Modèles formels et outils génériques pour la gestion et la recherche de composants

Oualid Khayati

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

T H E S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Systèmes d'information

préparée au laboratoire Logiciels Systèmes Réseaux (LSR)
dans le cadre de **l'Ecole Doctorale : Mathématiques, Sciences et technologies de l'information, Informatique**

présentée et soutenue publiquement

par

Oualid KHAYATI

Le 17 décembre 2005

Titre :

Modèles formels et outils génériques pour la gestion et la recherche de composants

_____

Directeurs de thèse : Jean-Pierre GIRAUDIN et Agnés FRONT

_____

JURY

Mme Christine COLLET	, Présidente
Mme Colette ROLLAND	, Rapporteur
M. Claude CHRISMENT	, Rapporteur
Mme Catherine BERRUT	, Examinateur
M. Jean-Pierre GIRAUDIN	, Directeur de thèse
Mme Agnès FRONT	, Co-encadrant

À mes parents pour tous les sacrifices qu'ils ont faits

À mon frère et mes soeurs à qui je souhaite de suivre le même chemin

À mon ange d'amour Nadia à qui je souhaite bon courage pour terminer la sienne

Remerciements

Mes plus sincères remerciements vont à M. Jean-Pierre GIRAUDIN, Professeur à l'Université Pierre Mendès France pour m'avoir accueilli au sein de son équipe et avoir dirigé ce travail, ainsi que pour les nombreuses discussions et les nombreux échanges qui ont eu lieu tout au long de cette thèse.

Je suis très reconnaissant envers Mme Agnès FRONT, Maître de conférences à l'Université Pierre Mendès France pour sa grande disponibilité, pour son soutien, et ses encouragements permanents. Tu m'as encadré depuis le DEA et maintenant grâce à toi j'arrive à voir la fin de ce long parcours qui est la thèse.

Je tiens à remercier Mme Christine COLLET, Professeur à l'Institut National Polytechnique de Grenoble d'avoir accepté de présider le jury, ainsi que Mme Colette ROLLAND, Professeur à l'Université Paris1-Panthéon-Sorbonne et M. Claude CHRISMENT, Professeur à l'Université Paul Sabatier (Toulouse III) pour l'évaluation de ce travail. Enfin je remercie Mme Catherine BERRUT, Professeur à l'Université Joseph Fourier d'avoir accepté de faire partie du jury.

Je remercie également M. François BERNIGAUD de la société ACTOLL pour sa disponibilité et les nombreux échanges qui ont eu lieu dans le cadre du projet Initiative Centr'ACTOLL.

Je tiens à remercier l'ensemble du personnel technique et administratif du laboratoire LSR-IMAG pour leur aide tout au long de ce travail et tout spécialement Liliane Di Giacomo, Pascale Poulet, Martine Pernice, Christiane Plumeré, Francois Challier et Gilles Thieblemont.

Je souhaite aussi remercier toute l'équipe SIGMA (Nicolas, Emmanuel, David, Laurent, Brahim, Ahmed, etc.) pour sa bonne humeur et sa cohésion ainsi que toute personne qui a contribué de près ou de loin à la réalisation de ce travail.

Enfin, j'associe mes remerciements à ma famille grenobloise grâce à laquelle j'ai pu supporter la vie loin des miens. Merci Ibtissem, Ouafa, Leyla, Mhamed, Ramzi, Mehdi, Mohamed, etc.

Table des Matières

Liste des figures

Liste des tableaux

I. Introduction générale

1. Contexte et problématique

Le travail présenté dans cette thèse s'inscrit dans le domaine de la réutilisation de composants et plus précisément dans la recherche et la classification des composants réutilisables. Faciliter et imposer la réutilisation dans l'ensemble du processus de développement des systèmes d'information (SI) nécessite un cadre technique, méthodologique et organisationnel. Pour ce faire, deux types de processus complémentaires doivent cohabiter comme le montre la figure 1 :

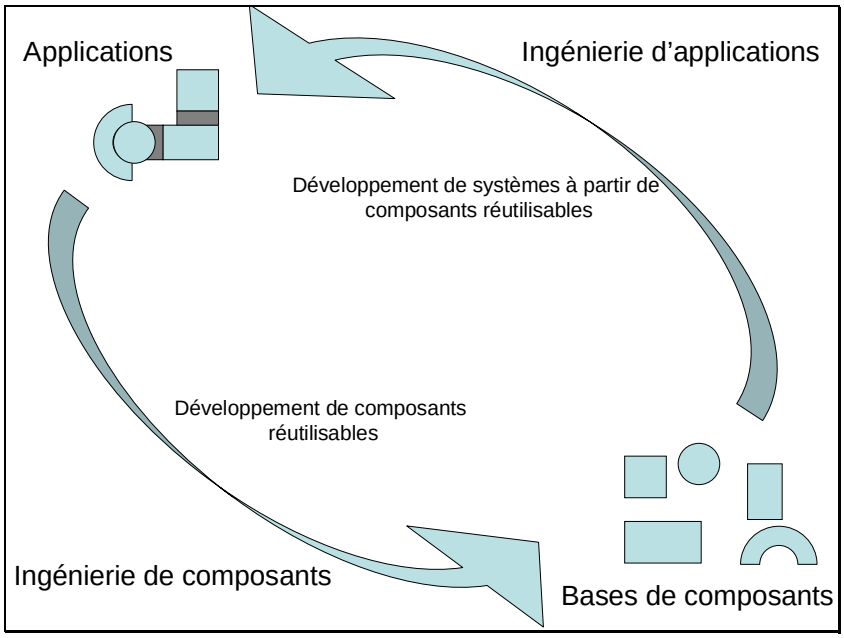


Figure 1. *Deux aspects de la réutilisation.*

- Développement de composants réutilisables (« Design for reuse ») : une ingénierie de composants est nécessaire afin de créer, d'enrichir puis de maintenir un référentiel de composants réutilisables. Le développement d'un système de réutilisation doit mettre en œuvre les fonctionnalités d'identification, de spécification, de développement, de validation et d'organisation des composants. Ce développement suit un processus coopératif et itératif demandant la participation et la mise en accord d'experts du domaine.
- Développement d'applications à base de composants (« Design by reuse ») : un tel développement nécessite de nouveaux environnements intègrant de manière systématique la réutilisation de composants. Ces nouveaux environnements doivent intégrer des fonctionnalités de sélection, d'adaptation et d'intégration de composants pour composer progressivement le système final.

La figure 2 montre un cycle de vie des composants réutilisables et les différents métiers à travers un exemple d'organisation d'une équipe de développement de SI adoptant une approche de développement par réutilisation de composants. L'exemple met en évidence le rôle central que jouent les bibliothèques de composants dans une telle organisation. Il est donc capital de bien réussir l'automatisation de la gestion des bibliothèques de composants pour garantir l'aboutissement et l'efficacité du projet de réutilisation.

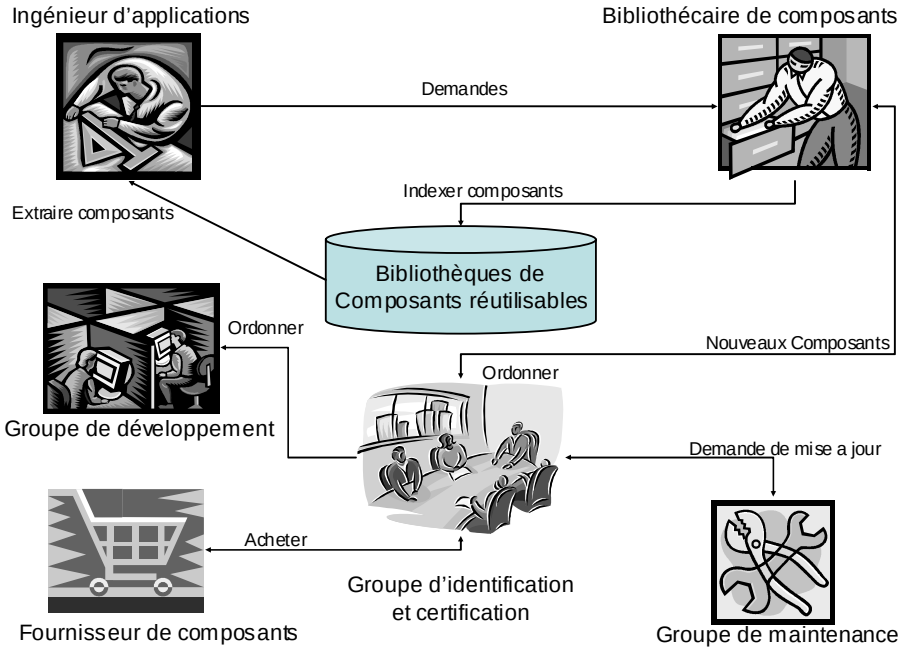


Figure 2. *Cycle de vie des composants réutilisables.*

Face à l'émergence de collections de composants réutilisables de différents types (conceptuels, logiciels, etc.), certains environnements professionnels de développement d'applications ont évolué vers une gestion sommaire de composants. Or, si ces outils permettent effectivement de gérer des bibliothèques de composants, ils n'en facilitent pas pour autant leur sélection et leur utilisation par une recherche ou une navigation adaptée. Les bibliothèques de composants sont des éléments clés dans les environnements de développement à base de composants et de réutilisation de composants (cf. figure 2), mais leur efficacité est optimale lorsque les développeurs et les concepteurs disposent de bibliothèques riches en composants testés et validés, ce qui augmente la fiabilité et diminue le temps de développement des systèmes d'information résultants. Malheureusement, mettre simplement à disposition des développeurs des bibliothèques de composants de taille importante n'augmente pas forcément la productivité. Le concepteur a besoin de techniques performantes de recherche et de classification de composants.

La figure 3 situe le domaine de la recherche de composants par rapport aux domaines de l'ingénierie des SI, de l'ingénierie des composants et de la recherche d'information.

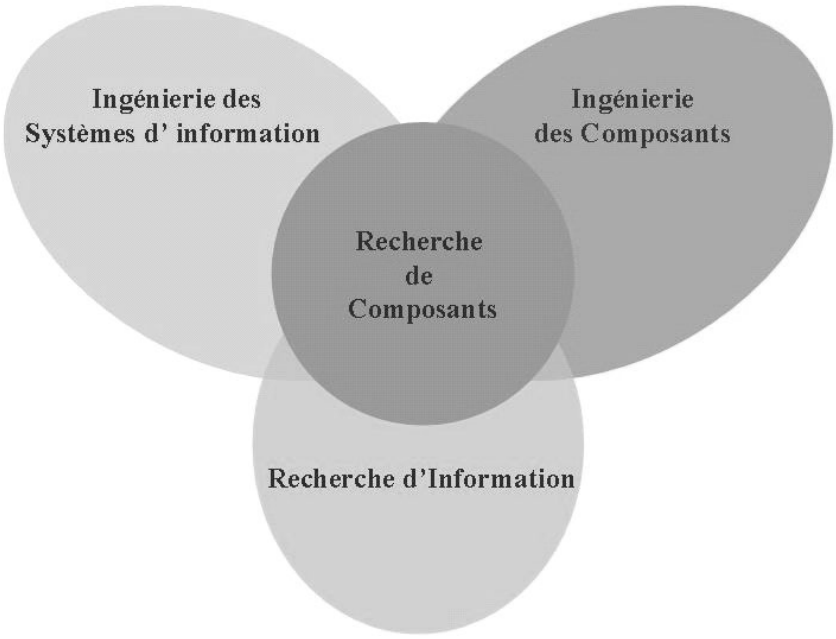


Figure 3. *Différentes facettes de la problématique de la recherche de composants.*

2. Objectifs

Le cycle de développement d'un SI comporte généralement quatre phases majeures : l'étude des besoins, l'analyse, la conception et enfin l'implantation. Chacune de ces phases définit un niveau d'abstraction caractérisé par ses propres acteurs, composants, modèles de composants et techniques de recherche de composants. En effet, un ingénieur intervenant pendant la phase de conception n'a pas besoin de chercher des composants de niveau logiciel. De plus, les techniques de recherche d'information efficaces pour la recherche de composants de niveau conceptuel ne sont pas forcément exploitables ou efficaces pour la recherche de composants de niveau logiciel.

Une bibliothèque de composants dont le but est d'aider à la réutilisation durant tout le cycle de développement d'un SI doit donc être capable de (cf. figure 4) :

- gérer une collection de composants de niveaux d'abstraction et de modèles de composants hétérogènes,
- proposer des techniques de recherche de composants adaptées pour répondre aux besoins de tous les membres de l'équipe à tous les niveaux de développement d'une application.

Dans ce cadre, le but de cette thèse est de proposer un environnement d'aide à la réalisation et à l'utilisation de composants qui supporte le processus de réutilisation en répondant au mieux aux besoins des différents membres d'équipes de développement de SI et de composants. La figure 4 résume les quatre challenges que doit relever un tel environnement :

- une hétérogénéité des équipes de développement, ce qui engendre une hétérogénéité des besoins utilisateurs,
- une hétérogénéité des niveaux d'abstraction de composants. Chaque phase de développement réutilise des composants d'un niveau d'abstraction spécifique,
- une hétérogénéité des modèles et des sources de composants. En effet, il existe plusieurs fournisseurs de composants (autres équipes de développement, fournisseurs

Internet, etc.) et chacun de ces fournisseurs peut utiliser une technologie ou un conditionnement spécifique pour la mise en oeuvre de ses composants,

- une hétérogénéité des techniques de recherche de composants. En effet, puisque les modèles de composants sont variés, une seule technique de recherche de composants ne peut pas s'appliquer à tous les besoins des utilisateurs.

Pour que notre environnement soit capable de relever les quatre challenges précédemment présentés, nous souhaitons atteindre les objectifs suivants :

- pouvoir gérer plusieurs bases de composants,
- supporter et intégrer plusieurs techniques de recherche de composants,
- permettre la représentation d'un composant indépendamment des techniques de recherche de composants qui permettent de le retrouver.

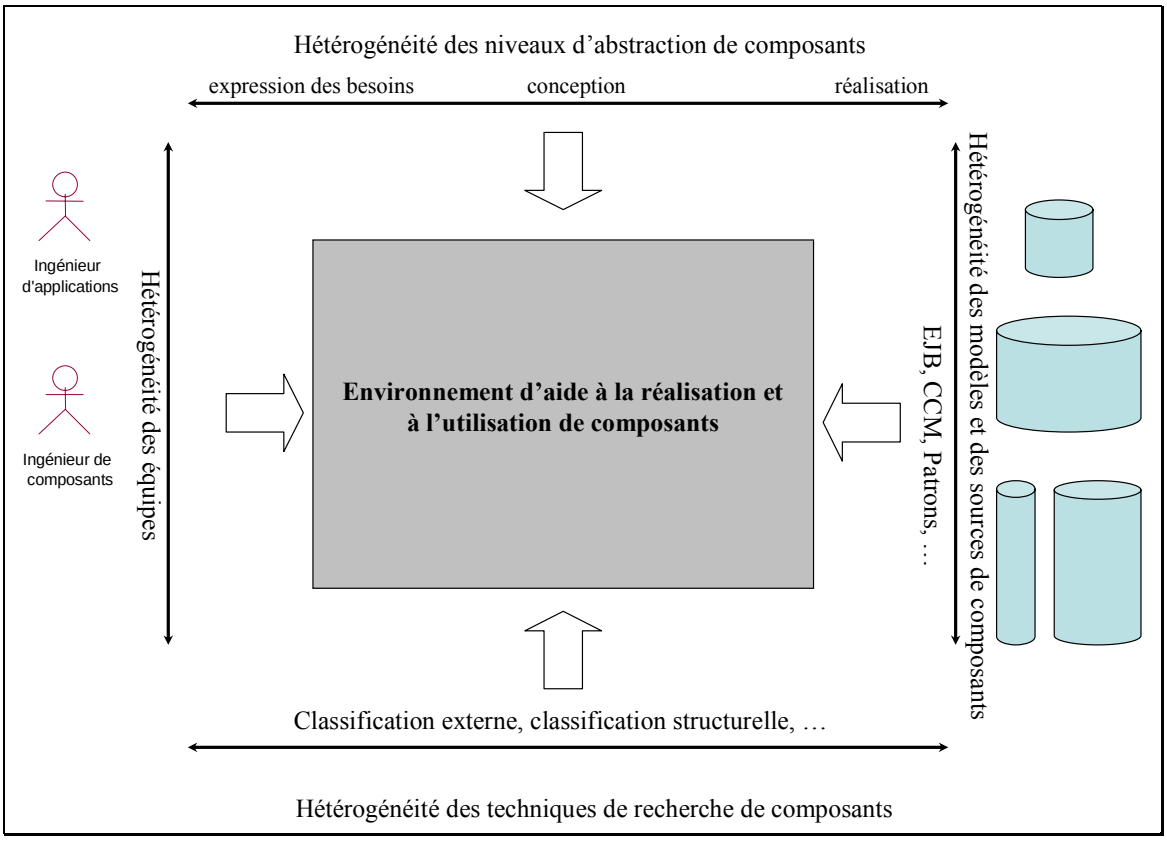


Figure 4. *Challenges de cette thèse*

3. Contributions de la thèse

Nous considérons dans cette thèse qu'une bibliothèque de composants est un cas particulier de système d'information. Elle est au centre du processus de réutilisation et doit être capable de gérer toutes les informations qui se rapportent aux composants et aux processus de réutilisation et de capitalisation des connaissances et du savoir-faire de l'équipe.

Nous adoptons une approche descriptive car nous pensons qu'il est aussi important de gérer une représentation des composants plutôt que les composants eux mêmes. L'approche descriptive garantit à notre bibliothèque de composants la possibilité de s'intégrer dans un environnement de développement déjà existant avec le minimum de modifications dans la façon de travailler de l'équipe de développement.

La première proposition se présente sous la forme du métamodèle *M-Sigma* pour la construction d'un métaoutil pour l'instanciation et la gestion de bases descriptives de composants. Grâce au métamodèle *M-Sigma*, une base descriptive de composants dispose d'une représentation des composants indépendante des techniques de recherche de composants qui permettent de les retrouver. Le métaoutil proposé est évolutif et facilement interfaçable avec d'autres outils de l'environnement de développement.

La deuxième proposition concerne le modèle de bases de composants *C-Sigma*. Le modèle *C-Sigma* tient compte de la richesse sémantique des différents modèles de composants. Ainsi, il permet de rechercher les composants selon des caractéristiques spécifiques à leurs modèles de composants. Il est évolutif et permet l'ajout de nouveaux modèles de composants et sources de composants. L'organisation d'une base descriptive de composants *B-Sigma* instance du modèle *C-Sigma* est évolutive.

La troisième proposition est un guide à l'*ingénieur de modèles de bases descriptives de composants* sous la forme de fragments de démarche permettent de faire évoluer le modèle *C-Sigma* selon ses besoins.

La quatrième proposition concerne le modèle d'un système de recherche de composants exploitant les instances de bases descriptives de composants pour sélectionner et retrouver les composants réutilisables selon les besoins des utilisateurs. Ce modèle permet de définir des requêtes composites exploitant plusieurs techniques de recherche de composants et d'intégrer leurs résultats. L'outil implantant ce modèle est évolutif car il permet d'intégrer de nouvelles techniques de recherche de composants d'une manière transparente sans changement de l'outil existant.

Nous avons développé un environnement d'aide à la réalisation et à l'utilisation de composants (cf. figure 5). Cet environnement est composé d'un SGBDC (système de gestion de bases descriptives de composants) capable de créer et d'instancier des modèles de bases descriptives de composants, d'un système de recherche de composants capable d'exécuter des requêtes utilisateurs sur une base descriptive de composants (BDC), d'un système d'interfaçage avec des outils externes et d'une interface d'administration permettant à un administrateur de gérer les utilisateurs du système et de gérer les serveurs (SGBDR).

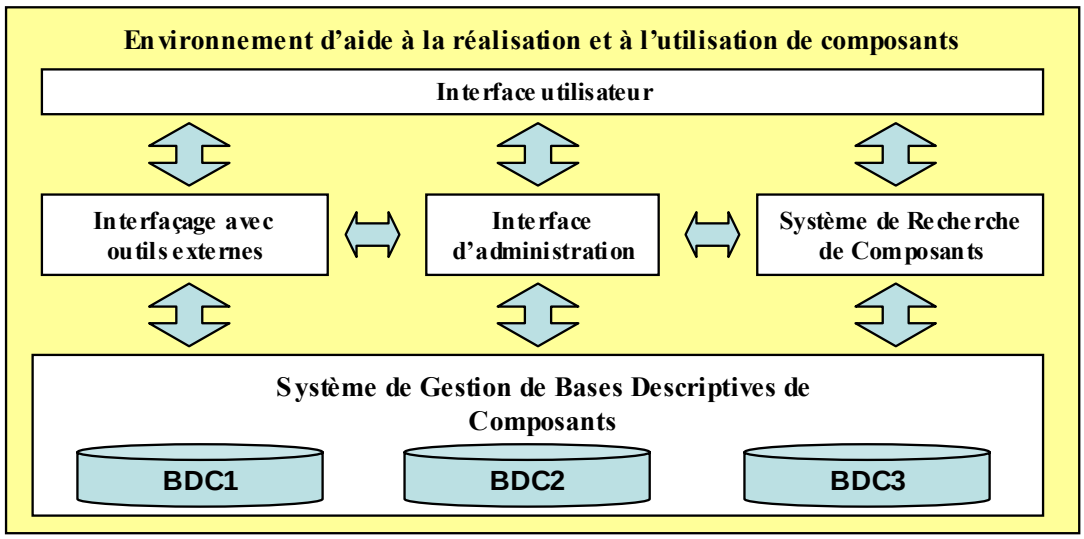


Figure 5. *Architecture de l'environnement d'aide à la réalisation et à l'utilisation de composants.*

4. Plan du mémoire

Ce mémoire est organisé en 9 chapitres (cf. figure 6).

Le chapitre II présente un état de l'art sur les approches à composants. Cette étude permet l'identification des différents types de composants susceptibles d'être utilisés lors du processus de développement de systèmes d'information.

Le chapitre III présente un état de l'art sur les techniques de recherche de composants (TRC). Après la présentation d'une classification des différentes techniques, nous en présentons des exemples pour chacune des catégories identifiées. Enfin, nous définissons des critères d'évaluation des différentes techniques qui nous permettent de les comparer

Le chapitre IV présente la base descriptive de composants *B-Sigma*. Nous considérons dans ce chapitre la base *B-Sigma* comme un système d'information spécialisé dans la gestion de descriptions de composants. L'étude des besoins des utilisateurs en ce qui concerne la base *B-Sigma* nous permet de proposer un modèle de conception *C-Sigma* pour la réalisation d'une telle base.

Le chapitre V présente le métamodèle de systèmes de gestion de bases descriptives de composants (SGBDC) *M-Sigma*. Ce métamodèle généralise les concepts présentés dans le modèle *C-Sigma* et donne plus de liberté aux administrateurs des bases descriptives de composants en leur permettant de faire évoluer les modèles de leurs bases sans pour autant modifier le code du SGBDC.

Le chapitre VI traite des mécanismes d'évolution qui sont offerts aux administrateurs d'un SGBDC. La première partie de ce chapitre présente des fragments de démarches pour guider l'évolution du modèle *C-Sigma*. La deuxième partie présente une architecture de systèmes d'information à base de composants connectables qui nous permet d'implanter un SGBDC acceptant des extensions sous la forme de « plugins ».

Le chapitre VII est divisé en deux parties. Nous commençons ce chapitre par une présentation d'un modèle de système de recherche de composants. La deuxième partie est consacrée à une technique de recherche de composants basée sur les diagrammes de classes UML.

Le chapitre VIII décrit l'expérimentation que nous avons menée en implantant un environnement d'aide à la réalisation et l'utilisation de composants. Cet environnement implante le métamodèle *M-Sigma* et permet d'instancier le modèle *C-Sigma* qui est à son tour instancié pour obtenir la base *B-Sigma*. Notre environnement est muni de la technique de recherche de composants proposée dans le chapitre VII. Un interfaçage avec l'atelier AGAP nous a permis de tester notre technique de recherche de composants sur une collection de patrons et sur une collection de composants que nous avons identifiées lors de la coopération avec notre partenaire industriel Actoll dans le cadre du projet Initiative Centr'Actoll.

Dans un dernier chapitre nous résumons les apports de cette thèse et les perspectives envisagées.

Des annexes complètent ce document pour en illustrer certaines parties un peu abstraites.

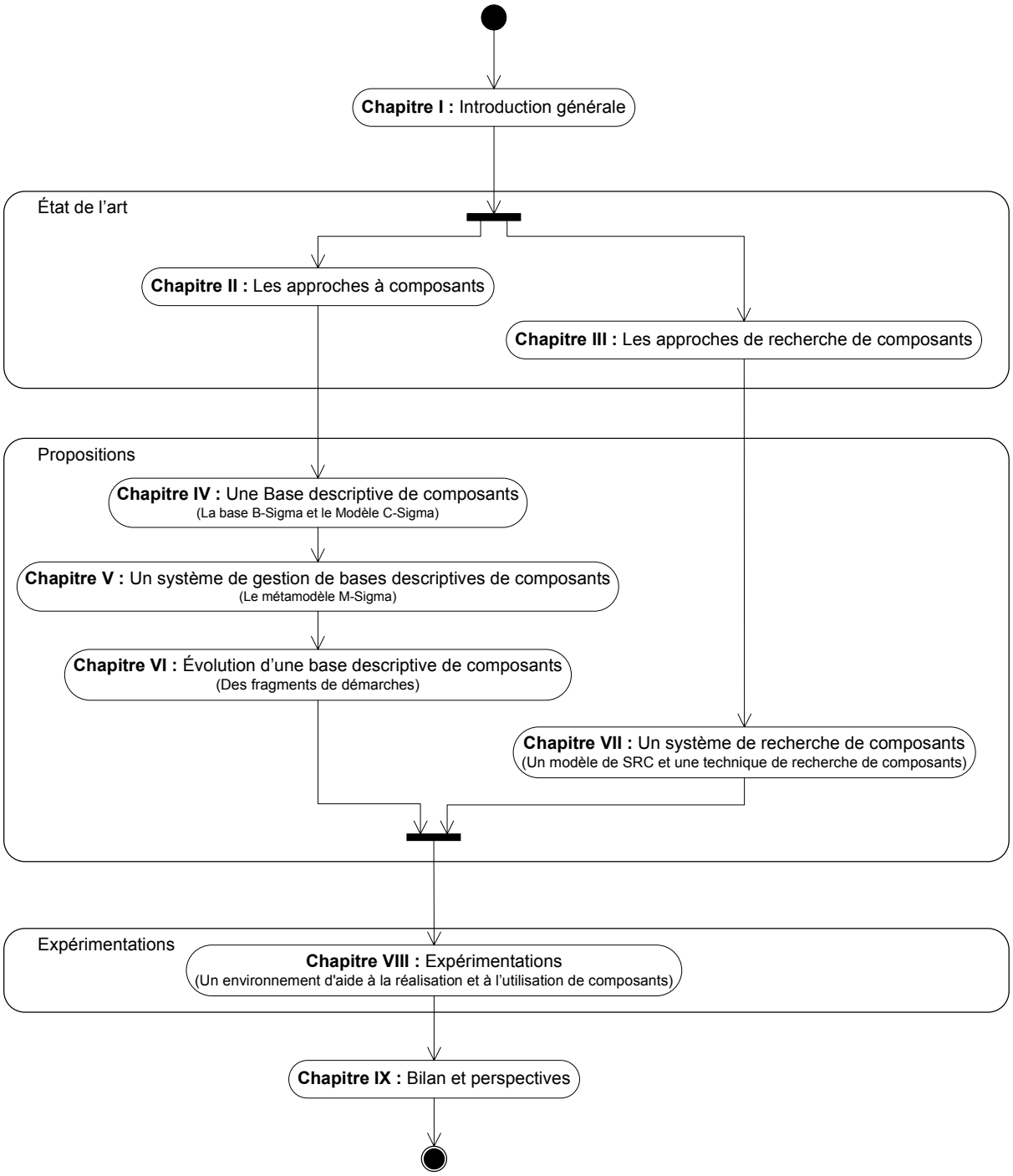


Figure 6. *Plan de la thèse.*

II. Les approches à composants

L'approche à base de composants est depuis une dizaine d'années considérée comme un nouveau paradigme de développement des systèmes d'information (Barbier, 2002). Les activités de programmation ont été les premières concernées par ce nouveau paradigme. Ce chapitre présente un état de l'art de différentes approches à base de composants. Nous présentons dans la section 1 les notions de composant, de modèle de composants et de type de composants. Ensuite, nous proposons un ensemble de caractéristiques permettant de décrire les composants réutilisables. Les sections 2, 3 et 4 présentent respectivement des exemples de modèles de composants conceptuel, logiciel et métier.

1. Notions générales sur les composants

1.1 Composants

A l'heure actuelle, il n'existe pas de normalisation de la notion de composant. Généralement un composant réutilisable est défini comme « *Une unité de conception (de n'importe quel niveau d'abstraction) identifiée par un nom, avec une structure définie et des directives de conception sous la forme de documentation pour supporter sa réutilisation* ». La documentation d'un composant illustre le contexte dans lequel il peut être utilisé en spécifiant les contraintes et les autres composants dont il a besoin pour offrir sa solution (Pernici, 2000). Nous adoptons cette définition générale du concept de composant car elle admet la possibilité d'utiliser le concept de composant dans des niveaux d'abstraction autres que le niveau d'implantation. En effet, le concept de composant a été largement utilisé dans les langages de programmation pour palier aux trois principales limites rencontrées dans la technologie orientée objet (Meyer, 1997) (Finch, 1998) (Villalobos, 2003) :

- un mécanisme d'assemblage assez limité : il s'agit en effet d'une simple relation entre deux classes : une classe hérite ou invoque les méthodes d'une autre classe. Cette construction, réalisée à l'étape d'implantation, est exprimée à travers le langage de programmation. Désormais, les classes sont étroitement liées, ce qui empêche de séparer le processus de développement en deux étapes : la construction (codage) des éléments composables et l'assemblage de ces éléments. Ceci est évidemment reflété dans les phases en amont de la phase d'implantation comme la phase de conception, d'où la discontinuité du processus de réutilisation entre la phase de conception et la phase d'implantation ;
- une réutilisation à grande échelle limitée qui se heurte aux difficultés d'une réutilisation de type « boîte blanche » fondée sur l'héritage où les implantations des entités réutilisées (i.e. des classes) sont visibles et modifiables ;
- une absence des caractéristiques non fonctionnelles dans le modèle à objet original. Pour surmonter cette limitation, certains modèles à objet ont été étendus afin de

supporter quelques caractéristiques non fonctionnelles. RMI (*Remote Method Invocation*) (Grosso, 2001) (Pitt, 2001) et CORBA (*Common Object Request Broker Architecture*) (Orfali, 1998) sont des illustrations d'extensions supportant la caractéristique non fonctionnelle de distribution. L'inconvénient de l'utilisation de telles extensions concerne l'obligation d'ajout du code visant à gérer les propriétés non fonctionnelles dans les méthodes de classes, ce qui engendre des coûts supplémentaires de codage et diminue l'adaptabilité du système d'information en rendant difficile le changement de la technologie de distribution (par exemple, passage de RMI vers CORBA).

1.2 Trois types de composants

Trois types de composants existent : les composants conceptuels, les composants logiciels et les composants métier. Pour chacun de ces types, la définition générale donnée dans la section précédente est complétée avec des définitions spécifiques à chaque type de composants.

1.2.1 Composants conceptuels

Un composant conceptuel est une solution à un problème conceptuel sous la forme d'un modèle (ou une partie d'un modèle) destinée à être réutilisée. Suivant l'approche de modélisation utilisée, la solution peut être spécifiée avec le langage UML (Unified Modeling Language) ou tout autre langage de modélisation. Les composants conceptuels peuvent être de deux types :

- les composants produit : un composant produit est une partie cohérente d'un modèle qui peut être réutilisée avec d'autres composants produit pour assembler un modèle complet. Un produit correspond au but à atteindre lorsqu'on utilise la solution offerte par le composant produit. Par exemple, le patron « composite » de Gamma (Gamma, 1995) peut être considéré comme un composant conceptuel produit.
- les composants processus : un composant processus est une partie cohérente d'un processus qui peut être réutilisée avec d'autres composants processus pour assembler un processus complet. Un processus correspond au chemin à parcourir pour atteindre la solution offerte par le composant processus. Par exemple, le patron « revue technique » d'Ambler (Ambler, 1998) est un composant conceptuel processus.

1.2.2 Composants logiciels

Un composant logiciel est un paquetage cohérent d'implantations logicielles qui peut être indépendamment développé et délivré. Il possède des interfaces explicites spécifiant les services offerts et les services requis par le composant. Il peut être composé avec d'autres composants et être éventuellement paramétrable sans pour autant modifier son implantation (D'Souza, 1999).

D'autres définitions s'intéressent à des aspects spécifiques des composants logiciels :

> *«Un composant est une brique logicielle préfabriquée conçue pour être composée et assemblée avec d'autres composants» (Meijler, 1997).*

La définition de Meijler insiste sur la séparation du processus de production et de réutilisation du composant. Cette propriété de séparation est fondamentale dans les approches de développement à base de composants. Elle permet l'industrialisation du processus de production des systèmes d'information divisé en deux sous-processus : le processus de développement pour la réutilisation dans lequel les composants sont développés avec une optique de réutilisation et le processus de développement par réutilisation de composants dans lequel les systèmes d'information sont construits en utilisant des assemblages de composants.

«A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces» (OMG, 2003).

La définition de l'OMG introduit la propriété d'autonomie qui permet de garantir un couplage faible entre les composants et ainsi permettre le remplacement d'un composant sans toucher à d'autres parties du système d'information.

«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» (Heineman, 2001).

Enfin, la définition de Heineman introduit la standardisation des composants conformément à des modèles de composants. Ainsi les modèles de composants permettent l'outillage des approches de développement à composants en fournissant des environnements de développement.

1.2.3 Composants métier

A l'heure actuelle, il n'existe pas de normalisation de la notion de composant métier. Le concept de composant métier résulte de celui d'objet métier ou « business object » en anglo-américain (Szyperski, 1998). Ainsi, l'OMG (Object Management Group) définit la notion d'objet métier comme suit :

« Business Objects are representations of the nature and behaviour of real world things or concepts in terms that are meaningful to the enterprise. Customers, products, orders, employees, trades, financial instruments, shipping containers and vehicles are all examples of real-world concepts or things that could be represented by Business Objects » (OMG, 1994).

Certaines définitions présentent les composants métier à un niveau logiciel. Par exemple, heineman et al adoptent la définition suivante dans :

« A business component is a software component that provides functions in a business domain (…), for example, an order management application is part of the Enterprise Resource Planning (ERP) business domain » (Heineman, 2001).

Une autre définition plus technique a été donnée par J. Sutherland :

« Technically, business objects encapsulate traditional lower-level objects that implement a business process (they are a collection of lower-level objects that behave as single, reusable units). User interfaces can be thought of as views of large-grained Business Objects. Databases maintain a record of the "state" of Business Objects as they change over time » (Sutherland, 1997).

D'autres spécialistes définissent un composant métier comme une unité de réutilisation de connaissances de domaines, d'un point de vue conceptuel uniquement, par exemple, Casanave dans la définition suivante :

« Un composant métier est vu comme une représentation de la nature et du comportement d'entités du monde réel dans des termes issus du vocabulaire d'une entreprise ». (Casanave, 1996).

1.3 Modèle de composants

Un modèle de composants consiste en un ensemble de conventions à respecter dans la construction et l'utilisation des composants. L'objectif de ces conventions est de permettre de définir et de gérer d'une manière uniforme les composants. Elles couvrent toutes les phases du cycle de vie d'un système d'information à base de composants : la conception, l'implantation, l'assemblage, le déploiement et l'exécution. Concrètement, un modèle de composants décrit certains aspects des composants comme la définition de composants à partir d'objets (classes, modules, etc.), les relations entre les composants, les propriétés

fonctionnelles et non fonctionnelles de chaque composant, les techniques d'assemblage des composants, le déploiement et l'exécution d'un système d'information à base de composants, etc.

Dans la pratique, un modèle de composants donné est spécifique à une phase du cycle de vie du composant et du système d'information. On trouve ainsi des modèles de composants pour la phase de conception (patrons de conception), et d'autres pour les phases d'implantation et de déploiement (EJB, CCM, etc.).

1.4 Les composants dans UML 2.0

La nouvelle spécification UML 2.0 (OMG, 2005a) adoptée récemment par l'OMG a fait évoluer un ensemble de constructions permettant de modéliser des systèmes complexes à base de composants. La spécification UML 2.0 met mieux en évidence le concept de composant qu'elle définit comme une unité autonome à l'intérieur d'un système ou d'un sous-système qui fournit ou requiert un ensemble d'interfaces potentiellement exposées via des ports, cache son état et sa structure interne et ne permet leur manipulation qu'à travers les services offerts par ses interfaces. Un composant doit spécifier ses besoins vis-à-vis de son contexte d'utilisation en termes d'interfaces requises. Ces dépendances d'un composant doivent être conçues de sorte à minimiser le couplage entre composants. Ces conventions facilitent par la suite la réutilisation de composants.

Un composant UML 2.0 possède deux vues, une vue interne et une vue externe.

- **Vue externe**

La vue externe appelée également « vue boîte noire » montre les opérations et les propriétés (attributs, ports, etc.) publiques d'un composant. Optionnellement, une machine d'état peut être attachée à une interface, un port ou au composant lui-même pour décrire un protocole de comportement ou de changement d'états. D'autres comportements peuvent être décrits à l'aide de diagrammes d'activités ou de diagrammes de cas d'utilisation. La spécification UML2.0 définit plusieurs notations possibles d'un composant. La figure 8 montre une notation qui met l'accent uniquement sur les interfaces offertes et requises par un composant. La notation présentée dans la figure 7 détaille la signature des interfaces.

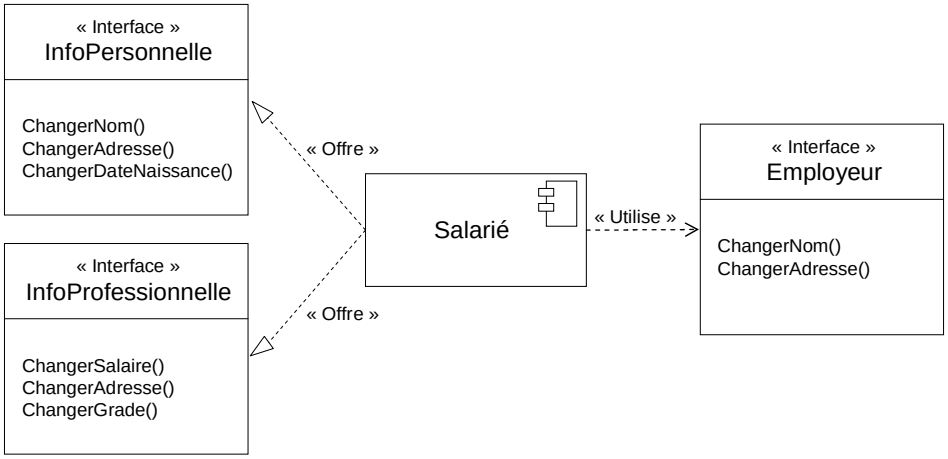


Figure 7. *Vue externe du composant Salarié (avec interfaces détaillées).*

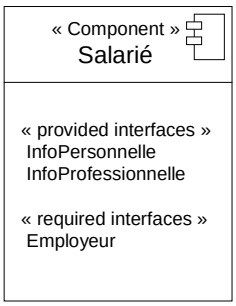


Figure 8. *Vue externe du composant Salarié (listes des interfaces).*

- **Vue interne**

La vue interne appelée également « vue boîte blanche » montre la structure interne du composant et ses propriétés privées. Le lien entre la vue interne et la vue externe est assuré par des liens de dépendance et des connecteurs de délégation à partir des interfaces vers la structure composite interne. Ainsi, la vue interne permet d'identifier les parties de la structure interne d'un composant responsable de la réalisation d'une interface particulière de la vue externe. La figure 9 montre la structure interne du composant *Salarié* en explicitant les classes qui implantent les services offerts par les interfaces offertes (*InfoPersonnelle* et *InfoProfessionnelle*) et qui requièrent les services fournis par l'interface requise (*Employeur*).

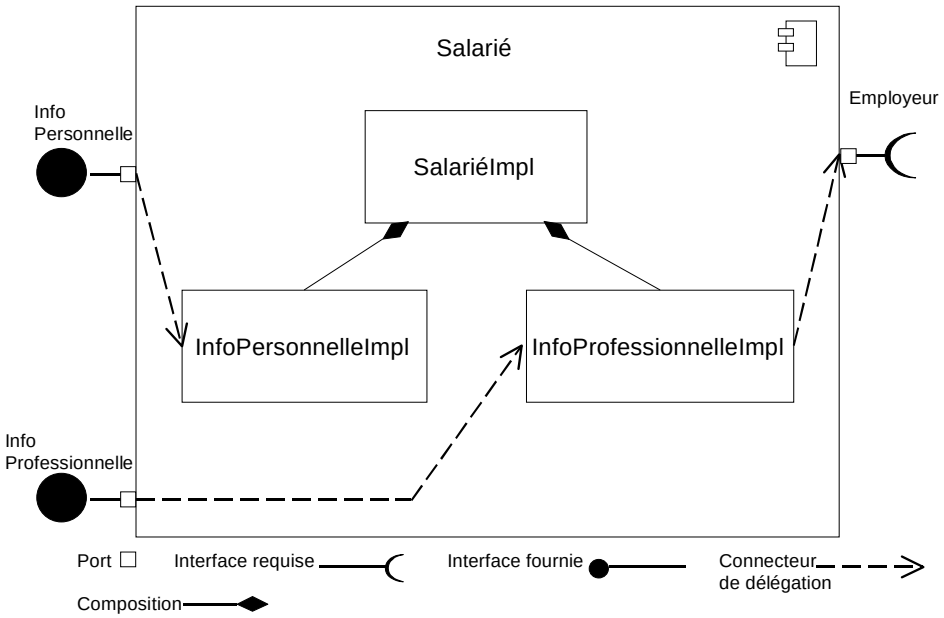


Figure 9. *Vue interne d'un composant simple.*

Un composant UML2.0 peut être un composant simple comme l'exemple de la figure 9 ou un composant composite construit en assemblant d'autres composants. La figure 10 montre l'exemple du composant *Facture* qui est composé par les composants *Bon de commande*, *Client* et *Produit*.

Un port représente un point d'interaction entre un composant et son environnement (cf. figure 9) ou entre un composant et les parties internes (classes ou sous-composants) qui le composent (cf. figure 10). Les interfaces associées à un port spécifient la nature de l'interaction supportée par ce port. Les interfaces requises (respectivement fournies) dans un port caractérisent les services requis (respectivement fournis) par le composant relativement à son environnement à travers ce port. Lors de la spécialisation d'un composant, les ports qu'il possède peuvent subir des redéfinitions : ajout de nouvelles interfaces ou remplacement d'une interface par l'une de ses spécialisations (sous-type).

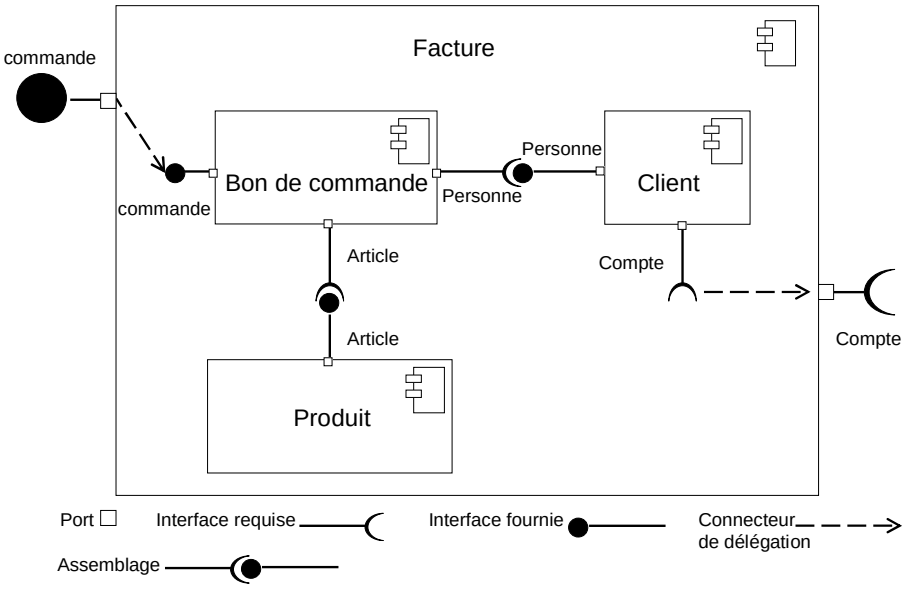


Figure 10. *Vue interne d'un composant composite.*

Il existe deux types de connecteurs : les connecteurs de délégation et les connecteurs d'assemblage. Un connecteur de délégation indique que l'interface offerte (respectivement requise) par le port n'est pas réalisée (respectivement requise) par le composant lui-même, mais par un autre composant (ou classe) qui est compatible avec cette interface. Un port peut être délégué vers un ensemble de ports des composants qui le composent. Un connecteur d'assemblage relie deux composants, un composant client et un composant fournisseur de services si et seulement si le composant fournisseur de services offre une interface compatible (la même ou une spécialisation) avec le composant client. Par exemple, dans la figure 10, le composant *Bon de commande* a besoin d'un composant qui offre l'interface *Article*.

1.5 Caractéristiques des composants réutilisables

Nous présentons dans cette section un ensemble de critères permettant de caractériser les composants réutilisables. Il est possible de caractériser les composants selon un certain nombre de critères, à savoir le type de connaissance, la couverture, la portée, la nature de la solution, la technique de réutilisation, l'ouverture, la granularité (Conte, 2001) et l'architecture.

- **Type de connaissance**

Le type de connaissance précise la nature (produit ou processus) d'un composant conceptuel. Une connaissance de type *produit* correspond à un but à atteindre tandis qu'une connaissance de type *processus* correspond au chemin à suivre pour atteindre un résultat.

- **Couverture**

Le degré de couverture d'un composant est défini par rapport à un domaine d'application. Un composant réutilisable peut être un composant *général* (resp. *domaine*, *entreprise*) si le problème traité est fréquent dans de nombreux domaines d'applications (resp. dans un domaine d'applications, dans une entreprise particulière).

- **Portée**

La portée d'un composant est évaluée en fonction de l'étape d'ingénierie (*analyse*, *conception*, *implantation*) à laquelle le composant s'adresse.

- **Technique de réutilisation**

Un composant peut être réutilisé par adaptation (imitation), par spécialisation, par composition etc.

- **Ouverture**

L'ouverture caractérise le niveau de transparence du composant :

- Boîte noire : seule l'interface du composant est visible, la structure interne n'est ni visible ni modifiable.
- Boîte blanche : le composant est modifiable.
- Boîte en verre : la structure interne est visible mais non modifiable.

- **Granularité**

La granularité des composants orientés objets est souvent mesurée en nombre de classes. De manière plus générale, elle peut être mesurée en nombre d'entités (modules, activités, etc.). Elle peut être faible (< 10), moyenne (< 100) ou forte.

- **Architecture**

L'architecture d'un composant peut être distribuée ou locale.

2. Les composants conceptuels

Nous présentons dans cette section les principaux concepts définis dans les patrons qui sont des exemples de composants conceptuels.

Nous avons choisi ce type de composants conceptuels car il est le plus représentatif et le plus complet tant en matière de solution décrite qu'en termes de problème à résoudre et de contexte d'application.

Des fragments de modèles UML peuvent être considérés comme composants conceptuels s'ils se limitent à des préoccupations de nature spécification ou implantation. Ce sera le cas pour certains composants UML 2.0.

2.1 Patrons

Une définition du terme patron adoptée par les concepteurs de systèmes est la suivante :

> *« Un patron est à la fois une partie du système et une description qui explique comment le construire ».*

Les patrons décrivent généralement des abstractions utilisées par les experts concepteurs ou programmeurs dans leurs réalisations. Ils utilisent les propriétés présentes dans d'autres abstractions comme les procédures et les objets, mais sont également capables de combiner ces propriétés pour atteindre un niveau d'abstraction supérieur (Coplien, 1992). Pour mieux comprendre cette notion de patron, nous étudions d'autres définitions données dans la littérature spécialisée :

> *« A pattern is a description of a general solution to a common problem or issue from which a detailed solution to a specific problem may be determined » (Ambler, 1998).*

> *« Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière » (Alexander, 1979).*

« A pattern is an idea that has been useful in one practical context and will probably be useful for others » (Fowler, 1997).

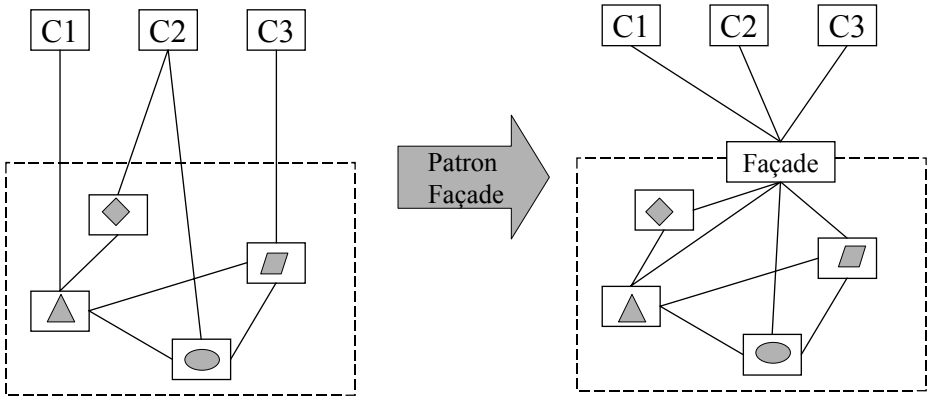
Toutes ces définitions s'accordent sur le fait qu'un patron est un couple formé par un problème et par une solution. Fowler (Fowler, 1997) introduit de plus la notion de contexte. Finalement, nous adoptons la définition suivante.

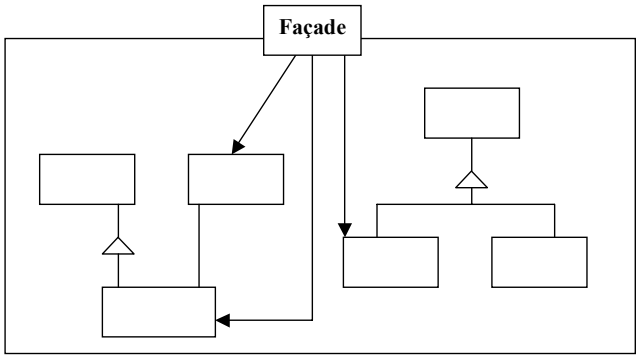
« Un patron est une solution à un problème dans un contexte ».

Chaque mot dans cette phrase a un sens. Le contexte se réfère à un ensemble de situations récurrentes pour lesquelles un patron peut être appliqué. Le problème définit un but à atteindre. La solution se réfère à un modèle ou à un processus que l'on peut adapter pour résoudre le problème.

2.2 Exemple de patron

Cette section présente comme exemple de patron le patron façade de Gamma (Gamma, 1995).

Nom	« Façade ».
Classification	Objet structurel.
Intention	Ce patron fournit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.
Motivation	Structurer un système en sous-systèmes aide à réduire la complexité de l'ensemble. Un des objectifs communs à toutes les conceptions est de réduire la multiplicité des communications et des liens de dépendance entre sous systèmes. Un moyen pour y parvenir, consiste à introduire un objet Façade, qui offre une interface unique et simplifiée aux services de haut niveau d'un sous-système. 
Indication d'utilisation	- On dispose d'un sous-système complexe - Plusieurs relations de dépendance entre les clients et les classes d'implémentation d'une abstraction - Pour structurer un sous-système en niveau.

Structure	
Constituants	- Façade : connaît les classes du sous-système compétentes pour une requête, - Classes du sous-système : implémentent les fonctionnalités du sous-système, gèrent les travaux assignés par l'objet façade, ne connaissent pas la façade.
Collaboration	- Les clients communiquent avec le sous-système en envoyant des requêtes à la façade, qui répercute celles-ci aux objets appropriés du sous-système. Bien que les objets du sous-système effectuent réellement le travail, la façade peut avoir à charge, en propre, le travail de transcription de son interface à celles du sous-système. - Les clients qui utilisent la façade n'ont pas à accéder directement aux objets du sous-système.
Conséquences	- Masquer aux clients les composants du sous-système, donc réduire le nombre d'objets. - Favoriser le couplage faible entre le sous-système et ses clients.
Patrons apparentés	Fabrique Abstraite, Médiateur

2.3 Systèmes de patrons

Un système de patrons (certains auteurs préfèrent le terme de catalogue ou langage de patrons) est un ensemble de solutions consensuelles à un ensemble de problèmes ponctuels et récurrents dans un domaine donné.

Un système de patrons est un graphe orienté contenant généralement peu de cycles, et où les nœuds sont les patrons et les arcs sont les relations entre patrons. Un patron racine décrit une problématique générale traitée par le système et explicite globalement les solutions offertes. Il doit fournir une solution partielle en résolvant certaines forces[1] et en en laissant d'autres non résolues. Ainsi, il décrit des sous-problèmes pour lesquels il faut faire appel aux autres patrons du système (Coad, 1995) (Gamma, 1995) (Ambler, 1998) (Gzara, 2000). Plus un système de patrons est spécialisé, plus il est connexe et moins il contient de points d'entrée.

[1] Les buts à atteindre et les contraintes à respecter.

2.4 Formalismes de patrons

Un formalisme est la forme, la structure, et la syntaxe que tout auteur utilise pour écrire ses patrons. D'après les définitions citées (cf. 2.1), tout patron peut être assimilé à un triplet « problème, contexte, solution » qui relie une solution à un problème dans un contexte donné. De nombreux formalismes de représentation de patrons intègrent ces trois aspects et sont globalement équivalents. Un patron peut se présenter sous une forme narrative ou structurée. La forme structurée améliore la lisibilité d'un formalisme de patrons. Un formalisme de patrons donne une description de la représentation de la solution offerte par le patron et décrit le contexte et le problème, ce qui facilite l'outillage des systèmes de patrons. De nombreux auteurs ont proposé des formalismes : Gamma (Gamma, 1995), Fowler (Fowler, 1997), Ambler (Ambler, 1997), etc. Certains de ces formalismes sont spécifiques à une phase de développement spécifique avec par exemple les patrons d'analyse de Coad (Coad, 1992) ou les patrons de conception de Gamma (Gamma, 1995).

2.5 Relations inter-patrons

Les relations inter-patrons offrent un moyen de structuration des systèmes de patrons. On distingue trois types de relations inter-patrons (Khayati, 2001) : primaires, secondaires et ternaires.

a) Relations primaires

Ce sont les relations de base offertes par la plupart des formalismes. Elles sont au nombre de trois : « utilisation », « raffinement » et « alternative » (Noble, 1998a) (Noble, 1998b) (Zimmer, 1994) (Conte, 2001).

- **Utilisation**

C'est la relation la plus utilisée dans les systèmes de patrons. Elle permet la décomposition d'un problème complexe en sous-problèmes de complexité moindre. On peut donner une définition formelle de cette relation : un patron *X* utilise un patron *Y* si et seulement si certains problèmes posés par *X* peuvent être résolus par *Y*.

Dans le cas du catalogue de Gamma (Gamma, 1995), cette relation « utilisation » est moins formelle et plus générale. Il s'agit pour un patron donné de lister des patrons « voisins » à utiliser conjointement.

- **Raffinement/Extension**

Ce sont deux relations réciproques. La relation « extension » tend à généraliser l'intention d'un patron alors que la relation « raffinement » la restreint. Donc si un patron *X* est une extension du patron *Y*, *X* peut résoudre le problème posé par *Y*. Réciproquement, on dit que *Y* est un raffinement de *X*.

- **Alternative**

Un patron *X* est une alternative d'un patron *Y*, si *Y* a la même intention que *X*, mais propose une solution différente. Cette deuxième solution peut être obtenue en appliquant des forces différentes et le même contexte pour résoudre le même problème.

b) Relations secondaires

Dans la pratique, les auteurs de patrons ont parfois besoin d'utiliser des relations dites secondaires. Cette appellation vient du fait que chacune de ces relations peut être exprimée en fonction des relations primaires.

- **Utilisé par** (Noble, 1998b)

C'est la réciproque de la relation «utilisation».

- **Variant** (Noble, 1998b)

J. Noble définit deux sortes de variants : les « variants solutions » et les « variants problèmes ».

Les « variants solutions » sont des patrons qui offrent différentes solutions à un même problème. Par exemple, le patron « Adaptor » (Gamma, 1995) a deux variants solutions : les patrons « Class adaptor » et « Object adaptor ».

Les « variants problèmes » sont moins nombreux que les variants solutions, ils offrent la même solution à des problèmes différents.

Zimmer (Zimmer, 1994) définit une troisième sorte de variant le « variant utilise ». En effet, il distingue deux types de relations « utilise ». La première est « *X* uses *Y* in its solution » : c'est la relation primaire « utilisation » précédemment décrite. La deuxième est « variant of *X* uses *Y* in its solution » : comme son nom l'indique, dans ce cas de figure, un variant du patron *X* utilise *Y* dans sa solution. Par exemple, une imitation du patron *Visiteur* (Gamma, 1995) utilise le patron Double-Dispatch, mais pas nécessairement le patron *Itérative*. De la même façon, un variant du patron *Composite* peut utiliser selon le besoin le patron *Itérative* ou le patron *Visiteur*.

- **Similarité** (Zimmer, 1994)

Cette relation a été introduite par Zimmer. Elle exprime les commentaires qui peuvent se trouver sous la rubrique « patrons apparentés » du formalisme de Gamma. Elle peut être utilisée par exemple pour exprimer la relation entre deux patrons qui utilisent deux stratégies de résolution de problèmes similaires (équivalence avec variant problème). Une deuxième utilisation consiste à exprimer le lien sémantique entre deux patrons qui résolvent le même type de problème (équivalence avec alternative).

- **Combine**

Les patrons d'architecture logicielle proposés dans (Buschmann, 1996) et les patrons proposés par (Dyson, 1997) introduisent la relation « Combines ». Cette relation peut exister entre deux patrons, si leur combinaison permet de résoudre un problème qui n'a été traité par aucun autre patron du système de patrons auquel ils appartiennent. Un exemple concerne les patrons Proxy et Forwarder-Receiver (Buschmann, 1996). Ces patrons peuvent être combinés pour implanter un service transparent de communication peer-to-peer.

- **Requiert**

Un patron *X* requiert un patron *Y* si le patron *Y* est un pré-requis pour résoudre le problème traité par *X*. Cette relation secondaire peut être exprimée à l'aide de la relation primaire « utilisation ». La seule différence entre ces deux relations réside dans l'ordre d'application des patrons. La relation requiert nécessite qu'il y ait au moins une application du patron *Y* avant d'envisager d'appliquer le patron *X*.

c) Relations ternaires

Les relations ternaires sont des relations de haut niveau qui permettent d'exprimer des relations entre patrons qui modélisent la récursivité, le séquencement, etc.

- **Tiling**

Cette relation exprime le fait que des patrons peuvent être appliqués successivement pour résoudre un problème. Cette relation a été introduite par D. H. Lorenz (Lorenz, 1997). Par exemple, on peut appliquer un Iterator (Gamma, 1995) sur d'autres Iterator. Un autre exemple consiste à coupler des patrons et leurs variants ou raffinements. L'exemple suivant nous montre comment construire un objet complexe en appliquant successivement le patron « Composite » ou l'un de ses patrons dérivés. Pour construire un composite à trois dimensions, il faut appliquer le patron Two-Dimensional Composite deux fois (cf. figure 11).

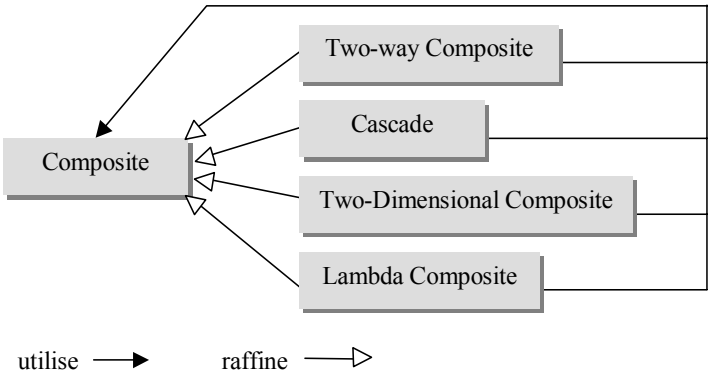


Figure 11. *Exemple de la relation Tiling.*

- **Séquence d'élaboration**

Une séquence d'élaboration est définie comme une séquence de patrons commençant par un patron simple, petit et de bas niveau qui engendre un petit nombre de conséquences négatives (sous problèmes non résolus). La complexité des patrons augmente au fur et à mesure qu'on avance dans la séquence jusqu'à atteindre les patrons d'architecture de grande envergure. On trouve souvent des séquences d'élaboration dans les systèmes de patrons. Les relations entre patrons au sein d'une séquence d'élaboration sont souvent du type « utilisation ». Par exemple la figure 12 schématise une séquence d'élaboration entre les patrons qui permettent de mettre en œuvre un SGBDR (Système de Gestion de Bases de Données Relationnelles). En effet dans un SGBDR, on ne manipule que des tuples et des relations. Les relations sont divisées en deux sous familles : relations système et relations application. Les mécanismes de base de manipulation de ces relations sont les mêmes, ce qui nous amène à encapsuler ce savoir-faire dans le patron « Gestion des relations ». Nous laissons les spécificités de la gestion des relations système et application dans les patrons « Gestion des relations application » et « Gestion des relations système ». L'étude des relations entre ces patrons nous montre que les patrons « Gestion des relations système » et « Gestion des relations application » sont des raffinements du patron « Gestion relation » et en même temps ils l'utilisent. Si on poursuit le même raisonnement, on constate le même type de relation entre « Gestion des relations » et « Gestion des fichiers » et entre « Gestion des fichiers » et « Gestion des E/S »

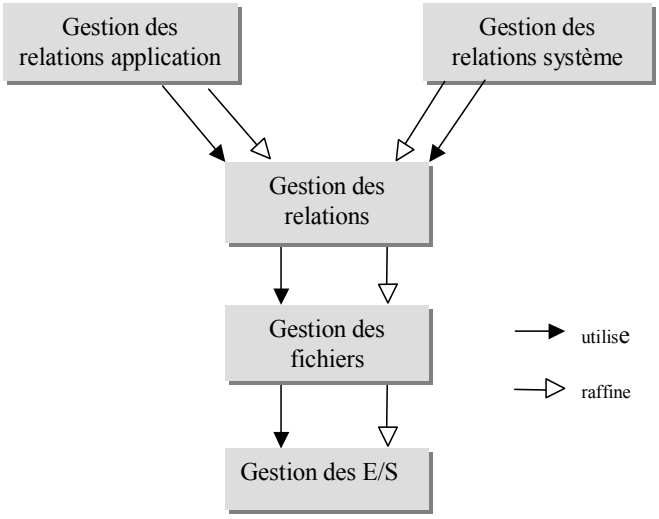


Figure 12. *Patrons de mise en place d'un SGBDR.*

2.6 Métamodèle de représentation de patrons

Le métamodèle présenté dans la figure 13 reprend les principales caractéristiques des patrons. Pour l'élaboration de ce métamodèle, nous avons fait certaines hypothèses :

- un système de patrons peut contenir des patrons décrits dans des formalismes de patrons différents.
- un rubrique peut appartenir simultanément à plusieurs catégories. Rien n'empêche qu'une rubrique réalisation soit aussi une rubrique interface utilisée pour la sélection d'un patron.

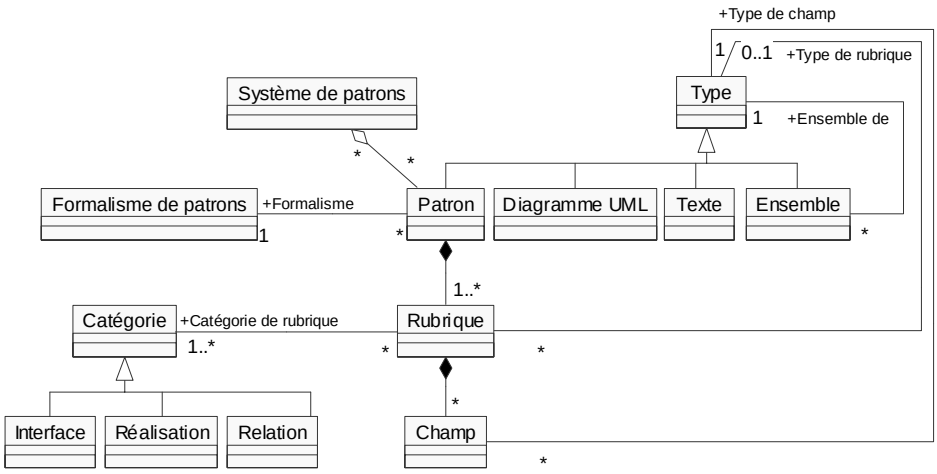


Figure 13. *Métamodèle de systèmes de patrons.*

2.7 Conclusion

Un patron représente une solution générique à un problème dans un contexte donné. Ce type de composant ne se limite pas à donner la solution mais permet d'indiquer dans quelle condition elle sera appliquée. Il définit le quand, le comment et le pourquoi.

Un patron est un moyen de décrire un savoir ou un savoir-faire. Il permet ainsi de capitaliser l'expertise et dans le cas d'un travail en équipe, d'avoir un langage commun entre tous les membres.

Les patrons donnent aux concepteurs un moyen pour structurer leurs idées, leurs démarches et leurs perceptions du monde sur lequel ils travaillent. C'est un moyen qui leur permet d'augmenter le niveau d'abstraction dans lequel ils travaillent pour pouvoir s'intéresser aux propriétés de l'univers de conception qui les intéressent sans se perdre dans les détails et la complexité du problème.

Les patrons permettent aussi au concepteur de modéliser leur démarche de développement en la décrivant sous la forme d'un système de patrons processus. De la même manière, ils permettent de décrire le processus de réutilisation et d'adaptation des composants (logiciels ou conceptuels) dans une démarche de développement de systèmes d'information à base de composants.

La figure 14 présente un exemple de classification de plusieurs systèmes de patrons. La classification se fait selon les caractéristiques des composants conceptuels présentées dans la section 1.5. La figure 14 place les patrons dans un espace à 3 dimensions et non à 7 car leur technique de réutilisation est toujours l'imitation, leur granularité est toujours faible, l'ouverture est toujours du type boîte blanche et la caractéristique d'architecture n'a pas de sens car un patron ne peut être distribué.

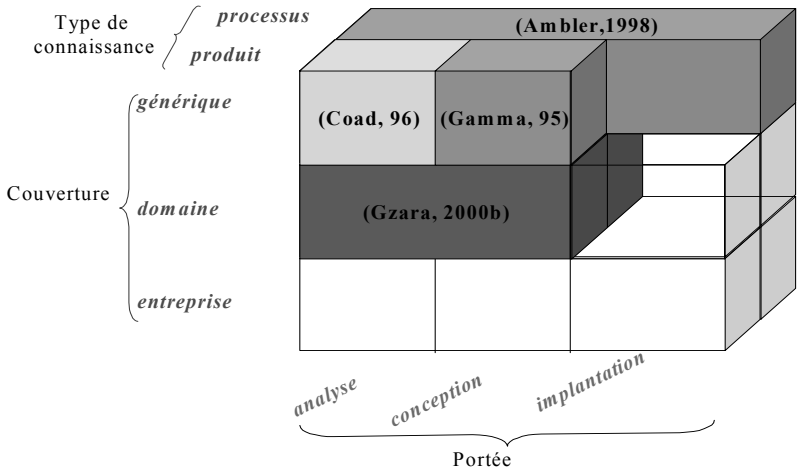


Figure 14. *Organisation des systèmes de patrons (Conte, 2001).*

3. Les composants logiciels

L'approche à base de composants logiciels introduit une démarche visant à faciliter la construction d'applications à partir d'assemblage de briques logicielles préfabriquées. Nous procédons dans ce qui suit à la présentation des principaux concepts utilisés dans les modèles de composants logiciels. Nous utilisons ensuite ces concepts pour la présentation d'un ensemble de modèles de composants logiciels.

3.1 Concepts

- **Classe de composants**

Une classe de composants se situe à un niveau équivalent à celui de la classe dans l'approche objet. Une classe de composants possède une *vue externe*, une *vue interne* et une *vue déploiement*.

La vue externe représente la vision qu'ont les clients et les autres composants des instances du composant. Elle contient les *interfaces fonctionnelles* et les *attributs de configuration*. Une *interface fonctionnelle* est constituée par un ensemble de méthodes relatives aux fonctions fournies ou requises par les instances du composant. Les *interfaces fonctionnelles* requises représentent des dépendances au niveau des instances du composant et doivent être respectées lors de l'instanciation de la classe de composants pour que la nouvelle instance soit capable de fournir des services à ses clients à travers ses interfaces fonctionnelles fournies. Les *attributs de configuration* permettent la réutilisation d'un composant par l'adaptation de son comportement à travers la modification de certains paramètres. Par exemple, la modification de l'attribut *langue* peut modifier la langue utilisée dans un composant qui affiche un formulaire de saisie de données.

La vue interne représente la vue que possède l'environnement d'exécution d'une instance du composant. Elle contient les *interfaces de contrôle* qui sont fournies ou requises. Une interface de contrôle fournie contient des méthodes permettant la gestion du cycle de vie d'une instance de composant pendant l'exécution. Ces méthodes sont destinées à être utilisées par l'environnement d'exécution du modèle de composants et non pas par les clients de l'instance. Une interface de contrôle requise représente par exemple l'interface qu'utilise une instance pour interagir avec l'environnement d'exécution (appelé communément *contexte*).

La vue déploiement contient les informations sur les *dépendances* et les *attributs de déploiement* qui sont propres à une implantation du composant. Les dépendances sont par exemple les ressources matérielles, logicielles (par exemple le n° de version) ou les services non fonctionnels requis par une implantation du composant. Les *attributs de déploiement* peuvent être une adresse d'un serveur utilisé par toutes les instances d'un composant.

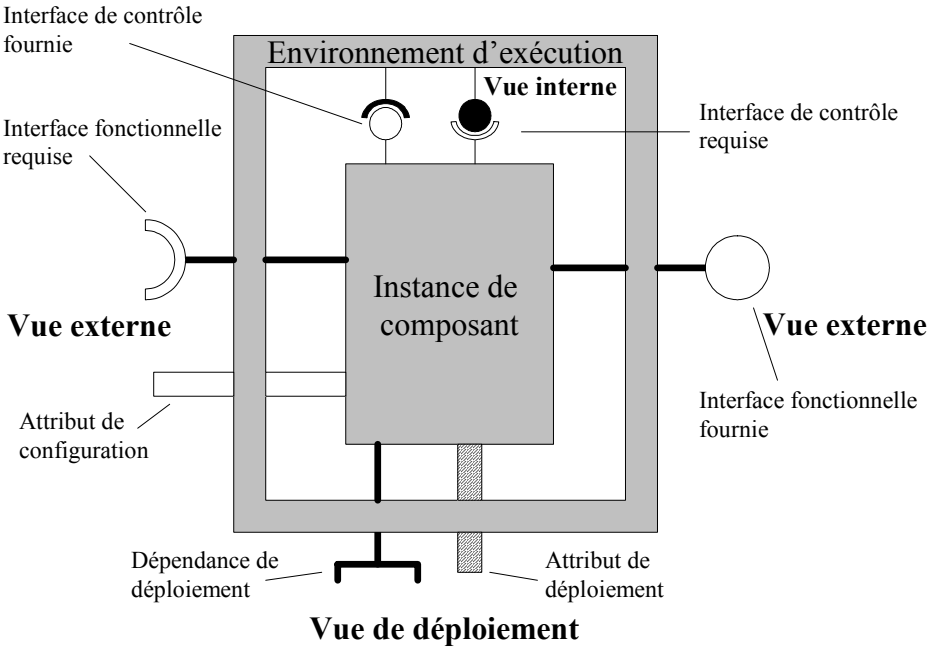


Figure 15. *Représentation des vues d'un composant logiciel.*

La figure 15 donne une représentation schématique d'un composant logiciel sous ses trois vues *externe*, *interne* et *déploiement*.

- **Instance de composant**

Une instance de composant se situe à un niveau équivalent à celui de l'objet dans l'approche objet. Une instance de composant est obtenue à partir d'une classe de composants. Elle fournit

une implantation des interfaces fonctionnelles offertes par la classe de composants. Les interfaces de contrôle sont généralement générées automatiquement par des outils supportant le modèle de composants. Comme la figure 15 le montre, un client n'a pas accès directement à une instance de composant. Les appels de services sont relayés à travers l'environnement d'exécution et transmis à l'instance de composant. Cette architecture permet à l'environnement d'exécution de contrôler le cycle de vie des composants et de leur offrir des services non fonctionnels non fournis par leur implantation (comme la persistance, la sécurité d'accès aux composants, etc.).

L'environnement d'exécution utilise généralement une fabrique de composants (Gamma, 1995) par classe de composants gérée. Le fait que la gestion du cycle de vie soit assurée par la fabrique de composants rend possible l'utilisation de différentes stratégies d'instanciation des composants. Une instance de composant peut avoir ou non un identifiant unique, et peut avoir ou non un état interne. Selon le cas de figure, la stratégie d'instanciation varie :

- Instances multiples : une instance est créée chaque fois qu'un client invoque la méthode de création de la fabrique. Cette stratégie gaspille les ressources système lorsque les instances de composants gérées par le système sont non identifiables et sans état interne (exemple des composants de service).
- Instance partagée : dans ce cas, une instance unique est créée par classe de composants. L'instance est retournée à chaque client demandant la création d'une instance de cette classe de composants. Deux cas de figure se présentent : soit l'instance ne possède pas d'état interne, soit elle est identifiable et possède un état interne, alors cet état sera partagé et visible à tous les clients. L'implantation de cette stratégie peut utiliser le patron Singleton (Gamma, 1995).
- Stock d'instances : dans ce cas, une quantité limitée d'instances est créée. Ces instances sont retournées aux clients dans la limite de disponibilité du stock. Quand un client demande la destruction d'une instance, celle-ci est retournée au stock pour être utilisée postérieurement avec un autre client. Cette politique est utile lorsque les ressources systèmes telles que la mémoire sont limitées ou bien lorsque le coût de création d'une instance est élevé.

- **Paquetage de composants**

Un *paquetage de composants* est l'unité de livraison appelée également unité de déploiement. Il permet la livraison et le déploiement d'une classe de composants de manière *indépendante* de sa phase de développement. Le paquetage doit contenir toutes les informations nécessaires à cette opération : l'environnement d'exécution requis (les ressources matérielles et logicielles), les autres paquetages de composants requis, le code binaire de l'implantation, des ressources comme des fichiers images ou sons, des fichiers de configuration, etc.

- **Les types de composition**

La composition permet la réutilisation et l'assemblage des composants. La technique et le type de la composition dépendent du modèle de composants utilisé. Ainsi nous avons identifié quatre types de composition :

- Composition visuelle : cette approche permet la composition des composants en utilisant une approche graphique et interactive. Les liens entre les composants sont réalisés à l'aide de traits et des boîtes de dialogue permettent de modifier les propriétés des composants.

- Composition déclarative : la composition déclarative est basée sur l'utilisation de langages décrivant des relations entre concepts propres à la composition tels que « composant », « port », « connecteur », etc.
- Composition impérative par langage de programmation : dans ce type de composition, un langage de programmation classique tel que Java est employé pour réaliser la composition. L'inconvénient de cette approche est qu'elle exige des différents acteurs (développeurs et assembleurs) d'avoir des connaissances poussées du langage de programmation utilisé. L'avantage de cette approche est que la composition de plusieurs composants offre un code exécutable, ce qui est synonyme de performance lors de l'exécution.
- Composition impérative par langage de script : dans ce type de composition, un langage de script de haut niveau d'abstraction contenant des concepts spécifiques à la composition est utilisé. Ces langages sont généralement interprétés et non typés, donc moins performants. L'avantage de cette approche est que l'assembleur des composants n'a pas besoin d'avoir les mêmes connaissances que le développeur surtout si le langage d'assemblage est plus simple que celui de programmation.

Les compositions visuelles et déclaratives ont tendance à donner lieu à des architectures statiques alors que la composition impérative permet de réaliser des changements dans l'architecture lors de l'exécution. Pour chacune de ces compositions que l'on pourrait qualifier de syntaxique, il faut naturellement connaître avec précision Les contraintes sémantiques dans l'environnement de développement et d'exécution choisi.

- **Introspection**

L'introspection permet de réaliser l'analyse pendant l'exécution de la structure d'une instance de composant (typiquement sa vue externe) et éventuellement de l'architecture même du SI. L'introspection nécessite que l'environnement d'exécution puisse disposer d'informations concernant la structure des composants et du SI (Marvie, 2001).

3.2 Exemples de modèles de composants logiciels

Nous présentons dans cette section quatre exemples de modèles de composants : les composants COM/DCOM/COM+, les composants EJB, les composants CCM et les composants Fractal. Ces modèles de composants logiciels sont décrits en annexe.

3.2.1 Les composants COM/DCOM/COM+

L'architecture COM (Component Object Model) de Microsoft est la première architecture à base de composants à être disponible en dehors des laboratoires des éditeurs de logiciels. Cette architecture a été proposée en 1994. Elle est maintenant la base de tous les logiciels 32-bits récents de Microsoft incluant la suite Office Windows 9x et NT et les outils de développement.

COM définit un standard binaire pour la définition de l'interface d'un composant (COM, 1994) (Gardarin, 1996) (Rogerson,1997) (Box, 1998) (COM, 2005). Autrement dit, pour chaque combinaison de système d'exploitation et de matériel que Microsoft supporte, COM définit une couche mémoire standard pour une interface COM. Une partie de ce standard est un ensemble spécifique de types de données (comme les nombres entiers flottants) qui peuvent être passés en tant qu'arguments d'opérations dans les interfaces de composants. COM n'est pas seulement une spécification, c'est également une implantation de Microsoft sur chacune de ses plates-formes. Un développeur peut utiliser n'importe quel langage de programmation, orienté objet ou non, parmi plusieurs (C++, Visual Basic, etc.) pour

construire des composants COM. L'indépendance par rapport au langage est un des points forts de COM.

COM était initialement positionné comme une architecture de composants orientée client qui permettait de construire des petites applications, centrées sur l'interface graphique. Mais avec le temps, cette architecture est devenue adaptée pour construire tout type d'application.

Ainsi COM définit un moyen unique d'accès aux services logiciels. Les objets COM fournissent des services au moyen de méthodes groupées en interfaces. Un objet COM (instance de composant COM) a entre une et plusieurs interfaces. Les méthodes de chaque interface sont axées sur la fourniture d'un service particulier. Chaque objet COM est l'instance d'une classe de composant (CoClass). Le seul moyen de communication avec un objet COM est un pointeur sur l'une de ses interfaces.

Un objet COM s'utilise de la même façon quelle que soit sa localisation. Pour l'utilisateur, il offre une vision unique à travers un « pointeur interface » :

- dans le même processus : appel de service rapide car il utilise un simple appel de fonction directe ;
- sur la même machine : appel de service rapide car il utilise une communication inter-processus ;
- entre machines (DCOM) : sécurisé et fiable, grâce au protocole RPC (Remote Procedure Control). L'utilisateur ne voit pas la différence entre un objet COM local ou distant. Le système d'exploitation délègue les appels à travers le réseau avec une parfaite transparence.

DCOM (Eddon, 1998) est l'acronyme de Distributed COM, il permet de faire communiquer des composants COM entre différentes machines. L'environnement d'exécution des composants COM et DCOM est intégré dans le système d'exploitation Windows. COM+ est une extension de COM qui introduit la notion d'environnement d'exécution indépendant du système d'exploitation Windows. Cet environnement offre des services supplémentaires pour la gestion du trafic, de la mémoire, des transactions et de la sécurité.

COM est orienté objet, mais diffère des technologies objet classiques :

- COM dispose du principe d'encapsulation : l'accès à l'état interne d'un objet ne peut se faire qu'à travers son interface.
- La réutilisation se fait par composition d'objets via deux mécanismes : la délégation et l'agrégation. Le modèle de composants COM ne fournit pas de mécanisme d'héritage.
- le polymorphisme : plusieurs composants peuvent implanter la même interface et en fournir différentes implantations. Deux composants sont dits polymorphes pour une interface s'ils l'implantent.

L'annexe B contient plus d'informations concernant les composants COM/DCOM et COM+, notamment les concepts définis par le modèle et l'environnement d'exécution.

3.2.2 Les composants EJB

Le modèle des composants EJB (Enterprise Java Bean) est le fruit d'un travail de normalisation, publié dans sa version initiale en juin 1998 (spécification 1.0). L'objet de cette norme est de définir un modèle standard d'architecture pour les applications distribuées Java construites sur une logique métier. Outre son intérêt technique (formation d'API (Application Programming Interface) ou plutôt d'interfaces d'objets), cette architecture a le mérite de décomposer la réalisation d'une application orientée objet en diverses tâches relevant de la

compétence d'acteurs bien déterminés. Un système d'information est ainsi construit selon une architecture répartie en trois tiers (Roman, 2002) : un tiers de présentation, qui réside principalement dans une machine du côté de l'utilisateur, un tiers applicatif, qui réside dans un serveur, et un tiers de données, correspondant par exemple à une base de données.

L'annexe C développe avec plus de précisions les composants EJB, notamment les concepts définis par le modèle et l'environnement d'exécution.

3.2.3 Les composants CCM (CORBA COMPONENT MODEL)

Les composants CORBA (BEA, 1999) (OMG, 1999) visent des systèmes d'information réalisés à partir de composants hétérogènes distribués. Les composants formant un système d'information ne s'exécuteront donc pas sur un site unique et n'utiliseront pas forcément la même technologie d'implantation.

Dans la définition de l'architecture CORBA (Common Object Request Broker Architecture), l'OMG (Object Management Group) n'a pas abordé directement les problèmes de diffusion et de déploiement d'applications distribuées. Le modèle objet de CORBA fournit uniquement des spécifications pour permettre l'interopérabilité entre objets hétérogènes. Pour cela l'OMG a défini le langage IDL (Interface Definition Language) pour exprimer les interfaces des objets ainsi que le protocole générique GIOP (General Inter-ORB Protocol) dont l'instanciation la plus utilisée est IIOP (Internet Inter-ORB Protocol) qui repose sur TCP/IP. Dans le cas de la norme CORBA 2, toutes les connexions entre objets doivent être réalisées de manière explicite par le développeur.

Suite à l'introduction par Sun de la notion de composant avec les JavaBeans, l'OMG a lancé le RFP (Request For Proposal) sur les composants qui demandait la définition d'un framework de composants dans l'esprit des JavaBeans. Mais entre le RFP et la première proposition, Sun a mis au point les Enterprise Java Beans (EJB). Avec l'apparition de ce nouveau modèle, les groupes qui travaillaient sur le RFP composants ont fait évoluer leur proposition dans l'esprit des EJB. Cette évolution du modèle de composants de CORBA permet de combler un manque dans l'architecture CORBA : la définition d'un framework de composants serveur. Ce framework est aujourd'hui quasiment défini, très inspiré des EJB et totalement compatible avec ce dernier.

La spécification du modèle de composants CORBA (CCM) définit cinq modèles (modèle abstrait, modèle de programmation, modèle de paquetage, modèle de déploiement, modèle d'exécution) et un métamodèle. La spécification du CCM définit un métamodèle, le métamodèle de composants, basé sur UML et qui est fourni avec des projections vers le MOF (Meta-Object Facility).

Une description plus précise des composants CCM, est fournie en annexe D, notamment les concepts définis par le modèle et l'environnement d'exécution.

3.2.4 Les composants Fractal

Fractal (Bruneton, 2004) est un framework de composition qui définit un modèle à composants modulaire extensible et générique. Ce framework peut être utilisé avec différents langages de programmation pour concevoir, implanter, déployer et reconfigurer les systèmes de toute nature : systèmes d'exploitation, intergiciels ou interfaces graphiques. Le modèle de composants Fractal adopte le principe de séparation des préoccupations dans la conception des composants et des systèmes d'information à base de composants. Le principe de séparation des préoccupations préconise la séparation des aspects et des préoccupations d'un système d'information en plusieurs entités distinctes : par exemple, implanter les services fonctionnels fournis par le système d'information dans une entité différente des entités qui assurent les aspects non fonctionnels comme la configuration, la sécurité, et la disponibilité,

etc. En particulier, le modèle de composants Fractal propose trois niveaux de séparation des préoccupations :

- Séparation des interfaces et des implantations : le découplage entre les interfaces et leurs implantations est assuré par le patron de conception *Pont* (Gamma, 1995). Ainsi, les interfaces et les implantations peuvent évoluer séparément, ce qui augmente l'adaptabilité et l'évolutivité des composants et des systèmes d'information à base de composants.
- Programmation orientée composants : ce deuxième niveau correspond à la décomposition des préoccupations d'implantation en plusieurs sous-préoccupations de plus petite taille et composables. Ces sous-préoccupations sont implantées dans des entités séparées appelées composants.
- Inversion du contrôle : la configuration et le déploiement d'un composant Fractal sont délégués à une entité extérieure au composant. Ainsi, les composants ne se préoccupent plus de trouver et de paramétrer les composants et les ressources dont ils ont besoin.

Les concepts définis par le modèle et l'environnement d'exécution concernant les composants Fractal sont développés en annexe E.

3.3 Etude comparative des composants logiciels

La famille de modèles de composants de Microsoft (COM, DCOM, COM+) représente le premier modèle industriel de composants qui a su s'imposer sur le marché. Le modèle COM est une norme de représentation des composants binaires tournant sous les systèmes d'exploitation Windows. Ces modèles ont été définis essentiellement pour satisfaire certaines contraintes requises par les applications Windows, parmi lesquelles :

- assurer une indépendance des applications vis-à-vis des composants qu'elles utilisent (bibliothèques dynamiques de composants) et améliorer la réutilisation du code ;
- faire évoluer séparément les applications et les composants sans recompilation obligatoire en cas de changement. Ceci est garanti par le fait que COM est un standard binaire et qu'il permet la gestion du versionnement ;
- assurer une gestion des composants intégrée directement dans le système d'exploitation Windows, ce qui facilite la tâche du programmeur car il n'aura pas à se soucier si le composant utilisé est local ou s'il tourne sur une machine distante.

Cependant, le modèle COM est une propriété de Microsoft et ne tourne que sur les systèmes Windows, ce qui représente un sérieux inconvénient pour des entreprises qui possèdent souvent des systèmes hétérogènes. C'est pourquoi, face à la complexité croissante des systèmes d'information et au besoin croissant de faire communiquer ces derniers, l'OMG a élaboré la norme CORBA.

CORBA a pour objectif de remédier aux faiblesses des modèles existants et apporte certaines nouveautés. D'abord, CORBA est indépendant de tout langage, de toute plate-forme et de tout vendeur. Il résout ainsi l'interopérabilité entre systèmes d'information implantés avec des langages de programmation différents et tournant sur des machines et des systèmes d'exploitation hétérogènes. Cette indépendance est obtenue grâce à la définition du langage IDL (Interface Definition Language). L'OMG s'étant limité à spécifier CORBA, différentes implantations ont été proposées par des vendeurs différents, ce qui a permis une indépendance de CORBA vis-à-vis des vendeurs. Enfin, CORBA permet une portabilité des codes sources grâce à la définition du framework POA (Portable Object Adapter) implanté par les serveurs

CORBA. Ainsi les composants CORBA s'exécuteront toujours dans un environnement identique quelque soit le fournisseur du serveur CORBA.

Avec la définition de la norme CORBA, la conception et l'implantation des systèmes d'information sont entrées dans l'ère industrielle. Cette évolution a permis l'apparition de nouveaux métiers comme fournisseur de composants, fournisseur de serveurs CORBA, fournisseur de containers, etc.. Les concepteurs de systèmes d'information se concentrent de ce fait plus sur la logique métier du système d'information, essayant de réutiliser ce qu'ils ont et achetant le reste chez des fournisseurs.

L'OMG n'est pas le seul organisme qui a eu l'idée de développer un modèle de composants distribués. Juste après la proposition de l'OMG, Sun a proposé le modèle de composants EJB. Ce dernier vise à remédier à certains problèmes non résolus par l'OMG. La principale nouveauté apportée par les EJB est le fait qu'ils sont de véritables composants orientés objets. En effet la spécification CORBA jusqu'à la version 2.0 gère les objets, mais pas les composants. Mais l'inconvénient des EJB est qu'ils sont écrits uniquement en Java, ce qui peut en même temps être considéré comme avantage car les EJB tirent profit de la puissance de Java et de sa portabilité sur tous les systèmes d'exploitation existants.

Les avantages apportés par les EJB ont poussé l'OMG à améliorer la norme CORBA et a proposer la version 3.0 qui intègre la notion de composant et qui fournit des outils de modélisation au concepteur des composants (CIDL).

France Telecom R&D et l'INRIA ont proposé en juillet 2002 la version 1.0 de la spécification du modèle de composants Fractal et en août 2003 la version 2.0. Le modèle de composants Fractal est un système général, minimal et extensible de *relations* entre des *concepts*. Fractal n'est pas dédié à un langage ou à un environnement d'exécution particulier. C'est un modèle général qui ne présuppose pas une sémantique ou une granularité particulière associée aux composants à l'instar des modèles de composants industriels EJB et CCM qui implantent des composants métier, de plutôt gros grain, dont les conteneurs fournissent d'une manière plus ou moins transparente certains services d'infrastructure (propriétés non fonctionnelles). Fractal est également moins figé qu'EJB et CCM. C'est au contraire un modèle minimal et extensible : « tout est à la carte » dans Fractal y compris les systèmes de types ou les capacités réflexives des composants. Le modèle de composants Fractal est intrinsèquement récursif et réflexif. Les composants Fractals sont auto-similaires (d'où le nom de Fractal) : quelque soit l'échelle à laquelle on les observe, ils apparaissent toujours comme étant la composition d'un contrôleur et d'un contenu. Les contrôleurs permettent une introspection totale de la structure des composants et permettent une navigation au sein des assemblages de composants. Cette réflexivité structurelle est essentielle pour l'administration et plus généralement pour le contrôle (monitoring, reconfiguration, etc.). Le modèle de composants Fractal donne la possibilité aux concepteurs d'applications de disposer de composants qui se recouvrent mutuellement. Ainsi un composant donné peut faire partie de plusieurs composants, d'où la possibilité d'utiliser l'agrégation au lieu d'utiliser systématiquement la composition comme dans les solutions EJB et CCM. Cette approche se révèle particulièrement bien adaptée à des composants modélisant des ressources. Enfin, le modèle de composants Fractal permet le contrôle lors du déploiement et de l'exécution. Cet aspect peut être considéré comme l'apport majeur par rapport aux autres modèles de composants industriels. Fractal est en effet un modèle essentiellement structurel (centrée sur l'architecture) qui permet de partitionner un système complexe en sous-éléments manipulables. Les composants sont présents à l'exécution, ce qui évite une dissémination des composants dans le code.

La figure 16 présente le positionnement des quatre modèles de composants logiciels que nous avons présentés dans cet état de l'art par rapport à leur domaine d'application et leur aspect architectural. Le modèle EJB est utilisé pour la construction des applications de gestion

tournées vers l'internet. Le modèle CCM est utilisé dans les applications middleware Internet, telecom, supervision industrielle, calcul numérique et parallélisme. La famille des modèles de composants COM est plus généraliste que EJB et CCM car elle permet de construire des applications variées, y compris des parties du système d'exploitation de Microsoft ; par contre elle est peu orientée architecture. Le modèle de composants Fractal est le plus généraliste et le plus orienté architecture. En effet comme nous l'avons dit, il permet de construire des composants logiciels quelconques : vision intégrée des systèmes d'exploitation (bas niveau avec *Think*), des interfaces graphiques, des middlewares et des systèmes d'information (haut niveau).

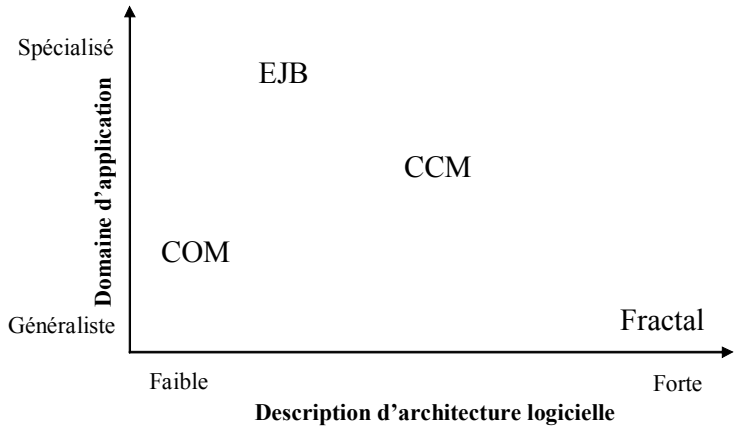


Figure 16. *Positionnement des modèles de composants.*

Le tableau 1 présente les caractéristiques des composants logiciels en tant que composants réutilisables (cf. section 1.5). Le tableau 2 et le tableau 3 définissent des critères de comparaison permettant de bien cerner les différences et les points communs et les différences entre les modèles de composants logiciels étudiés dans les sections précédentes.

Tableau 1. *Caractéristiques des composants logiciels réutilisables.*

Critères	**Les composants logiciels**
Type de connaissance	Processus / Produit
Couverture	Général / Domaine / Entreprise
Portée	Implantation
Technique de réutilisation	Spécialisation / Composition / Assemblage
Granularité	Généralement faible
Architecture	Locale / Distribuée

Tableau 2. *Tableau comparatif des modèles de composants logiciels (1/2).*

		COM/DCOM/COM+	**CCM**	**EJB**	**Fractal**
Interfaces	Interfaces offertes	N interfaces	N interfaces	1 objet	N interfaces
	Interfaces requises	N sources (COM+)	N réceptacles		N interfaces
	Connecteurs	Synchrone / asynchrone	Synchrone / asynchrone		Synchrone
	Langage de description d'interfaces	Non	Oui	Non	Oui
	Langage de description de composants	Non	Oui	Non	Oui
	Rétrospection	Oui	Uniquement pour les facettes	Oui	Oui
Réutilisation	Démarche de Conception/Développement	Non	Non	Non	Non
	Référentiel (service de nommage)	Oui	Oui	Non (mais utilise JNDI)	Oui
	Techniques	Agrégation et encapsulation	Agrégation	Héritage et agrégation	Agrégation
Composition	Langage d'assemblage	Non	Descripteur d'assemblage de composants	Non	ADL-Fractal
	Technique d'assemblage	Encapsulation et agrégation	Par liaison de connecteurs	Encapsulation et agrégation	Par liaison de connecteurs
	Type de composition	Impérative à travers un langage de programmation ou un langage de script	Composition déclarative	Impérative à travers le langage de programmation Java	Impérative, déclarative et visuelle
Déploiement	Unité de déploiement	DLL ou EXE ne contenant pas de ressources	Fichier Zip contenant du code d'implantation + des descripteurs de déploiement	Fichier Jar pour composants, EAR pour applications + descripteurs de déploiement	Non spécifié

Tableau 3. *Tableau comparatif des modèles de composants logiciels (2/2).*

		COM/DCOM/COM+	CCM	EJB	Fractal
Environnement de développement et plates-formes	Langages de développement	Visual Studio	Multiples	Java	Multiples
	Plates-formes	Windows	Toutes plates-formes	Toutes plates-formes	Toutes plates-formes
Environnement d'exécution	Fournisseurs de middleware	Microsoft	Plusieurs	Nombreux (30+)	Open source
	Middleware fourni avec le système d'exploitation	Oui	Non	Non	Non
	Runtime	Runtime COM	Intergiciel CORBA + conteneurs	Runtime EJB + conteneurs	Julia, Think, FraclTalk Proactive
Composant	Composants avec état interne	Non	Oui	Oui	Oui
	Composants sans état interne	Oui	Oui	Oui	Oui
	Composition de composant = un composant	Oui	Non	Oui	Oui
	Besoin de container	Non	Oui	Oui	Non

D'après cette étude comparative, nous pouvons dire qu'il n'existe pas de modèle idéal qui permette de résoudre tous les problèmes rencontrés par les concepteurs de systèmes d'information. Par exemple, EJB n'utilise que des composants Java, ce qui limite le champ de l'interopérabilité. De plus, il ne possède pas la notion de port qui se trouve dans CCM. Par contre, CCM ne possède pas certains avantages des EJB, par exemple une agrégation de composants n'est pas un composant. La famille des composants COM est la seule qui définisse les composants en terme d'une norme binaire, ce qui facilite par la suite la communication et la gestion des versions et des bibliothèques dynamiques de composants. Fractal peut être vu comme le modèle de composants « à tout faire », mais il est trop généraliste et son utilisation peut ne pas être le meilleur choix pour construire des applications spécifiques.

Nous pouvons donc conclure qu'à chaque besoin correspond un modèle de composants adéquat. De plus, les dernières versions des modèles de composants EJB et CCM ont défini des ponts (interfaces et protocoles) pour permettre la communication et l'interopérabilité entre systèmes d'information utilisant ces middlewares. Ceci ne peut qu'être positif pour les concepteurs de systèmes d'information.

La question qui se pose maintenant est de savoir si cette évolution va s'arrêter ou si on va voir apparaître prochainement d'autres modèles de composants plus efficaces utilisant d'autres technologies. Si l'évolution continue, les entreprises seront obligées de remettre en question la conception de leurs systèmes d'information chaque fois qu'une nouvelle technologie apparaîtra. Pour répondre à cette éventualité, l'OMG a introduit l'approche MDA (OMG, 2005) (Blanc, 2005) grâce à laquelle les entreprises pourront concevoir leurs systèmes d'information indépendamment du modèle de composants qui servira à l'implantation.

4. Les composants métier

4.1 Introduction

Les composants métier sont utilisés pour concevoir et réaliser des systèmes d'information adoptant un comportement proche du métier qu'ils supportent. Dans de telles approches, on cherche d'une part à représenter le domaine à un niveau générique, et d'autre part, à expliciter sa variabilité (Wartik, 1992).

Enfin, les composants métier sont parfois présentés comme regroupant un aspect conceptuel et un aspect logiciel. Par exemple, F. Barbier (Barbier, 2002a) affirme :

> *« A business component models and implements business logic, rules and constraints that are typical, recurrent and comprehensive notions characterizing a domain or business area ».*

Dans le cadre de ce mémoire, nous préférons utiliser le terme de composant métier où la réutilisation apparaît plus explicite que dans le terme d'objet métier. De plus, nous adoptons une vision bi-dimensionnelle du composant métier : de notre point de vue, un composant métier est défini au niveau conceptuel par la représentation de concepts métier qu'il représente, et se décline jusqu'au niveau logiciel en aidant à l'implantation de ces concepts métier. Cette implantation logicielle se traduit généralement par l'utilisation d'un ou de plusieurs composants logiciels selon deux facettes : une facette fonctionnelle éventuellement complétée par une facette non-fonctionnelle. Notre vision d'un composant métier couvre donc un spectre large résumé dans la figure 17.

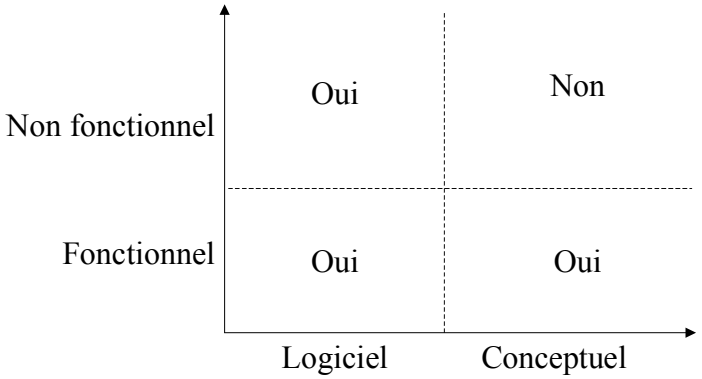


Figure 17. *Domaines couverts par les composants métier.*

Les composants métier fournissent donc des spécifications et des réutilisations compréhensibles à la fois par les techniciens et les experts du domaine. Le fait qu'ils soient indépendants des plates-formes d'exécution facilite leur distribution en tant qu'unités autonomes vers des plates-formes sur lesquelles ils seront installés.

Les composants métier nécessitent une infrastructure technologique pour supporter leur intégration dans les systèmes d'information. Une telle infrastructure a été en particulier définie par l'OMG sous le nom de BOF (Business Object Facility). Le BOF offre une sémantique pour supporter les composants métier. L'OMG définit le BOF comme :

« L'infrastructure (architecture d'applications, services, etc.) nécessaire pour supporter les composants métier comme composants d'applications coopératifs dans des environnements distribués : l'infrastructure technologique qui supporte les composants d'applications « plug and play » est le BOF. Cette infrastructure doit avoir un certain nombre de sémantiques métier pour supporter les composants métier, qu'ils soient communs ou encore spécifiques pour une entreprise ».

Nous présentons dans cette section deux exemples de modèles de composants métier : le modèle de l'OMG et le modèle de composants métier Symphony.

4.2 Les composants métier selon l'OMG

Selon l'OMG (OMG, 1994), un composant métier (CM) est une représentation d'un concept actif dans le domaine d'un métier, incluant au moins son nom, sa définition, ses attributs, son comportement ; ses relations et ses contraintes :

- **Nom Métier :** le terme utilisé par les experts pour désigner le CM.
- **Définition Métier :** la signification et le but assigné, par des experts, à un composant métier.
- **Attribut :** les faits pertinents concernant le CM pour répondre aux caractéristiques du métier.
- **Comportement :** les actions (ou services) qu'un CM est capable d'exécuter dans le but d'accomplir son objectif, c'est-à-dire reconnaître les événements dans son environnement, changer ses attributs et interagir avec les autres CM.
- **Relations :** les liaisons avec d'autres composants métier qui reflètent les interactions qui peuvent être unidirectionnelles ou bidirectionnelles.
- **Règles Métier :** les contraintes qui régissent le comportement, les relations et les attributs de l'objet métier.

Dans cette définition, le comportement du CM correspond au contrat de services du composant et les relations mettent en évidence les collaborations qu'il entretient avec les autres CM. L'implantation du composant peut être assimilée aux autres rubriques (attributs, nom métier, définition métier, règles métier).

Une représentation partielle d'un exemple de CM dans le modèle de l'OMG est donnée dans la figure 18.

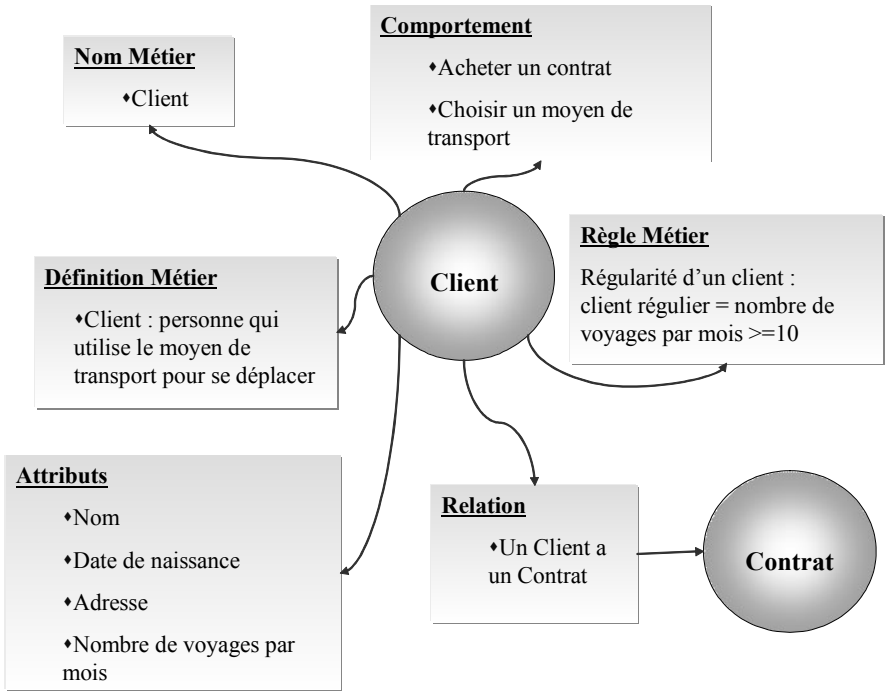


Figure 18. *Représentation du composant métier Client dans le modèle de l'OMG.*

4.3 Le modèle de composants métier Symphony

Symphony est un processus de développement logiciel élaboré par la société Umanis[2] (Hassine, 2002) en collaboration avec l'équipe SIGMA du laboratoire LSR-IMAG. Comme tout processus de développement, Symphony a pour but de spécifier les différentes phases d'un projet, de définir les tâches de chacun des intervenants et de contrôler les coûts, les délais et la qualité de l'application logicielle produite. Symphony se présente sous forme d'un guide méthodologique offrant une solution basée sur l'utilisation de composants dès les phases amont du processus. Cette démarche s'appuie sur le langage unifié UML et est construite autour d'un certain nombre de pratiques. Elle est itérative, incrémentale, orientée utilisateur, orientée composants, pilotée par les cas d'utilisation et adopte un modèle de cycle de vie en Y (André, 1994) (Larvet, 1994) (cf. figure 19).

- Symphony est orientée utilisateur, car la spécification et la conception sont construites à partir des modes d'utilisation attendus par les acteurs finaux du système.
- Symphony est orientée composant, car tant au niveau conceptuel que logiciel une application est vue comme un assemblage de composants indépendants et interconnectés. Cette pratique garantit une souplesse des modèles métier et logiciel de la démarche. Elle constitue un support nécessaire à la réutilisation et offre des perspectives de gains considérables.

[2] http://www.umanis.com

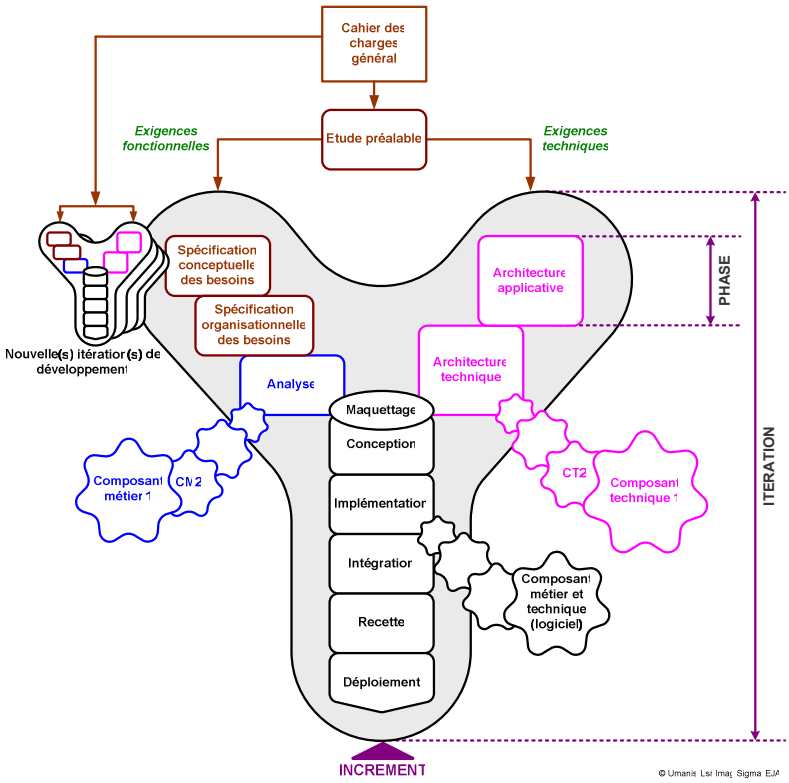


Figure 19. *Cycle de vie de la démarche Symphony (Hassine, 2005).*

La figure 19 met en évidence que les Composants Métier (resp. Composants Techniques) sont identifiés et analysés dans la branche « fonctionnelle » (resp. Technique). Dans la branche centrale du Y (conception, implantation, etc.), les composants conceptuels (métiers-CM et techniques-CT) sont progressivement combinés et transformés en composants logiciels.

4.4 Conclusion

Les concepts de base du modèle de composants métier de l'OMG sont très intuitifs et peuvent assez facilement être traduits en UML. Les attributs et le comportement d'un composant métier sont représentés par des attributs et des opérations dans une classe représentant le composant. Les relations sont traduites par des associations, des aggrégations et des compositions. Les règles métiers peuvent être exprimées sous la forme de contraintes OCL.

Symphony est une démarche de développement à base de composants et s'appuie sur un modèle conceptuel de composants métier (structure tripartie). Le modèle de composants Symphony est décrit en annexe F. La démarche est formalisée sous la forme d'un système formé de plusieurs patrons processus supporté par l'atelier AGAP[3] (Conte, 2001) (Conte, 2001a) élaboré au sein de l'équipe SIGMA. Les composants métier utilisés dans Symphony et la démarche elle-même sont exprimés en UML.

L'étude de ces deux modèles de composants métier nous permet de conclure qu'il est possible d'utiliser le langage de modélisation unifié (UML) pour la modélisation de la description et la conception de ces composants.

Dans le tableau 4 nous présentons les caractéristiques (cf. section 1.5) des composants métier en tant que composants réutilisables.

[3] Atelier de Gestion et d'Application de Patrons.

Tableau 4. *Caractéristiques des composants métier.*

Critères	**Les composants métier**
Type de connaissance	Processus / Entité
Couverture	Général / Domaine / Entreprise
Portée	Les composants métier couvrent toutes les phases du processus de développement : - Analyse : les CM ont des caractéristiques et des comportements qui leur permettent d'être utilisés naturellement dans la modélisation de produits et de processus métier, dans leurs relations et leurs interactions. - Conception : les CM représentent des concepts du monde réel, ce qui permet de concentrer la conception sur la manipulation de « grosses » entités avec une bonne maîtrise. - Implantation : les CM ont des liens et des interfaces bien définis qui leur permettent d'être implantés indépendamment les uns des autres.
Technique de réutilisation	Spécialisation / Composition
Granularité	Généralement faible
Architecture	Locale / Distribuée

5. Conclusion

Ce chapitre a présenté un état de l'art sur les composants. Trois catégories de composants sont identifiées et décrites : les composants conceptuels, les composants logiciels et les composants métier.

Différents types de composants susceptibles d'être utilisés lors du processus de développement de systèmes d'information existent. Pendant chacune des phases de développement (analyse, conception, implantation), Les acteurs d'une équipe de développement de systèmes d'information sélectionnent et réutilisent des composants qui répondent partiellement ou totalement à leurs besoins. Il est évident que la recherche d'un composant varie énormément selon son niveau d'abstraction et son modèle de composants. La construction d'un système de gestion de base de composants capable de gérer une collection hétérogène de composants (en termes de niveaux d'abstraction et de modèles de composants) nécessite de disposer de différentes techniques de recherche de composants pour répondre aux besoins spécifiques de chaque acteur et pour s'adapter aux caractéristiques spécifiques des composants. C'est l'objet des chapitres suivants.

III. Les approches de Recherche de Composants

La recherche d'information (RI) occupe une place importante dans les systèmes d'information. En effet, s'il est important de savoir modéliser l'information, il est également nécessaire de pouvoir y accéder facilement. L'augmentation du nombre de documents au niveau des entreprises et des institutions ainsi que l'avènement des documents électroniques, des documents multimédias et de l'Internet nécessite la mise en place de systèmes sophistiqués de recherche d'information.

De la même manière, la recherche de composants est fondamentale pour l'ingénierie des systèmes à base de composants. L'augmentation du nombre de modèles de composants ainsi que l'avènement des composants sur étagère (Components Off The Shell) nécessite la mise en place de systèmes sophistiqués de recherche de composants.

Ce chapitre présente un état de l'art des différentes techniques permettant la recherche de composants. il présente tout d'abord une introduction au domaine de la recherche de composants. La deuxième section est consacrée à une identification et classification des différents types de systèmes de recherche de composants. La troisième section présente les différentes techniques de recherche de composants que nous avons identifiées. Enfin, la quatrième section propose un ensemble de critères d'évaluation de techniques de recherche de composants permettant une étude comparative entre les différentes catégories de techniques de recherche de composants. Cette dernière section nous permet de mettre en évidence les limites des techniques de recherche de composants actuelles et de l'intérêt d'en proposer une plus adaptée et plus ouverte.

1. Introduction

La recherche d'information (RI) est un domaine relativement ancien (plus de 30 ans) qui a connu ces dernières années une évolution rapide surtout après l'avènement de l'Internet et des documents multimédias. Cependant les principes de base des techniques de RI ont peu évolué ; seuls les corpus documentaires ont changé. Un composant pouvant être assimilé à un document électronique, nous examinons dans un premier lieu s'il est possible d'appliquer les techniques de recherche d'information sur une collection de composants. Avant d'aller plus loin dans la présentation du domaine de la recherche d'information en général et du domaine de la recherche de composants en particulier, définissons le vocabulaire que nous utilisons dans la suite de ce mémoire.

- **Composants et artefacts**

Le chapitre 2 définit un composant comme une unité réutilisable de conception (de n'importe quel niveau d'abstraction) décrite selon un modèle de composants.

Dans la littérature, on trouve aussi le concept d'artefact (en anglais asset ou artefact). La notion d'artefact est encore plus générale et n'inclut pas seulement la notion de composant, mais elle peut s'étendre à la notion de procédure, module, classe, modèle, documentation, spécification, donnée de test, etc. Dans la suite de cet état de l'art, nous ne distinguons pas la notion de composant de la notion d'artefact car nous considérons qu'un composant peut être assimilé à un ensemble d'artefacts. Rechercher les artefacts revient donc à rechercher les composants. Nous considérons les artefacts et les composants comme étant des documents électroniques. Lors de la présentation des techniques de RI appliquées aux composants, nous parlons uniquement de composants et de bases de composants.

- **Base de composants**

Une base de composants est une collection de composants maintenue selon une organisation permettant la recherche et la sélection des composants.

- **Requête**

Une requête est une expression de termes qui décrivent les besoins en information de l'utilisateur. Le concept de terme est souvent associé à une expression textuelle sous la forme d'un mot. Dans ce mémoire, nous adoptons le sens général du concept : un terme peut être un mot, un diagramme, une image, etc.

2. Systèmes de recherche de composants

Il existe fondamentalement deux approches de systèmes de recherche de composants : l'approche par navigation et l'approche par requêtes (cf. figure 20). Dans l'approche par navigation, le système de recherche d'information exploite les liens sémantiques qui peuvent exister entre les composants pour faire de la navigation. Dans l'approche par requêtes, le principe de base consiste à faire correspondre une requête à une collection de composants et à retourner les composants pertinents à l'utilisateur. Le succès de ce type de systèmes dépend en partie de la qualité et de la quantité d'informations associées à la requête et aux composants de la base de composants. En effet avec une grande quantité d'informations définissant la pertinence des composants, le système sera en mesure d'employer des techniques plus avancées pour classer les composants pertinents et non pertinents.

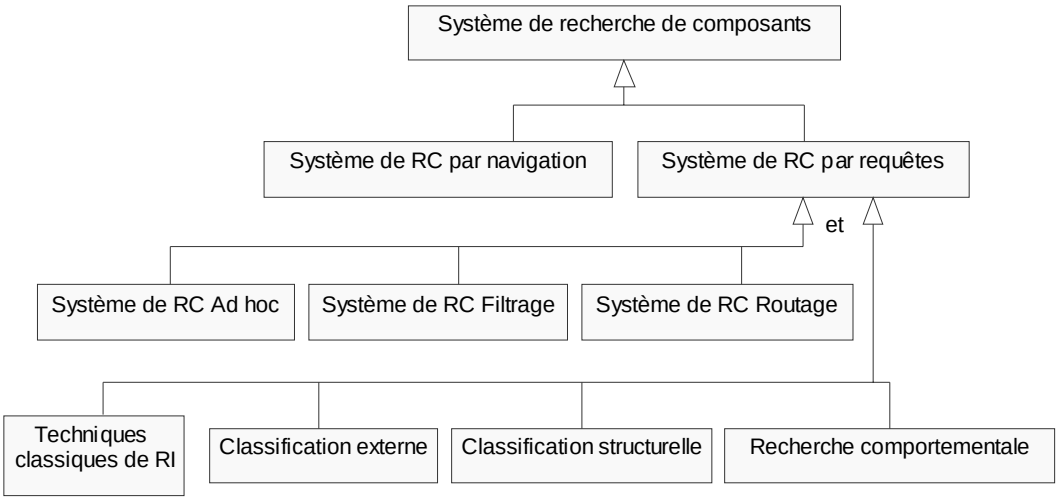


Figure 20. *Classification des systèmes de recherche de composants.*

Les systèmes à base de requêtes sont classés selon deux classifications orthogonales : le scénario d'utilisation (*ad hoc*, *routage* et *filtrage*) et la famille de techniques de recherche de composants qu'ils adoptent (*technique classique de RI*, *classification externe*, *recherche structurelle* et *recherche comportementale*).

2.1 Classification par scénario d'utilisation

La classification par scénario d'utilisation des systèmes de recherche de composants par requêtes se divise en trois sous catégories : la recherche *ad hoc*, le *routage* et le *filtrage* (Hull, 1994).

- **Recherche ad hoc**

Dans la recherche ad hoc, l'utilisateur pose des requêtes auxquelles le système est censé répondre en renvoyant un ensemble de composants ordonnés selon une mesure de pertinence. On considère généralement que le corpus est fixe et que les requêtes sont totalement ouvertes (formulées par l'utilisateur). Après avoir reçu une réponse, l'utilisateur peut reformuler sa requête pour affiner la recherche (cf. figure 21).

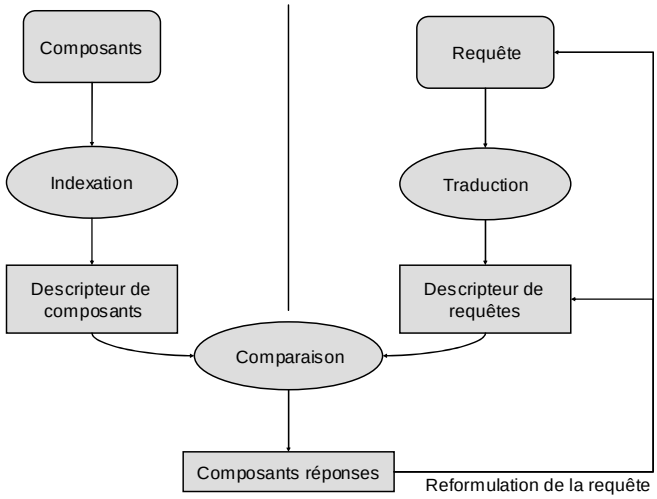


Figure 21. *Architecture classique d'un système de recherche d'information ad hoc appliqué aux composants.*

- **Routage**

Dans la technique de routage, les requêtes sont fixées, et il s'agit d'ordonner un ensemble de composants par rapport à ces requêtes. Chaque requête définit une classe de composants. Il s'agit donc d'un problème de classification pour lequel on désire ordonner les composants par rapport à chaque classe (cf. figure 22). Les corpus sont généralement dynamiques, ils peuvent être volatiles ou évoluer au cours du temps.

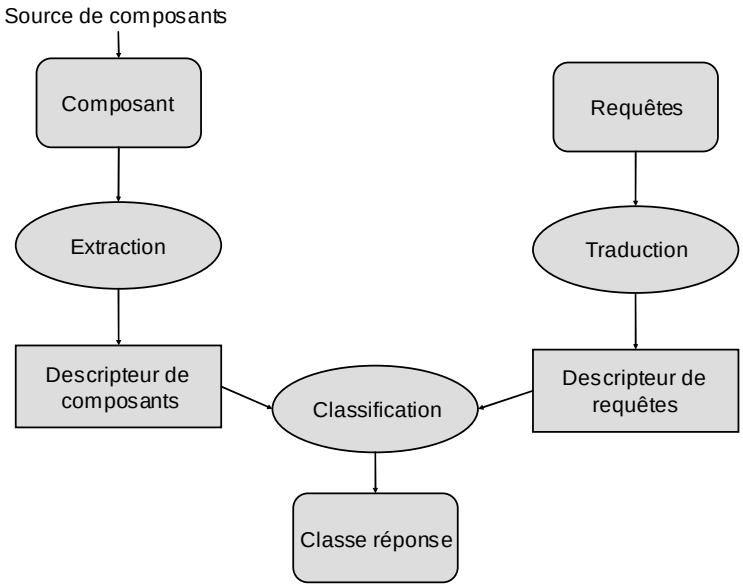


Figure 22. *Architecture classique d'un système de recherche d'information routage appliqué aux composants.*

- **Filtrage**

La fonction d'un système de filtrage est le tri d'un flux de composants issu de sources de composants, puis la présentation à l'utilisateur des composants qui correspondent à sa requête. C'est aussi une tâche de classification, mais cette fois-ci on se limite à prendre une décision sur la pertinence d'un composant pour une classe. A la différence du routage, il s'agit d'une décision binaire.

Nous avons présenté dans cette section une classification des systèmes de recherche de composants en trois catégories selon leurs scénarios d'interaction avec l'utilisateur. Chacune de ces trois catégories peut être utile pour la gestion des composants dans une approche de développement par réutilisation de composants. En effet le « filtrage » peut aider une équipe de développement de systèmes d'information à faire de la veille et à rester à l'écoute des fournisseurs de composants du marché et ne prendre selon ses besoins que les composants qui vérifient certaines contraintes de qualité. Le « routage » peut être utilisé après le « filtrage » pour classer les composants susceptibles d'intéresser l'équipe. La recherche « ad hoc » servira aux ingénieurs pour pouvoir sélectionner les composants dont ils ont besoin pour construire un système d'information. Enfin la navigation est un très bon outil pour explorer le contenu d'une base de composants ou bien raffiner les résultats obtenus à partir d'une approche de recherche de composants « ad hoc ».

2.2 Classification par famille

Les systèmes de recherche de composants adoptant une approche par requête peuvent être classés selon la famille de techniques de recherche de composants qu'ils adoptent :

- **les techniques classiques de recherche d'information** peuvent être appliquées directement sur les composants (code source).

- **les techniques de classification externe** indexent une représentation externe des composants. Cette représentation est souvent préparée manuellement sous forme textuelle (par exemple, une documentation en langue naturelle).

- **les techniques de classification structurelle** exploitent les propriétés structurelles des composants comme les signatures et les spécifications des services offerts par les composants.

- **les techniques de recherche comportementale** exploitent la propriété d'exécutabilité des composants logiciels pour les classer. La classification se fait en mesurant certains critères dynamiques comme le temps d'exécution ou la mémoire consommée, ou encore en utilisant les traces d'exécution.

Ces techniques de recherche de composants sont décrites dans la section suivante.

3. Les techniques de recherche de composants

Cette section détaille chacune des techniques de recherche de composants introduites dans la section 2.2.

3.1 Les techniques classiques de recherche d'information

Un composant peut être considéré comme un type particulier de documents. Il est donc trivial de tenter d'appliquer les techniques classiques de recherche d'information sur les composants. Dans cette section nous ne présentons pas un état de l'art des techniques de recherche

d'information, mais nous citons quelques expériences d'application sur les bases de composants.

Dans (Frakes, 1987) et (Frakes, 1987a), Frakes et Nejmeh appliquent la technique de recherche d'information textuelle sur une base d'artefacts logiciels (des codes sources en langage C). Les artefacts sont indexés par des termes extraits des entêtes et des commentaires se trouvant dans les codes sources. L'opération d'indexation est entièrement automatisée pour garantir une uniformité des termes d'indexation. Cette approche d'indexation dépend fortement des habitudes individuelles des programmeurs pour commenter le code source. De plus, la méthode n'utilise pas un vocabulaire contrôlé, ce qui demande un effort supplémentaire des programmeurs pour trouver les bons termes et des utilisateurs du système pour trouver ces mêmes termes.

Les techniques textuelles de recherche d'information sont appliquées sur les codes sources dans le domaine de la reingénierie des systèmes d'information. Elles aident l'ingénieur à comprendre la structure du code source et lui permettent de retrouver des constructions spécifiques (des patterns). Le système ESCAPE (Paul, 1994) analyse les codes sources pour extraire des représentations sous forme de graphes syntaxiques abstraits. Un graphe syntaxique abstrait est une représentation du code source dont les nœuds représentent le code source (exemple : *while-statement*, *relational-expression*, *statement-list*, etc.). L'ingénieur utilise une interface graphique pour exprimer sa requête qui est traduite par la suite sous la forme d'une expression algébrique en utilisant le modèle algébrique de code source défini par les auteurs. Enfin, le système applique des optimisations sur la requête et la compare à sa base des graphes extraits des codes sources.

Mishne et Rijke (Mishne, 2004) utilisent une approche similaire à celle utilisée dans le système ESCAPE. Ils exploitent le formalisme des graphes conceptuels pour décrire les codes sources et les requêtes. L'utilisateur soumet sa requête sous la forme d'un code source. Le système extrait une représentation sous forme d'un graphe conceptuel, puis la compare aux graphes conceptuels qui indexent les codes sources de la base.

3.2 Les techniques de classification externe

Cette section présente la famille des techniques de classification externe de composants. La figure 23 rappelle que ces techniques n'indexent pas les composants directement, mais plutôt leur représentation.

Figure 23. *Modèle d'indexation externe.*

3.2.1 Approches par mots-clés

Matsumoto (Matsumoto, 1987) représente les composants par un ensemble de mots-clés. L'indexation des composants dans la base est une simple classification par mots-clés. Karlsson (Karlsson, 1995) étend la technique de classification par mots-clés et associe un poids à chaque mot-clé. Karlson utilise donc des relations floues pour associer les termes d'indexation aux composants. La technique de recherche de composants par mots-clés a montré rapidement ses limites pour différentes raisons : lorsque la taille de la base de composants devient importante, il est difficile d'assurer la consistance de l'indexation des composants. De plus, la précision de la recherche chute rapidement lorsque les composants sont issus de plusieurs domaines.

3.2.2 Approches par langage naturel

Maarek, Berry et Kaiser (Maarek, 1989) (Maarek, 1991) présentent un système de recherche de composants basé sur l'application de techniques textuelles de recherche d'information sur des descriptions des composants en langage naturel. L'interrogation de la base de composants se fait à travers des requêtes en langage naturel. Chaque document (description textuelle d'un composant) est analysé pour l'extraction d'un ensemble de termes d'indexation. Les termes d'indexation extraits d'un document constituent le descripteur qui sera utilisé pour la correspondance avec les requêtes utilisateurs. L'implantation de ce système s'appelle *GURU*. L'évaluation des performances du système *GURU* s'est faite avec une collection de documents (composants) et de requêtes de test en évaluant l'effort de l'utilisateur, l'effort de maintenance, la précision et le rappel.

Helm et Maarek (Helm, 1991) font évoluer l'approche utilisée dans *GURU* pour indexer des bibliothèques de classes orientées objet en exploitant les informations supplémentaires fournies par la relation d'héritage. Cette technique peut être classée dans la catégorie des techniques de classification structurelle, mais nous la présentons dans cette section car elle s'appuie sur l'extraction de la structure des composants à travers l'analyse des descriptions textuelles et non pas des composants eux-mêmes.

Devanbu, Brachman, Selfridge et Ballard (Devanbu, 1991) proposent un système de recherche de composants appelé Lassie (Large Software System Information Environment). Lassie est composé d'une base de connaissances, d'un algorithme de recherche sémantique basé sur l'inférence formelle et d'une interface utilisateur avancée. La base de connaissances de Lassie est construite en utilisant un langage de représentation de connaissances basé sur la classification. Les représentations sont construites grâce à l'analyse des descriptions textuelles en langage naturel des composants. La base de connaissances est utilisée comme un index pour la recherche de composants réutilisables. Elle est structurée sous la forme d'une taxonomie de nœuds de quatre types (action, objet, acteur et état) reliés par le lien *IsA* et d'autres relations qui expriment les relations du niveau implantation comme *function-calls-function*, *sourcefile-includes-headerfile*, etc. Une requête Lassie est la description d'une action avec un certain nombre de paramètres. Lassie évalue la requête par la recherche dans la base de connaissances d'une instance qui vérifie ses paramètres.

ROSA (Reuse Of Software Artefact) est une base de composants qui exploite des descriptions textuelles des composants pour les classer et les rechercher (Girardi, 1993), (Girardi, 1994), (Girardi, 1995). Dans les approches textuelles de recherche d'information, les informations d'indexation sont extraites en effectuant une analyse lexicale, syntaxique et sémantique sur la description en langage naturel des composants. Cela revient à supposer que la description en langage naturel décrit avec précision le composant, d'où la faiblesse de cette approche. Le mécanisme d'interprétation automatique utilisé pour l'analyse des composants utilise des techniques linguistiques pour rechercher les descriptions qui correspondent le mieux aux besoins exprimés par la requête. Le mécanisme d'interprétation représente donc une deuxième source de difficultés. Cette méthode peut donner de bons résultats, mais elle reste difficile à mettre en œuvre. Elle est limitée à des bases de composants spécifiques pour des domaines restreints.

3.2.3 Approches par facettes

ASSET (Asset Source of Software Engineering Technology) (ASSET, 1993) est une base de composants conçue par l'agence ARPA (Advanced Research Projects Agency) sous le programme STARS. Un composant ASSET est défini comme un composant logiciel réutilisable ou un document qui décrit des aspects de la réutilisation ou du génie logiciel. Un composant est décrit par les informations suivantes : nom, autres noms alternatifs, date de création, taille, type de composant, code de distribution, domaine, mots-clés, numéro de

référence, langage d'implantation, environnement d'exécution, format, nom d'auteur, résumé, support disponible, personne contact, producteur, fournisseur, date d'approvisionnement. La base de composants ASSET permet d'effectuer une recherche selon les facettes (cf. figure 24) utilisées pour la description des composants. Chaque facette représente une information permettant d'identifier et de sélectionner un composant. Une facette est définie par son nom et son espace de termes appelé vocabulaire. Un vocabulaire est l'ensemble des termes possibles qui permettent de décrire les aspects de la facette. Un exemple de facette est la facette langage d'implantation qui peut prendre comme valeur ADA, C, Java, etc. L'approche de classification par facettes a été utilisée dans d'autres systèmes de recherche de composants comme (Prieto-Diaz, 1985), (Prieto-Diaz, 1987), (Prieto-Diaz, 1991), (Poulin, 1995) et (Zhang, 2000). Cette technique est puissante et donne de bons résultats à condition de bien choisir les facettes, de bien indexer les composants en donnant les bons termes et de bien définir l'espace des termes. La classification par facettes nécessite une indexation manuelle souvent coûteuse en temps et en argent.

Cette technique est analogue à la définition d'une base de données possédant les informations descriptives (facettes) des composants.

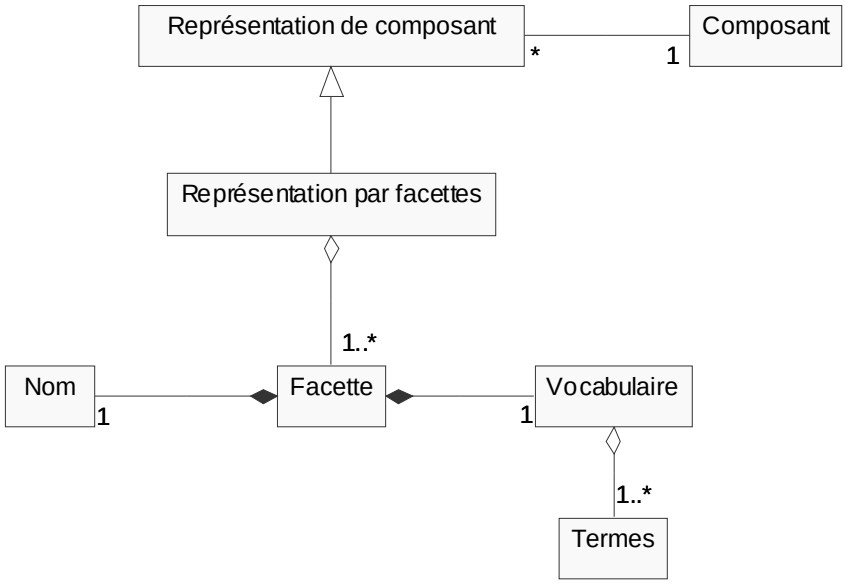


Figure 24 : *Modèle de représentation par facettes.*

3.3 Les techniques de classification structurelle

Les techniques qui s'intéressent à l'aspect structurel pour la classification des composants se divisent en deux catégories : les techniques d'appariement de signatures et les techniques d'appariement de spécifications. Les approches d'appariement de signatures sont en réalité un sous-ensemble des approches d'appariement de spécifications. En plus de l'aspect signature (interfaces), une approche par spécification s'intéresse à la transformation effectuée sur les données et à la conception interne du composant. Nous faisons cette distinction entre les deux catégories car les approches de classification par signatures sont exploitables avec les composants de type boîte noire, boîte en verre et boîte blanche. Les approches de classification par spécification ne sont par contre applicables qu'avec les composants de type boîte blanche et boîte en verre.

3.3.1 Techniques d'appariement de signatures

La figure 25 présente le modèle de représentation par signatures des composants. Un composant logiciel offre généralement ses services à travers des opérations regroupées dans des interfaces. La signature d'un composant est l'union des signatures de toutes les interfaces qu'il définit. De même, la signature d'une interface, c'est l'union des signatures des opérations qu'elle déclare.

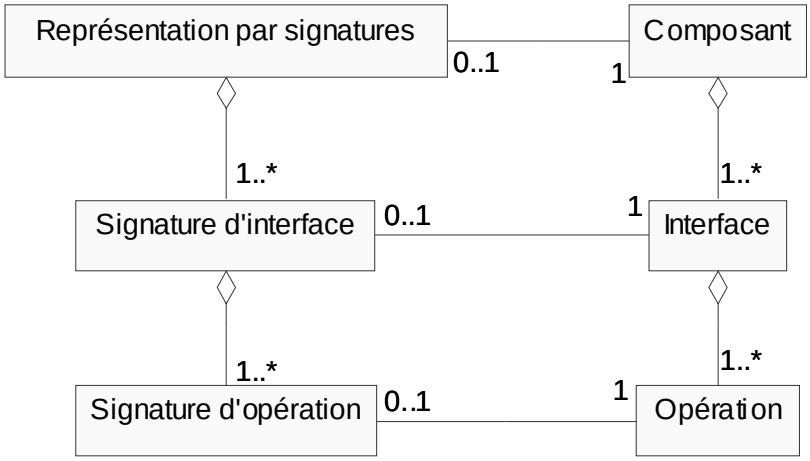


Figure 25. *Modèle de représentation par signatures.*

Dans (Ritti, 1989), Ritti propose une technique d'appariement de signatures et applique sa méthode sur une collection de composants écrits dans un langage fonctionnel. L'algorithme d'appariement (mesure de la distance) que Ritti adopte utilise un système de type polymorphique et garantit une invariance vis-à-vis de l'ordre des déclarations dans un type. Dans (Ritti, 1992), Ritti améliore, le rappel de sa technique en relaxant la condition d'appariement sur l'isomorphisme de types.

> Soit deux graphes, G = (V1, E1) et H = (V2, E2) tels que G et H représentent les structures des types. G est isomorphe à H s'il existe un sous-ensemble E ⊆ E1 et un sous-ensemble V ⊆ V1 tel que |V| = |V2| et |E| = |E2| et il existe une fonction de correspondance bijective (un à un) f : V2 → V satisfaisant {u,v} ∈ E2 si et seulement si {f(u), f(v)} ∈ E.

Runciman et Toyn (Runciman, 1989) proposent une approche similaire à celle de Ritti en utilisant des types polymorphiques pour définir des critères d'appariement de signatures dans le contexte de la programmation fonctionnelle. L'apport de l'approche de Runciman et Toyn consiste dans l'indépendance du critère d'appariement vis-à-vis du nombre d'arguments des méthodes, ce qui améliore le critère de rappel.

Gaudel et Moineau (Gaudel, 1991) introduisent la théorie de la réutilisabilité logicielle (Theory of software reusability) sur la base de la spécification algébrique (Ehrig, 1985) (Wirsing, 1991) des composants logiciels. Les composants sont spécifiés avec le langage PLUSS[4] (Bidoit, 1989). Une spécification décrit la signature d'un composant, plus un ensemble d'axiomes décrivant les interactions entre les méthodes. Gaudel et Moineau définissent la relation d'ordre « *réutilisabilité* » entre les spécifications. Le lien de réutilisabilité entre deux spécifications indique que moyennant un enrichissement, la spécification source peut devenir équivalente à la spécification cible. En variant la définition

[4] **P**roposition of a **L**anguage **U**seable for **S**tructured **S**pecifications.

du critère d'équivalence et la façon dont une spécification peut être enrichie, Gaudel et Moineau ont défini un large éventail de critères d'appariement. Ces critères tiennent compte du renommage de méthodes et de la relaxation du critère d'équivalence entre spécifications. Moineau et Gaudel valident leurs travaux en présentant la base de composants *ReuSig* (Moineau, 1991) qui exploite exclusivement une technique d'appariement de signatures. Badaro et Moineau proposent également une version spécialisée de la base de composants *ReuSig* baptisée ROSE-Ada (Badaro , 1991) qui gère des collections de composants Ada.

Zaremski et Wing (Zarimski, 1993) (Zarimski, 1995) proposent une technique de classification et de recherche de composants basée sur les signatures. Le processus de recherche consiste à parcourir la base de composants pour sélectionner les composants dont les signatures sont compatibles avec celles de la requête. Deux signatures sont considérées compatibles si elles sont identiques modulo un renommage et une permutation des noms et des paramètres des méthodes. Une relaxation des critères d'appariement (équivalence) permet d'améliorer le rappel de cette approche. Les auteurs proposent deux types de relaxations : par changement du critère d'appariement ou par transformation des signatures de la requête et des composants.

- Le critère d'appariement par équivalence précédemment présenté peut être remplacé par d'autres critères comme l'appariement par généralisation et l'appariement par spécialisation. Le critère d'appariement par généralisation (respectivement par spécialisation) sélectionne les composants dont les signatures sont identiques ou généralisent (respectivement spécialisent) la signature de la requête.
- Les processus de recherche des composant peuvent transformer la spécification de la requête et celles des composants pour arriver à les apparier. Les auteurs proposent deux types de transformation : enrichissement et appauvrissement. L'enrichissement (respectivement appauvrissement) consiste en l'ajout (respectivement la suppression) de méthodes ou de paramètres. Grâce à ces transformations, le système peut retrouver des composants qui n'ont pas exactement le même nombre de paramètres ou des méthodes qu'exige la requête, et qui ont un certain degré de ressemblance.

Ces deux types de relaxation peuvent être combinés pour produire d'autres critères d'appariement.

3.3.2 Techniques d'appariement de spécifications

Nous présentons dans cette section une sélection des travaux de recherche sur les techniques d'appariement de spécifications. Les techniques de recherche de composants par appariement de spécifications tentent de retrouver les composants de la base dont les spécifications correspondent (sont compatibles avec) à la spécification de la requête. Ces techniques peuvent être classées selon les types de composants auxquels elles sont applicables : les composants logiciels et les composants conceptuels.

3.3.2.1 Appariement de spécifications pour les composants logiciels

Les techniques de recherche de composants par appariement de spécifications représentent généralement les composants logiciels selon le modèle de la figure 26.

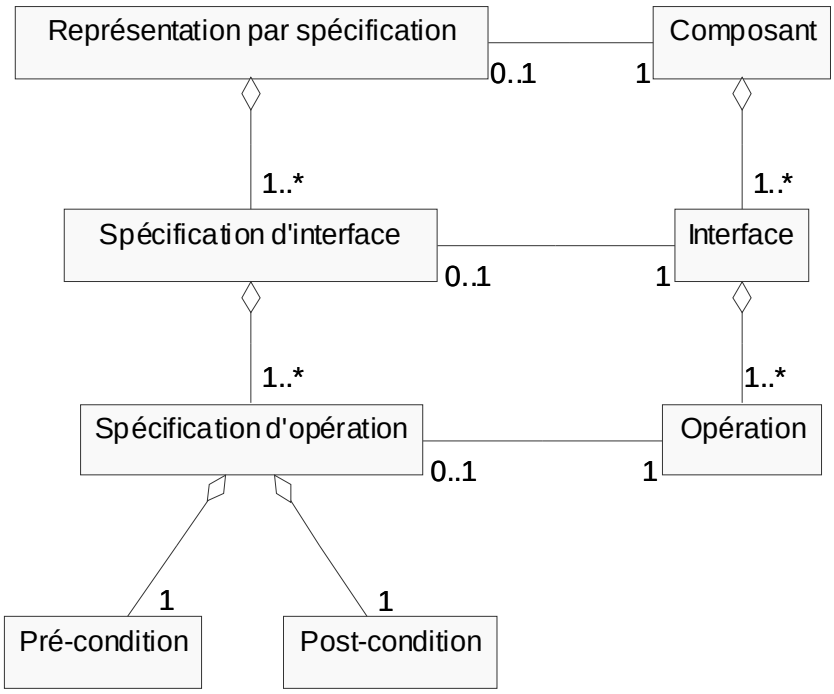


Figure 26. *Modèle de représentation par spécifications (Zaremski, 1995(a)).*

Zaremski et Wing ont étendu leurs travaux sur les signatures pour proposer une approche par appariement de spécifications de composants (Zaremski, 1995(a)). Ils représentent les requêtes et les composants par un ensemble de paires de pré et post conditions (une paire par méthode). Ils définissent aussi un critère général d'appariement défini par la formule

$$Match(S,Q) = (Q_{\text{Pr}\acute{e}} R_1 S_{\text{Pr}\acute{e}}) R_2 (S_{Post} R_3 Q_{Post})$$

où $S = (S_{\text{Pr}\acute{e}}, S_{Post})$ est la spécification d'un composant,

$Q = (Q_{\text{Pr}\acute{e}}, Q_{Post})$ est la spécification d'une requête,

R_1, R_2, R_3 sont trois relations logiquement connectives.

En variant les relations R_1, R_2, R_3, Zaremski et Wing obtiennent une hiérarchie de critères d'appariement.

Rollins et Wing (Rollins, 1991) présentent une base de composants utilisant une approche de classification par spécifications. Les composants sont spécifiés en langage λ-Prolog. Cette technique exploite les mécanismes d'inférence du langage λ-Prolog pour l'appariement des spécifications. Seuls les composants dont les spécifications sont pertinentes par rapport à la spécification de la requête sont sélectionnés.

Le système CAPES (Computer Aided Prototyping Embeded System) (Steigerwald, 1992) est un environnement de prototypage de systèmes temps réel embarqués. L'environnement CAPES comprend : un système de support d'exécution, un éditeur dirigé par la syntaxe, une base de composants et un environnement de conception. Les requêtes utilisateurs et les composants de la base de composants sont spécifiés en utilisant un langage de spécification augmenté avec OBJ. Le processus de recherche de composants utilise la technique d'interrogation par consistance. En exploitant les capacités d'inférence offertes par le langage

OBJ, le système sélectionne uniquement les composants de la base consistants par rapport à la spécification de la requête.

Hemer et Lindsay proposent une base de modules (Hemer, 2001). Un module est un ensemble d'unités. Une unité peut être une procédure, une classe ou une structure de données, Si un développeur a besoin d'un module qui implante toutes les opérations sur les listes chaînées alors il doit spécifier toutes les primitives dont il a besoin sous la forme d'unités, puis il cherche dans la base, le module qui contient ces unités. Hemer et Lindsay proposent trois stratégies de recherche de modules : ALL-match, SOME-match et ONE-match. Le critère ALL-match vérifie que toutes les unités de la requête correspondent à des unités distinctes du composant. Le critère SOME-match relaxe le critère ALL-match en se contentant de vérifier qu'au moins un sous-ensemble non vide d'unités de la requête correspond à un sous-ensemble non vide d'unités du composant. Le critère ONE-match vérifie qu'au moins une des unités de la requête correspond à une des unités du composant.

3.3.2.1.1 Approches par présélection

La complexité des algorithmes d'appariement est parfois un handicap qui cause des problèmes de performance lors de la montée en charge. Pour atténuer ce problème, des travaux ont adopté des techniques de présélection dont la complexité est faible ce qui améliore les performances générales du système. En contre partie il y a toujours un risque que la présélection filtre des composants pertinents par rapport à la requête et qui ne seront pas candidats pour l'appariement. L'efficacité des techniques de présélection peut affecter le rappel d'une base de composants.

Cheng et Jeng (Cheng, 1992) adoptent dans leur approche une organisation en deux couches. Ils utilisent un algorithme de groupement (*clustering*) qui rassemble les composants similaires dans des groupes (*clusters*). L'évaluation de la requête utilisateur s'effectue sur deux étapes : présélection qui élimine les groupes non conformes à la requête utilisateur, puis recherche plus détaillée effectuée sur les composants des groupes (cluster) présélectionnés. Après des études et des travaux d'investigation sur les critères d'appariement (Jeng, 1994), Cheng et Jeng proposent des critères d'appariement inter-composants et inter-méthodes invariants par rapport au changement de nom et à la permutation des paramètres des méthodes (Jeng, 1995).

Fisher, Kievernagel et Snelting (Fisher, 1995) proposent une approche mixte pour remédier au problème de montée en charge des techniques de recherche de composants par appariement de spécifications. L'originalité de cette approche réside dans la technique de présélection des composants candidats à l'appariement de spécifications avec la requête utilisateur. La phase de présélection est composée de deux filtres : le *filtrage par signature* et le *filtrage par rejet*. Le filtrage par signature élimine les composants dont les signatures sont incompatibles avec la signature de la requête utilisateur. Le filtrage par rejet utilise des techniques de simplification (résumé) de spécifications. Après simplification, le système tente de prouver que les deux spécifications simplifiées sont incompatibles. Ces deux techniques de filtrage permettent de réduire le nombre de composants candidats au fur et à mesure que la complexité des algorithmes de filtrage augmente. La phase d'appariement de spécifications (appelée aussi filtrage par confirmation) tente de prouver que les composants présélectionnés sont compatibles avec la requête grâce à un prouveur de théorèmes. L'expérimentation de cette approche s'est faite sur une collection de composants spécifiés avec le langage VDM. Elle a donné une bonne précision, mais un faible rappel dû principalement à la sévérité des critères de filtrage par signature.

3.3.2.1.2 Approches par spécification relationnelle

Boudriga, Mili et Mittermeir (Boudriga, 1992) discutent une technique de spécification relationnelle des composants et des requêtes. Ils proposent également la relation d'ordre

« *raffinement* » sur les composants. Le critère d'appariement est la conformité de la spécification d'un composant à la spécification d'une requête. Ces travaux ont abouti à un prototype expérimental (Mili, 1994) qui supporte deux types d'opération de recherche de composants : recherche exacte et recherche approximative. La recherche exacte utilise le démonstrateur de théorème *Otter*[5] pour sélectionner les composants de la base dont les spécifications sont équivalentes à la celle de la requête utilisateur. La technique approximative sélectionne les composants dont les spécifications approximent celle de la requête utilisateur. Dans (Labed Jilani, 1997), les auteurs présentent une extension de ces travaux et proposent des mesures de distances fonctionnelles entre les spécifications relationnelles. L'évaluation de la requête consiste alors à minimiser la distance entre la spécification relationnelle de la requête et celles des composants de la base de composants.

3.3.2.1.3 Approches par métaconnaissances (spécification de domaine)

Penix et Alexander (Penix, 1995) proposent une approche de recherche de composants à base d'appariement de spécifications de composants et orientée théorie de domaine[6]. Ils argumentent leur choix de procéder à la modélisation du domaine de travail (de la base de composants) par le fait que les approches à base d'appariement de spécifications utilisent des prouveurs de théorèmes ce qui les rend imprévisible vis-à-vis de la montée en charge. La définition d'un modèle de domaine permet d'avoir une définition des concepts de base du domaine et d'un savoir concernant ces concepts. Cette base de connaissances permet d'exprimer les spécifications des composants et des requêtes en fonction de ces connaissances de base. Le modèle de domaine permet donc de garantir un certain niveau d'abstraction des spécifications et facilite la tâche des prouveurs de théorèmes. Penix et Alexander proposent comme une extension de leurs travaux une variété de critères d'appariement et un raffinement des concepts de leur modèle de domaine. Comme Mili et al (Boudriga, 1992) (Mili, 1994) (Labed Jilani, 1997), Penix et Alexander font la distinction entre la recherche exacte et la recherche approximative en considérant la première comme étant un « heureux hasard » de la deuxième. Comme Zaremski et Wing (Zaremski, 1993), (Zaremski, 1995), (Zaremski, 1995(a)), Penix et Alexander définissent une hiérarchie de critères d'appariement et spécifient les composants avec le langage de spécification formelle Larch[7].

3.3.2.2 Appariement de spécifications pour les composants conceptuels

La base de composants proposée par Massonet et Van Lamsweerde (Massonet, 1997) gère une collection de frameworks de spécification des besoins (composants conceptuels). Elle est organisée sous la forme d'une collection structurée de frameworks qui ont déjà servi à la construction d'autres systèmes d'information. Un framework est une collection de concepts besoins (buts, contraintes, agents, entités, relations, événements, actions et scénarios). Les frameworks sont ordonnés selon la relation *IsA*, ce qui confère à la base de composants une structure hiérarchique.

Le problème de recherche de composants dans cette base se traduit par une requête qui donne une spécification incomplète d'un besoin utilisateur. Le système identifie les parties manquantes de la spécification et tente de la compléter avec un composant de la base. Le processus de recherche consiste donc en un processus de raisonnement par analogie qui part

[5] Otter, © Argonne National Laboratory. http://www-unix.mcs.anl.gov/AR/otter/

[6] Une théorie de domaine est un modèle algébrique d'un domaine de problèmes.

[7] http://www.sds.lcs.mit.edu/spd/larch/

de la spécification des parties manquantes d'une requête pour aboutir à des spécifications de frameworks de la base. La relation d'analogie est définie par un certain nombre de critères :

- analogie un-à-un des concepts : un concept source (respectivement cible) peut être analogue à un seul concept cible (respectivement source).
- métatype identique : si deux concepts sont analogues, ils ont le même métatype (exemple agent, contrainte, etc.).
- même multiplicité des relations et prédicats : deux relations (respectivement prédicats) sources et cibles sont analogues, ils ont le même nombre de liens (respectivement arguments).
- préservation de rôle : deux relations (respectivement prédicats) source et cible sont analogues si les paires d'objets (respectivement arguments) de la source et de la cible sont analogues dans le même ordre d'apparition. Par exemple, le prédicat $P_1(a,b)$ et $P_2(c,d)$ sont analogues si a (respectivement b) *et* c (respectivement d) sont analogues.
- catégories de métatypes identiques : en plus de l'obligation d'avoir deux métatypes identiques, ils doivent aussi appartenir à la même catégorie. Par exemple, une contrainte de la catégorie *contrainte faible* et une contrainte de la catégorie *contrainte forte* ne peuvent pas être considérées analogues car elles appartiennent à deux catégories différentes de contraintes.

La recherche d'un framework de spécification de besoins pertinent par rapport à une requête se fait selon trois étapes pour les mêmes raisons que dans (Fisher, 1995) : appariement structurel entre les déclarations, appariement sémantique en vérifiant la compatibilité des contraintes et des assertions, enfin, vérification de tous les critères d'analogie.

3.4 Techniques de recherche comportementale

3.4.1 Approches par analyse des traces d'exécutions

Les techniques de recherche comportementale s'intéressent à l'aspect dynamique des composants en analysant leur comportement lors de l'exécution. Podgursky et Pierce (Podgursky, 1992), (Podgursky, 1993) ont déduit après une observation statistique qu'un composant appartenant à une base de composants peut être identifié en se basant uniquement sur son comportement avec des paramètres d'entrée aléatoires. En se basant sur cette observation, ils ont construit une base de composants qui a les propriétés suivantes (cf. figure 27) :

- un composant est représenté par son code exécutable et la description de ses paramètres d'entrée. Un paramètre d'entrée est défini par son type ainsi qu'une distribution probabiliste des valeurs qu'il peut prendre par rapport à son domaine de définition. La distribution probabiliste permet d'estimer la pertinence du choix d'une valeur par rapport à son occurrence lors de l'utilisation du composant.
- la requête se compose de deux parties : l'espace des valeurs d'entrée désirées et une condition qui détermine si la réponse d'un composant est satisfaisante par rapport aux besoins de l'utilisateur.
- l'opération de comparaison de la requête et d'un composant de la base se fait en trois étapes : sélection aléatoire d'un certain nombre de valeurs d'entrée en se basant sur les paramètres de la requête et la description du composant dans la base, application des valeurs d'entrée sur tous les composants de la base, sélection des seuls composants qui vérifient les paramètres de la requête.

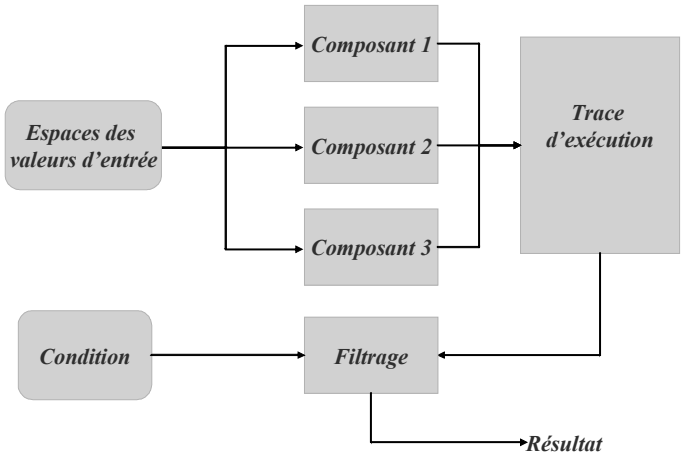


Figure 27. *Scénario d'exécution d'une requête dans l'approche de Podgursky et Pierce*

Podgursky et Pierce constatent que la probabilité qu'un composant de la base soit non pertinent par rapport à une requête bien qu'il satisfasse sa condition après son exécution avec *n* valeurs d'entrées aléatoires, décroît exponentiellement en fonction de *n*. On peut donc atteindre une très grande précision avec des valeurs petites de *n*. Malgré une bonne précision, la méthode de Podgursky et Pierce fournit un rappel très faible. Cette carence est due par exemple au fait que l'approche de Podgursky et Pierce ne prend pas en compte la notion de paramètres optionnels présente dans certains langages de programmation comme C et ADA. De plus, elle ne tient pas compte de la notion de sous-typage ou de la permutation des paramètres des fonctions.

Dans (Hall, 1993), Hall propose une approche de recherche comportementale généralisée des composants. Il critique certains aspects de l'approche de Podgursky et Pierce et propose des améliorations parmi lesquelles :

- la possibilité de définir des paramètres optionnels pour les fonctions, ce qui a tendance à améliorer le rappel ;
- la possibilité que l'utilisateur choisisse lui même les données d'entrée par rapport auxquelles les composants sont testés. Hall affirme que le fait de choisir aléatoirement les paramètres d'entrée n'améliore pas forcément la précision et le rappel, mais au contraire affecte fortement les performances du système ;
- la relaxation de la fonction de mesure de similarité entre les résultats d'exécution et les conditions de sélection précisées dans la requête utilisateur afin d'améliorer le rappel. La sémantique de la fonction de correspondance est modifiée. Au lieu de sélectionner uniquement les composants qui satisfont les besoins de l'utilisateur, la base renvoie aussi les composants qui pourraient être réutilisés par l'utilisateur après une légère adaptation.

Park et Bais (Park, 1997) proposent une autre approche pour la recherche comportementale qui représente une continuité des travaux de (Podgursky, 1992) et (Hall, 1993). Au lieu de générer statistiquement les valeurs d'entrée (Podgursky, 1992) ou de demander à l'utilisateur de les spécifier (Hall, 1993), Park et Bais utilisent une méthode inductive. L'idée consiste à faire une induction sur les structures d'entrée en utilisant une base de cas. Les entrées obtenues sont les cas sélectionnés dans la base de cas et les cas induits (générés).

3.4.2 Approches par spécification comportementale

Chou, Chen et Chung (Chou, 1996) présentent une base de composants orientés objet utilisant une technique comportementale de sélection de composants. Un composant est représenté dans la base non plus par son code exécutable, mais par des réseaux sémantiques décrivant son comportement. La requête est comparée aux composants pour sélectionner ceux qui maximisent la fonction de similarité qui est égale au rapport de la cardinalité des comportements communs sur la cardinalité totale des comportements de la requête (coefficient de Jaccard's (Rijsbergen, 1979)). L'avantage de cette technique est qu'elle permet non seulement de retrouver les composants simples (une classe), mais également les composants complexes (un graphe de classes). De plus, la technique est outillée par un éditeur de spécifications, un outil de classification de spécifications, un gestionnaire de bases de composants et un outil de recherche de spécifications.

La structure de stockage des composants n'est pas précisée dans les approches que nous avons présentées dans cette section. En effet, elle n'affecte en rien la précision ou le rappel des méthodes, mais elle peut par contre fortement affecter leurs performances.

3.5 Techniques de recherche par navigation

Les techniques de recherche de composants par navigation exploitent les relations implicites ou explicites qui peuvent exister entre les composants appartenant à une base de composants. Par exemple, certains environnements de développement adoptent les techniques de recherche de composants par navigation comme le browser de Smalltalk-80 (Goldberg, 1984). Nous présentons dans la suite de cette section deux exemples de sous-catégories des approches de recherche de composants par navigation : approche hypertexte et approche par clustering.

3.5.1 Approche hypertexte

L'approche hypertexte (Garg, 1990) (Cybulski, 1993) (Latroue, 1988) organise les informations en un graphe de nœuds appelés unités d'informations. Ces nœuds sont interconnectés avec plusieurs types de liens appelés relations. L'utilisateur navigue à l'intérieur du graphe en utilisant les relations. La sémantique associée à ces relations guide l'utilisateur selon ses besoins pour sélectionner le bon chemin qu'il doit suivre à travers le graphe.

Le problème majeur de l'approche hypertexte est que le processus de développement et de maintenance de ce genre de bases de composants demande un effort humain considérable. En effet, il n'existe pas une méthode claire et efficace permettant d'extraire et d'identifier les relations qui offrent une richesse sémantique suffisante pour aider au mieux l'utilisateur dans ses choix de navigation. De plus, l'ajout d'un nouveau composant au système peut remettre en cause la consistance des liens déjà existants. Pour remédier à ce problème, des travaux proposent de générer semi-automatiquement des liens entre les composants. En effet, le mécanisme d'extraction des relations varie suivant le modèle de représentation de composants utilisé par la base de composants.

La base de composants SEL[8] (Freitag, 1994) représente la collection de composants par un graphe hypertexte de documents reliés par des hyperliens. Les documents décrivent les composants. La recherche se fait à travers un système de requêtes composées sous la forme d'expressions booléennes de termes. SEL utilise un thésaurus pour étendre les requêtes par synonymie. Les documents retrouvés sont classés selon une mesure de pertinence. La

[8] Software Engineering Library

pertinence d'un composant par rapport à une requête est fonction de la fréquence d'apparition des termes de la requête dans le document et du nombre de documents appartenant à la base de composants. Une fois la requête évaluée, l'utilisateur part des documents retournés par SEL pour naviguer dans la base.

La technologie de l'hypertexte est utilisée dans les travaux de Isakowitz et Kaufmann (Isakowitz, 1996) pour construire la base de composants ORCA. Elle exploite une technique de classification des composants par facettes, implantée sous la forme d'un système hypertexte.

3.5.2 Approche par clustering

Jeng et Cheng (Jeng, 1993) construisent un graphe hiérarchique de spécifications permettant la navigation pour la recherche de composants. Le graphe est construit en utilisant un algorithme de clustering et une relation de subsomption. Dans un premier temps, la relation de subsomption est utilisée pour construire des graphes hiérarchiques de spécifications. Puis, les graphes sont reliés grâce à un algorithme de clustering sur les sommets des graphes hiérarchiques (cf. figure 28). Ainsi lors de la navigation, l'utilisateur commence par choisir le cluster qui l'intéresse, puis continue sa navigation dans la base de spécifications en suivant les liens de subsomption.

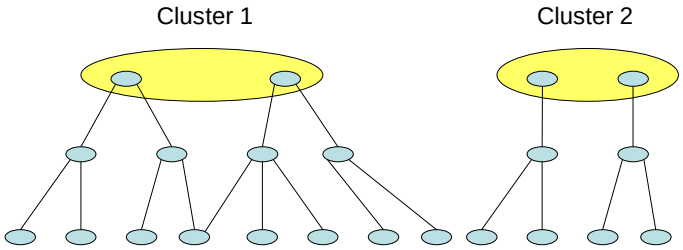


Figure 28. *Clusters de graphes hiérarchiques de spécifications de composants.*

Les approches de recherche de composants par clustering et hypertexte demandent à l'utilisateur de piloter le processus de navigation, ce qui risque de devenir une tâche difficile si la taille de la base de composants est grande et selon le niveau d'expertise de l'utilisateur. De plus, le résultat final du processus de recherche dépend fortement du nœud de départ dans le graphe de recherche. Pour cela, les approches de recherche de composants par navigation sont souvent utilisées comme un mécanisme complémentaire de raffinement de la recherche des systèmes utilisant les approches linéaires à base de requêtes. D'un autre côté, les approches par navigation demandent un effort important pour la construction des graphes qui sont utilisés lors de la navigation. Les algorithmes de construction des graphes de navigation sont généralement semi-automatiques.

4. Comparaison entre les techniques de recherche de composants

4.1 Critères d'évaluation

Des critères d'évaluation permettent de souligner les caractéristiques de chaque approche et constituent par la suite une bonne base pour comparer les différentes techniques de recherche de composants. Ils sont divisés en quatre catégories : critères techniques, critères économiques, critères humains et caractéristiques de conception.

4.1.1 Critères techniques

Nous identifions cinq critères techniques détaillés ci-dessous.

- **Précision**

La précision est définie par le rapport entre le nombre de composants pertinents retournés et le nombre total des composants retournés. La précision prend donc une valeur entre 0 et 1. Nous rapportons la mesure de la précision à une échelle discrète de cinq valeurs (Très Basse (--), Basse (-), Moyenne (=), Haute (+), Très Haute (++)).

- **Rappel**

Le rappel est défini par le rapport entre le nombre de composants pertinents retrouvés et le nombre total des composants pertinents d'une base de composants. Le rappel prend donc une valeur entre 0 et 1. Nous rapportons la mesure du rappel à une échelle discrète de cinq valeurs (Très Bas (--), Bas (-), Moyen (=), Haut (+), Très Haut (++)).

- **Couverture**

La couverture est définie par le rapport entre le nombre moyen de composants visités lors de l'évaluation d'une requête et le nombre total de composants d'une base de composants. La couverture prend donc une valeur entre 0 et 1. Nous rapportons la mesure de la couverture à une échelle discrète de cinq valeurs (Très Basse (++), Basse (+), Moyenne (=), Haute (-), Très Haute (--)).

- **Complexité d'appariement**

La complexité d'appariement mesure le nombre d'étapes nécessaires pour apparier une requête à un composant de la base de composants. La complexité est fonction de la taille d'une requête. Pour simplifier, nous définissons une échelle de comparaison à cinq valeurs.

Très Basse (++)	Basse (+)	Moyenne (=)	Haute (-)	Très Haute (--)
Constante	Linéaire	Polynomiale	Exponentielle	Non majorée

- **Potentiel d'automatisation**

Un critère important du succès d'une technique de recherche d'information est son potentiel d'automatisation. Plus une technique est automatisable, plus sa mise en place est facile. Le potentiel d'automatisation est un critère très subjectif que nous estimons sur une échelle de cinq valeurs.

Très Basse (--)	Basse (-)	Moyenne (=)	Haute (+)	Très Haute (++)
Non automatisable	Demande un effort important	Demande un effort non négligeable	Automatisable avec peu d'effort	Automatisation simple

Les critères de complexité et de couverture peuvent servir à estimer la performance en terme de temps d'exécution moyen d'une approche de recherche d'information.

4.1.2 Critères économiques

Nous identifions quatre critères économiques détaillés ci-dessous.

- **Coût d'investissement**

Le coût d'investissement reflète le coût de mise en œuvre d'un système de recherche de composants utilisant la technique de recherche d'information étudiée. L'unité de mesure de ce critère économique est normalement le mois-personne (temps de travail d'une personne pendant une durée d'un mois), mais dans cet état de l'art, nous utilisons une échelle discrète et subjective de cinq valeurs allant de très bas (++) jusqu'à très haut (--) car il n'est pas toujours possible de donner une valeur exacte. De plus, la valeur est rarement précisée par les auteurs de l'approche.

- **Coût de fonctionnement**

Le coût de fonctionnement reflète le coût de gestion et de maintenance de la base de composants. Pour les mêmes raisons que pour le coût d'investissement, nous utilisons une échelle discrète et subjective de cinq valeurs allant de très bas (++) jusqu'à très haut (--).

- **Degré de diffusion**

Le degré de diffusion estime la popularité d'une base de composants dans les laboratoires de recherche et l'industrie de développement de systèmes d'information. Ce critère prend une valeur selon une échelle discrète et subjective de cinq valeurs allant de très bas (--) jusqu'à très haut (++).

- **État de développement**

L'état de développement reflète le degré d'avancement d'une technique de recherche d'information.

Très Basse (--)	Basse (-)	Moyenne (=)	Haute (+)	Très Haute (++)
Spéculation	Prototype de laboratoire	Produit expérimental	Produit industriel	Produit industriel supporté par l'industrie : recherche, développement, formations, etc.

4.1.3 Critères humains

Nous identifions deux critères humains détaillés ci-dessous.

- **Difficulté d'utilisation**

La difficulté d'utilisation est un critère prédominant pour la réussite de l'intégration d'une base de composants dans un processus de développement de systèmes d'information. En effet, certaines techniques de recherche de composants exigent des connaissances avancées dans certains domaines comme la spécification, la modélisation, etc., ce qui ne facilite pas forcément leur utilisation par les simples utilisateurs. Nous utilisons une échelle discrète de cinq valeurs pour quantifier le critère de difficulté d'utilisation.

Très Basse (++)	Basse (+)	Moyenne (=)	Haute (-)	Très Haute (--)
Triviale	Facile	Non triviale	Difficile	Très difficile

- **Transparence**

Ce critère reflète dans quelle mesure la tâche de recherche de composants dans la base de composants est dépendante de la compréhension du mécanisme de recherche qu'elle utilise. Nous utilisons une échelle discrète de cinq valeurs pour quantifier ce critère.

Très Basse (--)	Basse (-)	Moyenne (=)	Haute (+)	Très Haute (++)
Connaissance détaillée du mécanisme de recherche	Bonne connaissance	Bonne idée	Légère idée	Transparence parfaite

4.1.4 Caractéristiques de conception

Nous identifions 12 critères permettant de classer les bases de composants selon leurs caractéristiques de conception.

- **Nature des composants**

La nature des composants gérés dans une base de composants influe fortement sur le type des techniques de recherche de composants que la base de composants peut utiliser. Les artefacts gérés dans une base de composants sont généralement des codes sources ou exécutables, mais on trouve également des spécifications, des modèles de conception, de la documentation, etc.

- **Couverture**

Une base de composants peut contenir des composants spécifiques à un domaine d'application ou des composants venant de plusieurs domaines. La généricité d'une base de composants peut influer fortement sur les performances de certaines techniques de recherche de composants. Par exemple, les techniques textuelles de recherche peuvent être fortement dépendantes du vocabulaire utilisé. Par exemple, si un terme a différentes significations dans plusieurs domaines d'applications, alors il peut fortement perturber le rappel et la précision d'une technique de recherche de composants qui représente les composants par des mots-clés.

- **Étendue**

L'étendue d'une base de composants peut être plus ou moins importante selon qu'elle est utilisée au sein d'une équipe de développement, de plusieurs équipes de développement ou par des milliers d'utilisateurs à travers le site Internet d'un fournisseur de composants.

- **Portée**

La portée d'une base de composants est évaluée en fonction de l'étape d'ingénierie à laquelle les composants qu'elle contient s'adressent (spécification, analyse, conception, implantation).

- **Ouverture**

Le niveau de transparence des composants d'une base de composants influe fortement sur le choix des techniques de recherche de composants qui peuvent être utilisées. Nous identifions trois niveaux de transparence : boîte noire (la structure interne du composant n'est ni visible, ni modifiable, seule l'interface est accessible), boîte blanche (le composant est complètement transparent) et boîte en verre (la structure interne du composant est visible mais non modifiable).

- **Représentation de la requête**

Une base de composants peut être caractérisée par la forme des requêtes auxquelles elle peut répondre : ensemble de termes, langage naturel, spécification, etc.

- **Représentation d'un composant**

La représentation des composants est un aspect important de la conception d'une base de composants. Elle influe sur la forme des requêtes auxquelles une base de composants peut répondre et sur la façon de les évaluer. Elle influence également la structure de stockage

utilisée par la base de composants. Dans une base de composants parfaitement transparente vis-à-vis de l'utilisateur, la représentation des composants à l'intérieur de la base est sans importance. Comme pour les requêtes, un composant peut être représenté par un ensemble de termes, du langage naturel, une spécification, etc.

- **Structure de stockage**

La structure de stockage d'une base de composants décrit l'organisation des représentations des composants dans la base. Souvent, il n'existe aucune organisation des représentations de composants : lors de la recherche, le système parcourt toutes les représentations et sélectionne les composants pertinents notamment dans la représentation des composants par des mots-clés. Il existe des exceptions à cette règle : par exemple, certaines approches de recherche de composants qui représentent des composants par une spécification arrivent à définir des relations de raffinement (relations d'ordre entre les spécifications). Cela permet d'avoir une structure de stockage hiérarchique ou sous la forme d'un graphe.

- **Schéma de navigation**

Les schémas de navigation et les structures de stockage sont fortement corrélés. En effet, le type de navigation possible dans une base de composants dépend fortement de la structure de stockage. Par exemple dans une organisation à plat (structure séquentielle), la seule navigation possible est la navigation séquentielle alors que dans le cas où les composants sont organisés selon une structure non triviale (qui exploite les liens sémantiques qui peuvent exister entre les composants), il est possible de faire des choix de navigation selon les besoins de l'utilisateur. Le schéma de navigation influe donc fortement sur les critères d'évaluation des performances comme le rappel et la couverture.

- **Objectif de la recherche**

L'objectif de la recherche dans une base de composants peut varier selon les besoins de l'ingénieur et le contexte de réutilisation. L'algorithme qui estime la pertinence d'un composant par rapport à une requête utilisateur peut donner deux types de résultats : une pertinence booléenne et une pertinence réelle entre zéro et un.

Dans le cas où la pertinence est une fonction booléenne, la base de composants retourne une liste de composants correspondant parfaitement à la requête. L'utilisateur continue alors à affiner sa recherche en raffinant les critères de la requête et réitère ce processus jusqu'à trouver le composant qu'il cherche. Dans certains cas, il n'existe aucun composant correspondant exactement aux attentes de l'ingénieur ; le processus d'affinement de la recherche s'arrête alors après une réponse nulle du système de recherche.

Dans le cas où la pertinence est une fonction réelle qui renvoie une valeur entre zéro et un, la sémantique de la pertinence désigne la mesure d'une distance entre les besoins de l'utilisateur et un composant de la base. Le processus d'interaction entre la base de composants et l'utilisateur est le même qu'avec le cas de la pertinence booléenne, sauf que l'utilisateur ne sera jamais confronté à une réponse nulle du système de recherche.

Dans le cas des techniques de recherches d'information appliquées aux descriptions des composants en langage naturel, le but de la recherche est de trouver la description du composant qui répond au problème posé par la requête. Le système peut uniquement détecter qu'une description risque de correspondre à une requête, mais ne peut affirmer qu'elle donne une solution complète.

- **Critère de pertinence**

Le critère de pertinence définit les conditions sous lesquelles un composant est considéré pertinent par rapport à une requête. Le niveau de transparence des composants gérés par une

base de composants influe énormément sur le choix de la fonction de pertinence. Le choix de la fonction de pertinence influe beaucoup à son tour sur la façon d'utiliser une base de composants. Par exemple, dans un contexte de réutilisation de composants « boîte blanche », l'ingénieur cherche en premier lieu les composants qui correspondent à ses besoins (avec une mesure de pertinence égale à 1). Sinon, il cherche les composants qui nécessitent le plus petit effort d'adaptation possible pour répondre à ses besoins (modification générative) ou bien il cherche un ensemble de composants qu'il pourra assembler pour créer un composant qui corresponde à ses besoins (modification compositionnelle). Dans le cas où la base de composants gère des composants du type boîte noire, l'approche générative n'est pas utilisable alors que l'approche compositionnelle et l'approche de pertinence booléenne sont envisageables.

- **Critère de présélection**

Le critère de présélection définit les conditions sous lesquelles un composant est considéré candidat pour mesurer sa pertinence par rapport à la requête. Lorsque la complexité de la fonction de pertinence est importante (critère d'appariement), il est intéressant de faire un filtrage des candidats à cette opération, ce qui améliore les performances globales du système de recherche.

4.2 Evaluation des différentes techniques étudiées

Les différents tableaux ci-dessous résument notre point de vue sur les différentes techniques étudiées : techniques classiques de recherche d'information (TRI), techniques de classification externes (TCE), techniques de classification structurelle (TCS), techniques de recherche comportementale (TRC) et techniques de recherche par navigation (TRN). Nous rappelons que nous avons homogénéisé la présentation des mesures des critères selon une échelle discrète à cinq valeurs : très mauvaise (--), mauvaise (-), moyenne (=), bonne (+), très bonne (++).

Tableau 5. *Comparaison par rapport aux critères techniques.*

Critères techniques	**TRI**	**TCE**	**TCS**	**TRC**	**TRN**
Précision	=	=	++	++	
Rappel	=	=	++	=	
Couverture	-	=	-	-	
Complexité d'appariement	++	+	--	=	
Potentiel d'automatisation	++	=	-	++	=

Il faut noter le score médiocre des TRI et TCE ainsi que la complexité des TCS actuelles.

Tableau 6. *Comparaison par rapport aux critères économiques.*

Critères économiques	**TRI**	**TCE**	**TCS**	**TRC**	**TRN**
Coût d'investissement	+	-	--	+	=
Coût de fonctionnement	+	-	--	+	-
Degré de diffusion	=	++	+	-	++
Etat de développement	+	++	=	-	+

Ce tableau met en évidence les résultats très contrastés des techniques selon les critères économiques.

Tableau 7. *Comparaison par rapport aux critères humains.*

Critères humains	**TRI**	**TCE**	**TCS**	**TRC**	**TRN**
Difficulté d'utilisation	=	+	--	=	++
Transparence	=	-	--	+	+

Ce tableau met en évidence les très faibles résultats des TCS selon les critères humains.

Tableau 8. *Comparaison par rapport aux caractéristiques de conception.*

Critères de conception	**TRI**	**TCE**	**TCS**	**TRC**	**TRN**
Nature des composants	Aucune restriction	Aucune restriction	Exécutable, frameworks de spécification de besoins	Exécutable	Aucune restriction
Couverture	Spécifique à un domaine	Spécifique à un domaine	Aucune restriction	Aucune restriction	Aucune restriction
Etendue	Equipe ou organisation	Equipe ou organisation	Aucune restriction	Aucune restriction	Aucune restriction
Portée	Aucune restriction	Aucune restriction	Aucune restriction	Implantation	Aucune restriction
Ouverture	Boîte blanche ou en verre	Aucune restriction	Aucune restriction	Aucune restriction	Aucune restriction
Représentation de la requête	Langage naturel ou code source	Langage naturel / Mots-clés	Spécification formelle	Trace d'exécution	
Représentation d'un composant	Langage naturel ou code source	Langage naturel / Mots-clés	Spécification formelle	Trace d'exécution	
Structure de stockage	Généralement plate	Généralement plate et indexée	Généralement sous forme de graphes de spécifications	Généralement plate	Graphe
Schéma de navigation	Généralement séquentiel	Généralement séquentiel indexé	Parcours de graphes	Généralement séquentiel	Chemin dans un graphe
Objectif de la recherche	Booléen ou approximatif	Booléen ou approximatif	Booléen ou approximatif	Booléen ou approximatif	Approximatif
Critère de pertinence	Informel	Informel	Formel	Formel	Informel

Critère de présélection	Pas de présélection	Pas de présélection	Une relaxation du critère de pertinence	Pas de présélection	Eventuellement une autre technique de recherche de composants

Ce tableau met en évidence que les TCS sont les plus polyvalentes.

Les techniques de recherche de composants structurelles sont les plus performantes du point de vue de la fiabilité des résultats des recherches. En effet, elles ont les meilleures précisions, couverture et rappel. Par contre, elles ont la plus haute complexité d'appariement ce qui risque de nuire aux performances des SRC qui utilisent directement ces techniques. Des techniques de relaxation peuvent diminuer cette complexité par contre elles détériorent la précision et le rappel. Les TRS utilisent des langages de spécification formelle ce qui engendre un potentiel d'automatisation très bas, les rend les moins avantageuses du point de vue critères économiques et humains. Enfin, elles ne sont pas spécifiques à un niveau d'abstraction de composants particulier.

Les techniques de recherche comportementales sont techniquement aussi performantes que les techniques structurelles (cf. tableau 5) et elle ont une complexité d'appariement moins importante ce qui les rend intéressantes. De plus, les critères économiques leur sont favorables.

5. Conclusion

Plusieurs types de systèmes de recherche de composants existent et peuvent être classifiés selon leurs scénarios d'utilisation et selon la famille de techniques de recherche de composants qu'ils exploitent. Des exemples de chaque famille de techniques de recherche de composants ont été présentés. Enfin nous avons défini des critères de comparaison (les critères techniques, les critères économiques, les critères humains et les caractéristiques de conception) pour procéder ensuite à une comparaison entre les différentes catégories de techniques de recherche de composants.

Chacun de ses types de TRC est efficace pour répondre à un type spécifique de requêtes utilisateurs (comportementale, spécification, signature, etc.). Une base de composants qui se trouve au centre d'une équipe de développement de systèmes d'information adoptant une démarche de développement par réutilisation de composants doit être capable de gérer plusieurs types de composants, donc doit fournir plusieurs techniques de recherche de composants. Ainsi, elle peut répondre aux besoins spécifiques de chaque type d'acteur. Dans la partie suivante nous proposons un métamodèle pour la construction d'un métaoutil capable d'instancier et de gérer des bases de composants hétérogènes. Nous proposons également un exemple de modèle de bases de composants.

IV. Une base descriptive de composants

Les ingénieurs d'applications dans les environnements de développement modernes utilisent souvent des composants réutilisables d'une hétérogénéité de plus en plus croissante. Cette hétérogénéité est omniprésente tant au niveau des sources de composants (développement interne, fournisseur internet, etc.), des modèles de composants (Patrons, composants EJB ou CCM, etc.), qu'au niveau d'abstraction où les composants sont utilisés (besoin, analyse, conception, implantation, etc.). Le but de ce chapitre est la construction d'une base descriptive de composants *B-Sigma* qui joue le rôle d'un référentiel fédérateur qui facilite le recensement, la documentation et la capitalisation de différents composants hétérogènes. L'étude des besoins des utilisateurs en ce qui concerne la base *B-Sigma* nous permet de proposer un modèle de conception pour la réalisation d'une telle base.

1. Étude des besoins

1.1 Problématique

La base *B-Sigma* est un système d'information spécialisé dans la gestion des descriptions de composants. Elle a pour ambition d'être au centre du processus de développement des systèmes d'information à base de composants. Les composants réutilisables sont généralement construits et gérés dans des ateliers de génie logiciel comme Rational Rose, Objecteering, ArgoUML ou des outils de développement intégrés comme Eclipse, Visual Studio, etc. Ils peuvent aussi être fournis par des fournisseurs de composants (autres équipes de développement ou sites Internet). La base *B-Sigma* doit donc fédérer les différentes sources de composants accessibles à une équipe de développement et centraliser toutes les informations concernant les composants. Ainsi, elle rend possible une recherche sur les différentes sources et favorise la réutilisation des composants.

Le rôle de la base *B-Sigma* ne se limite pas à la gestion des composants logiciels pendant la phase de codage, mais il s'étend sur tout le cycle de développement d'un système d'information. Par exemple, lors du développement d'un composant réutilisable destiné à la représentation du concept métier *Client*, l'équipe de développement commence d'abord par effectuer une étude des besoins, puis procède à une phase d'analyse suivie d'une phase de conception; enfin le composant est implanté pour aboutir à un composant logiciel. La base *B-Sigma* doit intégrer le nouveau composant métier *Client* avec ses représentations dans les différents niveaux d'abstraction tout en assurant la traçabilité d'un niveau à l'autre. De plus, lors de l'intégration d'un nouveau composant qui réutilise d'autres composants de la base, la base *B-Sigma* doit donner le moyen d'exprimer cette information en définissant des relations entre les composants.

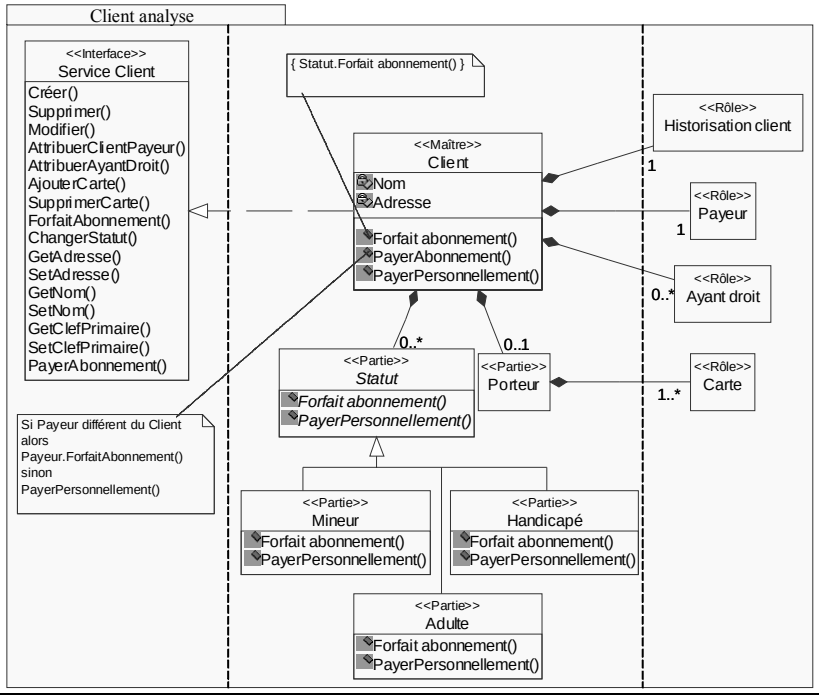
1.2 Exemple : le composant Client

Nous présentons dans cette section la représentation du concept métier *Client* dans le domaine des systèmes d'information de transport dans la base *B-Sigma*. Le concept métier *Client* est représenté par le composant métier *Client*. Ce dernier se décline en un ou plusieurs composants à chaque niveau d'abstraction (analyse, conception et logiciel).

1.2.1 Analyse

En guise d'exemple, le tableau 9 décrit le composant métier *Client* au niveau analyse. La solution conceptuelle est décrite en adoptant le modèle de composants Symphony (Hassine, 2005) (cf. annexe F). La description du composant comporte un ensemble de rubriques typées. Nous pouvons classer les rubriques utilisées pour la description des composants en trois catégories : les rubriques simples comportant une seule valeur (exemple : *Nom*, *Description*), les rubriques composites comportant un ensemble de sous-rubriques (par exemple, la rubrique *Participants* décrit trois classes qui peuvent être assimilées à trois sous-rubriques) et les rubriques relations (exemple : *Utilise*, *Utilisé par*) qui font référence à d'autres composants.

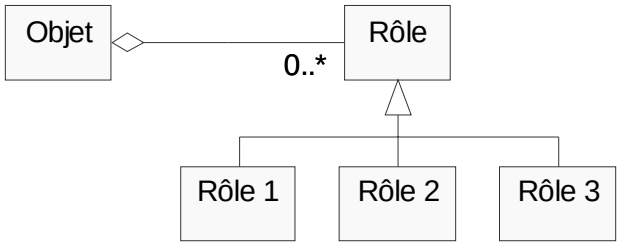
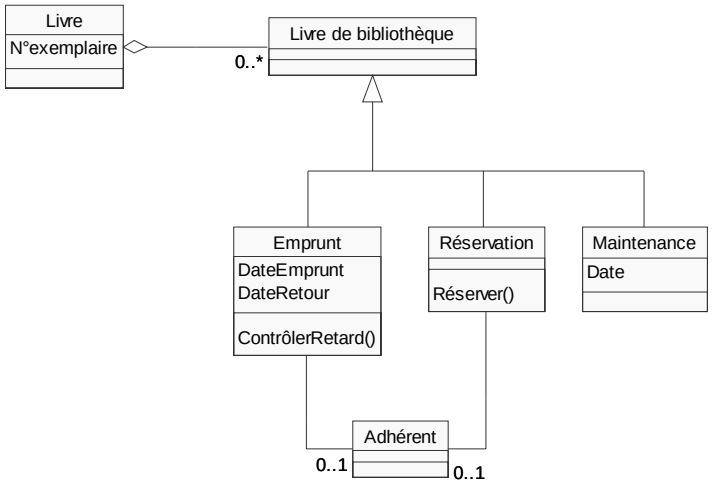
Tableau 9. *Représentation du composant métier Client au niveau analyse avec le modèle de composants Symphony.*

Nom	Client analyse
Auteur	Khayati
Intention	Présente une modélisation du concept *Client* dans le domaine des systèmes d'information de transport (SIT).
Modèle de composants	Symphony
Modèle	

Participants	Le composant *Client* contient les classes suivantes : - Client : la classe maître du composant *Client*. - Statut : une classe partie du composant *Client*. Cette classe gère le statut du client. Un client peut voir l'un des statuts suivant : mineur ou adulte ou handicapé. - Porteur : une classe partie du composant *Client*. Cette classe est instanciée dans le cas où le client est porteur de cartes.
Utilise	Le composant *Client* utilise d'autres composants via les rôles suivants : - Historisation client : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'un *Client*. - Payeur : c'est un rôle du composant *Client*. Le rôle *Payeur* indique le client payeur qui paye pour le client en cours. Si le client paye pour lui même, alors le rôle *payeur* pointe sur le composant *Client* qui le contient. - Ayant droit : c'est un rôle du composant *Client*. Le rôle *Ayant droit* indique le *Client* à travers lequel le client en cours a des droits. Les enfants de moins de 7 ans peuvent par exemple avoir le droit de circuler sur le réseau gratuitement ou avec un tarif réduit si l'un de leurs parents est un client. - Carte : c'est un rôle du composant *Carte*. Le rôle *Carte* indique les cartes que possède un client. Le rôle *Carte* est relié à l'objet maître via l'objet partie *Porteur* pour donner la possibilité au composant Client de réagir différemment suivant la carte qu'il utilise.
Utilisé par	Le composant *Client* est utilisé par les composants suivants : - Client : il joue les rôles de *Payeur* et *Ayant droit*. - Agent : il joue le rôle de *Client*.
Imite	Patron Rôle de Coad (Coad, 1992)

En observant attentivement le diagramme de classes décrivant la structure du composant *Client*, nous retrouvons une imitation du patron *Rôle* pour modéliser le concept de *statut*. Le tableau 10 décrit partiellement la solution offerte par le patron *Rôle* (Coad, 1992) en utilisant le formalisme P-Sigma (Conte, 2001b). Ainsi le client joue un rôle différent selon les statuts qu'il possède. Le formalisme *P-Sigma* et le modèle de composants Symphony ne permettent pas de dire qu'un patron est imité dans un autre. Or, il serait intéressant de conserver une trace de l'imitation du patron *rôle* dans le composant *Client analyse*. La base *B-Sigma* doit donc proposer des relations inter-composants autres que celles définies dans les modèles de composants.

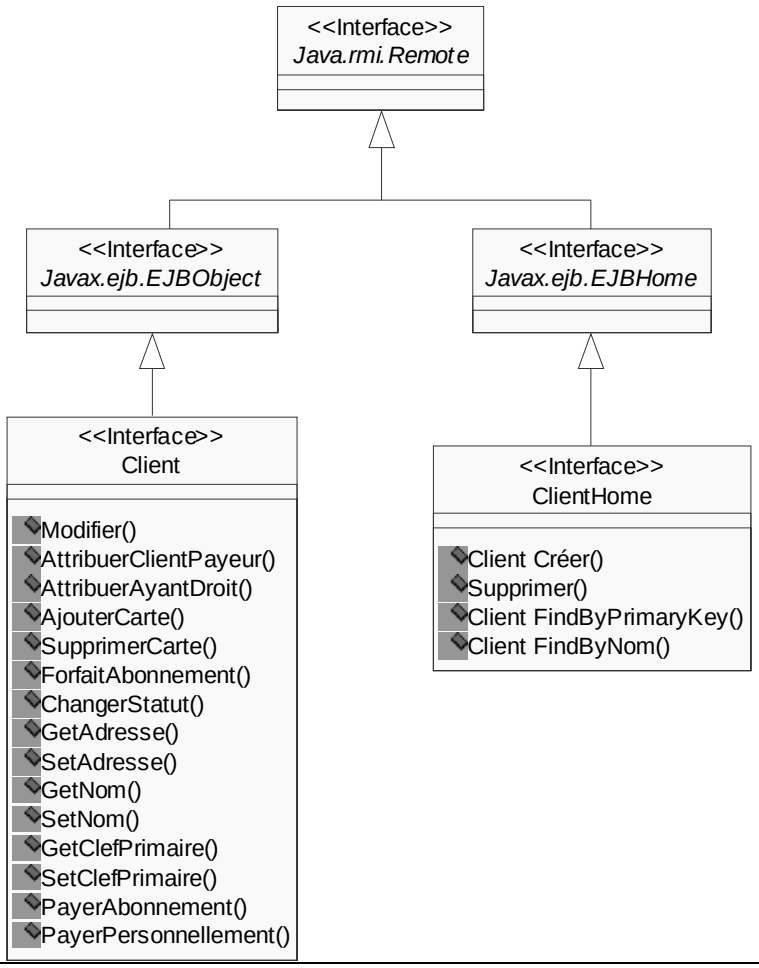
Tableau 10. *Patron Rôle (Coad, 1992).*

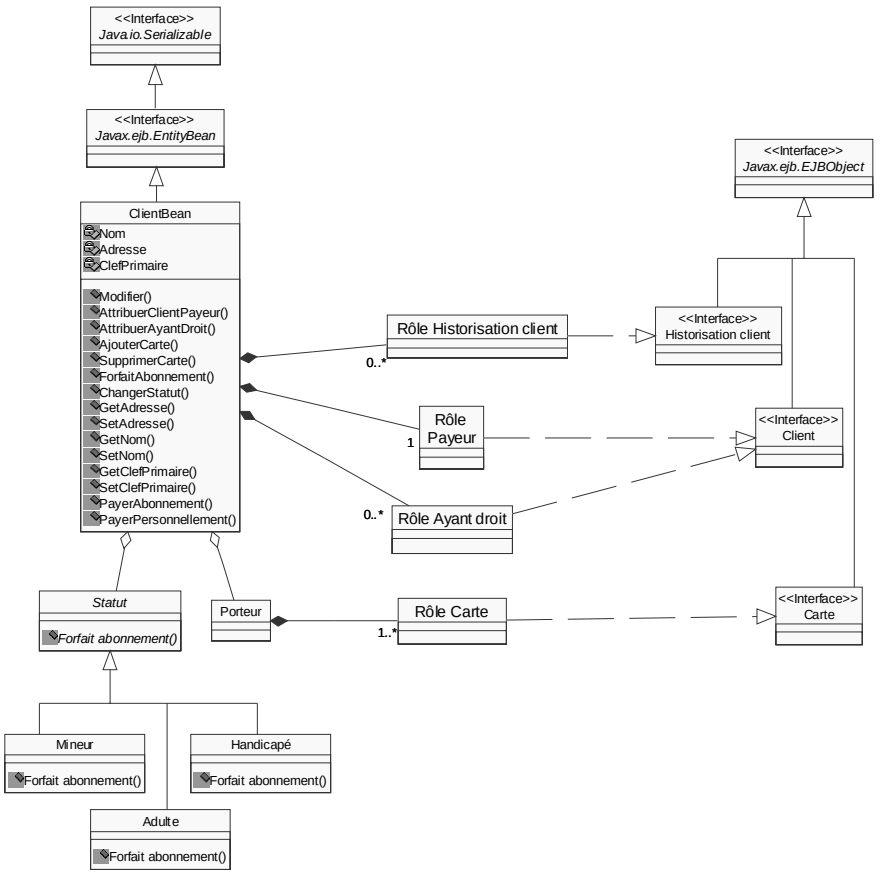
Nom	Rôle
Auteur	Coad
formalisme	P-Sigma
Problème	Un objet peut jouer plusieurs rôles successivement ou simultanément. Le patron rôle donne la possibilité à cet objet d'adhérer et/ou de perdre ces rôles.
Solution modèle	Ce patron permet la représentation des différents rôles, l'adhésion et ou la perte de ces derniers. Cette approche a l'avantage d'être plus concise et flexible que l'utilisation de l'héritage multiple pour modéliser les différents rôles d'un objet. 
Cas d'application	Le patron rôle est utilisé lorsqu'un objet a des valeurs d'attributs et des services variables selon le rôle joué. Il prend en compte les différentes perspectives d'objets.  Un livre a par exemple des caractéristiques intrinsèques telles qu'un Numéro d'exemplaire mais peut jouer différents rôles par ses instances correspondant à *l'Emprunt*, la *Réservation* et la *Maintenance*.

1.2.2 Conception

Pendant la phase de conception, le concepteur effectue le choix d'une technologie cible pour la réalisation du composant. Le tableau 11 décrit la conception du composant métier *Client* orientée vers la technologie des EJB.

Tableau 11. *Représentation du concept métier Client au niveau Conception.*

Nom	Client conception
Auteur	Khayati
Intention	Ce composant présente une conception orientée EJB du concept métier *Client* du domaine des systèmes d'information de transport.
Modèle de composants cible	Enterprise JavaBeans
Interfaces offertes	

Structure interne	
Participants	La classe *ClientBean* définit les services fonctionnels offerts par le composant EJB *Client*. Les interfaces *Client* et *ClientHome* sont générées à partir de la classe *ClientBean*. L'interface *Client* définit l'interface distante qui sera utilisée par les clients du composant EJB *Client* tandis que l'interface *ClientHome* définit la fabrique de composants qui gère toutes les instances du composant EJB *Client* gérées dans un containeur.
Utilise	Le composant *Client* utilise les composants suivants : - *Historisation* : permet d'historiser les événements de création ou de changement d'état d'un *Client*. - *Client* : une instance du composant *Client* a besoin de deux autres instance du composant client : la première indique le client payeur qui paye pour le client en cours, la deuxième indique le client à travers lequel le client en cours a des droits (par exemple, les enfants de moins de 7 ans peuvent par exemple avoir le droit de circuler sur le réseau gratuitement ou avec un tarif réduit si l'un de leurs parents est un client). - *Carte* : indique les cartes que possède un client.
Utilisé par	Le composant *Client* est utilisé par les composants suivants : - Client : le composant *Client* joue les rôles de *Payeur* et *Ayant droit*. - Agent : le composant *Client* joue le rôle de *Client*.

1.2.3 Logiciel

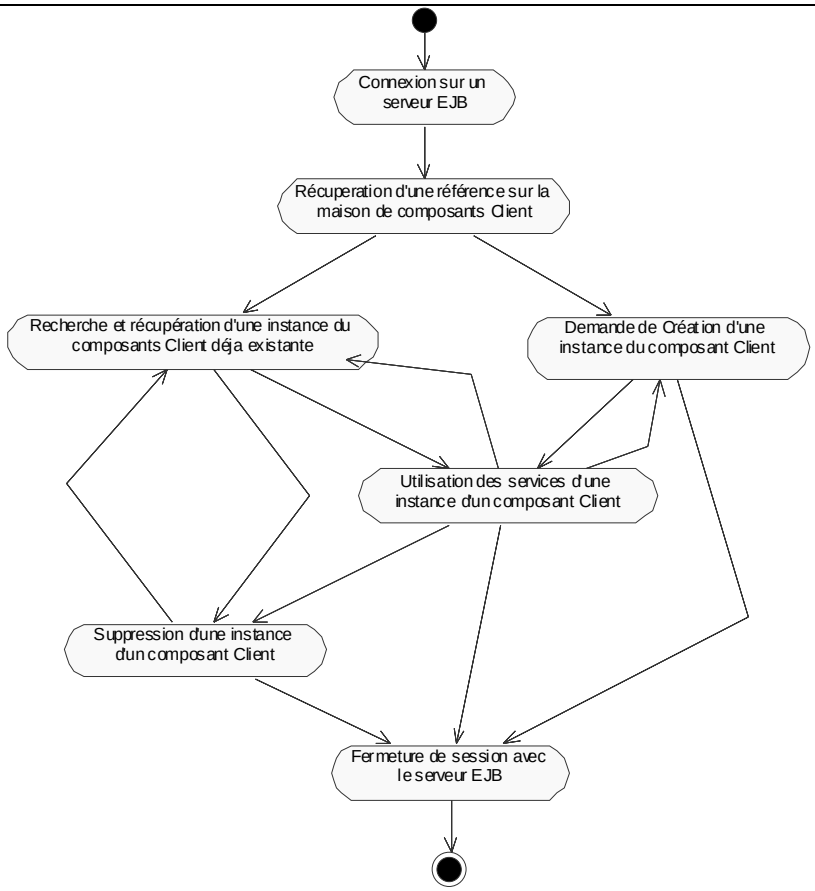
Le tableau 12 présente une description du composant logiciel EJB implantant le concept métier *Client*.

Tableau 12. *Représentation partielle du concept métier Client au niveau Logiciel.*

Nom	Client logiciel
Auteur	Khayati
Intention	Ce composant présente une implantation du concept métier *Client* du domaine des systèmes d'information transport en utilisant la technologie des EJB.
Modèle de composants	Enterprise JavaBeans
Type de Bean	Entity Bean

Interfaces offertes	public interface Client extends EJBObject{
	public void modifier();
	public void attribuerClientPayeur(String ClefPayeur);
	public void attribuerClientAyantDroit(String ClefPayeur);
	public void ajouterCarte(String ClefCarte);
	public void supprimerCarte(String ClefCarte);
	public abstract double forfaitAbonnement();
	public void changerStatut(String Statut);
	public String getAdresse();
	public void setAdresse(String Adresse);
	public String getNom();
	public void setNom(String Nom);
	public String getClefPrimaire();
	public void setClefPrimaire(String ClefPrimaire);
	public abstract void PayerAbonnement();
	public abstract void PayerPersonnellement();
	}
	
	public interface ClientHome extends EJBHome{
	public Client creer(String nom, String adresse, String Clefprimaire);
	public Client supprimer(String Clefprimaire);
	public Client findByPrimaryKey(String ClefPrimaire);
	public Collection findByNom(String nom) ;
	}

Classe Bean	public abstract class ClientBean implements EntityBean{
	public void modifier();
	public void attribuerClientPayeur(String ClefPayeur);
	public void attribuerClientAyantDroit(String ClefPayeur);
	public void ajouterCarte(String ClefCarte);
	public void supprimerCarte(String ClefCarte);
	public abstract double forfaitAbonnement();
	public void changerStatut(String Statut);
	public String getAdresse();
	public void setAdresse(String Adresse);
	public String getNom();
	public void setNom(String Nom);
	public String getClefPrimaire();
	public void setClefPrimaire(String ClefPrimaire);
	public abstract void PayerAbonnement();
	public abstract void PayerPersonnellement();
	}

Démarche de réutilisation	
Cas d'utilisation	Exemple d'utilisation de la fabrique de composants (home interface) par un client : Context initial = new InitialContext(); Object objref = initial.lookup("ClientBean"); ClientHome home = (ClientHome)PortableRemoteObject.narrow(objref, ClientHome.class) ; Client durand = home.creer("Durand") ; Exemple d'utilisation de l'interface distante (Remote interface) par un client : … Client durand = home.creer("Durand", "Albert", "1 rue des Champs Élysée Paris France) ; System.out.println(durand.getNom()) ; …

1.2.4 Synthèse

La figure 29 montre l'organisation des descriptions des composants que nous venons de présenter (cf. tableau 9, tableau 10, tableau 11, tableau 12). Ainsi, il est tout à fait possible d'avoir plusieurs composants du niveau conception qui correspondent au même composant analyse. Ce cas de figure est possible si on décide de réaliser un composant métier avec deux ou plusieurs technologies différentes.

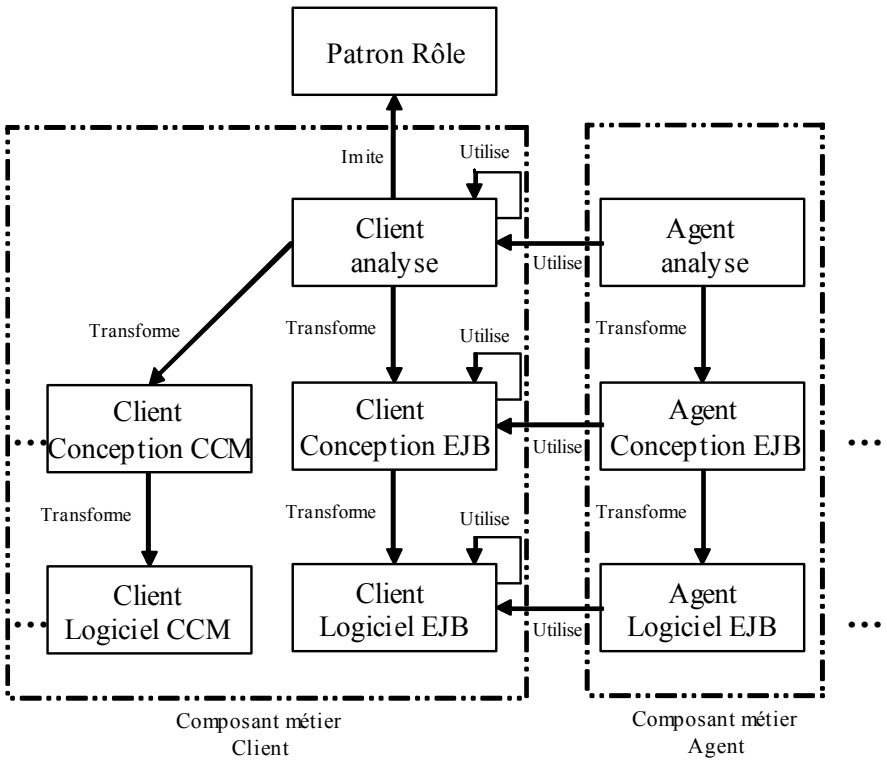


Figure 29. *Exemple d'un schéma partiel de l'organisation d'une collection de descriptions de composants gérée par la base B-Sigma.*

Dans la suite de ce chapitre nous présentons plus en détail tous les concepts que doit gérer la base *B-Sigma* et nous présentons le modèle de base descriptive de composants *C-Sigma* sous-jacent.

2. Principes fondamentaux de la base *B-Sigma*

La base descriptive de composants *B-Sigma* est un système d'information dédié à la gestion et la capitalisation des composants réutilisables. La section précédente a présenté les motivations et les besoins qui poussent à la réalisation d'un tel outil. Nous proposons dans cette section le modèle *C-Sigma* permettant de construire un tel outil capable de gérer la base *B-Sigma*. Nous étudions en premier lieu l'organisation interne d'une base *B-Sigma*. Puis, nous présentons l'organisation générale des concepts présents dans le modèle *C-Sigma*.

2.1 Organisation interne de la base *B-Sigma*

Lors de la réutilisation des composants, les besoins d'un ingénieur d'applications varient selon le niveau d'abstraction dans lequel il intervient. Garder un lien de traçabilité entre les représentations des composants dans les différents niveaux d'abstraction assure une meilleure navigabilité de la base de composants et facilite le travail des ingénieurs d'applications en assurant une continuité dans le processus de développement. Ceci est possible en adoptant un modèle d'organisation en couches de la base de composants (cf. figure 30). Chacune de ces couches représente un niveau d'abstraction de composants qui possède ses propres composants, modèles de composants et relations inter-composants. La base *B-Sigma* met à la disposition de ses utilisateurs au moins deux types de relations : les relation horizontales (par exemple les relations définies dans les modèles de composants) spécifiques à un niveau d'abstraction, et les relations verticales (par exemple la relation *Implante*) inter-niveaux d'abstractions qui permettent d'assurer la traçabilité et la navigabilité entre les différents niveaux.

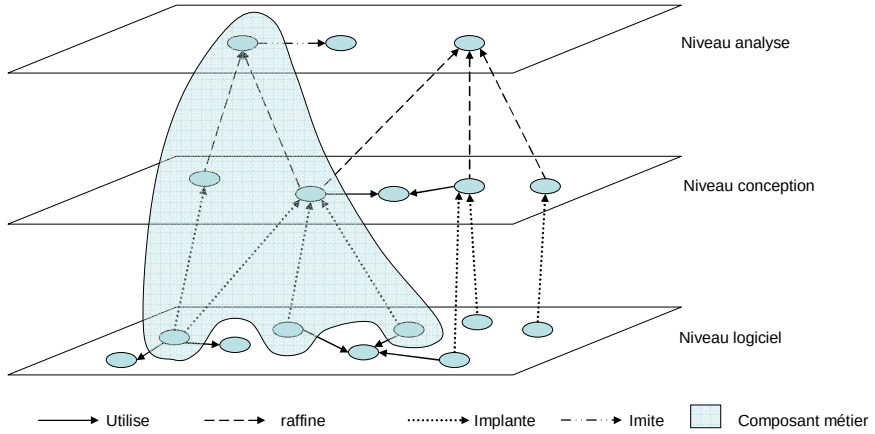


Figure 30. *Niveaux d'organisation d'une BDC B-Sigma.*

En plus des trois niveaux d'abstraction analyse, conception et logiciel, la base *B-Sigma* introduit le niveau d'abstraction métier. Un composant métier est la représentation d'un concept métier par un ensemble de composants appartenant aux différents niveaux d'abstraction de composants gérés par une base *B-Sigma* (cf. figure 29 et figure 30).

2.2 Modèle de la base *B-Sigma*

Une base *B-Sigma* gère principalement trois types d'entités : les composants, les descriptions et les relations (cf. figure 31). Les composants sont matérialisés par un ensemble d'artéfacts qui peuvent être des modèles, du code source, de la documentation etc. Une base *B-Sigma* gère principalement deux types de descriptions : les descriptions de sources de composants (*description de sources*) et les descriptions de composants (*description de composants*). Une *description de composants* regroupe toutes les informations relatives à un composant gérées par la base. Ces informations peuvent être générales (comme le nom des auteurs, date de création, etc.) ou bien spécifiques comme les informations facilitant la réutilisation (cf. la description *Réutilisation* figure 31) ou dépendantes du modèle de composants (cf. la description *Modèle de composants* figure 31).

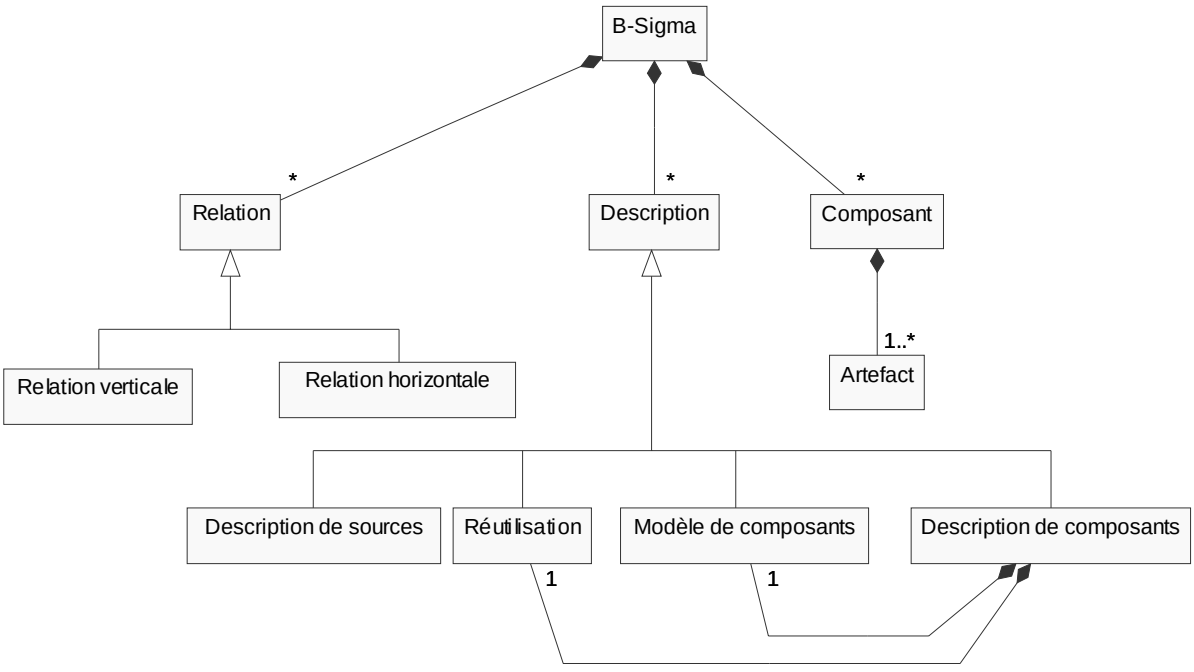


Figure 31. *Le modèle de la base B-Sigma.*

Comme tout système d'information, la base *B-Sigma* est appelée à s'adapter pour satisfaire les besoins des utilisateurs et à évoluer pour pouvoir intégrer les nouveaux composants qu'elle sera amenée à gérer. Le nombre de modèles de composants et les informations nécessaires à leur documentation peuvent évoluer au cours du temps. Maximiser la capacité d'évolution et d'adaptation de la base *B-Sigma* réside dans l'anticipation des besoins utilisateurs en utilisant un niveau d'abstraction supérieur : le modèle *C-Sigma*.

2.3 Le modèle *C-Sigma*

C-Sigma est le modèle permettant l'implantation d'un outil capable d'instancier et de gérer des bases telles que *B-Sigma*. Pour que cet outil soit évolutif et facilement adaptable, le modèle *C-Sigma* prévoit des points d'évolution. Il est en effet divisé en deux parties (cf. figure 32) : une partie abstraite qui définit les concepts gérés par une base *B-Sigma* (*Composant, Description, Relation, Description de composants*, *Modèle de composants*, *Description de sources*, etc.) ; et une partie concrète qui propose des spécialisations concrètes des abstractions définies dans la partie abstraite du modèle. La séparation de la partie abstraite et de la partie concrète du modèle *C-Sigma* garantit une meilleure évolutivité de la base *B-Sigma* vis-à-vis de nouveaux besoins comme l'ajout de nouveaux modèles de composants ou de relations. Dans ce cas de figure, il suffit de faire évoluer la partie concrète du modèle *C-Sigma* en ajoutant de nouvelles spécialisations concrètes de la classe abstraite *Modèle de composants* sans altérer l'intégrité et l'organisation des données gérées par la base *B-Sigma*.

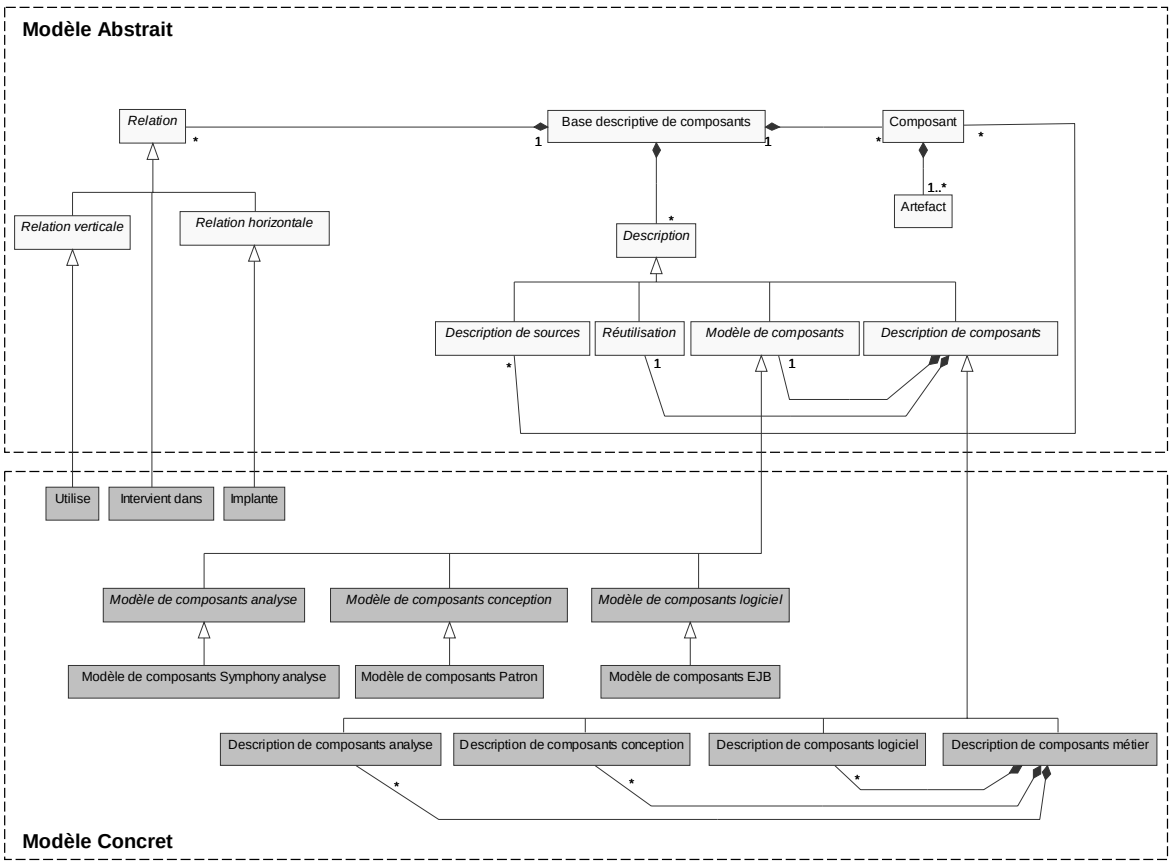


Figure 32. *Modèle C-Sigma de bases descriptives de composants.*

3. Les concepts du modèle *C-Sigma*

Nous présentons dans cette section chacun des concepts abstraits introduits dans la figure 32 tout en donnant des exemples concrets instanciables pour décrire des composants. Nous distinguons les spécialisations concrètes des entités présentes au niveau abstrait du modèle *C-Sigma* en utilisant un fond gris.

3.1 Base descriptive de composants

Le contenu de toute base descriptive de composants est décrit par les deux rubriques *Nom* et *Intention* (cf. figure 33). Le *Nom* représente un identifiant de l'instance de la base dans le système de gestion de bases descriptives de composants. La rubrique *Intention* définit une description du contenu de la base.

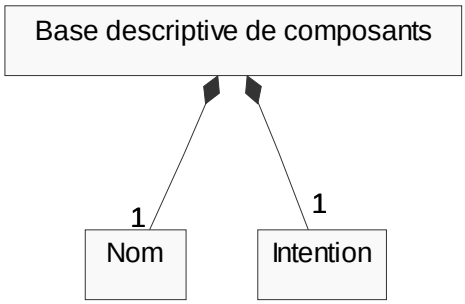


Figure 33. *Les bases descriptives de composants en C-Sigma.*

3.2 Composants et artefacts

Le modèle *C-Sigma* définit trois classes de composants : *Composant besoin*, *Composant conception* et *composant logiciel* (cf. figure 34). Un composant est souvent constitué d'un ou de plusieurs *artefacts*. Un *artefact* représente une portion physique d'un composant. Il peut être une archive contenant le code source ou le code binaire d'un composant ou bien un fichier contenant la documentation du composant sous forme textuelle ou semi formelle (diagrammes UML). Nous faisons le choix de ne pas gérer les composants directement : une instance de la classe *artefact* indique l'emplacement physique de la portion de composant. Le modèle *C-Sigma* définit deux types d'artefacts (cf. figure 35) : *artefact web* encapsule une *URL* indiquant l'emplacement sur le Web où le composant peut être récupéré ; *Artefact Outil externe* est utilisé dans le cas où le composant serait géré par un autre outil de l'environnement de développement, dans ce cas il indique le nom de l'outil et le nom du composant.

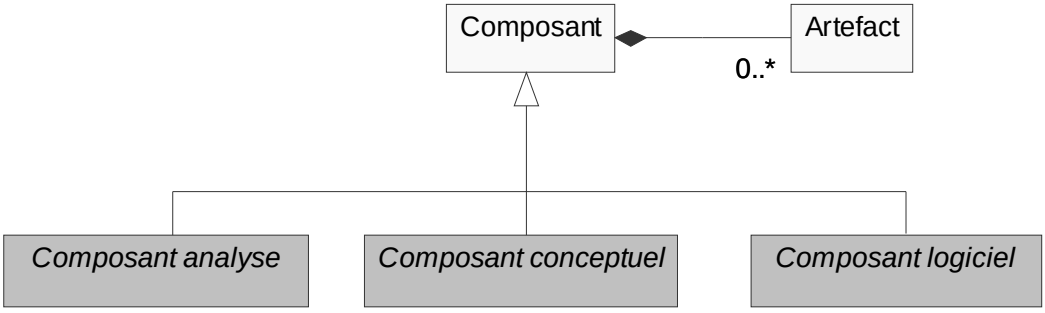


Figure 34. *Les composants en C-Sigma.*

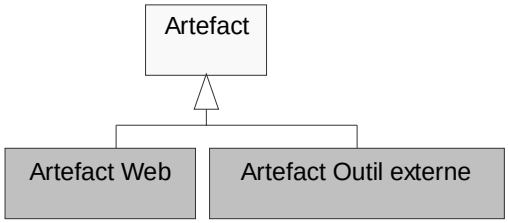


Figure 35. *Les types d'artefacts en C-Sigma.*

3.3 Relations

Une base descriptive de composants doit gérer un ensemble de descriptions de composants reliées entre elles par des relations. Chacune de ces relations représente un lien sémantique existant entre deux ou plusieurs entités gérées par la base.

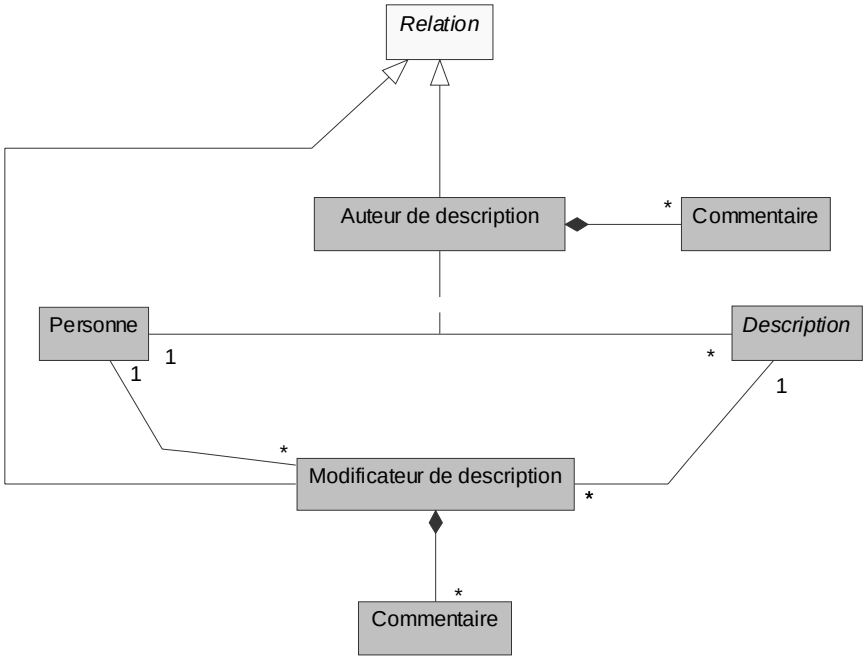


Figure 36. *Exemple de relation inter-descriptions.*

La figure 36 présente deux exemples de relations qui assurent la traçabilité de la création et de la modification des descriptions gérées par une base descriptive de composants. La relation *Auteur de description* permet d'indiquer la *Personne* auteur d'une *Description* gérée dans une base descriptive de composants. La relation *Modificateur de description* liste les personnes qui ont modifié une *Description*. Une relation peut déclarer une ou plusieurs rubriques pour décrire sa propre sémantique (cf. figure 36).

Le modèle *C-Sigma* définit deux autres types de relations permettant l'organisation en couches d'une base descriptive de composants (cf. figure 37) : les *relations horizontales* et les *relations verticales*. Une relation inter-composants est décrite par les deux rubriques textuelles *nom* et *sémantique*.

Le concept de personne représenté par la description *personne* est indépendant du niveau d'abstraction du composant qu'il modifie ou qu'il crée ; donc les relations *Auteur de description* et *Modificateur de description* ne sont ni des relations verticales ni des relations horizontales.

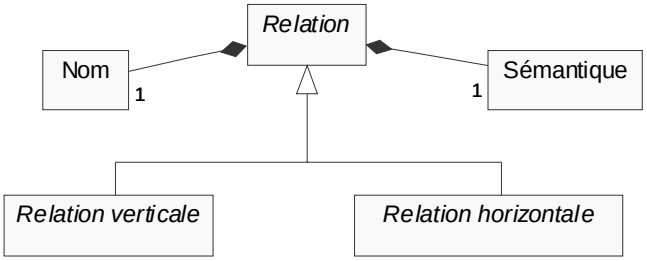


Figure 37. *Les deux types de relations inter-descriptions de composants en C-Sigma.*

3.3.1 Les relations horizontales

Les relations horizontales établissent des liens uniquement entre composants du même niveau d'abstraction (même couche). Un exemple de relations horizontales est la relation *utilise* présentée dans la figure 38. La relation abstraite *utilise* se spécialise en trois relations concrètes : *utilise analyse*, *utilise conception* et *utilise logiciel*. Chacune de ces relations d'utilisation est spécifique à un niveau d'abstraction bien déterminé.

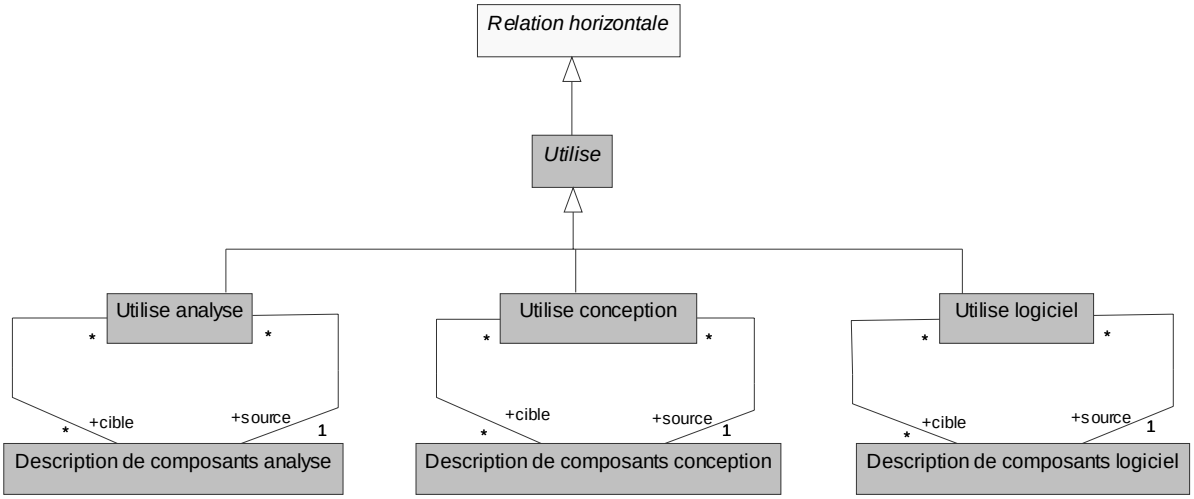


Figure 38. *La relation horizontale utilise.*

3.3.2 Les relations verticales

Les relations verticales établissent des liens entre les composants de niveaux d'abstraction différents (couches différentes). La figure 39 présente deux exemples de relations verticales : *raffine* et *implante*. La relation *raffine* relie la description d'un composant du niveau conception dépendant d'une technologie de réalisation à la description d'un composant du niveau analyse contenant la solution conceptuelle indépendante de toute technologie de réalisation. L'utilisateur de la base descriptive de composants peut documenter ce lien en ajoutant des commentaires grâce à la rubrique *commentaire* ou en indiquant le patron décrivant le processus de raffinement. La relation *implante* relie la description d'un composant du niveau spécification à la description d'un composant du niveau implantation. Elle peut indiquer éventuellement un patron processus décrivant le processus de transformation de la conception en code source : mise en œuvre d'un processus de transformation de modèles type MDE (OMG, 2005).

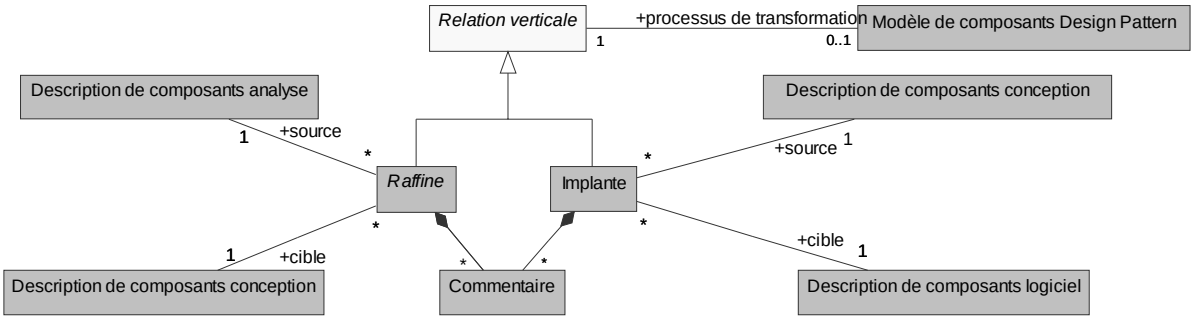


Figure 39. *Les relations verticales implante et raffine.*

3.4 Descriptions

Une *description* est une structure arborescente composée par un ensemble de rubriques et de sous-descriptions. Le modèle *C-Sigma* (cf. figure 32) définit quatre spécialisations de la classe *descriptions* : les *descriptions de composants*, les *modèles de composants*, les *réutilisations* et les *descriptions de sources*.

3.4.1 Description de composants

En plus des rubriques générales comme *Nom*, *Intention* et *Auteur*, une description de composants peut comporter deux autres types d'informations dans deux sous-descriptions : *modèle de composants* regroupe les informations dépendantes du modèle de composants, *réutilisation* regroupe les informations décrivant le processus de réutilisation. Le modèle *C-Sigma* définit quatre spécialisations concrètes de la classe abstraite *description de composants* : *description de composants analyse*, *description de composants conception*, *description de composants logiciel* et *description de composants métier*. Chacune de ces descriptions représente les composants d'un niveau d'abstraction différent. Le niveau d'abstraction métier est transversal aux trois autres niveaux d'abstraction (cf. figure 30) : analyse, conception et logiciel.

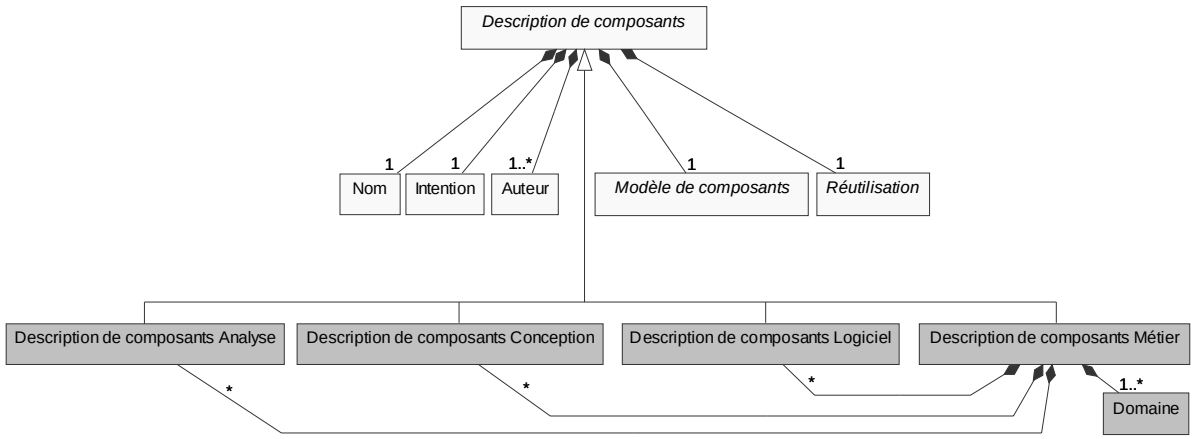


Figure 40. *Les descriptions de composants en C-Sigma.*

3.4.2 Modèle de composants

La description *modèle de composants* regroupe les informations dépendantes du modèle de composants utilisé pour la description de la solution offerte par un composant. Le modèle *C-Sigma* définit quatre niveaux d'abstraction de *modèle de composants* (cf. figure 41) : *modèle de composants analyse*, *modèle de composants conception*, *modèle de composants logiciel*, *modèle de composants métier*. Chacune des descriptions abstraites représentant les niveaux d'abstraction peut être spécialisée pour aboutir à une description d'un modèle de

composants comme EJB, les patrons de conception (*design patterns*) ou les composants métier Symphony (Hassine, 2002).

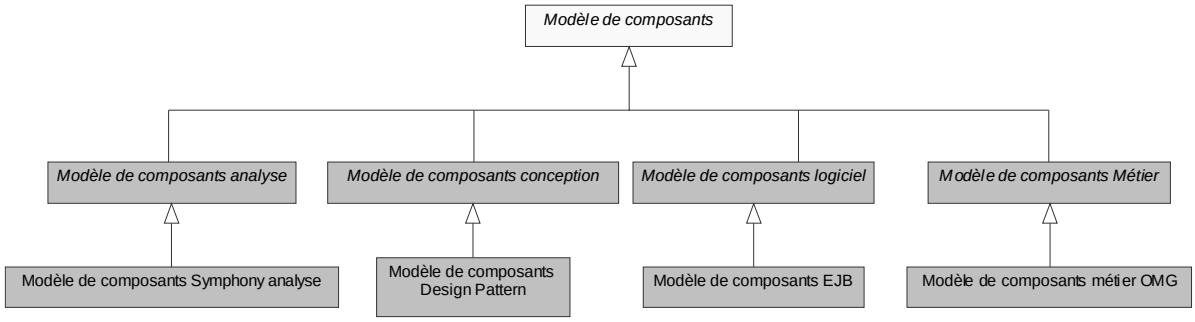


Figure 41. *Les modèles de composants en C-Sigma.*

La figure 42 montre la représentation en *C-Sigma* de la description d'un composant Symphony au niveau de l'analyse (cf. tableau 9). Les rubriques contenant des informations indépendantes du modèle de composants sont héritées de la classe abstraite *Description de composants* alors que les rubriques dépendantes du modèle de composants Symphony analyse sont regroupées dans la classes *Modèle de composants Symphony analyse*. La relation horizontale *Participant* pointe sur les descriptions des classes contenues dans la solution offerte par le composant. La description d'une classe contient son *Nom* et les descriptions des *méthodes* qu'elle déclare.

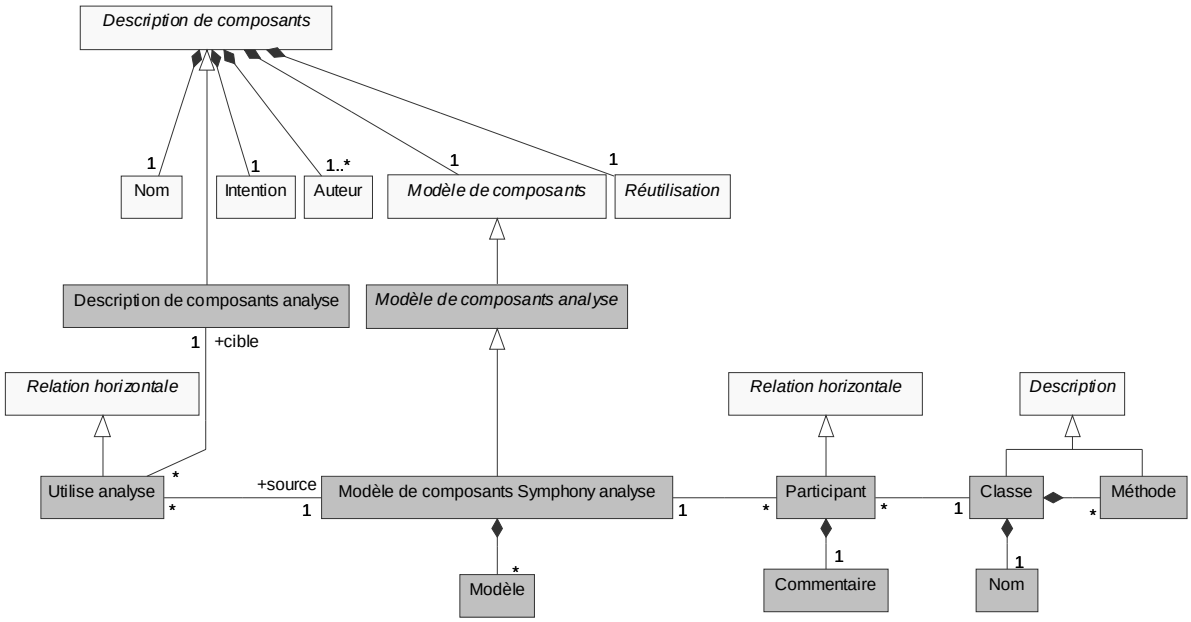


Figure 42. *Représentation C-Sigma du modèle de composants Symphony analyse.*

3.4.3 Réutilisation

La description *Réutilisation* (cf. figure 32) permet de générer l'historique des réutilisations d'un composant et documente le processus de réutilisation. Le modèle *C-Sigma* définit les rubriques *Forces* et *Faiblesses* (cf. figure 43) pour décrire les avantages ou les inconvénients engendrés lors des réutilisations précédentes d'un composant.

Les *Cas de réutilisation* d'un composant représentent une source d'information utile pour les réutilisations futures d'un composant. Le modèle *C-Sigma* définit la relation horizontale *Cas de réutilisation* qui relie un composant réutilisable à l'ensemble des composants qui l'ont réutilisé en adoptant la même technique de réutilisation (cf. figure 43). Si un composant réutilisable peut être adapté en utilisant plusieurs techniques de réutilisation, alors plusieurs

instances de cette relation seront reliées à la description de ce composant. Un ingénieur d'applications peut ajouter ses *Commentaires* lors de chaque réutilisation.

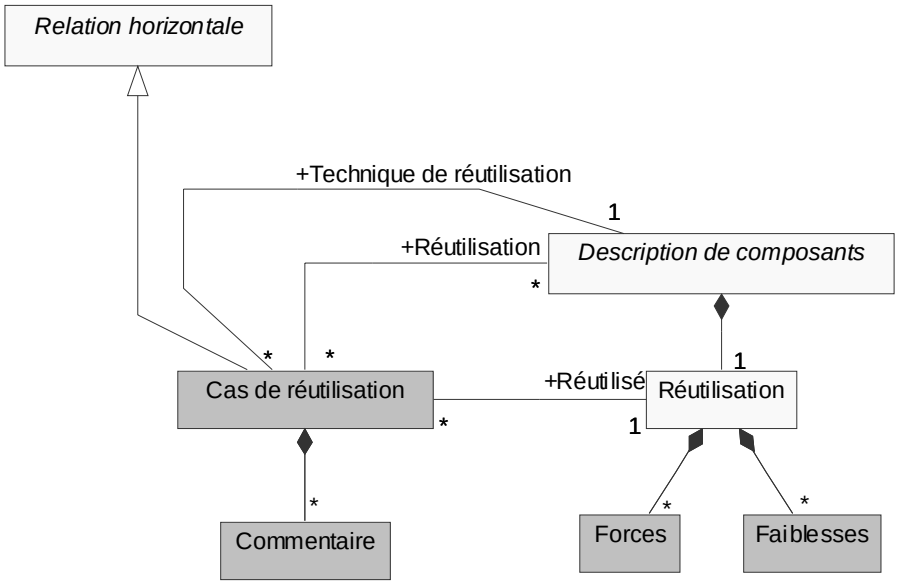


Figure 43. *Description C-Sigma de la réutilisation des composants.*

3.4.4 Description de sources

Une instance de la classe *Description de sources* décrit une source de composants alimentant une base descriptive de composants. Une source de composants est identifiée par un *Nom* et décrite par des *Commentaires*. Le modèle *C-Sigma* définit deux types de *Description de sources* (cf. figure 44) : *Description de sources internes* et *Description de sources externes*. Une source interne peut être une *Personne* ou une *Equipe de développement*. Une Source externe peut être un *Fournisseur internet*. Une instance de la relation *Fonction* indique si une *Personne* joue une ou plusieurs fonctions dans une ou plusieurs *Equipe de développement*.

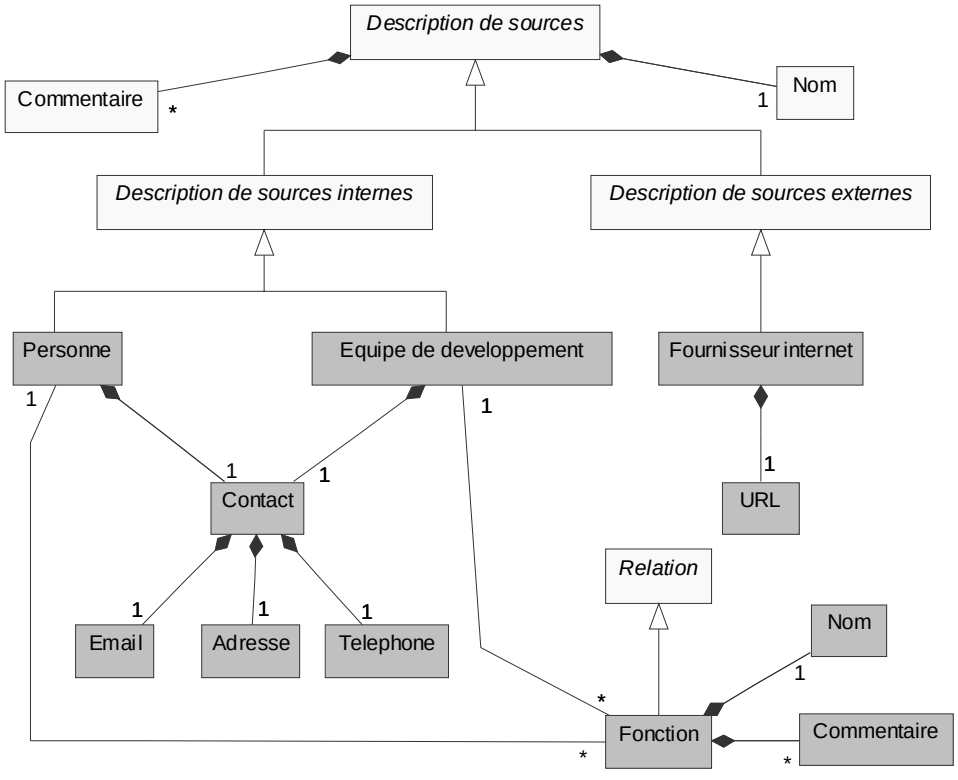


Figure 44. *Descriptions C-Sigma de sources de composants.*

4. Conclusion

La base *B-Sigma* est un système d'information dédié à la gestion des composants réutilisables. Son rôle est de promouvoir la réutilisation des composants tout au long des phases de développement et d'être au centre du processus de réutilisation. La section 1 étudie les besoins des utilisateurs d'une base *B-Sigma* à travers des exemples. Les sections 2 et 3 présentent le modèle *C-Sigma* dont la base *B-Sigma* est une instance. Le modèle de bases descriptives de composants *C-Sigma* est capable de représenter des bases de composants hétérogènes du point de vue niveau d'abstraction (cf. section 3.4.1), du point de vue modèle de composants (cf. section 3.4.2) et du point de vue source de composants (cf. section 3.4.3).

Le modèle *C-Sigma* anticipe certains aspects de l'évolution des besoins utilisateurs en proposant des points d'évolution. Il est ainsi divisé en deux parties : une partie abstraite et une partie concrète (cf. figure 32).

Ainsi, nous avons répondu partiellement dans cette section aux objectifs que nous nous sommes fixés dans la problématique concernant l'évolutivité de la base *B-Sigma*. D'un autre côté, nous avons relevé le défi concernant l'hétérogénéité des modèles de composants (respectivement des niveaux d'abstraction) des composants qu'elle gère.

Pour améliorer l'évolutivité des bases descriptives de composants, nous présentons dans le chapitre suivant le métamodèle *M-Sigma*. La figure 45 montre les trois niveaux d'abstraction nécessaires pour la réalisation d'un système de gestion de bases descriptives de composants.

Le chapitre VI quant à lui décrira un système de patrons s'appuyant sur le modèle *C-Sigma* et guidant l'évolution d'une base descriptive de composants.

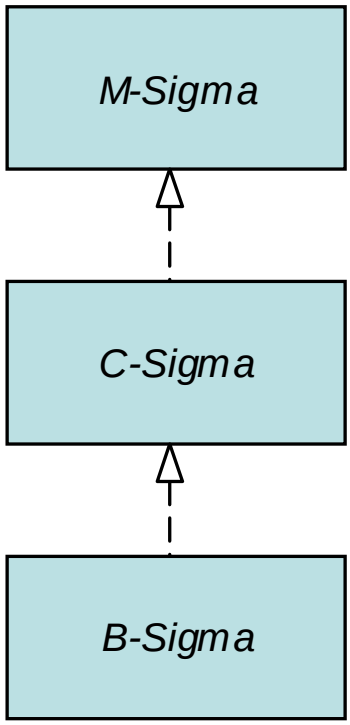


Figure 45. *Les trois niveaux d'abstraction gérés dans un SGBDC.*

V. Un système de gestion de bases descriptives de composants

Dans un contexte de réutilisation, un système d'information est un assemblage de composants sélectionnés dans des bases de composants réutilisables. Les besoins d'un ingénieur changent en terme de types et niveaux d'abstractions de composants selon la phase de développement où il intervient. Par conséquent, un environnement de développement supportant la réutilisation doit offrir dans chacune des phases de développement les techniques de recherche de composants adaptées aux besoins des ingénieurs d'applications. Cet environnement doit nécessairement inclure un modèle de représentation interne de composants et un modèle d'organisation des bases de composants compatibles avec toutes les techniques de recherche de composants qu'il propose.

Pour notre système de gestion de bases descriptives de composants, nous proposons le métamodèle *M-Sigma* qui permet d'instancier des modèles d'organisation de bases descriptives de composants (BDC) garantissant une représentation interne des composants découplée des techniques de recherche de composants qui les exploitent.

1. Principes de base

L'état de l'art présenté dans le chapitre 2 a présenté une classification des techniques de recherche de composants (TRC). Ces techniques sont adaptées aux différents aspects des composants, mais souvent, les bases de composants exploitent une seule TRC. Ceci est dû principalement au fait que chacune des bases adopte une représentation de composants dédiée à sa TRC. D'un autre côté, la représentation d'un composant dans une base de composants peut évoluer au cours du temps selon l'aspect des composants qui intéressent les ingénieurs d'applications. Il devient clair que le modèle de représentation des composants doit lui-même être adaptable et évolutif.

Nous présentons dans cette section le métamodèle *M-Sigma* (Khayati, 2004) (Khayati, 2004a) qui permet d'implanter un système de gestion de bases descriptives de composants adaptable et évolutif (SGBDC). Un SGBDC permet de décrire et d'instancier des modèles de bases descriptives de composants comme le modèle *C-Sigma* présenté dans le chapitre précédent et d'appliquer des techniques de recherche de composants.

Le concept de *Description* est le concept fondamental du métamodèle *M-Sigma*. Une *Description* est une structure composée d'un ensemble de rubriques (*Item*) pour la description d'un composant (*Component*). Une rubrique est une facette (ASSET, 1993) définie par un nom et un type. La valeur d'une rubrique ne fait pas nécessairement partie d'un vocabulaire fermé. Le type d'une rubrique peut être un type simple comme une chaîne de caractères, ou un type avancé comme un diagramme de classes UML ou une signature de méthode. Cette approche permet d'appliquer à chaque type de rubrique une ou plusieurs techniques de

recherche qui lui sont spécifiques. Il est donc possible d'utiliser une ou plusieurs techniques de recherche pour retrouver la description d'un composant et le composant qui lui est associé. Le concept de *Description* permet de découpler les TRC de la représentation des composants dans une base descriptive de composants. Il sera ainsi toujours possible d'intégrer de nouvelles TRC au SGBDC.

Le métamodèle *M-Sigma* est une adaptation du métamodèle UML 1.5 (OMG, 2003). Les quatre sous-paquetages *Core package Backbone*, *Core package Classifiers*, *Core package Relationships* et *Core package Auxiliary elements* du paquetage *Core package* sont étendus et deux nouveaux paquetages : *Package Data Types* et *Package Instances* sont ajoutés. Nous présentons dans la suite de cette section ces cinq paquetages (cf. figure 46). Les métaclasses dont la sémantique ou la structure (attributs et méthodes) ont été changées sont indiquées avec la mention *UML-étendu*. Les métaclasses spécifiques à *M-Sigma* sont indiquées avec la mention *M-Sigma* et représentées avec une couleur grise dans les diagrammes de classes UML.

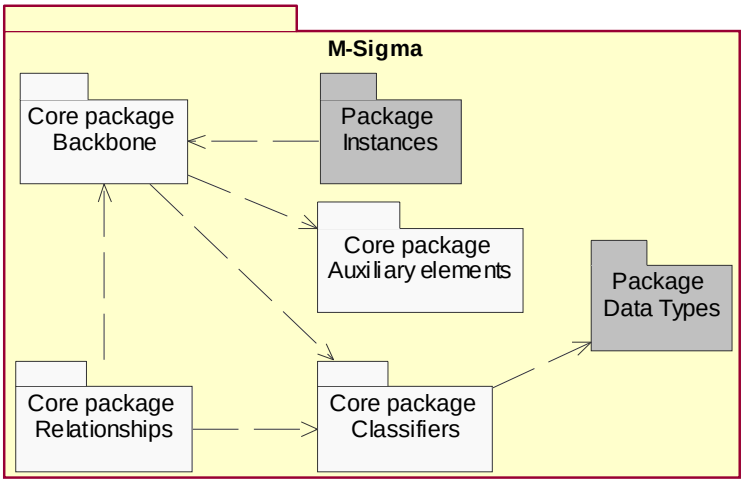


Figure 46. *Les paquetages du métamodèle M-Sigma.*

Pour assurer une bonne compréhension de l'ensemble du métamodèle, nous sommes contraints à présenter des éléments qui figurent déjà dans le métamodèle UML 1.5 et à ne pas nous limiter aux éléments spécifiques à *M-Sigma*.

2. Le Core package Backbone étendu

Le *Core package Backbone* (cf. figure 47) représente le squelette du métamodèle *M-Sigma*. Il définit les métaclasses de base à partir desquelles sont dérivées toutes les autres métaclasses ainsi que les mécanismes d'encapsulation (*Bundle*) et de composition (*Composition*). Le mécanisme d'encapsulation est modélisé par l'imitation du patron Composite de Gamma (Gamma, 1995).

Nous présentons dans la suite de cette section les métaclasses du paquetage *Core package Backbone*.

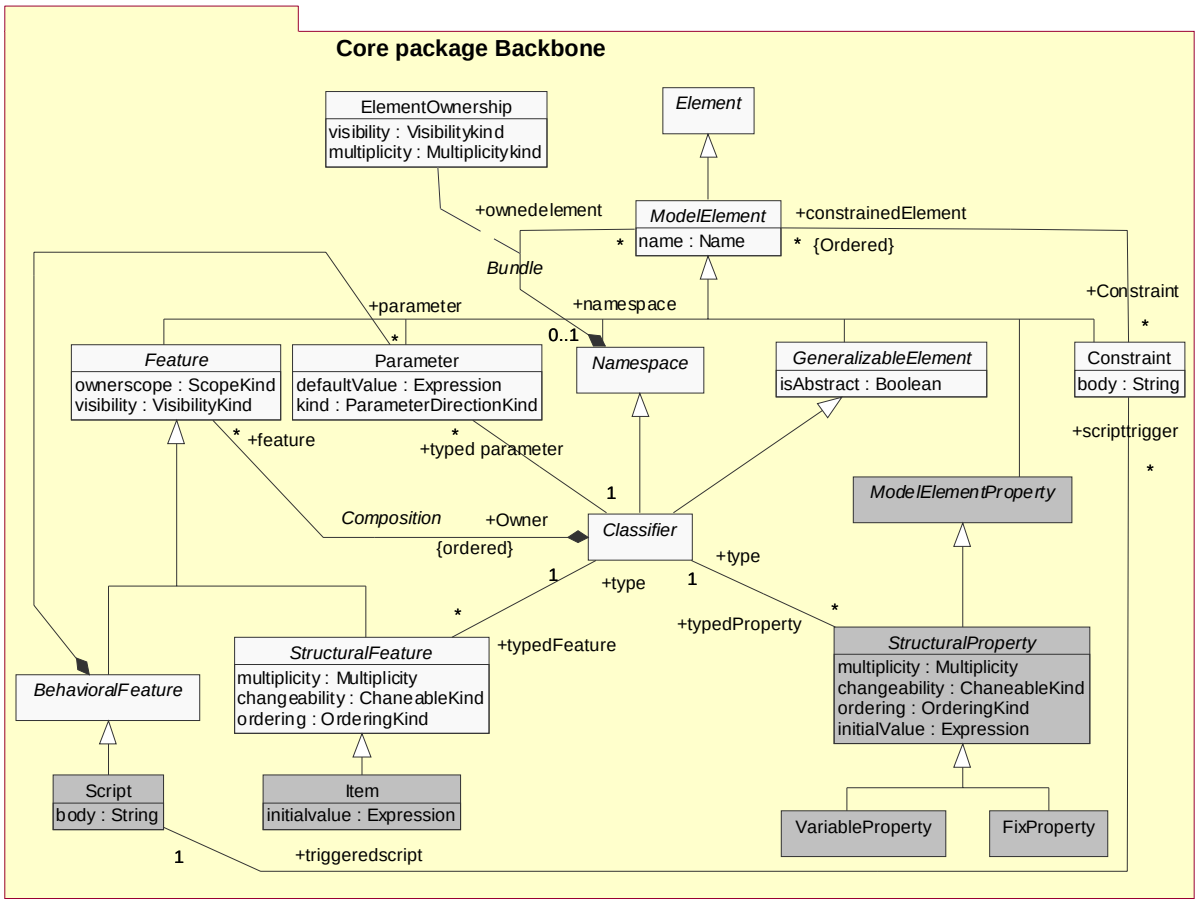


Figure 47. *Core package Backbone.*

2.1 Element

C'est la métaclasse abstraite qui se trouve au sommet de la hiérarchie des métaclasses du métamodèle *M-Sigma*.

2.2 ModelElement

C'est une abstraction de tous les éléments de modélisation servant à la modélisation d'une base de composants. L'attribut *name* est défini dans la classe abstraite *ModelElement*, ainsi tout élément de modélisation porte un nom.

Attributs

– name : une chaîne de caractères servant d'identifiant pour le *ModelElement* à l'intérieur du *Namespace* le contenant.

Associations

– constraint : un ensemble de contraintes (*Constraint*) affectant le *ModelElement*.

– namespace : désigne le *Namespace* contenant le *ModelElement*.

– presentation : un ensemble de *PresentationElements* qui présentent une vue du *ModelElement* (cf. section 5.1).

– comment : un ensemble de *Comments* qui commentent le *ModelElement* (cf. section 5.2).

2.3 Namespace

C'est une entité de modélisation abstraite du modèle encapsulant un ensemble de *ModelElement* grâce à la relation *Bundle*. Le nom d'un *ModelElement* est unique dans un *Namespace*. Une spécialisation concrète de *Namespace* doit contenir des contraintes supplémentaires sur le type d'éléments qu'elle peut contenir.

Associations

– ownedElement : désigne un ensemble de *ModelElements* possédés par le *Namespace*. Cette association possède une classe d'association *ElementOwnership*. C'est l'association inverse de *namespace* (cf. section 2.2)

2.4 ElementOwnership (UML étendu)

C'est une classe d'association qui définit la visibilité d'un *ModelElement* contenu dans un *Namespace* par la relation d'encapsulation (*Bundle*).

Attributs

– visibility : spécifie si un *ModelElement* est visible et peut être référencé par d'autres *ModelElements*. Il existe trois types de visibilité :

- public : tout *ModelElement* extérieur au *Namespace* (contenant) peut voir le *ModelElement* (contenu).
- protected : tout descendant du *Namespace* (contenant) peut voir le *ModelElement* (contenu).
- private : uniquement les *ModelElements* contenus dans le *Namespace* (contenant) peuvent voir son contenu.

– multiplicity : indique le nombre d'instances de *ModelElement* qui peuvent être encapsulées par une instance de *NameSpace* en utilisant la relation *Bundle*.

2.5 Constraint (UML étendu)

C'est une contrainte ou une condition sémantique définie sur les *ModelElements* du même *Namespace* et les *Features* du même *Classifier* pour que le modèle soit valide. La contrainte est exprimée sous la forme d'une expression booléenne. En cas de violation de la contrainte, il y a activation du *Script* qui lui est associé.

Attributs

– body : une expression booléenne qui doit être vraie si elle est évaluée par rapport à une instance du métamodèle *M-Sigma*.

Associations

– constrainedElement : liste les *ModelElements* affectés par la contrainte.
– triggeredScript : indique le *Script* déclenché en cas de violation de la contrainte.

2.6 GeneralizableElement (UML étendu)

Un *GeneralizableElement* représente une entité abstraite du modèle capable de participer à la relation *Generalization* (cf. section 3.2).

Attributs

– isAbstract : indique si un *ModelElement* est instanciable ou non. La valeur vraie indique que le *ModelElement* n'est pas instanciable.

Associations

– generalization : désigne une *Generalization* dans laquelle le *GeneralizableElement* joue le rôle de l'élément général (cf. figure 49).

– specialization : désigne une *Generalization* dans laquelle le *GeneralizableElement* joue le rôle de l'élément spécialisé (cf. figure 49).

Propriétés héritées

Les propriétés suivantes sont héritées par un *GeneralizableElement* spécialisé à partir du *GeneralizableElement* général à travers une relation de *Generalization*.

– constraint : les contraintes définies sur un *GeneralizableElement* général sont transmises au *GeneralizableElement* spécialisé.

Propriétés non héritées

Les propriétés suivantes ne sont pas héritées par un *GeneralizableElement* spécialisé à partir du *GeneralizableElement* général à travers une relation de *Generalization*.

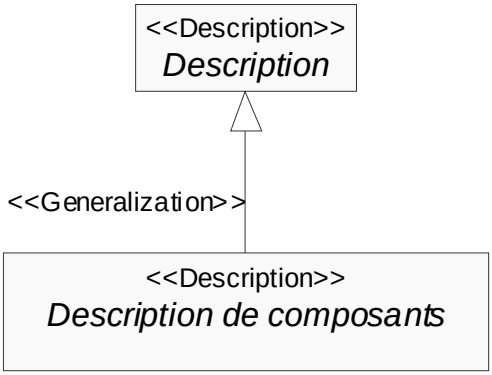


Figure 48. *Exemple d' instanciation de la métarelation de Generalization.*

– name : la valeur de cet attribut n'est pas transmise du *GeneralizableElement* général au *GeneralizableElement* spécialisé. Chaque *ModelElement* a son propre nom. Ainsi dans une relation de *Generalization* les deux classes ne porte pas le même nom (cf. figure 48).

– isAbstract : la valeur de cet attribut n'est pas héritée par une relation de *Generalization*. Si la classe générale est abstraite, alors la classe spécialisée peut ne pas l'être (cf. figure 48).

2.7 Classifier (UML étendu)

Un *Classifier* représente une entité du modèle capable de déclarer (grâce à la relation *Composition*) une collection de *Feature* comme les rubriques (*Item*) ou les scripts (*Script*). *Classifier* hérite de *Namespace* la capacité d'encapsuler des *ModelElements*. *Classifier* est une métaclasse abstraite capable de participer à une relation de *Generalization* (spécialise *GeneralizableElement*). Dans une relation de généralisation entre deux *Classifiers*, l'héritage porte sur les *Features* déclarés avec la relation de *Composition* et les *ModelElements* encapsulés avec la relation *Bundle*.

Associations

– feature : désigne des *Features* comme les *Items* et les *Scripts* possédés par un *Classifier* à travers la relation *Composition*.

– association : dénote une *AssociationEnd* d'une *Association* à laquelle le *Classifier* participe. C'est l'association inverse de l'association *participant* de *AssociationEnd* (cf. figure 49).

– typedFeature : désigne des *StructuralFeatures* dont le type est le *Classifier*.

– typedProperty : désigne des *StructuralProperties* dont le type est le *Classifier*.

– typedParameter : désigne une liste de *Parameters* déclarés par un *BehavioralFeature*.

Contraintes

– chacune des spécialisations concrètes de la métaclasse *Classifier* doit spécifier quels sont les *ModelElement* qu'elle est capable d'encapsuler en utilisant la relation *Bundle*.

2.8 Feature (UML étendu)

C'est une spécialisation de *ModelElement* représentant un attribut déclaré dans un *Classifier*. *M-Sigma* définit deux types de *Features* : *StructuralFeature* et *BehavioralFeature*. La métaclasse *Feature* représente un point d'évolution possible du métamodèle *M-Sigma*.

Attributs

– ownerscope : spécifie s'il existe une seule instance du *Feature* partagée par toutes les instances du *Classifier* ou si chaque instance du *Classifier* possède sa propre instance du *Feature*. Il peut prendre deux valeurs :

- instance : chaque instance du *Classifier* possède sa propre instance du *Feature*.
- classifier : il existe une seule instance du *Feature* partagé par toutes les instances du *Classifier*.

– visibility : spécifie si la propriété *Feature* peut être utilisée par d'autres *Classifiers*. La visibilité des *Classifiers* encapsulés est calculée en adoptant la visibilité la plus restrictive. L'attribut visibility peut prendre les valeurs suivantes :

- public : tout classifier extérieur qui voit le *Classifier* (contenant) par encapsulation (*Bundle*) ou agrégation (*Aggregation*) peut utiliser le *Feature*.
- protected : tout descendant du *Classifier* (contenant) peut utiliser le *Feature*.
- private : uniquement le classifier (contenant) lui même peut utiliser le *Feature*.

Associations

– owner : désigne le *Classifier* déclarant le *Feature*.

2.9 StructuralFeature (UML étendu)

C'est une spécialisation abstraite de la métaclasse *Feature* qui définit le type de propriétés structurelles (exemple *Item*). Une instance de la métaclasse *StructuralFeature* est associée à un type de données décrit par un *Classifier*. La relation de composition (*Composition*) combinée avec la possibilité d'avoir des *StructuralFeatures* de type complexe (*Classifier*) permet de construire des *Classifiers* composites sous la forme d'arbres de *Items* de type simple (mots, chaînes de caractères, etc.) ou avancé (diagrammes UML).

Attributs

– multiplicity : spécifie la cardinalité de l'ensemble des valeurs qui sont attribuées à une propriété *StructuralFeature*. Dans la plupart des cas, une propriété *StructuralFeature* est monovaluée, mais rien n'empêche d'avoir une propriété multivaluée.

– changeability : indique si la valeur d'un *StructuralFeature* peut être modifiée après l'instanciation et l'initialisation du *Classifier*. La *changeability* peut avoir l'une des valeurs suivantes :
  - changeable : pas de restriction sur la modification.
  - frozen : aucune valeur ne peut être ajoutée, modifiée ou supprimée après l'instanciation et l'initialisation du *Classifier* (contenant).
  - addOnly : des valeurs peuvent être ajoutées à tout moment mais aucune valeur ne peut être supprimée après l'instanciation et l'initialisation du *Classifier* (contenant).

– ordering : dans le cas où la *multiplicity* est supérieure à 1, indique si les instances d'une *StructuralFeature* sont ordonnées ou non. L'attribut *ordering* peut avoir l'une des valeurs suivantes :
  - unordered : les instances forment un ensemble non ordonné.
  - instanciationordered : les instances sont ordonnées par ordre d'instanciation.

Associations

– type : désigne le *Classifier* qui représente le type du *StructuralFeature*.

2.10 BehavioralFeature (UML étendu)

C'est une métaclasse abstraite qui spécialise la métaclasse *Feature*. Elle donne la possibilité à un *Classifier* de déclarer des propriétés comportementales comme les *Scripts*.

Associations

– parameter : une liste ordonnée de *Parameters*. Pour l'appel d'un *BehavioralFeature*, l'appelant doit fournir une liste de valeurs compatibles avec les types des *Parameters*.

2.11 Parameter (UML étendu)

C'est une déclaration d'un argument passé ou renvoyé par un *BehavioralFeature*.

Attributs

– defaultValue : désigne la valeur par défaut que doit prendre un paramètre dans le cas où sa valeur n'est pas précisée.

– kind : précise le type de paramètre :
  - in : un paramètre d'entrée.
  - out : un paramètre de sortie.
  - inout : un paramètre d'entrée qui peut être modifié pour retourner une valeur de sortie.
  - return : une valeur de retour renvoyée par un *BehavioralFeature*.

Associations

– type : désigne le *Classifier* qui représente le type du *Parameter*.

2.12 Script (M-Sigma)

C'est un ensemble d'instructions exécutées en cas de violation d'une contrainte (*Constraint*) définie sur le *ModelElement* qui l'encapsule (*Bundle*).

Attributs

– body : une chaîne de caractères représentant le corps du *script*.

Associations

- Scripttrigger : désigne une liste de *Constraints* pouvant déclencher le *Script*.

2.13 Item (M-Sigma)

C'est une spécialisation concrète de la métaclasse *StructuralFeature*. Elle représente une rubrique qui possède un nom (*name*), une visibilité (*visibility*), un type de données représenté par *Classifier* et une valeur initiale (*initialValue*).

Attributs

- initialValue : précise la valeur initiale affectée à la rubrique (*Item*) lors de son instanciation.

2.14 ModelElementProperty (M-Sigma)

C'est une métaclasse abstraite qui déclare une métapropriété identifiée par un nom (*name*) pouvant être associé à un *ModelElement* grâce à l'encapsulation (*Bundle*).

2.15 StructuralProperty (M-Sigma)

C'est une spécialisation abstraite de la métaclasse *ModelElementProperty*. Elle possède un nom (*name*) et un type désigné par un *Classifier*. Elle se spécialise en deux métaclasses concrètes *VariableProperty* et *FixProperty*.

Attributs

- multiplicity : spécifie la cardinalité de l'ensemble des valeurs qui sont attribuées à une propriété *StructuralProperty*. Dans la plupart des cas, une propriété *StructuralProperty* est monovaluée, mais rien n'empêche d'avoir une propriété multivaluée.
- changeability : indique si la valeur d'un *StructuralProperty* peut être modifiée après l'instanciation et l'initialisation du *ModelElement* (contenant). La *changeability* peut avoir l'une des valeurs suivantes :
  - changeable : pas de restriction sur la modification.
  - frozen : aucune valeur ne peut être ajoutée, modifiée ou supprimée après l'instanciation et l'initialisation du *ModelElement* (contenant).
  - addOnly : des valeurs peuvent être ajoutées à tout moment mais aucune valeur ne peut être supprimée après l'instanciation et l'initialisation du *ModelElement* (contenant).
- ordering : dans le cas ou la *multiplicity* est supérieure à 1, indique si les instances d'une *StructuralProperty* sont ordonnées ou non. L'attribut *ordering* peut avoir l'une des valeurs suivantes :
  - unordered : les instances forment un ensemble non ordonné.
  - instanciationordered : les instances sont ordonnées par ordre d'instanciation.
- initialValue : précise la valeur initiale affectée à la *StructuralProperty* lors de son initialisation.

Associations

- type : désigne le *Classifier* qui représente le type du *StructuralProperty*.

2.16 VariableProperty (M-Sigma)

C'est une spécialisation concrète de la métaclasse *StructuralProperty*. Elle désigne une propriété dont la valeur peut être modifiée par l'utilisateur après la première initialisation lors de l'instanciation du *ModelElement* qui la déclare.

2.17 FixProperty (M-Sigma)

C'est une spécialisation concrète de la métaclasse *StructuralProperty*. Elle désigne une propriété dont la valeur est constante et initialisée lors de l'instanciation du *ModelElement* qui la déclare.

3. Le Core package Relationships étendu

Le *Core package Relationships* (cf. figure 49) définit les métarelations sur les métaclasses définies dans le *Core package Classifiers* : la généralisation (*Generalization*), les associations (*Association*), les descriptions d'associations (*AssociationDescription*), etc.

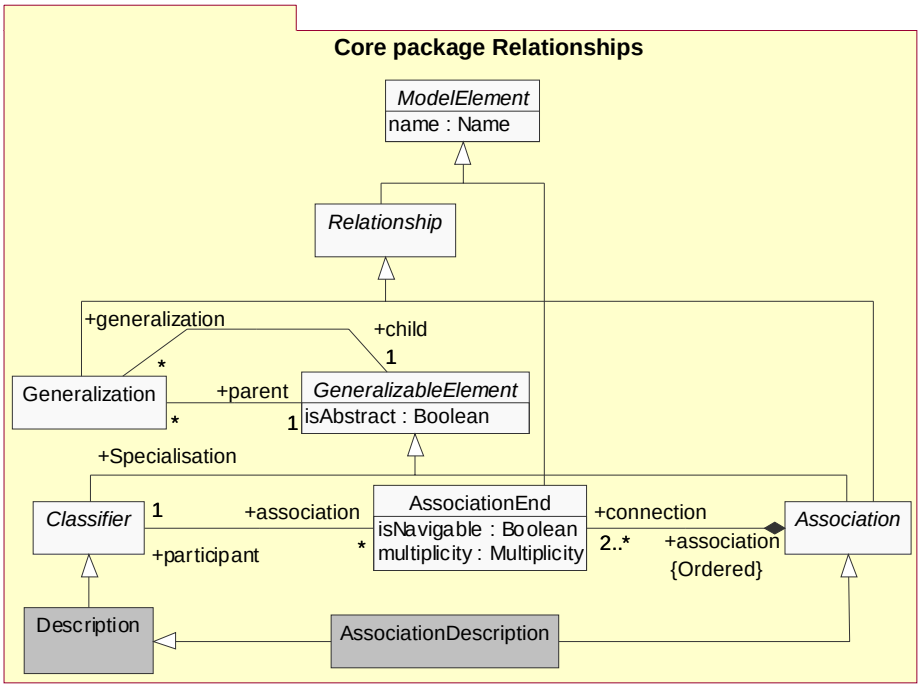


Figure 49. *Core package Relationships.*

3.1 Relationship

C'est la métarelation du plus haut niveau d'abstraction. Une *Relationship* est spécifique à un *Namespace*, ce qui servira par la suite à définir des relations qui ont une portée limitée à chaque base descriptive de composants (*ComponentsRepository*) ou à une *Description*. Par exemple, une *Relationship* définie sur les rubriques (*Item*) d'une *Description* est locale à toutes les instances de la *Description* et elle est héritée par ses sous-descriptions (Generalization) et les descriptions qui la composent (*Composition*).

3.2 Generalization (UML étendu)

C'est une spécialisation de la métaclasse *Relationship*. Elle représente une relation dirigée de généralisation entre deux *GeneralizableElement*.

Associations

- child : désigne le *GeneralizableElement* le plus spécialisé (fils) participant à la *Generalization*.
- parent : désigne le *GeneralizableElement* le plus général (père) participant à la *Generalization*.

3.3 Association (UML étendu)

C'est une métarelation abstraite qui définit une relation sémantique entre des *Classifiers* (ex. des *Descriptions*). Une instance d'une spécialisation concrète d'*Association* possède au moins deux instances de la métaclasse *AssociationEnd*. Un seul *Classifier* peut être connecté à plus d'une métaclasse *AssociationEnd* dans une *Association*.

Associations

- connection : désigne l'ensemble de *AssociationEnds* qui participent à une *Association*.

3.4 AssociationEnd (UML étendu)

C'est une extrémité d'une relation *d'Association*. *AssociationEnd* spécifie la relation entre une *Association* et un *Classifier*. Elle indique les *Classifiers* compatibles avec l'extrémité de *l'Association*. Les *AssociationEnds* d'une *Association* sont ordonnées.

Attributs

- name : hérité de *ModelElement*; spécifie le nom du rôle associé à cette terminaison de l'*Association*.
- isNavigable : indique par un booléen si une association est navigable en partant de l'*AssociationEnd* qui la définit.
- multiplicity : indique le nombre d'instances cibles qui peuvent être associées à une instance source dans une association.

Associations

- participant : désigne le *Classifier* participant à une *Association* à travers l'*AssociationEnd*.
- association : désigne l'*Association* à laquelle participe l'*AssociationEnd*.

3.5 AssociationDescription (M-Sigma)

C'est une spécialisation des deux métaclasses *Association* et *Description*. Elle permet d'étendre la sémantique d'une *Association* en l'autorisant à s'autodécrire avec des rubriques et des sous-descriptions. De plus, elle permet de faire des spécialisations et des compositions d'associations.

Contraintes

- ownedElement : une instance de *AssociationDescription* ne peut encapsuler (*Bundle*) que des instances des métaclasses suivantes ou des instances de leurs spécialisations concrètes : *Constraint*, *StructuralProperty*.

– typedFeature : une instance de *AssociationDescription* ne peut pas être utilisée pour typer une instance de *Feature*.

– typedProperty : une instance de *AssociationDescription* ne peut pas être utilisée pour typer une instance de *StructuralProperty*.

– typedParameter : une instance de *AssociationDescription* ne peut pas être utilisée pour typer une instance de *Parameter*.

4. Le Core package Classifiers étendu

Le *Core package Classifiers* (cf. figure 50) définit les concepts gérés par le métamodèle *M-Sigma* comme par exemple, les descriptions (*Description*), les bibliothèques de composants (*ComponentsRepository*), les composants (*Component*), etc.

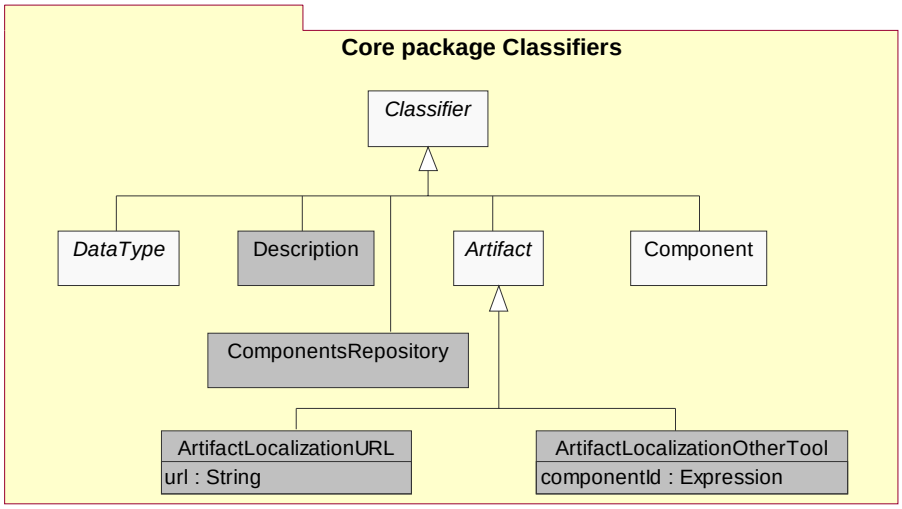


Figure 50. *Core package Classifiers.*

4.1 Description (M-Sigma)

C'est la métaclasse abstraite à partir de laquelle toutes les métaclasses de descriptions *ComponentDescription* (cf. figure 50) et *AssociationDescription* (cf. figure 49) sont spécialisées. Une description est composée (*Composition*) d'un ensemble de rubriques (*Item*) de type simple (entier, chaîne de caractères, etc.) ou avancé (diagramme UML, vocabulaire contrôlé, etc.). Elle peut être composée d'autres sous-descriptions (*Description*) en déclarant des rubriques du type *Description*. Une *Description* peut participer à une relation de *Generalization*. Une description peut définir des contraintes (*Constraint*) sur les rubriques (*Item*), les propriétés (*StructuralProperty*) et les sous-descriptions (*Description*) qu'elle contient. La figure 51 reprend le concept de *description de sources* de composants du modèle *C-Sigma* (cf. chapitre 3 section 3.4.4).

Contraintes

– ownedElement : une instance de *Description* ne peut encapsuler (*Bundle*) que des instances des métaclasses suivantes ou des instances de leurs spécialisations concrètes : *Constraint*, *StructuralProperty*.

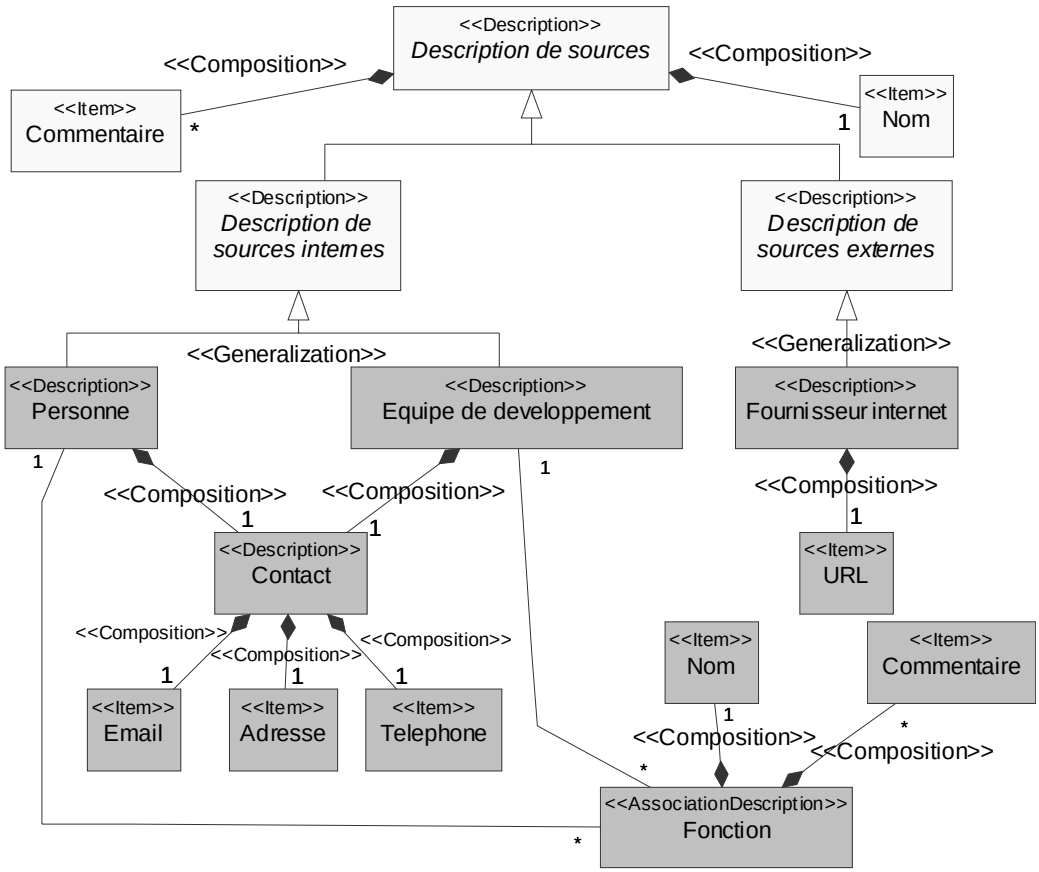


Figure 51. *Les sources de composants en C-Sigma comme instance de M-Sigma.*

4.2 ComponentsRepository (M-Sigma)

Un SGBDC doit pouvoir gérer plusieurs bases de composants. Chaque instance de cette métaclasse représente une base de composants indépendante qui a ses propres descriptions (*Description*), ses propres relations (*Relationship*) et ses propres composants (*Component*). La figure 52 présente une instanciation de la métaclasse *ComponentsRepository* extraite du modèle *C-Sigma* (cf. chapitre 3 section 3.1).

Contraintes

- ownedElement : une instance de *ComponentsRepository* ne peut encapsuler (*Bundle*) que des instances des métaclasses suivantes ou des instances de leurs spécialisations concrètes : *Constraint*, *StruturalProperty*, *Component*, *AssociationDescription*, *Description*.
- typedFeature : une instance de *ComponentsRepository* ne peut pas être utilisée pour typer une instance de *Feature*.
- typedProperty : une instance de *ComponentsRepository* ne peut pas être utilisée pour typer une instance de *StructuralProperty*.
- typedParameter : une instance de *ComponentsRepository* ne peut pas être utilisée pour typer une instance de *Parameter*.

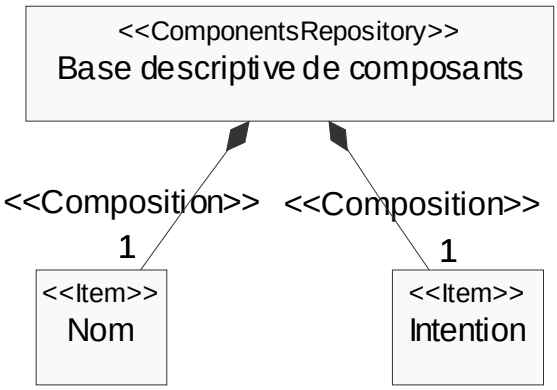


Figure 52. *Les bases descriptives de composants en C-Sigma comme instance de M-Sigma.*

4.3 Component (UML étendu)

Les instances de cette métaclasse référencent les composants gérés par la BDC. Notre SGBDC gère uniquement les références des composants et non pas les composants eux-mêmes. Une référence contient des informations sur l'emplacement physique où on peut récupérer le composant.

Une instance de *Component* encapsule avec la relation *Bundle* une ou plusieurs instances de *Artifact*. Un *Artifact* représente la portion physique d'un composant. La figure 53 présente la modélisation du concept de *composant* avec ses différents niveaux d'abstraction dans le modèle *C-Sigma* (cf. chapitre 3 section 3.2).

Contraintes

- ownedElement : une instance de *Component* ne peut encapsuler (*Bundle*) que des instances des métaclasses suivantes ou des instances de leurs spécialisations concrètes : *Constraint*, *StrutucralProperty*, *Artifact*.
- typedFeature : une instance de *Component* ne peut pas être utilisée pour typer une instance de *Feature*.
- typedProperty : une instance de *Component* ne peut pas être utilisée pour typer une instance de *StructuralProperty*.
- typedParameter : une instance de *Component* ne peut pas être utilisée pour typer une instance de *Parameter*.

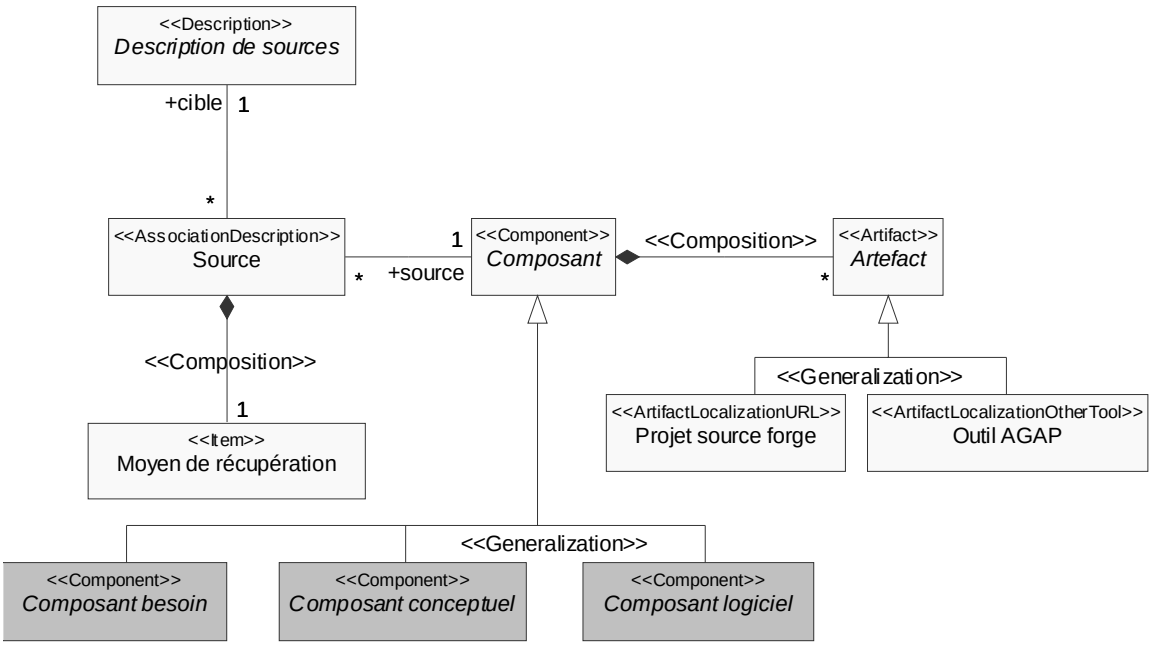


Figure 53. *Les composants et les artefacts en C-Sigma comme instance de M-Sigma.*

4.4 Artifact (UML étendu)

C'est une métaclasse abstraite représentant une portion physique d'un composant. Un *Component* peut encapsuler (Bundle) un ou plusieurs *Artifacts*.

Contraintes

– ownedElement : une instance de *Artifact* ne peut encapsuler (*Bundle*) que des instances des métaclasses suivantes ou des instances de leurs spécialisations concrètes : *Constraint*, *StructuralProperty*.

– typedFeature : une instance de *Artifact* ne peut pas être utilisée pour typer une instance de *Feature*.

– typedProperty : une instance de *Artifact* ne peut pas être utilisée pour typer une instance de *StructuralProperty*.

– typedParameter : une instance de *Artifact* ne peut pas être utilisée pour typer une instance de *Parameter*.

4.5 ArtifactLocalizationOtherTool (M-Sigma)

C'est une spécialisation concrète de *Artifact* qui localise une portion physique d'un composant gérée dans un autre outil de gestion de composants ou un atelier de programmation. La figure 53 présente la classe *Outil AGAP* comme une instance de *ArtifactLocalizationOtherTool*. Une instance de *Outil AGAP* est par exemple un patron géré par l'outil AGAP[9] (Conte, 2001) (Conte, 2001(a)).

Attributs

– name : hérité de *ModelElement* ; indique le nom d'un outil de gestion de composants externe.

[9] Atelier de Gestion et d'Application de Patrons.

– componentId : une chaîne de caractères contenant l'identifiant d'une portion de composant dans un outil extérieur dont le nom est indiqué par *name*.

Contraintes

– ownedElement : une instance de *ArtifactLocalizationOtherTool* ne peut encapsuler (*Bundle*) que des instances des métaclasses suivantes ou des instances de leurs spécialisations concrètes : *Constraint*, *StructuralProperty*.

– typedFeature : une instance de *ArtifactLocalizationOtherTool* ne peut pas être utilisée pour typer une instance de *Feature*.

– typedProperty : une instance de *ArtifactLocalizationOtherTool* ne peut pas être utilisée pour typer une instance de *StructuralProperty*.

– typedParameter : une instance de *ArtifactLocalizationOtherTool* ne peut pas être utilisée pour typer une instance de *Parameter*.

4.6 ArtifactLocalizationURL (M-Sigma)

C'est une spécialisation concrète de *Artifact* qui localise une portion physique d'un composant se trouvant sur le réseau (réseau local, intranet ou internet). Par exemple, *ArtifactLocalizationURL* peut préciser la localisation d'un composant sur le site web d'un fournisseur de composants en ligne. La figure 53 présente la classe *Projet source forge* comme une instance de *ArtifactLocalizationURL*. Une instance de *Projet source forge* désigne un projet open source sur le site Internet www.sourceforge.net.

Attributs

– name : hérité de *ModelElement*; indique le nom d'un emplacement réseau (site internet) source de composants.

– url : une chaîne de caractères contenant l'url où la partie physique d'un composant peut être récupérée.

Contraintes

– ownedElement : une instance de *ArtifactLocalizationURL* ne peut encapsuler (*Bundle*) que des instances des métaclasses suivantes ou des instances de leurs spécialisations concrètes : *Constraint*, *StructuralProperty*.

– typedFeature : une instance de *ArtifactLocalizationURL* ne peut pas être utilisée pour typer une instance de *Feature*.

– typedProperty : une instance de *ArtifactLocalizationURL* ne peut pas être utilisée pour typer une instance de *StructuralProperty*.

– typedParameter : une instance de *ArtifactLocalizationURL* ne peut pas être utilisée pour typer une instance de *Parameter*.

4.7 DataType (UML étendu)

Un *DataType* représente un type de données pouvant être pris par une propriété (*StructuralProperty*) ou une rubrique (*Item*). Un type de données peut être du type Primitive (entier, chaîne de caractères,…), Enumération (énumération booléenne {« vrai », « faux »}, etc.), ou Advanced (Diagramme UML, schéma, etc.). Les différents types de données définis dans *M-Sigma* sont détaillés dans la section 6.

Contraintes

– feature : une instance d'une spécialisation concrète de *DataType* ne peut pas définir des *Features*.

– ownedElement : une instance d'une spécialisation concrète de *DataType* ne peut pas encapsuler (*Bundle*) des *ModelElements*.

– association : une instance d'une spécialisation concrète de *DataType* ne peut pas être un membre d'une *Association*.

5. Le Core package Auxiliary elements étendu

Le *Core package Auxiliary elements* (cf. figure 54) définit des métaclasses additionnelles pour étendre le *Core package Backbone*. Il définit une infrastructure pour les éléments visuels.

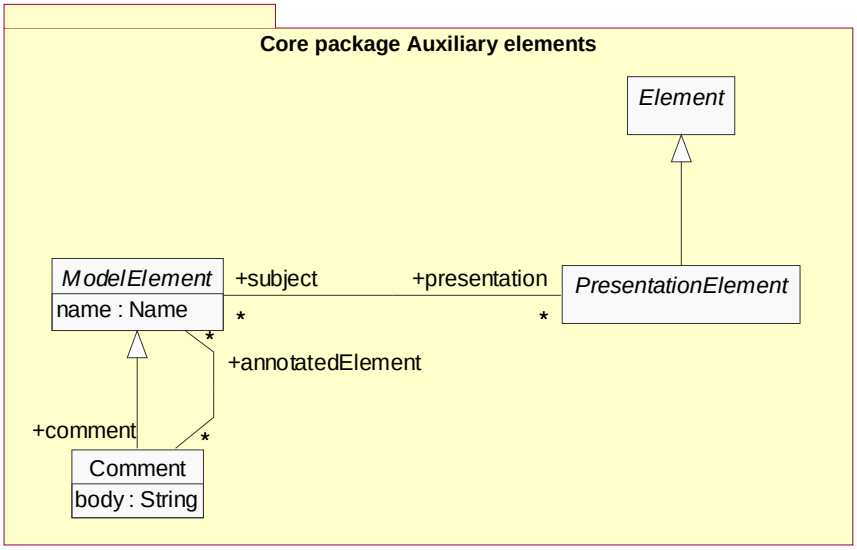


Figure 54. *Core package Auxiliary elements.*

5.1 PresentationElement

C'est un élément de présentation textuel ou graphique d'un ou plusieurs *ModelElement*. *PresentationElement* est une métaclasse abstraite, base de toutes les métaclasses utilisées pour la présentation.

Associations

– subject : désigne l'ensemble de *ModelElements* qu'un *PresentationElement* peut représenter.

5.2 Comment

C'est une annotation attachée à un *ModelElement* ou à un ensemble de *ModelElement*s. Un *Comment* peut contenir un ensemble d'indications écrites par le gestionnaire de BDC à l'intention de l'ingénieur de composants qui va instancier les descriptions de composants.

Attributs

– body : une chaîne de caractères représentant le commentaire.

Associations

– annotatedElement : désigne un *ModelElement* ou un ensemble de *ModelElements* annotés par le commentaire.

Contraintes

– ownedElement : une instance de *Comment* ne peut encapsuler (*Bundle*) d'autres instances de *ModelElement*.

– comment : une instance de *Comment* ne peut être annotée.

6. Le package Data Type

Le *package Data Type* (cf. figure 55) est un nouveau paquetage introduit spécifiquement dans *M-Sigma*. Il définit les types de données utilisés dans le métamodèle *M-Sigma* pour la définition des types de rubriques (*Item*), des types des propriétés (*StructuralProperty*) et des types de paramètres. Ce paquetage intègre des métaclasses issues du *Core package Classifiers* original de UML 1.5 : *Enumeration*, *Primitive*, *ProgrammingLanquageDataType*, etc. Nous présentons dans cette section uniquement les métaclasses introduites par *M-Sigma*.

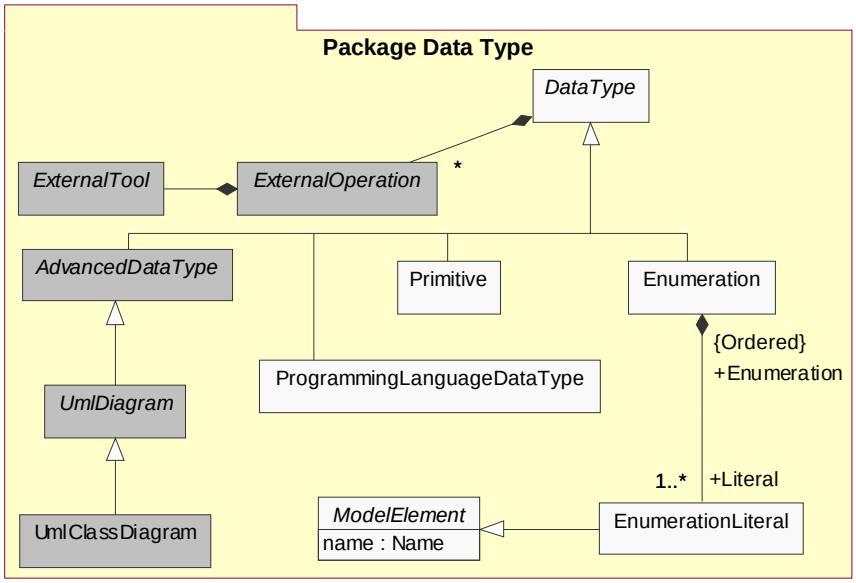


Figure 55. *Data type package.*

6.1 AdvancedDataType (M-Sigma)

C'est une métaclasse abstraite qui sert à introduire de nouveaux types de données comme les diagrammes UML (*UmlDiagram*, *UmlClassDiagram*) et d'autres types qui sont utilisés lors de la description des composants (spécifications formelles en B ou ACME, etc.).

Contraintes

– feature : une instance d'une spécialisation concrète de *AdvancedDataType* ne peut pas définir des *Features*.

– ownedElement : une instance d'une spécialisation concrète de *AdvancedDataType* ne peut pas encapsuler (*Bundle*) des *ModelElements*.

– association : une instance d'une spécialisation concrète de *AdvancedDataType* ne peut pas être membre d'une *Association*.

6.2 ExternalOperation (M-Sigma)

Cette métaclasse permet à l'administrateur du système de définir des opérations spécifiques aux types de données gérées par le métaoutil à travers l'utilisation de services offerts par des

outils externes. Un exemple d'opération peut être l'édition dans un AGL UML d'un diagramme de classes UML.

6.3 ExternalTool (M-Sigma)

C'est une interface qui représente un outil externe qui fournit des services au métaoutil, par exemple l'AGL Rational Rose[10] pour l'édition des diagrammes UML.

7. Le Package Instances

Le métamodèle *M-Sigma* est formé par 6 paquetages dont les 5 premiers (*Core package Backbone*, *Core package Classifiers*, *Core package Relationships*, *Core package auxiliary elements* et *package Data Types*) permettent l'instanciation de modèles de bases descriptives de composants (MBDC) comme le modèle *C-Sigma*. Le métamodèle *M-Sigma* réutilise des paquetages du métamodèle UML et a hérité de son pouvoir descriptif, mais il ne permet pas de représenter les instances des instances des métaclasses *Description*, *AssociationDescription*, etc. La réalisation du SGBDC nécessite l'ajout d'un sixième paquetage : le *Package Instances* (cf. figure 56) qui permet d'instancier les MBDC instanciés à partir du métamodèle *M-Sigma*. À chaque métaclasse du métamodèle *M-Sigma* (spécialisation de *ModelElement*) est associée une instance (*Instance-ModelElement*).

Un MBDC est représenté dans le SGBDC comme un graphe d'instances des spécialisations concrètes de la métaclasse *ModelElement*. Une BDC est un ensemble d'instances de la métaclasse *Instance-ModelElement*. Chaque instance de *ModelElement* est responsable de la gestion des instances de la classe *Instance-ModelElement*. Lors de l'instanciation ou de la modification d'une *Instance-ModelElement*, les règles d'intégrité et les contraintes définies dans le MBDC sont vérifiées.

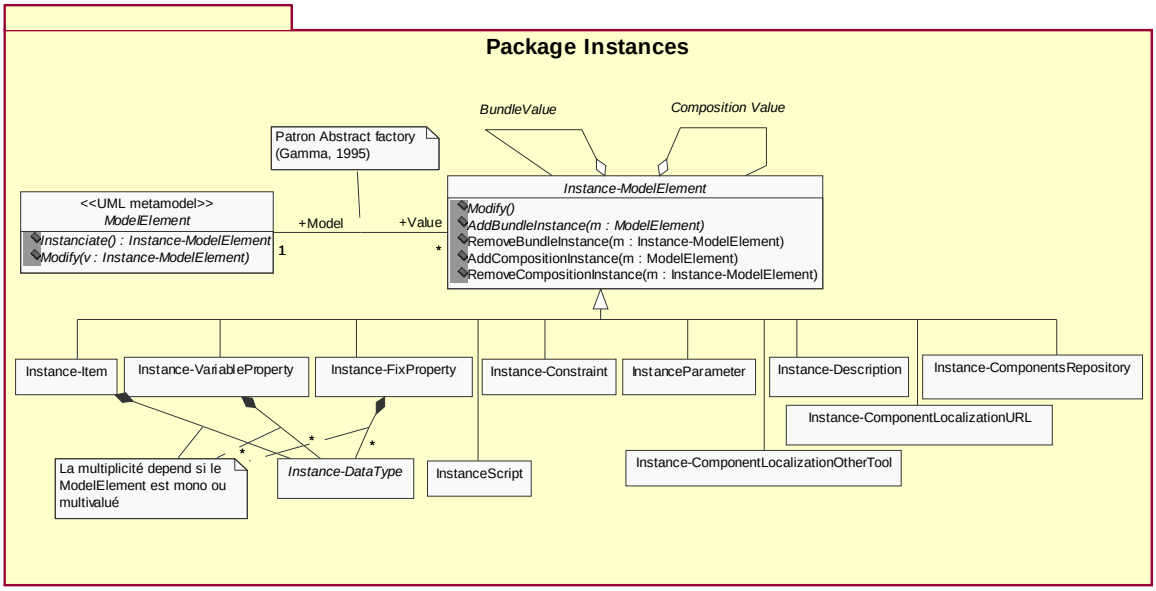


Figure 56. *Package Instances.*

[10] http://www-306.ibm.com/software/rational/

8. Exemples

Nous présentons dans cette section un exemple montrant comment le modèle *C-Sigma* instance du métamodèle *M-Sigma* décrit les composants du type patron écrit avec le formalisme *P-Sigma* (cf. annexe A). Les références au métamodèle *M-Sigma* sont indiquées par les stéréotypes des classes et les relations.

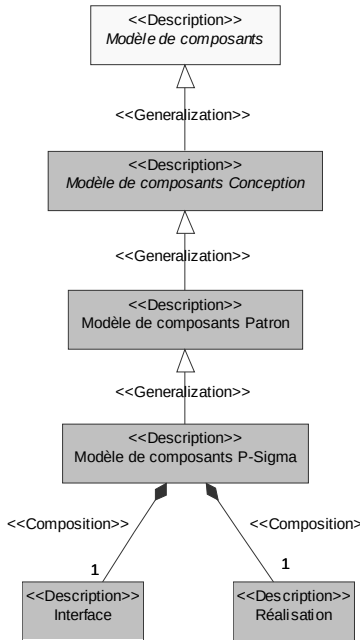


Figure 57. *Le modèle de composants P-Sigma en C-Sigma.*

La description *modèle de composants P-Sigma* est une spécialisation de la description *Modèle de composants Patron* (cf. figure 57) elle-même spécialisation de la description *Modèle de composants conception*. La solution décrite par un patron *P-Sigma* est divisée en deux sous-descriptions *Interface* (cf. figure 58) et *Réalisation* (cf. figure 59).

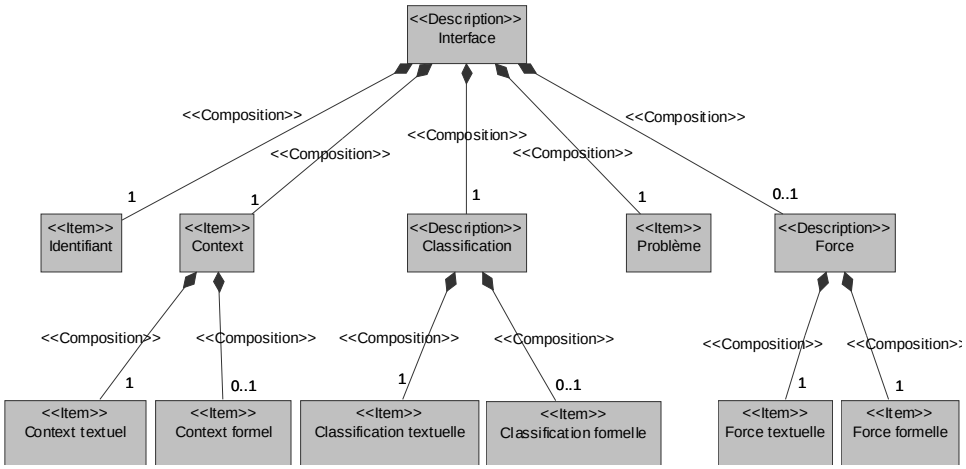


Figure 58. *Partie interface du modèle de composants P-Sigma en C-Sigma.*

La description *Interface* distingue en particulier des *items* textuels et des *items* formels pour décrire le contexte d'application d'un patron, sa classification et sa force. La description *Réalisation* met en évidence que dans le *Modèle de composants P-Sigma*, deux types de solutions sont proposés : *Solution produit* et *Solution démarche*. Un cas d'application peut completer la définition d'un patron.

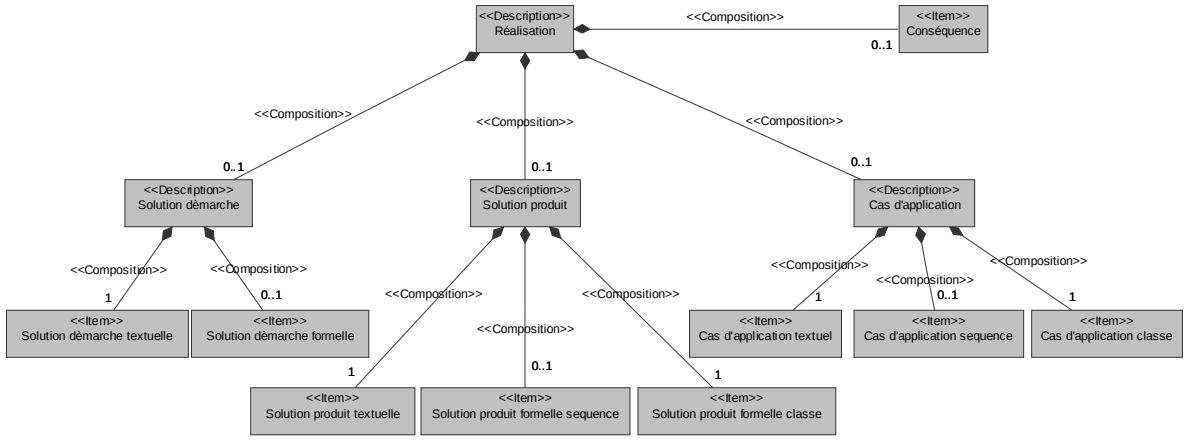


Figure 59. *Partie réalisation du modèle de composants P-Sigma en C-Sigma.*

Le *modèle de composants P-Sigma* définit quatre relations horizontales : *Utilise P-Sigma*, *Requiert P-Sigma*, *Raffine P-Sigma* et *Alternative P-Sigma* (cf. figure 60).

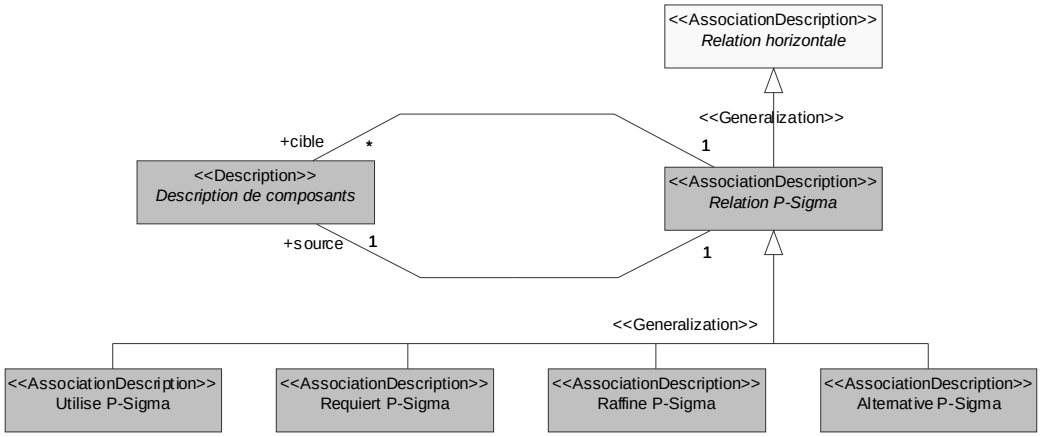


Figure 60. *Relations du modèle de composants P-Sigma en C-Sigma.*

9. Conclusion

Le métamodèle *M-Sigma* est conçu pour permettre l'implantation d'un système de gestion de bases descriptives de composants évolutif capable de gérer différentes bases descriptives de composants. Grâce aux concepts de rubrique et de description introduit dans *M-Sigma*, il est possible de découpler les techniques de recherche de composants de la représentation des composants dans les BDC. Ce résultat permet d'appliquer plusieurs TRC sur une même représentation des composants. De plus, il est possible d'ajouter de nouveaux types de données (*DataType*) avec leurs techniques de recherche dédiées, ce qui garantit une évolution des bases de composants en terme de puissance descriptive et en terme de techniques de recherche de composants.

Le métamodèle *M-Sigma* est conçu avec des points d'évolution permettant son enrichissement avec de nouvelles métaclasses sans remettre en cause ses instanciations existantes. Par exemple, il est possible d'ajouter de nouveaux types de *Feature* (cf. section 2.8) et de nouveaux concepts sous la forme d'une spécialisation concrète de *Classifier* ou de *ModelElement*. Le métamodèle *M-Sigma* satisfait donc les objectifs d'évolutivité et de découplage entre les TRC et la représentation des composants que nous nous sommes fixés dans le chapitre problématique.

Les principes d'évolution d'une BDC sont décrits dans le chapitre suivant sous forme d'un système de patrons.

VI. Évolution d'une base descriptive de composants

Comme nous l'avons précisé dans le chapitre IV, la base *B-Sigma* est un système d'information qui est appelé à évoluer avec les besoins utilisateurs. Nous avons anticipé certains besoins d'évolution lors de la conception du modèle *C-Sigma* en prévoyant des points d'évolution notamment dans la partie abstraite du modèle. Nous proposons dans ce chapitre un système de patrons offrant à l'ingénieur de modèles de bases descriptives de composants (*ingénieur de MBDC*) des fragments de démarches pour le guider dans l'exploitation des points d'évolution prévus dans le modèle *C-Sigma*. Ces patrons sont décrits en utilisant le formalisme *P-Sigma* présenté dans l'annexe A.

1. Le système de patrons

Le système de patrons que nous présentons dans cette section est composé de 11 patrons : 7 patrons processus et 4 patrons produit. La figure 61 présente une cartographie décrivant les relations qui existent entre les différents patrons. Le patron « Évolution de *C-Sigma* » est le point d'entrée du système mais, rien n'empêche l'*ingénieur de MBDC* d'utiliser les solutions offertes par les autres patrons directement sans passer par le point d'entrée. Le système de patrons est en fait l'union de deux sous-systèmes de patrons : le premier s'occupe de l'ajout de nouvelles entités au modèle *C-Sigma* alors que le deuxième permet de définir un mécanisme de classification des instances des entités du modèle. Nous nous limitons à présenter ce système de patrons uniquement par la description de ses patrons ; ceux-ci doivent être auto-explicatifs pour être utilisables.

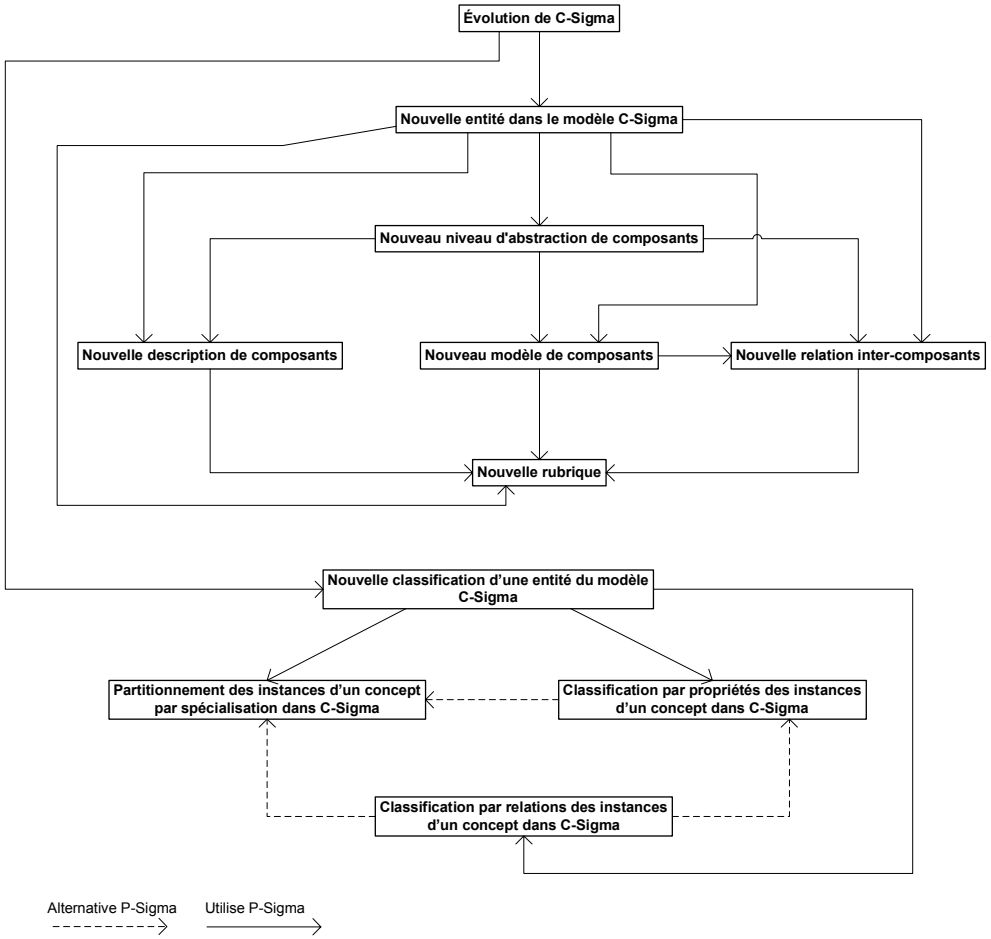
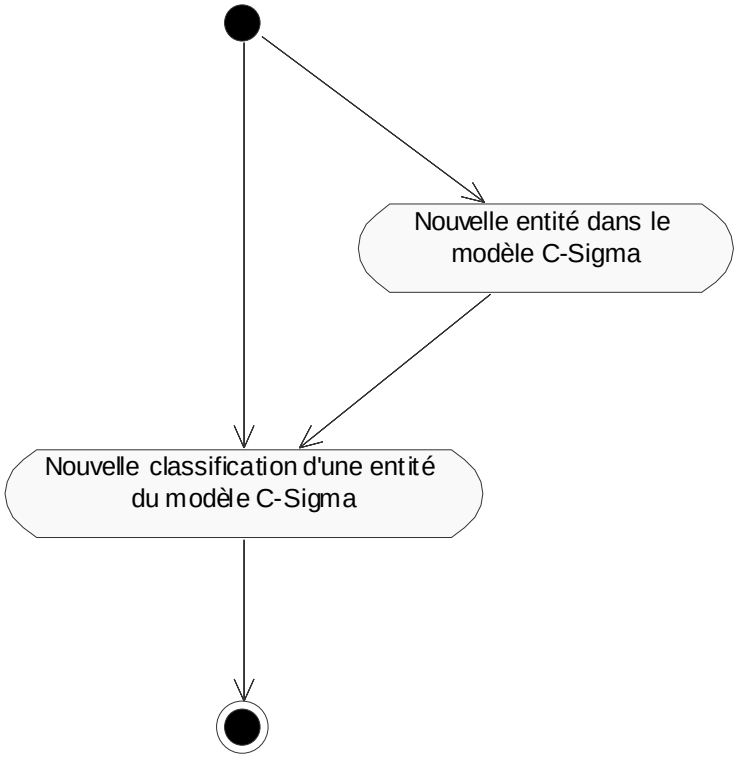


Figure 61. *Cartographie du système de patrons pour l'évolution du modèle C-Sigma.*

1.1 Évolution de *C-Sigma*.

Identifiant	Évolution du modèle *C-Sigma*.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron processus vise à guider l'*ingénieur de MBDC* pour faire évoluer le modèle *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *B-Sigma*.

Solution démarche	
Utilise	« Nouvelle entité dans le modèle *C-Sigma*», « Nouvelle classification d'une entité du modèle *C-Sigma* ».

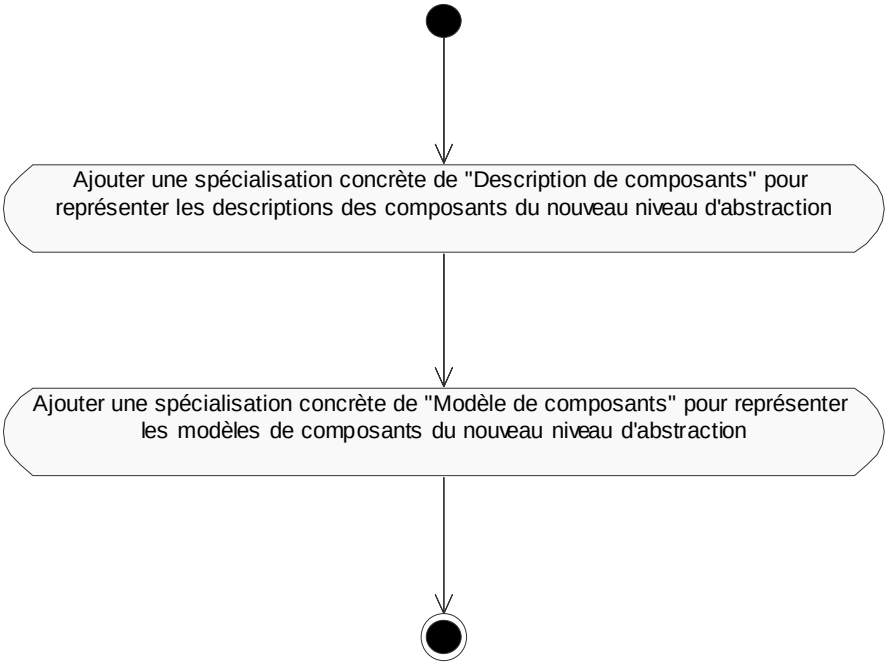
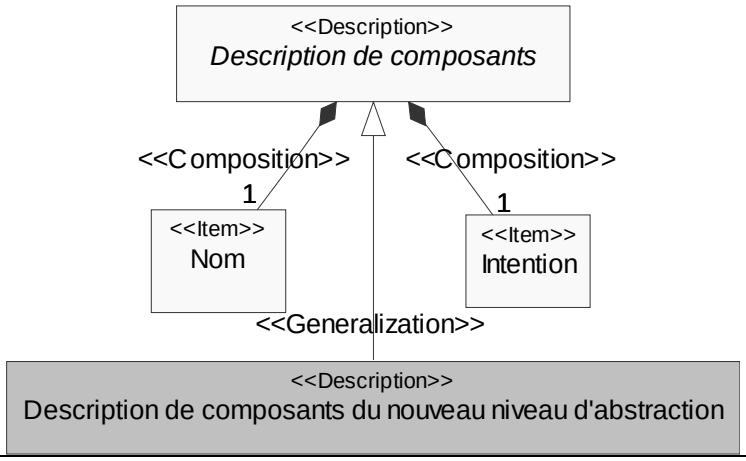
1.2 Nouvelle entité dans le modèle *C-Sigma*.

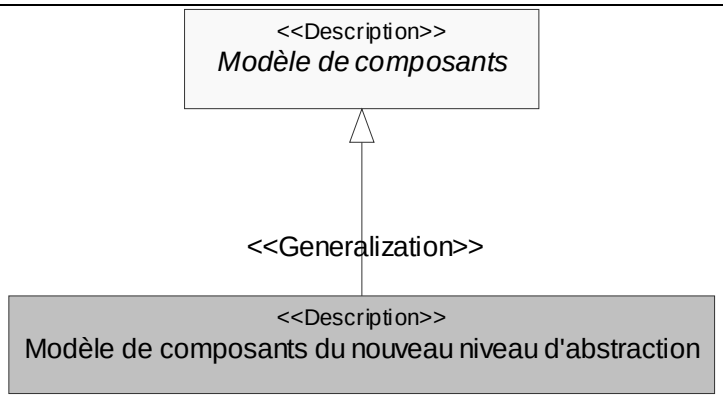
Identifiant	Nouvelle entité dans le modèle *C-Sigma*.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron processus vise à guider l'*ingénieur de MBDC* pour ajouter une nouvelle entité à un modèle *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *B-Sigma*.
Solution démarche	Le tableau suivant donne le nom du patron à appliquer selon l'entité à ajouter par l'*ingénieur de MBDC*.

Type d'entité	Patron à appliquer
Niveau d'abstraction de composants	Nouveau Niveau d'abstraction de composants
Description de composants	Nouvelle description de composants
Modèle de composants	Nouveau modèle de composants
Relation inter-composants	Nouvelle relation inter-composants
Rubrique	Nouvelle rubrique

Utilise	« Nouveau Niveau d'abstraction de composants », « Nouvelle description de composants », « Nouveau modèle de composants », « Nouvelle relation inter-composants », « Nouvelle rubrique ».

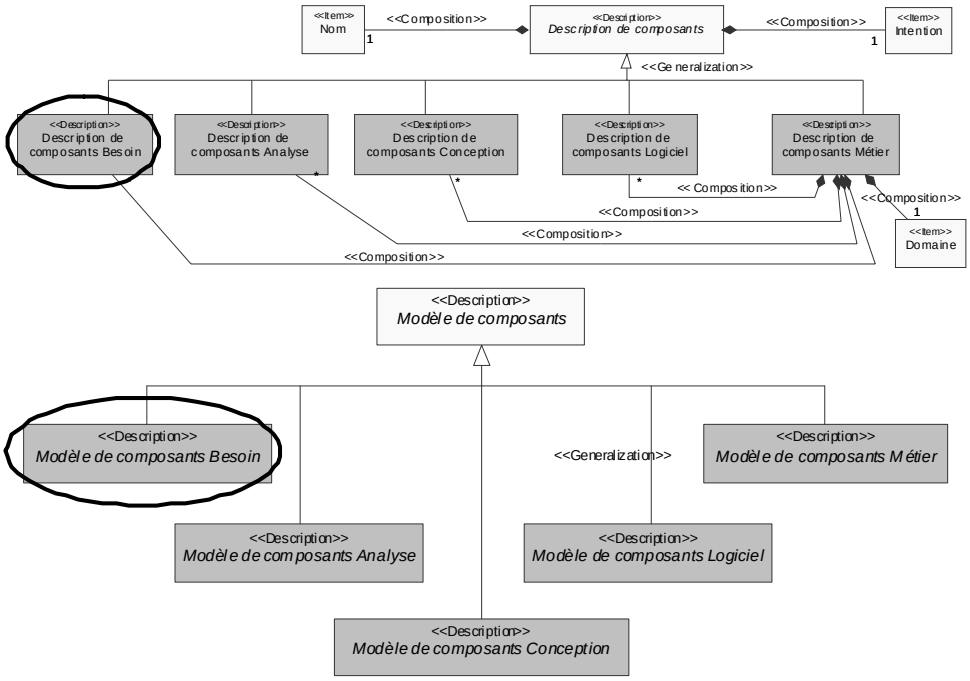
1.3 Nouveau niveau d'abstraction de composants

Identifiant	Nouveau niveau d'abstraction de composants.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron processus vise à guider l'administrateur d'une base *B-Sigma* pour ajouter un nouveau niveau d'abstraction de composants au niveau du modèle *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *B-Sigma*.
Solution démarche	
Solution produit	



Cas d'application

Ce cas d'application présente la nouvelle version du modèle *C-Sigma* supportant les composants besoin.



Conséquences d'application

L'application de ce patron ajoute un nouveau niveau d'abstraction à la base *B-Sigma*. L'ajout se fait par une spécialisation concrète des descriptions abstraites *Description de composants* et *Modèle de composants* donc il n'affecte en rien la cohérence des données (instances des descriptions) gérées dans la base.

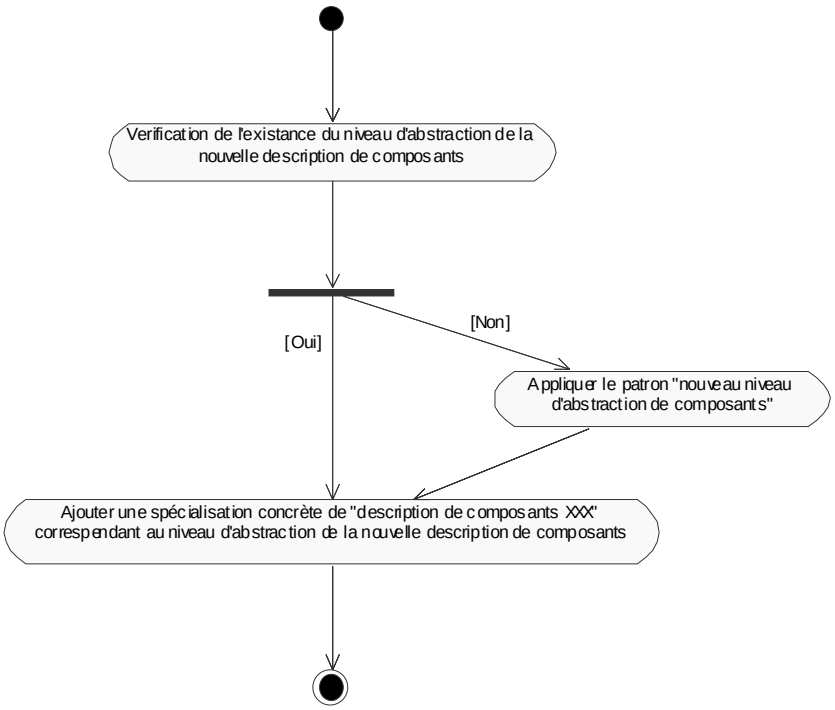
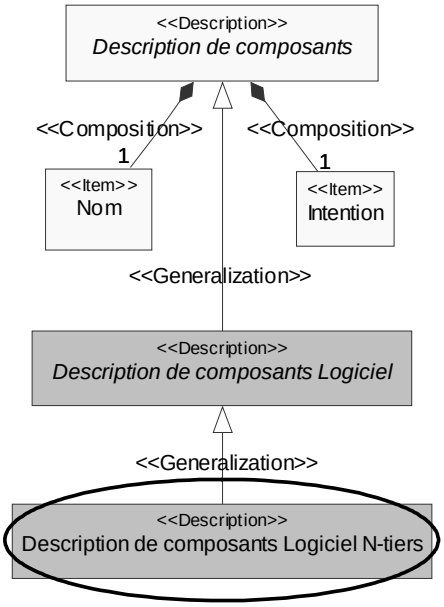
Si les composants du nouveau niveau d'abstraction possèdent des propriétés spécifiques qui nécessitent l'ajout de nouvelles rubriques pour les décrire, il est nécessaire d'appliquer le patron « Nouvelle rubrique ».

Les composants d'un niveau d'abstraction possèdent généralement leurs propres descriptions de composants, modèles de composants et relations inter-composants. Si tel est le cas pour le nouveau niveau d'abstraction qui a été ajouté, alors il faut appliquer les patrons « Nouvelle description de composants », « Nouveau modèle de composants » et « Nouvelle relation »

Utilise

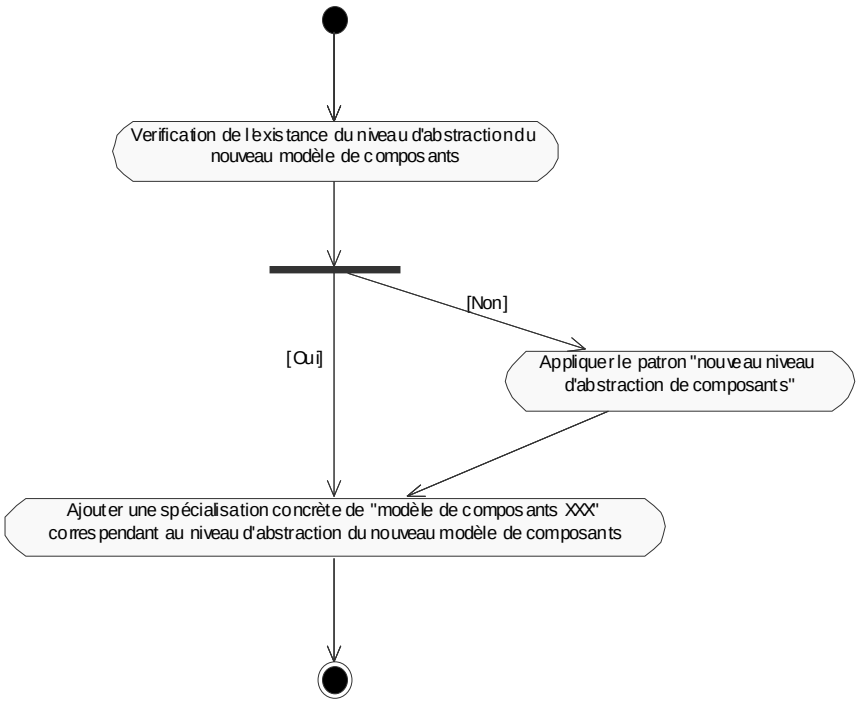
« Nouvelle rubrique », « Nouvelle description de composants », « Nouveau modèle de composants » et « Nouvelle relation inter-composants»

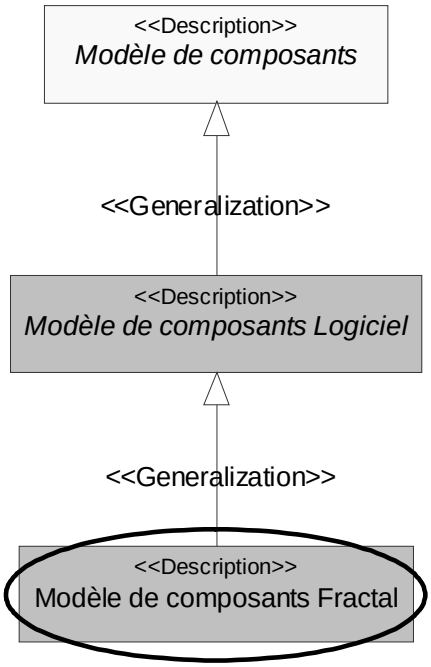
1.4 Nouvelle description de composants

Identifiant	Nouvelle description de composants.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron processus vise à guider l'administrateur d'une base *B-Sigma* pour ajouter une nouvelle description de composants au modèle *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *B-Sigma*.
Solution démarche	
Cas d'application	Ce cas d'application présente la nouvelle version du modèle *C-Sigma* intégrant une description de composants spécifique aux composants logiciels supportant l'architecture N-tiers. 

Conséquences d'application	L'application de ce patron ajoute une nouvelle description de composants. La nouvelle description de composants possède des caractéristiques spécifiques qui nécessitent l'ajout de nouvelles rubriques pour les décrire, d'où le besoin de faire appel au patron « Nouvelle rubrique ».
Utilise	« Nouvelle rubrique »

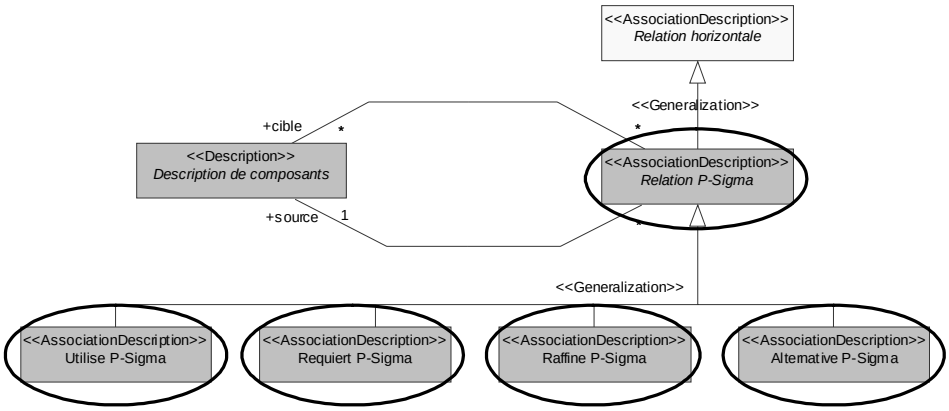
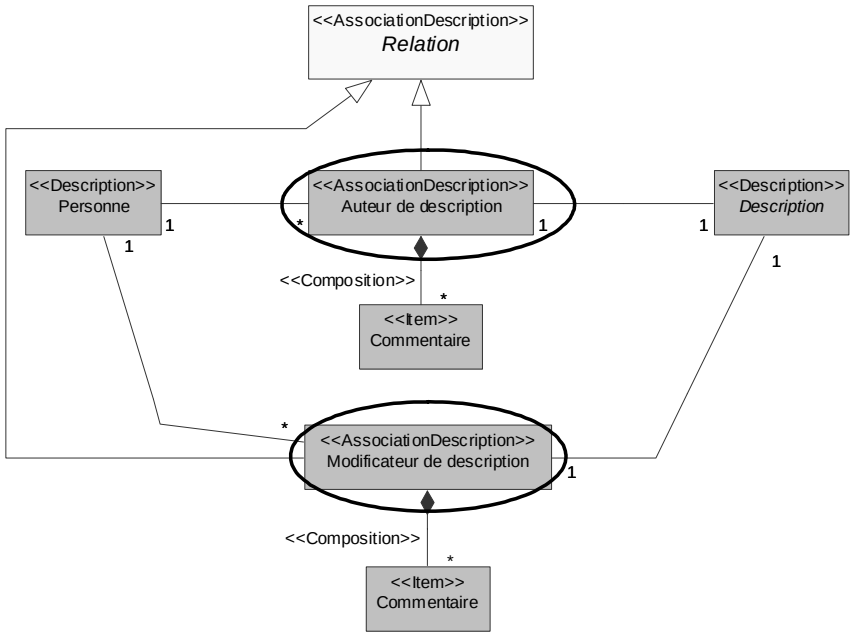
1.5 Nouveau modèle de composants

Identifiant	Nouveau modèle de composants.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron processus vise à guider l'administrateur d'une base *B-Sigma* pour ajouter un nouveau modèle de composants au modèle *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *B-Sigma*.
Solution démarche	

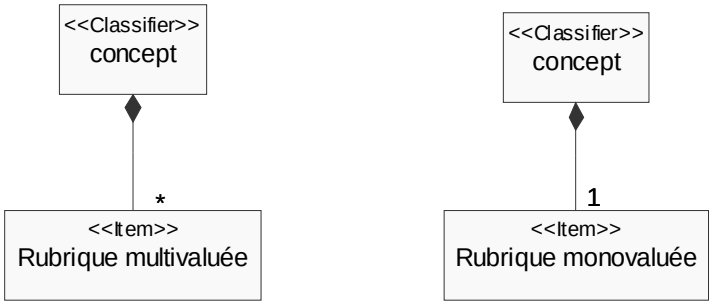
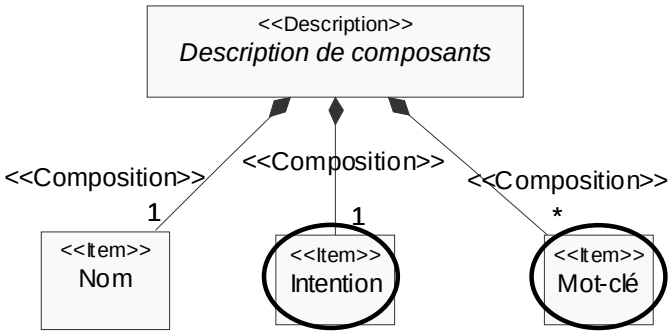
Cas d'application	Ce cas d'application présente la nouvelle version du modèle *C-Sigma* intégrant le modèle de composants Fractal (Bruneton, 2004). 
Conséquences d'application	L'application de ce patron ajoute un nouveau modèle de composants. Le nouveau modèle de composants possède des caractéristiques spécifiques qui nécessitent l'ajout de nouvelles rubriques pour les décrire, d'où le besoin de faire appel au patron « Nouvelle rubrique ». Un modèle de composants définit également ses propres relations inter-composants, d'où le besoin de faire appel au patron « Nouvelle relation inter-composants»
Utilise	« Nouvelle rubrique » et « Nouvelle relation inter-composants».

1.6 Nouvelle relation inter-composants

Identifiant	Nouvelle relation inter-composants.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron processus vise à guider l'administrateur d'une base *B-Sigma* pour l'ajout d'une nouvelle relation inter-composants au niveau du modèle *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *C-Sigma*.
Solution démarche	1) Consulter la hiérarchie d'héritage des relations du niveau concret déjà déclarées dans le modèle *C-Sigma* et essayer d'ajouter la nouvelle relation comme la spécialisation d'une relation existante. 2) Si la nouvelle relation n'est pas une spécialisation d'une relation du niveau concret alors consulter le niveau abstrait et réitérer la première phase en ajoutant la nouvelle relation au niveau concret. 3) Si la nouvelle relation n'est ni une spécialisation d'une relation du niveau concret ni du niveau abstrait (ni horizontale et ni verticale) alors il faut l'ajouter au niveau concret comme une spécialisation de la classe abstraite

	Relation. Pour les trois cas de figures précédents, la nouvelle relation peut avoir besoin de rubriques pour décrire sa propre sémantique. Dans ce cas de figure il faut appliquer le patron « Nouvelle rubrique »
Cas d'application	Le cas d'application suivant présente l'ajout des quatre relations définies dans le formalisme de patrons *P-Sigma*. Le patron « Nouvelle relation inter-composants » est imité à cinq reprises. L'ajout de la relation abstraite *relation P-Sigma* correspond au cas de figure 1 de la solution démarche. L'ajout des relations *utilise P-Sigma*, *requiert P-Sigma* , *raffine P-Sigma* et *alternative P-Sigma* correspond au cas de figure 2 de la solution démarche.  Le cas d'application suivant présente l'ajout des deux relations *Auteur de description* et *modificateur de description*. Ces deux relations assurent la traçabilité lors de la création et de la modification des descriptions d'une base *B-Sigma*. L'ajout de ces deux relations correspond au cas de figure 3 de la solution démarche. 
Conséquences d'application	L'application de ce patron ajoute une nouvelle relation au modèle *C-Sigma*. Si la nouvelle relation nécessite des rubriques pour la description de sa sémantique, alors il faut appliquer le patron « Nouvelle rubrique ».
Utilise	« Nouvelle rubrique ».

1.7 Nouvelle rubrique.

Identifiant	Nouvelle rubrique dans C-Sigma.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron produit vise à guider l'administrateur d'une base *B-Sigma* pour ajouter une rubrique à un *Classifier* (*Description*, *AssociationDescription*, etc.).
Solution produit	
Cas d'application	Deux applications de ce patron sont données : l'une pour ajouter une *intention* à la *Description de composants*, l'autre pour ajouter des *mots-clés* associés à une *Description de composants*. 

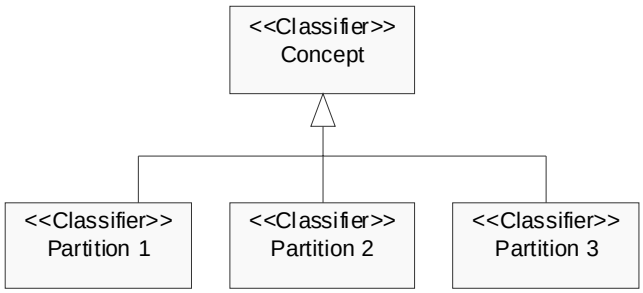
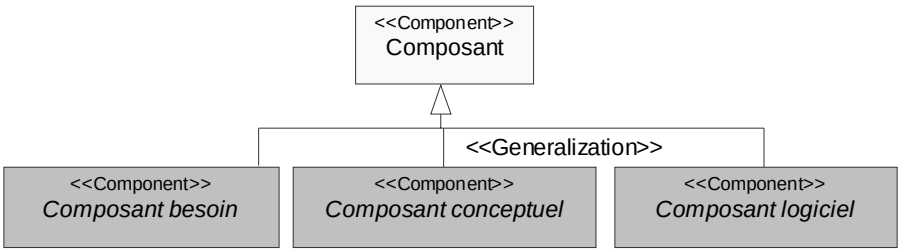
1.8 Nouvelle classification d'une entité du modèle *C-Sigma*.

<table>
<tr><td>Identifiant</td><td>Nouvelle classification d'une entité du modèle C-Sigma.</td></tr>
<tr><td>Contexte</td><td>Ce patron ne nécessite aucun autre patron pour être appliqué.</td></tr>
<tr><td>Problème</td><td>Ce patron processus vise à guider l'ingénieur de MBDC pour ajouter une nouvelle classification d'une entité du modèle C-Sigma.</td></tr>
<tr><td>Forces</td><td>Ce patron est applicable sur le modèle C-Sigma même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base B-Sigma.</td></tr>
<tr><td>Solution démarche</td><td>Le tableau suivant donne le nom du patron à appliquer selon le type de classification et le type d'entité que l'ingénieur de MBDC a besoin de classer.
<table>
<tr><th>Type de classification</th><th>Type d'entité</th><th>Patron à appliquer</th></tr>
<tr><td>Partitionnement</td><td>Classifier</td><td>Partitionnement des instances d'un concept par spécialisation dans C-Sigma.</td></tr>
<tr><td rowspan="2">Partitionnement ou Classification</td><td>ModelElement</td><td>Classification par propriétés des instances d'un concept dans C-Sigma.</td></tr>
<tr><td>Classifier</td><td>Classification par relations des instances d'un concept dans C-Sigma.</td></tr>
</table></td></tr>
<tr><td>Utilise</td><td>« Partitionnement des instances d'un concept dans C-Sigma », « Classification par propriétés des instances d'un concept dans C-Sigma », « Classification par relations des instances d'un concept dans C-Sigma ».</td></tr>
</table>

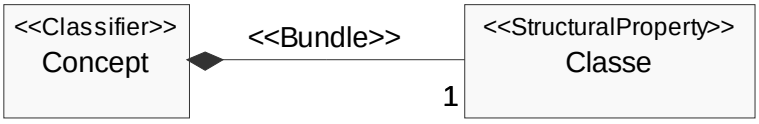
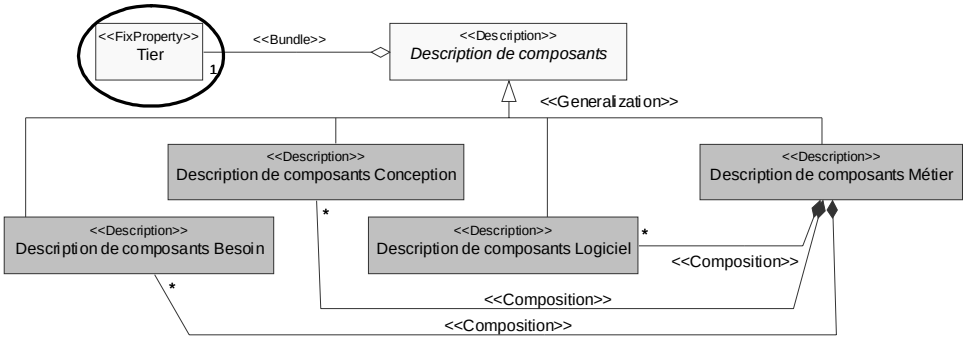
1.9 Partitionnement par spécialisation des instances d'un concept dans C-Sigma.

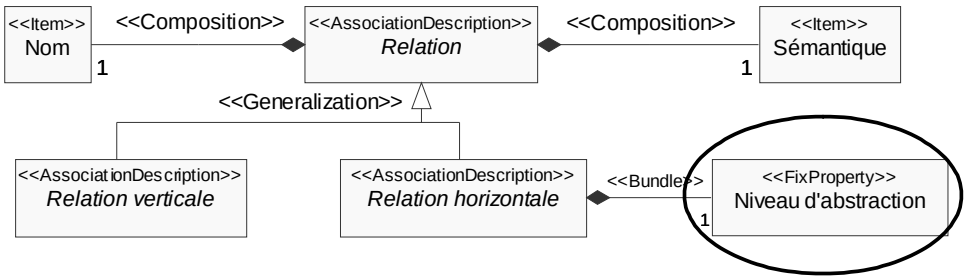
<table>
<tr><td>Identifiant</td><td>Partitionnement par spécialisation des instances d'un concept dans C-Sigma.</td></tr>
<tr><td>Contexte</td><td>Ce patron ne nécessite aucun autre patron pour être appliqué.</td></tr>
<tr><td>Problème</td><td>Ce patron produit vise à guider l'ingénieur MBDC pour établir un partitionnement des instances d'un concept dans C-Sigma.</td></tr>
<tr><td>Forces</td><td>Ce patron est applicable uniquement lors de la conception d'un modèle de base descriptive de composants ou lors de l'introduction d'un nouveau concept dans le modèle C-Sigma déjà existante. La solution de ce patron ne s'applique que sur des entités de modélisation capables de participer à une relation de Generalization, ce qui revient à dire sur toute instance d'une spécialisation concrète de la métaclasse Classifier (comme Description, AssociationDescription, Component, ComponentsRepository, etc.).

De plus la classification ne peut porter que sur un concept à la fois. Par exemple, on ne peut pas classer une instance de Description et une instance de Component dans la même classe.</td></tr>
</table>

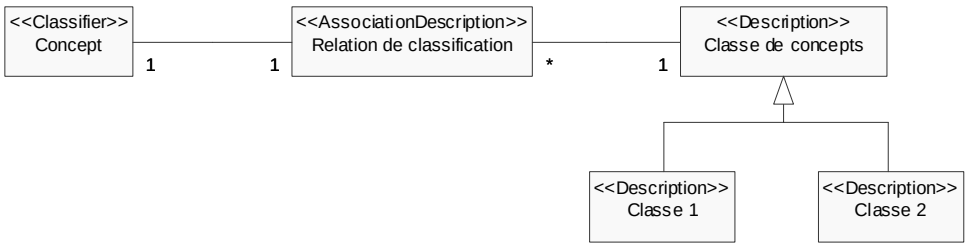
Solution produit	Dans une partition, un élément ne peut appartenir qu'à une seule classe. Ceci peut être parfaitement modélisé en utilisant la relation de *Generalization* définie dans *M-Sigma*. 
Cas d'application	Cet exemple est tiré du modèle *C-Sigma*. En effet un composant ne peut appartenir simultanément à deux niveaux d'abstraction. 
Conséquences d'application	L'application de ce patron fige la classification des concepts partitionnés avec la relation *Generalization*. Cette approche de modélisation a l'avantage de pouvoir définir des rubriques au niveau de chaque partition qui seront héritées par ses membres.

1.10 Classification par propriétés des instances d'un concept dans C-Sigma.

Identifiant	Classification par propriété des instances d'un concept dans C-Sigma.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron produit vise à guider l'administrateur d'une base *B-Sigma* pour établir une classification des instances d'un concept dans *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *C-Sigma*. La solution de ce patron est applicable sur toutes les métaclasses définies dans le métamodèle *M-Sigma*. Une classe peut contenir des instances de différentes métaclasses du métamodèle *M-Sigma* (*Description*, *Item*, etc.).
Solution produit	Si nous voulons classer les instances d'un concept *C-Sigma* tel qu'un concept peut appartenir à plusieurs classes simultanément alors il faut que la propriété soit mutivaluée. Sinon, dans le cas où nous voulons avoir un partitionnement de l'ensemble des instances, il faut que la propriété soit monovaluée. Si nous voulons que la classification ou le partitionnement soit statique, alors nous utilisons les propriétés *FixProperty*, sinon nous utilisons les propriétés *VariableProperty* 
Cas d'application	Dans le cadre de l'architecture N-tiers (Buschmann, 1996) un composant peut appartenir à la couche présentation (interface utilisateur), une des couches application (traitement des données) ou des couches de stockage de données (bases de données). Nous déclarons la propriété monovaluée *Tier* dans la *Description de composants* qui indiquera la couche à laquelle appartient le composant qu'elle décrit. Il s'agira donc d'un partitionnement car un composant ne peut pas appartenir à deux couches simultanément.  Une autre application possible de ce patron est son imitation pour classer les instances des relations horizontales selon le niveau d'abstraction où elles sont instanciées.

	
Conséquences d'application	Ce patron permet de créer une classification dynamique (qui peut être modifiée après l'instanciation de la base) des éléments instanciés dans une base *B-Sigma*. Combiné avec la relation de *Generalization*, il peut donner un grand pouvoir descriptif au modèle de bases descriptives de composants tout en gardant un bon compromis d'évolutivité.

1.11 Classification par relations des instances d'un concept dans *C-Sigma*.

Identifiant	Classification par relation des instances d'un concept dans *C-Sigma*.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Ce patron produit vise à guider l'administrateur d'une base *B-Sigma* pour établir une classification des instances d'un concept dans *C-Sigma*.
Forces	Ce patron est applicable sur le modèle *C-Sigma* même après son instanciation. Il n'affecte pas l'intégrité des données déjà stockées dans la base *C-Sigma*. La solution de ce patron ne s'applique que sur des entités de modélisation capables de participer à une relation d'*Association*, ce qui revient à dire sur toute instance d'une spécialisation concrète de la métaclasse *Classifier* (comme *Description*, *AssociationDescription*, *Component*, *ComponentsRepository*, etc.). Une classe peut contenir des instances de différentes métaclasses du métamodèle *M-Sigma* (*Description*, *Component*, etc.).
Solution produit	Un concept peut appartenir à plusieurs classes. Nous modélisons cette classification en ajoutant l'*AssociationDescription Relation de classification* qui associe un *concept* à sa *classe de concepts*. 
Conséquences d'application	Ce patron permet de créer une classification dynamique (peut être modifié après l'instanciation de la base) des éléments instanciés dans une base *C-Sigma*. Combiné avec la relation de *Generalization*, il peut donner un grand pouvoir descriptif au modèle de bases descriptives de composants tout en gardant un bon compromis d'évolutivité.

2. Conclusion

Nous avons présenté dans cette section des fragments d'une démarche sous la forme d'un système de patrons pour guider l'évolution du modèle *C-Sigma*. Ce guide méthodologique est destiné aux *ingénieurs de MBDC*. Ce guide permet d'étendre le modèle *C-Sigma* en ajoutant de nouveaux niveaux d'abstraction, de nouveaux modèles de composants, de nouvelles descriptions de composants, de nouvelles relations et de nouvelles classifications des entités du modèle. Nous avons ainsi augmenté l'adaptabilité et l'évolutivité de la base *B-Sigma* en fournissant un guide méthodologique permettant l'exploitation des points d'évolution présents dans le modèle *C-Sigma* (cf. chapitre IV).

Il est important de noter que ces fragments de démarche permettent de faire évoluer *C-Sigma* mais n'affectent en rien le métamodèle *M-Sigma* ou les descriptions déjà stockées dans une base *B-Sigma* conforme à un modèle *C-Sigma* initial.

VII. Un système de recherche de composants

L'état de l'art présenté dans le chapitre III sur les techniques de recherche de composants (TRC) met en évidence qu'une seule TRC ne peut pas répondre à tous les besoins utilisateurs. De plus, plusieurs TRC ne sont pas exploitables simultanément sur une base de composants car chacune de ces techniques utilise sa propre représentation interne des composants. Le métamodèle *M-Sigma* garantit l'indépendance de la représentation des composants vis-à-vis des TRC et constitue un premier pas vers l'exploitation de plusieurs TRC sur une base descriptive de composants (BDC). Ce chapitre propose le modèle d'un outil de recherche de composants qui nous permet d'exploiter plusieurs TRC sur les BDC gérées par un système de gestion de bases descriptives de composants (SGBDC). De plus, une technique originale de recherche de composants basée sur une approche structurelle adaptée aux diagrammes de classes est proposée.

1. Un modèle de Système de Recherche de Composants

Un modèle de SRC permet de construire des requêtes et de les évaluer en fonction des descriptions stockées dans une BDC. Le SRC récupère le modèle d'une BDC (MBDC). Ensuite, l'utilisateur exprime ses besoins sous la forme d'une requête simple (respectivement composite) portant sur une rubrique (respectivement plusieurs). Pour chacun des types de rubriques existant dans le SGBDC, il existe une ou plusieurs TRC qui lui sont compatibles.

Le modèle de notre SRC est composé des trois paquetages *Components Retrieval System Package*, *Simple Query Package*, *Query Integration Package* (cf. figure 62). Nous détaillons dans la suite de cette section chacun de ces paquetages.

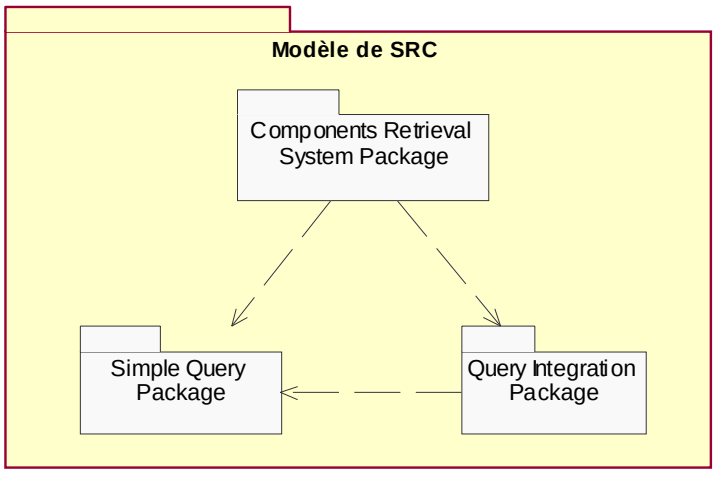


Figure 62. *Paquetages du modèle de systèmes de recherche de composants.*

1.1 Components Retrieval System Package

Le *Components Retrieval System Package* définit les principaux concepts du modèle de SRC : *ComponentsRetrievalSystem*, *ComponentsRetrievalTechnique*, *Query*, *QueryEditor*.

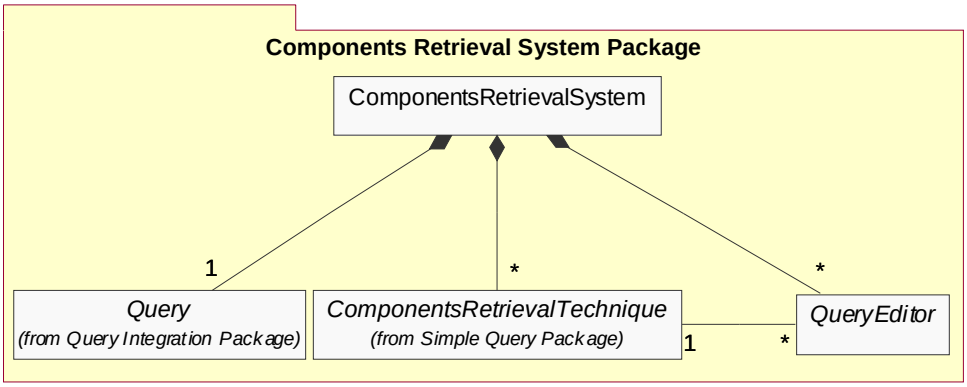


Figure 63. *Components Retrieval System Package.*

1.1.1 ComponentsRetrievalSystem

La classe *ComponentsRetrievalSystem* représente le système de recherche de composants. Cette classe offre une interface de l'outil SRC conforme au patron *Façade* de Gamma (Gamma, 1995).

1.1.2 QueryEditor

Chaque instance de la classe *QueryEditor* représente un éditeur de requêtes dédié à une technique de recherche de composants *ComponentsRetrievalTechnique*.

1.2 Simple Query Package

Le *Simple Query Package* définit et détaille le concept de requête simple utilisateur.

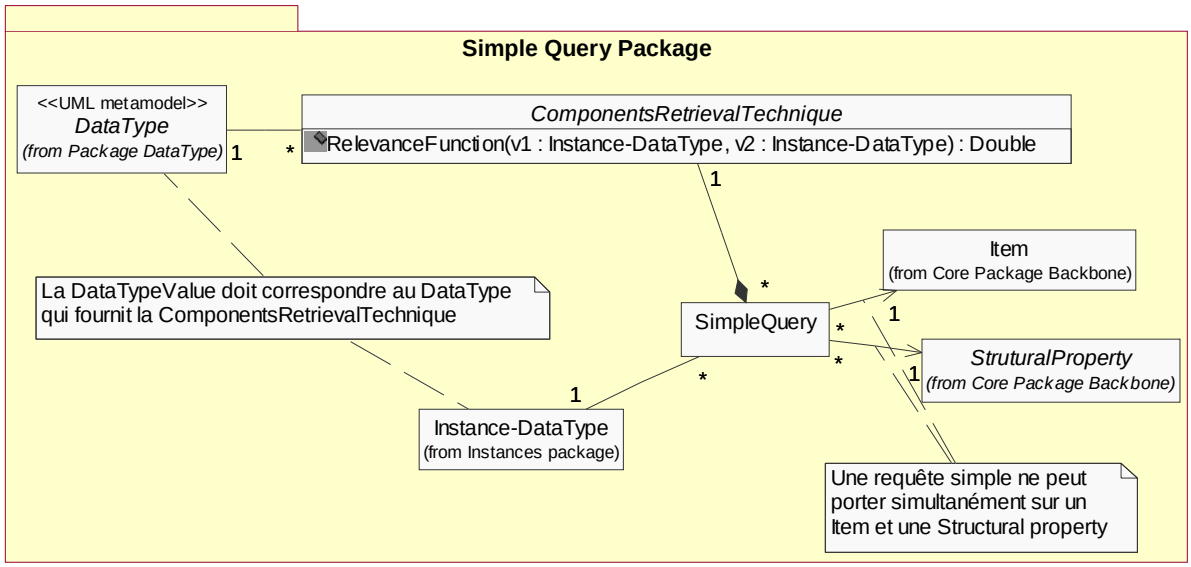


Figure 64. *Simple Query Package.*

1.2.1 SimpleQuery

Chaque instance de la classe *SimpleQuery* désigne une requête non composite utilisant une seule technique de recherche de composants (*ComponentsRetrievalTechnique*). Une requête simple (*SimpleQuery*) est appliquée sur une rubrique (respectivement une propriété) indiquée par une instance de la métaclasse *Item* (respectivement *StructuralProperty*) du métamodèle *M-Sigma* (cf. chapitre V). Un utilisateur du SRC fournit un ensemble de termes (valeurs constantes instances de la métaclasse *Instance-DataType*) pour paramétrer la requête *SimpleQuery*.

1.2.2 ComponentsRetrievalTechnique

Chaque instance de la classe abstraite *ComponentsRetrievalTechnique* représente une technique de recherche de composants applicable au type de données représenté par une spécialisation concrète de la métaclasse *DataType* du métamodèle *M-Sigma*. *ComponentsRetrievalTechnique* définit le contrat qui existe entre un SRC et une implantation d'une technique de recherche de composants.

1.3 Query Integration Package

Le *Query Integration Package* présente la partie du modèle du SRC qui gère le concept de requête composite et offre un mécanisme d'intégration des requêtes et un mécanisme de fusion de leurs résultats.

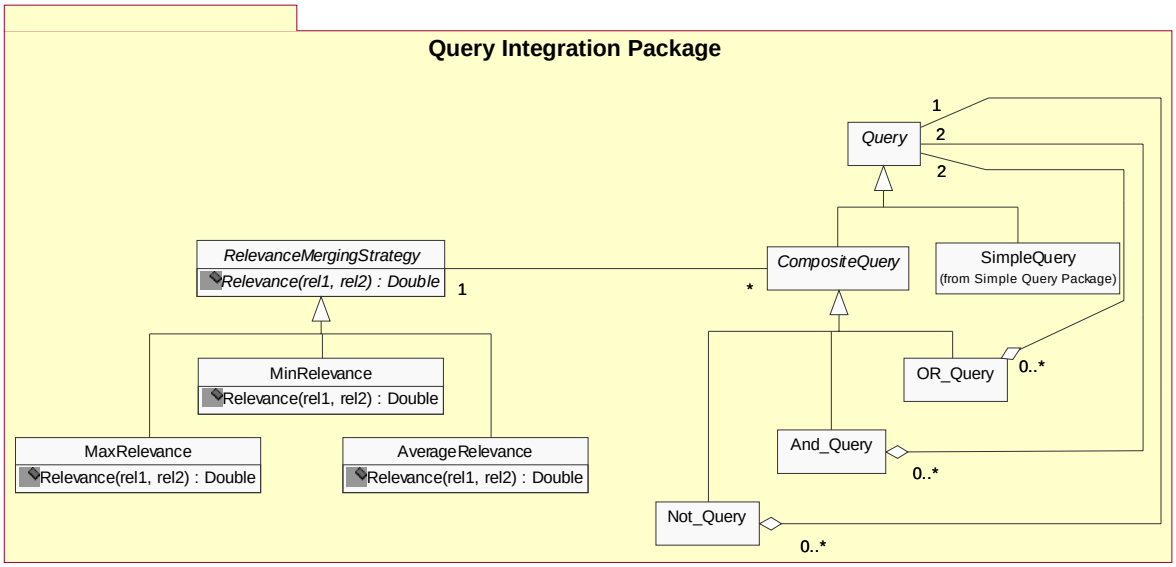


Figure 65. *Query Integration Package.*

1.3.1 Query

La classe abstraite *Query* représente une requête utilisateur pouvant être créée et évaluée par un SRC. Nous utilisons le patron composite (Gamma, 1995) pour organiser le concept de requête composite.

1.3.2 CompositeQuery

La classe abstraite *CompositeQuery* représente une expression de requêtes (*Query*) sous la forme d'un arbre utilisant des opérateurs logiques binaires (AND, OR) ou l'opérateur unaire Not. Une instance de la classe *CompositeQuery* est toujours associée à une instance de la classe *RelevanceMergingStrategy*.

1.3.3 RelevanceMergingStrategy

La classe abstraite *RelevanceMergingStrategy* définit une fonction abstraite de calcul de la pertinence utilisée pendant la fusion des résultats de deux sous-requêtes d'une requête composite (*CompositeQuery*) grâce à un opérateur logique binaire. Le *Query Integration Package* définit trois fonctions de calcul de pertinence : *MaxRelevance*, *MinRelevance*, *AverageRelevance*, ce qui calculent respectivement le maximum, le minimum et la moyenne des pertinences. Les spécialisations concrètes de la classe *RelevanceMergingStrategy* sont utilisées uniquement avec les spécialisations concrètes de la classe *CompositeQuery*. Ce sont des classes singletons.

1.4 Conclusion

Le modèle de systèmes de recherche de composants que nous avons présenté dans cette section permet à l'utilisateur d'utiliser plusieurs TRC sur un même SGBDC. Le SRC qui implante ce modèle est capable d'intégrer les résultats de plusieurs requêtes pour répondre à des requêtes composites (multi-techniques de recherche de composants) des utilisateurs. De la même manière que le SGBDC présenté dans la section précédente, Le SRC est évolutif. L'administrateur du SRC peut à tout moment ajouter de nouvelles techniques de recherche de composants et de nouvelles stratégies de fusion de requêtes composites. En particulier, la technique de recherche de composants, proposée dans la section suivante est applicable sur les rubriques et les propriétés du type diagramme de classes.

2. Une technique structurelle externe de recherche de composants

Cette section présente une technique de recherche de composants applicable sur les rubriques et les propriétés du type diagramme de classes UML (Khayati, 2005). Cette technique est de type « structurelle » car elle s'intéresse aux signatures des composants et à leur composition interne en terme de classes, d'associations, de compositions, etc. Elle est dite externe car elle ne traite pas les composants directement, mais leurs représentations sous la forme de diagrammes de classes. Les diagrammes de classes sont saisis manuellement ou extraits automatiquement en utilisant des outils de rétroconception fournis avec des AGL commerciaux comme Rational Rose[11], Objecteering[12] ou sous la forme de plugins à intégrer avec des environnements de développement intégré comme EclipseUML[13] (plugin d'Eclipse[14]).

La figure 66 présente un exemple de diagramme de classes documentant un composant réutilisable de la base *B-Sigma*. La notion de composant de type composite y est modélisée par imitation du patron *Composite* (Gamma, 1995) dont la solution est donnée en figure 67. Ce patron peut être vu comme un diagramme de classes correspondant à une requête d'un ingénieur d'applications qui veut retrouver tout composant de la base réutilisant dans sa structure la solution du patron *Composite*.

La représentation semi-formelle d'UML n'est pas bien adaptée à cette opération de recherche. Nous avons choisi d'utiliser une représentation plus formelle et plus opérationnalisable. Le

[11] http://www-306.ibm.com/software/rational/

[12] http://www.objecteering.com/

[13] http://www.omondo.com/

[14] http://www.eclipse.com/

formalisme de la logique du premier ordre offre l'avantage de permettre la description de la structure d'un diagramme de classes sous la forme de formules logiques. Le calcul des prédicats en tant que théorie du premier ordre définit des axiomes et des règles d'inférence qui permettent de prouver qu'une formule est une conséquence logique d'une autre formule. Nous utilisons donc cette technique pour détecter la structure d'un composant cible (par exemple le diagramme de classes cible de la figure 67) dans un autre composant source (par exemple le diagramme de classes source de la figure 66). Cela est le cas si la formule logique spécifiant le composant cible est une conséquence logique de la formule spécifiant le composant source. Un prouveur de théorèmes est appliqué sur les deux spécifications pour tenter de prouver cette implication. Ainsi il montrera que la structure du patron *Composite* de la figure 67 est présente dans le diagramme de la figure 66.

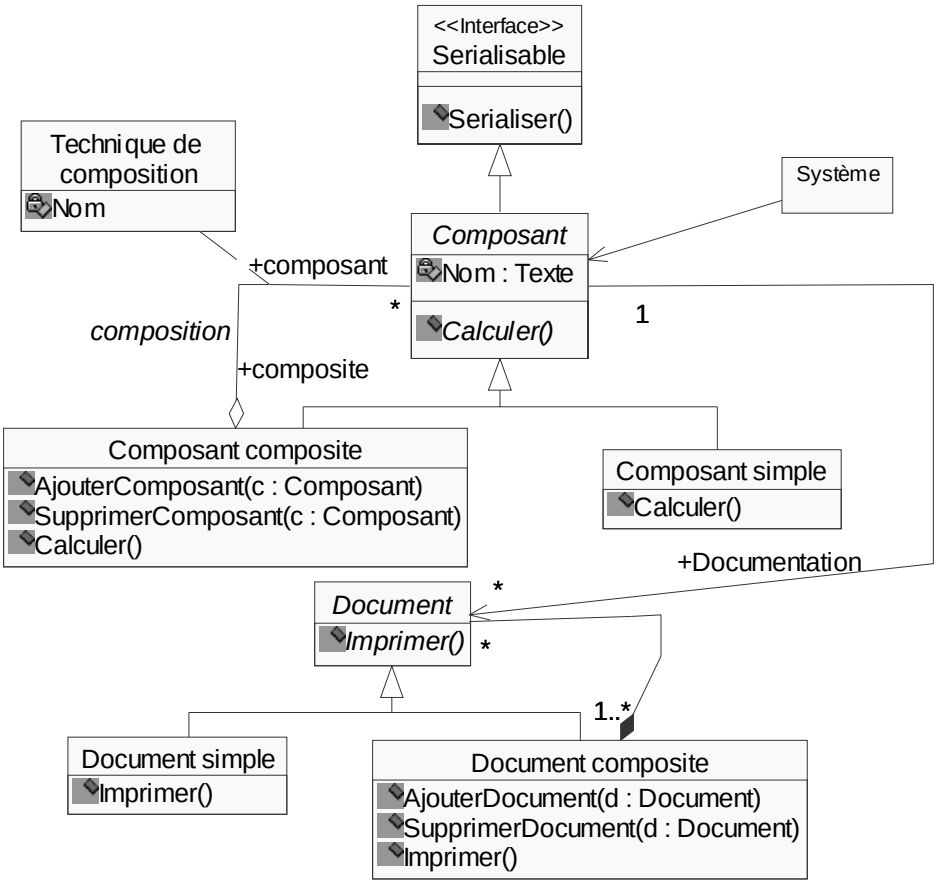


Figure 66. *Un diagramme de classes source d'une recherche.*

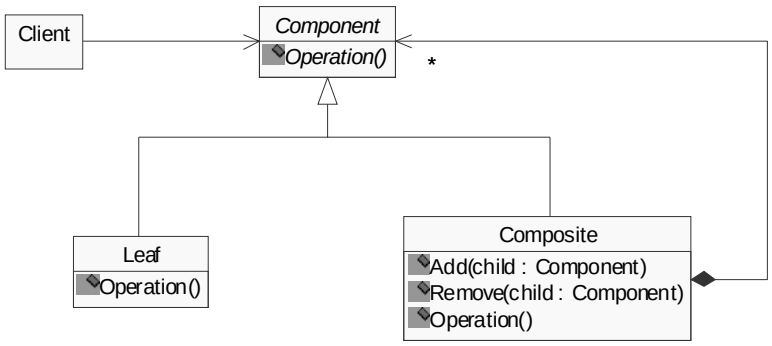


Figure 67. *Patron composite (Gamma, 1995).*

La figure 68 présente le processus de recherche de composants décrivant l'approche proposée dans cette section.

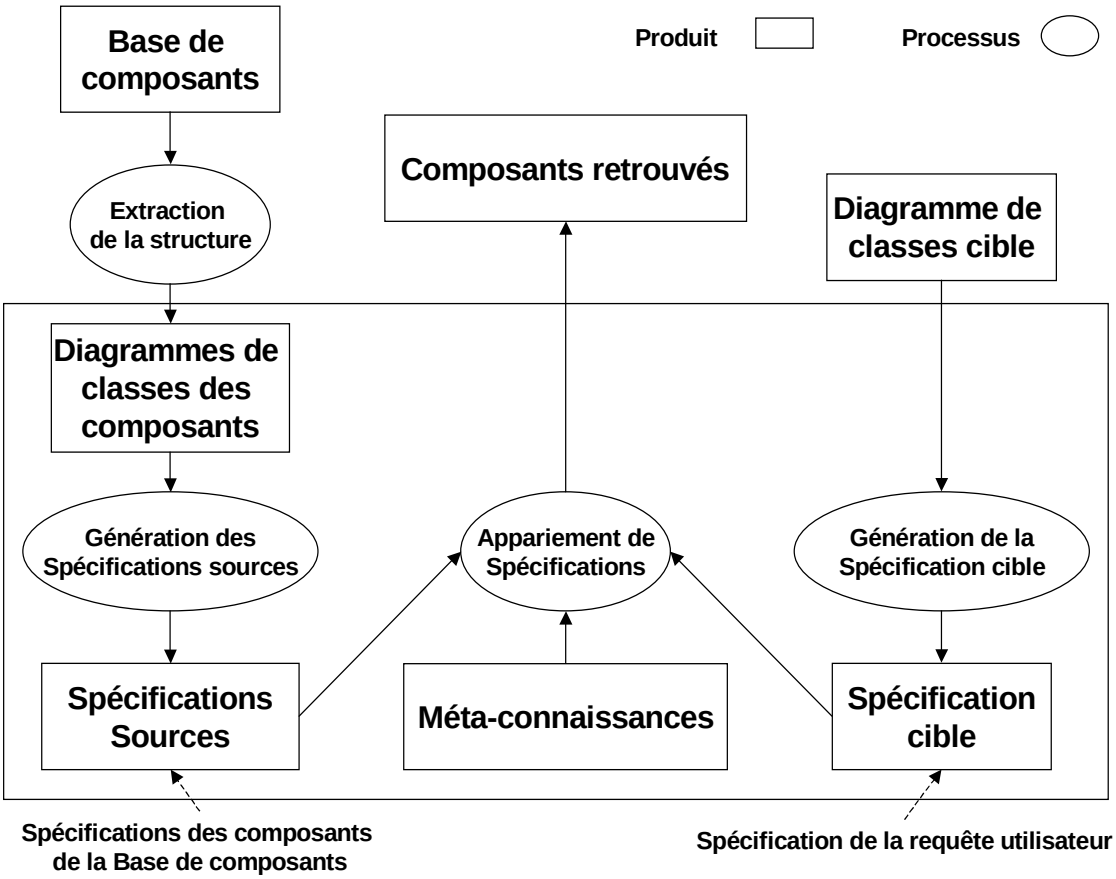


Figure 68. *Processus de recherche de composants.*

Les sections 2.1 et 2.2 présentent les techniques de génération des spécifications sources et cibles. La section 2.3 présente le mécanisme d'appariement de spécifications à base de métaconnaissances.

2.1 Processus de génération d'une spécification source

Une spécification source est générée à partir d'un diagramme de classes décrivant un composant de la base *B-Sigma*. Elle est entièrement exprimée avec des constantes qui représentent les instances des métaclasses du métamodèle UML (OMG, 2003) comme *associationend*, *class*, etc. Pour chacune de ces métaclasses, une spécification en logique du premier ordre est définie. Nous prenons comme exemple le diagramme de classes source présenté dans la figure 66. Dans le cadre de la génération de spécifications sources, les énoncés expriment des faits au travers de conjonctions de prédicats. Les constantes sont notées avec des minuscules.

2.1.1 Elément de modélisation

Chacun des éléments de modélisation présent dans un diagramme de classes est déclaré avec le prédicat *entité*() et nommé avec le prédicat *nom_entité*().

Exemple :

entité(id_classe_1) ^
nom_entité(id_classe_1, *Composant*)

2.1.2 Classe

Une classe est la description d'un ensemble d'objets qui partagent les mêmes attributs, opérations, associations et la même sémantique. Le prédicat *classe*() déclare [2] une classe. Une *classe* est définie par son nom (le prédicat *nom_entité()*[1]) et son abstraction (le prédicat *classe_abstraite()* [3] indique si une *classe* est abstraite).

Exemple :

entité(id_classe_1) ∧
nom_entité(id_classe_1, *Composant*)[1] ∧
classe(id_classe_1)[2] ∧
classe_abstraite(id_classe_1) [3]

2.1.3 Attribut

Un attribut est une propriété structurelle d'une classe permettant à un objet de stocker des informations. Le prédicat *attribut*() déclare [4] un attribut et le prédicat *classe_attribut*() associe un attribut à la classe qui le déclare[5]. Un *attribut* est défini par son nom[1], son type[2], sa valeur par défaut[3] et sa visibilité[6]. Le prédicat *visibilité*() définit la visibilité de l'attribut vis-à-vis des autres entités de modélisation dans le modèle (*public*, *privé* ou *protégé*). Dans l'exemple suivant, nous supposons qu'il existe une classe *Texte* identifiée par l'identifiant id_type_texte qui représente le type de données texte.

Exemple :

entité(id_attribut_1) ∧
nom_entité(id_attribut_1, *Nom*) [1] ∧
attribut(id_attribut_1) [4] ∧
classe_attribut(id_classe_1, id_attribut_1) [5] ∧
attribut_type(id_attribut_1,id_type_texte)[2] ∧
attribut_valeur_par_défaut(id_attribut_1, '') [3] ∧
visibilité (id_attribut_1, *privé*)[6]

2.1.4 Opération

Une opération est une propriété comportementale décrivant un service offert par une classe ou une interface. Une opération est déclarée par le prédicat *opération*() [4] et associée à la classe qui la déclare par le prédicat *classe_opération*() [5]. Une opération possède une *signature* qui décrit ses paramètres et sa valeur de retour (cf. 2.1.5). Une *opération* est définie par son nom[1], son abstraction [2] (une valeur booléenne qui indique si une opération est abstraite ou non), sa visibilité [6] et sa signature [3].

Exemple :

entité(id_opération_1) ∧
nom_entité(id_opération_1, *Calculer*)(1) ∧
opération(id_opération_1)(4) ∧
entité(id_signature_1) ∧
signature(id_signature_1)(3) ∧
classe_opération(id_classe_1, id_opération_1, id_signature_1)(5) ∧
opération_abstraite(id_opération_1)(2) ∧
visibilité(id_opération_1, *public*)(6)

2.1.5 Paramètre

Un *paramètre* est une variable qui peut être passée à une opération. Le prédicat *paramètre()* déclare un paramètre (5) et le prédicat *signature_paramètre*() l'associe à une signature (6). Un *paramètre* est défini par son ordre d'apparition dans la signature(7), son nom (1), son type(2), son type de communication(3) (paramètre_in (entrée), paramètre_out (sortie), paramètre_in_out (entrée sortie), paramètre_return (valeur de retour)) et sa valeur par défaut (4).

Exemple :

entité(id_paramètre_1) ∧ nom_entité(id_paramètre_1, opérande)(1) ∧
paramètre(id_paramètre_1) (5) ∧
signature_paramètre(id_signature_1, 1(7), id_paramètre_1)(6) ∧
paramètre_type(id_paramètre_1, id_type_texte)(2) ∧
paramètre_in(id_paramètre_1)(3) ∧
paramètre_valeur_par_défaut(id_paramètre_1, '')(4)

2.1.6 Association

Une *association* définit une relation sémantique entre *classifiers* (interfaces, classes, etc.). Une association possède un *nom*(1) et au moins deux instances de *associationend*. Un *classifier* peut être connecté à plus d'une *associationend* dans une *association*. Une association est déclarée avec le prédicat *association*()(2). Le prédicat *classe_associée*()(3) est utilisé pour déclarer une éventuelle classe d'association. Dans l'exemple suivant, *id_classe_2* représente l'identifiant de la classe d'association *technique de composition*.

Exemple :

entité(id_association_1) ∧
nom_entité(id_association_1, *composition*) (1) ∧
association(id_association_1) (2) ∧
classe_associée(id_association_1, id_classe_2) (3)

2.1.7 Associationend

Une *associationend* spécifie la relation entre une *association* et un *classifier*. C'est l'extrémité d'une *association*. Elle indique les classifiers compatibles avec l'extrémité de *l'association*. Une *associationend* est déclarée par le prédicat *associationend* ()(6) en précisant l'association qui la contient (par exemple, *id_association_1*) et les *classifiers* (interfaces ou classes) qu'elle accepte (par exemple, *id_classe_3*). Le nom de rôle(1) est déclaré en utilisant le prédicat *nom_entité()*. La navigabilité (2) et la multiplicité sont déclarées respectivement avec les prédicats *associationend_navigable()* et *associationend_multiplicité()*. Une *associationend* peut être du type association simple, agrégation (*associationend_agrégation()*) ou composition (*associationend_composition()*(4)). Dans l'exemple suivant, nous décrivons l'extrémité de la composition présentée dans la figure 66 qui se trouve du côté de la classe *composant composite*. *id_classe_3* est l'identifiant de la classe *composant composite*.

Exemple :

entité (id_association_end_1) ∧
nom_entité(id_association_end_1, *composite*) (1) ∧
associationend (id_association_1, id_association_end_1, id_classe_3)(6) ∧
associationend_multiplicité(id_association_end_1, '1')(5) ∧
associationend_navigable(id_association_end_1)(2) ∧
associationend_composition(id_association_end_1)(4)

2.1.8 Réalise

Réalise est une relation reliant une classe à une interface. La classe doit posséder toutes les méthodes définies dans l'interface. Une relation de réalisation est déclarée par le prédicat *réalise*()(1) et associée à la classe et à l'interface qu'elle relie par le prédicat *réalise_classe_interface*()(2). Dans l'exemple suivant, *id_interface_1* représente l'identifiant de l'interface *sérialisable* (cf. figure 66).

Exemple :

entité(id_realise_1) ∧
réalise(id_realise_1) (1) ∧
réalise_classe_interface(id_realise_1, id_classe_1, id_interface_1) (2)

2.1.9 Stéréotype

Le stéréotypage est un moyen de classification des éléments de modélisation. Un élément de modélisation stéréotypé se comporte comme s'il était une instance d'une métaclasse virtuelle. Un *stéréotype* est déclaré par le prédicat *stéréotype*() (3). Il est défini par son nom(2) et l'élément de modélisation qu'il stéréotype. Un stéréotype est associé à un élément de modélisation par le prédicat *stéréotype_élément_modélisation*()(1). L'exemple suivant montre la règle générale de déclaration d'un stéréotype (exemple concret dans la section suivante).

Exemple :

entité(id_stéréotype) ^
stéréotype(id_stéréotype)(3) ^
nom_entité(id_stéréotype, nom_stéréotype)(2) ^
stéréotype_élément_modélisation(id_stéréotype, id_model_element)(1)

2.1.10 Interface

Une interface est un ensemble nommé d'opérations et caractérise le comportement d'une classe. Une *interface* est définie par son nom et ses opérations. Nous faisons le choix de représenter une interface comme une classe stéréotypée « interface ». Ce choix de modélisation permet d'éviter de redéfinir les métaconnaissances pour l'héritage des attributs, des opérations et des relations. L'identifiant du stéréotype (2) « interface » est défini une seule fois dans un diagramme de classes qui définit des interfaces.

Exemple :

entité(id_stéréotype_interface) ^
stéréotype(id_stéréotype_interface)(3) ^
nom_entité(id_stéréotype_interface, *interface*)(2) ^
stéréotype_élément_modélisation(id_stéréotype_interface, id_interface_1)(1)

2.1.11 Généralisation

La généralisation est une relation taxonomique entre un élément général et un élément plus spécialisé. L'élément le plus spécialisé est complètement compatible avec l'élément général (il possède ses associations, ses attributs et ses opérations) et peut contenir d'autres informations. Une généralisation est déclarée par le prédicat *généralisation*()(1). Une *généralisation* est définie par les attributs suivants : *classifier* père (interface ou classe), *classifier* fils (interface ou classe). Dans l'exemple suivant, *Id_classe_1*, *Id_classe_3* et *Id_classe_5* sont respectivement *composant*, *composant composite*, *composant simple* dans la figure 66.

Exemple :

entité(id_généralisation_1) ^
généralisation(id_généralisation_1, id_classe_1, id_classe_3)(1) ^
généralisation(id_généralisation_1, id_classe_1, id_classe_5)

De la même manière que les spécifications sources sont générées, le processus présenté dans la section suivante permet de générer des spécifications cibles.

2.2 Processus de génération d'une spécification cible

Une spécification cible est générée à partir d'un diagramme de classes cible qui spécifie les besoins d'un ingénieur d'applications désirant retrouver un composant répondant à ses critères de recherche. Pour illustrer notre proposition, nous appliquons ces deux phases sur le diagramme de classes de la figure 67. Dans ce processus de génération d'une spécification cible interviennent des variables dont les noms commencent par le caractère '_' suivi d'une

majuscule. Le processus de génération d'une spécification cible est composé de deux phases entièrement automatisées : une phase de génération automatique et une phase de paramétrage. Nous présentons dans la suite de cette section les deux phases.

2.2.1 Phase 1 : Préparation d'une spécification cible

Dans un premier temps, le processus de génération d'une spécification cible utilise le processus de génération d'une spécification source présenté dans la section précédente. Le résultat obtenu est une spécification formée uniquement de constantes. Ensuite, toutes les constantes représentant les identifiants d'entités de modélisation sont remplacées par des variables et pour chaque variable, la clause $\forall$ *nom_variable* est ajoutée au début de la spécification. Enfin, la clause $\Rightarrow$ requête (nom_var1, nom_var2, etc.) est ajoutée à la fin de la spécification cible. La spécification suivante est obtenue suite à cette première phase appliquée à l'exemple de la figure 67.

$\forall$ **_R_id_classe_1** $\forall$ **_R_id_classe_2** $\forall$ **_R_id_classe_3** $\forall$ **_R_id_opération_1**
$\forall$ **_R_id_opération_2** $\forall$ **_R_id_opération_3** $\forall$ **_R_id_signature_1**
$\forall$ **_R_id_signature_2** $\forall$ **_R_id_signature_3** $\forall$ **_R_id_paramètre_1**
$\forall$ **_R_id_paramètre_2** $\forall$ **_R_id_association_1** $\forall$ **_R_id_association_end_1**
$\forall$ **_R_id_association_end_2**

// classe Component
entité(**_R_id_classe_1**) $\wedge$
nom_entité(**_R_id_classe_1**, *component*) $\wedge$
classe(**_R_id_classe_1**) $\wedge$
classe_abstraite(**_R_id_classe_1**) $\wedge$

// opération *component.operation()*
entité(**_R_id_opération_1**) $\wedge$
nom_entité(**_R_id_opération_1**, *operation*) $\wedge$
opération(**_R_id_opération_1**) $\wedge$
entité(**_R_id_signature_1**) $\wedge$
signature(**_R_id_signature_1**) $\wedge$
classe_opération(**_R_id_classe_1**, **_R_id_opération_1**, **_R_id_signature_1**) $\wedge$
opération_abstraite(**_R_id_opération_1**) $\wedge$
visibilité(**_R_id_opération_1**, *public*) $\wedge$

// classe Composite
entité(**_R_id_classe_2**) $\wedge$
nom_entité(**_R_id_classe_2**, *composite*) $\wedge$
classe(**_R_id_classe_2**) $\wedge$

// opération *composite.add()*
entité(**_R_id_opération_2**) ^
nom_entité(**_R_id_opération_2**, *add*) ^
opération(**_R_id_opération_2**) ^
entité(**_R_id_signature_2**) ^
signature(**_R_id_signature_2**) ^
classe_opération(**_R_id_classe_2**, **_R_id_opération_2**, **_R_id_signature_2**) ^
visibilité(**_R_id_opération_1**, *public*) ^

// paramètre *child* dans opération *composite.add()*
entité(**_R_id_paramètre_1**) ^
nom_entité(**_R_id_paramètre_1**, *child*) ^
paramètre(**_R_id_paramètre_1**) ^
signature_paramètre(**_R_id_signature_2**, 1, **_R_id_paramètre_1**) ^
paramètre_type(**_R_id_paramètre_1**, **_R_id_classe_1**) ^
paramètre_in(**_R_id_paramètre_1**) ^

// opération *composite.remove()*
entité(**_R_id_opération_3**) ^
nom_entité(**_R_id_opération_3**, *remove*) ^
opération(**_R_id_opération_3**) ^
entité(**_R_id_signature_3**) ^
signature(**_R_id_signature_3**) ^
classe_opération(**_R_id_classe_2**, **_R_id_opération_3**, **_R_id_signature_3**) ^
visibilité(**_R_id_opération_3**, *public*) ^

// paramètre *child* dans opération *composite.remove ()*
entité(**_R_id_paramètre_2**) ^
nom_entité(**_R_id_paramètre_2**, *child*) ^
paramètre(**_R_id_paramètre_2**) ^
signature_paramètre(**_R_id_signature_3**, *1*, **_R_id_paramètre_2**) ^
paramètre_type(**_R_id_paramètre_2**, **_R_id_classe_1**)^
paramètre_in(**_R_id_paramètre_2**) ^

// opération *composite.operation()*

entité(**_R_id_opération_4**) ^
nom_entité(**_R_id_opération_4**, *operation*) ^
opération(**_R_id_opération_4**) ^
entité(**_R_id_signature_4**) ^
signature(**_R_id_signature_4**) ^
classe_opération(**_R_id_classe_2**, **_R_id_opération_4**, **_R_id_signature_4**) ^
visibilité(**_R_id_opération_4**, *public*) ^
// classe *leaf*
entité(**_R_id_classe_3**) ^
nom_entité(**_R_id_classe_3**, *leaf*) ^
classe(**_R_id_classe_3**) ^

// opération *leaf.operation()*
entité(**_R_id_opération_5**) ^
nom_entité(**_R_id_opération_5**, *operation*) ^
opération(**_R_id_opération_5**) ^
entité(**_R_id_signature_5**) ^
signature(**_R_id_signature_5**) ^
classe_opération(**_R_id_classe_3**, **_R_id_opération_5**, **_R_id_signature_5**) ^
visibilité(**_R_id_opération_5**, *public*) ^

// relation de généralisation
entité(**_R_id_généralisation_1**) ^
généralisation(**_R_id_généralisation_1**, **_R_id_classe_1**, **_R_id_classe_2**) ^
généralisation(**_R_id_généralisation_1**, **_R_id_classe_1**, **_R_id_classe_3**) ^

// relation de composition
entité(**_R_id_association_1**) ^
association(**_R_id_association_1**) ^

// association end du côté de la classe *component*
entité (**_R_id_association_end_1**)^
associationend(**_R_id_association_1**, **_R_id_association_end_1**, **_R_id_classe_1**) ^
associationend_multiplicité(**_R_id_association_end_1**, '*')^

// association end du côté de la classe *composite*
entité (**_R_id_association_end_2**) ^
associationend(**_R_id_association_1**, **_R_id_association_end_2**, **_R_id_classe_2**) ^
associationend_multiplicité(**_R_id_association_end_2**, *'1'*)^
associationend_navigable(**_R_id_association_end_2**)^
associationend_composition(**_R_id_association_end_2**)

// classe Client
entité(**_R_id_classe_4**) ^
nom_entité(**_R_id_classe_4**, *client*) ^
classe(**_R_id_classe_4**)

// association end du côté de la classe *composite*
entité(**_R_id_association_2**) ^
association(**_R_id_association_2**) ^

// association end du côté de la classe *component*
entité (**_R_id_association_end_3**)^
associationend(**_R_id_association_2**, **_R_id_association_end_3**, **_R_id_classe_1**) ^
associationend_multiplicité(**_R_id_association_end_3**, *'*'*)^

// association end du côté de la classe *client*
entité (**_R_id_association_end_4**)^
associationend(**_R_id_association_2**, **_R_id_association_end_4**, **_R_id_classe_4**) ^
associationend_multiplicité(**_R_id_association_end_4**, *'*'*)^
associationend_navigable(**_R_id_association_end_4**)^

⇒ requete (**_R_id_classe_1**, **_R_id_classe_2**, **_R_id_classe_3**, **_R_id_classe_4**, **_R_id_opération_1**, **_R_id_opération_2**, **_R_id_opération_3**, **_R_id_opération_4**, **_R_id_opération_5** **_R_id_signature_1**, **_R_id_signature_2**, **_R_id_signature_3**, **_R_id_signature_4**, **_R_id_signature_5**, **_R_id_paramètre_1**, **_R_id_paramètre_2**, **_R_id_association_1**, **_R_id_association_2**, **_R_id_association_end_1**, **_R_id_association_end_2**, **_R_id_association_end_3**, **_R_id_association_end_4**)

2.2.2 Phase 2 : Paramétrage d'une spécification cible

La spécification cible obtenue après la première phase du processus de génération permet de détecter la structure du diagramme de classes de la figure 67. Cependant, son appariement

avec la spécification source générée à partir de la figure 66 n'aboutira pas à une réponse positive car les noms des méthodes et des classes ne sont pas les mêmes. L'ingénieur d'applications intervient alors grâce à une interface graphique pour modifier la spécification pour, par exemple, ignorer les noms de méthodes ou de paramètres en remplaçant les constantes (*component*, *composite*, *leaf*, *operation*, *add*, *remove*, *child*) qui les représentent par des variables (*_component*, *_composite*, *_leaf*, *_operation*, *_add*, *_remove*, *_child*). Il peut également conserver les constantes spécifiant les cardinalités de la composition ou les remplacer par des variables. La spécification cible adaptée est la suivante :

∀ _R_id_classe_1 ∀ _R_id_classe_2 ∀ _R_id_classe_3 ∀ _R_id_opération_1
∀ _R_id_opération_2 ∀ _R_id_opération_3 ∀ _R_id_signature_1
∀ _R_id_signature_2 ∀ _R_id_signature_3 ∀ _R_id_paramètre_1
∀ _R_id_paramètre_2 ∀ _R_id_association_1 ∀ _R_id_association_end_1
∀ _R_id_association_end_2 ∀ **_component** ∀ **_operation** ∀ **_composite** ∀ **_add**
∀ **_fils** ∀ **_remove** ∀ **_leaf**

// classe *component*
entité(_R_id_classe_1) ^
nom_entité(_R_id_classe_1, **_component**) ^
classe(_R_id_classe_1) ^
classe_abstraite(_R_id_classe_1)^

// opération *component.operation()*
entité(_R_id_opération_1) ^
nom_entité(_R_id_opération_1, **_operation**) ^
opération(_R_id_opération_1) ^
entité(_R_id_signature_1) ^
signature(_R_id_signature_1) ^
classe_opération(_R_id_classe_1, _R_id_opération_1, _R_id_signature_1) ^
opération_abstraite(_R_id_opération_1) ^
visibilité(_R_id_opération_1, public) ^

// classe *composite*
entité(_R_id_classe_2) ^
nom_entité(_R_id_classe_2, **_composite**) ^
classe(_R_id_classe_2) ^

// opération *composite.add()*
entité(_R_id_opération_2) ^

nom_entité(_R_id_opération_2, **_add**) ^
opération(_R_id_opération_2) ^
entité(_R_id_signature_2) ^
signature(_R_id_signature_2) ^
classe_opération(_R_id_classe_2, _R_id_opération_2, _R_id_signature_2) ^
visibilité(_R_id_opération_1, public) ^
// paramètre *child* dans opération *composite.add()*
entité(_R_id_paramètre_1) ^
nom_entité(_R_id_paramètre_1, **_child**) ^
paramètre(_R_id_paramètre_1) ^
signature_paramètre(_R_id_signature_2, 1, _R_id_paramètre_1) ^
paramètre_type(_R_id_paramètre_1, _R_id_classe_1) ^
paramètre_in(_R_id_paramètre_1) ^

// opération *composite.remove()*
entité(_R_id_opération_3) ^
nom_entité(_R_id_opération_3, **_remove**) ^
opération(_R_id_opération_3) ^
entité(_R_id_signature_3) ^
signature(_R_id_signature_3) ^
classe_opération(_R_id_classe_2, _R_id_opération_3, _R_id_signature_3) ^
visibilité(_R_id_opération_3, public) ^

// paramètre *child* dans opération *composite.remove()*
entité(_R_id_paramètre_2) ^
nom_entité(_R_id_paramètre_2, **_child**) ^
paramètre(_R_id_paramètre_2) ^
signature_paramètre(_R_id_signature_3, 1, _R_id_paramètre_2) ^
paramètre_type(_R_id_paramètre_2, _R_id_classe_1) ^
paramètre_in(_R_id_paramètre_2) ^

// opération *composite.operation()*
entité(_R_id_opération_4) ^
nom_entité(_R_id_opération_4, **_operation**) ^
opération(_R_id_opération_4) ^
entité(_R_id_signature_4) ^

signature(_R_id_signature_4) ^
classe_opération(_R_id_classe_2, _R_id_opération_4, _R_id_signature_4) ^
visibilité(_R_id_opération_4, public) ^

// classe leaf
entité(_R_id_classe_3) ^
nom_entité(_R_id_classe_3, **_leaf**) ^
classe(_R_id_classe_3) ^

// opération *leaf.operation()*
entité(_R_id_opération_5) ^
nom_entité(_R_id_opération_5, **_operation**) ^
opération(_R_id_opération_5) ^
entité(_R_id_signature_5) ^
signature(_R_id_signature_5) ^
classe_opération(_R_id_classe_3, _R_id_opération_5, _R_id_signature_5) ^
visibilité(_R_id_opération_5, public) ^

// relation de généralisation
entité(_R_id_généralisation_1) ^
généralisation(_R_id_généralisation_1, _R_id_classe_1, _R_id_classe_2) ^
généralisation(_R_id_généralisation_1, _R_id_classe_1, _R_id_classe_3) ^

// relation de composition
entité(_R_id_association_1) ^
association(_R_id_association_1) ^

// association end du côté de la classe *composant*
entité (_R_id_association_end_1) ^
associationend(_R_id_association_1, _R_id_association_end_1, _R_id_classe_1) ^
associationend_multiplicité(_R_id_association_end_1, '*')^

// association end du côté de la classe *composite*
entité (_R_id_association_end_2) ^
associationend(_R_id_association_1, _R_id_association_end_2, _R_id_classe_2) ^

associationend_multiplicité(_R_id_association_end_2, ‘1’) ^
associationend_navigable(_R_id_association_end_2) ^
associationend_composition(_R_id_association_end_2) ^

// classe Client
entité(_R_id_classe_4) ^
nom_entité(_R_id_classe_4, **_client**) ^
classe(_R_id_classe_4) ^

// association entre la classe client et la classe composite
entité(_R_id_association_2) ^
association(_R_id_association_2) ^

// association end du côté de la classe *component*
entité (_R_id_association_end_3)^
associationend(_R_id_association_2, _R_id_association_end_3, _R_id_classe_1) ^
associationend_multiplicité(_R_id_association_end_3, ‘*’)^

// association end du côté de la classe *client*
entité (_R_id_association_end_4)^
associationend(_R_id_association_2, _R_id_association_end_4, _R_id_classe_4) ^
associationend_multiplicité(_R_id_association_end_4, ‘*’)^
associationend_navigable(_R_id_association_end_4)^

⇒ requete (R_id_classe_1, _R_id_classe_2, _R_id_classe_3, _R_id_classe_4, _R_id_opération_1, _R_id_opération_2, _R_id_opération_3, _R_id_opération_4, _R_id_opération_5,_R_id_signature_1, _R_id_signature_2, _R_id_signature_3, _R_id_signature_4, _R_id_signature_5, _R_id_paramètre_1, _R_id_paramètre_2, _R_id_association_1, _R_id_association_2, _R_id_association_end_1, _R_id_association_end_2, _R_id_association_end_3, _R_id_association_end_4, **_component, _operation, _composite, _add, _child**, **_remove, _leaf, _client,**)

Les deux spécifications sources et cibles ayant ainsi été générées, il est désormais nécessaire de les apparier grâce au mécanisme présenté ci après.

2.3 Mécanisme d'appariement de spécifications à base de métaconnaissances

Le processus de recherche de composants que nous proposons se base sur un processus d'appariement de spécifications. Il recherche des instances des variables se trouvant dans la spécification source qui vérifient les contraintes spécifiées dans la spécification cible. Le mécanisme d'appariement de spécifications est un algorithme de résolution de formules (spécifications cibles) en logique du premier ordre implanté par un prouveur de théorèmes. Les processus de génération de spécifications présentés dans les sections 2.1 et 2.2 spécifient un diagramme de classes sous une forme déclarative en utilisant la logique du premier ordre. Or, des concepts comme la *généralisation* ont une sémantique plus large que le simple fait de relier deux *classifiers* (interfaces ou classes). La relation de *généralisation* propage les informations contenues dans le *classifier* général (attributs, opérations, associations, réalisations) vers ses spécialisations. Lors de la phase d'appariement des spécifications, il faut tenir compte de cette métaconnaissance. L'utilisateur doit choisir les métaconnaissances qu'il veut utiliser lors de la phase d'appariement dans le but d'améliorer les performances du système. Par exemple, s'il s'intéresse uniquement aux relations qui existent entre les classes, il n'a pas besoin de la propagation des attributs et des méthodes induites par les relations de spécialisation. Les métaconnaissances sélectionnées par l'ingénieur d'applications sont intégrées dans la spécification source pendant la phase d'appariement. Les métaconnaissances que nous présentons ci-dessous traitent des aspects sémantiques de la généralisation et des techniques de relaxation de l'appariement pour améliorer le critère de rappel.

2.3.1 Mécanismes de propagation des propriétés par la généralisation

La relation de généralisation entre deux classes implique la propagation des propriétés (relations, attributs, opérations) de la classe la plus générale vers les classes spécialisées.

2.3.1.1 Propagation des attributs

La règle suivante décrit la propagation des attributs dans une généralisation.

$\forall$ _id_généralisation
$\forall$ _id_classe_générale
$\forall$ _id_classe_spécialisée
$\forall$ _id_attribut
généralisation(_id_généralisation, _id_classe_générale, _id_classe_spécialisée) $\wedge$
classe_attribut(_id_classe_générale, _id_attribut)
$\Rightarrow$ classe_attribut(_id_classe_spécialisée, _id_attribut)

2.3.1.2 Propagation des opérations

La règle suivante décrit la propagation des opérations dans une généralisation.

∀ _id_généralisation
∀ _id_classe_générale
∀ _id_classe_spécialisée
∀ _id_opération
∀ _id_signature_opération
généralisation(_id_généralisation, _id_classe_générale, id_classe_spécialisée) ∧
classe_opération(_id_classe_générale, _id_opération, _id_signature_opération)
⇒ classe_opération(_id_classe_spécialisée, _id_opération, id_signature_opération)

2.3.1.3 Propagation des relations

a) Relation de réalisation

Si la classe générale réalise une interface, alors la classe spécialisée réalise aussi cette interface :

∀ _id_généralisation
∀ _id_classe_générale
∀ _id_classe_spécialisée
∀ _id_réalise
∀ _id_interface
réalise(_id_réalise, _id_classe_générale, _id_interface) ∧
généralisation(_id_généralisation, _id_classe_générale, _id_classe_spécialisée)
⇒ réalise(_id_réalise, _id_classe_spécialisée, _id_interface)

Si une classe réalise une interface spécialisée, alors elle réalise aussi l'interface générale :

∀ _id_généralisation
∀ _id_interface_générale
∀ _id_interface_spécialisée
∀ id_réalise
∀ id_classe
généralisation(_id_généralisation,_id_interface_générale, _id_interface_spécialisée) ∧
réalise(id_réalise, id_classe, id_interface_spécialisée)
⇒ réalise(id_réalise, id_classe, id_interface_générale)

b) Associations

Si un *classifier* (interface ou classe) est relié par une association à un autre *classifier*, alors sa spécialisation l'est aussi.

$\forall$ _id_généralisation
$\forall$ _id_classifier_général
$\forall$ _id_classifier_spécialisé
$\forall$ _id_association
$\forall$ _id_association_end
généralisation(_id_généralisation,_id_classifier_général,_id_classifier_spécialisé) $\wedge$
associationend(_id_association, _id_association_end,_id_classifier_général)
$\Rightarrow$ associationend(_id_association, _id_association_end,_id_classifier_spécialisé)

La clause de propagation des associations (association simple, agrégation et composition) doit être utilisée avec précaution car elle effectue une fermeture de toutes les associations. Elle doit donc être utilisée uniquement lorsque l'utilisateur désire retrouver tous les *classifiers* dont les instances peuvent être reliées à une instance du *classifier* source.

L'appariement de la spécification cible de la section 2.2 avec la spécification source de la section 2.1 permet de détecter la structure du patron composite. Le tableau suivant présente un sous-ensemble des instances des variables présentes dans la spécification cible et trouvées par le processus d'appariement.

Variable	Instance
_component	document
_composite	documentcomposite
_leaf	documentsimple
_operation	imprimer
_add	ajouterdocument
_remove	supprimerdocument
_child	d
_client	composant

2.3.2 Mécanisme de relaxation

Notre technique de recherche de composants a pour but de détecter la structure d'un diagramme de classes cible dans les composants de la base de composants. Sans le mécanisme de relaxation, le processus d'appariement retourne un résultat du type tout ou rien (composants détectés ou non). Or, il est parfois intéressant de retrouver des composants qui ressemblent à quelques détails près à la requête sans satisfaire toutes ses contraintes. Nous appelons ce cas de figure un appariement relaxé.

2.3.2.1 Relaxation des associations

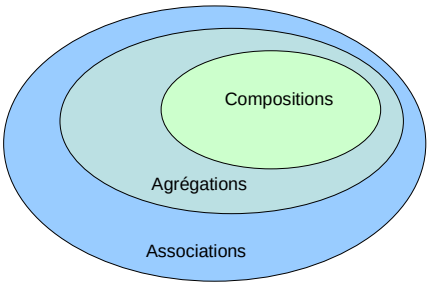


Figure 69. *Classification des associations.*

La figure 69 montre la classification des différents types d'association. Une agrégation est une association avec la sémantique supplémentaire qu'une classe appartient à une autre classe. Une composition est une agrégation avec un critère d'exclusivité supplémentaire. La clause suivante exprime qu'une composition est un type particulier d'agrégation :

Règle de relaxation descendante :

$\forall$ id_association_end
associationend_composition(id_association_end)

$\Rightarrow$ associationend_agrégation(id_association_end)

Nous pouvons opter également pour une relaxation dans le sens inverse de celui présenté dans la règle précédente. Dans ce cas de figure nous aboutissons à la règle suivante.

Règle de relaxation ascendante :

$\forall$ id_association_end
associationend_agrégation(id_association_end)

$\Rightarrow$ associationend_composition(id_association_end)

L'appariement de la spécification cible extraite du diagramme de la figure 67 avec la spécification source extraite du diagramme de la figure 66 sans relaxation détecte la structure du patron composite formée par les classes *Document*, *Document simple*, *Document composite* et *Composant*. L'utilisation de la règle ascendante pendant le processus d'appariement détecte en plus la structure formée par les classes *Composant*, *Composant composite*, *Composant simple* et *Système*. Dans cette imitation du patron *Composite*, l'ingénieur d'applications a remplacé la composition par l'agrégation.

D'autres types de relaxations sont possibles sur les associations comme la relaxation entre une cardinalité monovalueé (1, 0..1) et une cardinalité multivaluée (1..*, 0..*).

2.3.2.2 Relaxation des types de paramètres des opérations

Lors de la phase d'appariement, il est parfois intéressant de retourner une opération dont la signature est une spécialisation ou une généralisation de la signature recherchée par la spécification cible. Le prédicat suivant effectue la relaxation par spécialisation et on peut faire de même pour la relaxation par généralisation. Le prédicat *descendant*(*id1*,*id2*) exprime que le type id2 est une spécialisation directe ou indirecte du type *id1*.

Relaxation par spécialisation :

∀ _id_paramètre
∀ _id_type
∀ _id_type_spécialisé
paramètre_type(_id_paramètre, _id_type) ∧
descendant(id_type, _id_type_spécialisé)
⇒ paramètre_type(_id_paramètre, _id_type_spécialisé)

Relaxation par généralisation :

∀ _id_paramètre
∀ _id_type
∀ _id_type_spécialisé
paramètre_type(_id_paramètre, _id_type) ∧
descendant(_id_type_spécialisé, id_type)
⇒ paramètre_type(_id_paramètre, _id_type_spécialisé)

La règle de relaxation par généralisation est similaire à la règle de relaxation par spécialisation sauf qu'on inverse les paramètres dans le prédicat *descendant()* ce qui est équivalent à utiliser un prédicat *ancêtre()*.

3. Conclusion

3.1 Evaluation

La technique de recherche de composants proposée dans cette section exploite les rubriques du type diagramme de classes UML. L'intérêt de cette approche par rapport aux approches par signature (cf. chapitre III) est qu'elle ne s'intéresse pas uniquement à la signature du composant et de ses classes, mais elle permet aussi de décrire les relations qui peuvent exister entre les entités qui le forment (sous-composants, classes, interfaces, etc.). Ceci permet d'indexer des configurations de composants. De plus, notre technique est souple et facile à utiliser puisque l'utilisateur ne manipule pas directement les spécifications logiques. En effet, il décrit ses requêtes sous forme de diagrammes de classes et l'outil génère les spécifications semi-automatiquement. La phase d'indexation des composants est entièrement automatisable. D'un point de vue évolution des besoins utilisateurs, il est possible pour un administrateur de la base de composants de définir de nouvelles métaconnaissances qui peuvent être sélectionnées par les utilisateurs de la base pour influencer le mécanisme d'appariement des spécifications et l'adapter à leurs besoins.

Les différents tableaux ci-dessous comparent la technique de recherche de composants proposée dans ce chapitre avec les différentes techniques étudiées dans le chapitre III : techniques classiques de recherche d'information (TRI), techniques de classification externe (TCE), techniques de classification structurelle (TCS) et techniques de recherche comportementale (TRC). Nous rappelons que nous avons homogénéisé la présentation des mesures des critères selon une échelle discrète à cinq valeurs : très mauvaise (--), mauvaise (-), moyenne (=), bonne (+), très bonne (++).

Tableau 13. *Évaluation par rapport aux critères techniques.*

Critères techniques	**TRI**	**TCE**	**TCS**	**TRC**	**Technique proposée**
Précision	=	=	++	++	+
Rappel	=	=	++	=	+
Couverture	-	=	-	-	--
Complexité d'appariement	++	+	--	=	-
Potentiel d'automatisation	++	=	-	++	++

La technique proposée étant une TCS, elle hérite d'un bon rappel et d'une bonne précision, mais bénéficie en plus d'un très haut potentiel d'automatisation. Lors de l'évaluation d'une requête, tous les composants sont visités, d'où un taux de couverture maximal, qui a pour inconvénient de détériorer les performances du système.

Tableau 14. *Évaluation par rapport aux critères économiques.*

Critères économiques	**TRI**	**TCE**	**TCS**	**TRC**	**Technique proposée**
Coût d'investissement	+	-	--	+	++
Coût de fonctionnement	+	-	--	+	++
Degré de diffusion	=	++	+	-	--
Etat de développement	+	++	=	-	-

À l'inverse des TCS, la technique que nous proposons a des coûts d'investissement très bas (cf. tableau 14) et elle est très facile à utiliser et complétememnt transparente (cf. tableau 15).

Tableau 15. *Évaluation par rapport aux critères humains.*

Critères humains	**TRI**	**TCE**	**TCS**	**TRC**	**Technique proposée**
Difficulté d'utilisation	=	+	--	=	++
Transparence	=	-	--	+	++

Le tableau 16 montre que notre technique n'a pas de restrictions sur le type de composants, le domaine, etc. La seule faiblesse est la structure plate de l'organisation des spécifications qui risque d'engendrer des problèmes de performance. Nous répondons partiellement à ce problème en utilisant d'autres techniques de recherche de composants pour faire de la présélection car le modèle de SRC le permet.

Tableau 16. *Évaluation par rapport aux caractéristiques de conception.*

Caractéristiques de conception	**TRI**	**TCE**	**TCS**	**TRC**	**Technique proposée**
Nature des composants	Aucune restriction	Aucune restriction	Exécutable, frameworks de spécification de besoins	Exécutable	Aucune restriction
Couverture	Spécifique à un domaine	Spécifique à un domaine	Aucune restriction	Aucune restriction	Aucune restriction
Etendue	Equipe ou organisation	Equipe ou organisation	Aucune restriction	Aucune restriction	Aucune restriction
Portée	Aucune restriction	Aucune restriction	Aucune restriction	Implantation	Aucune restriction
Ouverture	Boîte blanche ou en verre	Aucune restriction	Aucune restriction	Aucune restriction	Aucune restriction
Représentation de la requête	Langage naturel ou code source	Langage naturel / Mots-clés	Spécification formelle	Trace d'exécution	Diagramme de classes
Représentation d'un composant	Langage naturel ou code source	Langage naturel / Mots-clés	Spécification formelle	Trace d'exécution	Diagramme de classes / Spécification formelle
Structure de stockage	Généralement plate	Généralement plate et indexée	Généralement sous forme de graphes de spécifications	Généralement plate	Plate
Schéma de navigation	Généralement séquentiel	Généralement séquentiel indexé	Parcours de graphes	Généralement séquentiel	Séquentiel
Objectif de la recherche	Booléen ou approximatif	Booléen ou approximatif	Booléen ou approximatif	Booléen ou approximatif	Booléen ou approximatif
Critère de pertinence	Informel	Informel	Formel	Formel	Formel
Critère de présélection	Pas de présélection	Pas de présélection	Une relaxation du critère de pertinence	Pas de présélection	Une relaxation en utilisant des méta-connaissances/ Eventuellement une autre technique de recherche de composants

3.2 Applications

Cette technique de recherche de composants peut également être exploitée dans d'autres domaines d'applications comme : la réingénierie des systèmes d'information (cf. section 3.2.1), l'assistance pour la réutilisation des composants d'une BDC (cf. section 3.2.2) ou la vérification de la cohérence des actions des ingénieurs d'applications (cf. section 3.2.3).

3.2.1 Réingénierie des systèmes d'information à base de composants

Dans cette section, nous ne nous intéressons plus aux bases de composants uniquement du point de vue recherche de composants, mais aussi du point de vue construction de bases de connaissances que nous exploitons pour la réingénierie de systèmes d'information (Khayati, 2005a). Nous adoptons les processus de génération de spécifications sources et cibles présentés pour la détection des composants d'une BDC dans des systèmes d'information existants. Les composants d'une BDC sont utilisés (cf. figure 70) pour la génération de spécifications cibles pour construire une base de connaissances qui résume « l'essence » des composants. L'essence d'un composant est la spécification minimale représentant la sémantique d'un composant. Si les éléments de modélisation représentés par l'essence d'un composant changent de manière à ne plus lui correspondre, on dit alors que la sémantique du composant a changé. Après la génération des spécifications cibles, un ingénieur de composants peut intervenir et modifier la spécification pour qu'elle corresponde au mieux à la sémantique des composants. Une fois la base de connaissances construite, alors elle peut être exploitée pour faire de la réingénierie en appliquant les requêtes cibles sur les requêtes sources extraites des systèmes d'information.

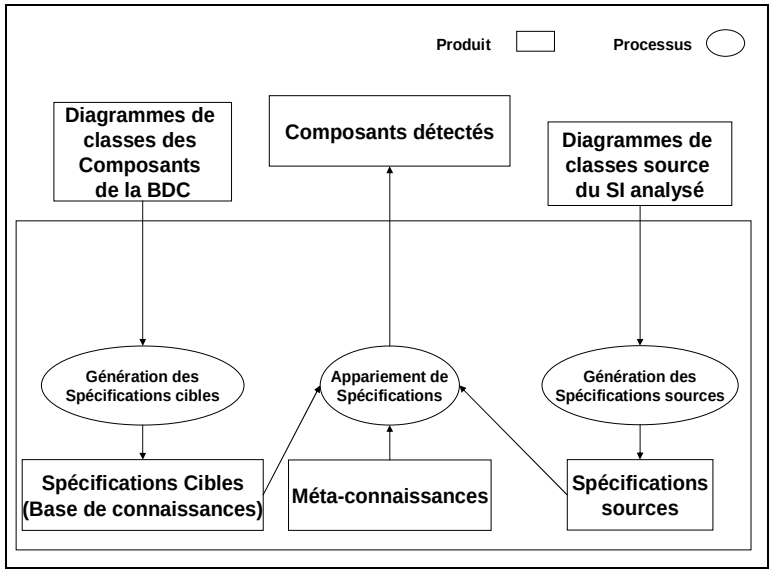


Figure 70. *Processus de réingénierie d'un système d'information.*

3.2.2 Réutilisation des composants d'une BDC

Une autre variante de l'application présentée dans la section précédente est l'intégration de cette technique dans l'éditeur UML utilisé par les ingénieurs d'applications ou de composants. Lorsqu'un ingénieur de composants construit un nouveau composant pour l'intégrer à une BDC, le système pourra détecter des ressemblances avec des composants déjà intégrés à la

base, ce qui facilitera l'intégration du nouveau composant en générant automatiquement certaines parties de sa documentation. De même, si un ingénieur d'applications construit un diagramme UML dont certaines structures ressemblent à des composants de la base, le système pourra lui proposer de réutiliser les composants testés et validés et ainsi promouvoir la réutilisation des composants de la BDC.

3.2.3 Vérification de la cohérence des actions des ingénieurs d'applications

La technique proposée peut finalement être utilisée également pour vérifier la cohérence des modifications appliquées à un composant pendant et après sa réutilisation. Il est possible lors de plusieurs modifications successives d'un composant, que le problème original résolu par le composant soit modifié, donc que l'intégrité de sa réutilisation soit violée. Notre système pourra vérifier après chaque modification du composant sa cohérence vis-à-vis de la définition de son « essence » dans la BDC et informer l'ingénieur d'applications. Le cas est fréquent lors de la construction d'un composant par imitation d'un patron. Après des modifications successives, ce composant peut ne plus correspondre à l' « essence » du patron initial.

VIII. Expérimentations

Ce chapitre présente un prototype d'un environnement d'aide à la réalisation et à l'utilisation de composants. Ce prototype permet de valider des propositions de cette thèse. Dans les sections suivantes, nous présenterons : une architecture générale de l'environnement, un atelier de gestion et d'application de patrons, un système de gestion de bases descriptives de composants, un système de recherche de composants.

1. Architecture générale

Le prototype d'environnement d'aide à la réalisation et à l'utilisation de composants présenté dans ce chapitre est composé de trois sous-systèmes (cf. figure 71) : l'atelier AGAP (Atelier de Gestion et d'Application de Patrons), un système de gestion de bases descriptives de composants et un système de recherche de composants.

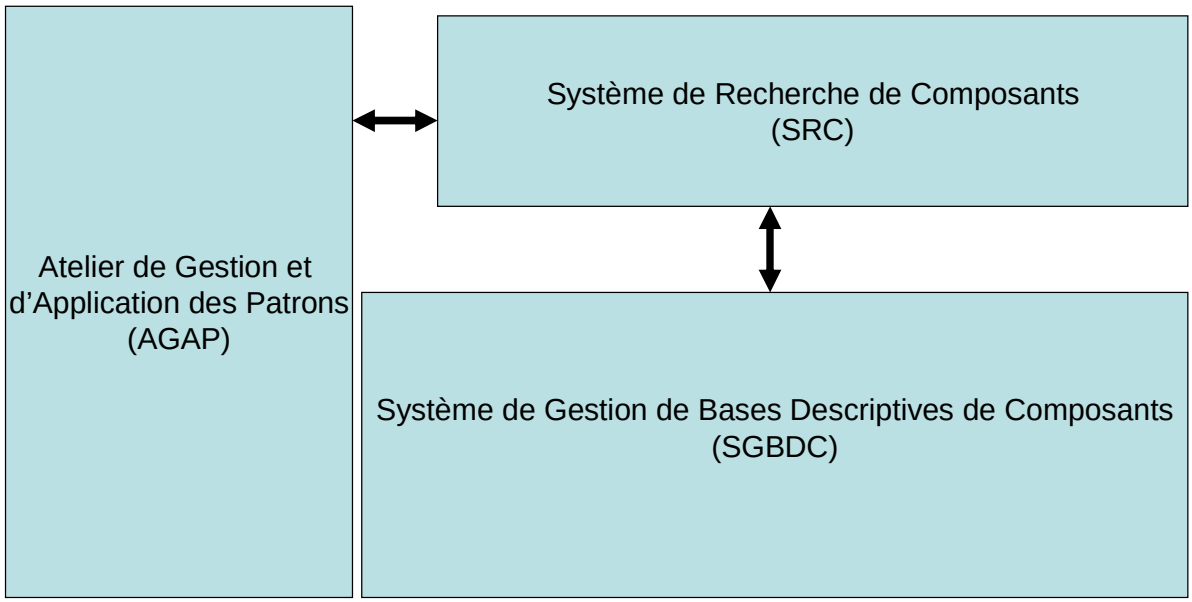


Figure 71. *Architecture générale du prototype.*

Nous présentons dans la suite de ce chapitre chacun des sous-systèmes de notre environnement.

2. AGAP : un Atelier de Gestion et d'Application des patrons

AGAP est développé au sein de l'équipe SIGMA[15] du laboratoire LSR-IMAG[16] (Conte, 2001) (Conte, 2001a) (Descombe-Perrière, 2003) (Tastet, 2004) (Arnaud, 2004) (Jausseran, 2005) permettant la réutilisation dans l'ingénierie des systèmes d'information. Deux processus cohabitent dans l'atelier AGAP : l'ingénierie des patrons (*for reuse*) et l'ingénierie des systèmes d'information (*by reuse*). L'ingénierie des patrons permet la capitalisation des patrons décrivant les savoirs (solutions de problèmes) et les savoir-faire (démarches de développement). L'ingénierie des systèmes d'information permet la construction de systèmes d'information en exploitant les savoirs et les savoir-faire capitalisés dans les patrons archivés dans l'atelier AGAP. La figure 72 montre le cycle de vie d'un système de patrons dans l'atelier AGAP.

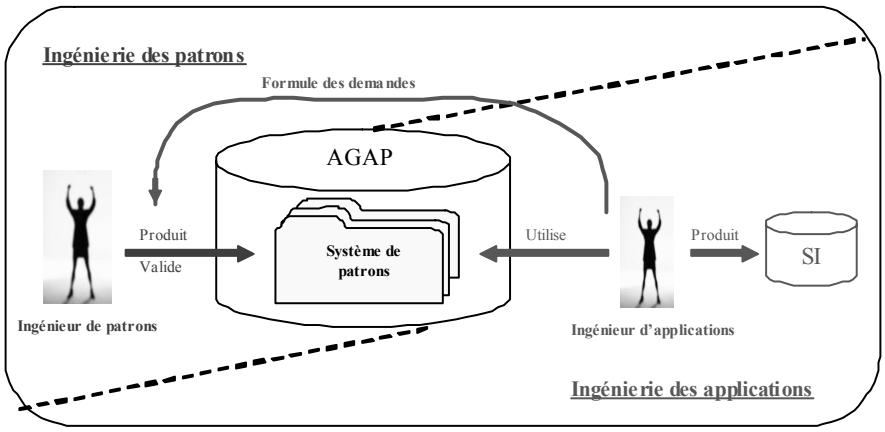


Figure 72. *AGAP, cycle de vie d'un système de patrons (Jausseran, 2005).*

Nous présentons dans la suite de cette section une brève description de l'atelier AGAP qui traite les aspects suivants : Les différents types d'utilisateurs, les fonctionnalités offertes et l'architecture fonctionnelle de l'atelier.

2.1 Les acteurs de l'atelier AGAP

L'atelier AGAP a potentiellement trois types d'acteurs (cf. figure 73): *Ingénieur de patrons*, *Ingénieur d'applications* et *Administrateur*.

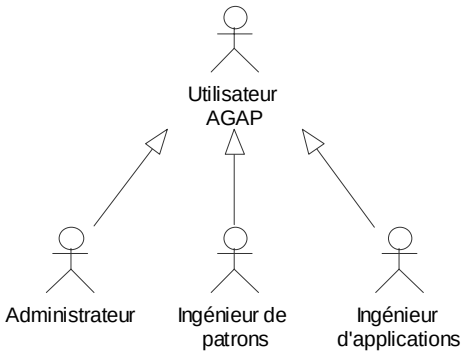


Figure 73. *Les acteurs de l'atelier AGAP.*

[15] http://www-lsr.imag.fr/sigma.html

[16] http://www-lsr.imag.fr/

2.1.1 Ingénieur de patrons

L'*ingénieur de patrons* a pour rôle de concevoir et de gérer des systèmes de patrons permettant d'offrir des solutions génériques à un ensemble de problèmes récurrents. Un ingénieur de patrons a besoin d'un formalisme de patrons pour pouvoir décrire les problèmes et leurs solutions. Il a aussi besoin d'identifier les relations qui existent entre les patrons pour pouvoir guider et faciliter leur utilisation (cf. section 2.5 chapitre II). AGAP doit fournir à l'*ingénieur de patrons* toutes les fonctionnalités lui permettant de gérer des formalismes et des systèmes de patrons. La figure 74 présente les principaux cas d'utilisation de l'ingénieur de patrons.

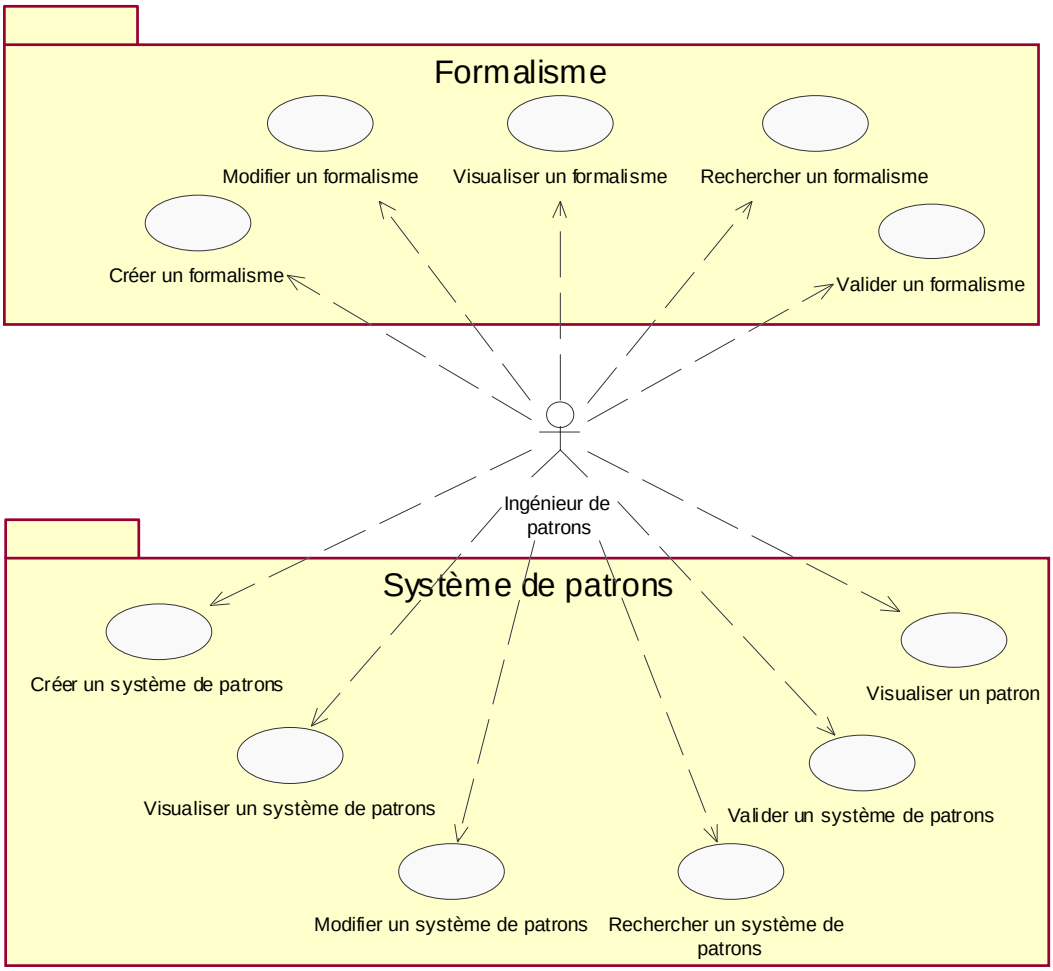


Figure 74. *Principaux cas d'utilisation de l'ingénieur de patrons (Tastet, 2004).*

2.1.2 Ingénieur d'applications

Un *ingénieur d'applications* utilise les systèmes de patrons conçus par l'ingénieur de patrons. Il doit pouvoir consulter, imiter, adapter et intégrer des patrons pour modéliser et concevoir un système d'information.

- **Imitation et adaptation :** l'imitation consiste en une duplication suivie d'une adaptation de la solution du patron au contexte spécifique du problème. Adapter un patron revient à appliquer des opérateurs de renommage, redéfinition, ajout et suppression sur les attributs, les méthodes et les associations afin d'arriver à une solution adaptée au nouveau contexte.
- **Intégration :** une fois le processus d'imitation effectué, l'ingénieur d'applications dispose de plusieurs modèles de solutions qu'il faut intégrer pour aboutir à un modèle cohérent du système à concevoir.

- **Traçabilité :** lors de la conception d'un système d'information, il est important de retrouver la trace des patrons utilisés et leur intégration au sein du SI.

La figure 75 présente les principaux cas d'utilisation de l'*ingénieur d'applications*.

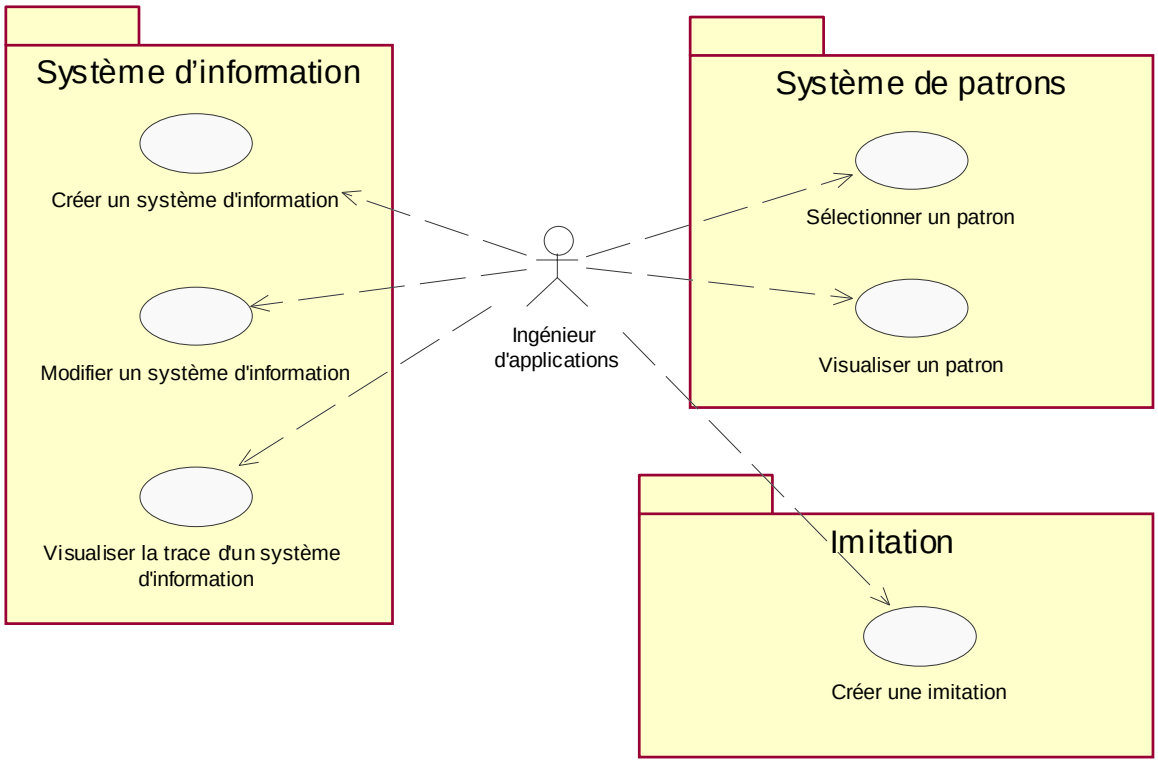


Figure 75. *Principaux cas d'utilisation de l'ingénieur d'applications (Tastet, 2004).*

2.1.3 Administrateur

Un *administrateur* gère les activités suivantes d'AGAP :

- **Gestion des utilisateurs :** Création et gestion des comptes utilisateurs AGAP. Attribution des droits d'accès aux différentes fonctionnalités d'AGAP selon le type d'utilisateur.
- **Gestion des serveurs :** gestion des différents serveurs permettant le fonctionnement d'AGAP (Tomcat, Postgres, etc.).
- **Suppression d'entités validées :** une fonctionnalité importante de l'*administrateur* est la possibilité de supprimer des entités validées (type de champ, domaine & technologie, formalisme, système de patrons).

La figure 76 présente les principaux cas d'utilisation de l'*administrateur*.

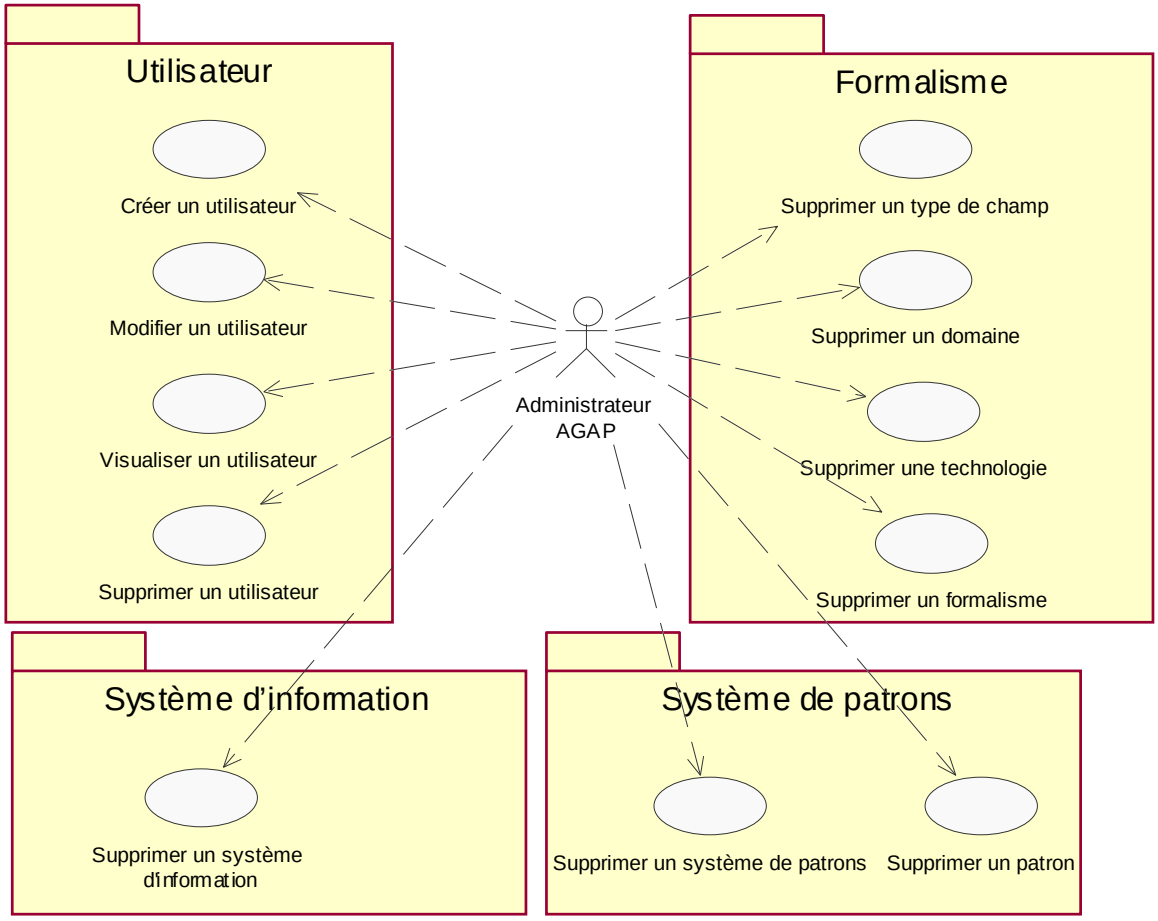


Figure 76. *Principaux cas d'utilisation de l'administrateur (Tastet, 2004).*

2.2 Architecture fonctionnelle

La version actuelle de l'atelier AGAP se concentre sur la mise en œuvre des fonctionnalités attendues par l'*ingénieur de patrons*. Il est décomposé en 9 composants métier :

- deux composants sur les métiers de base :
  - **composant Formalisme** : pour la formalisation des patrons
  - **composant Système de patrons** : pour la gestion des patrons
- trois composants auxiliaires :
  - **composant Type de champ** : pour le typage des rubriques du formalisme
  - **composant Domaine et technologie** : pour permettre la classification des patrons
  - **composant Utilisateur** : pour la gestion des utilisateurs d'AGAP
- deux composants pour la recherche de patrons :
  - **composant Extracteur de démarche** : permet la recherche de patrons au travers de la démarche exprimée dans un système de patrons.
  - **composant Catalogue de patrons** : permet de rechercher les patrons d'un systèmes de patrons, par leur nom ou par mots-clés.
- un composant pour l'ingénieur d'applications
  - **composant Système d'information** : permet de concevoir des systèmes d'information
- un composant médiateur entre les composants précédents :
  - **composant Atelier**

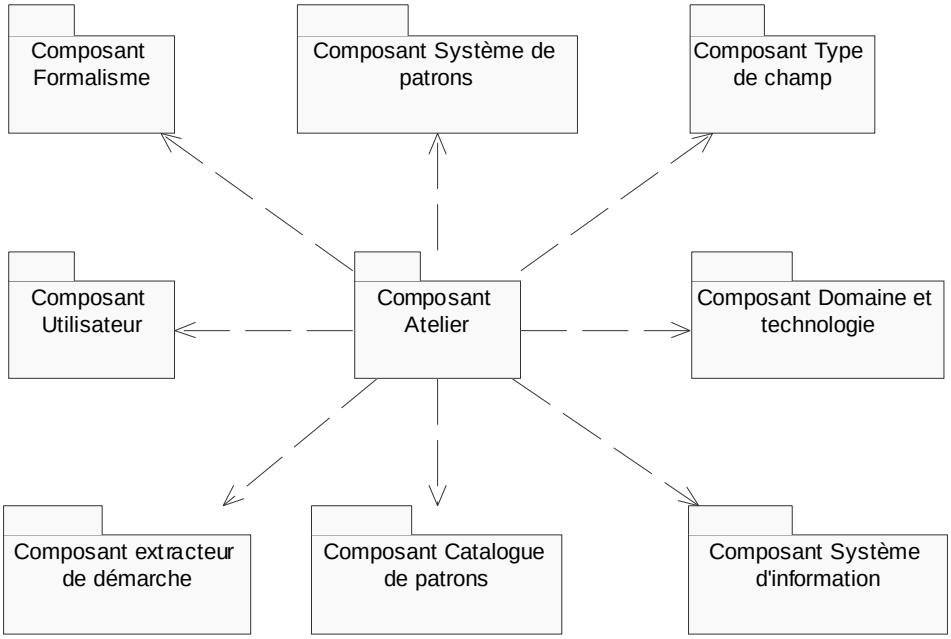


Figure 77. *Architecture fonctionnelle d'AGAP (Tastet, 2004).*

3. SGBDC : un Système de Gestion de Bases Descriptives de Composants

Le système de gestion de bases descriptives de composants réalisé est basé sur le métamodèle *M-Sigma* (cf. chapitre V) et destiné à différents acteurs ayant des besoins différents. Il est implanté sous la forme de plusieurs composants et est constitué de plusieurs modules.

3.1 Les acteurs d'un SGBDC

Trois types d'acteurs (cf. figure 78): l'*Ingénieur de modèles de bases descriptives de composants* (MBDC), l'*Ingénieur de bases descriptives de composants* (BDC) et l'*Administrateur*.

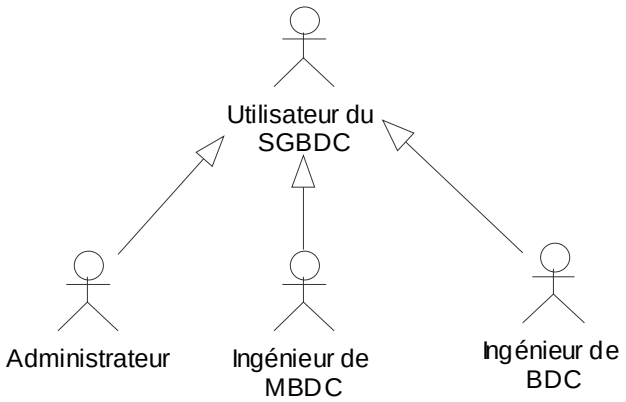


Figure 78. *Les acteurs d'un SGBDC.*

3.1.1 Ingénieur de MBDC

Un SGBDC gérant un ensemble de BDC, chaque BDC est une instance d'un modèle de bases descriptives de composants. L'*ingénieur de MBDC* a la charge de créer, modifier, valider, cloner et supprimer un MBDC. La figure 79 résume les principaux cas d'utilisation réalisés par un *ingénieur de MBDC*.

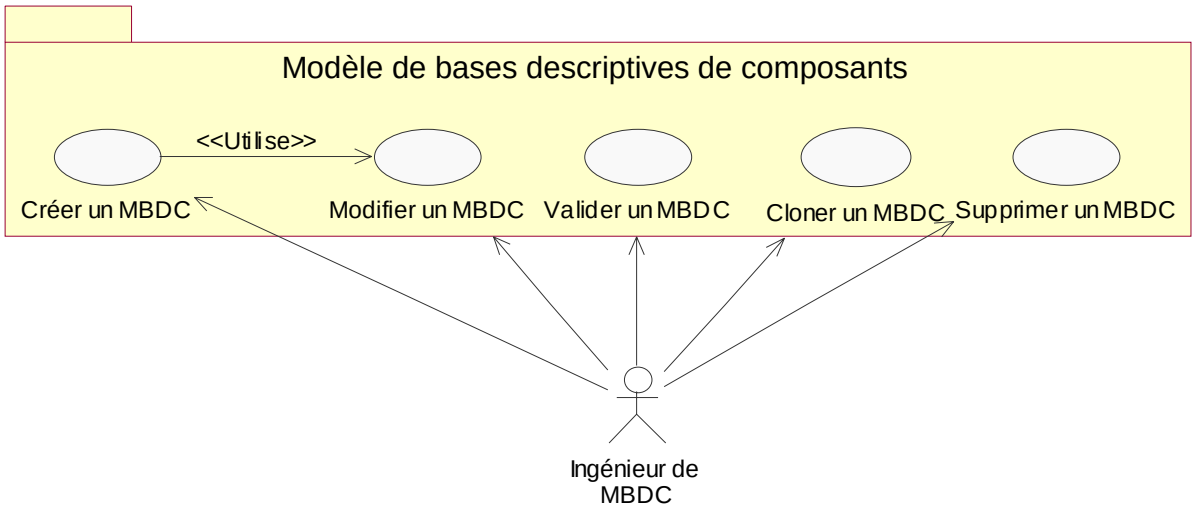


Figure 79. *Les cas d'utilisation d'un ingénieur de MBDC.*

3.1.2 Ingénieur de BDC

L'*ingénieur de BDC* doit pouvoir réaliser les opérations suivantes :

- création de nouvelles bases descriptives de composants à partir des modèles saisis et validés par l'*ingénieur de MBDC* ;
- suppression d'une base descriptive de composants existante ;
- exploration du contenu d'une base descriptive de composants par navigation ;
- instanciation, modification et suppression de descriptions de composants gérés par une base descriptive de composants ;
- instanciation, modification et suppression de relations inter-descriptions de composants gérés par une base descriptive de composants ;
- ajout, modification et suppression de composants gérés par une base descriptive de composants.

La figure 80 résume les principaux cas d'utilisation réalisés par un *ingénieur de BDC*.

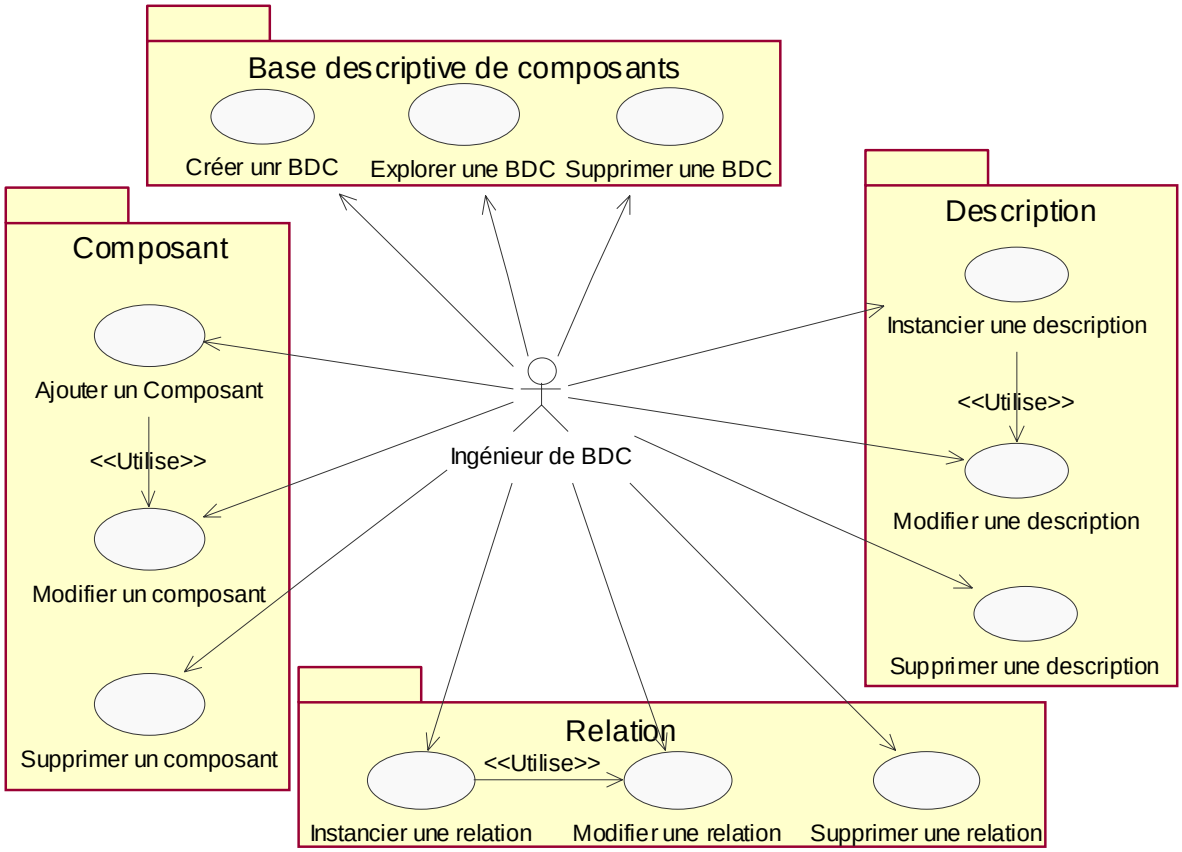


Figure 80. *Les cas d'utilisation d'un ingénieur de BDC.*

3.1.3 Administrateur

Un *administrateur* est responsable des opérations suivantes :

- gestion des utilisateurs et de leurs droits d'accès aux différentes fonctionnalités offertes par un SGBDC ;
- opérations de maintenance des serveurs (par exemple du serveur de bases de données) ;
- La gestion des extensions et de leur paramétrage (par exemple gestion des types de données (*DataType*)).

La figure 81 résume les principaux cas d'utilisation réalisés par un *administrateur*.

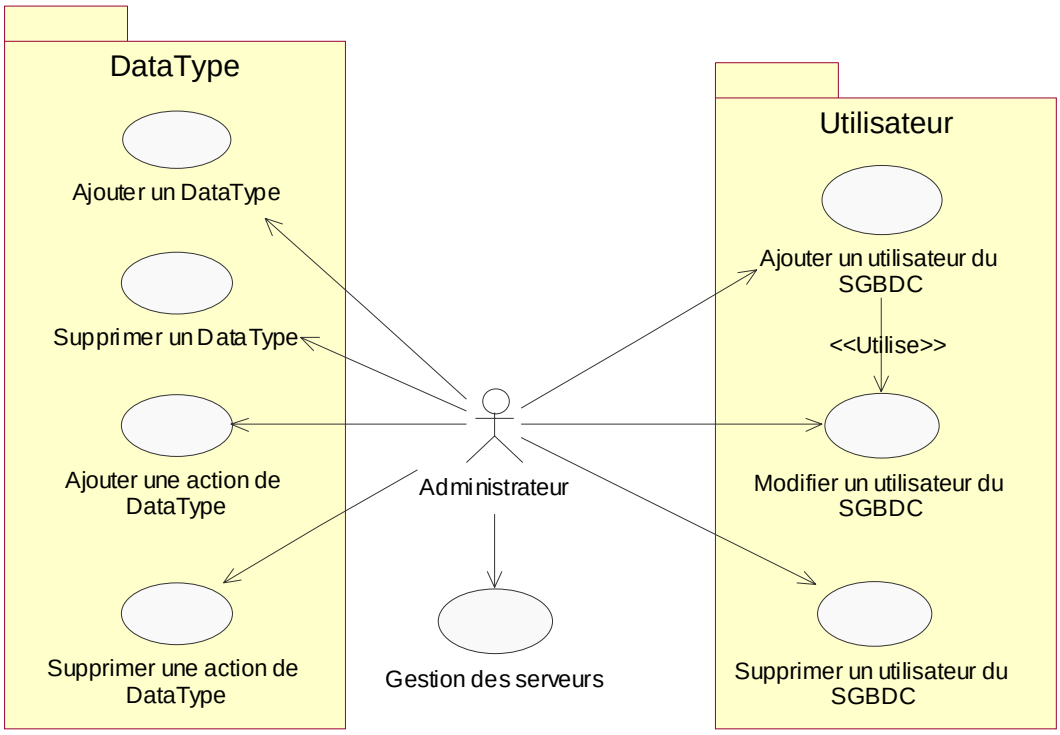


Figure 81. *Les cas d'utilisation d'un Administrateur.*

3.1.4 Scénario d'utilisation d'une BDC

La figure 82 décrit l'ordre d'intervention des différents types d'utilisateurs du SGBDC. Au début, l'*administrateur* installe les types de données (*DataType*) supportés par le SGBDC et paramètre les différents types d'opérations applicables sur ces types. Ensuite, il crée les différents utilisateurs en leur affectant leurs droits d'accès respectifs. L'ingénieur de MBDC a pour rôle la création et la gestion des MBDC. Enfin l'ingénieur de BDC a pour rôle d'instancier de nouvelles BDC en utilisant les MBDC créés par l'*ingénieur de MBDC* et d'alimenter les BDC avec de nouveaux composants et leurs descriptions.

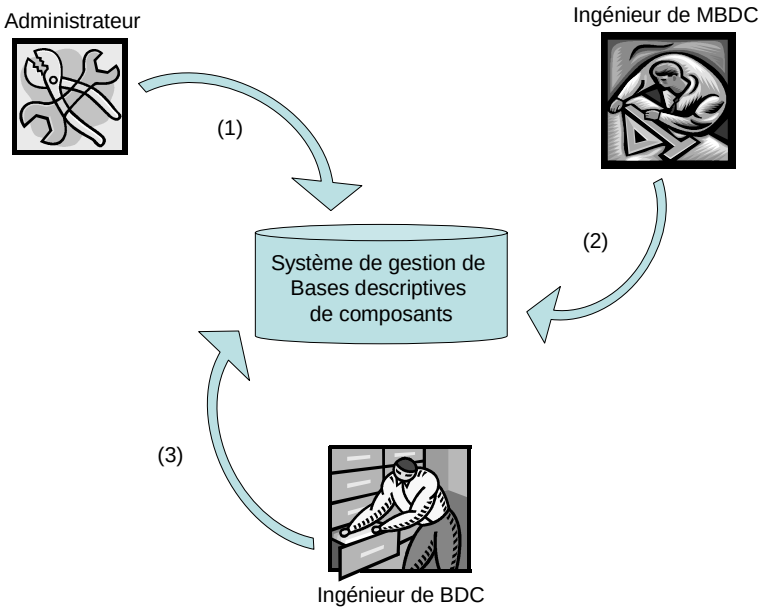


Figure 82. *Séquence d'utilisation d'un SGBDC.*

3.2 L'architecture fonctionnelle

La présentation des cas d'utilisation des trois types d'acteurs d'un SGBDC (cf. figure 79, figure 80, figure 81) a montré les principaux composants du SGBDC. La figure 83 présente une cartographie des principaux composants du SGBDC.

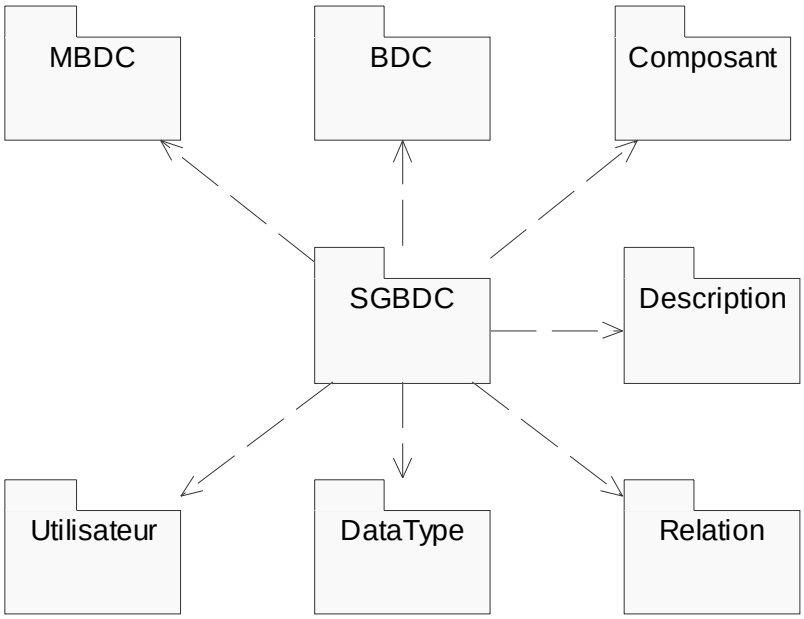


Figure 83. *Cartographie des principaux composants du SGBDC.*

3.3 Architecture technique

Le prototype de SGBDC que nous avons réalisé est basé sur l'architecture deux tiers. Le premier tiers contient la partie présentation et la partie logique du SGBDC alors que le deuxième tiers contient la base de données. Il est composé de 5 modules.

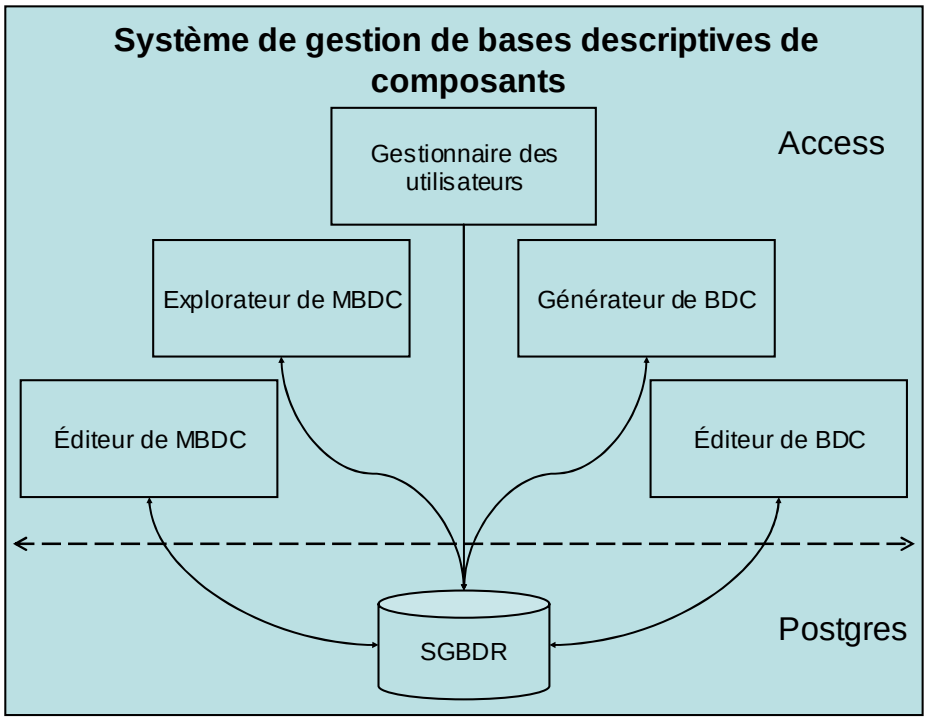


Figure 84. *Architecture technique du système de gestion de bases descriptives de composants.*

Nous avons choisi de développer les 5 modules clients en Microsoft Access pour sa facilité en ce qui concerne le développement rapide des interfaces utilisateurs. Ce qui nous a permis un gain important en temps de développement. Pour la partie base de données, nous avons opté pour le SGBD Postgres à l'inverse de Mysql il gére les contraintes d'intégrité.

La figure 85 et la figure 86 montrent respectivement les modules Éditeur et Explorateur de modèles de bases descriptives de composants. Ces deux modules sont utilisés par l'*ingénieur de MBDC*.

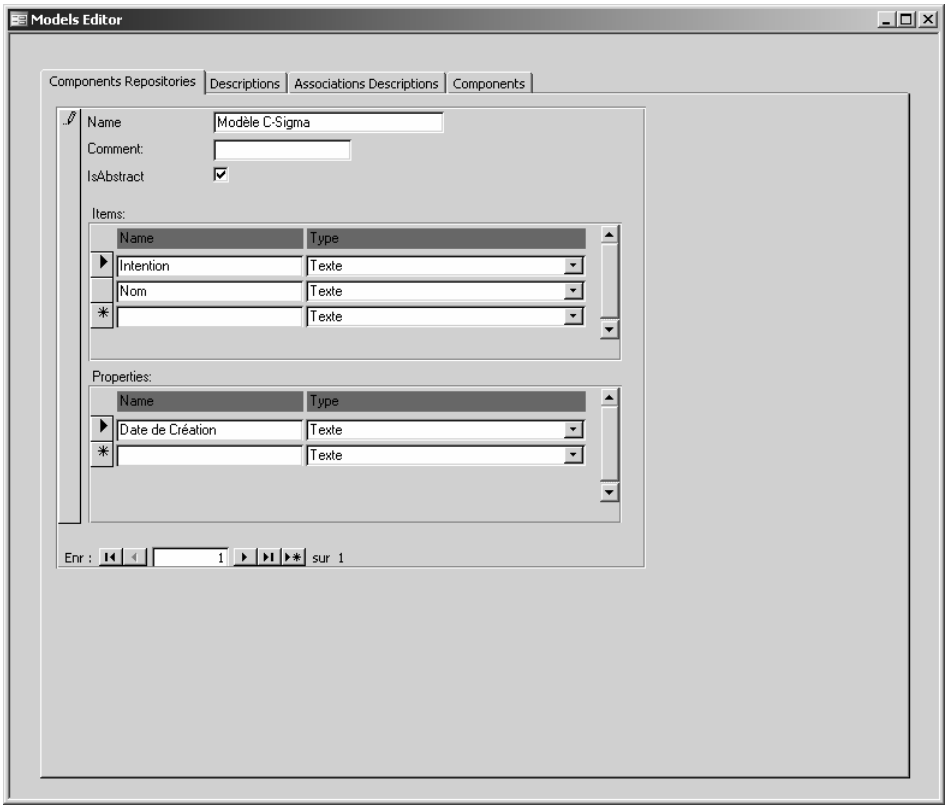


Figure 85. *Capture d'écran du module Éditeur de MBDC.*

L'éditeur de MBDC présente l'ensemble des métaclasses concrètes du métamodèle *M-Sigma*. L'explorateur de MBDC permet aux ingénieurs de MBDC et aux ingénieurs de BDC de naviguer dans les modèles de bases descriptives de composants.

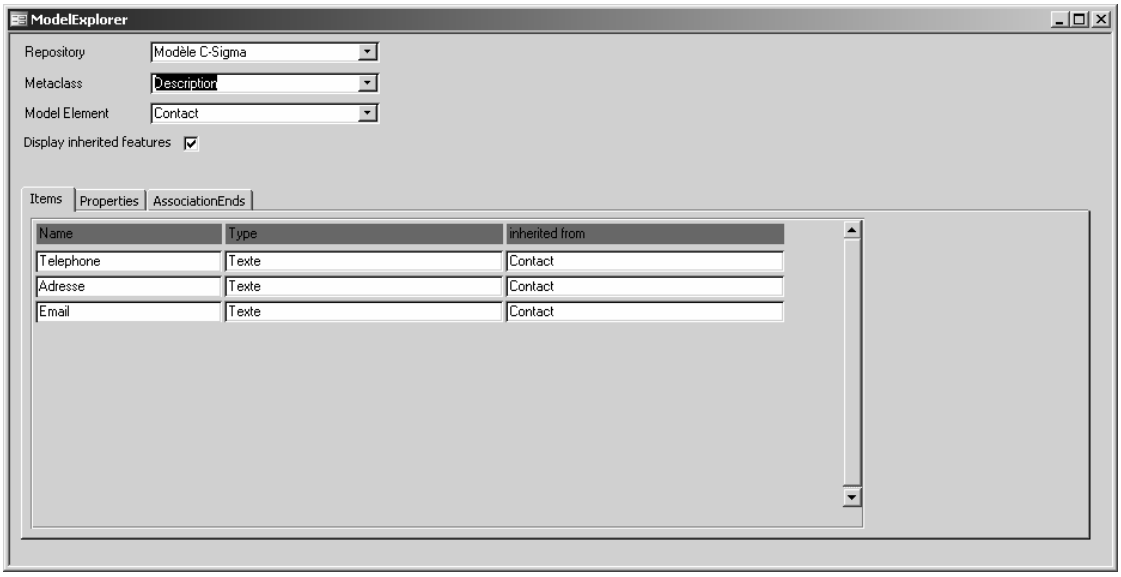


Figure 86. *Capture d'écran du module Explorateur de MBDC.*

La figure 87 et la figure 88 montrent respectivement les modules Générateur et Éditeur de bases descriptives de composants. Ces deux modules sont utilisés par l'*ingénieur de BDC*.

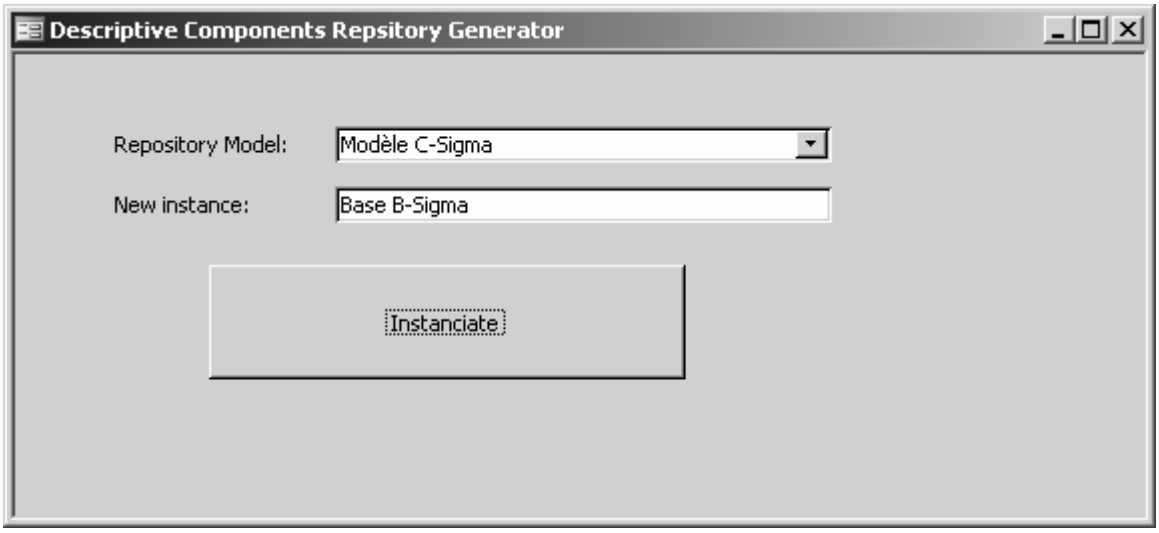


Figure 87. *Capture d'écran du module Générateur de BDC.*

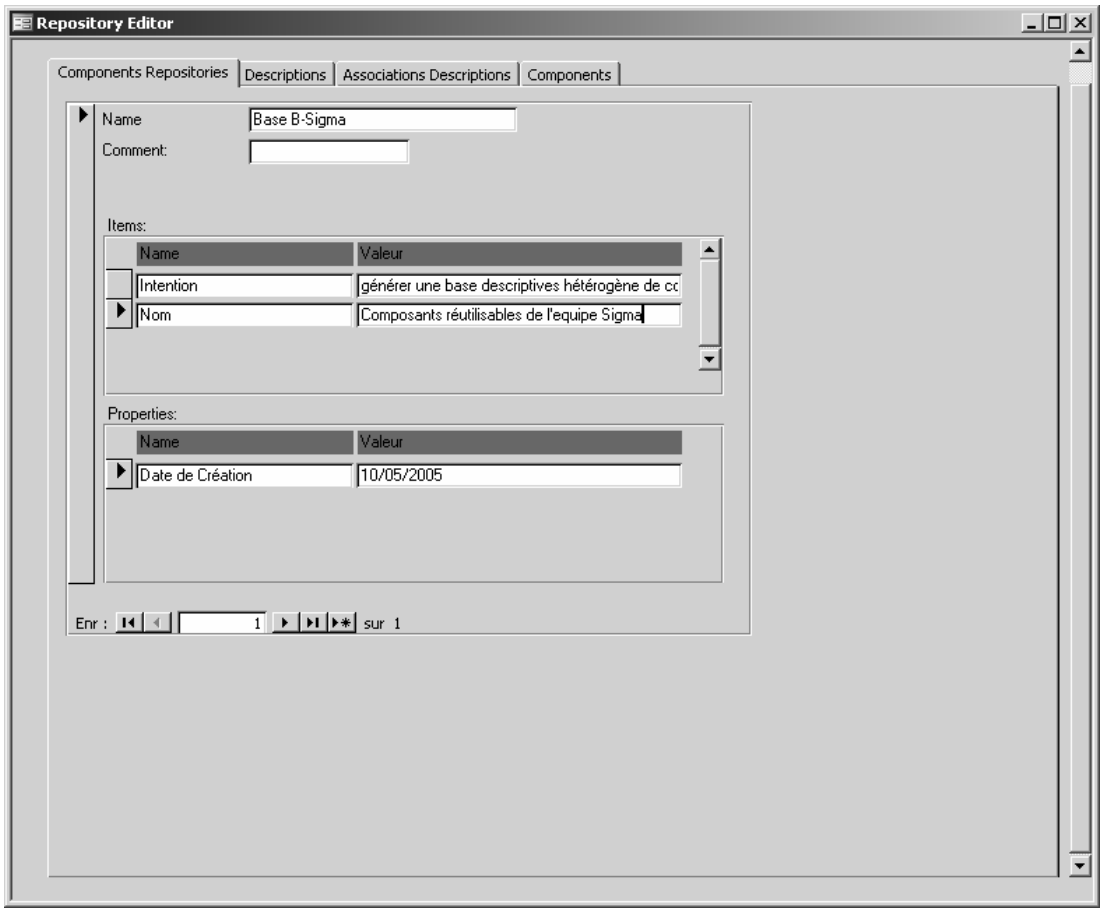


Figure 88. *Capture d'écran du module Éditeur de BDC.*

L'éditeur de bases descriptives de composants permet d'instancier les entités concrètes (descriptions, relations, etc.) définies dans les modèles de bases descriptives de composants.

La figure 89 montre le module Gestionnaire des utilisateurs exploité par l'*administrateur* pour la gestion des droits utilisateurs

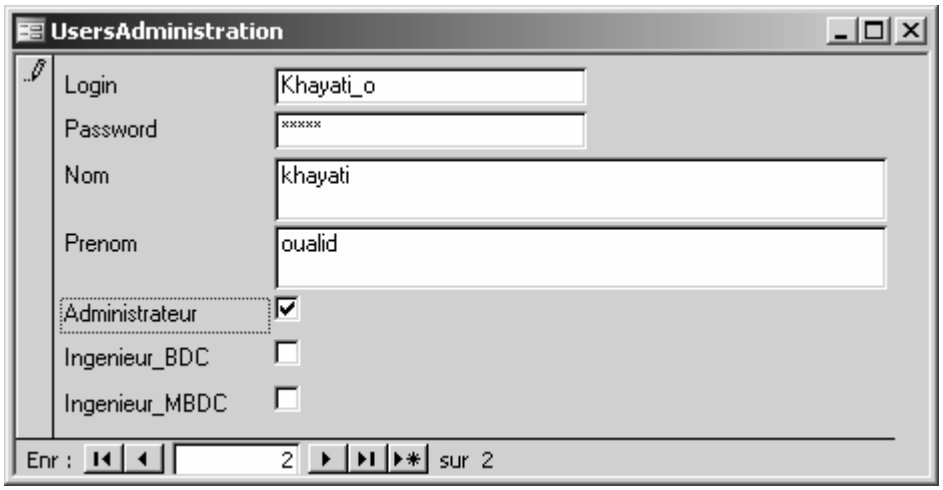


Figure 89. *Capture d'écran du module Gestionnaire d'utilisateurs.*

3.4 Conclusion

L'implantation partielle du métamodèle *M-Sigma* dans notre prototype a pour but de montrer la faisabilité des principales propositions faites dans les chapitres IV et V. Des concepts comme les contraintes (*Constraint*), les types de données (*DataType*) ou les composants connectables ne sont pas réalisés dans l'implantation actuelle, mais seront intégrés dans les prochaines versions.

Lors d'une refonte de notre prototype, nous adopterons une architecture quatre tiers (cf. figure 90). La modèle et persistance nous permettra d'avoir une indépendance entre la représentation des données (SGBDR) et la structure des données en mémoire (instances de *M-Sigma*), ce qui facilitera l'implantation du concept de Contraintes (*Constraint*). La couche présentation permettra la séparation de la gestion de l'interface utilisateur de la logique de traitement des requêtes utilisateurs.

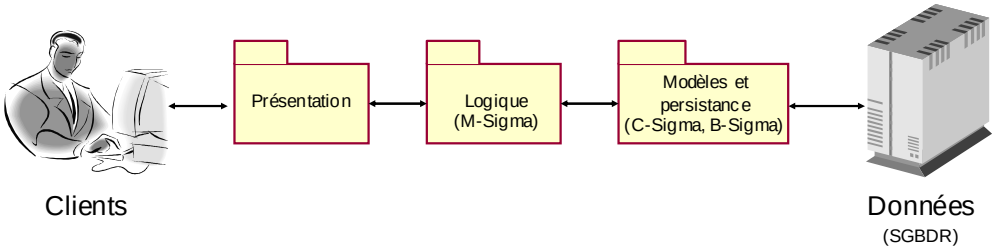


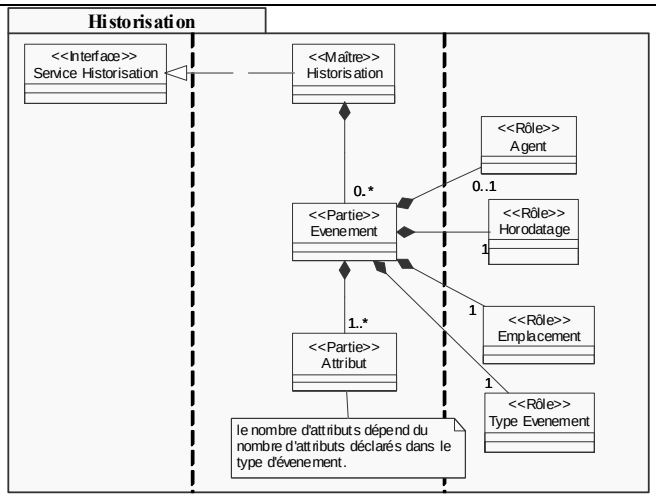
Figure 90. *Architecture quatre tiers.*

Des composants issus de la coopération avec notre partenaire industriel Actoll[17] dans le cadre du projet Initiative Centr'Actoll[18] ont été intégrés dans notre outil. À titre d'illustration, le tableau 17 montre l'exemple de la représentation du composant métier *Historisation* au niveau Analyse.

[17] http://www.actoll.com/

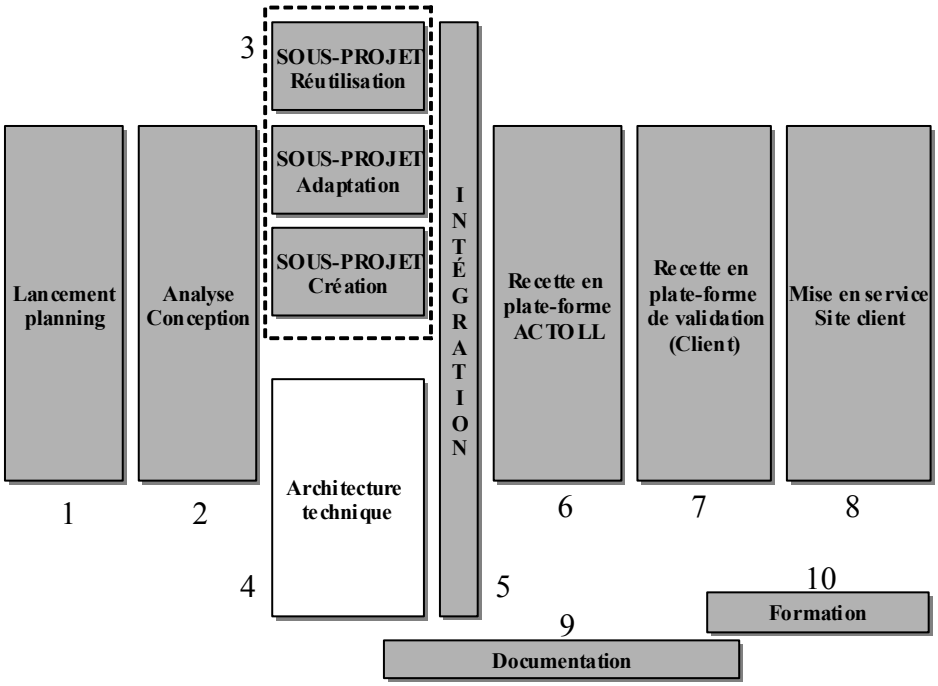
[18] http://www-adele.imag.fr/Les.Groupes/centractoll/

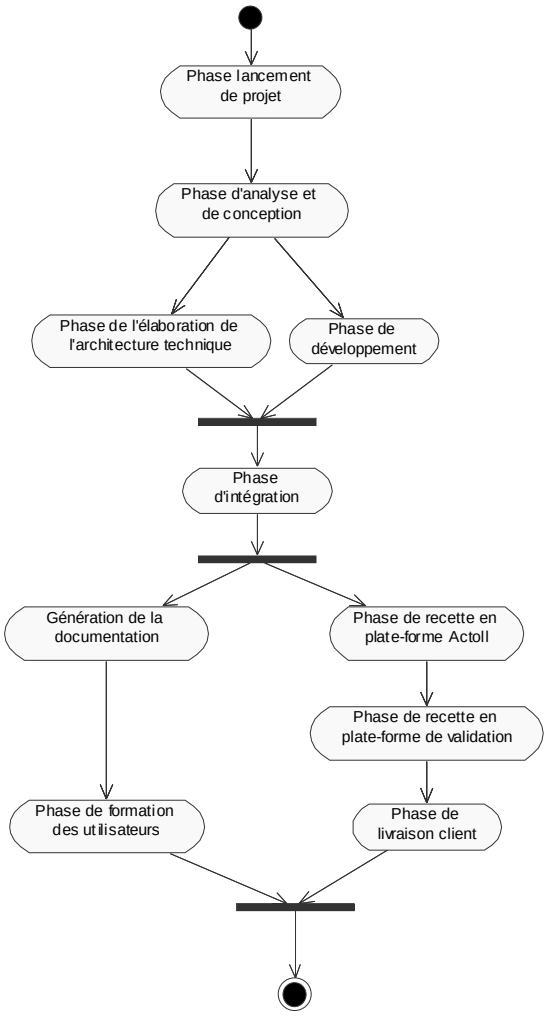
Tableau 17. *La représentation du composant métier Historisation au niveau Analyse.*

Nom	Historisation
Description	Le composant *Historisation* gère l'historisation des événements générés par les autres composants.
Utilise	Le composant *Historisation* utilise d'autres composants via les rôles suivants : - Agent : c'est un rôle du composant *Agent*. Ce rôle indique l'agent qui a généré l'évènement. - Horodatage : c'est un rôle du composant *Horodatage*. Ce rôle fournit l'heure et la date de l'événement. - Emplacement : c'est un rôle du composant *Emplacement*. Ce rôle fournit l'emplacement où l'événement est déclenché. - Type Evénement : c'est un rôle du composant *TypeEvénement*. Il donne des informations sur le type des évènements par exemple les noms et les types des attributs des événements.
Utilisé par	Le composant *Historisation* est utilisé par les composants suivants : - Carte : le composant *Historisation* joue le rôle de *Historisation carte*. - Client : le composant *Historisation* joue le rôle de *Historisation client*. - Agent : le composant *Historisation* joue le rôle de *Historisation Agent*. - Contrat : le composant *Historisation* joue le rôle de *Historisation contrat*.
Modèle	
Participants	Le composant *Historisation* contient les classes suivantes : - Historisation : la classe maître du composant *Historisation.* - Evénement : une classe partie du composant *Historisation*. Cette classe est instanciée chaque fois qu'une notification d'événement arrive au composant *Historisation*. - Attribut : une classe partie du composant *Historisation*. Cette classe est instanciée lors de l'instanciation de la classe *Evénement*. Le nombre d'attributs dépend du nombre d'attributs déclarés dans le type d'événements.

Dans le cadre de cette coopération, le processus de conduite de projet Actoll a été formalisé sous la forme d'un système de patrons. Le tableau 18 montre le patron processus « Guide de conduite de projet ». Le système de patrons plus complet se trouve dans l'annexe G.

Tableau 18. *Le patron processus « Guide de conduite de projet Actoll ».*

Nom	Guide de conduite de projet ACTOLL
Contexte	patron « *Cahier des charges* »
Problème	Ce patron processus permet de guider le chef de projet dans le déroulement d'un projet de développement de SI pour un client Actoll.
Solution démarche	Globalement, un projet de développement d'un système d'information pour un client Actoll se découpe en dix grandes phases :  Les phases 1 et 2 sont exécutées séquentiellement : 1. Phase de lancement de projet (patron « lancement ») 2. Phase d'analyse et de conception (patron « spécification ») Les phases 3 et 4 sont exécutées parallèlement : 3. Phase de développement (patron « sous-projet de réalisation ») 4. Phase d'élaboration de l'architecture technique (patron « architecture technique ») Les phases 5 à 8 sont exécutées séquentiellement : 5. Phase d'intégration (patron « intégration et recette interne ») 6. Phase de recette en plate-forme Actoll (patron « recette en plate-forme Actoll ») 7. Phase de recette en plate-forme de validation (patron « recette en plate-forme de validation »)

	8. Phase de livraison client (patron « mise en service site client ») Les phases 9 et 10 sont exécutées parallèlement : 9. Génération de la documentation : la génération de la documentation se fait en même temps que la phase validation 10. Phase de formation des utilisateurs 
Utilise	- Patron « lancement ». - Patron « spécification ». - Patron « sous-projet de réalisation ». - Patron « architecture technique ». - Patron « intégration et recette interne ». - Patron « recette en plate-forme Actoll ». - Patron « recette en plate-forme de validation ». - Patron « mise en service site client ».

Dans l'annexe G, nous proposons également deux autres systèmes de patrons : Fragments de démarche de développement de SI par réutilisation de composants, et Système de patrons pour l'intégration et l'adaptation des composants métier Symphony. La figure 91 montre la cartographie des composants Actoll (produit et processus) présentés dans l'annexe G.

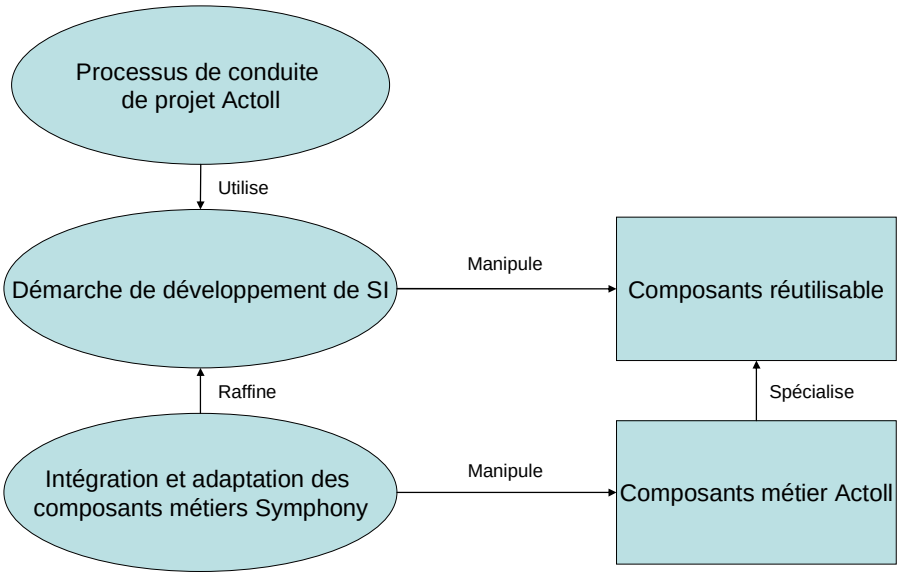


Figure 91. *Cartographie des composants Actoll (produit et processus) présentés dans l'annexe G.*

4. SRC : Système de Recherche de Composants

Le prototype de système de recherche de composants présenté ici réalise le modèle de SRC et la technique de recherche de composants proposés dans le chapitre VII. Les acteurs du système et leurs besoins sont tout d'abord identifiés, puis l'architecture technique adoptée pour la réalisation du prototype est présentée.

4.1 Les acteurs

Le système de recherche de composants que nous présentons dans cette section est destiné aux *ingénieurs d'applications* et aux *ingénieurs de BDC* (cf. figure 92). Un *ingénieur d'applications* recherche dans une BDC les composants qui répondent au mieux à ses besoins pour la construction de nouveaux systèmes d'information. Un *ingénieur de BDC* a besoin de disposer d'une fonctionnalité de recherche et de sélection d'un composant pour comparer le contenu d'une BDC avec les nouveaux composants qu'il veut ajouter.

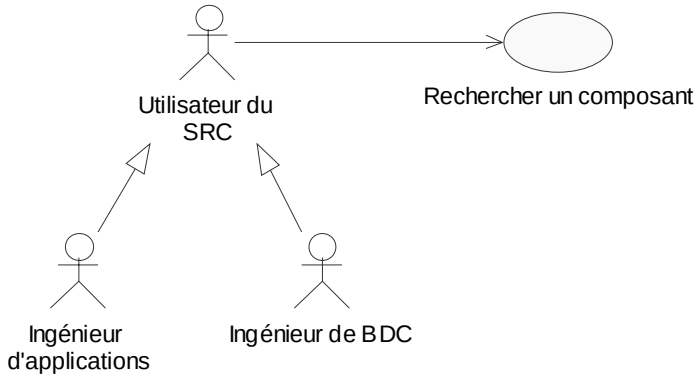


Figure 92. *Les acteurs du SRC.*

4.2 Architecture technique

Le prototype que nous présentons dans cette section implante partiellement le modèle de système de recherche de composants que nous avons proposé dans le chapitreVII. Il traite uniquement les requêtes simples applicables sur un seul type de données qui sont les

diagrammes de classes UML. Ainsi, nous validons la technique de recherche de composants présentée dans le chapitre VII. Le système de recherche de composants adopte l'architecture 2 tiers composée d'un SGBDR et de 5 modules : Éditeur de requêtes, Générateur de spécifications, Moteur d'évaluation de requêtes, Moteur d'indexation, Module d'import.

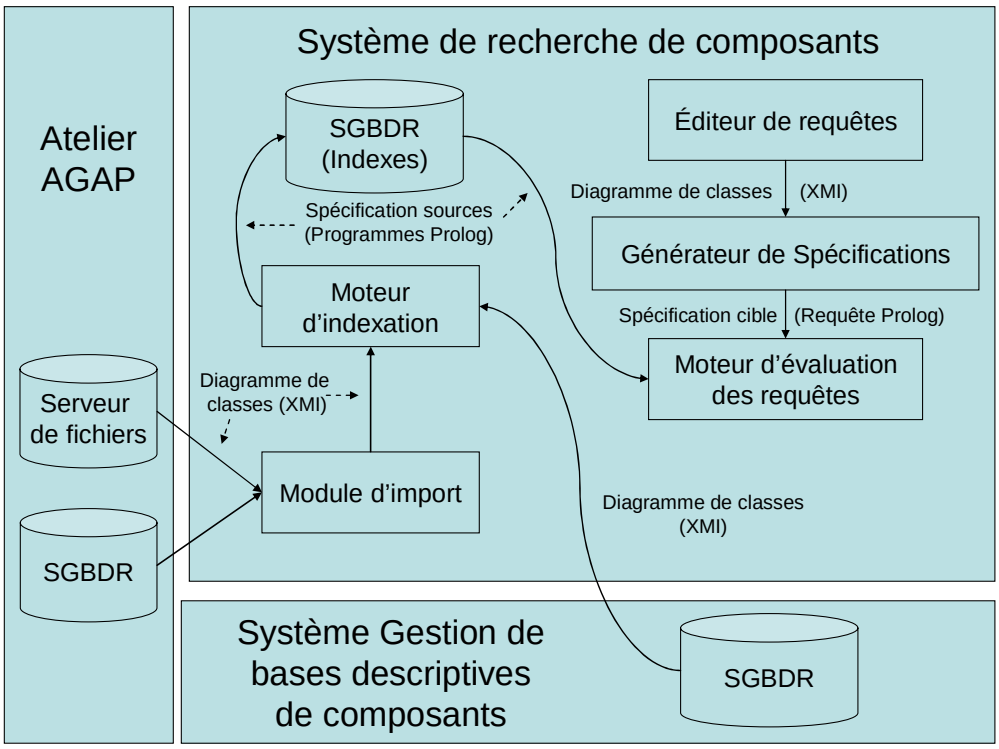


Figure 93. *Architecture technique du système de recherche de composants.*

4.2.1 Éditeur de requêtes

La requête de notre système de recherche de composants se présente sous la forme d'un diagramme de classes UML. L'atelier de génie logiciel *ArgoUML*[19] est utilisé pour la saisie des requêtes (cf. figure 94). Les diagrammes requêtes sont ensuite enregistrés en utilisant le format d'échange XMI.

[19] http://argouml.tigris.org/

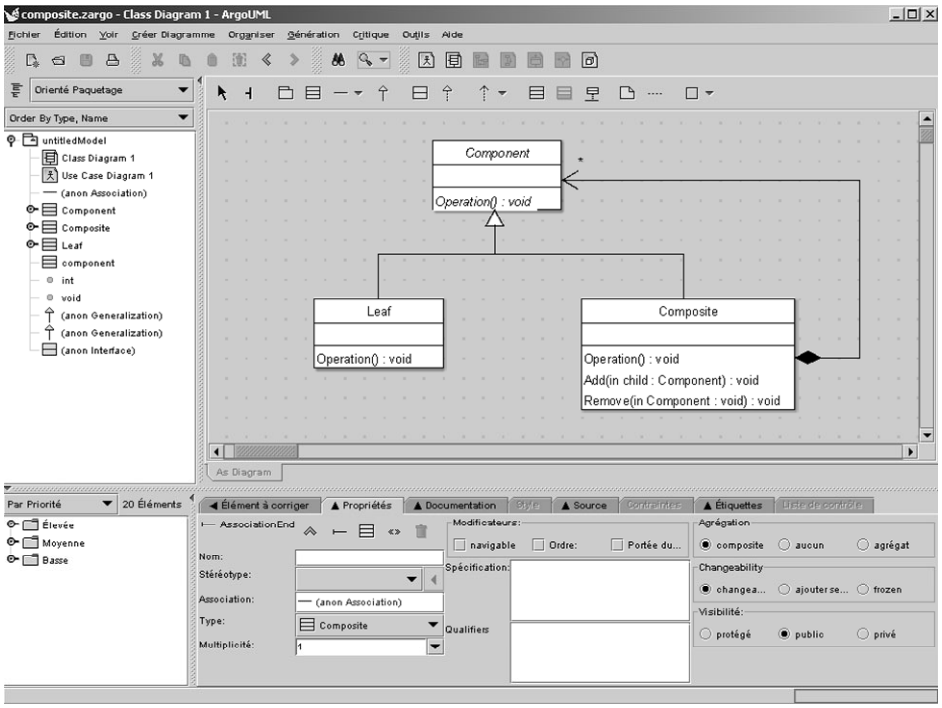


Figure 94. *Capture d'écran de l'AGL ArgoUML comme éditeur de requêtes.*

4.2.2 Générateur de spécifications

Le générateur de spécifications a pour rôle de récupérer le diagramme de classes requête et de le transformer en une spécification source. Ensuite, l'utilisateur modifie la spécification source en une spécification cible en suivant le processus décrit dans la section 2.2 du chapitre VII. La transformation peut être effectuée manuellement dans le champ textuel ou de manière assistée par le formulaire en bas de page (cf. figure 95).

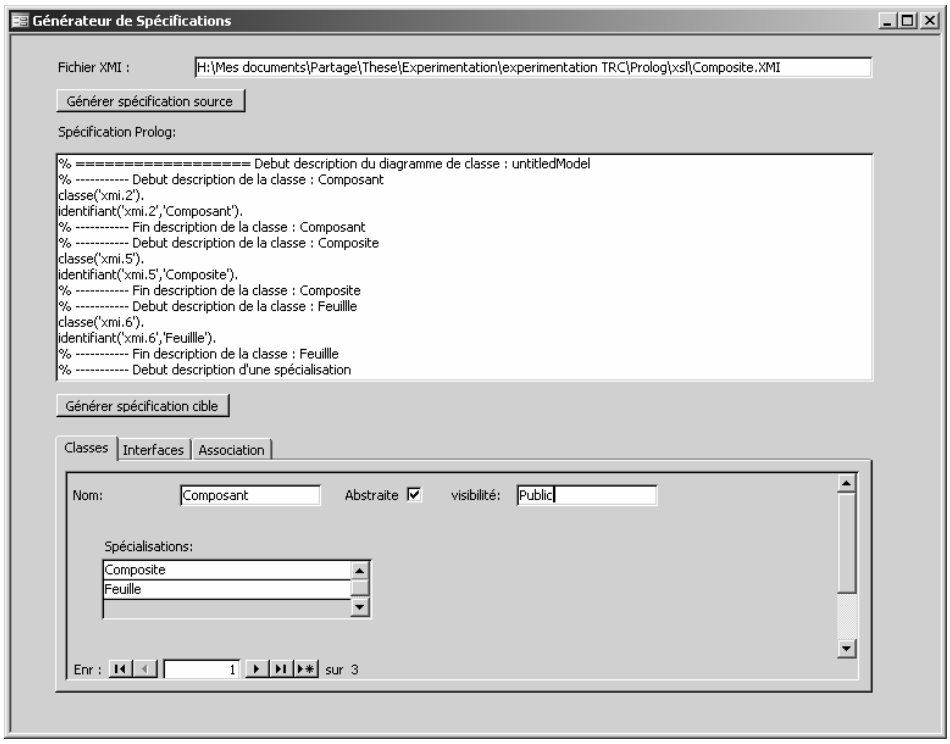


Figure 95. *Capture d'écran du générateur de spécifications.*

La transformation d'un diagramme de classes au format XMI par *ArgoUML* en une spécification formelle source en Prolog s'effectue grâce à une transformation XSLT (cf. figure 96) dans l'outil Altova XMLSPY[20].

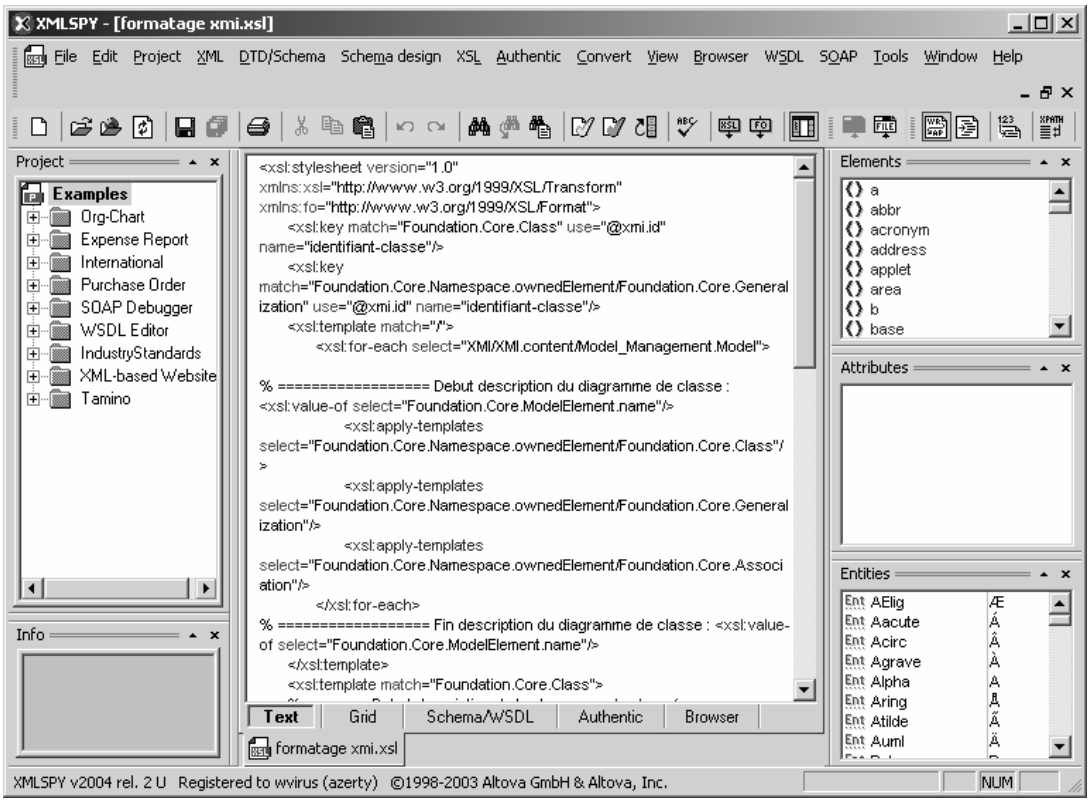


Figure 96. *Transformation XSLT dans l'outil XMLSPY.*

4.2.3 Moteur d'évaluation des requêtes

Le moteur d'évaluation des requêtes repose sur le moteur de Prolog pour apparier la spécification cible (requête) et les spécifications sources (programmes). Il récupère les spécifications sources stockées dans le SGBDR et les compare à la spécification cible, puis retourne les résultats.

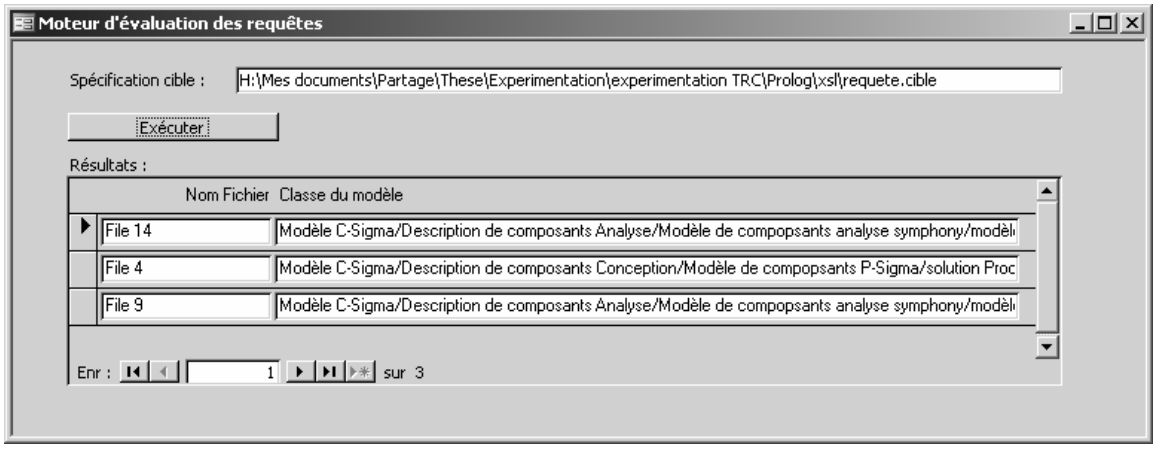


Figure 97. *Capture d'écran du moteur d'évaluation de requêtes*

[20] http://www.altova.com

4.2.4 Moteur d'indexation

Le moteur d'indexation récupère la liste de toutes les rubriques du type diagrammes de classes définies dans le modèle d'une base descriptive de composants. Ensuite, il indexe toutes les instances de ses rubriques avec la spécification source de son diagramme de classes UML. Ces informations sont utilisées par le moteur d'évaluation de requêtes déjà présenté.

4.2.5 Module d'import

Dans le cas d'une BDC archivant des descriptions de patrons, le module d'import récupère les noms de patrons comportant des champs de type diagrammes de classes UML à partir de l'atelier AGAP. Ensuite, les fichiers XMI correspondants aux patrons présélectionnés sont transmis au moteur d'indexation. AGAP stocke les fichiers XMI dans un serveur de fichiers.

4.3 Conclusion

L'implantation du SRC présenté dans cette section valide la technique de recherche de composants et partiellement le modèle de SRC présenté dans le chapitre VII. Le SRC est un outil permettant de faire des recherches de composants décrits dans le SGBDC et dans l'atelier AGAP. La version actuelle de notre prototype exploite une seule technique de recherche de composants et permet d'exprimer uniquement des requêtes simples. La prochaine étape sera l'implantation complète du modèle de SRC que nous avons proposé. Ainsi, le SRC sera capable de gérer plusieurs techniques de recherche de composants.

5. Conclusion générale

Nous avons présenté dans ce chapitre le prototype d'un environnement d'aide à la réalisation et à l'utilisation de composants. Cet environnement inclut trois outils : l'atelier AGAP, le système de gestion de bases descriptives de composants et le système de recherche de composants. L'atelier AGAP est développé par l'équipe SIGMA du laboratoire LSR-IMAG. Nous avons intégré AGAP dans notre environnement en l'interfaçant avec le SRC. Le SRC est ainsi capable d'indexer les composants gérés par AGAP et par le SGBDC. Le SRC exploite également les services de l'AGL open source ArgoUML pour la saisie des requêtes des utilisateurs.

L'expérimentation de notre environnement a été effectuée sur la collection de composants issus de la collaboration avec notre partenaire industriel Actoll (cf. annexe G). Cette expérimentation nous a permis de valider nos trois propositions : la technique de recherche de composants, le métamodèle *M-Sigma* et le modèle *C-Sigma*. Le modèle de SRC a été partiellement validé car nous n'avons pas encore implanté l'aspect des requêtes complexes (composite et multi-techniques de recherche de composants).

IX. Bilan et perspectives

Dans un contexte de réutilisation, un système d'information peut être vu comme un assemblage de composants réutilisables généralement définis comme des unités de conception (de différents niveaux d'abstraction), identifiées par un nom, possédant une structure définie et des directives de conception sous la forme de documentation pour supporter leur réutilisation. La documentation d'un composant illustre le contexte dans lequel il peut être utilisé en spécifiant les contraintes et les autres composants dont il a besoin pour offrir sa solution.

Les bases de composants sont des éléments clés dans les environnements de développement à base de composants et de réutilisation de composants. Malheureusement, mettre simplement à disposition des développeurs une base de composants testés et validés de taille importante n'augmente pas forcément la productivité faute d'une organisation et de techniques de recherche appropriées.

Dans ce mémoire, nous avons abordé cette problématique selon les deux aspects de l'organisation des bases de composants d'une part, et des techniques de recherche de composants d'autre part.

1. Bilan du travail réalisé

Le travail s'est déroulé selon plusieurs axes dont le premier a consisté à établir un état de l'art d'une part sur les approches à bases de composants, d'autre part sur les techniques de recherche de composants.

L'état de l'art sur les approches à composants a permis d'étudier les différents types de modèles de composants susceptibles d'être utilisés dans un processus de développement de systèmes d'information. Pendant chacune des phases du développement (analyse, conception, implantation), les acteurs d'une équipe de développement de systèmes d'information sélectionnent et réutilisent des composants qui répondent partiellement ou totalement à leurs besoins. Chaque acteur manipule des composants et des modèles de composants spécifiques au niveau d'abstraction où il intervient. Pour qu'une base de composants puisse satisfaire les besoins de tous les acteurs, elle doit donc gérer une collection de composants hétérogènes du point de vue modèle de composants (conceptuels, codes, etc.).

L'état de l'art sur les approches de recherche de composants a permis l'étude de différents types de systèmes de recherche de composants. Deux classements sont proposés, le premier selon les scénarios d'utilisation et le second selon la famille de techniques de recherche de composants exploitée. Des critères de comparaison (critères techniques, critères économiques, critères humains et caractéristiques de conception) sont définis, et une comparaison entre les différentes catégories de techniques de recherche de composants est effectuée. Cette comparaison montre qu'aucune des techniques de recherche de composants ne peut satisfaire les besoins de tous les acteurs d'un processus de développement de systèmes d'information à

base de composants. Une base de composants destinée à couvrir tout le cycle de développement doit donc offrir plusieurs techniques complémentaires de recherche de composants.

Partant de ce constat, les objectifs de cette thèse s'articulent autour de la construction d'un environnement d'aide à la réalisation et à l'utilisation de composants pour satisfaire les conditions suivantes :

❶ Hétérogénéité des besoins utilisateurs,

❷ Hétérogénéité des niveaux d'abstraction de composants,

❸ Hétérogénéité des modèles et des sources de composants,

❹ Hétérogénéité des techniques de recherche de composants,

❺ Adaptabilité et évolutivité.

Nous résumons ci-dessous les résultats de notre travail.

☙ Une base descriptive de composants (*B-Sigma*) et un modèle (*C-Sigma*)

Dans le chapitre IV, la base descriptive de composants *B-Sigma* joue le rôle d'un référentiel fédérateur qui facilite le recensement, la documentation et la capitalisation de composants hétérogènes (niveau d'abstraction, modèles de composants, etc.). Les composants sont disponibles depuis des sources de composants externes à l'équipe de développement (fournisseurs Internet, autres équipes de développement, projets open source, etc.) ou bien développés en interne en utilisant des ateliers de génie logiciel, des environnements de développement intégrés, etc.

Le modèle *C-Sigma* permet la réalisation de la base descriptive de composants *B-Sigma*. Comme tout système d'information, la base descriptive de composants *B-Sigma* est appelée à évoluer au cours du temps avec les besoins utilisateurs. Ces besoins d'évolution sont d'ores et déjà pris en compte grâce à des points d'extension permettant de faire évoluer le modèle *C-Sigma*.

☙ Un métamodèle de bases descriptives de composants (*M-Sigma*)

Dans le chapitre V, le métamodèle *M-Sigma* offre à l'ingénieur de modèles de bases descriptives de composants la possibilité de construire différents modèles de bases descriptives de composants et de les instancier dans le même système de bases descriptives de composants. Il découple également les techniques de recherche de composants de la représentation des composants dans la base descriptive de composants. En effet, à l'inverse des approches de systèmes de recherche de composants étudiées dans l'état de l'art, une technique de recherche de composants est associée dans *M-Sigma* à un type de données et non plus à la représentation du composant dans la base de composants. La description d'un composant est un arbre de rubriques typées. Ainsi, une requête simple consiste à naviguer dans le modèle d'une base descriptive de composants et à sélectionner une rubrique à partir de laquelle il est possible de choisir l'une des techniques associées au type de données de la rubrique sélectionnée.

☙ Des fragments de démarche pour guider l'évolution du modèle *C-Sigma*

Le chapitre VI propose un langage de patrons présentant des fragments de démarche permettant l'exploitation des points d'extension prévus dans le modèle *C-Sigma*. Les deux principaux types de problèmes traités sont l'ajout d'une nouvelle entité dans le modèle *C-Sigma* et l'ajout d'une nouvelle classification d'une entité du modèle *C-Sigma*.

Un modèle de système de recherche de composants

Le chapitre VII présente un modèle de système de recherche de composants permettant l'écriture de requêtes utilisateurs et leur exécution sur le contenu d'une base descriptive de composants gérée par le système de gestion de bases descriptives de composants. Une requête utilisateur peut être de type simple ou composite. Une requête simple utilise une seule technique de recherche de composants et porte sur une seule rubrique d'un modèle de bases descriptives de composants. Une requête composite est une expression de requêtes simples composée avec des opérateurs binaires booléens. À chaque opérateur binaire est associée une fonction définissant la stratégie de fusion des résultats des sous-requêtes. Un utilisateur du système de recherche de composants dispose ainsi d'une ou de plusieurs techniques de recherche de composants pour retrouver les composants qui correspondent à ses besoins. Pour optimiser le temps de réponse du système, un utilisateur peut faire une présélection avec des techniques ayant une faible complexité et ainsi diminuer le nombre de descriptions traitées par les techniques qui ont une complexité plus élevée.

Le modèle de systèmes de recherche de composants proposé est évolutif car il permet à tout moment d'ajouter de nouvelles techniques de recherche de composants associées à des types particuliers de données.

Une technique structurelle externe de recherche de composants

Enfin, le chapitre VII propose une technique de recherche de composants à la fois structurelle et externe : structurelle car elle représente la signature et la structure interne des composants en utilisant des spécifications formelles en logique du premier ordre, et externe car elle indexe les composants à travers leurs diagrammes de classes UML. La phase d'indexation est entièrement automatisée, ce qui diminue le coût de mise en œuvre. Ensuite, l'interrogation de la base se fait dans un langage familier aux ingénieurs puisqu'une requête n'est autre qu'un diagramme de classes UML, ce qui facilite son adoption par les *ingénieurs d'applications*. Enfin, le comportement de la technique de recherche de composants peut facilement être modifié par l'ingénieur d'applications en utilisant des métaconnaissances.

Les objectifs fixés au commencement de cette thèse sont ainsi atteints. En effet, les cinq résultats résumés ci-dessus ont permis la construction d'un environnement d'aide à la gestion et à l'utilisation de composants vérifiant les quatre degrés d'hétérogénéité :

- le modèle *C-Sigma* permet l'hétérogénéité des composants du point de vue besoins utilisateurs (❶), niveaux d'abstraction (❷), modèles et sources de composants (❸),
- le modèle de systèmes de recherche de composants exploite des techniques de recherche de composants hétérogènes (❹).

L'environnement réalisé est adaptable et évolutif (❺) :

- le métamodèle *M-Sigma* permet l'ajout de nouveaux types de données (*DataType*),
- le modèle de systèmes de recherche de composants permet l'ajout de nouvelles techniques de recherche de composants (*ComponentsRetrievalTechnique*).

2. Évolution du métamodèle M-Sigma

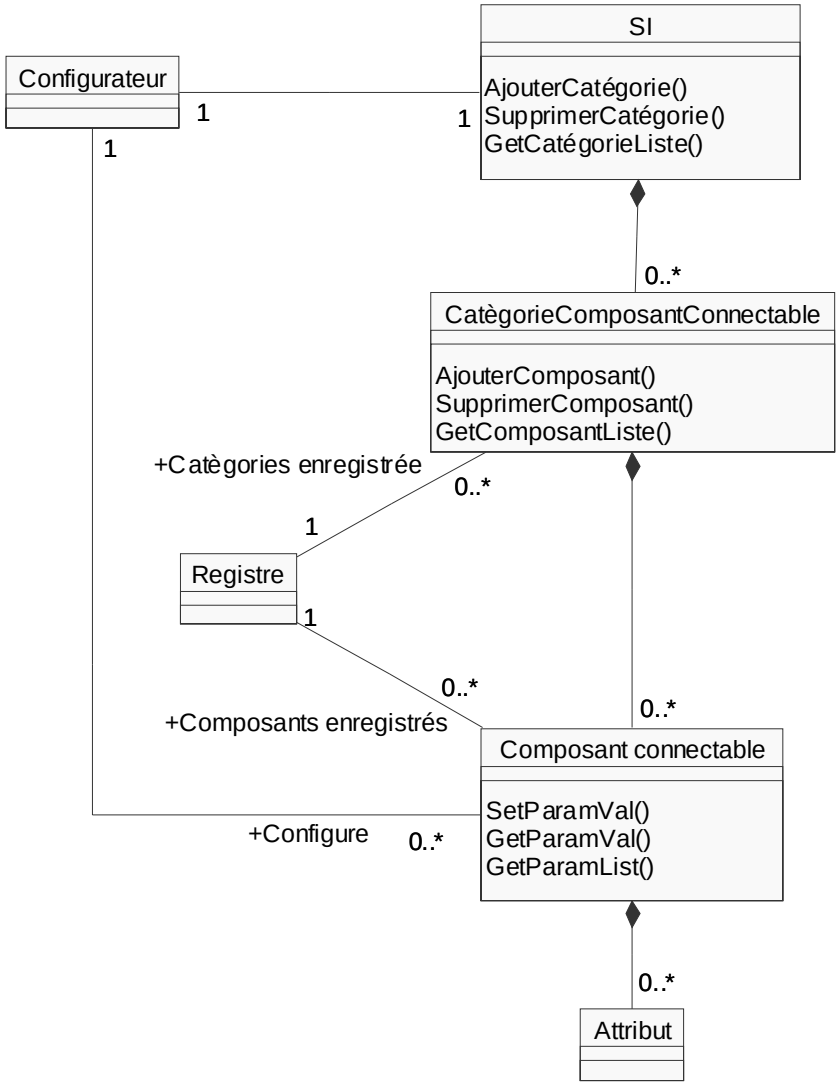
Dans le cadre de cette thèse, nous n'avons pas détaillé les concepts de contrainte (*Constraint*) et de script (*Script*). Ces concepts doivent être précisés en particulier en s'inspirant de concepts existants comme le mécanisme de règles actives dans le domaine des bases de données.

De plus, bien qu'évolutif, toute modification du métamodèle *M-Sigma* concerne le code source du système de gestion de bases descriptives de composants et risque de ne plus être compatible avec les base descriptive de composants déjà instanciées. Afin de rendre notre système plus ouvert et plus facile à faire évoluer, nous envisageons d'utiliser une architecture à base de composants connectables qui permettrait d'étendre notre environnement avec des composants sous la forme d'extensions intégrables par l'administrateur. Ainsi, de nouveaux types de données (*DataType*) ou de nouveaux outils permettraient à notre environnement d'offrir de nouvelles fonctionnalités. Le tableau 19 propose le patron « Architecture de systèmes d'information à composants connectables » qui présente une première solution pour ce problème.

Tableau 19. *Le patron Architecture de systèmes d'information à composants connectables*

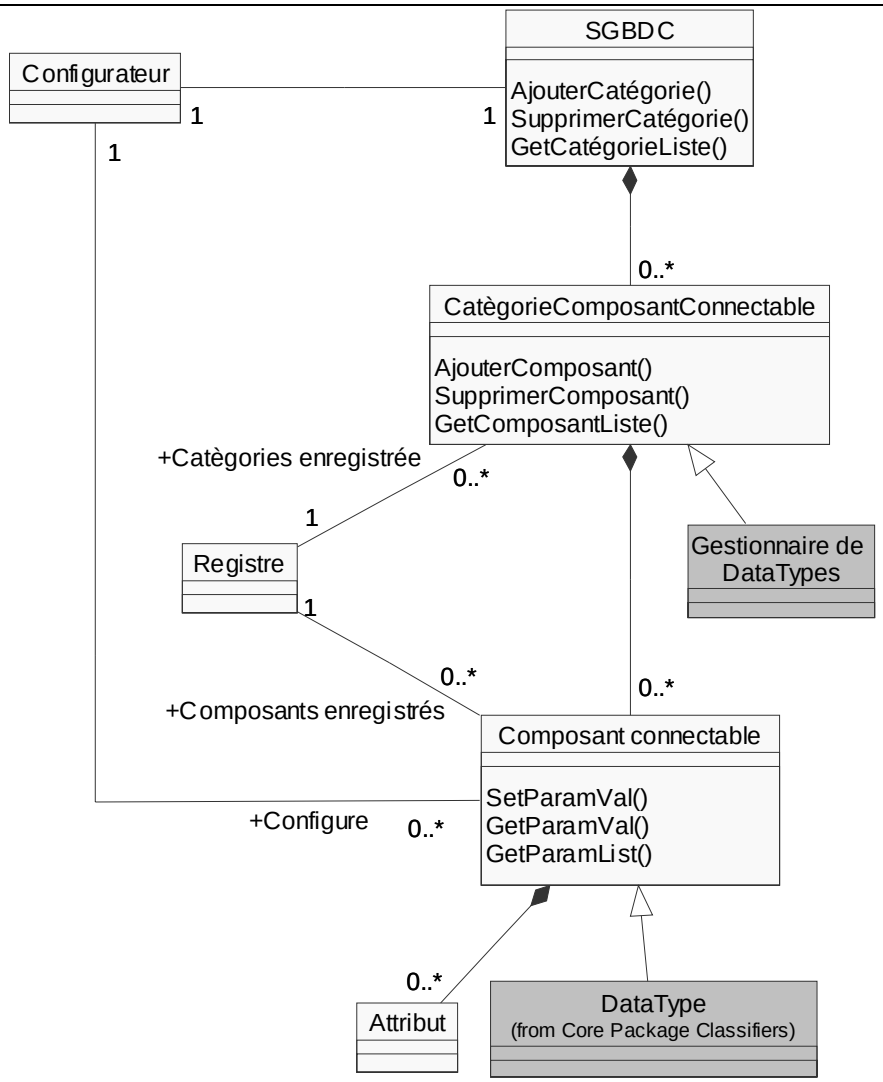
Identifiant	Architecture de systèmes d'information à composants connectables.
Contexte	Ce patron ne nécessite aucun autre patron pour être appliqué.
Problème	Pendant la durée de vie d'un système d'information, il est souvent nécessaire d'échanger ou de faire évoluer des composants. L'approche orientée objet standard consiste à utiliser une classe abstraite qui représente ces composants dans le SI pour pouvoir définir le contrat du composant vis-à-vis du SI et faciliter son remplacement par un autre composant du moment qu'il satisfait le même contrat. Cette approche nécessite de recompiler le code à chaque fois qu'on ajoute ou qu'on supprime un composant. L'intégration du nouveau composant est faite au niveau du code. L'instanciation et l'initialisation du composant sont réalisées lors de sa première utilisation. Ce genre d'architecture n'est pas satisfaisant si on veut créer un système d'information évolutif capable d'intégrer « à chaud » de nouveaux composants. Ces composants sont écrits a posteriori, ou adaptés (grâce à une enveloppe qui rend leur interface conforme à la classe *ComposantConnectable*) à partir de composants existants.
Forces	– Le système d'information est capable d'intégrer et de configurer de nouveaux composants. – Les nouveaux composants peuvent ne pas être disponibles lors de la compilation du code du système d'information. – L'administrateur du système d'information dispose d'un outil lui permettant de gérer et de configurer les composants du système d'information.
Solution produit	Un système utilisant des composants connectables utilise souvent plusieurs catégories de composants. Le composant *CatégorieComposant Connectable* joue donc le rôle de fabrique de composants pour les composants de cette catégorie. Il permet l'ajout et la suppression des composants connectables du système d'information. Le composant *ComposantConnectable* est composé d'*Attributs* servant par la suite à la configuration du composant. Les attributs sont utilisés par le *Configurateur* pour paramétrer les

composants. Le composant *Registre* sert à localiser et à identifier les composants utilisés dans le système d'information.



Cas d'application

Ce patron est applicable dans le cas où nous voulons gérer les types de données (*DataType*) comme étant des composants connectables. Lorsque les développeurs du système de gestion de bases descriptives de composants veulent ajouter un nouveau type de données comme par exemple les diagrammes de séquences UML, ils préparent un paquetage contenant les classes nécessaires et utilisent le configurateur pour l'ajouter au système de gestion de bases descriptives de composants.

	
Conséquences d'application	Les composants *Registre*, *Configurateur* et chacune des instances de *CatégorieComposantConnectable* sont uniques dans un système d'information. Ils doivent être accessibles par tous les autres composants du système d'information. Pour résoudre ce problème, on peut utiliser le patron Singleton de Gamma (Gamma, 1995).
Utilise	Patron Singleton de Gamma (Gamma, 1995).

3. Autres perspectives

Deux autres perspectives sont envisageables à court terme pour notre travail.

☙ Évolution du modèle de systèmes de recherche de composants

Le patron « Architecture de systèmes d'information à composants connectables » peut être appliqué aussi sur les techniques de recherche de composants. Ainsi, le système de recherche de composants devient plus facile à faire évoluer. La solution oblenue par imitation de ce patron sur le modèle du système de recherche de composants est présentée dans la figure 98.

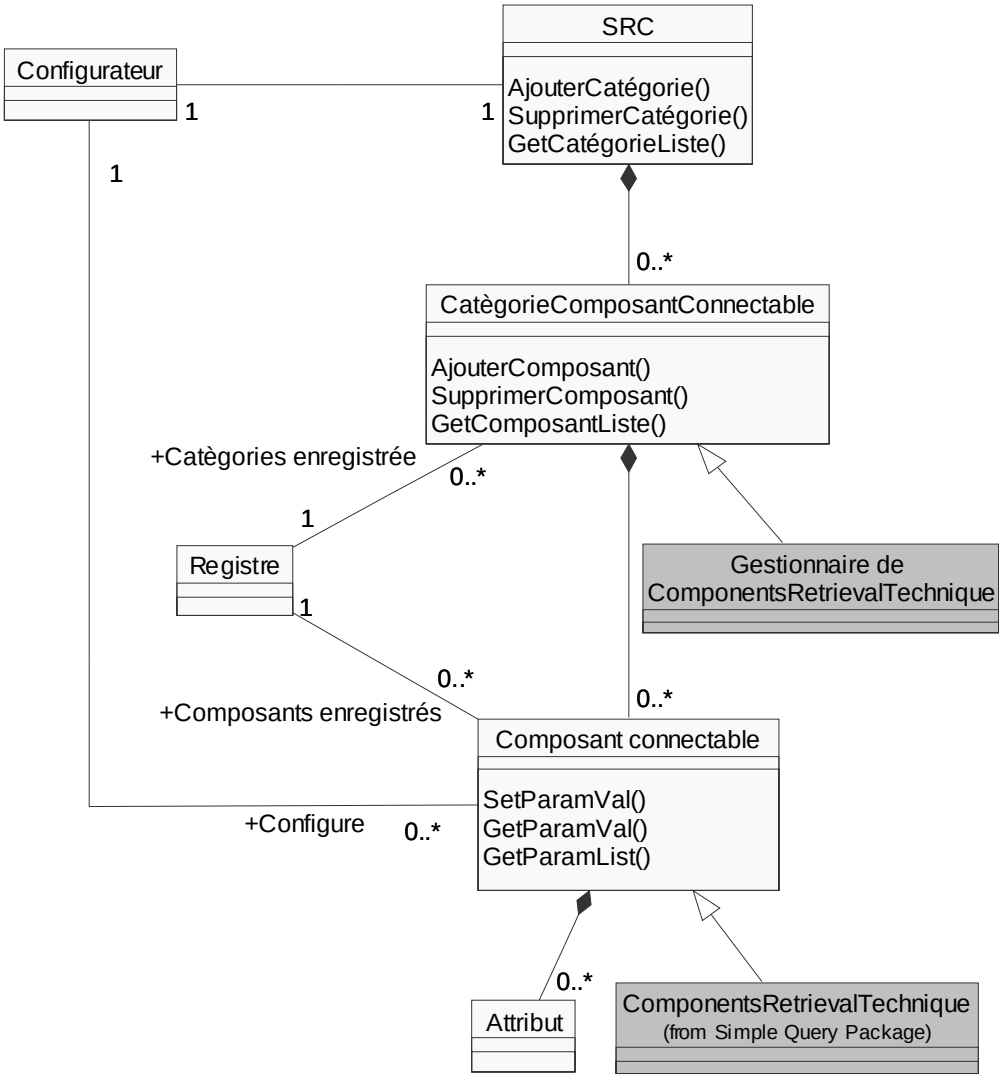


Figure 98. *Un système de recherche de composants à base de composants connectables.*

De plus la technique de recherche de composants proposée pour les diagrammes de classes doit être étendue pour d'autres types de diagrammes UML (séquence, collaboration, etc.), tout comme il est nécessaire de proposer d'autres techniques de recherche de composants adaptées aux autres types de rubriques.

Enfin, les base descriptive de composants peuvent et doivent être utilisées pour faire de la réingénierie de systèmes d'information, comme par exemple pour rechercher des structures types de composants dans les diagrammes de classes décrivant les systèmes d'information.

Processus d'alimentation des bases descriptives de composants

L'environnement d'aide à la réalisation et à l'utilisation de composants que nous avons proposé traite pour l'instant uniquement de la documentation, de la gestion et de la sélection des composants. Il est nécessaire de prévoir un processus d'alimentation des bases descriptives de composants. Ce processus d'alimentation peut soit être une automatisation de l'extraction des descriptions de composants à partir de fournisseurs Internet de composants, soit être le fruit d'un processus coopératif pour la proposition, la validation et enfin l'ajout de nouveaux composants aux bases descriptives de composants.

Nos contributions constituent des éléments qui devraient permettre de mieux aborder deux problèmes critiques conjoints en ingénierie des systèmes d'information : la traçabilité et la vérification de la cohérence des réutilisations de composants.

Annexes

Annexe A : Le formalisme de patrons P-Sigma

Le formalisme *P-Sigma* est un formalisme structuré (Conte, 2001b) dont le but est de permettre l'écriture de patrons de manière uniforme, qu'ils soient de type produit ou processus. *P-Sigma* permet aussi une meilleure structuration des systèmes de patrons pour faciliter la recherche et la sélection des patrons. Il est constitué des trois parties : interface, réalisation et relations. Chaque partie regroupe un certain nombre de rubriques. Chaque rubrique peut contenir un nombre variable de champs. Les rubriques ainsi que les champs d'un patron se divisent en deux groupes : optionnels ou obligatoires.

- **Interface**

La partie Interface est composée de 5 rubriques qui permettent la sélection des patrons.

Identifiant	Un nom ou une phrase qui décrit brièvement le patron. Cette rubrique permettra par la suite aux autres patrons du langage de le référencer. Cette rubrique ne compte qu'un seul champ textuel.
Contexte	Il exprime la pré-condition pour appliquer le patron. Il est exprimé sous une forme textuelle et/ou une forme formelle. L'expression formelle est constituée d'un ensemble de patrons.
Classification	C'est un ensemble de mots-clés du domaine définissant la fonction du patron dans son système. Elle donne intuitivement la classification du domaine. Elle est représentée soit de manière informelle au moyen d'un champ textuel, soit de manière formelle au moyen d'une expression logique des mots-clés (expression domaine).
Problème	Il donne une description textuelle du problème résolu par le patron. Un seul champ textuel.
Forces	Cette rubrique traite de la pertinence et des apports de l'application du patron. L'importance de la force dépend de l'intention de la solution. Cette rubrique apporte une valeur complémentaire indirecte au problème. Hormis le champ textuel, elle dispose d'un champ formel contenant une expression logique des critères de qualité associés à une technologie (expression qualité).

Le modèle de la partie interface du formalisme *P-Sigma* est présenté dans la figure 99.

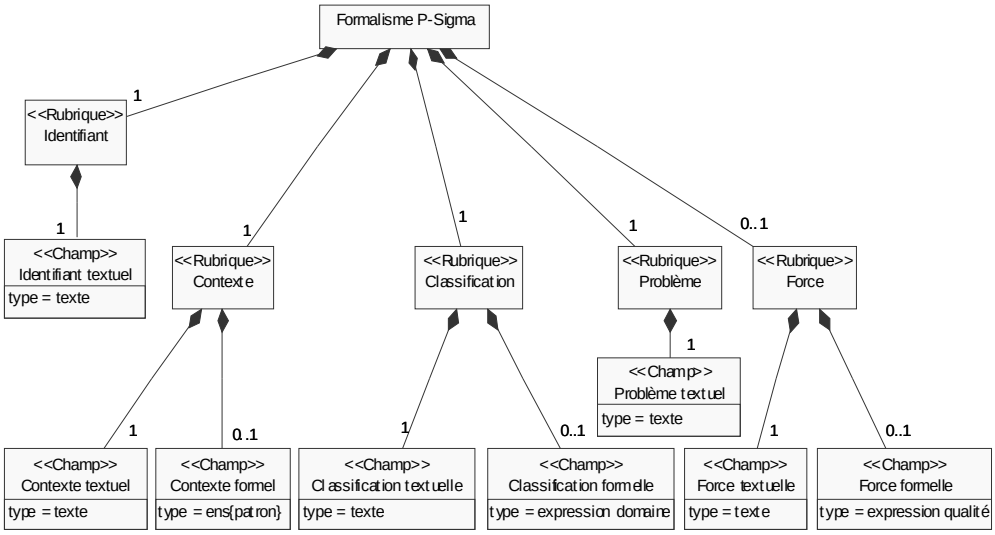


Figure 99. *Modèle de la partie « Interface » du formalisme P-Sigma.*

- **Réalisation**

La partie Réalisation comprend quatre rubriques :

Solution «modèle»	Chaque patron propose une solution au problème auquel il s'adresse. Cette partie décrit la solution en terme des produits attendus après l'application du patron (solution produit). Elle constitue une post-condition. Il y a deux possibilités d'exprimer cette solution : - informellement : par un champ textuel - formellement : sous forme d'un diagramme de classes et/ou de diagrammes de séquence. La rubrique Solution « modèle » est donc composée de trois champs : un champ textuel, un champ de type diagramme de classes, un champ optionnel de type ensemble de diagrammes de séquence.
Solution «démarche»	Cette rubrique indique le processus à suivre pour résoudre le problème. Outre la forme textuelle par défaut, l'utilisation d'un diagramme d'activités est possible. Si c'est le cas, il est formalisé sous la forme : [garde] Patron.
Cas d'application	Cette rubrique décrit des exemples d'imitation de la « solution Modèle ». Elle est optionnelle mais fortement conseillée pour faciliter la compréhension de la solution du patron. Comme le cas d'application est une imitation de la solution « Modèle », cette rubrique est composée de trois champs : un champ informel, un champ de type diagramme de classes et un champ de type ensemble de diagrammes de séquence.
Conséquences d'application	Aucune solution n'étant parfaite, c'est dans cette rubrique que sont discutés à la fois les limites et les bénéfices de l'application de la solution. Cette rubrique peut inclure un nouvel ensemble de problèmes faisant apparaître la nécessité d'appliquer de nouveaux patrons.

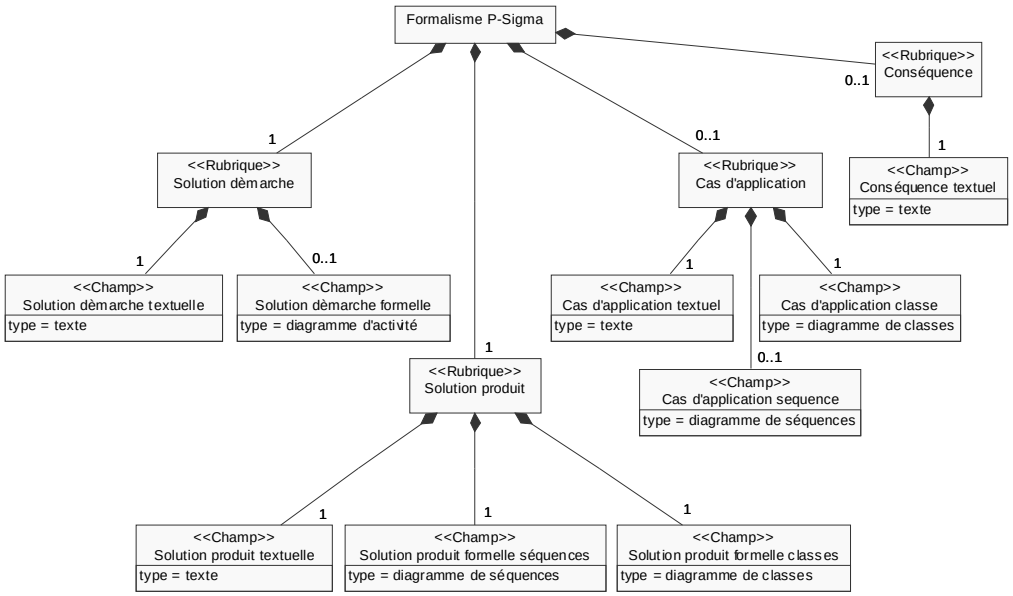


Figure 100. *Modèle de la partie « Réalisation » du formalisme P-Sigma.*

- **Relations**

Les relations entre le patron décrit et d'autres patrons dans le système de patrons sont exprimées au moyen de quatre rubriques correspondant aux quatre principales relations entre patrons : « utilise », « raffine », « alternative » et « requiert ». Chaque relation est exprimée par un champ représentant l'ensemble des patrons qui sont liés au patron en question par la relation concernée. Les relations définies dans le formalisme P-Sigma sont présentées dans le chapitre II section 2.5.

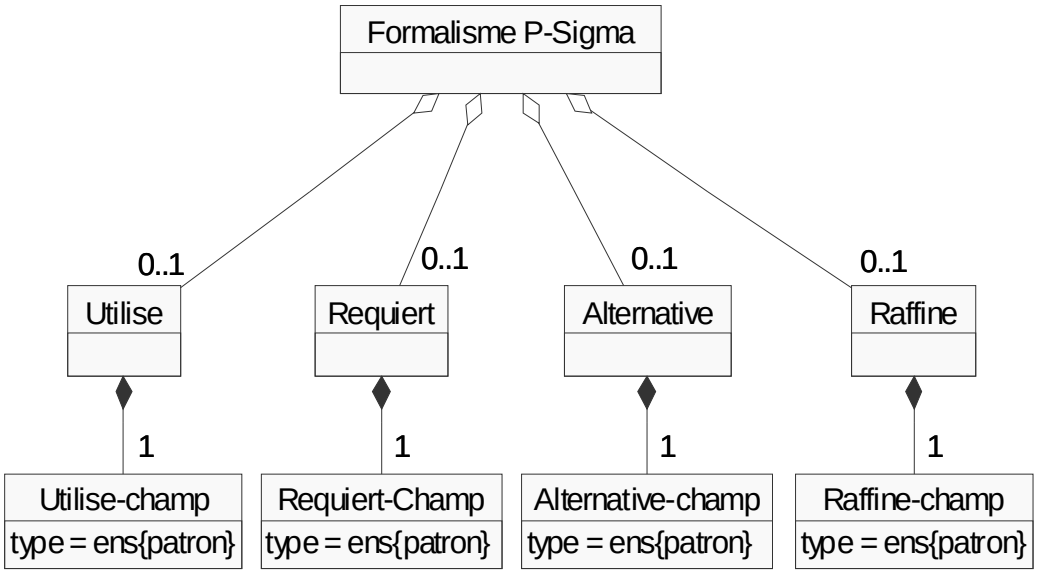


Figure 101. *Modèle de la partie « Relations » du formalisme P-Sigma.*

Annexe B : Les modèles de composants COM/DCOM/COM+

Cette annexe résume d'une part les concepts de base des modèles de composants COM/DCOM/COM+ et d'autre part l'environnement d'exécution de ces modèles.

1. Les concepts du modèle

Le modèle objet COM supporte les concepts usuels « objet », « type », « classe » et « composant », avec une interprétation particulière qui conduit souvent à confondre objet et classe.

- **Objet COM (*Instance de composant*)**

C'est une instance représentant une entité fonctionnelle avec des données cachées, manipulées par des fonctions (Gardarin, 1996). Comme dans la plupart des modèles objets, les instances offrant les mêmes interfaces sont regroupées en types. Un type est un ensemble de signatures de fonctions (méthodes).

- **Classe COM**

C'est une implantation d'un type composé de code machine exécuté lorsqu'on interagit avec un objet COM. En pratique, une classe correspond à un ensemble de services accomplis par des serveurs (EXE) ou des bibliothèques (DLL). Le principe d'encapsulation est complètement respecté : les données d'une implantation sont toujours cachées ; seules les fonctions des interfaces sont visibles.

- **Composant COM**

C'est un ensemble d'une ou plusieurs classes non indépendantes permettant de réaliser une fonctionnalité réutilisable. Une autre définition possible est la suivante : « un composant est une implantation concrète d'une ou plusieurs interfaces ». COM ne spécifie pas ce que doit être un composant. Seul le format binaire de l'interface est spécifié. Un exemple issu du monde Microsoft est le correcteur orthographique utilisé à la fois dans MS-WORD et MS-EXCEL (voir figure 102). Ce composant est implanté sous la forme d'une bibliothèque dynamique (DLL). Les applications MS-WORD et MS-EXCEL sont aussi des composants COM. Un tableau Excel peut être inclus dans un document Word.

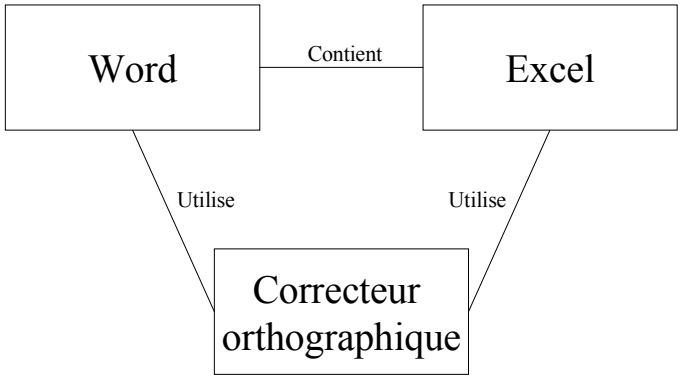


Figure 102. *Exemple de composants COM.*

- **Interface COM**

Une interface COM est un ensemble de fonctions sémantiquement liées groupées sous un même nom qui permet de manipuler un objet COM. Une interface COM spécifie la signature des méthodes qu'elle expose et elle est exprimée dans un langage de description spécifique (IDL) contenant des informations comme :

- le type de retour,
- le type des paramètres,
- le convention d'appel.

Une interface est une spécification sémantique car elle doit être documentée. L'ordre d'appel des méthodes et le type de comportement attendu font partie de la spécification de l'interface.

Tout composant COM implante l'interface « IUnknown » (cf. figure 103) dont le rôle est de permettre la gestion du cycle de vie du composant et la découverte des autres interfaces du composant (introspection) à travers la fonction « QueryInterface ».

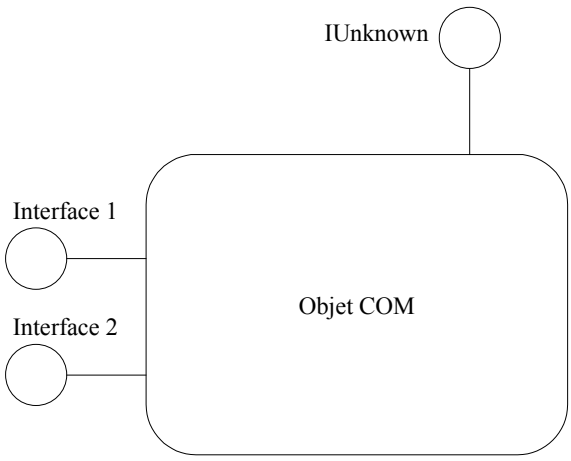


Figure 103. *Interface d'un objet COM.*

- **Pointeur d'interface**

Le pointeur d'interface pointe sur une table de pointeurs vers les implantations des fonctions d'une interface. D'un point de vue implantation, chaque interface possède un nom et un identifiant global. A partir de l'identifiant d'une interface et d'un objet, il est possible d'obtenir un pointeur permettant de référencer cette interface. La figure 104 montre un objet COM et ses interfaces.

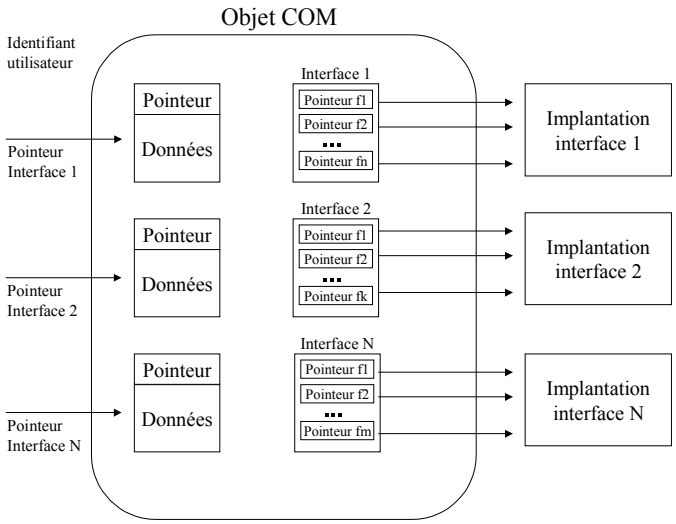


Figure 104. *Un objet COM et ses interfaces.*

- **Composition**

Les composants COM ne décrivant pas quelles sont les interfaces requises par les instances de composants, il est difficile de réaliser une composition de façon explicite. Les dépendances entre instances de composants ne sont pas créées par un tiers, mais elle sont décrites via des instructions de création enfouies à l'intérieur des instances mêmes. Par ailleurs, la composition peut être réalisée de façon impérative à travers des langages de script comme Visual Basic Script.

- **Paquetage de composant**

Les paquetages ou modules binaires COM sont les unités de déploiement des composants COM sur un système. Ils se présentent sous forme de programmes exécutables (EXE) ou de bibliothèques dynamiques (DLL). Les unités de déploiement peuvent contenir plusieurs types de composants et leurs fabriques de composants.

2. Environnement d'exécution

L'infrastructure d'accueil des composants COM est complètement intégrée dans le système d'exploitation Windows et les composants sont enregistrés dans le registre de Windows. L'environnement d'exécution de COM se charge de rendre transparente la communication entre les instances de composants. Un composant est lié à l'exécution (par un mécanisme de bibliothèque dynamique) soit au client (composant In-process), soit à un serveur de la même machine que le client (composant Out-process Local), soit à un serveur d'une machine distante (composant Out-process Remote). A partir de COM+, un composant peut être hébergé par un serveur standard appelé MTS (cf. figure 105).

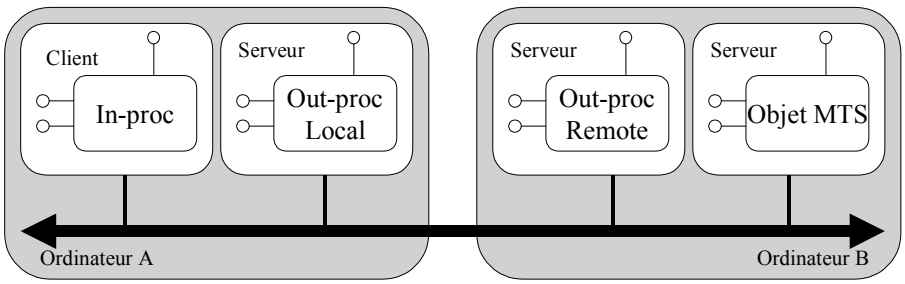


Figure 105. *Infrastructure d'accueil des composants COM+.*

Un composant COM invoque les opérations fournies par un autre composant et ne s'occupe pas de savoir si le composant s'exécute dans le même processus (même contexte d'exécution), dans un autre processus ou sur une machine distante (cf. figure 106). C'est le système d'exploitation qui délègue les appels entre processus ou à travers le réseau en utilisant un objet par procuration (proxy[21]).

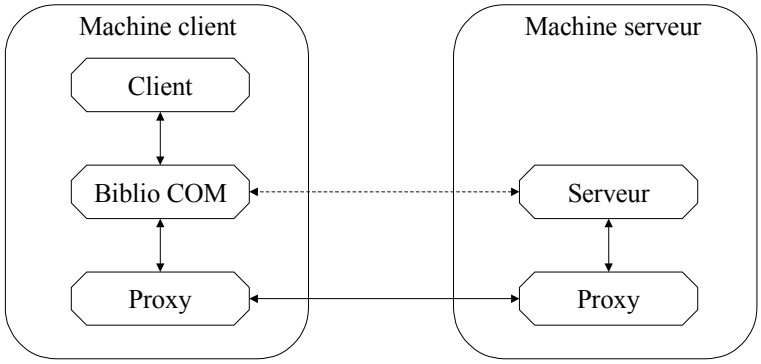


Figure 106. *Modèle client/serveur COM.*

L'infrastructure COM offre un certain nombre de services présentés ci-dessous.

- **Le stockage d'objets persistants**

Le modèle COM offre un ensemble d'interfaces et d'implantations associées permettant la création de supports de stockage (documents) d'objets structurés et partagés. Une gestion de transactions est aussi offerte. L'idée est de pouvoir créer et gérer des supports composés, de sorte à pouvoir placer dans un même support une structure de fichiers de données appelés « chaînes » d'octets et des « répertoires » représentant un document composé. L'avantage de cette technique est de pouvoir échanger des données complexes sous la forme d'un seul document et non sous la forme de plusieurs documents.

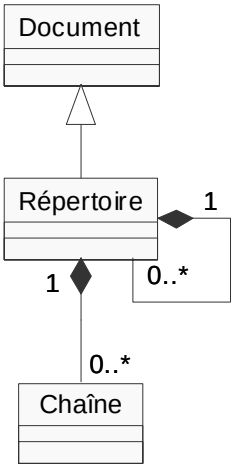


Figure 107. *Structure d'un document composé.*

- **Le transfert de données uniforme**

[21] Un proxy est un représentant d'un objet serveur dans le contexte d'un objet client, chargé de recevoir les appels du client, de les transmettre au serveur (via une souche), de récupérer les réponses et de les retourner au client.

COM offre des services standards pour échanger des données entre applications. Ces services sont implantés au dessus de COM et des services de persistance de la section précédente. Ils sont fournis au travers d'une implantation unifiée d'un objet mémorisant des données, appelé « objet à données » (data object) offrant une interface standard pour le transfert de données appelé « IDataObject ». Cet objet est capable de mémoriser les données dans un format interne et de les restituer dans différents formats.

- **Les noms intelligents**

COM est capable de gérer des noms intelligents persistants. Ces noms permettent de réaliser des liens depuis un objet source vers un objet cible pouvant être contenus dans des fichiers composés. De tels noms sont appelés les « noms intelligents » ou « monikers ». Un nom intelligent est représenté par un objet COM qui implante ce lien. Les noms intelligents peuvent appartenir à plusieurs composants suivant la sémantique du lien qu'ils représentent, donc ils peuvent offrir différentes interfaces selon la sémantique du lien. Un nom peut être composé ; dans ce cas, il est constitué d'une suite de noms qui seront interprétés en séquence. Ainsi, on peut par exemple enchaîner un nom désignant un fichier, puis un nom désignant un objet répertoire, enfin un nom désignant un objet chaîne. Les noms intelligents sont un aspect important de COM : ils permettent de lier intelligemment des objets, par exemple afin de composer des documents composites.

- **La sécurité**

L'infrastructure COM assure la sécurité des applications. Par défaut, les serveurs sont non sécurisés et s'exécutent sous l'authentification (userid, password) du client. Il existe alors un serveur par client. Les serveurs sécurisés nécessitent une authentification propre ; ils peuvent utiliser les services standards du RPC (userid, password) ou implanter des contrôles spécifiques. En principe, un serveur non sécurisé doit s'exécuter sur le site du client alors qu'un serveur sécurisé peut s'exécuter sur un site distant.

Annexe C : Les composants EJB

Cette annexe résume les concepts de base du modèle de composants EJB et de son environnement d'exécution.

1. Les concepts du modèle

Nous présentons dans cette section les principaux concepts définis dans le modèle de composants EJB (Morrison, 1997) (Johnson, 1998) (Matena, 1999).

- **Composant Enterprise JavaBean**

Pour développer un nouveau composant EJB, le concepteur de composants doit définir l'interface fonctionnelle du composant (vue externe) en définissant l'interface *RemoteInterface*. La vue interne est générée en utilisant des outils de génération de code fournis par le fournisseur de conteneur (environnement d'exécution) permettant de gérer les deux classes *EJBObject* et *HomeObject*. Par la suite, le concepteur du composant EJB conçoit et code la partie fonctionnelle du composant EJB. La partie fonctionnelle se présente sous la forme d'une classe *Bean*. Jusqu'à la spécification 1.1, le modèle de composants EJB comportait deux types de Beans : *EntityBean* et *SessionBean*. Avec l'arrivée de la spécification 2.0 du standard EJB, un troisième type de Beans a été ajouté : *Message-DrivenBean*.

- **Les trois sortes d'instances de composants EJB**

– Les EntityBeans : Les *EntityBeans* modélisent des objets du monde réel ; ces objets sont généralement des enregistrements persistants stockés dans une base de données. Ils représentent un client, un lieu ou une chose. Un *EntityBean* existe tant qu'il n'a pas été détruit explicitement. Pour récupérer un *EntityBean*, un client peut utiliser la clé primaire du *Bean* donnée lors de la création de ce dernier. L'accès à *l'EntityBean* peut être partagé entre plusieurs clients. La persistance d'un *EntityBean* peut être gérée de deux façons différentes :

  - Persistance gérée par le conteneur (environnement d'exécution): le conteneur est responsable de la sauvegarde de l'état du *Bean*. Il suffit d'indiquer dans le descripteur de déploiement que la persistance de l'*EntityBean* est assurée par le conteneur et le code assurant cette persistance sera automatiquement généré.
  - Persistance gérée par le *Bean* lui-même : le *Bean* gère lui-même sa persistance, aucun code n'est généré et aucune sauvegarde automatique dans une base de données n'est réalisée. Par conséquent, cette implantation est moins adaptable que l'implantation du premier type *d'EntityBean*, puisque le concepteur doit coder lui-même la partie de l'entité qui s'occupera de la persistance.

– Les SessionBeans : à chaque session cliente correspond une instance d'objet de type *SessionBean*. Cette instance englobe un processus métier ou un workflow qui définit comment les autres *beans* interagissent. Les *SessionBeans* ne sont pas persistants, et ils vivent tant que le client a besoin d'eux. Il existe deux types de SessionBeans :
  - « stateless » (sans état) : le *bean* ne maintient pas d'informations suite aux différentes requêtes qui lui sont demandées et le conteneur peut alors partager séquentiellement le *bean* entre plusieurs clients ;
  - « statefull » (avec état) : le *bean* conserve en mémoire l'état du client et chaque instance du *bean* n'appartient qu'à son client et ne peut être partagée.
– Les Message-Driven Beans : un *Message-Driven Bean* est utilisé lorsque les méthodes fonctionnelles doivent être asynchrones ; il est invoqué sur réception d'un message du client en utilisant le service JMS de Java (Java Messaging Service). Par exemple, un composant de compte client reçoit un événement à chaque fois que le client utilise son titre de transport pour voyager sur le réseau de transport.

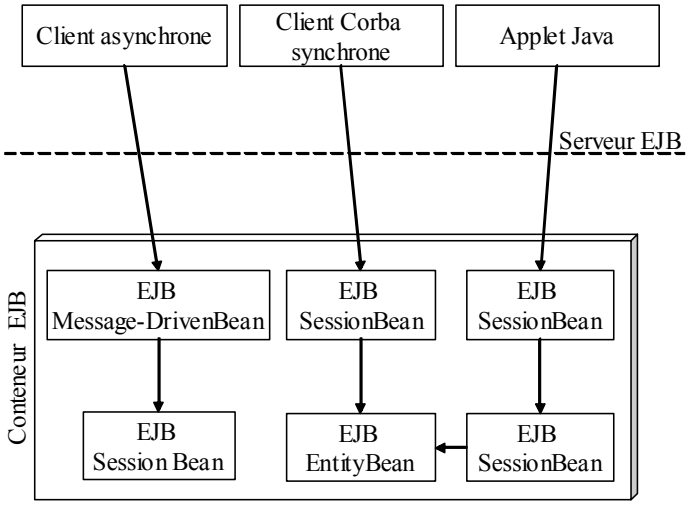


Figure 108. *Interaction des clients avec un système à base de composants EJB.*

- **Les interfaces locales**

La spécification EJB 2.0 introduit également la notion de localité du *bean*. Cette notion ne s'applique pas aux *beans* orientés messages qui sont nécessairement distants. Pour les *beans* « *Remote* » (issus des versions EJB 1.x), le client n'a pas besoin de connaître l'emplacement du *bean*. L'accès se fait obligatoirement à l'aide de Java RMI. Les *beans* « *Local* » sont par contre situés sur la même machine virtuelle que le client (un *bean* peut alors, à l'intérieur d'un même conteneur, faire appel à un autre *bean*).

Une conséquence fondamentale de la notion de localité concerne le passage de paramètres lors d'un appel de méthode : pour les *EJB Remote*, le passage se fait par valeur tandis que, pour les *EJB Local*, le passage se fait par référence, les arguments étant alors potentiellement partagés entre le client et le *bean*. La propriété *Local* ou *Remote* du *bean* est choisie lors de la création du *bean*.

- **Composition**

Développer une application à partir de composants EJB consiste à créer de nouveaux composants responsables d'instancier et de connecter des composants de base. La composition est normalement impérative à travers le langage Java. Les composants dont l'implantation réalise la composition sont typiquement des composants de type session avec état.

- **Paquetage de composants**

L'unité de livraison de composants EJB est un fichier de type JAR. En plus du code, elle contient un fichier appelé *descripteur de déploiement* dans lequel sont déclarées des propriétés de déploiement (qui incluent des propriétés non fonctionnelles) ainsi que des dépendances de déploiement.

Pour les versions EJB 1.X, il s'agissait d'un fichier au format XML contenant un certain nombre d'informations utilisées par le container lors du déploiement pour attribuer à l'EJB un comportement propre, comme le contrôle d'accès (service sécurité) sur telle fonction donnée. Les principales informations y figurant (pour le *Bean*, mais aussi pour chacune des méthodes spécifiées dans *RemoteInterface*) sont les suivantes :

- le nom du *Bean* déclaré dans le service de nommage ;
- le nom de *Home*, le nom de *Remote*, la clé primaire et la classe d'implémentation ;
- le mode de fonctionnement du *Bean* (avec ou sans état pour les EJB Sessions) ;
- les variables d'environnement ;
- les identités ayant accès au *Bean* ;
- l'identité exécutant le code du *Bean* ;
- le fonctionnement transactionnel désiré ;
- …

La version EJB 2.0 a introduit la notion de « Abstract Persistence Schema ». La nouvelle spécification définit un nouvel élément participant au processus de déploiement d'un EJB. C'est le gestionnaire de persistance (persistence manager). L'idée est de séparer le mécanisme permettant de gérer les relations entre *beans* du conteneur de composant, qui est responsable de la gestion de la sécurité, des transactions et des ressources. La séparation des responsabilités permet à différents gestionnaires de persistance de travailler avec différents conteneurs et donc d'améliorer la portabilité des *EntityBeans* entre vendeurs d'EJB aussi bien qu'entre gestionnaires de persistance.

2. Environnement d'exécution

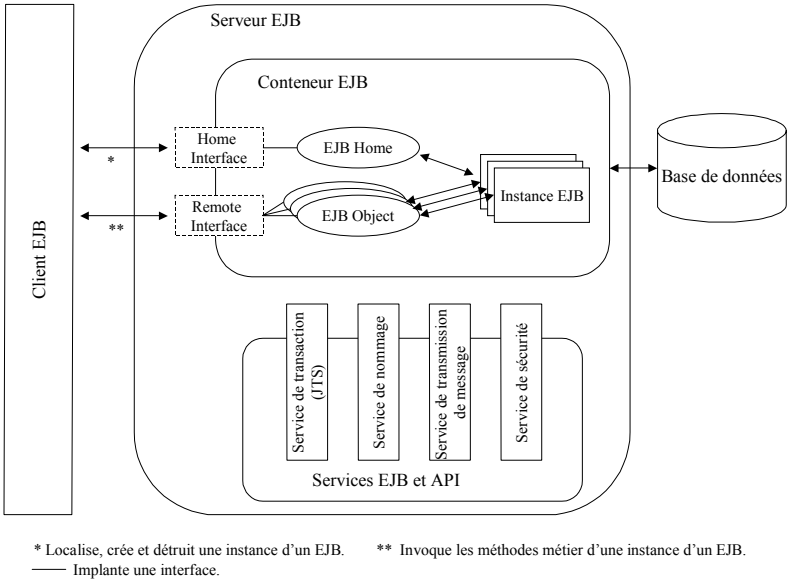


Figure 109. *Architecture d'un environnement à EJB.*

- **Les serveurs EJB**

Un serveur EJB (cf. figure 109) est un framework fournissant un environnement d'exécution pour les conteneurs des EJB. Le serveur EJB est l'intermédiaire entre le conteneur EJB et le système qui l'héberge. Il fournit les services systèmes aux conteneurs des EJB (accès aux bases de données, protocole de communication, gestion des ressources systèmes, services pour gérer le multiprocessus, service de nommage JNDI, service de gestion des transactions, etc.). Le serveur EJB peut offrir des services spécifiques au fournisseur de serveurs comme des interfaces optimisées pour l'accès aux données, les services CORBA, etc.

- **Les conteneurs EJB**

Un conteneur EJB (cf. figure 109) est responsable de la gestion du cycle de vie des composants EJB qu'il contient (fabrique de composants). Il offre des services aux composants EJB et joue le rôle de médiateur entre le composant EJB et ses clients. Cette médiation est faite à travers deux interfaces spécifiées dans le modèle de composants des EJB (*HomeInterface* et *RemoteInterface*). Le client n'accède pas directement au composant EJB. Les appels clients sont interceptés par le conteneur et ensuite délégués au composant EJB. La partie du conteneur qui s'occupe de cette délégation est la classe *EJBObject*. La classe *EJBObject* est générée par des outils de génération de code (fournis par le fournisseur de conteneurs) à partir de la définition des méthodes métier du composant EJB. Le développeur de composant décrit les méthodes du composant métier en écrivant l'interface *RemoteInterface*.

Pour la gestion du cycle de vie des composants, le conteneur utilise l'objet *HomeObject* qui implante l'interface *HomeInterface*. La *HomeInterface* définit les méthodes d'une fabrique de composants. Cette interface offre les méthodes de localisation, de création et de suppression des instances des composants EJB.

- **Les services**

Le conteneur fournit des services comme la sécurité d'accès aux composants (en vérifiant les droits d'accès des clients), la gestion des ressources système allouées aux composants, la persistance, les transactions, le nommage, etc. Le fournisseur de conteneur est libre d'ajouter des services supplémentaires spécifiques à son conteneur (voir figure 110).

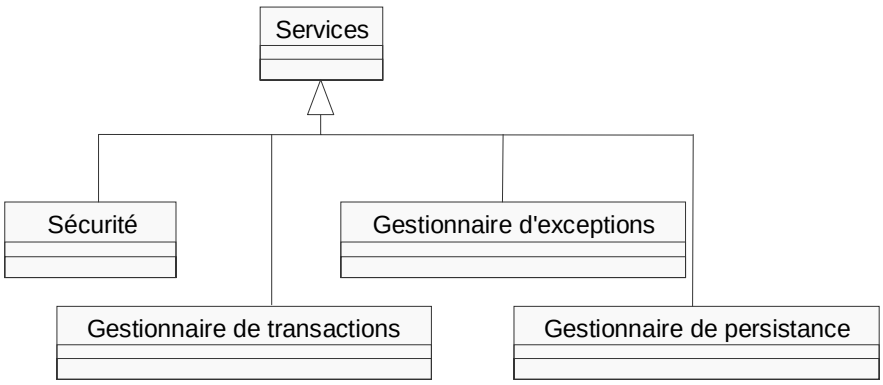


Figure 110. *Des services EJB.*

Annexe D : Les composants CCM

Cette annexe résume les concepts de base du modèle de composants CCM et de son environnement d'exécution.

1. Les concepts du modèle

a) Modèle abstrait

Le modèle abstrait permet à l'ingénieur d'applications de caractériser les interfaces fonctionnelles des composants CORBA. Il offre les outils nécessaires pour définir les composants du système d'information qu'il veut construire. La définition des composants se fait à travers la description de leurs vues externes en décrivant leurs propriétés et des interfaces offertes (respectivement requises) par ces composants. L'OMG a proposé le langage IDL3 (Interface Definition Langage), utilisé uniquement pour des besoins de modélisation. En effet, après que l'ingénieur d'applications ait défini les interfaces fonctionnelles de ses composants, un compilateur traduit cette description vers le langage IDL2[22]. Ainsi CCM garde la compatibilité avec l'ancienne version CORBA2. CCM ajoute une couche de haut niveau au-dessus de CORBA2 pour lui permettre de supporter le concept de composant.

Pour la modélisation des composants CCM, le modèle abstrait définit la notion d'attribut et un ensemble de ports qui sont utilisés pour définir la vue externe du composant.

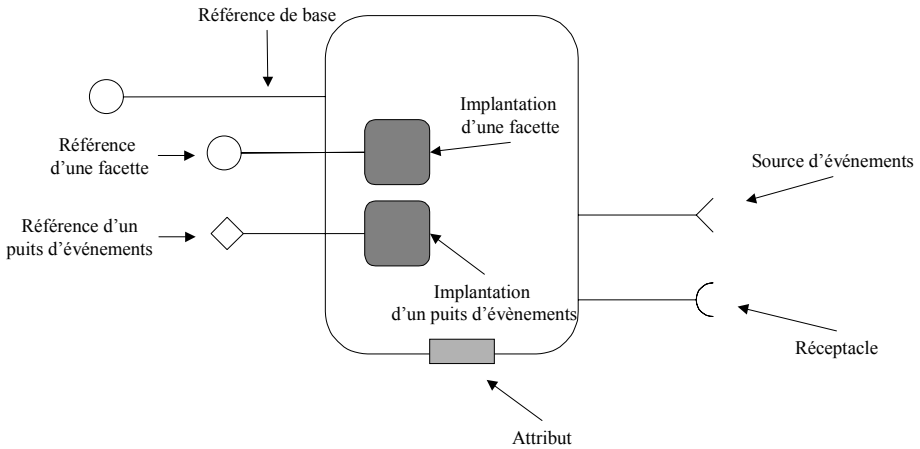


Figure 111. *Représentation d'un composant dans CCM.*

[22] IDL2 est l'IDL (Interface Definition Langage) défini par CORBA2

- **Attribut**

Les attributs (cf. figure 111) représentent les propriétés configurables du type de composant. Ainsi il est possible de paramétrer le comportement du composant en fonction des valeurs affectées à ses attributs. Une particularité des attributs dans le cadre de CCM est de pouvoir lever des exceptions lorsque les composants sont mal utilisés.

- **Port**

Un port représente une interface soit fournie, soit utilisée par le composant. Quatre types de ports sont définis dans CCM (cf. figure 111) :

- **une facette** : c'est une interface fournie par un composant et qui est utilisée par des clients en mode synchrone. La rétrospection est disponible avec les facettes.
- **un réceptacle :** c'est une interface utilisée par un composant en mode synchrone.
- **un puits d'événements :** c'est une interface fournie par un composant et utilisée par ses clients en mode asynchrone.
- **une source d'événements :** c'est une interface utilisée par un type de composants en mode asynchrone.

Les ports définis dans CCM sont utilisés lors de la phase de déploiement et d'exploitation des composants pour connecter (composer) les composants lors de la construction d'un système d'information. Les ports ont aussi une cardinalité associée qui peut être *simple* ou *multiple*. Elle permet de définir des contraintes sur le nombre de connexions vers une instance de composant.

Un composant est identifié par sa référence de base qui permet aux autres composants (clients) de le retrouver et d'interagir avec.

Les composants sous CCM supportent uniquement l'héritage simple, et utilisent le mécanisme des interfaces à la façon Java : un type de composant peut implanter plusieurs interfaces et ne peut dériver que d'un seul ancêtre.

- **Evénements**

Le modèle à base d'événements utilisé par CCM est le modèle classique de producteur/consommateur. Pour recevoir des événements, un client (ou consommateur) doit s'abonner au niveau de la source d'événements de ce type. Les opérations de traitement des événements sont définies au niveau des interfaces du consommateur. Seul le mode *push* est utilisé. Comme précédemment dit, il y a deux types de ports qui traitent les événements (les sources et les puits d'événements).

Les sources d'événements sont de deux ordres. *Emitter* est la catégorie de source qui n'accepte qu'un seul consommateur (un vers un). Dans ce cas, il n'y a pas de médiation par un canal d'événements. La connexion producteur-consommateur est directe et se fait à l'initialisation du composant. *Publisher* est la catégorie de source d'événements qui accepte plusieurs consommateurs (un vers n). Dans ce cas, l'abonnement d'un consommateur à un type d'événements est délégué à un canal d'événements fourni par la structure d'accueil du composant. La seule source pour ce canal d'événements est l'instance de composant pour qui il a été créé.

Un puits d'événements permet à un composant de recevoir des événements d'un certain type. C'est une facette dont le type est un consommateur d'événements. Il n'y a pas ici de différenciation entre connexion (pour *réceptacle*) et abonnement (source d'événements). Le

composant ne contrôle pas ce récepteur, il le rend public pour recevoir des événements. Le *puits* peut dont recevoir des événements en provenance de plusieurs producteurs.

- **Catégories de composants**

Le modèle de composants CCM définit quatre type de composants :

- **Service :** les composants *service* sont des composants sans état interne et sans identité. Leur durée de vie correspond à la durée de traitement requis par l'invocation d'une opération. Le patron de conception utilisé par le client est la *fabrique* (Gamma 1995). Un composant *service* peut être créé à la demande, ou être géré sous la forme d'un pool de composants. (cf. figure 112).
- **Session :** les composants *session* sont des composants avec un état *volatil*, et une *identité* qui n'est pas persistante. La durée de vie d'un tel composant est une interaction (suite d'invocations d'opérations) de la part du client. Ce type de composant est idéal pour implanter des itérateurs, des interfaces Homme-Machine ou tout objet similaire. Le patron de conception utilisé est aussi la *fabrique* (Gamma, 1995).
- **Processus :** les composants *processus* sont des composants avec un état persistant géré par le *composant* ou par le *container*. Le fait que cet état soit persistant n'est pas visible pour le client. L'identité du composant est persistante et gérée par le composant. Elle est rendue visible au client via des opérations définies par l'utilisateur. Le patron utilisé est toujours la *fabrique* (Gamma, 1995). Ce type de composant est utilisé pour modéliser les objets qui représentent des processus métiers, plutôt que des entités.
- **Entité :** les composants entités sont des composants semblables aux composants processus pour la persistance. Néanmoins, cette persistance est visible pour le client grâce à une *clé primaire* (déclarée dans la définition de la fabrique de composants). Les patrons utilisés dans le cadre des composants entités sont la *fabrique* (Gamma, 1995) et la *recherche*.

b) Modèle de programmation

Le modèle de programmation s'intéresse à l'implantation des composants. L'implantation d'un composant est définie comme étant la composition de sous-composants ou d'artefacts logiciels (exécuteurs). Le modèle de programmation définit le moyen de passer du modèle abstrait à un composant logiciel.

- **Eléments de comportement**

Ils sont appelés « exécuteurs ». Ce sont des artefacts de programmation qui offrent le comportement d'un composant ou d'une maison de composants. En général, le terme d'exécuteur de composant correspond à la classe qui implante la classe de composants.

- **Composition**

C'est le principe de base pour l'implantation de composants. L'implantation d'un composant de haut niveau comprend souvent un ensemble potentiellement complexe de sous-composants. Ces sous-composants présentent les relations et le comportement spécifiques du composant. Les facettes et les réceptacles appartenant à diverses instances des sous-composants sont connectés ensemble pour créer des canaux de communication synchrones alors que les sources et puits d'événements sont connectés ensemble pour donner des canaux de communication asynchrones. La description de cette composition est faite à l'aide du langage CIDL (Component Interface Description Language) qui permet de décrire la structure de l'implantation d'un composant et les éventuelles délégations effectuées par certaines

entités vers d'autres entités de la structure de composition. Cette description sera par la suite compilée à l'aide du compilateur et générera le code non fonctionnel du composant.

c) Modèle de paquetage

- **Paquetage de composant**

Une unité de déploiement (fichier ZIP) contient la description externe d'un composant, une ou plusieurs implantations du composant et un ensemble de descripteurs contenant des informations sur le composant ainsi que des caractéristiques non-fonctionnelles devant être appliquées sur les instances de composants.

2. Environnement d'exécution

Le modèle d'exécution est organisé autour des composants de fabrique de composants, container de composants et serveur de composants (cf. figure 112)

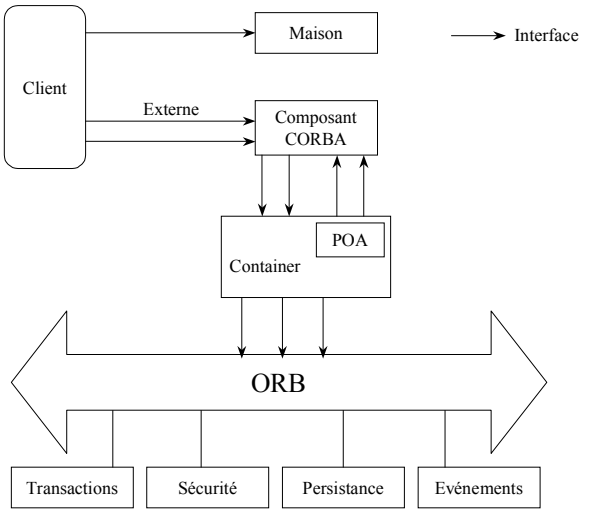


Figure 112. *L'architecture du modèle de programmation des containers.*

- **Fabrique de composants**

Une fabrique (ou maison) de composants par défaut est implantée automatiquement. Son implantation est générée par des outils en utilisant les informations de la vue externe du composant. Les fabriques spécialisées doivent être implantées par le développeur du composant. Les opérations de fabrique sont principalement utilisées par les clients, mais peuvent l'être aussi par l'implantation du composant (pour localiser sa fabrique par exemple).

Il existe quatre politiques différentes de gestion du cycle de vie des instances de composants :

- **Méthode :** le composant est activé à chaque requête, et désactivé aussitôt après.
- **Transaction :** le composant est activé pour la première requête de la transaction et désactivé à la fin de la transaction.
- **Composant :** le composant est activé à la première requête et ne sera désactivé que si son implantation le demande (requête au container)
- **Container :** le composant est activé à la première requête et ne sera désactivé que lorsque le container déterminera qu'il doit l'être.

- **Container de composants**

Un container est un environnement d'exécution pour une instance de composant CORBA. Toutes les instances de composants, quel que soit leur type, sont créées et gérées par un

container. Une instance de composant ne peut pas exister sans être supportée par un container. D'autre part, un type de containers ne peut accueillir qu'un unique type de composants pour lequel il a été conçu. Deux types de containers ont été définis : des containers *volatils* et des containers *persistants*.

Un container offre un ensemble standard de services aux instances de composants qu'il héberge. Les composants et les maisons de composants sont déployés dans des containers grâce à l'utilisation d'outils spécifiques. Ces outils permettent, entre autres, de générer les extensions utiles aux containers pour les accueillir.

Comme nous l'avons vu, il existe quatre catégories de composants. C'est pourquoi quatre catégories de containers spécialisées sont définies (plus un container vide) pour une utilisation dans le contexte du CCM. D'autre part, pour des raisons de compatibilité, les deux types de containers EJB (session et persistant) sont aussi inclus dans la spécification CCM.

- **Empty :** le container vide n'offre aucun framework de développement. Il offre directement au développeur les fonctionnalités de CORBA, ainsi que les fonctionnalités des composants (interfaces multiples, packaging, déploiement).
- **Service :** le container *service* implante l'environnement d'exécution des composants services.
- **Session :** le container *session* implante l'environnement d'exécution des composants session.
- **Processus :** le container *processus* implante l'environnement d'exécution des composants processus.
- **Entity :** le container *entity* implante l'environnement d'exécution des composants *entity*. Il offre des opérations qui associent une clé à un identifiant persistant (pid). A une instance de composant est associée une clé unique. La création d'une instance implique la création d'une entrée dans une « base de données ». Les fabriques peuvent être utilisées par les clients et par l'implantation d'interfaces fournies par le composant.

- **Serveur de composants**

Un serveur de composants est un processus qui contient un nombre arbitraire de containers de composants. Chaque type de containers est associé à un type d'implantation de container qui implique un mode d'interaction avec le POA (Portable Object Adapter) et l'ORB. Chaque type de containers inclut un POA spécialisé géré par un manager de container responsable de la création et de la destruction de containers. Un manager de containers est une fabrique de containers qui est créée lors de la phase d'installation et de déploiement. La création du container, du POA, la configuration des politiques, la liaison avec les services CORBA, … sont définies à partir du descripteur de déploiement.

- **Portable Object Adapter (POA)**

Un POA est utilisé pour créer des références qui seront exportées vers des clients. Il est aussi utilisé pour gérer l'activation d'instances de composants lorsque des requêtes sont reçues. Le fait de créer un container implique, en général, la création d'un POA que ce container utilisera. Pour l'installation d'un composant, une phase d'initialisation est nécessaire entre le composant et le container. Cette phase permet l'échange de références entre l'implantation du composant et le container.

- **Services CORBA**

Le type de containers définit les services CORBA nécessaires au container. Les types de containers de la spécification peuvent utiliser les services de sécurité, de transaction, de

persistance, de notification et de nommage. Lors de la phase de création d'un container, l'accessibilité à ces services doit être établie. Cela implique deux choses : d'une part trouver le service sur le bus, et d'autre part assurer la configuration initiale requise par un service.

Les types de containers définis dans la spécification offrent des frameworks pour le déploiement de composants CORBA. Chaque framework gère les interactions avec l'ORB, le POA et les services de CORBA selon la demande du composant. Cela permet au développeur de se concentrer sur la logique de l'application et d'oublier les problèmes techniques.

Annexe E : Les composants Fractal

Cette annexe résume d'une part les concepts de base du modèle Fractal et d'autre part des exemples d'environnements d'execution de ce modèle. Ce modèle est le fruit d'une collaboration entre France Telecom R&D et l'INRIA (Fractal, 2005).

1. Les concepts du modèle Fractal

- **Opération**

Les opérations sont les interactions de base entre les composants. Elles sont de deux types : unidirectionnelles et bidirectionnelles. Une opération unidirectionnelle consiste en l'invocation d'une opération alors qu'une opération bidirectionnelle consiste en l'invocation d'une opération et en la récupération d'un résultat (référence d'interface, valeur, etc.).

- **Interface**

Les interfaces jouent un rôle central dans Fractal. Elles appartiennent à deux catégories distinctes : Les interfaces métier et les interfaces de contrôle. La membrane d'un composant est l'ensemble de ses interfaces de contrôle. L'ensemble des interfaces métier et de contrôle d'un composant définit son type. Les interfaces métier sont les points d'accès externes au composant alors que les interfaces de contrôle prennent en charge des propriétés non fonctionnelles du composant comme la gestion de son cycle de vie ou de ses liaisons.

Chacune des interfaces des deux catégories précédemment présentées peuvent être de deux types : client et serveur. Une interface serveur reçoit les invocations et peut retourner des résultats de l'invocation d'une opération bidirectionnelle. Une interface client émet une invocation d'opération et peut recevoir les résultats d'invocation d'une opération bidirectionnelle. Le nombre d'interfaces offertes ou requises par un composant Fractal peut changer pendant son exécution.

Une interface Fractal peut être externe ou interne. Une interface externe est visible de l'extérieur du composant alors qu'une interface interne d'un composant composite ne peut être invoquée que d'un des composants qui le composent.

- **Composant**

Les composants Fractal possèdent une vue externe appelée *ComponentType* constituée par un ensemble d'interfaces qui sont fournies ou requises et peuvent être obligatoires ou optionnelles. Si elle est disponible une interface optionnelle requise (respectivement fournie) sera utilisée (respectivement fournie) par le composant ; dans le cas contraire, le composant l'ignore.

Le principe de séparation des préoccupations est également appliqué au niveau de la structure d'un composant Fractal. L'implantation d'un composant Fractal est divisée en deux parties :

- Le *contrôleur* : il implante une ou plusieurs interfaces de contrôle dont un certain nombre sont définies dans la spécification du modèle de composants comme l'introspection, la configuration, la sécurité, etc. Le contrôleur agit sur le contenu pour lui fournir les services non fonctionnels. Il joue un rôle similaire à celui du conteneur dans les modèles de composants EJB ou CCM. Le *contrôleur* décide de la visibilité des interfaces offertes par un composant ou l'un de ses sous-composants. Il peut ainsi publier ou cacher une ou plusieurs interfaces de ses sous-composants (respectivement lui même). Un contrôleur peut avoir des interfaces internes qui ne sont visibles que par les sous-composants.
- Le *contenu* : il peut contenir du code fonctionnel ou bien un ensemble d'autres contrôleurs. Dans le cas d'un composant composite, le *contenu* contient l'ensemble des *contrôleurs* de ses sous-composants.

Une instance de composant peut être partagée entre deux composants composites différents. Dans ce cas de figure, le composant partagé est sujet au contrôle des *contrôleurs* des deux composants composites. La sémantique exacte de ce partage est assurée par le *contrôleur* du composant composite qui encapsule toute la configuration des composants (les deux composants composites et le composant partagé).

Les instances de composants sont instanciées à partir de fabriques de composants (Gamma, 1995).

- **Composition**

Le modèle de composants Fractal supporte trois techniques de composition : impérative par langage de programmation, déclarative et visuelle. La composition impérative se réalise à travers différentes interfaces de contrôle dont *AttributeController* qui permet de réaliser la configuration d'une instance, *BindingController* qui permet de réaliser la connexion des instances et *ContentController* qui permet de créer des composites. La composition déclarative se réalise en utilisant le langage de description d'architecture Fractal (ADL Fractal). La composition visuelle est supportée dans un outil appelé FractalGUI.

- **Paquetage de composant**

La spécification de Fractal ne définit pas la manière de conditionner un composant.

2. Environnements d'exécution

Nous avons identifié quatre implémentations de l'environnement d'exécution des composants Fractal : *Julia*, *Think* (Fassino, 2002), *Proactive* (Proactive, 2005) et *FracTalk* (Fractalk, 2005)

Julia est l'implantation de référence et elle est fournie avec Fractal. Le framework *Julia* permet de décrire le contrôleur à partir d'un langage spécialisé et réalise ensuite un mélange du code de contrôle et du code fonctionnel à travers une approche de tissage.

Think est un framework logiciel pour la construction de systèmes à base de composants. *Think* fournit un ensemble de composants pour la construction de noyaux de systèmes d'exploitation (Kernel). Cette approche facilite la réutilisation et offre une flexibilité permettant de réduire le temps de développement des systèmes d'exploitation et de leur maintenance. *Kortex* est une implantation du framework *Think*.

ProActive est une implantation pour les objets actifs distribués. Elle se présente sous la forme d'un framework pour la construction de grilles de calcul distribuées, parallèles et concurrentes. *Proactive* offre aussi des fonctionnalités pour la sécurité et la mobilité.

FracTalk est l'implantation du modèle de composants Fractal en Smalltalk.

Annexe F : Les composants métier Symphony

Nous nous limitons ici à donner les concepts de modèle de composants métier Symphony. D'autres aspects comme les relations entre composants et la démarche Symphony sont largement décrits dans la thèse d'Ibtissem HASSINE (Hassine, 2005).

a) Les composants métier Symphony

Le modèle métier Symphony spécifie trois types de composants. Aux deux catégories fondamentales de composants métier **processus** et **produit**, Symphony ajoute un autre niveau de composants : les composants métier **données** (cf. figure 113) :

- un composant processus permet de décrire un processus applicatif (ex : le processus Facturation),
- un composant produit constitue un élément de la base des composants métier (ex : Client, Contrat, etc.),
- un composant données représente des données de référence (ex : Qualification Clientèle).

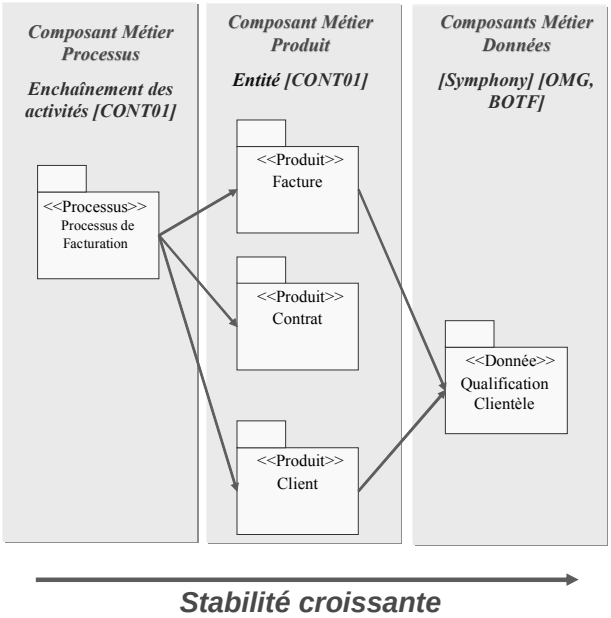


Figure 113. *Classification des composants métier de Symphony.*

b) Architecture d'un composant métier Symphony

Dans Symphony, l'architecture des composants métier (CM) est une structuration inspirée de la technique CRC (Classe-Responsabilité-Collaboration) (Wirfs-Brock, 1990). Un CM est modélisé par un paquetage composé de trois parties (cf. figure 114) : une partie contrat avec l'extérieur (*ce que je sais faire*), une partie structurelle (*ce que je suis*) et une partie collaboratrice (*ce que j'utilise*).

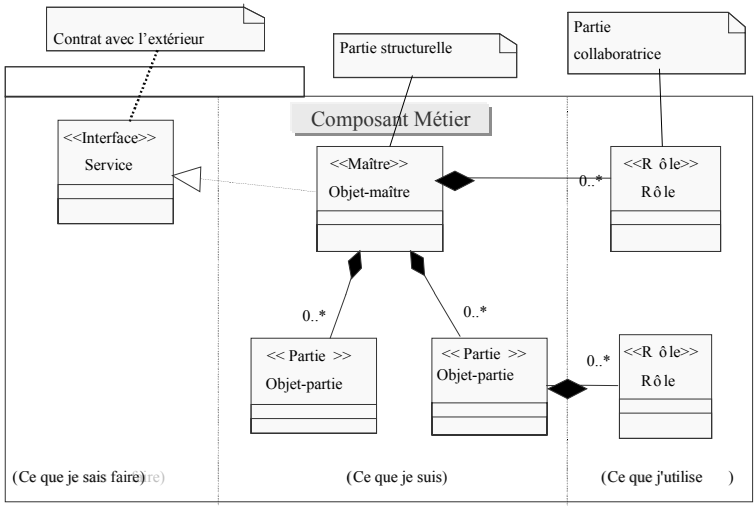


Figure 114. *Architecture d'un composant métier Symphony.*

- **Ce que je sais faire**

L'objet interface modélise les points d'entrée, le contrat du CM. Tout accès au CM doit bien entendu passer par son objet interface.

- **Ce que je suis**

L'objet maître est l'objet principal du composant. Il est qualifié de hiérarchiquement supérieur et est identifiable par tout acteur. Il réalise les opérations définies dans son objet interface, c'est-à-dire celles réalisant les cas d'utilisation sous sa responsabilité et d'autre part les services qu'il assure vis-à-vis des autres CM (cf. partie Rôle). L'objet maître est autonome : une fois créé, il ne dépend pas directement d'autres objets si ce n'est au travers de ses propres règles de gestion.

L'objet maître justifie son existence par la prise de responsabilités dans les finalités du système. Dans la figure 115, l'objet maître est le Contrat passé avec la société de transport. Il a en charge la réalisation de l'opération Créer Contrat. Pour être un objet maître, l'objet candidat doit vérifier les caractéristiques suivantes : être le centre du problème et exister en un seul exemplaire lorsque l'on traite le problème (mono-instancié).

Un ou plusieurs objets partie complètent l'objet maître auquel ils sont reliés par une relation de type composite. Un objet partie correspond à une structuration partielle des attributs de l'objet maître mettant en évidence un aspect métier du composant. Dans notre exemple, les divers abonnements passés dans le cadre d'un contrat sont modélisés par un objet partie (ils peuvent être des abonnements pour des moyens de transport différents, pour des bénéficiaires différents (par exemple, le conjoint ou les enfants du titulaire du contrat), etc.).

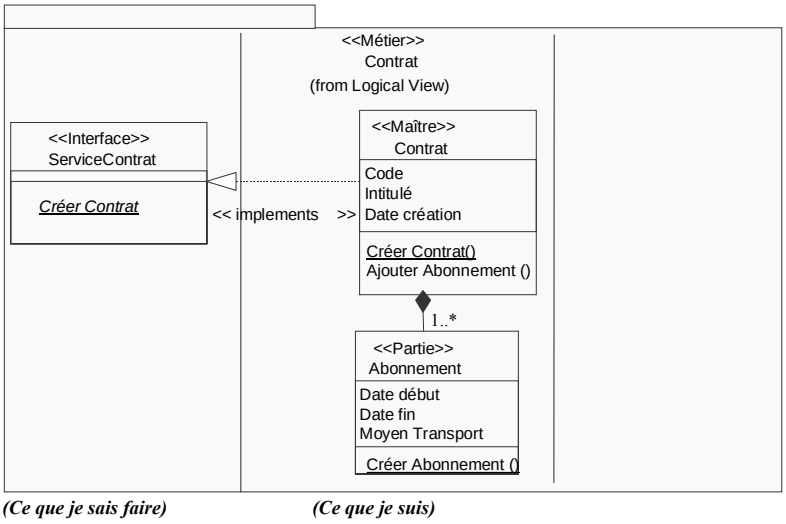


Figure 115. *Exemples d'objets interface, maître et partie dans Symphony.*

- **Ce que j'utilise**

L'objet rôle est un serviteur de l'objet maître. Un service offert par un autre composant métier est appelé au travers d'un rôle. Un objet rôle représente un fournisseur de services auprès du composant client. Autrement dit, un fournisseur est un autre composant qui rend un ou plusieurs services, ces services sont appelés chez un composant client au travers d'un objet rôle. Les attributs décrits dans l'objet rôle sont représentés par la notation d'attribut dérivé " / ". C'est par exemple le cas des attributs nom, prénom et adresse de Client. Un rôle peut également détenir des attributs dont la pertinence n'est effective que dans le cadre du composant. C'est par exemple le cas de l'attribut *Qualité* de la classe *Titulaire* qui correspond à la situation familiale du titulaire du contrat passé avec la société de transport.

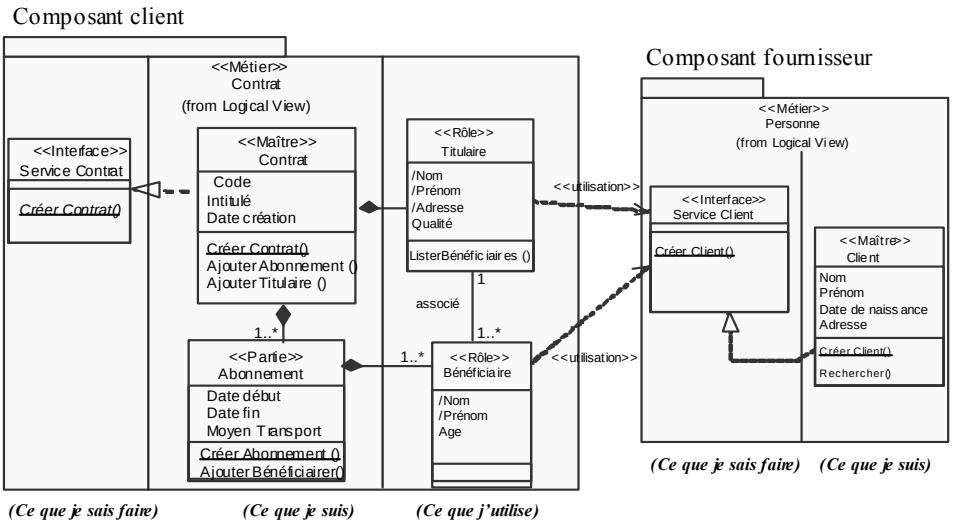


Figure 116. *Exemple de rôles dans Symphony.*

Annexe G : Les composants Actoll

Dans le cadre de cette thèse nous avons travaillé avec notre partenaire industriel la société Actoll en participant au projet Initiative Centr'Actoll. Dans le cadre de cette coopération nous avons étudié des modèles de systèmes d'information construits par la société Actoll pour identifier les composants métiers du domaine de la billettique et les systèmes d'information de transport (Khayati, 2003a).

Dans la deuxième partie de la coopération avec la société Actoll nous nous somme intéressé aux processus utilisés par Actoll en les formalisant en utilisant la technologie des patrons. Nous avons formalisé le processus de conduite de projet chez Actoll et nous l'avons complété en proposant des fragments d'une démarche générique pour le développement de SI par réutilisation de composants et des fragments de démarche pour l'intégration et l'adaptation des composants métier Symphony (Khayati, 2004b).

La figure 117 présente la cartographie et résume les relations qui existent entre les différents types de composants Actoll (processus et produit).

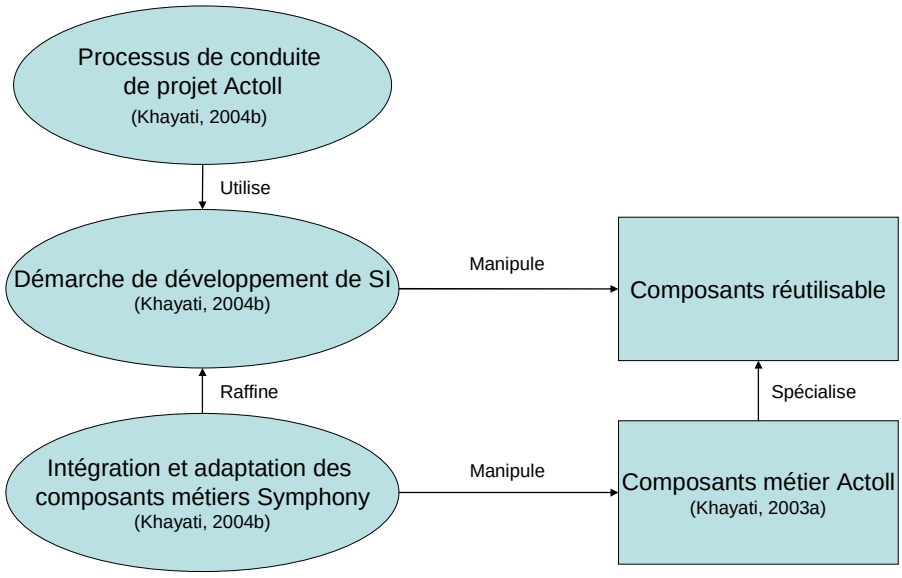
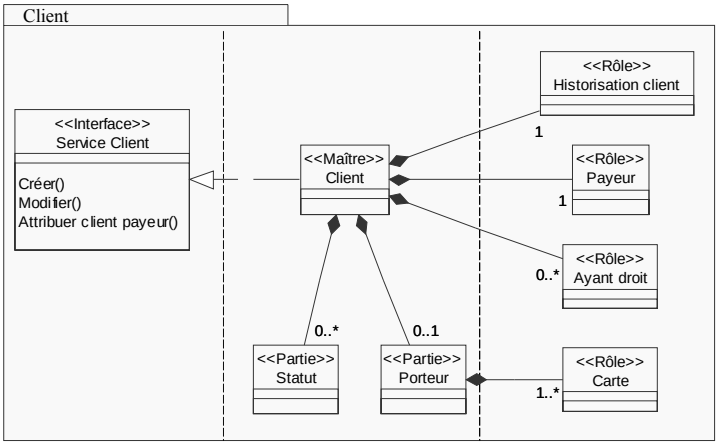


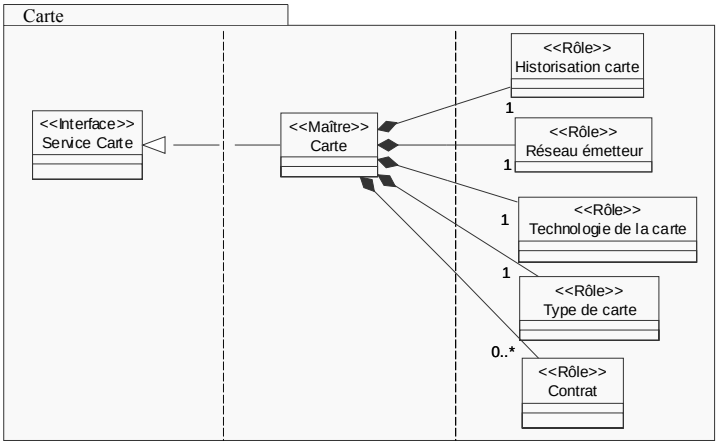
Figure 117. *Cartographie des composants Actoll (produit et processus).*

Nous présentons dans cette annexe une partie des composants métier identifiés dans le cadre de notre participation dans l'initiative Centr'Actoll.

1. Le composant Client

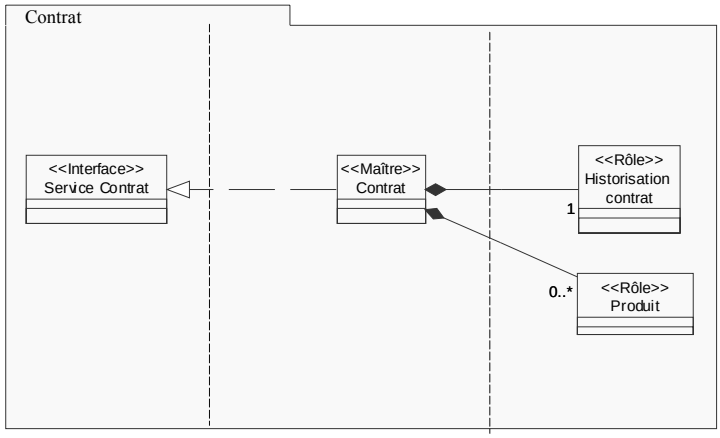
Nom	Client
Description	Tout client qui utilise le réseau de transport Actoll est représenté au sein du système d'information avec le composant *Client*.
Utilise	Le composant *Client* utilise d'autres composants via les rôles suivants : - Historisation client : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'un *Client*. - Payeur : c'est un rôle du composant *Client*. Le rôle *Payeur* indique le client payeur qui paye pour le client en cours. Si le client paye pour lui même, alors le rôle *payeur* pointe sur le composant *Client* qui le contient. - Ayant droit : c'est un rôle du composant *Client*. Le rôle *Ayant droit* indique le *Client* à travers lequel le client en cours a des droits. Les enfants de moins de 7 ans peuvent par exemple avoir le droit de circuler sur le réseau gratuitement ou avec un tarif réduit si l'un de leurs parents est un client. - Carte : c'est un rôle du composant *Carte*. Le rôle *Carte* indique les cartes que possède un client. Le rôle *Carte* est relié à l'objet maître via l'objet partie *Porteur* pour donner la possibilité au composant *Client* de réagir différemment suivant la carte qu'il utilise.
Utilisé par	Le composant *Client* est utilisé par les composants suivants : - Client : il joue les rôles de *Payeur* et *Ayant droit*. - Agent : il joue le rôle de *Client* (dans la solution présentée en section 4.5.1).
Modèle	
Participants	Le composant *Client* contient les classes suivantes : - Client : la classe maître du composant *Client*. - Statut : une classe partie du composant *Client*. Cette classe gère le statut du client. - Porteur : une classe partie du composant *Client*. Cette classe est instanciée dans le cas où le client est porteur de cartes.

2. Le composant Carte

Nom	Carte
Description	Les cartes sont les titres de transport qu'utilise un client du réseau pour se déplacer.
Utilise	Le composant *Carte* utilise d'autres composants via les rôles suivants : - Historisation carte : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'une *Carte*. - Réseau émetteur : c'est un rôle du composant Réseau émetteur. Il indique le réseau émetteur dans le cas où le réseau de transport utilisé par le client est composé de plusieurs sous réseaux (bus, tramway,…). - Type de carte : c'est un rôle du composant *Type de carte*. Une carte peut avoir plusieurs types (déplacement, dépannage, navette ou habilitation). - Contrat : c'est un rôle du composant *Contrat*. Une carte peut contenir zéro ou plusieurs contrats. - Technologie de la carte : c'est un rôle du composant *Technologie de la carte*. Une carte utilise une technologie
Utilisé par	Le composant *Carte* est utilisé par les composants suivants : - Client : il joue le rôle de *Carte*.
Modèle	
Participants	Le composant *Carte* contient les classes suivantes : - Carte : la classe maître du composant *Carte*.

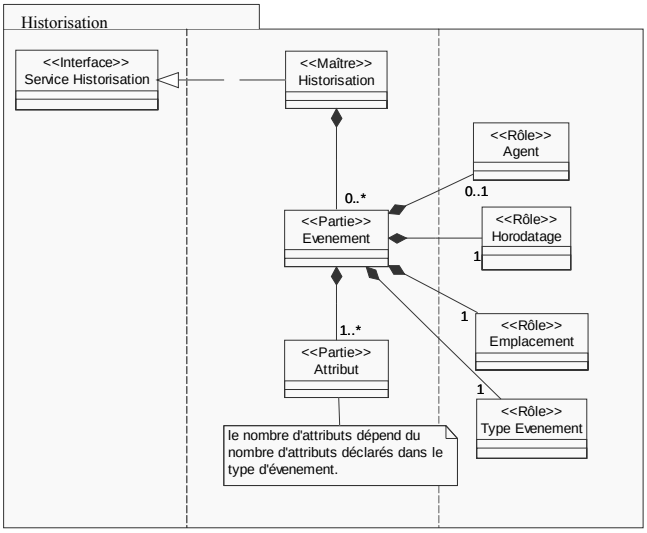
3. Le composant Contrat

Nom	Contrat
Description	Les contrats d'un client sont hébergés sur une carte. Chaque contrat donne des informations sur le produit qu'il ajoute à la carte du client.
Utilise	Le composant *Contrat* utilise d'autres composants via les rôles suivants : - Historisation contrat : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de

	création ou de changement d'état d'un *Contrat*. - Produit : c'est un rôle du composant *Produit*. Ce rôle permet au composant *Contrat* de gérer et de modifier les composants *Produit* qu'il contient.
Utilisé par	Le composant *Contrat* est utilisé par les composants suivants : - Carte : il joue le rôle de *Contrat*.
Modèle	
Participants	Le composant *Contrat* contient les classes suivantes : - Contrat : la classe maître du composant *Contrat*.

4. Le composant Historisation

Nom	Historisation
Description	Le composant *Historisation* s'occupe de l'historisation des événements générés par les autres composants.
Utilise	Le composant *Historisation* utilise d'autres composants via les rôles suivants : - Agent : c'est un rôle du composant *Agent*. Ce rôle indique l'agent qui a généré l'évènement. - Horodatage : c'est un rôle du composant *Horodatage*. Ce rôle fournit l'heure et la date de l'événement. - Emplacement : c'est un rôle du composant *Emplacement*. Ce rôle fournit l'emplacement où l'événement est déclenché. - Type Evénement : c'est un rôle du composant *Type événement*. Il donne des informations sur le type des évènements par exemple les noms et les types des attributs des événements.
Utilisé par	Le composant *Historisation* est utilisé par les composants suivants : - Carte : le composant *Historisation* joue le rôle de *Historisation carte*. - Client : le composant *Historisation* joue le rôle de *Historisation client*. - Agent : le composant *Historisation* joue le rôle de *Historisation Agent*. - Contrat : le composant *Historisation* joue le rôle de *Historisation contrat*.

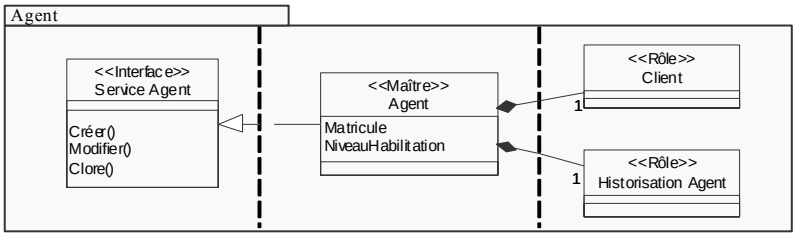
Modèle	
Participants	Le composant *Historisation* contient les classes suivantes : - Historisation : la classe maître du composant *Historisation.* - Evénement : une classe partie du composant *Historisation*. Cette classe est instanciée chaque fois qu'une notification d'événement arrive au composant *Historisation*. - Attribut : une classe partie du composant *Historisation*. Cette classe est instanciée lors de l'instanciation de la classe *Evénement*. Le nombre d'attributs dépend du nombre d'attributs déclarés dans le type d'événement.

5. Variantes du composant Agent

Dans cette section, nous présentons quatre variantes permettant de gérer les agents.

5.1 Composant Agent avec un rôle Client

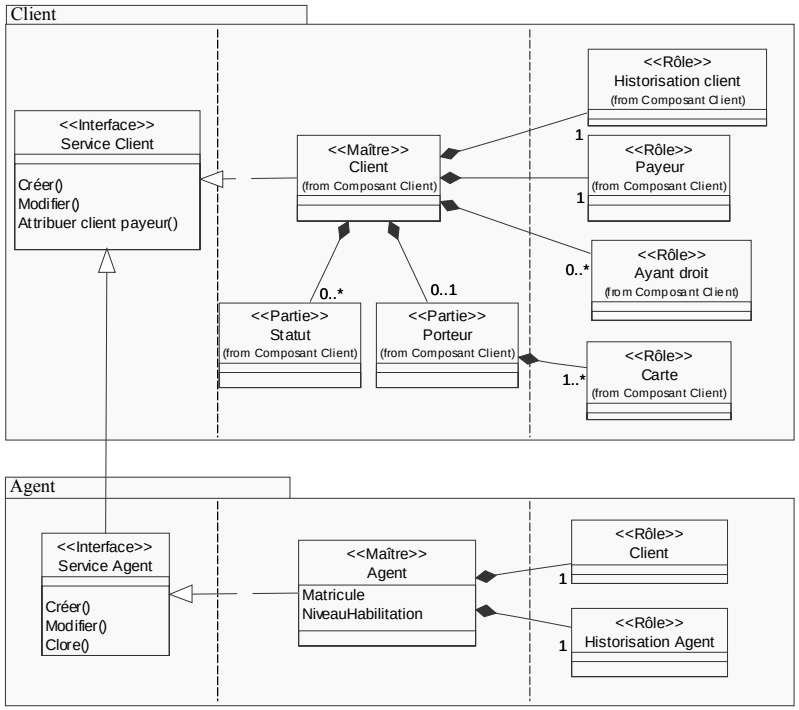
Nom	Agent
Description	Les agents sont les employés qui travaillent dans la société qui gère le réseau de transport utilisant le système d'information d'Actoll.
Utilise	Le composant *Agent* utilise d'autres composants via les rôles suivants : - Client : c'est un rôle du composant *Client*. Un agent est lui même un utilisateur du réseau de transport donc il peut avoir une carte et il peut être un client payeur pour ses enfants. - Historisation agent : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'un *Agent*.
Utilisé par	Le composant *Agent* est utilisé par les composants suivants : - Historisation : il joue le rôle de *Agent*.

Modèle	
Participants	Le composant *Agent* contient les classes suivantes : - Agent : la classe maître du composant *Agent.*

5.2 Composant Agent et héritage d'interface

Cette solution permet de considérer directement qu'un agent est un client. Le composant Agent hérite de l'interface du composant Client donc tous les agents se comportent comme des clients. Néanmoins, Agent et Client peuvent avoir chacun une implantation différente.

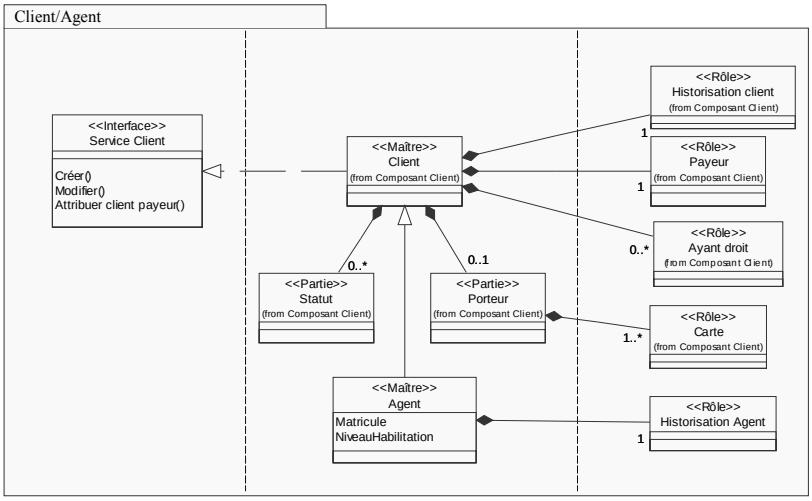
Nom	Agent
Description	Les agents sont les employés qui travaillent dans la société qui gère le réseau de transport utilisant le système d'information d'Actoll.
Utilise	Le composant *Agent* utilise d'autres composants via les rôles suivants : - Client : c'est un rôle du composant *Client*. Un agent est lui même un utilisateur du réseau de transport donc il peut avoir une carte et il peut être un client payeur pour ses enfants. - Historisation agent : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'un *Agent*.
Utilisé par	Le composant *Agent* est utilisé par les composants suivants : - Historisation : il joue le rôle de *Agent*.
Héritage d'interface	Le composant *Agent* spécialise l'interface du composant *Client*.

Modèle	
Forces	Le composant *Agent* hérite de l'interface du composant *Client* donc tous les agents sont des clients. *Agent* et *Client* peuvent avoir chacun une implantation différente.
Participants	Le composant *Agent* contient les classes suivantes : - Agent : la classe maître du composant *Agent.*

5.3 Agent vu comme spécialisation de client

Cette solution permet de considérer qu'un certain nombre de clients peuvent être des agents. Le composant *Agent* n'est pas présent dans le système en tant que composant à part entière, mais il est intégré dans le composant *Client*. Le tableau suivant propose donc une variante du composant *Client* qui intègre le concept d'agent.

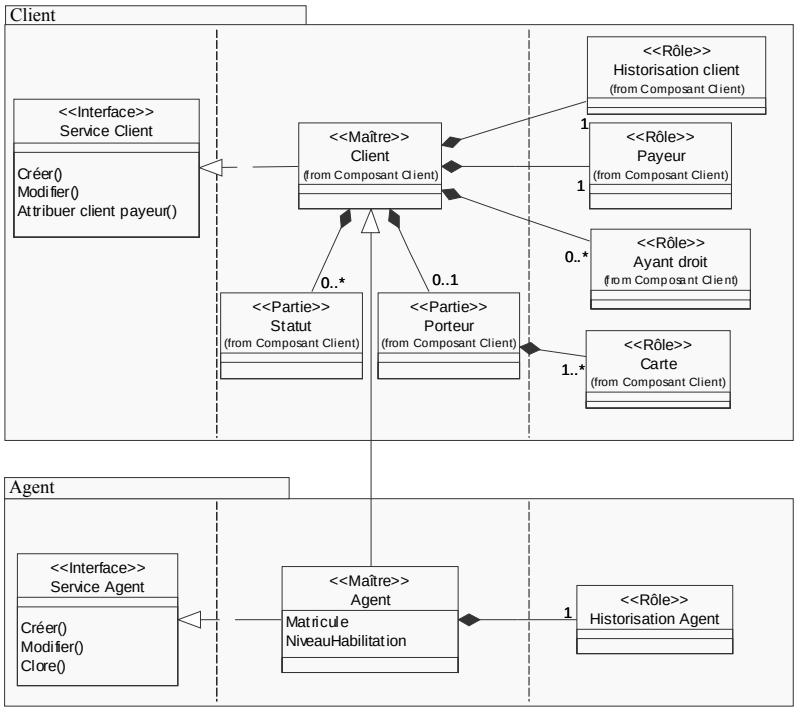
Nom	Client/Agent
Description	Tout client qui utilise le réseau de transport Actoll est représenté au sein du système d'information avec le composant *Client*.
Utilise	Le composant *Client* utilise d'autres composants via les rôles suivants : - Historisation client : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'un *Client*. - Historisation agent : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'un *Agent*. - Payeur : c'est un rôle du composant *Client*. Le rôle *Payeur* indique le client payeur qui paye pour le client en cours. Si le client paye pour lui même, alors le rôle *payeur* pointe sur le

	composant *Client* qui le contient. - Ayant droit : c'est un rôle du composant *Client*. Le rôle *Ayant droit* indique le *Client* à travers lequel le client en cours a des droits. Les enfants de moins de 7 ans peuvent par exemple avoir le droit de circuler sur le réseau gratuitement ou avec un tarif réduit si l'un de leurs parents est un client. - Carte : c'est un rôle du composant *Carte*. Le rôle *Carte* indique les cartes que possède un client. Le rôle *Carte* est relié à l'objet maître via l'objet partie *Porteur* pour donner la possibilité au composant *Client* de réagir différemment suivant la carte qu'il utilise.
Utilisé par	Le composant *Client* est utilisé par les composants suivants : - Client : il joue les rôles de *Payeur* et *Ayant droit*.
Modèle	
Forces	La classe maître *Agent* hérite de la classe maître *Client*. Ceci implique que tous les agents sont des clients et que *Agent* surcharge et enrichit les propriétés de *Client*. De plus, *Agent* ajoute des propriétés spécifiques à l'agent. Cette solution est utilisée si la classe maître Agent ne modifie pas beaucoup la classe maître *Client* et que la classe maître *Agent* ne subit pas beaucoup de modifications pendant le cycle de vie du système d'information. Sinon le composant *Client* peut devenir très complexe.
Participants	Le composant *Client* contient les classes suivantes : - Client : une classe maître du composant *Client*. - Statut : une classe partie du composant *Client*. Cette classe gère le statut du client. - Porteur : une classe partie du composant *Client*. Cette classe est instanciée dans le cas où le client est porteur de cartes. - Agent : une classe maître du composant *Client* spécialisant la classe maître Client. Cette classe est instanciée lorsque le client est aussi un agent.

5.4 Composant Agent avec spécialisation de classes de composants

Dans cette solution il y a spécialisation de la classe maître du composant *Client*. La classe maître du composant *Agent* hérite de la classe maître du composant *Client*. Il y a donc aussi héritage d'interface entre les deux composants. Tous les agents sont des clients et *Agent*

surcharge et enrichit les propriétés de *Client*. De plus, il ajoute les propriétés spécifiques à l'agent et est un composant à part entière. Cette solution permet une meilleure évolutivité des deux composants.

Nom	Agent
Description	Les agents sont les employés qui travaillent dans la société qui gère le réseau de transport utilisant le système d'information d'Actoll.
Utilise	Le composant *Agent* utilise d'autres composants via les rôles suivants : - Client : c'est un rôle du composant *Client*. Un agent est lui même un utilisateur du réseau de transport donc il peut avoir une carte et il peut être un client payeur pour ses enfants. - Historisation agent : c'est un rôle du composant *Historisation*. Ce rôle sert à historiser les événements de création ou de changement d'état d'un *Agent*.
Utilisé par	Le composant *Agent* est utilisé par les composant suivants : - Historisation : il joue le rôle de *Agent*.
Spécialisation de classes de composants	Le composant Agent spécialise la classe maître du composant Client.
Modèle	
Forces	La classe maître du composant *Agent* hérite de la classe maître du composant *Client*. Ceci implique que tous les agents sont des clients et que *Agent* surcharge et enrichit les propriétés de *Client*. De plus, *Agent* ajoute les propriétés spécifiques à l'agent . Une meilleure évolutivité du système d'information est possible grâce au composant *Agent* qui est ici un composant complètement autonome.

Participants	Le composant *Agent* contient les classes suivantes : - Agent : la classe maître du composant *Agent*. - Statut : une classe partie du composant *Client*. Cette classe gère le statut du client. - Porteur : une classe partie du composant *Client*. Cette classe est instanciée dans le cas où le client est porteur de cartes.

6. Conclusion

Nous avons présenté dans cette annexe un sous ensemble des composants métier que nous avons identifié en collaborant avec le société Actoll dans le cadre de l'initiative Centr'Actoll. Les composants que nous avons identifiés et documentés représentent la première étape dans le processus de capitalisation des connaissances (du savoir). La figure 118 présente une partie de la cartographie des composants métier Actoll. Pour réaliser cette cartographie nous avons du faire certaines hypothèses concernant les types de relations qui existent entre les composants Centr'Actoll. Ainsi, nous supposons que le composant *Historisation* est toujours utilisé avec enrichissement de son interface alors que toutes les autres relations d'utilisation entre les composants Centr'Actoll sont simples. De plus, nous optons pour la solution proposée dans la section 5.4 où le composant *Agent* est une spécialisation de classes de composants du composant *Client*.

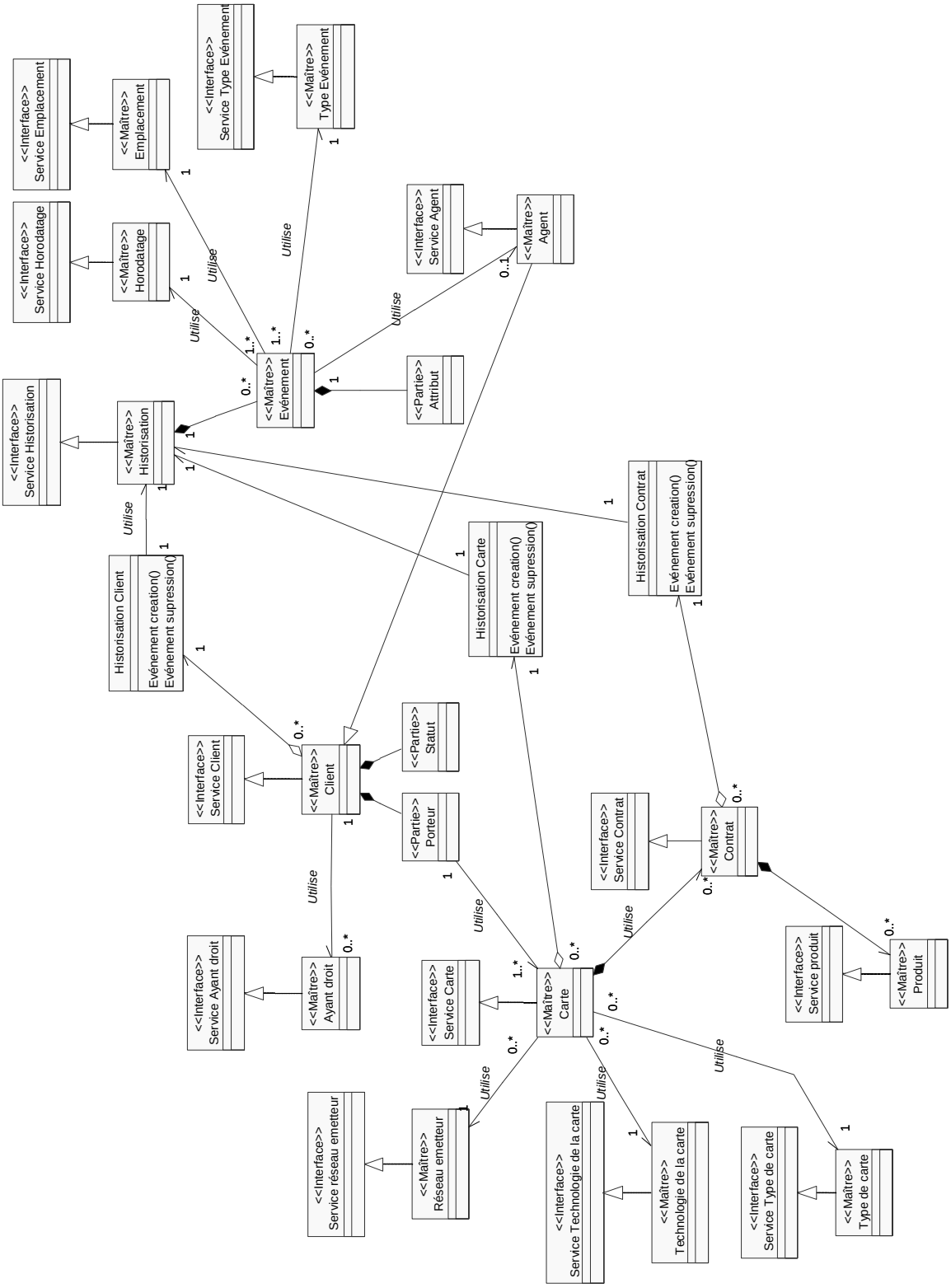


Figure 118. *Une cartographie des composants Initiative Centr'Actoll.*

Bibliographie

Bibliographie

A

(Alexander, 1979) Alexander C., *The Timeless Way of Building*, Oxford University Press, 1979.

(Ambler, 1998) Ambler S. W., *An Introduction to Process Pattern*, AmbySoft White paper, 1998. http://www.Ambysoft.com.

(André, 1994) André J., *Moving From Merise to Schlaer and Mellor*, SIG-Publication vol n°3, 1994.

(Arnaud, 2004) Arnaud N., Front A., Rieu D. *Une approche par méta-modélisation pour l'imitation des patrons*, Dans les actes du XXIIème congrès INFORSID 2004, Biarritz, Mai 2004.

(ASSET, 1993) ASSET, *Asset submittal guidelines*, Technical Report SAIC-92/7625-00, version 1.2, ASSET, Morgantown, WV, December 1993.

B

(Badaro, 1991) Badaro N., Moineau Th., *Rose-ada: A method and a tool to help reuse ada code*, In Proceedings, Ada Europe Conference. Springer Verlag (Lecture Notes in Computer Science, vol 499), 1991.

(Barbier, 2002) Barbier F., Cauvet C., Oussalah M., Rieu D., Bennasri S., Souveyet C., *Composants dans l'Ingénierie des Systèmes d'Information : Concepts clés et techniques de réutilisation*, Assises du GDR I3, Nancy, Décembre 2002.

(Barbier, 2002a) Barbier F., *Business Component-Based Software Engineering*, Kluwer, 2002.

(BEA, 1999) BEA et al. *CORBA Components: Joint Revised Submission. OMG*, OMG TC Document orbos/99-07-{01..03,05} orbos/99-08{05..07,12,13}, August 1999.

(Bidoit, 1989) Bidoit M., *Pluss, un langage pour le développement de spécifications algébriques modulaires*, PhD Thesis, Université de Paris-Sud – Centre d'Orsay, 1989.

(Blanc, 2005) Blanc X., *MDA en action - Ingénierie logicielle guidée par les modèles*, Editon Eyrolles, 2005.

(Boudriga, 1992) Boudriga N., Mili A., Mittermeir R.T., *Semantic based software retrieval to support rapid prototyping*, structured programming, 13:109-127, 1992.

(Box, 1998) Box D., *Essential COM. Addison-Wesley*, Janvier 1998.

(Bruneton, 2004) Bruneton E., Coupaye T., Stefani J.B., *The Fractal Component Model*, 2004, http://fractal.objectweb.org/specification/index.html.

(Buschmann, 1996) Buschmann F., Meunier R., Sommerald P., Stal M., *Pattern-Oriented Software Architecture : A System of Patterns* , éditions willy, 1996.

C

(Casanave, 1996) Casanave C., *Business Object Architectures and Standards*, Data Access Corporation, Miami USA, 1996.

(Cauvet, 2001) Cauvet C. & Rosenthal-Sabroux C. (coordination), *Ingénierie des Systèmes d'Information* dans la série du traité IC2 - Information Commande Communication, Editions HERMES, 2001.

(Cheng, 1992) Cheng B.H.C., Jeng J.J., *Formal methods applied to reuse*, In proceedings, WISR-5, Palo Alto, CA, 1992.

(Chou, 1996) Chou S.C., Chen J.Y., Chung C.G, *A behavior-based classification and retrieval technique for object oriented specification reuse*, Software – Practice and Experience, 26(7):815-832, July 1996.

(Coad, 1992) Coad P., *Object-Oriented Patterns*, Communication of ACM, vol. 35, n°. 9, Septembre 1992.

(Coad, 1995) Coad P., North D., Mayfield M., *Objet Models: Strategies, Patterns & Applications*, Yourdon Press Computing series 1995.

(COM, 1994) COM, *Component Object Model: Technical Overview*, Dr. Dobbs Journal, December 1994,

http://www.microsoft.com/com/wpaper/Com_modl.asp.

(COM, 2005) COM, *COM Fundamentals*, 2005. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/html/faa0dc85-2a66-4c69-acf6-d8d281063512.asp

(Conte, 2001) Conte A., Giraudin J.P., Hassine I., Rieu D., *Un environnement et un formalisme pour la définition, la gestion et l'application de patrons*, Revue ISI "Ingénierie des Systèmes d'Information", numéro spécial "Formalismes et Modèles pour les Systèmes d'Informations", volume 6, numéro 2, Hermès, 2001.

(Conte, 2001a) Conte A., Hassine I., Giraudin J.P., Rieu D., *AGAP : un Atelier de Gestion et d'Application de Patrons*, Congrès Inforsid, Genève, Mai 2001.

(Conte, 2001b) Conte A., Fredj M., Giraudin J-P, Rieu D., *P-Sigma : Un formalisme pour une représentation unifiée de patrons*, Inforsid'01, Martigny, Juin 2001.

(Coplien, 1992) Coplien J. O., *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

(Cybulski, 1993) Cybulski J. L. and Reed K., *The Use of Templates and Restricted English in Structuring and Analysis of Informal Requirement Specifications*, AAITP Technical Report TR024, 1993.

D

(D'Souza, 1999) D'Souza DF., Wills AC., *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison Wesley, Reading, MA, 1999.

(Descombe-Perrière, 2003) Descombe-Perrière C., *AGAP : un atelier de Gestion et d'Application de Patrons*, mémoire CNAM 2003

(Devanbu, 1991) Devanbu P., Brachman R., Selfridge P., Ballard B., *Lassie: A Knowledge-based software information system*. Communications of the ACM, 34(5):34-39, May 1991.

(Dyson, 1997) Dyson P., *Patterns for abstract design*, PhD thesis, University of Essex, 1997.

E

(Eddon, 1998) Eddon G. et Eddon H., *Au cœur de Distributed COM*, Microsoft Press, Les Ulis, France, 1998, ISBN : 2-84082-372-1.

(Ehrig, 1985) Ehrig H., Weber H., *Algebraic specification of modules*, In Formal Models in Programming, North-Holland, Amsterdam. 1985.

F

(Fassino, 2002) Fassino J.P., Stefani J.B., Lawall J., Muller G., *Think: A Software Framework for Component-based*, USENIX 2002 Annual Technical Conference Paper, pp. 73-86, 2002.

(Finch, 1998) Finch L., *So Much OO, So Little Reuse*. Dr. Dobb's Journal, mai 1998, http://www.ddj.com/articles/1998/9875/.

(Fisher, 1995) Fisher B., Kievernagel M., Snelting G., *Deduction based Software component retrieval*, In IJCAI Workshop on Reuse of Proofs, Plans and Programs, Montreal, Quebec, Canada, June 1995.

(Fowler, 1997) Fowler M., *Analysis Patterns – Reusable Object Models*, Addison Wesley, 1997.

(Fractal, 2005) Page web du projet Fractal.

http://fractal.objectweb.org/

(Fractalk, 2005) Fractalk, *FracTalk Documentation*, 2005,

http://csl.ensm-douai.fr/FracTalk/2.

(Frakes, 1987) Frakes W.B., Nejmeh B.A., *An information system for software reuse*, In Proceedings, 10th Minnowbrook Workshop on Software Reuse, Minnowbrook, NY, 1987.

(Frakes, 1987a) Frakes W.B., Nejmeh B.A., *Software reuse through information retrieval*. In Proceedings, 20th Annual Hawaii International Conference on System Sciences, pages 530-535, Kona, HI, January 1987.

(Freitag, 1994) Freitag B., *A Hypertext-Based Tool for Large Scale Software Reuse*, proceedings of the 6th Conference on Advanced Information Systems Engineering (CAISE'94), pages. 283-296, Ultrecht, The Netherlands, 6-10 Jun 1994.

(Front, 1999) Front A., Giraudin J.P., Rieu D., Saint-Marcel C., *Génie Objet, Analyse & Conception de l'évolution d'objets*, éditions Hermès, mai 1999.

G

(Gamma, 1995) Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns – Elements of Reusable Oriented Software*. Addison-Wesley, 1995.

(Gardarin, 1996) Gardarin G., Gardarin O., *Le Client-Serveur*. Eyrolles, 1996.

(Garg, 1990) Garg P., Scacchi W., *A hypertext system to manage software life-cycle documents*, IEEE software, pp. 90-98, mai 1990.

(Gaudel, 1991) Gaudel M.C., Moineau Th., *A theory of software reusability*, In Lecture Notes in Computer Science, Volume 300, pages 115-130. Springer-Verlag, 1991.

(Girardi, 1993) Girardi M.R., Ibrahim B., *A software reuse system based on natural language specifications*, Proceeding of International Conference on Computing and Information (ICCI'93), Sudbury, Ontario, Canada, pages 507-511, 1993.

(Girardi, 1994) Girardi M.R., Ibrahim B., *A Similarity Measure for Retrieving Software Artifacts*, Proceeedings of Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE'94), Jurmala,Latvia, pp. 478-485, June 21-23, 1994.

http://citeseer.nj.nec.com/girardi94similarity.html.

(Girardi, 1995) Girardi M.R., *Classification and Retrieval of Software through their Descriptions in Natural Language*, Ph.D. dissertation, No. 2782, University of Geneva, December 1995.

(Goldberg, 1984) Goldberg A., *SMALLTALK-80: the interactive programming environment*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1984.

(Grosso, 2001) Grosso W., *Java RMI*, O'Reilly, 2001.

(Gzara, 2000) Gzara L., *Les patrons pour l'ingénierie des Systèmes d'Information Produit*, Doctorat de l'INPG, spécialité Génie Industriel, Décembre 2000.

H

(Hall, 1993) Hall R.J., *Generalized behaviour-based retrieval*, In Proceedings, International Conference on Software Engineering, Baltimore, MD, May 1993.

(Hassine, 2002) Hassine I., Rieu D., Bounaas F., Seghrouchni O., *Symphony : un modèle conceptuel de composants métier*, Revue ISI, volume 7, numéro 4, Hermès, 2002.

(Hassine, 2005) Hassine I., *Spécification et formalisation des démarches de développement à base de composants métier : La démarche Symphony*, rapport de thèse, septembre 2005.

(Heineman, 2001) Heineman G., Councill W. T., *Component-Based Software Engineering : Putting the Pieces Together*, Addison-Wesley, 2001.

(Helm, 1991) Helm R., Maarek Y.S., *Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries*, In proceedings, OOPSLA'91, 26(11): 47-61, October 1991.

(Hemer, 2001) Hemer D., Lindsay P., *Specification-based Retrieval Strategies for Module Reuse*, In D. Grant and L. Sterling, editors, Proceedings 2001 Australian Software Engineering Conference, 27-28 August 2001, Canberra, Australia, pages 235-243, IEEE Computer Society, 2001.

(Hull, 1994) Hull D., *Information Retrieval using Statistical Classification*. Thèse de doctorat, Université de stanford, 1994.

I

(Isakowitz, 1996) Isakowitz T., Kauffman R.J., *Supporting search for reusable software objects*, IEEE Trans. On software engineering, 22(6):407-423, June 1996.

J

(Jausseran, 2005) Jausseran E., *Démarche Symphony étendue, formalisation et expérimentation sur un Système d'Information Hospitalier*, Mémoire d'Ingénieur C.N.A.M, Spécialité Informatique, Grenoble, juillet 2005.

(Jeng, 1993) Jeng J.-J., Cheng B. H. C., *Using formal methods to construct a software component library*, In Ian Sommerville and Manfred Paul, editors, Proceedings of the Fourth European Software Engineering Conference, pages 397-417. Springer-Verlag, 1993.

(Jeng, 1994) Jeng J.-J., B. H. C. Cheng, *Reusing analogous components,* Technical Report MSUCPS -94-28, Michigan State University, Department of Computer Science, A714 Wells Hall, East Lansing, MI 48824-1027, April 1994. http://citeseer.ist.psu.edu/cheng94reusing.htm

(Jeng, 1995) Jeng J.J., Cheng B.H.C., *Specification matching for software reuse: A foundation*, In ACM Symposium on Software Reuse, pages 97-105, seattle, WA, April 1995.

(Johnson, 1998) Johnson M., *A Beginner's guide to Enterprise JavaBeans*, Java World, October 1998.

K

(Karlsson, 1995) Karlsson E.-A., *Software Reuse – A Holistic Approach*. Willey, 1995.

(Khayati, 2001) Khayati O., *Vers des systèmes de patrons formalisés et outillés*, Rapport de DEA Système d'Information, Université Joseph Fourrier, Septembre 2001.

(Khayati, 2003) Khayati O., *Architecture d'un système de gestion de bibliothèques de multiples composants*, Workshop OCM-SI 2003, Nancy, France, 3 juin 2003.

(Khayati, 2003a) Khayati O., Front A., Bardou D., Giraudin J. P., *Formalisation de composants métier Centr'Actoll*, Rapport interne, 10-10-2003.

(Khayati, 2004) Khayati O., Front A., Giraudin J.P., *A metamodel for a metatool to describe and manage components*, IEEE International Conference on Information & Communication Technologies: from Theory to Applications (ICTTA'04), Damscus, Syria 19-23 Avril 2004, pp 403-405.

(Khayati, 2004a) Khayati O., Front A., Giraudin J.P., *Un méta-modèle pour Systèmes d'Accès à des Bases de Composants Hétérogènes*, International conférence CARI'04, Hammamet, Tunisie 22-25 Novembre 2004. pp 255-262.

(Khayati, 2004b) Khayati O., Front A., Bardou D., Giraudin J. P., *Formalisation d'éléments de démarche Centr'Actoll*. Rapport interne, 14-01-2004.

(Khayati, 2005) Khayati O., Front A., Giraudin JP., *Génération et appariement de spécifications formelles de diagrammes de classes pour la recherche de composants*, XXIIIéme Congrès INFORSID'05, Grenoble, France 24-27 Mai 2005.

(Khayati, 2005a) Khayati O., Front A., Giraudin J.P, *Une approche de réingénierie des systèmes d'information basée sur la génération et l'appariement de spécifications de composants*, Revue e-TI, Premier Numéro, 30 octobre 2005, http://www.revue-eti.netdocument.php?id=444.

L

(Labed Jilani, 1997) Labed Jilani L., Mili R., Mili A., *Approximate retrieval: an academic exercise or a practical concern?*, In proceedings, Eight Annual Workshop on Software Reuse, March 1997.

(Larvet, 1994) Larvet Ph., *Analyse des systèmes : de l'approche fonctionnelle à l'approche objet*, InterEditions, 1994.

(Latroue, 1988) Latroue L., Jhonson E., *Seer: A graphical retrieval system for reusable ADA software module*, Proceedings of International IEEE Conference on ADA Application environments, IEEE Computer Society Press, pages. 105-113, 1988.

(Lorenz, 1997) Lorenz, D.H., *Tiling design patterns – a case study*. In ECOOP Proceedings, 1997.

M

(Maarek, 1989) Maarek Y.S., Berry D.M., *The use of lexical affinities in requirements extraction*. In Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pa, May 1989.

(Maarek, 1991) Maarek Y.S., Berry D.M., Kaiser G.E., *An information retrieval approach for automatically constructing software libraries*. IEEE Trans. On Soft. Eng, 17(8):800-813, August 1991.

(Marvie,2001) Marvie R., *CODeX : proposition pour la description dynamique d'architectures à base de composants logiciels*, Dans Acte des Journées Composants, Besancon, Octobre 2001.

(Massonet, 1997) Massonet Ph. Van Lamsweerde A., *Analogical reuse of requirements frameworks*, In Third IEEE International Symposium on Requirements Engineering, Annapolis, MD, January 1997.

(Matena, 1999) Matena V., Hapner M., *Enterprise Java Beans Specification v1.1 – Public Draft*, Sun Microsystems, May 1999.

(Matsumoto, 1987) MatsumotoY., *A software factory: An overall approach to software production*, in "Software Reusability" ed. by P. Freeman, pp.155- 178, IEEE Computer Society, March 1987.

(Meijler, 1997) Meijler T., Nierstrasz O., *Beyond objects : Components*,. Cooperative Information Systems: Current Trends and Directions. Academic Press, 1997.

(Meyer, 1997) Meyer B., *Object-Oriented Software Construction*, Prentice-Hall, second edition, 1997.

(Mili, 1994) Mili R., Mittermeir R., Mili A., *Storing and retrieving software component: A refinement based approach*, In proceedings, International Conference on Software Engineering, Sorento, Italy, May 1994.

(Mili, 1996) Mili R., *Assessing the reuse worthiness of a component: empirical and analytical approaches*, Technical report, University of Ottawa, Ottawa, Ont. Canada, 1996.

(Mishne, 2004) Mishne G. M. de Rijke, *Source Code Retrieval using Conceptual Similarity*, In: Proceedings RIAO 2004, pages 539-554, 2004.

(Moineau, 1991) Moineau Th., Gaudel M.C., *Software reusability through formal specifications*, In Prieto-Diaz R., Schaefer W., Cramer J., editors, Proceedings, First international Workshop on software reusability, number Memo Nr 57 in Unido, Dortmund, 1991.

(Morrison, 1997) Morrison M., *Presenting JavaBeans*, Sams.net Publishing, 1997.

N

(Noble, 1998a) Noble J., *Towards a Pattern Language for Object Oriented Design*, 1998, http://citeseer.nj.nec.com/47369.html.

(Noble, 1998b) Noble, J., *Classifying relationships between object-oriented design patterns*. In Australian Software Engineering Conference (ASWEC), 1998.

O

(OMG, 1994) OMG (Object Management Group), *Business Object Management Special Interest Group – BOMSIG Activities and direction*, 1994.

(OMG, 1999) OMG (Object Management Group), *CORBA/IIOP 2.3 Specification*, OMG TC Document formal/98-12-01, August 1999.

(OMG, 2003) OMG (Object Management Group), *Unified Modeling Language (UML) Specification 1.5*, march 2003.

(OMG, 2005) OMG (Object Management Group), *Model Driven Architecture (MDA)*, OMG document ormsc/05-04-01, 2005

(OMG, 2005a) OMG (Object Management Group), *UML 2.0 superstructure final adopted specification*, 4 July 2005.

(Orfali, 1998) Orfali R., Harkey D., *Client/Server Programming with Java and CORBA, 2nd Edition*, John Wiley & Sons, March 1998.

(Oussalah, 1997) Oussalah M. (coordination). *Ingénierie Objet - concepts et techniques*, InterEditions, mai 1997.

(Oussalah, 1999) Oussalah M. (coordination). *Génie Objet, Analyse & Conception de l'Evolution d'Objets*, Editions HERMES, 1999.

(Oussalah, 2005) Oussalah M. (coordination). *Ingénierie des Composants Logiciels : principes et fondements*, Editions Vuibert, juin 2005.

P

(Park, 1997) Park Y., Bais P., *Generating samples for component retrieval by execution*, Technical report, University of Windsor, Windsor, Ontario, Canada, 1997.

(Paul, 1994) Paul S., Prakash A., *Querying Source Code using an Algebraic Query Language*, International Conference on Software Maintenance, IEEE Press, pp. 127-136, 19-23 September 1994.

(Penix, 1995) Penix J., Alexander P., *Design representation for automating software component reuse*, In Proceedings, Fifth International Workshop on Knowledge Based System for the (Re)Use of Software libraries, November 1995.

(Penix, 1996) Penix J., Alexander P., *Efficient specification based component retrieval*, Technical report, University of Cincinnati, Knowledge Based Software Engineering Laboratory, ECECS, July 1996.

(Pernici, 2000) Pernici B., Mecella M., Batini C., *Conceptual Modeling and Software Components Reuse: Towards the Unification*. In Solvberg A., Brinkkemper S., Lindencrona E. (eds.): Information Systems Engineering: State of the Art and Research Themes. Springer Verlag, 2000.

(Pitt, 2001) Pitt E., McNiff K., *Java RMI: The Remote Method Invocation Guide*, Addison Wesley, 2001.

(Podgursky, 1992) Podgursky A., Pierce L., *Behaviour sampling : a technique for automated retrieval of reusable components*, In proceedings, 14th International conference on software engineering, pages 300-3004, new york, NY:ACM Press, 1992.

(Podgursky, 1993) Podgursky A., Pierce L., *Retrieving reusable software by sampling behavior*, acm Trans on Software Engineering and Methodology, 2(3):286-303, July 1993.

(Poulin, 1995) Poulin J., Werkman K.J., *Modeling structured abstracts and the world wide webfor retrieval of reusable components*. In Proceedings, Symposium on software Reuse, Seattle, Washington, April 1995.

(Prieto-Diaz, 1985) Prieto-Diaz R., *A software classification scheme*, Technical report, University of California, Irvine, CA, 1985.

(Prieto-Diaz, 1987) Prieto-Diaz R., Freeman P., Classifying Software for Reusability. IEEE Software, 4(1), pages 6-16, 1987.

(Prieto-Diaz, 1991) Prieto-Diaz R., *Implementing faceted classification for software reuse*, Communications of the acm, 34(5):88-97, May 1991.

(Proactive, 2005) Proactive, *ProActive manual version 2.2*, 2005, http://www-sop.inria.fr/oasis/ProActive/.

R

(Rijsbergen, 1979) Rijsbergen C.J., *Information Retrieval*, Butterworth and Co., London, 2nd edition, 1979.

(Ritti, 1989) Ritti M., *Using types as search keys in function libraries*, In Proceedings of the fourth international conference on Functional programming languages and computer architecture, Imperial College, London, United Kingdom, pp 174 – 183, 1989, ISBN:0-89791-328-0

(Ritti, 1992) Ritti M., *Retrieving library identifiers via equational matching of types*, Technical Report 65, Programming Methodology Group, Dept of Computer Science, Chalmers University of Technology and University of Goteborg, Goteborg, Sweden, 1992.

(Rogerson, 1997) Rogerson D., *Inside COM*, Microsoft Programming Series, Microsoft Press, 1997.

(Rollins, 1991) Rollins E.J., Wing J.« specifications as search keys for software libraries », in Proceedings of the Eighth International Conference on Logic Programming, pp. 173--187, 1991.

http://citeseer.ist.psu.edu/434412.html

(Roman, 2002) Roman E., Ambler S., et Jewell T., *Mastering Enterprise JavaBeans*, Wiley Computer Publishing, second edition, 2002.

(Runciman, 1989) Runciman C., Toyn I., *Retrieving reusable software components by polymorphic type*, In proceedings, Conference on Functional Programming Languages and Computer Architectures, Reading, Ma: Addison Wesley, 1989.

S

(Steigerwald, 1992) Steigerwald R.A., *Reusable component retrieval with formal specifications*, In proceedings of the 5th Annual Workshop on Software Reuse, Palo Alto, CA, October 1992.

(Sutherland, 1997) Sutherland J., Patel D., Casanave C., Miller J., Hollowell G. *The Object Technology Architecture: Business Objects for Corporate Information Systems. Business Object Design and Implementation*, OOPSLA'97 Workshop Proceedings, 1997.

(Szyperski, 1998) Szyperski C., *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

T

(Tastet, 2004) Tastet L., *AGAP : un Atelier de Gestion et d'Applications de Patrons*, Mémoire d'Ingénieur C.N.A.M, Spécialité Informatique, Grenoble, mars 2004.

V

(Villalobos, 2003) Villalobos J., *Fédération de composants : une architecture logicielle pour la composition par coordination*. Thèse en Informatique à l'Université Joseph Fourier (Grenoble), juillet 2003.

W

(Wartik, 1992) Wartik S., Prieto-Diaz R., *Criteria for Comparing Reuse-Oriented Domain Analysis Approaches*, International Journal of Software Engineering and Knowledge Engineering, 2(3), pp. 403-431, 1992.

(Wirfs-Brock, 1990) Wirfs-Brock R., Wilkerson B., Weiner L., *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

(Wirsing, 1991) Wirsing M., *Algebraic specification*, In Handbook of Theoretical Computer Science, Vol. B. North-Holland, Amsterdam, 677-788, 1991.

Z

(Zaremski, 1993) Zaremski A.M., Wing J.M., *Signature matching: A key to reuse*, In Proceedings, SIGSOFT'93: ACM SIGSOFT Symposium on the Foundations of Software Engineering, Redondo Beach, CA, December 1993.

(Zaremski, 1995) Zaremski A.M., Wing J.M., *Signature matching: A tool for using software libraries*, ACM Transactions on Software Engineering and Methodology, 4(2):146-170, April 1995.

(Zaremski, 1995a) Zaremski A.M., Wing J.M., *Specification matching of software components*, In Proceedings, SIGSOFT'95: third ACM SIGSOFT Symposium on the Foundations of Software engineering, New York, NY: ACM Press, October 1995.

(Zhang, 2000) Zhang Z., Svensson L., Snis U., Srensen C., Fgerlind H., Lindroth T., Magnusson M., Stlund C., *Enhancing Component Reuse Using Search Techniques*, Proceedings of IRIS 23. Laboratorium for Interaction Technology, University of Trollhttan Uddevalla, 2000.

(Zimmer, 1994) Zimmer, W., *Relationships between design patterns In Pattern Languages of Program Design*. Addison-Wesley, 1994.

Résumé : Face à l'émergence de catalogues de composants de différents types, les environnements professionnels de développement d'applications ont évolué vers une gestion des composants. Ces premiers outils offrent peu de services pour faciliter une recherche appropriée de composants de différentes natures ; ils sont destinés à des experts ayant déjà une connaissance profonde des catalogues de composants. Ces outils restent insuffisants pour des ingénieurs d'applications non experts dont l'objectif est de rechercher les meilleurs composants répondant au problème posé.
En partant de ce constat, l'objectif de cette thèse est de s'appuyer sur les techniques de recherche d'information pour d'une part proposer une nouvelle technique de recherche de composants adaptée aux catalogues hétérogènes de composants et d'autre part organiser un environnement d'aide à l'application, la gestion, la recherche et la réutilisation de composants. La gestion des composants est réalisée au niveau d'une base de description de propriétés de composants. Pour des besoins d'évolution et d'adaptation, une telle base est mise en place par des instanciations successives du métamodèle *M-Sigma* et du modèle de composants *C-Sigma*.
La technique structurelle originale de recherche de composants proposée est basée sur un processus de génération de spécifications formelles à partir de diagrammes de classes décrivant la structure des composants archivés ou recherchés. Ces spécifications sont utilisées pour réaliser des appariements structurels entre composants. Cette technique est exploitable dans le domaine de la recherche de composants et de la réingénierie des systèmes d'information. Une expérimentation a été faite sur une base de composants issue du projet industriel Initiative Centr'ACTOLL.

MOTS-CLÉS : réutilisation, recherche de composants, bases de composants, modèle de composants, métamodèle.

Abstract : Facing the emergence of components catalogs of various types, the professional applications development environments evolved to integrate components management. These first tools offer few services to facilitate retrieval and selection of components of various natures. These services are intended for experts already having high knowledge of the components catalogs, but are not suited to nonexpert applications engineers whose goal is to find the best components matching their needs.
From this statement, this thesis aims first to study existing information and components retrieval techniques, and propose a new technique suited to components heterogeneous catalogues, and secondly to build a development environment for assistance to components management, retrieval, selection and re-use. Components management is carried out thanks to a components description repository built by successive instanciations of the *M-Sigma* metamodele and the *C-Sigma* model.
The proposed structural retrieval technique is based on an automatic generation process of formal specifications starting from UML classes diagrams describing the components signature and structure. These specifications are used to make structural matching between components and user queries. This technique can be used for components retrieval and information systems re-ingineering. An experimentation was made on components repository resulting from the industrial project Initiative Centr' ACTOLL.

KEYWORDS: reuse, components retrieval, components repository, components model, metamodel.