ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE ÉLECTRIQUE
M. Ing.

PAR
PATRICK HODOUL

ÉTUDE DE QUALITÉS ARCHITECTURALES DANS UN CONTEXTE INDUSTRIEL

MONTRÉAL, LE 7 SEPTEMBRE 2004

CE MÉMOIRE A ÉTÉ ÉVALUÉ
PAR UN JURY COMPOSÉ DE :

M. François Coallier, directeur de mémoire
Département de génie logiciel et des TI à l'École de technologie supérieure

M. Roger Champagne, codirecteur
Département de génie logiciel et des TI à l'École de technologie supérieure

M. Christopher Fuhrman, président du jury
Département de génie logiciel et des TI à l'École de technologie supérieure

M. Mikel Doucet, P. Eng., M. Eng.
Manager, Center of Competence, Software Engineering, Group Engineering
Bombardier Transportation

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC
LE 10 AOÛT 2004
À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ÉTUDE DE QUALITÉS ARCHITECTURALES DANS UN CONTEXTE INDUSTRIEL

Patrick Hodoul

SOMMAIRE

La conception d'une nouvelle architecture logicielle est toujours une étape importante et risquée de la conception d'un nouveau produit logiciel. Ainsi, l'entreprise ABC souhaite pouvoir réduire ce risque. La proposition faite à l'ÉTS consiste donc à utiliser cette problématique pour concevoir des outils pour simplifier et faciliter la conception d'une nouvelle architecture logicielle en se basant sur les attributs de qualité.

Notre recherche bibliographique nous a permis de trouver des méthodes qui couvrent cette problématique. Ces méthodes améliorent les processus tels que le processus unifié de Rational (RUP), pour mieux considérer les attributs de qualité lors de l'analyse de l'architecture logicielle, mais semblent porter peu d'attention à la qualité des artéfacts. Ainsi, nous avons amélioré certains concepts existants et proposé de nouveaux concepts dans le but d'améliorer cette qualité. De plus, nous avons introduit une étape intermédiaire basée sur des spécifications architecturales pour essayer de simplifier ces méthodes.

Notre proposition est une méthode systématique, complémentaire aux méthodes existantes, de définition des attributs de qualité dans le contexte de l'analyse d'architecture logicielle qui devrait répondre à notre problématique.

STUDY OF QUALITY ATTRIBUTES IN AN INDUSTRIAL CONTEXT

Patrick Hodoul

ABSTRACT

The software architecture design step is always a critical and risky phase during the design of a new software product. Hence, ABC Corporation wishes to reduce this risk. Our goal is to use this context to propose tools to simplify this step and decrease the associated risk.

Following our literature review, it appears that existing methods already offer the required techniques to support this step. These methods usually extend existing processes such as Rational Unified Process (RUP) to tackle the quality attribute impacts during the software architecture analysis. However, some weaknesses are still present. In order to address them, we first improve some existing concepts and introduce new ones, and we also extend existing methods by introducing an intermediary step based on specifications for software architecture. We hope that these two additions will simplify this step and reduce the risk.

Our proposition used in conjunction with existing methods, is a systematic method which should produce better quality attributes definitions.

REMERCIEMENTS

Je remercie sincèrement mon directeur de mémoire François Coallier, directeur du département de génie logiciel et des TI à l'ÉTS (École de technologie supérieure) sans lequel ce projet n'aurait pas vu le jour. Il me faut également remercier mon codirecteur Roger Champagne, professeur dans le même département. Ces deux professeurs m'ont toujours guidé, soutenu et encouragé durant mon année de recherche. De plus, je voudrais aussi les remercier pour la bourse qu'ils m'ont attribuée dans le but de finir ce mémoire.

Je tiens aussi à remercier l'ÉTS, pour la bourse d'excellence que j'ai reçue pour l'année 2004.

Je remercie les professeurs et les étudiants du GÉLOG qui m'ont laissé une place dans ce laboratoire pour toute la durée de mon projet.

Finalement, je tiens à remercier tout particulièrement ma petite famille : Catherine sans laquelle cette idée de reprendre mes études n'aurait jamais pu être possible et ma petite fille adorée Emmanuelle.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS

ÉTS	École de technologie supérieure
ABC	Nom fictif de notre partenaire industriel
COTS	Commercial Off The Shelf
ADD	Attribute-Driven Design
ATAM	Architecture Tradeoff Analysis Method
ADQA	Architecture Design based on Quality Attributes
QAAs	Quality Attribute Assessment method (Phase 1 de ADQA)
QAAg	Quality Attribute Aggregation method (Phase 2 de ADQA)
AV	Architecture Validation method (Phase 3 de ADQA)
RUP	Processus unifié de Rational[1]
SAAM	Software Architecture Analysis Method

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

INTRODUCTION

Cette recherche a été développée à partir d'un besoin spécifique de notre partenaire industrie[2]l exprimé par leurs appréhensions concernant la conception d'une nouvelle architecture logicielle.

Suite à une recherche bibliographique, nous avons généralisé ce besoin pour répondre à des manques présents actuellement dans les méthodes de conception des architectures logicielles se basant sur des scénarios pour modéliser les exigences non fonctionnelles. Ces manques concernent : 1) le contrôle de la qualité de la décomposition des exigences non fonctionnelles; 2) la manière de concevoir une architecture à partir d'une liste de scénarios; 3) la validation du résultat par rapport à son contexte. En effet, malgré l'utilisation de l'une ou l'autre des méthodes présentées dans la littérature, un architecte expérimenté est toujours indispensable car il y a très peu de support. Or un architecte de ce type n'est pas toujours disponible et le risque de « rater » une architecture logicielle est toujours très présent, ce qui pourrait être dommageable pour un produit logiciel. Ainsi dans le but de diminuer le risque, nous proposons une méthode systématique pour aider les architectes en simplifiant cette transition et en portant plus d'attention sur la qualité des artéfacts.

Cette étude est décomposée en trois parties : la première est une recherche bibliographique, la deuxième fournit une explication détaillée de notre proposition et la troisième donne les conclusions.

En conclusion, au lieu de fournir simplement des gabarits de document, nous allons fournir une méthode systématique. Ainsi, nous espérons combler un manque en proposant une méthode originale complémentaire aux méthodes existantes.

---

[2] Dans le but de préserver l'anonymat de notre partenaire industriel, le document utilisera toujours un nom fictif de compagnie. Le nom choisit est « l'entreprise ABC ».

CHAPITRE 1

MISE EN PLACE DE LA RECHERCHE

1.1 Méthode de travail

La méthode de travail utilisée pour cette recherche a été décomposée en plusieurs étapes indispensables pour bien couvrir ce problème. La première étape concerne une recherche dans la littérature de ce qui existe sur les attributs de qualité et les méthodes de conception architecturales pour discuter de leurs forces et de leurs faiblesses. La deuxième étape propose une méthode qui réponde à notre contexte. La troisième étape donne la conclusion de cette recherche.

La première étape est indispensable car toute recherche doit d'abord commencer par une étude de ce qui existe dans la littérature (voir le chapitre 2 et le chapitre 3). En effet, il existe déjà des méthodes pour couvrir cette problématique. Ces méthodes fournissent certainement des réponses. Cette partie se termine donc par une synthèse des recherches (voir le chapitre 4) qui, entre autres, aborde les forces et les faiblesses découvertes et en discute les conséquences par rapport à notre contexte.

La deuxième partie explique notre proposition (voir le chapitre 5), qui se décompose en trois phases. Ces trois phases s'intègrent dans les méthodes existantes et sont autonomes pour le cas où une seule phase serait nécessaire (voir le chapitre 6).

La troisième étape donne les conclusions de cette recherche et explique les points à étudier dans le cadre de recherches complémentaires. En effet, les circonstances de cette recherche ont fait que l'expérimentation n'a pu être réalisée, ce qui aurait validé la proposition. De plus, l'ampleur du travail a imposé de réduire l'étendue de la recherche.

En bref, la méthode de travail nous permet de toujours baser notre recherche par rapport à ce qui existe et par rapport au contexte de l'entreprise ABC. Maintenant, il convient de bien définir le contexte de l'étude elle-même avant de présenter le sujet.

1.2 Contexte de la recherche

Le contexte de l'entreprise ABC est un peu particulier, car, ces dernières années, elle a effectué de nombreuses acquisitions pour renforcer son activité. Entre autres, l'entreprise a acquis une compagnie en Europe, ce qui a eu pour effet de doubler ses effectifs et d'amener de nouvelles manières de travailler.
Il existe donc de grosses divisions telles que RCS avec 550 employés sur un total de 1100. Ces divisions ont leurs propres processus parfaitement adaptés à un contexte d'entreprise indépendante.

Ainsi, l'entreprise a identifié un besoin criant d'unification des processus, des environnements et des outils pour réduire les coûts et les incompatibilités. Par la même occasion, elle souhaite introduire des concepts plus modernes. Plusieurs initiatives en collaboration avec l'ÉTS sont en cours pour formaliser et mettre en place cette vision d'unification et de mise à jour. Ainsi, plusieurs professeurs sont présentement actifs au sein de notre partenaire. Cette étude se situe dans le cadre de l'une de ces initiatives autour du développement et de la maintenance de logiciels.

Donc, l'entreprise ABC souhaite harmoniser les processus. Ce souci d'harmonisation est aussi très présent au niveau du logiciel en particulier pour l'architecture logicielle. Profitant de ce processus de réingénierie, l'entreprise souhaite introduire des aspects plus novateurs pour préparer l'architecture logicielle à supporter de nouvelles demandes et pour accélérer le cycle de développement logiciel. Ces aspects sont les lignes de produits et les composants.

La figure 1 résume l'état actuel et la vision de l'entreprise ABC concernant les processus et l'environnement. Actuellement, il existe plusieurs processus dans des environnements différents (outils et technologies différentes) que l'entreprise voudrait unifier vers un processus unique dans un seul environnement (en laissant la place à des variantes pour mieux s'adapter à des besoins particuliers).

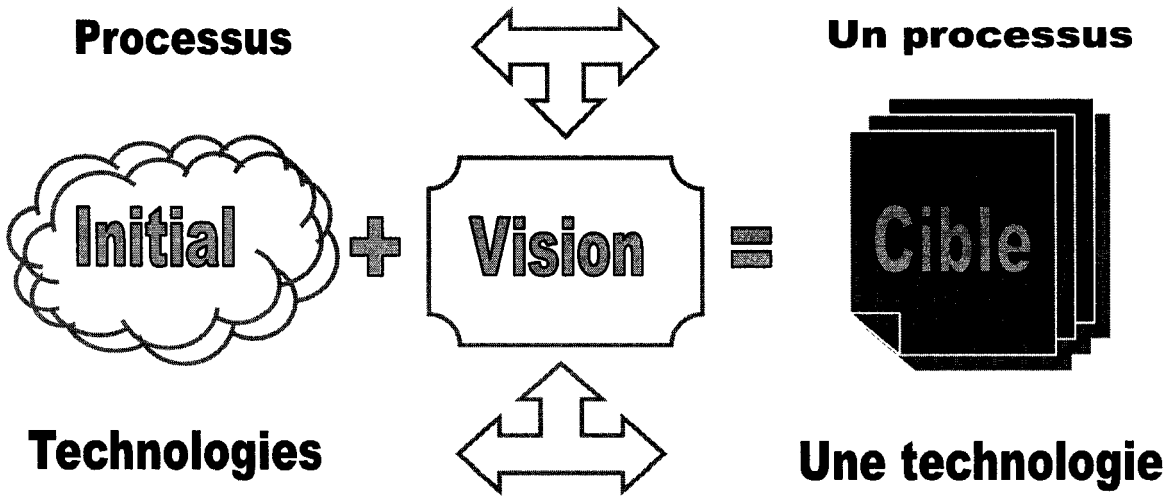

Figure 1 Vision pour les processus

Cependant, l'entreprise est consciente des limitations de cette vision du fait des contextes très différents du développement logiciel. Par exemple, certains développements ont des contraintes très spécifiques liées aux systèmes temps réel ou embarqués, imposant certains outils très spécialisés, ce qui montre bien les limites d'une unification complète.

En bref, si toutes les activités liées au développement logiciel sont considérées dans cette réforme, l'architecture logicielle doit aussi apporter ses propres solutions (pour mieux s'intégrer avec des systèmes).

Cette étude portera essentiellement sur l'architecture logicielle. L'entreprise ABC veut faire migrer des lignes de produits existantes et indépendantes vers une architecture standardisée et actualisée avec une attention toute particulière sur les techniques et les technologies modernes et les aspects « lignes de produits » et « composants ». Cette mise à jour aura des impacts importants sur l'architecture logicielle. Ainsi, l'entreprise veut étudier les solutions architecturales possibles répondant à ce nouveau cadre. De plus, l'entreprise ABC souhaite étudier l'architecture logicielle cible par rapport au contexte général de la solution. Premièrement, il faut s'assurer que la solution proposée est parfaitement soutenue par les technologies envisagées. Deuxièmement, il faut regarder ce qui se passe dans des domaines connexes ou avec des problèmes équivalents dans le but d'en extraire des règles ou des solutions (complètes ou partielles) applicables à son contexte. Ces deux points servent de validation de la solution proposée.

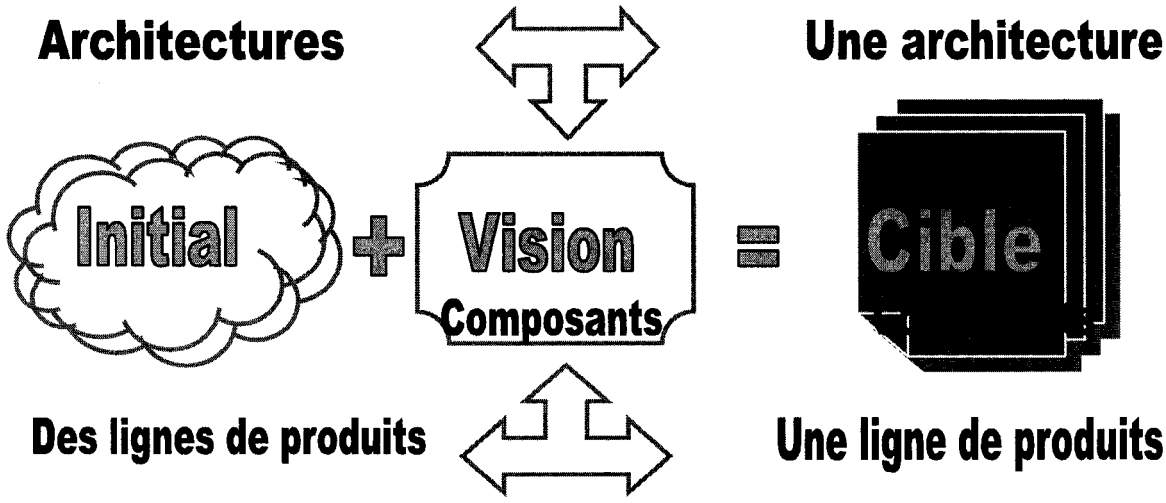

Figure 2 Vision pour l'architecture logicielle

La figure 2 résume la vision de l'entreprise au niveau de l'architecture logicielle. Ainsi, il existe plusieurs architectures (représentant plusieurs lignes de produits) que

l'entreprise ABC souhaite faire migrer vers une seule architecture pour une seule ligne de produits, qui doit inclure le support pour les composants.

En parallèle, l'entreprise a déjà commencé à réfléchir à sa vision très haut niveau de l'architecture logicielle. La figure 3 montre la vision basée sur les contraintes et les besoins de l'entreprise déjà identifiés.

Plusieurs points ont été identifiés pour la nouvelle architecture logicielle. Le premier point est que la partie applicative doit être indépendante de la plateforme (pour l'instant, le choix se porte sur Java[3] de Sun). L'entreprise ne souhaite pas se lier à un fournisseur de matériel unique dans le but de réduire les coûts d'achat et les frais de licences.

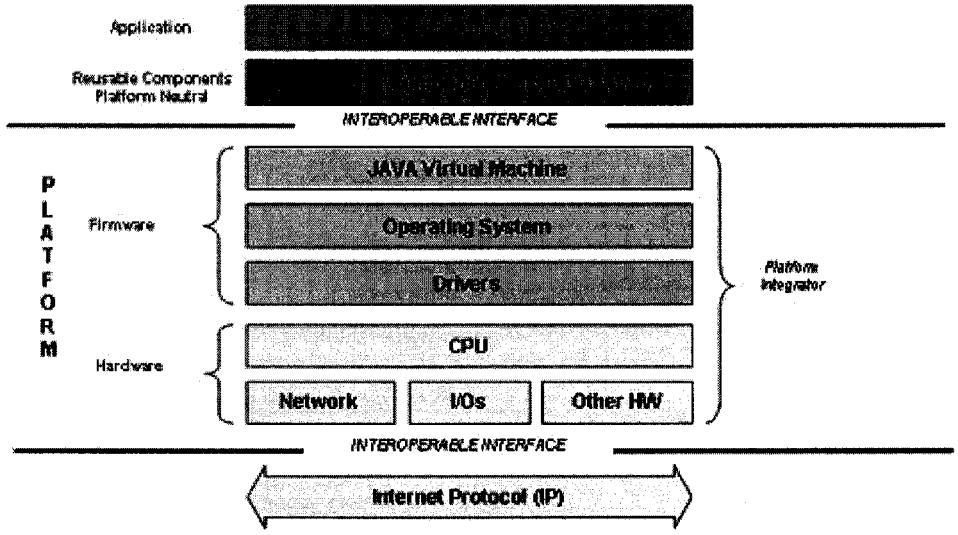

Figure 3 Proposition architecturale

Le deuxième point est la réutilisation (avec des composants) pour faciliter la sous-traitance de ces composants et surtout améliorer la réutilisation des composants disponibles entre les différents sites. Le dernier point est la ligne de produit qui n'est pas

[3] http://www.java.sun.com

visible dans la figure, mais qui impose une uniformisation de l'architecture et des solutions. Ce point va surtout imposer des règles à suivre par l'architecture logicielle pour permettre le support de plusieurs produits.

Toutefois, il est primordial d'utiliser la connaissance déjà existante dans les différentes divisions. Dans cette optique, il faut extraire la connaissance et les solutions existantes et les appliquer sur la vision proposée. La démarche doit inclure :

1. une extraction de la connaissance existante des lignes de produits dans les différentes divisions;
2. une application de cette connaissance sur la solution proposée, tout en ajoutant de nouveaux concepts.

Suite aux discussions avec ABC, l'aspect des systèmes patrimoniaux est plus interprété comme un cas particulier de l'aspect composant. L'entreprise souhaite réutiliser ce qui existe et, par conséquent, prévoir un mécanisme qui le favorise en cachant sous un composant les systèmes patrimoniaux par exemple.

1.3 Sujet de la recherche

Avant de faire une proposition d'étude, il convient de bien délimiter le champ d'investigation de cette étude. Le domaine dans lequel évolue ABC est très vaste et regroupe différentes divisions œuvrant dans différents domaines. En aucun cas, l'étude ne portera sur ces domaines. L'entreprise restera le maître d'œuvre et choisira les cas d'études concrets à utiliser pour la conception de l'architecture logicielle.

De plus, l'étude ne s'occupera pas de la partie organisationnelle liée à l'implantation des concepts de ligne de produits et de composants. En collaboration avec certains professeurs de l'ÉTS, l'entreprise travaille déjà sur tous ces aspects.

C'est pourquoi le domaine de recherche de cette étude est centré autour de l'étude de certaines caractéristiques logicielles dans le but de fournir une architecture logicielle répondant à ces critères très précis. Ces caractéristiques représentent des exigences non fonctionnelles influençant l'architecture logicielle. Par conséquent, l'étude se situe au niveau de la conception de l'architecture logicielle de la future ligne de produits.

Comme il a été mentionné dans la section précédente, deux aspects importants sont à considérer. D'une part, l'entreprise veut une seule ligne de produits pour toutes les divisions; d'autre part, elle veut pouvoir réutiliser les composants déjà existants. De plus, elle souhaite utiliser le processus unifié de Rational (Rational Unified Process[4] (RUP)). Donc, les propositions découlant de cette étude devront parfaitement s'intégrer dans ce contexte.

Avant de passer à la proposition, il convient d'expliquer plus en détail le concept de « caractéristiques architecturales ». Cette notion est une référence à des contraintes ou des besoins exprimés par le client qui ont un impact sur l'architecture logicielle, mais qui ne sont pas des besoins liés à de la fonctionnalité. Par exemple, la redondance (pour des systèmes qui doivent fonctionner 24h sur 24) a évidemment un impact sur l'architecture car le système peut avoir de la redondance matérielle et aussi logicielle. Toutefois, la redondance n'ajoute pas nécessairement de nouvelles fonctionnalités visibles par l'utilisateur. Ces concepts seront définis plus loin dans ce document (voir le chapitre 3).

La proposition initiale est donc décomposée en plusieurs étapes indépendantes mais qui forment une démarche globale de validation de la vision architecturale :

- offrir des gabarits pour l'étude de caractéristiques architecturales;

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

- étudier des caractéristiques architecturales sur certains projets existants (ABC se chargera de cette étape);
- étudier certaines caractéristiques présélectionnées dans le cadre de l'architecture proposée (ABC se chargera de cette étape);
- offrir des gabarits pour valider la solution architecturale dans son contexte technologique;
- étudier des solutions existantes dans des domaines connexes.

Le grand intérêt pour cette étude est d'arriver à concevoir une architecture logicielle respectant ses besoins et prenant en compte l'expertise existante. De plus, l'entreprise souhaite valider l'architecture logicielle par rapport à la technologie ainsi qu'aux solutions utilisées pour des problèmes équivalents. Ainsi, toutes les caractéristiques prises en compte auront été contrôlées et validées par des experts au sein de l'entreprise. L'entreprise souhaite mettre en place une méthode systématique d'étude pour diminuer le risque découlant d'un nouveau développement.

Le but de l'étude pour l'ÉTS est de fournir des méthodes et / ou des outils nécessaires pour l'étude de caractéristiques architecturales. Pour cela, il faudra améliorer le processus de développement RUP en ce qui concerne la définition d'une architecture logicielle par rapport aux caractéristiques architecturales recherchées.

Lors des dernières réunions, l'entreprise ABC a exprimé le souhait de recentrer l'étude sur l'extraction et la définition des caractéristiques architecturales. L'entreprise n'est pas prête pour l'instant à collecter l'information, du fait de changements majeurs en cours dans l'organisation de l'entreprise. Par conséquent, l'étude mettra l'accent sur cette partie uniquement. Un autre étudiant pourra finaliser la démarche globale lorsque l'entreprise ABC aura le temps et les ressources nécessaires pour faire les analyses des produits existants.

Maintenant que l'étude a bien été définie, nous allons présenter la recherche bibliographique pour trouver ce qui existe déjà et ce qu'il manque pour combler les besoins exprimés.

Cette recherche devra servir à définir toutes les notions importantes pour éviter des confusions entre intervenants, rechercher toutes les méthodes existantes traitant de l'architecture logicielle par rapport aux attributs de qualité et enfin, définir les concepts de lignes de produits et de composants par rapport à l'architecture logicielle.

CHAPITRE 2

RECHERCHES SUR LES DÉFINITIONS

2.1 Attributs de qualité

Nous avons décidé d'utiliser les termes « attributs de qualité » au lieu de « exigences non fonctionnelles » lors de la rédaction de ce mémoire pour se conformer au vocabulaire utilisé en architecture logicielle.

La norme ISO/IEC 14598-1 (ISO/IEC, 1998) donne la définition suivante pour un attribut : « *a measurable physical or abstract property of an entity* » et pour une qualité : « *the totality of characteristics of an entity that gear on its ability to satisfy stated and implied needs* ».

Nous allons donc adopter ces définitions pour la suite de la recherche.

2.2 Architecture

La norme IEEE 1471 (2000) traite spécifiquement de la description architecturale des systèmes à forte composante logicielle et par conséquent fournit une définition qui est « *The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution* ».

Toutefois, il existe d'autres références sur le sujet. En particulier, les ouvrages de Bass *et al.* (Bass, Clements, & Kazman, 1998, 2003) sur l'architecture logicielle offrent

beaucoup d'informations pertinentes. Ils ont décomposé leurs ouvrages en quatre parties suivant grossièrement le cycle de vie. La première partie traite de l'architecture envisagée et se concentre sur la technique Architecture Business Cycle (ABC) et les définitions. La seconde partie porte sur la conception de l'architecture en s'aidant des attributs de qualités et des tactiques, pour faciliter la transition des attributs de qualité vers une architecture. La troisième partie concerne l'analyse d'une architecture existante avec des techniques telles que ATAM (Architecture Tradeoff Analysis Method) et CBAM (Cost Benefit Analysis Method). Enfin, la dernière partie examine la question de la migration d'une architecture monolithique vers une architecture modulaire. Les deux ouvrages donnent des exemples réels très intéressants et proposent des techniques applicables au contexte de cette recherche. En effet, les discussions sur les qualités architecturales et les tactiques, plus la technique ATAM pour l'extraction de qualités architecturales d'une architecture existante, donnent de bonnes bases pour commencer la recherche dans le cadre de la problématique.

De plus, Bass *et al.* (2003) définissent l'architecture logicielle comme « *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the external visible properties of those elements, and their relationships among them* ».
Dans la première version du livre (Bass et al., 1998), le terme « composant » était utilisé plutôt que celui d'« élément ». En utilisant ce dernier, les auteurs veulent éviter toute confusion avec les techniques de développement logiciel basées sur les composants. Toutefois, dans le contexte de notre partenaire, l'utilisation du terme «composant» semble plus appropriée (tout en gardant la définition initiale), car l'entreprise souhaite mettre l'accent sur la réutilisation de composants.

Pour l'étude, nous allons considérer les deux définitions sans porter de jugement sur l'une et l'autre, car elles offrent une vision claire d'une architecture logicielle. Toutefois, il est possible de trouver d'autres définitions[5] toutes aussi acceptables.

2.3 Ligne de produits

Le concept de ligne de produits au niveau du développement logiciel est un concept relativement nouveau. Actuellement, plusieurs auteurs font des recherches dans ce domaine pour fournir des processus de support pour l'implémentation d'une ligne de produits dans une organisation.

Dans leur ouvrage, Clements et Northrop (2001) définissent « 29 skill areas in which an organization needs to be adept in order to achieve a software product line capability». De plus, ils proposent une définition d'une ligne de produits au niveau logiciel, qui est : « *a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* ». Enfin, ils donnent une définition d'une architecture de ligne de produits, qui est :

> « *A product line architecture is a software architecture that will satisfy the needs of the product line in general and the individual products in particular by explicitly admitting a set of variation points required to support the spectrum of products within the scope* ».

D'autres auteurs (Bass et al., 2003; Bosch, 2000) font ressortir deux principes de base pour une ligne de produits et proposent des processus pour soutenir ces deux principes. En effet, ces auteurs estiment que ces deux principes sont à la base de la définition d'une

[5] http://www.sei.cmu.edu/architecture/definitions.html

architecture logicielle dans le contexte d'une ligne de produits. Le premier principe est « la mise en commun » qui signifie de définir ce qui est commun entre les différents produits d'une ligne de produits et de le partager (c'est-à-dire de prévoir les mécanismes pour supporter les mises en commun). Le second principe est « la variabilité » qui signifie de définir ce qui est différent entre les produits formant la ligne de produits et de prévoir les mécanismes pour supporter ces différences. Ces deux principes permettent d'extraire des caractéristiques pour supporter les mises en commun et les différences au niveau de l'architecture. Par exemple pour les différences, Clements et Northrop (2001, page 64) soulignent que :

> « *In a conventional architecture, the mechanism for achieving different instances almost always comes down to modifying the code. But in a software product line, support for variation can take many forms. Variation can be accomplished by introducing build-time parameters to a component, subsystem, or a collection of subsystems, whereby a product is configured by setting a collection of values* ».

La définition fournie par Clements et Northrop est parfaite pour l'étude car elle montre bien la différence avec une architecture logicielle pour un seul produit et les points de variations. Donc, nous allons faire référence à cette définition tout au long de cette étude.

2.4 Composants

Cette section se concentre sur des définitions pour les composants seuls ou dans le contexte d'une ligne de produits. En effet, une architecture de ligne de produits doit supporter des mécanismes pour la mise en commun (voir la section précédente) et le concept de composants offre une solution concrète à cette problématique.

Dans son ouvrage, Bosch (2000) propose la définition suivante : « A software component is a unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes ».

Il ajoute (Bosch, 2000, page 220) que :

> « *An interface defines a contract between a component requiring certain functionality and a component providing that functionality. The interface represents a first-class specification of the functionality that should be accessible through it. The interface specification is, ideally, independent of the component or components implementing that interface* ».

D'autre part, Szyperski *et al.* (2002, page 41) suggère que : « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties* ». Il ajoute (2002, page 50) que : « *An interface is a set of named operations that can be invoked by clients. Each operation's semantics is specified, and this specification plays a dual role as it serves both providers implementing the interface and clients using the interface* ».

Les points communs entre ces deux définitions sont qu'un composant est une unité de composition avec des interfaces fournies aux utilisateurs de ce composant et des interfaces requises au bon fonctionnement de celui-ci. Toutefois, il y a des différences entre les deux définitions. Le premier point est que seul Bosch mentionne les attributs de qualité. Un attribut de qualité souligne l'importance des caractéristiques décrivant une ou plusieurs contraintes du composant. Or, le respect ou non par le composant de certaines caractéristiques désirées pour une architecture logicielle peut limiter les

possibilités de réutilisation. Le second point est que Bosch mentionne le besoin pour un composant d'avoir des interfaces pour sa configuration. Ces interfaces sont nécessaires pour gérer les points de variations (ceci fait référence aux besoins de gérer les variabilités comme expliqué dans la section précédente), ce qui est très important pour améliorer les possibilités de réutilisation du composant.

Ainsi en nous basant sur les deux définitions précédentes, la définition que nous adoptons pour un composant est :

> *A software component is a unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes. A software component can be deployed independently and is subject to composition by third parties.*

Voici une traduction possible de cette définition :

> Un composant logiciel est une unité composable qui spécifie explicitement des interfaces à fournir, des interfaces qui sont requises et d'autres dédiées à la configuration et des attributs de qualité. Un composant logiciel peut être déployé et composé avec d'autres.

De plus, la définition pour une interface est :

> *An interface defines a contract between a component requiring certain functionality and a component providing that functionality, and is ideally independent of the component implementing it. In other words, an interface represents a specification of the functionality that should be accessible.*

Voici une traduction possible de cette définition :

> Une interface définit un contrat entre un composant qui requiert une certaine fonctionnalité et un autre composant fournissant cette fonctionnalité. De plus, cette interface est idéalement indépendante du composant qui l'implémente. En d'autres mots, une interface représente une spécification de la fonctionnalité qui doit être accessible.

Voici une traduction possible pour la définition des interfaces nécessaires à un composant :

> Les trois types d'interface sont :
>
> - une interface qui fournit de la fonctionnalité visible à l'extérieur du composant;
> - une interface qui requiert de la fonctionnalité provenant d'un autre composant;
> - une interface de configuration est une interface pour configurer les points de variations implémentés par le composant. Notons que ces points de variations peuvent aussi altérer la fonctionnalité du composant.

Maintenant que nous avons fourni des définitions possibles pour les composants et pour les interfaces, il faut parler un peu de la réutilisation qui est l'idée majeure derrière le choix de l'introduction de composants.

En effet, Bosch discute (2000, pp 214-237) des composants dans un environnement de ligne de produits et indique que l'idée communément admise que la composition de composants ressemble à un assemblage de « Legos » ne reflète pas complètement la réalité. Pour une ligne de produits, la composition est aussi basée sur des adaptations et des variations. L'adaptation consiste à adapter le composant à son environnement et la variation consiste à prévoir des points de variations pour des changements futurs. De

plus, même la réutilisation doit être attentivement étudiée. Bosch signale que la réutilisation d'un composant peut se décomposer en trois niveaux :

1. « *system version* » qui est l'utilisation d'un composant au travers de plusieurs versions d'un système;
2. « *software product line* » qui est l'utilisation d'un composant au travers de plusieurs produits d'une ligne de produits. Le composant doit pouvoir accepter des changements d'exigences et des différences d'exigences entre les différents produits de la ligne de produits;
3. « *Third-Party component* » qui est l'utilisation d'un composant au travers des différentes versions d'un produit, d'une ligne de produits et dans différentes organisations.

Ainsi Bosch souligne le besoin de prévoir la réutilisation dans une ligne de produits. Il faut noter que Clements et Northrop (2001) soulignent la nécessité de mettre en place un répertoire de composants (« *core assets* ») dans le but de centraliser les composants et ainsi d'en faciliter la réutilisation entre les différents produits d'une ligne de produits.

Comme nous l'avons mentionné, la réutilisation nécessite plus d'étude pour être sûr de bien la prévoir et de correctement l'utiliser.

Cette section fournit des définitions utiles pour les composants et aussi des remarques importantes concernant notre contexte d'utilisation.

2.5 Systèmes patrimoniaux

Dans notre contexte, un système patrimonial (« *Legacy* ») est vu comme une forme particulière d'un composant. En effet, l'entreprise voudrait « encapsuler » les systèmes patrimoniaux pour les isoler et ainsi éviter de les modifier ou de les refaire. En effet, la durée de vie d'un produit logiciel dans le domaine ferroviaire peut être de 15 à 30 ans.

Ainsi, les problèmes liés aux systèmes patrimoniaux sont donc très présents. Ce point est très important car la vision n'est pas de tout recommencer mais plutôt de refaire ce qui est strictement nécessaire sans s'occuper de ce qui sera gardé tel quel.

Cette section conclut ce chapitre concernant les définitions utiles pour cette recherche. Il faut maintenant rechercher dans la littérature les méthodes concernant les attributs de qualité et l'architecture logicielle.

CHAPITRE 3

RECHERCHES SUR LES ATTRIBUTS DE QUALITÉ

3.1 Introduction

Il s'agit maintenant de faire une recherche sur les attributs de qualité dans la littérature. En effet, il existe déjà des normes, des méthodes et des modèles dédiés aux attributs de qualité qui s'appliquent à la conception d'architectures logicielles.

3.2 Normes

Les deux séries de normes ISO/IEC 14958 (ISO/IEC, 1998) et ISO/IEC 9126 (ISO/IEC, 1999) se concentrent sur les attributs de qualité pour un produit logiciel. La série ISO/IEC 14598 fournit des processus pour étudier les attributs de qualité lors du développement (ISO/IEC 14598-3), de l'acquisition (ISO/IEC 14598-4) et de l'évaluation indépendante (ISO/IEC 14598-5) d'un produit logiciel. La série ISO/IEC 9126 fournit les définitions des caractéristiques et des sous-caractéristiques de qualité (ISO/IEC 9126-1) avec des métriques externes (ISO/IEC 9126-2) et des métriques internes (ISO/IEC 9126-3) pour valider les caractéristiques.

De plus, la norme 9126-1 (ISO/IEC, 1999) fournit un modèle de qualité qui résume les dernières connaissances sur les spécifications des qualités d'un produit logiciel.

La norme ISO/IEC 14598-1 (1998) donne également une décomposition des attributs de qualité en fonction de leur moment de spécification (voir figure 4).

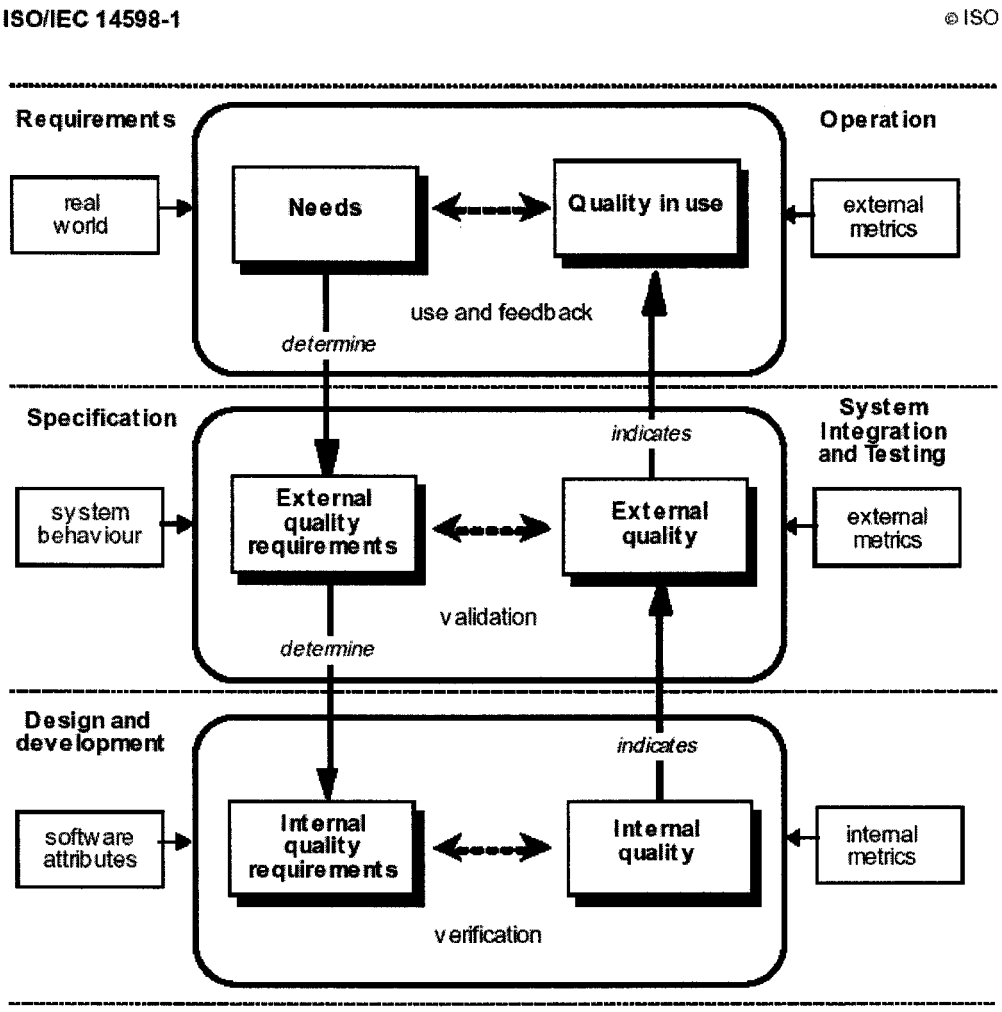

Figure 4 ISO/IEC 9126-1 Qualité dans un cycle de vie logiciel

Cette vision est décomposée en trois niveaux. Le premier niveau représente des attributs de qualité visibles à l'extérieur du produit logiciel lorsque le système est utilisé. Ces attributs sont reliés à des besoins d'usagers concernant les qualités d'un produit logiciel. Le deuxième niveau (sur lequel repose le premier niveau) représente des attributs de qualité dérivant des besoins des usagers. Ces attributs servent à valider le produit logiciel durant les différentes étapes de développement. Le troisième niveau (sur lequel

repose le deuxième niveau) spécifie des propriétés pour des produits intermédiaires (des modèles ou du code entre autres).

Pour les attributs de qualité visibles durant l'utilisation, le modèle proposé par ISO/IEC 9126-1 définit quatre caractéristiques de base avec des attributs mesurables (voir figure 5).

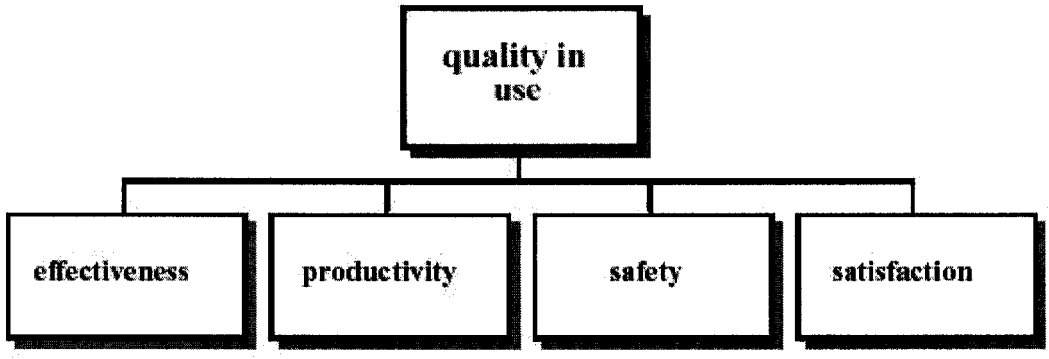

Figure 5 ISO/IEC 9126-1 Qualités durant l'utilisation

Pour les attributs de qualité externes et internes, le modèle proposé par ISO/IEC 9126-1 définit six caractéristiques de base qui sont détaillées par des sous-caractéristiques avec des attributs mesurables (voir figure 6).

La série de normes ISO/IEC 9126 fournit des informations importantes sur les attributs de qualité appliqués à un produit logiciel. Toutes ces informations sont également applicables à une architecture logicielle. En effet, plusieurs auteurs (Bass et al., 1998; Firesmith, 2003b; Losavio & al., 2003) ont déjà trouvé des équivalences entre un produit logiciel et une architecture dans le cadre d'une étude des attributs de qualité.

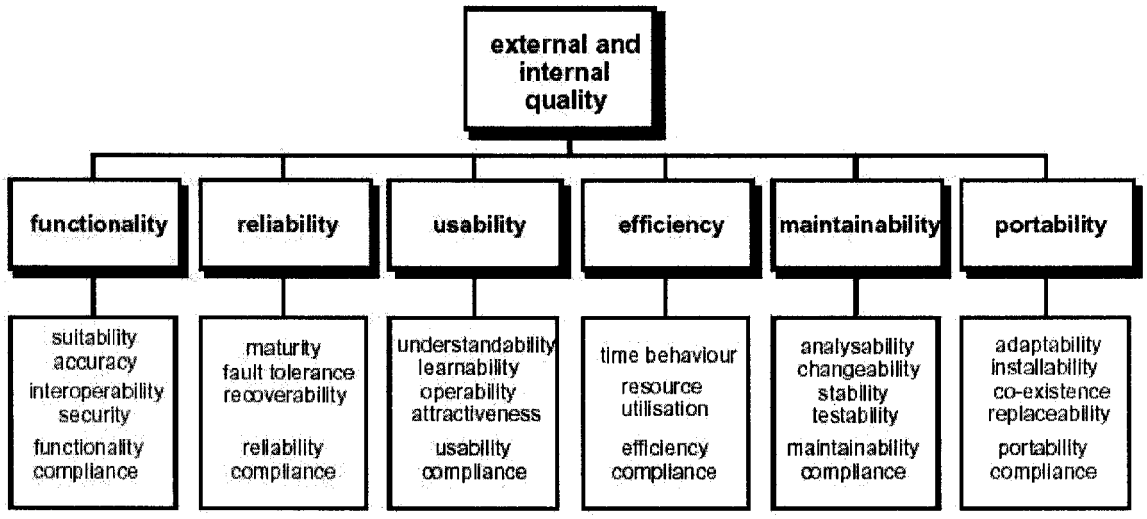

Figure 6 ISO/IEC 9126-1 Qualités internes et externes

Cela constitue une base solide pour l'étude des attributs de qualité car les normes ISO/IEC 9126 donnent un vocabulaire précis, des modèles et des métriques pour valider les attributs de qualité. Ainsi, les métriques, pour valider les attributs de qualité fournies par les normes, sont applicables aussi à une architecture logicielle. Or, la validation est un aspect critique de toute démarche car elle permet de confirmer ou non si un but a été atteint. Ce point doit être présent dans la proposition à développer.

En conclusion, ces normes fournissent des informations très intéressantes et réutilisables dans notre contexte. En effet, toutes ces caractéristiques et sous-caractéristiques et leurs métriques sont réutilisables dans le contexte d'une architecture logicielle.

3.3 Modèles

L'objectif de cette section est d'étudier les modèles dédiés aux attributs de qualité dans le but de les réutiliser complètement ou partiellement dans le cadre de cette étude. Pour

couvrir tous les besoins, il faudra aussi prendre en compte les concepts de lignes de produits et de composants.

Dans le cadre des lignes de produits, Schmid (2001) définit un modèle global pour aider à définir un modèle pour les attributs de qualité. Celui-ci met de l'avant un modèle de qualité s'intégrant dans l'approche PuLSE[6] mais sans y être restreint. La vision de Schmid est de proposer un métamodèle pour permettre aux utilisateurs de définir des modèles de qualité spécifiques à chacun de leurs besoins.

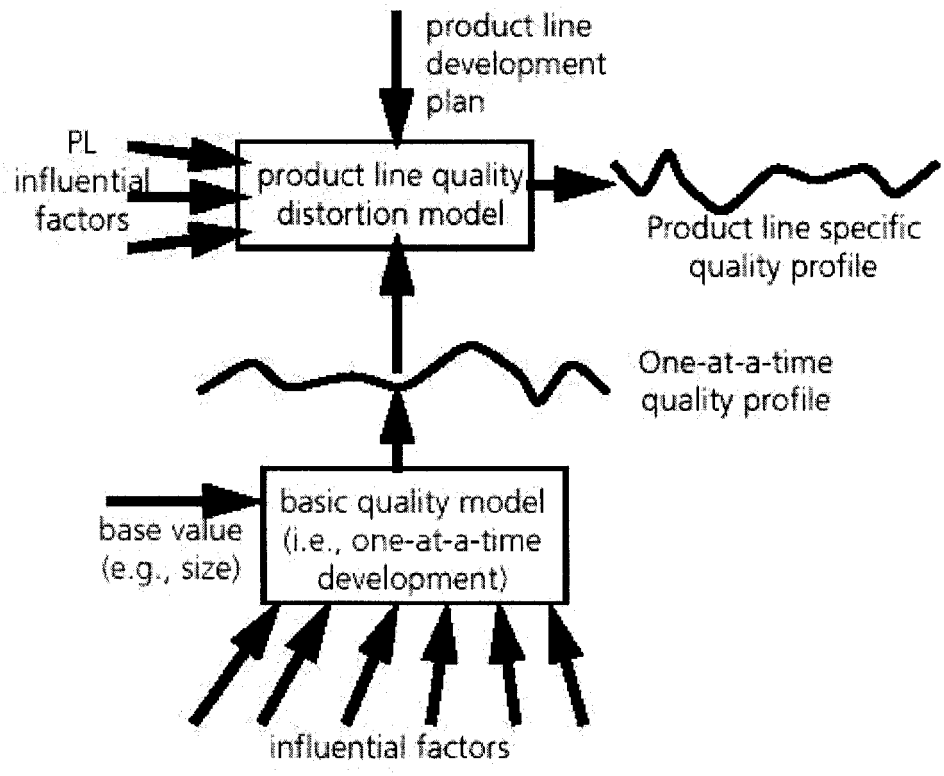

Figure 7 Métamodèle dans un contexte de ligne de produits

[6] PuLSE is a registered trademark of Fraunhofer IESE, http://www.iese.fraunhofer.de

La figure 7 montre cette proposition. La partie du bas de la figure souligne la récupération des modèles de qualité provenant de normes, tandis que la partie du haut souligne l'adaptation nécessaire des modèles de base au contexte de la ligne de produits.

Schmid (2001, pp. 13-17) fournit aussi une approche pour étudier l'aspect d'adaptation des attributs de qualité au contexte de la ligne de produits :

1. identifier les facteurs de qualité : des experts étudient et trient les attributs de qualité;
2. développer un modèle de cause à effet : des experts étudient les relations entre les attributs car les interactions peuvent changer l'impact des facteurs de qualité sur le résultat final;
3. développer un questionnaire : pour chaque facteur de qualité, il faut développer un questionnaire pour bien caractériser un facteur indépendamment des autres;
4. quantifier les relations : les relations doivent être quantifiées (c'est-à-dire définir le minimum, le maximum et la valeur souhaitable);
5. rendre opérationnel le modèle : il faut rendre les quantifications utilisables pour pouvoir prévoir le résultat final.

Dans le cas de l'architecture logicielle, Losavio *et al.* (2003) proposent une modification du modèle de qualité standard offert par ISO/IEC 9126 (1999), pour mieux l'adapter au contexte de l'architecture logicielle. En effet, les auteurs ont adapté toutes les caractéristiques et les sous-caractéristiques ce qui permet de réutiliser les métriques fournies par la norme. Toutefois, la méthode proposée se base sur une étude *a posteriori* des attributs de qualité.

De plus, Firesmith (2003b) fournit un modèle de qualité et une méthode simple pour définir et valider les attributs de qualité. Il propose un modèle de qualité qui met de l'avant trois niveaux différents. Le premier niveau utilise les modèles et les attributs provenant des normes ISO/IEC 9126 (1999). Le second niveau définit un modèle pour

tous les projets spécifiques à l'organisation. Enfin, le troisième niveau concerne le modèle spécifique à un seul projet. Firesmith fournit une méthode simple pour s'assurer de la bonne utilisation de ces modèles.

Ce processus comporte six étapes :

1. sélectionner le modèle de qualité;
2. sélectionner les facteurs de qualité significatifs (c'est-à-dire les attributs de qualité);
3. produire les critères de qualité (c'est-à-dire une description courte et pertinente);
4. sélectionner les mesures applicables aux qualités;
5. sélectionner les exigences sur les qualités (c'est-à-dire spécifier un degré de conformité pour chaque attribut de qualité);
6. valider la qualité avec tous les intervenants impliqués.

Il est à noter qu'aucun modèle de qualité dédié aux composants n'a été trouvé durant les recherches. Tous les modèles trouvés, par ailleurs, sont parfaitement applicables au contexte des composants. Seule une étude faite en milieu universitaire (Torchiano, Jaccheri, Sorensen, & Wang, 2002) est disponible au sujet des attributs de qualité applicables aux composants. De plus, cette étude fournit une correspondance entre les attributs de qualité trouvés et ceux de la norme ISOIEC 9126.

Ce survol des études sur les attributs de qualité montre que la définition et l'adaptation des attributs de qualité pour l'architecture logicielle ou pour le contexte de la ligne de produits sont bien documentées dans la littérature. Il existe cependant peu de références pour les composants.

3.4 Méthodes

Cette section fournit une étude de différentes méthodes disponibles pour la conception d'une architecture logicielle à partir des attributs de qualité et pour l'évaluation des attributs de qualité sur une architecture logicielle existante.
Dans notre contexte, toutes ces méthodes de conception sont à envisager car la proposition doit considérer l'extraction des attributs de qualité sur des architectures existantes et la conception d'une nouvelle architecture pour la future ligne de produits.

La méthode « *Architecture Tradeoff Analysis Method* » (ATAM) (Len Bass, 2003; Paul Clements, 2002) est une méthode d'évaluation des qualités architecturales d'une architecture logicielle. ATAM consiste à analyser les qualités architecturales d'une proposition d'architecture logicielle et à fournir des indications sur les interactions et les problèmes possibles entre les qualités architecturales supportées.

ATAM est décomposée en trois phases :

1. une activité initiale d'évaluation basée sur des réunions entre l'équipe d'évaluation et les personnes clés du projet pour collecter l'information et commencer l'analyse (petit groupe);
2. une analyse élargie avec les intervenants au niveau de l'architecture logicielle (groupe de la phase 1 plus tous les autres intervenants);
3. une activité de conclusion pour produire le rapport final et pour entamer des discussions sur les améliorations possibles.

De plus, ATAM raffine les phases 1 et 2 en neuf étapes :

1. début de la phase 1 : présenter la méthode ATAM;
2. présenter les catalyseurs opérationnels (« business drivers »);

3. présenter l'architecture : l'architecte en chef présente l'architecture logicielle en couvrant les contraintes et les styles utilisés pour répondre aux exigences;
4. identifier les approches architecturales;
5. produire l'arbre d'utilité pour les attributs de qualité : cela consiste à formaliser une décomposition des attributs de qualité en attributs plus restreints et plus précis jusqu'à un scénario présentant le comportement espéré et il faut aussi prioriser les scénarios;
6. analyser les approches architecturales : l'architecte doit expliquer comment chaque scénario est supporté par l'architecture logicielle;
7. début de la phase 2 : réfléchir sur les scénarios et les prioriser ce qui consiste à réfléchir avec tous les intervenants sur les scénarios et leur priorité pour comparer les résultats avec l'arbre d'utilité. Cette nouvelle mise en priorité permet d'extraire les scénarios les plus importants à transmettre à l'étape suivante;
8. analyser les approches architecturales : l'architecte explique les approches utilisées pour supporter les scénarios issus de l'étape 7;
9. présenter les résultats : cette présentation doit contenir des indications sur le support des attributs de qualité par l'architecture logicielle, les risques, les points sensibles et les compromis.

ATAM introduit des artéfacts très intéressants tels que :

- une présentation de l'architecture qui est un document pour présenter l'architecture dans son ensemble;
- un arbre des attributs de qualité qui est une méthode pour identifier, prioriser et détailler les attributs de qualité;
- un gabarit pour l'analyse des scénarios et enfin un rapport présentant les résultats.

Il est important de noter que chaque feuille de l'arbre des attributs est en fait un scénario qui doit capturer le comportement du produit logiciel pour un attribut de qualité spécifique.

La méthode « *Attribute-Driven Design* » (ADD) (Bass et al., 2003) est une méthode de conception d'une architecture. La conception de l'architecture logicielle se fait avec une méthode de décomposition basée entre autres sur les attributs de qualité. La notion de décomposition du problème est très importante pour cette méthode.

Les étapes sont :

1. choisir le nouveau module à décomposer (pour la première itération, il s'agit du système au complet);
2. spécialiser le module en :
    a. choisissant les scénarios critiques qui influencent l'architecture logicielle parmi l'ensemble des cas d'utilisation représentant les exigences fonctionnelles et des scénarios d'attributs de qualité;
    b. choisissant les patrons architecturaux ou en créant une solution répondant aux besoins des scénarios sélectionnés;
    c. instanciant les modules et en allouant la fonctionnalité aux différents modules;
    d. définissant les interfaces des modules enfants;
    e. vérifiant et raffinant les scénarios pour les allouer aux sous-modules.
3. répéter les étapes précédentes si nécessaire.

De plus, ADD mentionne l'utilisation de l'arbre d'utilité de l'ATAM pour aider à correctement exprimer les scénarios nécessaires pour ADD. Cette méthode met au cœur de ses activités la décomposition du problème et la conformité aux attributs de qualité. Ces deux points sont fondamentaux dans la vision de cette étude.

Bosch (2000) propose aussi une méthode pour la conception d'architectures logicielles qui considère les attributs de qualité en même temps que les exigences fonctionnelles. En effet, le cœur de la méthode QASAR (*basic*) (Bosch, 2000, page 24) est décomposé en cinq étapes qui sont :

1. sélectionner certaines exigences;
2. concevoir ou améliorer une architecture logicielle avec les nouvelles exigences;
3. estimer le support des attributs de qualité par l'architecture logicielle;
4. si nécessaire, modifier l'architecture logicielle pour mieux répondre aux attributs de qualité;
5. recommencer avec de nouvelles exigences.

Les attributs de qualité sont définis avec des profils qui sont des ensembles de scénarios. Un ou plusieurs scénarios servent à modéliser le comportement d'un attribut de qualité. Cependant, Bosch met en garde contre la profusion de scénarios dans le but de modéliser tous les cas possibles. Bosch a prévu deux cas principaux qui sont : 1) le profil complet pour des projets relativement petits où il est facile de considérer tous les cas possibles; 2) le profil sélectionné pour extraire seulement les scénarios les plus représentatifs en évitant ainsi la perte de contrôle du processus à cause du grand nombre de scénarios à considérer. Ce dernier point est également souligné dans l'ATAM.

La première partie de cette section se concentre sur des méthodes utilisables pour tous les types d'architectures logicielles. Toutefois, notre partenaire souhaite explorer plus particulièrement les aspects de ligne de produits et de composants dans la nouvelle architecture logicielle. Donc, la deuxième partie va présenter des méthodes considérant l'aspect ligne de produits.

La méthode COMPARE (Briand, Carrière, Kazman, & Wüst, 1998) fournit « a comprehensive, operational framework to evaluate and compare architectures ». Cette méthode est basée sur SAAM (Bass et al., 1998) qui est l'ancêtre de ATAM.

COMPARE fournit des étapes pour l'évaluation des attributs de qualité. Les étapes sont :

1. **obtenir les buts** pour fournir un scénario d'élaboration suivant les guides de SAAM;
2. **représenter l'architecture** pour donner une représentation de l'architecture;
3. **estimer les impacts qualitatifs** pour fournir un ensemble de règles et de patrons afin de décrire les manières bénéfiques ou dangereuses de structurer le système;
4. **décomposer les attributs de qualité externes** pour décomposer les attributs de qualité vers des attributs de qualité mesurables;
5. **appliquer des instruments de mesure** pour adapter une mesure dans le but de la rendre utilisable par un intervenant (avec des questionnaires ou des outils d'analyse);
6. **estimer les impacts quantitatifs** pour fournir un estimé de la sensibilité d'une architecture logicielle envers un scénario ;
7. **appliquer des modèles de décisions** pour décider du scénario à considérer basé sur les priorités.

La figure 8 montre les activités et les artéfacts de la méthode COMPARE.

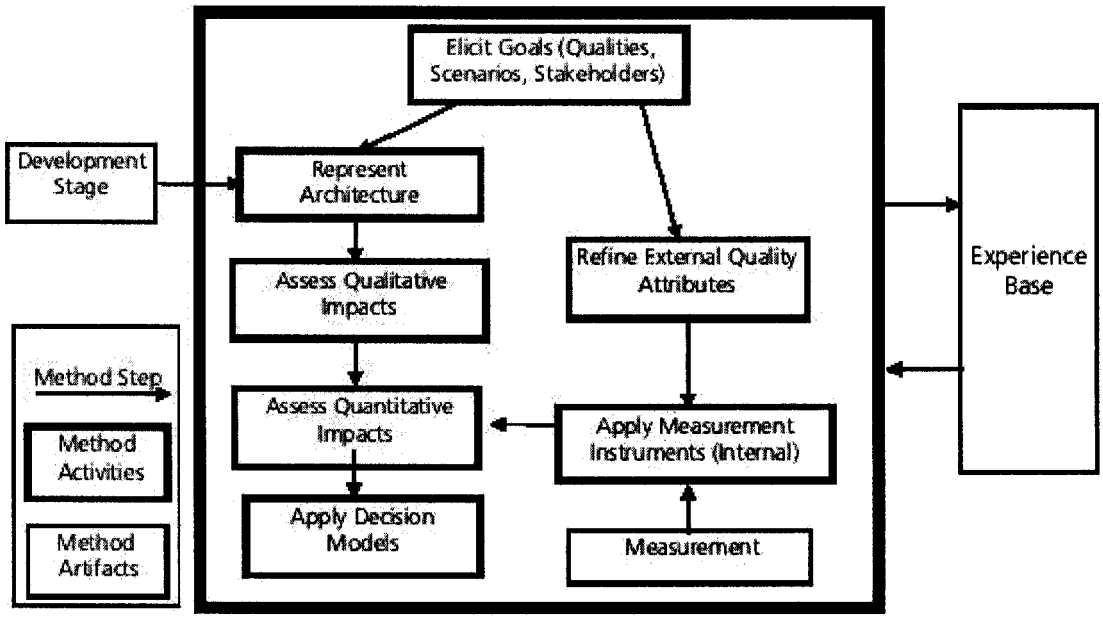

Figure 8 Présentation de COMPARE

COMPARE ajoute un aspect très important peu mentionné jusqu'à présent, il s'agit de mesurer les scénarios après leur implantation dans l'architecture. En effet, il est difficile d'affirmer qu'une architecture logicielle supporte un scénario précis, s'il n'existe aucun moyen de le vérifier impartialement.

Anastasopoulos *et al.* (2000) ont défini PuLSE-DSSA comme un processus modifiable et itératif qui intègre la conception et l'évaluation d'une architecture pour une ligne de produits. Ce processus est une partie d'un processus plus global PuLSE[7]. L'idée principale derrière ce processus est : « to develop a reference architecture[8] incrementally by applying generic scenarios in decreasing order of architectural significance and to integrate evaluation into architecture creation ».

[7] PuLSE is a registered trademark of Fraunhofer IESE, http://www.iese.fraunhofer.de
[8] Pour faciliter la compréhension du processus PuLSE-DSSA, il faut préciser que l'architecture de référence est l'architecture logicielle en cours de développement.

Les étapes sont :

1. créer les scénarios pour capturer le comportement des cas d'utilisation les plus importants;
2. sélectionner les scénarios et prévoir la prochaine itération;
3. définir des critères d'évaluation pour permettre l'évaluation de l'architecture à la fin de l'itération courante : l'auteur souligne la possibilité de l'utilisation des métriques telles que les mesures de la cohésion ou du couplage comme méthode d'évaluation;
4. appliquer les scénarios pour créer ou mettre à jour l'architecture;
5. évaluer l'architecture avec les critères d'évaluation : l'auteur souligne que l'évaluation doit tenir compte de l'aspect ligne de produits de l'architecture mais aussi, d'une instance pour un produit spécifique de l'architecture;
6. s'il y a un problème, analyser le problème et retourner à l'étape 2, sinon passer à la prochaine itération.

La figure 9 montre le processus proposé par PuLSE-DSSA.

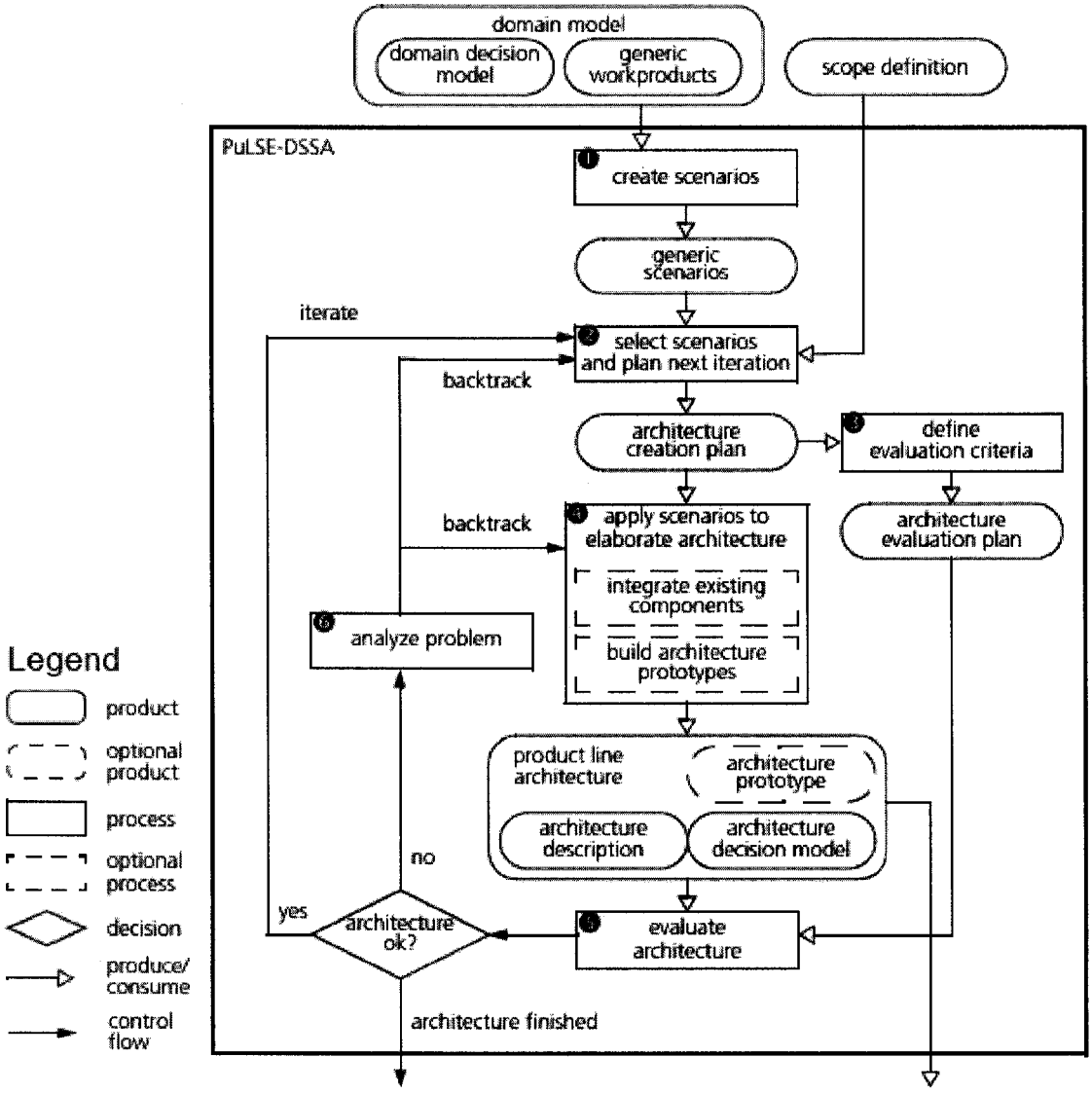

Figure 9 Présentation de PuLSE-DSSA

Ce processus insiste sur l'utilisation de scénarios à la base de chaque itération. Les scénarios les plus importants sont sélectionnés en premier en se basant sur la perspective d'une ligne de produit. En effet, un scénario très important pour un produit n'est peut-

être pas aussi important pour la ligne de produits et vice versa. Un autre point intéressant est l'intégration d'un mécanisme de validation dans le processus pour continuellement valider l'architecture. Les auteurs proposent d'utiliser certaines métriques (par rapport au couplage et à la cohésion) pour le contrôle. Un dernier point est que les auteurs mettent de l'avant certains attributs de qualité très importants dans le contexte d'une ligne de produits, tels que la maintenabilité, la facilité de compréhension et la réutilisation.

De plus, les auteurs Nord et Soni (2003) ont apporté une contribution supplémentaire en donnant une suite d'étapes pour passer d'un attribut de qualité à une partie de la conception. Les étapes sont :

> « *precise specification of quality attribute requirements, enumeration of fundamental design approaches to achieve various quality attributes, a linkage between the specification of the requirements, and the appropriate design approaches that yields a design fragment focused on achieving the requirement, and a method for composing the design fragments into an actual design* ».

Suite à l'étude de ces méthodes, plusieurs points de convergence sont très clairs et certaines divergences apparaissent. Les principaux points communs sont liés à la nécessité de décomposer les attributs des qualités provenant des processus en amont. En effet, les descriptions reçues ne sont habituellement pas assez précises et détaillées pour être utilisables telles quelles pour la conception d'une architecture logicielle (exemple, le système doit être maintenable). De plus, les méthodes utilisent toutes les scénarios pour capturer un comportement précis. Toutefois, ATAM et la méthode de Bosch soulignent l'importance de contrôler le nombre de scénarios pour éviter une prolifération qui entraînerait une perte de contrôle du projet. Les principales différences sont plus dans la manière de représenter les scénarios et les attributs de qualité. Seule la méthode ATAM

propose une représentation simple et explicite de la décomposition (d'ailleurs reprise par ADD) à l'aide de l'arbre d'utilité. La méthode de Bosch, quand à elle, offre un moyen simple pour limiter le nombre de scénarios en proposant les concepts de profils et de catégories. Enfin, la méthode PuLSE-DSSA mentionne la nécessité d'évaluer continuellement l'architecture durant le processus.

Dans toutes les recherches effectuées sur les composants par rapport aux attributs de qualités, aucune méthode n'a été trouvée concernant la conception ou l'utilisation de composants dans une architecture logicielle en se basant spécifiquement sur les attributs de qualité.

En conclusion, les méthodes mentionnées ci-dessus offrent toutes les éléments nécessaires pour notre contexte. Toutefois, aucune ne donne de méthode systématique pour valider la qualité de la décomposition et expliquer en détail le moyen de passer d'une liste de scénarios vers une architecture logicielle.

3.5 RUP

Notre partenaire a un intérêt pour la méthodologie Rational Unified Process (RUP) comme norme au travers des différentes divisions. Il est donc important de s'assurer que notre proposition va correctement s'intégrer dans ce contexte.

Le processus RUP est basé sur certains aspects centraux qui sont :

- le processus doit être incrémental et itératif ce qui signifie qu'il est basé sur des boucles successives où chaque boucle rajoute de la fonctionnalité et / ou des améliorations;
- la description de la fonctionnalité est basée sur des cas d'utilisation.

De plus, le modèle de développement prôné est décomposé en quatre phases (voir la figure 10). Les quatre phases (Jacobson, Booch, & Rumbaugh, 1999) sont :

1. l'**exploration** qui consiste à explorer les buts du produit logiciel basé sur les besoins des intervenants et à définir la portée du produit logiciel;
2. l'**élaboration** qui consiste à définir les exigences et l'architecture du produit logiciel;
3. la **construction** qui consiste à implémenter le produit logiciel, dans le but d'avoir des livrables opérationnels;
4. la **transition** qui consiste à installer le produit logiciel dans son environnement d'opération.

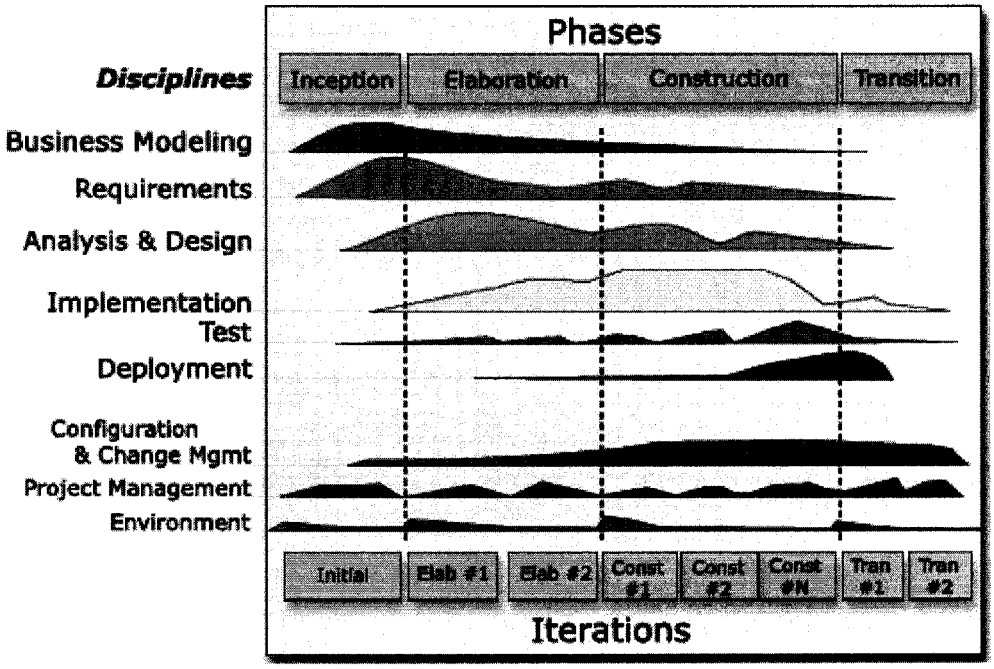

Figure 10 Phases dans RUP

La figure 11 montre les relations entre les principaux artéfacts de RUP. En particulier, la figure fait ressortir le lien entre le document SRS (Software Requirement Specifications)

et le document SAD (Software Architecture Document). Le SAD se base sur l'information du SRS pour établir l'architecture logicielle.

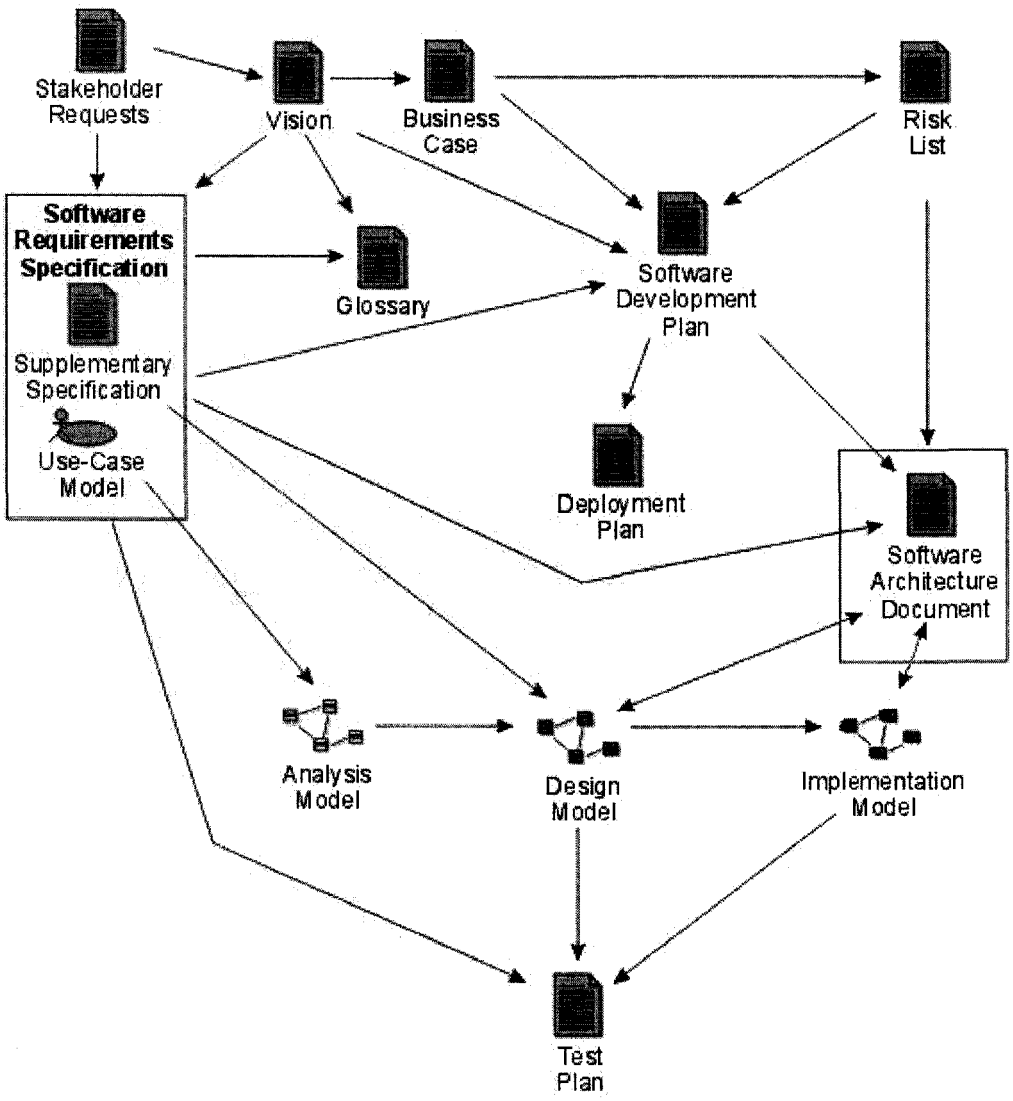

Figure 11 Artéfacts dans RUP

À titre de rappel, un document SRS[9] est « *The Software Requirements Specification (SRS) captures the complete software requirements for the system, or a portion of the system. When using use-case modeling, this artefact consists of a package containing use cases of the use-case model ...* ». De plus, un document SAD[10] est « *The Software Architecture Document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system* ».

Après cette revue très rapide, il semble clair que l'étude des exigences non fonctionnelles se situe dans le document SRS. Toutefois, le gabarit de ce document mentionne uniquement quatre attributs de qualité qui sont la facilité d'utilisation, la fiabilité, la performance et la maintenabilité, en ne fournissant aucune description sur la manière de les définir. Cela montre donc que les méthodes étudiées dans la section 3.4, vont parfaitement bien s'intégrer dans RUP.
De plus, la vision « RUP » considère toujours les exigences non fonctionnelles en second lieu lors de la conception de l'architecture logicielle ce qui est contraire aux recommandations de la communauté. En effet, la communauté converge vers le fait qu'il faut considérer les attributs de qualité très tôt lors de la conception de l'architecture logicielle et surtout en même temps que l'analyse des exigences fonctionnelles.

Par conséquent, cette courte description de RUP fait ressortir le besoin de définir une manière d'étudier les attributs de qualité avant de concevoir une architecture logicielle.

Il faut maintenant faire une synthèse de ce qui a été découvert dans le but d'en montrer les forces et les faiblesses.

---

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

CHAPITRE 4

SYNTHÈSE DES RECHERCHES BIBLIOGRAPHIQUES

4.1 Rappel des définitions

Voici un rappel des différentes définitions qui vont être utilisées tout au long de cette étude.

La définitions choisie pour un attribut est celle fournie par la norme 14598-1 (ISO/IEC, 1998) qui est « *a measurable physical or abstract property of an entity* ».

La définition choisie pour l'architecture logicielle est celle de Bass *et al.* (2003), qui est : « *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the external visible properties of those elements, and their relationships among them* ».

La définition choisie pour l'architecture d'une ligne de produits est celle de Clements et Northrop (2001, page 33), qui est :

> « *A product line architecture is a software architecture that will satisfy the needs of the product line in general and the individual products in particular by explicitly admitting a set of variation points required to support the spectrum of products within the scope* ».

La définition que nous proposons pour un composant est :

> Un composant logiciel est une unité composable qui spécifie explicitement des interfaces à fournir, des interfaces qui sont requises et d'autres dédiées à la configuration et des attributs de qualité. Un composant logiciel peut être déployé et composé avec d'autres.

De plus, la définition pour une interface est :

> Une interface définit un contrat entre un composant qui requière une certaine fonctionnalité et un autre composant fournissant cette fonctionnalité, et est idéalement indépendant du composant qu'il l'implémente. En d'autres mots, une interface représente une spécification de la fonctionnalité qui doit être accessible.

4.2 Rappel sur les solutions existantes

Actuellement, les méthodes existantes permettent de répondre à une partie des demandes pour cette recherche. La section 3.4 a présenté plusieurs méthodes dont certaines fournissent de bonnes bases. Donc, il faut maintenant proposer une solution en se basant sur ce qui existe et ainsi porter l'attention sur ce qu'il faudrait rajouter.

Ainsi, la méthode ATAM, qui permet de faire de la rétro-ingénierie sur une architecture existante (ou proposée), donne un bon moyen pour extraire des attributs de qualité des systèmes patrimoniaux.

De plus, la méthode ADD, qui est une méthode de conception d'architecture logicielle se basant sur les attributs de qualité, offre des aspects réutilisables et applicables dans notre contexte. En effet, il faut proposer une nouvelle architecture logicielle à partir des

attributs de qualité provenant des systèmes patrimoniaux et de nouveaux attributs de qualité.
En bref, ces deux méthodes donnent directement et indirectement des moyens pour définir et documenter les attributs de qualité dans le contexte de la conception de l'architecture logicielle. Une technique directement utilisable est l'arbre d'utilité d'ATAM qui donne un moyen de décomposition des attributs de qualité vers des scénarios. Une technique indirecte est que les deux méthodes utilisent des scénarios, ce qui implique d'avoir une manière de bien les définir et de bien les documenter.

Toutefois, l'étude ne doit pas s'arrêter sur une méthode proposant un mélange de rétro-ingénierie et de réingénierie comblant les besoins. Il ne faut surtout pas oublier l'aspect de définition des attributs de qualité, car ceux-ci sont nécessaires en entrée (provenant d'un document de spécification par exemple), mais aussi lors la décomposition d'un attribut de qualité en sous-attributs. Plusieurs auteurs (voir section 3.4) proposent des modèles de qualité qui essentiellement fournissent des moyens de définir des attributs de qualité plus adaptés aux besoins spécifiques du projet. Un de ces modèles devrait donc être utilisé durant la décomposition.

Pour conclure, il faut noter que tous les éléments sont présents pour remplir les besoins. En effet, l'utilisation de la méthode ATAM avec un modèle de qualité, répond aux besoins de notre partenaire en définissant et documentant les attributs de qualité et la méthode ADD explique comment organiser la conception de l'architecture logicielle.

Mais, notre partenaire souhaite également actualiser sa solution architecturale en introduisant les aspects de lignes de produits et de composants. Pour la ligne de produits, il existe des méthodes et des modèles de qualités qui sont applicables et fournissent la majeure partie des réponses aux besoins. Par contre, nous n'avons pas trouvé d'études sur les qualités architecturales en général par rapport aux composants.

Seules l'étude de solutions existantes utilisées par d'autres entreprises et la validation de la technologie par rapport à la proposition d'architecture logicielle reste à faire. La première est une simple recherche d'information alors que la validation technologique doit faire l'objet d'une recherche sur les forces et les faiblesses de la technologie pour vérifier que les faiblesses ne sont pas préjudiciables à la proposition d'architecture.

Presque toutes les étapes définies dans le sujet de cette étude sont couvertes avec les méthodes existantes. La section suivante va toutefois montrer qu'il reste des améliorations à apporter pour fournir une méthode systématique.

4.3 Problématique

La problématique est donc de définir une méthode simple et efficace d'étude de ces caractéristiques d'architecture logicielle, mais surtout de s'assurer que la méthode simplifie la conception architecturale. Cette simplification a pour but de diminuer le risque et de rendre cette étape plus contrôlable et répétable. Actuellement, les méthodes proposées dans la littérature reposent souvent sur des moyens ad hoc, ce qui limite leur utilisation dans un contexte plus large et les rendent dépendantes du niveau d'expertise de l'architecte. Aussi, il faut considérer les aspects des lignes de produits et des composants pour s'assurer qu'ils seront pris en compte par notre proposition.

En se basant sur la revue de littérature (faite dans le chapitre 3) et sur les solutions existantes (voir les sections précédentes), il est nécessaire de trouver des solutions aux problèmes suivants :

1. définition des attributs de qualité. Cet aspect est très important lors de la décomposition des attributs de qualité de haut niveau (venant d'un document de spécification par exemple) en sous-attributs plus spécifiques;

2. contrôle de la qualité des résultats obtenus lors de la décomposition. Cet aspect permet de s'assurer que l'architecture va répondre aux bons besoins;
3. simplification du passage d'une liste de scénarios de qualité vers une architecture logicielle. Cet aspect est au centre de nos préoccupations pour diminuer le risque lié à la conception d'une nouvelle architecture.

Ainsi en utilisant une méthode existante, seuls trois points restent à étudier. Mais ces trois points sont très importants car ils contribuent à diminuer le risque lié à toute nouvelle architecture.

Une discussion plus détaillée des ces trois points est faite dans la section suivante, qui introduit la solution proposée.

4.4 Discussion

Dans les chapitres précédents, nous avons présenté l'état de l'art concernant les modèles d'attributs de qualité et leur intégration dans la conception d'architecture logicielle. Ainsi, plusieurs méthodes prennent en compte les attributs de qualité ou, au minimum, s'assurent que l'architecture les supporte réellement.

Par exemple, les attributs de qualité sont au cœur de la méthode de conception d'architecture logicielle ADD pour s'assurer qu'ils sont bien supportés. De plus, l'arbre d'utilité d'ATAM (réutilisé par ADD) offre un moyen visuel simple et efficace pour la décomposition des attributs de qualité. Toutefois, il faut convenir que certains points nécessitent plus d'études car ils sont peu ou pas expliqués. Or, ils sont cruciaux pour bien remplir les besoins.

D'abord, toutes les méthodes suggèrent de procéder à une décomposition d'un attribut de qualité vers des scénarios plus précis, sans toutefois expliquer comment on peut vérifier que le scénario obtenu est correct. Cet aspect est pourtant primordial pour s'assurer que l'architecture répond au bon problème. De plus, un mécanisme de contrôle permettrait d'améliorer la qualité des artéfacts et de leur contenu. En effet, la validité du résultat obtenu ne fait l'objet d'aucune recherche pour le moment. La littérature propose peu de solutions, tout en indiquant que cette étape est critique. Cela révèle une contradiction que notre proposition tente d'éliminer.

Ensuite, la décomposition d'un ou plusieurs attributs de qualité en sous-attribut(s) et enfin en scénario est une activité complexe qui demande beaucoup de rigueur pour penser à tous les cas possibles ainsi que de connaissances du domaine et de la technique. L'architecte se repose donc sur ses connaissances et sur les avis d'experts tels que ceux du client. Mais même avec le support d'experts, les propositions de scénarios sont très liées à la compréhension de l'attribut de qualité et de son contexte. Aucune des méthodes mentionnées ne fournit d'explications sur le moyen de réduire ce risque. On revient donc au même type de questions dont nous avons débattu dans le paragraphe précédant, sur le ou les moyens de contrôler la qualité de la décomposition. Le problème potentiel engendré par cette ambiguïté est que le résultat peut être bon dans le sens d'une bonne décomposition, mais mauvais si le contexte a été mal compris. Quoi qu'il en soit, l'architecte ne répond pas au bon problème.

De plus, il est important de signaler que la transition d'une liste de scénarios vers une architecture logicielle est peu expliquée par toutes les méthodes mentionnées. En effet, ADD et la méthode de Bosch parlent d'utiliser les styles ou les patrons, en fait toutes les solutions prédéfinies connues et documentées. Mais, aucune n'explique en détail comment les patrons sont agrégés pour fournir l'architecture par exemple. Cependant, certains auteurs (Gamma, Helm, Johnson, & Vlissides, 1995; Schimdt, Stal, Rohnert, & Buschmann, 2000) très liés à la communauté s'intéressant aux patrons, se sont penchés

sur la problématique d'un « langage des patrons » dans le but de définir un métalangage pour le génie logiciel. Dans cette optique, ils ont montré et expliqué les relations de décomposition ou de délégation qui existent entre tous leurs patrons. Mais à notre connaissance, ils n'ont pas expliqué en détail la manière de les composer au niveau de l'architecture logicielle. Seuls Schmidt *et al.* (2000) font référence à des problèmes d'orthogonalité entre certains patrons (c'est-à-dire des patrons qui ne sont pas composables), ce qui met en évidence la difficulté de concevoir une architecture logicielle à partir de patrons uniquement.

Ainsi, l'architecte doit être un expert sinon les méthodes expliquées ci-dessus sont difficilement utilisables. Il est improbable qu'une entreprise ne se repose pas sur un architecte expérimenté pour définir une nouvelle architecture logicielle même en utilisant une des méthodes mentionnées. Comment l'architecte pourrait trouver les bonnes solutions et les bons compromis s'il n'a pas déjà réalisé plusieurs architectures ? Ce point est pourtant central pour toute méthode de conception d'architecture logicielle.

Enfin, tous les auteurs traitant des attributs de qualité par rapport à l'architecture logicielle proposent d'utiliser les scénarios pour modéliser ces attributs de qualité. Leurs propositions se basent sur le fait que les méthodes d'analyse des exigences fonctionnelles utilisent les scénarios. Mais, seuls Bass *et al.* (2003) se sont penchés sur le fait qu'un scénario pour des exigences non fonctionnelles n'a peut-être pas tout à fait la même forme qu'un scénario pour des exigences fonctionnelles. Cependant, même leurs travaux semblent incomplets. En effet, ils ont décomposé un scénario en plusieurs constituants, mais ils n'ont rien proposé pour contrôler la qualité de ces constituants, ce qui est un point important.

En résumé, les manques observés durant la recherche bibliographique sont très souvent liés au contrôle de la qualité des artéfacts. Le but visé par cette étude est de fournir une

méthode systématique pour concevoir une architecture logicielle adéquate à un contexte précis et qui ne néglige pas la qualité.

Pourtant, la littérature donne déjà certaines réponses à des problèmes très semblables. Il existe déjà des travaux réalisés pour contrôler la qualité d'exigences fonctionnelles. Par exemple, la norme IEEE Std 830 (1998) a défini des caractéristiques auxquelles une exigence et un ensemble d'exigences doivent se conformer. Leffingwell et Widrig (2000) ont d'ailleurs étendu cette liste. Ainsi, ces travaux peuvent être adaptés aux scénarios. La qualité d'un scénario pourra être contrôlée de manière plus objective. Le résultat de la décomposition peut être contrôlé de la même manière, car les travaux mentionnés traitent aussi d'une liste d'exigences.

De plus, Bass *et al.* (2003) ont donné des indications sur les constituants d'un scénario pour les exigences non fonctionnelles. Il faudrait en plus ajouter des caractéristiques sur chaque constituant et sur le scénario au complet. Nous n'avons pas trouvé de travaux concernant ce point. Nos expériences professionnelles nous ont donc servi pour établir une première liste.

Enfin, la transition d'une liste de scénarios vers une architecture logicielle reste peu documentée. Toutefois, on peut au minimum utiliser une technique très répandue qui consiste à diviser un problème en plusieurs sous-problèmes plus facilement identifiables et résolubles (« diviser et conquérir »). Au lieu de considérer la liste complète des scénarios, on peut étudier les solutions de chaque scénario un à un pour ensuite chercher la solution globale à partir de toutes les solutions partielles.

Cette solution impose alors une phase de composition de chaque solution partielle. Mais ce problème reste entier. En effet, les attributs de qualités peuvent être exclusifs et les patrons architecturaux peuvent proposer des solutions contradictoires. L'architecte expérimenté est capable de trouver les bons compromis en s'inspirant de son expérience.

Il serait donc opportun de trouver une manière de modéliser ce mécanisme de compromis et de composition.

Finalement, la partie de validation par rapport à la technologie nécessite aussi plus d'attention car la compréhension des limitations d'une technologie et d'une architecture logicielle est encore souvent laissée à un ou plusieurs experts (incluant l'architecte). La littérature est également peu abondante sur ce point. Par exemple, la récente controverse[11] entourant les performances de .NET de Microsoft par rapport à J2EE de Sun Microsystems montre bien qu'un choix technologique impose certains choix architecturaux et parfois impose aussi d'autres choix technologiques (c'est-à-dire que cela peut introduire un effet de cascade à cause des problèmes de comptabilités). En plus, le fait de choisir une architecture parmi plusieurs propositions représente aussi un problème. Toutes les propositions sont valables mais un architecte expert sera capable de sélectionner celle qui offrira le plus de chance de répondre le mieux aux besoins (c'est-à-dire aux exigences non fonctionnelles). Il faudra donc trouver un moyen pour aider l'architecte dans sa prise de décision.

4.5 Conclusion

Dans ce chapitre qui clôt la recherche bibliographique, nous nous sommes assurés de résumer clairement ce qui existe et ce qui manque.

En effet, nous avons repris telle quelles certaines définitions et ajouté ou modifié certaines autres pour mieux nous aligner sur le contexte de cette recherche. De plus, nous avons expliqué en détail chacune des solutions existantes et montré leurs forces et

[11] Le résultat de la controverse n'est pas envisagé ici. Seules les limitations introduites par une technologie sont intéressantes dans cette discussion pour montrer les conséquences d'un choix technologique. Ces limitations sont exprimées indirectement dans les publications bien qu'elles en soient les points les plus importants à notre avis.

leurs faiblesses. Enfin, nous avons approfondi la discussion sur les faiblesses pour bien montrer ce qu'il faudrait ajouter ou modifier pour les minimiser.

Les chapitres suivants expliquent notre proposition. En particulier, le chapitre 5 donne un aperçu global de notre méthode ADQA dans le but de bien comprendre sa finalité pour ensuite donner une explication détaillée de la première phase de ADQA dans le chapitre 6.

CHAPITRE 5

PROPOSITION D'UNE MÉTHODE

5.1 Vision globale

À partir des investigations au sujet des méthodes de conception d'architecture logicielle basées sur les exigences non fonctionnelles (c'est-à-dire les attributs de qualité), nous proposons de décomposer notre solution en trois phases indépendantes mais qui forment un tout cohérent. De plus, ces trois phases s'intègrent parfaitement aux méthodes existantes telles que ATAM, ADD et Bosch par exemple. Les trois phases sont : la définition, l'agrégation et la validation au niveau de l'architecture logicielle.

La première phase définit et documente les attributs de qualité soit dans le contexte de systèmes patrimoniaux, soit dans le contexte d'une nouvelle architecture. Il faut récupérer la connaissance existante dans les systèmes patrimoniaux et rajouter de nouveaux attributs de qualité pour actualiser et uniformiser la nouvelle architecture logicielle. Les attributs recherchés sont ceux dérivant du besoin d'avoir une architecture de lignes de produits et d'avoir un support adéquat pour les composants. De plus, cette phase va fournir des spécifications architecturales (incluant des stratégies telles que les patrons architecturaux (Schimdt, Stal, Rohnert, & Buschmann, 2000), les styles, etc.) pour l'architecture logicielle. Toutefois, il est à noter que cette phase devrait être utilisée en complément avec une méthode existante.

La seconde phase est l'agrégation des spécifications architecturales (en réalité des stratégies proposées pour les attributs de qualité) pour proposer une ou plusieurs architectures logicielles. Cette phase repose sur l'agrégation des stratégies et l'étude des relations entre ces stratégies. Cette partie s'avère complexe car les stratégies peuvent

être exclusives. Ces propositions d'architecture serviront d'étalons de comparaison avec l'architecture logicielle de haut niveau déjà imaginée.

La troisième phase valide la proposition d'architecture logicielle en fonction du contexte de sa mise en place. Les deux points cruciaux sont : le support de la technologie offert à l'architecture logicielle et l'étude des problèmes équivalents pour en faire ressortir des solutions documentées et étudier leur applicabilité.

Nous avons vérifié la pertinence de notre approche avec les travaux des auteurs Nord et Soni (2003). Les auteurs proposent quatre étapes principales que notre proposition va reprendre et étendre. En effet, les trois étapes de notre proposition offrent : la phase de définition qui supporte les étapes 1, 2 et 3, la phase d'agrégation qui supporte l'étape 4 et la phase de validation qui n'est pas couverte par les auteurs.

L'objectif ultime de cette proposition est de mieux définir et documenter les décisions derrière la transition entre les exigences non fonctionnelles et une architecture logicielle. Plus précisément, la nouveauté sera d'établir un lien entre une liste de scénarios (dérivant des exigences non fonctionnelles) et une architecture logicielle et de contrôler la qualité de ce ou ces liens. En effet, l'arbre d'utilité de l'ATAM couvre déjà le lien entre les exigences non fonctionnelles et la liste de scénarios mais non leur qualité. De plus, l'arbre d'utilité ne couvre pas le lien entre les scénarios et l'architecture.

Cette proposition d'architecture ne sera pas l'architecture finale car il faut la raffiner et, en parallèle, effectuer la décomposition fonctionnelle. En suivant ces trois phases intégrées dans une méthode, les liens avec les exigences non fonctionnelles seront plus clairs et mieux documentés, ce qui devrait donner un meilleur guide pour le conception de l'architecture logicielle. Habituellement, ceci est réalisé (mais pas toujours documenté) par un architecte expérimenté en développement logiciel et dans le domaine

d'activité. Avec cette proposition, les architectes devraient mieux documenter leurs décisions et la conception d'une architecture devrait être plus répétable et prévisible.

Évidemment, si cette méthode est utilisée seule, elle doit être itérative pour permettre le développement incrémental de l'architecture logicielle et la validation continuelle. Nous recommandons tout de même que notre méthode soit utilisée en complément des méthodes déjà existantes.

Toutefois, notre méthode n'explique pas quelle stratégie utiliser pour un attribut de qualité spécifique et ne propose pas de méthode pour extraire ou définir les besoins pour un produit logiciel. Donc, elle doit être supportée par un processus de développement tel que RUP ainsi que par des méthodes existantes (ADD, Bosch ...) et des catalogues de solutions documentées et vérifiées (patrons, styles, etc.).

Cette méthode se nomme ADQA pour « **A**rchitecture **D**esign based on **Q**uality **A**ttributes ».

La figure 12 présente les trois étapes de la méthode proposée. La phase 1 permet le passage des attributs de qualité vers des spécifications pour l'architecture logicielle. La phase 2 transforme les spécifications en propositions d'architecture logicielle. Enfin, la phase 3 choisit une proposition et la valide. Évidemment, la conception de l'architecture n'est pas terminée mais sa structure fondamentale sera documentée et approuvée.

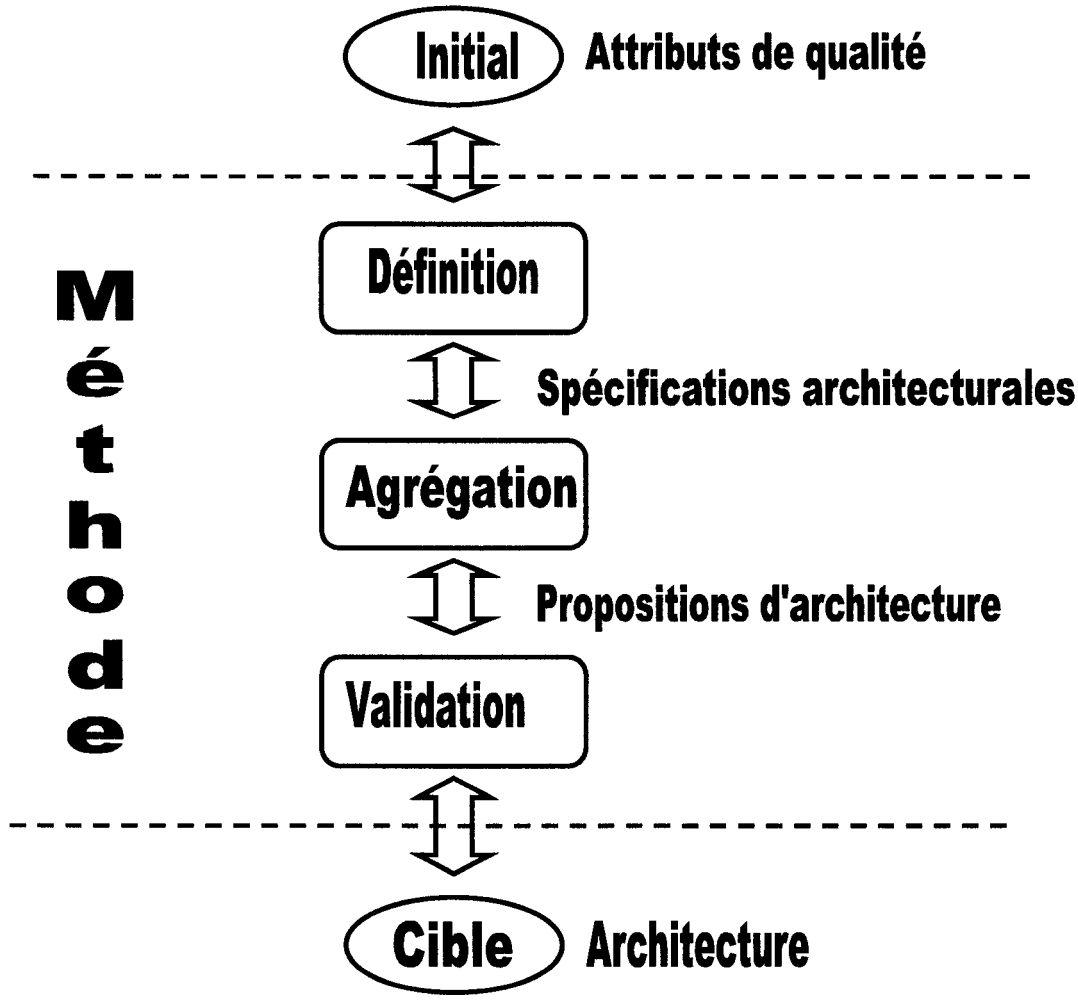

Figure 12 Présentation de la méthode ADQA

Comme on peut le constater, ces phases peuvent très bien s'intégrer en groupe ou séparément dans des méthodes déjà existantes, car toutes les démarches et les artéfacts seront utiles dans un contexte d'ingénierie d'une nouvelle architecture ou de réingénierie d'une architecture existante.

Cette proposition est basée principalement sur les méthodes ADD (Bass et al., 2003), ATAM (Bass et al., 2003) et Bosch (Bosch, 2000).

5.2 Modèle pour les attributs de qualité

Avant de présenter notre proposition pour un modèle servant à la définition des attributs de qualité, il convient d'expliquer les travaux existants qui vont servir de base à cette proposition.

En se basant sur les travaux de Firesmith (2003b) et de Anastasopoulos *et al.* (2000), il apparaît clairement que le modèle de qualité doit permettre d'adapter les attributs de qualité standard (ISO/IEC, 1999) au contexte du produit logiciel. En effet, même les attributs standard ont peut-être une étendue ou une signification un peu différentes suivant le domaine d'application.

Nous proposons donc une approche basée sur deux niveaux complémentaires :

- la perspective externe (à l'architecture logicielle étudiée) basée sur les attributs de qualité venant des normes, de la littérature et du domaine. Ces attributs sont bien définis et bien documentés car ils correspondent à des cas fréquents.
- la perspective interne basée sur des attributs de qualité venant des besoins techniques internes du fait d'un contexte particulier. Ces attributs peuvent provenir d'exigences ou de contraintes internes, très liées à la technique. Ils peuvent se transformer en attributs de qualité externes ou imposer des stratégies « maison » liées au contexte technique.

Toutefois, nous ne reprenons pas la notion de « visibilité ou non en exécution » venant de ISO/IEC 9126 car l'architecte considère l'architecture comme une « boite blanche » et non comme une « boite noire ». L'architecte doit pouvoir accéder à toute l'architecture et ne devrait donc pas faire de distinctions entre une vision externe et

interne de l'architecture. Pour lui, un attribut a des conséquences sur l'architecture qui lui seront toujours « visibles ».

Pour la perspective externe, l'architecte doit d'abord faire une recherche sur ce qui existe actuellement dans le but de le réutiliser et de l'adapter si nécessaire. La figure 13 montre les trois aspects et leur adaptation en entrée.

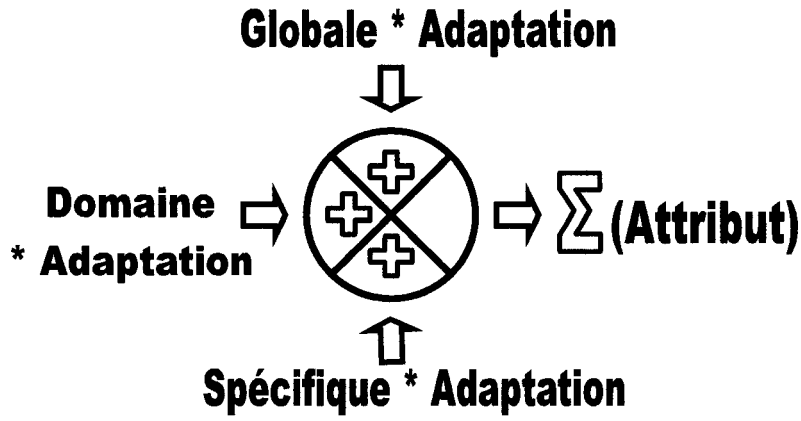

Figure 13 Modèle de qualité – perspective externe

La première entrée, nommée « Globale », comprend entre autres, les normes existantes qui reflètent le statut des connaissances (par exemple ISO/IEC 9126). Il peut s'agir de réutiliser un attribut de qualité déjà très bien défini dans la littérature qui sera applicable (après certaines adaptations) au contexte du produit en cours d'étude. La seconde entrée, nommée « Domaine », englobe les attributs de qualité (présents dans la littérature) spécifiques au domaine. Par exemple, des attributs de qualité spécifiques au temps réel sont peut-être sans signification ailleurs. Enfin, la troisième entrée, nommée « Spécifique », comprend les contraintes ou les besoins venant du projet ou de l'entreprise mais qui sont communs à tous les projets. Une entreprise pourrait avoir un besoin ou une contrainte, lié à l'organisation de son mode de production qui a un impact

sur l'architecture logicielle. Par exemple, l'entreprise possède déjà une organisation du travail efficace qui va imposer une certaine décomposition modulaire, même si cette décomposition n'est pas la solution architecturale optimale. Les auteurs Dikel, Kane et Wilson (2001) expliquent bien ce lien entre l'organisation d'une entreprise et l'architecture (et vice-versa). Un autre exemple possible est un attribut « créé » pour un contexte précis (comme « Iowa-bility » cité comme exemple dans l'ouvrage de Bass *et al.* (2003)) qui ne représente rien à l'extérieur du projet, mais a un impact à l'intérieur d'une entreprise.

Pour la perspective interne, l'architecte doit d'abord faire une recherche sur les besoins exprimés pour ce produit logiciel particulier. La figure 14 montre des entrées possibles et l'adaptation nécessaire. Les entrées dépendent évidemment du contexte de développement, des outils, du temps alloué, etc.

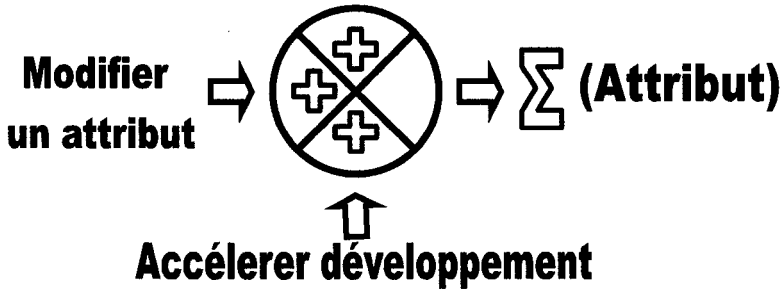

Figure 14 Modèle de qualité – perspective interne

Toutefois, cela va permettre d'accélérer le développement : en réduisant la courbe d'apprentissage (garder le même environnement de développement, même si ce n'est pas le mieux adapté), en réutilisant des composants existants (même s'ils ne sont pas très bien adaptés), etc. Par conséquent, cela impose d'adapter ou de créer des attributs de qualité sans signification dans un autre contexte. Enfin, un deuxième exemple est un changement majeur d'une fonctionnalité la faisant sortir de la vision originale du produit (peu importe les raisons de ce choix). Par exemple, un produit doit supporter une moyenne de 10 utilisateurs et le changement consiste à lui faire supporter 500

utilisateurs (cet exemple est librement adapté d'une expérience réelle). Dans un tel contexte, certains choix architecturaux ne sont plus valables et doivent être changés. Toutefois, il faut livrer cette fonctionnalité le plus vite possible sans radicalement changer l'architecture pour éviter de la déstabiliser. Une solution possible est de décomposer l'attribut en une suite de changements applicables sur une longue période, ce qui entraîne de créer des attributs « intermédiaires » très liés à ce contexte, sans aucune signification pour un autre projet.

De plus, notre partenaire souhaite s'intéresser aux aspects de lignes de produits et de composants. Ces deux aspects sont largement couverts dans la littérature. Par exemple, l'aspect de la ligne de produits peut être considéré comme la manière de gérer les différences entre les produits. Il faudrait avoir un attribut de qualité pour exprimer cette variabilité. À priori, ces deux aspects feront partie de la perspective externe car ils proviennent de demandes extérieures à l'architecture logicielle et, en plus, sont connus et bien documentés.

5.3 Survol de la méthode ADQA

Dans cette section, nous allons faire un survol de la méthode proposée dans le but de bien expliquer sa logique. En effet, il est important de bien comprendre chaque phase (avec ses intrants et ses extrants) avant de commencer une explication détaillée de la phase de définition.

5.3.1 Phase 1 : Définition

L'objectif de cette phase est de mieux définir les attributs de qualité. Le résultat de cette phase sera une suite de documents qui définiront précisément les attributs de qualité (avec un ou plusieurs scénarios) et qui donneront des stratégies pour répondre

adéquatement à ces attributs dans l'architecture logicielle. L'ensemble des stratégies architecturales forme les « spécifications architecturales ».

Le nom de cette phase est QAAs pour « **Q**uality **A**ttribute **A**ssessment method » et la figure ci-dessous en donne une vue globale.

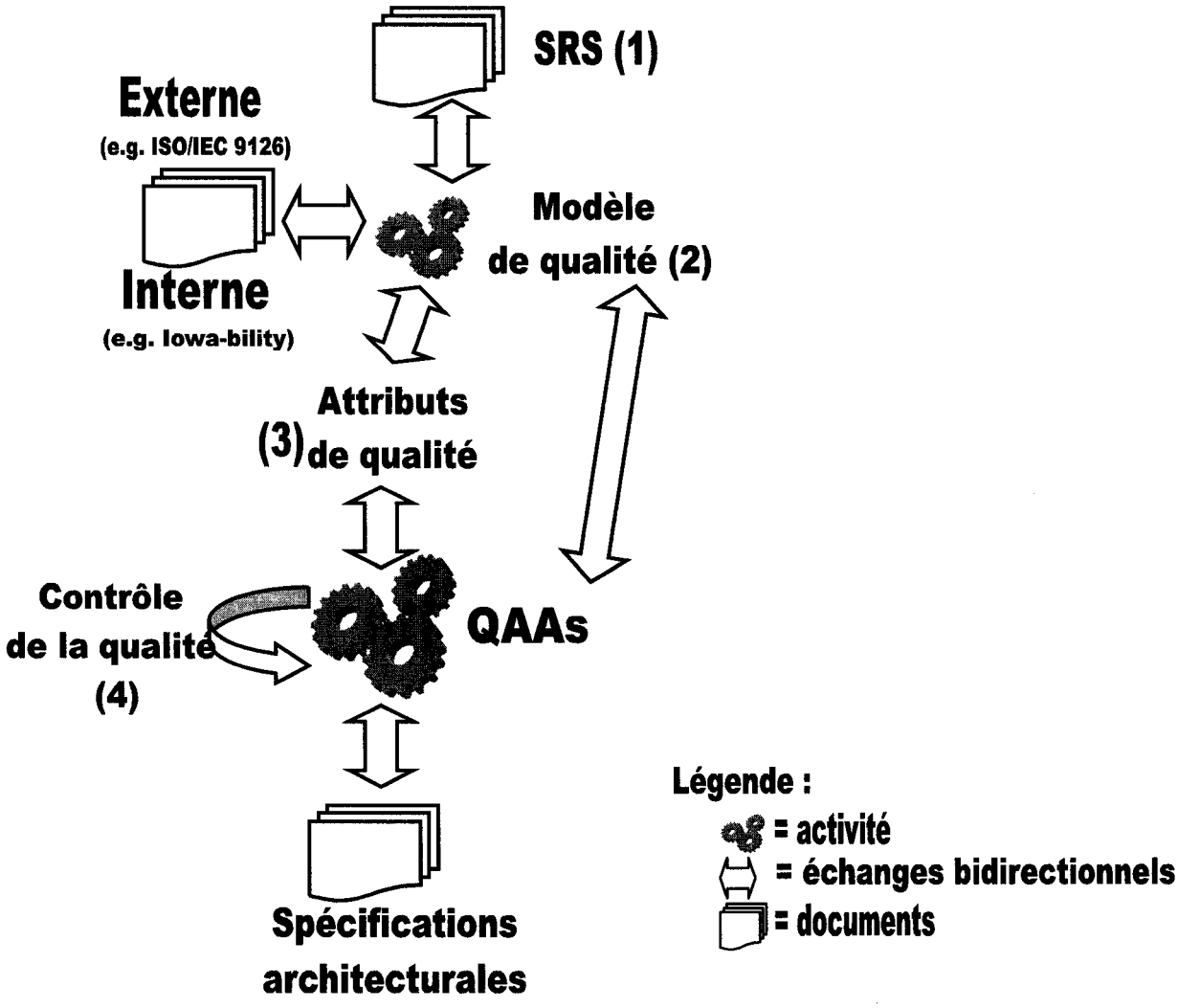

Figure 15 Phase de définition

La figure 15 montre les intrants, les extrants et les traitements de cette phase. D'abord, il convient d'établir la liste des attributs de qualité (voir (3) dans la figure précédente) en se basant sur les documents existants tel que le document SRS (voir (1) dans la figure précédente) dans le cas d'une nouvelle conception. Si l'étude se base sur des produits logiciels existants, la méthode ATAM sera un bon support pour extraire les attributs de qualité ou mettre à jour la documentation existante. Cette recherche doit se faire à partir du modèle de qualité proposé (voir (2) dans la figure précédente).

Ensuite, l'architecte décompose chaque attribut de qualité vers un ou plusieurs scénarios pour chaque attribut de qualité et les documente. Cette étape correspond réellement à la phase de définition (voir QAAs dans la figure précédente). Il existe également un mécanisme pour contrôler la qualité des paires (voir (4) dans la figure précédente). Le concept de paire sera expliqué en détail dans la section **6.1.2**. Cette décomposition va aussi utiliser les modèles de qualité (voir (2) dans la figure précédente).

Finalement, il devra fournir des stratégies pour supporter les attributs de qualité dans la nouvelle architecture. Cette association entre un scénario et une stratégie définit une spécification architecturale. Le concept sera expliqué en détail dans la section **6.1.3**. Le résultat final est donc une liste de spécifications applicables sur l'architecture logicielle.

Notre proposition repose sur un document par attribut de qualité qui contiendra la définition, les scénarios et les stratégies. Cette méthode supporterait très bien l'utilisation d'un outil pour maintenir le contenu et vérifier la complétude des données. Toutefois, l'utilisation d'outils sort du champ de cette étude.

5.3.2 Phase 2 : Agrégation

Le but de cette phase est d'agréger les stratégies architecturales pour fournir une ou plusieurs propositions d'architecture logicielle. Le nom de cette phase est QAAg pour

« **Q**uality **A**ttribute **Ag**gregation method » et la figure ci-dessous en donne une vue globale.

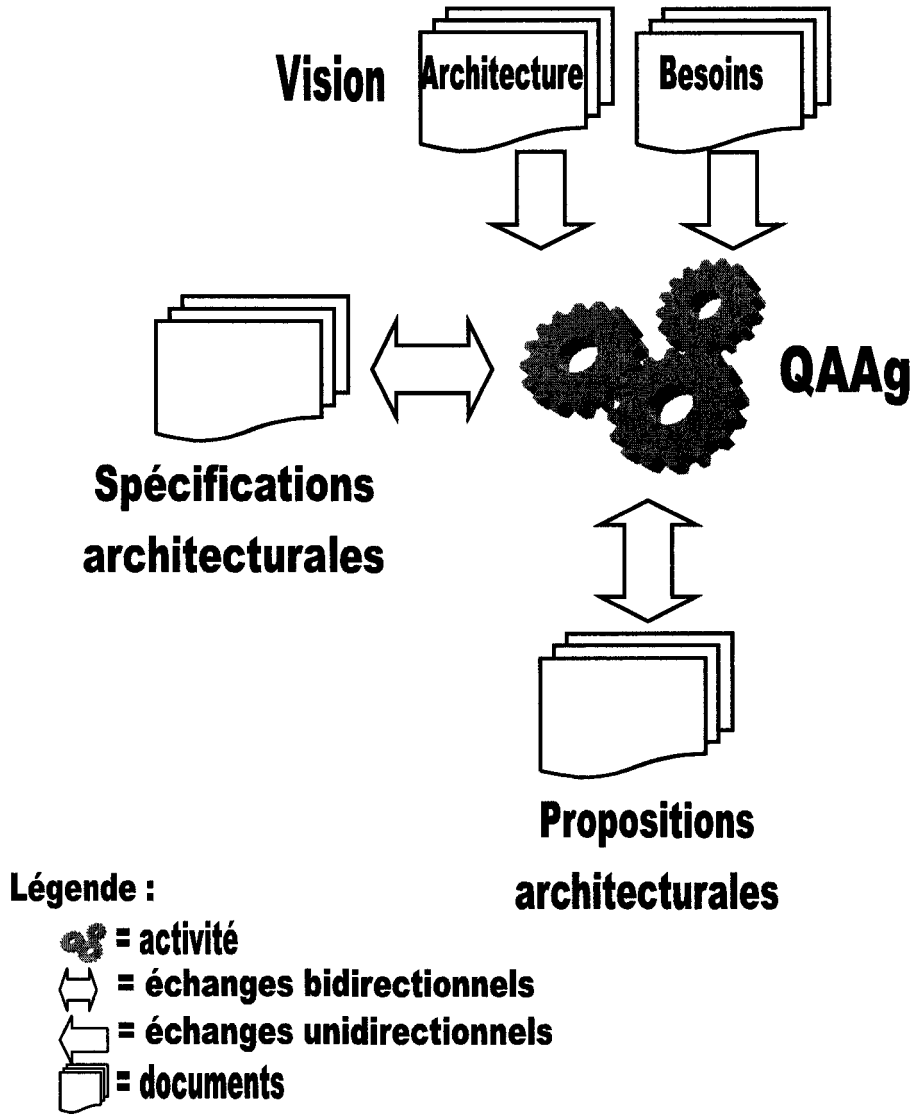

Figure 16 Phase d'agrégation

La figure 16 montre bien les intrants de cette phase qui sont : les spécifications architecturales (venant de la phase 1) et la vision (si elle existe) pour une architecture et / ou pour l'introduction d'attributs de qualité déjà définis et documentés (par exemple, les

aspects lignes de produits et composants). Ces nouveaux besoins ont peut-être été définis par la phase 1 ou, tout simplement, proviennent de la littérature. L'extrant consiste en une ou plusieurs propositions d'architecture. Dans notre cas, il existe déjà une solution architecturale envisagée (voir figure 2, page 5), qui devra être prise en compte durant cette phase.

5.3.3 Phase 3 : Validation

L'objectif de cette phase est de choisir et de valider une proposition parmi les propositions d'architecture logicielle venant de la phase 2. Le nom de cette phase est AV pour « Architecture Validation method ».

Comme le montre la figure 17, les intrants de cette phase sont les propositions d'architecture logicielle venant de la phase 2. Cette phase doit tenir compte de la technologie choisie (pour l'implantation du produit logiciel) pour supporter l'architecture logicielle. De plus, l'entreprise ABC souhaite rechercher des expériences équivalentes pour en extraire des enseignements applicables à son contexte. L'extrant doit être une seule architecture logicielle approuvée.

La prochaine étape serait de raffiner l'architecture pour obtenir une architecture logicielle détaillée. Dans le cas de réingénierie, il faudrait comparer l'architecture avec l'architecture existante et faire les correctifs.

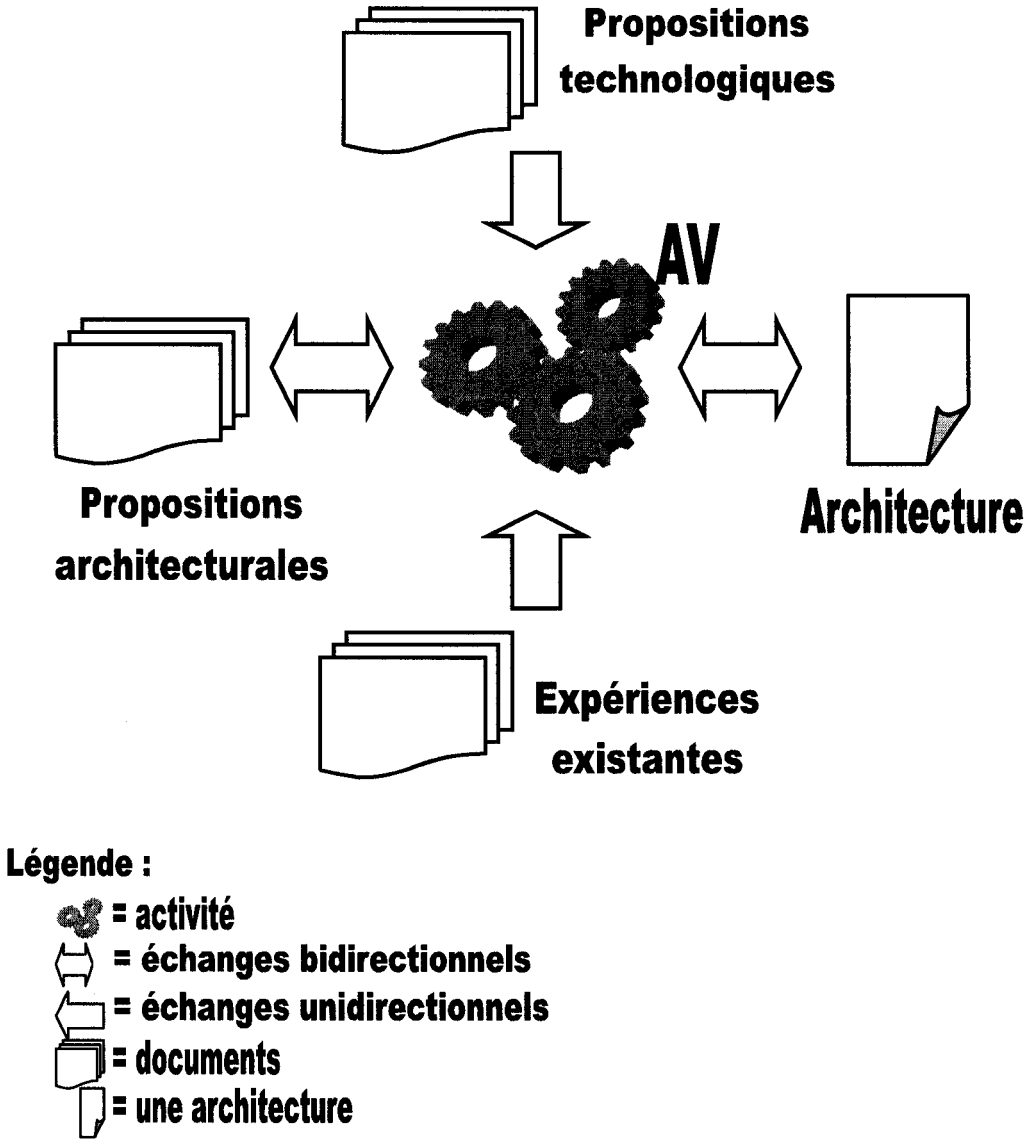

Figure 17 Phase de validation

5.4 Conclusion

L'objectif de la méthode ADQA est de faciliter la transition des attributs de qualité vers une architecture logicielle en améliorant la définition et la documentation des attributs de qualité mais, surtout, en simplifiant cette étape. Dans cette optique, la méthode a été décomposée en trois phases : la phase 1 (QAAs) concerne la définition des spécifications architecturales, la phase 2 (QAAg) se concentre sur l'agrégation des spécifications architecturales dans le but de fournir une ou plusieurs propositions d'architecture logicielle et enfin, la phase 3 (AV) doit choisir et valider une architecture logicielle.

Comme nous l'avons mentionné dans les chapitres précédents, certaines méthodes offrent déjà des éléments de réponse à cette problématique. Mais aucune ne fournit de moyen pour contrôler la qualité des artéfacts. Donc, la méthode ADQA donne des moyens de contrôle au niveau de chaque artéfact demandé.

Les chapitres suivants vont expliquer les trois phases de cette méthode. Du fait des contraintes de temps, seule la phase 1 a été élaborée en détail. Toutefois, les phase 2 et 3 sont quand même expliquées sommairement pour permettre de mieux comprendre la vision d'ensemble.

CHAPITRE 6

DÉTAILS DE LA MÉTHODE

6.1 Phase de définition

Comme nous l'avons expliqué dans la section 5.3, cette première phase sert à mieux définir les attributs de qualité pour en extraire les spécifications architecturales. Cette phase reprend l'arbre d'utilité de l'ATAM (Bass et al., 2003), qui donne une représentation visuelle et simple de la décomposition des attributs de qualité généraux venant des documents existants vers des attributs plus spécifiques puis vers des scénarios. Ces scénarios représentent donc la modélisation d'une partie d'un attribut de qualité.

Toutefois, les méthodes de conception d'architecture logicielle montrent quelques faiblesses (voir la section 4.3). Pour répondre à ces faiblesses et pour faciliter la transition entre les attributs de qualité et une architecture logicielle, cette section présente les concepts qui vont être utilisés par la première phase de notre méthode.

Premièrement, nous proposons le concept de "paire" pour nous assurer de la qualité des scénarios résultant de la décomposition des attributs de qualité. Ce concept est au coeur de cette phase car il s'agit du premier résultat dont la qualité peut être contrôlée. Une paire est composée d'un scénario et d'une validation. La validation est un moyen donné à un intervenant pour valider le scénario sur le produit logiciel. Donc, ce moyen doit être compréhensible et utilisable par tous les intervenants. De plus, une paire doit respecter certaines caractéristiques pour s'assurer qu'elle est de bonne qualité (voir la section 6.1.2 pour plus de détail). Cette technique représente la base du mécanisme de contrôle de la qualité dans cette phase.

Deuxièmement, nous proposons le concept de « spécification architecturale ». Une spécification est composée d'une paire et d'une ou plusieurs stratégies (voir la section 6.1.3 pour plus de détails). La définition des spécifications est une étape intermédiaire entre les scénarios et l'architecture logicielle dans le but de clarifier et de documenter les raisons derrière les décisions prises pour une architecture logicielle. De plus, une spécification architecturale doit respecter certaines caractéristiques pour être de bonne qualité. Ainsi, si toutes les spécifications architecturales sont de bonne qualité alors les artéfacts (qui se basent sur les spécifications) devraient aussi être de bonne qualité.

Enfin, l'arbre d'utilité de l'ATAM est réutilisé sous la même forme respectant les recommandations de la méthode ADD. La décomposition repose aussi sur des scénarios, comme proposé par toutes les méthodes existantes telles que ATAM, ADD et Bosch.

Après avoir introduit brièvement les nouveaux concepts qui ont été proposés pour répondre aux besoins de notre méthode, les sections suivantes vont les définir en détail.

6.1.1 Concept de scénario

La méthode est basée sur le décomposition d'attributs de qualité de haut niveau en un ou plusieurs scénarios. Évidemment, cette étude ne va pas définir le concept de scénario. Ce concept est déjà très bien défini et il existe plusieurs méthodes utilisant les scénarios et proposant des gabarits pour bien les documenter.

Le problème est plus dans la définition d'un « bon » scénario. Il s'agit d'un enjeu majeur pour toutes les méthodes se basant sur les scénarios. En effet, une mauvaise définition peut conduire à plusieurs interprétations et donc, à ne pas bien remplir les besoins des clients.

Pour étayer notre propos, voici un exemple typique de scénario qui va faire ressortir l'importance de contrôler la qualité du scénario. Le scénario est : « Le client doit pouvoir avoir les résultats d'une opération en moins de 30 secondes ». Ce scénario semble assez précis en quantifiant la performance demandée. Il serait ainsi facile de construire un environnement de tests et de mesurer la performance. Toutefois, ce scénario reste très ambigu car :

1. Que veut dire « avoir les résultats » ?
    a. Est-ce la disponibilité des résultats ?
    b. Est-ce la disponibilité à l'affichage ?
    c. Est-ce l'affichage du résultat et des résultats dérivés ?
2. Que veut dire « en moins de 30 secondes » ?
    a. Est-ce toujours en moins de 30 secondes toutes les fois (c'est-à-dire qu'il s'agit d'une limite infranchissable) ?
    b. Est-ce en moins de 30 secondes 95% du temps ?
3. Parlons-nous de toutes les opérations possibles du produit ?
    a. L'équipe de développement peut choisir l'opération qui lui plaît ?
    b. Y a-t-il des opérations plus simples que d'autres ?
    c. Si le client utilise très souvent une opération, doit-on faire le test sur cette opération ?
4. Y a-t-il des contraintes externes qui influencent la performance ?
    a. Combien y a-t-il de clients sur l'application lors du test ?
    b. Y a-t-il une dépendance sur la bande passante d'un réseau (échange de données sur le réseau? Taille des données à échanger?...) ?
    c. Y a-t-il une dépendance sur une base de données (échange de données sur le réseau? Taille des données à échanger?...) ?

Voici un petit aperçu d'un scénario qui semble peu ou non ambigu, mais qui en fait laisse beaucoup d'interprétations possibles. Ce problème va se refléter dans les stratégies choisies et, par conséquent, dans l'architecture logicielle.

Dans un premier temps, nous allons étudier la définition et les caractéristiques des constituants d'un « bon » scénario.

Larman (2002, pp. 49-59), Cockburn (2001, pp. 119-138) et Jacobson *et al.* (1999) proposent des gabarits pour les cas d'utilisation qui sont très liés à la modélisation d'exigences fonctionnelles. Cela pourrait en limiter l'application dans le contexte des attributs de qualité, mais cela donne des indications réutilisables concernant les éléments importants pour un scénario.

De plus, Bass, Klein et Moreno (2001) dans le rapport du SEI intitulé « Applicability of General Scenarios to the Architecture Tradeoff Analysis Method », discutent des difficultés de définir un bon scénario. Pour aider l'intervenant à définir un bon scénario, ils proposent que celui-ci soit toujours formé d'au moins un stimulus et d'une réponse. Les mêmes auteurs dans un autre ouvrage (2003, page 75) proposent de décomposer un scénario en six parties qui sont :

1. la source des stimuli, qui est ce qui génère les stimuli;
2. le stimulus, qui est la condition à considérer par le système;
3. l'environnement, qui représente les conditions d'exécution;
4. l'artéfact, qui est ce qui est stimulé par le stimulus;
5. la réponse, qui représente l'activité engendrée;
6. La mesure de la réponse, qui est un moyen de mesurer la réponse.

Il faut noter que les six éléments ne sont pas toujours pertinents pour tous les scénarios, mais il est nécessaire de les vérifier. Effectivement, cette approche simple devrait permettre de mieux définir les scénarios pour des attributs de qualité.

Notre objectif n'est pas de proposer une nouvelle représentation des scénarios, car ceci est déjà très bien documenté, mais plutôt d'améliorer le niveau de qualité des scénarios pour les attributs de qualité. En effet, peu d'auteurs proposent des moyens simples pour vérifier que le scénario est bien écrit. Donc, nous proposons une technique simple de

contrôle de la qualité qui devrait permettre une meilleure définition (et par conséquent une meilleure écriture) d'un scénario. Il existe des éléments minimaux et nécessaires pour définir un scénario et les caractéristiques auxquelles ces éléments doivent se conformer.

Donc, notre proposition est (détails dans le tableau I, page 69) :

1. la définition d'un scénario doit se composer de quatre parties :
    a. le stimulus qui représente le ou les déclencheurs pour ce scénario;
    b. l'artéfact qui représente le comportement de ce qui est simulé (c'est-à-dire du code, un système…);
    c. la réponse qui représente la ou les réponses pour ce scénario;
    d. l'environnement qui représente le contexte d'exécution de ce scénario.
2. le stimulus doit être au minimum :
    a. vérifiable
    b. reproductible
    c. constant (entre les différents essais)
    d. non ambigu
    e. faisable
3. l'artéfact doit être au minimum :
    a. reproductible
    b. constant (entre les différents essais)
4. la réponse doit être au minimum :
    a. vérifiable
    b. constante (entre les différents essais)
    c. non ambiguë
5. l'environnement doit être au minimum :
    a. reproductible
    b. faisable
    c. non ambigu

Si l'intervenant en charge de la rédaction des scénarios se conforme à ces recommandations, il devrait nettement améliorer la qualité des scénarios produits.
Cette technique n'est pas en contradiction avec le concept de paire (voir ci-dessous) car ces caractéristiques sont présentes également pour les paires ou réutilisables par les caractéristiques par paires. Ce constat est logique car la paire est un concept englobant le scénario en rajoutant ses propres contraintes.

Le tableau ci-dessous donne un récapitulatif des caractéristiques avec les définitions.

Tableau I

Caractéristiques pour les constituants d'un scénario

Caractéristiques	**Stimulus**	**Artéfact**
Vérifiable	Il est vérifiable si et seulement si, il existe un processus fini à un coût raisonnable grâce auquel on peut le vérifier.	Non applicable.
Non ambigu	Il est non ambigu si et seulement si, il n'a qu'une seule interprétation possible et pas d'informations cachées.	Non applicable.
Reproductible	Il est reproductible si et seulement si, il existe un processus fini à un coût raisonnable grâce auquel on peut le reproduire.	Non applicable. Remarque 1 : Dans certaines conditions, il peut être important ou nécessaire de simuler l'artéfact (dans le but de faire des tests dans des conditions « infaisables » à un coût raisonnable ou parce qu'il n'est pas disponible car lui-même est en développement). Donc, l'artéfact doit être reproductible. Mais en aucun cas, il nous semble qu'il s'agisse d'une caractéristique obligatoire.

Caractéristiques	Stimulus	Artéfact
Constant	Non applicable.	Il est constant si et seulement si, il réagit toujours de manière identique pour le même stimulus.
Faisable	Il est faisable si et seulement si, il existe un processus fini à un coût raisonnable grâce auquel on peut le faire.	Non applicable.

Tableau II

Caractéristiques pour les constituants d'un scénario (suite)

Caractéristiques	Réponse	Environnement
Vérifiable	Elle est vérifiable si et seulement si, il existe un processus fini à un coût raisonnable grâce auquel on peut la vérifier.	Non applicable.
Non ambigu	Elle est non ambiguë si et seulement si, elle n'a qu'une seule interprétation possible et ne dépend pas d'informations cachées.	Il est non ambigu si et seulement si, il n'a qu'une seule interprétation possible et pas d'informations cachées.
Reproductible	Non applicable.	Il est reproductible si et seulement si, il existe un processus fini à un coût raisonnable grâce auquel on peut le reproduire.
Constant	Elle est constante si et seulement si, elle est exactement identique pour le même stimulus.	Non applicable.
Faisable	Non applicable.	Il est faisable si et seulement si, il existe un processus fini à un coût raisonnable grâce auquel on peut le faire.

Dans un deuxième temps, nous allons étudier les caractéristiques d'un « bon » scénario. Maintenant que la définition d'un scénario est plus contrôlable grâce aux caractéristiques expliquées ci-dessus, il faut proposer des caractéristiques pour contrôler la qualité du scénario lui-même.

Le but de ce concept de « bon » scénario est de fournir un moyen indépendant des scénarios et de la manière de les obtenir pour contrôler leur qualité. Ce point est crucial lors de la décomposition car une mauvaise décomposition ou une décomposition de mauvaise qualité va conduire à une architecture qui ne répondra pas aux attributs de qualité demandés. Par exemple, un scénario ambigu va conduire à une architecture qui ne répondra peut-être pas à la vraie demande puisque les différents intervenants vont l'interpréter de manière différente. De plus, lorsque l'erreur sera détectée, le coût de la modification peut être élevé.

Notre proposition est de définir des caractéristiques auxquelles un scénario doit se conformer pour être de bonne qualité. Ces caractéristiques proviennent des travaux réalisés pour les exigences logicielles. En effet, la qualité des exigences doit pouvoir être contrôlée. La norme IEEE 830 (IEEE, 1998) parle de la qualité des exigences, mais surtout d'un ensemble d'exigences. Leffingwell & Widrig (2000, page 280) et Firesmith (2003a) ont revu et commenté cette liste.

Le tableau ci-dessous résume les travaux existants et montre les caractéristiques applicables au contexte des attributs de qualité.

Tableau III

Recherche des caractéristiques pour un scénario

Caractéristiques (Leffingwell & Widrig, 2000, page 280)	**Caractéristiques (Firesmith, 2003a)[12]**	**Scénario**	**Proposition**
Correct		Oui	Correct
Unambiguous	**Lack of Ambiguity**	Oui	Non ambigu
Complete	**Completeness**	Cela pourrait être applicable à un ensemble de scénarios.	Complet. Mais partiellement applicable, car la phase de définition peut ne considérer que certains scénarios.
Consistent	**Consistency**	Non	
Ranked for importance and/or stability	**Metadata**	Oui	Ordonnable (en utilisant certains critères à définir).
Verifiable	**Verifiability**	Oui	Vérifiable
Modifiable		Non	
Traceable		Oui	Traçable
Understandable		Oui	Compréhensible
	Cohesiveness		Représente un sous-ensemble de non ambigu.
	Correctness		Représente un sous-ensemble de non ambigu.
	Currency	Oui	
	Customer/User Orientation		Représente un sous-ensemble de non ambigu, tourné vers l'utilisateur.
	External Observability	Non	
	Feasibility	Oui	Faisable

[12] Dans l'article de référence, Firesmith ne donne pas de définitions précises mais plutôt une liste de questions pour chaque caractéristique. Donc, la correspondance entre IEEE 830 et Firesmith est basée sur notre interprétation uniquement.

Caractéristiques (Leffingwell & Widrig, 2000, page 280)	Caractéristiques (Firesmith, 2003a)[12]	Scénario	Proposition
	Mandatory		Représente un sous-ensemble de ordonnable (avec un critère dédié).
	Relevance		Représente un autre point de vue similaire à correct.
	Usability	Non. Firesmith mélange deux idées qui sont : 1) compréhensible par tous les intervenants qui est déjà considéré; 2) utilisable par tous les intervenants qui n'est pas pertinent pour un scénario.	
	Validatability		Représente un sous-ensemble soit correct, soit traçable.

Le tableau ci-dessus donne toutes les caractéristiques trouvées dans la littérature et propose de les considérer ou non dans le cas d'un scénario. Évidement, ces caractéristiques proviennent d'un contexte différent ce qui impose de revoir leurs définitions.

De plus, le tableau ci-dessous propose de nouvelles définitions basées sur celles disponibles dans la littérature.

Tableau IV

Définitions pour les caractéristiques d'un scénario

Caractéristiques	Définitions disponibles
Correct	IEEE 830 (1998), la définition est « *An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet* ».
Non ambigu	IEEE 830 (1998), la définition est « *An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation* ».
Ordonnable	IEEE 830 (1998), la définition pour « *Ranked for Importance and/or Stability* » est « *An SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate the importance and stability of that particular requirement* ». Basée sur le gabarit SRS de Rational, deux autres aspects sont ajoutés : • Bénéfice, ce qui correspond aux bénéfices espérés si l'exigence est implémentée; • Risque, ce qui correspond aux risques associés à cette exigence.
Vérifiable	IEEE 830 (1998), la définition est « *An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable is, and only if, there exists some finite cost-effective process with which a person, machine can check that the software product meets the requirement* ».
Traçable	IEEE 830 (1998), la définition est « *An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation* ».
Faisable	Pas de définition disponible mais Firesmith mentionne « *Requirements are of no value if the development team cannot implement them* ».
Compréhensible	Leffingwell & Widrig (2000, page 280), la définition est « *A requirement set is understandable if both the user and the developer communities are able to fully comprehend the individual requirements and the aggregate functionality implied by the set* ».
Simple	Venant des méthodes agiles (en particulier XP (Beck, 2002)), il nous apparaît que la simplicité est une des caractéristiques importantes pour de nombreux artéfacts.

Maintenant, nous allons proposer des définitions pour toutes les caractéristiques sélectionnées.

Tableau V

Caractéristiques pour un scénario

Caractéristiques	Proposition
Correct	Un scénario est correct si et seulement si, il est relié directement ou indirectement à au moins un besoin identifié.
Non ambigu	Un scénario est non ambigu si et seulement si, il n'a qu'une seule interprétation possible.
Ordonnable	Un scénario est ordonnable si et seulement si, il possède au moins les critères suivants : l'importance, le risque et la stabilité et qu'il est possible d'assigner une valeur objective à chaque critère. L'importance concerne l'importance du scénario par rapport aux autres, suivant les recommandations des intervenants. Le risque réfère aux risques encourus face à un scénario complexe. Enfin, la stabilité représente la confiance des intervenants par rapport à la stabilité de la définition et de la nécessité du scénario.
Vérifiable	Un scénario est vérifiable si et seulement si, il existe un processus fini à un coût raisonnable avec lequel on peut le vérifier.
Traçable	Un scénario est traçable si et seulement si, son origine est claire et si le référencement est facile dans de nouveaux développements ou de nouvelles améliorations.
Faisable	Un scénario est faisable si et seulement si, il existe un processus fini à un coût raisonnable avec lequel on peut le réaliser.
Compréhensible	Un scénario est compréhensible si et seulement si, tous les intervenants sont capables de le comprendre.
Simple	Un scénario est simple si et seulement si, il n'existe aucune information superflue ou dupliquée (librement adapté de (Beck, 2002, pp. 137-139)).

Pour conclure cette section, nous tenons à signaler que les caractéristiques sélectionnées représentent un minimum acceptable mais, l'architecture peut très bien en ajouter de nouvelles pour mieux refléter son environnement, sans briser le principe de contrôle de la qualité. Toutefois, il faut faire attention à ne pas vouloir trop bien faire (ajouter beaucoup de nouvelles caractéristiques) car cela risque de rendre le contrôle de la qualité trop contraignant ou trop complexe.

6.1.2 Concept de paire

Dans notre proposition, une paire est la plus petite entité contrôlable qui peut être agrégée et validée. Elle peut être agrégée dans un ensemble cohérent tel qu'un document par exemple. Elle peut être validée par une métrique ou un autre scénario. Ainsi, une paire est contrôlable en tout temps. Une définition sommaire pour une paire est : une paire est une entité constituée d'un scénario et d'une validation qui peut également être un scénario ou une métrique. Une définition plus précise sera donnée plus loin dans cette section.

Le but du concept est de matérialiser la décomposition finale d'un attribut de qualité de haut niveau. Ainsi, l'architecte a une liste de paires qui reflètent chacune un aspect très spécialisé. Ainsi, ce concept permet d'introduire un contrôle de qualité basé sur certaines caractéristiques détaillées dans le tableau ci-dessous.

Dans un premier temps, nous avons sélectionné les caractéristiques les plus importantes pour les paires en nous basant sur le travail de la section 6.1.1 sur les caractéristiques nécessaires pour définir un « bon » scénario. Cela signifie qu'une paire doit absolument respecter ces caractéristiques. Toutefois, un contexte particulier pourrait ajouter d'autres caractéristiques sans briser le concept initial.

Tableau VI

Caractéristiques pour une paire

Caractéristiques	**Proposition**
Correcte	Une paire est correcte si et seulement si, elle est reliée directement ou indirectement à au moins un besoin identifié. Remarque 1 : Cette caractéristique dérive de celle du scénario. Si le scénario est correct alors la paire le sera obligatoirement.

Caractéristiques	Proposition
Non ambiguë	Une paire est non ambiguë si et seulement si, elle n'a qu'une seule interprétation possible. Remarque 1 : Cette caractéristique dérive partiellement de celle du scénario. Si le scénario est ambigu alors la paire le sera aussi. Mais si le scénario est non ambigu, la paire peut l'être dépendamment du statut de la validation.
Ordonnable	Une paire est ordonnable si et seulement si, elle possède au moins les critères suivants : l'importance (qui se base sur celle du scénario la constituant), le risque (qui se base sur celui du scénario la constituant), et la stabilité (qui se base sur celle du scénario la constituant, plus sa propre stabilité), et qu'il est possible d'assigner une valeur objective à chaque critère.
Vérifiable	Une paire est vérifiable si et seulement si, il existe un processus fini à un coût raisonnable avec lequel on peut la vérifier. Remarque 1 : Cette caractéristique dérive de celle du scénario. Si le scénario est vérifiable alors la paire le sera obligatoirement. En effet, la validation est là pour permettre cette vérification.
Traçable	Une paire est traçable si elle seulement si, son origine est claire et si le référencement est facile dans de nouveaux développements ou de nouvelles améliorations. Remarque 1 : Cette caractéristique dérive de celle du scénario. Si le scénario est traçable alors la paire le sera obligatoirement.
Faisable	Une paire est faisable si et seulement si, il existe un processus fini à un coût raisonnable avec lequel on peut la réaliser. Remarque 1 : Cette caractéristique découle partiellement de celle du scénario. Si le scénario est ambigu alors la paire le sera obligatoirement. Mais si le scénario est non ambigu, la paire peut l'être dépendamment du statut de la validation.
Compréhensible	Une paire est compréhensible si et seulement si, tous les intervenants sont capable de la comprendre. Remarque 1 : Cette caractéristique dérive partiellement de celle du scénario. Si le scénario est incompréhensible alors la paire le sera aussi. Mais si le scénario est compréhensible, la paire peut l'être dépendamment du statut de la validation.

Caractéristiques	Proposition
Simple	Une paire est simple si et seulement si, il n'existe aucune information superflue ou dupliquée. Remarque 1 : Cette caractéristique dérive partiellement de celle du scénario. Si le scénario n'est pas simple alors la paire ne le sera pas. Mais si le scénario est simple, la paire peut l'être ou pas dépendamment du statut de la validation.

Tableau VII

Définitions pour les caractéristiques d'un ensemble de paires

Caractéristiques	Définition disponible
Cohérent	IEEE 830 (1998), la définition est « *An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict* ».

Tableau VIII

Caractéristiques pour un ensemble de paires

Caractéristiques	Proposition
Cohérent	Un ensemble de paires est cohérent si et seulement si, aucun sous-ensemble n'entre en conflit avec un autre.

Pour conclure, il faut donner une définition précise d'une paire. La définition proposée est la suivante :

Une paire est une abstraction conçue pour la conception d'architecture logicielle. Elle est composée d'un scénario et d'une validation (il s'agit d'une métrique ou d'un autre scénario) et elle doit respecter les critères suivants :

- l'abstraction doit être instanciée pour un cas précis;
- l'abstraction n'existe que dans le domaine de la conception d'architecture logicielle;

- l'instanciation doit répondre aux caractéristiques suivantes : correcte, non ambiguë, ordonnable, vérifiable, traçable, faisable, compréhensible et simple.

L'abstraction est décomposable et composable de toutes les manières. Le résultat de la composition doit respecter les critères suivants :

- un ensemble d'instanciations doit respecter les critères de ses constituants;
- un ensemble d'instanciations doit être cohérent.

Un des points les plus importants est le fait d'avoir une paire vérifiable par un utilisateur pour un produit logiciel final. Ainsi, la validation est le moyen trouvé pour s'assurer que la caractéristique est bien prise en compte. L'architecte ne peut pas oublier ce point car il doit fournir un moyen de valider dans le produit final le scénario proposé.

Il est intéressant de souligner que le fait de définir une entité de base pour l'étude des attributs de qualité permet de générer un modèle informatique des artéfacts. La documentation d'un attribut de qualité est en fait un ensemble de paires. Donc, il serait envisageable de développer un outil pour garder l'information des paires, de générer automatiquement tous les artéfacts et de contrôler automatiquement la qualité des artéfacts. D'ailleurs, il existe déjà un outil (Thiel, Hein, & Engelhardt, 2003) qui représente l'arbre d'utilité de l'ATAM pour l'évaluation d'une architecture logicielle. Cette perspective est donc envisageable.

6.1.3 Concept de spécification architecturale

L'un des problèmes majeurs dans la conception d'une architecture logicielle est de bien prendre en compte tous les scénarios (du moins ceux qui ont été sélectionnés), car il arrive souvent que les attributs de qualité et, par conséquent, les scénarios soient en contradiction. C'est pour cette raison que les architectes sont très souvent des experts car

ils sont plus capables de trouver des solutions acceptables en faisant les bons compromis.

Dans le but de diminuer le risque lié aux compromis, nous proposons d'introduire une étape intermédiaire qui permet de mieux formaliser les spécifications applicables à la future architecture. Cette approche est basée sur le principe de « diviser pour conquérir ». Pour ce contexte, cela consiste à décomposer le problème en plusieurs sous-problèmes plus spécifiques pour ensuite étudier l'ensemble des sous-problèmes un à un. Ainsi, une spécification architecturale représente une partie atomique du problème plus facilement compréhensible. Une spécification s'intéresse à un seul aspect à la fois et fournit des stratégies pour y répondre. La paire sera utilisée comme concept de base.

Pour répondre à cette séparation, nous proposons de créer une entité dédiée à un problème concret lié à l'architecture logicielle. Ainsi, une spécification architecturale est cette entité qui est composée d'une paire avec une stratégie. Les stratégies sont des solutions applicables pour implémenter un scénario précis. Cela peut être : 1) des patrons architecturaux (Schimdt et al., 2000) tels que l'agent (« broker »); 2) des styles (Bass et al., 1998; Buschmann, Meunier, Rohnert, Sommerland, & Stal, 1996) tels que le multiniveau, le tableau noir (« blackboard »); et enfin 3) des tactiques (Bass et al., 2003, chapitre 5). Mais il faut faire attention de ne pas confondre une stratégie avec les patrons de conception. Toutefois, il faut prendre en compte les incompatibilités entre scénarios et entre patrons et styles. Donc, les spécifications doivent aussi documenter les limitations (par exemple, un impact négatif sur un autre attribut de qualité) et / ou sur les stratégies à proscrire (par exemple, un style impose de ne pas utiliser un autre style).

En plus, cette entité permet d'ajouter une étape supplémentaire ce qui permet :

1. de diviser le problème avec une étape intermédiaire (encore le principe de « diviser pour conquérir »);

2. de mieux documenter les décisions prises pour la conception d'une architecture logicielle;
3. d'améliorer la traçabilité entre un scénario et une décision architecturale. Une stratégie est toujours associée à une paire (et par conséquent à un scénario). Ainsi, l'équipe de développement peut toujours retrouver l'origine d'une décision.

La question de la qualité d'une spécification est aussi à considérer. Nous proposons de réutiliser les caractéristiques de la paire dans ce nouveau contexte.

En résumé, une spécification architecturale est composée d'une paire, d'une stratégie et optionnellement d'une liste de stratégies à proscrire.

Le tableau ci-dessous utilise les mêmes sources que le tableau VI pour définir les caractéristiques applicables à une spécification architecturale.

Tableau IX

Caractéristiques pour une spécification architecturale

Caractéristiques	**Proposition**
Correcte	Une spécification architecturale est correcte si et seulement si, elle est reliée directement ou indirectement à au moins un besoin identifié. Remarque 1 : Cette caractéristique dérive de celle de la paire. Si la paire est correcte alors la spécification architecturale le sera obligatoirement.

Caractéristiques	Proposition
Non ambiguë	Une spécification architecturale est non ambiguë si et seulement si, elle n'a qu'une seule interprétation possible. Remarque 1 : Pour une spécification architecturale, cela ne signifie pas d'avoir une seule solution potentielle, mais plutôt d'éviter les ambiguïtés sur les stratégies. Par exemple, l'utilisation de patrons est une très bonne manière de limiter les ambiguïtés sur les stratégies. Remarque 2 : Cette caractéristique dérive partiellement de celle de la paire. Si la paire est ambiguë alors la spécification architecturale le sera obligatoirement. Mais si la paire n'est pas ambiguë, la spécification peut l'être dépendamment du statut de la stratégie.
Ordonnable	Une spécification architecturale est ordonnable si et seulement si, elle possède au moins les critères suivants : importance, risque, stabilité et degré de conformité minimale, et qu'il est possible d'assigner une valeur objective à chaque critère. L'importance se base sur celle accordée à la paire la constituant. Le risque se base sur celui de la paire la constituant et sur son propre risque. La stabilité se base sur celle de la paire la constituant et sur sa propre stabilité. Enfin, le degré de conformité minimal représente le degré acceptable de conformité. En dessous de ce degré, la spécification architecturale ne serait pas considérée comme présente dans l'architecture logicielle.
Traçable	Une spécification architecturale est traçable si et seulement si, son origine est claire et si le référencement est facile dans de nouveaux développements ou de nouvelles améliorations. Remarque 1 : Cette caractéristique dérive de celle de la paire. Si la paire est traçable alors la spécification architecturale le sera obligatoirement.
Vérifiable	Une spécification architecturale est vérifiable si et seulement si, il existe un processus fini à un coût raisonnable avec lequel on peut la vérifier. Remarque 1 : Cette caractéristique ne dérive pas de celle de la paire.
Actualisée	Une spécification architecturale est actualisée si et seulement si, son contenu reflète les dernières connaissances.

Caractéristiques	Proposition
Faisable	Une spécification architecturale est faisable si et seulement si, il existe un processus fini à un coût raisonnable avec lequel on peut la réaliser. Remarque 1 : Cette caractéristique dérive partiellement de celle de la paire. Si la paire est non faisable alors la spécification architecturale le sera obligatoirement. Mais si la paire est faisable alors la spécification architecturale peut ne pas l'être dépendamment du statut de la stratégie.
Compréhensible	Une spécification architecturale est compréhensible si et seulement si, tous les intervenants sont capable de la comprendre. Remarque 1 : Cette caractéristique dérive partiellement de celle de la paire. Si la paire est incompréhensible alors la spécification architecturale le sera obligatoirement. Mais si la paire est compréhensible alors la spécification architecturale peut ne pas l'être dépendamment du statut de la stratégie.
Simple	Une spécification architecturale est simple si et seulement si, il n'existe aucune information superflue ou dupliquée. Remarque 1 : Cette caractéristique dérive partiellement de celle de la paire. Si la paire n'est pas simple alors la spécification architecturale ne le sera pas. Mais si la paire est simple, la spécification architecturale peut l'être ou pas dépendamment du statut de la stratégie.

Dans le cas précis d'une liste de spécifications architecturales (qui est le résultat obtenu à la fin de la phase 1), il faudrait également s'assurer de sa qualité. Toutefois, la technique utilisée pour une liste de paires ne s'applique pas à ce cas précis car une liste de spécifications architecturales ne forme pas un tout cohérent. En effet, cette liste sera retravaillée dans la phase 2 (voir la section 6.2) pour en supprimer les conflits et en extraire une ou plusieurs propositions d'architecture logicielle.

Maintenant, il faut donner une définition précise d'une spécification architecturale. La définition proposée est :

Une spécification architecturale est une abstraction conçue pour la conception d'architecture logicielle. Elle est composée d'une paire avec une ou plusieurs stratégies et elle doit respecter les critères suivants :

- l'abstraction doit être instanciée pour un cas précis;
- l'abstraction n'existe que dans le domaine de la conception d'architecture logicielle;
- l'instanciation doit répondre aux caractéristiques suivantes : correcte, faisable, traçable, ordonnable, actualisable, non ambiguë et compréhensible.

L'abstraction est décomposable. Toutefois, elle est composable sans que l'ensemble ne représente une entité significative dans ce domaine.

6.1.4 Liens

Avant de considérer en détail l'explication de la méthode (voir la section 6.1.5), il convient de revoir les entités qui seront manipulées et d'étudier en détail leurs liens.

La figure 18 montre une vision haut niveau des relations entre elles.

Comme expliqué précédemment, une paire est constituée d'un scénario et d'une validation et une spécification architecturale est constituée d'une paire et de la stratégie associée au scénario.

Toutefois, il faut également montrer les relations qui existent entre les caractéristiques de toutes ces entités. En effet, les caractéristiques sont souvent très liées à celles des constituants ce qui impose de bien étudier chaque niveau pour comprendre le niveau de qualité du niveau supérieur.

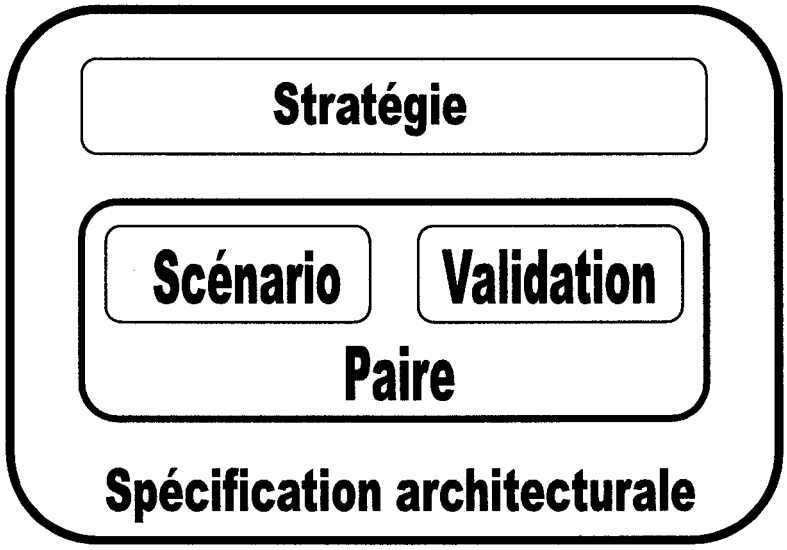

Figure 18 Modèle des entités

La **figure 19** ci-dessous montre les liens de dépendances entre les différentes entités utilisées dans notre méthode et montre les liens entre les caractéristiques de différentes entités. Par exemple, les caractéristiques d'une paire sont dépendantes mais différentes, de celles du scénario la constituant et ainsi de suite.

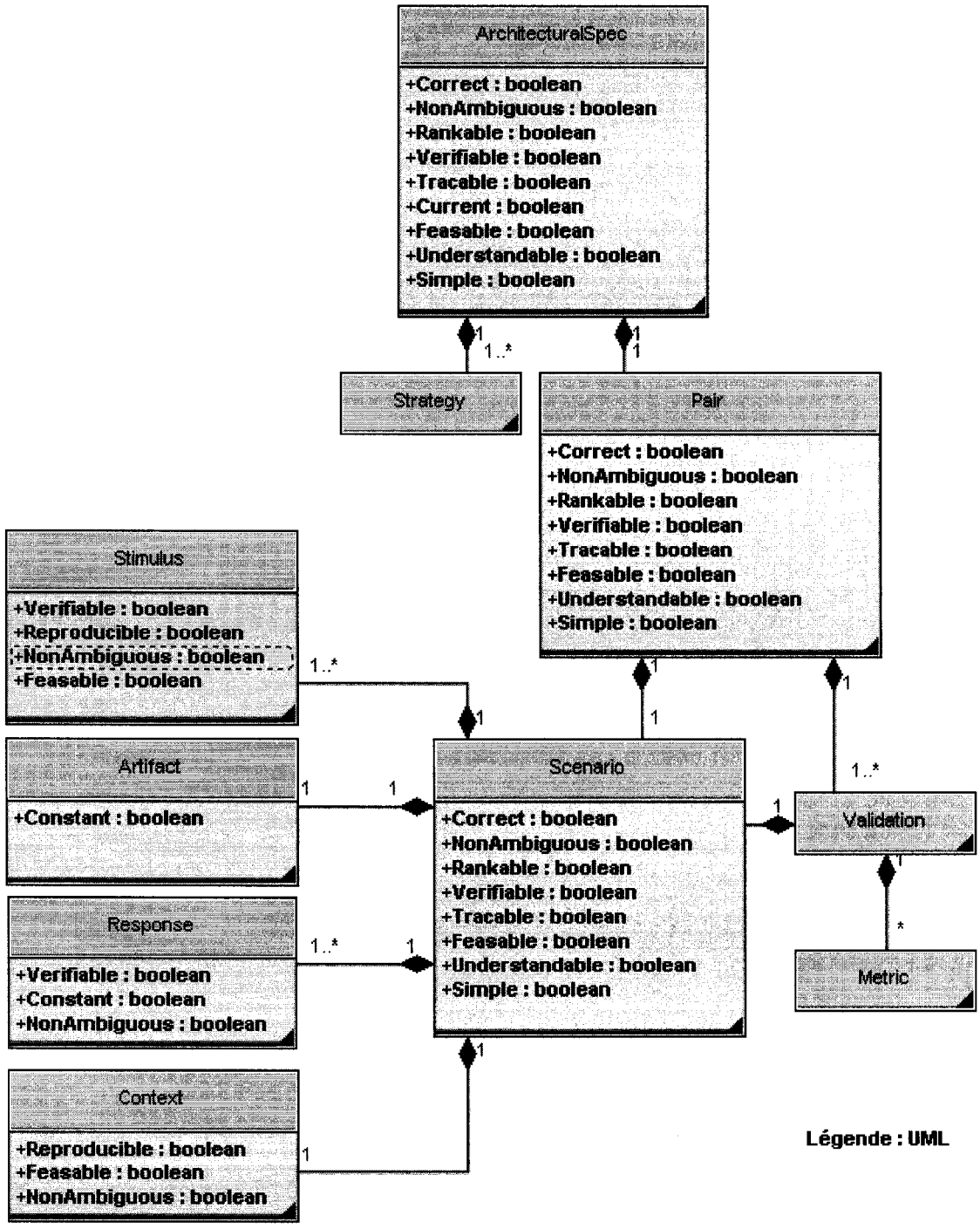

Figure 19 Liens entre les entités

6.1.5 Explication

Avant d'expliquer en détail cette phase, il faut donner le modèle utilisé pour la concevoir. La figure 20 montre que la méthode décompose les attributs de qualité (décomposition grossière) vers des paires (décomposition très fine) pour remonter vers des spécifications architecturales (décomposition fine).

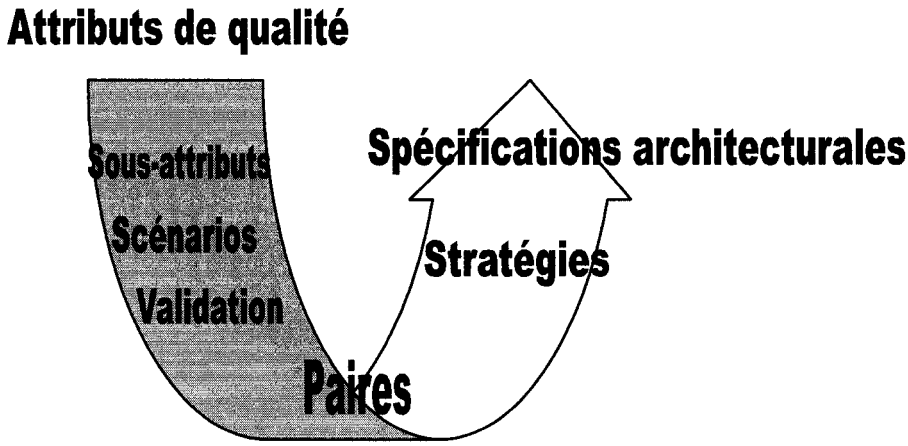

Figure 20 Modèle pour QAAs

La notion de décomposition est très importante car elle sera suivie d'une recomposition dans la phase 2 (agrégation des spécifications architecturales, voir la section 6.2) pour remonter vers une architecture logicielle.

Dans le but de bien expliquer une utilisation possible de cette méthode (en répondant aux deux besoins principaux exprimés), la figure 21 donne le diagramme d'activités. Le contexte est d'extraire les attributs de qualité des systèmes patrimoniaux pour les documenter (ou mettre à jour la documentation existante), de définir de nouveaux attributs de qualité suite à des exigences de mise à jour de l'architecture et enfin de proposer une liste de spécifications architecturales.

Il faut donc que la méthode soit incluse dans une méthode d'analyse d'architectures existantes (ATAM) pour la rétro-ingénierie ainsi que dans une méthode de conception d'architecture logicielle (ADD) pour l'ingénierie (voir le chapitre 7 pour avoir plus de détails sur l'intégration de QAAs avec ATAM et ADD).

La figure 21 donne les différentes activités qui sont :

1. **décrire l'attribut de qualité** pour fournir une description haut niveau de l'attribut à étudier. Cette activité utilise les modèles de qualité proposés pour s'assurer d'utiliser les connaissances existantes. Dans notre contexte, les attributs de qualité sont extraits de la documentation existante ou du code source. Mais, la plupart du temps, la description devrait être disponible dans le document SRS pour la méthodologie RUP ou dans un document résultant d'un processus de rétro-ingénierie;

2. **décomposer l'attribut de qualité** pour fournir la première étape d'une réelle investigation. Cette activité décompose une définition de haut niveau en un ou plusieurs sous-attributs plus spécialisés, eux-mêmes décrits par au moins un scénario. Cette technique de décomposition est la même que celle proposée dans l'ATAM pour l'arbre d'utilité;

3. **définir un scénario** pour expliquer concrètement le comportement espéré pour un sous-attribut. Cette activité peut donner un ou plusieurs scénarios, mais l'architecte doit se concentrer sur les plus critiques en premier. Certains auteurs (Bass et al., 2003; Bosch, 2000) mentionnent que trop de scénarios peuvent entraîner une perte de contrôle du déroulement du projet. D'ailleurs, Bosch (2000) propose l'utilisation de profils et de catégories pour contrôler le nombre de scénarios;

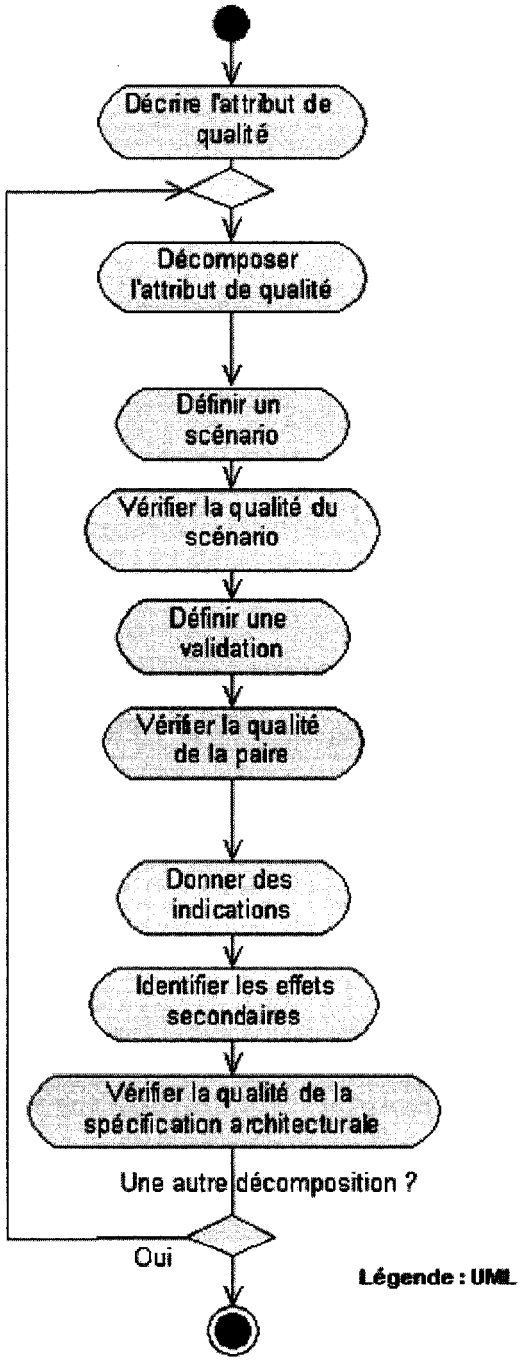

Figure 21 Diagramme d'activités pour QAAs

4. **vérifier la qualité du scénario** qui est l'activité pour : 1) vérifier que tous les constituants du scénario sont présents; 2) vérifier que tous les constituants respectent leurs caractéristiques; et 3) vérifier que le scénario est de bonne qualité (voir la liste des caractéristiques pour les constituants et le scénario à la section 6.1.1);

5. **définir une validation** pour s'assurer que l'architecte peut valider l'implémentation d'un scénario dans le produit logiciel. Cela implique au moins une validation par scénario qui peut être une métrique ou un autre scénario. L'architecte peut utiliser la série de normes ISO/IEC 9126 pour avoir des métriques. Cette activité produit la paire (un scénario et une validation), ce qui est une étape importante de cette phase;

6. **vérifier la qualité de la paire** qui est l'activité pour : 1) vérifier que la paire est de bonne qualité (en se basant sur les caractéristiques développées dans la section 6.1.2); 2) vérifier l'existence ou non d'informations disponibles dans les applications existantes lors d'un processus de rétro-ingénierie;

7. **donner des indications** consiste à explorer les solutions connues (patrons, styles, etc.) pour documenter les stratégies applicables au scénario. Dans le cas d'un processus de réingénierie, cette activité peut collecter les stratégies utilisées et les réutiliser complètement ou partiellement comme solutions possibles pour la nouvelle architecture. Par contre en ingénierie, cette activité doit fournir des stratégies en se basant sur la littérature et sur les connaissances de l'architecte. Attention, cette activité doit définir des stratégies possibles et non une architecture envisagée;

8. **identifier les effets secondaires** consiste à explorer les effets secondaires d'un scénario et d'une stratégie et à les documenter. Un scénario pourrait interdire

l'utilisation d'une stratégie car elle aurait un effet négatif sur celui-ci. Par exemple, un scénario concernant la performance pourrait interdire l'utilisation du style machine virtuelle comme stratégie pour un autre scénario. Un autre exemple serait qu'un scénario requiert l'utilisation du style machine virtuelle. Il serait donc important de signaler que cela a un impact majeur sur la performance (même avant de savoir si la performance fait partie des attributs de qualité considérés). Ainsi, en documentant les effets secondaires, l'architecte montre les zones de conflits qui mèneront à des compromis plus tard (ceci est assimilable à la notion d'identification de risques de l'ATAM (Bass et al., 2003));

9. **vérifier la qualité de la spécification architecturale** qui est l'activité pour vérifier que la ou les spécifications architecturales sont de bonne qualité (en se basant sur les caractéristiques développées dans la section 6.1.3).

Deux documents supportent la méthode QAAs. Le premier document est un gabarit pour aider à la définition et à la documentation d'un attribut de qualité (voir l'annexe 1 pour la version complète). Le deuxième document fournit une aide pour remplir le premier document. Ce dernier document est en fait une liste de questions qui permettent de vérifier les éléments clés du gabarit (voir l'annexe 2).

La description sommaire du gabarit pour les attributs de qualité est :

1. Description : Description d'un attribut de qualité à étudier. L'intention est de fournir une description utilisant les mots du domaine et non ceux du génie logiciel;
2. Raffinement de l'attribut : Définition en détail jusqu'à l'élaboration du scénario. À ce point, l'architecte doit vérifier que les paires se conforment aux caractéristiques de qualité;
3. Top xx : Mise en priorité des paires pour s'assurer que les plus critiques seront étudiées les premières;

4. Liste de paires : Description de toutes les paires obligatoires. Pour chacune, il faut :
    a. Scénario : Description du scénario;
    b. Validation : Description de la validation;
    c. Stratégies : Description des stratégies possibles;
    d. Effets secondaires : Description des effets secondaires sur d'autres attributs de qualité et / ou des stratégies à proscrire.

Le gabarit du document reste intentionnellement de haut niveau pour donner une grande latitude aux utilisateurs afin de mieux définir la manière dont ils veulent représenter les éléments de ce gabarit. Par exemple dans la littérature, il existe plusieurs manières de représenter les scénarios.

6.1.6 Exemple

Dans le but de bien montrer l'utilisation et l'utilité de la méthode QAAs, il est important de fournir un exemple simplifié tiré de notre expérience. Évidemment, il aurait été plus intéressant de fournir un exemple réel tiré de la mise en pratique chez l'entreprise ABC. Toutefois, ceci n'a pu être réalisé dans notre cas.

Dans le but de faciliter la compréhension de cet exemple, la numérotation utilisée suit exactement celle du diagramme d'activités (voir la figure 21 à la page 89).

Suite à la mise en place de la prochaine version pour une application existante, un architecte a reçu le mandat de mettre à jour l'architecture logicielle. Cette mise à jour consiste à rajouter un attribut de qualité déduit d'un problème rencontré par les utilisateurs de cette application.

Le contexte du problème est que le personnel du support technique reçoit des appels d'utilisateurs ayant des problèmes avec l'application. Or, la personne n'a pas un accès direct à l'application et doit donc demander à l'utilisateur d'étudier le problème sous sa supervision. Au niveau du support technique, cette méthode peut être difficile à suivre car il n'y a pas de contact direct entre les intervenants, peu fiable car le client peut avoir mal compris une demande ou avoir oublié de mentionner quelque chose et elle offre une mauvaise perception de l'entreprise auprès de l'utilisateur. Après discussion entre tous les intervenants, l'attribut de qualité à étudier est « la facilité du diagnostic » pour l'application. L'architecte fournit une explication plus détaillée qui est « Le temps de réponse pour diagnostiquer à distance un problème de l'application sur le site du client».

Premièrement, l'architecte utilise le modèle de qualité proposé pour rechercher s'il existe une connaissance semblable (soit cet attribut de qualité existe déjà, soit il existe un attribut semblable). Cette connaissance peut venir de normes, de publications scientifiques ou autres. En tout cas, elle fait partie de la perspective externe (visible à l'extérieur de l'architecture logicielle) ce qui implique qu'il est possible de trouver de l'information pertinente et des stratégies connues et documentées. Pour simplifier l'exemple, nous limitons les recherches à la série de normes ISO/IEC 9126 pour trouver des informations intéressantes. En effet, la sous-caractéristique « Analyzability » ressemble à ce qui est demandé. Cette sous-caractéristique concerne la capacité de pouvoir diagnostiquer un produit logiciel. De plus, la norme fournit des métriques externes telles que « Diagnostic Function Support » et des métriques internes telles que « Activity Recording ». Ces deux métriques correspondent parfaitement au problème. Il faut signaler que la norme fournit des métriques mais en plus les métriques donnent indirectement des solutions techniques partielles au problème. Ce point nous semble très important car il va grandement aider l'architecte. Donc, l'architecte décrit plus précisément cet attribut de qualité comme « l'employé du support technique doit être capable de diagnostiquer à distance une application en fonction sans le support de l'utilisateur et avec le minimum d'impact sur les affaires du client, en moins de cinq (5)

minutes ». La définition est beaucoup plus claire et suffisante pour un client. Toutefois, elle nécessite plus d'études pour en comprendre les impacts sur l'architecture logicielle. Deuxièmement, l'activité de décomposition permet de subdiviser l'attribut de qualité de haut niveau en un ou plusieurs sous-attributs plus spécialisés. Une décomposition possible est :

- diagnostic-ability (adapté de analyzability)
  - "status Management Capability"
    - "history Management Capability" : c'est la capacité de l'application logicielle à supporter la gestion de données d'un historique;
    - "memory Management Capability" : c'est la capacité de l'application logicielle à supporter la gestion de données concernant une image mémoire;
  - "status Diagnosis Response Time"
    - "history Diagnosis Response Time" : c'est le temps de réponse du support technique pour identifier un problème de corruption ou d'erreur au niveau des opérations de l'application logicielle;
    - "memory Diagnosis Response Time" : c'est le temps de réponse du support technique pour identifier un problème de corruption mémoire;

Dans le but de simplifier l'exemple, nous arrêtons la décomposition, mais il semble évident qu'il faudrait aller beaucoup plus loin. Ainsi, l'architecte propose d'ajouter de la fonctionnalité (qui devra être ajoutée et étudiée par des analystes en vue de l'intégration dans l'application logicielle). Mais, les sous-attributs reliés à la mémoire appartiennent à la perspective interne du modèle de qualité car ils sont incompréhensibles et inutiles pour un utilisateur. La conclusion de cette activité est de recommencer la décomposition sur les sous-attributs découverts et de faire des recherches dans la littérature sur tous les sous-attributs (ainsi le modèle de qualité proposé intervient avant l'utilisation de QAAs mais aussi pendant l'activité de décomposition).

Troisièmement, l'architecte se concentre sur la gestion de l'historique pour en extraire le ou les scénarios significatifs (en évitant une profusion de scénarios). Le scénario pour démontrer la capacité de l'application à gérer l'historique est « le personnel du support technique doit être capable de télécharger les données dans un format humainement compréhensible concernant l'historique de l'application en moins d'une (1) seconde avec un réseau en charge normale. Dans ce contexte, une charge normale pour le réseau est définie comme ... et un format compréhensible signifie ... ». Le scénario associé au temps de réponse est « le personnel du support technique doit prendre moins de quatre (4) minutes pour diagnostiquer un problème compréhensible à partir des données de l'historique, sinon il doit prendre moins de deux (2) minutes pour identifier le besoin d'informations supplémentaires. Dans ce contexte, le profil de cette personne est ... ».

Quatrièmement, il faut vérifier la qualité des scénarios produits par l'activité précédente. L'architecte doit donc impérativement vérifier que les quatre constituants sont définis et que chaque constituant répond bien aux caractéristiques qui lui sont associées. Par exemple, l'architecte doit s'assurer que l'environnement du scénario est bien défini et reproductible. Ensuite, l'architecte vérifie que le scénario lui-même respecte ses caractéristiques.

Cinquièmement, l'architecte doit maintenant trouver au moins une validation par scénario. En utilisant les métriques de la série de normes ISO/IEC 9126, il peut adapter les métriques « Failure Analysis Efficiency » de la sous-caractéristique « Analyzability » pour le temps de réponse, et ainsi de suite.

Sixièmement, afin de terminer avec cette phase de définition d'une ou plusieurs paires, l'architecte doit vérifier que les paires identifiées répondent aux critères de bonne qualité. Par exemple, une paire est correcte si elle répond directement ou indirectement à au moins un besoin. De plus, elle est vérifiable si elle offre un moyen à un coût fini et acceptable de vérifier que l'application logicielle délivre les résultats espérés par le

scénario (inclus dans la paire). Dans ce cas simplifié, ces deux caractéristiques sont bien présentes.

Septièmement, l'architecte passe dans la partie de définition de solutions partielles. Ainsi pour cet exemple, les stratégies possibles sont : 1) Avoir au moins un composant (processus ou thread) dédié à la gestion des données de l'historique; 2) Avoir au moins une interface pour gérer l'historique. L'architecte pourrait donc considérer les styles architecturaux « Repository » et « Backboard » (Bass et al., 1998; Buschmann et al., 1996) pour avoir un rafraîchissement continuel des données de l'historique.

Huitièmement, les styles décrits au point six introduisent des pertes de performance causées par ce rafraîchissement continuel et beaucoup d'interrogations concernant la cohérence des données lors de problèmes dans l'application (ce qui constitue justement les cas où le support technique intervient). De plus, ces paires interdisent l'utilisation d'un style de la famille « Data Flow » (Bass et al., 1998; Buschmann et al., 1996), car cette famille de styles pourrait bloquer le flux des données dans l'application si jamais un composant ne fonctionne plus (ou mal).

Neuvièmement, l'architecte dispose d'une liste de spécifications architecturales auxquelles l'architecture logicielle doit répondre. Il doit donc vérifier que chaque spécification architecturale respecte les caractéristiques d'une « bonne » spécification. Par exemple, elle doit être correcte et actualisée ce qui est le cas pour notre exemple.

Finalement, l'architecte a documenté tous les attributs de qualité nécessaires avec les scénarios associés et les solutions partielles. Ainsi, il sait exactement à quoi la future architecture logicielle doit se conformer. De plus, il a identifié la nécessité d'avoir de la fonctionnalité supplémentaire pour l'application logicielle. Il doit maintenant passer à la phase de modifications de l'architecture logicielle.

Pour conclure, cet exemple montre clairement une utilisation possible de la méthode QAAs dans un contexte de réingénierie et aussi le fait que le contexte de l'architecture sera mieux documenté et mieux contrôlé au niveau de sa qualité.

6.1.7 Conclusion

En résumé, cette section propose une méthode pour mieux définir les attributs de qualité. Cette méthode QAAg est basée sur le principe de « diviser pour conquérir ». Ainsi, chaque attribut de qualité est décomposé puis étudié un à un pour fournir des solutions partielles sous forme de spécifications architecturales.

La fin de cette phase doit donc donner une liste de spécifications architecturales qui seront utilisées dans la phase d'agrégation. Cette phase va être expliquée dans la section suivante.

6.2 Phase d'agrégation

Comme nous l'avons expliqué dans la section 5.3.2, cette phase (nommée QAAg pour « **Q**uality **A**ttribute **Agg**regation method ») doit utiliser les spécifications architecturales venant de la phase 1 pour faire la conception d'une (ou plusieurs) proposition(s) d'architecture logicielle. Une spécification architecturale est composée d'une paire avec une liste de stratégies possibles et optionnellement une liste de stratégies à proscrire.

Cette phase n'a pas été étudiée en détail, mais elle doit toutefois respecter certaines étapes indispensables.

Le modèle souhaité pour cette phase est le suivant :

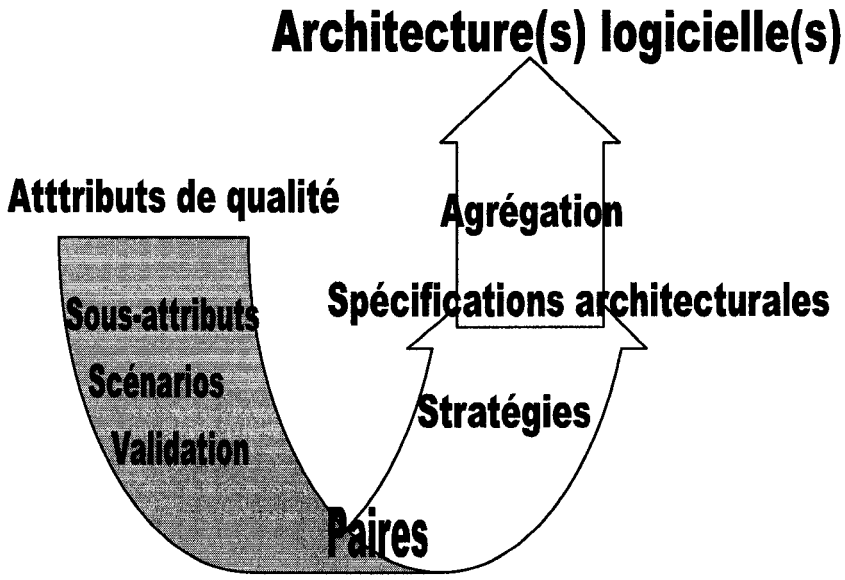

Figure 22 Modèle pour QAAg

Premièrement, nous proposons une étape de prétraitement dédiée à l'étude de la cohérence de la liste de spécifications architecturales fournie comme intrant. Suite à notre réflexion, il semble important de retravailler cette liste qui doit certainement inclure un certain nombre de spécifications identiques et / ou équivalentes (conduisant au manque de cohérence). Deux spécifications identiques représentent deux spécifications qui disent la même chose et qui proposent des stratégies identiques. Deux spécifications similaires représentent deux spécifications qui imposent des stratégies identiques ou qui s'accommodent de la même stratégie, sans toutefois signifier la même chose.

Deuxièmement, nous proposons d'étudier les conflits pour offrir au moins une solution applicable à chaque conflit en expliquant le degré de conformité lié à chaque spécification architecturale faisant partie de cette solution. La résolution de conflit va représenter une étape difficile car il nous semble qu'il faudra accepter une certaine dégradation des spécifications incluses dans la solution. Par dégradation, nous pensons à

un compromis qui ne permettrait pas de réaliser à 100% les spécifications en conflit. Ainsi, il semble important de rajouter ce concept de degré de conformité appliqué aux spécifications. Par exemple, si les spécifications A et B ne peuvent être présentes dans une solution en même temps, il est certainement possible de trouver une autre solution qui offre une conformité de 75% pour A et 50% pour B, ce qui est peut-être acceptable. Une conformité de 50% signifie que l'architecte estime que la spécification B est présente mais seulement à moitié (du fait du compromis). Il suffit de penser à des sous-attributs reliés à la maintenabilité et à la performance pour bien appréhender cette problématique du degré acceptable de conformité, car ces deux attributs de qualité semblent rarement présents à 100% dans une même solution.

Troisièmement, la technique d'agrégation des spécifications entre elles doit être étudiée en détail, car il nous semble qu'elle n'est pas commutative. Ainsi, l'agrégation des spécifications A et B ne donne pas forcément la même solution que l'agrégation de B avec A. Par exemple, l'agrégation de patron agent (« broker ») (Buschmann et al., 1996) et du style multiniveau (Bass et al., 2003) peut vouloir dire que l'agent lui-même est multiniveau. Par contre, l'agrégation du multiniveau et de l'agent peut signifier que les niveaux voisins communiquent entre eux à l'aide d'un agent.

En conclusion, il faut porter une attention importante sur les trois points expliqués ci-dessus pour envisager une solution pour cette phase. En plus, le développement d'un outil va nécessiter de bien les maîtriser car la conception et l'implémentation de l'agrégation et du degré de conformité risquent d'être complexes.

6.3 Phase de validation

Comme nous l'avons expliqué dans la section 5.3.3, cette phase (nommée AV pour « Architecture Validation method ») doit prendre la (ou les) proposition(s) d'architecture logicielle venant de la phase 2 pour en choisir une seule et la valider.

Le modèle souhaité pour cette phase est :

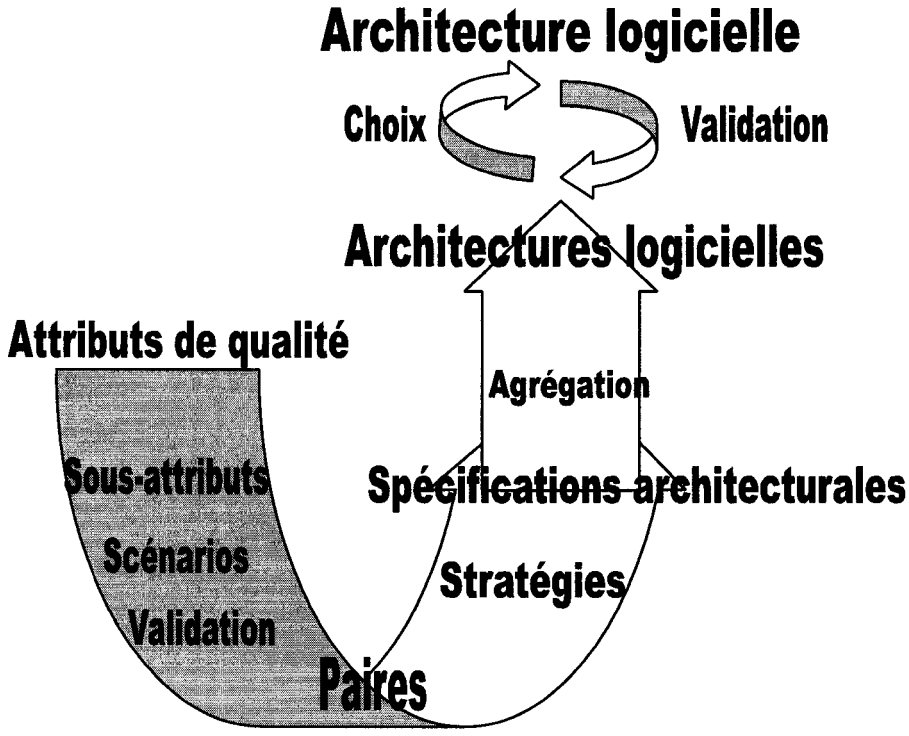

Figure 23 Modèle pour AV

Cette phase doit prendre en compte le contexte d'utilisation de l'architecture logicielle qui se décompose en contexte de développement, de tests et d'utilisation. Évidemment, si ces contextes imposent des contraintes, alors celles-ci auraient dû se transformer complètement ou partiellement en attributs de qualité. Ici, nous parlons plus de contrôler les faiblesses intrinsèques aux contextes. Par exemple, une technologie particulière est peut-être plus ou moins bien adaptée à certains types d'architecture (en particulier, les

technologies liées aux composants distribués telles que .NET[13] et J2EE[14] par rapport à CORBA[15]).

De plus, la littérature offre peut-être des solutions déjà appliquées et bien documentées ce qui permettrait de valider les propositions d'architecture. Si jamais une étude montre les forces et les faiblesses de certaines architectures dans un contexte précis, cela serait très utile pour le choix de la meilleure architecture. Enfin, il existe aussi des méthodes de choix d'une solution par rapport à d'autres. Entre autres, les méthodes Delphi et AHP pourraient aider à choisir la meilleure architecture parmi celles proposées. La méthode Delphi[16] est « a systematic interactive forecasting method based on independent inputs of selected experts ». La méthode AHP[17] est « powerful and flexible decision making process to help people set priorities and make the best decision when both qualitative and quantitative aspects of a decision need to be considered ».

En conclusion, il faut regarder très attentivement cette phase car elle doit permettre de choisir la bonne architecture logicielle et de la valider.

6.4 Récapitulatif

La méthode ADQA est une méthode qui se superpose sur les méthodes existantes (telles que ATAM et ADD) de conception d'une architecture logicielle basée sur des exigences non fonctionnelles. Cette méthode est décomposée en trois (3) phases :

1. **la phase de définition**, qui définit les attributs de qualité pour en extraire des spécifications architecturales;

---

[13] http://msdn.microsoft.com/
[14] http://java.sun.com/j2ee/index.jsp
[15] http://www.omg.org
[16] http://encyclopedia.thefreedictionary.com/Delphi%20method
[17] http://www.expertchoice.com/customerservice/ahp.htm

2. **la phase d'agrégation**, qui agrége les spécifications architecturales pour concevoir une ou plusieurs propositions d'architecture logicielle;
3. **la phase de validation**, qui choisit et valide une architecture logicielle par rapport à son contexte.

Cette recherche s'est limitée à la phase de définition uniquement.

La phase 1 consiste à décomposer les attributs de qualité de haut niveau en sous-attributs plus spécifiques et plus facilement étudiables, pour produire des spécifications architecturales, car nous pensons qu'il est plus facile de trouver des solutions à chaque sous-attribut plutôt qu'à un attribut de haut niveau. La phase 2 agrège les différentes spécifications architecturales pour concevoir une ou plusieurs propositions architecturales, ce qui reste toujours une étape difficile mais plus contrôlable. En effet, les propositions devront respecter toutes ou certaines spécifications architecturales clairement définies ce qui devrait simplifier la recherche de solutions. La phase 3 choisit une proposition en utilisant des méthodes bien connues (par exemple, la méthode Delphi ou la méthode AHP) et la valide par rapport à son contexte spécifique.

Cette méthode apporte des améliorations par rapport aux méthodes existantes dans la littérature. En effet, la transition entre les scénarios et l'architecture logicielle a été décomposée en deux phases (phase 1 et 2) ce qui la simplifie et la rend plus accessible. Ainsi, nous répondons au besoin de diminuer le risque. De plus, nous avons porté beaucoup d'attention sur la qualité des différents artéfacts pour s'assurer de la qualité de la nouvelle architecture logicielle. À notre point de vue, ce point mène aussi à une diminution du risque.

En conclusion, la phase 1 de la méthode devrait apporter une amélioration notable par rapport aux méthodes existantes. En effet, cette phase permet de mieux définir les attributs de qualité et de mieux les documenter, ce qui devrait améliorer la qualité de l'architecture logicielle proposée.

Les aspects originaux de la méthode proposée ont déjà été abondamment expliqués dans le chapitre 6 de ce mémoire. En bref, les points saillants sont :

- définition des constituants d'un scénario (pour des exigences non fonctionnelles) et des caractéristiques indispensables applicables à chaque constituant;
- définition de caractéristiques indispensables à un « bon » scénario;
- définition du concept de paire;
- définition de caractéristiques indispensables à une « bonne » paire;
- définition du concept de spécification architecturale;
- définition de caractéristiques indispensables à une « bonne » spécification architecturale.

Tous ces nouveaux concepts s'assurent de bien mettre au cœur de la méthode ADQA la notion de qualité des artéfacts liée à la conception architecturale en génie logiciel, ce qui est très peu étudié dans la littérature.

De plus, l'introduction d'une étape supplémentaire devrait diminuer le risque lié à la conception d'une architecture logicielle, en introduisant un support supplémentaire (c'est-à-dire les spécifications architecturales) aux méthodes existantes.

En bref, notre méthode devrait simplifier et rendre plus répétable cette phase de conception d'une architecture logicielle. Ainsi, la méthode ADQA s'assure de fournir une approche systématique (complémentaire à celles présentées dans la littérature) de conception architecturale.

CHAPITRE 7

POSITIONNEMENT DE LA MÉTHODE

7.1 ADQA & RUP

Comme mentionné dans les chapitre précédents, la méthodologie RUP ne couvre pas l'analyse des attributs de qualité. Mais, elle mentionne leur existence dans le document SRS. Donc, ADQA (en ajout sur ADD par exemple) devrait combler ce vide et aider un architecte à mieux réfléchir sur son architecture logicielle.

Toutefois, ADQA peut très bien utiliser les documents existants de RUP comme point de départ et comme artéfacts propres. En effet, le document SRS pourrait contenir l'arbre d'utilité de l'ATAM et les paires et le document SAD pourrait contenir les spécifications architecturales.

7.2 ADQA & ADD

Dans le contexte d'ingénierie, la méthode ADD sert de socle pour ADQA. En effet, ADD donne toutes les activités nécessaires pour concevoir une architecture logicielle basée sur les attributs de qualité. En plus, ADD peut utiliser l'arbre d'utilité de ATAM pour décomposer les attributs de qualité.

Toutefois, ADQA fournit une méthode plus détaillée pour passer d'une liste de scénarios vers une architecture logicielle. Ce besoin a été identifié comme une faiblesse dans ADD (comme mentionné en 4.3).

Tableau X

Comparaison entre ADD et ADQA

ADD	ADQA	Remarques
	0 – Modèle de qualité	
	1 – QAAs	
1 – Choisir le module		
2 – Raffiner le module		
2-a – Choisir les scénarios	2 – QAAg	Ces deux étapes de ADD vont certainement faire une agrégation (certainement avec des compromis) entre différentes stratégies pour obtenir une proposition pour l'architecture logicielle. Donc, QAAg sera utile pour faire la transition entre la liste de scénarios sélectionnés et une proposition d'architecture logicielle.
2-b – Choisir les patrons	2 – QAAg	
2-c – Instancier les modules		
2-d – Définir les interfaces		
2-e – Allouer les sous-modules		
3 – Répéter		
	3 – AV	

ADQA est un ajout à ADD pour faciliter la transition entre une liste de scénarios et l'architecture logicielle. En aucun cas, ADQA n'est une méthode complète de conception d'architecture logicielle, car elle ne s'intéresse pas à la décomposition fonctionnelle par exemple.

7.3 ADQA & ATAM

La méthode ATAM peut servir de socle pour ADQA. En effet, ces deux méthodes conjuguées devraient améliorer la documentation (ou la mise à jour de la documentation existante) concernant les paires et les spécifications architecturales. Le seul point à

souligner est que les spécifications ne donneront que la stratégie utilisée pour une paire spécifique et ne pourront pas donner les raisons de ce choix et les autres stratégies possibles comme ADQA le propose. Suivant le contexte, cela pourrait être un point à considérer.

Pour bien comprendre l'intégration de ADQA dans ATAM, le tableau ci-dessous montre une intégration possible :

Tableau XI

Comparaison entre ATAM et ADQA

ATAM	**ADQA**	**Remarques**
	0 – Modèle de qualité	Le modèle de qualité proposé est utilisé en amont de QAAs. Mais il va aussi être beaucoup utilisé durant les activités (voir figure 21, page 89) de décomposition d'un attribut de qualité.
1 – Présenter ATAM		
2 – Présenter conducteurs		Les étapes de 2 à 4 de ATAM vont pouvoir utiliser le modèle de qualité proposé comme support. Cela devrait permettre de mieux identifier les attributs de qualité.
3 – Présenter architecture		
4 – Identifier approches		
5 – Produire arbre	1- QAAs	Les étapes de 5 à 9 de ATAM vont servir à remplir l'artéfact proposé par QAAs, ce qui permettra de contrôler la qualité des artéfacts.
6 – Analyser approches	1- QAAs	
7 – Réfléchir scénarios	1- QAAs	
8 – Analyser approches	1- QAAs	
9 – Présenter résultats	1- QAAs	

ATAM	ADQA	Remarques
	2- QAAg	Comme ATAM étudie une proposition d'architecture ou une architecture existante, l'étape 8 pourrait contenir des informations pertinentes pour QAAg. Ces informations concernent les choix et les compromis faits pour avoir une architecture logicielle respectant les scénarios sélectionnés.
	3- AV	Même remarque que pour QAAg. Par exemple, les choix technologiques peuvent aussi imposer des compromis.

ADQA est un complément à ATAM dans ce contexte. Toutefois, ADQA couvre de manière plus approfondie la transition entre les scénarios et l'architecture logicielle (comme mentionné en 5.1) ce qui explique une partie des différences.

Cette comparaison permet de conclure que les deux méthodes, ATAM et ADQA, sont complémentaires dans un contexte d'ingénierie ou de réingénierie et que les méthodes ATAM et QAAs sont aussi très complémentaires dans un contexte de rétro ingénierie.

7.4 ADQA & contexte de notre partenaire

Pour l'entreprise ABC, le contexte est un mélange de rétro-ingénierie, de réingénierie et finalement d'ingénierie. La rétro-ingénierie est nécessaire pour documenter les architectures existantes (pour extraire les attributs de qualité et les stratégies utilisés), de réingénierie pour repenser les stratégies utilisées pour les architectures existantes (pour les réutiliser ou non dans la nouvelle architecture) et finalement d'ingénierie pour concevoir la nouvelle architecture (en utilisant certaines stratégies et de nouvelles liées aux nouveaux attributs de qualité).

Donc, une méthode possible serait à l'intérieur de RUP d'utiliser un mélange de ATAM et de ADD dans le but de couvrir tous leurs besoins.
La proposition pourrait être :

1. rétro-ingénierie pour documenter les systèmes patrimoniaux (en utilisant ATAM & QAAs);
2. ingénierie pour définir et documenter les nouveaux attributs de qualité (QAAs);
3. réingénierie pour documenter les spécifications architecturales (QAAs) venant de l'étape 1;
4. ingénierie pour documenter les spécifications architecturales (QAAs) venant de l'étape 2;
5. utilisation de QAAg supportée par ADD pour produire une (ou plusieurs) proposition(s) d'architecture logicielle;
6. utilisation de AV pour choisir et valider une architecture logicielle.

En conclusion, notre contexte spécifique devrait être bien soutenu par les méthodes ADQA, ATAM et ADD.

Ceci est en dehors des demandes liées à cette recherche qui se concentre sur les attributs de qualité en proposant la méthode ADQA. Mais, nous jugeons important de fournir une base pour son utilisation dans un contexte réel.

7.5 Conclusion

Ce chapitre a montré l'intégration de ADQA (et plus particulièrement de QAAs) dans les différentes méthodes existantes. Cette intégration était un point important pour cette recherche car le contexte d'utilisation était en voie de définition bien avant le début de cette étude.

RUP n'offrant que peu de support pour l'étude des exigences logicielles, la méthode ADQA doit être utilisée au complet en complément avec les méthodes ADD et ATAM.

CONCLUSION

Ce mémoire propose une méthode systématique d'étude des attributs de qualité pour la conception d'une architecture logicielle qui devrait répondre aux attentes.

En effet, cette méthode devrait améliorer les méthodes existantes car elle propose : 1) de mieux contrôler la qualité des artéfacts; 2) d'introduire une étape intermédiaire entre l'élaboration des scénarios et l'analyse de l'architecture logicielle. Le contrôle de la qualité est réalisé grâce à des caractéristiques auxquelles les différentes artéfacts proposés doivent se conformer. Ainsi, l'architecte peut s'appuyer sur ces caractéristiques pour vérifier si l'artéfact est complet et bien défini. L'étape intermédiaire devrait permettre de mieux comprendre toutes les spécifications applicables à l'architecture logicielle en développement. Cette étape devrait donc simplifier un peu la transition entre les scénarios et l'architecture logicielle.

En bref, notre méthode, en complément avec les méthodes existantes, devrait répondre aux critères définis, qui sont l'amélioration de l'étude des attributs de qualité, la simplification de la transition entre les scénario et l'architecture et par conséquent, la diminution du risque lié à la conception d'une nouvelle architecture logicielle.

RECOMMANDATIONS

Étant donné les contraintes de temps liées à cette recherche, nous n'avons pu répondre en détail à tous les aspects pour finaliser et valider notre méthode.

Ainsi, il ne nous a pas été possible de faire une expérimentation en milieu industriel de la phase 1, ce qui aurait permis de vérifier nos hypothèses dans un contexte réel. Toutefois, il est peut-être possible de faire une partie de cette vérification avec des étudiants suivant le cours sur l'architecture logicielle.

De plus, les phases 2 et 3 sont encore trop incomplètes. En effet, la phase 2 représente toujours un point crucial. L'agrégation des spécifications architecturales reste complexe, car elle doit répondre à trois points critiques qui sont : le prétraitement, l'analyse de conflit et la composition de stratégies. La phase 3 nécessite plus de recherches pour trouver des moyens pour choisir et valider une architecture logicielle parmi plusieurs.

Enfin, un outil devrait permettre d'automatiser certaines parties de notre méthode et de faciliter sa mise en pratique. Cet outil est un atout majeur pour simplifier davantage l'utilisation de cette méthode dans un contexte industriel, car il peut automatiser une bonne partie du contrôle de la qualité et il pourrait automatiser aussi la phase d'agrégation et de validation.

En bref, il reste encore de nombreux champs d'étude à explorer pour finaliser et valider cette méthode.

ANNEXE 1

Gabarit pour la définition d'un attribut

L'annexe 1 est le gabarit pour la définition d'un attribut de qualité. Les deux annexes sont en anglais car l'entreprise ABC souhaite avoir toute la documentation dans cette langue.

1. Goal
2. Glossary
3. Description
4. Quality Refinement

   [This activity provides the decomposition of a high-level quality attribute in one or several sub-qualities. Each of these scenarios plus their validation shall comply to a "good" pair characteristics]

   a. Refinement Hierarchy

      [The architect shall provide a refinement table to justify each selected pair. It could be the Utility Tree from ATAM]

   b. Pair Model

      [If needed, the architect shall highlight dependences between pairs]

5. Top xx

   [As all existing methods mentioned, architects shall not investigate all scenarios but rather only the most valuable ones. The Top xx list is here to indicate which scenarios require immediate attention.]

6. Pairs

   a. Pair Qx.Px

      i. Scenario

         [The scenario describes the behaviour of a specific sub-quality in order to enforce well-understanding of the sub-quality.]

      ii. Validation

         [The validation ensures that a stakeholder is able to validate the success of the scenario implementation in the envisioned software product.]

iii. Proposed Strategies

[This section indicates the possible strategies to handle the specific pair.]

7. Side-Effects

[This section indicates the side effects of a specific strategy promoted for a specific pair on other aspects of architecture. For example, a proposed strategy could decrease the performance, or a specific strategy forbids the use of another strategy.]

ANNEXE 2

Document de support pour la définition d'un attribut

L'annexe 2 est le document complémentaire qui aide à la création du document de définition d'un attribut de qualité (voir l'annexe 1). Le but de ces questions est de fournir un support pour bien comprendre et bien documenter un attribut de qualité.

Ce document devait être amélioré durant la phase d'expérimentation chez l'entreprise ABC. Mais, à court terme, cette phase ne sera pas faite. Donc, ce document ne reflète aucunement les questions concrètes issues de l'expérimentation, mais plutôt notre vision de questions potentielles.

1. **Global Rules**

   Rule 1: All stakeholders shall approve the definition document.

   Rule 2: Each assertion shall be verifiable which means that at least one stakeholder shall be able to corroborate this assertion or that documentation provides an explanation.

   Rule 3: Each assertion shall be correct and traceable which mean that this assertion derived directly or not from at least one documented need and that the path to find this need is documented also.

2. **Template**

   Rule 1: This document is an aggregation of pairs which implies it shall comply with characteristics from a "good" set of pairs.

   Rule 2: The quality attribute under investigation shall: 1) respect the product line vision from ABC; 2) respect the component vision from ABC.

The rules and questions from each section are:

a. Section 3

Rule 1: The description shall use as much as possible domain terminology only (and avoid software terminology).

Rule 2: The description shall not envision or impose complete or partial architectural solutions.

b. Section 4

Rule 1: The refinement is always based on subsequent decompositions with at least one sub branch ended by at least one pair. Each pair shall comply with characteristics for a "good" pair.

Question 1: Is there a scenario at the end of each branch?
If a branch doesn't have any scenario, it could mean that this sub attribute has no impact on the software architecture. However, architect shall double-check this assertion.

Question 2: Is there a validation at the end of each branch?
If a branch ends with no validation associated to a scenario, it means that the quality attribute will never be verifiable (so, the pair doesn't comply with the characteristics for a "good" pair).

Question 3: Are all stakeholders agreed on refinement steps?

Question 4: Are all scenarios mentioned representative of the quality attribute?

If for any reason, the list of scenario is not addressing all important or mandatory aspects of the quality attribute, then the derived software architecture will miss issues. This situation could lead to important reengineering work later in the project.

Question 5: Is there interactions between pairs?

The interactions represent links and potential issues to study in order to understand impacts and perhaps adjustments to perform. This question highlights the need to provide a figure with all pairs and their interactions.

c. Section 5

Rule 1: All stakeholders shall be agreed with the prioritization.

d. Section 6

Rule 1: All pairs are completely defined, no questions are left aside.

Rule 2: Each pair shall have at least one strategy.

Question 1: Are all stakeholders agreed on the detailed scenario?

Question 2: Are all stakeholders agreed on the detailed validation?

Question 3: Are all stakeholders agreed on the detailed strategy(ies)?

Question 4: Are all stakeholders agreed on the detailed side-effect(s)?

3. **Component**

Rule 1: Each pair shall comply with: 1) the component vision. For example, if a specific pair imply to not respect component rules, then the pair or the quality attribute shall be reviewed in order to remove this constraint; 2) rules and constraints derived from component vision, if present. For example, if rules are defined to ease usage of component, then each proposed strategy of all pairs shall respect these rules.

Rule 2: The rule 1 concerning pairs in also applicable to architectural specifications.

4. **Software Architecture**

5. **Product-Line Software Architecture**

Rule 1: Each pair shall comply with: 1) the product-line vision. For example, if a specific pair imply to not use commonalities or to not promote commonalities, then the pair or the quality attribute shall be reviewed in order to remove this constraint; 2) the product-line scope. For example, if a pair imposes to add new rules or constraints not recommended by the product-line scope, then this pair shall be reviewed; 3) rules and constraints derived from the product-line vision, if present. For example, if rules are defined to develop and maintain commonalities between products, then each proposed strategy of all pairs shall respect these rules.

Rule 2: The rule 1 concerning pairs in also applicable to architectural specifications.

BIBLIOGRAPHIE

Anastasopoulos, M., Bayer, J., Flege, O., & Gacek, C. (2000). *A Process for Product Line Architecture Creation and Evaluation, PuLSE-DSSA - Version 2.0* (No. IESE-038.00/E): Institut Expermientelles Software Engineering.

Bass, L., Clements, P., & Kazman, R. (1998). *Software Architecture in Practice* (1st ed.). Indianapolis: Pearson Education, Inc.

Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.). Indianapolis: Pearson Education, Inc.

Bass, L., Klein, M., & Moreno, G. (2001). *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method* (No. CMU/SEI-2001-TR-014): SEI.

Beck, K. (2002). *eXtreme Programming, La référence* (L. Bossarit, Trans. 1 ed.). Paris: CampusPress.

Bosch, J. (2000). *Design & Use of Software Architectures, Adopting and evolving a product-line approach* (1 ed.). New-York: Addison-Wesley.

Briand, L. C., Carrière, J., Kazman, R., & Wüst, J. (1998). *COMPARE: A Comprehensive Framework for Architecture Evaluation* (No. IESE-046.98/E): Einrichtung Experimentelles Software Engineering.

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture, A System of Patterns* (1 ed.). Chichester: John Wiley & Sons.

Clements, P., & Northrop, L. (2001). *Software Product lines, Practices and Patterns*: Addison-Wesley.

Cockburn, A. (2001). *Writing Effective Use Cases* (1 ed.). Upper Saddle River: Addison-Wesley.

Dikel, D. M., Kane, D., & Wilson, J. R. (2001). *Software Architecture, Organizational Principles and Patterns* (1 ed.). Upper Saddle River: Prentice Hall PTR.

Firesmith, D. (2003a, July-August 2003). *Specifying Good Requirements*. Retrieved 4, 2, from http://www.jot.fm/issues/issue_2003_07/column7

Firesmith, D. (2003b, September-October 2003). *Using Quality Models to Engineer Quality Requirements*. Retrieved 5, 2, from http://www.jot.fm/issues/issue_2003_09/column6

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software* (1 ed.). Reading: Addison-Wesley.

IEEE. (1998). *IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications*

IEEE. (2000). *IEEE Std 1471-2000, IEEE Recommended Practices for Architecture Description of Software-Intensive Systems*

ISO/IEC. (1998). *ISO/IEC 14598-1: Information Technology - Software Product Evaluation - Part 1: General Overview*

ISO/IEC. (1999). *ISO/IEC 9126-1: Information Technology - Software product quality - Part 1: Quality Model*

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*: Addison-Wesley.

Larman, G. (2002). *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and the Unified Process* (2th ed.): Prentice Hall PTR.

Leffingwell, D., & Widrig, D. (2000). *Managing Software Requirements, A Unified Approach*: Addison-Wesley.

Len Bass, P. C., Rick Kazman. (2003). *Software Architecture in Practice, Second Edition* (Addison-Wesley ed.): Pearson Education, Inc.

Losavio, F., & al. (2003, March-April 2003). *Quality Characteristics for Software Architecture*. Retrieved 2, 2, from http://www.jot.fm/issues/issue_2003_03/article2

Nord, R. L., & Soni, D. (2003). *Experience with Global Analysis: A Practical Method for Analyzing Factors that influence Software Architectures.* Paper presented at the

Second International SofTware Requirements to Architectures Workshop (STRAW'03), Portland (Oregon).

Paul Clements, R. K., Mark Klein. (2002). *Evaluating Software Architecture, Methods and Case Studies*: Addison-Wesley.

Schimdt, D., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects* (Vol. 2): John Wiley & Sons.

Schmid, K. (2001). *A Framework for Product Line Quality Model Development, The PuLSE-Eco Meta Quality Model* (No. IESE-047.00/E): Institut Expermientelles Software Engineering.

Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component Software, Beyond Object-Oriented Programming* (2nd ed.). New-York: ACM Press.

Thiel, S., Hein, A., & Engelhardt, H. (2003). *Tool Support for Scenario-Based Architecture Evaluation.* Paper presented at the Second International SofTware Requirements to Architectures Workshop (STRAW'03), Portland (Oregon).

Torchiano, M., Jaccheri, L., Sorensen, C.-f., & Wang, A. I. (2002). *COTS Products Characterization*: SEKE'02.