Chapitre 9

Assistance à l'évolution du logiciel dirigée par la qualité

L'évolution de l'architecture d'un logiciel à base de composants peut avoir des conséquences nuisibles sur ses attributs qualité. Sans une définition explicite de ces attributs non-fonctionnels et de leurs liens avec les décisions de conception prises lors de la description d'architecture, le développeur ne peut pas se rendre compte des conséquences de l'évolution sur la qualité de son système. Nous proposons dans ce chapitre un état de l'art sur les techniques utilisées dans la littérature pour capturer ce type de connaissances architecturales. Le lecteur sera donc informé des différentes méthodes de documentation des décisions architecturales, des contraintes architecturales et des attributs qualité. Nous présentons également quelques travaux sur l'assistance à l'évolution d'un logiciel, à la fin de cet état de l'art. Nous terminons ce chapitre par introduire notre approche de documentation de ces aspects logiciels à travers la notion de contrat d'évolution, ainsi que notre méthode d'assistance à l'évolution basée sur ces contrats.

Chapitre rédigé par Tarek ZERNADJI, Chouki TIBERMACINE, Régis FLEURQUIN et Salah SADOU .

9.1. Introduction : contexte et motivations

Durant son cycle de vie, un logiciel doit nécessairement subir un certain nombre de modifications qui le feront évoluer au cours du temps. Cette évolution est la conséquence naturelle des réponses aux besoins changeants, imposés par les utilisateurs ainsi que l'environnement avec lequel le système logiciel interagit ou dans lequel il s'exécute. Un aspect-clé de l'évolution du logiciel est l'évolution de son architecture. En effet, l'architecture logicielle est l'un des premiers artéfacts du processus de conception. Elle permet de représenter les décisions architecturales et leurs raisons, ce qui facilite considérablement la compréhension de la structure du logiciel et de ses origines (les besoins utilisateurs). Elle représente le premier point dans lequel ces décisions peuvent être analysées et évaluées [BAS 03].

L'évolution peut viser deux aspects : fonctionnel et non fonctionnel (qualitatif). Le premier concerne l'ajout, le retrait ou la modification de fonctionnalités, tandis que le second s'intéresse aux qualités que le logiciel doit refléter dans son architecture. Il est admis que les décisions architecturales sont conduites par les attributs qualité requis dans les documents de spécification [BAS 03]. En effet, ce ne sont pas les fonctionnalités attendues d'un logiciel qui déterminent principalement son architecture, mais plutôt la manière avec laquelle ses fonctionnalités vont être fournies qui permet de façonner et d'élaborer cette architecture. Ainsi, en voulant améliorer certaines qualités du logiciel telles que la maintenabilité, la performance ou la portabilité par l'introduction de nouvelles décisions architecturales, ou en voulant simplement ajouter, supprimer ou modifier une fonctionnalité, on pourra involontairement toucher d'autres décisions précédemment prises, et par conséquent certaines qualités peuvent être affectées. Ces problèmes sont souvent posés lors de la phase de maintenance. Le manque d'informations sur les décisions qui ont menées à l'architecture du logiciel, et d'une définition explicite des liens qui existent entre les caractéristiques non fonctionnelles et les décisions architecturales qui les implémentent en font l'une des principales causes. En effet, en l'absence d'une documentation précise décrivant les choix qui sont entrepris par les développeurs et qui justifient la forme de l'architecture, d'inévitables allers/retours se produisent entre la phase de test (de non-régression) et la phase de développement. Cela implique, une séquence d'itérations causant ainsi une augmentation du coût global du développement logiciel.

Il est donc très utile de pouvoir contrôler et maîtriser ce type d'évolution en assistant le mainteneur du logiciel tout au long de son activité. Nous proposons dans ce chapitre une approche de documentation basée sur le concept de contrat [TIB 06b]. Cette documentation regroupe les liens unissant les décisions architecturales et les attributs qualités qu'elles implémentent. Des améliorations ont été proposées dans la spécification du contrat (section 9.3.1). L'objectif est d'avoir une documentation plus raffinée contenant davantage d'informations pour mieux guider le mainteneur durant l'évolution du logiciel. La documentation (contrat) est exploitée par un algorithme d'assistance (section 9.3.2). Ce dernier, assiste le mainteneur à chaque pas d'évolution

en l'informant à la volée des conséquences des changements sur la cohérence globale de l'architecture vis à vis de la qualité. Le but est d'aboutir à chaque pas d'évolution, à minimiser les effets négatifs sur l'ensemble des qualités préalablement établies pour des architectures logicielles construites avec des composants.

Nous nous sommes focalisés dans ce chapitre sur la présentation d'un état de l'art sur l'ensemble des approches couvrant les aspects logiciels. Nous commençons par les méthodes de documentation des attributs qualité (section 9.2.1). Nous abordons ensuite les techniques utilisées pour la documentation et la formalisation de décisions architecturales (section 9.2.2). Finalement, nous clôturons l'état de l'art par des travaux sur l'assistance à l'évolution d'un logiciel (section 9.2.3). Nous terminons ce chapitre, par la présentation de notre approche pour l'assistance à l'évolution guidée par la qualité dans la section 9.3.

9.2. Etat de l'art

Le terme « qualité logicielle » est un concept complexe qui expose plusieurs facettes. Parmi ces facettes, on trouve la qualité d'un produit logiciel, la qualité d'un processus logiciel, qui vise les différentes phases de développement logiciel, ou bien aussi la qualité des ressources logicielles. Nous présentons dans cette section un panorama sur les différentes méthodes de documentation de la qualité d'un produit logiciel. Deux approches sont abordées, celle des modèles de qualité visant à caractériser la qualité d'un produit logiciel fini, et celle qui fait usage des architectures logicielles, qui sont considérées comme un produit logiciel spécifique, pour documenter les propriétés de qualité.

9.2.1. *Documentation de la qualité logicielle*

Le terme « qualité » vient du latin *qualitas*, issu du mot *qualis* qui veut dire « quel ». Il signifie la nature d'un élément. La qualité porte sur des caractéristiques non quantifiables, ce qui l'oppose linguistiquement à la quantité. Le terme « qualité » a été associé au logiciel depuis longtemps. Dès l'apparition des toutes premières méthodes de développement logiciel, on a commencé à identifier les indicateurs pouvant aider à donner une appréciation globale d'un logiciel.

Les recherches menées dans la modélisation de la qualité ont produit, durant les trente dernières années, une multitude de modèles de qualité qui ont été appliqués à différents degrés de succès. Malgré la diversité et l'hétérogénéité des modèles de qualité existants, il n'existe pas une définition claire de ce qui est un modèle de qualité. Cela provient du fait que ces derniers s'appliquent dans des contextes différents et ont des objectifs assez éloignés les uns des autres. Dans [DEI 09], un modèle de qualité est défini comme étant un modèle qui a pour objectif de décrire, d'évaluer

et/ou de prédire la qualité logicielle. Cette définition propose donc la classification des modèles de qualité selon trois objectifs différents à savoir la définition, l'évaluation, et la prévision de la qualité logicielle, séparant ainsi ces modèles en trois catégories : les modèles de définition et les modèles d'évaluation et les modèles de prédiction. Le travail de [KLA 09] fournit un schéma de classification plus étendu d'une large gamme de modèles de qualité, les plus connus dans la littérature. Les auteurs fournissent cinq dimensions pour la classification s'inspirant du Template GQM (*Goal/Question/Metric*) qui sont :

– l'objet : spécifie ce qui est examiné par le modèle de qualité. Les principales classes d'objets sont les produits, les processus et les ressources ;

– l'objectif : spécifie l'intention ou la motivation de la modélisation de la qualité (spécifier, mesurer, évaluer, contrôler, prédire, améliorer, gérer, etc.) ;

– la focalisation sur la qualité : spécifie la caractéristique qualité modélisée (exemple la fiabilité du produit logiciel) ;

– le point de vue : précise la perspective dans laquelle la caractéristique qualité est modélisée. Le point de vue peut être par exemple celui du développeur ou celui de l'utilisateur ;

– le contexte : spécifie l'environnement dans lequel la modélisation de la qualité est réalisée.

En se fixant les deux premières dimensions (l'objet : les produits, et l'objectif : spécifier, définir, contrôler, améliorer et gérer), nous allons donc nous limiter à présenter dans la suite de cette section les principales approches pour la documentation de la qualité d'un produit logiciel.

9.2.1.1. *Modèles de qualité*

Les modèles de qualité les plus connus sont basés sur une approche de décomposition communément connus sous le nom de modèle de qualité FCM (*Factor-Criteria-Metric*). Ils sont généralement conçus de manière arborescente où le niveau supérieur de la hiérarchie définit des attributs qualité abstraits de haut niveau, et le niveau inférieur définit des critères de qualité concrets qui peuvent être mesurés par des métriques.

L'un des modèles les plus connu est celui de McCall [MCC 77]. Il intègre onze facteurs couvrant trois perspectives pour définir et identifier la qualité d'un produit logiciel. La première perspective représentée par la maintenabilité, la flexibilité et la testabilité couvre la révision du produit logiciel (aptitude à subir des changements). La deuxième concerne la transition de produits qui définit l'adaptation du produit logiciel aux nouveaux environnements. Elle est représentée par la portabilité, la réutilisabilité et l'interopérabilité. La dernière perspective s'intéresse aux opérations des produits. Elle comporte les attributs suivants : correction, efficacité, fiabilité, intégrité et utilisabilité.

Le modèle décrit donc ces facteurs dans une hiérarchie de vingt-trois critères qualité. Les onze facteurs décrivent la vue externe du logiciel, telle que perçue par les utilisateurs. Les vingt-trois critères décrivent la vue interne du logiciel, telle que perçue par les développeurs. Le dernier niveau de cette décomposition représente les métriques qui sont associées aux critères. Ils sont utilisés comme méthode de mesure et visent à capturer certains aspects des critères de qualité.

Le deuxième des modèles fondateurs est celui de Barry Boehm [BOE 76] proposé en 1976. Il ressemble à celui de McCall dans le sens où il utilise la même méthode de décomposition pour caractériser les attributs qualité. C'est un modèle hiérarchique structuré sur trois niveaux. Le niveau supérieur répond aux préoccupations de l'utilisateur du logiciel :

– l'utilité du logiciel, de trois points de vue : efficacité, facilité d'utilisation et fiabilité ;

– la maintenabilité ;

– la portabilité.

Le niveau intermédiaire représente sept attributs qualité qui ensemble décrivent les qualités attendues du système logiciel : portabilité, fiabilité, efficacité, utilisabilité, testabilité, compréhensibilité et flexibilité. Le dernier niveau dans le modèle de Boehm représente les métriques associées aux caractéristiques du niveau précédent. Ce modèle est fondé sur une large gamme de caractéristiques comparé à celui de McCall et se focalise particulièrement sur la maintenabilité.

FURPS [GRA 92] est un autre modèle qui a été proposé plus tard par Robert Grady chez HP (Hewlett-Packard), moins connue que les deux modèles précédents. Il a connu une extension par IBM Rational Software vers FURPS+. FURPS signifie : *Fonctionnality, Usability, Reliability, Performance and Supportability*. Le modèle décompose les caractéristiques en deux catégories de besoins différentes :

i) fonctionnels : représentés par la caractéristique *Functionnality* qui désigne un ensemble de fonctionnalités, et inclut la caractéristique de qualité « sécurité » ;

ii) non fonctionnels (URPS), représentés par les caractéristiques suivantes :

– utilisabilité : peut inclure les facteurs humains, l'esthétique, la documentation d'utilisateur, etc ;

– fiabilité : peut inclure la fréquence et la sévérité de la panne, la récupération après la panne, la prévisibilité et le temps moyen entre les pannes (en anglais, MTBF (*Mean Time Between Failures*)) ;

– performance : inclut l'efficacité, le temps de réponse, l'utilisation de ressources, etc ;

– supportabilité : inclut la maintenabilité, testabilité, compatibilité, adaptabilité, etc.

En 1996, Geoff Dromey [DRO 95, DRO 96] présente un nouveau modèle de qualité lié aux produits logiciels, qui ressemble à ses prédécesseurs. La vision de Dromey, est que l'on ne peut pas construire des attributs qualité de haut niveau, tel que la maintenabilité ou la fiabilité directement dans un logiciel. Au lieu de cela, on peut identifier et construire un ensemble cohérent de propriétés ou caractéristiques internes tangibles (de bas niveau). Ces dernières déterminent et manifestent des attributs qualité externes de haut niveau. L'auteur a précisé trois principaux éléments pour un modèle de qualité générique :

– les propriétés du produit qui influence la qualité ;

– des attributs qualité de haut niveau ;

– les moyens pour lier les propriétés du produit avec les attributs qualité.

Une propriété de produit dans le modèle de Dromey est liée à un composant d'un type de produit dans le développement logiciel, en partant de la spécification de besoins, jusqu'à l'implémentation. La violation de ces propriétés entraînera des défauts de qualité dans le produit (les attributs qualité de haut niveau sont dans ce cas affectés). Un produit est constitué de plusieurs composants. Certains peuvent être simples, d'autres sont eux aussi constitués d'un ensemble de composants plus simples. Le modèle recense quatre types de propriétés de produit. Pour chaque type de propriété il définit un nombre d'attributs qualité qui sont influencés par ces derniers :

– propriétés d'exactitude (*Correctness*) : définissent les propriétés à respecter, soit directement par le composant ou par une composition de composants (un contexte), pour fonctionner correctement. A titre d'exemple, une variable dans une implémentation (produit) est un composant qui peut avoir comme propriété « attribuée » ou « précise ». Si elle ne porte pas l'une de ces propriétés, l'exactitude peut être affectée. Ce type de propriétés influence les attributs qualité « fonctionnalité » et « fiabilité » ;

– propriétés internes : précisent la forme normale d'un composant qui définit son intérieur (structure) et qui ne doit pas être violée quel que soit le contexte. A titre d'exemple, le corps d'une boucle doit assurer une progression vers la terminaison. Ces propriétés mesurent si un composant a été bien déployé ou composé. Ils influencent les attributs qualité, « efficacité », « maintenabilité », « fiabilité » ;

– propriétés contextuelles : ces propriétés sont associées aux composants individuels et traitent les problèmes de qualité qui découlent d'une composition d'un grand nombre de composants. Ils influencent les attributs qualité, « réutilisabilité », « portabilité », « maintenabilité », « fiabilité » ;

– propriétés descriptives : définissent si un produit logiciel est facile à comprendre et à utiliser. A titre d'exemple, donner un nom à une variable qui n'est pas évocateur

peut affecter ses propriétés descriptives. Ils influencent les attributs qualités, « utilisabilité », « réutilisabilité », « portabilité », « maintenabilité ».

Les liens entre les propriétés et les attributs qualité ne sont pas établis formellement, mais se basent sur la classification des propriétés. A titre d'exemple, pour qu'un produit satisfasse ses fonctionnalités, de manière fiable, les propriétés d'exactitude de tous ses composants doivent être satisfaites. Ainsi, ces dernières influences les attributs qualité, « fonctionnalité » et « fiabilité ». Il est à noter que les liens que l'on vient de mentionner pour chaque type de propriété, sont propres au modèle de qualité d'implémentation. Le modèle a été appliqué principalement sur des produits d'implémentation, mais il est générique et permet de construire des modèles de qualité pour les besoins ou la conception.

Kitchenham *et al.* [KIT 97] ont proposé un modèle de qualité nommé SQUID (*Software QUality In Development*) ayant une structure hiérarchique encapsulant et s'inspirant de la première version du modèle ISO/IEC 9126 [ISO 01] et du modèle de McCall. C'est un modèle composite reflétant les différents aspects d'un modèle de qualité sous forme de composants représentant la structure et le contenu de ce dernier. Il contient les éléments des deux modèles (caractéristiques de qualité, sous-caractéristiques de qualité, propriétés logicielles internes qui influent sur les sous-caractéristiques et propriétés mesurables). Les auteurs du modèle précisent que la philosophie de SQUID est que l'on ne peut pas obtenir une spécification des exigences de qualité seulement en faisant référence à un modèle de qualité, mais à partir de la spécification du comportement voulu d'un produit particulier. Par conséquent, un troisième composant est nécessaire, outre la structure et le contenu, pour répondre aux exigences de qualité, qui est celui d'un modèle de qualité du produit. Celui-ci est l'instanciation d'un modèle de qualité d'un produit particulier. Dans ce modèle tous les éléments sont mesurables par l'affectation de métriques et de valeurs. L'approche SQUID pour la modélisation de la qualité logicielle est dotée d'un ensemble d'outils permettant la spécification des exigences de qualité.

L'ISO a fourni en 2001 une nouvelle version du standard ISO/IEC 9126 [ISO 01] pour l'évaluation de produits logiciels : caractéristiques qualité et les directives pour leur utilisation. Le standard est basé sur le modèle de McCall et de Boehm. En plus d'être structuré de la même manière que ces modèles il inclut la fonctionnalité comme caractéristique ainsi que l'identification des caractéristiques de qualité internes et externes d'un produit logiciel. Il comporte quatre parties :

– modèle de qualité ;

– métriques externes ;

– métriques internes ;

– métriques de la qualité en utilisation.

Le modèle de qualité définit une hiérarchie de caractéristiques qualité (facteurs) sur deux niveaux, les caractéristiques de qualité (capacité fonctionnelle, portabilité, maintenabilité, efficacité, utilisabilité, fiabilité) et leurs sous-caractéristiques correspondantes.

Malgré cette diversité de modèles FCM proposés et leur popularité, ils ont montré quelques limites et ont reçu plusieurs critiques [BRO 06, DEI 09, DEI 07, KIT 97, MAR 04]. Ils ont donc échoué à établir une base acceptable pour l'évaluation de la qualité. La raison pour cela est de vouloir condenser des attributs qualité aussi complexes que la maintenabilité en une valeur unique et le fait que ces modèles se limitent généralement à un nombre fixe de niveaux. La plupart de ces modèles souffrent de l'absence de directives et de critères de décomposition des concepts de qualité complexe ce qui rend difficile leur raffinement ainsi que leur localisation dans certains modèles de qualité de grande taille (exemple l'utilisabilité ne peut pas être décomposée vers des propriétés mesurables en seulement deux étapes suivant les trois niveaux des modèles FCM). On reproche aussi à ce type de modèles de ne pas être capable de trouver les vraies causes des problèmes de qualité durant l'analyse ou l'évaluation d'une conception orientée objet. Cela est dû à la signification des valeurs de métriques qui reflètent la présence d'un problème (les symptômes) de conception ou d'implémentation et non pas le problème lui-même ce qui rend difficile le traitement [MAR 04].

D'autres travaux ont été menés au niveau du SEI (*Software Engineering Institute*) qui ont abouti en 2002 à un modèle d'un autre type qui propose une caractérisation des attributs qualité d'une architecture logicielle structurés en trois classes [BAS 03]. La première classe concerne les qualités du système et comporte les attributs suivants : disponibilité, modifiabilité, performance, sécurité, testabilité et utilisabilité. Les auteurs précisent que d'autres attributs peuvent être trouvés dans la taxonomie des attributs qualité et ajoutés, comme par exemple la portabilité, capturée en tant que modification de la plate-forme. La deuxième classe s'intéresse aux qualités d'affaire (*Business qualities*) et recense quelques attributs qui sont : le temps de mise sur le marché (*Time To Market*), le coût et les bénéfices, la durée de vie projetée du système, le marché visé, l'ordonnancement de la distribution du produit (*Rollout Schedule*) et l'intégration aux anciens systèmes. La troisième classe concerne les qualités directement liées à l'architecture : l'intégrité conceptuelle, la correction et la complétude, et la capacité de construction architecturale (*Buildability*). Toutes ces qualités représentent des objectifs pour l'architecte logiciel.

Les auteurs ont constaté parmi les problèmes soulevés, que les définitions de ces attributs qualité ne sont pas opérationnelles et peuvent se chevaucher, ne reflétant pas ainsi le contexte dans lequel ils sont appliqués. A titre d'exemple, tous les systèmes sont modifiables pour un certain nombre de changements et ne le sont pas pour d'autres. Ou encore, à quelle qualité doit-on classer un aspect, comme la panne d'un système (disponibilité, sécurité ou utilisabilité). Pour remédier à ce problème, ils ont

proposé un mécanisme pour caractériser les attributs qualité à travers les scénarios d'attributs qualité qui sont constitués de six parties :

– source du stimulus : c'est l'entité qui à générée le stimulus (humain, un système logiciel, ou tout autre déclencheur) ;

– stimulus : c'est un événement à prendre en considération quand il arrive au système ;

– environnement : représente les conditions sous lesquelles le stimulus se produit ;

– artéfact : représente les parties stimulées dans le système, ou le système entier ;

– réponse : c'est l'activité entreprise après l'arrivée du stimulus ;

– mesure de la réponse : la réponse doit être mesurable, de telle sorte on peut tester le besoin non fonctionnel.

Plus récemment, en 2003, Georgiadou *et al.* propose GEQUAMO (*GEneric QUAlity MOdel* [GEO 03]). Ce modèle encapsule les exigences de différentes parties (développeur, utilisateur, etc.) de manière dynamique et flexible permettant ainsi à chacun d'entre eux de construire et de personnaliser son propre modèle, qui reflète l'importance de chaque exigence selon son point de vue. C'est un modèle multiniveaux qui se construit en utilisant une combinaison de deux types de diagrammes à savoir, les CFD (*Composite Features Diagramming*) et les diagrammes de Kiviat. Les CFD fournissent un moyen qualitatif pour décrire le profil d'un élément sous évaluation. Ils se composent d'un ensemble de cercles concentriques qui expriment le niveau de détail des sous-caractéristiques de plus en plus bas en allant vers l'extérieur. Les caractéristiques et les sous-caractéristiques sont construites graduellement avec moins de détail de manière arborescente. A chaque nœud et en fonction du nombre de sous-caractéristiques, on peut construire un polygone (triangle, rectangle, etc.) qui constitue les diagrammes de Kiviat. Ces derniers peuvent représentés des informations quantitatives concernant les exigences ou les caractéristiques de chaque niveau.

En 2004, Marinescu et Ratiu [MAR 04] proposent dans le contexte des conceptions orientées objet un nouveau modèle nommé le modèle facteur-stratégie. Le modèle est basé sur l'utilisation du mécanisme de stratégie de détection. Celui-ci permet la construction de règles de bonne-conception et d'heuristiques de manière quantifiée, pour la détection de défauts de conception liés à un facteur de qualité donné. Le modèle utilise une approche de décomposition de la qualité en facteurs de la même façon que les approches FCM. Cependant, les facteurs de qualité sont exprimés et évalués avec des stratégies de détection. Cette approche se positionne à un haut niveau d'abstraction dans l'évaluation de la qualité, en faisant le lien avec des règles et des principes d'une bonne conception (orientée objet) quantifiés par le biais de stratégie de détection, au lieu d'interpréter la qualité avec des valeurs peu significatives.

Récemment encore, en 2006, Broy *et al.* [BRO 06] ont proposé un nouveau modèle de qualité bidimensionnel basé sur les activités. Il se focalise principalement sur l'évaluation de la maintenabilité d'un produit logiciel. Toutefois, il a été démontré et appliqué sur d'autres qualités telles que l'utilisabilité et la fiabilité. Le modèle est censé résoudre un certain nombre de problèmes dont souffrent les approches traditionnelles pour la modélisation de la qualité en général, par exemple :

– la décomposition inadéquate des attributs et des critères de qualité à un niveau qui convient pour leur évaluation pour un système ;

– l'absence des raisons (*Rationale*) de la présence de certaines propriétés du système ;

– le mélange dans ces modèles de deux dimensions, les propriétés du système et celles de l'utilisateur.

Le modèle décompose la qualité en deux concepts, les propriétés du système et leur impact sur les activités menées par l'utilisateur. Cette séparation constitue les deux dimensions du modèle qui sont respectivement : les faits et les activités. Le modèle permet une décomposition structurée sous forme d'arbre qui raffine les deux concepts à des niveaux de détail souhaités et non limités par le modèle. Ces derniers se croisent au niveau des feuilles formant ainsi une matrice qui montre la relation entre eux. Les faits servent à décrire une situation (les propriétés du système ou tout autre facteur influant sur les activités) avec les attributs qui leurs sont associés.

A l'inverse des autres modèles sauf le travail de Kitchenham *et al.* [KIT 97], le modèle proposé se base sur un métamodèle de qualité : QMM (*Quality Metamodel*) qui définit tous les éléments que nous venons de présenter et auquel toutes les instances doivent y adhérer. La définition d'un métamodèle constitue un avantage, puisque la sémantique des éléments du modèle de qualité construit est bien définie.

Pouvoir répondre aux besoins de la problématique évoquée dans l'introduction, impose le choix d'un modèle de qualité permettant la caractérisation des différents attributs qualité (concrétisés par des décisions architecturales) d'un produit logiciel. Hors, l'ensemble de modèles que l'on vient de synthétiser offre cette possibilité mais de différentes manières. En effet, chaque modèle possède sa propre vision et définit ses propres caractéristiques malgré le fait qu'ensemble ces modèles se partagent une multitude de caractéristiques de qualité. La plupart de ces modèles constitue le fruit des efforts personnels, ce qui explique cette diversité dans les points de vue et dans leurs interprétations pour les caractéristiques de qualité. En outre, ces modèles sont en majorité appliqués sur un produit logiciel en général, à l'exception du modèle de Dromey qui s'est focalisé principalement sur des produits d'implémentation (le code). De même, pour le modèle du SEI qui offre une classification des attributs qualité pour les architectures logicielles, mais qui s'éloigne de notre intérêt pour certains types d'attributs comme à titre d'exemple, les qualités d'affaires. Nous avons donc choisi d'adopter le standard ISO 9126 pour représenter et caractériser les propriétés de

qualité. Celui-ci constitue le résultat d'un consensus de la communauté internationale pour la qualité d'un produit logiciel, et s'applique sur tous les niveaux du processus de développement.

9.2.1.2. *Documentation des attributs qualité dans les architectures logicielles*

D'autres approches tentent d'anticiper l'évaluation de la qualité[1] et traitent l'aspect qualité en capturant et en documentant les exigences qualité (communément appelées besoins non fonctionnels ou NFR pour *Non-Functional Requirements*) de manière explicite dans l'architecture logicielle. Considérée comme un artéfact important dans le développement de logiciels, l'architecture logicielle apparaît comme étant un niveau approprié pour étudier la qualité d'un logiciel.

Dans la littérature, il n'y a pas de définition universellement acceptée pour le terme architecture logicielle. La définition la plus utilisée est celle du standard IEEE [IEE 00] qui précise qu'une architecture logicielle est « l'organisation fondamentale d'un système, incorporée dans ses composants, les relations entre ses composants et leur lien avec leur environnement, ainsi que les principes qui régissent sa conception et son évolution. »

Les travaux de recherche menés sur les NFR peuvent être classés en deux catégories, les approches centrées produit et les approches orientées processus. La première catégorie vise l'évaluation des NFR sur un produit logiciel, alors que la deuxième vise l'identification, la modélisation et la gestion des NFR.

L'un des travaux majeurs dans la littérature est celui de Mylopoulos *et al.* [MYL 92]. Suivant une approche orientée processus les auteurs proposent un *framework* pour la représentation et l'utilisation des besoins non fonctionnels durant le processus de développement. Le *framework* inclut cinq composantes qui, suivant une démarche guidée par les buts (*goals*), permet de justifier et d'argumenter les choix conceptuels réalisés pour satisfaire certaines exigences de qualité du logiciel. Les auteurs considèrent les besoins non fonctionnels comme des buts à atteindre en validant les bonnes décisions de conception et les raisons de ces dernières, considérées à leur tour comme des buts. Ainsi, la conception du système est guidée par les NFR, en construisant un graphe contenant les compromis possibles entre les décisions de conception qui les implémentent et leurs *Rationale* (raisons des décisions). Comparée aux approches abordées plus haut qui traitent de manière quantitative la qualité en utilisant des valeurs de métrique sur le produit final, cette approche traite de manière qualitative la satisfaction des exigences de qualité en anticipant l'évaluation durant le processus de conception.

1. Celle-ci est effectuée une fois la construction de l'architecture est finie.

Cyneirios *et al.* [CYS 04] proposent une approche qui se base sur le *framework* de Mylopoulos pour la capture et la représentation des NFR et de leurs interdépendances. Leur approche montre l'intégration de ces derniers dans les modèles FR (modèles de besoins fonctionnels). Ils se sont intéressés aux modèles conceptuels exprimés en UML en intégrant les descriptions des NFR dans les diagrammes de classes, de séquence et de collaboration.

D'autres méthodes de conception dans la littérature permettent la construction d'architectures logicielles qui répondent à des besoins non fonctionnelles. ADD (*Attribute Driven Design* [BAS 01]) est une méthode semblable dans l'esprit à celle de Mylopoulos *et al.* Elle suit une démarche de conception architecturale dirigée par les exigences de qualité. L'idée derrière est que les décisions de conception sont influencées par les exigences de qualité à satisfaire. Les auteurs ont proposé à cet effet des primitives d'attributs (des patterns architecturaux), qui sont des collections de composants et de connecteurs qui collaborent pour satisfaire quelques attributs qualité. Ces derniers sont caractérisés et documentés sous forme de scénarios généraux. Comme exemples de primitives d'attributs, nous avons : « un routeur de donnée », « un cache et les composants qui accèdent à celui-ci », ou bien « l'ordonnancement à priorité fixe ». Chacune de ces primitives d'attributs cible et réalise un attribut de qualité ; pour les exemples, nous avons la « maintenabilité » et la « performance ». En effet, la première primitive limite les connaissances que les producteurs et les consommateurs ont les uns sur les autres et affecte ainsi la maintenabilité. De même, la deuxième primitive permet de garder une copie des données à la portée des composants qui l'utilise offrant ainsi de meilleures performances. La conception de l'architecture suit un processus de décomposition et de raffinement pour utiliser la documentation des attributs qualité ainsi que les primitives d'attributs. A chaque étape de décomposition, les primitives d'attributs sont choisies pour satisfaire un ensemble de scénarios de qualité. Ensuite les fonctionnalités sont allouées pour instancier les connecteurs et les composants fournis par les primitives. Reprenons l'exemple de la primitive d'attribut « routeur de données » comme solution pour l'attribut de qualité « maintenabilité ». Cette primitive définit trois types d'éléments de conception : « producteur », « consommateur » et le « routeur de données ». Selon une certaine spécification de besoins fonctionnels, différentes fonctions sont définies. Une fonction « capteur » qui produit des valeurs de données, une fonction d'orientation et une fonction de diagnostique qui consomme ces données. Le type d'élément « producteur » est instancié pour le « capteur ». Le type d'élément « consommateur » est instancié pour les fonctions « orientation » et « diagnostique ». Tandis que le type « routeur de données » peut être instancié en « tableau noir » (*blackboard* en anglais).

Les ABAS (*Attribute-Based Architectural Style*) ont été proposés par Klein *et al.* [BAS 06] comme une amélioration des styles architecturaux en leur associant des *frameworks* de raisonnement basés sur des modèles d'attributs qualité. Les ABAS peuvent être utilisés durant l'analyse et la conception d'architecture logicielle. Ils permettent de raisonner sur les décisions architecturales garantissant certains attributs

qualité. La caractérisation de ces derniers est réalisée en se basant sur les scénarios. Les auteurs ont proposé plusieurs types d'ABAS, comme : l'ABAS client/serveur de synchronisation pour l'attribut qualité performance, l'ABAS en couches pour la maintenabilité, et l'ABAS de redondance (le simplex) pour la disponibilité.

Les tactiques architecturales, dans le même esprit que les primitives d'attributs, ont été utilisées dans d'autres travaux pour guider la conception d'architectures logicielles garantissant certaines caractéristiques qualité. Bass *et al.* [BAS 03] ont proposé un catalogue de tactiques architecturales pour satisfaire plusieurs types d'attributs qualité tels que, la maintenabilité, la performance et la sécurité.

Kim *et al.* [KIM 09] ont présenté une approche pour représenter les NFR dans l'architecture logicielle en utilisant les tactiques architecturales. Ces dernières ainsi que leurs relations sont exprimées par des *Feature Models* et leur sémantique avec le langage RBML (*Role-Based Metamodeling Language*). A partir d'une spécification des NFR, les tactiques architecturales répondant aux qualités souhaitées sont sélectionnées puis composées pour avoir une tactique qui englobe les solutions de toutes les tactiques. La tactique résultante est ensuite instanciée pour créer une architecture logicielle qui incorpore les NFR pour le système en cours de développement.

Le travail de [NIE 07] propose la méthode QRF (*Quality Requirements of a software Family*), qui se focalise sur la représentation et la transformation des exigences de qualité vers les modèles d'architecture pour les familles de produit logiciel. Elle permet aussi l'évaluation de la qualité lors des premières phases du développement. La représentation des exigences de qualité dans l'architecture se fait par le biais d'un ensemble de vues architecturales. Cette étape qui constitue la dernière étape après une suite d'étapes d'analyse (d'impact, de qualité, de variabilité et du domaine) inclut deux activités : la sélection de styles et de patterns qui supportent différentes qualités, et la description de contraintes qualitatives.

Marew *et al.* [MAR 09] ont proposé une approche inspirée principalement des travaux de Mylopoulos, Chang et Nixon [MYL 92, CHU 99]. Elle vise à intégrer les NFRs dans la phase d'analyse et de conception du cycle de vie du logiciel pour combler l'écart qui existe entre la capture des NFR et l'implémentation (prise en compte des NFR tard dans le processus de développement). Ceci étant fait tout en gérant les conflits entre ces NFR en introduisant des priorités indiquant l'importance portée sur eux. L'approche intègre aussi l'idée des « classpects » [RAJ 05] qui combine le concept de classe et celui d'aspect. Suivant l'approche orientée objet, les auteurs ont introduit d'autres phases pour la modélisation des NFR dans la phase d'analyse et de conception. Ils ont fourni une classification de tactiques pour modéliser les NFR à savoir les tactiques d'analyse et les tactiques de conception. Les « classpects », les classes et de nouveaux algorithmes sont utilisés pour modéliser les tactiques d'analyse. Les auteurs ont proposé des structures d'analyse améliorées par rapport au *framework* pour les NFR de Mylopoulos *et al.* Ils proposent Q-SIG qui est une version

quantifiée de SIG (*Softgoal Interdependency Graph*) pour incorporer l'aspect quantitatif.

Tous ces travaux traitent de différentes manières les NFR, néanmoins ils permettent tous de caractériser et de documenter la qualité. Nous avons mentionné plus haut que la qualité d'un produit logiciel est reflétée par un ensemble de décisions architecturales. La concrétisation et l'implémentation de ces décisions nécessite des moyens qui permettent de les décrire et de les documenter.

9.2.2. *Documentation des décisions architecturales*

Le concept d'architecture logicielle est reconnue comme moyen efficace pour faire face aux problèmes de conception de systèmes logiciels complexes. Une architecture logicielle constitue l'un des premiers artéfacts du processus de conception. Elle représente les toutes premières décisions de conception d'un système logiciel [BAS 03], et permet donc d'analyser et d'évaluer le système tôt dans le processus de développement. La pratique des architectures logicielles sur ces deux dernières décennies a connu d'importantes évolutions pour la représentation et la description de cette dernière. Le travail de Kruchten *et al.* [KRU 09], présente une vue historique intéressante sur la manière dont les architectures logicielles ont été abordées.

Les recherches menées dans la discipline des architectures logicielles ont montré les conséquences importantes dues au phénomène dit d'évaporation des connaissances [JAN 05] sur les décisions architecturales et sur lesquelles l'architecture logicielle est fondée. Dans ce contexte, diverses approches existent pour décrire et documenter cette connaissance. Une première catégorie de travaux se focalise sur l'utilisation de constructions langagières qui permettent d'exprimer ces décisions architecturales en respect à des concepts qui sont définis au niveau des « descriptions d'architecture ». Une autre catégorie de travaux traite la notion de décision architecturale comme une entité de première classe définie de façon explicite, indépendamment d'une description d'architecture particulière. Nous allons commencer d'abord par présenter la première catégorie de travaux représentée par les langages de description d'architecture, dits ADL.

9.2.2.1. *Documentation des décisions dans la description d'architectures*

Les langages qui permettent de spécifier des descriptions d'architectures logicielles sont les ADL (*Architecture Description Language*). Ils donnent le moyen d'organiser un système logiciel en un assemblage de composants et de connecteurs, une vue donc assez abstraite du système. Les composants sont les unités de calcul ou de stockage dans le système et les connecteurs, les unités de communication entre les composants. Outre cette capacité, certains ADL permettent la définition de contraintes architecturales qui régissent les types d'assemblages autorisés en imposant des restrictions sur la manière dont les éléments qui composent le système

sont arrangés. Parmi les ADL qui offrent cette possibilité, nous retrouvons le langage WRIGHT [ALL 97]. Celui-ci intègre les approches formelles pour la description et l'analyse des architectures logicielles et plus particulièrement la formalisation des connecteurs. Les contraintes dans WRIGHT sont définies par des prédicats et portent sur n'importe quel élément de l'architecture (*Components*, *Connectors*, *Ports*, *Roles*, etc.). La contrainte suivante stipule que la configuration d'une architecture doit avoir une topologie en étoile :

$\exists center : Components \bullet$
$\quad \forall c : Connectors \bullet \exists r : Role; p : Port \mid ((center, p), (c, r)) \in Attachments$
$\wedge$
$\quad \forall c : Components \bullet \exists cn : Connectors; r : Role; p : Port \mid ((c, p), (cn, r)) \in Attachments$

Le premier prédicat indique qu'il existe un composant (« center ») qui est attaché à tous les connecteurs de la description. Le second prédicat indique que tous les composants doivent être attachés à un connecteur. Ainsi, cette contrainte garantit que tout composant est connecté au composant représentant le centre de l'étoile.

Armani [MON 01] est un autre langage de contraintes qui étend et complète l'ADL Acme [GAR 00]. Il permet la description des architectures logicielles, d'imposer des contraintes sur l'évolution des éléments constituant ces architectures, et spécialement la capture d'expertise dans la conception des architectures logicielles. Il permet de décrire trois classes d'expertise de conception architecturale : le vocabulaire de conception, les règles de conception et les styles architecturaux. Le vocabulaire de conception spécifie les éléments de base pour la conception d'un système. Il décrit la sélection de composants, connecteurs, ports, systèmes, propriétés, rôles et de représentations qui peuvent être utilisés dans la conception du système. Les règles de conception permettent de spécifier des heuristiques, des invariants et des contraintes de composition pour assister les architectes à la conception et l'analyse des architectures logicielles. L'expression de contraintes sur l'architecture d'un système est sous la forme d'invariants en logique du premier ordre. Armani permet l'association de règles de conception avec un style complet, une collection d'éléments de conception reliés, un type d'élément de conception, ou bien une instance d'un composant ou connecteur. Le langage de prédicats d'Armani fournit diverses fonctionnalités, telles que la composition de termes, la possibilité de définir ses propres fonctions ou bien d'utiliser des fonctions prédéfinies. Parmi les fonctions prédéfinies dans ce langage, on retrouve les fonctions de vérification des types (par exemple, declaresType (e :Element,t :Type) :boolean), les fonctions de graphes (par exemple, connected(c1, c2 : Component) : boolean), les fonctions de relation père-fils (par exemple, parent(c :Connector) :System), les fonctions sur les ensembles (par exemple, sum(s : setnumber) : number). Le langage de prédicat d'Armani inclut aussi des opérateurs logiques, arithmétiques et de comparaison. Il distingue deux types de

contraintes : les « invariants » et les « heuristiques » (qui ne sont pas utilisées dans les prédicats de types). L'exemple suivant illustre leur utilisation :

```
1 Invariant Forall c1,c2 : component in sys.Components |
2               Exists conn : connector in sys.Connectors |
3                 Attached(c1,conn) and Attached(c2,conn);
4 Heuristic Size(Ports) <= 5;
```

La contrainte exprimée par l'invariant impose que les composants doivent être connectés deux à deux pour former un graphe complet. L'heuristique précise que le nombre de tous les ports doit être inférieur ou égal à cinq.

La dernière classe d'expertise concerne les styles architecturaux. Des exemples sur la spécification des styles architecturaux utilisant Armani peuvent être trouvés dans [MON 01].

Sans doute, l'une des solutions les plus pertinentes pour favoriser la réutilisation dans les architectures logicielles est l'utilisation des styles architecturaux. Un style architectural définit une famille de systèmes en fournissant un vocabulaire de conception architecturale spécifique à un domaine avec des contraintes sur la façon dont les parties peuvent se regrouper ensemble [KIM 10]. Une spécification d'un style en Armani consiste en une déclaration d'un vocabulaire de conception qui peut être utilisé pour la conception de styles, et un ensemble de règles de conception qui guident la composition et l'instanciation du vocabulaire de conception. Dans la catégorie des ADL, WRIGHT offre la possibilité de définir les styles architecturaux.

Dans [KIM 10], Kim et Garlan proposent une approche pour la transformation de styles architecturaux (une forme de décision architecturale) exprimés formellement par l'ADL ACME, vers des modèles relationnels exprimés avec le langage Alloy [JAC 02]. Alloy est langage de modélisation basé sur la logique du premier ordre (*first order relational logic*). La méthode permet donc de décrire des spécifications de styles architecturaux dans les constructions langagières du langage Alloy, par le biais d'un schéma de translation de style. L'objectif est de pouvoir vérifier des propriétés sur les styles, à savoir, si un style satisfait quelques prédicats posés sur sa structure architecturale, la cohérence d'un style (l'existence d'au moins une configuration conforme aux contraintes architecturales du style), ou encore si deux styles sont compatibles pour une composition.

9.2.2.2. *Documentation des décisions annexes aux descriptions d'architectures*

L'autre catégorie de travaux sur la documentation des décisions architecturales traite les décisions architecturales comme entités de première classe et vise à les représenter explicitement, ainsi que les raisons de ces décisions (*Design Rationale*) dans la documentation de l'architecture logicielle. Celle-ci est annexe à la description d'architecture, qui est spécifiée en utilisant un ADL. Le but est de capturer la connaissance

en relation avec les décisions architecturales prises durant le développement logiciel pour réduire les effets du phénomène d'évaporation de connaissances. Cette idée vient du fait que la plupart des décisions réalisées durant la construction de l'architecture logicielle reste implicite ou comme des intentions non persistantes. Par conséquent, considérer les décisions architecturales comme des éléments de première classe et les représenter de façon explicite dans la documentation architecturale est l'un des moyens les plus intéressants pour améliorer la qualité des architectures logicielles comme nous allons le démontrer plus loin.

Le travail de Kruchten *et al.* [KRU 09] présente une vue historique intéressante sur la manière dont les architectures logicielles ont été abordées. Le travail de Perry et Wolf [PER 92] fut l'une des premières contributions majeures pour la description d'une architecture logicielle. Ces derniers ont introduit la définition suivante pour une architecture logicielle :

Une architecture logicielle = {Eléments, Forme, *Rationale*}.

Les « éléments » constituent différents types d'unités, de traitements, de données et de connexions. Les éléments de traitements sont des composants qui réalisent des traitements sur les éléments de données. Les éléments de connexion font la liaison entre les deux premiers éléments. La « forme » est définie par un ensemble de propriétés et de relations, qui définissent des contraintes sur les éléments de l'architecture et leur mode d'interaction. Une architecture peut avoir différentes formes tout en représentant les mêmes fonctionnalités. Le « rationale » explique les motivations de l'architecte pour certains choix architecturaux. Perry et Wolf ont mentionné l'utilisation de « vues » pour la construction des architectures logicielles tout en respectant les concepts proposés dans leur modèle. Ces derniers représentent différents aspects de l'architecture logicielle, qui reflètent les différentes préoccupations des utilisateurs de celle-ci.

Le modèle 4+1 de Kruchten [KRU 95] a été proposé dans le même contexte. Il a marqué une nouvelle aire pour la description et la documentation des architectures logicielles. Il organise cette dernière selon quatre vues différentes à savoir : la vue logique, la vue de processus, la vue de développement, la vue physique et la vue de cas d'utilisation (plus un). Toutes ces vues représentent les préoccupations de différents participants dans le développement d'un système logiciel. UML (*Unified Modeling Language*) est le langage utilisé pour la représentation de ces vues. Les décisions architecturales qui se manifestent dans l'architecture logicielle peuvent être élaborées sur l'une des vues ou capturées en combinant les différentes vues du modèle 4+1. L'approche 4+1 a été adoptée comme une pièce fondatrice de l'approche RUP (*Rational Unified Process*) [CLE 03].

La notion de vue a été reprise quelques années plus tard avec l'émergence du standard IEEE 1471-2000 [IEE 00], mais avec plus de raffinement concernant la manière dont une vue doit exprimer certains aspects de l'architecture logicielle. Le standard définit la vue pour exprimer l'architecture d'un système conformément à un point de

vue particulier. Ce dernier détermine les langages à utiliser pour décrire une vue, les méthodes de modélisation ou les techniques d'analyse associées qui sont appliquées aux représentations de vues. Les points de vues répondent à certaines préoccupations (telles que les NFR) des participants dans le développement du système logiciel. Le modèle conceptuel du standard a été amélioré comparé à la version initiale, pour incorporer comme élément de première classe, les raisons des décisions architecturales (*Rationale*). Il inclut en outre, d'autres éléments, à savoir la mission que doit remplir le système, l'environnement du système et une bibliothèque de points de vue.

Clements *et al.* [CLE 03] ont proposé dans leur approche V&B (*Views and Beyond*) trois différents types de vue pour documenter une architecture logicielle :

– le type de vue module, qui répond à la manière dont l'architecture logicielle doit être structurée sous forme d'un ensemble d'unités d'implémentation ;

– le type de vue composant et connecteur (C&C pour *Component-and-Connector* en anglais), permet de structurer l'architecture en un ensemble d'éléments qui ont un comportement à l'exécution et des interactions ;

– le type de vue allocation, répond à la façon dont l'architecture est liée aux structures non logicielles de son environnement.

Les approches de documentation que l'on vient d'illustrer [CLE 03, IEE 00, KRU 95], ont pour objectif la définition d'un ensemble de vues sur les éléments d'un système et leurs relations pour décrire une architecture logicielle. Nous allons maintenant présenter quelques travaux qui ont été proposés dans le but de réduire l'évaporation de connaissances, en explicitant la représentation des décisions architecturales.

En 2005, Jansen et Bosch [JAN 05] ont proposé une nouvelle manière de percevoir une architecture logicielle. Ils la présentent comme un ensemble de décisions de conception architecturale. Selon eux, une décision de conception architecturale est définie comme la description de l'ensemble des ajouts, des retraits et des modifications des éléments d'une architecture logicielle, les raisons de ces décisions (*Rationale*), et les règles de conception, les contraintes de conception et les exigences supplémentaires. Cette définition met en avant l'importance de cette notion comme étant l'une des formes de connaissances architecturales (AK pour *Architectural Knowledge* en anglais) les plus importantes dans le processus de développement logiciel. La notion de « AK » est une nouvelle avancée de recherche dans la discipline des architectures logicielles, qui englobe toute connaissance acquise ou formulée dans cette dernière [JAN 08]. Jansen *et al.* ont mentionné que cette notion de AK est vitale pour le processus de construction d'architecture car elle améliore la qualité de ce processus et de l'architecture elle-même.

Dans ce travail, les auteurs ont présenté une nouvelle approche pour le développement d'architectures logicielles nommée « Archium », qui considère, comme cela est

précisé ci-dessus, qu'une architecture logicielle est un ensemble de décisions architecturales. L'approche est basée sur un modèle conceptuel de décisions architecturales qui décrit les éléments de décisions architecturales (problème, motivation, cause, solution, décision, modification architecturale et contexte) et leurs relations. La description d'une architecture logicielle se fait à travers le modèle conceptuel composé des notions de :

– *Deltas* (élément du modèle architectural) qui expriment un changement sur le comportement d'un composant et représentent ses fonctionnalités ;

– de fragments de conception ;

– de décisions architecturales.

L'ajout d'un incrément sur l'ancienne architecture logicielle du système est obtenu par l'utilisation d'un modèle de composition. Celui-ci fournit les éléments nécessaires pour lier les éléments du modèle architectural avec ceux du modèle de décisions de conception. Il relie les changements apportés par le modèle de décision de conception avec les éléments du modèle architectural. L'architecture finale n'est, ainsi, qu'un ensemble de décisions architecturales.

Kruchten *et al.* [KRU 06] définissent la connaissance architecturale comme suit :
AK = *Design Decision* (Décision de Conception) + *Design* (Conception)
Cette formule confirme bien la vision introduite par Jansen et Bosch. Elle considère les décisions architecturales comme partie essentielle de la connaissance qui contribue dans la construction des architectures logicielles. Les auteurs ont parlé de trois niveaux de connaissances qui peuvent être appliqués pour catégoriser la connaissance architecturale :

– tacite : comme des intentions ;

– documentée : il y a des traces de cette connaissance ;

– formalisée : non seulement documentée, mais organisée de manière systématique.

La meilleure manière qui nous paraît évidente pour conserver cette connaissance, est d'adopter le niveau de formalisation. Ce choix est motivé par la possibilité de réaliser une gestion automatique des décisions.

Le travail de Kruchten [KRU 04] propose une taxonomie sur les décisions architecturales. Il a défini un modèle pour la caractérisation des décisions architecturales par un ensemble d'attributs ainsi que leurs relations pour les systèmes logiciels complexes. Parmi ces attributs, on trouve le *Rationale* indiquant les raisons de la décision prise, l'état de la décision, l'auteur, la catégorie, le coût, etc. L'auteur a défini différents types de relations permettant de documenter les dépendances qui peuvent exister entre les décisions architecturales et permettent de raisonner sur eux.

Dans [TYR 05] Tyree et Akerman ont présenté l'importance des décisions architecturales dans le processus de développement des architectures logicielles en les documentant proprement et en les rendant explicite. Ils précisent qu'un simple document décrivant les décisions architecturales clés peut aider considérablement à éclaircir les architectures système. A cet effet, ils ont proposé un *Template* de description des décisions architecturales, dérivé de deux modèles le REMAP [BAL 91] et le DRL [JIN 89]. Le *Template* liste et décrit un certain nombre d'informations autour desquelles s'articule la notion de décision architecturale et sur lesquelles une description et une documentation d'une décision est élaborée (*Issue*, *Décision*, *Statut*, *Groupe*, *Suppositions*, *Contraintes*, *Argument*, *Implications*, etc.). Le modèle est doté d'un vocabulaire riche pour la description des décisions architecturales, et peut fournir un éventuel support pour l'analyse d'impact d'une évolution architecturale sur les fonctionnalités du logiciel et cela en se basant sur les informations documentées dans le champ « artéfacts relatifs ». Celui-ci précise les éléments sur lesquels la décision aura un impact (on pourra isoler les changements fonctionnels).

Lago et Van Vliet [LAG 05] parlent de la documentation explicite des raisons des décisions architecturales qu'ils ont nommés « suppositions » (*assumptions*, en anglais). Ils ont proposé une approche pour rendre explicite ces suppositions considérées comme des invariabilités dans le système, et les lier aux documentations des architectures logicielles. Ceci devrait enrichir la documentation et fournir un support meilleur pour l'évolution et la maintenance.

Bass *et al.* [BAS 06] considèrent les décisions de conception architecturale et le *Rationale* comme étant la forme de connaissance la plus importante à capturer. Ils ont défini la décision architecturale par la notion de transformation de l'architecture d'un état avant l'application de la décision vers l'état après son application. Sur la base de cette notion ils ont proposé de documenter les décisions architecturales par le biais de deux graphes. Le premier nommé graphe de causalité, c'est un graphe orienté acyclique qui organise les décisions (représentées par les nœuds) selon un ordre temporel. Ce dernier considère la conception comme une séquence de décisions (transformations) avec laquelle on peut tracer la généalogie de toutes les décisions prises. Le deuxième graphe représente un autre aspect des transformations qui est le choix de patrons ou de tactiques architecturales permettant d'implémenter les décisions architecturales. Cela donne l'ensemble d'éléments architecturaux tels qu'ils existent dans l'architecture logicielle à un moment donné durant le développement. La combinaison des deux graphes fournit des réponses sur la manière selon laquelle l'architecture logicielle prend une certaine forme, c'est-à-dire l'ensemble de décisions architecturales sous un ordre chronologique d'apparition sur l'architecture par le biais de graphe de causalité, ainsi que leurs effets sur l'architecture représentée par un graphe structurel pour un niveau donné du graphe de causalité. Les deux graphes peuvent fournir de précieuses informations sur l'architecture logicielle et améliorent ainsi la qualité du processus de conception d'architecture. Ils fournissent les raisons de toutes les décisions prises et les chemins inutiles déjà empruntés.

Dans [CAP 07], les auteurs ont proposé une manière pour caractériser la connaissance architecturale (*Architectural Knowledge*) et plus particulièrement les décisions architecturales afin de définir un processus de gestion de celles-ci dans un contexte d'évolution. La plus grande partie de cette connaissance reste implicite dans les intentions de l'architecte et tend à disparaitre avec le temps emportant avec elle tout le raisonnement (solutions alternatives et chemins empruntés) qui est lié à l'architecture courante du logiciel. Ce travail vient renforcer encore l'idée de documenter les décisions architecturales explicitement durant le processus de développement des architectures logicielles comme étant des entités de première classe. Pour ce faire, les auteurs ont défini des attributs pour décrire les décisions architecturales en séparant, selon leur degré d'importance, les attributs obligatoires et les attributs optionnels. La première classe d'attributs constitue les informations associées aux décisions architecturales qui doivent être définies tout au long de la période de vie du système à savoir : le nom de la décision, sa description, les contraintes, les dépendances (entre décisions), le statut, le *Rationale*, les patrons de conception, les solution architecturales, et les besoins menant à la décision. La deuxième classe fournit des informations complémentaires qui peuvent être définies au choix selon les préférences de l'utilisateur dans le projet. Outre ces attributs, ils ont défini des attributs pour supporter l'évolution des décisions architecturales à savoir : date/version, décision obsolète, validité, nombre de réutilisations et liens de traçabilité. Les décisions architecturales sont décrites par le modèle de décision faisant partie du métamodèle proposé par les auteurs pour la construction et l'évolution de la connaissance architecturale. Les deux autres parties du métamodèle sont : le « modèle du projet » contenant des informations pour la construction de l'architecture logicielle, et l'« architecture » représentant l'architecture logicielle décrite par une ou plusieurs vues architecturales. Le métamodèle intègre deux vues différentes sur la connaissance architecturale : la vue « produit » qui décrit les décisions architecturales à travers les attributs illustrés plus haut, et la vue « processus » qui exprime les activités de prise de décision menées par les architectes pour stocker, gérer, évaluer, documenter, communiquer, découvrir et réutiliser les décisions architecturales.

Kruchten *et al.* [KRU 09] confirment la nécessité d'intégrer les décisions architecturales et les raisons de celles-ci comme des éléments de première classe dans la documentation des architectures logicielles. Les auteurs ont présenté un historique sur l'évolution de la représentation des architectures logicielles, qui s'étend sur trois périodes. La première met l'accent sur l'utilisation des vues architecturales, notamment avec l'émergence du célèbre modèle de vues 4+1. La deuxième se caractérise par l'apparition de nouvelles méthodes venues compléter la description de vues, tel que la méthode RUP *(Rational Unified Process)* d'IBM, ou les méthodes du SEI *(Software Engineering Institute)* telles que ATAM *(Architecture Trade-off Analysis Method)*, ADD *(Attribute Driven Design)* et SAAM *(Software Architecture Analysis Method)*. Toutes ces méthodes sont utilisées, pour l'analyse et l'évaluation des architectures logicielles. Les auteurs ont mentionné que le point commun à toutes ces méthodes est l'utilisation de décisions architecturales. Ce qui nous mène à la dernière

période qui met en valeur cette notion. Elle s'intéresse à l'investigation dans la représentation, la capture, la gestion et la documentation des décisions de conception réalisées durant la construction d'architectures. Kruchten a renforcé cette idée et a inclus, dans le modèle 4+1, la « vue décision » qui incorpore les décisions de conception. Les auteurs ont fourni des directives pour permettre la capture et la représentation des décisions architecturales, et aider les architectes à les documenter.

9.2.3. *Assistance à l'évolution du logiciel*

Malgré le sens positif que le terme évolution porte, il peut s'avérer néfaste de conséquences en l'associant aux logiciels. En effet, l'évolution d'un logiciel ne manque pas de conséquences indésirables si elle ne se fait pas de manière contrôlée. Ainsi, en voulant répondre aux nouveaux besoins en les introduisant dans l'architecture d'un système logiciel, ou en voulant effectuer des opérations de maintenance en apportant des modifications sur cette dernière, on pourra facilement altérer d'autres besoins (fonctionnels ou non fonctionnels) et dévier de la spécification des besoins initialement planifiés. Ce processus d'évolution a pour effet de faire apparaître des phénomènes qui ont eu des appellations différentes par les chercheurs qui les ont identifiés dans la littérature, comme par exemple : « vieillissement du logiciel » (*Software Aging* [PAR 94]), « érosion de l'architecture » (*Architectural Erosion* [MER 10]), « érosion de la conception » (*Design Erosion*), « décadence du code » (*Code Decay* [EIC 01]) ou « dégénérescence de l'architecture » (*Architectural Degeneration* [LIN 02]).

Malgré la diversité de ces termes, tous ces phénomènes s'entendent sur le fait que le logiciel devient, après une succession de changements, difficile à maintenir et à faire évoluer. Ceci rend par conséquent le logiciel inexploitable et dans certains cas bon à « jeter » (arrêter sa mise en production). Cette situation montre la nécessité de faire recours à des méthodes et des techniques pour pouvoir contrôler et maîtriser les changements du logiciel.

Hochstein et Lindvall [HOC 05] ont présenté dans leur travail les différents phénomènes évoqués ci-dessus. Ils les ont représenté par le terme « dégénérescence » qui les englobe, malgré le fait que les chercheurs ayant proposé ces appellations faisaient référence à différents niveaux d'abstraction à savoir le niveau conception (érosion de l'architecture et érosion de la conception) et au niveau de l'implémentation (vieillissement du logiciel et décadence du code). Les auteurs ont discuté différentes manières pour traiter le contrôle de l'évolution du logiciel. Ils ont introduit diverses approches en partant du diagnostique, le traitement, la recherche, jusqu'à la prévention suivant un modèle médical. Le diagnostique de la dégénérescence consiste à identifier quand le logiciel dégénère. Dans cette phase, on retrouve, dans la littérature, des solutions de récupération d'architecture réelle qui consistent à extraire à partir du code source et d'autres artéfacts logiciels l'architecture concrète du logiciel. Ces méthodes se placent

dans le domaine de la rétroingénierie (*Reverse Engineering*). Le principe est de comparer l'architecture actuelle avec celle planifiée initialement, et voir si des violations au niveau de l'architecture ont été détectées. La plupart des méthodes citées se focalisent sur l'identification des styles architecturaux et des patrons de conception. Le diagnostique peut servir comme phase de prétraitement pour corriger les problèmes détectés. Les techniques présentées pour traiter la dégénérescence sont celles de refactorisation (*refactoring*) qui consistent à restructurer le code sans changer le comportement du système. La recherche de la dégénérescence vise la compréhension de l'évolution pour comprendre comment les systèmes dégénèrent. Parmi les techniques citées, on retrouve celles de la visualisation des changements architecturaux à travers les versions. Ces approches s'appliquent après évolution réelle du logiciel, c'est-à-dire après que l'application des changements souhaités ait pris place sur le logiciel et que l'architecture de celui-ci ait « dégénéré ». Même si ces méthodes garantissent que les changements se répercutent de manière correcte sur la/les parties ciblés, elles impliquent des coûts supplémentaires dus aux opérations de correction qu'elles apportent. Mieux vaut donc, prévoir les changements pour pouvoir les éviter (« mieux vaut prévenir que guérir »). Il s'agit de prendre ces précautions et de préparer le logiciel aux éventuels changements pouvant survenir dans le futur.

D'autres auteurs [BUR 06] ont mis l'accent sur l'importance des raisons des décisions prises durant le processus de développement (*Decision Rationale* : DR) et ont montré l'utilité de ces dernières durant l'évolution du système logiciel. Les auteurs de ce travail ont proposé le système SEURAT (*Software Engineering Using RATionale*) comme support pour la maintenance qui permet d'exploiter le DR en donnant la possibilité de représenter, de capturer et d'inférer sur ce dernier pour pouvoir éventuellement détecter les inconsistances et indiquer les problèmes de conception. La représentation du DR a été réalisée par RATSpeak, qui est basé sur le langage DRL (*Decision Representation Language*). RATSpeak est une structuration qui regroupe un ensemble d'éléments pour exprimer le DR. Les éléments définissant le DR proposés par les auteurs sont : les besoins, les problèmes de décision, les questions, les alternatives, les arguments, les prétentions, les suppositions, les ontologie d'arguments et les connaissances de base. La capture de ces éléments se fait par le système SEURAT qui offre un outil étroitement intégré dans l'IDE Eclipse, ce qui rend l'opération de documentation du *Rationale* pour les développeurs comme une partie intégrante et non pas séparée du processus de développement. Cela augmente par conséquent les chances pour que cette forme de connaissance très importante soit enregistrée aisément. Les développeurs peuvent associer entre autres des DR au code et préciser les parties du code qui présentent l'implémentation de ces derniers *via* les fonctionnalités de SEURAT. Le système permet aussi d'inférer la connaissance intégrée dans le code et assister le développeur dans le processus de développement durant certaines opérations de maintenance. Il offre un certain nombre de types d'inférence permettant de contrôler la structure (manque d'information) de cette connaissance et d'évaluer la consistance

de la conception résultante d'une séquence de prises de décisions, à partir de cette dernière. L'utilisation de métriques sur les arguments des alternatives permet de calculer si une décision prise précédemment est moins supportée que d'autres décisions alternatives candidates. SEURAT permet d'assister l'utilisateur du système (développeur ou mainteneur) au moment de faire des opérations de maintenance, en l'informant de l'impact d'une modification sur les choix effectués en relation avec celui en cours de modification. Il permet éventuellement d'assister le mainteneur durant les opérations de perfectionnement du système, ce qui consiste à apporter de nouvelles fonctionnalités au système à travers de nouvelles décisions, en vérifiant que ces dernières sont cohérentes avec les anciennes décisions mises en place dans le système d'origine. Cette opération de vérification est réalisée essentiellement à l'aide de l'élément du *Rationale* « connaissance de base » contenant les compromis (*Tradeoffs*) entre les décisions introduites par le développeur et qui sont à ne pas violer. Les éléments des compromis sont des items de l'ontologie d'arguments qui représente une hiérarchie extensible de raisons qui aide le développeur à faire un choix de conception parmi d'autres. La violation d'un compromis est détectée par SEURAT et l'utilisateur est notifié.

Dans le cadre de la gestion de l'évolution des systèmes logiciels orientés objet, Steyaert *et al.* [STE 96] ont proposé le concept de contrat de réutilisation (*Reuse Contracts*) dans la perspective d'une bonne réutilisation des artéfacts logiciels. Il s'agit de contrôler et de gérer la propagation des changements entrepris sur les modèles objet en modifiant les classes (artéfacts réutilisables) qui les composent, par l'utilisation de contrats de réutilisation. Ce dernier documente les intentions de deux parties, le développeur de l'artéfact et celui qui va réutiliser l'artéfact. Un contrat de réutilisation est une interface qui contient la spécification d'un ensemble de méthodes. Chaque méthode est identifiée par un nom avec une clause de spécialisation optionnelle listant uniquement les méthodes nécessaires à la conception de la méthode, et peut être de type *abstract* ou *concrete*. L'utilité des contrats de réutilisation a été montrée sur les classes abstraites comme artéfact réutilisable, en utilisant l'héritage comme mécanisme de réutilisation. L'approche a été exploitée sur les problèmes de conflits émanant de l'évolution des superclasses d'une hiérarchie de classe d'un modèle objet, et plus particulièrement sur l'échange des classes mère par d'autres classes. Les contrats de réutilisation sont en général implémentés par des classes abstraites et encapsulent certaines informations de conception sur les dépendances entres les méthodes qui les composent. Elles constituent donc une source d'information permettant de détecter les incohérences dans la hiérarchie de classes d'un modèle objet. Les auteurs ont proposé de catégoriser les différentes actions menées par le concepteur des classes abstraites et de leurs classes filles ainsi qu'aux changements que ces actions peuvent causer sur la structure des classes abstraites. Ils ont distingué trois opérateurs de base qui sont appliqués sur les contrats de réutilisation : la concrétisation pour implémenter les méthodes abstraites, le raffinement permettant la surcharge de méthodes et l'extension qui permet l'ajout de nouvelles méthodes. Trois opérateurs supplémentaires

s'appliquent sur les contrats de réutilisation : l'abstraction, le grossissement et l'annulation. Ils permettent d'appliquer les opérateurs inverses des premiers et de dériver à partir des contrats de base associés aux classes mères, des contrats associés aux classes filles. Ces opérateurs viennent consolider le contrat de réutilisation avec des informations supplémentaires sur la manière les classes vont être réutilisées.

Tom Mens et Theo D'hondt [MEN 00] ont proposé plus tard une généralisation du formalisme de contrat de réutilisation en l'intégrant dans le métamodèle UML. Ils introduisent le concept de contrat d'évolution pour la détection des conflits de conception dans un modèle objet. Celui-ci précise les clauses du fournisseur et du modificateur de l'artéfact logiciel. La première des clauses décrit les propriétés de l'artéfact, et la deuxième précise comment l'artéfact doit être modifié et, par conséquent, évoluer. Globalement, l'idée consiste à ajouter des contrats d'évolution entre les éléments d'un modèle objet représentant une phase donnée. Le contrat doit préciser les actions évolutives que le développeur doit suivre durant les modifications. Celles-ci constituent la base du processus de détection de conflits après que le modèle aura évolué. Les auteurs ont utilisé la notion de types de contrats pour restreindre les opérations régissant le travail du modificateur. Quatre types ont été proposés : addition, suppression, connexion et déconnexion. Un contrat d'évolution est défini comme une extension au métamodèle UML. Il peut être défini par la spécialisation de la métaclasse *Dependency*, et doit être stéréotypé pour décrire le type de modification à opérer (le type de contrats). Le rôle de la modification d'un contrat d'évolution est une séquence non vide d'éléments de la métaclasse *ModelElement* qui vont être modifiés, ajoutés ou supprimés par le contrat d'évolution. La sémantique de la modification (type de contrats) est spécifiée par des règles OCL. Un contrat d'évolution peut être défini seulement entre les éléments de la métaclasse *NameSpace*. Deux types de contrats d'évolution se distinguent selon le stéréotype (type de contrat) utilisés, les contrats d'évolution primitifs et les contrats d'évolution composites. La première sorte de contrat peut être spécialisée vers les stéréotypes primitifs évoqués plus haut et pouvant être appliqués sur les éléments de *NameSpace*. La deuxième permet de composer un contrat à partir d'autres contrats. Cette dernière comporte deux spécialisations : promotion et séquentialisation. La première permet de construire des contrats de haut niveau par un ensemble de contrats de bas niveau. La deuxième définit un contrat comme une séquence de contrats plus petits. L'introduction des contrats d'évolution dans un niveau de métamodélisation UML a permis d'attaquer les problèmes d'évolution sur différents niveaux d'abstraction, partant de la spécification des besoins jusqu'à l'implémentation.

D'autres travaux ont porté sur les systèmes logiciels à base de composants. Basé aussi sur la notion de contrat, Andreas Rausch [RAU 00] a proposé le concept de contrat de besoins/assurances (*Requirements/Assurances Contracts*) comme support pour gérer l'évolution d'un système logiciel à base de composants permettant ainsi de détecter et d'éviter les problèmes qui découlent de cette dernière. Ces contrats ont été utilisés dans le cadre d'une méthodologie de développement qui prend en considération l'évolution. Celle-ci est fondée sur un modèle formel qui capture l'aspect

structurel et comportemental des systèmes à base de composants et orientés objet. L'établissement du contrat de besoins/assurances se fait à l'aide de fonctions qui déterminent les obligations de chaque participant (composant). Elles permettent d'un côté (la fonction *REQUIRES*) de calculer à partir d'un ensemble de documents de développement, l'ensemble des propriétés (définies par des prédicats) dont le composant a besoin (requiert de son environnement). D'un autre côté (la fonction *ASSURES*) calcule l'ensemble des propriétés fournies par le composant à son environnement. Une fois toutes ces propriétés spécifiées pour chaque composant, le concepteur définit explicitement les dépendances comportementales entre composants en précisant pour chaque composant les assurances garantissant les besoins d'autres composants. La formulation de ces dépendances constitue un contrat besoins/assurances. L'étape suivante consiste à vérifier à chaque pas d'évolution (défini comme un changement dans les documents de développement dans une période de temps) si les besoins d'un composant donné restent satisfaits par les assurances d'un autre composant qui a subi une évolution. Les constructions formelles proposées dans ce travail permettent de décrire un système à base de composants ou à base d'objets. L'évolution de ces descriptions implique des changements qui peuvent s'avérer indésirables dans certains cas. Ces derniers sont détectés à travers les contrats de besoins/assurances, qui capturent les dépendances entres les composants du système et assistent le développeur durant les modifications apportées. Cependant, l'assistance fournie par ces contrats traite l'aspect fonctionnel (métier) et ne se préoccupe pas de l'impact des changements effectués sur les qualités du système.

Madhavji et Tassé [MAD 03] ont proposé une approche guidée par des politiques d'évolution pour préserver les qualités et les exigences imposées sur le logiciel lors de son évolution. L'approche introduit deux concepts. Le premier est un mécanisme qui permet la vérification de la violation de certaines politiques d'évolution. Le deuxième est un *framework* contextuel qui constitue un support pour les activités qui permettent l'évolution du système logiciel. La politique d'évolution est formulée comme une contrainte dans la logique du premier ordre, comme par exemple exprimer le besoin que la somme estimée du nombre de lignes de code ajoutés pour tous les composants du système ne doit pas dépasser la moyenne de croissance plus un pourcentage d'erreur. Le mécanisme de vérification, qui doit recueillir les informations à partir d'un modèle du produit ou du processus à améliorer, assiste le développeur du système en s'assurant de la validité de la contrainte et en le notifiant du résultat. Le résultat fourni est analysé et des informations de retour (*feedback*) sont présentées au développeur. Celui-ci se base sur ces informations pour décider des actions à prendre et apporter des améliorations au modèle. Le *framework* contextuel offre un environnement de raisonnement qui met en relation les groupes de développement et d'évolution exerçant chacun des activités en interaction. Celles-ci assurent le suivi d'évolution d'un système logiciel à travers un échange continu d'informations. Ce dernier permet de générer de nouvelles politiques traitant les défauts détectés, et qui seront vérifiés de

nouveau par le mécanisme de vérification de politiques incorporé dans les composants du *framework*.

Dans le présent travail nous allons présenter une approche pour contrôler l'évolution d'un logiciel à la volée, c'est-à-dire au moment de l'application des changements. Elle permet de vérifier si les changements nouvellement introduits sont homogènes avec l'ensemble des décisions architecturales déjà implémentées, et donc déduire les conséquences sur la qualité. Elle assiste donc le mainteneur du logiciel en fournissant un outil permettant de le notifier au moment même de l'application du changement, des conséquences de son acte en le laissant maître de décider de préserver ou d'ignorer le changement.

Ce travail enrichit un travail développé précédemment dans la thèse de Chouki Tibermacine [TIB 06a]. Ce travail s'appuie sur le concept de contrat d'évolution, introduit initialement dans [TIB 06b], qui diffère complètement des contrats d'évolution de Mens et D'Hondt [MEN 00]. Des précisions seront données dans la section suivante.

9.3. Assistance à l'évolution dirigée par la qualité

L'une des solutions de conception les plus utilisés ces dernières années dans l'ingénierie logicielle pour faire face aux problèmes de complexité des systèmes logiciels est l'utilisation des architectures logicielles. Durant le processus de construction de celles-ci, les développeurs manipulent divers types de connaissances (dont la plupart reste tacite et implicite) afin d'aboutir à une décomposition logique unissant les différentes parties du système logiciel. Ces parties, communément connues sous le nom de modules ou composants sont représentées dans un schéma de connexion faisant la jonction entre ces derniers en respectant un certain nombre de contraintes spécifiées par le développeur.

Il a été reconnu dans la littérature que c'est les besoins nonfonctionnels qui guident la construction de l'architecture logicielle et lui donnent son aspect final. Celui-ci est obtenu par l'application d'un ensemble de décisions architecturales, qui répondent aux besoins nonfonctionnels. En partant de cette vision, nous allons présenter dans cette section une approche basée sur les contrats [TIB 06b] pour la documentation des architectures logicielles des systèmes à base de composants. Cette approche vise à expliciter les liens reliant les décisions architecturales et les attributs qualité. L'objectif est de conserver ces liens initialement établis par les architectes du logiciel tout au long de l'évolution du système en voulant faire divers types d'opérations de maintenance. Ces derniers peuvent être l'œuvre d'un autre groupe de développeurs ou de mainteneurs, qui n'ont pas nécessairement une idée précise (ou qui ne connaissent

pas du tout), les décisions architecturales prises précédemment. Pour cette raison, le contrat doit contenir une sorte d'accord indiquant les autorisations et les interdictions de chaque personne participant dans le processus de maintenance. C'est ainsi que deux rôles s'identifient : le développeur de la première version du logiciel ayant la responsabilité de garantir les attributs qualité comme précisé dans les documents de spécification et le développeur de la nouvelle version (mainteneur) qui doit respecter les contraintes imposées auparavant.

9.3.1. *Documentation des architectures logicielles basée sur les contrats*

Comme nous l'avons présenté ci-dessus, il y a plusieurs travaux [MEN 00, RAU 00, STE 96] qui utilisent la notion de contrat dans le but d'assister les développeurs face aux situations problématiques rencontrées durant l'évolution d'un système logiciel.

Le modèle de contrat introduit dans [TIB 06b] appelé « contrat d'évolution architecturale » documente les décisions de conception architecturales. Il définit de manière formelle les liens unissant les décisions architecturales et les attributs qualité implémentés par ces dernières. Ce modèle a été enrichi avec différentes sortes de liens pour mieux exprimer les relations qui existent entre les décisions architecturales et les attributs qualité, ou entre les attributs qualité. Ces liens embarqués dans le contrat apportent des informations supplémentaires aux développeurs pour mieux gérer et contrôler les changements architecturaux qu'ils introduisent.

9.3.1.1. *Contrat d'évolution d'architecture*

La figure 9.1 illustre la structure d'un contrat d'évolution.

Ce nouveau modèle de contrat raffine la relation entre les décisions architecturales et l'attribut de qualité qu'elles implémentent, en associant un lien qui représente un degré de satisfaction. En effet, un attribut de qualité peut être concrétisé par plusieurs décisions architecturales qui collaborent ensemble, chacune participant avec un certain pourcentage. Dans la situation idéale, la somme de tous les degrés associée à un certain attribut de qualité dans un certain élément architectural doit être égale à 100. Les valeurs affectées dépendent totalement des décisions que prend le développeur. Par exemple, le choix d'un ensemble de décisions sélectionnées à partir d'un catalogue pour implémenter l'attribut de qualité disponibilité (*Availabilty*) peut être : la combinaison de la tactique « Ping/Echo » et la tactique « Active Redundancy » (*Redondance Active*), ou la combinaison de trois tactiques, « Ping/Echo », « Heartbeat » et la tactique « Active Redundancy » [KIM 09]. Le développeur peut associer des valeurs en pourcentage pour chaque décision, selon leurs participations dans la concrétisation de l'attribut de qualité. A titre d'exemple, pour le deuxième cas, il associe 35 % pour les deux premières tactiques et 25 % pour la dernière. On peut dire que le développeur a jugé que les tactiques concernées par la détection de défauts (*Fault detection*)

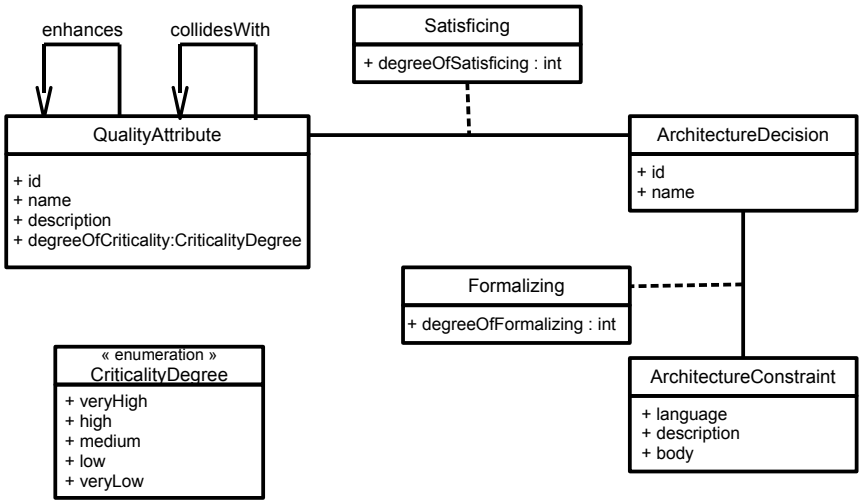

Figure 9.1. *Liens entre décisions architecturales et attributs qualité*

sont plus critiques que la troisième concernée par la récupération et la réparation du système après échec. Par analogie aux techniques utilisées en ingénierie des besoins, où l'analyste affecte des valeurs comme high, medium, ou low selon la difficulté de réalisation de chaque besoin, nous avons utilisé des valeurs numériques en raison de la complémentarité qui existe entre les décisions architecturales pour implémenter une qualité, comme illustré dans l'exemple.

Un autre type de lien est défini dans le modèle de contrat d'architecture, celui-ci précise le degré de formalisation d'une décision architecturale par la contrainte architecturale qui lui est associée. Dans l'implémentation actuelle du contrat d'évolution, le langage [TIB 05] utilisé pour spécifier les contraintes architecturales est une version modifiée d'OCL (*Object Constraint Language*) de l'OMG [OMG 06]. Ce langage est nommé ACL (*Architectural Constraint Language*). Une décision architecturale peut être formalisée par plusieurs contraintes architecturales. On peut donc de la même manière, associer des degrés en valeurs numériques pour chaque contrainte architecturale, et la somme doit être égale à 100 dans le cas idéal bien sûr (le développeur est sur de la complétude de ses contraintes).

Un attribut de qualité dans un contrat représente une propriété nonfonctionnelle conforme aux caractéristiques ou sous-caractéristiques du standard ISO 9126 (maintenabilité, efficacité, utilisabilité, fiabilité). Il possède un degré de criticité spécifié par le développeur indiquant l'importance de l'attribut de qualité dans l'architecture. Les valeurs possibles sont : *very high*, *high*, *medium*, *low*, et *very low*.

9.3.1.2. *Relations entre les attributs qualité dans le contrat*

Le modèle de contrat permet de spécifier les relations entre attributs qualité. Deux types de relations sont définies : « enhances » et « colidesWith ». A titre d'exemple, prendre la décision d'utiliser le style architectural *Pipeline* pour satisfaire la maintenabilité, aura un effet positif et améliore (*enhances*) l'efficacité. Au contraire, un attribut de qualité peut affecter négativement un autre. A titre d'exemple, en voulant apporter des améliorations à l'attribut de qualité sécurité on se trouve généralement en conflit avec, ou opposé à (*collidesWith*) l'attribut de qualité performance. Dans ce dernier cas de figure, une analyse de compromis pour les deux attributs qualité s'avère nécessaire pour minimiser les conflits. Il est recommandé de documenter cette connaissance (relations entre attributs) par le développeur. Nous considérons que, avoir connaissance de ce type d'informations peut servir durant le processus de prise de décision, et aider le développeur à prendre de nouvelles décisions architecturales (lors de l'évolution) avec un minimum d'effets sur la cohérence globale de l'architecture.

D'autres relations peuvent être définies entre attributs qualité. Dans le cas où un attribut (A) influe positivement sur un autre attribut (B), si on l'améliore A, B est amélioré aussi ; et si A est affaibli, B est affaibli aussi. Dans ce cas de figure on dit que l'attribut A est étroitement couplé (*tightly coupled*) avec l'attribut B. On dit que A est faiblement couplé (*weakly coupled*) avec B, si l'on améliore A, B est amélioré ; et lorsque A est affaibli, B n'est pas affecté (voir figure 9.2).

A titre d'exemple, l'ajout d'un composant de cryptage des informations d'authentification avant leur transmission dans une application client/serveur pour des raisons de sécurité, diminue souvent la performance du système. Si l'on veut dans un autre cas de figure, enlever un connecteur vers un composant d'authentification d'une application qui est exécutée avant d'accéder aux services de cette dernière, cette décision affectera positivement l'attribut de qualité performance, car les services sollicités s'exécuteront plus rapidement. On conclut que les deux attributs qualité dans les deux situations sont étroitement liés.

L'exemple suivant illustre deux attributs qualité qui sont faiblement liés. Si l'on réalise l'architecture d'un système qui soit « portable » sur différentes platesformes, on peut être amené à utiliser le patron de conception façade. Cet attribut présente un effet négatif sur l'attribut de qualité disponibilité (Availability). En effet, si le composant qui fournit les services métiers d'une application (composant façade) tombe en panne, les fonctionnalités du système ne seront plus disponibles. Dans un autre cas de figure, et toujours pour des raisons de « portabilité », le développeur d'une application *web* inclut un composant pour faire abstraction de la diversité des navigateurs *web* sur lesquels l'application s'exécute du côté client (Internet Explorer ne traite pas de la même manière que Firefox certains événements). Enlever ce composant d'abstration n'affectera pas l'attribut qualité disponibilité.

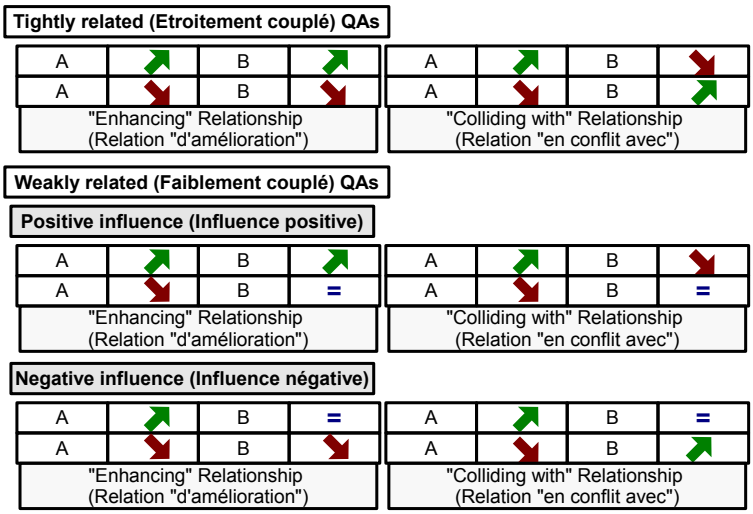

Figure 9.2. *Relations entre attributs qualité*

9.3.1.3. *Exemple d'illustration de contrat*

La représentation du contrat d'évolution est réalisée par le biais d'un document XML. Un contrat d'évolution associé à une architecture logicielle (d'un système à base de composant) donnée est défini par un ensemble de tactiques architecturales. Chacune est constituée d'un couple de décision architecturale et attribut de qualité. Le *listing* ci-dessous présente un exemple de contrat.

```
<evolution-contract id="000001">
     <architecture-tactic id="000100">
       <description>
           Cette tactique garantit la caractéristique de
           qualité portabilité par le biais du patron de
           conception Façade
       </description>
       <quality-attribute id="001000" name="Portabilité"
          characteristic ="Portabilité" degreeOfCriticality="high">
         <description>
           Le composant doit être portable sur différents environnements.
           Il peut servir différents types d'applications clientes.
         </description>
```

```
        <related-quality name="Disponibilité" relationship="enhances"
            relationType="weak" influence="negative">
        </related-quality>
       </quality-attribute>
       <architecture-decision id="010000" degreeOfSatisficing="90">
         <description>
             Patron de conception Façade
         </description>
         <architecture-constraint profile="Fractal" degreeOfFormalizing="80">
            <body>  <!-- Ici on introduit la contrainte-->  </body>
         </architecture-constraint>
       </architecture-decision>
     </architecture-tactic>
</evolution-contract>
```

Cet exemple illustre un contrat d'évolution composé d'une seule tactique architecturale. Celle-ci est représentée par la décision architecturale « patron de conception façade » pour satisfaire la caractéristique de qualité « portabilité ». Nous avons introduit le degré de criticité pour la qualité en question. Dans notre exemple, la valeur *high* est affectée pour indiquer un niveau élevé d'importance. Le contrat montre que l'attribut qualité « portabilité » est faiblement lié à l'attribut qualité « disponibilité » par une relation d'amélioration (*enhances*) à effet négatif. Ces informations sont documentées dans l'élément *related-quality*. Le choix du patron de conception façade satisfait avec 90 % (*degreeOfSatisficing*) les intentions de l'architecte pour la portabilité. La contrainte architecturale qui formalise la décision est écrite avec le profil ACL [TIB 05] pour le modèle de composant Fractal, avec un degré de formalisation de 80 %.

9.3.2. *Algorithme d'assistance*

Le contrat d'évolution est exploité par un algorithme qui assiste les développeurs à chaque pas d'évolution. Ce dernier, assure à la volée au moment de l'application des changements sur l'architecture du système d'alerter le développeur des conséquences des changements sur les différents attributs qualité dans le système (ceux impliqués directement et indirectement par le changement). Cette fonctionnalité est assurée par un mécanisme de notification qui assiste le mainteneur en lui donnant la liberté de décider de valider ou d'ignorer les changements.

9.3.2.1. *Présentation de l'algorithme*

L'algorithme est présenté dans les différents *listings* suivants :

```
(01)   algorithm ArchitectureChangeAssistance {
(02)   let new_Arch := New Architecture
(03)   and AE := Architectural Element
```

```
(04)   and AD := Architectural Decision
(05)   and new_AD := New Architectural Decision
(06)   and AC := Architectural Constraint
(07)   and QA := Quality Attribute
(08)   and new_QA := New Quality Attribute
(09)   and AT := Architecture Tactic (couple composed of a QA and an AD)
(10)   and new_AT := New Architecture Tactic
(11)   and NFR := Non-Functional Requirement
(12)   and affectedQAs := { }
(13)   and noAffectedQAs := true;
(14)   main() {
(15)     for-each new_Arch {
(16)       Contract := Contract associated to changed AE;
(17)       new_AD := ask for AD associated to the new_Arch;
(18)       noAffectedQAs := true;
(19)       for-each (AT in Contract) {
(20)         QA := QA in AT;
(21)         AD := AD in AT;
(22)         CheckArchitecturalConstraint () ;
(23)       }
(24)       AddNewArchitecturalTactic () ;
(25)    }
(26)    CheckAffectedQAs ();
(27)  }
(28)  }
```

A chaque pas d'évolution, un changement architectural est appliqué à la description de l'architecture sous forme d'une décision architecturale. Des éléments architecturaux (AE) sont obligatoirement affectés par ce changement. L'algorithme analyse à chaque pas d'évolution la nouvelle description du contrat (ligne 16 et 17). Il procède ensuite pour chaque tactique dans le contrat (lignes 19, 20, 21 et 22) à l'évaluation des contraintes qui formalisent les décisions architecturales par le biais de la procédure « CheckArchitecturalConstraint » (ligne 22), pour vérifier d'éventuelles violations de contraintes.

```
(01) CheckArchitecturalConstraint () {
(02)   Result := check the AC that formalizes AD;
(03)   if (result == false) {
(04)     AE := AE in the context of AC;
(05)     warn "The following architecture decision " +AD+" is affected.";
(06)     warn " This concerns the architectural element : "+AE;
(07)     warn " The affected architecture decision is  satisficing"+QA
(08)       + " with " +result.degreeOfSatisficing (AT)+" %";
(09)     warn " The affected architecture decision has a degree
(10)       of formalization of"+ result.degreeOfFormalization (AT)+ " %";
(11)     warn " The degree of criticality of this QA is : "
(12)       + result.getQA().degreeOfCriticality;
(13)     warn "Other QAs may be affected. This concerns : " +
(14)       QA_Relationships (QA,"enhances", "tight");
(15)     noAffectedQAs := false;
```

```
(16)     if (new_Arch maintained) {
(17)       affectedQAs := affectedQAs + QA + QA_Relationships(QA,
(18)         "enhances", "tight");
(19)       warn " Contract specification will be changed ...";
(20)       warn " Validate new_Arch ? [Yes | No]";
(21)       Contract := Contract - AT;
(22)       ask to review satisficing degrees of ATs of QA_Relationships(QA,
(23)         "enhances", "tight");
(24)       ask to review NFRs specification;
(25)     }
(26)   }
(27) }
```

La procédure évalue les contraintes formalisant les décisions. Si le résultat est faux (ligne 02 et 03) elle notifie le développeur (lignes 05 jusqu'à 14) de l'impact de son changement sur les autres décisions architecturales ainsi que les qualités affectées, par un certain nombre d'informations (l'élément affecté, décision concernée et son degré de formalisation, degré de satisfaction pour la qualité implémentée, entre autres). Le développeur est assisté aussi avec d'autres informations concernant les attributs qualité en relation avec celle affectée par le changement introduit dans l'architecture. Il est primordial de fournir un maximum d'informations pour pouvoir estimer l'impact des modifications apportées, et guider le développeur dans ses prises de décisions. Pour des raisons de brièveté on affiche les relations d'améliorations étroitement liées avec la qualité modifiée (lignes 13 et 14).

L'algorithme parcourt les relations prédéfinies par le développeur, et affiche toutes les relations qui existent entre la qualité affectée et les autres qui lui sont liées. A titre d'exemple, on peut notifier le développeur, que la qualité A (celle affectée par le changement) est étroitement liée à B, celle-ci possède un degré d'importance bas (*low*) dans le système. L'affaiblissement de A entraîne celui de B (relation d'amélioration). On le notifie de plus, que A est faiblement liée à C, celle-ci possède un degré d'importance très élevé (*very high*) dans le système. De même, l'affaiblissement de A entraîne celui de C. Le développeur peut décider sur la base de ces informations de ne pas valider le changement, vu l'importance de la qualité de C dans le système. Si le développeur décide à un moment donné de maintenir le nouveau changement, l'algorithme lui notifie que la spécification du contrat va changer et lui demande de confirmer l'opération de validation (lignes 16 à 20). Une fois la spécification de contrat de la nouvelle architecture est maintenue, la procédure met à jour le contrat en éliminant la tactique correspondante (ligne 21), demande la révision des degrés de satisfaction des attributs qualité affectés (ligne 22) ainsi que la spécification des NFR (ligne 24) permettant ainsi de maintenir la documentation des propriétés de qualité à jour. L'algorithme doit ensuite, après avoir analysé l'ensemble des tactiques, mettre à jour à nouveau le contrat en ajoutant la nouvelle tactique (ligne 12) dans la procédure « AddNewArchitecturalTactic ». Cette dernière notifie le développeur éventuellement des qualités qui peuvent être en conflit avec la nouvelle qualité implémentée (lignes 08 et 09). Elle permet aussi de garder la documentation des NFR à jour (ligne 13).

```
(01)  AddNewArchitecturalTactic () {
(02)    if (new_Arch generates a new AD) {
(03)      new_QA := ask for QA associated to new_AD;
(04)      new_AT := AT(new_QA, new_AD);
(05)      if (new_QA is in affectedQAs)
(06)        affectedQAs := affectedQAs - new_QA;
(07)      else {
(08)        warn "Other QAs may conflict with "+new_QA+": ";
(09)        warn QA_Relationships (QA,"collidesWith","both");
(10)      }
(11)      warn "Contract specification will be changed ...";
(12)      Contract := Contract + new_AT;
(13)      ask to change NFRs specification;
(14)    }
(15)  }
```

Finalement, l'algorithme d'assistance vérifie de nouveau la liste des qualités affectées (préalablement construite par la procédure « CheckArchitecturalConstraint() ») et notifie le développeur s'il y a d'autres qualités qui sont toujours affectées par le changement apporté, lui demande de réviser leur degré de satisfaction ainsi que la spécification des NFR. Cette tâche est réalisée par la procédure CheckAffectedQAs() (voir le *listing* ci-dessous).

```
(01)  CheckAffectedQAs () {
(02)    if (affectedQAs <> {} ) {
(03)      for-each (QA in affectedQAs) {
(04)        warn QA + "is still affected by your changes";
(05)        ask to review satisficing degrees of ATs implying QA;
(06)      }
(07)      ask to review NFRs specification;
(08)    }
(09)  }
```

9.3.2.2. *Exemple d'illustration*

Le fonctionnement de l'algorithme est illustré sur un exemple d'une application à base de composants, qui représente un système de contrôle d'accès à un musée (MACS). Un aperçu de son architecture est fourni dans la figure 9.3.

Le contrat d'évolution associé à cette application se compose de sept tactiques architecturales (AT) qui répondent à différents besoins de qualité définis dans les documents de spécification. Les AT documentent les liens entre les décisions architecturales (AD1, AD2, AD3, AD4, AD5, AD6) et leurs attributs qualité correspondants (QA1, QA2, QA3, QA4, QA5). Nous présentons dans ce qui suit une brève description des AT et des décisions architecturales associées.

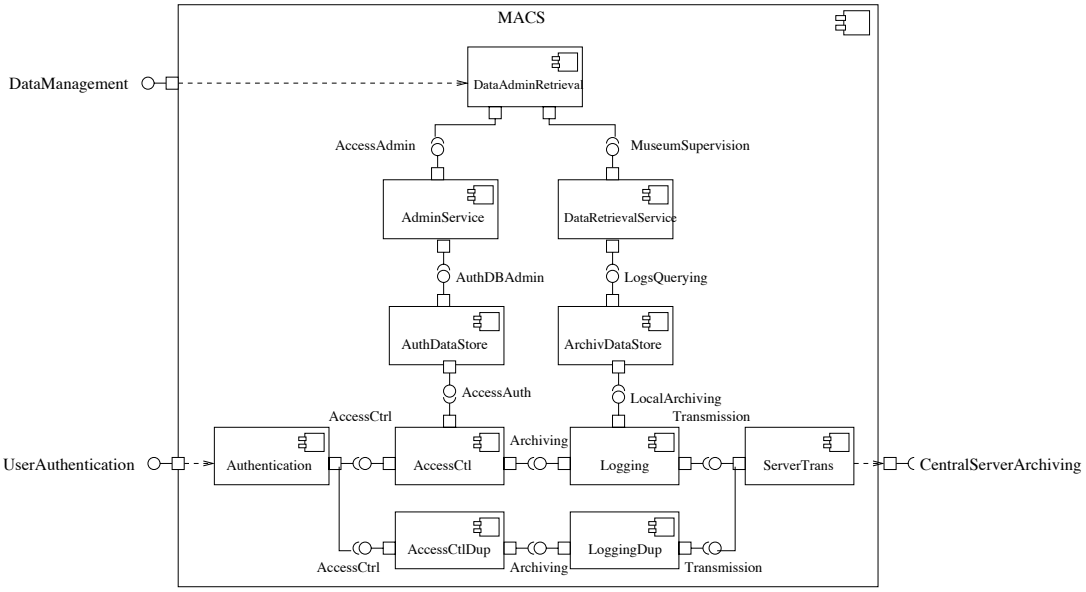

Figure 9.3. *Une architecture simplifiée d'un système de contrôle d'accès à un musée*

QA1 : maintenabilité	AD1 : choix du patron architectural système en couches, pour une décomposition verticale des composants de l'application MACS. AD4 : choix d'un style en *pipeline* pour organiser la séquence de composants dupliquée (en bas de la figure 9.3). AD5 : implémente le composant d'abstraction de données « DataRetrievalService » .
QA2 : portabilité	AD2 : choix du patron de conception façade, représenté par le composant « DataAdminRetrieval » .
QA3 : disponibilité	AD3 : implémente un schéma de redondance. Dupliquer les composants « AccessCtl » et « Logging » (« AccessCtlDup » et « LoggingDup »).
QA4 : performance	AD4 : choix d'un style en *pipeline*, pour organiser la séquence de composants dupliqués.
QA5 : sécurité	AD6 : implémente le composant d'authentification (*Authentication*).

Avant d'appliquer l'algorithme d'assistance, le développeur est amené à établir les différentes relations entre les attributs qualité (définis dans la spécification des NFR) propre à l'application au cours de développement, afin de les intégrer dans la documentation du contrat. Cette tâche fait intervenir l'expérience du développeur, ainsi que d'autres informations issues des études statistiques sur d'autres projets. On suppose l'existence des relations prédéfinies suivantes :

– QA4 et QA1 :{relationship= « collidesWith »,relationType= « weak », influence= « positive » }

– QA2 et QA3 :{relationship= « enhances »,relationType= « weak », influence= « negative » }

– QA3 et QA4 :{relationship= « enhances »,relationType= « weak », influence= « negative » }

– QA3 et QA1 :{relationship= « enhances »,relationType= « weak », influence= « negative » }

– QA5 et QA4 :{relationship= « collidesWith »,relationType= « tight » }

A titre d'exemple, dans le contexte de notre application, la première relation montre que la performance (QA4) et la maintenabilité (QA1) sont des attributs qui sont en conflit, faiblement liés, et la performance influe positivement sur la maintenabilité (l'amélioration de QA4 entraîne l'affaiblissement de QA1). La dernière relation précise que la sécurité (QA5) et la performance sont deux qualités en conflit, mais étroitement liées.

Appliquons maintenant l'algorithme d'assistance selon un scénario d'évolution qui comporte les changements architecturaux suivants :

– ACG1 : créer un lien entre le composant AccessCtl et ServerTrans, et le composant AccessCtlDup et ServerTrans. L'objectif est d'améliorer la performance du système (ne pas avoir à journaliser toutes les informations par le composant Logging). L'algorithme vérifie les contraintes ACL des décisions de chaque tactique dans le contrat, et détecte que la contrainte qui formalise la décision AD4 (le *pipeline*) à été violée. Par conséquent, les attributs qualité QA1 et QA4 sont affectés. Il informe d'abord le développeur que le degré de formalisation d'AD4 est de 75 %, satisfait QA4 à 60 % qui a un niveau d'importance élevé, que (QA4) et la maintenabilité (QA1) sont en conflit, faiblement liées, et que selon la relation « QA4 et QA1 » la performance influe positivement sur la maintenabilité (l'amélioration de QA4 entraîne l'affaiblissement de QA1). Il informe ensuite le développeur que la décision a un degré de formalisation de 85 %, satisfait de 30 % la maintenabilité, cette dernière a un niveau d'importance moyen. Le développeur décide alors, sachant que le changement apporté influe négativement sur QA1, de valider le changement apporté. La tactique associée à la décision affectée est supprimée du contrat, et la nouvelle tactique est ajoutée. L'algorithme demande au développeur est demandé de mettre à jour la spécification des NFR, et de réviser les degrés de satisfaction des qualités affectées ;

– ACG2 : ajouter un composant DB_UpdateNotification qui représente un service de notification. Il exporte une interface *publish/subscribe* à travers le port qui fournit l'interface DataManagement et utilise directement le composant ArchivDataStore. Le développeur est notifié que la décision AD2 et la qualité qui lui est associée QA2 sont probablement affectées. L'algorithme l'informe que le degré de formalisation de la décision est de 80 %, satisfait de 90 % la portabilité (QA2) qui à un niveau d'importance élevé (*high*), et que l'attribut qualité portabilité est faiblement lié à l'attribut qualité disponibilité (QA3) par une relation d'amélioration (*enhances*) à effet négatif (relation QA2 et QA3). Le développeur décide de valider le changement tout en ayant connaissance de l'importance de QA2 et son influence sur la disponibilité de tout le système (la défaillance du composant façade DataAdminRetrieval entraîne la non-disponibilité d'une partie importante du système) ;

– ACG3 : éliminer le composant LoggingDup de l'architecture du système, pour éviter le problème de cohérence de données avec le composant ArchivDataStore. Le développeur est notifié que AD3 est affectée et que la qualité qu'elle implémente QA3 est aussi altérée. L'algorithme d'assistance fournit les informations suivantes : degré de formalisation d'AD3 de 70 %, satisfait de 85 % QA3, qui à un niveau d'importance élevé. L'algorithme détecte la dépendance de cette dernière avec la performance (QA4) d'après la relation « QA3 et QA4 », et informe le développeur que QA3 est faiblement lié à l'attribut qualité QA4 par une relation d'amélioration (*enhances*) à effet négatif. Cela implique que la performance se dégrade par la dégradation de la disponibilité. D'autre part, sachant qu'AD3 est liée à AD4, l'algorithme d'assistance informe le développeur que QA1 (maintenabilité) et QA4 (performance) sont aussi affectées. D'après la relation « QA3 et QA1 » il notifie que la dégradation de la disponibilité entraîne celle de la maintenabilité. Le développeur décide alors d'ignorer le changement ;

– ACG4 : ajout d'un composant de cryptage DataEncryption qui a pour rôle le cryptage des données d'authentification avant leur passage vers le composant Authentication par le biais d'un nouveau connecteur. L'objectif est d'améliorer le niveau de sécurité. L'algorithme d'assistance détecte que la contrainte formalisant l'AD5 a été violée. Il notifie le développeur qu'AD5 et la qualité QA5 sont affectées par le changement introduit sur l'architecture, et l'informe que le degré de formalisation d'AD5 est de 90 %, satisfait de 40 % la sécurité (QA5) qui a un degré d'importance très élevé (*very high*), et que selon la relation « QA5 et QA4 » QA5 et la performance (QA4) sont deux qualités en conflit, mais étroitement liées. Etant donné que la sécurité est plus importante dans le contexte de notre application, le développeur décide de valider le changement. De même, le développeur est invité à faire des révisions sur les degrés de satisfaction des qualités affectées, et de revoir la spécification des NFR.

Nous avons montré, à travers cet exemple, le déroulement de l'algorithme d'assistance à l'évolution proposée dans notre approche. Une partie de l'algorithme a été implémentée dans l'outil AURES (*ArchitectURe Evolution aSsistant* [TIB 06b]). La

partie liée aux différents degrés associés aux éléments du contrat, ainsi que les relations entre attributs qualité reste à implémenter.

9.4. Conclusion : contribution et perspectives

Dans ce chapitre, nous avons évoqué quelques problèmes liés au processus de maintenance dans le domaine des architectures logicielles. Ces problèmes, qui démeurent à coût élevé, émergent au moment de l'application de nouveaux choix architecturaux venant en réponse aux changements des besoins. En partant du fait que la forme de l'architecture logicielle est façonnée par la volonté de vouloir aboutir à certaine qualité par certaines décisions architecturales, il est utile de penser à assister le développeur dans le choix de ces décisions. L'objectif de cette assistance est de minimiser les effets négatifs sur les qualités initialement établies dans les documents de spécification, et qui sont dus aux changements apportés durant le processus de développement. Ceci permet entres autre de réduire les coûts causés par les allers et retours entre la phase de test et celle de développement. L'approche proposée dans cet article, permet de répondre à ce besoin. Elle fait usage d'une documentation basée sur le concept de contrat, nommé « contrat d'évolution ». Celle-ci documente les décisions architecturales et les qualités qu'elles visent. Une telle représentation offre une meilleure compréhension de l'architecture logicielle, ce qui permet aussi d'améliorer la compréhension globale d'un système avant la mise en place des changements.

A cet effet, pour éclaircir l'approche, un état de l'art a été présenté. Celui-ci englobe les travaux sur la documentation de la qualité logicielle, la documentation des décisions architecturales, et finalement les travaux sur l'assistance à l'évolution du logiciel. Dans la première partie, deux catégories de travaux ont été présentées, les modèles de qualité, et les approches de documentation des attributs qualité dans les architectures logicielles. Dans la deuxième partie, nous avons séparé entre les travaux sur la documentation des décisions architecturales dans la description d'architectures, en se focalisant principalement sur les ADL, et ceux annexes aux descriptions d'architectures, qui tendent à expliciter la notion de décision architecturale comme entité de première classe. La dernière partie résume les travaux essentiellement basés sur le concept de contrat, pour l'assistance à l'évolution d'un logiciel.

Le travail sur l'assistance à l'évolution présenté dans ce chapitre, combine un contrat d'évolution, et un algorithme d'assistance qui exploite le contrat. Le contrat documente l'expertise du développeur, sous forme d'un ensemble de décisions architecturales qui concerne une application à base de composants.

Comme perspectives à ce travail, nous envisageons d'élargir la portée de ces contrats pour des architectures d'applications orientées service (SOA) ou des architectures SCA (*Service Component Architectures*). Dans ce cas, de nouvelles sortes de décisions architecturales et d'attributs qualité, inhérents au concept de service,

devront être gérées. Nous projetons par ailleurs de faire une étude de cas sur un projet SOA/SCA réel afin de valider ce travail.

9.5. Bibliographie

[ALL 97] ALLEN R., A Formal Approach to Software Architecture, Thèse de doctorat, Université de Carnegie Mellon, Pittsburgh, PA, Etats-Unis, mai 1997.

[BAL 91] BALASUBRAMANIAM R., VASANT D., Representation and Maintenance of Process Knowledge for Large Scale Systems Development, *Proceeding of the 6th Knowledge-based Software Engineering Conference, KBSE*, New York, Etats-Unis, p. 223-231, septembre 1991.

[BAS 01] BASS L., BACHMANN F., KLEIN M., Quality Attribute Design Primitives and the Attribute Driven Design Method, *Proceeding of the 4th International Conference on Product Family Engineering*, Bilbao, Espagne, Springer-Verlag, p. 169-186, octobre 2001.

[BAS 03] BASS L., CLEMENTS P., KAZMAN R., *Software Architecture in Practice, 2nd Edition*, Addison-Wesley, Boston, MA, Etats-Unis, 2003.

[BAS 06] BASS L., CLEMENTS P., NORD R. L., STAFFORD J., *Capturing and Using Rationale for a Software Architecture*, p. 255-272, Springer, in a. h. dutoit et al., rationale management in software engineering édition, 2006.

[BOE 76] BOEHM B. W., BROWN J. R., LIPOW M., Quantitative evaluation of software quality, *Proceeding of the 2nd International Conference on Software Engineering*, San Francisco, California, Etats-Unis, IEEE Computer Society Press, p. 592-605, 1976.

[BRO 06] BROY M., DEISSENBOECK F., PIZKA M., Demystifying maintainability, *Proceeding of the 2006 international workshop on Software quality (WoSQ'06)*, Shanghai, Chine, ACM Press, p. 21-26, 2006.

[BUR 06] BURGE J., BROWN D., *Rationale-based Support for Software Maintenance*, Chapitre Rationale Management in Software Engineering, p. 273-296, Springer-Verlag, in a. dutoit, r. mccall, i. mistrik, and b. paech édition, 2006.

[CAP 07] CAPILLA R., NAVA F., DUENAS J. C., Modeling and Documenting the Evolution of Architectural Design Decisions, *Proceeding of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI'07)*, Washington, DC, Etats-Unis, IEEE Computer Society, 2007.

[CHU 99] CHUNG L., NIXON B. A., YU E., J. M., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 1999.

[CLE 03] CLEMENTS P., BACHMANN F., BASS L., GARLAN D., IVERS J., LITTLE R., NORD R., STAFFORD J., *Documenting Software Architectures, Views and Beyond*, Addison-Wesley, 2003.

[CYS 04] CYSNEIROS L. M., SAMPAIO DO PRADO LEITE J. C., Nonfunctional Requirements : From Elicitation to Conceptual Models, *IEEE Transactions on Software Engineering*, vol. 30, n° 5, p. 328-350, IEEE Computer Society Press, 2004.

[DEI 07] DEISSENBOECK F., WAGNER S., PIZKA M., TEUCHERT S., GIRARD J.-F., An activity-based quality model for maintainability, *Proceeding of the 23rd International Conference on Software Maintenance (ICSM '07)*, IEEE Computer Society, p. 184-193, 2007.

[DEI 09] DEISSENBOECK F., JUERGENS E., LOCHMANN K., WAGNER S., Software Quality Models : Purposes, Usage Scenarios and Requirements, *Proceeding of 7th International Workshop on Software Quality (WoSQ '09)*, IEEE Computer Society, 2009.

[DRO 95] DROMEY R. G., A Model for Software Product Quality, *IEEE Transactions on Software Engineering*, vol. 21, n° 2, p. 146-163, IEEE Computer Society Press, 1995.

[DRO 96] DROMEY R. G., Cornering the Chimera, *IEEE Software*, vol. 13, p. 33-43, IEEE Computer Society, 1996.

[EIC 01] EICK S. G., GRAVES T. L., KARR A. F., MARRON J. S., MOCKUS A., Does Code Decay ? Assessing the Evidence from Change Management Data, *IEEE Transactions on Software Engineering*, vol. 27, n° 1, p. 1-12, 2001.

[GAR 00] GARLAN D., MONROE R. T., WILE D., Acme : Architectural Description of Component-Based Systems, LEAVENS G. T., SITARAMAN M., Eds., *Foundations of Component-Based Systems*, p. 47-68, Cambridge University Press, 2000.

[GEO 03] GEORGIADOU E., GEQUAMO—A Generic, Multilayered, Customisable, Software Quality Model, *Software Quality Control*, vol. 11, n° 4, p. 313-323, Kluwer Academic Publishers, 2003.

[GRA 92] GRADY R. B., *Practical software metrics for project management and process improvement*, Prentice Hall, 1992.

[HOC 05] HOCHSTEIN L., LINDVALL M., Combating Architectural Degenration : A Survey, *Information and Software Technology*, vol. 47, n° 10, p. 693-707, Elsevier, juillet 2005.

[IEE 00] IEEE, ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[ISO 01] ISO, Software Engineering - Product quality - Part 1 : Quality model, International Organization for Standardization. ISO/IEC 9126-1., 2001.

[JAC 02] JACKSON D., Alloy : a lightweight object modelling notation, *ACM Transactions on Software Engineering and Methodoly*, vol. 11, p. 256-290, ACM, avril 2002.

[JAN 05] JANSEN A., BOSCH J., Software Architecture as a Set of Architectural Design Decisions, *Proceeding of of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, 2005.

[JAN 08] JANSEN A., Architectural design decisions, Thèse de doctorat, Université de Groningen, Institute for Mathematics and Computing Science, août 2008.

[JIN 89] JINTAE L., Decision representation language (DRL) and its support environment, Rapport, Laboratoire de l'Intelligence Artificielle MIT, août 1989.

[KIM 09] KIM S., KIM D.-K., LU L., PARK S., Quality-driven architecture development using architectural tactics, *Journal of Systems and Software*, vol. 82, n° 8, p. 1211–1231, août 2009.

[KIM 10] KIM J. S., GARLAN D., Analyzing architectural styles, *Journal of Systems and Software*, vol. 83, n° 7, p. 1216-1235, Elsevier, 2010.

[KIT 97] KITCHENHAM B., LINKMAN S., PASQUINI A., NANNI V., The SQUID approach to defining a quality model, *Software Quality Control*, vol. 6, n° 3, p. 211-233, Kluwer Academic Publishers, 1997.

[KLA 09] KLAS M., HEIDRICH J., MÂ¸NCH J., TRENDOWICZ A., CQML Scheme : A Classification Scheme for Comprehensive Quality Model Landscapes, *Proceeding of the 35th EUROMICRO Conference Software Engineering and Advanced Applications*, IEEE Computer Society, p. 243-250, 2009.

[KRU 95] KRUCHTEN P., The 4+1 View Model of Architecture, *IEEE Softw.*, vol. 12, n° 6, p. 42-50, IEEE Computer Society Press, 1995.

[KRU 04] KRUCHTEN P., An Ontology of Architectural Design Decisions in Software Intensive Systems, *Proceeding of the 2nd Groningen Workshop Software Variability*, p. 54-61, 2004.

[KRU 06] KRUCHTEN P., LAGO P., VAN VLIET H., Building up and reasoning about architectural Knowledge, *Proceedings of the Second International Conference on the Quality of Software Architectures, QoSA*, Lecture Notes in Computer Science 4214, Springer-Verlag, p. 43-58, 2006.

[KRU 09] KRUCHTEN P., CAPILLA R., DUENAS J. C., The Decision View's Role in Software Architecture Practice, *IEEE Software*, vol. 26, n° 2, p. 36-42, IEEE Computer Society Press, 2009.

[LAG 05] LAGO P., VAN VLIET H., Explicit Assumptions enrich Architectural Models, *Proceeding of the 27th International Conference on Software Engineering (ICSE'05)*, ACM Press, p. 206-214, mai 2005.

[LIN 02] LINDVALL M., TESORIERO R., COSTA P., Avoiding Architectural Degeneration : An Evaluation Process for Software Architecture, *Proceeding of the IEEE Symposium on Software Mertrics (METRICS'02)*, Ottawa, Ontario, Canada, p. 77-86, Juin 2002.

[MAD 03] MADHAVJI N. H., TASSÉ J., Policy-guided Software Evolution, *Proceeding of the 19th International Conference on Software Maintenance (ICSM'03)*, IEEE Computer Society Press, p. 75-82, 2003.

[MAR 04] MARINESCU R., RATIU D., Quantifying the Quality of Object-Oriented Design : The Factor-Strategy Model, *Proceeding of the 11th Working Conference on Reverse Engineering (WCRE'04)*, IEEE Computer Society, p. 192-201, 2004.

[MAR 09] MAREW T., LEE J.-S., BAE D.-H., Tactics based approach for integrating non-functional requirements in object-oriented analysis and design, *Journal of Systems and Software*, vol. 82, n° 10, p. 1642-1656, Elsevier Science Inc., 2009.

[MCC 77] MCCALL J., RICHARDS P., WALTERS G., Factors in Software Quality, Rapport de recherche, (RADC)- TR-77-369, Vols. 1–3, Rome Air Development Center, Etats-Unis Air Force, Hanscom AFB, MA, 1977.

[MEN 00] MENS T., D'HONDT T., Automating Support for Software Evolution in UML, *Automated Software Engineering Journal*, vol. 7, n° 1, p. 39-59, Springer-Verlag, 2000.

[MER 10] MERKLE B., Stop the Software Architecture Erosion, Tutorial in SPLASH'10, Reno, Nevada, Etats-Unis, 2010.

[MON 01] MONROE R. T., Capturing Software Architecture Design Expertise with Armani, Rapport, School of Computer Science, Université de Carnegie Mellon , Pittsburgh, Pennsylvanie, Etats-Unis, 2001.

[MYL 92] MYLOPOULOS J., CHUNG L., NIXON B., Representing and Using Nonfunctional Requirements : A Process-Oriented Approach, *IEEE Transactions on Software Engineering*, vol. 18, n° 6, p. 483-497, Juin 1992.

[NIE 07] NIEMELÄ E., IMMONEN A., Capturing quality requirements of product family architecture, *Information and Software Technology*, vol. 49, n° 11-12, p. 1107-1120, Butterworth-Heinemann, 2007.

[OMG 06] OMG, Object Constraint Language Specification, Version 2.0, Document formal/2006-05-01, Object Management Group Web Site : http ://www.omg.org/spec/OCL/2.0/PDF, 2006.

[PAR 94] PARNAS D. L., Software Aging, *In Proceeding of the 16th International Conference on Software Engineering (ICSE'94*, Sorrento, Italie, IEEE Computer Society Press and ACM Press, mai 1994.

[PER 92] PERRY D. E., WOLF A. L., Foundations for the Study of Software Architecture, *SIGSOFT Software Engineering Notes*, vol. 17, n° 4, p. 40-52, ACM SIGSOFT, 1992.

[RAJ 05] RAJAN H., SULLIVAN K. J., Classpects : unifying aspect- and object-oriented language design, *Proceeding of the 27th international conference on Software engineering (ICSE'05)*, St. Louis, MO, Etats-Unis, ACM, p. 59-68, 2005.

[RAU 00] RAUSCH A., Software Evolution in Componentware Using Requirements/Assurances Contracts, *Proceeding of the 22nd International Conference on Software Engineering (ICSE'00)*, ACM Press, p. 147-156, 2000.

[STE 96] STEYAERT P., LUCAS C., MENS K., D'HONDT T., Reuse contracts : managing the evolution of reusable assets, *In Proceeding of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, ACM Press, p. 268-285, 1996.

[TIB 05] TIBERMACINE C., FLEURQUIN R., SADOU S., Preserving Architectural Choices throughout the Component-based Software Development Process, *Proceeding of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, Pittsburgh, Pennsylvanie, Etats-Unis, IEEE Computer Society Press, p. 121-130, novembre 2005.

[TIB 06a] TIBERMACINE C., Contractualisation de l'évolution architecturale de logiciels à base de composants : une approche pour la préservation de la qualité, Thèse de doctorat, Université Bretagne-Sud, 2006.

[TIB 06b] TIBERMACINE C., FLEURQUIN R., SADOU S., On-Demand Quality-Oriented Assistance in Component-Based Software Evolution, *Proceeding of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06)*, Vasteras, Suède, Springer LNCS, Juin 2006.

[TYR 05] TYREE J., AKERMAN A., Architecture Decisions : Demystifying Architecture, *IEEE Software*, vol. 22, n° 2, p. 19-27, March/April 2005.