THÈSE

pour obtenir le grade de

Docteur de l'Université de Savoie

UNIVERSITÉ DE SAVOIE

Spécialité : *Informatique*

Azadeh RAZAVIZADEH

BeeEye : approche et cadre de travail pour la construction des vues architecturales basées sur les points de vue

Soutenue le 25 octobre 2010 devant le jury composé de :

Mme Marianne Huchard	Rapporteur
M. Salah Sadou	Rapporteur
M. Patrice Moreaux	Examinateur
M. Stéphane Ducasse	Directeur de thèse
M. Hervé Verjus	Co–Directeur de thèse
Mme Sorana Cîmpan	Co–Directeur de thèse

Thèse préparée au sein du Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance

THÈSE

pour obtenir le grade de

Docteur de l'Université de Savoie

UNIVERSITÉ DE SAVOIE

Spécialité : *Informatique*

Azadeh RAZAVIZADEH

BeeEye : approche et cadre de travail pour la construction des vues architecturales basées sur les points de vue

Soutenue le 25 octobre 2010 devant le jury composé de :

Mme Marianne Huchard	Rapporteur
M. Salah Sadou	Rapporteur
M. Patrice Moreaux	Examinateur
M. Stéphane Ducasse	Directeur de thèse
M. Hervé Verjus	Co–Directeur de thèse
Mme Sorana Cîmpan	Co–Directeur de thèse

Thèse préparée au sein du Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance

A mon cher père ...

Remerciements

J'adresse mes plus vifs remerciements à Mme Marianne HUCHARD Professeur à l'Université Montpellier II et M. Salah SADOU Maitre de Conférences à l'Université de Bretagne Sud pour m'avoir fait l'honneur d'étudier mes travaux de thèse et pour les avoir cautionnés en qualité de rapporteurs.

Je tiens à remercier M. Patrice MOREAUX Professeur à l'Université de Savoie qui m'a fait l'honneur de présider le jury et d'avoir bien voulu examiner ces travaux.

Les travaux présentés dans cette thèse ont été effectués au sein du Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance (LISTIC, Annecy). Je tiens à remercier Monsieur Philippe BOLON, directeur du LISTIC pour m'avoir accueilli dans son laboratoire.

Je tiens à exprimer toute ma gratitude à M. Stéphane DUCASSE, Directeur de Recherches à l'INRIA Lille - Nord Europe de m'avoir donné la possibilité d'effectuer cette thèse. Je tiens aussi à le remercier pour ses conseils et son aide durant cette thèse, et pour les soins qu'il a bien voulu apporter à la lecture de ce mémoire.

Je voudrais adresser mes remerciements à M. Hervé VERJUS et Mme Sorana Cîmpan, Maîtres de Conférences à l'Université de Savoie, qui m'ont codirigé jusque' à l'achèvement de mon Doctorat. Leurs compétences, leurs conseils et leurs écoutes m'ont aidé tout au long de la thèse.

Je remercie également tous les membres de LISTIC pour leur accueil, leurs encouragements et leur sympathie. Une grande merci à mes amis et collèges Lavinia, Sylvie, Laurent, Yajing, Nacima et Dana pour leurs soutiens et leurs amitiés. Merci à Lionel pour les encouragements qui ont été essentiels pour l'aboutissement de ces travaux.

Enfin, je remercie Amin pour son soutien et son aide inconditionnels et sa présence inestimable à mes coté durant toute la durée de cette thèse. Je remercie de tout mon coeur mes parents, ma soeur et mon frère qui sont si loin de moi et si proche à la fois, par cette chaleur et cette force qu'ils me transmettent à chaque appel.

Résumé

Résumé : Une grande partie de l'effort de développement des logiciels de grande taille et de longue durée de vie est consacrée à la maintenance et à l'évolution de ces derniers ; et la compréhension de ces logiciels est une nécessité pour laquelle trop peu de propositions ont été faites. Le but de cette thèse est la prise en compte des architectures logicielles comme élément capital pour l'aide à compréhension et l'évolution des applications logicielles à objets. Les travaux de recherche des dix dernières années ont montré qu'il était très difficile voire impossible de raisonner sur le code source (niveau concret), tant les détails d'implémentation nuisent à la lisibilité du code, à sa maintenance/évolution. Au contraire, se situer à un plus haut niveau d'abstraction permet d'envisager davantage d'analyses. Nous proposons, dans cette thèse, notre approche BeeEye comme démarche d'ingénierie pour la construction de vues architecturales d'un système logiciel à objets existant. BeeEye utilise et permet à l'utilisateur de définir des points de vue qui peuvent être combinés selon un processus de construction de vues. Les points de vue permettent de modéliser les attentes et souhaits des utilisateurs ou/et la connaissance qu'ils ont (ou l'idée qu'ils se font) a priori du système logiciel étudié. Deux catégories de construction de vues architecturales sont proposées dans le cadre de la thèse : la construction par correspondance et la construction par exploration. L'approche proposée est suffisamment générique pour être utilisée avec tous les systèmes logiciels à objets pour lesquels nous disposons du code source.

Mots-clé : Evolution, compréhension, reconstruction, vue architecturale, point de vue.

Abstract : Changes and evolution of software systems constantly generate new challenges for system understanding. Recovering system architectural representations is particularly interesting when such representations are not available. The work presented in this thesis joins the effort on software architecture reconstruction. This thesis proposes approach BeeEye as a generic engineering approach to enable the construction of architectural views from an existing object-oriented system. BeeEye uses viewpoints and lets users define new ones. BeeEye provides different construction processes ; gives the possibility to define user-specific construction processes. The viewpoints are used to model the expectations and wishes (or priory knowledge) of users about the software system in question. Two categories of architectural views construction are proposed : construction by mapping and construction by exploration. The proposed approach is generic enough to be used with all objects software systems for which we have the source code.

Table des matières

Chapitre 1

Introduction

1.1 Problématique et objectifs de la thèse

Les logiciels sont condamnés à évoluer et à se complexifier continuellement pour satisfaire de nouveaux besoins. Ainsi, la maintenance et l'évolution sont devenues deux activités centrales et coûteuses du cycle de vie des logiciels à longue durée de vie et de grande taille [McK84]. La rétro ingénierie des systèmes logiciels est un problème important dans ce contexte. La rétro ingénierie est définie par Chikosfky et Cross comme *"le processus d'analyse d'un système afin d'identifier les composants du système et les relations entre ces composants, et de créer des représentations du système sous une autre forme ou à un niveau d'abstraction supérieur"* [CCI90].

La compréhension des systèmes logiciels est une condition préalable à la maintenance et à l'évolution de ces systèmes. Cependant, cette compréhension est difficile et coûteuse en terme de temps et d'efforts. Des études montrent que 50-60 % de l'effort en génie logiciel est passé à essayer de comprendre le code source [Bas97]. La tâche de compréhension lors des phases de maintenance et d'évolution est compliquée [Par94] :

- souvent les développeurs en place lors de la phase initiale du logiciel ne sont plus disponibles, et des développeurs inexpérimentés (par méconnaissance du domaine ou du système) sont affectés à maintenir et faire évoluer les systèmes ;
- les méthodes de développement et les langages de programmation qui ont été utilisés pour développer le système d'origine sont obsolètes, et il est difficile de trouver des gens connaissant ces techniques ou qui ont la volonté de les apprendre ;
- des modifications importantes ont été appliquées au cours du temps au système, ce qui conduit souvent à un décalage entre la documentation disponible et l'implémentation du système à cause des écarts entre l'architecture prévue et son implémentation. Ce manque de cohérence entre la documentation et l'implémentation rend difficile la compréhension du système ;
- la documentation est incomplète, non mise à jour ou complètement absente.

Face à la croissance de la taille et de la complexité des systèmes logiciels, les architectures logicielles s'imposent comme une solution possible, tout du moins une aide à la compréhension [Gar00]. La notion de vues abstraites des systèmes étudiés est devenue, pendant la dernière décennie, un sous-domaine central du génie logiciel [Gar00]. Les recherches ont démontré les avantages et l'étendue des utilisations possibles de ces architectures logicielles [SG96]. Ces avantages sont bénéfiques dans le cadre de la conception, de la maintenance, ainsi que dans la facilitation de la compréhension.

Depuis que les systèmes ont tendance à être de grande taille (plusieurs millions de lignes de code mal documentées), il existe un réel besoin d'avoir des approches efficaces qui aident à comprendre ces

systèmes. Ainsi, de nombreuses approches sont apparues dans le domaine de la rétro-ingénierie pour pallier aux problèmes exposés. Ce domaine vise à identifier les structures d'un système logiciel et à fournir une représentation du logiciel permettant d'avoir une meilleure vue du système. Ce domaine inclut l'extraction d'architecture, la re-documentation et l'exploration du code.

Une des notions importantes associée à l'architecture logicielle est celle de vue architecturale [CBB+02]. Une architecture logicielle est une entité complexe qui ne peut être décrite d'une façon simple et en une seule dimension. Il peut être déconcertant de constater qu'aucune vue unique ne peut représenter une architecture d'une façon complète. Mais l'essence de l'architecture est la suppression des informations non nécessaires, et de présenter une ou deux facettes du système à la fois. Chaque vue met l'accent sur certains aspects du système tout en ignorant d'autres aspects de ce système. Toutefois, une question se pose : quelles sont les vues appropriées ? Des vues différentes exposent des éléments différents à des degrés divers. Il apparaît donc essentiel de ne pas préconiser une vue particulière ou une collection des vues mais d'offrir un support à construire des vues multiples. L'ensemble des travaux sur l'extraction des vues architecturales a abouti à une amélioration et une meilleure maîtrise de l'architecture d'un logiciel. Mais les recherches tentent d'améliorer les techniques et de proposer des outils plus adaptés afin de répondre à certains problèmes : quelles sont les vues les plus adéquates qui doivent être construites ? Quels genres de techniques sont appropriées pour construire, reconstruire ou combiner ces vues ? Comment considérer les attentes et souhaits des utilisateurs dans des processus de construction des vues architecturales (en faisant l'hypothèse que ces attentes ne soient pas prédéfinies) ? Autrement dit, comment est-il possible de rendre génériques les processus d'extraction de vues architecturales ?

Partant de ces considérations, notre objectif, dans cette thèse est de proposer une approche générique pour construire des vues architecturales selon différents points de vue. L'idée de cette approche est de permettre une amélioration générale de la compréhension d'un système logiciel existant dans les phases de maintenance et d'évolution. Divers travaux ont porté sur la récupération d'une seule vue architecturale ou de quelques vues prédéterminées. La plupart des approches existantes diffèrent de notre proposition au sens où elles n'abordent que des algorithmes concrets, ou/et qu'un ensemble déterminé de vues (re)construites selon une bibliothèque figée de points de vue. Notre approche fournit un modèle générique de construction de vues architecturales sans que ces dernières soient prédéfinies. C'est une approche basée sur des points de vue afin d'assurer la prise en compte des diverses vues architecturales. Elle est basée non seulement dans la construction de l'architecture, mais, de manière plus fondamentale dans la définition des activités de reconstruction : là où les processus, les techniques et les vues peuvent être combinées de différentes manières en fonction des attentes des différents utilisateurs concernés par la compréhension du système (que nous appellerons les utilisateurs dans le suite du document).

1.2 Plan du mémoire

Cette thèse contient six chapitres. Les deux premiers chapitres présentent le contexte de notre travail et une synthèse des travaux existants. Les trois chapitres suivants décrivent notre proposition, et une mise en pratique de notre approche. Enfin, le dernier chapitre du manuscrit inclut un bilan de notre approche et indique un certain nombre de perspectives.

- **Architectures logicielles : quelques définitions et problématiques**

  Ce chapitre pose la terminologie associée aux architectures logicielles et présente différentes définitions existantes dans ce domaine de recherche. En première partie, différentes définitions

sont présentées selon la notion de temps. L'objectif est de présenter ces terminologies communes utilisées dans le domaine et celles que nous utiliserons dans le cadre de notre approche. Ensuite nous allons présenter les avantages apportés par l'architecture logicielle. Ainsi nous allons présenter le concept de vue architecturale et différents travaux effectués sur les vues.

- **Etat de l'art : la reconstruction de l'architecture logicielle**

  Ce chapitre introduit la problématique liée à la reconstruction de l'architecture logicielle lors des phases de maintenance et d'évolution. L'objectif de cette section est de montrer l'importance de prendre en compte l'architecture durant ces phases. Nous montrerons que la compréhension architecturale du logiciel est une partie indispensable du cycle de vie d'un système logiciel. Ensuite nous présentons deux modèles conceptuels existants dans la littérature pour l'extraction de l'architecture. A ce titre, notre modèle conceptuel sera également présenté afin d'être utilisé dans la reconstruction des vues architecturales. Nous classerons ensuite les approches et les travaux existants sur le thème de la reconstruction de l'architecture. Cette étude sera effectuée selon différents critères importants à prendre en compte dans la reconstruction de l'architecture. Ce chapitre se termine par une synthèse de cette étude.

- **La construction des vues par BeeEye**

  Ce chapitre présente les motivations et les principes qui ont guidé notre approche de construction de vues en s'appuyant sur l'état de l'art. Nous présenterons notre proposition *BeeEye*, une approche et un framework génériques basés sur des points de vue afin de construire des vues architecturales. Pour décrire l'approche *BeeEye*, nous introduirons d'abord les principes généraux. Ensuite le framework *BeeEye* sera introduit, ainsi que les principales caractéristiques de ce framework pour l'ingénierie de construction de vues architecturales au travers duquel nous allons préciser comment ces caractéristiques sont concrétisées. *BeeEye* considère deux catégories de points de vue (*Correspondance* et *Découverte*) pour la construction des vues architecturales. Différentes techniques de construction peuvent ainsi être considérées et supportées par notre approche. Nous montrerons comment notre approche permet de combiner des processus de construction et des vues architecturales construites. Nous présenterons ensuite le méta modèle du framework *BeeEye* supportant l'approche *BeeEye* et les mécanismes de construction de vues architecturales. Enfin, nous proposerons une étude possible sur les vues utilisées et/ou construites par l'approche.

- **Les primitives définies pour la construction des vues**

  Les points de vue proposés par notre approche incluent une partie algorithmique qui permet de construire des vues architecturales. Ces algorithmes de construction de vues sont proposés en fonction de chaque catégorie de points de vue introduite lors du chapitre précédent. L'approche *BeeEye* propose différentes techniques de construction telles que par *abstraction*, par *raffinement* et par *composition*. La suite du chapitre est organisée de la façon suivante : après avoir rappelé les principales notions de l'approche, nous présenterons les *primitives* de construction de vues architecturales. L'ensemble des primitives est présenté dans deux sous-parties : les primitives qui dépendent de la catégorie de points de vue ciblée (à savoir points de vue par correspondance, points de vue découverte) et les primitives qui dépendent de ou des techniques de construction de vue ciblées (abstraction, raffinement, composition) [ASHS09].

- **Mise en pratique et validation de BeeEye**

Ce chapitre propose une mise en pratique de l'approche *BeeEye*. Pour cela, différents points de vue seront présentés pour illustrer notre approche afin de réaliser la construction des vues architecturales à partir d'un système logiciel orienté objet.

– **Conclusions et Perspectives**

Cette partie fera un bilan du travail réalisé dans le cadre de la thèse et dressera quelques perspectives.

Première partie

Etat de l'art

Chapitre 2

Architectures logicielles

Les systèmes logiciels connaissent une croissance en terme de taille et de complexité. Dans ce cadre, les architectures logicielles sont une solution performante permettant d'améliorer la conception et la maintenance de ces systèmes logiciels complexes. En effet, en offrant une vue abstraite d'un système, cette représentation de haut niveau permet de gérer cette croissance. A ce titre, les architectures sont devenues un sous-domaine central du génie logiciel [Gar00] sur lequel de nombreux travaux de recherche ont porté ces dernières années.

Préalablement à la description des travaux réalisés dans le domaine de l'architecture logicielle au cours de cette thèse, il est nécessaire de présenter ce que nous entendons par le terme *architecture logicielle*. Afin de pouvoir répondre à la question "qu'est-ce que l'architecture d'un système logiciel ?", il faut bien creuser ce domaine. Bien que des avancées aient été réalisées dans ce domaine, la communauté scientifique a fourni de nombreuses propositions et définitions intéressantes pour l'étude de l'architecture logicielle. Nous présentons dans ce chapitre la notion d'architecture logicielle ainsi que le paradigme souvent utilisé dans le domaine des architectures logicielles. Nous discuterons des définitions proposées dans la littérature portant sur les architectures logicielles ainsi que sur les principaux utilisateurs du domaine. Sur la base des travaux évoqués, nous proposerons une synthèse couvrant la définition des concepts que nous utiliserons dans le cadre de cette thèse.

2.1 L'architecture logicielle : concept et terminologie

Plusieurs définitions de l'architecture logicielle sont proposées mais aucune d'entre elles ne s'impose vraiment. Cependant, les chercheurs sont d'accord pour dire qu'une architecture logicielle décrit les structures de haut niveau d'un système logiciel. Nous allons maintenant parcourir les principales références de ce domaine. Pour mettre en évidence l'évolution des propositions, nous présentons ces références dans leur ordre chronologique de publication.

Perry et Wolf 1992 [PW92]

Historiquement, la définition proposée par Perry et Wolf [PW92] est importante car l'article dans lequel elle a été donnée est considéré comme l'article fondateur de l'architecture logicielle en tant que domaine à part entière du génie logiciel. Perry et Wolf se sont basés sur le domaine de l'architecture des bâtiments (en génie civil) pour définir leur vision. Ils ont proposé une architecture logicielle comme l'ensemble d'éléments architecturaux qui ont une forme particulière.

Ils proposent un modèle pour l'architecture logicielle défini comme suit :

$$"SoftwareArchitecture = \{Element,\ Form,\ Rationale\}" \tag{2.1}$$

Dans leur définition, nous avons trois notions sur lesquelles l'architecture est basée :

Premièrement, les **éléments** architecturaux peuvent être de trois natures :

- les éléments de données : ces éléments contiennent de l'information utilisée et transformée ;
- les éléments de transformation : ces éléments sont des composants qui fournissent la transformation sur des éléments de données ;
- les éléments de connexion : ces éléments peuvent être les éléments de données, les éléments de transformation de données et les éléments de connexion entre les éléments précédents. Par exemple, ce sont les appels procéduraux, les données partagées ou les messages qui servent à relier ensemble les éléments architecturaux ;

Deuxièmement, la **forme** architecturale correspond à des propriétés qui définissent des contraintes sur des éléments, mais également à des relations qui définissent des contraintes sur la disposition des éléments entre eux.

Enfin, les **motivations** (**rationale**) capturent les raisons et des objectifs pour choisir une architecture plutôt qu'une autre. Ces motivations peuvent être de différentes natures liées à la conception, à la technique *etc.*

Un autre concept proposé par leur définition est le concept de vues architecturales, celles-ci permettent de représenter différents aspects d'une même architecture. Ces vues sont représentées de manière séparée mais elles sont interdépendantes. Cette définition propose trois vues : la vue des données, la vue de transformation et la vue de connexion. Ces trois vues correspondent aux trois types d'éléments proposés dans la définition de l'architecture.

Malgré le fait que cet article soit le premier à avoir été proposé traitant de la notion d'architecture logicielle, cet article fait toujours référence car il traite des fondements de la discipline. Nous adhérons au concept de vues multiples d'une même architecture. Ce concept a l'avantage de pouvoir traiter différents besoins pouvant être associés à différents utilisateurs. Nous reprendrons cette idée un peu plus loin en montrant qu'en plus de la notion de vues multiples, il existe la notion de points de vue multiples. En effet ces vues multiples peuvent être structurées par des points de vue.

Ainsi comme les auteurs, nous pensons que les descriptions architecturales n'ont pas pour unique objectif de constituer de la documentation, mais qu'il est intéressant de les exploiter pour des phases d'évolution, de maintenance et d'analyse du système étudié.

Garlan et Perry 1995 [GP95]

Cette définition a été proposée dans le premier numéro spécial du journal ("IEEE Transactions on Software Engineering") paru sur le domaine de l'architecture logicielle. Cette définition fournissait à l'époque une bonne synthèse des différents travaux d'alors et constituait ainsi un bon point de départ pour plusieurs approches dans ce domaine.

La définition proposée par Garlan et Perry est basée sur *la structure des composants d'un système, des relations entre ces composants, et des principes qui dirigent leur conception et leur évolution au fil du temps.* Les auteurs se sont focalisés sur plusieurs points dans cette définition. Certains de ces points sont présentés par la suite :

- cette définition est axée sur l'organisation des structures de haut niveau du système. Pendant les

phases d'évolution et de maintenance d'un système, de nouveaux intérêts/besoins apparaissent. Ils sont/deviennent souvent plus importants que le choix d'une structure de données ou d'un algorithme. L'architecture logicielle tente de fournir un cadre formel, ainsi que des notations et des outils pour l'étude de l'organisation des structures de haut niveau d'un système ;
- les méthodes de conception fournissent des liens clairs entre certains besoins et leur implémentation. Ainsi, l'architecture logicielle permet de gagner un degré de liberté en s'abstrayant au mieux de l'implémentation. Les auteurs ont focalisé leur présentation sur le fait que l'architecture logicielle ne se place pas au même niveau que l'algorithmique et les structures de données. Par contre, les méthodes de conception, tout comme l'architecture logicielle, essaient de cloisonner les besoins dans l'implémentation.

Bass, Clements et Kazman 1998 [BCK98]

Ce livre est consacré à l'architecture logicielle. La définition donnée comprend de nombreuses idées :

L'architecture logicielle d'un programme ou d'un système est la structure ou les structures de ce système, comprenant les composants, les propriétés visibles extérieurement de ces composants et les relations entre eux. "The software architecture of a program or computing system is the structure or structures which comprise software components, the externally visible properties of those components, and the relationships among them".

Nous allons détailler certains points présentés dans cette définition :

- La définition indique que l'architecture logicielle n'est qu'une abstraction de la réalité. Ce qui nous semble pertinent, sinon il faudrait reprendre entièrement le système.
- La définition insiste sur l'idée qu'un système peut intègrer plusieurs structures. De la même manière il existe plusieurs types d'éléments et plusieurs types d'interactions entre les éléments. Nous indiquons qu'une même structure peut avoir des vues multiples. Les vues permettent de montrer une structure à partir d'angles (d'observation) différents. Cependant, la plupart des approches portent sur la structure fonctionnelle car d'autres structures n'ont du sens qu'à partir d'une certaine taille des applications.
- Les auteurs révèlent deux points importants sur l'architecture :

  - Le premier point est la différence qui existe entre l'architecture concrète d'un système et une description architecturale d'un système. L'architecture concrète correspond à l'architecture définie dans le code source, ce qui correspond à ce qui est communément appelé la "réalité". Une description architecturale est une abstraction du code source qui tente de décrire au mieux cette architecture concrète.
  - Le second point est le fait qu'une description architecturale peut ne plus être en conformité avec l'architecture concrète qu'elle décrit. Alors que dans un contexte idéal, la description architecturale doit toujours être cohérente avec le code source.

La norme ANSI/IEEE 14-71-2000 [IEE00]

La norme IEEE 1471 a pour objectif de fournir des recommandations pour la description architecturale des systèmes logiciels en se basant sur un standard. *L'architecture logicielle, est l'organisation*

fondamentale d'un système inscrite dans ses éléments, dans les relations entre eux, ainsi que dans l'environnement et les principes qui guident leur conception et leur évolution.

Cette définition couvre une variété d'aspects dans l'utilisation du terme *architecture*. Elle met clairement en évidence une architecture qui incarne les aspects fondamentaux d'un système. Cette définition vise à compléter les autres définitions : un système est lié à un environnement qui l'influence. Un système doit respecter un certain nombre d'objectifs en fonction de son environnement. Cette définition montre que l'architecture peut être représentée par une ou plusieurs descriptions architecturales.

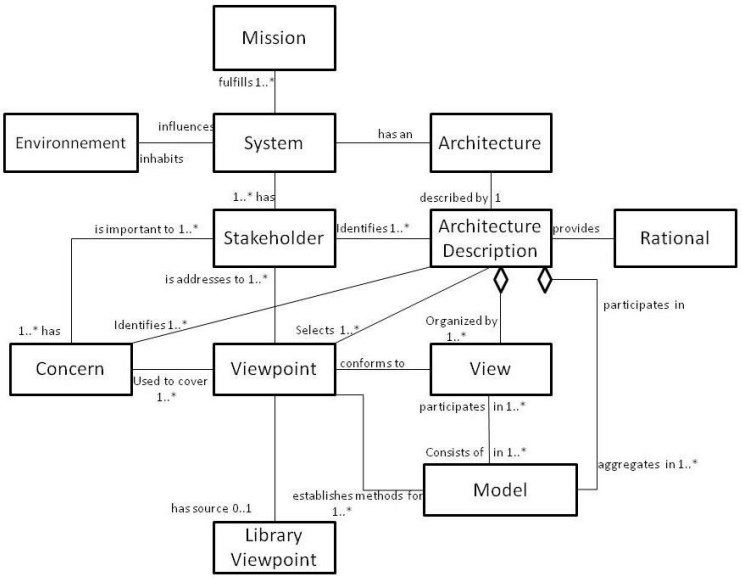

FIGURE 2.1 – Modèle conceptuel pour une description architecturale

Un autre point de cette définition, présenté dans la figure 2.1, est le fait que les différents utilisateurs sont impliqués et concernés par la description. Chacun de ces utilisateurs peut avoir des intérêts différents par rapport au système. La description architecturale peut être présentée par plusieurs vues qui correspondent à différents modèles. Chaque modèle est lié à l'environnement du système à partir d'un angle différent.

Dans la norme 1471, chaque vue doit être conforme à un point de vue. Chaque point de vue est défini pour couvrir des intérêts correspondant à un certain nombre d'utilisateurs. De bons exemples de point de vue sont fournis par les travaux de Kruchten [Kru95] et Hofmeister et al.[HNS00]. Nous allons présenter leur travaux dans les sections 2.3.1 et 2.3.2. Dans notre approche, nous nous sommes également focalisés sur les points de vue en tant que guides de construction des vues.

La définition proposée par la norme ANSI/IEEE 14-71 ne s'intéresse pas à la cohérence entre les différentes vues. La définition ne propose aucune manière de faire les combinaisons entre des vues ou des points de vue.

Nous avons pu constater que notre travail rentre bien dans le cadre de cette définition. Nous considérons cette définition comme une référence importante pour notre travail.

En revenant sur la définition de l'architecture logicielle donnée par IEEE 1471, nous détaillons la notion d'*utilisateur* (*stakeholder*) et de *vue* dans les sections suivantes.

2.2 Les utilisateurs

Les personnes concernées par un système logiciel ne sont pas forcément les utilisateurs finaux. Ce système doit être construit, examiné, exploité, réparé, ou souvent amélioré pour/par ces personnes. Ces personnes sont regroupées par le terme "stakeholders" dans la littérature :

Un **stakeholder** d'un système est un utilisateur, un groupe ou une organisation ayant des intérêts ou des préoccupations par rapport au système étudié [IEE00].

Nous utilisons le terme *utilisateur* pour les *stakeholder*.

Certains utilisateurs peuvent être :

- **un architecte logiciel** : un utilisateur qui a le rôle d'architecte d'un système logiciel doit avoir des compétences spécifiques. Par exemple, il doit pouvoir :
  - comprendre le contexte du système ;
  - identifier les objectifs des différentes parties de l'architecture ;
  - documenter des demandes fonctionnelles et non fonctionnelles du système ;
  - définir des concepts architecturaux et logiques architecturales ;
- **une personne effectuant la tâche de développement** : l'architecture envoie des ordres et fixe des contraintes sur la manière dont cette personne peut implémenter des nouvelles parties du système. Un développeur peut se retrouver à travailler sur des parties du système qu'il n'a pas développées. L'architecte doit donc pouvoir répondre à certaines de ces questions dans cette situation.
- **une personne effectuant la tâche de maintenance** : elle utilise l'architecture comme un point de départ pour les activités d'entretien. Ces personnes voudront voir les mêmes informations que les développeurs, car les deux doivent faire des modifications, en ayant souvent les mêmes contraintes. Par contre les personnes dédiées à la maintenance veulent également avoir à leur disposition une vue présentant la décomposition du système : ceci leur permet de repérer les endroits où réaliser d'éventuels changements. Les informations nécessaires des personnes dédiées à la maintenance se concentrent sur des questions telles que la documentation existante, l'instrumentation, le débogage du système, le contrôle des changements de production, la préservation des connaissances au fil du temps, *etc.*

L'utilisation de l'architecture comporte des avantages pour ces utilisateurs dans le cycle de vie d'un système logiciel. Cependant, toutes les exigences demandées par les utilisateurs n'ont pas le même degré d'importance. Dans la plupart des cas, seule une partie (un sous ensemble) des exigences est présente dans une architecture. Dans la phase de maintenance d'un système, l'architecture permet de comprendre la logique existant derrière les décisions conceptuelles prises pour ce système. Détecter suffisamment tôt les endroits à modifier et à traiter est une raison de plus justifiant l'intérêt de l'architecture pour les personnes occupées aux tâches de maintenance et d'évolution.

2.3 Vues architecturales

Plusieurs travaux ont mis en évidence la nécessité de prendre en compte les vues architecturales [MAvdLF00, CBB+02]. Chaque vue architecturale prend en compte certains aspects du système en ignorant d'autres aspects. En effet, un ensemble de vues architecturales permet de comprendre un système complexe. Selon la définition IEEE 1471, l'architecture logicielle est un ensemble de descriptions architecturales. Chaque description architecturale est organisée en une ou plusieurs vues architecturales. Une vue architecturale se conforme à un point de vue particulier. Les définitions

standards de vue et de point de vue sont données par la suite :

"A view is a representation of a whole system from the perspectives of a related set of concerns" [IEE00].

"A viewpoint is a specification of the conventions for constructing and using a view. A pattern or a template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis" [IEE00]

Le choix des vues dépend de différents critères tels que : l'environment, les intérêts des utilisateurs, *etc.* En 1995, P.Kruchten a proposé le modèle de "4+1" pour la description de l'architecture [Kru95]. Ce modèle distingue quatre vues différentes : logique, processus, développement, physique et scénarios. La dernière relie ensemble les autres vues. En 2000, Hofmeister et al. ont proposé quatre vues architecturales différentes [HNS00]. Il s'agit de : la vue conceptuelle, la vue des modules, la vue code et la vue exécution. Par la suite, nous allons nous focaliser sur ces deux travaux.

2.3.1 Les "4+1" vues de Kruchten

Le travail de Kruchten et al. [Kru95] a été le premier décrivant une approche pour spécifier une architecture logicielle en utilisant différentes structures. Les 4+1 vues proposées sont :

- **La vue logique** décrit les besoins fonctionnels d'un système. Elle se rattache à une modélisation d'un système orienté objet car elle définit les abstractions en termes de classes, d'objets et de leurs relations. Cette vue peut refléter les informations comme le regroupement des classes ayant des fonctionnalités communes.
- **La vue processus** reflète une décomposition en processus du système. Elle prend en charge des aspects non fonctionnels du système comme la performance, la disponibilité et la tolérance aux fautes. Cette vue peut être présentée en différents niveaux. Au niveau d'abstraction le plus élevé, les entités seront des processus définis comme des ensembles de tâches pouvant s'exécuter de manière autonome, être répliqués, arrêtés ou relancés. Plusieurs styles architecturaux correspondant à cette vue sont présentés : le client-serveur, les tuyaux et les filtres, *etc.*
- **La vue développement** propose une décomposition en sous-systèmes qui se focalise sur l'organisation du code en différents modules ainsi que leurs dépendances. Le système logiciel peut être découpé en sous systèmes qui peuvent être développés par un ou plusieurs développeurs. Ces sous-systèmes découpent le système en espaces de nommage (*namespaces*) ou en paquetages ou en bibliothèques ou en classes, *etc.* La description peut utiliser une décomposition hiérarchique en couches où chaque couche décrit les services disponibles pour la couche supérieure.
- **La vue physique** montre la correspondance entre le logiciel d'un système et le matériel d'un système. Elle prend en compte des aspects non fonctionnels comme la performance, la tolérance aux fautes, la disponibilité. Par contre, elle se place au niveau matériel où les différentes entités sont des machines placées sur un réseau, les processus et leur tâches. Au cours du cycle de vie d'un système logiciel, il est possible que de nombreuses configurations matérielles soient utilisées, par exemple dans les phases le développement, de test ou de déploiement. Dans ce cas, les éléments ont besoin d'un minimum de chacune de ces modifications de configuration. Cette vue s'adresse aux ingénieurs système et réseau qui spécifient comment les différents processus du logiciel doivent être répartis sur le réseau.
- **La vue scénarios** regroupe les 4 vues proposées en une seule. Cette vue montre comment les 4 autres vues peuvent être reliées. Ce n'est pas une vue pour la description de l'architecture du logiciel ; cette vue illustre la coopération entre les quatre vues précédentes.

Ces vues présentées par Kruchten ne sont pas indépendantes. Elles ont des relations entre elles. Ces relations sont exprimées afin de mettre en correspondance les noeuds des différentes vues.

2.3.2 Les 4 vues de Hofmeister

Nous présentons un second exemple d'utilisation des vues multiples issu des travaux de Hofmeister et al. [HNS00]. Les auteurs ont décrit l'architecture d'un système par quatre vues architecturales différentes et bien que ces structures ont des noms différents de celles de Kruchten, il existe plusieurs similarités entre les deux ensembles de vues.

Pour chacune de ces vues, les auteurs proposent un méta-modèle décrivant les différentes entités et relations disponibles pour réaliser ces vues.

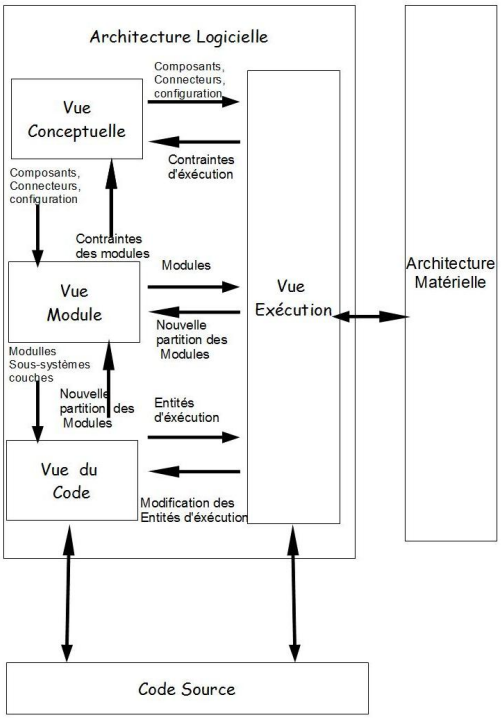

FIGURE 2.2 – Les quatre vues d'une architecture logicielle d'après Hofmeister et al. [HNS00]

- **La vue conceptuelle** : les éléments de base dans cette vue sont des composants avec les ports, et les connecteurs avec des rôles. Des rôles permettent de définir comment ces relations peuvent être liées aux ports des composants. Cette vue est la plus proche du domaine d'application car elle est la moins contrainte par les plate-formes logicielles et matérielles. Elle se rapproche de la vue logique de Kruchten.
- **La vue module** : dans cette vue, les sous-systèmes sont décomposés en modules. Elle prend en compte les besoins liés à l'implémentation du système et à l'organisation du développement. Cette vue n'est pas simplement un raffinement de la vue conceptuelle car elle décrit aussi une nouvelle structure à base de sous-systèmes, de modules et de couches hiérarchiques. Cette vue se rapproche de la vue développement de Kruchten.
- **La vue exécution** prend en compte les aspects dynamiques du logiciel. Elle décrit le système en termes d'éléments exécutables de la plate-forme (processus, tâches, ...), de ressources de

la plate-forme (mémoire, processeur, ...), de mécanismes de communication entre ces éléments exécutables (*DCOM : Component Object Model Distribué*). Cette vue se rapproche des vues processus et physique de Kruchten.
- **La vue code** montre l'organisation du code source. Elle est composée de fichiers contenant le code source, les bibliothèques, les fichiers binaires, exécutables, *etc.* Le principal but est de faciliter la construction, l'intégration, l'installation et le test du système en respectant l'intégrité des trois autres vues. Cette vue n'est pas présente dans les travaux de Kruchten.

Ces vues proposées ont des liens de dépendance. Certaines entités et relations peuvent se retrouver dans plusieurs vues. Par exemple, il existe une relation de correspondance entre les entités d'exécution dans la vue d'exécution et les fichiers dans la vue code. La figure (2.2) montre d'autres liens de dépendance. Certaines décisions de conception pour une vue peuvent affecter et remettre en cause la conception des autres vues.

Un ensemble de vues architecturales représente une description architecturale qui apporte des avantages pour la représentation de l'architecture logicielle. Par la suite, nous allons voir un certain nombre de ces avantages.

2.3.3 Et UML ?

Une question qui se pose est pourquoi ne pas utiliser UML pour la description d'une architecture logicielle ? En effet UML est devenu un standard de modélisation objet adopté par la majorité des industriels. De plus, il offre la possibilité de prendre en compte différents aspects et structures du logiciel en offrant plusieurs diagrammes de spécification. UML se concentre sur la description d'un système plutôt que sur le processus de développement qui, lui est décrit dans le RUP (Rationale Unified Process) [IBM]. Pour décrire sa propre modélisation, la notion UML met à disposition un certain nombre de diagrammes :

- **Les diagrammes de cas d'utilisation** qui permettent de définir l'ensemble des fonctionnalités du système du point de vue de l'utilisateur.
- **Les diagrammes de classe** qui modélisent la structure statique du logiciel en termes de classes et de relations entre elles.
- **Les diagrammes d'états-transition** permettent de décrire le comportement d'une classe sous forme d'automates d'états finis.
- **Les diagrammes d'activité** qui sont une variante des diagrammes d'états-transition et qui permettent de représenter le comportement interne d'une méthode ou le déroulement d'un cas d'utilisation.
- **Les diagrammes d'objets** montrent la structure statique du logiciel par un ensemble d'instances des différentes classes des diagrammes de classes.
- **Les diagrammes de collaboration** montrent les interactions entre les objets par une représentation spatiale. Ces diagrammes sont une extension des diagrammes d'objets.
- **Les diagrammes de séquence** qui, comme les diagrammes de collaboration, montrent les interaction entre les objets mais en privilégiant cette fois l'aspect temporel.
- **Les diagrammes de composants** qui décrivent les éléments physiques constituant le système ainsi que leurs relations. Les composants peuvent être organisés en paquetages, qui définissent des sous-systèmes.
- **Les diagrammes de déploiement** qui décrivent la disposition physique du matériel qui compose le système et la réparation des composants sur ce materiel.

Il est intéressant de souligner la similarité entre certains diagrammes d'UML et les vues des travaux de Kruchten : les diagrammes de cas d'utilisation pour la vue scénarios, les diagrammes de

classes pour la vue logique, les diagrammes de composants pour les vues processus et développement, et les diagrammes de déploiement pour la vue physique.

Les différents diagrammes UML sont les documents les plus couramment utilisés pour décrire un système. Ces diagrammes contiennent une image de l'architecture du système tel qu'il a été conçu. Ils peuvent donc être utilisés pour identifier des relations entre entités du code source qui n'apparaissent pas dans le code mais qui existent dans l'esprit des créateurs du système. Cependant, ces informations s'appuient sur une version idéalisée du système qui ne correspond pas forcément au système réel.

2.4 Les apports des architectures logicielles

Les architectures logicielles jouent un rôle important dans au moins six aspects du développement logiciel [Gar00] :

- la compréhension du système : les architectures logicielles rendent plus facile la compréhension d'un système complexe en le représentant à un haut niveau d'abstraction ;
- la réutilisation : la description architecturale permet la réutilisation à multiples niveaux. En identifiant les composants réutilisables d'un système, ainsi que les dépendances entre ces composants, la réutilisation du système est rendue plus facile ;
- l'évolution : l'architecture d'un système décrit comment il peut évoluer. En conséquence, il permet de connaître des limites à l'évolution d'un système ; ceci facilite la maintenance et l'estimation des coûts de modification. De plus, la description architecturale distingue les aspects fonctionnels des composants et les dépendances. Cette séparation permet de modifier facilement les mécanismes de dépendance à travers leur connexion ;
- l'analyse : les vues abstraites fournies par l'architecture permettent de mesurer différents aspects tels que la vérification de la cohérence du système [AG94][LKA+95], son respect du style [AAG93] ou l'un patron architectural, l'analyse des dépendances [AJL93] ou encore d'autres points liés à la qualité logicielle ;
- la gestion de projet : l'expérience montre que la définition précise d'une architecture logicielle est un facteur clé dans la réussite de processus de ré-ingénierie d'un système complexe en ordonnant les modifications selon leurs impactes sur la qualité du système. L'évaluation d'une architecture mène également à une meilleure compréhension des besoins, des stratégies d'implémentation, et des risques potentiels [Boe87].

Il y a un certain nombre de facteurs qui contribuent à dégrader une architecture logicielle. [Par94] a introduit la notion de vieillissement d'un logiciel montrant la façon dont un produit comme le logiciel peut vieillir au point de devenir inutilisable. D'autres raisons peuvent être *l'érosion architecturale* ou *la dérive architecturale* [PW92] ou *l'inadéquation architecturale* (*architectural mismatch*) [GAO95], *etc.* Bien que cette situation soit acceptable pendant un certain temps, elle représente une des conditions extrêmement dangereuses pour l'évolution du système. Dans le chapitre suivant, nous allons présenter les différentes raisons menant à la perte de l'architecture d'un système existant. Toutes ces raisons, par conséquent, révèlent la nécessité d'une reconstruction de l'architecture d'un logiciel existant.

2.5 Résumé

Dans ce chapitre, nous avons choisi une définition de l'architecture logicielle par rapport à notre vision du domaine. Nous avons, en premier lieu, repris les principales publications qui tentent de

définir ce qu'est l'architecture logicielle. L'étude de ces terminologies a permis de mettre en évidence l'évolution de la notion d'architecture logicielle à travers différents travaux et différentes définitions proposées. Nous n'avons pas cherché à proposer une définition, nous avons préféré choisir la définition proposée par la norme IEEE 1471, puisque cette définition couvre nos attentions.

Cette définition montre que la description architecturale peut être représentée par plusieurs vues. Dans la norme 1471, chaque vue doit être conforme à un point de vue. Chaque point de vue est défini pour couvrir des intérêts correspondant à un certain nombre d'utilisateurs. Ainsi, plusieurs travaux ont montré la nécessité de présenter l'architecture en plusieurs vues architecturales, ce qui permet de satisfaire plusieurs utilisateurs. Pourtant le fait d'avoir plusieurs vues pose la question des attentes et des intérêts à représenter par ces vues. Certaines approches ont proposé des catalogues des points de vue prédéfinis pour construire des vues.

Dans le chapitre suivant, nous allons présenter un certain nombre de critères afin d'étudier des approches de reconstruction de l'architecture existantes, puis nous classons ces approches.

Chapitre 3

La reconstruction de l'architecture logicielle

Comme nous l'avons vu dans le chapitre précédent, différents travaux ont été effectués dans le domaine de l'architecture logicielle. Cela s'explique car l'architecture fournit des avantages tout au long du cycle de vie d'un logiciel. Parmi les phases bénéficiant de la présence d'une architecture, la maintenance et l'évolution sont particulières. En effet, ces phases concernent un système existant. De ce fait, l'architecture permet de faciliter la compréhension nécessaire du système logiciel durant ces phases. Certains systèmes existants ne possèdent pas de représentation de haut niveau d'abstraction alors que d'autres ont une vue incomplète de leur architecture.

L'absence de représentation complète de l'architecture de nombreux systèmes a conduit à rechercher des méthodes pour extraire efficacement l'architecture d'un système existant.

Après avoir présenté la phase de maintenance et d'évolution et les principales problématiques existant durant ces phases, nous expliquerons les causes majeures expliquant l'absence de représentation de l'architecture ainsi que les conséquences dues à cette absence de représentation. Nous présenterons ensuite la reconstruction de l'architecture comme une approche globale acceptée par la littérature dont l'objectif est de reconstruire des vues architecturales d'un système logiciel. Ensuite nous présenterons trois modèles conceptuels pour spécifier les différents niveaux d'abstraction. Nous allons classer plusieurs approches de reconstruction architecturale selon certains critères tels que l'entrée de ces approches, la technique de reconstruction employée et le type de processus de reconstruction. Ensuite, nous présentons différentes approches existantes afin de mieux comprendre leurs mécanismes.

3.1 Evolution et maintenance des systèmes logiciels

L'évolution d'un système logiciel inclut toutes les phases du cycle de vie de ce système : développement, maintenance, migration, et retraite. Les logiciels sont condamnés à évoluer et à se complexifier continuellement pour satisfaire de nouveaux besoins, sinon ils deviennent obsolètes.

Les lois de l'évolution perpétuelle et de la complexité croissante énoncées par Lehman et Belady le stipulent [LB85] :

Loi de l'évolution perpétuelle. Un logiciel vivant se doit de changer continuellement. A défaut, il deviendra progressivement obsolète.

Loi de la complexité croissante. La structure d'un logiciel supporte mal l'évolution. Des ressources sont requises pour accompagner l'évolution d'un logiciel.

Pendant les différentes étapes d'évolution d'un système logiciel, ce dernier subit des modifications à tous les niveaux. De ce fait, l'architecture de ce système évolue, tant son implémentation que sa conception.

La maintenance d'un système logiciel est définie dans la norme IEEE [IEE98], comme étant la modification d'un logiciel après la livraison, pour corriger les fautes, pour améliorer la performance (ou attributs qualité), ou encore pour adapter le produit à un environnement modifié. En effet, cette phase est une étape importante du cycle de vie du logiciel. Elle peut avoir plusieurs parties telles que : la compréhension, les tests, les adaptations, les re-factoring, *etc.* La compréhension, en tant qu'une des étapes, est nécessaire pour des opérations de maintenance ; il faut bien comprendre le système afin de pouvoir réaliser les tâches demandées ou nécessaires.

Ainsi la maintenance et l'évolution sont devenues deux activités centrales et coûteuses du cycle de vie des logiciels de longue durée de vie et de grande taille [LS80, McK84, Som96]. Ces deux activités représentent à elles seules 50 % à 70% du coût total d'un logiciel. Par la suite, différents domaines sont apparus pour répondre aux nécessités de ces phases :

ré-ingénierie. Cette technique cherche à reconstruire le système sous une nouvelle forme. Elle vise à faire des changements importants dans le système.

rétro-ingénierie. Cette technique vise à identifier les structures du système et à fournir une représentation du logiciel à des niveaux d'abstraction plus élevés que celui de l'implémentation. Elle fournit une meilleure connaissance sur le système. Ce domaine inclut l'extraction d'architecture, la re-documentation et l'exploration du code. Selon [DDN02] pratiquement toutes les approches dédiées aux ré-ingénierie doivent commencer par une approche de rétro-ingénierie. Ces dernières permettent de fournir un modèle sur le système étudié, ce qui permettra de déterminer des obstacles existants dans le système étudié.

restructuration. C'est la transformation d'un système d'une représentation à une autre. Ce domaine inclut les techniques de re-factoring.

Les avantages de l'architecture pendant les différentes sous-phases existant durant la phase de maintenance ont rendu cette abstraction importante. L'architecture permet de faciliter les approches réalisant la maintenance et l'évolution, en proposant une vue simplifiée du système et en facilitant sa compréhension.

3.2 Une architecture pas toujours disponible

Il est important que les utilisateurs (ingénieurs, architectes logiciels, ...) aient confiance dans l'architecture fournie pour le système étudié afin de diriger les opérations de maintenance et d'évolution sur ce système. Le décalage entre l'architecture du système et la réalité implémentée dans le système peut provoquer des erreurs sur ce système. Cependant, il existe un certain nombre de facteurs qui contribuent à la perte de l'architecture d'un logiciel, justifiant ainsi la nécessité d'avoir des approches de reconstruction de l'architecture. [PW92] définissent *l'érosion* de l'architecture comme des violations dans l'architecture qui mènent à l'augmentation des problèmes du système. [GAO95] a introduit le terme *décalage* de l'architecture qui indique l'écart entre les descriptions de l'architecture et les réalisations dans le code (l'implementation).

Le phénomène de décalage entre les représentations existantes d'un système et son code source provoque des risques pendant l'évolution et la maintenance. Même les systèmes qui sont bien conçus subissent un décalage entre le niveau représentation et le niveau d'implémentation. Soit dès le dé-

but les représentations ne correspondent pas parfaitement au code source, soit des adaptations et modifications ne sont pas reportées au fil du temps dans ces représentations.

De nombreuses approches reposent en grande partie sur la représentation architecturale pour améliorer, analyser ou visualiser le système [Fav04, Riv00]. L'architecture logicielle est ainsi utilisée en tant que modèle mental partagé d'un système étudié qui représente une abstraction de haut niveau [Hol01]. Si cette architecture ne représente pas la réalité du système étudié, elle fournira une image fausse de ce système. Cela conduit à une mauvaise compréhension du système, ainsi que de mauvaises décisions qui peuvent en résulter et qui seront prises au cours de la maintenance ou de l'évolution, causant parfois des effets non désirés, inattendus, *etc.* Au fil du temps, ces problèmes peuvent entraîner la dégradation voire l'échec du système.

Pour résoudre ces manques et pouvoir bénéficier de toutes les approches de la maintenance et de l'évolution disponibles ainsi que tous les avantages de l'architecture, de nombreuses approches proposent d'extraire des représentations de l'architecture réelle d'un système logiciel.

3.3 La reconstruction de l'architecture logicielle

Riva [Riv00] définit l'extraction de l'architecture comme une activité archéologique où les analystes doivent révéler toutes les décisions de conception prises, en explorant l'implémentation et la documentation existante du système. Jazayeri [MAvdLF00] étend les sources d'informations utilisables pendant l'extraction en la décrivant comme les techniques et les processus utilisés pour révéler l'architecture d'un système à partir de l'ensemble des informations disponibles. Ducasse [DP09] présente la reconstruction de l'architecture logicielle comme une approche de rétro-ingénierie qui vise à reconstruire des vues architecturales à partir d'un système logiciel.

Différents termes sont proposés dans la littérature pour la reconstruction de l'architecture d'un logiciel tels que : rétro-architecturisation, extraction d'architecture, exploitation, récupération ou découverte. Souvent, la notion de récupération fait allusion au processus descendant alors que la découverte fait allusion au processus ascendant [DP09].

Nous présentons par la suite deux modèles conceptuels proposés par Kazman et al. [KOV01] et Koschke [Kos00]. Ces modèles décomposent l'extraction en quatre niveaux et permettent de décrire le processus d'extraction selon le passage d'un niveau à l'autre.

3.3.1 Modèle en fer à cheval de l'extraction d'architectures (*The Horse-Shoe Model*)

Kazman et al. [KWC98] présentent un framework pour analyser les processus d'extraction d'architectures. Ce framework s'appelle "HorseShoe" et il est composé de quatre niveaux. Il permet de décrire l'extraction d'architecture comme un ensemble de transitions entre ces niveaux. Les quatre niveaux sont :

- **le niveau du code source (*source level*)** : ce niveau est celui des fichiers du code source. En fait c'est une représentation textuelle du code source ;
- **le niveau de la structure du code source (*code structure level*)** : ce niveau présente le code source dans une forme plus formelle que le niveau précédent. Ce niveau permet une analyse syntaxique du code source ;
- **le niveau fonctionnel (*function level*)** : ce niveau représente les relations parmi les fonctions, données et modules. Il fournit une vue globale et résumée du système ;

– **le niveau architectural (*architectural level*)** : ce niveau est constitué des éléments architecturaux comme composants et connecteurs.

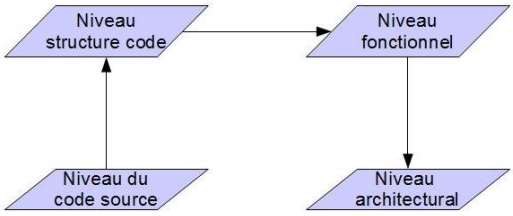

FIGURE 3.1 – Les quatre niveaux de l'extraction selon le modèle de Kazman

La figure (3.1) montre les quatre niveaux présentés. L'extraction de l'architecture commence souvent à partir du code source. Après avoir effectué des analyses syntaxique(s) et sémantique(s) sur le code source, certaines représentations sont produites au niveau structure du code source. Ces représentations sont souvent des arbres syntaxiques qui constituent le niveau de la structure du code. Les représentations produites vont être soumises à une analyse plus technique portant, par exemple, sur les flots de données ou de contrôle. Le niveau fonctionnel, donc, représente les fonctions et autre modules et leurs relations. Cependant des éléments représentés par cette phase ne sont pas assez abstraits pour être considérés comme l'architecture. La dernière étape consiste à abstraire les regroupements du niveau fonctionnel en entités architecturales (cf.Figure 3.1).

3.3.2 Modèle conceptuel de l'extraction d'architectures

Koschke [Kos00] propose une amélioration de la méthode de Kazman. Il considère que les deux premiers niveaux reflètent le code source du système. Il montre également que le niveau fonctionnel établit le lien entre les niveaux architectural et code source. Koschke définit les quatre niveaux suivants :

– **le niveau du code (*lower code level*)** : ce niveau contient les expressions et instructions contenues dans les fonctions ;
– **le niveau global du code (*global code level*)** : ce niveau contient les entités que Koschke appelle les quarks architecturaux. Ces quarks sont les éléments basiques de l'extraction d'architecture. Ces entités vont être regroupées pour former les éléments architecturaux ;
– **le niveau architectural inférieur (*lower architectural level*)** : ce niveau contient les regroupements des quarks. C'est le premier niveau du modèle qui possède des informations supplémentaires par rapport au code source. Ce niveau se situe entre le code source et la représentation architecturale ;
– **le niveau architectural supérieur (*higher architectural level*)** : ce niveau représente l'architecture du système en termes de sous-système(s) et connecteur(s).

La figure (3.2) montre ces quatre niveaux présentés par Koschke. Les niveaux de ce modèle mettent en évidence la séparation entre la partie code source et la partie architecturale. Les deux premiers niveaux contiennent des éléments présents dans le code source sans information supplémentaire, alors que les deux autres niveaux représentent la partie architecturale. Le niveau architectural inférieur présente les regroupements de quarks et leur relations. Ce niveau ne contient pas des éléments architecturaux puisque le niveau d'abstraction n'a pas changé par rapport au code source. Ce niveau établit les correspondances entre les éléments architecturaux et les entités du code source. Le niveau

architectural supérieur contient les éléments architecturaux abstraits depuis les regroupements du niveau architectural inférieur.

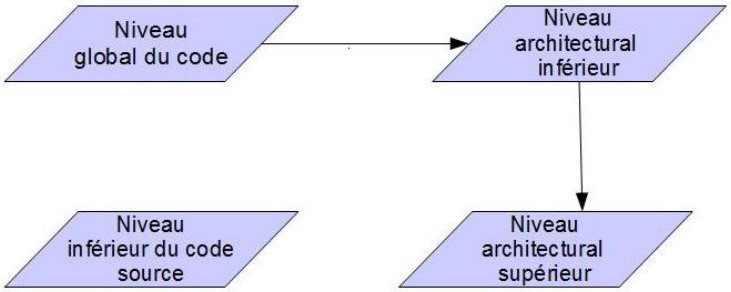

FIGURE 3.2 – Les quatre niveaux de l'extraction selon le modèle de Koschke

3.4 Classification des approches de reconstruction d'architectures logicielles

[DP09] propose une classification détaillée basée sur le cycle de vie des approches de reconstruction de l'architecture logicielle selon cinq critères : *but*, *processus*, *entrée*, *technique* et *sortie*.

Focalisant sur la définition de l'architecture logicielle proposée par la norme 1471 (présentée dans le chapitre précédent), nous avons pu remarquer l'influence d'un certain nombre des critères lors de l'étude sur la construction de l'architecture : *les vues architecturales*, *les points de vue* et *les utilisateurs*. Ces derniers sont ainsi inclus dans les critères catégorisées par [DP09]. Les vues architecturales et les points de vues sont considérés comme *les entrées* des approches de construction de l'architecture. Les utilisateurs fournissent des attentes et des souhaits pour une construction de l'architecture. Ces attentes et souhaits seront, éventuellement, retrouvés dans l'architecture construite. Le niveau d'intervention de ces utilisateurs pendant une construction de l'architecture représente le mode d'utilisation des algorithmes de construction liés aux *techniques* de construction.

Les approches ont différentes manières pour surmonter l'incohérence existant entre l'architecture implémentée et l'architecture attendue. Ce qui éclaircit l'existence de différents types de *processus* de construction.

Nous avons choisi un cadre de comparaison portant sur trois axes parmi les cinq précédemment proposés : *entrée*, *processus*, et *technique*. Avant d'aller plus loin, nous donnons des définitions générales pour ces trois critères :

- *Entrée* : la plupart des approches de reconstruction de l'architecture sont basées sur des informations du code source et des expertises. Cependant, d'autres types d'entrées existent tels que : les informations dynamiques ou des points de vue ;
- *Processus* : trois types de processus sont distingués pour la reconstruction de l'architecture : ascendant, descendant et hybride ;
- *Technique* : la communauté de recherche établit différentes techniques de construction selon l'autonomie des approches existantes : techniques "quasi-manuelles", "semi-automatiques" et "quasi-automatiques".

3.4.1 Les entrées pour la reconstruction de l'architecture logicielle

La plupart des approches proposées pour la reconstruction de l'architecture logicielle sont basées sur les représentations du code source, mais d'autres types d'entrées peuvent être également considérés tels que : les informations dynamiques ("dynamic information"), les informations historiques ("historical information"), des styles architecturaux, des points de vue, *etc.* Ces entrées sont classées en deux catégories [DP09] : entrées non-architecturales et entrées architecturales.

Entrées non-architecturales

Une information est qualifiée de *non-architecturale* si elle ne touche pas à des concepts architecturaux. Le code source ou les informations dynamiques sont considérés comme des entrées non-architecturales. Certaines entrées non-architecturales utilisées dans différentes approches sont représentées :

Code source : fréquemment, le code source de l'application est requis par les approches de reconstruction car il est bien souvent disponible et fiable en tant que source d'informations. Certaines approches interrogent le code source en utilisant des expressions régulières telles que des approches : RMTool [MN97], [Mur96] ou [PFGJ02]. La plupart des approches ne commencent pas directement par l'exploitation du code source sous sa forme textuelle, mais elles commencent à partir d'une représentation d'un modèle du code source. Cette représentation est fournie en utilisant des méta-modèles appropriés avec le paradigme des logiciels étudiés. Certaines approches comme Nimeta [Riv04] et ArchView [Pin05] ont utilisé le méta-modèle FAMIX [TDD00] pour les applications orienté-objet. Le méta-modèle FAMIX [TDD00] modélise les aspects conceptuels d'un logiciel orienté objets de manière indépendante du langage orienté objet utilisé [DTD01]. Les concepts de Famix sont des classes, méthodes, appels, *etc.* D'autres méta-modèles comme Dagstuhl Middle Méta-modèle [LTP04], et GXL [HSSW06] sont aussi proposés avec les mêmes intentions ; c'est-à-dire la représentation du code source.

Les informations textuelles et symboliques : certaines approches utilisent les informations symboliques retrouvées parmi des commentaires [PFGJ02] ou dans le nom des méthodes [KDG07].

L'organisation physique : l'organisation physique d'une application est composée par exemple des fichiers et dossiers ou paquetages. Ils révèlent souvent des informations architecturales. [HRY95] et [YHC97] prennent en compte l'organisation structurelle des éléments physiques (des éléments tels que les fichiers, les dossiers ou des paquetages). Certains comme [LSP05] et [PGF05] mettent en correspondance ces éléments (les paquetages ou les classes) avec des composantes architecturales. Des hiérarchies existantes entre les différentes organisations physiques sont aussi considérées comme une entrée architecturale.

L'expertise : les approches de reconstruction de l'architecture logicielle ont besoin d'interagir avec les utilisateurs afin de guider et d'approuver les résultats issus de la reconstruction. Pour proposer une architecture conceptuelle, comme dans les approches [MJ06] et [MNS95], des rétro-ingénieurs ont besoin d'étudier des exigences du système, lire des documents disponibles, effectuer des entretiens avec des utilisateurs, enquêter sur les hypothèses et analyser le domaine métier, *etc.* La connaissance d'experts est nécessaire également pendant la définition de points de vue ou la sélection des styles.

Entrées architecturales

Une information est qualifiée d'*architecturale* si elle touche à des concepts architecturaux. Les points de vue et les styles architecturaux sont des informations architecturales. Certaines entrées architecturales utilisées dans différentes approches sont représentées.

Style : les styles architecturaux populaires sont les *pipes and filters*, *layers* (couches), *data flow* (flux de données), *etc.* Ils sont populaires parce que, comme les modèles de conception, ils représentent des situations récurrentes d'architecture [BMR+96]. Les processus de reconnaissance des styles architecturaux sont encore un challenge à ce jour, car ils couvrent plusieurs éléments architecturaux et peuvent être appliqués de différentes façons [PFGJ02], [Riv04], [Kri99].

Dans l'approche Focus [DM01], les auteurs utilisent des styles architecturaux pour inférer une architecture conceptuelle. Cela permet de faire une comparaison avec l'architecture concrète extraite à partir du code source.

Medvidovic et al. ont proposé une approche afin d'arrêter l'érosion de l'architecture [MEG03]. Dans un processus ascendant, ils prennent en compte des exigences afin de définir un modèle conceptuel de haut niveau. Dans un processus descendant, ils utilisent l'implémentation du système en tant que connaissance de bas niveau afin d'extraire une architecture concrète. Les deux architectures conceptuelles et concrète sont construites en plusieurs étapes. Le rétro-ingénieur doit rapprocher les deux architectures en se basant sur les styles architecturaux. En effet, l'approche considère les styles architecturaux en tant qu'une clé de conception. L'objectif est d'arriver au point de prendre des décisions conceptuelles, de capturer des logiques concernées et de révéler des compositions d'éléments architecturaux efficaces.

Point de vue : l'architecture d'un système est comme un modèle mental pour des utilisateurs [Hol01]. Puisque des utilisateurs ont des intérêts différents, la considération des points de vue dans des approches de reconstruction de l'architecture est un aspect important [IEE00]. Pour cela, des catalogues de points de vue sont proposés : les 4+1 points de vue de Kruchten [Kru95] (logique, processus, développement, physique et scénario) et les quatre points de vue de Hofmeister [HNS00] (conceptuelle, module, exécution et code). La plupart des approches de reconstruction de l'architecture génèrent des vues architecturales correspondant avec un ou plusieurs points de vue pré-définis. Smolander et al. [SHI+01] ont souligné que des points de vue ne peuvent pas être standardisés mais ils peuvent être sélectionnés ou définis en accord avec l'environnement du système.

L'approche Symphony proposée par [DHK+04] a comme objectif la reconstruction de l'architecture en utilisant certains points de vue. Ces points de vue vont être également choisis à partir d'un catalogue de points de vue.

Le tableau 3.1 récapitule les différents types d'entrée ainsi que certaines approches étudiées.

Les processus de reconstruction de l'architecture logicielle

Les approches de reconstruction de l'architecture suivent un processus ascendant, descendant ou hybride [DP09]. La qualification du processus se base sur le niveau d'abstraction de l'information en entrée et en sortie. Le code source est une source d'information considérée comme étant à un bas niveau d'abstraction. A l'inverse, une architecture conceptuelle ou un style architectural sont des sources d'information considérées comme étant à haut niveau d'abstraction. Ainsi :

– processus ascendant : ce processus commence avec la connaissance de bas niveau (niveau d'im-

Processus de construction	
Processus	Approche
Code source et expertise	Dali [KC99] ARMIN [KOV03] Bunch [MM98] Intensive [WD01] RMTool [MNS95] [Mur96] PuLSE / SAVE [KMNL06]
Organisation physique	[HRY95] [YHC97] [LSP05] [PGF05]
Expertise et point de vue	[SHI+01] Cacophony [Fav04]
Code source, expertise et information textuel	[KDG07] Revealer [PFGJ02]
Code source, expertise et point de vue	SARTool [Kri99]
Code source, expertise, styles	Focus [MJ06] [MEG03] Alborz [Sar03]
Expertise, information dynamique et point de vue	Nimeta [Riv04] [Kru95] [HNS00]
Code source, expertise, information dynamique, et historique	ArchView [Pin05]

TABLE 3.1 – Des approches étudiées selon l'entrée considérée

plémentation) du système logiciel pour récupérer l'architecture de haut niveau. Ce processus commence généralement à partir de modèles de code source, et permet de glisser progressivement vers les niveaux plus abstraits que celui du code source [SFM99]. Les processus ascendants sont proches et reliés au processus extraction-abstraction-présentation décrit par Tilley et al. [TSP96]. Les analyses effectuées sur le répertoire contenu de code source, permettent de fournir des représentations abstraits du système. Certaines approches utilisant le processus ascendant sont : Dali [KC99], Revealer[PFGJ02], Intensive [MMW02]. La figure 3.3 (extraite de [DP09]) montre ce type de processus.

- processus descendant : il commence avec des connaissances de haut niveau telles que des besoins des utilisateurs ou des styles architecturaux et vise à découvrir l'architecture en formulant ces hypothèses conceptuelles et en vérifiant ces hypothèses vis-à-vis du code source [CTH95]. Certaines approches utilisant le processus descendant sont RMTool [MNS95] et PBS [FHK+97]. La figure 3.4 illustre un processus de type descendant. Une architecture de haut niveau a été définie, et comparée avec le code source. L'architecture sera raffinée à plusieurs reprises.
- processus hybride : la figure 3.5 montre un processus hybride qui combine les deux processus précédents. L'objectif est d'obtenir une conformité entre les deux résultats obtenus par les processus ascendant et descendant [SFM99]. D'une part, la connaissance de bas-niveau est extraite et abstraite en utilisant différentes techniques de construction, d'autre part, la connaissance de haut niveau est raffinée et confrontée en face de vues extraites précédemment. Cette combinaison va être fréquemment utilisée pour arrêter l'érosion architecturale en harmonisant l'architecture conceptuelle (par processus descendant) et concrète (par processus ascendant). Certaines approches utilisant le processus hybride sont : Nimeta [Riv04], Symphony [DHK+04].

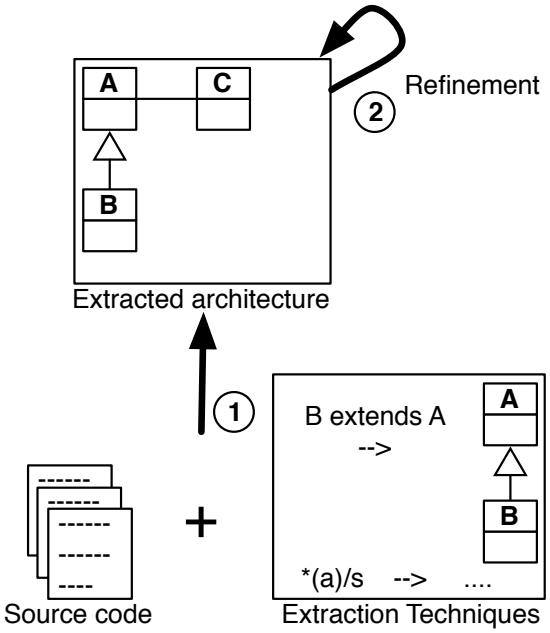

FIGURE 3.3 – Un processus ascendant à partir du code source : (1) : l'extraction de l'architecture et (2) le raffinement, extraite de [DP09]

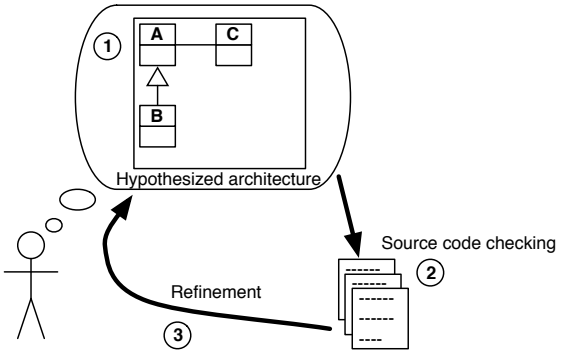

FIGURE 3.4 – Un processus descendant : un architecture proposée est définie, (2) l'architecture est comparée avec le code source, (3) le raffinement de l'architecture, extraite de [DP09]

Le tableau 3.2 récapitule les trois processus de construction proposés ainsi qu'un certain nombre des approches étudiées.

Par la suite, nous allons présenter différentes techniques de construction pour les approches de construction de l'architectures.

Les techniques de reconstruction de l'architecture logicielle

Les techniques de reconstruction de l'architecture sont souvent en corrélation avec des données qu'elles exploitent. Par exemple [Wuy01] et [MKPW06] réalisent des requêtes logiques sur des faits. Alors que [EKRW02] réalisent des requêtes sur des graphes. Le degré d'automatisation du processus de reconstruction est avant tout lié à l'algorithme sur lequel il repose. Les techniques ont été dans un premier temps classées selon leur niveau d'automatisation envisageable. Cependant nous précisons

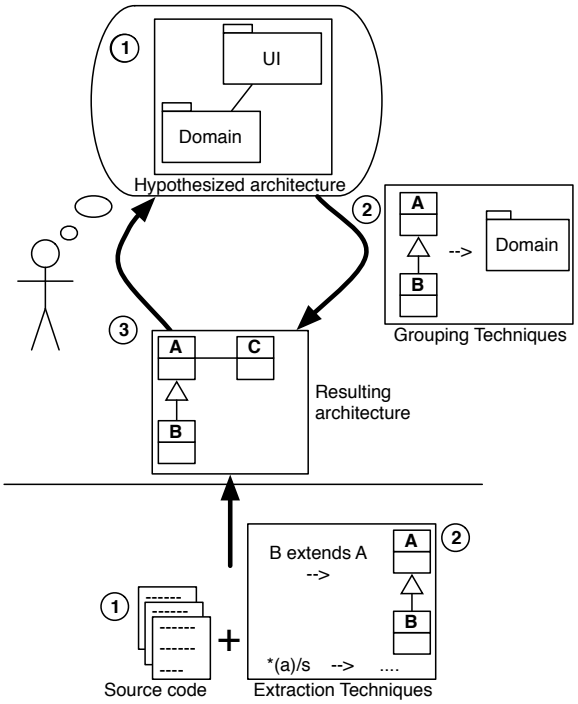

FIGURE 3.5 – Un processus hybride : une combinaison entre le processus descendant et ascendant,extraite de [DP09]

Processus de construction	
Processus	Approche
Ascendant	Dali [KC99] ARMIN [KOV03] Revealer [PFGJ02] Bunch [MM98] Intensive [WD01] ArchView [Pin05] SARTool [Kri99]
Descendant	RMTool [MNS95] PuLSE / SAVE [KMNL06]
Hybride	Focus [MJ06] Alborz [Sar03] Nimeta [Riv04] Cacophony [Fav04] Symphony [DHK+04]

TABLE 3.2 – Des approches étudiées selon le processus de construction

que cette classification n'est pas très précise et les catégories ne sont pas exclusives.

Techniques "Quasi-manuelles". En général, cette technique est employée par de nombreux outils de visualisation. Elle se résume à filtrer et grouper interactivement des entités. C'est-à-dire que le rétro-ingénieur identifie manuellement des éléments architecturaux. Les approches utilisent souvent un algorithme spécifique et un critère de regroupement d'une façon manuelle [Big89, FHK+97, MJ06]. Nous pouvons dénoter certaines approches se situant dans cette caté-

gorie telles que : CodeCrawler [LD03] ou Verso [LSP05]. Elles prennent en compte des connaissances de bas niveau pour reconstruire l'architecture d'un système étudié.

Techniques "Semi-automatiques". Cette technique permet au rétro-ingénieur de spécifier des règles d'abstraction réutilisables et de les exécuter de manière ascendante et automatique. Les entités du code source sont représentées sous forme de faits. Dans les approches utilisant cette technique, le rétro-ingénieur doit exécuter l'outil proposé d'une façon manuelle. Ensuite, l'outil continue à découvrir ou abstraire les entités du système étudié d'une façon automatique. Ces approches peuvent être classées en deux types. Les approches qui ressemblent aux approches automatiques. Elles proposent des mécanismes pour automatiser certains aspects de la reconstruction. Par contre, cette automatisation ne portent que sur des aspects simples du processus, elle ne permet pas de réduire le besoin en expertise. De ce point de vue, elles se situent au même niveau d'expertise de l'utilisateur que les approches manuelles. Elles utilisent souvent des algorithmes spécifiques [KOV01, MNS95]. Le second type d'approches semi-automatiques regroupe des approches qui sont en grande partie automatisées, mais qui permettent à l'utilisateur d'intervenir pour initier le processus. Elles sont en effet moins dépendantes de l'expertise de l'utilisateur puisqu'elles automatisent toutes les phases de la reconstruction. Elles utilisent des algorithmes de regroupement et d'exploration associés à un critère de regroupement à bases de métriques ou de graphes [MM01].

[DP09] classe des approches basées sur la technique semi automatique de la façon suivante :

- **approches basées sur l'abstraction** : ces techniques reposent sur une démarche consistant à faire la correspondance entre des concepts de bas-niveau et des concepts de haut-niveau. Le rétro-ingénieur spécifie des règles d'abstraction réutilisables afin d'être exécutées. Le tableau 3.3 représente les approches se situant dans cette catégorie avec des explications sur les façons de les appliquer.
- **approches basées sur l'investigation** : ces techniques reposent sur une démarche qui effectue la correspondance entre des concepts de haut-niveau vers des concepts de bas-niveau. Des concepts de haut-niveau considérés couvrent souvent des descriptions architecturales, des styles architecturaux, des patrons, *etc.* Le tableau 3.3 aussi représente les approches se situant dans cette catégorie avec des explications sur les façons de les appliquer.

Techniques "Quasi-automatiques". Ces techniques automatiques reposent sur des algorithmes de regroupement ou d'exploration à partir de critères de regroupement à base de métriques ou de graphes. L'outil automatise toutes les phases ; donc le rétro-ingénieur n'a qu'à contrôler des étapes itératives effectuées dans un processus d'extraction. Dans ces approches, l'utilisateur ne peut pas intervenir pendant le processus. La seule façon d'agir est donc de faire varier les différents paramètres modifiables du processus. Les approches faisant partie de cette catégorie combinent souvent différentes techniques utilisant des analyses reposant sur des concepts, dominance et regroupement (clustering). Le tableau 3.4 représente certaines des approches étudiées selon cette technique, ainsi des explications sur les façons de les appliquer.

Les modes d'utilisation

Un critère considéré pour définir les approches est le mode d'utilisation du processus de reconstruction. En effet, selon les approches, le processus peut être itératif ou séquentiel [Cha09].

Séquentiel. Les approches séquentielles sont les plus utilisées. Selon ces approches, le processus de reconstruction est exécuté une seule fois. Il ne prend en compte aucune exécution précédente.

Techniques Semi-Automatiques		
Technique	Approche	Description
Abstraction	Dali [KOV01] ARMIN [KOV03]	utilisent des requêtes relationnelles afin de définir un ensemble des règles d'abstraction ou de composition pour construire une vue architecturale.
	[GSZ04, KP96]	utilisent des requêtes logiques afin d'identifer des patrons conceptuels.
	[DGLD05]	utilisent la programmation objet-orientée afin de manipuler des modèles représentant différentes entrées
	[MNS95]	focalisent sur le champ lexical et les informations structurelles (requêtes lexicales) dans le code source afin d'identifer des éléments architecturaux et des relations entre eux.
Investigation	ManStart [YHC97] X-Ray [MK01]	sont basées sur une connaissance des styles architecturaux ou des patrons.
	ARM [GAK99]	fait des correspondances entre des patrons en forme de graphes avec des representations du code en forme de graphes.
	[MNS95, Mur96]	sont basées sur le modèle de Réflexion. En faisant la correspondance entre des entités de haut-niveau avec des entités du code source.

TABLE 3.3 – Des approches étudiées selon la technique Semi-Automatique.

Afin d'affiner des résultats (si nécessaire), le processus doit être appliqué une nouvelle fois en utilisant les paramètres fournis par l'utilisateur.

Itératif. Les approches itératives proposent des processus itératifs de reconstruction qui permettent de réutiliser les résultats des exécutions précédentes. Ces approches permettent d'affiner progressivement les résultats de manière automatique.

Par la suite, nous étudions certaines approches dans le domaine de la reconstruction de l'architecture.

3.4.2 Un regard plus précis sur les approches étudiées

La reconstruction de l'architecture fournit une ou plusieurs représentations architecturales pour : répondre aux objectifs des utilisateurs [DLdOdlP98], être une base pour la re-documentation, vérifier la conformité entre ces représentations architecturales et le code source, *etc.*

Les méthodes de reconstruction prennent bien souvent en compte un cycle d'extraction-abstraction-présentation. Des sources d'information disponibles utilisées par ces approches sont analysées afin de récupérer les informations nécessaires. Les résultats de ces analyses sont accumulées dans un répertoire. Ces informations sont ensuite traitées afin d'obtenir des représentations abstraites du système

Techniques Quasi-Automatiques		
Technique	Approche	Description
prise en compte des Concepts	[ABN04, EKS03]	l'objectif d'analyse des concepts est d'identifier des patrons conception, des features ou des modules
prise en compte des algorithmes de clustering	[ACN02] [AL99b]	utilisent des algorithmes de clustering selon des conventions de nom ou des conventions de interactions
prise en compte des algorithmes utilisant l'analyse de prédominance	[CV95] [GK97] [LL03]	appliquant des analysis prédominance, afin de retrouver des composant passives, mais pour reconstruire l'architecture d'autres types d'analyses sont nécessaires.
prise en compte des couches et des matrices	[SJSJ05] [BP01]	utilisent des matrices et des couches afin de montrer des dépendances potentielles entre des éléments d'un système étudié.

TABLE 3.4 – Des approches étudiées selon la technique quasi-automatique.

étudié. Tilley et al. [TSP96] décrivent une approche d'extraction-abstraction-présentation plus détaillée. Ils mentionnent plusieurs étapes : *la récupération des données*, *l'extraction des connaissances* et *la présentation*. Cette approche décrit comment les informations peuvent être extraites, ensuite être filtrées et enfin être représentées. L'objectif est d'avoir des représentations de haut niveau qui peuvent améliorer la compréhension.

Laine propose une approche manuelle réalisée chez Nokia pour la reconstruction de l'architecture [Lai01]. Cette approche permet de construire un aperçu de haut niveau d'un système étudié. Ainsi elle permet de faire la correspondance entre le code et différentes parties de ce modèle de haut niveau construit. Cela permet de remplir le vide existant entre un modèle de haut niveau et le code source. Par contre l'identification des composants de ce modèle se réalise d'une façon manuelle à partir du code source en utilisant les utilitaires d'UNIX, EMACS et GREP.

SAR Méthode de Krikhaar : Krikhaar [Kri99] propose un framework qui correspond également à l'approche extraction-abstraction-présentation pour la reconstruction de l'architecture. Ce framework inclut un processus précis pour sélectionner les sources d'information afin de créer des vues architecturales de haut-niveau. Dans cette approche deux termes sont introduits : *Infopack* et *ArchiSpect*. Un Infopack est un paquetage de l'information extraite à partir du code source, des documentations ou d'autres sources d'information. Une ArchiSpect est une vue sur le système ; elle rend explicite la structure de l'architecture du système. En effet elle est plus abstraite et largement plus applicable qu'un Infopack. Un ensemble complet de ArchiSpects décrit l'architecture représentée dans un système étudié ; en revanche des InfoPacks servent pour soutenir la construction des ArchiSpetcs. Cette approche, donc, focalise sur l'extraction d'un ensemble des vues architecturales afin de présenter l'architecture d'un système étudié. Cependant, elle ne prend en compte que le style module pour extraire les vues architecturales.

NIMETA : Riva [Riv04] propose une approche nommée NIMETA pour reconstruire une architecture basée sur l'extraction de vues. Cette approche permet de traiter le problème de la définition d'une démarche pour récupérer les informations architecturales qui sont importantes au niveau de l'implémentation d'un système logiciel. L'objectif est de permettre aux architectes

logiciels d'avoir une compréhension précise, cohérente et détaillée du système. Cette approche est basée sur la récupération des points de vue architecturaux. Il s'agit d'un processus de rétro-ingénierie dont le but est de récupérer une architecture logicielle documentée en traitant des artefacts disponibles (comme le code source) et en interrogeant les experts du domaine. Dans cette approche, les concepts architecturaux jouent un rôle de première classe dans le processus de reconstruction. L'activité de détermination du concept s'effectue en collaboration avec les architectes. NIMETA comporte trois phases : le processus de conception, la récupération de vue et l'interprétation de résultats. Chaque phase est composée de plusieurs activités avec des objectifs différents. L'objectif de l'activité de *définition du problème* est de connaître les objectifs et les résultats attendus de la reconstruction. L'objectif de la deuxième activité, *détermination de concept* vise à récupérer des concepts architecturaux ainsi qu'à définir les points de vue nécessaires pour résoudre des problèmes identifiés par l'étape précédant. L'objectif de la troisième activité, *l'acquisition de données*, est axé sur l'extraction des données à partir de sources d'informations et la création d'une vue du source. L'objectif de la quatrième activité, *extraction de la connaissance (knowledge inference)* est de déduire des vues architecturales à partir de la vue source. L'objectif de la cinquième activité, *la présentation* est de présenter et de communiquer ces vues architecturales.

Le modèle réflexif ("Reflexion model") : Murphy et al.[MN97] proposent une technique de reconstruction basée sur des modèles réflexifs. Dans ces modèles, l'utilisateur commence à partir d'une structure de haut niveau proposée en tant que vue, puis cette vue va être raffinée d'une façon itérative obtenir des connaissances sur le code source du système étudié. Le résultat montre la différence entre le modèle de haut niveau considéré par un développeur et le modèle extrait. La technique est basée sur une correspondance ("mapping") entre le code source et les concepts du modèle de haut niveau. Faire la correspondance entre un modèle de haut niveau et un modèle représentant le système étudié permet d'avoir une corrélation entre deux niveaux.

MITRE : Harris et al. [HRY95] proposent une approche pour la reconstruction de l'architecture en combinant la reconstruction suivant un processus descendant avec la reconstruction suivant un processus ascendant. L'analyse descendante permet de visualiser un aperçu de la structure des fichiers dans une vue ("bird's eye"). Ainsi, une technique de "grouping" permet d'organiser ces fichiers. L'analyse ascendante utilise des styles architecturaux pour guider le processus de reconstruction. Ces styles architecturaux supposés sont définis et recherchés dans l'implémentation. Une fois le style reconnu, la correspondance entre le style et sa réalisation permet de représenter l'architecture existante (*as-built*) du système. Les styles architecturaux recherchés sont pris à partir d'une bibliothèque prédéfinie de styles. L'approche est limitée aux styles architecturaux qui sont définis dans la bibliothèque. Cette approche essaye de construire des modèles de haut niveau en considérant des styles architecturaux du système. Cependant, ces styles sont prédéfinis et ils ne sont pas construit selon des propositions des utilisateurs au cours de la construction des modèles de haut niveau.

REVEALER : Pinzger et al. [PFGJ02] proposent une approche basée sur les patrons. Pour cela ils se sont basés sur les informations lexicales et structurelles dans le code source. Ils localisent certains points essentiels d'un patron dans le code source et considèrent ensuite ces points en tant que points de départ de la reconstruction de l'architecture. Leur approche permet de définir les dépendances architecturales appropriées qui doivent être récupérées à partir de l'implémentation. Ces objectifs sont réalisés dans REVEALER qui est un outil d'extraction à partir du code source. Cet outil combine les avantages des analyses lexicales et des analyses syntaxiques.

SOUL : Mens et al. [MMW01] propose une méthode pour vérifier la conformité de l'architecture basée sur PROLOG. Des artefacts implémentés vont être mis en correspondance ("mapping")

avec un LDA (Langage de Description Architectural) pour décrire l'architecture conceptuelle. L'utilisation de la programmation logique permet de définir facilement les règles de correspondance et les règles de conformité. La méthode ne supporte que le langage Smalltalk, par contre elle pourrait être étendue à d'autres langages.

ROMANTIC : ROMANTIC [SAMD08] vise à extraire une architecture à base de composants à partir d'un système orienté objet existant. L'idée première de cette approche est de proposer un processus quasi-automatique d'identification d'architectures en formulant le problème. Ce dernier est traité en tant qu'un problème d'optimisation, et en essayant de le résoudre au moyen de méta-heuristiques. Ces dernières explorent l'espace composé des architectures pouvant être abstraites du système en utilisant la sémantique et la qualité architecturales pour sélectionner les meilleures solutions. Pour identifer la meilleure architecture représentant un système donné, les auteurs ont proposé une fonction d'évaluation de la qualité afin de retrouver l'architecture optimisé. Cependant, différentes sources d'informations sont utilisées telles que : code source, informations intentionnelles et documentations. Certaines méthodes sont utilisées afin de réduire l'espace des recherches de ces informations.

Hapax : Kuhn et al. [KDG05] proposent une approche pour analyser les informations sémantiques dans une perspective de rétro-ingénierie. Ils utilisent LSI «Latent Semantic Indexing », une technique de recherche qui récupère la similarité sémantique dans des documents (classes, méthodes) en traitant les termes (identifiant, commentaires). Ils recoupent la similarité sémantique entre les différentes entités et les regroupent en fonction de leur similitude.

Intensive : Mens et al. [MKPW06] proposent un environnement d'extraction des vues intentionnelles. Une vue intentionnelle est un ensemble d'entités du code source (classes ou méthodes) qui sont similaires du point de vue structurel. Cet environment possède une partie pour créer et manipuler des vues. Dans cette partie, des auteurs utilisent des requêtes logiques écrites dans le langage SOUL [Wuy01] pour regrouper des entités associées au code source au sein des vues construites. Ils permettent également de vérifier la cohérence entre différentes vues alternatives. Intensive expose la cohérence et l'incohérence entre des vues extraites vers le code et entre les vues extraites. Les auteurs ont utilisé CodeCrawler [Lan03] pour visualiser les vues obtenues.

Cacophony : Favre [Fav04] présente une approche générique basée sur un méta-modèle pour reconstruire l'architecture. Cacophony reconnaît des points de vue selon les préoccupations des utilisateurs concernés. La seule différence entre cette approche et d'autres travaux comme Symphony est la vision différene qu'elle a du point de vue. L'approche déclare qu'un point de vue pour une vue est comme un méta-modèle pour un modèle. Selon Cacophony, ces points de vue peuvent être définis comme des méta-modèles ; ainsi le point de vue guide le processus de construction des vues ou le processus de construction des modèles.

PROCSSI : [MM00] propose une utilisation combinée de l'information sémantique et de l'information structurelle des systèmes étudiés. L'objectif est d'appuyer la compréhension nécessaire dans les phases de maintenance et ré-ingénierie des systèmes logiciels étudiés. Les informations sémantiques considérées concernent le domaine du problème et le domaine du développement du sytème logiciel étudié. Les informations structurelles considérées concernent la structure syntaxique d'un système étudié. Les auteurs utilisent la méthode "Latent Semantic Indexing" afin de définir une métrique de similarité sémantique entre des composants du système logiciel étudié. Les composants seront ensuite regroupés (en "cluster") selon cette métrique. Les auteurs considèrent des informations telles que l'organisation structurelle des fichiers du système étudié afin de pouvoir évaluer la cohésion entre ces "clusters". L'approche proposée fait partie des approches automatiques. Les regroupements se réalisent d'une façon purement automatique.

3.5 Conclusion

Dans ce chapitre, nous avons souligné le lien entre l'architecture logicielle et la maintenance et l'évolution des systèmes logiciels. Cependant, nous avons remarqué que beaucoup de systèmes ne disposent pas d'une représentation fiable de leur architecture pour différentes raisons. Cette indisponibilité peut amener un décalage entre la représentation et la réalité du système, ce qui, au fil du temps, augmente le risque d'incohérence du système.

Nous avons considéré certains critères dans le processus de reconstruction de l'architecture tels que : *processus*, *entrée* et *technique*. Ces critères nous ont permis de classer les approches existantes.

Les principales remarques concernant les approches proposées sont :

- la plupart des approches n'abordent souvent que le niveau du code source et les outils pour la compréhension de ce niveau, et leur objectif initial n'englobe pas le niveau conceptuel ou architectural ;
- certaines approches présentées s'appuient sur le concept de vues multiples mais la navigation horizontale ou verticale entre ces vues n'est que légèrement abordée.

Un des objectifs clairs de cette étude est de fournir un moyen d'identifier des abstractions qui représentent des vues architecturales d'un système logiciel existant. A ce titre, deux sources d'information peuvent être utiles : (a) des informations provenant des utilisateurs et (b) des artefacts du système étudié tel que le code source. Par contre ces deux types d'informations ont certaines limites. En effet, au moment de prendre en compte des informations fournies par l'utilisateur, certains problèmes apparaissent :

- à cause du manque de certitude dans les documentations existantes ou des architectures conceptuelles proposées par des utilisateurs, les informations proposées peuvent être incomplètes ou erronées. De ce fait, une approche proposée doit prendre en compte des moyens pour vérifier et justifier en quelque sorte ces informations extraites. L'objectif doit être de fournir des mécanismes simples pour assister des utilisateurs pour raffiner, composer ou améliorer les résultats obtenus selon ces souhaits ou selon d'autres informations supplémentaires ;
- une approche pour la reconstruction de l'architecture d'un système logiciel doit être en corrélation avec les attentes, les souhaits et les informations proposées par l'utilisateur, ainsi qu'avec le système implémenté. Pour la parti d'informations proposées (et les attentes et souhaits), cela peut être réalisable en s'appuyant sur des points de vue multiples.

D'un autre côté, prendre en compte des informations issues du code source est intéressant car le code source correspond très généralement à la réalité du système. Cependant la reconstruction de l'architecture à partir du code a également certaines limites :

- prise en compte d'une quantité importante d'informations difficiles de ce fait à traiter ;
- l'architecture n'est pas explicite dans le code source.

La plupart des approches proposées utilisent des techniques automatiques dans leur processus de rétro-ingénierie sur les systèmes logiciels existants. En revanche, peu d'approches ont essayé de capturer des informations et des attentes de la part des utilisateurs.

Nous avons remarqué dans ce chapitre le besoin d'avoir une approche fondamentale et générique supportant la construction des vues architecturales. Une approche qui se focalise sur la définition des activités de reconstruction : où les processus, les techniques et les vues peuvent être combinées de différentes manières en fonction des attentes des différents utilisateurs. Nous allons proposer l'approche BeeEye dans le chapitre suivant.

Deuxième partie

L'approche proposée

Chapitre 4

La construction des vues par BeeEye

Nous avons présenté, dans les deux chapitres précédents (chapitres 2 et 3), les concepts importants de l'architecture logicielle ainsi ses impacts sur le cycle de vie d'un logiciel. Nous avons également montré la nécessité d'extraire l'architecture de certains systèmes. Notre étude des travaux d'extraction a ensuite mis en lumière certaines nécessités dans les approches d'extraction existantes.

Ces nécessités motivent nos travaux et permettent de proposer notre approche BeeEye. Le contexte de BeeEye, comme certaines approches présentées dans le chapitre 3, repose sur une démarche de construction de multiples vues architecturales à partir d'un système logiciel existant.

Pour décrire l'approche BeeEye, nous en présentons d'abord les principes généraux. Ensuite nous présentons le framework BeeEye et les principales caractéristiques de ce framework pour l'ingénierie de construction de vues architecturales.

Nous présentons également le méta-modèle du framework BeeEye supportant l'approche BeeEye et les mécanismes de construction de vues architecturales.

4.1 BeeEye : une approche permettant la construction de vues architecturales

La reconstruction de l'architecture logicielle est une approche de rétro-ingénierie pour reconstruire des vues architecturales à partir d'un système logiciel existant [DP09]. Partant des constats faits sur les limites des approches existantes d'extraction des vues architecturales, nous proposons une approche appelée BeeEye. Cette dernière propose une démarche d'ingénierie pour construire des vues architecturales à base de point de vue. Ces vues sont constituées d'éléments architecturaux et de relations architecturales pour un système logiciel existant dont le code source est disponible.

4.1.1 Principales caractéristiques de BeeEye

Nous allons présenter la contribution de l'approche proposée :

- proposition d'une démarche générique pour l'ingénierie de la construction des vues architecturales basée sur des points de vue ;
- permet de construire des vues architecturale liées ;
- prise en compte des connaissances des utilisateurs ;

- prise en compte et distinction entre l'architecture existante ("as-implemented") et l'architecture supposée ("as-intended") ;
- processus de construction de vues architecturales qui repose sur des enchaînements de vues (en cascade) ;
- ensemble des vues architecturales non-figé ;
- mécanisme de définition de nouveaux points de vue (flexibilité).

Les apports de ces caractéristiques

Nous nous focaliserons sur chacune de ces caractéristiques de BeeEye par la suite, afin de clarifier nos contributions.

Proposition d'une démarche générique pour l'ingénierie de la construction des vues architecturales basée sur des points de vue

Il existe dans la littérature des approches qui se focalisent sur la construction des vues architecturales. Certaines de ces approches utilisent aussi des points de vue. Notre objectif est de proposer une fondation générique pour l'ingénierie de la construction des vues architecturales en se basant sur des points de vue. Cette approche prend en compte les deux concepts vue et point de vue.

Construction de vues architecturales liées

BeeEye repose sur la construction de l'architecture à base de vues explicites. Elle fournit des moyens pour construire des vues situées à des niveaux d'abstraction différents. L'approche permet d'établir des liens entre des vues construites en mettant des relations entre ces vues.

Niveau du code source et proche du code source

Face à la diversité des langages à base d'objets, il est nécessaire de proposer une approche suffisamment générique pour pouvoir être appliquée à tous les systèmes à objets. Le code source d'un système logiciel n'est pas facile à comprendre pour un utilisateur, c'est à la fois difficile manuellement et à la fois prend beaucoup de temps et de ressources.

Notre approche commence la construction des vues représentant le système étudié à partir d'une vue au niveau code source. L'approche a pour objectif de construire les vues génériques contenant les principales entités (éléments et relations) à partir de la vue code source. Ces entités sont principalement les paquetages, les classes, les méthodes ainsi que les relations entre les classes (telle que l'héritage) et entre les méthodes (telle que l'invocation de méthode).

Vers des architectures abstraites

L'approche prend en compte ces vues construites à partir du code source afin d'obtenir des vues architecturales de niveau plus abstrait que le code source (niveau implémentation). L'objectif initial a été de se rapprocher du paradigme architectural contenant des éléments architecturaux et des relations architecturales comme composants-connecteurs. Cependant des entités construites par BeeEye

ne représentent pas les mêmes objectifs que composants-connecteurs. Contrairement à ces dernières qui se focalisent sur une décomposition fonctionnelle, BeeEye n'impose pas un critère opérationnel des entités des vues construites. Elle se focalise plutôt sur le domaine d'application du système et des principes d'ingénierie logicielle.

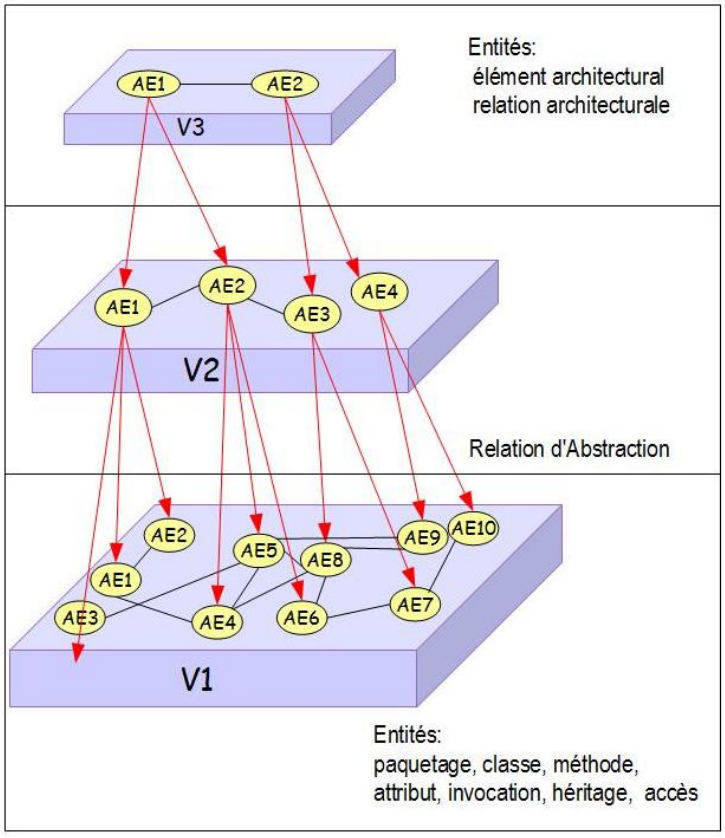

FIGURE 4.1 – Des vues interconnectées aux différentes niveaux d'abstraction

La mise en relation des vues

Au cours de la construction d'une vue à partir d'une autre, BeeEye effectue une mise en relation entre les deux vues. Dans notre approche, la vue architecturale située à un niveau d'abstraction *n* a une ou plusieurs relations vers la vue située au niveau d'abstraction *n-1*. Cela s'applique entre deux vues situées à des niveaux d'abstraction différents. Ces relations sont appelées des *relations d'abstraction*. Ces relations ont pour objectif d'établir des liens entre les deux vues (cf. la figure 4.1). Nous allons également présenter d'autres types de relations entre des vues dans la section 4.6.1.

Prise en compte des connaissance des utilisateurs

Par utilisateurs, nous comprenons le groupe d'acteurs qui travaillent sur un système : les ingénieurs du système, les personnes effectuant des essais sur le système, ou les personnes effectuant le développement. Ces groupes d'utilisateurs peuvent apporter des modifications sur un logiciel existant au fil du temps. Afin d'améliorer la compréhension du système en effectuant ces tâches au cours des phases la maintenance et l'évolution, l'approche BeeEye s'appuie sur l'interactivité. Elle permet à l'utilisateur de proposer ses attentes et propositions génériques (fournies par différentes sources d'information) afin de construire des vues architecturales. L'utilisateur, donc, peut décider de la façon

de proposer ces informations, la vue étudiée et la technique choisie pour la construction des vues.

Prise en compte de la distinction entre l'architecture existante ("as-implemented") et l'architecture supposée ("as-intended")
BeeEye prend en charge des vues architecturales construites à partir d'un système existant. Ces vues représentent l'architecture implémentée (as-implemented) du système. Afin de réaliser cet objectif, BeeEye commence la construction à partir du code source. Les vues construites sont appelées les vues concrètes. Ainsi l'approche considère des attentes et des propositions (génériques) par l'utilisateur comme l'architecture attendue (as-intended). Dans les sections 4.3.1 et 4.3.2, nous les détaillons.

Processus de construction de vues architecturales qui repose sur des enchaînements de vues (en cascade)
BeeEye permet d'enchaîner les étapes de construction de vues et de construire des vues architecturales à partir d'autres vues architecturales préalablement construites.

Ensemble des vues architecturales non-figées
BeeEye est une approche permettant de construire un ensemble des vues architecturale non-figées. C'est-à-dire l'approche propose une démarche d'ingénierie afin de construire des vues et de manipuler ces vues. En effet, plusieurs critères génériques sont considérés afin de construire des vues architecturales sans imposer des critères particuliers fixes, ou sans imposer d'avoir un catalogue des vues prédéfinies.

Mécanisme de définition de nouveaux points de vue (flexibilité)
BeeEye est une approche *flexible*. Elle permet de choisir la vue étudiée, de proposer et définir des attentes en forme de point de vue, de choisir les techniques de construction afin de construire des nouvelles vues architecturales. Le cadre de travail proposé permet de considérer chaque détail nécessaire selon les utilisateurs. Ainsi l'approche permet de construire des nouveaux points de vue.

L'approche est concrétisée par un framework que nous proposons : le framework BeeEye. Ce dernier définit un certain nombre de concepts liés aux caractéristiques données. Par la suite, nous montrons comment ces caractéristiques sont concrétisées dans le framework. Dans la suite nous présentons le framework BeeEye.

4.2 Le framework BeeEye

Dans cette section, nous allons détailler les caractéristiques de l'approche BeeEye et nous verrons comment elles sont prises en compte dans le framework.

4.2.1 Vue générale du framework BeeEye

Comme mentionné dans la section précédente, BeeEye permet de construire plusieurs vues architecturales. Le framework BeeEye repose sur trois notions principales :

- une vue étudiée : c'est une vue concrète en entrée représentant le système logiciel à objets étudié. L'objectif est de construire une ou plusieurs vues architecturales à partir de cette vue

donnée ;
- un point de vue : un point de vue consiste en un ensemble de propositions et attentes d'un utilisateur plus des algorithmes de construction des vues. En effet, le point de vue guide la construction d'une vue concrète à partir d'une autre vue concrète ;
- une vue construite : c'est une vue concrète construite par rapport aux perspectives données dans le point de vue et à partir de la vue étudiée. Cette vue donne une autre représentation du système selon le point de vue utilisé.

La vue issue de chaque construction peut être ensuite utilisée comme entrée lors d'une autre construction.

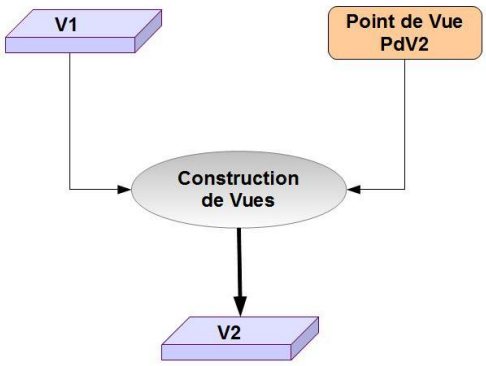

FIGURE 4.2 – Une itération du Framework BeeEye

La figure 4.2 montre une image générale de l'approche. Commençant à partir de la vue concrète étudiée V1 et du point de vue PdV2, le framework considère les attentes et les propositions du point de vue afin de construire la vue concrète V2. Dans cette section nous ne détaillerons pas l'expression d'un point de vue ; cela fera l'objet de la section (4.3.3).

Nous formalisons la construction d'une vue architecturale à partir d'une autre utilisant un point de vue comme suite :

$$Vue\ Construite = Application(PdV2, Vue\ etudie) \tag{4.1}$$

Par la suite nous précisons les informations utilisées par le framework.

4.2.2 Informations utilisées : Faits et Connaissance experte

Dans la construction des vues architecturales, le framework BeeEye considère deux aspects : *les faits* et *l'expertise*. A partir des faits existants le (système et son code source) et selon les expertises (attentes et propositions), on va construire d'autres faits. Ces derniers sont des vues construites représentant ce système. A chaque construction d'une vue architecturale, l'approche permet à l'utilisateur de proposer les informations qu'il possède ou qu'il pense posséder. Cette solution est valable autant pour la conception que pour la maintenance de systèmes logiciels complexes.

BeeEye prend en compte les deux cas de figures suivants concernant la façon de considérer de l'information :

- l'utilisateur propose et définit ses attentes et informations par une vue abstraite. C'est le cadre définissant une construction de vues par *correspondance*. Cette vue est fournie, donc, à partir

de la connaissance d'utilisateurs. Cette catégorie de points de vue est présentée dans la section (4.3.3).
- l'utilisateur propose et définit ses attentes et souhaits par certaines caractéristiques génériques. Ces caractéristiques révèlent des critères comme "découverte des similarités entre des éléments" ou "découverte des interactions entre des éléments" dans un système étudié afin d'être représentées dans la vue construite. C'est le cadre définissant une construction de vues par *découverte*. Cela se fait en l'absence d'une connaissance particulière de la part de l'utilisateur sur le système étudié. Cette catégorie de points de vue est présentée dans la section (4.3.3).

Par la suite, nous allons donner la définition de la vue architecturale, ainsi des différents types des vues proposées et les points de vue.

4.3 Vues architecturales et points de vue

La définition de la vue par le standard IEEE est :

A view is a representation of a whole system from the perspectives of a related set of concerns [IEE00].

Nous utilisons également cette définition donnée pour les vues architecturales de BeeEye. La construction d'une vue architecturale requiert l'identification des éléments et des relations faisant partie d'une vue. Pour mieux catégoriser les informations concernant un système étudié, ces informations peuvent être structurées en plusieurs facettes, telles que structuration fonctionnelle selon des patrons logiciels (ou architecture supposée), structuration selon les termes du domaine du problème, structuration relationnelle comme les interactions entre les éléments du système *etc.* Chaque facette peut être représentée par une ou plusieurs vues architecturales.

Dans notre approche, nous avons décidé de proposer certaines facettes qui peuvent apporter des informations de différentes natures pour les utilisateurs, ainsi les facettes nous permettant de mieux catégoriser ces informations. Ces facettes proposées sont :

- conceptuelle : *facette domaine d'application*
- structurelle : *facette patrons logiciels*
- relationnelle : *facette interaction*

Chaque facette proposée couvre un angle d'observation selon lequel un utilisateur perçoit le système étudié. Nous avons utilisé la notion de facette pour pouvoir mieux catégoriser des vues architecturales ainsi que des points de vue utilisés. En effet, une facette peut être décrite par une ou plusieurs vues architecturales. Ces vues architecturales peuvent être : *les vues abstraites* ou *les vues concrètes*.

4.3.1 Vue Abstraite

Les informations ou attentes des utilisateurs reflètent l'architecture supposée ("as-intended") pour un système étudié. Notre approche permet à l'utilisateur de proposer et définir ces informations et attentes par *une vue abstraite*. En effet, *une vue abstraite* donne une représentation possible d'un

système logiciel. Elle représente un modèle possible du système avec des éléments considérés comme pertinents selon une facette proposée pour le système. Chaque élément de la vue abstraite proposée est censé constituer une abstraction d'une partie du système.

Les sources utilisées pour proposer une vue abstraite peuvent être différentes : l'utilisateur peut avoir une connaissance a priori concernant l'application/le système ou il peut récupérer cette connaissance à partir d'autres sources telles que la documentation existante, les entretiens avec les développeurs originaux ou les opinions des experts disponibles, *etc.*

La figure 4.3 montre une vue abstraite contenant trois éléments architecturaux et des relations entre ces éléments. Le fait de proposer/définir ces entités dans une vue abstraite ne garantit pas leur existence dans le système étudié. Cette vue constitue des attentes de l'utilisateur selon les ressources qui peuvent ne pas être en accord avec le système actuel ou la documentation existante.

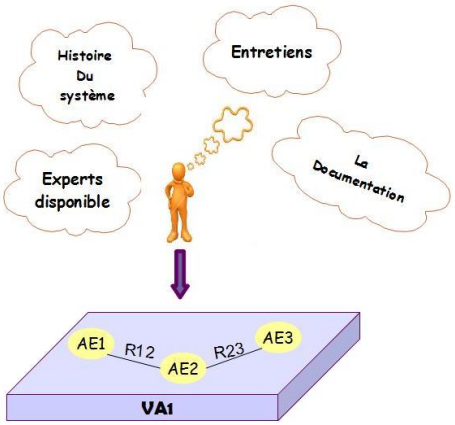

FIGURE 4.3 – Une vue abstraite proposée selon la connaissance de l'utilisateur

Partant de cette vue (VA1), nous supposons que le système étudié est composé par trois éléments architecturaux (VA1.AE1, VA1.AE2 et VA.AE3) et deux relations (VA1.R12, VA1.R23). Cette architecture supposée par l'utilisateur sert de guide pour la construction d'une autre vue concrète.

4.3.2 Vue concrète

Une vue concrète est construite à partir du code source ou d'une autre vue déjà construite représentant le système étudié. Elle fournit une représentation concrète ("as-implemented") d'un système existant. La construction d'une vue concrète se fait toujours à partir d'une autre vue concrète.

Une caractéristique importante d'une vue concrète est d'avoir des relations d'abstraction vers une autre vue concrète. En fait, une vue concrète située au niveau d'abstraction n a des relations d'abstraction vers une autre vue concrète au niveau d'abstraction *n-1*. A l'aide de ces relations d'abstraction, nous pouvons distinguer quel élément ou relation est fourni à partir de quels éléments ou quelles relations. Un élément (ou une relation) construit dans une vue concrète au niveau d'abstraction n, représente l'abstraction d'un ou plusieurs éléments (ou relations) d'une autre vue située au niveau d'abstraction *n-1*. En effet, une vue concrète construite représente une abstraction du système étudié. En conséquence, une vue concrète sans relation d'abstraction est une vue abstraite construite. Nous pouvons donc définir une vue concrète comme suit :

Une vue concrète est constituée par une vue abstraite ainsi que des relations d'abstraction qui la lient avec une autre vue concrète dont elle est issue.

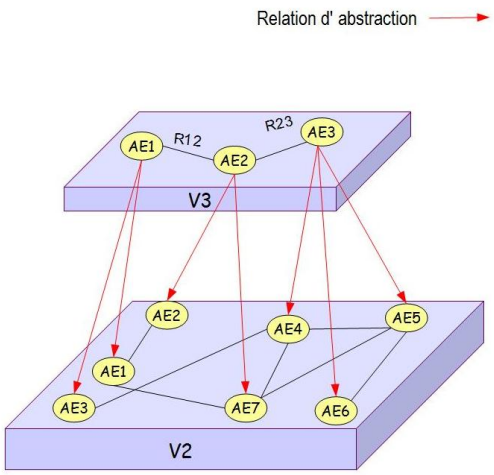

FIGURE 4.4 – V3 : une vue concrète représentant un système étudié

La figure 4.4 montre une vue concrète V3 représentant un système étudié. Cette vue contient trois éléments architecturaux (V3.AE1, V3.AE2, V3.AE3) et deux relations (V3.R12 et V3.R23) entre ces éléments. Les relations d'abstraction lient cette vue à une autre vue architecturale V2. Elles montrent quel élément de la vue V3 est une abstraction de quel(s) élément(s) de la vue V2. Ceci est vrai également pour les relations. Par exemple l'élément V3.AE1 appartenant à la vue construite V3 est une abstraction des éléments V2.AE1 et V2.AE3.

La construction des vues est faite en utilisant des points de vue, que nous allons présenter par la suite.

4.3.3 Points de Vue

Dans notre proposition, chaque construction de vue concrète utilise un point de vue. Dans le chapitre précédent (chapitre 3), nous avons présenté plusieurs approches qui ont utilisé la notion de point de vue dans leur processus de construction d'architecture [Kru95, WCK99, HNS00]. Un point de vue définit comment construire une vue. Il inclut, souvent, un nom, des faits donnés par les utilisateurs, et des conventions de construction de vue. La flexibilité d'utilisation d'un point de vue dans le processus de construction est une notion importante.

Un point de vue est une collection de patrons, modèles et conventions pour la construction d'un type de vue. Les intérêts des utilisateurs, les orientations et les principes de construction des vues sont exprimés dans ce point de vue afin de construire des vues [IEE00].

D'après la définition de la notion de point de vue donnée par IEEE, deux aspects sont importants : le premier aspect englobe des intérêts, des principes et des orientations fondamentales des utilisateurs tandis que le deuxième est la façon de les extraire. Dans un contexte général ces deux aspects permettent d'extraire une ou plusieurs vues concernant le système étudié.

Pour pouvoir proposer un point de vue (en se basant sur cette définition), nous proposons d'ajou-

ter un certain degré de flexibilité à cette définition. L'utilisateur est capable de prendre en compte les points suivants :

- le choix de la vue en entrée pour la construction ;
- le choix sur la façon de proposer des souhaits et des informations (par l'intermédiaire d'une vue abstraite ou par certaines caractéristiques génériques) (voir les sections (4.3.3) et (4.3.3)) ;
- le choix de la technique de construction de vues (voir section 4.6.1).

Nous allons reprendre la définition donnée par IEEE afin de proposer notre définition de point de vue :

Un point de vue en BeeEye comprend les attentes et les propositions d'un utilisateur du framework, exprimées soit par une vue abstraite, soit par une proposition d'une caractéristique générique à rechercher dans le système ; et les algorithmes de construction pour construire des vues architecturales.

Une des notions prises en compte dans notre définition est de séparer deux aspects : les préoccupations fondamentales et les algorithmes de construction de vues [ASHS09]. Ces préoccupations peuvent être des attentes représentées par une vue abstraite ou des attentes choisies en tant que caractéristique générique. Un algorithme de construction de vue sert à construire une vue architecturale selon ces attentes. En effet, ces deux aspects (attentes et algorithme) sont distincts mais liés.

Plusieurs bénéfices sont offerts en séparant ces deux aspects :

- **réutilisabilité** : chacun de ces aspects de point de vue est réutilisable indépendamment de l'autre aspect. Ceci est particulièrement vrai pour les algorithmes de construction de vue. Le même algorithme peut être utilisé avec des vues abstraites différentes. Bien que moins fréquent, l'inverse est également vrai. Ainsi les algorithmes de construction de vue peuvent être changés, tandis que la vue abstraite proposée reste inchangée.
- **accessibilité** et **sécurité** : cette séparation de deux aspects d'un point de vue donne la capacité d'utilisation du framework par des catégories différentes d'utilisateurs avec différents niveaux de connaissance sur un système. Ainsi, certains utilisateurs peuvent définir uniquement la partie vue abstraite, et réutiliser les algorithmes de construction de vue proposés. Ils n'ont pas besoin d'avoir des connaissances particulières sur ces algorithmes. Nous allons voir dans le chapitre 6, comment un algorithme de construction peut être défini.
- **flexibilité** : cette séparation en deux aspects d'un point de vue donne la capacité d'être flexible face à des modifications. L'indépendance entre l'aspect algorithmique et l'aspect attentes facilite les éventuels changements sur chacun.

Cette séparation a également permis de définir deux catégories de points de vue dans notre approche : la catégorie des points de vue *correspondance* et la catégorie des points de vue *découverte*. Par la suite, nous allons présenter ces deux catégories de points de vue.

Catégorie des points de vue correspondance ("mapping")

La première catégorie des points de vue définie par notre approche est la catégorie des points de vue *correspondance*. Un point de vue *correspondance*, comme la définition données pour les points de vue, contient deux aspects :

- les attentes et propositions de l'utilisateur données dans une vue abstraite ;
- un algorithme de construction de vues architecturales.

Les utilisateurs peuvent jouer un rôle important dans les approches de construction de l'architecture d'un système. L'article IEEE [IEE99] souligne également que les points de vue architecturaux sont dépendants de ce que les utilisateurs perçoivent comme important du système étudié. Chaque utilisateur a un certain niveau de connaissance sur le système étudié. Ces connaissances peuvent être plus ou moins précises en fonction de son expertise. Cependant si l'utilisateur est capable de proposer et définir une vue abstraite selon ses connaissances, notre approche propose d'utiliser la catégorie des points de vue *correspondance*.

Un algorithme de construction d'un point de vue *correspondance* est établi sur les notions suivantes :

- identifier des éléments et relations adéquates selon la vue abstraite proposée dans la vue concrète étudiée.
- construction des éléments et relations pour être représentés dans une vue construite.
- mettre en relation un ou plusieurs éléments et / ou relations identifées de la vue étudiée avec des éléments et relations de la vue construite.

Cet algorithme vise à regrouper des éléments de la vue concrète étudiée. Dans le cas des points de vue *correspondance*, ce regroupement est réalisé en fonction de la similarité textuelle existante entre le nom des éléments de la vue abstraite et le nom des éléments de la vue concrète étudiée. Pour chaque élément proposé dans la vue abstraite, l'algorithme consiste à identifier les éléments de la vue concrète (selon le critère de similarité textuelle) et à les regrouper (lier) au sein d'un même élément architectural de la vue concrète construite.

A travers notre démarche, certaines parties de la vue étudiée vont rester non-identifiées. L'algorithme permet d'identifier tous les éléments architecturaux qui n'ont pas été identifiés avec les critères de correspondance définis par la vue abstraite fournie. L'algorithme construit un élément architectural nommé *Hors-Domaine*. Ce dernier est lié aux éléments non-identifés de la vue étudiée. L'objectif est de permettre d'exploiter des nouveaux éléments architecturaux à partir de cette zone non-identifée. Cela sera utilisé par d'autres processus de construction.

Nous avons défini une vue concrète construite en associant des relations avec une autre vue concrète. L'objectif général d'avoir des relations vers une autre vue concrètre est de préserver les sources à partir des quelles un élément ou une relation est construite. Ainsi, nous pouvons citer les deux raisons suivantes :

- remplir le vide produit entre deux vues : après la construction d'une vue à partir d'une autre, nous avons souvent un manque d'information. Ce manque se résume sur les sources à partir des quelles les contenus des nouvelles vues ont construits. En mettant en relation les entités de deux vues, nous pouvons non seulement préserver ces informations, mais aussi connaître le niveau d'abstraction vis- à-vis de la vue étudiée.
- préciser l'objectif de construction de la nouvelle vue : dans les sections suivantes, nous allons

présenter les différentes techniques de construction d'une vue proposées par BeeEye (à savoir par abstraction, par raffinement, par identité ou par composition). En mettant en relation les vues, nous permettons à l'utilisateur de déterminer la technique utilisée pour construire de chaque vue. En effet, nous avons différents types de relations entre les vues.

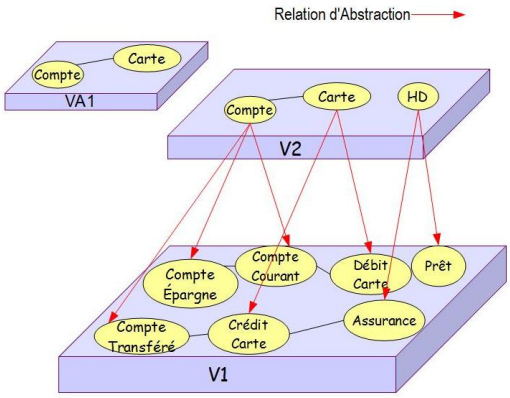

FIGURE 4.5 – Exemple sur la construction d'une vue concrète à partir d'une autre selon un point de vue *correspondance*. La correspondance se réalise entre V1 et V2 selon la vue abstraite VA1.

La figure 4.5 montre une construction utilisant BeeEye selon un point de vue appartenant à la catégorie *correspondance*. La vue abstraite proposée représente un domaine bancaire. Elle inclut des attentes d'un utilisateur à travers deux éléments architecturaux : "carte" et "compte" proposés. BeeEye construit la vue concrète V2 à partir de la vue étudiée V1. Utilisant l'algorithme de construction de point de vue défini, cette vue possède deux éléments architecturaux liés vers des éléments adéquats de la vue étudiée. Nous rappelons que cette identification a été faite selon une comparaison textuelle appliquée entre les noms des éléments. Ainsi, l'algorithme a découvert un certain nombre d'éléments qui ne sont pas proposés dans la vue abstraite, mais pourtant existent dans le système. Ces éléments sont liés vers l'élément *Hors-Domaine* construit. Dans la figure 4.5, cet élément est présenté par le nom "HD".

Catégorie des points de vue découverte ("Discovery")

Il existe de nombreuses raisons pour qu'un utilisateur ne souhaite pas proposer une vue abstraite : il ne possède aucune information sur le système, il souhaite faire une recherche par découverte, ou la connaissance qu'il possède n'est pas suffisante pour identifier tous les éléments de la vue étudiée, *etc.*

Selon la définition donnée pour les points de vue, un point de vue de la catégorie *découverte* contient les deux aspects suivants :

- une caractéristique générique à découvrir dans la vue étudiée, ainsi qu'un seuil comme une métrique de comparaison définie par l'utilisateur.
- un algorithme de construction afin de construire des vues architecturales.

En définissant une caractéristique comme étant un objectif à retrouver et rechercher parmi des entités de la vue étudiée, l'algorithme de construction d'un point de vue *découverte* applique une identification des entités dans cette vue. Cette identification repose donc sur la découverte des éléments respectant la caractéristique recherchée, et parmi ces éléments ceux qui respectent le seuil considéré. L'utilisation des seuils en tant que métriques nous permet d'évaluer et de comparer les résultats obtenus. Les métriques ont été longtemps étudiées comme un moyen d'évaluer la qualité et la complexité des logiciels [FP96]. Le seuil considéré sera utilisé dans l'algorithme de construction de

vues.

L'approche BeeEye propose trois caractéristiques à considérer par la catégorie de points de vue *découverte* :

- exploration basée sur *la similarité textuelle* entre le nom des éléments architecturaux : l'algorithme consiste à identifier des éléments architecturaux répondant au critère de similarité entre leurs noms.
- exploration basée sur *le niveau d'interaction* entre des éléments architecturaux (activité) : l'algorithme consiste à identifier des éléments architecturaux ayant un nombre d'interactions important. Notre définition de l'activité d'un élément architectural est liée au nombre d'interactions que l'élément possède avec d'autres éléments. La nombre d'interactions considérable sera proposé par l'utilisateur en tant que seuil.
- exploration basée sur *la fonctionnalité* des éléments architecturaux : l'algorithme consiste à identifier des éléments architecturaux avec certaines fonctionnalités proposées. Notre définition de la fonctionnalité d'un élément architectural est liée au nombre d'interactions en prenant en compte la direction de cette interaction. C'est-à-dire des messages envoyés et reçus par les éléments liés à cet élément architectural.

En conséquence, l'objectif d'un point de vue découverte est de permettre à l'utilisateur d'explorer de nouveaux éléments ou relations à partir de certaines caractéristiques sans avoir de connaissance sur le système. L'utilisateur propose des caractéristiques génériques qui peuvent être portées sur n'importe quel système (comme la fonctionnalité). Cependant, l'approche a pris en compte les caractéristiques proposées au-dessus.

Après utilisation d'un point de vue appartenant à la catégorie *correspondance*, souvent l'utilisateur applique un point de vue de catégorie *découverte*. Ce dernier permet de découvrir des parties non-identifiées et inconnues de la vue étudiée. Ces parties sont partitionnées par l'élément *Hors-Domaine*. En obtenant des nouveaux éléments architecturaux à partir de l'élément *Hors-Domaine*, l'utilisateur peut avoir une vision plus claire sur ces parties non-identifiées.

Nous allons préciser dans le chapitre suivant (chapitre 5) comment un algorithme de construction permet d'identifier des éléments et des relations adéquats en fonction de ces caractéristiques.

4.4 Le méta-Modèle BeeEye

Le méta-modèle BeeEye repose sur un concept : *vue architecturale*. Ce méta-modèle permet la représentation des vues architecturales (des éléments et des relations) ainsi que des relations entre des vues architecturales. Dans la section suivante, nous allons voir les différentes techniques de construction proposées par notre approche. Ces différentes techniques imposent d'avoir différents types de relations. C'est-à-dire, lors de la construction d'une vue à partir d'une autre (selon une technique), BeeEye construit également des relations appropriées à la technique utilisée entre des éléments et relations de deux vues. Le méta-modèle BeeEye est présenté dans la figure 4.6.

Le méta-modèle BeeEye modèlise les notions suivantes :

- les entités d'une vue architecturale : élément architectural et relation architecturale ;
- les relations qui sont :
  - relation entre deux éléments architecturaux appartenant à la même vue.
  - relation entre deux éléments architecturaux appartenant à deux vues différentes.
  - relation entre deux relations appartenant à deux vues différentes.

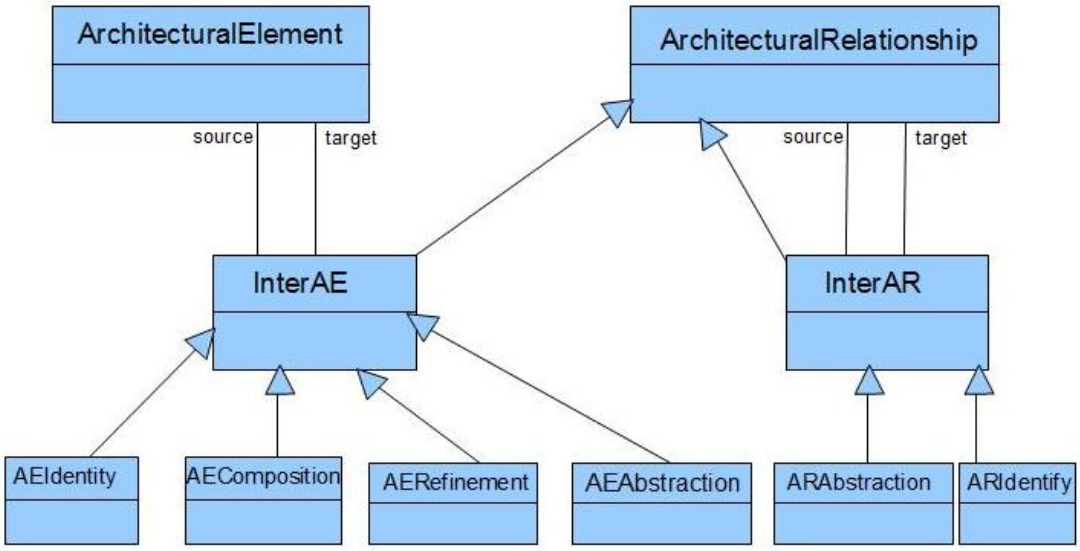

FIGURE 4.6 – Le méta-modèle du framework BeeEye

Des relations modélisées par le méta-modèle sont des relations architecturales. Ces relations sont soit de type *InterAE*, soit de type *InterAR*. Nous allons voir la différence entre ces deux types :

InterAE (inter-éléments architecturaux)

Une relation de type *InterAE* permet de lier deux éléments architecturaux. Deux cas de figures existent :

- soit cette relation s'établit entre deux éléments architecturaux appartenant à la même vue architecturale : ces deux éléments peuvent être proposés et définis dans une vue abstraite par l'utilisateur, ou ils peuvent être construits dans une vue construite par le framework. Cette relation représente l'existence d'une ou plusieurs relations entre des éléments de la vue étudiée.
- soit cette relation s'établit entre deux éléments architecturaux appartenant à deux vues architecturales différentes : ces deux vues peuvent être situées à un niveau d'abstraction différent ou au même niveau d'abstraction.

InterAR (inter-relations architecturales)

Une relation de type *InterAR* permet de lier deux relations :

- soit ces deux relations sont deux relations horizontales (InterAE) associant deux éléments architecturaux.
- soit ces deux relations sont deux relations verticales, associant deux autres relations. Nous en parlons au fur et à mesure dans le chapitre.

L'intérêt d'avoir une relation InterAR est de permettre d'établir des liens entre une relation construite vers une ou plusieurs relations étudiées. Dans le cadre de notre approche, une relation de type *InterAR* peut établir des liens entre deux relations différentes appartenant à deux vues. Ces deux vues se situent à deux niveaux d'abstraction différents (ARAbstraction). C'est pour cela que nous appelons ces relations des relations architecturales d'abstraction. Ainsi, une relation InterAR

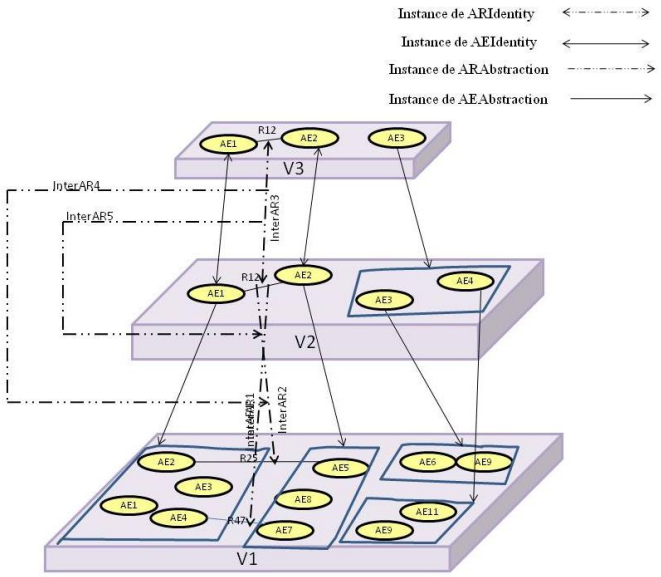

FIGURE 4.7 – Les relations InterAR entre les vues

peut établir des liens entre les relations identiques de deux vues architecturales (ARIdentity). Une relation identique est celle qui appartient à la vue étudiée, ainsi elle appartient à la vue construite. En effet, cette relation n'est pas modifée au cours de la construction de la nouvelle vue (cf. figure 4.7).

4.5 La construction des vues

Dans la littérature, de nombreuses approches ont été proposées pour construire des vues ; par contre il n'y a pas d'accord global sur les vues qui sont généralement nécessaires pour la représentation d'un système logiciel. Quelquefois, les vues proposées par différents auteurs sont en fait très similaires, alors que d'autres auteurs se concentrent sur des vues très spécifiques et variées. Chaque système a son architecture avec ses propres styles architecturaux, ses règles, ses codes sources, *etc.* Comme [PW92] ont noté, même si le système a été conçu conformément aux normes des styles architecturaux, la personnalisation des éléments architecturaux font que le système est une création unique. Chaque organisation, voie même chaque équipe a tendance à développer son propre ensemble de vues qui s'appliquent parfois sur des domaines limités, comme les vues focalisées sur les patrons logiciels.

BeeEye consiste à construire un ensemble de vues architecturales non-figées. La démarche de construction est générique pour n'importe quel système logiciel. En effet, notre approche permet de construire un ensemble des vues selon des attentes d'utilisateurs sans être dépendant du système logiciel existant.

La figure 4.8 montre une construction de vue architecturale : le framework prend en compte la vue concrète V2 en tant que la vue étudiée et le point de vue PdV3 en tant qu'un guide de construction. Ce dernier peut être un point de vue appartenant à la catégorie des points de vue *correspondance* ou à la catégorie des points de vue *découverte*. En appliquant ce point de vue, BeeEye construit la vue concrète V3. Nous remarquons la présence d'une autre vue V1 dans cette figure. Les éléments de la vue V2 sont liés par des relations d'abstraction (InterAE) vers des éléments de la vue V1. Ce qui signifie que la vue V2 est également une vue concrète déjà construite par BeeEye. En conséquence, nous avons un enchaînement entre ces trois vues, où chacune est obtenue selon un point de vue différent.

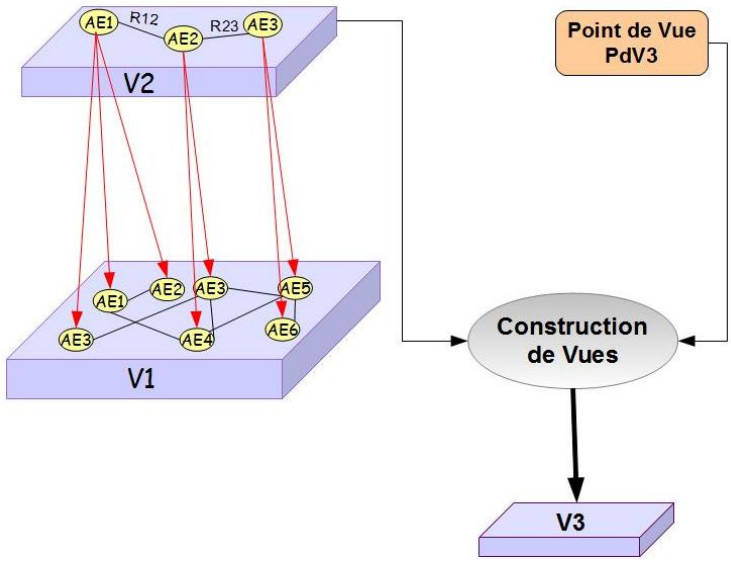

FIGURE 4.8 – Construction de vue concrète (V3) à partir d'une autre vue concrète (V2) selon un point de vue (PdV3)

Notre approche propose les trois différents enchaînements présentés ci-dessous :

- à partir de la même vue concrète, le framework peut construire plusieurs vues concrètes. Chaque construction sera réalisée selon un point de vue différent.
- à partir d'une vue concrète, le framework permet de construire plusieurs vues concrètes. Chaque construction prend en compte la vue construite par la construction précédente en tant que la vue étudiée. Ces enchaînements sont horizontaux. Ces vues construites peuvent être au même niveau d'abstraction.
- Chaque construction prend en compte la vue construite par la construction précédente en tant que la vue étudiée. Par contre ces enchaînements sont verticaux et les vues construites ne seront plus au même niveau d'abstraction.

Dans la suite, nous détaillons précisement les enchaînements horizontaux et verticaux.

4.5.1 Enchaînements des vues construites

Les approches les plus courantes sont des approches qui ont un processus d'extraction séquentiel. Dans ces approches, le processus d'extraction est exécuté une seule fois et il ne prend pas en compte les exécutions précédentes. Afin de raffiner les résultats, ces approches ont la nécessité d'exécuter une nouvelle fois le processus en modifiant les paramètres. Il existe des approches utilisant un processus d'extraction itératif [SAMD08]. Ces approches permettent de réutiliser des résultats des exécutions précédentes. Cette approche est cependant peu utilisée dans les autres travaux existants.

Notre approche utilise un processus de construction de vue itératif. L'utilisateur peut choisir la vue étudiée à chaque construction. Cette fonction permet d'avoir des enchaînements horizontaux et/ou verticaux. L'objectif de proposer une approche itérative flexible est de permettre de réutiliser des vues construites dans chaque nouvelle construction de vues, ainsi que de permettre de raffiner et effectivement d'améliorer les vues obtenues d'une manière interactive avec des utilisateurs. En effet, chaque vue se situe dans un enchaînement vertical et ou horizontal.

L'enchaînement vertical de vues :

Un enchaînement vertical dans la construction de vues architecturales a deux propriétés :

- chaque vue concrète peut être la vue étudiée du framework pour la construction suivante.
- la vue étudiée et la vue construite ne sont pas au même niveau d'abstraction.

En effet, un enchaînement vertical de vues de BeeEye est constitué d'au minimum de deux constructions verticales :

la vue code source vers la vue d'implémentation. La première construction commence à partir du code source considéré comme la vue étudiée. Le framework construit la vue concrète d'implémentation. Cette vue se situe à un niveau d'abstraction plus élevé que la vue étudiée

la vue d'implémentation vers d'autres vues. La deuxième construction commence à partir de la vue d'implémentation. BeeEye construit d'autres vues concrètes selon différents points de vues proposés. Ces vues construites sont à un niveau d'abstraction plus élevé que la vue d'implémentation ou la vue code source.

L'objectif de proposer un enchaînement vertical est d'éloigner des vues construites de paradigme objet et de se rapprocher des vues plus abstraites. Cet enchaînement vertical des vues peut continuer vers des niveaux de plus en plus abstraits selon des points de vue proposés.

L'enchaînement horizontal de vues :

Un enchaînement horizontal des vues architecturales a également deux propriétés :

- il commence à partir des vues concrètes construites à partir de la vue d'implémentation. C'est-à-dire que la vue étudiée du framework est élaborée après deux constructions verticales au minimum ;
- la vue étudiée et la vue construite sont au même niveau d'abstraction.

Effectivement, chaque construction fait référence à un point de vue. L'objectif d'avoir un enchaînement horizontal de vues architecturales est de raffiner ou composer les vues construites. Les constructions des vues par notre approche permettent d'avoir ces enchanînements pour réaliser ces objectifs ; ce qui permet de ne pas recommencer (à chaque fois) à partir de la première vue étudiée.

Les deux enchaînements proposés sont également reconnaissables à partir des relations entre des vues construites. Selon les contraintes considérées à chaque construction, l'approche met en relation des vues construites utilisant différents types de relations (raffinement, composition, abstraction). Nous détaillons ces relations dans la section 4.6.1.

Dans la figure 4.9, nous présentons plusieurs constructions réalisées par le framework. BeeEye construit la vue d'implémentation à partir de la vue code source utilisant un point de vue. Cette vue construite sera la vue étudiée pour plusieurs autres constructions de vues. La figure montre qu'à partir de n'importe vue construite, nous pouvons recommencer une nouvelle construction. A chaque niveau d'abstraction, nous pouvons faire des enchaînements horizontaux ou verticaux. Cette figure représente deux enchaînements des vues construites.

Par la suite, nous présentons les différentes techniques de construction des vues, proposées par BeeEye.

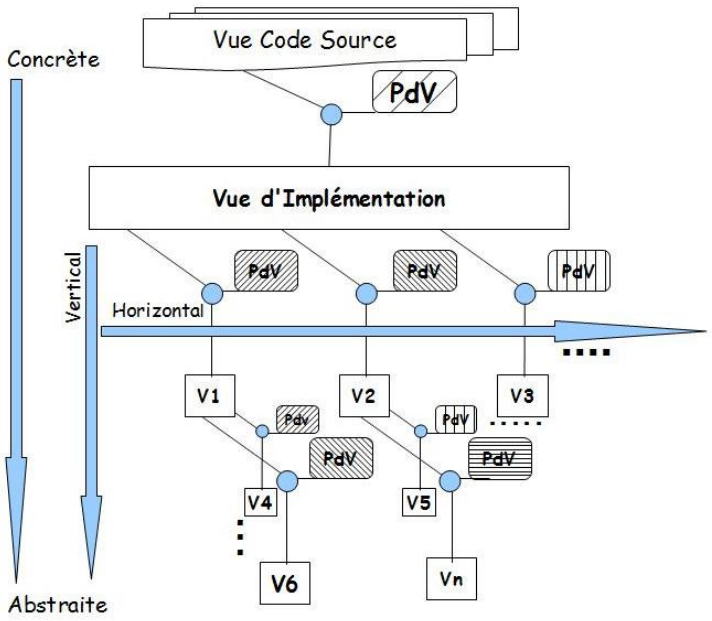

FIGURE 4.9 – Les vues et les points de vue composés

4.6 Les techniques proposées pour construire des vues architecturales

L'approche BeeEye prend en compte les informations d'utilisateurs dans la démarche de construction des vues architecturales. Par contre, les informations fournies ne sont pas toujours positives, telle que le fait que dans une vue construite il existe un élément *Hors-domaine*. Ainsi souvent, l'utilisateur souhaite raffiner la vue obtenue afin d'avoir des informations plus détaillées, composer la vue obtenue avec une autre, ou construire une vue plus abstraite. Pour ces raisons, nous mettons à la disposition de l'utilisateur différentes techniques de construction de vues. Par la suite, nous allons voir en détail chaque technique de construction proposée par BeeEye.

4.6.1 La construction des vues architecturales par abstraction

Pour obtenir des vues architecturales à partir de code source, BeeEye propose d'utiliser la technique d'abstraction. Nous souhaitons pouvoir aller au delà du paradigme objet, en représentant les vues dans un autre paradigme employant des concepts plus abstraits (comme des éléments architecturaux et des relations architecturales).

Avant d'aller plus loin nous allons nous attarder sur la signification de la notion d'abstraction pour mieux clarifier le concept de "haut niveau d'abstraction" dans notre approche.

La construction de vues à divers niveaux d'abstraction est une technique largement utilisée dans le domaine des architectures logicielles pour isoler les aspects technologiques et réduire la complexité des logiciels. Certains faits ne sont pas visibles dans l'implémentation d'un système, alors que ces informations cachées, comme présentées dans [Par72], ont un rôle important dans la compréhension de système. Par exemple, un ensemble d'éléments partagés mais liés par une similarité quelconque peuvent être représentés sous forme d'un nouvel élément ; cet élément englobe la similarité du tout. Une relation architecturale présentée dans une vue concrète construite est obtenue à partir d'un en-

semble de relations existantes dans le code source du système. Si la relation architecturale ne devient pas explicite, mais reste noyée au niveau du code, la compréhension du système au niveau relationnel entre des éléments ne sera pas facile. Plus les systèmes sont complexes, plus il est difficile pour une personne (ou groupe de personnes) de connaître la globalité de système.

La technique d'abstraction permet, dans plupart du temps, la construction d'une vue se situant à un niveau plus élevé. Des éléments de deux vues architecturales se situant à deux niveaux d'abstraction différents sont liés par des relations d'abstraction. Ces relations sont de type *AEAbstraction* présentées dans le méta-modèle BeeEye. Ce type de relation permet d'établir des liens entre des éléments étudiés et des éléments construits afin de savoir à partir de quels éléments ils sont construits.

Dans le cadre de notre approche, la seule particularité d'avoir des relations d'abstraction qui ne représentent pas un changement de niveau. La vue code source représente une vue concrète du système, donc (selon la définition de la vue concrète) elle doit avoir des relations d'abstraction. A ce titre, des éléments et des relations appartenant à cette vue ont des relations d'abstraction vers eux même. Dans ce cas là, des relations d'abstraction sont implicites.

La technique d'abstraction est utilisée pour la construction des vues architecturales présentées dans la figure 4.10. Dans cet enchaînement vertical, nous avons trois vues architecturales. Chacune est construite en utilisant un point de vue. Les éléments de la vue V2 abstraient des éléments de la vue V1. Ainsi, des éléments de la vue V3 abstraient des éléments de la vue V2.

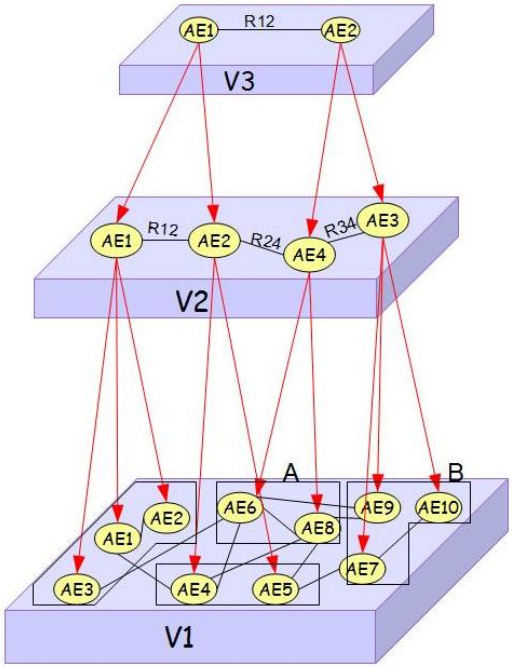

FIGURE 4.10 – Construction par abstraction partie éléments

Dans un enchaînement vertical (comme celui présenté dans la figure 4.10), le point de vue utilisé pour chaque construction peut appartenir à n'importe quelle catégorie proposée (*correspondance* ou *découverte*). La vue V3 construite est une vue abstraite représentant à la fois le système étudié, à la fois une abstraction de la vue V1 et à la fois une abstraction de la vue V2. Les relations d'abstraction associées à V3 nous permettent l'accès aux éléments liés à chaque niveau d'abstraction, c'est-à-dire des éléments qui appartiennent à V2 (directe) ainsi que des éléments qui appartiennent à V1

(indirecte).

Nous avons ainsi des relations entre des éléments architecturaux d'une vue. Ces relations sont de type InterAE (inter-éléments). Chacune de ces relations est obtenue à partir d'une ou plusieurs relations appartenant à une vue se situant à un niveau inférieur. Dans la figure 4.10, nous avons les deux éléments V2.AE3 et V2.AE4. Chacun de ces éléments est lié à un ensemble d'éléments de la vue V1. Il existe également des relations entre ces deux ensembles d'éléments. Effectivement, ces relations entre des ensembles doivent se représenter dans la vue abstraite construite afin de montrer l'existence d'interactions entre des éléments comme V2.AE3 et V2.AE4. La relation V2.R34, donc, représente une abstraction de ces relations entre les deux ensembles A et B.

La relation construite est liée par des relations d'abstraction de type ARAbstraction (inter-relation) vers des relations appartenant à la vue se situant à un niveau inférieur (entre A et B). En effet, ces relations construites lient des relations horizontales entre des éléments.

Chaque relation construite par la technique d'abstraction a une métrique décrivant son poids. Le *poids* montre le nombre des relations liées, appartenant à la vue vue située à un niveau inférieur à partir desquelles une relation est construite. La métrique de poids pour V2.R34 est égale à 2. Cette métrique est un outil afin de connaître le niveau de la cohésion entre deux éléments construits.

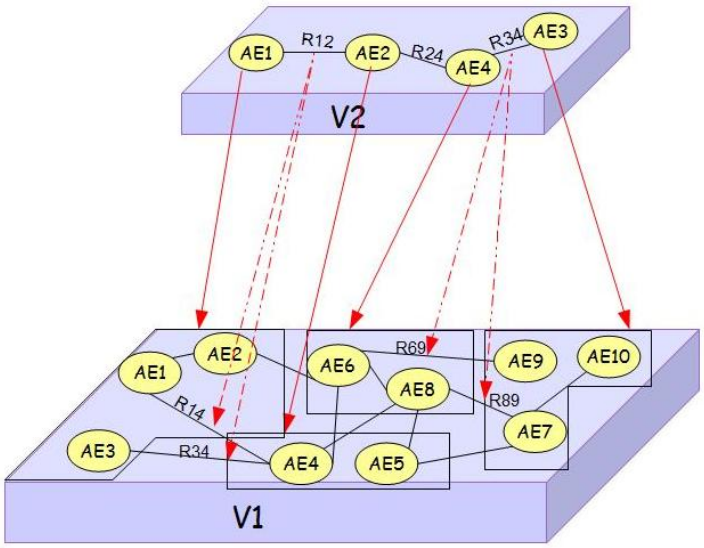

FIGURE 4.11 – Construction par abstraction partie relations

La construction des vues architecturales par raffinement

Nous permettons à l'utilisateur de raffiner une vue construite afin d'affiner un ou plusieurs éléments appartenant à cette vue. Plusieurs points peuvent être les raisons pour lesquelles un utilisateur vise à utiliser la technique de raffinement, tels que :

- la vue étudiée possède un élément architectural *Hors-Domaine* : cet élément est lié vers un groupe d'éléments architecturaux d'une autre vue située à niveau d'abstraction inférieure. L'élément *Hors-Domaine* ne montre qu'une partie non-identifiée du système selon les connaissances appliquées dans sa construction. Cependant, cet élément peut être raffiné en utilisant des nouveaux points de vue par la technique de raffinement.

- la vue en entrée possède des *éléments importants* : un élément *important* est un élément identifié qui couvre une grande partie de la vue étudiée. Cet élément est lié vers plusieurs éléments d'une autre vue concrète située à un niveau d'abstraction inférieure. Ces éléments liés peuvent (éventuellement) être raffinés en utilisant la technique de raffinement.
- chaque élément appartenant à une vue concrète peut être raffiné selon le souhait de l'utilisateur.

Une contrainte à respecter dans une construction par la technique de raffinement est : *le point de vue proposé pour une construction par raffinement à partir de V1 doit représenter la même facette que le point de vue utilisé pour la construction de V1.* Nous rappelons que la facette utilisée représente les perspectives considérées pour la construction de la vue. Ces perspectives restent les mêmes mais les analyses deviennent plus approfondies.

Dans la figure 4.12, nous montrons la construction de la vue V3 à partir de la vue V2 utilisant la technique de raffinement. Pour cela, nous appliquons le point de vue PdV3 sur la vue V2, en focalisant sur l'élément V2.AE1 :

$$Vue3 = Application\ (PdV3,\ V2,\ Raffinement, V2.AE1) \tag{4.2}$$

Cet élément choisi est une portée ("scope"). L'utilisateur peut préciser un ou plusieurs éléments d'une vue comme des portées afin d'être appliquées dans un nouveau processus de construction. L'utilisation du framework nous a permis de construire la vue architecturale V3 qui possède deux nouveaux éléments architecturaux, ainsi que des éléments qui ne font pas partie de la portée.

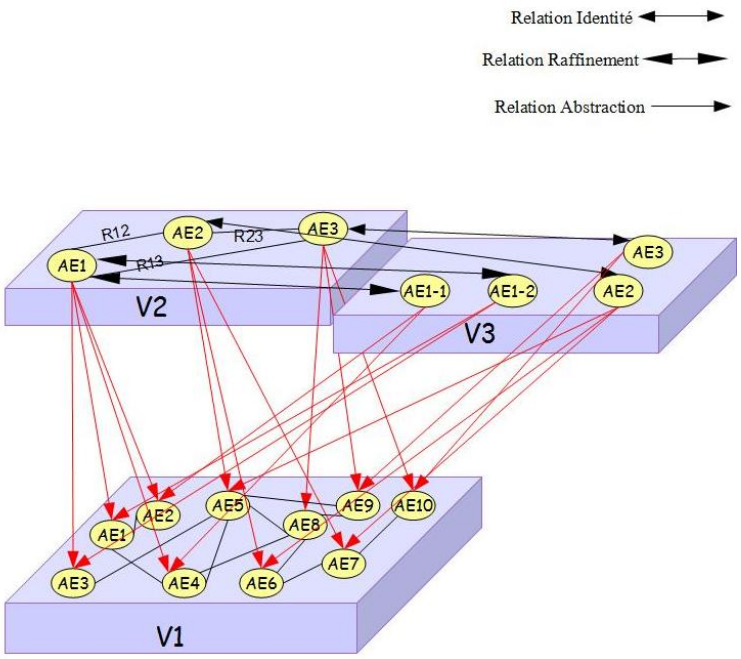

FIGURE 4.12 – Construction par technique raffinement

Le nouveau point de vue peut faire partie des points de vue de la catégorie *correspondance* ou des points de vue de la catégorie *découverte*. L'algorithme de construction de PdV3 vise à construire de nouveaux éléments à partir des éléments liés à V2.AE1 dans la vue V1. Il est possible d'utiliser des liens d'abstraction entre les deux vues. En effet, l'algorithme prend en compte l'élément V2.AE1 en tant que sous-système à traiter. L'objectif est d'améliorer la connaissance sur cet élément choisi.

En réalisant le raffinement de cet élément, le framework permet de construire une vue architecturale plus détaillée que la vue étudiée V2. Cette vue est tout simplement, une autre représentation

de V1 mais plus détaillée. Ainsi cette vue construite est une abstraction de la vue V1.

Les différentes relations présentées dans une construction par raffinement (montrées dans la figure 4.12) sont :

- *La relation identité* : BeeEye met en relation par des relations de type *AEIdentity* les éléments de deux vues qui ne sont pas modifiés au cours de la construction par raffinement. Une relation identité montre les éléments qui soit ne sont pas considérés dans les éléments étudiés, soit ne sont pas modifiés au cours du raffinement. Ce qui montre que de nouveaux critères visés dans le nouveau point de vue n'ont pas eu d'impact sur ces éléments.
- *La relation raffinement* : les relations raffinement sont de type *AERefinement*. L'élément "portée" (scope) est lié par des relations de type *AERefinement* vers des nouveaux éléments construits. L'objectif est d'établir des liens entre des nouveaux éléments et l'élément à partir duquel ils sont construits.
- *La relation d'abstraction* :
  - les éléments de deux vues qui sont liées par des relations d'identité abstraient les mêmes éléments. Par exemple, l'élément V3.AE3 abstrait les mêmes éléments de V1 que l'élément V2.AE3. Les relations d'abstraction, donc, sont obtenues par transitivité.
  - les nouveaux éléments construits sont liés par des relations d'abstraction vers les éléments appropriés de la vue V1.

La construction par raffinement permet d'avoir des enchaînements horizontaux. Cet enchaînement peut continuer horizontalement en utilisant différents points de vue. Ainsi l'utilisateur peut choisir n'importe vue construite (au cours de ces constructions) afin de continuer la construction d'une manière verticale. Dans ce cas l'objectif est d'élever le niveau d'abstraction d'une vue construite raffinée.

La construction des vues architecturales par composition

Cette technique permet de composer différentes perspectives (à savoir *la facette conceptuelle*, *la facette structurelle* et *la facette relationnelle*). Chaque point de vue utilisé dans un processus de construction par composition peut présenter une de ces facettes. Afin de construire une vue architecturale représentant deux perspectives différentes, nous pouvons appliquer la technique de composition. Par exemple, la composition entre les deux facettes conceptuelle et relationnelle donne une vue architecturale contenant des éléments architecturaux portant des propriétés issues de ces deux facettes.

Afin de pouvoir combiner deux facettes, l'approche commence à partir d'une vue concrète construite selon un point de vue représentant la première facette. Ensuite cette vue sera l'entrée d'une nouvelle construction utilisant un point de vue représentant la deuxième facette. Les points de vue peuvent appartenir à n'importe quelle catégorie de points de vue définis.

Un exemple intéressant consist à composer les deux points de vue Domaine d'application et Patron logiciel. La première construction construit des éléments architecturaux et des relations selon les attentes d'un point de vue représentant le Domaine d'application. C'est-à-dire que ces éléments représentent des concepts existant dans le système, ensuite la deuxième utilisation du framework considère cette vue comme la vue étudiée. Chaque élément représente donc un sous-système afin d'être étudié par le nouveau point de vue. L'objectif de cette composition est de mettre en valeur l'organisation d'un patron proposé (le second point de vue) sur une vue représentant le système selon un domaine (le premier point de vue). Cette vue obtenue par une construction successive horizontale permet de savoir : est-ce que chaque élément construit montrant un concept existant dans le domaine

du système respecte également des attentes d'un patron demandé ? C'est-à-dire existe-il une correlation entre un concept de domaine et l'architecture d'un patron. Nous pouvons en déduire si tous les éléments représentant le patron demandé, se retrouvent à l'intérieur d'un élément architectural étudié.

En outre, BeeEye permet de composer n'importe quel point de vue dans un enchaînement horizontal proposé par un utilisateur.

Dans la figure 4.13, nous montrons une construction utilisant la technique de composition. La vue V2 est une abstraction de la vue V1, représentant un système étudié. Le framework vise à construire la vue concrète V3 à partir de la vue étudiée V2 en utilisant la technique de composition. Cependant, le point de vue utilisé doit représenter une facette différente de celle utilisée pour la construction de V2.

$$V3 = Application\ (PdV3\ V2,\ Composition,\ V2.AE1) \tag{4.3}$$

Dans cet exemple, nous avons choisi d'appliquer cette technique sur un élément portée (scope) V2.AE1. Contrairement à la technique de raffinement, la vue construite en utilisant une technique de composition possède également l'élément portée considéré. Puisque l'objectif de composition est d'avoir une vue représentant les deux points de vue, cet élément révèle les perspectives de deux points de vue appliqués sur la vue étudiée.

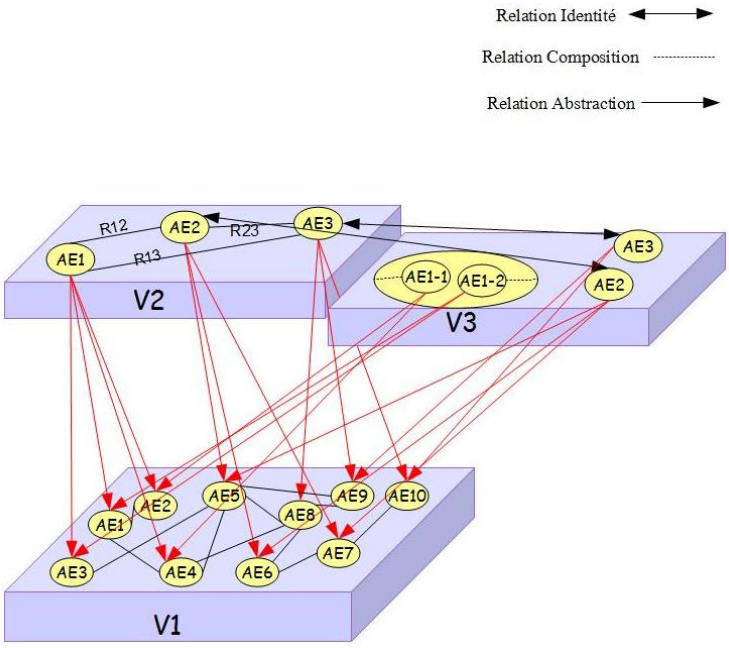

FIGURE 4.13 – Construction par la technique composition.

La vue V3 construite possède les nouveaux éléments V3.AE1-1 et V3.AE1-2 appartenant à l'élément V3.AE1 (la portée de V2). Les relations présentées dans une construction par composition (montrées dans la figure 4.13) sont :

- *La relation identité* : comme dans le cas de la technique de raffinement, cette relation lie des éléments identiques de deux vues au même niveau d'abstraction. Contrairement au raffinement, les deux vues sont identiques au niveau des éléments, donc nous aurons des relations d'identité entre tous les éléments des deux vues.
- *La relation composition* : des relations de composition sont de type *AEComposition*. Elles permettent d'établir des liens de type "appartenant à " entre l'élément portée et de nouveaux

éléments ajoutés à la vue. Dans l'exemple donné (figure 4.13), nous avons deux relations de composition entre l'élément V3.AE1 et les éléments V3.AE1-1 et V3.AE1-2. Nous remarquons qu'une relation de composition est présente dans toutes les vues de manière implicite. Une vue peut être considérée comme un élément architectural composite qui possède d'autres éléments. Ces éléments sont liés à la vue par des relations de composition.

- *La relation d'abstraction* : l'approche construit des relations d'abstraction de V3 à partir des relations d'abstraction des éléments de la vue V2. Les éléments de deux vues, liés par les relations d'identité, vont abstraire les mêmes éléments appartenant à la vue inférieure. Par exemple, l'élément V3.AE2 abstrait les mêmes éléments de V1 que l'élément V2.AE2. Par contre, dans le cas d'élément composite, les nouveaux éléments doivent partager des relations d'abstraction. L'élément V3.AE1-1 est construit à partir de certains éléments de V1 et l'élément V3.AE1-2 est construit à partir des éléments restants.

4.6.2 La construction de la vue d'implémentation

Dans le cadre de notre approche visant à construire des vues architecturale plus abstraites que le code source, l'approche doit permettre de construire la première vue située à un niveau supérieur de la vue code source. Nous appelons cette vue, la *vue d'implémentation*. Cette dernière contient des groupements liés aux entités de la vue code source. Le principe de construction de la vue d'implémentation repose sur une vue représentant le code source, en faisant abstraction de certaines entités que nous ne souhaitons pas prendre en compte.

Nous respectons le cadre général des approches de rétro-ingénierie, et nous ne souhaitons pas en inventer un nouveau. Nous proposons d'intégrer un outil proposé par une de ces approches existantes pour construire la vue représentant un modèle du code source. Ainsi le choix de ce modèle a été effectué parmi des modèles proposés existants. La communauté de rétro-ingénierie a fourni ses propres techniques pour modéliser des systèmes à objets. Les approches proposées sont focalisées sur le formalisme de transformation entre différents modèles, leur contenu et une bonne technique pour réaliser ces transformations entre modèles. Au sujet du contenu de ces modèles, il y a eu un effort pour créer un ensemble de méta-modèles de différents langages [HWS00][Ferenc 2002], [Tic01, DTD01]. Par exemple, le modèle proposé par [Tic01, DTD01] s'appelle FAMIX. C'est un modèle extensible indépendant des langages de programmation orienté objet.

Le modèle Famix

FAMOOS [DDN02] est un "Framework-based Approach for Mastering Object-Oriented Software Evolution". L'objectif de ce projet est de fournir un support pour l'évolution des logiciels orientés objet. Le projet a fourni différents outils et méthodes permettant d'analyser et de détecter les problèmes de conception et de transformation de ces systèmes pour utiliser des architectures plus flexibles. FAMOOS a conduit au développement d'un modèle fournissant une représentation du code source objet indépendant du langage de programmation. Ce modèle est appelé FAMIX [DTD01]. Ce modèle constitue, donc, une base des échanges d'information sur les systèmes orienté objets.

Les avantages du modèle FAMIX sont nombreux. Nous citons entre autres :

- *extensibilité* : FAMIX est facilement extensible pour incorporer les entités et propriétés spécifiques à chaque langage. Ainsi, plusieurs extensions ont été proposées, pour Smalltalk [Duc01], Java [Tic99], C++ [B9̈9] ou Ada [Neb99].
- *support complet pour les métriques, groupement et clustering et autres méthodes de réingénierie* :

FAMIX contient tous les aspects nécessaires pour l'ensemble des méthodes envisagées dans le cadre de FAMOOS.

Le modèle complet de FAMIX comporte les principales entités du paradigme objet : les paquetages, les classes, les méthodes et les attributs. Il comporte également les principales relations dans le monde des objets : invocation, héritage et accès. En effet, ce modèle représente la vue d'implémentation de notre approche.

La proposition d'avoir une vue indépendante du langage nous permet d'appliquer facilement notre approche à ces langages. Il suffit d'utiliser un outil permettant de construire cette vue. Dans le cadre de notre approche, nous utilisons l'outil MOOSE.

MOOSE

MOOSE est une plate-forme d'analyse de données et de logiciels qui offre une infrastructure commune pour plusieurs outils de re-ingénierie et rétro-ingénierie [NDG05, DGLD05, DGKR09]. MOOSE est un méta-environment permettant la manipulation de modèles décrits par une famille extensible de méta-modèles [DGKR09]. Autour de ce noyau sont fournis divers services disponibles dans les différents outils. Ces services comprennent les métriques d'évolution, la visualisation autour de Mondrian [MGL06], un répertoire pour stocker plusieurs modèles, un GUI générique, la navigation, l'interrogation et le groupement. Le lecteur intéressé par les détails sur cet outil peut consulter les articles [NDG05, DGLD05, DGKR09] ainsi que le site web http ://www.moose-technology.org.

Nous avons choisi Moose parmi les autres outils de réingénierie et rétro-ingénierie pour notre approche. Cet outil est approprié pour notre approche car il offre des mécanismes de ré-ingénierie des systèmes existants. Ainsi nous avons pu l'étudier grâce à l'aide des experts de cet outil dans notre groupe de travail. Cela dit cet outil est utilisé complètement indépendemment de notre approche, et rien n'empêche d'utiliser un autre outil.

Par la suite, nous présentons un certain nombre de métriques et principes permettant des comparaisons entre des vues architecturales.

4.7 Etude de vues

Des vues construites par l'approche peuvent être comparées l'une avec l'autre. Par la suite, nous proposons un certain nombre des métriques. Ces dernières peuvent être utilisées par un utilisateur de l'approche afin de comparer et d'analyser des vues.

Poids d'élément et **Poids de relation**

Le poids de chaque élément architectural (relation architecturale) d'une vue concrète est égal au nombre des relations d'abstraction qu'il (elle) possède.

Cette métrique calcule le nombre des éléments (relations) appartenant à la vue étudiée liés à un élément architectural (relation architecturale) construit dans une vue d'un niveau d'abstraction plus élevé. Le poids d'un élément architectural permet de connaître également la propagation du concept présenté par cet élément architectural parmi des éléments du système. En effet, la propagation d'un élément montre quel pourcentage de la vue étudiée est couvert par le concept proposé par l'élément.

Dans la figure 4.14, nous avons quatre éléments architecturaux qui partagent le système étudié en

quatre sous-systèmes (du point de vue sémantique). Nous remarquons que chaque élément de V2 est lié à un ensemble d'éléments de la vue étudiée V1. En effet, ces éléments architecturaux construits montrent l'existence de ces concepts de haut niveau. Ainsi, la propagation de chaque élément est calculée à partir du poids de l'élément construit et du nombre d'éléments de V1. Le diagramme présenté montre la propagation de chaque élément construit au sein de la vue étudiée.

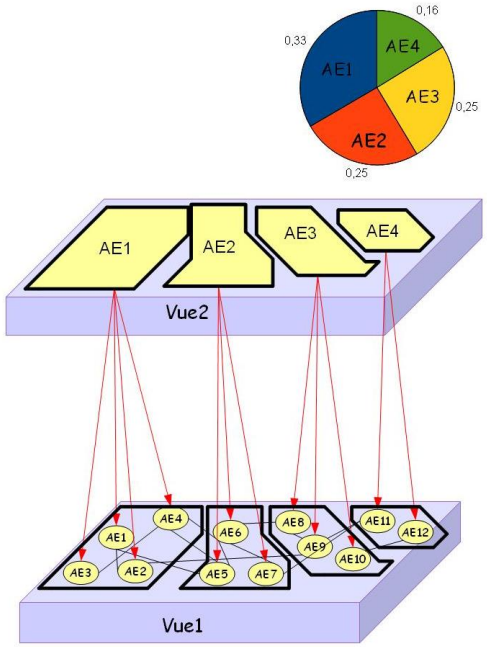

FIGURE 4.14 – La propagation des éléments construite dans une vue architecturale

4.7.1 Comparaisons des vues abstraites

Le point de départ d'une construction par un point de vue de catégorie des points de vue *correspondance* est la proposition et définition de la vue abstraite. Cette vue représente éventuellement une partie de la vue concrète construite. Nous remarquons qu'une vue concrète est une vue abstraite construite ayant des relations d'abstraction. Dans cette circonstance, nous pouvons comparer la vue abstraite proposée et la partie abstraite de la vue concrète construite. Cela présente l'avantage de pouvoir distinguer des éléments non-proposés mais construits (élément *Hors-Domaine*), et des éléments proposés mais non-trouvés (élément *Fantôme*).

Elément Hors-Domaine

Nous avons déjà présenté l'élément *Hors-Domaine*. Nous pouvons souvent constater l'existence d'éléments dans une vue concrète étudiée qui n'ont pas été identifiés avec des critères proposées. Ces éléments sont liés par des relations d'abstraction vers un élément nommé *Hors-Domaine*.

L'existence d'un élément *Hors-Domaine* est due à certaines raisons :

- changement d'orientation du domaine d'application du système : de nouveaux concepts dans un système ont été introduits sans que les informations formelles (documentations) ou informelles (utilisateurs) soient mises à jour .

– il reste des couches de code mort dans le code source : BeeEye est basé sur la vue du code source. Au cours des phases d'évolution et de maintenance, il se peut que certaines parties du code restent inchangées mais elles ne sont plus valables pour de nouvelles orientations du système. Souvent elles n'empêchent pas l'exécution du système, donc on ne peut pas les remarquer sans une attention particulière.

Elément Fantôme

Les informations qui ont permis de proposer et de définir une vue abstraite (expertise) peuvent ne pas être totalement conformes à ce qui existe dans une vue concrète étudiée (faits). La vue abstraite proposée ne garantit pas une concordance totale avec l'état actuel du système étudié.

Un élément proposé dans la vue abstraite qui n'est pas construit lors de démarche de construction est un élément fantôme.

L'existence d'un élément fantôme peut avoir différentes raisons telles que :

– changement d'orientation du domaine d'application du système au fil du temps : cela peut être une raison pour laquelle certains éléments proposés par l'utilisateur ne se trouvent pas dans la vue étudiée.
– manque de sagacité dans le choix des termes (symboles) dans le code source en fonction de leurs comportements, ce qui a un impact direct sur les vues construites à partir du code source. Il faut noter qu'un code de mauvaise qualité réduit la chance d'avoir des vues de bonne qualité sur le système.
– l'existence des éléments correspond à l'élément fantôme dans les couches plus fines de la vue étudiée. Cela nous a fait penser de prévoir des recherches plus attentives par les même algorithmes proposés mais à d'autres niveaux de granularité (par exemple en cherchant au niveau du nom de méthode plutôt qu'au niveau du nom des classes).
– l'utilisateur concerné n'a pas une bonne connaissance sur le système, ce qui l'a amené à proposer des éléments qui ne correspondent pas au système étudié.

4.7.2 Comparaisons des vues concrètes

Toutes les vues construites représentent le même système étudié. Cependant, elle peuvent être comparées et discutées l'une avec l'autre selon plusieurs aspects. Des entités appartenant aux vues concrètes sont liées par différents types de relation : abstraction, raffinement, composition et identité. Chacun de ces types précise la technique utilisée, ainsi elle révèle les raisons pour lesquelles cette technique a été utilisée. Par exemple, lors d'une construction par raffinement, nous supposons que la vue construite ne satisfaisait pas l'utilisateur à cause d'un manque d'affinité (élément *Hors-Domaine* ou élément *important*). En effet, deux vues liées par des relations de raffinement peuvent être comparées du point de vue des nouveaux éléments construits, ainsi que des éléments restés inchangés au cours de raffinement.

Supposons que la vue V1 soit construite selon un point de vue appartenant à la facette Domaine d'application, et la vue V2 soit construite selon un point de vue appartenant à la facette d'Interaction. En comparant ces deux vues, nous pouvons déduire si des éléments représentant une forte dépendance (cohésion) ont tendance à être dans le même concept d'un point de vue sémantique. Le résultat obtenu par cette étude permet aux personnes effectuant les tâches de la maintenance et de l'évolution de système existant de mieux catégoriser des éléments du système.

La comparaison entre des vues construites est intéressante dans une autre circonstance. Le système étudié se modifie au fil du temps, ce qui peut amener des vues différentes selon les mêmes critères. Par exemple, l'utilisateur peut découvrir l'évolution d'un concept particulier au fil de l'évolution de ce système. Cela est particulièrement remarquable lors de l'étude sur l'évolution d'un patron logiciel dans un système. Durant des phases d'évolution d'un système, un patron peut disparaître. Un des concepts appartenant à ce patron peut avoir des tendances évolutives. Par exemple lors d'une étude sur le patron "Modèle-Vue-Contrôleur" sur différentes versions d'un système étudié, nous avons remarqué l'évolution de l'élément "Vue". En effet, de nouvelles orientations dirigent le système vers ce concept particulier.

4.8 Conclusion

Dans ce chapitre, nous avons présenté notre approche d'ingénierie de construction des vues architecturales basées sur des points de vue. Cette approche prend en compte des attentes et souhaits des utilisateurs (sous forme d'une vue abstraite ou de caractéristiques génériques définies) afin de permettre de construire des vues architecturales. Cet aspect permet d'étendre les sources d'informations utilisées par le processus de construction au-delà du code source. Des vues architecturales reflètent ainsi le système dans les attentes d'utilisateurs et non plus simplement son implémentation.

Nous avons d'abord proposé le framework BeeEye permettant de prendre en compte des points de vue (attentes ou caractéristiques génériques plus des algorithmes de construction génériques) et la vue représentant le système étudié afin de construire une autre vue. Ce framework, donc, est établi à la fois sur des faits (le système étudié) et à la fois sur des connaissances (le point de vue).

Nous avons ensuite proposé deux catégories de points de vue : *correspondance* et *découverte*. Ces catégories permettent aux utilisateurs de diriger la construction selon leurs souhaits : soit par une recherche a priori, soit par une recherche exploratoire.

Notre approche permet d'avoir des enchaînements verticaux et horizontaux des vues construites. Nous mettons à disposition de l'utilisateur des techniques de construction afin de fournir des enchaînements. Ceci repose sur les décisions prises par l'utilisateur. L'approche permet de réaliser ces techniques sur n'importe quelle vue choisie et n'importe quel point de vue défini par l'utilisateur.

Chapitre 5

Les primitives définies pour la construction des vues

BeeEye est une approche d'ingénierie pour la construction de vues architecturales basées sur des points de vue. Les vues architecturales permettent de modéliser les attentes et les souhaits des utilisateurs. Ces attentes peuvent être définies soit par une vue abstraite (catégorie des points de vue *correspondance*), soit par des caractéristiques génériques (catégorie des points de vue *découverte*). Des algorithmes de construction de vues sont ainsi proposés selon ces deux catégories de points de vue. L'approche BeeEye propose également différentes techniques de construction telles que les techniques par *abstraction*, par *raffinement* et par *composition*.

Toutes ces notions, qui constituent le cœur de BeeEye, ont été introduites dans le chapitre précédent. Ces notions qui vont être détaillées dans ce chapitre, font partie de l'approche BeeEye et sont importantes pour comprendre les concepts et le fonctionnement de l'approche. Dans ce chapitre, la formalisation et les mécanismes reposant sur ces notions et permettant de construire des vues architecturales vont également être présentés.

La suite du chapitre est organisée de la façon suivante : après avoir rappelé les principales notions de l'approche, nous présenterons les *primitives* de construction des vues architecturales. L'ensemble des primitives est présenté dans deux sous-parties : les primitives dépendant de la catégorie de points de vue ciblée (points de vue *correspondance*, points de vue *découverte*) et les primitives dépendant des techniques de construction de vue ciblée (abstraction, raffinement, composition). Nous présenterons ainsi un certain nombre de primitives pour les opérateurs afin d'être appliquées sur des vues. Ces primitives sont : *union*, *intersection* et *différence*.

L'approche proposée est suffisamment générique pour être utilisée avec tous les systèmes logiciels à objets pour lesquels nous disposons du code source. Notre approche s'inscrit dans une démarche que l'on peut qualifier d'ascendante : les processus de construction de vues commencent par exploiter le code source de l'application à objets et permettent d'obtenir des vues à plusieurs niveaux d'abstraction. Cependant, la démarche que nous proposons permet de prendre en compte des expertises (connaissances et attentes) fournies par l'utilisateur. Les processus de construction de vues commencent par exploiter une vue concrète qui sera fournie comme l'élément d'entrée du framework BeeEyee, ainsi qu'un ensemble de points de vue (éventuellement un seul point de vue). Le framework construira alors une ou plusieurs autres vues concrètes du système.

Chaque algorithme de construction de vues et utilisant un point de vue spécifique est défini en utilisant les primitives proposées. Un algorithme de construction de vue représente l'application du framework BeeEye : admettant une vue concrète en entrée et produisant une autre vue concrète en

sortie.

La plupart des approches de construction d'architectures logicielles à partir d'une application existante utilisent des algorithmes de construction. Ces algorithmes sont classés en plusieurs catégories selon leurs propriétés principales [DP09]. Certaines de ces approches utilisent les algorithmes génétiques [DSS99], d'autres l'exploration avec tabou [SS02], ou les algorithmes de regroupement (clustering) [MM06],[AL99a]. Notre approche diffère des propositions existantes dans la mesure où nous détaillons les mécanismes de construction de vue (concepts, techniques, processus) et la façon de les utiliser. Nous ne nous focalisons pas sur des points de vue particuliers mais proposons une approche ouverte qui permet à l'utilisateur de proposer/définir ses propres points de vue.

5.1 Les primitives : vue d'ensemble

Nous allons dans la section suivante présenter en détail les primitives associées aux points de vue *correspondance*. Ces primitives ont pour objectif de construire une vue concrète à partir d'une autre vue concrète et d'une vue abstraite proposée par l'utilisateur. Les algorithmes de construction des vues sont définis en utilisant (et combinant) éventuellement plusieurs primitives.

Nous présentons également les primitives associées aux points de vue *découverte*. Ces primitives ont pour objectif de construire une vue concrète à partir d'une autre vue concrète selon une caractéristique particulière. Cette caractéristique permet d'identifier les éléments d'une vue architecturale concrète et, éventuellement, de regrouper ces éléments au sein d'une autre vue concrète. Cette caractéristique est fonction d'un critère (similarité des symboles, activité des éléments, fonctionnalité des éléments) que l'utilisateur souhaite identifier dans la vue étudiée et d'un seuil fixé également par l'utilisateur.

Afin de construire une vue concrète en sortie, l'utilisateur doit choisir un point de vue *correspondance* ou un point de vue *découverte*. Il peut préciser une portée (*scope*) qui permet de spécifier sur quels éléments de la vue concrète l'algorithme de construction doit être appliqué.

5.1.1 Primitives de construction par catégories de points de vue

Les primitives définies dans le tableau 5.1 qui permettent de construire des vues architecturales sont présentées en fonction des deux catégories de points de vue proposés (à savoir *correspondance* et *découverte*). Elles sont génériques et peuvent être utilisées quelle que soit la vue concrète étudiée.

Les primitives proposées par BeeEye sont des "briques de base" pour construire des algorithmes de construction des points de vue. Elles sont génériques et peuvent être réutilisées et combinées différemment et dans plusieurs processus et algorithmes de construction de vues ; chaque algorithme assemble deux primitives ou plus selon les critères proposés dans le point de vue. Nous allons voir dans le chapitre suivant (6) comment des points de vue utilisant ces primitives proposées peuvent être construits.

La syntaxe utilisée pour exprimer les primitives est celle du langage de programmation Smalltalk.

Nous allons présenter par la suite les primitives proposées.

	Primitives proposées selon les catégories de points de vue (correspondance ou découverte)
PE1	`mappingBySymbol: abstractView`
PE2	`mappingBySynonym: abstractView and: dictionary`
PE3	`mappingByTree: abstractViewElement and: elementName`
PE4	`notMapping: abstractView`
PE5	`discoveryBySimilarityCharacteristic: threshold`
PE6	`discoveryByActivityCharacteristic: threshold`
PE7	`discoveryByFunctionalityCharacteristic: threshold`
PR1	`mappingRelationship: abstractView`
PR2	`discoveryRelationship: threshold`

TABLE 5.1 – Primitives de BeeEye pour construire des éléments architecturaux et des relations architecturales selon les deux catégories de points de vue.

5.1.2 Primitives pour la construction des éléments architecturaux (PE)

Nous avons sept primitives génériques pour construire des éléments architecturaux d'une vue construite à partir d'une vue étudiée.

Le principe dans le cas des primitives concernant la catégorie des points de vue *correspondance* est qu'à partir de la vue en entrée du framework, la primitive identifie les éléments de la vue concrète étudiée et relie ces éléments avec un ou plusieurs autres éléments d'une autre vue concrète construite par la primitive. Ces relations entre les éléments sont obtenues sur la base des noms des éléments.

Le principe, dans le cas des primitives concernant la catégorie des points de vue *découverte*, est de permettre la découverte des éléments architecturaux à partir des éléments de la vue concrète étudiée. Cette découverte repose sur le critère de similarité textuelle, sur l'activité des éléments ou sur la fonctionnalité des éléments. A partir de la vue concrète étudiée, la primitive vise à identifier des éléments de la vue concrète construite. Elle relie les éléments construits avec un ou plusieurs éléments de la vue concrète étudiée.

PE1 : Mapping By Symbol

Nom de la primitive :

mappingBySymbol: abstractView.

Catégorie : Correspondance.

Description : Cette primitive permet de faire la correspondance entre les éléments de la vue en entrée et les éléments de la vue abstraite proposée et définie par l'utilisateur. La correspondance repose sur un dictionnaire de termes. Cette primitive crée une vue architecturale.

Principe : La primitive utilise un dictionnaire qui est constitué des noms des éléments architecturaux de la vue abstraite définie par l'utilisateur. Pour chaque nom du dictionnaire, un élément architectural portant ce nom est créé dans la vue concrète construite. Si le nom d'un élément architectural de la vue concrète étudiée (en entrée du framework) correspond à un des noms du dictionnaire, une relation (inter-élément) est ainsi créée afin de relier cet élément architectural avec l'élément architectural de la vue concrète construite. Un élément architectural de la vue concrète construite peut

être relié à plusieurs éléments architecturaux de la vue étudiée. Chaque relation inter-élément est de type InterAE (voir le méta-modèle section 4.4). L'algorithme de la primitive PE1 est le suivant :

```
mappingBySymbol: abstractView
  | ae r |
  outputView := View new.
  abstractView elements do: [ :each1 |
       self inputView elements do: [ :each2 |
       (each1 name match: each2 name)
         ifTrue: [ outputView do: [:eachAE |
                     (eachAE name match: each1 name)
                       ifFalse: [outputView elements add:
                                    (ae := ArchitecturalElement new name: each1 name).
                                 r := InterAE new.
                                 r source: ae; target: each2.
                                 ae relationList add: r].
                              ifTrue: [r:= InterAE new.
                                    r source: eachAE; target: each2.
                                    eachAE relationList add: r]]]].
  ^ outputView
```

PE2 : Mapping By Synonym

Nom de la primitive :

```
mappingBySynonym: abstractView and: dictionary
```

Catégorie : correspondance.

Description : Cette primitive permet de faire la correspondance entre les éléments de la vue en entrée du framework et les synonymes des éléments de la vue abstraite proposée et définie par l'utilisateur du framework. La correspondance repose sur un dictionnaire de termes. Ce dictionnaire est étendu par le nom des éléments et, pour chacun de ces noms, l'ensemble de leurs synonymes. Cette primitive crée une vue architecturale.

Principe : Cette primitive reprend le principe algorithmique de la primitive précédente. Le dictionnaire donné contient les noms de chaque élément de la vue abstraite et les listes des synonymes concernant chaque élément.

Cette primitive permet éventuellement d'élargir le champ des possibilités d'identification d'éléments. Les noms proposés pour les éléments de la vue abstraite peuvent ne pas correspondre aux éléments de la vue concrète étudiée alors qu'ils sont sémantiquement (conceptuellemment) proches. De plus le choix des noms donnés aux éléments architecturaux par l'utilisateur peut s'avérer plus ou moins pertinent mais il exclut surtout, plus ou moins arbitrairement, des alternatives.

```
mappingBySynonym: abstractView and: dictionary

   |synList |
   outputView := View new.
   synlist := List new.
```

```
abstractView elements do: [:each |
    dictionary do: [:eachN |
        (each match: eachN)
            ifTrue: [synlist add: eachN list].
                self inputView elements do: [:eachE |
                synlist do: [eachS | (eachE  name  match: eachS)
                    ifTrue: [ae:= ArchitecturalElement  new  name: each.
                        r := InterAE new.  r  source: ae;  target:eachE.  ae  relationList  add:r.
                        outputView  elements add: ae.]]]].
^ outputView
```

PE3 : Mapping By Tree

Nom de la primitive :

mappingByTree: abstractViewElement and: elementName

Catégorie : correspondance

Description : Cette primitive permet de faire la correspondance entre les éléments de la vue en entrée du framework et les éléments trouvés à partir de l'expertise donnée. En effet, cette primitive a besoin d'un degré de connaissance sur le système étudié. La correspondance repose sur le nom d'un élément donné et toutes les classes héritant de cet élément afin de construire un élément architectural nommé "abstractViewElement". Nous indiquons que cette primitive ne peut être appliquée que sur la vue d'implémentation en tant que vue étudiée. Effectivement, elle pourrait être utilisée par d'autres vues en entrée mais nous devons améliorer l'implémentation actuelle.

Principe : L'élément (classe) proposé par l'expertise ("elementName") encapsule une caractéristique (comportement) intéressante. Cette caractéristique est représentée par un terme donné en tant que *"abstractViewElement"*. L'objectif est de trouver des éléments dans la vue étudiée qui correspondent à cette caractéristique, sachant que cette caractéristique existe dans l'élément donné ("elementName"). De plus, nous supposons que toutes ses sous-classes ont également cette caractéristique. Par suite, le nom de chaque élément de la vue étudiée sera comparé avec le nom de cet élément donné et avec le nom de toutes ses sous-classes.

mappingByTree: abstractViewElement and: elementName

```
| ae  list |
outputView := View new.
list := OrderedCollection new.
list add: elementName allSubClasses.
self inputView elements do: [:each1 |
    list do: [:each2 |
        (each1 name match: (each2 name))
            ifTrue: [outputView elements add: ae := ArchitecturalElement  new.
                    ae  name: abstractViewElement.
                    r := InterAE new.
                    r  source: ae; target: each1.
                    ae  relationList  add: r]]].
```

```
        ^ outputView
```

PE4 : Not Mapping

Nom de primitive :

notMapping: abstractView

Description : La primitive de non-correspondance inverse le principe de la primitive PE1. Elle permet d'identifier tous les éléments architecturaux qui n'ont pas été identifiés avec les critères de correspondance définis par la vue abstraite fournie.

Les primitives proposées reposent sur une mise en correspondance des éléments de la vue concrète avec des éléments et des synonymes des éléments de la vue abstraite. Cependant, une partie des éléments de la vue concrète peut rester non mis en correspondance pour plusieurs raisons : par exemple, les informations sont incomplètes ou parce que des éléments supports ne représentent pas des éléments du domaine métier (*business domain*) mais du domaine du logiciel (une pile, une collection, etc.). C'est pour cela que nous proposons cette primitive. Elle permet de récupérer ces éléments non-identifiés de la vue concrète et de les regrouper. Elle constitue en quelque sorte une anti-primitive pour les deux primitives PE1 et PE2.

Principe : Cette primitive construit un élément architectural nommé *Hors-Domaine*. Cet élément est relié (par des relations inter-éléments) à tous les éléments architecturaux de la vue concrète étudiée qui ne correspondent à aucun élément de la vue abstraite fournie. A partir de la vue concrète étudiée, elle rejette des éléments qui correspondent à la vue abstraite proposée.

```
notMapping: abstractView
   | ae elements horsDomaine |
   outputView := View new.
   outputView add: ae:= ArchitecturalElement new name: "horsDomaine".
   abstractView elements  do: [:eachElement |
      elements add: [self inputView select: [:each |
        (each name match: (eachElement name)]]].
           ae add:[ self inputView reject: elements].
           ae do: [:each |
              outputView add: r:= InterAE new. r source: ae; target: each.
              ae relationList add: r]]].
            ^ outputView
```

PE5 : Discovery By Symbol

Nom de primitive :

discoveryBySimilarityCharacteristic: threshold

Catégorie : Découverte.

Description : Cette primitive repose sur la découverte du critère de similarité textuelle entre le nom des éléments de la vue concrète étudiée. Les éléments construits et les éléments identifiés seront liés.

Principe : La primitive a comme objectif de retrouver des éléments avec des noms proches. Pour réaliser cet objectif, nous utilisons la fonction "spellAgainst" :

spellAgainst : compte les caractères identiques entre deux noms puis elle convertit le résultat en pourcentage (un entier entre 1 et 100). Cette valeur représente une mesure de ressemblance entre les deux noms. L'utilisateur du framework doit donc proposer un seuil (*threshold*), représentant sa valeur de ressemblance.

discoveryBySimilarityCharacteristic: threshold

```
outputView := View new.
1  to: inputView size do: [:i |
  i+1  to: inputView size do: [:j |
    self inputView elements do:[ (ae_i setExtracted == 0)  &
                          (ae_j setExtracted == 0)
      ifTrue:[
        value := ae_i name spellAgaints: ae_j name.
        (value  >=  threshold)
          ifTrue: [
            ae:= ArchitecturalElement new name: (ae_i name).
            outputView add: ae.
            r_i:= InetrAE new source:outputView  ae ; target:inputView ae_i.
            r_j:= InetrAE new source:outputView  ae ; target:inputView ae_j.
            ae RelationList add: r_i; add:r_i.
            outputView add: r_i;  add:r_j. ]]]].
            ae_i setExtracted := 1.
            ae_j setExtracted := 1].
            ^ outputView
```

PE6 : Discovery By Activity

Nom de primitive :

discoveryByActivityCharacteristic: threshold

Catégorie : Découverte.

Description : La découverte repose sur le critère d'activité des éléments de la vue concrète étudiée. Notre définition de l'activité d'un élément architectural est liée au nombre d'interactions que l'élément possède avec les autres éléments. Cette primitive doit analyser la vue d'implémentation, mais la primitive reste générique car nous avons toujours accès aux vues situées au niveau d'abstraction inférieur. Soit cette analyse est directe, c'est-à-dire que la vue étudiée est liée directement vers des éléments de la vue d'implémentation, soit cette analyse se réalise par l'intermédiaire des relations entre vues. En effet, à partir des éléments et des relations d'une vue, nous pouvons toujours avoir

accès aux éléments et aux relations associés des autres vues. Nous pouvons considérer donc les trois cas de figure suivants :

- la primitive ne s'applique que pour la vue d'implémentation : elle calcule l'activité à partir des relations d'invocation entre des classes ;
- la primitive est appliquée sur une vue liée directement à la vue d'implémentation : elle calcule l'activité d'un élément architectural à partir des éléments liés (par des relations InterAE de type abstraction) situés dans la vue d'implémentation ;
- la primitive est appliquée sur une vue liée à travers des relations InterAE (de type abstraction) vers la vue d'implémentation : elle calcule l'activité d'un élément architectural à partir des éléments situés dans la vue d'implémentation concernée.

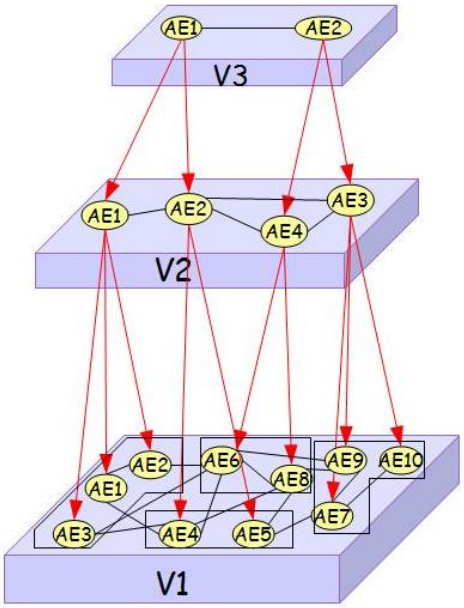

FIGURE 5.1 – Trois vues architecturales : la vue d'implémentation V1 et les vues plus abstraites V2 et V3

Nous prenons les trois vues de la figure 5.1 comme exemple. L'activité de l'élément architectural V2.AE1 va être calculé à partir de la somme d'invocations des éléments liés dans la vue V1 (V1.AE1, V1.AE2 et V1.AE3). Ainsi, l'activité de l'élément V3.AE1 sera calculée à partir des éléments V2.AE1 et V2.AE2 liés ; ce qui se calcule à partir des éléments liés dans la vue V1 (V1.AE1,V1.AE2,V1.AE3 et V1.AE4,V1.AE5).

$$L'activite(AE) = \sum(nb\ d'invocation\ des\ elements\ lies\ \ AE) \tag{5.1}$$

L'activité d'un élément architectural est définie en fonction du nombre d'interactions entre des éléments architecturaux du niveau inférieur. Par la suite, l'utilisateur définit un seuil. Si la valeur de l'activité d'un élément étudié est supérieure à ce seuil, l'élément architectural sera mis en évidence.

Principe : Dans le cadre de notre primitive, nous proposons une fonction pour mesurer le seuil à partir des relations inter-éléments (invocations) dans la vue d'implémentation. Cependant, l'utilisateur peut proposer un autre seuil.

$$Seuil = \frac{\sum nombre\ total\ d'interactions\ de\ la\ vue\ d'implementation}{nombre\ d'\ elements\ de\ la\ vue\ d'implementation} \tag{5.2}$$

Nous avons également une fonction pour mesurer l'activité de chaque élément architectural notée par la suite *findPartialActivity* : cette fonction trouve la somme des invocations par élément. Cette fonction est définie également d'une façon indépendante afin de garder la flexibilité de la primitive. Elle prend en compte les relations (invocations) de chaque élément lié à l'élément architectural étudié dans un niveau d'abstraction inférieur.

```
discoveryByActivityCharacteristic: threshold

   |partielActivity|
   outputView := View new.
   self inputView elements do:[:each |
         partialActivity := findPartialActivity: each.
          (threshold <  partialActivity)
            ifTrue: [outputView add:each. r:= InetrAE new.
                  r source:outputView each; target: each.
                  each RelationList add: r]].
  ^ outputView
```

PE7 : Discovery By Functionality

Nom de primitive :

```
discoveryByFunctionalityCharacteristic: threshold
```

Catégorie : Découverte.

Description : La découverte repose sur le critère de fonctionnalité des éléments de la vue étudiée. Le critère basé sur le nombre des intéractions entre les éléments est important à étudier, mais la direction de ces interactions fournit également de bonnes valeurs. A partir du nombre d'interactions, nous pouvons découvrir la cohésion entre des éléments ou l'activité des éléments, *etc.* Mais l'orientation des interactions nous permet d'obtenir aussi des informations sur la fonctionnalité des éléments. Notre définition de la fonctionnalité d'un élément architectural est liée au nombre de messages envoyés et reçus par les éléments liés à cet élément architectural dans un niveau d'abstraction inférieur. Selon l'importance du nombre des interactions (messages reçus ou envoyés), la fonctionnalité d'un élément architectural sera mise en évidence. Une valeur considérée comme importante peut être proposée en tant que seuil par l'utilisateur.

Notre approche propose quatre rôles globaux comme fonctionnalité d'un élément architectural :

- *Fournisseur : les éléments qui fournissent l'information* ;
- *Consommateur : les éléments qui consomment l'information* ;
- *Neutre : les éléments qui sont ni fournisseur ni consommateur* ;
- *Autre : les éléments qui fournissent et consomment de l'information.*

Tout d'abord, la particularité de cette primitive est qu'elle permet de savoir si la vue étudiée d'un système selon un angle d'observation (comme une vue construite selon un domaine d'application) peut être partagée par ces quatre concepts de haut niveau. Chaque élément étudié représente un concept existant du système. En utilisant cette primitive, l'utilisateur peut lier la fonctionnalité de haut niveau avec la fonctionnalité implémentée en établissant des relations inter-éléments. A noter

qu'un élément ne peut pas être généralement considéré à 100 % *fournisseur* ou *consommateur* ou *neutre*.

Principe : Dans le cadre de notre primitive, nous proposons une fonction afin de définir un seuil. Ce dernier se mesure à partir des relations inter-éléments de la vue d'implémentation de la façon suivante :

$$valeur\ Moyenne = \frac{\sum\ nombre\ d'invocations\ d'une\ vue\ d'implementation}{nombre\ de\ classes} \tag{5.3}$$

La vue architecturale construite possède alors quatre éléments architecturaux par rapport aux fonctionnalités proposées. Les éléments construits sont liés (par des relations inter-éléments) aux éléments architecturaux de la vue concrète étudiée qui leur correspondent par rapport au critère fourni. Finalement, tous les éléments qui ne correspondent pas aux critères fournis sont liés à l'élément architectural nommé "Autre".

discoveryByFunctionalityCharacteristic: threshold

```
| valueE  valueR  ae1 ae2 ae3  elements |
outputView := View new.
outputView add: (ae1 := ArchitecturalElement name: "Fournisseur").
outputView add: (ae2 := ArchitecturalElement name: "Consommateur").
outputView add: (ae3 := ArchitecturalElement name: "Neutre").
outputView add: (ae4 := ArchitecturalElement name: "Other").
[self inputView do:[:each|   elements add: getAbstractedElements: each.
          elements do:[:each |
            valueE := findSendInvocation:each.
            valueR := findReceivedInvocation:each.
             ((valueE  >   threshold) and: (valueR  <  threshold))
               ifTrue: [ae1 add: (r:=InterAE new; source: ae1;
                   target: inputView each).
                    ae1 relationList add: r].
             ((valueR   >   threshold) and: (ValueE <  threshold))
                 ifTrue: [ae2 add: (r:= InterAE new; source: ae2;
                      target: inputView each).
                      ae2 relationList add: r].
             ((valueR  <   threshold) and: (ValueE <  threshold))
                 ifTrue: [ae3 add: (r:= InterAE new; source: ae3;
                      target: inputView each ).
                      ae3 relationList add: r].
             ((valueR   >   threshold) and: (ValueE >  threshold))
                 ifTrue: [ae4  add: (r:= InterAE new; source: ae4;
                      target: inputView each).
                       ae4 relationList add: r]]]].
 ^ outputView
```

5.1.3 Primitives pour la construction des relations architecturales (PR)

Nous présentons les primitives concernant la construction des relations architecturales (PRs). Les primitives concernant des relations doivent être appliquées après l'utilisation des primitives de construction concernant des éléments architecturaux (PEs). Une vue construite par une primitive de PEs possède des éléments architecturaux. Cette vue est effectivement la même que la vue obtenue en sortie après l'utilisation d'une primitive de PRs. Cette fois des relations architecturales sont ajoutées.

Le principe dans le cas des primitives de la catégorie des points de vue *correspondance* est de faire la correspondance entre les relations entre les éléments de la vue étudiée et les relations entre les éléments de la vue abstraite proposée et définie par l'utilisateur. A partir de la vue étudiée et de la vue abstraite, la primitive construit des relations entre les éléments de la vue en sortie. Ainsi elle relie ces relations construites avec une ou plusieurs relations inter-relation vers la vue concrète étudiée. Ces relations de type inter-relation sont obtenues sur la base des relations proposées entre des éléments de la vue abstraite ainsi qu'en fonction de leur existence dans la vue étudiée. Afin d'établir une règle générale pour construire des relations architecturales entre des éléments architecturaux, nous suivons un contexte simple et raisonnable : *la construction d'une relation architecturale entre deux éléments architecturaux dans une vue au niveau d'abstraction* n *dépend de l'existence d'une ou plusieurs relations entre des éléments liés au niveau d'abstraction* n-1. Chaque relation architecturale construite possède une métrique (le poids) montrant le nombre de relations existantes entre des éléments associés.

Le principe dans le cas des primitives de catégorie des points de vue *découverte* est de découvrir des relations architecturales entre des éléments architecturaux de la vue en sortie donnée, à partir des relations entre des éléments de la vue concrète étudiée. La découverte repose également sur le critère d'existence d'une ou plusieurs relations entre des éléments associés à chaque élément architectural (de la vue sortie donnée).

Dans les deux cas, une relation architecturale construite est liée par des relations de type inter-relation (InterAR) vers des relations de la vue étudiée (cf. Méta-modèle BeeEye).

PR1 : Mapping Relationship

Nom de primitive :

mappingRelationship: abstractView

Catégorie : Correspondance.

Description : Cette primitive construit des relations architecturales dans la vue architecturale construite.

Principe : Pour vérifier la correspondance d'une relation entre deux éléments de la vue abstraite, la primitive vérifie tout d'abord l'existence de ces éléments dans la vue donnée en sortie de la primitive. Si ces éléments existent, elle recherche parmi des éléments liés par des relations inter-éléments (effectivement dans un niveau d'abstraction inférieur) afin d'obtenir d'éventuelles relations entre eux.

Si la primitive ne distingue pas des éléments correspondant à une relation recherchée dans la vue en donnée sortie, ou qu'aucune relation n'existe entre des éléments liés aux éléments distingués, nous en déduirons que cette relation est une relation *Fantôme*. Cette dernière est une relation que l'utilisateur a proposé dans la vue abstraite, mais elle n'existe pas (au sens où elle n'a pas été identifiée

dans la vue donnée en entrée). Soit des éléments connectés ne sont pas construits, soit cette relation n'a pas été implémentée dans la vue inférieure.

mappingRelationship: abstractView

```
| r  r1  r2 weight  elements elements1  elements2 outputView |
outputView := View new.
weight := 0.
elements1 := ArchitecturalElement new.
elements2 := ArchitecturalElement new.
abstractView  relations  do: [:eachRelation |
   (outputView elements do: [:eachE1 :eachE2 | eachE1 match: eachRelation source
                                          and: eachE2 match:eachRelation target])
      ifTrue: [ elements1  add: (self getAbstractedElements: eachE1 from: inputView).
          elements2  add: (self getAbstractedElements: eachE2  from: inputView).
          (elements1  hasRelationWith:elements2)
         ifTrue: [ outputView  add: (r:= ArchitecturalRelation new. r source:eachE1; target:eachE2).
             elements1  do: [:each1 |
                elements2  do: [:each2 | each2 relationWith:  each1.
                     r1:= InterAR new  source: outputView r; target: each1  relation.
                     r2:= InterAR new  source: outputView r; target: each2  relation]].
                 weight:= weight + 1]]].
 ^ outputView
```

getAbstractedElements retrouve tous les éléments liés à l'élément considéré dans un niveau d'abstraction inférieur. Le fonctionnement de *hasRelationWith* vérifie l'existence d'une ou plusieurs relations entre des éléments architecturaux (situés à un niveau inférieur). Si le résultat est positif, la primitive construit une relation architecturale dans la vue de sortie. *relationWith* permet au contraire de retrouver ces relations afin de les lier par des relations de type InterAR vers la relation construite. Ces relations permettent de savoir, à partir d'une relation, quelles autres relations ont été construites par la primitive. La métrique poids montre le nombre des relations retrouvées.

PR2 : Discovery Relationship

Nom de primitive :

discoveryRelationship: threshold.

Catégorie : Découverte.

Description : Cette primitive permet de découvrir des relations architecturales entre des éléments de la vue en sortie de la primitive.

Principe : Lorsque des éléments associés (situés à un niveau d'abstraction inférieur) à deux éléments architecturaux AE1 et AE2 ont des relations entre eux, la primitive déduit l'existence d'une relation architecturale entre ces deux éléments.

discoveryRelationship: threshold

```
|elements1 elements2 weight outputView |
outputView := View new.
weight := 0.
elements1 := ArchitecturalElement new.
elements2 := ArchitecturalElement new.
1  to: outputView size  do: [:i |
  i+1  to: outputView size  do: [:j |
    elements1 add: (self getAbstractedElements:outputView AE_i from:inputView).
      elements2 add: (self getAbstractedElements:outputView AE_j from:inputView).
        (elements1 hasRelationWith:elements2)
          ifTrue: [ outputView add: [r:= ArchitecturalRelation new.
                  r source:outputView AE_i; target:outputView AE_j].
            elements1  do: [:each1 |
              elements2  do: [:each2 | each2 relationWith: each1.
              r1:= InterAR new source: outputView r; target: each1 relation.
              r2:= InterAR new source: outputView r; target: each2 relation]].
              weight:= weight + 1]]].
 ^ outputView
```

5.1.4 Primitives concernant les techniques de construction (PT)

Dans cette section, nous présentons les primitives concernant les techniques de construction (PTs) proposées dans le chapitre précédent. Ces primitives sont toujours appliquées après les primitives de construction des éléments architecturaux (PEs) et les primitives de construction des relations architecturales (PRs). Par contre, nous devons faire certaines spécialisations sur les primitives proposées PEs et PRs. Bien que les principes restent les mêmes, certaines spécialisations sont nécessaires. Pour réaliser ces primitives, nous devons analyser les éléments et les relations liés à l'élément porté (scope) considéré. En effet, cet ensemble associé va être utilisé dans le mécanisme de ces primitives. Les principes définis par les primitives (PEs) et (PRs) sont utilisables en réalisant une adaptation.

Les primitives définies dans le tableau 5.2 sont fonction de trois techniques de construction de vues proposées, à savoir *abstraction*, *raffinement* et *composition*. Elles sont génériques et peuvent être utilisées quelle que soit la vue concrète étudiée.

Nous rappelons que des relations entre des éléments de la vue étudiée et la vue construite donnée sont de type InterAE et, par conséquent, les relations entre des relations de la vue étudiée et la vue construite donnée sont de type InterAR. Les primitives concernant les techniques vont spécialiser des relations entre la vue étudiée et la vue construite : les relations seront de type *AEAbstraction*, *AERaffinement*, *AEComposition* et *AEIdentity*.

	Primitives de construction de vues selon les techniques de construction
TP1	`abstractionTechnique:\ scope`
TP2	`refinementTechnique:\ scope`
TP3	`compositionTechnique:\ scope`

TABLE 5.2 – Primitives de BeeEye pour construire des relations entre des vues selon les techniques de construction (abstraction, raffinement et composition).

Par la suite, nous présentons ces primitives.

TP1 : Technique d'abstraction

Nom de primitive :

abstractionTechnique: scope

Description : La vue en entrée de cette primitive est située au niveau d'abstraction *n-1* et la vue en sortie est située au niveau d'abstraction *n*. Cette primitive a pour objectif de construire des relations d'abstraction à partir des éléments et relations de la vue construite vers des éléments et relations de la vue étudiée.

Comme mentionné auparavant, nous avons des relations de type inter-élément (InterAE) entre des éléments et des relations de type inter-relation entre des relations (InterAR). En choisissant la technique d'abstraction, la primitive construit des relations de type AEAbstraction entre les éléments de deux vues. Ainsi, des relations de type ARAbstraction entre les relations de deux vues sont construites. De ce fait, le type de relation sera précis entre les deux vues.

Chaque élément et relation construit dans la vue construite possède une liste de relations (respectivement InterAE et InterAR) à partir de laquelle la primitive est capable de construire des relations de type d'abstraction.

```
abstractionTechnique: scope
   (scope == nil)
      ifTrue:[ outputView  elements  do:[:eachElement |
               eachElement relationList do:[:eachR |
               self makeRelationFrom: eachR source to:eachR target  kind:AEAbstraction in:outputView].
                 outputView  relations  do:[:eachRelation|
                  eachRelation relationList do: [:eachR |
             self makeRelationFrom: eachR source to:eachR  target kind: ARAbstraction in: outputView]]]].
      IfFalse:[ scope relationList do:[:eachR |
               self makeRelationFrom: eachR source to:eachR target kind: AEAbstraction in: outputView.
                eachRelation relationList do: [:eachR |
              self makeRelationFrom: eachR source to:eachR target kind: ARAbstraction in: outputView]]].
```

TP2 : Technique de raffinement

Nom de primitive :

refinementTechnique: scope

Description : La primitive utilise une vue étudiée (en entrée) et une vue construite (en sortie), ainsi qu'un élément architectural portée (scope). Cet élément est considéré comme élément à raffiner. En effet, cet élément fait partie des éléments de la vue en entrée de la primitive. La vue en sortie possède des éléments et relations obtenus en raffinant cet élément particulier. La primitive TP2 consiste à construire des relations de raffinement de type AERefinement entre l'élément à raffiner et les éléments construits.

Cependant, la vue en entrée de la primitive contient également des éléments restés inchangés. La primitive TP2 a comme objectif de représenter ces éléments dans la vue de sortie, ainsi que de construire des relations identité de type AEIdentity entre ces éléments appartenant à deux vues.

```
refinementTechnique: scope
   outputView elements do: [:each | r := AERefinement new.
                              r  source: scope; target: each.
                              outputView add: r].
   inputView  do: [:eachElement | (eachElement != scope)
                              ifTrue: [outputView  add: eachElement ;
                                   add:  (r := AEIdentity  new; source: (outputView  eachElement); tar-
get: (inputView  eachElement))]].
                              inferRelationshipFrom: inputView to: outputView.
```

Le fonctionnement de *inferRelationship* est de transférer des relations entre des éléments identiques et l'élément à raffiner vers la vue construite.

TP3 : Technique de composition

Nom de primitive :

```
compositionTechnique: scope
```

Description : La primitive utilise une vue étudiée et une vue concrète construite, ainsi qu'un élément architectural à composer. En effet, cet élément fait partie de la vue étudiée, mais, comme mentionné précédemment, il doit également faire partie de la vue construite.

A partir de cet élément à composer, nous avons obtenu un ou plusieurs éléments architecturaux. La vue construite donnée possède ces éléments ainsi que les relations entre ces éléments. La primitive TP3 a comme objectif de construire d'abord un élément identique à l'élément à composer dans la vue construite. Puis, elle construit des relations de composition de type AEComposition à partir de cet élément vers les éléments obtenus à partir de cet élément. Ensuite, elle construit les relations identité de type AEIdentity entre les éléments identiques appartenant à la vue étudiée et à la vue construite.

```
compositionTechnique: scope

   outputView add: scope.
   outputView do: [:eachE |
         (eachE  != scope)
            ifTrue: [r:= AEComposition new.
                  r source: outputView scope; target: each.
                  outputView add: r]].
   inputView do: [:eachElement |
         (eachElement != scope)
            ifTrue: [outputView
                        add: eachElement ;
                        add: (r := AEIdentity  new; source: outputView  eachElement;
                                          target: inputView  eachElement)]].
   self inferRelationshipFrom: inputView to: outputView.
```

5.1.5 Primitives concernant les opérateurs de construction (OP)

Nous proposons dans le cadre de BeeEye un certain nombre d'opérateurs permettant de réaliser certaines tâches sur des vues construites. Ces opérateurs sont présentés dans le tableau 5.3 :

	Primitives des opérateurs
OP1	`unionOperator:\ view`
OP2	`intersectionOperator:\ view`
OP3	`complementOperator:\ view`

TABLE 5.3 – Les primitives proposées par l'approche BeeEye pour construire des vues utilisant des opérateurs

Chacun de ces opérateurs nous permet de construire une nouvelle vue à partir de deux ou plusieurs autres vues construites. Cette nouvelle vue construite représente les vues utilisées selon l'opérateur appliqué. Chaque primitive proposée dans les sections précédentes (PEs, PRs et PTs) permet de construire une vue architecturale.

5.1.6 OP1 : Opérateur d'Union

Nom de primitive :

```
unionOperator: view
```

Description : L'*union* de deux vues architecturales est une vue architecturale contenant chacun des éléments de la première vue et de la deuxième vue. Les éléments communs aux deux vues ne sont conservés qu'une seule fois, c'est-à-dire que l'opération d'union combine deux éléments identiques en un. Des éléments identiques sont ceux qui portent les mêmes noms. Dans ce cas, un élément architectural représentant ces deux éléments est construit. Cet élément porte le nom d'un des éléments identiques, ainsi il porte toutes les relations appartenant à chacun de ces deux éléments. Comme mentionné auparavant, chaque élément architectural a une liste de relations qui montre l'ensemble des relations auxquelles il appartient (vers d'autres éléments).

```
unionOperator: view
  | ouputView |
  outputView := View new.
  inputView elements do: [:each1 |
    view elements do:[:each2 |
      (each1 name match: each2 name)
       ifTrue:[outputView elements add: [ae:= ArchitecturalElement new.
                ae name: each1 name.
                ae relationList add: each1 relationList; add: each2 relationList]]
       ifFalse: [outputView elements add: each1; add:each2]]].
  ^ outputView
```

5.1.7 OP2 : Opérateur d'intersection

Nom de primitive :

intersectionOperator: view

Description : L'*intersection* de deux vues architecturales est une vue architecturale contenant seulement des éléments communs aux deux vues. La condition est la même que pour l'union. Cependant, la vue construite utilisant cet opérateur possède tous les éléments communs. Chaque fois que l'on trouve deux éléments portant le même nom, l'un issue de la première vue, et l'autre issue de la seconde vue, ces deux éléments sont inséré dans la vue construite. L'objectif est de pouvoir comparer les deux éléments entre eux afin de distinguer les parties non-communes liées par des relations inter-éléments. Deux éléments communs ne sont pas forcément liés aux mêmes éléments d'une autre vue.

```
intersectionOperator: view
   | ouputView |
   outputView := View new.
   inputView elements do: [:each1 |
      view elements do:[:each2 |
         (each1 name match: each2 name)
             ifTrue:[outputView elements add:each1; add:each2]]].
   ^ outputView
```

5.1.8 OP3 : Opérateur de différence

Nom de primitive :

complementOperator: view

Description : La *différence* de deux vues architecturales est une vue architecturale contenant des éléments de la première vue qu'on ne trouve pas dans la seconde. La condition est la même que pour l'union. Ainsi, la vue construite utilisant cet opérateur possède tous les éléments non-communs. L'objectif est de pouvoir reconnaître des éléments qui n'ont pas répondu à un certain nombre de critères considérés par la première vue.

```
complementOperator:view
   | ouputView |
   outputView := View new.
   inputView elements do: [:each1 |
      view elements do:[:each2 |
         (each1 name match: each2 name)
          ifFalse:[outputView elements add:each1; add:each2]]].
   ^ outputView
```

Dans le chapitre suivant, nous allons présenter une utilisation des primitives portant sur les éléments, les relations, les techniques et les opérateurs.

Chapitre 6

Mise en pratique de BeeEye

Ce chapitre illustre une utilisation des primitives proposées par l'approche en insistant sur leur généricité. Ceci sera montré en deux étapes : d'une part en combinant ces primitives afin de définir des algorithmes différents de construction de points de vue, et d'autre part en utilisant ces points de vue sur un cas d'étude pour construire des vues architecturales. Ainsi nous montrons le mécanisme de définition d'un point de vue proposé par BeeEye et comment les nouveaux points de vue peuvent être définis.

La suite du chapitre est organisée de la façon suivante. Nous décrivons d'abord le cas d'étude. Ensuite, nous présentons cinq exemples de points de vue de différentes natures. Pour la construction de ces points de vue, nous utilisons les primitives proposées dans le chapitre 5.

Certains points de vue permettent d'illustrer des combinaisons entre des vues architecturales construites ainsi que des enchaînements verticaux et/ou horizontaux de vues architecturales.

Système étudié : Nous présentons dans cette section notre cas d'étude. C'est un système à objets de moyenne taille. L'objectif est de mettre en évidence différents aspects proposés par notre approche à travers cette étude de cas.

Le système choisi se situe dans le domaine (*business domain)* bancaire. C'est l'exemple d'un système logiciel à objets d'environ 100 classes pour lequel la documentation n'est pas disponible. Nous allons utiliser notre approche pour construire des vues architecturales de ce système bancaire. L'approche BeeEye va nous permettre de construire des vues architecturales en exploitant le code source du système (le code source Smalltalk est disponible).

6.1 Quelques exemples de points de vue proposés par BeeEye

Dans cette section, nous donnons quelques exemples de points de vue proposés par BeeEye afin de construire des vues architecturales à partir d'un système logiciel bancaire. Ces points de vue sont donnés afin d'illustrer l'utilisation de l'approche, notamment la façon de définir des points de vue à partir des primitives. Ces points de vue définis peuvent être ensuite utilisés tels quels par des utilisateurs de l'approche. L'utilisateur spécifie la construction d'une vue en choisissant la vue étudiée, la vue abstraite (dans le cas de catégorie des points de vue *correspondance*) ou le seuil considéré (dans le cas de la catégorie des points de vue *découverte*) et le point de vue.

BeeEye prend en compte la vue code source étudiée et un point de vue défini afin de construire une autre vue concrète. Comme mentionné auparavant, le framework BeeEye est basé sur l'outil

d'analyse Moose [DGLD05]. Ceci nous permet de construire *la vue d'implémentation* (concrète) située au niveau d'abstraction plus élevé que la vue code source (cf. chapitre 4).

Lors de l'application des algorithmes de construction, les primitives traitent par défaut les éléments qui sont des classes et des relations d'invocation.

Par la suite, nous suivons le processus de définition des points de vue proposés. L'objectif est de montrer en détail la définition d'un point de vue proposé par l'utilisateur et de valider le fait que les primitives sont des briques de base de BeeEye pour définir des nouveaux points de vue.

6.1.1 Premier exemple sur le point de vue du domaine métier (*business domain*)

Nous proposons le point de vue domaine métier (*business domain viewpoint*) (BDVP) avec les propriétés suivantes :

Facette : Domaine d'application.

Point de vue : Domaine métier.

Catégorie : *Correspondance.*

Technique choisie : Abstraction.

Description :

$$V = Application(PdVDM\ (AV,\ algorithme), ImpV,\ Abstraction). \tag{6.1}$$

Ainsi le point de vue domaine métier appartient à la catégorie *correspondance* et il est appliqué sur la vue d'implémentation (ImpV). Le point de vue est composé par la vue abstraite VA1 et un algorithme de construction. L'algorithme est "faire la correspondance par symbole". Ainsi la technique choisie est une abstraction. La vue abstraite proposée par un utilisateur représente trois éléments architecturaux selon le cas d'étude considéré :

{(*Client, Card, Account*)}.

Certaines relations sont proposées entre ces éléments :

{(*un client peut avoir un ou plusieurs comptes : une relation est proposée à partir de Client vers Account; une carte porte sur un compte spécifique : une relation est proposée à partir de Card vers Account*)}.

La figure 6.1 montre la vue abstraite proposée et définie par un utilisateur du framework. Cette vue donnée est simple traduisant la connaissance de cet utilisateur. Le nombre d'éléments proposés est de 3, alors que la vue en entrée en possède 100. Nous définissons un *facteur d'abstraction d'une vue* qui correspond au ratio du nombre d'éléments de la vue en entrée par le nombre d'éléments de la vue construite. Ce facteur est égal à environ 30 pour la vue abstraite proposée.

A partir de ce facteur d'abstraction, nous pouvons étudier la croissance ou décroissance du niveau d'abstraction d'une vue construite envers une autre. La décroissance de ce facteur montre une augmentation du nombre d'éléments architecturaux retrouvés. Alors que la croissance de ce facteur montre une encapsulation plus élevée d'éléments retrouvés. La réduction du nombre d'éléments n'est pas forcément un bon indicateur pour faciliter la compréhension d'un système étudié. Cependant,

nous avons étudié quelques pistes afin de permettre à l'utilisateur de choisir le niveau d'abstraction souhaitable. Nous allons en discuter dans le chapitre 7.

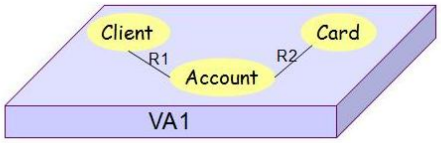

FIGURE 6.1 – La vue abstraite VA1 proposée pour le système étudié

Les primitives utilisées pour la définition de l'algorithme de construction

En utilisant les primitives génériques proposées dans le chapitre 5, nous définissons l'algorithme de construction. Parmi ces primitives, nous choisissons celles qui sont pertinentes par rapport aux principes de correspondance par symbole et la technique d'abstraction. Les primitives choisies sont les suivantes :

PE1: mappingBySymbol: abstractView ;

PE2: mappingBySynonym: abstractView and: dictionary ;

PE4: notMapping: abstractView ;

OP1: unionOperator: view ;

PR1: mappingRelationship: abstractView ;

TP1: abstractionTechnique: scope.

Nous avons considéré les trois premières primitives pour obtenir des éléments architecturaux et la primitive PR1 pour obtenir des relations entre ces éléments. La primitive TP1 sera appliquée sur la vue construite pour réaliser des relations d'abstraction entre la vue étudiée et cette vue construite. Evidemment, rien ne nous oblige à utiliser toutes ces primitives : par exemple si on ne souhaite pas avoir d'éléments qui ne correspondent pas aux éléments proposés dans la vue abstraite, l'utilisation de la primitive PE3 est inutile.

Construction des éléments architecturaux

Chacune des primitives PE1, PE2 et PE4 construit une vue architecturale contenant des éléments architecturaux utilisant la vue abstraite proposée. La vue construite par PE1 possède trois éléments

Algorithm 1 Construction d'une vue architecturale utilisant le point de vue DM proposé

Require: Input View impV, abstract view aV, dictionary dict.

```
V1 := impV mappingBySymbol: aV
V2 := impV mappingBySynonym: aV and: dict
V3 := impV notMapping: aV
V4 := V1 unionOperator: V2
V5 := V3 unionOperator: V4
V6 := impV mappingRelationship: aV
V7 := impV abstractionTechnique: scope
```

architecturaux, ainsi que les liens inter-éléments vers des éléments appropriés retrouvés dans la vue étudiée. La *métrique poids* concernant chaque élément architectural construit est précisée dans le tableau 6.1.

Nous fournissons également un *indice d'efficacité de la primitive* : cet indice montre le pourcentage des éléments reconnus par chaque primitive au sens de la vue étudiée. Cet indice est égal à *43 %* pour la primitive PE1. Ce qui montre que dans notre cas d'étude, presque la moitié des éléments de la vue étudiée sont reconnus/identifiés par l'utilisation de cette primitive.

Elément construit	poids des éléments
Account	16
Client	3
Card	20

TABLE 6.1 – Les éléments identifiés par PE1 et leur poids

Comme mentionné auparavant, l'utilisateur peut proposer (ou utiliser) un dictionnaire de synonymes des termes utilisés (dans la vue abstraite par exemple) ; ce dictionnaire est utilisé par la primitive PE2. Le tableau 6.2 représente un exemple de dictionnaire des synonymes proposés pour la vue abstraite VA1. Il ne possède qu'un synonyme donné pour l'élément *Client*. En utilisant ce dictionnaire et la vue abstraite VA1, la primitive PE2 construit la vue architecturale V2. Le tableau 6.3 montre les résultats obtenus après l'utilisation de cette primitive. Cette primitive permet souvent d'identifier une partie des éléments restés non-identifiés. Cependant, le résultat obtenu dans notre cas d'étude ne possède que trois éléments architecturaux. Ces trois éléments ne sont pas identifiés après l'utilisation de PE1.

élément construit	synonyme proposé
Account	
Client	Customer
Card	

TABLE 6.2 – Le dictionnaire proposé pour la vue abstraite VA1

L'indice d'efficacité de cette primitive est de *8 %*. Une explication possible pour ce faible taux est le nombre limité de synonymes proposés dans le dictionnaire. En étendant le dictionnaire proposé, nous donnons plus de possibilités pour retrouver d'autres éléments architecturaux.

La dernière primitive appliquée sur la vue étudiée (PE4) construit la vue architecturale V3 contenant l'élément *Hors-Domaine*. Le résultat obtenu est montré dans le tableau 6.4. Le poids de cet

élément construit	poids des éléments
Account	0
Client	3
Card	0

TABLE 6.3 – Les éléments identifiés et le nombre des éléments identifiés par PE2

élément montre qu'il reste 46 éléments de la vue étudiée qui ne sont pas pris en compte par la connaissance donnée dans la vue abstraite.

élément construit	poids d'élément
Hors-Domaine	46

TABLE 6.4 – L'élément construit et le nombre d'éléments identifiés par PE3

Union des éléments architecturaux construits

Le tableau 6.5 illustre le résultat d'union des vues, ainsi que la métrique de poids pour chaque élément construit. Nous avons également proposé une métrique montrant *la propagation* de chaque élément architectural construit au sein de la vue étudiée. Cette propagation montre le pourcentage couvert par un élément construit au sens de la vue étudiée. Cela montre quel pourcentage d'une vue étudiée (représentant le système étudié) est couvert par le concept révélé par un élément architectural.

élément construit	poids d'élément	propagation
Account	16	18.18%
Client	6	6.81%
Card	20	22.72%
Hors-Domaine	46	52%

TABLE 6.5 – Union des vues construites par chaque primitive proposée pour PdVDM

L'élément *Hors-Domaine* peut être considéré comme un *élément important* car il couvre un nombre important d'éléments de la vue étudiée. Cet élément, donc, peut être étudié en tant que l'élément portée (scope) dans un autre processus de construction de vues. Dans ce cas, cet élément est considéré comme une vue étudiée.

Construction des relations

A partir des relations proposées dans la vue abstraite, la primitive PR1 construit des relations architecturales entre des éléments construits. Tout d'abord cette primitive retrouve les éléments architecturaux proposés dans la vue construite, ainsi que les relations proposées parmi des relations de la vue étudiée. En effet, des relations architecturales sont établies entre des éléments de la vue construite.

La dernière étape est l'application de la primitive TP1 afin de construire des relations d'abstraction entre les éléments et les relations de la vue étudiée et la vue construite. En effet, cette primitive

construit des relations de type AEAbstraction entre des éléments de deux vues, et des relations de type ARAbstraction entre deux relations de deux vues.

Nous remarquons que le poids de chaque élément architectural correspond au nombre des relations d'abstraction liées à cet élément. Cela est effectivement vrai pour les relations.

Nous avons montré à travers ce point de vue, la construction d'une vue architecturale à partir de la facette Domaine d'Application. Cette vue valide l'existence de trois éléments proposés par la vue abstraite, mais aussi une partie non-identifiée. Ainsi les résultats obtenus par ce point de vue expliquent notre choix concernant la proposition de la catégorie des points de vue *découverte*. L'élément *Hors-Domaine* comprend une partie importante restée non-identifiée, ce qui montre la nécessité de considérer des analyses ultérieures.

6.1.2 Premier exemple sur le point de vue patron MVC

Nous proposons le point de vue MVC avec les propriétés suivantes.

Facette : Patron Logiciel (exemple considéré MVC).

Point de vue : Patron MVC.

Catégorie : *Correspondance*.

Technique choisie : Abstraction.

Description :

$$V = Application\ (\ PdVMVC\ (AV,\ algorithme),\ ImpV,\ Abstraction). \tag{6.2}$$

Nous avons choisi le patron *Modèle-Vue-Contrôleur* pour être appliqué en tant qu'une architecture supposée dans le cadre d'un point de vue de la catégorie *correspondance*. Cette architecture supposée est représentée en tant qu'une vue abstraite proposée, contenant des éléments architecturaux et des relations entre ces éléments. La figure 6.2 montre cette vue abstraite proposée.

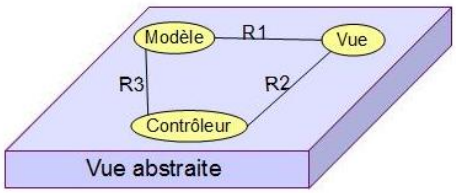

FIGURE 6.2 – La vue abstraite proposée pour le patron MVC considéré

L'utilisateur peut faire plusieurs usages des points de vue *Patron* :

- en analysant les résultats obtenus et les métriques proposées, il peut déduire si le patron choisi a été respecté dans l'étude de cas considérée.
- si le patron considéré a été respecté, donc la vue construite selon ce point de vue fournit une vision globale sur la manière dont les concepts du patron ont été partagés au sein de la vue étudiée.
- en réalisant le point de vue sur différentes versions de la vue étudiée d'un système, l'utilisateur peut découvrir la tendance sémantique des concepts du patron. C'est-à-dire qu'il peut avoir

une idée sur la façon d'évoluer du système, éventuellement du point de vue de ce patron, au fil du temps.

Les primitives utilisées pour la construction de l'algorithme de construction

Les primitives suivantes peuvent être utilisées pour définir l'algorithme de construction de PdVMVC :

PE2: mappingBySynonym: abstractView and: dictionary ;

PE3: mappingByTree: abstractViewElement and: elementName ;

PE4: notMapping: abstractView ;

OP1: unionOperator: view ;

PR1: mappingRelationship: abstractView ;

TP1: abstractionTechnique: scope.

Pour construire une vue architecturale selon un patron considéré, il n'est pas évident de retrouver des éléments proposés dans la vue étudiée en exploitant seulement leur nom. Pour cette raison nous commençons à partir de la primitive PE2. Cette primitive permet de définir un dictionnaire des synonymes : ce dictionnaire possède le nom d'éléments proposés dans la vue abstraite ainsi que des listes de synonymes concernant chaque nom proposé. Par exemple, dans le cas du point de vue *patron MVC*, nous avons les trois éléments de base : *Modèle, vue, contrôleur*. L'utilisateur propose une liste de synonymes pour chacun de ces éléments. La liste proposée pour l'élément "Modèle" doit comprendre les termes de domaine du métier. Ainsi il peut comprendre les termes correspondant au domaine utilitaire tels que : data, db, store, *etc.* De ce fait pour l'élément "Vue", le dictionnaire possède une autre liste qui comprend le domaine utilitaire graphique (GUI). Les termes peuvent être : éditeur, visualise, tableau, *etc.* En revanche, ce n'est pas possible de définir une liste pour l'élément "contrôleur" car l'identification se fait en analysant les relations et pas par une analyse syntaxique. Afin de retrouver les éléments liés à cet élément, nous devons étudier des interactions entre des éléments.

élément construit	synonyme proposé
Model	Account, Client, Card , Store, Data, DB
View	View, Display, UserInterface, Frame, Print

TABLE 6.6 – Le dictionnaire proposé pour la vue abstraite définissant le patron MVC

L'algorithme prend en compte ces primitives de la façon qui suive dans l'algorithme (7).

Algorithm 2 Construction d'une vue architecturale utilisant le point de vue MVC proposé

Require: La vue entrée impV, La vue abstraite aV

```
V1 := impV mappingBySynonym: aV and: dict
V2 := impV mappingByTree: abstractViewElement and: elementName
V3 := impV notMapping: aV
V4 := V1 unionOperator: V2
V5 := V3 unionOperator: V4
V5 := impV mappingRelationship: aV
V5 := impV abstractionTechnique: scope.
```

Dans la proposition d'algorithme de PdVMVC, nous avons intégré ainsi la primitive PE3. Comme mentionné auparavant, l'utilisation de PE3 a besoin d'une expertise. L'expertise donne des informations sur l'élément "Vue" dans le cas du *patron MVC*. Au cours de notre étude de l'approche sur le *patron MVC* dans un environment Smalltalk, nous avons pris connaissance de l'existence d'une classe nommée "VisualComponent". Cette classe et ses sous-classes permettent de représenter des fonctionnalités liées à l'élément "Vue". Cette expertise donnée révèle un indice qui facilite la recherche des éléments concernant l'élément architectural "Vue" dans la vue étudiée.

Nous avons montré les résultats obtenus en appliquant cet algorithme dans le tableau 6.7. Ce dernier représente l'union des éléments architecturaux construits, le poids de chaque élément ainsi que le nombre des éléments non-identifiés.

élément construit	poids d'élément	propagation
Modéle	42	47%
Vue	6	6.8 %
Contrôleur	0	0
Hors-Domaine	40	45 %

TABLE 6.7 – Union des vues construites par chaque primitive proposée pour PdVMVC

La majorité des éléments découverts font partie de l'élément "modèle" ; ce qui est cohérent par rapport à notre cas d'étude car il représente un système bancaire. C'est-à-dire qu'en regardant le code de près, nous réalisons que le système étudié ne décline pas le patron MVC. De fait, il n'a pas été programmé en utilisant ce patron de conception. Utiliser un point de vue de catégorie *Correspondant*, ne nous a permis de retrouver que des éléments concernant les deux éléments architecturaux "modèle" et "vue". Afin de retrouver les éléments appropriés au "contrôleur", nous avons besoin d'étudier le code plus attentivement. En effet, l'élément *Hors-Domaine* construit peut, éventuellement, posséder des éléments liés à "Contrôleur" ainsi que d'autres éléments non-identifiés.

Afin de distinguer ces éléments, nous devons établir la règle suivante : si un élément a des relations vers des éléments liés à "Vue" ainsi que des relations (invocation) vers des éléments liés au "Domaine", cet élément peut effectivement être "Contrôleur". Car un élément approprié avec les caractéristiques demandées pour être un "Contrôleur" gère les interactions entre des éléments modèle et des éléments vue. Prenant en compte cette règle, certains éléments seront distingués. Ces éléments sont des éléments liés au "Contrôleur".

6.1.3 Deuxième exemple sur le point de vue du domaine métier

Nous proposons un autre exemple concernant le domaine métier avec cette fois les propriétés suivantes :

Facette : Domaine d'Application.

Point de vue : Domaine métier.

Catégorie : *Découverte.*

Technique : Raffinement.

Description :

$$V = Application\ (\ PdVDM\ (\ algorithme), V1,\ Raffinement,\ V1.HorsDomaine). \qquad (6.3)$$

Comme mentionné auparavant, il existe plusieurs raisons conduisant un utilisateur à vouloir raffiner une vue construite. L'utilisation du point de vue du domaine métier défini dans la section 6.1.1 a permis de construire une vue contenant un élément architectural nommé *Hors-Domaine.* Cependant, nous présentons un point de vue permettant de raffiner cette vue construite selon les principes donnés au-dessus.

Les primitives doivent satisfaire l'objectif de "découverte" des nouveaux résultats, ainsi que fournir des liens de raffinement afin de montrer que la vue construite est un affinage d'une autre vue. Les primitives proposées sont les suivantes :

PE5: discoveryBySimilarityCharacteristic: threshold ;

TP2: refinementTechnique: scope ;

L'algorithme proposé prend en compte ces primitives de la façon suivante. L'élément *Hors-Domaine* est considéré comme une vue, et on ne cible que cet élément.

Algorithm 3 construction d'une vue architecturale utilisant le point de vue DM proposé

Require: La vue entrée horsDomaine

V1 := `horsDomaine discoveryBySimilarityCharacteristic: threshold`

V2 := `impV refinementTechnique: scope.`

Dans notre cas d'étude, la vue construite selon le point de vue correspondance (proposée dans la section 6.1.1) possède trois éléments architecturaux précisés et un élément architectural *Hors-Domaine.* Ce dernier couvre une partie assez importante de la vue étudiée. En utilisant le point de vue *découverte* proposé, les résultats obtenus valident l'efficacité de notre approche en découvrant de nouveaux éléments architecturaux.

Dans le cas d'étude, BeeEye a découvert six nouveaux éléments architecturaux. Ainsi que l'élément architectural *Hors-Domaine* qui regroupe des éléments restés non-identifiés. Par contre le taux de cette partie non-identifiée est éventuellement plus faible que l'élément *Hors-Domaine* de la vue étudiée. Ces nouveaux éléments sont construits utilisant le critère similarité textuelle. L'utilisateur du framework n'avait pas de connaissance sur l'existence de ces concepts dans le système étudié.

Les résultats obtenus en appliquant cet algorithme sont montrés dans le tableau 6.8. Ce dernier illustre chaque élément découvert, le poids de chaque élément et la propagation de chaque élément dans la vue construite.

élément construit	poids d'élément	propagation
Loan	3	3.4 %
Banking	4	4.54 %
Insurance	9	10.22 %
Investment	2	2.27 %
Service	7	7.95 %
Benefit	6	6.81 %
Hors-Domaine	16	18.18 %

TABLE 6.8 – Des éléments découverts, le poids de chaque élément et la propagation de chaque élément selon le PdVDM

Sans considérer les enchaînements verticaux, la figure 6.3 montre un enchaînement horizontal du framework. La première partie consiste à construire la vue V1 à partir de la vue d'implémentation et la deuxième consiste à faire un raffinement à partir de cette vue construite. Cette dernière nous a fourni la vue V2 obtenue utilisant le nouveau point de vue.

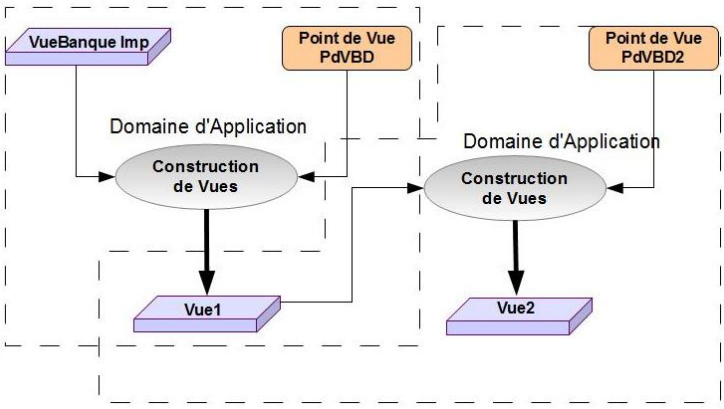

FIGURE 6.3 – Deux utilisations de BeeEye : une abstraction suivie d'un raffinement

6.1.4 Deuxième exemple sur le point de vue patron MVC

Nous proposons un autre exemple sur le point de vue du *patron MVC* afin de l'appliquer sur une vue construite utilisant le point de vue défini dans la section 6.1.3. Cette vue représente la facette *Domaine d'Application*. Nous souhaitons montrer comment deux facettes de différentes natures peuvent être composées. L'objectif est de composer la facette *Domaine d'Application* et la facette *Patron logiciel*. Pour cela, un point de vue concernant le *patron MVC* est défini. Ainsi, ce point de vue est appliqué sur l'élément architectural *Hors-Domaine* de la vue construite par le point de vue de la section 6.1.3. Cela permet ainsi d'avoir une autre vision sur les éléments restant non-identifiés de cette vue par rapport à une nouvelle facette.

Ce nouveau point de vue est défini selon les propriétés suivantes :

Facette : Patron Logiciel

Point de vue : Patron MVC

Catégorie : *Correspondance*

Technique : Composition

Description :

$$V = Apply\ (\ PdVMVC(AV,\ algorithme), V1,\ Composition,\ V1.HorsDomaine). \tag{6.4}$$

L'algorithme de ce point de vue est défini en utilisant les primitives suivantes :

PE1: mappingBySymbol: abstractView ;

PE2: mappingBySynonym: abstractView and: dictionary ;

PE4: notMapping: abstractView ;

OP1: unionOperator: view ;

TP3: compositionTechnique: scope.

L'algorithme prend en compte les primitives PE1, PE2 et PE4 afin de construire de nouveaux éléments architecturaux. Ensuite la primitive TP3 sera appliquée afin d'établir des liens de composition entre les éléments de la vue étudiée et de la vue construite.

Algorithm 4 Construction d'une vue architecturale utilisant le point de vue MVC proposé

Require: La vue entrée (IV HorsDomaine), La vue abstraite , Le dictionaire

```
V1 := horsDomaine mappingBySymbol: AV
V2 := horsDomaine mappingBySynonym: AV and: dict
V3 := horsDomaine notMapping: AV
V4 := V1 unionOperator: V2
V5 := V3 unionOperator: V4
V5 := impV compositionTechnique:\ scope.
```

La vue abstraite a été composée par l'architecture du patron *MVC*. Les résultats obtenus par la construction de cette vue sont présentés dans le tableau 6.9.

L'élément *Hors-Domaine* de la vue étudiée est une abstraction d'une partie des éléments de la vue d'implémentation selon la facette *Domaine d'Application*. Après réalisation de ce point de vue, nous avons étudié, à l'intérieur de cet élément, trois éléments représentant le patron MVC et un élément contenant des éléments non-identifiés selon ce patron. En effet l'élément étudié représente maintenant une composition de deux facettes considérées par l'approche.

Le poids de l'élément étudié est égal à la somme des poids d'éléments architecturaux construits. En regardant les métriques, nous découvrons que le taux de l'élément architectural *Hors-Domaine*

élément construit	poids d'élément	propagation
Modèle	11	68 %
Vue	5	31.25 %
Contrôleur	0	0
Hors-Domaine	1	6.25 %

TABLE 6.9 – Les éléments construits selon la technique de composition

est faible. Cela montre que le nouveau point de vue appliqué a pu découvrir les concepts souhaités dans la vue abstraite. Cependant, nous ne pouvons pas déduire que cet élément étudié (en tant que sous-système) respecte le *patron MVC*, avant de faire des analyses plus approfondies afin de découvrir les éléments liés au "contrôleur". Nous avons proposé certaines pistes, effectivement des nouvelles primitives, dans le chapitre suivant afin de découvrir ces éléments. Cela dit, notre objectif de définir ce point de vue a été de montrer comment notre approche permet de composer deux points de vue.

La figure 6.4 montre un enchaînement horizontal fourni par les deux constructions appliquées. La première consiste à construire la vue obtenue à partir de la vue d'implémentation, et la deuxième consiste en une construction par composition à partir de cette vue construite.

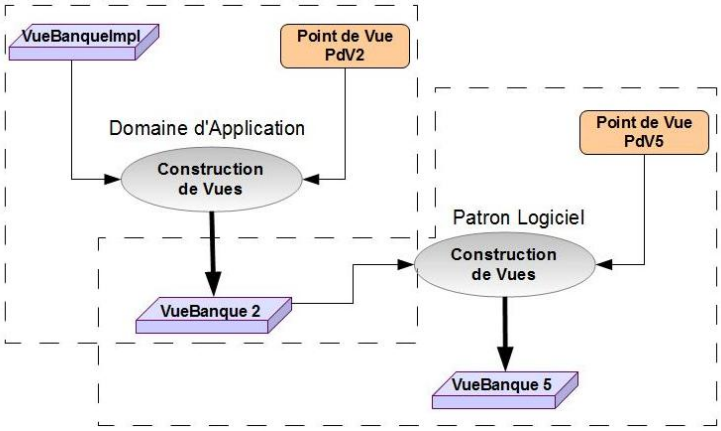

FIGURE 6.4 – Deux utilisations du BeeEye montrant la composition entre deux facettes

6.2 Conclusion

Nous avons présenté dans ce chapitre plusieurs mises en pratique de notre approche (et du framework) BeeEye. Les propositions faites peuvent être utilisées telles quelles par un utilisateur de BeeEye. Ces propositions nous ont permis de montrer la flexibilité de l'approche, car l'utilisateur n'a qu'à choisir la vue étudiée et un point de vue défini pour construire une vue architecturale. Ainsi nous avons montré précisément le déroulement de la définition d'un point de vue. Ceci valide également la flexibilité de BeeEye en terme de construction de nouveaux points de vue par un utilisateur de l'approche.

Nous avons également proposé un certain nombre de métriques qui peuvent être appliquées sur les résultats obtenus par chaque construction. Quoique les fondements théoriques de BeeEye permettent de définir une approche d'ingénierie pour la définition de points de vue et la construction de vues architecturales, cependant nous pouvons analyser des vues construites. Ce qui permet d'améliorer la compréhension obtenue (par ces vues) sur un système étudié.

Nous avons également validé notre proposition quant à l'utilisation des primitives proposées. Des combinaisons différentes de ces primitives peuvent être envisagées sur n'importe quelle vue en entrée choisie par l'utilisateur.

Chapitre 7

Conclusions et Perspectives

Dans le cadre de cette thèse, nous nous sommes intéressés à la problématique de l'ingénierie de construction de vues architecturales pour un système logiciel existant. Nous avons étudié différents aspects de cette problématique en proposant :

- **Une étude générale sur l'architecture logicielle**
  Un travail d'état de l'art a été effectué afin de connaître les différents travaux dans le domaine de l'architecture logicielle. Nous avons cherché à comprendre ce qu'est une architecture logicielle à travers plusieurs références données dans un ordre chronologique de publication.
  Avant de nous lancer dans une définition de l'architecture logicielle, nous avons regardé si nous ne pouvions pas utiliser des définitions existantes. La définition proposée par la norme IEEE 1471 couvre les notions nécessaires pour une définition de l'architecture logicielle selon les principes de notre approche. C'est-à-dire que cette définition prend en compte les notions de vue, de point de vue ainsi que les rôle des utilisateurs.

- **Un état de l'art sur la reconstruction de l'architecture logicielle**
  Nous avons étudié la littérature concernant la reconstruction d'architectures logicielles et ce qu'apportent les travaux existants. Selon la définition de l'architecture logicielle choisie dans le chapitre précédent, notre étude se focalise sur les approches permettant de construire des vues architecturales. Plusieurs approches sont présentées ; ces approches se différencient selon le but considéré, l'entrée utilisée, le processus et la technique appliquée et la sortie fournie. Parmi ces différents critères donnés par [DP09], nous en avons choisi trois : (i) l'entrée, (ii) la technique de construction et (iii) le processus de construction. Les raisons pour lesquelles nous avons décidé de focaliser notre étude autour de ces trois critères sont :
  - le but général de notre approche est d'améliorer la compréhension d'un système existant dans les phases de maintenance et d'évolution. Nous n'allons donc pas focaliser sur ces approches selon leurs objectifs.
  - Lors du choix de notre définition de l'architecture logicielle, nous avons argumenté de l'importance d'avoir des vues multiples à partir d'un système existant. Ainsi, nous avons expliqué dans le chapitre 4 que la démarche proposée par notre approche permet d'utiliser une vue construite en tant que l'entrée d'une autre construction. En effet, lors que nous étudions les entrées architecturales, nous étudions également les sorties. L'objectif de notre approche ne s'inscrit pas dans les vues construites (comme les sorties) mais sur la façon avec laquelle nous permettons de construire ces vues.

  La prise en compte des attentes des utilisateurs telles que l'architecture imaginée est un chal-

lenge dans ce domaine. Selon le critère technique, les approches de reconstruction se répartissent majoritairement en deux groupes : le premier groupe est constitué par les approches utilisant les techniques de constructions manuelles. Ces approches imposent la présence d'un ou plusieurs experts en offrant des informations. Par contre, elles sont coûteuses en expertise. Le second groupe est constitué par les approches utilisant les techniques automatiques (semi-automatiques et quasi-automatiques). Ces approches sont plus faciles à utiliser, mais elles prennent peu en compte les informations/attentes des utilisateurs. Effectuer une corrélation entre les informations et les attentes d'utilisateurs en tant qu'entrée et la réalité implémentée dans le système est un aspect important à creuser.
Nous avons constaté l'existence de plusieurs approches avec les différents types d'entrées, techniques et processus pour construire l'architecture d'un système existant. Cependant, nous avons distingué le besoin d'avoir une approche générique pour l'ingénierie de la construction des vues architecturales. Une démarche générique permet de construire, abstraire, raffiner et composer ces vues architecturales. Ainsi nous avons remarqué que peu de ces approches profitent des avantages d'utiliser les différents types d'informations. Ce qui nous semble être possible en se basant sur des points de vue. Par contre les points de vue doivent être définis d'une façon claire et utilisable à n'importe quel niveau d'abstraction.

- **une approche générique de construction des vues basée sur des points de vue : BeeEye**
  A partir de l'analyse des travaux existants, nous avons proposé une approche permettant de construire des vues architecturales à partir d'un système à objets : **BeeEye**. L'objectif de notre approche BeeEye est de proposer une démarche de rétro-ingénierie générique permettant d'obtenir des vues architecturales à des niveaux d'abstraction plus élevés que le niveau implémentation et permettant de combiner des processus de construction, des techniques proposées et des vues construites de plusieurs façons selon des attentes des utilisateurs. BeeEye est générique pour être utilisée avec tous les systèmes logiciels à objet pour lesquels nous disposons du code source.
  L'approche a les caractéristiques suivantes :
  - une approche générique : elle est basée sur des concepts importants afin de couvrir des techniques existantes dans le domaine de la reconstruction des vues architecturales. L'approche est générique car elle permet de fournir des représentations de n'importe quel système objet selon les perspectives d'utilisateurs en se basant sur les primitives de construction génériques.
  - une approche flexible et ouverte : elle permet d'avoir différents processus de construction. Ces processus peuvent être combinés de différentes manières selon des attentes d'utilisateurs.
  - une approche supportant la ré-utilisabilité : elle permet de réutiliser des vues construites par l'approche en tant que des nouvelles entrées dans une nouvelle construction. Ainsi cette contrainte est considérée sur la définition d'un point de vue de l'approche.
  - une approche interactive : elle permet de tenir compte des connaissances des utilisateurs.

  L'approche BeeEye est basée sur deux concepts : vue et point de vue ; les vues en tant que représentation d'un système étudié et les points de vue en tant que guides de construction. Le framework BeeEye est proposé afin de permettre la construction de vues architecturales : ce framework prend en compte une vue concrète en entrée et un point de vue afin de construire une autre vue concrète. En effet, ce framework est établi afin de manipuler et supporter deux types d'entrée : la vue concrète représentant le système implémenté et la vue abstraite ou des caractéristiques génériques représentant des attentes et souhaits des utilisateurs.

  Dans la modélisation d'un point de vue, nous avons fait une distinction entre l'aspect algo-

rithmique de point de vue et la prise en compte des attentes de l'utilisateur. Cette distinction a ajouté la performance de l'approche sur différents aspects tels que : la réutilisabilité d'un point de vue défini, l'accessibilité pour des utilisateurs à différents niveaux, et la flexibilité d'un point de vue face à des modifications. En nous basant sur cette distinction entre deux aspects, nous avons donc proposé deux catégories de points de vue : *correspondance* et *découverte*. Ces deux catégories représentent également les deux groupes d'utilisateurs : ceux qui connaissent le système, et ceux qui ne le connaissent pas.

- Le premier groupe propose et définit les informations a priori connues ou imaginées dans une vue abstraite. Cette vue est utilisée dans un point de vue de catégorie des points de vue correspondance. Le principe de cette catégorie est de construire une vue concrète en mettant en relation des entités (éléments et relations) de la vue en entrée et des entités de la vue concrète étudiée en fonction des entités de la vue abstraite.
- En absence d'une représentation abstraite, nous avons proposé la catégorie des points de vue découverte. Les utilisateurs souhaitent avoir certaines caractéristiques dans une vue construite. En définissant une caractéristique comme étant un objectif à retrouver et à rechercher parmi des entités de la vue en entrée, l'algorithme de construction d'un point de vue *découverte* effectue une identification des entités de cette vue. Cette identification permet la découverte des éléments respectant la caractéristique recherchée. Les caractéristiques considérées sont basées sur des critères génériques tels que : découverte par la similarité textuelle entre des éléments, découverte d'une nouvelle vue concrète basée sur le niveau d'interaction entre des éléments et découverte de la fonctionnalité des éléments de la vue étudiée.

Nous avons proposé différentes techniques de construction : abstraction, raffinement, composition et identité. Ces techniques de construction permettent d'enchaîner des vues et d'établir des liens entre des vues construites. Les liens construits entre des vues (plus précisément entre des éléments et des relations) précisent les raisons pour lesquelles deux ou plusieurs vues sont liées. Par exemple dans une construction par raffinement, l'utilisateur souhaite affiner une vue V déjà construite car il souhaite avoir une présentation plus détaillée. Pour cela, l'approche n'impose pas de recommencer la construction à partir de la vue d'implémentation. Il suffit de prendre la vue construite V en tant qu'entrée et d'appliquer la technique de raffinement afin de construire une nouvelle vue V'. Cette vue V' construite fournit une représentation plus détaillée de V ainsi qu'une abstraction de la vue d'implémentation.

La technique de composition est proposée afin de permettre aux utilisateurs d'avoir une vue représentant deux perspectives différentes selon leurs attentes. Par exemple en utilisant la technique de composition, l'approche permet de construire une vue représentant les perspectives de Domaine d'application ainsi que les perspectives d'un patron logiciel considéré.

Utilisant les différentes techniques proposées, BeeEye permet de construire et de manipuler plusieurs vues architecturales à différents niveaux d'abstraction, et ainsi d'établir différents types de relations entre les vues : *des relations d'abstraction*, *des relations de raffinement*, *des relations de composition* et *des relations d'identité*.

– **Les primitives définies pour la construction des vues**

Nous avons proposé un certain nombre de primitives afin de permettre de l'ingénierie la construction des vues. Nous avons décrit trois types de primitives : les primitives de catégorie, les primitives de techniques et les primitives d'opérateurs. Ces primitives proposées sont

des briques de base pour notre approche permettant de construire des éléments architecturaux et des relations architecturales.
Un algorithme de construction de vues est défini, donc, en utilisant et en combinant éventuellement plusieurs primitives. Les algorithmes de construction sont génériques, c'est-à-dire qu'ils peuvent être utilisés pour tous les systèmes logiciels dont on dispose le code source.
Notre approche est ouverte et flexible sur l'aspect de la définition des nouveaux points de vue : en utilisant et combinant ces primitives proposées, nous permettons à l'utilisateur de définir un nouveau point de vue.
Les primitives concernant des opérateurs peuvent être utilisées sur les aspects de comparaison entre des vues ou combinaison entre des vues. Elles permettent de faire l'union, l'intersection et la différence entre deux ou plusieurs vues.

– **Mise en pratique et validation de BeeEye**

Dans ce chapitre, nous avons proposé quatre conditions différentes afin de définir quatre exemples de points de vue. La définition des points de vue utilise des primitives proposées. Le mécanisme de définition d'un point de vue est montré étape par étape. Nous avons expliqué comment choisir des primitives selon les conditions données pour chaque exemple. Ensuite, nous avons validé ces points de vue sur une étude de cas afin de construire un certain nombre de vues architecturales représentant cette étude de cas.
Le premier exemple de point de vue est défini sur le domaine du métier, et le second point de vue est défini sur le patron "Modèle-Vue-Contrôleur". L'intérêt de la définition de ces deux points de vue a été d'illustrer la démarche de construction des vues architecturales utilisant les primitives de construction proposées par l'approche. Ce qui a permis également de construire deux vues architecturales qui représentent les perspectives des facettes conceptuelle (domaine du métier) et structurelle (patron Modèle-Vue-Contrôleur).
Un autre point de vue est proposé sur le domaine du métier ; mais cette fois nous utilisons des primitives concernant la technique de raffinement. Nous avons montré comment l'approche permet de combiner deux processus de construction de vues, ce qui fournit un enchaînement horizontal de vues. Cette étape s'est focalisée sur une partie de la vue en entrée (l'élément "Hors-Domaine"), qui a fait l'objet du traitement.
Le dernier point de vue est proposé sur le patron "Modèle-Vue-Contrôleur" choisi. L'objectif est de montrer comment l'approche permet de construire un enchaînement horizontal de vues, mais cette fois en utilisant la primitive concernant la technique de composition. Ce point de vue réalise une composition sur une vue obtenue selon la facette domaine d'application pour construire une autre vue selon la facette patron logiciel. Cependant, l'utilisateur peut appliquer un autre point de vue sur la vue construite afin de continuer d'une manière horizontale ou verticale.

L'approche proposée se situe dans la définition des activités de reconstruction : où les processus, les techniques et les vues peuvent être combinés de différentes manières en fonction des attentes des différents utilisateurs concernés par la compréhension du système.

Les perspectives qui se dégagent de ce travail sont classées en deux catégories : les perspectives générales et les perspectives algorithmiques.

Perspectives générales. La validation de notre approche sur un système de grande taille est la première perspective de notre approche. Nous avons mis en pratique notre approche en utilisant un système étudié contenant environ 100 classes. La validation sur un système plus large nous semble indispensable.

Notre approche est focalisée sur une démarche de construction à partir du code source focalisant sur des analyses statiques. L'approche propose des primitives permettant la construction des vues conceptuelles. En effet, nous n'avons pas proposé des primitives afin de construire des vues d'exécution ; celles qui permettent de décrire les aspects dynamiques du système logiciel étudié. Pour cela, nous devons définir des primitives qui permettent de décrive le système étudié en terme d'éléments exécutables et des mécanismes de communication entre ces éléments exécutables. L'avantage de ces primitives est de permettre de construire des vues qui présentent des informations liées à la distribution comme l'attribution des fonctionnalités du système sur les éléments exécutables.

La démarche proposée par notre approche se concentre sur la présentation du système implémenté selon les attentes des utilisateurs. Dans cette démarche de rétro-ingénierie, nous avons constaté souvent des entités Fantômes, ou des nouvelles entités construites. Il semble que les premières soient celles qui n'existent plus dans le système. Les secondes sont celles qui existent dans le système mais sont ignorées par l'utilisateur (ces deux cas sont liés aux points de vues *correspondance*). Nous pouvons effectuer des processus afin d'optimiser la vue étudiée. Cela est possible en adaptant les primitives proposées comme la primitive "correspondance par symbole". Une fois que l'approche retrouve les entités Fantômes, elle peut proposer de fournir des documentations sur le système concernant les concepts qui n'existent pas dans le système. Une fois l'approche découvre des nouvelles entités qui ne sont pas proposées par l'utilisateur, nous pouvons considérer la possibilité de découvrir des parties de code mort, ce qui est intéressant pour les personnes effectuant les tâches de maintenance et d'évolution.

Perspectives algorithmiques. Nous avons proposé, dans le cadre de notre approche, une identification des relations architecturales selon les deux catégories des points de vue *correspondance* et *découverte*. La règle générale pour construire une relation architecturale a été l'existence d'une ou plusieurs relations appropriées à la recherche dans un niveau inférieur. Mais nous n'avons pas imposé de contraintes particulières pour définir la sémantique d'une relation de haut niveau. A ce titre, les relations architecturales sont simplement décrites pour montrer l'existence d'un échange d'information entre deux éléments de haut niveau. Cependant, nous avons certaines pistes pour proposer des sémantiques de haut niveau pour des relations architecturales entre deux éléments appartenant à une vue architecturale. Nous proposons trois relations de haut niveau fournies à partir des relations entre des éléments de la vue d'implémentation. Ces relations proposées montrent le niveau de cohésion entre des éléments de haut niveau.

Commençant à partir de la vue d'implémentation construite à partir du code source d'un système étudié, l'approche peut se baser sur des relations d'héritage et des relations d'invocations afin de définir des relations de haut niveau :

- Une *Relation Extrême* lie deux éléments architecturaux dans une vue construite lors que les ensembles associés à chaque élément dans la vue d'implémentation ont au moins une relation d'invocation, selon la condition que cette relation d'invocation soit entre les parents de deux arbres d'héritage (situés dans chaque ensemble) ou entre un fils d'un arbre avec le parent d'un autre arbre. Cette relation représente une cohésion forte entre les deux éléments construits.
- Une *Relation Intense* lie deux éléments architecturaux dans une vue construite lorsque les deux ensembles associés à chaque élément dans la vue d'implémentation ont des relations

d'invocation. Cela se réalise selon la condition que tous les éléments (classes) appartenant à la même couche d'héritage appartenant à un ensemble lié ont des invocations vers l'autre ensemble. Cette relation construite représente une cohésion moins forte entre les deux éléments construits.

– Une *Relation Faible* lie deux éléments architecturaux dans une vue construite lorsque les ensembles associés à chaque élément dans la vue d'implémentation ont des relations d'invocation. Cela se réalise à la condition qu'il existe au moins une relation d'invocation à partir d'un ensemble associé vers un autre ensemble. Cette relation construite montre qu'il existe un échange d'informations quelque part entre les deux éléments architecturaux.

Notre approche prend en compte le code source en tant qu'une vue concrète représentant le système étudié. Par défaut, nous avons pris les éléments de type classe comme l'entité de première classe de l'approche BeeEye. Cependant, nous avons rencontré certaines limitations à ce niveau. Par exemple, dans notre étude sur le patron "MVC", en regardant le code de plus près nous avons pu constater que les éléments adéquats avec l'élément recherché "Vue" sont mieux reconnaissables en analysant le nom d'éléments méthodes. Ce qui montre que les primitives proposées par l'approche doivent pouvoir s'appliquer d'une manière plus raffinée. Les primitives proposées peuvent être facilement adaptées à un niveau de granularité plus fin. Les primitives concernant la construction des éléments architecturaux sont ouvertes à être modifiées pour traiter des éléments méthodes à la place des éléments classes.

Bibliographie

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings SIGSOFT 93, ACM Software Engineering Notes*, volume 18, pages 9–20, December 1993.

[ABN04] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, November 2004.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In *ECOOP'02 : Proceedings of the 16th European Conference on Object-Oriented Programming*, volume 2374 of *LNCS*, pages 334–367, Malaga, Spain, 2002. Springer-Verlag.

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings ICSE '94*, May 1994.

[AJL93] Stafford J. A., Richardson D. J, and Wolf A. L. Aladdin : A tool for architecture level depenance analysis of software. 1993.

[AL99a] Nicolas Anquetil and Timothy Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering)*, pages 235–255, 1999.

[AL99b] Nicolas Anquetil and Timothy C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance : Research and Practice*, 11 :201–21, 1999.

[ASHS09] Razavizadeh A., Cimpan S., Verjus H., and Ducasse S. Multiple viewpoints architecture extraction. pages 14–19, 2009.

[BÖ99] Holger Bär. FAMIX C++ language plug-in 1.0. Technical report, University of Bern, September 1999.

[Bas97] Victor Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1) :3–12, 1997.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[Big89] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22 :36–49, October 1989.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley Press, 1996.

[Boe87] B.W. Boehm. A spiral model of software development and enhancement. In R.H. Thayer, editor, *Tutorial : Software Engineering Project Management*, pages 128–142. IEEE Computer Society, Washington, 1987.

[BP01] Reinder J. Bril and André Postma. An architectural connectivity metric and its support for incremental re-architecting of large legacy systems. In *Proceedings of International Workshop on Program Comprehension (IWPC'01)*, pages 269–280. IEEE CS, 2001.

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures : Views and Beyond*. Addison-Wesley Professional, 2002.

[CCI90] Elliot Chikofsky and James Cross II. Reverse engineering and design recovery : A taxonomy. *IEEE Software*, 7(1) :13–17, January 1990.

[Cha09] Sylvain Chardigny. *Extraction d'une architecture logicielle à base de composants depuis un système orienté objet*. PhD thesis, October 2009.

[CTH95] Ian Carmichael, Vassilios Tzerpos, and Rick C. Holt. Design maintenance : Unexpected architectural interactions. In *International Conference on Software Maintenance (ICSM)*, pages 134–140. IEEE CS, 1995.

[CV95] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems and Software*, 28 :117–127, 1995.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DGKR09] Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, and Lukas Renggli. Meta-environment and executable meta-language using Smalltalk : an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 8(1) :5–19, February 2009.

[DGLD05] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose : a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[DHK+04] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony : View-driven software architecture reconstruction. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 122–134, 2004.

[DLdOdlP98] J. Dueñas, W. Lopes de Oliveira, and J. de la Puente. Architecture recovery for software evolution. In *Conference on Software Maintenance and Reengineering (CSMR)*, pages 113–120, 1998.

[DM01] Lei Ding and Nenad Medvidovic. Focus : A light-weight, incremental approach to software architecture recovery and evolution. In *Working Conference on Software Architecture (WICSA)*, pages 191–201, 2001.

[DP09] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction : A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4) :573–591, July 2009.

[DSS99] Doval D., Mancoridis S., and Mitchell S. Automating clustering of software systems using a genetic algorithm. pages 73–81, 1999.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[Duc01] Stéphane Ducasse. Squeak : Un smalltalk open-source détonnant. *Programmez ! Le Magazine du Développement*, 1(33), June 2001.

[EKRW02] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO — generic understanding of programs, an overview. Fachberichte Informatik 7–2002, Universität Koblenz-Landau, 2002.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3) :210–224, March 2003.

[Fav04] Jean-Marie Favre. CacOphoNy : Metamodel-driven software architecture reconstruction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 204–213, Los Alamitos CA, November 2004. IEEE Computer Society Press.

[FHK+97] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4) :564–593, November 1997.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics : A Rigorous and Practical Approach.* International Thomson Computer Press, London, UK, second edition, 1996.

[GAK99] Yanbing Guo, Atlee, and Kazman. A software architecture reconstruction method. In *Working Conference on Software Architecture (WICSA)*, pages 15–34, 1999.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch : Why reuse is so hard. *IEEE Software*, 12(6) :17–26, November 1995.

[Gar00] David Garlan. Software architecture : a roadmap. In *ICSE – Future of SE Track*, pages 91–101, 2000.

[GK97] Jean-Francois Girard and Rainer Koschke. Finding components in a hierarchy of modules : a step towards architectural understanding. In *ICSM*. IEEE Press, 1997.

[GP95] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), apr 1995.

[GSZ04] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Working Conference on Reverse Engineering (WCRE'04)*, pages 172–181, Los Alamitos CA, 2004. IEEE Computer Society Press.

[HNS00] Christine Hofmeister, Robert L. Nord, and Dilip Soni. *Applied Software Architecture.* Addison Wesley, 2000.

[Hol01] Ric Holt. Sofware architecture as a shared mental model. In *ASERC Workshop on Software Architecture*, University of Alberta, August 2001.

[HRY95] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, Seattle, Washington USA, April 1995. Association for Computing Machinery, Inc.

[HSSW06] Holt, Schürr, Sim, and Winter. Gxl : A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2) :149–170, April 2006.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL : Towards a standard exchange format. In *Proceedings WCRE '00*, November 2000.

[IBM] IBM. Ibm - rational unified process (rup).

[IEE98] Software engineering standards committee of the ieee computer society. *1219-1998 IEEE standard for software maintenance.*, 1998.

[IEE99] IEEE Architecture Working Group. *IEEE P1471/D5.0 Information Technology — Draft Recommended Practice for Architecural Description*, August 1999.

[IEE00] IEEE. Ieee recommended practice for architectural description for software-intensive systems. Technical report, The Architecture Working Group of the Software Engineering Committee, October 2000.

[KC99] Rick Kazman and S. J. Carriere. Playing detective : Reconstructing software architecture from available evidence. *Automated Software Engineering*, April 1999.

[KDG05] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE'05)*, pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press.

[KDG07] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering : Identifying topics in source code. *Information and Software Technology*, 49(3) :230–243, March 2007.

[KMNL06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *CSMR'06*, pages 279–294, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[Kos00] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[KOV01] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines. CMU/SEI-2001-TR-026, Carnegie Mellon University, Software Engineering Institute, August 2001.

[KOV03] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines, third edition. CMU/SEI-2002-TR-034, Carnegie Mellon University, Software Engineering Institute, November 2003.

[KP96] Christian Kramer and Lutz Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of WCRE '96 (3rd Working Conference on Reverse Engineering)*, pages 208–216. IEEE Computer Society Press, November 1996.

[Kri99] Rene Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.

[Kru95] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6) :42–50, November 1995.

[KWC98] R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models : Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998. ISBN : 0-8186-89-67-6.

[Lai01] Petri K. Laine. The role of sw architectures in solving fundamental problems in object-oriented development of large embedded sw system. In *WICSA*, pages 14–23, 2001.

[Lan03] Michele Lanza. CodeCrawler — a lightweight software visualization tool. In *Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)*, pages 51–52. IEEE CS Press, 2003.

[LB85] Manny Lehman and Les Belady. *Program Evolution : Processes of Software Change*. London Academic Press, London, 1985.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9) :782–795, September 2003.

[LKA+95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, 1995.

[LL03] Jonas Lundberg and Welf Löwe. Architecture recovery by semi-automatic component identification. *Electronic Notes in Theoretical Computer Science*, 82(5), 2003.

[LS80] Bennett Lientz and Burton Swanson. *Software Maintenance Management.* Addison Wesley, Boston, MA, 1980.

[LSP05] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM.

[LTP04] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel : A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, pages 7–18, 2004.

[MAvdLF00] JAZAYERI M., RAN A., and van der LINDEN F. *Software architecture for product families : principles and practice.* 2000.

[McK84] J. R. McKee. Maintenance as a function of design. In *Proceedings of AFIPS National Computer Conference*, pages 187–193, 1984.

[MEG03] Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by architectural discovery and recovery. In *Proceedings of the 2nd Second International Workshop from Software Requirements to Architectures (STRAW)*, 2003.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian : An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[MJ06] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2) :225–256, 2006.

[MK01] Nabor C. Mendonça and Jeff Kramer. An approach for recovering distributed system architectures. *Automated Software Engineering*, 8(3-4) :311–354, 2001.

[MKPW06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2) :140–156, 2006.

[MM98] Spiros Mancoridis and Brian S. Mitchell. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *Proceedings of IWPC '98 (International Workshop on Program Comprehension)*. IEEE Computer Society Press, 1998.

[MM00] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000)*, pages 46–53, November 2000.

[MM01] Brian S. Mitchell and Spiros Mancoridis. Comparing the Decompositions Produced by Software Clustering Algorithms using Similarity Measurements. In *Proceedings of ICSM '01 (International Conference on Software Maintenance)*. IEEE Computer Society Press, November 2001.

[MM06] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3) :193–208, 2006.

[MMW01] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.

[MMW02] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of SEKE 2002*, pages 289–296. ACM Press, 2002.

[MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models : A case study. *IEEE Computer*, 8 :29–36, 1997.

[MNS95] Gail Murphy, David Notkin, and Kevin Sullivan. Software reflexion models : Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[Mur96] Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution.* PhD thesis, University of Washington, 1996.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose : an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[Neb99] Robb Nebbe. FAMIX Ada language plug-in 2.2. Technical report, University of Bern, August 1999.

[Par72] David L. Parnas. A technique for software module specification with examples. *CACM*, 15(5) :330–336, May 1972.

[Par94] David Lorge Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, Los Alamitos CA, 1994. IEEE Computer Society.

[PFGJ02] Martin Pinzger, Michael Fischer, Harald Gall, and Mehdi Jazayeri. Revealer : A lexical pattern matcher for architecture recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*, pages 170–178, 2002.

[PGF05] Martin Pinzger, Harald Gall, and Michael Fischer. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3) :183–196, 2005.

[Pin05] Martin Pinzger. *ArchView — Analyzing Evolutionary Aspects of Complex Software Systems.* PhD thesis, Vienna University of Technology, 2005.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52, October 1992.

[Riv00] Claudio Riva. Reverse architecting : an industrial experience report. In *Proceedings WCRE 2000*, pages 42–50. IEEE Computer Society, 2000.

[Riv04] Claudio Riva. *View-based Software Architecture Reconstruction.* PhD thesis, Technical University of Vienna, 2004.

[SAMD08] Chardigny S., Seriai A., Oussalah M., and Tamzalit D. Extraction of component-based architecture from object-oriented systems. In *WICSA.*, pages 285–288, 2008.

[Sar03] Kamran Sartipi. *Software Architecture Recovery based on Pattern Matching.* PhD thesis, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2003.

[SFM99] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44 :171–185, 1999.

[SG96] Mary Shaw and David Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SHI+01] Smolander, Hoikka, Isokallio, Kataikko, Mäkelä, and Kälviäinen. Required and optional viewpoints — what is included in software architecture ? Technical report, Univ. Lappeenranta, 2001.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[Som96] Ian Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.

[SS02] Mitchell S. and Mancoridis S. Using heurstic search techniques to extract design abstraction form source code. *GECCO.*, 2002.

[TDD00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX : Exchange experiences with CDIF and XMI. In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, June 2000.

[Tic99] Sander Tichelaar. FAMIX Java language plug-in 1.0. Technical report, University of Bern, September 1999.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[TSP96] Scott R. Tilley, Dennis B. Smith, and Santanu Paul. Towards a framework for program understanding. In *WPC '96 : Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 19. IEEE Computer Society, 1996.

[WCK99] Steven G. Woods, S. Jeromy Carrière, and Rick Kazman. The perils and joys of reconstructing architectures. *SEI Interactive, The Architect*, 2, September 1999.

[WD01] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International Workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[YHC97] A.S. Yeh, D.R. Harris, and M.P. Chase. Manipulating recovered software architecture views. In *Proceedings of International Conference Software Engineering (ICSE'97)*, 1997.