Contribution à la Multi-modélisation des Applications Distribuées pour le Contrôle de l'Evolution des Logiciels

Adeel Ahmad

Université du Littoral Côte d'Opale

Thèse

Pour obtenir le grade de
DOCTEUR de l'Université du Littoral Côte d'Opale
Spécialité : Informatique

Présentée et soutenue publiquement par :

Adeel Ahmad

Le 09 décembre 2011

Contribution à la Multi-modélisation des Applications Distribuées pour le Contrôle de l'Évolution des Logiciels

---

Composition du jury :

M. Basson Henri	Professeur à l'Université Lille Nord de France	Directeur
M. Bouneffa Mohamed	Maître de Conférences à l'Université du Littoral	Co-directeur
Amaziane Mourad	Côte d'Opale	
M. Godart Claude	Professeur à l'Université de Lorraine	Rapporteur
M. Oussalah Mourad Chabane	Professeur à l'Université de Nantes	Rapporteur
Mme. Leclet Dominique	Maître de Conférences, HDR à l'Université de Picardie Jules Verne	Examinatrice
M. Fonlupt Cyril	Professeur à l'Université du Littoral Côte d'Opale	Examinateur (Président)

Thèse réalisée au sein du Laboratoire d'Informatique Signal et Image de la Côte d'Opale - EA 4491-
Équipe Model (multi-modélisation et évolution des logiciels)

Maison de la Recherche Blaise Pascal 50, rue Ferdinand Buisson BP 719 62228 Calais Cedex France

À la mémoire de mon défunt père.

Remerciements

Les travaux présentés dans ce manuscrit ont été effectués au sein du Laboratoire d'Informatique Signal et Image de la Côte d'Opale (LISIC) de l'Université du Littoral Côte d'Opale (ULCO). Je souhaite adresser mes remerciements à toutes les personnes qui ont contribué de près ou de loin à l'élaboration de ces travaux.

Je tiens à remercier en tout premier lieu le directeur de cette thèse, Prof. Henri Basson. Je lui présente ma plus profonde gratitude pour son aide inestimable et ainsi qu'à la réussite de ces années universitaire à l'ULCO. Il était toujours à côté de moi, côté professionnel ainsi que du côté personnel dans les moments de doutes.

Je remercie très fort Mourad Bouneffa pour le partage de ses connaissances, de son esprit d'équipe, et le temps qu'il a bien voulu me consacrer aussi pendant ses week-ends. Sans lui ce document n'aurait jamais vu le jour, dans ce délai.

Je remercie sincèrement Laurent Deruelle pour avoir aider et partager la richesse de ses connaissance. Je remercie Dominique Verhaghe pour son aide à l'installation du réseau de notre équipe et les servers de gestion de versions. Je tiens à remercier personnellement, Virginie Marion, Christophe Renaud, Arnaud Lowandowski, et tous mes collègues du LISIC, qui ont développé un environment de travail très sympathique.

Je chaleureusement remercie les rapporteurs M. Claude Godart, M. Mourad Oussalah et les examinateurs M. Cyril Fonlupt et Mme. Dominique Leclet pour avoir accepté de juger mon travail et me faire leurs précieuse observations.

La salle B125 du LISIC porte des souvenirs très spéciaux, je remercie Oussama Elgerari, MO Kherbouche, et Hanna Mazyad qu'ont crées un esprit très chaleureux et mature.

Je remercie cordialement la commission de l'enseignement supérieur du Pakistan (HEC : *Higher Education Commission of Pakistan*), qui a financé une grande partie de thèse et je remercie aussi tous mes camarades HEC, sans oublier, Abdul Wahab, Fazli, Mubeen, Qasim, Manzoom, et Tauqeer qui m'ont soutenue au cours de réalisation de ce mémoire. Je tiens aussi à remercier la communauté pakistanaise à Calais qui m'ont aidé à moins sentir la douleur d'être loin du pays.

Il est impossible à résumer mes gratitudes pour ma famille. Je remercie du fond du cœur ma mère, mon grand frère, mes sœurs, ma femme et les enfants, et tous les membres de ma famille pour leurs encouragements, et qui on fait ce qu'ils peuvent pour que je me sens pas seul et loin de mes proches.

Calais, le 08 November 2011

Adeel Ahmad

Résumé

Le contrôle de l'évolution des logiciels exige une compréhension profonde des changements et leur impact sur les différents artefacts du système.

Nous proposons une approche de multi-modélisation pour l'analyse d'impact du changement pour une compréhension des effets des modifications prévus ou réels dans les systèmes distribués. Ce travail consiste à élaborer une modélisation des artefacts logiciels et de leur différents liens d'interdépendance pour construire un système à base de connaissances permettant, entre autres, d'assister les développeurs et les chargés de l'évolution des logiciels pour établir une évaluation *a priori* de l'impact des modifications. La modélisation que nous élaborons intègre deux descriptions majeures des logiciels, dans un premier temps, la description structurelle sous-jacente qui englobe l'ensemble des niveaux granulaires et l'abstraction des constituants logiciels, et ensuite la description qualitative conçue pour s'intégrer à la description précédente.

Deux modèles, d'abord élaborés individuellement pour les deux descriptions respectives, ont été intégrés ou mis en correspondance dans l'objectif d'étudier l'impact de toute modification et sa potentielle propagation à travers les constituants logiciels concernés. Lors de chaque modification, il devient alors possible d'établir un bilan qualitatif de son impact. La modélisation intégrée est élaborée pour se prêter à un raisonnement à base de règles expertes. La modélisation proposée est en cours d'expérimentation et validation à travers le développement d'une plate-forme d'implémentation basée sur l'environnement Eclipse.

Mots clé : Evolution du Logiciel, Analyse d'Impact des Modifications, Multi-modélisation des Logiciels, Architecture du Logiciel, Modélisation Structurelle, Modélisation Qualitative.

Abstract

The software evolution control requires a complete understanding of the changes and their impact on the various system artifacts.

We propose a multi-modeling approach for the change impact analysis to provide assistance in understanding the effects of projected or actual changes in distributed software systems. This work elaborate the modeling of software artifacts along with their various interdependencies to build a knowledge-based system, which allows, among others, an assistance for the software developers or maintenance engineers to establish an *a priori* evaluation of impact of changes. The model we develop integrates two major descriptions of software, at first, the underlying structural description that encompasses the levels of granularity and abstraction of software artifacts, and then the qualitative description designed to integrate the structural description.

Initially, the formal models are designed separately for the respective descriptions, and then these are integrated for the objective to study the change impact and its potential propagation through the affected software artifacts. For a change, it is important to establish a qualitative assessment of its impact. The integrated modeling leads to a reasoning based on expert rules. The proposed model is being tested and validated through the development of a platform, implemented in the Eclipse environment.

Keywords : Software Evolution, Change Impact Analysis, Software Multi-modeling, Software Architecture, Structural Modeling, Qualitative Modeling

Table des matières

Table des figures

Liste des tableaux

Listings

Première partie

Introduction Générale

Introduction

Dans les applications informatisées associées à tous les domaines d'activités, l'évolution des logiciels est le processus majeur assurant l'innovation et l'amélioration continues des méthodes, des techniques et des outils de traitement de l'information. Ceci explique que le coût consacré à ce processus avoisine les 80% du coût global de l'ensemble des activités informatiques. Mener à bien l'évolution des logiciels en optimisant l'effort et le coût élevé est devenu ainsi un objectif principal de l'ingénierie du logiciel. L'optimisation du processus requiert une modélisation permettant de mieux contrôler les activités sous-jacentes de l'évolution. Une compréhension des connaissances descriptives des artefacts développés du logiciel est une condition sine qua none de la réussite du processus d'évolution. Ces connaissances englobent d'abord la description individuelle de chaque artefact et son interdépendance vis-à-vis du reste du logiciel. La description architecturale adoptée des artefacts logiciels et les graphes de vues partielles des relations (héritage, communications, l'appel, importation, etc.) entre artefacts, font partie de l'ensemble des connaissances à rendre accessibles lors des différentes situations du raisonnement qui peuvent jalonner le processus de l'évolution.

La mise en œuvre d'un processus d'évolution revient à réaliser des changements plus ou moins importants de certains constituants du logiciel. Ces changements peuvent être source de dégradation sur le plan fonctionnel, qualitatif, ou comportemental du logiciel modifié. D'où la nécessité d'une modélisation permettant de réaliser une analyse *a priori* de l'impact prévisionnel du changement du logiciel et une analyse *a posteriori* de l'impact effectif de ce changement.

Ces dernières années, les applications informatiques distribuées ont tendance à être plus hétérogènes avec une implémentation codée souvent en plusieurs langages de programmation sur différents plateformes. Ceci rend nécessaire une approche de modélisation globale permettant de répondre aux différents besoins de compréhension et du

raisonnement pour l'évolution des applications.

Les environnements récents de développement des logiciels facilitent de plus en plus la mise en œuvre de logiciels sans apporter l'aide nécessaire à la compréhension détaillée d'applications de complexité grandissante. UML (Unified Modeling Language) (1) préconise une description conceptuelle multi modèles de divers aspects du logiciel sans modélisation des liens d'interdépendance existants entre ces modèles. Bien sûr, il existe un large éventail d'outils pour assister les activités de l'ingénierie inverse, de refactoring, ou re-ingénierie (2, 3, 4, 5) mais ils offrent des connaissances limitées ou insuffisantes sur le flux de l'impact des modifications appliquées. L'exigence de l'analyse de l'impact du changement sur les artefacts logiciels ne peut être satisfaite sans modélisation adaptée et descriptive du logiciel de plusieurs points de vue : architectural, fonctionnel, comportemental et qualitative. Cette modélisation doit être capable d'assurer différentes vues stratifiées et détaillées des artefacts logiciels permettant aux chargés de l'évolution de mener à bien le raisonnement adopté au changement concerné.

Une opération de modification d'un artefact logiciel, ayant des liens d'interdépendance avec le reste des artefacts du logiciel, peut produire des situations d'incohérence de l'artefact modifié avec le reste du logiciel. De même que les variations qualitatives résultant du changement appliqué à un composant peuvent atteindre par un phénomène de propagation en chaîne ou « ripple effect » à plusieurs autres composants.

La compréhension du logiciel et l'acquisition des connaissances sur le système sont essentielles pour toutes les activités menées en génie logiciel. Or, il est à la fois difficile et complexe d'identifier avec détail les chemins des flux d'impact du changement à travers l'ensemble des artefacts concernés en partant de l'artefact modifié. En outre, le flux de propagation d'impact n'est pas uniquement structurel, il peut être fonctionnel, qualitatif ou comportemental. Un changement de tout artefact logiciel doit donc être pris en compte et il faudra le traiter en contrôlant ses effets par l'analyse de son impact (6). Cela ne peut pas être réalisé sans disposer de connaissances pertinentes et exploitables sur les artefacts logiciels, ceci inclut leur description individuelle dans ses formes les plus abstraites dont celles les plus détaillées ainsi que les différents types de relations entre les artefacts (7). Le savoir faire activé au sein du processus de l'évolution ainsi que les connaissances descriptives utilisées par ce processus peuvent être structurés et décrites par un système à base de connaissances qui évolue avec l'expertise des chargés de l'évolution et les changement apportés aux artefacts logiciels (8).

Dans le cadre du contrôle de l'évolution des logiciels, cette thèse propose une contribution de modélisation intégrant un modèle structurel pour l'évolution du logiciel (SMSE : *Structural Model for Software Evolution*) et un modèle qualitatif pour l'évolution de logiciels (QMSE : *Qualitative Model for Software Evolution*). Le SMSE vise une aide semi-automatique pour l'analyse de l'impact du changement et le contrôle de sa propagation à travers les éléments structurels constituant le logiciel. Pour une meilleure compréhension des applications logicielles les artefacts issus des différentes phases du cycle de développement sont structurés en couches de plusieurs niveaux (9, 10). Le niveau code source et son analyse est une tâche clé de notre approche. Les informations spécifiques aux logiciels sont extraites et adaptées pour la visualisation en forme des graphes. Bien que l'analyse du logiciel établit plusieurs représentations stratifiées en différents niveaux d'abstraction et de granularité, les résultats affichés peuvent être réduits ou détaillés à la demande de l'utilisateur chargé du processus d'évolution. L'approche est également appliquée pour contrôler le processus du changement de constituants architecturaux du logiciel (11).

L'approche est validée en réalisant une plateforme spécifique permettant la collecte des connaissances et un mécanisme semi-automatique facilitant à la fois l'analyse structurelle et qualitative de l'impact du changement. La validation est effectuée principalement dans la phase du codage pour valider la modélisation des artefacts logiciels et leurs interdépendances dans des applications distribuées. Le reste de la thèse est organisé comme suit :

- Le chapitre 1 décrit la problématique du domaine de recherche de la thèse. Avec l'introduction du domaine, il aborde également les avantages et les inconvénients des approches proposées en modélisation pour le contrôle de l'évolution du logiciel.
- Le chapitre 2 introduit le modèle structurel pour l'évolution du logiciel. Il aborde en particulier les détails des éléments impliqués pour l'identification des chemins de propagation de l'impact du changement et de son analyse avec des exemples d'illustration.
- Le chapitre 3 présente l'approche de modélisation de la SMSE dans le contexte des langages de descriptions d'architectures. Il explique brièvement les éléments des langages de description architecturale et leur correspondance avec le modèle SMSE.

– Le chapitre 4 définit en détail la description de la stratégie de mise en œuvre de l'approche proposée l'analyse de l'impact du changement. Il présente le système à base de connaissances destiné à l'évaluation de la SMSE. Le chapitre montre également la formulation de règles d'analyse d'impact du changement, et discute l'application de ces règles dans un contexte de relations des artefacts modélisés dans SMSE, avec l'aide d'exemples illustratifs.
– Le chapitre 5 est consacré à la formulation du modèle qualitatif pour l'évolution du logiciel. Le chapitre aborde les questions liées à l'assurance qualité des logiciels dans le contexte des principaux modèles de qualité et propose l'élaboration du modèle QMSE. La dernière partie du chapitre présente la mise en correspondance (mapping) entre le SMSE et le QMSE.
– Le chapitre 6 valide la mise en œuvre et les résultats de l'approche de la modélisation intégrée. Il démontre le développement d'une plate-forme de prototype pour l'analyse de la propagation de l'impact des changements, ce qui facilite l'analyse *a priori* de l'impact changement.
– Le chapitre 7 présente la conclusion de la contribution de cette thèse et les perspectives des travaux.

Chapitre 1

Évolution de Logiciels et État de l'Art

1.1 Introduction

L'évolution du logiciel est un champ relevant du domaine de l'ingénierie du logiciel et ayant comme but l'étude des moyens d'adaptation du logiciel aux changements continus qui affectent aussi bien les exigences ou besoins des utilisateurs que les environnements opérationnels supports. Cette étude ne se limitent pas seulement aux changements affectant le produit, à savoir le logiciel, mais également au processus de développement lui même. Certains travaux s'intéressent également au processus même d'évolution et cela en analysant des données provenant de bases d'objets de projets de développement pour extraire des connaissances relatives aux tendances et à la nature du changement affectant le logiciel.

Le génie ou ingénierie du logiciel est un terme issu de la première conférence consacrée à ce domaine et organisé par le comité scientifique de l'OTAN en Allemagne en 1968 (12). Cette conférence a posé les principes et les fondements permettant de faire en sorte que la production du logiciel soit semblable à toutes les disciplines relevant des différents domaines de l'ingénierie. Le but étant de produire des logiciels opérationnels et fiables et cela dans le respect des coûts et des délais. En 1970, Royce considérait la notion de maintenance du logiciel comme une dernière phase du cycle de vie du logiciel dit en cascade et qui faisait office de référence à l'époque (13). Cependant, cette phase était qualifiée de postproduction à l'image de la maintenance des objets manufactu-

rés (véhicules, etc.). Cela veut dire que c'est une phase qui n'est activable qu'après la livraison ou recette du logiciel. Cela revenait donc à limiter les actions ou tâches de la maintenance à la correction des erreurs ou défauts persistants et ayant donc échappé aux différentes phases de test.

Entre la fin des années soixante dix et le début de la décennie 1980, Manny Lehman avait énuméré ces fameuses lois concernant l'évolution des systèmes informatiques et qui sont connues jusqu'à nos jours sous le nom de lois de Lehman (14). Son principe d'incertitude et sa vue du processus de développement du logiciel comme un système de feedback a permis de poser les fondements de son travail concernant l'évolution continue des systèmes informatique (15). Il a, avec L.A. Belady (16), initié les fondements de la recherche en évolution du logiciel basée sur des études empiriques. Ils ont en effet appliqué cette approche dans le cadre de l'étude de l'évolution du système d'exploitation de l'IBM 360. Leur postulat était que des études empiriques pouvaient permettre une meilleure compréhension des processus du génie logiciel et de ce fait contribuer à un meilleur contrôle des coûts et de la qualité du logiciel.

La même époque a vu l'émergence des processus des activités importantes telles que l'analyse et la propagation de l'impact du changement du logiciel mettant déjà en évidence le fait qu'un changement du logiciel était rarement isolé (17).

A partir du début des années 90, on assistait à une large diffusion de travaux concernant l'évolution du logiciel qui devenaient de plus en plus présents et prépondérants dans la communauté du génie logiciel (18, 19). Cela s'était traduit, également, par la mise en œuvre de nombreux modèles de cycle de vie du logiciel qualifiés d'évolutionnaires tels que le modèle évolutionnaire de Gilb (20, 21), le modèle en spiral de Boehm (22) et le modèle par « étapes » de Bennett et Rajlich (23). Ces modèles sont itératifs et permettent un développement incrémental du logiciel. Nogueira et ses collègues ont résumé dans (24) les éléments constitutifs des modèles de cycle de vie évolutionnaires. Cela concerne d'abord le problème de la planification des projets de développement qui doivent inclure le fait que le nombre d'itérations n'est pas forcément connu dès le démarrage. Ils mettent également en évidence la nécessité pour ces processus de permettre une rapidité maximale dans la prise en charge de l'évolution et le fait qu'ils accordent plus d'importance à la flexibilité et la rapidité du développement qu'à la haute qualité. En effet la recherche du zéro défaut peut conduire à des retards de développement ce qui peut faire rater l'occasion de profiter des effets de niche. Cependant, ces processus

tablent sur le fait que la flexibilité et une meilleure évolutivité conduisent à terme à une meilleure qualité par des améliorations successive et facilite du produit logiciel.

Avec les années 2000, l'évolution du logiciel est devenue une phrase commune et pour ainsi dire ordinaire. Cependant, elle constitue un concept clés pour les processus de développement dits « agiles » (25, 26) dont l'extreme programming (27) est l'un des représentants les plus emblématiques. Ces processus sont des processus itératifs et incrémentaux (évolutionnaires) légers effectués dans des environnement hautement collaboratifs et considérant la prise en charge du changement comme un élément constitutif de base. Cela est du au fait que les méthodes dites agiles offrent une place de choix à l'utilisateur qui est considéré comme un membre de l'équipe de développement et dans le sens ou c'est l'un des principaux validateurs de toutes les activités occurant dans le cadre du processus de développement. D'une certaine façon, ces méthodes peuvent être considérées comme un retour aux anciennes méthodes de développement qui prévalaient avant l'établissement quasi systématique du cycle de vie en cascade (28).

Durant les dernières années plusieurs chercheurs ont réévalué leurs travaux pour mieux appréhender les changements technologiques. Le *Software Engineering Curriculum Guidelines 2004* de ACM/IEEE[1] considère l'évolution du logiciel comme l'un des dix champs clés de l'enseignement du génie logiciel. Il existe également de nombreuses équipes de recherche internationales ayant comme champ d'étude l'évolution du logiciel. Comme on pouvait s'y attendre, il peut exister quelques divergences dans tous ces efforts de recherches qui incluent des recherches théoriques et fondamentales, des recherches empiriques, des travaux en matière de construction d'outils de visualisation du logiciel au service de sa compréhension et donc de son évolution, etc.

La suite de ce chapitre est organisée comme suit : Nous présentons une différence entres les concepts de l'évolution, la maintenance, et le changement dans la section 1.2. Dans la section 1.3, nous classifions le notions d'évolution du logiciel en considérant les phases intra et post développement du logiciel. La section 1.4 adresse un état de l'art des principaux travaux concernant l'évolution du logiciel et particulièrement du logiciel. La section 1.5 présente brièvement notre approche de multi-modélisation pour l'évolution du logiciel utilisant les système à base de connaissance. La section 1.6 conclut le chapitre en résumant ses apports, notamment.

1. http://sites.computer.org/ccse/

1.2 Evolution, Maintenance, ou Changement du logiciel

Durant les quatre dernières décennies, l'évolution du logiciel a connu une importance de plus en plus grandissante faisant d'elle un concept clé du génie logiciel. Le standard IEEE 1219 de la maintenance du logiciel (29) définit la maintenance comme « *la modification du logiciel après sa livraison pour corriger des défauts, améliorer les performances ou tout autre attribut, ou adapter le produit logiciel aux changements affectant son environnement* ». Le terme évolution du logiciel manque de définition standard. Généralement, évolution et maintenance du logiciel sont utilisées comme synonyme l'une de l'autre. Ainsi, le standard international ISO/IEC 14764 de la maintenance du logiciel (30) documente l'importance des aspects ante-livraison de la maintenance comme la planification des actions de maintenance, etc. Le *Software Engineering Body of Knowledge* (SWEBOK) (31) met en relief le besoin en matière de support de la maintenance aussi bien dans les phases précédant la livraison du logiciel que dans celles qui succèdent à sa recette. Il considère certaines activités liées à l'évolution du logiciel comme des thèmes de recherche cruciaux. Ce sont notamment, la compréhension du logiciel, la rétro-ingénierie, le test, l'analyse d'impact du changement, l'estimation des coûts, la mesure du logiciel, la qualité du logiciel, la gestion des configurations, etc. Ces différentes activités seront discutées avec plus de détail dans la section 1.3.

Dans cette thèse nous préférons utiliser le terme d'évolution du logiciel en lieu et place de la maintenance du logiciel. En effet, la maintenance est un terme qui, à notre sens, connote une notion de détérioration physique qu'il faudra minimiser par des interventions faisant partie de l'activité de maintenance. Or, dans le cas du logiciel, il s'agit principalement d'adapter ce dernier aux évolutions qui peuvent affecter son environnement et les besoins fonctionnels ou non des utilisateurs. Les défauts dans le logiciel ne sont pas dus à une détérioration intrinsèque mais plutôt à une préexistence de ces défauts, avant même sa livraison, et à leur aggravation par des actions correctives, adaptatives et perfectives dont les effets n'ont pas été suffisamment bien maîtrisés. Nous distinguons également un autre concept clé qui est le changement du logiciel qui correspond à des ajouts, suppressions et/ou mises-à-jours d'artefacts logiciels rentrant dans le cadre de l'évolution. Un changement est en fait la différence entre l'état d'un artefact logiciel avant et après un événement de modification dictée par un besoin. Généralement, un changement consomme du temps et des efforts principalement en raison

de la difficulté à comprendre les divers aspects qui pourraient être concernés par les modifications appliquées sur le logiciel.

Les exemples qui suivent vont nous permettre d'illustrer ces différents concepts.

- Lorsqu'on réinstalle un logiciel, sans aucune modification de ses artefacts, sur une machine plus puissante permettant l'obtention de meilleures possibilités graphiques, les conséquences de cette migration sont une optimisation des temps de réponses, ainsi qu'une amélioration du rendu de l'interface utilisateur. Nous réalisons donc ici une évolution qualitative en termes d'efficacité et de convivialité. Cette évolution est réalisée sans modification d'artefacts constituant le logiciel.
- Dans un même esprit, dans une phase d'intégration de logiciel dans le Système d'Information de l'entreprise, le statut de ce logiciel peut passer de l'état en-test à l'état certifié pour la mise en service sans aucune modification de ces artefacts. Une fois encore nous réalisons une évolution qualitative en termes de certification sans pour autant avoir réalisé une quelconque modification de l'un des constituants de l'application.
- En revanche, une modification dans un logiciel peut, dans certains cas, n'entraîner que peu ou pas d'évolution. C'est en effet le cas lorsque l'on substitue les commentaires dans un code source par d'autres commentaires de même nature n'assurant pas plus de compréhensibilité du code.

Nous venons de voir au travers de ces exemples que les notions d'évolution et de modification peuvent être dissociées. Néanmoins, dans la plupart des cas, une évolution implique la mise en œuvre d'opérations de modification sur un ensemble de artefacts. De plus, comme il déjà été souligné, les effets que peut engendrer une opération de modification, ne sont généralement pas limités au seul artefact cible de cette opération. Ils se propagent à d'autres artefacts avec lesquels le constituant modifié est lié par différents types de relations qualifiées de propagatrices d'impact des modifications.

Ces relations potentiellement conductrices d'impact, sont très importantes car il n'est par rare, que quel que soit l'effort investi, une évolution soit entachée d'erreurs d'appréciation concernant les effets de modifications appliquées, et *a fortiori* son coût et son délai de réalisation. Ce qui peut, selon l'amplitude de l'évolution ciblée, placer l'entreprise dans une situation délicate. Il est donc impératif, pour minimiser ces erreurs, de procéder à une spécification du contexte de chaque cas d'évolution.

1.3 Typologie de l'évolution

Le vocable évolution est utilisé couramment dans la langue française. Cependant, dans le domaine de l'informatique, et plus particulièrement dans le domaine de l'ingénierie du logiciel, il existe plusieurs types d'évolution, réalisées par diverses opérations de modification.

Selon leur phase d'occurrence dans le cycle de vie du logiciel, deux grandes familles d'évolutions peuvent se distinguer :

1.3.1 Les évolutions intra-développement

Les évolutions intra-développement se traduisent par des modifications, non prises en compte lors de la planification du développement du projet, et qui s'effectuent à une des phases antérieures à l'exploitation. Ce premier type d'évolution concerne les évolutions intra-développement portant sur les constituants du développement de l'application. Une évolution de l'ensemble des acteurs, des procédés, des méthodes ou des formalismes peut s'avérer génératrice d'impacts, direct ou non, pouvant se propager à travers tous les artefacts du cycle de développement. Les éléments de l'évolution intra-développement sont présentés dans la figure 1.1.

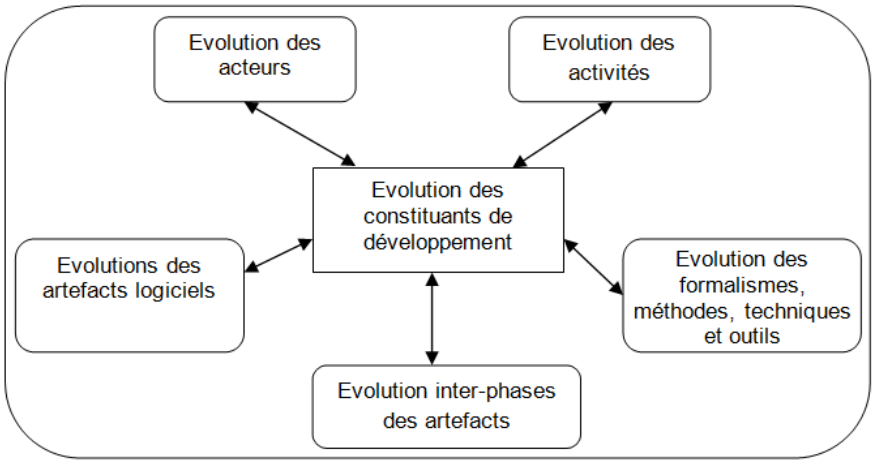


Figure 1.1: Les éléments de l'évolution intra-développement

Une évolution intra-développement est réalisée par une suite d'opérations de modification menées dans une ou plusieurs phases du cycle de vie. Ces opérations, qui s'effectuent selon un certain ordre, sont de deux types :

– Les modifications initiales ou modifications de départ, visant à atteindre un objectif d'amélioration structurelle, logique, fonctionnelle, comportementale, ou qualitative d'un artefact logiciel.
– Les modifications conséquentes, devenues nécessaires après une modification initiale. Ces dernières effectuées pour réparer un effet négatif, prévu ou non, causé par une modification initiale. Une opération conséquente typique visant à traiter l'incohérence générée par une opération initiale, entre un artefact logiciel et ce qui lui correspond comme différentes descriptions à travers les phases concernées du cycle de développement.

1.3.1.1 Evolution des constituants de développement

Ce sont les évolutions que peuvent subir les constituants des phases du développement. Hormis l'évolution des artefacts, l'évolution des autres constituants relève du domaine de la modélisation des procédés. En considérant des relations principales de dépendance entre constituants du développement, nous pouvons identifier les chemins principaux de propagation d'impacts entre ces constituants et les artefacts logiciels.

1.3.1.2 Evolution des acteurs

L'évolution des acteurs du développement se traduit par un changement, plus ou moins important, de l'ensemble des acteurs participant au développement. Ce changement peut porter sur leurs rôles respectifs dans le développement, le taux de contribution individuelle de chaque acteur par activité, les relations de hiérarchie ou de coopération entre acteurs, etc.

1.3.1.3 Evolution des activités

L'évolution des activités porte sur les activités planifiées ou en cours. Le déroulement d'une activité est principalement caractérisé par son état qui peut avoir des valeurs de type énumératif (non prévue, prévue, planifiée, en cours, suspendue, en attente de ressources, en attente de date d'échéance, en attente de décision, terminée, avortée, validée, etc.).

L'ensemble des activités en cours et les activités planifiées peuvent subir les changements de base suivants :

- Insérer une activité, ce qui a pour effet de repousser toutes les activités du planning, figurant désormais en aval de la nouvelle activité.
- Annuler une activité prévue, ce qui peut rendre moins difficile l'avancement dans le planning de toutes les activités qui figuraient en aval de l'activité supprimée.
- Substituer une activité prévue par une autre. Ceci revient à l'application successive des deux opérations précédentes.
- Modifier une activité prévue.
- Suspendre une activité en cours.
- Reprendre une activité suspendue.
- Modifier la planification d'une activité dans le temps (rapprocher ou différer).

1.3.1.4 Evolution des formalismes, méthodes, techniques et outils

Les ensembles de formalismes ou langages, de méthodes, de techniques et d'outils utilisés dans les phases du développement peuvent évoluer par des opérations d'adoption ou d'abandon d'un formalisme (respectivement une méthode, technique, ou outil) utilisé en développement. Ceci revient à :

- Adopter un formalisme non encore utilisé.
- Changer de formalisme utilisé au profit d'un autre.
- Intégrer ou mettre en correspondance deux formalismes.

Les mêmes opérations peuvent également être appliquées aux méthodes, aux techniques et aux outils. Les outils peuvent aussi bien être matériels que logiciels.

1.3.1.5 Évolution des artefacts logiciels

Dans la première famille d'évolutions (intra-développement), plusieurs types peuvent être définis et dont on distingue :

- L'évolution de la spécification des besoins : dans un projet de développement, les besoins à satisfaire sont exprimés dans le cahier des charges de l'application. Le contenu de ce cahier, satisfaisant pour une certaine durée, peut progressivement accuser une différence, qui peut devenir critique, avec l'expression des besoins engendrés par un monde évolutif. Il arrive souvent que ces besoins soient effectifs dès le début du projet, mais ils sont constatés tardivement et ne sont donc pas spécifiés dans le cahier des charges (signé auparavant). Ceci correspond à une prise en compte partielle, mais fréquente, des besoins réels. Dans un autre cas,

faute d'une bonne estimation, ces besoins qui ont été jugés secondaires en début du projet, ont pu regagner de l'importance au cours du développement. Dans les deux cas, une réactualisation de l'expression des besoins peut devenir nécessaire et provoquer une renégociation du cahier des charges entre les différentes parties du développement. La décision de faire évoluer l'expression des besoins implique souvent une réévaluation des coûts et de délais du développement. Elle rend particulièrement nécessaire la mise à jour d'un sous ensemble de composants déjà développés.

– Évolution des spécifications fonctionnelles : elle peut être déclenchée lors d'une constatation d'incomplétude ou d'incohérence des spécifications fonctionnelles par rapport à l'expression déjà acceptée et stabilisée des besoins. Elle peut aussi être réalisée suite à une évolution survenue en amont de l'expression des besoins ou en aval par des évolutions de la conception. Elle peut de même être exigée à l'issue d'une validation des spécifications par rapport à des critères de qualité relatifs au formalisme (ou à la méthode) de spécification fonctionnelle en vigueur. Enfin, une évolution des spécifications peut s'effectuer par l'adoption d'un nouveau formalisme, en transformant l'expression courante des spécifications en expression plus formelle.
– Évolution de la conception : cette évolution peut être conséquente, réalisée pour rétablir la cohérence avec les autres descriptions lors d'une évolution déjà entamée, en amont ou en aval de la conception. Elle peut aussi être causée par une décision d'amélioration de la conception par changement de formalisme ou de méthode.
– Évolution du codage ou programmation : elle peut être conséquente à une évolution entamée en dehors de la phase de programmation, ou une réponse à des exigences qualitatives mise en application en cours du développement. Une des sources principales de l'évolution de la programmation est le test individuel des composants modulaires ainsi que le test global lors de l'intégration des composants. Ces tests peuvent révéler dans le fonctionnement d'un composant, un comportement ou caractéristique qualitatif différent de ce qui est attendu dans le cahier des charges.

1.3.1.6 Evolution inter-phases des artefacts

En considérant le cycle de développement, une modification ayant lieu dans une phase quelconque, peut être initiale, ou conséquente à une modification effectuée en amont ou en aval de cette phase. Au sein d'une même phase, on considère deux familles d'évolutions : ce sont les évolutions intra-développement et les évolutions en post-développement.

1.3.2 Les évolutions post-développement

Les évolutions en post-développement, comme leur nom l'indique, ont lieu après la mise en exploitation du logiciel. Elles incluent les différents types de maintenance du logiciel. Le terme évolution a été préféré à celui de la maintenance en raison de la nette distinction entre la maintenance du matériel et ce qu'on a coutume d'appeler la maintenance des logiciels. On y distingue deux grandes familles exigeant chacune des modifications différentes en volume et en objectifs à atteindre. Ce sont les grandes évolutions que l'on qualifie avec le terme *Refaire* et les évolutions moins importantes en volume que l'on qualifie avec le terme de *Améliorer*.

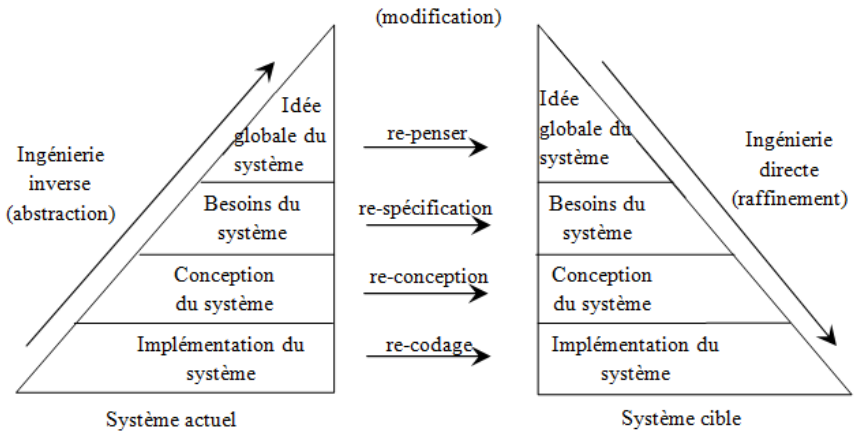


Figure 1.2: Modèle général de la réingénierie du logiciel source originale à (32)

1.3.2.1 Typologie des évolutions de type « Refaire »

Ces évolutions correspondent à des opérations de grande envergure visant tout un ensemble de composants issu d'une ou de plusieurs phases du cycle développement. En

général, ces évolutions rentrent dans le cadre d'activités dites de ré-ingénierie. La ré-ingénierie consiste à respécifier et reconcevoir différemment le logiciel existant pour en développer un autre avec des améliorations structurelles, fonctionnelles, comportementales ou qualitatives. Dans (32), Byrne identifie des types de changements rentrant dans le cadre de la ré-ingénierie et fournissant une assise conceptuelle à ce concept. Ces types de changements sont résumés par la figure 1.2. Ce sont :

- Le re-codage : il s'agit d'un changement qui se limite aux artefacts constituant le code source. Cela peut correspondre à une traduction des élements du code d'un langage à un autre, la restructuration ou le refactoring pour améliorer certains critères de qualité du code source tels que le respect des standard de nommage des variables, une meilleure encapsulation des classes, une restructuration du flot de contrôle pour réduire la complexité cyclomatique, etc.
- La re-conception : ces changements affectent en excluisivité des éléments de la conception du systèmes pour entre autres, en améliorer les qualités. Cela peut concerner un changement de langage ou formalisme de conception (du modèle entité-association au diagramme de classes UML) ou une modification d'un diagramme de classes UML pour améliorer son évolutivité (introduction des interfaces, application de design patterns, etc.). Ce type de changement peut concerner également des modifications d'architecture en terme de changement de la structure modulaire (diagramme de composants UML) du système ou de son déploiement (diagramme de déploiement UML).
- La re-spécification : il s'agit de changements affectant l'expression des exigences ou besoins. Cela peut consister en un simple changement du formalisme support tels que l'adoption des diagrammes de use case UML ou de changement de ces exigences elles mêmes.
- Re-penser : ce type de changement est beaucoup plus profond et peut induire le développement d'un système foncièrement différent. Il peut s'agir par exemple d'une opération d'inspiration du système existant pour en développer par exemple, un autre dans un domaine différent. C'est le cas par exemple, d'utiliser un système de réservation d'une compagnie aérienne pour en développer un autre utilisable dans le domaine des chemins de fer ou de l'hôtellerie, etc.

Il est tout à fait possible qu'une ré-ingénierie soit circonscrite à un ensemble restreint de phases du cycle de développement (conception + codage par exemple) ou concerner

toutes les phases du cycle de vie. On peut donc évoquer les termes de ré-ingénierie partielle ou totale du système logiciel.

Une réingénierie totale du système peut se décliner en deux grande activités :

- Une rétro-ingénierie qui consiste à partir des éléments ou artefacts de plus bas niveau d'abstraction tels que les code sources pour produire des artefacts de niveau d'abstraction plus élevés. Cette activité (retro-enginierie) se justifie par le fait dans la majorité des cas, les documents ou artefacts de haut niveau d'abstraction sont obsolètes voir même inexistants. Il est donc nécessaire de produire des abstractions de haut niveau qui correspondent à l'état actuel du système à faire évoluer pour mieux le refaire ou le faire autrement. Dans (33), Chikofsky et Cross ont décrit une taxinomie des activités rentrant dans le cadre de la rétro-ingénierie ou ingénierie inverse.
- Le « forward engineering »ou « ingénierie directe »qui a pour but de partir des abstractions de haut niveau obtenues par la rétro-ingénierie pour en faire de nouvelles qui serviront par la suite à dériver le nouveaux système. Il est possible que le nouveau système soit obtenu par des techniques manuelles ou par des techniques semi-automatiques telles que celles préconisées par l'ingénierie dirigée par les modèles. Dans ce dernier cas, on part des abstraction de haut niveau qu'on appelle des PIM (Platform Independent Models) pour obtenir des abstractions de niveaux plus bas qu'on appelle PSM (Platform Specific Model) en utilisant des transformateurs de modèles.

1.3.2.2 Typologie des changement de type perfectif « Améliorer »

Dans ce type de changements, le but n'est pas de redévelopper un nouveau système à partir d'un plus ancien mais de faire évoluer un système existant en le « maintenant ». En se basant sur des travaux plus anciens de Lientz et Swanson (34), le *ISO/IEC standard for software maintenance* (35) propose quatre catégories de changements de type à améliorer et qui sont qualifiés d'actions de maintenance. Ce sont :

- L'évolution ou maintenance corrective : il s'agit principalement de modifications ayant pour but la correction d'anomalies ou de défauts.
- L'évolution ou maintenance adaptative : ce sont des modifications qui consistent à porter le logiciel, d'un environnement d'accueil vers un autre. Le terme environnement d'accueil peut désigner aussi bien un système d'exploitation, qu'un serveur

d'application ou un environnement réseaux particulier (client/serveur, multi-tiers, etc.).

- L'évolution ou maintenance perfective : ces changements désignent toutes les actions ayant pour but d'améliorer des critères de qualité telles que les performances, l'évolutivité, la facilité d'utilisation, la montée en charge, etc.
- L'évolution ou maintenance préventive : il s'agit d'actions destinées à prévenir les dégradations qualitatives du logiciel. Un des exemples serait d'ajouter des traitements d'exception pour éviter des dysfonctionnements graves et des indisponibilités du système en lui permettant par exemple de fonctionner en mode dégradé au lieu d'un arrêt brutal à la suite d'anomalies.

Dans (36), Chapin *et al.* proposent des extensions du standard *ISO/IEC standard for software maintenance* (35) en prenant en compte les aspects non techniques comme la documentation, le consulting, la formation, etc. Dans (37), Buckley *et al.* proposent une taxinomie des changements du logiciel basée sur plusieurs dimensions.

Les quatre types d'évolutions évoqués ci-dessus affecte plusieurs aspects ou vues du logiciel. Ces vues ou aspects concerne la structure, les fonctionnalités, la qualité, et le comportement. L'évolution des fonctionnalités traduit un changement au niveau du contrat d'utilisation du logiciel. Ce type d'évolution constitue souvent le point d'entrée du processus d'évolution puisque les besoins en changement émanent souvent des utilisateurs et se traduisent par une modification ou changement des fonctionnalités rendues par le logiciel. La structure du logiciel peut modéliser les différents artefacts qui le composent et mettre en relief les liens sémantiques qui les relient. Une évolution de cette structure peut être due à l'évolution des fonctionnalités mais elle peut également traduire un changement de l'architecture du logiciel et de sa qualité.

L'évolution de la qualité se confond dans la majorité des cas avec la notion d'évolution perfective. Elle peut engendrer des changements au niveau de la structure du logiciel. Le comportement du logiciel traduit l'expression dynamique de l'architecture du logiciel. Ce comportement peut s'exprimer par des diagrammes d'états ou des diagrammes de séquences ou même des réseaux de Petri qui représentent aussi bien le comportement individuel des artefacts (digrammes d'états) ainsi que leurs interactions. Un changement du comportement peut consister en un redéploiement des artefacts ou un changement des canaux de communication, etc.

Dans le cadre de cette thèse nous nous intéressons particulièrement à l'évolution de la structure, de l'architecture et de la qualité du logiciel.

1.4 État de l'art en évolution du logiciel

Dans cette section, nous tentons d'élaborer une synthèse des travaux rentrant dans le cadre de l'évolution du logiciel. Nous focaliserons sur ceux concernant la gestion de l'impact du changement des codes sources qui est le champ de recherche privilégié de nos travaux dans le domaine (7, 9, 10, 38). La figure 1.3 résume les principales activités rentrant dans la définition des processus automatisés ou semi-automatisés de gestion du changement des applications informatiques. Ce sont notamment les activités suivantes :

- La récolte et abstraction de l'information : le but de cette activité est de construire un référentiel des artefacts logiciels sujets du changement. Ce référentiel doit être exploitable par des outils informatiques et donc se présenter comme une sorte de base de données. Ces référentiels peuvent se limiter à des artefacts d'une seule phase du cycle de vie et d'un même et seul langage, d'artefacts de plusieurs phases du cycle de vie et de plusieurs langages, etc.
- L'exploration des artefacts : il s'agit d'outils permettant d'explorer les différents artefacts stockés dans le référentiel dans le but d'une meilleure compréhension des applications informatiques. Cela peut se faire par des outils de visualisation de graphes, des outils d'extraction d'abstraction de haut niveau, des langages de requêtes, etc.
- L'analyse et propagation de l'impact du changement : méthodes algorithmiques, utilisation des systèmes à base de connaissances, utilisation des modèles à base de réécriture des graphes, etc. Ce sont des outils permettant de mesurer l'impact du changement, Il s'agit principalement de déterminer, *a priori*, les effets directs et indirects d'une opération de changement. En général, Cela est possible en exploitant l'ensemble des liens ou relations qui existent entre les différents artefacts du logiciels. Ces liens incluent des relations de mise en correspondance (ou lien de projection) entre artefacts du logiciels. Ces outils sont généralement implantés par des approches algorithmiques, des approches à base de règles ou de système à base de connaissances ou bien des systèmes de réécriture de graphes, etc.

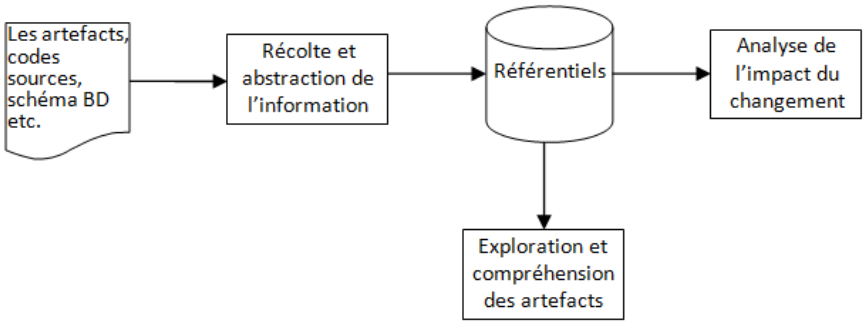


Figure 1.3: synthèse des travaux concernant la gestion de l'impact du changement des codes sources

Nous allons dans ce qui suit présenter les différentes approches de construction des référentiels des artefacts logiciels et plus particulièrement les approches de définitions des schémas de ces artefacts.

1.4.1 Modélisation des référentiel d'artefacts logiciels

En ingénierie des logiciels, la modélisation est un processus qui construit des abstractions du système en omettant les détails jugés non nécessaires, dans un premier temps, et qui permet d'automatiser le traitement des artefacts ainsi modélisés. Les Modèles peuvent être descriptifs, s'ils représentent des artefacts déjà existants ou prospectifs, s'ils représentent des schémas d'artefacts à construire.

Des bases de données peuvent être utilisées pour stocker les informations décrivant les artefacts logiciels extrait des systèmes légataires, etc. Des schémas sont nécessaires pour la manipulation des différents artefacts souvent hétérogènes et complexes constituant le logiciel. Il existe des schémas correspondant aux langages de programmation textuels, et d'autres pour des langages visuels, etc. Des schémas peuvent concerner des artefacts de niveau d'abstraction et/ou de granularité plus ou moins importants. L'un des principaux concepts utilisés pour la modélisation des artefacts logiciels est le concepts de graphes, que nous allons détailler dans les deux prochaines sections.

Les plusieurs approches de modélisation des artefacts logiciels ont été mise en œuvres notamment dans le cadre de la construction d'outils pour l'IDM (Ingénierie Dirigée par le Modèle). Cela inclut des techniques permettant la migration de systèmes légataires à

des systèmes construits selon l'architecture SOA (Service-Oriented Architecture) [1] [2], par exemple. Les bases de données relationnelles sont utilisées pour recueillir des informations de logiciels et l'organiser comme un ensemble de tableaux, décrite formellement. Le but d'un modèle relationnel ou de schéma a été de fournir une méthode déclarative pour analyser les informations relatives au logiciel. Les documents XML permet de stocker des informations de logiciels en format XML. Ce type de données peut aussi consulter et manipuler comme une base de données relationnelle. Les bases de données orientées objet représentent et stockent des informations de logiciels en tant que collection d'objets. Le système de gestion de base de données objet a une puissance intrinsèques à la gestion de bases de données pour les objets structurés en format des graphes.

Les techniques pour construire les référentiel d'artefacts logiciels utilisent généralement des systèmes de requêtes sur des graphes et des systèmes de réécritures de graphes. Depuis plusieurs décennies déjà les compilateurs on utilisé les structures d'arbres pour représenter les programmes. Cependant, les arbres sont limités en matière de représentation de différent artefacts logiciels. Le concept de graphe, plus particulièrement celui des graphes typés et attribués, est un moyen générique, semi-automatique, et plus adéquat que les arbres pour représenter la structure des artefacts ainsi que des informations supplémentaires les concernant (à travers les attributs).

1.4.2 Elaboration de l'abstraction et récolte de l'information

Dans les différents travaux sur ce sujet, une préoccupation constante concerne l'abstraction de l'information (39, 40). En effet, il est extrêmement difficile de gérer l'impact du changement des applications informatiques en se basant exclusivement sur leur code source. Il est alors nécessaire de disposer d'une représentation plus abstraite des l'information permettant une compréhension de la structure de ces applications pour faciliter l'expression et la gestion des opération de changement. La modélisation adoptée préconise :

1. La visualisation d'un graphe résumant la structure de l'application et cela à l'aide d'outils d'aide à la navigation dans ce type de graphes.

1. http://www.service-architecture.com
2. http://msdn.microsoft.com/en-us/library/aa480021.aspx

2. La génération de documents de haut niveau comme ceux de spécifications et de conception en appliquant des techniques de l'ingénierie inverse pour identifier les relations de traçabilité entre les codes sources et ces documents, etc.

D'un point de vue technique, l'abstraction de l'information se manifeste par la construction d'un référentiel d'artefacts logiciels de différents niveau d'abstraction. Le référentiel est une base de données qui peut être construite au tour d'un SGBD relationnel ou objets, d'un système de gestion d'hyper-documents XML représentant des graphes (GXL(41)) ou des modèles dans le sens de l'OMG (XMI(42)).

Nous allons dans ce qui suit présenter les différents approche du schema de la structure de référentiel.

1.4.3 Exploration et compréhension des artefacts

Afin de mieux traiter les informations hétérogènes de logiciels, on peut définir des schémas. Il y en a des schémas pour les langages de programmation textuels, et visuels. Différents schémas peuvent être dérivées en fonction de leur effet sur plusieurs niveaux de granularité d'artefacts logiciels. Ces schémas peuvent aussi être combinés à de plus grands schémas intégrés. Cette étude est abordée dans la discipline d'ingénierie du méta-modèle. Compte tenu d'un schéma graphique, il existe des outils logiciels s'y conformant. Ces outils sont capables d'extraire les faits pertinents de logiciels (par exemple par parseurs ou analyse) et les stocker dans un référentiel graphique. Ensuite, plusieurs services peuvent être appliquées sur ces dépôts, comme ailleurs, l'analyse, l'abstraction et la transformation etc. pour faciliter l'utilisation de ces graphes. Ces services peuvent être spécifique d'une application ou bien générique, en fonction de leur objectif. Il y a deux type d'approche principale pour l'analyse de données. La première est à la base des algorithmes ou les approches traditionnelles de programmation de logiciels, et la deuxième à la base de règles ou les approches de programmation déclarative.

La communauté de génie logiciel utilise également les langage de requête pour gérer tels graphes. Mais, il exige les développeur de maîtriser les langages et les outils associé. Une approche déclarative de langage de requête peut éviter une grande partie de ce travail. Une comparaison détaillée sur des langages de requête utilisés pour la réingénierie et leurs caractéristiques est représenté dans (43). Les langages de requêtes sont forts, mais en même temps, ils peuvent devenir complexes et difficiles pour la récupération des informations. La taille et la complexité des requêtes dépendent de la structure et

l'exhaustivité des graphes. Ce dépend aussi sur le méta-modèle et les parseurs qui sont utilisés pour extraire les graphes à partir du code source. Par conséquent, nous pensons que l'approche à base de règles peut donner une solution pour gérer des relations complexes entre les artefact, fournissant des informations de traçabilité d'impact du changement.

L'utilisation des *TGraphs* ou graphes typés est une des approches de modélisation basée sur une combinaison de langages de requêtes, au sens documentaire ou bases de données du terme, et la notion de graphes. Cette approche met en œuvre un certain nombre de langages de requêtels tels que : XQuery [1], XPath [2], SPARQL [3], etc. *Graph Repository Query Language* (GReQL) (44, 45) est un des langages de requêtes qui opère sur les TGraphs(46). Ce langage a la particularité d'être également utilisé de façon combinée avec le langage GReTL (Graph Repository Transformational Language(47)) pour mettre en œuvre des transformations de graphes. Il est ainsi possible, tout en explorant un Tgraph, de pouvoir y opérer des transformations.

Dans cette approche, le but est de profiter des capacités de langages de requêtes pour explorer des artefacts logiciels formalisés par des graphes typées, attribuées, orientées et ordonnées.

1.4.4 Analyse de l'impact du changement

Des techniques de visualization graphique ont été proposées dans la littérature pour la compréhension des systèmes logiciels de grande taille. Dans (48), les auteurs ont défini le langage *Graphlog* pour ces systèmes ont simplifiant notamment les liens de dépendance entre les artefacts logiciels et cela en utilisant quelques transformations comme la suppression des cycles dans la graphe des dépendances, etc. D'après, ils tentent de concevoir une structure efficace, recouvrement le code par le biais des techniques graphiques.

1. XQuery 1.0 : An XML Query Language, W3C Recommendation. Dec. 2010 http://www.w3.org/TR/xquery/
2. XML Path Language 2.0 W3C Recommendation. Dec. 2010 http://www.w3.org/TR/xpath20/
3. SPARQL Query Language for RDF. W3C Recommendation. Jan. 2008 http://www.w3.org/TR/rdf-sparql-query/

1.4.4.1 Dependency Structure Matrix (Matrice de dépendance Structurelle)

La « Dependency Structure Matrix (DSM) » est une technique permettant de découvrir des dépendances non explicites pouvant exister entre les artefacts logiciels (49, 50). Cette technique propose une représentation matricielle compacte des différents artefacts, qui sera exploitée de façon visuelle par l'utilisateur pour précéder à une décomposition modulaire du système logiciel. Il s'agit en effet de représenter les artefacts comme des indices des lignes et des colonnes de la matrice, les dépendances apparaissent au niveau de cellules. En s'appuyant sur un outil de visualisation, l'utilisateur procède au regroupement des artefacts en modules et de proche en proche une structure du logiciel plus compacte et plus simple à comprendre. Cette matrice avec les décompositions dont il est possible d'en déduire peuvent servir alors à bien comprendre le logiciel et bien cerner les impacts du changement à des différents niveaux d'abstractions. Dans (51), les auteurs donnent un aperçu de la technique qui est supportée, entre autre, par le système *Lattix LDM*[1] qui est un produit industriel utilisé dans les domaines du reverse engineering et celui de l'analyse de l'impact du changement du logiciel. Le but de cet outil est également de proposer des opérations de refactoring au niveau système logiciel pour éliminer les dépendances jugées incohérentes et source de manque d'évolutivité du système. Cet outil peut être également utilisé pour vérifier l'adéquation du système réel avec son architecture adoptée.

1.4.4.2 La modélisation réflexive

A l'image de la technique DSM, la modélisation réflexive a pour but la comparaison de la « modélisation » (architecture, structure) du système actuel avec sa modélisation adoptée. Cette technique (52) développée par G. Murphy consiste principalement à extraire la modélisation effective du système en essayent de trouver de façon semi-automatique, des correspondances entre les artefacts effectifs du système et ceux du modèle théorique ou mental fourni par le chargé d'évolution. La correction progressive des incohérence entre les deux modèles permettra donc l'amélioration du modèle théorique pour qu'il reflète avec précision l'état actuel du système. Cette technique soit donc dans la compréhension des grandes systèmes logiciels et par conséquent à l'étude de l'impact du changement à différents niveau d'abstraction. En effet, la technique va aider à la

1. http://www.lattix.com

découverte de liens de traçabilité entre les artefacts de différents niveaux d'abstraction qui seront exploités dans l'analyse d'impact du changement et sa propagation.

1.4.4.3 Utilisation des styles d'évolution

O. Le Goear, M. Oussalahl et D. Tamzalit (53, 54) préconisent, dans leurs travaux de recherche une gestion différente de l'évolution des logiciels moyennant la définition puis la réutilisation de styles génériques d'évolutions architecturales des applications. Ces styles sont classés et sauvegardés pour une utilisation future par extraction puis instanciation du style d'évolution choisi. Les travaux proposent un méta-modèle d'évolution à base de styles appelé SEAM (Style-based Architectural Evolution Model). SEAM est voulu générique, adaptable à diverses architectures pouvant être spécifiées avec l'un ou l'autre des ADLs. Le concept fondamental de l'approche est le style d'évolution exprimant chaque fois une opération de modification dans un large éventail de changements architecturaux possibles. Tout style individuel d'évolution est identifié par un nom puis une entête spécifiant les pré conditions et l'élément architectural ciblé, puis la compétence du style composée de son flot de contrôle et des éventuelles invocations d'autres styles.

L'objectif majeur de SEAM est de fournir aux utilisateurs du modèle et plus particulièrement les architectes chargés de l'évolution, des opérations d'évolution encapsulées dans un ensemble capitalisant des procédures de changement déjà validées sur des éléments architecturaux. Ayant une démarche basée intégralement sur la réutilisation des styles d'évolution architecturaux, la qualité de l'approche dépend intégralement de la qualité de la bibliothèque et sa richesse en styles réutilisables d'évolution. En conséquence, une analyse d'impact purement qualitatif se trouve en dehors des objectifs visés par SEAM.

1.4.4.4 Change-and-fix

Rajlich (55) a introduit l'approche dite « change-and-fix » basée sur une réécriture du graphe des dépendances du système logiciel durant le processus d'évolution. Cette approche modélise la propagation de l'impact du changement par une série de « snapshots » du graphe de dépendances. Chaque snapshot, ou description instantanée, représente l'état du graphe de dépendances à un moment donnée. En effet, une modification

initiale d'un nœud du graphe résulte en un ensemble de nœuds qui deviennent inconsistants par rapport à des contraintes préétablies sur ce graphe. L'approche change-and-fix va consister donc à générer un impact vers tous les liens qui sont devenus inconsistants pour les rendre consistants en appliquant des opérations de changements correcteurs (fix) et ainsi de suite.... Par exemple, la suppression d'une méthode d'une classe donnée va rendre inconsistants toutes les méthodes qui l'appellent. Il faudra donc corriger cette inconsistance en supprimant cet appel, en le remplaçant par un code ou en créant une méthode équivalente, etc. Et chacune des opérations ci-dessus peut violer une contrainte. Ainsi la création d'une nouvelle méthode peut provoquer un problème d'homonymie avec une méthode existante, etc.

L'approche change-and-fix considère une relation générique dite de dépendance et ne prend donc pas en compte les sémantiques particulières des différentes relations inter-artefacts dans l'évaluation et la caractérisation de l'impact. La décision est en effet laissée à l'utilisateur qui doit évaluer lui-même. Cet impact et des opérations de correction. De plus, cette technique ne prend pas en compte les différents niveaux d'abstraction. Ceci peut introduire une difficulté dans la compréhension de la structure du logiciel à modifier durant le processus. Cependant dans (55) les auteurs proposent une adaptation de l'approche change-and-fix qui tient compte de niveaux d'abstraction et de granularité en partitionnant les dépendances en définition, ou composition, et utilisation.

1.4.4.5 Les systèmes à base de connaissances

Les approches utilisant les systèmes à base de connaissances ont pour but de mettre en œuvre des systèmes d'assistance à la compréhension du logiciel et à son évolution. Ces systèmes se distinguent principalement par la nature des connaissances contenues dans leurs bases. Certains représentent une connaissance dite de type « patterns » ou « patrons » et qui est indépendante du domaine (56, 57). D'autre systèmes représentent connaissance liée au domaine et à la conception du système logiciel.

Nous pouvons également distinguer ces systèmes selon le degré d'automatisation qu'ils procurent. Ainsi, certains sont des systèmes d'assistance complètement automatisés alors que d'autres sont plutôt semi-automatiques. La compréhension automatique des programmes, telle que proposée par Quilici (58), extrait des information relatives à la conception du système en essayant de mettre en correspondance des éléments du code source avec un ensemble de « cliches » déjà enregistrés.

Les techniques semi-automatiques disposent d'un ensemble de pattern pré-codés qu'elles utilisent comme guide pour aider l'ingénieur à produire des représentations du haut niveau du système logiciel et ceci de façon incrémental et séquentielle (59, 60). L'approche de raffinements synchronises (61) est un exemple de technique manuelles.

1.4.4.6 Les outils d'analyse d'impact du changement

Plusieurs outils ont pour fonction l'extraction de relations de dépendances. En effet, ces relations sont le vecteur de la conduction et donc de la propagation de l'impact. Nous allons donc présenter deux de ces outils qui sont JRipples (62) et Chianti (63) qui sont les plus significatifs par rapport à l'approche que nous avons adoptée (7).

- JRipples (62) est un plug-in Eclipse [1] permettant la compréhension des programmes Java dans le cadre d'un processus de changement incrémental. Cet outil qui propose deux formes d'assistance consiste en une gestion en phases des différentes activités de l'analyse d'impact du changement. La seconde forme d'assistance est qu'il permettant la mise en œure de changements (exécution des changements) et leurs propagation de façon iterative. Cet outil est basé sur la notion de graphe de dépendance du système logiciel (64). Les fondements théoriques de JRipples sont définis dans (65). JRipples traite les artefacts appartenant aux codes source Java et considère les classes java comme les entités de première classe cibles des opérations de changement. JRipples détermine les inconsistances du système logiciel modifié et marque les classes devant être « visités » à reconsidérer par l'ingénieur.
- Chianti (63) est un outil d'analyse d'impact du changement d'applications Java. Il a également été implémenté comme un ensemble de plug-ins Eclipse. Il permet d'analyser deux version d'un programme (la version initiale et la version modifiée) et en extrait de façon automatique les différentes opérations atomiques de changement que le logiciel a subi. Par la suite, Chianti propose une visualisation du logiciel en termes de graphes et une analyse des impacts du changements extraits par l'analyse de deux versions.

1. http://www.eclipse.org

1.5 L'approche de multi-modélisation pour l'évolution de logiciels

Nous proposons une approche de multi-modélisation pour analyser l'impact du changement et assister à la compréhension des effets des modifications des systèmes logiciels distribués. Ces modèles représentent l'architecture des logiciels, leurs structures et leurs caractéristiques qualitatives et devraient satisfaire les différentes fonctionnalités nécessaires à la mise en œuvre du contrôle de l'évolution du logiciel. Ils sont validés par une plate-forme opérationnelle intégrée à l'environnement de développement Eclipse en forme de plug-ins.

L'approche de modélisation proposée inclut trois descriptions principales du logiciel :

- La description structurelle
- La description architecturale
- La description qualitative

La description structurelle repose sur une stratification des différents artefacts logiciels en couches ou niveaux abstro-granulaires et sur l'identification et la qualification de relations d'interdépendance responsables de la conduite de l'impact du changement. Cette description peut être considérée comme une transcription en un modèle automatiquement exploitables et explorables de la variété des artefacts. Ainsi, il est possible d'en déduire des faits qui alimenteront un système à base de connaissances implémentant de façon déclarative et extensibles différentes stratégies de propagation de l'impact. La description structurelle, basée essentiellement sur la notion de graphes sert également de référentiels pour des outils de visualisation et d'aide à la compréhension du logiciel et de son évolution.

La description architecturale fournit une modélisation de haut niveau du logiciel et focalise sur des aspects liés à son organisation en modules et/ou composants et sur le comportement de ces composants. Elle participe à la compréhension du logiciel et de son comportement et sa prise en compte par notre approche est justifiée par notre volonté de gérer l'impact du changement selon des points de vues différents. Cette description est mise en correspondance avec la description structurelle de sorte qu'il soit possible de répercuter les impacts d'une modification de la structure du logiciel sur son architecture et vice versa.

La description qualitative fournit une connaissance sur des aspects liés à la qualité du logiciel. En d'autres termes elle organise de façon cohérente et également exploitable une connaissance des facteurs clés de la qualité tels que l'utilisabilité, l'évolutivité, etc. Notre objectif est de fournir un moyen opérationnel de modéliser la qualité, en nous inspirant des modèles de la littérature, mais également de fournir un mécanisme permettant d'étudier l'impact des modifications sur la qualité future du logiciel.

Pour la définition de la description structurelle, nous nous sommes inspirés des travaux développés par H. Basson (66). Il a définit, dans le cadre de ces travaux, un modèle appelé Modèle Générique de Composants Logiciels qui se décline en deux sous-modèles : le modèle structurel de composants logiciels ou MSCL ayant pour finalité la représentation de la structure des constituants du logiciel et le modèle qualitatif des composants logiciels ou MQCL. Ce dernier formalise la qualité d'un logiciel en termes de graphes typés reliant les attributs de la qualité et ayant une correspondance avec les éléments structurels. Le schéma de la figure 1.4 résume les principales contributions de notre approche. Ce schéma présente les différentes modélisations que nous introduisons et qui sont instanciées pour les principaux langages utilisés dans le processus de développement des logiciels. Les modèles et leurs instances sont alors utilisés par un système à base de connaissances implémentant les différentes stratégies de gestion de l'impact des modifications. Le prochain chapitre décrit avec plus détails, l'approche de multi-modélisation au niveau de la structure du logiciel.

1.6 Conclusion

L'évolution est un processus continu nécessaire pour l'adéquation des logiciels par rapport à l'évolution des besoins ou à l'évolution technologique. Le contrôle de l'évolution a pu acquérir une attention grandissante proportionnelle à l'importance d'éviter des situations, inattendus et difficiles, de fonctionnement du logiciel, causées par des modifications dont l'impact a été insuffisamment analysé.

Dans ce chapitre, nous avons introduit l'évolution des logiciels en détaillant la typologie de l'évolution de logiciels comme une activité intra et post-développement du cycle de vie du logiciel. La mise en IJuvre de des opérations de changement d'une évolution reviennent à « refaire » ou « améliorer » partiellement ou totalement un logiciel d'un certain point de vue. La refonte implique principalement une réingénierie tandis

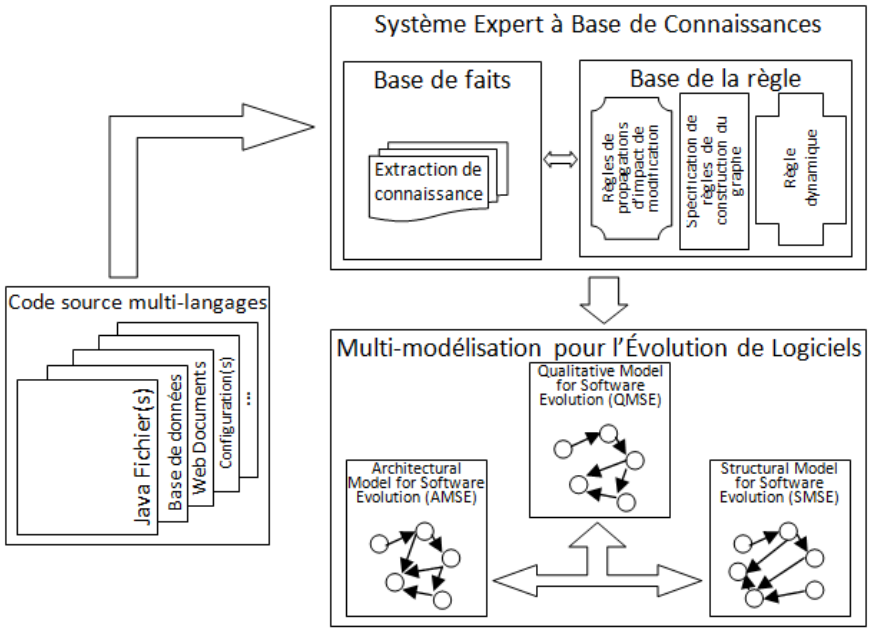


Figure 1.4: Schéma global du système à base de connaissance sur l'approche de multi-modélisation

que l'amélioration revient à une maintenance corrective, curative, perfective, etc., du logiciel. Le chapitre a également abordé l'état de l'art des approches et techniques les plus largement connues dans la communauté du domaine. Un bref aperçu de l'approche modélisation des logiciels pour le contrôle d'évolution a ensuite été donné pour servir de référence au contenu de la base de connaissances élaborée pour l'aide à la compréhension et à l'analyse d'impact des modifications. Cette approche de la modélisation intégrée du logiciel, argumentée dans sa constitution et son prototypage, a été illustrée pour des deux points conceptuels majeurs : le point de vue structurel et le point de vue qualitatif. La validation a porte sur l'aptitude effective de la modélisation adoptée à identifier les chemins du flux de propagation d'impact du changement.

Deuxième partie

Modélisation Structurelle

Chapitre 2

Modélisation et Analyse du Changement de la Structure du Logiciel

2.1 Introduction

Dans le cadre de l'évolution du logiciel, la modélisation consiste en un processus d'abstraction visant à éliminer les détails superflus. Les modèles de logiciels peuvent décrire aussi bien des artefacts existants ou représenter des schémas d'artefacts à développer. Le terme artefact désigne toute entité rentrant dans le cadre du développement du logiciel. Cela peut désigner une partie du code source, un schéma de base de données, un plan de test, un document de planification des tâches, etc. La modélisation de l'évolution du logiciel peut exiger une compréhension exhaustive de l'ensemble des artefacts composant le logiciel. Traiter le changement du logiciel peut donc nécessiter la compréhension de divers aspects du logiciel incluant sa structure, son architecture, ses fonctionnalités, son comportement et sa qualité. Tout changement du logiciel peut donc avoir des impacts selon chacun de ces aspects. Ainsi, la modification d'une classe peut avoir des impacts sur la structure du logiciel à savoir l'interrelation des différentes classes par divers liens sémantiques ainsi que les relations qui peuvent exister entre les classes des codes sources et celle de la conception et également entre ces classes et les tables des bases de données que ces classes manipulent ou qui gèrent la persistance des objets de ces classes par le biais de divers sortes de médiateurs (ORM, JDBC, etc.).

Ainsi, l'impact se propage sur les différents constituants du logiciel à travers les différents liens ou relations d'interdépendance. D'un autre côté, l'impact dit structurel peut s'accompagner de façon directe ou indirecte d'un impact au niveau de la qualité du logiciel se manifestant par une altération de certains critères telles que la réutilisabilité, la testabilité et également sur son comportement par la modification des performances, etc. Des changement plus ou moins importants de la structure du logiciel peuvent également conduire à une remise en compte du schéma architectural conduisant par exemple, dans le cas des applications distribuées, à la migration de certains composants d'un serveur d'applications à un autre, etc. Dans la littérature, il existe un terme assez répandu pour désigner la propagation de l'impact du changement et qui est désigné par effet de vague ou ripple effect (67).

Dans ce chapitre nous adressons la problématique de la propagation de l'impact du changement du point de vue structurel. Nous proposons donc un modèle structurel pour l'évolution du logiciel (SMSE : Structural Model for Software Evolution). Ce modèle est défini pour représenter et manipuler les différents artefacts logiciels ainsi que les différentes relations sémantiques qui lient. L'instantiation de SMSE produit une base d'objets qui servira à alimenter en faits un système à base de connaissances destiné à mettre en œuvre de façon flexible et déclarative différentes techniques rentrant dans le cadre de l'analyse et la propagation du changement du logiciel. Parmi ces techniques, la plus notable est l'analyse et la propagation *a priori* du changement du logiciel qui constitue un des points de départ de l'évolution du logiciel puisqu'elle permet d'en mesurer les conséquences avant sa mise en œuvre.

Le reste de ce chapitre est organisé comme suit : nous commençons par une présentation du modèle SMSE et son élaboration (section 2.2). Nous présentons ainsi une classification ou hiérarchisation des artefacts logiciels (sous-section 2.2.2) et des divers types de relations qui les lient (sous-section 2.2.3). Nous présentons alors un exemple de mise en œuvre du modèle appliquée à une application Java (section 2.3) puis à une application multi-tiers distribuée selon la technologie Java J2EE (section 2.4). La section 2.5 conclut le chapitre.

2.2 Le modèle structurel de l'évolution du logiciel

Le développement du logiciel s'effectue généralement selon un processus appelé cycle de vie du logiciel et concrétisant un modèle de processus tel que le modèle en cascade, le cycle en V, le modèle en spiral, le prototypage, les modèles transformationnels, les modèles évolutionnaires, les modèles agiles, etc. (68).

Le développement est généralement réalisé par un groupe d'acteurs (Σ_{act}) effectuant un ensemble d'activités (Σ_{aty}). A chaque acteur de Σ_{act} (l'ensemble des acteurs) est attribué un rôle définissant le sous-ensemble de Σ_{aty} (l'ensemble des activités) qu'il peut effectuées. Les activités peuvent locales à une phase, c'est-à-dire circonscrite à une seule phase du cycle de vie du logiciel (codage, test, etc.) ou transversales et ayant comme portée plusieurs phases du cycle de vie (gestion de projets, assurance qualité, gestion des configurations, etc.).

Une méthode particulière appartenant à Σ_{mtd} l'ensemble des méthodes est souvent utilisé par un processus pour accomplir une tâche telle que la description, l'implémentation et/ou le test d'un artefact, etc. DE la même manière on définira les ensembles Σ_{lng} des langages et Σ_{tls} des outils utilisés pour l'accomplissement des activités ou tâches relevant du processus de développement et de maintenance du logiciel. Une description de haut niveau des phases ($\Sigma\phi$) du processus qu'on notera ($d\Sigma\phi$) sera représentée par le n-uplet :

$$d\Sigma\phi =< \Sigma_{act}, \Sigma_{aty}, \Sigma_{art}, \Sigma_{mtd}, \Sigma_{lng}, \Sigma_{tls} >.$$

Le modèle SMSE prend en compte la majorité des concepts communs aux différents modèles de cycle de vie du logiciel (68). L'ensemble des phases du cycle de vie noté $\Sigma\phi$ peut alors être noté par l'expression :

$$\Sigma\phi = \phi_1 \bigcup \phi_2 \bigcup \ldots \bigcup \phi_n$$

$n \in N$ et $n > 1$, l'entier n représente le nombre total de phases définie par le modèle du cycle de vie adopté pour le développement du logiciel et ϕ_i représente la i^{me} phase de ce cycle.

Par exemple, supposant que ϕ_a, ϕ_d, ϕ_c, ϕ_t, et ϕ_m représentent respectivement les phases d'analyse, de conception, de codage, de test et de maintenance. L'ensemble ($\Sigma\phi$) est donc :

$$\Sigma\phi = \phi_a \bigcup \phi_d \bigcup \phi_c \bigcup \phi_t \bigcup \phi_m$$

Nous pouvons également définir des sous-phases ϕ_{p1}, ϕ_{p2}, $\ldots\phi_{pn}$ d'une phase ϕ_p. Ainsi, la phase de conception ϕ_d peut être constituée de trois sous-phases ϕ_{d1}, ϕ_{d2}, et ϕ_{d3} telle que ϕ_{d1} correspond à la conception fonctionnelle de haut-niveau, ϕ_{d2} à la conception de l'architecture et ϕ_{d3} à la conception détaillée. Cela permet donc de définir la phase de conception ϕ_d par l'expression :

$$\phi_d = \phi_{d1} \bigcup \phi_{d2} \bigcup \phi_{d3}$$

2.2.1 Modélisation des transformations inter-phases

Les artefacts générés par le développement et la maintenance ou évolution du logiciel durant les différentes phases sont inter-reliés et donc interdépendants. Si l'on considère ces artefacts comme des graphes alors chaque phase peut générer un graphe d'artefacts inter-reliés et pouvant être transformé pour produire un graphe correspondant à des artefacts de la phase qui la succède dans le cycle de vie du logiciel. Tout changement affectant un artefact est considéré donc comme une transformation ou réécriture du graphe correspondant à la phase générant cet artefact. Cette transformation génère un flux d'impacts se traduisant par des transformations affectant les graphes correspondant aux phases se situant en amont et en aval de la phase initiatrice du changement. Le flux d'impact ainsi généré est supporté par les relations de traçabilité reliant les artefacts de différentes phases.

Considérons le lien existant entre deux phases consécutives ϕ_i et $\phi_j\,(j = i+1)$ et $i = 1 \ldots n-1\,(n$ étant le nombre total de phases du cycle de vie). ϕ_i est alors dite d'un niveau d'abstraction plus élevé que ϕ_j et tous les artefacts de ϕ_i décrivent nécessairement des artefacts de ϕ_j. La transformation de ϕ_i vers ϕ_j peut être représentée par un morphisme τ_{ij} schématisant une mise en correspondance ou mapping entre les artefacts de ϕ_i et ceux de ϕ_j. Ce morphisme peut donc être défini par l'expression :

$$\tau_{ij} \,:\, d\Sigma\phi_i.\Sigma_{art} \rightarrow d\Sigma\phi_j.\Sigma_{art}$$

Le morphisme τ_{ij} traduit un artefact art_x de la phase $\phi_i\,(art_x \in \phi_i)$ en un autre artefact arty de $\phi_j\,(art_y \in \phi_j\,)$. Si $j > i$, on dira que $\phi_j.art_y$ est la représentation concrète de $\phi_i.art_x$ par τ_{ij} ou $\phi_j.art_y = \tau_{ij}(\phi_i.art_x)$. Inversement, $\phi_i.art_x$ est dite la description ou représentation abstraite de $\phi_j.art_y$ par τ_{ij}^{-1} i.e. $\phi_i.art_x = \tau_{ij}^{-1}(\phi_j.art_y)$.

On notera quand même que τ_{ij} n'est pas une fonction bijective et que ϕ_{j-1} n'existe pas toujours. Ceci est évident puisque si on considère la phase de codage et la phase de conception, tous les éléments de conception peuvent avoir des représentations concrètes au niveau du code mais l'inverse n'est pas vrai puisque certains artefacts du codage sont des éléments de détail d'implémentation qui n'ont pas leur place au niveau de la conception. D'un autre côté, le morphisme τ_{ij} ne définit pas forcément un mapping de type un-à-un (one-to-one). En effet, à un élément de spécification par exemple, peuvent correspondre plusieurs artefacts de conception, etc. Dans cette thèse nous nous intéressons beaucoup plus à l'exploitation de ce type de mapping dans le cadre de la traçabilité et donc de la propagation du flux de l'impact et non à la construction ou mise en œuvre de ce type de mapping qui relève plus du domaine du reverse engineering ou de la transformation de modèles, etc.

Nous utilisons donc les travaux de ces communautés. Nous discutons dans la section 2.2.3.1 la formulation de relations traduisant ces mappings et nous considérons brièvement dans le chapitre 3, le mapping entre les artefacts représentant la structure du logiciel et ceux relatifs à la description de son architecture.

Les artefacts définis dans une phase particulière du cycle de vie du logiciel peuvent être hiérarchisé selon différents critères dont les plus communément adoptés sont les niveaux d'abstraction et de granularité. D'un point de vue pragmatique et si on se limite par exemple à la phase de codage, on peut affirmer que le niveau de granularité c'est ce qui permet généralement de structurer les applications en des niveaux de type Applications composées de fichiers composées de classes composées de méthodes composées de blocs composées d'instructions, etc. C'est une sorte de relation de part-of qui trouve souvent son utilité dans la gestion des versions et configuration, etc. D'un autre côté le niveau d'abstraction permet de structurer un code par exemple en niveaux constitués d'interfaces implémentées par des classes ou de templates instanciés par des classes, etc. Dans une analyse du changement il est judicieux d'exploiter conjointement ces deux niveaux, ainsi il est des fois opportun d'analyser le flux de l'impact en utilisant les niveaux de granularité par rapport à des problématiques de type gestion de versions et de configuration mais si l'on considère des aspects liés par exemple à la sémantique même du changement à effectué, comme dans le cadre du refactoring, il est nécessaire de considérer les niveaux d'abstraction. C'est notamment du cas de l'utilisation des designs

patterns, etc. Dans la suite du manuscrit, nous focaliserons principalement sur le flux de l'impact du changement.

De toutes les manières, que l'on utilise le niveau de granularité, le niveau d'abstraction, une combinaison de ces deux niveaux ou tout autre type de hiérarchisation que l'on jugera opportune, l'analyse du changement et plus particulièrement la propagation de l'impact nous incite à considérer un lien de traçabilité au sein d'une même phase et reliant des artefacts de niveaux différents. Cela nous conduit donc à introduire cette notion de niveaux. Dans la suite du manuscrit nous focaliserons principalement sur les artefacts de la phase de codage mais les concepts que nous introduisons restent valables pour les artefacts des autres phases du cycle de vie.

2.2.2 Classification des artefacts de la phase de codage

La classification ou hiérarchisation des artefacts procède d'une stratégie de simplification du processus d'analyse du changement. En effet, cela nous permet de disposer d'une échelle de dimensions de l'analyse à l'image de ce qui se produit dans les processus d'analyse ou d'exploration des entrepôts de données. Ainsi il sera possible de définir des zoom et des zoom-arrière dans le cadre de l'analyse et propagation de l'impact du changement. Nous retenons deux principaux critères de classification nous permettant ainsi de définir deux niveaux (ou dimensions). Ce sont les niveaux d'abstraction et de granularité.

Il est important de noter que des artefacts peuvent avoir le même niveau d'abstraction sans avoir le même niveau de granularité. Dans le cadre du paradigme objet, par exemple, une variable d'instance (champ d'une classe) et une méthode ont le même niveau d'abstraction mais pas le même niveau de granularité (une méthode de granularité plus "grosse" qu'une variable d'instance). De la même manière des artefacts peuvent avoir le même niveau de granularité mais des niveaux d'abstraction différents. Ainsi, une classe a le même niveau de granularité qu'un template ou patron de classe mais pas le même niveau d'abstraction (un template a un niveau d'abstraction supérieur à celui d'une classe).

Les critères de classification abstro-granulaire sont définis de façon générique et peuvent être raffinés par langage de programmation. Ainsi bien qu'ils partagent beaucoup de concepts communs, les langages de programmation à objets et les langages procéduraux sont traités de façon séparée. Dans les langages procéduraux les niveaux

identifiés sont les modules génériques, les modules, les blocs, les instructions, les expressions et les symboles individuels. En plus de ces niveaux, les langages à objets définissent les niveaux classes génériques, classes, méthodes, champs, objets, etc. Des concepts communs aux deux types de langages peuvent être classés dans diférents sous-niveaux. Au niveau modulaire on peut considérer les sous-niveaux spécification et corps (spécification d'une classe C++, corps des méthodes de la classe, spécification d'un package Ada, corps du package Ada, etc.). Au niveau bloc on définira le sous-niveau bloc déclaratif, bloc d'instructions, bloc de contrôle, et bloc de commentaire, etc. Au niveau instruction on distinguera les sous-niveaux instruction de déclaration, instruction de contrôle, instruction fonctionnelle basique et ligne de commentaire. Au niveau des symboles individuels on distinguera les opérandes, les opérateurs et les symboles qui ne sont ni opérandes, ni opérateurs.

Un artefact ou un ensemble d'artefacts peut être analysé aux niveaux modulaire, objet, méthode, bloc ou symbole individuel (10). L'ensemble des artefacts peut être représenté selon la hiérarchisation en niveaux par la paire $< \Sigma_{Lv}, \Sigma_{art} >$ où Σ_{Lv} est l'ensemble de tous les niveaux. Le k^{me} artefact du j^{me} niveau est représenté par le couple $< Lv_j, art_k >$. Dans une notation ensembliste, nous définissions un artefact par :

$$< Lv_j, art_k > = \{< Lv_j, art_k > | art_k \in \Sigma_{art} \wedge \ art_k \in Lv_j\}$$

Par exemple, $< Lv_{Class}, Calculator >$ dénote le fait que Calculator est un artefact appartenant au niveau classe. Le n-uplet $< Lv_{j1/j2/.../jn}, art_k >$ signifie que art_k appartient au niveaux Lv_{j1} ou Lv_{j2} ou $...Lv_{jn}$. Par exemple, $< Lv_{Class/Interface}, Calculator >$ signifie que Calculator peut appartenir au niveau Classe ou au niveau Interface. Le triplet $< \phi_i, Lv_j, art_k >$ représente un artefact de la phase ϕ_i, et du niveau Lv_j. Il peut être défini par :

$$< \phi_i, Lv_j, art_k > = \{< Lv_j, art_k > | art_k \in \Sigma_{art} \wedge \ art_k \in Lv_j \wedge \ art_k \in \phi_i\}$$

Par exemple, $< \phi_c, Lv_{Class/Interface}, Calculator >$ signifie que $Calculator$ est un artefact pouvant appartenir au niveau classe ou au niveau interface dans $\Sigma_{art} \in \phi_c$.

2.2.3 Modélisation des relations entre artefacts logiciels

Modéliser les relations inter-artefacts est une tâche complexe et très importante. Pour le cas de relations que nous qualifions de simples, nous considérons trois types de relations (figure 2.1). Ce sont :

1. Les relations inter-pahes : ces relations relient des artefacts appartenant à deux phases différentes du cycle de vie. C'est le cas par exemple de la relation appelée "implementation" qui relie une classe UML à la clase Java qui l'implémente.
2. Les relations horizontales : elles représentent différentes sortes de liens sémantiques qui relient des artefacts d'un même niveau abstro-granulaire. C'est notamment le cas de la relation d'appel entre deux méthodes ou de la relation d'héritage entre deux classes, etc.
3. Les relations verticales : elles relient deux artefacts appartenant à des niveaux abstro-granulaires différents. Un exemple de ce type de relation est celle qui relie une méthode à son corps ou un bloc aux instructions qui le composent, etc.

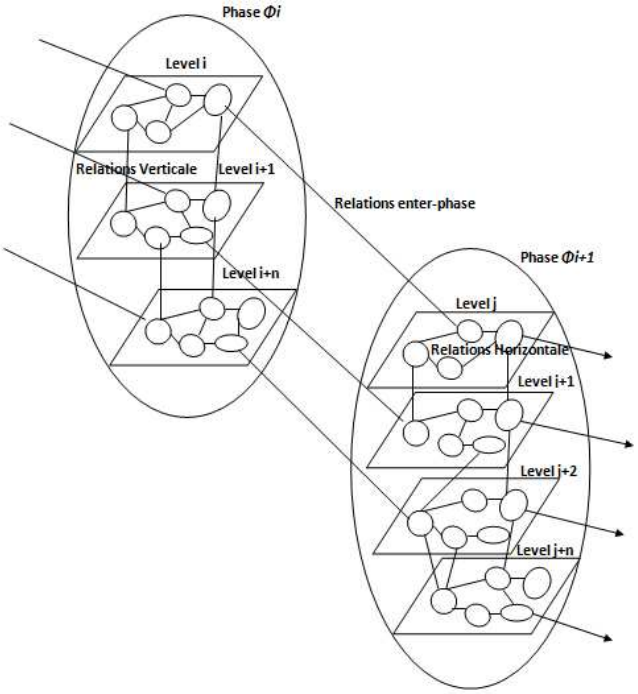


Figure 2.1: Classification des relations d'artefacts

Pour des relations plus complexes telles que les dépendances cachées nous adoptons une approche qui consiste à les définir de façon manuelle par l'expert chargé de l'évolution. Cela veut dire que ces dépendances sont extraites soit manuellement ou par des algorithmes spécifiques. C'est le cas, par exemple de relations de cause-à-effet de type les performances du module $M1$ sont inversement proportionnelles à celle du module $M2$,

etc. Ces types de relations peuvent être instanciées manuellement ou par un algorithme de fouille de données, etc.

Dans le cas de notre modèle, nous permettant aux utilisateurs (qui sont souvent les chargés de l'évolution) de définir les dépendances entre artefacts logiciels et aussi de définir leur rôle dans la propagation de l'impact des changements. Il est possible d'assigner des poids aux relations par rapport à la conductivité de l'impact. Ainsi, on pourra spécifier qu'un type de relation est plus critique et donc plus important à analyser lors du processus de propagation de l'impact. Pour un meilleur contrôle de l'évolution du logiciel, l'historique du changement doit être maintenu. L'ensemble des raisons et décisions managériales ayant conduit aux changement est dénotée Σ_R et peut être sauvegardé dans les artefacts de $\Sigma_{A.art}$ affectés par le changement. L'ensemble des types de relations entre chaque phase est la paire $< \Sigma_R, \Sigma_{A.art} >$ (historique sauvegardé) décrit les éléments potentiellement affectés par les décisions correspondantes. Ces éléments seront revus avec plus de détails dans le prochain chapitre.

2.2.3.1 Modélisation des relations inter-phases

L'ensemble des relations inter-phases traduit la communication entre les différentes phases du cycle de développement du logiciel. Il représente les liens de traçabilité entre artefacts issus de différentes phases. Comme discuté précédemment, nous définissions plusieurs niveaux au sein d'une même phase. Lors d'un changement, une analyse de l'impact peut se faire par la prise en compte des relations horizontales (même phase même niveau), verticales (même phase niveaux différents) et inter-phases. Les relations inter-phases peuvent être de deux sortes : les relations « forward » ou directes et les relations « backward » ou inverses. Les relations forward relient peuvent être notés $rel(art_x, art_y)$ où art_y est une représentation concrète de art_x. Dans le cas des relations backward arty est une abstraction de art_x. Lors de l'analyse du changement une relation de type forward traduit le fait de vouloir répercuter les effets du changement d'un artefact sur ces représentations concrètes comme le fait de vouloir gérer les changement d'une classe UML sur la ou les classes Java l'implémentant. Une relation de type backward traduit le fait de vouloir répercuter les effets du changement d'un artefact sur ses représentations abstraites comme le fait de répercuter les effets du changement d'une classe Java sur la classe UML qu'elle implémente. Dans le cadre de l'analyse de l'impact du changement du logiciel, une prise en compte des différents types de relations nous permet

d'effectuer une analyse horizontale, verticale et inter-phase en exploitant respectivement les relations horizontales, verticales et inter-phases. Bien entendu, l'analyse inter-phase peut se faire en exploitant les relations backward (à l'image de ce qui se fait dans le reverse engineering) et les relations forward (à l'image de ce qui se fait dans le forward engineering).

Une relation entre deux artefacts art_x et art_y est notée par l'expression $rel(art_x, art_y)$. Nous définissons l'ensemble des relations (Σ_{rel}) par :

$$\Sigma_{rel} = \Sigma_{ir} \bigcup \Sigma_{hr} \bigcup \Sigma_{vr}$$

Où Σ_{ir}, Σ_{hr}, et Σ_{vr} représentent respectivement les ensembles des relations inter-phases, horizontales et verticales.

De la même manière et en nous restreignant à une seule phase ϕ_i, l'ensemble des relations associées à cette phase est défini par l'expression :

$$\phi_i.\Sigma_{rel} = \phi_i.\Sigma_{ir} \bigcup \phi_i.\Sigma_{hr} \bigcup \phi_i.\Sigma_{vr}$$

Où $\phi_i.\Sigma_{ir}$, $\phi_i.\Sigma_{hr}$, et $\phi_i.\Sigma_{vr}$ sont les ensembles respectifs des relations inter-phases, horizontales et verticales associées à la phase .... Comme les relations inter-phases peuvent être de type forward ou backward, l'ensemble des relations inter-phases est alors défini par :

$$\phi_i.\Sigma_{ir} = \phi_i.\Sigma_{irf} \bigcup \phi_i.\Sigma_{irb}$$

Où Σ_{irf} et Σ_{irb} définissent les ensembles respectifs des relations forward et backward. Donc

$$\phi_i.\Sigma_{rel} = \phi_i.\Sigma_{irf} \bigcup \phi_i.\Sigma_{irb} \bigcup \phi_i.\Sigma_{hr} \bigcup \phi_i.\Sigma_{vr}$$

L'ensemble $\phi_i.\Sigma_{irb}$ inclut les relations directs en correspondance backward entre un artefact $art_y \in \phi_i.\Sigma art$ et son image abstraite $art_x \in \phi_{i-1}.\Sigma_{art}$ où ϕ_i représente la phase consécutive à ϕ_{i-1}. De la même manière, l'ensemble $\phi_i.\Sigma_{irf}$ inclut les relations de type forward entre un artefact $art_y \in \phi_i.\Sigma_{art}$ et son implémentation (raffinement direct) par l'artefact $art_z \in \phi_{i+1}.\Sigma_{art}$.

Les relations concernées par la conduction de l'impact du changement d'un artefact $art_x \in \phi_{i-1}.\Sigma_{art}$ peuvent causer une violation de la consistance entre art_x et ses possibles descriptions $art_y \in \phi_i.\Sigma_{art}$ et entre art_y et ses possibles implémentations $art_z \in \phi_{i+1}\Sigma_{art}$.

Par exemple, soit deux artefacts art_x (classe UML) et art_y (une classe C++) appartenant à deux phases distinctes du cycle de développement : la conception ϕ_d et le codage ϕ_c (figure 2.2). La relation $rel(art_x, art_y) \in \phi_d.\Sigma_{irf}$ traduit le fait que art_y est une implémentation de art_x et la relation $rel(art_x, art_y) \in \phi_c.\Sigma_{irb}$ implique que art_x est une abstraction de art_y.

La relation inter-phase $rel(art_x, art_y)$ de type forward est conductrice d'impact de tout changement $\Delta(art_x)$ affectant art_x. La relation inter-phase $rel(art_x, art_y)$ de type backward est conductrice de l'impact du changement $\Delta(art_y)$ vers art_x. On dit alors que la relation inter-phase $rel(art_x, art_y)$ est conductrice bi-directionnelle de l'impact du changement.

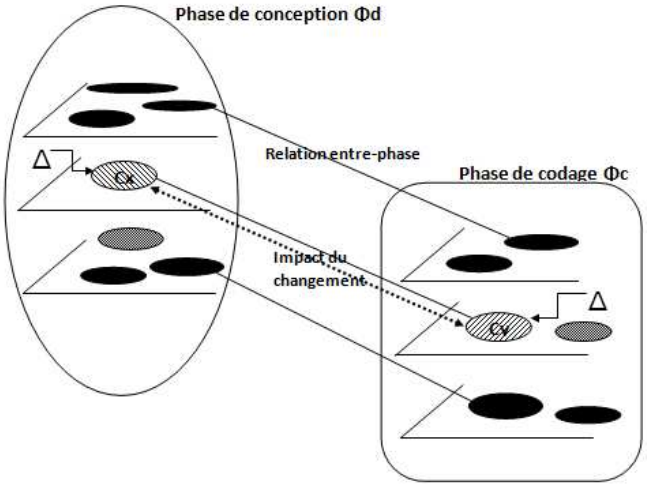


FIGURE 2.2: le flux de l'impact du changement entre les phases de conception et de codage

2.2.3.2 Modélisation des relations horizontales (intra-niveau)

Les relations horizontales traduisent des liens sémantiques existant entre des artefacts d'un même niveau abstro-granulaire. L'ensemble $\phi_i.\Sigma_{hr}$ représente toutes les relations horizontales associées à une phase 'i' et peuvent être hierarchisées selon le formalisme ou le langage utilisés pour définir les artefacts qu'elles relient. Les relations présentées dans les exemples qui vont suivre sont inspirés d'artefacts effectifs définis dans le cadre des langages de programmation procéduraux et/ou orientés objet. L'ensemble

des relations horizontales peut contenir les relations d'appel, d'importation, d'exportation, d'inclusion, d'héritage, d'instanciation, de communication, de synchronisation, de renommage, et de surcharge, etc.

La table 2.1 liste quelques relations horizontales pertinentes issus des codes source d'applications Java. La colonne conditions de cette table représentent les conditions d'existence ou d'applications de ces relations. En d'autres termes la violation de ces condition à la suite d'un changement traduisent une violation de la consistance ou une incohérence.Ces conditions vont donc servir comme un moyen très utile pour la propagation de l'impact du changement.

Comme exemple illustratif, nous considérons deux artefacts art_x ($< \phi_c, Lv_{method}, art_x > \in \Sigma_{art}$) et art_y ($< \phi_c, Lv_{method}, art_y > \in \Sigma_{art}$) comme un couple de modules de la phase de codage telle que art_y appelle art_x (figure 2.3). La relation d'appel $call(art_x, art_y) \in \phi_c.\Sigma_{hr}$ implique que toute modification $\Delta(art_x)$ de art_x (l'artefact appelé) affecte art_y (l'appelant) mais une modification de art_y n'a aucun impact sur art_x. Le cas où art_y est concerné par le changement de art_x dépend donc de la nature et du type de changement.

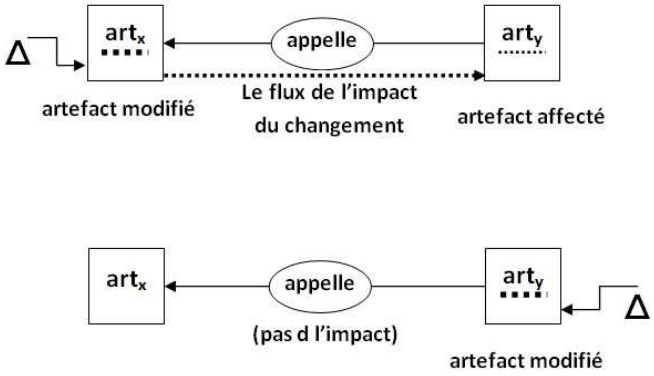


FIGURE 2.3: le flux de l'impact du changement dans la relation d'appel

2.2.3.3 Modélisation des relations verticales (inter-niveaux)

L'ensemble des relations verticales traduit des liens sémantiques existant entre des artefacts de niveaux différents. Cela peut souvent s'assimiler à des relations de type agrégation (part-of) mais pas seulement. L'ensemble $\phi_i.\Sigma_{vr}$ représente toutes les relations verticales associées à une phase 'i'.

TABLE 2.1: *Les relations horizontales entre les artefacts de Java*

Relationship type	**Source artifact** (art_x)	**Destination artifact** (art_y)	**Conditions**
Inheritance	Class	Class	$< Lv_{class}, art_x > \wedge < Lv_{class}, art_y > \wedge extends(art_x, art_y)$
	Interface	Interface(s)	$< Lv_{interface}, art_x > \wedge < Lv_{interface}, art_y > \wedge extends(art_x, art_y)$
Implementation	Class	Interface(s)	$< Lv_{class}, art_x > \wedge < Lv_{interface}, art_y > \wedge implements(art_x, art_y)$
Importation	Class/Interface	Class(es)/Interface(s)	$< Lv_{class/interface}, art_x > \wedge < Lv_{class/interface}, art_y > \wedge import(art_x, art_y)$
Instantiation	Class	Object	$< Lv_{class/interface}, art_x > \wedge < Lv_{object}, art_y > \wedge instantiate(art_x, art_y)$
Reference			
Call	Method	Method	$< Lv_{method}, art_x > \wedge < Lv_{method}, art_y > \wedge call(art_x, art_y)$
Inclusion	Class	Inner Class	$< Lv_{class}, art_x > \wedge < Lv_{class}, art_y > \wedge compose(art_x, art_y)$
Communication	Constructor/Object	Constructor	$< Lv_{constructor/object}, art_x > \wedge < Lv_{constructor}, art_y > \wedge communicate(art_x, art_y)$
Synchronization	Object	Object	$< Lv_{object}, art_x > \wedge < Lv_{object}, art_y > \wedge synchronize(art_x, art_y)$
	Method	Method	$< Lv_{method}, art_x > \wedge < Lv_{method}, art_y > \wedge synchronize(art_x, art_y)$
Overloading	Method	Method	$< Lv_{method}, art_x > \wedge < Lv_{method}, art_y > \wedge overload(art_x, art_y)$
	Constructor	Constructor	$< Lv_{constructor}, art_x > \wedge < Lv_{constructor}, art_y > \wedge overload(art_x, art_y)$
Overriding	Method	Method	$< Lv_{method}, art_x > \wedge < Lv_{method}, art_y > \wedge override(art_x, art_y)$

TABLE 2.2: *Les relations verticales entre les artefacts de Java*

Relationship type	**Source artifact (art_x)**	**Destination artifact (art_y)**	**Conditions**
Declaration	Class	Attribute(s)	$< Lv_{class}, art_x > \wedge < Lv_{attribute}, art_y > \wedge declare(art_x, art_y)$
	Interface	Constant Attribute(s)	$< Lv_{interface}, art_x > \wedge < Lv_{attribute/method}, art_y > \wedge declare(art_x, art_y)$
		Abstract Method(s)	
Initialization	Method/Constructor	Attribute/Variable	$< Lv_{method/constructor}, art_x > \wedge < Lv_{attribute/variable}, art_y > \wedge initialize(art_x, art_y)$
Read	Method/Statement	Attribute/Variable	$< Lv_{method/statement}, art_x > \wedge < Lv_{attribute/variable}, art_y > \wedge visible(art_x, art_y)$
Modify	Method/Statement	Attribute/Variable	$< Lv_{method/statement}, art_x > \wedge < Lv_{attribute/variable}, art_y > \wedge visible(art_x, art_y)$
Parameter	Prototype	Parameter	$< Lv_{prototype}, art_x > \wedge < Lv_{parameter}, art_y > \wedge parameter(art_x, art_y)$
Communication	Object	File	$< Lv_{object}, art_x > \wedge < Lv_{file}, art_y > \wedge write(art_x, art_y)$
	File	Object	$< Lv_{file}, art_x > \wedge < Lv_{object}, art_y > \wedge read(art_x, art_y)$
Use	Statement	Expression/Attribute/Variable/ Parameter	$< Lv_{statement}, art_x > \wedge < Lv_{expression/attribute/variable/parameter}, art_y > \wedge use(art_x, art_y)$
	Expression	Attribute/Variable/Parameter	$< Lv_{expression}, art_x > \wedge < Lv_{attribute/variable/parameter}, art_y > \wedge use(art_x, art_y)$
Encapsulation	Class	Method/Attribute	$< Lv_{class}, art_x > \wedge < Lv_{method/attribute}, art_y > \wedge encapsulate(art_x, art_y)$
Compose	Package	Class	$< Lv_{package}, art_x > \wedge < Lv_{class}, art_y > \wedge compose(art_x, art_y)$
	Method	Prototype/Body	$< Lv_{method}, art_x > \wedge < Lv_{prototype/body}, art_y > \wedge compose(art_x, art_y)$
	Body	Statement	$< Lv_{body}, art_x > \wedge < Lv_{statement}, art_y > \wedge compose(art_x, art_y)$
Protection	Package	Class	$< Lv_{package}, art_x > \wedge < Lv_{class}, art_y > \wedge define\_scope(art_x, art_y)$
	Class	Attribute/Method	$< Lv_{class}, art_x > \wedge < Lv_{attribute/method}, art_y > \wedge define\_scope(art_x, art_y)$

Si art_x désigne un artefact de niveau symbole individuel ($< Lv_{sym}, art_x > \in \Sigma_{art}$) et art_y un artefact de niveau modulaire ($< Lv_{method}, art_y > \in \Sigma_{art}$), la relation verticale entre ces deux artefacts peut être représentée par : art_x peut être déclaré, initialisé, lu, calculé, modifié, protégé, exporté, importé, visible, utilisé comme paramètre effectif dans un appel dans art_y, etc.

La table 2.2 résume quelques relations verticales pertinentes dans le cadre du code source des applications Java.

Considérons, à titre d'exemple, trois artefacts art_x, art_y, et art_z appartenant à la phase de codage (ϕ_c) du cycle de développement. art_x et art_y sont un couple de modules ($(< Lv_{method}, art_x > \wedge < Lv_{method}, art_y >) \in \Sigma_{art}$). L'artefact art_z est un symbole individuel ($< Lv_{sym}, art_z > \in \Sigma_{art}$). art_z est visible dans art_x et art_y. La relation verticale $rel(art_y, art_z) \in \phi_c.\Sigma_{vr}$ implique que art_y est capable de modifier art_z. La relation verticale $rel(art_x, art_z) \in \phi_c.\Sigma_{vr}$ implique que art_x est capable d'utiliser art_z. Toute modification $\Delta(art_z)$ de art_z par art_y peut affecter art_x (figure 2.4).

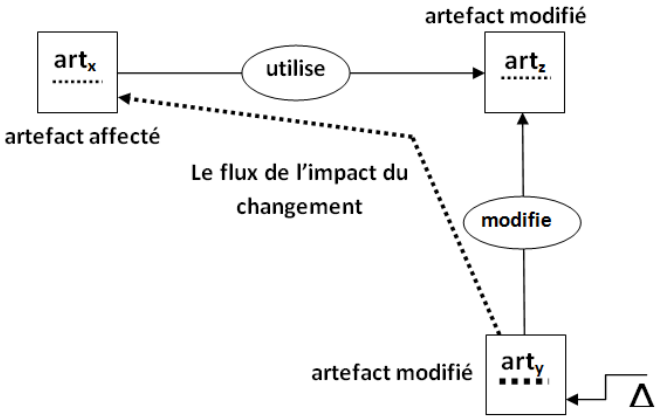


FIGURE 2.4: le flux de l'impact du changement dans les relations modifier et utiliser

2.3 Modélisation structurelle d'une application Java

Nous entamons la modélisation des applications Java en considérant une application exemple représentant une calculatrice et cela dans le but d'identifier les relations d'interdépendance d'artefacts du code source Java. Le code source de l'application est représenté par le listing 2.1. Ce code est constitué de deux classes nommées Calculator

et Main. La classe main encapsule l'invocation des l méthodes exécutant les opérations arithmétiques. Les opérations sont exécutées sur des paramètres. Chaque invocation est modélisée par un lien d'appel et les opérandes ou arguments de l'opération font partie des attributs décrivant la relation d'appel. Le diagramme d'objets de la figure 2.5 explicite les fonctionnalités de l'application Calculator.

Listing 2.1: La code source Java de l'application Calculateur

```
1  package fr.ulco. calculateur ;
2  public class Calculateur {
3      public int AddOperation(int opt1, int opt2) {
4          return opt1 + opt2;
5      }
6      public int SubOperation(int opt1, int opt2) {
7          return opt1 − opt2;
8      }
9      public int MulOperation(int opt1, int opt2) {
10         return opt1 ∗ opt2;
11     }
12     public int DivOperation(int opt1, int opt2) {
13         return opt1 / opt2;
14     }
15 }
16
17 package fr.ulco.main;
18 import java. util .Scanner;
19 import fr.ulco. calculateur . Calculateur ;
20 public class Main {
21     private static int operand1;
22     private static int operand2;
23     public static void main(String [] args) {
24         Calculateur calculate = new Calculateur();
25         Scanner operateur = new Scanner(System.in);
26         operand1 = operateur.nextInt ();
27         operand2 = operateur.nextInt ();
28         System.out. println (" Veuillez choisir le type d'opération \n 1)Addition \n
29         2) Soustraction \n 3) Multiplication \n 4) Division ");
30         switch (operateur. nextInt ()) {
31             case 1: System.out. println ( calculate .AddOperation(operand1, operand2)); break;
32             case 2: System.out. println ( calculate .SubOperation(operand1, operand2)); break;
33             case 3: System.out. println ( calculate .MulOperation(operand1, operand2)); break;
34             case 4: System.out. println ( calculate .DivOperation(operand1, operand2)); break;
35             }
36         }
37 }
```

Les deux classes appartiennent au même projet et à deux différents paquetages. Elles sont inter-reliées par les relations d'importation, d'encapsulation, et d'appel. Ces trois relations sont interdépendante. En effet, si l'on supprime la relation d'appel il n'y a pas d'erreurs de compilation mais dans ce cas l'importation et l'encapsulation sont superflues (ne servent à rien). D'un autre côté, la suppression de l'importation ou de l'en-

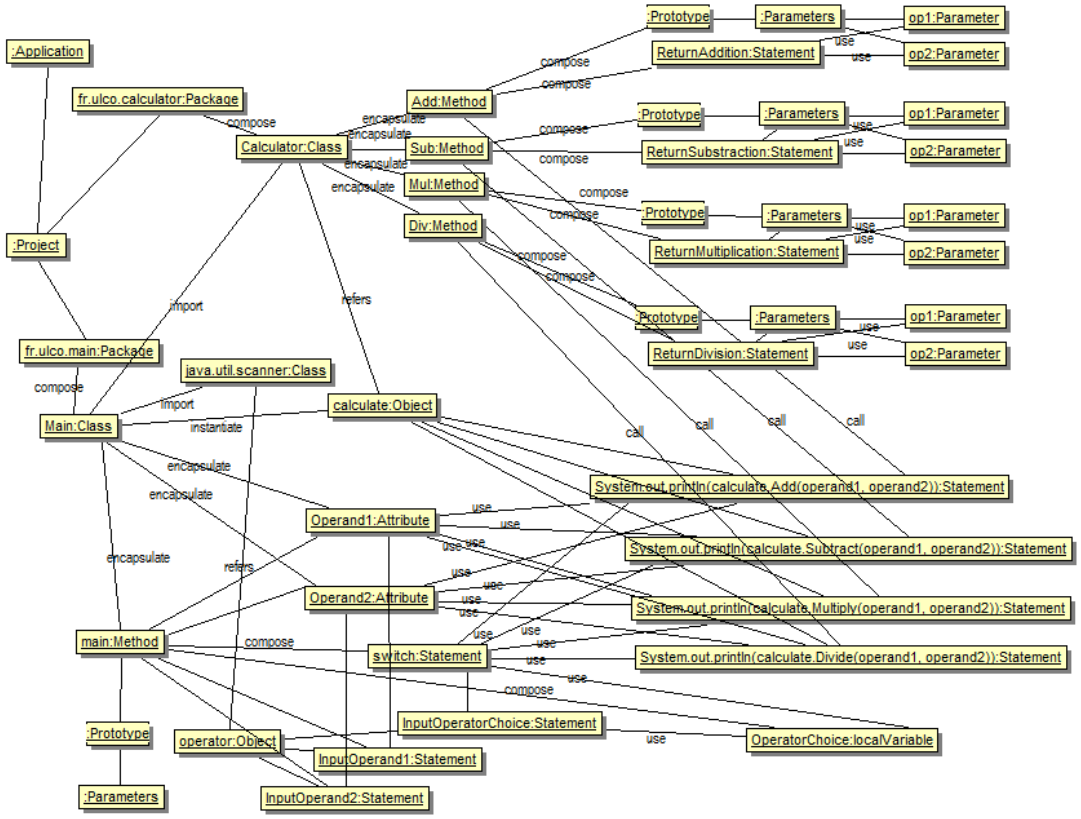


Figure 2.5: Le diagramme d'objet en UML de l'application Calculateur

capsulation provoque des erreurs de compilation au niveau des appels. L'appel est donc la relation la plus importante (la plus critique) dans cet exemple. Nous pouvons observer de façon détaillée les différentes inter-dépendances entre les artefacts de l'application en la représentant par un graphe d'artefacts et de relations obtenu après une modélisation à l'aide du modèle structurel de SMSE et une extraction automatique de cette modélisation par les outils implémentant SMSE. Une analyse d'impact peut, en effet, nécessiter une description exhaustive des différents artefacts logiciels et de leurs inter-dépendances. Ces descriptions sont, dans notre cas, représentées par des faits extraits, à partir du code source, par des analyseurs statiques. Ces analyseurs construisent ainsi, de proche en proche, des abstractions du code source permettant de mieux le comprendre et donc de cerner les impacts du changement. Les relations extraites de l'application Calculator sont des relations verticales pouvant être représentée par des faits destinés à alimenter un système à base de connaissance implémentant les stratégies et techniques de propagation et d'analyse du changement et résumées par le listing 2.2.

Listing 2.2: La base de faits de la class Calculateur

```
1  Protection
2  < Lv_package, fr.ulco.calculateur > ∧ < Lv_class, Calculateur > ∧
3  define_scope(fr.ulco.calculateur, Calculateur)
4
5  Encapsulation
6  < Lv_class, Calculateur > ∧ < Lv_method, Add > ∧encapsulate(Calculateur, Add)
7  < Lv_class, Calculateur > ∧ < Lv_method, Subtract > ∧encapsulate(Calculateur, Subtract)
8  < Lv_class, Calculateur > ∧ < Lv_method, Multiply > ∧encapsulate(Calculateur, Multiply)
9  < Lv_class, Calculateur > ∧ < Lv_method, Divide > ∧encapsulate(Calculateur, Divide)
10
11 Compose
12 < Lv_method, Add > ∧ < Lv_prototype, Add(int opt1, int opt2) > ∧compose(Add, Add(int opt1, int opt2))
13 < Lv_method, Subtract > ∧ < Lv_prototype, Subtract(int opt1, int opt2) > ∧
14 compose(Subtract, Subtract(int opt1, int opt2))
15 < Lv_method, Multiply > ∧ < Lv_prototype, Multiply(int opt1, int opt2) > ∧
16 compose(Multiply, Multiply(intopt1, intopt2))
17 < Lv_method, Divide > ∧ < Lv_prototype, Divide(int opt1, int opt2) > ∧
18 compose(Divide, Divide(int opt1, int opt2))
19
20 Parameter
21 < Lv_prototype, public int Add(int opt1, int opt2) > ∧ < Lv_parameter, opt1 > ∧
22 parameter(public int Add(int opt1, int opt2), opt1)
23 < Lv_prototype, public int Add(int opt1, int opt2) > ∧ < Lv_parameter, opt1 > ∧
24 parameter(public int Add(int opt1, int opt2), opt1)
25
26 < Lv_prototype, public int Subtract(int opt1, int opt2) > ∧ < Lv_parameter, opt1 > ∧
27 parameter(public int Subtract(int opt1, int opt2), opt1)
28 < Lv_prototype, public int Subtract(int opt1, int opt2) > ∧ < Lv_parameter, opt2 > ∧
29 parameter(public int Subtract(int opt1, int opt2), opt2)
30
31 < Lv_prototype, public int Multiply(int opt1, int opt2) > ∧ < Lv_parameter, opt1 > ∧
```

32 $parameter(public\ int\ Multiply(int\ opt1, int\ opt2), opt1)$
33 $< Lv_{prototype}, public\ int\ Multiply(int\ opt1, int\ opt2) > \wedge < Lv_{parameter}, opt2 > \wedge$
34 $parameter(public\ int\ Multiply(int\ opt1, int\ opt2), opt2)$
35
36 $< Lv_{prototype}, public\ int\ Divide(int\ opt1, int\ opt2) > \wedge < Lv_{parameter}, opt1 > \wedge$
37 $parameter(public\ int\ Divide(int\ opt1, int\ opt2), opt1)$
38 $< Lv_{prototype}, public\ intDivide(int\ opt1, int\ opt2) > \wedge < Lv_{parameter}, opt2 > \wedge$
39 $parameter(public\ int\ Divide(int\ opt1, int\ opt2), opt2)$
40
41 **Use**
42 $< Lv_{statement}, return\ opt1\ +\ opt2; > \wedge < Lv_{parameter}, opt1 > \wedge use(return\ opt1\ +\ opt2;, opt1)$
43 $< Lv_{statement}, return\ opt1\ +\ opt2; > \wedge < Lv_{parameter}, opt2 > \wedge use(return\ opt1\ +\ opt2;, opt2)$
44
45 $< Lv_{statement}, return\ opt1\ -\ opt2; > \wedge < Lv_{parameter}, opt1 > \wedge use(return\ opt1\ -\ opt2;, opt1)$
46 $< Lv_{statement}, return\ opt1\ -\ opt2; > \wedge < Lv_{parameter}, opt2 > \wedge use(return\ opt1\ -\ opt2;, opt2)$
47
48 $< Lv_{statement}, return\ opt1 * opt2; > \wedge < Lv_{parameter}, opt1 > \wedge use(return\ opt1\ *\ opt2;, opt1)$
49 $< Lv_{statement}, return\ opt1 * opt2; > \wedge < Lv_{parameter}, opt2 > \wedge use(return\ opt1\ *\ opt2;, opt2)$
50
51 $< Lv_{statement}, return\ opt1\ /\ opt2; > \wedge < Lv_{parameter}, opt1 > \wedge use(return\ opt1\ /\ opt2;, opt1)$
52 $< Lv_{statement}, return\ opt1\ /\ opt2; > \wedge < Lv_{parameter}, opt2 > \wedge use(return\ opt1\ /\ opt2;, opt2)$

2.4 Modélisation des applications distribuées

Le modèle SMSE tel que défini dans les sections précédentes peut apparaître générale et pas assez explicite par rapport à des applications réelles. Nous avons donc utilisé les concepts défini dans SMSE dans le cadre de l'étude de l'évolution des applications distribuées de type applications multi-tiers. Le choix de ce type d'applications découle principalement du fait qu'elles représentent une grande partie des projets de développements actuels et cela aussi bien aux niveaux académique qu'industriel. D'un autre côté ces applications fournissent un nombre important d'artifacts et relations significatives pour nos travaux.

Une application distribuée peut avoir plusieurs points d'accès à travers le réseau. Elle est également déployée sur plusieurs noeuds ou calculateurs géographiquement disséminés et relié par un réseau d'interconnexion. Les artefacts exécutables de l'application sont exécutées sur plusieurs machines virtuelles. Genéralement, la typologie d'une application distribuée est définie par des serveurs de développement, des serveurs de déploiement, et des serveurs de bases de données. Un ensemble de fichiers de configuration définissent les inter-connexions entre les différents noeuds ou points d'accès de l'application. Une représentation structurelle de l'environnement de développement d'applications distribuées peut être schématisé par la figure 2.6. Ce schéma représente

la structure générale des applications distribuées qui sont disséminées sur différents emplacements physiques et les différentes associations logiques permettant, entre autres, le partage des ressources. Les différentes machines virtuelles peuvent être sur des systèmes différents (e.g. serveur de version de maintenance, serveur de rapports de bug, etc.) ou sur le même système.

Une application logicielle est généralement développé par plusieurs personnes ou développeurs travaillant sur différents postes reliés par un intra_net ou internet. Ceci peut nécessiter la mise en œuvre de serveurs de développement et de bases de données différents. Le logiciel est agrégé ou intégré à l'aide de machines centralisées et déployé sur différents serveurs de déploiement. Ce type d'application peut concerner des systèmes de gestion de transactions bancaires, des applications de type réseaux sociaux ou de e-commerce, etc. C'est également le cas des différentes stratégies de mise en œuvre du e-learning, des e-services, du e-business,ainsi que toutes sortes d'applications supportant le travail collaboratif, etc.

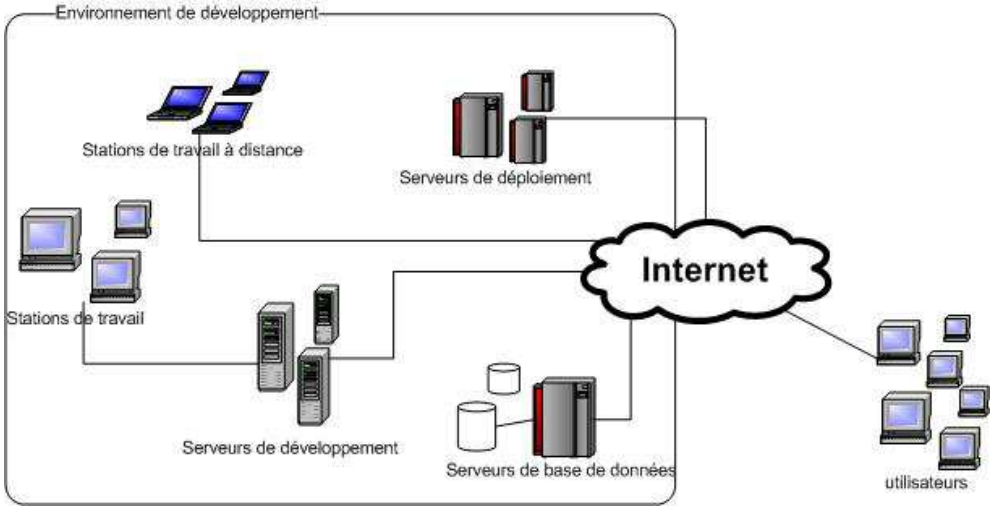


FIGURE 2.6: La structure générale d'application distribuée dans un environnement de développement

Dans le cadre des applications à objets écrites avec des langages tels que Java, Smalltalk, Python ou C++ un très grand nombre de classes sont développées et déployés en manipulant un grand nombre de données persistantes stockées par exemple dans des bases de données relationnelles. Si l'on veut tracer l'impact du changement d'une table (e.g. art_x) utilisée dans une classe art_y, nous devons d'abord comprendre les méthodes

ou fonctions définies dans art_y. Cela nous permettra par la suite de découvrir que art_y étend (hérite de) une classe (art_z) qui inclut une inner classe art_a configurée à travers un fichier xml (art_b). Un changement dans art_a peut donc affecter l'utilisation de art_x. Ce type d'inter-dépendance rend encore plus difficile l'analyse de la propagation de l'impact si des artefacts comme $art_x, art_y, art_z, art_a$, et art_b sont exécutés sur des plate-formes différentes. Cette distribution des applications logicielles sur différents serveurs rend la compréhension de leur structure et donc de leur processus changement très difficile.

SMSE tente de représenter les divers artefacts hétérogènes d'une application distribuée dans un graphe homogène permettant de simplifier la compréhension de leurs inter-connexions.

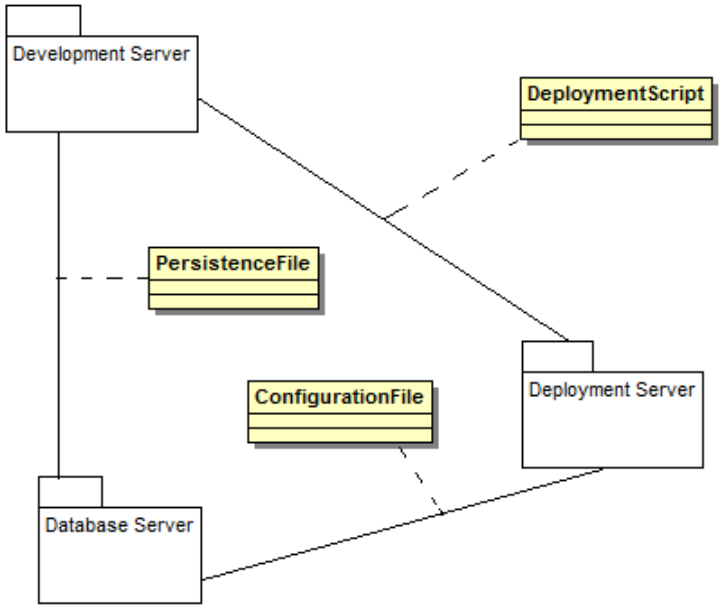


Figure 2.7: Inter-connectivité des différents serveurs d'une application J2EE

2.4.1 Modélisation de la structure d'une application J2EE

Dans cette section nous décrivons la modélisation et l'analyse des applications Java J2EE [1] dans le contexte de SMSE. Java J2EE définit une sorte de standard pour le développement d'applications distribués multi-tiers. Il simplifie la construction d'applications d'entreprise en utilisant des artefacts modulaires standardisés. Il fournit, éga-

1. http://java.sun.com/j2ee/

lement, un ensemble complet de services associés à ces artefacts. J2EE permet une automatisation de certains aspects concernant les détails de gestion du cycle de vie et du comportement des composants mis en œuvre, ce qui diminue de façon importante la complexité de la programmation des applications et augmente la réutilisabilité des composants ainsi développés.

La plate-forme J2EE profite de plusieurs avantages de la Java 2 Platform Standard Edition, de Java Database Connectivity (JDBC), de CORBA et des serveurs de déploiement d'applications (JBOSS, etc.) qui fournissent, entre autres, des technologies d'interaction avec les ressources existantes de l'entreprise. J2EE profite également des modèles et systèmes de gestion de la sécurité permettant de protéger les données et services transitant par Internet.

Une application J2EE repose sur plusieurs serveurs incluant des serveurs de développement, des serveurs de déploiement et des serveurs de bases de données. Un schéma logique d'interconnexion de ces multiples serveurs est présenté par la figure 2.7.

Le serveur de développement fournit généralement un environnement de développement permettant de mettre en œuvre une multitude de sortes de composants rentrant dans la construction des applications Java J2EE. Cela inclut les Enterprise Java Beans, les Java Servlets API, les Java Server Pages et tous les aspects relevant de la technologie XML, etc.

Le serveur de bases de données permet la définition, la manipulation et la recherche des données persistantes utilisées par l'application. Le serveur de déploiement héberge les projets de l'application et leurs codes exécutables. Il contient également les configurations permettant de notamment de gérer les connexions aux serveurs de bases de données, etc. Lorsque, l'applications est déployé sur le serveur de déploiement, ils se connecte au serveur de bases de données par les fichiers persistants et/ou par les objets de la base de données. La figure 2.8 présente le constitution de modules en application J2EE. Elle fournit un representation pour le développement, le déploiement et la gestion de l'exécution des modules Java, des modules web et des modules EJB. L'application J2EE est une archive EAR contenant le descripteur de déploiement (application.xml),les modules Java, les modules Web et EJB. Les modules Java du client incluent les fichiers Java pour les clients également une archive JAR avec le descripteur de déploiement (application-client.xml). Les modules web incluent, servlets, JSP, taglibs, JARs, HTML, XML, Images, etc. empaquetés dans un fichier d'archive web (WAR). Un WAR

s'apparente à un JAR avec en plus un répertoire WEB-INF contenant le descripteur de déploiement (web.xml). Les modules EJB incluent, les EJB (codes compilés) et leurs descripteurs de déploiement (ejb-jar.xml) empaquetés dans une archive JAR. Un schéma de base de données peut se lier par un discripteur de connection *-ds.xml. Supposant que l'ingénieur de développement veut modifier un column de la table de base de données. L'impacte de cette modification peut se propager dans l'application, par ex. dans ce cas là vers un bean qui peut être annoté par la persistance à cette table.

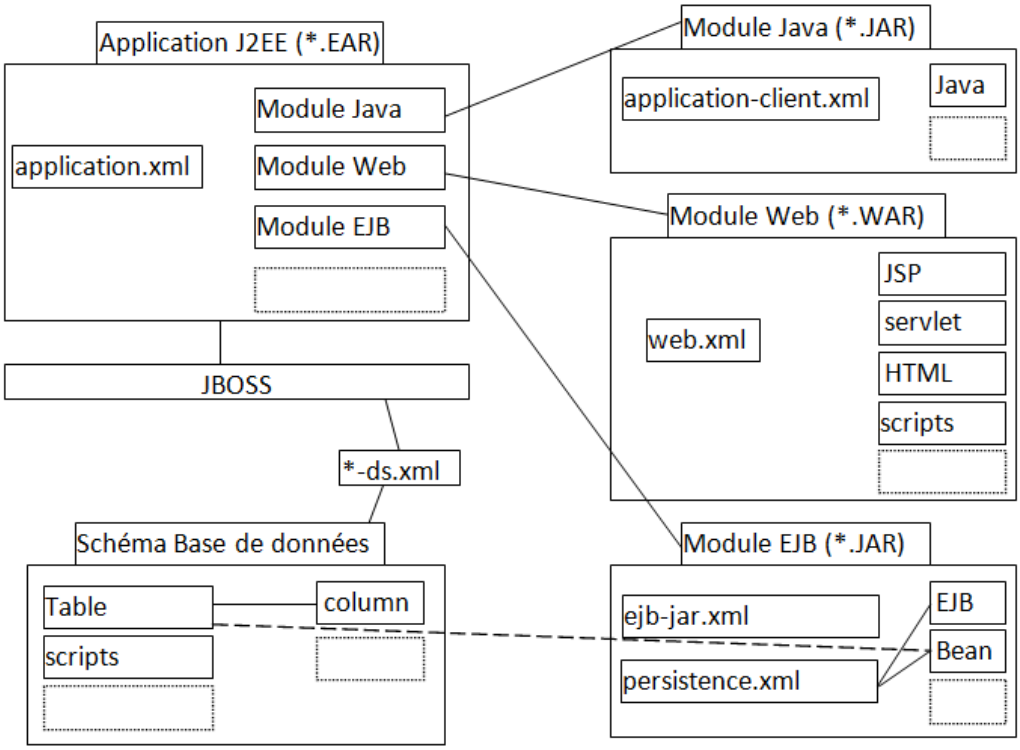


Figure 2.8: Constitution de modules en application J2EE

La figure 2.9 représente, à l'aide d'un diagramme de classes UML, le méta-schéma d'une application J2EE. Ce schéma décrit les associations logiques entre les artefacts de développement partagés, les bases de données et les serveurs de déploiement. Il décrit particulièrement la structure interne d'un code source. La structure d'un document Web et les autres fichiers sont représentés de façon abstraite au niveau fichier pour éviter une complexité du schéma. Les différents artefacts associés au concept de Classe sont décrit par le biais des relations qui les lient à ce concept. Ainsi, l'artefact annotation est

représenté par une relation qui relie une classe à la table qui gère la persistance des objets de cette classe, etc. Cela nous permettra, lors de l'analyse du changement, de tracer l'impact du changement d'une colonne de la table sur les méthodes des classes qui les manipulent, etc.

En se basant sur ce méta-schéma, nous identifions un ensemble de dépendances relatives aux applications Java J2EE et qui sont résumées dans la section suivante. Cette section résume en particulier les artefacts et leurs interdépendances en ayant comme point focal la typologie des changements des applications distribuées.

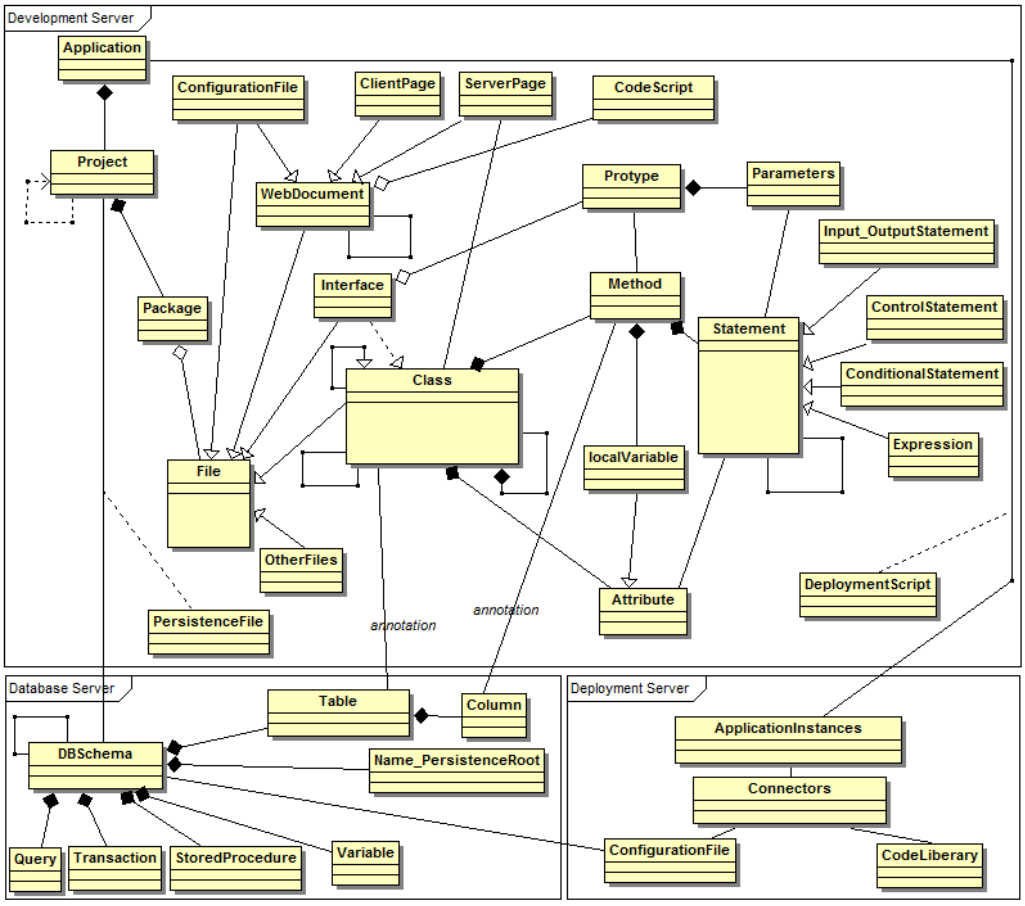


FIGURE 2.9: Représentation UML de la méta-structure de l'application J2EE

2.4.2 Modélisation de relations dans les applications J2EE

Dans les applications Java J2EE, les artefacts pertinents pour l'analyse du changement sont les fichiers de codes sources, les documents web, les schémas de bases de données, et les fichiers de configuration. Les fichiers de configuration définissent les protocoles de communication entre divers types d'artefacts. Ils sont généralement codés en utilisant le langage XML (eXtensible Markup Language) qui a été principalement défini pour l'échange de données semi-structurées entre applications hétérogènes. Les fichiers de configuration peuvent jouer un rôle très important dans le processus de propagation de l'impact du changement. Comparées aux applications Java StandAlone, l'analyse des applications J2EE peut nécessiter la définition de nombreuses relations complexes. L'ingénieur de développement peut définir ses propres types de relations permettant de représenter des dépendances cachées. Il peut également assigner des poids différents aux différents types de relations selon leur importance dans la propagation de l'impact du changement.

Dans cette section, nous présentons et discutons certains types de relations associées notamment aux serveurs de bases de données et de déploiement et leurs inter-connexion avec le serveur de développement, etc.

Supposant que l'ingénieur de développement considère que les classes persistantes (des classes particulières dont les instances sont stockées dans des bases de données) et un schéma de bases de données appartiennent au même niveau. De la même manière, il considère une table et une classe annotée ainsi qu'une colonne de la table et une méthode manipulant cette colonne comme faisant partie du même niveau respectivement.

Etant données ces décisions de modélisation, la table 2.3 (respectivement 2.4) liste quelques relations horizontales (respectivement verticales) pertinentes pour l'analyse et la propagation du changement. Ces types de relations sont relatifs à la définition des artefacts associés au serveur de bases de données et à leur connexion avec les artefacts du serveur de développement tout en respectant un ensemble de conditions.

2.5 Conclusion

Ce chapitre a présenté une classification exhaustive des artefacts logiciels et de leurs relations d'interdépendance pour une meilleure identification de leur rôle dans la

TABLE 2.3: *Les relations horizontales entre les artefacts Java et les artefacts base de données*

Relationship type	**Source artifact** (art_x)	**Destination artifact** (art_y)	**Conditions**
Persistence Implementation Instantiation	Persistent Class Configuration File	Database Schema	$< Lv_{persistentClass/configurationFile}, art_x > \wedge < Lv_{schema}, art_y > \wedge persistence(art_x, art_y)$
Annotation	Table	Class	$< Lv_{table}, art_x > \wedge < Lv_{class}, art_y > \wedge annotation(art_x, art_y)$
	Column	Method	$< Lv_{column}, art_x > \wedge < Lv_{method}, art_y > \wedge annotation(art_x, art_y)$
Database connection	Connection Object	Database Schema	$< Lv_{dbConnectionObj}, art_x > \wedge < Lv_{schema}, art_y > \wedge connection(art_x, art_y)$
	Data Source Object	Table	$< Lv_{dsObj}, art_x > \wedge < Lv_{table}, art_y > \wedge connection(art_x, art_y)$
	Database Statement Object	Query	$< Lv_{dbStatementObj}, art_x > \wedge < Lv_{query}, art_y > \wedge connection(art_x, art_y)$
Importation	Database Schema	Database Schema	$< Lv_{schema}, art_x > \wedge < Lv_{schema}, art_y > \wedge import(art_x, art_y)$
Cardinality	Table	Table	$< Lv_{table}, art_x > \wedge < Lv_{table}, art_y > \wedge cardinality(art_x, art_y)$
Override	Stored Procedure	Stored Procedure	$< Lv_{procedure}, art_x > \wedge < Lv_{procedure}, art_y > \wedge override(art_x, art_y)$
	Stored Function	Stored Function	$< Lv_{function}, art_x > \wedge < Lv_{function}, art_y > \wedge override(art_x, art_y)$
Call	Stored Procedure Stored Function	Stored Procedure Stored Function	$< Lv_{procedure/function}, art_x > \wedge < Lv_{procedure/function}, art_y > \wedge call(art_x, art_y)$
Reference	Data Structure	Object/Variable	$< Lv_{dataStructure}, art_x > \wedge < Lv_{object/variable}, art_y > \wedge refer(art_x, art_y)$

TABLE 2.4: *Les relations verticales entre les artefacts Java et les artefacts base de données*

Relationship type	**Source artifact** (art_x)	**Destination artifact** (art_y)	**Conditions**
Use	Persistent Object	Variable	$< Lv_{persistentObject/parameter}, art_x > \wedge <$ $Lv_{variable}, art_y > \wedge use(art_x, art_y)$
	Parameter		
	Transaction	Query/Table Stored Procedure/Function	$< Lv_{transaction}, art_x > \wedge <$ $Lv_{query/table/procedure/function}, art_y > \wedge use(art_x, art_y)$
Compose	Name Space	Schema	$< Lv_{nameSpace}, art_x > \wedge < Lv_{schema}, art_y > \wedge compose(art_x, art_y)$
	Schema	Query/Table	$< Lv_{schema}, art_x > \wedge <$ $Lv_{query/table/procedure/function}, art_y > \wedge compose(art_x, art_y)$
		Stored Procedure/Function	
	Table	Column	$< Lv_{table}, art_x > \wedge < Lv_{column}, art_y > \wedge compose(art_x, art_y)$
	Stored Procedure/Function	Prototype/Body	$< Lv_{procedure/function}, art_x > \wedge <$ $Lv_{prototype/body/query/cursor/trigger}, art_y > \wedge compose(art_x, art_y)$
		Query/Cursor/Trigger	
	Query	Clause	$< Lv_{query}, art_x > \wedge < Lv_{clause}, art_y > \wedge compose(art_x, art_y)$

propagation de l'impact du changement. Un aperçu a été donné des points de vue structurel, qualitatif, logique, fonctionnel, ou comportemental, sous lesquels un logiciel peut être considéré. Par rapport à une classification stratifiant les composants en niveaux d'abstraction, les types de relations conducteurs de l'impact du changement ont été présentés. Le chapitre a décrit ensuite en détail l'approche de l'analyse de l'impact du changement à travers les constituants structurels des applications logicielles.

Un modèle structurel pour l'évolution du logiciel (SMSE) a été détaillé et illustré son instanciation aux applications distribuées. Le modèle considérant les applications logicielles comme un résultat des transformations des artefacts issus de plusieurs phases de développement, au sein de chaque phase les constituants ont été classés en niveaux d'abstraction et niveaux de granularité. Le SMSE identifie les interconnexions des artefacts par les relations entre les phases de développement et les niveaux d'abstraction et de granularité. Le modèle propose une analyse *a priori* de la propagation d'impact du changement parmi les artefacts logiciels. Le chapitre traite l'instanciation de SMSE pour la phase de codage. Il précise en outre l'approche proposée, avec l'aide d'exemples précisant l'analyse structurelle des applications distribuées développées en J2EE. Il analyse les structures internes reliant les différents artefacts d'application J2EE et illustre le rôle relations d'interdépendance dans la propagation d'impact du changement.

Le SMSE a été appliqué au chapitre suivant à une description architecturale élaborée moyennant un ADL (Architecture Description Language).

Chapitre 3

Analyse du Changement de l'Architecture du Logiciel

3.1 Introduction

L'architecture du logiciel incarne une modélisation de haut niveau d'abstraction de la structure du logiciel. Elle joue un rôle important dans la construction des systèmes logiciels en fournissant une conception d'éléments abstraits constituant le logiciel et de leurs interconnexions. La notion d'architecture logicielle donne lieu à des modèles et techniques destinés à comprendre les systèmes complexes et fournit des frameworks pour la conception de tels systèmes en respectant les principes généraux d'abstraction et d'encapsulation.

La modélisation architecturale permet aux concepteurs d'analyser les différents blocs de base constituant le logiciel. Elle fournit particulièrement une aide pour mieux comprendre, construire, analyser, réutiliser et faire évoluer un système logiciel. L'architecture est utilisée, notamment, pour organiser le développement logiciel selon une partition architecturale préalable. En effet, les systèmes logiciels sont composés de milliers d'instructions et la simple structuration en modules relatifs aux constituants du code source rend leur compréhension très difficile. Une modélisation architecturale introduit en effet des éléments plus abstraits tels que ceux introduits par les notions de clients, serveurs, producteurs, consommateurs. Ces éléments permettent une compréhension plus aisée des systèmes logiciels de grandes tailles. Cette compréhension est en effet, un élément clé de l'évolution du logiciel. Il sera alors plus aisé d'analyser les effets du changement

logiciel à un niveau d'abstraction plus élevé, niveau qui est matérialisée par les modèles architecturaux.

Dans les récentes années, la notion d'architecture logicielle a constitué un champs de recherche ayant intéressé plusieurs équipes constituant ainsi un domaine important du génie logiciel. La littérature relative à ce champs de recherche s'est particulièrement étoffée.

Ces travaux ont d'abord concerné la modélisation des systèmes distribués mais n'ont pas permis l'émergence d'une définition « universellement » acceptée de la notion d'architecture logicielle. Nonobstant, il existe une multitude de définitions de cette notion dans la littérature dont nous énumérons certaines ci-dessous.

« L'architecture logicielle d'un programme ou d'un système informatique est la structure ou les structures du système comprenant les composants logiciels, les propriétés extérieurement visibles de ces composants et les relations qui les relient » (69).

Dans (70), Kruchten *et al.* définissent l'architecture logicielle par « la structure et l'organisation selon lesquelles des composants sont intégrés pour créer de nouvelles applications, possédant des propriétés rendant meilleures analyse et leur conception au niveau système ».

L'architecture logicielle permet aux développeurs de focaliser sur les éléments ou artefacts architecturaux dits de grosse granularité et sur leurs interconnexions au lieu des éléments de fine granularité comme les fonctions ou lignes de code (71). Bien que le modèle structurel pour l'évolution du Logiciel (SMSE), défini dans le chapitre 2, a concerné les aspects structurels du changement du logiciel, nous l'avons étendu pour l'analyse d'impact du changement des architectures des applications distribuées. Il peut donc fournir une assistance pour la compréhension des effets des changements projetés ou actuels du logiciel au niveau architectural. Dans ce contexte, SMSE représente l'architecture logicielle en termes d'artefacts dits architecturaux et de relations d'interdépendances entre ces artefacts. Dans ce chapitre nous présenterons les concepts essentiels des architectures logicielles dans la perspective d'une meilleure analyse et d'un meilleur control de l'évolution du logiciel. Nous présentons d'abord une brève description des caractéristiques des langages dédiés à la modélisation et l'analyse des architectures logicielles.

Le reste du chapitre est organisé comme suit : la section 3.2, présente le paradigme de l'évolution des architectures logicielles. La section 3.3, décrit brièvement les

ADL (Architecture Description Languages ou Langages de Description d'Architectures). Nous y présentons les concepts de bases introduits par les ADL et discutons leurs différentes familles. La section 3.4, introduit les principaux concepts du langage AADL (Architecture Analysis and Description Language) que nous utilisons dans cette thèse pour illustrer nos travaux sur l'évolution du logiciel du point de vue architectural. La section 3.5 décrit l'approche que nous adoptons pour l'analyse du changement du logiciel du point de vue architectural et l'implémentation de cette approche dans le contexte du modèle SMSE. Finalement, nous concluons notre approche basée modèle pour l'analyse de l'évolution et la compréhension des architectures logicielles dans la section 3.6.

3.2 L'évolution de l'architecture du logiciel

L'avènement du génie logiciel basé composants (*Component Based Software Engineering*) (72, 73) et son adoption à tous les niveaux du cycle de développement induit une amélioration de la réutilisation et une simplification de l'expression des dépendances entre les différents modules formant le logiciel. Construire une architecture durable peut nécessiter une grande aptitude à anticiper certains types de changements ayant le plus de chances de survenir. De telles architectures capable de répondre à ce types de changement sont donc robustes et ne sont pas détériorées par les évolutions futures du logiciel. Cela peut impliquer deux principales caractéristiques : la possibilité de faire évoluer les architectures logicielles, mesurer et monitorer cette évolution.

Pour permettre l'évolution de l'architecture, l'extensibilité doit être prise en considération. Dans (74), l'auteur suggère une sorte de check list à suivre dans le cadre du développement basé ou orienté composants pour assurer l'extensibilité de l'architecture du logiciel.

Monitorer et mesurer l'évolution des architectures signifie assurer la consistance post-développement. Généralement, la supervision de l'évolution de l'architecture est plus consommatrice de temps et d'efforts que son développement. Un grand nombre de travaux de la littérature est consacré aux changements post-développement. Dans (75), Balzer présente une méthode de supervision de l'architecture par des animations. Cette technique permet le monitoring du comportement de l'architecture par l'instrumentation de ses connecteurs.

Superviser le développement de l'architecture ainsi que les différentes activités de ce type a induit le développement d'outils dédiés à l'analyse de l'architecture (76, 77, 78). Ces outils constituent une base pour monitorer les architectures en fournissant des informations sur leurs structures et leurs propriétés. A titre d'exemple, nous pouvons noter le développement d'une technique semi-formelle pour comparer des descriptions d'architectures dans le but de mettre en lumière leurs différences et leurs similitudes (79). Une telle technique permet de comparer différentes versions d'une architecture permettant ainsi de monitorer son évolution. Dans (80), Jazayeri utilise des métriques et des outils pour analyser rétrospectivement des versions successives des architectures et déterminer leurs manières d'opérer.

La supervision est souvent combinée au contrôle de l'évolution des architectures. Elle implique la possibilité de modifier, ajouter, ou supprimer des items les composant. Donc l'analyse de l'impact du changement peut devenir nécessaire pour un contrôle effectif des opérations de changement. Dans (81, 82), par exemple, les auteurs proposent une technique générant une matrice de dépendances utilisée dans le cadre de l'analyse de l'impact des changements au niveau de l'architecture.

Plus récemment, l'étude des langages de description d'architecture a constitué une approche émergente (83). La description de l'évolution est réalisée par le biais de l'ADL utilisé pour la description de l'architecture. L'implémentation du changement devient également une partie de la spécification de l'architecture. Cependant, il est difficile de distinguer la partie de la description d'architecture dédiée à la spécification de la propre évolution de cette même architecture. Certains ADLs tels que AADL (84) et Rapide (78) permettent à une description architecturale d'évoluer de façon dynamique. AADL fournit différents modes permettant plusieurs configurations possibles du système. Ce mécanisme peut, par contre, être insuffisant et non efficient si les changements du système sont très fréquents. Rapide utilise des règles régissant la création ou la destruction de composants durant l'exécution du système. Ces changements doivent être préalablement connus ce qui n'est pas évident pour des applications complexes.

Dans la section suivante nous faisons une revue des éléments de base des descriptions d'architectures et discutons leur modélisation avec comme point d'intérêt le contrôle de leur évolution.

3.3 Description de l'Architecture du Logiciel

Dans cette section, nous traiterons les concepts de haut niveau relatifs aux descriptions d'architectures logicielles dans le cadre du Modèle Structurel de L'Evolution du Logiciel (SMSE). Le but est de représenter les architectures des applications distribuées en se basant sur les spécifications architecturales utilisant différents ADLs.

M.O. Hassan a présenté, dans sa thèse (85), un modèle pour l'analyse du changement des architectures logicielles. Nous avons approfondi les résultats de ces travaux en réalisant, entre autres, un mapping avec les éléments des codes sources en utilisant SMSE.

Plusieurs ADL (Architecture Description Languages) ont été définis comme structures associant des sémantiques formelles et des diagrammes de description d'architectures. Nous discutons dans cette section les ADL les plus utilisés dans la littérature.

L'existence des ADL fournit, surtout, un vocabulaire commun de concepts a destination des différents intervenant d'un projet. Pour illustrer ces concepts, plusieurs langages ont vu le jour tels que Rapide (78), Unicon (76), Wright (86), Acme (87), et AADL (84).

Les ADL sont généralement basés sur une combinaison de description textuelles et de diagrammes. Des ensembles d'outils leurs sont également associés tels que les éditeurs, les simulateurs, etc. Les concepts de base et récurrents constituant une architectures sont communément désignés par l'expression des *4Cs* à savoir : *Composant*, *Connecteur*, *Configuration* et *Contrainte*. La figure 3.1 montre un exemple de description d'architecture utilisant un ADL et dans laquelle deux composants communiquent par un connecteur formant ainsi une configuration.

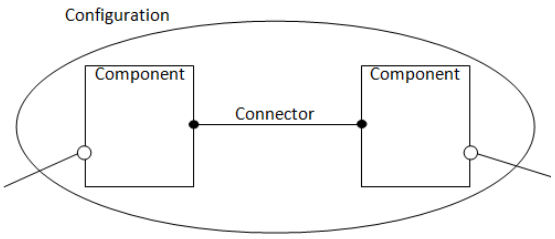


Figure 3.1: Exemple de description d'architecture en ADL

Dans (71), les auteurs définissent un cadre de classification de la majorité des ADL existants de nos jours et en proposent un ensemble de caractéristiques globales utilisées

comme base de comparaison.

Nous explicitons, dans ce qui suit, certaines de ces caractéristiques.

3.3.1 Les concepts de base d'une description d'architecture logicielle

Les concepts de base d'une description d'architecture logicielle sont généralement appelés les quatre « *Cs* » (88). Ce sont les Composants, les Connecteurs, les Configurations et les Contraintes. Dans (71), les auteurs ont établi un cadre de comparaison des ADL et ont classé leurs propriétés clés. Un ensemble de caractéristiques globales ont été proposées pour chaque élément constitutif d'une description architecturale. Ces caractéristiques peuvent être utilisées comme un ensemble de critères d'analyse et de comparaison des ADL. Un ADL doit satisfaire les propriétés de composition, abstraction, réutilisabilité, configuration, hétérogénéité, et analyse, etc (89).

La figure 3.1 montre un exemple d'architecture simpliste utilisant les concepts de base d'un ADL qui sont les composants communiquant par le biais d'un connecteur. Dans la section qui suit nous faisons une brève revue des caractéristiques relatives aux concepts formant les quatre *Cs* (71).

3.3.1.1 Les composants

Un composant est une entité de traitement ou de stockage de données. C'est une entité dynamique dont l'état évolue à travers le temps et en réponse aux requêtes qu'il reçoit. Un composant peut être primitif (e.g. une instruction d'un programme) ou composite en d'autres termes composé de composants primitifs et/ou composites (e.g. une application). La notion de composition de composants fournit la possibilité de modéliser à différents niveaux de détail. Cela nous permet donc de spécifier de l'échelle des simples procédures à celui des applications entières. L'interaction des composants d'effectue à travers des interfaces.

Les composants peuvent être de différents types. Le concept de type de composants permet de définir des propriétés communes à un ensemble d'instances du même composant. La notion de type introduit la réutilisabilité et l'abstraction de fonctionnalités dans le cadre d'une même architecture ou dans celui de plusieurs architectures. Ce concepts peut être assimilé à celui de la notion de classes dans le cadre du paradigme objet.

L'interface d'un composant forme un point d'interaction avec les autres composants. Cela peut être, par exemple, un port de communication entre deux applications ou un

ensemble de paramètres passés dans le cadre de l'appel d'une procédure, etc. L'interface inclut tous les services offerts et requis par le composant (e.g. messages, opérations) et exprime les contraintes liées à l'utilisation du composant. Les contraintes sont des assertions ou des propriétés physiques qui ne doivent pas être violées. Quand une contrainte est violée, le système se retrouve dans un état qualifié d'inconsistant. Un exemple de contraintes est le nombre maximal de chargement d'un composant ou le nombre maximal d'opérations d'appels que le composant peut supporter. C'est le cas par exemple du nombre de connexions à un S.G.B.D, etc. Ces propriétés sont qualifiées de non fonctionnelles et résultent de la spécification du comportement du logiciel. Elles incluent, entre autres, la robustesse, la performance, la dépendabilité, la fiabilité, et la disponibilité, etc.

3.3.1.2 Les connecteurs

Un connecteur modélise les interactions entre composants et définit les règles gouvernant ces interactions. Il vérifie l'intégrité de la communication et assure la possibilité aux composants d'interagir. Un connecteur peut décrire une simple interaction telle que l'appel de procédure ou l'accès à une variable partagée. Il peut également décrire des interactions complexes telles que les protocoles d'accès à une base de données, etc.

Une interface de connecteur définit l'ensemble des points d'interaction entre composants et connecteurs. Dans certains ADL, les points d'interaction des connecteurs sont apparentés à des rôles. Les connecteurs (comme les composants), peuvent avoir différents types définissant le mécanisme de communication entre composants. Le connecteur peut transporter des données, des événements, ou les deux. Des contraintes peuvent, également, être définies de façon explicite sur les connecteurs pour exprimer des limites sur leur utilisation et des dépendances entre connecteurs.

3.3.1.3 Les configurations

Les configurations ou topologies d'architectures, décrivent la structure et le comportement de l'architecture du logiciel. Ce sont de graphes connexes dont les noeuds sont les composants et les connecteurs. Elles fournissent l'information nécessaire sur le composants connectés, la mise en correspondance ou match de leurs interfaces, etc. La composition de la configuration donne la possibilité de décrire des architectures à

différents niveaux de détails. C'est une composition hiérarchique dans laquelle les composants primitives sont des unités atomiques et les composites sont constitués d'autres composants primitifs ou non.

De plus, une configuration est également une façon de raffiner une application d'un niveau d'abstraction élevé jusqu'à des descriptions beaucoup plus détaillées. Cela permet donc la création de liens de traçabilité permettant, par la suite, d'assurer la consistance du système en permettant, entre autres, de répercuter les changements d'un niveau de'abstraction élevé aux plus détaillés et vice versa.

L'hétérogénéité est une caractéristique important devant être supportée par les configurations d'architectures. En effet, les architectures logicielles sont essentiellement destinées à faciliter le développement d'applications ou systèmes de grandes tailles en réutilisant notamment des composants déjà existants ainsi que des connecteurs de différents niveaux de granularité. Ces composants et connecteurs pré-existants peuvent être issus de différents langages avec des exigences différentes en matière de systèmes d'exploitation, etc. La configuration d'architectures doit alors fournir un support pour la spécification et le développement d'un système à l'aide de composants et de connecteurs hétérogènes.

Les changements des plate-formes technologiques ou des besoins des utilisateurs nécessite l'évolution du système logiciel. Les configurations doivent donc être capables de répondre à ces évolutions en mettant à jour le système. Elle doit assurer l'ajout, le remplacement, la suppression et la reconnection de composants et de connecteurs.

Etant donnés les concepts formant les quatre *C*, il semble assez simple d'effectuer une taxinomie des différents ADL existants. Ces concepts décrivent en effet, des concepts communs fournissant une base de comparaison des différents ADL. Cependant, d'autres éléments plus spécifiques peuvent être nécessaire ou utiles pour la définition des architectures logicielles. Dans la section suivante, nous introduisons différents ADLs et leurs manières d'expliciter les quatre *C*.

3.3.2 Les Langages de Description d'Architectures (ADL)

Dans (90), les auteurs ont défini l'ADL par « Pour la spécification du logiciel, un ADL met plus l'accent sur la structure de haut-niveau de la globalité d'une application que sur les détails d'implémentation d'un module particulier ».

Les ADLs fournissent un vocabulaire de base pour la représentation des architectures des applications logicielles. Ils introduisent des syntaxes et sémantiques formelles pour la modélisation des architectures. Un ADL fournit donc un vocabulaire commun pour une communication mutuelle entre les concepteurs, les analystes, les développeurs, et les « déployeurs » dans le cadre de la construction des éléments d'une architecture. L'ADL facilite la construction, le déploiement et la configuration des systèmes logiciels. La gestion de configurations définit l'interaction des composants pour l'accomplissement d'une tâche importante du système logiciel. De plus, les ADL facilitent la compréhension des systèmes complexes et aident leur évolution et leur réutilisation.

La communauté du génie logiciel a proposé les ADL comme des notations de modélisation et des techniques de description des architectures des systèmes. L'objectif principal est d'améliorer la réutilisabilité et d'éviter les problèmes dus à des spécifications intuitives et non formelles des architectures logicielles.

Nous dresserons un rapport sur les ADL clés ou significatifs proposés dans la littérature du domaine.

Il existe plusieurs familles d'ADL destinées notamment à la représentation des systèmes distribués(91). Ce sont :

- Les ADL formels qui décrivent les architectures de façon formelles. Ils supportent les concepts abstraits de composants et de connecteurs sans indiquer leurs correspondance ou leur projection dans le système réel. Parmi ces ADL„ nous pouvons citer Wright (développé par le CMU : Carnegie Melon University)(86), Rapide (développé à Stanfordf)(78), et Acme (développé au CMU)(87), etc.
- Les ADL spécifiques qui spécifient la notion de composition en introduisons différentes catégories correspondant chacune à des entités logicielles ou matérielles particulières. Parmi ces ADLS on distinguera UniCon (développé au CMU)(76) et AADL (standardisé par SAE)(92), etc.

Les deux familles d'ADL incluent les aspects nécessaires à notre approche. L'aspect formel est crucial pour la modélisation et la vérification de l'architecture alors que les ADL spécifiques permettent une description plus concrète de l'architecture et une traçabilité entre les concepts de différents nivaux d'abstraction (jusu'au x codes sources des applications). Ce qui est crucial pour une analyse du changement. Nous discuterons, par la suite, des ADL de ces deux familles pour identifier les concepts communs a adopter dans notre approche.

Les ADL utilisent différentes terminologies. Cependant, ils permettent la spécification d'un composant et de son interface et son type. Ils permettent également de représenter les interactions entre composants d'une architecture. L'approche proposée par la majorité des ADL est plus statique et le comportement dynamique de l'application n'est pas exprimé.

Ces ADL permettent un support minimal de l'évolution d'une architecture vers une nouvelle configuration durant la construction de l'application. Dans la suite, nous prenons donc en compte les concepts communs de ces ADL et proposons une approche de gestion de l'évolution et de l'analyse de l'impact du changement.

Les ADL ont longtemps souffert de leur autarcie au niveau académique et de la rareté de leur utilisation dans l'industrie. Ceci est du à un manque de standardisation au niveau du langage pendant longtemps. Récemment, l'inclusion du concept de composant dans UML (Unified Modeling Language) (93), le développement et la large audience de l'OMG (Object Management Group) pour la conception d'applications objets et l'émergence du langage AADL (Architecture Analysis & Design Language) défini par SAE (Society of Automotive Engineer), ont montré le besoin des constructeurs de disposer d'un langage standardisé pour la description des architectures logicielles.

Pour illustrer notre approche, à travers ce manuscrit, nous avons adopté le langage AADL. L'un des aspects intéressants de AADL et l'inclusion de plusieurs composants sémantiques, entre autres et la description des entités matérielles qui n'est pas définie dans les autres ADL. De plus, il permet la spécification de l'architecture d'une application jusqu'à un niveau de détail très poussé. Il permet également, par la notion de propriété, d'enrichir les descriptions par des aspects fonctionnels tels que les codes source implémentant les composants non fonctionnels. Cela nous permet d'exprimer les besoins de l'utilisateur et les propriétés du déploiement et de la configuration du système.

AADL a déjà reçu une large acceptation chez les constructeurs et particulièrement ceux relevant du domaine du transport. De plus, des travaux récents ont été effectués pour une spécification et une génération automatiques des systèmes à partir de description architecturale AADL (94). Notre travail a pour but d'exploiter ceci dans le contexte de la gestion de l'impact du changement des architectures sur le code source, et vice versa. Dans la section suivante, nous présenterons le langage standard AADL.

3.4 AADL : Architecture Analysis and Design Language

AADL est un langage de modélisation standard développé par SAE (Society for Automative Engineers) (84) et destiné principalement à la modélisation de systèmes distribués et temps réel. Du point de vue de l'utilisateur, une spécification AADL et ses constituants peuvent être exprimés textuellement, de façon graphique ou en combinant ces deux représentations. Ils peuvent également être représentés par un document XML (eXtensible Markup Language)(95). Les notations textuelles et graphiques sont définies par le standard de la SAE AADL et ses extensions (84). La figure 3.2 (source de la figure [1]) montre une partie d'une spécification AADL selon les vues textuelle, graphique et celle utilisant XML [2]. AADL supporte une analyse itérative de l'architecture d'un système en prenant en compte, entre autres, des propriétés liées aux performances et en utilisant une notation extensible, un framework ou cadriciel composés d'un ensemble d'outils et d'une sémantique définie de façon précise.

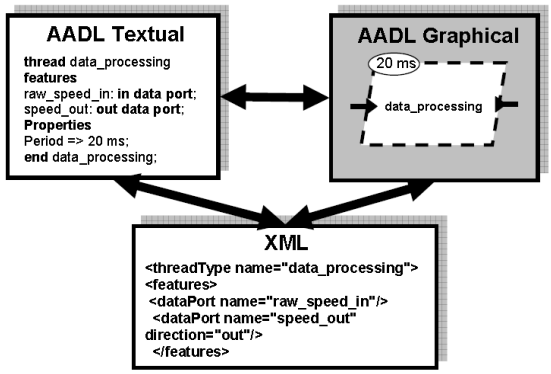


Figure 3.2: Différentes vues d'une description de l'architecture en AADL

AADL utilise des concepts formels pour la description et l'analyse d'architectures en termes de composants et de leurs interactions. Il a, à l'origine, été défini par divers patterns relatifs au domaine de l'avionique. En effet AADL voulait dire au départ Avionics Architecture Description Language. Dans la pratique, AADL est utilisé dans le domaine plus général des systèmes temps-réel et embarqués et n'est pas restreint qu'à celui de l'avionique. Les descriptions d'architectures avec AADL consistent en la description des composants et de leur organisation en forme d'arborescence. L'architecture

1. http://www.sei.cmu.edu/reports/06tn011.pdf
2. http://www.w3.org/TR/xml/

est une composition récursives de composants et de leurs sous-composants.

3.4.1 Les éléments AADL

Dans cette section nous explicitons les éléments constitutifs du langage ADL. Dans ce langage, les composants sont définis à travers les notions de type et d'implémentation. Les éléments de l'interface d'un composant et les points d'interaction avec d'autres composants, les spécifications de flots, et les valeurs des propriétés internes sont définis par la déclaration de Type de Composant ou Component Type Declaration. La structure interne d'un composant est constituée de ses sous-composants, de ses connexions, des séquences appels de sous-programmes, des implémentation des flux ou flots, et des propriétés. Les composants sont groupés en applications, plate-forme d'exécution, et catégories de composites. L'organisation des éléments de AADL en groupes nommés est assurés par la notion de packages. Les ensemble de propriétés et les librairies d'annexes aident à l'extension du langage et à la personnalisation ou customization de AADL aux exigences d'un domaine spécifiques. La figure 3.3 (source de la figure [1]) résument les concepts de base ou noyau de AADL.

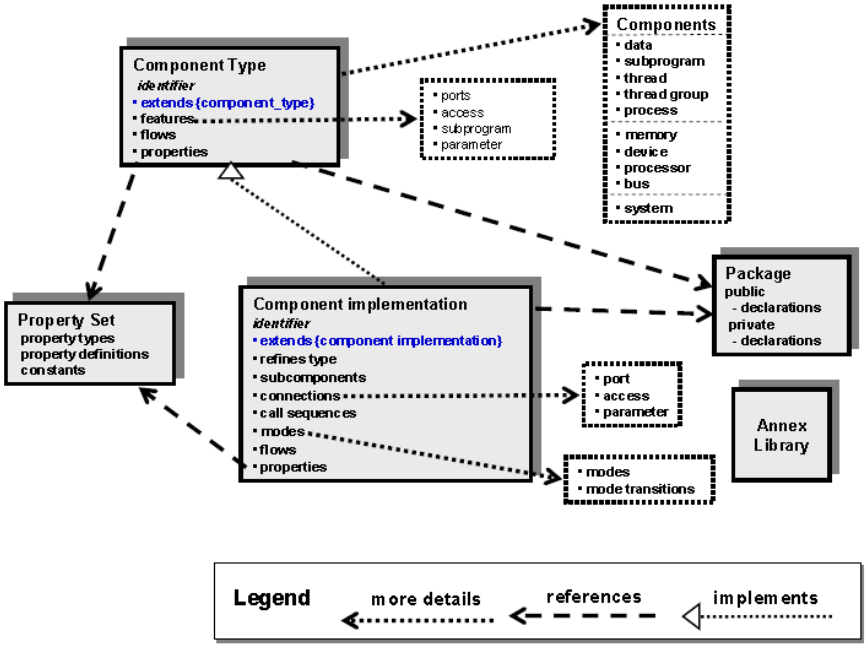


FIGURE 3.3: Les éléments clés de la spécification AADL

1. http://www.sei.cmu.edu/reports/06tn011.pdf

3.4.1.1 Les composants AADL

En plus d'être typé, chaque composant AADL appartient à une catégorie prédéfinie. Ces catégories représentent la manière ou nature de manipulation d'un composant. Elles sont divisées en trois :

- La catégorie des composants des applications logicielles servent à définir l'architecture d'une *application*. Les composants de cette catégories sont les *Processus*, les *Threads*, les *Groupes de Threads*, les *données* et les *sous-programmes*. Les processus sont des espaces mémoire pour l'exécution des *threads*. Les *threads* sont des éléments logiciels actifs pouvant s'exécuter en parallèle. Un *groupe de threads* est un ensemble de threads partageant les mêmes propriétés. Il peut également représenter une hiérarchie de threads. Les données représentent les structures de données pouvant être stockées ou échangées entre composants. Un sous-programme est un code exécutable séquentiel. Un *sous-programme* peut être appelé par un autre *sous-programme* ou *thread* et est considéré comme atomique.
- La catégorie des composants *plate-forme d'exécution* englobe la description du modèle dit hardware ou matériel. Les composants de cette catégorie peuvent être des *processeurs*, des *mémoires*, des *périphériques* ou des *bus*. Un *processeur* exécute des threads ou des sous-programmes. La *mémoire* stocke des données et des codes. Un *périphérique* est une sorte d'interface qui représente l'environnement externe. Un *bus* fournit la circulation des interactions entre les différents composants de la catégorie plate-forme d'exécution.
- Les composants de la catégorie *système* sont des éléments composites constitués de composants *software* ou *applications*, de composants de la catégorie *plate-forme d'exécution* ou d'autres composants de la catégorie *système*.

Dans nos travaux, nous focalisons, dans un premier temps, sur les composants logiciels (de la catégorie software ou application) dans le cadre de la gestion du changement de la structure d'une architecture logicielle.

3.4.1.2 Les types de composants

Les caractéristiques externellement visibles d'un composant sont matérialisé par un type de composant. Il définit un point de vue externe du composant fournissant ainsi une vue orientée-service à destination des autres composants qui peuvent le solliciter ou

qu'il peut solliciter. Un type de composant correspond à une interface fonctionnelle du composant et les ports du composants sont défini par son interface. La déclaration d'un type de composants consiste en la définition de clauses et de sous-clauses descriptives. Par exemple, une déclaration de type spécifie l'interface d'un composant de la catégorie Thread. Les sous-clauses peuvent être des caractéristiques, des flux ou des propriétés. Les caractéristiques ou features sont les interfaces du composants. Les flux spécifient des canaux abstraits pour le transfert de l'information. Les propriétés définissent les caractéristiques intrinsèques d'un composant. Il existe des propriétés prédéfinies pour chaque catégorie de composants telles que le temps d'exécution pour les threads, etc.

Il existe également la sous-clause extends permettant de définir un composant comme une extension d'un autre composant. Un composant ainsi déclaré hérite les caractéristiques du composant qu'il étend. Il est également possible d'ajouter ou d'étendre avec de nouvelles interfaces et propriétés et de nouveaux flux. Des éléments qui auraient été partiellement déclarés dans le composant originel (à étendre) peuvent être détaillés et des propriétés peuvent être modifiées ou raffinées. Cela permettra par la suite de définir une variété de familles de composants par le biais de la sous-clause extends.

3.4.1.3 Les implémentations de composants

Une implémentation de composant spécifie sa structure interne en termes de sous-composants, d'interactions (appels et connexions). Parmi les caractéristiques de ces sous-composants on définira notamment les flux à travers les séquences de sous-composants, es modes représentant les états opérationnels et les propriétés.

Les sous-clauses de la déclaration d'implémentation peuvent être les sous-composants, les connexions, et les appels. Elles spécifient la composition d'un composant comme une collection de sous-composants et leurs interactions. Les flux représentent l'implémentations des spécifications de flux apparaissant dans le type du composant ou des flux « end-to-end » à analyser (i.e. des flux qui commencent dans un sous-composant, vont de zero à plusieurs autres sous-composants, et finissent dans un autre sous-composant). Les modes représentent les modes opérationnels alternatifs apparaissant comme des configurations alternatives de sous-composants, de séquences d'appels, de connexions, de séquences de flux et de propriétés. Les propriétés définissent les caractéristiques intrinsèques d'un composant. Il existe des propriétés prédéfinies pour chaque implémentation de composant.

Une implémentation de composant peut étendre et raffiner d'autres implémentations préalablement déclarées. Les implémentations étendues (déclarées avec la sous-clause extends) héritent les caractéristiques de l'implémentation d'origine. Le raffinement permet à des implémentations partiellement spécifiées (templates) d'être complétée alors que l'extension permet à une implémentation d'être exprimée comme une variante d'une description de composant commune. De plus, une déclaration d'implémentation étendue peut ajouter des valeurs de propriétés aux caractéristiques de son type correspondant. Ces additions peuvent être effectuées à travers la sous-clause refine.

3.4.1.4 Les packages

Les packages permettent d'organiser des collections de déclarations de composants comme des unités séparées ayant leurs propres *espaces de noms*. Les éléments ayant des caractéristiques communes (e.g. tous les composants relatifs à la communication réseaux) peuvent être regroupés dans un package utilisant un nom spécifique. Les packages peuvent être utilisés comme support pour le développement indépendant de modèles AADL de différents sous-systèmes d'un système de très grande taille et cela grâce à l'utilisation d'*espaces de noms* distincts.

3.4.1.5 Les ensembles de propriétés

Un ensemble de propriétés est un groupe nommé de déclaration de propriétés définissant de nouvelles propriétés et des types de propriétés pouvant être inclues dans une spécification. Par exemple, un ensemble de propriétés relatives à la sécurité peut inclure des définitions des niveaux e sécurités requises pour l'utilisation d'un S.G.B.D. (Systèmes de Gestion de Bases de Données). Ces propriétés sont référencées en utilisant le nom de l'ensemble de propriétés et peuvent être associées à des composants et à d'autres éléments de modélisations (e.g. les ports ou les connexions).

3.4.1.6 Les annexes

Une annexe permet à un utilisateur d'étendre le langage AADL par l'incorporation de notations spécialisées. Par exemple, un langage formel permettant l'analyse d'aspects critiques d'un système (e.g l'analyse de la fiabilité, la sécurité, ou le comportement) peut être inclus à l'aide de spécifications AADL.

3.4.2 Modèles et Spécifications systèmes AADL

Un modèle de système AADL décrit l'architecture et l'environnement d'exécution d'une application en termes de ses constituants logiciels et ceux relatifs à la plate-forme d'exécution. Un modèle AADL est défini par une spécification constituée de déclarations AADL syntaxiquement et sémantiquement correctes. Un modèle complet de système AADL inclus toutes les déclarations requises pour créer une instance exécutable de l'application représentée par la spécification.

3.4.3 Les interactions entre composants

Les interactions entre les comosants sont définies à travers les connexions entre les interfaces. Ces connexions établissent une des interactions suivantes :

1. Les ports sont des connexions constituant la façon principale de décrire la transmission de l'information entre les composants. Un port est une interface de communication pour l'échange directionnel de données, d'événements, ou des deux entre des composants. Dans AADL, les ports sont déclarés comme des "features" ou caractéristiques dans la déclaration des types de composants.

   Les ports sont directionnels et déclarés comme entrées, sorties ou entrées/sorties :
   - Les ports événements représentent l'interface de communication des événements mentionnés par les routines et les threads. Ils sont similaires aux signaux des systèmes d'exploitation et peuvent déclencher l'exécution de threads.
   - Les ports données sont des interfaces de transmission de données entre les composants. La connexion entre les ports de données peut être immédiate ou différée. Cela veut dire que la donnée doit être transférée immédiatement ou avec un délai. Contrairement aux ports d'événements, les ports de données ne peuvent pas déclencher l'exécution d'aucune action.
   - Les ports événements données sont des interfaces de transmission de messages (signaux et données). Ils peuvent transporter des données en générant des événements lors de leur réception.
2. Les appels de procédures (subroutines) sont déclarées dans la définition d'un thread ou l'implémentation d'une subroutine (procédure). Ils peuvent être assimilés à des appels de procédures dans la programmation impérative.

3. Les connexions de types paramètres représentent la transmission d'information entre sous-programmes ou entre threads et procédures. Un paramètre est un argument d'une subroutine ou procédure.

3.5 Correspondance entre les éléments AADL et SMSE

Nous utilisons un exemple d'architecture décrivant le modèle de producteur/consommateur. Le listing 3.1 montre une description textuelle de l'architecture correspondant à ce modèle exprimée par les concepts de AADL. Il décrit la production de donnée par le producteur et la réception de la donnée par le consommateur. Cet exemple définit un processus *processusA.impl* contenant deux threads : *Prod.impl* de type *Producer* désignant le producteur et *Cons.Impl* de type *Result_Consumer* représentant le consommateur. Le paramètre de sortie *data_source* de l'instance du sousprogramme *P_Spg* est connecté au port d'entrée *Data_source_out*. Le thread *Producer* qui envoie le résultat au port de sortie de *AlphaprocessProcessusA.impl*. Il envoie la donnée à travers ses ports d'entrées au port d'entrée du thread *BetaData_Sink_inResult_Consumer* qui appelle la subroutine instance *Q_Qpg* à travers ses paramètres d'entrée *Data_Sink*. Le producteur et le consommateur sont connectés aux autres composants.

Listing 3.1: Exemple d'une représentation AADL

```
1  subprogram Produce_Spg
2  features
3    Data_Source : out parameter ;
4  end Produce_Spg;
5
6  thread Prod
7  features
8    Data_Source_out : out data port;
9  end Prod;
10
11 thread implementation Prod.Impl
12  calls  {
13   P_Spg : subprogram Produce_Spg;
14 };
15 connections
16   PC1 : parameter P_Spg.Data_Source -> Data_Source_out;
17 properties
18   Dispatch_Protocol => Periodic;
19   Period => 200 Ms;
20 end Prod.Impl;
21
22 process processusA
23 features
24   AlphaA : out data port Alpha_Type;
25   BetaA  : in  data port Alpha_Type;
```

```
26 end processusA;
27
28 process implementation processusA.Impl
29 subcomponents
30   Producer : thread Prod.Impl;
31   Result_Consumer : thread Cons.Impl;
32 connections
33   PC3 : data port Producer.Data_Source_out -> AlphaA;
34   PC4 : data port BetaA -> Result_Consumer.Data_Sink_in;
35 end processusA.Impl;
```

Nous tentons d'établir une correspondance entre les éléments de AADL et ceux du modèle SMSE destinés à l'analyse d'impact du changement des descriptions architecturales. Les éléments « interface de composants », « implémentation de composants », « port », « paramètres », « connexion », et « propriétés » sont représentés comme des ensembles d'éléments cibles du changement dans SMSE. Les relations entre « interface ou implémentation » d'un composant et de « propriétés », « interfaces » et « port », « interface ou implémentation » et « connexion », et entre « connexion » et « port » sont représentées comme des relations conductrices d'impact du changement.

Selon la discussion ci-dessus nous résumons les types de relations entre les éléments AADL par la table 3.1.

Nous considérons les éléments communs à la plus part es ADL. La construction d'une architecture dans un ADL spécifique peut être définie par sa grammaire et représentée dans SMSE. La table 3.1 illustre les types de relations de base selon AADL. La première colonne décrit les types de relations, et les éléments AADL correspondants sont indiqués dans la deuxième colonne. La troisième colonne définit les conditions de la propagation de l'impact du changement dans le cas de toute modification affectant un des éléments de la deuxième colonne. Pour une illustration détaillée, nous considérons l'exemple de code AADL donné dans le listing 3.1 et nous identifions les conditions de dépendances telles que listées par la table 3.2.

Pour plus de clarification, nous expliquons quelques éléments de la table 3.2. Considérons la troisième ligne, qui explicite les conditions de la propagation de l'impact du changement d'un artefact impliqué dans la relation $hasPort$. Cette condition signifie que s'il existe deux artefacts logiciels art_x et art_y tel que art_x est une interface de composant et art_y appartient à la catégorie port de donnée, port d'événement ou port de donnée événement et il existe une relation $hasPort$ entre art_x et art_y alors tout changement sur art_y concernera art_y et vice versa. Le même cas concerne la septième

Table 3.1: *Les Types de relations entre les éléments du langage AADL et leur conditions*

Relationship Type	Élément AADL	Conditions
Définition de l'interface du composant	System, Process, Thread, Subprogram, or Data	$< Lv_{system,process,thread,subprogram,data}, art_x > \wedge Interface(art_x)$
Définition de l'implémentation du composant	System, Process, Thread, Subprogram	$< Lv_{system,process,thread,subprogram}, art_x > \wedge Implementation(art_x)$
Déclaration des sous-composants	System, Process, Thread	$< Lv_{system,process,thread}, art_x > \wedge < Lv_{system,process,thread}, art_y > \wedge subcomponent(art_x, art_y)$
Subprogram call	Subprogram	$< Lv_{subprogram}, art_x > \wedge < Lv_{subprogram}, art_y > \wedge call(art_x, art_y)$
Port	Data port, Event port, Event data port	$< Lv_{dataPort,eventPort,eventDataPort}, art_x > \wedge Port(art_x)$
Propriété	Properties	$< Lv_{properties}, art_x >$
Connexion	Data connection, Event connection, Event data connection, Parameter connection	$< Lv_{dataCon,eventCon,eventDataCon,paramCon}, art_x >$
Paramètres de sous-programmes	Subprogram, Parameter	$< Lv_{subprogram}, art_x > \wedge < Lv_{param}, art_y > \wedge Parameter(art_x, art_y)$

ligne. Elle illustre la relation property association. Elle schématise le fait que deux artefacts art_x et art_y tel que art_x est un composant et art_y est une propriété et qu'il existe une association entre art_x et art_y. Cela signifie que le changement de l'artefact art_x concerne le changement de art_y et vice versa.

This shows the architecture modeling process starting from the grammar of the AADL language and indicating the change impact propagation conditions, as explained in SMSE. These conditions are used and formalized in a graph on which *a priori* change operations can be applied. We discuss the change impact propagation process and graph elements in chapter 4.

3.6 Conclusion

La prolifération, ces dernières années, d'applications distribuées hétérogènes a fait que des nombreuses applications ont été conçues en faisant appel à des langages architecturaux. Cela offre une description de haut niveau d'abstraction améliorant la

Table 3.2: *Les instances de relation sur le code AADL et les conditions de la propagation d'impact du changement*

Type de la relation	**Description de la relation**	**Les conditions**
Lien entre un sous-composant et sa définition (ex : *ligne 30 ou 31 du listing 3.1*)	Relation « aType » entre « Interface » ou « Implémentation » et « Instance »	$< Lv_{component}, art_x > \wedge < Lv_{object}, art_y > \wedge hasType(art_x, art_y)$
Lien entre l'interface d'un composant et son implémentation (ex : *lignes 6 et 11 ou 22 et 28 du listing 3.1*)	Relation entre « Interface » et « Implémentation »	$< Lv_{component}, art_x > \wedge < Lv_{object}, art_y > \wedge Interface(art_x) \wedge Implelementation(art_y) \wedge impFor(art_x, art_y)$
Lien entre l'interface d'un composant et ses ports (ex : *lignes 6 et 8 ou 22 et 24 du listing 3.1*)	Relation « possède » entre « Interface » et « Data port, Event port, Event data port »	$< Lv_{component}, art_x > \wedge Interface(art_x) \wedge < Lv_{dataPort,eventPort,eventDataPort}, art_y > \wedge hasPort(art_x, art_y)$
Lien entre l'implémentation d'un composant et ses connexions (ex : *lignes 11 et 16 ou 28 et 33 du listing 3.1*)	Relation entre « Implémentation » et « Connexions »	$< Lv_{component}, art_x > \wedge Implelementation(art_x) < Lv_{dataCon,eventCon,eventDataCon,paramCon}, art_y > \wedge hasConnection(art_x, art_y)$
Lien entre l'implémentation d'un composant et ses sous-composants (ex : *lignes 28 et 30 du listing 3.1*)	Relation entre « Implémentation » et « Instance »	$< Lv_{system,process,thread}, art_x > \wedge < Lv_{system,process,thread}, art_y > \wedge hasSubComp(art_x, art_y)$
Lien entre une connexion et ses ports source et destination (ex : *lignes 16 ou 33 du listing 3.1*)	Relation « estRelié » entre « Port » et « Connexions »	$< Lv_{dataPort,eventPort,eventDataPort}, art_x > \wedge < Lv_{dataCon,eventCon,eventDataCon,paramCon}, art_y > \wedge estReli(art_x, art_y)$
Lien entre l'interface ou l'implémentation et ses propriétés (ex : *lignes 11 et 18 ou 11 et 19 du listing 3.1*)	Relation « associe » entre « Interface » ou « Implémentation » et « Propriété »	$< Lv_component, art_x > \wedge < Lv_{properties}, art_y > \wedge associe(art_x, art_y)$
Lien d'appel entre une implémentation d'un Thread et l'instance d'un sous-programme (ex : *lignes 11 et 13 du listing 3.1*)	Relation entre « Implémentation » et « Instance »	$< Lv_{thread}, art_x > \wedge Implementation(art_x) \wedge < Lv_component, art_y > \wedge call(art_x, art_y)$

modélisation des systèmes complexes pour leur développement, déploiement ou leur évolution.

La description architecturale est formulée par ADL, permettant de spécifier les composants d'application et de leurs interactions via les connecteurs. Pour évaluer l'impact d'une modification apportée à un élément architectural, il est nécessaire d'extraire la

description des composants et leurs interactions. Dans ce chapitre, nous avons appliqué le modèle SMSE sur l'architecture logicielle décrite par un ADL. Une revue générale a été donnée des techniques utilisées pour l'analyse architecturale des systèmes logiciels. Pour un objectif de validation, l'application du SMSE a été illustrée sur des architectures logicielles décrites dans un langage architectural largement utilisé l'AADL (Architecture Analysis and Design Language). Le chapitre a examiné en détail les éléments de représentation préconisés par un AADL pour établir une mise en correspondance entre ces éléments et leur représentation en SMSE.

Le but étant de montrer la validité du modèle pour l'analyse l'impact du changement d'éléments architecturaux qui sont essentiellement formés de composants, connecteurs, ou de configurations décrivant les détails des architectures logicielles.

Ce chapitre a, en fait, prolongé la démarche d'application de l'analyse structurelle du logiciel sur les descriptions architecturales des logicielles. Nous avons ensuite illustré par un exemple d'application l'utilisation des règles de changement pour réaliser l'analyse correspondante à une évolution architecturale . Le but est de montrer l'applicabilité de l'approche proposée pour l'analyse de modification d'un élément architectural du logiciel et son impact sur le code source d'implémentation en termes de mise à jour afin de préserver la cohérence entre l'élément architectural modifié et son code d'implémentation.

Chapitre 4

Modélisation de la Propagation de l'Impact du Changement

4.1 Introduction

L'évolution du logiciel peut être vue comme un ou plusieurs changements opérés sur un ou plusieurs artefacts logiciels. Du fait des dépendances fortes des divers artefacts logiciels, ces changement ont de fortes chances d'affecter le reste du logiciel. Il est donc nécessaire de mener une analyse de l'impact de tout changement en identifiant de façon précise les différents éléments qu'il affecte. Une des difficultés majeures pour mener ce genre d'analyse et qui est encore plus accrue dans le cas des systèmes logiciel distribués consiste en l'existence d'un phénomène appelé effet de vague ou ripple effect se manifestant par une propagation des effets de tout changement aussi minime soit-il à un très grand nombre d'artefacts. Il est donc primordiale de mettre en oeuvre des mécanismes d'analyse *a priori* permettant de mieux comprendre et cerner la propagation de l'impact du changement des systèmes logiciels afin d'éviter des situations incontrôlés survenant généralement après le changement(7).

La compréhension du logiciel est d'une nécessité cruciale pour toute mise en oeuvre de changements. La difficulté du processus de compréhension est liée au nombre très important de lignes de code distribuées sur plusieurs plate-formes de développement et du nombre de plus en plus important de chemins logiques de leurs exécutions. L'analyse du changement et de la propagation de son impact peut donc nécessiter une description exhaustive des constituants du logiciel et de leurs interdépendances. Il nécessite

également la disposition d'un référentiel exploitable de façon automatique (une base de données par exemple) et servant au stockage et à la manipulation de la quantité phénoménale de données ou objets constituant le logiciel et qu'on appelle artefacts. De notre côté, nous adoptons le modèle SMSE comme base de mise en oeuvre d'un système à base de connaissances permettant de mettre en place des processus semi-automatique de compréhension et d'analyse du changement du logiciel. Les résultats majeures de recherche cette approche se manifestent par deux principaux constituants et qui sont une base de faits décrivant de façon assertionnelle l'ensemble des artefacts logiciels et de leurs interdépendances et une base de règles incarnant des règles génériques et d'autres plus spécifiques dédiées à la compréhension et à l'analyse du changement du logiciel et de son impact.

Dans le cas des applications logicielles faisant intervenir de nombreux artefacts interagissant dans un environnement distribué, l'incorporation du changement et l'analyse de ses impacts devient très complexe pour un acteur unique. En effet, ce dernier est rarement l'auteur originel de l'artefact à modifier et ne dispose pas forcément d'une vision ou vue claire des différentes interactions. Dans certains cas, l'acteur modifiant un artefact donné ne dispose pas forcément de droits lui permettant d'accéder et/ou de modifier les autres artefacts affectés par le changement. Ce type de situations survient de plus en plus avec la généralisation des applications distribuées et rendent les tâches associées à l'évolution du logiciel de plus en plus délicates à mettre en oeuvre dans la pratique.

Nous proposons l'application du modèle SMSE (défini dans le chapitre 2) comme support pour la mise en oeuvre effective des opérations de maintenance ou d'évolution du logiciel. La compréhension des artefacts cibles du changement, la propagation de l'impact ainsi que d'autres exigences du processus d'évolution sont des activités assez onéreuses en matière de temps et d'effort et il convient de les accompagner en développant des modèles et des outils qui les supportent et maximisent leurs rendements. L'un des aspects principaux de l'analyse de l'impact du changement est l'aspect structurel. En d'autres termes, cet impact est concerne les modifications et les effets de vague qui affectent la structure du logiciel et cela est valable pour toute opération de changement qu'elle soit une simple modification de paramètre dans une procédure au remplacement d'un paquetage par deux autres, etc.

Dans ce chapitre, nous présentons avec plus de détails les résultats de nos travaux concernant l'analyse de l'impact du changement du logiciel avec un intérêt particulier pour les applications distribuées. Cela nous a conduit à la construction d'un système à base de connaissances intégrant les concepts de base nécessaire pour l'analyse et la propagation du changement de la structure du logiciel.

Le chapitre est organisé de la façon suivante : la section 4.2 présente la structure générale du système à base de connaissances et ses différents constituants. La section 4.3 établit une typologie des opérations de changement structurel du logiciel. La section 4.4 discute de la modélisation des opérations de changement structurel. La section 4.5 fournit un mécanisme pour une définition d'un ensemble dit générique de règles de propagation de l'impact du changement. La section 4.6 montre l'analyse de l'impact du changement à travers les différents types de relations inter-artefacts. Finalement la section 4.7 conclut le chapitre.

4.2 Le système à base de connaissance

La mise en oeuvre effective du processus d'analyse de l'impact du changement nécessite une description exhaustive des différents artefacts logiciels. Cela engendre un nombre gigantesque de données. Il est donc nécessaire de disposer d'une « sorte de moyen » permettant la représentation de cette connaissance d'une façon simple et explicite tout en assurant la consistance du processus d'évolution du logiciel. L'utilisation des bases de connaissances peut aider dans la compréhension, à la fois du logiciel et du processus d'évolution. De telles bases sont utilisées depuis plusieurs années dans le cadre de la mise en oeuvre automatiquement assistée de différentes activités du génie logiciel. Cela à donné naissance à la notion de KBSE (*Knowledge-Based Software Engineering*) (96).

Nous adoptons l'approche consistant à mettre en oeuvre un système à base de connaissances pour le stockage et la manipulation de méta-information concernant les applications logicielles et leurs constituants, et le contrôle et la supervision des opérations de changement. L'avantage de cette approche consiste en une séparation du code des applications logicielles et des règles régissant leur évolution. Il existe donc une séparation entre la logique métier supportée par l'application et les règles de gestion de l'évolution de cette application.

Le système à base de connaissances, schématisé par la figure 4.1, est mis en oeuvre par un système à base de règles. La connaissance de base de ce système est matérialisée par des faits qui sont extraits en analysant par des parseurs spécifiques les différents constituants de l'application. Certains faits sont obtenus par analyse statique du code source alors que d'autres le sont par analyse dynamique des performances (profiling), par analyse de document de conception ou par le biais d'experts. La base de faits est utilisée, dans le cadre du raisonnement, par une base de règles dont la fonction première est d'implémenter le processus d'analyse et de propagation de l'impact du changement. Ces règles peuvent être classées en packages ou catégories. Ainsi, un ensemble de règles est dédié à la construction du graphe d'artefacts logiciels, d'autres règles intègrent les règles de propagation de l'impact et d'autres règles qualifiées de dynamiques sont construites à la volée au cours de l'analyse de l'impact selon les besoins de l'expert ou du chargé de l'évolution.

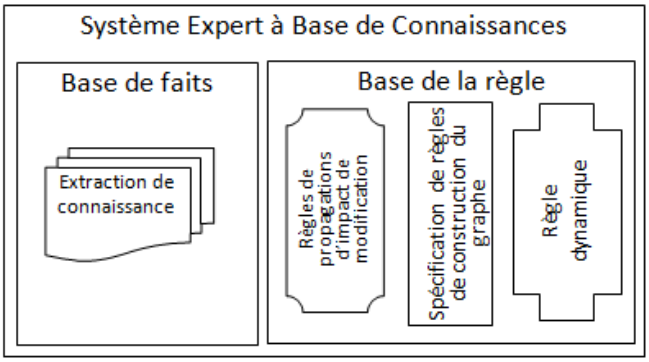


FIGURE 4.1: Le schéma du système à base de connaissance pour l'évolution de logiciels

4.2.1 La base de fait

La base de faits contient une connaissance pragmatique provenant, entre autres, de l'analyse statique des codes sources, de l'analyse de documents semi-structurés de description du déploiement et de l'architecture, etc. Dans notre approche, nous nous proposons d'exploiter les dépendances, décrites sous la forme de méta-connaissances dans la base de faits, pour modéliser les artefacts logiciels sous la forme de graphes. Par la suite, les changements seront "tracés" ou leurs effets identifiés en appliquant des règles de propagation de l'impact sur les éléments des graphes ainsi constitués.

Une partie de la base de faits est donc une transcription sous une forme assertionnelle de la structure de graphe. La base de faits contient également des informations ou faits dont la présence peut déclencher des actions relatives à l'analyse et la propagation de l'impact. Ainsi, des faits peuvent représenter des noeuds et des arcs du graphe des artefacts logiciels alors que d'autres peuvent représenter des événements tels que la création d'un noeud ou la suppression d'un arc, etc.

Le déclenchement des actions est régi par un moteur d'inférence qui est guidée par des règles définies dans une base de règles.

4.2.2 La base de règles

Ce que nous entendons par base de règles est constituée d'un moteur de génération de règles qui, partant d'un type de relation et des caractéristiques de ce type par rapport à la conduction de l'impact, produit des règles traduisant la propagation de l'impact à travers ce type de relation. Certaines de ces règles sont intuitives puisqu'elles reflètent la propagation de l'impact à travers des relations directement dérivables de l'analyse du code. C'est le cas notamment des règles traduisant la conduction de l'impact à travers la relation d'appels de méthodes, etc. D'autres règles, plus génériques, permettent la génération de l'impact à partir de relations indirectes traduisant une composition de relations de base comme celles obtenues par analyse du code. C'est le cas par exemple, d'une relation générique de dépendance entre deux classes traduisant le fait qu'une classe utilise une variable d'instance d'une autre classe ou appelle une méthode de cette classe et cette dépendance peut être directe ou indirecte (exploitation de la transitivité), etc. Certaines autres règles sont encore moins évidentes à définir puisqu'elles traduisent des dépendances dites cachées. C'est le cas par exemple, d'une relation pouvant exister entre deux modules dépendant et traduisant le fait que l'amélioration des performances du premier module entraîn une détérioration de celles du second module ou inversement. Ce type de dépendances ne peuvent s'obtenir que par analyse dynamique du logiciel avec des techniques de type profiling ou instrumentation du code, etc. (97, 98, 99). Ce dernier type de règles ainsi que toute sorte de règles plus générales sont directement écrites par l'utilisateur humain qui est un des acteurs participant au projet d'évolution du logiciel. Nous avons pour cela, adopté des outils utilisant les notions de DSL (Domain Specific Languages) permettant à chacun de ces utilisateurs qui peuvent avoir des profils différents d'utiliser leur propres vocabulaire dans la définition des règles.

La spécification des règles intègre des détails significatifs par rapport à la conduction de l'impact des modifications du logiciel, les actions devant être effectuées pour la gestion de cet impact et l'historique des décisions importantes déjà prises concernant l'artefact traité. Cela constitue une aide pour la gestion de la traçabilité du changement d'un artefact donné et un support de prise de décision durant le processus de changement de cet artefact. Si nous considérons la relation de composition définie entre une classe et une méthode (figure 4.2), le changement affectant la méthode produit un flux d'impact affectant la classe.

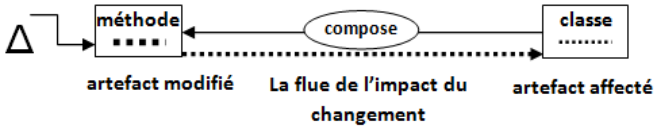


FIGURE 4.2: Propagation de l'impact à travers la relation de composition

Un système à base de règles assure deux principales fonctionnalités. La première fonctionnalité est matérialisée par toutes les règles identifiant les différents types de relations entre les différents artefacts. L'autre fonctionnalité est encapsulée par les règles traduisant la propagation de l'impact par l'identification des différents artefacts et relations directement ou indirectement affectés par une opération de changement donnée. Ces règles de propagation d'impact du changement que nous appellerons règles stratégiques ou règles d'implémentation de la stratégie de propagation utilisent en grande partie une connaissance concernant les relations inter-artefacts ainsi que la qualification de ces relations par des renseignements sur leur conductivité de l'impact, etc. Nous allons, dans ce qui suit, donner quelques exemples illustrant ces différentes règles en nous basant principalement sur le code source décrit par le listing 4.1.

Listing 4.1: Propagation de l'impact du changement entre les artefacts d'une exemple

```
1  public class Class_X {
2      public void Method_A() {}
3      public void Method_B() {}
4  }
5
6  public class Class_Y {
7      public void Method_C(){
8          Class_X Object_Z = new Class_X();
9          Object_Z.Method_B();
10     }
11 }
```

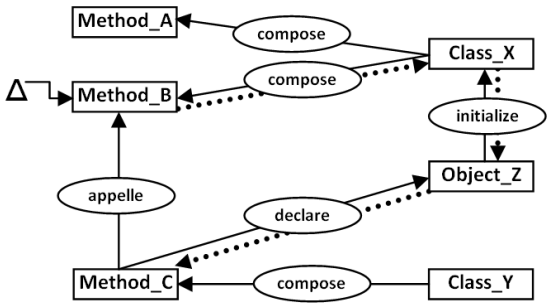


Figure 4.3: Propagation de l'impact du changement à travers les artefacts logiciels (1)

Dans le cas de la suppression de la méthode *Method_ B*, la règle stratégique propagera l'impact de cette suppression sur la classe *Class_ X*, qui est instanciée dans dans la méthode *Method_ C* de la classe *Class_ Y*. L'impact sera, par la suite, propagé à l'objet *Object_ Z* puis à la méthode *Method_ C*. Cette propagation de l'impact est décrite par la figure 4.3. D'un autre côté, lors de la suppression de la méthode *Method_ A*, la règle stratégique propage l'impact à l'objet *Object_ Z* et pas à la méthode *Method_ C* qui n'est pas concerné par ce changement (figure 4.4).

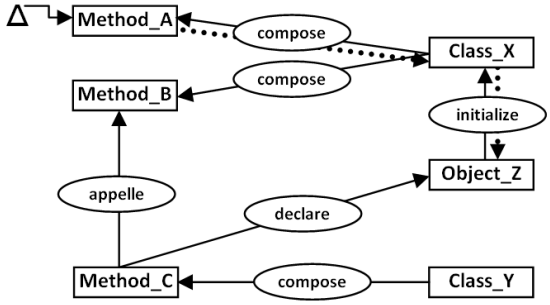


Figure 4.4: Propagation de l'impact du changement à travers les artefacts logiciels (2)

L'ensemble des règles est déclenchés par l'expert ou chargé de l'évolution de façon dynamique. En d'autres termes au fur et à mesure de la progression du processus de propagation de l'impact. L'occurrence d'un changement produit en effet, un événement se traduisant par un fait qui sera utilisé par le moteur d'inférence pour le déclenchement éventuel d'une règle de propagation de l'impact. Ces règles traite l'effet de l'opération de changement sur l'artefact directement affecté mais également sur les autres artefacts

qui lui sont directement ou indirectement reliés par divers types de relations.

Partant de ce principe, nous avons défini des ensembles cohérents de règles par "point de vue analysé". Ainsi, nous avons défini des règles qui ne considèrent que l'aspect ou point de vue structurel et d'autres règles qui considèrent l'architecture du logiciel, etc. Toutes ces règles fonctionnent selon le même principe qui consiste à exploiter les relations inter-artefacts. La différence étant dans la nature des relations et des artefacts considères. Ainsi, dans une analyse architecturale de l'impact nous allons focaliser sur les composants et les connecteurs, etc. Il existe par contre des règles gérant l'impact au niveau traçabilité comme celles qui répercutent l'impact de la modification d'un composant au sens architecture sur les codes sources qui le constituent, etc.

4.2.3 Exemple : propagation de l'impact au niveau d'un code source

Dans cette section, nous montrons le fonctionnement du processus de propagation d'impact utilisant la notion de faits en nous basant sur un exemple représentant une partie d'un code source C++. Le code exemple défini un ensemble d'artefacts (Σ_{art}) et de types de relations (Σ_{rel}). Tout changement sur un des artefacts peut affecter les autres.

Listing 4.2: Propagation de l'impact à travers des artefacts du code source - Fichier $art_x.h$

```
1  //prototypes
2  class art_w;
3  class art_x;
4
5  class art_w {
6      public: double art_p(art_x&x);
7      };
8
9  class art_x {
10     int art_a; double art_b;
11     // friendship
12     friend double art_w :: art_p(art_x&x);
13     public:
14     // constructor
15     art_x(inta, doubleb){ art_a = a; art_b = b; }
16     };
17
18     double art_w :: art_p(art_x&x) { return x.art_a * x.art_b; }
```

Listing 4.3: Propagation de l'impact à travers des artefacts du code source - Fichier $art_y.cpp$

```
1  #include < art_x.h >
2  //prototypes
```

```
3  void art_v(double);
4  double art_s();
5  double art_u();
6
7  //instantiation
8  art_x art_y(12, 14.7);
9  double art_z;
10
11 void art_v(double art_b){
12     double art_a;
13
14     //calling
15     art_a = art_s();
16
17     //modification
18     art_z = art_a * art_b;
19     }
20
21 double art_s(){
22     double art_q;
23
24     //declaration
25     art_w art_t;
26
27     /*invokes member function, effective parameter*/
28     Cq = art_t.art_p(art_y);
29     return art_q;
30     }
31
32 double art_u(){
33     double art_r;
34
35     //use
36     art_r = art_z * 1.16/5;
37     return art_r > art_z?art_z : art_r;
38     }
```

Les faits associés au code source du fichier $art_x.h$ (listing 4.2) traduisent les différents artefacts et relations de la manière suivante :

- artefacts et relations
  - $art_x, art_w \in < Lv_{cls}, \Sigma_{art} >$
  - $art_p \in < Lv_{method}, \Sigma_{art} > \wedge\ compose(art_w, art_p)$
  - $art_a, art_b \in < Lv_{sym}, \Sigma_{art} > \wedge\ compose(art_x, art_a) \wedge\ compose(art_x, art_a) \wedge\ Cx \in < Lv_{method}, \Sigma_{art} \wedge\ a, b \in < Lv_{sym}, \Sigma_{art} > \wedge\ parameter(art_x, a) \wedge\ parameter(art_x, b)$
- relations et règles de propagation de l'impact
  - $function(art_p, art_w) \in \Sigma_{rel}$
    $if \Delta_{art_p} then \Delta_{art_w}$
  - $friend(art_x, art_p) \in \Sigma_{rel}$
    $if \Delta_{art_x} then \Delta_{art_p}$

– $parameter(art_x, art_p) \in \Sigma_{rel}$
$if \Delta_{art_x} then \Delta_{art_p}$

– $use(art_x, art_a) \in \Sigma_{rel} \wedge \ use(art_x, art_b) \in \Sigma_{rel}$
$if \Delta_{art_a} \vee \Delta_{art_b} then \Delta_{art_x}$

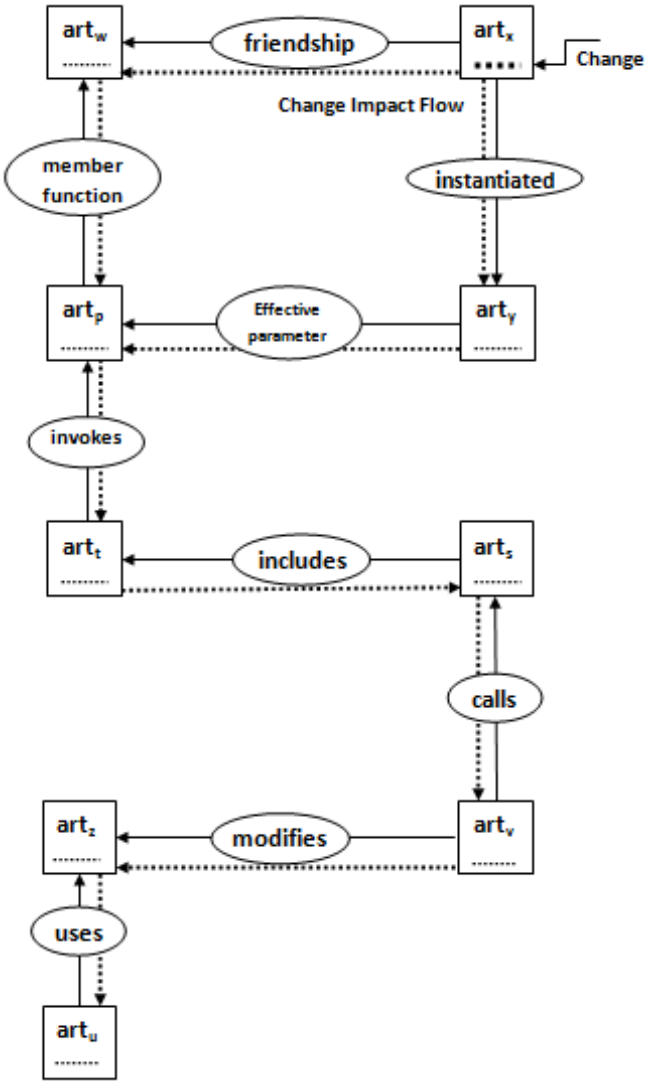


FIGURE 4.5: Propagation de l'impact du changement à travers les artefacts logiciels (3)

Les faits correspondant aux artefacts et relations extraits du fichier $art_y.cpp$ (listing 4.3) sont les suivants :

- artefacts
  - $art_v, art_s, art_u \in < Lv_{method}, \Sigma_{art} >$
  - $art_y \in < Lv_{obj}, \Sigma_{art} >$
  - $art_z \in < Lv_{sym}, \Sigma_{art} >$
  - $art_t \in < Lv_{obj}, \Sigma_{art} >$
- relations et règles de propagation de l'impact
  - $instance(art_y, art_x) \in \Sigma_{rel}$
    $if \Delta_{art_x} then \Delta_{art_y}$
  - $call(art_v, art_s) \in \Sigma_{rel}$
    $if \Delta_{art_s} then \Delta_{art_v}$
  - $modify(art_v, art_z), use(art_u, art_z) \in \Sigma_{rel}$
    $if \Delta_{art_z} then \Delta_{art_v}$
  - $declare(art_w, art_t) \in \Sigma_{rel}$
    $if \Delta_{art_w} then \Delta_{art_t}$
  - $include(art_w, art_s), include(art_t, art_s) \in \Sigma_{rel}$
    $if \Delta_{art_w} \vee \Delta_{art_t} then \Delta_{art_s}$
  - $invoke(art_t, art_p), parameter(art_y, art_p) \in \Sigma_{rel}$
    $if \Delta_{art_y} then \Delta_{art_p}$
    $if \Delta_{art_p} then \Delta_{art_t}$

Le flux d'impact du changement $\Delta_(art_x)$ sur l'artefact art_x est illustré par la figure 4.5.

4.3 Typologie des opérations de changement structurels

Tout changement structurel (Δ_S) affectant un artefact peut engendrer un changement correspondant dans les artefacts qui lui sont reliés. Le flux de l'impact des changement structurels implique que toute modification structurelle d'un artefact logiciel peut causer un changement structurel similaire au niveaux des artefacts qui en sont dépendants. Ces derniers peuvent être de différents niveaux abstro-granulaires. Un changement structurel peut traduire :

- l'ajout d'un nouvel artefact
- la suppression d'un artefact existant
- la modification d'un artefact existant

- la fusion de deux artefacts en un artefact d'un même niveau ou d'un niveau différent
- la décomposition ou fission d'un artefact en deux artefacts d'un même niveau ou de niveaux différents
- la création d'une relation entre deux artefacts
- la suppression d'une relation existant entre deux artefacts

Nous présentons, dans la section qui suit, la modélisation de ces opérations de changements structurels.

4.4 Modélisation des opérations de modifications

Dans un premier temps, nous proposons une formalisation de l'ensemble des opérations de changements structurels pouvant être appliquées à un modèle SMSE. La seconde étape consistera en la définition du processus générique de propagation des impacts qui peuvent être déclenchés ou générés durant les opérations de changement.

Cette section présente donc les opérations de base de manipulation de la structure du logiciel de façon assertionnelle. En effet, chaque opération sera spécifiée par la donnée de pré-conditions, de post-conditions et d'invariants. Les pré-conditions sont des assertions devant être vérifiées pour que l'opération de modification puisse s'exécuter, les post-conditions sont des assertions devant être vérifiées après l'exécution de l'opération alors que les invariants sont des assertions qui doivent être vérifiées avant et après l'exécution de l'opération.

L'une des raison qui nous ont poussés à choisir un tel formalisme pour la spécification des opérations de changement sont d'abord la large adoption de ce formalisme dans la communauté du génie logiciel et surtout au niveau des travaux concernant l'étude du comportement des programmes. Une autre raison est la proximité sémantique de ce type de formalisme avec la logique des prédicats et/ou des propositions ce qui facilite la transcription des opérations et de la gestion de leurs effets par des règles gérables par un système à base de connaissances et également la possibilité de procéder à des générations de plans d'exécution ou de rétablissement de cohérence. En effet, la majorité des systèmes de génération de plans adoptent la formalisation des opérations par des pré et post conditions (100).

Table 4.1: *Liste des assertions simples pour la manipulation du graphe de visualization de logiciel*

Notation	Signification
$+node(art)$	Existe un nœud ayant l'identifiant $art \in \Sigma_{art}$
$+arc(rel)$	Existe un arc ayant l'identifiant $rel \in \Sigma_{rel}$
$-node(art)$	N'existe pas un nœud ayant l'identifiant art
$-arc(rel)$	N'existe pas un arc ayant l'identifiant rel
$*node(art)$	Donne le réponse 'vrai' s'il existe un nœud ayant l'identifiant art
$*arc(rel)$	Donne le réponse 'vrai' s'il existe un arc ayant l'identifiant rel

Pour une meilleure compréhension des inter-connexions des artefacts, les éléments du modèle SMSE peuvent être représentés par des graphes. Un graphe $G = (\Sigma_{art}, \Sigma_{rel})$ peut être défini comme suit :

- L'ensemble des noeuds Σ_{art} représente les artefacts logiciels
- L'ensemble des arcs Σ_{rel} correspond aux relations reliant les artefacts

La table 4.1 résume des assertions ayant pour but de représenter les éléments d'un modèle SMSE. Les assrtions préfixés par le signe « + » indiquent la présence d'un fait (dans la base de fait). Cela veut dire qu'une telle assertion est vraie si le fait existe bel et bien dans la base de faits. Une assertion préfixé par le signe « − » indique l'absence d'un fait. Cela veut dire que cette assertion est vraie si le fait est absent de la base des faits. Une assertion préfixée par le signe « ∗ » est une sorte de question ou requête : elle renvoie vrai si le fait existe et faux si le fait n'existe pas dans la base de faits. De façon similaire, la table 4.2 spécifie de façon plus détaillée certaines assertions qui seront utilisées dans la spécification des opérations de modification ou changement du logiciel.

Nous avons défini également des assertions composites telles que celle définies par la table 4.3. Ces assertions ne sont rien d'autres que des expressions plus concises d'une ensemble d'assertions de base (pour alléger l'écriture). Par exemple, l'assertion $+arc(rel, art_x, art_y)$ signifie qu'il existe un arc rel avec comme source le noeud art_x et comme destination le noeud art_y. Cette assertion est, en fait, la composition de deux assertions : $+source(rel, art_x)$ et $+dest(rel, art_y)$.

Dans la table 4.4, nous utilisons les assertions précédemment définies pour la spécification des opérations de base de modification de la structure des artefacts logi-

Table 4.2: *Liste des operation simples pour la manipulation du graphe de visualization de logiciel*

Notation	**Signification**
$+source(rel, art)$	L'arc rel doit avoir le nœud art comme nœud source
$+source(\Sigma_{rel}, art)$	Un arc $rel \in \Sigma_{rel}$ doit avoir le nœud art comme nœud source
$+dest(rel, art)$	L'arc rel doit avoir le nœud art comme nœud destination
$+dest(\Sigma_{rel}, art)$	Un arc $rel \in \Sigma_{rel}$ doit avoir le nœud art comme nœud destination
$+state(el, label)$	Existe un du nœud ou arc $el \in \Sigma_{art} \vee\ el \in \Sigma_{rel}$ avec l'état $label$
$+conductive(op, rel, art_x, art_y)$	Le relation rel est conductrice d'impact de art_x vers art_y après la réalisation de l'opération op
$+type(art, t)$	Existe un du nœud art ayant le type t
$-source(rel, art)$	L'arc rel ne doit pas avoir le nœud art comme nœud source
$-source(\Sigma_{rel}, art)$	Aucun arc rel ne doit pas avoir le nœud art comme nœud source
$-dest(rel, art)$	L'arc rel ne doit pas avoir le nœud art comme nœud destination
$-dest(\Sigma_{rel}, art)$	Aucun arc $rel \in \Sigma_{rel}$ ne doit pas avoir le nœud art comme nœud destination
$-state(el, label)$	N'existe pas un du nœud ou arc $el \in \Sigma_{art} \vee\ el \in \Sigma_{rel}$ avec l'état $label$
$-conductive(op, rel, art_x, art_y)$	Le relation rel n'est pas conductrice d'impact de art_x vers art_y après la réalisation de l'opération op
$-type(art, t)$	N'existe pas un du nœud art ayant le type t
$*source(rel)$	Donne la source de l'arc rel
$*dest(rel)$	Donne la destination de l'arc rel
$*state(el)$	Donne l'état du nœud ou arc $el \in \Sigma_{art} \vee\ el \in \Sigma_{rel}$
$*conductive(op, rel, art_x, art_y)$	Donne le réponse 'vrai' s'il existe une relation rel qui est conductrice d'impact de art_x vers art_y après la réalisation de l'opération op
$*type(art)$	Donne le type du nœud art

Table 4.3: *Liste des assertions composites pour la manipulation du graphe de visualization de logiciel*

Notation	Signification
$+arc(rel, art_x, art_y)$	L'arc rel doit avoir le nœud art_x comme nœud source et le nœud art_y comme destination
$+arc(\Sigma_{rel}, art_x, art_y)$	Un arc $rel \in \Sigma_{rel}$ doit avoir le nœud art_x comme nœud source et le nœud art_y comme destination
$+arc(\Sigma_{rel}, art)$	Un arc $rel \in \Sigma_{rel}$ doit avoir le nœud art comme nœud source ou destination
$-arc(rel, art_x, art_y)$	L'arc rel ne doit pas avoir le nœud art_x comme nœud source et le nœud art_y comme destination
$-arc(\Sigma_{rel}, art_x, art_y)$	Aucun arc $rel \in \Sigma_{rel}$ ne doit pas avoir le nœud art_x comme nœud source et le nœud art_y comme destination
$-arc(\Sigma_{rel}, art)$	Aucun arc $rel \in \Sigma_{rel}$ ne doit pas avoir le nœud art comme nœud source ou destination
$*arc(rel, art_x, art_y)$	Donne le réponse 'vrai' s'il existe un arc rel ayant le nœud art_x comme nœud source et le nœud art_y comme destination
$*arc(\Sigma_{rel}, art_x, art_y)$	Donne le réponse 'vrai' s'il existe un arc $rel \in \Sigma_{rel}$ ayant le nœud art_x comme nœud source et le nœud art_y comme destination
$*arc(\Sigma_{rel}, art)$	Donne le réponse 'vrai' s'il existe un arc $rel \in \Sigma_{rel}$ ayant le nœud art comme nœud source ou destination

ciels. La première colonne de la table représente l'opération les trois colonnes suivantes représentent respectivement : les pré-conditions, les post-conditions, et les invariants. Nous allons expliciter certaines de ces opérations dans ce qui suit. L'opération $addNode(art)$ permet d'ajouter un noeud art au graphe des artefacts logiciels. La pré-condition $-node(art)$ signifie que ce noeud ne doit pas exister dans le graphe un noeud ayant comme identifiant art. La post-condition montre le résultat de cette opération qui consiste en l'existence du noeud ainsi créé formalisée par l'assertion $+node(art)$. L'invariant $-arc(\Sigma_{rel}, art)$ contraint l'existence de tout arc ayant art comme source ou destination jusqu'à la complétion de l'exécution de l'opération d'ajout du noeud art.

L'opération $modifyNode(art)$ a comme pré-condition l'assertion $+node(art)$ signifiant qu'on ne peut modifier un noeud que s'il existe. L'invariant $+state(art, label)$ signifie que l'état de art doit rester le même avant et après la modification. La post-condition $+node(art')$ signifie le changement d'art telle que $art' = art + \Delta_{art}$.

Table 4.4: *Liste des opérations de base s'appliquant sur un graphe*

Opération	Pré-condition	Post-condition	Invariant
$addNode(art)$	$-node(art)$	$+node(art)$	$-arc(\Sigma_{rel}, art)$
$deleteNode(art)$	$+node(art)$	$-node(art)$	$-arc(\Sigma_{rel}, art)$
$modifyNode(art)$	$+node(art)$	$+node(art')$	$+state(art, label)$
$mergeNode(art_x, art_y, art_z)$	$+node(art_x)$ $+node(art_y)$	$+node(art_z)$ $-node(art_x)$ $-node(art_y)$	$-arc(\Sigma_{rel}, art_x)$ $-arc(\Sigma_{rel}, art_y)$ $-arc(\Sigma_{rel}, art_z)$
$splitNode(art_x, art_y, art_z)$	$+node(art_z)$ $-node(art_x)$ $-node(art_y)$	$-node(art_z)$ $+node(art_x)$ $+node(art_y)$	$-arc(\Sigma_{rel}, art_x)$ $-arc(\Sigma_{rel}, art_y)$ $-arc(\Sigma_{rel}, art_z)$
$addArc(rel, art_x, art_y)$	$-arc(rel)$	$+arc(rel)$ $+arc(rel, art_x, art_y)$	$+node(art_x)$ $+node(art_y)$
$deleteArc(rel, art_x, art_y)$	$+arc(rel)$ $+arc(rel, art_x, art_y)$	$-rel$ $-arc(rel, art_x, art_y)$	$+node(art_x)$ $+node(art_y)$
$changeState(art, label)$	$+node(art)$	$+state(art, label)$	$+node(art)$
$changeState(rel, label)$	$+arc(rel)$	$+state(rel, label)$	$+arc(rel)$

L'opération $addArc(rel, art_x, art_y)$ crée un arc entre deux artefacts. Dans cette opération rel représente l'identifiant de l'arc alors que art_x et art_y représentent les identifiants des deux artefacts. La précondition $-arc(rel)$ signifie que l'arc rel n'existe pas dans le graphe. Le résultat de cette opération est symbolisée par la post-condition contenant les deux assertions $+arc(rel)$ et $+arc(rel, art_x, art_y)$ signifiant l'existence d'un arc identifié par rel et reliant les deux noeuds art_x et art_y. L'invariant spécifié par les deux assertions $+node(art_x)$ et $+node(art_y)$ signifie que les deux noeuds art_x et art_y doivent exister avant l'opération de création de l'arc rel et continuent d'exister après.

Nous allons dans la section qui suit expliciter le processus générique de génération et propagation de l'impact des opérations de changement de la structure du logiciel.

4.5 Le processus de propagation de l'impact du changement

Notre approche de mise en oeuvre de la gestion de l'impact du changement du logiciel est basée sur le formalisme de Evénément/Condition/Action ou ECA. Ainsi,

les opérations de changement génèrent des événements spécifiés par des des faits qui influent sur les conditions ou assertions qui contrôlent l'exécution de ces opérations. Les actions se traduisent bien entendu par la génération de nouveaux faits ou la suppression de faits existants. Partant de cette approche, nous avons défini un ensemble extensible de règles pour la propagation de l'impact du changement et un processus générique de marquage des artefacts affectés par le changement qui repose principalement sur l'application de ces règles.

4.5.1 Règle de propagation de l'impact du changement

Ces règles spécifient l'exécution des opérations de changement associées à l'évolution des artefacts logiciels modélisés par le modèle SMSE. Elles manipulent des faits et peuvent ainsi ajouter ou supprimer des faits.

Considérons, par exemple l'opération de suppression d'un artefact *art* appartenant au code source d'une application. L'invariant spécifie que *art* ne doit être lié à aucun autre artefact. La post-condition spécifie le résultat de l'opération qui se manifeste par la suppression d'un fait. Le formalisme ECA, par le C de condition, impose la vérification de la pré-condition et de l'invariant avant l'exécution de l'opération. Ce même formalisme doit également assuré la vérification de la post-condiion après l'exécution de l'opération. La non vérification de ces conditions implique le fait que l'opération ne puisse pas s'exécuter.

En réalité le processus d'exécution d'une opération peut conduire à plusieurs scénarios que nous décrivons ci-dessous :

1. Si la pré-condition et l'invariant sont vérifiés alors l'opération peut être effectuée. L'invariant est vérifié après l'opération et s'il est vérifié cela veut dire que l'opération a pu s'exécuter sans générer d'impact à propager au reste du logiciel.
2. Si la pré-condition est non satisfaite, l'opération n'est pas exécuté et il n'y a pas d'impact à générer. Cela correspond aux opérations prohibées comme la tentative d'ajout d'un artefact déjà existant.
3. Si la pré-condition est vérifiée l'opération peut être exécutée mais la non vérification de l'invariant après l'exécution de l'opération se traduit par la génération d'un impact et le processus de propagation de l'impact peut ainsi être initié. Ce cas est le plus répondu et est bien sûr le plus significatif pour l'activité d'analyse

Table 4.5: *Listes des operations pour le marquage de graphe*

Operation	**Description**
$mark(el)$	$el \in \Sigma_{art} \vee \ el \in \Sigma_{rel}$: $changeState(el, \Delta_{el})$

$propagateImpact(op, art_x)$	$mark(art_x)$
	if $+arc(rel, art_x, art_y) \wedge +conductive(op, rel, art_x, art_y) \wedge -state(art_y, \Delta_{art_y})$ *then*
	$mark(rel)$
	$propagateImpact(op, art_y)$
	endif

et de propagation de l'impact du changement. C'est le cas notamment qui peut survenir lors de la suppression d'un artefact qui est reliés aux autres. En effet, dans ce cas la suppression de l'artefact en question générera un impact traduisant la non consistance de tous les arcs qui relie l'artefact supprimé avec les autres.

Dans la suite, nous montrerons comment un invariant d'une opération *op* de changement d'un artefact peut causer un impact à propager aux autres artefacts. Les assertions de composant ces invariants peuvent être divisées en deux catégories :

1. La première catégorie concernent les assertions vérifiant l'existence d'un élément dans la base de faits Ce sont les les assertions sont préfixés par « + ».
2. La seconde catégorie concernent les assertions qui vérifient la non-existence d'un élément. Ce sont celles préfixées par « - ».

Les invariants déclenchent l'opération de marquage des éléments (noeuds ou arcs) qui leurs sont associés. Ce marquage signifie que ces éléments sont affectés par le changement. Par exemple, soit l'invariant $-arc(compose, art_x, \Sigma_{art})$ correspondant à l'opération de suppression d'un artefact art_x. Ainsi la violation de cet invariant déclenchera le marquage de tous les artefacts composés par art_x ($compose \in \Sigma_{rel}$).

La table 4.5 spécifie de façon plus formelle les opérations de marquage des éléments et de propagation de l'impact. L'opération $marque(el)$ consiste en l'appel de l'opération $changeState(el, \Delta_{el})$ dont l'effet est de spécifier que el est « affecté » par le changement. En effet, un élément peut avoir deux état « affecté » ou « non affecté ». L'état par défaut étant « non affecté ».

L'opération $propagateImpact(op, art_x)$ consiste à marquer l'élément cible de l'opération (à savoir art_x) et à marquer toutes les relations ayant art_x comme participant et

qui sont conductrices de l'impact de l'opération *op*. Ces relations sont en effet définies dans les arcs ayant art_x comme sommet. Le marquage des relations est suivi de celui des artefacts reliés à art_x par ces relations et cela de façon récursive.

Le marquage de des éléments, à travers les relations, dépend donc de la nature de ces relations. Nous exprimons cela par l'assertion $+conductive(op, rel, art_x, art_y)$. En effet, une relation peut ne pas conduire l'impact d'une opération de modification *op* comme elle peut le conduire dans un sens (de art_x vers art_y) ou dans l'autre ou ne pas le conduire. La relation compose reliant un artefact composant art_x et un composé art_y est l'exemple d'une relation qui conduit l'impact de façon bidirectionnelle. En effet, la suppression d'un composant peut avoir des impacts sur le composé et inversement. C'est le cas par exemple de la suppression d'une relation de composition reliant un projet et un fichier source qui le compose, etc.

L'opération de marquage $mark(el)$ change donc l'état d'un artefact el dans la base de faits et cela dépend, bien entendu, de la nature de l'opération de modification et de la conductivité des relations ayant l'artefact *el* comme participant. Le processus de marquage se propage ainsi, en traversant les différentes relations. Au niveau d'un noeud ou d'un arc donné la poursuite ou non du marquage est dictée par des règles spécifiques qui prennent en compte les natures des relations et des opérations, et en appliquant à chaque fois . Le processus de propagation s'arrête définitivement lorsqu'il ne reste plus aucune règle à déclencher. Comme nous l'avons signalé dans la section 4.3, les opérations de changement structurel peuvent être appliquées sur différents types d'artefacts et de relations du modèle SMSE. Nous avons donc défini deux types de règles génériques pour l'implémentation des opérations en assurant une propagation de leur impact.

Le premier type de règles consiste en celles qui déclenchent les opérations *op* sur un élément *el* dans le cas où la pré-condition et l'invariant de l'opération sont vérifiés. Cela conduit juste à la génération des faits formant la post-condition. On ne parle, dans ce cas, pas d'impact.

Le second type de règles concernent l'exécution des opérations dont la pré-condition est satisfaite mais pas l'invariant. La règle marque donc l'élément el en modifiant son état et déclenche la propagation de l'impact.

4.5.2 Définition des règles de propagation de l'impact du changement

Les règles de propagation de l'impact du changement intègrent chacune un traitement spécifique de l'impact d'un changement donné. Le changement est matérialisé par un fait à intégré dans la base des faits. Cela constitue, généralement, le point de départ du processus de propagation de l'impact du changement. En effet, le fait nouvellement intégré dans la base peut rendre déclenchables un certain nombre de règles dont la fonction principale est de gérer l'impact. Ce sont principalement les règles associées aux opérations ayant le fait nouvellement créé dans leurs pré-conditions.

Listing 4.4: La règle d'impact du changement

```
ruleExecuteChangeOperation(op, Δ_art){
if op.pre − conditions ∧ op.invariants then
    op.post − conditions
    mark(art′)
endif
}
```

Ainsi nous définissons les deux règles génériques de gestion de l'impact.

La règle $ruleExecuteChangeOperation(op, \Delta_{art})$ (listing 4.4) effectue l'opération de changement op (sur un artefact représenté par un noeud art) quand la pré-condition et l'invariant de l'opération sont vérifiés. Cela se traduit par la génération des faits relatifs à la post-condition (symbolisant ainsi l'exécution de l'opération) et le marquage du seul artefact art' qui représente la nouvelle version de l'artefact de départ art.

La règle $ruleChangeImpactPropagation(op, \Delta_{art})$ (listing 4.5) est exécutée dans le cas ou l'invariant n'est pas vérifié (le symbole $\neg$ représente un logical NOT opérateur). Le résultat de l'exécution se traduit donc par une génération de la post-condition mais également de la propagation de l'impact du changement à travers art' qui représente la nouvelle version de l'artefact art de départ.

Listing 4.5: La règle de propagation d'impact du changement

```
ruleChangeImpactPropagation(op, Δ_art){
if op.pre − conditions ∧ ¬(op.invariants) then
    op.post − conditions
    propagateImpact(op, art′)
endif
}
```

Il est nécessaire de disposer d'un processus permettant d'identifier l'impact des différents changements pouvant affecter les artefacts logiciels et cela dans le but de rétablir la consistance du logiciel ainsi affecté. La propagation de l'impact du changement consiste

à exécuter ou simuler (dans le cas de l'analyse *a priori*) le changement initial puis identifier les incohérences ou inconsistances que ce changement initial peut induire et définir donc les éléments à corriger par de nouveaux changements correcteurs dérivés ou conséquents. La base de faits du système à base de connaissances est mise à jour au fur et à mesure de l'exécution des différentes opérations (initiales ou conséquentes). L'idée est d'insérer dans cette base les faits correspondant aux artefacts à modifier et leurs voisins affectés ou marqués par le changement.

Nous définissions un algorithme simulant ce processus (listing 4.6). L'algorithme que nous appelons *Change Propagation Process* consiste d'abord à appliquer une opération de changement *op* sur un nœud du graphe d'artefacts G. Lors de l'exécution de l'opération sur un artefact art, l'algorithme calcule ou reconstitue le logiciel ainsi modifié et spécifié par le graphe G' et cela en utilisant le morphisme (Δ_{art}) correspondant à l'application de l'opération de changement sur un artefact art. Cet algorithme repose sur les deux règles génériques définies plus haut et propose donc d'opérer le changement en dérivant le nouveau logiciel sans impact dans le cas ou les pré-conditions et les invariants sont vérifiés ou d'effectuer les changements et générer des impacts dans le cas ou seules les pré-conditions sont vérifiées (sans les invariants). L'algorithme évite les appels récursifs infinis par le fait qu'une règle ne peut être exécutée plus d'une fois pour le même ensembles de faits (au niveau de ses prémisses).

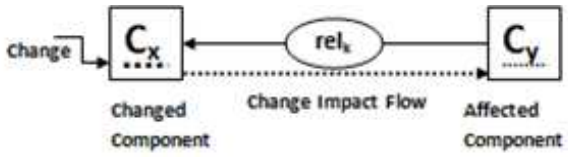
Listing 4.6: Algorithm : Le processus de la propagation d'impact du changement

```
1 Given a consistent program represented by G = (Σ_art, Σ_rel);
2 Select art from Σ_art;
3 art' = art + Δ_art; //if pre-conditions are satisfied then change operation is executed
4 ruleExecuteChangeOperation(op, Δ_art); //first rule is fired
5 ruleChangeImpactPropagation(op, Δ_art); //if invariants are not satisfied then the second rule is fired
6 G' = (Σ_art', Σ_rel');
```

4.6 Propagation de l'impact à travers les relations inter-artefacts

Il est nécessaire de déterminer le degré d'affectation d'un artefact donné par une opération ayant ciblé un autre artefact. Cela est rendu possible par une analyse détaillé des différents types de relations inter-artefacts. Il s'agit de disposer d'une connaissance sur ces relations incluant les cardinalités, la direction (navigabilité de la relation), et la notion de conductivité de l'impact. Dans la figure 4.6, nous définissions une relation *rel*

entre deux artefacts art_x et art_y avec comme sens de propagation ou de conduction de l'impact le chemin allant de art_x vers art_y et en sachant que ce chemin est le chemin inverse de la direction de base de la relation.



The relationship type rel_k implies the primary rule as,
if <C_x is changed> then <C_y is concerned> endif

FIGURE 4.6: Le type de la relation et la propagation d'impact du changement

Dans une première étape, nous dérivons une règle générale concluant en l'existence d'un impact et déterminant le flux de sa propagation. Cela peut, bien entendu, nécessiter l'intervention d'un expert humain qui renseignera avec plus de précisions les propriétés des différents types de relations. Les Le listing 4.7 décrit la règle générale de la propagation de l'impact à travers un type de relation rel reliant deux types d'artefacts art_x et art_y appartenant à deux phases éventuellement distinctes du cycle de vie du logiciel.

Listing 4.7: la règle de propagation d'impact du changement pour un type de relation rel

$ruleRel(op, \Delta_{art_x})\{$
  if $art_x \in \phi_i.\Sigma_{art} \wedge art_y \in \phi_j.\Sigma_{art} \wedge rel(art_x, art_y)$ **then**
    $mark(art_{x'})$
    $mark(rel(art_{x'}, art_{y'}))$
    $propagateImpact(op, art_{y'})$;
  endif
}

De la même manière, nous définissons des règles associées à tous les types de relations. Nous illustrons cela, dans ce qui suit, par quelques exemples.

4.6.1 Propagation de l'impact du changement à travers des relations inter-phases

Nous considérons deux artefacts art_x (une classe UML) et art_y (une classe Java) appartenant à deux phases distinctes du cycle de vie du logiciel (la conception ϕ_d et le codage ϕ_c). La relation $implements(art_x, art_y) \in \phi_d.\Sigma_{irf}$ implique que art_y est une implémentation de art_x et le relation $describes(C_x, C_y) \in \phi_c.\Sigma_{irb}$ signifie que art_x est une

description abstraite ou conception de art_y. La relation directe $implements(art_x, art_y)$ est conductrice de l'impact du changement pour toute modification Δ_{art_x} affectant art_x. La relation inverse $describes(art_x, art_y)$ est conductrice de l'impact de modification de art_y (Δ_{art_y}) vers art_x.

La règle de propagation de l'impact associée à la relation $describes(art_x, art_y)$ peut donc être spécifiée par le listing 4.8. De façon similaire le listing 4.9 spécifie la règle de propagation de l'impact du changement à travers la relation $implements(art_x, art_y)$.

Listing 4.8: La règle de propagation d'impact du changement pour une relation inter-phase (describes)

```
ruleDescribes(op, Δ_art_y){
    if art_x ∈ φ_d.Σ_art ∧ art_y ∈ φ_c.Σ_art ∧ describes(art_x, art_y) then
        mark(art_y')
        mark(describes(art_x', art_y'))
        propagateImpact(op, art_y')
    endif
}
```

Listing 4.9: La règle de propagation d'impact du changement pour une relation inter-phase (implements)

```
ruleImplements(op, Δ_art_x){
    if art_x ∈ φ_d.Σ_art ∧ art_y ∈ φ_c.Σ_art ∧ implements(art_x, art_y) then
        mark(art_x')
        mark(implements(art_x', art_y')
        propagateImpact(op, art_y')
    endif
}
```

4.6.2 Propagation de l'impact du changement à travers les relations inter-niveaux

Pour illustrer ce type de règles nous considérons trois artefacts art_x, art_y et art_z appartenant à la phase de codage. art_x et art_y sont deux modules représentés par les assertions $(+type(art_x, method) \wedge +type(art_y, method))$. art_z est un symbole individuel représenté par une variable $(+type(art_z, variable))$. art_z est visible au niveau de art_x et art_y. La relation verticale $modifies(art_y, art_z) \in \phi_c.\Sigma_{vr}$ implique que art_y a le droit de modifier art_z. La relation verticale $uses(art_x, art_z) \in \phi_c.\Sigma_{vr}$ signifie que art_x peut utiliser art_z. Toute modification Δ_{art_z} sur art_z par art_y peut affecter art_x. La règle de conductivité de l'impact du changement est alors donnée par le listing 4.10.

Listing 4.10: La règle de propagation d'impact du changement pour une relation inter-niveau

```
ruleUses(op, Δ_{art_z}){
    if  art_x ∈ φ_c ∧ +type(art_x, method)∧
        art_y ∈ φ_c ∧ +type(art_y, method)∧
        art_z ∈ φ_c ∧ +type(art_z, variable)∧
        modifies(art_y, art_z) ∧ uses(art_x, art_z) then
            mark(art_{z'})
            mark(uses(art_{x'}, art_{z'}));
            mark(modifies(art_{y'}, art_{z'}));
    endif
}
```

4.6.3 Propagation de l'impact du changement à travers des relations intra-niveaux

Considérons deux artefacts art_x et art_y qui sont deux modules appartenant à la phase de codage $(art_x, art_y \in < \phi_c, Lv_{method}, \Sigma_{method} >)$ et que art_y appelle art_x. La relation horizontale $calls(art_x, art_y) \in \phi_c.\Sigma_{hr}$ implique que toute modification Δ_{art_x} de art_x (le module appelé) affecte le module art_y (l'appelant) et peut donc conduire à une modification conséquente Δ_{art_y}) de art_y. Cependant, une modification de art_y n'implique pas d'impact sur art_x. La règle résumant cette situation est représentée par le listing 4.11.

Listing 4.11: La règle de propagation d'impact du changement pour une relation intra-niveau

```
ruleCall(op, Δ_{art_x}){
    if  art_x ∧ art_y ∧ calls(art_x, art_y) then
        mark(art_{x'})
        mark(calls(art_{x'}, art_{y'}))
        propagateImpact(op, art'_y)
    endif
}
```

4.7 Conclusion

Ce chapitre a présenté le système à la base de connaissances pour l'évolution de logiciels et le développement de ses principaux constituants base des faits et base des règles. La typologie des opérations de changement structurel a été brièvement abordée. Nous avons ensuite présenté la modélisation des opérations de changement structurel. Le processus de la propagation de l'impact du changement a été expliqué en détail. Il

donne les détails méthodologiques pour établir des règles de la propagation d'impact du changement.

Il formule les relations responsables de la conduction d'impact du changement structurel et architecturale, et identifie les éléments intervenants dans l'importance de l'impact du changement. Nous avons ensuite formulées dans ce chapitre les règles de la propagation d'impact du changement pour différents types de relations.

L'objectif principal étant de déterminer la conduction de l'impact du changement depuis un artefact modifié vers d'autres artefacts affectés par la modification. La plateforme développée selon la modélisation proposée est conçue pour permettre, à la demande de l'utilisateur, de générer des vues spécifiques par des graphes interactifs. Ces graphes permettent de visualiser les éléments du logiciel qui peuvent être directement, ou indirectement, affectés par l'application du changement à un constituant du logiciel.

Troisième partie

Modélisation Qualitative

Chapitre 5

Modélisation Qualitative Intégrée pour le Contrôle de l'Evolution du Logiciel

5.1 Introduction

Nous présentons, dans ce chapitre, une approche de traçabilité des flux fiables d'impacts des changements durant le processus d'évolution du logiciel. Cette approche est implémentée par un framework intégrant des outils d'extraction d'information concernant le logiciel et d'autres outils mettant en œuvre des mécanismes semi-automatiques d'analyse de l'impact du changement aussi bien au niveau structurel que qualitatif. Ceci permet, en effet, de faciliter le mécanisme de prise de décision ou ce qui est communément appelé conduite du changement durant le processus d'évolution ou de maintenance du logiciel.

Contrôler l'évolution du logiciel en prenant en compte les contraintes en matière de délais et de ressources et sans dégradation de la qualité alors que les changements transforment le logiciel en un système plus large et plus complexe est un véritable challenge. En effet, les changements de la structure du logiciel peuvent avoir des effets sur la qualité originelle du logiciel. On observe en effet, une tendance à une dégradation de la qualité du logiciel proportionnelle à l'accroissement de sa taille et de sa complexité qui est une conséquence des additions de nouvelles fonctionnalités se traduisant, généralement, par une modification de sa structure. La structure du logiciel change donc avec le

développement de ces fonctionnalités. Cela peut conduire à des effets indésirables qu'on appelle aussi des effets de bord pouvant transformer l'architecture du logiciel en une sorte de plat de spaghetti.

De nombreux travaux, en génie logiciel, ont été consacrés, depuis de nombreuses années, à l'évaluation de la qualité du logiciel et ont aboutit à la définition d'un nombre important de métriques mesurant différents aspects de cette qualité. De nombreux frameworks théoriques et numériques ont été proposés pour l'évaluation de la qualité. Les modèles de la qualité du logiciel proposés par B. Bohem (101) et Mc. Call (102, 103) sont connus pour être les modèles historiques de représentation, conception et évaluation de la qualité. Plusieurs variations du modèle dit de Boehm-McCall ont été mises en œuvre et cela dès sa proposition (104, 105, 106, 107, 108, 109, 110, 111). Ces variations concernent l'analyse des facteurs de la qualité du logiciel et leur mesure ou « métrification » quantitative et qualitative. Par la suite, des études empiriques (112, 113, 114, 115, 116, 117, 118, 119) ont étudié l'implication de nombreux facteurs dans l'évaluation de la qualité du logiciel et l'interdépendance de ces facteurs. La modélisation des artefacts et de leurs nombreuses interactions peut aider à l'analyse des implications qualitatives du changement du logiciel (120, 121).

Dans ce chapitre, nous nous intéressons particulièrement à l'impact du changement de la structure des artefacts logiciels sur la qualité ou ce que nous appelons impact qualitatif du changement. Le nombre de modifications d'un artefact peut être un indicateur important du taux de fautes ou défaillances de cette artefact, de sa stabilité et de sa complexité. Nous discuterons l'analyse de la propagation des impacts structurels et qualitatifs des changements. Notre objectif étant de proposer une approche systématique de mise en œuvre de changements « reussis ». En d'autres termes minimiser les risques de dégradations de la qualité engendrés par les changements.

Ce chapitre est organisé comme suit : la section 5.2 est un aperçu des notions d'assurance qualité des systèmes logiciels. La section 5.3 explore la notion d'évaluation de la qualité avec l'aide du modèle qualitatif de l'évolution du logiciel. La section 5.4 explicite l'élaboration de l'approche que nous proposons à l'aide d'un exemple. La section 5.5 décrit le mapping ou la mise en correspondance du modèle qualitatif et du modèle SMSE. Dans la section 5.6 nous concluons le chapitre en soulignant les perspectives de notre travail.

5.2 L'Assurance Qualité du Logiciel

L'assurance qualité du logiciel ou SQA pour *Software Quality Assurance* a toujours été une tâche compliquée et cela est, en grande partie du, à la difficulté d'évaluer la qualité générale du logiciel en l'absence d'information concernant la traçabilité. Une période assez longue de « compétition » entre différents modèles et approches dédiés à l'assurance qualité du logiciel a conduit à des travaux combinant les résultats de ces différentes approches. Le modèle de Boehm-McCall (101, 103) et ses nombreuses variantes ont été proposés pour concevoir la qualité générale du logiciel comme une combinaison de plusieurs facteurs, raffinés par des critères ou attributs internes de la qualité et qui sont évalués par des métriques.

Un facteur représente un attribut plus large de la qualité alors qu'un critère représente un attribut interne d'un artefact logiciel. Un facteur est généralement extérieurement appréciable par l'utilisateur. Il représente généralement un des besoins non fonctionnels exprimés par l'utilisateur tels que l'utilisabilité ou la fiabilité, etc. Le critère représente des considérations plus détaillées ou techniques. Il n'est généralement pas appréciable ou non visible par l'utilisateur du logiciel. Il exprime des perspectives de qualités concernant le développeur et l'ingénieur en assurance qualité. Parmi les critères, on notera la modularité, etc. Si l'on considère un modèle de qualité comme où les nœuds sont les facteurs (F) raffinés par des critères (C) mesurés par des métriques (M) (102), ces dernières représenteraient donc les feuilles de l'arbre. Elles servent à mesurer ou estimer les différents critères. C'est le cas par exemple du nombre de lignes de codes, ou du nombre de commentaires, etc.

Nous tenons en compte les dépendances entre Facteurs et Critères pour analyser les variations de la qualité dans le cadre d'une analyse qualitative des changements dans les différentes phases du cycle de développement du logiciel. Cela peut aider l'expert de tracer l'impact sur la qualité et cela dans le sens d'une amélioration ou détérioration d'un facteur de cette qualité. Dans les sections qui suivent nous discuterons, avec plus de détails, les améliorations apportées au modèles traditionnels de la qualité.

5.2.1 Représentation synthétique de la qualité

Il est largement admis qu'à travers le cycle de développement du logiciel, chaque phase a ses pré-requis et ses objectifs. Pour atteindre ces objectifs, des méthodes et

techniques spécifiques peuvent être adoptées. Dans les modèles traditionnels de la qualité du logiciel, les artefacts spécifiques à chaque phase et les attributs de leurs qualités sont présentés de façon synthétique. Tous les attributs de la qualité sont évalués de façon globale dans la perspective d'une évaluation de la qualité de la totalité du logiciel. Ceci aurait été suffisant si l'objectif de l'évaluation de la qualité était de fournir un seul indicateur sur la qualité globale du logiciel. Or, dans la pratique les ingénieurs qualité sont généralement intéressés par une évaluation de la qualité d'artefacts particuliers les conduisant ainsi à privilégier une analyse par phases. De plus, les approches d'évaluation globales ne sont pas adaptées aux approches nouvelles de développement comme celles dirigées par les modèles, etc. Ces approches ne sont également pas adaptés pour une prise en compte des changements et de leurs impacts sur l'évaluation de la qualité.

Une représentation simplifiée des attributs de la qualité associées aux artefacts logiciels structurellement organisés en niveaux abstro-granulaires peut aider à minimiser les difficultés d'évaluation de la qualité en présence de changements du logiciel. Un effet de vague ou ripple effect d'un changement appliqué à un artefact logiciel peut concerner plusieurs autres artefacts du système logiciel (9). Ce changement peut, également, affecter les métriques de la qualité concernant l'artefact ainsi modifié. L'interdépendance des attributs de la qualité peut alors conduire à une mauvaise situation dans laquelle la détérioration d'une métrique de la qualité peut avoir un impact défavorable sur d'autres métriques. Une connaissance exhaustive de la qualité du logiciel peut aider à mieux tracer la propagation de l'impact du changement de façon progressive entre métriques, critères et facteurs de la qualité du logiciel.

5.2.2 Hétérogénéité des Facteurs et Critères

Cet aspect concerne la représentation exhaustive des attributs de la qualité du logiciel. Dans le graphe de la qualité du modèle de de Boehm-McCall (101, 103), les éléments hétérogènes de la qualité du logiciel sont représentés de façon uniforme. Ceci est du probablement à une recherche de simplicité dans la définition du modèle. L'uniformité recherchée s'est traduite, au niveau du modèle, par un nombre de conséquences dont le fait de fixer à la l'avance les facteurs et les critères.

Tous les facteurs et critères, n'ont généralement pas la même complexité ni la même granularité. Les artefacts associés peuvent également varier en nombre, en complexité, en le fait qu'ils soient directement mesurables ou pas, etc. Par exemple, les caractéristiques

nécessaires à la mesure de la maintenabilité d'un logiciel sont connus pour être plus complexe que celles relatives à la portabilité. En conséquence, le nombre d'attributs qualitatifs relatifs à la maintenabilité est plus important que celui associé à la portabilité.

Dans les modèles traditionnels de la qualité, l'hétérogénéité intrinsèque est considérée de façon globale. Elle se traduit par un nombre plus ou moins important des descendants d'un facteurs ou d'un critère mais n'affecte pas la profondeur de l'arbre de la qualité. Il est donc nécessaire de définir, *a priori*, le raffinement des facteurs en critères et des critères en métriques avant de procéder à une mesure quantitative de la qualité dont le point de départ sont les évaluations des métriques. Ceci se traduit par une complexité et une difficulté du processus de raffinement de chaque attribut de la qualité. Il serait plutôt judicieux de procéder à un raffinement de critères en sous-critères, rendant ainsi la profondeur de l'arbre variable, mais simplifiant le processus de raffinement.

5.2.3 Les dépendances horizontales entre les attributs de la qualité

Dans les modèles traditionnels de la qualité, seules les relations de dépendance verticales sont considérées. (116) a identifié des relations (inverse, directe et neutre) entre onze critères du modèle FCM (102). Les dépendances horizontales entre facteurs, critères et sous-critères doivent donc être considérées. Durant l'évolution d'un facteur de la qualité, les autres facteurs qui lui sont reliés peuvent également évoluer. Cela correspond à l'existence ou l'absence de la propagation de l'impact entre les nœuds du graphe de la qualité. Par exemple, il est largement admis que deux critères tels que la traçabilité et l'efficacité évoluent de manière totalement indépendante mais l'efficacité est généralement en conflit avec l'utilisation de la mémoire. Il est également connu que l'amélioration d'un critère comme la modularité est favorable à la testabilité, etc. Il est à noter que les modèles traditionnels ne fournissent pas de projection des relations horizontales pour l'évaluation de l'impact de certaines améliorations ou dégradations d'un facteur, critère ou sous-critère sur les autres éléments ou nœuds du graphe de la qualité.

Nous présentons dans la section suivante une re-formalisation incrémentale des modèles traditionnels de la qualité permettant de combler leurs lacunes e matière de prise en compte de l'impact du changement.

5.3 Modèle qualitatif pour l'évolution du logiciel

Dans son HDR, H. Basson a présenté un modèle de la qualité du logiciel appelé Modèle Qualitatif des Composants Logiciels (MQCL) (66). Le modèle Qualitatif pour l'Evolution du Logiciel (QMSE :*Qualitative Model for Software Evolution*) intègre de façon plus opérationnelle les concepts déjà présents dans MQCL. Nous focalisons principalement sur la notion de relations horizontales entre attributs de la qualité. En effet, ces dernières modélisent la complémentarité ou la contradiction de deux attributs dans l'évaluation de la qualité. Il est ainsi possible de spécifier des relations de types : l'amélioration de la qualité d'un attribut $A1$ participe à améliorer (respectivement détériorer) celle d'un autre attribut $A2$, etc. Nous avons également traité, de façon particulière, la propagation de l'impact des changements structurelles sur la qualité. Cela est rendu possible par la définition de correspondances ou mappings entre les attributs de la qualité et les éléments du modèle structurel. Il s'agit de définir une sorte de mécanisme permettant une traçabilité des opérations de changement sur plusieurs points de vue du logiciel.

QMSE est destiné à servir d'outil de modélisation dans le cadre d'une assistance semi-automatique dans l'analyse de l'impact du changement sur la qualité du logiciel. Ce modèle est couplé avec le modèle structurel SMSE (7, 8) décrit dans le chapitre 2. Le modèle structurel SMSE sert de pré-requis pour l'évaluation de la qualité à la suite du changement de la structure du logiciel et cela par le fait qu'il permette une traçabilité des changements à travers les liens d'inter-dépendance qu'il définit entre les différents artefacts logiciels. Le modèle qualitatif, QMSE sert de moyen d'analyse des répercussions causales de modification d'un attribut de la qualité sur les autres attributs de la qualité.

5.3.1 Le sous-graphe de la qualité d'une phase

Les facteurs et critères de la qualité sont généralement des attributs associés à une phase individuelle du cycle de vie du logiciel. Les métriques destinées à leur évaluation sont donc associées à un sous-graphe particulier de la qualité.

Cette représentation permet à la fois d'adopter des formalismes spécifiques pour chaque phases et une évaluation e la qualité par phase ce qui est plutôt une nécessité dans des projets de grandes tailles.

Table 5.1: *Racines des sous-graphes de la qualité par phase du cycle de vie*

Racine du sous graphe de la qualité	Phase concernée du cycle de vie
$Qi(\Phi_0.\Sigma_{art})$	Étude de faisabilité
$Qi(\Phi_1.\Sigma_{art})$	Définition des besoins
$Qi(\Phi_2.\Sigma_{art})$	Spécification fonctionnelle
$Qi(\Phi_3.\Sigma_{art})$	Conception préliminaire
$Qi(\Phi_4.\Sigma_{art})$	Conception détaillée
$Qi(\Phi_5.\Sigma_{art})$	Codage
$Qi(\Phi_6.\Sigma_{art})$	Tests individuels des artefacts
$Qi(\Phi_7.\Sigma_{art})$	Intégration des artefacts
$Qi(\Phi_8.\Sigma_{art})$	Test global
$Qi(\Phi_9.\Sigma_{art})$	Évolution post-développement (Maintenance)
$Qs(\Sigma_\Phi.\Sigma_{art})$	Sous graphe qualitatif de synthèse

Le modèle QMSE considère le graphe de la qualité comme un graphe reliant des nœuds ou artefacts de différents niveaux. La racine du modèle ou graphe de la qualité, dénotée par $QG(\Sigma_\Phi.Art)$, représente la qualité générale des artefacts logiciels. Pour une phase j la qualité est dénoté par $Qi(\Phi_j)$ et le nombre de phases du modèle de développement est dénoté $|\Sigma_\phi|$. Le nœud $(QG(\Sigma_\Phi.\Sigma_{art}))$ est alors raffiné telle que l'ensemble de ses descendants (Dsc) :

$$Dsc(QG(\Sigma_\Phi.\Sigma_{art})) = \{Qi(\Phi_1.\Sigma_{art}), Qi(\Phi_2.\Sigma_{art}), \ldots, Qi(\Phi_{|\Sigma_\Phi|}.\Sigma_{art}\}$$

De la même manière, les descendants contiennent une racine d'un sous-graphe de la qualité associée à une phase. Nous considérons les phases communément usitées dans la majorité des modèles de développement du logiciel (68) (Cascade, V, Spiral, Incrémental, Evolutionnaire, Transformationnel, Agiles, Prototypage rapide, etc.). La table 5.1 relie chaque descendant de $QG(\Sigma_\Phi.\Sigma_{art})$ à la phase correspondante.

Le sous-graphe synthétique de la qualité $(Qs(\Sigma_\Phi.\Sigma_{art}))$ est telle que chaque nœud représente la vue qualitative d'un ensemble de phases spécifiques à l'expert qualité. Cette vue est basée sur l'occurrence d'attributs qualitatifs à travers les artefacts logiciels de phases particulières. Cette vue se réfère à des facteurs, critères et sous-critères de sous-graphes de la qualité associés à différentes phases. Les descendants du nœud dénoté

$Qs(\Sigma_\Phi.\Sigma_{art})$ peuvent se référer à un nœud $\in Qi(\Phi_j), i = 1 \ldots |\Sigma_\Phi|$ ou représenter un autre nœud $\in Qs(\Sigma_\Phi.\Sigma_{art})$ (Figure 5.1).

5.3.2 Graphe de profondeur variable

La qualité globale du logiciel est généralement évaluée comme le résultat d'une évaluation sur trois niveaux (facteurs, critères, et métriques). Les facteurs représentent des attributs de la qualité du point de vue externe. En d'autres termes ce sont des attributs extérieurement appréciables alors que les critères sont des attributs internes concernant principalement les développeurs ou les chargés de la qualité, etc. Les métriques étant des outils de mesure et/ou d'estimation de la qualité directement applicables associés aux critères. La restriction du raffinement du niveau facteurs au niveau critères seulement avec comme outils de mesure les métriques associés critères semble insuffisant au regard de la complexité de certains facteurs et critères qui nécessite un peu plus de raffinement pour une explicitation plus précise. Nous avons notamment discuter cet aspect dans (120, 121).

Donc pour une évaluation plus précise et plus détaillée de la qualité, il est nécessaire de disposer d'un nombre suffisant de raffinements qui varient en fonction du facteur ou du critère à raffiner. Soit donc $G = (X, U)$ le graphe FCM (Facteur Critère Métrique) de la qualité établi pour une application donnée. L'ensemble X des nœuds du graphe est l'union des sous-ensemble de nœuds représentant les divers éléments du graphe de la qualité telle que : $X = X_0 \bigcup \Sigma_F \bigcup \Sigma_C \bigcup \Sigma_M$

- X_0 est la racine du graphe. Il représente l'élément le plus abstrait de la qualité qu'on appellera la qualité générale ou globale du logiciel. Il correspond au niveau 0 du raffinement.
- $\Sigma_F = \{f_1, f_2, \ldots, f_m\}$ est l'ensemble des facteurs constituant le premier raffinement de la qualité générale. C'est le niveau 1 du raffinement.
- $\Sigma_C = \{c_1, c_2, \ldots, c_p\}$ représente l'ensemble des critères raffinant les facteurs. C'est le niveau 2 du raffinement.
- $\Sigma_M = \{m_1, m_2, \ldots, m_n\}$ représente l'ensemble des métriques permettant une évaluation quantitative et/ou qualitative des critères.

Nous adoptons une variabilité de la profondeur du graphe selon laquelle le nombre de niveaux de raffinement (de facteurs ou de critères) varie en fonction de la complexité du facteur ou du critère correspondant. Le raffinement d'un critère en sous-critères et de

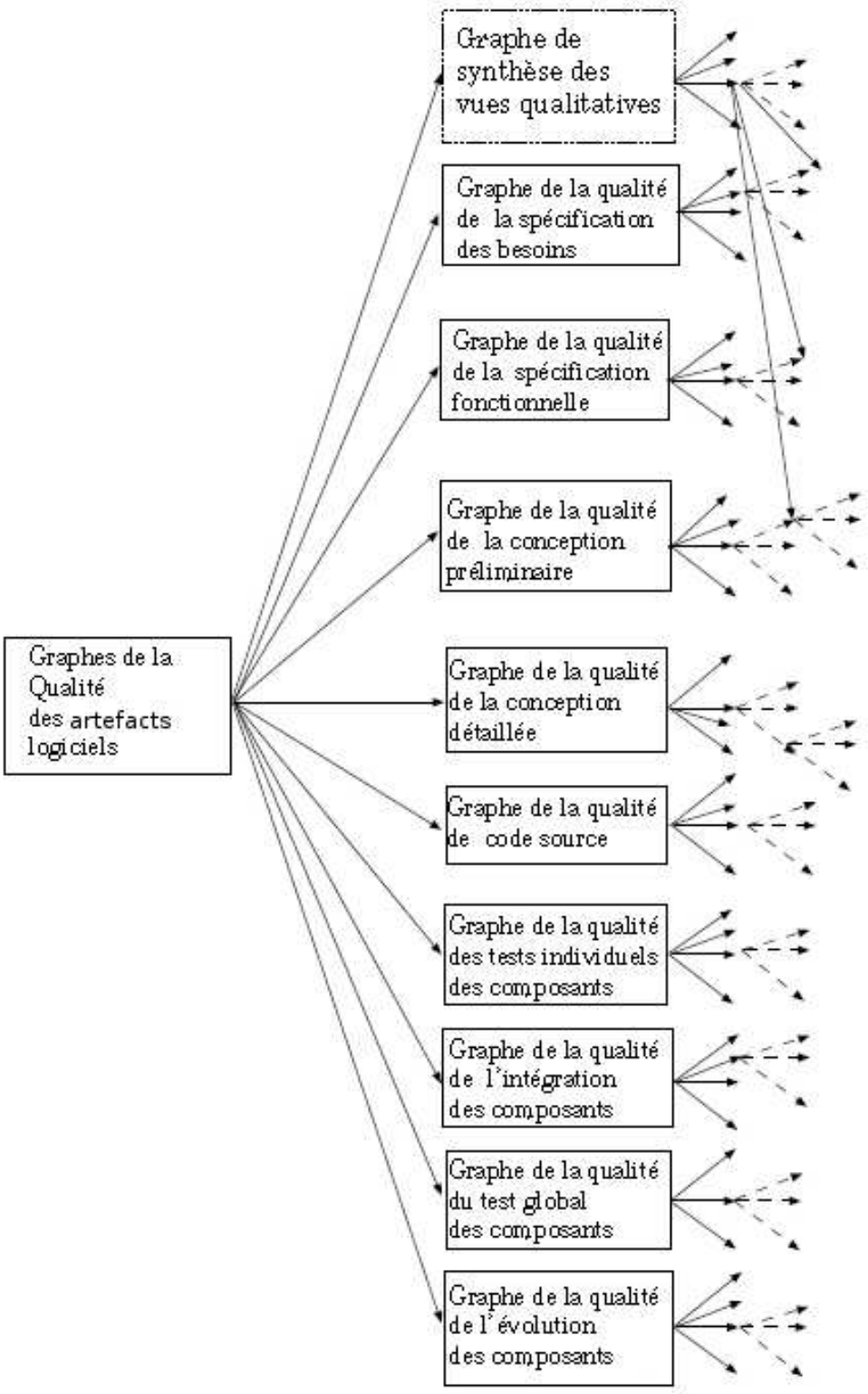


FIGURE 5.1: Graphe de la qualité des artefacts des phases individuelles

façon récursive en des sous-critères encore plus détaillés est une activité effectuée par des experts en modélisation de la qualité. L'ingénieur qualité conduit ce processus de façon récursive jusqu'à aboutir à des attributs précis pouvant être évalués par une ou plusieurs métriques. Nous avons également défini la propagation de l'impact du changement entre facteurs, critères et sous-critères du même niveau de raffinement. La représentation des relations horizontales permet en effet de considérer les contradictions existant entre différent attributs de la qualité. Cela permet alors de définir une construction évolutive de la qualité.

Il est à noter qu'il n'y a pas de différence conceptuelle entre les critères et les sous-critères. Les deux représentent des attributs techniques ou internes reliés entre eux par la relation de composition. Cela signifie que tous les sous-critères aident à une définition plus précise ou plus détaillée de leurs ascendants.

De façon pratique, dans toutes les applications logicielles les niveaux associés aux facteurs et aux critères sont identiques. Ils décrivent des propriétés générales contribuant chacune dans une certaine mesure à l'assurance de la qualité globale. Le raffinement d'un critère en sous-critères permet le passage d'une généralité partagée par une famille de langages, formalismes, et méthodes à la particularité de chaque artefact individuel exprimée selon son utilisation spécifique dans une application. Le raffinement reflète la connaissance d'un expert qualité concernant la définition de relations entre critères et sous-critères. Il reflète aussi l'importance qu'on accorde à chaque facteur qui sera plus prioritaires que d'autres (certaines facteurs étant raffinés de façon plus détaillée et plus précise que d'autres). Cette expertise est alors utilisée pour spécifier la manière de détailler l'expression de la qualité. Cela conduit donc à fournir une expression générale de objectifs qualitatifs et l'expression des sous-objectifs définissant les conditions de les atteindre.

Entre les critères et les métriques, plusieurs niveaux Si de sous-critères peuvent exister (Figure 5.2). L'ensemble X des nœuds du graphe de la qualité peut donc être défini par :

$$X = X_0 \bigcup \Sigma_F \bigcup \Sigma_{SC} \bigcup \Sigma_M$$

où :

$$\Sigma_{SC} = \{SC_1, SC_2, \ldots, SC_z\}$$

représente l'ensemble des différents niveaux de sous-critères, et

$$\Sigma_{SC_i} = \{SC_{i,1}, SC_{i,2}, \ldots, SC_{i,n}\}$$

représente l'ensemble des sous-critères du i^{me} raffinement.

Généralement, un nœud du graphe $Nd_i \in X$ obtenu après h nombres de raffinements est dit être de niveau (ou de profondeur) h et peut être dénoté par$Nd_i(h)$. La profondeur d'un facteur est définie par le nombre de raffinements.

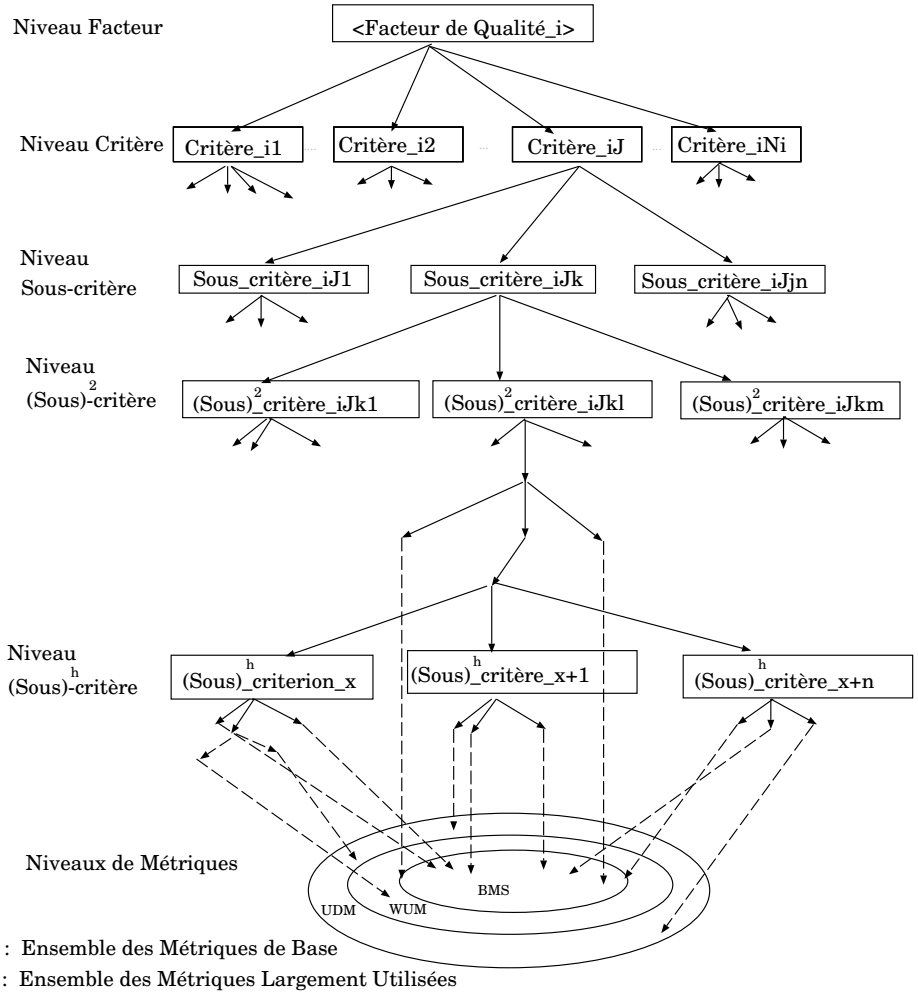


Figure 5.2: Représentation arborescente des critères, sous-critères et métriques.

5.3.3 Notations et définitions

Considérons un artefact logiciel $Nd_i(h)$ comme un nœud du graphe de la qualité obtenu après h nombres de raffinements. Le nœud $Nd_i(h)$ est évalué numériquement par une fonction Val définie par :

$$Val(Nd_i(h)) = \Theta(y_{i1}\ , y_{i2},\ \ldots,\ y_{ij})$$

où y_{ij} représente les paramètres dont les valeurs effectives sont obtenues par analyse statique suivie d'un calcul des valeurs des métriques. L'impact de la variation de la qualité sur l'artefact logiciel $Nd_i(h)$ peut être comme suit :

1. Un nœud $Nd_i(h)$, est dit évoluer avec un impact favorable par rapport à un autre nœud $Nd_j(h)$, si et seulement si toutes les variations positives de $Val(Nd_i(h))$, impliquent des variations positives ou nulles de $Val(Nd_j(h))$.
2. Un nœud $Nd_i(h)$ est dit évoluer avec un impact défavorable par rapport à un autre nœud $Nd_j(h)$, si et seulement si toutes les variations positives de $Val(Nd_i(h))$, impliquent des variations négatives ou nulles de $Val(Nd_j(h))$.
3. Un nœud $Nd_i(h)$ est dit évoluer de façon neutre par rapport à un autre nœud $Nd_j(h)$, si et seulement si toutes les variations positives de $Val(Nd_i(h))$ n'impliquent pas de variations de $Val(Nd_j(h))$.
4. Deux nœuds $Nd_i(h)$ et $Nd_j(h)$ évoluent mutuellement avec un impact favorable (respectivement neutre ou défavorable) si et seulement si, $Nd_i(h)$ évoluent avec un impact favorable (respectivement neutre ou défavorable) par rapport à $Nd_j(h)$ et réciproquement.

5.3.4 Fonctions associées

Ces fonctions, applicables aux éléments du graphe de la qualité sont destinées à manipuler et calculer les chemins de propagation de l'impact qualitatif à travers les nœuds :

1. La fonction Γ_v

   Cette fonction, appliquée à un nœud, retourne les descendants, les raffinements ou les mesures de ce nœud. Appliqué au nœud racine X_0, $\Gamma_v(X_0) = \Sigma_F$, retourne l'ensemble des facteurs associés à la qualité générale représentée par le nœud racine X_0.

Appliquée à un facteur $F_i \in \Sigma_F$, Γ_v retourne les sous-ensemble de critères $C_a \subset \Sigma_C$ permettant d'implémenter F_i :

$$\Gamma_v(F_i) = SC_a \subset \Sigma_C(\forall F_i,\ i = 1 \dots m)$$

Tous les critères $C_i \in \Sigma_C$ sont raffinés par un ensemble de sous-critères $SC_a \subset C_i$: $(\forall C_i, \in \Sigma_C)$, $(\Gamma_v(C_i))$ telle que :

$$\Gamma_v(C_i) = SC_a \subset C_i$$

De la même manière, un sous-critère $SC_i \in C_i$, peut être raffiné par un autre sous-critère SC_{ij}, ou il peut être directement évalué par un sous-ensemble de métriques $M_{ia} \subset \Sigma_M$. En conséquence, l'application de $\Gamma_v(SC_i)$ est telle que :

- $\Gamma_v(SC_i) = SC_{ij} \subset SC_i$, si SC_i est raffiné par un autre sous-critère,
- $\Gamma_v(SC_i) = M_{ia} \subset \Sigma_M$, si SC_i peut être directement évalué par des métriques.

2. La fonction Γ_{h+}

   Pour tous les nœuds $Nd_i(h)$ du graphe de profondeur de raffinement h, $(\exists \Gamma_{h+}(Nd_i))$ telle que :

   $$\Gamma_{h+}(Nd_i) = \{Nd_{i1},\ Nd_{i2},\ \dots,\ Nd_{i_{n+}}\}(\forall Nd_i, i = 1 \dots m)$$

   Cet ensemble correspond à tous les nœuds du même niveau, dont au moins une évolution positive de Nd_i leur est favorable, sinon elle leur est neutre.

3. La fonction Γ_{h-}

   Pour tous les nœuds $Nd_i(h)$ du graphe de profondeur de raffinement h, $(\exists \Gamma_{h-}(Nd_i))$ telle que :

   $$\Gamma_{h-}(Nd_i) = \{Nd_{i1},\ Nd_{i2},\ \dots,\ Nd_{i_{n-}}\}(\forall Nd_i, i = 1 \dots m)$$

   Cet ensemble correspond nœuds du même niveau, dont au moins une évolution positive de Nd_i leur est défavorable, sinon elle leur est neutre.

4. La fonction Γ_{h*}

   Pour tous les nœuds $Nd_i(h)$ du graphe de profondeur de raffinement h, $(\exists \Gamma_{h*}(Nd_i))$ telle que :

   $$\Gamma_{h*}(Nd_i) = \{Nd_{i1},\ Nd_{i2},\ \dots,\ Nd_{i_{n*}}\}(\forall Nd_i, i = 1 \dots m)$$

   Cela correspond à tous les nœuds du même niveau, dont au moins une évolution, de sens positif ou négatif, de Nd_i leur est favorable, et dont au moins une autre évolution de Nd_i et dans le même sens leur est défavorable.

5.3.5 Dépendances entre les attributs de la qualité

Nous définissons dans cette section, les relations responsables de la propagation de l'impact du changement entre les facteurs, critères et sous-critères de la qualité d'un même niveau de raffinement. La présentation de ces relations horizontales permet de considérer les contradictions qui peuvent exister entre les différents facteurs de la qualité du logiciel. Elle permet également de définir un processus de construction évolutif ou incrémental du modèle la qualité du logiciel. Nous introduisons d'abord quelques notations permettant d'expérimenter les relations horizontales et les fonctions de manipulation du graphe de la qualité présentées dans la figure 5.3.

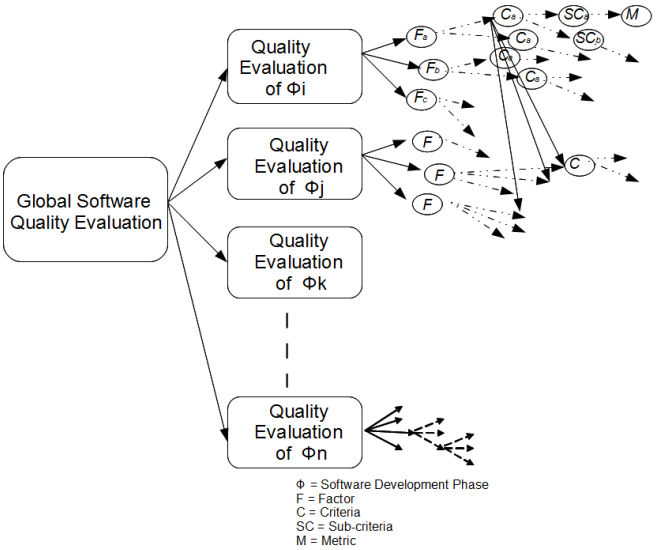


FIGURE 5.3: Les dépendances horizontales dans l'évaluation de la qualité des artefacts logiciels

Dans la suite nous considérons dans l'ensemble U des arcs du graphe de la qualité deux sous-ensembles disjoints : l'ensemble V représentant les relations verticales entre facteurs, critères et métriques ; et l'ensemble H des arcs reliant des nœuds d'un même niveau. L'orientation de l'arc indique la nature de la propagation de l'impact qualitatif qui peut être libellé par les symboles $+$, $-$, ou $*$ indiquant respectivement un impact positif, négatif ou un changeant.

L'ensemble des arcs est alors défini par $U = V \bigcup H$, où H est l'ensemble des arcs horizontaux. H représente les relations de dépendance entre des nœuds d'un même niveau, telle que :

$$H = H^+ \bigcup H^- \bigcup H^*$$

Où, H^+, H^-, et H^* désignent respectivement les arcs libellés par le symbole « + », « − », et « ∗ ».

Donc, $U = V \bigcup H^+ \bigcup H^- \bigcup H^*$ telle que :

- $(Nd_i(h), Nd_j(h)) \in H^+) \iff Nd_i(h)$ présente une évolution favorable pour $Nd_j(h)$;
- $(Nd_i(h), Nd_j(h)) \in H^-) \iff Nd_i(h)$ présente une évolution défavorable pour $Nd_j(h)$;
- $(Nd_i(h), Nd_j(h)) \in H^*) \iff Nd_i(h)$ présente une évolution changeante de $Nd_j(h)$.

5.3.6 Les coefficients de pondération des attributs de la qualité

Un attribut de la qualité peut être considéré comme plus ou moins critique par rapport à une application logicielle donnée. En dépit d'une importance minimale que tout attribut doit revêtir eu égard au domaine d'application, un facteur peut être considérée de première importance dans une application donnée et d'une importance moindre dans d'autres applications. L'importance d'un facteur peut donc varier en fonction de la hiérarchie des priorités que l'on donne aux facteurs lors de la définition ou de la redéfinition des besoins non fonctionnels. Ainsi, la priorité peut dans certains cas être maximale par rapport à des facteurs relevant de la fiabilité alors que dans d'autres cas c'est plutôt l'évolutivité qui l'emporte en termes de priorité. Il est possible ainsi de définir une hiérarchie des risques sur lesquelles reposera la conduite du projet de développement ou d'évolution du logiciel et de dériver les priorités en termes d'attributs qualitatifs à partir de cette hiérarchie. L'évaluation de la qualité d'un logiciel donné reflète donc, en quelques sortes, la hiérarchie des priorité et des risques exprimées au niveau de l'analyse des besoins non fonctionnels. L'importance d'un facteur de la qualité F_i donné peut être prise en compte par le biais d'un coefficient de pondération $\alpha_i \in (0 \ldots 1)$, telle que $\Sigma_{i=1}^{|F|} \alpha_i = 1$.

Si $Qt(F_i)$ désigne l'estimation du degré avec lequel le facteur F_i est assuré (une valeur quantifiant le facteur F_i), alors l'estimation quantitative de la qualité générale

est donnée par l'expression :

$$Qt(X_0) = \Sigma_{i=1}^{n} \alpha_i * Qt(F_i)$$

L'évaluation de tous les facteurs F_i est basée sur celle de leurs descendants. Si un critère C_j admet différents prédécesseurs, en d'autres termes il participe à la définition de la qualité de plusieurs facteurs, l'importance de sa contribution dans chacun des facteurs n'est souvent pas la même. Nous associons donc à chaque arc (F_i, C_j) connectant un facteur F_i à un critère $C_j) \in \Gamma_v(F_i)$ un coefficient de pondération $\beta_{(i,j)} \in (0 \dots 1)$ telle que :

$$\Sigma_{j=1}^{|\Gamma_v(F_i)|} \beta_{(i,j)} = 1\,;$$

L'évaluation d'un facteur Fi est alors définie par :

$$Qt(F_i) = \Sigma_{j=1}^{|\Gamma_v(F_i)|} \beta_{(i,j)} * Qt(C_j)\ (C_j \in \Gamma_v(F_i))$$

De la même manière, le sous-critère $SC_k \in \Gamma_v(C_j$ admet une importance relative par rapport à C_j et pour laquelle on définit un coefficient de pondération $\gamma_{(j,k)}$. Finalement l'évaluation du critère C_j est définie par l'expression :

$$Qt(C_j) = \Sigma_{k=1}^{|\Gamma_v(C_j)|} \gamma_{(j,k)} * Qt(SC_{j+1,k})\ SC_{j+1,k} \in \Gamma_v(C_j)$$

De la même façon, un sous-critère SC_n, le n^{me} descendant du sous-critère SC_m admet une importance par rapport à SC_m évaluée par le coefficient $\lambda_{(m,n)}$ et le calcul de SC_m est assurée par l'expression :

$$Qt(SC_m) = \Sigma_{n=1}^{|\Gamma_v(SC_n)|} \lambda_{(m,n)} * Qt(SC_n)$$

Si le sous-critère SC_n est directement mesurable et si M_p est la p^{me} métrique fournissant une estimation qualitative de SC_n, cette métrique admet également un coefficient de pondération $\tau(n,p)$ L'évaluation e SC_n est alors définie par l'expression :

$$Qt(SC_n) = \Sigma_{p=1}^{|\Gamma_v(SC_n)|} \tau(n,p) * M(p)\ M(p) \in \Gamma_v^{-1}(SC_n)$$

Les coefficients de pondération sont initialisés par l'expert de la qualité d'une application qui, pour cela, se réfère à la hiérarchie des priorités des différents attributs qualitatifs. Cette initialisation peut se faire de façon incrémentale. Ainsi, il est possible qu'au début seule les facteurs et les critères sont pondérés et en fonction du processus de raffinement les sous-critères et les métriques sont pondérés de façon interactive et incrémentale au fur et à mesure de leur raffinement.

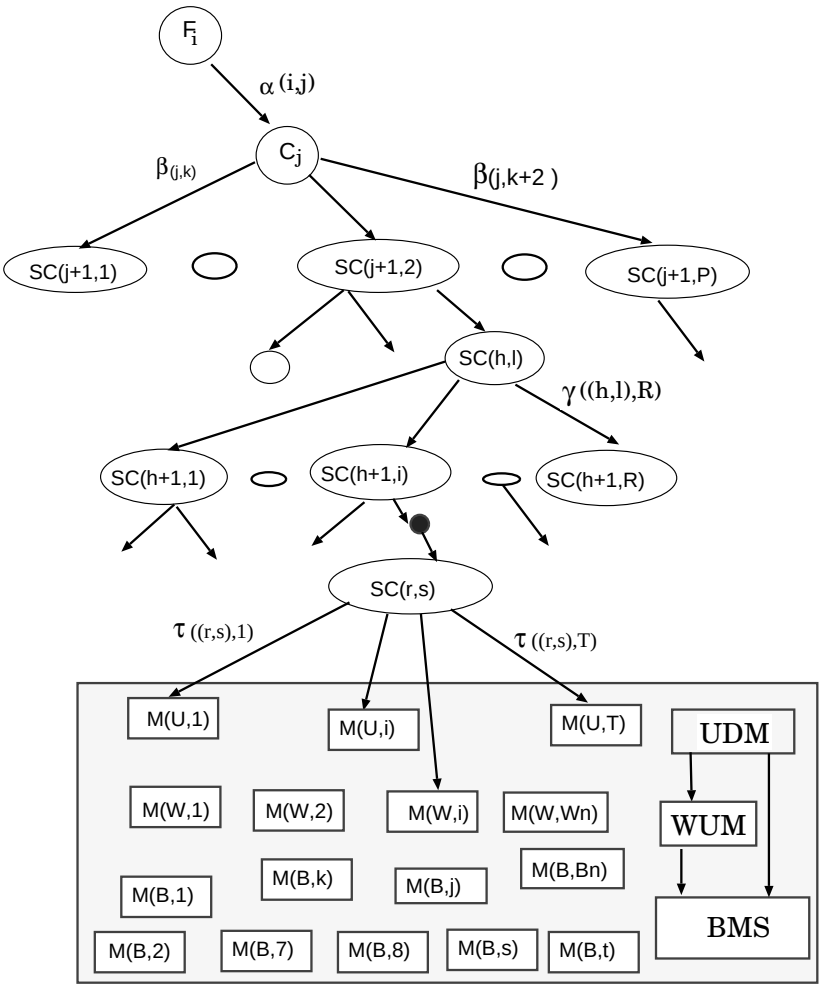


Figure 5.4: Coefficients de pondération associés aux attributs qualitatifs

5.3.7 Classification des métriques en couches

Dans le cadre du modèle QMSE, nous avons défini une classification des métriques en couches comprenant un noyau et deux autres couches. Le noyau est constitué de l'ensemble des métriques dites de base ou BMS (Basic Metric Set). Ce sont des métriques qu'on qualifiera d'atomiques dans le sens où leurs valeurs ne peuvent pas être obtenues par combinaison de celles d'autres métriques. Comme toutes les autres métriques, une métrique de base est destinée à vérifier la présence d'un attribut de la qualité dans un ou plusieurs artefacts, et à qualifier ou quantifier cet attribut. Le choix des attributs est pragmatique et est basé sur l'expérience en matière d'évaluation qualitative des applications logicielles.

Les métriques de base sont insuffisantes pour répondre aux besoins des diverses évaluations des attributs de la qualité. Nous avons donc défini deux autres couches de métriques que sont :

- WUM (Widely-Used Metrics) représentant les métriques largement utilisées et proposées dans la littérature comme moyen d'évaluation de certains attributs qualitatifs. Ces métriques sont généralement largement utilisées dans l'industrie. Elles sont formulées par des fonctions ayant comme paramètres des métriques de base.
- UDM (User-Defined Metrics) sont des métriques définies par l'utilisateur qui est un expert de la qualité. L'utilisation de ces métriques se justifie par l'expérience de l'expert. Cette expérience peut reposer sur des études empiriques qui justifient leur utilisation dans un contexte bien précis.

Cela peut être pertinent pour la mesure d'un aspect spécifique d'une application logicielle ou d'un contexte de développement donnés. UDM répond donc à des besoins spécifiques et exprime le point de vue de l'expert par rapport à une situation donnée. La définition de ces métriques est généralement basée sur les deux couches internes à savoir les BSM et les WUM. La définition de ces métriques par l'expert qualité également contenir les conditions de leurs utilisation.

Finalement, une métrique de l'ensemble WUM est une fonction ayant comme paramètres :

1. un ou plusieurs artefacts cibles de l'évaluation
2. une ou plusieurs métriques appartenant à BMS

Chaque métrique de UDM est une fonction ayant comme paramètres :

1. un ou plusieurs artefacts cibles de l'évaluation
2. une ou plusieurs métriques appartenant à BMS $\bigcup$ WUM

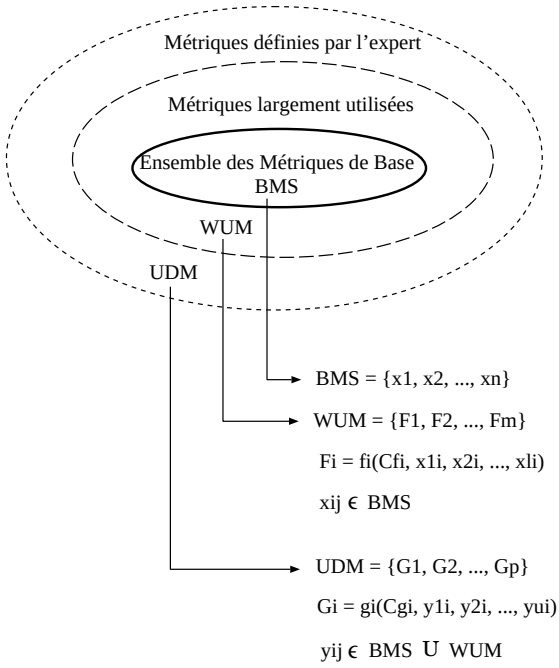


Figure 5.5: Classification des métriques en couches

5.4 Instanciation du modèle

La phase la plus importante dans l'activité de l'assurance qualité est l'instanciation de l'arbre de la graphe étendu. Cela se traduit par :

- La désignation des facteurs de la qualité $_{\mathcal{F}} = \{F_1,\ F_2,\ \ldots,\ F_m\}$.
- Pour chaque facteur F_i, la désignation des critères sous-jacents
$$\Gamma_v(F_i) = \{C_{i1},\ C_{i2},\ \ldots,\ C_{in_i}\}$$
- Pour chaque critère, la désignation des sous-critères qui le raffinent, ou des métriques permettant de l'évaluer dans son état courant de raffinement. Cela revient à définir pour tout critère C_i, les valeurs de $\Gamma_v(C_i)$:
$$\Gamma_v(C_i) = \{SC_{i,1},\ SC_{i,2},\ \ldots,\ SC_{i,n_i}\}$$

– Réitérer la même démarche sur les sous-critères jusqu'aux sous-critères descendants directement mesurables.
– Choisir pour chaque critère C_i ou sous-critère $SC_{i,j}$ considéré comme directement mesurable par les métriques d'évaluation $\{M_{i1}, M_{i2}, \ldots, M_{in}\}$, avec les conditions éventuelles d'application de chaque métrique.

5.4.1 Exemple : évaluation de la modularité

Pour illustrer la démarche d'instanciation, nous l'appliquons sur les codes écrits en Java, langage objet orienté par excellence, en nous focalisant sur l'évaluation de la modularité.

La modularité sera évaluée en fonction des facteurs prioritaires de l'application en cours et des critères sous-jacents à ces facteurs. Pour cela, on doit désigner un ensemble de sous-critères considérés comme les plus prioritaires pour la modularité.

Dans certains cas et pour des exigences fortes, on serait amené à "sacrifier" *certains* sous-critères de la modularité au profit d'autres critères, jugés plus prioritaires, mais présentant une évolution conflictuelle défavorable avec les sous-critères de la modularité. Par exemple, une dérogation de limitation de la hiérarchie (*sous-critère de la modularité*) au profit de la performance en temps de réponse (*sous-critère de la performance*).

Ainsi, un critère de la qualité ne peut pas être ramené à une liste figée de sous-critères.

5.4.2 Elaboration des sous-critères de la bonne modularité

Une partie des sous-critères de la modularité est *invariante*, donc valable quelle que soit l'application, l'autre partie peut être modulée en fonction des paramètres suivants :

1. les méthodes de conception préliminaire et détaillée,
2. la taille de l'application,
3. le type de l'application en cours,
4. l'expérience de l'équipe du développement.

5.4.2.1 Sous-critères dérivés de la méthode de conception

Afin d'assurer une bonne traçabilité, les critères d'un code modulaire sont dépendants des méthodes de conception adoptées. Ainsi, l'utilisation d'une méthode de

conception visant une modularité hiérarchique, avec des types de dépendances spécifiques entre composants, conditionne une partie des sous-critères de la modularité du code source correspondant.

En effet, un sous-critère relatif à l'absence de variables globales, ne peut être considéré de la même façon selon que l'application soit conçue avec la méthode HOOD ou avec une méthode autorisant des variables globales. L'adoption des mêmes sous-critères de modularité pour des codes sources associés à deux méthodes de conception différentes, comporte un risque évident de contradiction avec la traçabilité.

5.4.2.2 Sous-critères dérivés de la spécificité de l'application

Une application peut être caractérisée comme l'une et/ou l'autre des suivantes :

1. application évolutive
2. application destinée à une utilisation locale
3. application destinée à un public spécialisé
4. application destinée à une diffusion commerciale visant un large public
5. application exigeant des performances particulières en temps de réponse et/ou en espace-mémoire
6. application où des vies humaines (ou sommes importantes d'argent) sont impliquées
7. application embarquée
8. application parallèle
9. application distribuée
10. etc.

5.4.2.3 Sous-critères dérivés de la taille de l'application

1. Sous-critères liés à l'utilisation de paquetage ;
2. Sous-critères liés à l'utilisation de la généricité.

5.4.2.4 critères choisis par l'équipe de développement et le responsable de l'AQL

Ces critères sont purement pragmatiques et basés sur l'expérience particulière d'une équipe dans un environnement bien défini du développement. Ce sont par exemple :

1. le taux de composants modulaires surchargés,
2. le seuil de composants modulaires renommés,
3. etc.

5.4.2.5 Critères invariants

1. Complexité des relations horizontales entre composants de même niveau abstro-granulaire,
2. complexité des relations verticales entre composants appartenant à des niveaux différents

Comme exemple d'évaluation de la qualité, nous avons choisi la quantification des relations modulaires, que nous raffinons de la manière suivante :

$< Qt[Relations\_modulaires] >$ [1] $\Longrightarrow$

$< Qt\ [Appel] >$
$< Qt\ [Inclusion] >$
$< Qt\ [Importation] >$
$< Qt\ [Exportation] >$
$< Qt\ [Communication] >$
$< Qt\ [Utilisation\_de\_ressources\_communes] >$
$< Qt\ [Héritage] >$
$< Qt\ [Dépendance] >$
$< Qt\ [Hiérarchie] >$

À chaque type de relation, un certain nombre d'attributs sont à définir en fonction des futures besoins contrôle de l'évolution, en observation et en évaluation qualitative.

1. $< Qt[A] > \Longrightarrow < Qt[B] > < Qt[C] >$ signifie que les quantifications de A exigent des quantifications de B et de C.

5.5 Mapping entre les modèles SMSE et QMSE

Chaque ensemble $\Phi_i.Art$ d'artefacts d'une phase du cycle de développement du logiciel correspond à un multi-graphe construit selon le modèle structurel instancié (Figure 5.6). L'ensemble qualitatif $\Phi_i.\Sigma_{art}$ correspond au graphe de la qualité dénoté par $Qi(\Phi_i.\Sigma_{art})$ élaboré par l'expert qualité et permettant d'évaluer les artefacts de Φ_i selon plusieurs points de vue qualitatifs représentés par des facteurs et critères.

Tout changement structurel (Δ_S) opéré sur un artefact peut causer un changement qualitatif (Δ_Q) sur le critère qualitatif correspondant. L'impact de Δ_Q propagé progressivement aux critères prédécesseurs dans le graphe de la qualité et ainsi jusqu'aux facteurs peut être tracé jusqu'à la qualité globale ou générale du logiciel.

Nous pouvons déterminer le changement qualitatif Δ_Q résultant d'un changement structurel affectant un artefact art_x. Alors, la notation $\Delta_S \Rightarrow \Delta_Q$ signifie que le changement structurel Δ_S implique un changement qualitatif Δ_Q. Considérons un ensemble de métriques $M = \{m_1, m_2, m_3, \dots m_n\}$ associées à l'artefact art_x. Les valeurs associées à ces métriques peuvent être représentées par l'ensemble $V = \{v_1, v_2, v_3, \dots v_n\}$. Un changement structurel Δ_S appliqué sur art_x peut résulter en de nouvelles valeurs de ses métriques associées que l'on notera $V' = \{v'_1, v'_2, v'_3, \dots v'_n\}$. Ceci est formulé par :

$$\begin{aligned} m_1, m_2, m_3, \dots m_n\,(art_x) &= v_1, v_2, v_3, \dots v_n \\ &\Rightarrow M(C_x) = V \end{aligned}$$

Après le changement, $\Delta_Q(art_{x'}) = \Delta_Q(art_x) + \Delta_Q(change)$, alors nous obtenons la valeur de V' telle que :

$$\begin{aligned} m_1, m_2, m_3, \dots m_n(art_{x'}) &= v'_1, v'_2, v'_3, \dots v'_n \\ &\Rightarrow M(art_{x'}) = V' \end{aligned}$$

La différence des valeurs de métriques $(\Delta M = V' - V)$ dénote une variation des critères qualitatifs correspondants. La différence, correspondante, des valeurs de métriques $\Delta m_i, \Delta m_{i+1}, \Delta m_p$ représente le changement du critère de la qualité qu'on notera ΔC. Le changement d'un critère peut affecter la valeur d'autres critères associés $(\Delta C_j, \Delta C_{j+1}, \Delta C_q)$ du même niveau. Cette différence identifiée au niveau des attributs qualitatifs matérialise l'impact qualitatif.

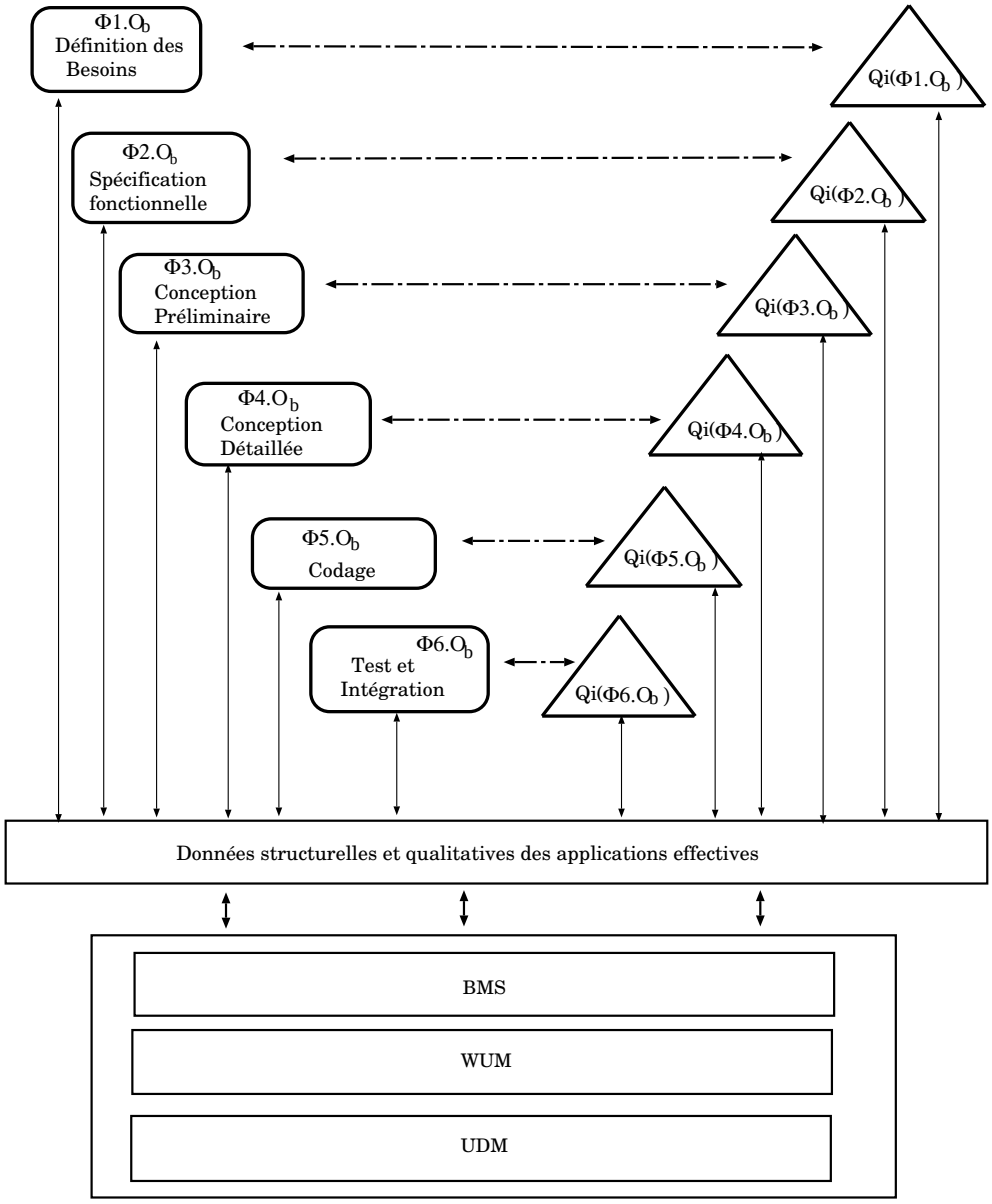


Figure 5.6: Graphe structurel des artefacts et le graphe qualitatif associé

5.6 Conclusion

Améliorer le contrôle de l'évolution du logiciel en évitant la dégradation de la qualité dépend de la compréhension des dépendances des artefacts logiciels incluant ceux relatifs à la qualité. Ce chapitre présente un modèle intégré du logiciel permettant de traiter la problématique de l'analyse de l'impact du changement selon les points de vue structurel et qualitatif. L'approche proposée permet plus de traçabilité de flux de l'impact du changement.

Dans ce chapitre, nous avons traité la problématique de représentation des attributs qualitatifs associés à des artefacts appartenant à différentes phases du cycle de développement du logiciel. Cela nous a conduit à définir un graphe de la qualité du logiciel que nous avons mis en correspondance avec le graphe représentant la structure du logiciel. Cela a rendu possible l'implémentation d'outils automatiques permettant d'effectuer une analyse de l'impact du changement du point de vue qualitatif.

Nous avons particulièrement adressé la problématique de représentation des attributs qualitatifs de chaque phase du cycle de vie du développement du logiciel de façon à permettre l'évaluation de chacun de ces attributs y compris celui représentant la qualité générale du logiciel. Cela repose sur un processus de raffinement de ces attributs assurant ainsi une correspondance entre les facteurs qui représentent des attributs extérieurement appréciables de la qualité et les métriques directement applicables sur des artefacts logiciels n passant par des critères et des sous-critères. Nous avons également représenter des relations horizontales entre attributs de la qualité permettant de définir des liens de contradiction ou plus généralement de causalité en matière de contribution à la qualité générale du logiciel.

Quatrième partie

Validation

Chapitre 6

Prototype de Validation

6.1 Introduction

Pour une validation opérationnelle des modèles que nous proposons pour l'évolution du logiciel, nous avons mis en oeuvre une plate-forme intégrée et hébergeant différents modules ou composants implémentant chacun un des aspects de l'évolution évoqués dans cette thèse. Cette plate-forme permet entre autres une manipulation simple des différents graphes des artefacts logiciels incluant le graphe de la qualité et de l'architecture, etc.

Nous avons utilisé un système de réécriture de graphes comme un moyen de formalisation opérationnelle ou constructive des opérations de gestion de l'évolution du logiciel. Notre plate-forme ainsi que tous les modules qui la constituent ont été implémentés au sein de l'IDE (Integrated Development Environment) *Eclipse*[1] en forme de plug-ins. Le choix d'Eclipse a été dicté par le fait que c'est un IDE open source intéressant un nombre de plus en plus croissant de développeurs à travers le monde (122). Ceci est, constitue à notre sens le meilleur moyen de diffuser nos implémentations au sein e la communauté des développeurs pour qui elles ont été conçues en premier lieu. Cela, nous semble t-il, devrait favoriser l'utilisation puis le test de nos implémentations et donc leur amélioration continue. C'est aussi une bonne opportunité de trouver des partenaires testant nos outils sur des projets de tailles réelles. Un projet Eclipse manipule et gère généralement un ensemble de ressources pouvant être des fichiers de code source, des documents web, des librairies, des digrammes de conception, des modèles,

1. http://www.eclipse.org/

etc. La plate-forme que nous avons développée analyse, dans le cadre d'Eclipse, des artefacts hétérogènes par différentes sortes de parsers et permet une représentation de ces artefacts de façon homogène à l'aide de graphes.

Nous avons implémenté un système à base de connaissances (KBS : *Knowledge Based System*) permettant d'analyser l'impact du changement de façon semi-automatique. Il permet également la gestion de méta-modèles dans une base de connaissances. La base de faits du KBS est constituée du référentiel d'artefacts logiciels extraits par différents parseurs et différents outils d'analyse statique et également par des informations fournis par les différents experts ou acteurs du développement (architectes, spécialistes qualité, etc.). Les règles du KBS implémentent les algorithmes d'analyse et de propagation de l'impact et sont en grande partie des implémentations des formalisations introduites au chapitre 4.

Dans ce chapitre, nous décrivons la structure et la conception du prototype de validation de l'approche de modélisation intégrée que nous proposons. Le prototype permet, entre autres, d'analyser les différentes artefacts constituant une application logicielle et a principalement été expérimenté dans le cas des applications distribuées de type Java 2 Platform Entreprise Edition (Java J2EE). Le prototype contient un éditeur de graphes permettant une visualisation, exploration et manipulation simplifiées des différents graphes d'artefacts logiciels que nous avons définis. Il est ainsi possible d'invoquer une opération de modification sur un élément du graphe et de déclencher une analyse *a priori* de l'impact de ce changement. Le résultat de cette invocation se traduira par une visualisation de l'impact sur les différents nœuds et arcs du graphe avec la possibilité de passer à tout moment au code source ou à d'autres graphes (comme le graphe de la qualité, etc.). Tout ceci s'effectue bien entendu au sein de la plate-forme Eclipse. En d'autres termes le développeurs dispose d'information sur l'analyse de l'impact sans quitter son environnement de travail habituel et il est donc possible de mener en même temps une activité de développement et une activité d'analyse et de mesure du changement. Le prototype permet également de définir des règles définies par l'utilisateur, en plus de celles déjà définies pour une gestion plus spécifique de l'impact. Il permet également de tracer l'impact du changement sur un artefact particulier et fournit un support de prise de décision durant le processus de changement.

Ce chapitre est organisé comme suit : la section 6.2 résume l'architecture et la conception générale du prototype de validation. La section 6.3 décrit l'environnement

et le contexte d'implémentation du système à base de connaissances. La section 6.4 présente les résultats empiriques de notre approche en considérant un scénario exemple d'une application e-commerce de type projet web dynamique multi-tiers. La section 6.5 discute de façon détaillée l'analyse de l'impact qualitatif comme une conséquence d'un changement structurel. La section 6.6 conclut le chapitre et montre brièvement les perspectives de développement du prototype.

6.2 Architecture Globale du prototype de validation

Notre plate-forme, appelée *Architect* est un outil d'analyse d'impact du changement des applications distribuées. Il est construit à l'aide de Eclipse plug-in Development Environment (PDE) [1]. Eclipse est une plate-forme open source largement utilisée pour construire des plate-formes ouvertes et extensibles de développement constituées d'outils et de runtimes destinés à la construction, au déploiement et à la gestion de logiciels en incluant toutes les phases du cycle de vie. Eclipse Plug-in Development Environment ou PDE fournit un ensemble d'outils assistant le développeur dans chaque étape du développement de plug-ins de l'analyse au déploiement. Nous avons développé un ensemble de plug-ins Eclipse destinés à l'analyse de l'impact du changement affectant les applications distribuées développées à l'aide d'Eclipse. La figure 6.1 montre une copie d'écran de *Architect*.

Architect est constitué de quatre plug-ins majeurs, ce sont : Eclipse multilingual parser, Eclipse software modeler, Eclipse graph visualizer, et Eclipse change propagation analyser. L'architecture interne montrant l'interaction de ces plug-ins est schématisée par la figure 6.2. Nous discuterons les détails de l'implémentation de ces quatre plug-ins dans les sections qui suivent.

6.2.1 Le plug-in Eclipse multi-lingual parser

Eclipse multi-lingual parser est un plug-in étendant la plate-forme Eclipse en incluant des parsers permettant l'analyse de documents sources de divers langages incluant C, C++, Java, COBOL, MySQL (schémas de BD), PHP, Perl, Python, etc. (123). En effet, notre but a toujours été de proposer un modèle d'artefacts logiciels multi-langages. Il a donc été naturel pour nous de proposer des analyseurs pour un nombre important

1. http://www.eclipse.org/pde/

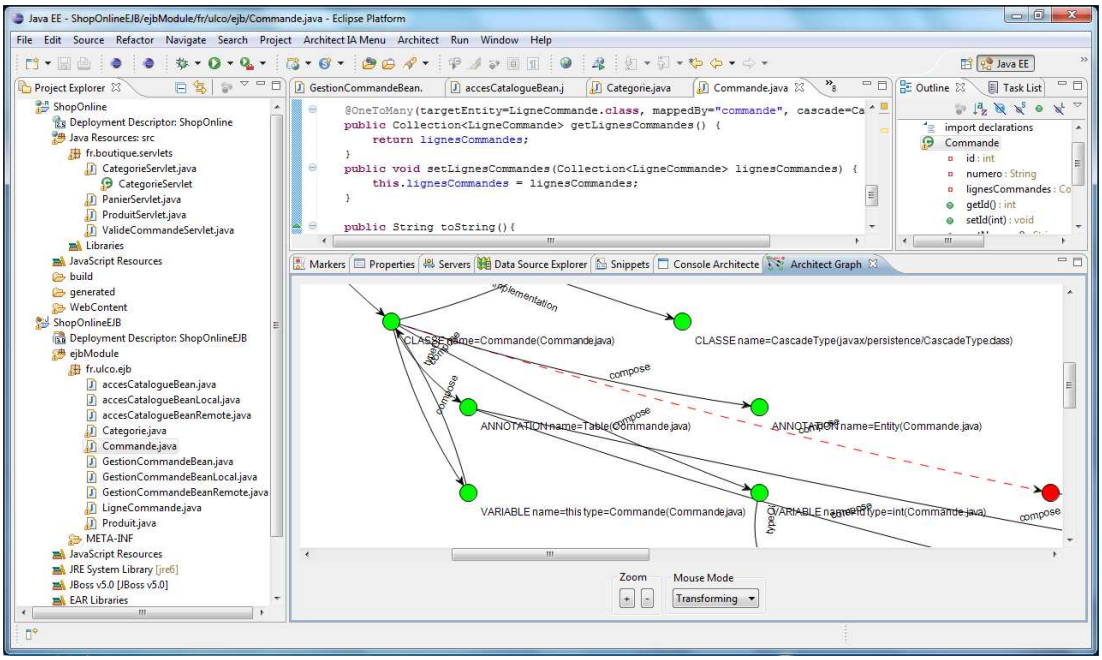


FIGURE 6.1: Impression d'écran d'élaborer la propagation d'impact du changement avec *Architect*

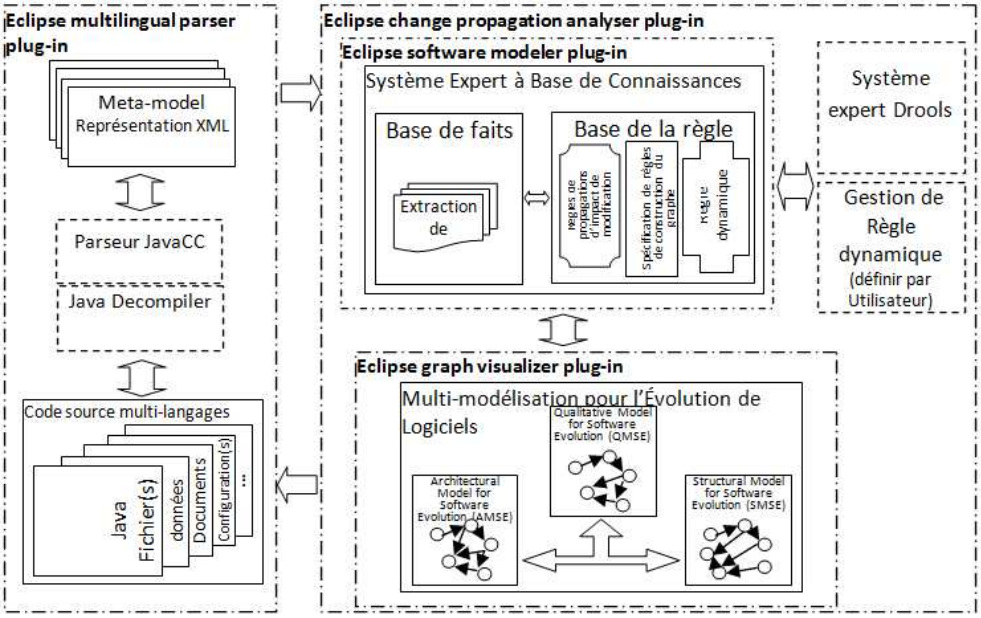


FIGURE 6.2: L'architecture du système expert en forme de plug-ins dans l'environnement de développement Eclipse

de langages et spécialement ceux généralement utilisés dans les projets réels. Les parsers ont été développés à l'aide de *Java Compiler Compiler* (JavaCC). JavaCC est un compilateur de compilateurs et un générateur d'analyseurs lexicaux entièrement écrit en JAVA et intégré la plate-forme Eclipse. Il permet d'interpréter la grammaire d'un langage donné et d'en générer un parser. Cela nous a simplifié la possibilité de rajouter de nouveaux langages au fur et à mesure de l'avancement du développement de notre prototype et ceci est d'autant vrai qu'il est possible de réutiliser les éléments des grammaires des langages pour lesquels nous avions déjà implémenté des parsers. Nous avons donc développé de nombreux parsers incluant également un parser de bytecode Java. Pour ce dernier parser, une étape de décompilation a été nécessaire. Nous avons donc intégré un décompilateur Java appelé *JreversePro*[1]. C'est un décompilateur Java open source utilisé sous la licence GNU GPL.

A partir des fichiers source d'entrée, Eclipse multilingual parser génère un ensemble de représentations à base de XML représentant les différents artefacts logiciels et leurs relations tels que spécifiés par notre modèle SMSE. Ce sont des instanciations de SMSE pour des codes sources, des descriptions d'architectures, les librairies utilisées, les schémas de bases de données, etc. Les descriptions XML résultat sont alors utilisées par le plug-in Eclipse modeler pour construire le graphe des artefacts logiciels.

6.2.2 Le plug-in Eclipse software modeler

Le plug-in *Eclipse software modeler* implémente les deux modèles SMSE et QMSE. Les deux modèles sont instanciés par des graphes dont les nœuds représentent les artefacts logiciels et les arcs les relations reliant ces artefacts. Software modeler génère des faits qui alimentent le système à base de connaissances en forme d'assertions. C'est en effet le système à base de connaissances qui constitue le composant centrale de notre plate-forme puisqu'il implémente tous les processus d'analyse de l'impact du changement. Le plug-in Eclipse software modeler fournit des services de base pour la création et la transformation de graphes des artefacts logiciels affectés par le changement en forme de documents XML. L'analyseur de flux XML réalise une mise en correspondance entre les artefacts et les relations de nos modèles et des balises XML générés par les parsers multi-langages implémentés par le plug-in Eclipse Multi-lingual parser.

1. http ://sourceforge.net/projects/jrevpro/

6.2.3 Le plug-in Eclipse graph visualizer

Le plug-in *Eclipse graph visualizer* permet la visualisation des graphes ainsi que la manipulation de ces graphes en permettant de définir des opérations de changement sur les nœuds et les arcs des graphes. Il permet ainsi, une gestion interactive de l'évolution et la maintenance des systèmes logiciels.

Nous avons utilisé le framework *Java Universal Network/Graph (JUNG)*[1]. C'est une librairie open source pouvant être réutilisée pour la modélisation, l'analyse et la visualisation de données en forme de graphes ou réseaux. La librairie permet la définition de la structure de données Graphe et d'utiliser certaines primitives de construction d'interfaces utilisateurs associées aux outils de manipulation de graphes. Nous avons spécialisé la classe Graphe, disponible dans la librairie JUNG en une classe appelée *ArchitectGraph*. Cette classe a été utilisée pour intragir avec les autres classes de l'API Java et celles du système à base de règles *Drools*[2] qui a servi à l'implémentation de notre système à base de connaissances.

6.2.4 Le plug-in Eclipse change propagation analyser

Le plug-in *Eclipse change propagation analyser* a été implémenté à l'aie du moteur de règles orienté objet Drools. Il permet la propagation de l'impact du changement des artefacts logiciels. Drools est une plate-forme de gestion de règles métier fournissant un atelier intégré pour la définition et l'exécution de règles. Il permet également la définition et l'exécution de workflow ainsi que la gestion des événements. Nous avons utilisé Drools pour l'écriture des règles d'inférence, des règles de propagation de l'impact du changement et des règles expertes. Ces différentes règles sont invoquées de façon interactive au cours de la manipulation interactive des graphes d'artefacts logiciels. La propagation de l'impact est basée sur des règles de gestion de l'impact du changement déclenchées par l'insertion de faits effectuée par le facts builder. Le facts builder insère des faits dans la base de faits représentant des artefacts logiciels, leurs relations ainsi que des événements comme la modification d'un artefact, etc. Une règle de propagation de l'impact du changement est une règle de production ont les parties gauches et droite sont des prédicats du premier ordre. Le processus de matching des faits avec les parties gauche ou condition des règles est réalisé par un algorithme *Rete*.

1. http://jung.sourceforge.net/
2. http://www.jboss.org/drools/

6.3 Implémentation du système à base de connaissances

Nous avons développé un système expert en guise d'implémentation du système à base de connaissances. Il a pour principales fonctions la gestion des méta-information sur les artefacts logiciels et l'implémentation du processus de propagation de l'impact du changement. Les deux principaux constituants du système à base de connaissances sont la base des faits et la base des règles. Ces deux constituants requièrent les fonctionnalités développées par les différents plug-ins. Le multilingual parser et le software modeler jouent en effet, un rôle important dans la construction et la gestion de la base de faits. Le graph visualizer agit sur la base de faits, représentant une partie du référentiel des artefacts logiciel, et interagit avec le change propagation analyser qui requiert l'utilisation de la base de règles. La figure 6.3 explicite brièvement le workflow de la plate-forme *Architect*. Dans les sections suivantes, nous discutons, avec plus de détails, la construction des bases de faits et de règles.

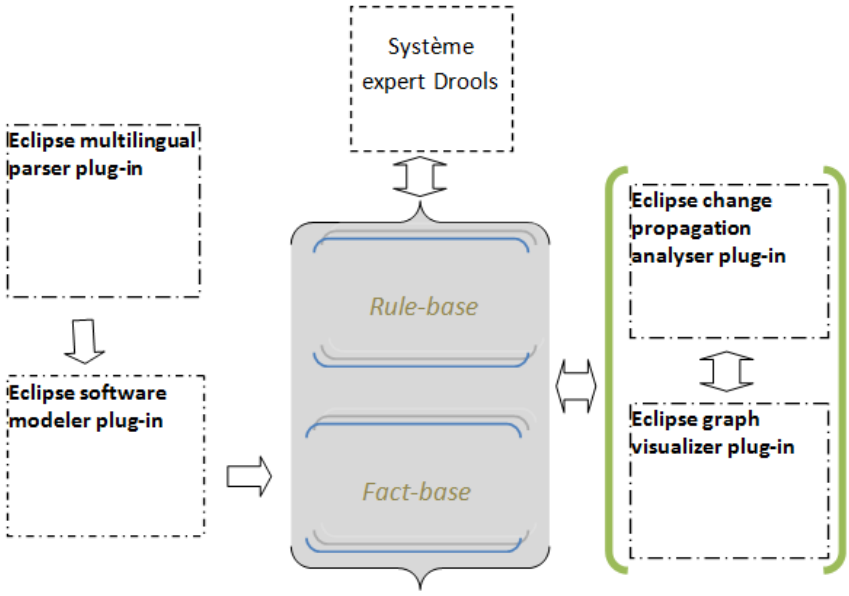


Figure 6.3: Flux de travail en *Architecte*

6.3.1 Synthèse de la base de faits

La base de faits est, en grande partie construite par l'exécution des plug-ins Eclipse multilingual parser et Eclipse software modeler. Nous discutons, dans ce qui suit, le

fonctionnement de chacun de ces deux constituants.

6.3.1.1 Fonctions du plug-in Eclipse multilingual parser

Le plug-in *Eclipse multilingual parser* est une extension de Eclipse conçue dans le but de mettre en oeuvre des analyseurs statiques de différents types de codes sources. Comme discuter plus haut, les codes sources peuvent être des fichiers de programmes développés dans différents langages, des schémas de bases de données, des fichiers de configuration écrits en XML, des descriptions d'architectures, etc. Le plug-in fournit une représentation uniforme et homogène de ces différents types d'artefacts en utilisant le langage XML.

Le parsing ou analyse est basée sur la grammaire de chacun des langages considérés. A ces grammaires sont associées des routines sémantiques pour la construction d'arbres abstraits syntaxiques ou AST (*Abstract Syntax Tree*) dont le parcours et l'exploitation serviront à la production des représentations XML. Ces représentations forment en effet l'instanciation du modèle SMSE exprimée en termes d'artefacts et de relations.

The *multi-lingual parser* plug-in trigger the parsing of the development source codes of each project file. The *JavaCC* produces a lexical analyzer recognizing grammar and syntax from specifications. The plug-in is extensible, to consider more source code languages, by the addition of a grammar describing the semantic instructions necessary for the extraction of artifacts and relations depending on the SMSE model. The semantic instructions can translate a language element in an XML tag representing an artifact or a relationship.

Le plug-in *Eclipse multilingual parser* déclenche l'exécution des différents parsers (selon la nature du langage de l'artefact à analyser). JavaCC produit un analyseur lexical reconnaissant les différentes entités lexicales de la grammaire considérées (mots clés, identifiants, opérateurs, etc.). Une grande partie des grammaires et des analyseurs lexicaux sont réutilisables favorisant ainsi l'extensibilité du plug-in par rapport à l'adjonction de nouveaux langages.

Le résultat e l'exécution du plug-in Eclipse multilingual parser, en d'autres termes les documents XML générés, est utilisé pour la construction du graphe d'artefacts logiciels par le plug-in Eclipse software modeler.

6.3.1.2 Fonctions du plug-in software modeler

Les représentations XML, obtenues par le biais du plug-in *multilingual parser*, représentent les artefacts et les relations extraits à partir des différents codes sources. Le plug-in *software modeler* est la deuxième extension d'Eclipse permettant l'implémentation du modèle SMSE par le biais, entre autres, d'analyseurs de flux XML. Les représentations XML ont une forme de customisation ou adaptation du langage *GXL (Graph eXchange Language)*[1] qui est une langage standard adopté par la communauté du génie logiciel pour la représentation et la manipulation des artefacts logiciels en forme de graphes (41). Il s'agit, donc de réutiliser à la fois les tags définis par GXL (qui sont des tags génériques) et de les spécialiser par des tags spécifiques au modèle SMSE.

Le *software modeler* fournit un ensemble de services de base pour la création et la transformation de graphes représentés par des documents XML. Il permet, entre autres, d'instancier les éléments du modèle SMSE en forme de nœuds et arcs de graphes.

Le *software modeler* peut être utilisé pour l'analyse es graphes ainsi construits et cela pour inférer des relations complexes non explicitement représentées dans les documents XML. C'est notamment le cas de relations telles que la surcharge des méthodes, le polymorphisme des objets, la relation de projection d'un composant d'une architecture en une entité d'un code source, etc. Ces relations sont déterminées par l'utilisation de règles que nous avons appelées règles d'inférence. Il est ainsi possible d'étendre ce type de règles pour prendre en compte un nombre croissant de relations complexes.

6.3.2 Synthèse de la base de règles

Le développement de la base de règles requiert des invocations mutuelles des plug-ins *graph visualizer* et *software propagation analyser*.

Le système expert a pour but l'implémentation du processus de propagation de l'impact du changement. Comme explicité plus haut, notre système expert consiste en une base de faits contenant les descriptions des artefacts logiciels et de leurs relations. La base de faits est manipulée lors de la production ou le changement des graphes d'artefacts logiciels. Ainsi, lors de l'application e changements sur ces graphes, par exemple, les faits ainsi générés peuvent déclencher les règles destinées à la propagation de l'impact.

1. http://www.gupro.de/GXL/

Pour cela, nous adoptons un système d'exécution des règles de types ECA (Evénement-Condition-Action). Donc un fait représentant un événement peut si une condition est vérifiée déclencher une action. Les conditions peuvent elles mêmes être des combinaisons de faits et les actions sont génératrices d'autres faits qui peuvent à leurs tours déclencher d'autres règles, etc.

Nous avons défini trois catégories de règles :

1. Les règles d'inférence : Ces règles ont comme principal fonction l'identification de relations complexes. En effet, certaines relations ne sont pas évidentes à identifier par simple analyse des codes sources.
2. Les règles stratégiques : Ces règles implémentent le processus de propagation e l'impact du changement tel que défini dans le chapitre 4. Quand un artefact est affecté par une opération de changement, le fait correspondant est modifié pouvant ainsi déclencher les règles de propagation de l'impact. Cela va initier le processus de marquage des artefacts et relations affectés par le changement.
3. Les règles expertes : Ce sont des règles définies par l'utilisateur pour la gestion de dépendances ad hoc.

6.3.2.1 Les règles d'inférences

L'analyse ou parsing des codes sources permet de générer un graphe partiel des artefacts logiciels. Comme discuté plus haut, certaines relations peuvent difficilement être instanciées et ne sont donc pas prise en compte dans ces graphes. Nous pouvons donc utiliser les faits disponibles à partir des graphes partiels pour générer l'instanciation des relations complexes. Le système à base de règles Drools, que nous avons utiliser comme moteur de règles, utilise donc les données disponibles dans les graphes partiels pour générer les relations complexes.

Nous avons ainsi définies règles ayant pour but de compléter le processus de construction des graphes d'artefacts. Ces règles sont déclenchées par le système expert à chaque insertion de faits formant leurs prémisses. Le système expert déduit donc, en utilisant les règles et les faits disponibles, des relations entre deux artefacts ou nœuds déjà disponibles dans la base de faits mais no encore reliés par ce type de relations.

Nous explicitons, prendre des exemples montrant le fonctionnement de certaines règles d'inférence.

1. Une annotation entre des méthodes Java et des colonnes d'une table d'une base de données relationnelle peuvent indiquer que toute opération de manipulation ou de recherche de données impliquant cette colonne concernent la méthode Java (on dit que la méthode java est annotée par la colonne) . Si le parsing du code java extrait quelques annotations, (telles que @Entity, @Table, etc.) cela veut dire que ces annotations représentent un mapping objet-relationnel (ORMap) et Drools peut inférer ce type de relation telle que nous le montrons dans le code de la règle Drools (listing 6.1).

   Dans notre règle, les artefacts sont définis par la classe « ArchitectNode » et les arcs sont définis par la classe « ArchitectEdge ». Ces classes sont défini par le plug-in graph visualiser et peuvent être utilisées, comme toutes les classes Java, pour l'expression des faits dans Drools. En effet, Drools est un système à base de règle permettant d'exprimer des faits à partir d'objets d'un programme Java en utilisant des expressions de type « MaClasse(attr1==val1, attribut2==val2, ...) ». La règle du listing 6.1 considère quatre artefacts : une annotation entity, une annotation de table, une classe et un champ. Ces artefacts sont reliés par les relations « compose » et « annotation ».

   La clause « when » de la règle spécifie que :

   – s'il existe deux nœuds dans la base de faits instances du type « ANNOTATION » et dont les noms sont « Entity » et « Table » (c'est la manière de désigner une annotation d'entité et une annotation de table) et s'il existe un nœud de type « CLASSE » qui contient ces deux annotations (cela est représenté par l'existence d'un arc de label « compose » et ayant le nœud CLASSE comme source et les nœuds annotations comme destination)

   – et s'il existe un autre nœud de type « ENTITY_FIELD » représentant l'annotation d'un champ de la classe et relié à la table par le lien « compose »

   Alors :

   – La règle génère un arc de type « ORMapped » entre le nœud de type « ENTITY_FIELD » et la table.

   Listing 6.1: Spécification de la règle pour la relation de *mapping* entre la classe et annotation d'entité

```
rule "ClassEntityRule"
```

```
 when
  nEntity : ArchitectNode (typeNode=="ANNOTATION", name=="Entity")
  nTable : ArchitectNode (typeNode=="ANNOTATION", name=="Table")
  nClass : ArchitectNode (typeNode=="CLASSE")
  eClassEntity : ArchitectEdge( nodeDest ==nEntity, nodeSrc ==nClass, label=="compose")
  eClassTable : ArchitectEdge( nodeDest ==nTable, nodeSrc ==nClass, label=="compose")
  nEntityField : ArchitectNode (typeNode=="ANNOTATION_FIELD", name=="name", attrValue : value)
  eEntFieldTable : ArchitectEdge( nodeDest ==nEntityField, nodeSrc ==nTable, label=="compose")
  n : ArchitectNode (typeNode == "TABLE", attrName : name -> (attrName.equals("'"+attrValue+"'")) )
 then
  System.out.println ( "Knowledge about CLASS ENTITY - TABLE " + n.getLabel() );
  n.getArchitectGraph().addEdge("ORMapped", n, nTable);
end
```

2. Une autre relation complexe concerne les paramètres utilisés dans une instruction. La clause « when » considère deux nœuds : le premier nœud est une instruction et le second nœud est de type paramètre. La règle décrite s'il un paramètre tel que son nom apparaître dans l'expression d'instruction. Dans ce cas, la clause « then » vérifie la scope de paramètre, parce que on peut avoir des attribut et paramètre avec la même nom, en suite elle décrite l'action d'ajout d'un arc « use » entre instruction et paramètre. La règle d'inférence pour cette relation est montrer dans listing 6.2.

   Listing 6.2: Spécification de la règle *Drools* pour la relation 'use' entre instruction et paramètre

```
rule "StatementUseParameter"
 when
  n2 : ArchitectNode (attrName : name, typeNode == "STATEMENT")
  n : ArchitectNode (attrName2 : name -> (attrName.equals(attrName2)) , typeNode == "PARAMETER")
 then
  if (n.getArchitectGraph().isInScope(n2, n)){
   System.out.println ( "Knwoledge about variable " + n.getLabel() +
   " -> " + n2.getLabel() + " on " + attrName);
   System.out.println ( "STATEMENT USE PARAMETER " + attrName);
   n.getArchitectGraph().addEdge("use", n, n2);
  }
end
```

3. Le listing 6.3 montre une règle simple inférant un lien entre une balise hyperlien et une page html. La règle est auto-descriptive. La clause « when » identifie deux nœuds dont un est de type « LINK » et l'autre de type « HTML_DOCUMENT » et un attribut « href » du nœud de type « LINK » référence le document html. Dans ce cas, la règle crée un arc « call » entre le document html et le lien.

Listing 6.3: Spécification de la règle pour la relation *link* d'un document web

```
rule "LinkWebPageRule"
 when
  n1 : ArchitectNode (typeNode=="LINK")
  n : ArchitectNode (typeNode == "HTML_DOCUMENT")
  eval (n1.getAttributes("href").endsWith(n.getAttributes("source")+".html") );
 then
  System.out.println ( "Knowledge about LINK - HTML " + n.getLabel() );
  n.getArchitectGraph().addEdge("call", n, n1);
end
```

4. Un autre exemple de relations complexes concerne la relation d'appel entre une page web et une servlet. Comme les patterns url public pour accéder aux servlets (dans J2EE) sont donnés dans des fichiers de configuration XML, le listing 6.4. tente d'établir une relation « call » entre un élément d'un formulaire html et une servlet à travers le fichier web.xml.

   Listing 6.4: Spécification de la règle pour la relation *call servlet* entre une formulaire web et servlet

```
rule "FormURLPatternPageRule"
 when
  link : ArchitectNode (typeNode=="form")
  urlpattern : ArchitectNode (typeNode == "url-pattern")
  eval ( urlpattern.getAttributes("text").endsWith(link.getAttributes("href")) );
  servletMap : ArchitectNode (typeNode == "servlet-mapping")
  urlToMap: ArchitectEdge (label=="compose", nodeSrc==servletMap, nodeDest==urlpattern)
  servletClass : ArchitectNode (typeNode == "servlet-class")
  mapToClass: ArchitectEdge (label=="mapped", nodeSrc==servletMap, nodeDest==servletClass)
  classOfServlet : ArchitectNode (typeNode == "CLASSE")
  instanceOfClass: ArchitectEdge (label=="instanceof", nodeSrc==servletClass, nodeDest==classOfServlet)
  doPost : ArchitectNode (typeNode == "METHODE", name == "doPost")
  MapToClass: ArchitectEdge (label=="compose", nodeSrc==classOfServlet, nodeDest==doPost)
 then
  System.out.println ( "Knowledge about FORM - DOGET METHOD" );
  urlpattern.getArchitectGraph().addEdge("call", doPost, link );
end
```

5. Comme exemple de génération de relations complexes à partir de descriptions architecturales, nous considérons la règle de déduction d'une relation entre un connecteur de type « EDC » (Event Data Connection), « EC » (Event Connection) et « DC » (Data Connection) et des nœuds de type port. Dans le graphe initial, obtenu par analyse du code de l'architecture, il n'y a pas de relations entre le connecteur et les ports.

Le listing 6.5 montre la règle qui infère la relation entre un connecteur et son port source et donc génère un arc entre les nœuds correspondants. La règle vérifie l'existence d'un fait, « nPort », correspondant à un nœud ayant un nom et de type « EDP », « DP » ou « EP ». La règle vérifie également l'existence d'un nœuds « nConnect » de type « EDC », « EC » ou « DC ». Si ces deux faits existent alors la règle vérifie la condition testant si le nœud « nConnect » a un attribut « src » contenant le nom du nœud « nPort ». La règle se déclenche alors si ces conditions sont vérifiées et crée un arc entre le nœud « nPort » et « nConnect » et libellé « usePort ».

Listing 6.5: Spécification de la règle pour les relations entre les connecteurs et les ports

```
rule "ConnectConnecToItsPortSrc"
 when
        nPort : ArchitectNode ( portName : name)
        eval (nPort.getTypeNode().equalsIgnoreCase("EDP") ||
              nPort.getTypeNode().equalsIgnoreCase("EP") ||
              nPort.getTypeNode().equalsIgnoreCase("DP")
        )
        nConnect : ArchitectNode( )
        eval (nConnect.getTypeNode().equalsIgnoreCase("EDC") ||
              nConnect.getTypeNode().equalsIgnoreCase("EC") ||
              nConnect.getTypeNode().equalsIgnoreCase("DC")
        )
        eval(nConnect.getAttributes("src").equalsIgnoreCase(portName))
 then
        nConnect.getArchitectGraph().addEdge("usePortSrc",nPort,nConnect);
end
```

6.3.2.2 Les règles stratégiques

Ces règles gèrent la propagation de l'impact des changements. Ce sont des règles qui calculent l'effet de vague ou ripple effect du à un changement d'un artefact. La propagation de l'impact dépend de l'opération de changement, des types de relations reliant les artefacts et de la conductivité de l'impact à travers ces relations. Les règles de propagation de l'impact sont basées sur la notion de violation des invariants des opérations. Le calcul ou l'identification de la propagation de l'impact est obtenu par la $ruleExecuteChangeOperation(op, \Delta_{art})$ (listing 4.4) et $ruleChangeImpactPropagation(op, \Delta_{art})$ (listing 4.5) telle que définie dans le chapitre 4. La règle $ruleChangeImpactPropagation(op, \Delta_{art})$ joue le rôle le plus significatif dans la simulation de la propagation de l'impact. Elle

marque les artefacts affectés si au moins une des assertions de l'invariant de l'opération exécutée est non respecté. Cela provoque le marquage de tous les éléments affectés par la propagation de l'impact. Le marquage d'un nœud change l'état de ce nœud. Le marquage atteint également les arcs symbolisant les relations inter-artefacts.

La propriété *stateNode* de *ArchitectNode* et *ArchitectEdge* décrit l'état d'un élément. La valeur par défaut de cette propriété est `STATE_CREATED`. L'ensemble des états qu'un nœud peut avoir sont définis dans la classe *ArchitectStateNode* appartenant au plug-in graph visualiser. Les valeurs notables de cette propriété sont : `STATE_ADDED`, `STATE_CHANGED`, `STATE_DELETED`, `STATE_DELETED_TEMP`, et `STATE_IMPACTED`. La classe *ArchitectStateNode* est instanciée par un objet *states* qui dispose de variables entières positives ou nulles représentant les différents états (par exemple `STATE_CREATED` = 0). Cela aide à simplifier la formulation de la coexistence de plusieurs états dans un objet. Par exemple la condition `stateNode >= states.STATE_CREATED` est vraie si l'état du nœud appartient à tous les états possibles (puisque un état est forcément $\geq$ `STATE_CREATED` qui est égal à 0). Les opérations *a priori* de changements structurels peuvent être appliquées de façon interactive par le biais du graph visualizer. La suppression *a priori* d'un artefact ou d'une relation fait passer l'état de cet artefact ou relation à `STATE_DELETED_TEMP`.

Nous avons défini une règle pour chaque type d'opération de changement correspondant à chaque type d'artefact. Le résultat d'une règle est le marquage des nœuds et des arcs affectés par le changement. Le marquage change l'état d'un artefact ou d'une relation et permet de générer de nouveaux faits pouvant déclencher de nouvelles règles, etc.

Nous explicitons, prendre des exemples montrant le fonctionnement de certaines règles de propagation d'impact du changement.

1. Le listing 6.6 montre un exemple de règle de propagation de l'impact correspondant à la suppression temporaire d'une classe annotée. L'annotation est associée à la classe par la relation compose. La règle considère deux artefacts : une classe et une annotation. La clause *when* montre que la règle est applicable si l'annotation est dans l'état de suppression temporaire. Dans ce cas, la classe associée est marquée comme affectée, à son tour, dans le cas où elle ne l'est pas encore.

Listing 6.6: La règle stratégique pour la propagation d'impact de *annotation* à *class*

```
rule "EntityAnnotationPropagationClass"
 when
  annotNode: ArchitectNode (typeNode == "ANNOTATION", stateNode==states.STATE_DELETED_TEMP)
  classNode: ArchitectNode (typeNode == "CLASSE")
  edge: ArchitectEdge ( label=="compose", nodeSrc==classNode, nodeDest==annotNode)
 then
  System.out. println ("IMPACT PROPAGATION");
  if (edge.getState() < states.STATE_IMPACTED)
   edge. setState( states .STATE_IMPACTED);
  if (classNode.getStateNode() < states.STATE_IMPACTED)
   classNode.setStateNode(states .STATE_IMPACTED);
end
```

2. Le listing 6.7 montre la règle de propagation de l'impact de l'opération de changement d'une variable. Le changement de l'état d'une variable provoque le marquage de tous les artefacts qui sont liés à cette variable par un lien d'utilisation (use).

Listing 6.7: La règle stratégique pour la propagation d'impact de *variable* à *statement*

```
rule "VariablePropagationUse"
 when
  varNode: ArchitectNode (typeNode == "VARIABLE", stateNode>=states.STATE_CHANGED)
  statNode: ArchitectNode ()
  edge: ArchitectEdge ( label=="use", nodeSrc==statNode, nodeDest==varNode)
 then
  System.out. println ("IMPACT PROPAGATION " + statNode);
  if (edge.getState() < states.STATE_IMPACTED)
   edge. setState( states .STATE_IMPACTED);
  if (statNode.getStateNode() < states.STATE_IMPACTED)
   statNode.setStateNode(states .STATE_IMPACTED);
end
```

3. Le listing 6.8 montre la règle associée au changement de l'annotation de table ou de colonne. Cette règle stipule que s'il existe un nœud de type Table ou Colonne et s'il existe un nœud annotation avec une relation « ORMapped » entre le nœud annotation et le nœud table ou colonne alors la suppression de la table ou de la colonne affecte l'annotation.

Listing 6.8: La règle stratégique pour la propagation d'impact de *table* à *annotation*

```
rule "PropagationInverseORMapped"
 when
  aNode: ArchitectNode (stateNode>=states.STATE_DELETED_TEMP)
  eval (aNode.getTypeNode().equals("TABLE") || aNode.getTypeNode().equals("COLUMN"))
  bNode: ArchitectNode (typeNode == "ANNOTATION")
  edge: ArchitectEdge (nodeSrc==bNode, nodeDest==aNode)
  eval(edge.getLabel (). equals("ORMapped"))
 then
```

```
  System.out. println ("IMPACT PROPAGATION");
  if (edge.getState() <= states.STATE_ADDED)
   edge. setState ( states .STATE_IMPACTED);
  if (bNode.getStateNode() <= states.STATE_ADDED)
   bNode.setStateNode(states.STATE_IMPACTED);
end
```

4. Le listing 6.9 est un exemple de règle déterminant la propagation de l'impact de la suppression d'un nœud de type EDP (Event Data Port) d'une architecture AADL. Cela change l'état du composant indiquant qu'il est supprimé (`states.STATE_DELETED`). Cet état provoque le déclenchement de la règle de propagation de l'impact. Quand les conditions de la règle sont vérifiées, le moteur de règles exécute la partie action de la première règle. Cela change le label du nœud supprimé et indique que le composant est affecté par l'opération de suppression en changeant son état. La deuxième règle est responsable de la propagation de l'impact d changement si un élément de la base de connaissances est dans l'état impacté (`states.STATE_IMPACTED`). Elle change l'état de l'arc concerné en `STATE_IMPACTED`. Cela veut dire que ces arcs sont affectés par la suppression et provoque le marquage des nœuds connectés à travers ces arcs.

   Listing 6.9: La règle stratégique pour la propagation d'impact de *Event Data Port* à les nœudes liées

```
1  rule "ImpactDeletePort"
2   when
3    n : ArchitectNode (stateNode==states.STATE_DELETED, typeNode=="EDP")
4    n1 : ArchitectNode( )
5    e : ArchitectEdge()
6    eval((e.getNodeDest()==n && e.getNodeSrc()==n1)||
7         (e.getNodeSrc()==n && e.getNodeDest()==n1))
8   then
9    n.setLabel("DELETED "+n.getLabel());
10   n.setStateNode(states.STATE_IMPACTED);
11 end
12
13 rule "PropagerImpactDeletePort"
14  when
15   n : ArchitectNode (stateNode==states.STATE_IMPACTED, typeNode=="EDP")
16   n1 : ArchitectNode( )
17   e : ArchitectEdge()
18   eval((e.getNodeDest()==n && e.getNodeSrc()==n1)||
19        (e.getNodeSrc()==n && e.getNodeDest()==n1))
20  then
21   e. setState ( states .STATE_IMPACTED);
22   n1.setStateNode(states.STATE_IMPACTED);
23 end
```

5. Si les résultats de l'exécution d'une opération sur plusieurs artefacts sont similaires, nous pouvons généraliser et grouper des règles telle que schématisé dans les listings 6.10 et 6.11.

Listing 6.10: Exemple de plusieurs règles exécutant la même action sur plusieurs types de composant

```
1  rule "PropagerImpactDeleteEDPort"
2   when
3    n : ArchitectNode (stateNode==states.STATE_DELETED, typeNode=="EDP")
4    n1 : ArchitectNode( )
5    e : ArchitectEdge()
6    eval((e.getNodeDest()==n && e.getNodeSrc()==n1)||
7         (e.getNodeSrc()==n && e.getNodeDest()==n1))
8   then
9    n.setLabel("DELETED "+n.getLabel());
10   n.setStateNode(states.STATE_IMPACTED);
11 end
12
13 rule "PropagerImpactDeleteDPort"
14  when
15   n : ArchitectNode (stateNode==states.STATE_DELETED, typeNode=="DP")
16   n1 : ArchitectNode( )
17   e : ArchitectEdge()
18   eval((e.getNodeDest()==n && e.getNodeSrc()==n1)||
19        (e.getNodeSrc()==n && e.getNodeDest()==n1))
20  then
21   n.setLabel("DELETED "+n.getLabel());
22   n.setStateNode(states.STATE_IMPACTED);
23 end
24
25 rule "PropagerImpactDeleteEPort"
26  when
27   n : ArchitectNode (stateNode==states.STATE_DELETED, typeNode=="EP")
28   n1 : ArchitectNode( )
29   e : ArchitectEdge()
30   eval((e.getNodeDest()==n && e.getNodeSrc()==n1)||
31        (e.getNodeSrc()==n && e.getNodeDest()==n1))
32  then
33   n.setLabel("DELETED "+n.getLabel());
34   n.setStateNode(states.STATE_IMPACTED);
35 end
```

Dans le listing 6.10, nous définissons des règles décrivant l'opération de suppression d'un port. L'invariant de cette opération et de vérifier que il ne existe aucun arc du port supprimé. Les conditions sont les mêmes pour tous les trois type de port DP, EDP, et EP. Le résultat de l'exécution de la règle de propagation de l'impact est le marquage des nœuds concernés. Nous pouvons donc combiner ces règles (listing 6.11).

Listing 6.11: Règle obtenue en combinant les différentes règles du listing 6.10

```
1  rule "PropagerImpactDeletePort"
2   when
3    n : ArchitectNode (stateNode==states.STATE_DELETED)
4    eval(n.getTypeNode().equalsIgnoreCase("DP") ||
5    n.getTypeNode().equalsIgnoreCase("EP") ||
6    n.getTypeNode().equalsIgnoreCase("EDP")
7        )
8    n1 : ArchitectNode(  )
9    e : ArchitectEdge()
10   eval((e.getNodeDest()==n && e.getNodeSrc()==n1)||
11        (e.getNodeSrc()==n && e.getNodeDest()==n1))
12  then
13   n.setLabel("DELETED "+n.getLabel());
14   n.setStateNode(states.STATE_IMPACTED);
15  end
```

6.3.2.3 Les règles expertes

Le modèle SMSE peut être utilisé pour générer une partie importante des règles d'inférence et de propagation de l'impact. Il peut être nécessaire de disposer d'une petite aide des experts pour créer l'ensemble des règles. Les développeurs des applications peuvent avoir la liberté de définir des règles pour une gestion plus précise et/ou plus spécifique de l'impact du changement. Ils peuvent éditer des règles définies par l'utilisateur. Le système *Drools* (ou JBoss Rules), qui est un moteur de règles métier basé sur l'algorithme Rete, supporte le développement de systèmes expert embarqués dans des applications Java. Un éditeur de règles pour *Drools* existe en forme de plug-in Eclipse. Il est possible, à travers cet éditeur, d'intégrer des objets, méthodes et attributs d'objets Java dans la définition des règles. Il est également possible d'utiliser des fonctions propres à *Drools* comme la fonction update qui permet de modifier l'état d'un objet, etc. La plate-forme *Architect* inclut une interface pour la création et l'édition de règles expertes que l'on peut ajouter à Drools par le biais du fichier *Drools* rule resource editor file ou *Drools guided rules* resource editor file (Figure 6.4).

Dans les sections suivantes, nous proposons un scénario illustrant les représentations à base de graphes dans *Architect* et le processus de propagation de l'impact.

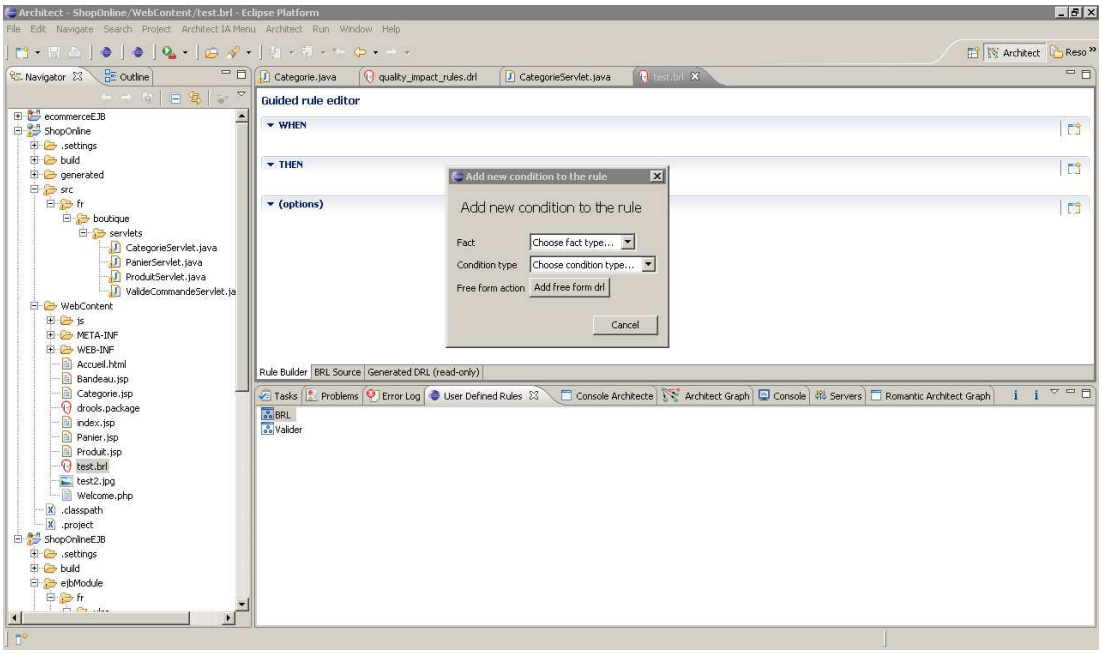


Figure 6.4: Drools guided rule resource editor

6.4 Scénario de propagation de l'impact

Le plug-in *Eclipse graph visualizer* fournit des facilités pour la définition des opérations de changement des nœuds du graphe des artefacts logiciels. Il fournit également une interface pour l'ajout de règles définies par l'utilisateur de façon dynamique.

Le processus de visualisation de graphes transforme la base de faits en un graphe. L'édition visuelle de ces graphes permet à l'ingénieur de percevoir certaines caractéristiques cachées des données pouvant être utile pour l'exploration et l'analyse du logiciel. Elle peut également aider à la compréhension de systèmes logiciels complexes. L'interface utilisateur de *Architect* permet donc une visualisation des graphes d'artefacts logiciels et permet une gestion simplifiée des changements. Architect permet également de répercuter directement les changements sur le code source et la visualisation en temps réel de warnings pou avertissement et de tooltips pouvant aider le développeur dans l'accomplissement des opérations de changement.

Comme expliqué plus haut, l'interface utilisateur du système Architect a été mise en œuvre en utilisant la librairie Java JUNG supportant diverses représentations de graphes telles que les graphes orientés et attribués, les sous-graphes, les hyper-graphes, etc.

ArchitectGraph permet la représentation du sous-graphe associé à une partie d'un code source sélectionné dans l'atelier Eclipse. Il est bien entendu de développer la visualisation du graphe en choisissant de visualiser plus de détails ou l'un des sous-graphes connexes au graphe visualisé. Cela permet de réduire la complexité du graphe à visualiser et de l'explorer de proche en proche.

Les opérations de changement structurel peuvent être déclenchées par sélection d'artefacts dans le code source ou par un menu contextuel associé aux nœuds et arcs du graphe. Les types de modifications ou changements proposés dépendent du type du nœud ou de l'arc sélectionnés. La sélection d'une opération de changement sur un nœud spécifique conduit au déclenchement de la propagation de l'impact sur les autres nœuds et arcs du graphe si les invariants de l'opération sont violés. Les nœuds et arcs affectés par le changement sont identifiés par un label ou étiquette indiquant le changement ainsi que par un changement au niveau de la couleur de visualisation.

Un scénario typique de l'utilisation concerne la gestion du changement d'une application web dynamique mutli-tiers. C'est une application développée en utilisant le pattern MVC (*Model View Controller*) qui fait appel aux pages JSP (*Java Server Pages*) comme moyen de réaliser la vue, une combinaison de Servlets et de EJB (Enterprise Java Beans) comme contrôleurs et des Java Beans comme modèle ou données. Comme discuté dans le chapitre 2, la représentation de la méta-structure d'une application Java J2EE peut être schématisée par la figure 2.7. Les champs de la base de données utilisée par l'application peuvent être annotés dans les classes implémentant les Java Beans. Pour commencer l'analyse *a priori* de l'impact du changement, nous exécutons l'opération d'exécution temporaire d'un champ de la base de données qui est annoté dans un Java Bean.

Dans ce scénario, nous analysons une application de e-commerce qui est un site web dynamique. Le site web a un classe *Client* (figure 6.5). Cette classe est un Java Entity Bean, qui est « mappé » à une table *tclient* de la base de données *bd_commerce*. Le code source SQL de la table *tclient* est montré par la figure 6.6. Pour les besoins de la démonstration, nous sélectionnons le champ *pk_tclient* dans le code source et l'opération de suppression (disponible en cliquant sur le bouton droit de la souris : figure 6.7). Comme le champ *pk_tclient* est annoté dans la classe *Client*, la suppression du champ provoque l'insertion de cette annotation dans le graphe visualisé comme artefact affecté par la suppression (figure 6.8). La figure 6.9 montre la propagation de l'impact

au niveau du Java Bean Entity Client. Il est également possible de montrer tous les impacts de cette opération dans le reste du code source (figure 6.10).

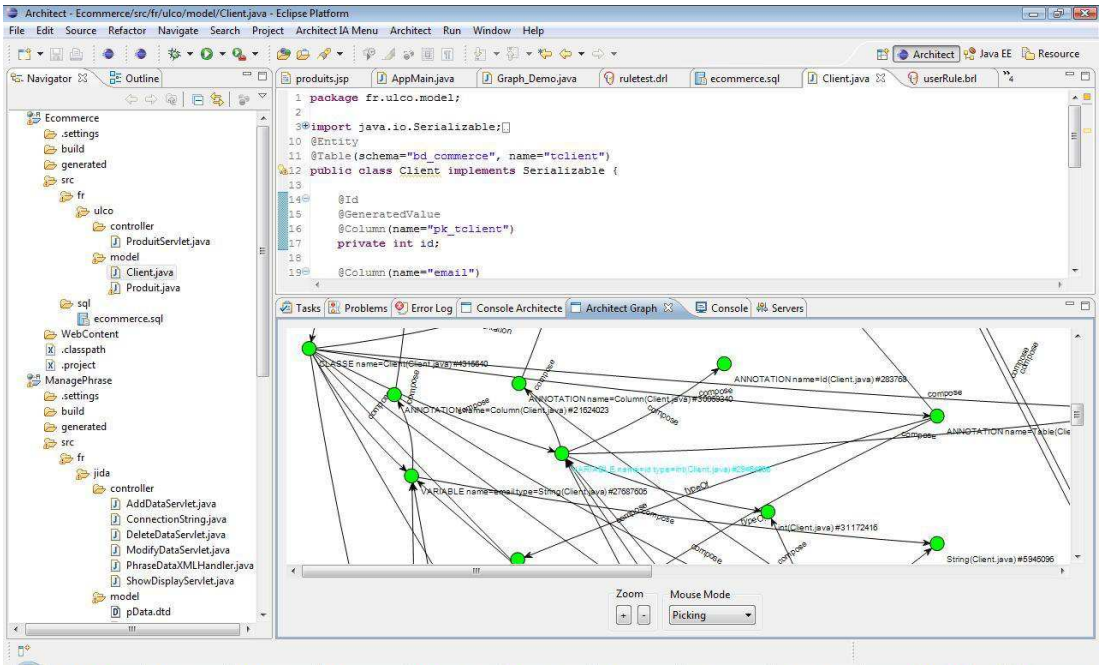


Figure 6.5: J2EE Application (Graphe du code JAVA)

6.5 Analyse de la propagation de l'impact qualitatif du changement

Un changement (Δ) sur un artefact peut provoquer la propagation d'un impact structurel, qualitatif, fonctionnel, logique et/ou comportemental aux artefacts qui lui sont reliés. Nous considérons dans ce qui suit les cas des impacts structurels et qualitatifs. Pour l'aspect qualitatif il est important de déterminer le taux avec lequel la qualité d'un artefact est affectée par le changement opéré sur un autre artefact qui lui est relié. La création de règles de propagation d'impact du changement au niveau de l'aspect qualitatif est basé sur une connaissance des relations incluant notamment la conductivité de l'impact et les variations des valeurs de métriques, etc.

Les changements structurels peuvent affecter la qualité du logiciel. Supposant que $\Delta_S \Rightarrow \Delta_Q$ dénote le fait qu'un changement structurel Δ_S implique un changement qualitatif Δ_Q. Pour illustrer l'impact d'un changement structurel sur le modèle de la

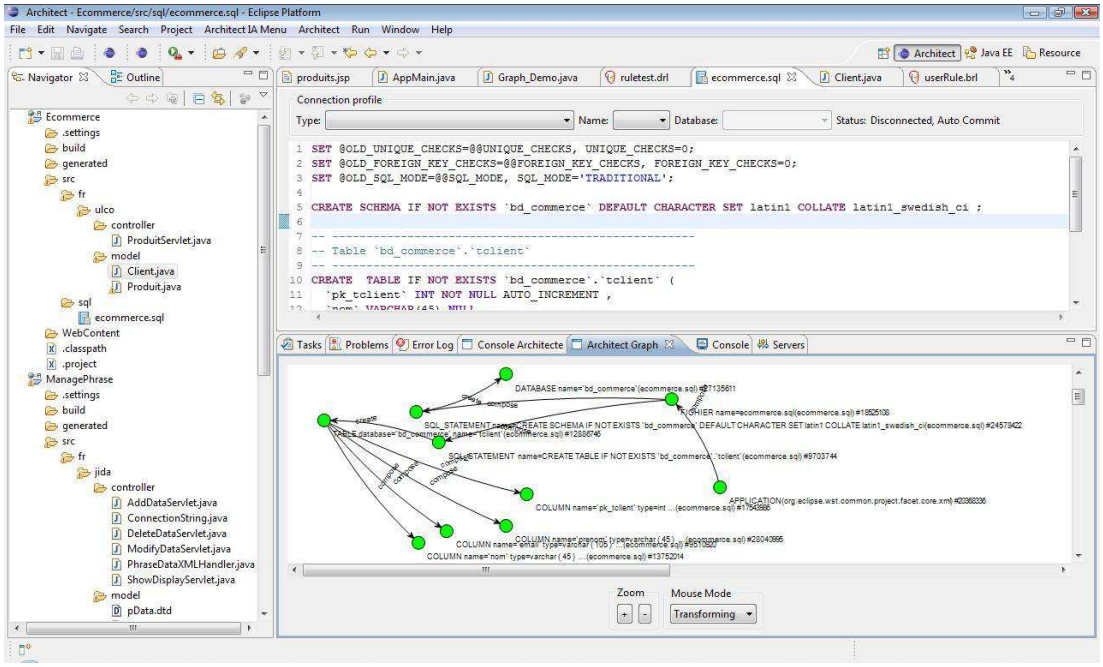


FIGURE 6.6: J2EE Application (Graphe de la base de données)

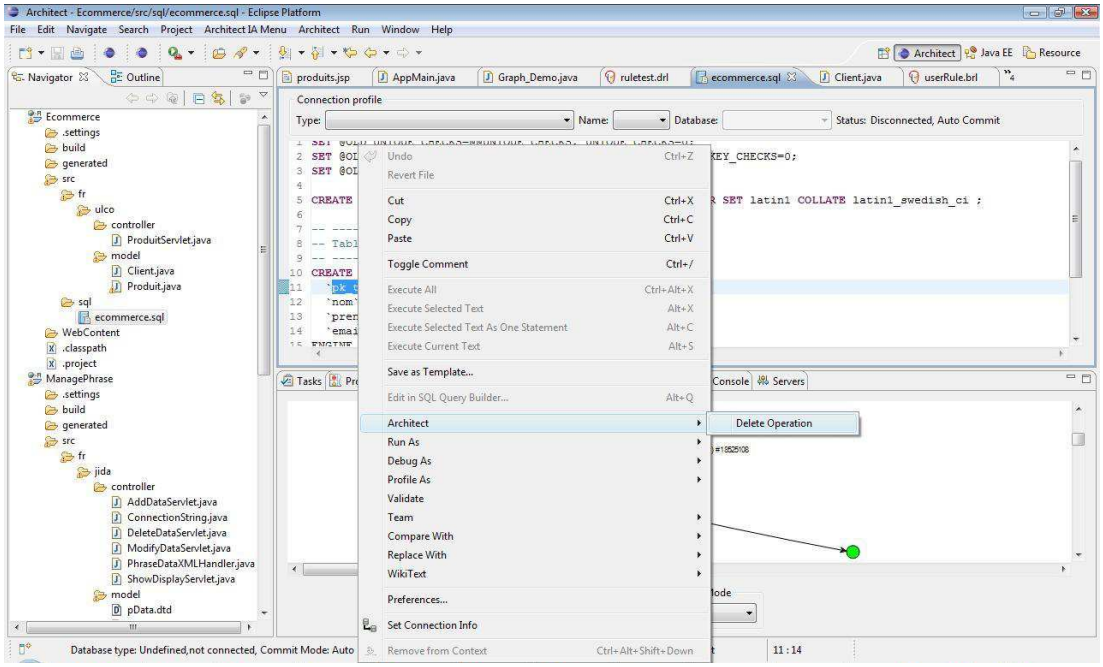


FIGURE 6.7: Une Analyse *a priori* : exécution de l'operation delete

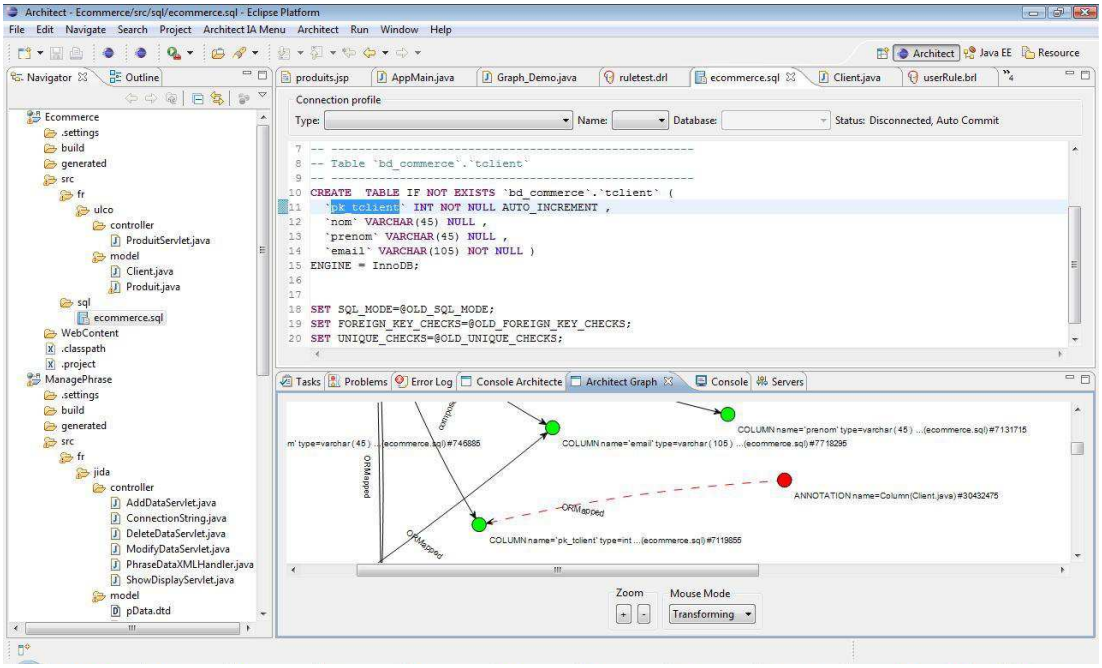


Figure 6.8: Traçabilité de la propagation de l'impact du changement au niveau primaire

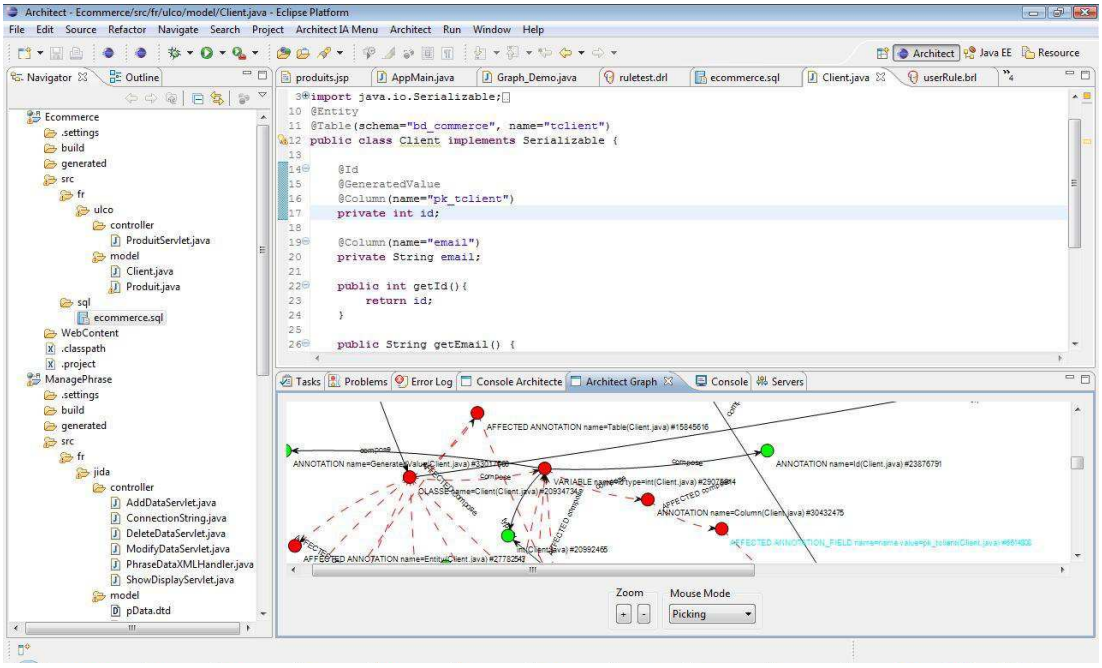


Figure 6.9: Traçabilité au niveau profond

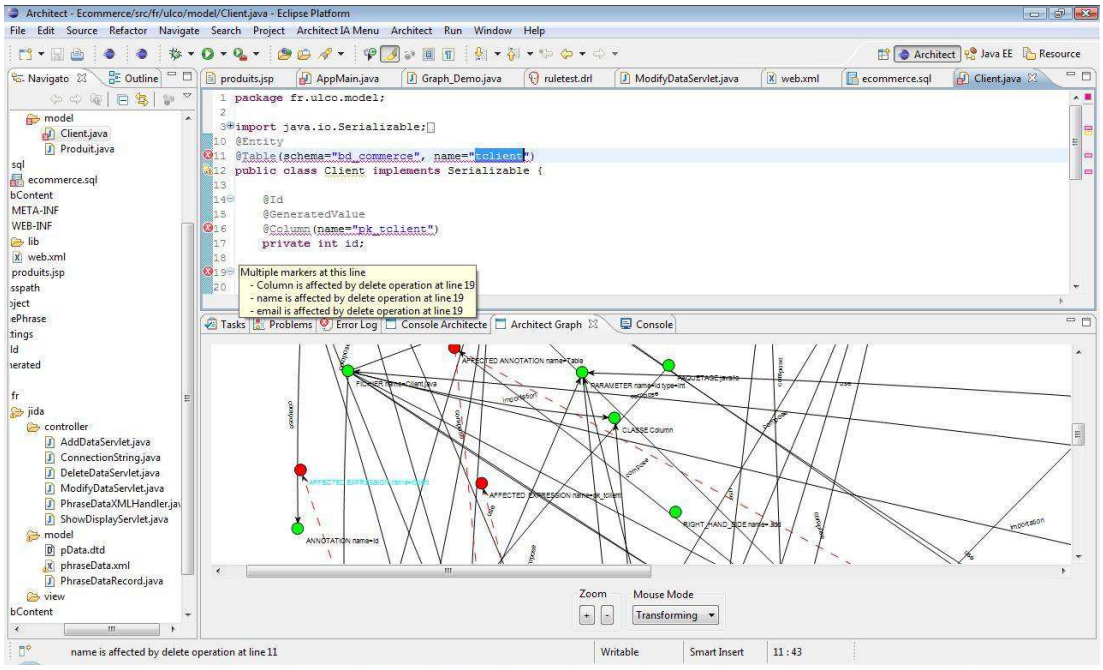


Figure 6.10: Traçabilité de la propagation de l'impact du changement au niveau du code source

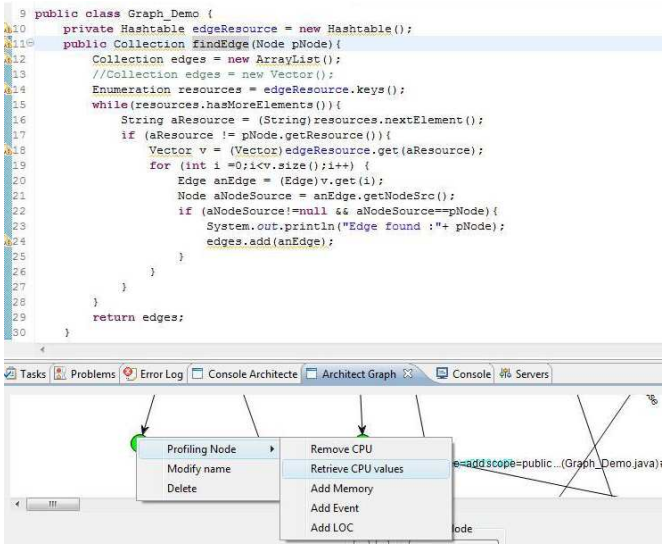


Figure 6.11: Le code Java (Snippet - I)

qualité, nous allons utiliser un exemple (celui du code de la figure 6.11). Pour expliquer la propagation de l'impact qualitatif, nous mesurons et observons les performances de la méthode `findEdge` de la classe `Graph_Demo`. Nous pouvons déterminer le changement qualitatif Δ_Q du au changement structurel ayant affecté `edges` de type `Collection`. Nous invoquons un changement de `edges` en l'initialisant par une instance de `Vector` au lieu de celle de `ArrayList`. Nous observons les performances de `findEdge` à travers 5 exécutions avant et après la modification de `edges`. La figure 6.12 montre le graphique des performance avec `ArrayList` et la figure 6.13 celui associé à `Vector`.

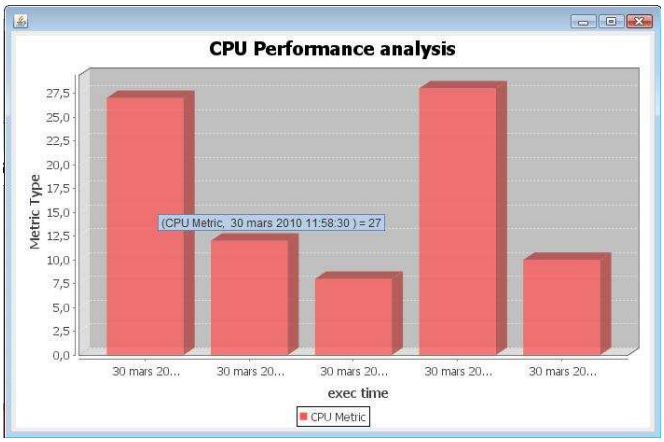


FIGURE 6.12: La graphe de la performance du code d'exemple (avant la modification structurelle)

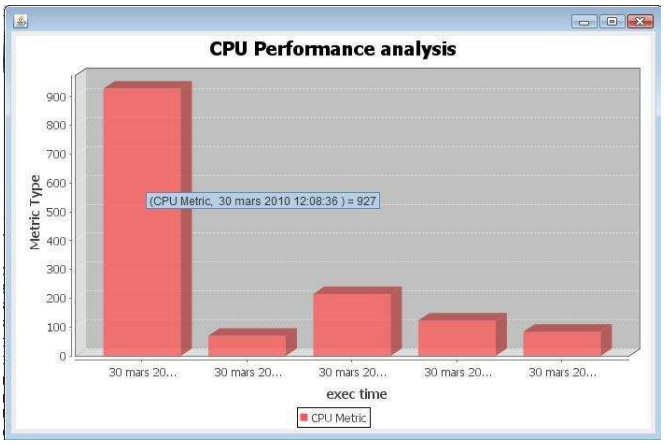


FIGURE 6.13: La graphe de la performance du code d'exemple (après la modification structurelle)

Pour observer l'impact qualitatif du changement, nous notifions que `findEdge` est appelée dans la méthode `main` de la classe `AppMain` (figure 6.14).

Comme discuté plus haut, tout changement structurel (Δ_S) invoqué sur un artefact peut causer un impact relatif (Δ_Q) sur un critère de la qualité. L'impact de Δ_Q peut être propagé progressivement aux critères précédents (dans le sens arbre de la qualité) et aux facteurs correspondants, etc. L'exemple que nous venons de décrire a montrer l'existence d'un impact qualitatif en matière de variation des performances d'une méthode suite à un changement structurel alors que ce changement n'avait pas provoqué d'impact structurel.

```
1  package fr.ulco.app;
2
3  import fr.ulco.model.Edge;
4  import fr.ulco.model.Graph_Demo;
5  import fr.ulco.model.Node;
6
7  public class AppMain {
8
9      public static void main(String[] args) {
10         Graph_Demo demo = new Graph_Demo();
11         Node src2 = new Node("node0");
12         for(int i=0; i<10000; i++){
13             Node src1 = new Node("node"+i+1);
14             Edge edge1 = new Edge(src1, src2);
15             demo.add("node"+i, edge1);
16         }
17         demo.findEdge(src2);
18     }
19 }
20
```

Collection fr.ulco.model.Graph_Demo.findEdge(Node pNode)

Press 'F2' for focus

Figure 6.14: Le code Java (Snippet - II)

6.6 Conclusion

Nous avons décrit, dans ce chapitre, la conception et les fonctionnalités de la plate-forme que nous avons développée et déployée pour la validation de notre approche d'analyse et de propagation e l'impact du changement des artefacts logiciels. Cette plate-forme a été développé en forme de plug-ins Eclipse pour une meilleure utilisation auprès de la communauté des développeurs et un meilleur retour d'expérience. Dans ce chapitre nous avons d'abord décrit l'architecture globale de la plate-forme qui est constituée de quatre principaux plug-ins Eclipse. Ce sont les plug-ins multi-lingual parser, software modeler, graph visualizer plug-in, et change propagation analyser. Nous avons donc présenté les fonctionnalités de chacun de ces plug-ins et par la suite nous

avons particulièrement décrit le change propagation analyser en explicitant sa structure en termes de base de faits et en décrivant particulièrement les trois catégories des règles formant sa base de règles. Ce sont notamment les règles d'inférence qui identifient les relations complexes, les règles stratégiques qui implémente le processus de propagation de l'impact et les règles expertes qui permettent à l'utilisateur de définir des règles spécifiques ou ad hoc.

Finalement nous avons montré, à travers des exemples, l'utilisation de la plate-forme pour la propagation de l'impact du changement des artefacts logiciels au niveau structurel et au niveau qualitatif. Les expérimentations que nous avons effectuées à l'aie de la plate-forme Architect ont particulièrement concerné les applications distribuées multi-tiers Java J2EE et leurs descriptions architecturales à l'aide du langage AADL. Nous avons en effet expérimenté la propagation de l'impact du changement d'une description architecturale sur les codes sources et inversement.

Nos travaux d'implémentation continuent et nous sommes entrain d'utiliser Architect pour l'analyse et la propagation du changement des modèles de processus métiers (BPM) sur les applications logiciels qui les implémentent et vice versa.

Cinquième partie

Conclusions

Chapitre 7

Conclusions and Perspectives

7.1 Conclusions

Ce mémoire a d'abord dressé l'état de l'art des approches et techniques les plus largement connues dans le domaine de l'évolution des logiciels. Pour une meilleure compréhension du logiciel et une meilleure analyse de l'impact de sa modification, nous avons adopté une modélisation reflétant les niveaux d'abstraction de description des artefacts logiciels. Une présentation d'abord générale de l'approche modélisation des logiciels pour le contrôle d'évolution a été donnée pour servir de référence au contenu de la base de connaissances élaborée pour l'aide à la compréhension et à l'analyse d'impact des modifications.

Cette approche de la modélisation intégrée du logiciel a été ensuite argumentée dans sa constitution incluant deux points conceptuels majeurs : le point de vue structurel et le point de vue qualitatif. Nous avons alors présenté un système intégré de modélisation basée sur la structure et analyse qualitative de l'évolution des logiciels. Elle a alors conduit à concevoir un modèle structurel d'évolution de logiciels (SMSE) et un modèle qualitatif pour l'évolution de logiciel (QMSE). Une mise en correspondance entre les deux modèles a été ensuite définie.

Pour la modélisation structurelle, une classification exhaustive des artefacts logiciels et de leurs relations d'interdépendance a été donnée. Cette classification stratifie les composants en niveaux d'abstraction en définissant ensuite les types de relations conducteurs de l'impact du changement au sein de modèle structurel pour l'évolution de logiciels.

Ce modèle a été ensuite détaillé et instancié pour des applications distribuées en identifiant les interdépendances par mise en correspondance d'abord entre artefacts issus de différentes phases de développement puis entre niveaux d'abstraction et niveaux granulaires. Le modèle ainsi élaboré permet une analyse *a priori* de la propagation d'impact du changement entre artefacts logiciels.

L'instanciation de SMSE a été détaillé pour la phase de codage puis illustrée par des exemples d'analyse structurelle d'applications distribuées développées en J2EE. Le modèle SMSE a été ensuite appliquée sur l'architecture logicielle décrite par un ADL, l'illustration a été ensuite donnée sur des architectures logicielles décrites dans un langage architectural largement utilisé l'AADL (Architecture Analysis and Design Language). Une mise en correspondance a été ensuite donnée entre les éléments architecturaux décrites par AADL et leur représentation en SMSE. Afin de valider l'applicabilité de notre approche, une analyse de modification d'un élément architectural du logiciel et son impact sur le code source d'implémentation a été donnée. L'approche présentée a ainsi appliqué la modélisation sur l'architecture logicielle et le niveau du code. Il a montré sa validité pour formaliser les éléments de l'architecture logicielle, de leur relations d'interdépendance puis leurs correspondances avec les codes source d'implémentation permettant ainsi d'analyser l'impact de la propagation d'un changement souhaité. Les modèles obtenus sont génériques et indépendants du code source et des langages de description architecturale.

Pour une évolutivité des connaissances requises pour une analyse exhaustive d'impact des modifications, nous avons adopté un développement de système à base de connaissances dont la base des faits et la base des règles ont été formulées en référence à la modélisation adoptée.

Les deux modèles SMSE et QMSE instanciés pour une application permettent de structurer le contenu d'un système de base de connaissances ainsi que le stockage et l'analyse du changement moyennant les faits et les règles concernés. Les formulations développées dans le chapitre 2 et 3 fournissent la base conceptuelle pour le développement des faits de base tandis que les formulations expliquées dans le chapitre 4 présente les concepts fondamentaux pour construire la base règles. Le chapitre 5 présente les données descriptives de l'aspect qualitatif et qui permettent de raisonner sur l'évolution qualitative.

Le processus de la propagation de l'impact du changement a été présenté en détail en désignant les relations qui véhiculent la propagation, d'impact du changement structurel et architecturale, et en identifiant les paramètres qui agissent sur l'importance de l'impact du changement. Les règles de la propagation d'impact du changement ont été formulées pour différents types de relations.

Nous avons prototypé une plateforme développée conformément à la modélisation proposée pour permettre, à la demande de l'utilisateur, de construire des vues spécifiques par des graphes interactifs qui permettent de visualiser les éléments du logiciel affectés par l'opération d'une modification à un constituant du logiciel. La validation de l'approche la proposée a fait l'objet du chapitre 6.

Notre approche étant destinée à intégrer des modélisations partielles des points de vue structurel, qualitatif, fonctionnel ou comportemental, dans ce mémoire, l'illustration a été principalement structurelle avec une prise en compte des cas de description architecturale de certaines applications distribuées conçues moyennant un ADL. En dépit de l'utilisation d'AADL pour des raisons d'illustration, l'instanciation du modèle peut se faire pour une description architecturale utilisant tout autre ADL.

La typologie des impacts de modification est pertinente de plusieurs cas d'évolution ayant des objectifs spécifiques d'amélioration. De point de vue validation un développement à base des connaissances permet une traçabilité des conclusions obtenues et une meilleure évolutivité des connaissances impliquées dans le raisonnement sur l'évolution. La base des règles peut être régulièrement enrichie par des nouvelles règles rajoutées par le chargé de l'évolution lors d'une expérience de changement dans un contexte logiciel impliquant un type de changement appliqué sur un type de composant pour aboutir à une certaine amélioration. La modélisation adoptée, indépendante d'un langage particulier de programmation ou d'un SGBD, elle permet de raisonner sur les constituants logiciels quelle que soit leur granularité.

7.2 Perspectives

Dans la modélisation proposée, nous avons souligné l'importance de considérer l'évolution de différents points de vue conceptuels d'où une perspective principale de rendre la modélisation actuelle plus exhaustive en tenant compte des points de vue fonctionnel et comportemental en plus deux points de vue structurel et qualitatif déjà abordés.

Afin de rendre cette modélisation utile à un éventail plus large d'applications, il est également important d'aborder par le contrôle de l'évolution la mobilité dans ses différentes dimensions incluant celle du code, des terminaux, des sessions, etc. .

Les étapes ou pas du raisonnement dans l'analyse d'impact des modifications nécessitent des vues ou des analyses sur les artefacts logiciels dont le volume des connaissances concernées varie remarquablement. Afin de rendre l'analyse plus adaptée à différents cas du raisonnement, il devient plus efficace de pouvoir réaliser une analyse d'impact guidée dans ses détails par le chargé de l'évolution ou de l'expert qualité. D'où l'importance d'exprimer les besoins d'analyse et d'observation non seulement à travers des items des menus déroulants mais également par des requêtes d'analyse ou d'observation, simples ou composées. Cela revient ainsi à activer des analyses partielles d'impact, précisées par les requêtes du chargé de l'évolution, nécessitant de la base des connaissances un sous-ensemble de règles et un sous-ensemble de faits.

Par ailleurs, il faut examiner les techniques d'analyse partielle, qui peut cibler un ensemble bien spécifique des connaissances descriptives du logiciel, ou une partie du logiciel, d'un point de vue conceptuel.

De la même manière, l'analyse incrémentale est une perspective consistant à prendre en compte des effets, bien déterminés, de la dernière modification d'une application pour actualiser les connaissances concernées et surtout éviter de recommencer une analyse globale souvent longue et coûteuse.

Sixième partie

Bibliographie et Annexes

Bibliographie

[1] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. 4

[2] Marco D'Ambros and Michele Lanza. **Reverse Engineering with Logical Coupling**. In *WCRE '06 : Proceedings of the 13th Working Conference on Reverse Engineering*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society. 4

[3] Mircea Lungu, Michele Lanza, Tudor Girba, and Reinout Heeck. **Reverse Engineering Super-Repositories**. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 120–129, 2007. 4

[4] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. **Mining sequences of changed-files from version histories**. In *MSR '06 : Proceedings of the 2006 international workshop on Mining software repositories*, pages 47–53, New York, NY, USA, 2006. ACM. 4

[5] Filip Van Rysselberghe and Serge Demeyer. **Studying Software Evolution Information by Visualizing the Change History**. In *ICSM '04 : Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 328–337, Washington, DC, USA, 2004. IEEE Computer Society. 4

[6] Adeel Ahmad and Henri Basson. **Software evolution modelling : an approach for change impact analysis**. In *Proceedings of the 7th International Conference on Frontiers of Information Technology*, FIT '09, pages 56 :1–56 :4, New York, NY, USA, 2009. ACM. 4

[7] Adeel Ahmad, Henri Basson, Laurent Deruelle, and Mourad Bouneffa. **A knowledge-based framework for software evolution control**. In *INFORSID'09 : 28th Conference on Informatique des organisation et systèmes d'information et de décision*, pages 286–291, Toulouse, France, May 2009. IRIT Press (www.irit.fr). 4, 20, 28, 85, 118

[8] Adeel Ahmad, Henri Basson, and Mourad Bouneffa. **Rule-Based Approach for Software Evolution Management**. In *IEEE APSSC 2009 : IEEE Asia-Pacific Services Computing Conference*, December 2009. 4, 118

[9] Adeel Ahmad, Henri Basson, and Mourad Bouneffa. **Software Evolution Control : Towards a better identification of change impact propagation**. In *ICET'08 : Proceedings of the 4th IEEE International Conference on Emerging Technologies*, pages 286–291. IEEE Computer Society, October 2008. 5, 20, 116

[10] Adeel Ahmad, Henri Basson, Laurent Deruelle, and Mourad Bouneffa. **Towards a better control of Change Impact Propagation**. In *INMIC'08 : 12th IEEE International Multitopic Conference*, pages 286–291. IEEE Computer Society, December 2008. 5, 20, 41

[11] M.O. Hassan, Laurent Deruelle, Henri Basson, and Adeel Ahmad. **A Change Propagation Process For Distributed Software Architecture**. In *ENASE 2010 : 5th International Conference on Evaluation of Novel Approaches to Software Engineering*, Athens, Greece, 22 - 24 July 2010. 5

[12] P. Naur and B. Randell. *Software Engineering*. NATO, Scientific Affairs Division, Brussels, 1969. 7

[13] Walker W. Royce. **Managing the development of large software systems : concepts and techniques**. *Proc. IEEE WESTCON, Los Angeles*, pages 1–9, August 1970. Reprinted in *Proceedings* of the 9th International Conference on Software Engineering, March 1987, pp. 328–338. 7

[14] Meir M. Lehman. **Programs, life cycles, and laws of software evolution**. *Proc. IEEE*, **68**(9) :1060–1076, September 1980. 8

[15] M. M. Lehman and L. A. Belady, editors. *Program evolution : processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. 8

[16] L. A. Belady and M. M. Lehman. **A model of large program development**. *IBM Syst. J.*, **15** :225–252, September 1976. 8

[17] S. S. Yau, J. S. Colofello, and T. MacGregor. **Ripple Effect Analysis of Software Maintenance**. In *Proc. Int'l Computer Software and Applications Conf. (COMPSAC)*, pages 60–65. IEEE Computer Society Press, 1978. 8

[18] Lowell Jay Arthur. *Software evolution : the software maintenance challenge*. Wiley-Interscience, New York, NY, USA, 1988. 8

[19] Magnus C. Ohlsson, Anneliese Von Mayrhauser, Brian McGuire, and Claes Wohlin. **Code Decay Analysis of Legacy Software through Successive Releases**. In *Proceedings of the IEEE Aerospace Conference*, pages 69–81, 1999. 8

[20] Tom Gilb. **Evolutionary development**. *SIGSOFT Softw. Eng. Notes*, **6** :17–17, April 1981. 8

[21] Tom Gilb. *Principles of software engineering management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. 8

[22] Barry W. Boehm. **A Spiral Model of Software Development and Enhancement**. *Computer*, **21** :61–72, May 1988. 8

[23] Keith H. Bennett and Václav T. Rajlich. **Software maintenance and evolution : a roadmap**. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM. 8

[24] Juan C. Nogueira, Jones Carl, and Luqi. **Surfing the Edge of Chaos : Applications to Software Engineering**. *Command and Control Research and Technology Symposium*, June 2000. 8

[25] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2001. 9

[26] Robert Cecil Martin. *Agile Software Development : Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. 9

[27] Kent Beck and Cynthia Andres. *Extreme Programming Explained : Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. 9

[28] Craig Larman and Victor R. Basili. **Iterative and incremental development : A brief history**. *IEEE Computer*, **36**(6) :47–56, June 2003. 9

[29] E. Iee. *Standard IEEE Std 1219-1999 on Software Maintenance*, **2**. IEEE Press, 1999. 10

[30] International Standards Organisation (ISO). *Standard on Software Engineering - Software Maintenance*. ISO/IEC, 1999. 10

[31] Pierre Bourque, François Robert, Jean-Marc Lavoie, Ansik Lee, Sylvie Trudel, and Timothy C. Lethbridge. **Guide to the Software Engineering Body of Knowledge (SWEBOK) and the Software Engineering Education Knowledge (SEEK) - A Preliminary Mapping**. In *Proceedings of the 10th International Workshop on Software Technology and Engineering Practice*, STEP '02, pages 8–, Washington, DC, USA, 2002. IEEE Computer Society. 10

[32] E.J. Byrne. **A conceptual foundation for software re-engineering**. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 226 –235, nov 1992. 16, 17

[33] Elliot J. Chikofsky and James H. Cross II. **Reverse Engineering and Design Recovery : A Taxonomy**. *IEEE Softw.*, **7** :13–17, January 1990. 18

[34] Bennet P. Lientz and E. Burton. Swanson. *Software maintenance management : a study of the maintenance of computer application software in 487 data processing organizations / Bennet P. Lientz, E. Burton Swanson*. Addison-Wesley, Reading, Ma. :, 1980. 18

[35] IEEE ISO. *International Standard - ISO/IEC 14764 IEEE Std 14764-2006 - Software Engineering - Software Life Cycle Processes - Maintenance*. IEEE, 2 edition, September 2006. 18, 19

[36] Ned Chapin, Joanne E. Hale, Juan Fernandez-Ramil, and Wui-Gee Tan. **Types of software evolution and software maintenance**. *Journal of Software Maintenance and Evolution : Research and Practice*, **13**(1) :3–30, 2001. 19

[37] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. **Towards a taxonomy of software change : Research Articles**. *J. Softw. Maint. Evol.*, **17** :309–332, September 2005. 19

[38] Laurent Deruelle, Mourad Bouneffa, N. Melab, and Henri Basson. **Analysis and Manipulation of Distributed Multi-Language Software Code**. In *IEEE Intl. Workshop on Source Code Analysis and Manipulation (IEEE-SCAM'01)*, November 2001. 20

[39] W. Yuan, X. Chen, T. Xie, H. Mei, and F. Yang. **C++ Program Information Database for Analysis Tools**. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '98, pages 173–, Washington, DC, USA, 1998. IEEE Computer Society. 22

[40] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. **The C Information Abstraction System**. *IEEE Trans. Softw. Eng.*, **16** :325–334, March 1990. 22

[41] Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. **GXL : a graph-based standard exchange format for reengineering**. *Sci. Comput. Program.*, **60** :149–170, April 2006. 23, 149

[42] OMG. *XML Metadata Interchange (XMI ?)*. OMG, 2007. 23

[43] Tiago L. Alves, Jurriaan Hage, and Peter Rademaker. **Comparative study of code query technologies.** 23

[44] Jürgen Ebert and Daniel Bildhauer. **Reverse Engineering Using Graph Queries.** In *Principle Advancements in Database Management Technologies'10*, pages 335–362, 2010. 24

[45] Daniel Bildhauer and Jürgen Ebert. **Querying Software Abstraction Graphs**. In *Working on Query Technologies and Applications for Program Comprehension*, 2008. 24

[46] Jürgen Ebert and Angelika Franzke. **A Declarative Approach to Graph Based Modeling**. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '94, pages 38–50, London, UK, 1995. Springer-Verlag. 24

[47] Tassilo Horn and Jürgen Ebert. **The GReTL Transformational Language**. 2009. 24

[48] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. **Visualizing and querying software structures**. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 23–41, New York, NY, USA, 2010. ACM. 24

[49] Tian-Li Yu, David E. Goldberg, Kumara Sastry, Claudio F. Lima, and Martin Pelikan. **Dependency structure matrix, genetic algorithms, and effective recombination**. *Evol. Comput.*, **17** :595–626, December 2009. 25

[50] Bixin Li. **Managing Dependencies in Component-Based Systems Based on Matrix Model**. In *Proc. Of Net.Object.Days 2003*, September 2003. 25

[51] Renbin Xiao and Tinggui Chen. **Research on design structure matrix and its applications in product development and innovation: an overview**. *Int. J. Comput. Appl. Technol.*, **37** :218–229, March 2010. 25

[52] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. **Software Reflexion Models : Bridging the Gap between Design and Implementation**. *IEEE Trans. Softw. Eng.*, **27** :364–380, April 2001. 25

[53] Olivier Le Goaer, Dalila Tamzalit, and Mourad Oussalah. **Evolution Styles to Capitalize Evolution Expertise within Software Architectures**. In *SEKE*, pages 159–164, 2010. 26

[54] Olivier Le Goaer, Mourad Oussalah, and Dalila Tamzalit. **Reusing Evolution Practices onto Object-Oriented Designs : an Experiment with Evolution Styles**. In *SEDE*, pages 134–139, 2010. 26

[55] Vaclav Rajlich. **A Model for Change Propagation Based on Graph Rewriting**. In *Proceedings of the International Conference on Software Maintenance*, pages 84–91, Washington, DC, USA, 1997. IEEE Computer Society. 26, 27

[56] Peter Clark, John A. Thompson, and Bruce W. Porter. **Knowledge Patterns**. pages 191–208, 2004. 27

[57] S. K. Abd-El-Hafiz. **Evaluation of a knowledge-based approach to program understanding**. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, pages 259–, Washington, DC, USA, 1996. IEEE Computer Society. 27

[58] Alex Quilici. **A memory-based approach to recognizing programming plans**. *Commun. ACM*, **37** :84–93, May 1994. 27

[59] Rainer Koschke. **Software Engineering**. pages 140–173, 2009. 28

[60] Alexander S. Yeh, David R. Harris, and Melissa P. Chase. **Manipulating recovered software architecture views**. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 184–194, New York, NY, USA, 1997. ACM. 28

[61] Olga Kouchnarenko and Arnaud Lanoix. **Refinement and Verification of Synchronized Component-Based Systems**. **2805** :341–358, 2003. 28

[62] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Vaclav Rajlich. **JRipples : A Tool for Program Comprehension during Incremental Change**. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 149–152, Washington, DC, USA, 2005. IEEE Computer Society. 28

[63] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. **Chianti : a tool for change impact analysis of java programs**. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM. 28

[64] Kunrong Chen and Václav Rajlich. **Case Study of Feature Location Using Dependence Graph, after 10 Years**. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 1–3, Washington, DC, USA, 2010. IEEE Computer Society. 28

[65] Václav Rajlich. **Modeling software evolution by evolving interoperation graphs**. *Ann. Softw. Eng.*, **9** :235–248, January 2000. 28

[66] Henri Basson. **Contôle de l'évolution des logiciels : modélisation pour l'analyse d'impact des modifications**. 30, 118

[67] Sue E Black. **Computing ripple effect for software maintenance**. *Journal of Software Maintenance and Evolution Research and Practice*, **13**(4) :263–279, 2001. 36

[68] Roger S. Pressman. *Software Engineering : A Practitioner's Approach, 7th edition*. McGraw Hill Higher Education, January 2009. 37, 119

[69] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. 64

[70] P. Kruchten, H. Obbink, and J. A. Stafford. **The Past, Present, and Future for Software Architecture**. *IEEE Softw.*, **23**(2) :22–30, 2006. 64

[71] Nenad Medvidovic and Richard N. Taylor. **A Classification and Comparison Framework for Software Architecture Description Languages**. *IEEE Trans. Softw. Eng.*, **26** :70–93, January 2000. 64, 67, 68

[72] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 65

[73] George T. Heineman and William T. Councill, editors. *Component-based software engineering : putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 65

[74] Matthias Zenger. **Extensibility in the Large**. *First Doctoral Workshop on Global Computing*, June 2002. 65

[75] Robert Balzer. **Instrumenting, Monitoring, Debugging Software Architectures**, 1997. 65

[76] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. **Abstractions for Software Architecture and Tools to Support Them**. *IEEE Transactions on software engineering*, **21** :314–335, 1995. 66, 67, 71

[77] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. **Applying static analysis to software architectures**. *SIGSOFT Softw. Eng. Notes*, **22** :77–93, November 1997. 66

[78] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. **Specification and Analysis of System Architecture Using Rapide**. *IEEE Transactions on Software Engineering*, **21**(4) :336–355, 1995. 66, 67, 71

[79] Paola Inverardi and Alexander L. Wolf. **Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model**. *IEEE Trans. Softw. Eng.*, **21** :373–386, April 1995. 66

[80] Mehdi Jazayeri. **On Architectural Stability and Evolution**. In *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '02, pages 13–23, London, UK, UK, 2002. Springer-Verlag. 66

[81] Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf. **Aladdin : A Tool for Architecture-Level Dependence Analysis of Software Systems**. Technical report cu-cs-858-98, University of Colorado at Boulder, Avril 1998. 66

[82] Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf. **Chaining : A Software Architecture Dependence Analysis Technique**, 1997. 66

[83] Shawn A. Bohner. **Extending Software Change Impact Analysis into COTS Components**. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 175–, Washington, DC, USA, 2002. IEEE Computer Society. 66

[84] P. H. Feiler, B. A. Lewis, and S. Vestal. **AADL standard SAE Architecture Analysis and Description Language**. Technical report, Society of Automotive Engineers (SAE), November 2004. 66, 67, 73

[85] M.O. Hassan. **Contribution à l'analyse d'impact des modification des architecture logicielles**. 67

[86] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997. 67, 71

[87] D. Garlan, R. Monroe, and D. Wile. **Acme : An Architecture Description Interchange Language**. In *in Proceedings of CASCON'97*, pages 169–183, 1997. 67, 71

[88] Alexander L. Wolf. **Succeedings of the second international software architecture workshop (ISAW-2)**. *SIGSOFT Softw. Eng. Notes*, **22** :42–56, January 1997. 68

[89] Mary Shaw and David Garlan. **Characteristics of Higher-level Languages for Software Architecture**. Technical Report CMU-CS-94-210, Carnegie Mellon University, School of Computer Science, December 1994. 68

[90] Steve Vestal. **A Cursory Overview and Comparison of Four Architecture Description Languages**. Technical report, Honeywell Technology Center, 1993. 70

[91] A-M. Déplanche and S. Faucou. *Systèmes temps réel 1 : Techniques de description et de vérification (Chapitre Description d'architectures pour le temps réel : l'approche AADL)*. Hermes, Lavoisier, 2006. 71

[92] H. Feiler, B. Lewis, and S. Vestal. **The SAE Avionics Architecture Description Language (AADL) Standard : A Basis for Model-Based Architecture-Driven Embedded Systems Engineering**. In *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems*, Washington, D.C., May 2003. 71

[93] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual*. Addison-Wesley, 2005. 72

[94] T. Vergnaud. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique dŠapplications formellement vérifiées*. PhD thesis, Ecole Nationale Supérieure des Télécommunications de Paris, Décembre 2006. 72

[95] **World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Third Edition)**, 2004. 73

[96] Peter G. Selfridge. **Knowledge-based software engineering**. *IEEE Expert*, pages 11–12, 1992. 87

[97] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. **Profile-guided static typing for dynamic scripting languages**. *SIGPLAN Not.*, **44** :283–300, October 2009. 89

[98] Kingshuk Karuri, Mohammad Abdullah Al Faruque, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. **Fine-grained application source code profiling for ASIP design**. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 329–334, New York, NY, USA, 2005. ACM. 89

[99] Mikhail Dmitriev. **Profiling Java applications using code hotswapping and dynamic call graph revelation**. *SIGSOFT Softw. Eng. Notes*, **29** :139–150, January 2004. 89

[100] T. Mens. **Transformational Software Evolution by Assertions**. *Workshop on Formal Foundations of Software Evolution, CSRM 2001*, 2001. 96

[101] B. W. Boehm, J. R. Brown, and M. Lipow. **Quantitative evaluation of software quality**. In *ICSE '76 : Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. 114, 115, 116

[102] James A. Mccall, Joseph P. Cavano, and G. Walters. **Factors in Software Quality**. **1, 2, and 3**, November 1977. 114, 115, 117

[103] Joseph P. Cavano and James A. McCall. **A framework for the measurement of software quality**. *SIGSOFT Softw. Eng. Notes*, **3**(5) :133–139, 1978. 114, 115, 116

[104] Kaoru Ishikawa. *What Is Total Quality Control ? : The Japanese Way*. Prentice Hall, March 1985. 114

[105] Y. Akao. **QFD : Past, present, and future**. In *Transactions of the Third International Symposium on Quality Function Deployment*. vol. 1. Plenary Session (downloadable from the QFD InstituteŠs website : http ://www.qfdi.org), October 1Ű2, 1997. 114

[106] Akao Yoji. **History of quality function deployment in Japan**. In *The Best on Quality, IAQ Book Series Vol. 3*, page 183Ũ196. International Academy for Quality, 1990. 114

[107] Barbara Kitchenham and L. Pickard. **Towards a constructive quality model Part 2 : Statistical techniques for modeling software quality in the ESPRIT REQUEST project**. *Softw. Eng. J.*, july 1987. 114

[108] P.G. Petersen and B.A. Kitchenham. **The development of a software quality model**. In *Proceedings of 1st European Conference on Software Quality*, Brussels, April 1988. 114

[109] S. Chulani and B. Boehm. **Modeling Software Defect Introduction and Removal :COQUALMO (COnstructive QUALity MOdel)**, 1999. 114

[110] Version Copyright University. **COCOMO II Model Definition Manual**, 1999. 114

[111] V. Basili, G. Caldiera, and D. Rombach. **Goal/-question/metric paradigm**. *Encyclopedia of Software Engineering*, **1** :528–532, 1994. 114

[112] Bernard Wong and D. Ross Jeffery. **Cognitive Structures of Software Evaluation : A Means-End Chain Analysis of Quality**. In *PROFES '01 : Proceedings of the Third International Conference on Product Focused Software Process Improvement*, pages 6–26, London, UK, 2001. Springer-Verlag. 114

[113] B. Wong and R. Jeffery. **Quality Metrics : ISO9126 and Stakeholder Perceptions**, 1995. 114

[114] Barbara Kitchenham, Steve Linkman, Alberto Pasquini, and Vincenzo Nanni. **The SQUID approach to defining a quality model**. *Software Quality Control*, **6**(3) :211–233, 1997. 114

[115] Vidgen. **A multiple perspective approach to information systems quality**, 1996. 114

[116] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. **Empirical studies of software engineering : a roadmap**. In *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, New York, NY, USA, 2000. ACM. 114, 117

[117] N. Juristo and A.M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic, 2003. 114

[118] Barry Boehm, Hans Dieter Rombach, and Marvin V. Zelkowitz. *Foundations of Empirical Software Engineering : The Legacy of Victor R. Basili*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 114

[119] S. Chulani, B. Ray, P. Santhanam, and R. Leszkowicz. **Metrics for Managing Customer View of Software Quality**. In *METRICS '03 : Proceedings of the 9th International Symposium on Software Metrics*, page 189, Washington, DC, USA, 2003. IEEE Computer Society. 114

[120] Adeel Ahmad, Henri Basson, Laurent Deruelle, and Mourad Bouneffa. **Towards an Integrated Quality-Oriented Modeling Approach for Software Evolution Control**. In *ICSTE 2010 : International Conference on Software Technology and Engineering*. IEEE Computer Society, October 03-05 2010. 114, 120

[121] Adeel Ahmad, Henri Basson, Laurent Deruelle, and Mourad Bouneffa. **Modeling and analysis of change impact in quality-oriented representation of the software**. In *GCSE 2011 : Global Congress on Science and Engineering*. ELSEVIER, December 28-30 2011 to appear. 114, 120

[122] G. Goth. **Beware the March of this IDE : Eclipse is overshadowing other tool technologies**. *IEEE Software*, **22**(4) :108–111, August 2005. 141

[123] Laurent Deruelle and Henri Basson. **An Eclipse Platform for Analysis and Manipulation of Distributed Multi-Language Software**. In *CAINE : Proceedings of the ISCA 21st International Conference on Computer Applications in Industry and Engineering*, pages 100–105. ISCA, 2008. 143

Annexe - L'Architecte (Project outlines)

Figure 1: Le plug-in Eclipse Multi-lingual Parser

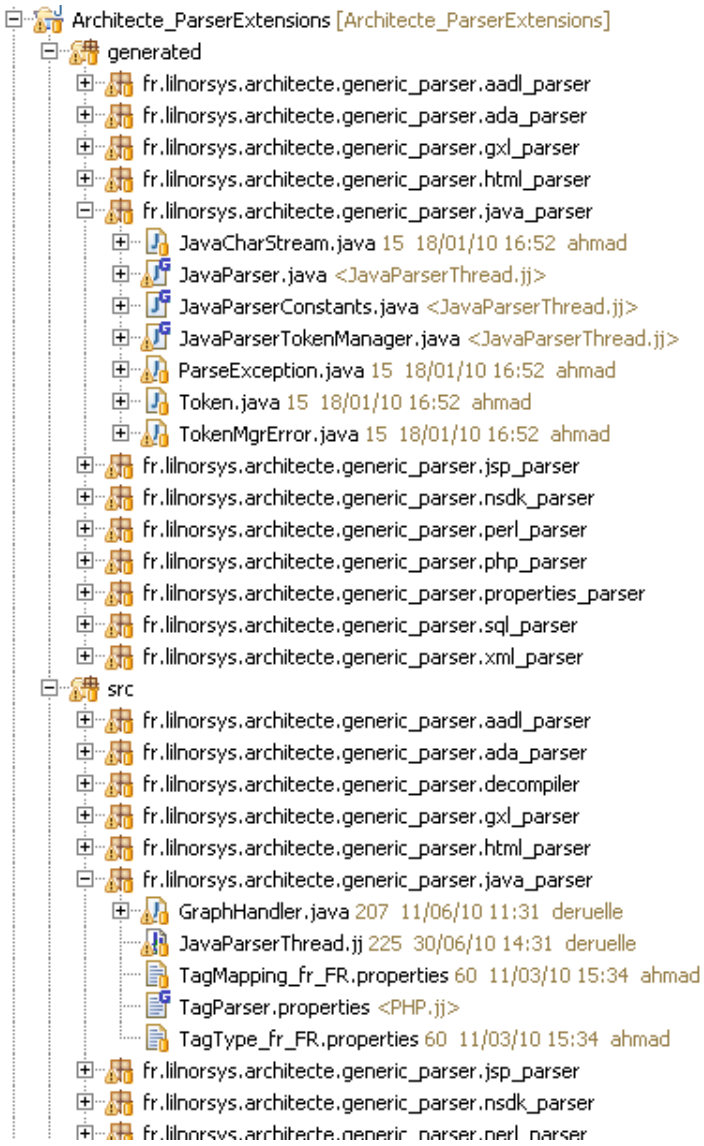


Figure 2: Le plug-in Eclipse Multi-lingual Parser - Package Java

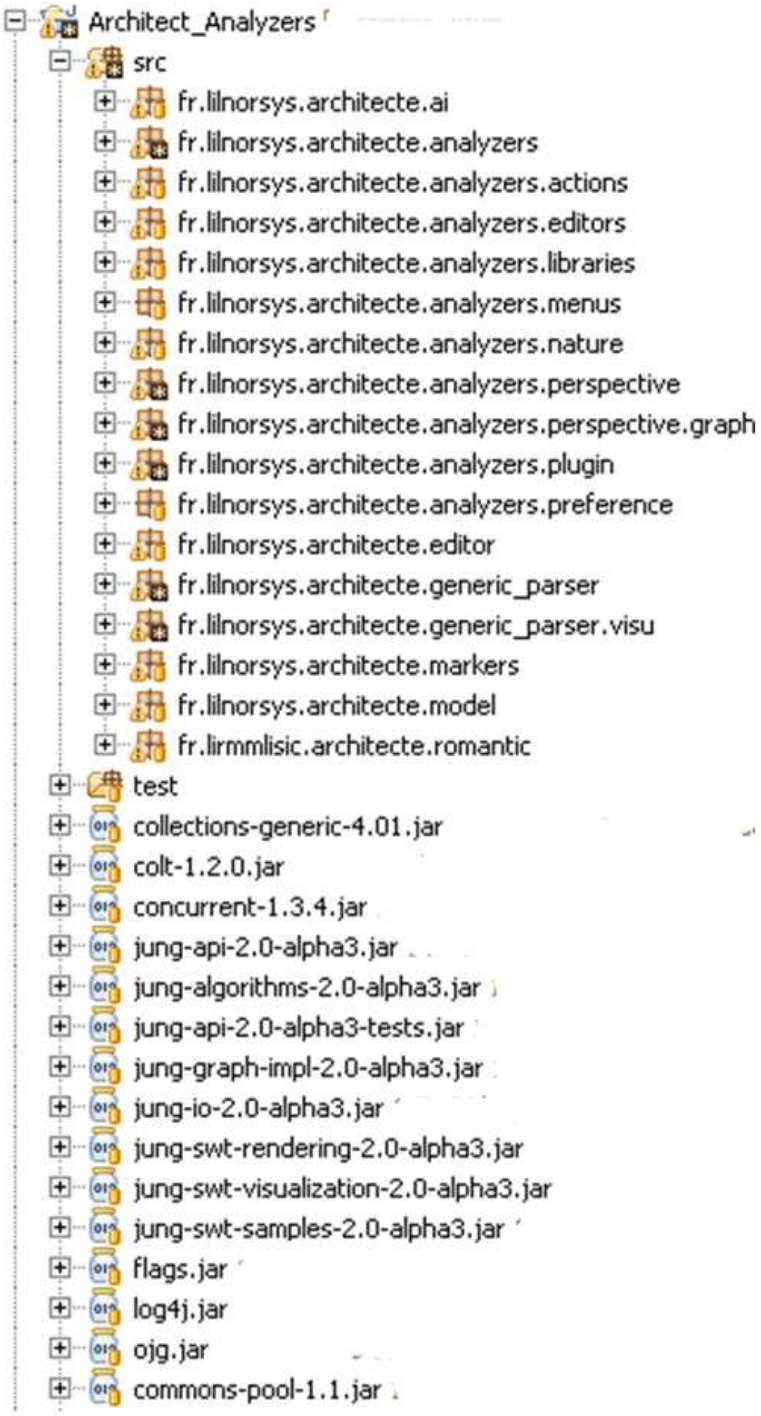


Figure 3: Le plug-in Eclipse graph visualizer

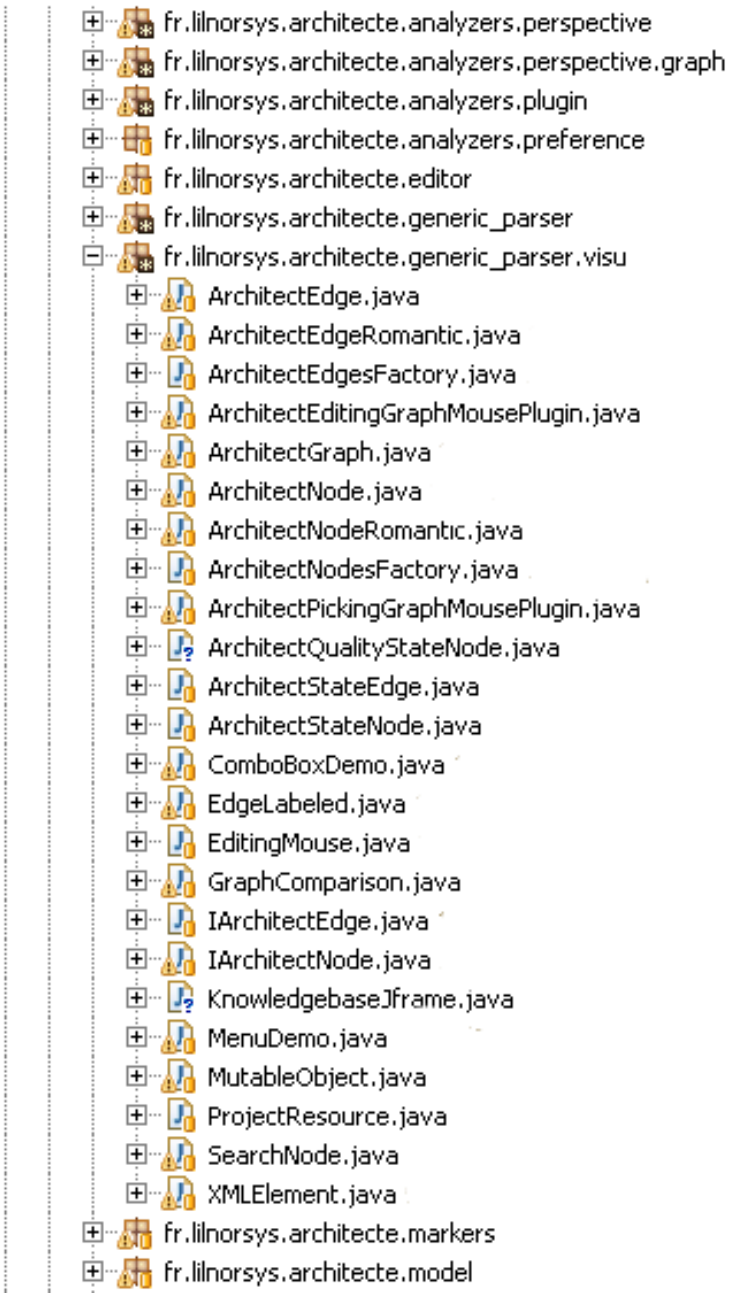


Figure 4: Le plug-in Eclipse graph visualizer - Package visu

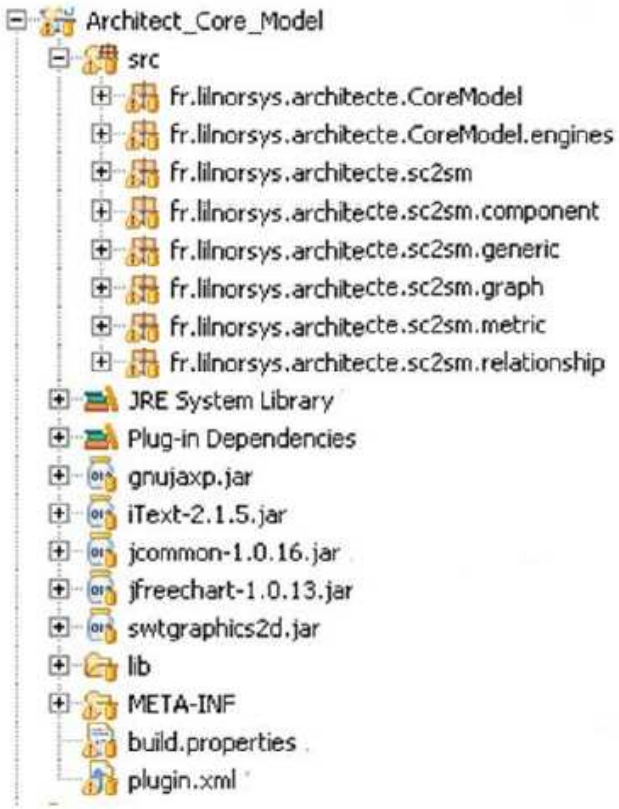


Figure 5: Le plug-in Eclipse Software Modeler

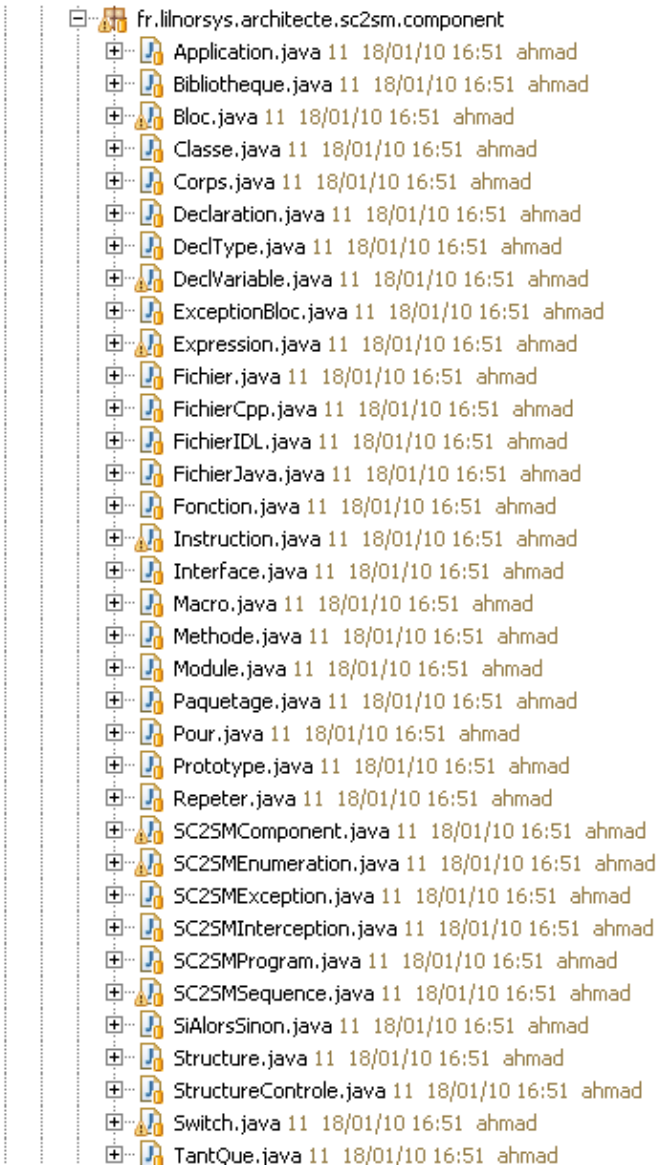


Figure 6: Le plug-in Eclipse Software Modeler (source code artifacts)

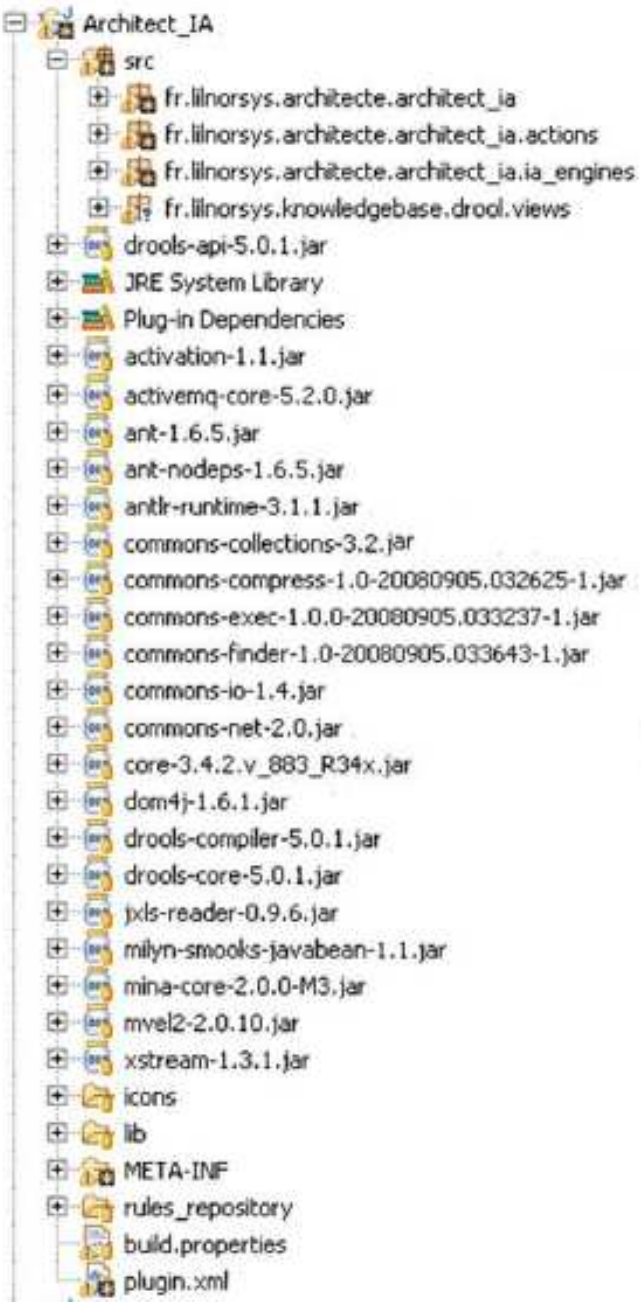


Figure 7: Le plugin Eclipse change propagation analyser

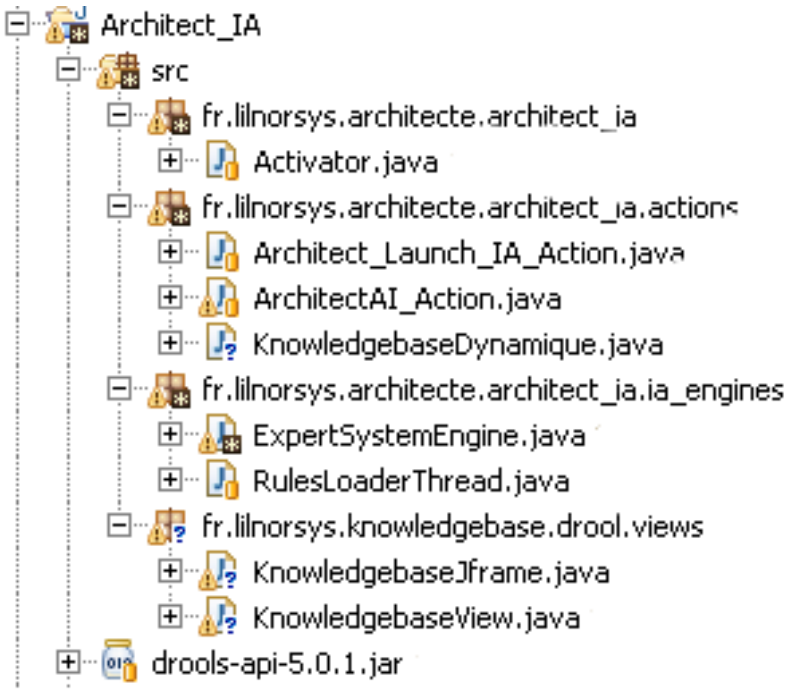


FIGURE 8: Le plugin Eclipse change propagation analyser - Les Classes

Contribution à la Multi-modélisation des Applications Distribuées pour le Contrôle de l'Évolution des Logiciels

Adeel Ahmad

Résumé : Le contrôle de l'évolution des logiciels exige une compréhension profonde des changements et leur impact sur les différents artefacts du système.

Nous proposons une approche de multi-modélisation pour l'analyse d'impact du changement pour une compréhension des effets des modifications prévus ou réels dans les systèmes distribués. Ce travail consiste à élaborer une modélisation des artefacts logiciels et de leur différents liens d'interdépendance pour construire un système à base de connaissances permettant, entre autres, d'assister les développeurs et les chargés de l'évolution des logiciels pour établir une évaluation *a priori* de l'impact des modifications. La modélisation que nous élaborons intègre deux descriptions majeures des logiciels, dans un premier temps, la description structurelle sous-jacente qui englobe l'ensemble des niveaux granulaires et l'abstraction des constituants logiciels, et ensuite la description qualitative conçue pour s'intégrer à la description précédente.

Deux modèles, d'abord élaborés individuellement pour les deux descriptions respectives, ont été intégrés ou mis en correspondance dans l'objectif d'étudier l'impact de toute modification et sa potentielle propagation à travers les constituants logiciels concernés. Lors de chaque modification, il devient alors possible d'établir un bilan qualitatif de son impact. La modélisation intégrée est élaborée pour se prêter à un raisonnement à base de règles expertes. La modélisation proposée est en cours d'expérimentation et validation à travers le développement d'une plate-forme d'implémentation basée sur l'environnement Eclipse.

Mots clé : Evolution du Logiciel, Analyse d'Impact des Modifications, Multi-modélisation des Logiciels, Architecture du Logiciel, Modélisation Structurelle, Modélisation Qualitative.

Abstract : The software evolution control requires a complete understanding of the changes and their impact on the various system artifacts.

We propose a multi-modeling approach for the change impact analysis to provide assistance in understanding the effects of projected or actual changes in distributed software systems. This work elaborate the modeling of software artifacts along with their various interdependencies to build a knowledge-based system, which allows, among others, an assistance for the software developers or maintenance engineers to establish an *a priori* evaluation of impact of changes. The model we develop integrates two major descriptions of software, at first, the underlying structural description that encompasses the levels of granularity and abstraction of software artifacts, and then the qualitative description designed to integrate the structural description.

Initially, the formal models are designed separately for the respective descriptions, and then these are integrated for the objective to study the change impact and its potential propagation through the affected software artifacts. For a change, it is important to establish a qualitative assessment of its impact. The integrated modeling leads to a reasoning based on expert rules. The proposed model is being tested and validated through the development of a platform, implemented in the Eclipse environment.

Keywords : Software Evolution, Change Impact Analysis, Software Multi-modeling, Software Architecture, Structural Modeling, Qualitative Modeling