

Benoît Verhaeghe

Migration d'application GWT vers Angular



Tuteurs entreprise : M. Deruelle, M. Seriai

Tuteur 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 GUI Décomposition	5
2.1 Interface utilisateur	5
2.2 Code comportemental	5
2.3 Code métier	5
3 Description du problème	6
3.1 Contraintes	6
3.2 Comparaison de GWT et Angular	7
3.3 Stratégies de migration	7
4 Mise en place de la migration par via les modèles	9
4.1 Processus de migration	9
4.2 Méta-modèle d'interface utilisateur	10
4.3 Méta-modèle du code comportemental	11
4.4 Implémentation du processus	12
4.4.1 Meta-modèle	12
4.4.2 Importation	13
4.4.3 Exportation	14
5 Etat de l'art	15
5.1 Technique de migration	15
5.1.1 Rétro-Ingénierie d'interface graphique	15
5.1.2 Transformation de modèle vers modèle	16
5.1.3 Transformation de modèle vers texte	17
5.1.4 Migration de librairie	17
5.1.5 Migration de langage	18
5.2 Positionnement sur la migration via les modèles	19
5.2.1 Méta-modèle d'interface utilisateur	19
5.2.2 Méta-modèle de navigation et de state flow	20
5.2.3 KDM	21
5.2.4 IFML	23
6 Résultats et Discussion	25
6.1 Résultats	25
6.1.1 Rétro-Ingénierie	25
6.1.2 Exportation en Angular	25
6.2 Visualisation	27
6.3 Discussion	28

7	Travaux futurs	29
7.1	Amélioration des modèles	29
7.2	Outil de validation	29
7.3	Complétude du travail	30
7.4	Exportation du Core	30
8	Conclusion	32
	Bibliographie	33
9	Annexe	35

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'Étude 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 framework¹ graphique est imposé par l'arrêt du développement de GWT par Google et le problème de compatibilité arrière entre Angular 6 et Angular 1 (AngularJS). 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

Les applications front-end de Berger-Levrault sont développées en Java en utilisant le framework GWT de Google. Dans l'optique d'homogénéiser le visuel de leurs applications, Berger-Levrault a étendu ce framework. Cette extension s'appelle **BLCore**. La Figure 1 représente les différentes couches de framework utilisé par une application de Berger-Levrault. Les applications utilisent et/ou étendent **BLCore**, qui lui-même utilise et/ou étend **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**.

Les applications de Berger-Levrault sont complexes. Ce sont les plus importantes applications GWT en terme de ligne de code et de classes dans le monde. Elles définissent plusieurs centaines de pages web. Bien qu'une migration complète de l'application en réécrivant l'ensemble du code est possible, c'est 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 pourrait refuser une telle solution. Une alternative pour contourner ce problème serait de créer des outils facilitant la migration et en migrer une partie de

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).

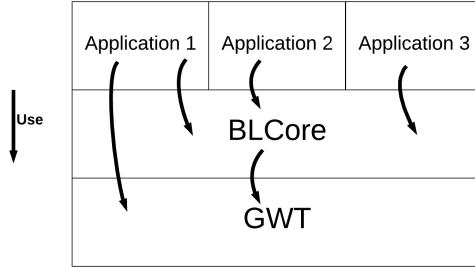


FIGURE 1 – Structure application

l’application. Les développeurs pourront alors effectuer la fin de la migration rapidement et seront formés sur le nouveau langage et sur l’application en le pratiquant assisté de nos outils.

La complexité de la migration des applications de Berger-Levrault réside dans leurs tailles mais aussi et surtout dans le changement de langage. En effet, les applications sont développées intégralement en Java tout comme le framework GWT. Or, pour utiliser Angular 6, le programme doit être écrit en TypeScript. La question qui se pose est de savoir quelle couche de la Figure 1 nous devons migrer et comment ?

En plus des difficultés techniques inherentes 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 devra, entre autre, conserver l’architecture sur laquelle se base les applications de Berger-Levrault et ne pas perturber les clients de Berger-Levrault du point de vue visuel des applications et comportemental.

Mon 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, j’ai dans un premier temps étudié la structure d’une application de Berger-Levrault. Puis, j’ai défini avec un expert Angular l’architecture attendu pour les applications post-migration. Ensuite, j’ai définie une stratégie pour faire la migration en m’inspirant d’une étude de l’état de l’art que j’ai menée. Enfin, j’ai commencé le développement d’une suite d’outil permettant de mettre en application la stratégie définie et l’évaluer.

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 depuis BLCore. Bien que plus petite que les applications en production, elle contient tout de même plusieurs centaines de classes. Cependant, j’ai régulièrement vérifier que mon travail pouvait s’appliquer sur les projets plus important de Berger-Levrault.

2 GUI Décomposition

Les applications que nous devons migrer ont des interfaces graphiques. Avant de créer l'outil de migration, il faut comprendre ce qu'est une telle interface et comment nous pouvons la diviser. Diviser un problème en petits sous-problèmes est une méthode efficace pour résoudre des problèmes complexes. Nous avons identifié trois parties dans une interface graphique :

- L'interface utilisateur
- Le code de l'entreprise
- Le code comportemental

2.1 Interface utilisateur

L'interface utilisateur est la partie visible. Cet élément représente l'interface de l'application. Elle comprend les composants de l'interface. L'interface utilisateur ne contient pas la visualisation exacte d'un composant, mais elle peut préciser certaines caractéristiques inhérentes au composant, 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. Dans le cas où une application est composée de plusieurs fenêtres (ou de pages web pour une application web), l'interface utilisateur contient toutes les fenêtres.

2.2 Code comportemental

Le code comportemental est la partie *executable* de l'application. Cela correspond à la logique de l'application. Il peut avoir deux manifestations du code comportemental. Il peut être exécuté soit par une action de l'utilisateur sur un composant d'interface (comme un clic) ou par le système lui-même. Comme un langage de programmation “*classique*”, le code métier contient des structures de contrôle (*i.e.* boucle et alternative). Lié à l'interface utilisateur, le code comportemental définit la logique de l'interface utilisateur. Cependant, le code comportemental n'exprime pas la logique de l'application. Cette partie est dédiée au code comportemental.

2.3 Code métier

Le code métier définit les informations spécifiques d'une application. Il est composé par les règles générales de l'application (comment calculer les taxes ?), le lien services distants (quel serveur mon code métier doit demander), les données de l'application (quelle base de données ? quel type de *object*). Le code métier n'est donc pas directement lié à l'interface utilisateur.

3 Description du problème

Dans le contexte de l'évolution des applications de Berger-Levrault, l'entreprise a estimé la transformation du code à X jours 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 réduit le temps nécessaire pour la transformation des programmes.

3.1 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 que j'ai produite doit respecter les contraintes suivantes :

- *Depuis GWT (BLCore)*. La solution doit au moins fonctionner dans le cas de Berger-Levrault. Elle peut être plus général mais ne doit pas faire de concession sur le résultat final.
- *Vers Angular*. Dans le cas d'une automatisation ou semi-automatisation du processus de migration, celle-ci doit s'achever par la génération de code Angular. La solution peut contenir des structures facilitant son utilisation pour d'autre langages cible mais pas au détriment du projet fixé avec Berger-Levrault.
- *Approche modulaire*. La migration doit être divisée en petites étapes. Cela permet de facilement remplacer une étape ou de l'étendre sans introduire d'instabilité. Cette contrainte est essentielle pour les entreprises qui désirent avoir un contrôle fin du processus de migration. L'approche modulaire permet entre autres aux entreprises de modifier l'implémentation de la stratégie pour respecter leurs contraintes spécifiques.
- *Préservation de l'architecture*. Après la migration, nous devons retrouver la même architecture entre les différents composants de l'interface graphique (*c.-à-d.* un bouton qui appartenait à un panel dans l'application source appartiendra au même panel dans l'application cible). Cette contrainte permet de faciliter le travail de compréhension de l'application cible par les développeurs. En effet, ils vont retrouver la même architecture qu'ils avaient dans l'application source.
- *Préservation du visuel*. Il ne doit pas y avoir de différence visuelle entre l'application source et l'application cible. Cette contrainte est particulièrement importante pour les logiciels commerciaux. En effet, les utilisateurs de l'application ne doivent pas être perturbés par la migration.
- *Automatique*. La solution apportée doit être automatique. Les utilisateurs de l'outil ne devrait pas intervenir pendant le processus de migration ou très peu. Ainsi, l'outil peut être utilisé avec un minimum de connaissance préalable.
- *Amélioration de la qualité*. La migration doit permettre de traiter les possibles déviances du programme source. Par exemple, dans le cas de Berger-Levrault, l'outil de migration doit être capable de gérer les éléments utilisés par l'application à migrer et provenant du framework GWT. Cette exemple d'utilisation du framework GWT par l'Application 1 est représenté Figure 1.

Une dernière contrainte inhérent aux entreprises est la possibilité pour les équipes de développement de continuer la maintenance des applications pendant le développement de la stratégie de migration et la migration elle-même.

2. MLOC : Million lines of code

3.2 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, le langage de programmation source et cible ont deux architectures différentes. Les différences sont syntaxicales, semanticales et architecturales. Pour la migration d’application GWT vers Angular, les fichier *.java* sont séparés 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 est nécessaire pour représenter une page web. L’ensemble de la page web peut donc être contenu dans ce fichier, il reste toutefois possible de créer d’autres fichiers pour séparer les différents éléments de la page web. Les fichiers java contiennent les différents 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 executer lorsque l’action est réalisée. En Angular, on crée une hiérarchie de fichier correspondant à un *sous-projet* pour chaque page web. Ce sous-projet 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 les widgets du fichier HTML. On a donc la séparation d’un fichier java pour GWT vers deux fichiers HTML et TypeScript en Angular pour représenter les widgets et le code qui leur sont 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, cependant, c’est un fichier CSS que l’on doit créer par sous-projet 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 devra utiliser pour y accéder. En Angular, il y a deux fichiers de configuration générale. Le premier, *module*, explicite les différentes pages web accessible dans l’application ainsi que les services distant et les composants graphiques (widgets) utilisable dans l’application. Le second, de *routing*, définit pour les différentes pages web de l’application leurs chemins d’accès.

3.3 Stratégies de migration

Il existe plusieurs manières d’effectuer la migration d’une application. Toutes les solutions doivent respecter les contraintes définies Section 3.1.

- *Migration manuelle*. Cette stratégie correspond au re-développement 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

```

Parser: ExpressionParser
>>>`a` `op{beToken}` `b`<<<
->
>>>`a` `b` `op`<<<

```

$$(3 + 4) * (5 - 2) ^ 3 \rightarrow 3 4 + 5 2 - 3 ^ *$$

FIGURE 2 – Exemple de règle de transformation

re-concevoir 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 [1]–[3]. 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. La Figure 2 montre un exemple de règle de transformation. Dans ce cas, elle permet de changer la position des opérateurs dans une expression mathématique source. L’opérateur est maintenant en suffixe de l’expression. Il est possible que la migration ne soit pas complète. 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 pour effectuer. 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 compléter la migration.

4 Mise en place de la migration par via les modèles

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

Comme vu Section 3.3, seule la migration en utilisant les modèles nous permet de respecter toutes les contraintes. Ce type de migration nous impose la conception de métamodèles.

Nous allons présenter dans cette partie le processus de migration que nous avons conçu, puis le métamodèle d'interface utilisateur utilisé dans ce processus, et enfin expliquer l'implémentation de cette stratégie.

4.1 Processus de migration

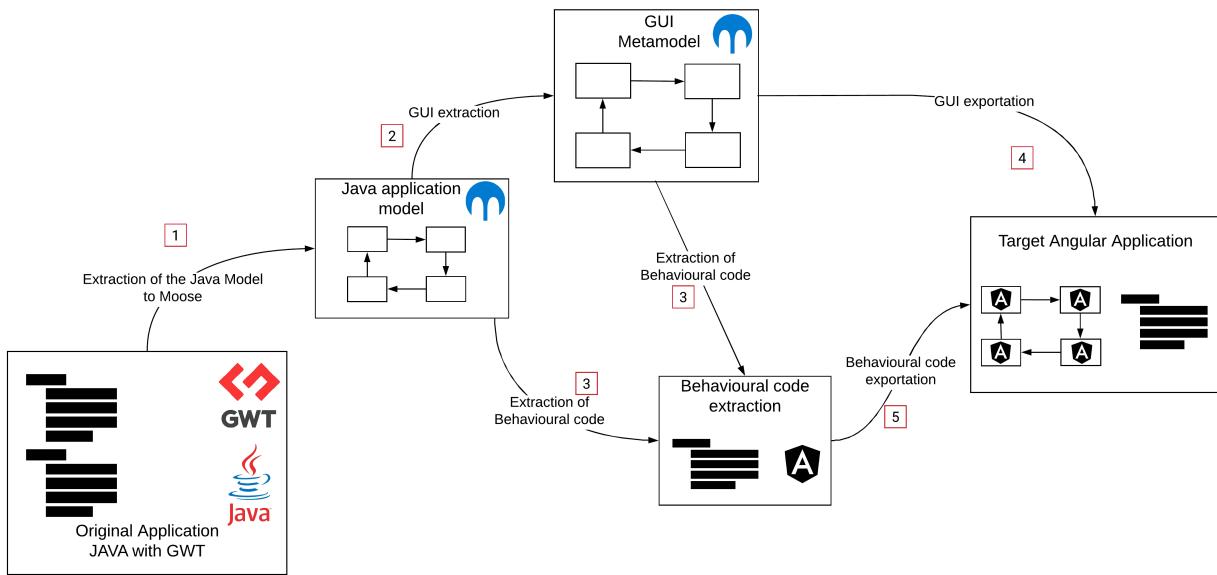


FIGURE 3 – Schema processus de migration

À partir de l'état de l'art et des contraintes que nous avons explicités, nous avons conçu une stratégie pour effectuer la migration. Le processus que l'on a représenté Figure 3 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 métamodèle permettant de représenter une application écrite en java.
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 4.2.
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. Par exemple, si un clique sur un bouton agit sur un texte dans l’interface graphique. L’extraction du code comportemental permet de définir que pour le bouton, définit dans le modèle UI, lorsqu’un clique est effectué, on effectue un certain nombre d’actions dont une sur le texte, lui aussi définit 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 à 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 nécessaire au fonctionnement de l’application cible ainsi que la création des fichiers de configuration inhérent 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 quatrième.

4.2 Méta-modèle d’interface utilisateur

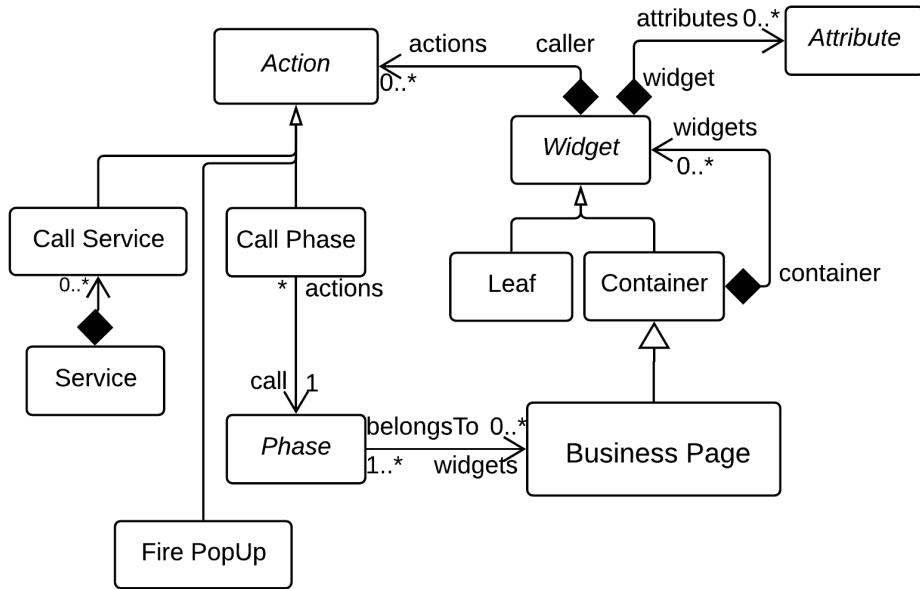


FIGURE 4 – Méta-Modèle d’un application de Berger-Levrault

Afin de représenter une interface utilisateur, nous avons conçu le méta-model proposé Figure 4. Dans la suite de cette partie, nous présentons les différentes entités du méta-modèle.

La **Phase** représente le conteneur principal d’une page interface utilisateur. Cela peut correspondre à une *fenêtre* d’une application du bureau, une page web, ou dans notre cas d’un onglet d’une page web. Une Phase peut contenir plusieurs Business page. Elle peut aussi être appelée par un widget grâce à une Call Phase Action. Lorsqu’une Phase est appelée, l’interface change pour afficher la Phase. Dans le cas d’une application de bureau, l’interface change ou une nouvelle fenêtre est ouverte avec l’interface de la Phase. Avec une application web, l’appelle 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.

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 un autre widget dans l'interface. Le **Containers** peut contenir un autre widget. Ce dernier permet de séparer les widgets en fonction de leur place dans l'organisation de la page web représenté.

Les **Attributes** représentent les informations appartenant à un Widget et peuvent changer son aspect visuel ou son comportement. Les attributs communs sont la hauteur et la largeur pour définir précisément la dimension d'un widget. Il y a aussi des attributs pour contenir des attributs tels que le texte contenu par un widget. Par exemple, un widget représentant un bouton peut avoir un attribut *text* explicite le texte du bouton. Un attribut peut changer le comportement, ce pourrait être le cas d'un 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. Cet attribut définit la disposition de ses enfants et potentiellement sa propre dimension pour respecter la mise en pages.

Les **Actions** sont propres aux Widgets. Elles représentent des actions qui peuvent être exécutées dans une interface graphique. **Call Service** représente un appel à un service distant tel Internet. **Fire PopUp** est l'action qui affiche un PopUp sur l'écran. Le PopUp 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 à la fonctionnalité distante que l'application peut appeler à partir de son interface graphique. Dans un contexte Web, il peut s'agir du côté serveur de l'application.

4.3 Méta-modèle du code comportemental

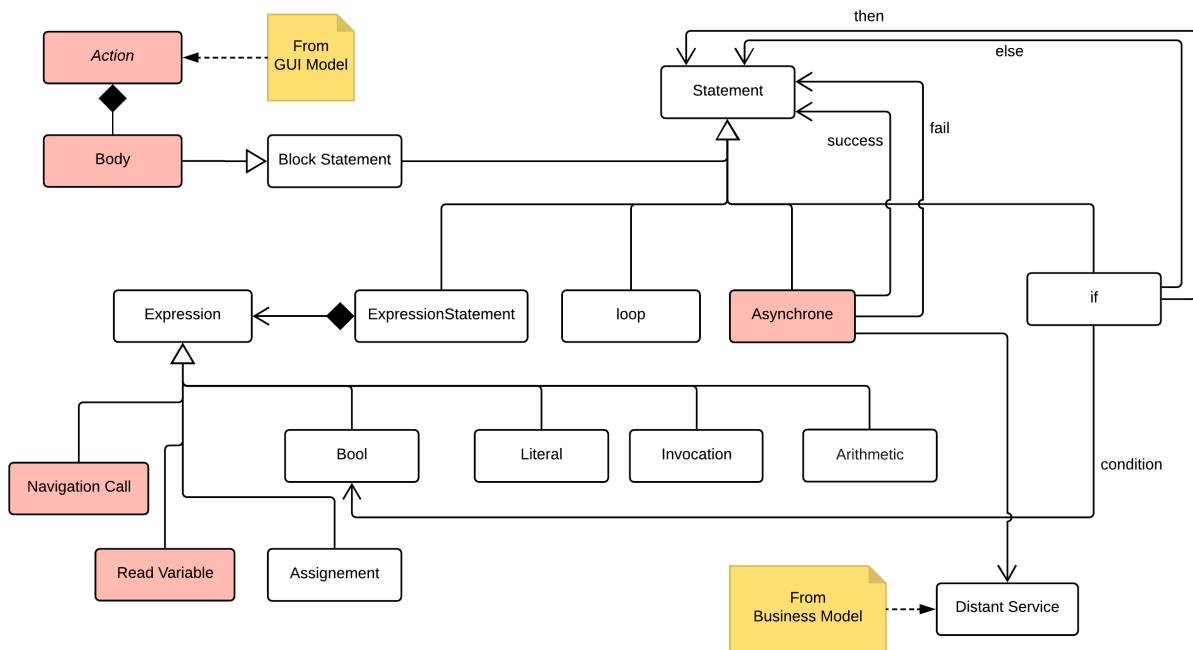


FIGURE 5 – Méta-Modèle d'un application de Berger-Levrault

Le méta-modèle du code comportemental présenté Figure 5 contient le code lié au com-

portement 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 utilisé 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 exécuté par un lorsqu'un événement est déclenché par une action sur un widget du modèle d'interface graphique.

4.4 Implémentation du processus

Pour tester la stratégie, nous avons implémenté un outil qui suit le processus de migration. L'outil a été implémenté en Pharo³ et nous avons utilisé la plateforme Moose⁴.

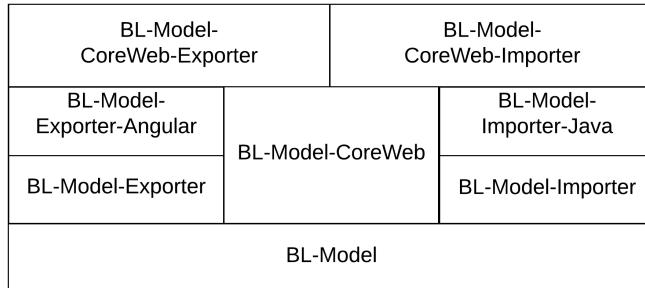


FIGURE 6 – Implémentation de l'outil

Le Figure 6 présente la logique d'implémentation. Le bloc principal est *BL-Model*. Ce bloc contient l'implémentation du méta-modèle GUI. 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*), un importateur abstrait et 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 “CoreWeb”, alors que la stratégie de migration ne dépend 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.

4.4.1 Meta-modèle

Pour implémenter le méta-modèle GUI (voir Figure 4), nous avons utilisé la dernière version de Famix dans Moose. Cet outil est pratique car il fournit un générateur de méta-modèle, le builder. Pour définir les entités du méta-modèles, il faut les nommer dans la méthode `defineClasses` du builder. Pour les relations entre les entités, nous pouvons utiliser le format

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

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

UML. Ainsi une relation “*oneToMany*” entre deux entités se définit de la manière suivante : `entity1 -* entity2`.

Afin de tester la stratégie sur l’application de Berger-Levrault, nous avons créé des types spécifiques de Widget pour Berger-Levrault. Des exemple de ces widgets sont le *SplitButton*, *RichTextArea* ou *Switch*. Ces éléments n’appartiennent pas au modèle GUI d’origine et, combiné avec le cadre que nous avons créé, ils rendent l’implémentation de l’outil plus modulaire.

4.4.2 Importation

La création des modèles représentant l’interface graphique est divisée en trois étapes comme présenté Section 4.1. Dans le cas de Berger-Levrault, nous avons implémenté la stratégie en Pharo avec Moose.

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 re-concevoir un modèle pré-existant. 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 existait. Les outils sont *verveineJ*⁵ et *jdt2Famix*⁶. Ces deux derniers permettent de créer depuis le code source un fichier *mse*. Le fichier *mse* peut ensuite être importé dans la plateforme Moose. Pour le cas de 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.

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 Page. 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*. Une fois les classes trouvées, nous avons recherché les appels des constructeurs des classes. Puis, en faisant le lien entre le constructeur et la phase qui “*ajoute*” la business page à leur contenu, nous avons détecté les liens d’appartenances entre les pages métiers 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 instancié et utilisé 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 ajouté.

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 effectué depuis ces variables java. Les appels aux méthodes

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

6. *jdt2famix* : <https://github.com/feenkcom/jdt2famix>

“*addActionHandler*” sont transformés en action tandis que les appels aux méthodes “*setX*” ont été transformé en attribut.

4.4.3 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 source de l’application cible.

La première étape de l’implémentation de l’exportation est l’utilisation d’un patron de conception “*visiteur*”. Ce dernier est ajouté au modèle d’interface graphique, aux phases et business pages.

La visite du modèle GUI va créer la hiérarchie de l’application cible ainsi que les fichiers de configuration. Ensuite, l’exportation visite toutes les phases. Pour chacune des phases, considéré comme des sous-projets en Angular dans l’architecture de l’application cible que nous avons défini, le visiteur génère les fichiers de configurations. Puis, pour chaque business page, le visiteur va générer un fichier HTML et un fichier TypeScript. Pour le fichier html, le visiteur construit le DOM à partir des widgets contenu 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.

5 Etat 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 5.1 présente les différentes techniques qui sont utilisées pour faire effectuer la migration d'application. Dans un second temps, la Section 5.2 positionnera notre travail par rapport à ceux proposés dans la littérature et utilisant la même technique de migration.

5.1 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.

5.1.1 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. Le choix de représentation des interfaces graphiques est discuté Section 5.2.

Les auteurs utilisent trois techniques pour instancier leurs métamodèles.

Statique. La stratégie statique consiste à analyser le code source de l'application comme on analyserait du texte. En fonction du langage source, l'analyse statique peut être plus ou moins facile à mettre en place.

Dans le cas de [4], les auteurs ont pu analyser directement les fichiers HTML, CSS et JavaScript. Ceci leur permet de construire un arbre syntaxique du code source du site web et d'extraire les différents widgets depuis le fichier HTML.

D'autres auteurs comme [5]–[7] utilisent un outil qui va analyser du code source provenant d'un langage non destiné au web mais permettant de décrire une interface graphique. Leur cas d'étude sont des applications de bureau ayant une interface graphique. Ces logiciels vont chercher les définitions des widgets dans le code source. Une fois les créations des widgets trouvées dans le code source, le logiciel va analyser les méthodes invoquées ou invoquant les widgets afin de découvrir les relations entre les widgets et leurs attributs.

Sánchez Ramón *et al.* [8] ont développé une solution permettant d'extraire depuis un ancien logiciel son interface graphique. Le cas d'étude des auteurs est une application source ayant été créée avec Oracle Forms. Contrairement aux cas d'étude 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 de l'auteur consiste à déterminer la hiérarchie des widgets depuis ces informations.

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

Les auteurs [9]–[12] 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 au spécificité des applications web comme dans notre cas.

Mixe. L'objectif de la stratégie mixe est d'utiliser la stratégie statique et la stratégie dynamique. Gotti *et al.* [13] utilise une stratégie mixe 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 va exécuter les différentes actions possibles sur tout les widgets et analyser les modifications potentielles sur l'interface.

5.1.2 Transformation de modèle vers modèle

La *transformation de modèle vers modèle* traite de la modification d'un modèle source vers un modèle cible.

L'article de Baki *et al.* [14] 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 sortie le SQL définit 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. A 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 analyser si le modèle en sortie ressemble à celui explicité pour tous les exemples. Enfin, l'algorithme est appliqué sur de vrais données. Ce travail peut être utilisé dans mon projet. En effet, je peux aussi effectuer la migration en utilisant un modèle de l'application source et un modèle de l'application cible.

Wang *et al.* [15] ont créé une méthodologie et un outil permettant d'automatiquement faire 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. Une condition d'utilisation contraignante décrite par les auteurs est la nécessité d'avoir un méta-méta-modèle pour tous les méta-modèles intermédiaires. Les auteurs ont implémenté un méta-méta-modèle dans leur outil. Dans mon travail, une solution pour effectuer la migration serait d'utiliser des méta-modèles. Le premier modèle proviendrait de l'application source, le second modèle respecterait le méta-modèle de destination. J'ai donc la même problématique que les auteurs de passage d'un modèle à un autre. En définissant un méta-méta-modèle que respecterai le méta-modèle de départ de l'application source et un méta-modèle de destination, la méthodologie proposée par les auteurs devrait pouvoir résoudre, totalement ou partiellement, mon problème de migration.

Fleurey *et al.* [16] et Garcés *et al.* [17] 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 7. 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 la structure des données, les actions et algorithmes, l'interface graphique et la navigation dans l'application.

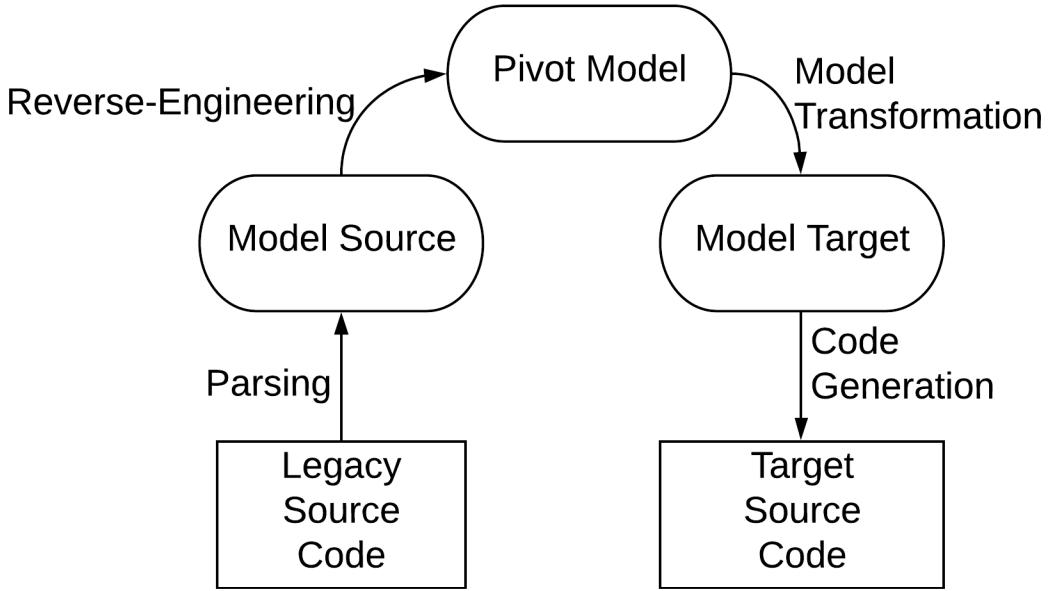


FIGURE 7 – Migration selon le *fer à cheval*

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, je vais devoir faire la migration d’une application et je vais devoir conserver structure de données, actions, interface graphique et navigation dans l’application. Mon travail peut donc s’inspirer de celui proposé par les auteurs si nous souhaitons utiliser les modèles pour effectuer la migration.

5.1.3 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.* [18] 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. Je pourrai réutiliser ce travail si je passe par un modèle pour la migration. En effet, le fichier XML pris en entrée de l’outil des développeurs peut être assimilé à un modèle suivant un méta-modèle (défini dans l’article). Dans le cas où nous utilisons un modèle dans le cadre de la migration des applications de Berger-Levrault. Nous pourrions aussi être amené à utiliser un système de règle pour faire la migration du modèle au code source.

5.1.4 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.* [19] 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 langages et leurs libraries ainsi que des équivalences entre librairies de langage différent. Pour la migration de Java/GWT vers Angular, il est possible que nous ayons besoin de changer de librairie (qui n'est pas BLCore). Plutôt que de récrire 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 mettant faire correspondre les anciennes librairies utilisées par les applications de Berger-Levrault avec d'autre compatible utilisable avec Angular.

5.1.5 Migration de langage

La *migration de langage* traite de la transformation du code source directement (*i.e.* sans passer par un modèle). Pour cela, les auteurs créent des “règles” permettant de modifier le code source.

Brant *et al.* [1] ont écrit un compilateur utilisant un outil nommé SmaCC. SmaCC est un générateur d'analyseur pour Smalltalk. Ils ont aussi utilisé le SmaCC Transformation Toolkit qui permet de définir des règles de transformations qui seront utilisées par SmaCC. Ainsi, les auteurs sont parvenus à migrer une application Delphi de 1,5 million de lignes de code en C#. Comme les auteurs, je veux effectuer la migration du code source d'une application. Mon cas se différencie par les langages source et cible. Ce travail peut me servir si nous souhaitons effectuer la migration sans passer par un modèle intermédiaire.

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 règle de transformation. Pour cela, l'outil “normalise” le code source en entré et essayer 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, je dois 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 un modèle.

Rolim *et al.* [21] ont créé un outil qui apprend des règles de transformation de programme à partir d'exemple. Pour cela, les auteurs ont défini un DSL⁷ permettant d'exprimer les modifications faites sur l'AST⁸ 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. De plus, on peut imaginer un outil qui apprendrai au fur et à mesure des développement des développeur et qui les conseillerez dans un second temps sur d'autre développement avec les même problématiques déjà résolu. Ou encore, l'outil pourrait servir de “guide” pour les nouveaux développeurs participants à la migration ou au développement courant des applications.

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

8. AST : Arbre Syntaxique Abstrait

5.2 Positionnement sur la migration via les modèles

Comme décrit Section 4.1, nous avons décidé d'effectuer la migration en utilisant les modèles. Nous avons utilisé une stratégie selon le *fer à cheval*.

Afin d'appliquer cette stratégie de migration, nous avons dû définir plusieurs méta-modèles décrit Section 4.2 et Section 4.3. Dans cette Section, nous allons nous positionner par rapport aux modèles proposées dans les autres papiers traitant de la représentation d'une application contenant une interface graphique. Puis nous présenterons et discuterons les modèles KDM et IFML qui ont été proposées par l'OMG⁹ (<https://www.omg.org/spec/KDM/>) pour représenter respectivement des applications et des interfaces graphiques.

5.2.1 Méta-modèle d'interface utilisateur

Nous avons cherché les différences et points commun entre notre méta-modèle d'interface utilisateur et ceux proposé dans la littérature.

[13] ont proposé un méta-modèle inspiré du modèle KDM (voir la Section 5.2.3). Le modèle a les principales entités que nous avons également définies. On retrouve le patron de conception composite pour représenter le DOM d'une interface graphique. Leur widget s'appelle *Component* et comme nous ils ont des propriétés. Ils ont également implémenté une propriété *Event* pour le widget. Mais ils inversent le nom *event* et *action* comparés à notre solution.

[16] n'ont pas décrit le méta-modèle de l'interface graphique directement, mais nous avons extrait des informations de leur modèle de navigation (voir 5.2.2). Ils ont au moins deux éléments dans leur modèle d'interface graphique, *Window* et *GraphicElement*. *Window* semble correspondre à notre entité *Phase*. On peut supposer que le *GraphicElement* est un widget. 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 au notre.

Le méta-modèle graphique de [8] est très similaire au notre. Il y a les entités *Widget* et *Window* qui correspondent respectivement à notre *Widget* et *Phase*. Leurs widgets ont une position x et y et la largeur et hauteur en tant que propriétés. Ceci est similaire au lien entre *Widget* et *Attribute* dans notre méta-modèle. Il y a aussi le patron de conception composite pour représenter le DOM.

[12] utilisent un méta-modèle graphique mais ne le décrit 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 [17] diffère beaucoup du notre. Il y a les attributs, les événements, les windows (qui sont comme les phases) mais il n'y a pas de widget. Cette absence s'explique par la différence dans la technologie source. Les auteurs ont travaillé sur un projet utilisant des Oracle Forms. Cette technologie 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. Nous pouvons encore remarquer qu'ils utilisent une entité *Event* pour représenter l'action de l'utilisateur avec le interface utilisateur.

[9] représentent une interface utilisateur graphique avec seulement deux entités dans leur méta-modèle d'interface graphique. Une GUIWindow similaire à notre phase qui est constituée d'un ensemble de widgets. Ces widgets peuvent avoir des propriétés et toutes les propriétés

9. OMG : Object Management Group

ont une valeur associée. Les auteurs ont défini une interface utilisateur comme ensemble de widget 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 que nous avons proposés car, dans notre conception si la valeur d'une propriété change, nous sommes toujours dans la même interface utilisateur mais un code de comportement a été exécuté.

[10] ont travaillé sur la migration de l'application Java-Swing vers l'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 peut déclencher un événement lorsqu'un *Input* GUI est exécutée. Dans notre travail, nous avons le même système pour l'événement, mais l'*Input* est englobé sous le concept d'action.

[11] utilisent une arbre pour représenter l'interface graphique. La racine de l'arbre est une *Frame* ce qui correspond à notre phase. La racine contient des *Composants* avec leurs propriétés. L'entité *Composant* est similaire à notre entité Widget.

[22] représentent une interface utilisateur avec un ensemble d'éléments d'interface utilisateur. Ces éléments sont notre définition d'un widget sans le patron de conception container. Pour chaque élément d'interface utilisateur, l'outil des auteurs peut gérer la détection de plusieurs attributs et de l'événement.

[23] utilisent un modèle d'interface graphique pour représenter l'état d'une application (voir 5.2.2). Comme nous, ils ont les widgets avec des propriétés. L'auteur a fait la différence entre *Widget* et *Container*, il est similaire à l'utilisation du patron de conception container que nous utilisons. Avec la notion de *Container*, l'auteur est capable de représenter un DOM.

[24] n'ont pas présenté directement le méta-modèle de l'interface utilisateur qu'ils utilisent. Cependant, ils expliquent qu'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 nos entités Phase.

5.2.2 Méta-modèle de navigation et de state flow

On retrouve dans la littérature deux autres méta-modèles très présent. 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 différents. 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.* [12] et Fleurey *et al.* [16] ont utilisé un méta-modèle de navigation pour représenter les liens entre les différentes interfaces utilisateurs 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'arrivé. Le méta-modèle possède tout de même une entité appelé *Operation*, mais qui ne semble pas impliqué dans la navigation.

Les informations de ces méta-modèles sont complètement contenu 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, nous un nouvel état est généré.

[5], [23]–[26] ont tous utilisé un méta-modèle de state flow afin de représenter les différentes transition entre les interfaces graphiques. Pour définir un état, leurs outils va analyser les valeurs des propriétés des widgets visibles sur un écran. Une fois une action exécuté, l'outil va détecter 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.* [22] ont décidé d'effectuer une comparaison d'image. Après chaque action sur un widget, exécuté par un outil, il prenne une image de l'application et la compare 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 contenu 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 possible de l'application.

5.2.3 KDM

L'OMG a défini la norme KDM¹⁰ pour prendre en charge l'évolution des logiciels. Le standard est un méta-modèle pour représenter un logiciel dans un haut niveau d'abstraction.

Le Figure 8 présente un aperçu de l'architecture KDM. Il est divisé 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. Il est similaire à notre méta-modèle du code comportemental. Le paquetage Data représente les données utilisé dans l'application pouvant provenir de services distant 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 9 est le diagramme de classes UIResources. Il définit de nombreuses entités pour l'abstraction d'une interface utilisateur.

UIResource peut être défini comme UIDisplay, UITextField ou UIEvent. Un UITextField est utilisé pour représenter des formulaires, des champs de texte, des panneaux, etc. Il est similaire au Widget de notre méta-modèle d'interface utilisateur. Nous pouvons également détecter le patron de conception *composite* avec UIResource et AbstractUIElement. Cela permet la représentation de l'architecture d'interface utilisateur possible.

10. KDM : Knowledge Discovery Metamodel

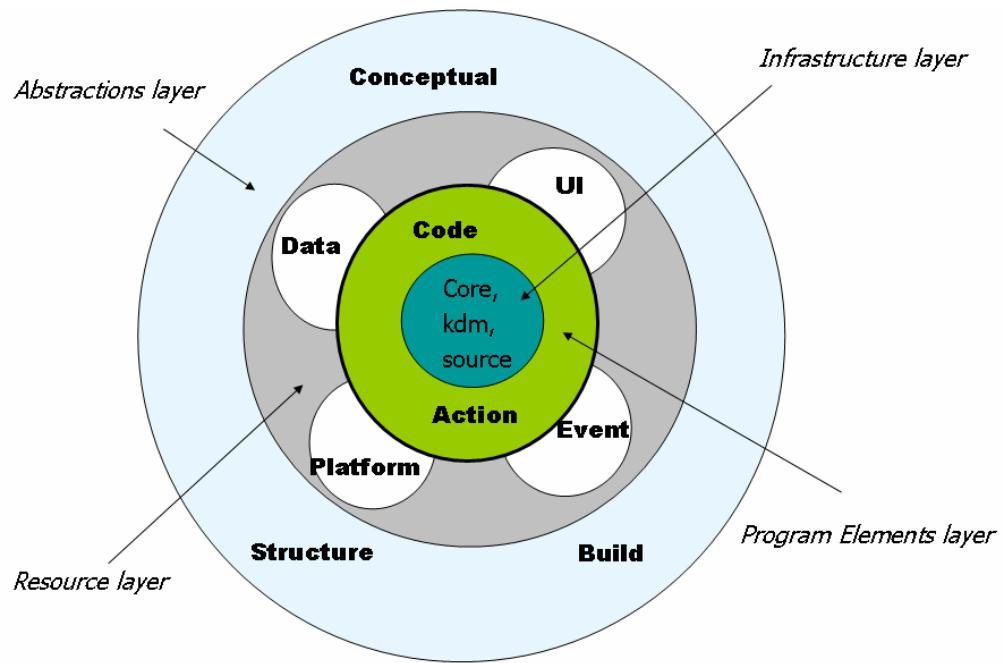


FIGURE 8 – Architecture KDM

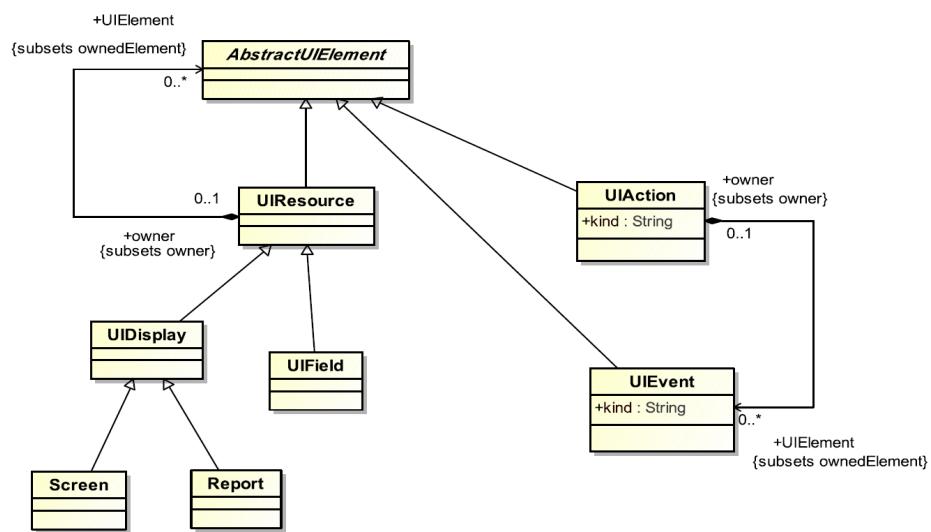


FIGURE 9 – KDM Diagramme de Classe UIResources

UIDisplay peut être un Screen ou un Report. Screen représente une fenêtre pour un logiciel de bureau ou une page Web. Report représente un élément qui sera imprimé. Cette différence entre Report et Screen n'est pas représentée dans nos méta-modèles.

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

5.2.4 IFML

Brambilla *et al.* [27] a défini le langage IFML (Interaction Flow Modeling Language) pour représenter une l'interface graphique d'une application. Les méta-modèles ont été défini avec l'OMG (<http://www.ifml.org/>). L'IFML a pour but de fournir un système pour décrire la partie visuel 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 d'une 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.

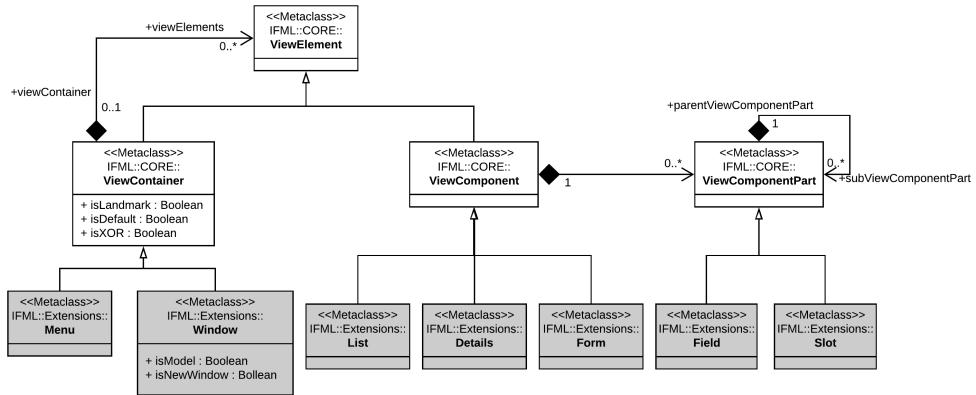


FIGURE 10 – IFML View Elements

La Figure 10 présente le méta-modèle *View Elements* proposé par IFML. Ce méta-modèle a le même objectif que le modèle d'interface graphique que nous avons conçu. Il représente le partie visible de l'interface utilisateur. Nous remarquons qu'ils sont légèrement différents. Les deux utilisent un patron de conception container et ont donc la notion de conteneur et de composant, respectivement conteneur et feuille dans notre méta-modèle. Le méta-modèle IFML a introduit 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 des éléments tels qu'une liste ou un formulaire en tant que composant. Dans notre cas, une liste

est représentée comme un conteneur, donc nous n'avons pas besoin d'un ComponentPart parce que nous pouvons utiliser le patron de conception container pour représenter les sous-éléments d'un composant.

6 Résultats et Discussion

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

Dans un premier temps, Section 6.1 nous décrire les résultats de l'importation et de l'exportation. Puis nous verrons Section 6.2 la représentation d'une application après l'avoir importée dans notre outil. Enfin, nous discuterons des résultats Section 6.3.

6.1 Résultats

Une fois l'outil implémenté, nous avons cherché à vérifier nos résultats. Comme la stratégie mis en place est en deux étapes, *i.e.* importation et exportation, nous avons séparé la vérification de nos résultats en deux parties.

6.1.1 Rétro-Ingénierie

Phases	Business Pages	Widgets	Link between Phases
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 existante. Pour cela nous avons regardé dans le fichier de configuration de l'application dans lequel toutes les phases sont définis. Puis nous avons comparé le chiffre obtenu avec le nombre de phase que nous instancions dans notre modèle. Dans la 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 le nombre classe qui implémente *IPageMetier*. Il y en a 45. Après l'importation, nous générions 76 pages métiers. Nous retrouvons dans les pages métiers générées celles qui implémente l'interface *IPageMetier* ainsi que 31 que nous avons généré. Ces dernières proviennent de code qui a été factorisé. Cette difficulté d'évaluation est discutée Section 6.3.

Nous réussissons à généré 2081 widgets, cependant avec les heuristiques nous avons défini Section 4.4.2 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 6.3.

Finalement, la detection du nombre de lien entre les phases est réussi à 100%. Nous détectons correctement 101 liens de navigation qui existent dans l'application. Les liens sont tous correctement liée au widget sur lequel il faut faire une action pour déclencher la transition et amène vers la bonne Phase.

6.1.2 Exportation en Angular

Une fois l'importation complété, notre outil procède à l'exportation. L'exportation est évaluée selon les contraintes décrites Section 3.1. Le code doit permettre de garder le même

visuel et préserver l'architecture des éléments de l'interface. De plus, le programme exporté doit être compilable et facile à lire pour un développeur, cette dernière contrainte réduit la difficulté pour les équipes de Berger-Levrault d'accepter la migration.

La lisibilité du code est évaluée selon les critères suivant :

- Nom significatif pour les variables/fonction/class
- Respect des convention du langage cible
- Indentation du code

Pour la lisibilité du code, dans le cadre de la migration l'outil conserve au maximum les noms des éléments de l'application source. Nous supposons donc que, même si les développeurs ont mal nommé leurs variables dans l'application source, ils ne seront pas perturbés par les nouveaux noms de variable.

Le respect des conventions est n'est pas respecté à 100% actuellement. Nous avons réussi à exporter correctement vers l'architecture Angular et la disposition des différents éléments de chaque fichier et fait correctement (par exemple, les variables sont déclarées juste après la déclaration d'une classe). Cependant certaines convention ne sont pas respectées, notamment dans le nommage des classes que nous exportons en majuscule au lieu de respecter le pascalCase¹¹.

Pour l'indentation du code, la convention est que l'indentation correspond à 4 espaces. L'outil de migration exporte le code complètement indenter sauf pour les fichiers HTML.

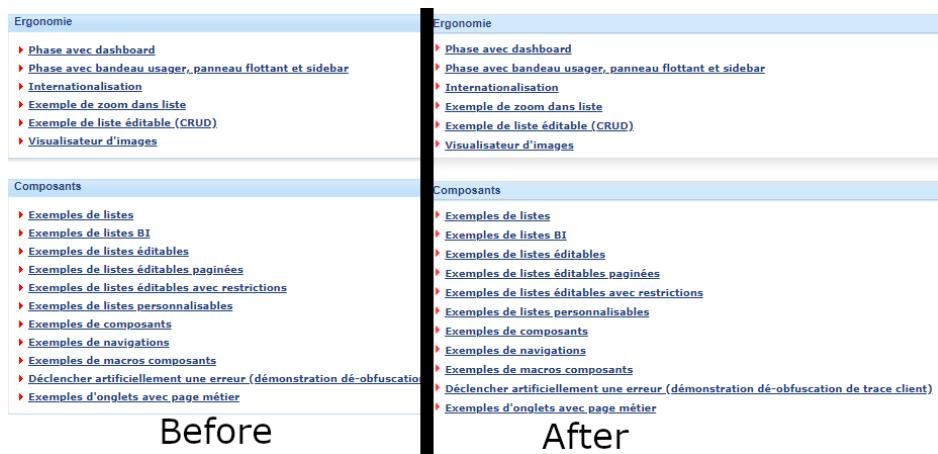


FIGURE 11 – Home - Avant/Après

Le code exporté est compilable à 100% et est exécutable. L'exportation conserve l'architecture entre les éléments de l'interface graphique tel que détecté dans la partie importation. La Figure 11 présente les différences visuelles entre l'ancienne version, à gauche, et la nouvelle, à droite. Nous pouvons voir que les différences sont minimales. Dans la version exportée, les couleurs de l'en-tête des panel est un peu plus claire et l'ombre porté des panel est plus dégradé.

La Figure 12 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 grande, nous les avons rogner pour afficher cette zone d'intérêt. La migration a bien permis de conserver l'architecture entre les éléments, ce que l'on peut voir avec les composants de formulaire à l'intérieur du

11. Pascal Case : les mots sont liés sans espace. Chaque mot commence par une Majuscule.

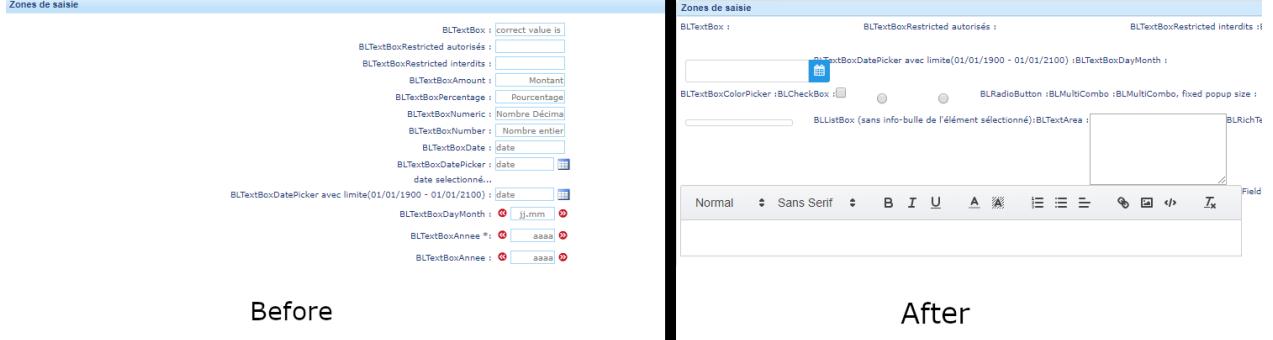


FIGURE 12 – Zone de saisie - Avant/Après

panel “zone de saisie”. Cependant le layout n'a pas été correctement respecté, ce qui explique les différences visuelles entre les deux images.

6.2 Visualisation

Pendant la construction de l'outil de migration, nous avons créé un 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.

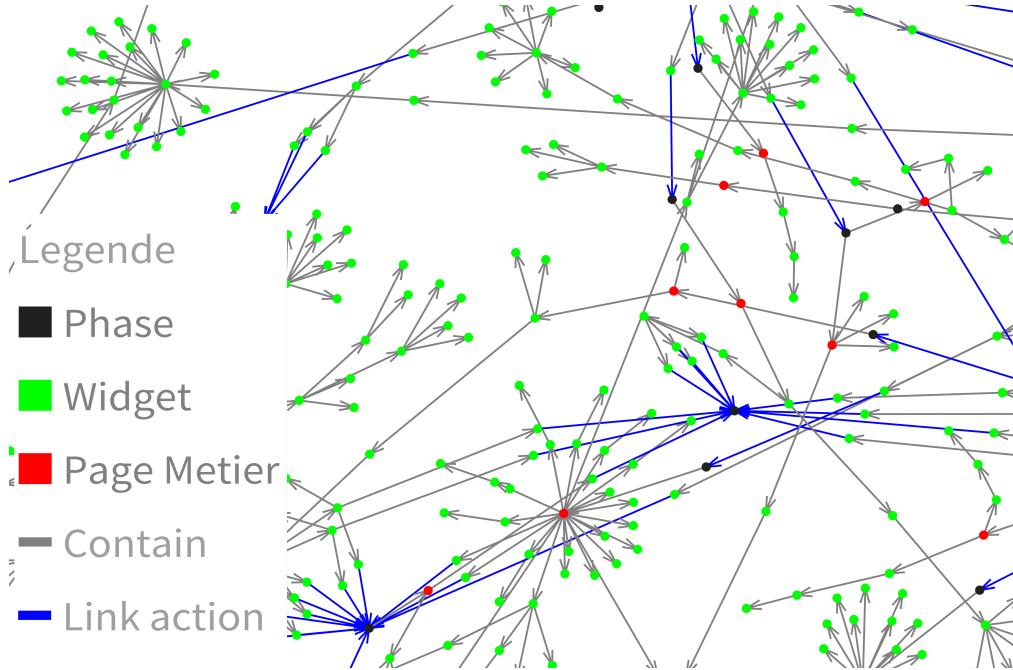


FIGURE 13 – Représentation de l'application bac à sable dans sa globalité

La Figure 13 présente une extrait de la visualisation des relations entre les différentes entités de l'interface graphique de l'application *bac-à-sable*. Les Phases sont représentées en noires, les widgets en vert et les business page en rouge. Une Phase contenant une business page, une business page en contenant une autre ou un widget et un widget en contenant un autre sont représentés par une flèche partant du conteneur vers le contenu. Les liens de navigation sont représentés par une flèche bleue partant du widget sur lequel il faut effectuer une action vers la Phase qui est appelée.

On peut voir sur la Figure 13 des agglomérats de widget. Ceux ci représente des widgets container comme des panels, qui contiennent d'autre widgets, comme des boutons ou du texte.

6.3 Discussion

Les résultats que nous avons obtenu peuvent être discuter. En effet, ils peuvent être remis en cause en partie pour les raisons suivante.

Bien que nous ayons testé régulièrement notre travail sur les applications de production de Berger-Levrault, la recherche des pattern 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 visuelle. Il est aussi possible que les autres logiciels de Berger-Levrault contiennent des deviances dans le code que nous n'avons pas prévu ce qui peut nuire au résultat final.

```
1 <bl_grille id="panelCoche" largeur="100%" remplissage="5" espace="1" hauteur  
= "300px">  
2   <bl_ligne>  
3     <bl_cellule alignementhorizontal="centre" alignementvertical="milieu">  
4       <bl_cadre_epais titre="Boutons radio" largeur="100%">  
5         <bl_bouton_radio id="radio1" groupe="groupe 1" libelle="option 1"/>  
6         <bl_bouton_radio id="radio2" groupe="groupe 1" libelle="option 2"/>  
7         <bl_bouton_radio id="radio3" groupe="groupe 2" libelle="option 3"/>  
8         <bl_bouton_radio id="radio4" groupe="groupe 2" libelle="option 4"/>  
9       </bl_cadre_epais>  
10    </bl_cellule>  
11  </bl_ligne>  
12 </bl_grille>
```

FIGURE 14 – Définition d'interface graphique via fichier xml

En GWT, il est possible de définir une interface graphique grâce à un fichier xml. La Figure 14 présente un extrait d'un fichier de l'application *bac-à-sable* qui permet de générer une interface graphique. La ligne déclare un panel de type grille. Il contient une ligne déclaré ligne 2. Au final, il contient 4 radio button déclarés ligne 5-8. Dans le cadre de ce projet, seul l'application *bac-à-sable* utilise cette technique pour définir des interfaces. Nous avons donc décidé de ne pas traiter pour l'importation des Widgets les business pages définie de cette manière. En les considérant, le pourcentage de Widget que nous arrivons à importer ce voit réduire.

7 Travaux futurs

Nous avons réussi à construire des bases solides pour améliorer l'outil de migration que nous avons commencer. Afin de l'améliorer, il faudrait encore ajouter la gestion des layout, du code comportemental et métier. Nous allons devoir corriger les quelques erreurs restante qui nous empêche 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.

7.1 Amélioration des modèles

Comme expliqué Section 6.1.2, 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 12. Ceci est dû à la manière dont est retranscrit l'idée de layout dans le méta-modèle d'interface graphique. Pour le moment, le layout est considéré comme un Attribut, il nous est difficile de représenter des interfaces complexes. Plusieurs solutions peuvent être envisagées pour représenter les interfaces graphiques.

Gotti et Mbarki [13] ont décidé d'utiliser le meta-modèle proposé par KDM.¹² Le méta-modèle KDM de layout utilise une association entre deux *AbstractUIElement* pour représenter la position de l'un part rapport à l'autre. C'est ainsi qu'est défini le layout d'une application.

Sánchez Ramón *et al.* [8] ont crée 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 possible a connecter aux widgets. Cette solution est celle qui se rapproche le plus des techniques de conception que nous utilisons.

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 comment décrit Section 2. Une fois le layout implémenté nous aurons bien représenter l'interface utilisateur, il nous faudra encore définir et implementer 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 4.3. Il faut encore implémenter ce modèle et modifier en conséquence les importer et exporter que nous avons créé. 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.

7.2 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 effectué, mais sur des applications de la taille de celle de Berger-Levrault il faudrait un système automatique. Les éléments à évaluer serait la validité des 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 envisagés 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 l'architecture d'une page web

12. KDM : Knowledge Discovery Metamodel

et non pas ça 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 serait 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.

Il est possible que les tests automatiques ne fonctionnent plus après la migration, en effet, ils peuvent être basés sur des méta-données 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 compatible 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.

7.3 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 6.1.1, 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 même page web qui nous ont permis de créer les outils de migration, nous n'avons pas connaissance de potentiel erreurs, ou oublis, qui rendez l'outil moins performants. Afin de créer un outil complet, nous devrions chercher et gérer tous les écarts de programmation potentiels pour l'importer.

De même, nous savons que nous n'avons pas géré le cas des interfaces graphiques définis via fichier xml. Bien que cela ne semble pas bloquant dans le cas des projet de Berger-Levrault car cette solution n'est utilisé que dans l'application *bac-à-sable*. Il serait intéressant de voir s'il est "facile" d'ajouter cette fonctionnalité. En particulier, l'ajout de cette fonctionnalité permettrait d'analyser l'extensibilité de l'importer.

7.4 Exportation du Core

La stratégie que nous avons créer permet d'effectuer la migration d'interface graphique d'un langage source vers un langage cible. Entre autre, il est possible de configurer l'outil pour que l'application cible utilise tel ou tel autre framework pour représenter les éléments graphique de l'interface final.

Dans le cas d'une application Web, un certains nombre de widget sont déjà pré-existants. Pour chacun d'entre eux nous retrouvons un tag HTML.

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'existe pas, ou partiellement, dans le langage cible. Dans ce cas, il faut soit accepter une différence entre les deux interfaces, soient créer ou utilisé 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 décrit dans BLCore. Il reste cependant des écarts visuel entre les deux framework et certains widget, 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, leurs compositions, et le style qui leurs sont attachés. Les widgets peuvent aussi contenir du comportement qu'il faudra détecter en migrer.

Un travail futur serait de développer un outil qui permettra 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 Conclusion

Durant ce stage, j'ai continué le projet de migration que j'ai initié 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 créer des outils permettant de faciliter la migration des applications de Berger-Levrault.

La première étape de mon stage a été la définition formelle 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 que l'implémentation du modèle d'interface graphique avec les outils d'importation et d'exportation donnent des résultats encourageant 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*, le code comportemental 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 étudié 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] 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.
- [2] S. I. Feldman, “A fortran to c converter,” in *ACM sigplan fortran forum*, 1990, vol. 9, pp. 21–22.
- [3] 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.
- [4] J. Cloutier, S. Kpodjedo, and G. El Boussaïdi, “WAVI : A reverse engineering tool for web applications,” 2016, pp. 1–3.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.

- [17] K. Garcés *et al.*, “White-box modernization of legacy applications : The oracle forms case study,” *Computer Standards & Interfaces*, 2017.
- [18] 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.
- [19] 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.
- [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 *et al.*, “Learning syntactic program transformations from examples,” in *Proceedings of the 39th international conference on software engineering*, 2017, pp. 404–415.
- [22] M. E. Joorabchi and A. Mesbah, “Reverse engineering iOS mobile applications,” in *Reverse engineering (wcre), 2012 19th working conference on*, 2012, pp. 177–186.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] M. Brambilla, P. Fraternali, and others, “The interaction flow modeling language (ifml),” 2014.

9 Annexe

BENOÎT VERHAEGHE

Ingénieur Logiciel

benoitverhaeghe59@gmail.com [+33 6 58 33 53 74](tel:+33658335374) [40, rue Pasteur, 59260 Lezennes](http://40.rue.Pasteur.59260.Lezennes)
Lezennes, FRANCE [badetitou.github.io/](https://github.io/badetitou) [@badetitou](https://twitter.com/@badetitou) linkedin.com/in/benoitverhaeghe
github.com/badetitou



EXPERIENCE

Ingénieur Logiciel

Berger-Levrault

[Mars - Août 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)

Développeur 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 15 – CV