Support à la rénovation d'une architecture logicielle patrimoniale : Un cas réel chez Thales Air Systems

Brice Govin

Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique
UFR

École doctorale SPI Lille

Support à la rénovation d'une architecture logicielle patrimoniale : Un cas réel chez Thales Air Systems

THÈSE

présentée et soutenue publiquement le 26 juin 2018

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Brice Govin

Composition du jury

Rapporteurs :	Salah Sadou (Professeur – Université de Bretagne Sud) Christelle Urtado (Maître de conférences (HDR) – Institut Mines-Télécom Mines d'Alès)
Examinateurs :	Luc Fabresse (Professeur – Institut Mines-Télécom Lille Douai) Alain Plantec (Professeur – Université de Bretagne Occidentale)
Directeur de thèse :	Nicolas Anquetil (Maître de conférences (HDR) – Université Lille 1)
Co-Encadrante de thèse :	Anne Etien (Maître de conférences (HDR) – Université Lille 1)
Invité :	Arnaud Monégier du Sorbier (Thales)

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR USTL/CNRS 9189
INRIA Lille - Nord Europe

Numéro d'ordre: XXXXX

Remerciements

Table des matières

Table des figures

Liste des tableaux

CHAPITRE 1

Introduction

Contents

La maintenance et l'évolution prennent une part importante de la vie des logiciels patrimoniaux [Nosek 1990, Krogstie 2015, Malhotra 2016]. La maintenance est définie par IEEE dans ces standards [IEEE 1998], comme :

> *"La modification d'un produit logiciel après sa livraison, afin de corriger des erreurs, améliorer les performances ou d'autres caractéristiques, ou d'adapter le produit à un environnement modifié."*

Au sens des standards IEEE, la maintenance contient également les évolutions apportées pour l'adaptation du logiciel dans un nouvel environnement (*e.g.,* client lourd vers client léger). Maintenir des logiciels qui gagnent en âge est une tâche qui s'intensifie d'année en année, d'autant plus lorsqu'ils ont une valeur métier forte pour les entreprises. Les industriels cherchent donc constamment des techniques pour pouvoir les modifier, les faire évoluer ou les remettre au goût du jour, dans le but de rester compétitif.

Thales Air Systems fait partie de ces entreprises souhaitant améliorer leur compétitivité tout en mettant en valeur leur patrimoine logiciel. Ainsi, dans le cadre de la rénovation physique d'un de leurs systèmes, Thales Air Systems a mis en place un projet de rénovation du logiciel inclus dans ce système. Le but de ce projet est de pouvoir assurer des modifications futures plus simples et moins coûteuses, autant pour le client que pour l'entreprise. L'entreprise visant des projets similaires sur d'autres systèmes, elle souhaiterait pouvoir automatiser et reproduire la démarche réalisée pour ce premier projet. L'enjeu serait donc de fournir une démarche de rénovation d'architecture automatisée et réplicable sur d'autres logiciels de l'entreprise.

Dans ce cadre, Thales Air Systems a souhaité financer une thèse CIFRE afin de faciliter, améliorer et accélérer la démarche de rénovation d'architecture logicielle par l'entreprise sur son premier projet. Cela permettra finalement de diminuer les coûts directement liés à ces projets, mais également les coûts de maintenance pour l'entreprise.

1.1 Contexte

Dans cette thèse, nous avons étudié et travaillé sur les logiciels développés par Thales Air Systems. Ceux-ci possèdent trois caractéristiques principales :

Critique : *i.e.,* être fiables à tout moment et conserver cette fiabilité malgré les évolutions [Hinchey 2010].

Temps réel : *i.e.,* respecter un temps de réponse inférieur à une certaine contrainte de temps (*e.g.,* 15 ms) [Ben-Ari 2006].

Embarqué : *i.e.,* faire partie d'un système mécanique, électronique et informatique complet [Barr 2006].

Ces caractéristiques impliquent donc un coût de maintenance plus élevé que la moyenne (60% en moyenne d'après [Glass 2001]) et poussent les logiciels à devenir *patrimoniaux* (*i.e.,* anciens et à forte valeur métier pour les entreprises qui les possèdent [Bisbal 1999]).

Cependant, une longue durée de vie (et donc une longue maintenance) entraîne souvent une perte de connaissance à propos du logiciel [Ducasse 2009]. Cette perte de connaissance induit une augmentation non négligeable des coûts de maintenance et d'évolution. Afin de réduire ces coûts, deux choix sont disponibles, soit redévelopper le logiciel de zéro soit rénover son architecture [Perry 1992, Oquendo 2016]. La première a été jugée trop coûteuse par l'entreprise puisque la connaissance domaine n'est plus suffisante pour redévelopper exactement la même solution. Dans le cadre d'un plan de maintenance global, Thales Air Systems a choisi de rénover l'architecture de ses logiciels en appliquant la deuxième solution. En effet, ses systèmes ont tous été développés sur des périodes d'une dizaine d'années chacun et ont tous nécessité un nombre important d'ingénieurs. Afin de rester compétitif, Thales Air Systems ne peut pas se permettre de redévelopper chacun de ses logiciels de zéro et souhaite donc réutiliser leur code source dans des architectures rénovées. De cette manière, l'entreprise souhaite mettre en valeur son patrimoine logiciel tout en l'améliorant.

Cette thèse se situe dans un contexte de rénovation de l'architecture d'un système logiciel patrimonial, critique, temps réel et embarqué. Elle s'inscrit dans un projet industriel plus important qui vise la rénovation d'architecture d'un parc logiciel afin de réduire les coûts d'évolution et de maintenance. À travers cette thèse,

nous cherchons à identifier une démarche de rénovation d'architecture supportée par un outillage qui permette l'automatisation et la réplication de cette démarche aux autres projets rénovations de l'entreprise.

1.2 Problématique

La maintenance de logiciels patrimoniaux est une activité très importante pour les entreprises et il arrive que de simples évolutions ou des corrections de bugs deviennent extrêmement coûteuses. Afin de diminuer ces coûts et de pouvoir continuer à maintenir de tels logiciels, Thales Air Systems a mis en place un plan de maintenance qui vise l'évolution d'un ensemble de systèmes en rénovant leur architecture.

Face aux deux choix exposés ci-dessus (redéveloppement de zéro ou réutilisation du code actuel au travers d'une rénovation de l'architecture), Thales Air Systems a choisi la seconde solution qui lui permet de mettre en valeur le code existant du logiciel tout en l'améliorant et en minimisant les risques inhérents au développement. De plus, l'entreprise souhaite profiter de l'expérience gagnée lors d'un premier projet pour la réappliquer sur les suivants.

Notre objectif principal est donc de définir une démarche de rénovation d'architecture qui soit *(i)* automatisable, autant que faire se peut, et qui soit *(ii)* réplicable sur d'autres projets de l'entreprise. Celle-ci souhaite aussi profiter le plus possible du code source existant pour garantir la qualité du résultat final.

La rénovation d'architecture est un sujet traité depuis de nombreuses années dans la littérature [Chikofsky 1990]. Mais beaucoup des solutions proposées ne s'appliquent pas à notre cas. Par exemple, du point de vue de la démarche, très peu ont été complètement formalisées et quand c'est le cas (par exemple, le *Reflexion Model* [Murphy 1995]), le niveau d'automatisation peut ne pas être suffisant pour satisfaire les besoin de l'entreprise. Ou bien certaines solutions ne traitent que l'évolution de l'architecture du logiciel et n'ont pas pour objectif de traiter le code source [Fahmy 2000, Zhang 2010a]. D'autres solutions, encore, permettent d'associer du code source à une architecture afin de le réutiliser ou le transformer [Murphy 1995, Kazman 1998, Allier 2011]. Toutefois, la littérature a montré que ces solutions ne supportent pas un passage à l'échelle sur des logiciels patrimoniaux, i.e., en termes de nombre lignes de code [Bhatti 2012]. Parmi les solutions restantes, beaucoup possèdent des contraintes particulières qui sont difficiles à respecter dans le cas de Thales Air Systems. Par exemple, ces solutions peuvent nécessiter d'exécuter le logiciel analysé [Allier 2009] ; des contraintes sur le type d'architecture existante ou visée [Khadka 2011] ; des exigences sur les artefacts disponibles pour la rénovation [Yan 2004].

Ainsi un des problèmes que nous devons affronter est de trouver une démarche

de rénovation d'architecture permettant la réutilisation de code source et qui respectent les différentes contraintes liées au projet de Thales Air Systems, comme par exemple de s'adapter aux habitude de travail des développeurs de l'entreprise.

Nous devrons aussi analyser les solutions proposées par le passé dans la littérature, dans le cadre de notre processus. Nous devrons donc identifier les contraintes générales d'utilisations de notre démarche, les parties automatisables, et finalement définir les moyens de cette automatisation.

1.3 Notre solution en bref

Afin d'assurer que notre solution puisse être appliquée sur les projet de Thales Air Systems (passage à l'échelle et contraintes particulières), nous avons étudié le premier projet mis en place par l'entreprise. Ce projet a été réalisé de manière libre par les développeurs dans le respect donc de leur habitudes. En nous basant sur l'analyse de ce projet et de ses contraintes, et à partir des propositions existantes dans la littérature, nous avons proposé une démarche structurée de rénovation d'architecture qui permette la réutilisation du code source.

Pour répondre au souci d'automatisation, nous avons initialement choisi parmi les techniques connues celles qui semblaient les plus prometteuses et les plus à même de s'adapter à notre démarche. Nous avons ensuite réalisé diverses expériences d'automatisation de cette démarche en appliquant ces techniques. Les résultats ne se sont pas montrés suffisamment satisfaisants.

Nous nous sommes alors tournés vers des solutions d'automatisations plus spécifiques à notre démarche. Nous avons proposé un outillage constitué de ce que nous appelons des opérateurs qui se basent sur les activités réalisées ou planifiées par les ingénieurs lors des projets de l'entreprise. Ces opérateurs permettent donc aux ingénieurs de sauvegarder, partager et automatiser leur démarche afin qu'elle puisse être réappliquée. Nos opérateurs tiennent compte des points suivants :

— les contraintes provenant des projets de Thales Air Systems ;

— de techniques existantes ;

— les besoins exprimés par les ingénieurs ;

— la réutilisation du code source du logiciel à rénover.

1.4 Notre contribution

À travers cette thèse, nous avons répondu aux attentes de l'entreprise : nous avons proposé une démarche structurée de rénovation d'architecture logicielle basée sur l'étude d'un premier projet. Nous avons ensuite expérimenté les solutions de rénovation d'architecture existantes et montré que sur notre cas réel, elles ne

donnaient pas de résultats satisfaisant les attentes de cet industriel. Nous avons donc proposé un outillage constitué d'un ensemble d'opérateurs qui facilite la réplication et l'automatisation de notre démarche structurée. Ces opérateurs ont été mis en pratique pour tester leur applicabilité et la validité de leur résultats.
Les principales contributions de cette thèse sont donc :

1. la structuration de la démarche de rénovation logicielle appliquée dans le cadre d'un projet industriel ;
2. l'évaluation de techniques existantes pour la rénovation logicielle sur un projet industriel ;
3. la description d'un ensemble d'opérateurs facilitant la réplication et l'automatisation de la démarche structurée et qui tiennent compte des contraintes liées à un cas réel.

1.5 Structure de la thèse

Cette thèse est structurée de la manière suivante :

— le chapitre 2 décrit le projet de maintenance logicielle de Thales Air Systems suivant l'ontologie des projets de maintenance de Kitchenham *et al.* ;

— le chapitre 3 fournit l'état de l'art des techniques de rénovation logicielle et identifie celles pouvant être appliquées au projet de Thales Air Systems ;

— le chapitre 4 décrit la démarche de maintenance réalisée par les ingénieurs de l'entreprise et propose une structuration de cette démarche ;

— le chapitre 5 décrit les expériences que nous avons réalisées afin d'évaluer, sur le projet de l'entreprise, les techniques identifiés lors de l'état de l'art ;

— le chapitre 6 présente notre solution : des opérateurs facilitant la réplication et l'automatisation de notre démarche structurée.

— le chapitre 7 conclut cette thèse et fournit les perspectives pour de futurs travaux.

1.6 Liste des publications

Voici la liste des articles publiés ou soumis à révision dans le cadre de cette thèse par ordre chronologique :

1. Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Reverse Engineering Tool Requirements for Real Time Embedded Systems*. In SATToSE'15, Mons, Belgium, July 2015. short paper

2. Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Measuring the progress of an Industrial Reverse Engineering Process*. In BENEVOL'15, Lille, France, December 2015
3. Brice Govin, Nicolas Anquetil, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Clustering Techniques for Conceptual Cluster*. In IWST'16, Prague, Czech Republic, August 2016
4. Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *How Can We Help Software Rearchitecting Efforts ? Study of an Industrial Case*. In Proceedings of the International Conference on Software Maintenance and Evolution, (Industrial Track), Raleigh, USA, October 2016
5. Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Managing an Industrial Software Rearchitecting Project With Source Code Labelling*. In Complex Systems Design & Management, pages 61–74, Paris, France, December 2017. Springer
6. Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Large Legacy Software Re-Architecting Project: an Experience Report*. Journal of Software: Evolution and Process, (MAJOR REVISION) 2018

CHAPITRE 2

Description du projet

Contents

Cette thèse a accompagné Thales Air Systems dans le premier projet d'un plan global de maintenance d'un ensemble de logiciels patrimoniaux. Ce premier projet vise l'évolution du logiciel impliqué en modifiant son architecture. L'entreprise souhaite automatiser tout ou partie de la démarche réalisée durant ce projet afin de pouvoir la reproduire sur d'autres projets impliquant les logiciels du plan de maintenance. Il est alors nécessaire de décrire le premier projet de ce plan pour pouvoir identifier les caractéristiques pouvant contraindre la démarche appliquée.

Kitchenham *et al.* [Kitchenham 2002] proposent une ontologie de description des projets de maintenance suivant quatre dimensions : le produit, les activités de maintenance, la démarche de maintenance logicielle et les ressources humaines du projet. La Section 2.1 décrit les quatre dimensions telles qu'expliquées par Kitchenham *et al.*, en mettant en avant les différentes caractéristiques de description. La Section 2.2 suit ces quatre dimensions pour décrire le projet de Thales Air Systems et identifie, à partir de ces caractéristiques, des exigences que doivent respecter de potentielles techniques pour être applicables sur ce projet. Finalement, la Section 2.3 conclut ce chapitre.

2.1 Caractérisation des projets de maintenance

Kitchenham *et al.* proposent une ontologie de description des projets de maintenance [Kitchenham 2002] découpée en quatre dimensions : le produit, les activités de maintenance, la démarche de maintenance logicielle et les ressources humaines du projet. Ces quatre dimensions sont expliquées dans les sections suivantes.

2.1.1 Produit

La première dimension, le produit, décrit le logiciel maintenu à travers plusieurs caractéristiques : la taille du produit ; son âge ; sa maturité ; sa composition ; et la qualité dee ses artefacts.

Taille du produit correspond à la taille du logiciel en terme de nombre de ligne de code, de nombre d'artefact de langage (*e.g.,* nombre de classes en Java) ou de la complexité des artefacts du langage (*e.g.,* complexité cyclomatique), par exemple.

Domaine d'application du produit correspond au domaine pour lequel le produit est fait (*e.g.,* finance, télécommunications, défense *etc.*).

Âge du produit correspond à l'âge du logiciel depuis sa première livraison et à l'âge des technologies employées pour le développer.

Maturité du produit est différente de son âge car elle vise à caractériser la phase du cycle de vie dans laquelle se trouve le logiciel. Selon Kitchenham *et al.*, un logiciel peut se trouver dans quatre états de maturité différents :

— Enfance : Juste après la première livraison, les utilisateurs rapportent les premiers bugs.

— Adolescence : Le logiciel gagne en utilisateurs ce qui entraîne des rapports de bugs plus fréquents. Ces bugs mènent parfois à des modifications du logiciel.

— Âge Adulte : Les utilisateurs sont aussi nombreux qu'auparavant, mais les rapports de bugs sont plus rares. Les utilisateurs commencent à demander de nouvelles fonctionnalités. De plus, les fréquentes modifications peuvent entraîner un besoin de restructuration du logiciel (pour faciliter les évolutions).

— Sénilité : Dans cette phase, les utilisateurs se font rare et le logiciel n'est maintenu que de manière corrective (*i.e.,* correction de bugs).

Composition du produit consiste à décrire les éléments qui composent le logiciel. Un logiciel peut être composé, par exemple, de COTS (*Component Off The Shelf*) [1], de librairies tierces et d'éléments architecturaux développés *from scratch*.

Qualité des artefacts du produit correspond à la description et à l'exactitude des éléments d'information disponibles, à propos du logiciel. Typiquement, les artefacts du produit sont le code source, des modèles (*e.g.,* diagramme de classes), ou de la documentation, dont l'exactitude peut varier (*e.g.,* modèle non à jour, version de documentation différente de la version du code source).

1. Composant boite noire développé en général par une compagnie tierce

2.1.2 Activités de maintenance

La deuxième dimension servant à la description des projets concerne les activités de maintenance mises en place sur le projet. Les activités de maintenance sont séparées en quatre grands types : les activités d'investigation, de gestion, de modification et d'assurance qualité.

Activités d'investigation consistent à rechercher dans le code source la cause d'un événement d'entrée (*e.g.,* un bug).

Activités de gestion se concentrent sur la gestion du processus de maintenance (*e.g.,* la gestion de tickets de bug).

Activités de modification sont des activités pouvant être soit de la correction (*e.g.,* correction de bug) soit de l'enrichissement (*e.g.,* changer d'architecture).

Activités d'assurance qualité visent à vérifier que les modifications réalisées n'endommagent pas l'intégrité du logiciel (*e.g.,* qu'elles n'impactent pas le comportement du logiciel).

2.1.3 Démarche de maintenance logicielle

La troisième dimension de l'ontologie de description des projets de maintenance concerne la démarche de maintenance logicielle. Elle est séparée en deux parties : les procédures de maintenance et l'organisation de la démarche de maintenance.

Procédures de maintenance Les procédures de maintenance correspondent à la façon dont les ingénieurs réalisent leurs activités de maintenance. Typiquement, les procédures de maintenance sont faites de :

— Méthode : Procédure systématique qui définit des étapes et/ou des heuristiques (*e.g.,* une technique existante de maintenance).

— Technique : Procédure similaire à une méthode servant à accomplir une activité mais où les étapes sont moins rigoureusement définies (*e.g.,* une technique empirique).

— Script : Un ensemble de lignes directrices servant à construire ou modifier un certain type d'artefact (*e.g.,* des conventions de nommage).

Organisation du processus de maintenance Cette caractéristique permet de détailler le processus suivi afin de réaliser les activités de maintenance. Par exemple, un processus de correction de bugs pourrait être le suivant : *(i)* réception d'un ticket de bug ; *(ii)* investigation du bug ; *(iii)* envoi de la cause du bug ; *(iv)* correction du bug ; *(v)* campagne de test pour acceptation de la correction ; *(vi)* envoi de la résolution aux utilisateurs.

2.1.4 Ressources humaines

La quatrième dimension de description des projets de maintenance concerne les ressources humaines engagées sur le projet de maintenance. Cette dimension se décrit à travers 4 caractéristiques des ressources humains du projet : leurs compétences, leurs niveaux d'expérience, leur attitude face au projet et leurs responsabilités.

Compétences des ingénieurs visent à identifier comment sont perçus les ingénieurs dans le projet (*e.g.,* expert en architecture, débutant en intégration logicielle)

Niveau d'expérience des ingénieurs correspond aux années d'expérience des ingénieurs autant dans le développement logiciel que dans le domaine d'application du logiciel (*e.g.,* 5 ans d'expérience en développement et 3 ans d'expérience dans le domaine des applications financières).

Attitude face au projet correspond à leur motivation face au projet (*e.g.,* s'ils sont réfractaires à la maintenance de code ou non).

Responsabilité des ingénieurs concerne la séparation des activités des ingénieurs (*e.g.,* si les activités de maintenance et les activités de modification sont réalisées par la même équipe ou non).

2.2 Caractérisation du projet de maintenance de Thales Air Systems

Cette section applique l'ontologie de description des projets de maintenance de Kitchenham *et al.* au projet de maintenance de l'entreprise Thales Air Systems. Chacune des sections suivantes décrit le projet de l'entreprise suivant chacune des dimensions expliquées dans la Section 2.1. Ces sections permettent de déterminer les caractéristiques du projet de l'entreprise qui peuvent impacter la démarche de transformation de l'architecture.

2.2.1 Produit

Cette section décrit le produit impliqué dans le projet de maintenance de Thales Air Systems.

Taille du produit Le logiciel patrimonial que l'entreprise souhaite retravailler est un logiciel contenant plus de 300 kLoC[2]. Le logiciel est développé en Ada 83, langage de programmation constitué principalement de packages et de sous-programmes. Les 300 kLoC sont réparties dans 13 474 sous-programmes, eux-mêmes contenus dans 1 494 packages. Nous n'avons pas pu avoir accès à une mesure de la complexité de ce logiciel durant nos expériences. Les logiciels produit par Thales Air Systems étant de taille équivalente voire supérieure à celui-ci, la démarche de modification d'architecture doit supporter une telle l'échelle.

Domaine d'application du produit Le produit impliqué dans le projet de l'entreprise est un logiciel critique, temps-réel, embarqué dont le domaine d'application est la défense aéronautique. Le caractère critique du logiciel implique qu'il ne peut pas être stoppé pendant une longue période et nécessite un calendrier qui restreint le temps disponible pour appliquer la démarche de l'entreprise.

Âge du produit La première version du logiciel a été livrée au début des années 1990. Le logiciel est donc âgé d'une vingtaine d'années. D'autre part, les technologies utilisées pour le développement du logiciel ont été choisies au début du processus de développement et sont âgées de plus de 30 ans (*e.g.,* Ada 83). Ces technologies doivent être prises en compte par la démarche de modification d'architecture de l'entreprise.

Maturité du produit Le produit de Thales Air Systems se situe dans sa phase adulte. Il est toujours en fonctionnement chez les utilisateurs, il y a peu de rapports de bugs et les utilisateurs demandent de nouvelles fonctionnalités. Pour éviter de laisser le logiciel tomber dans sa phase de sénilité, l'entreprise souhaite le restructurer. Toutefois, l'entreprise ne souhaite pas limiter sa démarche aux logiciels "matures".

Composition du produit Thales Air Systems a conçu le logiciel suivant une architecture de référence, propriétaire, basée sur un ensemble de module et un intergiciel de communication propriétaire. Cet intergiciel de communication fonctionne par envois et réceptions d'événements, appelés "messages". Le logiciel est donc composé de :

2. kilo *Lines Of Code*

— Module : un élément architectural qui réalise plusieurs fonctionnalités ;

— Message : une structure d'échange contenant le module émetteur, le module récepteur et les données à transmettre.

Les modules comme les messages sont implémentés par des patrons de conception spécifiques à l'entreprise (*i.e.,* il ne s'agit pas de patrons usuels du type *observer etc.*). Par exemple, les modules correspondent aux packages Ada contenant un élément de code particulier (une tâche Ada). Les patrons usuels sont inutilisables, ainsi la démarche doit pouvoir prendre en compte des patrons propriétaires.

Qualité des artefacts du produit Le code source disponible correspond à la version du logiciel qui est actuellement en fonctionnement. La documentation disponible correspond à une version plus ancienne du code source qui ne comprend pas toutes les évolutions et corrections faites dans le logiciel. Les bancs d'essai logiciel (machines simulant l'environnement du système) n'étant pas disponibles pour cause d'avarie matériel, le logiciel ne peut être exécuté. En effet, étant un logiciel embarqué, celui-ci ne peut fonctionner en dehors de son environnement (ou d'une simulation de cet environnement). Ainsi, la démarche de l'entreprise ne peut pas reposer sur une analyse dynamique du logiciel.

2.2.2 Activités de maintenance

Cette section décrit les activités de maintenance que l'entreprise a réalisées durant son projet.

Activité d'investigation Thales Air Systems a d'abord mené des activités d'investigation qui ont consisté à :

— parcourir la documentation

— interviewer des experts du domaine

— parcourir le code source suivant les informations récoltées

— redocumenter l'architecture existante

— estimer le coût des évolutions futures du logiciel (*i.e.,* prix et complexité de l'évolution)

Ces activités sont nécessaires aux ingénieurs pour acquérir une connaissance du logiciel leur permettant de le traiter correctement. L'entreprise souhaite pouvoir utiliser les informations retrouvées pde cette manière pour faciliter la démarche de modification d'architecture. Il est donc important que cette démarche puisse être alimentée par les connaissances acquises lors des activités d'investigation.

Activité de gestion La gestion du processus de maintenance du logiciel de Thales Air Systems a été déterminée en collaboration avec les utilisateurs. Puisqu'aucune modification du comportement du logiciel n'a été définie, le logiciel sera remplacé entièrement lors de l'arrêt technique du système physique dans lequel il fonctionne (un bateau). De plus, des jalons de présentations et de livrables ont été définis par Thales Air Systems et les utilisateurs en amont du projet. Cette activité de gestion impacte fortement le temps alloué à la démarche de l'entreprise.

Activité de modification Les activités de modification menées par l'entreprise sur le système sont les suivantes :

— modification d'une partie de la plateforme :
La modification d'une partie de la plateforme consiste à changer un ancien calculateur par un nouveau calculateur plus récent et plus puissant. Cette modification nécessite de devoir extraire et modifier le code source développé pour l'optimisation du logiciel sur sa plateforme actuelle. La démarche de l'entreprise doit donc tenir compte du code pouvant être réutilisé ou non.

— portage du langage de programmation :
Le portage consiste à modifier le langage de programmation d'Ada 83 en Ada 2012. Ce portage nécessite d'adapter le code source actuel aux spécificités de l'Ada 2012 (*e.g.,* nouvel signification du mot-clé "interface"). L'entreprise ne souhaite pas se servir des nouvelles fonctionnalités de l'Ada 2012 mais uniquement pouvoir compiler son logiciel avec un compilateur Ada 2012. Ainsi, la démarche de l'entreprise ne nécessite pas de tenir compte du portage du langage de programmation du logiciel.

— modification de l'intergiciel de communication :
Cette activité consiste à changer l'intergiciel actuel par un nouvel intergiciel standardisé, développé par une entreprise tierce. Cette modification nécessite de modifier les appels à l'ancien intergiciel mais également de modifier certains accès aux variables dans le code source pour respecter la politique du nouvel intergiciel. Comme l'intergiciel de communication est considéré dans l'architecture du logiciel, alors la démarche souhaitée par Thales Air Systems doit tenir compte de sa modification.

— modification de l'architecture :
Cette activité consiste à changer l'architecture actuelle (sous forme de modules) par une architecture à composants. Ces deux architectures (ancienne et nouvelle) sont détaillées dans le chapitre 4. La nouvelle architecture du logiciel contient deux niveaux de composants : 15 composants dits "composites", contenant chacun entre 1 et 6 composants dits "atomiques". La démarche que l'entreprise souhaite automatiser visent à modifier l'architecture d'un logiciel, cette activité en est donc le but.

Finalement, chaque activité de modification doit être considérée par la démarche menée par l'entreprise, excepté le portage du langage de programmation. En effet, ce portage consiste uniquement à utiliser un compilateur plus récent et non à utiliser les fonctionnalités introduites par les nouvelles versions du langage.

Activités d'assurance qualité Les activités d'assurance qualité mises en place pour ce projet sont doubles :

— bénéficier du code source existant :
Le fonctionnement du logiciel pendant plus de 20 ans sans modification prouve la qualité de son code source. Ainsi l'entreprise souhaite réutiliser le code source existant afin de bénéficier de sa qualité. La démarche de modification d'architecture menée par l'entreprise doit tenir compte de la réutilisation du code source existant, par exemple en l'allouant aux éléments de la nouvelle architecture (les composants).

— assurer la conservation des résultats et des performances du logiciel :
Cette activité consiste à vérifier que les résultats et les performances du logiciel avec sa nouvelle architecture sont au moins les mêmes que celles du logiciel avec son ancienne architecture. Afin d'assurer la conservation des performances (autant temporelles que spatiales), l'entreprise a mis en place une campagne de tests fonctionnels. Pour chacun des tests fonctionnels, le logiciel doit donc fournir le même résultat, qu'il soit avec sa nouvelle ou son ancienne architecture. De plus, des simulations ont été mises en place pour vérifier les performances temporelles (*e.g.,* mêmes temps de réponses) et spatiales (*e.g.,* pas de fuite mémoire) de la nouvelle architecture du logiciel. Cette activité est réalisée *a posteriori* de la démarche de modification d'architecture et n'a pas pour but de changer l'architecture du logiciel.

2.2.3 Démarche de maintenance logicielle

Cette section décrit la démarche de maintenance appliquée par l'entreprise sur son projet, selon l'ontologie de Kitchenham *et al.*.

Procédure de maintenance Les activités de maintenance réalisées durant le projet de maintenance de Thales Air Systems se basent sur :

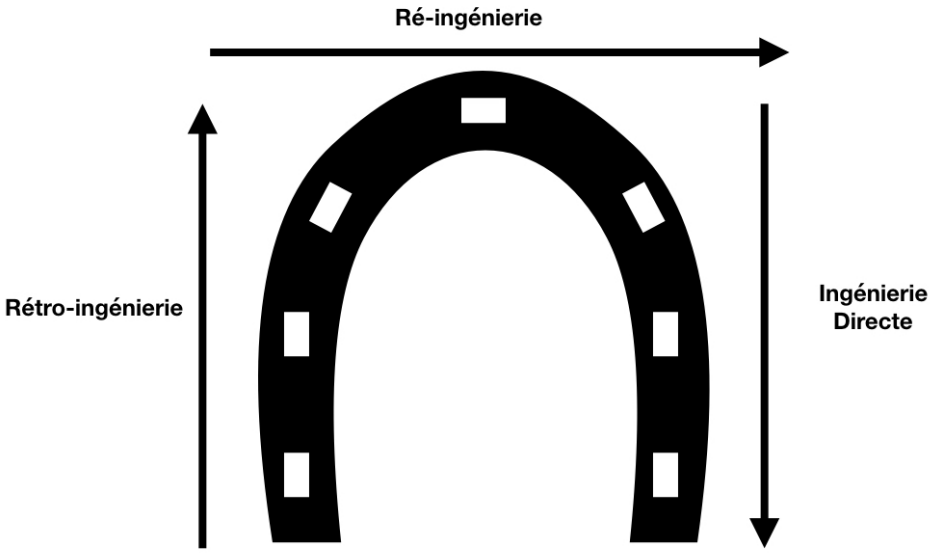
— un ensemble de scripts *bash* développés pour ce projet (*e.g.,* script d'extraction de fichiers suivant une convention de nommage) ;

— un ensemble d'actions manuelles (*e.g.,* copier coller manuel de code source) qui suivent une méthode sans description formelle. Cette méthode est connue uniquement des ingénieurs et n'est documentée nulle part.

L'entreprise souhaite pouvoir se resservir de ces activités dans sa démarche de modification d'architecture afin de garder l'investissement réalisé. Ainsi la démarche de l'entreprise doit tenir compte de ces deux activités.

Organisation du processus de maintenance La processus de maintenance proposée par l'entreprise suit l'*Horseshoe* [Kazman 1998]. L'*Horseshoe* est un processus composé de 3 phases (comme représenté dans la Figure 2.1) : la rétro-ingénierie, la ré-ingénierie et l'ingénierie directe.

FIGURE 2.1: Le processus *Horseshoe* adapté de [Kazman 1998]



La rétro-ingénierie consiste à étudier et analyser le logiciel existant pour en extraire une représentation plus abstraite (*e.g.,* une architecture). La ré-ingénierie est la phase qui consiste à modifier la représentation abstraite définie lors de la rétro-ingénierie pour qu'elle s'adapte à une nouvelle vision (*e.g.,* transformation d'architecture). L'ingénierie directe correspond à la re-génération du code source suivant les transformations réalisées lors de la phase précédente, il s'agit en fait de la phase de développement de projet informatique classique (*e.g.,* cycle en V).

Pour Thales Air Systems, la phase de rétro-ingénierie correspond aux activités d'investigation, permettant de comprendre l'architecture existante du logiciel. Ces activités d'investigation ont été réalisées de manière manuelle. Puis, l'entreprise a réalisé les activités de modification de plateforme, du portage et de l'intergiciel de communication. Ces activités de modification ont été réalisées manuellement, l'entreprise ne souhaitant pas les automatiser car elle les juge trop spécifiques au projet. Durant la phase de ré-ingénierie, l'entreprise a mené l'activité de modification de

l'architecture et la première activité d'assurance qualité (réutiliser le code source). La phase de ré-ingénierie permet une allocation itérative du code source existant dans les composants de l'architecture. Cette allocation de code source est le point central de la démarche de modification d'architecture logicielle menée par l'entreprise, qui souhaite l'automatiser. Finalement, durant la phase d'ingénierie directe, l'entreprise s'est appuyée sur les activités réalisées précédemment pour redévelopper le logiciel suivant la nouvelle architecture. Ce redéveloppement a également été accompagné de la deuxième activité d'assurance qualité (assurer la conservation des résultats).

2.2.4 Ressources humaines

La dernière dimension de description des projets de maintenance concerne les ressources humaines engagées dans le projet. Le projet de maintenance a nécessité différents domaines d'ingénierie, de l'ingénierie logicielle à l'ingénierie système. Étant donné que l'étude décrite dans cette thèse se focalise sur le travail effectué au niveau du logiciel, seules les équipes logicielles seront décrites ici. Durant le reste de cette thèse, les termes ingénieurs et architectes désigneront systématiquement les ingénieurs et architectes logiciels à moins d'être spécifiés autrement.

Compétences des ingénieurs Au total 12 ingénieurs ont travaillé sur le projet de maintenance, dont un ingénieur possédant la double casquette ingénieur/architecte. Sur les onze autres ingénieurs, deux d'entre eux ont également le titre d'expert du domaine d'application de l'entreprise. Au final, tous les ingénieurs ont été choisis pour leurs fortes compétences techniques ainsi que pour leur connaissance du domaine. L'entreprise souhaite inclure les compétences des ingénieurs dans sa démarche, celle-ci doit donc laisser la possibilité aux ingénieurs d'introduire leurs connaissances.

Niveaux d'expérience des ingénieurs Les ingénieurs engagés sur le projet ont tous plus de 5 ans d'expérience en développement, et certains même plus de 15 ans d'expérience. Ils ont également tous plus de 5 ans d'expérience dans le domaine d'application. D'autre part, ces ingénieurs ne possèdent pas d'expérience dans les activités de modification d'architecture. Les ingénieurs ne possèdent également pas d'expérience concernant le logiciel patrimonial impliqué dans le projet de Thales Air Systems. L'entreprise souhaite impliquer des ingénieurs seniors dans sa démarche ainsi elle doit être accessible à de tels ingénieurs.

Attitude face aux projets Nous n'avons pas mené d'études approfondies sur l'attitude des ingénieurs face au projet de maintenance de l'entreprise. Néanmoins,

leur investissement et leur coopération à nos recherches montrent qu'ils n'y ont pas été réfractaire.

Responsabilité des ingénieurs Les 12 ingénieurs ayant travaillés sur ce projet ont été répartis sur les activités de maintenance comme montré dans la Figure 2.2. Le nombre d'ingénieurs alloué à chaque activité est une caractéristique que l'entreprise souhaite garder dans le futur. Les activités de modification d'architecture et de réutilisation du code source (assurance qualité) sont traitées par un architecte, deux experts et un ingénieur.

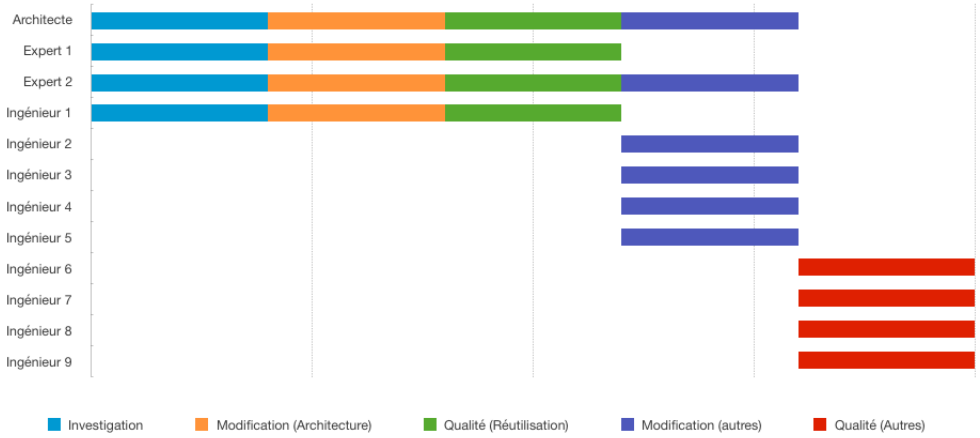


FIGURE 2.2: Répartition des ingénieurs suivant les activités de maintenance

2.2.5 Exigences discriminantes du projet de Thales Air Systems

Les sections précédentes ont identifié les caractéristiques importantes du projet de Thales Air Systems, grâce à l'ontologie des projets de maintenance de Kitchenham *et al.* [Kitchenham 2002]. Ces caractéristiques permettent de définir des exigences discriminants qu'une potentielle technique automatisant la démarche allocation doit respecter afin de pouvoir être retenue :

(exi_i) technique automatique (ou au moins automatisable) ;

(exi_{ii}) technique d'analyse statique (ne nécessitant pas d'exécution du logiciel) ;

(exi_{iii}) technique supportant le passage à l'échelle, *i.e.,* supportant des logiciels de taille supérieure à la centaine de kLoC ;

(exi_{iv}) technique applicable pour la reconnaissance d'une architecture à composants sur un logiciel avec une architecture différente ;

(exi_v) technique adaptable au projet ;

(exi$_{vi}$) technique pour logiciel sans base de données.

L'exigence (exi$_i$) provient du besoin de Thales Air Systems d'automatiser sa démarche d'allocation du code source, ainsi que des caractéristiques liées aux ressources humaines du projet (seulement 4 ingénieurs). L'automatisation de la démarche d'allocation permettra d'allouer un nombre restreint d'ingénieurs à cette activité.

L' exigence (exi$_{ii}$), une technique d'analyse statique, provient de la qualité des artefacts. Puisque le logiciel ne peut pas être exécuté, une approche dynamique ne pourra être appliquée sur le projet de l'entreprise.

L' exigence (exi$_{iii}$), passage à l'échelle de la technique, est lié à la taille du projet qui est importante. D'autre part, les activités de gestion contraignent le temps alloué au projet, ainsi une technique ne supportant pas un passage à l'échelle pourrait mettre en danger le succès du projet.

L' exigence (exi$_{iv}$), une technique pour la reconnaissance d'une architecture à composants, provient des caractéristiques de composition du produit, de l'activité de modification d'architecture et de l'activité d'assurance qualité. L'entreprise souhaite automatiser l'allocation du code source existant dans les éléments d'une nouvelle architecture à composants bien que le logiciel soit actuellement architecturé en module.

L' exigence (exi$_v$), une technique adaptable, provient du fait que l'entreprise souhaite pouvoir appliquer une automatisation des activités et du processus décrits dans la dimension de démarche de maintenance logicielle. En effet, les techniques pouvant automatiser la démarche proposée par l'entreprise doit pouvoir s'adapter à cette démarche.

L' exigence (exi$_{vi}$), un logiciel sans base de données, est lié à la composition du logiciel actuel. Les techniques pouvant automatiser la démarche de l'entreprise doivent en effet respecter la composition du logiciel afin de pouvoir être applicables.

Ces six exigences doivent permettre d'identifier de potentielles techniques qui pourraient être appliquées sur le projet de Thales Air Systems, afin d'automatiser la démarche de l'entreprise.

2.3 Conclusion

Thales Air Systems a réalisé un projet de maintenance d'un de ses logiciels patrimoniaux visant à transformer l'architecture de ce logiciel. Ce chapitre décrit le projet de l'entreprise en suivant l'ontologie des projets de maintenance définie par Kitchenham *et al.* [Kitchenham 2002]. Cette description a mis en avant le fait que l'entreprise souhaite automatiser la démarche d'allocation du code source existant aux composants de la nouvelle architecture de ce logiciel. Dans le but d'automatiser cette démarche, ce chapitre a identifié six exigences qu'une potentielle technique se doit de respecter pour être appliquée sur le projet de Thales Air Systems. Le prochain chapitre détermine le domaine de la littérature associé au besoin de l'entreprise, ainsi que les techniques de ce domaine répondant aux exigences définies dans la section précédente.

CHAPITRE 3

Évolution d'architecture : un état de l'art

Contents

Thales Air Systems a initié un projet d'évolution de l'architecture d'un de ces logiciels. Ce projet étant la première étape d'un plan d'évolution de plusieurs logiciels au sein de l'entreprise, nous cherchons à faciliter l'automatisation et la réplication de la démarche appliquée sur ce premier projet. Le chapitre précédent a mis en évidence des exigences pour la sélection de techniques pouvant faciliter l'automatisation et la réplication de cette démarche.

Ce chapitre étudie la littérature associée à l'évolution logicielle afin d'identifier les approches et les techniques existantes respectant les exigences définies au chapitre précédent. La Section 3.1 définit les concepts associés à l'évolution logicielle. La Section 3.2 décrit ensuite les techniques existantes de reconstruction d'architecture. La Section 3.3 confronte ces techniques avec les exigences définies dans le chapitre précédent, ce qui permet d'identifier trois techniques pouvant potentiellement y répondre. Ces trois techniques sont détaillées dans la Section 3.4. Finalement, la Section 3.5 conclut ce chapitre.

3.1 L'évolution logicielle

Dans le but de déterminer le type d'évolution logicielle ou de maintenance adéquat, Chapin *et al.* classifient les projets d'évolution et de maintenance en se basant sur les travaux qui y sont réalisés [Chapin 2001]. Cette classification étend celle proposée par Kitchenham *et al.* [Kitchenham 2002] et est associée à un arbre de décision, représenté en figure 3.1, permettant d'identifier les types d'évolution

et de maintenance logicielle. Suivre les questions principales (dans les rectangles pointillés) permet de déterminer les artefacts du projet qui seront touchés par l'évolution ou la maintenance. Les feuilles terminales (rectangles en traits pleins) permettent ensuite de déterminer exactement le type d'évolution ou de maintenance logicielle d'un projet. Chapin *et al.* précisent cependant que de tels projets peuvent être de plusieurs types, par exemple lorsqu'un projet vise la mise à jour de la documentation ainsi que l'amélioration des performances d'un logiciel. Typiquement, un projet d'évolution de l'architecture d'un logiciel peut avoir pour but de faciliter la maintenance d'un logiciel (*Groomative*) et d'utiliser de nouvelles technologies (Adaptation).

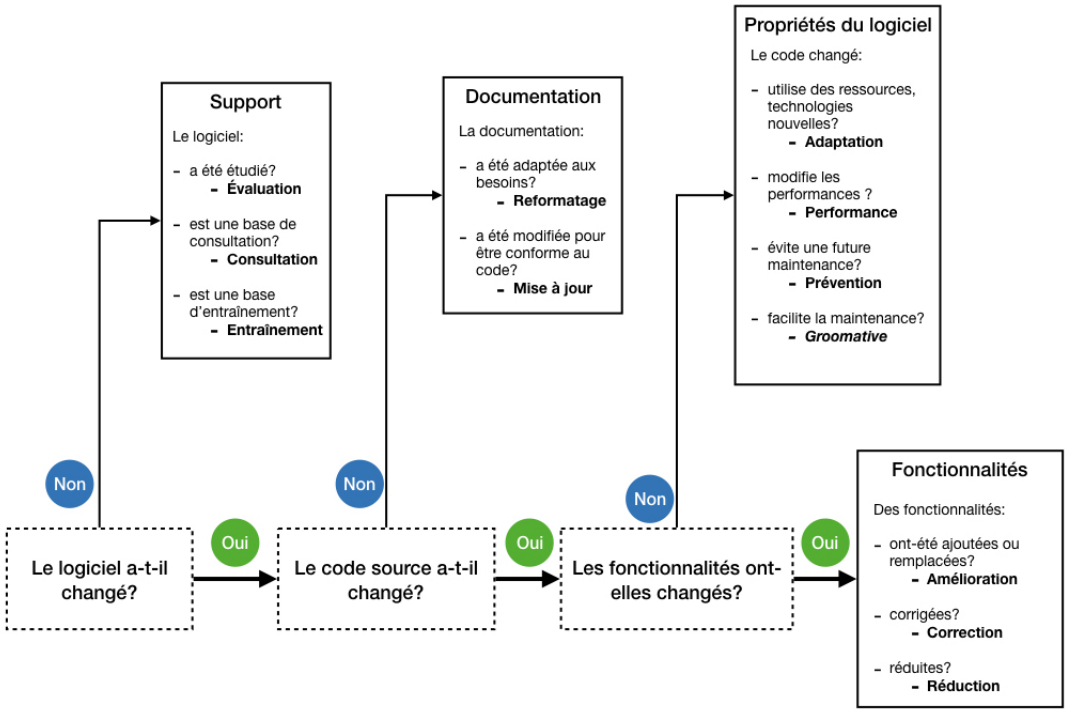


FIGURE 3.1: Arbre de décision du type d'évolution et de maintenance logicielle d'un projet, adapté de [Chapin 2001]

D'autres concepts sont associés à l'évolution et la maintenance logicielle. Chikofsky et Cross ont fourni une première définition de ces termes au début des années 90 [Chikofsky 1990] qui a été ensuite reprise et mise à jour par Mens *et al.* en 2008 [Mens 2008] :

Rétro-Ingénierie La rétro-ingénierie est une démarche d'analyse d'un logiciel ayant pour but d'identifier les éléments du logiciel et leurs relations. Cette identification d'éléments permet ensuite de créer des représentations du lo-

giciel à des niveaux d'abstraction différents (*e.g.,* une architecture) [Chikofsky 1990, Mens 2008]. Chikofsky et Cross identifient alors deux types de rétro-ingénierie : la redocumentation et la restauration de *design*.

Redocumentation La redocumentation est la création ou la révision d'une représentation d'un niveau d'abstraction particulier d'un logiciel [Chikofsky 1990]. La redocumentation permet de modifier une documentation existante en y ajoutant ou en modifiant les éléments qui la composent, par exemple en modifiant la représentation graphique d'une fonctionnalité.

Restauration de *design* (ou reconstruction d'architecture) La restauration de *design* est un sous-ensemble de la rétro-ingénierie qui permet de recréer des abstractions de conception grâce à une combinaison :

— du code du logiciel ;

— de la documentation sur le *design* existant ;

— de l'expérience personnelle d'un ingénieur ;

— de la connaissance domaine d'un ingénieur.

La restauration de *design* permet donc de retrouver un *design* (ou une architecture [Kazman 1996]) d'un logiciel à partir de différentes informations. De plus, la restauration de *design* est également appelée reconstruction d'architecture [Kazman 1998] dans la littérature. En effet, la reconstruction d'architecture est définie comme la reconstruction d'éléments architecturaux d'un logiciel à partir de son implémentation [Krikhaar 1999, Kazman 2003]. Cette définition correspond donc à la définition de la restauration de *design*.

Ré-ingénierie (ou rénovation d'architecture) La ré-ingénierie est le processus d'évaluation et de modification interne d'un logiciel [Chikofsky 1990] sans en changer les fonctionnalités. Le terme rénovation est parfois employé pour désigner la ré-ingénierie.

Restructuration La restructuration est définie comme la transformation d'artefact logiciel, d'une certaine forme à une autre, au même niveau d'abstraction [Canfora 2000]. Bien que les artefacts du logiciel soient transformés, le comportement externe du logiciel ne doit pas l'être, *i.e.,* ses fonctionnalités restent les mêmes [Chikofsky 1990, Mens 2008]. Mens *et al.* emploie le terme *refactoring* pour désigner la spécialisation de la restructuration aux logiciels orientés objet [Mens 2008]. De plus, le terme de "remodularisation" est parfois utilisé à la place de "restructuration", bien que ce dernier lui soit préféré, car plus générique [Anquetil 2011]. Au final, le terme restructuration désigne tout autant le *refactoring* que la remodularisation et correspond à la modification d'élément logiciel d'une forme à une autre sur le même niveau d'abstraction (*e.g.,* ancien élément architectural vers un nouvel élément architectural)

Ingénierie directe L'ingénierie directe est le processus de création d'un logiciel en partant d'un haut niveau d'abstraction (*e.g.,* exigences et fonctionnalités) pour descendre au niveau de l'implémentation du logiciel (*e.g.,* le code).

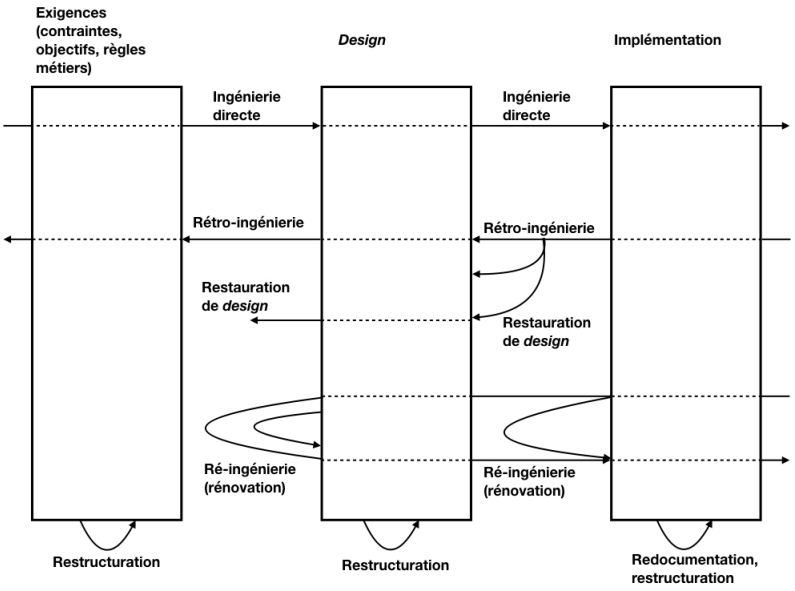


FIGURE 3.2: Relations entre les concepts d'évolution et de maintenance logicielle [Chikofsky 1990]

Chikofsky et Cross définissent également des relations entre ces différents concepts, représentées dans la figure 3.2. Cette figure représente ces concepts et leurs relations dans trois niveaux d'abstractions différents : les exigences, le *design* et l'implémentation. La restauration de *design* est représentée de manière légèrement différente des autres concepts, car comme indiqué plus tôt il s'agit d'un sous-ensemble d'un autre concept (la rétro-ingénierie).

Selon Chikofsky et Cross, la ré-ingénierie (également appelé rénovation) "implique une certaine forme de rétro-ingénierie suivie par de la restructuration et de l'ingénierie directe". Ainsi la rénovation de l'architecture d'un logiciel en partant de son implémentation (code source) peut nécessiter d'appliquer des phases :

— de restauration de *design* (*e.g.,* retrouver une architecture en cohérence avec le code source) ;

— de restructuration de cette architecture (*e.g.,* évolution vers une nouvelle architecture) ;

— d'ingénierie directe (*e.g.,* redévelopper le code suivant la nouvelle architecture).

Appliquer chacune de ces phases pour rénover l'architecture d'un logiciel n'est pas obligatoire. Cela dépend des informations disponibles au départ du projet et de la démarche réalisée, comme indiqué par Chapin *et al.* dans la figure 3.1.

Selon sa définition, la restructuration est une transformation [Chikofsky 1990] d'éléments sur un niveau d'abstraction donné. Lorsque ce niveau d'abstraction correspond à l'architecture (ou *design*), la restructuration devient une transformation d'architecture. La transformation d'architecture permet d'appliquer des transformations (*e.g.,* regroupement, dissociation) d'éléments architecturaux représentés sous la forme d'un modèle (ou d'un graphe) [Fahmy 2000]. Il arrive parfois que la restructuration vise la transformation d'éléments architecturaux à plusieurs niveaux d'abstraction différents [Zhang 2010a]. Dans de tels cas, les transformations réalisées au niveau architectural peuvent être propagées automatiquement aux autres niveaux, dès lors que des liens de traçabilité ont été définis entre ces niveaux [Zhang 2010b]. Des langages de description d'architecture permettent alors de modéliser les différents niveaux d'architecture, les liens de traçabilité entre ces niveaux et les transformations architecturales [Zhang 2010b].

La phase d'ingénierie directe suivant la restructuration d'architecture correspond à la propagation, au niveau du code source, des transformations réalisées pendant la restructuration. Cette propagation correspond en général à des transformations de l'organisation physique du logiciel (*e.g.,* déplacement ou suppression de fichiers) ou à de la modification de code source (*e.g.,* extraction de fonctions) [Tsantalis 2011, Santos 2015b]. Cela montre que, depuis sa définition par Chikofsky et Cross, l'ingénierie directe a évolué pour prendre en compte des activités n'ayant pas trait à de la création [Kazman 2005].

Par la suite, un nombre important de recherches [Murphy 1995, Christl 2006, Zheng 2012] a introduit un nouveau concept lié à l'évolution logicielle : **l'allocation de code source à une architecture**. Ce concept est apparu dans la littérature après les travaux de Chikofsky et Cross, il n'apparaît donc pas dans la figure 3.2. L'allocation de code source à une architecture est utilisée pour créer des liens de traçabilité entre le code source et les éléments d'une architecture [Murphy 1995, Koschke 2000, Christl 2006, Zheng 2012]. Lorsque ces éléments appartiennent à l'architecture existante du logiciel, ce concept permet de reconstruire une architecture, *i.e.,* de la retrouver dans le code source [Murphy 1995]. De ce fait, l'allocation de code source aux éléments d'une architecture existante fait partie de la rétro-ingénierie. Lorsque ces éléments appartiennent à une architecture différente, il s'agit de ré-allouer le code source à des éléments architecturaux différents [Koschke 2000, Christl 2006]. Comme la ré-allocation de code source modifie le logiciel ce concept fait partie de la ré-ingénierie.

Telle qu'utilisée dans la littérature, l'allocation de code source permet de mettre en évidence une architecture logicielle existante ou future en regroupant des élé-

ments de code similaires. L'allocation de code source peut donc être associée à la reconstruction d'architecture, car ces deux concepts permettent de réaliser ces regroupements d'éléments. Ainsi, l'allocation peut également être une étape de la rénovation, telle que définit par Chikofsky et Cross. De plus, lorsque la rénovation d'un logiciel nécessite de réutiliser des éléments de code source, l'allocation est souvent utilisée [Murphy 1995, Koschke 2000, Christl 2006, Constantinou 2015].

Comme décrit au chapitre précédent, Thales Air Systems a fait évoluer son logiciel en allouant le code source de celui-ci aux éléments d'une nouvelle architecture. Cette évolution a été réalisée par l'entreprise en appliquant trois phases : (1) compréhension de l'architecture existante ; (2) allocation du code source existant à une nouvelle architecture ; (3) redéveloppement. Ces trois phases correspondent à celles décrites par Chikofsky et Cross pour la rénovation. Thales Air Systems a donc réalisé une rénovation d'architecture logicielle pour son projet.

3.2 Approche pour l'évolution d'architecture

La rénovation d'architecture telle que réalisée par l'entreprise se base sur l'allocation du code source à des éléments d'architecture. Allouer du code source à une architecture nécessite les mêmes techniques que la reconstruction d'architecture, puisqu'il faut identifier des éléments de code similaires. La littérature fournissant peu de techniques d'allocations, mais de nombreuses techniques de reconstruction d'architecture, nous nous concentrons donc sur celles-ci pour faciliter l'automatisation et la réplication de la démarche de rénovation de l'entreprise.

Plusieurs articles ont cherché à lister les techniques existantes de reconstruction d'architecture [O'Brien 2002, Gerardo 2007, Ducasse 2009]. Notamment Ducasse et Pollet [Ducasse 2009] ont listé et proposé une catégorisation des techniques de reconstruction d'architecture. Selon ces derniers, ces techniques sont classées suivant trois catégories : (1) quasi-manuelles ; (2) semi-automatiques ; (3) quasi-automatiques.

3.2.1 Techniques quasi-manuelles

Les techniques quasi-manuelles de reconstruction d'architecture se basent sur une identification manuelle des éléments d'une architecture. L'identification manuelle des éléments d'architecture se fait par deux procédés : (1) par construction ; (2) par exploration.

Construction Les techniques de construction fonctionnent en allouant manuellement du code source à une représentation des éléments d'architecture. Cette allocation permet par la suite de calculer des dépendances entre les éléments

d'architecture à partir des dépendances réelles entre les éléments ayant été alloués, comme le *Reflexion Model* par exemple [Murphy 1995].

Exploration Les techniques d'exploration fournissent des visualisations agrégeant les informations du code source à des niveaux plus abstraits du logiciel. Par exemple, au lieu de présenter chaque fichier source avec leur nombre de lignes de codes, ces informations sont agrégées et présentées au niveau des dossiers contenant ces mêmes fichiers [Pinzger 2005, Lungu 2006]. Ces représentations permettent d'identifier visuellement des ensembles d'éléments de code source pouvant correspondre à des éléments d'architecture.

3.2.2 Techniques semi-automatiques

Les techniques semi-automatiques de reconstruction d'architecture se basent sur des règles de reconnaissance des éléments d'architecture servant à leur identification. Ces règles sont créées par un ingénieur et sont ensuite appliquées sur le logiciel entier pour retrouver automatiquement les éléments d'architecture, sous la forme de requêtes. Les techniques semi-automatiques sont séparées en six sous-catégories : (1) requêtes relationnelles ; (2) requêtes logiques ; (3) requêtes lexicales et structurelles ; (4) *recognizers* ; (5) reconnaissance de patrons de graphes ; (6) moteurs d'états.

Requêtes relationnelles Les requêtes relationnelles se basent sur l'algèbre relationnelle de Tarski [Tarski 1941] pour identifier les éléments d'architectures [Holt 1998]. L'algèbre relationnelle permet de définir des ensembles de transformations (abstractions ou décompositions) pouvant être répétées [Holt 1998]. Typiquement, les requêtes relationnelles sont utilisées sur des bases de données entités-relations pour réaliser des regroupements hiérarchiques de données [Ducasse 2009].

Requêtes logiques Les requêtes logiques se basent sur des systèmes de prédicats à retrouver dans une base de connaissance, de manière identique au langage Prolog. Les techniques utilisant les requêtes logiques ont notamment été utilisées pour l'identification de patrons de conceptions usuels (types GOF) [Guéhéneuc 2004]. Elles ont également été utilisées par Mens *et al.* pour la reconnaissance d'entités de code source structurellement similaire appelées *Intensional Views* [Mens 2006].

Requêtes lexicales Les requêtes lexicales se basent sur les informations textuelles du code source pour identifier les éléments d'architecture. Les requêtes lexicales sont appliquées sous forme de recherche textuelle afin de retrouver les éléments d'architecture [Sim 1999, Marcus 2003, Poshyvanyk 2007]. Par exemple, pour retrouver les fichiers de code source qui implémentent un service de calcul de coût, une requête sur le mot *"cost"* pourra être effectuée.

Recognizers Les *recognizers* sont des outils provenant des machines à états finis et qui permettent d'y détecter des séquences d'états. Dans le cadre de l'identification d'éléments d'architecture, les *recognizers* sont utilisés pour reconnaître des styles architecturaux (*e.g.,* client-serveur [Taylor 2009]) et/ou des patrons de conceptions [Fiutem 1999].

Reconnaissance de patrons de graphes Les techniques de reconnaissance de patrons de graphes représentent le logiciel et les requêtes sous la forme de graphe et appliquent une méthode de comparaison entre ces graphes. Chaque sous-partie du graphe du logiciel correspondant au graphe d'une requête est alors identifiée comme appartenant à un élément d'architecture [Guo 1999, Sartipi 2003b]. Typiquement, retrouver les composantes fortement connexes d'un graphe d'appel de fonction permet d'identifier les fonctions fortement couplées d'un logiciel.

Moteurs d'états Les moteurs d'états se basent sur une représentation dynamique du logiciel afin de vérifier la conformité des éléments architecturaux et de leurs connecteurs [Yan 2004]. Les techniques de ce genre utilisent des descriptions de machines à états pour extraire des événements architecturaux (*e.g.,* appels de méthodes, instanciation d'objets). Lorsqu'une exécution satisfait une de ces descriptions, les événements architecturaux associés sont extraits et présentés à l'utilisateur [Seriai 2014]. La différence entre les moteurs d'états et les *recognizers* reposent sur le fait que les premiers fonctionnent en dynamique (*i.e.,* ils nécessitent l'exécution du logiciel) alors que les seconds travaillent en statique.

3.2.3 Techniques quasi-automatiques

Les techniques quasi-automatiques sont des techniques qui, bien que guidées par les ingénieurs, réalisent un traitement complet sans apport humain intermédiaire. Ces techniques se séparent en quatre sous-catégories : (1) analyse de concepts ; (2) dominance ; (3) matrices de dépendances structurelles ; (4) regroupements [1].

Analyse de Concepts L'analyse de concepts [Tilley 2003, Greevy 2005, Bhatti 2012] sert principalement à identifier des patrons de conception [Arévalo 2004, Azmeh 2008] ou des fonctionnalités [Greevy 2005, Al-Msie'Deen 2013]. Ce genre de technique calcule tous les groupes possibles d'éléments du logiciel, appelées "objets" (*e.g.,* les classes) qui ont certaines propriétés, appelées "attributs", en commun (*e.g.,* les classes dont elles dépendent), et les restitue sous forme d'un treillis. L'analyse de concepts va ensuite calculer tous les groupes d'objets qui ont des attributs en commun (*e.g.,* toutes les classes qui déclarent des variables, puis toutes les classes qui déclarent des fonctions) et

1. *clustering* en anglais

les restituer à l'utilisateur. Cette technique a été utilisée dans le but de localiser des fonctionnalités (*Feature Location*) [Azmeh 2008] ou localiser des services [Khadka 2011].

Dominance Les techniques de dominance se basent sur le concept de dominance de la théorie des graphes [Cimitile 1995] afin de trouver des ensembles d'éléments logiciels qui sont déclenchés par le même événement. Comme pour les techniques de reconnaissance de patrons de graphes, les techniques de dominance représentent les logiciels sous la forme d'un graphe (*e.g.,* graphe d'appels). À partir d'un sommet donné du graphe, la dominance retourne tous les éléments atteignables uniquement depuis ce sommet donné. Ces techniques permettent d'identifier des groupes d'éléments de code source qui dépendent du même événement déclencheur, appelés composants passifs par Lundberg et Löwe [Lundberg 2003].

Matrice de dépendances structurelles Les matrices de dépendances structurelles sont utilisées pour analyser les dépendances architecturales dans les architectures logicielles. Ces techniques sont adaptées à la reconnaissance de couches dans les logiciels (*e.g.,* pour une architecture logicielle n-tiers) [Sullivan 2001, Sangal 2005, Breivold 2008].

Regroupement Les techniques de regroupement [Anquetil 2009, Koschke 2000] visent à grouper ensemble des éléments qui sont similaires, suivant une métrique donnée. Ce genre de technique est applicable sur un modèle ou graphe extrait d'une analyse statique du logiciel (parsing du code source) [Mancoridis 1998, Koschke 2000, Maqbool 2007, Garcia 2013] ou d'une analyse dynamique du logiciel (*e.g.,* traces d'exécution) [Allier 2009, Allier 2010, Allier 2011].

3.3 Les techniques de reconstruction d'architecture adaptées aux besoins de Thales Air Systems

La section précédente a montré que la littérature offre un grand nombre de solutions au problème de reconstruction d'architecture. Cependant, ces solutions doivent répondre aux exigences définies au chapitre précédent afin de pouvoir être retenues pour automatiser l'évolution de l'architecture du logiciel de Thales Air Systems. Nous rappelons ici les exigences ayant été définies en conclusion du chapitre précédent :

— (exi$_i$) technique automatique (ou au moins automatisable)

— (exi$_{ii}$) technique statique,

— (exi$_{iii}$) technique à grande échelle, *i.e.,* supportant des logiciels de taille supérieure ou égale à 300 kLoC.

— (exi_{iv}) technique applicable pour la reconnaissance d'une architecture à composants sur un logiciel avec une architecture différente

— (exi_{v}) technique adaptable (à la démarche proposée par l'entreprise)

— (exi_{vi}) technique pour logiciel sans base de données

A priori, aucune technique ne vérifiera l'exigence d'adaptabilité ((exi_{v})) à la démarche de l'entreprise, car ces techniques n'ont pas été appliquées pour cette démarche. Bien que cette exigence soit importante, les techniques n'auront pas à la vérifier pour le moment Ainsi, si une technique vérifie les exigences (exi_{i}), (exi_{ii}), (exi_{iii}), (exi_{iv}) et (exi_{vi}), elle sera retenue.

3.3.1 Techniques quasi-manuelles pour Thales Air Systems

L'exigence (exi_{i}) nécessite que les techniques pouvant être utilisées sur le projet de l'entreprise soient automatiques ou au moins automatisables (*i.e.,* dont on peut automatiser tout ou partie). La plupart des techniques quasi-manuelles décrites dans la section précédente ne présentent pas d'automatisation possible dans la littérature. À notre connaissance, la seule technique quasi-manuelle dont le potentiel d'automatisation a été prouvé est le *Reflexion Model*, proposé par Murphy *et al.* dans [Murphy 1995]. En effet, le *Reflexion Model* est une technique ayant été partiellement automatisée par Murphy et Notkin [Murphy 1997] et par Koschke et Simon [Koschke 2003]. Le *Reflexion Model* comme proposé par Murphy *et al.* a été appliqué dans des cas d'études se basant uniquement sur des informations statiques du logiciel [Murphy 1997], respectant donc l'exigence (exi_{ii}). La possibilité d'appliquer cette technique à plus grande échelle (exigence (exi_{iii})) a été prouvée par son application sur des logiciels de différentes tailles (500 kLoC [Koschke 2003], $1,2$ MLoC [2] [Murphy 1997], $1,5$ MLoC [Knodel 2002]). L'automatisation du *Reflexion Model* par Kosche et Simon avait pour but de retrouver une architecture à composants à partir d'un logiciel ayant une architecture différente (confirmant donc l'exigence (exi_{iv})). De plus, toutes ces applications ont eu lieu sur des logiciels ne possédant pas de bases de données ce qui confirme la dernière exigence. Le *Reflexion Model* respecte toutes les exigences du projet de Thales Air Systems étudiées ici. Ainsi, cette technique est retenue pour l'automatisation de ce projet, sous réserve de pouvoir être adaptable à la démarche proposée par l'entreprise.

3.3.2 Techniques semi-automatiques pour Thales Air Systems

En raison de leur nature, les techniques semi-automatiques sont partiellement automatisées et répondent toutes à la première exigence énoncée en début de section.

2. *million of lines of code*

Requêtes relationnelles De par leur définition, les requêtes relationnelles ont été appliquées sur des logiciels comportant des bases de données. Cela contredit l'exigence (exi$_{vi}$) et ainsi les requêtes relationnelles n'ont pas été retenues pour le projet de l'entreprise.

Requêtes logiques Les requêtes logiques nécessitent de pouvoir définir un ensemble de prédicats contenant toutes les implémentations possibles des patrons de conception. Cependant, définir toutes ces implémentations entraîne une explosion combinatoire de la complexité de cette technique [Guéhéneuc 2004, Guéhéneuc 2008]. Cela contredit donc l'exigence (exi$_{iii}$) du projet de l'entreprise visant un passage à l'échelle, ce qui implique que les requêtes logiques n'ont pas été retenues pour le projet de Thales Air Systems.

Requêtes lexicales Par leur nature, les requêtes lexicales se basent sur des informations textuelles et en font ainsi une approche statique, ce qui respecte l'exigence (exi$_{ii}$). Le passage à l'échelle (exigence (exi$_{iii}$)) de telles techniques a été prouvé par Poshyvanyk *et al.* [Poshyvanyk 2006], avec un cas d'étude sur un logiciel de plus de $3,7$ MLoC. Plusieurs cas d'études disponibles en littérature se concentrent sur la reconnaissance des fonctionnalités (*feature location*) [Marcus 2004, Marcus 2005, Poshyvanyk 2006, Bavota 2010]. D'autres cas d'études ont pour but la reconnaissance de composants [Braga 2001, Sugumaran 2003, Alnusair 2010] dans des applications sans base de données, ce qui confirme les exigences (exi$_{iv}$) et (exi$_{vi}$). Les requêtes lexicales respectent la totalité des exigences étudiées ici. Ainsi, cette technique est retenue pour automatiser le projet de l'entreprise, sous réserve de respecter l'exigence d'adaptabilité. La technique, utilisant les requêtes lexicales, la plus reconnue est la recherche d'information (*information retrieval*) et a été utilisée par Marcus et Poshyvanyk pour chacun des articles cités précédemment.

Recognizers Les *recognizers* sont utilisés dans plusieurs outils de reconstruction d'architecture [Yeh 1997, Fiutem 1999, Guo 1999]. Cependant, Mendonça et Kramer [Mendonça 2001] ont noté que les *recognizers* souffrent d'un problème lors d'un passage à grande échelle (grande variabilité des implémentations d'une fonctionnalité) ce qui infirme l'exigence (exi$_{iii}$). Ainsi, les *recognizers* ne sont pas retenus pour le projet de Thales Air Systems.

Reconnaissance de patrons de graphes Les techniques utilisant la reconnaissance de patrons de graphes sont également sujettes à des problèmes lors de leur application sur de grands projets (passage à l'échelle). En effet, Sartipi [Sartipi 2003a] explique dans sa thèse de doctorat qu'un logiciel de grande taille augmente significativement le temps de calcul de cette technique (plusieurs dizaines d'heures pour des logiciels d'une dizaine de kLoC). Cela empêche la technique d'être performante avec de grands logiciels ce qui contredit l'exi-

gence (exi_{iii}). Ainsi la reconnaissance de patrons graphe n'est pas retenue pour le projet de l'entreprise.

Moteurs d'états Comme indiqué dans la Section 3.2.3, les moteurs d'états sont des techniques nécessitant une exécution du logiciel. Or cela contredit l'exigence (exi_{ii}) et implique donc que les moteurs d'états ne sont pas applicables dans le cadre du projet de Thales Air Systems.

3.3.3 Techniques quasi-automatiques pour Thales Air Systems

De par leur nature, les techniques quasi-automatiques sont presque entièrement automatisées et répondent donc toutes à la première exigence énoncée en début de section.

Analyse de Concepts Certaines études [Siff 1999, Bhatti 2012] ont montré que l'analyse de concepts est une technique dont l'application sur de grands projets reste difficile. Notamment, Siff et Reps [Siff 1999] mettent en évidence le fait que l'analyse de concepts peut produire beaucoup plus de résultats (plus de 13 000 concepts) que d'entrées fournies (189 objets et 32 attributs). L'analyse de concepts ne répond donc pas à l'exigence de passage à l'échelle (exigence (exi_{iii})) et n'est pas retenue comme technique applicable pour le projet de Thales Air Systems.

Dominance Lundberg et Löwe ont montré que les techniques de dominance ne sont pas adaptées à la reconnaissance d'une architecture entière [Lundberg 2003]. Cela contredit autant l'exigence (exi_{iv}) que l'exigence (exi_{iii}), ainsi la dominance n'est pas retenue pour le projet de l'entreprise.

Matrice de dépendances structurelles Bien que les matrices de dépendances structurelles puissent être calculées sur de grands logiciels, Ducasse *et al.* [Ducasse 2007] ont montré que l'exploitation de ces matrices sur de grands logiciels reste complexe. Ainsi cette technique souffre de difficulté de passage à l'échelle ce qui contredit l'exigence (exi_{iii}). Cette technique n'est donc pas adaptée au projet de Thales Air Systems.

Regroupement Le regroupement est une technique quasi-automatique ayant été appliquée sur des logiciels en se basant sur leurs informations statiques [Mancoridis 1998, Koschke 2000, Maqbool 2007, Garcia 2013], comme requis par l'exigence (exi_{ii}). Koschke a montré dans sa thèse de doctorat [Koschke 2000] que le regroupement peut être appliqué dans le cadre de la reconnaissance d'une architecture à composants dans un logiciel avec une architecture différente et sans bases de données (exigence (exi_{iv}) et (exi_{vi})). La possibilité d'appliquer le regroupement sur de grands projets a été démontrée par des cas d'études sur des logiciels de plus d'une centaine de kLoC [Koschke 2000, Koschke 2003, Maqbool 2007]. Le regroupement respecte donc

l'exigence (exi$_{iii}$). Finalement, cette technique respecte toutes les exigences du projet de Thales Air Systems étudiées dans cette section. Koschke décrit une technique de regroupement qu'il a appliquée sur un projet présentant de fortes similitudes avec celui de Thales Air Systems (reconnaissance de composants dans un logiciel critique, embarqué, avec une architecture différente et développée en Ada 83). Cette technique a été retenue pour être appliquée sur le projet de l'entreprise, car ces similitudes laissent à penser que l'adaptabilité de la technique sera assurée.

3.4 Techniques retenues pour le projet de Thales Air Systems

La section précédente a permis de mettre en évidence trois techniques pouvant être appliquées sur le projet de Thales Air Systems afin d'automatiser la démarche d'évolution d'architecture décrite en Section 2.2.3. La présente section détaille ces trois techniques (*Reflexion Model*, recherche d'information, regroupement)

3.4.1 Reflexion Model

Le *Reflexion Model* a pour but d'extraire des représentations d'une architecture d'un logiciel, appelées points de vue architecturaux (*e.g.,* composants, fonctionnalités, organisation physique sur des cartes embarquées). Cette technique, décrite par Murphy *et al.* [Murphy 1995], est principalement utilisée pour visualiser un modèle architectural espéré (ou architecture espérée) d'un système. Elle vérifie que cette architecture espérée correspond effectivement au code source actuel. Cette vérification se fait grâce à un *mapping* entre un modèle représentant le code source (modèle source) et un modèle architectural abstrait (*i.e.,* l'architecture espérée).

Comme décrit dans la Figure 3.3, le *Reflexion Model* est séparé en cinq étapes :

(rm$_i$) Les ingénieurs formalisent un modèle architectural espéré qui contient des éléments architecturaux de haut niveau (*e.g.,* des composants), qu'ils pensent être présents dans le logiciel. Ce modèle décrit également les relations, interdites ou obligatoires, entre les éléments architecturaux. Habituellement, ce modèle provient de la documentation disponible, mais aussi de la connaissance métier et de l'expertise des ingénieurs.

(rm$_{ii}$) La deuxième étape du *Reflexion Model* consiste à extraire un modèle source à partir du système logiciel. Cette extraction peut être réalisée grâce à une analyse statique du code source logiciel ou grâce à une analyse dynamique (*e.g.,* traces d'exécutions). Le modèle extrait décrit les éléments logiciel du

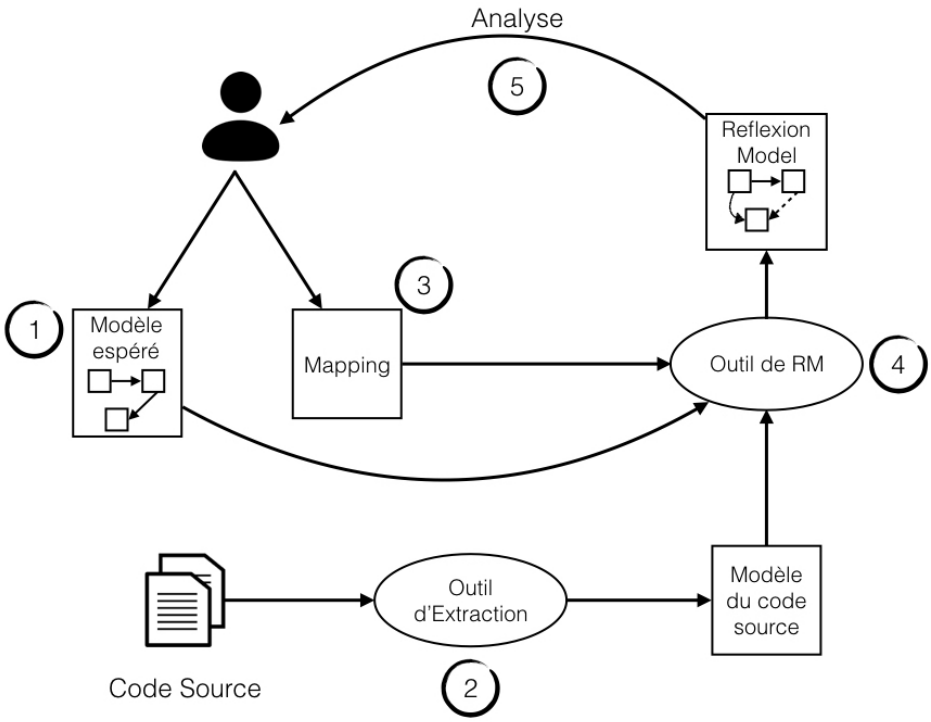


FIGURE 3.3: L'approche du *Reflexion Model* adaptée de Murphy *et al.* [Murphy 1995]

code (*e.g.,* les paquetages, classes, méthodes *etc.*) ainsi que leurs relations (*e.g.,* liens d'invocations entre méthodes).

(rm_{iii}) Les ingénieurs définissent un *mapping* entre les éléments du modèle espéré et les entités du modèle source. Ce *mapping* peut être défini automatiquement (*e.g.,* grâce à des conventions de nommage dans le code), ou manuellement.

(rm_{iv}) Dans, la quatrième étape, un outil vérifie si le *mapping*, réalisé à l'étape 3 entre le modèle source et le modèle espéré, satisfait toutes les contraintes définies lors de l'étape 1.

(rm_{v}) Finalement, les ingénieurs analysent les résultats obtenus lors de l'étape 4 et statuent si ces résultats satisfont ou non leur vision. Dans le cas où ils jugeraient les résultats non satisfaisants, les ingénieurs peuvent changer soit le *mapping* (et reprendre le *Reflexion Model* à l'étape 3), soit le modèle espéré (et donc reprendre le *Reflexion Model* depuis l'étape 1). Ce choix dépend entièrement des ingénieurs et du but à atteindre.

3.4.2 Recherche d'Information

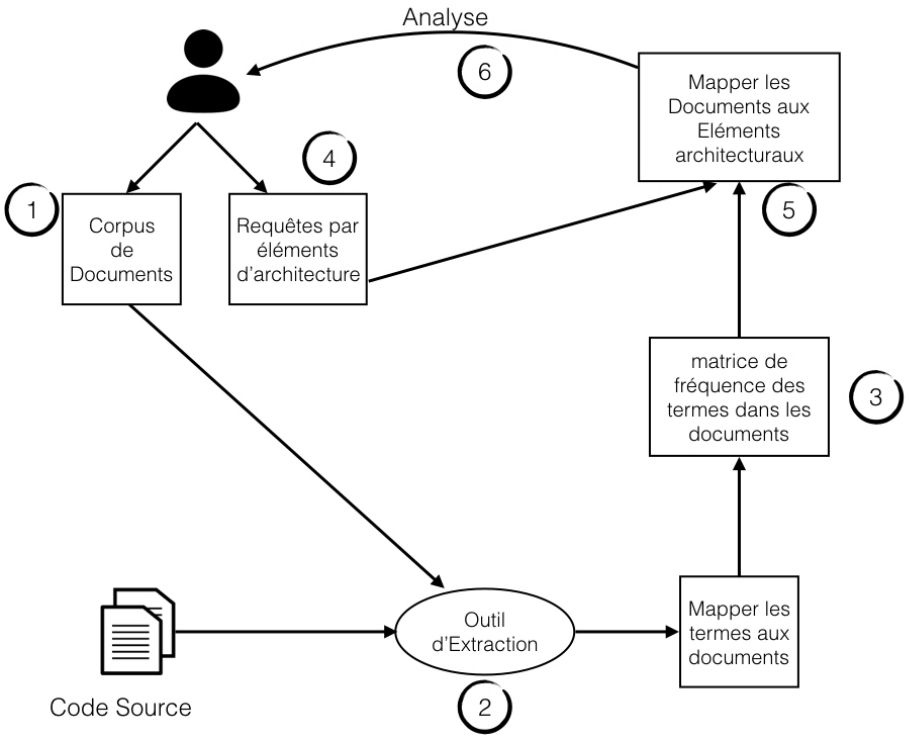


FIGURE 3.4: L'approche de recherche d'information pour la recherche d'éléments architecturaux

La recherche d'information est une technique qui retrouve, dans un corpus de documents, ceux qui correspondent le mieux à une requête utilisateur. Une requête est un ensemble de mots devant être retrouvés dans les documents du corpus.

Initialement, la recherche d'information a été définie pour des recherches textuelles dans plusieurs fichiers, mais a été adaptée, par la suite, à la recherche dans du code source [v. Rijsbergen 1979]. Aujourd'hui, elle est utilisée dans le cadre de la recherche d'éléments architecturaux [Maletic 2000, Maletic 2002, Poshyvanyk 2006, Bavota 2010]. Afin de réaliser cette recherche, on associe aux éléments architecturaux des requêtes qui sont composées de mots (ou termes) caractéristiques de ces éléments (*e.g.,* pour un élément traitant de la vitesse, les mots "speed" et "vitesse"). Ensuite, les entités du code source qui correspondent le mieux à ces termes sont associées à l'élément architectural correspondant.

L'utilisation de la recherche d'information pour la recherche d'éléments architecturaux se décompose en six étapes (résumées dans la Figure 3.4) :

(ri_i) La première étape consiste à identifier le corpus de documents. Le corpus de documents peut être constitué de tous les fichiers de code source, ou d'éléments logiciels (*e.g.,* classes, méthodes).

(ri_{ii}) La seconde étape consiste à extraire les informations textuelles de chacun des documents du corpus. Dans le cas d'éléments logiciels (comme une méthode), cela revient à extraire les identifiants (*e.g.,* les noms de variables, les noms des procédures *etc.*) et les commentaires pour chacun de ces éléments. Cette étape contient également une activité (automatique) de normalisation des informations textuelles. Cette normalisation consiste en général à : découper les identifiants selon des conventions (*e.g.,* le "Camel case" qui transforme "allAdaVariables" en "all","Ada","Variables") ; retirer les mots vides [3] (*e.g.,* mots communs de l'anglais comme "an", "the", ou mots clés des langages de programmation comme "for", "if", etc.) ; raciniser (*i.e.,* identifier la racine d'un mot) les termes en retirant les préfixes et suffixes communs (*e.g.,* terminaison de verbes, accords en genre et nombre).

(ri_{iii}) Une technique d'indexation est appliquée (typiquement la méthode TF-IDF [Chowdhury 2010]) afin d'obtenir une matrice de la fréquence d'apparition des termes en fonction des documents.

(ri_{iv}) Dans la quatrième étape, des requêtes sont définies pour chaque élément architectural. Une requête est un ensemble de termes que doivent satisfaire les documents. Généralement, les requêtes permettent de distinguer et de caractériser les éléments architecturaux.

(ri_v) Les requêtes sont alors appliquées et donnent une distance (ou, inversement, une similarité) entre les documents et les éléments architecturaux. Les documents sont alors, en général, rangés du plus proche de la requête au plus

3. *stopwords* en anglais

éloigné. Cela permet ensuite d'associer chaque document à l'élément dont il est le plus proche.

(ri_{vi}) Finalement, les ingénieurs analysent le *mapping* résultant de cette technique et peuvent raffiner leurs requêtes (et repartir de l'étape 4), soit pour exclure certains documents d'un élément architectural donné soit pour associer de nouveaux documents aux éléments architecturaux.

3.4.3 Regroupement

Koschke définit le regroupement suivant une similarité, comme une approche qui :

> "groupe des entités de base (sous-programmes, types utilisateurs, et variables globales) suivant la proportion de fonctionnalités (entités auxquelles elles accèdent, leurs noms, leurs fichiers de définition, etc.) qu'elles ont en commun" [4] [Koschke 2000].

Koschke a appliqué son travail de regroupement afin d'extraire des composants d'applications patrimoniales développées en C et en Ada. Cette technique de regroupement se décompose en cinq étapes :

(rg_{i}) La première étape consiste à extraire un modèle du code source du système : les éléments logiciels (*e.g.,* classes) et leurs relations (*e.g.,* imports).

(rg_{ii}) Ensuite, une métrique de similarité est définie. Cette métrique (définie dans [Koschke 2000, Chapitre 7]) se base sur les relations extraites à l'étape précédente.

(rg_{iii}) L'algorithme de regroupement mis en place dans le cadre de cette technique est itératif. Il est donc nécessaire de définir un critère d'arrêt, qui peut être objectif (*e.g.,* un nombre d'itérations) ou subjectif (*e.g.,* un choix utilisateur).

(rg_{iv}) Cette étape correspond à l'exécution de l'algorithme de regroupement. Cet algorithme forme des groupes d'éléments logiciels les plus proches, entre eux, en se basant sur les techniques de regroupement hiérarchique. Initialement, chaque élément logiciel est considéré comme un groupe atomique. La métrique de similarité est alors calculée entre chacun de ces groupes atomiques. Ensuite à chaque itération, les deux groupes les plus similaires (ceux pour lesquels la métrique est la plus grande) sont groupés ensemble, puis la similarité est recalculée entre ce nouveau groupe et tous les anciens. La Figure 3.5 montre un exemple de l'exécution de cet algorithme avec 4 éléments

4. "*group base entities (subprograms, user-defined types, and global variables) according to the proportion of features (entities they access, their name, the file where they are defined, etc.) they have in common.*"

logiciels et un critère d'arrêt arbitrairement choisi à "avoir seulement 2 groupes".

(rg_v) Finalement, la dernière étape est d'analyser les groupes formés par cet algorithme. Si les groupes ne sont pas considérés comme satisfaisants par les ingénieurs, alors ils peuvent soit changer le critère d'arrêt (et reprendre depuis l'étape 3) soit changer la métrique de similarité (et reprendre depuis l'étape 2).

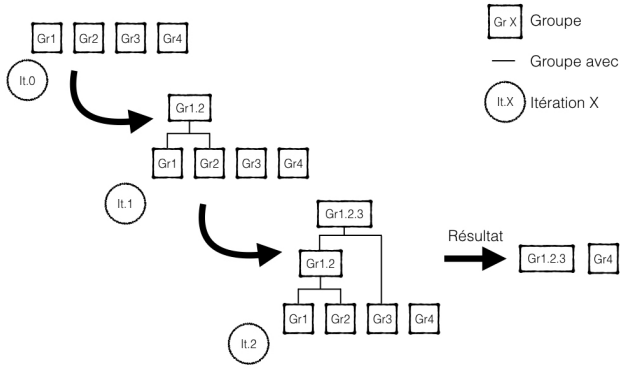


FIGURE 3.5: Exemple d'exécution de l'algorithme de regroupement (le critère d'arrêt est "avoir seulement 2 groupes")

3.5 Conclusion

Ce chapitre a défini précisément les différents types d'évolution logiciel et montré qu'il existe des recouvrements : la majorité d'entre eux commencent par une reconstruction de l'architecture. Si l'architecture cible est différente de l'existante, des techniques comparables à celles de reconstruction d'architecture peuvent être mises en œuvre pour identifier les éléments de code à allouer à la nouvelle architecture. Parmi toutes les techniques de reconstruction d'architecture, trois d'entre elles répondent aux exigences du projet de l'entreprise définies dans le chapitre 2. Ces trois techniques, le *Reflexion Model*, la recherche d'information et le regroupement décrites dans ce chapitre, présentent une adaptabilité potentielle au projet de l'entreprise. Afin de vérifier cette adaptabilité, le prochain chapitre fournit une description détaillée de la démarche réalisée par par les ingénieurs de Thales Air Systems sur ce projet.

CHAPITRE 4

Démarche de rénovation d'architecture par Thales Air Systems

Contents

Le chapitre précédent a identifié trois techniques de reconstruction d'architecture respectant les exigences du projet de l'entreprise. Ce chapitre se base sur ces trois techniques pour structurer la démarche réalisée par les ingénieurs sur ce projet afin de pouvoir la répliquer sur d'autres projets avec des ingénieurs différents. Nous décrivons l'architecture visée (Section 4.1) ainsi que la démarche d'évolution d'architecture *ad hoc* des ingénieurs conduisant à cette architecture (Section 4.2). Nous proposons par la suite une structuration de la démarche des ingénieurs qui se base sur l'application de la technique du *Reflexion Model* (présentée au chapitre précédent) à différents niveaux d'abstraction (Section 4.3). Finalement, la Section 4.4 conclut ce chapitre.

4.1 Description de la nouvelle architecture

Cette section décrit l'architecture visée par l'entreprise suivant la définition de l'architecture logicielle donnée par Perry et Wolf [Perry 1992]. Cette définition a été choisie, car elle est reconnue comme une définition fondamentale d'une architecture logicielle [Kruchten 1995, Malavolta 2013]. Perry et Wolf la définissent comme étant un ensemble de trois éléments :

les constituants de comportement correspondent au comportement du logiciel, *i.e.,* la valeur ajoutée du logiciel (*e.g.,* un *package* calculant une vitesse dans différents référentiels) ;

les constituants de données correspondent aux données utiles au comportement du logiciel, *i.e.,* les données qui seront utilisées et transformées par le logiciel (*e.g.,* une structure de données représentant une vitesse) ;

les connecteurs représentent les interactions entre les constituants ainsi que les règles qui régissent ces interactions (*e.g.,* un package définissant des envois et réceptions de messages).

Nous présentons l'architecture à composants visée par Thales Air Systems suivant ces trois éléments.

4.1.1 Description de la nouvelle architecture

Pour le projet qui nous intéresse, l'architecture visée par Thales Air Systems est une architecture à composants. Cette architecture à composants a été définie en amont du projet par les ingénieurs système et logiciel engagés sur le projet de l'entreprise, suivant leur connaissance du domaine.

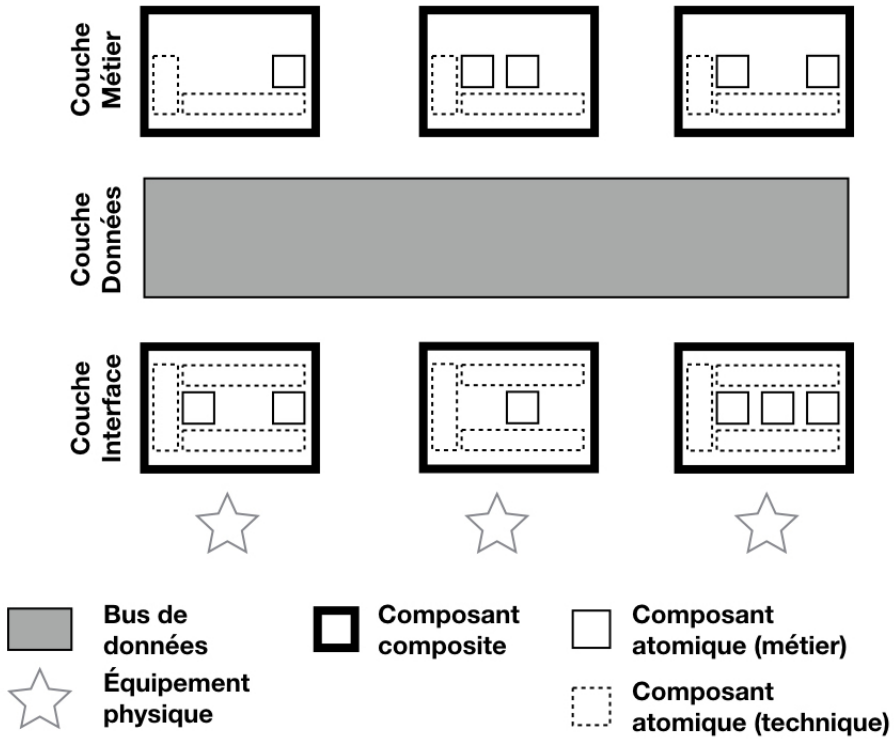


FIGURE 4.1: Nouvelle architecture à composants du projet de Thales Air Systems

La Figure 4.1 représente une forme simplifiée de cette architecture, par souci de lisibilité. La nouvelle architecture conçue par les ingénieurs de l'entreprise est séparée en trois couches :

— couche métier, qui contient les algorithmes et la valeur ajoutée du logiciel ;
— couche de données, qui contient les données internes du logiciel ;
— couche d'interface, qui contient les échanges entre le logiciel et les équipements externes ainsi que la traduction des données provenant de ces équipements vers le format de donnée interne au logiciel (et *vice-versa*).

La couche de données est composée des constituants de données du logiciel.

Les couches métiers et d'interface contiennent les constituants de comportement et les connecteurs de l'application. Les constituants de comportement et les connecteurs sont répartis sous la forme, respectivement, de composants atomiques métiers et techniques contenus dans 15 composants composites. Chaque composant composite compte de 1 à 6 composants atomiques métiers (constituants de comportement) et 2 ou 3 composants atomiques techniques (connecteurs).

Finalement, les constituants de comportement, de données et les connecteurs de cette architecture sont les suivants :

Constituants de comportement : Composants atomiques métiers de la couche d'interface et de la couche métier (petits carrés bords pleins dans la Figure 4.1) ;

Constituants de données : Toutes les données de la couche de données (rectangle gris au centre dans la Figure 4.1) ;

Connecteurs : Composants atomiques techniques de la couche d'interface et de la couche métier (rectangles pointillés noirs dans la Figure 4.1).

4.1.2 Identification des éléments architecturaux

Comme énoncé dans le chapitre 2, l'entreprise souhaite réutiliser le code source actuel du logiciel dans cette nouvelle architecture. Ainsi, les ingénieurs ont cherché à relier le code source actuel aux éléments de l'architecture visée.

Parmi les 15 composants composites de la nouvelle architecture, les ingénieurs ont déterminé que l'un de ces composants serait développé *from scratch* (de zéro). Cela signifie que ce composant ne réutilisera pas de code source actuel dans son implémentation. Ainsi, seuls 14 composants composites réutiliseront du code source existant.

D'autre part, les ingénieurs ont identifié que les composants atomiques techniques ne réutiliseront pas non plus de code source. En effet, ils considèrent que ces composants atomiques sont spécifiques à la nouvelle architecture et que le code source actuel n'est pas adapté à leur future implémentation. Sans le savoir, les ingénieurs ont suivi les préconisations de Perry et Wolf qui considèrent les connecteurs comme de la "glue" entre les autres éléments et qui sont, à ce titre, fortement dépendants de l'architecture. Seuls les composants atomiques métiers, *i.e.,* les constituants comportementaux, réutiliseront donc du code source existant.

En outre, les ingénieurs ont déterminé que le code de la couche de données sera généré à partir d'un outil de modélisation géré par l'entreprise.

L'architecture espérée qui est impliquée dans la réutilisation du code source se résume finalement à 14 composants composites et aux composants atomiques métier qui les constituent (entre 1 et 6 composants atomiques métiers par composant composite).

4.2 Démarche manuelle de rénovation d'une architecture logicielle

La démarche appliquée par Thales Air Systems pour son projet d'évolution d'architecture se base sur des activités de modification d'architecture et de réutilisation de code. La nouvelle architecture à composants a été définie dans la section précédente. Parmi les éléments de cette architecture, les ingénieurs ont identifié qu'ils pouvaient réutiliser le code source existant dans 14 composants composites (ainsi que dans les composants atomiques métiers qui les constituent). Cette section présente en détail la démarche qu'ils ont appliquée pour pouvoir réutiliser le code source dans ces composants (composites et atomiques).

4.2.1 Description de la démarche d'allocation

Afin de réutiliser le code source, les ingénieurs ont décidé d'allouer le code source actuel du logiciel aux composants de la nouvelle architecture. Telle que nous avons pu l'observer, la démarche d'allocation du code source réalisée a été séparée en deux phases :

— une première phase, à gros grains, basée sur l'allocation des *packages* Ada aux composants composites. Les ingénieurs étudient le code source des *packages* pour les allouer à un des 14 composants composites. L'allocation des *packages* se base sur des critères textuels (*e.g.,* les noms des *packages*) ou structurels (*e.g.,* les éléments qu'ils contiennent ou leurs relations).

— une deuxième phase, à grains plus fins, basée sur l'allocation des sous-programmes Ada aux composants atomiques métiers. Cette phase a pour but de raffiner les allocations faites lors de la phase précédente. Chaque composant composite est séparé en plusieurs composants atomiques métiers, auxquels sont alloués les sous-programmes des *packages* de ce composant composite. Pour le reste de ce document, "composant atomique" désignera un composant atomique métier.

  La figure 4.2 représente un exemple de raffinement du composant composite C en deux composants atomiques A1 et A2. Durant la première phase, les

packages P1 et P2 ont été alloués au composant C. Les sous-programmes (S1, S2, S3 et S4) de ces deux *packages* seront alors alloués aux composants atomiques A1 ou A2. Le choix d'allocation des sous-programmes dans les composants atomiques se base sur des critères textuels, structurels et sur les connaissances des ingénieurs dans le domaine du logiciel.

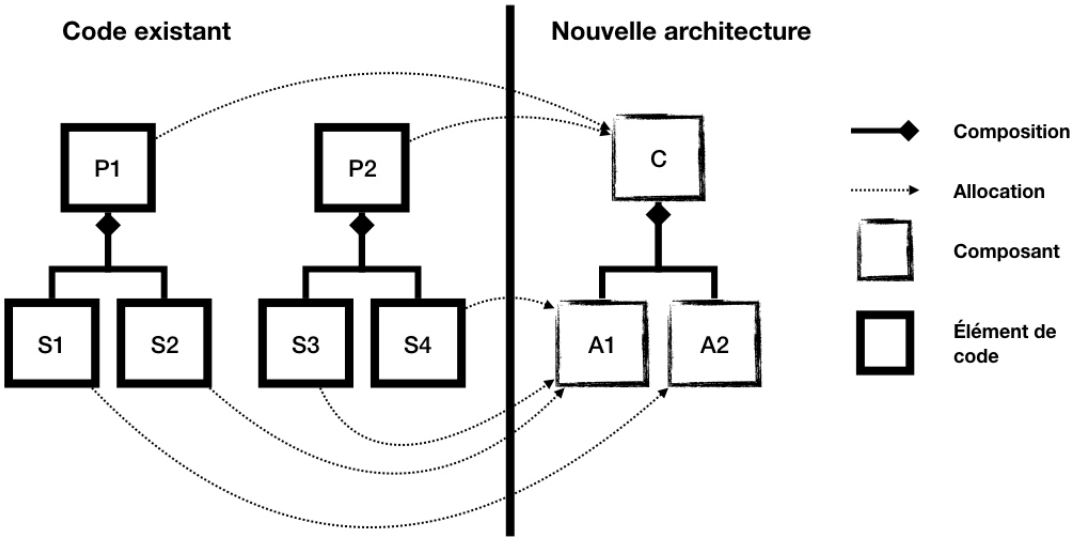


FIGURE 4.2: Exemple d'allocation de sous-programmes dans les composants atomiques pour le raffinement des composants composites

4.2.1.1 Phase 1 : Allocation des *packages*

La première phase de la démarche suivie a consisté à allouer les *packages* Ada aux composants composites de l'architecture. Lors de cette première phase, les ingénieurs ont sélectionné les *packages* Ada qui déclarent un élément du langage appelé "tâche" (*task*). En effet, chez Thales Air Systems, les tâches Ada sont utilisées pour représenter un processus réalisant une fonctionnalité métier. Ces fonctionnalités sont définies dans les documents de spécification du système logiciel. Les ingénieurs se sont alors basés sur les informations écrites dans ces documents et leur connaissance du domaine pour allouer, aux composants composites, les *packages* retrouvés de cette manière.

À partir de cette première allocation, les ingénieurs ont ensuite suivi les imports entre *packages* afin d'allouer les *packages* restants. Ils ont décidé que chaque *package* importé serait alloué au même composant composite que le *package* qui

l'importe. Ils ont implémenté un script qui étudie les clauses d'imports en début de fichier afin d'automatiser cette activité.

Problèmes rencontrés par les ingénieurs :

Les ingénieurs ont rencontré deux problèmes majeurs durant le suivi des imports :

1. Ils ont dû gérer les dépendances cycliques entre *packages* en empêchant leur script de passer deux fois par un même package.
2. Les ingénieurs ont identifié que leur script pouvait allouer un package à plusieurs composants composites. Chaque package ayant été alloué à plusieurs composants composites (comme le package P5 de la figure 4.3) a été traité individuellement et l'allocation finale laissée à la charge de l'ingénieur. Dans l'exemple de la Figure 4.3, le package P5 est indirectement importé par P1 et P6. Cependant, P1 et P6 sont alloués à deux composants composites différents. Il existe alors un conflit sur l'allocation du package P5, qui pourrait être alloué au composant composite 1 ou 2. Ce conflit est traité manuellement par les ingénieurs en se basant sur leur connaissance du domaine. Bien que la plupart des conflits soient traités manuellement, il apparaît que lorsqu'un conflit implique plus de 3 composites, ces *packages* sont finalement alloués à un composant de librairie. Ce nouveau composant de librairie ne fait pas partie des 15 composants composites identifiés par les ingénieurs dans la nouvelle architecture.

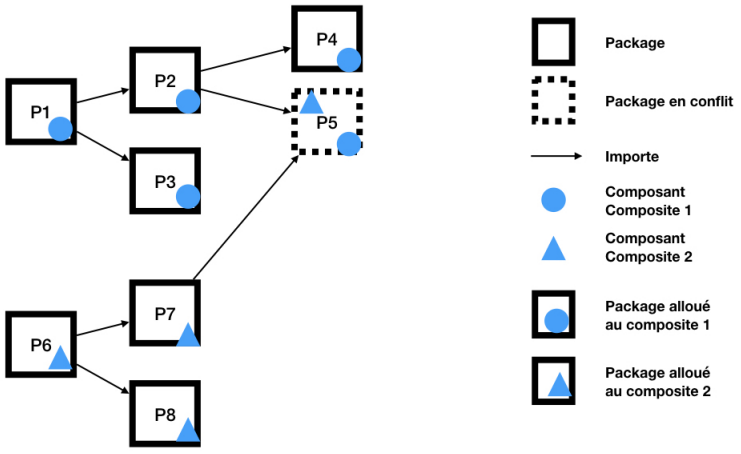


FIGURE 4.3: Exemple de conflits dus à l'allocation des *packages* importés

4.2.1.2 Phase 2 : Allocation des sous-programmes

La deuxième phase de la démarche réalisée par l'entreprise a consisté à allouer les sous-programmes aux composants atomiques. Cette phase de la démarche a eu pour but de raffiner l'allocation faite durant la première phase, excepté pour le composant de librairie pour lequel le besoin de raffinage n'a pas été retenu par les ingénieurs.

Pour chaque composant composite, les ingénieurs ont d'abord alloué certains sous-programmes suivant leur connaissance du logiciel ainsi que des conventions de nommage identifiées dans les documents de spécification du logiciel. Ils ont ensuite étudié les sous-programmes appelés par ceux déjà alloués. Les ingénieurs ont alors alloué les sous-programmes appelés en se basant sur la connaissance du domaine ainsi que sur les variables utilisées par ces sous-programmes. Cette démarche est similaire à l'utilisation des imports entre *packages* réalisée à la première phase.

Contrairement à la phase précédente, cette tâche a été réalisée manuellement et la multi allocation a été traitée différemment durant cette phase. Les ingénieurs ont indiqué qu'ils avaient laissé intentionnellement certains sous-programmes en conflits. Ils considéraient alors qu'un sous-programme alloué à plusieurs composants atomiques est un sous-programme qui réalise plusieurs fonctionnalités. De tels sous-programmes devraient alors être retravaillés lors du redéveloppement (pour les décomposer notamment).

Problèmes rencontrés par les ingénieurs :
Durant cette phase, les ingénieurs n'ont pas automatisé leurs activités et se sont basés principalement sur un travail manuel. Cela a pu entraîner de nombreuses erreurs humaines (*e.g.,* oubli de l'analyse de certaines méthodes) ainsi qu'une durée importante pour réaliser cette phase et pour corriger ces erreurs.

4.3 Structuration de la démarche de l'entreprise et *Reflexion Model*

Pour l'entreprise, un des premiers objectifs de cette thèse était de structurer la démarche décrite dans la section précédente. En effet, l'entreprise souhaiterait pouvoir capitaliser les connaissances gagnées lors de ce projet pour les appliquer sur d'autres projets similaires potentiels. Ayant repéré de fortes similitudes entre la démarche des ingénieurs et la technique du *Reflexion Model*, nous avons utilisé cette technique pour structurer la démarche décrite précédemment. Afin qu'elle soit appliquée sur d'autres projets d'évolution d'architecture potentiels au sein de

l'entreprise, nous avons identifié que notre démarche s'applique sur un projet impliquant au minimum :

— un logiciel dont le code source existant est constitué d'une hiérarchie d'éléments basée sur la contenance (*e.g., packages* et sous-programmes en Ada ou classes et méthodes en Java).

— une architecture visée également basée sur une hiérarchie de contenance (*e.g.,* composants composites contenants des composants atomiques).

Ces contraintes proviennent du fait que notre démarche a pour but d'allouer les éléments de code existants aux éléments de l'architecture visée en identifiant des correspondances entre eux. De plus, ces contraintes sont habituellement respectées par la plupart des systèmes existants et notamment tout ceux de Thales Air Systems.

4.3.1 Vue globale de notre démarche structurée

Le but de notre démarche est de mettre en correspondance les niveaux hiérarchiques des éléments de code avec ceux de l'architecture visée. Cette correspondance est réalisée en s'appuyant sur l'utilisation du *Reflexion Model*. En effet, la démarche que nous proposons, comme représentée en Figure 4.4, est découpée en un nombre fini de phases dans lesquelles nous appliquons un ou plusieurs *Reflexion Model*. Notre démarche pourrait être constituée d'une seule phase (et donc d'une seule application du *Reflexion Model*) dans le cas où l'architecture visée (respectivement le code source) ne comporte qu'un seul niveau hiérarchique. Dans le cas du projet que nous avons suivi, la démarche était constituée de deux phases.

La première phase est constituée d'une seule application du *Reflexion Model*, permettant d'allouer les éléments de code du premier niveau hiérarchique choisi aux éléments architecturaux du niveau hiérarchique correspondant. Pour la démarche des ingénieurs, les éléments de code du premier niveau sont les *packages* et les éléments architecturaux correspondants sont les composants composites.

Chaque phase supplémentaire est constituée de manière identique. On y applique autant de *Reflexion Model* qu'il y a d'éléments architecturaux dans la phase précédente afin de les raffiner. Lors de la deuxième phase de leur démarche, les ingénieurs ont raffiné les composants composites (éléments de premier niveau) en composants atomiques (élément de deuxième niveau).

Comme indiqué au chapitre 3, le *Reflexion Model* est une technique en cinq étapes : (rm_1) définition d'une architecture espérée (éléments et relations architecturales) ; (rm_2) extraction d'un modèle du code source ; (rm_3) mapping entre les éléments architecturaux et les éléments du modèle source ; (rm_4) vérification des contraintes définies à l'étape 1, à partir des données de l'étape 3 ; (rm_5) analyse des résultats obtenus à l'étape 4.

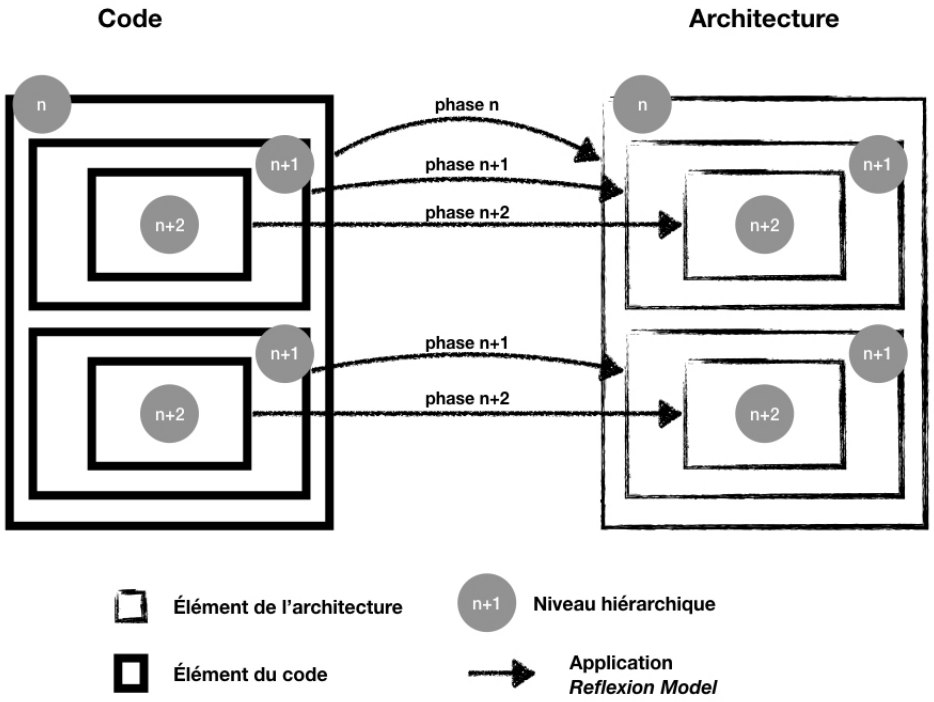


FIGURE 4.4: Vue globale de notre structuration de la démarche des ingénieurs (niveau n correspond au nième niveau de la hiérarchie de contenance ciblée)

4.3.2 Détail de notre démarche structurée : première phase

Dans la première phase de notre démarche, nous appliquons un seul *Reflexion Model* tel que suit :

1. Comme lors de l'étape (rm_1), les ingénieurs vont définir ici l'architecture qu'ils espèrent avoir à la fin de cette phase. Ils définissent alors les éléments du premier niveau hiérarchique de leur architecture visée. Contrairement à la démarche des ingénieurs, il est préférable de définir les relations attendues entre ces éléments architecturaux. La documentation des relations attendues permet à d'autres ingénieurs, avec moins de connaissances du domaine, de les vérifier par la suite.
   Dans le cas du projet de l'entreprise, les ingénieurs ont défini 14 composants composites (éléments architecturaux de premier niveau) et n'ont pas documenté les relations qu'ils attendaient. En effet, ils ont considéré que documenter ces relations n'était pas nécessaire, car ils réalisaient eux-mêmes leur vérification.

2. Contrairement à la deuxième étape du *Reflexion Model*, nous ne spécifions pas d'extraction particulière du code source. N'importe quel type d'informations et de techniques permettant de produire un modèle du code source peut être utilisé ici. Dans notre cas, nous nous appuyons sur la plateforme Moose [Moo] pour extraire un modèle du code source. Ce modèle doit toutefois extraire au moins un type d'éléments du code source (*e.g., package*) ainsi qu'au moins un type de relations entre ces éléments (*e.g.,* imports de *packages*)

3. L'étape (rm_3) du *Reflexion Model* est alors appliquée afin de réaliser un *mapping* qui alloue les éléments de code du niveau hiérarchique choisi aux éléments architecturaux définis lors de la première étape. Cette allocation est obtenue grâce à plusieurs critères d'allocation se basant sur les connaissances des ingénieurs, mais également des conventions de nommage et les dépendances entre les éléments de code. Ces critères d'allocation sont détaillés dans la Section 4.3.4, par la suite.

4. À partir de l'allocation faite à l'étape précédente, nous appliquons l'étape rm_4 du *Reflexion Model* en calculant les relations "induites" entre les éléments architecturaux. Une relation induite est une relation entre deux éléments de l'architecture visée qui abstrait les relations entre les éléments du code source alloués à ces deux éléments architecturaux. Un exemple de relation induite est représenté dans la Figure 4.5, où les éléments de code alloués sont des *packages* et les éléments architecturaux sont des composants composites. Dans cette figure, les composants C1 et C2 ont une relation induite due à la relation entre les deux *packages* P1 et P2 qui leur sont alloués. La

relation entre P2 et P3 n'apparaît pas au niveau des composants composites car ces deux *packages* sont associés au même composant composite (C2).

5. Finalement, la dernière étape du *Reflexion Model* est reproduite en comparant les relations induites calculées précédemment avec les relations attendues. Cette comparaison est alors réalisée automatiquement ce qui peut amener trois cas, soit une relation attendue est respectée, soit une relation attendue n'est pas respectée, soit une relation induite n'est pas attendue. Dans les deux derniers cas, l'action à entreprendre (*e.g.,* modification de l'allocation impactant ces relations) est laissée à la charge de l'ingénieur, suivant sa connaissance du domaine. Cela leur permet d'affiner les allocations réalisées durant la troisième étape.
   Dans le cadre de la démarche décrite à la Section 4.2, les ingénieurs ont réalisé la comparaison manuellement, car ils n'avaient pas documenté les relations attendues lors de la première étape. Ils se sont donc servis de leur connaissance du domaine et du logiciel pour réaliser cette comparaison.

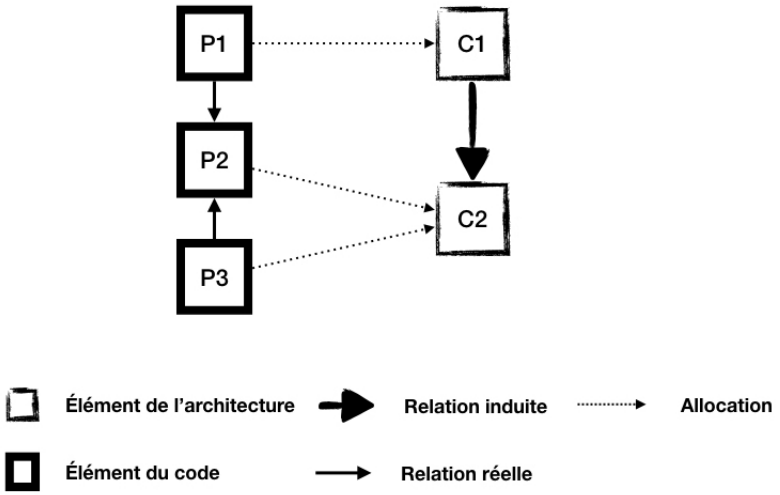


FIGURE 4.5: Exemple de relation induite entre composites (les éléments de code sont des *packages* et les éléments architecturaux sont des composants composites)

4.3.3 Détail de notre démarche structurée : phase supplémentaire

Chaque phase supplémentaire réalisée permet aux ingénieurs de raffiner les allocations réalisées lors de la phase précédente. Ce raffinement est obtenu en allouant les éléments de code d'un niveau hiérarchique inférieur avec les éléments

architecturaux d'un niveau hiérarchique inférieur. Nous proposons alors d'appliquer un *Reflexion Model* pour chaque élément architectural du niveau hiérarchique précédent. La figure 4.6 représente un exemple de raffinement obtenu grâce à la phase n+1, à la suite d'une phase n. Les éléments de code 1.1, 1.2, 2.3 et 2.4 sont alloués aux éléments architecturaux 1.1 et 1.2, car ces derniers composent l'élément architectural 1, auquel sont alloués les éléments de code 1 et 2.

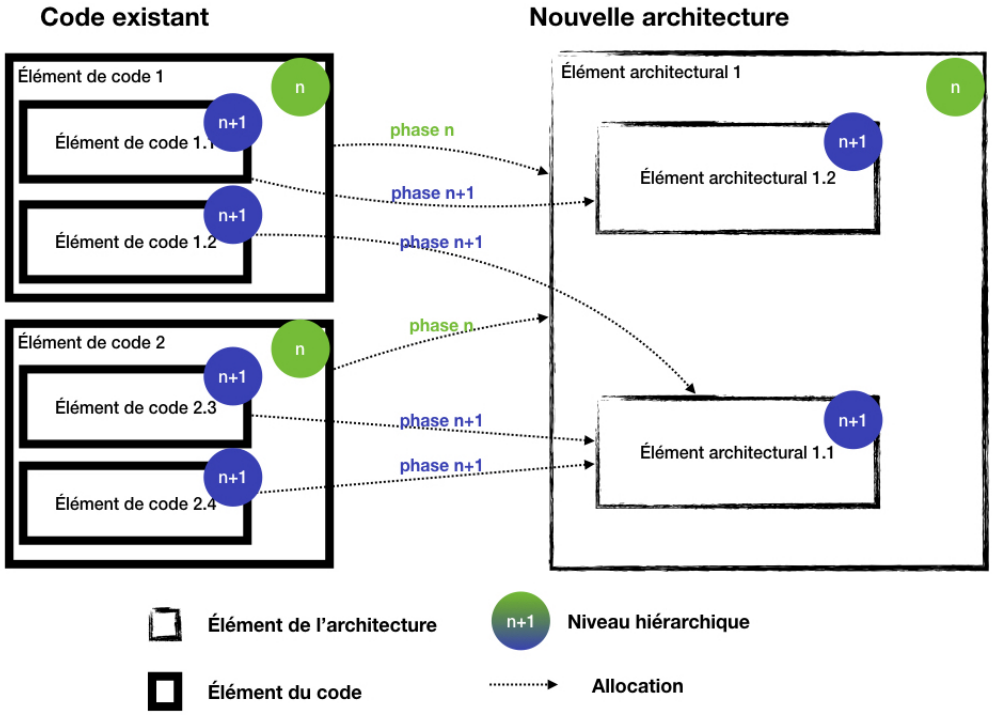


FIGURE 4.6: Exemple de raffinement d'allocation entre deux phases consécutives de notre démarche structurée

Les *Reflexion Model* appliqués lors de cette phase se déroule de la même manière, quel que soit l'élément architectural devant être raffiné. Cette application se base sur cinq étapes :

1. Les ingénieurs définissent les éléments architecturaux de niveau hiérarchique inférieur qui composent l'élément architectural de niveau supérieur pour lequel ils appliquent ce *Reflexion Model*. Il est préférable de définir les relations attendues entre ces éléments architecturaux inférieurs, en utilisant la connaissance des ingénieurs ou la documentation disponible, par exemple. Cette étape est similaire à la première étape de la phase précédente.
2. Un modèle du code source peut être extrait en utilisant n'importe quelles techniques disponibles. Dans notre cas, comme pour la première phase, nous

avons utilisé la plateforme Moose pour réaliser cette extraction.

3. Les ingénieurs réalisent un *mapping* en allouant les éléments de code aux éléments de l'architecture, comme requis dans l'étape (rm_3) du *Reflexion Model*. Nous définissons plusieurs critères permettant d'allouer les éléments de code et qui se basent sur les connaissances des ingénieurs ainsi que sur les conventions de nommage et les relations des éléments contenus. Ces critères d'allocation sont détaillés dans la Section 4.3.4, par la suite.
4. L'allocation réalisée à l'étape précédente sert de base pour calculer les relations induites entre les éléments architecturaux impliqués dans cette phase. Les relations induites sont définies de la même manière que dans la première phase.
5. Finalement, les relations induites sont comparées automatiquement avec les relations attendues ayant été définies dans la première étape. Cette comparaison peut amener trois cas : une relation attendue est respectée ; une relation attendue n'est pas respectée ; une relation induite n'est pas attendue. L'action à entreprendre lors des deux derniers cas est laissée à la charge des ingénieurs, suivant leur connaissance du domaine, ce qui leur permet d'affiner les allocations réalisées dans la phase courante.

4.3.4 Détail de notre démarche structurée : critères d'allocation

En nous basant sur la démarche effectuée par les ingénieurs, nous avons identifié que la troisième étape des phases de notre démarche structurée s'appuient sur plusieurs critères possibles d'allocation. Ces critères d'allocation visent à allouer les éléments de code aux éléments de l'architecture visée :

CA_c Critère d'allocation par contenance d'éléments.
Chaque élément de code contenant un certain type d'éléments est alloué automatiquement aux éléments de l'architecture visée. Les éléments architecturaux dans lesquels sont alloués ces éléments de code sont choisis par les ingénieurs suivant leur connaissance du domaine du logiciel.
Dans le cas du projet de Thales Air Systems, ce critère a été appliqué durant la première phase des ingénieurs afin d'allouer chaque *package* contenant une tâche Ada aux composants composites de l'architecture visée.

CA_o Critère d'allocation par l'organisation des fichiers sources.
L'organisation des fichiers sources en dossier peut servir de base à l'allocation des éléments de code à des éléments architecturaux représentant cette organisation. En effet, certains dossiers existant peuvent représenter un élément de l'architecture courante que l'on souhaite garder (*e.g.,* un dossier de librairie). Suivre ou non l'organisation des fichiers sources en dossier est laisser à la discrétion des ingénieurs.

Ce critère a été appliqué dans le cadre du projet de l'entreprise pour associer automatiquement chaque *package* dont le fichier source est contenu dans un dossier nommé "librairie" à un composant de "librairie", lors de la première phase de la démarche des ingénieurs.

CA_n Critère d'allocation par les noms.
Les ingénieurs utilisent des conventions de nommage pour allouer certains éléments de code aux éléments de l'architecture visée. Ces conventions et allocations se basent sur leur connaissance du domaine et du système.
Ce critère a été utilisé lors de la deuxième phase de la démarche des ingénieurs pour allouer des sous-programmes à des composants atomiques suivant des conventions de nommage propres au logiciel.

CA_d Critère d'allocation par les dépendances.
Les ingénieurs utilisent les dépendances entre les éléments de code pour les allouer aux éléments architecturaux. Ils propagent l'allocation d'un élément de code, que nous appelons "élément graine", à tous les éléments de code pouvant être atteints en suivant un type de dépendance donné. Cette propagation peut alors être réappliquée depuis les éléments voisins atteints précédemment [1].
La Figure 4.7 présente un exemple de la propagation de l'allocation d'un *package* Pkg 1 au composant composite C1 en suivant les imports. Ce critère a été utilisé par les ingénieurs dans les deux phases de leur démarche. Dans la première phase, ils ont propagé récursivement l'allocation des *packages*, en suivant les imports entre ceux-ci. Dans la seconde phase, ils ont propagé récursivement l'allocation des sous-programmes, en suivant les invocations entre ceux-ci.

CA_r Critère d'allocation par la résolution des conflits.
Un élément de code alloué à plusieurs éléments architecturaux différents est considéré comme conflictuel. En effet, les critère d'allocations précédent peuvent produire des cas de multi allocations d'éléments de code (*e.g.,* les conventions de nommage). Les conflits d'allocation pour les éléments de code peuvent alors être traités de trois manières différentes : (1) résolution automatique suivant une convention ; (2) résolution manuelle, au cas par cas, par les ingénieurs qui se servent alors de leur connaissance ; (3) non-résolution, ce qui signifie que cet élément devra être séparé dans la phase suivante du projet. Ce traitement peut être effectué dès l'apparition du conflit ou après avoir alloué tous les éléments de code.
Ce critère a été utilisé par les ingénieurs dans les deux phases de leur démarche. Lors de la première phase, les ingénieurs ont résolu les conflits automatiquement en supprimant la multi-allocation des *packages* conflictuels

1. il est nécessaire de tenir compte des éléments déjà atteints afin d'éviter les cycles

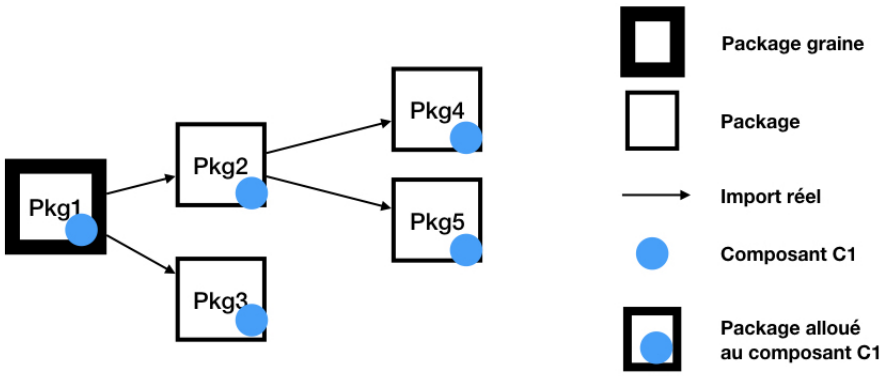


FIGURE 4.7: Exemple de propagation d'une allocation de *package* en utilisant les imports entre *packages*

et en les allouant au composite de librairie. Lors de la seconde phase, ils ont traité les conflits au cas par cas soit en décidant de l'allocation finale d'un sous-programme soit en ne résolvant pas le conflit. Lors du redéveloppement, les ingénieurs ont alors séparé le code source de ces sous-programmes encore en conflits dans plusieurs nouveaux éléments de code.

4.3.5 Discussion sur la démarche structurée

La démarche que nous proposons se base sur plusieurs applications du *Reflexion Model*, à différents niveaux hiérachiques et généralise la démarche des ingénieurs. En effet, nous retrouvons la démarche réalisée par les ingénieurs en appliquant notre démarche avec deux phases : (1) une première pour l'allocation des *packages* aux composants composites ; (2) une deuxième pour l'allocation des sous-programmes aux composants atomiques. De plus, la démarche que nous proposons ne modifie en rien les activités mises en place dans la démarche des ingénieurs et montre que le *Reflexion Model* peut être appliqué dans le cadre du projet de Thales Air Systems.

Cependant, bien que l'applicabilité du *Reflexion Model* au projet de l'entreprise soit montrée, cette démarche reste essentiellement manuelle. Or, l'entreprise souhaiterait pouvoir automatiser la démarche des ingénieurs afin qu'elle soit reproductible sur d'autres projets et par d'autres ingénieurs. De plus, la plupart des étapes reposent majoritairement sur la connaissance des ingénieurs, qui est une caractéristique subjective et non reproductible.

L'étape d'allocation des éléments de code présente un potentiel d'automatisation. En effet, cette étape s'appuie sur plusieurs critères d'allocation dont certains sont basés sur des informations objectives telles que des dépendances entre élé-

ments par exemple. Notamment, les critères CA_d et CA_r se servent des dépendances entre éléments de code ou d'un nombre d'allocations pour décider des allocations. Le prochain chapitre présentera les expériences que nous avons menées afin d'essayer d'automatiser cette étape grâce à :

— l'application de la recherche d'information ou du regroupement puisque ces deux techniques fournissent des *mapping* entre code et éléments architecturaux ;

— l'automatisation de la vérification des critères d'allocation CA_c, CA_o, CA_n, CA_d, CA_r.

D'autre part, nous ne garantissons pas l'exhaustivité des critères d'allocation identifiés pour chaque phase de notre démarche structurée. En effet, nous nous sommes basés uniquement sur notre compréhension et notre observation de la démarche des ingénieurs. Une telle démarche ne garantit pas de couvrir tous les critères d'allocation possibles dans le cadre d'un projet d'évolution d'architecture. Néanmoins, nous verrons au chapitre suivant que les critères, que nous avons identifiés, fournissent une base permettant l'automatisation de la démarche des ingénieurs.

4.4 Conclusion

Ce chapitre a décrit la nouvelle architecture à composants que Thales Air Systems vise pour son logiciel, ainsi que la démarche d'allocation du code source réalisée par l'entreprise pour arriver à cette architecture. Cette démarche est constituée d'un nombre fini de phases dépendant du niveau de détail souhaité dans l'architecture (deux dans le cas de l'entreprise). Ce chapitre a également structuré la démarche des ingénieurs en appliquant plusieurs *Reflexion Model* et en s'appuyant sur plusieurs critères d'allocations. De cette manière, nous avons montré que le *Reflexion Model* est une technique adaptable et applicable dans le cadre du projet de Thales Air Systems. Cette démarche structurée reste fortement manuelle, mais présente néanmoins un potentiel d'automatisation. Le prochain chapitre présente les expériences d'automatisation que nous avons réalisées via la recherche d'information et le regroupement (deux techniques identifiées dans le chapitre 3) ainsi qu'avec les critères d'allocation de notre démarche.

CHAPITRE 5

Expériences d'automatisation de la démarche de Thales Air Systems

Contents

Dans le chapitre précédent, nous avons structuré la démarche réalisée par les ingénieurs dans le cadre du projet de Thales Air Systems, en nous basant sur le *Reflexion Model*. Bien que notre démarche repose sur un ensemble de critères d'allocation des éléments de code aux éléments d'une architecture visée, elle n'en reste pas moins manuelle. De ce fait, nous souhaitons évaluer les possibilités d'automatiser cette démarche complètement ou en partie. Nous avons réalisé trois expériences d'automatisation dont nous avons évalué les résultats. Deux expériences se basent sur les techniques identifiées au chapitre 3 : la recherche d'information et le regroupement. La troisième utilise les critères d'allocation identifiés au chapitre précédent (Section 4.3.4) provenant directement de la démarche manuelle des ingénieurs.

La Section 5.1 présente le protocole d'expérimentation commun aux trois expériences. La Section 5.2 détaille l'adaptation des deux techniques de la littérature et de nos critères d'allocation au projet de l'entreprise. Nous décrivons et commentons les expériences que nous avons menées sur ces techniques ainsi que leurs résultats dans la Section 5.3. La Section 5.4 conclut ce chapitre en montrant que ni la recherche d'information ni le regroupement ne fournissent de résultats satisfaisants pour l'entreprise sur son projet, au contraire de l'automatisation de nos critères d'allocation.

5.1 Protocole d'expérimentation

Cette section décrit le protocole des expériences réalisées pour l'évaluation de la recherche d'information et du regroupement. Le but de cette évaluation est de

déterminer si ces techniques peuvent fournir des résultats similaires à ce qui a été réalisé par les ingénieurs sur le projet de l'entreprise, mais de façon automatisée.

Nous nous basons sur la démarche réalisée par les ingénieurs et décrite au chapitre précédent, pour réaliser nos expériences. Ainsi, nous réalisons deux expériences par technique, pour chaque phase de la démarche des ingénieurs : (1) pour l'allocation des *packages* aux composants composites ; (2) pour l'allocation des sous-programmes aux composants atomiques. Pour chaque expérience, les allocations réalisées par les ingénieurs sont considérées comme l'attendu, appelé "oracle".

La Section 5.1.1 décrit les oracles de chaque expérience. La Section 5.1.2 explicite la méthode de comparaison employée entre les oracles et les résultats fournis par les différentes techniques.

5.1.1 Définition de la base de comparaison

Oracle de l'allocation des *packages* aux composants composites : Le logiciel impliqué dans le projet de l'entreprise est constitué de $1\,474$ *packages*. Cependant, comme indiqué dans le chapitre précédent, les ingénieurs ont identifié qu'une partie de ces *packages* devaient être alloués à un composant de "librairie". Les ingénieurs ont explicité que tous les logiciels de l'entreprise suivent la même convention concernant les éléments composant la librairie, une convention qui se base sur l'organisation physique des fichiers de code source. Cette convention correspond au critère d'allocation CA_o que nous avons identifié au chapitre précédent. Nous avons alors automatisé ce critère d'allocation en utilisant le système de fichier pour retrouver le répertoire, nommé "librairie", contenant la librairie. Cela a permis d'allouer automatiquement les 609 *packages* respectant ce critère. Ce critère étant général à tous les logiciels de l'entreprise, nous excluons ces *packages* de l'oracle de la première phase pour nous concentrer sur les 865 *packages* restants. L'oracle de la première expérience est donc constitué de l'allocation de ces 865 *packages* aux 14 composants composites de l'architecture visée par l'entreprise. Les 14 composants composites seront appelés par la suite C1, C2, ..., C14.

Oracle de l'allocation des sous-programmes aux composants atomiques : Suivant la démarche structurée au chapitre précédent, la deuxième phase d'allocation sert à raffiner l'allocation faite à la première phase. Ainsi, l'allocation des sous-programmes aux composants atomiques doit se faire en tenant compte de l'allocation des *packages* aux composants composites. Pour ne pas introduire de bruit dans les expériences sur la deuxième phase, celles-ci se feront en se basant sur l'allocation des *packages* aux composants composites faites dans l'oracle de la première phase. En effet, nous considérons que l'oracle de la première phase est sûr et que les *packages* incluent dans cet oracle fourniront les bons sous-programmes. De

plus, il s'avère que les *packages* ayant été retirés de l'oracle pour la première phase (package de la librairie) possédaient également des sous-programmes, ceux-ci ont donc été retirés de l'oracle de la deuxième phase. De cette manière, l'oracle de la seconde phase est composé des allocations des sous-programmes faisant partie des *packages* non alloués à la librairie.

5.1.2 Méthode de comparaison pour les allocations

Nous évaluons chaque technique suivant la qualité des résultats. Cette qualité de résultats est calculée à partir de deux métriques statistiques : la précision et le rappel. Ces deux métriques se calculent de la manière suivante :

$$\text{précision} = \frac{\text{nb vrai positif}}{\text{nb vrai positif} + \text{nb faux positif}} \tag{5.1}$$

$$\text{rappel} = \frac{\text{nb vrai positif}}{\text{nb vrai positif} + \text{nb faux negatif}} \tag{5.2}$$

Avant de pouvoir calculer ces deux métriques, il est nécessaire de définir une méthode de comparaison entre les allocations résultantes de l'application des techniques et de l'oracle. La méthode de comparaison définie devrait alors permettre de calculer le nombre de vrai positif, de faux positif et de faux négatif.

D'après Anquetil et Lethbridge la comparaison entre deux allocations peut être réalisée de deux manières [Anquetil 1999] : (1) soit en comparant les allocations de chaque composant un à un ; (2) soit en comparant ce que les auteurs appellent des "intra-paires" et des "inter-paires", termes que nous définissons par la suite.

Comparaison des éléments alloués composant par composant Comparer les composants un à un requiert de savoir à quel composant de l'oracle correspond chaque composant obtenu par l'application d'une technique. La recherche d'information étant une approche *top-down* [Ducasse 2009], une telle identification entre les composants calculés et attendus est possible. Comme expliqué au chapitre 3, cette technique part des éléments d'une architecture visée pour retrouver les éléments de code qui leur sont associés. De cette manière, cette technique ne calcule pas les éléments architecturaux parmi les éléments de code, mais calcule les éléments de code correspondant à un élément architectural donné. Ainsi, les équivalents des éléments architecturaux de l'oracle peuvent être exactement identifiés parmi les éléments calculés par la recherche d'information.

A contrario, le regroupement est une technique *bottom-up* [Koschke 2000, Ducasse 2009]. Cette technique part des éléments de code pour calculer des regroupements d'éléments de code représentant les éléments de l'architecture sans lien direct avec sur les éléments architecturaux eux-mêmes. Donc, l'identification exacte

des composants de l'oracle qui correspondent aux groupes calculés par la technique n'est plus possible. En effet, comme représenté en figure 5.1, le regroupement ne permet pas de connaître exactement le nombre de groupes qui seront calculés ni à quels composants de l'oracle ils correspondent.

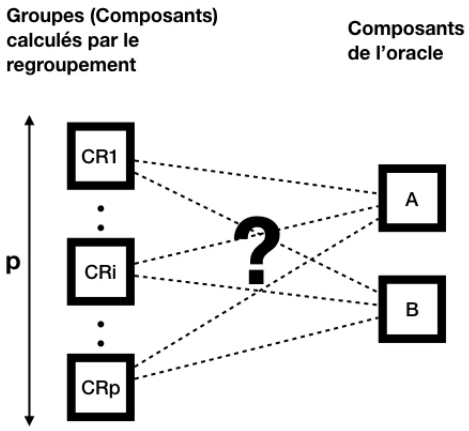


FIGURE 5.1: Comparaison de composants un par un entre ceux de l'oracle et ceux calculés par le regroupement

Afin d'établir une cohérence entre la qualité des résultats de chaque technique, nous souhaitons pouvoir employer la même méthode de comparaison pour évaluer la recherche d'information, le regroupement et la vérification de nos critères d'allocations. Ainsi, nous n'utiliserons pas la comparaison composant par composant dans le contexte d'évaluation de ces trois techniques.

Comparaison de paires d'éléments alloués La deuxième méthode de comparaison utilise les "intra-paires" et les "inter-paires" entre éléments de code alloués aux composants. Anquetil et Lethbridge définissent une intra-paire comme une paire d'éléments de code étant allouée à un même groupe (ou dans notre cas, à un même composant). Ils définissent ensuite une inter-paire comme une paire d'éléments de code étant allouée à deux groupes (ou dans notre cas, composants) différents. Le calcul des intra-paires et des inter-paires ne nécessite pas de savoir quel composant calculé correspond à quel composant de l'oracle. Nous avons donc choisi cette méthode de comparaison pour calculer la précision et le rappel de chaque technique.

En utilisant donc la deuxième méthode de comparaison, on définit les paramètres des équations 5.1 et 5.2 de la manière suivante :

Vrai positif = nb intra-paires calculées qui sont des intra-paires de l'oracle (l'intersection des deux cercles de la Figure 5.2)

Faux positif = nb intra-paires calculées qui ne sont pas des intra-paires de l'oracle (la partie non recouverte du cercle bleu de la Figure 5.2)

Faux négatif = nb inter-paires calculées qui sont des intra-paires de l'oracle (la partie non recouverte du cercle rouge de la Figure 5.2)

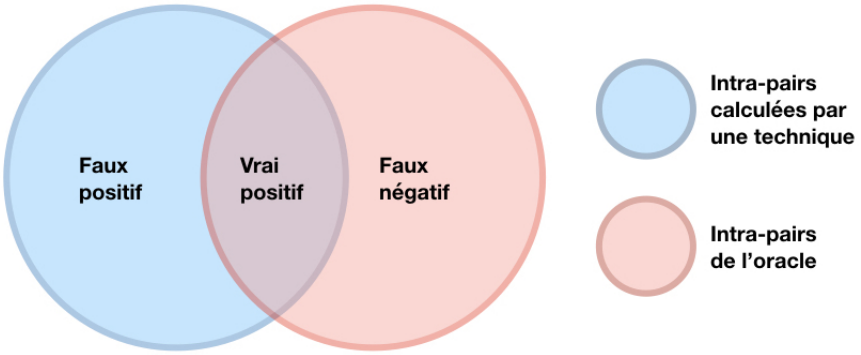


FIGURE 5.2: Représentation des vrais positifs, faux positifs et faux négatifs

De cette manière, la précision correspond au pourcentage d'intra-paires calculées qui sont aussi dans l'oracle. Le rappel correspond au pourcentage d'intra-paires de l'oracle qui sont effectivement retrouvées par les différentes techniques. Le calcul des inter-paires n'est donc pas nécessaire pour la précision et le rappel de chacune des techniques, ainsi seulement les intra-paires seront calculées.

$$precision = \frac{|intrapair_{technique} \cap intrapair_{oracle}|}{|intrapair_{technique}|} \tag{5.3}$$

$$rappel = \frac{|intrapair_{technique} \cap intrapair_{oracle}|}{|intrapair_{oracle}|} \tag{5.4}$$

5.1.3 Critères d'évaluation

L'évaluation que nous cherchons à réaliser pour chaque technique doit donner satisfaction à l'entreprise, en termes de précision et de rappel des résultats obtenus. Nous avons donc interrogé les ingénieurs ayant réalisé l'allocation du code source (1 architecte, 2 experts et 1 ingénieur) afin de déterminer les seuils de satisfaction nécessaire pour ces deux métriques. Chacun de ces ingénieurs a été interviewé séparément afin d'éviter qu'un ingénieur en influence un autre par ses réponses. En premier lieu, nous avons donné notre définition de la précision et du rappel à chaque ingénieur. Puis nous leur avons demandé quel serait le seuil de satisfaction qui leur conviendrait pour chacune de ces deux métriques, c'est-à-dire le pourcentage à

TABLE 5.1: Seuil d'acceptation des résultats de l'application d'une technique par les ingénieurs de l'entreprise

Ingénieurs	Précision	Rappel
Architecte	75%	50%
Expert A	75%	50%
Expert B	70%	70%
Ingénieur	80%	50%

partir duquel ils accepteraient les résultats de la technique (de 0 à 100% par tranche de 5%). Nous avons regroupé dans le tableau 5.1 les résultats de ces interviews.

Il apparaît que les ingénieurs ont des seuils d'acceptation du rappel des résultats d'une technique relativement similaire (3 ingénieurs à 50% et 1 ingénieur à 70%). Pour la précision, les réponses sont un peu plus disparates, mais reste néanmoins autour de 75%. Nous avons donc proposé aux ingénieurs que les seuils d'acceptation soient de 75% pour la précision et de 50% pour le rappel, seuils qu'ils ont acceptés. Ainsi, nous considérerons qu'une technique donne des résultats satisfaisants à une expérience dès lors que la précision des résultats est d'au moins 75% et le rappel d'au moins 50%.

5.2 Adaptation des techniques

Afin de pouvoir appliquer la recherche d'information, le regroupement et la vérification automatique de nos critères d'allocations, il est nécessaire d'adapter ces trois techniques au projet de Thales Air Systems. Chaque sous-section suivante détaille comment nous avons adapté chaque technique à ce projet.

5.2.1 Adaptation générale

Certaines adaptations ont été similaires entre la recherche d'information et la technique de regroupement choisie. Premièrement, nous avons dû adapter les techniques au langage de programmation (Ada 95) du logiciel impliqué dans le projet de l'entreprise. En effet, ces techniques n'ont pas été spécifiquement définies pour ce langage et nous avons dû inclure des éléments de langages particuliers (*e.g.,* tâche Ada) qui n'existent pas dans les autres langages (*e.g.,* Java). Les relations liées à ces nouveaux éléments de langage ont également dû être ajoutées aux techniques.

Pour rappel, les techniques ont dû être adaptées deux phases de la démarche des ingénieurs : (1) allocation des *packages* aux composants composites ; (2) allocation des sous-programmes aux composants atomiques. Nous avons alors décidé

d'évaluer chaque technique sur chacune des deux phases, donnant donc deux expériences par technique.

5.2.2 Adaptation de la recherche d'information

La recherche d'information, comme définie dans la section 3.4.2, est une technique découpée en six étapes : ri_1 définir un corpus de documents représentant le logiciel ; ri_2 extraire des documents les termes utiles en utilisant des conventions de nommage, en retirant les mots vides et en prenant le radical des termes des textes associés à chaque document ; ri_3 appliquer la méthode TF-IDF [Sparck Jones 1972] afin d'obtenir une matrice de fréquence des termes dans les documents ; ri_4 définir une requête composée d'un ensemble de mots ; ri_5 appliquer les requêtes ; ri_6 analyser les allocations faites à l'étape précédente.

Pour tenir compte des spécificités liées au projet de l'entreprise, ces étapes ont été adaptées de la manière suivante :

1. Définition du corpus : Comme énoncé plus haut, deux expériences ont été réalisées. Dans la première expérience, les documents sont des *packages* Ada. Dans la seconde expérience, les documents sont des sous-programmes Ada.

2. Extraction des termes utiles : Pour chaque document (*package* ou sous-programme), on extrait le code source et les commentaires de l'élément logiciel grâce à un parseur. Le texte résultant est ensuite découpé en termes qui sont eux-mêmes découpés suivant les conventions CamelCase et SnakeCase. Ces termes sont ensuite filtrés suivant une liste de mots vides de sens (*e.g.,* le, la, des). Ces mots vides sont soit les mots clés du langage Ada soit ceux de l'anglais (puisque les commentaires du logiciel sont écrits en anglais). Finalement, le radical de chacun des termes restants est extrait afin de normaliser ces termes (*e.g.,* pour éviter une différence entre "identification" et "identifications").

3. Calcul de la matrice de fréquence de termes : Le calcul de la matrice de fréquence des termes dans les documents a été réalisé par une librairie externe [Kuhn 2006].

4. Définition des requêtes : Les requêtes ont été définies pour chaque composant, créant donc une requête par composant. Ces requêtes sont obtenues à partir de courtes descriptions des composants, qui proviennent d'un document de spécification de conception écrit par l'entreprise. Le document étant en français et le code source en anglais, les mots extraits du document ont été traduits en anglais. Les requêtes traduites ont été soumises à vérification auprès des membres du projet de l'entreprise, qui les ont révisés et validés.

5. Application de la recherche d'information pour allouer les documents aux requêtes : Pour chaque requête, l'algorithme utilisé renvoie les documents rangés du plus similaire au moins similaire. Les documents (*i.e., packages* ou sous-programmes) ayant une similarité strictement supérieure à 0 avec la requête ont été alloués au composant associé à cette requête. Il est important de noter qu'un élément de code peut être alloué à plusieurs composants en même temps (*e.g.,* un élément dont la similarité est supérieure à 0 avec les requêtes de différents composants).
6. Analyse de l'allocation : Finalement, les résultats ont été analysés en comparant les allocations faites par l'adaptation de la recherche d'information, avec ce qui a été effectivement réalisé par les ingénieurs.

5.2.3 Adaptation du regroupement

La technique de regroupement proposé par Koschke dans sa thèse de doctorat [Koschke 2000] se découpe en cinq étapes (définies dans la Section 3.4.3) : rg_1 extraire les informations structurelles du code source du logiciel ; rg_2 définir et calculer une métrique de similarité entre tous les éléments logiciels et entre les groupes ; rg_3 définir un critère d'arrêt pour l'algorithme de regroupement ; rg_4 appliquer l'algorithme de regroupement ; rg_5 analyser les groupes résultants de l'algorithme.

Comme pour la recherche d'information précédemment, ces cinq étapes ont été adaptées au projet de l'entreprise :

1. Extraction des informations structurelles du code source : Les informations structurelles ont été extraites grâce à un parseur. Le résultat de cette phase de *parsing* est basé sur le meta-modèle Famix [Demeyer 2001] de la plateforme d'analyse logicielle Moose [Ducasse 2000]. Ce modèle contient la définition de tous les éléments logiciels (*e.g., packages*, tâches, sous-programmes, variables, types) et leurs relations (*e.g.,* invocations de sous-programmes, accès aux variables).
2. Définition et calcul de la métrique de similarité entre éléments logiciels : La métrique de similarité proposée par Koschke est basée sur les appels de fonctions C ainsi que sur les références aux variables et aux types [Koschke 2000, Chapter 7.6]. Cette métrique de similarité a été adaptée au langage Ada ainsi qu'aux spécificités de chacune de nos expériences. Pour la première expérience, la métrique se basera sur les imports entre *packages* et sur les variables utilisées dans les *packages*. Pour la seconde expérience, la métrique se basera sur les sous-programmes Ada et considérera toutes leurs relations, *i.e.,* invocations de sous-programme, utilisation de variables, références à des types utilisateurs. La métrique n'inclura donc pas les relations

liées aux pointeurs.

3. Définition d'un critère d'arrêt pour l'algorithme de regroupement : Le critère d'arrêt correspond au nombre de composants à retrouver ou lorsque toutes les similarités entre les groupes sont nulles. L'algorithme de regroupement sera donc stoppé après une itération si le nombre de groupes restant est égal au nombre de composants à retrouver ou si, après calcul des similarités entre chaque groupe, celles-ci sont toutes nulles. La deuxième partie du critère (similarités nulles) signifie que les groupes restants n'ont aucune des relations utilisées dans la métrique de l'étape 2 entre eux. Pour la première expérience, les composants sont les composants composites de l'architecture. Pour la seconde expérience, les composants sont les composants atomiques d'un composant composite de l'architecture.

4. Application de l'algorithme de regroupement : À cette étape, aucune adaptation n'a été réalisée. L'algorithme est tout simplement lancé jusqu'à atteindre le critère d'arrêt.

5. Analyse des groupes formés : Finalement, les groupes résultants de l'algorithme de regroupement ont été confrontés au groupe d'éléments logiciels réalisés par les ingénieurs.

5.2.4 Adaptation des critères d'allocation de notre démarche structurée

Lors de la structuration de la démarche des ingénieurs au chapitre précédent (Section 4.3.4), nous avons identifié plusieurs critères permettant d'allouer automatiquement des éléments de code à des éléments architecturaux. Puisque ces critères proviennent de l'étude de la démarche des ingénieurs, nous pensons qu'ils sont de bons candidats à l'automatisation de l'allocation d'éléments de code à des éléments architecturaux et qu'il est pertinent d'évaluer l'automatisation de leurs vérifications.

Comme pour les deux techniques précédentes, nous avons réalisé deux expériences : (1) allocation des *packages* aux composants composites ; (2) allocation des sous-programmes aux composants atomiques. Pour chacune de ces expériences, nous avons établi (en collaboration avec les ingénieurs) les critères d'allocation devant être vérifiés ainsi qu'un ordre de vérification. De plus, pour chaque expérience, nous avons expérimenté différentes adaptations des critères d'allocations vis-à-vis des *packages* et des sous-programmes. Cela nous a permis de déterminer la combinaison d'adaptations de critères donnant les résultats les plus élevés pour chaque expérience et si ces résultats dépassent ou non les seuils définis par les ingénieurs.

5.2.4.1 Adaptation de nos critères d'allocation dans la première expérience

Lors de la première expérience, les ingénieurs ont indiqué que suivant leur démarche, ils n'avaient pas utilisé de conventions de nommage pour allouer les *packages* aux composants composites. Ainsi, nous avons exclu le critère CA_n des critères d'allocations à vérifier pour cette expérience. Nous avons alors vérifié automatiquement les quatre critères d'allocation CA_o, CA_c, CA_d et CA_r dans l'ordre suivant :

(1) CA_o **:** Ce critère a été utilisé pour allouer les *packages* suivant les dossiers dans lesquels sont contenus leurs fichiers sources. Le code source que nous avions est réparti en deux dossiers : un dossier nommé "src-main" et un dossier nommé "Librairie". Les *packages* contenus dans le dossier "Librairie" ont alors été alloués au composant de "Librairie". Ce critère ayant été utilisé dans l'oracle, nous ne l'avons pas adapté. Les *packages* alloués grâce à ce critère ont donc été exclus de l'analyse suivante, comme indiqué à la Section 5.1.1.

(2) CA_c **:** Ce critère a été utilisé pour sélectionner des *packages* qui contiennent une tâche Ada. Ces *packages* sont ensuite alloués aux composants composites par les ingénieurs suivant leur connaissance du logiciel. Selon les ingénieurs, seules les tâches Ada permettent d'identifier les *packages* comme potentiel sujet d'une allocation. Ainsi, nous n'avons pas adapté ce critère d'allocation.

(3) CA_d **:** Ce critère permet d'allouer les éléments de code à des éléments architecturaux en utilisant les dépendances entre les éléments de code. Dans la première expérience, nous avons adapté ce critère pour allouer des *packages* à des composants composites selon les imports entre *packages*.
À partir d'un *package* dit "graine", tous les *packages* importés sont alloués au même composant composite que le *package* graine (et transitivement pour les autres *packages*). Nous nous sommes servis des *packages* alloués par le critère précédent (CA_c) comme graines de départ de la vérification du critère actuel.
Comme indiqué au chapitre précédent, les ingénieurs ont utilisé deux types d'imports, nous avons donc vérifié ce critère avec ces deux types d'imports :

- "imports déclarés", *i.e.,* les imports déclarés dans les clauses d'imports des *packages* (instruction Ada `with`). Il s'agit des imports que les ingénieurs ont utilisés dans leur script pour la première phase d'allocation Ces imports sont descendants, *i.e.,* du *package* qui importe vers le *package* importé ;
- "imports réels", *i.e.,* les imports déclarés qui sont réellement utilisés dans les *packages* (*e.g.,* en utilisant un sous-programme du *package*

importé). Ces imports sont descendants, *i.e.,* du *package* qui importe vers le *package* importé.

(4) CA_r **:** Le critère de résolution de conflits permet d'allouer automatiquement au composant de librairie, chaque *package* initialement alloué à plusieurs composants. Nous avons vérifié ce critère *a posteriori* de la vérification du critère précédent (CA_d). En effet, résoudre les conflits durant ou après la vérification du critère précédent ne modifie pas l'allocation obtenue, comme représenté en figure 5.3. Nous expérimentons alors deux solutions :

— "Avec Librairie", *i.e.,* les *packages* en conflit sont alloués à un composant de librairie (comme dans la démarche des ingénieurs) ;

— "Sans Librairie", *i.e.,* les *packages* sont laissés en conflit, les résolutions devront alors être faites manuellement par les ingénieurs.

Bien que ce critère alloue des *packages* à la librairie, nous n'avons pas exclu ces *packages* de l'analyse car il s'agit d'une manière différente d'allouer des *packages* à la librairie.

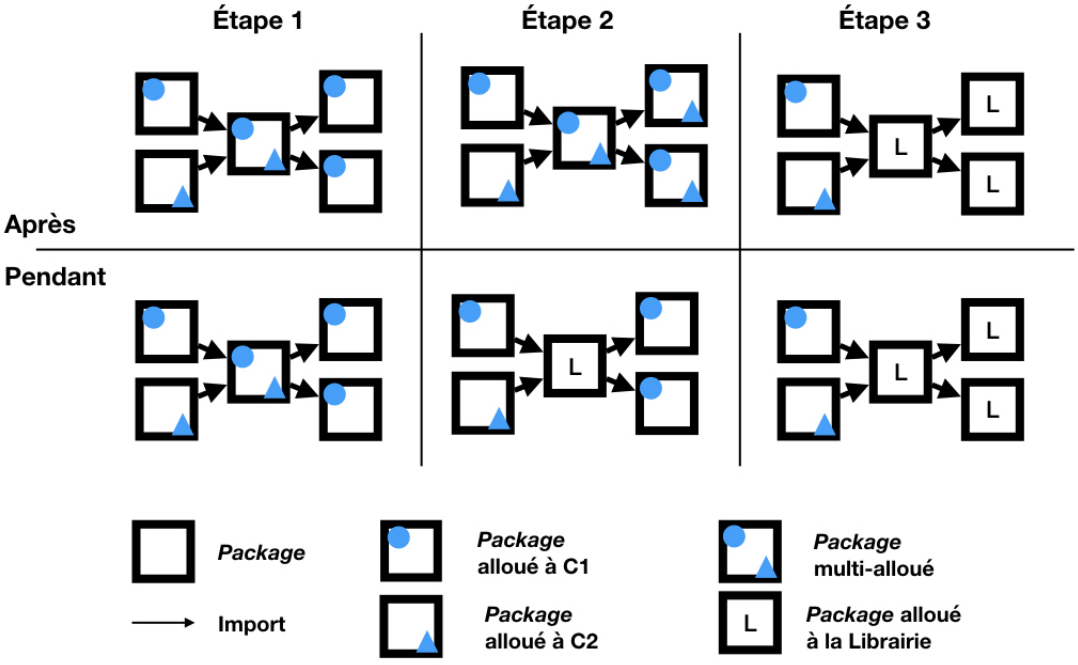


FIGURE 5.3: Vérification du critère CA_r : résolution de conflits par la librairie après (en haut) et pendant (en bas) vérification du critère CA_d

5.2.4.2 Adaptation de nos critères d'allocation dans la seconde expérience

Lors de la seconde expérience, les ingénieurs ont indiqué qu'aucun sous-programme n'a été alloué à un composant atomique en utilisant l'organisation physique du projet ou les éléments contenus dans les sous-programmes. Ainsi, nous avons exclu les critères CA_c et CA_o des critères d'allocations à vérifier pour cette expérience. Nous avons alors vérifié automatiquement les trois critères d'allocation CA_n, CA_d et CA_r dans l'ordre suivant :

(1) CA_n **:** La vérification de ce critère d'allocation a permis d'allouer, grâce à leurs noms, certains sous-programmes aux composants atomiques. Les conventions de nommage utilisées dans la vérification de ce critère proviennent de la connaissance des ingénieurs. Nous avons donc utilisé les conventions qui nous ont été données par les ingénieurs.

(2) CA_d **:** Ce critère permet d'allouer des éléments de code à des éléments architecturaux en utilisant les dépendances entre éléments de code. Nous avons adapté ce critère pour allouer les sous-programmes aux composants atomiques en utilisant les invocations entre sous-programmes. Les sous-programmes alloués par le critère précédent n'étant pas forcément des points d'entrée de comportement, nous avons expérimenté deux solutions :

- "Invocations descendantes", *i.e.,* les invocations allant des sous-programmes graines aux sous-programmes qu'ils appellent (et transitivement) ;
- "Invocations ascendantes et descendantes" allant des sous-programmes graines aux sous-programmes qu'ils appellent (et transitivement) et aussi des sous-programmes graines aux sous-programmes qui les appellent (et transitivement).

(3) CA_r **:** Nous avons adapté le critère de résolution des conflits pour allouer les sous-programmes en conflit aux composants atomiques. Notre adaptation se base sur les variables utilisées par les sous-programmes afin de résoudre un conflit.
Contrairement à la phase précédente, vérifier ce critère pendant ou après la vérification du critère CA_d donne des résultats différents. En effet, vérifier le critère CA_r *a posteriori* de la vérification de CA_d correspond à résoudre chaque conflit indépendamment les uns des autres. Ainsi, un sous-programme peut être alloué à un composant différent de celui auquel est alloué un sou-programme qui l'appelle (comme représenté dans la figure 5.4). Vérifier le critère CA_r durant la vérification du critère CA_d permet d'allouer les sous-programmes en tenant compte des résolutions précédentes (comme représenté dans la figure 5.5). Néanmoins, les résolutions finales obtenues avec cette façon de vérifier le critère CA_r ne sont pas dépendantes de l'ordre

dans lequels apparaissent les conflits car nous basons ces résolutions sur des critères indépendants de cet ordre (les variables utilisées par les sous-programmes). *A contrario*, dans la première expérience, quel que soit le nombre de composants auxquels un *package* est alloué, la résolution sera unique.
En outre, comme indiqué dans le chapitre précédent, les ingénieurs ont sciemment laissé certains sous-programmes en conflits. Nous avons alors expérimenté trois solutions :

— "Utilisation des variables (après)", un sous-programme conflictuel est alloué au composant atomique dans lequel les sous-programmes qui y sont alloués utilisent les mêmes variables que lui. Cette résolution de conflits est réalisée après avoir alloué tous les sous-programmes grâce au critère d'allocation CA_d.

— "Utilisation des variables (pendant)", un sous-programme conflictuel est alloué au composant atomique dans lequel les sous-programmes qui y sont alloués utilisent les mêmes variables que lui. Cette résolution de conflits est réalisée dès qu'un sous-programme devient conflictuel, *i.e.,* dès qu'il devient multi-alloué par le critère d'allocation CA_d.

— "Pas d'actions", *i.e.,* les conflits ne sont pas résolus et les sous-programmes sont laissés multi-alloués.

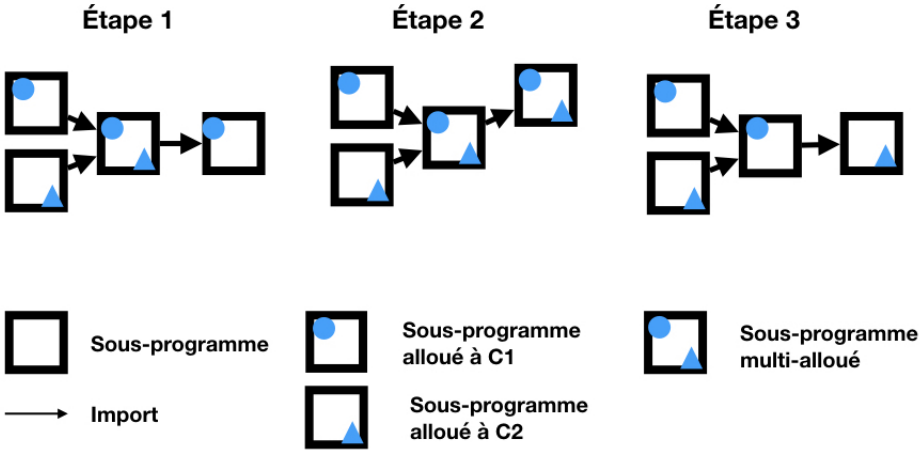


FIGURE 5.4: Vérification du critère CA_r : résolution de conflits (variables utilisées) après vérification du critère CA_d

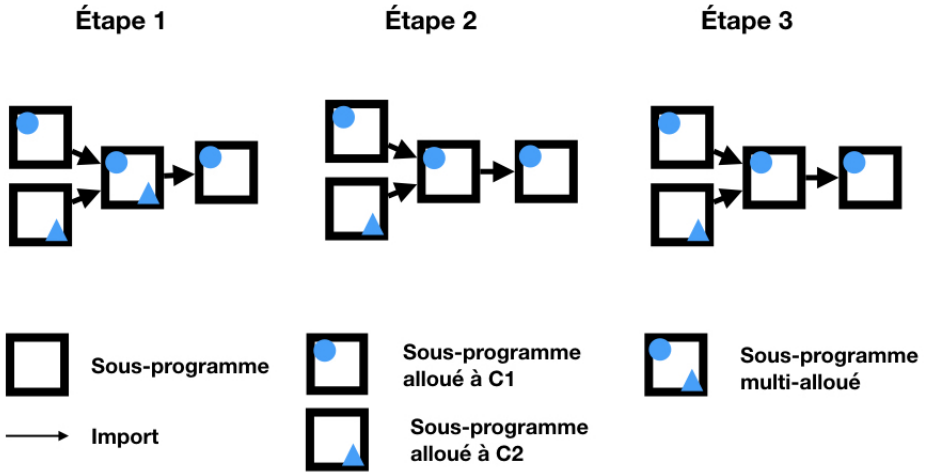


FIGURE 5.5: Vérification du critère CA_r : résolution de conflits (variables utilisées) durant la vérification du critère CA_d

5.3 Expériences avec les techniques

Cette section présente les expériences mises en place pour évaluer les techniques de recherche d'information, de regroupement et la vérification automatique de nos critères d'allocation sur le projet de Thales Air Systems. Ces expériences suivent le protocole défini à la Section 5.1. Chaque technique a fait l'objet de deux expériences : (1) évaluation de l'allocation des *packages* dans les composants composites ; (2) évaluation de l'allocation des sous-programmes dans les composants atomiques.

5.3.1 Expérience avec la recherche d'information

Le tableau 5.2 résume les résultats de l'application de la recherche d'information pour les deux phases d'allocation, au niveau des *packages* et au niveau des sous-programmes. La première ligne de ce tableau fournit le résultat de la première phase de cette allocation (*packages* alloués aux composants composites) et est détaillée dans la Section 5.3.1.1. Les lignes suivantes donnent le résultat de la seconde phase d'allocation en fonction de chaque composant composite et sont détaillées dans la Section 5.3.1.2.

TABLE 5.2: Résultats de l'application de la recherche d'information pour l'allocation des *packages* et l'allocation des sous-programmes (Sous-Prog.). Pour la partie basse du tableau, le contexte est chaque composant composite (C1 à C14). Le nombre entre parenthèses est le pourcentage d'éléments alloués aux composants (correctement ou non)

Contexte	Éléments alloués	#Éléments	#Allocation automatique	Précision	Rappel
Système	*Packages*	865	505 (58%)	9%	12%
C1	Sous-Prog.	265	214 (81%)	100%	81%
C2	Sous-Prog.	572	483 (84%)	5%	15%
C3	Sous-Prog.	45	4 (9%)	0%	0%
C4	Sous-Prog.	41	9 (22%)	100%	22%
C5	Sous-Prog.	734	213 (29%)	0%	2%
C6	Sous-Prog.	1291	1049 (81%)	0%	0%
C7	Sous-Prog.	21	4 (19%)	0%	0%
C8	Sous-Prog.	143	21 (15%)	0%	0%
C9	Sous-Prog.	209	126 (60%)	0%	0%
C10	Sous-Prog.	181	94 (52%)	3%	10%
C11	Sous-Prog.	227	187 (82%)	31%	54%
C12	Sous-Prog.	116	66 (57%)	100%	57%
C13	Sous-Prog.	553	529 (96%)	8%	36%
C14	Sous-Prog.	544	154 (28%)	0%	0%

5.3.1.1 Phase 1 : Allocation des *packages* dans les composants composites grâce à la recherche d'information

Seulement 58% des 865 *packages* ont été alloués automatiquement grâce à la technique de recherche d'information, comme le montre la première ligne du tableau 5.2. L'allocation des *packages* réalisée par la recherche d'information n'obtient que 9% de précision. Cela signifie que l'allocation n'est juste qu'à 9%, et que la recherche d'information alloue ensemble un nombre important de *packages* qui ne devraient pas l'être. Le rappel de la recherche d'information pour l'allocation des *packages* est également faible (12%) indiquant que moins d'un huitième des intra-paires voulues a été retrouvé par cette technique.

Ces résultats s'expliquent par le fait que les informations textuelles des *packages* couvrent des sujets qui seront traités par différents composants composites. Cela est également dû au fait que certaines conventions de nommages n'ont pas été respectées rigoureusement tout au long de la vie du logiciel. Ainsi, un nombre important de noms de variables ou de noms de méthodes sont aujourd'hui des acronymes du domaine (*e.g.,* DataSurveillanceEngagement deviendrait "DSE") ou des mots coupés (*e.g.,* Kinematic devient "kin" ou "kine"). Pour potentiellement améliorer les résultats, il serait nécessaire d'énumérer tous les acronymes possibles de chaque mot ainsi que toutes les découpes possibles de ces mots. Toutefois, d'une part, le lexique résultant de cette énumération serait fortement dépendant de l'application et devrait être adapté pour une autre application. D'autre part, la définition d'un tel lexique serait une activité coûteuse en termes humains et temporels. De plus, un lexique regroupant cette énumération pourrait aussi introduire du bruit dû à des abréviations similaires.

Une autre manière d'améliorer les résultats serait également de changer le seuil d'allocation des éléments. En effet, les éléments de code sont alloués aux composants pour lesquels la similarité avec la requête du composant est supérieure à un seuil donné, comme expliqué dans la Section 3.4.2. Dans la présentation de l'expérience (voir Section 5.2.2), ce seuil avait été fixé à 0. L'augmentation de ce seuil filtrerait plus de résultats et donc diminuerait le nombre de *packages* alloués, ce qui aurait deux conséquences :

1. diminution du rappel (ou au mieux le laisserait tel quel)
2. augmentation de la précision (ou au pire la laisserait telle quelle)

Ainsi, bien que la précision puisse être améliorée, cela se fera au détriment du rappel or, celui-ci (12%) est déjà très en dessous du seuil de satisfaction des ingénieurs (50%). Il sera donc inutile d'essayer d'appliquer cette amélioration.

La précision et le rappel des résultats de l'application de cette technique pour l'allocation des *packages* sont nettement inférieurs aux seuils (75% de précision et 50% de rappel) requis par les ingénieurs. Nous concluons donc que la recherche d'information, appliquée à l'allocation des *packages* aux composants composites,

ne fournit pas de résultats satisfaisants pour les ingénieurs et n'est pas retenue par l'entreprise pour automatiser son processus.

5.3.1.2 Phase 2 : Allocation des sous-programmes dans les composants atomiques grâce à la recherche d'information

Dans la seconde phase, la recherche d'information est appliquée afin d'allouer les sous-programmes aux composants atomiques. Chaque composant composite (appelés C1, C2 ..., C14) est découpé en plusieurs composants atomiques (de 1 à 6 par composite). Chaque composite ayant été peuplé par des *packages* durant la première phase de la démarche, les sous-programmes contenus dans les *packages* d'un composant composite seront alors alloués aux composants atomiques de ce composant composite. Par exemple, les *packages* alloués au composant composite C1 contiennent au total 265 sous-programmes. Ces sous-programmes devront donc être alloués uniquement aux composants atomiques de C1.

La deuxième partie du tableau 5.2 détaille les résultats de l'application de la recherche d'information pour allouer le code source à chaque composant composite. Il apparaît alors que la recherche d'information ne donne des résultats de rappel et de précision supérieurs à 0 que pour les composants C1, C2, C4 et les composants de C10 à C13.

Les résultats pour C1, C4, et C12 peuvent être considérés comme satisfaisants du point de vue de la précision pour les ingénieurs. Cependant en étudiant plus précisément C1, C4 et C12, il apparaît que ce sont des composites qui possèdent chacun un seul atomique dans l'oracle, ce qui signifie que, pour ces composants, la précision sera toujours de 100%. Les résultats de ces composants composites ne peuvent suffire à statuer sur la qualité des résultats de la recherche d'information pour l'allocation des sous-programmes, car il s'agit de cas particuliers.

C11 fournit des résultats dont le rappel dépasse le seuil défini par les ingénieurs (50%), mais avec une précision largement inférieure au seuil des ingénieurs (75%). Ceci est dû au fait que, dans l'oracle, le composant C11 est possède seulement deux composants atomiques. De cette manière, les allocations retrouvées ont une probabilité de 50% d'être correcte.

Excepté pour C1, C4, C11 et C12, les précisions de tous les autres composants composites sont inférieures à 10%. Le problème est que ces précisions sont valables pour des pourcentages de sous-programmes alloués automatiquement souvent élevés (plus de 50% comme indiqué dans la colonne *#Allocation automatique* du tableau 5.2). Cela signifie que les allocations obtenues par la recherche d'informations sont souvent fausses. En expliquant cela aux ingénieurs, ils ont indiqué qu'ils ne pouvaient pas se fier à de tels résultats et que cela les obligerait alors à réaliser la même étude manuelle que dans leur démarche.

Ces résultats sont explicables de la même manière que pour la première phase

(voir Section 5.3.1.1). La recherche d'information est une technique qui requiert que les informations textuelles utilisées soient fiables, *i.e.,* que les conventions soient toutes les mêmes et soient respectées, ce qui n'est pas le cas pour Thales Air Systems comme vu à la section précédente.

Finalement, la recherche d'information ne fournit pas de résultats atteignant les seuils d'acceptation donnés par les ingénieurs sur cette phase de la démarche structurée. Par conséquent, la recherche d'information ne peut pas être retenue par l'entreprise pour automatiser son processus d'évolution d'architecture.

5.3.2 Expérience avec le regroupement

Le tableau 5.3 résume les résultats de l'application du regroupement pour les deux phases de la démarche de l'entreprise que nous avons structurée, au niveau des *packages* et au niveau des sous-programmes. La première ligne du tableau correspond au résultat de l'allocation des *packages* dans les composants composites pour la technique de regroupement décrite au chapitre 3. Les lignes suivantes correspondent à l'allocation des sous-programmes dans les composants atomiques en utilisant le regroupement.

5.3.2.1 Phase 1 : Allocation des *packages* dans les composants composites grâce au regroupement

Dans cette première phase, la technique de regroupement choisie est appliquée afin de retrouver les composants composites de l'application. Suivant la description de l'adaptation de la technique de regroupement décrite en Section 5.2.3, au départ de l'algorithme, chaque package est considéré comme un groupe atomique. Le critère d'arrêt pour cette phase est de trouver 14 groupes (les 14 composants composites) ou d'avoir plus de groupes, mais avec des similarités nulles entre eux.

La première ligne du tableau 5.3 montre que l'algorithme de regroupement retrouve 80 groupes au lieu des 14 groupes attendus (soit 5 fois plus de groupes). Cela est dû au fait que l'algorithme a atteint un point où les similarités entre les 80 groupes sont nulles et ne peut donc pas trouver de nouveaux groupes parmi eux. Néanmoins, cet algorithme fournit des allocations précises à 25% et retrouve 63% des allocations de l'oracle, ce qui semble être un bien meilleur résultat que la recherche d'information (précision de 9% et rappel de 12 %).

Parmi les 80 groupes résultants de l'algorithme, nous avons identifié deux types de groupes :

— **groupes singletons**, *i.e.,* des groupes ne contenant qu'un seul *package* ;

— **groupe trou noir**, *i.e.,* un groupe contenant la plupart des *packages*. En l'occurrence, le groupe trou noir que nous avons identifié contient plus de 700 *packages*.

TABLE 5.3: Résultats de l'application du regroupement pour l'allocation des *packages* et l'allocation des sous-programmes (Sous-Prog.). Pour la partie basse du tableau, le contexte est chaque composant composite (C1 à C14). Le nombre entre parenthèses indique de combien de fois le nombre de groupes calculés dépasse le nombre de groupes attendus.

Contexte	Éléments alloués	#Éléments	#Groupes attendus	#Groupe calculés	Précision	Rappel
Système	*Packages*	865	14	80 (5×)	25%	63%
C1	Sous-Prog.	265	1	27 (27×)	100%	41%
C2	Sous-Prog.	572	8	100 (12×)	18%	8%
C3	Sous-Prog.	45	4	41 (10×)	0%	0%
C4	Sous-Prog.	41	1	15 (15×)	100%	27%
C5	Sous-Prog.	734	4	137 (34×)	0%	42%
C6	Sous-Prog.	1291	6	53 (9×)	0%	0%
C7	Sous-Prog.	21	2	14 (7×)	0%	0%
C8	Sous-Prog.	143	3	21 (7×)	0%	1%
C9	Sous-Prog.	209	3	107 (35×)	0%	0%
C10	Sous-Prog.	181	6	30 (5×)	4%	40%
C11	Sous-Prog.	227	2	30 (15×)	24%	48%
C12	Sous-Prog.	116	1	9 (9×)	100%	45%
C13	Sous-Prog.	553	6	64 (10×)	18%	32%
C14	Sous-Prog.	544	3	31 (10×)	0%	18%

Les groupes dits "singletons" n'impactent en rien les métriques que nous utilisons, car ils ne forment pas d'intra-pairs.

Au contraire, le groupe dit "trou noir" impacte fortement la précision et le rappel. En effet, à travers ce groupe, un nombre important d'intra-paires sont créées ce qui augmente la possibilité qu'une intra-paire attendue soit retrouvée (fort rappel), mais augmente aussi fortement la possibilité qu'une intra-paire soit fausse (faible précision).

Finalement, bien que le regroupement fournisse des résultats numériquement meilleurs à la recherche d'information, ils restent inférieurs aux seuils définis par les ingénieurs. D'autre part, les résultats obtenus par le regroupement cachent certaines failles (groupes "singletons" et "trous noirs") qui impliqueraient un travail d'analyse important pour les ingénieurs. De cette manière, le regroupement ne peut pas être choisi comme technique pour l'automatisation de l'allocation des *packages* dans les composants composites.

5.3.2.2 Phase 2 : Allocation des sous-programmes dans les composants atomiques grâce au regroupement

La seconde phase de l'expérience avec le regroupement concerne l'allocation des sous-programmes dans les composants atomiques. Comme pour la recherche d'information en Section 5.3.1.2, cette phase vise à raffiner les composants composites. La deuxième partie du tableau détaille les résultats du regroupement pour allouer les sous-programmes aux composants atomiques de chaque composant composite.

Les résultats du regroupement sur les composants C1, C4 et C12 suivent la même règle que ceux de la recherche d'information (1 seul composant atomique) avec une précision de 100% pour ces 3 composants (mais avec un rappel plus faible pour le regroupement sur C1 et C12). Comme expliqué pour la recherche d'information, les résultats de l'application d'une technique sur ces composants ne sont pas suffisants pour statuer de son utilisation ou non pour l'allocation des sous-programmes dans les composants atomiques.

Certains autres composants semblent sortir du lot avec des précisions entre 18% et 24% et/ou des rappels entre 8% et 48%. Toutefois, ces précisions et rappels sont toujours trop faibles pour être acceptables pour l'entreprise. En effet, en couplant ces résultats avec le nombre de groupes calculés, il apparaît que le regroupement ne fournit jamais le bon nombre de groupes et que les groupes formés ont une précision et un rappel faibles. Comme pour l'allocation des *packages*, l'algorithme de regroupement s'arrête avant d'avoir réduit le nombre de groupes à celui attendu, car les similarités sont toutes nulles.

En résumé, l'application du regroupement sur chacun des composants composites ne fournit pas de résultat satisfaisant avec une précision ou un rappel supé-

rieur à 50%. De plus, l'algorithme de regroupement ne permet pas de retrouver exactement le nombre de groupes attendus. Ainsi, le regroupement ne peut pas être considéré comme une technique viable par l'entreprise.

5.3.3 Expériences d'automatisation des critères d'allocation de notre démarche structurée

Nous présentons ici les résultats de nos expérimentations sur l'automatisation de la vérification des critères d'allocation que nous avons identifiés dans notre démarche structurée. Chaque section suivante présente les résultats de la vérification automatique des différentes combinaisons de nos critères d'allocation pour chaque expérience .

5.3.3.1 Phase 1 : Allocation des *packages* dans les composants composites grâce au regroupement

Comme indiqué dans la section 5.2.4.1, nous avons successivement vérifié les critères CA_o, CA_c, CA_d et CA_r dans notre première expérience. Le critère CA_o ayant été utilisé dans l'oracle, nous ne l'avons pas fait varier. De plus, selon les ingénieurs, le critère CA_c ne présente pas de variations possibles, nous ne l'avons donc pas fait varier.
Les résultats de l'application des critères CA_o et CA_c ainsi que des différentes combinaisons des critères CA_d et CA_r sont représentés dans le tableau 5.4. Chaque ligne de ce tableau représente une combinaison différente d'adaptations pour les critères CA_d et CA_r.

TABLE 5.4: Automatisation de la vérification des critères d'allocations de la première phase (asc=ascendants/desc=descendants)

Critère d'allocation			
CA_r	CA_d	Précision	Rappel
Avec librairie	Imports déclarés	68%	93%
	Imports réels	94%	89%
Sans librairie	Imports déclarés	6%	32%
	Imports réels	9%	31%

Les deux premières lignes du tableau montrent que la vérification du critère CA_r donne des résultats numériquement meilleurs en utilisant la solution "Avec librairie" . Cette solution propose de résoudre les *packages* en conflits en les assignant automatiquement au composant de librairie. Lorsque ces conflits ne sont

pas résolus (troisième et quatrième ligne du tableau), le rappel et la précision diminuent fortement. Ils semblent donc que le critère d'allocation CA_o ne permet pas de retrouver tous les *packages* de la librairie de la nouvelle architecture.

D'autre part, en comparant les résultats obtenus pour les différentes adaptations du critère CA_d pour une même adaptation du critère CA_r (*e.g.,* comparer la première et la deuxième ligne), nous voyons que la solution des "imports réels" fournit des résultats numériquement meilleurs. Cela est dû au fait que certains *packages* sont importés (dans les imports déclarés), mais ne sont jamais utilisés et que ces *packages* sont alors analysés par les ingénieurs et réalloués correctement par la suite. Les résultats de cette expérience confirment alors la faiblesse que nous avions identifiée au chapitre précédent dans le script que les ingénieurs avaient développé pour suivre les imports entre *packages*. Selon les résultats de la deuxième et de la troisième ligne, utiliser les imports réels au lieu des imports déclarés augmente la précision de 26% tout en diminuant légèrement le rappel de 4%, lorsque l'on résout les conflits. En effet, les imports réels correspondent à un sous-ensemble des imports déclarés ce qui explique que le rappel puisse diminuer et la précision augmenter en utilisant les imports réels. En outre, les résultats de ces deux métriques pour la deuxième ligne du tableau dépassent les seuils établis par les ingénieurs pour le rappel (50%) et la précision (75%).

Finalement, la solution utilisant les imports réels (pour CA_d) et la résolution de conflits vers la librairie(pour CA_r) est la seule solution pour laquelle les deux résultats (précision et rappel) dépassent les seuils requis par les ingénieurs. Cette solution pourrait donc être appliquée pour automatiser l'allocation des *packages* aux composants composites.

5.3.3.2 Deuxième phase : automatisation des critères d'allocation

Le tableau 5.4 présente les résultats de nos expérimentations sur la vérification des critères d'allocation de la deuxième phase d'allocation pour le composite C13. En effet, comme indiqué dans la section 5.2.4.2, nous n'avons pu vérifier nos critères que sur un seul composant composite, le composant C13.

Nous avons alors successivement vérifié les critères CA_n, CA_d et CA_r pour le composant composite C13 dans notre seconde expérience. Chaque ligne de ce tableau correspond à une combinaison des différentes solutions d'adaptations pour les critères CA_d et CA_r.

À la lecture du tableau 5.5, il apparaît que le rappel est meilleur lorsque les conflits ne sont pas résolus (première et deuxième ligne). Cela s'explique par le fait que les conflits non résolus laissent les sous-programmes alloués à plusieurs composants, il y a donc plus de chances que les allocations soient correctes. Cependant, cela augmente également le nombre de faux positifs (*i.e.,* mauvaise allocation) et donc fait diminuer la précision.

TABLE 5.5: Automatisation de la vérification des critères d'allocations de la seconde phase (asc=ascendants/desc=descendants) pour le composant composite C13

Critère d'allocation			
CA_r	CA_d	Précision	Rappel
Pas d'action	Invocations desc	57%	51%
	Invocations asc/desc	37%	65%
Action : Utilisation des variables (pendant)	Invocations desc	74%	48%
	Invocations asc/desc	53%	62%
Action : Utilisation des variables (après)	Invocations desc	60%	49%
	Invocations asc/desc	53%	58%

D'autre part, à solutions de résolutions équivalentes, utiliser uniquement les invocations descendantes (ligne 1, 3 et 5) fournit une meilleure précision alors qu'utiliser les invocations descendantes et ascendantes fournit un meilleur rappel (ligne 2, 4 et 6). Ceci s'explique de la même manière que pour l'utilisation des imports de *packages*. Suivre les relations ascendantes et descendantes fournit potentiellement plus de vrais positifs (bonne allocation) que simplement suivre les relations descendantes. Cependant, cela se fait au détriment de la précision, car il y aura potentiellement plus de faux positifs (mauvaise allocation).

De plus, résoudre les conflits en se servant des variables utilisées diminue le rappel, mais augmente la précision. En effet, cette résolution de conflit se base sur ce que les ingénieurs ont eux-mêmes réalisé (voir Section 4.2.1.2 au chapitre précédent). Il semble donc logique que la précision augmente, car nous suivons le même critère que les ingénieurs. Néanmoins, le rappel diminue quelque peu en utilisant cette solution de résolution de conflits, ce qui indique qu'il existe des exceptions à la résolution des conflits suivant l'utilisation des variables.

Finalement, la solution utilisant les invocations descendantes (pour CA_d) et une résolution courante des conflits par l'utilisation des variables (pour CA_r) est la solution qui obtient les résultats les plus élevés (précision de 74% et un rappel de 48%). Cette solution s'approche fortement des seuils requis par les ingénieurs (précision = 75% et rappel = 50%) et pourrait être acceptée par ceux-ci.

5.4 Conclusion

Les expériences avec la recherche d'information et le regroupement ont montré que les résultats de l'application de ces deux techniques sur le projet de l'entreprise ne satisfont pas les seuils définis par les ingénieurs (précision = 75% et rappel = 50%). Cela peut s'expliquer de deux manières :

— Premièrement, les conventions de codage strictes (*e.g.,* conventions de nommage ou structurelles) adoptées pour ce logiciel ont été affaiblies par les différentes évolutions et un développement partagé en plusieurs équipes. De cette manière, les conventions ne peuvent plus être considérées comme un critère de sélection fiable pour distinguer les éléments du code source.

— Deuxièmement, les techniques existantes décrites dans le Chapitre 3 partagent toutes un point commun, elles sont basées sur un seul critère de sélection et de rassemblement. Par exemple, la recherche d'information repose sur le lexique utilisé dans le logiciel et le regroupement sur les dépendances structurelles. Or, il apparaît que pendant les processus de définition d'architecture et de développement, les ingénieurs utilisent plusieurs types de conventions (*e.g.,* conventions de nommage et sur les dépendances) pour représenter leur architecture.

Ainsi, les techniques existantes de reconstruction d'architecture fournissent, de manière indépendante, des résultats de bonne qualité dès lors qu'un seul type de convention suffit pour décrire l'architecture elle-même. Pour la recherche d'information, un exemple serait de décrire l'architecture en préfixant les noms des éléments du code source avec l'élément architectural qu'il représente, *e.g.,* une classe nommée "ComposantKinematic".

La qualité des résultats obtenus par l'application des techniques existantes est dépendante du respect des conventions choisies durant le développement (*e.g.,* convention de nommage). Toutefois, la recherche a montré que ces conventions sont rarement respectées strictement et notamment dans le cas de convention de nommage *e.g.,* [Furnas 1987], *"Malheureusement, les gens sont souvent en désaccord sur les mots qu'ils utilisent pour les choses"* [1]. De plus, il peut s'avérer difficile de maintenir les conventions choisies initialement sur des projets réels de l'envergure de celui de Thales Air Systems. Une automatisation complète de la démarche proposée par l'entreprise ne pourra se faire en utilisant seulement une technique existante.

Ce constat est renforcé par les résultats d'expériences obtenus en vérifiant automatiquement les critères d'allocation lors de la première phase de notre démarche structurée. La deuxième ligne du tableau 5.4 montre que les résultats obtenus (précision = 94% et rappel = 89%) sont largement supérieurs aux seuils de satisfaction donnés par les ingénieurs (précision = 75% et rappel = 50%). Ainsi, nous pourrions automatiser la première de phase la démarche des ingénieurs en utilisant la vérification automatique des critères d'allocation :

— CA_d = "imports réels"

— CA_r = "Avec Librairie"

1. "Unfortunately, people often disagree on the words they use for things"

.

D'autre part, les meilleurs résultats obtenus (précision = 74% et rappel = 48%) pour la vérification automatique des critères d'allocation de la deuxième phase sont proches des seuils de satisfaction des ingénieurs (précision = 75% et rappel = 50%). Bien qu'ils ne dépassent pas ces seuils, les ingénieurs ont considéré que les résultats obtenus sur cette expérience étaient prometteurs et nécessitaient un approfondissement. En effet, il semblerait que la granularité de ces critères ne soit pas suffisante pour capter toutes les activités des ingénieurs. Ainsi, une analyse plus détaillée de la démarche permettra d'extraire les activités qu'ils réalisent le plus fréquemment afin de pouvoir les automatiser.

Le prochain chapitre présente un ensemble d'activités que nous avons identifiées dans la démarche des ingénieurs ainsi qu'un ensemble d'opérateurs que nous proposons pour les automatiser.

CHAPITRE 6

Opérateurs d'aide à la réplication et l'automatisation d'une démarche de rénovation d'architecture

Contents

Le chapitre précédent a présenté les expériences que nous avons menées afin d'automatiser notre démarche structurée définie au chapitre 4. À travers ces expériences, nous avons montré que seule l'automatisation de la vérification de nos critères obtient des résultats pouvant dépasser les seuils de satisfaction définis par les ingénieurs de l'entreprise.

Ce chapitre propose d'étudier plus finement les tâches réalisées par les ingénieurs lors de différents projets de rénovation d'architecture. Cette étude a mis en évidence des activités génériques qui visent la compréhension, l'analyse ou la réutilisation du code source. Nous proposons des opérateurs pour faciliter l'automatisation et la réplication de ces activités.

La Section 6.1 présente les activités mises en place dans les différents projets de rénovation d'architecture par les ingénieurs que nous avons pu interroger. La Section 6.2 présente ensuite les opérateurs que nous proposons pour généraliser et structurer les activités des ingénieurs. La Section 6.3 montre la complétude de nos opérateurs vis-à-vis des activités identifiées plus tôt et montre un exemple d'application sur le projet décrit dans cette thèse. Enfin, nous concluons ce chapitre dans la Section 6.4.

6.1 Activités d'ingénierie pour la rénovation d'architecture d'un logiciel

Le chapitre précédent a montré que l'utilisation d'une seule technique sur la totalité de l'application ne fournit pas de résultats satisfaisants. À l'inverse, la vérification automatique de nos critères d'allocation donne des résultats très prometteurs. Ainsi, nous allons chercher à décomposer au maximum notre démarche afin d'en automatiser le plus de parties possible. Pour cela, nous avons étudié le projet décrit dans cette thèse ainsi que de nouveaux projets au sein de l'entreprise afin d'y identifier ce que nous appelons des "activités".

Une "activité" est une tâche réalisée (ou planifiée) dans le cadre d'une démarche de rénovation d'architecture d'un logiciel et qui vise la compréhension, l'analyse ou la réutilisation du code source de ce logiciel. Les activités peuvent correspondre aux tâches visant :

— la sélection d'éléments de code (*e.g.,* sélectionner des *packages*) suivant différents critères ;

— la représentation du code source ou de la nouvelle architecture (*e.g.,* graphe de dépendances) ;

— l'allocation d'éléments de code à des éléments architecturaux ;

— le calcul de métrique sur le code source ou la nouvelle architecture (*e.g.,* nombre de lignes de code).

À partir d'entrevues avec les ingénieurs participant aux différents projets de rénovation d'architecture initiés au sein de Thales Air Systems, nous avons pu déterminer une liste de ces activités.

6.1.1 Entrevues avec les ingénieurs

Ces trois dernières années, en plus du projet de rénovation d'architecture que nous avons suivi, quatre autres projets du même type ont démarré au sein l'entreprise. Ils concernent des systèmes logiciels écrits dans trois langages de programmation différents (Ada, C , C++) et visent la rénovation de l'architecture du logiciel existant en la faisant évoluer vers une architecture à composants.

Lors des entrevues, trois de ces projets en étaient à un stade précoce, *i.e.,* les ingénieurs n'avaient pas encore démarré d'activités d'allocation de code source, bien qu'ils aient déjà commencé à les planifier. Le quatrième projet était en cours au moment des entrevues, ce qui signifie que les ingénieurs avaient planifié et démarré l'allocation du code source aux nouveaux composants.

Nous avons alors interrogé les ingénieurs travaillant sur ces projets, car ils avaient déjà une vision sur les activités qu'ils allaient ou avaient réalisées. Pour

chaque projet, les ingénieurs (architectes logiciels, experts et ingénieurs) que nous avons sélectionnés ont participé à la planification et à la réalisation de ces activités. Les neuf ingénieurs interrogés sont présentés dans le Tableau 6.1 suivant les projets auxquels ils ont participé ainsi que suivant leur responsabilité sur ces projets.

TABLE 6.1: Ingénieurs interrogés par projet et par responsabilités

	Projet A (précoce)	Projet B (précoce)	Projet C (précoce)	Projet D (en cours)
Architecte	1	1	1	1
Expert			1	1
Ingénieur(s)			1	2

Les projets A et B étant dans une phase précoce, nous n'avons pu interroger qu'un seul architecte travaillant sur le projet.

Le projet C était légèrement plus avancé que le A et le B, nous avons pu interroger une personne de chaque responsabilité. Tous ces ingénieurs ont participé à la planification des activités devant être réalisées.

Le projet D étant en cours, la réalisation de certaines activités avait déjà démarré ce qui nous a permis d'interroger un architecte, un expert et deux ingénieurs. Ces quatre personnes ont activement participé à la planification et la réalisation de ces activités.

Nous avons alors mené des entrevues informelles avec chacun de ces neuf ingénieurs, experts et architectes. Nous avons demandé à chaque personne de décrire :

— sa connaissance de l'ensemble du processus mis en place pour son projet ;

— les tâches qui allaient être ou avaient été mises en place pour son projet.

Tous les architectes ainsi que tous les experts ont indiqué avoir une forte connaissance de l'ensemble du processus. Les ingénieurs ont indiqué avoir une connaissance partielle du processus, *i.e.,* forte connaissance du processus d'allocation du code source aux nouveaux composants, mais peu de connaissance sur l'élaboration des nouvelles architectures elles-mêmes. Les résultats des entrevues sont présentés dans la section suivante et servent de base pour l'identification des activités.

6.1.2 Activités identifiées sur les différents projets

Afin de ne pas influencer les réponses des personnes interrogées dans nos entrevues, nous les avons laissées nous décrire les activités dans leurs propres termes. De ce fait, dans cette section, nous présentons uniquement un résumé des réponses de ces personnes au travers des activités que nous identifions.

Les activités décrites par les ingénieurs des quatre projets présentent des similitudes entre elles. Cela confirme le fait que les ingénieurs du projet que nous avons

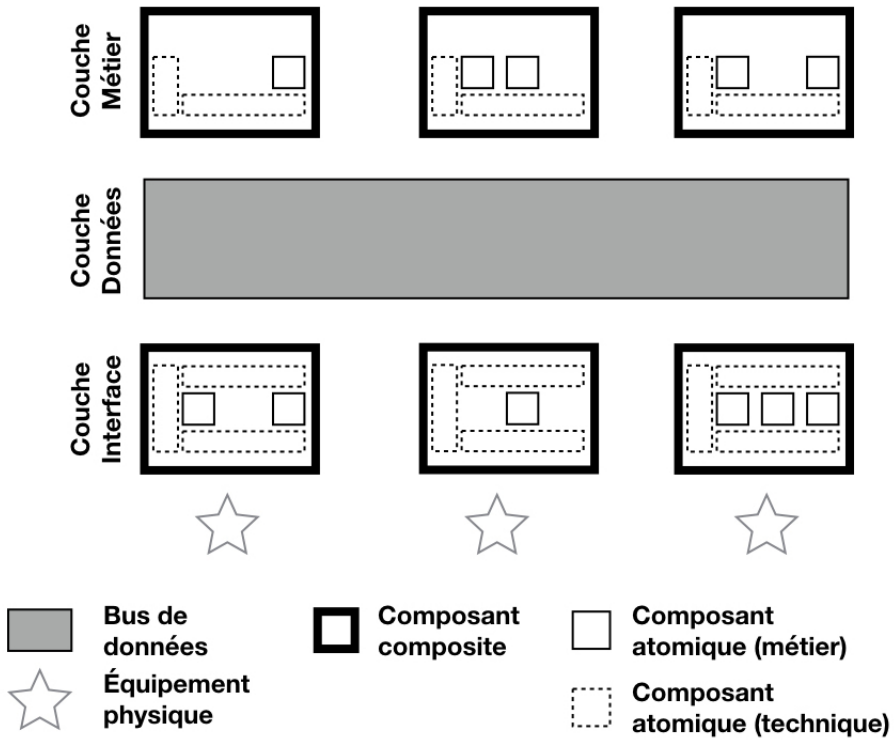


FIGURE 6.1: Exemple de la représentation d'une architecture faite par les ingénieurs de l'entreprise

suivi n'ont pas réalisé des activités spécifiques. Nous avons alors pu établir une liste d'activités d'allocation de code :

(ACT_1) Dans tous les projets, les ingénieurs ont ressenti le besoin de matérialiser les éléments d'une architecture visée, par exemple dans la Figure 6.1. Représenter graphiquement des éléments abstraits est une pratique courante en génie logiciel (*e.g.,* diagrammes UML, diagrammes SysML).

(ACT_2) Les ingénieurs ont également exprimé le besoin de matérialiser des contraintes entre les éléments de leurs architectures visées. Ces contraintes spécifient que certaines relations entre éléments architecturaux doivent exister ou à l'inverse qu'aucune relation entre ces éléments ne doit exister. Dans le cadre du projet décrit dans cette thèse, certaines contraintes entre composants composites et atomiques ont été identifiées.

(ACT_3) Dans tous les projets, les ingénieurs ont ressenti le besoin d'allouer des éléments du code à des éléments architecturaux. Cette allocation doit alors leur permettre de naviguer d'un élément architectural jusqu'au code source qui lui est alloué et inversement.

(ACT_4) Sur plusieurs projets, les ingénieurs ont sélectionné des éléments du code suivant l'emplacement de leurs fichiers sources dans l'organisation physique (*i.e.,* l'arborescence de dossiers et fichiers du projet). Par exemple, dans la démarche décrite au chapitre 4, les ingénieurs ont sélectionné tous les *packages* dont les fichiers sources sont contenus dans un dossier donné (le dossier "Librairie"). Cela correspond au critère d'allocation CA_o de notre démarche structurée. Dans un autre projet, les ingénieurs ont réalisé la même activité avec un dossier "Algorithme".

(ACT_5) Dans plusieurs projets, les ingénieurs ont sélectionné des éléments en fonction de leur nom, comme décrit dans le chapitre 4. Cela correspond au critère d'allocation CA_n sur les règles de nommage.

(ACT_6) Plusieurs ingénieurs ont mentionné au cours des entretiens que les conventions de nommage (recherche d'éléments selon leur nom, (ACT_5)) n'étaient pas suffisantes. Ils voulaient également sélectionner les éléments du code en fonction de leur contenu textuel (code et commentaires). Cela correspond à la technique de recherche d'information que nous avons expérimentée (dans le chapitre précédent) en tant que recherche ponctuelle plutôt que comme une recherche globale à la démarche des ingénieurs.

(ACT_7) Sur trois projets différents, les ingénieurs ont sélectionné des éléments du code en fonction du type d'éléments qu'ils contiennent, par exemple tous les *packages* Ada contenant une tâche Ada. Cette activité correspond à la vérification du critère d'allocation CA_c de la première phase de notre démarche structurée.

(ACT$_8$) Les ingénieurs ont également sélectionné des éléments suivant leurs relations avec un élément de départ donné. Par exemple, ils ont sélectionné tous les *packages* qui sont importés par un *package* de départ qu'ils ont choisi. Dans un autre projet, la relation d'invocation a été utilisée pour sélectionner des sous-programmes à partir d'un sous-programme initial. Cette activité correspond à la vérification du critère d'allocation CA_d de notre démarche structurée.

(ACT$_9$) Sur trois projets différents, dont celui décrit dans cette thèse, les ingénieurs ont sélectionné des éléments du code qui utilisent une même variable donnée. Contrairement à (ACT$_8$), ceci ne peut pas être fait récursivement, car le type des éléments sélectionnés (*e.g.,* sous-programmes) est différent du type de l'élément de départ (variable). Dans la deuxième phase de notre démarche structurée, nous avons indiqué que la vérification du critère d'allocation CA_r nécessite d'identifier les sous-programmes qui utilisent les mêmes variables afin de les allouer au même composant atomique. Cette identification de sous-programmes correspond à cette activité.

(ACT$_{10}$) Sur chacun des projets, les ingénieurs ont exprimé le besoin de sélectionner des éléments de code alloués à plusieurs éléments architecturaux. Par exemple, dans le projet décrit dans cette thèse, les ingénieurs ont sélectionné tous les *packages* qu'ils avaient alloués à plusieurs composants composites.

(ACT$_{11}$) Dans les quatre projets, les ingénieurs ont indiqué avoir besoin de visualiser la "responsabilité d'une variable" d'un élément architectural, c'est-à-dire si cet élément architectural est le seul à lire (respectivement, écrire) une variable. Ces informations proviennent du code source alloué à cet élément architectural. Bien que cette activité présente des similarités avec (ACT$_8$) et (ACT$_9$), elle diffère de ces deux activités sur l'objectif final. Les activités (ACT$_8$) et (ACT$_9$) ont pour but de sélectionner des éléments du code, alors que l'activité (ACT$_{11}$) a pour but de visualiser certaines propriétés d'un élément architectural donné.

(ACT$_{12}$) Pour chaque projet, les ingénieurs ont indiqué qu'ils souhaitaient pouvoir visualiser les dépendances entre les éléments architecturaux. Ces dépendances proviennent des relations réelles entre les éléments de code alloués aux éléments architecturaux. Les accès aux variables, les références aux types et les invocations des sous-programmes définis sur les éléments du code sont agrégés au niveau des éléments architecturaux auxquels ils sont alloués. Par exemple, dans un projet, les ingénieurs visualisaient toutes les semaines les relations induites entre les éléments architecturaux afin d'établir un état des lieux subjectif de l'architecture. Bien que cette activité et l'activité (ACT$_{11}$) visent à abstraire des informations relationnelles à un

niveau hiérarchique donné, elles diffèrent sur les éléments impliqués dans les relations. En effet, l'activité (ACT_{11}) permet de visualiser une relation entre un élément architectural et un élément de code, donc deux éléments d'un niveau d'abstraction différents. Au contraire, l'activité (ACT_{12}) permet de visualiser une relation entre deux éléments architecturaux, donc deux éléments du même niveau d'abstraction.

(ACT_{13}) Pour chaque projet, les ingénieurs ont indiqué devoir visualiser les valeurs de différentes métriques calculées sur les éléments de leur architecture. Par exemple, dans un de ces projets, les ingénieurs ont voulu visualiser le nombre de *packages* alloués aux différents éléments architecturaux. L'objectif était d'identifier l'élément architectural avec le plus de *packages* afin de commencer la phase de raffinage par celui-ci. Toutefois, de manière générale, l'objectif peut être différent, les ingénieurs ont donc indiqué qu'ils veulent principalement pouvoir visualiser les valeurs d'une métrique. La vérification de la répartition des métriques ne fait pas partie du périmètre de cette activité.

(ACT_{14}) Dans chaque projet, les ingénieurs ont exprimé le besoin de comparer les relations attendues ou non entre éléments architecturaux et les relations induites entre ces mêmes éléments. Typiquement, dans le projet décrit dans cette thèse, les ingénieurs ont voulu comparer les relations induites entre les composants composites avec celles auxquelles ils s'attendaient. Cette comparaison leur a permis d'identifier tout d'abord que certains *packages* avaient été incorrectement alloués et ensuite de modifier ces allocations pour que les relations induites soient conformes aux attendues. Dans d'autres cas, cela leur a permis d'identifier que les relations, auxquelles ils s'attendaient, étaient incorrectes et ils ont ainsi pu rectifier leur architecture attendue.

(ACT_{15}) Dans trois projets, les ingénieurs ont voulu vérifier la répartition de différentes métriques calculées sur les éléments de leur architecture. Par exemple, dans un de ces projets, les ingénieurs ont vérifié la répartition de la métrique du nombre de lignes de code sur les éléments architecturaux. À travers cette vérification, ils ont constaté qu'un élément (sur les 6) contenait la moitié des lignes de code de l'application. Cela leur a permis d'identifier un problème dans l'allocation de code source qu'ils avaient réalisée.

À travers les réponses des ingénieurs, nous avons vérifié que les activités couvrent les cinq critères d'allocation que nous avons définis au chapitre 2. Toutefois, l'inverse n'est pas vrai et nous voyons que plusieurs activités sont décorrélées de nos critères. Or, ces activités sont très chronophages, car elles sont fastidieuses, exécutées manuellement et répétées plusieurs fois sur un ou plusieurs projets. Généraliser et structurer ces activités permettra de les automatiser et de les répliquer sur

d'autres projets plus facilement.

6.2 Opérateurs : outillages d'une démarche de rénovation d'architecture logicielle

Comme énoncé plus tôt, nous avons souhaité généraliser et structurer les activités décrites dans la section précédente. Dans ce but, nous proposons ce que nous appelons des "opérateurs" qui correspondent à une généralisation et une structuration de ces activités.

Nous avons organisé nos opérateurs en cinq catégories : Modélisation, Allocation, Sélection, Abstraction et Vérification. Chaque opérateur est décrit selon le même schéma :

— une courte description de l'opérateur ;
— ses entrées attendues ;
— une description de son comportement ;
— les sorties qu'il produit ;
— des exemples de son utilisation ;
— les activités qu'il automatise.

6.2.1 Opérateur de Modélisation

Afin de pouvoir créer des liens entre les éléments de code et une architecture, il est nécessaire de modéliser cette dernière. Nous proposons pour cela deux opérateurs.

6.2.1.1 Modélisation d'élément

Comme son nom l'indique, cet opérateur permet de modéliser un élément d'une architecture.

Entrée : un nom (une chaîne de caractère).

Comportement : Cet opérateur crée et renvoie simplement un élément architectural avec le nom donné.

Sortie : Un élément architectural.

Exemple : La création du composant composite appelé "Librairie", de la nouvelle architecture du projet que nous avons suivi, peut être réalisée en utilisant l'opérateur de **Modélisation d'élément** avec le nom "Librairie". Cet opérateur peut être utilisé dès lors que l'on souhaite matérialiser les éléments d'une architecture.

Activité liée : (ACT_1).

6.2.1.2 Modélisation de relations entre éléments

Cet opérateur permet de modéliser une relation entre deux éléments d'une architecture.

Entrée :

- deux éléments architecturaux
- un booléen "attendu"/"non-attendu"

Comportement : Cet opérateur crée et renvoie une relation entre deux éléments architecturaux. Cette relation peut être soit "attendu" (c'est-à-dire qu'elle doit exister) ou "non-attendu" (c'est-à-dire qu'elle ne doit pas exister)

Sortie : Une relation entre deux éléments architecturaux.

Exemple : Dans le projet décrit dans cette thèse, les ingénieurs ont identifié que certains composants composites ne devaient pas avoir de relations entre eux. Ces relations sont alors considérées comme "non-attendues".

Activité liée : (ACT_2).

6.2.2 Opérateur d'Allocation

Pour réutiliser les éléments de code existant d'un logiciel dans une nouvelle architecture, ces éléments de code doivent être associés à des éléments de cette architecture. Pour cela, nous proposons un opérateur.

6.2.2.1 Allocation d'éléments

Cet opérateur a pour objectif d'allouer des éléments de code à un élément architectural.

Entrée :

- un ensemble d'éléments du code ;
- un élément architectural ;
- un booléen "remplace"/"ajoute" (par défaut, "ajoute").

Comportement : Cet opérateur crée une relation d'allocation entre l'ensemble d'éléments de code et l'élément architectural donné en entrée. Par défaut, une nouvelle relation d'allocation est ajoutée aux relations d'allocations déjà existantes entre l'ensemble d'éléments de code et d'autres éléments architecturaux. Lorsque le booléen donné en entrée est "remplace" alors cet opérateur remplace les relations d'allocation déjà existantes par la nouvelle relation d'allocation.

Sortie : Relations d'allocation entre les éléments de code et l'élément architectural.

Exemple : L'allocation des *packages* aux composants composites (comme décrit au chapitre 2) est réalisable grâce à cet opérateur.

Activité liée : (ACT_3).

6.2.3 Opérateurs de sélection

Parmi les activités décrites dans la Section 6.1, les activités (ACT_4) à (ACT_9) visent à sélectionner des éléments du code selon différentes caractéristiques (nom, code source, relations). Nous décrivons les opérateurs créés pour automatiser ces activités dans cette section.

6.2.3.1 Organisation physique

Cet opérateur sélectionne les éléments du code selon le dossier dans lequel ils sont contenus.

Entrée :

- un ensemble d'éléments du code (par défaut pour tous les éléments du système) ;
- un dossier de l'organisation physique du logiciel (*i.e.,* l'arborescence de dossiers et fichiers sources du logiciel).

Comportement : Cet opérateur parcourt l'organisation physique du logiciel pour sélectionner les éléments de l'ensemble d'entrée dont les fichiers de code source sont contenus dans le dossier donné en entrée.

Sortie : un ensemble d'éléments du code, qui est un sous-ensemble de l'ensemble d'éléments en entrée.

Exemple : Cet opérateur peut être utilisé pour vérifier le critère d'allocation CA_o de notre démarche structurée qui sélectionne les *packages* Ada contenus dans le dossier de "Librairie".

Activité liée : (ACT_4).

6.2.3.2 Convention lexicale

L'objectif de cet opérateur est de sélectionner tous les éléments de code dont la distance par rapport à une certaine convention lexicale est supérieure à un seuil donné.

Entrée :

— un ensemble d'éléments du code (par défaut pour tous les éléments du système) ;

— une chaîne de caractère à rechercher ;

— un contexte, *i.e.,* une combinaison quelconque de : "commentaires", "code" et "nom" (tous par défaut) ;

— un seuil d'acceptation en pourcentage (par défaut 0%).

Comportement : L'opérateur de **Convention lexicale** classe les éléments fournis en entrées, en fonction de leur distance par rapport à la chaîne de caractère recherchée. Le contexte indique où chercher la chaîne de caractère, *i.e.,* dans les commentaires, dans le nom de l'élément (convention de nommage) ou dans le corps de l'élément. L'opérateur filtre les éléments du code ayant une similarité supérieure au seuil choisi (le seuil par défaut est de 0%, toute similarité non nulle est acceptée). La similarité est calculée selon un algorithme TF-IDF par défaut.

Sortie : un ensemble d'éléments du code, qui est un sous-ensemble de l'ensemble d'éléments en entrée.

Exemples : Sélectionner les sous-programmes dont le code source contient "identification" correspond à l'application de l'opérateur de convention lexicale où : tous les sous-programmes du logiciel forment l'ensemble d'éléments en entrée ; la chaîne de caractère à rechercher est "identification" ; le contexte est {"commentaires", "code", "nom"} ; le seuil est 0%.

Activités liées : (ACT_5), (ACT_6).

6.2.3.3 Convention structurelle

Cet opérateur sélectionne les éléments du code qui contiennent (ou non) un autre type d'élément.

Entrée :

— un ensemble d'éléments du code (par défaut, tous les éléments du système) ;

— un type d'élément de code (*e.g.,* tâche Ada) ;

— une condition booléenne "contient" / "ne contient pas".

Comportement : Suivant la condition booléenne, l'opérateur de **Convention structurelle** filtre les éléments du code de l'ensemble d'entrée qui contiennent ou non le type d'éléments du code recherché. La contenance entre deux éléments du code A et B peut être directe (A contient B) ou transitive (A contient un élément C, qui contient B).

Sortie : un ensemble d'éléments du code, qui est un sous-ensemble de l'ensemble d'éléments en entrée.

Exemples : La sélection des *packages* qui contiennent une Tâche Ada, comme pour le critère d'allocation CA_c de notre démarche structurée, peut être effectuée en utilisant l'opérateur de Convention structurelle. Tous les *packages* du logiciel forment l'ensemble d'éléments en entrée ; le type d'éléments du code recherché est la "tâche Ada" ; et la condition booléenne est "contient".

Activité liée : (ACT_7).

6.2.3.4 Extraction d'éléments liés

L'opérateur d'extraction d'éléments liés sélectionne tous les éléments du code dans le système, qui sont liés à un autre élément de code de l'ensemble d'entrée à travers un type de relation spécifique.

Entrée :

- un ensemble d'éléments du code (par défaut, tous les éléments du système) ;
- un type de relations entre éléments du code (*e.g.,* invocations de sous-programme, import de *package*, *etc.*) ;
- un critère d'arrêt sur le nombre d'itérations et sur les éléments extraits (par défaut, une seule exécution et aucun critère sur les éléments extraits).

Comportement : Cet opérateur sélectionne tous les éléments de code qui sont liés à un élément donné par un type de relation donné. Contrairement aux deux opérateurs précédents, il ne s'agit pas d'un filtre sur l'ensemble d'éléments de code en d'entrée, mais d'une sélection d'autres éléments de code du système. Il peut effectuer une recherche récursive jusqu'à ce que le critère d'arrêt soit satisfait. Ce critère d'arrêt porte sur le nombre d'itérations de l'opérateur ainsi que sur les éléments extraits. Par défaut, une seule itération est effectuée et l'opérateur accepte tous les éléments afin de copier le comportement d'une navigation à un niveau à travers un graphe de dépendances. D'autres critères d'arrêts sont fournis dans l'exemple.

Sortie : un ensemble d'éléments du code.

Exemples : La Figure 6.2 illustre comment l'opérateur d'**Extraction d'éléments liés** pourrait aider à réaliser le critère d'allocation CA_d où les packages sont sélectionnés suivant les imports. À l'itération 0, l'ensemble d'entrée contient un seul package (Pkg1) pris comme point de départ ("noyau"). Le type de relation choisi est l'import réel entre *packages* Ada. Le critère d'arrêt est 3

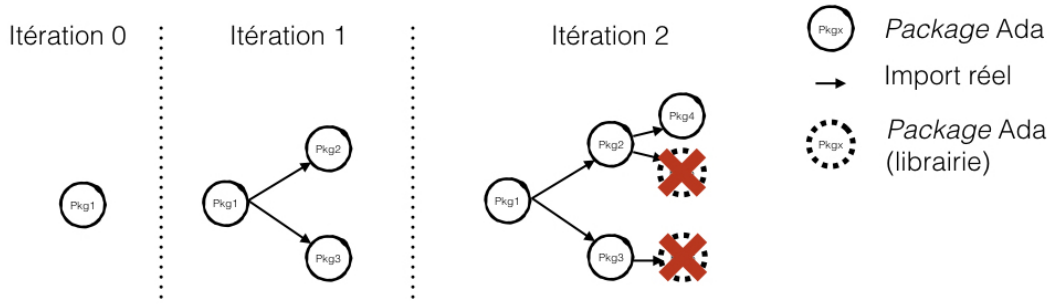


FIGURE 6.2: Exemple d'utilisation de l'opérateur d'Extraction d'éléments liés depuis un seul *package* suivant les imports réels et se stoppant après 3 itérations en atteignant les *packages* de "Librairie"

itérations et si le package importé est alloué au composant spécial "Librairie". L'itération 1 suit les imports de Pkg2 et Pkg3 par le Pkg1. L'itération 2 suit les imports de Pkg4 et Pkg5 en provenance du Pkg2, et les imports de Pkg6 en provenance du Pkg3. D'autre part, Pkg5 et Pkg6 sont des *packages* de la librairie et l'itération s'arrête ici pour ces branches. Pkg4 n'a aucun import, et l'itération se termine. L'ensemble de sortie est {Pkg1, Pkg2, Pkg3, Pkg4}.

Dans la pratique, nous avons utilisé cet opérateur avec la condition d'arrêt par défaut (une seule itération). Typiquement, nous avons utilisé cet opérateur pour fournir aux utilisateurs la liste des sous-programmes appelés par un autre sous-programme. Les utilisateurs ont alors pu sélectionner manuellement certains des sous-programmes de cette liste et demander une nouvelle itération sur la sous-liste obtenue.

Activités liées : (ACT_8), (ACT_9)

6.2.3.5 Multiplicité d'allocation

Cet opérateur sélectionne les éléments du code qui sont alloués à un nombre d'éléments architecturaux donné.

Entrée :

- un ensemble d'éléments du code (par défaut, tous les éléments du système) ;
- une multiplicité (par défaut 1).

Comportement : Cet opérateur sélectionne les éléments de l'ensemble d'entrée pour lesquels la multiplicité d'entrée correspond au nombre d'éléments architecturaux auxquels ils sont alloués.

Sortie : un ensemble d'éléments du code, qui est un sous-ensemble de l'ensemble d'éléments en entrée.

Exemples : Dans la première phase de notre démarche structurée, nous cherchons à sélectionner les *packages* alloués à plusieurs composants composites. Cette sélection peut être réalisée grâce à l'opérateur de **Multiplicité d'allocation** avec en entrée : tous les *packages* du logiciel ; une multiplicité de (2..*) .

Activité liée : (ACT_{10}).

6.2.4 Opérateurs d'abstraction

Les opérateurs d'abstraction visent à abstraire des informations d'un niveau hiérarchique donné à des niveaux hiérarchiques plus élevés (éléments de code conteneurs ou éléments architecturaux). Nous proposons deux opérateurs dans cette catégorie.

6.2.4.1 Abstraction de relations

Cet opérateur abstrait les dépendances entre éléments du code au niveau des éléments conteneurs (éléments de code conteneurs ou éléments architecturaux) auxquels ils sont contenus ou alloués.

Entrée :

- deux éléments conteneurs de même niveau hiérarchique (*e.g., packages* ou éléments architecturaux) ;
- un ensemble de types de relations (*e.g.,* invocation de sous-programme, import de *package* , *etc.*. Par défaut, il s'agit de toutes les relations possibles).

Comportement : Cet opérateur se base sur le concept de relations induites explicité au chapitre 4. Pour rappel, une relation induite est une relation réelle entre deux éléments de code (*e.g.,* invocation de sous-programmes, imports de *packages*) qui est abstraite à un niveau hiérarchique plus élevé. En se basant sur le type de relations en entrée, cet opérateur calcule les relations induites entre les deux éléments conteneurs d'entrée.

Sortie : Les relations induites (s'il y en a) entre les deux éléments d'entrée.

Exemples : La Figure 6.3 montre un exemple d'abstraction de relations entre trois composants. Dans cette figure, les relations d'imports réels entre les *packages* (partie gauche) sont abstraites au niveau des composants auxquels ils sont alloués (partie droite). L'import entre les *packages* Pkg1 et Pkg2 n'apparaît pas au niveau des composants, car ils sont tous les deux alloués au même composant (C1). *A contrario*, la relation entre Pkg2 et Pkg3 est abstraite pour

devenir la relation induite entre les composants C1 et C2.
begindescription
Un autre exemple d'abstraction de relations est montré dans La Figure 6.4. Cet exemple montre l'abstraction des relations d'utilisation de variables par les sous-programmes. Dans cette figure, les sous-programmes et les variables sont déclarés et contenus dans les *packages*. L'utilisation des variables par les sous-programmes (partie gauche) est alors abstraite au niveau des *packages* pour devenir leurs relations induites (partie droite). La relation d'utilisation d'une variable par un sous-programme contenu dans le même *package* ne produit pas de relation induite au niveau du *package*.

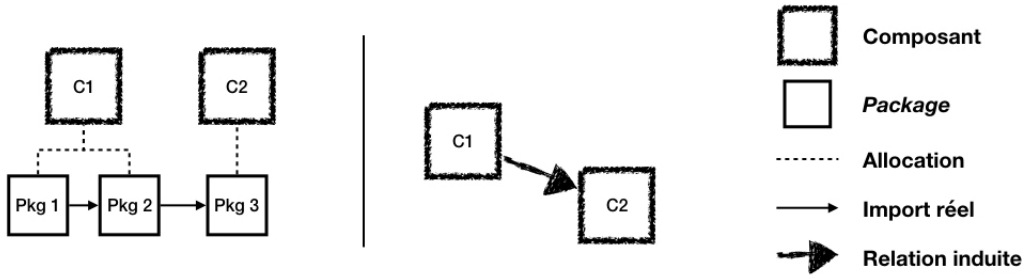


FIGURE 6.3: Exemple d'abstraction de relations entre deux composants

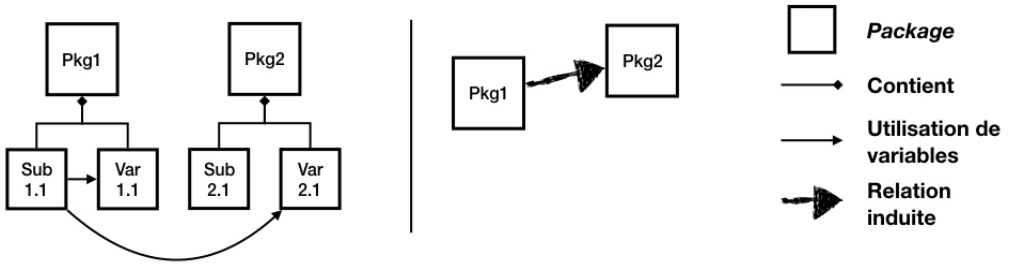


FIGURE 6.4: Exemple d'abstraction de relations entre deux *packages*

Activités liées : (ACT_{11}), (ACT_{12}).

6.2.4.2 Abstraction de métriques

Cet opérateur abstrait, au niveau d'un élément donné, une métrique appliquée sur les éléments de code qu'il contient.

Entrée :

- un élément conteneur (*e.g., package* ou élément architectural) ;
- une métrique logicielle existante (*e.g.,* nombre de lignes de code) ;
- une méthode d'agrégation (par défaut, la somme).

Comportement : L'opérateur d'abstraction de métriques a pour but d'agréger, au niveau d'un élément conteneur, une métrique calculée sur les éléments de code qu'il contient. L'élément conteneur peut être un *package* qui contient des sous-programmes par exemple ou alors un élément architectural auquel des *packages* sont alloués. La méthode d'agrégation utilisée est en général assez simple, telle qu'une somme pour agréger le nombre de lignes de code (LoC) des sous-programmes au niveau de leur *package* ou une moyenne pour agréger la complexité cyclomatique des méthodes au niveau de leur classe.

Sortie : un nombre

Exemples : Un exemple du fonctionnement de cet opérateur pour abstraire le nombre de *LoC* à un composant est montré en Figure 6.5. Dans cet exemple, l'opérateur abstrait la métrique du nombre de *LoC* au niveau du composant C1 en sommant les valeurs de cette métrique pour chaque élément de code alloué au composant C1.
La Figure 6.6 fournit un autre exemple d'abstraction d'une métrique. Dans cette figure, l'opérateur fait la moyenne des complexités cyclomatiques des éléments contenus dans le *package* Pkg1 et donne cette moyenne comme complexité cyclomatique du Pkg1.

Dans (ACT_{13}), nous avons expliqué que les ingénieurs ont découvert que, sur 6 composants atomiques d'un composant composite donné, l'un d'entre eux contenait la moitié de toutes les lignes de code du composant composite. L'opérateur d'**Abstraction de métriques** permet de calculer les différentes valeurs de cette métrique.
L'opérateur d'abstraction de métriques permet de calculer les différentes valeurs des métriques.

Activité liée : (ACT_{13}).

6.2.5 Opérateurs de vérification

Lors de la création d'une architecture, les ingénieurs ont établi que certaines propriétés devaient être vérifiées après l'allocation des éléments de code aux éléments architecturaux. Nous proposons ici deux opérateurs permettant de vérifier ces propriétés.

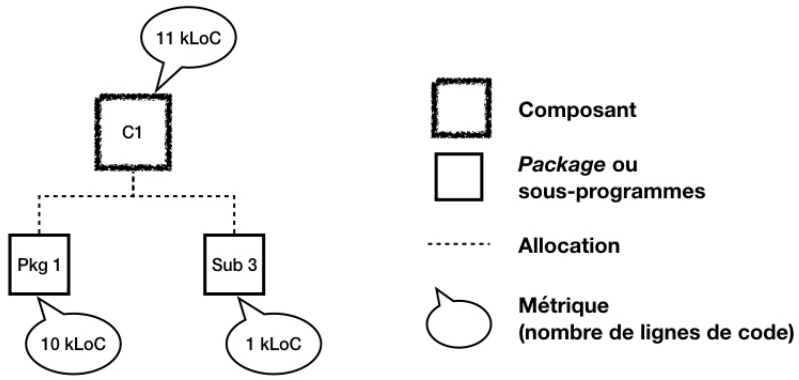


FIGURE 6.5: Exemple d'abstraction du nombre de *LoC* à un composant par une somme

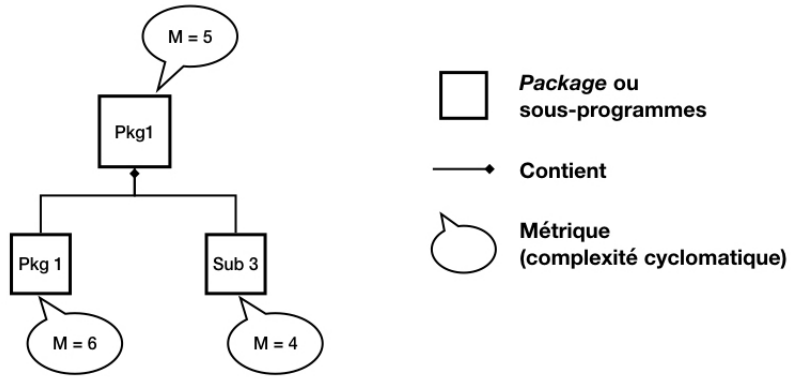


FIGURE 6.6: Exemple d'abstraction de la complexité cyclomatique au niveau d'un *package* par la moyenne

6.2.5.1 Vérification de relations

Cet opérateur vérifie que les relations induites entre plusieurs éléments architecturaux respectent bien les relations (attendues ou non-attendues) définies entre eux.

Entrée :

- un ensemble d'éléments architecturaux ;
- un ensemble de relations induites (*e.g.,* calculées par l'opérateur d'**Abstraction de relations**) ;
- un ensemble de relations attendues ou non (*e.g.,* définies grâce à l'opérateur de **Modélisation de relations**).

Comportement : Cet opérateur vérifie que les relations induites entre les éléments architecturaux d'entrée respectent leurs relations attendues. Les relations induites sont calculées à partir du type de relations donné en entrée et des éléments de code alloués aux éléments architecturaux d'entrée.

Sortie : l'ensemble des relations induites non-conformes aux relations attendues

Exemples : Dans la première phase de la démarche de l'entreprise présentée au chapitre 4, les ingénieurs ont vérifié manuellement les relations attendues et induites des composants composites. Une telle vérification peut être réalisée grâce à cet opérateur.

Activité liée : (ACT_{14}).

6.2.5.2 Vérification de répartition de métrique

Cet opérateur vérifie la répartition d'une métrique sur des éléments architecturaux de même niveau.

Entrée :

- un ensemble de couples (éléments architecturaux, valeurs). Les éléments architecturaux doivent être du même niveau et les associées correspondre à la même métrique ;
- une loi de répartition à vérifier (par défaut, loi uniforme continue) ;
- un pourcentage de marge d'erreur (par défaut, 5%).

Comportement : Cet opérateur vérifie que les valeurs de la métrique sur les éléments d'entrée, respectent la loi de répartition choisie selon la marge d'erreur.

Sortie : La répartition de la métrique d'entrée et sa conformité à la loi de répartition.

Exemples : Dans un des projets de notre étude, les ingénieurs ont voulu vérifier que le nombre de lignes de code du système était uniformément réparti. Une telle vérification peut être réalisée grâce à cet opérateur.

Activité liée : (ACT_{15}).

6.3 Opérateurs : activités et démarche de rénovation d'architecture chez Thales Air Systems

Dans la section précédente, nous avons décrit un ensemble d'opérateurs généralisant les activités identifiées dans la première section. Dans un premier temps, cette section propose de montrer le recouvrement de ces activités par nos opérateurs. Dans un second temps, nous montrons comment nos opérateurs auraient pu être appliqués pour la démarche des ingénieurs décrite au chapitre 2.

6.3.1 Recouvrement des activités des ingénieurs par nos opérateurs

Nous avons proposé nos opérateurs dans le but de généraliser et structurer les activités identifiées dans la Section 6.1, nous devons donc vérifier que nous avons couvert chacune de ces activités. Suivant les définitions de chacun des opérateurs, nous avons résumé dans le Tableau 6.2 la correspondance entre les activités des ingénieurs et les opérateurs qui les automatisent.

TABLE 6.2: Correspondance entre les opérateurs et les activités des ingénieurs qu'ils automatisent

	Mod. d'élts	Mod. de rel.	Alloc. d'élts	Orga. phys.	Conv. lex.	Conv. struct.	Extract. d'élts liés	Mult. d'alloc.	Abstr. de rel.	Abstr. de mét.	Vérif. de rel.	Vérif. de mét.
(ACT_1)	X											
(ACT_2)		X										
(ACT_3)			X									
(ACT_4)				X								
(ACT_5)					X							
(ACT_6)					X							
(ACT_7)						X						
(ACT_8)							X					
(ACT_9)							X					
(ACT_{10})								X				
(ACT_{11})									X			
(ACT_{12})									X			
(ACT_{13})										X		
(ACT_{14})											X	
(ACT_{15})												X

Mod. = Modélisation
Extract. = Extraction
élts = éléments
struct. = structurelle

Alloc. = Allocation
Mult. = Multiplicité
rel. = relations
mét. = métriques

Orga. = Organisation
Abstr. = Abstraction
phys. = physique

Conv. = Convention
Vérif. = Vérification
lex. = lexicale

À partir de ce tableau, nous pouvons voir que toutes les activités que nous avons identifiées dans la Section 6.1 sont couvertes par nos opérateurs. D'autre part, le tableau permet de visualiser que certains opérateurs généralisent plusieurs activités alors que d'autres n'en généralisent qu'une.

En se basant sur les entrevues avec les ingénieurs, experts et architectes, ils semblent cependant qu'ils ne prévoient aucune activité de vérification automatique des relations architecturales qu'ils définissent grâce à l'activité (ACT_2). Ils nous ont indiqué qu'ils ne nécessitaient pas d'opérateur automatisant une telle activité, car les relations attendues étaient manuellement vérifiables en utilisant l'opérateur d'abstraction des dépendances. Néanmoins, nous pensons que d'autres projets pourraient nécessiter des contraintes s'avérant plus complexes et un opérateur plus complet serait alors nécessaire. De nombreuses propositions ont été faites à cet égard dans le domaine de la recherche en langage de la description architecturale [Garlan 2000, Passos 2010]. On pourrait également se tourner vers des exemples de langages simples pour exprimer de telles contraintes [Terra 2012, Santos 2015a]. Une étude sur un tel opérateur devrait être réalisée pour continuer les travaux sur ce sujet.

D'autre part, nos opérateurs respectent chacune des exigences du projet de l'entreprise définies au chapitre 2 :

(exi_i) technique automatisable : nos opérateurs peuvent être automatisés ;

(exi_{ii}) technique d'analyse statique : nos opérateurs sont basés sur des activités d'analyse statique réalisées par les ingénieurs de l'entreprise ;

(exi_{iii}) technique supportant le passage à l'échelle : ils ont tous été appliqués sur au moins un projet de l'entreprise (tous supérieur à 300kLoC) ;

(exi_{iv}) technique applicable pour la reconnaissance d'une architecture à composants sur un logiciel avec une architecture différente : nos opérateurs se basent sur les activités mises en place pour automatiser l'allocation du code source existant d'un logiciel actuellement architecturé en module dans les éléments d'une nouvelle architecture à composants.

(exi_v) technique adaptable : les définitions de ces opérateurs montrent qu'ils sont adaptables à d'autres projets ;

(exi_{vi}) technique pour logiciel sans base de données : à travers leur définition aucun de ces opérateurs ne nécessite que le logiciel à retravailler contienne une base de données

6.3.2 Exemple d'application de nos opérateurs sur la démarche de Thales Air Systems

Nos opérateurs ont pour objectif de faciliter la réplication et l'automatisation de la démarche appliquée par Thales Air Systems. Puisque nous avons structuré

cette démarche dans le chapitre 4, nous montrons dans cette section la réplication de notre démarche structurée grâce à nos opérateurs.

Pour rappel, cette démarche est constituée de deux phases : (1) une première phase visant l'allocation de *packages* Ada à des composants composites ; (2) une deuxième phase visant l'allocation de sous-programmes Ada à des composants atomiques. De plus, chaque phase se base sur le *Reflexion Model* et est donc décomposée en cinq étapes :

1. Matérialisation de l'architecture attendue ;
2. Extraction d'informations à partir du code source ;
3. Allocation d'éléments de code aux éléments de l'architecture attendue ;
4. Calcul des relations induites ;
5. Comparaison des relations induites avec les relations attendues.

Les réplications des deux phases de notre démarche sont présentées dans les sous-sections suivantes.

6.3.2.1 Première phase : allocation des *packages* aux composants composites

La figure 6.7 présente l'utilisation de nos opérateurs pour répliquer et faciliter l'automatisation de la première phase de notre démarche structurée.

La première étape de cette phase nécessite de matérialiser l'architecture attendue, *i.e.,* matérialiser les éléments qui la composent et les relations entre ces éléments. Les éléments de l'architecture sont matérialisables en utilisant l'opérateur de **Modélisation d'éléments** pour chacun des éléments architecturaux. Les relations entre ces éléments, quant à elle, sont matérialisables grâce à l'opérateur de **Modélisation de relations**.

La deuxième étape a pour objectif d'extraire les informations nécessaires à la réalisation de cette phase à partir du code source. Cette étape nécessite l'utilisation d'un parseur pour extraire tous les *packages* de l'application ainsi que leurs relations (*e.g.,* imports). Elle est indépendante de nos opérateurs, mais fournit une représentation des éléments du code source qui peut ensuite être utilisée par ces opérateurs.

La troisième étape a pour objectif d'allouer les éléments du code source aux éléments architecturaux attendus. En l'occurrence, nous cherchons à allouer les *packages* extraits précédemment aux composants composites de l'architecture attendue. Comme expliqué au chapitre 5, les *packages* sont alloués aux composants composites en vérifiant quatre critères d'allocation dans un ordre précis :

CA_o **:** La vérification de cette instance du critère CA_o se base sur l'organisation physique du code pour sélectionner les *packages* contenus dans un dossier

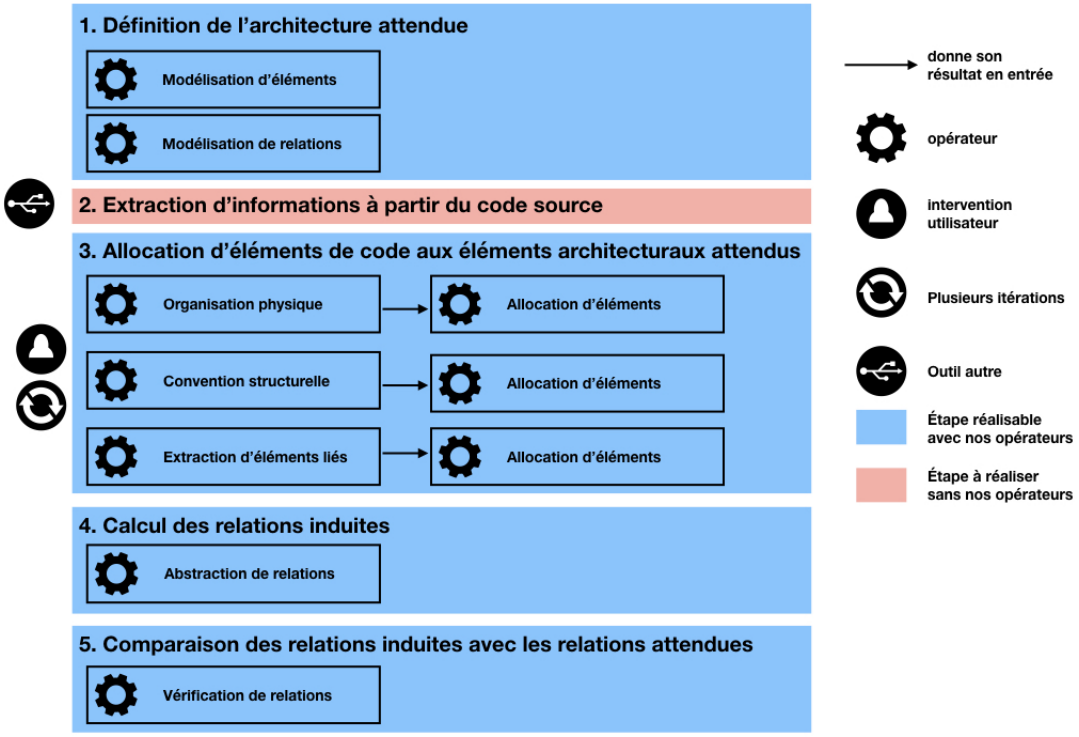


FIGURE 6.7: Première phase de notre démarche structurée en utilisant les opérateurs définis en Section 6.2

nommé "Librairie". Ces *packages* sont ensuite alloués au composant Librairie. Ceci peut être réalisé en utilisant l'opérateur d'**Organisation physique** avec en entrée :

— tous les *packages* de l'application ;

— le dossier nommé "Librairie".

Les *packages* ainsi obtenus peuvent alors être alloués à la librairie en utilisant l'opérateur d'**Allocation d'éléments** avec en entrée :

— les *packages* en sortie de l'opérateur précédent ;

— l'élément architectural matérialisant la "Librairie" ;

— le booléen "ajoute".

CA_c **:** La vérification de l'instanciation de ce critère pour la première phase de notre démarche structurée permet de sélectionner les *packages* contenant une tâche Ada. Les *packages* sélectionnés sont ensuite alloués aux composants composites par les ingénieurs suivant leur connaissance. Cette instance est vérifiable grâce à l'opérateur de **Convention structurelle** avec en entrée :

— tous les *packages* de l'application ;

— les tâches Ada comme type d'éléments de code ;

— et la condition "contient".

Les *packages* obtenus en sortie de cet opérateur peuvent alors être alloués aux composants composites en utilisant l'opérateur d'**Allocation d'éléments** avec en entrée :

— un *package* ;

— l'élément architectural matérialisant un certain composant composite (choisi par l'ingénieur) ;

— le booléen "ajoute".

Cette instance nécessite toutefois l'intervention d'un ingénieur pour déterminer le composant composite auquel sera alloué chacun des *packages* obtenus grâce à l'opérateur de **Convention structurelle**.

CA_d **:** La vérification de cette instance du critère CA_d se base sur les relations et dépendances entre *packages* pour sélectionner certains d'entre eux. Cela permet de sélectionner tous les *packages* dont les *packages* déjà alloués sont dépendants. Les *packages* ainsi sélectionnés sont ensuite alloués par les ingénieurs au même composant composite que le *package* qui dépend d'eux. Cette instance est vérifiable, pour chaque composant composite, à travers l'utilisation de l'opérateur d'**Extraction d'éléments liés** avec en entrée :

— tous les *packages* alloués à ce composant composite ;

— les imports réels entre les *packages* (*i.e.,* les packages importés et utilisés) ;

— ne passer qu'une seule fois par *package* importé (cela permet d'éviter les cycles), pas de limite d'itérations ;

Chaque *package* ainsi obtenu est alors alloué au composant composite pour lequel est vérifié ce critère grâce à l'opérateur d'**Allocation d'éléments** avec en entrée :

— les *packages* obtenus par l'opérateur précédent ;

— l'élément architectural matérialisant le composant composite choisi ;

— le booléen "ajoute".

CA_r La vérification de l'instance de ce critère pour la première phase de notre démarche structurée permet de résoudre les cas d'éléments de code multi-alloués suivant une règle donnée. Dans cette phase, cette vérification permet de résoudre l'allocation de chaque *package* en conflit par son allocation à la librairie. L'instance de ce critère pour cette phase est vérifiable par l'application de l'opérateur de **Multiplicité d'allocations** avec en entrée :

— tous les *packages* de l'application ;

— une multiplicité de (2..*).

Les *packages* ainsi obtenus sont ensuite alloués à la librairie en utilisant l'opérateur d'**Allocation d'éléments** avec en entrée :

— les *packages* obtenus par l'opérateur précédent ;

— l'élément architectural représentant la "Librairie" ;

— le booléen "remplace".

La quatrième étape a pour objectif de calculer les relations induites entre les éléments architecturaux modélisés en se basant sur les éléments de code alloués et leurs relations existantes. Le calcul des relations induites entre les composants composites correspond à l'application de l'opérateur d'**Abstraction de relations**. Pour chaque paire de composants composites (C_1, C_2), cet opérateur est appliqué avec en entrée :

— les éléments architecturaux matérialisant C_1 et C_2

— toutes les relations possibles à abstraire (valeur par défaut).

Enfin, la cinquième étape a pour objectif de comparer les relations induites précédemment calculées avec les relations attendues qui ont été matérialisées à l'étape 2. Dans cette phase, les relations induites entre composants composites sont comparées aux relations attendues et non-attendues que les ingénieurs ont définies plus tôt. Cette étape peut être réalisée en utilisant l'opérateur de **Vérification de relations** avec en entrée :

— tous les éléments architecturaux matérialisant des composants composites ;

— les relations induites entre ces éléments architecturaux ;

— les relations attendues ou non (matérialisées à l'étape 2).

.

6.3.2.2 Deuxième phase : allocation des sous-programmes aux composants atomiques

La figure 6.8 présente l'utilisation de nos opérateurs pour répliquer et faciliter l'automatisation de la deuxième phase de notre démarche structurée. Cette phase a pour but de raffiner l'allocation des *packages* aux composants composites. Ce raffinage est réalisé en allouant les sous-programmes contenus dans les *packages* d'un composant composite aux composants atomiques contenus dans ce composant composite. La réplication que nous expliquons est faite pour un composant composite donné (que nous appellerons C par commodité), mais est applicable pour n'importe quel autre composant composite.

La première étape nécessite de matérialiser les composants atomiques et leurs relations au sein d'un composant composite donné. Cette matérialisation est réalisable grâce aux opérateurs de **Modélisation d'éléments** et de **Modélisation de relations**.

La deuxième étape vise l'extraction des informations nécessaires à cette phase à partir du code source. De la même façon que pour la phase précédente, cette ne nécessite pas d'utiliser nos opérateurs, mais un parseur.

Cette étape ayant déjà été automatisée par l'utilisation de Moose, nous n'utilisons pas nos opérateurs pour la réaliser. Durant cette phase, Moose extrait les sous-programmes ainsi que toutes leurs relations (*e.g.,* accès aux variables, invocations de méthodes).

La troisième étape vise l'allocation aux composants atomiques des éléments de code extraits à l'étape précédente. Dans le chapitre 5, nous avons expliqué que cette étape est réalisée en vérifiant trois critères d'allocation dans un ordre défini :

CA_n **:** La vérification de cette instance du critère CA_n permet de sélectionner des éléments de code à partir de leurs informations textuelles. Ici, il s'agit de sélectionner tous les sous-programmes dont le nom contient une chaîne de caractère donnée et de réitérer ceci pour plusieurs chaînes. Chaque chaîne de caractère choisie est associée à un composant atomique donné. Cela permet ensuite d'allouer les sous-programmes sélectionnés pour une chaîne de caractère au composant atomique donné. Pour un composant atomique A, ce critère est vérifiable à l'aide de l'opérateur de **Convention lexicale** avec en entrée :

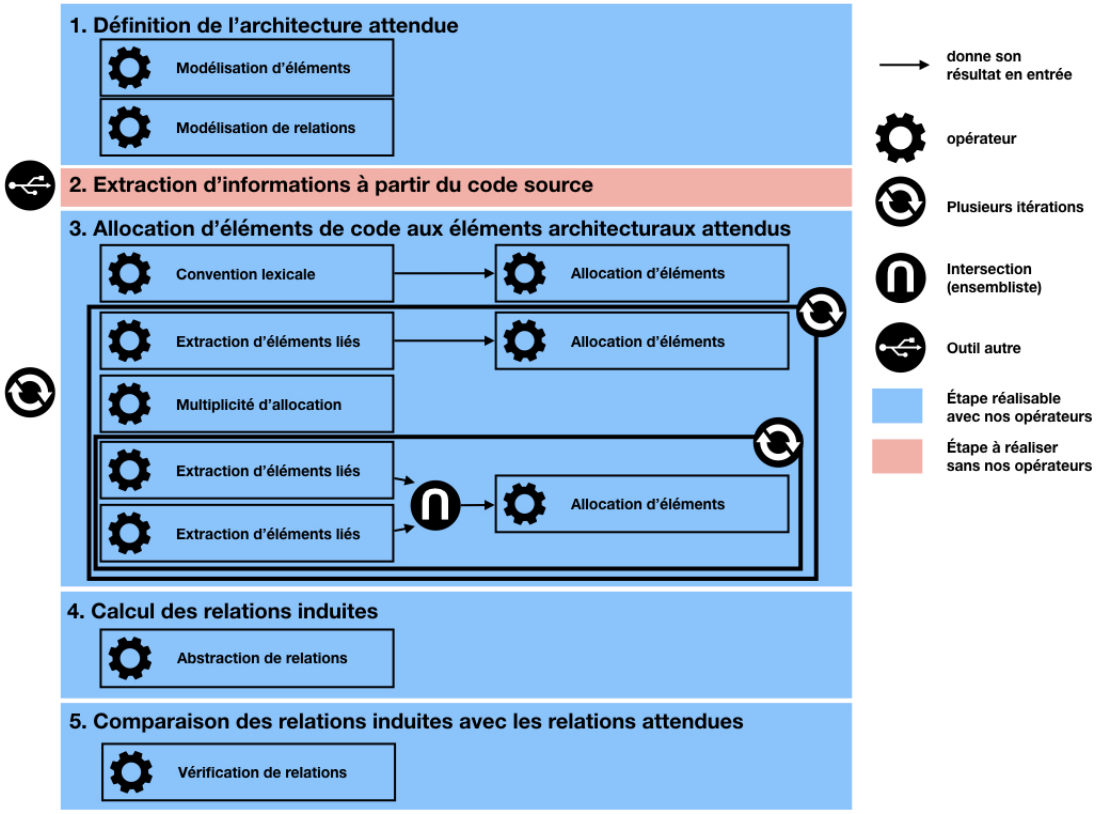


FIGURE 6.8: Deuxième phase de notre démarche structurée en utilisant les opérateurs définis en Section 6.2 (pour un composant composite donné)

— tous les sous-programmes contenus dans les *packages* alloués au composant composite C ;

— une chaîne de caractère correspondant au composant atomique A ;

— le "nom" est le contexte de recherche ;

— le seuil d'acceptation d'un résultat est de 0%.

Les sous-programmes ainsi sélectionnés peuvent être ensuite alloués au composant atomique A grâce à l'opérateur d'**Allocation d'éléments** avec en entrée :

— les sous-programmes en sortie de l'opérateur précédent ;

— l'élément architectural représentant le composant atomique A ;

— le booléen "ajoute"

CA_d **et** CA_r **:** Comme indiqué pour cette phase dans le chapitre 5, cette instance du critère de résolution des conflits CA_r doit être vérifiée pendant la vérification de l'instance du critère de dépendances CA_d. Les opérateurs suivants doivent donc être appliqués plusieurs fois pour un composant atomique A donné, jusqu'à ce que l'opérateur vérifiant le critère CA_d ne sélectionne plus de sous-programmes.

CA_d **:** La vérification de cette instance permet de sélectionner des éléments de code suivant leurs relations. Il s'agit ici de sélectionner tous les sous-programmes qui sont appelés directement par un sous-programme donné. Les sous-programmes ainsi sélectionnés sont ensuite alloués par les ingénieurs au composant atomique auquel est alloué le sous-programme donné qui les appelle. Pour un composant atomique A, cette instance est vérifiable par l'utilisation de l'opérateur d'**Extraction d'éléments liés** avec en entrée :

— tous les sous-programmes alloués au composant atomique A ;

— les invocations entre les sous-programmes ;

— limité à une itération (ne nécessite pas d'autre critère d'arrêt).

Les sous-programmes alors sélectionnés peuvent être alloués au composant atomique A grâce à l'opérateur d'**Allocation d'éléments** avec en entrée :

— les sous-programmes obtenus par l'opérateur précédent

— l'élément architectural matérialisant le composant atomique A ;

— le booléen "ajoute"

CA_r **:** La vérification de cette instance permet de résoudre les cas d'éléments de code multi-alloués suivant une règle donnée. Il s'agit ici d'identifier les sous-programmes alloués à plusieurs composants atomiques.

Le conflit d'allocation est résolu en allouant le sous-programme en question au composant atomique dont il utilise le plus de variables communes.

Dans cette phase, la vérification de ce critère permet de résoudre l'allocation de chaque sous-programme en conflit par son allocation au composant atomique avec lequel il possède le plus de variables utilisées communes. Le composant atomique auquel est finalement alloué le sous-programme fait partie des composants pour lequel il est en conflit.

Cette instance est vérifiable par l'utilisation de plusieurs opérateurs.
En premier, on utilise une première instance de l'opérateur de **Multiplicité d'allocations** avec en entrée :

— les sous-programmes obtenus avec l'opérateur d'**Extractions d'éléments liés** dans l'étape précédente.

— une multiplicité de (2..*).

Pour chacun des sous-programmes obtenus, nous appliquons premièrement, l'opérateur d'**Extraction d'éléments liés** avec en entrée :

— le sous-programme choisi (parmi ceux obtenus précédemment) ;

— les accès aux variables des sous-programmes ;

— limité à une itération (ne nécessite pas d'autre critère d'arrêt).

On applique ensuite l'opérateur d'**Extraction d'éléments liés** pour chaque composant atomique impliqué dans le conflit avec en entrée :

— les sous-programmes déjà alloués à un composant atomique impliqué dans le conflit ;

— les accès aux variables des sous-programmes ;

— limité à une itération (ne nécessite pas d'autre critère d'arrêt).

Nous appliquons ensuite une intersection ensembliste entre les deux ensembles de variables obtenus par les deux dernières applications de l'opérateur d'**Extraction d'éléments liés**. Afin d'identifier le composant atomique dont le sous-programme utilise le plus de variables. Le sous-programme est alors alloué à ce composant atomique grâce à l'opérateur d'**Allocation d'éléments** avec en entrée :

— le sous-programme en conflit

— l'élément architectural matérialisant le composant atomique différent de A auquel est alloué le sous-programme en conflit ;

— le booléen “remplace”

La quatrième étape a pour objectif de calculer les relations induites entre tous les composants atomiques du composant composite C. Le calcul des relations induites entre les composants atomiques correspond à l'application de l'opérateur d'**Abstraction de relations**. Pour chaque paire de composants atomiques (A_1, A_2), cet opérateur est appliqué avec en entrée :

— les éléments architecturaux matérialisant A_1 et A_2

— toutes les relations possibles à abstraire (valeur par défaut).

Enfin, la cinquième étape a pour objectif de comparer les relations induites précédemment calculées avec les relations attendues qui ont été matérialisées à l'étape 2. Dans cette phase, les relations induites entre composants atomiques sont comparées aux relations attendues et non-attendues que les ingénieurs ont définies plus tôt. Cette étape peut être réalisée en utilisant l'opérateur de **Vérification de relations** avec en entrée :

— tous les éléments architecturaux matérialisant des composants atomiques ;

— les relations induites entre ces éléments architecturaux ;

— les relations attendues ou non (matérialisées à l'étape 2).

Finalement, nous avons pu proposer une réplication de chacune des deux phases de notre démarche structurée en utilisant nos opérateurs. Cette réplication permet d'avoir alors une démarche semi-automatique pouvant être utilisée sur le projet de Thales Air Systems, mais aussi sur d'autres.

6.4 Conclusion

À travers ce chapitre, nous avons décrit un certain nombre d'"activités" réalisées par les ingénieurs de Thales Air Systems dans le cadre de plusieurs projets de rénovation d'architecture logicielle. Ces activités correspondent aux tâches réalisées par les ingénieurs dans le but de comprendre, analyser ou réutiliser le code source d'un logiciel lors de ces projets. Nous proposons alors des "opérateurs" généralisant ces activités dans le but de faciliter l'automatisation et la réplication de la démarche décrite dans cette thèse. Nous avons alors montré que nos opérateurs répondent à chacune des exigences du projet de Thales Air Systems décrites au chapitre 2. Enfin, nous proposons une réplication de la démarche décrite au chapitre 4 qui se base entièrement sur nos opérateurs.

CHAPITRE 7

Conclusion

Contents

7.1 Synthèse des travaux présentés

Thales Air Systems a mis en place un plan de maintenance logicielle ayant pour but l'évolution de certains de ses logiciels. Ces évolutions ont pour objectif de rénover les architectures de ces logiciels afin de faciliter leur maintenance. À la suite du premier projet d'évolution (étudié dans cette thèse), l'entreprise souhaite répliquer sur d'autres projets la démarche qui a été appliquée. Cette thèse a alors pour but de fournir une démarche d'évolution visant la rénovation de l'architecture d'un logiciel ; qui soit supportée par des moyens facilitant son automatisation et sa réplication à d'autres projets.

En premier lieu, le chapitre 2 a présenté une description du premier projet de ce plan de maintenance réalisé par l'entreprise. Cette description se base sur l'ontologie des projets de maintenance définie par Kitchenham *et al.* [Kitchenham 2002] décomposée en quatre grandes caractéristiques : (1) le produit impliqué dans le projet (*e.g.,* son âge, ses dimensions), (2) les activités réalisées durant ce projet (*e.g.,* investigation, modification), (3) la démarche réalisée durant ce projet (*e.g.,* organisation du projet, procédure systématique), (4) les ressources humaines (*e.g.,* le nombre de personnes, leurs niveaux d'expertise).

Le projet de Thales Air Systems concerne un logiciel âgé d'une vingtaine d'années, développé en Ada 83 et faisant plus de 300kLoC. Ce logiciel a fait l'objet de plusieurs activités d'investigation et de modification visant la rénovation de son architecture tout en réutilisant le code source existant. Ces activités ont été réalisées par les ingénieurs de l'entreprise suivant une démarche informelle constituées de scripts et d'actions manuelles. Douze personnes ont travaillé sur ce projet avec une

expérience différente pour chacun (de 5 à 15 ans d'expérience) et des niveaux de responsabilités différents (ingénieurs, experts ou architecte).

L'entreprise a pour objectif de répliquer sur d'autres projets et d'automatiser la démarche informelle appliquée par les ingénieurs. Grâce à la description du projet, nous avons alors identifié six exigences pour la sélection d'une potentielle technique automatisant cette démarche :

(exi_i) technique automatique (ou au moins automatisable) ;

(exi_{ii}) technique d'analyse statique (ne nécessitant pas d'exécution du logiciel cible) ;

(exi_{iii}) technique supportant le passage à l'échelle, *i.e.,* supportant des logiciels de taille supérieure à la centaine de kLoC ;

(exi_{iv}) technique applicable pour la reconnaissance d'une architecture à composants sur un logiciel avec une architecture différente (*e.g.,* modules) ;

(exi_{v}) technique adaptable au projet ;

(exi_{vi}) technique pour logiciel sans base de données.

Le chapitre 3 étudie ensuite l'état de l'art de l'évolution logicielle afin d'identifier les concepts liés à la problématique de l'entreprise ainsi que les techniques existantes pouvant répondre aux exigences identifiées précédemment. Premièrement, ce chapitre définit précisément les différents types d'évolution logicielle et montre qu'il existe des recouvrements : la majorité d'entre eux commencent par une reconstruction de l'architecture. Lorsque l'architecture cible est différente de l'existante, des techniques comparables à celles de reconstruction d'architecture peuvent être mises en œuvre pour identifier les éléments de code à allouer à la nouvelle architecture. Deuxièmement, ce chapitre identifie trois techniques parmi celles de reconstruction d'architecture qui répondent aux exigences du projet de l'entreprise définies dans le chapitre 2. Ces trois techniques, le *Reflexion Model*, la recherche d'information et le regroupement, sont ensuite décrites.

Le chapitre 4 présente une description détaillée de la démarche informelle utilisée par les ingénieurs pour réaliser la rénovation d'architecture logicielle du proejt de l'entreprise. Nous avons proposé une structuration et généralisation de cette démarche en nous basant sur une des trois techniques identifiées plus tôt, le *Reflexion Model*. La démarche que nous proposons est constituée de deux phases : une première phase visant l'allocation d'éléments de code (*i.e.,* des *packages*) aux éléments de la nouvelle architecture (*i.e.,* composants composites) ; une deuxième phase visant le raffinage de l'allocation réalisée dans la première phase en allouant des éléments de code contenus dans les *packages* (*i.e.,* sous-programmes) aux éléments architecturaux contenus dans les composants composites (*i.e.,* composants atomiques).

Nous proposons de structurer la première phase en appliquant une instance du *Reflexion Model* pour allouer les *packages* aux composants composites, et de façon similaire, de structurer la deuxième phase en appliquant une instance du *Reflexion Model* par composant composite afin d'allouer les sous-programmes aux composants atomiques.

En nous basant sur la démarche de l'entreprise, nous avons également identifié cinq critères d'allocations utilisés pour allouer les éléments de code aux éléments de la nouvelle architecture :

CA_o Critère d'allocation par l'organisation des fichiers sources (*e.g.,* éléments de code contenus dans un dossier donné) ;

CA_c Critère d'allocation par contenance d'éléments (*e.g.,* les éléments de code contenant un type d'élément de code donné) ;

CA_n Critère d'allocation par les noms (*e.g.,* éléments de code respectant une convention de nommage donnée) ;

CA_d Critère d'allocation par les dépendances (*e.g.,* les éléments de code utilisant une variable ou un sous-programme donné) ;

CA_r Critère d'allocation par la résolution des conflits (*e.g.,* choix d'allocation pour les éléments de code alloués à plusieurs éléments architecturaux).

Grâce à l'utilisation du *Reflexion Model* dans notre démarche, nous montrons que cette technique est applicable sur le projet décrit dans cette thèse. Toutefois, malgré les critères d'allocation identifiés, cette démarche reste principalement manuelle.

Le chapitre 5 présente les expérimentations que nous avons réalisées pour essayer d'automatiser notre démarche structurée. Ces expériences passent par l'application de la recherche d'information et du regroupement ainsi que par la vérification automatique de nos critères d'allocation. Ces techniques et nos critères d'allocation sont alors évalués pour vérifier si leur application fournit des résultats d'une qualité similaire à ceux obtenus manuellement par les ingénieurs. Nous montrons que la vérification automatique de nos critères d'allocation est la seule à fournir des résultats respectant les seuils de satisfaction définis par les ingénieurs de l'entreprise au contraire de la recherche d'information et du regroupement. En effet, les techniques existantes de reconstruction d'architecture fournissent, de manière indépendante, des résultats de bonne qualité dès lors qu'un seul type de convention suffit pour décrire l'architecture elle-même. Or dans le cas du projet de Thales Air Systems, les conventions nécessaires à la description de la nouvelle architecture du logiciel sont multiples et de différentes natures (*e.g.,* critères d'allocations CA_n et CA_c). De plus, même lorsque les conventions sont de mêmes natures (*e.g.,* conventions de nommage), la qualité des résultats obtenus par l'application des techniques existantes est dépendante du respect de ces conventions. Comme plusieurs équipes

ont travaillé sur le premier développement du logiciel, ces conventions n'ont pas été strictement respectées. En outre, la recherche a montré que des conventions de même nature sont rarement respectées strictement et notamment dans le cas de convention de nommage [Furnas 1987].

Finalement, le chapitre 6 présente un ensemble d'activités ayant été réalisées ou planifiées sur les projets du plan de maintenance de Thales Air Systems. Une activité est une tâche qui vise la compréhension, l'analyse ou la réutilisation du code source d'un logiciel dans le but de rénover son architecture. Ces activités ont été obtenues à partir d'entrevues avec plusieurs ingénieurs ayant travaillé sur le projet décrit dans cette thèse ou ayant pris part à la planification des futurs projets de rénovation d'architecture. Nous proposons alors une généralisation et une structuration de ces activités sous la forme d'opérateurs (*e.g.,* opérateur de **Convention lexicale** pour retrouver les éléments de code dont le code source contient une chaîne de caractère donnée). Ces opérateurs ont pour but de faciliter la réplication et l'automatisation de la démarche structurée que nous avons proposée au chapitre 4. Pour cela, d'une part nous montrons que nos opérateurs respectent chacune des six exigences du projet de l'entreprise définies au chapitre 2. D'autre part, nous répliquons entièrement notre démarche structurée grâce à l'utilisation de nos opérateurs.

7.2 Contributions

Les principales contributions de cette thèse sont :

— la proposition d'une démarche structurée de rénovation d'architecture logicielle qui se base sur le *Reflexion Model* et sur un cas concret observé à Thales Air Systems ;

— l'expérimentation de deux techniques de reconstruction d'architecture (recherche d'information et regroupement) sur un projet de rénovation d'architecture logicielle dans l'industrie ;

— les conclusions sur cette expérimentation, qui montre que les résultats obtenus par l'application de ces deux techniques n'atteignent pas les seuils de satisfaction donnés par les ingénieurs de l'entreprise ;

— la définition d'un ensemble d'activités pour la compréhension, l'analyse ou la réutilisation du code source d'un logiciel dans le but de rénover son architecture ;

— la proposition d'un ensemble d'opérateurs basés sur ces activités et qui facilitent la réplication et l'automatisation de notre démarche structurée.

7.3 Perspectives et travaux futurs

Dans le chapitre 6, nous avons décrit la réplication de notre démarche structurée grâce à nos opérateurs. À travers cette description, nous avons noté que les opérateurs peuvent être utilisés suivant un patron particulier. En effet, plusieurs opérateurs ont été chaînés, en utilisant la sortie des uns comme entrée pour les autres. Par exemple, la sortie de l'opérateur de **Convention lexicale** a été utilisée comme entrée pour l'opérateur d'**Allocation d'éléments**. Un travail futur serait alors de définir des règles pour ce phénomène de "chaînage" des opérateurs (*e.g.,* les opérateurs pouvant être chaînés, l'ordre de chaînage, les couples d'opérateurs pour le chaînage).

Malgré la définition de nos opérateurs, nous n'avons pas pu implémenter la totalité de ces opérateurs dans un outil de qualité industriel. Cela a freiné leur utilisation sur le projet décrit dans cette thèse. Afin de pouvoir être utilisable sur les autres projets de l'entreprise, un travail futur serait d'implémenter la totalité de ces opérateurs dans un outil de qualité industriel. Ces implémentations "industrielles" de nos opérateurs permettraient alors de répliquer concrètement et de manière semi-automatique notre démarche structurée.

Enfin, la définition de notre démarche structurée et son outillage par nos opérateurs sont la première étape vers la réplication et l'automatisation de la rénovation d'architecture logicielle. Nos opérateurs pouvant être utilisés en dehors du cadre de cette démarche, leur application sur d'autres projets permettra de les compléter et d'étudier leurs utilisations. Cela permettra d'identifier, par exemple, les chaînages d'opérateurs, ceux qui sont les plus utilisés, les séquences d'utilisation des opérateurs ou les étapes du projet pendant lesquelles ils sont utilisés. Une telle étude nécessite un moyen de collecte de données tel qu'un outil d'enregistrement de l'utilisation des opérateurs. L'analyse de ces informations pourra ensuite aider à la définition d'autres démarches, à leur réplication et à leur automatisation.

Bibliographie

[Al-Msie'Deen 2013] Ra'Fat Ahmad Al-Msie'Deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier and Hamzeh Eyal-Salman. *Mining Features from the Object-Oriented Source Code of a Collection of Software Variants Using Formal Concept Analysis and Latent Semantic Indexing*. In The 25th International Conference on Software Engineering and Knowledge Engineering, page 8, United States, June 2013. Knowledge Systems Institute Graduate School.
pp. 28.

[Allier 2009] Simon Allier, Houari A Sahraoui and Salah Sadou. *Identifying components in object-oriented programs using dynamic analysis and clustering*. In Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research, pages 136–148. IBM Corp., 2009.
pp. 3 and 29.

[Allier 2010] Simon Allier, Houari Sahraoui, Salah Sadou and Stéphane Vaucher. *Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces*. Component-Based Software Engineering, pages 216–231, 2010.
pp. 29.

[Allier 2011] Simon Allier, Salah Sadou, Houari Sahraoui and Regis Fleurquin. *From object-oriented applications to component-oriented applications via component-oriented architecture*. In Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on, pages 214–223. IEEE, 2011.
pp. 3 and 29.

[Alnusair 2010] Awny Alnusair and Tian Zhao. *Component search and reuse : An ontology-based approach*. In Information Reuse and Integration (IRI), 2010 IEEE International Conference on, pages 258–261. IEEE, 2010.
pp. 31.

[Anquetil 1999] Nicolas Anquetil and Timothy Lethbridge. *Experiments with Clustering as a Software Remodularization Method*. In Proceedings of Working Conference on Reverse Engineering (WCRE'99), pages 235–255, 1999.
pp. 57.

[Anquetil 2009] Nicolas Anquetil and Timothy C Lethbridge. *Ten years later, experiments with clustering as a software remodularization method*. In Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, pages 7–7. IEEE, 2009.
pp. 29.

[Anquetil 2011] Nicolas Anquetil and Jannik Laval. *Legacy Software Restructuring : Analyzing a Concrete Case*. In Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11), pages 279–286, Oldenburg, Germany, 2011.
pp. 23.

[Arévalo 2004] Gabriela Arévalo, Frank Buchli and Oscar Nierstrasz. *Detecting Implicit Collaboration Patterns*. In Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering), pages 122–131. IEEE Computer Society Press, November 2004.
pp. 28.

[Azmeh 2008] Zeina Azmeh, Marianne Huchard, Chouki Tibermacine, Christelle Urtado and Sylvain Vauttier. *Wspab : A tool for automatic classification & selection of web services using formal concept analysis*. In on Web Services, 2008. ECOWS'08. IEEE Sixth European Conference, pages 31–40. IEEE, 2008.
pp. 28 and 29.

[Barr 2006] Michael Barr and Anthony Massa. Programming embedded systems : with c and gnu development tools. " O'Reilly Media, Inc.", 2006.
pp. 2.

[Bavota 2010] Gabriele Bavota, Andrea De Lucia, Andrian Marcus and Rocco Oliveto. *Software Re-Modularization Based on Structural and Semantic Metrics*. In Reverse Engineering, Working Conference on, pages 195–204, 2010.
pp. 31 and 36.

[Ben-Ari 2006] Mordechai Ben-Ari. Principles of concurrent and distributed programming. Pearson Education, 2006.
pp. 2.

[Bhatti 2012] Muhammad Usman Bhatti, Nicolas Anquetil, Marianne Huchard and Stéphane Ducasse. *A Catalog of Patterns for Concept Lattice Interpretation in Software Reengineering*. In Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE 2012), pages 118–24, 2012.
pp. 3, 28, and 32.

[Bisbal 1999] Jesús Bisbal, Deirdre Lawless, Bing Wu and Jane Grimson. *Legacy information systems : Issues and directions*. IEEE software, vol. 16, no. 5, pages 103–111, 1999.
pp. 2.

[Braga 2001] Regina MM Braga, Marta Mattoso and Cláudia ML Werner. *The use of mediation and ontology technologies for software component informa-*

tion retrieval. In ACM SIGSOFT Software Engineering Notes, volume 26, pages 19–28. ACM, 2001.
pp. 31.

[Breivold 2008] Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land and Stig Larsson. *Using dependency model to support software architecture evolution*. In Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on, pages 82–91. IEEE, 2008.
pp. 29.

[Canfora 2000] Gerardo Canfora, Aniello Cimitile and Palazzo Bosco Lucarelli. *Software maintenance*. Handbook of Software Engineering and Knowledge Engineering, vol. 1, pages 91–120, 2000.
pp. 23.

[Chapin 2001] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil and Wui-Gee Than. *Types of software evolution and software maintenance*. Journal of software maintenance and evolution, vol. 13, no. 1, pages 3–30, 2001.
pp. vii, 21, and 22.

[Chikofsky 1990] Elliot Chikofsky and James Cross II. *Reverse Engineering and Design Recovery : A Taxonomy*. IEEE Software, vol. 7, no. 1, pages 13–17, January 1990.
pp. vii, 3, 22, 23, 24, and 25.

[Chowdhury 2010] Gobinda G Chowdhury. Introduction to modern information retrieval. Facet publishing, 2010.
pp. 36.

[Christl 2006] Andreas Christl and Margaret-Anne Storey. *Semi-automated mapping for the reflexion method*. PhD thesis, Citeseer, 2006.
pp. 25 and 26.

[Cimitile 1995] A. Cimitile and G. Visaggio. *Software Salvaging and the Call Dominance Tree*. Journal of Systems and Software, vol. 28, pages 117–127, 1995.
pp. 29.

[Constantinou 2015] Eleni Constantinou, Athanasios Naskos, George Kakarontzas and Ioannis Stamelos. *Extracting reusable components : A semi-automated approach for complex structures*. Information Processing Letters, vol. 115, no. 3, pages 414–417, 2015.
pp. 26.

[Demeyer 2001] Serge Demeyer, Sander Tichelaar and Stéphane Ducasse. *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Rapport technique,

University of Bern, 2001.
pp. 62.

[Ducasse 2000] Stéphane Ducasse, Michele Lanza and Sander Tichelaar. *Moose : an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, CoSET '00, June 2000.
pp. 62.

[Ducasse 2007] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen and Ilham Alloui. *Package Surface Blueprints : Visually Supporting the Understanding of Package Relationships*. In ICSM'07 : Proceedings of the IEEE International Conference on Software Maintenance, pages 94–103, 2007.
pp. 32.

[Ducasse 2009] Stéphane Ducasse and Damien Pollet. *Software Architecture Reconstruction : A Process-Oriented Taxonomy*. IEEE Transactions on Software Engineering, vol. 35, no. 4, pages 573–591, July 2009.
pp. 2, 26, 27, and 57.

[Fahmy 2000] Hoda Fahmy and Richard C. Holt. *Software architecture transformations*. In Software Maintenance, 2000. Proceedings. International Conference on, pages 88–96. IEEE, 2000.
pp. 3 and 25.

[Fiutem 1999] R. Fiutem, G. Antoniol, P. Tonella and E. Merlo. *ART : an Architectural Reverse Engineering Environment*. Journal of Software Maintenance : Research and Practice, vol. 11, no. 5, pages 339–364, 1999.
pp. 28 and 31.

[Furnas 1987] G. W. Furnas, T. K. Landauer, L. M. Gomez and S. T. Dumais. *The Vocabulary Problem in Human-system Communication*. Commun. ACM, vol. 30, no. 11, pages 964–971, November 1987.
pp. 78 and 114.

[Garcia 2013] Joshua Garcia, Igor Ivkovic and Nenad Medvidovic. *A comparative analysis of software architecture recovery techniques*. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pages 486–496. IEEE Press, 2013.
pp. 29 and 32.

[Garlan 2000] David Garlan, Robert T. Monroe and David Wile. *Acme : Architectural Description of Component-Based Systems*. In Gary T. Leavens and Murali Sitaraman, editeurs, Foundations of Component-Based Systems, chapitre 3, pages 47–67. Cambridge University Press, New York,

NY, USA, 2000.
pp. 101.

[Gerardo 2007] Canfora Gerardo and Di Penta Massimiliano. *New Frontiers of Reverse Engineering*. In FOSE '07 : 2007 Future of Software Engineering, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.
pp. 26.

[Glass 2001] Robert L Glass. *Frequently forgotten fundamental facts about software engineering*. IEEE software, vol. 18, no. 3, pages 112–111, 2001.
pp. 2.

[Govin 2015a] Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Measuring the progress of an Industrial Reverse Engineering Process*. In BENEVOL'15, Lille, France, December 2015.
pp. 6.

[Govin 2015b] Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Reverse Engineering Tool Requirements for Real Time Embedded Systems*. In SATToSE'15, Mons, Belgium, July 2015. short paper.
pp. 5.

[Govin 2016a] Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *How Can We Help Software Rearchitecting Efforts ? Study of an Industrial Case*. In Proceedings of the International Conference on Software Maintenance and Evolution, (Industrial Track), Raleigh, USA, October 2016.
pp. 6.

[Govin 2016b] Brice Govin, Nicolas Anquetil, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Clustering Techniques for Conceptual Cluster*. In IWST'16, Prague, Czech Republic, August 2016.
pp. 6.

[Govin 2017] Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Managing an Industrial Software Rearchitecting Project With Source Code Labelling*. In Complex Systems Design & Management, pages 61–74, Paris, France, December 2017. Springer.
pp. 6.

[Govin 2018] Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier and Stéphane Ducasse. *Large Legacy Software Re-Architecting Project : an Experience Report*. Journal of Software : Evolution and Pro-

cess, (MAJOR REVISION) 2018.
pp. 6.

[Greevy 2005] Orla Greevy and Stéphane Ducasse. *Correlating Features and Code Using A Compact Two-Sided Trace Analysis Approach*. In Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.
pp. 28.

[Guéhéneuc 2004] Yann-Gaël Guéhéneuc, Houari Sahraoui and Farouk Zaidi. *Fingerprinting Design Patterns*. In Working Conference on Reverse Engineering (WCRE'04), pages 172–181, Los Alamitos CA, 2004. IEEE Computer Society Press.
pp. 27 and 31.

[Guéhéneuc 2008] Yann-Gaël Guéhéneuc and Giuliano Antoniol. *Demima : A multilayered approach for design pattern identification*. IEEE Transactions on Software Engineering, vol. 34, no. 5, pages 667–684, 2008.
pp. 31.

[Guo 1999] Yanbing Guo, Atlee and Kazman. *A Software Architecture Reconstruction Method*. In Working Conference on Software Architecture (WICSA), pages 15–34, 1999.
pp. 28 and 31.

[Hinchey 2010] Mike Hinchey and Lorcan Coyle. *Evolving critical systems : A research agenda for computer-based systems*. In Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on, pages 430–435. IEEE, 2010.
pp. 2.

[Holt 1998] Richard Holt. *Structural Manipulations of Software Architecture Using Tarski Relational Algebra*. In Proceedings of WCRE '98, pages 210–219. IEEE Computer Society, 1998. ISBN : 0-8186-89-67-6.
pp. 27.

[IEEE 1998] IEEE, editeur. Ieee standard for software maintenance. Wiley-IEEE Computer Society Pr, oct 1998.
pp. 1.

[Kazman 1996] R. Kazman. *Tool support for Architecture Analysis and Design*, 1996. Proceedings of Workshop (ISAW-2) joint Sigsoft.
pp. 23.

[Kazman 1998] R. Kazman, S.G. Woods and S.J. Carriére. *Requirements for Integrating Software Architecture and Reengineering Models : CORUM II*. In Proceedings of WCRE '98, pages 154–163. IEEE Computer Society, 1998.

ISBN : 0-8186-89-67-6.
pp. vii, 3, 15, and 23.

[Kazman 2003] Rick Kazman, Liam O'Brien and Chris Verhoef. *Architecture Reconstruction Guidelines, Third Edition*. CMU/SEI-2002-TR-034, Carnegie Mellon University, Software Engineering Institute, November 2003.
pp. 23.

[Kazman 2005] Rick Kazman and Len Bass. *Categorizing Business Goals for Software Architectures*. Cmu/sei-2005-tr-021, Carnegie Mellon University, Software Engineering Institute, December 2005.
pp. 25.

[Khadka 2011] Ravi Khadka, Gijs Reijnders, Amir Saeidi, Slinger Jansen and Jurriaan Hage. *A method engineering based legacy to SOA migration method.* In Software Maintenance (ICSM), 2011 27th IEEE International Conference on, pages 163–172. IEEE, 2011.
pp. 3 and 29.

[Kitchenham 2002] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam and Jarrett Rosenberg. *Preliminary guidelines for empirical research in software engineering*. IEEE Trans. Softw. Eng., vol. 22, no. 8, pages 721–734, 2002.
pp. 7, 17, 19, 21, and 111.

[Knodel 2002] Jens Knodel. *Process models for the reconstruction of software architecture views*. PhD thesis, Universitätsbibliothek der Universität Stuttgart, 2002.
pp. 30.

[Koschke 2000] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.
pp. 25, 26, 29, 32, 37, 57, and 62.

[Koschke 2003] Rainer Koschke and Daniel Simon. *Hierarchical Reflexion Models*. In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), page 36. IEEE Computer Society, 2003.
pp. 30 and 32.

[Krikhaar 1999] Rene Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.
pp. 23.

[Krogstie 2015] John Krogstie. *Systems Development Process Improvement Using Principles from Organization Development*. Modern Techniques for Successful IT Project Management, pages 97–117, 2015.
pp. 1.

[Kruchten 1995] Philippe B. Kruchten. *The 4+1 View Model of Architecture*. IEEE Software, vol. 12, no. 6, pages 42–50, November 1995.
pp. 39.

[Kuhn 2006] Adrian Kuhn. Semantic clustering : Making use of linguistic information to reveal concepts in source code. Master's thesis, University of Bern, March 2006.
pp. 61.

[Lundberg 2003] Jonas Lundberg and Welf Löwe. *Architecture Recovery by Semi-Automatic Component Identification*. Electronic Notes in Theoretical Computer Science, vol. 82, no. 5, 2003.
pp. 29 and 32.

[Lungu 2006] Mircea Lungu and Michele Lanza. *Softwarenaut : Exploring Hierarchical System Decompositions*. In Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering), pages 351–354, Los Alamitos CA, 2006. IEEE Computer Society Press.
pp. 27.

[Malavolta 2013] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione and Antony Tang. *What industry needs from architectural languages : A survey*. IEEE Transactions on Software Engineering, vol. 39, no. 6, pages 869–891, 2013.
pp. 39.

[Maletic 2000] Jonathan I. Maletic and Andrian Marcus. *Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding*. In Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000), pages 46–53, November 2000.
pp. 36.

[Maletic 2002] Jonathan Maletic, Michael Collard and Andrian Marcus. *Source Code Files as Structured Documents*. In Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002), pages 289–292. IEEE, June 2002.
pp. 36.

[Malhotra 2016] Ruchika Malhotra and Anuradha Chug. *Software Maintainability : Systematic Literature Review and Current Trends*. International Journal of Software Engineering and Knowledge Engineering, vol. 26, no. 08, pages 1221–1253, 2016.
pp. 1.

[Mancoridis 1998] Spiros Mancoridis and Brian S. Mitchell. *Using Automatic Clustering to produce High-Level System Organizations of Source Code*.

In Proceedings of IWPC '98 (International Workshop on Program Comprehension). IEEE Computer Society Press, 1998.
pp. 29 and 32.

[Maqbool 2007] Onaiza Maqbool and Haroon Babri. *s*. IEEE Trans. Softw. Eng., vol. 33, no. 11, pages 759–780, 2007.
pp. 29 and 32.

[Marcus 2003] Andrian Marcus and Jonathan Maletic. *Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing*. In Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pages 125–135, May 2003.
pp. 27.

[Marcus 2004] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich and Jonathan Maletic. *An Information Retrieval Approach to Concept Location in Source Code*. In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004), pages 214–223, November 2004.
pp. 31.

[Marcus 2005] Andrian Marcus and Václav Rajlich. *Panel : Identification of Concepts, Features, and Concerns in Srource Code*. In Proceedings of the 21st International Conference on Software Maintenance (ICSM'05), 2005.
pp. 31.

[Mendonça 2001] Nabor C. Mendonça and Jeff Kramer. *An Approach for Recovering Distributed System Architectures*. Automated Software Engineering, vol. 8, no. 3-4, pages 311–354, 2001.
pp. 31.

[Mens 2006] Kim Mens, Andy Kellens, Frédéric Pluquet and Roel Wuyts. *Co-evolving Code and Design with Intensional Views — A Case Study*. Journal of Computer Languages, Systems and Structures, vol. 32, no. 2, pages 140–156, 2006.
pp. 27.

[Mens 2008] Tom Mens. *Introduction and roadmap : History and challenges of software evolution*. In Software evolution, pages 1–11. Springer, 2008.
pp. 22 and 23.

[Moo] *Moose*. http://www.moosetechnology.org/.
pp. 48.

[Murphy 1995] Gail Murphy, David Notkin and Kevin Sullivan. *Software Reflexion Models : Bridging the gap between Source and High-Level Models*. In Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 18–28. ACM Press, 1995.
pp. vii, 3, 25, 26, 27, 30, 33, and 34.

[Murphy 1997] Gail C. Murphy and David Notkin. *Reengineering with Reflexion Models : A Case Study*. IEEE Computer, vol. 8, pages 29–36, 1997.
pp. 30.

[Nosek 1990] J. T. Nosek and P. Palvia. *Software Maintenance Management : changes in the last decade*. Software Maintenance : Research and Practice, vol. 2, no. 3, pages 157–174, 1990.
pp. 1.

[O'Brien 2002] Liam O'Brien, Christoph Stoermer and Chris Verhoef. *Software Architecture Reconstruction : Practice Needs and Current Approaches*. Rapport technique CMU/SEI-2002-TR-024, Carnegie Mellon University, August 2002.
pp. 26.

[Oquendo 2016] Flavio Oquendo. *Software Architecture Challenges and Emerging Research in Software-intensive Systems-of-Systems*. In Proceedings of the 10th European Conference on Software Architecture (ECSA 2016), numéro 9839, pages 3–21. Springer, 2016.
pp. 2.

[Passos 2010] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz and Nabor das Chagas Mendonca. *Static architecture-conformance checking : An illustrative overview*. IEEE software, vol. 27, no. 5, pages 82–89, 2010.
pp. 101.

[Perry 1992] Dewayne E. Perry and Alexander L. Wolf. *Foundations for the Study of Software Architecture*. ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pages 40–52, October 1992.
pp. 2 and 39.

[Pinzger 2005] Martin Pinzger. *ArchView — Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, 2005.
pp. 27.

[Poshyvanyk 2006] Denys Poshyvanyk, Andrian Marcus, Giuliano Antoniol and Vaclav Rajlich. *Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification*. In Proceedings of the 2nd international conference on program comprehension (ICPC). ACM Press, 2006.
pp. 31 and 36.

[Poshyvanyk 2007] Denys Poshyvanyk and Andrian Marcus. *Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code*. In ICPC '07 : Proceedings of the 15th IEEE International Conference on Program Comprehension, pages 37–48, Washington, DC,

USA, 2007. IEEE Computer Society.
pp. 27.

[Sangal 2005] Neeraj Sangal, Ev Jordan, Vineet Sinha and Daniel Jackson. *Using Dependency Models to Manage Complex Software Architecture*. In Proceedings of OOPSLA'05, pages 167–176, 2005.
pp. 29.

[Santos 2015a] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stéphane Ducasse and Marco Túlio Valente. *OrionPlanning : Improving Modularization and Checking Consistency on Software Architecture*. In 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015) – Tool track, pages 190–194, 2015.
pp. 101.

[Santos 2015b] Gustavo Santos, Nicolas Anquetil, Anne Etien, Stéphane Ducasse and Marco Túlio Valente. *System Specific, Source Code Transformations*. In 31st IEEE International Conference on Software Maintenance and Evolution, pages 221–230, 2015.
pp. 25.

[Sartipi 2003a] Kamran Sartipi. *Software Architecture Recovery based on Pattern Matching*. PhD thesis, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2003.
pp. 31.

[Sartipi 2003b] Kamran Sartipi and Kostas Kontogiannis. *On modeling software architecture recovery as graph matching*. In Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, pages 224–234. IEEE, 2003.
pp. 28.

[Seriai 2014] Abderrahmane Seriai, Salah Sadou and Houari A Sahraoui. *Enactment of components extracted from an object-oriented application*. In European Conference on Software Architecture, pages 234–249. Springer, 2014.
pp. 28.

[Siff 1999] Michael Siff and Thomas Reps. *Identifying Modules via Concept Analysis*. Transactions on Software Engineering, vol. 25, no. 6, pages 749–768, November 1999.
pp. 32.

[Sim 1999] Susan Elliott Sim, Charles L.A. Clarke, Richard C. Holt and Anthony M. Cox. *Browsing and Searching Software Architectures*. In International Conference on Software Maintenant (ICSM), pages 381–391.

IEEE CS, 1999.
pp. 27.

[Sparck Jones 1972] Karen Sparck Jones. *A statistical interpretation of term specificity and its application in retrieval*. Journal of documentation, vol. 28, no. 1, pages 11–21, 1972.
pp. 61.

[Sugumaran 2003] Vijayan Sugumaran and Veda C Storey. *A semantic-based approach to component retrieval*. ACM SIGMIS Database, vol. 34, no. 3, pages 8–24, 2003.
pp. 31.

[Sullivan 2001] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai and Ben Hallen. *The Structure and Value of Modularity in SOftware Design*. In ESEC/FSE 2001, 2001.
pp. 29.

[Tarski 1941] Alfred Tarski. *On the calculus of relations*. The Journal of Symbolic Logic, vol. 6, no. 3, pages 73–89, 1941.
pp. 27.

[Taylor 2009] R. N. Taylor, N. Medvidovic and E. M. Dashofy. Software architecture : Foundations, theory, and practice. Wiley, 2009.
pp. 28.

[Terra 2012] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki and Roberto S. Bigonha. *Recommending Refactorings to Reverse Software Architecture Erosion*. In 16th European Conference on Software Maintenance and Reengineering, Early Research Achievements Track, pages 335–340, 2012.
pp. 101.

[Tilley 2003] Thomas Tilley, Richard Cole, Peter Becker and Peter Eklund. *A Survey of Formal Concept Analysis Support for Software Engineering Activities*. In Gerd Stumme, editeur, Proceedings of ICFCA '03 (1st International Conference on Formal Concept Analysis). Springer-Verlag, February 2003.
pp. 28.

[Tsantalis 2011] Nikolaos Tsantalis and Alexander Chatzigeorgiou. *Identification of extract method refactoring opportunities for the decomposition of methods*. Journal of Systems and Software, vol. 84, no. 10, pages 1757–1782, 2011.
pp. 25.

[v. Rijsbergen 1979] C. v. Rijsbergen. Information retrieval. Butterworths, London, 1979.
pp. 36.

[Yan 2004] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich and Rick Kazman. *DiscoTect : A System for Discovering Architectures from Running Systems*. In International Conference on Software Engineering (ICSE), pages 470–479, 2004.
pp. 3 and 28.

[Yeh 1997] A.S. Yeh, D.R. Harris and M.P. Chase. *Manipulating Recovered Software Architecture Views*. In Proceedings of International Conference Software Engineering (ICSE'97), 1997.
pp. 31.

[Zhang 2010a] Huaxi Yulin Zhang, Christelle Urtado and Sylvain Vauttier. *Architecture-centric component-based development needs a three-level ADL*. In European Conference on Software Architecture, pages 295–310. Springer, 2010.
pp. 3 and 25.

[Zhang 2010b] Huaxi Yulin Zhang, Christelle Urtado and Sylvain Vauttier. *Architecture-centric development and evolution processes for component-based software*. In Proc. of 22nd SEKE Conf., Redwood City, USA (July 2010), 2010.
pp. 25.

[Zheng 2012] Yongjie Zheng and Richard N Taylor. *Enhancing architecture-implementation conformance with change management and support for behavioral mapping*. In Proceedings of the 34th International Conference on Software Engineering, pages 628–638. IEEE Press, 2012.
pp. 25.