

Benoît Verhaeghe

Migration d'application GWT vers Angular



Tuteurs entreprise : M. Deruelle, M. Seriai

Tutrice-école : Mme Etien

Superviseurs Inria : Mme Etien, M. Anquetil

Berger-Levrault
Inria Lille Nord Europe - RMod
août 2018

Table des matières

1 Contexte	3
1.1 Contexte général	3
1.2 Problématique	3
1.3 Description du problème	3
2 Description du problème	5
2.1 Contraintes	5
2.2 Comparaison de GWT et Angular	6
3 État de l'art	7
3.1 Technique de migration	7
3.1.1 Rétro-Ingénierie d'interface graphique	7
3.1.2 Transformation de modèle vers modèle	9
3.1.3 Transformation de modèle vers texte	10
3.1.4 Migration de librairie	11
3.1.5 Migration de langage	11
3.2 Représentation d'interface graphique dans la littérature	12
3.2.1 Standard défini par l'OMG	12
3.2.2 Méta-modèle d'interface utilisateur	14
4 Décomposition d'une application avec GUI	16
4.1 Partie visuelle	16
4.2 Code comportemental	16
4.3 Code métier	16
5 Mise en place de la migration par les modèles	16
5.1 Approches pour effectuer la migration	17
5.2 Processus de migration	17
5.3 Méta-modèle d'interface utilisateur	18
5.4 Implémentation du processus	20
5.4.1 Importation	20
5.4.2 Exportation	22
6 Résultats	22
6.1 Résultat de l'importation	23
6.2 Visualisation	23
6.3 Exportation en Angular	24
7 Discussion	25
7.1 Gestion des layout	25
7.2 Gestion du comportement	26
8 Travaux futurs	26
8.1 Outil de validation	26
8.2 Complétude du travail	27
8.3 Exportation du Core	27
8.3.1 Méta-modèle de navigation et de state flow	28

8.4	Méta-modèle du code comportemental	29
9	Conclusion	29
	Bibliographie	30
10	Annexe	32

1 Contexte

1.1 Contexte général

Mon stage en entreprise est un travail qui s'inscrit dans le contexte d'une collaboration entre l'équipe RMod d'Inria Lille - Nord Europe et Berger-Levrault.

Berger-Levrault invente et développe des solutions pour les administrations et les collectivités locales, pour les établissements d'éducation et de santé publics comme privés, les universités et les entreprises. L'entreprise est implantée en France, au Canada et en Espagne.

J'ai travaillé dans l'équipe recherche et développement de Berger-Levrault à Montpellier. Mes superviseurs entreprises étaient M. Laurent Deruelle et M. Abderrahmane Seriai. Ma tutrice-école était Mme Anne Etien. J'ai, dans le cadre de la collaboration entre Berger-Levrault et l'Inria Lille - Nord Europe, travaillé aussi avec M. Nicolas Anquetil.

Ce travail est la suite du travail préliminaire que j'ai mené pendant mon Projet de Fin d'Études à Polytech Lille.

1.2 Problématique

Berger-Levrault possède des applications client/serveur qu'elle souhaite rajeunir. En particulier, le *front-end* est développé en GWT et doit migrer vers Angular 6. Le *back-end* est une application monolithique et doit évoluer vers une architecture de services web. Le changement de *framework*¹ graphique est imposé par l'arrêt du développement de GWT par Google. Le passage à une architecture à services est aussi souhaité pour améliorer l'offre commerciale et la rendre plus flexible.

Mon travail durant ce stage ne traite que de la migration des applications *front-end*.

1.3 Description du problème

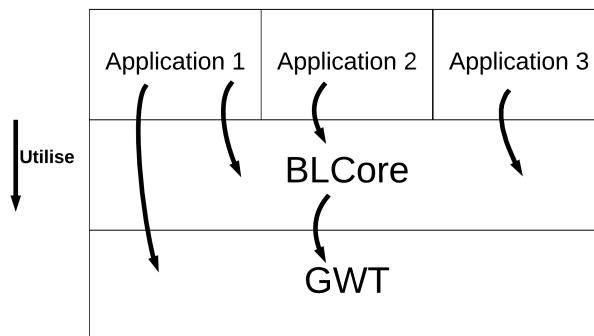


FIGURE 1 – Structure application

Présentation général appli BL

Les applications *front-end* de Berger-Levrault sont développées en Java en utilisant le *framework* GWT de Google. Ce sont les plus importantes applications GWT en termes de ligne de code et de classes dans le monde. Elles définissent plusieurs centaines de pages web.

Présentation du framework BLCore

1. *Framework* : ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel (architecture).

Dans l'optique d'homogénéiser le visuel de leurs applications, Berger-Levrault a étendu GWT. Cette extension, qui s'appelle BLCore, permet à Berger-Levrault de définir des composants qui seront communs à tous leurs développements.

Introduction utilisation du framework

La taille des applications ainsi que l'architecture des application de Berger-Levrault les rendent complexes La Figure 1 représente les différentes couches de *framework* utilisées par les applications de Berger-Levrault. Toutes les applications fonctionnent de manière indépendantes. Comme le représentent les flèches, les applications utilisent BLCore, qui lui-même utilise GWT. On retrouve aussi des applications, qui ne respectent pas la convention décidée par les équipes de Berger-Levrault, ayant un lien direct avec le *framework* GWT.

complexité de la migration

En plus de l'architecture des applications à étudier, la complexité de la migration des applications de Berger-Levrault réside dans leurs tailles, mais aussi et surtout dans le changement de langage d'implémentation. En effet, les applications actuelles et les *frameworks* BLCore et GWT sont développées intégralement en Java. Or, pour utiliser Angular 6, l'application doit être écrit en TypeScript. La question qui se pose est de savoir quelle couche, de la Figure 1, nous devons migrer et comment ?

compléxité lié à BL

En plus des difficultés techniques inhérentes à un tel projet, une entreprise comme Berger-Levrault a aussi des contraintes provenant de leurs développeurs et de leurs clients. Parmi ces contraintes, la migration ne devra pas perturber les clients de Berger-Levrault du point de vue visuel et comportemental.

proposition de solution

Bien qu'une migration complète de l'application en réécrivant l'ensemble du code soit possible, ce serait une tâche coûteuse et sujette à erreurs. Automatiser toute la migration semble donc être la bonne solution, cependant les développeurs ne seront pas formés sur la nouvelle technologie et sur son utilisation dans les nouvelles applications. Le manque de connaissance du langage cible va ralentir les développements et peut faire peur aux développeurs qui pourraient refuser une telle solution. Une alternative pour contourner ce problème serait de créer des outils facilitant la migration et de ne migrer qu'une partie de l'application. Les développeurs pourront alors effectuer la fin de la migration assistée par l'outil ce qui les formera sur le nouveau langage et sur l'application.

plan

Notre objectif est donc de trouver des solutions pour aider à la migration des applications de Berger-Levrault, de les évaluer et d'implémenter la meilleure solution respectant les contraintes de l'entreprise. Pour cela, nous avons dans un premier temps étudié les contraintes de Berger-Levrault présenté Section 2. Section 3, nous avons étudié les techniques utilisées dans la littérature pour représenter une interface graphique. Puis, Section 4, nous avons étudié les différentes composantes d'une application graphique. Ensuite nous avons crée une approche et mis en place un outil, présenté Section 5 pour expérimenter la migration d'une application de Berger-Levrault. Enfin, Section ?? et Section 8 nous avons discuté des résultats obtenus avec notre prototype et nous présentons les améliorations que nous devons encore apporter au travail effectué.

desc rapide bac à sable

J'ai effectué cette étude sur l'application *bac à sable* de Berger-Levrault. Cette dernière permet aux employés de Berger-Levrault de consulter les éléments disponibles dans BLCore. C'est une application complète qui fonctionne exactement comme une destinée aux clients de Berger-Levrault. En raison de son statut d'application modèle pour les développeurs, elle

contient tout ce qui est utilisable dans les autres applications. En plus des outils utilisable par les développeurs, l'application est aussi composé de code déviant des règles de programmation définit par Berger-Levrault.

Bien que plus petite que les applications en production, elle contient tout de même plus de 1 000 classes Java. Le *bac à sable* définit plus de 50 pages web qui peuvent être simples ou complexes (*c.-à-d.* qui comporte des formulaires, effectue des requêtes sur des serveurs distants, etc.).

explication travail pour comprendre

Nous avons aussi régulièrement vérifié que notre travail pouvait s'appliquer sur les projets plus importants de Berger-Levrault. Pour cela, nous avons utilisé notre prototype sur l'application *RH* de Berger-Levrault. L'application *RH* est destiné aux clients désirant centralisé la gestion du personnel de leurs compagnie dans une solution logiciel sur le web. Elle est constitué de plus 450 pages web et de 19 000 classes Java.

2 Description du problème

Dans le contexte de l'évolution des applications de Berger-Levrault, l'entreprise a estimé la transformation du code à 4000 jours-hommes de développement. Ceci s'explique majoritairement par les 1,67 MLOC² utilisés pour les logiciels. Un de mes objectifs à Berger-Levrault est de définir une stratégie de migration qui réduise le temps nécessaire pour la transformation des programmes.

2.1 Contraintes

Berger-Levrault étant une importante entreprise dans le domaine de l'édition de logiciel, elle a des contraintes spécifiques vis-à-vis d'un outil de migration. En effet, la solution logicielle doit respecter les contraintes suivantes :

- *Depuis GWT (BLCore) et Vers Angular.* Dans le cas d'une automatisation ou semi-automatisation du processus de migration, celui-ci doit pouvoir prendre du code GWT en entrée et s'achever par la génération de code Angular. La solution peut contenir des structures facilitant son utilisation pour d'autres langages cibles mais pas au détriment du projet fixé avec Berger-Levrault.
- *Approche modulaire.* Une approche divisée en petites étapes améliorera la maintenabilité de l'outil de migration [1]. Elle permettra de facilement remplacer une étape ou de l'étendre sans introduire d'instabilité. Le respect de cette contrainte offre plus de contrôle sur le processus de migration aux les entreprises. L'approche modulaire permet, entre autres, aux entreprises de modifier l'implémentation de la stratégie pour respecter leurs contraintes spécifiques.
- *Préservation de la structure.* Après la migration, nous devons retrouver la même structure entre les différents composants de l'interface graphique (*c.-à-d.* un bouton qui appartenait à un panel dans l'application source appartientra au panel correspondant dans l'application cible). Cette contrainte permet de faciliter le travail de compréhension de l'application générée par les développeurs. En effet, ils vont retrouver la même structure qu'ils avaient dans l'application source.
- *Préservation du visuel.* La migration doit pouvoir conserver le visuel aussi proche que possible. Cette contrainte est particulièrement importante pour les logiciels commer-

2. MLOC : *Million Lines of Code*

ciaux. En effet, les utilisateurs de l'application ne doivent pas être perturbés par la migration. Il est aussi possible que Berger-Levrault est envie de profiter de la migration pour rafraîchir le visuel de leurs applications. Dans ce cas le l'outil peut proposer de faciliter certains points de cette transformation graphique.

- *Automatique*. Une solution automatique facilite l'accessibilité de l'outil. Pour simplifier le processus de migration, il serait bien que les utilisateurs de l'outil n'aient pas à intervenir pendant le processus de migration ou très peu. Ainsi, l'outil peut être utilisé avec un minimum de connaissance préalable. Dans le cas où le prototype n'a pas besoin de l'intervention humaine pendant le processus de migration, il sera plus facile à utiliser sur des grands systèmes [2].
- *Amélioration de la qualité*. La migration doit permettre de traiter le maximum de déviance du programme source possibles. Il est possible que certaines déviations ne soient pas traitables ou demandent un effort trop important, elles seront alors traitées post-migration. Dans le cas de Berger-Levrault, l'outil de migration peut gérer les éléments, utilisés par l'application à migrer, provenant du *framework* GWT. Cet exemple d'utilisation du *framework* GWT par l'Application 1 est représenté Figure 1.
- *Continuation du service*. Pendant la conception de la stratégie de migration, le développement du prototype permettant la migration et la migration elle-même, les équipes de développement doivent pouvoir continuer la maintenance des applications. Cette contrainte est essentielle puisque Berger-Levrault, en raison de son activité, ne peut pas demander à ses clients d'accepter un arrêt des améliorations et correction de bug pendant plusieurs mois.
- *Lisibilité*. Afin de faciliter le travail de compréhension du code migré, il serait bien que la migration produise une application respectant les normes définies par les développeurs. Dans le cas de Berger-Levrault, il s'agit du respect du nommage des variables en CamelCase³ et l'utilisation de noms significatifs.

2.2 Comparaison de GWT et Angular

	GWT	Angular
Page web	Une classe Java	Un fichier TypeScript et un fichier HTML
Style pour une page web	Inclus dans le fichier Java	Un fichier CSS optionnel
Nombre de fichiers de configuration	Un fichier de configuration	Quatre fichiers plus deux par sous-projets

Tableau 1 – Comparaison des architectures de GWT et Angular

Dans le cas de ce projet, les langages de programmation source et cible ont deux architectures différentes. Les différences sont syntaxiques, sémantiques et architecturales. Pour la migration d'application GWT vers Angular, les fichiers *.java* seront décomposés en plusieurs fichiers Angular.

Le Tableau 1 synthétise les différences entre l'architecture d'une application en Java et celle en Angular. Les différences se font pour trois notions, les pages web, leurs styles et les fichiers de configuration.

Avec le *framework* GWT, un seul fichier Java est nécessaire pour représenter une page web. L'ensemble de la page web peut donc être contenu dans ce fichier, même s'il reste possible de décomposer les différents éléments de la page web en plusieurs classes. Les fichiers Java

3. *Camel Case* : les mots sont liés sans espace. Chaque mot commence par une Majuscule.

contiennent les différents composants graphiques (widgets) de la page web, leurs positions les uns par rapport aux autres et leurs organisations hiérarchiques. Dans le cas d'un widget sur lequel une action peut être exécutée (comme un bouton), c'est dans ce même fichier qu'est contenu le code à exécuter lorsque l'action est réalisée.

En Angular, on crée une hiérarchie de fichier correspondant à un sous-projet pour chaque page web. Cette hiérarchie permet de séparer le visuel d'une page web, des scripts qu'il utilise. Elle contient plusieurs fichiers dont un fichier HTML qui contient les widgets de la page web et leurs organisations, et un fichier TypeScript contenant le code à exécuter quand une action se produit sur un widget. On a donc la décomposition d'un fichier Java pour GWT en deux fichiers HTML et TypeScript en Angular pour représenter les widgets et le code qui leur est associés.

Pour le style visuel d'une page web, dans le cas de GWT, il y a un fichier CSS commun à toutes les pages web et des modifications qui sont appliquées directement dans le fichier Java de la page web. Ces modifications peuvent porter sur la couleur ou les dimensions.

En Angular, on retrouve le même fichier CSS général pour tout le projet. Il est aussi possible de définir un fichier CSS pour chaque sous-projet (page web) qui va définir le visuel des éléments de la page web. Il y a donc création d'un fichier supplémentaire en Angular par rapport à GWT.

Pour les fichiers de configurations, GWT n'a besoin que d'un fichier de configuration qui définit les fichiers java correspondant à une page web et les URL que l'on doit utiliser pour y accéder. En Angular, il y a deux fichiers de configuration générale. Le premier, *module*, explicite les différentes pages web accessibles dans l'application ainsi que les services distants et les composants graphiques utilisables dans l'application. Le second, de *routing*, définit pour les différentes pages web de l'application leurs chemins d'accès.

3 État de l'art

Dans le cadre de la conception de l'outil de migration, nous avons étudié la littérature pour identifier les techniques respectant les critères définis par Berger-Levrault. Dans un premier temps, la Section 3.1 présente les différentes techniques qui sont utilisées pour faire effectuer la migration d'application. Dans un second temps, la Section 3.2 présentera les différentes représentations d'interface graphique que les auteurs de la littérature ont utilisé.

3.1 Technique de migration

Nous avons extrait de la littérature plusieurs domaines de recherche connexes au travail que nous voulons mener sur la migration d'application. Les approches proposées permettent soit d'effectuer la migration d'application sans répondre aux critères que nous avons définis avec Berger-Levrault, soit de proposer des pistes à explorer pour créer une solution.

3.1.1 Rétro-Ingénierie d'interface graphique

La rétro-ingénierie discute de comment représenter une interface graphique dans l'objectif de pouvoir l'analyser et comment générer cette représentation.

Les auteurs utilisent trois techniques pour créer ces représentations : statique, dynamique ou hybride.

Statique. La stratégie statique consiste à analyser du code source et à en extraire de l'information. La stratégie statique n'exécute pas le code de l'application à analyser.

Dans le cas de Cloutier *et al.* [3], les auteurs ont analysé directement les fichiers HTML, CSS et JavaScript. Ceci leurs permet de construire un arbre syntaxique du code source du site web et d'extraire les différents widgets depuis le fichier HTML. Le travail des auteurs ne permet pas de capturer le comportement des éléments graphique. De plus, le travail consiste surtout en la recherche de lien entre les différents composants du programme (classes JavaScript, tag HTML, etc.) et non pas la représentation d'une interface graphique.

Silva *et al.* [4], Lelli *et al.* [5] et Staiger *et al.* [6] utilisent des outils qui analysent du code source provenant de langage non destiné au web, mais permettant de décrire une interface graphique. Leurs cas d'études sont des applications de bureau ayant une interface graphique. Ces logiciels cherchent la définition des widgets dans le code source. Une fois les créations des widgets trouvées dans le code source, les logiciels analysent les méthodes invoquées ou invoquant les widgets afin de découvrir les relations entre les widgets et leurs attributs. Bien que le travail des auteurs semble intéressant dans le cadre du projet de migration que nous menons, les auteurs ne proposent pas de solution sur la manière d'évaluer le résultat de l'outil de rétro-ingénierie sur des applications composées de beaucoup d'écran.

Sánchez Ramón *et al.* [1] ont développé une solution permettant d'extraire depuis un ancien logiciel son interface graphique. Le cas d'étude des auteurs est une application source créée avec Oracle Forms. Contrairement aux cas d'études précédents, l'interface est définie dans un fichier à part explicitant pour chaque widget sa position fixe. Chaque widget a ainsi une position X, Y ainsi qu'une hauteur et une largeur. La stratégie des auteurs consiste à déterminer la hiérarchie des widgets depuis ces informations. Cependant, Oracle Forms est utilisée pour créer une interface simple avec seulement des champs texte ou des formulaires. Les champs texte contiennent des données provenant d'une base de données. La disposition des éléments est aussi très simple dans l'exemple fourni par les auteurs car les champs texte ou les formulaires sont affichés les uns en dessous des autres. Dans notre cas, les interfaces sont plus complexe et demande une analyse plus poussé que la recherche de la position des widgets.

La stratégie statique permet d'effectuer l'analyse d'une application sans avoir besoin de l'exécuter. Cependant elle a des lacunes sur l'analyse des structures de contrôles comme les boucles si, si elles contiennent des informations à rechercher dans le code. Dans le cas des interfaces graphiques, il est possible qu'une partie du code soit hébergé par un serveur distant, dans ce cas, l'analyse statique n'a pas accès aux résultats des appels au serveur distant (surtout si ceux-ci dépendent de l'appel originel), ce qui amène à un manque d'information.

Dynamique. La stratégie dynamique consiste à analyser l'interface graphique d'une application pendant qu'elle est en fonctionnement. L'utilisateur lance un logiciel dont l'interface est à extraire, puis il lance l'outil d'analyse dynamique. Ce dernier est capable de détecter les différents composants de l'interface et les actions qu'il peut leur appliquer. Puis l'outil applique une action sur un élément de l'interface et détecter les changements s'il y en a. En répétant ces actions, l'approche permet d'explorer les différentes interfaces qui sont utilisées dans l'application à analyser et comment interagir avec ces dernières.

Les auteurs Memon *et al.* [7], Samir *et al.* [8], Shah et Tilevich [9] et Morgado *et al.* [10] ont développé des logiciels implémentant la stratégie dynamique. Cependant, toutes les solutions proposées s'appliquent sur des applications exécutées sur un ordinateur (application de bureau). Une adaptation serait nécessaire pour coller aux spécificités des applications web comme dans notre cas.

L'analyse dynamique permet de parcourir toutes les fenêtres d'une application et d'obtenir des informations provenant de code exécuté. Cependant, l'analyse impact les performances de l'application à analyser. Dans le cas de petites applications ou d'application statique,

l'impact n'est pas bloquant ou peut être fait en interne. Mais dans le cadre des applications de Berger-Levrault, l'utilisation de la stratégie dynamique ne peut pas être envisagé car l'impact sur les performances pose un problème aux clients de l'entreprise.

Hybride. L'objectif de la stratégie hybride est d'utiliser la stratégie statique et la stratégie dynamique. Gotti *et al.* [11] utilisent une stratégie hybride pour l'analyse d'applications écrites en Java. La première étape consiste en la création d'un modèle grâce à une analyse statique du code source. Les auteurs retrouvent la composition des interfaces graphiques, les différents widgets et leurs propriétés. Ensuite, l'analyse dynamique exécute les différentes actions possibles sur tous les widgets et analyse les modifications potentielles sur l'interface après avoir effectué les actions.

Cette stratégie semble permet de collecter un maximum d'information en combinant les avantages des stratégies statique et dynamique. Bien que la stratégie dynamique permet de réduire les problèmes d'analyse de la stratégie statique. Les contraintes inhérentes à la stratégie dynamique restent présent et important dans le cadre de notre projet.

3.1.2 Transformation de modèle vers modèle

La *transformation de modèle vers modèle* traite de la modification d'un modèle source vers un modèle cible. Dans le cadre d'une migration, si nous décidons d'utiliser des modèles pour effectuer des transformation, la transformation de modèle est une étape essentielle du processus.

L'article de Baki *et al.* [12] présente un processus de migration d'un modèle UML vers un modèle SQL. Pour faire la migration, les auteurs ont décidé d'utiliser des règles de transformation. Ces règles prennent en entrée le modèle UML et donne en sortit le SQL définis par les règles. Plutôt que d'écrire les règles de migration à la main. Les auteurs ont décomposé ces règles en petites briques. Chaque brique peut correspondre soit à une condition à respecter pour que la règle soit validée, soit à un changement sur la sortie de la règle. Ensuite, les auteurs ont développé un algorithme de programmation génétique pouvant manipuler ces règles. À partir d'exemples l'algorithme apprend les règles de transformation à appliquer afin d'effectuer la transformation du modèle. Pour cela, il modifie les petites briques composant les règles et analyse si le modèle en sortie ressemble à celui explicité pour tous les exemples. Enfin, l'algorithme est appliqué sur de vraies données. Nous avons décidé de ne pas expérimenter cette stratégie de migration. En effet, dans le cas d'étude des auteurs, il préexiste un certain nombre d'exemples qui favorise l'apprentissage de l'algorithme de machine learning. Nous ne possédons pas dans notre projet d'exemples et le nombre de déviations dans le code peut vite poser un souci à la formation de l'intelligence artificielle.

du coup oui, le méta-méta-modèle c'est comme fame

Wang *et al.* [13] ont créé une méthodologie et un outil permettant de faire automatiquement la transformation d'un modèle vers un autre modèle. Leur outil se distingue en effectuant une migration qui se base sur une analyse syntaxique et sémantique. L'objectif de la méthodologie est d'effectuer la transformation d'un modèle vers un autre de manière itérative en modifiant le méta-modèle. Le processus de transformation des auteurs est divisé en quatres étapes.

1. Crédit de règles de mise en correspondance grâce à une recherche sémantique et syntaxique sur les éléments en entrée du processus et ceux désiré en fin de processus.
2. Génération du nouveau modèle grâce aux règles découvertes
3. Evaluation des règles
4. Crédit des règles au niveau du méta-modèle et génération du nouveau méta-modèle.

Une condition d'utilisation contraignante décrite par les auteurs est la nécessité d'avoir un méta-méta-modèle pour tous les méta-modèles intermédiaires. Le méta-méta-modèle des auteurs définit seulement 9 éléments principaux qui sont modèle, élément, noeud, relation entre deux noeuds (*edge*), propriété, primitif, énumération, relation sémantique et relation syntaxique. Les auteurs ont implémenté ce méta-méta-modèle dans leur outil. Cette solution permet de définir des règles de transformation et des méta-modèles pour effectuer la transformation de modèle. Cependant, dans notre cas cela demande un grand nombre d'exemples de code source et de code déjà migré.

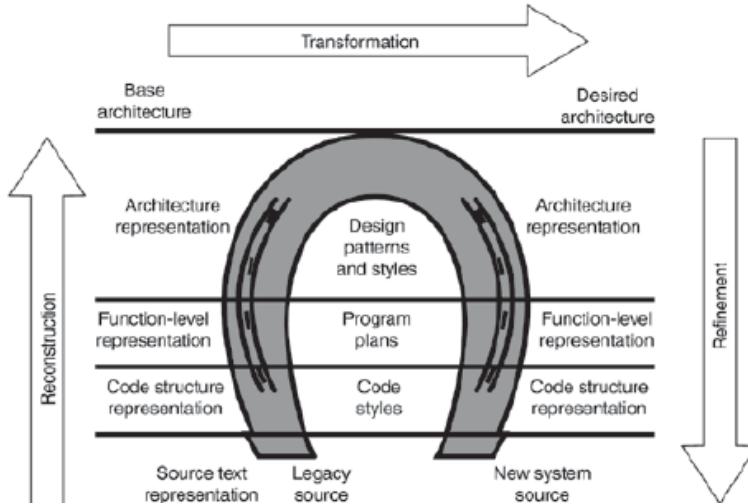


FIGURE 2 – Migration selon le *fer à cheval*

Fleurey *et al.* [14] et Garcés *et al.* [15] ont travaillé sur la modernisation et la migration de logiciel. Ils ont développé des logiciels permettant de semi-automatiser la migration d'applications. Pour cela, ils ont suivi le principe du "*fer à cheval*" présenté Figure 2 proposé par Zang *et al.*[16]. La migration se passe en quatre étapes.

1. Ils génèrent un modèle de l'application à migrer.
2. Ils transforment ce modèle en un "modèle pivot". Ce dernier contient les structures des données, les actions et algorithmes, l'interface graphique et la navigation dans l'application.
3. Ils transforment le modèle pivot en un modèle respectant le méta-modèle du langage cible.
4. Ils génèrent le code source final de l'application.

Comme les auteurs, nous devons conserver structures de données, actions, interface graphique et navigation dans l'application. Ce sont des éléments que nous avons représentés dans les méta-modèles que nous avons conçus. De plus, nous suivons le principe du "*fer à cheval*".

3.1.3 Transformation de modèle vers texte

La *transformation de modèle vers texte* traite du passage d'un modèle source vers du texte. Le texte peut être du code source compilable ou non.

L'article de Mukherjee *et al.* [17] présente un outil permettant de prendre en entrée les spécifications d'un programme et donne en sortie un programme utilisable. L'entrée est un fichier en XML et la sortie est un programme écrit en C ou en Java (en fonction du choix de

l'utilisateur). Pour effectuer les transformations, les auteurs ont utilisé un système de règles de transformation. Ce travail est lié à notre problématique d'exportation des méta-modèles. En effet, le fichier XML pris en entrée de l'outil des développeurs peut être assimilé à un modèle suivant un méta-modèle. Dans ce cas, la génération de code depuis un fichier XML n'est pas très différente de celui de la génération de code depuis des modèles.

3.1.4 Migration de librairie

La *migration de librairie* présente des solutions sur comment changer de *framework*. Ce travail est lié à notre problématique puisque, pour la migration des applications de Berger-Levrault, nous passons de l'utilisation du *framework* GWT à l'utilisation de frameworks associé à Angular.

Chen *et al.* [18] ont développé un outil permettant de trouver des librairies similaires à une autre. Pour cela, les auteurs ont miné les tags des questions de Stack Overflow. Avec ces informations, ils ont pu mettre en relation des librairies avec d'autres librairies quelque soit le langage d'implémentation de ces derniers. Pour la migration de Java/GWT vers Angular, nous avons besoin de changer de librairie. Plutôt que de réécrire la librairie, la recherche d'une autre librairie permettant de résoudre les mêmes problèmes peut être une solution. C'est dans ce contexte que le travail des auteurs peut guider notre recherche de librairie en faisant correspondre les anciennes librairies utilisées par les applications de Berger-Levrault avec d'autres compatibles avec Angular.

La *migration de librairie* cherche des équivalences entre des librairies, pour cela les outils doivent analyser les librairies. Bien que cela puisse nous aider, la recherche de correspondance entre des librairies n'est pas suffisant pour effectuer la migration, la recherche de correspondance peut ne pas être assez précise. De plus, ce travail ne prend pas en compte les autres détails inhérents à l'application originel.

3.1.5 Migration de langage

La *migration de langage* traite de la transformation du code source directement (*c.-à-d.* sans passer par un modèle). Pour cela, les auteurs travaillent sur la création de règles permettant de modifier le code source.

Brant *et al.* [19] ont développé un outil de définition de règle de transformation. Ainsi, les auteurs sont parvenus à migrer une application Delphi de 1,5 million de lignes de code en C#. Comme les auteurs, nous voulons effectuer la migration du code source d'une application. Notre cas se différencie par les langages source et cible. Cette solution permettrait de faire la transcription du code GWT vers du code Angular. Cependant, les auteurs n'ont pas appliqué leurs outils sur un logiciel qui comporte une interface graphique. Nous ne savons donc pas si la solution est applicable en totalité.

Un des problèmes de la migration du code source est la définition des règles. Newman *et al.* [20] ont proposé un outil facilitant la création de règles de transformation. Pour cela, l'outil "normalise" le code source en entrée et essaie de le simplifier. Ainsi, les auteurs arrivent à réduire le nombre de règles de transformations à écrire et leurs complexités. Dans le cas de migration de Berger-Levrault, nous devons gérer les multiples manières dont les fonctionnalités sont écrites. La normalisation du code source peut simplifier l'écriture des règles de transformation ou les règles permettant de créer les méta-modèles.

Rolim *et al.* [21] ont créé un outil qui apprend des règles de transformation de programme

à partir d'exemples. Pour cela, les auteurs ont défini un DSL⁴ permettant d'exprimer les modifications faites sur l'AST⁵ d'un programme. Ensuite, à partir d'une base d'exemple de transformation, l'outil recherche les règles de transformation entre les fichiers d'entrée des exemples et ceux de sorties. Une fois les règles trouvées et écrites dans le DSL prédéfini, l'outil prend en entrée un bout de code et donne en sortie le résultat des transformations. Ce travail peut nous servir pour la migration des applications de Berger-Levrault. En effet, nous pouvons imaginer faire la migration de tout ou partie des applications en utilisant un tel outil. Par exemple, une fois la migration semi-automatique effectuée par la solution que nous avons proposée, ce type d'outil pourrait servir de "guide" pour les nouveaux développeurs participants à la migration ou au développement courant des applications.

3.2 Représentation d'interface graphique dans la littérature

Nous avons vu dans la Section précédente que la représentation abstraite des interfaces graphiques est souvent utilisé. Nous avons donc recherché et comparé les différentes représentations existantes.

Section 3.2.1, nous présentons les deux ensemble de méta-modèles défini par l'OMG⁶ pour représenter une application. Le premier KDM permet de représenter une application de manière générale tandis que le second, IFML, est spécialisé dans la représentation d'application ayant une interface graphique. La Section 3.2.2 détaille les représentations des interfaces graphiques décrites dans la littérature.

3.2.1 Standard défini par l'OMG

L'OMG a défini la norme KDM⁷ (<https://www.omg.org/spec/KDM/>) pour prendre en charge l'évolution des logiciels. Le standard utilise un méta-modèle pour représenter un logiciel dans un haut niveau d'abstraction.

L'architecture KDM est divisée en douze paquetages organisés en quatre couches. Le paquetage d'interface utilisateur est composé d'un ensemble de méta-modèles pour représenter les composants et le comportement de l'interface graphique. Le paquetage Action définit des méta-modèles pour représenter le comportement d'une application. Il peut être utilisé pour décrire la logique de l'application, par exemple conditions, boucle, appel. Le paquetage Data représente les données utilisées dans l'application pouvant provenir de services distants ou de bases de données. Il définit également un méta-modèle représentant les actions que les bases de données peuvent utiliser, par exemple, insertion, mise à jour, suppression. Ces actions peuvent être exécutées automatiquement lorsqu'un événement se produit.

La Figure 3 est le diagramme de classes UIResources. Il définit de nombreuses entités pour l'abstraction de l'interface utilisateur.

UIResource peut être défini comme UIDisplay, UIField ou UIEvent. Un UIField est utilisé pour représenter des formulaires, des champs de texte, des panels, etc. Nous pouvons également détecter le patron de conception *composite* avec UIResource et AbstractUIElement. Cela permet la représentation de l'architecture d'interface utilisateur possible.

4. DSL : Domain Specific Language est un langage de programmation destiné à générer des programmes dans un domaine spécifique.

5. AST : Arbre Syntaxique Abstrait

6. OMG : Object Management Group

7. KDM : Knowledge Discovery Metamodel



FIGURE 3 – KDM Diagramme de Classe UIResources

UIDisplay peut être un **Screen** ou un **Report**. **Screen** représente une fenêtre d'un logiciel de bureau ou une page web. **Report** représente un document qui sera imprimé.

Chaque **AbstractUIElement** peut avoir une **UIAction**. Une **UIAction** peut effectuer plusieurs **UIEvent**. **UIEvent** représente les événements qui impactent l'interface utilisateur. Il peut s'agir d'un *Callback* ou de l'idée de navigation (*c.-à-d.* passer d'une page web à une autre).

Le langage IFML proposé par Brambilla *et al.* [22] et adopté par l'OMG permet de représenter une l'interface graphique d'une application. Les méta-modèles ont été définis avec l'OMG (<http://www.ifml.org/>). L'IFML a pour but de fournir un système pour décrire la partie visuelle d'une application, avec les composants et les conteneurs, l'état des composants, la logique de l'application et la liaison entre les données et l'interface utilisateur.

Avec IFML, un logiciel peut être modélisé avec un ou plusieurs conteneurs racines. Un conteneur représente une fenêtre principale pour une application de bureau ou une page web pour une application web. Ensuite, chaque conteneur a des sous-conteneurs ou peut contenir des composants.

Un composant est le niveau abstrait d'un widget visible. Cet élément peut avoir des paramètres d'entrée ou de sortie. Un paramètre d'entrée décrite comme une donnée signifie que le widget affiche la valeur de la donnée.

Les conteneurs et les composants peuvent être associés à des événements. Un élément lié à un événement prend en charge l'interaction des utilisateurs, par exemple, cliquer, glisser-déposer, *etc.* Une fois l'action effectuée, l'effet est représenté par une connexion de flux d'interaction qui connecte l'événement et les éléments affectés par l'événement.

La Figure 4 présente le méta-modèle *View Elements* proposé par IFML. Ce méta-modèle a pour objectif de représenter la partie visible de l'interface utilisateur. Il utilise le patron de conception *composite* et ont donc la notion de conteneur et de composant afin de représenter le DOM. Le méta-modèle IFML introduit aussi la notion de *ComponentPart*. Cette entité est nécessaire pour représenter tous les composants dans le méta-modèle IFML car les auteurs ont décidé de définir les éléments tels que les listes ou les formulaires en tant que composant. Ils ne peuvent donc pas contenir d'autres éléments, ce qui reviendrait à n'avoir que des listes, tableaux et formulaires vides. L'utilisation d'un *ComponentPart* permet d'ajouter d'autre composant aux listes et formulaire sans pour autant les considérer comme présent dans le

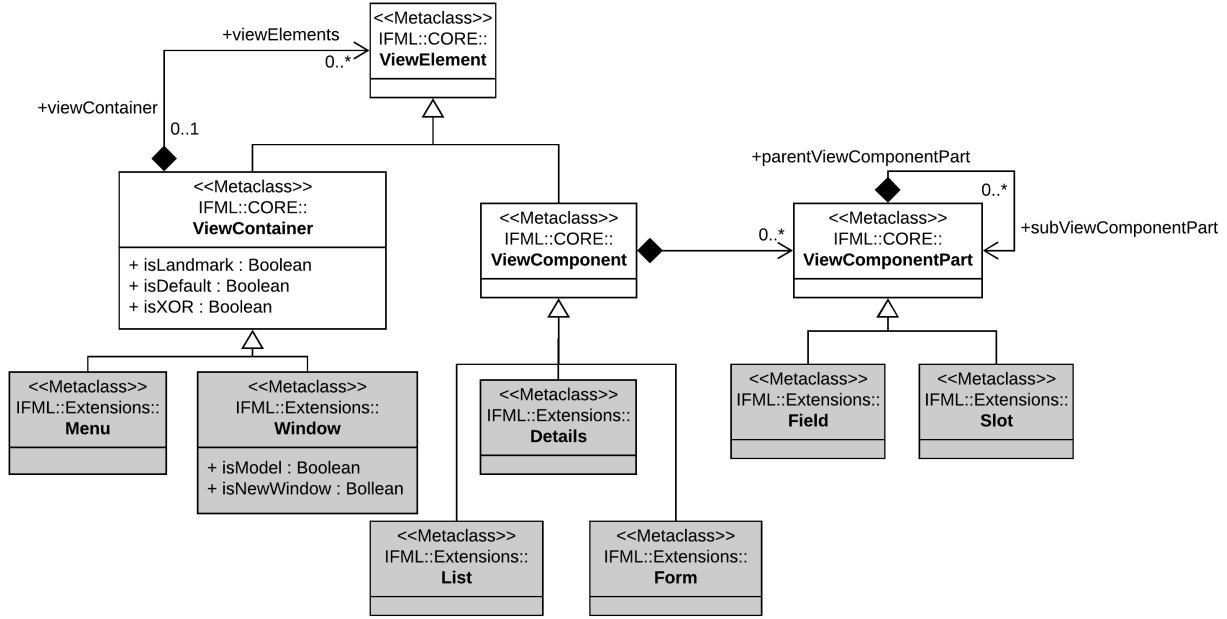


FIGURE 4 – IFML View Elements

DOM mais plus comme des données appartenant aux composants.

3.2.2 Méta-modèle d’interface utilisateur

Nous présentons dans cette Section les méta-modèles ou représentation d’interface graphique proposés dans la littérature. Nous comparons ces propositions avec ceux proposés par l’OMG.

Gotti *et al.*[11] ont proposé un méta-modèle inspiré du modèle KDM (voir la Section 3.2.1). Le méta-modèle a les principales entités définies dans le modèle KDM. On retrouve le patron de conception *composite* pour représenter le DOM d’une interface graphique. La notion de *UIElement* s’appelle *Components*. Les components comme les fenêtres ont une notion de *Property* qui a été ajouté par les développeurs pour représenter des informations comme, par exemple, la couleur d’un bouton. Ils ont également implémenté une propriété *Event* pour les components. Mais ils inversent les noms des notions *Event* et *Action* comparées aux modèles KDM.

Fleurey *et al.*[14] n’ont pas décrit le méta-modèle de l’interface graphique directement, mais nous avons extrait des informations de leur méta-modèle de navigation. Ils ont au moins deux éléments dans leur méta-modèle d’interface graphique, *Window* et *GraphicElement*. *Window* semble correspondre à la notion de *Display* du méta-modèle KDM et au conteneur racine de IFML. On peut supposer que le *GraphicElement* est un *UIResource*. Le *GraphicElement* a un *Event*. Nous n’avons pas la totalité du méta-modèle de l’interface graphique, mais nous pouvons voir qu’il ressemble à ceux proposés par l’OMG.

Le méta-modèle graphique de Sanchez *et al.*[1] est très à ceux de l’OMG. Il y a les entités *Widget* et *Window* qui correspondent respectivement aux entités *GUIElement* et *UIDisplay*. Leurs widgets ont une position X et Y et la largeur et hauteur en tant que propriétés. Ceci n’est pas représenté dans les méta-modèles KDM et IFML. Il y a aussi le patron de conception *composite* pour représenter le DOM.

Morgado *et al.*[10] utilisent un méta-modèle graphique, mais ne le décrivent pas. Nous savons seulement que l'interface graphique est représentée comme un arbre ce qui est similaire à un DOM et peut être représenté grâce au patron de conception *composite*.

Le méta-modèle graphique de Garces *et al.*[15] diffère beaucoup de ceux précédemment décrit. Il y a les attributs, les événements, les windows, mais il n'y a pas de widget. Cette absence s'explique par la différence dans le langage source à migrer. Les auteurs ont travaillé sur un projet utilisant des Oracle Forms. L'interface est décrite dans des fichier à part et est souvent composé d'affichage d'éléments d'une base de donnée plutôt que de widget. Nous pouvons tout de même remarquer qu'ils utilisent une entité *Event* pour représenter l'action de l'utilisateur avec l'interface utilisateur.

Memon *et al.*[7] représentent une interface utilisateur graphique avec seulement deux entités dans leur méta-modèle d'interface graphique. Une GUIWindow similaire au UIDisplay qui est constituée d'un ensemble de widgets. Ces widgets peuvent avoir des propriétés et toutes les propriétés ont une valeur associée. Les auteurs ont défini une interface utilisateur comme ensemble de widgets et leurs propriétés, ainsi, si un widget peut avoir deux valeurs différentes pendant l'exécution du programme, il appartient à deux interfaces utilisateur différentes. Ce point est la différence majeure avec les méta-modèles proposés par l'OMG car, dans la conception proposée par IFML si la valeur d'une propriété change, nous sommes toujours dans la même interface utilisateur, mais un flux d'interaction a été exécuté.

Smair *et al.*[8] ont travaillé sur la migration d'une application Java-Swing vers une application web Ajax. Ils ont créé un méta-modèle pour représenter l'interface utilisateur de l'application d'origine. Ce méta-modèle est stocké dans un fichier XUL et représente les widgets avec leurs propriétés ainsi que la mise en page. Ces widgets appartiennent à une fenêtre, appelée Phase dans notre travail, et peuvent déclencher un événement lorsqu'un *Input* GUI est exécuté. Dans les modèles de l'OMG, on ne retrouve pas la notion de propriété et l'*Input* est englobé soit dans paquetage *Event* pour KDM, soit dans flux d'interaction pour IFML.

Shah et Tilevich[9] utilisent un arbre pour représenter l'interface graphique. La racine de l'arbre est une *Frame* ce qui correspond à la notion de *UIDisplay*. La racine contient des *Composants* avec leurs propriétés. L'entité *Composant* est similaire à l'entité *UIField* bien que les propriétés ne sont pas représentées dans les méta-modèles KDM.

Joorachi *et al.*[23] représentent une interface utilisateur avec un ensemble d'éléments d'interface utilisateur. Ces éléments correspondent à la notre définition d'un *UIField*. Pour chaque élément d'interface utilisateur, l'outil des auteurs peut gérer la détection de plusieurs attributs et d'événements. Les attributs sont absents des modèles proposés par l'OMG. On ne retrouve pas le patron de conception *composite* dans le travail de ces auteurs.

Memon *et al.*[24] utilisent un modèle d'interface graphique pour représenter l'état d'une application. Ils ont aussi utilisé la notion de *UIField*. Les auteurs utilise le patron de conception *composite* afin de représenter le DOM d'une application.

Mesbah *et al.*[25] n'ont pas présenté directement le méta-modèle de l'interface utilisateur qu'ils utilisent. Cependant, ils utilisent une représentation avec un arbre pour analyser différentes pages web. Ils utilisent également la notion d'événement pouvant être déclenchée. Les auteurs instancient plusieurs méta-modèles d'interface utilisateur pour représenter les différentes pages web de l'application. Ces instances peuvent être comparées à plusieurs éléments *UIDisplay*.

Nous retrouvons dans les papiers la représentation du DOM grâce à avec un arbre ou un méta-modèle utilisant le patron de conception *composite*. Les notions de *UIDisplay* est aussi omniprésente. Contrairement à ce qui est utilisé dans les méta-modèles KDM et IFML,

l'utilisation d'une entité *Attribut* est souvent utilisé.

Maintenant que nous avons étudié comment les interfaces graphiques sont représentées dans la littérature et comment ces représentations sont construites. Il faut que l'on détaille les couches composantes une interface graphique.

4 Décomposition d'une application avec GUI

Avant de créer l'outil de migration, nous avons étudié puis décomposé la notion d'interface graphique. Diviser un problème en sous-problèmes est une méthode efficace pour résoudre des problèmes complexes. Nous avons identifié trois parties dans une interface graphique :

- La partie visuelle
- Le code comportemental
- Le code métier

4.1 Partie visuelle

La partie visuelle peut être mise en correspondance avec le méta-modèle UI de KDM (voir Section 3.2.1). Elle contient les différents éléments de l'interface. La partie visuelle définit les caractéristiques inhérentes aux composants, comme la possibilité d'être cliqué, ou certaines propriétés du composant, comme sa couleur ou sa taille. Plus que les composants, elle décrit également la disposition de ces composants par rapport aux autres. Nous avons vu que cela est souvent représenté grâce au patron de conception *composite*. Dans le cas où une application est composée de plusieurs fenêtres (ou de pages web pour une application web), la partie visuelle contient toutes les fenêtres.

4.2 Code comportemental

Le code comportemental définit le *flow* d'action qui s'exécute lorsqu'un utilisateur interagit avec la partie visuelle de l'application. L'utilisateur peut effectuer une action sur un composant de l'interface (comme un clic). Il est aussi possible que ce soit le système lui-même qui décide de déclencher une suite d'action suite à un événement extérieur. Comme dans un langage de programmation “*classique*”, le code comportemental contient des structures de contrôle (*c.-à-d.* boucle et alternative).

4.3 Code métier

Le code métier définit tout ce qui n'est défini ni dans la partie visuelle ni dans le code comportemental. Ce correspond à tous ce qui est lié à l'application mais pas à la partie visuelle. Il est composé des règles générales de l'application (comment calculer les taxes ?), de l'adresse des services distants (quel serveur mon code métier doit demander), des données de l'application (quelle base de données ? quel type d'objet).

5 Mise en place de la migration par les modèles

Suite à l'étude des contraintes inhérentes aux problèmes de migration dans le cadre d'une entreprise, et à l'état de l'art que nous avons mené, nous avons travaillé sur la conception et l'implémentation d'une stratégie de migration respectant les critères que nous avons fixés.

Section 5.1, nous présentons les différentes approches que nous avons identifié pour effectuer la migration. Puis, Section 5.2 nous décrivons le processus de migration que nous avons conçu. Et enfin Section 5.3 et Section 8.4 nous présentons les différents outils que nous avons du créer ainsi que leurs implémentations.

5.1 Approches pour effectuer la migration

Nous avons identifié plusieurs manières d'effectuer la migration d'une application. Toutes les solutions doivent respecter les contraintes définies Section 2.1.

- *Migration manuelle.* Cette stratégie correspond au redéveloppement complet des applications sans l'utilisation d'outils aidant à la migration. La migration manuelle permet de facilement corriger les potentielles erreurs de l'application d'origine et de reconcevoir l'application cible en suivant les préceptes du langage cible.
- *Utilisation d'un moteur de règles.* L'utilisation d'un moteur de règles pour migrer partiellement ou en totalité une application a déjà été appliquée sur d'autres projets [19], [26], [27]. Pour utiliser cette stratégie, nous devons définir et créer des règles qui prennent en entrée le code source et qui produisent le code pour l'application migrée. par exemple, une règle de transformation permettant de déplacer l'opérateur d'une expression mathématique en la plaçant en tant que suffice de l'expression transformerai l'expression "`4 + 2`" en "`4 2 +`". Il est envisageable d'effectuer la migration partiellement avec cette approche. Dans ce cas, les développeurs devront finir le processus de migration avec du travail manuel. L'utilisation d'un moteur de règles, bien qu'efficace, implique une solution qui n'est ni indépendante de la source ni indépendante de la cible de la migration.
- *Migration dirigée par les modèles.* La migration dirigée par les modèles implique le développement de méta-modèles. La stratégie respecte l'ensemble des contraintes que nous avons défini. Une migration semi-automatique ou complètement automatique est envisageable avec cette stratégie de migration. Comme pour l'utilisation d'un moteur de règle, dans le cas d'une migration semi-automatique, il peut y avoir du travail manuel à effectuer pour achever la migration.

L'utilisation d'un moteur de règles comme la migration dirigée par les modèles sont envisageable. Cependant, la migration par les modèles pourrait permettre de généralisé nos prototypes et d'après notre étude de la littérature c'est une approche courante pour effectuer la migration d'application. Nous avons donc décidé d'utiliser cette approche pour effectuer la migration.

5.2 Processus de migration

À partir de l'état de l'art et des contraintes que nous avons explicitées, nous avons conçu une stratégie pour effectuer la migration. Le processus que l'on a représenté Figure 5 est divisé en cinq étapes :

1. *Extraction du modèle de la technologie source* est la première étape permettant de construire l'ensemble des analyses et transformations que nous devons appliquer pour effectuer la migration. Elle consiste en la génération d'un modèle représentant le code source de l'application originel. Dans notre cas d'étude, le programme source est en Java et donc le modèle que nous créons est une implémentation d'un méta-modèle permettant de représenter une application écrite en Java.

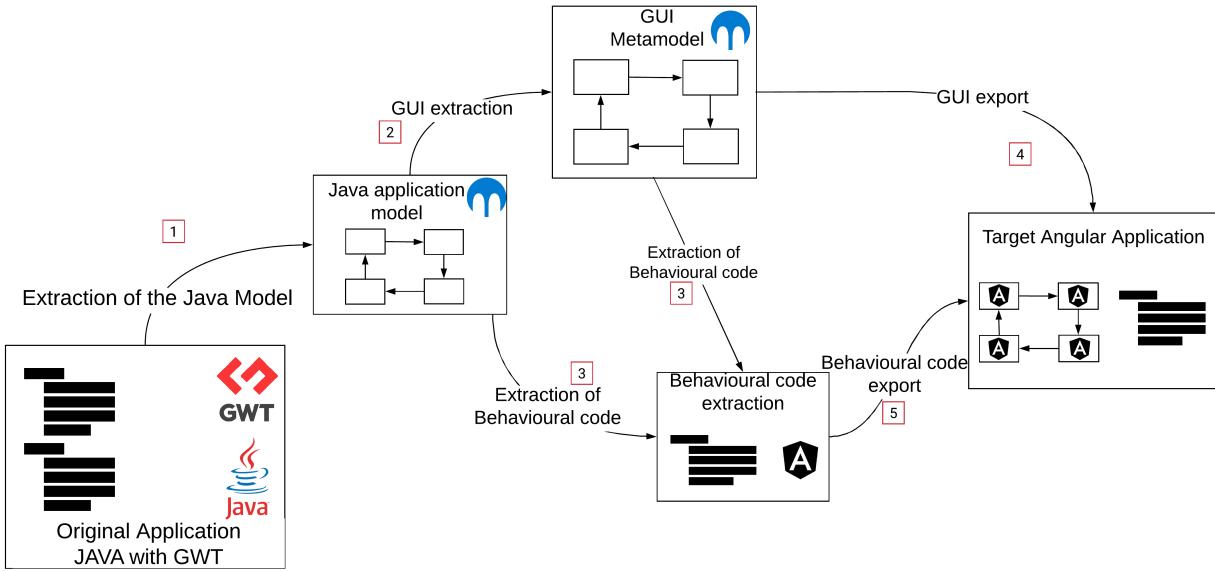


FIGURE 5 – Schéma processus de migration

2. *Extraction de l'interface utilisateur* est l'analyse du modèle de la technologie source pour détecter les éléments qui relèvent du modèle d'interface utilisateur. Ce dernier, que nous avons dû concevoir, est expliqué Section 5.3.
3. *Extraction du code comportemental*. Une fois le modèle d'UI généré, il est possible d'extraire le code comportemental du modèle de la technologie source et de créer les correspondances entre les éléments faisant partie à la fois du code comportemental et du modèle d'interface utilisateur (UI). Par exemple, si un clic sur un bouton agit sur un texte dans l'interface graphique. L'extraction du code comportemental permet de définir que pour le bouton, défini dans le modèle UI, lorsqu'un clic est effectué, on effectue un certain nombre d'actions, dont une sur le texte, lui aussi défini dans le modèle UI.
4. *Exportation de l'interface utilisateur*. Le modèle d'interface graphique étant construit et les liens entre interfaces utilisateur et code comportemental créés, il est possible d'effectuer l'exportation de l'interface utilisateur. Cela consiste en la génération du code du langage source exprimant uniquement l'interface graphique. C'est aussi à cette étape que l'on génère l'architecture des fichiers nécessaires au fonctionnement de l'application cible ainsi que la création des fichiers de configuration inhérente à l'interface.
5. Finalement, l'*Exportation du code comportemental* est la génération du code comportemental qui est lié à l'interface utilisateur. Cette étape peut être effectuée en parallèle de la précédente.

5.3 Méta-modèle d'interface utilisateur

Afin de représenter les interfaces utilisateurs des applications de Berger-Levrault, nous avons conçu le méta-modèle illustré Figure 6. Ce méta-modèle est la synthèse des méta-modèles que nous avons trouvé lors de l'état de l'art et présenté Section 3.2.2. Dans la suite de cette section, nous présentons les différentes entités du méta-modèle.

La **Phase** représente le conteneur principal d'une page interface utilisateur. Cela peut

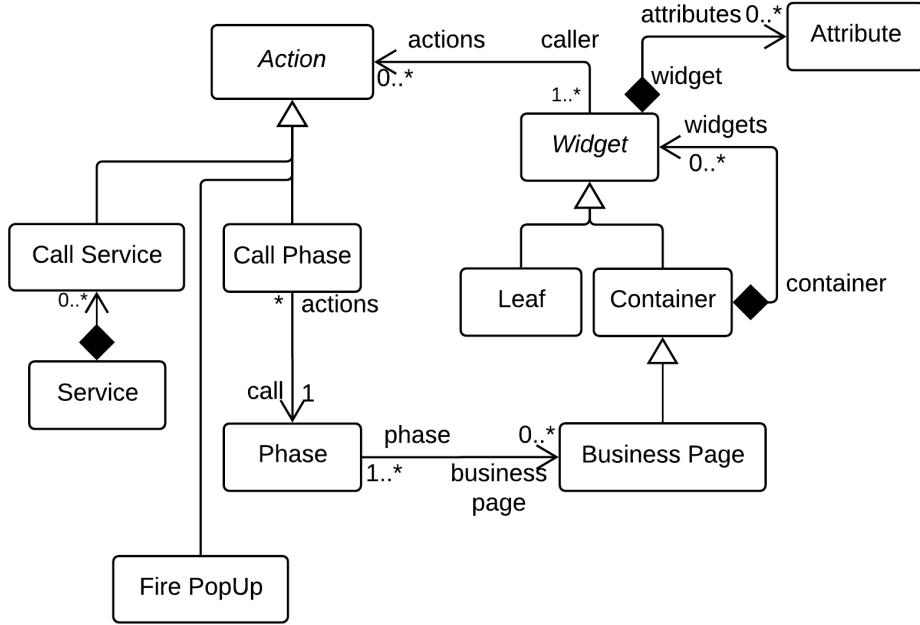


FIGURE 6 – Méta-Modèle d’interface utilisateur

correspondre à une *fenêtre* d’une application du bureau, une page web, ou dans notre cas d’un onglet à une page web. Dans le modèle KDM, une Phase correspond à un *Screen* (type de UIDisplay).

Une Phase peut contenir plusieurs **Business Page**. Elle peut aussi être appelée par un widget grâce à une **Action** de type **Call Phase**. Lorsqu’une Phase est appelée, l’interface change pour l’afficher. Dans le cas d’une application de bureau, l’interface change ou une nouvelle fenêtre est ouverte avec l’interface de la Phase. Pour une application web, l’appel d’une Phase peut correspondre à l’ouverture d’un nouvel onglet, le changement d’onglet actif ou la transformation de la page web courante. Cette notion de business page n’est présente dans aucun des autres papiers, elle est inhérente au projet de Berger-Levrault et peut facilement être modifier pour correspondre à ce que l’on trouve dans la littérature. Il suffirait de changer la relation entre phase et business page pour une relation entre phase et widget.

Les **Widgets** sont les différents composants d’interface et les composants de disposition. Il existe deux types de widgets. Le **Leaf** est un widget qui ne contient pas d’autre widget. Le **Container** qui peut contenir un ou plusieurs autres widgets. Ce dernier permet de séparer les widgets en fonction de leur place dans l’organisation de la page web représentée, pour cela nous utilisons le patron de conception *composite* fortement utilisé dans la littérature.

Les **Attributes** représentent les informations appartenant à un widget et peuvent changer son aspect visuel ou son comportement. Des attributs communs sont la hauteur et la largeur pour définir précisément la dimension d’un widget. Il y a aussi des attributs qui contiennent des données. Par exemple, un widget représentant un bouton peut avoir un attribut *text* qui explicite le texte du bouton. Un attribut peut changer le comportement d’un widget, c’est le cas de l’attribut *enable*. Un bouton avec l’attribut *enable* positionné sur *false* représente un bouton sur lequel nous ne pouvons pas cliquer. Enfin, les widgets peuvent avoir un attribut qui aura un impact sur le visuel de l’application. Ce type d’attribut permet de définir un

layout à respecter par les widgets contenus dans un autre et potentiellement les dimensions de ce dernier pour respecter une mise en pages particulière. Nous ne retrouvons pas la notion d'attribut dans les métamodèles proposé par l'OMG, cependant ils ont été introduit par les papiers [1], [4], [7]–[11], [15], [23]–[25], [28].

Les **Actions** sont propres aux widgets. Elles représentent des actions qui peuvent être exécutées dans une interface graphique. C'est une notion qui est présente dans les métamodèles proposé par l'OMG. **Call Service** représente un appel à un service distant comme une URL sur internet. **Fire PopUp** est l'action qui affiche un pop-up sur l'écran. Le pop-up ne peut pas être considéré comme un widget, il n'est pas présent dans l'interface graphique, il apparaît seulement et disparaît.

Le **Service** est la référence de fonctionnalité distante que l'application peut appeler à partir de son interface graphique. Dans le contexte d'une application client/serveur, il peut s'agir du côté serveur de l'application.

5.4 Implémentation du processus

Pour tester la stratégie, nous avons implémenté un outil qui suit le processus de migration. L'outil a été implémenté en Pharo⁸ et nous avons utilisé la plateforme Moose⁹.

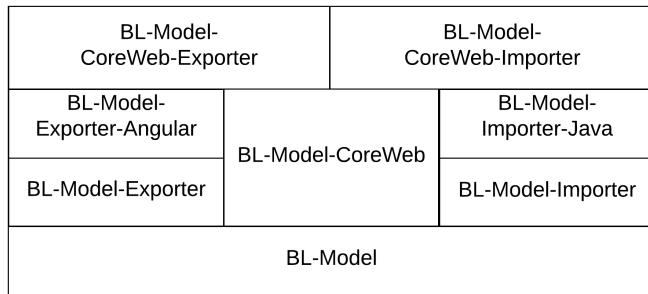


FIGURE 7 – Implémentation de l'outil

Le Figure 7 présente la logique d'implémentation. Le bloc principal est *BL-Model*. Ce bloc contient l'implémentation du métamodèle GUI¹⁰. En plus du modèle, il y a un exportateur abstrait et une implémentation de l'exportateur pour Angular (*BL-Model-Exporter* et *BL-Model-Exporter-Angular*) et un importateur abstrait ainsi que le code spécifique pour Java (*BL-Model-Importateur* et *BL-Model-Importer-Java*). Parce que nous testons notre solution sur le système de Berger-Levrault, nous avons également implémenté l'extension “Core Web”, la stratégie de migration ne dépend cependant pas de cette extension. Ces paquets étendent les précédents pour avoir un contrôle fin du processus de migration. Ce contrôle est important pour améliorer le résultat final.

5.4.1 Importation

La création des modèles représentant l'interface graphique est divisée en trois étapes comme présenté Section 5.2. Dans le cas de Berger-Levrault, nous avons implémenté l'approche en Pharo avec Moose.

8. Pharo est un langage de programmation objet, réflexif et dynamiquement typé - <http://pharo.org/>

9. Moose est une plateforme pour l'analyse de logiciels et de données - <http://www.moosetechnology.org/>

10. GUI : Graphical User Interface - Interface Graphique

La première étape est la conception du modèle de la technologie source. Ce modèle avait déjà une implémentation existante dans Moose avec le projet *Famix-Java*. Nous avons donc réutilisé ce modèle pour ne pas avoir à reconcevoir un modèle préexistant. De plus, ce travail préliminaire est compatible avec plusieurs outils qui ont été développés en interne à RMod. Entre autres, deux logiciels de génération du modèle Famix-Java depuis du code source Java existent. Les outils sont verveineJ¹¹ et jdt2Famix¹². Ces deux derniers permettent de créer depuis le code source un fichier *mse*. Le fichier *mse* peut ensuite être importé dans la plateforme Moose. Dans le cadre du projet avec Berger-Levrault, nous avons utilisé verveineJ car ce dernier permet aussi de *garder un lien* entre le modèle généré et le code à partir duquel il l'a été.

Une fois le modèle de la technologie source créé, et après avoir implémenté nos métamodèles, nous avons développé des outils en Pharo permettant d'effectuer la transformation du modèle source vers le modèle GUI. Nous allons maintenant décrire les techniques utilisées pour retrouver les éléments définis dans le modèle GUI depuis le modèle de technologie source.

```

1 public class SamplePageMetier1 extends AbstractSimplePageMetier {
2     @Override
3     public void buildPageUi(Object object) {
4         BLLinkLabel lblPgSuivante = new BLLinkLabel("Next page");
5         lblPgSuivante.addClickHandler(new ClickHandler() {
6             @Override
7             public void onClick(ClickEvent event) {
8                 SamplePageMetier1.this.fireOnSuccess("param 1");
9             }
10        });
11        lblPgSuivante.setEnabled(false);
12        // add the content
13        vpMain.add(new Label("<Business content>"));
14        vpMain.add(lblPgSuivante);
15        super.setBuild(true);
16    }
17 }
```

FIGURE 8 – Définition d’interface graphique en Java

La Figure 8 présente un extrait du code de l’application *bac à sable*. Il s’agit de la méthode `buildPageUi(Object object)` qui contient le code à exécuter pour construire l’interface graphique de la business page “SamplePageMetier1”.

Les premiers éléments que nous avons voulu reconnaître sont les phases. En analysant les projets GWT, nous avons repéré un fichier *.xml* dans lequel est stocké toutes les informations des phases. Nous avons donc ajouté une étape à l’importation qui est l’analyse d’un fichier *XML*. Ce fichier nous permet de “*facilement*” récupéré la classe java correspondant à une phase, ainsi que le nom de la phase.

Ensuite, nous avons développé l’outil d’importation de manière incrémentale. Nous avons donc cherché les business pages. Grâce à l’analyse préliminaire des applications de Berger-Levrault, nous avons détecté que les business pages en GWT correspondent à des classes qui implémentent l’interface *IPageMetier*. Dans la Figure 8, la classe *SamplePageMetier1* étend *AbstractSimplePageMetier*, et ce dernier implémente l’interface. Une fois les classes trouvées,

11. verveineJ : <https://rmod.inria.fr/web/software/>

12. jdt2famix : <https://github.com/feenkcom/jdt2famix>

nous avons recherché les appels des constructeurs des classes. Puis, en faisant le lien entre les appels et les phases qui “ajoute” à leurs contenus, nous avons détecté les liens d’appartenances entre les business pages et les phases.

Pour les widgets, nous avons dû tout d’abord trouver tous les widgets potentiellement instanciable. Pour cela, nous avons cherché toutes les sous-classes Java de la classe GWT *Widget*. Ce sont les classes qui vont pouvoir être instanciées et utilisées pour la construction du programme. Ensuite, comme pour les business pages, nous avons cherché les appels des constructeurs des widgets et avons relié ces appels à la business page qui les a ajoutés. Dans la Figure 8, il y a deux appels à des constructeurs de widget. Le constructeur de BLLinkLabel est appelé ligne 4 et celui de Label ligne 13. La variable `vpMain` correspond au panel principal de la business page. Les lignes 13 et 14 correspondent à l’ajout d’un widget dans une business page grâce aux appels à la méthode `add()`.

Enfin, pour la détection des attributs et des actions associés à un widget. Nous avons, pour chaque widget, cherché dans quelle variable Java il a été affecté. Puis nous avons cherché les appels de méthodes effectués depuis ces variables java. Les appels aux méthodes “`addActionHandler`” sont transformés en action tandis que les appels aux méthodes “`setX`” ont été transformés en attribut. Cette approche pour détecter les attributs et les actions provient des papiers de la littérature [4], [8]. Dans le cas de la Figure 8, le `BLLinkLable`, dont la variable est `lblPgSuivante`, est lié à une action et à un attribut. Les lignes 5 à 10 correspondent à l’ajout d’une action et le code à exécuter quand l’action est exécutée. La ligne 11 correspond à l’ajout de l’attribut “Enabled” avec comme valeur `false`.

5.4.2 Exportation

Une fois la génération du modèle d’interface graphique et du modèle du code comportemental terminé, il est possible de lancer l’exportation. L’exportation consiste en la génération du code de l’application cible.

La première étape de l’implémentation de l’exportation est l’utilisation du patron de conception “visiteur”. Ce dernier est appliqué au modèle d’interface graphique.

La visite du modèle GUI crée la hiérarchie de l’application cible ainsi que les fichiers de configuration. Ensuite, l’exportation visite toutes les phases. Pour chacune des phases, considérées comme des sous-projets en Angular dans l’architecture de l’application cible que nous avons définie, le visiteur génère les fichiers de configurations. Puis, pour chaque business page, le visiteur génère un fichier HTML et un fichier TypeScript. Pour le fichier html, le visiteur construit le DOM à partir des widgets contenus dans la business page. Les widgets connaissant leurs attributs et actions, ils fournissent eux-mêmes leurs caractéristiques aux visiteurs. Ces caractéristiques englobent la génération du code comportemental.

6 Résultats

Nous allons maintenant présenter les résultats que nous avons obtenus suite à l’implémentation de la stratégie de migration.

Nous avons expérimenté notre approche sur l’application *bac à sable* de Berger-Levrault. Nous présentons dans cette Section les résultats que nous obtenons aux différentes étapes du processus de migration. Section 6.1 présente les résultat de la phase de rétro-ingénierie. Puis, nous présentons, Section 6.2, une visualisation que nous avons créé pour analyser le modèle que nous avons instancié. Finalement, Section 6.3, nous comparons le résultat final avec les

contraintes que nous avons fixé Section 2.1.

6.1 Résultat de l'importation

Phases	Business Pages	Widgets	Link between Phases
56	76	2081	101
100 %	100 %	98 %	100 %

Tableau 2 – Résultat de l'importation

Le Tableau 2 présente les résultats que nous obtenons en exécutant la création du modèle d'interface graphique.

Pour les phases, nous avons regardé dans l'application cible le nombre de phases existantes. Pour cela nous avons regardé dans le fichier de configuration de l'application dans lequel toutes les phases sont définies. Puis nous avons comparé le chiffre obtenu avec le nombre de phases que nous instancions dans notre modèle. Dans le cas d'étude avec Berger-Levrault, nous avons recensé 56 phases, ce qui correspond exactement au nombre de phases déclaré dans le fichier de configuration.

Pour les business pages, nous avons compté 45 classes qui implémentent *IPageMetier*. Après l'importation, nous identifions 76 pages métiers. Nous retrouvons dans les pages métiers celles qui implémentent l'interface *IPageMetier* ainsi que 31 qui dernières proviennent de code qui a été factorisé par les développeurs de l'application. La factorisation du code est une complication dans le calcul du nombre exact de business page que nous devons détecter pendant l'importation. Cette difficulté d'évaluation est discutée Section 7.

Nous réussissons à identifier 2081 widgets, cependant avec les heuristiques que nous avons définis Section 5.4.1 nous devrions avoir 2141 widgets. Ce qui correspond à un total de 98 % de widget que nous réussissons à créer. Il existe cependant un écart que nous n'arrivons pas encore à évaluer dont l'on discute Section 7.

Finalement, la détection du nombre de liens entre les phases est réussie à 100 %. Nous détectons correctement 101 liens de navigation qui existent dans l'application. Les liens sont tous correctement connectés aux widgets sur lesquels il faut faire une action pour déclencher la transition et amène vers la bonne Phase.

6.2 Visualisation

Pendant la construction de l'outil de migration, nous avons créé des requêtes sur le modèle d'interface graphique. Ces requêtes permettent de créer des graphiques et d'analyser la construction de l'interface graphique sans regarder le code source.

La Figure 9 présente un extrait de la visualisation des relations entre les différentes entités de l'interface graphique de l'application *bac à sable*.

Le rond noir est la représentation d'une phase. Ici, il correspond à la phase “*Sample_Editable_List*”. Nous pouvons retrouvé comment la phase est appelé en suivant la flèche bleue. La phase contient aussi une business page, représenté en rouge,. Il s'agit de la page “*SamplePageEditableLists*”. Cette dernière contient 9 widgets, représentés en vert. Celui du dessous est la représentation d'une instance d'un composant BLGrid provenant de BLCore. Ce type de composant peut contenir d'autre widgets. Dans cette exemple il en contient deux.

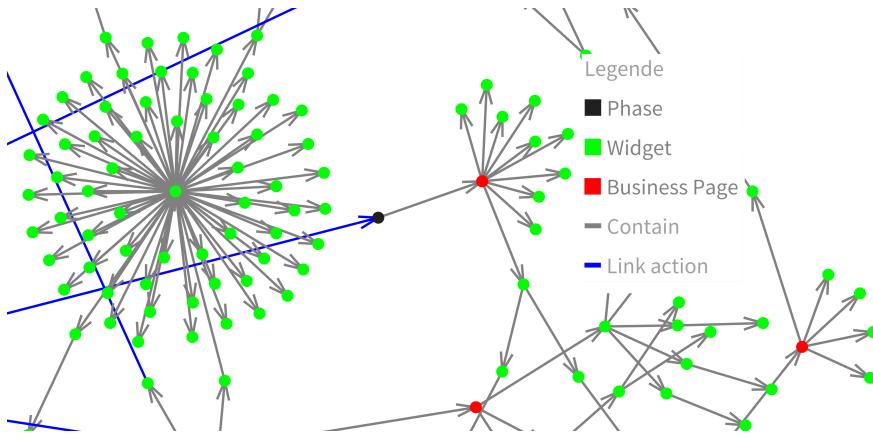


FIGURE 9 – Extrait de la représentation de l'application *bac à sable*

Nous pouvons voir sur le côté gauche de l'image un agglomérat de widget. Cette disposition d'élément est courante, elle représente un widget de type *container* qui contient beaucoup d'autre widgets pouvant être des *leafs* ou des *containers*. Ici, il s'agit d'un BLGrid qui contient des textes et des champs de saisies.

La représentation complète que nous obtenons est disponible en Annexe.

6.3 Exportation en Angular

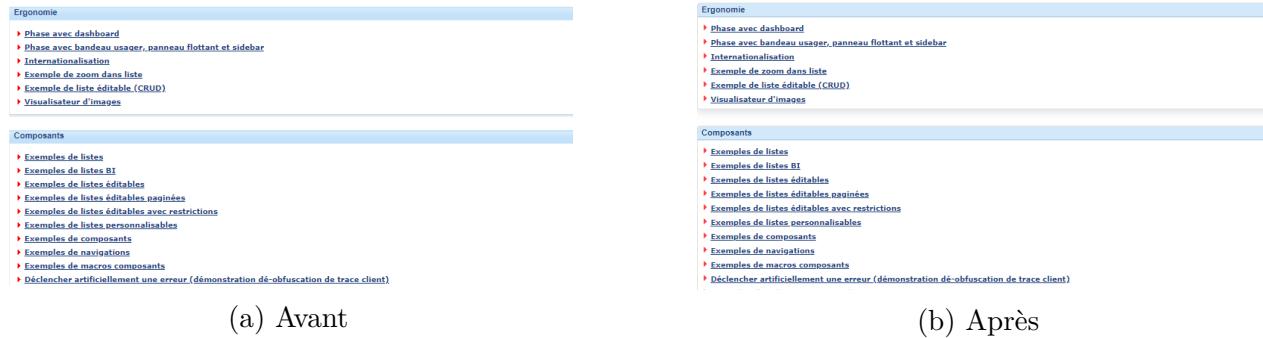


FIGURE 10 – Phase Home

Le code exporté est compilable à 100 % et est exécutable. L'exportation conserve l'architecture entre les éléments de l'interface graphique tels que détectés dans la partie importation. La Figure 10 présente les différences visuelles entre l'ancienne version, à gauche, et la nouvelle, à droite. Nous pouvons voir que les différences sont minimes. Dans la version exportée, les couleurs de l'en-tête des panels sont un peu plus claires et l'ombre portée des panels est plus dégradée.

La Figure 11 présente les différences visuelles pour la Phase *Zone de saisie* de l'application *bac à sable*. L'image de gauche correspond à la phase avant la migration tandis que celle de droite est la même Phase après la migration. Les deux images étant grandes, nous les avons rognées pour afficher cette zone d'intérêt. Bien que les deux images ont l'air complètement différentes, tous les widgets sont présent dans la version migré. La différence visuel est du à un problème dans la gestion des layout. Ce point est discuté Section 7.1.

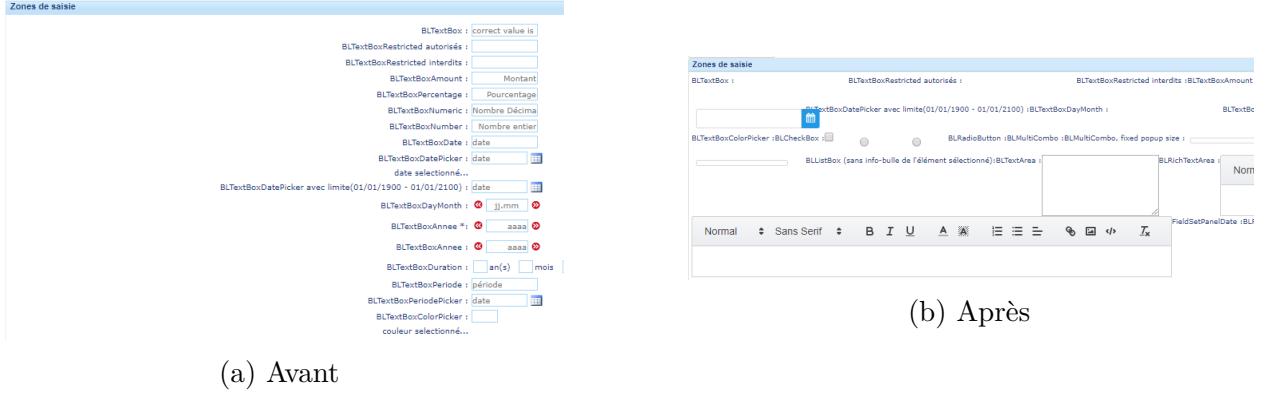


FIGURE 11 – Phase Zone de saisie

7 Discussion

Bien que nous ayons testé régulièrement notre travail sur les applications de production de Berger-Levrault, la recherche des patterns pour l’importation ainsi que l’évaluation de la migration n’a été faite que sur l’application *bac à sable*. Nous savons que l’application après migration compile, mais nous n’avons pas de retours sur la réussite de l’exportation du visuel. Il est aussi possible que les autres logiciels de Berger-Levrault contiennent des déviations dans le code que nous n’avons pas prévu ce qui peut nuire au résultat final.

En GWT, il est possible de définir une interface graphique grâce à un fichier XML. Dans le cadre de ce projet, seule l’application *bac à sable* utilise cette technique pour définir des interfaces. Nous avons donc décidé de ne pas traiter l’importation des widgets pour les business pages définit de cette manière. En les considérant, le pourcentage de widget que nous arrivons à importer se voit réduit.

7.1 Gestion des layout

Comme expliqué Section 6.3, certains résultats de l’exportation ne sont pas correct à cause d’un manque de respect des layouts. C’est le cas de la Figure 11. Ceci est dû à la manière dont est retranscrite l’idée de layout dans le méta-modèle d’interface graphique. Pour le moment, le layout est considéré comme un attribut, ce qui rend difficile la représentation d’interfaces complexes. Plusieurs solutions peuvent être envisagées pour représenter les interfaces graphiques.

Gotti et Mbarki [11] ont décidé d’utiliser le méta-modèle proposé par KDM¹³. Le méta-modèle KDM de layout utilise une association entre deux *AbstractUIElement* pour représenter la position de l’un par rapport à l’autre. C’est ainsi qu’est défini le layout d’une application.

Sánchez Ramón *et al.* [1] ont créé un méta-modèle pour les layouts. Ce méta-modèle propose de lier la notion de widget avec celle de layout et de combiner les layouts afin de créer un layout précis. Les auteurs ont défini un sous-ensemble de layout qu’il est possible de connecter aux widgets. Cette solution est celle qui se rapproche le plus des techniques de conception que nous utilisons.

13. KDM : Knowledge Discovery Metamodel

7.2 Gestion du comportement

Afin d'améliorer la représentation que nous avons d'une application graphique, il faudrait que l'on représente toutes les couches d'une interface graphique comme décrite Section 4. Une fois le layout implémenté nous pourrons bien représenter l'interface utilisateur, il nous faudra encore définir et implémenter des méta-modèles pour le code comportemental et le code métier.

Nous avons déjà défini un méta-modèle de code comportemental qui est présenté Section 8.4. Il faut encore implémenter ce modèle et modifier en conséquence les importateur et exportateur que nous avons créés. Cette amélioration devrait nous permettre de mieux représenter les comportements de l'application lorsqu'un utilisateur effectue une action sur l'interface.

Enfin, il nous faut concevoir un méta-modèle pour le code métier. Cela devrait permettre d'extraire et représenter les règles métiers d'une application.

8 Travaux futurs

Nous avons réussi à construire des bases solides pour l'outil de migration. Afin de l'améliorer, il faudrait encore ajouter la gestion des layout, du code comportemental et du code métier. Nous allons devoir corriger les quelques erreurs restantes qui nous empêchent d'importer 100 % des widgets que nous souhaitons exporter. Enfin, nous devons définir une ou plusieurs stratégies pour évaluer correctement la complétude de notre travail.

8.1 Outil de validation

Un des problèmes que nous avons actuellement est l'évaluation de la migration. Il est facile de prendre l'application source et l'application cible et de regarder *manuellement* si la migration a bien été effectuée, mais sur des applications de la taille de celles de Berger-Levrault nous avons besoin d'un système automatique. Les éléments à évaluer sont les règles métiers dans l'application cible, le respect du visuel et l'application du code comportemental.

Pour le respect du visuel, plusieurs solutions peuvent être envisagées dans le cas de Berger-Levrault.

Il est possible de comparer les DOM des applications car nous travaillons avec des applications web. Cependant le DOM ne prend en compte que la structure d'une page web et non pas sa représentation visuelle. Une étude des fichiers CSS serait alors indispensable. De plus, le fichier généré pour Angular n'est pas forcément une copie du fichier généré par GWT, il peut y avoir des différences dans le DOM qui ne sont pas répercuté dans l'aspect visuel de la page web.

Une autre solution est de comparer le visuel des pages web par comparaison d'image. Il faudrait alors prendre une impression de chaque phase dans chacun de ses états, qui peut varier si l'on exécute du code comportemental, et la comparer avec l'impression de la page générée. Pour comparer les images, il est possible d'utiliser des solutions de comparaison pixel par pixel ou d'utiliser une intelligence artificielle qui reconnaît les images similaires.

Pour la validation des règles métiers et l'application du code comportemental, il est possible d'utiliser les tests utilisés en recette à Berger-Levrault ou d'utiliser de la comparaison d'image.

Il est possible que les tests automatiques ne fonctionnent plus après la migration, en effet, ils peuvent être basés sur des métadonnées que nous modifions pendant la migration, par exemple pour des raisons de compatibilité dans le langage cible. Il nous faudra alors effectuer

la migration des tests d'intégration de Berger-Levrault pour les rendre compatibles avec l'application migrée.

Dans le cas d'un manque de tests pour une fonctionnalité, nous pourrions aussi créer des outils qui se basent sur les modèles de l'application à migrer pour générer des tests. Ces outils assisteront les recetteurs de l'entreprise et permettront de tester l'outil de migration final.

Dans le cas de la comparaison d'image, nous pouvons nous inspirer des travaux de Joorabchi *et al.* [23] qui l'ont utilisé pour la génération des états pour leur méta-modèle de state flow.

8.2 Complétude du travail

Le travail que nous avons mené nous a permis de migrer des pages web de l'application *bac à sable*. Cependant, comme présenté Section ??, nous n'arrivons pas à migrer 100 % des widgets. De plus, à cause du problème d'outil de validation, il est difficile de savoir si la migration s'effectue complètement, et correctement.

Comme nous avons vérifié nos travaux avec les mêmes pages web qui nous ont permis de créer les outils de migration, nous n'avons pas connaissance de potentielles erreurs, ou oubli, qui peuvent rendre l'outil moins performant. Afin de créer un outil complet, nous devrions chercher et gérer tous les écarts de programmation potentiels pour l'importateur.

De même, nous savons que nous n'avons pas géré le cas des interfaces graphiques défini via fichier XML. Bien que cela ne semble pas bloquant dans le cas des projets de Berger-Levrault car cette solution n'est utilisée que dans l'application *bac à sable*. Il serait intéressant de voir s'il est “*facile*” d'ajouter cette fonctionnalité. En particulier, cela permettrait d'analyser l'extensibilité de l'importateur.

8.3 Exportation du Core

L'approche que nous avons créée permet d'effectuer la migration d'interface graphique d'un langage source vers un langage cible. De plus, il est possible de configurer l'outil pour que l'application cible utilise tel ou tel *framework* pour représenter les éléments graphiques de l'interface finale.

Dans le cas d'une application web, un certain nombre de widgets sont déjà préexistants. Pour chacun d'entre eux, nous retrouvons un tag HTML. Par exemple, il y a *input*, *div*, *a* (pour un lien), etc.

Lors de la migration, l'utilisation de ces widgets nous permet d'avoir rapidement un retour visuel. Mais sans l'exportation du style précis des widgets, il est impossible de respecter la contrainte de *préservation du visuel*.

De plus, il peut exister des widgets définis dans le langage source qui n'existent pas, ou partiellement, dans le langage cible. Dans ce cas, il faut soit accepter une différence entre les deux interfaces, soit créer ou utiliser un autre *framework* dans le langage cible.

Pour la migration que veut mettre en place Berger-Levrault, le *framework* utilisé dans le langage source est BLCore. Comme il n'existe pas de *framework* BLCore utilisable en Angular, nous avons utilisé le *framework* PrimeNG qui nous permet de facilement retrouver la majorité des widgets décrits dans BLCore. Il reste cependant des écarts visuels entre les deux frameworks et certains widgets, comme la BLTableBulk qui est un widget abondamment utilisé dans les applications de l'entreprise, n'a pas de correspondance.

Afin de respecter la contrainte de *préservation du visuel*, une solution est donc de créer un équivalent du *framework* BLCore de Java en Angular. La migration d'un *framework* consiste

en la détection des différents widgets qu'il définit, avec leurs compositions, leurs styles et leurs code comportemental.

Un travail futur serait de développer un outil qui permettrait de détecter tous ces composants d'un widget et ainsi de faciliter leurs migrations. En fonctionnant en symbiose avec l'outil que nous avons créé pendant ce stage, il serait ainsi possible d'effectuer la migration d'une application graphique en respectant la hiérarchie des widgets, les contraintes de layout et le visuel et comportement des widgets.

8.3.1 Méta-modèle de navigation et de state flow

On retrouve dans la littérature deux autres méta-modèles très présents. Le méta-modèle de navigation et le méta-modèle de state flow.

Le méta-modèle de navigation permet de représenter un lien entre deux pages web ou fenêtres différentes. Le lien peut être fait d'une page web à une autre, ou d'un widget vers une page web. Le méta-modèle de navigation peut aussi contenir un lien d'un widget vers un événement, et un autre de cet événement vers une page web.

Morgado *et al.* [10] et Fleurey *et al.* [14] ont utilisé un méta-modèle de navigation pour représenter les liens entre les différentes interfaces utilisateur qu'ils détectent. Les premiers utilisent un méta-modèle supplémentaire qui décrit simplement l'ensemble des *fenêtres* possible dans l'application. Son méta-modèle de navigation permet de faire le lien entre une action sur un widget et l'action de navigation qui en résulte.

Pour Fleurey *et al.*, le méta-modèle contient directement un lien entre la fenêtre de départ et celle d'arrivée. Le méta-modèle possède tout de même une entité appelée *Operation*, mais qui ne semble pas impliquée dans la navigation.

Les informations de ces méta-modèles sont complètement contenues dans notre méta-modèle du code comportemental. Donc, bien que nous n'ayons pas de méta-modèle de navigation, nous faisons au moins ce qui est proposé dans ces papiers.

Le méta-modèle de state flow permet de créer le lien entre différents états de l'interface utilisateur. Un état de l'interface utilisateur est défini par les widgets visibles et leurs propriétés. Ainsi, si la valeur d'une propriété change, un nouvel état est généré.

Les auteurs Memon *et al.*[24], Mesbah *et al.*[25], Amalfitano *et al.*[28] , Silva *et al.*[4] et Aho *et al.*[29] ont tous utilisé un méta-modèle de state flow afin de représenter les différentes transitions entre les interfaces graphiques. Pour définir un état, leurs outils vont analyser les valeurs des propriétés des widgets visibles sur un écran. Une fois une action exécutée, l'outil détecte si l'état d'un widget a changé, dans ce cas un nouvel état de l'application est créé. Ainsi, les auteurs sont capables de représenter les impacts d'une action sur l'interface graphique.

Pour définir un état, Joorabchi *et al.* [23] ont décidé d'effectuer une comparaison d'image. Après chaque action sur un widget, exécuté par un outil, ils prennent une image de l'application et la comparent avec une image prise avant l'action. Si les deux images sont différentes, alors les auteurs ont découvert un changement d'état provenant d'une action.

Le méta-modèle de state flow permet de représenter l'impact d'un événement sur l'interface utilisateur. Le code à executer sur l'interface afin de passer d'un état à un autre est contenu dans notre méta-modèle du code comportemental.

Notre méta-modèle de code comportemental permet de représenter les informations contenues par les méta-modèle de navigation et de state flow. Cependant, nous ne représentons pas l'état d'entrée et l'état de sortie après une action, mais l'état d'entrée d'une fenêtre et la logique à appliquer après une action pour obtenir l'état de sortie. Cette différence est dû à

notre objectif qui est de migrer cette logique dans un nouveau langage et non d'analyser les différents états possibles de l'application.

8.4 Méta-modèle du code comportemental

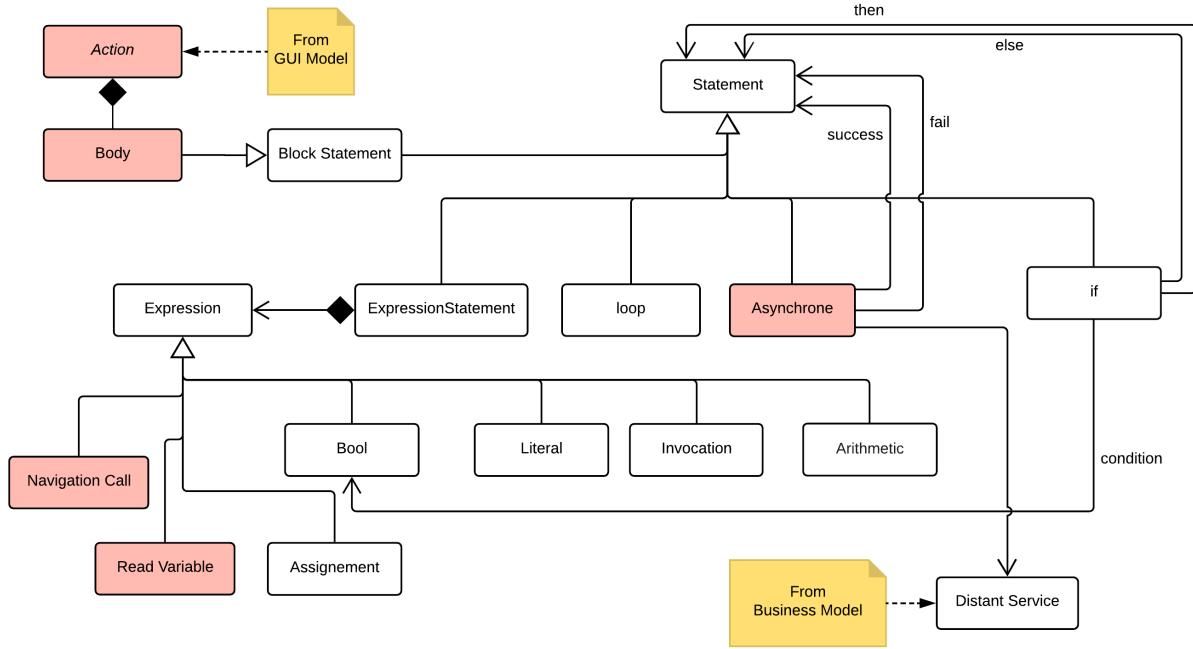


FIGURE 12 – Méta-Modèle du code comportemental

Le méta-modèle du code comportemental présenté Figure 12 représente le code lié au comportement de l'application. Il y a deux éléments principaux, Statement et Expression.

Le **Statement** est la représentation des structures de contrôle des langages de programmation. Il y a l'alternative, la boucle, la notion de bloc et un lien vers une expression.

Une **Expression** représente un morceau de code qui peut être évalué directement. Cela peut être une affectation avec la valeur de l'affectation, un littéral (directement la valeur de l'élément écrit), une expression booléenne, une expression arithmétique ou une invocation. Dans le cas d'une invocation, c'est la valeur de retour de la méthode qui est utilisée comme valeur de l'expression.

Les éléments **Action** constituent le lien vers le modèle d'interface graphique. C'est le conteneur de la logique d'un événement déclenché par une action.

Grâce à ce modèle, nous pouvons représenter la logique exécutée par un lorsqu'un événement est déclenché par une action sur un widget du modèle d'interface graphique.

9 Conclusion

Durant ce stage, j'ai continué le projet de migration que j'ai entrepris pendant mon Projet de Fin d'Études à Polytech Lille. Après une première étape d'analyse que j'avais menée durant ce stage, j'ai approfondi les recherches de l'état de l'art et créé des outils permettant de faciliter la migration des applications de Berger-Levrault.

La première étape de mon stage a été la définition précise des éléments composant une application d'une interface graphique. Une application graphique est divisée en trois parties, l'interface graphique, le code comportemental et le code métier. Ensuite, j'ai défini les différentes contraintes à respecter pour répondre aux besoins de Berger-Levrault et conçu un processus permettant d'effectuer la migration en suivant ces obligations. Puis, j'ai implémenté en partie les outils nécessaires pour effectuer la migration en suivant le processus de migration. Ainsi, j'ai pu commencer l'exportation de certains éléments des applications GWT en Angular.

La stratégie de migration est bien définie et l'implémentation du méta-modèle d'interface graphique avec les outils d'importation et d'exportation donne des résultats encourageants sur la suite du projet. Il reste encore du travail d'implémentation et de définition de méta-modèles. En effet, je n'ai pas encore conçu les méta-modèles pour les *layout* et le code métier. Une fois conçus, il faudra les implémenter tout en gardant la structure du projet cohérent.

Un autre travail doit être mené sur la validation des résultats. En effet, même si pour les cas simples que nous avons étudiés il est possible de vérifier la validité de l'exportation “à la main”, une fois l'exportation complètement implémentée, il faudra développer des outils permettant d'évaluer la complétude de mon travail.

Bibliographie

- [1] Ó. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina, “Model-driven reverse engineering of legacy graphical user interfaces,” in *Proceedings of the ieee/acm international conference on automated software engineering*, 2014, pp. 147–186.
- [2] M. M. Moore, S. Rugaber, and P. Seaver, “Knowledge-based user interface migration.” in *ICSM*, 1994, vol. 94, pp. 72–79.
- [3] J. Cloutier, S. Kpodjedo, and G. El Boussaidi, “WAVI : A reverse engineering tool for web applications,” 2016, pp. 1–3.
- [4] J. C. Silva, C. Silva, R. D. Gonçalo, J. Saraiva, and J. C. Campos, “The guisurfer tool : Towards a language independent approach to reverse engineering gui code,” in *Proceedings of the 2nd acm sigchi symposium on engineering interactive computing systems*, 2010, pp. 181–186.
- [5] V. Lelli, A. Blouin, B. Baudry, F. Coulon, and O. Beaudoux, “Automatic detection of gui design smells : The case of blob listener,” in *Proceedings of the 8th acm sigchi symposium on engineering interactive computing systems*, 2016, pp. 263–274.
- [6] S. Staiger, “Reverse engineering of graphical user interfaces using static analyses,” in *Reverse engineering, 2007. WCRE 2007. 14th working conference on*, 2007, pp. 189–198.
- [7] A. M. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping : Reverse engineering of graphical user interfaces for testing,” in *Proceedings of the 10th working conference on reverse engineering*, 2003.
- [8] H. Samir, E. Stroulia, and A. Kamel, “Swing2script : Migration of java-swing applications to ajax web applications,” in *Reverse engineering, 2007. WCRE 2007. 14th working conference on*, 2007, pp. 179–188.
- [9] E. Shah and E. Tilevich, “Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms,” in *Proceedings of the compilation of the co-located workshops on dsm’11, tmc’11, agere! 2011, aoopes’11, neat’11, & vmil’11*, 2011, pp. 255–260.
- [10] I. C. Morgado, A. Paiva, and J. P. Faria, “Reverse engineering of graphical user interfaces,” in *The sixth international conference on software engineering advances, barcelona*, 2011, pp. 293–298.

- [11] Z. Gotti and S. Mbarki, “Java swing modernization approach : Complete abstract representation based on static and dynamic analysis,” in *ICSOFT-ea*, 2016, pp. 210–219.
- [12] I. Baki and H. Sahraoui, “Multi-step learning and adaptive search for learning complex model transformations from examples,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 20, 2016.
- [13] T. Wang, S. Truptil, and F. Benaben, “An automatic model-to-model mapping and transformation methodology to serve model-based systems engineering,” *Information Systems and e-Business Management*, vol. 15, no. 2, pp. 323–376, 2017.
- [14] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel, “Model-driven engineering for software migration in a large industrial context,” in *International conference on model driven engineering languages and systems*, 2007, pp. 482–497.
- [15] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Viera, F. Melo, and J. M. Soto, “White-box modernization of legacy applications : The oracle forms case study,” *Computer Standards & Interfaces*, 2017.
- [16] W. Zhang, A. J. Berre, D. Roman, and H. A. Huru, “Migrating legacy applications to the service cloud,” in *14th conference companion on object oriented programming systems languages and applications (oopsla 2009)*, 2009, pp. 59–68.
- [17] S. Mukherjee and T. Chakrabarti, “Automatic algorithm specification to source code translation,” *Indian Journal of Computer Science and Engineering (IJCSE)*, vol. 2, no. 2, pp. 146–159, 2011.
- [18] C. Chen, S. Gao, and Z. Xing, “Mining analogical libraries in q&a discussions–incorporating relational and categorical knowledge into word embedding,” in *Software analysis, evolution, and reengineering (saner), 2016 ieee 23rd international conference on*, 2016, vol. 1, pp. 338–348.
- [19] J. Brant, D. Roberts, B. Plendl, and J. Prince, “Extreme maintenance : Transforming delphi into c,” in *Software maintenance (icsm), 2010 ieee international conference on*, 2010, pp. 1–8.
- [20] C. D. Newman, B. Bartman, M. L. Collard, and J. I. Maletic, “Simplifying the construction of source code transformations via automatic syntactic restructurings,” *Journal of Software : Evolution and Process*, vol. 29, no. 4, 2017.
- [21] R. Rolim, G. Soares, L. D’Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, “Learning syntactic program transformations from examples,” in *Proceedings of the 39th international conference on software engineering*, 2017, pp. 404–415.
- [22] M. Brambilla, P. Fraternali, and others, “The interaction flow modeling language (ifml),” 2014.
- [23] M. E. Joorabchi and A. Mesbah, “Reverse engineering iOS mobile applications,” in *Reverse engineering (wcre), 2012 19th working conference on*, 2012, pp. 177–186.
- [24] A. M. Memon, “An event-flow model of GUI-based applications for testing,” *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, Sep. 2007.
- [25] A. Mesbah, A. Van Deursen, and S. Lenselink, “Crawling ajax-based web applications through dynamic analysis of user interface state changes,” *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, p. 3, 2012.
- [26] S. I. Feldman, “A fortran to c converter,” in *ACM sigplan fortran forum*, 1990, vol. 9, pp. 21–22.
- [27] R. W. Grosse-Kunstleve, T. C. Terwilliger, N. K. Sauter, and P. D. Adams, “Automatic fortran to c++ conversion with fable,” *Source code for biology and medicine*, vol. 7, no. 1, p. 5, 2012.

- [28] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, “Using gui ripping for automated testing of android applications,” in *Proceedings of the 27th ieee/acm international conference on automated software engineering*, 2012, pp. 258–261.
- [29] P. Aho, M. Suarez, T. Kanstrén, and A. M. Memon, “Industrial adoption of automatically extracted gui models for testing.” in *EESMOD models*, 2013, pp. 49–54.

10 Annexe

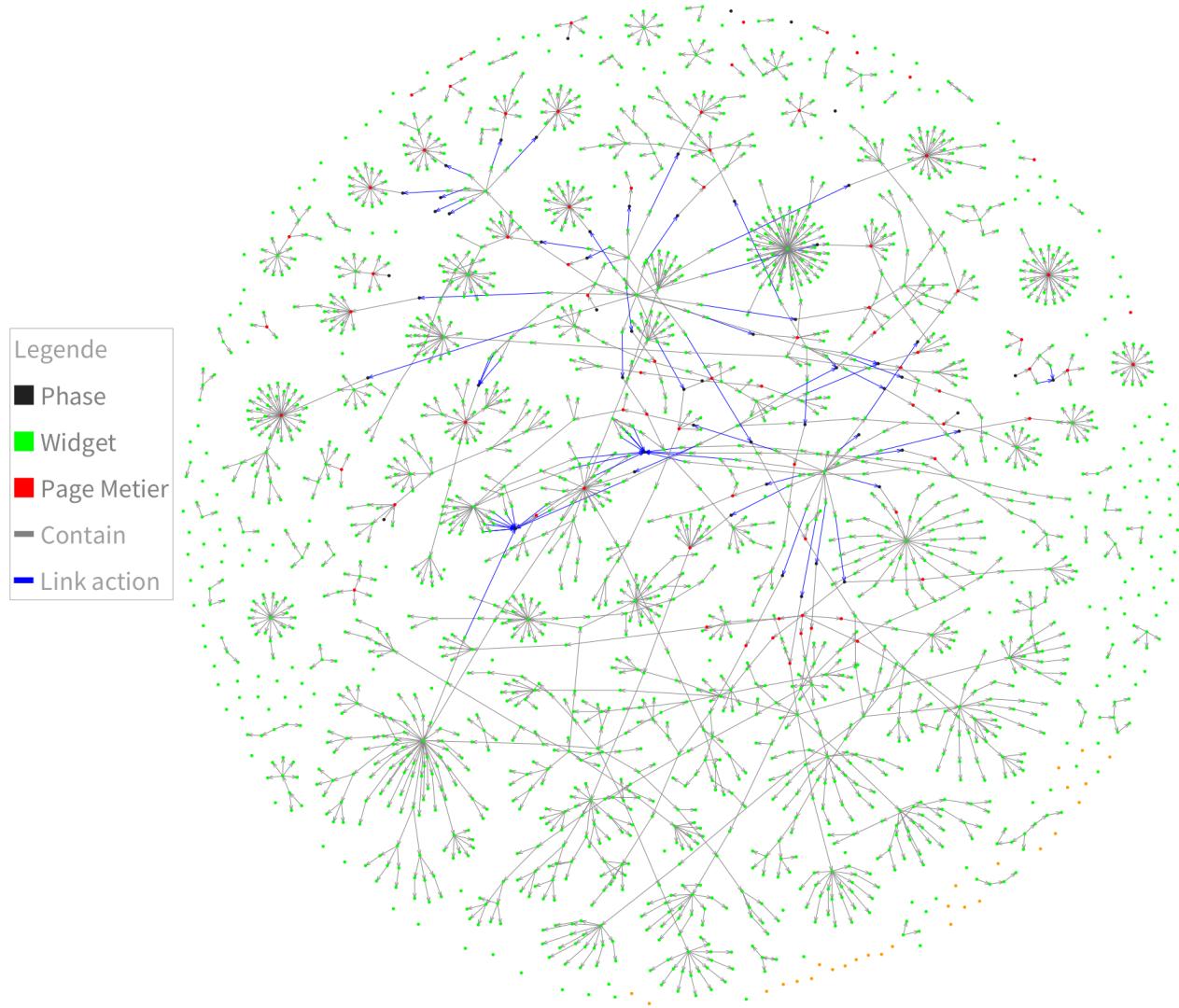


FIGURE 13 – Représentation de l’application *bac à sable* dans sa globalité

BENOÎT VERHAEGHE

Ingénieur Logiciel

@ benoit.verhaeghe59@gmail.com +33 6 58 33 53 74 40, rue Pasteur, 59260 Lezennes
Lezennes, FRANCE badetitou.github.io/ @badetitou linkedin.com/in/benoitverhaeghe
github.com/badetitou



EXPERIENCE

Ingénieur Logiciel

Berger-Levrault

⌚ Mars – Aout 2018 🗂 Montpellier, France

- Analyse de l'architecture des applications de Berger-Levrault
- Développement d'outils pour la migration d'application GWT vers Angular
- Recherche bibliographique

Etudiant-Chercheur

RMoD - Inria Lille Nord Europe

⌚ Mai – Septembre 2017 🗂 Villeneuve d'Ascq, France

- Développement en Pharo d'un "spy" (TestsUsageAnalyser)
- Ecriture d'un article scientifique pour l'IWST
- Développement d'un outil de sélection de tests (SmartTest)

Developpeur Informatique

University of Emden, Department Computer Science

⌚ Avril – Juillet 2015 🗂 Emden, Allemagne

- Développement de l'application "Factory Interface"

Animateur

Centre de loisir de La Rochette

⌚ été 2013 à 2016 🗂 La Rochette (Savoie) , France

PROJETS

SmartTest

Inria

⌚ Juillet 2017 - En Développement

- Sélection de tests automatiques (Pharo)
- Développement de différentes stratégies d'exécution de test (Pharo)

PUBLICATIONS

Conference Proceedings

- Demeyer, Serge et al. (Mar. 2018). "Evaluating the Efficiency of Continuous Testing during Test-Driven Development". In: *VST 2018 - 2nd IEEE International Workshop on Validation, Analysis and Evolution of Software Tests*. Campobasso, Italy, pp. 1–5. URL: <https://hal.inria.fr/hal-01717343>.
- Verhaeghe, Benoît et al. (Sept. 2017). "Usage of Tests in an Open-Source Community". In: *IWST'17*. Maribor, Slovenia. URL: <https://hal.inria.fr/hal-01579106>.

SUCCÈS

2ème place Innovation Award

SmartTest est arrivé deuxième à l'Innovation Award de ESUG 2017

2ème place Nuit de l'Info

J'étais responsable d'une équipe durant la Nuit de l'Info 2016.

COMPÉTENCES

Pharo JAVA/JEE Spring SQL
C R git C#

LANGUES

Français ●●●●●

Anglais ●●●●●

FORMATION

Ingénieur - Génie Informatique et Statistique

Polytech Lille

⌚ Septembre 2015 – Juillet 2018

DUT Informatique

IUT A Lille 1

⌚ Septembre 2014 – Juin 2015

HOBBIES

Club Info – Président

Polytech Lille

⌚ 2016 – 2018

Tennis de Table – Trésorier

Lezennes, France

⌚ 2016 – 2018

Recherche Historique – Secrétaire

CRHL Lezennes, France

⌚ 2014 – Février 2018

FIGURE 14 – CV