وزارة التعليم العالي و البحث العلمي

BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITE BADJI MOKHTAR-ANNABA

جامعة باجي مختار – عنابـــــــــة

Faculté des Sciences de l'Ingéniorat

Département d'Informatique

THÈSE

Présentée en vue de l'obtention du diplôme de DOCTORAT ès sciences

---

Un modèle de description d'architecture basé service

Discipline

INFORMATIQUE

Par

Noureddine GASMALLAH

DEVANT LE JURY

Président	Mr. Yacine LAFIFI	Professeur	Université 8 Mai 1945	Guelma
Examinateurs	Mr. Allaoua CHAOUI	Professeur	Université Constantine 2	Constantine
	Mme. Nora BOUNOUR	MCA	Université Badji Mokhtar	Annaba
Directeur de thèse	Mr. Abdelkrim AMIRAT	Professeur	Université M.C.Messaadia	Souk-Ahras
Co-encadrant	Mme. Hassina SERIDI	Professeur	Université Badji Mokhtar	Annaba
Invité	Mr. Mourad OUSSALAH	Professeur	Université Nantes Atlantique	France

Année 2019

A ma défunte mère,
« qui a fermé sa vie comme un beau livre de
fées sur les mots les plus tendre que du lilas.

Que dieu lui accorde
toutes ses miséricordes.

A mon père, pour
n'avoir jamais cessé d'éclairer et de
m'accompagner sur le chemin de la réussite.

A toute ma chère
Famille surtout Naim-eddine, Rani,
Abdelmadjid et Jehane.

A mes frères et sœurs.

Remerciements

Mes vives remerciements à mon directeur de thèse Mr Abdelkrim Amirat Professeur à l'université de Souk-Ahras et à ma co-encadrante Mme Seridi Hassina professeur à l'université d'Annaba pour leurs dévouements, rigueurs et précieux conseils qu'ils m'ont dispensés tout le long de cette thèse.

Un remerciement particulier à Mr Mourad Chabane Oussalah Professeur des universités à l'université de Nantes Atlantiques pour l'accueil qu'il m'a réservé au sein de son équipe de recherche Alleos du laboratoire LS2N (ex-LINA) de Nantes (France). Sa patience pour me motiver et sa rigueur pour me faire avancer et son amour à servir et aider les étudiants de son pays étaient des qualités qui m'ont poussé pour finir ce travail.

Je tiens vivement à remercier l'ensemble des membres du jury, et je suis très honoré que la soutenance de thèse soit présidée par notre Professeur Mr Yacine LAFIFI , professeur à l'université de 8 mai 45de Guelma. Je tiens également à remercier le Professeur Mr Allaoua Chaoui professeur à l'université Abdelhamid Mehri Constntine-2 et M^{elle} Nora Bounour Maître de conférences classe A à l'université de Badji Mokhtar Annaba d'avoir examiné tous les deux le travail et accepté de faire partie de ce jury.

Je remercie Mr Imed Boucherika Professeur à l'université de Souk-Ahras pour le soutien qu'il m'a porté durant l'élaboration de la thèse et surtout à rédaction des textes en anglais. Je n'oublierai pas tous les membres du staff administratif qui ont participé de près ou de loin pour que ce travail voie le jour.

En fin, je pense à mes enseignants de tous les paliers scolaires et universitaires qui ont su inculquer en moi l'amour de la science et de la recherche.

Modèle de description d'architecture basé service

Résumé

Depuis plus de deux décennies de recherches, beaucoup d'efforts sont déployés pour traiter l'avènement de la problématique d'évolution en génie logiciel. Le recours à l'abstraction fut une des solutions clé pour réduire la complexité liée à ce problème. En effet, l'architecture logicielle est le niveau d'abstraction le plus favorable pour anticiper la description de l'évolution et répondre aux besoins émergents. La capitalisation du savoir-faire lié à l'évolution nécessite systématiquement des outils d'analyse et de compréhension des techniques déjà réalisées dans le domaine. C'est dans ce cadre que s'inscrit la problématique de notre thèse ayant pour objectif d'identifier, d'analyser et de comparer les méthodes d'évolution des architectures logicielles.

Les contributions de la thèse couvrent l'aspect structurel, comportemental et qualitatif et qui sont structurées en trois réflexions de fond. La première se focalise sur la modélisation d'un référentiel de description d'architecture permettant de capitaliser les caractéristiques liées aux différentes dimensions qui articulent le référentiel proposé. Dans la deuxième, nous proposons une classification dont l'objectif principal est de décrire la couverture actuelle des travaux dans le domaine. Nous proposons dans la dernière contribution, un modèle de description de la qualité décrivant les critères de qualité attendus d'une approche d'évolution. Nos propositions sont appuyées par une étude expérimentale exhibant l'applicabilité du référentiel pour assurer une bonne capitalisation des compétences consenties en évolution architecturale.

Mots-clés: *Architecture logicielle, Evolution d'architecture, Modélisation d'architecture, Méta-modélisation, ADL, Service, Architecture orientée service, Architecture orientée composants et Qualité d'évolution.*

Model for describing architecture based service

Abstract

For over two decades of research, considerable effort is dedicated to addressing evolution issues within the area of software engineering. The use of abstraction was one of the key solutions to reduce the complexity linked to this issue. Indeed, software architecture is the most favorable level of abstraction to anticipate the description of software evolution in order to meet emergent needs. The capitalization of knowledge and know-how linked to the evolution activity requires tools for analyzing and understanding of existing techniques carried out to date. It is in this context that the problem statement of this thesis aims to identify, analyze and compare the methods of evolution. Further, we focus on the main challenge which is to identify a framework for analyzing and comparing architectural evolution methods.

Our contributions cover several aspects of evolution such as structural, behavioral and qualitative. The achievements are structured into three basic reflections. The first focuses on the modeling of an evolution framework describing and addressing characteristics related to various dimensions which articulate the proposed framework. The second results from the exploitation of the first contribution by proposing a classification based on a set of dimensions. This aims to describe the coverage of current research in the field. For the third aspect, we propose a quality description model describing the expected quality criteria from an evolution approach.

We support our proposals with an experimental study illustrating the applicability of the Framework. This attempt to provide ease-of-use of our proposals to capitalize knowledge and know-how linked to a given architectural evolution.

Keywords- Software Architecture, Architecture Evolution, Architecture Modeling, Meta-Modeling, ADL, Service, Service-Oriented Architecture, Component-Oriented Architecture and Quality of evolution.

نموذج لوصف بنية برنامج النظام القائم على الخدمة

الملخص

لقد تم تكريس جهدا كبيرا لمعالجة إشكالية التطور في هندسة البرمجيات، لأكثر من عقدين من الأبحاث، حيث أستخدم التجريد كأحد الحلول الرئيسية للتقليص في حدة تعقيدها. و ففي الواقع، تعتبر عمارة البرمجيات المستوى التجريدي الأكثر ملائمة لوصف تنبؤات التطور لتلبية الاحتياجات الجديدة الناشئة. و تُكَوِنُ عملية رصد جميع الخبرات و المهارات المرتبطة بنشاط التطور أهم أدوات لتحليل وفهم التقنيات المنفذة حتى الآن.

وفي هذا السياق، تومي الدراسة إلى وضع إطار لتحليل و مقارنة أساليب تطور عمارة البرمجيات من خلال تغطية الجوانب الهيكلية و السلوكية و النوعية. وعليه يتحقق الهدف المرجو من خلال وضع:

- نَمْذَجَة إطار مفاهيمي لوصف العمارة بغرض الاستفادة من الخصائص المتعلقة بالأبعاد التقنية المتبعة في عملية التطور،
- اقتراح تصنيفا يهدف أساسا لتغطية الأعمال المنجزة في المجال حتى الآن،
- اقتراح نموذجا للجودة يكشف عن المعايير المَرْجُوَة في عمليات تطور.

كما تم تعزيز كل المساهمات الخاصة بالأفكار الأساسية الثلاث بدراسة تجريبية في المجال.

مفاتيح البحث: بنية البرمجيات ، تطور بنية البرامج ، نمذجة البنية ، النمذجة الفوقية ، اللغة الوصفية للبنية ، الخدمة ، الهندسة الموجهة خدمة ، البنية الموجهة كائن و جودة تطور البرمجيات.

Table des matières

Liste des Tables

Liste des Figures

CHAPITRE 1

Introduction générale

1.1 Cadre général de la thèse

La présente thèse s'inscrit dans le cadre d'une recherche dans le domaine des architectures logicielles. Plus précisément, elle adresse l'aspect évolutif de l'architecture. En effet, les architectures logicielles offrent une vue d'abstraction de plus haut niveau car à ce niveau les spécificités techniques d'implémentation, des niveaux inférieurs, sont reléguées aux niveaux supérieures ce qui offre une vue plus simplifiée du système (Perry, 1992). A ce niveau, la complexité est réduite et le système est traduit en termes d'éléments constitutifs. Nous disons à ce niveau que l'architecture permet de représenter une vue macro du système. A ce stade, l'ingénierie des modèles s'avère un outil propice pour l'anticipation de la prise en charge des besoins émergents ; surtout ceux qui sont plus complexes émanant des mutations qu'impose l'environnement direct et/ou indirect du système logiciel d'où la problématique d'évolution logicielle. Depuis, l'intérêt d'évoluer a émerge comme une des préoccupations les plus importantes dans le domaine de l'ingénierie logicielle motivé principalement par ce qu'elle offre comme avantages en gain de temps et réduction des couts de développement et ou de maintenance de logiciel (M. Oussalah, 2005).

En effet, le système logiciel est construit autours un ensemble de composants qui peuvent subir des actions de maintenance de mise à jour ou d'adaptation. Alors, si le logiciel est assujetti à des éventuelles modifications alors son architecture l'est aussi on parlera donc de l'évolution architecturale. L'évolution à ce stade prend le plein avantage de l'abstraction du système qu'offre l'architecture logicielle. En effet, les architectes peuvent anticiper certaines modifications basées sur des faits ou de l'expertise acquise face aux besoins d'évolution exprimés. Pour ce faire, les architectes sont tenus à identifier les évolutions réutilisables dans un certain contexte bien précis. L'identification des tuples *‹besoin-solution-contexte›* est la solution clé pour intégrer l'évolution au niveau architectural. Selon la littérature, les éléments de l'architecture sont soumis à évoluer sur plusieurs volets :

- Structurel concernant les entités composants le système,
- Comportemental reflétant tous les processus fonctionnels assurant l'interaction fonctionnelle des entités,

- Qualitatif représentant une préoccupation d'amélioration du système.

Quoique les langages de description d'architecture (ADL) représentent la solution clé pour la formalisation des éléments nécessaires à la construction d'architecture, basée service ou composant, par à la fois la description, la définition, le raisonnement sur les éléments, ils définissent aussi les limites techniques du processus d'évolution qui sont liés à l'ADL lui même.

Cependant, la compréhension et l'analyse de la description des architectures logicielles à base de composants ou de services dans son contexte évolutif constituent une thématique prometteuse pour la modélisation des ces architectures. C'est dans ce cadre que s'inscrit le présent manuscrit pour la description de l'architecture logicielle (basée service ou composant) à travers la compréhension du phénomène d'évolution.

1.2 Problématique et objectifs de la thèse

Les systèmes logiciels ne doivent pas rester inertes face aux défis. Autrement, ils risquent de vieillir et devenir obsolètes même s'ils apparaissent opérationnels à l'œil nu de l'utilisateur final. A cet effet, les chercheurs sont conscients des difficultés susceptibles de se produire pendant le cycle de vie d'un logiciel et qui sont essentiellement dues à l'évolution rapide de l'environnement. Cependant, cerner tous les changements qui peuvent être sollicités est l'un des facteurs clés pour doter les systèmes logiciels d'un avantage concurrentiel à travers la réduction du temps d'intervention, les coûts associés ainsi que les efforts de développement (Williams & Carver, 2010). De nos jours, l'évolution du logiciel reste une cause majeure de préoccupation pour les intervenants d'un système. En effet, depuis le brainstorming de Lehman sur les lois de l'évolution, de nombreux efforts sont déployés pour gérer et/ou résoudre les problèmes cruciaux relatifs à l'émergence de besoins multiples d'utilisateurs qui entravent souvent l'utilisabilité et les fonctionnalités du système. Cela est principalement dû aux diverses attentes des clients et des utilisateurs, des nouvelles technologies, des nouvelles tendances en matière d'environnement, etc. L'évolution du logiciel fait référence à la transformation, à l'adaptation à des nouvelles situations et à toute forme de modification impactant le produit logiciel aux phases de la description ou de la modélisation ou du code (Madhavji, Fernandez-Ramil, & Perry, 2006). Il est à noter que l'évolution du code concerne plus de 90% des systèmes logiciels (Brooks Jr, 1995), et qui, s'intéresse exclusivement à la mise à jour des codes en raison de l'apparition de nouveaux besoins. Tandis que l'évolution au niveau de la modélisation décrit les modifications développées à des stades avancés du logiciel. Dans ce manuscrit, l'évolution désigne toutes les méthodes et tous les mécanismes utilisés pour traiter et gérer les contraintes dans plusieurs conditions d'environnement prédéfinies du système considéré. Ici, cette évolution englobe (i) la maintenance du logiciel (ii) la migration des anciens systèmes (iii) la gestion des versions du système et (iv) la reconfiguration et l'adaptation du système.

Sachant que l'architecture logicielle doit évoluer au sein des systèmes de production existants (Jazayeri, 2005), l'évolution pourrait être gérée durant les premières phases de modélisation où les modifications futures pourraient être anticipées beaucoup plus efficacement. Dans cette optique, l'utilisation de l'architecture logicielle offre l'avantage conceptuel d'engager des opérations d'évolution avec l'avantage d'être située à un niveau où la complexité est beaucoup plus réduite. Ainsi, cette anticipation permet d'éviter les modifications tardives qui

peuvent être fatales pour le système. Cela augmenterait également la flexibilité du système face aux menaces croissantes auxquelles il est confronté (Port & Huang, 2003).

Plusieurs études ont explicitement reconnu l'architecture comme un artefact essentiel permettant de réduire la complexité du système. Depuis, plusieurs problématiques ont vu le jour pour inclure le processus d'évolution à savoir: la dégénération du code en termes de modélisation des violations de l'architecturale logicielle (Perry, 1992), la dégénération de l'architecture par la mesure de la déviation de l'architecture pour pouvoir décider de sa reconstruction (Ducasse & Pollet, 2009), vieillissement des systèmes, réparations de systèmes, érosion et dérives (Brooks Jr, 1995; O'Reilly, Morrow, & Bustard, 2003; Parnas, 1994; Perry, 1992), ont tous traité la problématique d'évolution. Cependant, de nombreux changements ont un impact sur l'architecture lors de leurs exécutions. Souvent, ils entraînent des écarts importants entre les deux modélisations initiale et finale, et cela aux différents niveaux de conception. En conséquence, des mécanismes, des référentiels et des méthodes d'évolution spécifiques ont été développés pour faire face aux changements au niveau de la conception afin de préserver la cohérence du système en tant qu'entité unique.

Les architectes ont alors besoin de modèles explicites basés sur des dimensions bien définies, qui offrent à notre connaissance, une bonne opportunité pour comprendre, analyser et positionner une approche donnée de l'évolution d'architecture logicielle. En réalité, les architectes ont besoin de modèles explicites basés sur des métriques bien définies, leur permettant de décrire, d'analyser et de positionner une approche d'évolution donnée s'appuyant sur l'architecture comme une entité de première classe. Sur le plan pratique, l'évaluation de ces dimensions permettra de définir des approches et des mesures particulières des nouvelles techniques possibles. Or, à notre connaissance, la littérature sur l'évolution des logiciels révèle un manque d'études de recherche concernant l'introduction d'un modèle explicite pour positionner une approche donnée sur la base de critères bien définis s'appuyant en premier plan sur l'architecture logicielle.

En outre, malgré les larges consensus enregistrant plusieurs vues au sein d'un modèle de vue architecturale par référence aux vues logiques, d'implémentation, de mise en œuvre, de processus et de déploiement (P. Kruchten, Capilla, & Dueñas, 2009), le modèle proposé est basé sur un point de vu purement architectural se focalisant sur six questions W et H (5Wh), à savoir: Quoi ? Pourquoi ? Où ? Qui ? Quand ? et Comment ? Le but de ce focus est d'identifier les dimensions fondamentales pour la description de l'évolution d'une architecture logicielle. Le modèle proposé fournit la nécessité d'extraire et de gérer les aspects influant l'architecture logicielle.

En résumé, le référentiel de description d'évolution d'architecture logicielle propose une modélisation structurée et cohérente permettant d'examiner les aspects structurels, comportementaux et qualitatifs de l'évolution d'une architecture logicielle. Cela pourrait aussi aider les architectes à prendre en compte des nouvelles décisions concernant une architecture initiale en matière de nouvelles exigences formulées en termes d'opérations dévolution conduisant à une architecture finale. En fait, l'introduction d'un modèle d'évolution clair et concis répondra aux besoins de:

- Identifier certaines spécifications en fonction des préoccupations majeures liées à la nature évolutive de l'architecture logicielle.
- Fournir un vocabulaire et un référentiel conceptuel communs pour comparer et classer les supports offerts par les divers outils et techniques d'évolution d'architecture.

1.3 Contributions

Les contributions de cette thèse sont la résultante de trois objectifs principaux de recherche :

- La définition d'un modèle d'évolution (référentiel d'évolution) pour la description des architectures logicielles permettant de positionner une méthode d'évolution architecturale dans les domaines de l'orienté composant, ou service ou objet,
- La proposition d'une technique de classification se basant sur des dimensions explicites autres que des facteurs techniques identifiées de manière thématique comme la fréquence d'apparition des mots clés,
- La proposition d'un référentiel explorant le volet qualitatif pour permettre d'analyser les critères qualitatifs les plus traités dans le domaine.

Ces objectifs ont formalisés l'espace de notre recherche et ont convergé sur trois contributions corrélées présentées comme suit :

Première contribution: Présentation d'un référentiel pour l'évolution des architectures logicielles (orientée composant, objet ou service). L'approche adoptée se focalise dans la description et l'identification des différentes dimensions à la base de la réponse à un ensemble de questions qui sont nécessaires à la compréhension du processus d'évolution. Ces questions sont inspirées du modèle de Zachman (J. Zachman, 2002) utilisé dans la description des systèmes d'informations dans les entreprises. Le modèle définit les six dimensions d'évolution Quoi ? Pourquoi ? Qui ? Où ? Quand ? et le Comment ? Ces dimensions sont, à notre opinion, nécessaires à la compréhension, analyse et comparaison d'une architecture logicielle. La validation de notre modèle ou référentiel (dans la suite du manuscrit) suscite premièrement une mesure de chaque dimension pour pouvoir la positionner sur le référentiel proposé ce qui décrit l'orientation de cette approche par la considération d'une telle dimension. Cependant, l'étude bibliographique abordée a pratiquement mis en avant des limites concernant :

- La détermination explicite de dimensions pour évaluer une évolution,
- L'adoption d'un cadre général permettant de décrire les orientations adoptées par les travaux courants d'évolution et de statuer sur ce qui reste encore à explorer.

Deuxième contribution: Présentation d'une classification basée sur un sous-ensemble des dimensions parmi celles qui sont proposées. La littérature existante a montré que la majorité d'entre elles épouse la classification thématique c.à.d. des classifications qui proposent des classes selon la problématique abordée. Cependant, la spécificité de la classification proposée est qu'elle se base purement sur des dimensions explicites s'intéressant exclusivement à la thématique de compréhension de la méthode d'évolution à un niveau plus abstrait. Un autre avantage de la taxonomie proposée, de même importance que la première, est la flexibilité de cette classification. En effet, l'utilisation d'un plus grand nombre de dimensions fournit des classes plus fines et vise

versa ce qui permet de répondre à une très grande variété de besoins d'intervenants. De plus, cette classification peut être employée comme un méta-classificateur i.e. un classificateur des plusieurs classifications existantes pour situer ces études en fonction de la classification proposée.

Troisième contribution: Nous sommes partis du fait qu'un modèle d'évolution doit respecter des contraintes de qualités préalables. La présentation d'un modèle qualitatif pour l'évolution est une activité importante permettant d'évaluer l'architecture résultante à la base d'un ensemble de critères qualitatifs. Notre contribution s'articule sur la sélection et le regroupement des critères jugés inéluctables à la qualification de la qualité d'une évolution en termes de capacités attendues, et qui offrent la faculté de:

- S'informer sur l'aspect qualitatif d'une architecture,
- Décider sur des évolutions futures pour bénéficier d'une qualité maximale,
- Comparer les différentes approches d'évolution par rapport à des critères qualitatifs.

Le modèle de qualité propose une vue plus globale (macro) appréhendant les besoins qualitatifs sollicités par l'ensemble des intervenants en ingénierie logicielle (Orientée composants, orientée services et orientée objets). La Figure 1.1 dresse l'articulation globale du travail de recherche effectué. Elle présente une articulation basée sur les trois contributions sus citées, les relations directes entre elles. Le schéma met aussi en relief l'avantage qu'exhibe chacune des contributions avec le domaine d'évolution concerné.

1.4 Organisation du manuscrit

La présente thèse est structurée, en dehors de l'introduction et la conclusion, en quatre principaux chapitres, comme le présente la Figure 1.2.

Chapitre 2- Architecture logicielle : Ce chapitre expose la terminologie associée aux architectures logicielles et cerne aussi les travaux de recherches qui ont encadré ce domaine. Il est scindé en plusieurs sections dont l'objectif est de présenter:

- Le paradigme orienté composant où nous présentons les définitions les plus pertinentes du paradigme d'architecture logicielle ainsi que les différents concepts de base associés.
- Le paradigme orienté service dans lequel nous nous focaliserons sur les travaux basées services où l'architecture logicielle est au cœur de la modélisation ou du développement.
- Le paradigme de style architectural qui présente un des concepts de vogues les plus liées à l'architecture logicielle orientée. Nous illustrons dans cette section les différents patrons de conception et styles architecturaux émergeant dans le domaine pour la modélisation et l'évolution des architectures logicielles.
- Le paradigme vues architecturales, une notion intrinsèque à la perception d'une architecture et qui formalise donc un avant plan de modélisation. Le but ici est d'exposer que la modélisation est soumise à la vision des intervenants qui désirent modéliser et/ou développer une architecture logicielle. Nous définissons alors la notion de vue, ces types et les travaux de recherches qui mettent cette perception en premier plan de modélisation.

Chapitre 3- Evolution et référentiels d'évolution logicielle : Dans ce chapitre, nous introduisons la problématique de l'évolution des architectures et l'étendu des travaux dans cette discipline. Il est organisé de façon à parfaire les objectifs de:

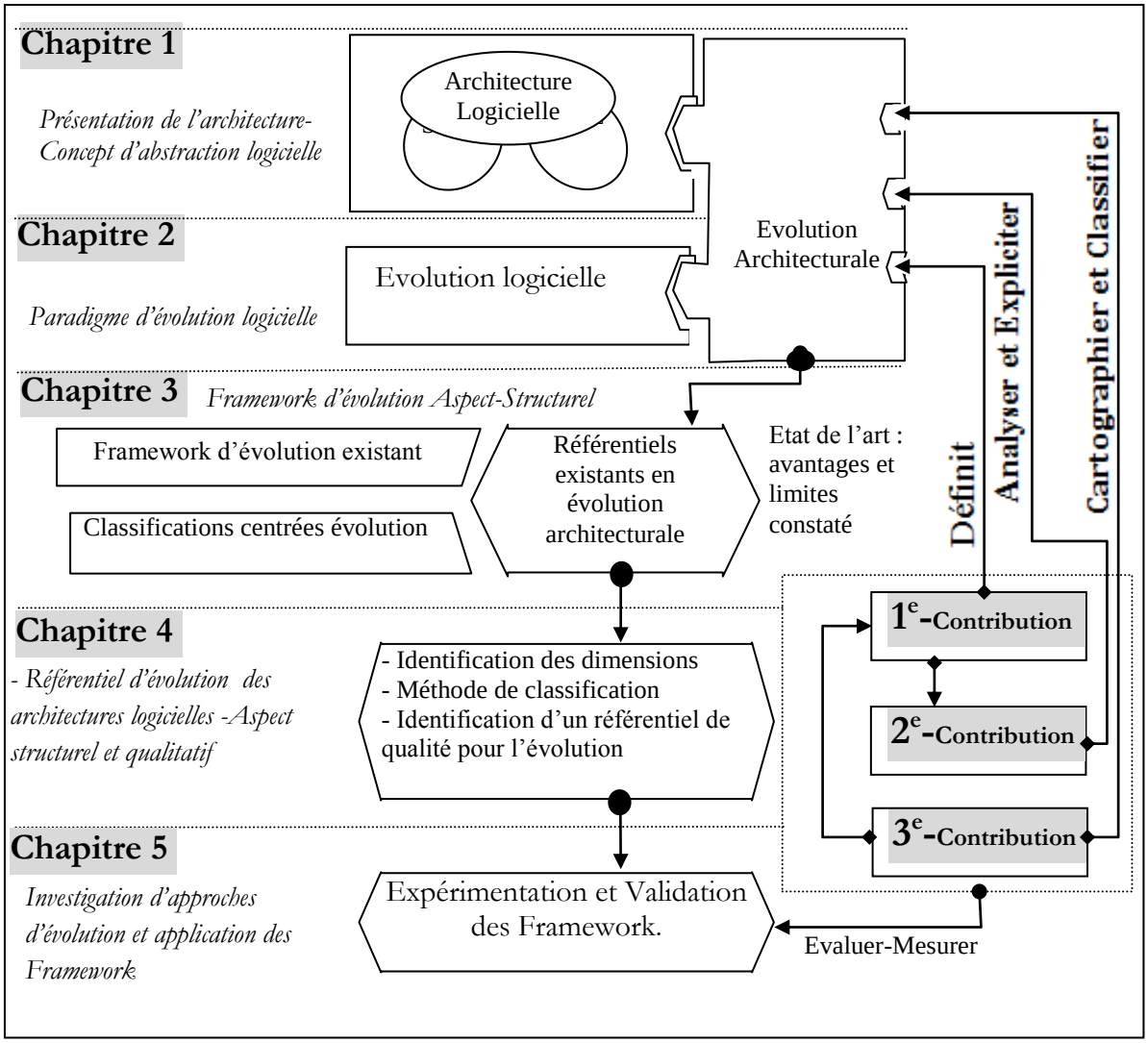

Figure 1.1- Représentation des relations entre chapitres, contributions et contexte.

- Cerner les différents concepts liés à l'évolution considérée elle même comme un nouveau paradigme en ingénierie logicielle.
- Identifier les Framework existants de description d'architecture logicielle qui sont dédiés aux architectures considérées comme systèmes ou logicielles.

Cette étude est fondamentale pour démonter à la fois l'utilité de notre travail et la nécessité de trouver un référentiel commun pour l'analyse et la compréhension de l'évolvabilité de l'architecture.

Chapitre 4- Référentiel d'évolution des architectures logicielles : Ce chapitre met en exergue notre proposition et illustre les avantages du Framework proposée. Bifurqué essentiellement en deux grandes sections concernant la modélisation de la description de l'architecture sous son aspect structurel et comportemental et la deuxième section pour l'aspect qualitatif de la description.

Dans la première, nous proposerons des dimensions composant notre Framework de description d'évolution d'architecture. Nous définissons les différents concepts identifiés et nous présenterons la modélisation de notre référentiel. Les relations inter et intra concepts sont illustrées et un exemple d'illustration servira comme un running exemple tout le long du processus de définition. Dans l'aspect qualitatif, nous introduisons notre référentiel qualitatif pour décider de l'évolution de l'architecture dans son aspect évolutif. Nous présentons le référentiel en termes de capacités de qualité et nous déterminons les fonctions nécessaires à l'évaluation de la qualité. A l'instar de ces critères qualitatifs, l'évolution architecturale peut être cartographiée et illustre ainsi les zones grises et d'ombres dans le domaine.

Chapitre 5- Etude expérimentale et validation du référentiel : Ce chapitre est consacré pour présenter l'application du modèle sur un échantillon d'approches d'évolution. Il est nécessaire d'évaluer une méthode d'évolution par application du Framework proposé autant sur son aspect structurel que qualitatif. Un ensemble de résultats sera exposé et argumenté pour montrer la validité de notre référentiel.

Dans le dernier chapitre, nous concluons notre travail puis nous dressons les lignes directrices envisagées pour des nouvelles perspectives de recherche.

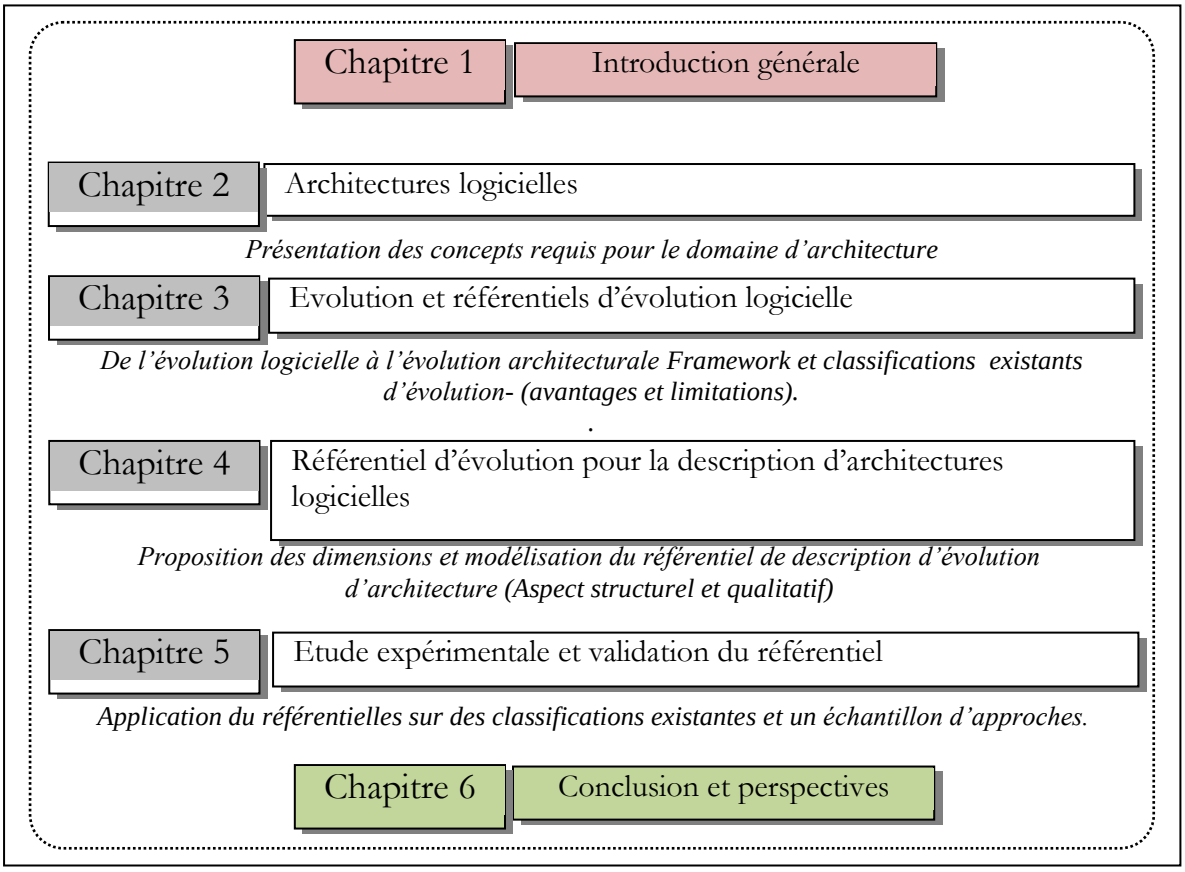

Figure 1.2- Structure du manuscrit et objectifs des chapitres.

CHAPITRE 2

Architecture Logicielle

« La pierre la plus solide d'un édifice est la plus basse de la fondation.»
- Gibran Khalil Gibran (1883-1931)

2.1 Introduction

L'abstraction est un instrument favorable pour universaliser les systèmes et de réduire leurs complexités faisant ainsi émergé le concept d'architecture logicielle qui n'est autre qu'une vue simplifiée de la conception du système. Contrairement à la communauté du développement agile des logiciels (Boehm & Turner, 2003), la communauté du génie logiciel croit fortement que l'architecture est l'une des pistes pertinentes à exploiter pour traiter les problématiques d'ingénierie logicielle à des niveaux plus abstraits. Ainsi, plusieurs recherches se sont focalisées sur l'importance et l'intérêt de ce paradigme au cours de la modélisation des applications informatiques. Plusieurs concepts et éléments de bases ont été développés pour permettre de décrire une bonne architecture et de définir l'ensemble des opérations qu'y peuvent être appliquées donnant ainsi naissance à toute une discipline à part. Néanmoins, une forte divergence, dans les définitions des éléments composants l'architecture fait encore couler beaucoup d'encre entre chercheurs dont l'objectif est de réifier et de standardiser les notions de bases.

Dans ce chapitre, nous exposons la terminologie associée aux architectures logicielles et nous cernerons aussi les travaux de recherches les plus pertinents dans le domaine. Nous se focaliserons à ce titre sur les différentes définitions des éléments constitutifs d'une architecture à base de composant. Puis, nous réservons toute une section à la description de l'architecture orientée service. Une troisième section sera consacré à présenter l'aspect qualitatif d'une architecture. Les modèles de qualité les plus connus seront définis et une présentation de différentes caractéristiques de chacun est illustrée.

2.2 Architecture orientée composant

Cette section est consacrée à la présentation de l'ensemble des concepts de base de l'orienté composant.

2.2.1 Concepts de base

L'architecture a été adoptée dans le génie-logiciel pour refléter un aspect conceptuel très important dans le cycle de vie d'un logiciel. Depuis, plusieurs définitions ont été adoptées dans la littérature pour désigner une architecture logicielle. Nous avons choisi de présenter, à notre connaissance, les plus communes et les plus pertinentes qui positionnent ce qu'il faut savoir sur une architecture sans pour autant dévaloriser ceux qui n'ont pas été évoquées.

2.2.1.1 Architecture

Définition 1: Perry et Wolf décrivent par analogie l'architecture logiciel à l'instar des architectures bâtiments (Perry, 1992) comme:

"*The architecture of a software system is a metaphor, analogous to the architecture of a building*".

Ils identifient un ensemble d'éléments de conception qui sont catégorisés en trois types d'éléments : éléments de calculs, éléments de données et éléments de connexions.

Définition 2: Clements et al. (Clements, 2002) définissent l'architecture logicielle comme des structures minutieusement choisies et conçues par les architectes d'applications pour permettre la réalisation et le raisonnement sur des objectifs fixés par le système. Ces structures sont le fil conducteur à la compréhension de l'architecture.

Il est clair que cette définition se focalise sur la documentation d'une architecture plus que les détails techniques des composants d'une architecture. Les éléments architecturaux sont référencés par les structures où leurs propriétés et les relations inter structures ont été spécifiées pour mieux documenter l'architecture.

Définition 3: Une définition plus technique est donnée par (Bass, Clements, & Kazman, 2003) où l'architecture est vue comme:

"*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them*".

Cette définition décrit l'architecture comme une ou un ensemble de structures du système constitué de différents éléments logiciels et leurs assomptions par référence aux informations et ou caractéristiques sur le détail de ces éléments. Ils formalisent donc la partie visible de la structure à travers les interactions inter structures qui peuvent s'établir. De ce fait, l'architecture est vue de l'extérieur comme une abstraction du système car ces structures sont souvent utilisées pour simplifier la description du système. Il est à noter qu'une bonne architecture doit permettre au système d'accomplir ses fonctionnalités et d'assurer sa cohérence durant tout le cycle de vie. Ce qui soulève l'importance du processus d'évaluation de l'architecture elle-même.

Définition 4: L'architecture est définie selon la norme ANSI (US)/IEEE Std 1471-2000 (Recommended Practice for Architectural Description of Software-Intensive Systems) comme:

"*The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution*"(Hilliard, 1999).

Cette définition soulève de multiple utilisation de la notion d'architecture. En effet, la compréhension d'éléments architecturaux et des contrôles inter éléments sont essentiels pour organiser, spécifier l'utilité et la qualité du système. Les éléments peuvent être des entités physiques (composants physiques) ou logiques (composants logiques).

De toutes ces définitions, nous pouvons dire qu'une architecture logicielle regroupe un ensemble de concepts intrinsèques à sa description et qui présente une vue plus abstraite et donc moins complexe à décrire.

2.2.1.2 Composant

Le composant est considéré comme l'élément architectural essentiel pour la description d'une architecture logicielle ce qui justifie une panoplie de définitions pour ce concept. Nous notons que la définition la plus adoptée en littérature est celle de SZYPERSKI qui définit le composant comme (Szyperski, Bosch, & Weck, 1999):

"*a reusable, deployable piece of software without persistent state*".

C'est une unité de composition ayant des interfaces spécifiées par des contrats et des dépendances explicites avec le contexte. Le composant est une partie de logiciel indépendante déployée sur un tiers pour accomplit un objectif précis tel que la vente, la réutilisabilité, l'intégration et la composition.

Dans la même ligne, le composant et considéré comme un élément déployé indépendamment qui est conforme à un modèle ou à un standard de composition (Heineman & Councill, 2001):

"*a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard*".

Une définition plus articulée en trois caractéristiques est donnée par Lüer et al. (Lüer & Van Der Hoek, 2002):

"*A component is a software element that (a) encapsulates a reusable implementation of functionality, (b) can be composed without modification, and (c) adheres to a component model*".

Cette définition spécifie qu'un composant logiciel est doté d'une structure encapsulant une fonctionnalité métier autonome conforme à un modèle de composant et qui est peut être réutilisable par une tiers pour une éventuelle composition.

Donc et à partir de toutes ces définitions, nous pouvons dire qu'un composant logiciel est un élément logiciel justifiant les caractéristiques de : autonomie, composition, réutilisation, déploiement, utilisation et d'interopérabilité. De ce fait, il est considéré comme un élément architectural à part entière.

Nous précisons que cette définition couvre la notion de composant que ce soit dans le domaine de l'orienté objet tel que COM (Component Object Model) ou le domaine de l'orienté composants CCM (Component CORBA Model). Il est à noter que l'utilisation du concept *composant* s'étale de la phase conception jusqu'à la phase d'exécution du cycle de vie du logiciel. Le *composant* est souvent décrit selon deux aspects:

a. Aspect structurel d'un composant logiciel

Cet aspect formalise l'ensemble des éléments qui permettent d'identifier et de spécifier le composant. Selon (Oussalah, 2005), l'identification est présentée à travers les services requis et fournis du composant et sont appelées les *interfaces des composants*. A moment où les spécifications des caractéristiques indiquent le volet *propriétés du composant* et qui servent essentiellement à fournir les éléments afférent à la conception et à l'analyse du composant.

- *Interfaces :* Aussi appelées services, elles représentent la partie transparente à l'utilisateur où sont définis l'ensemble de fonctionnalités fournies par le composants ou celles qui lui sont requises pour assurer une bonne utilisation. L'interface du composant est positionnée selon deux dimensions (Chefrour & André, 2002). La première est une *dimension sémantique* où toute la sémantique liée aux fonctionnalités requises ou fournis proposées par le composant sont décrites. Le service décrivant le fonctionnement d'un composant réfère à une *interface fournie* alors que la description des entrées nécessaires au service pour s'exécuter réfère à une *interface requise.* La seconde est une *dimension structurelle* à travers laquelle le composant est décrit comme une entité séparée pour identifier le point d'interaction *port* entre le composant et l'environnement de l'évolution du composant. Cette dimension se focalise sur les fonctionnalités qui doivent fournir ou requérir le composant pour qu'il puisse être utilisable.

- *Propriétés.* Sont les spécifications techniques d'un composant et peuvent être répertoriées en deux types:

  - *Propriétés non fonctionnelles*: Se sont des propriétés associées aussi bien à la structure du composant qu'aux fonctionnalités proposées. De ce fait, elles versent essentiellement dans le volet qualitatif du composant par référence à la performance, sécurité, portabilité ...etc. Les caractéristiques des ces propriétés permettent de simuler le fonctionnement préalable d'un composant avant son utilisation effective.

  - *Propriétés fonctionnelles*: Ces propriétés font références aux caractéristiques sémantiques. D'une part, les propriétés peuvent spécifiés des contraintes décrivant l'aspect fonctionnels qui est assuré par la vérification de la cohérence du system i.e. que les propriétés ont été vérifiées. D'autre part, elles spécifient le contrat qui définit l'ensemble des interfaces nécessaires pour l'utilisabilité du composant. Elles englobent les contraintes sur les services fournis et requis du composant. Dans le premier cas, les contrats assurent que les services fournis respectent les contraintes définies. Par contre dans le deuxième cas, les contrats sont définis sur les interfaces requises du composant.

b. Aspect fonctionnel d'un composant logiciel

Cet aspect désigne souvent la structure interne liée au composant. On trouve deux structures de fonctionnalités: *Atomique* ou *composite.*

- *Aspect atomique :* est assigné au composant dont les fonctionnalités sont implémentées via un langage de programmation bien défini. Les composants de tel aspect forment des éléments architecturaux atomiques qui sont souvent sollicités pour des opérations de composition.

- *Aspect composite des fonctionnalités :* cet aspect stipule l'utilisation d'autres structures de composants déjà définis. Ces composant peuvent donc englober une partie atomique i.e. une implémentation propre en plus de l'utilisation d'au moins une structure fonctionnelle d'un autre composant. Logiquement dans un tel cas, le composant composite d'un aspect fonctionnel définit les interactions qui peuvent être entre les composants internes du composant ou seulement entre les composants internes et le composant qui les compose. Un composant peut être ajouté pour jouer le rôle d'un médiateur pour synchroniser les interactions avec le composant hôte. Aussi, l'interaction peut être assurer par délégation des interfaces soit par encapsulation des services des composants internes au profit du composant composite, soit par délégation des interfaces aux composants constitutifs en définissant des ports bien spécifiques (Barnes, Hill, & Busateri, 2015).

Il important ici de mettre en lumière les différentes relations possibles entre les composants d'un composant composite et qui sont réparties en deux grandes catégories: relations horizontales et verticales. Les premières sont les plus simples et qui désignent l'association des composants et des connecteurs de sources différentes et ce sans poser de contraintes particulières entres les éléments associés. Quant aux relations dites verticales, à la différence des relations horizontales, elles imposent, selon le type de relation, des contraintes entre les différentes acteurs du composant composite (Ahmad, Belloir, & Bruel, 2015).

Par exemple : *Les études de (Bigot & Pérez, 2010) et (Ilic, Pratas, Trancoso, & Sousa, 2011) présentent un modèle de composants logiciels hiérarchique (HLCM : High Level Component Model) générique qui permet d'exécuter des composants atomiques en se basant sur d'autre modèle de composants. Il adopte le concept de connecteur pour permettre les interactions de composants composites (Matougui & Beugnard, 2005). L'assemblage concret est assuré par le biais de la technique de la transformation de modèle entre modèle existant de logiciel et le composants/connecteurs HLCM.*

2.2.1.3 Connecteur

Le connecteur est l'un des fondements essentiels de l'une architecture logicielle. A ce titre, il représente à part entière un élément architectural tout comme le composant. Ce qui permettra clairement de distinguer les différentes interactions ou communications entre ces deux éléments. Parmi toutes les définitions présentées dans le domaine, nous citons ici celles qui sont les plus souvent citées dans le domaine d'architecture logicielle:

Définition 1: "*A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components*" (Shaw & Clements, 1996) .

La définition précise que les connecteurs sont des abstraction qui gèrent les interactions entre les différents composants d'un système en terme de spécifications des règles et mécanismes indispensables pour médiatiser la communication, la coordination et la coopération à travers les composants. Ces connecteurs sont modélisés, distribuées et implémentés explicitement dans les composants adéquats du système. Les connecteurs peuvent être simples comme les procédures d'appelles automatiques ou le partage des données d'accès ou complexes comme les protocoles clients/serveur et protocoles d'accès aux bases de données par exemple.

Définition 2: "*Connectors mediate interactions among components: that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required*" (Mary & David, 1996).

Cette définition stipule clairement que le connecteur joue un rôle de médiateur habilité non seulement à établir des règles d'interactions gouvernant tous les acteurs (composants) intervenants mais aussi de spécifier tout mécanisme utile pour la réussite de ce processus.

Des définitions précédentes nous pouvons définir le connecteur comme:

"*Un médiateur abstrait mis en service pour faciliter les mécanismes de communication, coordination, coopération et le contrôle de flot de données échangé entres composants*".

a. Typologie des connecteurs

Les connecteurs se partagent en quatre types en fonction des services qu'ils offrent (Mehta, Medvidovic, & Phadke, 2000).

- *Communication:* Dont la responsabilité est d'effectuer le transfert des données, et d'assurer de ce fait l'interaction entre les différents composants du système. Ces données représentent l'ensemble de messages, de données ou de résultats que les composants offrent ou requièrent pour la bonne exécution du système.

- *Coordination:* Ce type de connecteur est utilisé lorsqu'il s'agit de contrôler un flot de données entre composants. Il supervise l'interaction en matière d'invocation et d'appel de fonctions pour permettre la communication.

- *Conversion:* Ce connecteur est un résultat logique d'interaction entres composants hétérogènes non planifiés à priori pour interagir. Il servira alors à formaliser un service fournit d'un composant fournisseur à un autre format prêt à être consommé par un autre composant. La conversion peut concerner la fréquence, le nom des services et leurs nombres, l'ordre d'interactions ...etc.

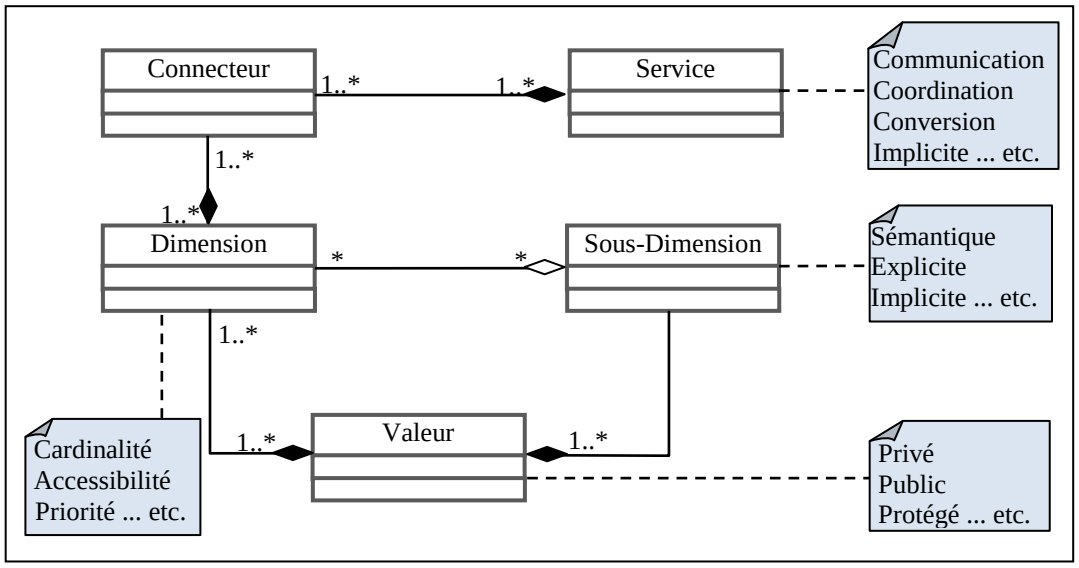

Figure 2.1- Modélisation d'un connecteur.

- *Facilitation:* Les données échangées doivent être soumises à des règles d'optimisation pour faciliter l'interopérabilité entre composants hétérogènes ou non. C'est dans cette vision que ce type de connecteurs est mis en place. Les problèmes de confidentialité, de compétitivité, de performance et autres sont à l'origine de ce type de connecteur.

Il est à noter que ces catégories ne présentent pas une classification exclusive, cela dit qu'un connecteur peut appartenir à plusieurs types différents catégories, par exemple un connecteur peut assurer à la fois la communication et la coordination. Serte que cette classification fournit une large catégorisation des connecteurs qui en manque beaucoup de détails au niveau des feuilles. Ce qui rond les processus de modélisation et de facilitation à l'exploration d'autres type de connecteurs une opération fastidieuse (Mehta, et al., 2000). Ces derniers proposent une autre classification basée sur la manière dont le connecteur opère l'interaction des services. La classification respecte le modèle présenté dans la Figure 2.1. La taxonomie a convergée sur huit classes de connecteurs que l'architecte considère lors de la phase de modélisation.

- *Appels de procédures:* ce type de connecteurs offre l'avantage de fournir des services de communication et de coordination. La première est assurée par une fonctionnalité de paramétrisation entre services pour réaliser le transfert. Quant à la coordination est substitué aux invocations interpellées pour modéliser le flot de données. Ce sont parmi les connecteurs les plus utilisés surtout dans le langage d'assemblage des logiciels interconnectés (Shaw, 1993).

Un exemple des appels de procédures incluant les connecteurs dans les méthodes orientées objets, *fork* et *exec* dans l'environnement Unix et les appels de système d'exploitation.

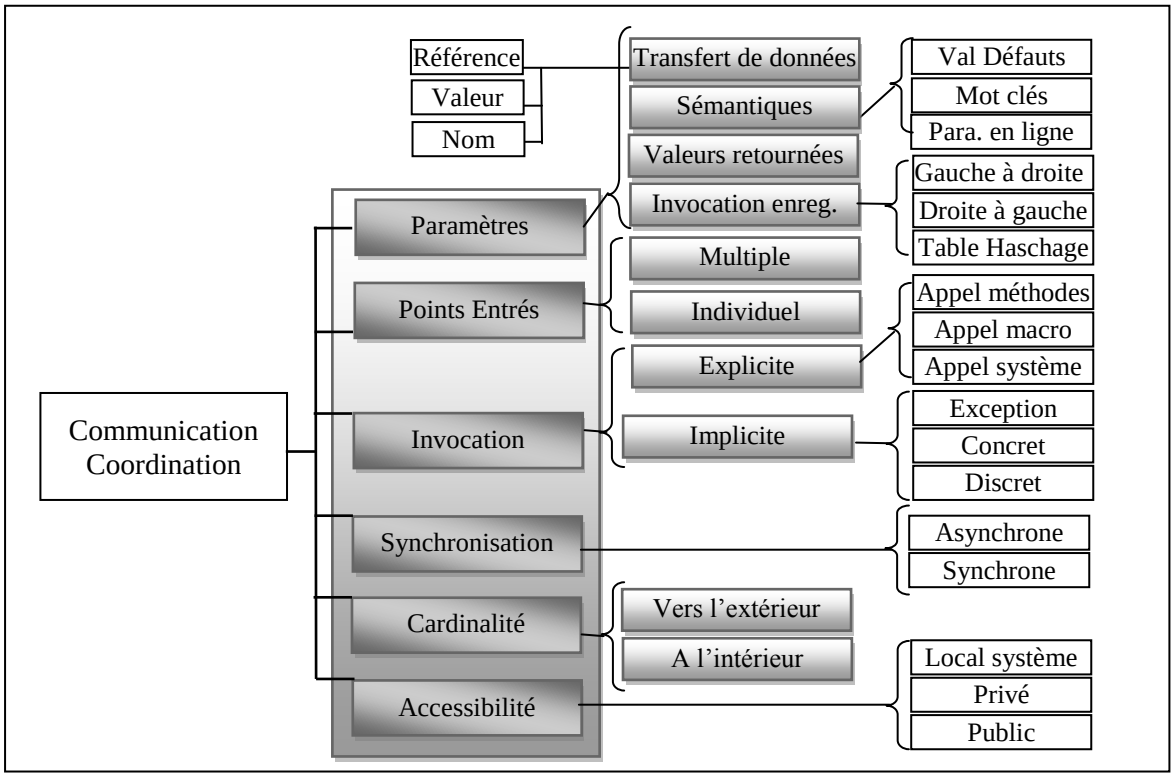

Figure 2.2- Connecteurs de type Appels de procédures.

Les appels de procédures sont souvent utilisés pour composer des connecteurs tel est le cas de la plateforme RPC (Amirat & Oussalah, 2009) permettant de traiter les facilitations de services. La structure d'un connecteur de type appel de procédure est montré dans la Figure 2.2 en termes de dimensions, de sous dimensions et de valeurs.

- *Evènements:* Souvent, le résultat d'une invocation d'opérations sur un objet donne lieu à deux événements différents. Le premier est le cas où l'invocation s'est déroulé normalement l'autre cas

est que cette invocation a générée un état anormal dans l'objet. Les informations que génèrent ces événements sont nécessaires pour réaliser la communication entres ce qui est fournit et requis. De plus, ce type permet de modéliser le flot de contrôle ce qui relève de la coordination entre composants de la première classification. Une fois un événement est détecté, le connecteur événement procède à la génération de messages pour toutes les parties intervenantes et effectue un control au niveau de tous les composants intéressés par ces messages. Le contenu d'un événement peut être détaillé pour pouvoir renseigner le temps, le lieu et autres applications spécifiques comme il est indiqué dans la Figure 2.3.

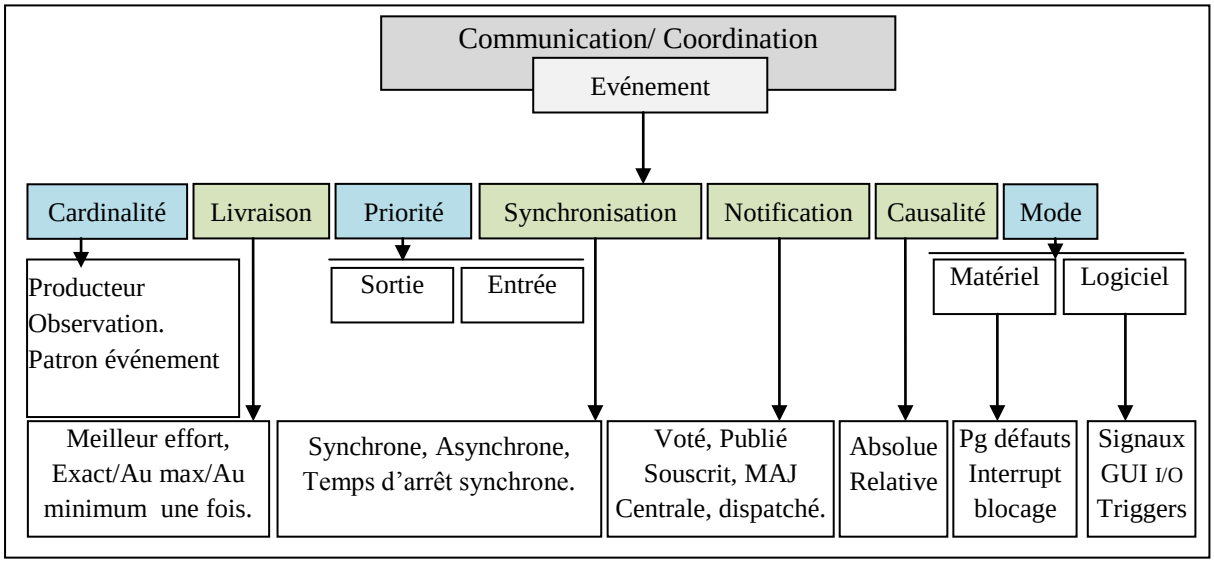

Figure 2.3- Connecteur de type événement.

Par exemple, les systèmes distribués à base événement, et spécialement les systèmes à communication asynchrone, constituent un exemple de ce type de connecteurs puisqu'ils tablent sur la notion de temps et d'actions commandées.

- *Accès aux données:* les connecteurs d'accès aux données fournissent un service de communication entre les composants et les composants de stockage de données. Les données ne sont pas toujours livrées dans les formats dont ils ont été stockés, de ce fait le connecteur doit être à la mesure de les fournir dans le format sollicité pas le composant ce qui relève du type de connecteurs de conversion. Notons aussi que les mécanismes d'accès dépendent étroitement de la façon d'enregistrement de données permanentes ou temporaires.

Parmi les exemples d'accès persistant aux données citons, *les mécanismes de requête, tels que SQL pour l'accès à la base de données, et l'accès aux informations stockées dans des référentiels, tels que le référentiel d'interface CORBA. Un autre exemple d'accès transitoire aux données: accès à la pile et à la mémoire, et mise en cache des informations.*

- *Liens:* Sont utilisés pour maintenir la liaison d'un ensemble de composants pendant l'exécution d'une opération. Le connecteur établit ce lien par un ensemble de canaux de communication et de coordination et qui sont par la suite utilisés par d'autres connecteurs de commandes pour instruire les interactions sémantiques et donc assurer un service de facilitation. Ces connecteurs peuvent soit disparaitre lorsqu'ils terminent leurs rôles soit être maintenus pour des raisons d'évolution du système. Des exemples de connecteurs de liaison sont les liens entre les

composants et les bus dans une architecture de style C2 (Taylor et al., 1996) et les relations de dépendance entre les modules logiciels décrits par les langages d'interconnexion de modules (MIL) (Prieto-Diaz & Neighbors, 1986).

Notons que les connecteurs de liaison n'améliorent pas la fonctionnalité d'un système, mais qu'ils sont nécessaires pour développer, surveiller et réparer les systèmes existants. La description architecturale qui base les interactions uniquement sur les connecteurs de liaison peut entraver la compréhension du système. L'architecture Linux est un exemple concret de ce type d'architecture comme il a été présenté dans (Bowman, Holt, & Brewster, 1999).

- *Flots:* Ce type de connecteurs servent à transférer un flot de données de grande quantité entre composants. On fait généralement appel à ce type de connecteur dans les modélisations qui utilisent des protocoles de communication bien complexes (Mehta, et al., 2000). Les flots peuvent être combinais avec d'autres types de connecteurs par exemple avec les connecteurs d'accès aux données pour fournir un connecteur composite de transfert et d'accès aux données.

Des exemples de connecteurs de flots sont : *les canaux en UNIX, les sockets de communication TCP/UDP[1] et les protocoles client-serveur propriétaires.*

- *Arbitres:* L'interaction entre les composants d'un système n'est pas toujours définie de manière stricte. Les besoins ou les états de composants peuvent ne pas exprimer ce qu'ils offrent ou requièrent. Dans de tel cas, les connecteurs arbitres permettent de résoudre les conflits et de rationaliser l'interaction (facilitation). De même, ils doivent assurer la coordination entre composants pour rediriger le flux de contrôle résultant (coordination). En plus, les arbitres offrent des services d'équilibrage de charge, de planification, de fiabilité et de sécurité du système (Stavridou, 1999). Par exemple*, les systèmes multithreads qui nécessitent un accès à la mémoire partagée, utilisent la synchronisation et le contrôle de simultanéité pour garantir la cohérence et l'atomicité des opérations.*

- *Adaptateurs:* Ces connecteurs interviennent dans le cas où l'interaction implique d'autres composants non planifiés pour inter-opérer. Dans ce cas, les adaptateurs font appel à une fonction de conversion dont l'utilité est de :

- Mettre en correspondance des stratégies et protocoles de communication entre composants hétérogènes.
- Optimiser des interactions pour mieux servir le système en exécution.
- Effectuer des transformations par « matching » des services requis avec les facilitations désirées.

Exemple : XML metadata interchange (XMI) est un exemple concret de connecteurs qui supporte l'échange de données entre applications hétérogènes.

- *Distributeurs:* Les connecteurs distributeurs sont chargés d'identifier les différents chemins d'interactions, du routage des informations et de la coordination entre les composants interagis. De ce fait, ces connecteurs assurent essentiellement une fonction de facilitation. Alors, ils sont toujours utilisés conjointement à d'autres connecteurs pour les assister à accomplir les tâches qui

[1] TCP/UDP pour Transport Control Protocol-User Datagram Protocol

lui sont assignées. Par exemple, *les systèmes embarqués ont besoin d'identifier les localisations et les chemins des différents nœuds par l'utilisation des DNS (Domaine Name Service).*

La Table 2.1 résume les différents services de connecteurs par référence aux critères de communication, coordination, conversion et facilitation.

Table 2.1- Récapitulatif des catégories de connecteurs

Classification par Service	Communication	Coordination	Conversion	Facilitation
Appels de procédures	●	●	○	⊙
Evènement	●	●	○	⊙
Accès aux données	○	●	●	○
Liens	⊙	○	○	●
Flots	○	●	○	○
Arbitres	⊙	●	○	●
Adaptateurs	○	○	●	○
Distributeurs	○	○	○	●

Légende- ● *Fort* ⊙ *Moyen* ○ *Faible*

b. Topologie des connecteurs

La topologie des connecteurs est répartie en deux vues distinctes.

Vue externe : Le *connecteur* est caractérisé dans sa structure externe par deux éléments essentiels:

- *Interfaces* : représentent la partie visible par les utilisateurs. Dans certains ADL comme Wright (Graiet, Bhiri, Dammak, & Giraudin, 2006) les interfaces sont appelées rôles. Une interface serve essentiellement pour décrire les participants de l'interaction avec le connecteur. Ils décrivent les mécanismes de connexion et le rôle de chaque composant engagé dans la dite interaction.
- *Propriétés des connecteurs :* sont regroupées en deux types différents :
    - Propriétés fonctionnelles qui représentent tous ce qui est relatif aux besoins du connecteur pour assurer une bonne implémentation. Ces informations sont indispensables pour pouvoir analyser le comportement des connecteurs pour permettre une meilleure sélection de connecteurs.
    - Contraintes qui représentent les conditions nécessaires et suffisantes pour utiliser le connecteur. En pratique, ces conditions doivent être remplies pour dire que le système est cohérent.

Vue interne du connecteur : Nous distinguons dans la littérature deux types de connecteurs par considération des caractéristiques internes.

- *Connecteurs atomiques :* identifiés généralement avec le concept *Glue*, se sont des points d'accès des composants. L'intérêt est de décrire le protocole de communication entre les interfaces des connecteurs. La présentation des connecteurs dans certains ADL est soit simple présentée sous forme d'une liste prédéfinie (Bálek & Plášil, 2001) ou soit introduite pour permettre à l'utilisateur de définir lui même son propre connecteur atomique.

- *Connecteurs composites :* sont des connecteurs dont la structure est formalisée en termes de plusieurs autres composants, connecteurs et configurations. Cette structure est appelée ainsi architecture interne du connecteur.

2.2.1.4 Configuration

La configuration n'est autre qu'un graphe regroupant un ensemble de composants et de connecteurs qui ont pour rôles d'assurer les liaisons adéquates. Les ports de composants sont donc associés aux rôles des connecteurs appropriés pour la conception d'une architecture logicielle. La configuration est indispensable à l'architecture pour assurer le bon fonctionnement et la communication entre interfaces des composants/connecteurs. Elle représente la solution clé pour faciliter l'utilisation et la réutilisation des composants/connecteurs par les divers intervenants. De plus, les configurations offrent aux développeurs une abstraction du système moins complexe car elles relèguent certains détails techniques ce qui les rendent plus faciles à comprendre par une plus grande frange d'intervenants. *La configuration permet par exemple de décrire le flux de données associant les ports des composants de type « filter » aux rôles des connecteurs de type « pipe ».*

La gestion de la configuration peut déterminer la profondeur ou la largeur d'une architecture pour pouvoir juger respectivement de la performance et du degré de dépendance inter composants de l'architecture(Smeda, Oussalah, & Khammaci, 2005).

a. Typologie des configurations

Les configurations sont considérées comme des classes qui peuvent être instanciées pour pouvoir construire de différentes architectures. Il existe peu d'ADL qui permettent la définition des types de configuration (Barais & Duchien, 2005) comme COSA (Maillard, Smeda, & Oussalah, 2007). Les configurations peuvent se présentées sous deux formes distinctes :

- *Configuration de type multi graphe :* où les connecteurs (arêtes dans le graphe) peuvent connecter plus de deux composants (représentées par les sommets). L'inconvénient d'un tel graphe est que les composants et les connecteurs ne sont pas placés dans le même niveau.
- *Configuration de type bipartie :* ce type de configuration se base sur des graphes où chaque arête du graphe associe deux sommets qui sont obligatoirement de différents types (connecteur et composant). Dans ce cas, les deux éléments architecturaux sont représentés au même niveau et le probléme exposé dans le type multigraphe est résolu. La configuration doit vérifier que les sommets connecteurs doivent reliés au moins deux composants.

b. Topologie des configurations

La topologie de la configuration représente la structure des connexions établies entres composants et connecteurs. Donc, une configuration lui est impérativement associée, en plus de son type, des contraintes et des PNFs.

- *Contraintes de la configuration :* elles expriment les différentes dépendances entre les éléments architecturaux qui les composent. Elles peuvent être de types locales pour exprimer les spécificités d'un composant et/ou connecteur ou globales lorsqu'elles portent sur un ensemble d'éléments architecturaux.
- *Propriétés non fonctionnelle d'une configuration:* la présence de telles propriétés justifie que les propriétés n'ont pas pu être décrites au niveau des éléments architecturaux associés. On fait

alors appel à la configuration pour abriter ces propriétés puisqu'elles présentent une spécification de l'architecture de connexion. Par exemple, *ces propriétés peuvent servir à sélectionner les meilleurs candidats de composants et de connecteurs pour une configuration donnée.*

2.2.2 Styles architecturaux

Une architecture type peut être une description de solution particulière où convergent plusieurs problèmes différents, ce qui reflète le concept de style d'architecture ou patron d'architecture. En effet, ce dernier permet de générer à partir d'un type de composants ou de connecteurs toutes les règles permettant de configurer et d'assembler les composants constituant la solution. Le style peut aussi disposer d'une spécification associée à la description du comportement. L'intérêt primordial est de permettre la capitalisation du savoir, de l'expertise et de l'expérience des intervenants des différents niveaux pour permettre plus de réutilisabilité, compréhension, performance et documentation. Il existe plusieurs styles d'architectures logicielles.

2.2.2.1 Architecture tube et filtre

Tube et filtre est un patron de conception architecturale permettant un traitement de messages asynchrones. Dans ce modèle, il existe de nombreux composants, appelés filtres, et des connecteurs entre les filtres appelés tubes (pipe). Chaque filtre est responsable de l'application d'une fonction aux données fournies c'est ce qu'on appelle le filtrage. Les filtres fonctionnent de manière asynchrone. L'architecture en tube stipule l'utilisation d'une file d'attente pour véhiculer les messages entre composants du système. Les filtres sont des composants concurrentiels qui traitent, indépendamment et de manière asynchrone, le flot de données, reçu des canaux d'entrée. Le résultat de traitement est ensuite envoyé à travers plusieurs autres canaux de sortie. Il convient donc de dire que les canaux se réservent la fonction de faire circuler le flot de données entre les filtres producteurs et les filtres consommateurs via un buffer pour permettre la synchronisation.

Exemple : *Dans les compilateurs, les filtres consécutifs effectuent l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique et la génération de code*.

Nous pouvons illustrer les différentes caractéristiques d'un style de type tube et filtre comme suit :

- Caractéristiques au niveau pratique : ce type de style présente l'avantage d'être :

- Bon pour traitement en lot (batch).
- Très flexible.
- Analyse facile : performance, synchronisation, goulot d'étranglement, etc.
- Se prête bien à la décomposition fonctionnelle d'un système. Le seul inconvénient est que, le style présente des limites pour le traitement interactif.

- Caractéristiques au niveau de conception : L'utilisation de ce style architectural permet au niveau conceptuel de :

- *Diviser pour régner* : les filtres peuvent être conçus indépendamment.
- *Cohésion* : les filtres sont un type de cohésion fonctionnelle.
- *Couplage* : les filtres n'ont qu'une entrée et une sortie en général.

- *Abstraction* : les filtres cachent généralement bien leurs détails internes.
- *Réutilisabilité* : les filtres peuvent très souvent être réutilisés dans d'autres contextes. Il est souvent possible d'utiliser des filtres déjà existants pour les insérer dans le pipeline.

2.2.2.2 Architecture avec référentiel de données

Un référentiel de données est un dispositif permettant la gestion mutualisée et le pilotage dans le temps des données de référence, à savoir, l'actualisation et la mise à disposition de ces données. Un référentiel de donnée comprend aussi les éléments permettant de piloter et faire évoluer cette gestion de données en fonction des besoins des «clients» du référentiel, et en fonction des problèmes de qualité de données rencontrés (Fielding & Taylor, 2000).

L'architecture avec référentiel de données est utilisée dans le cas où des données sont partagées et fréquemment échangées entre les composants. Dans cette architecture, il existe deux types de composants : composant référentiel de données, et composant accesseur de données. Les composants référentiels constituent le medium de communication entre les accesseurs. Un connecteur est relié à un référentiel accesseur pour assurer la communication, et aussi la coordination, la conversion et la facilitation.

2.2.2.3 Architecture Modèle-Vue-Contrôleur (MVC)

L'architecture MVC est un style architectural permettant d'isoler les données de leurs représentations par la séparation de la couche interface utilisateur des autres parties du système. La proposition de cette architecture est motivée par la rapidité de changement des interfaces utilisateurs par rapport à la base de connaissance. Elle permet donc de distinguer la consultation de la modification. L'MVC renferme trois composants des données:

- **Modèle :** Représente la structure de données qui rassemble les données du domaine et les connaissances du système. Il définit les accès aux propriétés de ces données à travers les classes dont les instances doivent être vues (dans le composant Vue) et manipulées (par le composant Contrôleur). Le modèle est indépendant des vues et des contrôleurs et il est considéré comme le noyau de l'application car il enregistre les vues et les contrôleurs qui en dépendent (il maintient une liste d'écouteurs qui ne sont que des vues et des contrôleurs). Cela permet de prévenir ces écouteurs lorsque la donnée est modifiée.

- **Vue:** Définit la représentation des données pour affichage. Elle est utilisée pour présenter/afficher les données encapsulées par le modèle dans l'interface utilisateur. La vue est mise à jour lorsque le modèle est modifié. La vue est considérée comme l'interface graphique de l'application et s'enregistre comme un écouteur du modèle, son rôle est de:

  - Créer et initialiser ses contrôleurs,
  - Afficher les informations destinées aux utilisateurs,
  - Préserver la cohérence du modèle via l'implantation des procédures de mise à jour nécessaires et finalement
  - Consulter les données du modèle.

- **Contrôleur :** Représente la partie de l'application qui prend en charge les décisions, il dispose des moyens permettant de modifier la valeur des données. Un contrôleur permet à l'utilisateur de modifier la donnée encapsulée dans le modèle. Il doit éventuellement s'enregistrer comme un

écouteur du modèle pour être mis à jour si le modèle est modifié. Son rôle selon (Deacon, 2009), (Qazi, McElholm, & Maguire, 2018) est de :

- Appeler les accesseurs du modèle en fonction des actions de l'utilisateur,
- Accepter les événements correspondant aux entrées de l'utilisateur et puis
- Traduire ces évènements soit en demande de service destinée au modèle ou bien en demande d'affichage destinée à la vue.

2.2.2.4 Architecture multicouche

L'architecture multicouche est généralement utilisée pour les systèmes implémentant des protocoles de communication (TCP/IP). C'est une organisation hiérarchique du système en un ensemble de couches. Elle offre des interfaces bien définies entre les couches, où chaque composant (couche) réalise un service de la façon suivante:

- Une couche offre un service (serveur) aux couches externes (clients),
- Chaque couche regroupe les composants (objets, méthodes) partageant les mêmes fonctionnalités et les mêmes rôles,
- Le respect de la notion d'abstraction, où les couches externes sont plus abstraites (haut niveau) que les couches internes.

Les connecteurs sont des protocoles de la couche d'interaction, il existe deux types de protocoles.

- **Système fermé :** une couche n'a accès qu'aux couches adjacentes. Donc, les connecteurs ne relient que les couches adjacentes.

- **Système ouvert** : toutes les couches ont accès à toutes les autres. Donc, les connecteurs peuvent relier n'importe quelles deux couches.

2.2.2.5 Architecture n-niveaux

L'architecture n-niveaux est comparable à une architecture multicouche, dont les couches seraient distribuées, alors elle est utilisée dans les systèmes distribués. Cette architecture fragmente le système en plusieurs niveaux, par exemple le niveau présentation (Vue), le niveau logique (Contrôleur) et le niveau données (Modèle). Où chaque niveau ne dépend que du niveau qui lui est dessus (de haut niveau).

La notion de tiers a pris la signification de couche distribution. Chaque niveau est représenté par un composant qui peut être un client, un serveur ou les deux à la fois. Les connecteurs sont basés sur une connexion asymétrique reliant un client à un serveur. Ils doivent supporter les communications distantes comme les requêtes HTTP, RPC, et TCP/IP. Nous trouvons en pratique ces architectures présentées sous formes:

- **Architecture 2-niveaux :** c'est à dire une architecture Client-serveur (ou client lourd) ou une architecture pair-à-pair.

- **Architecture Client-serveur :** composée de deux types de niveaux ou composants (Fielding et al., 2017) :

- *Client :* son rôle consiste à recevoir les données des utilisateurs, initier les transactions …etc.
- *Serveur:* son rôle consiste à exécuter les transactions, assurer l'intégrité des données …etc.

- **Architecture pair-à-pair:** est une généralisation de l'architecture client-serveur dont les composant peuvent tous à la fois être client et serveur (Ripeanu, 2001; van der Linde et al., 2017). Cette architecture est caractérisée par le fait que sa conception est plus difficile et le flot de contrôle est plus complexe donc il est possible de tomber dans une situation d'inter-blocage.

- Architecture 3-niveaux: L'architecture 3-niveaux est considérée comme une organisation en trois couches qui sont :

- *Couche interface:* composée d'un ensemble d'objets interfaces (boundary objects) pour interagir avec l'utilisateur (GUI : Graphic User Interface, par exemples : fenêtres, pages web, formulaires, etc...).
- *Couche logique applicative (couche métier):* Comporte tous les objets de contrôle et les entités nécessaires qui assurent le traitement des données, la vérification des règles et les notifications requises par l'application.
- *Couche de stockage (Base de données):* réalise le stockage, la récupération et la recherche des objets persistants.

L'avantage de cette architecture par rapport à l'architecture client-serveur est le fait que l'architecture 3-niveaux permet le développement, la réutilisation et la modification des interfaces utilisateurs indépendamment de la logique métier de l'application. La partie cliente doit vérifier les propriétés de:

- Atomicité c.-à-d. qu'elle devient la plus fine possible. Donc le but est de diminuer le rôle du client sur l'affichage d'un contenu HTML et
- Ajouter le rôle de création des pages web à afficher par le client au niveau de la couche logique applicative.

- Architecture 4-niveaux : L'architecture 4-niveaux est comparable à une architecture 3-niveaux dont la couche logique applicative est décomposée en deux couches :

- *Couche présentation (JSP, servlets)*- constituée de deux types de présentation qui sont la présentation du contenu statique affiché par le client (contenu HTML ou XML) et la présentation du contenu dynamique qui est un contenu organisé et créé dynamiquement par le serveur web (pour ensuite être affiché sous forme HTML/XML par le client).
- *Couche logique applicative*- considérée comme le cœur de l'application, puisqu'elle ne contient que les calculs purs qui réalisent les traitements de l'application.

La Table 2.2 présente une comparaison exhaustive entre les différents styles architecturaux pour statuer sur les caractéristiques de chacune d'elles.

2.2.3 Vues architecturales

Dans les systèmes complexes, le processus de développement intègre souvent plusieurs acteurs chacun désir intervenir selon ses propres connaissances et/ou préoccupations. Cela engendre une complexité supplémentaire à l'égard des architectes du système spécialement quand ces points de vue sont complexes et non précis pour qu'ils soient bien pris en charge.

Table 2.2- Comparaison entre styles architecturaux.

Style / Critères	Pipeline	Référentiel	MVC	Multi couches	n-niveaux
Interopérabilité	○	○	⊙	⊙	⊙
Flexibilité	●	●	○	⊙	●
Portabilité	⊙	○	○	○	●
Maintenabilité	●	●	●	●	●
Réutilisabilité	●	●	●	●	●
Extensibilité	●	●	⊙	●	●
Efficacité	⊙	⊙	○	○	⊙
Transparence	○	○	○	⊙	⊙
Composabilité	⊙	⊙	⊙	⊙	●
Simplicité	●	⊙	⊙	●	●

Légende- ● Fort ⊙ Moyen ○ non renseignée

[Source : par inspiration des travaux cités dans cette sous section]

2.2.3.1 Définitions

Une vue est définie comme:

"*The construction of a complex description or model involves many agents (participants or actors). These agents have different perspectives or views of the artifact or system they are trying to describe or model (the domain of discourse). These perspectives or views are partial or incomplete descriptions which arise because of different responsibilities or roles assigned to the agents*" (Finkelstein & Sommerville, 1996).

Et ausi comme,

"*A widely used approach is to attack the problem from different directions simultaneously. In this approach, the architectural description is partitioned into a number of separate views, each of which describes a separate aspect of the architecture*" (Rozanski & Woods, 2012).

Sur l'aspect informel, la vue est définie selon (Kheir, Naja, Oussalah, & Tout, 2013) comme,

"*A view is an architectural design of a stakeholder's viewpoint that covers delicately all its interests and required functionalities, and represents them formally in a structured architecture*".

Les définitions ci-dessus appuient le fait qu'une vue représente la modélisation de l'architecture du système pour parvenir à traduire de manière structurée et formelle tous les points de vue exprimés par les intervenants et qui reflètent des préoccupations parfois divergentes. Plusieurs domaines du génie logiciel se sont préoccupés de la vue comme une thématique au centre des préoccupations des architectes(Knapp & Mossakowski, 2018).

2.2.3.2 Les vues en modélisation logicielle

Avant l'apparition et l'adoption incontournable de l'UML comme outils et langage de modélisation dans le domaine du génie logiciel, Mullery avait développé une méthode, appelée CORE, pour la spécification et la conception des exigences (Mullery, 1979) . Le CORE est une approche de modélisation reposant sur une notation schématique qui décompose le processus de modélisation en plusieurs étapes et définit pour chacune d'elles un niveau de définition et un

ensemble de points de vue pour lesquels elles modéliseront leurs besoins et établiront de manière itérative la cohérence entre elles. Au début des années 1990, (Booch, 2006) a développé le langage de modélisation unifié (UML), qui était adopté par l'OMG en 1997. Dans la version actuelle de UML, 13 types de diagrammes, où chaque type de diagramme prescrit un point de vue implicite. Plusieurs chercheurs ont travaillé pour étendre le modèle UML afin d'ajouter le concept de vues dynamiques et points de vue comme dans (Nassar, 2003), (Bruneliere, Burger, Cabot, & Wimmer, 2017). En 2003, une extension UML basée sur les vues, VUML (Nassar, 2003), a été introduite pour offrir un outil de modélisation UML dans lequel les multiples points de vue du système sont pris en compte lors de la phase de modélisation. Ils ont donc proposé une méthodologie pour construire plusieurs modèles de classe, un pour chaque point de vue puis les fusionner pour obtenir un seul modèle VUML décrivant tous les points de vue. De plus, (Nassar et al., 2009) (Nassar, 2005) ont proposé un outil de génération de code qui génère un code objet du modèle VUML correspondant. Enfin, ils ont travaillé sur l'automatisation du processus de composition dans le modèle VUML. Un travail d'extension a été mené par (Anwar, Dkaki, Ebersold, Coulette, & Nassar, 2011) pour inclure les graphiques nécessaires afin d'atteindre cet objectif.

L'un des travaux importants réalisés dans ce domaine a été celui de (R. Dijkman & Dumas, 2004; R. M. Dijkman, Quartel, & van Sinderen, 2008), où ils ont présenté un cadre permettant à l'architecte de modéliser un système selon différents points de vue en définissant un ensemble de concepts de base relatif aux niveaux qui leurs sont reliés. Dans ce travail, ils se sont concentrés sur la définition des relations de cohérence entre ces vues.

2.2.3.3 Vues en développement logiciel

Dans le domaine du développement logiciel, les techniques de développement orientées aspects considèrent qu'un certain nombre de problèmes de développement logiciel ne peuvent pas être résolus à cause des limites de modularités posées par les langages orientés objet. Elles proposent de nouveaux artefacts (au-delà de la méthode, de la classe ou du package) pour séparer les nouveaux types de processus et de préoccupations qui sont auparavant fusionnés dans des paradigmes orientés objets (Mili, Sahraoui, Lounis, Mcheick, & Elkharraz, 2006). Dans ce domaine, plusieurs méthodes sont proposées:

- La technique de programmation par sujet (SOP) (Ossher & Tarr, 1999; Ward & Gullekson, 1994) : répond aux exigences fonctionnelles. Elle considère les applications orientées objet comme la composition de plusieurs tranches d'application représentant des domaines fonctionnels distincts. Chaque tranche appelée sujet consiste en un objet autonome programme orienté, avec sa propre hiérarchie de classe. Ensuite, un langage de composition est défini, permettant la composition hiérarchique des classes (sujets).

- La technique de programmation orientée aspect (AOP) répond aux exigences non fonctionnelles, y compris les aspects architecturaux, la gestion des erreurs, la sécurité, la distribution, la persistance, …etc. (Kiczales, 1996) et (Boudaa, Hammoudi, Mebarki, Bouguessa, & Chikh, 2017). Elle définit un aspect comme la mise en œuvre d'une préoccupation liée à plusieurs objets d'une collaboration. Un aspect est un module de code définit à travers des composants génériques qui répondent à une préoccupation spécifique. Par ce fait, il inclut les propriétés transversales des divers composants d'une application.

- La technique de programmation par vues (Ito & Fujimoto, 2015) considère chaque objet d'une application comme un ensemble de fonctionnalités de base, directement ou indirectement

disponibles, pour tous les utilisateurs de l'objet. Les vues représentent un ensemble d'interfaces spécifiques, à des utilisations particulières, et qui peuvent être ajoutées ou supprimées pendant l'exécution.

2.2.3.4 Vues en architecture logicielle

Dans l'architecture logicielle, les différents modèles ont été proposés pour créer une documentation (c'est-à-dire une description architecturale) en séparant les préoccupations. Chaque modèle décrit un ensemble de points de vue et identifie l'ensemble des préoccupations de chacun d'eux. Les différents modèles couvrent le même domaine d'architecture logicielle, y compris l'environnement structurel, organisationnel, commercial et technologique associé. Le concept de *Vue* apparaît dans l'un des premiers articles classiques de Perry et Wolf (Perry, 1992) sur l'architecture logicielle au début des années 90. En 1995, Philippe Kruchten a proposé quatre vues différentes (Kruchten, 1995) d'un système et l'utilisation d'un ensemble de scénarios (cas d'utilisation) pour vérifier leurs exactitudes. Dans (Hilliard, 1999), l'auteur propose un cadre de description d'architecture (ADF) dans lequel les vues sont des entités de première classe régies par des entités de type, appelées points de vue. Elles sont décrites en termes d'un ensemble de propriétés appartenant à l'application. Aussi, l'auteur a reconnu l'importance des vues dans la description de l'architecture et a adopté le concept de point de vue. Dans (Zarvić & Wieringa, 2014), l'approche présente un vaste ensemble de constructions appelées colonnes, très similaires aux vues. Un langage de point de vue formalise les concepts utilisés dans l'architecture logicielle et apporte une normalisation de la terminologie utilisée dans ce domaine.

- **Vue:** est une représentation de tout un système selon une perspective d'un ensemble de préoccupations connexes.

- **Point de vue:** spécification des conventions de construction et d'utilisation d'une vue. C'est un modèle à partir duquel on peut développer des vues individuelles en définissant les objectifs et le public visé, ainsi que les techniques de création et d'analyse de celle-ci. Un point de vue est une prise de conscience des préoccupations d'un ou plusieurs acteurs.

Il est utile de passer en revue certaines approches, basées sur les définitions ci-dessus de *vues* et *points de vue*, pour leurs caractéristiques principales:

Dans(Rozanski & Woods, 2012), les auteurs proposent un catalogue de points de vue pour les systèmes d'information, élargissant l'ensemble des points de vue 4 + 1 identifiés par Kruchten (Kruchten, 1995) et regroupant 6 points de vue fondamentaux, à savoir: fonctionnel, informationnel , développement, concurrentiel, déploiement et opérationnel.

Dans (Clements, 2002) trois points de vue, appelés types de vue, sont identifiés: Module, Composant et connecteur et Allocation. En outre, l'auteur définit une procédure en trois étapes pour choisir les vues pertinentes pour un système basé sur les préoccupations des intervenants. En outre, cette approche utilise le concept de vues combinées et la définition de priorités pour ramener l'ensemble de vues à une taille gérable pour les projets du monde réel.

L'enquête précieuse réalisée par (May, 2005) compare plusieurs modèles basés sur des vues et a identifié les relations de correspondances et de divergences entre les différentes vues. Il a proposé une couverture optimale du référentiel comprenant un ensemble de points de vue sélectionnés parmi les différents modèles et couvrant le plus grand nombre de concepts du

référentiel. La limite de cette approche est que même si les vues sélectionnées ne sont pas indépendantes aucune connexion ou relation n'est définie entre les vues.

2.3 Architecture Orientée service

Le Service Oriented Computing est devenu de nos jours la plate-forme pour les solutions d'automatisation pour les entreprises contemporaines. Au cœur de cette plate-forme se trouve le puissant modèle de l'architecture orientée service (SOA). De facto, l'intégration des principes et de la technologie SOA a permet aux organisations de créer des applications a plusieurs niveaux d'abstraction où la flexibilité, l'agilité et l'évolutivité sont l'hypocentre des préoccupations.

Dans cette section, nous donnons une définition détaillée de l'architecture orientée service qui nous permet de bien comprendre ce type d'architecture. Ensuite, nous expliquons tous les concepts et les principes liés à cette architecture dans le but d'assurer une bonne exploitation.

2.3.1 Concepts et principes SOA

2.3.1.1 Définitions

L'architecture orientée service (SOA) n'est pas une architecture mais un paradigme qui nous conduit à une architecture, c'est une approche ou une réflexion qui nous conduit à certaines décisions. L'architecture est basée sur l'utilisation des services "composants logiciels" qui effectuent une petite fonction, telle que la production de données, la validation d'un client, ou la prestation de services simples.

- **Service:** Le service est le composant de base de l'architecture SOA. C'est un composant logiciel autonome fonctionnant indépendamment de tout contexte ou service externe. Un service est un composant logiciel comme le montre la définition de (Schmidt, Hutchison, Lambros, & Phippen, 2005):

"A service is a software component that is described by meta-data, which can be understood by a program".

Il consiste en une fonction ou fonctionnalité bien définie permettant d'accéder à une ou à plusieurs ressources. Il est divisé en opérations qui constituent autant d'actions spécifiques que le service peut réaliser. En générale, on peut faire une analogie entre les concepts *services* et *opérations* dans le mode orienté service et les concepts: c*lasses* et *méthodes* dans le mode orienté objet. La construction et l'exposition des services n'est pas la seule capacité de SOA, mais elle permet aussi d'utiliser ces services dans plusieurs applications différents. La SOA lie ces services à travers l'orchestration, ou en chorégraphie. Ainsi, la SOA est basée sur la fixation des architectures déjà existantes en traitant les plus grands systèmes en tant que services interconnectés ou séparés et puis les abstraire dans des domaines spécifiques.

- **Interopérabilité:** Elle permet la propagation des fonctionnalités des services web via des systèmes hétérogènes. L'interopérabilité facilite la coordination entre les services invoqués et ou enregistrés dans le dépôt de services. Par exemple, pour assurer une interopérabilité pertinente au niveau des Web Service, des règles et des profils spécifiques ont été introduits pour réduire les problèmes d'interopérabilité, au moins au niveau du format de message. Le profil de base qui spécifie plusieurs règles différentes, y compris ce qui concerne le formatage et les relations à

maintenir entre les informations contenues dans les fichiers WSDL associés, les messages SOAP et les entrées UDDI, afin de faciliter la communication entre les différents services.

L'interopérabilité d'entreprises désigne la faculté des institutions, tels que les administrations, organisations et autres, d'inter-opérer des entités fonctionnelles comme les services ou activités ou données. Ceci représente un avantage concurrentiel pour ces institutions puisqu'elle leur permet de développer des solutions pratiques, modernes et surtout flexibles (Pessoa, 200). L'interopérabilité pour les entreprises est liée à deux aspects (Chen & Daclin, 2006): le premier est organisationnel tandis que le second est technologique. L'interopérabilité d'entreprises et des activités est liée au premier aspect alors que celle des services et de données (informations) relève de l'aspect technologique.

- **Couplage faible:** Les interactions dans l'architecture orientée service sont mise en œuvre avec des services faiblement couplés capables de fonctionner indépendamment les uns des autres. Le point fort de l'architecture SOA est qu'elle permet la réutilisation des services, ceci offre un grand avantage pour les grandes entreprises lorsqu'elles sont forcées de faire des mises à niveau et d'autres modifications pour économiser le temps et l'argent. Le couplage reflète le niveau de dépendance entre les composants logiciels. Par conséquence, le couplage peut être fort est on dit que les composants sont étroitement couplés (couplage lâche) et on dit qu'ils sont faiblement couplés. Ces derniers présentent la faculté d'être plus flexible puisque les dépendances sont bien réduites et définies. Souvent, la majorité des architectures logicielles tentent de fournir moins de composants ou de modules pour offrir une organisation plus simple. Le couplage faible est le concept qui assure l'évolutive, la flexibilité et la tolérance pour un service. Donc, le but est de minimiser les dépendances entre services dans une architecture orientée. Quand on minimise les dépendances on va aussi minimiser l'influence sur les autres systèmes.

2.3.1.2 Caractéristiques

L'architecture SOA est basée essentiellement sur une collection de services communiquant entre eux via des messages. Cette communication peut consister à un simple retour de données ou à toute une activité. Ci-après, nous citons brièvement les principales caractéristiques d'une architecture SOA.

- **Interface:** Un service peut implémenter plusieurs interfaces, et aussi plusieurs services peuvent partager une interface commune,

- **Localisable:** Chaque service ayant une localisation bien définie,

- **Large Granularité:** Les opérations proposées par un service encapsulent plusieurs fonctions et traitent une large échelle de données contrairement à la notion de composant technique. Une granularité fine d'un service, comme l'accès aux données de base par exemple, offre peu d'utilité au processus métier. Contrairement, une grosse granularité est une composition de plusieurs services de fines granularités considérée en somme comme un composant conforme à l'entreprise. Cette granularité dépend essentiellement des scénarios d'utilisation et des exigences de l'entreprise (Papazoglou & Van Den Heuvel, 2006).

- **Instance unique:** Un service est unique c'est-à-dire on ne peut pas avoir plusieurs instances d'un service en même temps. Il correspond au design pattern (patron de conception créateur) *Singleton*,

- Couplage faible: Les services sont connectés aux clients et autres services via des standards sous forme des documents XML. Ces standards assurent la réduction des dépendances,

- Type de communication: La communication entre les services entre eux ou bien avec les clients peut être *synchrone* ou *asynchrone*. La Figure 2-4 montre les deux types de communications qui sont largement utilisées dans les web services.

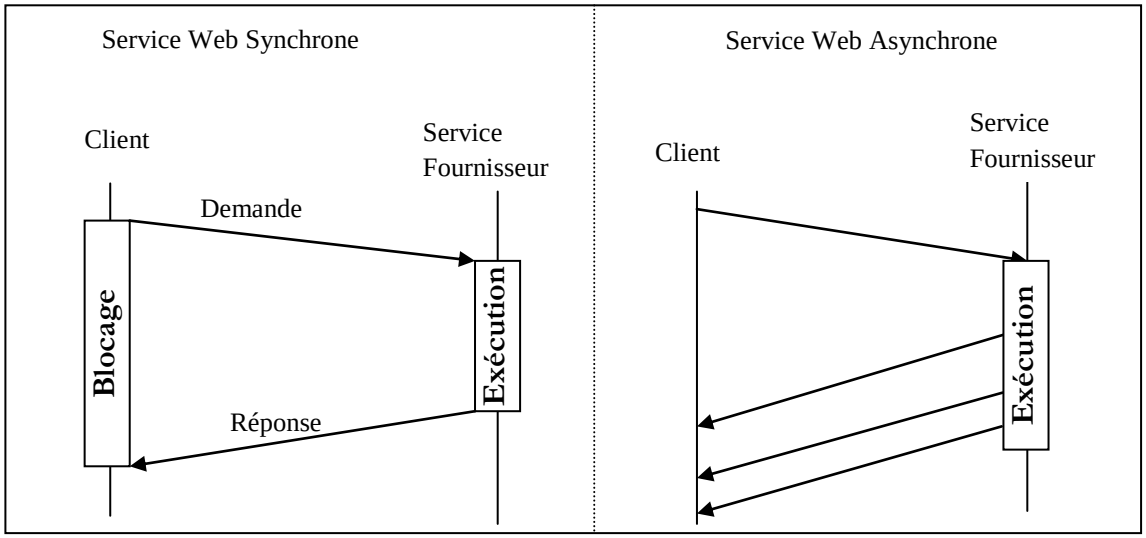

Figure 2.4- Communication synchrone et asynchrone d'un Web service.

2.3.1.3 Principes de SOA

L'architecture SOA se conforme à divers principes de gestion des services influençant directement le comportement intrinsèque d'une solution logicielle et le style de sa conception. Les concepts SOA sont présentés dans la Figure 2.5, dont le concept essentiel est le service qui est le composant de base de l'application. Les services gèrent des états et sont pilotés par des politiques qui imposent des exigences opérationnelles. Les services échangent ainsi des messages conformément à des contrats qui respectent les politiques connues à priori. Elles contiennent des schémas qui décrivent les messages échangés en respectant plusieurs principes à savoir :

- L'encapsulation des services,
- Le faible couplage des services avec la maintenance d'une relation réduisant les dépendances,
- Le contrat de service adhère à un accord de communication défini avec un ou plusieurs documents de description,
- L'abstraction des services cachant la logique du service à l'extérieur,
- La réutilisation des services partageant la logique entre plusieurs services avec l'intention de promouvoir la réutilisation,
- La composition des services,
- L'autonomie des services car ils peuvent fonctionner indépendamment,
- L'optimisation des services,
- La découverte des services depuis leurs descriptions extérieures.

L'architecture orientée service considère toutes les ressources informatiques d'une entreprise comme un service à l'aide d'un moyen technique d'intégration des différents systèmes d'information de cette entreprise.

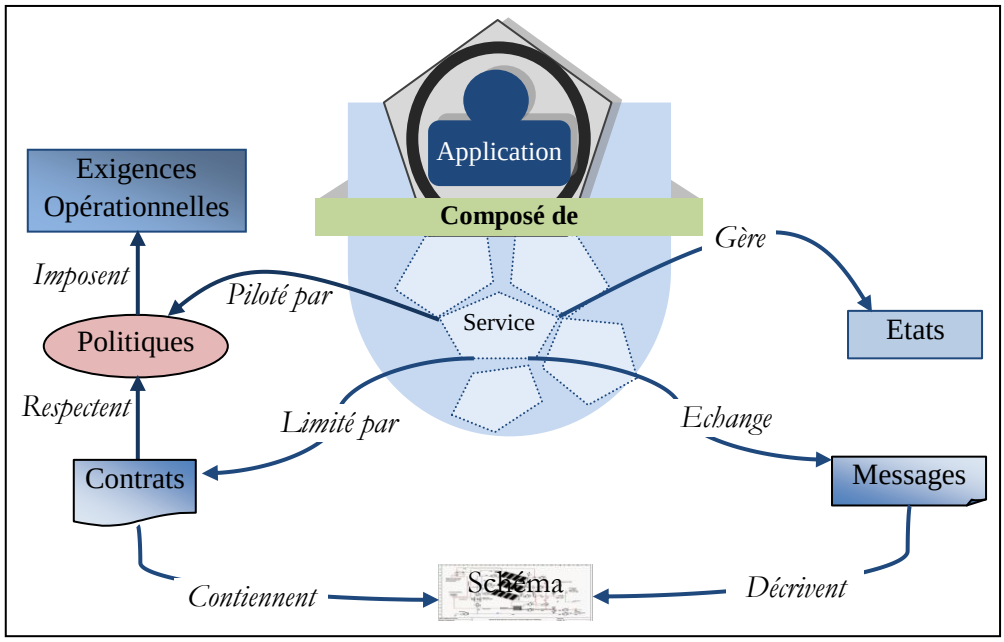

Figure 2.5- Concepts liés à la technologie SOA

2.3.1.4 Acteurs SOA

La Figure 2.6 résume les principaux acteurs de l'architecture SOA.

- Service fournisseur: Il met le service web en application et il le rend publiquement disponible sur internet,

- Service consommateur: C'est un client consommateur ou demandeur qui fait la demande d'un service web bien précis et qui répond parfaitement à ses besoins,

- Service annuaire: Le registre fournit un dépôt où le consommateur peut trouver des nouveaux services web et le fournisseur dispose une description pour des nouveaux services web publiés.

2.3.1.5 Avantages d'une architecture orientée service

Une architecture orientée service permet d'obtenir tous les avantages d'une architecture client-serveur et notamment :

- Une modularité permettant de remplacer facilement un composant (service) par un autre,
- Une réutilisabilité possible des composants (par opposition à un système tout-en-un sur mesure fait pour une organisation),
- Une meilleure opportunité d'évolution (il suffit de faire évoluer un service ou d'ajouter un nouveau service),
- Une plus grande tolérance aux pannes,
- Une maintenance plus aisée.

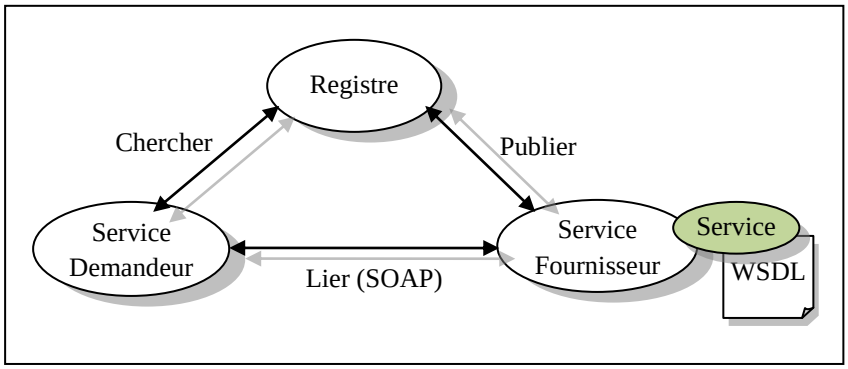

Figure 2. 6-Architecture de l'orienté services

[Source : (Sommerville, 2011)].

2.3.2 Modèles et approches orientées services

2.3.2.1 Modèle d'architecture

Il existe selon (Erl, 2008) huit principaux modèles des architectures orientées services :

- **Service contrat standard:** Les services au sein d'une même entreprise ou d'un même domaine sont conformes aux mêmes normes de conception de contrat.

- **Service couplage faible:** Dans le couplage faible ou lâche, les contrats de service imposent de faibles exigences en matière de couplage consommateurs et sont eux-mêmes découplés de leurs environnements.

- **Service d'abstraction:** Les contrats de service ne contiennent que des informations essentielles et les informations sur les services sont limitées à ce qui est publié dans les contrats de service.

- **Service de ré-utilisation:** Les services ont le potentiel d'être réutilisés. Ces services réutilisables sont conçus de manière à ce que la logique de leur solution soit indépendante de tout processus métier ou technologie.

- **Service autonome:** Les services exercent un haut niveau de contrôle sur leur environnement d'exécution sous-jacent.

- **Service statelessness:** Les services minimisent la consommation de ressources en différant la gestion des informations d'état lorsque cela est nécessaire.

- **Service de découverte:** Les services sont complétés par des métadonnées de communication permettant leurs découvertes et leurs interprétations efficaces.

- **Service de composabilité:** Les services sont des participants effectifs à la composition, indépendamment de la taille et de la complexité de la composition.

2.3.2.2 Approches de SOA

L'architecture SOA est implémentée selon deux approches :

- **Bottom Up :** Cette approche est initiée par Krafzig et Slama (Krafzig, Banke, & Slama, 2005), le principe est de commencer par un petit groupe et en ajouter jusqu'à on obtient une grande entreprise. Cette approche est dite aussi approche ascendante et qui reconnait l'importance de l'exécution réelle des applications (Berman, 1978, p. 156)(Berman, 1978). La conception

ascendante exploite le fait que les utilisateurs réagissent aux exigences en fonction de leurs expériences et connaissances pour procéder à des modifications et/ou amélioration du système.

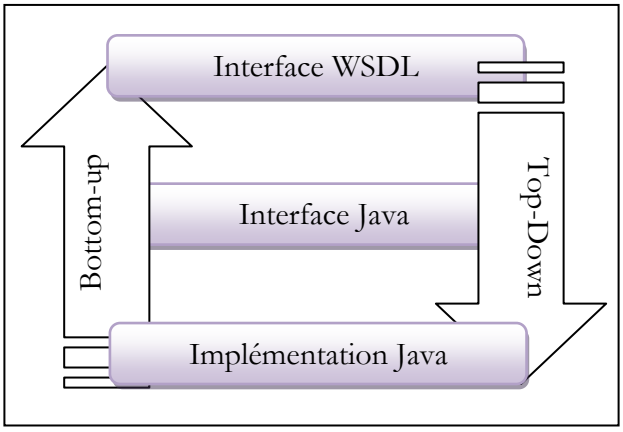

Figure 2.7- Approches SOA.

Ce comportement peut être d'un apport positif pour le développement du projet informatique. Cependant, on reproche à cette méthode de perception, la dépendance locale des actions d'évolution aux ressources qui sont associées à ces applications. Les théoriciens de Bottom-Up sont critiqués pour la surévaluation du degré d'indépendance locale de la part des décideurs car la mise en œuvre ne pourrait pas fonctionner sans les ressources et la structure institutionnelle fournies par les planificateurs centraux. Les ressources financières et humaines peuvent avoir un impact important sur le processus de mise en œuvre (Matland, 1995).

Table 2.3- Comparaison entre approches ascendants et descendantes.

Approches ascendantes (Bottom-up)	Approches descendantes (Top-Down)
Initiatives à court termes	Initiatives à long termes
Construire une solution avec des exigences immédiates (concevoir par assemblage)	Se concentre sur les propriétés du système du domaine entier (domaine métier, domaine technique, infrastructure, etc.).
Fortement aligné pour les objectifs locaux	Créer de la valeur ajoutée en exploitant des ressources disponibles
Rentabilise l'utilisation des ressources disponibles par improvisation par exemple	Bénéficier indirectement des relations inter-domaine, coûts et risques
Lien directe entre les avantages locaux, les coûts et les risques.	Il est souvent difficile de créer/maintenir une analyse de rentabilisation pour un investissement adéquat dans les ressources et l'infrastructure
Ne donne pas assez d'attention aux opportunités offertes à long termes.	Souvent difficile à démontrer le retour sur investissement

[Source: (Rodriguez, Crasso, Mateos, Zunino, & Campo, 2013)]

- Top Down : Cette approche aussi est présentée par Krafzig et Slama. On peut dire que c'est l'inverse du Bottom-Up. Le Top-Down est la décomposition d'un système ou un problème jusqu'à l'obtention d'un petit service de base. Elle est dite approche descendante car le point de

départ est la décision faisant autorité; comme son nom l'indique, les acteurs situés au centre sont considérés comme les plus pertinents pour produire l'effet souhaité (Matland, 1995).

Les principaux acteurs sont les concepteurs ou les architectes qui sont chargés de formuler le modèle efficace qui convient au type de problème confronté. Cela exige en effet une formulation claire et cohérente des objectifs à atteindre. Cependant, la critique principale à laquelle l'approche descendante est confrontée est que ses modèles peuvent être assujettis à des modifications lors et après implémentation. La Figure 2.7 schématise le sens de développement des différentes approches de SOA. Tandis que la Table 2.3 synthétise et compare les deux approches.

2.4 Modèles de qualité d'une architecture logicielle

Un grand nombre d'études dans la littérature se sont consacrées à l'analyse de l'aspect qualitatif du système en se basant sur des caractéristiques qualitatives de l'architecture. En effet, l'abstraction du système fournit plus de facilitation pour l'évaluation de plusieurs attributs significatifs de la qualité du système. Dans cette section, nous nous restreindrons à aborder que les approches de modélisation de qualité puisqu'elles sont en étroite relation avec notre thématique. Cependant, l'évaluation de la qualité propose une grande variété de méthodes qui ont été réifiés dans des modèles de qualité.

2.4.1 Modèle de McCall

Le modèle proposé par McCall (Cavano & McCall, 1978) est l'un des tous premiers modèles qui a initié la recherche dans le domaine de modélisation de la qualité logicielle. Le modèle McCall est présenté en trois niveaux:

- Facteurs de qualités: représentent les attributs de première ordre ou de plus haut niveau qualifiant un système logiciel. Ils cernent les caractéristiques impactant immédiatement la qualité d'un système. Ces facteurs peuvent être en relation avec le processus de maintenabilité ou de fiabilité.

- Critères de qualités: viennent en second lieu dans la hiérarchie de la description de la qualité d'un système. Ils permettent de spécifier les caractéristiques en relation avec le même facteur. Cela permettra de bien faciliter le processus d'évaluation des facteurs. La décomposition des facteurs en critères introduit un raffinement dans l'évaluation d'un système formalisant un niveau plus bas de description. La Table 2.4 présente les deux niveaux de description ainsi que la liaison entre chaque facteur et les critères qui le définissent. Par exemple, le facteur d'efficacité est défini par l'évaluation de l'efficacité des deux critères d'exécution et de stockage.

- Métriques d'évaluation: représente le niveau le plus bas dans la hiérarchie d'évaluation de la qualité. Ils permettent à cet effet de mesurer chaque critère de qualité. Le modèle ne présente pas l'association des critères avec les métriques associés et laisse donc la solution ouverte aux utilisateurs pour les adapter chacun selon la spécificité du domaine ou du système ou à ses propres objectifs.

Table 2.4- Modèle de McCall: Facteurs et critères de qualité.

Facteurs→ / ↓ Critères	Correction	Fiabilité	Efficacité	Intégrité	Utilisabilité	Maintenabilité	Testabilité	Flexibilité	Portabilité	Réutilisabilité	Interopérabilité
Traçabilité	☑										
Complétude	☑										
Consistance	☑	☑									
Précision		☑									
Tolérance aux erreurs		☑									
Efficacité d'exécution			☑								
Efficacité de stockage			☑								
Contrôle d'accès				☑							
Audit d'accès				☑							
Opérabilité					☑						
Formation					☑						
Communicative					☑						
Simplicité						☑	☑	☑	☑	☑	
Concision						☑					
Instrumentation							☑				
Auto-Descriptive						☑	☑				
extensibilité								☑			
Généralité								☑		☑	
Modularité						☑	☑	☑		☑	☑
Independence logiciel									☑	☑	
Independence machine									☑	☑	
Similitude de communication											☑
Similitude de données											☑

2.4.2 Modèle de la Norme ISO-9126

L'Organisation Internationale de Normalisation (ISO) et la Commission Electrotechnique Internationale (CEI) ont définit la norme multi-parties ISO/IEC9126 dont l'objectif est de spécifier les différentes caractéristiques et métriques qui leur sont associées pour un produit logiciel.

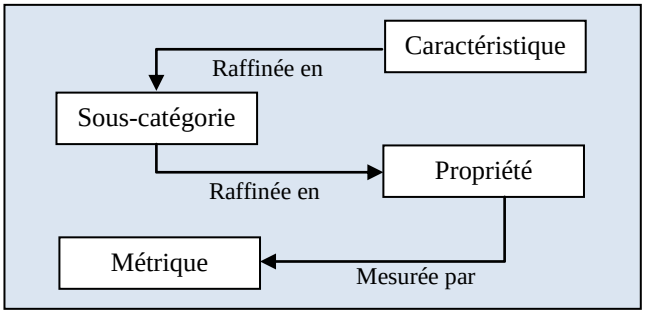

Figure 2.8- Méta-modèle ISO 9126.

Ces éléments peuvent être utilisés à la fois pour évaluer et définir les objectifs en termes de qualité logicielle. La norme ISO-9126 (Iso, 2001) définit un méta-modèle représenté en quatre composants présentant ainsi quatre niveaux d'évaluation et précise en plus la relation entre les différents niveaux comme il est schématisé dans la Figure 2.8.

- Premier niveau: représente les caractéristiques correspondant aux facteurs introduis par McCall. Elles présentent donc les éléments essentiels qui influencent directement la qualité du système. On trouve de la même façon les caractéristiques de fiabilité, d'efficacité, d'utilisabilité, de fonctionnalité, maintenabilité et de portabilité comme le schématise la Figure 2.9.

- Deuxième niveau: concerne les sous catégories qui sont introduites pour raffiner les catégories du premier niveau. A cet effet, elles facilitent la compréhension de chaque caractéristique pour assurer une bonne évaluation. Il est notable que certaines sous catégories dans le modèle ISO 9126 ont été considérées dans le modèle de McCall comme des caractéristiques telle que l'interopérabilité.

- Troisième niveau: reflète les propretés mesurables des sous catégories, de même ces valeurs ne sont pas identifiées de façon bien précise mais le modèle offre une méthodologie instanciable selon les objectifs fixés.

- Quatrième niveau: est similaire au troisième niveau du modèle McCall et qui décrit les valeurs afférentes aux propriétés de chaque sous-catégorie. Le modèle laisse libre l'utilisateur pour fixer et décrire les valeurs selon la spécification du domaine ou des objectifs.

Plusieurs travaux se sont inspirés de ce modèle pour proposer un ensemble de propriétés et de métriques ainsi que la relation entre ses composants et les différents liens les associant. Par exemple, le modèle SAAM[1] (Kazman, Bass, Abowd, & Webb, 1994) utilise trois perspectives pour comprendre et décrire les architectures: la fonctionnalité, la structure et l'allocation. SAAM a utilisé ces trois perspectives de description d'architecture pour évaluer le système du point de vue de la modifiabilité. Cependant, SAAM peut également être utilisé pour évaluer d'autres attributs de qualité.

2.4.3 Modèle FURPS/FURPS+

Ce modèle étendu par Rational Software initialement introduit par Robert Grady (Grady, 1992) est organisé de la même façon que les modèles précédents mais n'est pas assez réputé que les premiers. FURPS définit les attributs de qualité suivants :

- Fonctionnalité: Elle comprend un ensemble de caractéristiques, de capacités et de sécurité.

- Utilisabilité: Elle considère essentiellement les facteurs humains, ergonomiques, cohérence des interfaces utilisateurs, la documentation pour l'utilisateur, assistance en ligne et des outils de formation.

[1] A Method for Analyzing the properties of Software Architecture.

- **Fiabilité:** C'est la mesure de fréquence et sévérité d'échec, la prévisibilité, l'exactitude, la récupérabilité et temps moyen d'échec qui est le temps moyen des temps de bon fonctionnement.

- **Performance:** Elle représente l'évaluation des conditions associées aux besoins fonctionnels comme la vitesse, l'efficacité, le temps de réponse, le temps de rétablissement, le débit et l'exactitude.

- **Supportabilité:** Pour évaluer l'extensibilité, l'adaptabilité, maintenabilité, la testabilité, aptitude de configuration et la capacité de service.

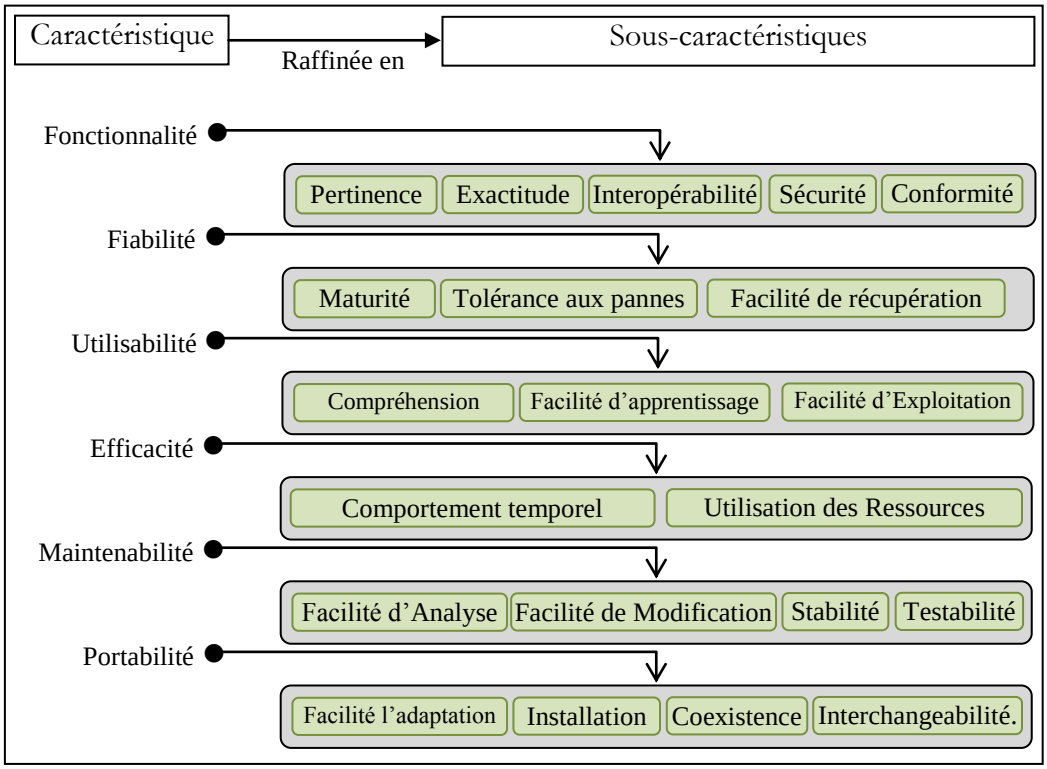

Figure 2.9- Présentation du modèle de la Norme ISO-9126

Les principales catégories adoptées dans FURPS, présentées à l'aide de la Figure 2.10, sont regroupées selon les préoccupations des utilisateurs et qui sont :

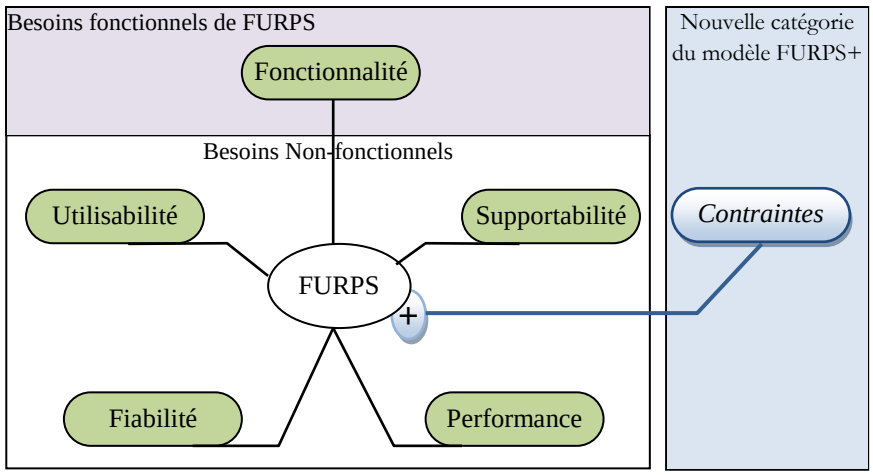

Figure 2.10- Présentation du modèle PURPS/FURPS+.

[*Source : par inspiration de (Al-Badareen, Selamat, Din, Jabar, & Turaev, 2011)*]

- Fonctionnelles (F) : qui sont définies par les entrées et les sorties attendues,

-Non fonctionnelles (URPS): pour définir l'Utilisabilité, la fiabilité (R), la Performance et la Supportabilité.

Le modèle a fait l'objet d'une extension par IBM rational software est présenté comme FURPS+ par l'ajout d'une nouvelle catégorie « contraintes » (+) (Al-Badareen, et al., 2011). Elle désigne l'ensemble des préoccupations, leurs interfaces et des préoccupations physiques.

2.4.4 Modèle SOAQE

SOAQE, Software Oriented Architecture Quality evaluation, est une méthode semi automatique pour l'évaluation de la qualité des architectures orientées services depuis la décomposition de l'architecture orientée service en attributs jusqu'à l'octroie des résultats d'évaluation (Belkhatir, Oussalah, & Viguier, 2012). Le modèle adopté s'inspire du modèle à trois niveaux de hiérarchie tout comme celui de McCall (McCall, 1994) auquel ils lui ont ajouté le niveau supérieur de point de vue qualité comme le montre la Figure 2.11.

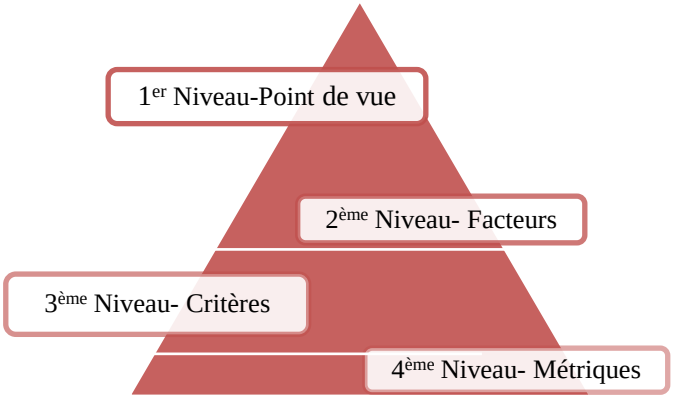

Figure 2.11- Hiérarchie du modèle SOAQE.

Les auteurs ont défini le triptyque composée de la réutilisabilité, la compossibilité et la dynamicité comme trois facteurs de base de la vue qualitative permettant de diriger la définition des différents paradigmes architecturaux à savoir l'objet, le composant et le service. Puis chaque facteur est étudié en profondeur pour dénicher les différentes caractéristiques dans le domaine SOA. La Figure 2.12 présente les caractéristiques du triptyque considéré en mettant en lumière les différents niveaux des critères de qualité. Le modèle SOAQE pourra convenir à la plupart des autres modèles de qualité puisqu'il convient parfaitement à l'ensemble des critères de comparaison de qualité relatif à l'aspect technique de l'architecture logicielle. La méthode à été validée par une étude de cas d'un projet existant (BeOtic[1]).

[1] BeOtic est un éditeur de logiciel Innovants de solution de gestion de projetcollaborative, pour plus de détails consulter *http://www.beotic.com/index.php?option=com_content&view=article&id=3&Itemid=7&lang=fr*

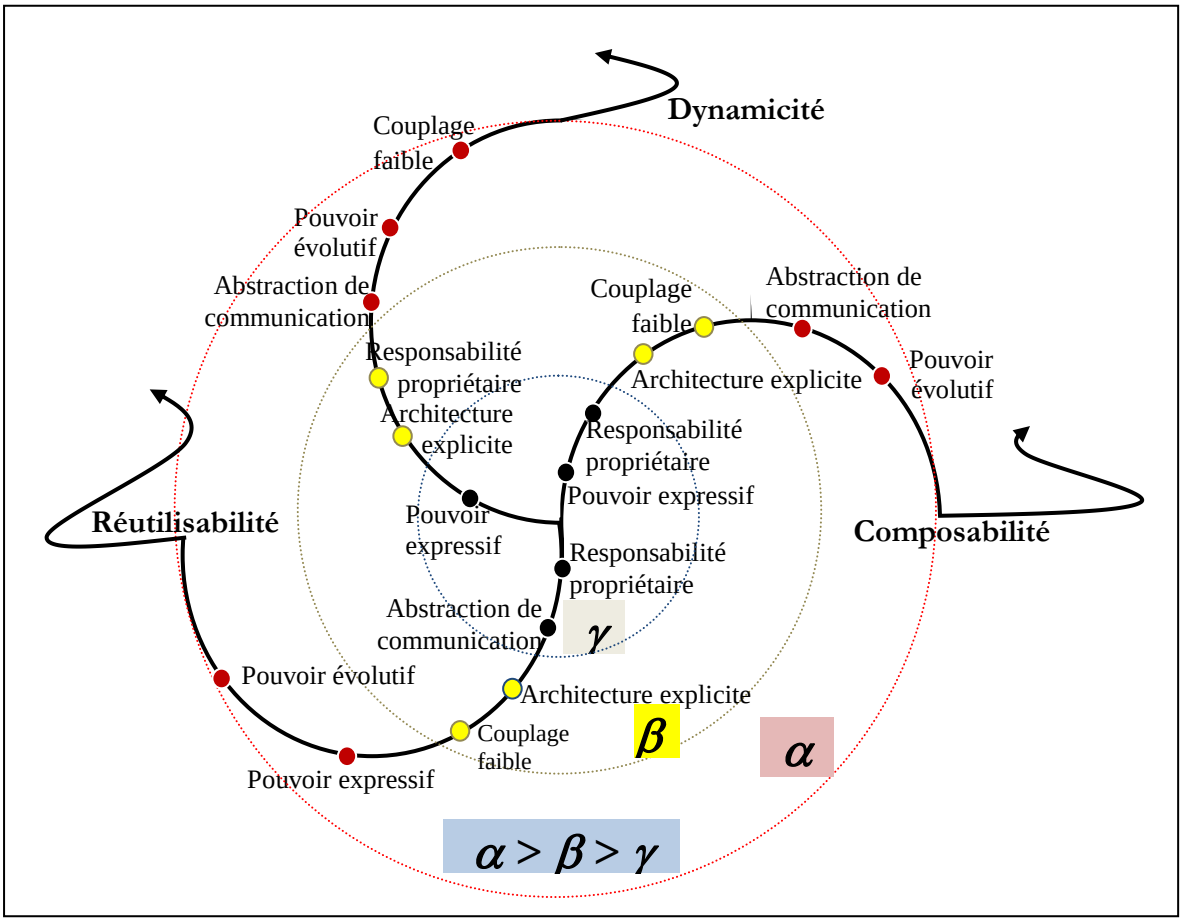

Figure 2.12- Expression du triptyque du modèle SOAQE.

[*Source : par inspiration de (Kheir, et al., 2013)*]

2.5 Conclusion

Dans ce chapitre, nous avons présenté un bref aperçu de l'architecture logicielle dans les deux domaines CBSE et SOA. Les concepts les plus importants ont été introduits afin de garantir une bonne compréhension du contexte de recherche de la thèse. Nous avons expliqué comment l'architecture logicielle s'est progressivement imposée et joue désormais un rôle vital dans le développement des logiciels.

Nous avons également consacré une section pour l'évaluation de la qualité logicielle, pour expliquer l'importance de maintenir une architecture de qualité qui repend aux performances exigées par les organisations. La qualité d'architecture logicielle est soumise à des processus complexes impliquant plusieurs groupes d'intervenants pour la simple raison de satisfaction d'utilisateurs. Cet aspect qualitatif reflète en effet la nécessité d'assurer une efficacité et une précocité d'évaluation afin de supporter des choix et des décisions qualitatives.

Dans le chapitre suivant, nous allons plonger dans le vif de notre thématique de description de l'évolution logicielle, afin encadrer et bien cerner le contexte dans lequel notre travail de thèse s'insère.

CHAPITRE 3

Evolution et référentiels d'évolution logicielle

« L'innovation, c'est une situation qu'on choisit parce qu'on a une passion brûlante pour quelque chose. »
- Steve Jobs (1955-2011)

3.1 Introduction

Dans la suite de notre thèse et après avoir introduit le contexte et les définitions relatifs aux principaux concepts inhérents au domaine des architectures orientées composants et services, nous enchainerons dans le présent chapitre avec le paradigme d'évolution qui est de nos jours une problématique dont les architectes s'en soucient de plus en plus durant la conception et l'implémentation des systèmes. Sachant que l'évolution logicielle constitue un domaine à part, nous allons se focaliser précisément dans ce travail à la problématique de l'évolution architecturale des logiciels mais avant tout, nous réservons la première section à la description du domaine d'évolution. Ensuite, nous présenterons les référentiels les plus connus dans le domaine des architectures logicielles et enfin nous positionnons notre choix sur le Framework d'inspiration qui a guidé notre proposition.

3.2. Evolution logicielle

Dans cette section, nous essayons d'esquisser les contours l'évolution et les concepts d'évolution logicielle avant d'aiguiller vers l'évolution architecturale en tant que concept clé.

3.2.1 Synopsis d'évolution

L'évolution est un concept qui s'est connu de plus en plus en science humaine et sociale. Il a été utilisé pour représenter un critère de croissance des objets matériels ou immatériels comme l'évolution démographique, évolution des espèces, évolution des prix, …etc. En ingénierie logicielle, ce paradigme a connu une prolifération très accélérée que nous résumons par la suite à

la base de leurs temps d'apparitions. Au début, l'évolution en informatique s'est restreinte essentiellement pour poursuivre l'innovation technologique en termes de matériels informatiques, de composants et de langages de programmation. Ce n'est qu'au vingtième siècle que le logiciel a été reconnu comme une structure pouvant être soumise à des mises à jour correctives ou prédictives suite aux fameuses lois de Lehman (Meir M Lehman, 1980). En effet, le domaine d'évolution logicielle est apparu dont l'objectif était le traitement des processus liés à la modification et le changement des programmes suite à un changement de besoins ou d'environnement (Herraiz, Rodriguez, Robles, & Gonzalez-Barahona, 2013). La Table 3.1 montre que le raffinement des lois vient après de longues intenses années de recherche.

Table 3.1- Evolution des études de Lehman sur l'évolution logicielle.

Année	Objet de l'étude	Limites associées	Nbr lois
1974	Etude empirique : recherche des propriétés invariantes des classes durant le développement logiciel.	Les lois sont très génériques et peuvent par conséquent engendrer quelques confusions.	3
1978	Elaboration de plusieurs études empiriques par utilisation des méthodes statiques mentionnées dans les travaux précédents.	Présente qu'une seule étude de cas OS/360 d'IBM.	5
1980	L'évolution est considérée comme une ressource fondamentale pour la reconstitution de la traçabilité d'un système logiciel. Introduction de la taxonomie SPE type pour l'évolution des systèmes logiciels.	Concentration principalement sur le E-type au détriment des deux autres types (P-type and S-type).	5
1996	Raffinement des lois d'évolution	Depuis les lois restent non modifiées.	8

3.2.1.1 Avant 1974

Initialement les lois d'évolution ont été de nombre de trois comme publié dans (Meir M Lehman & Belady, 1985), introduisant trois principes de base pour l'évolution des logicielles (Table 3. 2):

- Les deux premières lois représentent le fondement de l'opération d'évolution des systèmes logiciels. En pratique, les programmes ne sont pas stables et doivent changer continuellement. Ce phénomène reconnait de plus grande envergure lorsqu'il s'agit de projet nécessitant plusieurs niveaux managériaux au cours de son cycle de vie. Notons que la deuxième règle a connue deux différentes appellations: entropie et complexité. La première utilisée avant 1974, évoquée en guise du concept en vogue pendant cette période, pour mettre sous lumière les difficultés rencontrées à l'issue des opérations de changement tels le désordre et la dégradation des structures des programmes modifiés. En conséquence, les systèmes ayant subit ces changements deviennent par la suite plus difficiles à comprendre, à gérer et à être modifiés une autre fois. La troisième loi propose une approche statistique pour l'étude de l'évolution logicielle. L'idée est d'introduire une méthode stochastique autorégulatrice des tendances croissantes (*Stochastic Growth Trends*). Des méthodes de régression, de corrélations automatiques et les séries temporelles ont été proposées pour l'analyse de l'évolution. Mais en

pratique, le recours de ces méthodes est un peu difficile du fait qu'elles consomment un temps colossal pour l'analyse.

Table 3. 2- Lois d'évolution jusqu'à 1974.

Lois	Désignation
1	**Loi du changement continu** Le logiciel tourne dans un environnement soumis à plusieurs nouvelles exigences et doit s'en adapter. Ce qui rond parfois son remplacement plus économique que sa maintenance.
2	**Loi de l'entropie croissante** Le nombre des changements introduit dans le système sont souvent pharaonique ce qui engendre l'extension du code et des procédures plus complexes pour le maintien de la documentation. En bref, le système devient au fil du temps difficile à modifier et/ou à comprendre.
3	**Loi statistique d'évolution** La tendance d'évolution de logicielle à travers certaines propriétés du système montre qu'elle est stochastiquement mesurable en terme de temps et d'espace par l'adoption de méthodes statistiques souples.

3.2.1.2 Avant 1978

De plus, l'accès aux données des projets est sanctionné par la réticence des managers par des soucis de confidentialité, de peur et de sécurité. Les chercheurs ont données de grandes importances aux méthodes statistiques pour valider ces lois (Belady & Lehman, 1976; M. Lehman & Parr, 1976; Meir M Lehman, 1978). Le résultat de ces études a permis de mettre à jour les lois et d'en ajouter deux autres lois comme illustré dans la Table 3.3. Nous remarquons qu'à cette édition, les descriptions de lois ont été modifiées par exemple la loi de l'entropie croissante a été remplacée par la complexité croissante. De plus, deux nouvelles lois ont été introduites (la quatrième et cinquième) suite à des constations statistiques sur un ensemble de métriques (ajout, modification et suppression des modules) utilisées pour l'évolution du système OS-360 de IBM. La décennie 80 a été caractérisée par une stabilité des lois avec un souci de les valider sur des programmes de différentes tailles. Le constat a montré que l'activité d'évolution change selon que le programme est de grande ou petite taille, car les programmes sont supervisés et maintenus par une équipe ce qui favorisera le processus de la rétroaction (feedback). En réalité, deux programmes de même taille ne peuvent pas toujours avoir le même comportement évolutif. Pour cette raison, le domaine d'application des lois a été changé de l'application sur les longs programmes vers un schéma de classification appelé SPE (Specific-Problem- Evolutionary). Ce schéma repose sur trois classes de programmes:

- *S-type* (Specific type) : désignent des types spécifiques de programmes qui sont une vue statique de spécification et dont leurs validité ne peut pas être formellement prouvée.
- *P-type* (Problem Solving) : sont des programmes rédigés pour résoudre les problèmes qui peuvent être exprimés formellement mais leurs implémentation est difficile.
- *E-type* (Evolutionary type) : sont des programmes formalisant des réflexions humaines ou une conception du monde réel. Un E-type englobe la majorité des logiciels et représente la catégorie qui a eu une large acceptation à travers les études établis par la communauté du génie logiciel.

Table 3.3- Les lois jusqu'à 1978.

Lois	Description
1	**Loi du changement continu** Avec les nouveaux besoins utilisateurs ou environnementaux, un programme s'use et présente un disfonctionnement ce qui réduit son utilisabilité. La décision de son remplacement est relativement liée à l'évaluation des coûts effectifs engendrés par sa maintenance.
2	**Loi de complexité croissante** Le nombre de ligne de code et la structure de code implémentant le logiciel sont systématiquement liés à la complexité du logiciel. Toute modification va forcement influencer ces structures ce qui favorise la détérioration des structures initiales et entache la compréhension et la lecture du code. A moins que des outils spécifiques ne soient utilisés pour réduire sa complexité ou à sa maintenance.
3	**Loi de croissance statistique régulière** L'utilisabitilé du logiciel impose que celui-ci soit capable d'intégrer de nouvelles fonctionnalités afin de préserver la satisfaction de ses utilisateurs aussi longtemps que possible dans son cycle de vie.
4	**Loi du taux d'activité invariante** Cette loi statue que le taux général de l'activité dans un grand projet de développement logiciel est quasiment invariant.
5	**Loi de la limite de la croissance incrémentale** Les modifications dans un programme fonctionnel nécessitent une planification préalable des délais pour sa mise à la disposition de l'utilisateur final, autrement un disfonctionnement d'exécution menacera son fonctionnement voire même sa fiabilité. De plus, cela engendrera forcement des coûts supplémentaires inattendus qui accélèrent la décision de sa mise hors fonctionnement i.e. la mort du système en termes de notion de cycle de vie. Cela signifie que le système développe une augmentation moyenne qui caractérise la croissance logicielle qui, en cas de dépassement, stimule des surcoûts et des délais importants nocifs à la qualité du logiciel.

Il est à noter, que malgré la forte reconnaissance de la catégorie E-type. Les deux autres catégories et la taxonomie elle-même n'ont suscité que peu d'attention de la part des chercheurs (Cook, Harrison, Lehman, & Wernick, 2006).

3.2.1.3 Années 85

Les années 80 sont caractérisées par la modification des lois (3, 4 et 5), comme il est indiqué dans la Table 3.4, pour montrer que les E-types sont autorégulateurs en se basant sur les notions statistiques de variance et de tendance. De plus, les chercheurs ont montré l'intérêt de l'évaluation par le biais des mesures et de métriques spécifiques aux produits et aux processus. La reformulation de la désignation de la loi malgré la même signification, accentue sur l'importance de cette propriété pour l'évolution des logicielles. Le contenu de la quatrième loi reste le même avec la substitution du concept grand programme avec programme E-type. La cinquième loi, indique que le taux d'activité d'un projet reste invariant durant son exploitation, ce qui est étroitement lié à la quatrième loi (conservation de la stabilité organisationnelle). Cette loi émane du fait que le taux de changement entre les versions est constant cela dit qu'on ne peut pas trouver deux versions considérablement modifiées sans qu'il y est d'autres versions intermédiaires.

Table 3.4- Modification des lois d'évolution (3-5) au 1985.

Lois	Description
3	**Loi fondamentale de l'évolution des programmes** L'aspect dynamique impose une régulation automatique (Self-regulation) aux programmes en respectant les variances de tendances émergentes du système. Cela exige la détermination des attributs globaux du projet et du système en utilisant des méthodes statistiques.
4	**Conservation de la stabilité organisationnelle** Cette loi stipule que le taux d'activité globale afférant au programme de développement E-type est statistiquement inchangeable.
5	**Conservation de la familiarité** Durant l'exploitation du logiciel, le contenue des versions consécutives du logiciel en termes de modifications, ajouts et suppressions montre statistiquement que l'activité est invariante.

3.2.1.4 A ce jour

Les années 90 sont caractérisées par le développement du principe donnant jour à une large théorie pour l'évolution logicielle (Manny M Lehman & Ramil, 2002) présentée par la Table 3.5.

Table 3.5- Formulation courante des lois d'évolution.

Lois	Description
1	**Loi du changement continu** Un système E-type doit s'adapter aux exigences continuellement. Autrement il s'érode progressivement induisant à une non-satisfaction des utilisateurs ce qui influera automatiquement sur la qualité et la fiabilité du logiciel.
2	**Loi de complexité croissante** Autant qu'un E-type subit des modifications, sa complexité augmente. Cela se traduit par une difficulté pour faire évoluer ces nouvelles structures à moins que des actions de préservations et de réductions soient planifiées, comme il a été déjà mentionné.
3	**Loi de la régulation automatique** L'évolution globale d'un système E-type est régulée automatiquement en réponse à des feedbacks.
4	**Loi conservation de la stabilité organisationnelle** Le taux d'activité, d'une organisation évoluant en système logiciel E-type, justifie statistiquement une stabilité durant son temps exploitation ou durant les phases de son cycle de vie.
5	**Loi conservation de la familiarité** En général, la croissance progressive des systèmes E-type est limitée par la nécessité de maintenir la familiarité.
6	**Loi de croissance continuelle** Durant le cycle de vie d'un système E-type, la satisfaction de ses utilisateurs tend a diminué. IL est donc opportun que le dit système doit assurer la faculté d'amélioration de ses activités opérationnelles.
7	**Loi de la qualité déclinante** Nouveau code ou nouvelles structures peuvent engendrer des dégradations dans les performances du système influençant ainsi sa qualité. Sauf, si elles sont rigoureusement planifiées pour de telles améliorations.
8	**Loi de la rétroaction du system (feedback)** Les processus d'évolution de systèmes E-type sont des systèmes de rétroaction multi-niveaux, multi-boucles et multi-agents.

L'incertitude a conduit les auteurs d'introduire d'autres lois additionnelles sur l'évolution intégrant les relations d'interactions entre incertitude et le domaine d'application du logiciel (l'ingénierie logicielle, le processus logiciel et leurs supports) sans avoir autant donner d'amples précisions dans le travail (Meir M Lehman, 1991).

3.2.2 Définitions d'évolution logicielle

Plusieurs recherches ont données des définitions qui s'alignent dans la même vision que celle présentée par Lehman. Nous citons :

Définition 1- Selon (Kemerer & Slaughter, 1999) l'évolution logicielle est définie comme :

"Software evolution refers to the dynamic behavior of software systems as they are maintained and enhanced over their lifetimes. Software evolution is particularly important as systems in organizations become longer-lived".

Cette définition stipule que la maintenance désigne les activités menées à tout moment après la mise en place du nouveau projet. Alors qu'une évolution logicielle consiste à examiner le comportement dynamique des systèmes et la manière dont ils évoluent. Elle englobe alors les opérations de maintenances et d'améliorations du système pendant son cycle de vie. Cette définition montre en clair que le processus d'évolution est plus significatif que celui de la maintenance.

Définition 2- La définition de (Bennett & Rajlich, 2000) a introduit l'importance de l'évolution des architectures logicielles:

"Both software architecture and software team knowledge make evolution possible. They allow the team to make substantial changes in the software without damaging the architectural integrity".

Cette définition montre que l'évolution touche aussi à l'architecture logicielle et de ce fait les intervenants doivent être conscients des impacts des modifications pour assurer l'intégrité du système.

Dans ce qui suit, nous entendons par évolution architecturale, toute modification subite par un logiciel au niveau d'un ou plusieurs éléments qui le composent. Ces modifications se traduisent généralement par des opérations de mise à jour habituelles telles que la modification, la création ou l'ajout ou la suppression des éléments, et qui sont souvent connues dans le jargon par les opérations d'évolution.

3.3 Approches d'évolutions existantes

3.3.1 Approches d'évolution basées chemins d'évolutions

Dans ce type d'évolution, l'état de l'évolution représente une architecture intermédiaire ou de transition. Ces transitions permettent de faire passer un état de départ dit initial à un état de final dit architecture cible. L'état initial présente la description de l'architecture avant l'opération d'évolution tandis que l'état final capture l'architecture reflétant le système après la fin du processus d'évolution. Toute architecture permettant de capturer un état transitoire servant de passer d'un état à un autre est appelée architecture intermédiaire. Un état représente une

description complète de l'architecture à l'aide d'un langage de description d'architecture adéquat (Garlan, Monroe, & Wile, 1997) qui présente la façon la plus appropriée pour cette description (Baranov, 2017; Medvidovic & Taylor, 2000). Dans les approches basées chemins, l'évolution de l'architecture est formalisée à travers un graphe orienté dans lequel les sommets représentent les composants de calculs et de stockages des données tandis que et les liaisons représentent les connecteurs entre composants(Perry, 1992; Shaw, DeLine, & Zelesnik, 1996). De plus, d'autres éléments fonctionnels ou non fonctionnels peuvent être intégrés au graphe comme les interfaces et les points de liaisons (ports-rôles) et certaines autres propriétés supportant des contraintes et permettant certaines analyses sur l'architecture. Ces contraintes architecturales représentent pour le graphe d'évolution les contraintes de chemins d'évolution. Il est important de bien choisir le langage de description d'architecture applicable au type associé au problème à résoudre. Alors, si un ADL ne supporte pas une fonction ou une opération sur un élément architectural, il n'est plus possible d'utiliser ce type de fonction pour les éléments de l'architecture. Sans oublier aussi, la nécessité de disposer d'un bon outil conceptuel pour manager et analyser les différents éléments architecturaux. Ces outils respectent des modèles bien définis permettant la visibilité, l'accessibité et la modifiabilité de ces éléments. Certaines approches de chemins se focalisent essentiellement sur les ADL formels (Acme par exemple), mais peu entre elles prennent en considération les langages de modélisation non formels comme UML et SysUML(Friedenthal, Moore, & Steiner, 2008; Platt & Thompson, 2019). Le travail de (Barnes, Pandey, & Garlan, 2013) sur l'évolution logiciel a introduit l'idée de la planification de l'évolution par l'utilisation de plusieurs vues architecturales comme elles ont été définies par (Clements et al., 2002). Elles représentent plusieurs vues architecturales représentant des architectures intermédiaires de l'architecture à évoluer. Cette technique permet en effet aux architectes d'inclure les vues selon les contraintes formulées par l'architecture et donc de formuler le plan de l'évolution qui est sensé être appliqué en réponse de l'analyse menée à priori. En outre, les d'états d'évolution peuvent être annotés avec des propriétés supplémentaires facilitant la documentation et l'utilisabilité par exemple des notes concernant le délai toléré pour élaborer une transition d'évolution.

3.3.2 Evolution par opérateurs d'évolution

Les opérations de transformation d'une architecture sont regroupées en opérateurs d'évolution. Les transformations sont par exemple l'ajout d'un composant, la suppression d'un port ou la modification d'une propriété d'un élément architectural ... etc.

Nous distinguons alors les opérations d'encapsulation des composants sous forme de service pour les architectures orientées services. Les opérateurs incarnent les descriptions relatives aux:

- Modifications structurelles de l'opérateur, par exemple, envelopper un composant en tant que service nécessite un ensemble de transformations composées pour décrire les changements liés à cet opérateur.
- Pré-conditions d'application de l'opérateur.
- Informations supplémentaires pour établir des analyses par exemple, information sur le temps et le cout nécessaire pour cet opérateur.

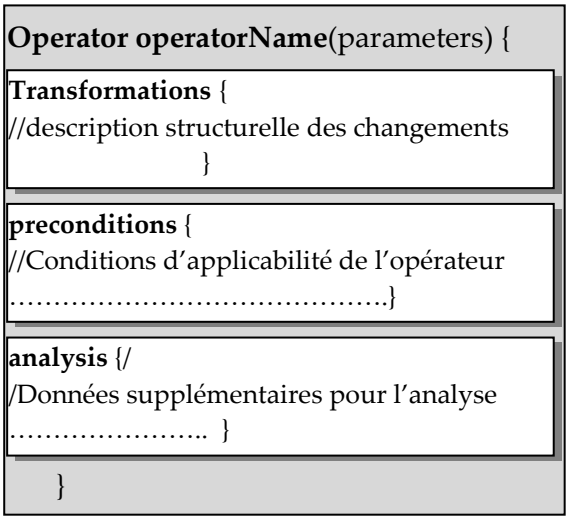

Figure 3.1- Structure d'un opérateur d'évolution.

Conformément à la Figure 3.1, les pré-conditions définissent les conditions à satisfaire pour que l'exécution de la transformation soit déclenchée, c'est une restriction de l'opérateur. Elles représentent les contraintes architecturales à priori. Il est important de noter que l'expression des pré-conditions traduisent un nombre spécifique de points de vues c.-à-d. que les pré-conditions deviennent de plus en plus complexes lorsqu'on désire couvrir plusieurs vues architecturales. Par exemple, *on peut forcer le remplacement d'un composant que lorsqu'il exprime un certain type de port ou lorsqu'il contient un type bien précis de connecteurs*. Enfin, des données supplémentaires d'analyse peuvent être incluses pour supporter d'éventuelles contraintes et de pouvoir établir des analyses sur l'évolution. Par exemple, *un opérateur peut inclure des informations telles que l'effort requis pour sa réalisation, son coût, ses implications, ... etc*. Ces informations sont moins structurées par rapport aux deux autres parties (pré-conditions et transformations). De même, les informations d'analyses dépendent des vues et des styles considérés et deviennent par conséquent plus complexes. Cette manière de définition des opérateurs d'évolution permet de définir une transition d'évolution entre les différents états d'évolution, comme l'expose la Figure 3.2.

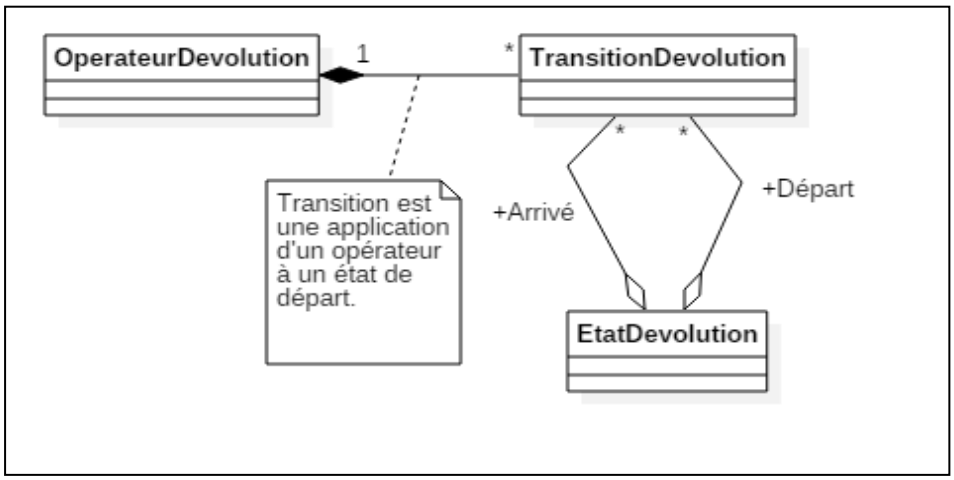

Figure 3. 2- Modèle des transitions d'évolution.

Les transformations reflètent les modifications opérées sur les éléments architecturaux de base et qui sont évidemment des caractéristiques du langage de description avec lequel elles ont été développées. Par exemple, (Oreizy, Medvidovic, & Taylor, 1998) préconisent un gestionnaire d'évolution d'architecture (AEM) pour gérer toutes les opérations d'évolution à travers le modèle architectural. Une modification est exprimée comme une transaction composée d'au moins de deux opérations de base. Tout changement s'effectuant de manière saine est considéré comme atomique. Les opérations de la transaction permettent la modification de la typologie de l'architecture à travers les opérations d'ajout, de suppression et de remplacement de composants et/ou connecteurs. Dans (Barnes, et al., 2013), les auteurs ont développé un pseudo-code permettant une spécification informelle des opérateurs d'évolution et qui permet d'exprimer et de composer des transformations simples à la base de la structure de définition d'un opérateur.

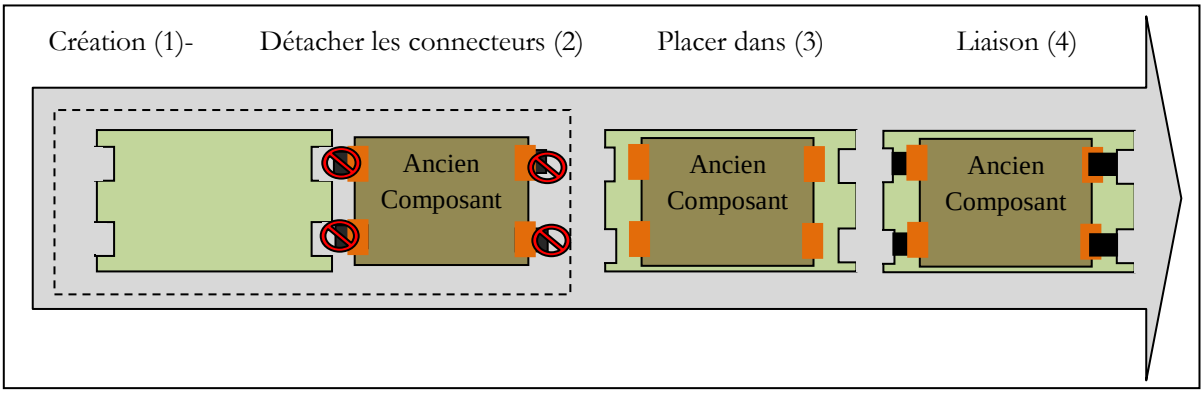

Figure 3.3- Etapes d'encapsulation d'un ancien composant dans un service.

Par exemple, *pour envelopper un ancien composant dans une enveloppe de service, il faut comme illustré dans la* Figure 3.3*:*

1. Créer un nouveau service d'encapsulation,
2. Détacher tous les connecteurs attachés aux ports de l'ancien composant,
3. Relier les ports au service d'encapsulation,
4. Placer l'ancien composant dans le service d'encapsulation et associer chacun des ses ports aux ports correspondants au service créé.

3.3.3 Evolution par évaluation de fonction

L'évolution d'une architecture peut être présentée sous forme d'un graphe contenant les différents chemins techniques que les architectures définissent par considération de toutes les contraintes d'évolutions. Une telle méthode permet d'illustrer les différents chemins valides pour que les architectes du système puissent décider du meilleur choix compte tenu de l'environnement d'évolution. Ici intervient le rôle important de l'emploi de fonctions d'évaluation pour guider le choix des bonnes solutions. Ces fonctions établissent une évaluation quantitative de la qualité et offrent une valeur estimative de la qualité qui est une valeur logique utilisée pour évaluer les contraintes d'évolution. La raison est qu'une valeur numérique peut être employée dans d'autres fonctions souvent numériques servant de raisonner sur un choix de chemin quelconque. Les contraints d'évolution préservent l'avantage d'être mieux adaptées en qualité de prédicat pour l'expression des règles de base du domaine et de réduire le graphe d'évolution pour

qu'il soit plus clair et plus utilisable. Par contre, les fonctions d'évaluation permettent d'effectuer un jugement sur les chemins candidats ce qui permet de guider la décision des architectes.

La Figure 3.4 montre une estimation du coût de trois chemins d'évolution facilitant la décision du choix. La sélection de chemin optimal pour ce type de graphe constitue une opération privilégiée pour les architectes. La fonction d'évaluation de qualité de chaque chemin permet de comparer et sélectionner le chemin le plus optimal selon les contextes professionnels et/ou selon les différentes vues des intervenants. La fonction peut être simple en considérant qu'une seule propriété de sélection comme elle peut être une combinaison ou une pondération de plusieurs attributs. Il est judicieux d'associer des valeurs à des transactions comme une combinaison des attributs d'effort, temps, coûts et autres. Les architectes font souvent recours à des fonctions de compromis entre temps/coûts pour choisir les nœuds les plus prioritaires dans le cas des nouvelles versions.

3.3.4 Evolution par graphe de transformation

La tâche principale d'un architecte réside dans l'élaboration d'un plan pour faire évoluer le système d'un état de départ à un état final désiré. Le plan est construit à partir d'un ensemble d'état de transitions que l'architecture logicielle doit parcourir pour épouser l'état final escomptée donnant ainsi naissance à plusieurs chemins reliant plusieurs nœuds. Ces nœuds représentent en fait les architectures intermédiaires et les nœuds de l'extrémité présentent le nœud de départ et cible. Les chemins reliant les deux extrémités représentent des chemins candidats ou des chemins solutions comme l'illustre la Figure 3.4.

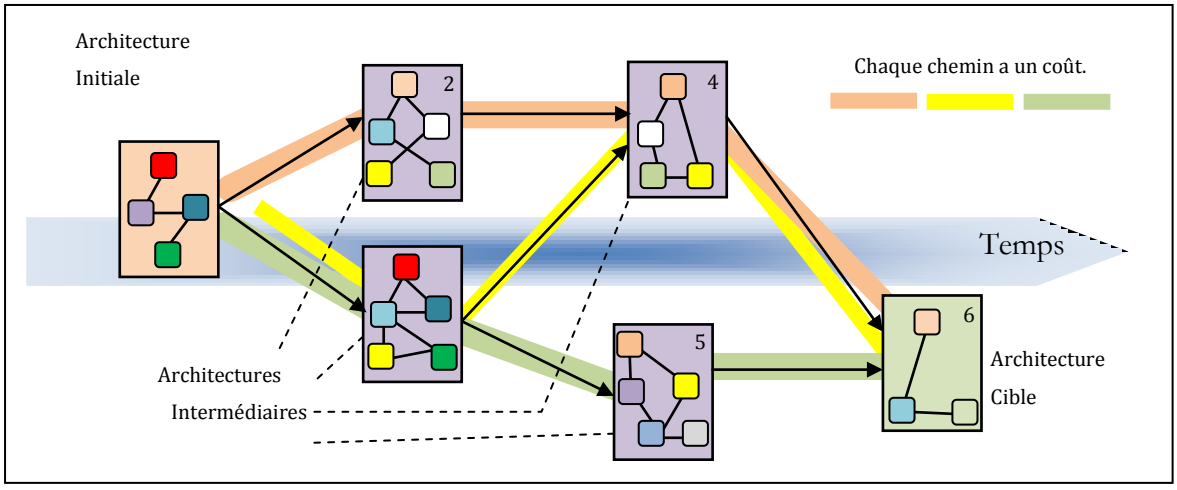

Figure 3.4-Evolution par graphe d'évolution.

A travers ce graphe orienté la tâche de l'architecte est bien simplifiée. Il s'agit de choisir le chemin le plus optimal pour faire évoluer l'architecture initiale dont il dispose. Le chemin est sélectionné au sens de la théorie des graphes puisque le graphe est orienté et construit le plan d'évolution à adopter par l'architecte. Le graphe présente ainsi plusieurs solutions candidates qui peuvent être considérées comme des scénarios d'évolution. Le choix du scénario repose sur l'ensemble de contraintes et d'expérience dont dispose l'architecte. Cela dit que les solutions réalisées peuvent construire une base de connaissances dont dépend l'automatisation du processus de planification, comme :

- La méthode de distinction entre les états intermédiaires concurrents,

- Détermination des impacts architecturaux à chaque transition,
- Identification des contraintes sur le domaine d'application du système,
- Identification des propriétés de chemin et les priorités entre eux.

3.4 Evolution des styles architecturaux

La notion de style présenté dans cette section s'inspire essentiellement du domaine des architectures orientées composants et orientées services. Nous nous intéressons ici à la notion de style dans la perspective que les architectures préconisent au même titre des style d'évolution qui permettent d'effectuer les mêmes opérations d'évolutions pour un même style.

Nous essayons donc de présenter un aperçu de l'état de l'art actuel des styles architecturaux en ce qui concerne les différents domaines pertinents pour la modélisation de l'évolution de l'architecture logicielle.

3.4.1 Cycle de style évolution

La pensée de l'évolution architecturale peut être formalisée par des experts en architecture surtout lorsque les évolutions peuvent être quasi similaire et/ou récurrentes entre plusieurs architectures. Les activités d'évolution peuvent alors être appliquées sur plusieurs familles d'architectures dites styles d'architectures et qui permettent entre autre de rentabiliser le coût d'évolution. L'évolution des styles est une bonne pratique pour les architectes car elle permet en plus de ce que nous avons avancé de capitaliser et partager l'expertise des architectes. Un style encapsule plusieurs informations sémantiques caractérisant un ensemble d'entités qui sont ventilées en trois grandes parties. Il est évident de chercher un corpus représentatif pour permettre une bonne compréhension des styles d'évolution. Cette compréhension passe par trois étapes essentielles selon (Le Goaer, Tamzalit, & Oussalah, 2010): la thésaurisation, extraction et la présentation.

3.4.1.1 Thésaurisation de l'évolution

Systématiquement, la thésaurisation présente le moyen propice pour modéliser un corpus relatif à un domaine quelconque. L'objectif est de capitaliser l'ensemble des expertises et de savoir les liées à la problématique d'évolution en vu d'une réutilisation future dans des situations similaires. Dans le domaine d'industrie, ce thésaurus forme un historique qui est souvent stocké dans des bases de connaissances. Ce qui permet d'assurer un avantage concurrentiel et à éviter les pertes de coûts et du temps colossal pour traiter ce qui a été déjà confronté. En bref, un bon thésaurus augmente considérablement la capacité de réutilisabilité des systèmes logiciels.

3.4.1.2 Extraction de l'évolution

Partons du fait que chaque élément architectural encapsule des caractéristiques liées à sa structure, son comportement par références aux différentes fonctionnalités dont il dispose et aussi des caractéristiques décrivant son aspect évolutif face à des situations bien définies à priori. Mais souvent, nous trouvons que ces éléments évolutifs ne sont pas considérés de manière indépendantes des deux premiers cela rend la faculté de leurs réutilisation très délicate. Pour faire face à une telle situation, (Le Goaer, 2009) proposent de scinder le comportement en partie stable comprenant tous les fonctionnalités fixées et une deuxième partie comprenant le comportement évolutif comme il est présenté dans la Figure 3.5.

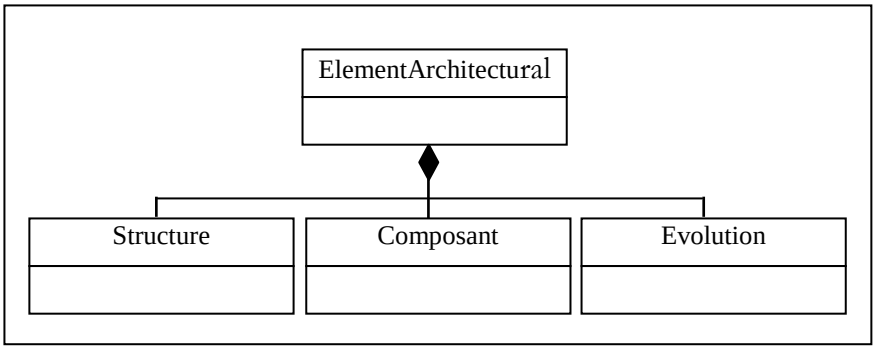

Figure 3.5- Composition d'un élément architectural.

[*Source :* (Le Goaer, et al., 2010)]

3.4.1.3 Représentation de l'évolution

Les connaissances et compétences concernant les opérations d'évolution constituent un capital essentiel pour pouvoir affronter les nouvelles exigences de l'environnement. Le style d'évolution apporte la solution adéquate pour de tel cas. La présentation et la structuration des caractéristiques associées forment une solution clé pour une bonne exploitation de ces évolutions. Pour ce faire, l'élément évolutif est présenté de façon à décrire :

- *Type du style*- chaque style lui est approprié une définition qui lui est unique,
- *Implémentation du style*- contenant l'ensemble des fonctions et comportements nécessaires à son fonctionnent,
- *Instance de style*- présente la faculté de décrire une instance de ce style pour qu'elle soit exécutable.

3.4.2 Notion de style d'évolution

Les concepts mentionnés sont spécifiques au domaine, par exemple dans le domaine de SOA, il y a des opérateurs spécifiques comme l'encapsulation d'un ancien composant en service, ou la création d'un nouveau bus par une entreprise où les différents services du composants doivent communiquer à travers ce bus. En effet, plusieurs préoccupations sont aussi domaines spécifiques et n'apparaissent qu'au cours de la planification. Les styles d'évolution sont le moyen favorable pour réaliser la spécialisation de domaine car ils encapsulent une expertise architecturale du domaine. Un style d'évolution fournit un ensemble commun de propriétés, opérateurs, contraintes et fonctions d'évaluation pour supporter la modélisation et le raisonnement sur les évolutions à entreprendre. Les domaines suivants représentent des exemples de domaines spécifiques d'évolution:

- Evolution de l'architecture client léger à des applications web,
- Evolution depuis une architecture applications web J2EE vers une architecture nuage,
- Evolution pour l'amélioration de la sécurité de l'architecture orientée service.

Nous constatons que chacun de ces problèmes d'évolution d'architecture sont récurrent et concernant un domaine spécifique. Chacun possède un état de début et des états finaux adhérents à un certain style d'architecture avec des contraintes spécifiques. Les styles d'évolution sont une réflexion drivée des styles architecturaux dans le domaine de l'ingénierie logicielle. Le style

d'architecture et la classe d'architecture partagent des éléments et des propriétés communes comme le style d'architecture orienté service. Un style architectural définit le vocabulaire des types éléments architecturaux avec un ensemble de contraintes guidant les instances et les types de composition de ces éléments dans le système. Dans le même sens, un style d'évolution définit le vocabulaire des opérateurs d'évolution avec les contraintes dirigeant leurs compositions. De manière plus précise, un style d'évolution comporte :

- Les caractéristiques du domaine auquel le style d'évolution est destiné,
- Un ensemble d'opérateurs décrivant les transformations relatives au domaine,
- Un ensemble de chemins de contraintes pour définir les règles du domaine,
- Un ensemble de fonctions relatives au domaine.

3.5. Modèles d'évolution architecturale

Nous désignons par un modèle d'évolution, la méthode ou le mécanisme adopté pour réaliser une évolution logicielle. En génie logiciel, cette évolution impacte souvent plusieurs niveaux de modélisation d'un système.

3.5.1 Logiciels de lignes de production

La ligne de produit est définie selon (Clements & Northrop, 2002) comme :

« *Un ensemble de système qui partage en commun une panoplie de fonctionnalités qui satisfaits à un besoin spécifique d'un segment particulier de marché et qui sont développés par un ensemble commun de ressources de base*».

L'intérêt de ce type de logiciel est de réduire les coûts de développement ainsi que les délais de l'opérabilité du système. Il offre en plus la possibilité d'optimiser les processus liés au domaine, ce qui augmente la qualité du logiciel en se basant essentiellement sur la réutilisation (Bosch, 2000; Pohl, Böckle, & van Der Linden, 2005). On trouve trois grandes catégories de ce type de système.

3.5.1.1 Méthodes de ligne de produits logiciels

Il est nécessaire quand nous évoquons le domaine de ligne de produits logiciels de présenter le travail de (Bosch, 2000) et (Pohl, et al., 2005) dans lequel les auteurs présentent des méthodes de conception architecturale des logiciels. Ils ont insisté sur la séparation entre le développement du logiciel et son application tout en définissant la possibilité d'inclusion de points de variabilités qui articulent les artefacts du développement. (Schmid & Verlage, 2002) proposent la construction du domaine autour de familles de systèmes ou de ligne de produits d'un point de vue économique pour pouvoir les évaluer. Plusieurs méthodes pour l'évaluation ont été présentées dans le domaine:

Dans (Van der Linden, Schmid, & Rommes, 2007), l'évaluation repose sur quatre dimensions pour permettre de cartographier le domaine de ligne de produits qui sont : activité, architecture, organisation et processus.

Plusieurs méthodes de migration ont présentées des métriques pour la migration des instance du systèmes d'information vers les logiciels de lignes de produits (Assunção & Vergilio, 2014).

(Faust & Verhoef, 2003) présentent des métriques pour la généricité et la migrantion de plusieurs instances d'un système d'information vers une ligne de produit. Une méthode d'analyse de composants PuLSETM (Bayer et al., 1999; Schmid & Verlage, 2002) est introduite pour améliorer la réutilisabilité d'un composant. L'évaluation de la faisabilité de la décision organisationnelle a été traitée par (Schroeder et al., 2015).

(Matinlassi, 2004) fournit une étude comparative sur les méthodes de ligne de produits concernant les méthodes de conception. De plus, ils explorent les méthodes d'analyse d'architecture logicielle en prenant en compte les méthodes de conception d'architecture logicielle.

3.5.1.2 Evolution de ligne de produits

Les logiciels de ligne de produits sont le plus exposés aux variations à cause de l'amélioration continue exigée par les utilisateurs, les concurrents et les consommateurs ce qui oblige leurs évolution. Celle-ci s'opère selon deux dimensions (Rabiser, Grunbacher, & Dhungana, 2007) qui peuvent évoluer indépendamment.

- Les méta-modèles peuvent être évolués suite à une évolution de contraintes de l'architecture du système logicielle (Knieke et al., 2017; Kuhrmann, Ternité, Friedrich, Rausch, & Broy, 2016).

- Les modèles de variabilité qui peuvent être changés suite à des extensions de fonctionnalités (Baresi & Quinton, 2015; Berger et al., 2014).

3.5.1.3 Processus d'ingénierie de ligne de produits

Selon la littérature, il existe deux processus pour les logiciels de ligne de produits (Pohl, et al., 2005).

- Processus d'ingénierie de domaine définissent les points communs de la variabilité des lignes de produits pour construire des artefacts réutilisables. Ce processus comporte cinq activités:

- La définition les principales caractéristiques communes et variables des produits futurs, ainsi qu'un calendrier avec les dates de sortie planifiées,
- La documentation des exigences communes pour les applications,
- La définition de l'architecture de référence raffinée de la famille de ligne de produits,
- La conception détaillée et mise en œuvre des composants,
- Les tests pour la validation et la vérification.

- Processus d'ingénierie d'application consiste à l'utilisation des caractéristiques communes de la variabilité du domaine lors du développement de l'application. Dans le même ordre d'idée, l'ingénierie d'application s'étale sur quatre activités :

- La spécification des exigences pour une application spécifique,
- La conception de l'application ce qui revient à spécifier l'architecture à adopter,
- La réalisation de l'application,

3.5.2 Réingénierie et ingénierie inverse

Reverse engineering présente une activité très importante dans l'évolution puisqu'elle permet de récupérer et d'enregistrer les caractéristiques de plus haut niveau de l'architecture par référence aux représentations abstraites de l'architecture; ce qui facilite la compréhension de l'évolution logicielle (Keller, Schauer, Robitaille, & Pagé, 1999; Kleber, Maile, & Kargl, 2018). Ces représentations sont les solutions clés pour la réutilisabilité de l'activité car elles contiennent les descriptions du comportement dynamique, de fonctionnement, de documentation et de tests du système. La technique de réingénierie est souvent utilisée pour doter le système d'une caractéristique évolutive et adaptative au système initial par recours à l'ingénierie inverse ou la restructuration d'architectures (Barnes, Garlan, & Schmerl, 2014; Mall, 2018). La première apporte une solution à l'évolution quant à la complétude des descriptions servant comme des connaissances pertinentes aux architectes sur les outils mis en place (ADL, système d'exploitation, documentation et autres) pour l'architecture du logicielle (Govin, Anquetil, Etien, Du Sorbier, & Ducasse, 2015). Les challenges liés à ces activités restent en vogue pour les travaux de nos jours pour permettre aux solutions de (Bruneliere, Burger, Cabot, & Wimmer, 2017):

- Eviter la perte d'informations due essentiellement à l'hétérogénéité des plateformes des anciens systèmes,
- Améliorer la compréhension des architectures qui représente une activité très complexe,
- Adapter les solutions à un vaste choix d'exigence.

Tandis que la structuration joue un rôle primordial dans la rénovation de l'architecture logicielle (Mirakhorli, 2015), la compréhension et analyse de l'impact de changement. C'est un processus d'amélioration continue de l'architecture permettant de préserver le fonctionnement et la cohérence des systèmes vis-à-vis des nouvelles exigences.

Notons que l'ingénierie en avant (Forward Engineering) désigne la construction du logiciel à partir du plus haut niveau de son architecture. Le cycle en fer à cheval est l'exemple le plus concret sur cette technique puisque le processus commence de la récupération du code de l'architecture dans l'objectif d'évaluer la conformité code-design, puis procède à la transformation de l'architecture et finalement entame le processus de développement à la base de la nouvelle architecture i.e. l'instanciation de l'architecture transformée. Un exemple très courant pour la programmation avancée est le refactoring qui est méthode de restructuration et d'amélioration d'architecture s'opérant à tous les niveaux d'abstraction même au niveau code (Samarthyam, Suryanarayana, & Sharma, 2016), (Zimmermann, 2015). De plus, le processus du refactoring permet d'apporter des améliorations qualitatives pour introduire plus de facilitation à la compréhension de l'architecture et de prévoir les failles susceptibles à apparaitre dans le système.

3.5.3. Modèles d'évolutions en architecture logicielle

Les architectures logicielles modélisent la structure et le comportement d'un système et le présente en adoptant une vue de plus haut niveau présentant tous les éléments architecturaux. Au même titre des logiciels, les architectures logicielles sont eux aussi assujettis à évoluer et peuvent exposer les dimensions selon lesquelles un système est appelé à évoluer (Garlan, 2000) et constituer la base de l'évolution du logiciel (Medvidovic, Rosenblum, & Taylor, 1999). Les évolutions ont été classées en évolution internes et externes selon (De Leenheer & Mens, 2008).

- L'évolution interne sous entend la modélisation des modifications des composants et des interactions au niveau de l'ontologie et cela au fur et à mesure de leur création ou destruction au cours de l'exécution. Il s'agit d'une capture dynamique du système.
- L'évolution externe permet de concevoir les modifications de la spécification des composants et des interactions nécessaires pour faire face aux nouvelles exigences des intervenants. Cela représente un phénomène d'adaptation de l'architecture logicielle à une nouvelle situation.

Il existe plusieurs approches pour décrire et faire évoluer l'architecture logicielle. (Aoyama, 2002) propose des métriques de coût de changement pour les opérations afférentes à l'évolution de l'architecture logicielle. Il opère les métriques proposées pour l'évolution continue et discontinue, qui ne sont autres que des modèles d'évolution observés à partir de l'évolution de plusieurs systèmes logiciels. Une évolution discontinue apparaît entre certaines périodes d'évolution continues et successives.

(Lung, Bot, Kalaichelvan, & Kazman, 1997) décrient une approche basée sur des scénarios qui capture et évalue les architectures logicielles en vue de leur évolution et de leur réutilisation. L'approche consiste en un cadre pour la modélisation de divers types d'informations pertinentes et en un ensemble de vues architecturales pour la réingénierie, l'analyse et la comparaison d'architectures logicielles. Ce cadre est utilisé pour modéliser plusieurs types d'informations, à savoir:

- Les informations sur les intervenants décrivent leurs objectifs, qui fournissent des limites pour l'analyse;
- Les informations sur l'architecture font référence à des principes de conception ou à des objectifs architecturaux;
- Les informations sur la qualité font référence à des attributs non fonctionnels.
- Les scénarios décrivent les cas d'utilisation du système pour capturer les fonctionnalités du système. Les scénarios qui ne sont pas directement pris en charge par le système actuel peuvent être utilisés pour détecter d'éventuelles failles ou pour évaluer la prise en charge de l'architecture par des améliorations potentielles. Les scénarios sont dérivés des objectifs des intervenants, des objectifs architecturaux et des attributs ou objectifs de qualité système souhaités.

L'architecture logicielle d'un système évolutif doit permettre des modifications logicielles et d'évoluer de manière contrôlée sans compromettre l'intégrité du système et les invariants (Bennett & Rajlich, 2000). Cependant, l'évolution de l'architecture logicielle implique souvent l'intégration de préoccupations transversales. Par conséquent, l'intégrité architecturale est un aspect à prendre en compte. Sinon, ces problèmes transversaux pourraient, s'ils ne sont pas traités avec précaution, introduire des incohérences et conduire à une dégradation de l'évolutivité à long terme. Pour résoudre ce problème d'incohérence, (Barais, Cariou, Duchien, Pessemier, & Seinturier, 2004) décrivent un cadre nommé TranSAT[1]. Le cadre utilise l'aspect architectural pour décrire les nouvelles préoccupations et leur intégration dans l'architecture existante. Le cadre permet à l'architecte logiciel de concevoir l'architecture logicielle par étapes en termes d'aspects au stade de la conception.

[1] Tranform Software Architecture Tehnologies.

Selon (Jansen & Bosch, 2004, 2005), une décision de conception architecturale est un concept clé dans l'évolution de l'architecture logicielle. Capturer ces décisions est donc essentiel pour traiter les connaissances architecturales afin de remédier au problème d'évaporation. Autrement, les connaissances conceptuelles ayant conduits à l'architecture seront perdues. En outre, des modifications apportées à l'architecture logicielle pourraient entraîner une violation des décisions de conception antérieures, entraînant une érosion accrue de la conception (Van Gurp & Bosch, 2002).

3.5.4 Modèle d'évolution en orienté service

L'avantage majeur des logiciels basés composants (CBSE) est qu'ils permettent la composition et l'assemblage des composants pour une éventuelle réutilisation et ou construction d'un système logiciel. Cette technique permet donc d'atténuer la complexité de développement par l'augmentation de la réutilisation des composants déjà existants et de réduire les délais de la mise en œuvre compte tenu de l'évolution en temps réels de ces applications. Cependant, la croissance du nombre d'application évoluant sur le Net et le besoin accru des systèmes logiciels d'opérer dans des environnements ouverts et en temps réels sur les réseaux locaux et/ou d'internet ont fait émerger les approches orientées services (SOSE). Ces approches, où le service est l'élément fondamental pour le développement, ont vu le jour à partir des Framework CBSE et outils et référentiels orientés objets. De plus, des avantages liés aux CBSE, SOSE permettent de gérer l'interopérabilité qui offre la construction des applications multiplateformes. La combinaison entre les deux approches est devenue un atout des applications pour favoriser à la fois la réutilisation des composants et l'interopérabilité de ses composants logiciels. A ce jour, plusieurs recherches se sont investis dans ce contexte dans l'objectif est d'augmenter la qualité des solutions logicielles.

(Brown, Johnston, & Kelly, 2002) présentent un modèle pour la bonne compréhension des services des architectures orientées services pour prendre en charge les solutions logicielles pour les entreprises. Précisément, ils exploitent la relation service et composant et décrivent le pourquoi et en quoi les composants logiciels constituent une solution pour le développement des architectures orientées services. Cette relation est basée surtout sur l'exploration des interfaces entre les deux concepts compte tenu des contraintes divergentes qui leurs sont associées. Dans de tel cas, l'UML est considéré comme un outil efficace pour la description logique et la mise en œuvre de telles interfaces.

(Jiang & Willey, 2005), les auteurs suggèrent une architecture à plusieurs niveaux dans le but de fournir plusieurs alternatives d'évolution faciles et flexibles à être intégrées dans les larges systèmes et/ou les systèmes distribués tout en maintenant le degré de performance du système.

(Wang & Fung, 2004) la proposition stipule l'organisation de l'entreprise sous forme de systèmes à base de composants puis d'identifier les valeurs ajoutées en les formalisant en tant que services flexibles et extensibles.

(Cervantes & Hall, 2004) introduisent les concepts orientés services dans les modèles de composants afin de prendre en charge la liaison tardive et la disponibilité dynamique des composants dans les modèles à composants.

(O'Brien, Merson, & Bass, 2007) Les auteurs se sont focalisés sur l'étude de l'impact qu'apporte l'architecture orientée services sur un ensemble d'attributs de qualité tout en

identifiant les problèmes et les compromis qui leur sont liés. Les attributs de qualité sont l'interopérabilité, la performance, la sécurité, la fiabilité, la disponibilité, la modifiable, la testabilité, la convivialité et l'évolutivité.

3.6. Référentiels existants d'évolution architecturale

La sélection des outils appropriés aux architectures logicielles, selon les spécificités de l'environnement, a besoin de plus de précision et d'informations utiles dans les différents aspects. Cette préoccupation a entrainé forcement un besoin intense à la création de référentiels permettant de guider les traitements opérés. Dans ce qui suit, nous allons analyser les outils les plus caractérisés dans le domaine de l'architecture logicielle aussi bien en industrie qu'en organisations gouvernementales.

3.6.1 Référentiel de Zachman

Le cadre conceptuel qu'offre Zachman décrit plusieurs sujets et modèles selon plusieurs vues pour permettre un développement et/ou une documentation plus appropriés. L'objectif est de mettre à la disposition des intervenants un fondement de base pour supporter l'organisation, l'accès, l'intégration, le développement et la modification d'un ensemble de représentations architecturales des différents systèmes d'informations. Il est donc bien adapté en industrie. Zachman fournit 36 catégories organisées sur une matrice de six lignes présentant les perspectives globales de l'architecture et de six colonnes exhibant des différents artefacts de l'architecture. Le référentiel avance six niveaux d'abstraction (les lignes) définis en termes de rôles dans l'entreprise ce qui met à la disposition de chaque acteur des outils bien appropriés exprimant sa propre vue.

- Vues contextuelles: représente la portée du système qui est établie par un planificateur,
- Vues conceptuelles: Le modèle représente une vue commerciale dont le rôle est assigné aux propriétaires du système. Cette perspective est orientée activité commerciale, on trouve par exemple les modèles : sémantiques, logistiques, business plan …etc.
- Vues logiques: les vues pour caractériser les modèles des systèmes d'informations dont le rôle revient au concepteur. Cette vue est analogue au model de cas d'utilisation d'UML. Parmi les modèles logiques on trouve : Model logique de données, modèle du processus métier, …etc.
- Vues physiques: se sont les modèles représentant le système du point de vue technologique. Donc, le constructeur a pour rôle de confectionner plusieurs modèles comme les modèles: physique des données, présentation de l'architecture, control de structures, …etc.. En UML cette vue est représenté par le diagramme de déploiement.
- Vues hors-contextuelle: les systèmes où les spécifications de l'architecture sont bien détaillées par les sous-traitants. Elles engendrent la réalisation des définitions de données, les programmes, l'architecture réseaux, …etc.
- Vue de fonctionnement d'entreprise. Cette vue mappe les systèmes fonctionnels dans l'entreprise représentant les produits effectifs de l'activité du génie logiciel tel que : les données, les fonctions, les stratégies, …etc

Le modèle de Zachman est indépendant de toutes spécifications de méthodologies (Frankel et al., 2003). En outre, aucune technique spécifique n'est décrite pour créer les documents de

spécification suggérés. Nous précisons que les techniques de modélisation employées sont principalement les modèles orientés architectures de l'OMG. Le BPML est le langage utilisé par le Framework pour la modélisation métiers. La Table 3.6 décrit les avantages et les inconvénients que présente le modèle.

Table 3.6- Caractéristiques du Framework Zachman.

Avantages	Inconvénients
Fournit et améliore la communication professionnelle entre les membres de la communauté.	Peut conduire à la rédaction de plusieurs documentations selon les cas.
Aide à comprendre les raisons et les risques dépendant ou non du développement de la représentation de l'architecture.	Peut guider à un processus lourd dans le développement.
Fournit une variété d'outils et une méthodologie favorable pour les concepteurs.	N'est pas bien adopté par la communauté des développeurs.
Permet l'amélioration des approches existantes.	

3.6.2 NATO Architecture Framework: DoDAF

Le département des outils architecturaux de la défense (Department of defence Architecture Framework) a proposé l'organisation d'architecture au sein des entreprises donnant naissance à ce que nous appelons aujourd'hui l'architecture d'entreprise. Cet outil a montré une bonne adaptation avec les grands systèmes et qui sont même compliqués. Il fournit un outil pour l'organisation de l'architecture de l'entreprise par la considération de plusieurs vues (d'ensemble, opérationnelle, systémique et technique standardisée). La Table 3.7 résume les avantages et inconvénients de ce Framework utilisé essentiellement pour les systèmes à grandes échelles.

Table 3.7- Caractéristiques du DoDAF Framework.

Avantages	Inconvénients
Présente une approche commune de description et de comparaison des architectures entreprise (EA) du département de la défense.	Pas d'ontologie commune pour les éléments architecturaux.
Repose sur des principes communs par l'utilisation de la notion d'assomption et des outils UML pour la modèlisation.	Le cas de spécification des architectures cibles n'est pas traité pas ce Framework.
Coût est bien réduit pour le développement des systèmes similaires.	Les plans d'activités financières ne sont pas du tout abordés.

3.6.3 TOGAF (The Open Group Architecture Framework)

Ce référentiel représente une méthodologie d'architecture technique. Il a été développé par The Open Group dans un cadre global d'architecture d'entreprise, connu sous le nom de référentiel d'architecture technique pour la gestion des informations (TAFIM[1]). TOGAF comprend deux parties principales (Hornford et al., 2011): une méthode de développement d'architecture et un référentiel de contenu d'architecture.

[1] TAFIM : Tachnical Architecture Framework Information Management.

Table 3.8- Vues architecturales TOGAF.

Vues	Définition
Architecture commerciale	Traitement des préoccupations des utilisateurs dans le souci de répondre à un besoin bien exprimé.
Sécurité d'entreprise	Traite l'aspect sécuritaire du système.
Génie logiciel	Pour le développement des nouveaux systèmes.
Ingénierie système	Pour monter et construire des systèmes logiciels par intégration des composants (logiciels ou matériels) dans le système en cours.
Communications	Structuration de la communication entre composants et entre composant du réseau pour simplifier la conception et la planification.
Flot de données	Concerne le stockage, restauration, sécurité et archivage des données.
Gestion de l'entreprise	Traite les opérations administratives et la gestion du système.
Acquisition	Concerne l'acquisition des composants logiciels pris sur étagère (Commercial Off-the-Shelf) ainsi que matériels.

Ces parties permettent de fournir les procédures, les lignes directrices et la classification pour le développement et l'utilisation des points de vues et vues architecturales. Il respecte entre autre la norme ISO/IEC 42010:2007 relative à la création de points de vues et des vues architecturales comme le montre la Table 3.8).

Table 3.9- Caractéristiques TOGAF.

Avantages	Inconvénients[1,2]
Transparence renforcée de la responsabilité.	Détails très conséquents.
Les risques sont mieux contrôlés avec la possibilité d'un contrôle proactif.	Les méthodes sont planifiées et guidées ce qui favorise la dépendance.
Protection des actifs des entreprises.	Disposition de faibles informations sur les architectures.
Création de valeur	

En résumé, TOGAF est un outil qui fournit une compréhension sur la modélisation, la planification, l'implémentation et la gouvernance de l'architecture d'information de l'entreprise dont les avantages et les désavantages sont ventilés dans la Table 3.9.

Table 3.10- Caractéristiques TEAF.

Avantages	Inconvénients
Fournit un guide pour les utilisateurs de la trésorerie.	Ne contient pas assez de détails sur la façon avec laquelle sont générés les documents de spécifications qui ont été indiqués par TEAF.
Permet d'adoptés des plans stratégiques et les propriétés individuelles.	
Entraine une large réutilisabilité et interopérabilité.	

[1] https://publications.opengroup.org/n180 Consulté le 12/05/2018)
[2] https://bizzdesign.com/products/enterprise-studio/enterprise-architecture/ COnsulté le (12/05/2018)

3.6.4 TEAF (Treasury Enterprise Architecture Framework)

Basé sur Zackman Framework, Il est développé par la trésorerie américaine. Il supporte la ré-modélisation et permet de guider le développement des activités relatives aux différents bureaux en termes de produit. De la même façon, la Table 3.10 esquisse les limites et avantages que porte ce référentiel essentiellement orienté activités d'entreprise. La modélisation s'appuie principalement sur les diagrammes de flux et diagrammes UML.

3.6.5 FEAF (Federal Enterprise Architecture Framework)

FEAF a été développé pour le gouvernement fédéral afin de fournir une méthodologie pour l'utilisation dans le domaine de la prestation de service. Le référentiel est axé sur les activités d'entreprises, données et applications et aussi utilisé dans les phases de communication et de planification. Le référentiel adopte la méthode UML comme la principale technique de modélisation et le langage BPML pour la modélisation métiers.

Table 3.11- Caractéristiques FEAF.

Avantages	Inconvénients
Promotion de l'interopérabilité fédérale.	Nécessite une expertise d'acquisition de techniques pour l'architecture d'entreprise.
Promotion du partage des ressources des agences délocalisées.	Le fédéral doit payer le droit d'exercer.
Réduction des coûts que ce soit pour les agences ou le fédéral.	Les préoccupations hors territoire ou la perte d'autonomie entravent les efforts du fédéral à long terme.
Amélioration du partage des informations.	Il est vraiment difficile d'avoir des modèles et des normes communs à toutes les agences pour assurer l'interopérabilité.
Planification rationnelle du capital IT.	

Pour se positionner en termes d'avantages et d'inconvénients la Table 3.11 est réalisée à cet intérêt. La méthodologie des systèmes de développement: utilise aussi la méthodologie RUP[1] comme dans TEAF.

Table 3.12- Caractéristiques ISO RM-ODP.

Avantages	Inconvénients
Plusieurs détails pour les phases d'analyse et de développements des applications.	RM-ODP présente un problème de consistance entre les différents points de vues même des opérations de contrôles ne peuvent pas garantir la cohérence de ces vues.
Une plateforme pour intégrer des besoins consistants pour des différents langages.	
Un ensemble de patron de raisonnement pour pouvoir identifier les entités fondamentales du système.	

[1] Rational Unified Process est utilisé comme une méthodologie de développement des systèmes.

3.6.6 ISO RM-ODP (ISO- Reference Model for Open Distributed Processing)

Ce modèle fournit un outil standard pour supporter l'activité distribuée sur les plateformes hétérogènes. Les approches de modélisation objets sont utilisées pour décrire les systèmes dans l'environnement distribué. Selon (Vallecillo, 2001), le référentiel propose plusieurs points de vues : *entreprise*, *information*, *computationnel*, *ingénierie* et *technologique*. Le référentiel favorise les outils UML et OMG (MDA) comme techniques de modélisations des systèmes. Les avantages et limites sont tabulés dans la Table 3.12.

La chronologie de développement des différents Framework (Figure 3.6) montre que le Framework de Zachman était le tout premier et constitue donc la fondation de tous les autres référentiels. Les comparaisons effectuées dans cette section peuvent servir d'un outil ou d'un référentiel assistant le choix d'un outil parmi d'autres. Cependant, il permet à l'utilisateur de choisir à la base de ses besoins l'outil le plus approprié. Il permet aussi d'identifier les outils possibles pour son choix.

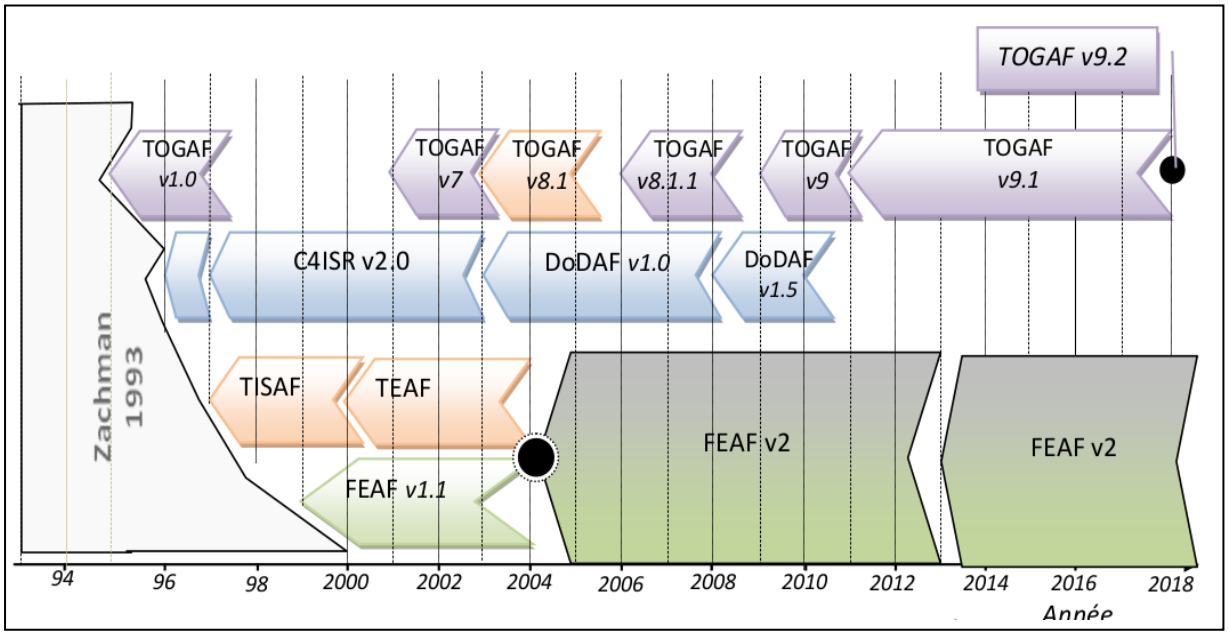

Figure 3.6- Historiques de développement des Framework pour l'évolution.

Table 3.13- Synthèse des référentiels existants.

Framework →	Zachman	DoDAF	TOGAF	TEAF	FEAF	ISO RM
Spécification de documents	●	●	○	●	●	●
Méta-Modèle	⊙	⊙	○	⊙	○	●
Rôle	○	⊙	○	●	●	○
Technique	○	●	⊙	○	○	●
Modèle de la procédure	⊙	●	●	●	●	⊙

Légende- ● *Accomplit* ⊙ *Partiellement accompli* ○ *non accompli*

[Source: (Leist & Zellner, 2006)]

La Table 3.13 présente une synthèse sur les référentiels par la considération d'un ensemble de spécifications tels que : la documentation, l'utilisation de méta modèles, le rôle, l'existence de techniques et l'existence d'un modèle de la procédure. Nous constatons que le modèle de Zachman est plus consacré pour l'analyse et la compréhension des architectures d'un système. Le modèle ISO-RM est considéré le modèle le plus technique pour les architectures logicielles.

3.7 Taxonomies d'évolution

Le processus de l'évolution logicielle est complexe car toutes les opérations sur les artefacts s'opèrent durant la phase de développement du cycle de vie depuis l'expression de besoins en passant par l'analyse, la conception et la documentation jusqu'à la validation du code exécutable. Comme conséquence, plusieurs études ont eu le tact de chercher et d'analyser les changements et les impacts de ces changements sur le produit logiciel. Certains proposent un vocabulaire commun en se basant sur un ensemble de dimensions pour pouvoir classifier ce qui et fait dans le domaine. La préoccupation principale de toutes ces études est de comprendre la cartographie du domaine en matière d'évolution.

3.7.1 Taxonomie en évolution logicielle

Nous allons dans ce qui suit présenter les taxonomies les plus intéressantes et qui sont les plus référencées dans le domaine de l'ingénierie logicielle.

3.7.1.1 Taxonomie d'évolution basée maintenance

(Lientz & Swanson, 1980) dans cette étude les auteurs fondent leur analyse sur la maintenance logicielle. Ils proposent une typologie exhaustive pour la maintenance logicielle qui est bifurquée en trois catégories fondamentales :

- **Maintenance perfective:** Qualifie la maintenance effectuée pour éliminer les défauts du traitement, améliorer les performances ou améliorer la maintenabilité,

- **Maintenance adaptative:** Désigne la maintenance effectuée en réponse aux modifications des environnements de données et de traitements. Cette maintenance est une réflexion issue de la portabilité des logicielles sur des environnements différents.

- **Maintenance corrective:** C'est la maintenance effectuée en réponse aux défaillances survenant au cours ou après l'application des autres maintenances cités ci-dessus.

Par cette étude, les auteurs ont illustré le volet de la motivation derrière une maintenance, ce qui revient à s'intéresser exclusivement sur la réponse à la question Pourquoi? Cependant, il est important de souligner que cette définition a été initiatrice pour la normalisation ISO par des éventuelles extensions donnant éclot à la norme (ISO/IEC Standard for Software Maintenance (ISO 1999) qui a introduit une nouvelle catégorie de maintenance relevant des activités de maintenances préventives.

Plus de précision ont été apportées par Chapin et al. (Chapin, Hale, Khan, Ramil, & Tan, 2001) qui apportés plus de clarifications et une redéfinition de l'évolution et la maintenance logicielle. La classification proposée ne repose pas sur l'intention les utilisateurs mais sur l'objectivité de l'activité de la maintenance à partir de l'observation des artefacts avant et après

comparaison des documentions logicielles. La classification proposée adopte une évidence hiérarchiques prenant en considération: i) le logiciel ii) documentation iii) les propriétés du logiciel iv) expérience de l'utilisateur. Comme résultat, ils ont présenté douze différents types d'évolution et de maintenances logicielles se basant sur trois critères d'évolution ou de maintenance du système à savoir le logiciel, le code et l'expérience de l'utilisateur. La Figure 3.7 présente les différents types identifiés par Chapin et al. Selon un degré d'impact échelonné en dix graduations (de 0 à 9) dans le processus commercial.

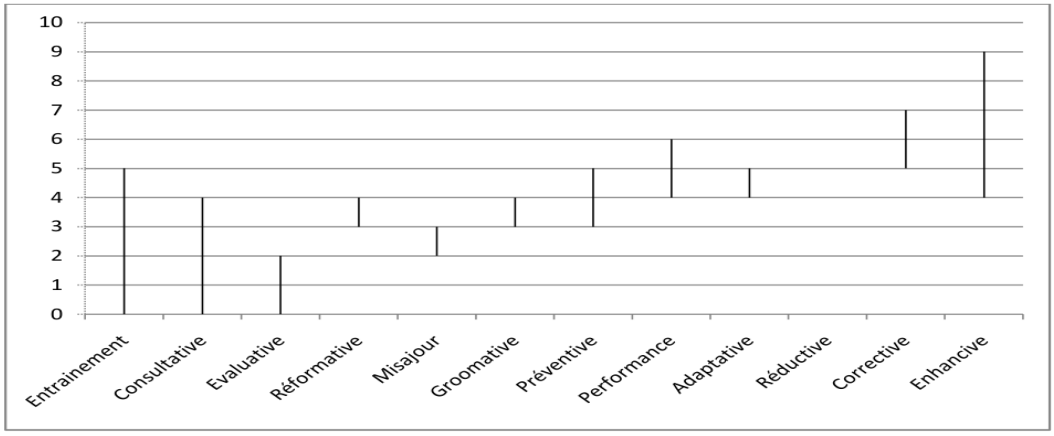

Figure 3.7- Représentation du degré d'impact du processus commercial.

La typologie introduite distingue entre les types de maintenance et les types destinés pour l'évolution. Dans la Table 3.14, nous présentons que les facteurs liés aux règles commerciales et les propriétés logicielles puisqu'elles traitent le l'évolution.

Table 3.14- Relations entre les différents types concernés par l'évolution.

N°	Types spécifiques	Evolution	Maintenance
A-Règles commerciales			
1	Enrichissement (Enhancive)	●	●
2	Correction	●	●
3	Réduction	●	
B- Propriétés logicielles			
1	Perfective	●	●
2	Adaptative	●	●
3	Préventive		●
4	Groomative		●

3.7.1.2 Taxonomie d'évolution basée changement

Dans (Buckley, 2005), les auteurs ont mis en place une taxonomie d'évolution plus complexe se focalisant sur des thèmes de changement en évolution logicielle et les mécanismes et facteurs influençant le processus d'évolution. Dans cette étude, les auteurs ont essayé d'analyser le changement en apportant des réponses au questionnement Quoi ? Comment ? Pourquoi ? Quand? Et Où? Cependant, les dimensions de l'évolution sont classées dans des thèmes caractérisant le mécanisme ou comme facteurs d'influence comme le clarifie la Figure 3.8. Un facteur déterminant, par exemple, la «disponibilité» d'un système peut être affecté par le

mécanisme de changement appliqué à ce système. Si un système doit être hautement disponible, un mécanisme de modification à l'exécution doit être appliqué. Les caractéristiques du changement de logiciel sont les facteurs de changement de logiciel qui peuvent être utilisés pour décrire la nature des changements (par exemple, «moment du changement»).

Afin de faciliter une meilleure compréhension du travail de cette thèse, le thème «objet de changement» est présenté plus en détail ici. Ce thème décrit le ou les emplacements dans le système où des modifications sont apportées, divisés en quatre facteurs:

- Les propriétés temporelles : ces propriétés viennent comme une réponse au questionnement Quand ? Ils ont identifié des caractéristiques du :

- *Le temps du changement :* le temps d'effectuer le changement peut être : *statique*, *dynamique*, *chargement*,
- *Historique du changement :* pour permettre le suivi des changements qui peuvent être de nature : *séquentiel* ou *parallèle*,
- *Fréquence de changement :* ayant pour valeur continue, périodique et aléatoire,

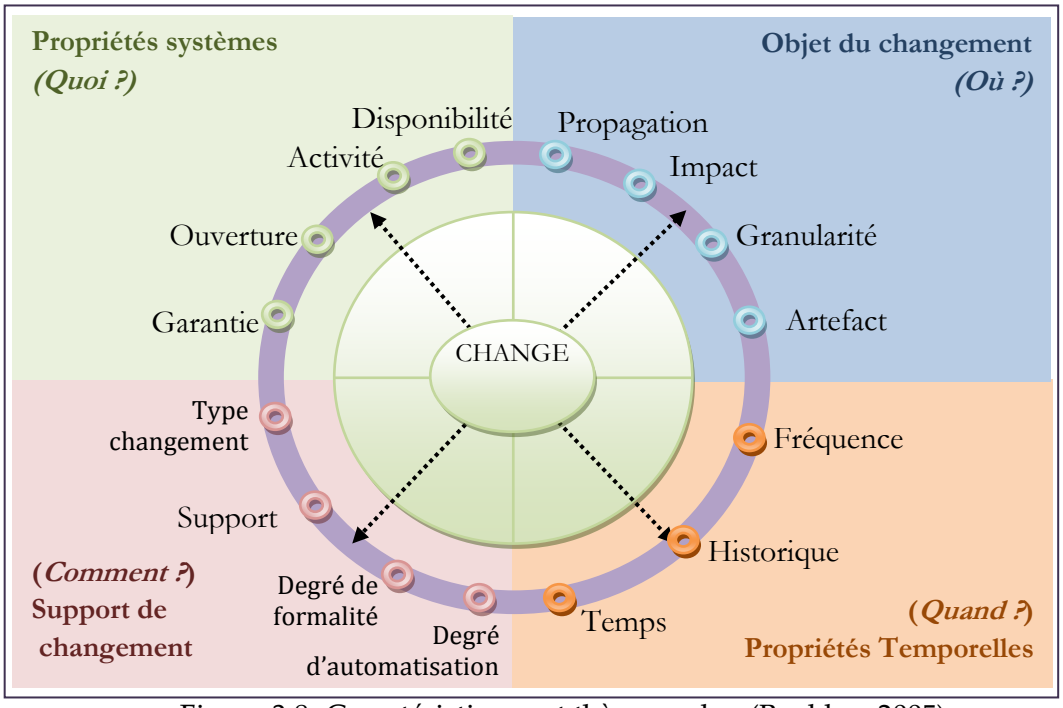

Figure 3.8- Caractéristiques et thèmes selon (Buckley, 2005)

- *Anticipation :* Cette dimension décrit le moment où sont considérés et prévus les changements émanant d'un besoin de changement. Le cycle de vie du logiciel représente la plateforme des valeurs d'anticipation c.-à-d. que l'anticipation peut être prévue à la phase de développement comme au niveau de la phase de conception et de décisions. L'anticipation est présentée selon deux métriques *anticipé* et *non anticipé*.

- L'objet du changement : cet axe vient en réponse du questionnement Où ?

- *Artefact:* fait référence à tout artefact produit et/ou utilisé tout au long du cycle de développement du logiciel et susceptible d'être modifié, allant des exigences à l'architecture et à la conception, en passant par le code source, la documentation, les configurations et les tests.

- *Granularité:* fait référence à l'échelle des artefacts à modifier et peut aller du très grosse (architecture système) au support (modules/composants) jusqu'à un degré de granularité très fin (classes, méthodes).
- *L'impact:* L'impact d'un changement, qu'il s'agisse d'un changement local ou global, peut englober différents artefacts avec de différents niveaux d'abstraction (du code source à l'architecture).
- *Propagation du changement:* fait référence au processus permettant de s'assurer qu'un changement ayant un impact global est propagé à toutes les entités associées du système logiciel. La propagation peut être effectuée dans deux directions; horizontale ou verticale (de haut en bas ou de bas en haut).

- Les propriétés du système: cette dimension apporte la réponse à la question Quoi ? Les propriétés réservées pour répondre à ces questions sont :

- *Disponibilité :* est préconisée pour déterminer si un système est disponible en permanence ou non. Souvent, cette dimension est réservée pour représenter une propriété de qualité du système (Offutt, 2002). En pratique, la plupart des systèmes tolèrent des arrêts occasionnels dans le but d'effectuer des modifications généralement pour des besoins d'amélioration ou d'extension du code source. Les valeurs admissent sont *toujours disponible* ou *non*.
- *Activité :* Cette dimension vise à mesurer la manière avec laquelle les modifications sont conduites. Le système est dit réactif si les changements sont établis de manière externe par exemple conduites par un utilisateur externe. Contrairement, le système est proactif s'il produit lui-même les modifications désirées. Il est opportun dans ce cas que le système garde trace sur tous ses états pour pouvoir autoriser le déclanchement des modifications. Les métriques de la propriété activité sont *Réactive* et *Proactive.*
- *Ouverture :* Cette propriété peut être *ouvert* ou *fermée.* L'ouverture des logiciels n'implique pas un accès au code source, mais une approche méthodologique permettant de définir des méthodes pour une entrée ouverte, un processus ouvert et une sortie ouverte vers des outils de modélisation. L'ouverture peut être fermée lorsque les résultats sont fixés au préalable et on veut seulement ajuster une solution répondant à une certaine perception de solution. L'ouverture est la capacité logicielle de:
  - Echanger des informations avec d'autres logiciels (interopérabilité ouverte) et avec l'environnement physique (intégration d'informations virtuelles);
  - Modifier les informations échangées et le processus de traitement des informations,
  - Utiliser, comparer et consolider les variations de conception générées pour l'analyse, la simulation et la fabrication (Salim & Burry, 2010).
- *Garantie* : Cette dimension encapsule plusieurs autres caractéristiques comme :
  - *Sécurité*, présente une assurance pour le bon fonctionnement du système dans les environnements où il évolue,
  - Sécurité comportementale pour s'assurer le système ne crash pas au cours de l'exécution et/ou qu'aucun comportement imprévisible se produira,
  - Compatibilité ascendante, qui garantit que les versions antérieures d'un composant logiciel peuvent être remplacées en toute sécurité par des versions plus récentes.

4-Supports de changement : dans cette catégorie nous trouvons les propriétés liées aux :

- *Degré d'automatisation :* L'automation implique des machines, des outils, des dispositifs, des installations et des systèmes qui sont tous des plates-formes développées par des humains pour effectuer un ensemble d'activités donné sans implication humaine au cours de ces activités,
- *Degré de formulation :* Désigne le mécanisme mis en œuvre pour soutenir le changement. Les auteurs précisent que ces mécanismes peuvent être de manière ad-hoc ou avoir des bases mathématiques comme les graphes orientés ou les fonctions mathématiques d'optimisation.
- *Type de changement :* Cette dimension est introduite comme une spécification aux la typologie initiée par (Chapin, et al., 2001). Le type ici, distingue entre les changements structurels et les changements sémantiques. Le premier type de changement concerne la modification de structure du logiciel (comme l'adition, la suppression, le remplacement, …etc.). Tandis que le deuxième type concerne le volet sémantique du changement. Les auteurs distinguent entre modifications sémantiques (modification de visibilité et accessibilité du code par exemple) et conservation-sémantique (activité de refactoring, activité de restructuration). Les changements structurels sont des changements qui modifient la structure du logiciel. Dans de nombreux cas, ces modifications modifieront également le comportement du logiciel. Les dimensions peuvent être impactées entre elles dans le même thème ou celles d'autres thèmes.

Par exemple, dans une architecture logicielle l'évolution, un artefact peut affecter sa granularité qui est un facteur du même thème comme elle peut aussi avoir un impact directe sur son degré d'automatisation et son degré d'ouverture qui sont des facteurs d'autres thèmes. De plus, l'impact du changement peut rester dans le même niveau d'abstraction de l'architecture (impact vertical) ou s'étendre sur différents niveaux (impact horizontal). La propagation peut être effectuée dans le sens vertical aussi bien que horizontal i.e. de l'architecture vers les instances ou l'inverse des instances à l'architecture. La dimension du temps de changement lors de l'évolution de l'architecture peut influer largement le mécanisme de changements, en les organisant en temps statiques, passant par le temps de chargement et finalement en temps d'exécution. L'évolution dynamique se réfère à la modification de l'architecture et à la mise en œuvre de ces modifications dans le système sans arrêter le système. Il s'agit d'une exigence essentielle pour de nombreux logiciels, tels que les systèmes de contrôle du trafic aérien, de télécommunications et financiers (Hassan, Queudet, & Oussalah, 2016).

3.7.2 Taxonomie d'évolution architecturale

Peu de travaux de taxonomies, à notre connaissance, ont été formulés dans le domaine de l'évolution d'architectures logicielles et, s'ils existent, ils reposent souvent sur les techniques utilisées comme: le code source, les graphes, les modèles, les styles ...etc. Ces taxonomies sont donc plus spécialisées et se limitent à un sous ensemble de travaux dans le domaine.

3.7.2.1 Selon Garlan et al. (Garlan & Schmerl, 2009)

La classification découle du fait que les approches existantes manquent de stratégies de raisonnement pour étendre l'évolution sur plusieurs niveaux de modélisation. Ils préconisent quatre grandes catégories approches:

- Catégorie des travaux supportant l'évolution logicielle : qui touche les approches orientées code pour la maintenabilité, le ré-factoring et le couplage.

- Catégorie des travaux supportant des outils pour le visionnement et projet de planification : Dans lesquels les artefacts sont consultés et comparés pour décider sur l'évolution. Les outils déployés s'intéressent plus à la composition plutôt qu'au raisonnement sur les architectures logicielles.

- Catégorie des approches formelles pour la transformation d'architectures : qui permettent de décrire une séquence de transformation à appliquer sur les modèles qui capturent les structures et les comportements d'une architecture ce qui permet de dériver une autre appartenant à l'espace de toutes les combinaisons de transformations. Ces approches ne préconisent aucun raisonnement sur les modifications apportées sur une architecture.

- Catégorie des approches d'arbitrage et d'analyse pour l'évolution architecturale : pour désigner les approches introduisant des techniques d'analyse d'arbitrage pour faire améliorer une architecture et de planifier cette évolution.

3.7.2.2 Selon Chaki et al. (Chaki, Diaz-Pace, Garlan, Gurfinkel, & Ozkaya, 2009)

Cette classification des types communs pour les travaux de l'évolution d'architectures et préconise trois classes d'évolution d'architecture:

- Classe maintenance basée évolution: sont les travaux qui s'intéressent aux décisions de correction et de modifiabilité d'architecture pour répondre aux préoccupations des architectes. Ces travaux nécessitent souvent une intervention au niveau de modélisation le plus bas (code) et englobent les travaux d'évolution aux niveaux code tel que le re-factoring.

- Classe évolution ouverte: désigne l'ensemble de travaux dont l'architecture d'arrivée n'est pas connue à priori. L'approche infère, à partir d'une architecture de départ, une ou plusieurs solutions respectant un contexte connu d'avance.

- Classe évolution fermée: catégorise les travaux qui se fixent une architecture cible dès le départ et modélise la façon de l'atteindre à partir d'une architecture initiale.

3.7.2.3 Selon Rose et al. (Rose, Paige, Kolovos, & Polack, 2009)

Cette classification présente une catégorisation des approches existantes pour modéliser la migration des modelés. Mens et al. (M. H. a. G. Wachsmuth, 2013) ont repris cette classification à travers un ensemble explicite de caractéristiques pour motiver cette classification. La classification de Rose traite les approches de co-évolutions entre méta-modèles et modèles pour résoudre le problème de consistance entre les deux. Elle est répartie en trois catégories:

- Spécification manuelle: concerne les travaux dont la stratégie de migration est codée manuellement par le développeur du méta-modèle par des langages habituels comme Java ou par des langages de transformation de modèle à modèle comme Query/Views/Transformations (QVT) (Gardner, Griffin, Koehler, & Hauser, 2003) et ATL2 (Jouault, 2006). Cette stratégie manipule les instances du méta-modèle à travers des manipulations autorisées par le Framework de modélisation.

- Opérateur basé coévolution: Ce sont les travaux qui utilisent une bibliothèque d'opérateurs de coévolution à travers lesquels les modèles sont manipulés aux niveaux méta-modèles. La migration des modèles au niveau méta est générée automatiquement. Par exemple, COPE est une

approche basée opérateur de coévolution pour EMF (Bürger et al., 2018; Herrmannsdoerfer, Benz, & Juergens, 2009; Kessentini, Sahraoui, & Wimmer, 2019; G. Wachsmuth, 2007).

- Méta-modèle Matching: Ces approches infèrent la stratégie de migration des modèles à travers l'analyse du méta-modèle évolué et avec son historique. Ces approches sont réparties en approches de différenciation si les modifications ne s'opèrent pas sur le méta-modèle d'origine sinon elles sont dites approches d'enregistrement des modifications (Cicchetti, Di Ruscio, Eramo, & Pierantonio, 2008; Garcés, Jouault, Cointe, & Bézivin, 2009; Herrmannsdoerfer, et al., 2009).

3.7.2.4 Selon Breivold et al. (Breivold, Crnkovic, & Larsson, 2012)

A travers une étude systématique sur 82 travaux existants dans le domaine de l'évolution d'architecture, cinq thématiques principales de catégories ont été présentées:

- Considération de la qualité au cours de la conception de l'architecture logicielle: Concerne les travaux qui se focalisent sur la façon avec la quelle la qualité a été introduite explicitement durant la phase de conception d'architecture. Elle est subdivisée en trois sous-classes:

- *Attributs de qualité axé besoins*, où chaque décision conceptuelle se base sur une déduction à partir de ces attributs (qui peuvent être des attributs orientés conception, prototype d'architecture, …),
- *Attributs de qualité axé scénario* qui considèrent les attributs de qualité à travers un scénario réel,
- *Attributs de qualité axé facteur d'influence* à travers une gestion des attributs significatifs et des contraintes liées à la conception.

- Évaluation de la qualité au niveau d'architecture logicielle: Ce sont les approches d'évaluation conçues pour évaluer et améliorer l'architecture initialement construite après la phase de conception. Elle englobe trois sous classes:

- *Basée expérience* pour les travaux qui exploitent les expériences techniques des intervenants,
- *Basée scénario où les attributs de la qualité* sont évalués à travers un scénario pour une description concrète ce qui à pour objectif d'éviter les ambiguïtés et les interprétations conflictuelles des attributs,
- *Basée métrique* qui s'appuient sur des métriques pour évaluer les attributs de qualités comme la taille du logiciel, l'évolution du ratio (nombre d'évolution par apport à la taille du software), la vitesse d'évolution ...etc.

- Évaluation économique dans la détermination du niveau d'incertitude: Elle regroupe les travaux d'évolution de rentabilité c.-à-d. en terme de coûts et d'avantages. Ses attributs peuvent être le coût de la maintenabilité, l'effort de prédiction de maintenabilité durant la conception architecturale, la quantification des décisions conceptuelles...etc.

- Gestion de la connaissance architecturale: Ce sont les approches se basant sur des sources d'informations pour capturer la connaissance architecturale.

- Techniques de modélisation: Concernent les travaux qui se focalisent sur la modélisation d'artefacts pour supporter l'évolution d'architecture. Par exemple, la modélisation de la traçabilité entre les besoins, évaluation de l'architecture logicielle et réutilisation, gestion des propriétés de

qualité durant tout le cycle de vie, l'utilisation des tactiques architecturales pour incarner les besoins non fonctionnels dans l'architecture logicielle ...etc.)

3.7.3 Problèmes liés aux classifications existantes

La complexité, les différents changements et la visibilité de représentation des structures du système logiciel ont montré la nécessité accrue à considérer le phénomène d'évolution logicielle comme une problématique fondamentale pour le développement des logiciels (Brooks, 1989). Cependant, l'activité évolutive ne doit pas ignorer l'aspect qualitative du système puisque une nouvelle opération peut dégrader ou voire même néfaste pour le système. L'évolution n'est pas une opération disjointe de la qualité mais plutôt deux opérations complémentaires. L'étude de (Parnas, 1994) a démontré que le vieillissement des logiciels les rendent plus fragiles et obsolètes. Il présente deux raisons majeures qui accélèrent le déclin du système jusqu'à sa mort totale. La première est due principalement aux responsables du système qui ne sont pas investis ou ne se sont pas inquiétés de le modifier pour qu'il répond aux nouveaux besoins de l'organisation. La deuxième est la résultante des modifications non appropriées et non cohérentes. A travers ces deux raisons, il a associé ces problèmes au vieillissement des logiciels :

- Inaptitude à suivre le marché en raison de la taille et de la complexité croissante,
- Rendement réduit en raison de la détérioration progressive de la structure,
- Diminution de la fiabilité en raison d'erreurs introduites lorsque des modifications sont apportées.

3.8 Conclusion

Dans ce chapitre, nous avons évoqué la progression des lois de l'évolution qui ont été initiées par Lehman. La prolifération des lois s'est stabilisée au nombre de huit qui sont depuis adoptées par la communauté du génie logiciel. Nous avons aussi mis en exergue les différentes méthodes formalisant cette opération par l'utilisation des techniques de chemin ou de transformation de graphe ou par application d'opérateur d'évolution. Cependant, l'utilisation des styles architecturaux est une procédure favorable pour modéliser l'évolution d'une architecture par extraction puis représentation des connaissances de l'opération. Le phénomène d'évolution touche à plusieurs domaines aussi bien industriels qu'académique que nous avons énumérés et positionnés l'intérêt fondamental derrière cette opération.

De plus, nous avons présenté un ensemble de référentiel d'évolution bien connus en ingénierie logicielle tel que le référentiel de Zachman, TOGAF, TEAF et autres. Les études taxonomiques pour l'évolution ont été introduites et décrites dont l'objectif est de servir comme un référentiel de réflexion puisqu'elle nous a permis de dévoiler les limites que posent les taxonomies existantes.

Dans le chapitre qui suit, nous allons présenter notre proposition du référentiel à travers l'identification des différentes dimensions nécessaires pour la description des architectures basées composants et/ou services. Notre proposition s'inspire du modèle de Zachman et d'autres

CHAPITRE 4

Référentiel d'évolution des architectures logicielles

« Nous devons accepter le changement mais conserver nos principes.»
- Jimmy Carter (1924-)

4.1 Introduction

Dans les chapitres précédents nous avons présenté le fondement théorique autour de l'évolution des logiciels. Tout système que ce soit orienté composant ou orienté service est contraint d'évoluer, d'être en état attentif et d'être prêt à réagir face à des nouvelles préoccupations souvent intégrées dans la phase de conception. Dans ce chapitre, nous allons introduire et illustrer les différents aspects dont se base notre référentiel de description de l'évolution. Nous rappelons que ce référentiel est principalement conçu pour analyser et comprendre l'évolution architecturale d'un système logiciel.

4.2 Référentiel taxonomique proposé

Dans ce qui suit nous nous focaliserons principalement sur l'aspect évolutif en premier plan pour dresser un Framework d'évolution pour les systèmes logiciels. L'aspect structurel est formulé ici pour spécifier les évolutions structurelles désignant que le changement à opérer est intrinsèque aux éléments constituant l'architecture logicielle par référence aux éléments architecturaux. Pour chaque aspect, nous présentons alors les dimensions qui définissent l'outil ainsi que les interactions prescrivant l'orthogonalité de ces dimensions.

4.2.1 Définitions

Dans cette section, nous réservons un grand intérêt à la définition et la précision de la signification des concepts de base représentant notre proposition.

4.2.1.1 Référentiel

Un référentiel permet de quantifier les choix d'une approche d'évolution architecturale par rapport à un certain nombre de dimensions. Présenter un Framework pour l'évolution dans un domaine revient à définir les dimensions sur lesquelles une approche ou une méthode d'évolution peut être évaluée, positionnée et finalement décider sur la catégorie d'appartenance. Il permet donc de représenter et de délimiter l'espace de solutions qu'il couvre. L'utilité d'un référentiel unifié offre la faculté de paysager les travaux d'évolution existants.

4.2.1.2 Aspect structurel

On désigne par l'aspect structurel d'une architecture logicielle la description des différents éléments qui la constituent par référence aux composants, services, connecteurs et autres. L'aspect structurel d'une architecture logicielle spécifie les différentes abstractions qui composent le système dans ces différents niveaux d'abstractions. C'est aussi, la collection des entités organisées pour fournir les fonctionnalités requises par le système. Entre autres, ces éléments doivent communiquer, coordonner et structurer toutes les opérations que le système doit réaliser dans le cadre des contraintes formulées pendant les phases de conception, de développement et de maintenance. Il est évident que l'épicentre de l'aspect structurel dans une architecture dépend directement du domaine où l'architecture logicielle est décrite. On trouve alors qu'elle peut se focaliser autour du:

- Composant dans une architecture orientée composant,
- Service dans une architecture orientée service.

4.2.1.3 Aspect comportemental

On désigne par l'aspect comportemental d'une architecture logicielle, la présentation et la description de l'activité de l'architecture. Cette activité est structurée en termes d'opérations qui peuvent être appliquées à un élément architectural. De même, ces opérations peuvent être atomiques pour décrire des opérations appliquant un seul opérateur d'évolution ou complexe pour traduire une combinaison d'opérateurs.

4.2.1.4 Aspect qualitatif

On désigne par l'aspect qualitatif d'une architecture logicielle l'ensemble des facteurs et critères mesurables pouvant donner un jugement sur une opération d'évolution. Les facteurs de qualité représentent des métriques qui sont souvent influencés au cours d'une opération d'évolution. Les facteurs peuvent être soit proportionnels soit disproportionnels entre eux, c'est-à-dire que l'augmentation de la valeur d'un des facteurs de qualité entrainera automatiquement l'augmentation (respectivement la diminution) des autres.

4.2.2 Dimensions du référentiel

Comme déjà mentionné à l'introduction de ce chapitre, cette appellation extrait ses fondements de la recherche d'un ensemble de dimensions de base capable de décrire et de comprendre les opérations d'évolutions menées pour faire évoluer l'architecture logicielle. Cependant, cette dernière est souvent formulée à l'aide de plusieurs descriptions dites «*descriptions d'architectures*» couvrant les aspects : *structurel, comportemental, qualitatif, technique* et autres. Ici, nous nous focaliserons sur les aspects: structurel, comportemental et qualitatif pour

déterminer les dimensions permettant à la fois d'analyser et de comparer une approche d'évolution. Le Framework adopté se focalise sur la description des différentes préoccupations formulées par Zachman (Zachman, 1987) pour la description d'un système d'information pour l'entreprise(Quoi ? Pourquoi ? Qui ? Où ? Quand ? Comment ?). L'idée était de considérer l'évolution comme un système d'information capable de fournir les connaissances nécessaires pour la compréhension et le positionnement d'une méthode d'évolution. Le référentiel proposé est structuré en six dimensions d'évolution qui seront développées par la suite. Ce référentiel décrit l'évolution de l'architecture et servira par conséquent comme un outil de compréhension de l'évolution de cette architecture (Gasmallah, Amirat, & Seridi, 2019).

Nous abordons les dimensions proposées et nous représentons l'appartenance de chaque dimension à l'aspect qui lui correspond. Nous avons estimé intéressant de développer des exemples illustrant les dimensions proposées au travers l'exemple de l'unité arithmétique et logique (UAL) d'un ordinateur.

4.2.2.1 Dimensions relevant de l'aspect structurel

Nous rappelons que les dimensions ont été extraites par l'adoption des préoccupations présentées par Zachman pour la description des systèmes d'informations d'une entreprise. Les questionnements *(Quoi ? Où ?)* sont de premier ordre. La première question présente là où est focalisée l'opération d'évolution. Par contre, la seconde démontre le niveau d'abstraction associé aux opérations à modéliser pour atteindre la cible escomptée. Ces deux dimensions dressent l'aspect structurel de l'évolution des architectures logicielles.

- Objet de l'évolution

Cette préoccupation vient en réponse à la question: *Quoi* évoluer? Elle identifie la cible principale sur laquelle porte l'évolution de l'architecture. Logiquement, les opérations d'évolution manient souvent deux cibles différentes selon des besoins exprimés. L'objet de l'évolution présente le support architectural sur lequel la modification à effectuer est opérée autrement dit que l'objet de modification désigne le sujet de l'intervention évolutive. L'objet de l'évolution est la réification des éléments architecturaux considérés comme entité fondamentale de l'intervention d'évolution et qui est assujettis aux modifications. Par exemple pour un ADL, l'objet de l'évolution fait référence à l'entité considérée de première classe. Par conséquent, l'évolution dans un ADL peut concerner les différents concepts de description d'architecture par référence aux composants, services, connecteurs, configurations et interfaces. A la lumière de ce qui a été dit nous pouvons faire une séparation nette entre des modifications qui sont orientées artefacts et d'autres qui sont orientées processus.

- *Artefact :* est l'abstraction de tout élément appartenant à la structure de l'architecture. Il peut s'agir d'une architecture, d'un composant, d'un objet, d'un service ou autres.

Exemple : IMoTEP[1] (Integrated Model-based Testing of Continuously Evolving Software Product Lines) étudie la coévolution des modèles de ligne de produits logiciels (SPL) et des artefacts de test, en particulier la propagation de modifications dans les modèles SPL.

- *Processus :* Conformément à la norme ISO 9000, un processus est :

[1] Voir http://www.dfg-spp1593.de/imotep pour plus de détails.

«*Un ensemble d'activités corrélées ou interactives qui transforment des éléments d'entrée en éléments de sortie*» (ISO, 2009).

Le processus représente donc l'abstraction de l'ensemble des opérations ou méthodes isolées ou combinées à appliquer sur un processus dans l'objectif est d'apporter une valeur ajoutée à l'activité elle-même.

Exemple: Les langages de conception d'entreprise (EML : Entreprise modelling Languages) définissent les concepts *Constructs* chargés de décrire les modèles: de l'activité humaine de l'entreprise, les processus métiers et technologie qui lui sont associés et leurs supports(Alves, de Campos, & Souza, 2016). L'évolution d'un *construct* dans la partie relative au processus métier représente une évolution de processus.

- Niveau d'évolution

Le référentiel préconise qu'à un instant donné l'architecture est assignée implicitement ou explicitement à un niveau de modélisation et un niveau d'abstraction et par conséquent, elle peut pallier plusieurs positions, au cours de son évolution. Les systèmes logiciels suivent plusieurs phases de modélisation à différents niveaux avant la livraison finale. Deux niveaux hiérarchiques d'architecture sont principalement identifiés dans la littérature en génie logiciel: niveau de modélisation et niveau d'abstraction. Ces deux niveaux répondent explicitement à la question où?. Communément, deux types de niveaux de hiérarchie sont manipulés en littératures :

- *Niveau de modélisation :* Les systèmes logiciels sont mappés à travers plusieurs niveaux d'hiérarchiques au cours de leurs évolutions par référence à la pyramide d'architecture des modèles. En effet, ce passage représente soit une factorisation (Bottom-Up) i.e. le passage du niveau de construction du code (niveau inférieur) au niveau de construction de modèles (niveau supérieur) soit un raffinement pour le passage inverse c.à.d. que le passage se fait du niveau supérieur au niveau inférieur (Top-Down). Nous avons déjà vu que le niveau de modélisation fait référence à l'un des quatre niveaux (de M_0 à M_3) définis par l'OMG (Bézivin, 2003; Friedenthal, Moore, & Steiner, 2008).

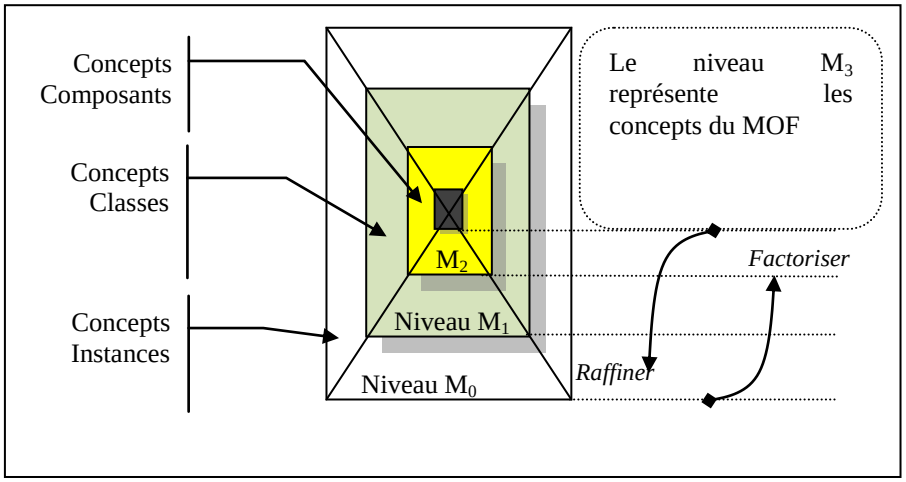

Figure 4.1- Niveaux de modélisation v.s concepts.

Il est à noter qu'à l'exception du niveau méta-méta-modélisation (M_3), l'évolution peut être réalisée dans un ou plusieurs niveaux de modélisation par indication aux niveaux M_0, M_1 et M_2.

Par évidence, le passage entre ces niveaux nécessite l'utilisation de différents ensembles de concepts et d'outils liés aux niveaux associés.

Par exemple, un changement d'architecture peut s'opérer sur un même niveau de modélisation ***(e.g. comme une transformation d'une méta-architecture en C2 sous une représentation UML2.0)*** ou qu'il suscite plusieurs niveaux de modélisation qui se traduit par la manipulation de nouveaux concepts liés à deux niveaux différents et on peut dire alors que l'évolution est orientée niveau de modélisation.

Par exemple, *l'instanciation d'un composant d'une architecture considéré comme type en plusieurs instances d'objets représente une évolution au niveau modélisation*. La Figure 4.1 illustre l'association des concepts par niveau de modélisation.

Une autre illustration concrète relative à l'exemple d'unité logique arithmétique (ALU), comme indiqué précédemment: *Le code d'implémentation associé à une représentation de circuit additionneur présente un niveau de modélisation inférieur (M_0), tandis que le diagramme de conception logique associé est considéré comme un niveau supérieur de modélisation (M_1).*

- *Niveau d'abstraction :* le niveau d'abstraction est généralement utilisé par les experts pour traiter les problèmes présentant un certain degré de complexité. Ceci est encore souvent utilisé pour résoudre des problèmes liés à la modélisation logicielle. De ce fait, une solution est réalisée via une série de descriptions de modèles à un certain nombre de niveaux d'abstraction et qui représentent une alternative de transformations horizontales de l'architecture ou de modèle(Gardner, Griffin, Koehler, & Hauser, 2003). En définissant un niveau d'abstraction approprié pour une phase d'application donnée, la quantité d'informations sémantiques, considérée en reléguant des détails non pertinents à un niveau d'abstraction inférieur, est limitée (M. Oussalah, 2014). Ce qui signifie, que le nombre de niveaux d'abstraction affecté par un modèle dépend de la complexité du modèle lui même. L'évolution peut porter sur :

- Un même niveau d'abstraction : donc les concepts ne subissent aucune dérivation,

- Plusieurs niveaux d'abstraction: les nouveaux concepts se présentent comme une dérivation premiers ou/et manipulés avec des outils différents indépendamment du niveau de modélisation de l'architecture à évoluer.

Par exemple*, raffiner une classe existante en plusieurs autres classes spécifiques (sous-classes), représente une autre abstraction dans un même niveau de modélisation. En pratique, les organigrammes, algorithmes et pseudo-codes souvent utilisés comme point de départ pour la création d'un programme d'ordinateur correspondent à différents niveaux d'abstraction intégrés au niveau de modélisation M_0.*

Un autre exemple d'ALU, *un circuit additionneur à 8 bits est représenté par un ensemble d'additionneur à 1 bit interconnecté. En raffinant cette représentation par l'introduction des portes logiques nécessaires à l'implémentation d'un additionneur à 1 bit, plus de détails est introduit. Cette nouvelle représentation reflète une nouvelle abstraction du système.*

Il est donc intéressant de noter que la relation entre les niveaux de modélisation et d'abstraction souligne le non-isomorphisme entre les deux concepts. Cela indique qu'un niveau de

modélisation peut être composé de plusieurs niveaux d'abstraction (voir Figure 4.2). De plus, chaque objet d'évolution est assujetti à être modifié sur plusieurs niveaux hiérarchiques.

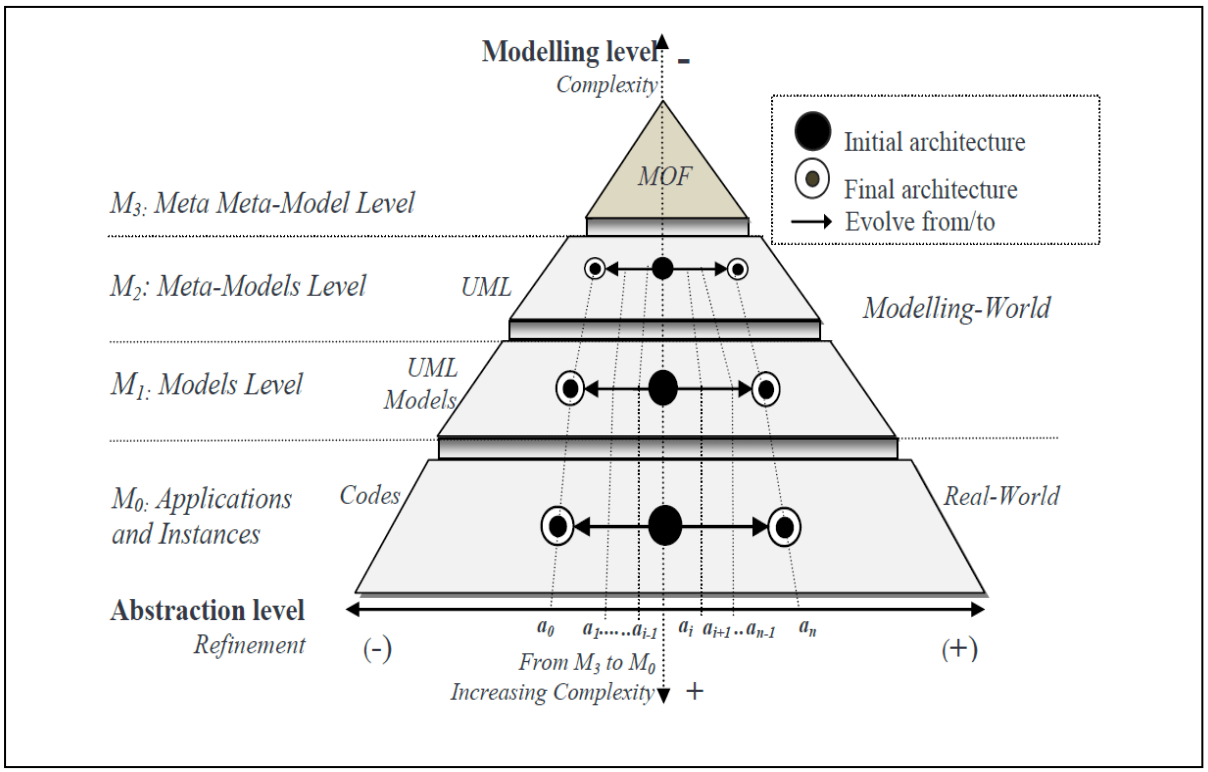

Figure 4.2- Projection des niveaux abstraction sur la pyramide des niveaux de modélisation.

- Temps dévolution

Cette dimension confirme l'importance de l'aspect temporel en termes de modifications d'architecture (quand?). Cela indique que les mécanismes d'évolution se produisent à de différents moments des phases de développement logiciel. D'un point de vue architectural, le temps d'évolution peut être lié à un ou plusieurs niveaux hiérarchiques. Cependant, l'environnement du système a un impact profond sur le temps et les outils à utiliser lors de l'évolution du logiciel.

Par exemple, l'évolution d'architecture en cours d'exécution doit assurer la reconfiguration et l'intégrité du système à tous les niveaux du logiciel (Oreizy & Taylor, 1998). Cependant, la majorité des études de recherches ont adoptées en termes de temps les métriques de temps de conception et de temps d'exécution (Ahmad, Jamshidi, & Pahl, 2014). Les deux métriques pallient toutes les autres mesures de temps à savoir le temps statique, le temps de chargement (temps de construction ou du temps des éditions de liens) et le temps d'initiation et le temps d'exécution concepts qui ont été utilisés adopté pour le changement du logiciel (Buckley, 2005), (Bass, Clements, & Kazman, 2003). De plus par considération de cette dimension, nous pouvons identifier deux grandes catégories d'approches en littérature synthétisant l'aspect temporel en approches anticipées et approches non anticipées. Les premières sont adoptées lorsque les modifications requises ont été recensées à priori. De ce fait, elles sont intégrées et prises en considération pendant les premières phases du cycle de vie du système. Par contre, les approches non-anticipatives dénotent la capacité de modifier l'architecture en raison d'un besoin

exceptionnel ou occasionnel non prévu d'avance. En se focalisant sur une vue purement architecturale, le temps de l'évolution peut s'opérer en (Gasmallah, Amirat, & Seridi, 2018):

- *Temps de conception :* la prévision de l'évolution aux premières étapes de la conception permet d'améliorer et d'étendre l'architecture du système. A ce stade, l'évolution est appelée évolution statique ou évolution anticipée puisque l'évolution de l'architecture est planifiée durant les phases de spécification et/ou de conception du système. L'architecture dirigée par les modèles est un bon exemple de méthodes prenant en compte l'évolution, en particulier les approches de coévolution.
- *Temps réel :* L'évolution peut avoir lieu pendant que le système est en cours d'exécution (Oreizy & Taylor, 1998). Jusqu'ici, l'évolution au moment de l'exécution est considérée comme un sujet majeur pour l'adaptation de l'architecture (Mens et al., 2005). Cependant, une évolution dynamique doit assurer la propagation des modifications ou changements réalisées quelque soit le niveau d'intervention pour qu'elle soit directement implémentée dans le système en état d'exécution. L'anticipation considérée comme un facteur temporel est un déterminant de base pour scinder les approches dynamiques en approches anticipées et curatives. L'évolution en temps réel comprend les métriques suivantes:
  - *Temps de compilation* où le code source du système implique nécessairement un processus de recompilation pour générer le nouveau système après chaque modification,
  - *Temps de chargement* qui désigne le temps écoulé entre l'arrêt du logiciel et sa nouvelle exécution pour que les changements prennent effet (Kniesel, Costanza, & Austermann, 2001) et
  - *Temps dynamique* est considéré comme une sous-dimension prometteuse (Mens, et al., 2005), dans laquelle les objets sont mis à niveau lorsque le système fonctionne, à la lumière d'un logiciel d'adaptation ou de reconfiguration. L'évolution en temps a un lien intrinsèque avec l'évolution dynamique où les modifications de l'architecture sont contraintes d'être opérées pendant l'exécution du système.

Par exemple. *Un logiciel d'auto-reconfiguration permet de modifier au moment de l'exécution le comportement approprié en réponse aux modifications de l'environnement. Ainsi, des éléments architecturaux peuvent être ajoutés dynamiquement dans divers systèmes logiciels sans entraîner le redémarrage du système. Ceci est considéré comme une évolution temporelle dynamique.*

- Intervenant d'évolution

Les intervenants contribuent à l'amélioration de l'architecture à plusieurs niveaux de responsabilités. Le rôle d'un intervenant dépend à la fois de l'expertise de ce dernier aussi bien que de la nature du défi auquel le système est confronté (Terho, Suonsyrjä, Systä, & Mikkonen, 2017). Cette dimension est associée à la question qui ? Les intervenants englobent tous ceux qui gèrent l'évolution du système logiciel lui-même en faisant référence à l'équipe de développement comprenant des architectes, des concepteurs, des développeurs, etc. (Mens, et al., 2005). Ainsi qu'à l'auto-automatisation pour traiter les problèmes automatisés. Cette dimension dépend principalement de l'élargissement de l'expertise et des connaissances des acteurs de la prise de décision, en particulier des architectes, ce qui leur permet d'évaluer les besoins futurs éventuels en intégrant les changements anticipés aux différents stades du cycle de vie et niveaux hiérarchiques (Jazayeri, 2005), (Clements et al., 2002).

Par conséquent, ils ont besoin d'une bonne définition des exigences et d'un retour précis d'information de la part de tous ceux qui sont impliqués dans ce système logiciel afin de dissiper le chaos (Rising & Janoff, 2000). Par exemple, *les programmeurs de logiciels mettent en œuvre des programmes conformes aux nouvelles spécifications*. La Figure 4.3 présente le digramme associé à la dimension des intervenants. Un intervenant lui est associé un ou plusieurs rôles dans le l'activité d'évolution. De plus, l'agrégation entre l'intervenant et la classe nature permet de déterminer la manière dont le système va opérer l'intervention (automatique, manuelle ou semi-annuelle).

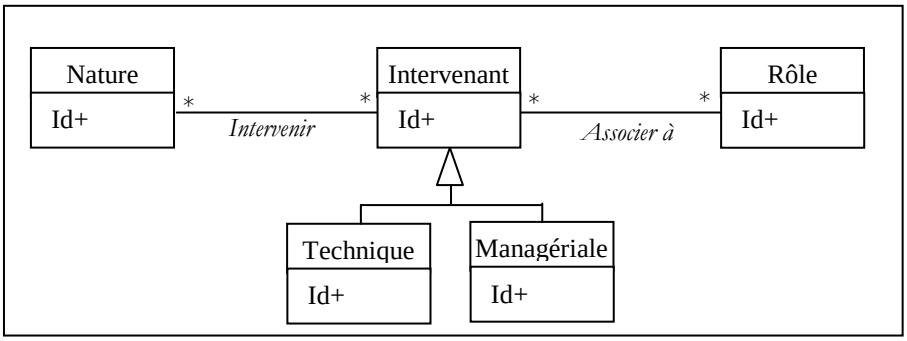

Figure 4.3- Dimension intervenant d'évolution.

4.2.2.2 Dimensions relevant de l'aspect comportemental

Ces dimensions servent à identifier l'aspect comportemental de la méthode d'évolution. On s'intéresse ici aux éléments nécessaires pour comprendre le comportement de l'architecture évoluée ou à faire évoluer.

-Type de l'évolution

Cette dimension exprime la motivation dernière laquelle l'évolution est sollicitée ou planifiée. Le type d'évolution peut être de type curatif ou de type préventif. Le type curatif peut être assuré pour des nouveaux besoins qui apparaissent de façon imprévue durant le cycle de vie, ou lorsque ces besoins ont été mal ou peu spécifiés et que l'environnement permet enfin de les enrichir ou de les corriger. Le système doit être en mesure de les prendre en compte en les intégrant ou en les modifiant. Cette évolution, dite également non-anticipée, n'est pas considérée dans notre étude du faite qu'on traite que l'anticipation de l'évolution d'une architecture. Nous analysons ici en détails le type d'évolution préventive qui peut être assurée pour les besoins d'évolution pris en compte lors de la phase d'analyse et spécifiés durant la conception. Deux réflexions ont caractérisé le type de l'évolution: la première selon le raisonnement adopté pour réaliser l'évolution que nous désignons ici par les formes principales d'évolution. La deuxième est selon la continuité de l'architecture évoluée et que nous indiquerons dans la suite par les catégories principales d'évolution.

- *Formes principales d'évolution :* caractérisent le raisonnement suivi, et qui diffère selon l'hypothèse qu'on dispose ou non de l'architecture cible. On trouve dans la littérature deux formes différentes d'évolution. La première traduit l'acheminement et le raisonnement adopté pour résoudre le problème d'évolution. Dans cette forme, nous trouvons:

- Evolution ouverte (Open evolution) : Cette évolution tente à partir d'une architecture initiale, d'induire une solution meilleure (*Open-ended*) à travers la manipulation de plusieurs artefacts ou processus tout en respectant un ensemble de contraintes ou d'invariants.
- Evolution fermée (Closed evolution) : Cette évolution se fait par déduction c.-à-d. déduire qu'elles sont les opérations possibles à appliquer pour aboutir à un résultat prédéfini (*Close-ended*). Malgré que ces approches soient prometteuses, elles soufrent de manque d'outils et de pratiques d'assistance aux architectes d'une part et d'autre part du manque de méthodes pertinentes d'analyse pour formaliser les décisions automatiques qui sont souvent faites d'une manière ad-hoc.

La deuxième forme prévoit deux types d'évolution selon (C. Oussalah, 1999):

- Evolution avec rupture (Break) : désigne que les interventions sont directement appliquées sur l'architecture de départ sans avoir à les stocker. Cette manière de procéder donne la priorité à la réaction du système sans se soucier à enregistrer l'historique de l'opération, qui présente parfois une connaissance très importante.
- Evolution continue (Seamless) : dénote une évolution avec préservation de la trace d'évolution i.e. que l'architecture garde les propriétés de l'architecture de départ ainsi que les opérations effectuées pour atteindre la nouvelle architecture. Cette continuité d'évolution est assurée par la sauvegarde des séquences d'opérations appliquées.

- *Catégories principale d'évolution :* Cette catégorie considère que l'activité d'évolution est entamée pour répondre à une motivation quelconque. Nous voulons dire par motivation la cause principale derrière cette opération d'évolution qui peut être de plusieurs types. Cependant dans la littérature et selon (Williams & Carver, 2010), il a été constaté que les méthodes d'évolution traitent principalement les modifications correctives, perfectives et adaptatives. C'est la raison pour laquelle que nous considérons ces trois modifications comme les trois motivations principales d'évolution.
  - Corrective : désigne que l'évolution est introduite pour effectuer une correction suite à une détection d'erreur, par exemple dans (Medvidovic, Rosenblum, & Taylor, 1999), les auteurs décrivent un ADL pour prendre en charge les modifications de l'architecture dues à la modification de l'environnement et vise versa.
  - Adaptative : représente la catégorie d'évolution évoquée par souci d'adaptation de l'architecture existante suite à la présence de nouvelles décisions ou de changement de l'environnement d'évolution de l'architecture. Par exemple, l'application constante du changement dans le temps et la planification d'un cycle de détection de circonstances de changements (Oreizy et al., 1999) répondent parfaitement à une motivation adaptative.
  - Perfective : celle-ci reflète les types d'évolutions demandées par les intervenants afin d'améliorer et de rentabiliser le système actuel suite à des nouvelles opportunités. Par exemple, l'intégration des relations de dépendance d'interopérabilité par exploration des techniques de visualisation (Bohner, 2002) a permis une navigation plus efficace pour la gestion des impacts de changements pour les applications distribuées.

En effet, un type de l'évolution ouvert ou fermé peut être soit en rupture ou en continu. La Figure 4.4 présente les différents types d'un type de l'évolution (au sens d'un diagramme BPMN) qui peuvent être appropriés à une évolution d'une architecture et qui sont annotés dans la suite du manuscrit comme suit:

- POB: Préventif Ouvert en Rupture,
- POS: préventif Ouvert en Continu,
- PCB: Préventif Fermé en Rupture,
- PCS: Préventif Fermé en Continu.

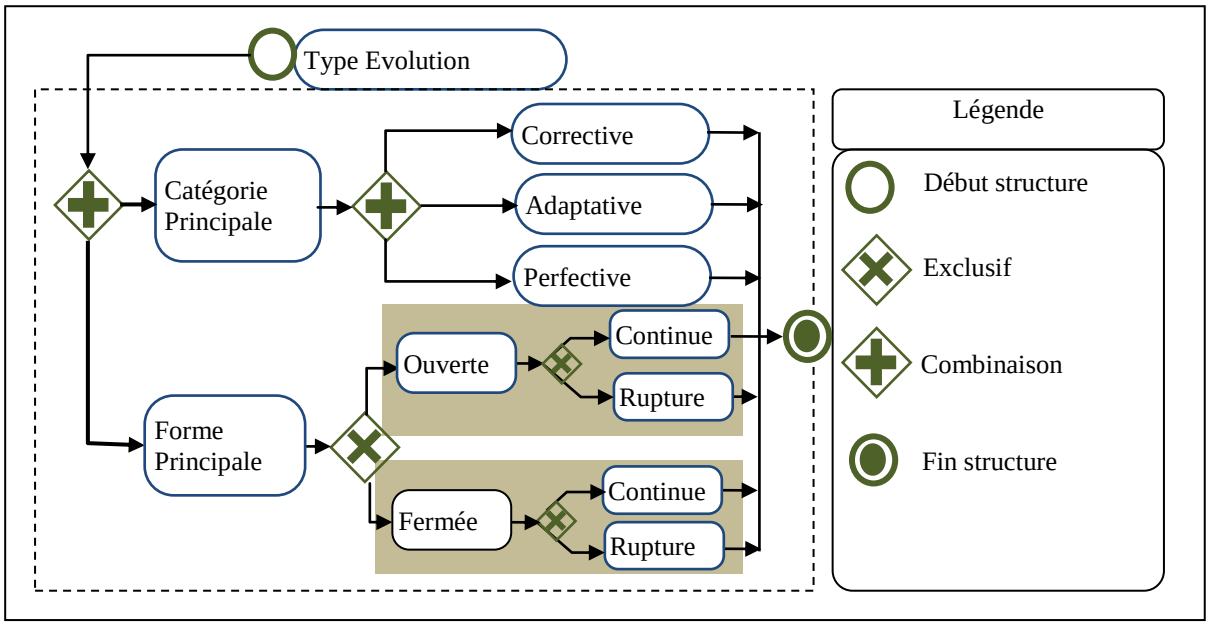

Figure 4.4- Structure du type d'évolution.

- Mécanique opératoire d'évolution (MOE)

La mécanique opératoire d'évolution (MOE) est introduite pour faire référence au comportement général des changements qui sont opérés sur l'architecture du système. C'est aussi la technique employée en prenant en compte uniquement les niveaux hiérarchiques qui sont considérés dans le processus d'évolution. L'MOE nous permettra donc de dévoiler l'impact de l'évolution sur les niveaux hiérarchiques tout au long du processus de changement. Principalement, les approches de résolution d'évolution adhèrent généralement à l'une des deux méthodes principales suivantes:

Evolution orientée réduction : C'est une évolution réductionniste, le processus parcours un chemin d'évolution prédéfini; jusqu'à ce que la solution soit satisfaite (modèle évolué). Selon (Brooks, 1989), l'approche «réductionniste » est définie comme une approche classique de résolution de problème pour laquelle la tâche globale de résolution est décomposée en sous-tâches comme l'*instanciation des classes en objets par exemple.* Un autre exemple de l'UAL, *les instances correspondant à la classe d'un additionneur dans un modèle de circuit électronique sont appelées fonctions/procédures déclarées au niveau inférieur de conception. Le 4008 est une*

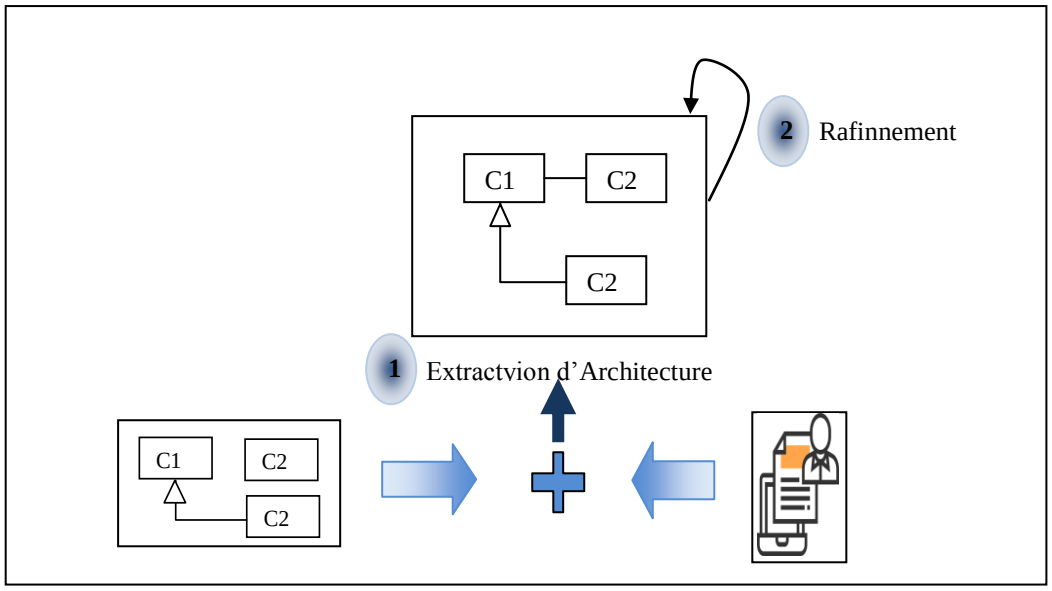

Figure 4.5- Processus émergence : de la source vers les modèles (1-extraction 2-raffinement).
[Source (Ducasse & Pollet, 2009)]

- *Evolution orientée émergence :* Dans l'approche «émergentiste» au contraire, lors de l'évolution, le processus construit le cheminement vers une solution (Figure 4.5). Dans (De Wolf & Holvoet, 2004), les auteurs définissent l'émergence comme:

"*a system exhibits emergence when there are coherent emergent at the macro-level that dynamically arise from the interactions between the parts at the micro-level*".

L'émergence dénote que le niveau de modélisation ou d'abstraction de l'architecture de départ bascule vers les niveaux supérieurs une fois évoluée. L'émergence pouvant se révéler un moyen de passage entre l'activité du "micro-niveau" et celui du "macro-niveau". Par exemple, ***la catégorisation des classes en super classes de niveau supérieur***. L'application de l'émergence dans l'exemple d'UAL traduit par exemple l'agrégation de plusieurs additionneurs (additionneur 1 bit), il est possible de créer des additionneurs plus importants (additionneur 32 bits).

Cependant, la Figure 4.6 expose le modèle conceptuel de notre référentiel à la base des six dimensions introduites dans ce chapitre. Nous avons mis au centre la méthode architecturale d'évolution qui est composée des six dimensions que nous avons développées précédemment. Les différentes métriques sont représentées à travers des relations d'héritages. La question associée à chaque dimension est bien définie sur le lien approprié.

4.2.3 Interactions des dimensions

La Figure 4.6 présente les six dimensions sur lesquelles un modèle d'évolution architecturale peut être élaboré. Le référentiel utilise des relations de composition où toutes les dimensions sont requises pour l'évaluation. En effet, toutes les dimensions peuvent être évaluées en répondant aux questions appropriées, de manière implicite ou explicite. La composition entre la méthode d'évolution d'un côté et la dimension temporelle (respectivement la mécanique opératoire d'évolution) révèle qu'une méthode donnée résout le problème de l'évolution en se concentrant sur le temps d'exécution ou sur temps de conception (respectivement sur le raisonnement de réduction ou d'émergence).

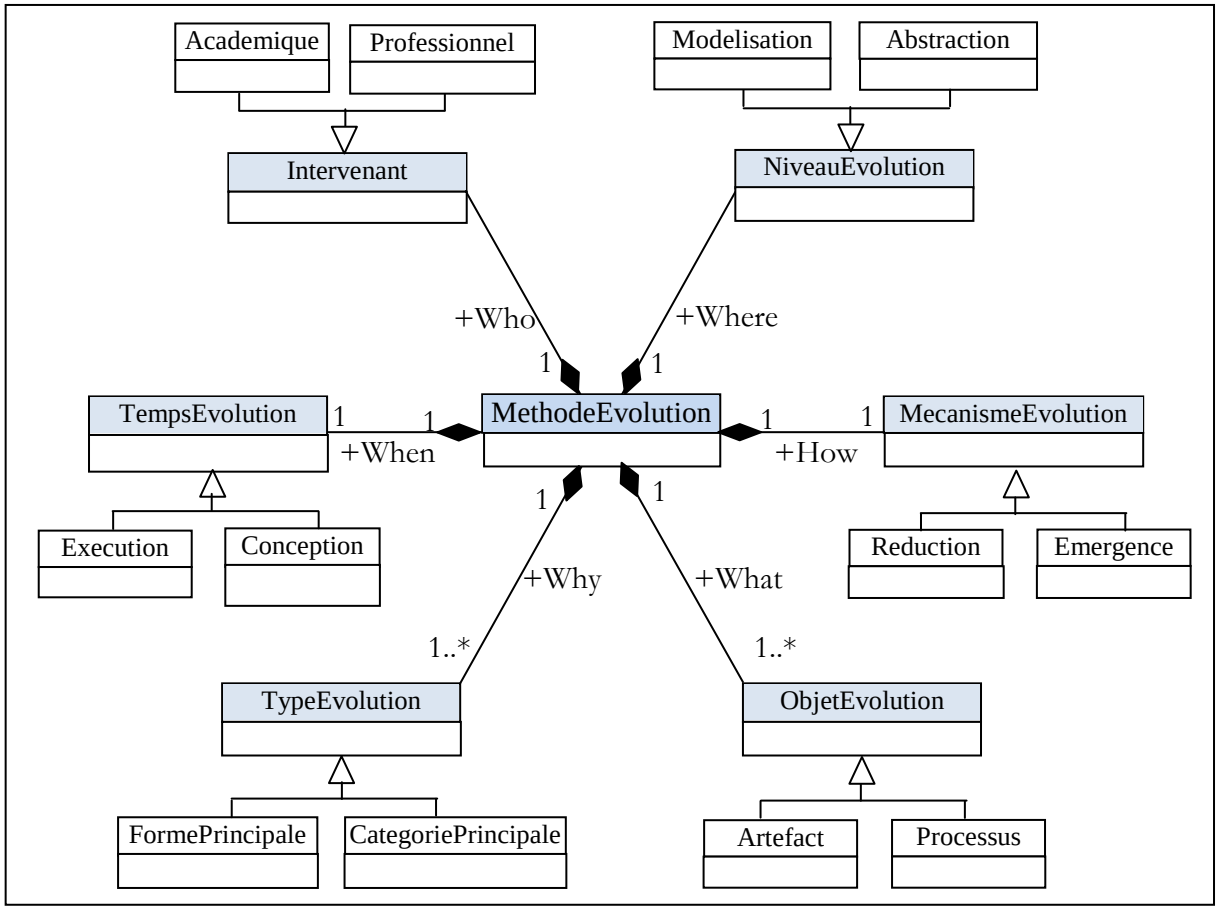

Figure 4.6- Le modèle conceptuel du référentiel proposé.

Cependant, la multiplicité 1,* sur une dimension donnée implique la nécessité d'avoir une valeur. Deux types de relations peuvent être générés en comparant une paire de dimensions incluant la dimension à elle-même. Ceci est mieux expliqué dans le tableau 4.1. La première relation est *intra-dimension* qui indique si les métriques appartenant à la même dimension s'excluent mutuellement (⊥). Dans ce cas, la métrique est quantifiée à un (i.e. 100%) lorsque l'approche traite explicitement la métrique considérée l'autre métrique est forcement égale à zéro. Dans le cas où les métriques sont combinées (⊙), les métriques sélectionnées sont donc calculées relativement à la somme de toutes les métriques (Gasmallah, Amirat, & Seridi, 2019).

Table 4.1- Relations inter et intra dimensions du référentiel proposé.

Dimensions	1.Objet	2.Type	3.Niveau	4.Temps	5.Intervenant	6.Mec.Oper.
1.Quoi ?	⊥					
2.Pourquoi ?	∅	⊙				
3.Où ?	∅	∃	⊙			
4. Quand ?	∅	∃	∃	⊥		
5.Qui ?	∅	∅	∃	∃	⊥	
6.Comment ?	∅	∅	∅	∅	∅	⊥

∅ : Pas de dépendance- ⊥ : métriques exclusives- ∃ : dépendance faible- ⊙ métriques combinées

Par exemple, une MOE d'une approche d'évolution peut être définie exclusivement en *Réduction* ou *Émergence*, tandis que les niveaux hiérarchiques affectés par l'évolution peuvent être:

les niveaux d'abstractions, de *modélisations* ou les deux à la fois. De plus, la dimension type est combinée (50% pour chaque sous-dimension: catégorie principale et forme principale) et que les valeurs de chaque sous-dimension sont distribuées équitablement.

Le deuxième type de relation est une *inter-dimension* qui identifie les dépendances possibles entre des métriques appartenant aux différentes dimensions. Les cellules avec le symbole (∃) dans la Table 4.1 montrent l'existence d'une faible dépendance entre le temps, les niveaux hiérarchiques, les intervenants et le type d'évolution en raison du chevauchement potentiel entre certaines valeurs telles que: temps d'exécution, niveaux inférieurs d'abstraction, type de correction et les programmeurs. Sur la base des relations analysées, il est observé que les métriques ont une faible dépendance entre les dimensions proposées.

4.3 Aspect qualitatif

Il est important qu'un système logiciel puisse projeter les impacts de l'évolution par la considération de l'aspect qualitatif du produit. En effet, une évolution doit non seulement satisfaire des besoins fonctionnels i.e. d'utilisabilité mais doit aussi assurer un ensemble de paramètres qualitatifs fixés par les intervenants. C'est dans cette optique que cette section est introduite pour faire valoir le Framework proposé en matière de satisfaction des objectifs de qualité. Nous avons regroupé les critères de qualités en trois grandes capacités (Gasmallah, Amirat, & Oussalah, 2016) comme l'illustre la Figure 4.7.

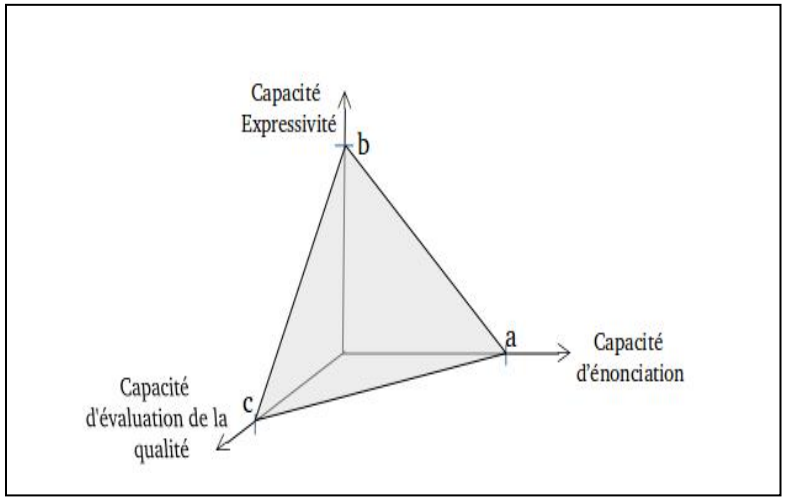

Figure 4.7- Positionnement des capacités d'un modèle d'évolution.

4.3.1 Capacité d'énonciation de l'évolution

Selon les travaux de Oussalah (C. Oussalah, 1999), les objectifs de qualité se formalisent selon une pyramide de besoin se basant sur une vue macro de la qualité. Cette vue peut être approfondie selon des degrés dits *degrés de qualité* sachant que plus ils sont plus élevés plus ils s'approfondissent dans la technique ou la méthode d'évolution associée.

Selon une vue architecturale, les paramètres énonçant l'évolution d'une architecture se résument principalement dans les caractéristiques de (i) formulation de l'évolution, (ii) gestion des

impacts de l'évolution et (iii) gestion de la traçabilité de l'évolution. Nous définissons ci-après ces caractéristiques associées à la capacité du besoin d'énonciation.

4.3.1.1 Formalisation

Comme nous l'avons déjà mentionné, nous désignons par éléments évolutifs l'ensemble des éléments architecturaux pouvant subir des changements par l'intermédiaire de l'application d'un certains nombre d'opérateurs d'évolution. Ces opérateurs sont de deux natures : atomiques ou complexes. Les opérateurs atomiques sont des opérateurs simples tels que l'ajout ou suppression ou modification d'un élément. Ces opérateurs peuvent être combinés pour construire d'autres opérateurs plus complexes tels que : l'opération de remplacement qui peut être une combinaison des opérateurs atomiques de création et de suppression.

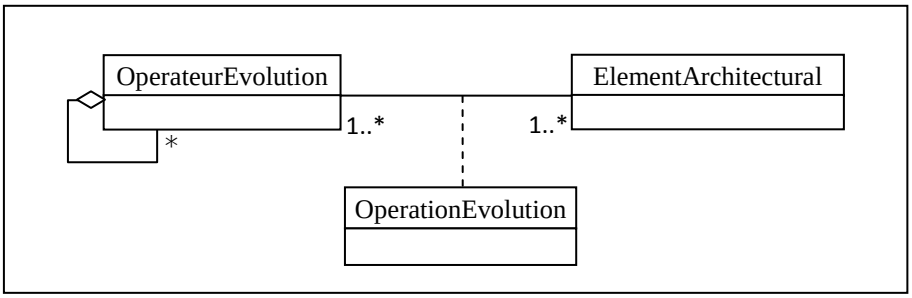

Figure 4.8- Modélisation des opérations d'évolution architecturales.

Nous qualifierons désormais une opération d'évolution comme l'association d'un opérateur avec les différents opérandes qui ne sont autres que des éléments évolutifs nécessaires à son exécution. La Figure 4.8 illustre cette vue de formalisation des éléments évolutifs. Pour plus d'illustration, nous avons dressé un récapitulatif des opérateurs d'évolution les plus abordés dans la plus part des ADLs traitant la problématique d'évolution d'architecture logicielle.

Table 4.2- Formalisation des éléments évolutifs pour une architecture logicielle.

Opérateur	Paramètres	Type	Description
ADD	2 EA	Atomique	Permet d'ajouter un élément architectural définit par son nom à un autre élément
Modify	1 EA		Modifie le nom ou la valeur de l'élément architectural
Remove	1 EA		Supprime un EA d'un autre de même type.
Replace	1 EA		Remplace un EA par un autre du même type.
Connect	2 EA		Etablit une association entre les deux éléments passés en paramètres.
Disconnect	2 EA		Suspend la connexion établit entre deux éléments architecturaux.
Split	1 EA et une liste	Composite	Dissocie un élément spécifié par un nom et son type en plusieurs autres selon une liste de noms d'éléments donnée en paramètre.
Unify	Une liste et 1 EA		Associe plusieurs EA donnés une liste de noms dans un seul EA
Version	1 EA		Fournit une nouvelle version d'un élément spécifié avec son type en paramètres

Légende- EA : Elément Architectural.

La Table 4.2 décrit chaque opérateur d'évolution, les types associés qu'ils soient atomiques ou composites ainsi que le nombre et la nature des paramètres qui lui sont utiles pour son exécution. Il est intéressant de noter que ces opérateurs fournissent une description de haut niveau d'abstraction. Cela permet d'assurer une faible dépendance entre le langage de description de l'architecture utilisé et ses opérateurs.

4.3.1.2 Impacts d'évolution

Par définition, l'impact désigne l'ensemble des effets engendrés par un élément ayant subit une évolution de n'importe quelle taille sur les autres éléments. Ces effets doivent être identifiés en termes d'une séquence de changements entre l'élément évolué et les autres éléments entrainés systématiquement dans le processus d'évolution. Ce mécanisme d'entrainement est appelé processus de propagation d'impact ou processus à « effet de bord » pour statuer qu'une évolution locale d'un élément peut se manifester sur plusieurs autres et devient par ce fait globale.

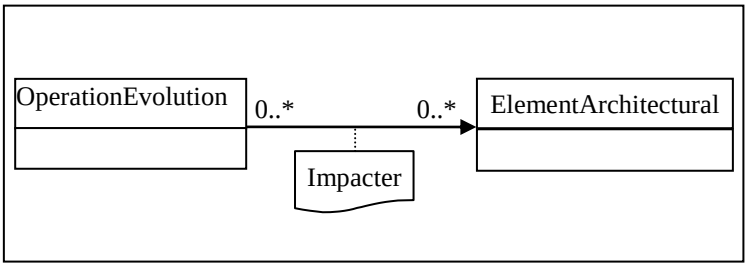

Figure 4.9- Diagramme de propagation des impacts.

L'intention d'identifier la propagation des impacts est essentielle pour définir la portée d'une opération d'évolution pour pouvoir aider les architectes à définir les mécanismes adéquats pour le raisonnement sur ces impacts (Figure 4.9). La propagation des impacts est un facteur prépondérant dans la compréhension de l'énonciation de l'évolution. Par sa nature, les modifications entrainées peuvent être à l'origine de plusieurs changements de l'architecture sur ses différents aspects. Il est alors impératif d'évaluer ces impacts en considérant les vues multiples de l'architecture par référence aux aspects: structurel, comportemental, qualitatif et managérial. Selon des études dans le domaine du «S*oftware-change*» (Basson, 1998), (Goknil, Kurtev, van den Berg, & Spijkerman, 2014), (H. Zhang et al., 2014), nous avons identifié l'ensemble des impacts selon leurs types pour un changement donné. Cette typologie comporte:

- Les impacts portant sur l'aspect structurel,
- Les impacts relatifs à l'aspect logique,
- Les impacts purement fonctionnels,
- Les impacts relevant de l'aspect comportemental,
- Les impacts pour la performance comme par exemple les impacts de coût de composant (Bohner, 2002).
- Les impacts liés à une vue qualitative (Aleti, Buhnova, Grunske, Koziolek, & Meedeniya, 2013).

Ces aspects révèlent un impact critique des choix architecturaux sur la création d'une conception pouvant répondre aux exigences de multiples dimensions de performances, parfois

concurrentes (fonctionnalité, rapidité, facilité d'utilisation, fiabilité, évolutivité, etc.) (MacCormack & Sturtevant, 2016).

Dans cette thèse, nous prêtons un intérêt spécial aux propagations des différents types et donc d'assurer une bonne compréhension de l'évolution. Car les impacts structurels reflètent un changement effectif des éléments architecturaux ce qui influe sur la structure globale de l'architecture du système. Par exemple, *la modification de la structure interne d'un élément architectural comme une configuration ou une interface d'un composant ou connecteur s'inscrit dans le cadre d'une évolution structurelle*. Cependant, les impacts comportementaux agissent sur l'activité ou le processus d'activité sous contraintes que les niveaux hiérarchiques respectent le contexte du comportement fonctionnel (Fielding & Taylor, 2000). Par exemple, Garlan et Shaw (Garlan & Shaw, 1993) ont décrit les avantages du style architectural Pipe&Filter qui permet aux architectes de premièrement comprendre les entrées/sorties du système à travers une simple composition des filtres et aussi de prendre en charge la réutilisabilité comme caractéristique du style. Alors, deux filtres peuvent être reliés ensemble à condition que les données transmises coïncident entre les deux. Cette composition permet aussi une amélioration et une évolution aisée du comportement de ce type de système. Sur la plan qualitatif, (Breivold, Crnkovic, & Larsson, 2012) ont évalué l'impact qualitatif de la qualité de l'évolution logicielle sur des sous ensembles d'études. En effet, ils précisent que l'impact d'évolution est un souci exprimé par plusieurs catégories d'études à savoir :

- Etudes considérant la qualité durant la conception : Ces études se distinguent par la focalisation sur des attributs de besoins comme la simulation et les modèles mathématiques (Bengtsson & Bosch, 1998), les techniques expérimentales (Christensen & Hansen, 2010), les méthodes orientées NFR objectifs (Chung, Cooper, & Yi, 2003; Chung, Nixon, Yu, & Mylopoulos, 2012) …etc.) ou les scénarios d'évolution comme le modèle analytique de mesure de la qualité (Bagheri & Gasevic, 2011), les scénarios centrés intervenants (Fenton & Bieman, 2014; Rozanski & Woods, 2012)…etc.) ou par l'étude des facteurs influençant l'évolution comme les techniques d'optimisation par interviews (Al-Naeem, Gorton, Babar, Rabhi, & Benatallah, 2005), les techniques orientées objectifs commerciaux (Clements & Bass, 2010), orientée propriétés de contraintes conceptuelles (Van den Berg, Tang, & Farenhorst, 2009), …etc.).

- Etudes considérant la qualité du point de vue niveau architectural : Quant à ces études, elles se basent sur les expériences des utilisateurs (interviews (Bouwers & Van Deursen, 2010), questionnaires (Fricke, Gebhard, Negele, & Igenbergs, 2000), rencontres (Svahnberg, 2004) …etc., les scénarios et des métriques bien spécifiques (brainstorming) (Bengtsson, Lassing, Bosch, & van Vliet, 2004), estimations pondérées (Lassing, Bengtsson, Van Vliet, & Bosch, 2002), procédure de vote (Kazman, Bass, Abowd, & Webb, 1994)…etc.).

- Etudes se basant sur la gestion des connaissances architecturales : Ayant comme caractéristique principale l'étude de la réutilisabilité des connaissances acquises pour perfectionner l'opération d'évolution (Babar & Gorton, 2007; Giesecke, Hasselbring, & Riebisch, 2007; Jansen, Van Der Ven, Avgeriou, & Hammer, 2007).

Toutes ces études montrent l'importance de cette propriété avec l'obligation de bien administrer la propagation d'évolution pour s'assurer de la cohérence du système suite aux modifications opérées dans une architecture. A titre d'exemple ne pas gérer l'impact de

suppression d'un composant par exemple sans intervenir aux niveaux des différents éléments qui lui sont directement attachés entrainera une incohérence du système.

4.3.1.3 Traçabilité de l'évolution

Chaque évolution doit au préalable définir l'intention si l'évolution envisagée doit être enregistrée pour garder la trace des opérations effectuées. Cette traçabilité est un critère favorable pour élaborer un éventuel retour en arrière. En concevant les multiples traces au cours du cycle de vie du système, le raisonnement sur l'évolution peut cumuler les connaissances et/ou scénarios d'évolutions possibles en réaction à un besoin d'évolution. Cette intention de traçabilité assure la continuité du système et nous parlerons donc d'évolution continue (*seamless*) contrairement à une évolution avec rupture (*break*) ou les étapes qui ont menées à l'architecture évoluée ne sont pas conservées (Tamzalit & Oussalah, 2003). Les traces peuvent être évidemment de type atomique par référence aux simples opérations d'ajout d'un composant/service ou complexe c.à.d. incluant une combinaison de plusieurs opérations atomiques. La traçabilité est souvent schématisée à l'aide d'un graphe orienté spécifiant l'ensemble des configurations d'architectures (représentées en nœuds) reliées par les opérations d'évolution qui ont permis une telle transition (représentée en arcs unidirectionnels). Entre deux nœuds, on peut décrire la séquence d'opération qui a permis l'évolution d'une architecture ou planifier une évolution en considérant un chemin bien spécifique. Il est à noter que le graphe doit être épuré de toute forme de confusion par référence aux :

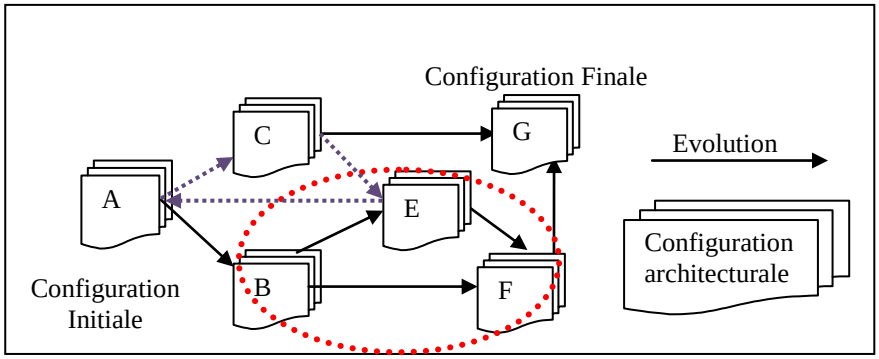

Figure 4. 10- Graphe d'évolution par transition d'une architecture logicielle.

- Chemins en cercles, i.e. les chemins où coïncide la configuration finale avec la configuration initiale, ne sont pas considérés puisqu'ils présentent une situation inerte de l'architecture (chemins en pointillé dans la Figure 4. 10).
- Chemins multiples existant entre deux nœuds (en rouge dans la Figure 4. 10).

Notons aussi que le chemin inverse entre deux nœuds donne la séquence qu'il faut suivre pour faire un retour de l'architecture à son état initial. De toute évidence, les approches de gestion des versions respectent parfaitement cette capacité. Cependant, pour développer une mesure directe de la capacité d'énonciation, nous utilisons le même raisonnement que dans (Pressman, 2005) pour calculer un facteur de qualité. La capacité d'énonciation ($Q_n(p)$)) peut donc être donnée sous la forme d'une équation paramétrique dans laquelle p présente le nombre de métriques pondérées pour chaque critère et a, b et c sont des coefficients par lesquels nous

pouvons établir un ordre hiérarchique de préférence entre les critères. L'expression de la capacité d'énonciation est une équation paramétrée de différentes intentions:

$$Q_n(p) = \frac{a \times F(p) + b \times I(p) + c \times T(p)}{a + b + c} \quad (1)$$

Avec *p*: le degré de paramétrage de chaque critère, *a*, *b* et *c* sont les poids associés à chaque critère respectivement formulation (F), Impact (I) et traçabilité (T).

4.3.2 Capacité d'expressivité

La deuxième capacité cerne des intentions relatives à l'expressivité de l'approche d'évolution. Nous entendons par expressivité de modèle l'expression des niveaux d'abstraction/modélisation, mode, domaine et la mécanique opératoire employée pour exprimer l'approche. L'expressivité désigne la capacité de ce que ce modèle est effectivement en mesure de décrire (Maisonnasse, Berrut, & Chevallet, 2009) quant à l'évolution de l'architecture de départ en architecture d'arrivée. Cette capacité est en fonction de cinq critères d'expressivité pertinents. Dans ce qui suit, nous définissons séparément les différents critères intervenant dans le calcul de cette capacité.

4.3.2.1 Abstraction

Comme il a été déjà défini dans la sous-section (4.2.2.1) précédente que nous définissons le niveau d'abstraction comme le degré de raffinement supérieure ou inférieur adopté pour évoluer l'architecture. La relégation des propriétés sémantiques associées à un degré dénote le passage à un raffinement au niveau supérieur ce qui mène à considérer que la spécification de l'architecture se situera au niveau le plus haut d'abstraction tandis que la représentation en code précisera le niveau le plus inférieur qui est caractérisé par la transparence et visibilité du code. Cette abstraction est exprimée en plusieurs représentations :

- Représentation en boites noires : Elle offre la faculté de réutiliser l'évolution sans avoir à se préoccuper de la composition ou le fonctionnement interne de l'évolution c.-à-d. que les composants en boites noires encapsulent la logique métier et le code construit pour les protéger de toutes modifications extérieures. Ce mode d'abstraction est le plus favorable pour le domaine de l'orienté service (Pei Breivold, 2009) du fait que les utilisateurs composent des services qu'ils ne peuvent pas modifier. Par exemple, les composants logiciels sont souvent livrés dans des boîtes-noires. Alors, nous appelons les composants, dont les codes sources ne sont pas disponibles pour les utilisateurs, des «composants en boîte noire». Le service est considéré comme une entité où est encapsulé la logique métier qui ne peut être modifiée. Son exploitation est garantit par l'intermédiaire des interfaces et des protocoles de communications bien définies via les fichiers de description de service. Cela entraine forcement une rigidité de manipulation ou d'utilisation par les utilisateurs de services (Crnkovic, Chaudron, & Larsson, 2006).

Par exemple dans (B. Zhang, Bao, Zhou, Hu, & Chen, 2008), les auteurs proposent des composants de type "boîte noire" pour exporter des fonctionnalités interactives personnalisées de systèmes existants vers des services Web à l'aide d'une méthodologie d'encapsulation adaptée aux systèmes existants basés sur une interface graphique. La stratégie propose également une solution

de déploiement de services Web consistant en une interopérabilité entre les systèmes utilisateurs et les anciens systèmes dans une architecture orientée services.

- Représentation en boites grises ou boite en verres : Ici, la transparence n'est pas totale i.e. que les détails nécessaires pour la compréhension de l'évolution sont visibles à l'utilisateur mais en contre partie il ne peut pas intervenir pour modifier ou personnaliser l'évolution en question. Autrement dit, cette représentation permet de voir à la fois l'intérieur et l'extérieur de la boîte, mais comme à travers un verre transparent i.e. l'intérieur n'admet aucune intervention. D'une vue conceptuelle, les composants, en particulier les composants hiérarchiques, peuvent être vus comme des entités de type "boîte en verre" ou « boite grise » avec la structure interne visible comme un ensemble de sous-composants communicants (Hnětynka & Plášil, 2006). Le gain ici est que le ré-utilisateur du système peut beaucoup mieux comprendre la boîte ainsi que son utilisation (Adekola, Idowu, & Adebayo, 2016).

SOFA 2.0 (Bures, Hnetynka, & Plasil, 2006) est un système typiquement académique basé sur des composants. Il utilise un modèle de composant hiérarchique avec des composants primitifs ou composites. Un composant composite est constitué d'autres composants, tandis qu'un composant primitif ne contient aucun sous-composant. Un composant est décrit par son cadre et son architecture. Le cadre est une vue en boîte noire d'un composant où est défini les interfaces fournies et requises du composant. L'architecture est une boite grise d'un composant dans laquelle est implémenté le contexte du composant en spécifiant les composants successifs et leurs interconnexions au premier niveau d'encapsulation. Les composants sont interconnectés via les liaisons entre les interfaces.

- Représentation en boites blanches : Elle désigne toutes les représentations flexibles où l'utilisateur peut intervenir sur le code ou l'activité du composant. On dit alors que cette évolution est transparente puisque tous les détails internes de l'évolution sont accessibles (aussi bien pour la compréhension ou la modification). Cette représentation est très sollicitée puisqu'elle offre l'avantage de personnaliser l'évolution selon les besoins des utilisateurs qui sont souvent concurrents. Les boites blanches permettent de modifier l'intérieur de la boîte et son interface. La structure interne d'une boîte blanche peut être partagée avec une autre boîte par le biais de techniques bien connues telles que l'héritage. Il faut souligner l'utilité d'effectuer des nouveaux tests lorsque la modification est effectuée (Adekola, et al., 2016).

Exemple, *Le problème de la migration de l'interface d'un système existant vers de nouvelles plates-formes, la plupart d'entre elles sont des approches de type boîte blanche, basées sur des techniques de reverse engineering visant à décomposer le système existant et à extraire ses composants d'interface et ses composants d'application via une analyse de code* (Gaeremynck, Bergman, & Lau, 2003; Rugaber, 1999). Un autre exemple évoqué dans (Garcés et al., 2018), *les auteurs présentent une approche de transformation par boîte blanche qui modifie l'architecture de l'application sans perdre de valeur, ni d'attributs de qualité*.

4.3.2.2 Modélisation

Reflète les niveaux de modélisation affectés par le modèle pour aboutir à l'architecture évoluée. Il désigne les niveaux de modélisation de l'architecture de départ et celui de l'architecture d'arrivée vus dans la sous-section 4.2. Il est important de savoir que l'architecture peut être opérée sur les trois niveaux de modélisations précédemment décrits : Le niveau de méta-modélisation (M_2), le niveau de modélisation (M_1) et le niveau application (M_0). L'architecture se

réfère à ces trois niveaux de modélisations dans la même logique de conception en A_2, A_1 et A_0. Le niveau modèle de l'architecture (A_1) représente une description d'une instance de la famille des concepts du niveau méta (A_2). Quant au niveau application A_0 désigne l'instanciation du système représentée par un modèle. Comme l'évolution d'un système est un processus inéluctable, le besoin d'évolution peut se manifesté aux différents niveaux du cycle de vie du logiciel de la spécification à l'exécution. Ainsi et selon (Amirat, Menasria, & Gasmallah, 2011), nous pouvons identifier trois niveaux de modélisation de l'évolution.

- Niveau d'évolution E_2 : Les méta-modèles fournissent la syntaxe abstraite d'un langage de modélisation, ce qui signifie que tous les concepts de modélisation disponibles sont définis, ainsi que les méthodes autorisées pour les combiner afin de créer des modèles. Un méta modèle spécifie le langage de modélisation que les modèles utilisent pour les définir et les exprimer. Ce méta modèle est défini ou écrit dans un langage de méta modelage et est similaire à un langage de modélisation qui peut également être spécifié par un méta-méta modèle. (C'est-à-dire que ce méta modèle est défini par un langage de méta-modelage, qui peut également être spécifié par un méta-méta modèle). Exemple- MES (Méta-Evolution Style) (Hassan & Oussalah, 2016) vise à développer une bibliothèque de styles d'évolution à partir des solutions unifiées et standardisées des concepts de modélisation en exploitant les meilleures techniques existantes. Pour cette fin, ils proposent le style MES pour l'évolution d'architecture logicielle, qui favorise la cartographie et la comparaison des styles d'évolution. De plus MES fourni une méthode pour assister les architectes à la réutilisation et à l'échange d'éléments entre les styles d'évolution.

- Le niveau de modélisation E_1 : Les types de composants ou services sont instanciés au niveau de l'architecture pour qu'ils soient utilisés. On parlera ici du sous typage, du fait que chaque type du niveau E_2 est développé en termes d'éléments plus spécifiques pour décrire des fonctionnalités bien précises du type E_1. Par exemple, en CBSE on peut évoluer un composant de différentes techniques d'évolution comme le *refactoring* ou la *restructuration.*

- Le niveau de modélisation E_0 : Ce niveau d'évolution est plus explicite puisque il représente l'ensemble des opérations nécessaires au système pour assurer ses fonctionnalités et son utilisabilité face aux besoins d'évolution émergents. Cette évolution est contrainte à plusieurs facteurs qui rendent cette opération plus délicate telle que l'environnement dynamique, et l'exécution en temps réel.

4.3.2.3 Domaine

Il représente le champ d'application que couvre le modèle d'évolution qui peut être (i) générique à plusieurs domaines d'application ou (ii) spécifique à un domaine quelconque et le modèle d'évolution ne peut pas être réutilisable qu'à l'intérieur du même domaine ou (iii) encore orienté application i.e. dont l'évolution est spécifique qu'à une seule application et donc peu réutilisable.

L'évolution en domaine reflète aussi les différentes couches du système où l'évolution est opérée. Il est important alors de distinguer la structure des couches pour pouvoir identifier les évolutions possibles à l'intérieur de la même couche au entre couches voisines. Nous entendons ici par couche, le niveau de dépendance entre le domaine d'application et l'évolution à effectuer. L'indépendance est plus forte aux niveaux inférieurs et elle est plus faible en direction des niveaux les plus élevés. Ainsi, le domaine est dit connexe au niveau le plus haut et nous y trouvons les évolutions l'ensemble des opérations de modification afférentes aux domaines

connexes à l'application. A ce niveau, nous pouvons dire que plus que l'évolution est spécifique à un domaine plus qu'elle devient une contrainte qui entrave la réutilisabilité de l'évolution. Delà découle :

- Evolutions génériques : Ce sont des évolutions où la portée des évolutions est importante. Ici, l'évolution est dite indépendante et peut donc être réutilisée indépendamment du domaine d'application. C'est ce type d'évolution que cherche souvent les architectes pour anticiper les modifications à un niveau bien avancé de la conception voire l'architecture du logiciel.

- Evolutions dépendantes : Ce sont des évolutions de portée moindre, mais qui sont toujours réutilisables sous la condition de préserver toujours le même domaine d'application. Elle est aussi importante pour les architectes puisque cela leurs permet de dresser des catégories ou des familles d'architectures susceptibles d'agir de la même manière d'évolution.

- Evolutions spécifiques applications : Elles sont moins appréciées par les architectes car leurs portées sont minimales c.-à-d. que les opérations d'évolutions sont bien spécifiques à ces applications. Nous parlons de rigidité d'évolution pour ces types car elles sont peu réutilisables.

4.3.2.4 Mode

Il traduit les différentes représentations du modèle d'évolution et qui peuvent être: graphique, représentation textuelle, logique, code. Elles doivent être simples, compréhensives reflétant une sémantique claire. L'expressivité est proportionnellement liée à la granularité du modèle d'évolution présenté. Plus la granularité est fine plus l'expressivité est bonne. Il est important de préciser que la caractéristique de granularité est fortement liée à la prise en charge des opérations d'évolution atomiques (primitives) ou complexes (composites) par le langage de description. Le mode de représentation définit l'utilisation possible d'une évolution comme l'évolution pour l'analyse, la compréhension et les techniques liées à la représentation des plateformes.

4.3.2.5 Mécanique opératoire

Elle prescrit le mode de raisonnement associé au modèle pour manager l'évolution. Le raisonnement dépend en effet de l'ensemble des moyens d'application possibles pour procéder à l'évolution comme la factorisation, la spécification, l'instanciation et la composition des évolutions.

Il existe trois types de mécanique opératoire :

- la déduction qui consiste à considérer les propositions actuelles, avec un enchaînement logique, pour aboutir à une solution,
- l'induction qui, à partir de la comparaison des cas quasi-similaires au problème, tente de conclure sur une proposition générale dont la solution fera partie, et
- la classification qui est orientée sur des solutions réparties en groupes basés sur des caractéristiques communes des solutions (norme ISO/CEI 11179-2:2005).

Parmi les propriétés intéressantes du mécanisme opérateur :

- elles peuvent être centralisées ou distribuées entre les éléments évolutifs d'une architecture,
- elles peuvent être câblées c.-à-d. offertes par l'ADL lui-même,

- elles peuvent répondre à des besoins spécifiques et par conséquent, elles sont décrites par les architectes eux-mêmes.

Conformément au raisonnement donné ci-dessus pour la capacité d'énonciation, la capacité d'expressivité peut être exprimée par :

$$Q_x = \frac{d \times M(p) + e \times A(p) + f \times E(p) + g \times O(p) + h \times D(p)}{d + e + f + g + h} \quad (2)$$

Avec p: le degré de paramétrage de chaque critère, *d*, *e*, *f*, *g* et *h* sont les poids associés à chaque critère respectivement : niveau de modélisation (M), niveau d'abstraction (A), le mode d'expressivité (E), la mécanique opératoire (O) et le domaine (D).

4.3.3 Capacité d'évaluation de la qualité

La troisième dimension traduit l'aptitude de mesurer les capacités du modèle à évaluer sa qualité à travers l'expression des caractéristiques d'évaluation qu'un modèle d'évolution satisfait.

4.3.3.1 Réutilisabilité

La réutilisation forme en réalité la motivation principale derrière toute opération d'évolution. La réutilisabilité offre l'avantage de la réduction de temps d'implémentation. La réutilisation d'un ensemble d'opérations d'évolution constitue l'avantage de pouvoir les intégrer dans un seul opérateur composé et qui sera invoqué au moment aux des exigences qualifiées pareilles se présentent dans le système. Il est aussi important de préciser que la réutilisabilité permet de localiser les changements à effectuer sur le code quand un besoin d'implémentation est requis. L'optimisation de la réutilisation des composants ou services facilitent les opérations de sélection et de composition à travers la connaissance du domaine d'application et la disponibilité des services ou composants standards. Ce qui offre en aval, une fluidité de modification voire même la construction de nouvelles architectures (Tahir, Khan, Babar, Arif, & Khan, 2016).

La réutilisabilité offre l'avantage aussi bien en coût qu'en temps qui sont généralement relatifs au marché approprié (Wirfs-Brock & Wilkerson, 1989), (Kaur, Harrison, & West, 2015). Selon l'analyse de la littérature, il a été démontré que de nombreux chercheurs se concentrent sur la mesure de la réutilisabilité (Tahir, et al., 2016). En conséquence, Poulin (Poulin & Caruso, 1993) a proposé un modèle commun de réutilisabilité où il a décrit les résultats de l'enquête et met en évidence certains des facteurs importants influant la réutilisabilité. En suite, (Frakes & Terry, 1996) ont introduit autres éléments clés supplémentaires pour la thématique. En plus, (Ampatzoglou, Kritikos, Kakarontzas, & Stamelos, 2011) révèlent que l'utilisation des patrons de conception augmente la capacité de réutilisation et soutient donc l'évolution des systèmes logiciels. Par exemple parmi les composants réutilisables, nous citons le masquage des données, complexité, couplage, généricité (Padhy, Panigrahi, & Baboo, 2015).

Dans notre Framework de qualité, nous désignons par la réutilisabilité le degré de réutilisation qu'offre un modèle d'évolution. Le degré de réutilisation désigne le niveau de la réutilisation dans l'architecture du système. La réutilisation peut être appliquée au niveau code ici, elle est faible ou aux niveaux les plus avancés ce qui rend le sa réalisation plus profonde et donc fortement couplée à l'architecture du système.

4.3.3.2 Support d'évolution

Les supports d'évolution désignent les outils nécessaires proposés par la méthode d'évolution pour exécuter une ou plusieurs opérations d'évolutions atomiques ou composites. Cependant, plusieurs patrons de modélisations sont proposés dans la littérature et qui représentent un moyen pour traiter la problématique de la modularité et l'évolution du code dans le domaine du génie logiciel (Gamma, Helm, Johnson, & Vlissides, 1994). Ces patrons préservent la trace des décisions de conception au niveau code pour une bonne application des changements avec la faculté de préserver une bonne structuration de code. Cependant, l'application de ces patrons montre à travers plusieurs études qu'elle limite souvent l'évolution et la maintenance si les patrons sont mal appliqués et mis en œuvre (Khomh, Di Penta, & Gueheneuc, 2009).

Nous nous limiterons dans notre étude de compréhension et analyse du modèle d'évolution à l'existence d'un support d'évolution pour la méthode concernée ou de son utilisation dans le milieu académique ou industriel.

4.3.3.3 Adaptabilité

En tout premier lieu, nous définissons l'adaptation ou l'adaptabilité comme elles ont été présentées dans plusieurs travaux représentatifs. Quoique ces définitions semblent dresser la même problématique mais, elles présentent quelques différences qui peuvent même entrainer certaines confusions. Selon (Tekinerdogan & Aksit, 1996), l'adaptabilité est définie comme la facilité avec laquelle un système ou une partie du système peut s'adapter avec les exigences de changements. La définition de (Lieberherr, 1995) stipule qu'un programme est dit adaptable s'il peut être facilement changé c.-à-d. que son comportement peut facilement changé en fonction des changement de son contexte. Cependant, (Dorfman & Thayer, 1990) qualifie l'adaptabilité comme une métrique l'évaluation de la facilitation avec laquelle le système permet l'intégration et la satisfaction des différents contraintes et besoins d'utilisateurs. Alors que (Sanders, 1994) la définit comme une caractéristique permettant au système de s'adapter à des environnement spécifiques sans avoir à être amener à appliquer d'autres actions autres que ceux fournis par le produit logiciel lui-même. Par contre, (Oreizy, Medvidovic, & Taylor, 1998) définissent l'évolution adaptative comme la modification du logiciel pour lui permettre d'exécuter dans un nouveau environnement. Cependant, de nombreuses techniques ont été développées pour traiter l'adaptation des systèmes logiciels et chacune d'elles a son propre contexte d'application. Cependant, très peu de ces techniques suivent les solutions développées selon leurs besoins (Subramanian & Chung, 2001).

Par exemple la SOA doit permettre l'adaptation de l'architecture aux besoins évolutifs et variant d'une catégorie d'exigences. Les architectures basées SOA sont appuyées par la diversité des interfaces entre services ce qui augmente leurs agilités et leurs réactivités face aux besoins multiples, ce qui exige une forte capacité d'adaptation et d'agilité pour les systèmes (Valipour, AmirZafari, Maleki, & Daneshpour, 2009).

Cependant, dans un premier lieu de notre étude, nous nous focaliserons sur la présentation de l'adaptabilité comme une contrainte sur laquelle une méthode d'évolution demeure fonctionnelle après variation de l'environnement d'évolution du système.

4.3.3.4 Performance

L'opération de mesure de la performance est une des préoccupations la plus importante pour les systèmes. La performance touche aux différentes phases du cycle et devient plus importante après la mise en œuvre du système puisque c'est une mesure d'un avantage concurrentiel de l'organisation. Il est a noté que le caractère dynamique d'une application peut entraver la performance en causant des surcharges dues à la prise en charge des liens entre les procédures de modifications et leurs segments d'exécution. En conséquence, le besoin d'évolution d'un système évoluant dans un environnement dynamique est une préoccupation extrêmement difficile à anticiper. Notons que la flexibilité de certains langages dynamiques comme Smalltalk ou Lisp dépend directement de la performance liée aux applications implémentées à l'aide de ces langages. Les mutations sont directement effectuées au niveau des instructions et procédures sans avoir à interrompre le système. En CBSE, le remplacement d'un composant d'une architecture est simple quand il se fait en mode fermé (statique). En environnement dynamique, le processus devient plus délicat puisque il doit assurer la performance du système (Oreizy, et al., 1998). Ce mode de replacement est favorable lorsque le système est conçu pour détecter la perte d'état du composant pour permettre de basculer vers un mode de fonctionnement dégradé tout le long du processus de récupération. Il est aussi possible d'incorporer un modèle opérationnel pendant l'exécution par l'utilisation de style d'architecture (Sha, Rajkumar, & Gagliardi, 1996). Le système rejette carrément les composants évolués s'ils ne répondent pas aux critères de performances fixés aux préalables. Dans la même vision, les connecteurs doivent inclure des mécanismes pour permettre l'ajout, la modification des liens les composants pour soutenir le processus de la reconfiguration. Par conséquent, ces mécanismes peuvent avoir des conséquences néfastes sur les appels de procédures du connecteur puisqu'elles intègrent des fonctionnalités supplémentaires (Reiss, 1990).

La question de la performance prend toujours la grande part des préoccupations quand les problématiques d'évaluation ou de qualité sont initiées. Plus de 50% des entreprises ont changé leurs système de mesure de performance (Ling Sim & Chye Koh, 2001) rien parce qu'elles changent à chaque fois leurs perception aux : quoi ? et comment ? mesurer. La question de compréhension de ce critère reste évolutive et non clair pour l'ensemble (Kennerley & Neely, 2002). Cependant, la performance se greffe à tous les niveaux d'un système. Au niveau architectural, la performance touche aux éléments architecturaux. Il est important de savoir que l'évolution vise à garder ou à améliorer la performance du système initial. Les mesures de performance deviennent de plus en plus délicates dans les évolutions dynamiques car il faut garder trace sur les performances après la mise en service de l'architecture finale. Cela impose que ces critères doivent être réactifs aux circonstances dans lequel le système est entrainé. Ces mesures représentent un facteur déterminent pour la pertinence de la méthode d'évolution. En effet, pour parler systématiquement de la performance, les forces déterminant l'évolution des systèmes ont été répartis en deux (Waggoner, Neely, & Kennerley, 1999). Les auteurs en (Kennerley & Neely, 2002) ont présenté un référentiel modélisant les facteurs qui impactent l'évolution des mesures de performances des systèmes. Le Framework se concentre au noyau sur un ensemble de mesures de performance qui peuvent être acquises, collectées, analysées, interprétées et partagées. Ce noyau est affecté par un ensemble d'événements déclencheurs (Triggers) qui peuvent être de natures externes ou internes, comme il est spécifié dans la Figure 4.11.

- Des forces extérieures : Ce sont les mesures de performance de nature externe au produit logiciel tels que : la législation, besoin des clients, position du marché, des nouvelles orientations industrielles, stratégies …etc. Ces forces sont dues à: l'utilisation, la réflexion, la modification et le déploiement

- Des forces internes : Elles cernent l'ensemble des performances qui sont liées au produit logiciel lui-même, nous citons à titre d'exemple : les capacités interne du produit, incorporation de nouvelles technologies, réorganisation, le processus, les composants des systèmes.

Dans notre référentiel qualitatif, nous nous intéresserons simplement à la vitesse d'exécution de l'évolution d'un modèle. Cette vitesse devient plus dégradée lorsque le système est assez réactif aux changements de l'environnement dynamique où il se trouve.

Il est à noter que pour augmenter la performance, le degré de paramétrage peut inclure d'autres critères à savoir :

Extensibilité : L'extensibilité est la capacité d'un système à étendre sa capacité d'adaptation lorsque des événements imprévus défient ses frontières (D. Woods, 2015). Tous les systèmes ont une enveloppe de performance, ou un ensemble de comportements adaptatifs, en raison de ressources limitées et du changement inhérent de l'environnement. Ainsi, il existe une zone de transition dans laquelle les systèmes changent les régimes de performances lorsque des événements poussent le système au bord de son enveloppe. La limite fait référence à la zone de transition où les systèmes changent les régimes de performance. Cette zone peut être plus nette ou plus floue, plus stable ou dynamique, partiellement bien modélisée ou mal comprise. L'extensibilité fait référence à la capacité du système à adapter son fonctionnement pour étendre les performances au-delà de la zone limite dans un nouveau régime de performances, appelant de nouvelles ressources, réponses, relations et priorités (D. D. Woods, 2018),(Brown, Knepley, & Smith, 2015). Par exemple l'évolution d'un élément par ajout par exemple ne chevauche pas l'évolution des autres éléments,

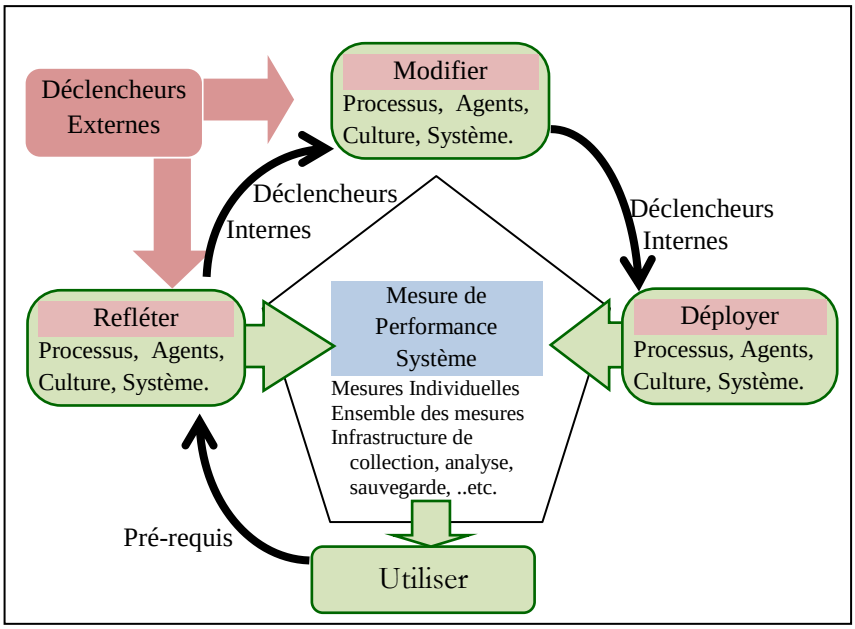

Figure 4.11- Framework de mesure de la performance d'évolution.

[Source : inspiré du travail de (Kennerley & Neely, 2002)]

Composabilité : La composabilité est la capacité de combiner et de recombiner des blocs de construction de code tels quels dans des différents systèmes à des fins différentes. La composabilité nécessite que : (i) les blocs de construction de code rendent leurs propriétés explicitement disponibles et que (ii) le système garantit que ces propriétés restent invariantes dans leur composition (Lutz et al., 2019). De facto, la composabilité est le concept clé pour répondre aux inconvénients de l'intégration de blocs de construction préexistants puisqu'elle nécessite généralement de nombreuses adaptations pour répondre aux besoins de l'utilisateur et du système. Ces adaptations sont non seulement coûteuses, mais entravent également la réutilisabilité et la maintenabilité du système. Par exemple pour composer une nouvelle évolution, l'évolution peut déléguer ses fonctionnalités à d'autres évolutions qui lui sont internes.

Remplaçabilité- La remplaçabilité fournit l'avantage de flexibilité des architectures logicielles. Cela est assuré au moyen de la substitution d'un ou plusieurs composants ou services pour parvenir à répondre à des besoins plus élaborés. Cette qualité est bien présente dans les systèmes à base de composants puisque les composants ou les connecteurs peuvent être remplacés sous conditions qu'ils préservent les fonctionnalités existantes. Le remplacement est donc une caractéristique qui supporte l'extensibilité du logiciel et représente aussi une conséquence de l'opération de mise à jour puisque souvent les architectes mettent à jours des services en utilisant exclusivement des opérations de remplacements (Plasil, Balek, & Janecek, 1998). Par exemple, un service T peut remplacer un service S si tous les contextes admissibles pour le service S sont également admissibles pour T. Les contextes admissibles sont définis en tant que contrôleurs; le service R est appelé contrôleur du service S si R et S peuvent interagir sans blocage. Cette notion de remplacement est formalisée par la conformité pre-order (Stahl, Massuthe, & Bretschneider, 2009) i.e. on dit que T est en accord avec S si chaque contrôleur de S est un contrôleur de T. La construction de services remplaçables par exemple, est une activité essentielle non seulement pour que de nombreuses organisations restent compétitives, mais elle est également connue pour être sujette aux erreurs et prendre beaucoup de temps. Cependant dans notre contexte, deux évolutions offrant les mêmes fonctionnalités ou des fonctionnalités incluses doivent offrir l'opportunité d'être substituées.

Enfin et de la même façon, nous exprimons la capacité d'évaluation par une expression paramétrée dépendant du degré de paramétrage de chaque critère *p* tel que :

$$Q_v(p) = \frac{i \times R(p) + j \times A(p) + k \times P(p) + l \times S(p)}{i + j + k + l} \quad (3)$$

Avec *p*: le degré de paramétrage de chaque critère, *i*, *j*, *k* et *l* sont les poids associés à chaque critère respectivement réutilisabilité (R), l'adaptabilité (A), la performance (O) et le support d'évaluation (S).

4.3.4 Expression des attentes de la qualité.

Comme mentionné ci-dessus, en appliquant le raisonnement McCall (McCall, 1994) concernant l'évaluation des facteurs de qualité, les attentes en matière de qualité peuvent être exprimées comme le produit des trois capacités Q_n, Q_x et Q_v, respectivement, pour l'énonciation, l'expressivité et les capacités d'évaluation, comme suit:

$$Q = \frac{\alpha \times Q_n(p) + \beta \times Q_x(p) + \gamma \times Q_v(p)}{\alpha + \beta + \gamma} \quad (4)$$

Enfin, il est important de noter que le pourcentage de l'attribut de chaque sous-caractéristique reflète l'intérêt que présente l'approche pour chacune d'elles. Par conséquent, nous devons considérer les attentes en matière de qualité comme une unité (100%), ce qui implique l'expression des divers attributs dans un tout ($N = \sum_1^{12} a_i$).

L'aspect qualitatif du méta-modèle respecte la fondation ISO/IEC 9126 où la qualité est considérée comme un ensemble de caractéristiques. Chaque caractéristique est une agrégation de sous-caractéristiques (critères), qui comprendra à son tour une collection d'attributs. Nous étendons cette perception au fait que la qualité est une généralisation des capacités d'énonciation, d'expression et d'évaluation. La Figure 4.12 met en évidence les liens conceptuels entre les éléments évolutifs et les caractéristiques associées.

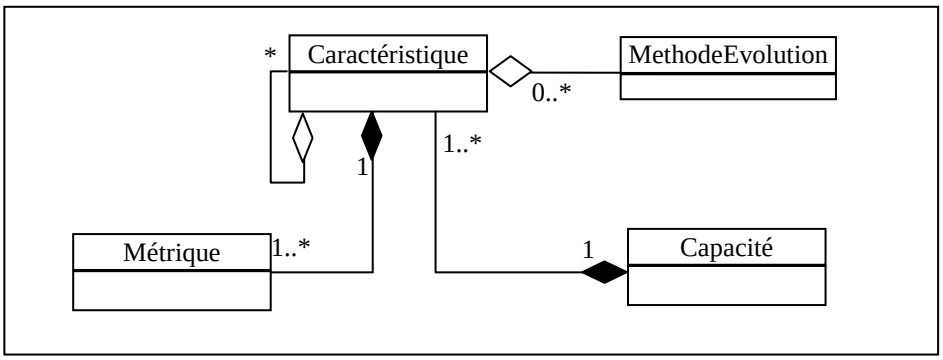

Figure 4.12- Modèle qualitatif d'une évolution architecturale.

4.4 Taxonomie d'évolution basée référentiel

Une taxonomie fournit un vocabulaire commun pour la comparaison et la distinction des différentes études de recherche. De plus, elle permet de fournir une classification basée sur des dimensions sélectionnées en fonction d'un besoin bien spécifique. La classification proposée comprend six classes basées sur le raisonnement de la mécanique opératoire d'évolution (MOE) (Gasmallah, Oussalah, & Amirat, 2016) plutôt que d'aborder la classification par thématique comme dans (Breivold, et al., 2012), (Aleti, et al., 2013). Où, la première présente une revue systématique de l'architecture pour l'évolutivité des logiciels. L'objectif était d'établir un aperçu exhaustif sur les approches existantes d'analyse et d'amélioration de la capacité d'évolution des logiciels académiques et industriels. Cette évolution concerne aussi bien le niveau architectural que le niveau d'analyse des impacts d'évolution. L'identification des principales études était élaborée à travers un processus de sélection en plusieurs étapes. Comme conclusion, les auteurs ont révélé des suggestions pour des recherches et des pratiques plus avancées dans le domaine, telles que la nécessité d'élaborer une base théorique pour la recherche sur l'évolution des logiciels, i.e. construire une connaissance généralisée, car l'expertise dans ce domaine est toujours construite sur la base d'études de cas. Notre recherche s'insère dans cette perspective.

La seconde classification s'est focalisée sur le domaine industriel et précisément les méthodes d'optimisation des architectures logicielles sur la base d'un ou plusieurs attributs de qualité. Cependant, les résultats rapportés sont fragmentés sur différentes communautés de recherche, plusieurs domaines et plusieurs attributs de qualité. L'étude s'est déroulée à travers une revue systématique de 188 études incluant les différentes communautés de recherche dans le domaine. Les auteurs ont présenté une taxonomie des méthodes d'optimisation basées attributs de qualité dont l'intérêt est de fournir une aide aux chercheurs pour consolider les efforts de recherches et ouvrir d'autres perspectives de recherches.

Cependant, il serait préférable pour une classe de taxonomie de représenter des approches partageant une méthode de raisonnement commune selon un point de vue architectural. Ici, une classe taxonomique présente une méthode courante de classification basée sur un point de vue architectural de première classe et se concentre donc sur la dimension du mécanisme d'évolution (Comment ?) à travers la dimension de niveau hiérarchique d'architecture (Où ?). Pour chaque métrique de la dimension MOE, l'architecture développée passe par l'un des trois cas suivants: i) modification des propriétés de certains éléments architecturaux et/ou fonctionnalités, ce cas est appelé *niveau intra-abstraction*, ii) déduction de plusieurs concepts au sein du même niveau de modélisation, appelé *niveau inter abstraction* et iii) divers concepts étant traités à différents niveaux de modélisation, ce cas est appelé *niveau inter modélisation* (Gasmallah, Oussalah, & Amirat, 2016).

Le choix des deux dimensions pour la classification a développé six classes différentes pour classer les approches conformément aux hypothèses ci-dessus, comme il est indiqué dans la Figure 4.13. Étant donné que toutes les dimensions sont importantes et jouent un rôle essentiel dans la classification, le fait de prendre en compte d'autres dimensions augmenterait considérablement le nombre de classes et aboutira sans aucun doute à une classification plus fine ou plus profonde. En outre en raison de limitation de nombre de paramètres intervenants dans la taxonomie et par souci de consolidation de cette thèse, seules six classes sont utilisées pour présenter le référentiel (Gasmallah, Amirat, Oussalah, & Seridi, 2019). Ainsi, la taxonomie est orientée par deux orientations majeures : *réduction* et *émergence*.

4.4.1 Evolution orientée réduction

Cette évolution regroupe un ensemble de classes d'évolution qui se basent sur la même valeur de la dimension de la mécanique opératoire d'évolution (MOE) la *réduction* en premier lieu puis sur les niveaux hiérarchiques de l'architecture logicielle déjà définis dans la section précédente 4.2. Le résultat nous a permis d'identifier trois grandes classes d'évolution architecturale.

Classe 1- Évolution en réduction orientée niveau intra abstraction : Le processus d'évolution consiste à faire évoluer un ou plusieurs éléments d'une architecture initiale sans avoir d'impact sur son niveau de modélisation ou son niveau d'abstraction.

Par exemple, *la modification d'une ou plusieurs propriétés de composant d'une architecture.*

Classe 2- Évolution en réduction orientée niveau inter abstraction : L'incidence de l'évolution sur les éléments architecturaux est améliorée sur plusieurs niveaux d'abstraction tout en maintenant le même niveau de modélisation de l'architecture initiale.

Par exemple, *l'évolution de la classe impacte plusieurs sous-classes.*

Classe 3- Évolution en réduction orientée niveau de modélisation : Les éléments architecturaux sont améliorés sur un chemin de modélisation descendant à travers plusieurs niveaux de modélisation architecturale.

Par exemple, *l'évolution de la classe impacte plusieurs instances de classe.*

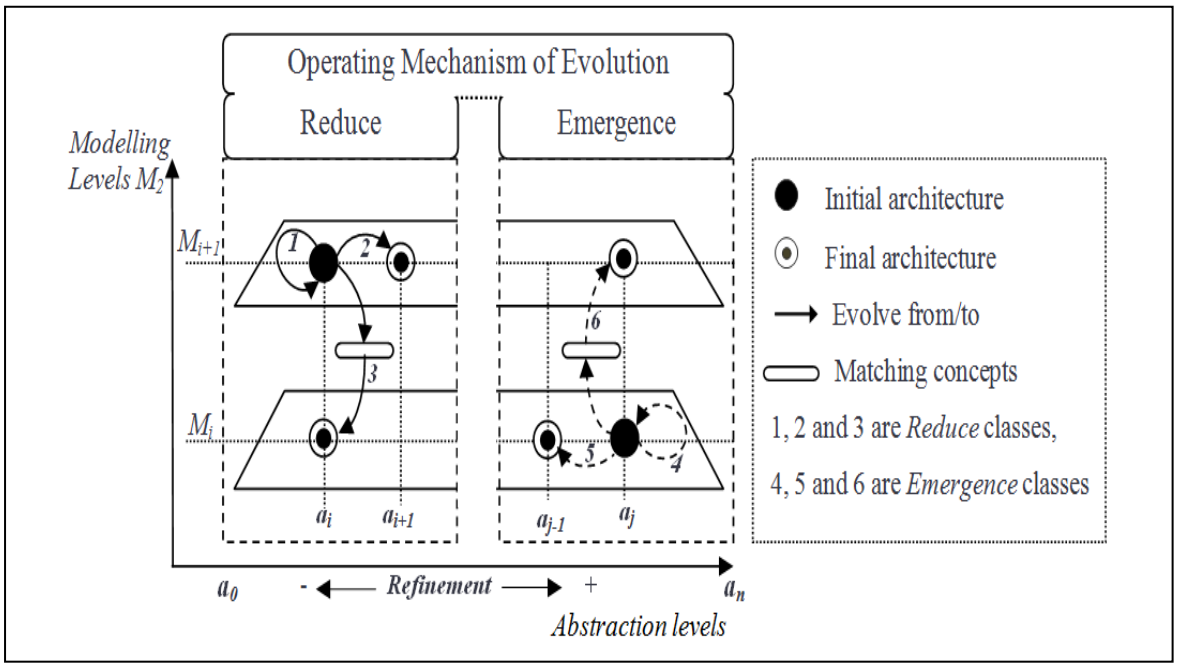

Figure 4.13- Présentation de la classification basée MOE et niveaux hiérarchiques.

4.4.2 Evolution orientée émergence

Sur le même principe de classification, nous avons considéré la valeur de l'MOE relative à l'émergence sur les différents niveaux hiérarchiques; trois classes différentes émergentes.

Classe 4- Evolution en émergence orientée niveau intra abstraction: L'architecture évoluée s'enrichit de nouveaux éléments architecturaux en appliquant des opérations qui n'ont pas d'impact sur les niveaux d'abstraction et de modélisation autres que ceux de l'architecture initiale.

Par exemple, *la catégorisation des classes par la création d'une superclasse*.

Classe 5- Évolution en émergence orientée niveau inter abstraction: En utilisant le même niveau de modélisation que l'architecture initiale, l'émergence de nouveaux éléments architecturaux se manifeste par l'implication de plusieurs niveaux d'abstractions pour des raisons de simplicité. Par exemple, *l'agrégation de classes dans une classe*.

Classe 6- Evolution en émergence orientée niveau de modélisation: Le processus d'évolution implique plusieurs niveaux de modélisation sur un chemin de modélisation ascendant afin de fournir à l'architecture finale de nouveaux éléments architecturaux.

Par exemple, *la création de nouvelles classes (au niveau M_1) pour traiter les différences sur plusieurs instances appartenant au niveau M_0 de modélisation.*

4.5 Conclusion

Dans ce chapitre, nous avons focalisé notre attention sur la description et la modélisation de l'évolution logicielle en considérant l'architecture du système comme l'entité principale pour l'évolution. Premièrement, nous avons mis l'accent sur le fondement de l'aspect structurel et comportemental de l'évolution d'une architecture. Les dimensions susceptibles à décrire le processus d'évolution d'une architecture ont été bien identifiées exhibant ainsi six dimensions orthogonales. Puis, nous avons achevé notre étude par un récapitulatif des différentes relations entre les dimensions pour montrer l'orthogonalité entre ces dimensions. Dans un second lieu, nous avons abordé l'aspect qualitatif attendu de l'évolution en se basant sur trois capacités de qualité cernant le référentiel qualitatif à savoir:

- énonciation justifiant l'intention et la motivation de l'approche en termes de trois critères,

- expressivité de l'approche adoptée portant sur cinq critères d'évolution et finalement

- la capacité évaluation qualitative reposant sur quatre critères au premier niveau.

A la base de tout ce qui est dit, et dans le chapitre suivant, nous sommes dans l'obligation de vérifier et de valider les référentiels proposés (structurel et qualitatif) par une investigation et

CHAPITRE 5

Etude descriptive et validation du référentiel

«Tout obstacle renforce la détermination. Celui qui s'est fixé un but n'en change pas.»
- Léonard De Vinci (1452-1519)

5.1 Introduction

Dans ce chapitre, nous allons explorer le Framework proposé pour monter son applicabilité et aussi de dresser une cartographie permettant de comprendre et de comparer les méthodes existantes d'architecture logicielle. Pour ce faire, nous avons défalqué le chapitre en trois grandes sections : la première pour montrer l'applicabilité de notre Framework proposé sur deux approches bien connues celle de Garlan et al. (Garlan, Barnes, Schmerl, & Celiku, 2009) et celle d'Oussalah et al. (Oussalah, Sadou, & Tamzalit, 2006) qui est une méthode issue d'une réflexion de recherche du laboratoire LS2N[1] ex Lina[2]. La deuxième section est réservée à une étude empirique comportant une analyse et une comparaison d'un ensemble de méthodes relevant du domaine de l'architecture logicielle orientée composant ou service. Alors que la troisième section sera dédiée à une étude orthogonale entre nos propositions et certaines autres classifications bien connues dans le domaine.

5.2 Evaluation et comparaison centrées Référentiel

Dans cette partie, et comme nous l'avons déjà mentionné, nous allons montrer l'application du référentiel et son utilisation pour la comparaison de deux différentes méthodes. L'intérêt de cette section est de montrer comment est appliqué le référentiel pour permettre à la fois la compréhension des méthodes et ainsi leurs comparaisons.

[1] Laboratoire des Sciences du Numérique de Nantes.
[2] Laboratoire d'Informatique de Nantes Atlantique.

5.2.1 Présentation de méthodes

Nous avons choisi deux méthodes d'évolution architecturale, l'une appartenant à une école anglo-saxonne et une autre méthode européenne et précisément française. Les deux méthodes sont sélectionnées par rapport à leurs importances et leurs impacts dans le domaine.

5.2.1.1 La méthode de (Garlan, et al., 2009)

Cette approche a été déjà évoquée au cours du chapitre 3, mais pour bien comprendre son fonctionnement nous la présentons ici d'une manière plus détaillée. Les auteurs proposent dans cette approche un style d'évolution qui fournit une vision complète sur la réutilisabilité dans le domaine de l'évolution architecturale. L'approche définit un style d'évolution pour la modélisation de l'évolution architecturale dont le but est de planifier le raisonnement dans ce domaine spécifique. L'idée principale des auteurs, concernant un style d'évolution, est de capturer et partager les meilleures pratiques et connaissances relavant d'un domaine spécifique de l'évolution d'architecture. A travers ce style d'évolution, les auteurs visent à modéliser les scénarios potentiels comme un ensemble fini de chemins d'évolution où chacun désigne une séquence d'états architecturaux. La transition entre deux états consécutifs s'effectue par l'application d'un opérateur d'évolution d'un état en entrée pour fournir un état résultat. Le style d'évolution définit les contraintes que chaque chemin doit respecter. L'approche introduit une assistance aux architectes pour sélectionner le chemin le plus optimal parmi un ensemble de chemins possibles en utilisant une fonction d'évaluation pour permettre l'évaluation des différents chemins.

Cependant, le style d'évolution définit un vocabulaire de concepts nécessaires pour la modélisation du processus d'évolution. Ce vocabulaire couvre les concepts afférents à :

- Un ensemble d'opérateurs d'évolution valables pour l'application des différentes transitions,
- Un ensemble d'état représentant les étapes ou phases que peut prendre une architecture à partir d'un état dit initial à un état dit final et dont la séquence reliant l'état initial, les états intermédiaires et l'état final forment plusieurs chemins qui ne sont que des alternatives de solutions,
- Un ensemble de contraintes de chemins d'évolution définissant les conditions à satisfaire pour que le chemin en question soit une alternative possible pour le style d'évolution et
- Un ensemble de fonctions d'évaluation pouvant être utilisées pour évaluer et mesurer en totalité chaque chemin d'évolution. Cela rend possible la comparaison entre plusieurs chemins et donc fournir un moyen aidant à sélectionner le chemin le plus optimal justifiant les besoins et les objectifs fixés par les intervenants.

Fondement théorique de la méthode

Cette méthode permet aux architectes de formaliser le problème d'évolution comme un graphe orienté où l'évolution est considérée comme une séquence conditionnée d'états architecturaux. Cette séquence débute avec l'état initial qui n'est autre que le style initial de l'architecture du système ou l'architecture courante et se termine par l'architecture cible ou finale. Tout le long du chemin, des états intermédiaires sont produits par application d'un opérateur d'évolution qui caractérise la transition d'un état à un autre pour atteindre l'architecture cible. La raison d'associer des contraintes aux chemins et de pouvoir les intégrer dans le style d'évolution.

Fondement technique

Cette approche suppose que l'architecture logicielle est bien documentée et utilise la notion de style d'architecture pour représenter la famille d'architecture. Ainsi, l'état initial et l'état cible sont bien représentés dans un langage de modélisation formel. Ils illustrent l'évolution de l'architecture sous forme de chemins graphiques allant de l'architecture initiale à l'architecture cible et chaque chemin représente un scénario possible d'évolution dans lequel il peut comprendre plusieurs états intermédiaires. Deux outils prototypes ont été développés pour mettre en œuvre la conception des styles d'évolution.

- L'atelier Ævol (Figure 5.1) a été développé en tant que plug-in Acme Studio (Garlan & Schmerl, 2009). La structure prend en charge l'analyse et la planification de l'évolution en permettant aux architectes de définir un graphique d'évolution et d'associer ses nœuds à des architectures système représentées dans Acme Studio. À chaque transition architecturale, un ensemble de propriétés est associé pour calculer l'utilité globale du chemin. La comparaison des chemins peut donner lieu à la sélection d'un chemin d'évolution.
- La deuxième implémentation représente une extension de l'outil MagicDraw qui est un outil UML commercial. Ce prototype fournit certaines fonctionnalités et prend en charge plusieurs vues architecturales de l'état d'évolution, à travers lequel un diagramme de composant et un diagramme de déploiement peuvent être liés à des nœuds pour représenter ces deux vues (Barnes & Garlan, 2013).

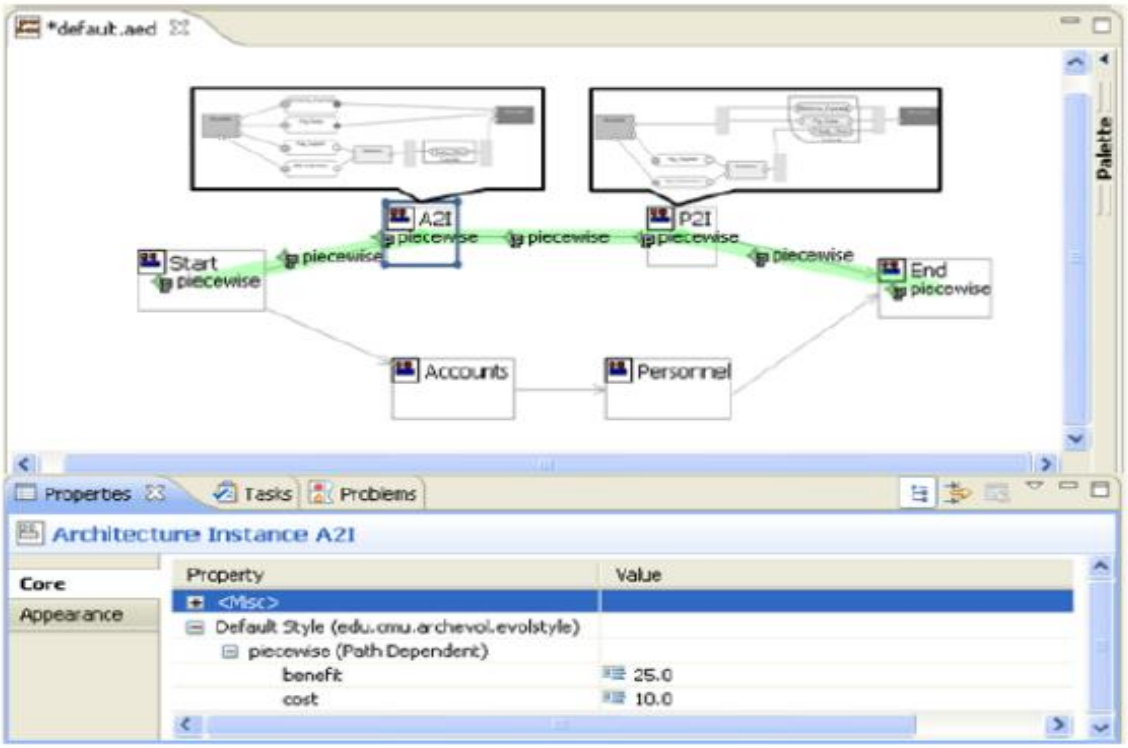

Figure 5.1- Capture écran Ævol Workbench.

5.2.1.2 La méthode Oussalah(Oussalah, et al., 2006)

Cette méthode avait pour objectif d'améliorer l'évolution des logiciels en proposant un modèle baptisé SAEv (Software Architecture Evolution model). Le travail se focalise

exclusivement sur l'aspect structurel pour décrire et gérer l'évolution logicielle à partir de son architecture. A cet effet, ils considèrent les éléments architecturaux tels que les composants, connecteurs, interfaces et configurations comme des entités de première classe. SAEv s'appuie sur ses propres concepts et son propre mécanisme d'évolution. L'évolution de chaque élément architectural est gérée à travers les stratégies d'évolution et les règles d'évolution. Toutes ces dernières décrivent les opérations d'évolutions que l'on peut appliquer sur un élément d'architectural.

Fondement théorique

SAEv consiste à considérer une architecture logicielle cohérente et d'appliquer des changements qui sont exprimés par les architectes puis de suivre les impacts survenus et enfin d'aboutir à une architecture logicielle évoluée supposée cohérente. Le modèle s'appuie sur des niveaux d'abstraction et répond à un ensemble de préoccupations énumérées ci-après :

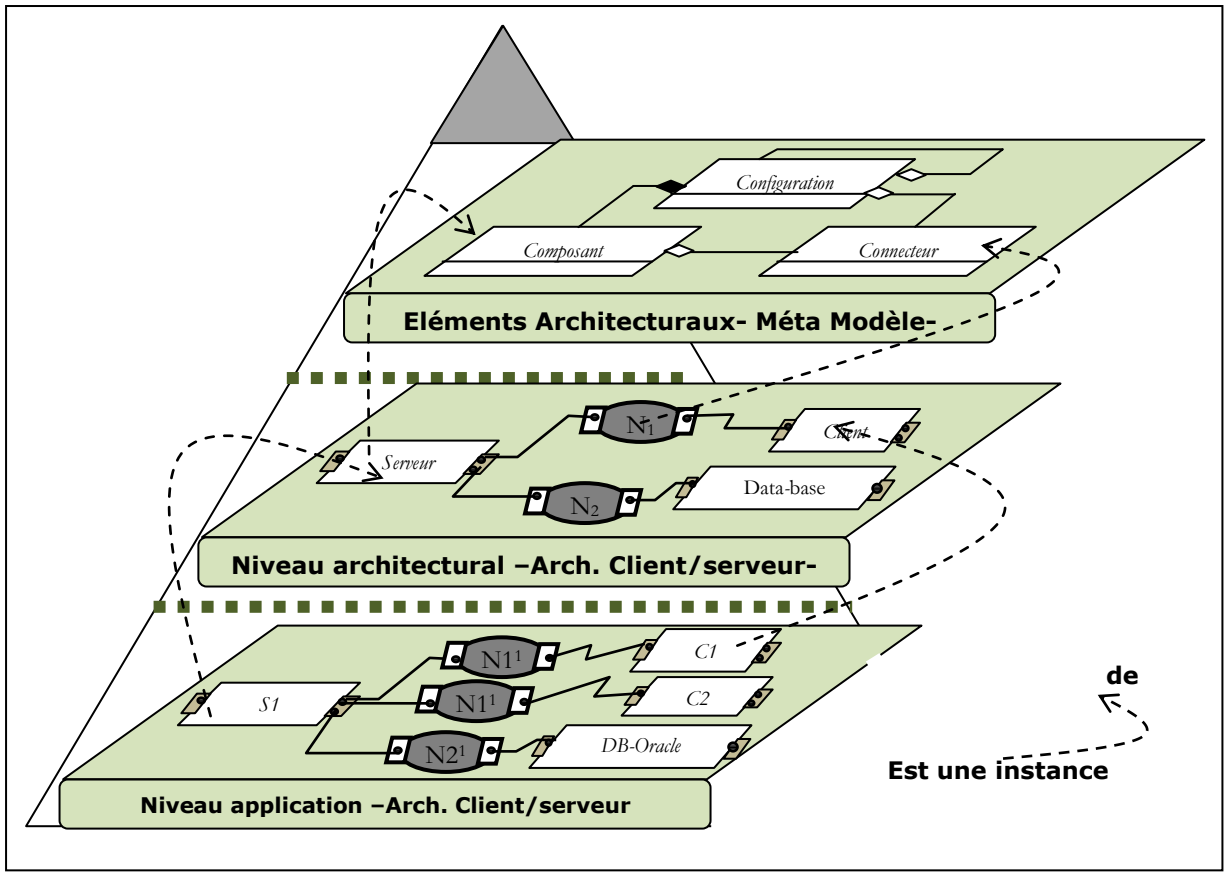

Figure 5.2- Présentation du modèle SAEv.

[Source: (Sadou-Harireche, 2007)].

- *Uniformiser le processus d'évolution de tous les éléments*- Cela passe impérativement par la réification des descriptions de ces éléments à travers l'ADL associé aux différents niveaux d'abstraction de l'architecture logicielle.
- *Gérer la propagation des impacts*- à travers le suivi de tous les changements opérés afin de préserver la cohérence du modèle.

- *Permettre l'évolution dynamique au même titre que l'évolution statique*- L'évolution peut être opérée pendant l'arrêt (statique) ou dynamiquement au cours de son exécution (temps de chargement ou temps de compilation … etc.)
- *Permettre la réutilisabilité des spécifications liées à l'évolution*- le modèle permet une facilitation de sélection et de réutilisation des spécifications d'évolution déjà définies.

Le niveau d'abstraction est un élément essentiel dans l'opération d'évolution d'architecture. L'objectif est d'assurer un couplage faible entre le traitement d'évolution des éléments de chaque niveau de conception et du comportement propre de chacun d'eux. Cela revient à expliciter l'évolution elle-même indépendamment de la spécification des éléments. Cela est souvent une caractéristique inhérente au langage de description avec lequel ces éléments ont été spécifiés la première fois (les éléments C1 et C2 dans la Figure 5.2). A l'issue de cette description un méta-modèle présenté en diagramme classe UML, montre les concepts clés en termes de classes, d'héritages, de compositions et d'associations qui sont utilisés dans le modèle SAEv (Figure 5.3).

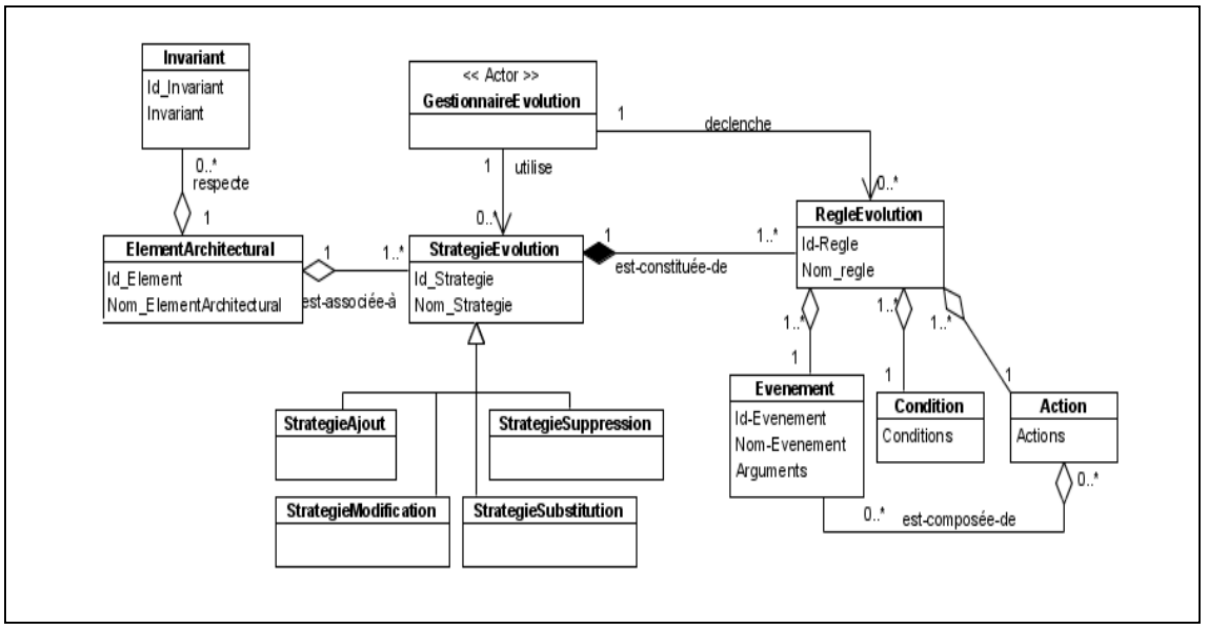

Figure 5.3- Méta-modèle SAEv.

[Source: (Sadou-Harireche, 2007)].

Fondement technique

Le processus d'évolution a été scindé en deux phases consécutives. La première phase est la spécification de l'évolution et la deuxième phase est l'exécution de l'évolution. Les propositions ont fait l'objet d'une illustration dans le cadre de l'ADL COSA (Component-Object based Software Architecture) (Adel & Abbdellah, 2010; ALTI, 2014; Alti & Smeda, 2006).

Etape 1-La spécification de l'évolution- Cette étape consiste à spécifier les :

- Eléments architecturaux à évoluer,
- Stratégies d'évolution associées aux éléments architecturaux à travers des règles d'évolution,

- Règles d'évolution jugées nécessaires pour l'évolution afin d'implémenter une architecture de type Client/Serveur,
- Invariants associés aux éléments architecturaux.

Etape 2- L'exécution d'une évolution- Cette phase se fait après la spécification de l'évolution, un gestionnaire d'évolution effectue l'opération d'évolution comme par exemple, l'ajout ou la suppression. Ensuite, Il entame les opérations suivantes en fonction des stratégies et des règles d'évolution bien définies à priori de la façon suivante :

- Identification de la stratégie et de la règle d'évolution à exécuter- celle-ci est déterminée en fonction de l'élément et de l'opération invoquée puis sélectionner les règles d'évolution dont les conditions sont satisfaites.
- Exécution de la règle d'évolution- le gestionnaire procède à l'exécution des règles action par action,
- Vérification des cohérences, le gestionnaire lance une opération de vérification de la cohérence par la considération des invariants associés aux éléments ayant fait l'objet d'évolution. C'est au concepteur de corriger les incohérences si elles sont détectées.

Il est important de souligner qu'avec SAEv des propriétés sémantiques ont été intégrées dans l'ADL COSA afin de permettre la gestion et la propagation des impacts d'une évolution. Une illustration de l'applicabilité de la méthode à travers les propriétés sémantiques a été bien démontrée par l'adoption de la spécification d'architecture Client/serveur en COSA. L'opération d'intégration des propriétés sémantiques dans le concept connecteur de COSA est effectuée par la modification de la méta-classe associée. Il s'agissait d'ajouter ces propriétés en tant qu'attributs descriptifs. De plus, pour chaque propriété représentant l'attribut ajouté, il lui est associé une valeur par défaut. Sous la contrainte qu'ils ne considèrent que les connecteurs qui sont partagés, indépendants et non prédominants.

5.2.2 Méthode d'évaluation

Dans cette sous section, nous allons présenter la méthode de calcul adoptée pour évaluer l'approche d'évolution. Cette évaluation permet d'estimer les dimensions que nous avons proposées dans le cadre de cette étude. Ainsi pour simplifier la compréhension de l'exemple, certaines hypothèses ont été formulées.

5.2.2.1 Hypothèses de l'évaluation structurelle

Nous avons opté pour une échelle de trois valeurs pour représenter trois informations extraites de la méthode à évaluer. Ces valeurs sont :

- Zéro pour les critères qui ne sont pas traités dans la méthode ou si la méthode ne fait pas référence à ce critère.
- 0.5 pour représenter une métrique des critères proposés qui a été implicitement évoquée au cours de l'élaboration de la dite méthode.
- 1 pour les dimensions qui sont explicitement mentionnées et traitées dans la technique proposée.

Pour permettre la représentativité sur l'unité i.e. 100%. Toutes les valeurs sont calculées à la base des données exprimées puis transformées à l'unité. Comme, il a été déjà présenté dans le chapitre précédent dans la section des relations inter et intra dimensions, les dimensions exclusives sont évaluées à l'unité, tandis que pour les dimensions qui sont complémentaires la valeur est une combinaison des métriques associées.

5.2.2.2 Hypothèses de l'évaluation qualitative

De même pour la qualité, l'évaluation touche à deux représentations différentes :

- Pour les poids associés à chaque capacité de qualité, nous optons pour une estimation à poids égale i.e. que a = b = c = e = d. Cela veut dire que dans notre cas, nous ne donnons aucune priorité pour aucune capacité. Il est important de savoir que les valeurs peuvent être choisies de façon à favoriser certaines capacités par rapport à d'autres pour imposer une qualité qui prend en charge une capacité au détriment d'autres.
- Le paramètre p qui représente le degré de complexité du critère est mis à un pour dire que les paramètres ne considèrent qu'un seul niveau de description du critère. Le paramètre p indique la profondeur de représentation d'un critère de qualité.
- La représentativité indique le pourcentage de considération d'une capacité par rapport aux autres. Nous avons défini trois niveaux de représentativité : le premier est faible pour ceux qui justifient un rapport inférieur à 1/3, moyenne pour un rapport supérieur ou égale à 1/3 et strictement inférieur à 2/3 et forte autrement.
- Pour permettre la représentativité sur l'unité i.e. 100%. Tous les critères sont calculés à la base de ce qu'ils expriment puis sont transformés à l'unité. Puisque nous avons considéré que le degré de complexité vaut un, alors tout critère représente l'unité. Il s'agit seulement de transformer les différentes capacités par rapport à l'unité qui représente l'unité de la qualité composée de la somme des trois capacités exprimées dans notre recherche.

5.2.3 Résultats et discussion

A travers les résultats obtenus que nous avons récapitulés dans la Table 5.1, nous pouvons quantifier les deux méthodes selon les deux aspects que nous avons traités dans notre référentiel.

5.2.3.1 Aspect structurel et comportemental

Pour la dimension de niveau hiérarchique par exemple, la première approche a explicitement exprimé l'évolution aux différents niveaux de modélisation et a montré implicitement la prise en compte des niveaux d'abstraction. Cela conduit à considérer la valeur de 1,5 (75%) de toute l'expressivité de cette dimension. Tandis que la seconde approche exprime implicitement le niveau d'abstraction ayant pour pourcentage 25%. Ces pourcentages servent à l'évaluation et à la comparaison pour déterminer l'orientation de l'approche par rapport aux dimensions proposées. De la même manière, les autres dimensions ont été analysées et comparées en conséquence. Il est intéressant de noter le cas de la dimension type, qui est une métrique combinée, c'est-à-dire que toutes les valeurs possibles constituent l'unité entière à 100%. Dans ce cas, chaque sous-dimension est divisée par deux et la somme des options obtenues est calculée en conséquence.

Table 5.1- Résultat de l'étude structurelle des deux approches évolutions.

Dimensions	Evaluation Métrique		Approche 1 (Oussalah, et al., 2006)		Approche 2 (Garlan, et al., 2009)	
			Valeur	%	Valeur	%
Objet d'évolution	Artefact Processus		1 0	100	1 0	100
Niveau hiérarchique	Modélisation Abstraction		1 0.5	75	0 0.5	25
Temps d'évolution	Exécution Conception		0 1	100	0 1	100
Mécanisme opératoire	Réduction Emergence		1 0	100	0.5 0	50
Type d'évolution	Forme	Open Break Open Seamless Close Break Close Seamless	1 0 0 0	12.50	0 0 1 0	12.50
	Catégorie	Corrective Perfective Adaptative	1 0 0	16.67	0.5 0.5 0	16.67
Intervenant	Académique Professionnel		1 0	100	1 0	100

Les sous dimensions, formes et catégories principales, sont calculées comme $\sum valeurs\ des\ métrics/2$

5.2.3.2 Classification

Le calcul de la dimension des niveaux hiérarchiques (en réponse à la question où ?) et de la dimension de la mécanique opératoire utilisée pour traiter l'évolution (en réponse à la question comment ?) révèlent que l'approche SAEv présente un mécanisme opératoire de réduction aux différents niveaux hiérarchiques, par conséquent, l'approche est classée dans la catégorie de l'évolution en réduction orientée niveau de modélisation (classe 3). Tandis que la méthode de Garlan et al. décrit l'évolution en réduction à travers plusieurs niveaux d'abstraction pour le même niveau architectural de modélisation, elle est donc associée à l'évolution en réduction orientée niveau intra abstraction (classe 2) de la classification proposée. En résumé, la première approche est axée sur la modélisation des niveaux de l'architecture, tandis que la seconde repose sur les niveaux d'abstraction principalement utilisés pour réduire la complexité de l'évolution. Il convient de noter que les deux approches montrent des valeurs identiques pour l'objet, le type, les intervenants, le temps d'évolution et toutes les deux s'appuient sur le «réductionnisme» comme mécanisme opératoire d'évolution.

5.2.3.3 Aspect qualitatif

L'évaluation de l'aspect qualitatif en fonction des hypothèses formulées précédemment nécessite la lecture et la prospection des documents. L'analyse du travail de (Oussalah et al., 2006) a conduit aux résultats de critères suivants: capacité d'énonciation (F = 1, I = 1, T = 0), capacité d'expressivité (M = 1, A = 0,5, E = 1, O = 1, D = 1) et la capacité de qualité d'évaluation (R = 1, A = 0.5, P = 0, S = 0). On applique ensuite les équations (1), (2) et (3) du chapitre précédent pour obtenir:

$Q_n = (1 \times 1 + 1 \times 1 + 1 \times 0) / 3 = 0{,}67;$

$Q_x = (1 \times 1 + 1 \times 0{,}5 + 1 \times 1 + 1 \times 1 + 1 \times 1) / 5 = 0{,}90;$

$Q_v = (1 \times 1 + 1 \times 0{,}5 + 1 \times 0 + 1 \times 0) / 4 = 0{,}375.$

La qualité attendue exprimée par les différentes capacités:

$$Q \text{ (Oussalah et al., 2006)} = (0,\ 67 + 0,\ 90 + 0,\ 375) / 3 = 0.64$$

Ce qui signifie que le modèle présente une grande représentativité pour l'évolution de l'architecture logicielle. De la même manière, nous avons procédé pour le calcul des capacités de la deuxième méthode. La présentation des différentes capacités, Table 5.2 récapitule les valeurs obtenues et montre par conséquent que les critères de qualité n'ont pas été considérés de la même façon dans les deux approches. Cela est certainement dû aux variations des préoccupations conceptuelles pour traiter l'évolution d'architecture.

Table 5.2- Comparaison de l'aspect qualitatif des deux approches.

Les attentes qualitatives	Critères de qualités	Approche 1 (Oussalah, et al., 2006)	Approche 2 (Garlan, et al., 2009)
Capacité d'énonciation	Formalisation	1	1
	Propagation des impacts	1	1
	Traçabilité	0	1
Capacité d'expressivité	Niveau de modélisation	1	0
	Niveau d'abstraction	0.5	0.5
	Mode d'expressivité	1	1
	Mode opératoire	1	1
	Domaine	1	1
Capacité d'évaluation	Réutilisabilité	1	1
	Adaptabilité	0.5	0.5
	Perfective	0	1
	Support	0	0

De plus, la Figure 5.4 montre que l'approche de SAEv s'est focalisée fortement sur la capacité d'expressivité et d'énonciation avec un degré moindre au détriment de la capacité d'évaluation. Ce qui signifie que l'approche est bien orientée vers l'expressivité d'une méthode d'évolution. De la même façon, la deuxième approche montre un pourcentage fort de l'énonciation et de l'expressivité mais avec une tendance favorable pour l'énonciation. Cela est justifié par le fait que l'approche prend en charge une problématique bien définie et bien cadrée. Les deux approches considèrent moyennement la capacité d'évaluation ce qui atténue son application ou l'adoption de la méthode par le secteur économique et ou industriel.

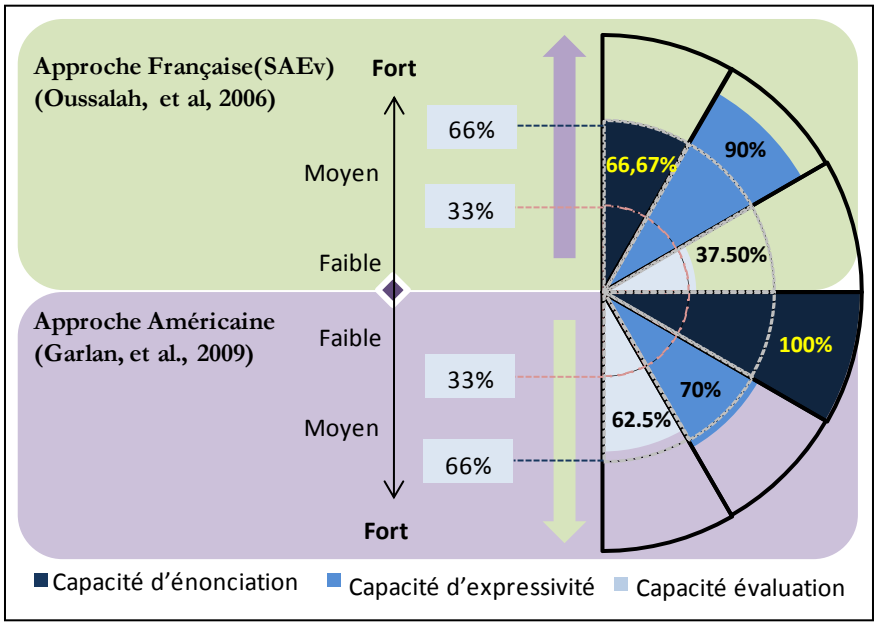

Figure 5.4- Présentation de l'aspect qualitatif des deux approches.

5.3 Etude analytique de méthodes

Après avoir montré la méthode de calcul des dimensions et des différents critères de qualité d'une approche d'évolution, nous allons dans ce qui suit, présenter une analyse sur un échantillon d'approches bien connues dans le domaine.

5.3.1. Présentation de l'échantillon

Pour mieux représenter notre échantillon, nous avons sélectionné un ensemble d'articles qui sont les plus connus dans le domaine de l'architecture logicielle. Nous nous sommes basés essentiellement sur les articles et les rapports accessibles au public. En contre partie, les outils industriels et livres ne sont pas pris en compte pour des raisons de limitation d'accès pour les premiers et pour leurs généricités pour les seconds. Sachant que jusqu'à ce jour, les approches d'extraction ne sont pas bien spécifiées, la sélection des approches doit être faite alors à la base de mots clés bien spécifiques. De plus et par manque d'une réification du concept d'architecture logicielle, la recherche des approches et méthodes suppose une bonne lecture et compréhension des idées avancées. Cependant, nous avons appliqué un processus de sélection rigoureux par l'application d'un processus de sélection en deux étapes différentes.

5.3.1.1 Première étape: Sélection des approches

Nous avons considéré les approches évoquant l'évolution dans le domaine de l'ingénierie logicielle qui représente un domaine plus vaste que celui des architectures logicielles et qui a donné le premier lancement pour le phénomène d'évolution logicielle. Nous considérons aussi les approches qui n'évoquent pas spécifiquement l'évolution des architectures logicielles, mais aussi des artefacts connexes tels que les patrons de conception, les processus d'évolution, le changement, la transformation et styles d'évolution, car ils font référence souvent à l'architecture logicielle. De plus, les études d'évolution estimant la qualité d'évolution, en utilisant des

techniques d'évolution spécifiques, sont aussi essentielles et importantes car elles étendent la taxonomie d'évolution architecturale par la considération des facteurs de qualité.

Afin de fournir des éclaircissements et donc une compréhension plus large de la taxonomie proposée, les documents sélectionnés ont été divisés en cinq catégories thématiques afin de refléter le plus large éventail de mécanismes bien connus pour faire face à l'évolution (Gasmallah, Amirat, & Oussalah, 2016). Chaque catégorie comprend un ensemble de méthodes qui partagent des réflexions conceptuellement bien proches malgré l'utilisation de mécanismes différents sans se soucier à ce que l'environnement de prolifération d'architecture est statique ou dynamique. Les approches ont été ventilées comme suit :

- Approches d'évolutions basées changement (Evolution Change-based Approaches)

Evolution Change-based Approaches (Bengtsson, Lassing, Bosch, & van Vliet, 2004a; Gottlob, Schrefl, & Röck, 1996; Oreizy, Gorlick, Taylor, Heimbigner, et al., 1999)- Ces travaux englobent les approches traitant la problématique du changement du logiciel, plusieurs techniques de changement ont été abordées pour manager des corrections et ou des adaptations. Ils incluent les travaux relatifs à la:

- *Maintenabilité :* selon la norme (Lane, Gu, Lago, & Richardson, 2010), la maintenance reflète les différentes modifications apportées à un logiciel, après sa mise en œuvre, sans avoir le risque d'ajouter de nouvelles erreurs. Ces modifications sont de nature (i) corrective qui relèvent de la modification de code pour répondre aux bugs rencontrés sinon elles interviennent en tant que curative, (ii) la maintenance adaptative qui vient en réponse aux problème liés à l'anticipation des changements environnementaux «portabilité» i.e. le transfert des données à d'autres représentations de données ou sur d'autres environnements (iii) la maintenance perfective qui vise à rendre les programmes plus parfaits au niveau de la conceptualisation et l'implémentation. Les opérations de maintenance sont déclenchées une fois certaines conditions sont vérifiées (Bennett & Rajlich, 2000; Chapin, Hale, Khan, Ramil, & Tan, 2001; Swanson, 1976).
- *Modifiabilité:* décrit la capacité d'une architecture à être modifiée en réponse à des modifications dues soit aux exigences de l'environnement soit à des besoins ou à des spécifications fonctionnelles (Bass, Clements, & Kazman, 2003; Bengtsson, Lassing, Bosch, & van Vliet, 2004b),
- *Auto-Changements(self-*):* regroupe les travaux de: (i)-Computation automatique- «*Automation is to replace human manual control, planning and problem solving by automatic devices and computers*» (Bainbridge, 1983). La computation automatique se fonde sur les techniques self-configuration, self optimization (pour rendre le système fonctionnel et effectif) , self-protection (pour protéger le système des attaques malveillantes) et le self-healing (pour doter le système de la faculté de couvrir les erreurs de façon automatique)(Bahl & Wason, 2012). (ii) Self-management- Consiste à implémenter non seulement les changements à l'intérieur des architectures mais aussi à initialiser, sélectionner et à évaluer le changement sans une intervention externe. C'est un type spécifique des architectures dynamiques (Bradbury, Cordy, Dingel, & Wermelinger, 2004). (iii) Self-organisation- Consiste à produire, spontanément sans contrôle externe, une nouvelle organisation suite à un changement d'environnement ou d'organisation (De Wolf & Holvoet, 2004; Di Marzo Serugendo, Fitzgerald, Romanovsky, & Guelfi, 2006; Haken & Jumarie, 2006). (iv) Self-adaptative- est le mécanisme dans lequel le système modifie son propre comportement ou structure architecturale de façon autonome par le biais d'opérations de compositions, d'ajouts

et de suppressions de composants. Le but derrière cette opération est de s'adapter aux changements liés à l'environnement d'exploitation (Aponte & Simonot, 2008; Oreizy, Gorlick, Taylor, Heimhigner, et al., 1999; Salehie & Tahvildari, 2009).

- Approches d'évolutions basées Algorithmiques (Algorithmic-evolution based) (Engel & Browning, 2008; Wermelinger & Fiadeiro, 2002) : Elles incluent toutes les réflexions pour améliorer l'architecture logicielle en fournissant des structures simples traitant la similarité des objets. Cette catégorie d'approches contient :

- *Catégorisation:* consiste à identifier les similitudes des objets et de les regrouper en catégories (Jäkel, Schölkopf, & Wichmann, 2008),
- *Réorganisation, restructuration, reconfiguration et adaptation:* la restructuration et la réorganisation consistent à modifier l'architecture initiale pour rendre l'architecture plus simple, plus compréhensible et moins sensible à l'erreur en cas de changement. Ce sont des transformations dans le même niveau d'abstraction avec préservation du comportement externe du système (Mens, Buckley, Zenger, & Rashid, 2003; Opdyke, 1992). Quant à l'adaptation et la reconfiguration consistent à modifier la structure du comportement de l'architecture à tout moment et quel que soit le temps d'adaptation. Cette activité est connue par une évolution en temps d'exécution (Bradbury, et al., 2004).
- *Approches incrémentales:* il s'agit de capturer l'évolution des d'informations à traiter par interaction avec l'environnement à travers des outils associés (l'IDE par exemple) (Robbes & Lanza, 2007).
- *Refactoring:* reflète le processus de changement de façon à ne pas altérer le comportement extérieur du code qui a fait l'objet d'une amélioration (Mens, et al., 2003).

- Approches d'évolution basées traçabilité (Cicchetti, Di Ruscio, Eramo, & Pierantonio, 2008; Herrmannsdoerfer, Benz, & Juergens, 2009): Elles incluent des mécanismes dans lesquels la traçabilité est considérée comme un élément clé de l'évolution pour promouvoir la cohérence et la conformité des changements. Cette catégorie englobe les mécanismes liés à:

- *Migration :* désigne l'activité de développement dans laquelle les instances du modèle sont mises à jour pour rétablir la consistance entre le modèle conforment à un méta-modèle d'évolution (Rose et al., 2014). Ce mécanisme consiste à migrer une architecture initiale vers une autre nouvelle suite à un besoin de nature quelconque. Quand la migration de l'architecture cible est achevée, l'ancien système devient inopérationnel (Tilley & Smith, 1995).
- *Versioning* désigne le processus d'enregistrement des différentes étapes conceptuelles et de mise à jour établit sur une version d'un système (Casais, 1992). Le versioning permet de savoir quand les modifications ont été faites sur l'ancienne version et de capturer les variations de modélisation intrinsèques à un domaine d'application sans qu'il y soit une perte d'informations.
- *Point-Vues,* une bonne architecture est celle qui a la capacité d'accumuler les traces de son évolution. Les vues représentent des collections d'éléments du système et les relations qui les associent. Les vues d'une architecture sont documentées selon des modèles (templates) par le concepteur de l'approche. En effet, une vue d'une architecture est toujours conforme à un style d'un viewtype (Dijkman, Quartel, & van Sinderen, 2008; Kheir, Naja, Oussalah, & Tout, 2013).

- Approches d'évolutions basées sur la transformation (Engels & Heckel, 2000; Zhao, Kong, Dong, & Zhang, 2007) : Elles incluent des mécanismes dans lesquels les modifications sont appliquées en utilisant une ou plusieurs règles de transformation pour transformer ou vérifier ou valider des modèles (Mens & Van Gorp, 2006). Elles renferment :

- *Approches de transformation des graphes :* C'est une abstraction qui désigne une séquence de modification par application d'une ou plusieurs règles de transformation (Amirat, Bouchouk, & Gasmallah, 2011). Ces approches incarnent souvent soit des outils de transformation générique (AGG, PROGRESS) ou d'ingénierie (Fujaba) soit de transformation de modèle (CreAT, MOLA) soit de modèle de vérification et de validation (VIATRA, GROOVE, CheckVML).
- *Approches MDA (Model Driven Architecture): Sont* appelées aussi approches de transformation de modèles. Elles consistent à faire des raffinements horizontaux (PIM-PIM, PSM-PSM). Ils consistent aussi à un raffinement hiérarchique à trois paliers qui partent d'un modèle indépendant de l'automatisation (CIM) à un modèle indépendant de la plate-forme (PIM) et finalement traduit en un modèle spécifique à la plate-forme (PSM).

- Approches basées style d'évolution : Ces approches incluent les approches de modélisation d'architecture à la base de style ou pattern de haut niveau de modélisation. Nous rappelons qu'une évolution de style définit une famille de domaines spécifiques qui partagent des propriétés communes et qui doivent satisfaire des contraintes communes. Les styles aident à spécifier la structure fondamentale pour faire évoluer une architecture logicielle. Nous avons considéré un nombre important de ce type études du fait que ces approches mettent l'architecture au centre du processus de développement.

En bref, la sélection des articles s'est essentiellement basée sur les critères de sélection suivants:

- Articles traitant l'évolution des architectures logicielles ou la thématique de l'évolution du génie logiciel, ou des deux,
- Articles utilisés dans le cadre d'autres recherches de classifications pour l'évolution,
- Articles originaux qui ont marqué un impact dans la communauté, et ce par la considération du nombre de citation du papier.

Table 5.3- Répartition de l'échantillon par catégorie thématique.

Catégories	Présélection	Sélection
Approches d'évolutions basées changement	23	18
Evolution Approches basées Algorithmique	21	11
Evolution Approches basées traçabilité	18	10
Approches d'évolutions basées sur la transformation	10	04
Approches basées sur le style d'évolution	36	24
Total	108	67

Nous avons exploré 108 travaux (articles, thèses et rapports) appartenant aux catégories citées ci-dessus. Nous avons sélectionné ceux qui sont les plus significatifs pour la problématique que nous adressons. Cela sous-entend que les articles qui n'ont pas été retenus ne manquent en aucun cas de pertinence du fait de ne pas les inclure dans notre analyse. Au total, nous avons

sélectionné 67 papiers, dont certains décrivent et définissent l'architecture logicielle comme le montre la Table 5.3. La sélection a été motivée par l'excellence du travail, son originalité ou son impact sur la communauté, tel qu'il est perçu à travers le nombre de références de la méthode revue dans la littérature. Nous avons également considéré les approches de type enquête (survey) dans l'objectif est de capturer les études qui ont eu un impact dans le domaine et qui ont servi comme une base de connaissances pour effectuer des classifications ou de feuille de route dans des domaines connexes.

5.3.1.2 Deuxième étape: Evaluation des approches

Dans cette étape, nous soutenons la comparaison des approches avec un tableau pour chaque aspect (structurel et qualitatif) de cette enquête. Dans ces tableaux, nous répertorions les récapitulations de l'évolution de chaque étude sélectionnée à travers l'utilisation d'une fiche, en annexe, de prospection du papier de recherche étudié. Les tableaux récapitulatifs montrent par des chiffres les contributions importantes, c'est-à-dire celles qui ont été d'influence importante pour la communauté et qui ont suscité plus de travaux et celles adoptant une perspective ou une approche spécifique abordant le problème d'évolution de manière générale c.-à-d., qui n'ont pas montré beaucoup d'intérêt mais qu'elles sont toujours intéressantes du point de vue de l'enquête.

Nous rappelons que la préparation de l'analyse, conformément à (Wohlin et al., 2012), a été réalisée en élaborant une fiche d'étude prospective. Une fois que les informations appropriées ont été recueillies, les calculs ont été effectués comme nous les avons présentés dans l'exemple de la section 5.2. Par la suite, le pourcentage de représentativité a été évalué et les études à faible estimation ont été supprimées. Enfin, 67 communications ont été sélectionnées et sont, à notre connaissance, les études les plus représentatives dans le domaine.

5.3.1.3 Troisième étape: Comparaison des approches

La troisième étape consiste à calculer les différents taux de représentativité des dimensions de notre référentiel. Cette étape permet à notre référentiel de décrire puis de classifier les approches existantes. Il est aussi important de savoir que cette description peut être faite en aval de l'implémentation d'une évolution architecturale pour guider et assister un choix quelconque d'évolution. En pratique, cette comparaison permet aux architectes d'explorer des pistes non encore abordées dans la problématique d'évolution.

5.3.2 Résultats et discussions

Les résultats obtenus après prospection des 67 papiers sélectionnés pour l'évolution ont été dispatchés selon les aspects du référentiel.

5.3.2.1 Référentiel de description de l'évolution (aspect structurel)

Les travaux examinés ont été comparés au référentiel à l'aide des pourcentages de représentativité, qui représentent le quotient du nombre d'expressions explicites et implicites (pondérées) de critères par rapport au nombre total des travaux dans une catégorie. Par la suite, les pourcentages ont été reportés dans le tableau 5.4.

A travers les résultats obtenus structurés dans la Table 5.4, nous pouvons tirer les conclusions suivantes :

- Pour objet d'évolution : L'artefact a pris plus de considération en tant qu'objet d'évolution avec plus de 70%, au détriment de l'objet processus, qui reste une direction très prometteuse (moins 30%), notamment parce que le processus est généralement considéré comme une spécification dynamique de l'objet. De plus, le choix est supporté par le fait que l'artefact constitue une alternative avantageuse grâce aux spécifications de modélisation UML,

Table 5.4- Résultats de la description structurelle des approches d'évolution.

N°	Dimensions	Métriques	%
01	Objet d'évolution	Artefact Processus	71.64 28.36
02	Niveaux hiérarchiques	Modélisation Abstraction	67.76 32.24
03	Mécanismes opératoire de l'évolution	Réduction Emergence	89.56 10.44
04	Type d'évolution	Forme Catégorie	44.67 55.33
05	Temps d'évolution	Exécution Design	20.97 79.03
06	Intervenant	Académique Professionnel	86,49 13.51

- Le niveau d'abstraction : Ce niveau a suscité moins d'intérêt (près de 33%) dans la communauté de l'évolution. La seule raison justifiable à notre connaissance est que les limites de distinction entre les concepts de modélisation et d'abstraction ne sont pas encore bien claires,

- La mécanique opératoire : La réduction est clairement le plus répondue dans le domaine d'évolution, avec un score supérieur à 89% contre 11% pour l'émergence principalement parce que le développeur opte souvent pour un raisonnement mathématique rigoureux basé sur une logique formelle, qui favorise le processus de déduction au lieu de l'induction,

Table 5.5- Répartition des pourcentages selon les sous-dimensions du type d'évolution.

Main	Métriques	%	∑ %
Formes	Ouvert-rupture Ouvert-continu Fermé-rupture Fermé-continu	26.00 6.67 6.67 5.33	44.67
Catégories	Corrective Perfective Adaptative	28.00 18.00 9.33	55.33

- Le type d'évolution : Le type catégorie principale (55%) est relativement la mieux représentée par rapport au type de forme principale. Cela est principalement dû au nombre conséquent des travaux d'évolution qui sont basés sur la maintenance. Si nous forons dans cette dimension, l'étude a également révélé un taux écrasant pour le type "Prédictible" (97%), parfaitement justifié par la dynamique approximative de l'environnement et l'instabilité des exigences des intervenants

aux différents niveaux de modélisation, avec un avantage pour le type "Ouvert" plus de 70%. En outre, l'étude indique que plus de 80% des sélections sont consacrées à une typologie corrective et perfective. Il convient en outre de noter que l'adaptabilité a présenté un faible pourcentage de représentation (moins de 20%), principalement en raison de la difficulté d'évolution dynamique au moment de la conception (Table 5.5).

- Le temps de l'évolution : Le résultat montre que presque le quatre cinquième des études d'évolution architecturale sont focalisées sur le temps de conception (Design-time) au détriment du temps d'exécution (Run-time). Il est donc plus intéressant que les architectes et chercheurs prêtent plus d'attention à l'évolution architecturale pour les systèmes qui sont en temps réels, et

- Les intervenants : Ils sont plus orientés vers le domaine académique qu'industriel. Cependant, les professionnels croient plus à la spécification de l'évolution pour chaque système contrairement aux académiques qui cherchent à trouver des évolutions (styles d'évolutions) génériques et qui peuvent être réutilisables. Il sera à notre avis plus bénéfique aux deux catégories d'intervenants de regarder dans la même direction par l'intégration des professionnels dans les études de recherches ou d'encourager les études se basant sur des cas réels du secteur économique.

5.3.2.2 Classification des approches

Les mécanismes d'évolution ont été classés en fonction de la taxonomie suggérée. Les buts sont de permettre l'évaluation de la couverture obtenue par chaque catégorie de mécanismes, puis de déduire les classes de la taxonomie les moins prises en compte. En utilisant le pourcentage de représentativité de la même manière que nous l'avons décrit précédemment, les résultats obtenus ont été tabulés puis présentés dans la Figure 5.5. Cette dernière présente les différentes couvertures de chaque mécanisme et précise que les études existantes ont favorisé la classe d'évolution en réduction orientée niveau de modélisation (classe 3), les approches basées sur les traces étant les plus significatives avec 68% de l'ensemble des travaux.

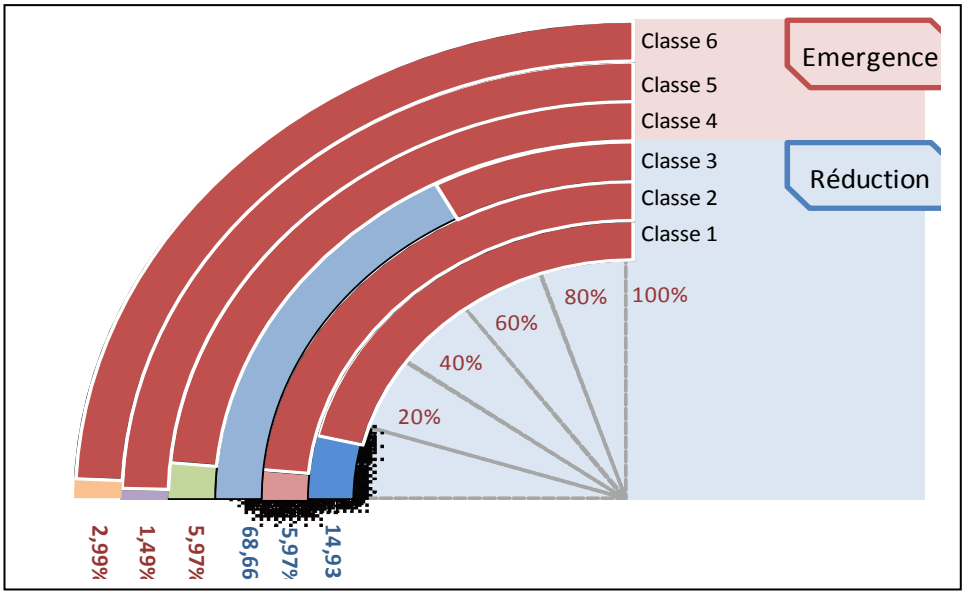

Figure 5.5- Présentation des classes d'évolution proposées.

Les 14% d'études au niveau d'abstraction intra-abstraction (classe 1) sont justifiées par la présence des techniques d'appui à la qualité, à l'évaluation et à l'analyse au niveau architectural. La classification de la sélection révèle un manque important de recherches qui se focalisent sur le mécanisme d'émergence de la dimension MOE.

5.3.2.3 Aspect qualitatif

Dans cette partie, nous avons répertorié les résultats de la prospection des travaux de recherches sélectionnés selon les critères de qualité pour permettre de monter comment notre référentiel dans son aspect qualitatif peut décrire les couvertures en matière critères et facteurs de qualités des méthodes existantes. Selon les résultats obtenus dans la Table 5.6, nous pouvons dire que :

Table 5.6- Evaluation de l'aspect qualitatif de l'échantillon des méthodes d'évolution.

Attentes qualitatives	Critères de qualités	%	Σ %
Capacité d'énonciation	Formalisation	11.80	22.60
	Propagation des impacts	6.16	
	Traçabilité	4.64	
Capacité d'expressivité	Niveau de modélisation	12.51	54.29
	Niveau d'abstraction	6.56	
	Mode d'expressivité	12.21	
	Mode opératoire	10.90	
	Domaine	12.11	
Capacité d'évaluation	Réutilisabilité	6.66	23.11
	Adaptabilité	5.95	
	Perfective	6.26	
	Support	4.24	

La capacité expressive est la plus représentative sur l'ensemble de l'échantillon relativement aux deux autres capacités. Cependant, les critères de qualité les plus respectés dans le domaine de l'évolution sont le niveau de modélisation, le mode d'expressivité, le domaine d'application et finalement la formalisation. Alors que les critères qui suscitent plus d'attention de la part de la communauté sont l'utilité d'implémentation des approches par la fourniture des outils qui supportent ces méthodes de plus de pencher sur la traçabilité des opérations d'évolution effectuées. Les pourcentages donnent une idée claire sur ce qui est fait dans la majorité des techniques proposées et ce qui reste encore à faire.

5.4 Méta classification

Plusieurs classifications se sont investies dans la classification des approches d'évolution d'architectures. La limite qu'ellels posent dés le départ est de ne considérer qu'une sous-catégorie par la spécification de certaines dimensions. Certes ces études passent du niveau général au niveau le plus spécifique et présentent donc plus de détails techniques ce qui restreindra la compréhension de certaine frange d'intervenants non spécialisés. La caractéristique principale de notre référentiel proposé réside dans le fait que le traitement de l'évolution architecturale vise la description pour la compréhension aussi bien des intervenants spécialisés en ingénierie que ceux non spécialisés ce qui permet de trouver un compromis des deux. Notre référentiel représente

donc une vue macro d'évolution et de plus, il ne s'intéresse pas aux détails techniques utilisés pour réaliser l'évolution.

Dans cette section, nous allons procéder à une classification de certaines classifications existantes. C'est ce que nous avons appelé la méta-classification. Les classifications sélectionnées sont les plus souvent connues dans le domaine de l'évolution architecturale.

5.4.1 Présentation des classifications

Nous nous sommes restreints à trois études significatives de classification des approches existantes dans le domaine pour illustrer comment les analyser et les comparer à travers notre référentiel à l'aide des métriques proposées.

5.4.1.1 Classification de Breivold et al. (Breivold, Crnkovic, & Larsson, 2012)

A travers une étude systématique sur 82 travaux existants dans le domaine de l'évolution d'architecture, cinq thématiques principales de catégories ont été présentées:

- Considérations de qualité au cours de la conception de l'architecture logicielle : Cette catégorie concerne les travaux qui se focalisent sur la façon avec laquelle la qualité a été introduite explicitement durant la phase de conception d'architecture. Elle est subdivisée en trois sous-classes:

- Attributs de qualité axés besoins, où chaque décision conceptuelle se base sur une déduction à partir de ces attributs (qui peuvent être des attributs orientés conception, prototype d'architecture, …),
- Attributs de qualité axés scénarios qui considèrent les attributs de qualité à travers des scénarios réels et
- Attributs axés facteurs d'influences à travers une gestion des attributs significatifs et des contraintes liées à la conception.

- Évaluation de la qualité au niveau d'architecture logicielle : Elle concerne les approches d'évaluation conçues pour évaluer et améliorer l'architecture initialement construite après la phase de conception. Elle englobe trois sous classes:

- Classe basée expérience pour les travaux qui exploitent les expériences techniques des intervenants,
- Classe basée scénario où les attributs de la qualité sont évalués à travers un scénario pour une description concrète ce qui a pour objectif d'éviter les ambiguïtés et les interprétations conflictuelles des attributs,
- Classe basée métrique qui s'appuie sur des métriques pour évaluer les attributs de qualités comme la taille du logiciel, l'évolution du ratio (nombre d'évolution par apport à la taille du software), la vitesse de l'évolution ...etc.,

- Évaluation économique dans la détermination du niveau d'incertitude : Ici, la catégorie regroupe les travaux d'évolution de rentabilité c.-à-d. en terme de coûts et d'avantages concurrentiels. Ces attributs peuvent être les coûts de la maintenabilité, l'effort de prédiction de maintenabilité durant la conception architecturale, la quantification des décisions conceptuelles ...etc.

- Gestion de la connaissance architecturale : elle cerne les approches qui se basent sur des sources d'informations dont l'objectif est de capturer la connaissance architecturale.

- Techniques de modélisation : Elles font référence aux travaux qui se focalisent sur la modélisation d'artefacts pour supporter l'évolution d'architecture (eg. Modélisation de la traçabilité entre les besoins, évaluation de l'architecture logicielle et réutilisation, gestion des propriétés de qualité durant tout le cycle de vie, utilisation des tactiques architecturales pour incarner les besoins non fonctionnels dans l'architecture logicielle, développement orienté préoccupation, modélisation de systèmes évaluables par construction d'enveloppes, modélisation de l'impact de changement, modélisation orientée règles commerciales, typologie d'opérations de changement, formalisation du mécanisme de propagation du changement...etc.)

5.4.1.2 Classification de Chaki et al. (Chaki, Diaz-Pace, Garlan, Gurfinkel, & Ozkaya, 2009)

Dans cette étude, les auteurs démontrent que l'évolution architecturale doit être conçue par des modèles formels et soutenus par des analyses et des décisions rationnelles et objectives où l'expérience des architectes fait la différence. Ils croient que l'évolution répond à un schéma d'évolution de forme restreinte où les points conceptuels de départs et d'arrivés sont connus d'avance et peuvent donc être modélisés. L'objectif de cette étude est d'éclaircir les principaux défis de la modélisation de l'évolution qui sont ciblés architecture finale. Donc, elle considère des évolutions pas à pas pour chaque application d'un opérateur d'évolution. La perception des auteurs a ventilé l'existant des méthodes d'évolution en trois grandes classes d'approches.

-Maintenance basée évolution : Ce sont les travaux qui s'intéressent aux décisions de correction et de modifiabilité d'architecture pour répondre aux préoccupations des architectes. Ces travaux nécessitent souvent une intervention au niveau de modélisation le plus bas (code) et englobent les travaux d'évolution au niveau code, le refactoring.

- Evolution ouverte : Elle désigne l'ensemble de travaux dont l'architecture d'arrivée n'est pas connue à priori. A partir d'une architecture de départ, l'approche infère une ou plusieurs solutions respectant un contexte connu à priori. L'incertitude représente l'aspect critique pour de telles approches.

- Evolution fermée : Elle catégorise les travaux qui se fixent une architecture cible dès le départ et modélise la façon de l'atteindre à partir d'une architecture initiale. Il s'agit ici de trouver une séquence d'opérations pouvant guider et assister l'architecte pour atteindre l'architecture qu'il a fixée à priori.

5.4.1.3 Classification Ahmad et al. (Ahmad, Jamshidi, & Pahl, 2014)

A partir de l'étude (Jamshidi, Ghafari, Ahmad, & Pahl, 2013) de classification et de comparaison revue systématique des pratiques et travaux de recherche sur l'évolution logicielle centrée sur l'architecture (ACSE) et qui a statué sur l'importance croissante de la réutilisation dans ACSE. Les auteurs dans la recherche de (Ahmad, et al., 2014) ont déployé des efforts supplémentaires pour prendre en considération la réutilisation de l'évolution et l'adaptation dans le domaine d'évolution architectural. Dans ce contexte, les auteurs ont développé le REVOLVE Framework qui intègre l'acquisition de connaissances et l'exécution de connaissances spécialement pour permettre la réutilisation des connaissances d'architecture d'évolution.

REVOLVE offre une acquisition continue des connaissances en analysant les historiques d'évolution de l'architecture, appelés extraction de l'évolution de l'architecture. Ensuite, cette connaissance peut être exécutée pour effectuer une évolution de l'architecture appelée application de la connaissance de l'évolution de l'architecture (voir Figure 5.6). La classification thématique proposée est axée à la fois sur le temps d'évolution et sur le type d'évolution de la connaissance réutilisée. Un ensemble de caractéristiques spécifiques est fourni dans le but d'affiner chaque catégorie en sous-catégories reflétant une spécification commune en termes d'objet, de concepts de recherche et de contexte.

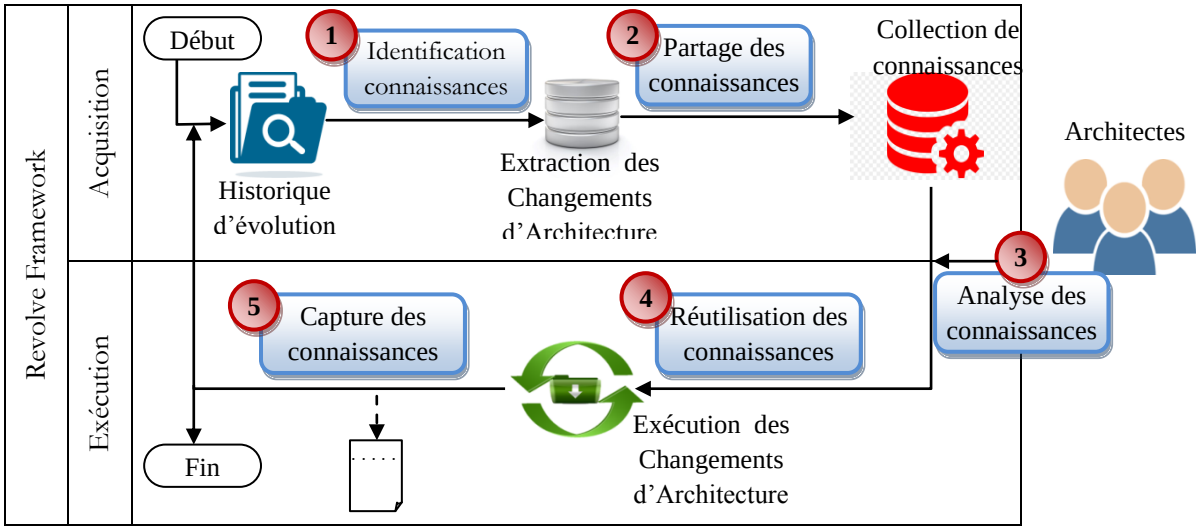

Figure 5.6- Framework REVOLVE: Vue architecturale intégrale.

L'étude a exploré 32 travaux de recherche et se focalise sur les approches dans lesquelles les modifications ont un impact sur le niveau architectural lors de l'analyse et de l'amélioration de l'évolutivité des logiciels. L'investigation des méthodes ou techniques existantes, que ce soit pour une application systématique ou pour l'acquisition empirique de connaissances en architecture, a identifié six grandes catégories de connaissances en matière de réutilisation des évolutions.

- Style d'évolution (Evolution Style) : Les sept approches sélectionnées s'inspirent d'un concept conventionnel de styles d'architecture représentant un vocabulaire réutilisable d'éléments architecturaux (service ou composant ou connecteur) et d'un ensemble de contraintes pour exprimer un style. Les styles d'évolution se concentrent sur la définition, la classification, la représentation et la réutilisation de plans d'évolution fréquents et de l'expertise en matière de modification d'architecture.

- Modification des patrons (Change Patterns) : Ici, treize approches ont été choisies. Elles visent à fournir une solution générique et itérative aux problèmes récurrents de conception. En revanche, les modèles de changement suivent des méthodes et des techniques axées sur la réutilisation pour offrir une solution générique aux problèmes d'évolution. L'évolution est donc prise en charge au moment de la conception et de l'exécution également. Les modèles d'adaptation et de reconfiguration sont les solutions d'évolution à l'exécution. Les solutions traitent également de la coévolution des processus, des exigences et des modèles d'architecture sous-jacents.

- Stratégies et politiques d'adaptation (adaptation strategies and policies) : La catégorie, formée de six études, se concentre sur la réutilisation et la personnalisation des stratégies d'adaptation, des stratégies réutilisables. Elle est fondée sur les connaissances et les aspects destinés à faciliter la réutilisation des stratégies dans les architectures auto-adaptatives. Ce type de catégorie axe sur le traitement de la réutilisation basé sur les connaissances lors de l'exécution.

- Découverte de modèle (Pattern discovery) : Les études sélectionnées, de nombre de deux, représentent des méthodes et techniques d'analyse post-mortem de l'historique d'évolution (changements logs et contrôle de version) afin de détecter les modifications récurrentes en tant qu'instances de modèle.

- Analyse de configuration d'architecture (architecture configuration analysis) : Les deux études sélectionnées reflétant les techniques de gestion de configuration pour analyser des configurations architecturales. Cette catégorie se concentre sur l'historique des révisions d'architecture afin de capturer l'évolution et la variabilité puis, de représenter les relations transversales entre les éléments d'architecture en évolution. Cela est particulièrement utile pour classer les modifications en tant que types atomiques et composites et permet entre autre de déterminer dans quelle mesure les modifications architecturales peuvent être mises en parallèles (modifications commutatives et dépendantes).

- Prédiction d'évolution et de maintenance (evolution and maintenance prediction) : les deux études sélectionnées portent sur la prédiction des efforts de maintenance et d'évolution des architectures logicielles. Les études représentent un ensemble de scénarios de changement permettant de prévoir des tâches d'évolutions évolutives, adaptatives dans des architectures et d'une prévision de maintenance.

Le seul reproche pour ces méthodes d'analyse et de comparaison et qu'elles mettent l'accent sur l'aspect technique ce qui limite leurs utilisations que sur les intervenants spécialisés tels que les architectes ou les concepteurs et parfois développeurs. Nous croyons que l'ouverture de ce type de Framework aux intervenants non spécialisés (managers, utilisateurs, stratèges) permettra d'enrichir la vision des intervenants pour la bonne prise de décision.

5.4.2 Analyse des classifications

La projection des différentes classifications vues précédemment sur le référentiel proposé permet de définir la pertinence des différentes dimensions proposées.

5.4.2.1 Aspect structurel

De la même manière de procéder, nous avons calculé les pourcentages de la représentativité selon les différents aspects exposés dans cette thèse.

- Référentiel : Après études des différentes classifications à travers l'étude prospective, nous étions parfois dans l'obligation d'explorer les articles en question pour pouvoir déterminer les valeurs non évoquées dans la prospection. La Table 5.7 contient les différents pourcentages pondérés de tous les documents par rapport à chaque dimension. Le pourcentage de la méta-classification structurelle est calculé en divisant la somme des études exprimant la dimension

multipliée par le nombre des études de la classification considérée sur l'ensemble total des études des trois classifications.

Table 5.7- Evaluation des classifications sélectionnées avec le référentiel proposé.

Main	Etudes 119	Ahmed et al. 32	Breivold et al. 82	Chaki et al. 5	Moyenne pondérée
Objet (Quoi ?)	Artefact	66.67	74.60	20	70.17
	Processus	33.33	25.40	80	29.83
Niveaux (Où ?)	Modélisation	48.38	28.58	60.00	35.22
	Abstraction	51.62	71.42	40.00	64.78
Intervenants (Qui ?)	Académique	90.62	70.73	90.00	76.89
	Professionnel	9.38	29.27	10.00	23.11
Temps (Quand ?)	Temps-réel	34.62	24.44	22.22	27.08
	Design	65.38	75.56	77.78	72.92
OME (Comment ?)	Réduction	96.78	94.29	100	95.20
	Emergence	3.22	5.71	0	4.80
Type (Pourquoi ?)	Formes	10.59	41.86	100	35.89
	Catégories Principales	89.41	58.14	0	64.11

Note : Moyenne pondérée $= \frac{\sum pourcentages \times ni}{\sum ni}$ *où ni le nombre des travaux de l'enquête.*

Les illustrations détaillées, des résultats fournis dans la Table 5.7 concernant la dimension type (y compris les formes principales et catégories principales), sont plus détaillées dans la Table 5. 8. Les résultats présentés dans les deux tableaux indiquent la juste proportion (ou disproportion) entre les pondérations accordées pour chaque dimension proposée. Cependant, le tableau 5.7 indique la détermination implicite de la couverture en utilisant la classification proposée. Ici, les caractéristiques de proportion et de disproportion sont liées à la détermination de la tendance des approches selon une dimension donnée à savoir la dimension type. Les deux approches sont proportionnelles (réciproquement disproportionnelles) lorsque la tendance des deux approches est similaire (réciproquement inverse) pour une dimension donnée, par exemple, les approches de (Ahmad, et al., 2014) et (Breivold, et al., 2012) sont proportionnelles à la dimension *Objet de l'évolution* car elles ont une orientation artefact.

Table 5. 8- Détails du type d'évolution en termes de formes et catégories d'évolution.

Main	Etudes 119	Ahmed et al. 32	Breivold et al. 82	Chaki et al. 5
Forme	Ouvert-rupture	04.71	36.05	80.00
	Ouvert-continu	03.53	04.65	00.00
	Fermé-rupture	02.35	01.16	00.00
	Fermé-continu	00.00	00.00	20.00
Catégorie	Corrective	37.65	24.42	-
	Perfective	28.24	26.74	-
	Adaptative	23.52	06.98	-

En revanche, ces même études sont disproportionnées par rapport à l'approche de (Chaki, et al., 2009) qui est quant a elle orientée processus. De manière explicite, les résultats montrent que plus de trois quarts des études des classifications existantes encouragent explicitement le

niveau de modélisation au détriment du niveau d'abstraction. Cela est essentiellement dû au fait que la relation entre niveaux de modélisation et d'abstraction est souvent confondue. Les architectes favorisent d'autres concepts comme le raffinement, restriction, instanciation …etc. Cependant, ce taux présente pour nous un indicateur sur la pertinence de ces études pour le traitement de la thématique d'évolution architecturale. Le constat empirique tend donc à montrer que le niveau de modélisation est plus adopté par la communauté de l'évolution architecturale.

De plus, les pourcentages montrent qu'un grand nombre d'études portent sur les artefacts en tant qu'objet d'évolution, à l'exception de (Chaki, et al., 2009) qui se concentre sur les études des processus d'évolution. Cela est spécialement dû au fait que les formalismes supportant l'évolution tels que SAEv (Oussalah, et al., 2006) et les langages de description d'architecture tels que par exemple (ADL, UML, xADL) sont tous orientés sur les artefacts. Il est à noter que les styles et les modèles sont en fait plus adaptés aux éléments d'architecture évolutifs (artefacts) mais contribuent peu à résoudre le problème de l'évolution des styles eux-mêmes (Hassan & Oussalah, 2016).

De la même manière, nous constatons un pourcentage écrasant concernant la réduction de l'évolution du niveau supérieur au niveau inférieur, c'est-à-dire la modélisation par hiérarchie descendante. À notre avis, cette prépondérance est principalement due à une influence du raisonnement déductif lié à la méthode descendante (c'est-à-dire réduire l'MOE), comme le montre le Table 5.7. Cependant, ce réductionnisme peut priver les systèmes d'architecture logicielle d'introduire de nouvelles opportunités de solution en fonction des hypothèses d'environnement actuelles et/ou futures.

Table 5.9- Pourcentages des classes des classifications sélectionnées.

MOE réduction				
Etudes	Nombre	Intra-abstraction	Inter-abstraction	Modélisation
Ahmad et al.	32	32.26	16.13	48.39
Breivold et al.	82	48.56	17.15	28.58
Chaki et al.	5	20	20	60
Moyenne pondérée		42.98	17.00	35.23
Moyenne de l'MOE réduction		31.73		
MOE Emergence				
Ahmad et al.	32	3.23	0	0
Breivold et al.	82	5.71	0	0
Chaki et al.	5	0	0	0
Moyenne pondérée		**4.80**	**0**	**0**
Moyenne de l'MOE réduction		**1.60**		

En ce qui concerne la ***dimension temporelle***, les valeurs sont systématiquement en faveur du temps de conception, ce qui, à notre avis, reflète l'intérêt pour l'activité d'anticipation de faire face à l'évolution. Les pourcentages montrent que la sous-dimension du ***type d'évolution*** (catégories principales) a suscité un intérêt explicite dans toutes les études, à l'exception de (Chaki, et al., 2009) comme le montre la Table 5. 8. Il est à noter que la plupart de ces études sont le fruit d'un travail académique et que l'implication des professionnels du secteur industriel reste au dessous des attentes espérées compte tenu de la grandeur de la problématique d'évolution.

- Classification : L'aspect structurel des classifications proposées nous a permis d'élaborer une classification bien détaillée par l'adoption des mêmes critères vus précédemment. La Table 5.9 montre en clair que les classes 1 et 3 sont celles où la plupart des problèmes d'évolution ont forés le plus, tandis que les classes 5 et 6 représentent les maillons faibles des approches d'évolution pour cette classification. Cette classification souligne que les mécanismes d'émergence de l'évolution sont encore sous-exploités et constituent un terrain favorable pour la recherche sur l'évolution des architectures.

5.4.2.2 Aspect qualitatif

Il est toujours intéressant que le développement de systèmes logiciels doit s'effectuer à l'aide de plates-formes permettant de décrire des modèles architecturaux complexes et de qualité. L'évaluation de la qualité des architectures logicielles concerne des aspects quantitatifs et qualitatifs. Elle permet entre autre de guider et de prendre en charge les évolutions associées ainsi que de présenter des limites théoriques qui peuvent entraver le processus d'évolution. Notre référentiel nous a permis dans son aspect qualitatif d'identifier les principales capacités de qualité pour le domaine d'évolution architecturale. La quantification de ces capacités est une vue précoce pour évaluer la qualité d'un système aussi bien que par des intervenants spécialisés que d'autres qui ne le sont pas. C'est sous cette vision que nous avons conçu notre référentiel de qualité. Nous avons effectué un travail d'analyse des différentes classifications en termes de capacités attendues de qualité dont les résultats obtenus sont synthétisés dans la Table 5.10.

Le récapitulatif des capacités montre pour chaque capacité, que:

- Capacité d'énonciation : La capacité d'énonciation est relativement la capacité qui n'attire pas suffisamment l'attention des architectes au cours des opérations d'évolution. Il est claire que le critère de formulation est le plus considéré dans cette capacité ce qui revient à dire que les architectes fixent souvent les formalismes à adopter avant la réalisation de l'opération d'évolution.

- Capacité d'expressivité : C'est la capacité la plus exprimée dans toutes les approches d'évolution. Cela est dû au faite que ses critères sont fondamentaux pour exprimer l'évolution car, même s'ils ne sont pas considérés concrètement, ils sont automatiquement impliqués dans le processus.

- Capacité d'évaluation : Les critères d'évaluations viennent en seconde place. L'évaluation est un processus très important dans l'évolution architecturale du fait que toute expression de méthode doit être évaluée pour montrer sa valeur ajoutée par rapport à autres méthodes similaires. C'est la cause pour laquelle les pourcentages sont aussi élevés que l'énonciation. Notre prototype offre une nouvelle manière de traiter la problématique d'évaluation d'architecture aussi bien pour les systèmes orientés services que pour les systèmes orientés composants. L'intérêt est de fournir à l'utilisateur des résultats « moins abstraites » que ceux que les méthodes actuelles proposent aujourd'hui tels que ATAM[1] ou SAAM[2]. Le concept de qualité demeure subjectif mais permet d'offrir des gains considérables en temps et en coûts.

[1] Achitecture tradeoff analysis method
[2] Software architecture Analysis Method

Table 5.10- Evaluation de la qualité des classifications d'évolution architecturales.

Capacité	Critères	Ahmad et al. (32)	Breivold et al. (82)	Chaki et al. (5)	Capacités (119)
Enonciation	Formalisation	12.34	10.16	13.16	10.88
	Gestion les impacts	01.90	4.26	10.53	3.89
	Traçabilité	00.00	2.79	02.63	2.03
		14.24	**17.21**	**26.32**	**16.80**
Expressivités	Niveau modélisation	15.19	14.75	13.16	14.80
	Niveau d'abstraction	04.43	1.97	2.63	2.66
	Mode d'expressivité	17.41	13.93	13.16	14.84
	Mode opératoire	1.58	8.85	9.21	6.91
	Domaine	10.76	15.57	13.16	14.18
		50.37	**55.08**	**51.32**	**53.39**
Evaluation	Réutilisabilité	14.24	3.61	10.53	6.76
	Adaptabilité	7.59	3.11	3.95	4.35
	Perfective	8.23	7.87	2.63	7.75
	Support	6.33	13.11	5.26	10.96
		36.39	**27.70**	**22.37**	**29.82**
	Attente de qualité Q	**100**	**100**	**100**	**100**

5.5 Conclusion

Dans ce chapitre, nous avons montré l'application de notre référentiel sur ses deux aspects structurel et qualitatif. Le premier aspect a consisté d'évaluer et de quantifier les six dimensions proposées pour décrire et comprendre une architecture logicielle. Cette évaluation permet aux intervenants aussi bien que spécialisés ou non de comprendre la tendance de leurs propositions. Ainsi, elle leurs permet de cartographier l'ensemble des recherches existantes dans le domaine ou encore de détecter les dimensions qui n'ont pas été encore abordées. Le deuxième aspect, qualitatif, nous a permis de montrer l'évaluation et la discussion des capacités qualitatives d'une architecture qui sont attendues d'une évolution logicielle. Nous nous sommes intéressés à représenter la qualité comme une unité regroupant les trois capacités dans le seul objectif est de montrer comment ces capacités ont été traitées par les évolutions existantes. L'application du référentiel a été menée à travers trois niveaux différents :

- Sur un exemple, deux méthodes d'évolution ont été sélectionnées évaluées, classées. L'évaluation qualitative a permis de comparer les deux méthodes,
- Sur un échantillon, l'application des évaluations présentées sur un ensemble de méthodes d'évolution, nous a servi de décrire les architectures d'évolution existantes dans leurs deux aspects. Le constat de manque d'évolution prenant en charge le processus comme objet d'évolution ainsi que le manque d'approches favorisant l'émergence comme une mécanique opératoire sont les deux opportunités principales pour les prochaines approches dédiées pour l'évolution,
- Une Méta-classification, Nous avons aussi mené une étude empirique en examinant et analysant 119 méthodes de recherche. L'étude de la couverture de l'évolution architecturale a

permis de tirer un certain nombre de conclusions sur les opportunités, les forces et les faiblesses des approches existantes (nous y reviendrons dans la partie conclusion et perspectives). Cette méta-classification des approches de classification permet au même titre que l'échantillon d'approches de prévaloir les classes d'approches d'émergence et les capacités d'évaluation pour la qualité attendue.

En résumé, l'application du référentiel montre plus de souplesse aux intervenants pour l'évolution d'architecture et permet entre autre aux architectes de dresser une cartographie des efforts d'évolution à la base des dimensions proposées. Par conséquent, les résultats obtenus serviront à la prise de décisions et d'en tirer les opportunités, les forces et les faiblesses des approches existantes. Dans le prochain chapitre, nous concluons notre travail et nous dressons l'ensemble des perspectives que pose la modélisation de l'évaluation d'architecture à travers le

CHAPITRE 6

Conclusion et perspectives

6.1 Conclusion

Un système logiciel peut être assujetti à plusieurs évolutions tout le long de son cycle de vie. L'ultime et principal intérêt est de l'adapter aux nouvelles exigences technologiques, commerciales et concurrentielles que développe l'environnement ouvert dans lequel le système évolue. De facto, le processus d'évolution est une opération complexe consommant beaucoup de temps et parfois même très couteuse si elle n'est pas planifiée dans les premiers stages de développement du logiciel. Les architectures logicielles en ingénierie fournissent l'artefact le plus approprié pour faire face aux limites présentées par leurs habilités de traitement aux niveaux les plus abstraits. Ce qui a motivé plusieurs recherches d'évolution d'aborder la problématique à ce niveau d'abstraction puisqu'elles s'appuient sur deux visions différentes de l'architecture logicielle : La première est de guider la planification et la restructuration du système aux niveaux les plus élevés et la deuxième est de considérer l'architecture comme un artefact qui lui-même doit évoluer pour garder et assurer la cohérence des changements effectuées sur le système. Le travail de cette thèse a permis le développement d'un cadre conceptuel pour la description de l'évolution des architectures logicielles à base de services et à base de composants. Nous avons identifié puis capitalisé les concepts fondamentaux pour la compréhension, analyse et comparaison de l'évolution à l'échelle de l'architecture des systèmes. Nous avons aussi montré que les dimensions proposées sont orthogonales et que la précision des corrélations inter dimensions est un élément clé pour enrichir la description et la spécification des dimensions. Cela aura un apport positif sur l'amélioration de la compréhension de l'évolution aussi bien sur son aspect structurel et comportemental que sur son aspect qualitatif.

Ainsi, nous avons contribué à des thématiques différentes dans le domaine de l'évolution et modélisation de l'évolution des architectures logicielles et leurs descriptions. La proposition s'inscrit dans une stratégie plus large d'évaluation de la tendance des recherches existantes dans l'évolution de l'architecture logicielle par l'adoption de dimensions bien définies et structurées. Dans cet objectif, nous nous sommes focalisé sur la description et l'analyse du phénomène de

l'évolution par la considération d'une vue purement architecturale basée sur les niveaux de modélisation du système. Le référentiel de description d'une approche d'évolution est un outil macro-descriptif instauré pour permettre la compréhension du processus même pour les intervenants non spatialisés. De plus, il permet de capturer et positionner les meilleures techniques d'évolution dans le domaine de l'orienté service et l'orienté composant et ce à travers six dimensions. A cette fin, nous avons opté à répondre à un ensemble de questions (Quoi ? Pourquoi ? Quand ? Comment ? Où ? Qui ?) Pour pouvoir cerner les éléments de description de l'évolution des architectures aussi bien orientées composants que celles orientées services. Le référentiel est modélisé pour fournir une vue unifiée sur la description de toute évolution après et avant l'implémentation. Le référentiel a été exploré conformément a l'aspect structurel c'est-à-dire que nous sommes intéressés aux éléments composants l'architecture logicielle. Par la suite, nous avons achevé notre étude par un récapitulatif des différentes relations entre les dimensions pour montrer l'orthogonalité des dimensions. Ce qui conclu que ce référentiel est un outil permettant d'analyser l'existant et aussi éclairer les architectes sur les pistes qui ont été déjà abordées et ceux qui ne l'ont pas été encore. De plus, cet outil explore les approches dédiées dans le domaine pour permettre une cartographie du domaine et propose ainsi une multitude de classification qui peut être effectuée via une combinaison d'un ensemble de dimensions. La classification proposée pour exposer le caractère architectural de l'approche, s'appuie sur les niveaux architecturaux et le raisonnement élaboré pour faire évoluée l'architecture. Six classes émergent et sont réparties en trois approches orientées réductions et trois autres orientées émergences avec chacune d'elles regroupant des approches évoluant sur le même niveau d'abstraction ou qui changent entre niveaux d'abstraction du même niveau de modélisation ou sur plusieurs niveaux de modélisation.

Le référentiel permet donc d'offrir une assistance à travers des dimensions bien définies Sur le plan taxonomique, ce référentiel peut être instancié selon des objectifs fixés par les intervenants. La taxonomie permet donc de classifier une ou plusieurs approches et permet en conséquence de mettre en lumière les zones grises. Cela aidera à dessiner le design intérieur de l'espace de solution pour l'évolution logicielle. Il est démontré que la taxonomie est expressive, c'est-à-dire capable de représenter un large spectre de méthodes d'évolution architecturale, et efficace c'est-à-dire qu'elle facilite les similitudes et les comparaisons d'architectures impliquées dans l'évolution du logiciel.

Le deuxième aspect relève de l'étude de l'aspect qualitatif afin d'évaluer les critères qui sont impactés avec le processus d'évolution. Le modèle qualitatif proposé regroupe trois capacités de qualité de l'évolution. La première, capacité d'énonciation décrit les critères permettant d'exprimer la formalisation de l'évolution. La deuxième est la capacité d'expressivité dont les critères évaluent les propriétés intrinsèques à l'opération d'évolution vue par les architectes et la troisième comprenant les critères d'évaluation de la qualité.

Nous avons mené une étude empirique en examinant et en analysant 119 méthodes de recherche liées à l'évolution projetée par rapport aux trois paradigmes de classification existants dans la littérature. L'étude de la couverture de l'évolution de l'architecture logicielle a permis de tirer un certain nombre de conclusions sur les opportunités, les forces et les faiblesses des approches existantes. Les résultats obtenus confirment la cohérence du cadre proposé par rapport aux autres études et enquêtes existantes basées sur les mesures sélectionnées. Le

référentiel proposé présente l'avantage de comparer les différentes approches d'évolution architecturale en plus de la facilité de classification basée sur les dimensions proposées.

6.2 Perspectives

Nous avons cherché à travers ce référentiel d'assister les intervenants en leurs fournissant une vue plus claire est compréhensible aussi bien par ceux qui sont spécialisés ou non sur le processus d'évolution d'architecture logicielle. Cependant et compte tenu des nombreuses implications de nos conclusions, le référentiel proposé dévoile la nécessité d'entamer plusieurs travaux qui dressent notre perspective. Ces travaux s'insèrent dans une perspective à court terme et à long terme. Les travaux à court thermes représentent les extensions ou prolongation futures qui peuvent appuyer notre référentiel d'une part et d'autre part ceux qui sont à l'origine des limites qui circonscrivent notre travail. Les perspectives envisagées peuvent être formulées dans les six points suivants :

-Automatisation du référentiel

L'idée est d'implémenter un outil dont le modèle est donné par notre référentiel qui s'opère en background d'un des outils de conception et de modélisation d'architecture tel que Archstudio par exemple. A notre connaissance, peu d'outils de conception s'en soucient du problème de description des évolutions et de leurs propagations dans la phase de spécification et de réalisation. De plus, cette perspective permet de fournir aux architectes et/ou intervenants d'évaluer les décisions sur l'évolution à adopter avant même de les réaliser ce qui fournira un bon apport en termes de temps et de coûts pour une organisation. Une autre mise en œuvre peut être archivée dans l'éditeur, où davantage de fonctionnalités peuvent être fournies. Par exemple, dans l'atelier Ævol, une analyse de chemin a été fournie. Une telle fonctionnalité peut être utilisée pour aider l'outil à comparer les chemins d'évolution. De plus, la cohérence entre les points de vue devrait être prise en compte. L'outil permettra alors la capture et la réutilisation des connaissances en évolution architecturale et fournira aussi un ensemble de vues cohérentes à partir de différentes perspectives et aux différents niveaux hiérarchiques.

- Intégration du modèle dans COSA

La projection de notre référentiel vers COSA va permettre de préparer le terrain pour prendre en charge des artefacts spécifiques tel que les styles architecturaux. Effectivement, une fois les styles architecturaux seront décrits et implémentés par le biais d'un ADL, ils peuvent être aussi décrits pour fournir une description sur des styles d'évolution qui seront réutilisables. Il est important de rappeler que les styles d'évolutions peuvent s'opérer sur des niveaux de modélisation A_2, A_1 et A_0 (Méta-modélisation, modélisation et application) peuvent donc fournir des niveaux de description relatifs à chaque niveau. Des niveaux de causalité entre descriptions doivent être maintenus pour permettre aux intervenants spécialisés de bien gérer les propagations des impacts d'évolution entres les différents niveaux de description.

- Référentiel dirigé par la sémantique

Cette perspective stipule la considération de la variabilité du référentiel à la base de l'existence de plusieurs relations sémantiques entre dimensions. Ces relations présentent une

solution clé pour analyser la méthode d'évolution. Dans cette optique, nous pouvons considérer des propriétés sémantiques exprimant le degré d'analyse de la dimension ou des dimensions correspondantes. L'idée ici serait à notre avis d'offrir une description multi vues pouvant aller du niveau plus général au niveau le plus spécialisé c'est-à-dire à partir d'une vue macro à une vue micro.

- Référentiel orienté compossibilité

Nous avons considéré dans notre thèse la problématique de la description des architectures orientées. Nous avons considéré les aspects : structurel, comportemental et qualitatif. Nous estimons que l'intégration d'autres aspects décisionnels, comportementaux ne sera qu'un travail complémentaire pour le référentiel que nous avons proposé. De plus, la précision sur l'aspect de compossibilité des objets d'évolution fournira un apport bien consistant pour la description des architectures orientées services et à ceux qui sont orientés aspects.

- Raffinement et instanciation.

Le problème de raffinement du référentiel n'a pas été abordé dans ce travail. Nous envisagerons d'intégrer un processus de raffinement sélectif à plusieurs niveaux d'approfondissement c'est-à-dire permettre une description à plusieurs niveaux d'une sélection de dimensions. En effet chaque dimension peut être raffinée par l'expression des mêmes questions (Quoi ? Pourquoi ? Où? Comment ? Quand? Et Qui? du niveau supérieur). Il est ainsi primordial d'introduire des relations sémantiques entre les dimensions et sous dimensions pour maintenir la description à multi vues.

- Analyse du raisonnement

Afin de bien analyser une méthode d'évolution, il est souhaitable de s'assurer de la nouveauté du mécanisme employé par cette technique. Une multi-classification de la dite méthode par rapport à des méthodes représentatives des classes spécifiées permet d'analyser les écarts. En utilisant la puissance de l'IA, du cloud computing et du big-data, le développement d'un produit de processus peut être considérablement amélioré. Nous pouvons utiliser l'apprentissage automatique dans des scénarios tel que la conception de processus afin de synthétiser un processus. Cela peut aider le développeur en extrayant un grand nombre de connaissances et de pratiques de processus antérieurs, notamment en ce qui concerne les domaines, afin d'estimer les scénarios attendus pour le processus d'évolution.

Liste des publications de nos travaux

Publication

1. Gasmallah, N., Amirat, A., Oussalah, M., & H., Seridi, (2018, May). Developing an Evolution Software Architecture Framework Based on Six Dimensions. *International Journal of Simulation and Process Modelling (IJSPM), Vol. 14,* N.4, 13 p.

Conférences internationales avec comité de lecture

1. Gasmallah, N., Amirat, A., Oussalah, M., & Seridi, H. (2018, May). *Developing a Conceptual Framework for Software Evolution Methods via Architectural Metrics*. In Proceedings of the 6th IFIP TC 5 International Conference on Computational Intelligence and Its Applications, CIIA 2018 (pp. 140-149), Oran, Algeria. Springer International Publishing.

2. Gasmallah, N., Amirat, A., & Oussalah, M. (2016). *Evolution Taxonomy for Software Architecture Evolution*. In: Proceedings of the 11th Evaluation of Novel Approaches to Software Engineering ENASE 2016. ACM SIGSOFT (pp. 124-131).

3. Gasmallah, N., Oussalah, M., & Amirat, A. (2016, Juin). *Towards a classification for software architecture evolution studies*. In: Proceedings of Conférence francophone sur les Architectures Logicielles : CAL'2016, (pp. 40-51), Besançon, France.

4. Amirat, A., Bouchouk, A., Yeslem, M. O., & Gasmallah, N. (2012, September). *Refactor Software architecture using graph transformation approach*. In: 2nd International Conference on Innovative Computing Technology (INTECH'2012). IEEE (pp. 117-122).

5. Amirat, A., Menasria, A., & Gasmallah, N. (2011). *Evolution Framework for Software Architecture Using Graph Transformation Approach*. In The 2012 International Arab Conference on Information Technology: ACIT'2011 (pp. 96-108), Riyadh, K-S-A.

Journées doctorales nationales

1. Gasmallah, N., Amirat, A., Bouchrika, I. & Layouni, Z. (2013, December). *Transparent Achitecture for Handling Binary.Attachments for Data-Centric Web Services*. In: 3ème Journées Doctorales en Informatique JDI 2013 (Guelma-Algérie).

Bibliographie

A

- Adekola, O. D., Idowu, S. A., & Adebayo, A. O. (2016). Component Based Software Engineering in Student Management System Domain: A Development for Reuse Approach. *International Journal of Software Engineering and Its Applications, 10*(9), 149-162.
- Adel, A., & Abbdellah, B. (2010). *Semantic mapping of ADLs into MDA platforms using a meta-ontology*. International Conference on Machine and Web Intelligence (ICMWI'10) (pp. 426-433). IEEE.
- Ahmad, M., Belloir, N., & Bruel, J.-M. (2015). Modeling and verification of functional and non-functional requirements of ambient self-adaptive systems. *Journal of Systems and Software, 107*, 50-70.
- Ahmad, A., Jamshidi, P., & Pahl, C. (2014). Classification and comparison of architecture evolution reuse knowledge—a systematic review. *Journal of Software: Evolution and Process, 26*(7), 654-691.
- Al-Badareen, A. B., Selamat, M. H., Din, J., Jabar, M. A., & Turaev, S. (2011). Software quality evaluation: User's view. *international journal of applied mathematics and informatics, 5*(3), 200-207.
- Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., & Meedeniya, I. (2013). Software architecture optimization methods: A systematic literature review. *IEEE Transactions on software engineering, 39*(5), 658-683.
- Al-Naeem, T., Gorton, I., Babar, M. A., Rabhi, F., & Benatallah, B. (2005). *A quality-driven systematic approach for architecting distributed software applications*. Paper presented at the 27th Proceedings of the international conference on Software engineering (pp. 244-253). ACM.
- Alti, A. (2011). *Coexistence de la modélisation à base d'objets et de la modélisation à base de composants architecturaux pour la description de l'architecture logicielle,* Sétif (Algérie), 2011, 221 p. Disponible sur *https://www.researchgate.net/publication/328687375_Coexistence_de_la_modelisation_a_base_d%27objets_et_de_la_modelisation_a_base_de_composants_architecturaux_pour_la_description_de_l%27architecture_logicielle*.
- Alti, A., & Smeda, A. (2006). *Un profil UML 2.0 pour l'architecture logicielle COSA*. Paper presented at the Workshop Objets, Composants et Modéles dans l'Ingénierie des Systèmes d'Information (OCM-SI'06), In conjunction with INFORSID'06 (pp. 62-73).
- Alves, D. F., de Campos, R., & Souza, F. B. (2016). *Sustainable Development Within Enterprise Architecture*. Paper presented at the IFIP International Conference on Advances in Production Management Systems (pp. 552-559). Springer, Cham.
- Amirat, A., Bouchouk, A., Yeslem, M. O., & Gasmallah, N. (2012, September). *Refactor Software architecture using graph transformation approach*. In 2nd International Conference on Innovative Computing Technology (INTECH'2012) (pp. 117-122). IEEE.
- Amirat, A., Menasria, A., & Gasmallah, N. (2011). *Evolution Framework for Software Architecture Using Graph Transformation Approach*. Paper presented at The 2012 International Arab Conference on Information Technology (ACIT'2012).
- Amirat, A., & Oussalah, M. (2009, May). *C3: A metamodel for architecture description language based on first-order connector types*. Paper presented at the 11th International Conference on Enterprise Information Systems (ICEIS 2009) (pp. 76-81).

- Ampatzoglou, A., Kritikos, A., Kakarontzas, G., & Stamelos, I. (2011). An empirical investigation on the reusability of design patterns and software packages. *Journal of Systems and Software, 84*(12), 2265-2283.
- Anwar, A., Dkaki, T., Ebersold, S., Coulette, B., & Nassar, M. (2011). *A formal approach to model composition applied to VUML*. Paper presented at the 16th International Conference on Engineering of Complex Computer Systems (ICECCS) (pp. 188-197). IEEE.
- Aoyama, M. (2002). *Metrics and analysis of software architecture evolution with discontinuity*. In Proceedings of the International Workshop on Principles of Software Evolution (pp. 103-107). ACM.
- Aponte, M.-V., & Simonot, M. (2008). *Une approche formelle de la reconfiguration dynamique*: Technical Report 1590, CNAM-CEDRIC.
- Assunção, W. K. G., & Vergilio, S. R. (2014). *Feature location for software product line migration: a mapping study*. In Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2 (pp. 52-59). ACM.

B

- Babar, M. A., & Gorton, I. (2007). *A tool for managing software architecture knowledge*. Paper presented at the 2nd Workshop on Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent, 2007. SHARK/ADI'07: ICSE Workshops 2007 (p. 11). IEEE.
- Bagheri, E., & Gasevic, D. (2011). Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal, 19*(3), 579-612.
- Bahl, D., & Wason, R. (2012). Autonomic Computing: Further Maturation in IT Industry. *International Journal of Science and Research, 1*(3), 50-58.
- Bainbridge, L. (1983). *Ironies of automation Analysis, Design and Evaluation of Man–Machine Systems*. In Proceedings of IFAC/IFIP/IFORS/IEA Conference (pp. 129-135). Elsevier.
- Bálek, D., & Plášil, F. (2001). *Software connectors and their role in component deployment*. In IFIP International Conference on Distributed Applications and Interoperable Systems (pp. 69-84). Springer, Boston, MA.
- Barais, O., & Duchien, L. (2005). *Safarchie studio: Argouml extensions to build safe architectures*. In IFIP World Computer Congress , TC 2 (pp. 85-100). Springer, Boston, MA.
- Barais, O., Cariou, E., Duchien, L., Pessemier, N., & Seinturier, L. (2004). *Transat: A framework for the specification of software architecture evolution*. Issues on Coordination and Adaptation Techniques, (pp. 31-38).
- Baranov, E. (2017). A Semantic Framework for Architecture Modelling, sous la direction de . Sifakis, Suisse, EPFL, 2017, 161 p. Disponible sur *https://infoscience.epfl.ch /record/224049/files/EPFL_TH7325.pdf*
- Baresi, L., & Quinton, C. (2015). *Dynamically evolving the structural variability of dynamic software product lines*. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (pp. 57-63). IEEE Press.
- Barnes, A., Hill, G., & Busateri, J. (2015). *Platform for generating composite applications*: U.S. Patent No 9,032,312, 16 May 2015.
- Barnes, J. M., & Garlan, D. (2013). *Challenges in developing a software architecture evolution tool as a plug-in*. Paper presented at the 3rd International Workshop on Developing Tools as Plug-ins (TOPI) (pp. 13-18). IEEE.
- Barnes, J. M., Garlan, D., & Schmerl, B. (2014). Evolution styles: foundations and models for software architecture evolution. *Software & Systems Modeling, 13*(2), 649-678.
- Barnes, J. M., Pandey, A., & Garlan, D. (2013). *Automated planning for software architecture evolution*. Paper presented at the 28th International Conference on Automated Software Engineering (pp. 213-223). IEEE/ACM.
- Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*: Addison-Wesley Professional.

- Basson, H. (1998). *An integrated model for impact analysis of software change*. Software Quality Management VI (pp. 260-269). Springer, London.
- Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., et al. (1999). *PuLSE: a methodology to develop software product lines*. In Proceedings of the 5th ACM SIGSOFT symposium on Software reusability (SSR'99).
- Belady, L. A., & Lehman, M. M. (1976). A model of large program development. *IBM Systems journal, 15*(3), 225-252.
- Belkhatir, R., Oussalah, M. C., & Viguier, A. (2012). *SOAQE-Service Oriented Architecture Quality Evaluation*. Paper presented at the 7th ENASE-International Conference on Evaluation of Novel Approaches to Software Engineering (pp. 199-202), Wroclaw, Poland. SciTePress.
- Bengtsson, P., & Bosch, J. (1998). *Scenario-based software architecture reengineering*. Paper presented at the 5th International Conference on Software Reuse Cat. No. 98TB100203, (pp. 308-317). IEEE.
- Bengtsson, P., Lassing, N., Bosch, J., & van Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software, 69*(1-2), 129-147.
- Bennett, K. H., & Rajlich, V. T. (2000). *Software maintenance and evolution: a roadmap*. In Proceedings of the Conference on the Future of Software Engineering (pp. 73-87). ACM.
- Berger, T., Nair, D., Rublack, R., Atlee, J. M., Czarnecki, K., & Wąsowski, A. (2014). *Three cases of feature-based variability modeling in industry*. Paper presented at the International Conference on Model Driven Engineering Languages and Systems (pp. 302-319). Springer, Cham.
- Berman, P. (1978). *The study of macro and micro implementation of social policy*. Santa Monica, Calif.: Rand.
- Bézivin, J. (2003). *La transformation de modèles*. INRIA-ATLAS & Université de Nantes, 13.
- Bigot, J., & Pérez, C. (2010). *Enabling connectors in hierarchical component models*. Research Report RR-7204. INRIA-00456608v2.
- Boehm, B., & Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed, Portable Documents*: Addison-Wesley Professional.
- Bohner, S. A. (2002). *Extending software change impact analysis into COTS components*. In 27th NASA Goddard Software Engineering Workshop (pp. 175-182). IEEE.
- Booch, G. (2006). *Object oriented analysis & design with application*: Pearson Education India.
- Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*: Pearson Education.
- Boudaa, B., Hammoudi, S., Mebarki, L. A., Bouguessa, A., & Chikh, M. A. (2017). An aspect-oriented model-driven approach for building adaptable context-aware service-based applications. *Science of Computer Programming, 136*, 17-42.
- Bouwers, E., & Van Deursen, A. (2010). A lightweight sanity check for implemented architectures. *IEEE software, 27*(4), 44-50.
- Bowman, I. T., Holt, R. C., & Brewster, N. V. (1999). *Linux as a case study: Its extracted software architecture*. In Proceedings of the 21st international conference on Software Engineering Cat. No. 99CB37002 (pp. 555-563). IEEE.
- Bradbury, J., Cordy, J., Dingel, J., & Wermelinger, M. (2004). *A classification of formal specifications for dynamic software architectures*. In International Workshop on Self-Managed Systems.
- Breivold, H. P., Crnkovic, I., & Larsson, M. (2012). A systematic review of software architecture evolution research. *Information and Software Technology, 54*(1), 16-40.
- Brooks Jr, F. P. (1974). The mythical man-month. *Datamation, 20*(12), 44-52.
- Brooks, R. A. (1989). A robot that walks; emergent behaviors from a carefully evolved network. *Neural computation, 1*(2), 253-262.
- Brown, A., Johnston, S., & Kelly, K. (2002). Using service-oriented architecture and component-based development to build web service applications. *Rational Software Corporation, 6*, 1-16.

- Brown, J., Knepley, M. G., & Smith, B. F. (2015). *Run-time extensibility and librarization of simulation software. Computing in Science & Engineering, 17*(1), 38-45.
- Bruneliere, H., Burger, E., Cabot, J., & Wimmer, M. (2017). A feature-based survey of model view approaches. *Software & Systems Modeling, 18*(3), 1931-1952.
- Buckley, J. a. M., Tom and Zenger, Matthias and Rashid, Awais and Kniesel, Günter. (2005). Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice, 17*(5), 309-332.
- Bures, T., Hnetynka, P., & Plasil, F. (2006, August). *Sofa 2.0: Balancing advanced features in a hierarchical component model.* In 4th International Conference on Software Engineering Research, Management and Applications (pp. 40-48). IEEE.
- Bürger, J., Strüber, D., Gärtner, S., Ruhroth, T., Jürjens, J., & Schneider, K. (2018). A framework for semi-automated co-evolution of security knowledge and system models. *Journal of Systems and Software, 139*, 142-160.

C

- Casais, E. (1992). *An incremental class reorganization approach*. In European Conference on Object-Oriented Programming (pp. 114-132). Springer, Berlin, Heidelberg.
- Cavano, J. P., & McCall, J. A. (1978). *A framework for the measurement of software quality*. In ACM SIGMETRICS Performance Evaluation Review. ACM, Vol. 7, No. 3-4, 133-139.
- Cervantes, H., & Hall, R. S. (2004). *Autonomous adaptation to dynamic availability using a service-oriented component model.* In Proceedings of the 26th International Conference on Software Engineering (pp. 614-623). IEEE Computer Society.
- Chaki, S., Diaz-Pace, A., Garlan, D., Gurfinkel, A., & Ozkaya, I. (2009). *Towards engineered architecture evolution*. Paper presented at the Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering (pp. 1-6). IEEE Computer Society.
- Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., & Tan, W. G. (2001). Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice, 13*(1), 3-30.
- Chefrour, D., & André, F. (2002). *Aceel: modèle de composants auto-adaptatifs*. Systèmes à composants adaptables et extensibles, Grenoble, France.
- Christensen, H. B., & Hansen, K. M. (2010). An empirical investigation of architectural prototyping. *Journal of Systems and Software, 83*(1), 133-142.
- Chung, L., Cooper, K., & Yi, A. (2003). Developing adaptable software architectures using design patterns: an NFR approach. *Computer Standards & Interfaces, 25*(3), 253-260.
- Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (2012). *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media.
- Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008). *Automating co-evolution in model-driven engineering*. In 12th International IEEE Enterprise Distributed Object Computing Conference, EDOC'08 (pp. 222-231). IEEE.
- Clements, P. C. (2002). *Software architecture in practice*. Diss. Software Engineering Institute.
- Clements, P., & Bass, L. (2010). *Business goals as architectural knowledge*. In Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge (pp. 9-12). ACM.
- Clements, P., & Northrop, L. (2002). *Software product lines: practices and patterns*. Vol. 3. Addison-Wesley Reading.
- Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., et al. (2002). *Documenting software architectures: views and beyond.* Pearson Education.
- Cook, S., Harrison, R., Lehman, M. M., & Wernick, P. (2006). Evolution in software systems: foundations of the SPE classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice, 18*(1), 1-35.

- Crnkovic, I., Chaudron, M., & Larsson, S. (2006). *Component-based development process and component lifecycle*. Paper presented at the International Conference on Software Engineering Advances ICSEA'06 (p. 44). IEEE.

D

- De Leenheer, P., & Mens, T. (2008). *Ontology evolution*. In Ontology management (pp. 131-176). Springer, Boston, MA.
- De Wolf, T., & Holvoet, T. (2004). *Emergence versus self-organisation: Different concepts but promising when combined*. In : International workshop on engineering self-organising applications, (pp. 1-15). Springer, Berlin, Heidelberg.
- Deacon, J. (2009). *Model-view-controller (mvc) architecture*. Online][Citado em: 10 de março de 2006]. *http://www. jdl. co. uk/briefings/MVC.pdf*.
- Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A., & Guelfi, N. (2006). *Dependable self-organising software architectures-An approach for self-managing systems*. Technical report, BBKCS 05-06. School of Computer Science and Information Systems, Birkbeck College.
- Dijkman, R. M., Quartel, D. A., & van Sinderen, M. J. (2008). Consistency in multi-viewpoint design of enterprise information systems. *Information and Software Technology, 50*(7-8), 737-752.
- Dijkman, R., & Dumas, M. (2004). Service-oriented design: A multi-viewpoint approach. *International journal of cooperative information systems, 13*(04), 337-368.
- Dorfman, M., & Thayer, R. H. (1990). *Standards, guidelines and examples on system and software requirements engineering*. IEEE Computer Society Press Tutorial, Los Alamitos: IEEE Computer Society Press, edited by Dorfman, Merlin; Thayer, Richard H.
- Ducasse, S., & Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on software engineering, 35*(4), 573-591.

E

- Engel, A., & Browning, T. R. (2008). Designing systems for adaptability by means of architecture options. *Systems Engineering, 11*(2), 125-146.
- Engels, G., & Heckel, R. (2000). *Graph transformation as a conceptual and formal framework for system modeling and model evolution*. In International Colloquium on Automata, Languages, and Programming (pp. 127-150). Springer, Berlin, Heidelberg.
- Erl, T. (2008). *Soa: principles of service design*. Vol. 1. Prentice Hall Upper Saddle River.

F

- Faust, D., & Verhoef, C. (2003). Software product line migration and deployment. *Software: Practice and Experience, 33*(10), 933-955.
- Fenton, N., & Bieman, J. (2014). *Software metrics: a rigorous and practical approach*: CRC press.
- Fielding, R. T., & Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures*. Vol. 7, University of California, Irvine Irvine, USA.
- Fielding, R. T., Taylor, R. N., Erenkrantz, J. R., Gorlick, M. M., Whitehead, J., Khare, R., et al. (2017). *Reflections on the REST architectural style and principled design of the modern web architecture (impact paper award)*. In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (pp. 4-14). ACM.
- Finkelstein, A., & Sommerville, I. (1996). The viewpoints FAQ. *BCS/IEE Software Engineering Journal, 11*(1), 2-4.
- Frakes, W., & Terry, C. (1996). Software reuse: metrics and models. *ACM Computing Surveys (CSUR), 28*(2), 415-435.
- Frankel, D. S., Harmon, P., Mukerji, J., Odell, J., Owen, M., Rivitt, P., et al. (2003). The Zachman framework and the OMG's model driven architecture. *Business Process Trends, 14*(2003), 2003.
- Fricke, E., Gebhard, B., Negele, H., & Igenbergs, E. (2000). Coping with changes: causes, findings, and strategies. *Systems Engineering, 3*(4), 169-179.

- Friedenthal, S., Moore, A., & Steiner, R. (2008). *OMG Systems Modeling Language (OMG SysML™) Tutorial*. Paper presented at the INCOSE International Symposium Vol 9 (pp. 65-67).

G

- Gaeremynck, Y., Bergman, L. D., & Lau, T. (2003). *MORE for less: model recovery from visual interfaces for multi-device application design*. In Proceedings of the 8th international conference on Intelligent user interfaces (pp. 69-76). ACM.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
- Garcés, K., Casallas, R., Álvarez, C., Sandoval, E., Salamanca, A., Viera, F., et al. (2018). White-box modernization of legacy applications: the oracle forms case study. *Computer Standards & Interfaces, 57*, 110-122.
- Garcés, K., Jouault, F., Cointe, P., & Bézivin, J. (2009). *Managing model adaptation by precise detection of metamodel changes*. In European Conference on Model Driven Architecture-Foundations and Applications (pp. 34-49). Springer, Berlin, Heidelberg.
- Gardner, T., Griffin, C., Koehler, J., & Hauser, R. (2003). *A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard*. In MetaModelling for MDA Workshop, 13, p. 41.
- Garlan, D. (2000). *Software architecture: a roadmap*. In Proceedings of the Conference on the Future of Software Engineering (pp. 91-101). ACM.
- Garlan, D., & Schmerl, B. (2009). *Ævol: A tool for defining and planning architecture evolution*. In Proceedings of the 31st International Conference on Software Engineering (pp.591-594). IEEE.
- Garlan, D., & Shaw, M. (1993). *An introduction to software architecture Advances in software engineering and knowledge engineering*. World Scientific, ed. 1.1993, (pp. 1-39).
- Garlan, D., Barnes, J. M., Schmerl, B., & Celiku, O. (2009). *Evolution styles: Foundations and tool support for software architecture evolution*. In 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (pp. 131-140). IEEE.
- Garlan, D., Monroe, R., & Wile, D. (1997). *Acme: An architecture description interchange language*. In Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (p. 7). iBm press.
- Gasmallah, N., Amirat, A., Oussalah, M., & H., Seridi, (2018, May). Developing an Evolution Software Architecture Framework Based on Six Dimensions. *International Journal of Simulation and Process Modelling (IJSPM), Vol. 14,* N.4, 13 p.
- Gasmallah, N., Amirat, A., Oussalah, M., & Seridi, H. (2018, May). *Developing a Conceptual Framework for Software Evolution Methods via Architectural Metrics*. In Proceedings of the 6th IFIP TC 5 International Conference on Computational Intelligence and Its Applications, CIIA 2018 (pp. 140-149), Oran, Algeria. Springer International Publishing.
- Gasmallah, N., Amirat, A., & Oussalah, M. (2016). *Evolution Taxonomy for Software Architecture Evolution*. In Proceedings of the 11th Evaluation of Novel Approaches to Software Engineering ENASE 2016 (pp. 124-131). ACM SIGSOFT.
- Gasmallah, N., Oussalah, M., & Amirat, A. (2016, Juin). *Towards a classification for software architecture evolution studies*. In Proceedings of Conférence francophone sur les Architectures Logicielles : CAL'2016, (pp. 40-51), Besançon, France.
- Giesecke, S., Hasselbring, W., & Riebisch, M. (2007). Classifying architectural constraints as a basis for software quality assessment. *Advanced Engineering Informatics, 21*(2), 169-179.
- Goknil, A., Kurtev, I., van den Berg, K., & Spijkerman, W. (2014). Change impact analysis for requirements: A metamodeling approach. *Information and Software Technology, 56*(8), 950-972.

- Gottlob, G., Schrefl, M., & Röck, B. (1996). Extending object-oriented systems with roles. *ACM Transactions on Information Systems (TOIS), 14*(3), 268-296.
- Govin, B., Anquetil, N., Etien, A., Du Sorbier, A. M., & Ducasse, S. (2015). *Reverse engineering tool requirements for real time embedded systems*. Paper presented at the SATToSE'15.
- Grady, R. B. (1992). *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc.
- Graiet, M., Bhiri, M. T., Dammak, F., & Giraudin, J.-P. (2006). *Adaptation d'UML2. 0 à l'ADL Wright*. Paper presented at the CAL (pp. 83-100).

H

- Haken, H., & Jumarie, G. (2006). *A macroscopic approach to complex system*. Springer.
- Hassan, A., & Oussalah, M. (2016). *Meta-evolution style for software architecture evolution*. In International Conference on Current Trends in Theory and Practice of Informatics (pp. 478-489). Springer, Berlin, Heidelberg.
- Hassan, A., Queudet, A., & Oussalah, M. (2016). *Evolution Style: Framework for Dynamic Evolution of Real-Time Software Architecture*. In European Conference on Software Architecture (pp. 166-174). Springer, Cham.
- Heineman, G. T., & Councill, W. T. (2001). *Component-based software engineering*. Putting the pieces together, addison-westley, p. 5. Springer.
- Herraiz, I., Rodriguez, D., Robles, G., & Gonzalez-Barahona, J. M. (2013). The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR), 46*(2), 28.
- Herrmannsdoerfer, M., Benz, S., & Juergens, E. (2009). *COPE-automating coupled evolution of metamodels and models*. In European Conference on Object-Oriented Programming (pp. 52-76). Springer, Berlin, Heidelberg.
- Hilliard, R. (1999). *Views and viewpoints in software systems architecture*. Paper presented at the Position paper from the First Working IFIP Conference on Software Architecture, San Antonio.
- Hnětynka, P., & Plášil, F. (2006). *Dynamic reconfiguration and access to services in hierarchical component models*. In International Symposium on Component-Based Software Engineering (pp. 352-359). Springer, Berlin, Heidelberg.
- Hornford, D., Paider, T., Forde, C., Josey, A., Doherty, G., & Fox, C. (2011). *TOGAF Version 9.1, Enterprise Edition*. Standard G116, the Open Group.

I

- Ilic, A., Pratas, F., Trancoso, P., & Sousa, L. (2011). *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*, chapter High-Performance Computing on Heterogeneous Systems: Database Queries on CPU and GPU (pp. 202-222). IOS Press.
- Iso. (2009). Document: ISO/TC 176/SC 2/N 544R3. Retrieved 8th Sep 2017, 2017, from http://www.iso.org/iso/fr/ 04 concept and use of the process approach for management systems.pdf
- Iso, I. (2001). Iec 9126-1: *Software engineering-product quality-part 1: Quality model. Geneva, Switzerland*. International Organization for Standardization, *21*.
- Ito, E., & Fujimoto, T. (2015). *Proposal of Multi View Programming*. Paper presented at the The 3rd International Conference on E-Technologies and Business on the Web: EBW2015 (p. 187).

J

- Jäkel, F., Schölkopf, B., & Wichmann, F. A. (2008). Generalization and similarity in exemplar models of categorization: Insights from machine learning. *Psychonomic Bulletin & Review, 15*(2), 256-271.

- Jamshidi, P., Ghafari, M., Ahmad, A., & Pahl, C. (2013). *A framework for classifying and comparing architecture-centric software evolution research*. In 2013 17th European Conference on Software Maintenance and Reengineering (pp. 305-314). IEEE.
- Jansen, A., & Bosch, J. (2004). *Evaluation of tool support for architectural evolution*. In Proceedings of the 19th IEEE international conference on Automated software engineering (pp. 375-378). IEEE Computer Society.
- Jansen, A., & Bosch, J. (2005). *Software architecture as a set of architectural design decisions*. In 5th Working IEEE/IFIP Conference on Software Architecture: WICSA'05 (pp. 109-120). IEEE.
- Jansen, A., Van Der Ven, J., Avgeriou, P., & Hammer, D. K. (2007). *Tool support for architectural decisions*. In 2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07) (pp. 4-4). IEEE.
- Jazayeri, M. (2005). *Species evolve, individuals age*. In 8th International Workshop on Principles of Software Evolution: IWPSE'05 (pp. 3-9). IEEE.
- Jiang, M., & Willey, A. (2005). *Architecting systems with components and services*. In IRI-International Conference on Information Reuse and Integration, Conf (pp. 259-264). IEEE.
- Jouault, F. (2006). *Contribution à l'étude des langages de transformation de modèles*. Sous la direction de Consel C., Nantes, UFR Nantes, 190 p.

K

- Kaur, N., Harrison, R., & West, A. A. (2015). *A service-oriented approach to embedded component-based manufacturing automation*. In 2015 IEEE International Conference on Industrial Technology (ICIT) (pp. 2964-2969). IEEE.
- Kazman, R., Bass, L., Abowd, G., & Webb, M. (1994). SAAM: *A method for analyzing the properties of software architectures*. In Proceedings of 16th International Conference on Software Engineering (pp. 81-90). IEEE.
- Keller, R. K., Schauer, R., Robitaille, S., & Pagé, P. (1999). *Pattern-based reverse-engineering of design components*. In Proceedings of the 1999 International Conference on Software Engineering (Cat. No. 99CB37002) (pp. 226-235). IEEE.
- Kemerer, C. F., & Slaughter, S. (1999). *An empirical approach to studying software evolution*. *IEEE Transactions on software engineering, 25*(4), 493-509.
- Kennerley, M., & Neely, A. (2002). A framework of the factors affecting the evolution of performance measurement systems. *International journal of operations & production management, 22*(11), 1222-1245.
- Kessentini, W., Sahraoui, H., & Wimmer, M. (2019). Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology, 106*, 49-67.
- Kheir, A., Naja, H., Oussalah, M. C., & Tout, K. (2013). *Overview of an approach describing multi-views/multi-abstraction levels software architecture*. In Evaluation of Novel Approaches to Software Engineering (p. 140).
- Khomh, F., Di Penta, M., & Gueheneuc, Y.-G. (2009). *An exploratory study of the impact of code smells on software change-proneness*. In 16th Working Conference on Reverse Engineering (pp. 75-84). IEEE.
- Kiczales, G Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997, June). *Aspect-oriented programming*. In European conference on object-oriented programming (pp. 220-242). Springer, Berlin, Heidelberg.
- Kleber, S., Maile, L., & Kargl, F. (2018). Survey of Protocol Reverse Engineering Algorithms: Decomposition of Tools for Static Traffic Analysis. *IEEE Communications Surveys & Tutorials, 21*(1), 526-561.
- Knapp, A., & Mossakowski, T. (2018). *Multi-view Consistency in UML: A Survey Graph Transformation, Specifications, and Nets* (pp. 37-60). Springer, Cham.
- Knieke, C., Körner, M., Rausch, A., Schindler, M., Strasser, A., & Vogel, M. (2017). *A Holistic Approach for Managed Evolution of Automotive Software Product Line Architectures*. Special Track:

Managed Adaptive Automotive Product Line Development (MAAPL), along with ADAPTIVE, 43-52.
- Kniesel, G., Costanza, P., & Austermann, M. (2001). *Jmangler-a framework for load-time transformation of java class files*. In Proceedings 1st International Workshop on Source Code Analysis and Manipulation (pp. 98-108). IEEE.
- Krafzig, D., Banke, K., & Slama, D. (2005). *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional.
- Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *IEEE software, 12*(6), 42-50.
- Kruchten, P., Capilla, R., & Dueñas, J. C. (2009). The decision view's role in software architecture practice. *IEEE software, 26*(2), 36-42.
- Kuhrmann, M., Ternité, T., Friedrich, J., Rausch, A., & Broy, M. (2016). Flexible software process lines in practice: A metamodel-based approach to effectively construct and manage families of software process models. *Journal of Systems and Software, 121*, 49-71.

L

- Lane, S., Gu, Q., Lago, P., & Richardson, I. (2010). *Adaptation of service based applications: a maintenance process*. Lero, the Irish Software Engineering Research Centre, Limerick, Ireland, Tech. Rep. Lero-TR-2010-08.
- Lassing, N., Bengtsson, P., Van Vliet, H., & Bosch, J. (2002). Experiences with ALMA: architecture-level modifiability analysis. *Journal of Systems and Software, 61*(1), 47-57.
- Le Goaer, O. (2009). *Styles d'évolution dans les architectures logicielles*, sous la direction de Oussalah M., Nantes, Ecole Centrale de Nantes, 173 p.
- Le Goaer, O., Tamzalit, D., & Oussalah, M. (2010). *Evolution styles to capitalize evolution expertise within software architectures*. In SEKE 2010 (pp. to-appear).
- Lehman, M. M. (1978). *Programs, cities, students—limits to growth?* Programming Methodology (pp. 42-69). Springer, New York, NY.
- Lehman, M. M. (1980). *Programs, life cycles, and laws of software evolution*. Proceedings of the IEEE, 68(9), 1060-1076.
- Lehman, M. M. (1991). Software engineering, the software process and their support. *Software Engineering Journal, 6*(5), 243-258.
- Lehman, M. M., & Belady, L. A. (1985). *Program evolution: processes of software change*: Academic Press Professional, Inc.
- Lehman, M. M., & Ramil, J. F. (2002). *Software uncertainty Soft-Ware 2002: Computing in an Imperfect World* (pp. 174-190): Springer, Berlin, Heidelberg.
- Lehman, M., & Parr, F. (1976). *Program evolution and its impact on software engineering*. In Proceedings of the 2nd international conference on Software engineering (pp. 350-357). IEEE Computer Society Press.
- Leist, S., & Zellner, G. (2006). *Evaluation of current architecture frameworks*. In Proceedings of the 2006 ACM symposium on Applied computing (pp. 1546-1553). ACM.
- Lieberherr, K. (1995). *Workshop on adaptable and adaptive software*. ACM SIGPLAN OOPS Messenger, 6(4), 149-154.
- Lientz, B. P., & Swanson, E. B. (1980). *Software maintenance management*. Addison-Wesley.
- Ling Sim, K., & Chye Koh, H. (2001). Balanced scorecard: a rising trend in strategic performance measurement. *Measuring business excellence, 5*(2), 18-27.
- Lüer, C., & Van Der Hoek, A. (2002). *Composition environments for deployable software components*. Department of Information and Computer Science, University of California, Irvine.
- Lung, C.-H., Bot, S., Kalaichelvan, K., & Kazman, R. (1997). *An approach to software architecture analysis for evolution and reusability*. In Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (p. 15). IBM Press.

- Lutz, M., Inglés-Romero, J. F., Stampfer, D., Lotz, A., Vicente-Chicote, C., & Schlegel, C. (2019). *Managing Variability as a Means to Promote Composability: A Robotics Perspective New Perspectives* on Information Systems Modeling and Design (pp. 274-295). IGI Global.

M

- Madhavji, N. H., Fernandez-Ramil, J., & Perry, D. (2006). *Software evolution and feedback: Theory and practice*: John Wiley & Sons.
- MacCormack, A., & Sturtevant, D. J. (2016). Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software, 120*, 170-182.
- Maillard, S., Smeda, A., & Oussalah, M. (2007). *Cosa: An architectural description meta-model.* Paper presented at the ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies (pp. 445-448). INSTICC Press.
- Maisonnasse, L., Berrut, C., & Chevallet, J.-P. (2009). L'expressivité des modèles de recherche d'informations précises. *Document numérique, 12*(1), 107-128.
- Mall, R. (2018). *Fundamentals of software engineering*: PHI Learning Pvt. Ltd.
- Mary, S., & David, G. (1996*). Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.
- Matinlassi, M. (2004). *Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA*. In Proceedings of the 26th International Conference on Software Engineering (pp. 127-136). IEEE Computer Society.
- Matland, R. E. (1995). Synthesizing the implementation literature: The ambiguity-conflict model of policy implementation. *Journal of public administration research and theory, 5*(2), 145-174.
- Matougui, S., & Beugnard, A. (2005). *Two ways of implementing software connections among distributed components.* In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (pp. 997-1014). Springer, Berlin, Heidelberg.
- May, N. (2005). *A survey of software architecture viewpoint models*. In Proceedings of the 6th Australasian Workshop on Software and System Architectures (pp. 13-24).
- McCall, J. A. (1994). *Quality factors*. Encyclopedia of Software Engineering vol. 2, J. Marciniak editor.
- Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on software engineering, 26*(1), 70-93.
- Medvidovic, N., Rosenblum, D. S., & Taylor, R. N. (1999). *A language and environment for architecture-based software development and evolution.* In Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002) (pp. 44-53). IEEE.
- Mehta, N. R., Medvidovic, N., & Phadke, S. (2000). *Towards a taxonomy of software connectors*. In Proceedings of the 22nd international conference on Software engineering, (pp. 178-187). ACM.
- Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science, 152*, 125-142.
- Mens, T., Buckley, J., Zenger, M., & Rashid, A. (2003). *Towards a taxonomy of software evolution*. In Proceedings of the International Workshop on Unanticipated Software Evolution (No. CONF).
- Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., & Jazayeri, M. (2005). *Challenges in software evolution*. In 8th International Workshop on Principles of Software Evolution: IWPSE'05 (pp. 13-22). IEEE.
- Mili, H., Sahraoui, H., Lounis, H., Mcheick, H., & Elkharraz, A. (2006). *Concerned about separation.* In International Conference on Fundamental Approaches to Software Engineering (pp. 247-261). Springer, Berlin, Heidelberg.

- Mirakhorli, M. (2015). *Software architecture reconstruction: Why? What? How?* In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER) (pp. 595-595). IEEE.
- Mullery, G. P. (1979). *CORE-a method for controlled requirement specification*. In Proceedings of the 4th international conference on Software engineering (pp. 126-135). IEEE Press.

N

- Nassar, M. (2003). *VUML: a Viewpoint oriented UML Extension*. In 18th IEEE International Conference on Automated Software Engineering (pp. 373-376). IEEE.
- Nassar, M. (2005). *Analyse/conception par points de vue: le profil VUML,* Informatique et Télécommunication, sous la direction de Coulette B., Toulouse, Institut National Polytechnique, 242 p.
- Nassar, M., Anwar, A., Ebersold, S., Elasri, B., Coulette, B., & Kriouile, A. (2009). *Code generation in VUML profile: A model driven approach*. In 2009 IEEE/ACS International Conference on Computer Systems and Applications (pp. 412-419). IEEE.

O

- O'Brien, L., Merson, P., & Bass, L. (2007). *Quality attributes for service-oriented architectures*. In Proceedings of the international Workshop on Systems Development in SOA Environments (p. 3). IEEE Computer Society.
- O'Reilly, C., Morrow, P., & Bustard, D. (2003). *Lightweight prevention of architectural erosion*. In 6th International Workshop on Principles of Software Evolution (pp. 59-64). IEEE.
- Offutt, J. (2002). Quality attributes of web software applications. *IEEE software 19*(2), 25-32.
- Opdyke, W. F. (1992). *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.
- Oreizy, P., & Taylor, R. N. (1998). On the role of software architectures in runtime system reconfiguration. *IEE Proceedings-Software, 145*(5), 137-145.
- Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimhigner, D., Johnson, G., Medvidovic, N., et al. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications, 14*(3), 54-62.
- Oreizy, P., Medvidovic, N., & Taylor, R. N. (1998). *Architecture-based runtime software evolution*. In Proceedings of the 20th international conference on Software engineering (pp. 177-186). IEEE.
- Ossher, H., & Tarr, P. (1999). *Using subject-oriented programming to overcome common problems in object-oriented software development/evolution*. In Proceedings of the 21st international conference on Software engineering (pp. 687-688). ACM.
- Oussalah, C. (1999). *Génie objet: analyse et conception de l'évolution*: Hermès Science Publ.
- Oussalah, M. (2005). *Ingénierie des composants: concepts, techniques et outils*: Vuibert informatique.
- Oussalah, M. (2014). *Software Architecture 1*: Wiley-ISTE.
- Oussalah, M., Sadou, N., & Tamzalit, D. (2006). *SAEV: A model to face evolution problem in software architecture*. In Proceedings of the International ERCIM Workshop on Software Evolution (pp. 137-146).

P

- Parnas, D. L. (1994). *Software aging*. In Proceedings of 16th International Conference on Software Engineering (pp. 279-287). IEEE.
- Padhy, N., Panigrahi, R., & Baboo, S. (2015). *A Systematic Literature Review of an Object Oriented Metric: Reusability*. In 2015 International Conference on Computational Intelligence and Networks (pp. 190-191). IEEE.
- Pei Breivold, H. (2009). *Software Architecture evolution and software evolvability*. School of Innovation, Design and Engineering, Directed by Crnkovic I., Mälardalen University, 85 p.
- Perry, D. E. W., Alexander L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes, 17*(4), 40-52.

- Plasil, F., Balek, D., & Janecek, R. (1998). *SOFA/DCUP: Architecture for component trading and dynamic updating*. In Proceedings of the 4th International Conference on Configurable Distributed Systems: Cat. No. 98EX159 (pp. 43-51). IEEE.
- Platt, R., & Thompson, N. (2019). *The Past, Present, and Future of UML Advanced Methodologies and Technologies*. In Network Architecture, Mobile Computing, and Data Analytics (pp. 1452-1460). IGI Global.
- Pohl, K., Böckle, G., & van Der Linden, F. J. (2005). Software *product line engineering: foundations, principles and techniques*. Springer Science & Business Media.
- Port, D., & Huang, L. (2003). *Strategic architectural flexibility*. In Proceedings of the International Conference on Software Maintenance: ICSM'2003 (pp. 389-396). IEEE.
- Poulin, J. S., & Caruso, J. M. (1993). *A reuse metrics and return on investment model*. In [1993] Proceedings Advances in Software Reuse (pp. 152-166). IEEE.
- Pressman, R. S. (2005). *Software engineering: a practitioner's approach*: Palgrave Macmillan.
- Prieto-Diaz, R., & Neighbors, J. M. (1986). Module interconnection languages. *Journal of Systems and Software, 6*(4), 307-334.

Q

- Qazi, N., McElholm, M., & Maguire, L. (2018). A Model-View-Controller (MVC) architecture for contextual visualisation of task-based multi-dimensional energy KPIs in a manufacturing process. *International Journal of Ambient Energy, 39*(4), 406-413.

R

- Rabiser, R., Grunbacher, P., & Dhungana, D. (2007). *Supporting product derivation by adapting and augmenting variability models*. In 11th International Software Product Line Conference: SPLC 2007 (pp. 141-150). IEEE.
- Reichwein, A., & Paredis, C. J. *Overview Of Architecture Frameworks And Modeling Languages For Model-Based Systems Engineering*. In Proceedings of ASME (pp. 1-9).
- Reiss, S. P. (1990). Connecting tools using message passing in the Field environment. *IEEE software 7*(4), 57-66.
- Ripeanu, M. (2001). *Peer-to-peer architecture case study: Gnutella network*. In Proceedings of the 1st international conference on peer-to-peer computing (pp. 99-100). IEEE.
- Rising, L., & Janoff, N. S. (2000). The Scrum software development process for small teams. *IEEE software, 17(4),* 26-32.
- Robbes, R., & Lanza, M. (2007). A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science, 166*, 93-109.
- Rodriguez, J. M., Crasso, M., Mateos, C., Zunino, A., & Campo, M. (2013). Bottom-up and top-down cobol system migration to web services. *IEEE Internet Computing, 17*(2), 44-51.
- Rose, L. M., Herrmannsdoerfer, M., Mazanek, S., Van Gorp, P., Buchwald, S., Horn, T., et al. (2014). Graph and model transformation tools for model migration. *Software & Systems Modeling, 13*(1), 323-359.
- Rose, L. M., Paige, R. F., Kolovos, D. S., & Polack, F. A. (2009). *An analysis of approaches to model migration*. In Proc. Joint MoDSE-MCCM Workshop (pp. 6-15).
- Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*: Addison-Wesley.
- Rugaber, S. (1999). *A tool suite for evolving legacy software*. In Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360) (pp. 33-39). IEEE.

S

- Sadou-Harireche, N. (2007). *Evolution Structurelle dans les Architecture Logicielles à base de Composants*. PhD thesis, sous la direction de Oussalah M, Université de Nantes, 182 p.

- Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS), 4*(2)*,* 14.
- Salim, F. D., & Burry, J. (2010). *Software openness: evaluating parameters of parametric modeling tools to support creativity and multidisciplinary design integration*. In International Conference on Computational Science and Its Applications (pp. 483-497). Springer, Berlin, Heidelberg.
- Samarthyam, G., Suryanarayana, G., & Sharma, T. (2016). *Refactoring for software architecture smells*. In Proceedings of the 1st International Workshop on Software Refactoring (pp. 1-4). ACM.
- Sanders, J. (1994). *Software quality: a framework for success in software development and support*. (No. 04; QA76. 76. Q35, S3.).
- Schmid, K., & Verlage, M. (2002). The economic impact of product line adoption and evolution. *IEEE software, 19*(4)*,* 50-57.
- Schmidt, M.-T., Hutchison, B., Lambros, P., & Phippen, R. (2005). The enterprise service bus: making service-oriented architecture real. *IBM systems journal, 44*(4), 781-797.
- Schroeder, J., Holzner, D., Berger, C., Hoel, C.-J., Laine, L., & Magnusson, A. (2015). *Design and evaluation of a customizable multi-domain reference architecture on top of product lines of self-driving heavy vehicles: an industrial case study*. In Proceedings of the 37th International Conference on Software Engineering-Volume 2 (pp. 189-198). IEEE Press.
- Sha, L., Rajkumar, R., & Gagliardi, M. (1996). *Evolving dependable real-time systems*. In Proceedings of the 1996 IEEE Aerospace Applications Conference. (Vol. 1, pp. 335-346). IEEE.
- Shaw, M. (1993*). Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status*. In Workshop on Studies of Software Design (pp. 17-32). Springer, Berlin, Heidelberg.
- Shaw, M., & Clements, P. (1996). *Toward boxology: Preliminary classification of architectural styles*. In Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops (pp. 50-54). ACM.
- Shaw, M., DeLine, R., & Zelesnik, G. (1996). *Abstractions and implementations for architectural connections*. In Proceedings of International Conference on Configurable Distributed Systems (pp. 2-10). IEEE.
- Smeda, A., Oussalah, M., & Khammaci, T. (2005). *Madl: Meta architecture description language*. In Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05) (pp. 152-159). IEEE.
- Sommerville, I. (2011). *Software engineering 9th Edition*. ISBN-10, 137035152.
- Stahl, C., Massuthe, P., & Bretschneider, J. (2009). *Deciding substitutability of services with operating guidelines*. In Transactions on Petri Nets and Other Models of Concurrency II (pp. 172-191). Springer, Berlin, Heidelberg.
- Stavridou, V. (1999). *Provably dependable software architectures for adaptable avionics*. In Gateway to the New Millennium. 18th Digital Avionics Systems Conference. Proceedings (Cat. No. 99CH37033) (Vol. 2, pp. 9-C). IEEE..
- Subramanian, N., & Chung, L. (2001). *Software architecture adaptability: an NFR approach*. In Proceedings of the 4th International Workshop on Principles of Software Evolution (pp. 52-61). ACM.
- Svahnberg, M. (2004). An industrial study on building consensus around software architectures and quality attributes. *Information and Software Technology, 46*(12), 805-818.
- Swanson, E. B. (1976). *The dimensions of maintenance*. In Proceedings of the 2nd international conference on Software engineering (pp. 492-497). IEEE Computer Society Press.
- Szyperski, C., Bosch, J., & Weck, W. (1999). *Component-oriented programming*. Lecture Notes in Computer Science, 1743, 184-184.

T

- Tahir, M., Khan, F., Babar, M., Arif, F., & Khan, F. (2016). Framework for Better Reusability in Component Based Software Engineering. *Journal of Applied Environmental and Biological Sciences (JAEBS), 6*, 77-81.
- Tamzalit, D., & Oussalah, M. (2003). *A Conceptualization of OO evolution.* In International Conference on Object-Oriented Information Systems (pp. 274-278). Springer, Berlin, Heidelberg.
- Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Robbins, J. E., Nies, K. A., et al. (1996). A component-and message-based architectural style for GUI software. *IEEE Transactions on software engineering, 22*(6), 390-406.
- Tekinerdogan, B., & Aksit, M. (1996). *Adaptability in object-oriented software development: Workshop report.* In Proceedings of the 10th Annual European Conference on Object-Oriented Programming (ECOOP).
- Terho, H., Suonsyrjä, S., Systä, K., & Mikkonen, T. (2017). *Understanding the relations between iterative cycles in software engineering.* In Proceedings of the 50th Hawaii International Conference on System Sciences (pp 5900–5909).
- Tilley, S. R., & Smith, D. (1995). *Perspectives on legacy system reengineering.* Reengineering Center, Software Engineering Institute, Carnegie Mellon University.

V

- Valipour, M. H., AmirZafari, B., Maleki, K. N., & Daneshpour, N. (2009). *A brief survey of software architecture concepts and service oriented architecture.* In 2009 2nd IEEE International Conference on Computer Science and Information Technology (pp. 34-38). IEEE.
- Vallecillo, A. (2001). RM-ODP: The ISO reference model for open distributed processing. *DINTEL Edition on Software Engineering, 3*, 66-69.
- Van den Berg, M., Tang, A., & Farenhorst, R. (2009). *A constraint-oriented approach to software architecture design.* In 2009 9th International Conference on Quality Software (pp. 396-405). IEEE.
- Van der Linde, A., Fouto, P., Leitão, J., Preguiça, N., Castiñeira, S., & Bieniusa, A. (2017). *Legion: Enriching Internet Services with Peer-to-Peer Interactions.* In Proceedings of the 26th International Conference on World Wide Web (pp. 283-292). International World Wide Web Conferences Steering Committee.
- Van der Linden, F. J., Schmid, K., & Rommes, E. (2007). *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media.
- Van Gurp, J., & Bosch, J. (2002). Design erosion: problems and causes. *Journal of Systems and Software, 61*(2), 105-119.

W

- Wachsmuth, G. (2007). *Metamodel adaptation and model co-adaptation.* In European Conference on Object-Oriented Programming (pp. 600-624). Springer, Berlin, Heidelberg.
- Wachsmuth, M. H. a. G. (2013). *Coupled Evolution of Software Metamodels and Models.* In Evolving Software Systems (pp. 33-63). Springer, Berlin, Heidelberg.
- Wang, G., & Fung, C. K. (2004). *Architecture paradigms and their influences and impacts on component-based software systems.* In 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the (pp. 10-pp). IEEE.
- Waggoner, D. B., Neely, A. D., & Kennerley, M. P. (1999). The forces that shape organisational performance measurement systems: An interdisciplinary review. *International Journal of Production Economics, 60*, 53-60.
- Ward, P. T., & Gullekson, G. (1994). *Real-time object-oriented modeling*: New York; Toronto: Wiley & Sons.
- Williams, B. J., & Carver, J. C. (2010). Characterizing software architecture changes: A systematic review. *Information and Software Technology, 52*(1), 31-51.

- Wirfs-Brock, A., & Wilkerson, B. (1989). Variables limit reusability. *Journal of Object-Oriented Programming, 2*(1), 34-40.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*: Springer Science & Business Media.
- Woods, D. (2015). Four concepts for resilience and their implications for systems safety in the face of complexity. *Reliability Engineering and System Safety, 141*, 5-9.
- Woods, D. D. (2018). The theory of graceful extensibility: basic rules that govern adaptive systems. *Environment Systems and Decisions, 38*(4), 433-457.

Z

- Zachman, J. (2002). The zachman framework for enterprise architecture. *Zachman International, 79*.
- Zachman, J. A. (1987). A framework for information systems architecture. *IBM systems journal, 26*(3), 276-292.
- Zarvić, N., & Wieringa, R. (2014). An integrated enterprise architecture framework for business-IT alignment. *Designing Enterprise Architecture Frameworks: Integrating Business Processes with IT Infrastructure, 63*, 9.
- Zhang, B., Bao, L., Zhou, R., Hu, S., & Chen, P. (2008). *A black-box strategy to migrate GUI-based legacy systems to web services.* In 2008 IEEE International Symposium on Service-Oriented System Engineering (pp. 25-31). IEEE.
- Zhang, H., Li, J., Zhu, L., Jeffery, R., Liu, Y., Wang, Q., et al. (2014). Investigating dependencies in software requirements for change propagation analysis. *Information and Software Technology, 56*(1), 40-53.
- Zhao, C., Kong, J., Dong, J., & Zhang, K. (2007). Pattern-based design evolution using graph transformation. *Journal of Visual Languages & Computing, 18*(4), 378-398.
- Zimmermann, O. (2015). Architectural refactoring: A task-centric view on software evolution. *IEEE software, 32*(2), 26-29.

Fiche de prospection d'un papier de recherche

N°…….. Fait le :…./…/…..

Titre (Citation:…….): ....
....
Auteurs et H Index: 1 ..... HI:….2 ..... HI…3 .....HI
Universités :1..... ……..2..... ….....3 .....
Description du papier : ....
....
....
....
....

A-Description structurelles

MOE		Object ev.		Type	Niv. Hiérarchique		Classe	Observation
Em.	Red	Art	Prc	POB-POC-PCB-PCS	Du	Au	De 1 à 6	

B-Description de qualité : attentes d'un modèle d'évolution

1. Capacité d'énonciation-

Ev Formulée ?	Ev. Gère les impactes ?	Garde la trace	Observation
Descriptif/ théorique ?- Argumentatif ?- Application réelle ?-	*Peut-on connaitre d'avance la conséquence de l'application de l'opérateur avant et après évolution ?*	*Peut-on revenir sur l'architecture de départ après l'évolution ?*	

2. Capacité d'expressivité-

Niveau Modélisation	Niveau Abstraction	Mode d'expressivité	Mécanique opératoire	Domaine	Observation
A_0 ----> A_1 ----> A_2 ----->	*Boite Noire? Boite Blanche ? Boite Grise ?*	*Graphique ? Textuel ? Code ?*	*Déductif ? Inductif ? Classification ?*	*Général ? Spécifique ?*	

3. Capacité d'évaluation-

Réutilisation	Adaptation	Performance	Support d'évaluation	Observation
Forte ou Moyenne ou Faible	*Se fait sans stopper l'architecture en cours*	*Gère-t-il ses propres erreurs en cours d'exécution ? Est-il extensible ?*	*Possède-t-il des outils ou des mécanismes d'évaluation ?*	

C-Notes

……
……
……
……
……
……

Modèle de description d'architecture basé service

Noureddine Gasmallah

Résumé

Depuis plus de deux décennies de recherches, beaucoup d'efforts sont déployés pour traiter l'avènement de la problématique d'évolution en génie logiciel. Le recours à l'abstraction fut une des solutions clé pour réduire la complexité liée à ce problème. En effet, l'architecture logicielle est le niveau d'abstraction le plus favorable pour anticiper la description de l'évolution et répondre aux besoins émergents. La capitalisation du savoir-faire lié à l'évolution nécessite systématiquement des outils d'analyse et de compréhension des techniques déjà réalisées dans le domaine. C'est dans ce cadre que s'inscrit la problématique de notre thèse ayant pour objectif d'identifier, d'analyser et de comparer les méthodes d'évolution des architectures logicielles.

Les contributions de la thèse couvrent l'aspect structurel, comportemental et qualitatif et qui sont structurées en trois réflexions de fond. La première se focalise sur la modélisation d'un référentiel de description d'architecture permettant de capitaliser les caractéristiques liées aux différentes dimensions qui articulent le référentiel proposé. Dans la deuxième, nous proposons une classification dont l'objectif principal est de décrire la couverture actuelle des travaux dans le domaine. Nous proposons dans la dernière contribution, un modèle de description de la qualité décrivant les critères de qualité attendus d'une approche d'évolution.

Nos propositions sont appuyées par une étude expérimentale exhibant l'applicabilité du référentiel pour assurer une bonne capitalisation des compétences consenties en évolution architecturale.

Mots-clés: *Architecture logicielle, Evolution d'architecture, Modélisation d'architecture, Méta-modélisation, ADL, Service, Architecture orientée service, Architecture orientée composants et Qualité d'évolution.*

Abstract

For over two decades of research, considerable effort is dedicated to addressing evolution issues within the area of software engineering. The use of abstraction was one of the key solutions to reduce the complexity linked to this issue. Indeed, software architecture is the most favorable level of abstraction to anticipate the description of software evolution in order to meet emergent needs. The capitalization of knowledge and know-how linked to the evolution activity requires tools for analyzing and understanding of existing techniques carried out to date. It is in this context that the problem statement of this thesis aims to identify, analyze and compare the methods of evolution. Further, we focus on the main challenge which is to identify a framework for analyzing and comparing architectural evolution methods.

Our contributions cover several aspects of evolution such as structural, behavioral and qualitative. The achievements are structured into three basic reflections. The first focuses on the modeling of an evolution framework describing and addressing characteristics related to various dimensions which articulate the proposed framework. The second results from the exploitation of the first contribution by proposing a classification based on a set of dimensions. This aims to describe the coverage of current research in the field. For the third aspect, we propose a quality description model describing the expected quality criteria from an evolution approach.

We support our proposals with an experimental study illustrating the applicability of the Framework. This attempt to provide ease-of-use of our proposals to capitalize knowledge and know-how linked to a given architectural evolution.

Keywords- *Software Architecture, Architecture Evolution, Architecture Modeling, Meta-Modeling, ADL, Service, Service-Oriented Architecture, Component-Oriented Architecture and Quality of evolution.*