La définition d'un profil UML passe par une suite d'étapes présentée dans la figure 5.7. Dans ce qui suit nous utiliserons la terminologie suivante :

- Les participants : artefacts du patron de sécurité
- Les éléments : artefacts (méta-classes, relations, ...) du méta-modèle à base de composants

Ces différentes étapes peuvent être résumées comme suit :

- **Mise en correspondance.** Cette étape consiste à analyser le problème de sécurité donné et faire coïncider le problème avec la politique de sécurité qui permet de le résoudre. Ainsi un arbre de décision de patrons de sécurité parmi ceux qui sont enregistrés dans la cartographie (voir figure 3) peut être extraite. Le concepteur est responsable de déterminer quelle politique de sécurité et par conséquent quels patrons de sécurité appliquer pour quel problème. Une fois la politique de sécurité et l'ensemble de patrons de sécurité correspondant sont choisis. L'étape de mise en correspondance peut être déclenchée afin de produire une correspondance entre les participants des patrons choisis et les éléments du méta-modèle de composants. Ces participants vont étendre les éléments du méta-modèle ainsi choisis. La similarité ou encore la correspondance peut être déterminée à l'aide de simples heuristiques. Par exemple, nous pourrions utiliser un dictionnaire de termes similaires. Supposons que nous utilisions un simple dictionnaire des termes similaires, dans lequel par exemple : "User" = "Composant" et "ProtectionObject" = "Composant". Ce dictionnaire comportera les participants des patrons choisis et leurs éléments du méta-modèle correspondants.
- **Élaboration du profil.** Après la mise en correspondance entre les participants des patrons de sécurité et les métaclasses du méta-modèle de composants, l'identification des stéréotypes parmi les participants du patron peut être lancée ainsi que l'identification des métaclasses à étendre. Tous les participants des patrons de sécurité choisis sont désignés comme des stéréotypes qui étendront les métaclasses du méta-modèle de composants. Ces dernières sont définies à partir des éléments du méta-modèle dans le dictionnaire. Les métaclasses sont des concepts du méta-modèle qui peuvent être étendus et adaptés à un domaine plus spécifique caractérisant le profil. Des valeurs marquées peuvent également être ajoutés à des stéréotypes qui sont associés aux métaclasses UML adéquates.

Le résultat de ces étapes est la définition d'un profil de sécurité UML représentant la solution de sécurité choisie. Ce profil UML fournit un moyen approprié pour définir une correspondance pour chaque solution proposée par les patrons de sécurité et le méta-modèle de composants pour permettre l'application de cette solution aux éléments du modèle de l'application.

Exemple : *Élaboration de profils UML de sécurité pour l'exemple GPS et la politique de contrôle d'accès retenue*

Avant d'illustrer la méthode de l'élaboration du profil UML de sécurité relatif à la politique de contrôle d'accès retenue dans la section . Nous rappelons le noyau du méta-modèle de composants UML 2.0.

La figure 5.8 présente un extrait du méta-modèle UML2.0 pour la description de composants.

Un composant hérite de la mèta-classe Class. Il peut définir des opérations et participer dans des généralisations. Il hérite également de EncapsulatedClassifer. Il peut donc posséder des ports, typés par des interfaces fournis et requises. La métaclasse EncapsulatedClassifer hérite de StructuredClassifier. Par conséquent, un composant peut avoir une structure interne et peut aussi définir des connecteurs entre ses composants internes Après avoir rap-

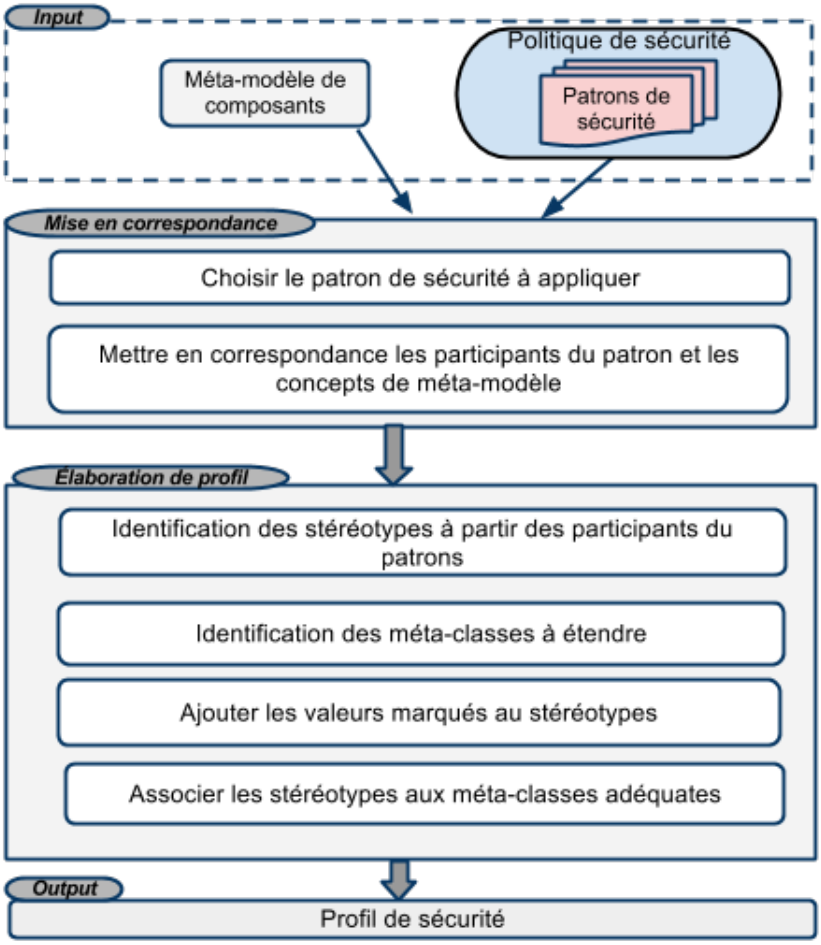


Figure 5.7 — Étapes de production de profils UML pour l'intégration des patrons de sécurité. Illustration avec les patrons de droit d'accès

pelé le noyau du méta-modèle de composants UML 2.0. Nous allons reprendre l'exemple de système GPS pour illustrer l'élaboration de profil UML correspondant à la politique de sécurité retenu (contrôle d'accès) afin de sécurisé l'accès au satellite.

L'élaboration du profile de contrôle d'accès commence par l'étape de mise en correspondance qui identifie les correspondances entre les participants des patrons de sécurité appartenant à l'arbre de décision choisis selon la politique de sécurité retenu et les éléments du méta-modèle de composants.

Les correspondances déduites sont représentées par le tableau 5.2. Ce tableau illustre la relation entre les participants du patron et les éléments du méta-modèle de composants. Par exemple le participant Sujet ainsi que le participant ProtectionObject sont assimilables à l'élément composant du méta-modèle de composants. Une correspondance à été établie aussi entre le participant Right est l'élément connecteur.

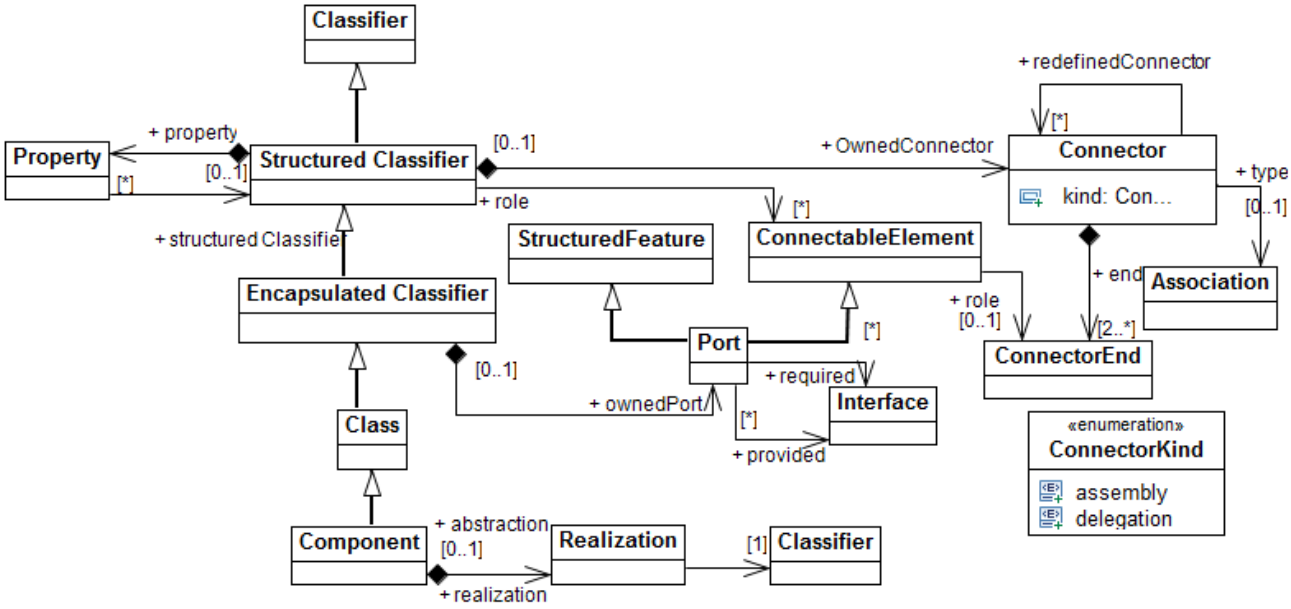


Figure 5.8 — Extrait de méta-modèle de composants UML2.0

Tableau 5.2 — Phase de mise en correspondance

Politique de contrôle d'accès	**Participants des patrons de contrôle d'accès**	**Méta-classes de méta-modèle de composants UML 2.0**
Patron RBAC	Right User Role ProtectionObject	Connecteur Composant Port Composant
Patron d'Autorization Patron Session	Autorization Context Session	Connecteur Port
Patron CheckPoint	CheckPoint	Port
Patron Reference Monitor	Reference Monitor	Port

Le profil obtenu pour la politique de sécurité de contrôle d'accès ou encore pour les patrons appartenant à cette famille est présenté dans la figure 5.9.

Dans le but d'automatiser l'application de ces profils UML de sécurité à des modèles à base de composants nous allons définir des règles permettant la transformation d'un modèle non sécurisé à un modèle sécurisé en d'autre terme en un modèle qui embarque les solutions de sécurité proposées par les patrons de sécurité choisis.

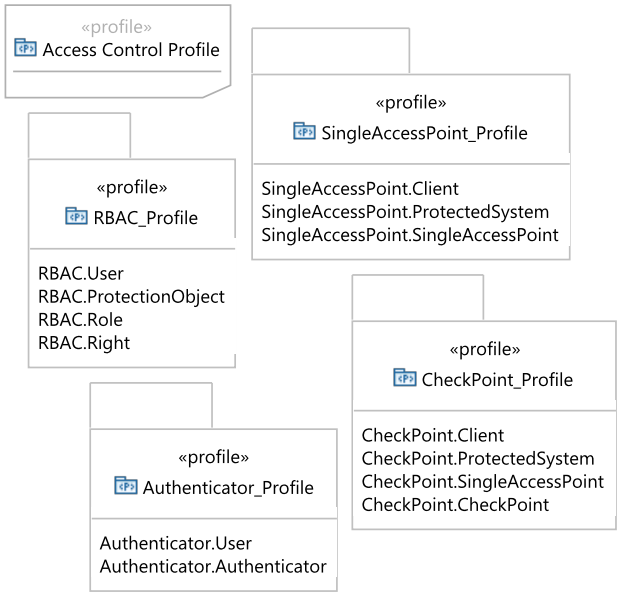


Figure 5.9 — Extrait de profil de la politique de contrôle d'accès

5.7 Règles d'intégration des patrons de sécurité

Cette section présente la méthodologie suivie pour décrire les règles d'intégration des patrons de sécurité. Nous allons aussi illustrer la procédure d'application de ces règles lors de la modélisation d'une application à base de composants dans le but d'intégrer une solution de sécurité proposée par des patrons.

5.7.1 Introduction

Dans les sections précédentes nous avons présenté, dans un première temps une cartographie des patrons de sécurité permettant de structurer différents patrons de sécurité proposés dans la littérature. Cette cartographie permettra d'extraire un arbre de décision de patrons à appliquer une fois la politique de sécurité retenue. Dans un second temps, nous avons proposé d'utiliser la notion de profil UML, cette dernière sera utilisé pour étendre les concepts de méta-modèle à base de composants par les notions de sécurité encapsulées dans les patrons. La dernière étape de ce processus consiste donc à automatiser ou semi automatiser l'application de ces profils de sécurité. Dans cette perspective, un ensemble de règles d'intégration seront définit.

5.7.2 Description des règles d'application des patrons

Nous avons défini un ensemble de règles d'intégration des patrons de sécurité appelé SPARs pour Security Pattern Application Rules. Ces règles vont permettre d'automatiser l'intégration de ces patrons dans les modèles à base de composants.

Méthode de description des SPARs

Dans le but d'intégrer les patrons de sécurité dans un modèle à composants nous utilisons un ensemble de règles. Ces règles permettent une fois exécutées d'annoter le modèle de l'application à base de composants par les concepts de sécurité. Autrement dit le modèle de l'application sera décoré par des stéréotypes qui correspondent aux solutions de sécurité proposées par les patrons appliqués (selon la politique de sécurité choisie).

Pour qu'une règle soit générique et que la transformation dans son ensemble puisse s'appliquer à différents modèles, elle ne référencera pas directement les éléments du modèle source. Autrement dit, l'expression des règles de transformation se fera au niveau des méta-modèles. L'automatisation de cette intégration se décompose en trois phases :

Identifier les paramètres. Il s'agit des paramètres que le développeur doit saisir comme le nom de l'élément dans le modèle de l'application (port, composant, connecteur, etc.) et qui correspond au rôle dans le patron de sécurité à appliquer. Selon la solution proposée par le patron ainsi que le problème pour lequel le patron a été proposé, une ou plusieurs pré-conditions sont nécessaires pour déclencher l'exécution de la transformation. Par exemple pour le patron « Single Access Point » le paramètre d'entrée est la désignation du système ou composant composite à sécuriser. Pour le patron RBAC le paramètre d'entrée est la nomination préalable de l'élément à protéger et de son utilisateur. Il faut noter aussi que la définition de ces paramètres dépend du ou des patrons de sécurité précédemment appliqués sur le modèle de composants.

Déterminer les opérations d'intégration. Dans notre contexte les transformations sont exogènes c'est-à-dire que le méta-modèle source et le méta-modèle cible sont les mêmes.

- La première opération d'intégration consiste en une transformation de type « chaque élément du modèle source, donne lieu à la création d'un élément de même type dans le modèle cible, et prend le nom de l'élément source». Autrement dit le modèle source sera copié dans le modèle cible et les annotations seront ajoutées automatiquement à des éléments du modèle cible via des transformations déclenchées automatiquement une fois le développeur valide les pré-conditions.
- Pour déterminer les règles à appliquer automatiquement au modèle source nous allons nous baser sur les paramètres donnés par le développeur à la première étape. Pour traduire la description des paramètres en règle exprimée en langage naturel, on dit alors plutôt que « chaque élément du modèle source correspondant au paramètre donné par le développeur, donne lieu à l'ajout d'un stéréotype de la forme « Nom du Patron appliqué. paramètre» à l'élément de même type dans le modèle cible, et portant le nom de l'élément source».
- Partant des éléments ainsi annotés, d'autres règles doivent être définies pour faire la correspondance entre les autres participants du patron de sécurité et les éléments du modèle. Cette correspondance sera guidée par le profil UML. Concrètement, cette mise en correspondance est réalisée en appliquant les stéréotypes respectifs définis dans le profil UML correspondant à l'ensemble des patrons de sécurité appliqués.

Exprimer les opérations dans le langage de transformation de modèle. Enfin, une fois la sémantique des règles de transformations extraites en langage naturel, il s'agit d'exploiter les fonctionnalités du langage de transformation de modèle pour décrire des sémantiques

permettant d'appliquer ces règles automatiquement.

Extrait de l'algorithme de description des SPARs

Dans cette partie, nous présentons l'algorithme général qui décrit la suite de règles qui permettent d'intégrer, au niveau modélisation, des patrons de sécurité dans un modèle à base de composant. L'intégration des solutions de sécurité relatives à une politique de sécurité est traduite par l'application du profil UML spécifique à cette politique, et par l'application des stéréotypes définis dans ce profil sur les artefacts du méta-modèle à composant. Ainsi, la principale action effectuée par l'algorithme est la construction du modèle à composant sécurisé. Cette action se décompose en trois tâches qui sont :

1. La copie des éléments du diagramme de composant,
2. L'application du profil UML relié à la politique de sécurité ce qui implique entre autre la possibilité d'appliquer un ou plusieurs patrons de sécurité appartenant à cette même politique,
3. L'application des stéréotypes de sécurité sur les artefacts adéquats du méta-modèle du diagramme de composant.

Avant la réalisation de ces trois tâches, le système vérifie si oui ou non le diagramme de composant proposé par l'utilisateur est valide. Cette vérification s'effectue au cours de l'action de la construction du diagramme de composant PIM (FAIRE : Construire le PIM) où le concepteur doit modéliser un diagramme de composant valide.

Il est à noter que cet algorithme (figure xxxx) n'est pas intégralement automatique, en effet l'utilisateur est régulièrement appelé à prendre des décisions qui interviennent dans le comportement de la transformation. C'est pour quoi, dans le cadre de notre travail, l'application des partons de sécurité sur une architecture à base de composant est dite semi-automatique.

Nous avons jusqu'à présent pu décrire les règles en langage naturel (algorithmique). Une fois, ces règles sont définit ainsi que le méta-modèle cible et source et le modèle de l'application qui doit être conforme à ce méta-modèle UML 2.0, nous pouvons implémenter ces règles en utilisant un langage de transformation de modèle. Dans le cadre de ce travail notre choix s'est porté sur le langage ATL (pointeur vers section ATL).

5.7.3 Processus d'application des règles d'intégration des patrons de sécurité

Les règles d'application sont spécifiées à chacune des arbres de décision en sécurité. Ces arbres sont relatifs à une politique de sécurité bien déterminée. Une fois ces règles définies elles seront ensuite en mesure d'être appliquées. La figure suivante présente les étapes de d'application de ces règles (figure 5.10) :

1. Le développeur modélise son application sous forme d'un diagramme de composants
2. L'application ainsi modélisée représente la première entrée du système
3. Le développeur choisit la politique de sécurité à appliquer. Un arbre de décision sera ensuite extrait dynamiquement de la cartographie présenté dans la section 1.4. Les solu-

Algorithme 5.1 — Extrait de l'algorithme de description des SPARs

Données : Modèle du domaine (PIM)
- Diagramme de composant UML2.0 syntaxiquement correct et sémantiquement satisfaisant conformément au méta-modèle de composants UML 2.0

Résultat : Modèle de l'application sécurisé (secPIM)
- Diagramme de composants UML 2.0, copie du PIM conformément à UML 2.0, annoté avec les stéréotypes représentant la solution proposée par les patrons de sécurité appliqués

```
1  Algorithme (S = Système / C = Concepteur)
2  FAIRE : Construire le PIM S : {
3  Vérifier la validité du modèle à composants conçu par C
4  tant que Le PIM n'est pas valide faire
5  |   Corriger le PIM par C
6  fin
7  C : { Définir les paramètres d'entrée permettant de déclencher les transformations }
   Proposition du concepteur <- Vrai
8  }
9  FAIRE : Vérifier les paramètres donnés par le concepteur
10 C : {
11 - Choisir l'artefact
12 - Choisir le stéréotype pour l'artefact
13 }
14 S : {
15 - Parcourir le profil UML appliqué sur le modèle
16 - Extraire la méta-classe qui étend le stéréotype choisi par C
17 si méta-classe extraite = artefact choisi par C alors
18 |   Retourner Vrai ;
19 sinon
20 |   Retourner Faux ;
21 finsi
22 }
23 FAIRE : Construire le modèle PIM
24 Paramètres : On dispose des paramètre(s) donnés par C
25 S : Appliquer la règle d'application des paramètres ;
26 {
27 tant que proposition du concepteur = Vrai faire
28 |   Copier les éléments du PIM et intégrer la sécurité :
29 |   {
30 |   parcours le modèle PIM
31 |   tant que Il existe des éléments dans PIM faire
32 |   |   copier l'élément dans secPIM :
33 |   |   {
34 |   |   tant que Il existe des caractéristiques dans artefact du PIM faire
35 |   |   |   artefactsecPIM.caractéristique <- artefactPIM.caractéristique
36 |   |   fin
37 |   fin
38 fin
```

tions de sécurité proposées par ces patrons seront présentées sous forme d'un ensemble de règle de transformation SPARs

4. Le concepteur peut appliquer les patrons de sécurité successivement selon les dépendances les reliant. C'est ainsi que les SPARs seront appliqués pour générer un diagramme de composant sécurisé.

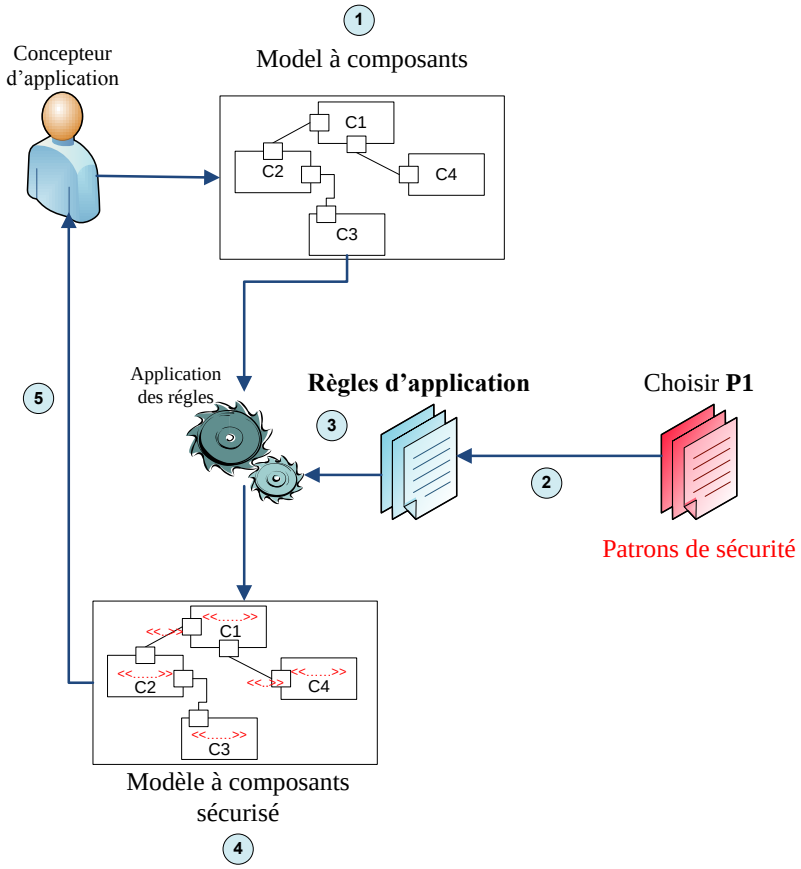


Figure 5.10 — Procédure d'application des règles d'intégration des patrons de sécurité

A la fin de ce processus, nous obtenons un modèle de l'application qui intègre l'ensemble des solutions de sécurité proposées par les patrons sélectionnés. Dans ce qui suit, nous proposons une formalisation de cette démarche sous forme d'un processus décrit selon le standard SPEM

5.8 SCRIP : un processus pour l'intégration des patrons de sécurité dans une architecture à base de composants

Dans la première partie de ce chapitre, nous avons détaillé notre méthodologie d'intégration des patrons de sécurité dans des modèles à base de composants. Nous avons décrit la démarche à suivre pour bénéficier de l'avantage des patrons de sécurité dans un modèle à base de composant. Dans ce chapitre, nous formalisons l'approche proposée en utilisant le standard SPEM (Software Process Engineering Meta-model) [SPEM, 2008]. Ce standard

à été proposé par l'OMG et permet de définir des processus de manière simple et compréhensible pour les utilisateurs non expérimentés. La formalisation de notre approche produit un processus SCRIP permettant de définir les étapes (ou phases), leurs ordonnancement et permet aussi l'expression des modèles et des artefacts à la fois produits et nécessaires. Ce formalisme permettra aux concepteurs d'applications d'effectuer pas-à-pas les étapes décrites dans le processus pour atteindre leurs objectifs.

L'objectif de la formalisation de notre approche en SPEM est de définir les concepts nécessaires pour la modélisation, la documentation, la présentation et l'interchangeabilité de la méthode proposée. Cette formalisation permet entre autres la réutilisation de la méthode. Cette formalisation permettra également le support de la méthode par des outils supportant SPEM.

5.8.1 SPEM : Software and System Process Engi-neering Metamodel

5.8.1.1 Origine et Définition

La première version de la norme SPEM a été introduite par l'OMG en 2002. Sa première version été basé sur le standard UML 1.4. Cette version a été révisée en 2005 et de nouveau en 2007. Des changements majeurs ont conduit à la version SPEM 2.0, qui est compatible avec UML 2. En raison de la conformité avec le standard UML, les diagrammes UML tels que les diagrammes d'activités ou diagrammes états peuvent être utilisé pour visualiser des modèles du processus SPEM.

Le développement du méta-modèle SPEM a été motivé par :
- L'abondance de différents concepts de modélisation des processus décrits dans des formats différents, en utilisant différentes notations
- La violenté d'assurer la cohérence entre ces différentes approches. Ainsi le besoin de standardisation se pose et la norme SPEM a vu le jour.

Une définition de ce standard, qui correspond à nos besoins est présentée dans [SPEM, 2008]

"The Software and Systems Process Engineering Meta-Model (SPEM) is a process engineering metamodel as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes.

SPEM permet de décrire les processus de développement logiciel. Il est structuré comme un profil UML. Le but de SPEM est aussi de permettre la réutilisation des processus et de la documentation.

5.8.1.2 Principaux concepts

Dans cette section, nous allons présenter les principaux concepts du méta-modèle SPEM. Les concepts que nous utilisons pour décrire notre approche seront présentés en détails. Pour le formalisme SPEM, un processus de développement logiciel est une collaboration entre des entités abstraites actifs appelés rôles de processus, qui effectuent des opérations

appelées tâches sur des entités concrètes appelées produits. Cette association entre ces trois concepts peu être décrite par la figure 5.11.

Figure 5.11 — Relations entre les principaux concepts de SPEM

Dans la suite de ce manuscrit et en particulier dans ce chapitre, nous utiliserons la terminologie décrite dans le standard SPEM 2. Cette dernière sera utilisée pour spécifier le processus SCRIP d'intégration des patrons de sécurité dans des modèles à composants. Ce processus est décrit en termes de phases, de rôles et d'étapes. Dans le tableau 5.11, nous présentons les métaclasses du méta-modèle SPEM qui sont utilisées pour décrire notre processus.

Tableau 5.3 — Principaux concepts SPEM

Concepts	**Définitions**	**Icones**
RoleUse	définit les responsabilités plus spécifiques	
TaskUse	Décrit un travail effectué par un RoleUse, qui peut consister en éléments atomiques appelées Step	
WorkProductUse	est un artefact (élément d'information, document, code source du modèle, etc) produite, consommée ou modifiée par un processus	
Step	Un TaskUse peut être composé d'éléments atomiques appelés étapes (steps)	

Nous avons également utilisé les concepts de « phase » hérité de SPEM 1 et qui sont maintenant définis dans le plug-in SPEM2.0. Un processus est composé d'une séquence de phases ( ), chaque phase comportant une ou plusieurs itérations.

5.8.1.3 Pourquoi modéliser le processus SCRIP en SPEM ?

SPEM propose deux types de description de processus :

– Représentation textuelle, en utilisant une structure hiérarchique entre les concepts du SPEM
– Représentation graphique plus parlante en utilisant des icones relatives à chaque concept de SPEM.

Comme présenter précédemment, les notations utilisées dans SPEM sont assez claires et efficaces pour la diffusion de la méthode proposée. Ce formalisme permet de définir des processus de manière simple et compréhensible Comme présenter dans la section sur les travaux connexes, un grand nombre de méthodes et outils d'intégration des patrons ont été proposés. Cependant, ces méthodes ne sont pas assez utilisées par les concepteurs et les développeurs car ils manquent de supports clairs et de documentation qui les rend simple à utiliser pour un non expert. Ce qui peut être, en partie, un des freins à (i) l'utilisation même de ces méthode et (ii) l'utilisation des patrons et particulièrement les patrons de sécurité par les non-spécialistes.

5.8.2 SCRIP : SeCurity patteRn Integration Process

Nous présentons ci-dessous une vue d'ensemble du processus SCRIP. Les sections suivantes donnent une description détaillée des trois phases que nous considérons dans notre processus. Par ailleurs, pour compléter la notation habituelle graphique, nous proposons une notation textuelle-conforme à la spécification SPEM-pour décrire une structure de processus, comme illustrer par la figure 5.12.

```
Process :
    Activity Kind : Phase :

        Activity kind : Iteration : First
          TaskUse :
            ProcessPerformer kind : primary
              RoleUse :
            WorkDefinitionParameter kind : in
              WorkProductUse :
            WorkDefinitionParameter kind : out
              WorkProductUse :
            Steps
              Step :
              Step :
```

Figure 5.12 — Description textuelle de la structure d'un processus SPEM

Le processus d'intégration de patron de sécurité (SCRIP) est composé d'une séquence de phases qui couvrent l'ensemble du cycle de vie d'une application à base de composant, de la conception à la mise en œuvre. Ce processus a été décrit avec SPEM et détaillé dans la figure 5.13, qui illustre les phases, la séquence des utilisations des tâches "TaskUse", ainsi que l'ensemble des produits consommé par chaque TaskUse. Dans ce qui suit, nous décrivons en détail les différentes phases qui constituent le processus SCRIP à savoir :

– La phase de déclenchement,

- La phase de modélisation
- La phase d'implémentation.

Vue d'ensemble du processus SCRIP

SCRIP est structuré en trois phases consécutives : La phase de déclenchement, de modélisation et d'implémentation. Le style itératif doit être appliquée à toutes les étapes du processus. Quatre rôles ont été identifiés dans ce processus :

- Spécialiste de la sécurité : Qui est censé (dans notre contexte applicatif) avoir quelques connaissances en ingénierie à base de composants.
- Concepteur de logiciel : Qui a également besoin d'avoir une expertise minimale dans le domaine des solutions de sécurité.
- Model2Model transformateur : Il s'agit dans notre cas d'un outil logiciel qui applique automatiquement les règles d'application de patron de sécurité.
- Weaver : Il s'agit aussi d'un outil logiciel qui, à travers un code d'application et un code fonctionnel aspect en entrée, à la capacité de délivrer un code unique sécurisé en sortie.

Le processus SCRIP commence par la phase de déclenchement. Les deux WorkProductUse «patron de sécurité» et «méta-modèle à composant» sont censés être disponibles comme entrées à cette phase. WorkProductUse «patron de sécurité» contient la spécification et la conception de solutions spécifiques de sécurité. Ce même WorkProductUse sera utilisé dans les deux TaskUse T1.1 et T1.2 pour obtenir deux sorties : (1) «profil de sécurité» qui étend le méta-modèle à composante avec de nouveaux concepts du patron de sécurité, (2) «Règles d'intégration des patrons de sécurité (SPERs [1])» qui sont un ensemble de règles exprimées dans un langage de transformation comme le langage ATL par exemple [3].

La phase de modélisation comprend deux TaskUse : T2.1 et T2.2. TaskUse T2.1 prend en entrée la WorkProductUse «méta-modèle à composante» et produit un «modèle de l'application» instance du méta-modèle à composant, tandis que TaskUse T2.2 prend en entrée le WorkProductUse «modèle de l'application» obtenue à partir de TaskUse T2.1 et le WorkProductUse «Règles d'intégration des patrons de sécurité (SPERs)» obtenue à partir de la phase précédente pour produire un «modèle d'application sécurisé».

Enfin, la phase de l'implémentation est effectuée à travers les TaskUse T3.1 et T3.3, T3.2. TaskUse T3.1 nécessite en entrée le WorkProductUse «modèle d'application sécurisé» obtenu à partir T2.2 et produit le «code fonctionnel de l'application», tandis que TaskUse T3.2 génère le «code de l'aspect». T3.3 prend comme entrées le WorkProductUse «code fonctionnel de l'application» et «code de l'aspect» et produit le «code de l'application sécurisé».

Dans ce qui suit, nous détaillons les trois phases du processus SCRIP et les TaskUse qui le constituent. Nous tenant à préciser que le processus SCRIP est un processus itératif, et que dans cette illustration, nous allons nous limité à une seul itération (la première itération).

1. SPERs : Security Pattern Application Rules

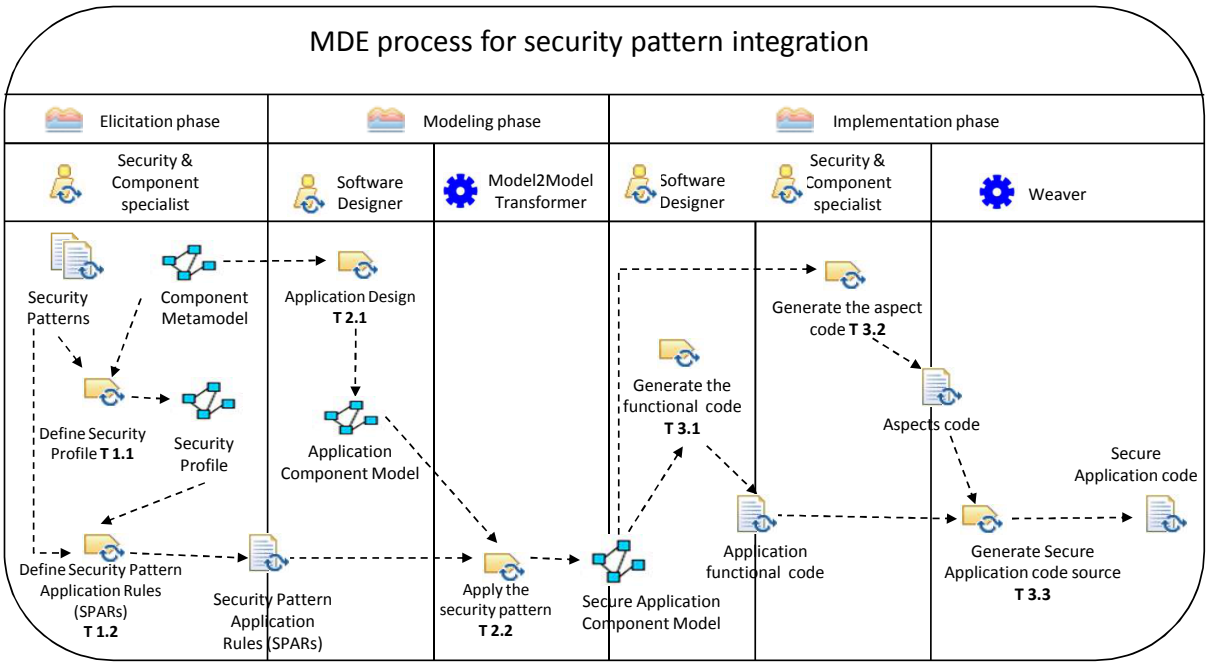


Figure 5.13 — Diagramme structurel décrivant le processus SCRIP d'intégration des patron de sécurité en SPEM2

Phase de déclenchement

Comme décrit ci-dessus, la phase de déclenchement comprend deux TaskUse : TaskUses T1.1 Définir un profil de sécurité, qui prend en entrée les WorkProductUse «patrons de sécurité» et «méta-modèle de composant», aussi TaskUse T1.2 Définir des règles d'intégration des patrons de sécurité. Cette TaskUse prend en entrée les sorties de TaskUse T1.1, c'est à dire, le «profil de sécurité» ainsi que le WorkProductUse «patron de sécurité». Afin de mener à bien ces deux TaskUse, plusieurs étapes sont effectuées au sein de chaque TaskUse, comme on peut le voir dans la figure 5.13.

l'activité : Définir le profil de sécurité TaskUse T1.1 est effectuée par le RoleUse «spécialiste en composant et en sécurité». Pour débuter, cette TaskUse à besoin de deux entrées. C'est inputs sont «méta-modèle à composant» et «patron de sécurité». Le méta-modèle de composant permet de définir les concepts de base de modèle et des applications à base de composants. Tant dis que les patrons de sécurité permettent de définir et collecter les solutions de sécurité d'une manière structurée. Plusieurs étapes sont nécessaires pour produire le WorkProduct. Le résultat de cette TaskUse le «profil de sécurité».

La définition du profil de sécurité consiste à faire une correspondance entre les concepts du patron de sécurité choisi avec les concepts du méta-modèle de composant. Nous présentant par la figure 5.14 les étapes de cette TaskUse en utilisant la représentation proposée dans le standard SPEM2.0.

Comme le montre la figure 5.15, description de TaskUse en utilisant la notation textuelle SPEM2.0, cette TaskUse fait partie de la phase de déclenchement du processus SCRIP.

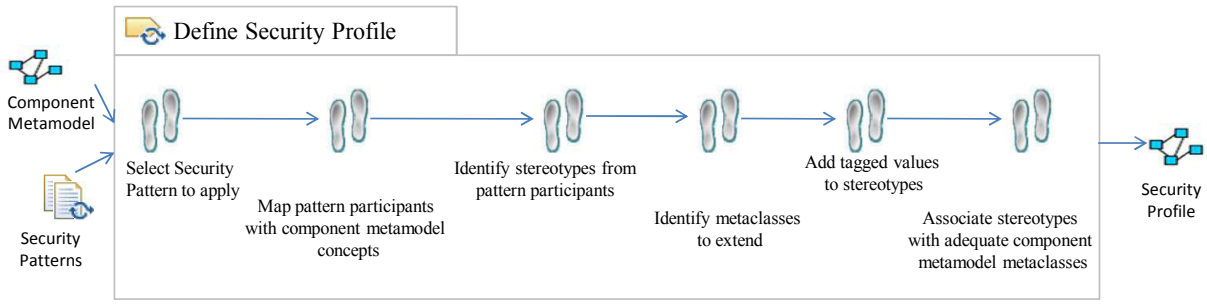


Figure 5.14 — Description de TaskUse : définition du profil de sécurité en SPEM 2

Dans la figure 6, les mèta-classes du méta-modèle SPEM, les mèta-associations et les mèta-attributs sont représentés en police normale tandis que les instances correspondantes apparaissent en caractères **gras**.

Process : **SCRIP**
Activity Kind : Phase : **Eliciting**

Activity kind : Iteration : **First iteration**
TaskUse : **Define Security Profile**
ProcessPerformer kind : primary
RoleUse : **Security and Component Specialist**
WorkDefinitionParameter kind : in
WorkProductUse : **Component Metamodel**
WorkProductUse : **Security Pattern**
WorkDefinitionParameter kind : out
WorkProductUse : **Security Profile**
Steps
Step : **Select a security pattern to apply**
Step : **Map pattern participants with component metamodel concepts**
Step : **Identify stereotypes from pattern participants**
Step : **Identify metaclass to extend**
Step : **Add tagged values to stereotypes**
Step : **Associate stereotypes with adequate UML metaclasses**

Figure 5.15 — Description de TaskUse : Définir le profil de sécurité

L'Activité : Définir des règles d'application des patrons de sécurité La figure 5.16 représente TaskUse de définition des règles d'intégration des patrons de sécurité, qui est effectuée par le RoleUse «spécialiste en composant et en sécurité».

Nous avons défini un ensemble de règles (SPARs) pour automatiser l'intégration des patrons de sécurité dans une application à base de composants. Ces règles sont déduites par les relations entre les concepts de sécurité du patron sélectionné et le profil UML correspondant. Ces règles sont appliquées en deux étapes :

– La première étape consiste à assurer la correspondance entre les concepts de patron et les principaux éléments du modèle en correspondance (défini comme un composant, un lien ou un port). Pour chaque patron de sécurité, nous déterminons le concept

principal qui doit être appliquée par le concepteur (ie, le nom de l'artefact dans l'application qui correspond au rôle dans le patron de sécurité appliquée). La définition de cette correspondance dépend des patrons de sécurité précédemment appliqués sur le même modèle.

– La seconde étape correspond à la mise en correspondance des autres concepts du patron de sécurité aux éléments de modèle correspondant. Concrètement, cette mise en correspondance est réalisée en appliquant les stéréotypes respectifs définis dans le profil UML correspondant.

Process : **SCRIP**
Activity Kind : Phase : **Elicitation**

Activity kind : Iteration : **First iteration**
TaskUse : **Define Security Patterns Application Rules**
ProcessPerformer kind : primary
RoleUse : **Security and Component Specialist**
WorkDefinitionParameter kind : in
WorkProductUse : **Security Profile**
WorkProductUse : **Security Pattern**
WorkDefinitionParameter kind : out
WorkProductUse : **Security Patterns Application Rules**
Steps
Step : **Define parameter(s) that software developer should input**
Step : **Map a set of concepts among security participants**
Step : **Identify stereotypes from pattern participants**
Step : **Look form common roles between the to-be-applied pattern and the application concepts**

Figure 5.16 — Description de TaskUse : Définir Pattern Application Security règles (T1.2)

Nous avons mis en œuvre cette mise en correspondance comme une transformation modèle-à-modèle en utilisant le langage ATL. LA figure présente un extrait de SPARs du patrons RBAC .

Extrait de SPARs du patrons RBAC

```
helperdef : getStereotype(p : UML2!Profile, name : String) :
UML2!Stereotype =
p.ownedStereotype->select(s | s.name=name)
->first() ;
rule Package ....
do
t.applyProfile(UML2!Package.allInstances()
->select(s | s.name='RBACProfile')
->first()) ; thisModule.entityProfile<UML2!Package.allInstances()
->select(s | s.name=' RBACProfile ')
->first() ;
rule Component ....
do
if(s.name='SecureSattelite')
t.applyStereotype(thisModule.getStereotype(thisModule.
entityProfile,'ProtectionObject)) ;
rule Port
do
if(s.clientDependency.isEmpty()
ands.owner.name='GPSTerminal')
:t.applyStereotype(thisModule.getStereotype(thisModule.entityProfile,'Role))
```

Phase de modélisation

Cette phase produit un modèle de composant sécurisé après avoir appliqué un ou plusieurs patron de sécurité à un modèle de composant initiale non sécurisé. Cette phase de modélisation est effectuée au moyen de deux TaskUse : la modélisation de l'application (T 2.1) et l'application du patron de sécurité (T 2.2).

Activité Modélisation de l'application à base de composants Le but de TaskUse T2.1 est de modéliser l'application réel à base de composants. Le RoleUse «concepteur de logiciels» est responsable de cette TaskUse. Le concepteur peut utiliser l'outil Papyrus [UML, b], par exemple, pour modéliser son application à l'aide du diagramme de composants UML2.0. Il peut également utiliser n'importe quel profil UML qui supporte des modèles de composants spécifiques tels que CCM, EJB ou fractale. Le modèle de composant qui en résulte ne prend pas en charge les concepts de sécurité.

Activité Appliquer le patron de sécurité TaskUse T2.2 est réalisée pour obtenir un modèle d'application sécurisée à base de composant. Cette TaskUse prend en entrée deux WorkProducts : «Modèle d'application» et «Règles d'intégration des patron de sécurité». Le RoleUse

«Model2Model transformateur» exécute TaskUse. le datail de cette étape est décrit par la figure 5.17.

Une fois les étapes TaskUse 2.2 ont été exécutées, nous obtenons en sortie un WorkProduct modèle de composant sécurisé.

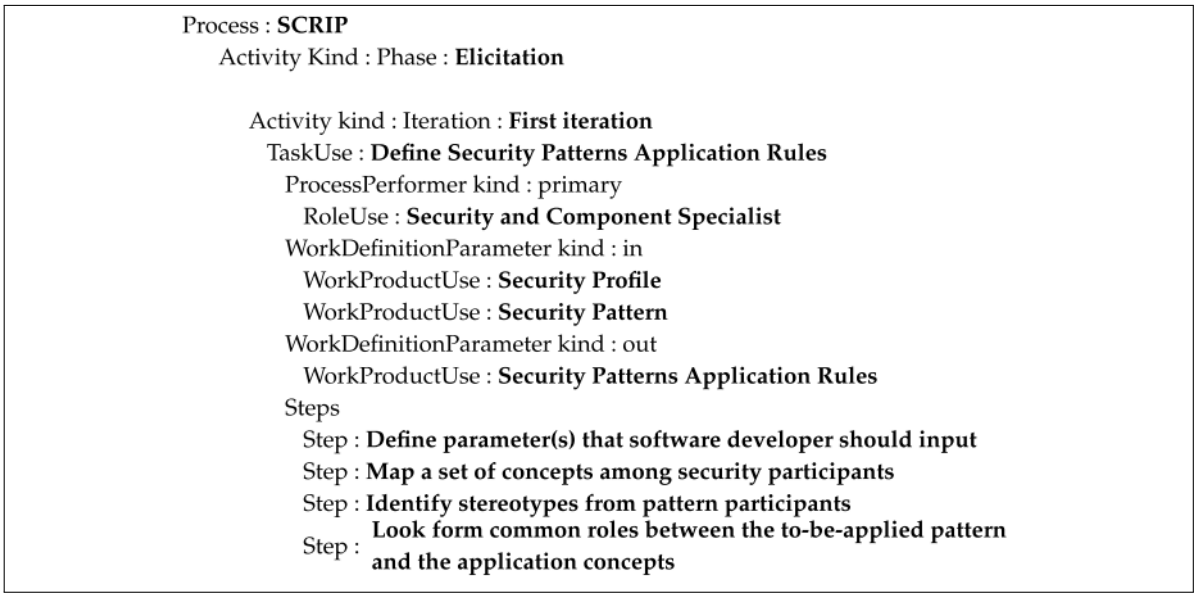


Figure 5.17 — Appliquer du patron de sécurité (T2.2)

5.8.2.1 Phase d'implémentation

Cette phase est consacrée à produire un code d'application sécurisé grâce à trois TaskUse. T 3.1, T 3.2 et T3.3. Cette phase d'implémentation génère deux objets intermédiaires : le «code fonctionnel» de l'application à base de composants et le «code de l'aspect» qui encapsule le code de la solution de sécurité proposé par le patron. Pendant cette phase le RoleUse «Spécialistes en composant et en sécurité» et le RoleUse «concepteur de logiciels» coopèrent pour définir le code de l'application finale sécurisé comme il est expliqué dans les TaskUse suivantes.

Activité Générer le code fonctionnel TaskUse T3.1 a pour objectif de générer le «code fonctionnel» de l'application à base de composants. Nous réutilisons les approches existantes pour la génération de code fonctionnel. En effet plusieurs approches et d'outils commerciaux (comme Rational Rose [IBM] et Enterprise Architect [Architect]) supporte la production de squelette de code correspondant au différentes technologies (EJB,. NET, C + +, etc) à partir d'un diagramme de composants UML et en se basant sur un ensemble de bibliothèques prédéfinies. Le concepteur peut également produire le code correspondant en utilisant par exemple l'approche MDA. Il transforme d'abord le modèle de l'application à base de composant à un modèle spécifique à une plate-forme. Le code correspondant est alors généré en utilisant le générateur de modèle au texte.

Activité Générer le code aspect TaskUse T3.2 prend en entrée le modèle d'application sécurisé pour définir les aspects. Au cours de cette TaskUse, le RoleUse «spécialiste en composant et en sécurité» et le «concepteur de logiciel» collaborent en vue de générer le «code aspect». Pour chaque patron de sécurité, nous proposons un modèle (Template) pour générer du code AspectJ à l'aide d'un ensemble de classes Java. Nous générons une squelette de code aspect qui doit être rempli par le développeur, selon le code de l'application fonctionnelle générée lors de TaskUse T3.1.

Activité Générer du code d'application sécurisée Enfin, TaskUse T3.3 prend en entrée le WorkProducts «code de l'application fonctionnelle» et le WorkProducts «Code aspects» pour produire un «code d'application sécurisée». Ce dernier est obtenu par tissage du «Code aspects» résultant de TaskUse 3.2 avec le code de l'application fonctionnelle obtenue à partir de l'TaskUses T3.1. Cette TaskUse est assuré par le RoleUse «weaver».

5.9 Conclusion

Dans ce chapitre, nous avons introduit et décrit l'approche proposée pour l'intégration des patrons de sécurité dans une architecture à base de composants. Cette intégration passe par la production de profils UML produits à partir des patrons de sécurité. Cette représentation nous permet d'étendre le méta-modèle de composant UML 2.0 avec les concepts de sécurité du patron et nous permet aussi de faire la correspondance entre les concepts de ce méta-modèle et ceux des patrons de sécurité. Dans le cadre de ce chapitre nous avons aussi présenté la méthode de description et d'application des règles d'intégration des patrons de sécurité. Ces règles d'intégration permettent d'appliquer d'une façon semi-automatique les solutions de sécurité.

Aussi dans ce même chapitre, nous avons proposé une formalisation de cette démarche sous forme d'un processus décrit selon le standard SPEM. Nous avons présenté en premier lieu un aperçu du méta-modèle SPEM que nous avons utilisé pour représenter le processus SCRIP. Ce choix s'explique par l'adéquation de se standard avec le processus présenté précédemment. En deuxième lieu, nous avons fourni une description détaillée du processus SCRIP, allant de la conception jusqu'à l'implémentation. Le processus décrit est composé de trois phases principales : La phase d'Elicitation, la phase de modélisation et la phase d'implémentation. Ces phases comprennent différentes TaskUse qui permettent de une application des patrons de sécurité au cours du cycle de développement d'une application à base de composants en utilisant le processus SCRIP.

Le chapitre suivant est consacré à la validation de notre approche. Cette validation a été effectuée tout d'abord à travers le développement sous Eclipse d'un outil support à l'intégration des patrons de sécurité. En second lieu, nous avons appliqué notre approche sur l'étude de cas de systéme de soin présentée.

Troisième partie

Réalisation et cas d'étude

6 Mise en œuvre de l'approche et application à un cas d'étude

6.1 Introduction

Comme vu dans le chapitre précédent, nous avons présenté une approche permettant l'intégration de la sécurité dans les modèles à bases de composants en utilisant les patrons de sécurité. Ce chapitre porte sur l'implémentation de l'approche sous la forme d'un prototype appelé SCRI-TOOL ainsi que sur l'évaluation de cette approche sur un cas d'étude inspiré d'un système réel.

6.2 L'environnement de développement

L'implémentation du prototype a été réalisée sous la plate-forme de développement Eclipse qui est considérée comme l'incubateur des projets de développement la plus sollicitée par la communauté IDM. [Eclipse] Ce choix a été aussi motivé par le fait que cette plateforme est libre et open source. Ce choix est aussi justifié par l'existence d'un grand nombre d'outils et de technologies autour de l'IDM tels que les API graphiques et les technologies de gestion de modèles comme EMF (Eclipse Modeling Framework). Ce framework nous a permis d'implémenter le prototype SCRI-TOOL permettant l'intégration des patrons de sécurité. Au niveau du langage de transformation, notre choix s'est porté sur le langage de transformation ATL qui est considéré maintenant comme un standard de transformation dans Eclipse et est intégré depuis 2007 dans le projet M2M (voir section 3.2.4.3 du chapitre 3).

6.2.1 L'environnement Eclipse

Eclipse est une plate-forme qui offre un environnement de développement intégré qui fournit un environnement modulaire pour permettre de réaliser facilement des développements informatiques [Eclipse]. Le développement de nouvelles fonctionnalités se fait grâce à la notion de modules supplémentaires appelés plug-ins. Ce concept permet de fournir un mécanisme pour l'extension de la plate-forme et offre ainsi la possibilité à des tiers de développer des fonctionnalités qui ne sont pas fournies en standard par Eclipse. Eclipse utilise intensément les plug-ins dans son architecture puisqu'en dehors du « Runtime », tout le

reste est développé sous la forme de plug-ins (Figure 6.1).

Le développement de la plate-forme Eclipse est supervisé par le consortium d'Eclipse Eclipse.org composé d'une quarantaine de membres dont IBM, Borland, Oracle, SAP, Telelogic et l'OMG.

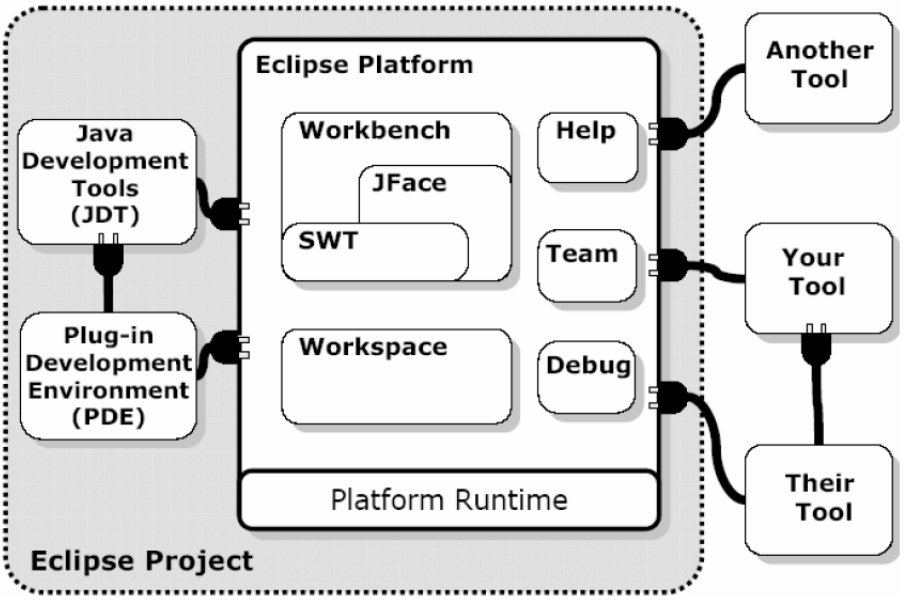


Figure 6.1 — Architecture générale de la plate-forme Eclipse [Griffin, 2004]

La Figure 1 illustre l'architecture générale d'Eclipse. La plate-forme définit un ensemble de frameworks et de services communs pour assurer une interopérabilité des outils ajoutés. C'est l'équivalent d'un noyau pour un système d'exploitation. Le workbench comprend un système de gestion concurrente de ressources distribuées (Versioning and Configuration Management), une gestion de projets, une gestion des versions, une infrastructure de débogage indépendante des langages de développement, un framework d'extension des fonctionnalités, une bibliothèque graphique portable (StandardWidgetToolkit, SWT). Le workspace comprend un espace de stockage des ressources manipulées (.java,.class,.xml,etc.),un framework pour une interface graphique portable (JFace),un système générique d'aide, de recherche, de comparaison et de mise à jour des plates-formes, des parseurs, le support de scripts, etc.

Après avoir présenté brièvement Eclipse et son architecture de base, nous allons voir dans les sections qui suivent comment cette plate forme peut être utilisée pour réaliser des opérations de gestion de modèles comme l'édition ou la transformation.

6.2.1.1 La transformation de modèles dans Eclipse

La plate-forme Eclipse étant implémentée en Java, elle possède par défaut les outils de développement Java. En général,les différentes opérations de transformation d'un artefact logiciel sont gérées par ces outils. Les modèles de conception sont transformés en premier lieu en code source, puis les fichiers de code source sont transformés en fichiers binaires qui sont transformés à leur tour en modules installables. À chaque étape de ce processus

de transformation, les outils fournissent les moyens de visualisation et de manipulation des artefacts utilisés.

Grâce au mécanisme d'extension d'Eclipse, les transformations peuvent être identifiées comme des composants indépendants et réutilisables [Griffin, 2004].Ceci est possible en incluant un point d'extension pour définir les transformations, ainsi qu'une interface logicielle pour permettre l'invocation et l'interrogation des transformations disponibles. Avec ces extensions, la plate-forme Eclipse peut être dotée de nouvelles transformations qui pourront être utilisées directement par des utilisateurs ou par programmation dans des outils. Ce mécanisme sera utilisé afin de gérer les transformations proposées dans notre approche.

Nous avons présenté dans les sections précédentes les fonctionnalités de base liées à la manipulation de modèles dans la plate-forme Eclipse. Nous allons voir dans la suite comment ces fonctionnalités peuvent être étendues et utilisées afin de créer un environnement générique d'intégration des patrons. Dans cette perspective, nous commençons par présenter la plate-forme UML2. En effet, et comme présenté dans la partie 2, nous allons illustrer l'approche à travers des modèles de composant UML 2.0.

6.2.1.2 La plate-forme UML2

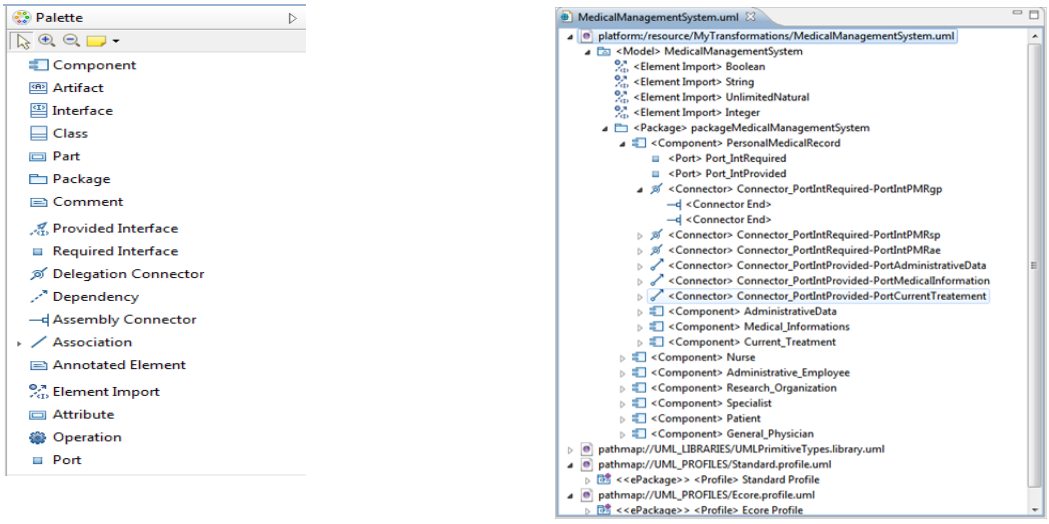
Pour modéliser l'application à base de composants à sécuriser, nous avons utilisé le plug-in UML2 d'Eclipse. Cette implémentation est basée sur le méta-méta-modèle EMOF. Ce dernier est un langage de méta-modélisation qui fait partie d'EMF (Eclipse Modeling Framework), résultat des efforts du projet Eclipse (Eclipse Tools Project). Ecore ressemble dans sa structure à un diagramme de classe UML. Il est basé sur des classes, des attributs, des associations pour lier les classes, des généralisations / spécialisations entre classes, etc. Ecore permet la réalisation de méta-modèles pour la transformation de modèles. Cette plate-forme utilise le Framework EMF pour accéder à l'implémentation du méta-modèle standard UML 2.0, et permet de créer des modèles à composants UML correspondant à ce méta-modèle ainsi que des profils UML correspondant à ce même méta-modèle.

Cette plate-forme est un environnement Eclipse pour la gestion de modèles UML qui fait partie du projet Eclipse MDT pour Model Development Tools. Cette plate-forme permet, moyennant une palette graphique qui comporte la plupart des symboles de composants, de concevoir des applications conformes au méta-modèle UML 2.0. La figure 2(a)présente un aperçu de cette palette.

Pour le concepteur, la représentation du modèle à composants est donc réalisée à l'aide de cette plateforme. Pour la suite du processus d'intégration, cette phase de modélisation doit impérativement respecter le méta-modèle UML 2.0 (défini dans le premier chapitre).

La plate-forme UML2 permet de générer une définition UML à partir de l'application conçue graphiquement. La figure 2(b) illustre un exemple d'arborescence UML à partir d'un modèle graphique. Cet exemple porte sur la modélisation d'une application à base de composants pour la gestion des dossiers médicaux. Dans cette représentation, différents composants (médecin, infirmière, etc.) sont inter-connectés entre eux via des connecteurs reliant leurs ports respectifs.

Le modèle à composants étant représenté, nous pouvons réaliser la première phase du



(a) Palette du plug-in UML2 (b) Représentation arborescente d'un diagramme de composants

processus qui consiste à faire des transformations de modèle afin d'intégrer les patrons de sécurité.

6.2.2 L'outil ATL

La difficulté rencontrée lors de la mise en place de notre atelier a été le choix d'un langage de transformation qui permette de répondre à nos besoins. En effet, la création d'un langage de transformation normalisé est devenue un point important autour de l'approche MDA [Jouault et Kurtev, 2006b] [Kleppe *et al.*, 2003b] et [Kleppe Anneke, 2002]. Les langages de transformation peuvent être classés suivant plusieurs critères comme la nature des règles (impératives, déclaratives ou hybrides), la gestion des modèles (textuels ou graphiques), la gestion de la traçabilité des transformations, le fait que l'outil soit libre ou pas, etc [Touzi, 2007]. Il est alors possible qu'un langage de transformation soit plus adapté qu'un autre dans un contexte spécifique. En conclusion, il apparaît difficile de converger vers un langage unique. Dans notre travail, nous avons défini deux critères, sur lesquels nous nous sommes basés, et qui nous semblent pertinents, liés aux transformations de modèles que nous souhaitons établir :

- La compatibilité avec la plate-forme utilisée
- La possibilité de gérer des modèles ou des profils UML

ATL semble répondre à ces deux critères. En plus d'être un langage libre, les règles ATL sont simples à mettre en œuvre. Son langage à base de variables et d'affectation le rend accessible. De plus, le langage ATL est défini dans un IDE d'Eclipse, et donc permet de générer des modèles UML (moyennant le méta-modèle d'UML) qui sont compatibles et utilisables dans la plateforme UML2 d'Eclipse, utilisé aussi dans le cadre de ce travail. Enfin, un autre avantage d'ATL, est qu'il supporte les profils UML. En effet, ce langage permet d'introduire efficacement un profil UML (réalisé avec un éditeur UML) en entrée de la transformation (avec le fichier source de la transformation). Des instructions particulières d'ATL permettent

par la suite d'utiliser le profil UML pour typer les éléments du modèle UML en sortie de la transformation.

Dans cette section, nous allons nous limiter à présenter uniquement les caractéristiques de l'outil ATL, qui nous ont été utiles dans l'implémentation de notre atelier. Une des caractéristiques du langage ATL, relève de son langage qui présente la particularité d'être hybride (déclaratif et impératif).

La partie déclarative permet de faire correspondre directement un élément du méta-modèle source de la transformation avec un élément du méta-modèle cible de la transformation (matched rule). L'exemple ATL suivant montre une transformation complètement déclarative de UML vers Java qui à toute classe UML fait correspondre une classe Java :

```
1 module UML2JAVA ;
2 create OUT : JAVA from IN : UML ;
3 rule Class2Jclass
4 {
5  from uclass : UML !Class
6 to jclass : JAVA !JClass(
7 uclass.name <- jclass.name
8  ) }
```

En ATL une transformation s'appelle module. Le mot-clé OUT désigne le méta-modèle cible JAVA (ligne 2). Le mot clé IN désigne le méta-modèle source (UML) (ligne 2). La règle (Class2Jclass) du fichier de transformation (UML2JAVA) est déclarative. L'élément class (méta-classe qui décrit une classe UML) du méta-modèle source va être traduit en l'élément Jclass du méta-modèle cible (ligne 6). On précise ensuite en utilisant l'opérateur de liaison <– , que l'attribut « name » de la nouvelle classe va être égal au nom de la classe UML source (ligne 7).

La partie impérative d'ATL permet de décrire les correspondances directes entre éléments. Elle comporte des déclarations conditionnelles (if, then, else...endif), des déclarations de variables (let VarName : varType = initialValue), des déclarations de boucle (while (condition...do), etc. Cette partie permet également de manipuler les éléments générés par les règles déclaratives (modification d'attributs, etc.).

ATL supporte aussi le mécanisme des Helpers. Ce sont simplement des mots-clés faisant référence à des procédures stockées (qui peuvent ainsi être appelées dans une règle lorsqu'il y en a besoin). Les helpers servent à éviter la redondance de code et la création de grandes expressions dans une règle. Ceci conduit aussi à une meilleure lisibilité des modules ATL. Un helper en ATL peut recevoir des paramètres et retourner une valeur.

Les règles appelées (called rule) ressemblent aux helpers. L'exécution d'une règle appelée est déclenchée par son appel explicite depuis une autre règle. Ces règles s'exécutent pour générer des éléments dans le modèle cible après une première génération assurée par les règles déclaratives. Ces dernières s'exécutent en effet en même temps lors de l'exécution de la transformation.

6.2.2.1 Gestion de méta-modèles avec ATL

ATL permet la gestion des méta-modèles liés à la transformation. Il gère notamment les méta-modèles écrit sous Ecore. Dans ce travail, nous nous intéressons au méta-modèle UML 2.0 comme seule méta-modèle de la transformation. Dans ce cas, la transformation est dite «endogène » car les méta-modèles source et cible sont identiques comme le montre la figure 6.2.

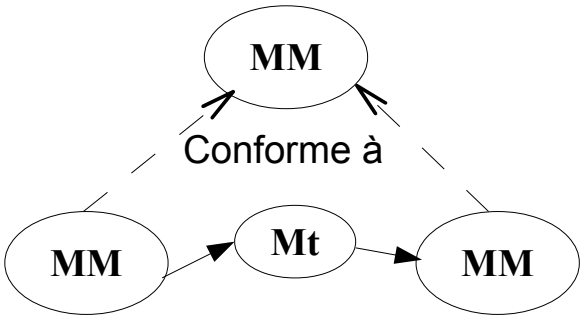


Figure 6.2 — Principe de la transformation endogène

Une fois les deux méta-modèles de la transformation source et cible réalisés (dans notre cas il s'agit du même), il faut les déclarer en utilisant une fenêtre de configuration de la transformation. La figure 6.3 illustre l'utilisation de la fenêtre de configuration pour l'initialisation de la transformation « Security_Pattern_Integration PIM2secPIM ». Les champs permettent de faire correspondre les modèles (in et out), le méta-modèle ainsi que le profil UML de la transformation à leurs déclarations dans le code ATL ainsi que leurs chemins d'accès.

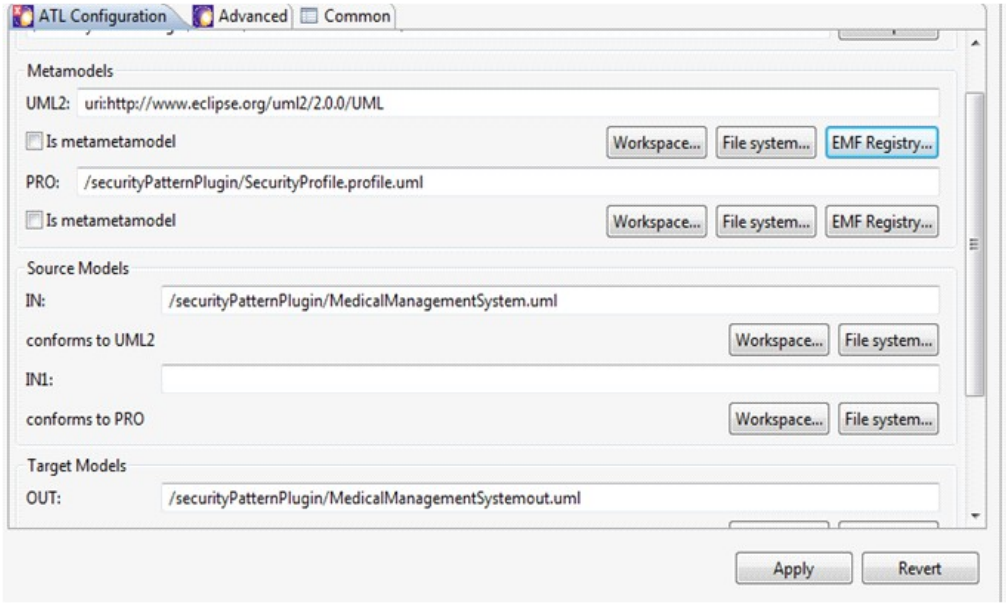


Figure 6.3 — Initialisation d'une transformation avec ATL

6.2.2.2 Gestion des profils UML avec ATL

Parmi les avantages de l'utilisation d'ATL, on peut citer sa capacité à gérer les profils UML. Dans ATL, un profil UML se présente comme un fichier créé par un éditeur UML (Plug-in UML2, etc.) et portant l'extension (.profile.uml). En ATL, nous pouvons ainsi utiliser un fichier profil UML conjointement avec le fichier du méta-modèle cible, qui doit être celui d'UML. La déclaration du profil UML se fait en même temps que la déclaration des méta-modèles de la transformation (voir figure 6.3) (ligne 2) comme suit :

```
10 create OUT : UML2 from IN:Meta-modèle source, NomDeProfile : UML2;
11 ...
```

L'application du profil ne peut commencer qu'une fois la partie déclarative du fichier de transformation exécutée. Elle se fait dans la partie impérative limitée par l'instruction do ? (ligne 3) : la commande prédéfinie dans ATL applyProfile permet d'appliquer le profil UML sur le modèle UML généré (ligne 5) :

```
13 -- apply the profile to the model created
14 out.applyProfile(UML2!Profile.allInstances()->select(e | e.name =
       nomDeProfile?).
15 asSequence().first());
16 ...
17 }
```

Enfin, la commande applyStereotype permet d'appliquer un stéréotype de profil sur un artefact UML généré au sein du modèle cible (ligne 9) :

```
19 nomDeLaClasseUML.applyStereotype(nomDeLaClasseUML.
20 getApplicableStereotype('nomDeStereotype'));
21 ?
```

6.2.3 Le générateur de code Acceleo

Acceleo [Acceleo] est un générateur de code. Il permet d'exploiter les données contenues dans un modèle pour générer le code d'une application. La syntaxe mise au point par Acceleo est intuitive, extensible et dédiée à la génération de code. Un modèle en entrée d'Acceleo est considéré comme un arbre de données que l'on parcourt efficacement avec l'éditeur Acceleo, puis on spécifie le code à générer grâce à la notion de template. Acceleo est simple à utiliser car il dispose de générateurs prêt à l'emploi (JEE, .Net, PHP...) et d'éditeurs de templates de génération sous Eclipse.

Le principe de fonctionnement d'Acceleo est assez basique. La réalisation d'un nouveau module de génération passe par la création de templates (voir figure 6.4 ci-dessous).

Un template de génération de code doit délimiter de façon claire le texte statique généré et les éléments variables. Le texte variable commence par les symboles "<Un template possède la structuration décrite dans le code suivant :

```
1 <%metamodel http :
2  <%script type="uml.model" name=?"model" file="<%name%>.uml"%>
```

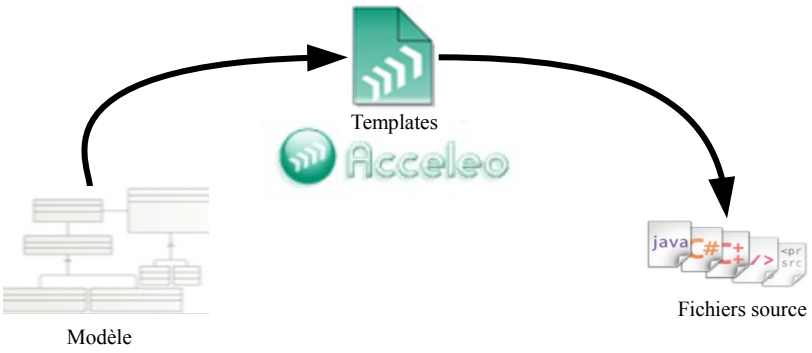


Figure 6.4 — Principe de génération d'application par Acceleo

La première ligne d'un template définit sur quel méta-modèle ce template s'applique. Dans cette zone sont également réalisés des "imports" pour appeler d'autres scripts ou des services Java. La deuxième ligne d'un template est divisée en un certain nombre de scripts. Le script est l'unité élémentaire d'un template. Il évalue un élément pour produire du texte. Par convention, la balise script contenant l'attribut "file" correspond au point d'entrée du template. L'attribut "type" permet, quant à lui, de spécifier sur quel type d'élément le template va être déclenché (un modèle UML dans notre cas). Un nom de template est également précisé avec l'attribut name.

Dans le cadre de ce travail nous allons utiliser cet outil de génération de code pour définir, tout d'abord, des templates de génération de code fonctionnel ainsi que des templates de génération des aspects de sécurité. Ce même outil sera ensuite utilisé pour exécuter les templates et produire le code fonctionnel et le code correspondant aux aspects de sécurité, respectivement en utilisant les templates de code fonctionnel et les templates aspects, propres à un modèle d'application donné. Ce choix est justifié par le fait qu'Acceleo est tout d'abord une solution open source et aussi nativement intégré à Eclipse ce qui lui permet de se fondre dans l'interface graphique et réutilise des composants existants et éprouvés par la plateforme.

6.3 Présentation de l'atelier d'intégration des patrons de sécurité

Dans cette section,nous présentons les différents éléments qui caractérisent le développement de l'atelier SCRI-TOOL. Après avoir rappelé l'objectif et le fonctionnalités principales de l'atelier, nous présentons l'architecture technique générale de l'atelier, puis nous détaillons l'implémentation des principaux modules le constituant.

6.3.1 Objectif de l'atelier logiciel

L'objectif principal de l'atelier SCRI-TOOL est de valider le processus SCRIP (que nous avons défini dans le chapitre précédent) permettant l'intégration des patrons de sécurité

dans un modèle à base de composants. L'approche adoptée consiste à proposer un outil facilitant le transfert de connaissance entre un expert en sécurité chargé de la proposition des patrons de sécurité et un expert d'un domaine applicatif (non-expert en sécurité)chargé de modéliser une application à base de composants qui va supporter les aspects de sécurité définis par les patrons (Figure 6.5).

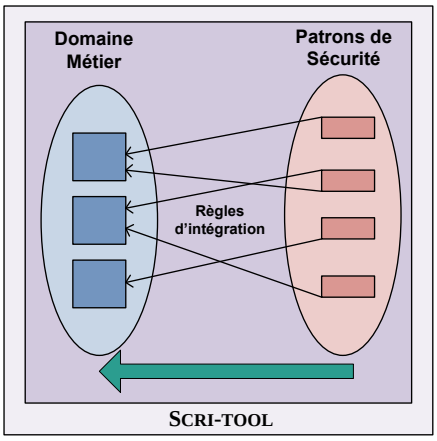


Figure 6.5 — Objectif principal de l'atelier logiciel à développer

6.3.2 Fonctionnalités de l'atelier logiciel SCRI-TOOL

Afin de répondre à l'objectif décrit dans la section précédente,nous proposons une maquette de l'atelier logiciel, qui est, en fait, une combinaison de trois outils offrant les fonctionnalités suivantes :

- **Modélisation de l'application métier** : l'utilisateur qui joue le rôle du concepteur métier utilise cette fonctionnalité pour modéliser l'application à base de composants. Le modèle résultant constitue une entrée pour l'outil.
- **Transformation :PIM2secPIM :** cette fonctionnalité permet de générer un modèle UML de l'application sécurisé après l'application successive d'un ensemble de patrons de sécurité. Cette fonctionnalité inclut une « sous-fonctionnalité » qui consiste à choisir le patron à intégrer. Pour le moment, ce choix est effectué manuellement par l'expert du métier.
- **Génération de code :** l'utilisateur qui joue le rôle du concepteur métier peut générer le code fonctionnel de l'application sécurisé, résultat de la fonctionnalité décrite précédemment. Par ailleurs, il peut également générer du code aspect de sécurité, à partir des templates déjà définis, à intégrer dans l'application. Cet enrichissement permet par la suite de générer un modèle secPSM qui dépend de la spécification d'une plate-forme technologique enrichie par du code aspects.

La figure 6.6 présente un extrait du diagramme de cas d'utilisation relatif à l'outil SCRI-TOOL. Les trois fonctionnalités citées ci-dessus sont représentées par des cas d'utilisation. La deuxième fonctionnalité (intégration des patrons de sécurité) représente notre contribution majeure en matière de définition et d'implantation des règles d'intégration et des fichiers de transformation. Nous détaillons ce travail ultérieurement.

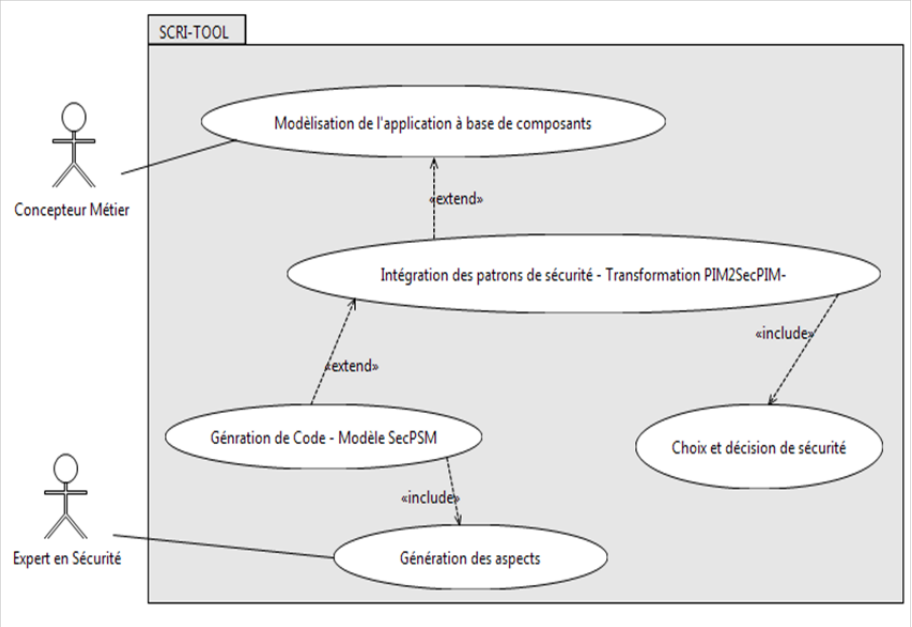


Figure 6.6 — Extrait du diagramme de cas d'utilisation de l'outil SCRI-TOOL

6.3.3 Architecture fonctionnelle et technique

La figure 6.7 présente l'architecture technique générale de l'atelier SCRI-TOOL. Ce dernier est basé sur quatre outils « logiciels libres » qui s'exécutent tous dans un IDE (Integrated Development Environment) sous Eclipse :

- **la plate-forme UML2** permet la modélisation de l'application à base de composants conforme au méta-modèle de composants UML 2.0.
- **L'outil Atlas Transformation Language(ATL)**permet de récupérer la définition exploitable d'un modèle à composants de l'application à sécuriser (format .uml) et de générer un modèle sécurisé de cette même application UML(fichier.uml). Cet outil produit le résultat de l'exécution des règles d'intégration que nous avons définies et implantées.
- **L'outil Acceleo** assure la génération du code de l'application ainsi que la génération des aspects à partir des templates aspects déjà produits.
- **L'outil aspectsJ** est utilisé pour tisser le code fonctionnel et le code des aspects générés dans le but de produire un code sécurisé de l'application.

6.4 Architecture générale de la partie modélisation de l'atelier logiciel SCRI-TOOL

Le diagramme de composants élaboré par le concepteur avec le modeleur UML2.0 Tools est une entrée de la partie ?Modélisation' de l'outil SCRI-TOOL (figure 6.8). Afin d'assurer une interaction dynamique entre l'utilisateur et les fenêtres de notre plugin nous faisons passer ce diagramme à travers un analyseur syntaxique XML (étape 1). Un analyseur syntaxique XML (ou parseur), permet de récupérer dans une structure XML, des balises, leur

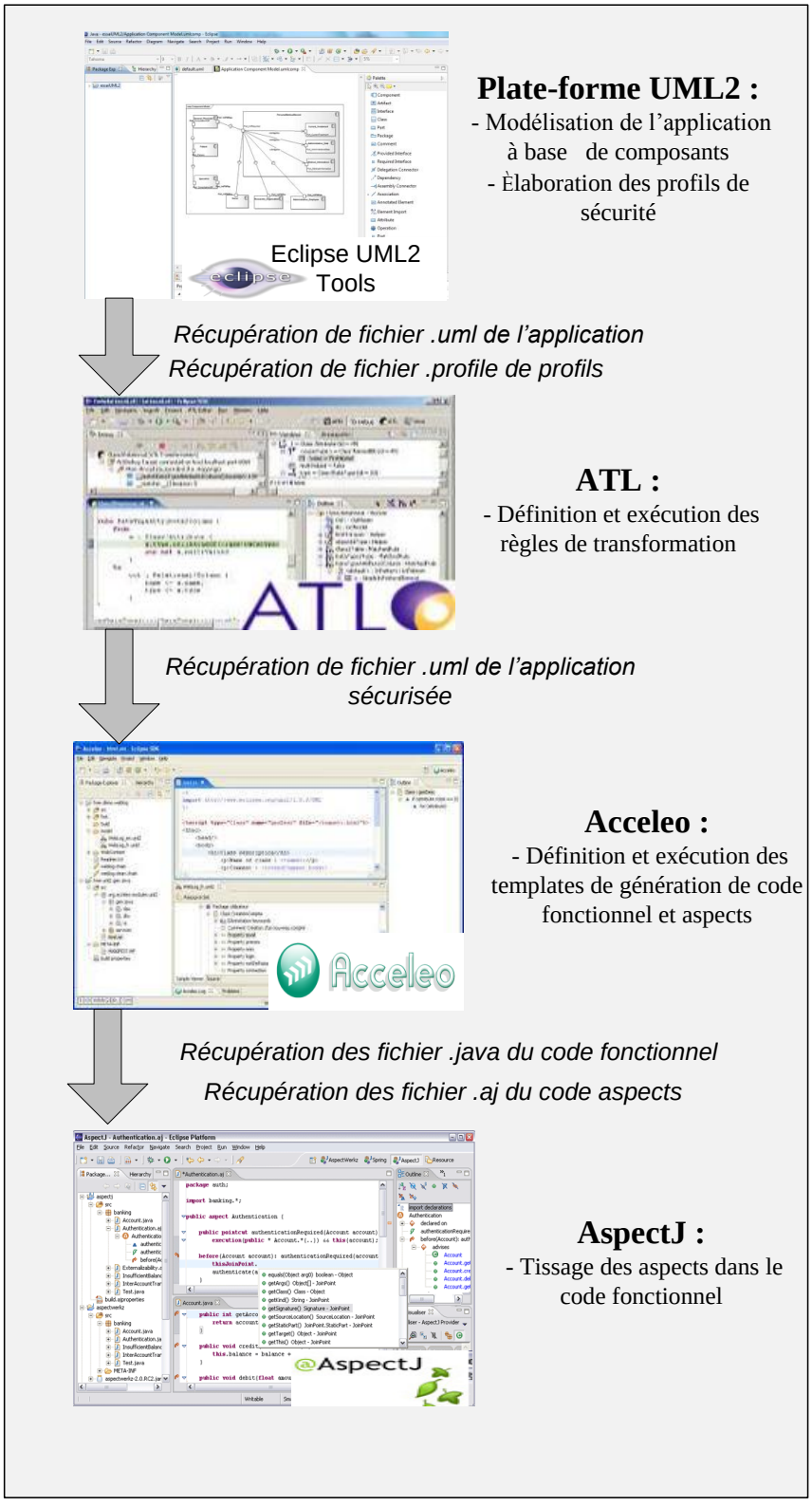


Figure 6.7 — Architecture technique générale de l ?atelier logiciel SCRI-TOOL

contenus, leurs attributs et de les rendre accessibles. Ce parseur parcourt tout le diagramme et extrait la liste des artefacts (composants, ports, connecteurs, etc.) qui le constituent (étape

2). Ces artefacts sont passés en paramètres des interfaces homme/machine. A travers ces interfaces, l'utilisateur pourra modifier et/ou ajouter des attributs dynamiquement (nom des composants, nom des ports, stéréotypes, etc.) (Étape 3).

Notre outil fournit à l'utilisateur des interfaces qui lui permettent de régler la configuration de sécurité qu'il souhaite apporter à son application. Un fichier sous format XML est donc généré à l'issu de l'étape 4 ;ce fichier enregistre la configuration de sécurité défini par le concepteur (ce fichier sera ensuite sollicité pour l'élaboration des modules ATL ainsi que des templates aspects). D'autre part, ces interfaces proposent à l'utilisateur de choisir le patron de sécurité qu'il veut appliquer sur son diagramme de composant, ainsi que les artefacts sur lesquels il veut appliquer les stéréotypes de sécurité du patron choisi. Les choix effectués par le concepteur sont sauvegardés par le système (étape 5) ; les détails relatifs à ces interfaces sont illustrés par le cas d'étude présenté dans la section 1.6 de ce chapitre.

La partie modélisation de notre atelier contient des modules ATL de transformation de modèle (étape 6) ; l'implémentation des ces modules est décrit dans la section 1.4.2 de ce même chapitre). Ces modules sont traités par des analyseurs de texte (étape 7) afin d'y injecter les choix de l'utilisateur sauvegardés auparavant dans (étape 5). Nous obtenons ainsi des modules ATL enrichis conformes aux choix effectués par le concepteur lors de son interaction avec les interfaces homme/machine de l'outil SCRI-TOOL(étape 8).

Après avoir configuré la transformation, les modules ATL d'application de patron de sécurité sur un modèle à base de composants peuvent être exécutés. Le résultat de cette transformation ATL produit la deuxième sortie de notre outil qui est un diagramme de composants annoté par des stéréotypes de sécurité. Cette annotation constitue l'application du patron de sécurité choisi par le concepteur sur le modèle à composants initial.

Dans les deux sections suivantes, nous détaillons l'implémentation des deux principaux artefacts de la partie modélisation de l'outil SCRI-TOOL, à savoir les profils UML de sécurité, et les règles d'intégration des patrons.

6.4.1 Implémentation des profils de sécurité

Nous rappelons tout d'abord que la notion de profil de sécurité présentée dans le Chapitre 5 a pour objectif de spécialiser UML afin de supporter les concepts des patrons de sécurité. Pour modéliser ces concept de sécurité, nous avons étendu le méta-modèle UML 2.0 en introduisant un certain nombre de stéréotypes, à savoir :« ProtectionObject »,« role »,« right», comme présenté par la figure 6.9.

Afin d'implémenter les profils de sécurité, comme présenté dans le chapitre précédent, dans le prototype développé, nous avons utilisé la plate-forme UML2 d'Eclipse. Cette implémentation est basée sur le méta-méta-modèle EMOF. Comme il est basé sur Ecore, ce plug-in utilise le framework EMF pour accéder à l'implémentation du méta-modèle standard UML 2.0, et permet de créer des profils UML correspondant à ce méta-modèle. Nous avons déjà présenté dans la section 5.6 du chapitre 5 le processus d'élaboration du profil. En utilisant l'éditeur de modèles UML fourni par ce plug-in et en suivant le processus proposé nous pouvons implémenter le(s) profil(s) UML de sécurité.

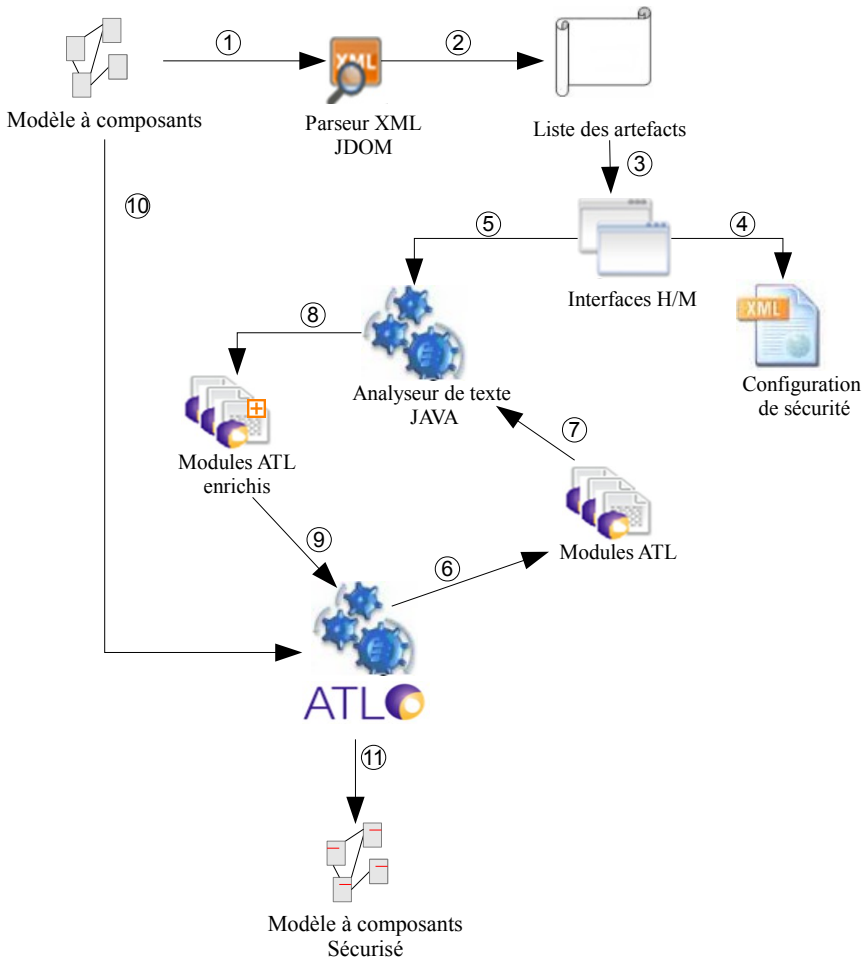


Figure 6.8 — Architecture générale de la partie modélisation de l'atelier logiciel

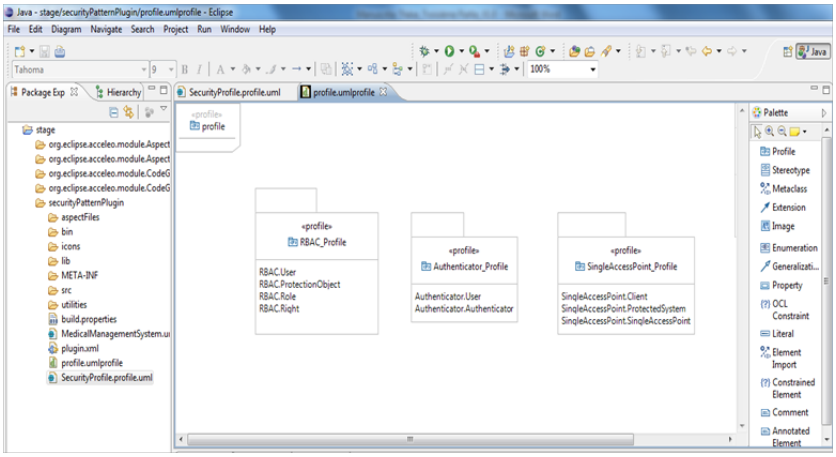


Figure 6.9 — Profil UML pour la politique de contrôle d'accès

6.4.2 Implémentation des règles d'intégration de la sécurité

Nous rappelons qu'une définition de transformation ATL consiste en un ou plusieurs modules de transformation. Chaque module est constitué de quatre sections : une section entête, une section d'importation de bibliothèques, une section de déclaration des "helpers" et une section de spécification des règles de transformation.

La figure 6.10 présente l'architecture générale du module d'intégration. Ce module prend en entrée le modèle à composants UML 2.0 et le profil UML contenant la définition de stéréotype de sécurité, et génère en sortie le modèle de composants sécurisé qui contient les différentes annotations de sécurité selon le patron appliqué. Un exemple est présenté dans la section 6.6.4.

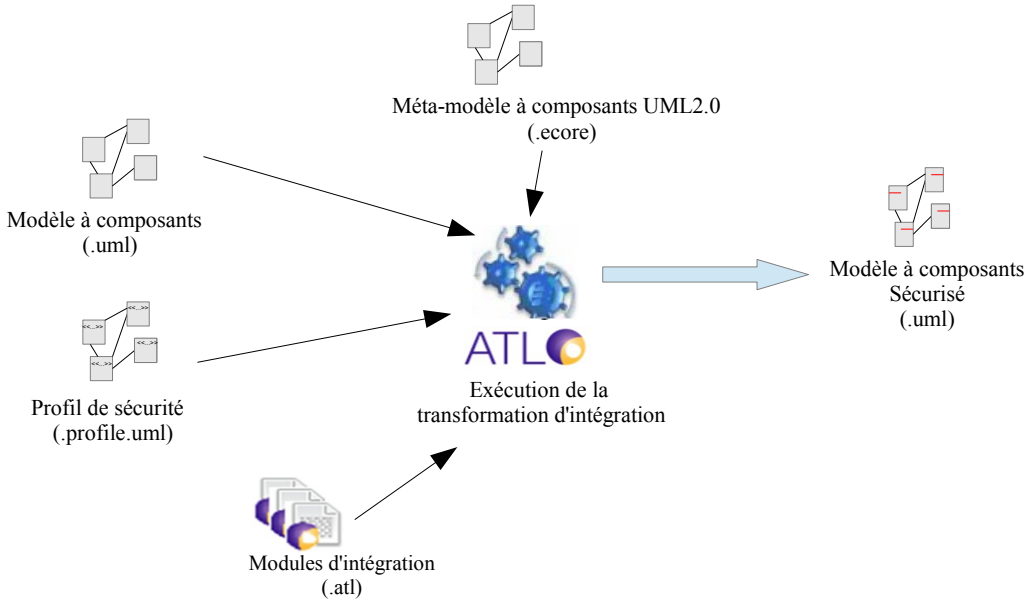


Figure 6.10 — Schéma général de la transformation d'intégration

Nous présentons ci-dessous dans ci-dessous quelques règles d'intégration. Nous rappelons que les règles d'intégration permettent d'annoter les éléments du modèle de l'application avec les profils de sécurité. Nous définissons aussi des règles de translation servant à retranscrire les éléments du modèle PIM dans le modèle PIMsec avant d'être annotés.

Pour des raisons de lisibilité, nous ne présentons pas le code source complet de cette règle ; le détail du module ATL d'intégration est fourni en annexe ?.

```
1 rule Connector2Connector {
2  --Copy Connector Bloc--
3 from s : UML2!"uml::Connector" in IN
4 to t : UML2!"uml::Connector" (
5      --__xmiID__ <- s.__xmiID__,
```

```
      name <- s.name.debug('Connector'),
      visibility <- s.visibility,
      isLeaf <- s.isLeaf,
      isStatic <- s.isStatic,
      eAnnotations <- s.eAnnotations,
      ownedComment <- s.ownedComment,
      clientDependency <- s.clientDependency,
      nameExpression <- s.nameExpression,
      type <- s.type,
      redefinedConnector <- s.redefinedConnector,
      end <- s.end,
      contract <- s.contract)
   --End Copy Connector Bloc--
--Operations on Connector--
   do {
     --Reapply existant Stereotypes--
     for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
     --End Reapply existant Stereotypes--
}
   --End Operations on Connector--
}
```

La règle de translation *Connector2Connector* ci-dessus définit le comportement de translation des connecteurs . Cette règle permet une recopie de l'élément connecteur dans le modèle cible et de vérifier si il est annoté par un stéréotype. Cette vérification permettra entre autre de ré-appliquer le même stéréotype sur la copie du connecteur.

Cette règle fait appel dans sa partie impérative (bloc do) au helper *getAppliedStereotypes* défini dans le figure ci-dessous pour vérifier si cet élément est déjà stéréotypé et le récupérer pour empêcher l'application multiple d'un même stéréotype sur un même élément.

```
% section entête
module InitialRBACTransformation;
create OUT : UML2 from IN : UML2, IN1 : PRO;

% Ce helper permet de retourner les stéréotypes du profil déjà appliqués
helperdef: getStereotype(p: UML2!Profile, name: String): UML2!Stereotype
    =
  p.ownedStereotype -> select(s | s.name = name) -> first();

% Ce helper permet de vérifier si un stéréotype est appliqué à un élément
helpercontext UML2!Element def: hasStereotype(stereotype : String) :
    Boolean =
  self.getAppliedStereotypes() -> collect(st | st.name) -> includes(
     stereotype);

% Cette règle permet d'appliquer des stéréotypesà un connecteur
rule Connector2secConnector {
  from s : UML2!"uml::Connector" in IN
to t : UML2!"uml::Connector" (
   do {
```

```
18      for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
19
20  if(((s.end->exists(a | a.role.owner.hasStereotype('RBAC.User')))or(s.end
        ->exists(a | a.role.owner.hasStereotype('Authenticator.User'))))and(s.
        end->exists(a | a.role.owner.hasStereotype('RBAC.ProtectionObject')))
        or (s.owner.hasStereotype('RBAC.ProtectionObject')))
21    {t.applyStereotype(thisModule.getStereotype(thisModule.entityProfile,'
        RBAC.Right'));  }
22  }
23      }
```

Dans la section entête du module, on trouve le nom du module, la déclaration des modèles source, cible, avec leurs méta-modèles correspondants (ligne 1-2). Sur l'exemple de la figure ci-dessus, IN et PRO représentent les modèles source (PRO est le modèle de définition du profil UML de sécurité) qui sont conformes au méta-modèle UML2. OUT dénote le modèle de sortie à générer qui est conforme lui aussi au méta-modèle UML2.

Le helper *getStereotype* (lignes 4-7) s'applique à un élément de modèle, et retourne les stéréotypes d'un profil appliqué à un modèle.

Les règles de transformation sont déclarées sous forme de Matched Rules (la forme déclarative d'ATL) ; leurs invocations se fait explicitement par le moteur ATL. Sur l'exemple, la règle *Connector2secConnector* (lignes 14-20) définit en sortie un connecteur stéréotypé (annoté par des concepts du patron de sécurité) à partir d'un connecteur du modèle à composant de l'application non sécurisé. Cette règle s'applique à tous les connecteurs de modèle avec une condition spécifiée par le helper *getAppliedStereotypes* (lignes 10-13). La ligne 24 fait un appel explicite à la méthode prédéfinie dans le langage ATL *applyStereotype*. C'est une autre forme de règle déclarative appelée "unique lazy rule". Ces règles ont la caractéristique d'être invoquées explicitement par d'autres matched rules. Elles s'appliquent sur les éléments fournis comme paramètre ; ici il s'agit du connecteur.

6.5 Architecture générale de la partie génération de code de l'atelier logiciel SCRI-TOOL

Après avoir intégré les patrons de sécurité dans le diagramme de composant, l'outil propose une partie de génération de code de l'application. La génération de code est une technique permettant à travers un modèle UML donné en paramètre, de générer des fichiers de code. Le type de fichier généré ainsi que son contenu sont spécifiés dans des templates de génération de code. Cette section présente dans un premier temps l'implémentation des templates de génération de code fonctionnel et dans un deuxième temps, l'implémentation des templates de génération des aspects de sécurité correspondant aux patrons de sécurité appliqués lors de la phase de conception.

Après avoir produit le diagramme de composant sécurisé ainsi que la configuration de sécurité dans la partie ?modélisation', notre outil réutilise ces deux résultats pour passer à la phase de génération de code (flèche 1 dans la figure 6.11). Dans cette partie SCRI-TOOL contient deux types de module Acceleo pour la génération de code (flèche 3 et 6). Le premier

type de module présente des templates de génération du code fonctionnel de l'application conçue alors que le deuxième assure la génération de code aspects. Différentes plates-formes peuvent être ciblées par cette génération de code. Dans le cadre de notre travail nous avons opté pour des templates de génération EJB.L'outil Acceleo utilise ces templates (flèche 2) ainsi que le diagramme de composant sécurisé (flèche 1) pour générer les beans, les interfaces remote et locale pour chaque composant (flèche4). Ces fichiers (.java) représentent la troisième sortie de l'outil développé. Le deuxième type de module présente des templates de génération d'aspect de sécurité élaborées via Acceleo et écrites en se basant sur le fichier de configuration de sécurité fourni par la partie ?Modélisation' de SCRI-TOOL. En effet, l'outil Acceleo utilise ces templates (flèche 5) ainsi que le diagramme de composant sécurisé afin de produire des aspects (.aj). Ces derniers permettent l'introduction de la sécurité définie par les patrons de sécurité -appliqués sur le diagramme de composants lors de la phase de modélisation- au niveau de l'implémentation (flèche 7). Ces fichiers (.aj) présente la quatrième sortie de l'outil SCRI-TOOL.

Le code fonctionnel généré ainsi que les aspects générés à partir des templates présentés dans ce travail constituent des squelettes de code. Ces derniers seront complétés par la suite par le concepteur selon ses besoins et les spécificités de chaque application.

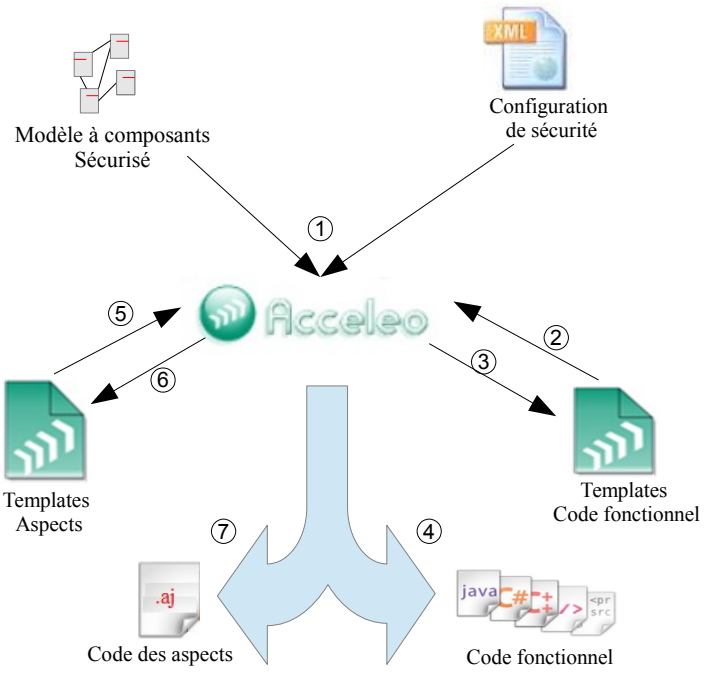


Figure 6.11 — Architecture générale de la partie génération de code de l'atelier logiciel

6.5.1 Implémentation des templates de génération de code fonctionnel

Afin de faciliter la phase de transition d'une conception à base de composants UML vers la phase de programmation, nous avons défini des templates de génération de code avec l'outil Acceleo. Comme nous l'avons déjà présenté dans la section 5.4 du chapitre précédent, nous proposons une technique de génération de code à base de composants multi-cibles à partir d'un modèle UML sécurisé. Dans ce qui suit, et afin d'illustrer notre approche nous avons choisi la plateforme EJB qui est une technologie largement utilisée pour développer des composants Java réutilisables [EJB].

Dans ce contexte, nous avons réutilisé des travaux existants comme celui de [Ziadi *et al.*, 2002] qui décrivent comment transcoder les éléments d'un modèle à base de composants UML vers des éléments d'un langage à base de composants cible. Dans la terminologie MDA, ceci consiste à transformer un modèle PIM de conception en un modèle PSM spécifique à une plate-forme orientée composants.

La dernière étape consiste à générer le code texte à partir du modèle PSM. Même si nous avons choisi de cibler dans un premier temps la plateforme EJB dont le profil est décrit dans la figure 6.12, les techniques utilisées dans cette génération de code sont réutilisables pour cibler d'autres plates-formes comme CCM, Fractal ou Sofa. Dans la suite de cette section, nous présentons les détails de l'implémentation en ATL des règles de traduction d'un modèle à base de composants conforme au méta-modèle UML 2.0 en du code EJB. En d'autres termes, nous présentons les étapes permettant de produire des templates de génération de code EJB.

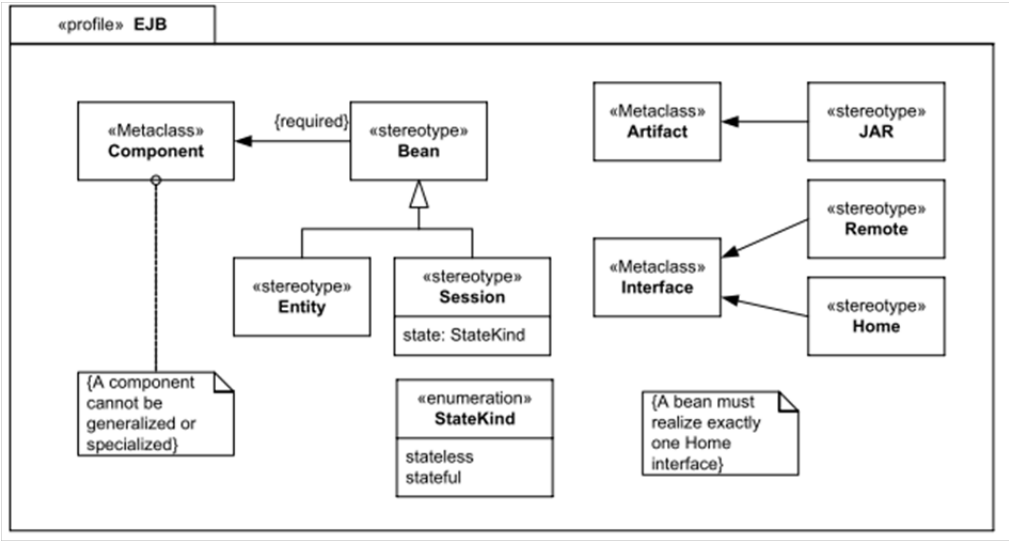


Figure 6.12 — Profil EJB défini par le standard UML 2.0

Afin de générer le code fonctionnel du diagramme de composants conçu par l'utilisateur lors de la phase de conception, une première étape consiste à développer des templates. Ces derniers permettent de générer des squelettes de code fonctionnel qui pourront être enrichis dans un second temps par le concepteur.

La génération du code fonctionnel nécessite alors, pour notre cas, des templates de gé-

nération de code fournissant des fichiers (.java) qui constituent un EJB. Le mappage diagramme de composants-EJB que nous adoptons est basé sur les travaux de Jean-Marc Jézéquel [Ziadi *et al.*, 2002]. Un composant du diagramme se transforme en une classe Bean, le port et le connecteur représentent les interfaces par laquelle nous instancions chaque composant, ici nous parlons de deux interfaces pour notre EJB qui sont l'interface Remote et Locale. Enfin le connecteur qui relie les composants représente l'invocation de méthodes entre les EJBs. Cette invocation ne peut pas être traduite sous forme de règles de génération de code car elle est décrite par l'utilisateur selon les besoins de l'application cliente qui utilise ces EJBs.

Nous présentons dans la suite la démarche adoptée afin d'élaborer un template permettant de générer le code fonctionnel de notre application. Le template de génération du code fonctionnel de notre diagramme de composant doit générer des fichiers (.java) pour la création d'EJBs, c'est pourquoi ce template est divisé en trois sous-parties. La première est consacrée à la définition des règles de génération des Bean des composants tandis que la deuxième et la troisième permettent de définir les règles de génération des interfaces Remote et Locale qu'implémentent le Bean généré. Pour la génération des fichiers Bean, nous procédons comme suit :

- D'abord nous parcourons le diagramme de composants afin de détecter tous les composants qui le constituent.
- Nous définissons ensuite la règle de génération du Bean pour chaque composant détecté dans l'étape précédente.
- Dans la règle de génération du Bean, nous parcourons la liste de propriétés de chaque composant afin de détecter ses « property » et ses « operations ».
- Pour chaque « property » détectée nous élaborons une règle de génération d'attribut ainsi que les règles de génération des méthodes getter et setter liées àcet attribut.
- Pour chaque opération détectée nous élaborons une règle de génération du squelette de la méthode correspondante.

Après la génération du Bean, nous passons à la génération les interfaces Remote et Locale de chaque composant. Les étapes de génération d'interface sont les mêmes pour le type Remote et Locale :

- D'abord nous parcourons le diagramme de composants afin d'identifier tous les composants qui le constituent.
- Nous définissons ensuite la règle de génération de l'interface Remote/Locale pour chaque composant détecté.
- Dans la règle de génération de l'interface Remote/Locale, nous parcourons la liste des propriétés de chaque composant afin de détecter ses « operations ».
- Pour chaque opération détectée nous élaborons une règle de génération de la déclaration de méthode correspondante. Ce template ne permet bien entendu d'obtenir que des squelettes de code. Une fois encore le développeur est amené à intervenir pour compléter le code des méthodes et des constructeurs générés. Un exemple complet de template de génération de code fonctionnel EJB est présenté en annexe ?.

6.5.2 Implémentation des templates de génération de code des aspects

Dans le contexte de ce travail, nous avons défini des aspects relatifs à chaque patron de sécurité. Ces aspects seront donc appliqués dans le modèle à composants de façon à garantir la séparation entre les aspects fonctionnels des composants et les aspects de sécurité proposés par les patrons de sécurité. Dans ce qui suit nous détaillons la démarche adoptée afin d'élaborer un template permettant de générer un aspect qui assure l'intégration de la sécurité définie par un patron de sécurité.

Nous commençons d'abord par définir la règle de génération d'un aspect de base où nous définissons le nom de l'aspect et le nom et l'extension du fichier à générer. Ensuite nous parcourons le diagramme de composant sécurisé afin de détecter les composants qui possèdent des stéréotypes de sécurité. Pour chacun de ces composants nous définissons une règle de génération d'un mécanisme d'introduction. Ce mécanisme est élaboré en effectuant les étapes suivantes :

- Selon le stéréotype appliqué sur le composant, nous définissons les attributs à introduire dans la classe du composant. Ces attributs seront utilisés plus tard pour sauvegarder les informations concernant la sécurité à appliquer pour l'application développée.
- Les attributs ainsi définis seront enrichis à travers des méthodes qui permettent l'extraction d'informations qui se trouvent dans un fichier de configuration de sécurité spécifique au patron appliqué.
- Créer le mécanisme qui permet d'introduire ces attributs et ces méthodes. Ce mécanisme consiste en une déclaration de ces attributs et méthodes dans le code de l'aspect.
- Le code du mécanisme d'introduction ainsi obtenu, nous passons à la définition de la règle de génération qui permet d'obtenir automatiquement, en partant d'un diagramme de composants, le traitement adapté aux informations du diagramme. Après avoir défini la règle de génération du mécanisme d'introduction pour chaque composant détecté, nous passons à la règle de génération des coupes et des advices qui se chargent de définir le comportement de la sécurité que nous souhaitons apporter à notre application.

Pour réaliser cette règle nous effectuons les tâches suivantes :

- Nous définissons d'abord, selon le besoin de notre application, les points de jonction où nous interrompons l'exécution pour introduire un nouveau comportement dans l'application spécifique à la sécurité que nous voulons produire.
- Nous insérons ensuite dans les blocs du code advice l'invocation des méthodes déjà introduites dans une étape précédente ainsi que d'autres méthodes de test pour assurer le comportement voulu.
- Enfin nous définissons une règle de génération des squelettes de ces coupes et advices que nous intégrons par la suite dans le template de génération d'aspect de sécurité. La génération de code du template que nous avons effectué permet d'obtenir des squelettes de code. Une fois ces derniers générés, le développeur doit compléter le code des coupes et des advices.

Le code complet du template de génération du code de l'aspect de sécurité relatif aupatron RBAC, est fourni comme exemple dans l'annexe ? de ce manuscrit.

Nous avons ainsi réalisé un atelier logiciel permettant l'intégration de la sécurité proposée par les patrons dans des modèles à bases de composants. Nous illustrons dans la prochaine section l'utilisation de cet atelier par une étude de cas portant sur la gestion d'un système de santé personnalisé.

6.6 Application à un cas d'étude

Cette section a pour objet de présenter l'application de l'approche proposée pour l'intégration des patrons de sécurité en utilisant le prototype SCRI-TOOL développé. Pour illustrer cette application, nous nous appuyons sur la sécurisation des modèles de conception à base de composants issus de la modélisation d'un cas d'étude réel. Nous avons choisi la modélisation d'un Système de Gestion de Dossier Médical Personnel (SGDMP).

Le choix d'une telle étude de cas est justifié par l'importance des exigences en termes de sécurité requises pour le bon fonctionnement d'une telle application. En effet, vu le grand nombre d'acteurs pouvant interagir sur ce type de système, et la nature des informations traitées, la sécurité de fonctionnement d'un tel système est une exigence critique.

6.6.1 Présentation du cas d'étude

Récemment, la recherche et le développement concernant les systèmes de santé électronique sont retenu l'attention du monde universitaire et industriel. À cet égard, l'initiative Integrating the Healthcare Enterprise (IHE) aide les professionnels de la santé et de l'industrie à améliorer la façon dont les systèmes de santé partagent l'information médicale [IHEa].

IHE opte pour l'unification de plusieurs domaines intervenant dans les systèmes de soins de santé (comme administration, radiologie.) afin de permettre le partage sécurisé et efficace des données. Toutefois, la sécurité de l'information partagée dans les environnements distribués et hétérogènes est une tâche difficile. L'IHE propose des méthodes de base afin de fournir des Healthcare Information Exchanges (HIE) [IHEa]. Au niveau technique l'initiative IHE définit un ensemble commun de contrôle de sécurité et de confidentialité, qui ont été identifiées à travers l'expérience des partenaires IHE. Les contrôles de sécurité comprennent les contrôles de base concernant la confidentialité des données, l'intégrité et la disponibilité, et les contrôles avancés telles que l'authentification, la non-répudiation et le contrôle d'accès.

L'étude de cas que nous présentons dans ce manuscrit présente l'utilisation de l'approche proposée pour prendre en compte les différentes exigences en termes de sécurité décrit pour un système de gestion de dossier médical personnel. Dans ce qui suit nous décrivons avec plus de détails les besoins en termes de sécurité pour un système de gestion de soins. Nous présentons l'importance de la sécurité et la confidentialité dans ce type de systèmes. Nous présentons aussi le scénario d'étude complet.

6.6.2 La sécurité et la confidentialité dans les systèmes de soins

La sécurité et la confidentialité des dossiers médicaux des citoyens est une question très sensible. La législation donne des droits au citoyen en tant que propriétaire de ses données médicales Cela implique par exemple qu'il est obligatoire d'obtenir l'autorisation du citoyen en question avant d'utiliser ses données à d'autres fins.

Dans le but de fournir des services de soins de qualité et à temps opportun pour les différents patients, différents rôles et interactions entre les différents intervenants existent. La figure 16 présente un simple exemple d'interaction entre les différents acteurs. Le patient est au centre de cette interaction. Pour chaque patient, un Dossier Médical Personnel (DMP) est crée et maintenu. En résumé, un DMP représente un dossier médical virtuel assemblé à partir d'informations saisies par différents fournisseurs de services de soins de santé, qui ont acquis différentes informations cliniques au cours des consultations, analyses ou traitements antérieurs.

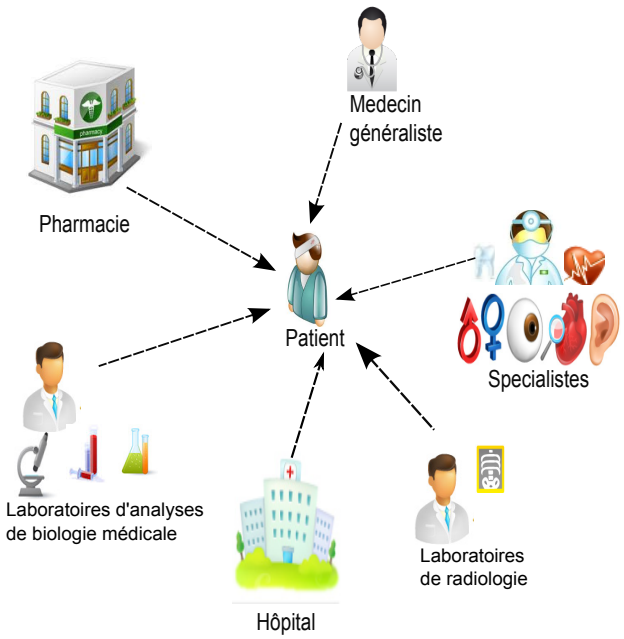


Figure 6.13 — Différents acteurs du système de soin

Le système de soin porte essentiellement sur la gestion collaborative de dossiers médicaux. Le dossier médical d'un patient est un outil de communication entre les acteurs de soins et les patients dans le cadre d'une prise en charge pluri-professionnelle (voir figure 6.13). Il permet de suivre le parcours hospitalier du patient, et il assure la traçabilité des actions effectuées.

Pour des raisons de simplification, nous limitons cette étude aux acteurs et activités suivants :

– Les patients effectuent des consultations chez les médecins, ils suivent des traitements et passent des examens et des tests dans les laboratoires d'analyses. Ils peuvent consulter leur dossier.

– Les médecins généralistes (médecin traitant) effectuent les diagnostics et les examens, rédigent les rapports de consultation, prescrivent les traitements, consultent les antécédents, etc.
– Les infirmiers dans un hôpital consultent les prescriptions rédigées par les médecins, notent les observations et les mesures systématiques d'éléments significatifs (tension, température, etc.) dans le dossier médical des patients, assurent le suivi quotidien des patients hospitalisés et communiquent leurs observations aux médecins responsables.
– Le personnel administratif dans un hôpital se charge de la saisie des informations administratives des patients, de la création des dossiers, de l'enregistrement des informations administratives et de leurs mises à jour.
La Figure 6.14 illustre un sous-ensemble des cas d'utilisation identifiés : Gestion de dossiers, Consultations, Traitements et Paiements.

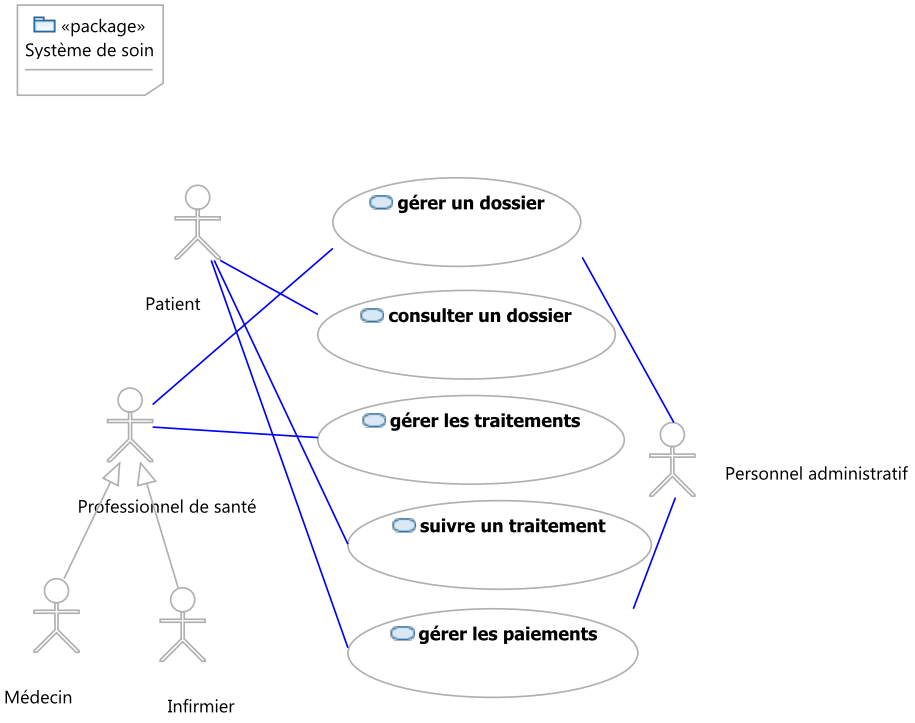


Figure 6.14 — Extrait du diagramme de cas d'utilisation du SGDMP

Le dossier médical d'un patient contient deux types d'informations.
– Le premier type d'information concerne l'identité du patient. Cette partie est connue sous le terme Données d'Identification Personnelles(DIP) [3]. Les DIP contiennent le nom, l'adresse, le numéro de sécurité sociale, le numéro de téléphone ou toute autre information liée à l'identité du patient.
– Le deuxième type d'information constitue l'historique médical du patient. Il contient différentes informations sur les maladies, les médicaments prescrits, les différents effets constatés, les analyses ainsi que les radios réalisées, etc. Cette partie du dossier médical est utilisée dans le domaine de la recherche médicale. En effet, en utilisant des techniques d'anonymisation, il sera possible d'exploiter ce type d'information sans divulgation de l'identité du patient.

Dans le cadre de ce manuscrit, nous allons appliquer l'approche proposée dans le chapitre 5 afin de concevoir une application de gestion de soins intégrant les différents besoins de sécurité.

6.6.3 Exemple de scénario de soins

La figure 6.15 présente les différentes parties interagissant dans un système de gestion de soin. Les différents acteurs sont représentés comme des rôles (médecin, radiologue, etc.) et leurs interactions avec un système de DMP. Ce dernier est modélisé comme des échanges de documents dans un scénario typique explicité ci-dessous.

Dans ce qui suit, nous présentons un scénario nominal d'une visite chez un médecin généraliste pour faire un bilan de santé annuel. Nous décrivons les différentes étapes par lesquelles passe le DMP lors des interactions avec les différents acteurs. Nous pouvons considérer le scénario nominal suivant :

« Mr. Dubon, un patient, consulte son médecin généraliste, pour son bilan de santé annuel (A. Ce dernier aura besoin d'accéder au DMP propre à Mr. Dubon pour voir éventuellement les antécédents de son patient. Dans l'étape 1, le médecin accède au dossier médical personnel (DMP)du patient. Après un premier examen, le médecin décide de diriger le parient chez un radiologue pour faire des radios. Pour cela, le généraliste met à jour le DMP avec les nouveaux diagnostics (étape 2) et émet un rapport médical électronique. Le patient se rend donc chez le radiologue (B), ce dernier accède au DMP du patient en question (étape 4) réalise les radios demandées et met à jour le DMP avec la radiographie produite(étape5), avant sa visite à nouveau à son médecin de famille, qui met à jour son DMP ».

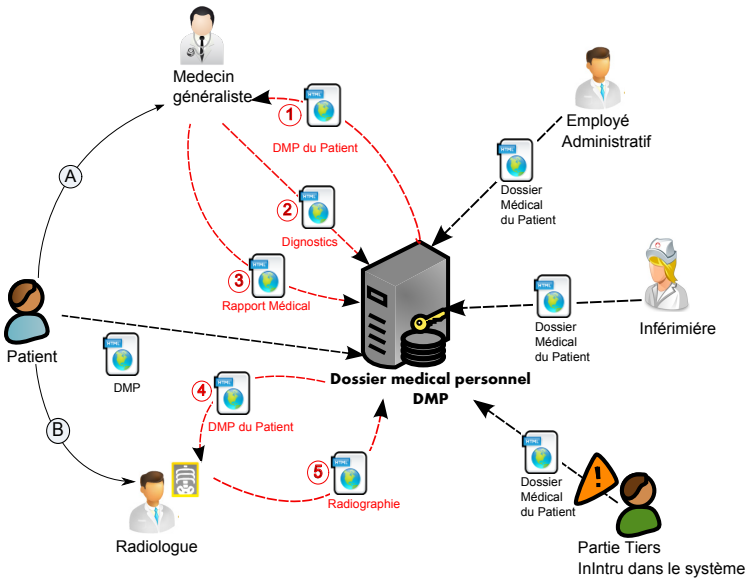


Figure 6.15 — Gestion d'un DMP

Dans un scénario assez simple comme celui-ci, et a fortiori dans des scénarios plus complexes, il existe un certain nombre d'exigences de sécurité et de confidentialité à respecter lors de l'accès aux DMP par les différents intervenants. Les utilisateurs dans les différentes organisations de soins tels que les cliniques, les laboratoires d'analyses ou les pharmacies

sont assignés à certains rôles en fonction de leurs responsabilités et implications dans le suivi du patient. Ces exigences de sécurité et de confidentialité empêchent les accès non autorisés aux données médicales par les personnes qui n'ont pas le droit. Dans ce cas, les rôles définissent les autorisations d'accès aux services médicaux. Un médecin généraliste, par exemple, peut mettre à jour un DMP. Tandis qu'un cardiologue devrait être en mesure d'accéder seulement aux parties du DMP dont il a besoin pour mener un diagnostic cardiaque.

Outre l'utilisation principale des données médicales dans le traitement des patients, les données médicales peuvent être également utilisées à des fins secondaires comme des enquêtes nationales ou à des fins de recherches publiques ou privées.Là aussi les exigences en termes de sécurité et de confidentialité sont assez nombreuses et nécessitent la mise en place d'applications sécurisées pour éviter tout débordement et utilisation inappropriée des différentes données personnelles.

Nous nous intéressons dans la suite aux exigences en termes de sécurité et de confidentialité qui doivent être prises en compte lors de ce scénario nominal.

Exigences en termes de sécurité et de confidentialité Sur la base du scénario décrit dans la section précédente, qui montre l'interaction entre les différents acteurs ainsi que leurs communications respectives, nous pouvons identifier un certain nombre d'exigences de sécurité. Ces acteurs créent, stockent, envoient et reçoivent différents types de documents relatifs à la santé de leurs patients (par exemple, un rapport médical, un rapport de diagnostic, des radiographies, etc.). Ces différents documents font partie du DMP, comme le montre la figure 6.15. La nature distribuée des systèmes de santé et l'hétérogénéité des applications intégrées soulèvent de nombreux défis en matière de sécurité. Il faut s'assurer que seuls les utilisateurs autorisés à consulter, à mettre à jour ou à supprimer un rapport peuvent le faire.

Un utilisateur autorisé est identifié par une identité unique ainsi qu'un rôle spécifique qu'il peut jouer. Dans ce contexte, l'identification d'un utilisateur est une des exigences de sécurité à définir. Cela implique que les différents acteurs aient une certaine politique pour l'identification des utilisateurs sur le système. Il faut avoir des politiques d'autorisation qui définissent les autorisations par rapport aux rôles joués. La définition des autorisations permet de spécifier, pour chaque acteur, quel rôle particulier il est autorisé à jouer une fois qu'il est identifié comme un utilisateur valide.

En outre, les scénarios de soins de santé impliquent l'envoi et la réception de différents documents médicaux, il est donc important que, une fois un document particulier envoyé ou reçu par un utilisateur, ce dernier ne devrait pas nié avoir envoyer ou recevoir certain documents. Ce résultat est obtenu grâce à la politique de non-répudiation. La non-répudiation assure la responsabilisation dans les systèmes de santé en matière d'accès aux documents ou aux services médicaux.

6.6.4 Modélisation du système SGDMP

Dans cette section, nous présentons la modélisation à base de composants du système de gestion du dossier médical de patients. La figure 6.16 présente le digramme de composants conforme au méta-modèle à composants retenu dans cette thèse qui est UML 2.0.

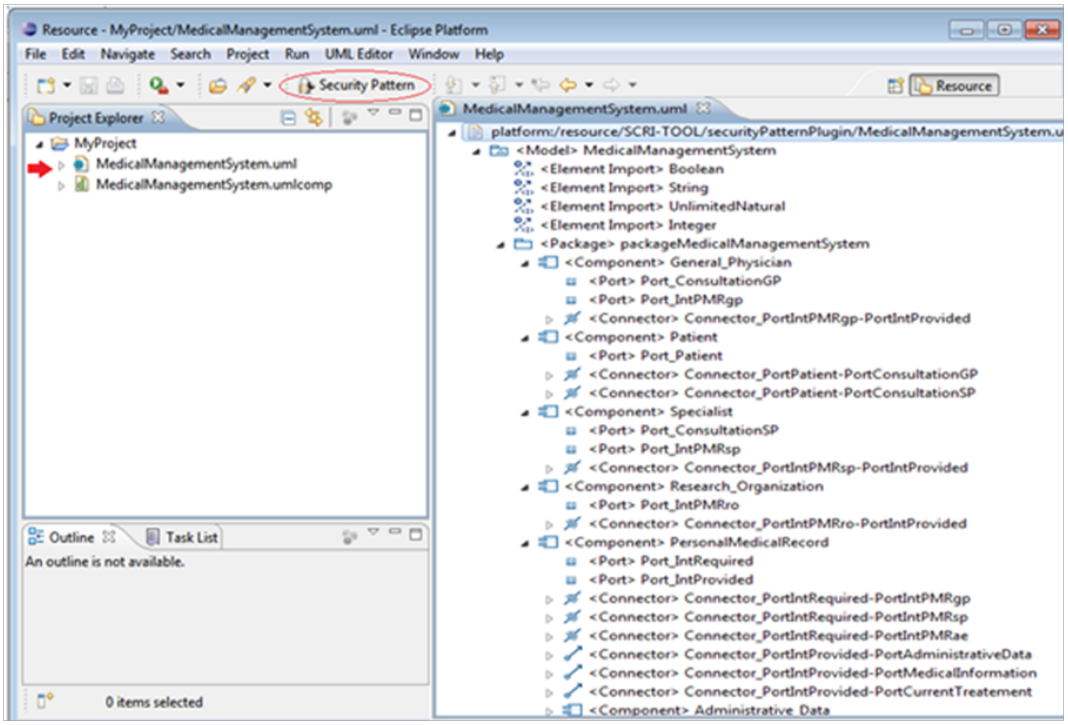


Figure 6.16 — Représentation arborescente du digramme de composants du SGDMP

Le même modèle de conception de l'application peut être présenté selon la vue arborescente de l'éditeur de modèles du Framework UML2 (figure 6.17).Cela permet de visualiser les éléments composés (composants, ports, etc.) sous forme iconifiée ou développée en visualisant leur contenu (attributs, opérations, ports, connecteurs, etc.).

6.6.4.1 Choix de patron de sécurité à appliquer

Dans la section 6.6.2, nous avons identifié les types de politiques qui doivent être appliquées dans le cas d'étude de système de soin : la politique de contrôle d'accès, la politique de confidentialité et celle de la non-répudiation. Par conséquent, nous considérons que les arbres de décision ayant la "confidentialité", le "contrôle d'accès" et la "non-répudiation "comme politiques racines dans la cartographie des patrons présentée dans la section 5.6.2 du chapitre 5 (figure 5.5). Nous illustrons la méthodologie par la politique de contrôle d'accès. En suivant les flèches "réalisation", la cartographie nous guide vers la sélection des patrons " RBAC", "Session", "System check Point", "Authorization" et "Reference Monitor" pour la politique de contrôle d'accès.

Le politique de contrôle d'accès est ainsi choisis, le profil UML de sécurité correspondant est alors élaboré pour permettre le passage à l'étape d'application des patrons appartenant à cette famille (contrôle d'accès). Après l'analyse des patrons de contrôle d'accès, nous avons choisi d'appliquer le patron "RBAC".

Dans ce cas d'étude, l'utilisation du patron RBAC nous parait adapté pour contrôler l'accès à des entités sécurisé du système. Dans ce modèle de contrôle d'accès chaque décision d'accès est basée sur le rôle auquel l'utilisateur est attaché. Un rôle découle généralement de

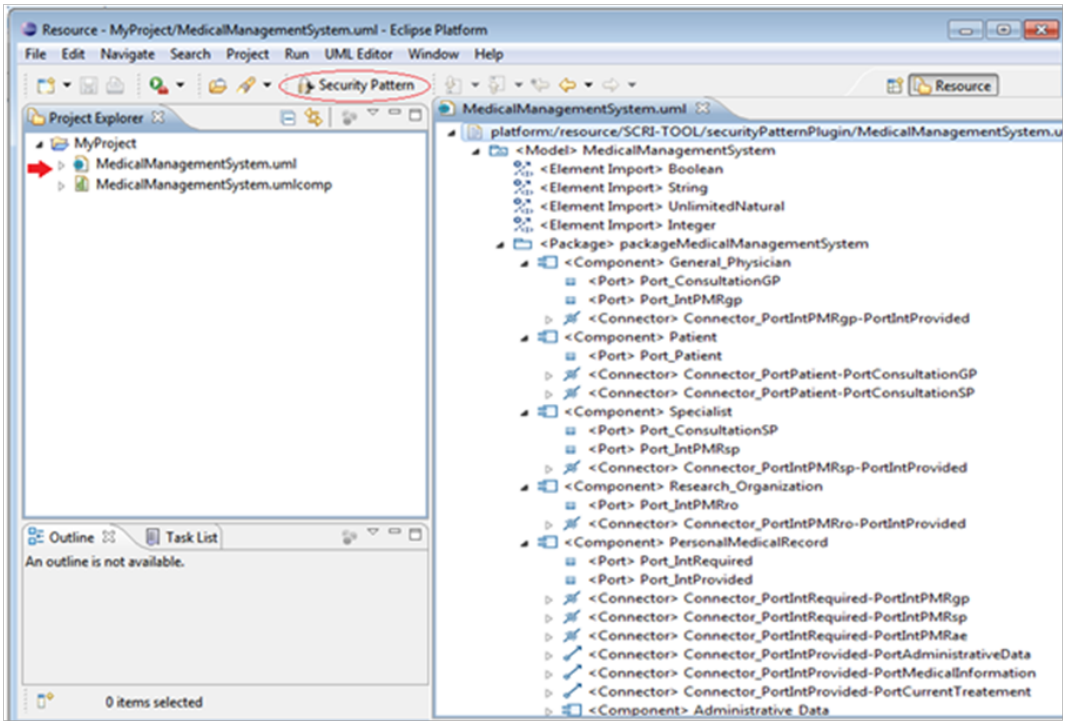


Figure 6.17 — Représentation arborescente du digramme de composants du SGDMP

la structure du système. De cette manière nous pouvons regrouper les utilisateurs exerçant des fonctions similaires sous le même rôle. Un rôle, associe à un sujet des autorisations d'accès sur un ensemble d'entités sécurisé.

La modification des contrôles d'accès n'est pas nécessaire chaque fois qu'une personne rejoint ou quitte le système. Par cette caractéristique, RBAC est considéré comme un patron « idéal » pour les systèmes dont la fréquence de changement d'acteurs est élevée.

L'interface d'initialisation du projet Eclipse permet d'appliquer des patrons de sécurité (bouton encerclé en rouge dans la figure 6.17). Un clic sur ce bouton permet l'affichage de la fenêtre de la figure 6.18. Cette interface nous permet d'effectuer trois actions qui sont : (1) le choix du modèle de composants à sécuriser (le choix est entre le modèle de l'application actuelle ou un autre modèle déjà existant), (2) le choix du patron de sécurité que nous voulons appliquer sur notre diagramme et enfin (3) la possibilité de consulter les détails du patron de sécurité choisi comme le montre la figure 6.19.

choixparomp

Le patron de sécurité "RBAC" sélectionné sera intégré au modèle de l'application en cours de construction. D'autres fenêtres peuvent s'afficher pour permettre l'application du patron de sécurité comme le montre les deux figures suivantes. Ces deux fenêtres sont spécifiques au patron "RBAC". Nous en avons besoin pour pouvoir spécifier les rôles et les droits dans cette étude de cas puis pour pouvoir attribuer ces rôles et ces droits aux différents acteurs de l'application.

Ces interfaces (figures 6.20, 6.21) permettent de définir les 'roles' et les 'right' de l'application système DMP (Figure 6.20) et de les affecter par la suite aux composants stéréotypés

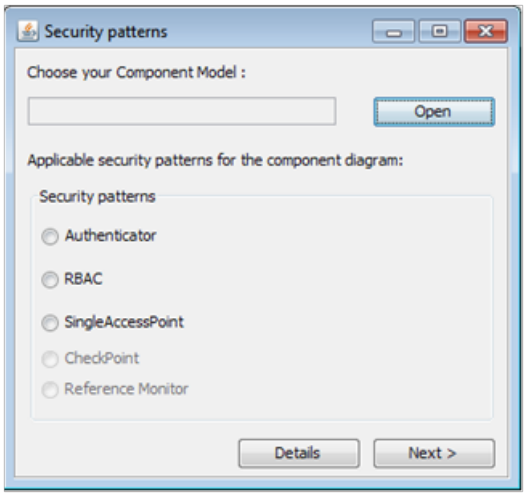


Figure 6.18 — Choix de patron de sécurité à appliquer

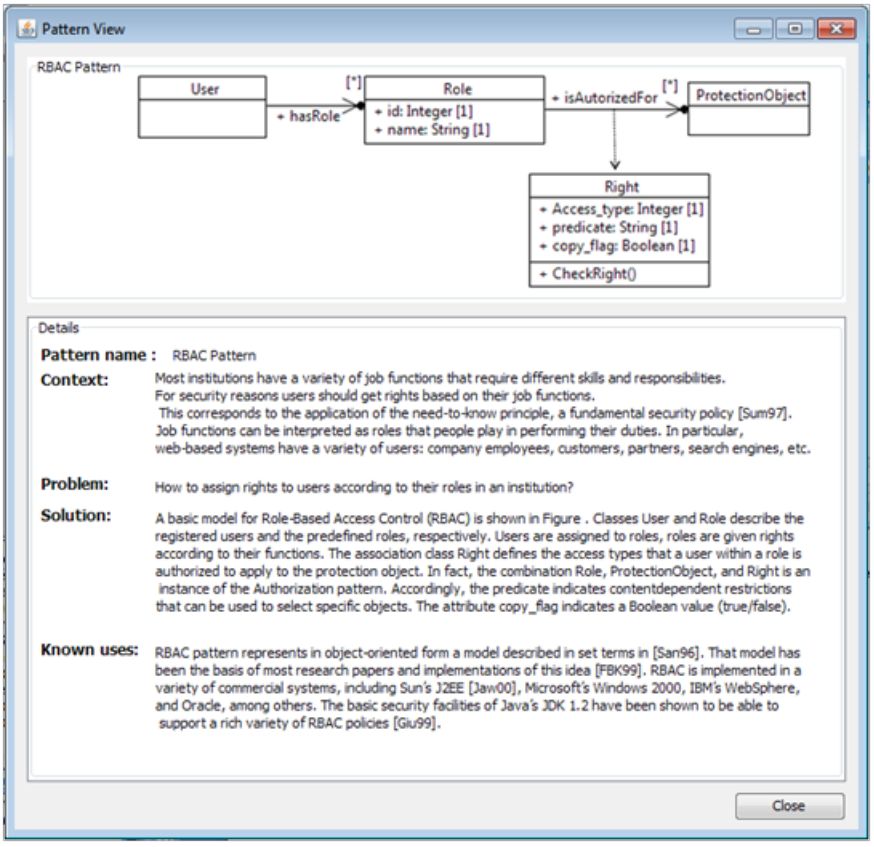


Figure 6.19 — Présentation détaillée du patron de sécurité

« User » (Figure 6.21). L'étape de choix du patron se termine par un clic sur le bouton " Apply Pattern" ce qui permet l'exécution des règles d'intégration relatives au patron choisi, ici "RBAC".

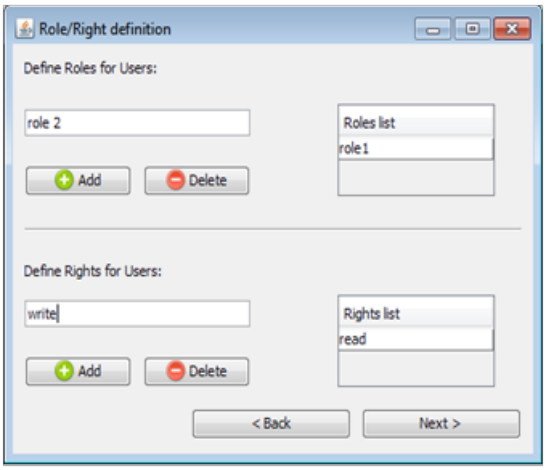


Figure 6.20 — Interface de définition des 'roles' et des 'rights'

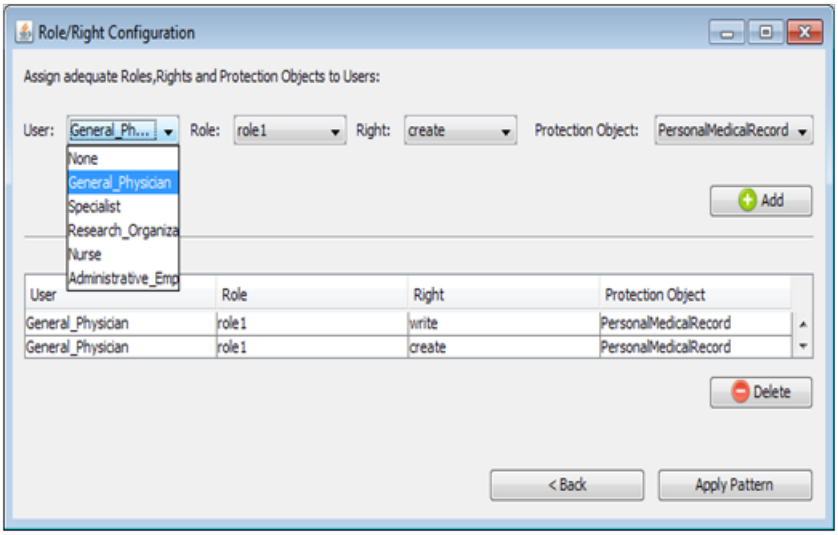


Figure 6.21 — Interface de configuration de sécurité

6.6.4.2 Intégration des patrons de sécurité

La figure 6.22 donne un aperçu du résultat généré après l'application des transformations ATL relatives au patron RBAC. Le fichier UML medicalmanagmentsystem.uml décrit le modèle de DMP qui embarque la solution de sécurité. Ce dernier, dans la figure 6.22, présente une description selon une vue arborescente des différents éléments UML (stéréotypés) qui composent le modèle.

Nous avons ouvert deux vues d'éditeur sur cette instance afin de pouvoir comparer et détecter facilement les modifications apportés au modèle initial (vue de gauche) par l'application du patron RBAC. La vue de droite contient le modèle du diagramme sécurisé obtenu suite à l'application du patron RBAC.

Dans la vue de droite, la flèche en bleu indique l'application du profil UML de sécurité sur le modèle à composant. Les cadres en rouge représentent les stéréotypes appliqués sur les composants que nous avons choisis auparavant à travers les interfaces des figures 6.20 et

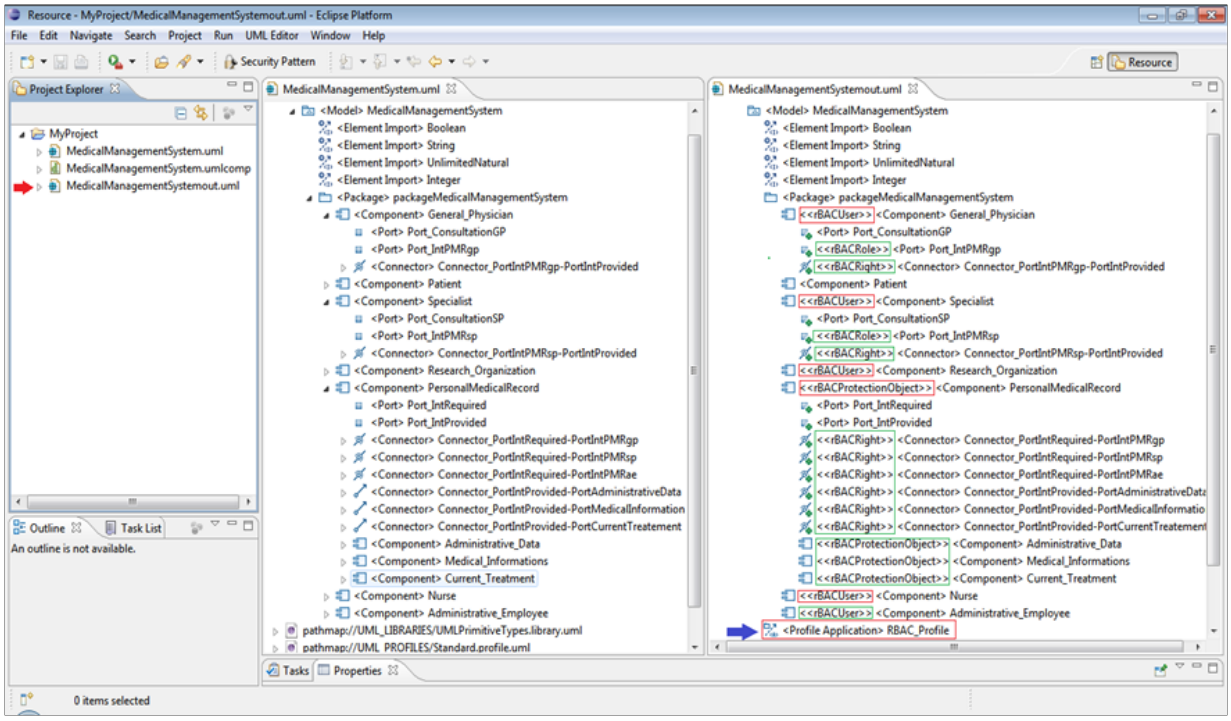


Figure 6.22 — Résultat de l'application du patron RBAC : Diagramme de composants sécurisé

6.21. Les cadres verts sont les stéréotypes appliqués automatiquement sur des artefacts du diagramme afin de compléter les notions de sécurité spécifiques à l'intégration du patron RBAC dans un diagramme de composants. L'application automatique de ces stéréotypes est assurée par les transformations ATL de notre approche.

SCRI-TOOL fournit aussi une deuxième sortie qui est un fichier XML comportant la configuration de sécurité définie à travers les interfaces présentées par les figures 6.20 et 6.21. Le code ci-dessous présente un extrait du ficher XML de la configuration générée après application du patron RBAC. L'intérêt de ce fichier est de permettre de restituer les choix de l'utilisateur pour la phase de génération de code.

```
1  <?xml version="1.0"?>
2  <users>
3      <user name="General_Physician">
4    <protectionObject name="Current_Treatement">
5      <right name="create"/>
6      <right name="read"/>
7      <right name="update"/>
8      <right name="delete"/>
9    </protectionObject>
10   ?
11      </user>
12 ?
13      <user name="Nurse">
14   <protectionObject name="Current_Treatement">
```

```
15      <right name="read"/>
16    </protectionObject>
17       </user>
18  </users>
```

Une fois le premier patron de sécurité intégré, il est possible de sélectionner de nouveaux patrons en utilisant à nouveau l'interface d'initialisation du projet (bouton encerclé en rouge dans la figure 6.17). D'autres patrons de sécurité peuvent être sélectionnés à leur tour selon la politique de sécurité choisie. Les patrons de sécurité peuvent appartenir à une même politique de sécurité (ici contrôle d'accès) ce qui permet l'utilisation du même profil de sécurité déjà établi lors de l'application du premier patron. Sinon, le concepteur peut changer de politique de sécurité. Dans ce cas, un autre arbre de décision sera extrait de la cartographie des patrons et ainsi un nouveau profil de sécurité sera élaboré correspondant à cette nouvelle politique.

6.6.5 Génération de code

Une fois la phase de modélisation et d'application de(s) patron(s)achevée,le concepteur d'application, d'une façon générale l'utilisateur del'outil SCRI-TOOL, peut entamerla phase de génération de code. Cette section décrit, en premier temps, le processus de génération de code en utilisant la plateforme SCRI-TOOL. Ce processus est composé de deux phases : la première consiste à générer le code fonctionnel de l'application en Java qui permet le développement des systèmes à base composants spécifique à la plateforme EJB. La deuxième phase s'intéresse à la génération d'un code Aspect selon le langage AspectJ à partir des préoccupations liées à l'aspect sécurité de tels systèmes. Une fois les deux codes sont générés, le tisseur AspectJ assemble le code technique dans notre programme métier à des points de jonction

Les deux fonctionnalités (génération de code fonctionnel et génération de code aspects) sont implémentés dans l'outil. L'exécution de l'une de ces fonctionnalités implique l'exécution des templates de génération de code développés dans le cadre de ce même travail et présentés dans la section 6.5 de ce même chapitre. Ces templates sont intégré à l'outil SCRI-TOOL comme présenter dans la figure 6.23.

6.6.5.1 Génération de code fonctionnel

Comme illustrer par le code ci-dessous une partie du code Java généré correspondant au composant "General_Physician". Ce code a été généré en utilisant les templates Acceleo présenté dans la section 6.5.2 de ce même chapitre. Le code complet généré est présenté en annexe ?.

```
1  package packageMedicalManagementSystem;
2  import javax.ejb.Stateless;
3  import javax.ejb.Remote;
4  import javax.ejb.Local;
5  @Stateless
6  % la classe bean du composant
```

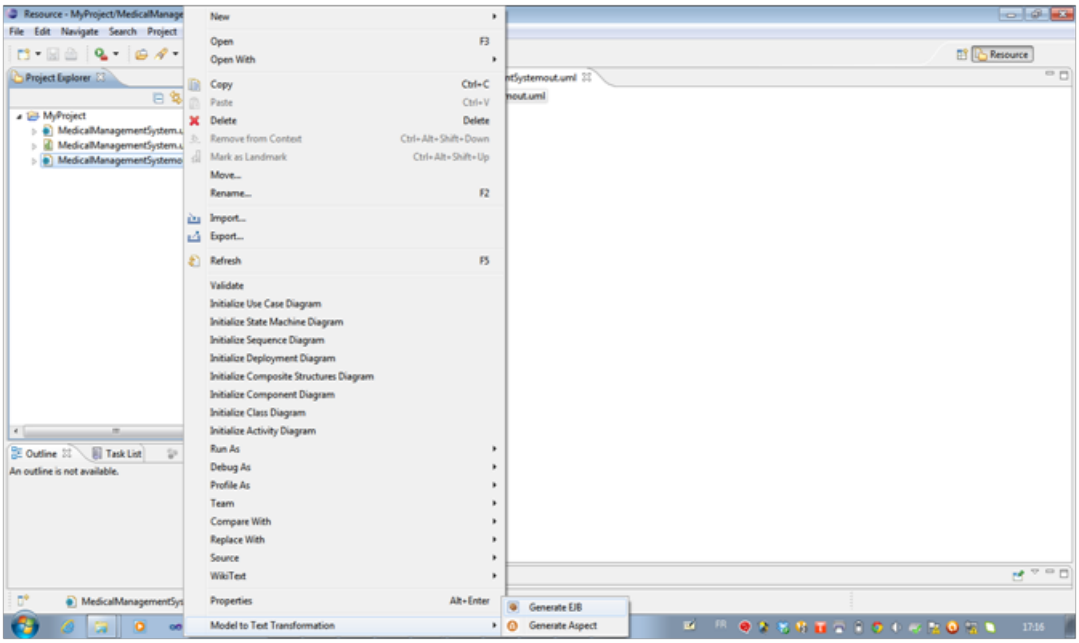


Figure 6.23 — Commandes permettant la génération de code

```
7  Publicclass General_Physician implements General_PhysicianRemote,
       General_PhysicianLocal{
8      public General_Physician(){
9        super();
10     }
11     public String getClassName(){
12       return "General_Physician";
13     } }
14 % génération de l'interface Remote du composant
15 package packageMedicalManagementSystem;
16 import javax.ejb.Remote;
17 @Remote
18 publicinterface General_PhysicianRemote{
19   String getClassName();
20   }
21 % génération de l'interface Locale du composant
22
23 package packageMedicalManagementSystem;
24 import javax.ejb.Local;
25 @Remote
26 publicinterface General_PhysicianLocal{
27   String getClassName();
28   }
```

6.6.5.2 Génération du code des aspects

Nous avons de même intégré les templates de génération d'aspects dans l'outil SCRITOOL (figure 6.23). En appliquant par exemple le template relatif au patron RBAC, nous obtenons le code aspect suivant présenté par le code ci-dessous.

```
1 publicaspect General_PhysicianSecurityAspect {
2 pointcut SecurityGeneral_PhysicianAspectPointcut() :
3 call(void packageMedicalManagementSystem.General_PhysicianRemote.create
      (..));
4 after() : SecurityGeneral_PhysicianAspectPointcut() {
5   % TODO Auto-generated method stub}
6 before() : SecurityGeneral_PhysicianAspectPointcut() {
7   % TODO Auto-generated method stub}}
```

Une fois les différents aspects définis, ils doivent être intégrés automatiquement au code métier pour construire l'application. Dans ce qui suit, nous allons présenter le tissage de ces aspects à travers un exemple.

6.6.5.3 Tissage d'aspects et tests

Après l'étape de génération de code, le développeur doit utiliser un tisseur d'aspect afin de voir le résultat final de l'intégration des patrons de sécurité sur l'application développée. Puisque notre outil génère le code fonctionnel et les aspects de sécurité en langage java le choix s'est porté sur le tisseur d'aspect AspectJ basé sur ce même langage. La figure 6.24 illustre cette étape. Nous obtenons donc un code sécurisé dont le comportement et la structure sont étendus par les aspects de sécurité générés par notre plug-in. Dans ce qui suit, nous testons ce comportement.

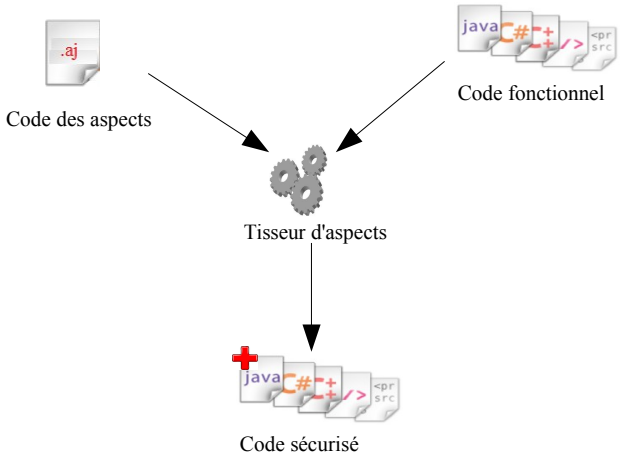


Figure 6.24 — Principe du mécanisme de tissage

Pour pouvoir tester le code de l'application généré, il faut tout d'abord compléter le

développement du code fonctionnel par les spécificités de l'étude de cas avec des méthodes permettant de tester l'accès des différents utilisateurs aux entités protégé pour la création, la lecture, l'écriture et la modification des données relatif à un Dossier Médical Personnel. Nous avons ainsi développé une interface permettant de tester (figure 6.25) ces différentes fonctionnalités. Selon le fichier de configuration de sécurité présenté précédemment,l'utilisateur <General_Physician> a tous les droits d'accès, par exemple, sur l'entité <Current_Treatement>, tandis que l'utilisateur <Nurse> possède seulement le droit de lecture pour cette même entité.

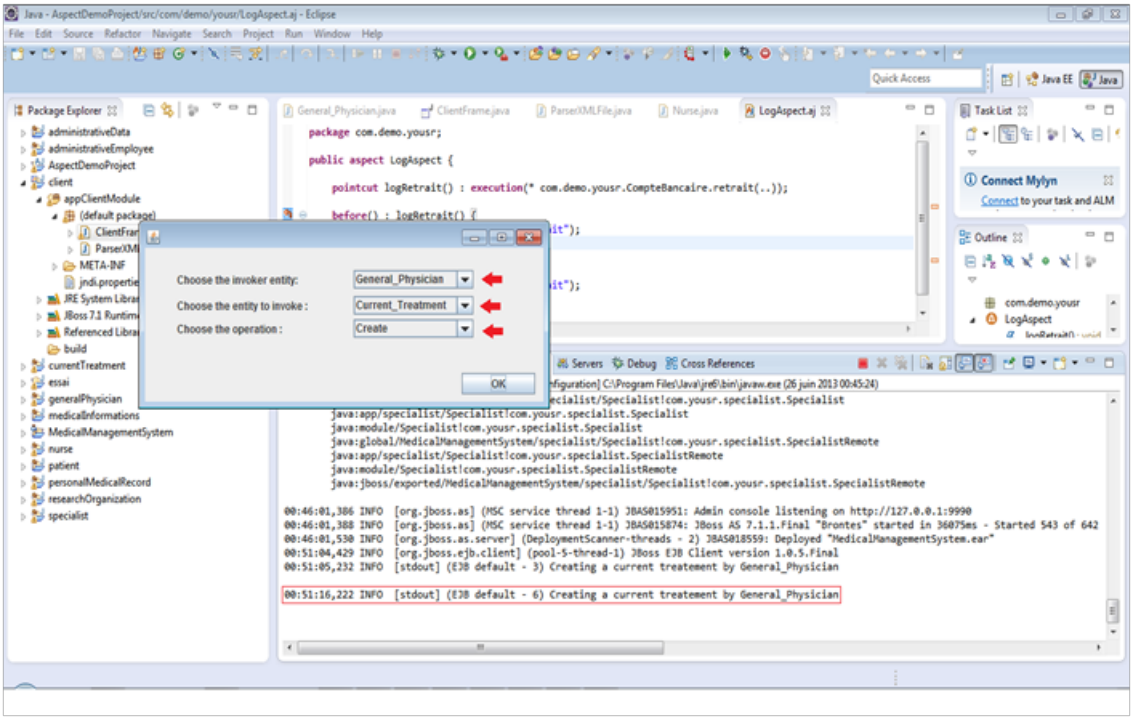


Figure 6.25 — Premier cas de test de l'étude de cas

Nous choisissons l'utilisateur, l'entité à invoquer et le type d'accès à cette entité. Dans notre cas nous choisissons d'abord l'utilisateur <General_Physician>, l'entité <Current_Treatement> et le type d'accès "Create". Cette méthode est exécutée avec sucées car l'utilisateur possède les droits de création. Nous passons maintenant au test de cette opération pour l'utilisateur<Nurse> pour le même type d'accès "Create". La figure 6.26 montre que la méthode n'est pas exécutée et l'accès est refusé.Ceci est conforme à l'information présente dans le fichier de configuration de sécurité. De la même façon, les accès pour la modification est aussi impossible pour l'entité <Nurse>.

6.7 Conclusion

Dans ce chapitre, nous avons présenté l'implémentation et l'expérimentation de l'atelier qui permet de mettre ?uvre l'approche présentée dans le chapitre précédent sur l'intégration des patrons de sécurité dans une application à base de composants.

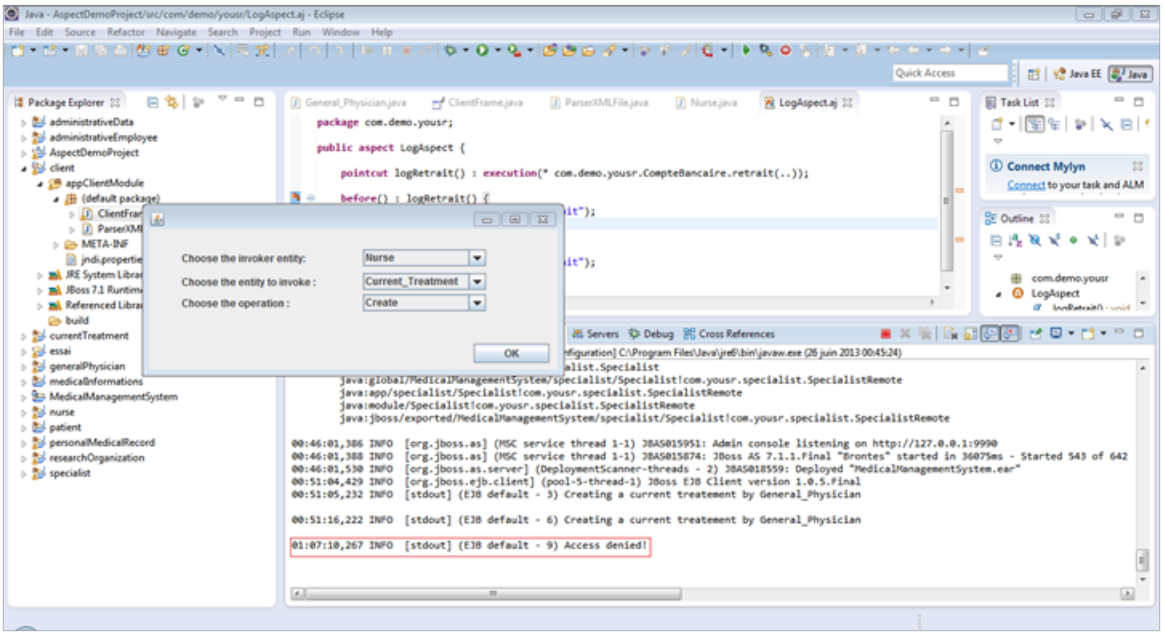


Figure 6.26 — Deuxième cas de test de l'étude de cas

Cette réalisation a profité des avantages offerts par l'approche IDM, en termes d'automatisation de certaines tâches de développement. De plus, l'environnement Eclipse nous a permis de minimiser le temps nécessaire pour mettre en ?uvre l'outil SCRI-TOOL. En effet, la plupart des outils utilisés sont supportés par Eclipse. Par exemple, pour la transformation de modèles, nous avons utilisé le langage ATL, et le framework UML2 a été utilisé pour la manipulation des modèles conformes au métamodèle UML 2 .0.

Nous avons détaillé dans ce chapitre le scénario de fonctionnement de notre atelier (transformation, manipulation de modèles, etc.). Les caractéristiques principales des transformations ATL que nous avons développées ont ensuite été présentées. Enfin, nous avons validé le prototype implémenté sur un exemple d'application à base de composants, celui d'un système de soin et particulièrement la gestion du dossier médical. Nous avons présenté et commenté des impressions d'écran qui illustrent les résultats produits par l'outil implémenté. Toutefois, il devient intéressant et pertinent de tester et d'expérimenter notre atelier sur un périmètre plus élargi.

Des améliorations peuvent être envisagées lors de l'implémentation de cet outil afin de supporter l'ensemble des éléments de modélisation des solutions des patrons de sécurité (fragments dynamiques en particulier). D'autres fonctionnalités peuvent ainsi être rajoutés comme la manipulation de la cartographie des patrons de sécurité et l'automatisation de l'élaboration des profiles relatifs au politiques de sécurité. D'autres améliorations peuvent être envisagées comme le développement d'autres templates de génération de code ciblant d'autres plateformes. Le chapitre suivant 7 conclue ce manuscrit et présente des perspectives.

Quatrième partie

Conclusion générale et perspectives

7 Conclusion et perspectives

7.1 Conclusion

L'approche à base de composants permet de concevoir des applications logicielles flexibles, modulaires et réutilisables. Le travail proposé dans le cadre de ce manuscrit consiste à sécuriser une application basé sur les composants en utilisant les patrons de sécurité.

Comme présenté dans la partie état de l'art, et dans la plus part des approches existantes (voir le chapitre 4), la sécurité est généralement intégrée à la fin du processus de développement des systèmes. Cette façon de faire manque de flexibilité puisque la sécurité devient étroitement liée au code fonctionnel des applications et ne peut être configurée indépendamment. En effet, cette intégration tardive rend difficile la manipulation ou la modification des exigences de sécurité [Kanneganti et Chodavarapu, 2008]. Dans ce travail, nous proposons d'intégrer la sécurité dés les premières phases de développement d'une application en utilisant la notion des patrons de sécurité en fonction des besoins et des exigences du système. En particulier, nous nous sommes intéressés aux applications à base de composants. Dans ce travail de thèse, nous avons proposé un processus complet pour sécuriser les applications orientées composants. Ce processus est basé sur l'approche MDA. En effet ce choix nous a permis d'appliquer dynamiquement les patrons de sécurité de la phase de modélisation jusqu'à la génération de code.

Dans la première partie de ce travail, nous avons dressé un état de l'art qui nous a permis de nous positionner par rapport à l'existant. Dans cette première étape, nous nous sommes focalisés en particulier sur la modélisation à base de composants. Nous avons présenté les différents concepts relatifs à cette approche ainsi que les modèles académiques et industrielles existants. Dans ce sens nous avons illustré en détail ce paradigme à travers le modèle UML 2.0 sur le quel nous nous sommes basé dans le cadre de ce travail de thèse. Dans un second temps nous avons étudié la notion de patron et particulièrement les patrons de sécurité. Nous avons présenté ainsi une cartographie des principaux patrons de sécurité proposés dans la littérature. Et nous nous sommes focalisé en particulier sur les patrons traitants les problèmes liés aux droits d'accès. Cette cartographie sera en effet utilisée lors de l'implémentation de l'approche en permettant une sélection guidée des patrons à utiliser. L'objectif de ce travail est de proposer un processus basé sur l'IDM. Dans cet esprit nous nous sommes intéressés en particulier à présenter les principaux concepts et notions

liés à l'IDM. Afin d'assurer la séparation entre le code fonctionnelle des applications et les propriétés non fonctionnelles à rajouter (la sécurité dans notre cas), nous avons choisi l'utilisation de la programmation par aspects pour assurer cette séparation. Dans cette même partie nous avons donc présenté les principaux avantages de cette approche.

Nous avons aussi présenté, d'une manière non exhaustive, les principaux travaux déjà existants visant à sécurisé des applications. Ces travaux et comme déjà présentés (pointeur vers section travaux connexes) restent limités par le manque de supports et ne se sont pas intéressé, comme dans le cadre de se travail, à l'utilisation des patrons de sécurité pour des applications à base de composants.

En se basant sur cet état de l'art et l'ensemble des travaux connexes présentés, nous avons présenté le processus SCRIP. Il s'agit d'un processus IDM permettant la prise en compte des aspects relatif à la sécurité dans les applications basées sur les composants. La sécurité est donc intégrée à travers les patrons dès les premières phases de conception. Cette application passe par la définition d'un ensemble de règles et de transformations. Nous utilisons ainsi la notion de profil pour étendre les modèles UML. En effet nous définissons pour chaque politique de sécurité un profil UML correspondant... Ainsi et à travers un ensemble de règle de transformation, nous assurons l'intégration dynamique de ces notions dans le modèle de l'application. A la phase de modélisation, les règles d'intégration sont appliquées semi-automatiquement après le choix de la politique à appliquer puis le choix du patron approprié afin de produire un modèle à composants sécurisé.

Le processus présenté définit explicitement une méthodologie permettant la prise en compte des solutions proposées par les patrons dans les modèles à base de composants. Ce processus couvre les différentes phases du processus de développement.

Ce processus présente l'avantage de la séparation entre l'expertise du domaine d'application et l'expertise en matière de sécurité. L'inclusion de la sécurité au cours du processus de développement de logiciels devient donc plus pratique pour les architectes/designers. L'approche que nous proposons reste relativement simple et parfaitement utilisable par des non experts en sécurité. Comprendre les patrons de sécurité à partir de leur description lisible et avoir une connaissance sur les applications à base de composants sont les compétences suffisantes pour utiliser ce processus.

Une fois définis, nous avons formalisé le processus SCRIP avec le formalisme SPEM. Cela nous a permis de vérifier l'applicabilité et la cohérence du processus à travers la cohérence des différentes étapes ainsi que les différents inputs et outputs.

La troisième partie de ce manuscrit présente l'implémentation et la mise en ?uvre du processus présenté dans la deuxième partie. Un premier prototype nommé SCRI-TOOL supportant la démarche d'intégration des patrons de sécurité, allant de la modélisation jusqu'à la génération de code a été réalisé. Ce prototype, opérationnel, a été développé sous Eclipse. Le choix du langage d'implantation des règles s'est porté sur le langage ATL. Ce choix nous a permis de coder la plupart des règles sous une forme déclarative.

Ce travail offre aux développeurs non experts en sécurité la possibilité d'exploiter de façon pertinente les patrons de sécurité lors du développement de leurs applications. De plus l'utilisation du plug-in proposé ne nécessite pas de connaissances spécifiques.

Le prototype proposé se présente sous forme d'un plug-in qui peut être installé sous l'environnement de développement Eclipse. Le concepteur ayant déjà le modèle à base de composants de son application, peut sélectionner dans un premier temps le patron de sécurité qu'il souhaite intégrer. En lançant le processus, la politique de sécurité encapsulée par ce patron sera intégrée dans le modèle à composants. Comme résultat de cette application, le prototype nous propose un modèle à composants sécurisé intégrant les solutions proposées par le patron. Cette application peut être faite de manière itérative en sélectionnant à chaque fois les patrons adéquats selon les besoins de l'application.

Lors de la génération de code, et dans le but de garder la séparation entre les propriétés fonctionnelles de l'application à modéliser et la sécurité, nous avons opté pour les technologies aspects. Ce choix nous a permis d'intégrer proprement la sécurité au niveau code tous en assurant cette séparation. Les aspects sont codés en Java et sont intégrés au code fonctionnel en utilisant un tisseur d'aspects, AspectJ dans notre cas.

Afin de valider le prototype proposé, nous avons appliqué l'approche proposée à travers l'outil SCRI-TOOL sur divers exemples. Cette application nous a permis dans un premier temps de valider les différents choix techniques réalisés comme l'utilisation du langage ATL pour coder les règles de transformation ou bien le choix des aspects pour la génération du code. Nous avons ainsi présenté un cas d'étude global dans la troisième partie de ce manuscrit. Pour cela nous avons choisi un système de gestion des soins et particulièrement la gestion des dossiers médicaux personnels.Le choix de ce cas d'étude s'explique par l'importance ainsi que les différentes exigences en terme de sécurité requise pour le bon fonctionnement d'une telle application.

7.2 Perspectives

Le travail décrit dans cette thèse a permis la mise en place d'un processus IDM pour l'intégration des patrons de sécurité dans une architecture à base de composants. Dans cette thèse, nous avons cherché à couvrir l'ensemble des étapes de développement logiciel, allant de la modélisation à la génération de code.

Nous avons proposé un processus d'élaboration de profils UML à partir des patrons de sécurité. Ce travail laisse entrevoir plusieurs perspectives. A court terme, il serait possible de définir des règles permettant de (semi)-automatiser l'élaboration de ces profils. Puisque, dans les patrons de sécurité plusieurs participants partagent la même sémantique et dans un but de réutilisation ces règles peuvent être factoriser pour définir de nouvelles règles.

7.2.1 Vers un méta-modèle de composants générique

L'approche proposée est illustré dans un premier temps en utilisant le modèle à composant UML 2.0. L'utilisation d'autres modèles à composants reste parmi les perspectives intéressantes à traiter. Cela permettra de généraliser l'approche et de la rendre plus universelle. En effet pouvoir intégrer les patrons de sécurité dans différents modèles architecturaux comme les modèles multi-agents ou orientées services est parmi les perspectives intéressantes qui s'ouvrent à nous.

Une première réflexion consiste à proposer un cadre générique pour concevoir une application à base de composants, c'est-à-dire proposer un méta-modèle générique, complet et extensible qui couvre l'ensemble des modèles déjà proposés. L'idée principale est de proposer une représentation commune pour cibler plusieurs modèles à base de composants existants (par exemple CCM, Fractal, etc.). L'avantage principal de proposer un modèle de composants générique est d'avoir un modèle indépendant de toute technologie pour pouvoir utiliser (exporter) les applications générées dans n'importe quel environnement quelle que soit la technologie utilisée. Ce premier travail a déjà fait l'objet d'une présentation dans une conférence internationale. Nous pensons à court terme proposer un éditeur sous Eclipse permettant la représentation de modèles en se basant sur le méta-modèle proposé.

7.2.2 Couvrir d'autres propriétés non fonctionnelles

Un des thèmes sous-jacents à notre thèse et qui nous semble particulièrement important à approfondir est la traçabilité des choix de politiques et aussi des patrons de sécurité appliqué à une même application.Couvrir d'autres politiques de sécurité et d'autres propriétés non-fonctionnelles comme la sûreté de fonctionnement ou la tolérance aux fautes est aussi parmi nos perspectives. Cela nous amène à enrichissement de la cartographie des patrons de sécurité, afin de «couvrir» la plupart voir toutes les politiques de sécurité. En utilisant l'arbre de décision, et selon la politique de sécurité choisie, des patrons pré-sélectionnés peuvent être proposés au concepteur afin de faciliter la sélection des patrons appropriés à partir d'un nombre potentiellement énorme des patrons de sécurité.

7.2.3 Vers un atelier logiciel plus riche

L'outil devrait être en mesure de faire des suggestions précises de patrons de sécurité lors du développement d'une application utilisant la dépendance entre les patrons. L'implémentation de cette phase de prédiction est en cours. Le processus proposé est en adéquation avec les besoins et les objectifs fixés. L'illustration à travers l'étude de cas met en évidence l'efficacité d'une telle approche. Cependant, il serait aussi intéressant de vérifier d'une façon formelle l'étape de transformation appliquée et de s'assurer que le modèle en sortie reste en adéquation avec le modèle en entrée en ajoutant seulement les contraintes de sécurité. Dans cet esprit beaucoup de travaux peuvent être réutilisés comme [Hatebur *et al.*, 2007] [Robinson, 2007].

Cinquième partie

Bibliographie et Annexes

Bibliographie personnelle

Rahma BOUAZIZ, Bernard COULETTE : An MDE Approach for Domain based Architectural Components Modelling. *In Proceedings of the 18th IEEE symposium on Computers and Communication , Split, Croatia*, :2013, IEEE Computer Society.

Rahma BOUAZIZ : GRIMACE : GeneRIc MetAmodel for domain Component modEling. *Dans European Workshop on Dependable Computing, Coimbra, Portugal*, p.181-184, 2013, Springer, LNCS.

Rahma BOUAZIZ, Slim KALLEL Bernard COULETTE : An engineering process for security patterns application in component based models. *Dans IEEE International Workshop on Enabling Technologies : Infrastructure for Collaborative Enterprises, Hammamet*, p. 231-236, 2013, IEEE Computer Society.

Rahma BOUAZIZ, Bernard COULETTE : Secure Component Based Applications Through Security Patterns. *Dans Workshop on Security of Systems and Software Resiliency, Besançon*, p. 749-754, 2012, IEEE Computer Society.

Rahma BOUAZIZ, Bernard COULETTE : Applying Security Patterns for Component Based Applications Using UML profile. *Dans International Conference on Computational Science and Engineering, Paphos*, p. 186-193 , 2012, IEEE Computer Society.

Rahma BOUAZIZ, Brahim HAMID, NicolasDESNOS : Towards a Better Integration of Patterns in Secure Component-Based Systems Design. *Dans International Conference on Computational Science and Applications, University of Cantabria*, Vol.5, p.607-621, 2011, Springer.

Rahma BOUAZIZ, Brahim HAMID, NicolasDESNOS : DDroits d'accès aux services offerts par des composants logiciels– Profil UML à base de patrons de sécurité. *Dans Sécurité des Systèmes d'Information et les Environnements Collaboratifs - INFORSID 2010, Marseilles*, p.13-25, mai 2010.

Rahma BOUAZIZ, Brahim HAMID, NicolasDESNOS : Droits d'accès aux services offerts par des composants logiciels. *Dans Génie Logiciel, GL & IS, Meudon, ,* Vol.94, p.6-11, 2010.

Bibliographie

Integrating Security and Software Engineering : Advances and Future Vision, chapitre 5, page 107 ?126. Idea Group Publishing, 2006.

J.P. Giraudin D. Rieu A. CONTE, M. Fredj : P-sigma : un formalisme pour une représentation unifiée de patrons. *Dans In INFORSID'01 Congres*, pages 554–563, 2001.

Anas ABOU EL KALAM : *Modèles et Politiques de sécurité pour les domianes de la santé et des affaires sociales, school = Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique, year = 2003,*. Thèse de doctorat.

Jenny ABRAMOV, Arnon STURM et Peretz SHOVAL : A pattern based approach for secure database design. *Dans* Camille SALINESI et Oscar PASTOR, éditeurs : *CAiSE Workshops*, volume 83 de *Lecture Notes in Business Information Processing*, pages 637–651. Springer, 2011. ISBN 978-3-642-22055-5. URL http://dblp.uni-trier.de/db/conf/caise/caisews2011.html#AbramovSS11.

ACCELEO : Acceleo : générateur mda [online], consulté en mars 2013. URL http://www.acceleo.org/pages/accueil/fr.

Masoom ALAM, Ruth BREU et Michael HAFNER : Model-Driven Security Engineering for Trust Management in SECTET, 2009. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.94.2395.

Christopher ALEXANDER : *The Timeless Way of Building*. Oxford University Press, later printing édition, 1979. ISBN 0195024028.

Christopher ALEXANDER, Sara ISHIKAWA et Murray SILVERSTEIN : *A Pattern Language : Towns, Buildings, Construction (Cess Center for Environmental)*. Oxford University Press, later printing édition, août 1977. ISBN 0195019199. URL http://downlode.org/etext/patterns/.

S. W. AMBLER : An introduction to process patterns, available online at. URL http://www.ambysoft.com/downloads/processPatterns.pdf. 1998a.

Scott W. AMBLER : *Process patterns : building large-scale systems using object technology*. Cambridge University Press, New York, NY, USA, 1998b. ISBN 0-521-64568-9.

ANDROMDA : Andromda model driven architecture framework [online], consulté en mars 2013. URL http://www.andromda.org/index.html.

Entreprise ARCHITECT : Entreprise architect website [online], consulté en juin 2013. URL http://www.sparxsystems.com/.

Nicolas ARNAUD : *Fiabiliser la réutilisation des patrons par une approche orientée complétude, variabilité et généricité des spécifications*. These, Université Joseph-Fourier - Grenoble I, octobre 2008. URL http://tel.archives-ouvertes.fr/tel-00331750.

ASPECTJ : Aspectj : Tisseur d'aspects [online], consulté en mars 2013. URL http://eclipse.org/aspectj/.

Colin ATKINSON et Thomas KÜHNE : Model-driven development : A metamodeling foundation. *IEEE Softw.*, 20(5):36–41, septembre 2003. ISSN 0740-7459. URL http://dx.doi.org/10.1109/MS.2003.1231149.

David BASIN, Jürgen DOSER et Torsten LODDERSTEDT : Model driven security : From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, janvier 2006. ISSN 1049-331X.

Kent BECK et Ward CUNNINGHAM : Using pattern languages for object oriented programs. *Dans Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987.

Bastian BEST, Jan JURJENS et Bashar NUSEIBEH : Model-based security engineering of distributed information systems using umlsec. *Dans Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 581–590, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7.

Jean BÉZIVIN : In Search of a Basic Principle for Model Driven Engineering. *UPGRADE – The European Journal for the Informatics Professional*, 5(2):21–24, 2004b.

Jean BÉZIVIN et Olivier GERBÉ : Towards a precise definition of the omg/mda framework. *Dans Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=872023.872565.

Eric BRUNETON, Thierry COUPAYE, Matthieu LECLERCQ, Vivien QUÉMA et Jean-Bernard STEFANI : The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, septembre 2006. ISSN 0038-0644. URL http://dx.doi.org/10.1002/spe.v36:11/12.

Tomás BURES, Petr HNETYNKA et Frantisek PLASIL : Sofa 2.0 : Balancing advanced features in a hierarchical component model. *Dans SERA*, pages 40–48, 2006.

Frank BUSCHMANN, Regine MEUNIER, Hans ROHNERT, Peter SOMMERLAD et Michael STAL : *Pattern-oriented software architecture : a system of patterns*. John Wiley & Sons, Inc., 1996. ISBN 0471958697.

Jean BÉZIVIN : La transformation de modèles, cours-6. 2003. URL unemmaroc.free.fr/cours/Uml/Cours06.v1-01.ppt.

Jean BÉZIVIN : Sur les principes de base de l'ingénierie des modèles. *L'OBJET*, 10(4):145–157, 2004a. URL http://dblp.unitrier.de/db/journals/Lobjet/Lobjet10.html#Bezivin04.

Jean BÉZIVIN : Model-driven engineering : Principles, scope, deployment and applicability. *Dans In Proceedings of 2005 Summer School on Generative and Transformation Techniques in Software Engineering*, 2005.

CCM : Corba component model , v4.0, avril 2012. URL http://www.omg.org/spec/CCM/.

P. COAD, D. NORTH et M. MAYFIELD : *Object models : strategies, patterns and applications*. Prentice Hall, 1995.

Peter COAD : Object-oriented patterns. *Commun. ACM*, 35(9):152–159, 1992.

Benoit COMBEMALE : *Approche de métamodélisation pour la simulation et la vérification de modèle*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, juillet 2008. URL http://ethesis.inp-toulouse.fr/archive/00000666/,http://www.combemale.fr/research/phd/phd-2008-combemale-finalversion.pdf.

James O. COPLIEN : Software design patterns : Common questions and answers. *Dans In : Rising L., (Ed.), The Patterns Handbook : Techniques, Strategies, and Applications*, pages 311–320. University Press, 1998.

Krzysztof CZARNECKI et Simon HELSEN : Classification of model transformation approaches. *Dans Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003.

Samba DIAW : *SPEM4MDE : Un métamodèle et un environnement pour la modélisation et la mise en œuvre assistée de processus IDM*. Thèse de doctorat, Université de Toulouse, France, Septembre 2011.

H. Washizaki an d J. Jurjens E. B. FERNANDEZ, N. Yoshioka : Using security patterns to build secure systems. *Dans Proceedings of the 1st International Workshop on Software Patterns and Quality*, SPAQu'07, 2007.

ECLIPSE : Eclipse - the eclipse foundation open source community website. URL http://www.eclipse.org/.

EJB : Enterprise java bean. URL http://java.sun.com/products/ejb/.

Paul ELKHOURY : *Security Patterns, their Applicability, Retrieval and Integration, school = SAP Recherche Security and Trust - SAP Labs France, Sophia Antipolis - Université Claude Bernard Lyon 1, France, year = 2009a,*. Thèse de doctorat.

Paul ELKHOURY, Pierre BUSNEL, Sylvain GIROUX et Keqin LI : Enforcing security in smart ho mes using security patterns. *International Journal of Smart Home*, 2009b.

W ESSMAYR, G. PERNUL et A.M. TJOA : Access controls by object-oriented concepts. *Dans 11th IFIP WG 11.3 Working Conf. on Database Security*, 1997.

Eduardo B. Fernández FERNÁNDEZ, María M. LARRONDO-PETRIE et Ehud GUDES : A method-based authorization model for object-oriented databases. *Dans Security for Object-Oriented Systems*, pages 135–150, 1993.

Eduardo B. FERNÁNDEZ, Hironori WASHIZAKI, Nobukazu YOSHIOKA, Atsuto KUBO et Yoshiaki FUKAZAWA : Classifying security patterns. *Dans* Yanchun ZHANG, Ge YU, Elisa BERTINO et Guandong XU, éditeurs : *Progress in WWW Research and Development, 10th Asia-Pacific Web Conference, APWeb 2008, Shenyang, China, April 26-28, 2008. Proceedings*, volume 4976 de *Lecture Notes in Computer Science*, pages 342–347. Springer, 2008. ISBN 978-3-540-78848-5.

David FERRAIOLO et Richard KUHN : Role-based access control. *Dans In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

Martin FOWLER : *Analysis Patterns : Reusable Object Models*. Addison-Wesley Professional, 1 édition, octobre 1996. ISBN 0201895420.

Robert FRANCE et Bernhard RUMPE : Model-driven Development of Complex Software : A Research Roadmap. *Dans FOSE '07 : 2007 Future of Software Engineering*, pages 37–54, Washington, DC, États-Unis, 2007. IEEE Computer Society. URL http://hal.inria.fr/inria-00511368.

Andreas FUCHS, Sigrid GÜRGENS et Carsten RUDOLPH : Towards a generic process for security pattern integration. *Dans DEXA Workshops*, pages 171–175, 2009.

Eric GAMMA : *Object-Oriented Software Development Based on ET++ : Design Patterns, Class Library, Tools (in German)*. Thèse de doctorat, University of Zurich, Institut für informatik, 1991.

Erich GAMMA, Richard HELM, Ralph JOHNSON et John VLISSIDES : *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Geri GEORG, Indrakshi RAY et Robert FRANCE : Using aspects to design a secure system. *Dans Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, ICECCS '02, pages 117–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1757-9. URL http://dl.acm.org/citation.cfm?id=857175.857337.

Frédéric GILLIERS : *Développement par prototypage et Génération de Code à partir de LfP, un langagede modélisation de haut niveau*. Thèse de doctorat, Université Pierre et Marie Curie, France, Septembre 2005.

Catherine GRIFFIN : Transformations in eclipse. *Dans Workshop on Model-Driven Development WMDD / IBM / ATHENA Consortium 2004*, 2004.

Carlos GUTIÉRREZ, Eduardo FERNÁNDEZ-MEDINA et Mario PIATTINI : Web services enterprise security architecture : a case study. *Dans* Ernesto DAMIANI et Hiroshi MARUYAMA, éditeurs : *SWS*, pages 10–19. ACM, 2005. ISBN 1-59593-234-8.

Ouafa HACHAN : *Patrons de conception à base d'aspects pour l'ingén ierie des systèmes d'information par réutilisation*. Thèse de doctorat, Université Joseph Fourrier -Gronoble I, 2006.

Ouafa HACHANI : Apport de la programmation par aspects dans l'implantation des patrons de conception par objets. Mémoire de D.E.A., Université Joseph Fourier, 2002.

Munawar HAFIZ, Paul ADAMCZYK et Ralph E. JOHNSON : Organizing Security Patterns. *IEEE Softw.*, 24:52–60, juillet 2007. URL http://dl.acm.org/citation.cfm?id=1435593.1435870.

M. HAFNER : *SECTET : A Domain Architecture for Model Driven Security*. Thèse de doctorat, University of Innsbruck, 2006.

M. HAFNER, M. BREU, R. BREU et A. NOWAK : Modelling inter-organizational workflow security in a peer-to-peer environment. *Dans Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages –540, 2005b.

Michael HAFNER et Ruth BREU : *Security Engineering for Service-Oriented Architectures*. Springer Publishing Company, Incorporated, 1 édition, 2008. ISBN 3540795383, 9783540795384.

Michael HAFNER, Ruth BREU, Berthold AGREITER et Andrea NOWAK : Sectet - an extensible framework for the realization of secure inter-organizational workflows. *Dans* Eduardo FERNÁNDEZ-MEDINA et Mariemma Inmaculada Yagüe del VALLE, éditeurs : *WOSIS*, pages 47–57. INSTICC Press, 2006.

Michael HAFNER, Ruth BREU et Michael BREU : A security architecture for inter-organizational workflows : Putting security standards for web services together. *Dans* Chin-Sheng CHEN, Joaquim FILIPE, Isabel SERUCA et José CORDEIRO, éditeurs : *ICEIS (3)*, pages 128–135, 2005a.

Denis HATEBUR, Maritta HEISEL et Holger SCHMIDT : A security engineering process based on patterns. *Dans Proceedings of the 18th International Conference on Database and Expert Systems Applications*, DEXA '07, pages 734–738, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2932-1. URL http://dx.doi.org/10.1109/DEXA.2007.23.

George T. HEINEMAN et William T. COUNCILL, éditeurs. *Component-based software engineering : putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70485-4.

Viktor HORVATH et Till DÖRGES : From security patterns to implementation using petri nets. *Dans* Bart De WIN, Seok-Won LEE et Mattia MONGA, éditeurs : *SESS*, pages 17–24. ACM, 2008a.

Viktor HORVATH et Till DÖRGES : From security patterns to implementation using petri nets. *Dans SESS*, pages 17–24. ACM, 2008b. ISBN 978-1-60558-042-5. URL http://dblp.uni-trier.de/db/conf/icse/sess2008.html#HorvathD08.

IBM : rational rose website [online], consulté en juin 2013. URL http://www-01.ibm.com/software/awdtools/developer/rose/.

IHEA : Ihe : Integrating the healthcare enterprise [online], consulté en septembre 2013. URL http://www.ihe.net.

Jean M. JÉZÉQUEL, Sébastien GÉRARD, Chokri MRAIDHA et Benoit BAUDRY : *L'ingénierie dirigée par les modèles au dela du MDA*, chapitre Le génie logiciel et l'IDM : une approche unificatrice par les modèles, pages 35–52. Numéro ISBN 2-7462-1213-7. Hermes, 2006.

Frédéric JOUAULT et Ivan KURTEV : On the architectural alignment of atl and qvt. *Dans Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1188–1195, New York, NY, USA, 2006a. ACM. ISBN 1-59593-108-2.

Frédéric JOUAULT et Ivan KURTEV : Transforming models with atl. *Dans Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Berlin, Heidelberg, 2006b. Springer-Verlag. ISBN 3-540-31780-5, 978-3-540-31780-7. URL http://dx.doi.org/10.1007/11663430_14.

Jan JUERJENS : Umlsec : Extending uml for secure systems development. *Dans Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44254-5. URL http://dl.acm.org/citation.cfm?id=647246.719625.

Jan JUERJENS : *Secure Systems Development with UML*. SpringerVerlag, 2003. ISBN 3540007016.

Jan JURJENS, Gerhard POPP et Guido WIMMEL : Towards using security patterns in model-based system development, 2002.

R. KANNEGANTI et P. CHODAVARAPU : *SOA Security*. Manning Pubs Co Series. Manning Publications Company, 2008. ISBN 9781932394689. URL http://books.google.tn/books?id=BpnmAAAACAAJ.

Stuart KENT : Model driven engineering. *Dans Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43703-7. URL http://dl.acm.org/citation.cfm?id=647983.743552.

Gregor KICZALES, John LAMPING, Anurag MENDHEKAR, Chris MAEDA, Cristina LOPES, Jean marc LOINGTIER et John IRWIN : Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

Anneke G. KLEPPE, Jos WARMER et Wim BAST : *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003a. ISBN 032119442X.

Anneke G. KLEPPE, Jos WARMER et Wim BAST : *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003b. ISBN 032119442X.

Jos Warmer KLEPPE ANNEKE : Do mda transformations preserve meaning ? an investigation into preserving semantics. *Dans Proceedings of the First International Workshop on Metamodelling for MDA*, UK, 2002.

U. LANG et R. SCHREINER : *Developing Secure Distributed Systems with CORBA*. Artech House computer security series. Artech House, 2002. ISBN 9781580535618. URL http://books.google.tn/books?id=J1-qNbp3y0kC.

J.C. LAPRIE : *Le guide de la sûreté de fonctionnement. 2ème édition*. Cépaduès-Editions, 1996. ISBN 9782854283822. URL http://books.google.fr/books?id=GuSVAAAACAAJ.

C. LARMAN : *UML et les Design Patterns*. CampusPress Référence. CampusPress, 2002. ISBN 9782744013010.

Torsten LODDERSTEDT, David A. BASIN et Jürgen DOSER : Secureuml : A uml-based modeling language for model-driven security. *Dans Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44254-5. URL http://dl.acm.org/citation.cfm?id=647246.719477.

R. Breu M. ALAM, M. Hafner et S. UNTERTHINER : A framework for modeling restricted delegation of rights in sectet. *International Journal of Computer Systems Science & Engineering*, 2007.

Tom MENS, Krzysztof CZARNECKI et Pieter Van GORP : A taxonomy of model transformation. *Dans Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl*. Electronic, 2005.

Haralambos MOURATIDIS, Paolo GIORGINI et Markus SCHUMACHER : Security patterns for agent systems. *Dans PROCEEDINGS OF THE EIGHT EUROPEAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS (EUROPLOP), IRSEE*, 2003.

OCL : Constraint language, version 2.3.1, janvier 2012. URL http://www.omg.org/spec/OCL/2.3.1/PDF/.

Model Driven Architecture OMG : "mda - the architecture of choice for a changing world", [online], consulté en mars 2013. 2013. URL http://www.omg.org/mda/.

Object Management Group OMG : Meta object facility (mof) 2.4.1 core specification. Août 2011. URL http://www.omg.org/spec/MOF/2.4.1/PDF.

OSOA : Service component architecture. URL http://www.oasis-opencsa.org/[ONLINE],consultéenAoût2013.

QVT : Meta object facility (mof) 2.0 query/view/transformation, v1.1. Janvier 2011. URL http://www.omg.org/spec/QVT/1.1/PDF/.

Indrakshi RAY, Robert FRANCE, Na LI et Geri GEORG : An aspect-based approach to modeling access control concerns. *Information and Software Technology*, 46:575–587, 2004.

Julia REZNIK, Tom RITTER, Rudolf SCHREINER et Ulrich LANG : Model driven development of security aspects. *Electronic Notes in Theoretical Computer Science*, 163(2):65–79, 2007.

Philip ROBINSON : Extensible security patterns. *2012 23rd International Workshop on Database and Expert Systems Applications*, 0:729–733, 2007. ISSN 1529-4188.

Fumiko SATOH, Nirmal MUKHI, Yuichi NAKAMURA et Shinichi HIROSE : Pattern-based policy configuration for soa applications. *Dans IEEE SCC (1)*, pages 13–20, 2008a.

Fumiko SATOH, Yuichi NAKAMURA, Nirmal MUKHI, Michiaki TATSUBORI et Kouichi ONO : Methodology and tools for end-to-end soa security configurations. *Dans SERVICES I*, pages 307–314, 2008b.

Fumiko SATOH, Yuichi NAKAMURA et Koichi ONO : Adding Authentication to Model Driven Security. *Web Services, IEEE International Conference on*, 0:585–594, 2006. URL http://dx.doi.org/10.1109/icws.2006.25.

Fumiko SATOH et Yumi YAMAGUCHI : Generic security policy transformation framework for ws-security. *Dans ICWS*, pages 513–520. IEEE Computer Society, 2007. URL http://dblp.uni-trier.de/db/conf/icws/icws2007.html#SatohY07.

Douglas C. SCHMIDT : Using design patterns to develop reusable object-oriented communication software. *Commun. ACM*, 38(10):65–74, octobre 1995. ISSN 0001-0782. URL http://doi.acm.org/10.1145/226239.226255.

Markus SCHUMACHER : *Security Engineering with Patterns : Origins, Theoretical Models, and New Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN 3540407316.

Markus SCHUMACHER, Eduardo FERNANDEZ-BUGLIONI, Duane HYBERTSON, Frank BUSCHMANN et Peter SOMMERLAD : *Security Patterns : Integrating Security and Systems Engineering (Wiley Software Patterns Series)*. John Wiley and Sons, mars 2006. ISBN 0470858842. URL http://www.worldcat.org/isbn/0470858842.

Ed SEIDEWITZ : What models mean. *IEEE Software*, 20(5):26–32, 2003.

Richard SOLEY : Model driven architecture (mda, white paper, draft 3.0. Rapport technique, OMG, Novembre 2000. URL http://www.omg.org/~soley/mda.html.

SPEM : Software and systems process engineering metamodel specification (spem) version 2.0. Avril 2008. URL http://www.omg.org/spec/SPEM/2.0/PDF.

SYSML : Systems modeling language, version 1.3. Juin 2012. URL http://www.omg.org/spec/SysML/1.3/PDF.

Clemens SZYPERSKI : *Component software : beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998. ISBN 0-201-17888-5.

M. TOUZI : *Aide à la conception de Système d'Information Collaboratifsupport de l'interopérabilité des entreprises*. Thèse de doctorat, Institut National Polytechnique de Toulouse, 2007.

Theodore TRYFONAS, Evangelos A. KIOUNTOUZIS et Angeliki POULYMENAKOU : Embedding security practices in contemporary information systems development approaches. *Inf. Manag. Comput. Security*, 9(4):183–197, 2001. URL http://dblp.uni-trier.de/db/journals/imcs/imcs9.html#TryfonasKP01.

UML : Unified modeling language, v2.4.1, août 2012. a. URL http://www.omg.org/spec/UML/2.4.1/.

Papyrus UML : Entreprise architect website [online], consulté en juillet 2013. b. URL http://www.papyrusuml.org/scripts/home/publigen/content/templates/show.asp?L=EN&P=55&vTicker=alleza&ITEMID=3.

Rob van OMMERING, Frank van der LINDEN, Jeff KRAMER et Jeff MAGEE : The koala component model for consumer electronics software. *Computer*, 33(3):78–85, mars 2000. ISSN 0018-9162. URL http://dx.doi.org/10.1109/2.825699.

Rayford B. VAUGHN, Jr., Ronda HENNING et Kevin FOX : An empirical study of industrial security-engineering practices. *J. Syst. Softw.*, 61(3):225–232, avril 2002. ISSN 0164-1212. URL http://dx.doi.org/10.1016/S0164-1212(01)00150-9.

José Luis VIVAS, José A. MONTENEGRO et Javier LOPEZ : Towards a business process-driven framework for security engineering with the uml. *Dans ISC*, pages 381–395, 2003.

Guido WIMMEL et Alexander K. WIPEINTNER : Extended description techniques for security engineering. *Dans* Michel DUPUY et Pierre PARADINAS, éditeurs : *Trusted Information : The New Decade Challenge, IFIP TC11 Sixteenth Annual Working Conference on Information Security (IFIP/Sec 01), June 11-13, 2001, Paris, France*, volume 193 de *IFIP Conference Proceedings*, page 469. Kluwer, 2001. ISBN 0-7923-7389-8.

Christian WOLTER, Michael MENZEL et Christoph MEINEL : Modelling security goals in business processes. *Dans In Modellierung 2008, volume P-127 of LNI*, pages 201–216. Köllen, 2008.

Christian WOLTER, Michael MENZEL, Andreas SCHAAD, Philip MISELDINE et Christoph MEINEL : Model-driven business process security requirement specification. *J. Syst. Archit.*, 55(4):211–223, avril 2009. ISSN 1383-7621.

Roel WUYTS et Stéphane DUCASSE : Composition languages for black-box components. *Dans First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, 2001.

Joseph YODER et Jeffrey BARCALOW : Architectural patterns for enabling application security. *Dans Proc. Fourth Conf. Pattern Languages of Programming (PLoP)*, 1997.

Joseph YODER et Jeffrey BARCALOW : Architectural patterns for enabling application security. *Dans PatternLanguages of Program Design*, volume 4. Addison-Wesley, 2000.

Tewfik ZIADI, Tewfik ZIADI, Bruno TRAVERSON, Bruno TRAVERSON, Jean-Marc JEZEQUEL et Jean marc JÉZÉQUEL : From a uml platform independent component model to platform specific component models. *Dans In International workshop in Software Model Engineering (WiSME02) at UML2002*, pages 200–2, 2002.

Walter ZIMMER : Relationships between design patterns. *Dans PATTERN LANGUAGES OF PROGRAM DESIGN*, pages 345–364. Addison-Wesley, 1994.

8 Annexe A : Sources ATL du module d'intégration des patrons

Transformation ATL d'application du patron RBAC sur un diagramme de composants

L'application d'un patron de sécurité sur un diagramme de composants passe par une succession de transformations ATL. Le nombre de transformations utilisées pour l'application d'un patron de sécurité change d'un patron à un autre. Dans cet annexe, nous présentons les transformations qui permettent l'application du patron RBAC sur un diagramme de composants. Pour ce faire, nous utilisons trois transformations ATL.

La première transformation applique les stéréotypes « RBAC.User » et « RBAC.ProtectionObject » sur les composants choisis par l'utilisateur (dans cet annexe nous appliquons le stéréotype « RBAC.User » sur un composant nommé <General_Physician> et le stéréotype « RBAC.ProtectionObject » sur un composant nommé <PersonalMedicalRecord>).

La deuxième transformation applique le stéréotype « RBAC.Right » sur les connecteurs qui relient les composants stéréotypés « RBAC.User » et « RBAC.ProtectionObject ».

La troisième transformation applique le stéréotype « RBAC.Role » sur les ports qui appartiennent à un composant stéréotypé « RBAC.User » et qui sont impliqués dans une connexion établi à travers un connecteur stéréotypé « RBAC.Right ».

? Première transformation module InitialRBACTransformation ; create OUT : UML2 from IN : UML2, IN1 : PRO ;

```
10
11 --Helper 'getStereotype'
12 --Return the stereotype from the applied profile
13 helper def: getStereotype(p: UML2!Profile, name: String): UML2!Stereotype
       =
14   p.ownedStereotype -> select(s | s.name = name) -> first();
15 --End Helper 'getStereotype'
16
17 --Helper 'hasStereotype'
18 --Return whether the element has the stereotype in the helper parameter
19 helper context UML2!Element def: hasStereotype(stereotype : String) :
       Boolean =
20   self.getAppliedStereotypes() -> collect(st | st.name) -> includes(
       stereotype);
21 --End Helper 'hasStereotype'
22
23 ----------------------------END HELPERS-----------------
```

```
--------------------------BEGIN RULES----------------
rule ElementImport {
   from
s : UML2!"uml::ElementImport" in IN
   to
t : UML2!"uml::ElementImport" (
        visibility <- s.visibility,
        alias <- s.alias,
       eAnnotations <- s.eAnnotations,
       ownedComment <- s.ownedComment,
       importedElement <- s.importedElement)}

rule Model {
  --Copy Model Bloc--
  from
    s: UML2!"uml::Model" in IN
  to
    t: UML2!"uml::Model" (
     --__xmiID__ <- s.__xmiID__,
     name <- s.name.debug('Model'),
     visibility <- s.visibility,

     --Operations on Model--
    do {
     --Reapply existant Profile--
for (p in s.getAllAppliedProfiles()) {t.applyProfile(p);}
     --End Reapply existant Profile--
t.applyProfile(UML2!Package.allInstances() -> select(s | s.name ='
    RBAC_Profile') -> first());
thisModule.entityProfile<-UML2!Package.allInstances() -> select(s | s.
    name = 'RBAC_Profile') -> first();}
   --End Operations on Model--}

rule Package {
   from s : UML2!"uml::Package" in IN (s.oclIsTypeOf(UML2!"uml::Package")
       )
   to t : UML2!"uml::Package" (
      name <- s.name,
      profileApplication <- s.profileApplication)}

rule Component {
  --Copy Component Bloc--
    from s : UML2!"uml::Component" in IN
    to t : UML2!"uml::Component" (
     -- __xmiID__ <- s.__xmiID__,
       name <- s.name.debug('Component'),
    --Operations on Component--
  do {
  --Reapply existant Stereotypes--
    for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
```

```
71   --End Reapply existant Stereotypes--
72   if((s.name='General_Physician')and(not s.hasStereotype('Authenticator.
         User')))
73  {t.applyStereotype(thisModule.getStereotype(thisModule.entityProfile,'
        RBAC.User'));}
74  if(s.name = 'PersonalMedicalRecord')
75  {t.applyStereotype(thisModule.getStereotype(thisModule.entityProfile,'
        RBAC.ProtectionObject'));}
76   }
77   --End Operations on Component--}
78
79  rule Connector {
80   --Copy Connector Bloc--
81     from s : UML2!"uml::Connector" in IN
82     to t : UML2!"uml::Connector" (
83        --__xmiID__ <- s.__xmiID__,
84        name <- s.name.debug('Connector'),
85     --Operations on Connector--
86     do {
87       --Reapply existant Stereotypes--
88       for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
89       --End Reapply existant Stereotypes--}
90     --End Operations on Connector--}
91
92  rule ConnectorEnd {
93   --Copy ConnectorEnd Bloc--
94     from s : UML2!"uml::ConnectorEnd" in IN
95     to t : UML2!"uml::ConnectorEnd" (
96        --__xmiID__ <- s.__xmiID__,
97        isOrdered <- s.isOrdered.debug('ConnectorEnd'),
98        isUnique <- s.isUnique,
99        eAnnotations <- s.eAnnotations,
100       ownedComment <- s.ownedComment,
101       upperValue <- s.upperValue,
102       lowerValue <- s.lowerValue,
103       role <- s.role,
104       partWithPort <- s.partWithPort)
105  --Copy ConnectorEnd Bloc-- }
106
107 rule Port {
108  --Copy Port Bloc--
109    from s : UML2!"uml::Port" in IN
110    to t : UML2!"uml::Port" (
111       --__xmiID__ <- s.__xmiID__,
112       name <- s.name.debug('Port'),
113       visibility <- s.visibility,
114    --Operations on Port--
115    do {
116      --Reapply existant Stereotypes--
117      for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
```

```
    --End Reapply existant Stereotypes--}
  --End Operations on Port--}
-------------------------END RULES-----------------
```

La deuxième transformation

```
create OUT : UML2 from IN : UML2, IN1 : PRO;

-------------------------BEGIN HELPERS--------------------

--Helper 'getStereotype'
--Return the stereotype from the applied profile
helper def: getStereotype(p: UML2!Profile, name: String): UML2!Stereotype
    =
 p.ownedStereotype -> select(s | s.name = name) -> first();
--End Helper 'getStereotype'

--Helper 'hasStereotype'
--Return whether the element has the stereotype in the helper parameter
helper context UML2!Element def: hasStereotype(stereotype : String) :
   Boolean =
 self.getAppliedStereotypes() -> collect(st | st.name) -> includes(
    stereotype);
--End Helper 'hasStereotype'
-------------------------END HELPERS----------------

-------------------------BEGIN RULES--------------
rule ElementImport {
   from
s : UML2!"uml::ElementImport" in IN
   to
t : UML2!"uml::ElementImport" (
         visibility <- s.visibility,
        alias <- s.alias,
        eAnnotations <- s.eAnnotations,
        ownedComment <- s.ownedComment,
        importedElement <- s.importedElement)}
rule Model {
 --Copy Model Bloc--
 from
   s: UML2!"uml::Model" in IN
 to
   t: UML2!"uml::Model" (
     --__xmiID__ <- s.__xmiID__,
     name <- s.name.debug('Model'),
     --Operations on Model--
   do {
     --Reapply existant Profile--
for (p in s.getAllAppliedProfiles()) {t.applyProfile(p);}
     --End Reapply existant Profile--
t.applyProfile(UML2!Package.allInstances() -> select(s | s.name ='
```

```
    RBAC_Profile') -> first());
thisModule.entityProfile<-UML2!Package.allInstances() -> select(s | s.
    name = 'RBAC_Profile') -> first();}
    --End Operations on Model--}

rule Package {
    from s : UML2!"uml::Package" in IN (s.oclIsTypeOf(UML2!"uml::Package")
        )
    to t : UML2!"uml::Package" (
        name <- s.name,
        profileApplication <- s.profileApplication)}

rule Component {
  --Copy Component Bloc--
    from s : UML2!"uml::Component" in IN
    to t : UML2!"uml::Component" (
      -- __xmiID__ <- s.__xmiID__,
        name <- s.name.debug('Component'),
  do {
  --Reapply existant Stereotypes--
    for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
    if(s.owner.hasStereotype('RBAC.ProtectionObject'))t.applyStereotype(
        thisModule.getStereotype(thisModule.entityProfile,'RBAC.
        ProtectionObject'));
  --End Reapply existant Stereotypes--}
  --End Operations on Component--}

rule Connector {
  --Copy Connector Bloc--
    from s : UML2!"uml::Connector" in IN
    to t : UML2!"uml::Connector" (
        --__xmiID__ <- s.__xmiID__,
        name <- s.name.debug('Connector')
    --Operations on Connector--
    do {
      --Reapply existant Stereotypes--
      for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
      --End Reapply existant Stereotypes--

if(((s.end->exists(a | a.role.owner.hasStereotype('RBAC.User')))or(s.end
    ->exists(a | a.role.owner.hasStereotype('Authenticator.User'))))and(s.
    end->exists(a | a.role.owner.hasStereotype('RBAC.ProtectionObject')))
    or (s.owner.hasStereotype('RBAC.ProtectionObject')))
  {t.applyStereotype(thisModule.getStereotype(thisModule.entityProfile,'
      RBAC.Right'));  }}
    --End Operations on Connector--}

rule ConnectorEnd {
  --Copy ConnectorEnd Bloc--
    from s : UML2!"uml::ConnectorEnd" in IN
```

```
   to t : UML2!"uml::ConnectorEnd" (
      --__xmiID__ <- s.__xmiID__,}
rule Port {
  --Copy Port Bloc--
   from s : UML2!"uml::Port" in IN
   to t : UML2!"uml::Port" (
      --__xmiID__ <- s.__xmiID__,
      name <- s.name.debug('Port'),
   --Operations on Port--
   do {
     --Reapply existant Stereotypes--
     for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
     --End Reapply existant Stereotypes--}
   --End Operations on Port--}


---------------------------END RULES------------
```

La troisième transformation

```
create OUT : UML2 from IN : UML2, IN1 : PRO;

--------------------------BEGIN HELPERS------------

--Helper 'getStereotype'
--Return the stereotype from the applied profile
helper def: getStereotype(p: UML2!Profile, name: String): UML2!Stereotype
    =
  p.ownedStereotype -> select(s | s.name = name) -> first();
--End Helper 'getStereotype'

--Helper 'hasStereotype'
--Return whether the element has the stereotype in the helper parameter
helper context UML2!Element def: hasStereotype(stereotype : String) :
    Boolean =
  self.getAppliedStereotypes() -> collect(st | st.name) -> includes(
     stereotype);
--End Helper 'hasStereotype'

---------------------------------END HELPERS------

---------------------------------BEGIN RULES-----------
rule ElementImport {
   from
s : UML2!"uml::ElementImport" in IN
   to
t : UML2!"uml::ElementImport" (
        visibility <- s.visibility,
         alias <- s.alias,
         eAnnotations <- s.eAnnotations,
         ownedComment <- s.ownedComment,
```

```
249           importedElement <- s.importedElement)}
250
251 rule Model {
252   --Copy Model Bloc--
253   from
254     s: UML2!"uml::Model" in IN
255   to
256     t: UML2!"uml::Model" (
257       --__xmiID__ <- s.__xmiID__,
258       name <- s.name.debug('Model'),
259       --Operations on Model--
260     do {
261       --Reapply existant Profile--
262 for (p in s.getAllAppliedProfiles()) {t.applyProfile(p);}
263 thisModule.entityProfile<-UML2!Package.allInstances() -> select(s | s.
      name = 'RBAC_Profile') -> first();
264       --End Reapply existant Profile--}
265     --End Operations on Model--}
266
267 rule Package {
268     from s : UML2!"uml::Package" in IN (s.oclIsTypeOf(UML2!"uml::Package")
        )
269     to t : UML2!"uml::Package" (
270        name <- s.name,
271        visibility <- s.visibility,
272        profileApplication <- s.profileApplication)}
273 rule Component {
274   --Copy Component Bloc--
275     from s : UML2!"uml::Component" in IN
276     to t : UML2!"uml::Component" (
277       -- __xmiID__ <- s.__xmiID__,
278         name <- s.name.debug('Component'),
279     --Operations on Component--
280   do {
281   --Reapply existant Stereotypes--
282     for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
283   --End Reapply existant Stereotypes--
284   }
285   --End Operations on Component--}
286
287 rule Connector {
288   --Copy Connector Bloc--
289     from s : UML2!"uml::Connector" in IN
290     to t : UML2!"uml::Connector" (
291        --__xmiID__ <- s.__xmiID__,
292        name <- s.name.debug('Connector'),
293     --Operations on Connector--
294     do {
295       --Reapply existant Stereotypes--
296       for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
```

```
297      --End Reapply existant Stereotypes--}
298 rule ConnectorEnd {
299   --Copy ConnectorEnd Bloc--
300     from s : UML2!"uml::ConnectorEnd" in IN
301     to t : UML2!"uml::ConnectorEnd" (
302        --__xmiID__ <- s.__xmiID__,
303        isOrdered <- s.isOrdered.debug('ConnectorEnd'),
304        isUnique <- s.isUnique,
305        role <- s.role,
306        partWithPort <- s.partWithPort)
307   --Copy ConnectorEnd Bloc-- }
308
309 rule Port {
310   --Copy Port Bloc--
311     from s : UML2!"uml::Port" in IN
312     to t : UML2!"uml::Port" (
313        --__xmiID__ <- s.__xmiID__,
314        name <- s.name.debug('Port'),
315     --Operations on Port--
316     do {
317       --Reapply existant Stereotypes--
318       for (st in s.getAppliedStereotypes()) {t.applyStereotype(st);}
319       --End Reapply existant Stereotypes--
320 if(((s.owner.hasStereotype('RBAC.User'))or(s.owner.hasStereotype('
      Authenticator.User')))and (s.end->exists(a |a.owner.hasStereotype('
      RBAC.Right'))))
321 {t.applyStereotype(thisModule.getStereotype(thisModule.entityProfile,'
      RBAC.Role'));}}
322     --End Operations on Port--}
323 ---------------------------END RULES----------
```

9 Annexe B : Template de génération de code fonctionnel de l'application

```
1  [module CodeGen('http://www.eclipse.org/uml2/2.1.0/UML')/]
2
3  [template public generateComponent(c : Component)]
4
5   [comment @main /]
6   [file (c.name.concat('.java'), false,'UTF-8')]
7   package [c.getNearestPackage().name/];
8
9   %import javax.ejb.Remote;
10  %import javax.ejb.Local;
11
12  %@Stateless
13  public class [c.name.toUpperFirst()/] implements [c.name.toUpperFirst().
        concat('Remote')/],[c.name.toUpperFirst().concat('Local')/]{
14
15    public [c.name.toUpperFirst()/](){
16      super();
17    }
18
19    public String getClassName(){
20      return "[c.name.toUpperFirst()/]";
21    }
22
23    [for (p: Component |c.getAppliedStereotypes())]
24    [if p.getLabel().contains('RBAC.ProtectionObject')]
25    public void create(     ){
26      %TODO Auto-generated method stub
27    }
28
29    public void read(     ){
30      %TODO Auto-generated method stub
31    }
32
33    public void update(     ){
34      %TODO Auto-generated method stub
35    }
36
```

```
37    public void delete(     ){
38      %TODO Auto-generated method stub
39    }
40    [/if]
41    [/for]
42
43     }
44  [/file]
45
46  [file (c.name.concat('Local.java'), false,'UTF-8')]
47  package [c.getNearestPackage().name/];
48  %import javax.ejb.Local;
49
50  %@Local
51  public interface [c.name.toUpperFirst().concat('Local')/]{
52    String getClassName();
53
54    [for (p: Component |c.getAppliedStereotypes())]
55    [if p.getLabel().contains('RBAC.ProtectionObject')]
56    void create(     );
57    void read(     );
58    void update(     );
59    void delete(     );
60
61    [/if]
62    [/for]
63    }
64  [/file]
65
66  [file (c.name.concat('Remote.java'), false,'UTF-8')]
67  package [c.getNearestPackage().name/];
68  %import javax.ejb.Remote;
69
70  %@Remote
71  public interface [c.name.toUpperFirst().concat('Remote')/]{
72    String getClassName();
73    [for (p: Component |c.getAppliedStereotypes())]
74    [if p.getLabel().contains('RBAC.ProtectionObject')]
75    void create(     );
76    void read(     );
77    void update(     );
78    void delete(     );
79
80    [/if]
81    [/for]
82  }
83  [/file]
84
85 [/template]
```

10 Annexe C : Template de génération de code aspects

Template de génération d'aspect en aspects J pour l'intégration du patron RBAC

```
[comment encoding = UTF-8 /]
[module generateAspect('http://www.eclipse.org/uml2/2.1.0/UML')/]
[template public generateAspect(c : Class)]
 [file ('ParserXMLFile.java', false, 'UTF-8')]

import java.io.IOException;
import java.util.ArrayList;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
public class ParserXMLFile {
 public ParserXMLFile() {
  super();
   TODO Auto-generated constructor stub
 }

 public boolean verifUser(Object O){
  boolean trouveUser=false;
  try {
   DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();

   File file = new File("C:/SecurityPluginFiles/aspectFiles/
       RBACFileConfiguration.xml");
      if (file.exists()) {
           Document doc = db.parse(file);
           Element docEle = doc.getDocumentElement();
           NodeList userList = docEle.getElementsByTagName("user");
           int j= 0;
           while((j<userList.getLength())&&(trouveUser==false)){
            String S=userList.item(j).getAttributes().getNamedItem("name
                ").getNodeValue();
```

```
37              if (O.equals(S))trouveUser=true;
38              j++;
39             }
40             }
41
42
43    } catch (ParserConfigurationException e) {e.printStackTrace();}
44    catch (SAXException e) {e.printStackTrace();}
45    catch (IOException e) {e.printStackTrace();}
46    return trouveUser;
47  }
48
49  public void collectProtectionObjectsAndRights(Object O,ArrayList<String>
        po,ArrayList<String> r){
50
51     try {
52       DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
53     DocumentBuilder db = dbf.newDocumentBuilder();
54     File file = new File("C:/SecurityPluginFiles/aspectFiles/
           RBACFileConfiguration.xml");
55     Document doc = db.parse(file);
56          Element docEle = doc.getDocumentElement();
57          NodeList poList = docEle.getElementsByTagName("protectionObject"
               );
58          NodeList rightList=docEle.getElementsByTagName("right");
59
60          for(int i=0;i<poList.getLength();i++){
61            String S="";
62          if(poList.item(i).getParentNode().getAttributes().getNamedItem("
               name").getNodeValue().equals(O)){
63
64         po.add(poList.item(i).getAttributes().getNamedItem("name").
              getNodeValue());
65          for(int j=0;j<rightList.getLength();j++){
66
67            if((rightList.item(j).getParentNode().getAttributes().
                 getNamedItem("name").getNodeValue().equals(poList.item(i).
                 getAttributes().getNamedItem("name").getNodeValue()))&&(
                 rightList.item(j).getParentNode().getParentNode().
                 getAttributes().getNamedItem("name").getNodeValue().equals(
                 O))){
68             S=S+rightList.item(j).getAttributes().getNamedItem("name").
                   getNodeValue();
69           }
70          }
71          r.add(S);
72          }}
73
74    } catch (ParserConfigurationException e) {
75     e.printStackTrace();
```

```
76      } catch (SAXException e) {
77        e.printStackTrace();
78      } catch (IOException e) {
79        e.printStackTrace();}
80    }
81
82  }
83    [/file]
84    [comment @main /]
85    [for (p: Component |c.getAppliedStereotypes())]
86      [if p.getLabel().contains('RBAC.User')]
87    [file (c.name.concat('IntroductionAspect.aj'), false,'UTF-8')]
88    public aspect [c.name.concat('IntroductionAspect')/] {
89
90    [comment Introduce attributes and methods for ssecurity /]
91
92      public ArrayList<String> [c.name/].protectionObjects=new ArrayList<
          String>() ;
93    public ArrayList<String> [c.name/].rights=new ArrayList<String>() ;
94
95    public ArrayList<String> [c.name/].getProtectionObjects() {
96      return protectionObjects;
97    }
98
99    public void [c.name/].setProtectionObjects(ArrayList<String>
          protectionObjects) {
100     this.protectionObjects = protectionObjects;
101   }
102
103   public void [c.name/].setRights(ArrayList<String> rights) {
104     this.rights = rights;
105   }
106
107   public ArrayList<String> [c.name/].getRights() {
108     return rights;
109   }}
110
111   [comment pointcut before constructors /]
112   Constructor pointcut
113     ParserXMLFile P= new ParserXMLFile();
114
115   [c.name/]
116   pointcut [c.name.concat('Constructeur')/]([c.name/] a) : execution ([c.
          name/].new(..))&& target(a) ;
117
118   after ([c.name/] a) : [c.name.concat('Constructeur')/](a){
119     System.out.println("Après constructeur "+a.getClassName());
120     if(P.verifUser(a.getClassName())){P.collectProtectionObjectsAndRights(
          a.getClassName(),a.protectionObjects,a.rights);}
121   }[/file][/if] [/for]
```

```
122 [comment pointcut around CRUD methods to verify access rights /]
123 [for (pp: Component |c.getAppliedStereotypes())]
124    [if pp.getLabel().contains('RBAC.ProtectionObject')]
125  [file (c.name.concat('SecurityAspect.aj'), false,'UTF-8')]
126  public aspect [c.name.concat('SecurityAspect')/]{
127  pointcut [c.name.concat('AspectPointcut_create')/]() : call(* [c.name/].
        create(..));
128
129  void around() : [c.name.concat('AspectPointcut_create')/]() {
130 [for ( e: Component | c.getModel().allOwnedElements())]
131 [for ( a: Component |e.getAppliedStereotypes() )]
132
133 [if a.getLabel().contains('RBAC.User')]
134 [e.name/]
135    if(thisJoinPoint.getThis().getClass().getName().equals("[e.name/]")){
136    if((([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf("[c.
        name/]")!=-1){
137     int i=(([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf(
          "[c.name/]");
138     if((([e.name/])thisJoinPoint.getThis()).rights.get(i).indexOf("create
          ") != -1){proceed();}
139     else System.out.println("Access denied!");
140    }  else System.out.println("Access denied!");
141  }
142  [/if][/for][/for]}
143
144    pointcut [c.name.concat('AspectPointcut_update')/]() : call(* [c.name
        /].update(..));
145
146  void around() : [c.name.concat('AspectPointcut_update')/]() {
147 [for ( e: Component | c.getModel().allOwnedElements())]
148 [for ( a: Component |e.getAppliedStereotypes() )]
149
150 [if a.getLabel().contains('RBAC.User')][e.name/]
151  if(thisJoinPoint.getThis().getClass().getName().equals("[e.name/]")){
      if((([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf("[
      c.name/]")!=-1){
152     int i=(([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf(
          "[c.name/]");
153     if((([e.name/])thisJoinPoint.getThis()).rights.get(i).indexOf("update
          ") != -1){proceed();}
154     else System.out.println("Access denied!");
155    }  else System.out.println("Access denied!");
156  }[/if][/for][/for]}
157  pointcut [c.name.concat('AspectPointcut_read')/]() : call(* [c.name/].
        read(..));
158
159  void around() : [c.name.concat('AspectPointcut_read')/]() {
160 [for ( e: Component | c.getModel().allOwnedElements())]
161 [for ( a: Component |e.getAppliedStereotypes() )]
```

```
162
163 [if a.getLabel().contains('RBAC.User')][e.name/]
164 if(thisJoinPoint.getThis().getClass().getName().equals("[e.name/]")){
165    if((([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf("[c.
       name/]")!=-1){
166     int i=(([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf(
        "[c.name/]");
167     if((([e.name/])thisJoinPoint.getThis()).rights.get(i).indexOf("read")
         != -1){proceed();}
168     else System.out.println("Access denied!");
169    }  else System.out.println("Access denied!");}
170  [/if][/for][/for]}
171  pointcut [c.name.concat('AspectPointcut_delete')/]() : call(* [c.name/].
     delete(..));
172
173
174  void around() : [c.name.concat('AspectPointcut_delete')/]() {
175 [for ( e: Component | c.getModel().allOwnedElements())]
176 [for ( a: Component |e.getAppliedStereotypes() )]
177 [if a.getLabel().contains('RBAC.User')][e.name/] if(thisJoinPoint.getThis
    ().getClass().getName().equals("[e.name/]")){
178    if((([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf("[c.
       name/]")!=-1){
179     int i=(([e.name/])thisJoinPoint.getThis()).protectionObjects.indexOf(
        "[c.name/]");
180     if((([e.name/])thisJoinPoint.getThis()).rights.get(i).indexOf("delete
        ") != -1){proceed();}
181     else System.out.println("Access denied!");
182    }  else System.out.println("Access denied!");}
183    [/if][/for][/for]}}[/file][/if][/for]
184 [/template]
```