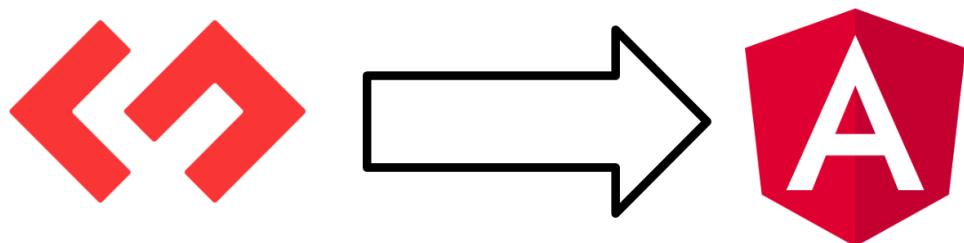


Benoît Verhaeghe

Migration d'application GWT vers Angular



Tuteurs entreprise : M. Deruelle, M. Seriai

Tuteur école : Mme Etien

Responsables Inria : Mme Etien, M. Anquetil

Table des matières

1 Contexte	2
1.1 Contexte général	2
1.2 Problématique	2
1.3 Description du problème	2
2 Etat de l'art	4
2.1 Migration de plateforme	4
2.2 Migration de librairie	6
2.3 Transformation de modèle vers modèle	7
2.4 Transformation de modèle vers texte	8
2.5 Migration de langage	8
3 GUI Décomposition	10
3.1 Interface utilisateur	10
3.2 Code comportemental	10
3.3 Code métier	10
4 Description du problème	11
4.1 Contraintes	11
4.2 Comparaison de GWT et Angular	11
4.3 Stratégies de migration	12
5 Mise en place de la migration par via les modèles	14
5.1 Processus de migration	14
5.2 Méta-modèle d'interface utilisateur	15
5.3 Méta-modèle du code comportemental	16
5.4 Implémentation du processus	17
5.4.1 Meta-modèle	17
5.4.2 Importation	17
5.4.3 Exportation	18
6 Résultats et Discussion	20
6.1 Résultats	20
6.2 Visualisation	20
6.3 Discussion	20
7 Travaux futurs	22
8 Conclusion	23
Bibliographie	24

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 superviseure 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**. Les applications de Berger-Levrault utilisent et/ou étendent **BLCore**, qui lui-même utilise et/ou étend **GWT** comme présenté Figure 1.

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 tout ou partie de 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. Une alternative pour contourner ce problème serait de créer des outils facilitant la migration. Les développeurs pourront alors effectuer la migration rapidement et seront formés sur le nouveau langage et sur l'application.

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

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).

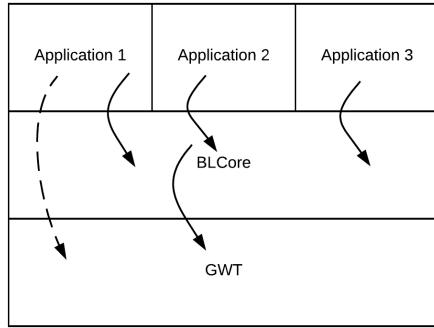


FIGURE 1 – Structure application

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 quel couche de la Figure 1 nous devons migrer et comment ?

En plus des difficultés techniques inherent 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 comportementales.

Mon objectif est donc de trouver des solutions pour aider à la migration des applications de Berger-Levrault, de les évaluer et d'appliquer la migration évaluer la plus satisfaisantes vis-à-vis des contraintes posées. 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 mené. 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 Etat de l'art

La première étape de mon travail a été de le positionner par rapport à la littérature existante.

2.1 Migration de plateforme

La *migration de plateforme* traite de comment migrer une interface graphique.

En particulier, l'article de Samir *et al.* [1] dans lequel il est question de la migration de Swing vers Ajax. Les auteurs ont développé un outil appelé Swing2Script qui permet d'automatiser la migration d'application Java-Swing en une application web Ajax. Cet outil permet d'extraire depuis une application Java les différentes fenêtres et de les transcrire au format XUL². Ensuite, l'outil ajoute à chacun des fichiers un fichier JavaScript contenant l'ensemble des fonctions pouvant être appelées. Cependant, l'outil utilise une analyse dynamique de l'application pour détecter son fonctionnement. Au vu de la taille des applications de Berger-Levrault ainsi que de l'impact de l'utilisation d'une telle stratégie sur les utilisateurs, je ne pourrai pas utiliser la même stratégie.

L'article de Staiger [2] présente un outil d'analyse statique de code source en C qui permet d'extraire l'interface graphique générée par le programme. Pour cela, l'outil va rechercher les constructeurs de composant graphique dans le code et leurs appellent. Il va ensuite "suivre" les pointeurs (C) pour détecter les ajouts de composants dans d'autres composants. Ainsi, il arrive à créer un modèle de l'interface graphique avec l'ensemble des widgets. L'outil permet aussi de récupérer les événements liés aux composants détectés. Dans le cas d'utilisation d'un modèle pour la migration des applications de Berger-Levrault, ce travail peut nous permettre d'extraire le modèle contenant les informations vis-à-vis de la représentation graphique de l'application.

Morgado *et al.* [3] ont créé un outil permettant d'extraire les différents composants d'une interface graphique ainsi que les changements qui s'opèrent lorsque l'on effectue une action sur un composant. Pour cela, les auteurs ont utilisé une stratégie de recherche dynamique. Une fois lancée sur une interface en cours d'exécution, le logiciel va détecter tous les composants graphiques. Puis, il va exécuter des cliques sur les composants de l'interface et détecter les modifications apportées à cette dernière. Comme Berger-Levrault souhaite conserver la même interface graphique, je vais aussi avoir besoin d'extraire la structure de l'interface actuelle. L'outil développait par les auteurs peut peut-être s'appliquer à notre problématique ou nous guider pour la création d'une solution.

L'article de Shah *et al.* [4] présente un logiciel capable d'extraire l'interface graphique d'une application java pendant l'exécution. Le logiciel récupère dans la mémoire de l'ordinateur l'architecture de l'interface graphique avec la composition des widgets. De cette manière, les auteurs arrivent à concevoir, pour chaque "écran" de l'application, un modèle contenant les informations sur l'interface graphique. C'est-à-dire, les différents widgets, leurs propriétés, la composition d'un widget avec un autre (ex : un panel qui contient du texte, un autre panel et un bouton). Comme les auteurs, la migration des applications de Berger-Levrault peut nécessiter que l'on extrait les interfaces graphiques des logiciels. La stratégie d'extraction présentée dans l'article peut guider notre travail.

Sánchez Ramón *et al.* [5] ont développé une solution permettant d'extraire depuis un ancien logiciel son interface graphique. Les auteurs importent cette interface dans un modèle qu'ils ont créé. Leur solution permet ensuite d'extraire des informations de leur modèle. Pour cela, l'outil va délimiter pour chaque widget la taille de sa représentation visuelle. Il connaît donc, la position, la largeur et la hauteur de chaque widget. Les widgets sont alors contenus ou non dans un autre en fonction de leurs positions les

2. XUL : XML-based User interface Language est un langage de description d'interfaces graphiques fondé sur XML créé dans le cadre du projet Mozilla

uns avec les autres. Exemple : un widget qui est à l'intérieur d'un autre visuellement est un contenu par cet autre. Ce travail se rapproche de celui que nous devons effectuer pour la migration des applications de Berger-Levrault. Extraire l'interface graphique pour ensuite la migrer fait aussi partie des tâches essentielles de notre migration. Le modèle représentant l'interface graphique utilisé par les auteurs peut être un point de départ à notre travail d'extraction.

Silvia *et al.* [6] ont créé un logiciel nommé “GUISurfer”. Ce logiciel permet d'extraire une interface graphique d'un code source et de le convertir en un modèle. Le code source peut être du java avec le framework Swing. GUISurfer parcourt l'AST^[^ast] de l'application source pour détecter les composants qu'on lui a donnés en paramètre. On peut donc avoir en sortie un modèle complet de l'interface graphique ou ne contenant que les actions (entrée, clique, etc.). L'article précise qu'un travail sur l'extraction d'interface GWT est en cours. Pour la migration des applications de Berger-Levrault, je vais devoir extraire l'interface graphique des applications écrites avec GWT. Le logiciel “GUISurfer” peut peut-être me permettre d'extraire un modèle de l'interface, dans un premier temps sans les spécificités de Berger-Levrault.

L'article de Gotti *et al.* [7] présente une méthodologie pour extraire les composants d'une interface graphique et leurs relations. Pour cela, les auteurs construisent plusieurs modèles. Le passage entre les modèles se fait grâce à des transformations QVT³. Les auteurs ont appliqué leur projet sur des programmes java utilisant le framework graphique Swing. Dans le cadre de la migration des applications de Berger-Levrault, l'extraction des composants graphiques et de leurs relations peut être nécessaire. En modifiant le travail des auteurs, je pourrai le réutiliser dans le cas de l'extraction de composant GWT.

Memon *et al.* [8] ont développé un logiciel nommé “GUI Ripper”. Cet outil permet d'extraire d'un logiciel java ou MS Windows les différents composants visuels. L'outil fait une recherche dynamique des composants instanciés durant l'exécution du programme. Les auteurs obtiennent un modèle de l'application qu'ils vont ensuite pouvoir utiliser pour générer des tests. L'extraction de composants visuels est aussi une tâche importante pour mon projet avec Berger-Levrault. Cependant, la recherche dynamique proposée par les auteurs ne semblent pas applicable en l'état dans notre cas puisque nous n'avons pas d'exécution de l'interface, mais sa compilation par GWT. Cependant, on peut imaginer implémenter l'algorithme de GUI Ripper en JavaScript pour extraire les widgets pendant l'exécution du script dans le navigateur web.

L'article de Lelli *et al.* [9] propose un outil permettant de détecter les composants graphiques pouvant faire plus de deux actions. L'outil, qui est une extension d'eclipse, va tout d'abord faire une analyse statique du code source. Cela lui permet de repérer les différents widgets. Puis il va détecter les widgets qui ont plus de deux ajout de Listener. Puisque le plugin est capable de détecter les ajout de listener, il doit être capable de détecter les ajouts d'autres widgets. Nous pourrions donc modifier le plugin pour qu'il détecte en plus les compositions de widgets afin d'extraire un modèle contenant l'aspect graphique des applications de Berger-Levrault. Cette dernière étape nous sera utile pour migrer correctement les applications.

Cloutier *et al.* [10] ont conçu un outil nommé “Wavi” qui permet d'extraire d'une page web les différents composants. Pour cela, l'outil se base sur les fichier html, css et JavaScript. L'outil va dans un premier temps construire l'arbre syntaxique du code source du site web. Puis il extrait les éléments importants du fichier html (*i.e.* les hyperliens, formulaires, appel JavaScript, etc.). Enfin il va relier les éléments de l'étape une et deux. Les applications web de Berger-Levrault sont développées en java avec GWT, mais une fois compilées sont des fichiers html, css et JavaScript. Toutes les conditions semblent donc remplit pour pouvoir utiliser le travail des auteurs dans notre cas et ainsi extraire l'architecture des applications de Berger-Levrault. Ce travail est nécessaire si l'on souhaite conserver la même structure visuelle pendant la migration.

Aho *et al.* [11] ont développé un logiciel appelé “Murphy”. Murphy permet d'extraire dynamiquement les widgets d'une application. Murphy est compatible avec de nombreux langages de programmation car il

3. QVT est un langage permettant de définir des transformation de modèle.

utilise des “drivers” qui lui permettent d’interagir avec l’application en cours de fonctionnement grâce à une interface abstraite du langage de programmation de l’application. Le projet des auteurs se rapprochent du nôtre puisqu’ils souhaitent extraire l’interface graphique ce qui est une de nos tâches pour la migration des applications de Berger-Levrault. Il nous faudrait cependant trouver comment créer un “driver” qui permettrait à Murphy d’interagir avec une application web.

2.2 Migration de librairie

La *migration de librairie* présente des solutions sur comment changer de framework.

Teyton *et al.* [12] ont proposé un outil permettant d’extraire depuis des migrations déjà effectuées les correspondances entre les appels d’un framework avec ceux d’un autre. Pour cela, ils se basent sur les différences textuelles entre plusieurs versions d’un même projet. À cause du nombre de versions qu’un projet peut avoir, ils utilisent un algorithme “diviser pour régner” afin d’accélérer le temps de calcul. Ce travail peut nous servir une fois que BLCore aura migré, afin d’automatiser ou créer des outils facilitant le travail des développeurs. Cependant, ce travail ne parle que de la migration au sein d’un même langage de programmation. Il ne peut donc être utilisé tel quel pour faire migrer BLCore de GWT vers Angular 4.

L’article de Hora *et al.* [13] explique une démarche pour extraire automatiquement des modifications faites dans un code afin d’en déduire des patterns. Son objectif est de détecter les conventions de développement qui évoluent. Son travail de recherche de correspondance entre certains morceaux de code et des nouveaux peut m’être utile pour la migration. En effet, je vais aussi devoir trouver les correspondances entre du code Java et du code Angular.

Zhong *et al.* [14] ont voulu créer une approche visant à faire correspondre l’API⁴ d’un framework avec celui d’un autre, tous deux étant écrits dans des langages différents. Pour cela, ils ont développé une stratégie appelée MAM (Mining API Mapping) qui prendra en paramètres deux projets dans deux versions différentes. Ensuite, l’algorithme essaie de regrouper par minage les éléments identiques des deux projets (*i.e.* les classes, les méthodes). Puis il construit un arbre d’exécution pour les méthodes et cherche les correspondances entre les méthodes qui sont appelées. Cette approche ne peut pas m’aider à migrer BLCore, mais, si je migre quelques applications de Berger-Levrault, je pourrai réutiliser les stratégies employées par Zhong pour faciliter la migration d’autres applications.

Nguyen *et al.* [15] ont travaillé sur un outil, nommé StaMiner, permettant de mettre en correspondance des API de framework écrit en Java, avec des API écrite en C#. L’apprentissage des correspondances utilisent des morceaux de code source et cible ayant le même comportement. Il se fait en trois étapes.

1. Le Groum, qui consiste à représenter le code source sous la forme d’un graphe. On y retrouve les appels à des fonctions, des alternatives, des boucles, etc.
2. L’extraction des séquences d’utilisation des différents éléments.
3. L’alignement des séquences entre le résultat pour le code source et le code cible.

Pour effectuer l’alignement, les auteurs ont utilisé des outils probabilistes sur les symboles et sur les séquences. Ce travail peut être utilisé pour la migration des applications de Berger-Levrault. En utilisant cet outil sur une application source et un bout migration *fait à la main*, je pourrai apprendre des règles de transformation à appliquer sur la migration.

L’article de Phan *et al.* [16] décrit une manière de faire correspondre des éléments de code du langage Java vers le langage C#. Plus précisément, ils ont développé un outil permettant de mettre en correspondance du code d’un langage utilisant une ou plusieurs API vers un autre langage utilisant une ou plusieurs autres

4. API : Interface de programmation

API prédéfinies. Pour cela, les auteurs utilisent un outil de recherche de correspondances entre des API provenant de Java vers C#. Puis, ils utilisent une machine statistique de traduction automatique pour faire correspondre les utilisations des API Java et C#. Une fois le modèle entraîné, ils arrivent à automatiser une partie de la migration. Ce travail peut nous servir si l'on a migré BLCore et que l'on souhaite ensuite migrer les applications. Comme lui, nous travaillons sur la migration de librairies de langages différents.

Chen *et al.* [17] 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 même problème peut être une solution. C'est dans ce contexte que le travail des auteurs peut guider notre recherche de librairie en mettant faisant correspondre les anciennes librairies utilisées par les applications de Berger-Levrault avec d'autre compatible utilisable avec Angular.

2.3 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.* [18] 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. L'algorithme va, à partir d'exemples, apprendre les règles de transformation à appliquer afin d'effectuer la transformation du modèle. Pour cela, il va modifier les petites briques composants les règles et analyser si le modèle en sortie ressemble à celui explicité pour tous les exemples. Puis, il sera utilisé 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. Toutefois, la complexité d'un code source Java semble plus grande que celle d'un modèle UML. Il est possible que l'algorithme de programmation génétique ne soit pas assez performant pour régler mon problème de manière satisfaisante.

Falleri *et al.* [19] ont travaillé sur le passage d'un méta-modèle à un autre. C'est ce que l'on appelle l'alignement des méta-modèle. Pour cela, les auteurs ont utilisé l'algorithme de *Similarity Flooding*. Cet algorithme permet de trouver les similarités entre les deux graphes orientés et étiquetés aux arcs en entrée du programme. Les auteurs ont proposé des solutions pour convertir un méta-modèle en graphes orientés et étiquetés aux arcs. Comme les auteurs, je peux décider d'effectuer la migration en passant par des modèles et méta-modèles. Une fois les méta-modèle et modèles créés, je pourrai utiliser l'algorithme *Similarity Flooding* utilisée dans l'article pour effectuer la migration.

Wang *et al.* [20] 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 une méta-méta-modèle pour tous les méta-modèle intermédiaire. 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-modèle. 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.* [21] ont travaillé sur la modernisation et la migration de logiciel au sein d'une entreprise d'informatique. Ils ont développé un logiciel permettant de semi-automatiser la migration des applications de l'entreprise. Pour cela, ils ont utilisé la transformation de modèle sur trois modèles. 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.
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.

L'article de Garcés *et al.* [22] décrit les trois étapes que les auteurs ont suivies pour effectuer une migration d'anciens codes.

1. L'analyse de l'ancien code source pour créer un modèle de l'application suivant un méta-modèle. Ce premier modèle est proche de la technologie de départ.
2. La transformation de ce modèle vers un nouveau modèle plus abstrait.
3. La génération du nouveau code source depuis le dernier modèle.

Comme pour l'article, le travail de migration de Berger-Levrault peut se faire en utilisant des modèles et méta-modèles. De plus, les auteurs ont travaillé sur une migration d'interface graphique ce qui est aussi notre besoin. Nous pourrions donc réutiliser des outils ou résultats de ce travail dans notre cas.

2.4 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.* [23] 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 règle 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.

2.5 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.* [24] 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 parvenues 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.* [25] ont proposé un outil facilitant la création de règle de transformation. Pour cela, l'outil va “normaliser” 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. J'aurai à gérer les multiples manières dont les fonctionnalités seront écrites. La normalisation du code source pourra simplifier l'écriture des règles de transformation ou les règles permettant de créer un modèle.

Rolim *et al.* [26] ont créé un outil qui apprend des règles de transformation de programme à partir d'exemple. Pour cela, les auteurs ont défini un DSL⁵ permettant d'exprimer les modifications faîtes sur l'AST[^ast] d'un programme. Ensuite, à partir d'une base d'exemple de transformation, l'outil recherche les règles de transformation entre les fichiers d'entrés des exemples et ceux de sorties. Une fois les règles trouvées et écrites dans le DSL prédéfini, l'outil prend en entré 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 apprendrait au fur et à mesure des développements des développeurs et qui les conseillerait dans un second temps sur d'autre développement avec les mêmes problématiques déjà résolus. Ou encore, l'outil pourrait servir de “guide” pour les nouveaux développeurs participants à la migration ou au développement courant des applications.

Aucun des papiers trouvés et cités ne peut nous aider réellement à migrer le framework BLCore ou les applications si on décide que dans le futur, on supprime BLCore, puisqu'Angular, contrairement à GWT n'est pas du Java. En revanche, si Berger-Levrault souhaite garder l'équivalent de BLCore dans le futur, alors ces travaux pourraient nous aider à migrer dans un deuxième temps les applications.

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

3 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 petit sous-problème 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

3.1 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.

3.2 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.

3.3 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.

4 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 MLOC⁶ 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.

4.1 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 :

- *Indépendance du langage source.* La solution doit être facile à adapter pour tous projets utilisant GWT ainsi que d'autre logiciel n'étant pas écrit en JAVA, mais possédant une interface utilisateur. Cette contrainte doit être respectée pour pouvoir être réutilisé sur différents projet. Dans le cas de Berger-Levrault, cela permet aux développeurs d'appliquer la solution sur d'autres logiciels qu'ils veulent migrer.
- *Indépendance du langage cible.* Il doit être facile de changer le langage cible de la migration sans devoir restructurer ou développer l'implémentation de la stratégie de migration. Cette contrainte garantie que la solution peut être utilisée quelle que soit l'architecture du langage cible. Dans notre cas, nous migrons des applications de GWT vers Angular. Angular a vu 6 versions majeurs sortir depuis 2016. Grâce à l'indépendance du langage cible, la stratégie de migration reste valide quelle que soit la version du langage cible.
- *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.

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.

4.2 Comparaison de GWT et Angular

Dans le cas de ce projet, le langage de programmation source et cible ont deux architectures différentes. Les différences sont syntaxical, semantical et architectural. Pour la migration d'application GWT vers Angular, les fichier `.java` sont séparés en plusieurs fichiers Angular.

6. MLOC : Million lines of code

Tableau 1 – Comparaison des architectures de GWT et Angular

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

Comme présenté dans le Tableau 1, la séparation des fichiers Java en fichier Angular se fait à trois endroits, les fichiers de configuration, les fichiers définissant la page web et les fichiers de style.

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.

4.3 Stratégies de migration

Il existe plusieurs manières d'effectuer la migration d'une application. Toutes les solutions doivent respecter les contraintes définis Section 4.1.

- *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 re-concevoir l'application cible en suivant les préceptes du langage cible.
- *Utilisation d'un moteur de règle*. L'utilisation d'un moteur de règle pour migrer partiellement ou en totalité une application a déjà été appliqué sur d'autres projets [24], [27], [28]. 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

```

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

```

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

FIGURE 2 – Exemple de règle de transformation

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 règle, 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 métamodèles 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.

5 Mise en place de la migration par via les modèles

Suite à l'étude des contraintes inhérentes au problème 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 4.3, seule la migration en utilisant les modèles nous permet de respecter toutes les contraintes. Ce type de migration nous impose la conception de méta-modèles.

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

5.1 Processus de migration

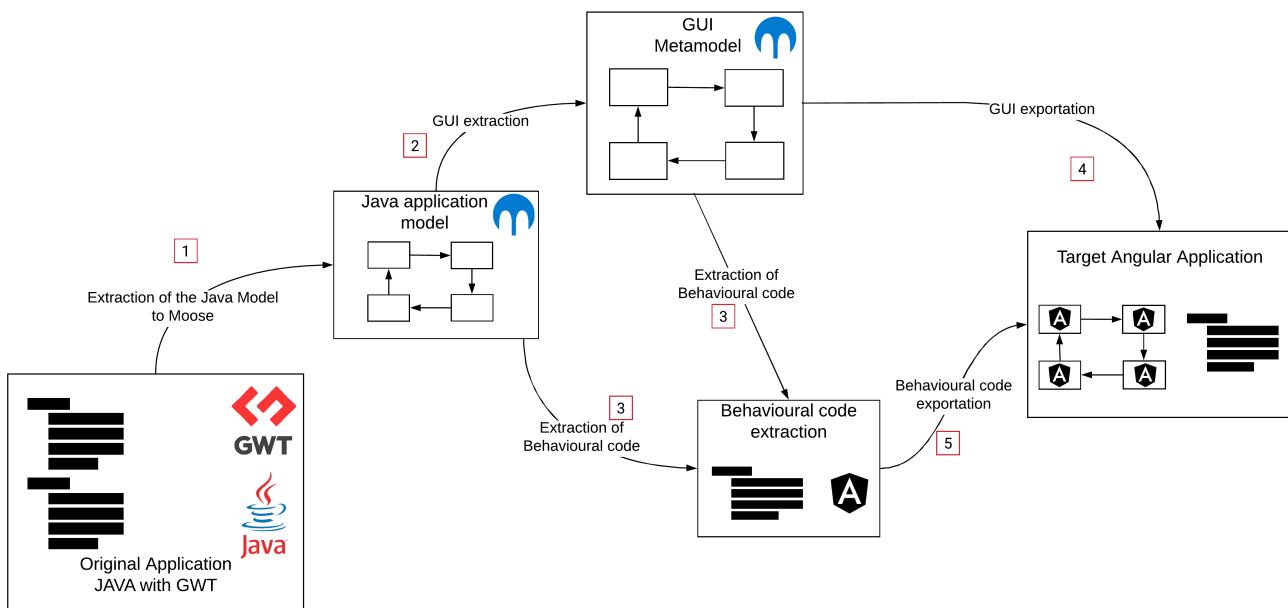


FIGURE 3 – Schema processus de migration

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 méta-modèle permettant de représenter une application écrite en java.
2. L'*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 5.2.
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, défini dans le modèle UI, lorsqu'un clique 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 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.

5.2 Méta-modèle d'interface utilisateur

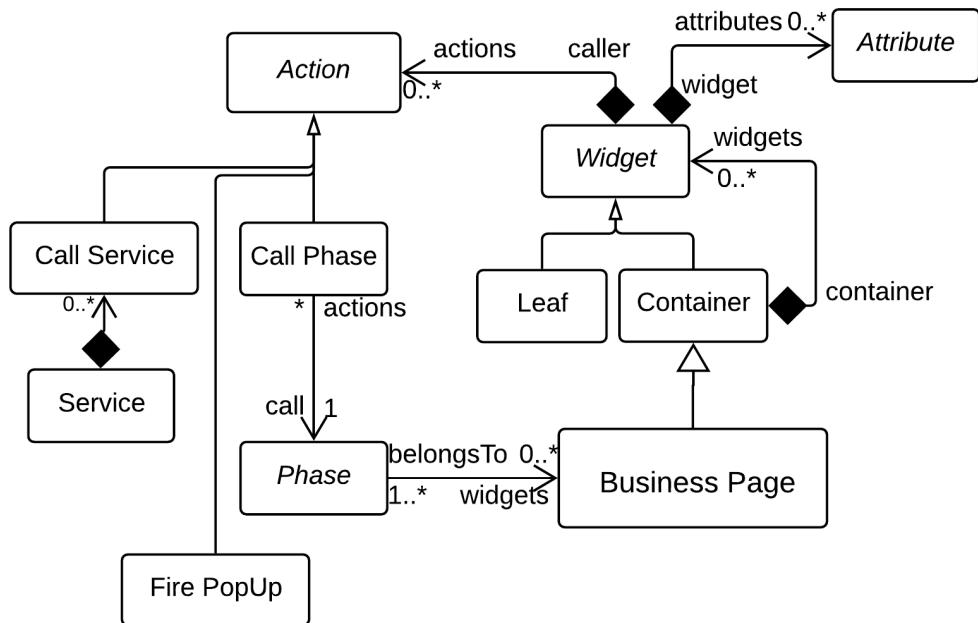


FIGURE 4 – Méta-Modèle d'un application de Berger-Levrault

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'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.

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.

5.3 Méta-modèle du code comportemental

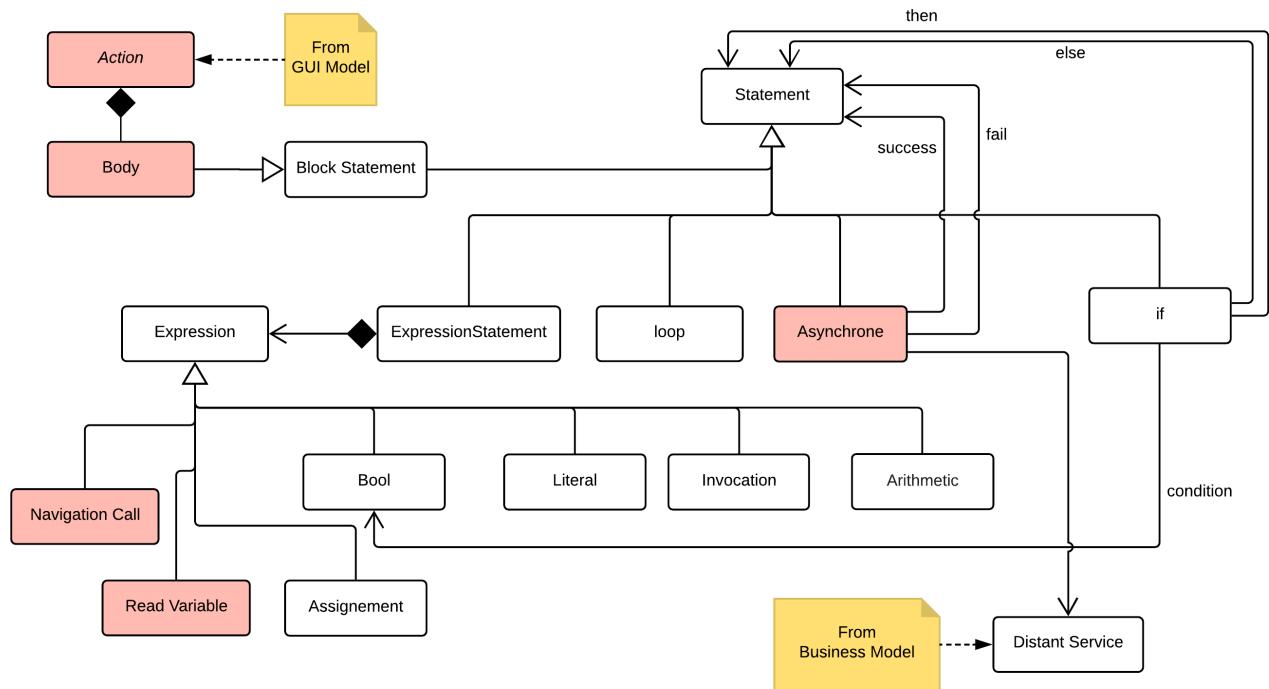


FIGURE 5 – Méta-Modèle d'un application de Berger-Levrault

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 Pharo⁷ et nous avons utilisé la plateforme Moose⁸. Moose est une plateforme pour l'analyse de logiciels et de données.

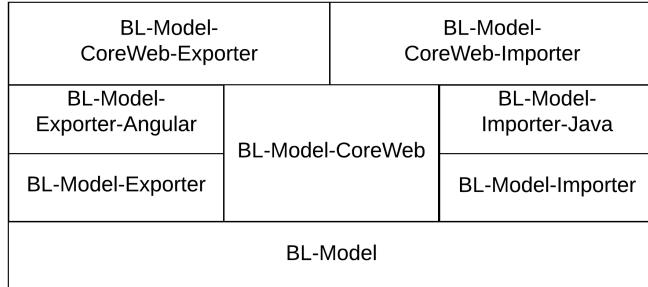


FIGURE 6 – Implémentation de l'outil

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 “Core Web”, 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.

5.4.1 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 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.

5.4.2 Importation

La création des modèles représentant l'interface graphique est divisée en trois étapes comme présenté Section 5.1. 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

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

8. <http://www.moosetechnology.org/>

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 verveineJ⁹ et jdt2Famix¹⁰. 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 méta-modèles, 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 "*addActionHandler*" sont transformés en action tandis que les appels aux méthodes "*setX*" ont été transformé en attribut.

5.4.3 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

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

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

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 et Discussion

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

6.1 Résultats

6.2 Visualisation

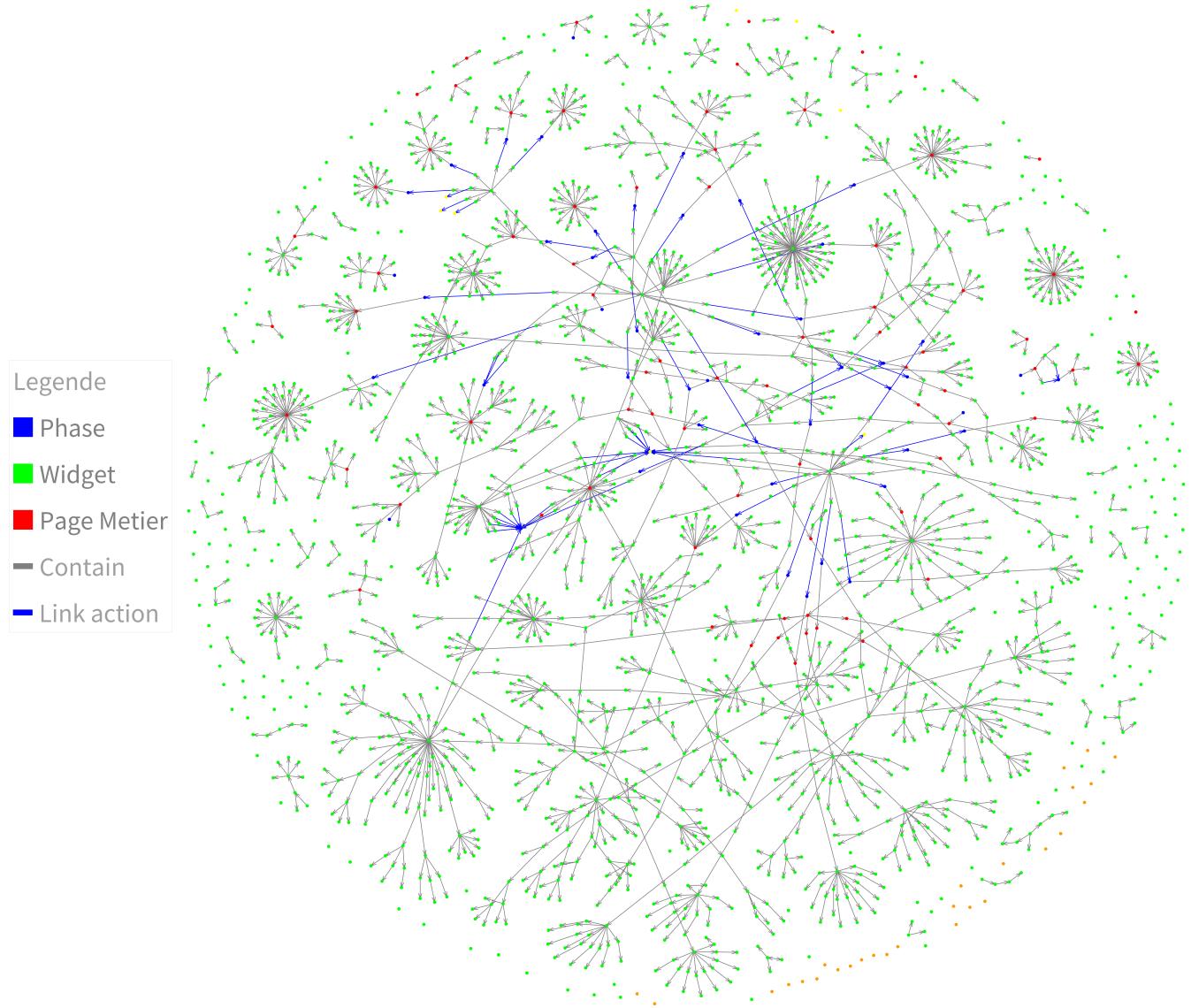


FIGURE 7 – Représentation de l'application bac à sable dans sa globalité

6.3 Discussion

- Les évaluation n'ont été faite que sur l'application *bac-à-sable*

- Il n'y a pas de gestion de certaine manière de décrire une interface en GWT (cf xml file) - mais ça ne sera pas fait
- La migration de l'application ici ne prend pas en compte les eventuelles evolutions parallèle du backend

7 Travaux futurs

Il manque

- layout
- behavior complet
- outil de validation
- business code
- missing widget

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] 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.
- [2] 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.
- [3] 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.
- [4] E. Shah and E. Tilevich, “Reverse-engineering user interfaces to facilitate reporting 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] J. Cloutier, S. Kpodjedo, and G. El Boussaidi, “WAVI : A reverse engineering tool for web applications,” 2016, pp. 1–3.
- [11] 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.
- [12] C. Teyton, J.-R. Falleri, and X. Blanc, “Automatic discovery of function mappings between similar libraries,” in *Reverse engineering (wcre), 2013 20th working conference on*, 2013, pp. 192–201.
- [13] A. Hora, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, “Automatic detection of system-specific conventions unknown to developers,” *Journal of Systems and Software*, vol. 109, pp. 192–204, 2015.
- [14] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, “Mining api mapping for language migration,” in *Proceedings of the 32nd acm/ieee international conference on software engineering-volume 1*, 2010, pp. 195–204.
- [15] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Statistical learning approach for mining api usage mappings for code migration,” in *Proceedings of the 29th acm/ieee international conference on automated software engineering*, 2014, pp. 457–468.
- [16] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, “Statistical migration of api usages,” in *Software engineering companion (icse-c), 2017 ieee/acm 39th international conference on*, 2017, pp. 47–50.

- [17] 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.
- [18] 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.
- [19] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut, “Metamodel matching for automatic model transformation generation,” in *International conference on model driven engineering languages and systems*, 2008, pp. 326–340.
- [20] 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.
- [21] 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.
- [22] K. Garcés *et al.*, “White-box modernization of legacy applications : The oracle forms case study,” *Computer Standards & Interfaces*, 2017.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] R. Rolim *et al.*, “Learning syntactic program transformations from examples,” in *Proceedings of the 39th international conference on software engineering*, 2017, pp. 404–415.
- [27] S. I. Feldman, “A fortran to c converter,” in *ACM sigplan fortran forum*, 1990, vol. 9, pp. 21–22.
- [28] 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.