République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la recherche Scientifique
وزارة التعليم العالي و البحث العلمي

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني لتكوين في الإعلام الآلي سابقا)
Ecole nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

Mémoire de fin d'études

Pour l'obtention du diplôme d'Ingénieur d'État en Informatique

Option : Systèmes Informatiques et Logiciels

Approche générique pour la migration de la partie client d'application web en utilisant l'ingénierie dirigée par les modèles : Application à la migration de GWT vers Angular

Réalisé par :
M. BEGOUG Mahi

Encadré par :
M. SERIAI Abdelhak-Djamel (LIRMM)
Mme. ZELLAGUI Soumia (ESI)

Promotion : 2020/2021

Remerciements

Tout d'abord, Je remercie **Allah** le tout puissant, le miséricordieux qui sans sa bénédiction rien de tout cela n'aurait été possible.

Je tiens à exprimer ma profonde reconnaissance envers **Dr. Seriai Abdelhak-Djamel** et **Dr. Zellagui Soumia**, mes encadrants, pour leurs précieux conseils, leurs critiques pertinentes, leur patience, leur encouragement et leur disponibilité.

Je suis très reconnaissant aux **enseignants** de l'**École nationale Supérieure d'Informatique** pour la vertu de la formation qu'ils donnent et qui se concrétisent aujourd'hui et demain, au terme de notre parcours par ce projet. Je suis également très heureux d'être un des étudiants de cette école.

Je ne saurais terminer sans exprimer mes remerciements les plus chaleureux à **mes parents** et **mes frères** dont l'encouragement et le soutien continu m'ont permis de mener ce travail à son terme.

Résumé

Ces dernières années, la nécessité de faire évoluer les applications web existantes est devenue l'un des problèmes majeurs rencontrés par les entreprises. En réalité, les applications entreprises connaissent une augmentation d'exigences et des besoins relatifs aux différentes demandes de ces clients ce qui les rends plus volumineuses, complexes et difficiles à maintenir

Parmi les défis, la partie client de leurs applications web ne bénéficie pas des avantages des nouvelles technologies web qui améliore l'interaction et la qualité d'interface graphique. Dans la même optique, ces nouveaux framework ont apporté des nouveaux langages de programmation différents et d'organisation de code source totalement modulaire. En effet, les entreprises prédisent que le coût de redévelopper cette partie peut être coûteux en termes du temps et disponibilité d'experts de domaine. D'autre part, les méthodes d'ingénierie dirigées par les modèles ont prouvé leur productivité dans ce type d'évolution et de modernisation, mais elles ne deviennent productives que lorsqu'elles sont automatisées à l'aide d'outils adéquats.

Dans le cadre de ce projet de fin d'étude, nous proposons une approche générique pour la migration des applications web, particulièrement pour leur côté client, en utilisant l'ingénierie dirigée par les modèles. Pour valider sa validité, nous choisissons un cas d'étude dont nous voulons migrer une application développée en GWT (Google Web Toolkit) vers une application Angular.

Mots-clés : Migration Logicielle, Ingénierie dirigées par les modèles, Évolution des applications web dirigée par les modèles, Réingénierie, Rétro-ingénierie, Transformation, Analyse statique, Génération du code, Modernisation dirigée par l'architecture (ADM), Modèle de découverte des connaissances (KDM), modèle d'arbre syntaxique abstrait (ASTM)

Abstract

In recent years, the need to evolve existing web applications has become one of the major problems faced by companies. In fact, business applications are experiencing an increase in requirements and needs relating to the various demands of these customers, which makes them more voluminous, complex and difficult to maintain.

Among the challenges, the client part of their web applications does not benefit from the advantages of new web technologies which improve the interaction and the quality of the graphical interface. In the same vein, these new frameworks have brought new different programming languages and completely modular source code organization. Indeed, companies predict that the cost of redeveloping this part can be costly in terms of time and availability of domain experts. On the other hand, model-driven engineering methods have proven their productivity in this kind of evolution and modernization, but they only become productive when they are automated with the right tools.

As part of this graduation project, we are proposing a generic approach for migrating web applications, particularly for their client side, using model-driven engineering. To validate its validity, we choose a case study from which we want to migrate an application developed in GWT (Google Web Toolkit) to an Angular application.

Key-words : Software Migration, Model driven engineering, Model driven web application evolution, Reengineering, Reverse engineering, Transformation, Static Analysis, Code Generation, Architecture Driven Modernization (ADM), Knowledge Discovery Model (KDM), Abstract Syntax Rree Model (ASTM)

TABLE DES MATIÈRES

Liste des Figures ix

Liste des Tables x

Introduction générale 1

Liste des sigles et acronymes 1

I Etat d'art 6

1 Évolution des logiciels 7

1 Introduction 8
2 Maintenance Logicielle 8
3 Réingénierie Logicielle 10
4 Migration Logicielle 11
5 Conclusion 13

2 Ingénierie Dirigée par les Modèles (IDM) 14

1 Introduction 15
2 Ingénierie Dirigée par les Modèles (IDM) 15
3 Transformation des modèles 20
4 Défis de l'IDM 23
5 Conclusion 26

3 Rétro-Ingénierie dirigée par les modèles (Model Driven Reverse Engineering) 27

1 Introduction 28

2 Définition . . . 28
3 Processus de Base de Model Driven Reverse Engineering . . . 28
4 L'initiative de Modernisation Dirigée par l'Architecture et Standardisation des modèles pour MDRE . . . 29
5 Conclusion . . . 37

4 Travaux connexes **38**
1 Introduction . . . 40
2 Le processus des approches de migration . . . 42
3 Application des critères de classification sur les approches existantes de migration des interfaces graphiques . . . 46
4 Conclusion . . . 50

II Contribution **52**

5 Approche proposée **53**
1 Introduction . . . 55
2 Vue globale de l'approche proposée . . . 55
3 Cas d'étude : Migration de la partie client d'une application Web GWT vers une application Web en Angular . . . 59
4 Conception . . . 88
5 Conclusion . . . 101

6 Réalisation, tests, résultats et verrous techniques **102**
1 Introduction . . . 103
2 Outils utilisés . . . 103
3 Jeu de données des applications GWT . . . 107
4 Expérimentation des heuristiques et résultats . . . 111
5 Test bout-en-bout (e2e) . . . 113
6 Conclusion . . . 119

Conclusion et perspectives **120**

Annexes **130**

A Méta-modèle Java **131**
1 Principales entités du modèle Java . . . 131

TABLE DES FIGURES

1 Matrice décisionnelle des systèmes hérités (De Lucia, Fasolino et Pompelle 2001). . . . 3

2 Distribution des types de maintenance dans l'industrie (Christa et al. 2017) . . . 9

3 Modèle Conceptuel de la réingénierie (E. Chikofsky et J. Cross 1990) . . 10

4 Modèle en fer de cheval représentant le processus de migration (Al-Shara 2016) . . . 12

5 Les Méthodes de Transformations (Al-Shara 2016) . . . 13

6 Principaux concepts de métamodélisation (extrait de MOF 2.0 (Group January 2006)). . . . 16

7 Relations entre système, modèle, métamodèle et langage . . . 18

8 Pyramide de l'OMG(Diaw, Lbath et Coulette 2010) . . . 19

9 Pile de modélisation (Diaw, Lbath et Coulette 2010) . . . 20

10 La transformation de modèles basée sur les méta-modèles (Bezivin, Jouault et Paliès 2005) . . . 21

11 Transformation endogène et exogène . . . 22

12 Transformation horizontale et Verticale . . . 22

13 Classification des défis (Antonio et al. 2020) . . . 26

14 Processus de MDRE (Raibulet, Arcelli Fontana et Zanoni 2017) . . 29

15 Processus d'ADM (Santos et al. 2019) . . . 30

16 Architecture en couche de KDM . . . 32

17 Extrait 1 du paquet code (group 2012) . . . 33

18 partie du package de code (Martín Santibáñez, Durelli et Camargo 2015) . . . 34

19 Objectifs de MoDisco (Brunelière et al. 2014) . . . 34

20 Architecture de MoDisco (Brunelière et al. 2014) 35
21 Modèle Java généré par MoDisco (Brunelière et al. 2014) 36
22 Processus global de notre approche 58
23 Processus d'exécution d'un code GWT sur touts les navigateurs (Jaber 2007) 60
24 Une partie du client GWT 61
25 Architecture Angular en générale (Angular 2021a) 63
26 Processus de construction du méta-modèle GWT 63
27 Partie ajoutée au méta-modèle KDM 64
28 Partie 1 extraite d'éléments graphiques de GWT 65
29 Partie 2 extraite d'éléments graphiques de GWT 66
30 Example d'un modèle KDM sous format XMI d'une application GWT généré par Modisco 67
31 Example d'un modèle Java sous format XMI d'une application GWT généré par Modisco 68
32 Processus de transformation d'un modèle KDM vers GWT 69
33 Vue d'ensemble des variables de classe StudentDashBoard 70
34 La classe StudentDashBoard d'une application GWT 71
35 Modèle client sous format JSON 72
36 Exemple de déclaration de routes en Angular 73
37 Exemple du code de navigation vers une autre page 74
38 Les différents cas de type d'un gestionnaire d'événement GWT vue d'un modèle JAVA 75
39 Instance comme argument 77
40 Variable affectée comme argument 77
41 Invocation à une méthode comme argument 78
42 Cas d'une 78
43 Diagramme d'activité pour la traçabilité d' argument injectées dans la méthode *RootPanel.get().add(...)* 80
44 Développeur déclare dans ça méthode 82
45 Processus de détection des actions/attributs d'un élément graphique 82
46 Déroulement du processus de détection de l'élément parent dans chaque page 83
47 Code HTML d'élément Button *GWT* après la correspandance 85
48 Architecture générale de l'environnement de l'application généré 86
49 Processus de traitement d'un élément graphique 87
50 Architecture de notre solution pour la migration GWT vers Angular 88
51 Diagramme de classe du composant modelTransformationRunner 90

52 Diagramme de packages de composant navigationConstruction 92
53 Diagramme de classes du sous package « nextPages » représentant le patron de conception visiteur 6 94
54 Diagramme de classes de sous package « arguments » représentant le patron de conception visiteur 7 95
55 Diagramme de classes représentant le patron Stratégie 8 97
56 Diagramme de package du composant « angularGenerator » 98
57 Patron chaîne de responsabilité pour la génération du contenu d'un composant en Angular 99
58 Patron de méthode pour la configuration d'environnement d'application Angular généré 100

59 Logo Java 103
60 Logo ATL 104
61 Logo Angular Material 105
62 logo d'Eclipse 105
63 logo Eclipse Modeling Tools 105
64 logo Eclipse Modeling FrameWork 105
66 Logo Github 106
67 Logo Jira 107
68 Vue globale de la création du jeu de données 107
69 Échantillon du jeu de données 108
70 Distribution les types d'applications 109
71 Structure d'un projet GWT (JDN 2012) 111
72 Application Simple GWT mono-page `https://github.com/manolo/gwt-stockwatcher` 113
73 Extrait du modèle Java de l'application généré par Modisco 114
74 Extrait du modèle KDM de l'application généré par Modisco 114
75 Extrait du modèle GWT 115
76 Extrait 1 du modèle intermédiaire 116
77 Extrait 2 du modèle intermédiaire 116
78 Le Template de génération de l'application cible 117
79 Génération de l'application cible 118
80 Application générée en Angular 118

81 (Gabriel et Fabien 2021) 131
82 Entité ASTNode (Gabriel et Fabien 2021) 132
83 Entité NamedElement (Gabriel et Fabien 2021) 132
84 Entité Expression et ses descendantes (Gabriel et Fabien 2021) 133

85 Entité Statement et ses descendantes GABRIEL et FABIEN 2021 134

LISTE DES TABLEAUX

1 Les entrées des approches des migrations GUI 47
2 Processus des approches de migration GUI 48
3 Les sorties des approches de migration GUI 49
4 Validations des approches de migration GUI 50

5 Correspondance entre les éléments graphiques *GWT* et *Angular Material* 84
6 Participation de classes du package « nextPages » dans le patron visiteur 53 93
7 Participation de classes du package « arguments » dans le patron visiteur 54 93
8 Participation de classes du package « stratégies » 4.2.1 dans le patron stratégie 55 96
9 Participations des classes de package « iterators » dans le patron chaîne de responsabilité présenté dans 57 99
10 Participation des classes de package « angular » dans le patron de méthode 100

11 Nombre de Classes et lignes pour chaque application 110
12 Les Résultats de l'extraction 112

LISTE DES SIGLES ET ACRONYMES

OCL *Object Constraint Language*

IDM *Ingénierie Dirigée par les Modèles*

API *Application Programming Interface*

UML *Unified Modeling Language*

MOF *Meta Object Facility*

MDA *Model Driven Architecture*

ADM *Architecture Driven Modernization*

MDRE *Model Driven Reverse Engineering*

KDM *Knowledge Discovery Metamodel*

ASTM *Abstract Syntax Tree Metamodel*

SMM *Structured Metrics Metamodel*

PDM *Platform Description Model*

PIM *Platform Independent Model*

CIM *Computational Independent Model*

PSM *Platform Specific Model*

GUI *Graphical User Interface*

AST *Abstract Syntaxe Tree*

GWT *Google Web Toolkit*

SDK *Software Development Kit*

URL *Uniform Resource Locator*

AJAX *Asynchronous JavaScript and XML*

RPC *Remote Procedure Call*

HTTP *Hypertext Transfer Protocol*

DOM *Document Object Model*

SPA *Single Page Application*

JSON *JavaScript Object Notation*

INTRODUCTION GÉNÉRALE

Contexte

Les pays les plus développés investissent de plus en plus dans le développement des systèmes logiciels qui sont devenus fondamentaux dans l'exécution d'une grande variété de tâches quotidiennes. Afin d'obtenir un retour sur les coûts dépensés sur le développement, il est nécessaire que les systèmes logiciels puissent être utilisés pour une longue période de temps. La durée de vie moyenne des systèmes logiciels est supérieure à 10 ans avec un minimum de deux ans et un maximum de trente comme indiqué par Tamai et al (TAMAI et TORIMITSU 1992). Daniel et al (C, DANIEL et A 2003) croient qu'un système de plus de cinq ans appartient déjà à la catégorie des systèmes hérités.

Dedeke et al (DEDEKE 1995) définissent un système hérité comme *"un ensemble agrégé de solutions logicielles et matérielles dont les langages, les normes, les codes et les technologies appartiennent à une génération d'innovation antérieure"*. Également, Bennett et al (K. BENNETT 2012) décrivent les systèmes hérités comme *"des grands systèmes que nous ne savons pas gérer mais qui sont vitaux pour les organismes et entreprises. De plus, le logiciel hérité a été écrit il y a des années en utilisant des techniques obsolètes, mais il continue à faire un travail utile"*. Ces définitions partagent le fait qu'un système hérité est un ancien système qui reste en vigueur au sein d'une entreprise. Les caractéristiques des systèmes hérités sont résumées par Crotty (CROTTY et HORROCKS 2017) en : critiques pour l'entreprise, utilisés pour une longue période de temps, développées à l'aide de technologies obsolètes, avec une documentation médiocre si disponible, une structure dégradée et nécessitant beaucoup de temps pour les tâches de maintenance.

La dépendance croissante aux systèmes logiciels dans tous les domaines de la société nécessite des changements continus. Les facteurs qui initient les changements comprennent : des changements dans les besoins des utilisateurs, l'émergence de nouvelles technologies d'actualité, les changements dans les objectifs commerciaux et la nécessité pour les entreprises d'exploiter de nouvelles opportunités. Les changements progressifs des systèmes logiciels conduisent à la dégradation de leur qualité (LEHMAN et al. 1997).

Plusieurs solutions ont été proposées pour traiter les systèmes hérités lorsque leur dégradation de qualité dépasse un seuil critique ou lorsque le besoin d'adaptation à de nouvelles technologies ou environnements se manifeste. Ces solutions peuvent être résumées en (K. H. BENNETT, RAMAGE et MUNRO 1999) :

— Faire de la maintenance malgré le coût.

— Moderniser le système par la réingénierie, la migration ou la maintenance avec des coûts acceptables.

— Remplacer complètement le système.

Plusieurs modèles décisionnels ont été proposés afin de prendre une décision sur la meilleure solution, parmi celles susmentionnées, à considérer. Ils reposent sur des valeurs à la fois commerciales et techniques. Les valeurs commerciales concernent la satisfaction des utilisateurs. Les valeurs techniques sont des mesures calculées sur le système (par exemple, la taille, la complexité, etc). La sortie de ces modèles est tracée sur une matrice décisionnelle qui indique une solution recommandée tel que décrit dans la Figure 1.

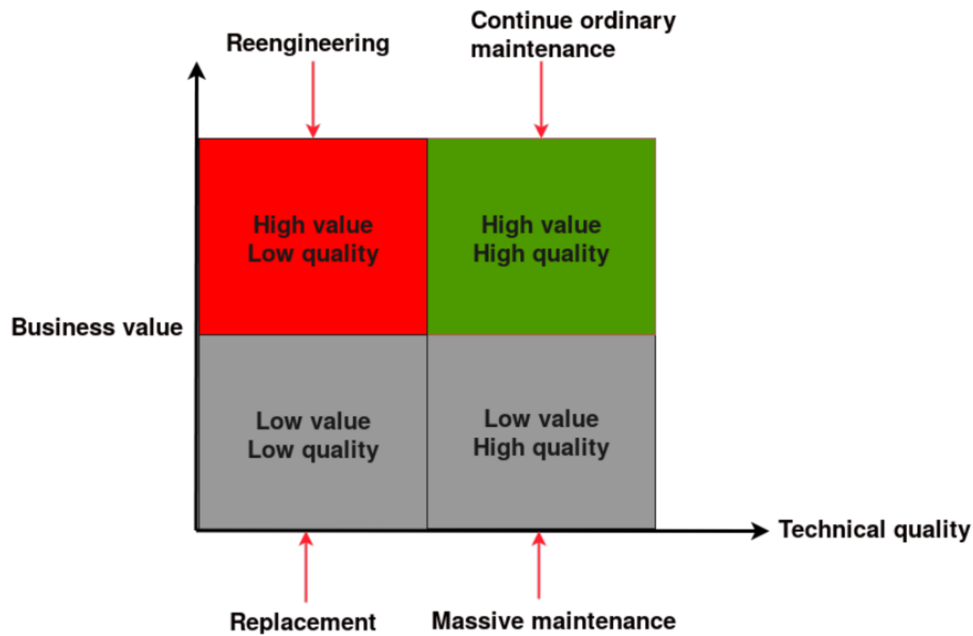


FIGURE 1 – Matrice décisionnelle des systèmes hérités (DE LUCIA, FASOLINO et POMPELLE 2001).

La première et la troisième solutions, maintenance massive et le remplacement du système, ne sont pas des solutions adaptées lorsque le coût et le temps sont à prendre en compte, car elles nécessitent un effort et un personnel considérables pour assurer l'accomplissement du système dans un délai modéré. Cependant, la réingénierie et la migration sont des exigences nécessaires pour la modernisation et l'évolution des systèmes logiciels. Par conséquent, dans ce mémoire, nous nous intéressons à la solution de modernisation.

Au fil des années, plusieurs options différentes ont vu le jour pour la modernisation des systèmes hérités. Chacune d'elles a rencontré un succès et une adoption variables. Les plus connus sont la réingénierie logicielle et la migration logicielle. La réingénierie logicielle représente le processus de génération de systèmes logiciels évolutifs (). En général, il inclut toutes les activités après la livraison du logiciel au consommateur dans le but d'améliorer les attributs de qualité. La migration logicielle est une variante de la réingénierie logicielle dans laquelle la transformation est entraînée par un changement technologique majeur(WAGNER 2014). Par exemple la migration d'une partie client développée en Java Server Page [1] d'une application Web vers une partie client développée en Angular.

1. Java Page Server (JSP) : est une technologie Java qui permet la génération de pages Web dynamiques.

Problématique et Motivation

Les technologies des applications Web ont évolué rapidement ces dernières années. *à* titre d'exemple, les frameworks Web côté client ont connu des évolutions conséquentes en 2018, il y avait deux versions majeures de Angular [2], trois versions majeures de ReactJS [3], quatre versions de Vue.js [4], et trois versions d'Ember.js [5]. Cette évolution pousse les organisations à mettre à jour constamment leurs applications pour éviter de se retrouver coincés dans les anciennes technologies (Abderrahmane Seriai 2019).

Actuellement, les entreprises qui ont développé leurs applications Web en utilisant des technologies héritées sont confrontées à des défis majeurs. D'un coté, le coût de la migration manuelle vers de nouvelles technologies est très élevé dépendant de la complexité de l'application. D'un autre coté, le redéveloppement de ces applications peut prendre un temps considérable. Entre autre, la qualité de l'application redéveloppée peut être non satisfaisante aux utilisateurs fidèles de l'application, surtout visuellement. Par conséquent, plusieurs interrogations surviennent :

1. Est qu'il est faisable de rendre la migration une activité automatique ?
2. Comment pouvons nous faire pour baisser le coût de la migration d'une application Web utilisant une technologie coté client obsolète ?

Dans ce travail, nous nous intéressons à la migration de la partie client des applications Web développée en GWT vers Angular. Ce choix est motivé par le fait qu'une grande partie de développeurs a abandonné la technologie GWT car ils pensent qu'elle est devenue une technologie obsolète (GWT 2020). De plus, Google a fait une migration de sa platforme AdSense [6] développés en GWT vers Angular (Dartlang p. d.). L'objectif dans ce mémoire consiste à :

— Faire une étude bibliographique liée à la problématique de migration d'interfaces graphiques des applications, en particulier, la migration de la partie client (Front-End) des applications Web.

— Exploiter les approches de migration des interfaces graphiques existantes pour proposer une approche générique de migration des applications spécialement côté client (partie visuelle) en utilisant l'ingénierie dirigée par les modèles.

— Étudier la faisabilité de l'approche proposée en prenant une application Web développée avec une technologie obsolète et de la migrer vers une application développée avec une nouvelle technologie moderne. Nous avons choisi la technologie Google Web Toolkit (GWT) comme la technologie source et Angular une technlogie ciblée.

Organisation du rapport

Afin de bien présenter notre travail, nous structurons notre mémoire en deux parties comme suit :

2. Verions d'Angular : `https://www.npmjs.com/package/@angular/cli?activeTab=versions`
3. Versions de ReactJS : `https://fr.reactjs.org/versions/`
4. Versions de Vue.js : `https://github.com/vuejs/vue/releases`
5. Versions d'Ember.js : `https://github.com/emberjs/ember.js/releases`
6. AdSense est le gestionnaire publicitaire de Google utilisant les sites web ou les vidéos YouTube comme support pour ses annonces.

La Première Partie

Elle représente l'étude bibliographique sur le domaine de migration des logiciels et l'exploitation de modèles au cœur de ce domaine. Elle est divisée en 4 chapitres afin d'exposer d'une manière descendante les notions du domaine d'étude pour faciliter la compréhension en partant du générale vers le détail, comme suit :

Chapitre 1 Évolution des logiciels : Dans ce premier chapitre, nous présentons les différentes notions de base liées aux : maintenance logicielle, réingénierie logicielle et migration logicielle.

Chapitre 2 Ingénierie Dirigée par les Modèles (IDM) : Dans ce chapitre, nous découvrons les principes fondamentaux d'IDM, la transformation des modèles et les défis confrontés dans ce domaine.

Chapitre 3 Rétro-Ingénierie dirigée par les modèles : Dans ce chapitre, nous définissons la notion de rétro-ingénierie dirigée par les modèles et nous présentons également son processus. Ensuite, nous découvrons l'initiative de Modernisation Dirigée par L'architecture.

Chapitre 4 Travaux connexes : Dans ce chapitre, nous présentons les approches existantes traitant la migration des interfaces utilisateur graphiques. L'objectif de ce chapitre est de synthétiser ces approches et de positionner notre problématique par rapport aux travaux existants.

La Seconde Partie

Elle représente l'approche proposée et le cas d'étude ainsi que les résultats obtenus et leur interprétation. Cette partie est également divisée en deux chapitres :

Chapitre 5 Approche proposée : Dans ce chapitre, nous introduisons la démarche et l'approche que nous avons proposée pour répondre à la problématique et aux objectifs de notre travail. Ensuite, nous étudierons le cas de migration des applications web développées en GWT vers Angular. Puis, nous présentons la conception de ce cas d'étude à travers des diagrammes représentant l'architecture de migration de GWT vers Angular ainsi que d'autres diagrammes UML (diagrammes de classes et packages).

Chapitre 6 Réalisation, Tests, Résultats : Dans ce dernier chapitre, nous avons un aperçu sur les outils qui nous ont permis de réaliser d'implémenter la solution pour le cas d'étude. Par la suite, nous démontons les résultats des expérimentations effectuées sur le cas d'étude. Après, nous effectuons un test de bout-en-bout pour montrer la faisabilité de la migration d'une application GWT vers Angular. Enfin, nous interprétons les résultats obtenus à travers les expérimentations et le test de bout-en-bout.

Première partie

Etat d'art

CHAPITRE 1

ÉVOLUTION DES LOGICIELS

Sommaire

1 Introduction 8
2 Maintenance Logicielle 8
2.1 Définition 8
2.2 Types de maintenance 8
2.3 Coût de maintenance 9
3 Réingénierie Logicielle 10
3.1 Définition 10
3.2 Modèle conceptuel de réingénierie logicielle 10
4 Migration Logicielle 11
4.1 Définition 11
4.2 Processus de migration des systèmes 11
5 Conclusion 13

1 Introduction

L'évolution du logiciel est le processus de révision des changements majeurs, y compris l'architecture du système pour diverses raisons au cours du cycle de vie du logiciel. De nombreuses études ont montré que la plupart des efforts dépensés dans les grands projets logiciels concernent l'évolution des systèmes logiciels (CHRISTA et al. 2017). Par conséquent, cette activité constitue un enjeu crucial qui nécessite des formalismes, outils, techniques et méthodologies adaptés.

Ce chapitre est consacré à la définition des différents concepts liés à la problématique de l'évolution logicielle : Maintenance, Réingénierie et Migration.

2 Maintenance Logicielle

2.1 Définition

La maintenance logicielle est définie comme "*le processus de modification d'un système ou d'un composant logiciel après livraison pour corriger les défauts, améliorer les performances ou d'autres attributs, ou s'adapter à un environnement modifié.*" (IEEE 1998)

Cette définition reflète l'opinion courante qui considère la maintenance logicielle comme une activité après livraison : elle commence lorsqu'un système est mis à la disposition du client ou de l'utilisateur et englobe toutes les activités qui maintiennent le système opérationnel et réponds aux besoins de l'utilisateur. Cette vue est bien résumée par les modèles classiques en cascade [1] du cycle de vie du logiciel, qui comprennent généralement une phase finale d'exploitation et de maintenance.

2.2 Types de maintenance

Il existe quatre types de maintenance (IEEE 1998) : corrective, perfective, d'urgence et adaptative. Ces différents types sont définis comme suit :

— *Maintenance corrective* : modification d'un produit logiciel effectuée après livraison pour corriger les défauts découverts

— *Maintenance perfective* : modification d'un produit logiciel effectuée après livraison pour en améliorer les performances.

— *Maintenance d'urgence* : maintenance corrective non programmée effectuée pour maintenir un système opérationnel.

— *Maintenance adaptative* : modification d'un produit logiciel effectuée après livraison pour conserver un programme informatique utilisable dans un environnement modifié ou évolutif.

1. les modèles classiques en cascade : un modèle linéaire de gestion de projet de développement qui divise les processus en phases de projet successives.

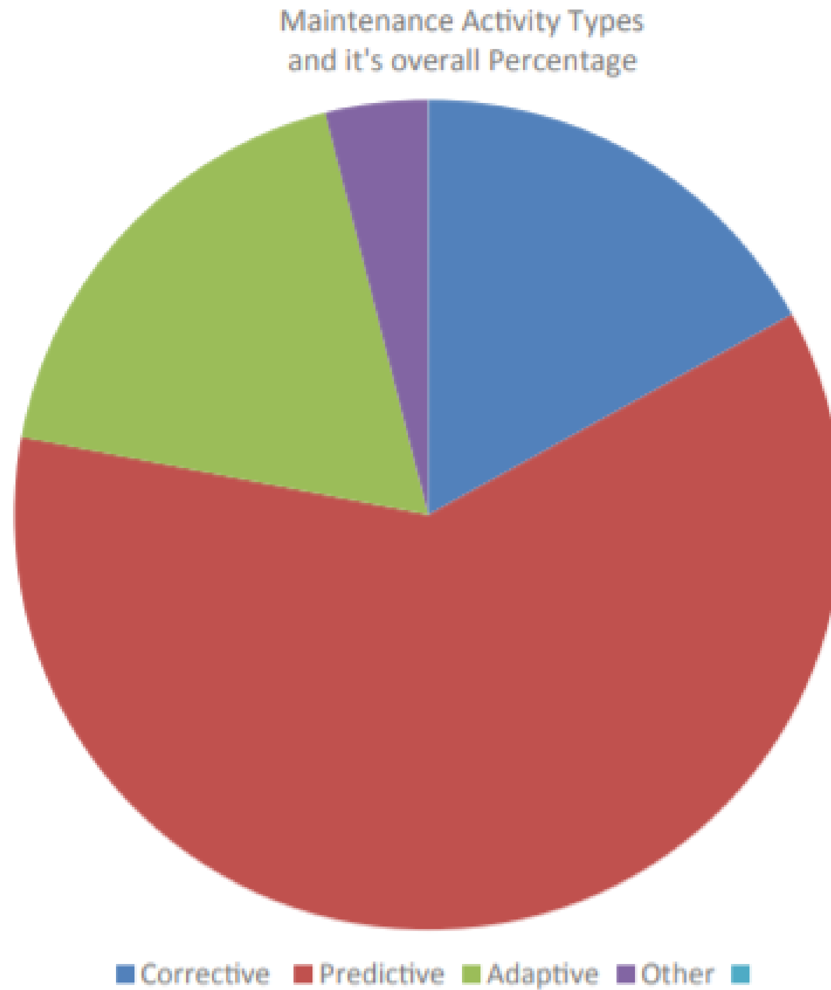


FIGURE 2 – Distribution des types de maintenance dans l'industrie (CHRISTA et al. 2017)

La Figure 2 illustre la répartition de l'effort de maintenance requis pour divers types d'activités de maintenance, comme expliqué ci-dessus. 75 % de l'effort de maintenance est consacré à la maintenance adaptative et perfective alors que 17 % de l'effort est consacré à la correction d'erreurs. Tandis que 3% est alloué au d'autre type de maintenance.

2.3 Coût de maintenance

La maintenance logicielle entraîne des coûts importants pendant le cycle de vie d'un logiciel. Près de 70% du temps et des ressources sont consacrés aux activités de maintenance (CHRISTA et al. 2017) . Parmi les facteurs qui influencent le coût de maintenance est la qualité du code. Un code mal écrit est difficile à comprendre et donc difficile à maintenir.

3 Réingénierie Logicielle

3.1 Définition

La réingénierie logicielle consiste à prendre un système existant et à en extraire des connaissances (sa structure interne ou son processus métier) et à le reconstituer sous une nouvelle forme à l'aide de nouvelles technologies pour améliorer sa qualité et l'adapter à de nouvelles technologies (E. J. Chikofsky et J. H. Cross 1990). Généralement, le processus de réingénierie comprends toutes les activités ultérieures à la livraison de logiciels qui visent à améliorer la compréhension du logiciel ainsi qu'à améliorer divers paramètres de qualité tels que la maintenabilité, la réutilisabilité et la complexité du système. Ce qui prolonge énormément la durée et le cycle de vie du logiciel (Wagner 2014).

Dans la littérature, la définition récurrente pour le terme réingénierie est celle de Chikofsky et Cross : *"la réingénierie logicielle, également connue sous le nom de rénovation et réclamation, est l'évaluation, l'inspection et l'altération d'un système donné pour le reconstruire dans une nouvelle forme et la mise en œuvre ultérieure de la nouvelle forme"* (E. Chikofsky et J. Cross 1990).

3.2 Modèle conceptuel de réingénierie logicielle

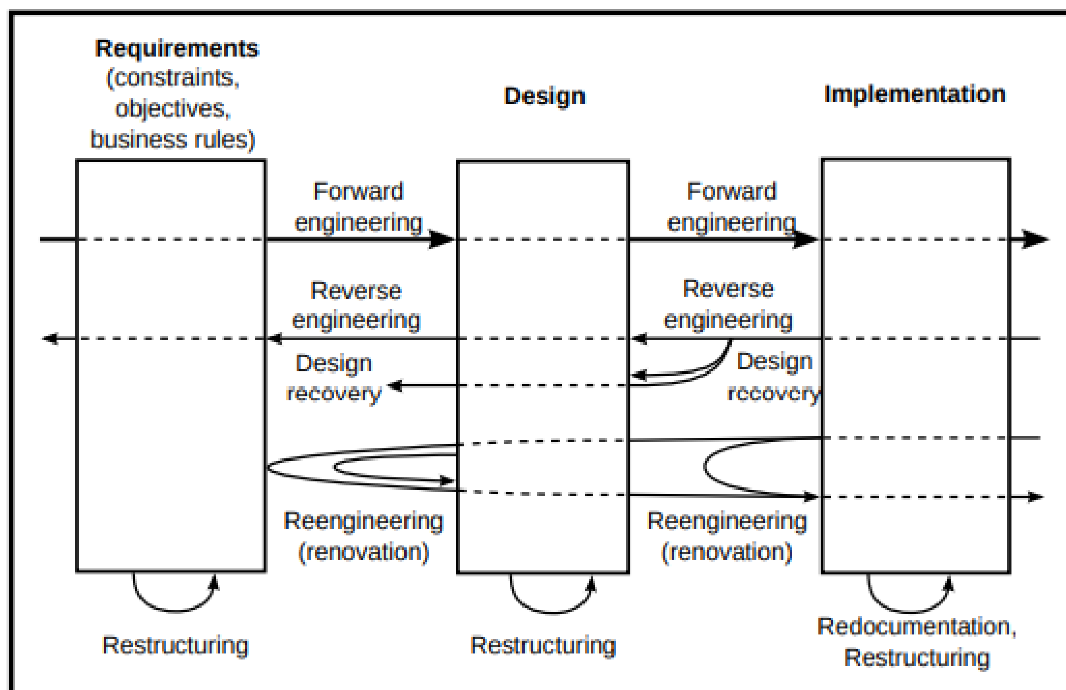


Figure 3 – Modèle Conceptuel de la réingénierie (E. Chikofsky et J. Cross 1990)

La Figure 3 montre le modèle conceptuel de la réingénierie logicielle. Le terme de l'ingénierie directe (Forward enginnering) représente le processus classique de développement de logiciels. Il débute par un niveau supérieur d'abstraction, celui de besoins et spécifications, jusqu'au un niveau inférieur représenté par l'implémentation.

Chikofsky et Cross ont défini d'autres termes de la famille *Re**. Chaque terme est lié à un ou plusieurs des trois niveaux d'abstraction. Ils couvrent les principaux niveaux : exigence (Requirements), conception (Design) et mise en œuvre (Implementation). Nous expliquons quelques termes utilisées dans le modèle conceptuel :

Redocumentation

La redocumentation est une activité de la phase de la maintenance logicielle dont l'objectif est de produire une explication textuelle simplifiée par rapport à certains éléments qui peuvent aider à comprendre le code source que nous avons changé. Par exemple : que fait le code, comment ce code fait ce qui sensé faire, quelles sont les entrées de ce code, quelles sont les formes de ses sorties, etc.

Reconstruction de conception (Design recovery)

La reconstruction de conception est une activité dont l'objectif consiste à établir d'extraire et identifier les différents composants d'un système logiciel et établir les relations entre ces composants. Par exemple, l'extraction de l'architecture "architecture recovery" est une des forme de la reconstruction de conception (Abdelhak Seriai et Chardigny 2010).

Restructuration (Restructuring)

La restructuration est la transformation d'une forme de représentation en une autre sans changer la fonctionnalité. Par exemple, le développeur peut ajouter des annotations qui comprennent l'introduction de directives de codage ou éliminer du code source dupliqué (Ouni et al. 2017). Cela ne change pas la sémantique. Ces ajustements ne sont pas visibles pour les parties prenantes. La restructuration s'effectue toujours à l'intérieur d'un niveau d'abstraction (Wagner 2014).

4 Migration Logicielle

4.1 Définition

La migration des systèmes est une variante de la réingénierie, Elle est définie comme suit : "*La migration permet de déplacer les anciens systèmes hérités vers de nouveaux environnements qui permettent aux systèmes d'information d'être facilement maintenus et adaptés aux nouvelles exigences commerciales, tout en conservant les fonctionnalités et les données des systèmes d'origine sans avoir à les redévelopper complètement*"(Bisbal et al. 1999).

4.2 Processus de migration des systèmes

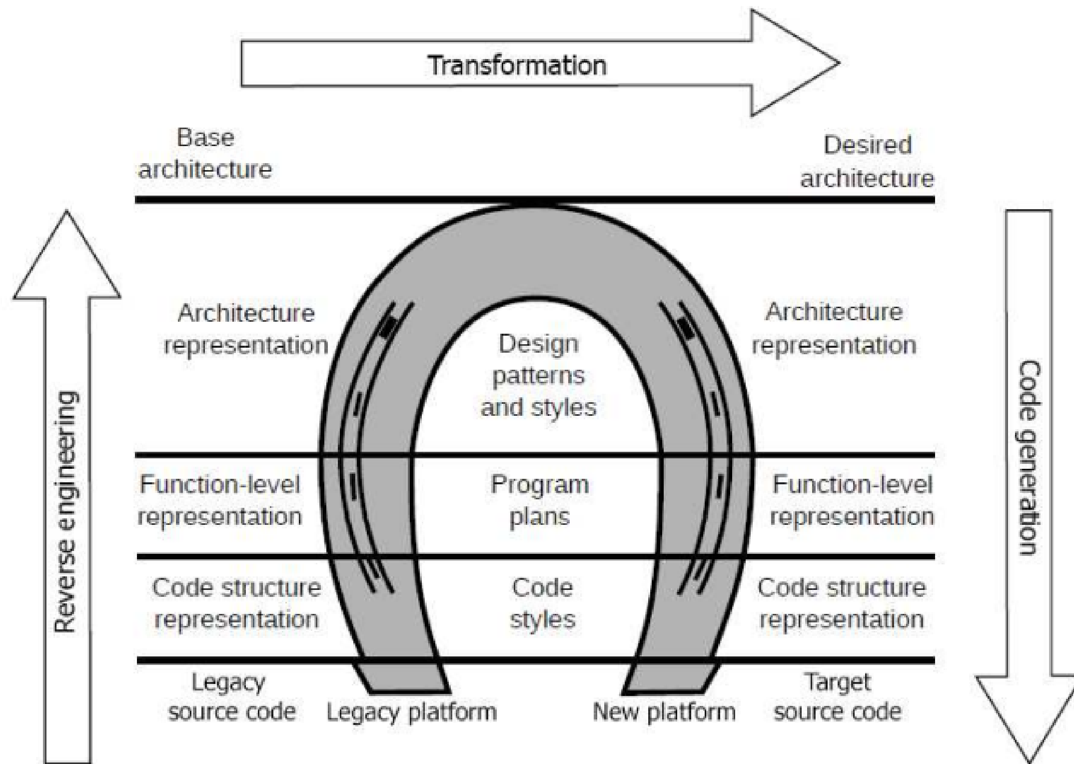


FIGURE 4 – Modèle en fer de cheval représentant le processus de migration (AL-SHARA 2016)

La Figure 4 représente le modèle en fer de cheval exprimant le processus de migration utilisée dans la littérature. Le processus se compose de trois parties essentielles :

1. Rétro-ingénierie (Reverse engineering) : cette étape est importante pour comprendre et obtenir une abstraction sur le logiciel hérité. Elle repose sur le principe qu'un système logiciel ne peut pas être modifié si ces différents artefacts ne sont pas clairs. Ce processus de compréhension peut avoir une routine semi-automatique(WAGNER 2014). Comme exemple de Rétro-ingénierie, nous pouvons donner l'extraction de l'architecture d'une application Web développée en PHP à partir du code source.
2. Transformation : Un processus de changement d'une forme de représentation à une autre en restant dans le même niveau d'abstraction et gardant les fonctionnalités représentées.
   - source-to-source : une méthode purement implémentation qui consiste à établir une correspondance (mapping) entre les éléments de la source et la cible. reformule la phrase. Ce n'est pas clair
   - model-to-model : une technique qui consiste à transformer un modèle de la source vers un modèle cible. Ces modèles sont des outils pour une meilleure rationalisation de transformation en appliquant l'ingénierie dirigée par les modèles (nous découvrirons l'ingénierie dirigée par les modèles dans les sections suivantes). La Figure 5 illustre le processus de transformation basé sur les modèles.

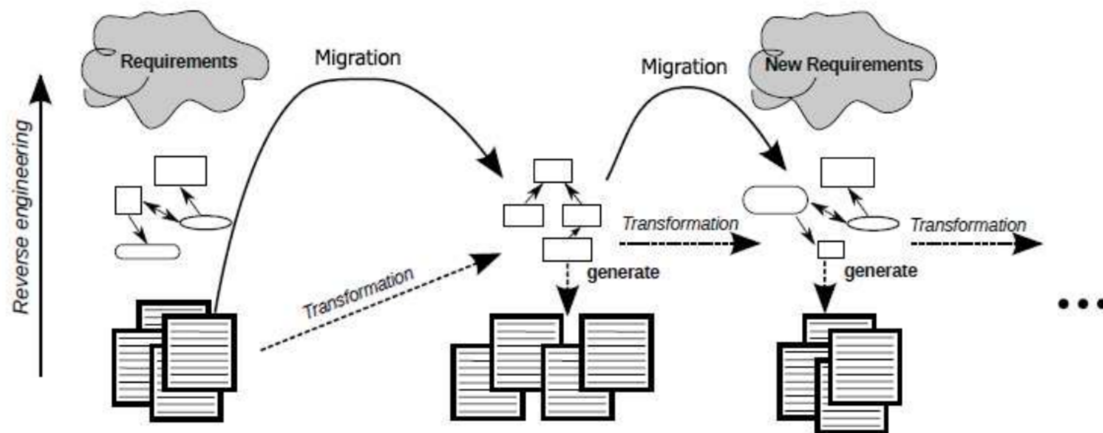


FIGURE 5 – Les Méthodes de Transformations (AL-SHARA 2016)

Par exemple, l'architecture d'une application Web utilisant le style MVP (Model View Presenter) [2] récupérée dans la partie de la rétro-ingénierie sera transformée en une autre comme une architecture MVVM (Model View ViewModel) [3]. Lors de transformation, la structure du code source peut être améliorée dans la direction de nettoyer le code qui optimise l'introduction des bugs (FOWLER 1999).

3. Génération du code (Code generation) :
   Après que la transformation est faite, une étape de d'ingénierie directe est appliquée pour passer de la représentation extraite au niveau d'implémentation. La technologie cible doit être fixée auparavant. En gardant le même exemple (cité dans Transformation), l'architecture transformée en MVVM peut être générée d'une manière automatique ou semi-automatique vers une application cible.

5 Conclusion

Dans ce chapitre, nous avons présenté les différents concepts liés à la problématique de l'évolution logicielle tels que la réingénierie et la migration. D'après les concepts présentés, nous avons pu constater que l'utilisation de modèles et de transformations de modèles acquiert un rôle central dans une approche de migration logicielle pilotée par les modèles. Par conséquent, nous découvrons dans le prochain chapitre la notion du modèle, méta-modèle à travers l'introduction de l'approche *Ingénierie Dirigé par les Modèles*

2. MVP : est un style architectural destiné pour les applications web. c'est complètement séparé de la présentation ou de la vue (View). Dans ce pattern, le Presenter (Contrôle) joue le rôle de contrôleur afin d'isoler toute communication directe entre la View (présentation) et le Model (objet métier).

3. MVVM : est un style architectural proposé par Microsoft. il permet de séparer la vue de la logique et de l'accès aux données en appliquant les principes de liaison et d'événement.

CHAPITRE 2

INGÉNIERIE DIRIGÉE PAR LES MODÈLES (IDM)

Sommaire

1 Introduction 15
2 Ingénierie Dirigée par les Modèles (IDM) 15
2.1 Définition 15
2.2 Principes fondamentaux 15
2.2.1 Système 15
2.2.2 Modèle 16
2.2.3 Langage 16
2.2.4 MétaModèle 17
2.2.5 MétaMétaModèle MOF (Meta Object Facility) 18
2.2.6 Approche de l'architecture dirigée par modèle 19
3 Transformation des modèles 20
3.1 Taxonomie de transformation des modèles 21
3.2 Les approches de transformation des Modèles 22
3.2.1 Transformations Modèle vers Modèle 22
3.2.2 Transformations modèle vers code 23
4 Défis de l'IDM 23
4.1 Défis de recherche 24
4.2 Défis techniques 25
4.3 Défis sociales et communautaires 25
5 Conclusion 26

1 Introduction

À la fin des années 90, une nouvelle approche est proposée en génie logiciel nommée l'Ingénierie Dirigée par les Modèles (IDM). Cette approche a été réputée à la fois en *continuité* et en *rupture* avec les travaux d'approche objet (Bézivin 2004). D'abord, en continuité car les recherches concernent l'approche objet qui a poussé l'évolution vers les modèles. En effet, la liaison d'objets entre eux dans un logiciel porte le réflexe de les classifier en fonction de leurs différents types, par exemple, objet technique, objet métier et d'autres. L'objectif d'IDM est de fournir un grand nombre de modèles pour exprimer séparément les domaines d'objets (types). Néanmoins, IDM ne prend pas en considération l'intervention des patrons de conception (Helm et Johnson 1994) et la programmation par aspects (Kiczales et al. June 1997) d'où elle est considérée comme rupture d'après (Bezivin 2005a).

Puisque notre travail porte sur la thématique de migration logicielle basée sur l'ingénierie dirigée par les modèles, nous allons consacrer ce chapitre à la présentation de cette nouvelle approche (IDM).

2 Ingénierie Dirigée par les Modèles (IDM)

Avec l'approche objet, le principe de base « Tout est objet » a été utile pour la simplicité, la généralité et puissance d'intégration des problèmes affrontés sur la scène industrielle. De plus, l'approche objet est basée sur les relations entre les objets telles que l'héritage, la composition et d'autres (Henry 1993).

De façon similaire, en ingénierie des modèles, le principe de base « Tout est modèle » possède plusieurs propriétés intéressantes. Nous découvrons dans les sections suivantes ces propriétés.

2.1 Définition

L'IDM est une technique de développement mettant à disposition des concepts, des langages et des outils afin d'utiliser les modèles pour décrire à la fois le problème et sa solution à différents niveaux d'abstraction, et fournir un cadre pour définir quel modèle utiliser à un moment donné (Fondement et Silaghi 2004).

2.2 Principes fondamentaux

Dans cette section, nous définissons les principes de bases de l'IDM.

2.2.1 Système

Un système est un objet complexe constituée d'un ensemble d'objets liés les uns aux autres et qui interagissent entre eux.

2.2.2 Modèle

Le modèle est une abstraction dans le sense où il peut ne pas représenter tous les aspects et propriétés du système réel mais seulement ceux qui sont pertinents dans un contexte donné (Bezivin 2005b).

2.2.3 Langage

Un langage (L) est défini par le couple {S, Sem} où S est sa syntaxe et Sem sa sémantique. Cette définition est très générale et assez abstraite pour caractériser l'ensemble des langages, quel que soit le domaine. La syntaxe décrit les différentes constructions du langage et les règles d'agencement de ces constructions. La sémantique désigne le lien entre un signifiant (un programme, un modèle, etc.), et un signifié (p. ex. un objet mathématique) afin de donner un sens à chacune des constructions du langage (Chomsky 1956).

Un langage spécifique à un domaine (DSL) ou langage de modélisation est un langage informatique qui cible un type de problème particulier, plutôt qu'un langage à usage général qui vise tout type de problème logiciel (Fowler 2010). Un langage de modélisation (L_m) est défini selon le tuple { AS, CS, M^*_{ac} , SD*, M^*_a } où :

— ***AS*** **(Syntaxe Abstraite)** : exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations. Parmi les langages de métamodélisation, le standard MOF(Group January 2006) de l'OMG qui offre les constructions élémentaires qui permettent de décrire une telle syntaxe abstraite d'un langage de modélisation. La Figure 6 représente un extrait du standard MOF, nous découvrons son objectif dans la partie 2.2.5.

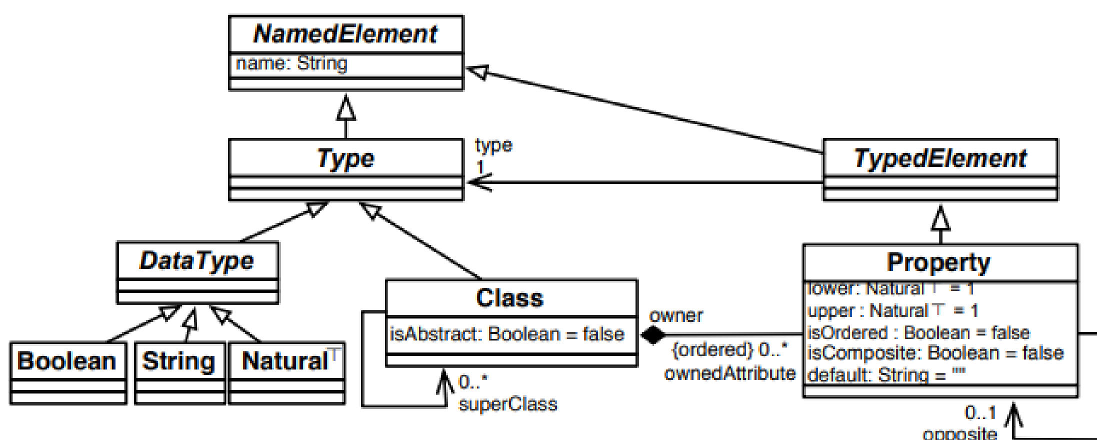


Figure 6 – Principaux concepts de métamodélisation (extrait de MOF 2.0 (Group January 2006)).

Nous remarquons qu'il y a une inspiration de l'approche objet, car il offre le concept de classe (*Class*) pour définir les concepts du domaine. Une classe est composée

de propriétés (Property) qui la caractérisent. Une propriété est appelée référence lorsqu'elle est typée (TypedElement) par une autre classe, et attribut lorsqu'elle est typée par un type de donnée (p. ex. booléen, chaîne de caractère et entier).
Dans la même optique, les langages de modélisation ne permettent pas d'introduire formellement les *conditions*. Pour cela, l'OMG recommande vivement d'utiliser OCL (Object Constraint Language) qui nous permet de préciser et limiter la vue sémantique d'un concept.

— ***CS* (Syntaxe Concrète)** : est un formalisme, graphique et/ou textuel, pour manipuler les concepts de la syntaxe abstraite et de créer des modèles. Ces modèles seront conformes à la structure définie par la syntaxe abstraite. Une syntaxe concrètre consiste à définir l'ensemble des mappings d'AS vers la CS ($M*_{ac}$). Ces mappings annotent chaque construction du langage de modélisation définie dans la syntaxe abstraite par une/plusieurs décoration(s) de la syntaxe concrète. Il existe en effet de nombreux projets servant à définir la syntaxe concrète, principalement, basés sur EMF(*Eclipse Modeling Framework*)[1] : GMF(*Generic Modeling Framework*)[2], OBEO Designer[3]. Ces derniers sont principalement graphiques, d'autres sont textuelles par exemple Xtext[4]. La diversité (graphique, textuelle) de la définition de syntaxe concrète a l'avantage de la normaliser.

— ***SD**** **(Domaine Sémantique)** : La sémantique du langage définit d'une manière non ambiguë la signification des constructions du langage. Il existe différentes techniques pour définir une sémantique (Harel, Rumpe et Braunschweig 2004) et les importants travaux sur les langages de programmation (Winskel 1993) on autorisa d'établir une taxonomie de ces techniques en fonction des besoins.
Le domaine sémantique d'un langage de modélisation est l'ensemble des informations permettant de distinguer les différents états du modèle au cours de son exécution. Définir une sémantique d'un langage de modélisation revient à définir ou à choisir le domaine sémantique *SD* et à définir le mapping $M*_{as}$ entre la syntaxe abstraite et le domaine sémantique (*AS* <-> *SD*).

2.2.4 MétaModèle

Un méta-modèle est un modèle qui permet de définir le langage utilisé pour exprimer le modèle. Il représente donc les entités d'un langage, leurs relations ainsi que leurs contraintes, Autrement dit, le méta-modèle est une spécification de la syntaxe du langage (Group January 2006).
En Figure 7, nous remarquons qu'il y a une relation primordiale entre la carte en bleu et la vue réelle (capture par satellite), appelée **Représente**. Dans cet exemple, une carte est un modèle (une représentation) de la réalité. Axiomatiser cette relation joue un rôle essentiel pour répondre à la question « qu'est ce qu'un bon modèle ? ». D'autre part, il est nécessaire d'accorder à chaque carte la description du « langage » (métamodèle) utilisé pour formaliser cette carte sous forme d'une légende expliparencite. Pour que la carte doit être utilisable, elle doit être conforme à cette légende. D'où, on conduit à l'identification d'une seconde relation, liant la carte et légende, appelée **conformeA**. La légende

1. http://www.eclipse.org/modeling/emf/
2. https://www.eclipse.org/gmf-tooling/
3. https://www.obeodesigner.com/en/
4. http://www.eclipse.org/Xtext/

est considérée comme un modèle représentant l'ensemble de cartes et à laquelle chacune d'entre elles doit se conformer.

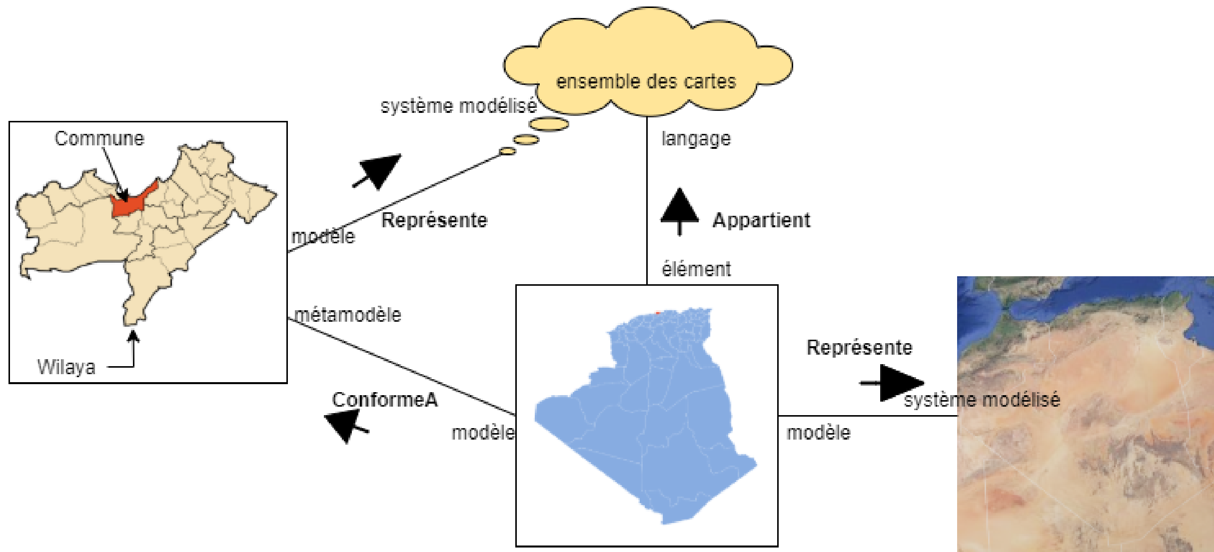


FIGURE 7 – Relations entre système, modèle, métamodèle et langage

2.2.5 MétaMétaModèle MOF (Meta Object Facility)

L'OMG avait proposé une forme général (BÉZIVIN, BRETON et al. 2003), représenté dans la Figure 8, pour affronter le défi de voir remonter indépendamment la variété de métamodèles et pour limiter le niveau d'abstraction. La solution logique fut donc de proposer un langage de définition de métamodèles qui a la capacité de se décrire lui-même dans le sense qu'il doit être réflexif et adaptatif pour limiter le nombre de niveaux d'abstraction, ce fut le métamétamodèle MOF (Meta-Object Facility). Le méta-métamodèle (MOF) est un modèle qui analyse Les éléments de modélisation nécessaires à la définition du langage de méta-modélisation pour pouvoir exprimer un méta-modèle.

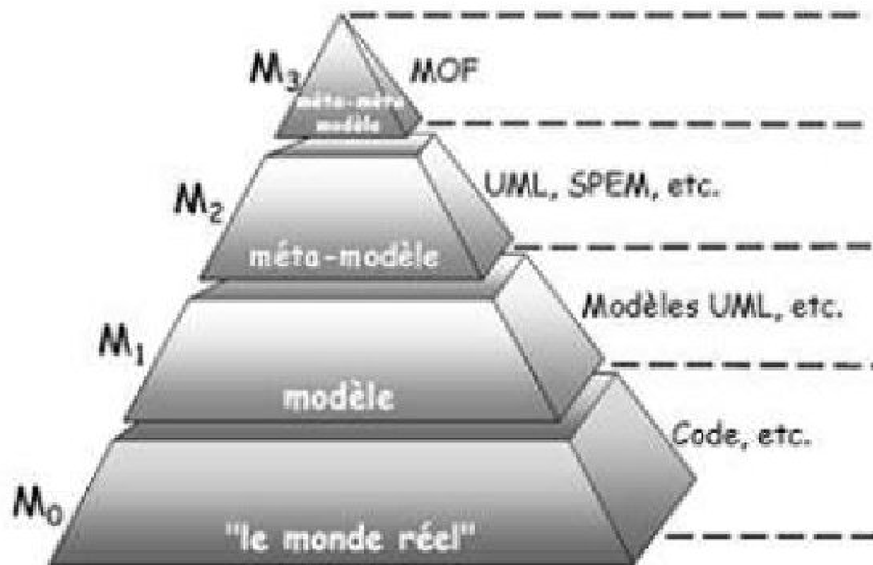


FIGURE 8 – Pyramide de l'OMG(DIAW, LBATH et COULETTE 2010)

La Figure 8 représente une hiérarchie pyramidale à quatre niveaux (DUCHIEN et DUMOULIN 2006) :

— Niveau M0 : Le monde réel qui l'environnement du système à étudier.

— Niveau M1 : Les modèles représentant le système défini dans un certain langage.

— Niveau M2 : Les méta-modèles permettant la définition de ces modèles.

— Niveau M3 : Le méta-méta-modèle (MOF).

2.2.6 Approche de l'architecture dirigée par modèle

ADM (Architecture Dirigée par Modèle) est une démarche de modélisation, développement et maintenance proposée en 2000 par L'OMG basé sur l'IDM pour émettre les bonnes pratiques de la modélisation en séparant les spécifications fonctionnelles d'un système des détails de son implémentation sur une plat-forme donnée. La démarche d'ADM se base sur quatre niveaux de modélisation représentés par des modèles, présentée dans la Figure 9, appelée *Pile de modélisation* ou cycle de développement en Y (DIAW, LBATH et COULETTE 2010) (BÉZIVIN, BLAY et al. 2004), sont :

— **CIM (*Computational Independent Model*)** : est un modèle métier modélisant les exigences du système tout en cachant les détails d'implémentions et décrivant le système d'un point de vue indépendant de l'informatisation. Son but est de faciliter la compréhension du domaine en réduisant l'écart entre les experts du domaine et les concepteurs.

— **PIM (*Platform Independent Model*)** : est un modèle qui modélise le système sans décrire les détails de son environnement et son utilisant sur une plat-forme particulière. Sa mission est de bien présenter la logique métier du système.

— **PDM (*Platform Description Model*)** : est un modèle décrivant la plat-forme technique qui permet d'implanter le système avec la vision d'analyse et de conceptions du système décrite dans le PIM. Le but du PDM est de permettre l'intégration de plat-forme technique dans la construction logiciel.

— **PSM (*Platform Specific Model*)** : est un modèle produit par la transformation du PIM dépendant d'une plate-forme technologique. Il spécifie l'utilisation du système dans la plate-forme choisie et qui serve à la génération de code exécutable vers la même plate-forme technologique choisie.

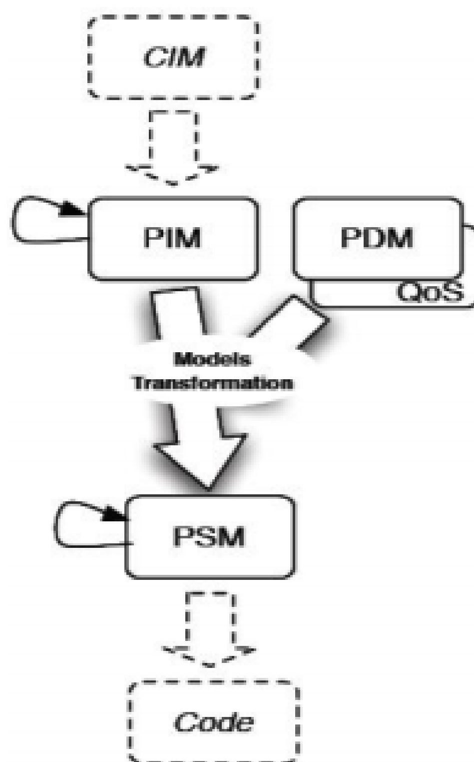


FIGURE 9 – Pile de modélisation (DIAW, LBATH et COULETTE 2010)

3 Transformation des modèles

L'ingénierie dirigée par les modèles propose une démarche dont les deux originalités sont d'une part la formulation des modèles, et d'autre part des programmes de transformation de modèles que nous allons y s'expliquer dans la suite.

Une définition précise, de transformation des modèles, est énoncé dans (BEZIVIN, JOUAULT et PALIÈS 2005) : *"La transformation de modèle est le processus de conversion d'un modèle en un autre modèle du même système, selon le guide Model Driven Architecture (ADM) (*OMG *2003). Cela signifie passer d'un modèle source à un modèle cible. Les deux doivent se conformer à leur métamodèle respectif. Ces deux métamodèles*

se conforment à un métamétamodèle. La transformation elle-même est un modèle, décrit dans un langage, c'est-à-dire conforme à son métamodèle. Ce métamodèle de transformation est conforme au métamétamodèle. Le métamodèle de transformation définit le langage de transformation, qui peut par exemple être déclaratif ou impératif."

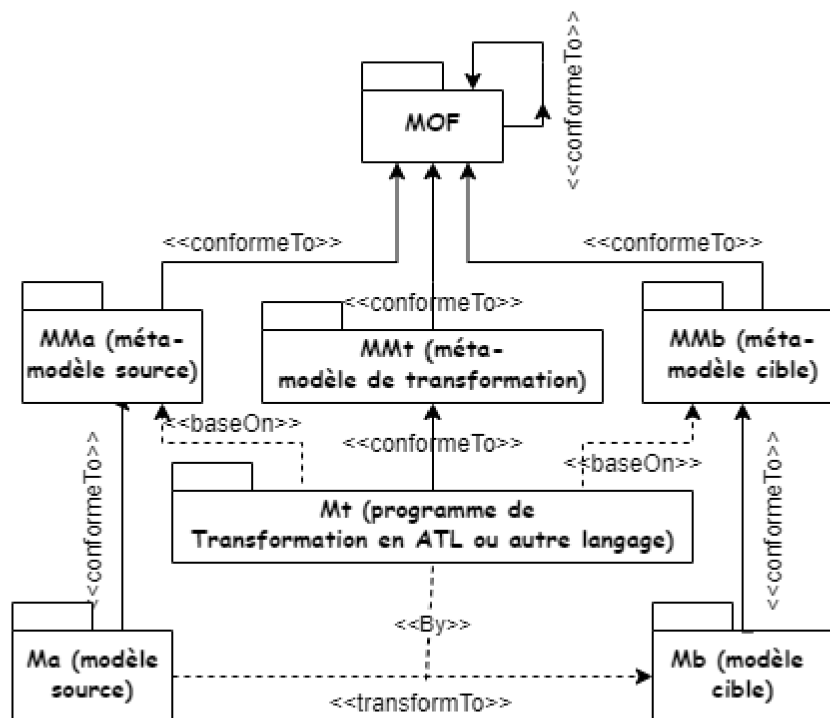


FIGURE 10 – La transformation de modèles basée sur les méta-modèles (BEZIVIN, JOUAULT et PALIÈS 2005)

La Figure 10 représente le processus de transformation des modèles. Dans (JOUAULT et al. 2008), ce processus est nommé comme un patron d'ingénierie pilotée par les modèles. L'intérêt principale de cette phase est de produire des modèles compréhensibles facilitant le guidage d'une activité humaine de programmation et l'alimentation des outils de production automatique de logiciel (génération de code, refactoring, migration technologique, la mesure de qualité du code source etc).(BÉZIVIN 2004)

3.1 Taxonomie de transformation des modèles

La transformation des modèles comprend plusieurs types ou chacun est caractérisé par son niveau d'abstraction qui sont (DOLQUES 2010) :

— *Transformation endogène* :une transformation horizontale entre des modèles (sources/ cibles) exprimés dans même le langage de modélisation. Parmi les utilisations de ce type est améliorer certaines qualités opérationnelles d'un modèle optimisation des performances, restructurer un modèle pour le rendre maintenable tout en préservant sa sémantique.

— *Transformation exogène* :une transformation horizontale ou chaque modèle (source/ cible) conforme à un méta-modèle unique. Ces méta-modèles sont dans le même niveau

d'abstraction, exprimées dans des langages de modélisation différents. Par exemple transformation d'un modèle diagramme de classes UML vers un un diagramme entité/association. La Figure 11 illustre la transformation exogène et endogène.

— *Transformation verticale* :lorsque la source et la cible sont définies à des niveaux d'abstraction différents. Une baisse du niveau d'abstraction d'une transformation est appelé raffinement. Par contre, une abstraction consiste en une élévation du niveau. La Figure 12 montre la transformation horizontal et verticale.

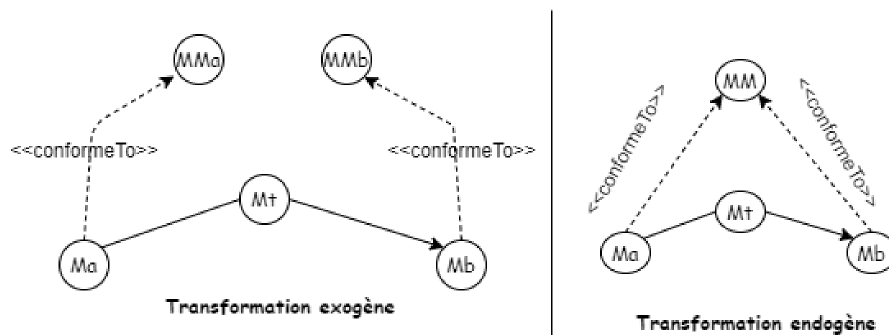


FIGURE 11 – Transformation endogène et exogène

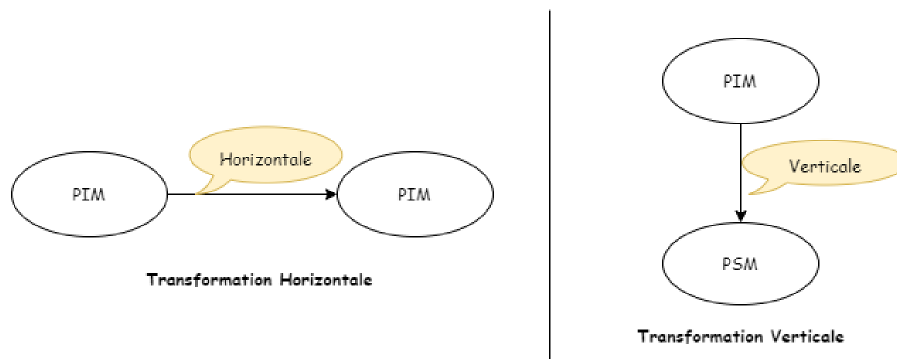


FIGURE 12 – Transformation horizontale et Verticale

3.2 Les approches de transformation des Modèles

Il existe deux approches de transformation importantes (CZARNECKI et HELSEN 2003) :

3.2.1 Transformations Modèle vers Modèle

Les transformations de modèle à modèle traduisent les modèles source en modèle cible, qui peuvent être des instances de méta-modèles identiques ou différents. Nous distinguons cinq techniques de transformation « Modèle vers Modèle » :

— *Approche par manipulation directe :* cette approche de transformation est basique, consiste à faire la transformation d'une manière programmable codé en dur en utilisant un langage de programmation standard. Le développeur se préoccupe par l'écriture des règles de transformation entre les entités des modèles (source/cible).

— *Approche relationnelle :* cette approche est basée sur la programmation déclarative qui exige l'utilisation de relations d'ordre mathématique pour identifier les relations entre les éléments du modèle source et ceux du modèle cible sous des contraintes en les formulant comme des règle de transformation.
— *Approche basée sur les transformations de graphes :* cette catégorie d'approches de transformation de modèle s'appuie sur les travaux théoriques sur les transformations de graphes. L'idée est se représenter le modèle source comme un graphe et par la suite en appliquant les travaux théoriques pour le transformer en un autre graphe qui représente un modèle cible. En particulier, cette approche applicable sur des graphes typés, attribués, étiquetés, qui sont des sortes de graphes spécifiquement conçus pour représenter des modèles de type UML. Cette approche est similaire à l'approche relationnelle dans le sens où elle permet d'exprimer les règles de transformations sous un style déclaratif.
— *Approche basée sur la structure :* le processus de l'approche consiste à faire une adaptation qui créer une arborescence du modèle cible par le modèle source et par la suite il l'ajuste par les attributs et références de la source.
— *Approche hybride :* les approches hybrides sont des approches couplant différent approches en utilisant la logique déclarative et impérative à la fois. Elle permet de combiner les différents avantages des approches proposées. Parmi les langages utilisées dans ce contexte est OCL.

3.2.2 Transformations modèle vers code

Ce type de transformation est un cas particulier du type précédent. Les techniques de transformations se résument dans :

— *L'approche Visitor-Based* : cette approche base sur un mécanisme de visiteur pour traverser la représentation interne du modèle source et écrire par la suite un code sous forme d'un flux de texte. Un exemple de cette approche est Jamda [4], qui est un framework orienté objet fournissant un ensemble de classes pour représenter les modèles UML, une API pour manipuler les modèles et un mécanisme de visiteur (appelé CodeWriters) pour générer du code. Jamda ne prend pas en charge la norme MOF pour définir de nouveaux métamodèles.

— *L'approche Template-Based* : cette approche conduit à générer un modèle cible sous forme d'un flux du code source utilisant des morceaux du méta-code (méta-programming) pour arriver aux informations du modèle source. Il y a plusieurs outils de transformation des modèles supportant cette approche, par exemple : AndroMDA [5], b+m Generator Framework [6].

4 Défis de l'IDM

A partir de la fin des années 90 jusqu'à 2007, il y avait un corps très important de recherche sur les langages de modélisation en tenant compte de ses sémantiques, ainsi

4. http://jamda.sourceforge.net/
5. http://andromda.sourceforge.net/
6. https://www.omg.org/mda/mda_files/b+m_OMGCommittment.pdf

que le processus de méta-modélisation. C'était la période d'efforts de standardisation des spécifications qui aboutit à la production du Méta-Objet Facility (MOF), Model-Driven Architecture (MDA) et Object Constraint Language (OCL). Le cas d'utilisation clé pour la modélisation était la génération de code. Plus récemment, les chercheurs du domaine IDM rencontrent une variété de défis. Chaque défi suit un aspect. Pour l'instant trois ailes de défis existent : recherche, technique et sociales (Antonio et al. 2020).

4.1 Défis de recherche

Les défis de recherche sont des défis qui résultent des concepts théoriques. Les principaux problèmes qui ont été identifiés dans diverses feuilles de route de recherche comprennent les suivants :

— *Ingénierie linguistique (Language engineering)* : concerne l'ensemble des principes, pratiques et patrons pour spécifier la syntaxe abstraite, concrète et sémantique des langages de modélisation.

— *Ateliers de langues (Language workbenches)* :concentre la modernisation des outils liés aux langages spécifiques à un domaine et IDM. Actuellement, des outils intéressants qui existent, par exemple : JetBrains MPS [7], Xtext [8], Kermeta [9], Racket [10] et Spoofax [11].

— *Gestion des modèles (Model management) :* l'automatisation des tâches et processus de manipulation et analyse de modèles tels que les testes de validations des modèles, comparaisons entre les modèles et fusion des modèles.

— *Analyse du modèle (Model analysis) :* concentre sur les pratiques d'analyse d'un modèle ainsi que les principes qui amènent à dire qu'est ce qu'un bon modèle.

— *Modèles à l'exécution (Models at runtime) :* est l'espace de recherche qui collecte l'ensemble des techniques et des outils pour refléter automatiquement les changements d'un système en changements de modèles, et vice versa. Ce défi particulier se situe à l'intersection de la modélisation et de la recherche en intelligence artificielle.

— *Référentiels de modélisation (Modeling repositories) :* porte sur l'assurance de persistance des produits de modélisation , tels que les anciens modèles, modèles transformations et les méta-modèles à travers les dépôts de produits (npm -> js, packagist -> php). L'objectif est propager et supporter l'idée d'open source en IDM et de rendre des produits réutilisables, de stocker et d'acquérir plus facilement de tels produits.

— *Extensibilité (Scalability) :* implique les techniques d'optimisation de la complexité d'un très grand modèle (avec des centaines de milliers d'éléments, sinon plus), grand méta-modèle, grand modèle de transformation ainsi que les techniques d'optimiser les performances.

Des progrès substantiels ont été réalisés dans ces domaines au cours de la dernière période de temps, et la recherche active se poursuit contre de nombreux ces domaines.

7. https://www.jetbrains.com/mps/
8. https://www.eclipse.org/Xtext/
9. http://diverse-project.github.io/k3/
10. https://racket-lang.org/
11. http://strategoxt.org/Spoofax

4.2 Défis techniques

Cette partie décrit les défis techniques. Les défis ont été divisés en catégories. La catégorisation n'est pas stricte car il n'a pas de limites des problèmes ; au contraire, c'est un pragmatique qui vise à faciliter la présentation du défis. D'après (Antonio et al. 2020), il existe actuellement 5 catégories, qui sont :

— *Fondation :* la dimension fondation comprend tous les défis concernant les aspects conceptuels et théoriques de toutes les phases du développement logiciel (c'est-à-dire la modélisation, le déploiement, l'exécution et la maintenance). Parmi les défis actuels dans ce contexte, comment étendre les techniques de l'IDM pour automatiser et rendre plus performantes toutes les solutions de maintenance en exploitant les techniques d'intelligence artificielle qui sont aujourd'hui prêtes à être utilisées pour les systèmes complexes et hautement dynamiques.

— *Domaine :* cette catégorie traite les questions liées au domaine d'application des systèmes développés à l'aide de l'IDM, tels que les villes intelligentes, l'automobile, l'aérospatiale, le nucléaire et les soins de santé. Ces domaines visent à garantir un ensemble de propriétés particulières telles que la confidentialité, la sécurité et intégrité. Deux objectifs essentiels des solutions liées à cette catégorie sont de réduire les risques et de garantir la fiabilité du logiciel développé.

— *Outil :* le bon outillage joue un rôle primordial pour l'exploitation de l'approche IDM en industrie. Pour l'instant, plusieurs défis confortés dans cette catégories tels que le problème de visualisation qui consiste à créer des éditeurs ou des outils facilitant la description d'un ou plusieurs problèmes complexes liées aux modèles. Cette visualisation nécessite points de vue hétérogènes.

4.3 Défis sociales et communautaires

Afin de savoir si les solutions techniques sont applicables dans la réalité, des aspects sociaux et communautaires sont parfois également considérés pour pouvoir vérifier de telles solutions techniques ou s'assurer qu'elles sont adoptées dans la pratique.

— *Communauté :* cette partie met des conditions sur les référentiels de modélisation 4.1 discuté auparavant. Parmi les questions posées : comment assurer que les modèles de dépôts de partage sont de bons qualités d'une manière automatique (avoir le principe de Bump de github) ? Un autre problème majeur liée à l'enseignement d'IDM, c'est comment on mise en place d'environnements pédagogiques adaptés à l'IDM qui ne soit pas d'une dureté décourageante ? car l'enseignement d'IDM actuelle concentre sur la construction des outils avant leurs utilisations en industrie.

— *Social :* L'IDM devrait être un encouragement pour permettre aux personnes non techniques de construire les outils dont elles ont besoin dans leurs domaines (Modélisation par le peuple, pour le peuple (Kelly 2018)). Ce point représente un des arguments pour rendre l'IDM accessible par toutes les parties. Mais ceci peut égarer ces personnes si elles ne sont pas guidé par d'experts d'IDM.

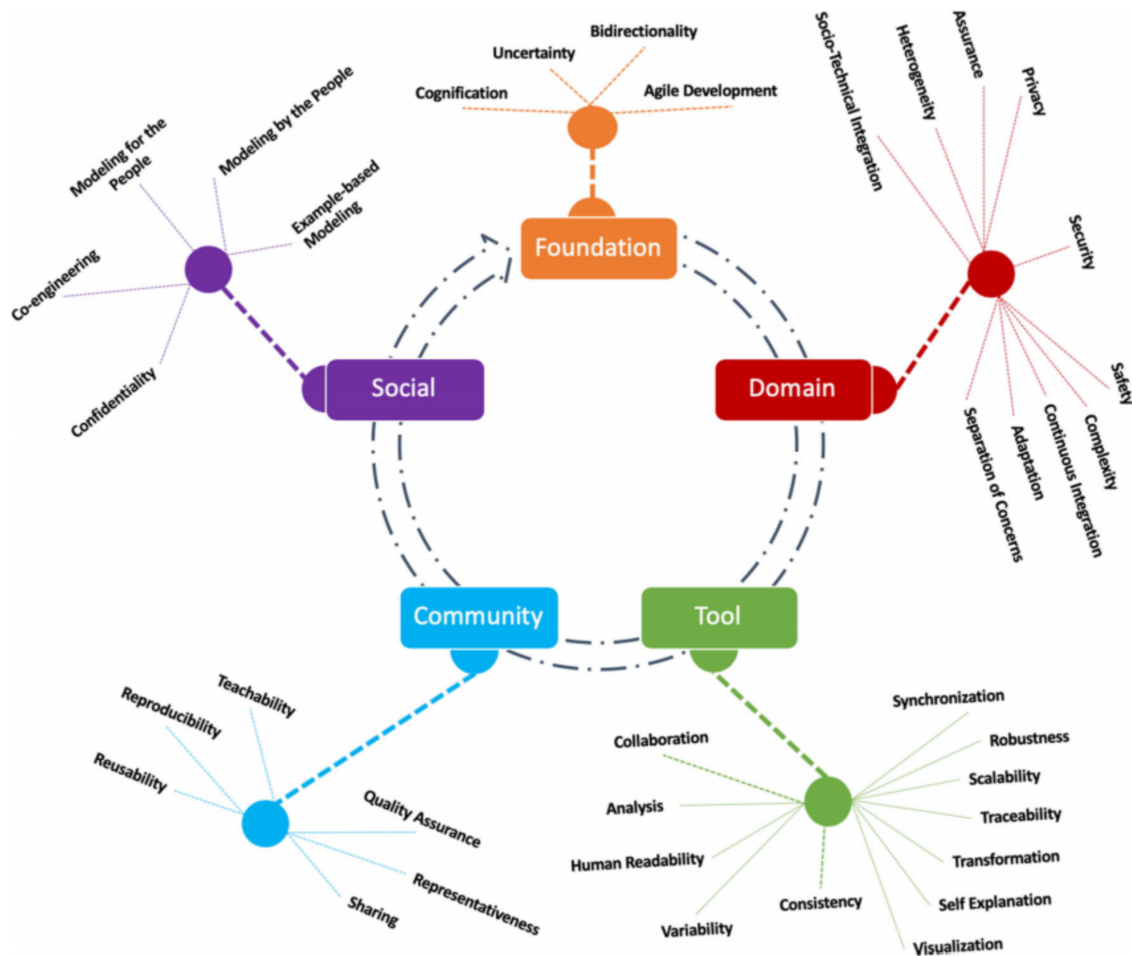


FIGURE 13 – Classification des défis (ANTONIO et al. 2020)

5 Conclusion

Dans ce chapitre, nous avons présenté les concepts fondamentaux, les approches de transformation des modèle et les défis de l'IDM. Nous avons pu constater que la méta-modélisation est appliquée pour la résolution de beaucoup de problèmes, tels que les problèmes nécessitant la réalisation de systèmes en temps réel et systèmes embarqués et systèmes de télécommunication, et, plus récemment, au développement et à l'intégration des systèmes d'information d'entreprise, les systèmes sensibles au contexte.

Dans ce qui suit, nous allons voir l'utilité de l'IDM pour la rétro-ingénierie des logiciels dans le chapitre suivant.

CHAPITRE 3

RÉTRO-INGÉNIERIE DIRIGÉE PAR LES MODÈLES (MODEL DRIVEN REVERSE ENGINEERING)

Sommaire

1 Introduction 28
2 Définition 28
3 Processus de Base de Model Driven Reverse Engineering . . 28
4 L'initiative de Modernisation Dirigée par l'Architecture et Standardisation des modèles pour MDRE 29
4.1 Définition de Modernisation Dirigée par l'Architecture (ADM) : 29
4.2 Processus d'ADM 30
4.2.1 Rétro-ingénierie 31
4.2.2 Restructuration 31
4.2.3 Ingénierie 31
4.3 Aperçu sur le méta-modèle KDM 31
4.3.1 Architecture du KDM 32
4.3.2 Paquet Code du KDM 33
4.4 Aperçu sur l'outil MoDisco 34
4.4.1 Architecture 35
5 Conclusion 37

1 Introduction

Généralement, il est difficile de prédire les délais nécessaires pour terminer le processus de rétro-ingénierie. De plus, l'exploitation et l'application rapide de la rétro-ingénierie sur des systèmes existants reste un problème majeur en génie logiciel car le défi majeur est de pouvoir découvrir et comprendre les fonctionnalités de ces systèmes, leurs architectures, pouvoir représenter leurs données en une représentation significative qui peut être plus tard manipulée et implémentée (BRUNELIERE et al. 2010). De plus, il n'existe pas des normes pour évaluer la qualité des résultats de la rétro-ingénierie (YANG et al. 2021). Pour pallier à ces défis, le *Model Driven Reverse Engineering* (SABIR et al. 2019) est apparu pour enrichir la rétro-ingénierie standard (YANG et al. 2021) et réintroduire la notion d'abstraction dans la représentation des systèmes héritées car les concepts sont plus résilient que leurs implémentations (PASCAL 2019).

Dans ce chapitre, nous définissons la notion de *Model Driven Reverse Engineering*. Après, nous expliquons son processus de base. Par la suite, nous allons découvrir l'initiative de *Modernisation Dirigée par l'Architecture* et normalisation des modèles.

2 Définition

La rétro-ingénierie dirigée par les modèles (MDRE) est une approche qui se base sur l'abstraction représentée par les modèles d'un logiciel donné pour faciliter la compréhension des fonctionnalités du logiciel. Il s'agit de l'application des principes et des techniques d'ingénierie dirigée par les modèles (IDM) à la ré-ingénierie afin de générer des vues pertinentes basées sur des modèles pour des systèmes existants, facilitant ainsi leur compréhension et leur manipulation.» (REIS et A. SILVA 2017) (BRUNELIÈRE et al. 2014) (YANG et al. 2021)

3 Processus de Base de Model Driven Reverse Engineering

Généralement, le processus de MDRE est divisé en deux étapes selon (SABIR et al. 2019) et (BRUNELIÈRE et al. 2014). La Figure16, représente ce processus :

— *Reconstruction du modèle (Model Discovery)* : cette étape consiste à extraire un modèle représentant le logiciel donné ou au moins une partie, à partir de son code source, des données brutes, documentation, etc. Ce modèle fournit une représentation uniforme du système, qui conforme à un métamodèle donné affirmant un point de vue abstrait (SABIR et al. 2019). Ce métamodèle peut être spécifique à la technologie (JAVA, XML, JSP) ou plus générique, selon des spécifications exprimées. L'extraction du modèle est réalisée par un composant appelé découvreur qui est guidé par le métamodèle. Un découvreur peut être codé en dur ou semi-généré grâce à la transformation des modèles. Dans des cas réels, plusieurs modèles sont parfois nécessaires pour représenter un grand système donné. Cela comprend l'utilisation combinée de plusieurs découvreurs différents, basés sur des méta-modèles

distincts pour obtenir une vue générale du système.

— *Compréhension du modèle (Model Understanding)* :Après que les modèles du système donné sont extraits. Des transformations de modèles ou des chaînes de transformations sont appliquées jusqu'à obtenir les modèles finaux représentants les données souhaitées sur le système.

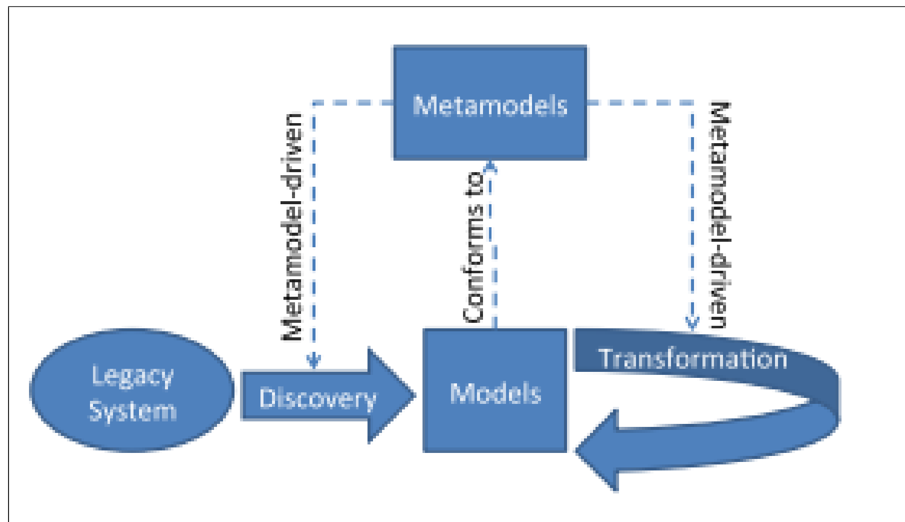


FIGURE 14 – Processus de MDRE (RAIBULET, ARCELLI FONTANA et ZANONI 2017)

Par la suite, le contenu de ces modèles est analysé et calculé. Ce processus peut souvent être automatisé pour certains scénarios déjà identifiés, puis exploités dans le cadre de cas d'utilisation plus spécifiques

Après que les données soient obtenues et analysées, nous pouvons réaliser efficacement un scénario cible (objectif) de la ré-ingénierie qui peut être rétro-documentation, calcul de métriques, analyse qualité, refactoring, génération de code, migration à une nouvelle technologie, etc.

4 L'initiative de Modernisation Dirigée par l'Architecture et Standardisation des modèles pour MDRE

Selon l'OMG, le manque de normes sur la façon de construire des outils de modernisation a augmenté le nombre des échecs dans les projets de modernisation des systèmes. Plusieurs problèmes sont apparus par exemple : la productivité de l'équipe de développement diminue, la difficulté de la réutilisation des concepts, algorithmes et des techniques utilisées. Pour cela, L'OMG propose la modernisation dirigée par l'architecture (ADM) (SANTOS et al. 2019).

4.1 Définition de Modernisation Dirigée par l'Architecture (ADM) :

D'abord, La modernisation est un terme général pour diverses activités qui ont déjà été expliquées en détail dans le chapitre 1.

L'ADM est une initiative proposée par l'Object Management Group (OMG) qui combine le processus de réingénierie logicielle avec l'architecture dirigée par les modèles (présentée dans la section 2.2.6) pour soutenir la maintenance et le développement de logiciels dans l'industrie (WAGNER 2014).

L'ADM repose sur des modèles standards qui peuvent représenter un système, parmi ces modèles les suivants (SANTOS et al. 2019) :

— Knowledge Discovery Metamodel (KDM) : c'est un méta-modèle pour représenter les informations relatives au logiciel existant et ses éléments tels les concepts structurels, par exemple : modules, classes, méthode y compris les associations et environnements opérationnels (GROUP 2012).
— Abstract Syntax Tree Metamodel (ASTM) : c'est un méta-modèle permettant de représenter l'arbre syntaxique abstraite (AST) pour n'importe quel langage de programmation. L'objectif d'ASTM est de fournir à un analyseur du code source d'un système une abstraction sur AST sans tenir au compte les détails d'implémentations (OMG 2011).
— Structured Metrics Metamodel (SMM) : il a été introduit pour définir des métriques de qualité du logiciel et de représenter leurs résultats de mesure sous forme un modèle. Parmi les métriques calculées : la complexité cyclomatique qui est calculée en fonction du nombre chemins dans le code (MOKEDDEM 2019).

4.2 Processus d'ADM

Le processus d'ADM (SANTOS et al. 2019) comporte trois phases qui sont la rétro-ingénierie, la restructuration et l'ingénierie représenté par la Figure 15.

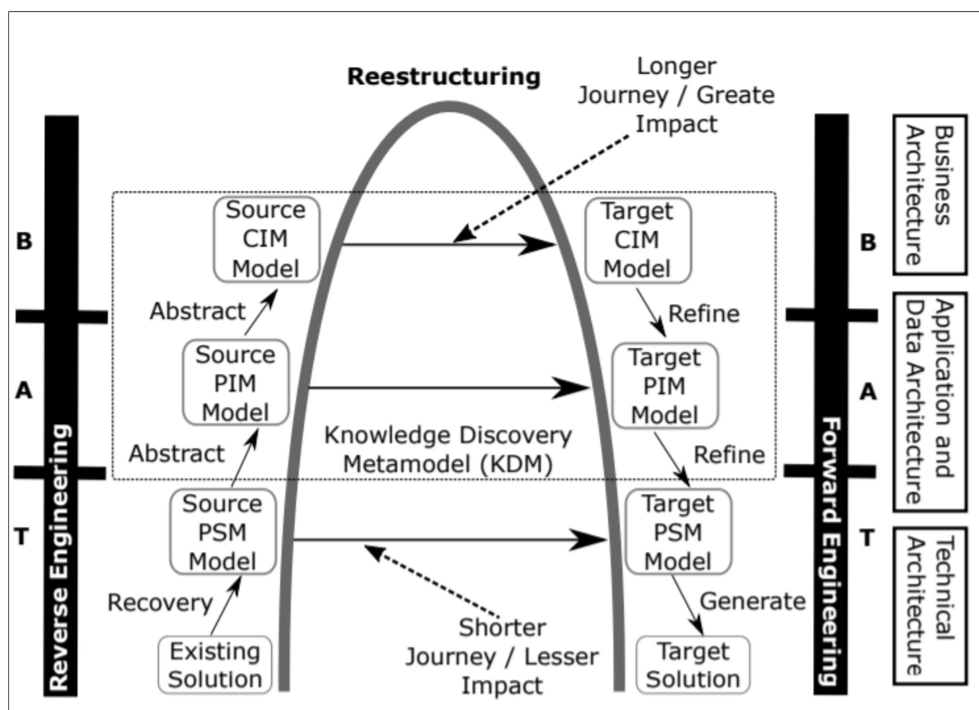


FIGURE 15 – Processus d'ADM (SANTOS et al. 2019)

4.2.1 Rétro-ingénierie

A partir d'une solution existante (système), un modèle spécifique à une plate-forme (PSM) est généré. Ce niveau d'abstraction est également appelé l'architecture technique. Une modernisation peut être effectuée à ce niveau telle qu'une migration d'un langage (p. ex. Pascal) à un autre langage de programmation (p. ex. Java).

Une transformation du PSM vers un modèle indépendante à une plate-forme (PIM) peut être effectuée pour abstraire l'architecture de la solution existante. Dans ce niveau, le modèle KDM est considéré comme un modèle PSM. Avec le PIM, il est possible d'effectuer des modernisations telles que : migration vers des architectures orientées services.

Le model Independent du calcule (CIM) peut être généré à partir d'un PIM. Ce niveau d'abstraction est le plus élevé dans le processus de modernisation. A ce niveau, les règles métier d'un système peut être représenté comme un moyen indépendant de l'ordinateur. Le niveau CIM est également appelé l'architecture métier. Il est possible d'effectuer des modernisations telles que : le déploiement de l'entrepôt de données et la gestion de portefeuille d'applications.

4.2.2 Restructuration

La phase de restructuration peut se créer dans n'importe quel niveau d'abstraction (PSM, PIM et CIM). Dans cette phase, la refactorisation des modèles peut être effectuée afin d'obtenir une version améliorée des structures représentées par les modèles en gardant ses comportements d'origines. Dans la même optique, plus le niveau d'abstraction est élevé, plus l'impact des changements dans le système est important. Dans ce contexte, Une refactorisation est une transformation de modèle qui peut améliorer la conception, la maintenance et la réutilisation des systèmes existants.

4.2.3 Ingénierie

Après que la restructuration a été effectuée, la phase d'ingénierie directe est déclenchée. Elle consiste d'effectuer un ensemble de transformations des modèles dépendant d'un niveau d'abstraction pour générer un nouveau code source (la solution cible).

4.3 Aperçu sur le méta-modèle KDM

L'une des caractéristiques les plus importantes de ce modèle est son exhaustivité et son étendue, car il est capable de représenter des détails de bas niveau, tels que les régions source, les méthodes et les propriétés, ainsi que ceux de niveau supérieur, tels que l'architecture, les règles métier et les événements. D'ailleurs, un autre caractéristique importante est que toutes ces caractéristiques de bas et de haut niveau sont liées dans le même méta-modèle, permettant une traçabilité entre eux. En 2011, KDM devient une norme ISO/IEC sous le numéro 19506.

4.3.1 Architecture du KDM

L'architecture de KDM est organisée en quatre couches dédiées à un point de vue particulier d'une application, représentées dans la Figure 16 où chaque couche dans un niveau dépend d'une ou de plusieurs couches inférieures. Chaque couche se compose de paquets où Chacun contient un ensemble d'éléments du méta-modèle KDM qui représentent les faits particuliers du système étudié, Nous les définissons brièvement (Martín Santibáñez, Durelli et Camargo 2015) :

— **Couche d'infrastructure** : cette couche représente le niveau de base d'abstraction dans tous les niveaux KDM. Elle se compose de trois paquets essentiels : Core, KDM et Source. Le Core et KDM s'occupe à définir des entités du base du méta-modèle KDM.Le package source définit un modèle nommé *Inventory Model* qui énumère les artefacts du logiciel existant et définit le mécanisme des liens de traçabilité entre les éléments KDM et leur représentation originale dans le code source du logiciel.

— **Couche des éléments de programme** : elle représente les éléments du code du système donné et leurs associations. Elle fourni une représentation indépendante du langage pour augmenter le niveaux d'abstraction et couvrir l'ensemble des langages de programmation. Il y a deux paquets dans cette couche : Code et Action.

— **Couche des ressources d'exécution** : elle représente une importante connaissance (comme l'environnement opérationnel) sur le système donné. Ce type de connaissance est extrait par une analyse dynamique au moment d'exécution (incrémental). Il y a quatre paquets dans cette couche : Data, Event, UI et Plate-form.

— **Couche des abstractions** : elle représente le domaine d'application et l'aspect métier du système donnée. Ce représentation de connaissances concerne l'intervention des experts et d'analystes. Conceptuel, Structure et Build sont les trois domaine de cette couche.

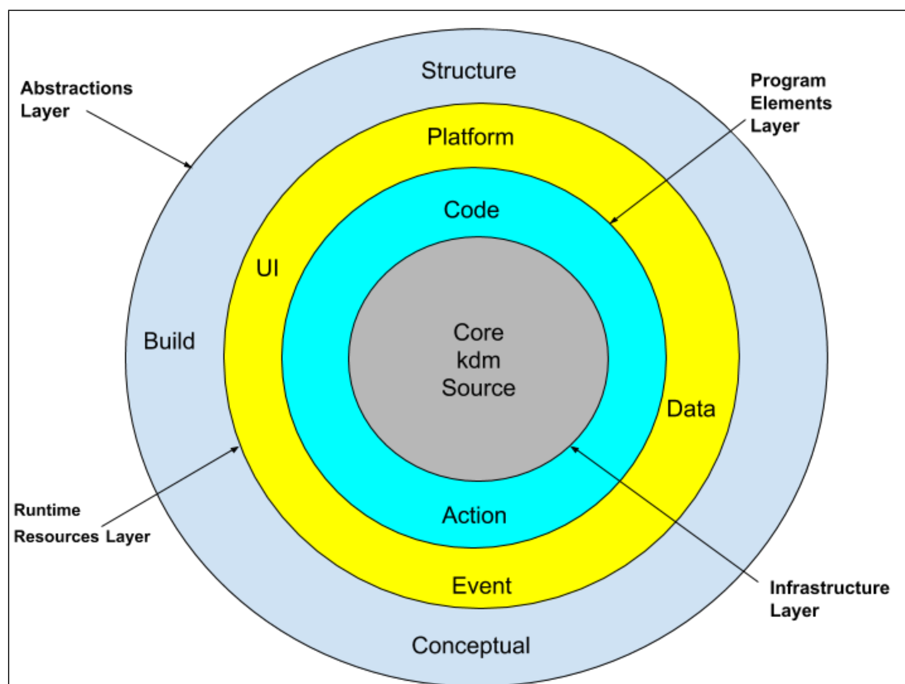


Figure 16 – Architecture en couche de KDM

4.3.2 Paquet Code du KDM

Le paquet de code du KDM se compose de 24 entités et contient tous les éléments abstraits pour la modélisation de la structure statique du code source. Dans une modèle KDM donné, chaque instance de l'entité de méta-modèle de paquet code représente une construction de langage de programmation, déterminée par le langage de programmation du logiciel existant. Chaque instance d'un élément de méta-modèle de code correspond à une certaine région du code source dans l'un des artefacts du logiciel existant. L'élément parent dans le paquet est le CodeModel qui est un conteneur pour les éléments de code, représenté dans la Figure 17. Dans la Figure 18, la ClassUnit représente les classes définies par l'utilisateur dans les langages orientés objet.

L'entité DataElement est un élément de modélisation générique qui définit les propriétés communes de plusieurs ClassUnit par exemple, les variables globales et locales, paramètres d'une fonction et d'autres. Le StorableUnit est un objet de calcul auquel différentes valeurs du le même type de données (DataType) peut être associé à des moments différents. Du point de vue de l'exécution, un élément StorableUnit représente un objet de calcul unique, qui est identifié soit directement (par son nom), soit indirectement (par référence). De plus, Le MethodUnit représente les fonctions membres appartenant à un ClassUnit, y compris les opérateurs, constructeurs et destructeurs définis par l'utilisateur. Du point de vue de l'exécution, chaque élément MethodUnit représente un objet de calcul qui existe dans le contexte d'une instance de classe

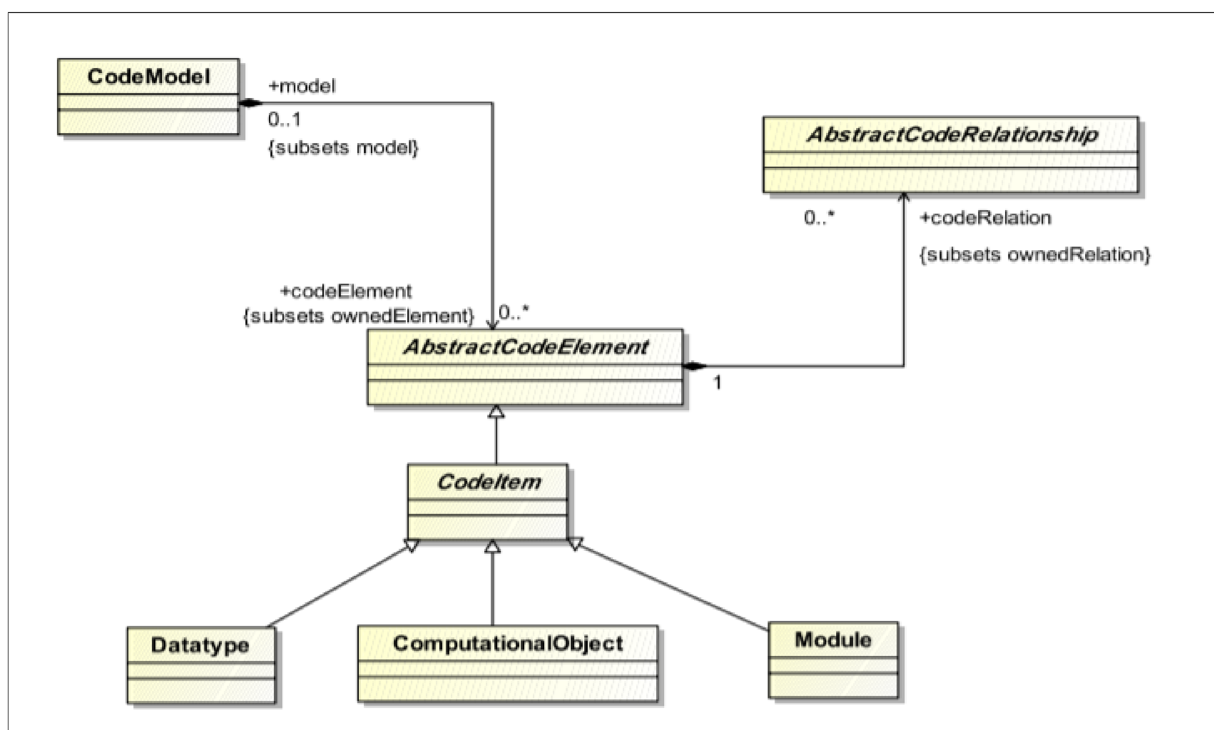


Figure 17 – Extrait 1 du paquet code (group 2012)

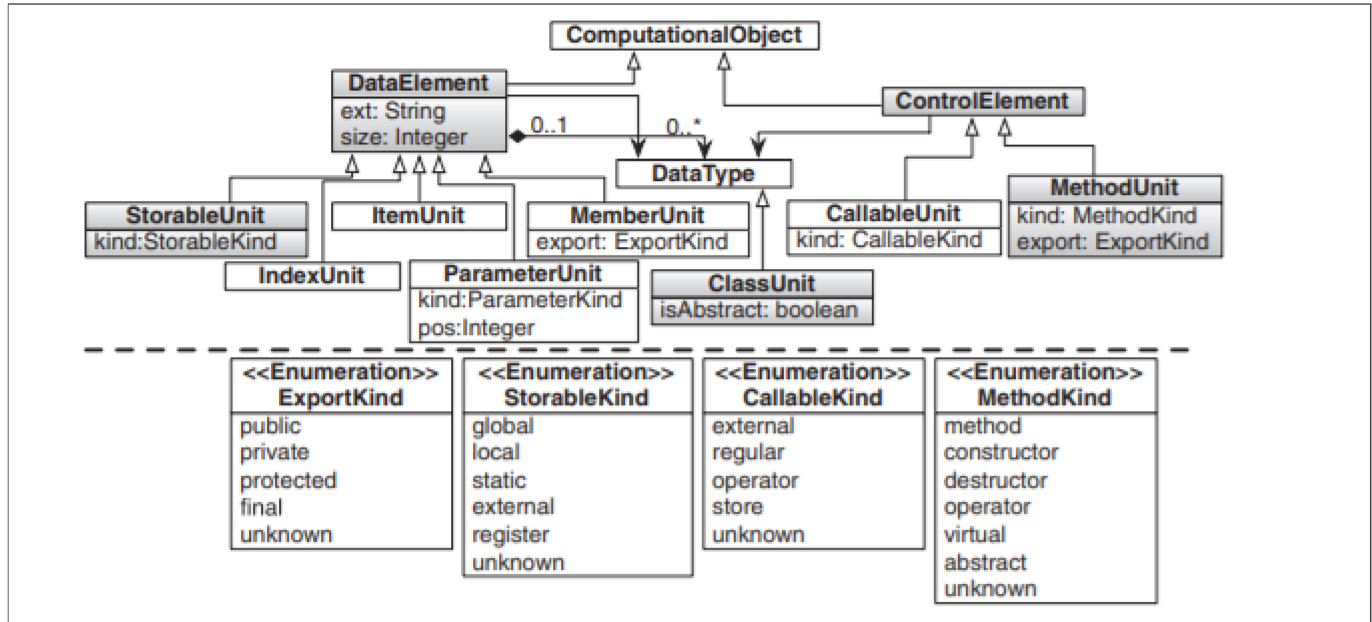


FIGURE 18 – partie du package de code (MARTÍN SANTIBÁÑEZ, DURELLI et CAMARGO 2015)

4.4 Aperçu sur l'outil MoDisco

MoDisco est un cadre de modélisation générique et extensible dédié au MDRE, est une preuve concrète que les principes et techniques IDM peuvent être efficacement appliqués dans le contexte de domaines importants et complexes tels que la rétro-ingénierie. MoDisco s'inscrit dans un projet open source sous Eclipse spécialement pour la modélisation. L'objectif est de faciliter le développement d'outils d'extraction de modèles des systèmes existants et les utiliser dans des cas d'utilisation de modernisation (BRUNELIÈRE et al. 2014).La Figure 19 illustre un aperçu sur les objectifs derrières MoDisco.

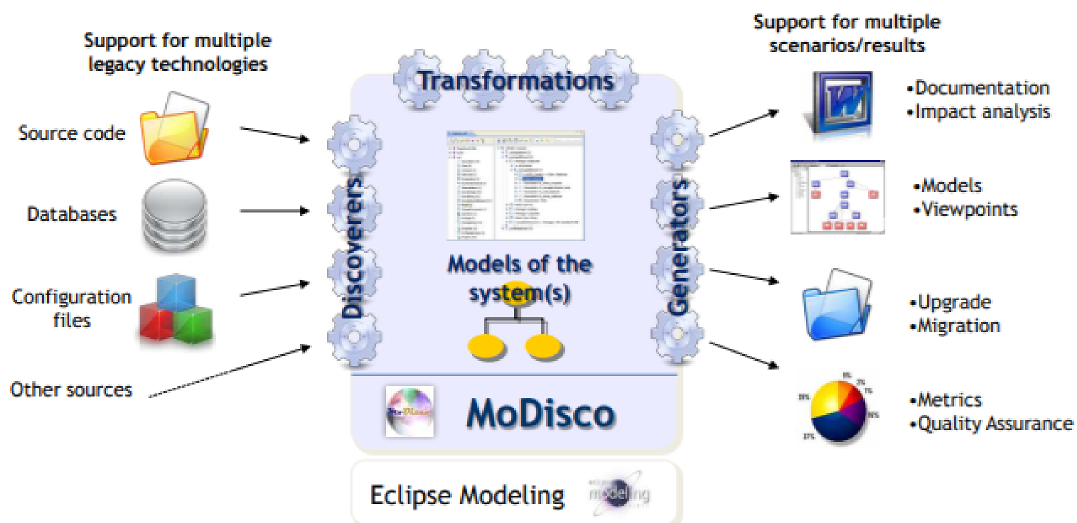


FIGURE 19 – Objectifs de MoDisco (BRUNELIÈRE et al. 2014)

4.4.1 Architecture

MoDisco vise à fournir une plate-forme prenant en charge divers cas d'utilisation de modernisation hérités pour divers types de technologies existantes. Pour faciliter la réutilisation des composants entre plusieurs cas d'utilisation, MoDisco a été organisé en trois couches, La Figure 20 représente les différentes couches de MoDisco :

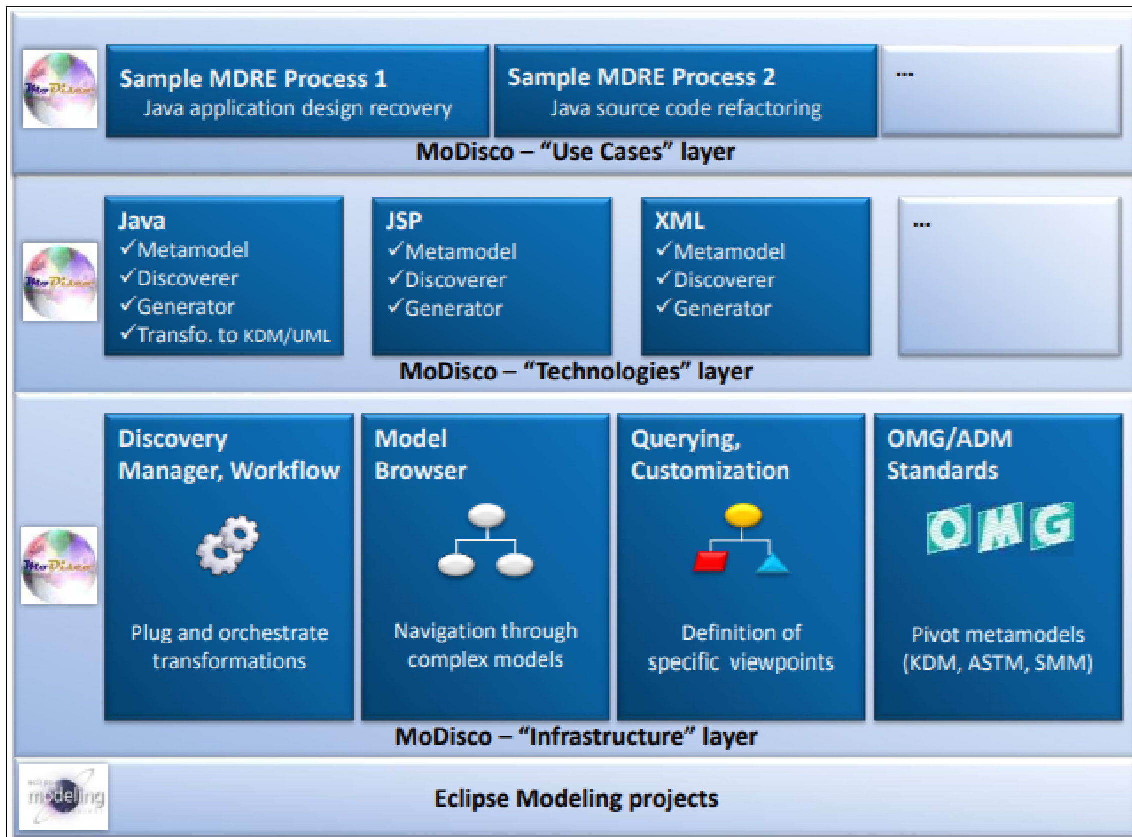


FIGURE 20 – Architecture de MoDisco (BRUNELIÈRE et al. 2014)

1. Cas d'utilisation (Use Cases) : couche contenant des composants fournissant une solution pour un cas d'utilisation de modernisation spécifique. Les principaux de cas sont bien identifiés : Comparaison (comparaison entre les versions du logiciel au niveau structurel), Analyse Qualité, Cartographie (détection des principaux composants d'un système et de leurs dépendances) , Compréhension, Reverse-Modeling(création de modèles à partir de systèmes existants pour alimenter les outils de modélisation), Refactoring et Migration. Un autre cas l'extraction de règles métier à partir de programmes pour alimenter un moteur de règles métier, la modification d'un système existant pour mieux s'intégrer à un autre système.

2. Technologie (Technologie) : couche contenant des composants dédiés à une technologie existante mais indépendant du cas d'utilisation de la modernisation. Ces composants peuvent être réutilisés entre plusieurs composants de cas d'utilisation impliquant la même technologie héritée. Par exemple, un cas d'utilisation de métriques de calcul sur le code source Java et un autre fournissant une refactorisation pour les applications Java pourraient réutiliser le même composant. La Figure re-

présente un modèle Java qui conforme à un modèle ASTM d'une application généré par MoDisco.

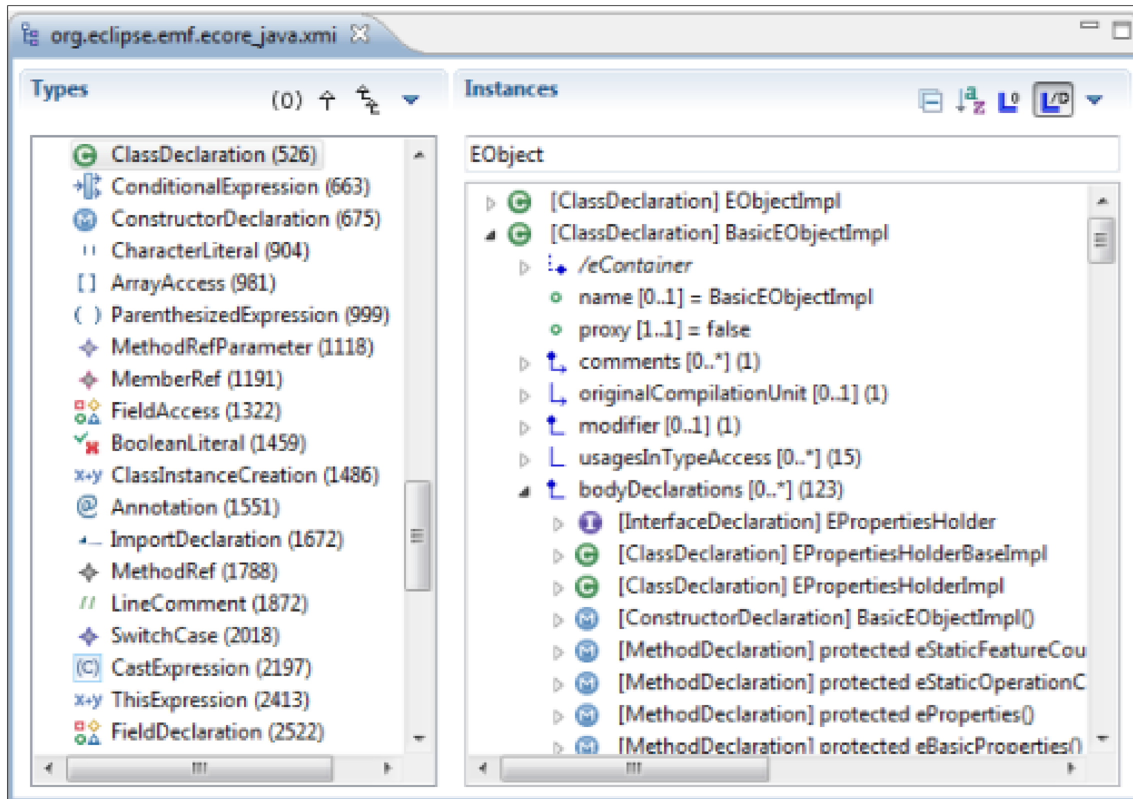


FIGURE 21 – Modèle Java généré par MoDisco (BRUNELIÈRE et al. 2014)

Chaque composante technologique est composée, au minimum, d'un métamodèle de la technologie dédiée. Ce métamodèle décrit les éléments requis pour prendre en charge les cas d'utilisation de modernisation pour la technologie correspondante. Selon le type de cas d'utilisation, le métamodèle peut être complet (pour la refactorisation ou la migration) ou partiel (pour la cartographie ou certains scénarios d'analyse de la qualité).

3. Infrastructure : couche contenant des composants génériques indépendants de toute technologie existante. Il y a deux types de composants dans cette couche. Des composants de connaissance fournissent des métamodèles décrivant les systèmes existants indépendamment de leur technologie. Comme les composants de la couche Technologie, ces composants peuvent être livrés avec des découvreurs ou des utilitaires. Des exemples de composants MoDisco Knowledge sont les métamodèles d'OMG/ADM : KDM, ASTM et SMM. D'autres types sont les composants techniques représentées par des découvreurs abstraits dont peuvent dériver les découvreurs concrets. Ainsi qu'un métamodèle de système de fichiers décrivant l'organisation des fichiers et des répertoires. un navigateur de modèles facilitant la visualisation des modèles MoDisco.

5 Conclusion

Dans ce chapitre, nous avons vu l'importance des modèles pour la compréhension du code source d'un logiciel existant. Nous Nous avons décrit brièvement l'initiative ADM et la standardisation des modèles pour l'évolution de logiciels. Par la suite, nous avons donné un aperçu sur le méta-modèle KDM et son exploitation à travers l'outil MoDisco.

Dans le chapitre suivant, nous nous focalisons sur l'étude des approches de migration logicielles et leur différentes classifications pour cerner au mieux l'existant et notre contribution par rapport à cet existant.

CHAPITRE 4

TRAVAUX CONNEXES

Sommaire

1 Introduction 40
1.1 Les entrées nécessaires pour les approches de migration 40
1.1.1 Code source 40
1.1.2 Contexte du GUI 40
2 Le processus des approches de migration 42
2.1 Analyse du code 42
2.1.1 Analyse Statique 42
2.1.2 Analyse Dynamique 42
2.1.3 Analyse Hybride 42
2.2 Automatisation 42
2.2.1 Automatique 42
2.2.2 Semi-Automatique 43
2.2.3 Manuelle 43
2.3 Directions du processus 43
2.3.1 Descendante 43
2.3.2 Ascendante 43
2.3.3 Hybride 43
2.4 Méthodes de Migration 44
2.4.1 Nouveau développement 44
2.4.2 Wrapping 44
2.4.3 Migration 44
2.5 Techniques 44
2.5.1 AST 44
2.5.2 MDRE 45
2.6 Aspect 45
2.6.1 Aspect Structurel 45
2.6.2 Aspect Comportemental 45
2.6.3 Aspect Métier 45
2.7 Les sorties des approches de migration 45

2.8 Validation des approches de migration 45
3 Application des critères de classification sur les approches existantes de migration des interfaces graphiques 46
3.1 Les entrées nécessaires des approches de migration GUI 46
3.2 Le Processus des approches de migration GUI 47
3.3 Les sorties des approches de migration GUI 49
3.4 La validation des approches de migration GUI 49
4 Conclusion 50

1 Introduction

Il est à noter que ce qui distingue, principalement, les applications Web côté client et les applications côté serveur est l'interface Web (l'interface GUI web). Cette caractéristiques nous permet de fixer notre objet d'étude par rapport aux travaux connexes : migration des applications possédant des interfaces graphiques (GUI).

Nous proposerons dans ce chapitre une classification de ces travaux en se basant sur un certain nombre de critères, illustrés dans la page suivante, tels que : leurs objectifs, leurs entrées, leur processus, leur sorties et enfin leur démarche de validation.

L'objectif des approches de migration des interfaces graphiques est de reproduire automatiquement/semi-automatiquement les interfaces graphiques d'une application dans un nouvel environnement autres que ceux dans lesquels elles ont été initialement mises en œuvre. Les nouvelles interfaces doivent être fonctionnelles dans le nouvel environnement avec un minimum d'interactions humaines sur les résultats produits.

1.1 Les entrées nécessaires pour les approches de migration

1.1.1 Code source

Le code source est l'ensemble des instructions décrivant l'exécution d'un système (Harman 2010). Il est écrit dans un langage de programmation tels que C, C++, Java, Python. Le code source est utilisé pour effectuer des actions exécutées par un ordinateur. Le code source est l'artefact le plus utilisé par les approches de migration existantes car il est une ressource primordiale de connaissance sur un système donnée (Raibulet, Arcelli Fontana et Zanoni 2017). Le code source peut être regroupé comme un ensemble de fonctions utilitaires afin de pouvoir être utilisées sans avoir à les réécrire d'où il est nommé *bibliothèque*.

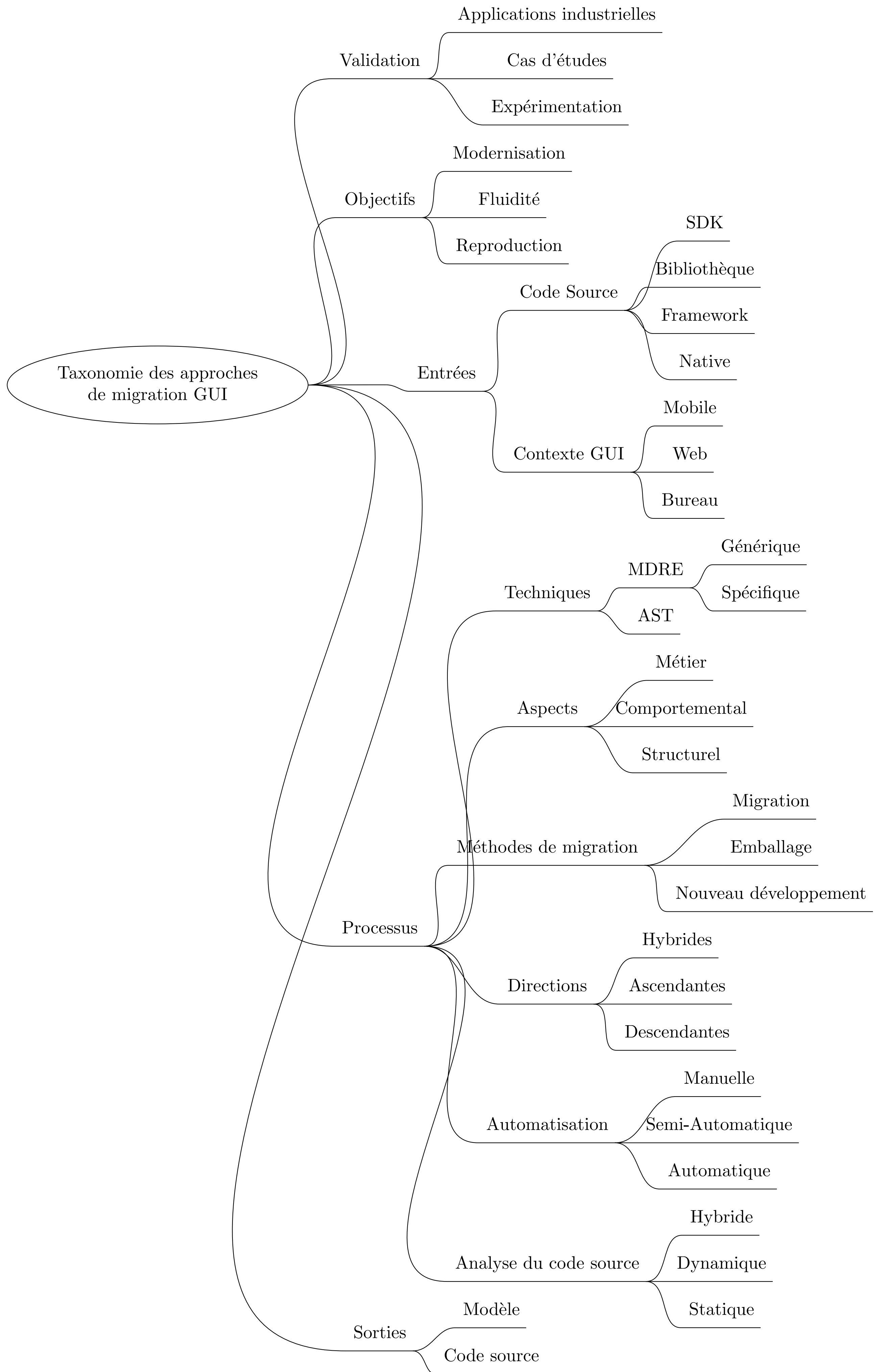
Le code source peut avoir aussi une structure de support conceptuelle et technologique définie, généralement, avec des artefacts logiciels ou des modules spécifiques, qui peuvent servir de base à l'organisation et au développement de logiciels d'où il est nommé *Framework* comme les frameworks Web coté client : VueJS [1], Angular [2], etc.

1.1.2 Contexte du GUI

La plupart des applications logicielles modernes orientées utilisateur sont centrées sur l'interface graphique et reposent sur des interfaces utilisateur (UI) attrayantes et des expériences utilisateur intuitives (UX) pour attirer les clients, faciliter l'exécution efficace des tâches informatiques et engager les utilisateurs.

Les interfaces utilisateur graphiques sont largement répandues et présentes principalement dans les ordinateurs tels que les applications bureaux, mobiles, Web ou nous les appelons des pages Web. Aujourd'hui, diverses actions sont effectuées via des éléments d'interface graphique, par exemple des buttons, des fenêtres, des menus et des icônes et d'autres élément (Jylhä et Hamari 2020).

1. https ://vuejs.org/
2. https ://angular.io/



Taxonomie des approches de migration GUI
Validation
Applications industrielles
Cas d'études
Expérimentation
Objectifs
Modernisation
Fluidité
Reproduction
Entrées
Code Source
SDK
Bibliothèque
Framework
Native
Contexte GUI
Mobile
Web
Bureau
Processus
Techniques
MDRE
Générique
Spécifique
AST
Aspects
Métier
Comportemental
Structurel
Méthodes de migration
Migration
Emballage
Nouveau développement
Directions
Hybrides
Ascendantes
Descendantes
Automatisation
Manuelle
Semi-Automatique
Automatique
Analyse du code source
Hybride
Dynamique
Statique
Sorties
Modèle
Code source

2 Le processus des approches de migration

2.1 Analyse du code

2.1.1 Analyse Statique

L'analyse statique du code consiste à analyser le code source et à extraire des informations telles que les relations entre les classes comme l'héritage, invocation des méthodes, accès aux variables. L'avantage de base de l'analyse statique est qu'il n'exécute pas le code de l'application analysée (Gosain et Sharma 2015) . Cependant, il ne traite pas le cas du polymorphisme, code inutilisé et affectation dynamique.

2.1.2 Analyse Dynamique

L'analyse dynamique consiste à analyser le code source d'un système au moment d'exécution. Elle explore les états du système en faisant des actions sur le système pour extraire les informations. Le problème d'analyse statique du polymorphisme et le code inutilisé est résolu par la stratégie dynamique. Cependant, si une action est perdue, l'ensemble des états ne sera pas couvert ce qui affecte la non exécution de classes correspondantes (Marek et al. 2015).

2.1.3 Analyse Hybride

L'analyse hybride est la combinaison de l'analyse statique et dynamique (Tsutano et al. 2019). L'avantage est de réduire la limitation de chaque type d'analyse lorsqu'il est appliqué séparément. Par exemple, pour identifier toutes les dépendances dans un code source orienté objet, l'analyse statique peut servir à identifier les dépendances statiques telles que les invocations des méthodes et l'héritage. D'autre part, l'analyse dynamique peut servir à identifier les entités qui sont utilisées au moment d'exécution.

2.2 Automatisation

L'automatisation du processus de migration correspond à combien une approche s'appuie sur l'interaction des experts du domaine. nous distinguons trois types d'automatisation : complètement automatique, semi-automatique et manuelle.

2.2.1 Automatique

Les approches entièrement automatisées ne nécessitent pas l'interaction des experts. Une approche de migration complètement automatique (100%) est l'un des défis dans l'évolution de logiciel, pour cela dans notre étude nous considérons que les approches qui n'ont pas un effet élevé dans leurs résultats sont automatique.

2.2.2 Semi-Automatique

Les approches semi-automatique nécessitent des interactions des experts sur les entrées ou les sorties pour compléter ou corriger les informations extraites automatiquement. Les futures activités de ré-ingénierie devraient intégrer les retours des experts pour produire automatiquement les résultats en utilisant l'apprentissage automatique, méta-heuristique et l'intelligence artificielle (Canfora et Di Penta 2007).

2.2.3 Manuelle

Les approches manuelles nécessitent massivement l'interaction des experts pour identifier les éléments d'entrées et les transformer manuellement en sorties. Généralement, les approches manuelles supportent le codage en dur pour effectué une telle migration.

2.3 Directions du processus

Le processus des approches de migrations des interfaces graphiques peut s'effectuer dans trois directions. Ce sont des directions descendantes, ascendantes et hybrides (Wagner 2014)

2.3.1 Descendante

Des directions qui utilisent les connaissances concernant le domaine du logiciel tels que les spécifications, la documentation sur des fonctionnalités pour établir des correspondances avec le code source. Dans ce cas le domaine du logiciel est le niveau haut d'abstraction. En revanche, le code source est considéré comme état le niveau bas (implémentation) .

2.3.2 Ascendante

Ce type commence par le code source pour extraire les informations, les structures et autres artefacts qui sont nécessaires pour former une représentation abstraite à un niveau supérieur.

2.3.3 Hybride

C'est la combinaison des deux méthodes (Descendantes, Ascendantes). (Wagner 2014) a donner un exemple ou une approche de migration de logiciel à reposer sur l'approche descendante pour la compréhension du système, son architecture et ses dépendances. Après pour une analyse détaillée d'un programme, l'approche ascendante est essentielle.

2.4 Méthodes de Migration

2.4.1 Nouveau développement

Une stratégie qui consiste à ré-implémenter complètement le logiciel hérité à partir du zéro, en parallèle au système existant. Elle est désignée dans la littérature sous le nom *Big Bang* ou *Cold Turkey*.Cette méthode n'est pas préconisé par les chercheurs, par exemple Martin Fowler a dit : « la seule chose qu'une réécriture du big bang garantit est un big bang ! » (KAPOOR 2021). Elle est associée à des coûts considérables et au risque que certaines parties de l'entreprise ne soient pas opérationnelles pendant la phase de transition entre les deux logiciels.

2.4.2 Wrapping

Cette méthode (en français "L'emballage") permet de transformer que des parties du système d'origine. L'avantage est de pouvoir réutiliser des composants logiciels connus et testés depuis des années. Ce type de modernisation ressemble à une boîte noire car une simple connaissance des interfaces externes de l'ancien logiciel suffisante d'où la fabrication des interfaces de connexion (API) est nécessaire. Cependant, cette méthode n'a qu'un caractère de solution à court terme. En effet, les problèmes de base ne sont pas nécessairement résolus comme la persistance des données, de la synchronisation, ainsi que de la gestion des exceptions et des erreurs(WAGNER 2014).

2.4.3 Migration

La migration des systèmes logiciels est un équilibre entre les deux méthodes mentionnées ci-dessus. Elle déplace le système existant dans un environnement flexible tout en maintenant sa fonctionnalité. Ce type est considérée comme une modernisation à boite blanche car la révision fondamentale du logiciel à moderniser. Néanmoins, cette méthode est orientée vers la durabilité, ce qui est plus complexe que le Wrapping, mais moins risquée qu'un nouveau développement(Big Bang)(WAGNER 2014).

2.5 Techniques

2.5.1 AST

Cette technique consiste à utiliser directement l'Arbre Abstraite Syntaxique (AST) pour effectuer l'activité de la migration. Des outils techniques sont utilisées pour permettre l'analyse comme JDT [3] , Spoon [4].

3. JDT : Une vue pour visualiser l'AST (arbre de syntaxe abstraite) d'un fichier Java ouvert sous l'éditeur. L'objectif est de naviguer de la sélection de texte aux nœuds AST et des nœuds aux sélections. `https://www.eclipse.org/jdt/ui/astview/index.php`

4. Spoon : Spoon est une bibliothèque open source pour analyser, réécrire, transformer le code source Java. Il analyse les fichiers sources pour créer un AST bien conçu avec une puissante API d'analyse et de transformation. Il prend entièrement en charge les versions Java modernes jusqu'à Java 11, 12, 13, 14. Spoon est un projet open source officiel d'Inria `https://spoon.gforge.inria.fr/`

2.5.2 MDRE

Lors du processus de migration, certains approches peuvent se base sur les modèles. Elles utilisent les modèles au lieu d'utiliser directement l'arbre syntaxe pour faciliter la représentation du code source et comprendre rapidement les éventuelles problèmes. le type des modèles peut être générique dans le cas l'approche base sur des standards OMG, ou spécifique dans leur propre contexte.

2.6 Aspect

2.6.1 Aspect Structurel

L'aspect structurel décrit l'aspect visuel de l'interface graphique tels que les éléments utilisées dans les interfaces.Il définit les caractéristiques des éléments d'interfaces graphiques, comme leur couleur et leur taille. Il décrit également la position des composants les uns par rapport aux autres.

2.6.2 Aspect Comportemental

L'aspect comportemental définit le flot des actions , des navigations et des évènements qui est exécuté lorsqu'un utilisateur interagit avec l'interface graphique. Le code comportemental contient généralement une logique décrit par les structures de contrôle (boucle et alternative).

2.6.3 Aspect Métier

L'aspect métier représente les activitées qui conforment au domaine d'application. Il comprend la persistance de données spécifiques à l'application.

2.7 Les sorties des approches de migration

Les sorties des approches de migration peuvent être diviser en deux type :

— Code : est l'artefact textuel qui représente le résultat de migration. Il peut exécuté sur des plateformes cibles (c.-à-d., code source, byte-code, ou code machine).

— Modèle : représente une code migré avant sa génération. Il ne peuvent pas être compilés ou exécutés.

2.8 Validation des approches de migration

Pour voir la faisabilité des approches de migrations des interfaces graphiques, nous aimerons voir qu'est ce qu'ils sont utilisé pour valider la convivialité de ces processus. Nous fixons trois :

— Cas d'étude : représente un particulier ou spéciale. il est utilisée pour effectué une étude qualitative sur un thème.

— Expérimentation : sert à tester les contraintes définis par les approches pour établir une étude quantitative sur un thème
— Industrie : représente un variant de cas d'étude mais il est utilisé par les approches dans le contexte des applications des entreprises.

3 Application des critères de classification sur les approches existantes de migration des interfaces graphiques

Dans cette section, nous discutons des résultats de notre classification sous forme des tables. Les résultats sont organisés en fonction de l'entrée, du processus, de la sortie des approches d'identification et des méthodes de validation des approches.

3.1 Les entrées nécessaires des approches de migration GUI

Toutes les approches existantes se sont appuyées sur le code source comme principal artefact à analyser, de sorte que 53,3% (en fonction du nombre d'approches) des approches traitent la migration des applications qui ont été développées avec des Frameworks dont 62,5% à un contexte sur le web et 25% pour bureau et 12,5 pour mobile, tandis que 40% des approches ne dépendent que des applications qui se sont implémentées utilisant un code native. 83,3% des ces approches traitent que le développement web et 33,3% traitent que les applications bureau. Une approche a traité le cas de la migration d'une bibliothèque web et une autre a abordé le cas de la migration des SDK destinée au développement web. La table 1 ci-dessous présente la classification selon les entrées nécessaires pour les approches de migrations des interfaces graphiques.

Approche	Code Source				Contexte du GUI
	Native	Bibliothèque	Framework	SDK	
(Bünder 2019)			Swing		web
(Trias 2015)	PHP				web
(J. C. Silva 2010)			Swing		bureau
(Morgado 2012)	Java				bureau/web
(Thilanka 2021)			AngularJS		web
(M. Samir 2016)			Swing		bureau
(Hayakawa 2014)	JS/Ajax		Flash		web
(Abderrahmane Seriai 2019)				GWT	web
(Sánchez R 2014)	Oracle Forms				bureau
(Heil 2018)	C/C++				web

(Shah 2011)			Swing		mobile
(H. Samir 2007)			Swing		web
(Ducasse 2019)			Spec		web
(Kusumoto 2018)		jQuery			web
(El Boussaidi 2016)	Html/Css/Js				web

Table 1 – Les entrées des approches des migrations GUI

3.2 Le Processus des approches de migration GUI

Le processus appliqué par les approches de migration des interfaces graphiques a de nombreuses dimensions : type d'analyse, l'automatisation, la direction, l'aspect, méthode et technique. Toutes les approches de migration entièrement automatisées sont relativement préférées aux approches semi-automatiques ou manuelles. 86,7 % des approches sont entièrement automatiques, tandis que 13,3% sont semi-automatiques. La sélection d'une direction de processus est basée sur les artefacts d'entrée. Si l'entrée est le code source, un processus ascendant est appliqué avec un type d'analyse statique. C'est le cas de 86,7% des approches. Presque toutes les approches traitent l'aspect structurel et comportemental des interfaces graphiques (80%) et d'autres pour l'aspect métier des interfaces. Deux approches qui ont utilisé la méthode d'emballage pour migrer qu'une partie de l'application qui est représentée par l'interface graphique tandis que d'autres utilisent directement une migration technologique. D'autre part, 80% des approches utilisent des modèles lors de la migration et d'autres abordent l'arbre abstraite syntaxique pour l'effectuer.

La table 2 ci-dessous présente la classification selon les éléments de processus que nous avons établi.

Approche	Analyse du code			Auto			Direction			Aspect			Méthode			Technique	
	stat	dyna	hybr	auto	saut	man	des	asc	hbr	str	com	mét	dev	war	mig	ast	mdl
(Bünder 2019)	✓			✓				✓		✓	✓			✓		✓	✓
(Trias 2015)	✓			✓				✓							✓		✓
(J. C. Silva 2010)	✓			✓				✓			✓					✓	✓
(Morgado 2012)		✓		✓				✓		✓	✓						✓
(Thilanka 2021)	✓			✓				✓		✓	✓				✓	✓	
(M. Samir 2016)			✓	✓				✓		✓	✓	✓					✓
(Hayakawa 2014)	✓			✓				✓		✓	✓				✓		✓
(Abderrahmane Seriai 2019)	✓			✓				✓		✓	✓				✓		✓
(Sánchez R 2014)	✓			✓				✓		✓	✓	✓			✓		✓
(Heil 2018)	✓				✓			✓		✓	✓	✓		✓			
(Shah 2011)		✓		✓			✓			✓	✓	✓			✓		✓
(H. Samir 2007)		✓		✓			✓			✓	✓				✓		✓
(Ducasse 2019)	✓			✓				✓		✓	✓	✓					✓
(Kusumoto 2018)	✓				✓			✓							✓	✓	
(El Boussaidi 2016)	✓			✓				✓		✓	✓	✓				✓	

Table 2 – Processus des approches de migration GUI

3.3 Les sorties des approches de migration GUI

D'après la table 3, 73,3% des approches génèrent un code source exécutable de l'application qu'était traiter. Par contre, 26,7% des approches génèrent un modèle d'application ce explique que ces approches n'effectuent pas une migration complète, mais seulement en partie par exemple, les travaux présentés par (El Boussaidi 2016),(M. Samir 2016) n'abordent pas le processus complet d'une migration d'interface graphique.

Approche	Code	Modele
(Bünder 2019)	✓	
(Trias 2015)		✓
(J. C. Silva 2010)	✓	
(Morgado 2012)	✓	
(Thilanka 2021)	✓	
(M. Samir 2016)		✓
(Hayakawa 2014)		✓
(Abderrahmane Seriai 2019)	✓	
(Sánchez R 2014)	✓	
(Heil 2018)	✓	
(Shah 2011)	✓	
(H. Samir 2007)	✓	
(Ducasse 2019)	✓	
(Kusumoto 2018)	✓	
(El Boussaidi 2016)		✓

Table 3 – Les sorties des approches de migration GUI

3.4 La validation des approches de migration GUI

Afin de voir la faisabilité des approches de migrations GUI, 46,7% abordent des cas d'étude, par exemple, (H. Samir 2007) teste sur qu'une simple application développée en Swing. 20% ont pris un échantillon des applications pour tester leurs heuristiques de migrations comme (Heil 2018). Autres approches teste sur des applications industrielles par exemple (Ducasse 2019) et (Abderrahmane Seriai 2019). La table 4 ci-dessous présente la classification selon les méthodes de validation .

Approche	Cas d'études	Expérimentation	Applications industrielles
(Bünder 2019)			
(Trias 2015)	✓		
(J. C. Silva 2010)	✓		
(Morgado 2012)	✓		
(Thilanka 2021)			✓
(M. Samir 2016)	✓		
(Hayakawa 2014)		✓	
(Abderrahmane Seriai 2019)			✓
(Sánchez R 2014)			✓
(Heil 2018)		✓	
(Shah 2011)	✓		
(H. Samir 2007)	✓		
(Ducasse 2019)			✓
(Kusumoto 2018)	✓		
(El Boussaidi 2016)		✓	

Table 4 – Validations des approches de migration GUI

4 Conclusion

Il est important de positionner notre problématique dans l'espace de recherche des approches de migration des interfaces graphiques présenté précédemment. Notre contexte est les applications web, en particulier la migration coté front en gardant dans le même contexte. Nous avons pu synthétiser les approches précédentes :

— Presque toutes les approches proposées focalisent sur l'automatisation du processus de migration afin d'introduire la simplicité de l'activité. Cela peut rendre leurs approches accessibles. Cette automatisation ne veut pas dire qu'elles sont parfaites, elles nécessitent l'interaction humaine pour valider leurs résultats.

— Nous pouvons dire qu'il y a un facteur commun entre l'analyse statique et la direction ascendante du processus de migration. Nous pouvons dire également que la direction ascendante est difficile pour être automatisée, celle-ci nécessite des techniques avancées comme l'intelligence artificielle et méta-heuristiques pour être automatisé.

— La stratégie *Big Bang* (nouveau développement) n'a été jamais utilisé par les approches de migration ce qui prouve qu'elle est coûteuse et risquée. La plupart abordent la méthode de migration technologique.

— L'arbre abstraite syntaxique reste toujours la vraie source d'informations sur le code source d'un logiciel car les modèles sont des représentations simplifies du code source. Pour produire les modèles d'application, il nécessaire d'établir un mécanisme de transformation de AST vers un modèle d'application.

— Plusieurs approches utilisent les modèles pour représenter l'interface graphiques de l'application source sous format d'un méta-modèle, par exemple Mbarki M. SAMIR 2016 et Sánchez Ramán et al SÁNCHEZ R 2014 ont crée un méta-modèle pour leur interfaces graphiques en adaptant le méta-modèle KDM. L'adptation du méta-modèle KDM comprend l'ajout d'une entité Attribute et Widget qui peut réprésenté les éléments d'une interface tels que Button, Label, Panel, ,etc. D'autres n'ont pas détaillé leur méta-modèle utilisé par exemple Morgado et al MORGADO 2012 a décrit que son méta-modèle est similaire au DOM. Donc, nous distinguons deux types de modèle : générique et spécifique.

— Trois parmi quinze des approches abordent la migration du côté client d'une application web vers les technologies web actuelles (React, VueJS, Angular, EmberJS). Celle-ci rencontrent des défis de migrations des architectures dont les applications source ont été développées.

Deuxième partie

Contribution

CHAPITRE 5

APPROCHE PROPOSÉE

Sommaire

1 **Introduction** **55**
2 **Vue globale de l'approche proposée** **55**
2.1 Contraintes 55
2.2 Entrées 56
2.3 Processus 56
2.3.1 Construction du méta-modèle de la technologie Web héritée 56
2.3.2 Extraction des modèles 56
2.3.3 Transformation du modèle KDM à un modèle GWT . 56
2.3.4 Construction du graphe de navigation 57
2.3.5 Construction d'un modèle intermédiaire 57
2.3.6 Génération de l'application cible 57
2.4 Sorties 57
3 **Cas d'étude : Migration de la partie client d'une application Web GWT vers une application Web en Angular** **59**
3.1 Technologies GWT et Angular 59
3.1.1 GWT (Google Web Toolkit) 59
3.1.2 Angular 62
3.2 Application de l'approche 63
3.2.1 Construction du méta-modèle de GWT 63
3.2.2 Extraction des modèles d'une application GWT . . . 67
3.2.3 Transformation du modèle KDM vers un modèle GWT 68
3.2.4 Construction du modèle de navigation 72
3.2.5 Construction d'un modèle intermédiaire 81
3.2.6 Génération du code Angular 83
4 **Conception** **88**
4.1 Conception architecturale 88
4.2 Conception détaillée 89
4.2.1 Diagramme de classes 89

5 Conclusion 101

1 Introduction

Nous rappelons que l'objectif principal de notre travail est la proposition d'une nouvelle approche basée sur l'ingénierie dirigée par les modèles pour la migration des applications Web héritées (legacy Web applications). Nous traitons dans notre travail seulement la partie client des applications Web héritées.

Dans ce chapitre, nous présentons une vue globale de l'approche dans laquelle nous citons les entrées, le processus, les sorties et des contraintes de l'approche. Ensuite, nous détaillons chaque étape de l'approche. Puis, nous étudions le cas d'une migration d'une application GWT vers Angular.

2 Vue globale de l'approche proposée

À partir de l'état de l'art, en particulier, les travaux connexes, nous avons conçu notre approche pour la migration des applications Web héritées spécialement pour le côté client (front-end). Particulièrement, nous intégrons le modèle KDM utilisé dans l'initiative ADM (voir la section 15) proposé par l'OMG pour rendre cette approche conforme à cette norme. La Figure 22 représente une vue globale sur notre approche.

2.1 Contraintes

Pour que notre approche puisse faire partie de l'ensemble des approches MDRE (Model Driven Reverse Engineering), elle doit fournir les caractéristiques suivantes afin de répondre aux exigences exprimées :

— *Généricité* : l'approche proposée doit utilisée le méta-modèle KDM qui est un modèle indépendant de la technologie et conforme au standard OMG afin de faciliter la compréhension de l'application source d'une manière universelle (Une approche basée sur des standards).

— *Extensibilité et maintenbilité* : l'approche proposée doit s'appuyer sur un découplage des informations représentées par les modèles, c'est-à-dire la capacité d'ajouter facilement des nouvelles étapes dans le processus de l'approche (maintenabilité) qui accepteront des modèles de différents méta-modèles sans subir des modifications sur d'autres étapes de l'approche.

— *Automatisation* : le processus de migration de l'approche doit être automatique, c'est-à-dire, le développeur étant le client ne peut pas interagir au moment d'exécution du processus de migration. Il peut effectuer des modifications simples sur les résultats obtenus.

— *Réflexivité* : le processus de migration peut avoir des exigences du développeur (client) pour migrer l'application source. Cela rend notre approche auto-adaptable au moment d'exécution de ces étapes.

— *Réutilisabilité et intégration* : les différents éléments et résultats obtenus tels que les modèles doivent être conçus pour être réutilisés. De plus, Il faut être capable d'intégrer facilement les modèles réutilisables obtenus.

2.2 Entrées

Les entrées que nous avons sélectionnées pour notre approche sont présentées dans la Figure 22 :

— Infrastructure de développement de l'application source qui peut être : une bibliothèque, kit de développement (SDK) ou framework.
— Le code source de l'application Web héritée qui représente la vraie ressource de connaissance.
— Un modèle client est un modèle qui résume les exigences du client, par exemple, il indique les vraies pages à migrer et la configuration de l'environnement de transformation des modèles.
— Méta-modèle de la technologie cible : est une entrée optionnelle car elle dépend du type du modèle intermédiaire que nous allons l'expliquer dans le processus de l'approche.

2.3 Processus

Le processus globale de l'approche comprend six étapes, illustrées dans la Figure 22 :

2.3.1 Construction du méta-modèle de la technologie Web héritée

À partir de l'infrastructure de l'application cible qui peut être une bibliothèque, un SDK ou un framework, nous construisons un méta-modèle KDM.

Tout d'abord nous analysons manuellement la partie qui contient l'ensemble des éléments graphiques qui peuvent être situés dans des paquets (package) nommés UI, user-interface, client-side, etc. Ensuite, nous extrayons ces éléments pour les intégrer dans le méta-modèle KDM d'où nous construisons un méta-modèle KDM adapté pour la technologie. Cette étape est nécessaire pour spécialiser la migration à une technologie particulière.

2.3.2 Extraction des modèles

À partir du code source de l'application source, nous extrayons deux modèles : 1) un modèle KDM décrivant l'aspect structurel de l'application source. 2) un modèle d'application conforme au méta-modèle ASTM décrivant l'analyse syntaxique de l'application. Cette extraction est faite par un découvreur de modèles.À titre exemple l'utilisation de Modisco (voir section 4.4).

2.3.3 Transformation du modèle KDM à un modèle GWT

Cette étape consiste à transformer le modèle KDM de l'application source à un modèle nommée *modèle de domaine* conforme au nouveau méta-modèle construit dans la première étape du processus. Cette transformation permet de faciliter la représentation des concepts de la technologie héritée.

2.3.4 Construction du graphe de navigation

Dans les applications Web, toutes les pages sont accessibles en fonction d'un texte qui correspond aux liens et aux paramètres nommé URL [1].

Donc, dans cette étape nous construisons un modèle qui décrit les transitions entre les pages de l'application à partir du nouveau modèle construit dans l'étape précédente et le modèle ASTM (application). La construction de ce modèle peut nécessiter des heuristiques car parfois l'infrastructure de l'application source n'implémente pas clairement le principe de navigation entre les pages Web, elle repose dans ce cas sur la programmation événementielle ou elle ne possède pas un patron de navigation bien défini.

2.3.5 Construction d'un modèle intermédiaire

À partir du modèle d'application, modèle du domaine et du modèle de navigation, nous transformons ces modèle obtenus dans les étapes précédentes en à un seul modèle intermédiaire qui représente les connaissances obtenus sur l'application source tels que les pages, l'ensemble des éléments de chaque page, leurs positionnement, leurs styles, etc. Ce modèle peut avoir deux types :

— un modèle basique : contient une représentation simple des éléments de l'application source.

— un modèle KDM de l'application cible : ce modèle ressort d'une transformation des concepts de la technologie source vers la cible. Il est conforme au méta-modèle de la technologie cible. Ce modèle peut optimiser la tâche suivante ??? de la vue d'implémentation.

2.3.6 Génération de l'application cible

Une fois le modèle intermédiaire généré, il est possible de générer l'application à l'aide d'un générateur. Ce dernier transforme les éléments de l'application cible présentés dans le modèle intermédiaire à une application cible au dessus d'une technologie visée.

2.4 Sorties

La sortie du processus de migration est une application générée automatiquement. Elle est exécutable sur les différents navigateurs actuels supportés par les nouvelles technologies. Le client peut intervenir sur le code de l'application cible pour effectuer une modification mineure et de rendre par la suite son retour au développeur.

1. URL : son nom complét est "Uniform Resource Locator". Elle est une chaîne de caractères uniforme qui permet de localiser une ressource du web par son emplacement en utilisant un protocole internet.

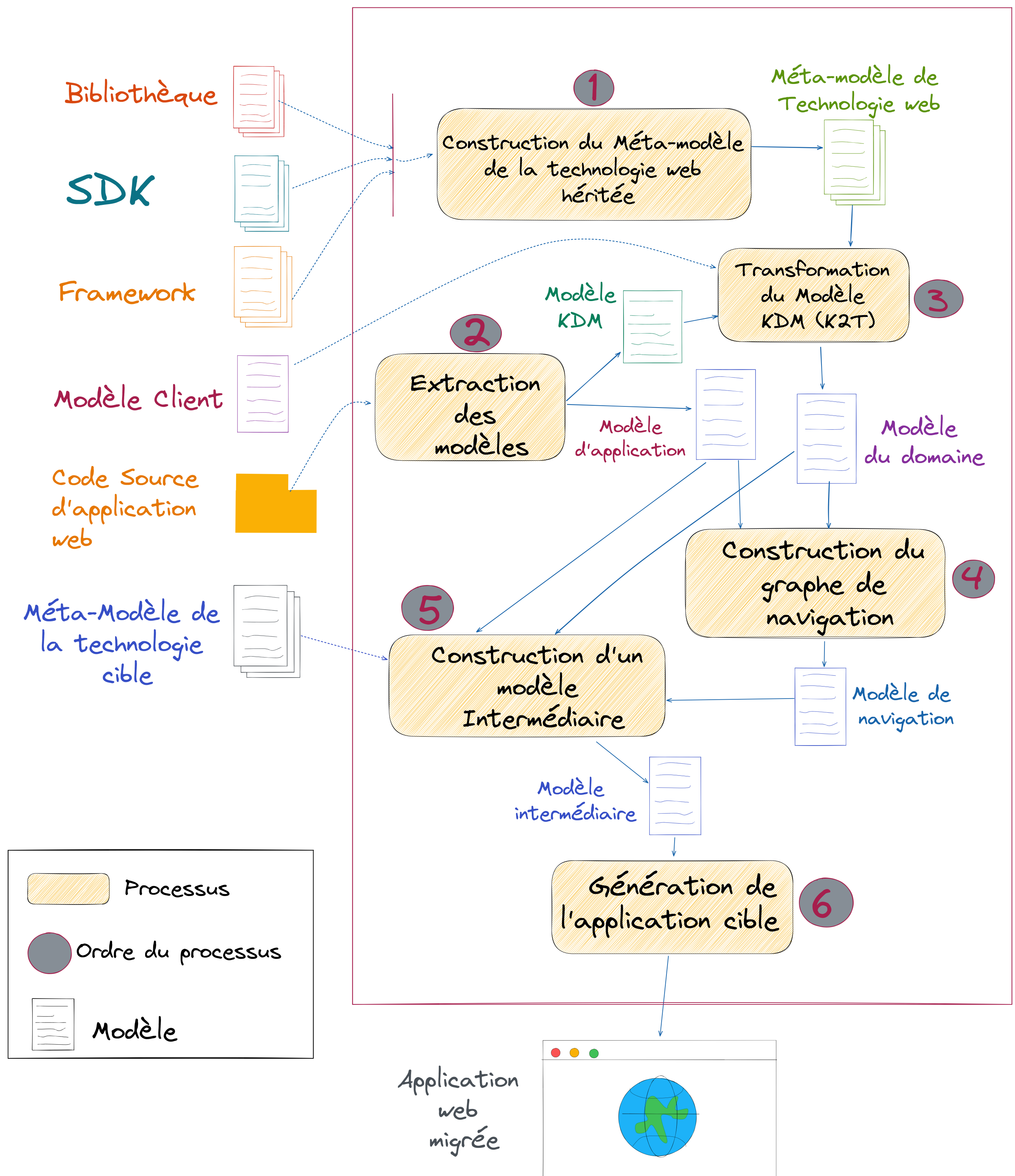


Figure 22 – Processus global de notre approche

3 Cas d'étude : Migration de la partie client d'une application Web GWT vers une application Web en Angular

Pour étudier la faisabilité de l'approche proposée, nous l'appliquons dans un cas d'étude où nous prenons une application Web développée en GWT comme application source, que la partie client (interface utilisateur), pour la migrer vers une application Web développée en Angular.

3.1 Technologies GWT et Angular

3.1.1 GWT (Google Web Toolkit)

GWT (Google Web Toolkit) est une boîte à outils open source de développement d'applications Web mettant en œuvre AJAX [2]. Le langage de développement utilisé par GWT est le Java. Le développeur peut également utiliser XML pour implémenter leur pages web comme le cas des applications mobile. Ce type est appelé UiBinder [3] en GWT. Le GWT adapte le style architecturale Client/Serveur pour développer une interface web (GWT p. d.).

Le client présente la partie graphique ou visuelle de l'application GWT. Elle se base sur HTML, CSS et de classes Java dans lesquelles des éléments graphiques fournis par SDK GWT comme : Button, Label, Menu et d'autres. Les éléments graphiques de GWT peuvent être utilisés avec des gestionnaires d'événements pour effectuer des actions des utilisateurs. La figure 24 représente une page nommée *Hello* en GWT qui contient un button lors de clique il renvoie un message "Hello, AJAX".

Le serveur s'occupe à la persistance de données et l'aspect métier de l'application. De plus, toute communication de cette couche est effectuée par le protocole RPC (Remote procedure call) Ce protocole permet facilement d'échanger entre des objets Java par HTTP entre le client et le serveur. Il s'agit simplement d'une méthode légère pour transférer les données. Le client utilise interface appelée service pour interagir avec le serveur afin d'effectuer un changement des appeles. Dans ce cas, le patron de conception Proxy [4] est impliqué au serveur.

GWT propose de nombreuses fonctionnalités pour développer une application web exécutable dans un navigateur en présentant des comportements similaires à ceux d'une application bureau (DOUDOUX. p. d.) :

— Le développeur n'a pas besoin d'utiliser le JavaScript pour développer son application Web. Il utilise seulement Java qui n'est pas compilé en bytecode mais traduit en

2. AJAX : AJAX (Asynchronous JavaScript + XML) n'est pas une technologie en soi, mais un terme désignant une « nouvelle » approche utilisant un ensemble de technologies existantes, dont : HTML ou XHTML, les feuilles de styles CSS, JavaScript, le modèle objet de document (DOM), XML, XSLT, et l'objet XMLHttpRequest. `https://developer.mozilla.org/fr/docs/Web/Guide/AJAX`

3. Plus de documentation sur UiBinder : http ://www.gwtproject.org/doc/latest/DevGuideUiBinder.html

4. Proxy : est un patron de conception structurel qui vous permet d'utiliser un substitut pour un objet. Elle donne le contrôle sur l'objet original, vous permettant d'effectuer des manipulations avant ou après que la demande ne lui parvienne.`https://refactoring.guru/fr/design-patterns/proxy`

JavaScript. Le cœur de GWT est composé du compilateur/traducteur de code Java en JavaScript optimisé. ce code produit est capable de s'exécuter sur les principaux navigateurs. La Figure 23 représentante l'idée d'exécution d'un code GWT.

— Le GWT offre un ensemble riche d'éléments graphiques (button, panels) similaires au Swing ou AWT [5] .

— Le GWT repose sur AJAX pour la communication avec le serveur grâce à des appels asynchrones en échangeant des objets Java.

— Le principe de transition entre les interfaces de l'application web basé sur un système de gestion de l'historique sur le navigateur.

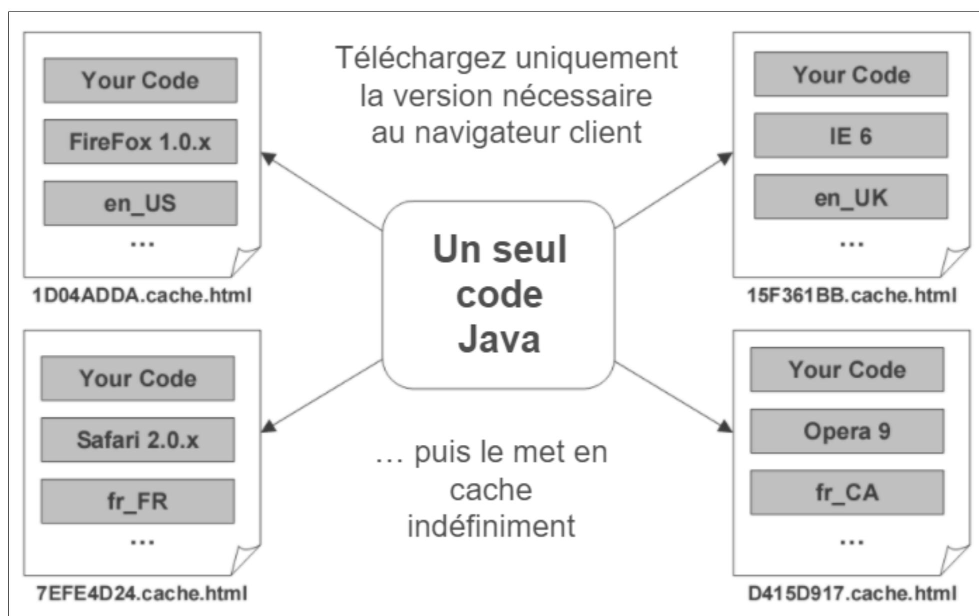
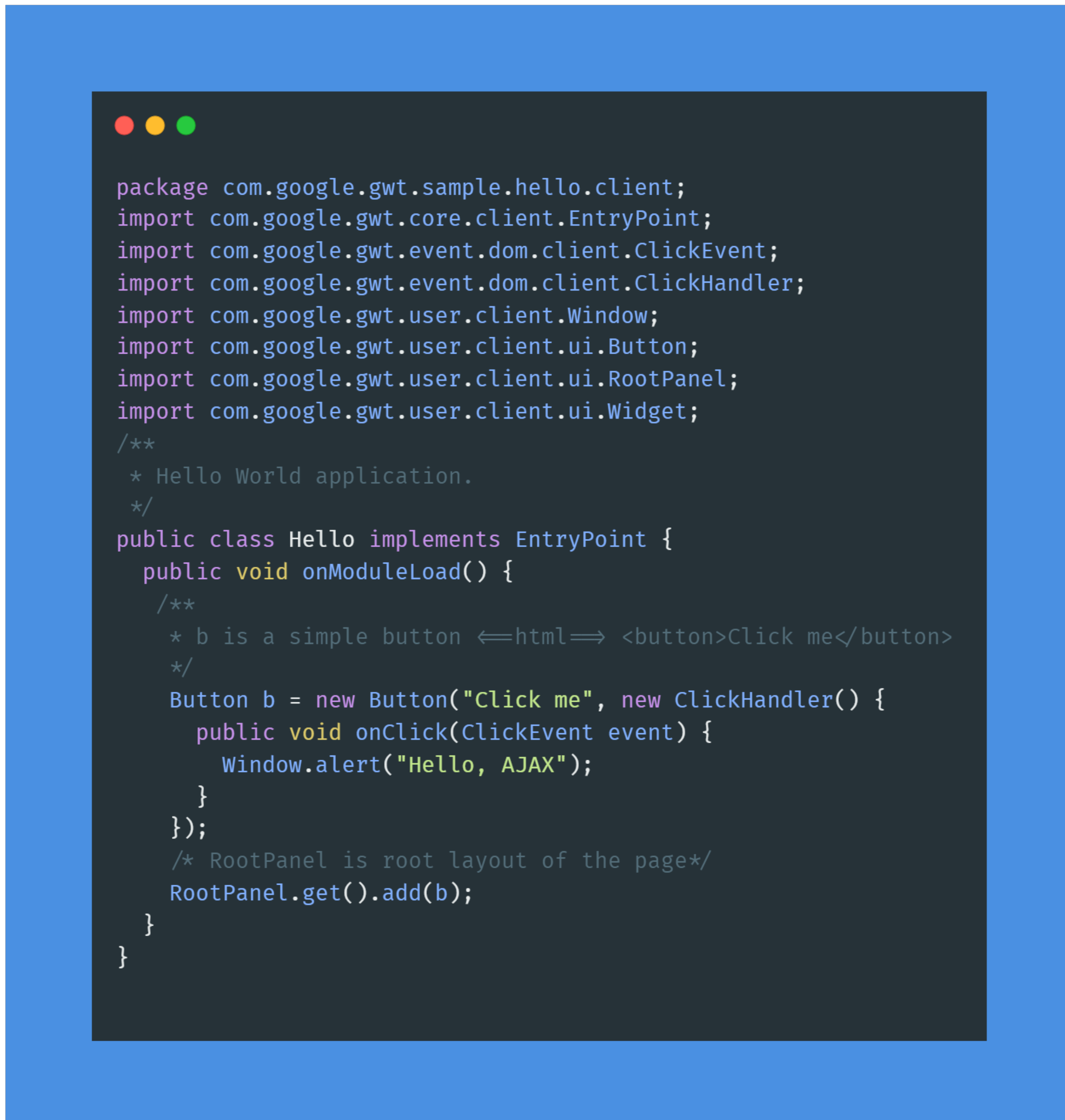


FIGURE 23 – Processus d'exécution d'un code GWT sur touts les navigateurs (JABER 2007)

5. AWT : ("Abstract Window Toolkit") est une bibliothèque graphique pour Java.



```
package com.google.gwt.sample.hello.client;
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.Widget;
/**
 * Hello World application.
 */
public class Hello implements EntryPoint {
  public void onModuleLoad() {
   /**
    * b is a simple button ⟸html⟹ <button>Click me</button>
    */
    Button b = new Button("Click me", new ClickHandler() {
      public void onClick(ClickEvent event) {
        Window.alert("Hello, AJAX");
      }
    });
    /* RootPanel is root layout of the page*/
    RootPanel.get().add(b);
  }
}
```

FIGURE 24 – Une partie du client GWT

L'idée de compilation derrière GWT est d'écrire une application en Java, de la déboguer en Java, et de transformer le code Java en Javascript. La phase de compilation commence par analyser le code java puis elle crée un arbre syntaxique AST. Puis, le compilateur s'occupe à optimiser l'arbre et la transformer en un code Javascript. Enfin, il s'occupe à optimiser le Javascript encore une fois (VALLÉE p. d.).

3.1.2 Angular

Angular est une plateforme de développement créée par Google en 2016. Il permet de créer des applications Web dynamiques pour fluidifier l'expérience utilisateur par le principe de SPA (Single Page Applications) [6] et d'éviter la surcharge de la cache du navigateur par les chargements de pages dès démarche de l'application en utilisant des techniques (ANGULAR 2021b).

Angular est écrit avec le langage TypeScript [7] . En tant que plate-forme, Angular comprends :

— Un framework basé sur des composants web qui sont des ensembles d'éléments graphiques pour la création d'applications Web évolutives.

— Une collection de bibliothèques bien intégrées qui couvrent une grande variété de fonctionnalités, notamment le routage, la gestion des formulaires, la communication client-serveur, etc.

— Une suite d'outils de développement pour aider les développeurs à implémenter, créer, tester et mettre à jour leurs codes.

Une application Angular a toujours au moins un module racine qui permet l'amorçage, et a généralement beaucoup plus de modules de fonctionnalités. Elle est définie par un ensemble de NgModules. Ce dernier collecte le code associé dans des ensembles fonctionnels nommée Module. Le code associé est nommée Composant (ANGULAR 2021a).

Les composants définissent des vues, qui sont des ensembles d'éléments d'écran parmi lesquels Angular peut choisir et modifier en fonction de la logique et des données du programme de développeur. De plus, les composants utilisent des services, qui fournissent des fonctionnalités spécifiques non directement liées aux vues. Les fournisseurs de services peuvent être injectés dans les composants en tant que dépendances, rendant le code modulaire, réutilisable et efficace.

6. SPA : est une application Web monopage dont la caractéristique est de n'avoir qu'une seule page. Lors de la navigation celle-ci n'est jamais rechargée, seules des portions de la page sont dynamiquement mises à jour à l'aide de code Javascript (ORANGE 2021).

7. TypeScript : est un langage de programmation développé par Microsoft qui garde la même syntaxe de Javascript et lui ajoute les types. https://www.typescriptlang.org/

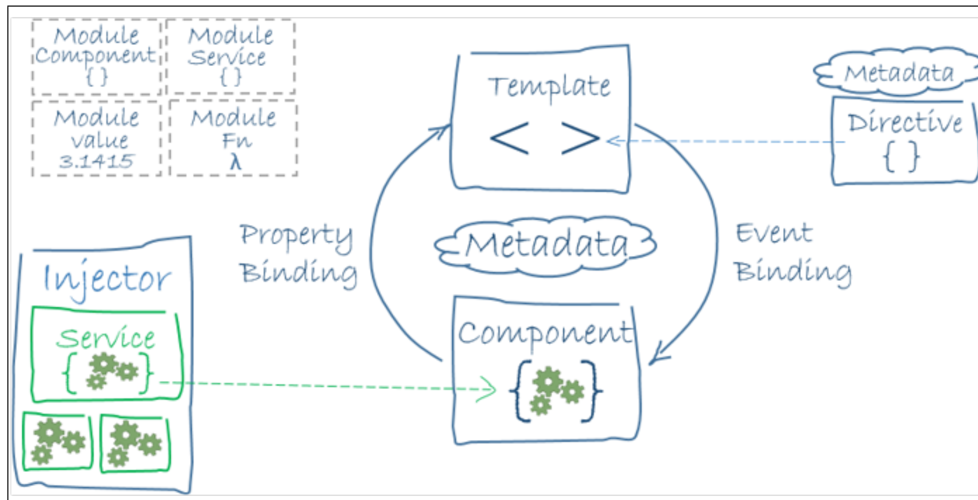


FIGURE 25 – Architecture Angular en générale (ANGULAR 2021a)

3.2 Application de l'approche

Dans ce qui suit, nous allons appliquer l'approche illustrée dans la figure 22 pour le cas de migration du GWT à Angular. Nous détaillons chaque étape de notre processus de migration. Nous discutons la solution pour les défis rencontrés et dans la partie conception nous donnons plus de diagramme pour concrétiser la solution. Il est à noter que nous traitons dans un premier temps que les applications GWT développées seulement avec Java (sans UiBinder).

3.2.1 Construction du méta-modèle de GWT

La partie client de GWT est conforme à un ensemble de pages Web contenants une hiérarchie des nœuds graphiques. Au meilleur de notre connaissance, il n'existe pas dans la littérature un méta-modèle complet qui peut représenter les concepts de la partie client de GWT. Nous suivons la démarche proposée, illustrée dans la figure 22.

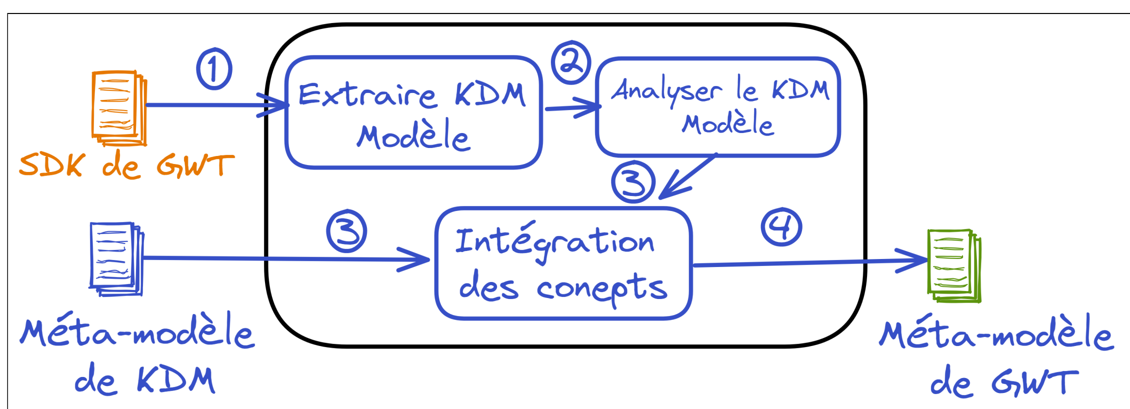


FIGURE 26 – Processus de construction du méta-modèle GWT

D'abord, nous faisons l'extraction du modèle KDM de SDK[8] de GWT en utilisant l'outil Modisco (présenté dans la Section 4.4). D'après la documentation officielle de GWT,

8. https://github.com/gwtproject/gwt/tree/master/user/src/com/google/gwt

l'élément parent des éléments graphiques de GWT est nommée *UIObject*[9].

Ensuite, nous cherchons dans le modèle KDM établie toutes les entités descendantes de *UIObject* à l'aide d'une fonction récursive qui parcourt l'arbre d'héritage des entités dans le modèle KDM. En effet, nous récupérons cette hiérarchie et nous pouvons éventuellement créer un diagramme qui peut représenter leurs hiérarchie, représentées dans les Figures 28 et 29. Les deux Figures illustrent que des parties d'ensemble totale (88 éléments de GWT).

Après la récupération des éléments, nous ajoutons la notion de GwtModel au méta-modèle KDM. GwtModel est un fils directe de CodeModel, présenté dans l'état d'art 17, pour couvrir les nouveaux concepts de GWT. GwtModel peut avoir au moins une page. Une page peut avoir plusieurs widgets. Le parent de tous les éléments est UIObject qui devient être un fils de ClassUnit. La Figure 27 représente la partie ajouté au méta-modèle KDM. cette partie fait le lien d'adaptation entre le méta-modèle KDM et la partie client de GWT.

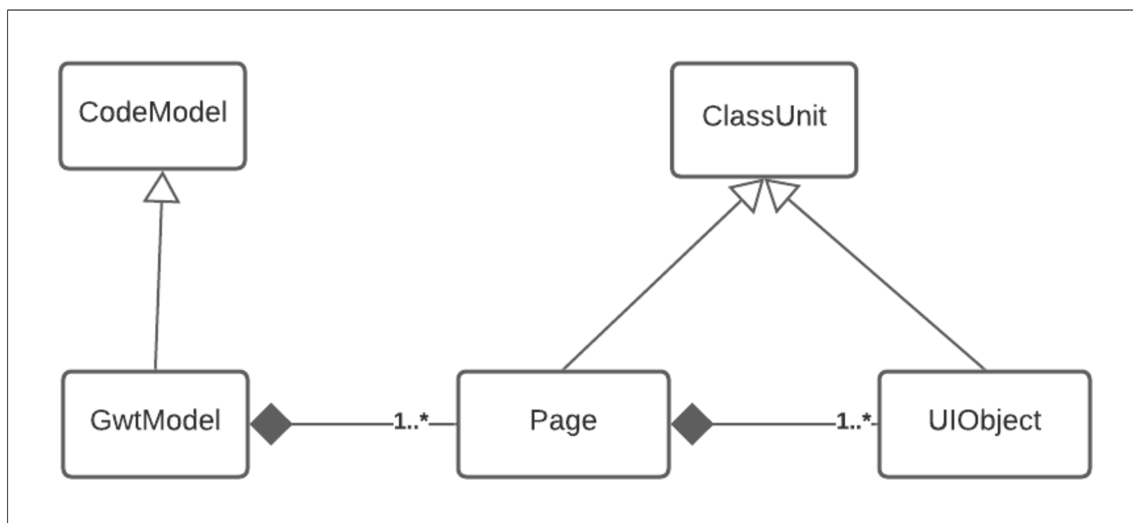


Figure 27 – Partie ajoutée au méta-modèle KDM

9. http://www.gwtproject.org/javadoc/latest/com/google/gwt/user/client/ui/UIObject.html

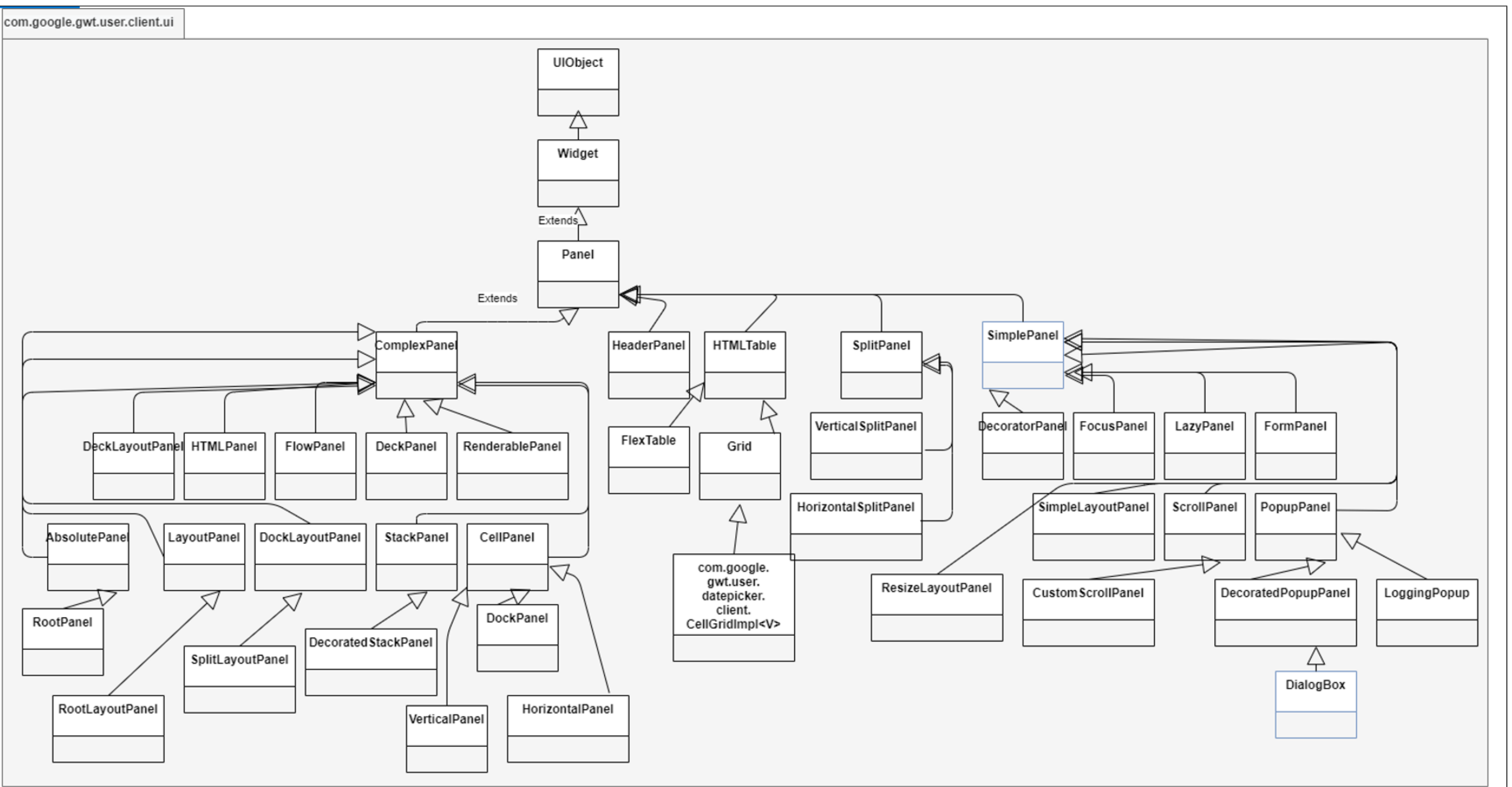


FIGURE 28 – Partie 1 extraite d'éléments graphiques de GWT

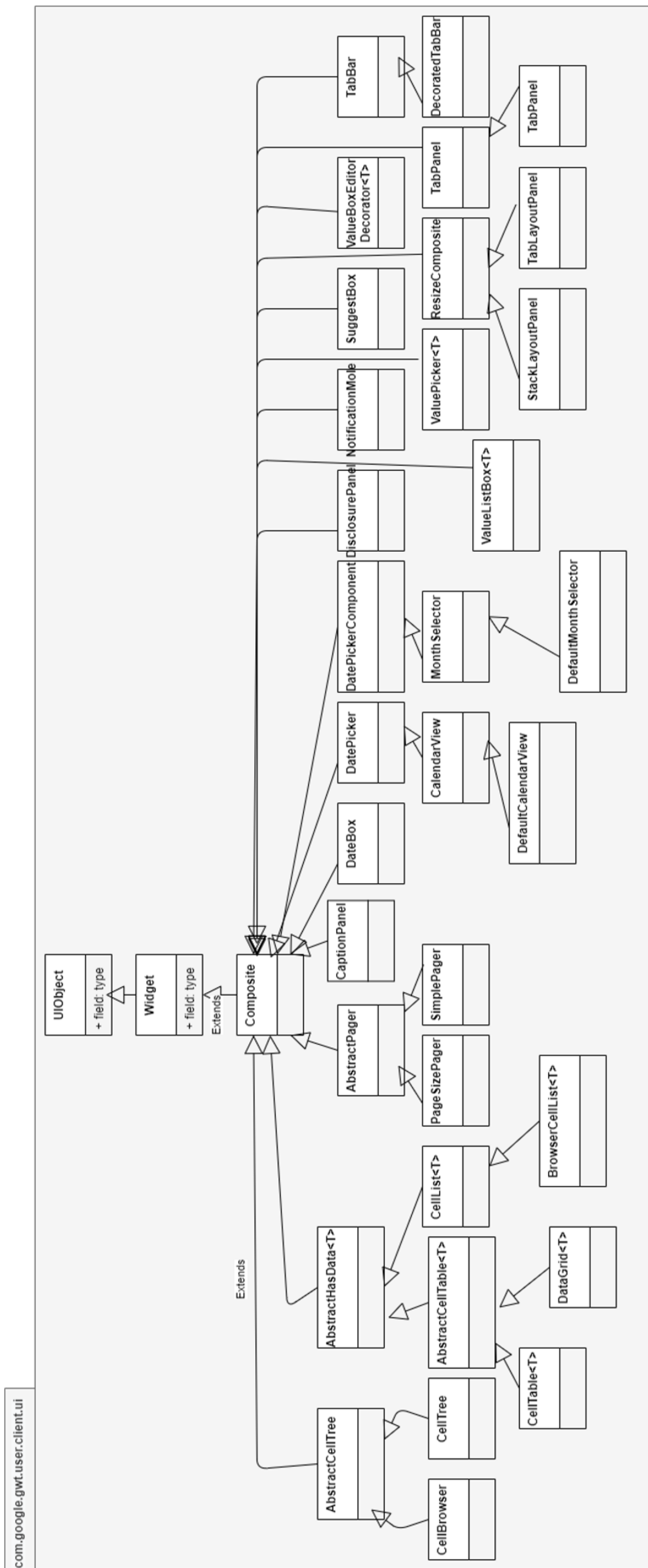


Figure 29 – Partie 2 extraite d'éléments graphiques de GWT

3.2.2 Extraction des modèles d'une application GWT

À partir d'une application GWT donnée, nous générons deux modèle : un modèle KDM et un modèle d'application qui est un modèle Java qui conforme au méta-modèle ASTM, présenté dans la Figure 22.

Pour extraire ces deux modèles, nous utilisons l'outil MoDisco. L'intégration de cet outil dans un projet autonome "standalone" est une tâche complexe car le MoDisco est une extension (plugin) d'Eclipse et il doit s'exécuter sur une instance d'Eclipse. Donc deux solution existent : 1) utilisation manuelle et 2) application de technique "headless application".

Une "headless application" est une instance du Eclipse lancée sans aucune interface utilisateur graphique à partir des lignes de commande seulement. Elle est utilisée pour intégrer les solutions sous Eclipse avec des applications externes. La façon à utiliser pour transformer un plugin a une "headless application" concentre sur la création d'un espace de travail d'Eclipse virtuel (virtual workspace) qui utilise les paquets propres à Eclipse en les exportant dans des fichiers JAR [10] précis en utilisant seulement l'outil d'extraction et d'exportation d'Eclipse (Christian 2012).

Pour l'instant, nous utilisons la première technique car elle est rapide et facile mais non automatique vu que la deuxième est une solution automatique mais elle prend un temps de réalisation. Celle-ci, nous la notons dans la partie des travaux futurs à réaliser. Les Figures 30, 31 représentent un modèle KDM et Java respectivement d'une application GWT.

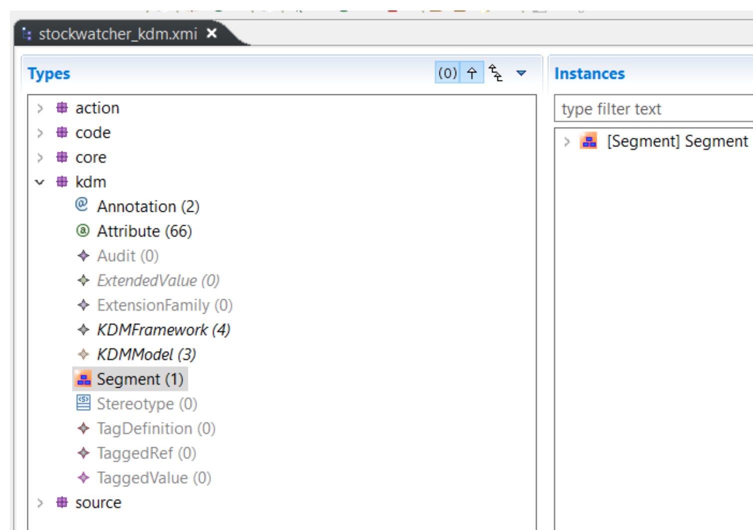


Figure 30 – Example d'un modèle KDM sous format XMI d'une application GWT généré par Modisco

10. JAR : est une format utilisée pour stocker les définitions des classes et les métadonnées constituants un ensemble d'un programme.

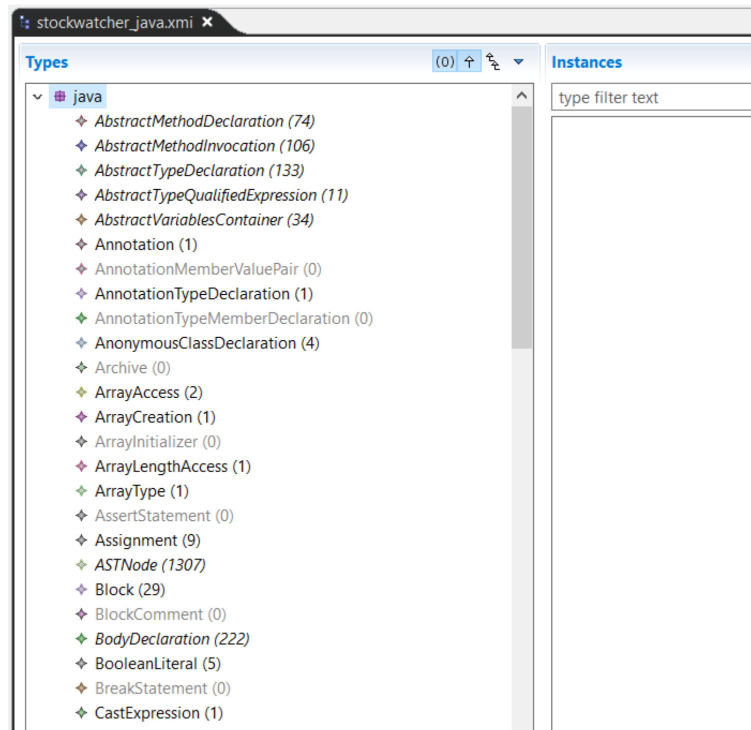


FIGURE 31 – Example d'un modèle Java sous format XMI d'une application GWT généré par Modisco

3.2.3 Transformation du modèle KDM vers un modèle GWT

Dans cette section nous présentons l'étape de transformation d'un modèle KDM à un modèle GWT. Cette transformation est un type de modèle vers modèle le (ce type est présenté dans la Section 3.2.1), en particulier, nous adoptons l'approche hybride (regardez la Section 3.2.1) car elle nous offre le style déclaratif qui facilite l'écriture des règles de transformation qui nous les expliquons par la suite.

La Figure 32 illustre une schématisation du processus de transformation d'un modèle KDM à un modèle GWT. Il comprends comme entrées : un modèle KDM et un modèle client. Ces deux entrées seront injectées par la suite dans un modèle de transformation. Ce dernier contient un ensemble de règles de transformation des entités du modèle KDM, conformes au méta-modèle KDM, vers un modèle GWT qui est conforme au méta-modèle GWT. D'autre part, le méta-modèle GWT est une adaptation ou enrichissement du méta-modèle KDM par des nouveaux concepts liées à GWT tels que ses éléments graphiques (Button, Label, , TextBox, etc) et la notion du page Web ce qui implique qu'une grande partie du modèle KDM sera clonée dans le nouveau modèle GWT. Pour détecter ces nouveaux concepts dans le modèle KDM nous présentons deux heuristiques principales. La première concerne la détection des éléments graphiques dans une page et l'autre la détection de pages d'une application.

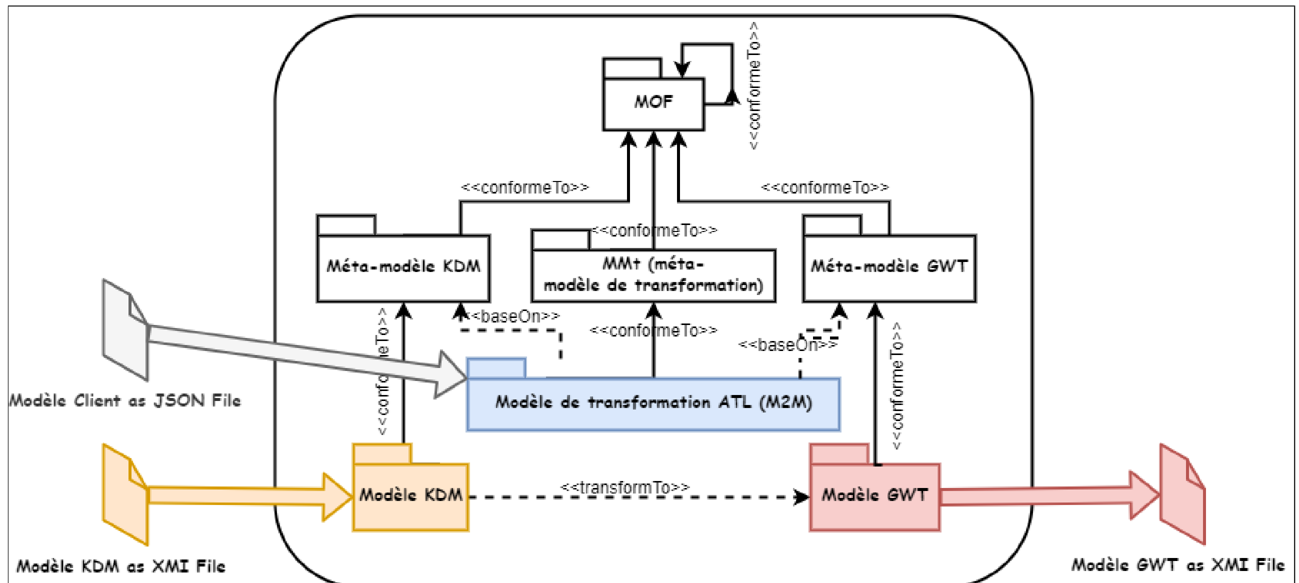


FIGURE 32 – Processus de transformation d'un modèle KDM vers GWT

Pour instance, la Figure 33 représente une pseudo dispersion des variables d'une classe nommée StudentDashBoard d'une application GWT illustrée dans la Figure 34. La classe StudentDashBoard contient un ensemble de variables qui sont mappées à l'entité StorableUnit du modèle KDM. En d'autres termes, chaque variable a un type par exemple : le type du closeButton est ClassUnit et il représente StorableUnit.

Nous voulons changer cette représentation à une qui prends en considération les nouveaux concepts liée à GWT (ses éléments graphiques) afin de simplifier l'analyse par la suite de l'application GWT. Des transformations sont effectuées pour changer la représentation de chaque variable qui représente StorableUnit à une entité qui modélise les concepts de GWT avec la contrainte qu'elle soit réellement une instance d'élément graphique de GWT.

L'idée de transformation, illustré par le pseudo-algorithme 1, se focalise sur le type de StorableUnit qui est mappé à l'entité ClassUnit du modèle KDM. Nous déterminons l'arbre d'héritage de ce type et nous analysons cet arbre dans le but de trouver un des noeuds dont le nom appartient à l'ensemble des mots clés d' éléments graphiques de GWT.

Algorithme 1 : parcourirArbreHeritage(classe : ClassUnit)

Entrée : classe de type ClassUnit
Sortie : boolean
parents = { }
si $type(classe) n'estpasuntypePrimitive$ & $pre(classe) <> nil$ & $classe.name \notin ensWidgetsGWT$ **alors**
| $retournerparents.union(parcourirArbreHeritage(pre(classe)));$
sinon
| $retournerparents;$
finsi

Pour bien formuler l'heuristique de détection de pages web d'une application GWT en une condition déclarative, nous exploitons deux principes de base. La première réflexion qui nous vient à l'esprit, une page peut être un conteneur d'éléments graphiques. En

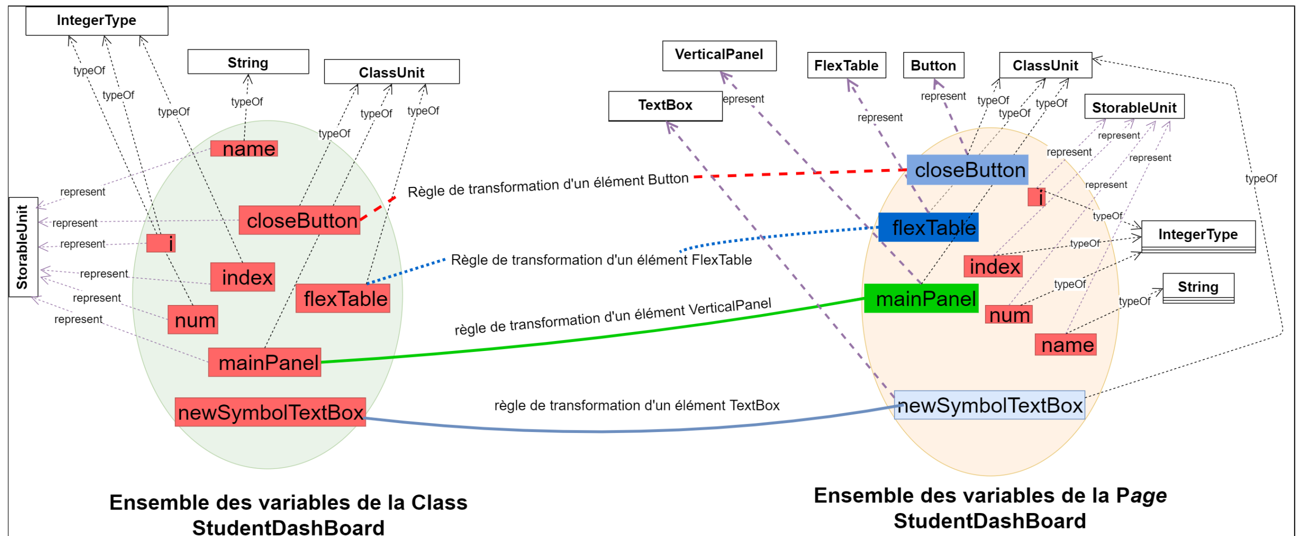


FIGURE 33 – Vue d'ensemble des variables de classe StudentDashBoard

GWT, cela veut dire qu'une classe doit être un fils directe/indirecte d'élément graphique de GWT appelé *Composite*[11]. Nous formulons cette déclaration par le pseudo-algorithme 2.

Tout d'abord, pour une entité de type *ClassUnit* dans le modèle KDM, nous récupérons son père s'il existe et vérifions par la suite son type s'il est de *ClassUnit*. Après, nous voyons le nom du père s'il est égale au mot clé "Composite" Alors nous détectons que c'est un fils de *composite*. Cependant, les développeurs de GWT peuvent faire un arbre d'héritage (les fils indirects) à partir de *Composite* alors nous rendons cet algorithme récursif pour le parcourir.

Algorithme 2 : Algorithme de vérification du type Composite pour une entité ClassUnit du modèle KDM

Entrée : class de type ClassUnit
Sortie : boolean
$parents = parcourirArbreHeritage(parent);$

$dernierParent = parents.recuprerDernierElement();$

si $parent.name =' Composite'$ **alors**
  $retournerVrai;$
finsi
$retournerFaux$

Deuxièmement, une page peut avoir des éléments graphiques, c'est-à-dire en GWT, une classe a des attributs ou elle peut déclarer des variables locales dans leurs types sont des éléments graphiques de GWT. Alors à partir d'une classe donnée, nous récupérons ses méthodes. Ensuite, nous analysons le corps de chaque méthode pour récupérer l'ensemble des variables déclarées et nous récupérons l'ensemble d'attributs de la classe pour les filtrer selon leurs types.

11. Composite : est un type de Widget qui peut encapsuler un autre Widget, masquant les méthodes du widget encapsulé. http://www.gwtproject.org/javadoc/latest/com/google/gwt/user/client/ui/Composite.html

```java
package fr.lirmm.gwt.migration.quizApp.client;
/*** GWT Handler */
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
/*** Part Of GWT Element: Window,Button,RootPanel,Widget*/
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.Widget;
import com.google.gwt.user.client.ui.TextBox
/*** StudentDashboard will be a Page*/
public class StudentDashBoard extends Composite {

  private Button closeButton;
  private VerticalPanel mainPanel;
  private TextBox newSymbolTextBox;

  /***  StudentDashboard @constructor */
  public StudentDashboard() {
    closeButton = new Button("close quiz");
    quizflexTable = new FlexTable();
    mainPanel = new VerticalPanel();
    newSymbolTextBox = new TextBox("entre your response");

    quizflexTable.addStyleName("watchList");
    /**
     * Make a positions for elements
     */
    mainPanel.add(newSymbolTextBox);
    mainPanel.add(closeButton);
    mainPanel.add(quizflexTable);

    RootPanel.get().add(mainPanel);
  }

  public void addQuiz() {
     final String name = newSymbolTextBox.getText().toUpperCase().trim();
     // ....
     int row = quizFlexTable.getRowCount();

     for (int i = 0; i < row ; i ++ ) {
       // ...
     }
  }
  // ....
}
```

Figure 34 – La classe StudentDashBoard d'une application GWT

Entre autres, le modèle client, Figure 35 le représente comme un exemple, peut contenir une déclaration de différentes pages d'applications, nous les intégrons dans le modèle de transformation. Le besoin du modèle client est de porter une solution alternative si les deux principes n'entraînent pas à des bons résultats.

```
1 {
2   "Pages": ["'LoginPage'", "'TeacherDashBoard'", "'Quiz'", "'StudentDashBoard'"]
3 }
```

FIGURE 35 – Modèle client sous format JSON

Algorithme 3 : Algorithme de vérification du contenu d'une entité ClassUnit du modèle KDM

Entrée : classe de type ClassUnit
Sortie : boolean
méthodes = Récupérer les méthodes de classe
attributs = Récupérer les attributs de classe
ensWidgetsGWT = Ensemble d'éléments graphiques du GWT
ensWidgetOfUiEntity = { }
pour *chaque méthode* $\in$ *méthodes* **faire**
  declaredVariables = Récupérer les variables déclarées dans méthode
  pour *variable* $\in$ $declaredVariables$ **faire**
    si $variable.type.name \in ensWidgetsGWT$ **alors**
      $ensWidgetOfUiEntity.ajouter(variable)$
    finsi
  fin
fin
pour *chaque attribut* $\in$ *attributs* **faire**
  si $attribut.type.name \in ensWidgetsGWT$ **alors**
    $ensWidgetOfUiEntit.ajouter(attribut)$
  finsi
fin

3.2.4 Construction du modèle de navigation

La technologie GWT implémente le protocole AJAX (COMMUNITY p. d.). En donnant un exemple, un étudiant clique sur un bouton d'une page d'authentification d'une application, étant développée en GWT, pour suivre ses cours/Travaux pratiques, un tableau de bord, par la suite, va remplacer l'ancienne page d'où l'URL de navigateur va être changé sans remise à niveau.

AJAX repose sur les appels asynchrones entre client et serveur. L'idée de ce dernier est de mettre à jour la page sans la recharger. L'utilité derrière AJAX est d'éviter la surcharge du navigateur d'un côté et être beaucoup plus réactif et fluide tout en réduisant les besoins en bande passante de l'application et la charge aussi sur le serveur.

Malgré les avantages d'AJAX, GWT n'offre pas à un développeur de définir clairement les routes de l'application (transitions entre les pages) si nous comparons celles-ci avec la déclaration de routes en Angular par exemple, présenté dans la Figure 36 :

```
const routes: Routes = [
  { path: 'loginPage', component: LoginComponent },
  { path: 'StudentDashBoard', component: StudentComponent },
];
```

FIGURE 36 – Exemple de déclaration de routes en Angular

Cependant, GWT propose par défaut un modèle de gestion des événements similaires au modèle d'événement du JavaScript (FIREFOX p. d.) tels que clic, double clic, appuyant sur une touche et d'autres. Le principe du modèle d'événement est lors d'une interaction avec un composant d'une page par cliC, double clic ou survol, un gestionnaire d'événement va être déclenché pour traiter l'action affectée dans une page. GWT utilise ce principe pour altérer les pages dans le gestionnaire d'événement en invoquant deux méthodes :

1. RootPanel.get().clear() :
   La méthode clear() a un but de supprimer tous les éléments du DOM (document object model) qui représente sauf le parent qui représente en GWT par l'instance RootPanel.get(). RootPanel représente une implémentation d'un patron de conception de création Singleton qui garantit que l'instance de RootPanel n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.

2. RootLayout.get().add(Widget widget) :
   la méthode add(Widget widget) permet d'ajouter un composant graphique au DOM appelé par l'instance de la classe RootLayout. La classe StudentDashBoard présenté dans la Figure 34 peut être afficher dans le navigateur depuis en utilisant ce code suivant illustré par la Figure 37.

Notre objectif est de construire un modèle de navigation qui représente un graphe de transitions entre les pages d'une application GWT. Le besoin de ce graphe est de comprendre le processus de circulation dans l'application et d'extraire les vraies pages de l'application car l'heuristique de détection des pages présentée dans la section précédente 3.2.3 peut nous retourner des faux résultats, c'est-à-dire, des pages ne représentent pas réellement des pages. Notre démarche pour proposer une solution à cette problématique est de faire une heuristique de recherche pour trouver la prochaine page d'une page. Celle-ci consiste à chercher l'apparition d'une page dans les méthodes qui traitent les évènements.

```
// une autre Page
public Class LoginPage extends Composite {
  // ....
  LoginPage() {
    // .........
    // ajouter un gestionnaire d'évènement
    // une évènement sera déclenchée lors du click
    // sur le l'élement button button
    Button button = new Button("Next");
    button.addClickHandler(new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        RootPanel.get().clear()
        RootPanel.get().add(new StudentDashboard())
      }
    }); // ......
  } // ......
}
```

FIGURE 37 – Exemple du code de navigation vers une autre page

A partir du modèle GWT nous récupérerons l'ensemble des pages détectées avec leurs composants graphiques en utilisant un analyseur du modèle GWT que nous avons implémenté. Ensuite, un modèle JAVA de l'application GWT donnée qui est conforme au modèle ASTM, est injecté pour récupérer l'ensemble des gestionnaires des événements guidé par les pages récupérées. Le besoin de renforcer l'étude avec le modèle JAVA est que ce dernier est plus précis que le modèle GWT pour détecter les déclarations et les expressions du langage JAVA utilisées dans l'implantation de l'application GWT. De la même façon, Nous avons conçu un analyseur spécial pour traiter le modèle JAVA. Après que les gestionnaires sont récupérés, nous les analysons à l'aide d'un trancheur des programmes orienté modèle (nous les expliquons dans le paragraphe suivant) pour construire un ensemble de transitions représentant un modèle de navigation de l'application où chaque transition est représentée par la forme { from ; by ; token ; to } où :

1. from : représente la page source auquel l'événement est déclenché.
2. by : représente l'élément qui a déclenché l'événement.
3. token : représente le chemin vers la prochaine page.
4. to : représente la page destination.

Nous présentons un pseudo-algorithme 4 qui extrait les gestionnaires des évènements et qui crée l'ensemble des transitions. Par la suite, ces derniers vont être sérialiser dans un modèle nommé modèle de navigation. Dans le pseudo-algorithme 4, il existe un appel à une fonction nommée *createTransition*. Celle-ci base sur le trancher du programme pour créer les transitions. Elle comprend comme arguments :

1. methodeHandler : représente la méthode qui traite l'évènement. Pour la récupérer, nous analysons l'argument injecté dans la méthode qui ajoute un gestionnaire d'événement. De plus, cet argument peut avoir plusieurs façon de déclaration, présenter dans la figure 38.

2. classDeclaration : est l'entité récupérer par l'analyseur du modèle JAVA qui représente la classe java qui conforme à une page, nous la récupérons par une opération de transposition appliqué à la page.

3. pages : est une liste des pages récupérées par l'analyseur du modèle GWT. cette liste est utiliser au plus tard pour vérifier l'existence d'une page détecté dans methodeHandler.

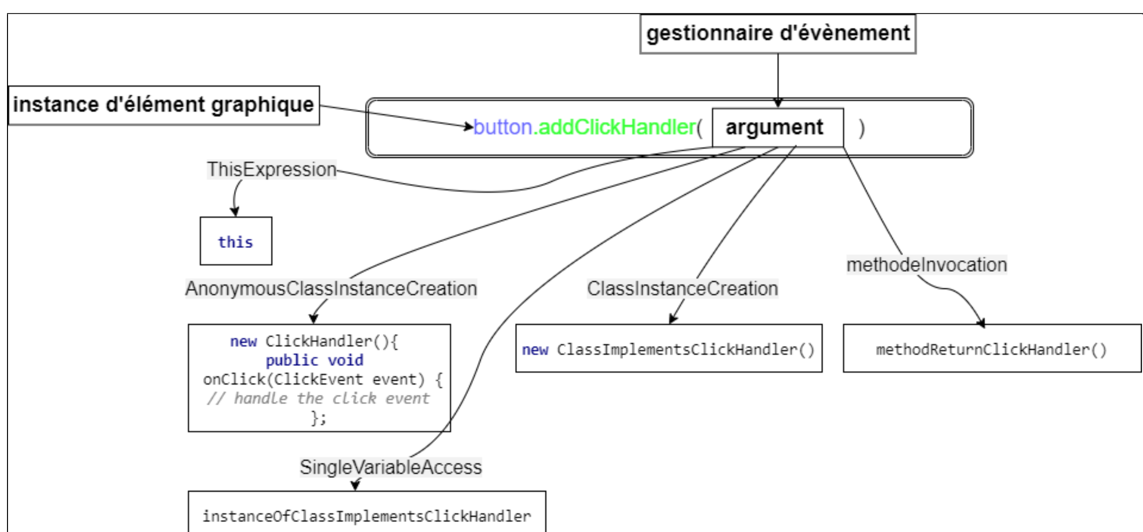


FIGURE 38 – Les différents cas de type d'un gestionnaire d'événement GWT vue d'un modèle JAVA

Algorithme 4 : Algorithme du construction de transitions

```
Entrée :
Sortie : transitions
pages = l'ensemble des pages de l'application récupérer depuis modèle GWT
gestionnairsEvenements = { } ;
méthodes = { } ;
statements = { } ; // représentent l'ensemble d'instructions
argument = nil ; // représente l'argument injecté dans méthode qui ajoute // un
  gestionnaire d'évènement
methodeHandler = nil ; // représente la méthode qui traite l'évènement
transitions = { }

pour chaque actualPage ∈ pages faire
    classDeclaration = TransposerLaPage(actualPage)
    méthodes = Récupérer l'ensemble des méthodes de classDeclaration ;
    pour chaque méthode ∈ méthodes faire
        déclarations = Récupérer l'ensemble de déclarations de méthode ;
        pour chaque statement ∈ statements faire
            si (statement.type(Appel méthode) ) Et( statement.méthode ∈
            méthodes d'ajout d'un gestionnaire d'évènement de GWT ) alors
                argument = récupérer l'argument de la méthode invoquée ;
                methodeHandler = Récupérer la méthode d'évènement;
                transitions.ajouter(createTransition(methodeHandler,classDeclaration,pages)) ;
            finsi
        fin
    fin
fin
retourner transitions ;
```

Nous présentons dans cette section un mécanisme de découpage au niveau modèle pour ressortir les pages qui ont été appelées dans le corps de la méthode qui traite l'événement (onClick, onBlur et d'autres). L'idée de base est de suivre les utilisations des instances des classes qui conforment aux pages. Nous focalisons sur les arguments des méthodes invoquées dans la méthode traitante car le changement de la page se fait par un appel : *RootPanel.get().add(......)*. Un argument en modèle Java est représenté par une entité appeler Expression. D'autre part, Chaque développeur a sa propre méthode d'implémentation alors nous avons identifié, au premier temps, les cas possibles que nous pouvons les rencontrer très souvent :

1. Un développeur peut faire le changement vers une autre pages en instanciant directement une classe qui conforme à une page, La Figure 39 suivante montre l'exemple :

```
button.addClickHandler(
  new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        RootPanel.get().clear();
        /** ClassInstanceCreation **/
        RootPanel.get().add(new StudentDashBoard());
      }
  }
);
```

FIGURE 39 – Instance comme argument

   cette instance est modéliser en modèle Java une entité appelé ClassInstanceCreation. Dans ce cas, nous analysons l'instance en vérifiant si elle est d'une classe qui conforme à une page, alors nous créons l'objet Transition qui prend comme from : le nom de la page actuelle, by : l'élément qui déclenche l'évènement, to : le nom de la page détecté, token : est généré aléatoirement.

2. Autre cas, il peut affecter l'instanciation à une variable et par la suite il l'injecte dans la méthode d'échange ou il peut affecter également la variable affecté à une autre variable, la Figure 40 suivante montre ce cas

```
button.addClickHandler(
  new ClickHandler() {

      @Override
      public void onClick(ClickEvent event) {
        StudentDashBoard dashBoard = new StudentDashBoard();
        /** le cas d'affectation **/
        StudentDashboard dash1 = dashBoard;
        RootPanel.get().clear();
        RootPanel.get().add(dash1);
      }
  }
);
```

FIGURE 40 – Variable affectée comme argument

   Ce cas est l'un des cas complexes, notre réflexion repose sur la traçabilité de la variable injectée dans la méthode *add()*. La traçabilité de la variable comprend tous les chemins de son utilisation dans une classe. D'abord, nous localisons cette traçabilité juste dans la méthode qui traite l'événement. Ensuite, nous focalisons sur ses affectations (qu'est ce qu'il a reçu) afin d'arriver à son initialisateur qui devrait être une instance d'une classe.

3. Il peut appeler une méthode qui nous permet de retourner une instance d'une classe, La Figure 41 suivante montre ce cas :

```java
button.addClickHandler(
  new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        RootPanel.get().clear();
        /** argument as Method Invocation **/
        RootPanel.get().add(getStudentDashBoard());
      }
  }
);

/** Déclaration de la méthode goToNextPage **/
public StudentDashBoard goToNextPage() {
    return getStudentDashBoard();
}
```

FIGURE 41 – Invocation à une méthode comme argument

Pour traiter ce cas, nous cherchons dans le corps de la méthode invoquée toutes les instructions de retour et par la suite nous analysons ses expressions (la partie droite du return en Java) pour trouver une instanciation d'une classe qui conforme à une page. Les méthodes invoquées en modèle Java sont représentés par l'entité *Method Invocation* et les instructions du retour par l'entité *ReturnStatement*.

4. Il peut également encapsuler les étapes de changement dans une autre méthode et il fait par la suite une appel à celle-ci dans la méthode d'évènement, illustré par la Figure 42 suivante :

```java
button.addClickHandler(
  new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        /** MethodeInvocation **/
        goToNextPage();
      }
  }
);
/*Déclaration de la méthode getToNextPage*/
public void goToNextPage() {
  RootPanel.get().clear();
  RootPanel.get().add(new StudentDashBoard());
}
```

FIGURE 42 – Cas d'une

Le premier pas à faire est d'aller récupérer la méthode de déclaration et par la

suite nous analysons son corps qui nous amener à revoir les cas précédents d'où la récursivité est impliquée.

Nous montrons notre solution pour les quatre cas par un diagramme d'activité pour simplifier sa visualisation. Tout d'abord, nous récupérons les méthodes invoquées qui sont dans la méthode d'évènement par la suite deux branchement sont impliqués :

1. pour extraire le corps de leurs déclarations et chercher récursivement les invocations de d'autres méthodes. Après, le retour est rebranché dans la première étape.
2. pour extraire les arguments des méthodes invoquées.

Après, nous analysons le type d'argument dans le but de trouver l'instance qui impliqué par l'argument. Pour l'instance nous avons traité que trois types d'arguments : *ClassInstanceCreation*, *SingleVariableAccess* et *MethodInvocation*. Nous pouvons ultérieurement traiter touts les cas possibles en ajoutant les sous fils d'une expression Java présentés dans l'annexe 49.

La vraie sortie qui nous retourne la transition est établie par l'étape **Analyse l'instance** dans le diagramme 43

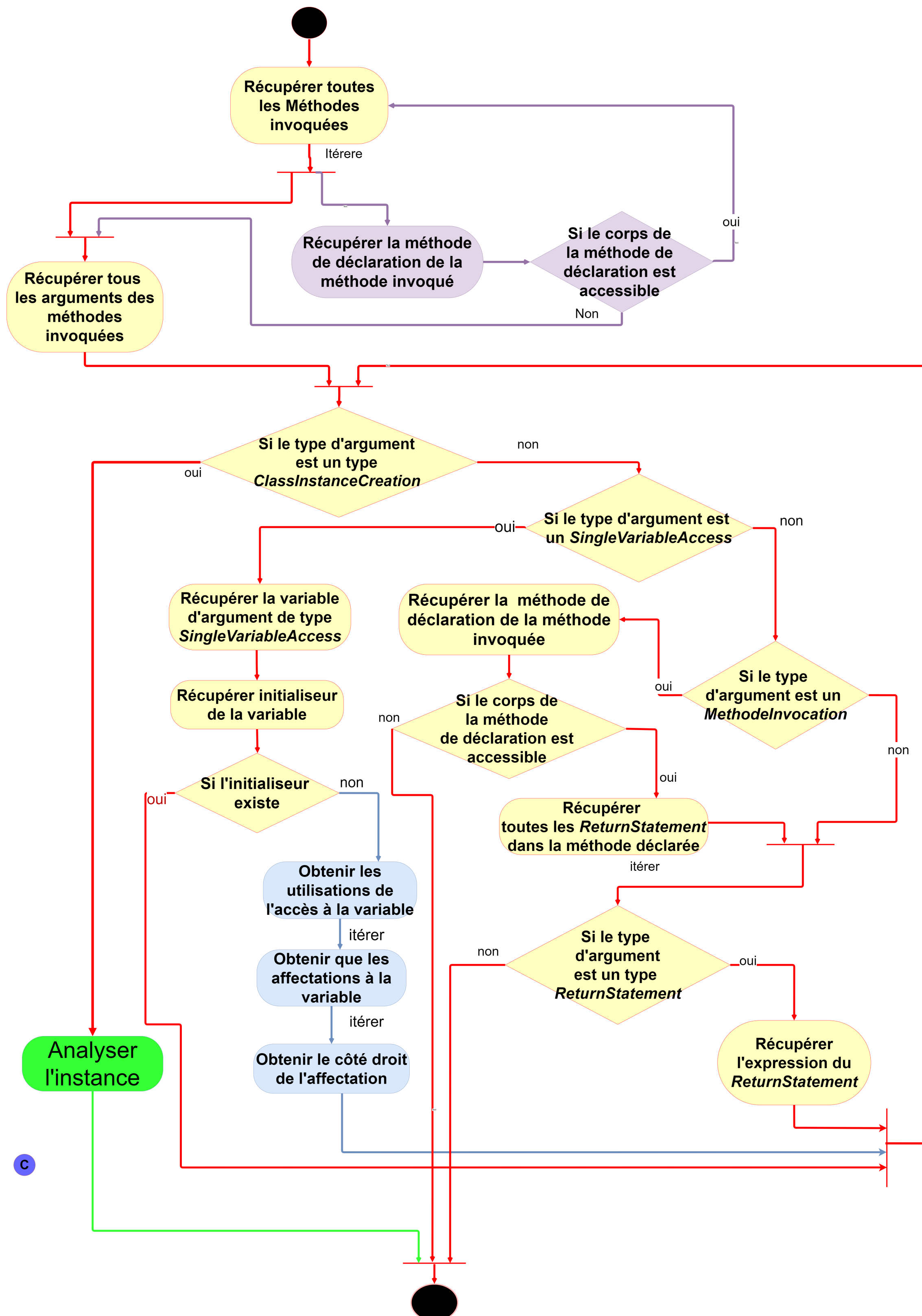


FIGURE 43 – Diagramme d'activité pour la traçabilité d' argument injectées dans la méthode *RootPanel.get().add(...)*

3.2.5 Construction d'un modèle intermédiaire

Notre objectif dans cette partie est de construire un modèle qui résume ce que nous avons fait dans les étapes précédentes du processus de migration. Nous avons extrait les pages de l'application avec leurs les composantes utilisées dans chacune des pages après que nous avons identifiés les transitions entre eux. La figure **??** montre qu'est ce que nous cherchons afin de visualiser l'application GWT pour comprendre sa structuration.

D'abord, nous récupérons l'ensemble des pages du modèle GWT. Ensuite, nous appliquons un filtrage sur cet ensemble pour ne ressortir que les pages qui sont utilisées pendant l'exécution de l'application à l'aide du modèle de navigation. Après, pour chaque page nous récupérons l'ensemble des éléments graphiques. Puis, pour chaque élément graphique, nous extrayons l'ensemble de ces attributs et actions, par exemple la méthode *setText* initialise l'attribut texte d'une instance de *Button* en GWT. Après que les actions sont extraites, nous volons savoir l'élément parent parmi les éléments d'une page car il est indispensable pour nous donner une vision sur le positionnement entre les éléments d'une page. Enfin, nous sérialisons tous ce que nous avons fait dans un ficher sous format JSON.

Maintenant, nous découvrons comment nous avons procédé pour détecter les méthodes invoquées par les instances des composants de GWT. Après, nous expliquons un algorithme génétique pour localiser l'élément parent.

Pour détecter les attributs et les actions qui appartiennent à un élément, nous utilisons le modèle pour détecter dans quelle variable Java l'élément a été affecté. Puis, nous recherchons les méthodes invoquées sur cette variable. Parfois les développeurs encapsulent ces variables dans des méthodes qui reviennent elles même. Dans ce cas, nous essayons de faire une traçabilité sur ces méthodes pour détecter les méthodes qui sont invoquées.

Une autre façon, si la variable qui représente l'élément graphique a été affecté par une autre variable, alors nous essayons d'appliquer le même processus pour la variable affectant. Un cas compliqué que nous avons laissé à traiter ultérieurement, lorsque le développeur crée son propre composant et par la suite il fait une affectation à une variable qui devrait être affichée dans la page. Généralement, il utilise des méthodes ou des classes qui prennent la responsabilité de fabriquer un nouveau composant décoré. La Figure 44 montre ce cas.

Le diagramme d'activité de la Figure 45 représente notre solution pour extraire l'ensemble des méthodes invoquées par une variable qui représente un élément graphique. Cette solution ne traite que les trois cas mentionnés ci-dessus. Dans l'étape de couleur verte du digramme, nous extrayons le nom de la méthode avec ses arguments.

```
public class BLCore implements EntryPoint {
    Button Mybutton = new Button();

    public void onModuleLoad() {
        Mybutton = createMyButton();
        RootPanel.get().add(Mybutton);
    }

    public Button createMyButton() {
        Button button = new Button();
        button.setText("Add");
        button.addStyleName("btn")
        return button;
    }
}
```

FIGURE 44 – Développeur déclare dans ça méthode

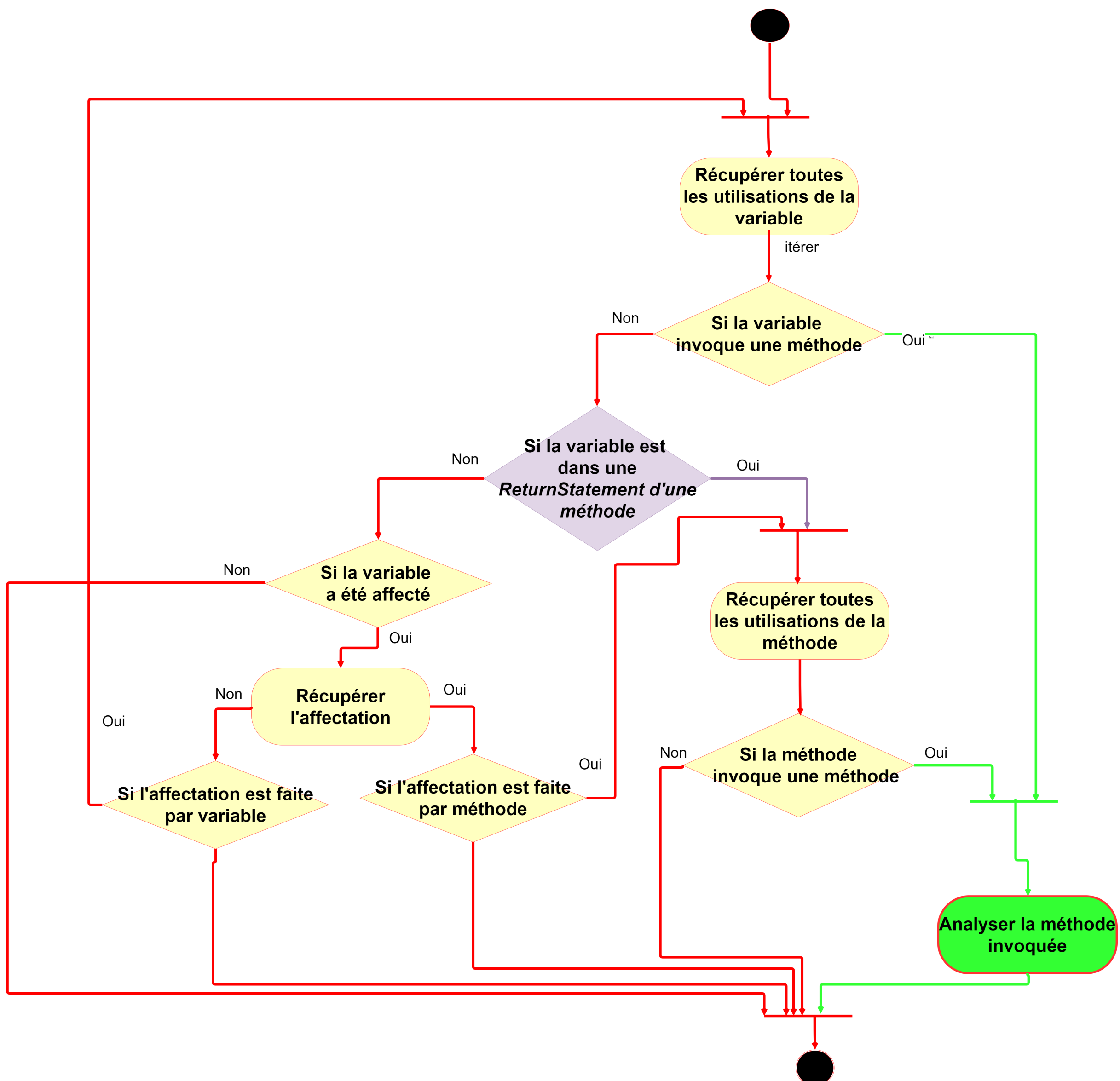


FIGURE 45 – Processus de détection des actions/attributs d'un élément graphique

En ce qui concerne la détection de l'élément parent dans chaque page, notre solution

ce problème est de créer pour chaque élément qui appelle la méthode *add*, un arbre de positionnement qui comprend comme racine l'élément appelant et dans les fils les éléments les arguments passées à la méthode. À partir de l'ensemble d'arbres créés, nous faisons une intersection qui nous donne un ensemble d'éléments communs entre eux. Puis, nous supprimons les arbres dans lesquels leurs racines existent dans la liste d'intersection. Après la suppression, nous devons avoir qu'un seul arbre qui est dans la racine l'élément qui nous vient de le rechercher. La Figure ci-dessous montre les étapes de ce processus.

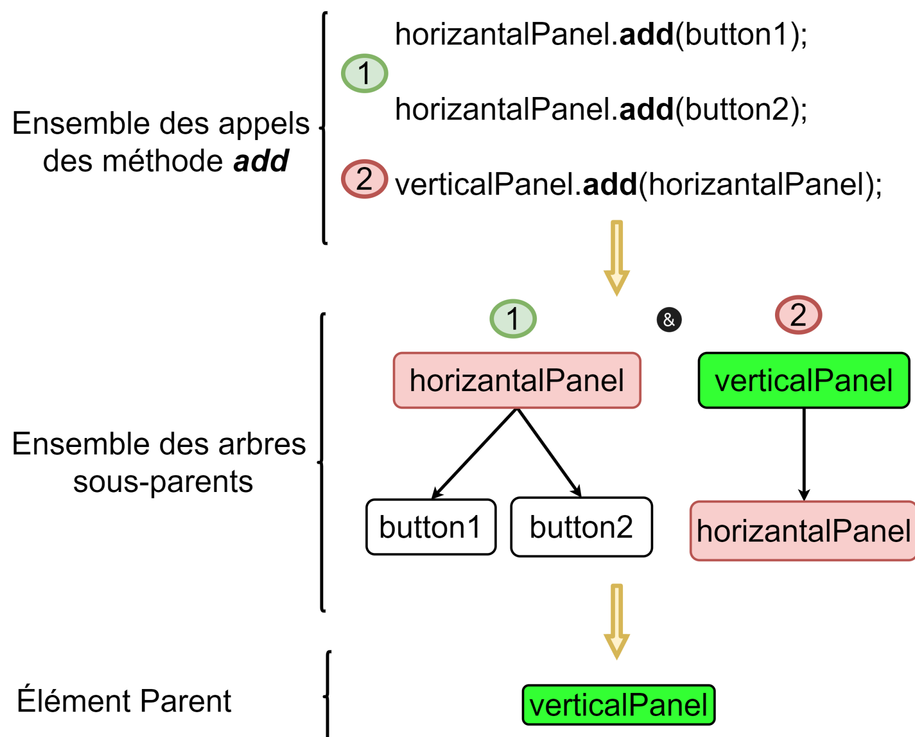


FIGURE 46 – Déroulement du processus de détection de l'élément parent dans chaque page

Un cas que nous pouvons observé est l'interblocage entre les arbres de positionnement, c'est-à-dire, après l'intersection entre les arbres, l'algorithme ne peut pas décider quel arbre qu'il prend, alors nous renforçons notre précision par l'ajout d'une condition liée à GWT. Cette technologie permet aux classes qui héritent de l'élément 'Composite' d'appeler dans leurs constructeurs une méthode nommée *initWidget(Widget w)* qui initialise l'élément racine de la page. Donc si nous trouvons *initWidget* dans une classe qui conforme à une page, nous récupérerons son argument qui est l'élément racine.

3.2.6 Génération du code Angular

Une fois le modèle intermédiaire généré, il est possible de générer l'application en Angular. Mais avant de générer, une question conceptuel techniques sont posées comme suit :

1. Est ce que nous devons faire une transformation du modèle intermédiaire vers un autre ? Pourquoi ?
2. Comment l'infrastructure *Angular* va connaître les nouvelles composantes graphiques générées ?

3. Comment faire pour établir la correspondance entre les éléments *GWT* et les d'*Angular Material* ?
4. Comment pouvons-nous transformer cette correspondance en un code d' *Angular* ?
5. Quelle est la meilleure architecture Angular que nous devons la prendre pour migrer l'application source ? et pourquoi ?
6. Comment le processus de génération de l'application va fonctionner ?

— *Correspondance entre les éléments GWT et Angular Material :* Dans cette partie, nous répondons à les questions 2, 3 et 4.
GWT contient 88 éléments graphiques et *Angular Material* contient 36 composant graphique (la dernière version). Nous voyons que les tailles du deux ensembles ne sont pas égales d'un côté. Pour cela, nous faisons une correspondance manuelle selon la vue sémantique de l'élément *GWT* en *HTML* par exemple :

GWT	HTML	Angular Material
Button	$< button > Basic < /button >$	$< buttonmat - button > Basic < /button >$

TABLE 5 – Correspondance entre les éléments graphiques *GWT* et *Angular Material*

Certains éléments de GWT sont complexes à les convertir vers des éléments d'Angular Material comme Frame, DialogBox et d'autres. D'autre côté, GWT dépose d'un mécanisme de mise en page (*Layout*) à l'aide d'une partie de ces éléments comme *DockPanel*, *FlowPanel* et d'autres. Nous ne pouvons pas faire la correspondance car Angular Material n'offre pas le mécanisme de mise en page pour ces composants. Alors, nous utilisons les balises *div* d'*HTML* avec les flexbox[12] de *CSS3* qui sont des normes de disposition des éléments dans une page web.

Après que les correspondances sont établies entre les éléments *GWT* et *Angular Material*, nous voulons les implémenter sous forme d'une bibliothèque *Angular* pour viser la réutilisation des éléments migrés de GWT. Elle peut aussi contribuer à réduire un peu le temps de migration. Une bibliothèque *Angular* est un ensemble de composants, modules et services réutilisables. Alors, Pour chaque élément graphique GWT, nous générons un composant Angular qui contient quatre fichiers essentiels, un fichier nommé :

1. 'nom-element-gwt-en-miniscule'.component.html : nous insérons dans celui-la la balise d'appel d'un composant d'Angular Material correspondante à un élément graphique de GWT.
2. 'nom-element-gwt-en-miniscule'.component.css : nous appliquons les styles CSS aux balises HTML.
3. 'nom-element-gwt-en-miniscule'.component.ts : celui-la est pour gérer dynamiquement les configurations qui seront insérés dans les balises.
4. 'nom-element-gwt-en-miniscule'.component.spec.ts : Nous pouvons ici faire des cas de test sur le composant que nous avons produit pour voir.

12. flexbox : un modèle de disposition unidimensionnel et une méthode permettant de distribuer l'espace entre des objets d'une interface ainsi que de les aligner. Pour plus de détails : `https://developer.mozilla.org/fr/docs/Web/CSS/CSS_Flexible_Box_Layout/Basic_Concepts_of_Flexbox`

Nous donnons un exemple, pour un élément Button de *GWT*, son correspondant en *Angular material* est le composant Button, donc 4 fichiers sont créés : button.component.css, button.component.ts, button.component.spec.ts et button.component.html contient le code suivant illustré par la Figure :

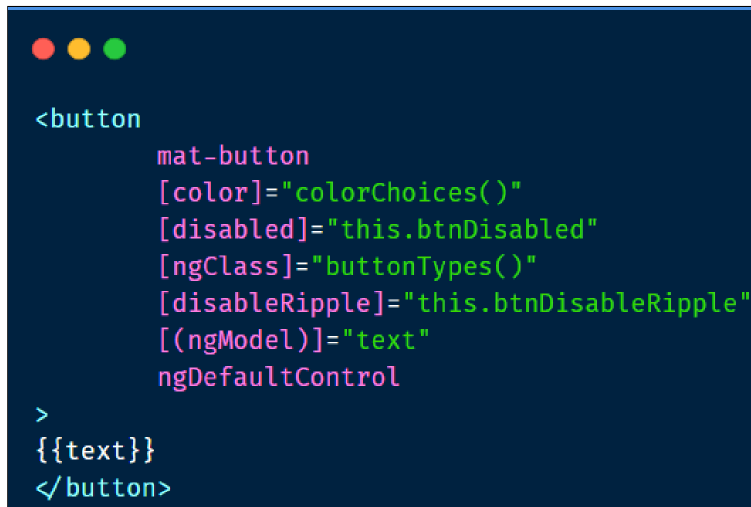


FIGURE 47 – Code HTML d'élément Button *GWT* après la correspandance

— *Architecture de l'application cible :* Dans cette partie, nous répondons à la questions 5 : "quelle est la meilleure architecture Angular que nous devons la prendre pour migrerl'application source ? et pourquoi ? "
En 2001, Martin Fowler a proposé un patron de conception appelé Lazy Loading ("chargement fainéant") pour augmenter les performances de chargement des données d'une base en mémoire. L'idée de base était la métaphore : "Si vous êtes paresseux pour faire des choses, vous gagnerez quand il s'avérera que vous n'avez pas du tout besoin de les faire". Le problème de ce patron était lorsque les développeurs récupèrent des données d'une base sous forme des objets liée entre eux en mémoire, les performances de l'application diminue. La solution était d'interrompre le processus de chargement pour un moment, laissant un marqueur (jeton) dans la structure de l'objet en mémoire de sorte que si les données sont nécessaires, elles ne peuvent être chargées que lorsqu'elles sont utilisées (FOWLER 2002).

  Avec les applications web, la vitesse pour afficher une page web est l'un des contraintes les plus primordiales pour la satisfaction de l'utilisateur de l'application. Le framework Angular a une configuration pour effet d'accélérer le fonctionnement d'application web. Cette configuration implémente le pattern lazy loading. Elle permet de spécifier quelles parties d'une application web doivent être chargées lors du démarrage afin de réduire le temps de chargement d'une application au niveau de navigateur surtout avec les applications volumineuses. En Angular, Le lazy loading fonctionne en utilisant la notion de modules et non plus celle des composants. Pour les raisons citées ci-dessus, nous allons adapter l'architecture de l'application générée pour qu'elle soit basée sur lazy loading.

  La solution que nous proposons en créant pour chaque page un module et pour chaque élément graphique de la page nous générons un composant à l'intérieur du module. Ce code généré doit etre inséré dans un modèle (template) du projet Angular.

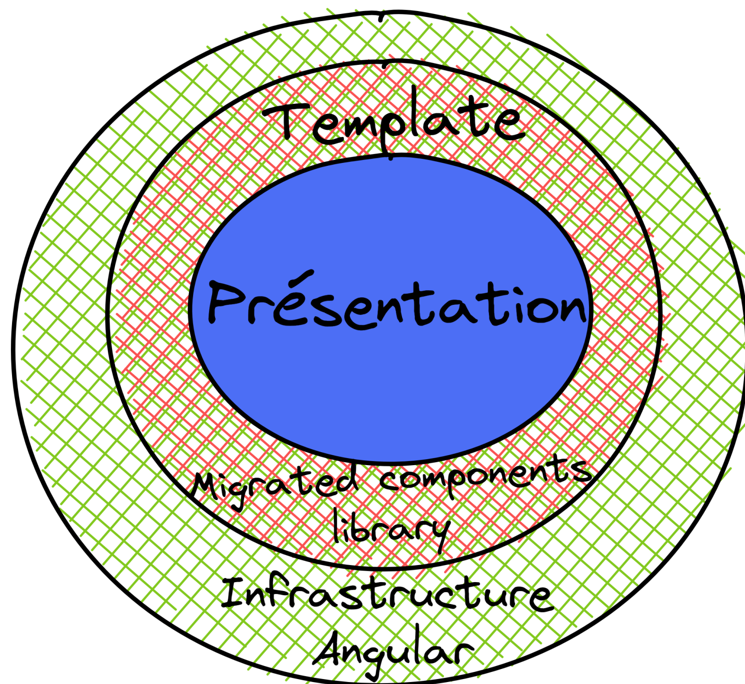


Figure 48 – Architecture générale de l'environnement de l'application généré

La Figure 48 présente l'architecture de notre projet. Tout d'abord, l'infrastructure représente l'environnement de développement du framework Angular. Le template est la place ou nous allons ajouter notre code migrée qui contient un ensemble de modules et des composants (Angular). Ceux-ci utilisent les éléments que nous avons migrés en une bibliothèque Angular pour migrer l'aspect visuelle d'application.

— *Processus de génération d'un code :* Dans cette partie, nous allons parler sur comment le processus de migration se fonctionne et comme nous pouvons le voir.
Une fois l'architecture du projet est implémentée, il est possible d'exporter l'application à partir du modèle intermédiaire pour générer le code de l'application cible. D'abord, nous commençons par créer pour chaque page un module nommé par le nom de la page. Ensuite, pour chaque élément de page nous générons un composant nommé par le nom de la variable. Cette tâche est faite d'une manière automatique en exploitant le composant de commande d'*Angular* nommé 'ng'. Puis nous effectuons des étapes de configuration dans l'architecture en utilisant le patron *Template Method*[13]. Ce patron nous permet de rendre la configuration d'intégration maintenable et extensible dans la direction si nous proposons des nouvelles configurations pour le code généré. Nous expliquons les détails des configurations dans la partie réalisation.
Puis, nous revisitons les éléments de chaque page pour remplir le fichier *HTML* du composant (Angular) générer avant cette étape. Le contenu du composant est généré par un traiteur d'élément qui convient au type d'élément. Nous appliquons un patron de conception appelé *Chain of Responsibility*[14] .

13. Template Method : est un patron de conception comportemental qui permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure.(Shvets p. d.)

14. Chain of Responsibility : est un patron de conception comportemental qui permet de faire circuler des demandes dans une chaîne de handlers. Lorsqu'un handler reçoit une demande, il décide de la traiter

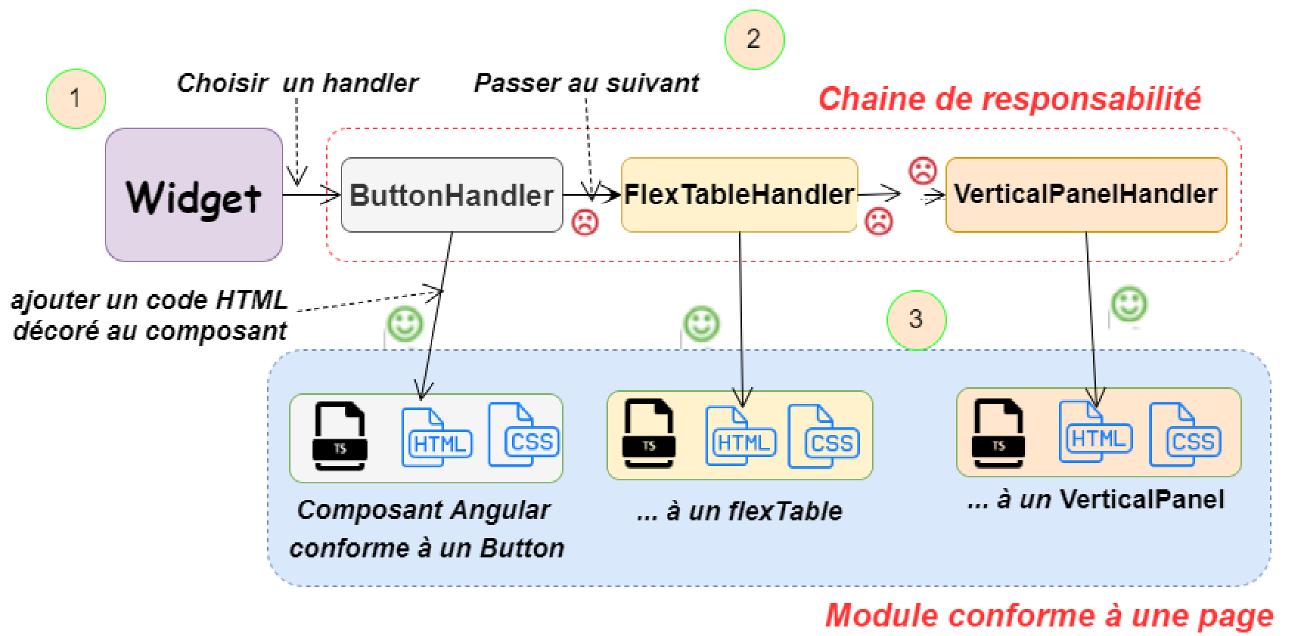


FIGURE 49 – Processus de traitement d'un élément graphique

ou de l'envoyer au handler suivant de la chaîne.(SHVETS p. d.)

4 Conception

4.1 Conception architecturale

Une conception architecturale représente un modèle définissant comment sera le système, nommée *style architecturale*. Un style architectural définit quels sont les composants, les connecteurs et les contraintes définissant l'architecture d'un système (MOSTEFAI 2016).

Pour l'instant, nous adaptons le style architectural *Pipe/Filter*, ce choix est justifié par les simplicités qu'offre cette architecture. En effet, les étapes de notre processus de migration peuvent être mises en place d'une manière claire, séquentielles, configurable et maintenable. Notre objectif en générale est de faire une série de transformations/ traitements des informations que nous avons récupérer depuis le code source jusqu'à la génération de l'application cible. Ceci est assuré par les principes du style architectural *Pipe/Filter* qui permet à l'information d'être traitée par plusieurs composants d'une manière séquentielle à travers :

— *Filter* : est un composant qui traite l'information.

— *Pipe* : est un canal par lequel transite l'information.

La Figure 50 illustre notre solution architecturale pour le processus de migration de GWT vers Angular.

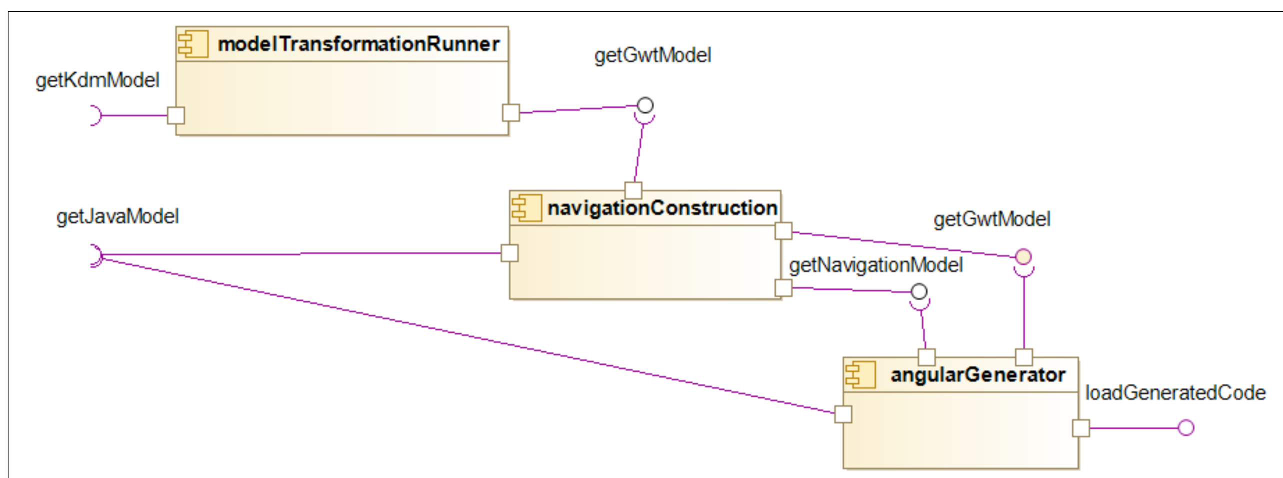


FIGURE 50 – Architecture de notre solution pour la migration GWT vers Angular

Dans ce qui suit, le rôle de chaque composants :

— Le composant « **modelTransformationRunner** » : sert à exécuter les règles de transformation implémentées par ATL, que nous découvrons dans la partie implémentation, du modèle kdm d'une application GWT à un modèle GWT (un modèle KDM adapté par les concepts GWT).

— Le composant « **navigationConstruction** » : sert à construire le modèle de navigation en récupérant le modèle GWT et un modèle Java de l'application source.

— Le composant « **angularGenerator** » : assure les deux principales tâches qui sont : 1) la construction d'un modèle intermédiaire (pivot) et 2) la génération du code Angular.

Dans la section suivante, nous détaillons ces composants en spécifiant leurs diagrammes de classes ainsi que leur logique métier

4.2 Conception détaillée

4.2.1 Diagramme de classes

Le diagramme de classes est le diagramme UML le plus important de la modélisation orientée objet. Il permet de modéliser les classes du système et leurs relations indépendamment d'un langage de programmation de programmation. Une classe en UML se modélise par un rectangle à trois compartiments. Le compartiment supérieur contient le nom de la classe, le compartiment du milieu comprend les attributs de la classe et l'autre contient les méthodes de la classe. Dans ce qui suit, nous donnons les diagrammes de classes des différentes composantes.

Composant «modelTransformationRunner» Il contient trois package essentiels, représenter par la Figure 51 :

— Package « main.java.lirmm.modelTransformation.atl.runner » : il contient la classe qui s'occupe à recevoir le modèle KDM sous format XMI depuis un dossier de ressource nommé « models.inputs »en utilisant la méthode getModelHandler qui localise le chemin completet du modèle source. La méthode launch va charger les règles de transformation ATL et transforme par la suite le modèle source en un modèle KDM.

— Package « main.java.lirmm.modelTransformation.atl.json » : il se compose de deux classes une abstraite nommée *JSONFileReader* et son implémentation ClientModelFileReader. celle-ci contient une méthode getPages qui renvoie l'ensemble des pages déclarées dans le modèle client qui située dans un dossier externe nommée ressource.

— Package « main.java.lirmm.modelTransformation.atl.writer » : il contient deux classes. La classe SimpleAtlFileWriter s'occupe à faire un mise à jour du modèle de transformation sous format ATL que vous la découvrez dans la partie implémentation. La mise à jour représente l'ajout de l'ensemble des pages récupérer par la classe ClientModelFileReader.

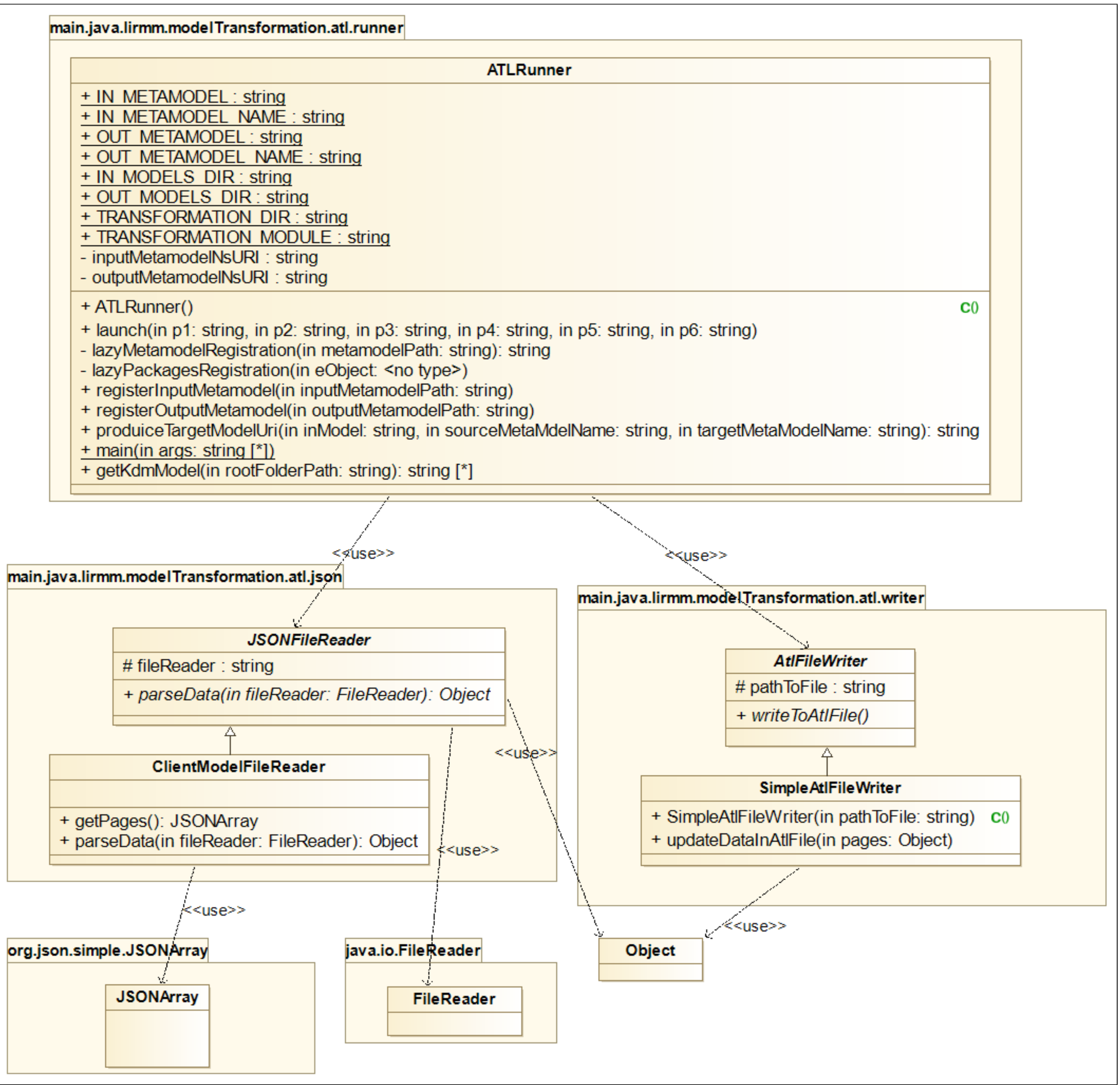


Figure 51 – Diagramme de classe du composant modelTransformationRunner

Composant «navigationConstruction»

Après la transformation du modèle Kdm en un modèle GWT par le composant précédant, comme discuté dans la section précédente, nous établir un modèle de navigation qui résume la flux de navigation de l'application source GWT. Ce package contient un ensemble de package importants qui sont représenté dans la Figure 52 :

— Package « extractors » : il contient deux analyseurs pour modèle Java et GWT réspectivement. Chaque analyseur représente un package qui contient un ensemble de classe qui s'occuper à d'interagir avec les modèles. Pour une solution qui supporte la maintenabilité et la modularité, nous avons regroupe dans un package, nommé interfaces, un ensemble des interfaces. Ce package joue le rôle d'un médiateur entre l'aspect technique représenté par les analyseur et l'aspect logique par les autres packages.

— Package « tests » : contient la partie de testes pour les visiteurs. la méthodoglie que nous avons appliquée est le TDD[15] pour concevoir les visiteurs du package suivant par un petit pas assuré par une série de tests unitaires fonctionnement des visiteurs.

— Package « visitors » : ce package couvre la solution discuté dans l'ancien chapitre pour détecter le page suivante à une page illustrée par le diagramme d'activité 43. Il se compose de deux sous package où chacun contient l'implémentions du patron de conception visiteur. le package « nextPages » conforme à la partie avant la condition *Si le type d'argument est un type ClassInstanceCreation* dans le diagramme 43. Le deuxième est désigné par « arguments » qui représente la solution à partir de la condition mentionné auparavant.

— Package « stratégies» : ce package isole la logique de représenter le graphe de transitions de l'implémentation des algorithmes dont les détails de création une transition à l'aide du package « visitors ». Il crée le modèle de navigation sous format JSON et également les règles Dot pour visualiser le modèle sous forme d'un graphe.

— Package « runner » : s'occupe à exécuter la classe de base qui amène à la création de modèle de navigation.

15. TDD : Test driven developpement est une stratégie de conception et développement pilotée par une série de tests unitaires

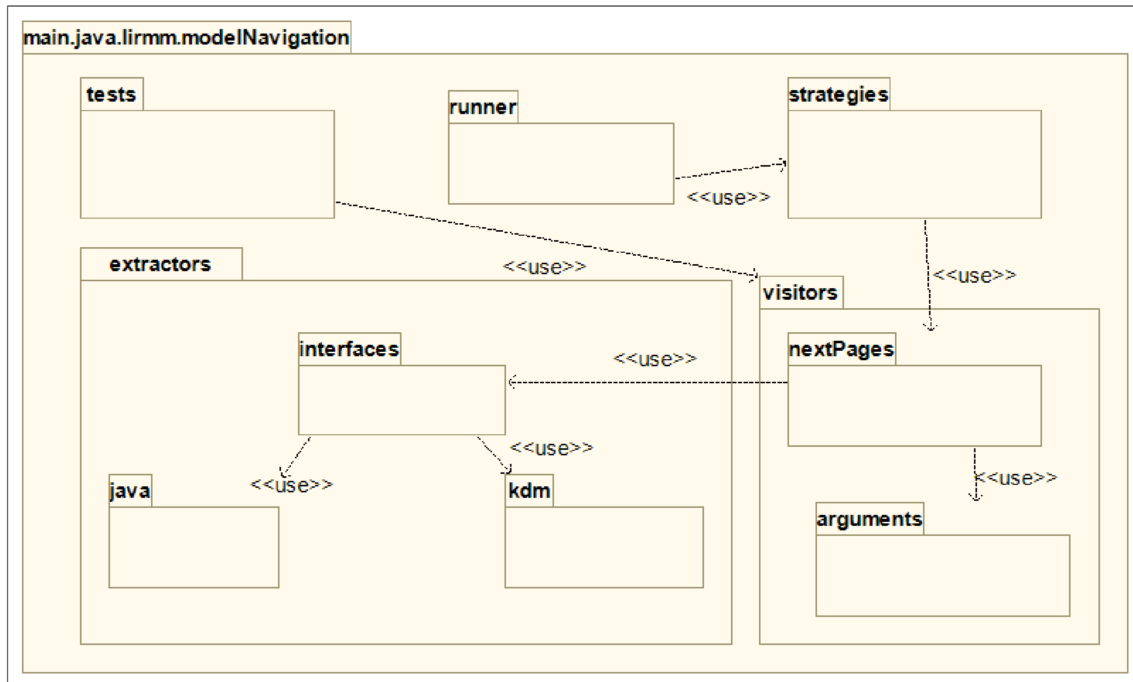


FIGURE 52 – Diagramme de packages de composant navigationConstruction

Application du patron « visiteur » : Le patron de conception Visiteur est un patron de conception comportemental qui vous permet de séparer les algorithmes et les objets sur lesquels ils opèrent(SHVETS p. d.). Ils est structuré en 5 participants comme suit :

1. l'interface Visiteur : s'occupe à déclarer un ensemble de méthodes de parcours qui peuvent prendre les éléments concrets d'une structure d'objets en paramètre.
2. La classe Visiteur Concret : implémente plusieurs versions des mêmes comportements, en fonction des classes des Élément Concret .
3. La interface Élément : déclare une méthode qui « accepte » les visiteurs. Cette méthode déclare un paramètre du type de l'interface visiteur.
4. La classe Élément Concret : implémente la méthode d'acceptation. Le but de cette méthode est de rediriger l'appel vers la méthode appropriée du visiteur en fonction de la classe de l'Élément actuel.
5. Client : représente en général une collection ou tout autre objet

Nous avons utilisé ce patron deux fois. Tous d'abord, pour filtrer l'ensemble des méthodes invoquée et les arguments des méthodes utilisées dans une méthode d'évènement. Par la suite, nous lançons des traitements à l'aide d'autre visiteur sur les éléments récupérées par le premier afin de localiser s'il y a une page sous forme d'une ClassInstanceCreation, SingleVariableAccess, MethodInvocationExpression.

La Figure 53 illustre le premier patron visiteur utiliser spécialement dans le package « nextPages » et la Figure 54 pour le deuxième visiteur. Les tables 6, 7 montre les classes qui ont participé dans les deux patrons visiteurs respectivement :

Classes du package	Participants du pattern visiteur				
	Visiteur	Visiteur C	Élément	Élément C	Client
Visitor	✓				
NextPageVisitor		✓			
Acceptor			✓		
InjectedInstance				✓	
MethodInvocationInstance				✓	
NextPageService					✓

Table 6 – Participation de classes du package « nextPages » dans le patron visiteur 53

Classes du package	Participants du pattern visiteur				
	Visiteur	Visiteur C	Élément	Élément C	Client
ExpressionInspector	✓				
ExpressionInspection		✓			
CoverExpressionInArguments			✓		
ClassInstanceCreationExpression				✓	
SingleVariableAccessExpression				✓	
MethodInvocationExpression				✓	
NextPageVisitor					✓

Table 7 – Participation de classes du package « arguments » dans le patron visiteur 54

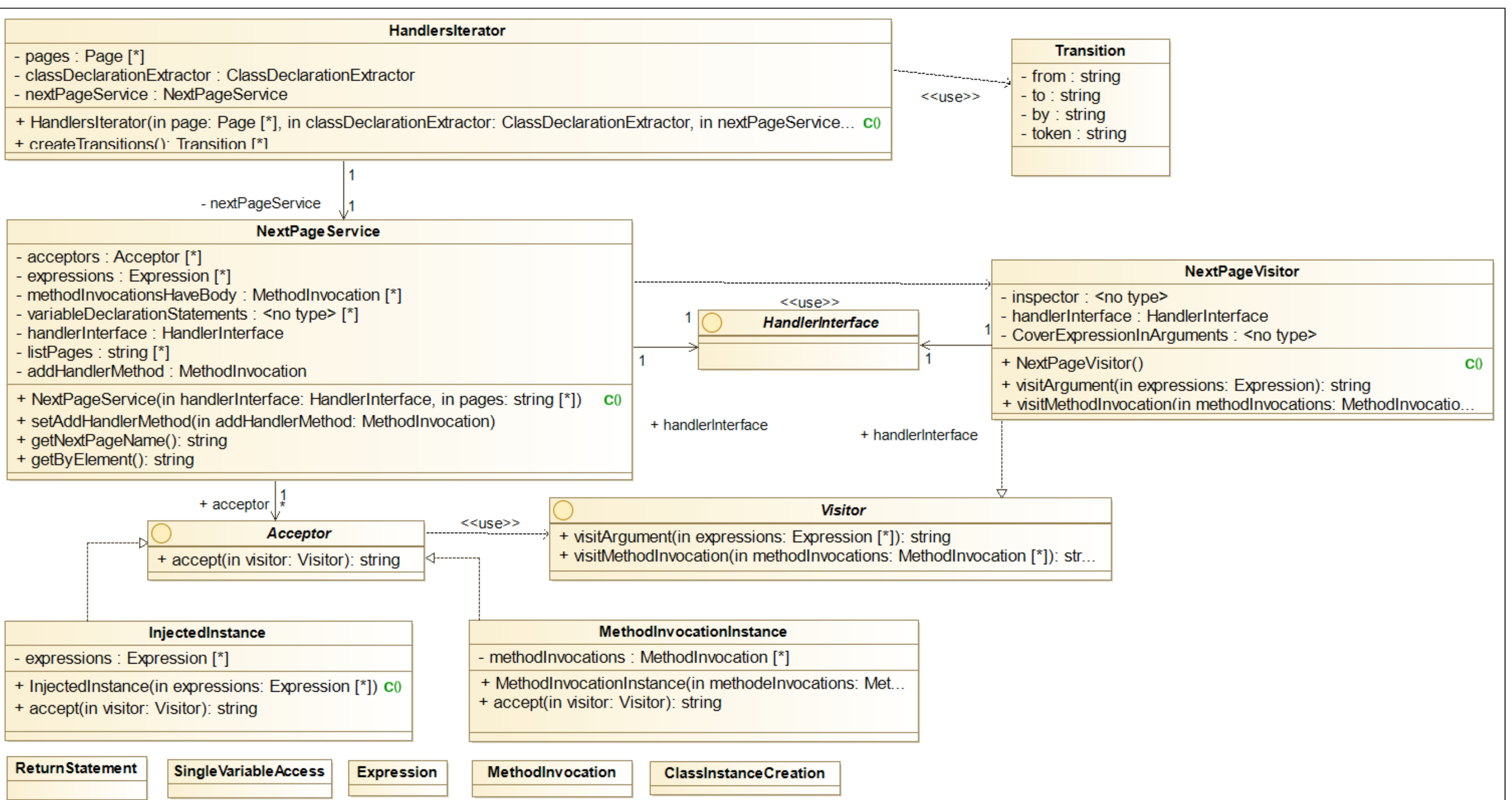


FIGURE 53 – Diagramme de classes du sous package « nextPages » représentant le patron de conception visiteur 6

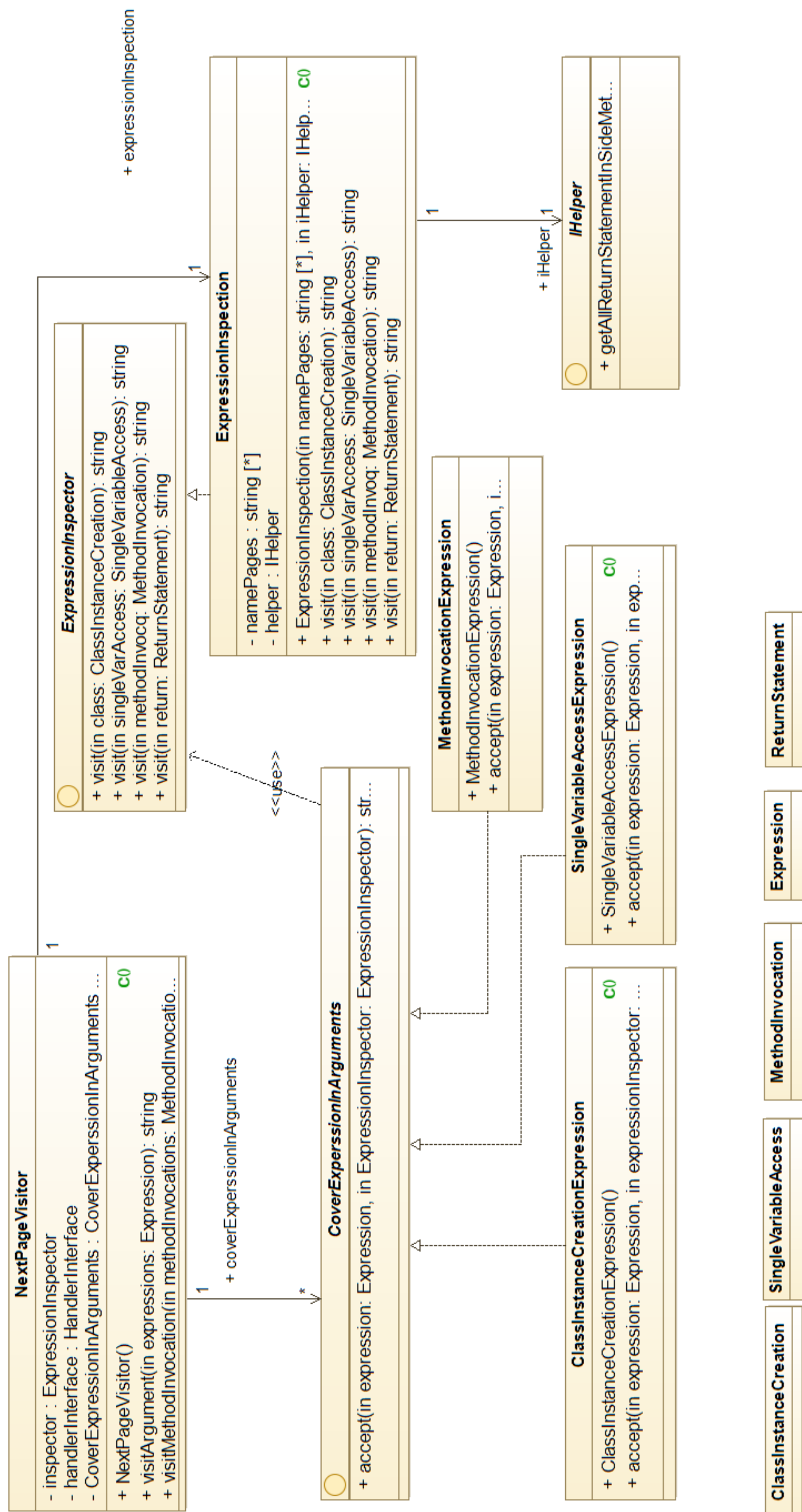


FIGURE 54 – Diagramme de classes de sous package « arguments » représentant le patron de conception visiteur 7

Application du patron « Stratégie » : Stratégie est un patron de conception comportemental qui permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables(Shvets p. d.). sa structure comprend quatre éléments importants :

— Contexte : garde une référence vers une des stratégies concrètes et communique avec cet objet uniquement à travers une interface stratégie.

— Stratégie : est commune à toutes les stratégies concrètes. Elle déclare une méthode que le contexte utilise pour exécuter une stratégie.

— Stratégie Concrète : implémente une variante d'algorithme déclarée dans la Stratégie et utilisée par le contexte.

— Client : s'occupe à créer l'objet Stratégie et le passe au contexte. Le contexte expose une méthode qui permet aux clients de remplacer la stratégie associée au contexte lors de l'exécution.

Nous voulons, à partir des transitions créées, générer deux modèles : 1) un modèle sous format json décrivant les transtions et 2) un modèle qui décrit les transitions utilisant le langage DOT [16] pour être afficher sous forme d'un graphe au niveau du navigateur. La seule différence entre les deux est la façon d'exécuter un comportement de construction des deux modèles. Pour cela, nous utilisons le patron stratégie qui permet de séparer le code, les données internes (transitions) et les dépendances des divers algorithmes du reste du code. La Figure 55 représente le diagramme de classe de patron stratégie que nous avons utiliser. De plus, la table suivante illustre le rôle du chaque classe dans le diagramme.

Classes du package	Participants du pattern stratégie			
	Contexte	Stratégie	Stratégie C	Client
NavigationModelCreator	✓			
NavigationModelStrategy		✓		
DotRules			✓	
JsonModel			✓	
NavigationModelGenerator				✓

Table 8 – Participation de classes du package « stratégies » 4.2.1 dans le patron stratégie 55

16. DOT : un langage de description de graphe dans un format texte.https://graphviz.org/doc/info/lang.html

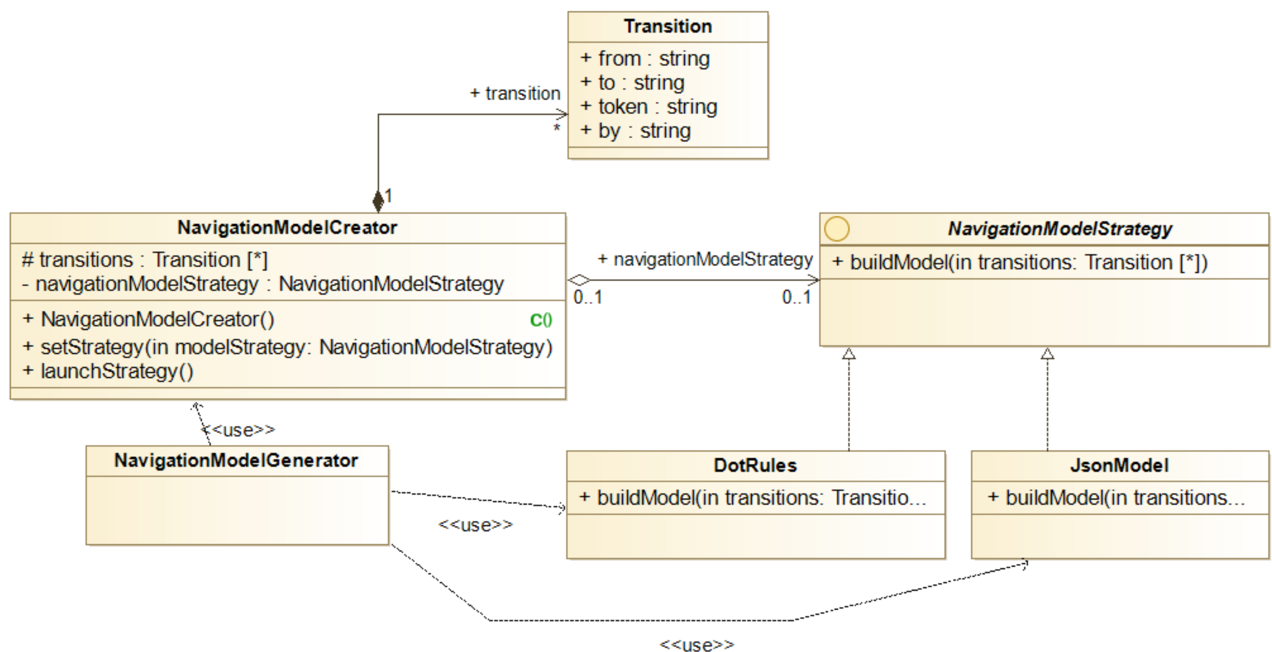


FIGURE 55 – Diagramme de classes représentant le patron Stratégie 8

Composant « angularGenerator »

Il contient les packages suivants, représenté par le diagramme 56 :

— Package « extractors » : se compose de quatre packages les suivants : json, gwt, java et helpers. Ce dernier va jouer le rôle d'un médiateur entre les classes techniques et d'autres de haut niveau à travers la création du modèle intermédiaire. Les autres sous-package ou chacun contient un analyseur pour um modèle Java, GWt et modèle de navigation.

— Package « angular » : exécuter les commandes d'Angular (ng) pour configurer l'environnement de l'application qui sera généré. Pour chaque page il crée un module et pour chaque widget correspond à un composant.

— Package « iterators » : s'occupe à remplir les composants créés précédemment par un code en invoquant à des composantes dans la bibliothèque que nous avons créé (voir cette partie).

— Package « runner » : sert à exécuter le processus de génération de l'application cible.

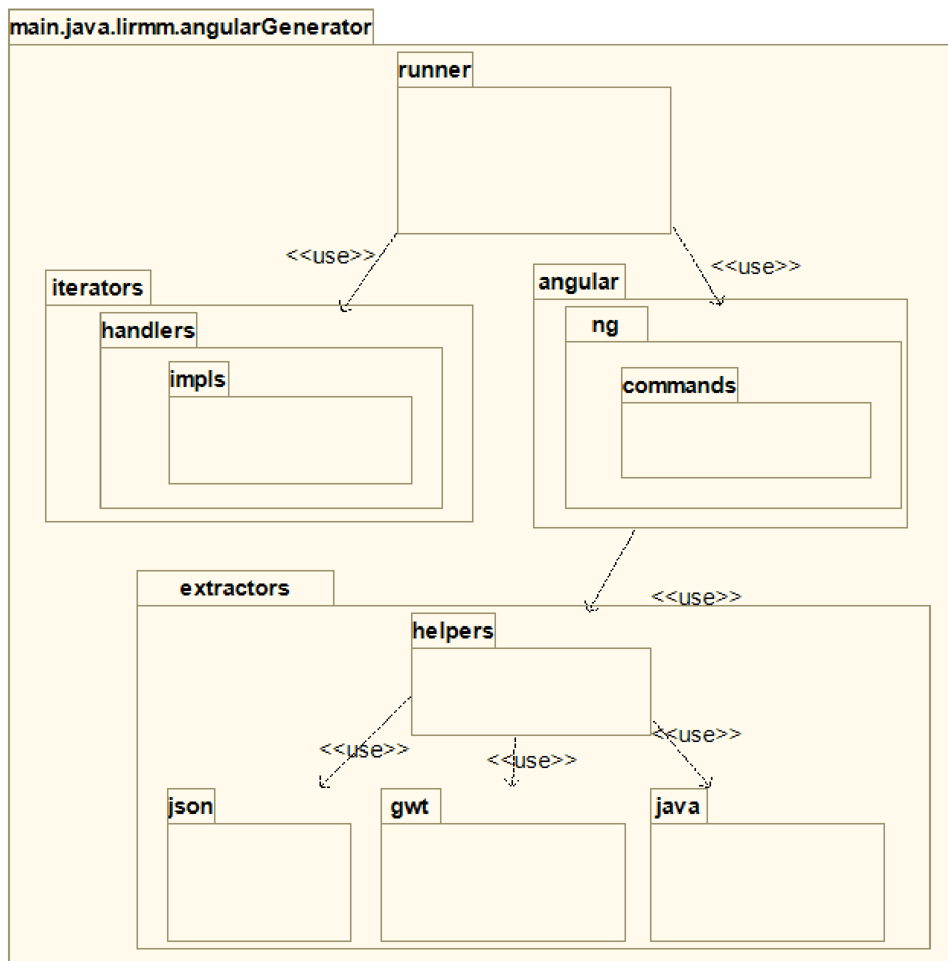


FIGURE 56 – Diagramme de package du composant « angularGenerator »

Application du patron « chaîne de responsabilité » : C'est un patron de conception comportemental qui permet de faire circuler des demandes dans une chaîne de traiteur. Lorsqu'un traiteur reçoit une demande, il décide de la traiter ou de l'envoyer au traiteur suivant de la chaîne (SHVETS p. d.). Ce patron comprend les éléments suivants :

- Traiteur : déclare une interface pour gérer les demandes, mais il peut parfois en contenir une autre pour désigner le prochain Traiteur de la chaîne.
- Traiteur Concret : contiennent le code qui traite les demandes. Lors de la réception d'une demande, chaque handler décide s'il doit la traiter et s'il doit l'envoyer plus loin dans la chaîne. Les handlers sont généralement autonomes et non modifiables, et n'accepteront qu'une seule fois les données nécessaires par le biais du constructeur.
- Client : peut créer les chaînes juste une fois ou les assembler dynamiquement en fonction de la logique métier.

La table 9 résume la participation des éléments du package « iterators » 56 dans ce patron de conception. Le besoin de ce patron tombe avec notre logique que nous avons établie dans l'étude de cas 5.

Classes du package	Participants du pattern chaîne de responsabilité		
	Traiteur	Traiteur Concret	Client
Handler	✓		
* tous qui implémente Handler		✓	
ComponentCodeGenerator			✓

Table 9 – Participations des classes de package « iterators » dans le patron chaîne de responsabilité présenté dans 57

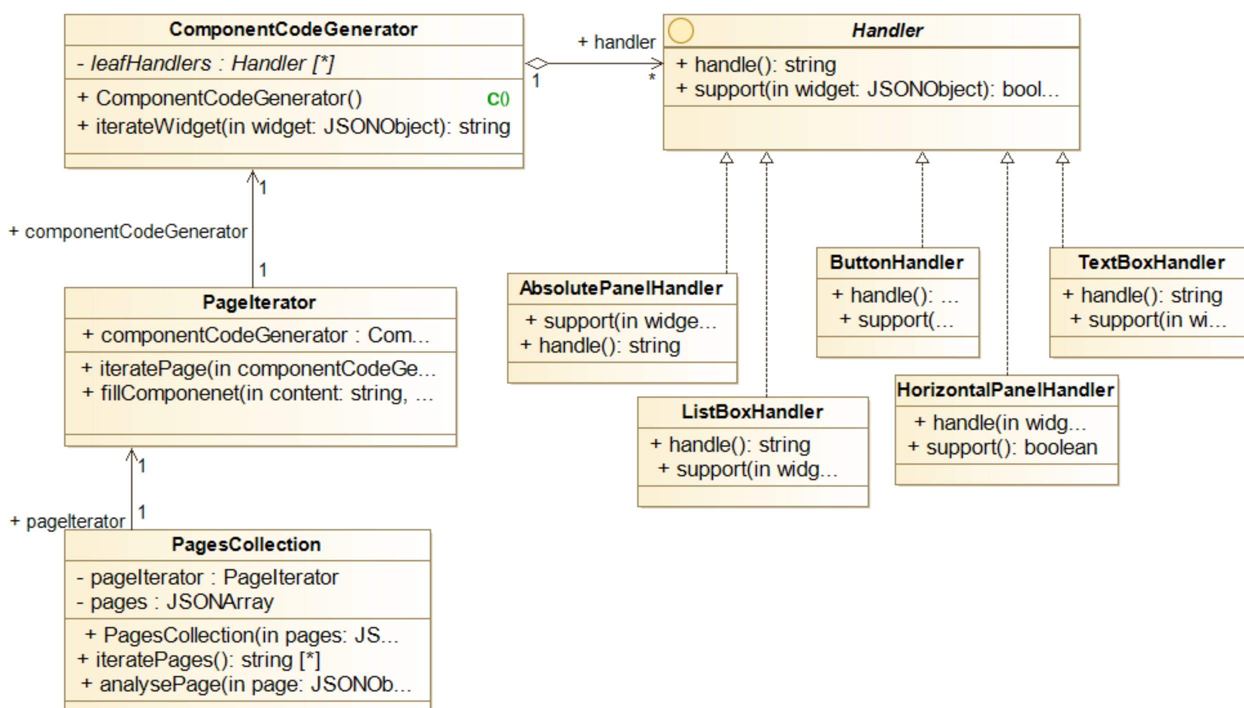


Figure 57 – Patron chaîne de responsabilité pour la génération du contenu d'un composant en Angular

Application du patron « Template method » C'est un patron de conception comportemental qui permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure (Shvets p. d.). Ce patron se compose d'une :

— Classe Abstraite : déclare des méthodes qui représentent les étapes d'un algorithme et une méthode spécifique qui appelle toutes ces méthodes dans un ordre.

— Classe Concrète : peuvent redéfinir toutes les étapes, mais pas la méthode spécifique de la classe abstraite.

La table 10 illustre la particpation des éléments de package « angular » dans la configuration de l'environnement de l'application Angular qui sera généré. Le besoin derrière l'utilisation de ce patron est que le Framework Angular principalement propose

plusieurs configuration pour configurer l'environnement de développement. D'autre coté, nous comptons au plus tard étendre les étapes que nous avons choisie pour l'instant et qui sont définie dans la classe abstraite AngularConfiguration 58. La méthode de patron est nommée *doConfiguration*.

Classes du package	Participants du pattern chaîne de responsabilité		
	Classe Abstraite	Classe Concrète	Client
AngularConfiguration	✓		
DefaultAngularConfiguration		✓	
GeneratorRunner			✓

TABLE 10 – Participation des classes de package « angular » dans le patron de méthode

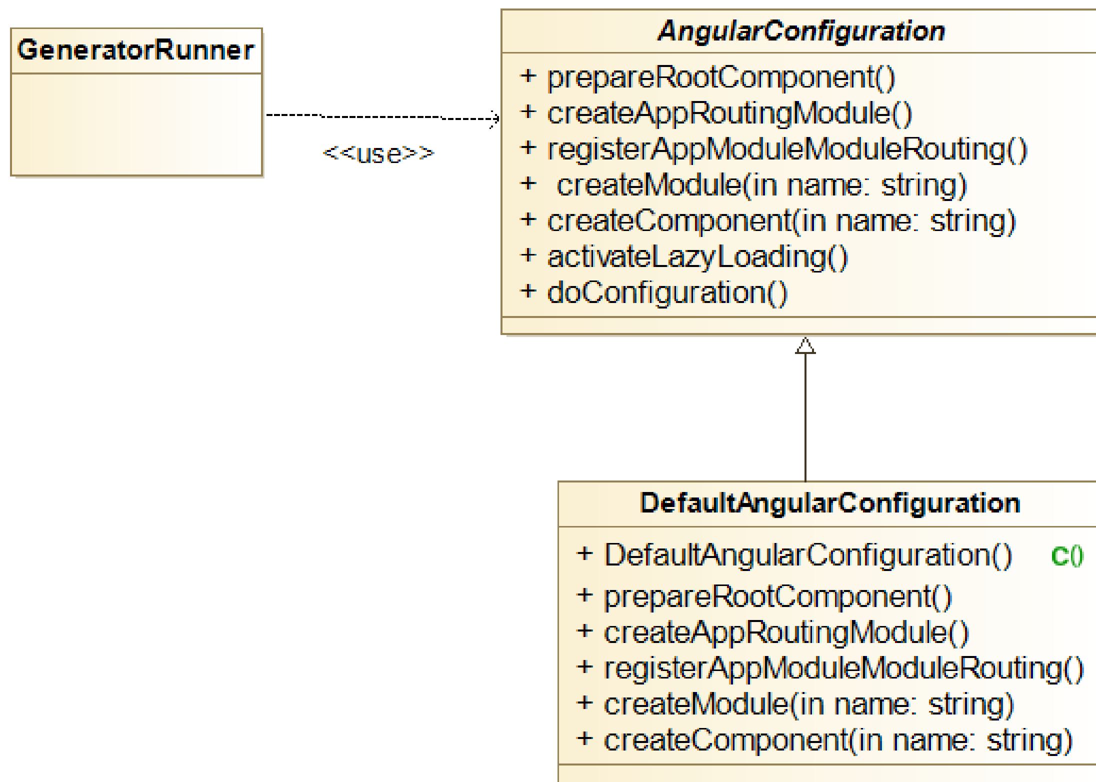


FIGURE 58 – Patron de méthode pour la configuration d'environnement d'application Angular généré

5 Conclusion

Nous avons décrit au cours de ce chapitre des différentes parties conceptuelles pour mettre en œuvre un processus de migration d'une application GWT vers Angular. D'abord, nous avons exposé la solution d'une manière globale suivie d'une étude théorique par rapport à la migration d'une application GWT vers Angular. Dans ce cadre, nous avons présenté différents algorithmes à implémenter dans la phase de réalisation à l'aide des patrons de conception. Par la suite, nous avons décrit l'aspect architectural et conceptuel à travers les diagrammes de classes pour les patrons de conceptions utilisés, y compris les diagrammes de packages pour une vue modulaire.

Nous présentons dans le prochain chapitre la partie réalisation, tests et résultats où nous exposons les différents outils et technologies utilisés.

CHAPITRE 6

RÉALISATION, TESTS, RÉSULTATS ET VERROUS TECHNIQUES

Sommaire

1 **Introduction** **103**
2 **Outils utilisés** **103**
2.1 Langage de programmation Java 103
2.2 ATL 104
2.3 Angular Material 104
2.4 Environnement de développement 105
2.4.1 Eclipse Modeling Tools IDE pour le langage Java . . . 105
2.4.2 PhpStorm IDE pour Angular 106
2.5 Environnement de collaboration 106
2.5.1 Git/Github 106
2.5.2 Jira 106
3 **Jeu de données des applications GWT** **107**
3.1 Choix des applications et téléchargement du code source 108
3.2 Annotation manuelle 108
3.3 Analyse du qualité du code 109
3.4 Structuration des applications 110
4 **Expérimentation des heuristiques et résultats** **111**
4.1 Synthèse des résultats du groupe de 23 applications GWT : . . 112
5 **Test bout-en-bout (e2e)** **113**
5.1 Extraction du Modèle Java et Modèle KDM 113
5.2 Transformation du Modèle KDM vers Modèle GWT 115
5.3 Construction d'un modèle intermédiaire 115
5.4 Génération d'un code Angular 117
6 **Conclusion** **119**

1 Introduction

Dans la phase de conception, nous avons pris connaissance de la conception de notre solution. Pour cela, nous avons exploré la conception de chaque phase de migration de cas d'étude de GWT vers Angular, présenté les diagrammes explicatifs sur nos démarches et abordé les concepts fondamentaux. Nous passons maintenant à la phase de réalisation et test de la solution. Dans cette phase, nous exposons les outils qui ont permis de réaliser notre solution. Ensuite, nous présentons un jeu de données que nous avons construits à partir du « Google Code Archive ». Puis, nous montrons les expérimentations effectuées sur des applications GWT pour tester les heuristiques établies précédemment. Après, nous réaliserons un test de bout en bout pour valider effectivement la migration d'une application GWT et pouvoir ainsi valider le passage à l'échelle (scalabilité) des composantes de migrations (voir la figure 50) .

2 Outils utilisés

2.1 Langage de programmation Java

Java est un langage de programmation orienté objet. Il a la propriété d'un langage multi plates-formes qui permet aux développeurs d'applications d'écrire une fois, d'exécuter n'importe où grâce au byte code (code compilé Java). Nous avons utilisé Java pour implémenter notre solutions dans tous les composants.

Figure 59 – Logo Java

2.2 ATL

ATL (ATLAS Transformation Language)[1] est un langage et une boîte à outils de transformation de modèle. Dans le domaine de l'ingénierie dirigée par les modèles (IDM), ATL fournit des moyens de produire un ensemble de modèles cibles à partir d'un ensemble de modèles sources. Développé sur la plate-forme Eclipse, l'environnement intégré ATL (IDE) fournit un certain nombre de développements standard. des outils (mise en évidence de la syntaxe, débogueur, etc.) visant à faciliter le développement des transformations ATL. Le langage ATL s'appuie sur la norme OMG OCL (Object Constraint Language)[2] pour définir ses types de données que pour ses expressions déclaratives. Il existe quelques différences mineures entre la définition OCL[3] et l'implémentation ATL actuelle.

FIGURE 60 – Logo ATL

2.3 Angular Material

Angular Materials[4] est un ensemble de composants sous forme d'une bibliothèque Angular[5] dédies pour l'interface qui aide à concevoir une application de manière moderne.Ils donnent également la possibilité de concevoir les applications de manière attrayante, avec des styles et des formes uniques. Ces composants les rendent plus cohérentes, rapides, polyvalentes et réactifs. Nous avons reposé sur ces composants pour ré-implémenter les éléments de GWT en Angular.

1. https://www.eclipse.org/atl/
2. https://wiki.eclipse.org/ATL/User_Guide_\protect\discretionary{\char\hyphenchar\font}{}{}_The_ATL_Language
3. https://www.omg.org/spec/OCL/2.4/PDF
4. Disponible sur : https://material.angular.io/
5. Disponible sur : https://angular.io/

Figure 61 – Logo Angular Material

2.4 Environnement de développement

2.4.1 Eclipse Modeling Tools IDE pour le langage Java

Eclipse[6] principalement est un environnement de développement intégré libre dont le but est de fournir une plate-forme modulaire pour permettre de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. L'architecture d'Eclipse base sur la notion de extensions. Nous utilisons l'extension Eclipse Modeling Tools[7]. Celui-là contient un framework et des outils pour exploiter les modèles : un modeleur graphique Ecore (diagramme de type classe), utilitaire de génération de code Java pour les applications de modélisation EMF[8], prise en charge de la comparaison de modèles et transformation des modèles.

Figure 62 – logo d'Eclipse

Figure 63 – logo Eclipse Modeling Tools

Figure 64 – logo Eclipse Modeling FrameWork

6. https ://www.eclipse.org/
7. https ://www.eclipse.org/downloads/packages/release/2021-09/r/eclipse-modeling-tools
8. https ://www.eclipse.org/modeling/emf/

2.4.2 PhpStorm IDE pour Angular

PhpStorm [9] est un environnement de développement pour éditer principalement les projets PHP [10] comme son l'indique, lancé par JetBrains [11]. Il a plusieurs extensions pour supporter les projets JavaScript/TypeScript tels que Angular, React, Vue. Il supporte également les gestionnaires de de dépendance comme Composer, npm. Parmi les avantages de cet éditeur,il possède une auto-complétion dynamique, rapide et intelligente pour le code. Le but d'utiliser cet éditeur est de faciliter la fixation des bugs mineurs qui peuvent nous sortir de la génération automatique du code Angular.

(a) Logo de PhpStorm

(b) Logo d' Angular

2.5 Environnement de collaboration

2.5.1 Git/Github

C'est un système de gestion de versions complet et très performant. Nous avons utilisé cet outil pour la gestion de versions du code source de différents composantes développées.

FIGURE 66 – Logo Github

2.5.2 Jira

Jira [12] est un outil de collaboration qui organise nos tâches de projet en tableaux et suit l'évolution de l'implémentation pour détecter les éventuelles bugs. En un coup d'œil, Jira indique sur les tâche de la semaine, les tâche encours et les développeurs affectés aux tâche. Nous avons utilisé cet outil pour le suivi de notre projet en utilisant la méthodologie KANBAN [13].

9. https ://www.jetbrains.com/phpstorm/
10. https ://www.php.net/
11. https ://www.jetbrains.com/
12. `https://www.atlassian.com/fr/software/jira`
13. La méthode kanban est une méthodologie agile simple pour la gestion du projet, visuelle et facile à comprendre. Elle est basée sur le principe du « juste à temps » http ://www.albert-deloin.fr/kanban.htm

Figure 67 – Logo Jira

3 Jeu de données des applications GWT

Un jeu de données est important pour tester les heuristiques que nous avons conçus précédemment tels que :

1. détection des éléments GWT utilisées dans une application.
2. détection des pages d'une application GWT.
3. détection des transitions entre les pages.
4. détection d'élément racine de chaque page.

Pour la création de du jeu de données, nous avons effectué plusieurs étapes permettant d'obtenir plusieurs représentations nécessaires à l'extraction de la forme finale utilisée pour tester ces heuristiques. La Figure 49 illustre une vue globale du processus suivi pour la création du jeu de données.

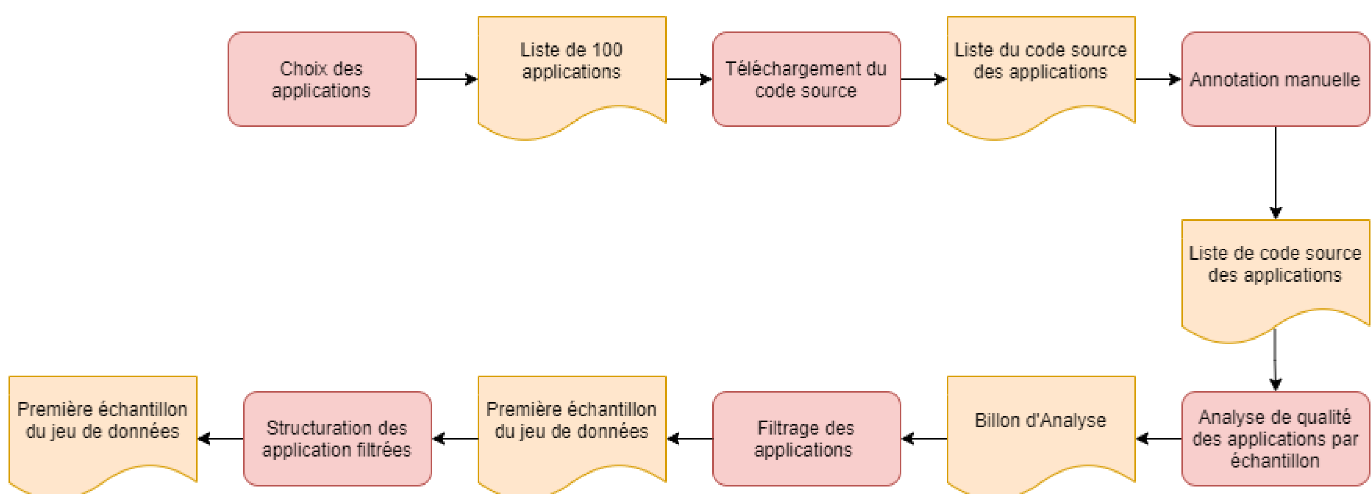


Figure 68 – Vue globale de la création du jeu de données

Dans ce qui suite, nous expliquons les étapes présentées dans la Figure 68 telles que : Choix des applications et téléchargement du code source, Annotation manuelle, Analyse du qualité du code et Structuration des applications.

3.1 Choix des applications et téléchargement du code source

D'abord, nous commençons par choisir 100 applications GWT à partir de plateforme Google Code Archive [14] qui était largement utilisé par la communauté des projets open source avant 2016. Ensuite, nous téléchargeons par la suite le code source de ces applications.

3.2 Annotation manuelle

Après, nous effectuons l'annotation manuelle du jeu de données pour obtenir une première forme. Cette forme comprend la description de nos données par les attributs suivants :

- *Lien d'application* : les applications qui ont été choisies sont des projets de développement open sources dont le code est disponible sur la plateforme Google Code Archive. Ce champ permet d'accéder à leurs code source.
- *Type d'application* : le développement d'une application GWT peut avoir plusieurs manière d'implémentation par exemple : utilisant XML pour programmer l'interface graphique nommée UiBinder ou le langage Java ou des objets JavaScript pour manipuler le DOM.
- *L'année de la dernière validation (commit)* : Ce champ nous permet de voir la dernière version de l'application pour mesurer l'âge de l'application.
- *Email du développeur de l'application* : Les coordonnées du propriétaire de l'application est indispensable, par exemple pour connaître quel style architectural utilisé, le droit de modification et l'exploitation dans notre étude.

type d'application	Lien d'application	L'année du validation	Author
java	https://code.google.com/archive/p/spelstegen/	2006	
java	https://code.google.com/archive/p/schoolscooperative/	2006	
java	https://code.google.com/archive/p/talklively/	2004	
java	https://code.google.com/archive/p/market-tahboaliha/	2010	
java	https://code.google.com/archive/p/issue-tracker/	2011	
java	https://code.google.com/archive/p/quati-project/		
java	https://code.google.com/archive/p/larbc/		
java	https://code.google.com/archive/p/gwt-mvp-sample/		
java	https://code.google.com/archive/p/grandpasgames/		
java	https://code.google.com/archive/p/gwt-games		
java	https://code.google.com/archive/p/helloseries/		
java	https://code.google.com/archive/p/satellitecloud		
java	https://github.com/ptanov/rent-rooms		
Uibinder	https://code.google.com/archive/p/plumb		

FIGURE 69 – Échantillon du jeu de données

La Figure 69 une partie Échantillon du jeu de données que nous avons annoté et la Figure 70 illustre la distribution des types des applications dans l'ensemble de 1000

14. Disponible sur : `https://code.google.com/archive/search?q=domain:code.google.com%20label:GWT`

applications.
Une note fait état importante, pendant le révision manuelle des applications (review) sous Eclipse, nous avons remarqué que certains applications possèdent des références ou dépendances vers d'autres projets externes et comme ces applications sont bien héritées par exemple certaines avaient des validations en année 2004, 2006, donc il est difficile d'accéder à leurs dépendances même avec l'utilisation d'un gestionnaire de dépendance comme Maven[15] ou Gradle[16]. La Figure suivante montre la distribution de types des applications GWT sélectionnées depuis Google Code Arhive.

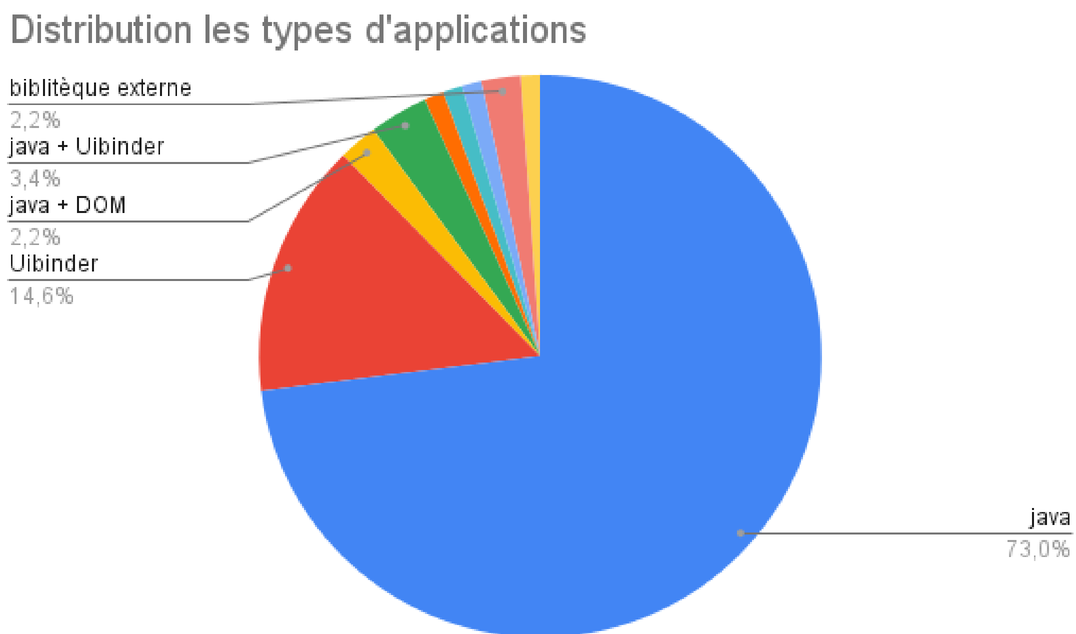


Figure 70 – Distribution les types d'applications

3.3 Analyse du qualité du code

Pour l'instant nous avons choisi un ensemble primaire seulement de 23 applications, nous analysons leurs code pour obtenir une description sur le nombre de classes et nombre de ligne pour chacune. L'outil utilisé dans cette partie est CLOC[17].

Applications	Métriques	
	Nbr Lignes	Nbr Classes
app-gwt-1	2249	27
app-gwt-2	386	12
app-gwt-3	1447	14

15. Disponible sur : https ://maven.apache.org/
16. Disponible sur : https ://gradle.org/
17. disponible sur :http://cloc.sourceforge.net/

app-gwt-4	574	23
app-gwt-5	1010	32
app-gwt-6	723	8
app-gwt-7	975	7
app-gwt-8	706	7
app-gwt-9	1266	6
app-gwt-10	1877	28
app-gwt-11	772	11
app-gwt-12	448	11
app-gwt-13	14576	161
app-gwt-14	670	13
app-gwt-15	1075	20
app-gwt-16	967	25
app-gwt-17	5415	94
app-gwt-18	1124	12
app-gwt-19	4315	72
app-gwt-20	329	6
app-gwt-21	524	22
app-gwt-22	1760	29
app-gwt-23	193	2
totale =	43381	642

TABLE 11 – Nombre de Classes et lignes pour chaque application

3.4 Structuration des applications

Dans cette étape, nous essayons de structurer manuellement les applications que nous avons sélectionnées auparavant (23 applications) dans une même structure des répertoires pour les rendre dans la forme standard d'un projet GWT pour cela nous prenons que la partie client développement en GWT pour des applications qui se basent sur des environnement différent comme spring boot/python pour la partie serveur.

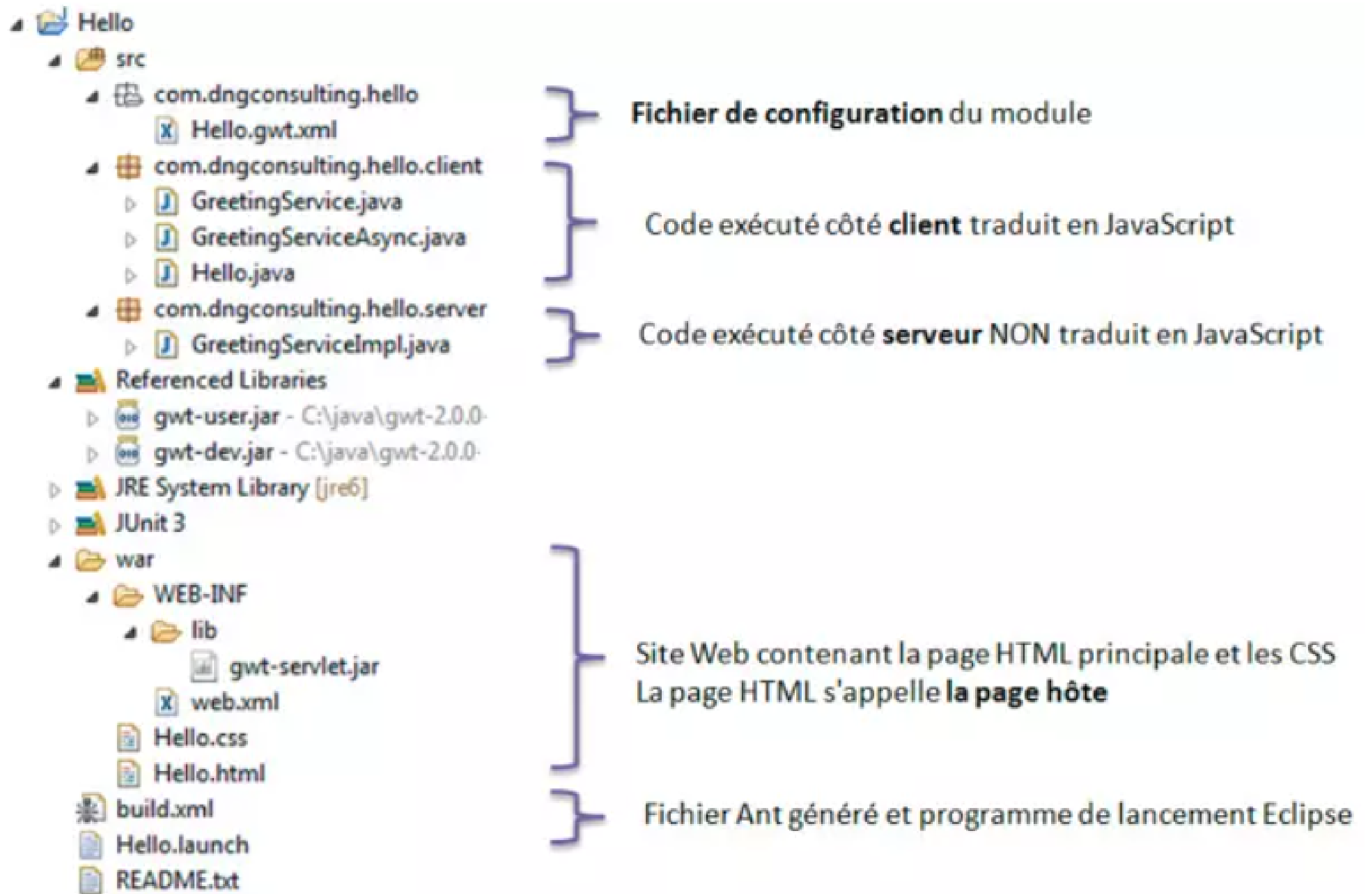


FIGURE 71 – Structure d'un projet GWT (JDN 2012)

4 Expérimentation des heuristiques et résultats

Dans l'optique d'expérimenter les heuristiques d'extraction afin de mesurer leurs performances, nous devons choisir des indicateurs d'évaluation pertinents. Les indicateurs choisi sont (le terme "classe i" dans les équations suivantes veut dire l'application GWT i, les éléments dans les quations peuvent être des pages, des éléments graphiques, élément racine et transitions) :

Précision (Precision) : est la proportion des éléments pertinents parmi l'ensemble des éléments proposés.

$$Precision_i = \frac{\text{nombre d'éléments correctement attribués à la classe i}}{\text{nombre d'éléments attribués à la classe i}}$$

Rappel (Recall) : est la proportion des éléments pertinents proposés parmi l'ensemble des éléments pertinents.

$$Rappel_i = \frac{\text{nombre d'éléments correctement attribués à la classe i}}{\text{nombre d'éléments appartenant à la classe i}}$$

Indicateurs \ Heuristiques	Widgets	Pages	Root Widget	Transitions
$Precision_t$	99%	79,3%	95%	99,9%
$Rappel_t$	87,9%	98%	85,8%	63,7%

Table 12 – Les Résultats de l'extraction

$$Precision_t = \frac{\sum_{n=1}^{n} Precision_i}{n}$$

$$Rappel_t = \frac{\sum_{n=1}^{n} Rappel_i}{n}$$

Le tableau 12 présente les résultats globaux de nos expérimentations sur l'ensemble des applications GWT choisies dont les colonnes (Widgets, Pages, Root Widget, Transitons) sont les heuristiques et les lignes sont les indicateurs (Précision totale, Rappel totale). Les valeurs du tableau 12 sont calculées d'une manière manuelle car les applications que nous avons pris ne possèdent pas de documentation (même pas des écrans photos), donc nous ont été obligé de lire manuellement leurs code source et par la suite ressortir les résultats attendus avant d'exécuter les heuristiques.

4.1 Synthèse des résultats du groupe de 23 applications GWT :

À travers l'expérimentation des heuristiques en fonction des indicateurs le nous avons constaté ce qui suit :

— L'heuristique d'extraction d'éléments graphiques a réussi à distinguer entre les types primitives et les type des widgets de GWT dans l'ensemble de variables d'une applications. Cette résultat ressort de l'utilisation de l'arbre d'héritage pour distinguer les types des variables utilisées. Cependant, elle n'a pas pu détecter une partie d'éléments car lorsque nous avons analysé les applications qui sont la source du problèmes, nous avons trouvé que certaines développeurs instancier directement la classe d'élément graphique sans l'affecter à une variable.

— L'heuristique d'extraction des pages d'une application GWT nous retourne des fausses pages. Cela peut causer des problèmes au moment de génération de l'application (explosion du code source), donc la solution est de filtrer l'ensemble de pages avant la génération par l'intersection avec l'ensemble des pages qui sont dans les transitions.

— L'heuristique d'extraction d'élément racine de chaque page a réussi presque dans toutes les applications mais dans certaines n'a pas pu marcher d'une manière sensible à cause de l'héritage entre les pages par exemple : nous avons que certains applications comportent des pages qui héritent des classes abstraites qui définissent son propre élément racine.

— L'heuristique d'extraction donne des bons résultats sur les applications GWT simples mais elle ne donne pas des résultats efficaces avec le style MVP [18] et PlaceActivites [19]. Cette heuristique a besoin de critères pour traiter les deux styles citées.

5 Test bout-en-bout (e2e)

Les tests de bout en bout permettent de vérifier toutes les facettes et composantes de notre solution pour une migration d'une application GWT vers une application Angular. Pour notre test, nous choisissons une application GWT au hasard sur github disponible sous `https://github.com/manolo/gwt-stockwatcher`. Dans les sous-sections suivantes, nous expliquons chaque étape de migration de l'application. Le figure 72 représente partie visuelle de cette application GWT.

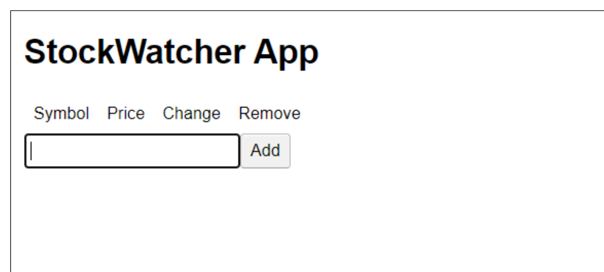


FIGURE 72 – Application Simple GWT mono-page `https://github.com/manolo/gwt-stockwatcher`

5.1 Extraction du Modèle Java et Modèle KDM

Nous utilisons l'outil Modisco 4.4 pour extraire les deux modèles. Le type de l'utilisation est manuelle pour l'instant comme décrit précédemment. Dans l'annexe , il existent une image descriptive sur l'utilisation. Les Figures suivantes 73 et 74 illustrent respectivement des extraits de modèle Java et KDM.

18.
19.

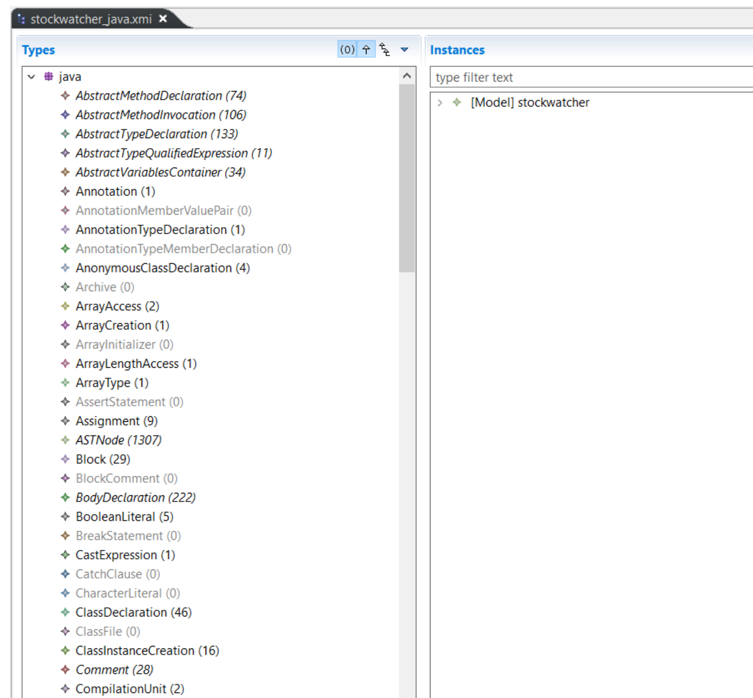


FIGURE 73 – Extrait du modèle Java de l'application généré par Modisco

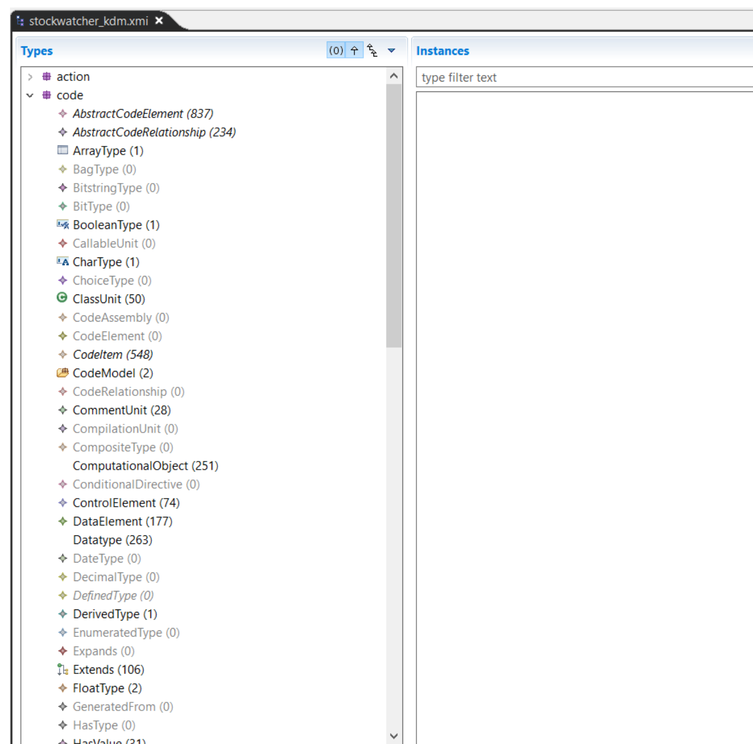


FIGURE 74 – Extrait du modèle KDM de l'application généré par Modisco

5.2 Transformation du Modèle KDM vers Modèle GWT

Après que nous avons pu extraire le modèle KDM, nous le passons au composant Transformation. La figure 75 illustre le modèle résultat de cette transformation.

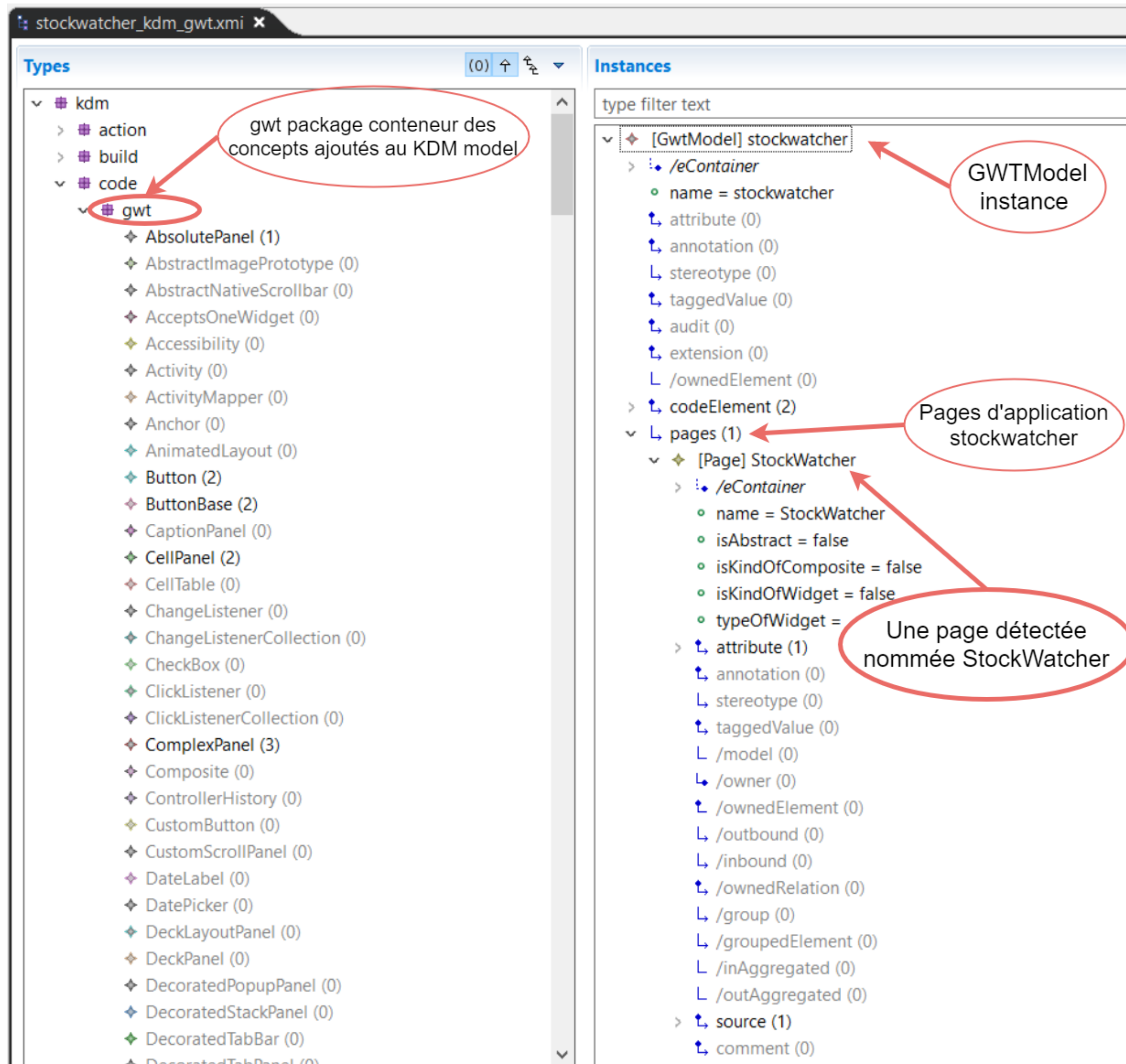


FIGURE 75 – Extrait du modèle GWT

5.3 Construction d'un modèle intermédiaire

L'application que nous avons choisi contient qu'une seule page. Donc à partir du modèle Java, modèle GWT et modèle de transition, nous les passons vers le composant de génération. Celui-là les transforme vers un modèle intermédiaire. Les Figures 76 et 77

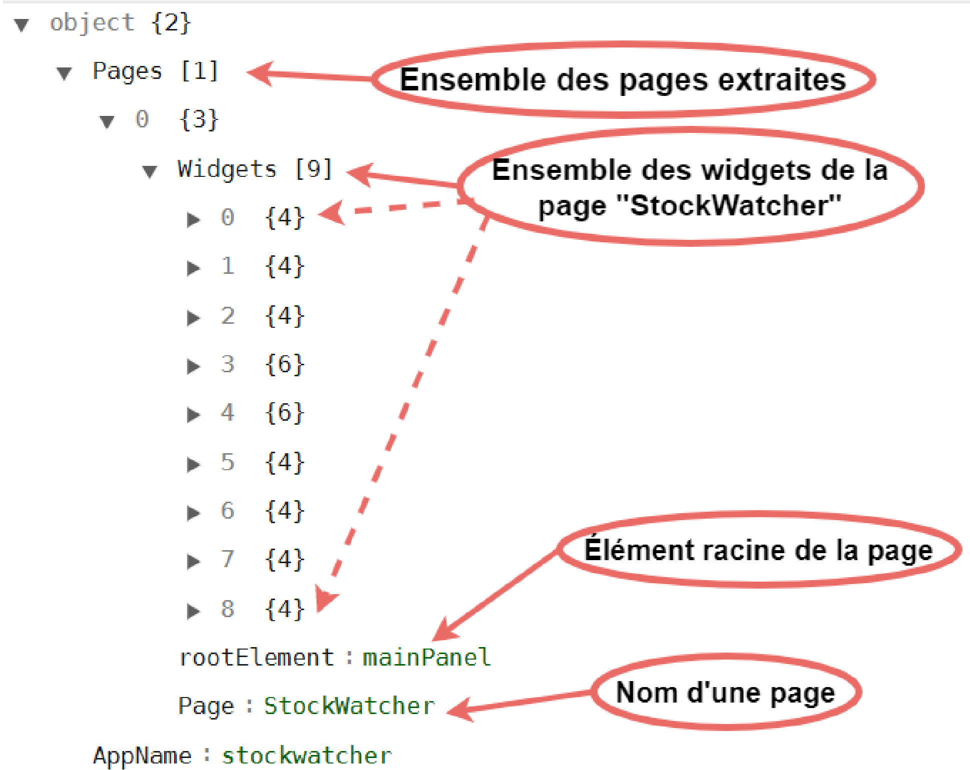


Figure 76 – Extrait 1 du modèle intermédiaire

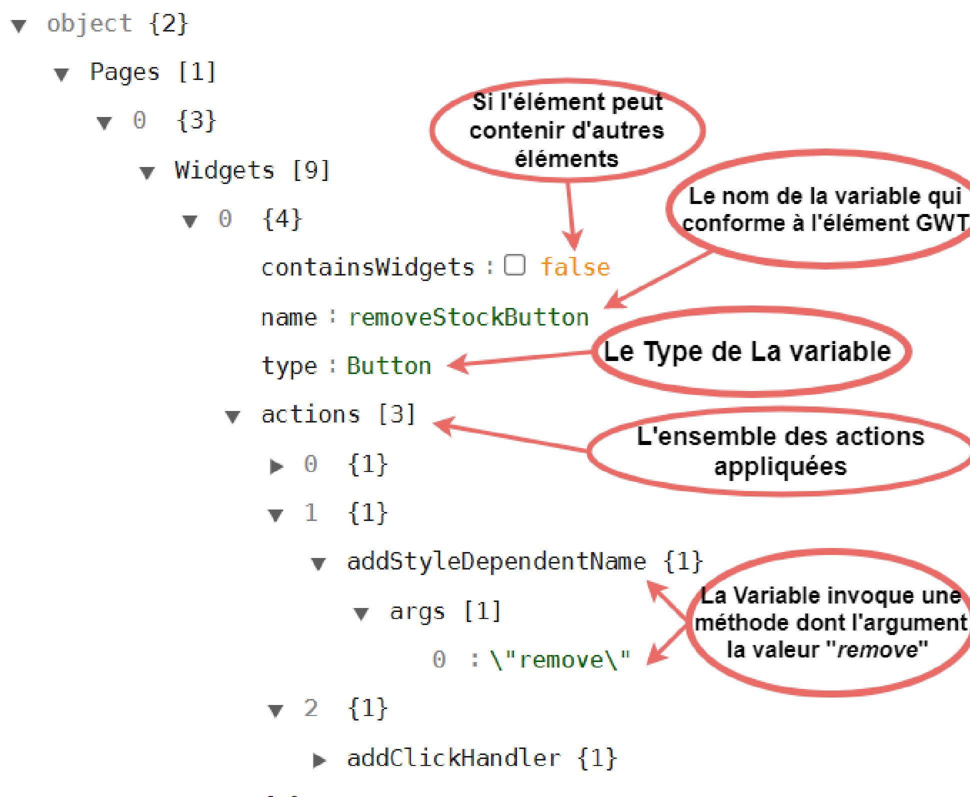


Figure 77 – Extrait 2 du modèle intermédiaire

5.4 Génération d'un code Angular

Après que le modèle intermédiaire est obtenu, nous passons à la génération du code source en Angular. La figure 78 illustre l'infrastructure de développement et la figure 79 montre le code généré ainsi que la figure 80 présente le résultat d'exécution de l'application généré.

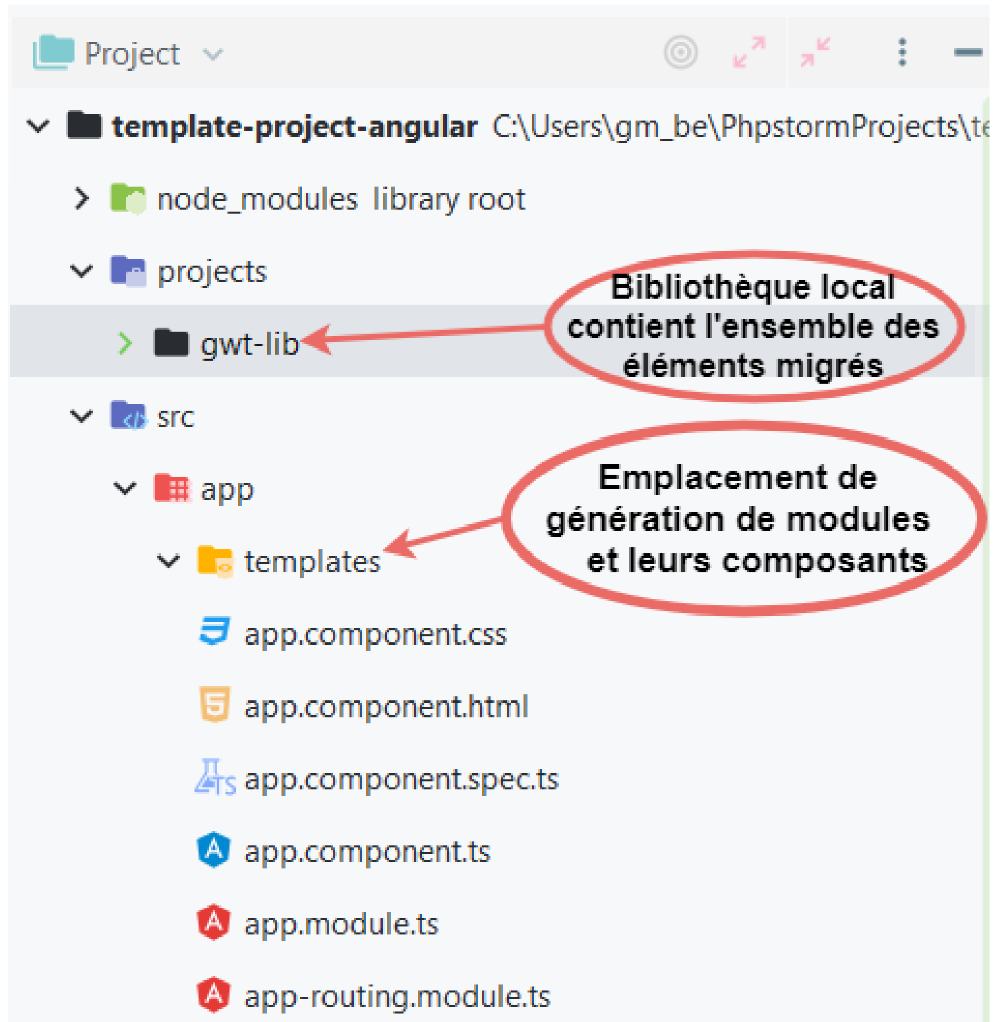


FIGURE 78 – Le Template de génération de l'application cible

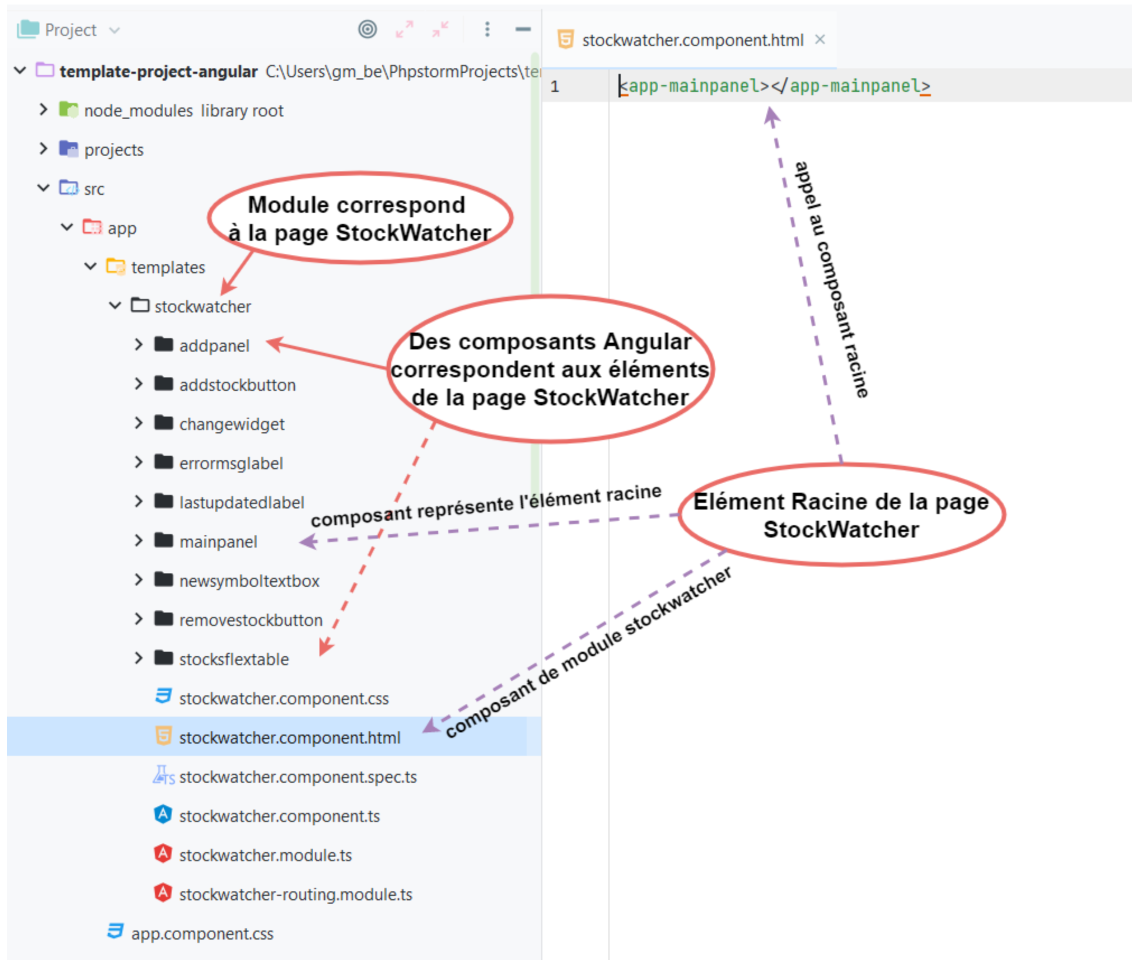


Figure 79 – Génération de l'application cible

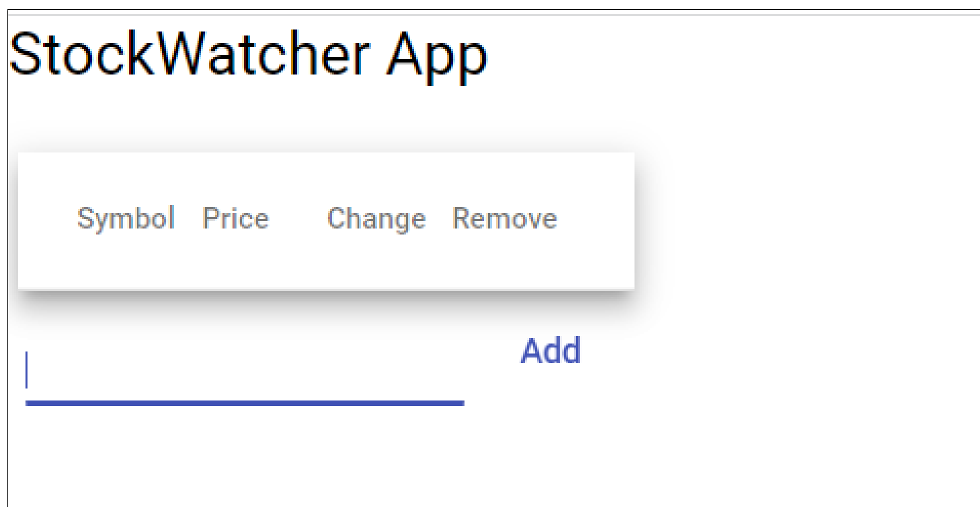


Figure 80 – Application générée en Angular

6 Conclusion

Ce chapitre synthétise les résultats de notre étude en mettant en relief certaines conclusions. Dans la première partie de ce chapitre, nous avons exposé les outils utilisés pour la mise en œuvre de la solution de migration de cas d'étude. Ensuite, nous avons listé un jeu de données afin d'expérimenter les heuristiques d'extraction (éléments graphiques, pages, racine, transition). Après, nous avons présenté les résultats des expérimentations avec le calcul des indicateurs de performances (précision, rappelle). Puis, nous avons interprété les résultats obtenus tout en mettant l'accent sur le menace de validité de l'approche proposée. Finalement, nous avons présenté un test de bout en bout afin de montrer toutes les étapes de migration d'une application GWT vers Angular.

CONCLUSION ET PERSPECTIVES

Conclusion générale

Le projet rapporté dans ce mémoire est né de la volonté de l'équipe MAREL du laboratoire d'informatique,de robotique et de micro électronique de Montpellier (LIRMM) dans le but de proposer une approche générique de migration de côté client des applications web en utilisant l'ingénierie dirigée par les modèles.

Pour ce faire, nous avons divisé le déroulement du projet en deux étapes. La première ayant pour but de l'exploration et documentation où nous avons fait un état de l'art sur l'évolution logicielles, l'ingénierie dirigée par les modèles, rétro-ingénierie dirigée par les modèles. Ensuite, nous avons étudié les approches existantes de migrations des interfaces graphiques afin de fixer notre objet d'étude.

La seconde étape consiste à établir une vision globale de notre approche de migration en utilisant les modèles. Puis, nous avons étudié un cas de migration des applications GWT vers Angular afin de voir la faisabilité de l'approche proposée. Dans cette étape, nous avons proposé des heuristiques d'extractions de concepts de GWT (éléments graphiques, pages, éléments racine de chaque page, transitions entre les pages). Nous n'avons traité que la partie visuelle des applications GWT, pour l'instant, à cause des défis confrontés. Ensuite, nous avons conçu une solution conceptuelle où nous avons cité l'utilisation du patron de conception afin de la rendre maintenable et évolutive au future. Après, nous l'avons réalisée en utilisant des outils, ce sont les deux phases qui ont pris le plus de temps. Nous avons par la suite présenté les tests d'expérimentation sur des heuristiques et un cas de test de bout-en-bout qui montre le déroulement de la migration.

Dans ce travail nous avons eu beaucoup de difficultés. Les problèmes majeurs que nous avons rencontrés sont les suivants :

- Migration des applications GWT abordant le style MVP et Place&Activities [20] car les deux types utilisent le bus d'événement et celui-là nécessite un temps considérable pour établir sa solution de migration.
- La sélection des projets GWT développées en Java exécutables (sans UiBinder) de tests pour vérifier le bon déroulement de notre migration.
- Chaque développeur a son style de développement donc il est difficile de traiter toutes les cas surtout dans localisation des variables (Slicing).
- Le test de migration à large échelle pour des raisons qui nous dépassent qui concernent le droit d'acquérir les applications et les ressources.
- La non-disponibilité des outils de validation de similarité visuelle entre les applications sources et cibles car la validation automatique de la migration est actuellement un problème non résolu. Il est possible de vérifier manuellement le résultat de la migration pour un nombre dénombrable de pages mais elle deviendra coûteuse à long terme.

20. Place&Activities : est un framework intégré inroduit par GWT pour la gestion de l'historique du navigateur. Il permet de permet de créer des URL pouvant être mises en signet dans une application GWT, permettant ainsi au bouton de retour et aux signets du navigateur de fonctionner comme les utilisateurs s'y attendent. Il s'appuie sur le mécanisme d'historique de GWT et peut être utilisé en conjonction avec le style MVP (GWT p. d.).

Perspectives

Comme tout travail scientifique, le nôtre peut être sujet à amélioration et enrichissement en vue des perspectives suivantes :

— Établir des solutions pour les verrous techniques confrontés pour compléter la migration d'une application GWT vers Angular.

— Inclure la migration du code comportemental et métier.

— Implémenter la méthode "headless application" pour séparer l'outil Modisco de l'environnement d'Eclipse pour que la migration soit automatique.

— Créer un outil capable de migrer automatiquement une application GWT vers Angular pour rendre la migration accessible.

— Effectuer une migration pour d'autres cas par exemple de GWT vers React, de JSF vers VueJS pour ressortir les menaces de validité de l'approche proposée.

Appréciation personnelle

Ce stage de fin d'études au sein du LIRMM a été une expérience très enrichissante sur le plan professionnel. Ce laboratoire m'a permis de rentrer dans le monde de la recherche et développement. Ayant acquis de nouvelles connaissances et compétences, plus particulièrement dans l'ingénierie dirigée par les modèles et l'évolution logiciels. je tiens compte que ce stage a concrètement valorisé ma formation d'ingénieur en Systèmes Informatiques et Logiciels (SIL) au sein de mon école, l'ESI.

BIBLIOGRAPHIE

ANTONIO, Bucchiarone et al. (2020). “Grand challenges in model-driven engineering : an analysis of the state of the research”. In : DOI : 10.1007/s10270-019-00773-6.

BENNETT, K. (2012). “Improving Legacy-System Sustainability : A Systematic Approach”. In : *IT Professional* 14, p. 38-43. ISSN : 1520-9202. DOI : 10.1109/MITP.2012.10.

BENNETT, Keith H, Magnus RAMAGE et Malcolm MUNRO (1999). “Decision model for legacy systems”. In : *IEE Proceedings-Software* 146.3, p. 153-159.

BEZIVIN, Jean (2005a). “On the unification power of models”. In : *Software and System Modeling (SoSym)*, 4(2) :171-188. (Cité pages 3).

— (2005b). “On the unification power of models”. In : *Software and Systems Modeling* 4.

BEZIVIN, Jean, Frédéric JOUAULT et Jean PALIÈS (2005). “Towards model transformation design patterns”. In : *ATLAS group, University of Nantes*.

BÉZIVIN, Jean (2004). “Sur les principes de base de l’ingénierie des modèles”. In : *RSTI-L’objet*, p. 145-157. DOI : 10.3166/objet.10.4.145-157. (Cité pages 3, 11, et 23.)

BÉZIVIN, Jean, Mireille BLAY et al. (2004). *Rapport de Synthèse de l’AS CNRS sur le MDA (Model Driven Architecture)*. Rapp. tech. University of Nantes.

BÉZIVIN, Jean, Erwan BRETON et al. (2003). *The ATL Transformation-based Model Management Framework*. Rapp. tech. University of Nantes.

BISBAL, J. et al. (1999). “Legacy Information System Migration : A Brief Review of Problems, Solutions and Research Issues”. In :

BRUNELIERE, Hugo et al. (sept. 2010). “MoDisco : A Generic And Extensible Framework For Model Driven Reverse Engineering”. In : *25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*. Anvers, Belgium, p. 173-174. DOI : 10.1145/1858996.1859032. URL : https://hal.archives-ouvertes.fr/hal-00534450.

BRUNELIÈRE, H. et al. (2014). “MoDisco : A model driven reverse engineering framework”. In : *Inf. Softw. Technol.* 56, p. 1012-1032.

BÜNDER, Hendrik (2019). “A model-driven approach for graphical user interface modernization reusing legacy services”. In : 30.

C, Seacord Robert, DANIEL et Lewis Grace A (2003). *Modernizing Legacy Systems : Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co. ISBN : 0321118847.

CANFORA, Gerardo et Massimiliano DI PENTA (2007). "New Frontiers of Reverse Engineering". In : *Future of Software Engineering (FOSE '07)*, p. 326-341. DOI : 10.1109/FOSE.2007.15.

CHIKOFSKY, E. et J. CROSS (1990). "Reverse engineering and design recovery : a taxonomy". In : *IEEE Software* 7, p. 13-17.

CHIKOFSKY, Elliot J. et James H CROSS (1990). "Reverse engineering and design recovery : A taxonomy". In : *IEEE software* 7.1, p. 13-17.

CHOMSKY, N. (1956). "Three models for the description of language". In : 2 (3). DOI : 10.1109/TIT.1956.1056813. (Cité page 1, Inroduction).

CHRISTA, Sharon et al. (2017). "Software Maintenance : From the Perspective of Effort and Cost Requirement". In : *Proceedings of the International Conference on Data Engineering and Communication Technology.* Sous la dir. de Suresh Chandra SATAPATHY, Vikrant BHATEJA et Amit JOSHI. Singapore : Springer Singapore, p. 759-768. ISBN : 978-981-10-1678-3.

CROTTY, James et Ivan HORROCKS (2017). "Managing legacy system costs : A case study of a meta-assessment model to identify solutions in a large financial services company". In : *Applied computing and informatics* 13.2, p. 175-183.

CZARNECKI, K. et S. HELSEN (2003). "Classification of Model Transformation Approaches". In :

DE LUCIA, Andrea, Anna Rita FASOLINO et E POMPELLE (2001). "A decisional framework for legacy system management". In : *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* IEEE, p. 642-651.

DEDEKE, Adenekan (1995). "Legacy systems : coping with success". In : *IEEE Software* 12, p. 19-23. DOI : 10.1109/52.363157.

DIAW, Samba, Rédouane LBATH et Bernard COULETTE (2010). "État de l'art sur le développement logiciel basé sur les transformations de modèles". In : DOI : 10.3166/tsi.29.505-536.

DOLQUES, Xavier (2010). "Génération de Transformations de Modèles : une approche basée sur les treillis de Galois". Thèse de doct. Université Montpellier II - Sciences et Techniques du Languedoc. URL : https://tel.archives-ouvertes.fr/tel-00916856.

DUCASSE, Stephane (août 2019). "Switching of GUI framework : the case from Spec to Spec 2". In : *International Workshop on Smalltalk Technologies.* Cologne, Germany.

DUCHIEN, Laurence et Cedric DUMOULIN, éd. (juin 2006). *Actes des 2èmes journées sur l'Ingénierie Dirigée.* Lille, France. URL : https://hal.inria.fr/hal-01136454.

EL BOUSSAIDI, Ghizlane (2016). "WAVI : A reverse engineering tool for web applications". In : *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, p. 1-3. DOI : 10.1109/ICPC.2016.7503744.

FONDEMENT, Frédéric et Raul SILAGHI (2004). *Defining Model Driven Engineering Processes.*

FOWLER, Martin (1999). *Refactoring : Improving the Design of Existing Code.* Boston, MA, USA : Addison-Wesley. ISBN : 0-201-48567-2.

— (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc.75 Arlington Street, Suite 300 Boston, MAUnited States. ISBN : 978-0-321-12742-6.

— (2010). *Domain Specific Languages.* 1st. Addison-Wesley Professional. ISBN : 0321712943.

GOSAIN, Anjana et Ganga SHARMA (2015). “Static Analysis : A Survey of Techniques and Tools”. In : *Intelligent Computing and Applications*. Sous la dir. de Durbadal MANDAL et al. New Delhi : Springer India, p. 581-591.

GROUP, Object management (2012). “Knowledge Discovery Metamodel (KDM)”. In : URL : `http://www.omg.org/technology/kdm/index.html`.

GROUP, Object Management (January 2006). *Meta Object Facility (MOF) 2.0 Core Specification*. Rapp. tech.

HAREL, David, Bernhard RUMPE et Technische Universität BRAUNSCHWEIG (2004). *Modeling Languages : Syntax, Semantics and all that Stuff (or, What's the Semantics of "Semantics" ?)* Rapp. tech.

HARMAN, Mark (2010). “Why Source Code Analysis and Manipulation Will Always be Important”. In : *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, p. 7-19. DOI : `10.1109/SCAM.2010.28`.

HAYAKAWA, Tomokazu (2014). “Maintaining Web Applications by Translating Among Different RIA Technologies”. In : *GSTF Journal on computing* 2.

HEIL, Sebastian (2018). “ReWaMP : Rapid Web Migration Prototyping Leveraging WebAssembly”. In : *ICWE*.

HELM, Erich Gamma ; Richard et Ralph JOHNSON (1994). “Desing Patterns. Elements of Reusable Object-Oriented Software”. In : *Addison-Wesley Professional Computing Series*. (Cite page 3).

HENRY, W. Li ; S. (1993). “Maintenance metrics for the object oriented paradigm”. In : *Proceedings First International Software Metrics Symposium*. DOI : `10.1109/METRIC.1993.263801`. (Cité page 2).

IEEE (1998). “IEEE Standard for Software Maintenance”. In : *IEEE Std 1219-1998*, p. 1-56. DOI : `10.1109/IEEESTD.1998.88278`.

JOUAULT, Frédéric et al. (2008). “ATL : A model transformation tool”. In : *Science of Computer Programming* 72.1. Special Issue on Second issue of experimental software and toolkits (EST), p. 31-39. ISSN : 0167-6423. DOI : `https://doi.org/10.1016/j.scico.2007.08.002`. URL : `https://www.sciencedirect.com/science/article/pii/S0167642308000439`.

JYLHÄ, Henrietta et Juho HAMARI (2020). “Development of measurement instrument for visual qualities of graphical user interface elements (VISQUAL) : a test in the context of mobile game icons”. In : *User Modeling and User-Adapted Interaction* 30.5. ISSN : 1573-1391. DOI : `10.1007/s11257-020-09263-7`. URL : `https://doi.org/10.1007/s11257-020-09263-7`.

KELLY, S (2018). “Modelling by the People, for the People”. In : *Seidl M., Zschaler S. (eds) Software Technologies : Applications and Foundation* 10748.

KICZALES, Gregor et al. (June 1997). “Aspect-Oriented Programming”. In : *M. Aksit and S. Matsuoka, editors, Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)* 1247 Of lecture Notes in Computre Science, p. 220-242. (Cité page 3).

KUSUMOTO, Shinji (2018). “unjQuerify : Migration of jQuery Snippets to Modern Vanilla JavaScript APIs”. In : *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, p. 618-622. DOI : `10.1109/APSEC.2018.00077`.

LEHMAN, Meir M et al. (1997). “Metrics and laws of software evolution-the nineties view”. In : *Proceedings Fourth International Software Metrics Symposium*. IEEE, p. 20-32.

MAREK, Lukáš et al. (2015). “Introduction to dynamic program analysis with DiSL”. In : *Science of Computer Programming* 98. Fifth issue of Experimental Software and

Toolkits (EST) : A special issue on Academics Modelling with Eclipse (ACME2012), p. 100-115. ISSN : 0167-6423. DOI : https://doi.org/10.1016/j.scico.2014.01.003.

MARTÍN SANTIBÁÑEZ, Daniel San, Rafael Serapilha DURELLI et Valter Vieira de CAMARGO (2015). “A combined approach for concern identification in KDM models”. In : *Journal of the Brazilian Computer Society* 21.1, p. 10. DOI : 10.1186/s13173-015-0030-3.

MORGADO, Ines Coimbra (2012). “Dynamic Reverse Engineering of Graphical User Interfaces”. In :

OMG (2003). *MDA guide version 1.0.1.* Rapp. tech. OMG document.

OMG (2011). “Architecture-Driven Modernization : Abstract Syntax Tree Metamodel (ASTM)”. In : URL : http://www.omg.org/spec/ASTM.

OUNI, Ali et al. (2017). “MORE : A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells”. In : *Journal of Software : Evolution and Process* 29.

PASCAL, André (2019). “Case Studies in Model-Driven Reverse Engineering”. In : *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development.* Setubal, PRT, p. 256-263. DOI : 10.5220/0007312502560263.

RAIBULET, Claudia, Francesca ARCELLI FONTANA et Marco ZANONI (2017). “Model-Driven Reverse Engineering Approaches : A Systematic Literature Review”. In : *IEEE Access* 5, p. 14516-14542. DOI : 10.1109/ACCESS.2017.2733518.

REIS, A. et A. SILVA (2017). “XIS-Reverse : A Model-driven Reverse Engineering Approach for Legacy Information Systems”. In : *MODELSWARD*.

SABIR, Umair et al. (2019). “A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code”. In : *IEEE Access* 7, p. 158931-158950.

SAMIR, Hani (2007). “Swing2Script : Migration of Java-Swing Applications to Ajax Web Applications”. In : *14th Working Conference on Reverse Engineering (WCRE 2007)*, p. 179-188. DOI : 10.1109/WCRE.2007.48.

SAMIR, Mbarki (2016). “Java Swing Modernization Approach - Complete Abstract Representation based on Static and Dynamic Analysis”. In : *ICSOFT-EA*.

SÁNCHEZ R, Ó (2014). “Model-driven reverse engineering of legacy graphical user interfaces”. In : *Automated Software Engineering* 21.2, p. 147-186. ISSN : 1573-7535. DOI : 10.1007/s10515-013-0130-2.

SANTOS, Bruno Marinho et al. (2019). “Towards a Reference Architecture for ADM-Based Modernization Tools”. In : *Proceedings of the XXXIII Brazilian Symposium on Software Engineering.* New York, NY, USA : Association for Computing Machinery, p. 114-123. ISBN : 9781450376518. DOI : 10.1145/3350768.3350792. URL : https://doi.org/10.1145/3350768.3350792.

SERIAI, Abdelhak et Sylvain CHARDIGNY (2010). “Software Architecture Recovery Process Based on Object-Oriented Source Code and Documentation”. In : *Software Architecture.* Berlin, Heidelberg : Springer Berlin Heidelberg, p. 409-416.

SERIAI, Abderrahmane (juil. 2019). “Migrating GWT to Angular 6 using MDE”. In : *SATToSE 2019 - 12th Seminar on Advanced Techniques Tools for Software Evolution.* Bolzano, Italy. URL : https://hal.inria.fr/hal-02304301.

SHAH, Eeshan (2011). “Reverse-Engineering User Interfaces to Facilitate porting to and across Mobile Devices and Platforms”. In : New York, NY, USA : Association for Computing Machinery. ISBN : 9781450311830. DOI : 10.1145/2095050.2095093.

AL-SHARA, Zakaria (2016). “Migrating Object Oriented Applications into Component-Based ones”. Thèse de doct. Université Montpellier.

SILVA, João Carlos (2010). “The GUISurfer Tool : Towards a Language Independent Approach to Reverse Engineering GUI Code”. In : New York, NY, USA : Association for Computing Machinery, p. 181-186. ISBN : 9781450300834. DOI : 10.1145/1822018.1822045.

TAMAI, Tetsuo et Yohsuke TORIMITSU (1992). “Software lifetime and its evolution process over generations.” In : *ICSM*. Citeseer, p. 63-69.

THILANKA, Kaushalya (2021). “Framework to Migrate AngularJS Based Legacy Web Application to React Component Architecture”. In :

TRIAS, Feliu (2015). “RE-CMS : A Reverse Engineering Toolkit for the Migration to CMS-Based Web Applications”. In : *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA : Association for Computing Machinery, p. 810-812. DOI : 10.1145/2695664.2696049.

TSUTANO, Yutaka et al. (2019). “Jitana : A modern hybrid program analysis framework for android platforms”. In : *Journal of Computer Languages* 52, p. 55-71. ISSN : 2590-1184. DOI : https://doi.org/10.1016/j.cola.2018.12.004.

WAGNER, Christian (2014). *Model-Driven Software Migration : A Methodology Reengineering, Recovery and Modernization of Legacy Systems*. Springer Vieweg. ISBN : 3658052694.

WINSKEL, Glynn (1993). *The formal semantics of programming languages : an introduction*. en. MIT Press.

YANG, Zhibin et al. (2021). “C2AADL$_Reverse : A model-driven reverse engineering approach to develop$ $critical software$”. In : *Journal of Systems Architecture* 118, p. 102202. ISSN : 1383-7621. DOI : https://doi.org/10.1016/j.sysarc.2021.102202.

WEBOGRAPHIE

Angular (2021a). *Architecture Angular*. URL : https://angular.io/guide/architecture.

— (2021b). *What is Angular*. URL : https://angular.io/guide/what-is-angular.

Christian, Pontesegger (2012). *Creating a headless application*. URL : http://codeandme.blogspot.com/2012/02/creating-headless-application.html.

Community, GWT (p. d.). *Ajax Communication : Introduction*. URL : http://www.gwtproject.org/doc/latest/tutorial/clientserver.html.

Dartlang, Community (p. d.). *The new Google AdSense user interface : built with AngularDart*. URL : https://news.dartlang.org/2016/10/google-adsense-angular-dart.html.

DOUDOUX., Jean-Michel (p. d.). *GWT (Google Web Toolkit)*. URL : https://www.jmdoudoux.fr/java/dej/chap-gwt.htm#gwt-1.

FireFox, Mozilla (p. d.). *Event reference*. URL : https://developer.mozilla.org/en-US/docs/Web/Events.

Gabriel, Barbier et Giquel Fabien (2021). *Java metamodel*. URL : https://help.eclipse.org/latest/index.jsp.

GWT (2020). *The GWT Release Notes*. URL : http://www.gwtproject.org/release-notes.html.

— (p. d.). *Compilation*. URL : http://www.gwtproject.org/doc/latest/tutorial/compile.html.

Jaber, Sami (2007). *La face cachée de GWT*. URL : https://docs.google.com/presentation/d/1vhe1sFDNovwLlhE8OnOyatMETVtvnJWTat7Xl87FuPk/edit?usp=sharing.

JDN (2012). *Structure d'un projet GWT*. URL : https://www.journaldunet.com/web-tech/developpeur/1105944-google-web-toolkit-creer-son-premier-projet/1105950-structure-d-un-projet-gwt.

Kapoor, Abhishek (2021). *Patterns to know before migrating your monolith to microservices*. URL : https://levelup.gitconnected.com/patterns-to-know-before-migrating-your-monolith-to-microservices-72fcbcc7846e.

MOKEDDEM, Hakim (2019). *Les outils d'analyse de qualité du code*. URL : https://drive.google.com/drive/folders/1VLAynzUxrJMChyL8e4C_Ijm_nRIAxoYQ.

Mostefai, A ; Batata S (2016). *Cours 6 – Architectures de Logiciels*. URL : https://drive.google.com/drive/folders/14VT1Aixr-Q0CeQH5uad7bQ0GUtnqsUUJ.

ORANGE (2021). *Recommandations pour les Single Page Applications*. URL : https://a11y-guidelines.orange.com/fr/articles/single-page-app/.

SHVETS, Alexander (p. d.). *PATRONS de CONCEPTION*. URL : https://refactoring.guru/fr/design-patterns/.

VALLÉE, Université Paris Est-Marne la (p. d.). *GWT : Google Web Toolkit*. URL : http://igm.univ-mlv.fr/~dr/XPOSE2008/GWT/architecture.html#1.

Annexes

ANNEXE A

MÉTA-MODÈLE JAVA

1 Principales entités du modèle Java

Le métamodèle Java est le reflet du langage Java, tel que défini dans la version 3 de « Java Language Specification »[1] (« JLS3 » correspond au JDK 5) (Gabriel et Fabien 2021). Nous découvrons dans la suite les principales entités du méta-modèle Java.

1.1 ASTNode

Chaque entité (à l'exception d'une entité nommée Model) hérite d'ASTNode. Comme son nom l'indique, ASTNode représente un nœud de graphe. ASTNode a une référence à la entité **Comment** car presque tous les éléments java peuvent être associés à un commentaire (bloc ou ligne de commentaire).

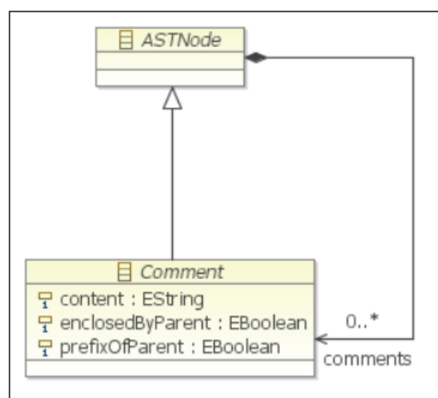


Figure 81 – (Gabriel et Fabien 2021)

1. Disponible sur : https://docs.oracle.com/javase/specs/jls/se6/html/j3TOC.html

1.2 Model, Package, AbstractTypeDeclaration

L'élément racine de chaque modèle Java est une instance de l'entité *Model*. Il s'agit d'une traduction du concept d'application Java, il contient donc des déclarations de packages (instances de l'entité Package). Et les déclarations de package contiennent des déclarations de type (instances compatibles avec l'entité AbstractTypeDeclaration).

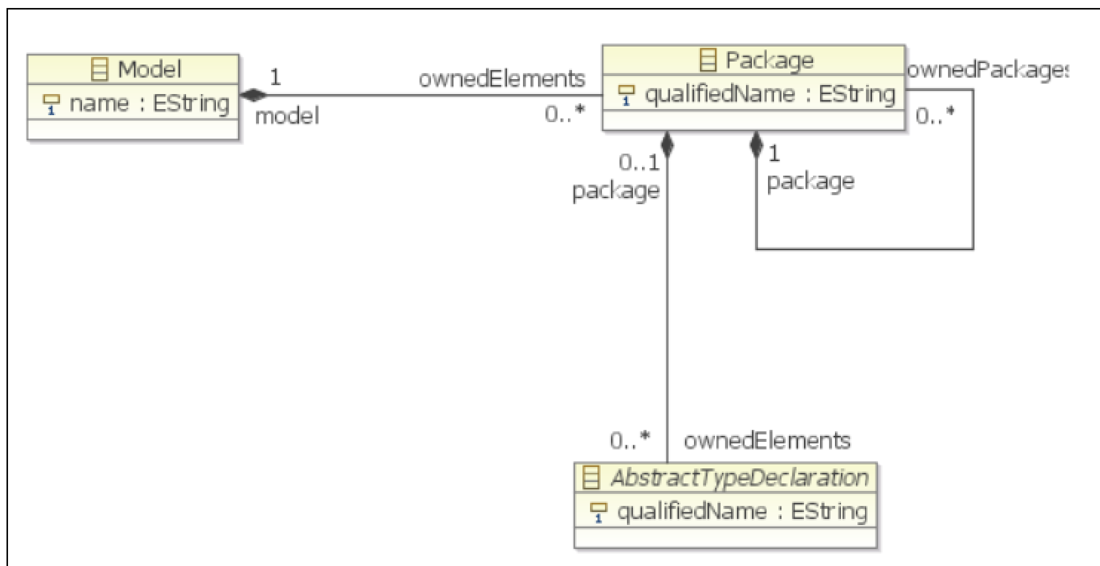


FIGURE 82 – Entité ASTNode (GABRIEL et FABIEN 2021)

1.3 NamedElement

Beaucoup d'éléments java sont nommés, et ce nom pourrait être considéré comme un identifiant de méthodes, packages, types, variables, champs, ... Donc toutes les entités correspondantes héritent de l'entité *NamedElement*.

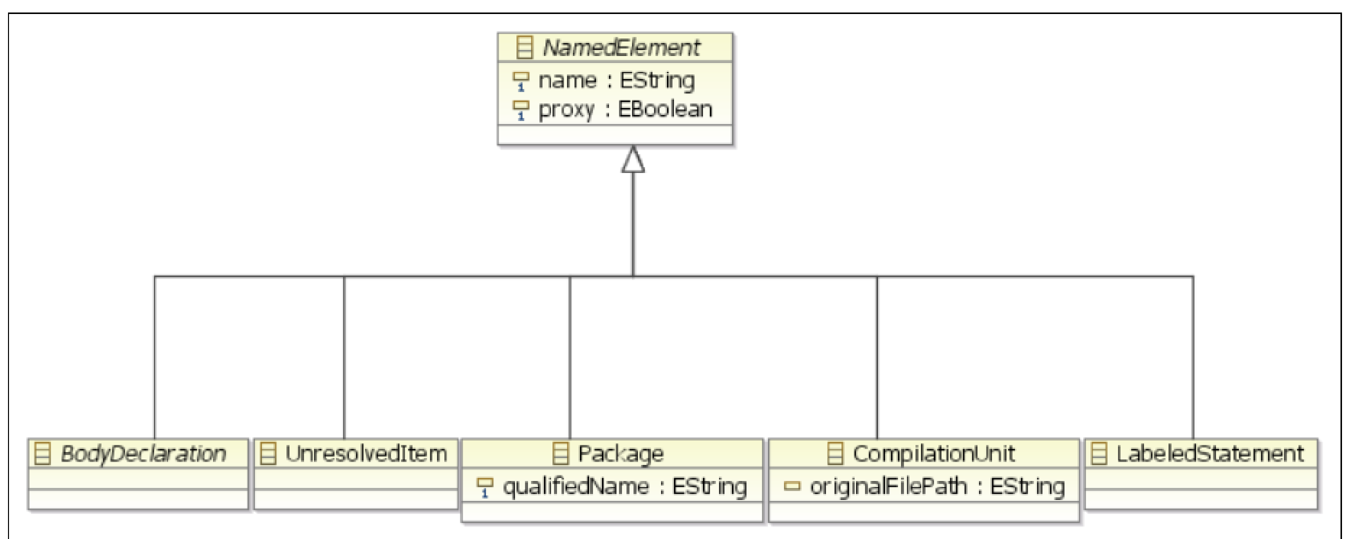


FIGURE 83 – Entité NamedElement (GABRIEL et FABIEN 2021)

1.4 Expressions

Comme dans beaucoup de langages, le concept d'expression existe en Java c'est une portion de code, sans déclarations, et l'évaluation retourne une valeur, numérique ou booléenne ou autre... Par exemple, ++i est une expression et ce sera traduit au concept de l'entité *PrefixExpression*. Tous les types d'expressions doivent hériter de l'entité Expression.

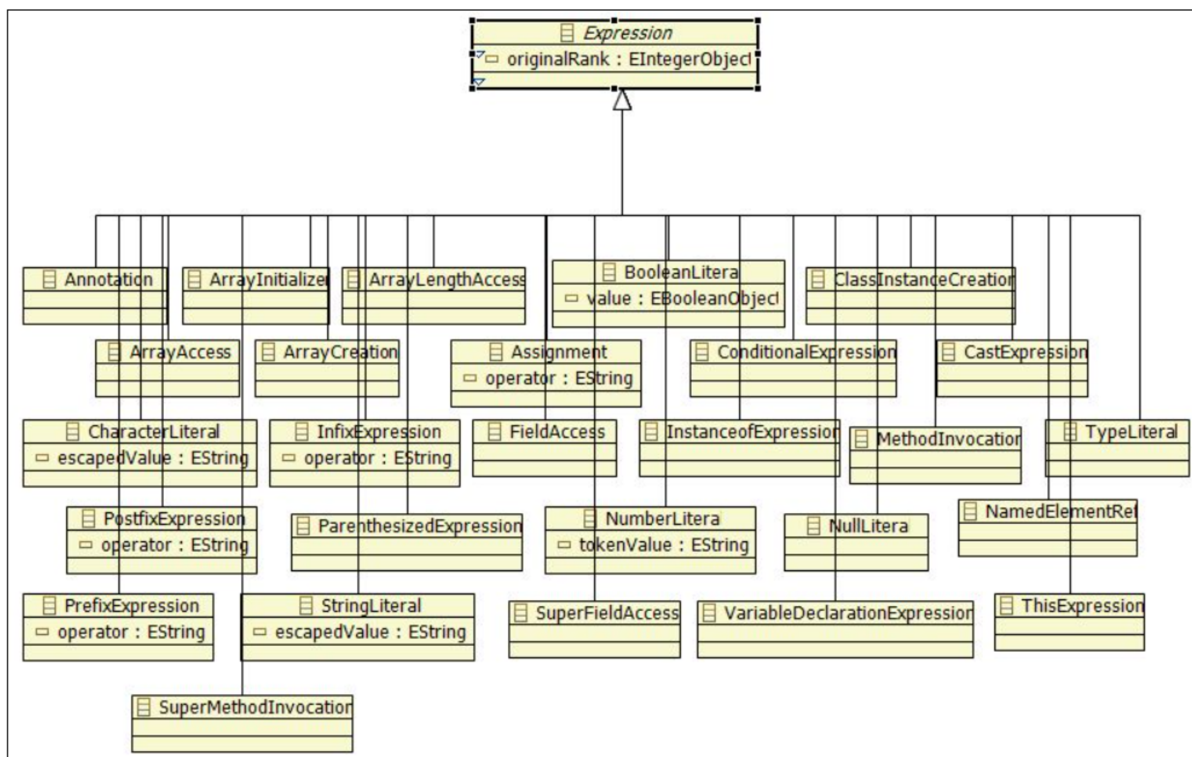


FIGURE 84 – Entité Expression et ses descendantes (GABRIEL et FABIEN 2021)

1.5 Statements

Une instruction en Java est représentée par l'entité Statement. Un bloc de code (Entité Block) contient une collection d'instructions et un bloc de code peut être contenu par une méthode. Quelques exemples d'instructions en java if, while, for, do, ... Toutes leurs définitions utilisent le concept d'expression pour séparer la valeur du mot-clé d'instruction.

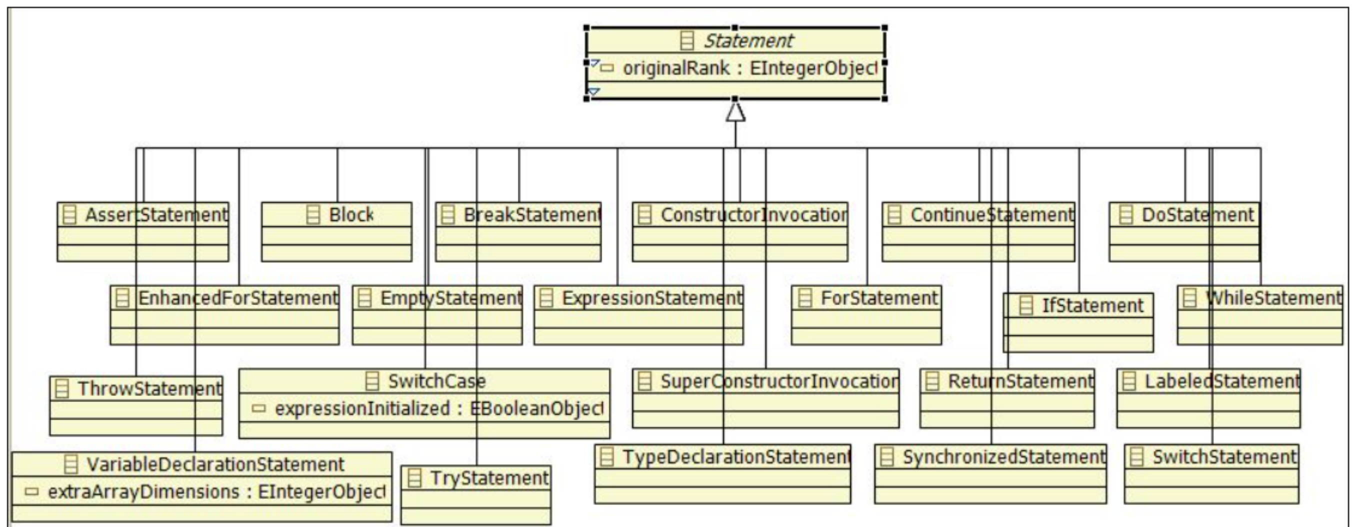


FIGURE 85 – Entité Statement et ses descendantes GABRIEL et FABIEN 2021