ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAITRISE AVEC MÉMOIRE EN GENIE LOGICIEL
M. Sc. A.

PAR
Christel KAPTO DJAMPOU

INFÉRENCE DE L'ÉVOLUTION ARCHITECTURALE À PARTIR DU CODE SOURCE : UNE APPROCHE BASÉE SUR LA DÉTECTION DES TACTIQUES ARCHITECTURALES

MONTRÉAL, LE 15 SEPTEMBRE 2016

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

Mme Ghizlane El Boussaidi, directeur de mémoire
Département de génie logiciel et TI à l'École de technologie supérieure

M. Sègla Kpodjedo, codirecteur de mémoire
Département de génie logiciel et TI à l'École de technologie supérieure

M. Talhi Chamseddine, président du jury
Département de génie logiciel et TI à l'École de technologie supérieure

M. Gherbi Abdelouahed, membre du jury
Département de génie logiciel et TI à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 08 SEPTEMBRE 2016

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je dédie ce mémoire à mes parents qui ont toujours cru en moi et n'ont jamais ménagé d'effort pour m'aider et me soutenir dans mes choix.

Je remercie énormément Mme Ghizlane El Boussaidi, ma directrice de mémoire, de m'avoir guidé dans mes choix académiques, de son suivi très rigoureux de mes travaux et de toutes ses nuits sans sommeil dans la correction de mes livrables. Je tiens aussi à remercier M. Sègla Kpodjedo, codirecteur de mémoire, de son assistance et de ses conseils précieux pour enlever les obstacles dans mes travaux. Je remercie également mes camarades des recherches du Laboratoire en Architecture de Systèmes Informatiques (LASI) en particulier Alvine, Nesrine, Rajesh, Elyes, Nicolas, Sana, … de leur aide dans la réalisation de mes travaux.

Je remercie également le personnel des services aux étudiants de l'École de Technologie supérieure, en particulier Mme Alison Threatt de ses conseils pour mon intégration sociale dans la communauté de l'école et Christine Richard pour ses conseils et son mentorat dans la rédaction de ce mémoire.

Je remercie ma conjointe Nicaise de sa patience et ses encouragements. Un grand merci à mes frères et sœur Serge, Stephan, Olivier, Sandrine et Nathanaël qui ont toujours cru en moi et de leur soutien multiforme. Je remercie également la famille Girard et la famille Ndomkap de tous leurs efforts afin que je me sente comme chez moi à Montréal. Un merci particulier au collègue Pat sans qui cette aventure n'aurait pas commencée.

A toutes les personnes qui ont contribuées sous toutes les formes à la réalisation de ce projet de recherche trouvez en ces lignes ma profonde reconnaissance.

INFÉRENCE DE L'ÉVOLUTION ARCHITECTURALE À PARTIR DU CODE SOURCE : UNE APPROCHE BASÉE SUR LA DÉTECTION DES TACTIQUES ARCHITECTURALES

Christel KAPTO DJAMPOU

RÉSUMÉ

Plusieurs travaux ont été proposés pour étudier et extraire de l'information sur l'évolution d'un système logiciel. Mais peu d'approches analysent ces informations et les interprètent du point de vue architectural.

Dans ce mémoire, nous proposons une approche pour inférer l'évolution architecturale d'un système à partir du code source. Notre approche se base sur l'idée qu'une tactique architecturale peut correspondre à plusieurs représentations opérationnelles. Chaque représentation opérationnelle étant une transformation du système décrite à l'aide d'actions élémentaires sur les éléments du code source (ex. ajout d'un paquetage, déplacement d'une classe d'un paquetage à un autre, suppression d'un attribut ou méthode, etc.). Ces représentations opérationnelles permettent de : 1) détecter les occurrences d'applications (ou annulations) des tactiques en analysant différentes versions du code source des systèmes étudiés, et 2) de comprendre l'évolution architecturale de ces systèmes.

Pour évaluer notre approche, nous avons implémenté un outil appelé TacMatch (Tactic Matcher) qui permet de définir et détecter des représentations opérationnelles des tactiques. Nous avons procédé à une expérimentation sur plusieurs versions du projet libre **jFreeChart**. Nous nous sommes concentrés sur l'attribut de qualité **modificabilité**. Les résultats de cette expérimentation ont montré des cas intéressants où la connaissance des changements architecturaux par l'équipe de développement aurait facilité les changements dans le système. Nous avons aussi pu constater des habitudes de programmation de l'équipe de développement tel que les applications progressives de tactiques architecturales sur plusieurs versions.

Mots-Clés : Évolution logicielle, évolution architecturale, tactique architecturale, tactiques de modificabilité, détection des tactiques.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

CSV	Comma-Separated Values
OO	Orienté objet
ACS	Abstract common Service
SR	Split responsibility
UE	Use Encapsulation
IC	Increase cohesion

INTRODUCTION

Contexte et problématique

Tout au long du cycle de vie d'un logiciel, un ensemble de documentations est produit. Parmi ces documentations, l'une des plus importantes est la documentation de l'architecture du système (Clements et al., 2003). Elle est produite généralement dans une étape antérieure à l'implémentation du logiciel et devrait être mise à jour durant l'implémentation et la maintenance évolutive ou corrective du logiciel.

Dans la documentation de l'architecture, on consigne la manière dont le système sera organisé pour répondre aux besoins exprimés par les utilisateurs que ce soit des besoins fonctionnels (i.e., quelles fonctions sont supportées par le système) ou des besoins non fonctionnels (i.e., comment les fonctions sont supportées par le système). Les besoins non fonctionnels définissent des contraintes et critères qui permettent au système de supporter plus efficacement les besoins fonctionnels. Ces besoins non fonctionnels sont définis sous forme d'attributs de qualité tels que la performance, la disponibilité et la modificabilité. Pour aider les concepteurs dans la prise en compte des besoins non fonctionnels exprimés sous forme d'attributs de qualité, des décisions architecturales élémentaires qui permettent d'adresser les attributs de qualité ont été recensées. Elles sont appelées tactiques architecturales (Bachmann, Bass et Nord, 2007).

Vu les délais de production des logiciels de plus en plus courts et le volume sans cesse croissant des demandes de maintenance, la documentation de l'architecture après sa production n'est pas mise à jour. À travers les modifications non documentées du système, les décisions architecturales se perdent et l'on peut constater une déviation de l'architecture du système des attributs de qualité qui étaient supportés par son architecture initiale. Dans ce contexte, il est nécessaire d'analyser l'évolution d'un système et d'extraire les informations sur l'architecture pour permettre à l'équipe de développement de : 1) comprendre la manière

dont l'architecture du système a évolué ; et 2) prendre en compte les décisions architecturales implémentées par le système même quand elles n'ont pas été documentées.

Des travaux ont déjà été menés pour analyser l'évolution de l'architecture logicielle. Ces travaux peuvent être regroupés en deux groupes: les travaux analysant l'évolution de l'architecture en s'appuyant sur des métriques (Tonu, Ashkan et Tahvildari, 2006) et les travaux s'appuyant sur la représentation structurelle de l'architecture (Garlan et al., 2009; McNair, German et Weber-Jahnke, 2007). Bien qu'ils permettent d'avoir une vue globale de l'évolution de l'architecture, ces travaux ne se sont pas intéressés à l'analyse de l'évolution d'un système en tenant compte des décisions architecturales élémentaires (telles que les tactiques) que le système implémente. Une analyse de l'application ou annulation des tactiques architecturales à travers l'évolution d'un système permettrait de comprendre quels attributs de qualité sont supportés par le système et si ces attributs sont maintenus durant cette évolution.

Objectifs de recherche

L'objectif général de notre travail de recherche est d'analyser l'évolution d'un système logiciel pour inférer une tendance de son évolution au niveau architectural. En particulier, notre objectif est d'analyser différentes versions du code source d'un système et de comprendre quelles décisions architecturales élémentaires ont été appliquées ou défaites à travers l'évolution du code. Pour ce faire, nous nous sommes concentrés sur les tactiques architecturales qui sont des décisions élémentaires visant les attributs de qualité. Dans ce contexte, nous nous sommes fixé les sous-objectifs suivants :

1) Proposer des représentations opérationnelles des tactiques qui permettent de les détecter à travers l'évolution du code source ;
2) Mettre en œuvre un processus qui exploite ces représentations pour générer des algorithmes de détection des occurrences/annulations des tactiques ;
3) Implémenter les représentations proposées et le processus de détection dans un environnement de développement.

Dans ce mémoire, nous nous sommes limités à l'attribut de qualité qui est la modificabilité (Bachmann, Bass et Nord, 2007). La modificabilité est la capacité d'un système à subir des modifications à faible coût. Elle est prise en compte par un ensemble de tactiques qui répondent à 4 préoccupations : réduction de la taille des modules, augmentation de la cohésion, réduction du couplage et anticipation des modifications tardives. Nous nous sommes penchés dans notre projet de recherche sur 4 tactiques de la modificabilité : séparation de responsabilités, augmentation de cohésion, encapsulation et abstraction des services communs. Notre travail se concentre aussi sur les systèmes orientés objet (OO).

Méthodologie de recherche

Pour atteindre notre objectif, notre méthodologie a comporté 4 étapes :

- **Revue de la littérature :**

  Dans cette étape, nous nous sommes familiarisés avec des concepts de base reliés à notre approche (ex. attributs de qualité et tactique architecturales) et nous avons fait un état de l'art des approches portant sur l'analyse de l'évolution de l'architecture logicielle. Cette étape nous a permis aussi de positionner l'approche que nous proposons par rapport aux approches existantes.

- **Étude et analyse manuelle de l'évolution d'un système :**

  Dans cette seconde étape, nous avons analysé manuellement des versions consécutives d'un système OO et nous avons identifié les changements élémentaires (ajouts, suppressions, déplacements et renommage de paquetage, classe, méthode et attribut) qui ont été effectués entre ces versions. En nous basant sur la description des tactiques de la modificabilité obtenue de la première étape de notre méthodologie, manuellement nous avons effectué les regroupements des changements identifiés en tactiques. Ce travail nous a permis d'établir une correspondance entre certains groupes de changements élémentaires et certaines tactiques et de proposer pour ces

tactiques des représentations opérationnelles sous forme de groupes de changements élémentaires du code source.

- **Proposition et mise en œuvre de l'approche pour la détection des tactiques :**
  Dans cette étape, nous avons proposé une approche pour inférer l'évolution architecturale d'un système existant. Notre approche se base sur la représentation opérationnelle des tactiques. Cette représentation permet la détection des applications (ou annulation) de tactiques architecturales dans le système en analysant différentes versions de son code source. En particulier, nous définissons un pseudo-langage de représentation des tactiques et un processus qui exploite ces représentations pour détecter les tactiques dans le code source. Nous avons implémenté l'approche dans un outil appelé **TacMatch** (Tactic Matcher). TacMatch nous permet d'enregistrer les représentations des tactiques, d'enregistrer les différentes versions d'un système, de générer les changements entre versions en nous servant de MADMatch (Kpodjedo et al., 2013) et de faire la détection des tactiques représentées sur un ensemble donné de versions.

- **Étude de cas et analyse des résultats**
  La dernière étape de notre méthodologie visait à évaluer l'approche que nous proposons. Pour ce faire, nous avons appliqué notre approche sur plusieurs versions d'un système libre OO et nous avons analysé les résultats. Notre expérimentation avait pour objectif d'évaluer la fiabilité de l'approche et le support qu'elle offre pour inférer une tendance d'évolution au niveau architectural.

Organisation du mémoire

Dans le CHAPITRE 1 de ce mémoire, nous présentons un état de l'art des travaux sur l'analyse de l'évolution de l'architecture de même que les concepts de base reliés à notre approche. Le CHAPITRE 2 présente notre approche en détail. Dans le CHAPITRE 3, nous présentons le contexte dans lequel nous avons expérimenté notre approche. Les résultats et

leurs analyses sont présentés ensuite dans le même chapitre de même que les limites de l'expérimentation. Ce mémoire se termine par la conclusion et les perspectives de notre travail de recherche.

CHAPITRE 1

REVUE DE LITTÉRATURE

Dans ce chapitre, nous commençons par présenter des concepts pertinents à notre projet de recherche telle que l'architecture logicielle, les attributs de qualité et les tactiques architecturales. Nous faisons ensuite un survol des travaux qui portent sur l'évolution de l'architecture d'un système.

1.1 Concepts de base

1.1.1 Architecture logicielle

Dans les différents processus de développement, l'architecture du logiciel revient régulièrement comme un des livrables de la phase de conception. Comme Garlan et Shaw (1993) l'ont spécifié, la croissance et la complexification des logiciels ont entrainé les activités de développement des logiciels à aller au-delà de la simple spécification des structures de données et des algorithmes, pour prendre aussi en compte des aspects conceptuels et de spécification de la structure globale du système.

Garlan et Shaw (1993) définissent donc **l'architecture logicielle** comme étant une vue qui représente le système comme un ensemble d'unités de traitement (encore appelées **composants**), qui peuvent être internes ou externes au système, ainsi que les interactions entre ses composants (encore appelées **connecteurs**).

1.1.2 Attribut de qualité

Pour Bass, Clements et Kazman (2012), 3 groupes d'éléments influencent l'architecture d'un système : 1) les besoins fonctionnels du système ; 2) les besoins en ce qui concerne les attributs de qualité du système et 3) les contraintes. Les besoins fonctionnels expriment les services que le système doit fournir à l'utilisateur. Les contraintes sont des restrictions qui

sont exprimées par rapport au système, par exemple une contrainte sur l'environnement de déploiement peut avoir des implications sur l'architecture en fonction de l'environnement de déploiement cible.

Les besoins en ce qui concerne les d'attributs de qualité du système expriment les contraintes sur la façon dont le système doit être conçu pour mieux répondre aux besoins fonctionnels. Bass, Clements et Kazman (2012) identifient 7 principaux attributs de qualité : disponibilité, performance, convivialité, interopérabilité, sécurité, modificabilité et testabilité. Ces attributs de qualité sont regroupés en 2 catégories en fonction de la phase où ils s'appliquent au système : durant les phases de réalisation du système (modificabilité et testabilité) ou durant l'exécution du système (disponibilité, performance, convivialité, interopérabilité et sécurité).

1.1.2.1 Exemple d'attribut de qualité : la modificabilité

Dans (Bachmann, Bass et Nord, 2007) et (Bass, Clements et Kazman, 2012), la **modificabilité** est l'attribut de qualité d'un logiciel qui caractérise le coût des modifications de ce logiciel ainsi que la facilité avec laquelle il prend en compte les modifications.
Les changements pris en compte par la modificabilité sont ceux effectués par les développeurs et les architectes sur le système. Ces changements sont ceux qui interviennent uniquement entre la phase de conception et le déploiement de l'application.

1.1.3 Tactique architecturale

En définissant les attributs de qualité, Bass, Clements et Kazman (2012) ont aussi recensé des patrons élémentaires que les architectes peuvent utiliser pour satisfaire les attributs de qualité dans le système. Ces patrons élémentaires sont appelés tactiques architecturales. Bass, Clements et Kazman (2012) décrivent les tactiques architecturales comme des décisions de conception et Bachmann, Bass et Nord (2007) complètent cette définition en qualifiant les tactiques de transformation architecturale du système. Chaque tactique architecturale est une solution qui a été éprouvée dans le domaine de l'ingénierie logicielle pour prendre en compte l'attribut de qualité concerné par la tactique.

1.1.3.1 Exemples de tactiques de modificabilité

Dans (Bachmann, Bass et Nord, 2007) et (Bass, Clements et Kazman, 2012), la **modificabilité** est prise en compte par des tactiques qui répondent chacune à l'un des 4 objectifs suivants : réduction de la taille des modules, augmentation de la cohésion, réduction du couplage et anticipation des modifications tardives.

Réduction de la taille des modules

Les tactiques de ce groupe ont pour objectif de trouver une logique pour diviser les modules. L'idée ici est que plus un module est grand plus sa modification sera couteuse de même que l'impact de cette modification sur les autres modules. Il est donc nécessaire de diviser le module en petits modules pour réduire ce coût. La tactique de **séparation des responsabilités** permet de répondre à cet objectif.

Augmentation de la cohésion

Stevens, Myers et Constantine (1974) définissent la cohésion comme une mesure pour qualifier la force des associations intramodules. Pour Bachmann, Bass et Nord (2007), la modification d'une responsabilité serait moins couteuse si les autres responsabilités avec lesquelles elle est associée se trouvent dans le même module. Les tactiques identifiées sont :

1. **Augmentation de la cohérence sémantique :** Elle préconise de regrouper les responsabilités ayant une même sémantique (donc sujettes à être modifiées ensemble en cas de changement) dans un module.
2. **Abstraction des services communs :** Elle préconise d'identifier les services similaires présents dans différents modules et de les fusionner dans un module. La fusion de ces services devra les rendre assez génériques pour continuer à fournir le même service à chaque module source de ces services.

Réduction du couplage

Stevens, Myers et Constantine (1974) définissent le couplage comme la mesure des associations entre deux modules. Cette relation est asymétrique. Si un module X a un fort couplage avec un module Y, les modifications effectuées sur X ont plus de probabilités de se répercuter sur le module Y. Les tactiques pour réduire le couplage sont :

1. **Encapsulation :** L'objectif de l'encapsulation est de réduire l'impact des changements d'un module sur les autres modules en introduisant une interface entre ces derniers et le module. Le module garde un couplage fort avec son interface, mais le couplage entre l'interface et le module est très faible, voire nul.
2. **Utilisation d'un "*wrapper" :*** Le "*wrapper"* est une forme d'encapsulation où l'interface transforme les données et les appels pour le module.
3. **Élévation du niveau d'abstraction :** Élever le niveau d'abstraction d'une responsabilité revient à rendre ses fonctionnalités paramétrables. Cette tactique implique donc l'ajout d'une responsabilité pour interpréter les paramètres de la responsabilité de départ. Les modifications subies par des responsabilités externes ne se répercuteront donc pas sur la responsabilité, mais sur son interpréteur de paramètre.
4. **Utilisation d'un intermédiaire :** Le couplage entre deux responsabilités peut être réduit en introduisant un intermédiaire entre les deux. Mais de peur de juste déporter le couplage sur les modules, le type intermédiaire doit avoir un faible couplage avec chacune des deux responsabilités.

Anticipation des modifications tardives (« Deferred Binding »)

Pour Bachmann, Bass et Nord (2007), une modification convenablement préparée même si elle survient tard dans le cycle de vie du système coutera moins cher que la même modification apportée plus tôt. La modification devrait donc être prise en compte au moment de la conception du système pour anticiper sa survenance plus tard. Les tactiques dans cette catégorie s'appuient sur les tactiques discutées précédemment. Le Tableau 1.1 présente une synthèse des tactiques d'anticipation de modifications tardives ainsi que les tactiques sur lesquelles ces dernières s'appuient.

Tableau 1.1 Correspondance entre les tactiques d'anticipation des modifications tardives dans le cycle de vie et les autres tactiques de modificabilité

Moment survenance	**Tactiques anticipation des modifications tardives**	**Réduction de la Taille des Modules**	**Réduction du Couplage**			**Augmentation de la Cohésion**
		Sep. Resp.	Encaps.	Aug. Niv. Abst.	Interm.	Abstr. serv. Comm.
codage	Programmation orientée aspect	✓				✓
	polymorphisme	✓				✓
	Modules paramétrés			✓		
construction	composants remplaçables	✓				✓
déploiement	fichiers de configuration			✓		✓
initialisation	fichiers de ressources			✓		
exécution	enregistrements à l'exécution		✓	✓		
	Interprétation des paramètres			✓		
	Liaison au démarrage		✓	✓		✓
	Liaison à l'exécution		✓	✓		✓
	serveurs de noms			✓	✓	
	Plug-Ins			✓	✓	
	Éditeur et abonné			✓	✓	

1.2 Contexte des travaux analysant l'évolution des architectures

Les travaux portant sur l'évolution de l'architecture se décomposent généralement en 2 étapes : 1) la reconstruction de l'architecture ; et 2) l'analyse de l'architecture. La reconstruction s'effectue la plupart du temps par diverses méthodes de la rétro-ingénierie des codes sources, cette étape est hors du cadre de notre étude. Ce chapitre se concentre donc sur l'étape d'analyse de l'architecture. Les travaux d'analyse de l'évolution de l'architecture peuvent être regroupés en 2 grands groupes en fonction de leur approche d'analyse : 1) les

travaux utilisant une évaluation métrique de l'architecture pour analyser l'évolution ; et 2) ceux effectuant une comparaison structurelle de l'architecture pour analyser l'évolution.

Dans les travaux de ces deux groupes on y retrouve des travaux qui effectuent aussi bien des analyses rétrospectives que prédictives sur l'architecture. L'analyse rétrospective porte sur des versions[1] antérieures du système pour en extraire des informations sur l'évolution passée du système, tandis que l'analyse prédictive se sert des versions antérieures pour faire des propositions aux architectes sur les modifications futures du système.

1.3 Approches d'analyse de l'évolution architecturale centrées sur l'évaluation de métriques

1.3.1 Présentation générale

Une métrique est une mesure qui permet de quantifier un élément. Dans le domaine du logiciel, les métriques sont utilisées pour quantifier des propriétés du logiciel. Parmi les métriques les plus couramment utilisées, on trouve celles de (Chidamber et Kemerer, 1994) utilisées pour évaluer les systèmes orientés objet et celles de (McCabe, 1976) utilisées pour évaluer la complexité des logiciels.

Dans l'analyse de l'évolution architecturale d'un système, des métriques sont définies pour caractériser l'architecture du système. Ces métriques sont évaluées pour chaque version du système et des graphiques de ces métriques sont construits. L'analyse des graphiques permet d'obtenir des tendances sur l'évolution de l'architecture.

Des travaux utilisent les métriques existantes (ex. (Tonu, Ashkan et Tahvildari, 2006)), tandis que d'autres définissent leurs propres métriques en adaptant les métriques existantes (ex. (Le et al., 2015)).

---

[1] La notion de version ici faire référence au système à un instant donné. Elle peut correspondre effectivement à une version au sens normal du système ou juste une validation du système dans le gestionnaire de versions.

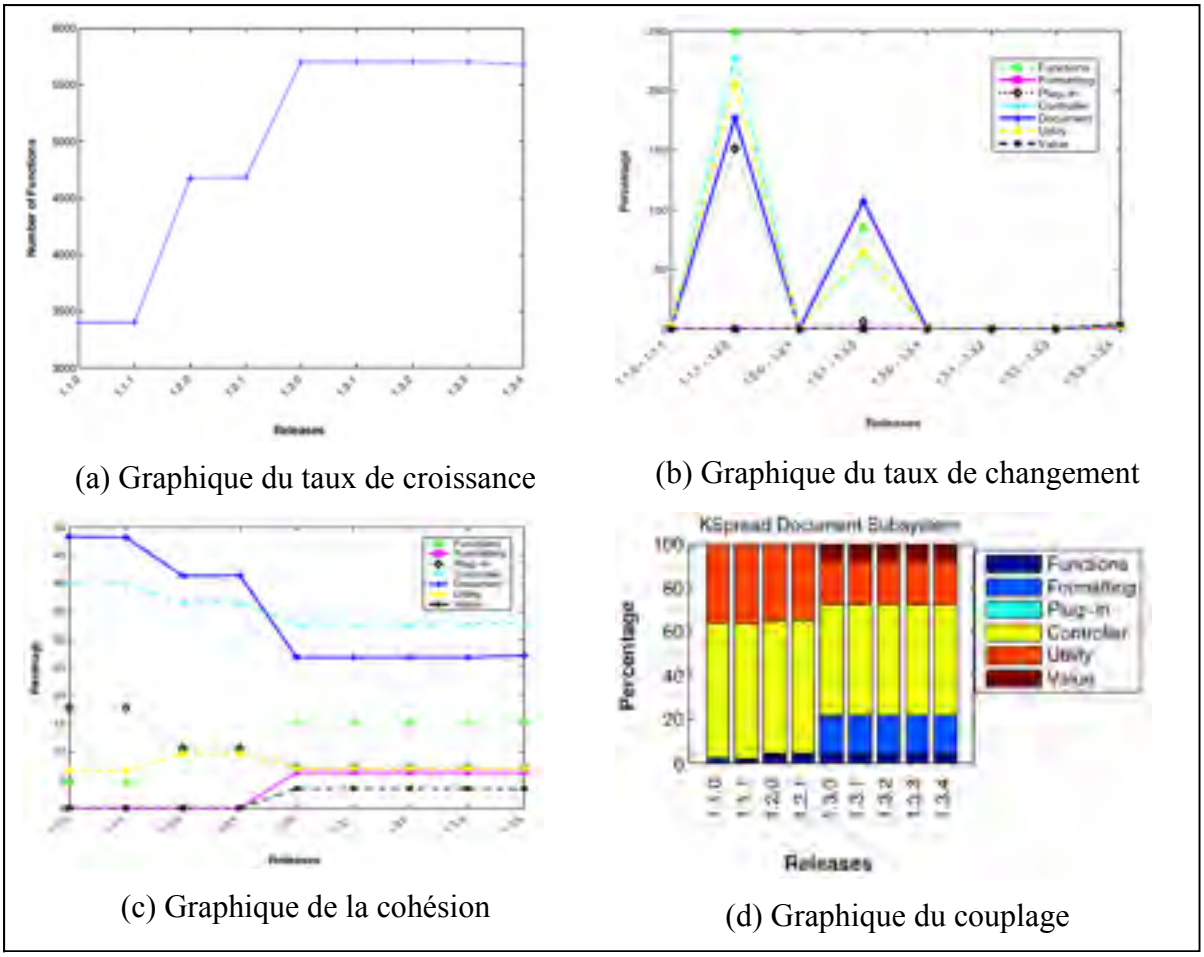
1.3.2 L'approche de (Tonu, Ashkan et Tahvildari, 2006)

Tonu, Ashkan et Tahvildari (2006) effectuent une analyse rétrospective et prédictive de la stabilité de l'architecture du système. La stabilité pour les auteurs est la capacité du système à évoluer en gardant son architecture cohérente.

Tonu, Ashkan et Tahvildari (2006) identifient 4 mesures pour effectuer les analyses :

1) **Le taux de croissance du système** : il permet d'évaluer la complétude du système. Il est évalué avec des métriques telles que le nombre de lignes de code, le nombre de fichiers, le nombre de fonctions, le nombre d'appels de fonction ou le nombre de sous-systèmes ;
2) **Le taux de changement du système :** il complète l'analyse du taux de croissance et permet de définir le niveau d'évolution du système. Il est évalué avec des métriques telles que le nombre de fonctions modifiées ou le nombre d'appels de fonctions modifiés.
3) **La cohésion :** elle définit les interactions entre les objets d'un même sous-module.
4) **Le couplage :** cette mesure donne le degré de dépendances entre deux sous-modules du système. Tout comme pour la cohésion, des seuils sont définis pour calibrer et évaluer le système selon cette mesure.

Ces travaux ont été expérimentés sur plusieurs versions des tableurs libres *Gnumeric* et *KSpread*. Les résultats de (Tonu, Ashkan et Tahvildari, 2006) permettent d'identifier la version à partir de laquelle le système peut être qualifié de stable. Par exemple, dans la Figure 1.1 on peut observer que pour *KSpread* à partir de la version 1.3.0 tous les graphes deviennent constants.



(a) Graphique du taux de croissance

(b) Graphique du taux de changement

(c) Graphique de la cohésion

(d) Graphique du couplage

Figure 1.1 Graphes obtenus pour l'analyse de KSpread
Tirée de (Tonu, Ashkan et Tahvildari, 2006)

1.3.3 L'approche ARCADE de (Le et al., 2015)

Le et al. (2015) présentent une approche appelée *ARCADE* ("*Architecture Recovery change, And Decay Evaluator*"). *ARCADE* effectue une analyse rétrospective du système pour évaluer les changements architecturaux au niveau système et composant.

ARCADE utilise 3 métriques créées en adaptant les métriques MoJoFM (Zhihua et Tzerpos, 2004) : 1) architecture à architecture (*a2a*) mesure les différences entre les architectures de deux versions du système ; 2) cluster à cluster (*c2c*) mesure le pourcentage d'entités

communes à deux clusters ; 3) couverture d'un cluster (*cvg*) mesure le pourcentage de clusters communs aux architectures de deux versions du système.

Le et al. (2015) procèdent à une étude empirique sur différentes versions de 14 systèmes libres. Les résultats obtenus avec *ARCADE* permettent à Le et al. (2015) de conclure que les changements architecturaux majeurs s'effectuent tout autant entre versions mineures qu'entre versions majeures et que le passage d'une version majeure à une autre version majeure entraine beaucoup d'instabilité.

1.3.4 Limites des travaux centrés sur l'utilisation des métriques

Les travaux de (Le et al., 2015; Tonu, Ashkan et Tahvildari, 2006) basés sur les métriques pour analyser les systèmes permettent d'avoir un bon aperçu sur la stabilité globale de l'architecture. Cependant, ces approches ne permettent pas d'identifier et d'observer facilement les changements effectués entre les versions. En effet pour l'architecte, ce serait une information pertinente de savoir que l'instabilité provient du déplacement d'un groupe particulier de classes ou bien de la suppression d'un paquetage.

1.4 Approches d'analyse de l'évolution architecturale centrées sur la représentation structurelle de l'architecture

1.4.1 Présentation générale

Les travaux centrés sur la représentation structurelle comparent deux versions différentes d'une architecture du point de vue structurelle et identifient les changements (ajout, modification, suppression, fusion, séparation) qui ont été faits sur les constituants de l'architecture (composant, connecteur) entre ces deux versions.

Les changements peuvent être présentés aux architectes visuellement dans une architecture ((Garlan et al., 2009), (McNair, German et Weber-Jahnke, 2007)), comme une liste de

changement ((Abi-Antoun et al., 2006), (Kpodjedo et al., 2013)) ou en proposant une interprétation des changements observés ((Kim et al., 2010))

1.4.2 L'outil *Ævol* de (Garlan et al., 2009)

Garlan et al. (2009) proposent une approche prédictive qui permet aux architectes de choisir le meilleur ordonnancement pour faire évoluer l'architecture du système. Leurs travaux s'appuient sur la notion de chemin d'évolution qui est une succession d'architectures d'une architecture source vers une architecture cible. Garlan et al. (2009) ont implémenté leur approche dans l'outil *Ævol* qui est un plug-in eclipse et AcmeStudio.

Pour calculer le meilleur chemin d'évolution, *Ævol* permet de : 1) spécifier les architectures ; 2) regrouper les architectures en chemin d'évolution ; 3) définir des contraintes sur les chemins d'évaluation et 4) évaluer les chemins pour identifier le meilleur chemin en fonction des contraintes définies pour partir de l'architecture source vers l'architecture cible.

Pour présenter les chemins d'évolution à l'architecte, *Ævol* utilise une représentation en graphe. Dans ce graphe, les nœuds sont les architectures et les arcs sont les transitions entre architectures. Par exemple, la Figure 1.2 montre un chemin d'évolution pour passer d'une architecture source vers une architecture cible.

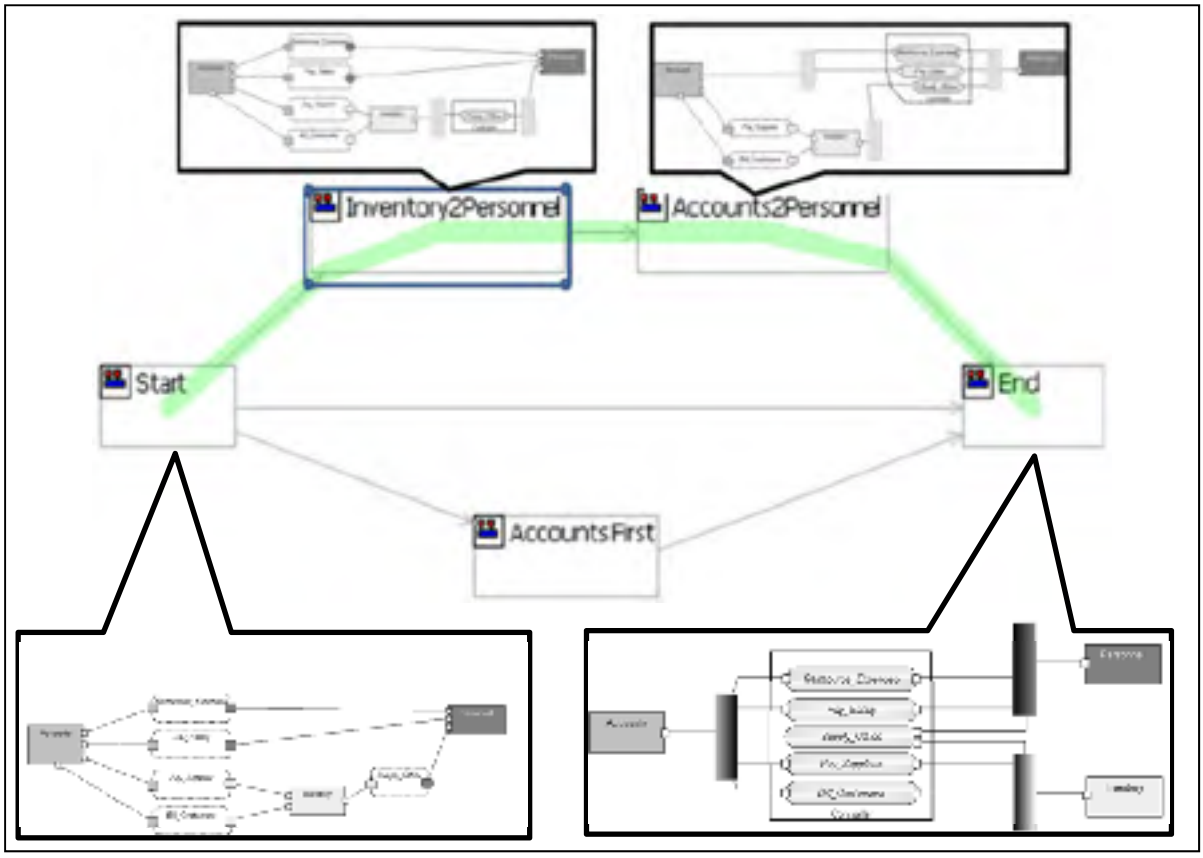


Figure 1.2 Exemple de chemin d'évolution
Adaptée de (Garlan et al., 2009)

1.4.3 L'outil *Motive* de (McNair, German et Weber-Jahnke, 2007)

McNair, German et Weber-Jahnke (2007) proposent une approche rétrospective pour visualiser les changements survenus sur l'architecture d'un système. *Motive* est le prototype mis œuvre par les auteurs pour supporter leur approche.

Motive utilise le gestionnaire de versions pour extraire les changements survenus sur l'architecture entre deux dates données. Avec ces changements, *Motive* présente une vue de l'architecture dans laquelle la couleur de chaque composant définit l'action qu'il a subie (voir Tableau 1.2). La vue peut être filtrée par le développeur ayant effectué la modification.

Tableau 1.2 Liste des actions répertoriées sur les composants de l'architecture
Adapté de (McNair, German et Weber-Jahnke, 2007)

Action	Description de l'action sur l'entité/composant	Couleur
Ajouté	N'existe pas dans la version de départ.	Vert
Supprimé	N'existe pas dans la dernière version de l'évaluation.	Noir
Fantôme	N'existe pas dans la version de départ et dans la dernière version, mais figure dans les versions intermédiaires.	Rose
Modifié	Toutes modifications autres que l'ajout ou la suppression.	Jaune
MetaData	Modification des métadonnées telles que le nom.	Bleue
Inchangé	Aucune modification n'affecte l'entité.	Gris

McNair, German et Weber-Jahnke (2007) appliquent Motive sur 2 systèmes libres : *JGraphpad* et *SQuirreL-SQL*. Les résultats obtenus leur permettent de répondre aux préoccupations de 3 intervenants dans la réalisation d'un système à savoir : les chercheurs, les développeurs et les gestionnaires.

1.4.4 L'approche de (Abi-Antoun et al., 2006)

Abi-Antoun et al. (2006) proposent aussi une approche que l'on peut qualifier de rétrospective pour visualiser les changements survenus sur l'architecture d'un système. Leur approche s'appuie sur un algorithme qui effectue la comparaison des représentations des architectures sous forme d'arbre. L'algorithme de (Abi-Antoun et al., 2006) permet la détection des changements suivants sur les nœuds et connecteurs de l'architecture : ajout, suppression et renommage. L'algorithme de (Abi-Antoun et al., 2006) permet de comparer les arbres en se libérant de certaines contraintes sur les nœuds des arbres comme : avoir un identifiant identique pour les nœuds, avoir les mêmes types de nœuds, ou avoir le même ordre dans les deux arbres. Cette flexibilité permet à l'algorithme de (Abi-Antoun et al., 2006) de détecter aussi des déplacements de composant entre les niveaux architecturaux.

Abi-Antoun et al. (2006) ont réalisé une validation empirique de leur algorithme en l'appliquant à 3 systèmes : *Duke's Bank*, *Asphyds* et *HillClimber*. Dans les 3 cas, les auteurs effectuent une comparaison de l'architecture définie par les concepteurs et celle obtenue par rétro-ingénierie des codes sources. Leurs résultats permettent de constater des violations dans les architectures implémentées dans les codes sources.

1.4.5 L'outil MADMatch de (Kpodjedo et al., 2013)

Dans le même ordre d'idée que les travaux de (Abi-Antoun et al., 2006), Kpodjedo et al. (2013) proposent l'outil *MADMatch* pour identifier les changements entre deux architectures. Mais à la différence des autres outils, tels que celui développé dans (Abi-Antoun et al., 2006), *MADMatch* permet de retrouver des correspondances plusieurs à plusieurs des entités des différentes versions. Cette particularité permet à *MADMatch* de détecter des actions de fusion et de division d'entité.

MADMatch prend en entrée une représentation du système sous forme de graphe étiqueté. Ce graphe représente un diagramme de classes recouvert à partir du code source en utilisant l'outil Ptidej (Guéhéneuc et Antoniol, 2008). Dans ce graphe, les nœuds représentent les entités du système (paquetage, classe, méthode, attribut) et les arcs représentent les relations entre ces entités. Le Tableau 1.3 et le Tableau 1.4 listent respectivement les types de nœuds et les types d'arcs utilisés dans *MADMatch*.

Tableau 1.3 Types de nœuds utilisés dans MADMatch
Adapté de (Kpodjedo et al., 2013)

Nœuds	
Type 0	Sous-programmes et paquetages
Type 1	Classes et interfaces
Type 2	Méthodes
Type 3	Attributs

Tableau 1.4 Types d'arcs utilisés dans MADMatch
Adapté de (Kpodjedo et al., 2013)

Arcs	$A \rightarrow B$
Type 1	La classes A utilise la classe B
Type 2	La classe A agrège la classe B
Type 3	La classe A hérite de la classe B
Type 4	La méthode A appelle la méthode B
Type 5	La méthode A utilise l'attribut B
Type 6	L'attribut A est de type B
Type 7	La méthode A a pour type de retour B
Type 8	La méthode A a un paramètre de type B
Type 9	L'entité A contient l'entité B

Pour comparer les graphes des deux versions, *MADMatch* effectue une correspondance entre les nœuds et leur voisinage (liste des relations du nœud). La réduction de l'ensemble des correspondances est effectuée sur la base des informations lexicales et structurelles des entités concernées. En sortie *MADMatch* produit un fichier CSV contenant les changements identifiés niveau code source entre les deux versions.

Kpodjedo et al. (2013) ont réalisé une validation empirique de *MADMatch* sur plusieurs versions du projet libre *JFreeChart*. Les résultats obtenus permettent de voir des opérations de refactoring systématiques dans le système (renommage, déplacement, fusion ou division d'entité).

1.4.6 L'outil *Ref-Finder* de (Kim et al., 2010)

Plusieurs travaux de détection de patron ont déjà été proposés dans la littérature (Arcelli Fontana et Zanoni, 2011; Arcelli et Cristina, 2007; Guéhéneuc et Antoniol, 2008), *Ref-Finder* de Kim et al. (2010) est un cas intéressant à noter. *Ref-Finder,* à la différence des autres outils de détection, effectue sa détection entre plusieurs versions. *Ref-Finder* analyse les

codes sources de 2 versions d'un système et détecte les occurrences de 63 des 72 refactorings du catalogue de Fowler (Fowler, 1999).

La mise en œuvre de *Ref-Finder* a été faite selon les étapes suivantes : étude des refactorings à détecter, représentation des refactorings suivant un formalisme spécifique (i.e., les refactorings sont représentés sous forme de requêtes de condition logique sur le système avant et après le refactoring) et détection de cette représentation dans le système à analyser (i.e., *Ref-Finder* a été implémenté sous forme de plug-in eclipse).

Kim et al. (2010) ont évalué *Ref-Finder* sur une paire de versions de jEdit et une paire de révisions de "*Columba and Carol*". Les résultats obtenus de ces détections ont une précision de 79% et un rappel de 95%.

1.4.7 Limite des travaux centrés sur la représentation structurelle de l'architecture

A la différence des travaux basés sur les métriques, les travaux basés sur la représentation structurelle de l'architecture permettent d'identifier les changements subis par les systèmes à travers leur évolution. Certains permettent de souligner les violations par rapport à l'architecture initiale du système.

Cependant, une liste de changements observés ne donne pas encore assez d'informations qualitatives à l'architecte ou aux équipes de développement pour apprécier si le changement est bon ou pas, en particulier, l'effet des changements sur les attributs de qualité qui ont guidé la conception du système analysé. Dans ce contexte, il y a besoin d'approches pouvant regrouper les changements identifiés et les interpréter à la lumière des décisions architecturales prises par les concepteurs et les attributs de qualité que ces décisions supportent.

1.5 Conclusion

Deux principaux courants sont observés pour l'analyse de l'évolution des architectures. Les travaux du premier courant ((Tonu, Ashkan et Tahvildari, 2006) et (Le et al., 2015)) effectuent leur analyse en déduisant des tendances à partir des graphes des métriques obtenues à partir de différentes versions du système. Bien qu'intéressants, les résultats obtenus ne permettent pas d'identifier ni d'interpréter en détail les changements observés.

Le second courant d'approche centre son analyse sur la comparaison structurelle des architectures. Dans ce courant, on trouve les travaux de (Garlan et al., 2009) et ceux de (McNair, German et Weber-Jahnke, 2007) qui donnent une représentation visuelle des changements respectivement à faire dans l'architecture et observés dans l'architecture. On peut aussi mentionner les travaux, telle que ceux de (Abi-Antoun et al., 2006) et ceux de (Kpodjedo et al., 2013), qui proposent des algorithmes de comparaison pour identifier les changements entre les versions d'un système. Tous ces travaux nécessitent encore une interprétation des changements observés par un architecte.

L'outil *Ref-Finder* de (Kim et al., 2010) est un cas particulier notable puisqu'il permet la détection des changements subis par un système et l'interprétation de ces changements. Mais, cette interprétation est faite en se basant sur des refactorings de plus bas niveau que les décisions architecturales.

Ce manque d'interprétation des changements au niveau architectural a été la motivation pour notre approche. Cette dernière a comme objectif d'analyser l'évolution d'un système pour identifier et regrouper les changements correspondant à des applications (ou annulations) des tactiques architecturales. Détecter des applications ou annulations de tactiques architecturales permet à un concepteur d'interpréter immédiatement les effets des changements subis par le système en termes d'effets sur les attributs de qualité supportés par le système.

CHAPITRE 2

UNE APPROCHE POUR INFÉRER L'ÉVOLUTION ARCHITECTURALE D'UN SYSTÈME À PARTIR DU CODE SOURCE

Ce chapitre présente l'approche que nous proposons pour inférer l'évolution architecturale d'un système existant. Notre approche se base sur la détection des applications (ou annulation) de tactiques architecturales dans le système en analysant différentes versions de son code source. Après une présentation de la vue générale de l'approche, chaque étape de l'approche sera détaillée.

2.1 Présentation générale

Comme présentée dans la Figure 2.1, l'approche proposée se décompose en deux processus : définition des tactiques et analyse de l'évolution du système.

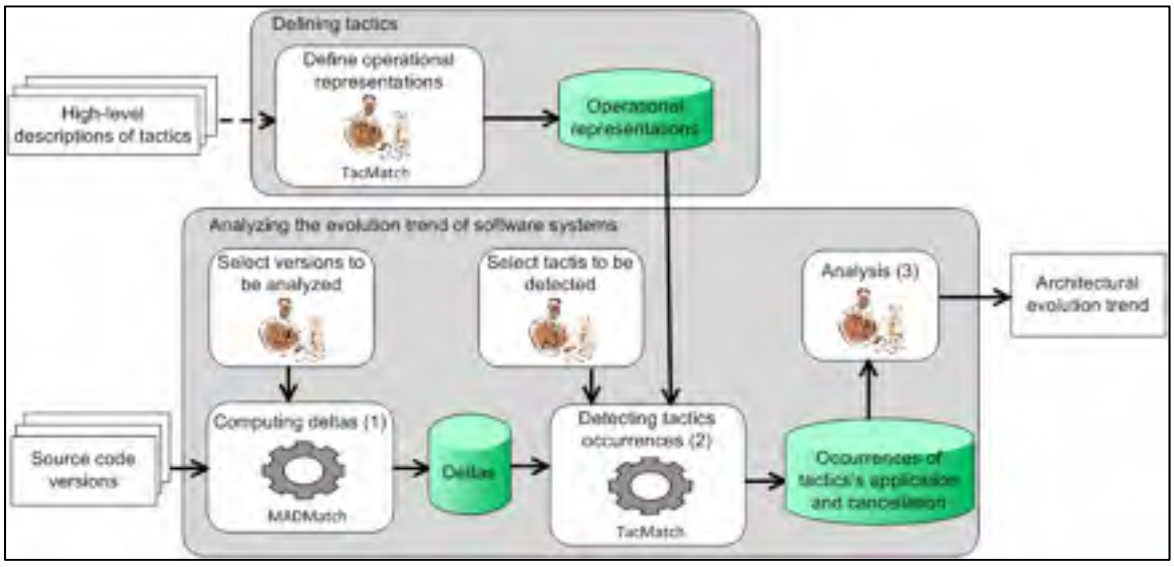


Figure 2.1 Vue globale de l'approche
Tirée de (Kapto et al., 2016)

Le processus de définition des tactiques permet d'analyser les définitions des tactiques afin d'obtenir des représentations opérationnelles de ces dernières. Ce processus prend en entrée la description de haut niveau d'une tactique architecturale telle que définie dans la littérature (Bass, Clements et Kazman, 2012) et produit des représentations concrètes/opérationnelles de cette tactique selon le contexte ciblé; le contexte est défini par le paradigme (ex. orienté objet) et le langage de programmation (ex. Java, C++) du système à analyser. Ces représentations opérationnelles sont formalisées dans un pseudo-langage et stockées dans une base de données. Ce processus est décrit en détail dans la section 2.2.

Le processus d'analyse de l'évolution est composé de trois étapes qui sont numérotées de 1 à 3 dans la Figure 2.1. La première étape s'appuie sur un outil de différentiation (i.e. MADMatch (Kpodjedo et al., 2013)) pour générer les différences, appelées Deltas, entre différentes versions du système étudié. Ces Deltas sont exprimés sous forme d'un ensemble de changements du code source (ex. classe ajoutée, classe modifiée, etc.). La deuxième étape vise à mettre en correspondance les Deltas générés et les représentations opérationnelles des tactiques pour détecter des applications ou annulations de tactiques. Pour ce faire, nous avons conçu et implémenté un outil, appelé **TacMatch** ("*Tactic Matcher*"), qui génère automatiquement les algorithmes de détection à partir des représentations opérationnelles des tactiques et applique ces algorithmes sur les deltas étudiés pour trouver les occurrences des tactiques (ou leur annulation). Dans la dernière étape (étape 3 dans la Figure 2.1), le concepteur analyse les occurrences détectées pour inférer la tendance de l'évolution du système au niveau architectural. L'ensemble du processus est décrit en détail à la section 2.3.

2.2 Définition des tactiques

2.2.1 Description de haut niveau des tactiques

Comme mentionné dans la section 1.1.3, une tactique peut être définie comme une transformation effectuée sur un système pour satisfaire un attribut de qualité donné. La description de haut niveau est une formalisation de cette transformation.

Nous rappelons aussi que l'architecture logicielle est communément définie en termes de composants et connecteurs (Garlan et Shaw, 1993). En s'appuyant sur cette représentation de l'architecture, la transformation effectuée par la tactique peut se traduire par un ensemble d'actions sur les éléments constituants l'architecture : composants et connecteurs. Les actions suivantes sont applicables sur ces constituants de l'architecture :

- Actions sur les composants : Créer, supprimer et déplacer un composant de l'architecture. Un composant peut être modifié par l'ajout, la suppression d'une responsabilité ou le déplacement d'une responsabilité vers un autre élément de l'architecture ;
- Actions sur les connecteurs : Ajouter un nouveau connecteur, modifier un connecteur existant ou supprimer un connecteur. L'action sur un connecteur est le résultat de la modification, la suppression ou l'ajout de composants.

Pour illustration, considérons la tactique de modificabilité "Abstraction de services communs" (ACS). Selon la tactique ACS, les services communs qui sont localisés dans des modules différents sont extraits et regroupés dans un nouveau module qui devra fournir ce service aux modules précédents. La Figure 2.2 illustre la représentation de haut niveau de cette tactique. Dans cette figure, les modules A et B contiennent respectivement les responsabilités A' et A", B' et B". À' et B' fournissent des variantes du même service respectivement à A" et B". La transformation appliquée par la tactique ACS est de fusionner A' et B' en un nouveau module (le module C dans la figure) paramétrable et modifier A" et B" pour utiliser le nouveau module avec les paramètres conséquents. L'application de la tactique ACS permet (Bachmann, Bass et Nord, 2007; Bass, Clements et Kazman, 2012) : 1) de réduire le coût des modifications en regroupant ensemble les services communs et ainsi localiser la modification dans un seul module, et 2) d'augmenter la cohésion des modules qui dépendent des services communs et prévenir l'effet de propagation des changements sur ces modules.

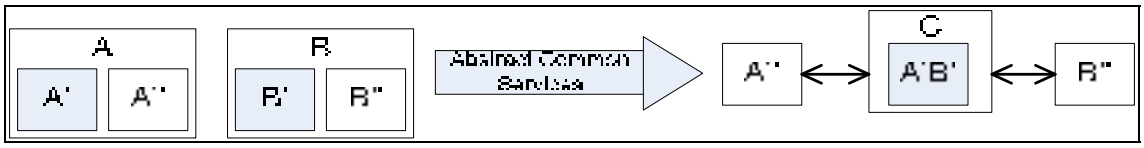


Figure 2.2 Représentation de haut niveau de l'abstraction de services communs
Adaptée de (Bachmann, Bass et Nord, 2007)

Dans le Tableau 2.1 on peut voir la représentation de haut niveau de la tactique ACS sous forme d'actions sur les composants et les connecteurs architecturaux.

Tableau 2.1 Description de haut niveau de la tactique ACS

Type d'action	**Description de haut niveau**
Actions sur les composants	Créer C
Actions sur les composants	Déplacer A' de A vers C
Actions sur les composants	Déplacer B' de B vers C
Actions sur les connecteurs	Modifier A" pour dépendre de C
Actions sur les connecteurs	Modifier B" pour dépendre de C

2.2.2 Représentation opérationnelle des tactiques

La description de haut niveau d'une tactique donne une représentation abstraite de la tactique. Il faut raffiner et contextualiser cette représentation afin d'avoir une représentation concrète de la mise en œuvre de la tactique selon le contexte du système à analyser. Cette contextualisation consiste à mettre en correspondance les composants et connecteurs architecturaux définis par une tactique avec des entités définies par le paradigme/langage de programmation utilisé dans l'implémentation du système analysé.

Dans le cadre ce mémoire, nous nous concentrons sur les systèmes orientés objet en java. La représentation opérationnelle va donc mettre correspondances les composants et les responsabilités de haut niveau avec des unités d'implémentation en java. Les composants correspondent aux paquetages et classes implémentés dans le système. Tandis que les responsabilités d'un composant correspondent à l'ensemble des attributs et méthodes implémentées par les classes quand le composant est une classe ou aux classes que le

paquetage contient quand le composant est un paquetage. Cette mise en correspondance entraîne la possibilité d'avoir plus d'une représentation concrète pour une représentation de haut niveau donnée d'une tactique. Par exemple, pour une action sur les composants architecturaux, il faut prendre en compte le cas où l'élément est mis en correspondance avec un paquetage et le cas où il est mis en correspondance avec une classe.

Les connecteurs architecturaux quant à eux ne sont pas explicitement exprimés en langage OO (Aldrich et al., 2003); mais ils sont implicitement spécifiés par les appels de méthodes, d'évènements et de références.

Partant de ces contextualisations, la représentation opérationnelle d'une tactique est un ensemble d'**actions** (ajout, suppression, modification et déplacement) sur les paquetages, classes, méthodes, attributs, appels de méthode, d'évènements et de références. Cependant, le même ensemble d'actions peut être commun à plusieurs tactiques. Par exemple, le déplacement d'un groupe de responsabilités vers un module (classe ou paquetage dans la représentation opérationnelle) peut correspondre à la tactique ACS ou à la tactique de séparation de responsabilité (SR). Toutefois, dans le cas de la tactique ACS, les responsabilités déplacées appartenaient à des modules différents avant le déplacement, tandis que dans le cas de la tactique de SR, elles appartenaient au même module. Pour distinguer ces tactiques, nous avons ajouté des **contraintes** dans les représentations opérationnelles des tactiques. Ces contraintes permettent de définir des préconditions et des post-conditions sur les éléments et actions impliqués dans la définition de la tactique.

La représentation opérationnelle d'une tactique se définit donc comme un ensemble d'actions sur les entités d'implémentation et un ensemble de contraintes sur ces entités et actions. Le Tableau 2.2 liste des exemples de représentations opérationnelles pour quatre tactiques de modificabilité.

Tableau 2.2 Exemple de représentations opérationnelles

Tirée de (Kapto et al., 2016)

Légende :
• Actions
⇐ Contraintes en mode précondition
⇒Contraintes en mode post-condition.

Tactique	Représentation opérationnelle
Abstract common services **(ACS)**	• $\mathbf{P_{dest}}$: added or existing package • **C**: moved classes to $\mathbf{P_{dest}}$ ⇐ Classes in **C** did not belong to the same package in the previous release
Abstract common services **(ACS)**	• $\mathbf{C_{dest}}$: added class or existing class • **M:** moved methods to $\mathbf{C_{dest}}$ ⇐ Methods in **M** did not belong to the same class in the previous release
Abstract common services **(ACS)**	• **Cp**: added class • Inherits **Cp**: added inheritance ⇐ All classes involved in "Inherits **Cp**" existed in the previous release ⇒ These classes belong to at least two different packages in next release.
Split responsibilities **(SR)**	• $\mathbf{P_{dest}}$: added package • **C**: moved classes to $\mathbf{P_{dest}}$ ⇐ All classes in **C** belonged to the same package in the previous release
Split responsibilities **(SR)**	• $\mathbf{C_{dest}}$: added class • **E**: moved elements (attribute and method) to $\mathbf{C_{dest}}$ ⇐ All elements in E belonged to the same class in the previous release
Use encapsulation **(UE)**	• **Cp**: added class • Inherits **Cp**: added inheritance ⇐ All classes involved in "Inherits **Cp**" existed in the previous release ⇒ All classes involved in "Inherits **Cp**" belong to the same package in the next release
Increase cohesion **(IC)**	• **C**: moved classes from $\mathbf{P_{src}}$ to $\mathbf{P_{dest}}$ ⇐ $\mathbf{P_{dest}}$ existed ⇐ All classes in **C** belonged to the same package ($\mathbf{P_{src}}$) in the previous release ⇐ Cohesion of $\mathbf{P_{src}}$ increased
Increase cohesion **(IC)**	• **E**: moved elements (attribute and method) from $\mathbf{C_{src}}$ to $\mathbf{C_{dest}}$ ⇐ $\mathbf{C_{dest}}$ existed ⇐ All elements in E belonged to the same class ($\mathbf{C_{src}}$) in the previous release ⇐ Cohesion of $\mathbf{C_{src}}$ increased

Si nous prenons par exemple la ligne 1 du Tableau 2.2, elle décrit une représentation concrète d'ACS. La ligne $\mathbf{P_{dest}}$ décrit l'action de recherche d'un paquetage $\mathbf{P_{dest}}$ ajouté ou existant. La ligne **C** décrit l'action de recherche de l'ensemble **C** de classes déplacées dans le paquetage $\mathbf{P_{dest}}$. La contrainte exprimée porte sur la position des classes déplacées dans la version de départ. Cette représentation correspond effectivement à la représentation de la tactique ACS faite dans Figure 2.2. Le module **C** de la Figure 2.2 correspond à $\mathbf{P_{dest}}$, tandis que les responsabilités **A'** et **B'** de la Figure 2.2 correspondent à l'ensemble **C** de classe de la représentation concrète.

Un second exemple ligne 7 du Tableau 2.2, elle décrit une représentation de la tactique d'UE. La ligne **Cp** décrit l'action de recherche d'une classe **Cp** qui a été ajoutée. La ligne "*Inherits **Cp***" décrit l'action de recherche de l'ensemble des classes avec qui une relation d'héritage a été ajoutée avec **Cp**. Dans cette représentation, nous avons deux contraintes exprimées, l'une portant sur la version de départ et la seconde sur la version d'arrivée. Dans la version de départ, les classes identifiée dans la ligne "*Inherits **Cp***" doivent déjà exister. Dans la version d'arrivée, les classes identifiée dans la ligne "*Inherits **Cp***" doivent appartenir au même paquetage.

2.2.3 TacMatch : outil de spécification des représentations opérationnelles des tactiques

Pour faciliter la spécification des représentations opérationnelles des tactiques par les concepteurs de logiciel, un langage de spécification formel qui ressemble au langage naturel a été mis en œuvre. Ce langage de spécification a été implémenté dans l'outil TacMatch pour permettre au concepteur de logiciel de définir ses tactiques sans avoir à connaître un langage spécifique pour faire cela. Le concepteur doit uniquement connaître les actions de la tactique (ajout, suppression, déplacement) sur les composants architecturaux et les contraintes sur ces composants et actions. TacMatch permet aussi de sauvegarder les représentations spécifiées par le concepteur.

Ainsi, pour définir les représentations opérationnelles, nous avons conçu et implémenté une interface personnalisée laquelle a été inspirée des langages de requêtes tels que SQL et QBE (Query By Exemple). La Figure 2.3 présente l'interface de TacMatch pour la définition des représentations opérationnelles des tactiques. Cette fenêtre comporte 4 sections :

(1) **Section "*Tactic Description*" :** Cette section permet d'identifier une tactique (le champ "*tactic name*") et de nommer la représentation opérationnelle de cette tactique qui sera définie (le champ "*Variant name*").

(2) **Section "*Selector*" :** Cette section permet à l'utilisateur de sélectionner les types d'actions (i.e., changements) introduits par la représentation opérationnelle en cours de définition. Un ensemble d'actions prédéfinies est fourni à l'utilisateur. Ces actions

sont présentées dans la sous-section 2.2.3.1. Le bouton (+) permet d'ajouter une action sélectionnée dans les instructions de la représentation opérationnelle.

(3) **La section "*Filter*" :** Cette section permet à l'utilisateur de spécifier des contraintes sur les éléments sélectionnés précédemment à travers le "*Selector*" ; i.e., ces contraintes correspondent à des pré- et post-conditions sur les entités sélectionnées (i.e., classes, paquetages, méthodes ou attributs). Chaque contrainte choisie peut être paramétrée à travers les champs affichés par TacMatch dans cette section. Le bouton (+) permet d'ajouter la contrainte sélectionnée dans les instructions de la représentation opérationnelle. Plusieurs contraintes peuvent être sélectionnées par l'utilisateur pour définir une représentation opérationnelle. Les contraintes sont présentées plus en détail dans la sous-section 2.2.3.2.

(4) **Section "*Preview*" :** cette section donne un aperçu de la représentation opérationnelle dans un formalisme proche du langage SQL.

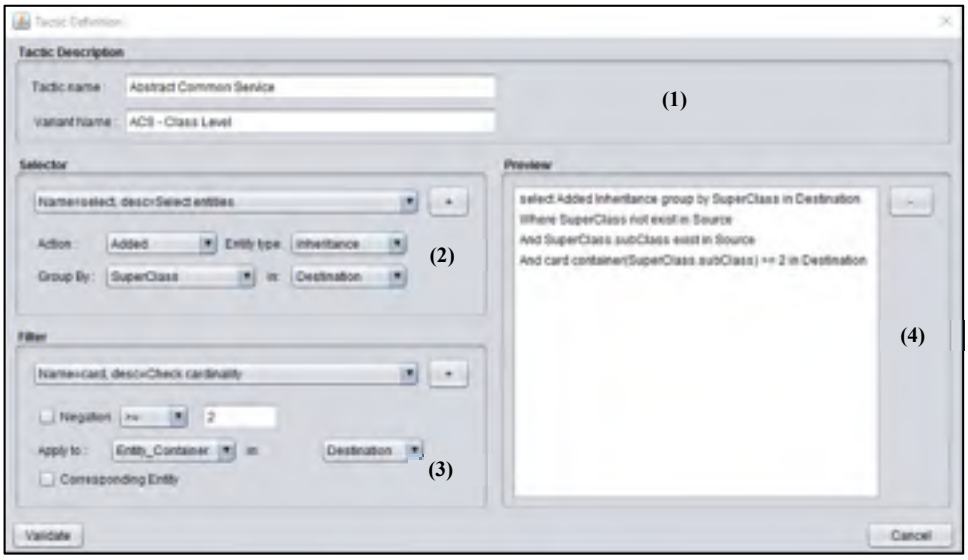


Figure 2.3 Configuration d'une représentation opérationnelle
de tactique avec TacMatch

Dans la Figure 2.3, nous avons un exemple de spécification de la représentation opérationnelle de la tactique ACS décrite à la ligne 3 du Tableau 2.2. Ainsi, la spécification d'une représentation opérationnelle est composée d'une clause "*select*" qui sélectionne des actions (i.e., changements) faites entre deux versions, suivie d'une ou plusieurs contraintes qui filtrent (i.e., réduit le nombre) des actions sélectionnées.

2.2.3.1 Sélecteur d'actions

Pour sélectionner les actions faites sur les entités du code source, une instruction a été implémentée : **select**. Cette instruction permet de sélectionner les unités d'implémentation qui correspondent aux paramètres de l'instruction. La syntaxe du sélecteur "select" est la suivante :

select *action type* ***group by*** *entiteDeGroupement* ***in*** *versionDeGroupement*

Les paramètres sont les suivants :

- *Action* : Ce paramètre définit l'action à rechercher qui s'effectue entre les deux versions. Les valeurs possibles sont : ajout, suppression ou déplacement ;
- *Type* : Ce paramètre définit le type d'entités sur lesquelles l'action s'est produite. Les valeurs possibles sont : paquetage, classe, méthode, héritage ;
- *VersionDeGroupement :* Ce paramètre définit la version dans laquelle les entités seront sélectionnées. Les valeurs possibles sont : Source ou destination ;
- *entiteDeGroupement* : Ce paramètre définit le type d'entité autour du quel vont se regrouper les entités sélectionnées. Ce groupement permet de différencier les occurrences de la tactique. Les valeurs possibles de ce paramètre sont : Classe, paquetage ou superclasse.

Exemple : ***select*** *moved class* ***group by*** *package* ***in*** *destination*

Cette instruction permet de rechercher la liste des classes dans la version de destination qui ont été déplacées et de les regrouper par paquetage. Elle correspond notamment à l'instruction de sélection de la tactique d'ACS décrite à la ligne 1 du Tableau 2.2.

2.2.3.2 Filtrage par des contraintes

La section de filtrage permet de réduire la liste des occurrences obtenue des sélecteurs, en définissant des contraintes que ces occurrences doivent vérifier. TacMatch met en œuvre 3 contraintes usuelles rencontrées dans la spécification des tactiques qui ont été étudiées : existence d'une entité, cardinalité d'un ensemble d'entités, augmentation de la cohésion d'un ensemble d'entités.

Existence d'une entité (exist) : Cette contrainte permet de garder uniquement les occurrences dont l'entité existe dans la version opposée à la version de sélection. La syntaxe est la suivante :

entConcernée ***exist in OpposéVersionDeGroupement***.

Le paramètre *entConcernée* définit quelle entité doit être vérifiée. Les valeurs possibles de ce paramètre sont : l'entité de groupement, l'entité qui subit l'action, la superclasse de l'entité qui subit l'action ou le conteneur (paquetage ou classe) de l'entité qui subit l'action.

Exemple : ***select*** *Moved Class* ***group by*** *Package* ***in*** *Destination*
Where *Package* ***exit in Source***

Dans cet exemple (qui correspond aux 2 premières instructions de la ligne 7 du Tableau 2.2.) on sélectionne les classes qui ont été déplacées et on les groupe par paquetage dans la version de destination. Ensuite on garde uniquement les groupes pour lesquelles le paquetage de destination existait dans la version source.

Cardinalité d'un ensemble d'entités (card) : Elle permet de ne garder que les occurrences dans lesquelles la cardinalité d'un groupe d'entités vérifie une condition. La syntaxe est la suivante :

card *entConcernée ope valeur* ***in*** *versionDeVerification entCorresp*.

Les paramètres sont les suivants :

- *Ope* : Ce paramètre définit l'opérateur de condition de la valeur de la cardinalité. Les valeurs possibles sont : supérieure ou égale à (>=), inférieure ou égale à (<=), égale à (==), pas égale à (<>), supérieure à (>), inférieure à (<) ;
- *Valeur* : Ce paramètre définit la valeur de la condition sous forme de nombre entier ;
- *VersionDeVerification* : Ce paramètre définit la version dans laquelle le calcul de la cardinalité devra s'effectuer. Les valeurs possibles sont : Source ou destination ;
- *EntConcernée* : Ce paramètre définit l'entité sur laquelle le calcul de la cardinalité va s'effectuer. Les valeurs possibles sont : l'entité de groupement, les entités ayant subi l'action, les superclasses des entités ayant subi l'action ou le conteneur (paquetage ou classe) des entités ayant subi l'action ;
- *EntCorresp* : Ce paramètre définit si la cardinalité se calcule sur les unités d'implémentation choisie (No) ou leur correspondant dans la version vis-à-vis (Yes). Il est optionnel.

Exemple : ***select*** *moved class* ***group by*** *package* ***in*** *destination*

Where card *container(Class) >=2* ***in*** *Source*

Dans cet exemple (qui correspond la ligne 1 du Tableau 2.2.) on sélectionne les classes qui ont été déplacées et on les groupe par paquetage dans la version de destination. Ensuite on garde uniquement les groupes pour lesquelles les classes proviennent d'au moins 2 paquetages dans la version source.

Augmentation de la cohésion d'un ensemble d'entités (coh) : Elle permet garder uniquement les occurrences dans lesquelles on observe une augmentation de la cohésion. La syntaxe est la suivante :

coh *entConcernée* ***increase in*** *versionDeVerification*.

Les paramètres sont les suivants :

- *VersionDeVerification :* ce paramètre définit la version dans laquelle la vérification devra s'effectuer. Les valeurs possibles sont : Source ou destination

- *EntConcernée* : ce paramètre définit l'entité dont l'augmentation de la cohésion sera vérifiée. Les valeurs possibles sont : l'entité de groupement, toutes les entités ayant subi l'action, toutes les superclasses des entités ayant subi l'action ou tous les conteneurs (paquetage ou classe) des entités ayant subi l'action.

Exemple : ***select*** *Moved Class* ***group by*** *Package* ***in*** *Destination*
Where *Package* ***exit in Source***
And card *container(Class) == 1* ***in*** *Source*
And coh *container(Class)* ***increase in*** *Source* .

Dans cet exemple (ligne 7 du Tableau 2.2.) on sélectionne les classes qui ont été déplacées et on les groupe par paquetage dans la version de destination. Ensuite on garde uniquement les groupes pour lesquelles le paquetage de destination existait dans la version source. Des groupes de classes obtenues de la précédente contrainte, on garde uniquement ceux dont toutes les classes déplacées proviennent du même paquetage dans la version source. Pour finir on garde uniquement les groupes dont la cohésion de ce paquetage de la version source a augmenté.

Pour permettre d'exprimer la négation d'une contrainte, il est possible de mettre le symbole "*!*" devant l'instruction. Cette négation se fait sur l'ensemble de l'instruction et non sur les paramètres pris un à un (*!Filtre parametre1 parametre2 ... ≠ Filtre !parametre1 !parametre2 !*...).

2.3 Analyse de l'évolution de l'architecture

2.3.1 Extraction des deltas entre les versions

Dans la section précédente, nous avons vu que l'approche de détection se base sur l'identification de groupes de changements/d'actions élémentaires entre deux versions du système et la définition de contraintes sur les entités ayant subits ces changements. La mise en œuvre de notre approche se base sur un outil qui prend en entrée deux versions d'un système et qui fournit en sortie la liste des changements subits (delta) par le système entre ces

deux versions. MADMatch (Kpodjedo et al., 2013) a été choisi pour ce travail du fait de sa capacité à retrouver des correspondances plusieurs à plusieurs entre les versions.

2.3.1.1 TacMatch : Outil d'automatisation de l'extraction et de la sauvegarde des deltas

L'exécution de la tâche d'extraction des deltas a été automatisée dans **TacMatch**. Ce dernier fournit des interfaces pour l'enregistrement des représentations des versions analysées fournies par Ptidej et le lancement de la comparaison de ces représentations avec MADMatch. Derrière ces interfaces, une base de données est mise en place pour enregistrer les représentations des versions ainsi que les changements identifiés par MADMatch. Cette base de données sera interrogée lors de l'étape de détection des occurrences de tactiques. Le modèle physique de données associé à cette base de données est illustré dans la Figure 2.4.

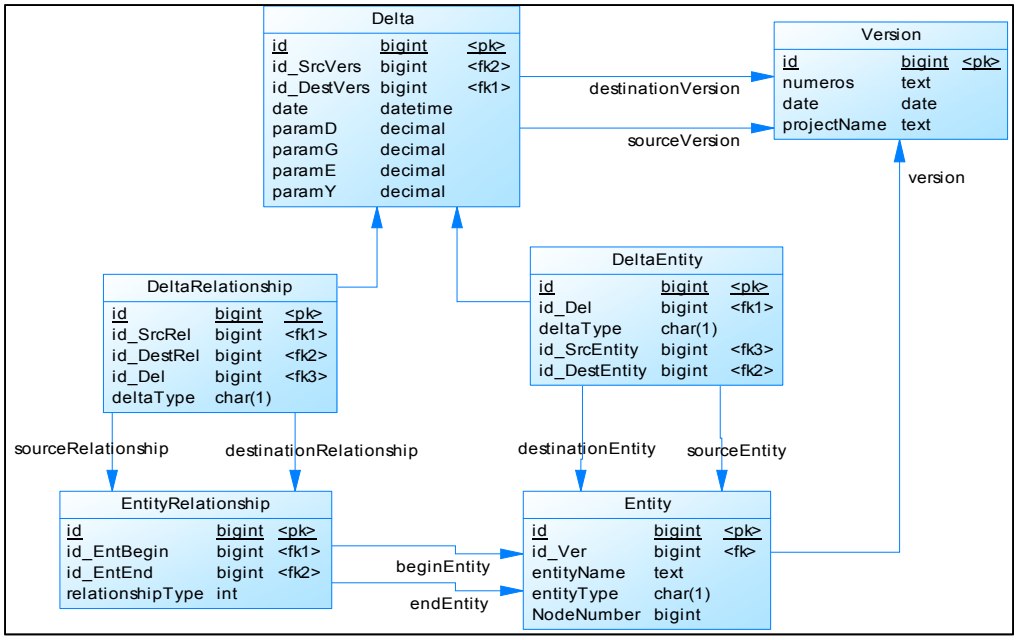


Figure 2.4 Modèle physique de donnée de la gestion des versions et des deltas

Dans la Figure 2.4, les versions du système sont sauvegardées dans les tables "*Version*" (identifie la version du système), "*Entity*" (identifie les entités du système, à savoir, paquetage, classe, méthode ou attribut) et "*EntityRelationShip*" (enregistre la relation entre

les entités en se utilisant les mêmes types que ceux adoptés par MADMatch). Les changements entre versions sont sauvegardés dans les tables suivantes :

- "*Delta*" : cette table permet d'enregistrer les caractéristiques d'une exécution de MADMatch pour comparer les versions. On y retrouve la version de départ ("*id_SrcVers*") et la version d'arrivée ("*id_DestVers*"), de même que les paramètres de l'exécution de MADMatch ("*paramD*", "*paramG*", "*paramE*", "*paramY*") ;
- "*DeltaEntity*" : cette table enregistre les changements observés entre les deux versions au niveau des entités. On y retrouve l'entité dans la version de départ ("*id_SrcEntity*") et l'entité dans la version d'arrivée ("*id_DestEntity*"), de même que le type de changement observé ("*deltaType*"). Les valeurs possibles pour le type de changement sont : ajouté, supprimé, modifié, identique ;
- "*DeltaRelationShip*" : cette table enregistre les changements observés entre les deux versions au niveau des relations entre entités. On y retrouve la relation dans la version de départ ("*id_SrcRel*") et la relation dans la version d'arrivée ("*id_DestRel*"), de même que le type de changement observé ("*deltaType*"). Les valeurs possibles pour le type de changement sont identiques à celles du type de changement de la table "*DeltaEntity*".

2.3.2 TacMatch : Outil de détection des occurrences de l'application ou annulation des tactiques

Prenant en entrée une représentation opérationnelle d'une tactique et un delta entre 2 versions données, TacMatch génère à la volée l'algorithme associé à cette représentation opérationnelle et exécute cet algorithme sur le delta pour donner en sortie des occurrences possibles de la tactique. Une occurrence retournée par TacMatch regroupe un ensemble de changements qui est conforme aux actions et contraintes définies par la représentation opérationnelle considérée.

TacMatch s'appuie sur un ensemble de classes qui lisent la spécification de la représentation opérationnelle d'une tactique et générèrent différentes parties de **l'algorithme** de détection de la tactique. La Figure 2.5 donne un aperçu des classes centrales de TacMatch, lesquelles

ont été organisées selon le patron de conception chaîne de responsabilité (CoR)(Gamma et al., 1995). La clause "*select*" de la représentation opérationnelle est implémentée dans la classe "*Selector*" : Cette classe accède à la base de données pour sélectionner les occurrences (actions ou changements) qui répondent aux critères du sélecteur. Le type "*Filter*" définit une interface pour filtrer les occurrences sélectionnées selon une contrainte. Chaque type de contraintes, tel que présenté dans la section 2.2.3.2, est représenté par une classe qui implémente cette interface. La Figure 2.5 montre deux classes filtres *Cardinality* et *Existence*, correspondant respectivement aux contraintes de cardinalité (card) et existence (exist).

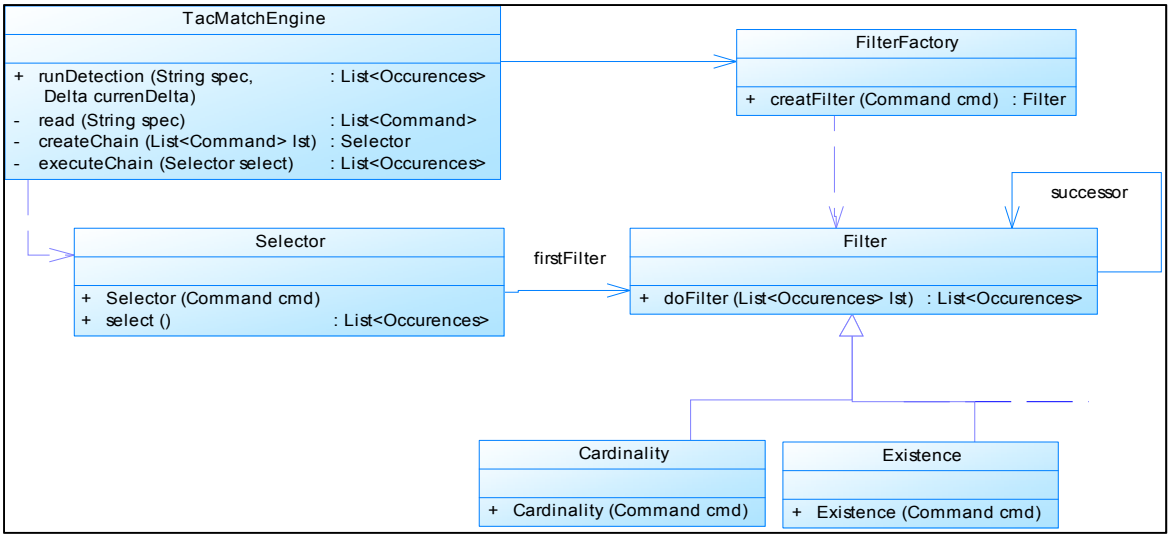


Figure 2.5 Génération des algorithmes de détection.
Tirée de (Kapto et al., 2016)

Le patron CoR a été choisi pour permettre de configurer et d'instancier à l'exécution les filtres correspondant à la représentation opérationnelle considérée. Il permet aussi d'ajouter facilement de nouveaux filtres (i.e., contraintes).

L'Algorithme 2.1 donne un aperçu de la façon dont TacMatch génère et exécute un algorithme à partir d'une représentation concrète d'une tactique. Le point d'entrée de TacMatch est la classe "*TacMatchEngine*" qui lit la représentation opérationnelle d'une

tactique et la traduit en une liste ordonnée de commandes où chaque commande correspond à une ligne de la représentation opérationnelle (Ligne 2 de l'Algorithme 2.1). Cette liste de commandes est utilisée pour créer une chaîne d'objets qui commence par une instance de la classe "*Selector*" suivie d'une chaîne appropriée d'instances de filtre (lignes 9 à 25 de l'Algorithme 2.1). Ce travail est effectué par la méthode "*TacMatchEngine.createChain*" qui utilise l'objet "*FilterFactory*" pour créer dynamiquement des objets de type filtre. Les objets de type *Selector* et *Filter* ont leur propres attributs qui sont initialisés durant leurs instanciations en utilisant le paramètre « *cmd : Command* » passé à leur constructeurs respectifs.
Par exemple, pour la tactique ACS de la ligne 1 du Tableau 2.2, l'objet de type *Selector* sera instancié avec la commande « ***select*** *moved class* ***group by*** *package* ***in*** *destination* » et l'objet de type *Filter* qui lui est lié avec la commande « ***card*** *container(Class) >=2* ***in*** *Source* ».

La méthode "*TacMatchEngine.executeChain*" permet d'exécuter la chaîne d'objets instanciée précédemment (lignes 26 à 34 de de l'Algorithme 2.1). Elle prend en paramètre l'objet sélecteur et appelle sa méthode "*select*". Cette dernière sélectionne les occurrences appropriées d'actions/changements dans le delta considéré, et elle les passe comme paramètre à la méthode "*doFilter*" du premier filtre référencé par le sélecteur. La méthode "*doFilter*" réduit la liste de occurrences en fonction de la contrainte implémentée par le filtre et passe la liste résultante des occurrences à son suivant (successeur) par appel de la méthode "*doFilter*" de ce dernier. Le parcours de la chaîne s'effectue de cette façon jusqu'au dernier filtre.

Algorithme 2.1 Génération et exécution de l'algorithme de détection

```
 1: + TacMatchEngine.runDetection (spec: String, curentDelta Delta) : List<Occurences>{
 2:     List<Commande> lstCommand = TacMatchEngine.read(spec);
 3:     //Initialization of the chain of selector and filters
 4:     Selector select = TacMatchEngine.createChain(lstCommand);
 5:     // Run the chain in order to obtain candidate occurrences
 6:     List<Occurences> lstOccurences = TacMatchEngine.executeChain(select);
 7:     Return lstOccurences;
 8: }
 9: - TacMatchEngine. createChain (List<Commande> lstCommand) : Selector {
10:     Selector select;
11:     Filter previousFilter;
12:     Int counter = 0;
13:     While (counter < lstCommand.size()){
14:             String command = lstCommand(counter));
15:             If (counter==0) select = new Selector (command);
16:             Else {
17:                     Filter filter = FilterFactory.createFilter(command);
18:                     If (counter ==1) select.firstFilter = filter;
19:                     Else previousFilter.successor = filter;
20:                     previousFilter=filter;
21:             }
22:             Counter ++;
23:     }
24:     return select;
25: }
26: - TacMatchEngine. executeChain (Selector select) : List<Occurences> {
27:     List<Occurences> lstOccurences = select.select();
28:     Filter nextFilter = select.firstFilter();
29:     Do {
30:             lstOccurences = nextFilter.doFilter(lstOccurences);
31:             nextFilter = nextFilter.successor;
32:     }while (nextFilter not null);
33:     Return lstOccurences;
34: }
```

TacMatch se base sur la représentation opérationnelle définie pour une tactique pour générer automatiquement une représentation de l'annulation de cette tactique. Cette annulation se déduit tout simplement par l'inversion de la source et la destination des différentes actions et contraintes utilisées dans la représentation opérationnelle. Par exemple pour un sélecteur de classes ajoutées dans la version de destination, l'inverse sera un sélecteur de classes supprimées dans la version source. Le Tableau 2.3 résume les fonctions inverses des actions et contraintes implémentées.

Tableau 2.3 Fonctions inverses de chaque instruction de la représentation opérationnelle

Instruction	**Fonction inverse ($f_{inverse}$)**
select action type **group by** entiteDeGroupement **in** versionDeGroupement	**select** $f_{inverse}$(action) type **group by** entiteDeGroupement **in** $f_{inverse}$(versionDeGroupement)
- Action	$f_{inverse}$(added) = deleted $f_{inverse}$(deleted)=added $f_{inverse}$(moved)=moved
- versionDeGroupement / versionDeVerification	$f_{inverse}$(Source)= Destination $f_{inverse}$(Destination)=Source
entConcernée **exist in** **OpposéVersionDeGroupement**.	entConcernée **exist in** **OpposéVersionDeGroupement**.
card entConcernée ope valeur **in** versionDeVerification entCorresp.	**card** entConcernée ope valeur **in** $f_{inverse}$(versionDeVerification) entCorresp.
coh entConcernée **increase in** versionDeVerification	**coh** entConcernée **increase in** $f_{inverse}$(versionDeVerification)

Se basant sur le tableau ci-dessus, la représentation opérationnelle de l'annulation de la tactique ACS illustrée à la Figure 2.3 est présentée dans le Tableau 2.4.

Tableau 2.4 Exemple de représentation d'annulation de la tactique ACS

Application ACS	**Annulation ACS**
select Added Inheritance **group by** SuperClass **in** Destination **Where** SuperClass **not exist in** Source **And** SuperClass.subClass **exist in** Source **And** card container(SuperClass.subClass) >= 2 **in** Destination	**select** deleted Inheritance **group by** SuperClass **in** *Source* **Where** SuperClass **not exist** in *Destination* **And** SuperClass.subClass **exist in** *Destination* **And** card container(SuperClass.subClass) >= 2 **in** *Source*

TacMatch permet à l'utilisateur de consulter les occurrences détectées avec la représentation de la tactique. Cette fenêtre de consultation est présentée dans la Figure 2.6. Cette fenêtre se divise en 3 sections :

(i) Cette section identifie le projet et les versions impliquées dans la détection.

(ii) Cette section identifie la tactique et donne un rappel de la représentation opérationnelle qui a permis la détection des occurrences.

(iii) Cette section présente les détails d'une occurrence sélectionnée par l'utilisateur ;

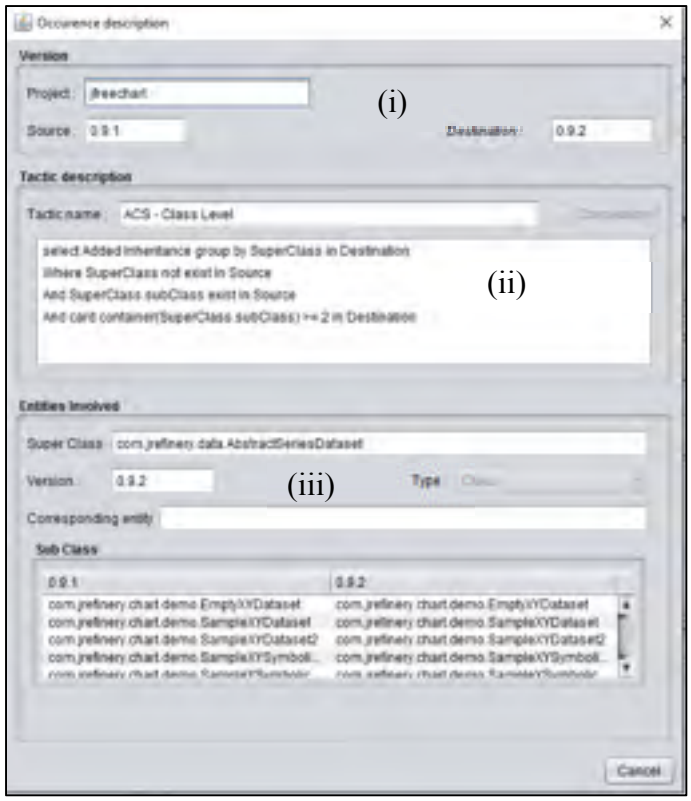


Figure 2.6 Interface de présentation d'une occurrence
de la tactique ACS

Par exemple dans la Figure 2.6, l'utilisateur visualise une des occurrences détectées pour la tactique ACS présentée à la Figure 2.3. La détection a été faite entre les versions 0.9.1 et 0.9.2 du projet jFreeChart. Dans la section (iii) on peut voir la classe qui a été créée (le champ "*Super Class*") et les classes pré existantes qui héritent de cette classe (le tableau "*Sub Class*").

2.3.3 Analyse des occurrences

Avec les occurrences ainsi détectées et extraites, le concepteur peut effectuer toute sorte d'analyse en fonction de la prise en compte ou non d'un attribut de qualité sur le système. Le CHAPITRE 3 présente des analyses qui ont été effectuées sur un projet libre Java dans le cadre de l'expérimentation. Ces analyses représentent des exemples d'analyse qu'un concepteur pourrait faire en fonction de ses besoins.

2.4 Conclusion

Ce chapitre a présenté l'approche proposée pour inférer l'évolution l'architecturale d'un système à partir de son code source. Cette approche comporte 2 étapes :

(i) La spécification des représentations opérationnelles des tactiques selon le contexte du projet;

(ii) L'analyse de l'évolution architecturale qui se base sur l'extraction des changements (delta) entre les différentes versions du système analysé, la génération d'algorithmes de détection à partir des représentations opérationnelles des tactiques et l'exécution de ces algorithmes sur le delta considéré pour détecter des occurrences des tactiques considérées.

Nous avons aussi présenté l'outil TacMatch qui implémente notre approche. Cet outil supporte l'utilisateur dans la spécification des représentations opérationnelles des tactiques et il permet l'automatisation de l'extraction des deltas, de la génération des algorithmes de détection des tactiques, et de l'exécution de ces algorithmes.

CHAPITRE 3

EXPÉRIMENTATION

Ce chapitre présente l'expérimentation que nous avons réalisée pour évaluer l'approche proposée pour inférer l'évolution architecturale à partir des codes sources. Ce chapitre commence par la présentation du contexte de l'expérimentation incluant les questions de recherches auxquelles devra répondre l'expérimentation, les tactiques architecturales visées par l'expérimentation et le projet choisi comme sujet de l'expérimentation. Nous présentons aussi les résultats obtenus de cette expérimentation et nous analysons ces résultats à la lumière de nos questions de recherche. Enfin, nous discutons des limites de l'expérimentation et de la validité des résultats.

3.1 Définition de l'expérimentation

3.1.1 Questions de recherche

Notre expérimentation a pour objectif de répondre aux questions suivantes de recherche :

Question 1 (Q1) : Quelle est l'efficacité de l'approche proposée pour la détection des tactiques ?

Pour répondre à cette question, nous avons utilisé notre outil TacMatch pour analyser plusieurs versions d'un projet libre Java et détecter un ensemble de tactiques architecturales. Nous avons calculé la précision et le rappel des résultats obtenus. Pour ce faire, nous avons réalisé une analyse manuelle du code source et des deltas générés par MADMatch pour confirmer ou invalider les occurrences détectées pour TacMatch et pour identifier les occurrences qui n'ont pas été détectées par TacMatch.

Question 2 (Q2) : Sommes-nous en mesure de déduire une tendance dans l'évolution de l'architecture en utilisant TacMatch ?

Pour répondre à cette question, nous avons analysé les résultats obtenus de deux façons. Tout d'abord, nous avons analysé et comparé les occurrences détectées d'applications et d'annulations de tactiques pour vérifier si les modifications faites au système étudié suivent un modèle cohérent d'évolution architecturale. Ensuite, nous avons essayé d'inférer une tendance architecturale en comparant les résultats obtenus du processus de détection lorsque ce dernier est appliqué à des versions majeures, des versions mineures et des révisions.

3.1.2 Choix de l'attribut de qualité

Comme présenté dans la section 1.1, il existe 7 principaux attributs de qualité : disponibilité, interopérabilité, modificabilité, performance, sécurité, testabilité, et utilisabilité (Bass, Clements et Kazman, 2012). Étant donné que notre approche se base sur la différence entre versions au niveau du code source, nous avons choisi un attribut qui se caractérise par des actions élémentaires sur les éléments du code (ex. ajout/suppression/déplacement de paquetage/classe/méthode). Notre choix s'est porté sur la **modificabilité** parce qu'il est un attribut de qualité qui présente le plus de tactiques ayant des impacts observables dans le code source.

Nous rappelons que la **modificabilité** (Bachmann, Bass et Nord, 2007) est la capacité d'un système à subir des modifications à faible coût. Dans ce contexte, les modifications sont celles qui ont lieu uniquement entre la phase de conception et le déploiement de l'application. Elles sont effectuées par les développeurs et les architectes sur le système.

3.1.3 Choix des tactiques

Le choix des tactiques de la modificabilité a été fait suivant la même logique que celui de l'attribut de qualité modificabilité. Les tactiques ayant un impact sur le code source ont été choisies. Autrement dit, nous nous sommes concentrés sur les tactiques de modificabilité qui impliquent des actions qu'on peut détecter par une analyse statique de différentes versions

d'un système. Ce choix a exclu les tactiques qui répondent à la préoccupation d'anticiper les modifications tardives parce que ces tactiques utilisent essentiellement des configurations externes au code source pour passer des paramètres lors de l'exécution du système.

Ainsi, l'expérimentation porte sur 4 tactiques de la modificabilité à savoir : **séparation de responsabilités (SR), augmentation de cohésion (IC : Increase Cohesion), encapsulation (UE : Use encapsulation) et abstraction des services communs (ACS : Abstract Common Services)**. Ces tactiques ont été décrites dans la section 1.1.

3.1.4 Choix du projet

Le projet utilisé pour l'expérimentation doit remplir les critères suivants :

(i) Ses codes sources doivent être disponibles ;
(ii) Le projet doit avoir un minimum de 10 versions ;
(iii)Le projet doit être réalisé en Java;
(iv)L'équipe qui effectue l'expérimentation doit avoir une bonne connaissance du projet.

En accord avec ces critères, le choix s'est porté sur le projet **jFreeChart** (Gilbert et Morgner, 2007). jFreeChart a aussi été l'objet des expérimentations faites avec l'outil de génération des deltas (MADMatch), ce qui nous donne une base de connaissance sur le projet et de comparaison des résultats.

JFreeChart est un projet libre Java, son objectif est de fournir une bibliothèque d'outils aux développeurs pour intégrer des diagrammes et graphes dans leurs applications. L'expérimentation porte sur 37 versions de jFreeChart à savoir de la version 0.5.6 à la version 1.0.6. Les tailles de ces versions varient de 26 à 141 paquetages et de 100 à 1198 classes.

Le schéma de numérotation des versions dans jFreeChart utilise une séquence de trois chiffres "M.m.r". Dans ce schéma "M" représente une version majeure et il est incrémenté

lorsque des changements significatifs ont été effectués sur le système. La seconde position, à savoir "m", correspond à une version mineure et est incrémentée pour des changements mineurs du système ou des corrections d'erreurs significatives. La dernière position ("r") correspond au numéro de révision qui est incrémenté suite à des corrections d'erreurs mineures.

3.2 Résultats bruts

L'exécution de l'approche sur les 36 deltas de jFreeChart a permis de détecter 103 occurrences d'applications de tactiques et 33 occurrences d'annulations de tactique. La distribution de ces occurrences par delta est présentée dans le Tableau 3.1.

Tableau 3.1 Distribution des occurrences détectées par deltas extraits de versions successives

Delta	Application	Annulation	Total	Delta	Application	Annulation	Total
v0.5.6_v0.6.0	4	0	4	v0.9.9_v0.9.10	2	1	3
v0.6.0_v0.7.0	0	0	0	v0.9.10_v0.9.11	1	0	1
v0.7.0_v0.7.1	1	1	2	v0.9.11_v0.9.12	5	0	5
v0.7.1_v0.7.2	0	0	0	v0.9.12_v0.9.13	3	0	3
v0.7.2_v0.7.3	0	0	0	v0.9.13_v0.9.14	2	1	3
v0.7.3_v0.7.4	3	0	3	v0.9.14_v0.9.15	3	0	3
v0.7.4_v0.8.0	1	0	1	v0.9.15_v0.9.16	1	1	2
v0.8.0_v0.8.1	1	0	1	v0.9.16_v0.9.17	8	1	9
v0.8.1_v0.9.0	6	5	11	v0.9.17_v0.9.18	4	0	4
v0.9.0_v0.9.1	0	1	1	v0.9.18_v0.9.19	5	3	8
v0.9.1_v0.9.2	3	0	3	v0.9.19_v0.9.20	2	0	2
v0.9.2_v0.9.3	1	0	1	v0.9.20_v0.9.21	21	5	26
v0.9.3_v0.9.4	0	0	0	v1.0.0_v1.0.1	0	0	0
v0.9.4_v0.9.5	13	2	15	v1.0.1_v1.0.2	0	0	0
v0.9.5_v0.9.6	0	0	0	v1.0.2_v1.0.3	1	0	1
v0.9.6_v0.9.7	6	0	6	v1.0.3_v1.0.4	0	0	0
v0.9.7_v0.9.8	0	0	0	v1.0.4_v1.0.5	1	0	1
v0.9.8_v0.9.9	4	12	16	v1.0.5_v1.0.6	1	0	1
				TOTAL	**103**	**33**	**136**

Le Tableau 3.2 et le Tableau 3.3 présentent respectivement les distributions des occurrences d'applications et d'annulations détectées par tactique et par delta. Il est à noter que les deltas pour lesquels aucune occurrence n'a été détectée ne sont pas listés dans ces tableaux ainsi que tous les tableaux suivants de ce chapitre.

Tableau 3.2 Distribution des applications par delta et tactique

Delta	SR	UE	ACS	IC	Delta	SR	UE	ACS	IC
v0.5.6_v0.6.0	1	2	1	0	v0.9.11_v0.9.12	1	1	1	2
v0.7.0_v0.7.1	1	0	0	0	v0.9.12_v0.9.13	1	2	0	0
v0.7.3_v0.7.4	1	2	0	0	v0.9.13_v0.9.14	0	0	2	0
v0.7.4_v0.8.0	0	1	0	0	v0.9.14_v0.9.15	1	2	0	0
v0.8.0_v0.8.1	0	1	0	0	v0.9.15_v0.9.16	1	0	0	0
v0.8.1_v0.9.0	0	4	1	1	v0.9.16_v0.9.17	0	3	0	5
v0.9.1_v0.9.2	0	0	3	0	v0.9.17_v0.9.18	0	3	0	1
v0.9.2_v0.9.3	0	1	0	0	v0.9.18_v0.9.19	0	3	2	0
v0.9.4_v0.9.5	3	6	2	2	v0.9.19_v0.9.20	0	0	2	0
v0.9.6_v0.9.7	1	4	0	1	v0.9.20_v0.9.21	11	4	2	4
v0.9.8_v0.9.9	1	1	0	2	v1.0.2_v1.0.3	0	0	1	0
v0.9.9_v0.9.10	0	1	1	0	v1.0.4_v1.0.5	0	1	0	0
v0.9.10_v0.9.11	0	1	0	0	v1.0.5_v1.0.6	0	1	0	0
					TOTAL	**23**	**44**	**18**	**18**

Tableau 3.3 Distribution des annulations par delta et tactique

Delta	SR	UE	ACS	IC	Delta	SR	UE	ACS	IC
v0.7.0_v0.7.1	0	1	0	0	v0.9.13_v0.9.14	0	1	0	0
v0.8.1_v0.9.0	1	3	1	0	v0.9.15_v0.9.16	0	0	1	0
v0.9.0_v0.9.1	0	0	1	0	v0.9.16_v0.9.17	0	1	0	0
v0.9.4_v0.9.5	0	2	0	0	v0.9.18_v0.9.19	0	2	1	0
v0.9.8_v0.9.9	0	12	0	0	v0.9.20_v0.9.21	1	2	2	0
v0.9.9_v0.9.10	0	1	0	0	**TOTAL**	**2**	**25**	**6**	**0**

3.3 (Q1) Quelle est l'efficacité de l'approche proposée pour la détection des tactiques ?

Pour répondre à notre première question de recherche, nous avons calculé la précision et le rappel des résultats obtenus. Pour ce faire, nous avons analysé les occurrences détectées pour déterminer si elles sont effectivement des vraies occurrences de tactiques et nous avons identifié les occurrences de tactiques qui n'ont pas été détectées par TacMatch.

Ces vérifications sont faites par des analyses manuelles des codes sources du projet analysé et des sorties de MADMatch. La vérification des occurrences détectées consiste à prendre chaque occurrence et à vérifier que les actions décrites dans la formalisation de la tactique détectée par TacMatch ont effectivement eu lieu dans le code source. L'identification des occurrences non détectées par TacMatch s'effectue par le regroupement manuel des actions des deltas fournis par MADMatch qui correspondent à des formalisations de tactiques et la vérification que ces groupements ont été détectés, ou non, par TacMatch comme des occurrences de tactique.

Les résultats de ces vérifications sont regroupés dans le Tableau 3.4. Concernant les occurrences d'applications de tactiques, nous avons pu confirmer que 85 des 103 occurrences détectées sont de vraies occurrences (i.e., vrais positifs) ; ce qui résulte en une précision de 82.52%. Nous avons aussi identifié 3 occurrences d'applications de tactiques qui n'ont pas été détectées par TacMatch ; ce qui donne un rappel de 96.59%. Pour les annulations de tactiques, seulement 19 des 33 occurrences détectées sont de vrais positifs, résultant en une précision de 57.57%. Cependant, notre analyse manuelle n'a révélé aucun faux négatif donnant un rappel de 100%. Ces résultats suggèrent que nos représentations opérationnelles des tactiques permettent de détecter des occurrences d'applications de tactiques, mais elles ne sont pas efficaces ou suffisantes pour détecter des occurrences d'annulations de tactiques.

En effet, notre technique de générer la représentation opérationnelle de l'annulation d'une tactique en inversant simplement la source et la destination des différentes actions et contraintes utilisées dans la représentation opérationnelle de la tactique n'est pas adéquate pour caractériser précisément les annulations. Le patron inverse d'une tactique peut donner un grand nombre de faux positifs ou coïncider avec une autre tactique donnant lieu à des interprétations erronées. Par exemple, entre les versions 0.7.0 et 0.7.1, la tactique SR a été détectée par TacMatch avec le déplacement de certaines classes du paquetage *com.jrefinery.chart* vers le nouveau paquetage *com.jrefinery.chart.combination*. Cependant, lors du passage de la version 0.8.1 à la version 0.9.0, le paquetage *com.jrefinery.chart.combination* a été supprimé et les classes qu'il contenait déplacé vers 2 paquetages (*com.jrefinery.chart* et *com.jrefinery.data*). Ce qui a été identifié par TacMatch comme une annulation d'ACS. En effet, la SR détectée entre les versions 0.7.0 et 0.7.1 peut être considérée comme une étape dans l'application d'une ACS qui a été progressivement intégrée dans le système entre les versions 0.7.0 et 0.8.1 pour être finalement annulée lors du passage à la version 0.9.0. Des travaux futurs sont nécessaires pour définir précisément les relations entre les applications de tactiques et les annulations de tactiques.

Tableau 3.4 Précision et Rappel des résultats obtenus par TacMatch

	Vrais positifs	**Faux positifs**	**Faux négatifs**	**Précision**	**Rappel**
Application	85	18	3	82.52%	96.59%
Annulation	19	14	0	57.57%	100%

L'étude des faux positifs et faux négatifs a permis de constater que l'efficacité de l'approche est influencée par des facteurs externes : l'outil de rétro-ingénierie de la description des versions et l'outil de générations des deltas.

15% des faux résultats sont dû à l'outil de rétro-ingénierie de la description des versions à savoir PtiDej. Par exemple, entre les versions 0.9.20 et 1.0.0, TacMatch identifie une annulation d'ACS par la suppression du paquetage *org.jfree.chart.ui*. Cependant dans les

codes sources de la version 1.0.0 ce paquetage existe, mais PtiDej ne l'a pas identifié dans la description de la version qu'il a extraite.

Par contre, 85% des faux résultats sont dû à l'outil de génération des deltas (MADMatch). MADMatch utilise des paramètres en entrée de son processus pour spécifier si ce dernier doit privilégier la précision ou le rappel. Dans le contexte de notre expérimentation, les paramètres privilégiant le rappel par rapport à la précision ont été choisis pour permettre d'avoir un maximum de données pour les valider avec l'analyse manuelle.

3.4 (Q2) Sommes-nous en mesure de déduire une tendance dans l'évolution de l'architecture en utilisant TacMatch ?

Pour identifier des tendances dans l'évolution de l'architecture en se basant sur les résultats de l'approche, deux (2) analyses ont été effectuées sur les données obtenues à partir de notre expérimentation.

3.4.1 Analyse des relations entre les applications et les annulations de tactiques détectées

La première analyse de tendance sur l'évolution de l'architecture en utilisant TacMatch visait à comprendre la relation entre les applications et les annulations de tactiques détectées. Le Tableau 3.5 présente la distribution des applications et annulations réelles (vrais positifs). Sur un plan quantitatif, si l'on considère l'ensemble des occurrences réelles détectées à travers l'ensemble des deltas analysés, les annulations de tactiques représentent 22% des applications de tactiques. Pour comprendre la raison de ce pourcentage élevé, nous avons analysé les annulations réelles détectées. Notre analyse a révélé que sur les 19 annulations détectées, 11 annulations sont reliées à des tactiques qui étaient présentes déjà dans la première version disponible de jFreeChart (0.5.6) alors que 8 annulations sont reliées à des tactiques qui ont été introduites durant les versions subséquentes. Des exemples de ces annulations sont présentés dans les deux sous-sections suivantes.

Tableau 3.5 Distribution des applications et annulations
réelles par deltas extraits de versions successives et tactiques

Delta	Application de tactique				Annulation de tactique			
	SR	UE	ACS	IC	SR	UE	ACS	IC
v0.5.6 v0.6.0	1	2	1					
v0.7.0 v0.7.1	1							
v0.7.3 v0.7.4	1	2						
v0.7.4 v0.8.0		1						
v0.8.0 v0.8.1		1						
v0.8.1 v0.9.0		3	1	1	1	1	1	
v0.9.1 v0.9.2			1					
v0.9.2 v0.9.3		1						
v0.9.4 v0.9.5	3	5	2			1		
v0.9.6 v0.9.7	1	4		1				
v0.9.8 v0.9.9	1	1		1		6		
v0.9.9 v0.9.10		1	1			1		
v0.9.11 v0.9.12	1	1	1	2				
v0.9.12 v0.9.13	1	2						
v0.9.13 v0.9.14			2			1		
v0.9.14 v0.9.15	1	1						
v0.9.15 v0.9.16	1						1	
v0.9.16 v0.9.17		2		5				
v0.9.18 v0.9.19		3	2			2	1	
v0.9.19 v0.9.20			1					
v0.9.20 v1.0.0	9	3	2	4		2	1	
v1.0.2 v1.0.3			1					
v1.0.4 v1.0.5		1						
v1.0.5 v1.0.6		1						

3.4.1.1 Analyse des annulations de tactiques préexistantes dans le système

Notre approche considère une tactique comme une transformation du système, c'est-à-dire une modification qui affecte des éléments déjà existants dans le système. De ce fait, lorsque l'ensemble des éléments correspondant à la structure d'une tactique est ajouté lors d'un changement de version, la tactique n'est pas détectée par TacMatch du fait qu'elle ne transforme pas des éléments existants dans le système, mais l'étend au contraire. Cependant, l'annulation de cette tactique plus tard, dans une version subséquente, affecte des éléments existants du système et elle est donc détectée par TacMatch. C'est le cas des 11 annulations mentionnées plus haut. Par exemple, entre les versions 0.9.8 et 0.9.9, la superclasse

org.jfree.chart.annotations.Annotation a été supprimée, mais pas ses sous-classes, ce qui correspond effectivement à une annulation d'UE telle que détectée par TacMatch. Cependant, entre les versions 0.9.2 et 0.9.3 la superclasse Annotation a été ajoutée en même temps que ses sous-classes. Ce qui fait que TacMatch n'a pas détecté ce changement comme une application de la tactique d'UE vu que ce changement ne transformait pas le système, mais l'étendait.

3.4.1.2 Analyse des annulations de tactiques mises en correspondance avec des applications de tactiques

L'approche décrit une occurrence de tactique comme un ensemble d'entités ayant subi les actions et respectant les contraintes décrites dans la représentation de la tactique. Pour chacune des occurrences, l'approche regroupe l'ensemble des entités de l'occurrence autour d'une entité qui va identifier de façon unique l'occurrence de la tactique. La mise en correspondance des annulations et des applications se fait par la mise en correspondance d'une occurrence d'annulation avec l'occurrence d'application ayant la même entité d'identification.

Parmi les 8 annulations mises en correspondance avec des applications, 5 sont des annulations de tactiques dont l'application a été détectée auparavant. Par exemple, entre les versions 0.9.16 et 0.9.17, la classe *org.jfree.chart.renderer.AbstractSeriesRenderer* a été introduite comme superclasse de 2 classes existantes (*org.jfree.chart.renderer. AbstractCategoryItemRenderer* et *org.jfree.chart.renderer.WaferMapRenderer*) et elle a été supprimée entre les versions 0.9.18 et 0.9.19. Ce qui correspond à l'application suivie de l'annulation de la tactique UE.

Dans ce contexte, nous avons observé un patron récurrent d'évolution adopté par les développeurs de jFreeChart ; on a observé la création de superclasses et l'externalisation des constantes dans ces dernières, ensuite quelques versions plus tard le retour des constantes vers leurs classes d'origine et la suppression de ces superclasses. Par exemple, entre les versions 0.8.1 et 0.9.0 les classes *CategoryPlotConstants* et *ChartPanelConstants* (les deux

appartenant au paquetage *com.jrefinery.chart*) ont été créées pour externaliser les constantes présentes respectivement dans les classes *CategoryPlot* et *ChartPanel* (appartenant aussi au paquetage *com.jrefinery.chart*). Pour permettre à *CategoryPlot* et *ChartPanel* de continuer à utiliser leurs constantes, des relations d'héritages ont été créées avec les nouvelles classes. Par la suite, entre les versions 0.9.9 et 0.9.10, les constantes de *CategoryPlotConstants* ont été retournées dans *CategoryPlot* et *CategoryPlotConstants* a été supprimée. Le processus similaire a été fait avec *ChartPanelConstants* et *ChartPanel* entre les versions 0.9.20 et 1.0.0. Cette tendance à appliquer des tactiques et à les annuler plus tard soulève des questions à propos de la cohérence de l'évolution du système, en général, et du maintien de la conformité aux décisions architecturales, en particulier. En fait, ce cas est un exemple de situation où il aurait été important de détecter les tactiques architecturales implémentées par le système et les communiquer aux développeurs pour prévenir des modifications erratiques du système.

Pour les 3 dernières annulations mises en correspondance avec des applications, on constate que la tactique annulée qui est détectée ne correspond pas à la tactique appliquée. Par exemple, entre les versions 0.7.0 et 0.7.1 la tactique de SR a été détectée du fait du déplacement de classes du paquetage *com.jrefinery.chart* vers le nouveau paquetage *com.jrefinery.chart.combination*. Cependant, lors du passage de la version 0.8.1 à 0.9.0, les classes contenues dans le paquetage *com.jrefinery.chart.combination* ont été déplacées vers 2 paquetages (*com.jrefinery.chart* et *com.jrefinery.data*) et ce dernier a été supprimé. Cet ensemble d'actions est détecté par TacMatch comme une annulation d'ACS. En fait, l'occurrence de SR qui a été détectée en premier fait partie de l'application d'une ACS, laquelle a été introduite de façon incrémentale à travers plusieurs transitions des versions 0.7.0 jusqu'à la version 0.8.1. Dans ce contexte, des travaux futurs sont nécessaires pour établir les relations entre différentes tactiques de façon à agréger et interpréter correctement l'application successive de certaines tactiques.

3.4.2 Recherche de tendance entre les versions majeures, mineures et les révisions

Pour cette analyse, nous avons comparé les résultats du processus de détection lorsqu'il est appliqué sur les deltas générés à partir de deux versions mineures (respectivement majeures) successives et les résultats du processus de détection lorsqu'il est appliqué sur les révisions intermédiaires qui ont eu lieu entre ces versions mineures (respectivement majeures). Nous supposons que si les développeurs ont fait évoluer le système de façon cohérente d'une révision à l'autre, les résultats agrégés du processus de détection appliqué aux versions intermédiaires seraient les mêmes que ceux détectés entre les versions mineures (respectivement majeures). Le Tableau 3.6 présente la distribution des applications et annulations de tactique détectées entre les versions successives mineures ou majeures. Ce tableau n'affiche pas les passages de versions mineures ou majeures n'ayant pas d'occurrences détectées (par exemple entre les versions 0.6.0 et 0.7.0) et les passages entre les versions mineures ou majeures n'ayant pas de révision (par exemple entre les versions 0.5.6 et 0.6.0).

Tableau 3.6 Distribution des applications et annulations réelles par deltas générés de versions successives mineures ou majeures et par tactiques

Delta	Application de tactique				Annulation de tactique				Total
	SR	UE	AC	IC	SR	UE	ACS	IC	
v0.7.0 v0.8.0	2	5							7
v0.8.0 v0.9.0		4	1	1	1	1	1		9
v0.9.0 v1.0.0	10	5	14	1		4			34

3.4.2.1 Analyse des résultats entre les versions 0.7.0 et 0.8.0

Pour les 7 occurrences détectées entre les versions mineures 0.7.0 à 0.8.0, il y a 2 occurrences détectées en plus qu'entre les révisions intermédiaires entre ces deux versions. La première occurrence est un cas de faux négatif sur les révisions intermédiaires (observée lorsque la fiabilité de l'approche a été analysée), par contre la seconde est un cas d'application progressive d'une tactique. Entre les versions mineures 0.7.0 et 0.8.0,

TacMatch identifie la création de la classe *com.jrefinery.data.SignalsDataset* et l'ajout de relations d'héritage entre elle et ses sous-classes. Ceci correspond à une occurrence d'UE. Cependant, dans les révisions entre les versions mineures 0.7.0 et 0.8.0, *SignalsDataset* est créée entre les révisions 0.7.0 et 0.7.1 alors que les relations d'héritage entre *SignalsDataset* et ses sous-classes ont été ajoutées plus tard entre les révisions 0.7.1 et 0.7.2. Ce qui explique pourquoi UE n'est pas détectée au niveau des révisions, mais juste au niveau des versions mineures.

3.4.2.2 Analyse des résultats entre les versions 0.8.0 et 0.9.0

Entre les versions mineures 0.8.0 et 0.9.0, les occurrences détectées entre les versions mineures correspondent exactement aux occurrences détectées entre ses révisions.

3.4.2.3 Analyse des résultats entre les versions 0.9.0 et 1.0.0

Entre les versions 0.9.0 et 1.0.0 (qui sont des versions majeures), TacMatch détecte 34 occurrences d'applications et d'annulations de tactiques, par contre TacMatch détecte un total de 83 occurrences d'applications et d'annulations sur les révisions successives entre ces deux versions. L'analyse de ces occurrences a permis de constater que 31 occurrences figurent dans les deux groupes d'occurrences, 52 occurrences détectées uniquement sur les révisions et 3 détectées uniquement entre les versions 0.9.0 et 1.0.0.

Parmi les 3 occurrences présentes uniquement entre les versions 0.9.0 et 1.0.0, on distingue 1 cas de faux négatif sur les révisions, 1 cas d'application progressive d'une tactique au travers des révisions et 1 cas d'application progressive de tactique dont une étape de l'application prend la forme d'une autre tactique.

L'application progressive de tactique dont une étape correspond à autre tactique a été constatée entre les révisions 0.9.2 et 0.9.3; la classe *org.jfree.chart.renderer.AbstractRenderer* a été créée et des liens d'héritage ajoutés avec des classes existantes dans le même paquetage, ce qui correspond bien à une UE. Ensuite dans les révisions suivantes les

sous-classes ont été déplacées dans des paquetages différents. À l'analyse des versions 0.9.0 et 1.0.0, TacMatch détecte donc la création de la classe *AbstractRenderer* et l'ajout des liens d'héritage avec des classes existantes dans des paquetages différents, ce qui correspond à une ACS (voir ACS ligne 3 du Tableau 2.2). Il ressort de cet exemple que ce cas d'ACS peut se mettre en place progressivement premièrement par l'application d'une UE suivie par un ensemble de déplacements de classes qui, dans certains cas, peut s'apparenter à des SR.

La répartition des occurrences détectées uniquement sur les révisions est la suivante :

- 3 applications de tactique et leurs annulations : Il s'agit ici de tactiques qui ont été appliquées dans des révisions et annulées dans les révisions suivantes (telle que décrit dans la section 3.4.1). Ce qui fait que du point de vue des versions mineures/majeures les changements ne sont pas constatés.
- 34 occurrences d'applications ou annulations dont les structures ont été modifiées durant les révisions : Telle que décrite dans la section 3.4.1.1, une tactique est une transformation du système. Les occurrences de ce groupe ont été détectées dans les révisions, mais leurs structures ont été progressivement dégradées au point où entre les versions mineures ils ne correspondent plus à la représentation d'une tactique. Par exemple, une encapsulation a été détectée entre les révisions 0.9.4 et 0.9.5 avec la création de la classe com.jrefinery.data.Value et l'ajout de relation d'héritage avec la sous-classe com.jrefinery.data.MeterDataset (existante depuis la version 0.8.1). Il faut aussi noter que la classe Value a été créée dans la même révision que sa sous-classe com.jrefinery.data.KeyedValue. Entre les versions 0.9.0 et 1.0.0, Value est effectivement identifiée comme étant créée et dispose de 2 sous-classes org.jfree.data.general.ValueDataset et org.jfree.data.KeyedValue créées respectivement dans les révisions 0.9.7 et 0.9.5. Ce qui correspond à une extension et non une transformation du système.
- 1 occurrence d'application étant une étape de l'application d'une tactique au niveau des versions mineures et majeures (tel que décrit dans le paragraphe précédent);

- 11 occurrences d'application et d'annulation qui ne figurent pas dans l'analyse des versions mineures et majeures à cause des erreurs de MADMatch et de PtiDej.

3.5 Limites de l'expérimentation et validité des résultats

Nous discutons dans cette section certains facteurs qui peuvent impacter la validité des résultats obtenus avec notre expérimentation.

3.5.1 Validité externe

Cette section décrit les limites à l'application de notre approche dans un contexte autre que celui de l'expérimentation.

L'approche décrite dans ce mémoire se base sur la description des tactiques par un ensemble de modifications élémentaires observées au niveau du code source entre différentes versions d'un système. L'approche a été appliquée sur un ensemble de tactiques de la modificabilité. Elle peut aussi s'appliquer sur d'autres tactiques telles que la tactique de gestion des exceptions reliée à l'attribut de qualité disponibilité ou la tactique de création de processus additionnel reliée de l'attribut de qualité performance.

Cependant, certaines tactiques peuvent nécessiter une analyse dynamique du code source (i.e. il faut exécuter le code pour pouvoir les détecter) telle que la tactique d'augmentation de l'efficacité des ressources reliée à l'attribut de qualité performance. D'autres tactiques peuvent nécessiter d'autres types d'analyses. Par exemple, pour certaines tactiques de chargement différé reliées à l'attribut de qualité modificabilité, il faut pouvoir analyser les fichiers de configuration du système ; pour la tactique de maintien de plusieurs copies des données reliées à l'attribut de qualité performance, il faudrait analyser les sources de données du système. Ainsi, notre approche est limitée aux tactiques qui ont un impact observable sur le code source.

3.5.2 Validité interne

Cette section a pour objectif de décrire les facteurs qui pourraient briser la relation de cause à effet entre les données et les résultats obtenus avec l'approche.

Le premier facteur identifié est la relation entre les tactiques. En effet, l'expérimentation a permis de constater qu'une tactique peut être composée d'autres tactiques plus élémentaires comme cela a été constaté avec la tactique ACS dont une étape a été détectée par l'outil comme une UE ou une SR. Puisque nous n'avons pas encore implémenté un mécanisme qui relie et agrège les tactiques, nous avons tendance à interpréter chaque tactique détectée de façon individuelle et locale.

Le second facteur est la dépendance de TacMatch à des outils externes, notamment pour la génération des deltas entre versions (MADMatch). L'efficacité de notre approche est donc dépendante de l'efficacité de ces outils.

3.6 Conclusion

Dans ce chapitre, nous avons d'abord présenté le contexte de l'expérimentation de l'approche. Deux questions de recherches ont orienté cette expérimentation : (**Q1**) Quelle est l'efficacité de l'approche proposée pour la détection des tactiques ? et (**Q2**) Sommes-nous en mesure de déduire une tendance dans l'évolution de l'architecture en utilisant TacMatch ?

Pour répondre à ces questions, l'approche a été appliquée sur plusieurs versions du projet Java libre jFreeChart et 4 tactiques (**séparation de responsabilités, augmentation de cohésion, encapsulation et abstraction des services communs**) de l'attribut de qualité **modificabilité**.

Pour la première question de recherche qui porte sur la validité des résultats, les valeurs obtenues pour la précision et le rappel de l'approche conduisent à la conclusion selon laquelle l'approche est plus efficace pour la détection des applications de tactiques, mais elle

nécessite encore des améliorations en ce qui concerne la logique de génération de la représentation des annulations de tactiques.

Pour répondre à la seconde question de recherche qui est la possibilité d'identifier des tendances dans l'évolution de l'architecture en utilisant TacMatch, 2 analyses ont été faites et elles ont ressorti 3 tendances observables dans la configuration de l'expérimentation. La première tendance est l'application d'une tactique et son annulation dans une révision ultérieure. La seconde tendance est l'application progressive des tactiques, ce qui les rend non détectables entre les révisions, mais visibles entre les versions mineures ou majeures du système. La dernière tendance est l'application de tactiques entre les révisions, mais dont toutes les entités impliquées dans la tactique sont créées entre les versions mineures ou majeures, ce qui rend la tactique non visible entre les versions mineures ou majeures.

Ce chapitre se termine par une discussion des limites de l'approche. L'approche se limite à la détection de tactiques dont l'effet est observable sur le code source et cela par une analyse statique. Aussi, l'efficacité de notre approche dépend des outils sur lesquels se base l'approche pour faire la rétro-ingénierie et la génération des deltas.

CONCLUSION

Afin de répondre au manque de suivi de la documentation de l'architecture durant l'évolution du système par rapport à la prise en compte des besoins non fonctionnels, cette recherche a développé une approche qui s'appuyant sur les attributs de qualité pour représenter les besoins non fonctionnels permet de définir et détecter des tactiques architecturales en se basant sur les changements observés entre les versions du système. Cette approche a été automatisée par l'outil **TacMatch**.

En ce qui concerne la définition de la tactique architecturale, l'approche s'inspire du formalisme SQL pour définir un langage permettant de représenter la tactique sous forme d'actions élémentaires (Ajout, suppression et déplacement) sur les paquetages, classes, attributs et méthodes. Des contraintes sont aussi définies sur les actions et entités impliquées pour différencier les tactiques ayant des groupes d'actions similaires.

TacMatch est l'outil qui a été réalisé pour la détection des tactiques formalisées. Il lit la représentation et s'appuie sur le principe de chaîne de responsabilités pour générer automatiquement l'algorithme de détection de la tactique. Il génère aussi automatiquement la représentation de l'annulation et la tactique. TacMatch prend en entrée les descriptions de 2 versions obtenues par rétro-ingénierie avec PtiDej et les changements observés entre ces deux versions obtenues avec MADMatch.

L'approche a été expérimentée sur le projet libre jFreeChart pour la détection de certaines tactiques de l'attribut de qualité modificabilité. De cette expérimentation, des occurrences d'application et d'annulation de la modificabilité ont été détectées. L'analyse de ces occurrences a révélé des comportements des développeurs contraires à des schémas d'évolution de l'architecture détectés. Ces comportements auraient pu être évités si les développeurs avaient pris connaissance de ces schémas.

L’expérimentation a aussi permis de déceler des limites et pistes d’amélioration pour cette approche de détection des tactiques architecturales dans l’évolution d’un système.

Les valeurs de la précision et du rappel de la détection des annulations lors de l’expérimentation suggèrent la nécessité d’améliorer la génération des annulations des tactiques. De plus, de nouvelles expérimentations pour déterminer les meilleurs paramètres de MADMatch et PtiDej permettraient d’améliorer la précision pour la détection des applications et annulations des tactiques.

Du fait que l’approche s’appuie sur des tactiques représentables à partir de la structure du code source, des travaux doivent aussi être effectués pour l’étendre aux tactiques faisant intervenir des éléments autres que le code source (fichier de configuration, base de données, etc.).

La comparaison des occurrences des versions mineures/majeures à celles des révisions a permis de constater qu’une tactique peut s’appliquer progressivement sur plusieurs révisions et certaines étapes de cette décomposition peuvent correspondre à d’autres tactiques élémentaires. Pour faciliter l’agrégation de ces changements sur des révisions successives, il faudrait pouvoir définir les relations de composition entre tactiques et le mécanisme de détection de ces agrégations.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Abi-Antoun, M., J. Aldrich, N. Nahas, B. Schmerl et D. Garlan. 2006. « Differencing and Merging of Architectural Views ». In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. (18-22 Sept. 2006), p. 47-58.

Aldrich, Jonathan, Vibha Sazawal, Craig Chambers et David Notkin. 2003. « Language support for connector abstractions ». In *17th European Conference on Object Oriented Programming, ECOOP 2003*. Vol. 2743, p. 29. Springer Verlag. In *Compendex*.

Arcelli Fontana, Francesca, et Marco Zanoni. 2011. « A tool for design pattern detection and software architecture reconstruction ». *Information Sciences,* vol. 181, n° 7, p. 1306-1324.

Arcelli, Francesca, et Luca Cristina. 2007. « Enhancing software evolution through design pattern detection ». In *3rd International IEEE Workshop on Software Evolvability 2007, SE 2007, October 1, 2008 - October 1, 2008*. (Paris, France), p. 7-14. Coll. « 3rd International IEEE Workshop on Software Evolvability 2007, SE »: Inst. of Elec. and Elec. Eng. Computer Society. < http://dx.doi.org/10.1109/SE.2007.11 >.

Bachmann, Felix, Len Bass et Robert Nord. 2007. *Modifiability tactics*. CMU Software Engineering Institute Technical Report CMU/SEI-2007-TR-002.

Bass, Len, Paul Clements et Rick Kazman. 2012. *Software architecture in practice*. Pearson Education India.

Chidamber, S. R., et C. F. Kemerer. 1994. « A metrics suite for object oriented design ». *IEEE Transactions on Software Engineering,* vol. 20, n° 6, p. 476-493.

Clements, P., D. Garlan, R. Little, R. Nord et J. Stafford. 2003. « Documenting software architectures: views and beyond ». In *IEEE 25th International Conference on Software Engineering, 3-10 May 2003*. (Los Alamitos, CA, USA), p. 740-1. Coll. « Proceedings 25th International Conference on Software Engineering »: IEEE Comput. Soc.

Fowler, Martin. 1999. *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Gamma, Erich, Richard Helm, Ralph Johnson et John Vlissides. 1995. « Design Patterns: Abstraction and Reuse of Object-Oriented Design ». In. Gamma1993.

Garlan, D., et M. Shaw. 1993. « An introduction to software architecture ». In *Advances in software engineering and knowledge engineering*. p. 1-39. Singapore: World Scientific.

Garlan, David, Jeffrey M Barnes, Bradley Schmerl et Orieta Celiku. 2009. « Evolution styles: Foundations and tool support for software architecture evolution ». In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. p. 131-140. IEEE.

Gilbert, D, et T Morgner. 2007. « JFreeChart, a free Java class library for generating charts ». *Publisher Full Text*.

Guéhéneuc, Yann-Gaël, et Giuliano Antoniol. 2008. « DeMIMA: A Multilayered Approach for Design Pattern Identification ». *IEEE Transactions on Software Engineering,* vol. 34, n° 5, p. 667-684.

Kapto, Christel, Ghizlane El Boussaidi, Sègla Kpodjedo et Chouki Tibermacine. 2016. « Inferring Architectural Evolution from Source Code Analysis ». In *European Conference on Software Architecture, ECSA 2016*. Coll. « ACM International Conference Proceeding Series »: Spring-Verlag Berlin Heidelberg.

Kim, Miryung, Matthew Gee, Alex Loh et Napol Rachatasumrit. 2010. « Ref-Finder: A refactoring reconstruction tool based on logic query templates ». In *18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE-18, November 7, 2010 - November 11, 2010*. (Santa Fe, NM, United states), p. 371-372. Coll. « Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering »: Association for Computing Machinery. < http://dx.doi.org/10.1145/1882291.1882353 >.

Kpodjedo, Segla, Filippo Ricca, Philippe Galinier, Giuliano Antoniol et Y-G Gueheneuc. 2013. « Madmatch: Many-to-many approximate diagram matching for design comparison ». *Software Engineering, IEEE Transactions on,* vol. 39, n° 8, p. 1090-1111.

Le, Duc Minh, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian et Nenad Medvidovic. 2015. « An empirical study of architectural change in open-source software systems ». In *12th Working Conference on Mining Software Repositories, MSR 2015, May 16, 2015 - May 17, 2015*. (Florence, Italy) Vol. 2015-August, p. 235-245. Coll. « IEEE International Working Conference on Mining Software Repositories »: IEEE Computer Society. < http://dx.doi.org/10.1109/MSR.2015.29 >.

McCabe, Thomas J. 1976. « A complexity measure ». In *Proceedings of the 2nd international conference on Software engineering*. (San Francisco, California, USA), p. 407. 807712: IEEE Computer Society Press.

McNair, Andrew, Daniel M German et Jens Weber-Jahnke. 2007. « Visualizing software architecture evolution using change-sets ». In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*. p. 130-139. IEEE.

Stevens, Wayne P., Glenford J. Myers et Larry L. Constantine. 1974. « Structured design ». *IBM Systems Journal,* vol. 13, n^{o} 2, p. 115-139.

Tonu, Subrina Anjum, Azin Ashkan et Ladan Tahvildari. 2006. « Evaluating architectural stability using a metric-based approach ». In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. p. 10 pp.-270. IEEE.

Zhihua, Wen, et V. Tzerpos. 2004. « An effectiveness measure for software clustering algorithms ». In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*. (24-26 June 2004), p. 194-203.