Migration de GWT vers Angular 6 en utilisant l'IDM

Benoît Verhaeghe, Anne Etien, Stéphane Ducasse, Abderrahmane Seriai, Laurent Deruelle, Mustapha Derras

Migration de GWT vers Angular 6 en utilisant l'IDM

Benoît Verhaeghe[1,2,3], Anne Etien[1,3], Stéphane Ducasse[3,1], Abderrahmane Seriai[2],
Laurent Deruelle[2], Mustapha Derras[2]
[1]Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 – CRIStAL, 59650 Villeneuve d'Ascq, France
{firstname.lastname}@univ-lille.fr
[2]Berger-Levrault, France
{firstname.lastname}@berger-levrault.com
[3]RMod team, INRIA Lille Nord Europe, Villeneuve d'Ascq, France
{firstname.lastname}@inria.fr

Résumé

Dans le cadre d'une collaboration avec Berger-Levrault, une société d'édition logicielle, nous travaillons à la migration d'une application GWT vers Angular. Nous nous concentrons sur l'aspect GUI de cette migration qui, même si les deux frameworks sont des frameworks d'interface graphique (GUI) pour le web, est rendue difficile parce qu'ils utilisent des langages de programmation différents (Java pour l'un, Typescript — un surensemble de JavaScript — pour l'autre) et différents schémas d'organisation (*e.g.* différents fichiers XML). De plus, la nouvelle application doit pouvoir imiter l'aspect visuel de l'ancienne afin que les utilisateurs de l'application ne soient pas perturbés dans leurs habitudes de travail. Nous proposons une approche en trois étapes qui utilise un méta-modèle pour représenter l'interface graphique. Ce méta-modèle permet à notre approche d'accepter différentes langues sources et cibles. Nous avons évalué cette approche sur une application comprenant 470 classes Java (GWT) représentant 56 pages web. Nous sommes capables de modéliser toutes les pages web de l'application et 93% des widgets qu'elles contiennent, et nous avons migré avec succès (*i.e.* le résultat est visuellement égal à l'original) 26 pages sur 39 (66%). Nous donnons des exemples de pages migrées, avec ou sans succès. Nous présentons également les résultats de quelques expériences de migration sur une application de bureau, non implémentée avec GWT, vers une application web, sans utiliser Angular.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

1 Introduction

Lors de l'évolution d'une application, il est parfois nécessaire de migrer son implémentation vers un langage de programmation ou une interface graphique (GUI) différente [2, 27]. Les frameworks d'interface graphique Web, en particulier, évoluent à un rythme rapide. Par exemple, en 2018, il y avait deux versions majeures de Angular, trois versions majeures de React.js, quatre versions de Vue.js, et trois versions d'Ember.js. Cela oblige les entreprises à mettre à jour régulièrement leurs logiciels afin d'éviter d'être bloquées avec d'anciennes technologies.

Notre travail se fait en collaboration avec Berger-Levrault, une importante entreprise informatique qui vend des applications Web développées avec GWT. Cependant, GWT n'est plus mis à jour avec une seule version majeure depuis 2015. En conséquence, Berger-Levrault a décidé de migrer les interfaces graphiques des ses applications vers Angular 6.

La société développe 8 applications utilisant GWT. Chaque application a plus de 1.5 MLOC [1] représentant plus de 400 pages web. Les applications sont composées de plus de 45 types de widgets et 29 types d'attributs. La société a estimé la migration pour une application à 4 000 jours-homme. Ainsi, migrer automatiquement la partie visuelle d'une application est déjà une étape utile pour la modernisation des applications de l'entreprise. En raison de l'évolution rapide des frameworks d'interface graphique, l'entreprise a également besoin d'une solution réutilisable pour la migration vers le prochain langage de programmation.

Il existe de nombreux articles publiés sur l'aide à la mi-

1. MLOC : Million de lignes de code

gration d'interfaces graphiques (*e.g.* [11, 22, 24]). Aucun d'entre eux ne parle du cas de la migration d'interfaces graphiques d'applications Web.

Nous présentons une approche pour aider les développeurs à migrer l'interface graphique de leurs logiciels basés sur le Web. Cette approche inclut un méta-modèle d'interface graphique, une stratégie pour générer le modèle, et l'exportation du modèle pour créer l'interface graphique cible. Pour valider cette approche, nous avons développé un outil qui migre des applications Java GWT vers Angular. Ensuite, nous avons validé notre approche sur un projet industriel qui sert à présenter tous les widgets et leur utilisation. Il est composé de 470 classes Java et 56 pages web. Notre approche a importé correctement 93% des widgets et 100% des pages. Comme tous les widgets existants ne sont pas réimplémentés dans Angular, nous avons essayé de migrer 39 pages et avons réussi (même apparence visuelle) pour 26 d'entre elles (66%).

Section 2, nous passons en revue la littérature sur la méta-modélisation de l'interface graphique. Nous décrivons le contexte de notre projet Section 3. Section 4, nous décrivons notre approche de migration. Nous présentons notre implémentation Section 5. La Section 6 décrit l'expérience que nous avons faite pour valider notre approche. Section 7, nous présentons nos résultats. Enfin, la Section 8 et la Section 9 discutent des résultats obtenus avec notre outil et des travaux futurs.

2 État de l'art

La Section 2.1 présente les techniques utilisées pour migrer une application. Section 2.2, nous décrivons les méta-modèles d'interface graphiques trouvés dans la littérature.

2.1 Stratégie de migration existantes

Pour définir une stratégie de migration, nous avons identifié des travaux de recherche liés à la migration des applications. Certaines des approches proposées n'effectuent pas une migration complète, mais seulement en partie. Toutes les approches utilisent des techniques de rétroingénierie. De plus, il existe de nombreuses publications traitant de la migration de langage de programmation. Nous ne les considérons cependant pas s'ils ne discutent pas explicitement de la migration d'interfaces graphiques. C'est le cas, par exemple, du travail de Brant *et al.* [2] qui fait état d'une importante migration de Delphi vers C#.

Nous avons identifié trois techniques pour créer une représentation d'interfaces graphiques : statique, dynamique ou hybride.

Statique. La stratégie statique consiste à analyser le code source et à en extraire des informations. Il n'exécute pas le code de l'application analysée.

Cloutier *et al.* [3] analyse directement les fichiers HTML, CSS et JavaScript. L'analyse construit un arbore syntaxique du code source du site web et extrait les widgets des fichiers HTML. Le travail consiste principalement à identifier les liens entre les éléments du programme source (classes JavaScript, balises HTML, etc.). Le travail présenté n'aborde pas la migration complète de l'interface graphique.

Lelli *et al.* [13], Silva *et al.* [25] et Staiger [26] ont utilisé des outils qui analysent le code source des applications de bureau. Les outils recherchent la création des widgets dans le code source, puis ils analysent les méthodes qui ont invoqué ou sont invoquées par les widgets pour identifier les relations entre les widgets et leurs propriétés visuelles.

Sánchez Ramán *et al.* [23] et Garcés *et al.* [8] ont développé des approches pour extraire l'interface graphique des applications Oracle Forms. Avec ce framework, les développeurs définissent les interfaces dans des fichiers externes où la position de chaque widget est spécifiée. Leur approche consiste à créer la hiérarchie des widgets à partir de leur position. Cependant, ces études de cas sont simples et ne comportent que peu de formulaires ou de textes. La mise en page est également simple car tous les éléments sont affichés les uns en dessous des autres.

La stratégie statique permet d'analyser une application sans avoir à l'exécuter ni même à la compiler. Outre le problème classique de montrer tous les faits potentiels plutôt que seulement les faits réels, une autre limitation apparaît par exemple, avec une application client/serveur, quand une partie de l'interface graphique dépend du résultat d'une requête à un serveur.

Dynamique. La stratégie dynamique consiste à analyser les interfaces graphiques d'une application en cours d'exécution. Elle explore les états de l'application en effectuant toutes les actions possibles sur l'interface graphique du logiciel et extrait les widgets et leurs informations.

Memon *et al.* [15], Samir *et al.* [22], Shah and Tilevich [24] et Morgado *et al.* [20] ont développé des outils qui mettent en œuvre une stratégie dynamique. Cependant, les solutions proposées ne sont disponibles que pour les applications de bureau et non pour les applications Web.

L'analyse dynamique permet d'explorer toutes les fenêtres d'une application et de recueillir des informations détaillées à leur sujet. Cependant, l'exécution automatique d'une application pour capturer méthodiquement tous ses écrans peut être une tâche difficile en fonction de la technologie utilisée. De plus, si une requête serveur est utilisée pour construire une des interfaces graphiques, l'analyse dynamique ne détecte pas cette information qui peut être essentielle pour la représentation complète des interfaces.

Hybrid. La stratégie hybride tente de combiner les avantages des analyses statiques et dynamiques.

Gotti and Mbarki [9] a utilisé une stratégie hybride pour analyser des applications Java. Ils créent d'abord un modèle à partir d'une analyse statique du code source. L'analyse statique trouve les widgets et attributs d'une interface graphique et leurs structures. Ensuite, l'analyse dynamique exécute toutes les actions possibles liées à un widget et ana-

lyse si une modification se produit sur l'interface.

Malgré l'utilisation de l'analyse statique et dynamique, la stratégie hybride ne résout pas le problème de requête inhérent aux applications client/serveur. Il a également les mêmes problèmes que l'analyse dynamique de l'exécution automatique d'une application et de la capture de ses écrans.

Fleurey *et al.* [7] et Garcés *et al.* [8] ont travaillé sur la migration complète de logiciels. Ils ont développé un outil qui effectue la migration de façon semi-automatique. Pour ce faire, ils ont utilisé le procédé du fer à cheval (Kazman *et al.* [12]). La migration est divisée en quatre étapes :

1. Génération du modèle de l'application originale.
2. Transformation de ce modèle en modèle pivot. Cela comprend les structures de données, les actions et algorithmes, l'interface utilisateur et la navigation.
3. Transformation du modèle pivot en modèle du langage cible.
4. Génération du code source dans le langage cible.

Aucun des auteurs n'a envisagé la migration des interfaces graphiques du Web vers des interfaces graphiques du Web. De plus, aucun n'avait la contrainte de garder une mise en page similaire sauf Sánchez Ramán *et al.* [23] ; cependant, ils travaillaient sur des applications Oracle Forms qui ont des interfaces très différentes de celles que l'on retrouve dans le Web. Par conséquent, leur travail n'est pas directement applicable à notre étude de cas.

2.2 Représentation de l'interface utilisateur

Dans la section précédente, de nombreuses représentations abstraites d'interfaces graphiques sont utilisées. Nous avons examiné les représentations proposées et les avons comparées. Nous présentons maintenant les deux méta-modèles d'interface graphique définis par l'OMG. Le "Knowledge Discovery Metamodel" (KDM) permet de représenter tout type d'application. L'"Interaction Flow Modeling Language" (IFML) est spécialisé dans les applications avec interface graphique. La Section 2.2.2 présente d'autres représentations décrites dans la littérature et les compare à celles de l'OMG.

2.2.1 Les standards de l'OMG

L'OMG définit la norme KDM pour aider l'évolution des logiciels. La norme définit un méta-modèle pour représenter un logiciel à un haut niveau d'abstraction. Il inclut un module d'interface graphique qui représente les composants et le comportement d'une interface graphique.

La Figure 1 représente les entités principales de la partie de l'interface utilisateur appelée diagramme de classes UIResources. L'entité principale est **UIResource**. Elle peut être affinée comme **UIDisplay** ou **UIField**. **UIDisplay** correspond au support physique sur lequel l'interface sera affichée, *e.g.* un écran d'ordinateur, un rapport imprimé, etc.

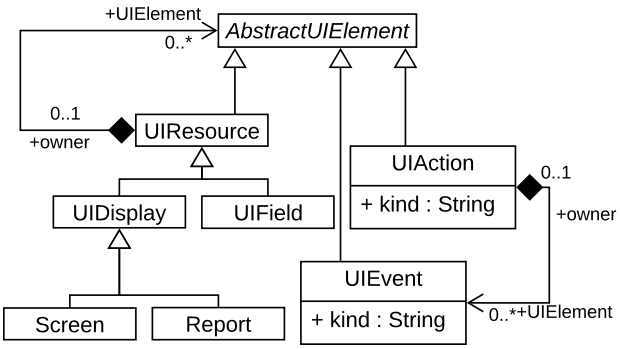


FIGURE 1 – KDM - Diagramme de classes **UIResources**

UIField correspond à un widget d'une interface graphique, *e.g.* un formulaire, un champ texte, un panneau, *etc.* La composition entre **UIResource** et **AbstractUIElement** est utilisée pour définir le DOM (Document Object Model). Chaque **UIResource** peut en contenir un autre pour représenter un widget qui contient un autre widget.

Un **UIResource** peut avoir, par composition, une UIAction pour représenter le comportement de l'interface graphique.

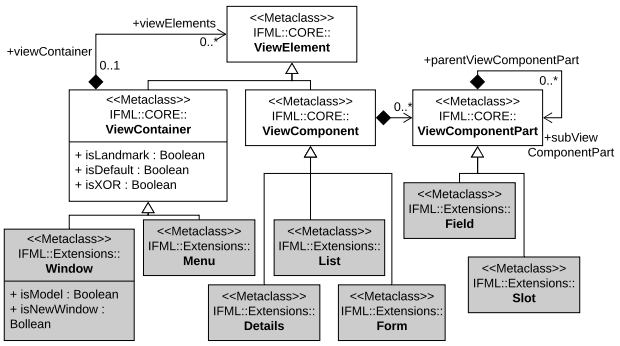


FIGURE 2 – IFML - View Elements

Le but de IFML [1] est de fournir des outils pour décrire la partie visuelle d'une application, avec les composants et les conteneurs, l'état des composants, la logique de l'application et le lien entre les données et l'interface graphique.

La Figure 2 représente le méta-modèle de la partie visuelle d'une application. Les éléments visibles de l'interface graphique sont appelés **ViewElement**. Un **ViewElement** peut être affiné comme un **ViewContainer** ou un **ViewComponent**.

Un **ViewContainer** représente un conteneur d'autres **ViewContainer**s ou **ViewComponent**s. Par exemple, cela peut être une fenêtre, une page HTML, un panneau *etc.*. La composition entre **ViewContainer** et **ViewElement** est utilisée pour définir le DOM.

Un **ViewComponent** correspond à un widget qui affiche son contenu, par exemple un formulaire, un tableau, une galerie d'images, *etc.* Il peut être lié à plusieurs **ViewCom-**

ponentPart. Un **ViewComponentPart** représente un élément d'un **ViewComponent**. Par exemple, un champ de saisie dans un formulaire, un texte qui est affiché à l'intérieur d'un tableau, ou une image d'une galerie.

2.2.2 GUI meta-models

D'autres méta-modèles d'interfaces graphiques ont été proposés dans la littérature. Nous les comparons aux normes de l'OMG.

Tous les méta-modèles utilisent le patron de conception Composite pour représenter le DOM d'une interface graphique et définissent une entité correspondant à **UIResource** pour représenter un élément graphique d'une interface.

Gotti and Mbarki [9] et Sánchez Ramán *et al.* [23] ont proposé un méta-modèle inspiré du modèle KDM. Le méta-modèle est composé des entités principales définies par KDM. Les deux auteurs ont ajouté l'entité Attribute au méta-modèle. Ils définissent également différents types de widgets tels que Button, Label, Panel, *etc.*

Fleurey *et al.* [7] ne décrivaient pas explicitement le méta-modèle de l'interface graphique, mais nous avons extrait des informations de leur modèle de navigation. Ils ont au moins deux éléments dans leur modèle d'interface graphique qui représentent une **Window** et un **GraphicElement**. La **Windows** correspond à l'entité Display du modèle KDM. Et parce que le **GraphicElement** et le **Window** ne sont pas liés, on peut supposer que le **GraphicElement** est une source UIRessource. Le **GraphicElement** a un **Event**.

Morgado *et al.* [20] ont utilisé un méta-modèle d'interface graphique mais ne l'ont pas décrit. Nous savons seulement que l'interface graphique est représentée sous la forme d'un arbre similaire au DOM.

Le méta-modèle UI de Garcés *et al.* [8] diffère beaucoup des précédents. Il y a les attributs, les événements et la page mais la notion de widget est présente comme un champ qui affiche les données d'une table. Ils ont également utilisé une entité **Event** pour représenter l'interaction de l'utilisateur avec l'interface utilisateur. L'entité **Event** correspond aux entités **Action** et **Event** du modèle KDM.

Memon *et al.* [15] ont représenté une interface graphique avec seulement deux entités. Une fenêtre de l'interface graphique qui est composée d'un ensemble de widgets qui peuvent avoir des attributs. Représenter les DOM n'était pas dans le cadre de leur travail. Il n'est pas possible de le représenter avec leur méta-modèle.

Samir *et al.* [22] ont travaillé sur la migration des applications Java-Swing vers des applications Web Ajax. Ils ont créé un méta-modèle pour représenter l'interface graphique de l'application originale. Ce méta-modèle est stocké dans un fichier XUL (langage d'interface graphique basé sur XML) et représente les widgets avec leurs attributs et la mise en page. Ces widgets appartiennent à une **Window** et peut déclencher des événements lorsqu'une action est effectuée sur l'interface graphique. L'action et l'événement correspondent aux entités **Action** et **Event** du modèle KDM. Le format XUL a été abandonné.

Shah and Tilevich [24] ont utilisé une architecture arborescente pour représenter les interfaces graphiques. Cela leur permet de modéliser le DOM. La racine de l'arbre est un **Frame**. Il correspond à l'entité **UIDisplay**. La racine contient les composants avec leurs attributs.

Joorabchi and Mesbah [11] ont représenté une interface graphique avec un ensemble d'éléments graphiques. Ces éléments correspondent à la définition d'un **UIField**. Pour chaque élément de l'interface, l'outil des auteurs détecte de multiples attributs et actions.

Memon [16] utilise un modèle d'interface graphique pour représenter l'état d'une application. Un état est défini à partir des widgets de l'interface graphique et leurs propriétés.

Mesbah *et al.* [17] n'ont pas présenté directement leur méta-modèle pour les interfaces graphiques. Cependant, ils expliquent utiliser une représentation arborescente pour analyser différentes pages Web. Ils ont également utilisé la notion d'événements qui peuvent être déclenchés. Ils ont utilisé différentes instances de leur méta-modèle d'interface graphique pour représenter les pages Web de l'application. Ces instances peuvent être comparées à plusieurs entités UIDisplay.

Tous les auteurs ont utilisé la notion de widget qui représente une entité visuelle de l'interface graphique. La plupart d'entre eux ont une entité attribut qui représente une caractéristique d'un widget. Enfin, les liens de navigation sont représentés par une entité d'action.

3 Contexte du projet de migration

Le but de notre travail est de migrer les interfaces graphiques d'un framework d'interface graphique à un autre. Il s'agit d'un projet industriel de migration d'applications web de GWT vers Angular. L'objectif est de produire une interface graphique exécutable dans le framework cible. Nous présentons maintenant les conditions du projet. Dans la Section 3.1 nous énumérons quelques contraintes que nous devons respecter. Dans la Section 3.2 nous décrivons les principales différences entre les applications GWT et Angular. Dans la Section 3.3 nous présentons une catégorisation du code source du front-end.

3.1 Contraintes

D'après les travaux précédents de Moore *et al.* [18] et Sánchez Ramán *et al.* [23], nous identifions les contraintes suivantes pour notre étude de cas :

— *Depuis GWT vers Angular*. Dans le cadre de la collaboration avec Berger-Levrault, notre approche de migration doit fonctionner avec Java GWT comme langue source et TypeScript Angular comme langue cible.

TABLE 1 – Comparaison de l'organisation des applications GWT et Angular

	GWT	Angular
Définition d'une page web	Une classe Java	Un fichier TypeScript et un fichier HTML
Aspect visuel principal de l'application	Un fichier CSS	Un fichier CSS
Aspect visuel spécifique d'une page web	Inclus dans le fichier Java	Un fichier CSS optionnel
Nombre de fichiers de configuration	Un fichier	Deux fichiers

— *Adaptabilité de l'approche*. Notre approche doit être aussi adaptable que possible pour des contextes différents. Par exemple, elle peut être utilisée avec différentes langues source et cible. Cette contrainte inclut les contraintes *Source and target independence* et de *Modularity*.

— *Préservation du visuel*. La migration doit générer l'aspect visuel de l'application cible le plus près possible de l'original. Cette contrainte inclut le *Layout-preserving migration* ce qui est en opposition avec le *GUI Quality improvement*.

— *Conservation de la qualité du code*. En tant que contrainte *Code Quality improvement* allégé, notre approche devrait produire un code qui semble familier aux développeurs de l'application source. Dans la mesure du possible, le code cible devrait conserver la même structure, les mêmes identificateurs et les mêmes commentaires. Cependant, nous verrons dans la prochaine section qu'il existe de fortes différences dans l'organisation des applications GWT et Angular.

— *Automatique*. Une solution automatique rend l'approche plus accessible. Il serait plus facile d'utiliser une approche automatique sur un grand système [18]. Cette contrainte correspond à la contrainte *Automation* de la littérature.

3.2 Comparison de GWT et Angular

Dans ce projet, la langue source et la langue cible imposent deux schémas d'organisation différents. Leurs différences sont syntaxiques et sémantiques.

GWT est un framework qui permet aux développeurs d'écrire une application web en Java. Le code GUI est compilé en code HTML, CSS et JavaScript. Angular est un framework d'application Web qui permet aux développeurs d'écrire une application web avec le langage TypeScript. Il est utilisé pour créer des Single-Page Applications [2]

La Table 1 résume les différences entre les applications GWT et Angular concernant : la définition des pages web, leur style et les fichiers de configuration. Avant d'expliquer ces trois différences, nous notons une similitude majeure : les applications GWT et Angular ont toutes deux un fichier CSS principal pour définir l'aspect visuel général de l'application.

2. Les Single-Page Applications (SPA) sont des applications Web qui chargent une seule page HTML et mettent à jour dynamiquement cette page lorsque l'utilisateur interagit avec l'application.

— **Définition de page web.** Dans le framework GWT, un seul fichier Java est nécessaire pour définir une page web (un extrait est proposé Figure 5, page 7). Le fichier Java (GWT) comprend les principaux composants graphiques (widgets) de la page Web, leurs positions et leurs organisations hiérarchiques. Dans le cas d'un widget actionnable (comme un bouton), l'action est implémentée dans le même fichier. Dans Angular, il existe une hiérarchie de fichiers pour chaque page Web. Chaque page web est considérée comme un sous-projet indépendant des autres. Un sous-projet contient deux fichiers : un fichier HTML, contenant les widgets de la page Web et leurs organisations ; et un fichier TypeScript, contenant le code à exécuter lorsqu'une action est exécutée.

— **Aspect visuel** L'aspect visuel d'une page Web comprend la couleur ou les dimensions spécifiques des éléments affichés. Dans le cas de GWT, l'aspect visuel spécifique est défini dans le fichier Java du fichier de définition de la page web. Dans Angular, il existe un fichier CSS distinct optionnel.

```
1 <application name="CORE-Incubator">
2   <module name="KITCHENSINK">
3     <phase codePhase="KITCHENSINK_HOME"
4       className="fr.bl.client.kitchensink.
      PhaseHomeKitchenSink"
5       title="Home"/>
6   </module>
7 </application>
8
```

FIGURE 3 – Exemple d'un fichier de configuration GWT en XML

— **Fichiers de configuration** Pour les fichiers de configuration, GWT utilise un fichier XML qui définit les liens entre un fichier Java, une page Web et l'URL de la page Web. La Figure 3 présente un extrait du fichier XML d'une application de Berger-Levrault. La balise **application** est la balise racine du fichier. Elle définit le nom de l'application GWT. La balise **phase** (ligne 3) définit une page Web de l'application GWT : Le titre de la page Web est "Home" ; elle est définie par la classe Java `PhaseHomeHomeKitchenSink` (dans le paquetage `fr.bl.client.kitchensink`); et l'URL pour accéder à la page Web est

`mserver.com/KITCHENSINK_HOME`. Pour Angular, il existe deux fichiers de configuration : *module* qui définit les composants de l'application, *e.g.* pages web, services distants et composant graphiques, et *routing* qui définit pour chaque page web son URL associée.

3.3 Structure d'application front-end

Comme proposé par Hayakawa *et al.* [10], nous avons divisé le projet de migration en plusieurs sous-problèmes. Pour ce faire, nous définissons trois catégories de code source : le code visuel ; le code comportemental ; et le code métier.

— **Code visuel** Le code visuel décrit l'aspect visuel de l'interface graphique. Il contient les éléments de l'interface. Il définit les caractéristiques inhérentes aux composants, telles que la possibilité d'être cliqué ou leur couleur et leur taille. Il décrit également la position des composants par rapport aux autres.

— **Code comportemental** Le code comportemental définit le flux d'action/navigation qui est exécuté lorsqu'un utilisateur interagit avec l'interface graphique. Le code comportemental contient les structures de contrôle (boucle et alternative).

— **Code métier** Le code métier est spécifique à une application. Il comprend les règles de l'application, les adresses des serveurs distants et les données spécifiques à l'application.

En raison de la taille et de la diversité du code source, la migration d'une de ces catégories de code est déjà un problème important.

4 Approche pour la migration

Cette section présente l'approche pour la migration que nous avons conçue. Dans la Section 4.1, nous décrivons le processus de migration que nous avons conçu. La Section 4.2 présente notre méta-modèle GUI.

4.1 Processus de migration

À partir de l'état de l'art, des contraintes et de la décomposition des interfaces utilisateurs, nous avons conçu une approche pour la migration.

Le processus, représenté Figure 4, est divisé en trois étapes :

1. *Extraction du modèle du code source.* Nous construisons un modèle représentant le code source de l'application originale. Dans notre étude de cas, le programme source est écrit en Java GWT. L'extraction produit un modèle FAMIX [5] de l'application utilisant un méta-modèle capturant les concepts Java. Nous devons également analyser le fichier de configuration XML décrit dans la Section 3.2.

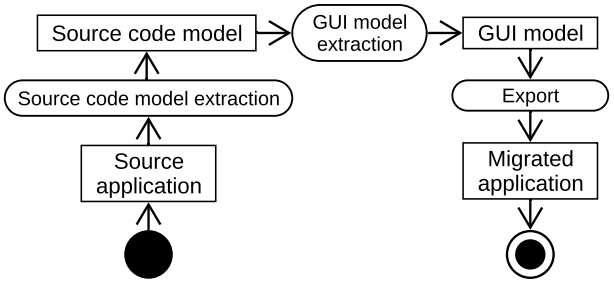


FIGURE 4 – Notre processus de migration

2. *Extraction du modèle GUI.* Nous analysons le modèle de code source pour détecter les éléments du *code visuel* décrivant l'interface graphique et nous construisons un modèle d'interface graphique à partir de ces éléments. Le méta-modèle de l'interface graphique est décrit Section 4.2.
3. *Export.* Nous recréons l'interface graphique dans la langue cible. Cette étape exporte les fichiers de l'interface utilisateur et les fichiers de configuration de l'application.

Notez qu'actuellement nous ne traitons ni le *code métier* ni le *code comportemental* de l'application. C'est sur ce point que porteront les travaux futurs.

4.2 Méta-modèle GUI

Afin de représenter les interfaces graphiques d'applications de bureau ou Web, nous avons conçu un méta-modèle GUI à partir de ceux présentés dans la Section 2.2.2. Dans la suite de cette section, nous présentons les entités du méta-modèle.

Notre méta-modèle est une adaptation du méta-modèle KDM (voir Figure 1). Comme beaucoup d'autres, nous séparons les éléments graphiques correspondant au DOM des actions et événements. Dans notre méta-modèle, les éléments graphiques sont appelés **Widget**. Ils peuvent être raffinés comme **Leaf** ou **Container**.

Dans notre contexte, l'interface graphique sera toujours affichée sur un écran. Nous ne représentons donc pas tout le type d'UIDisplay et définissons une entité **Page**. La **Page** représente le conteneur principal d'une interface graphique. Cela peut être une fenêtre d'une application de bureau ou une page web. La **Page** est un type de **Container**.

Comme proposé par de nombreux autres auteurs, nous avons ajouté l'entité **Attribute** dans notre méta-modèle d'interface graphique. Un **Attribute** représente les informations d'un widget et peut changer son aspect visuel ou son comportement. Quelques attributs communs sont la hauteur et la largeur pour définir précisément la taille d'un widget. Il y a aussi des attributs qui contiennent des données. Par exemple, un widget représentant un bouton peut avoir un attribut *text* qui contient le texte du bouton. Un attribut peut

changer le comportement d'un widget, c'est le cas de l'attribut *enable*. Un bouton avec l'attribut *enable* réglé sur *false* représente un bouton sur lequel on ne peut pas cliquer. Enfin, les widgets peuvent avoir des attributs qui ont un impact sur l'aspect visuel de l'application. Ce type d'attribut permet de définir une mise en page à respecter par les widgets contenus dans le widget principal et peut éventuellement modifier les dimensions de ce dernier pour respecter une disposition particulière.

5 Implémentation

Pour tester notre approche, nous avons implémenté un outil de migration. Il est implémenté en Pharo [3] et le méta-modèle est représenté à l'aide de la plateforme Moose [4].

5.1 Étude de cas

Les applications chez Berger-Levrault (notre partenaire industriel) sont basées sur le framework BLCore. Ce framework se compose de 763 classes dans 169 paquetages. Il est développé par la société comme extension de GWT et définit les widgets que les développeurs doivent utiliser, l'aspect visuel par défaut des applications, et les classes Java pour connecter le front-end de l'application au back-end. Il encourage également certaines conventions de développement.

Afin d'adapter notre approche aux applications de Berger-Levrault, nous ajoutons une nouvelle entité (**Business Page**) au méta-modèle GUI présenté Section 4.2. C'est une sorte de **Container**. Une convention est que chaque **Page** a une ou plusieurs **Business Pages** représenté comme onglets dans la **Page**. Les **widgets** (boutons, champs texte, tables, ...) sont inclus dans les **Business Pages**, jamais directement dans la **Page**.

5.2 Importation

En partie à cause de la complexité de la mise en place d'un outil pour capturer automatiquement tous les écrans de ces grandes applications web, nous comptons sur l'analyse statique pour créer notre modèle. Les résultats obtenus jusqu'à présent semblent indiquer qu'elle sera suffisante.

Tel que présenté Section 4.1, la création du modèle d'interface graphique est divisée en deux étapes : l'extraction du modèle de code source et l'extraction du modèle GUI. Pour le méta-modèle de code source, nous utilisons le méta-modèle Java de Moose [5, 21] qui vient avec un extracteur de modèle Java [5]. La Figure 5 présente un extrait du code source d'une application de Berger-Levrault. Il montre la méthode buildPageUi(Object object) qui construit l'interface graphique de la "business page" SPMetier1 (une "page métier simple").

```
1  class SPMetier1 extends AbstractSimplePageMetier
       {
2    @Override
3    public void buildPageUi(Object object) {
4      BLLinkLabel lblPg = new BLLinkLabel("Next");
5      lblPg.addClickHandler(new ClickHandler(){
6        public void onClick(ClickEvent event) {
7          SPMetier1.this.fireOnSuccess("param");
8        }
9      });
10     lblPg.setEnabled(false);
11     vpMain.add(new Label("<Business content>"));
12     vpMain.add(lblPg);
13     super.setBuild(true);
14   }
15 }
```

FIGURE 5 – Création de l'interface graphique en GWT

Pour la deuxième étape de l'extraction, notre outil crée le modèle GUI à partir du modèle de code source et une analyse du fichier de configuration XML. Les entités que nous voulons extraire sont d'abord les **Pages**. Nous analysons le fichier de configuration XML dans lequel est définie l'information sur les pages (voir Section 3.2). Il fournit pour chaque **Page** (appelé *phase* dans le fichier XML, Figure 3) son nom et le nom de la classe Java qui le définit. Ensuite, l'outil recherche les **Widgets**.

Tout d'abord, l'outil détermine les widgets disponibles. Pour ce faire, il collecte toutes les sous-classes Java de la classe GWT Widget. Pour les **Business Pages**, l'outil recherche les classes qui implémentent l'interface IPageMetier. Ensuite, l'outil regarde où les constructeurs des **Widgets** sont appelés et crée les liens entre les **Widgets** et leurs **Widget** parents. Dans la Figure 5, il y a deux appels aux constructeurs d'un**Widget** : ligne 4, le constructeur de BLLinkLabel est appelé, et ligne 11, le constructeur de Label. La variable vpMain correspond au panneau principal de la **Business Page**. Les lignes 11 et 12 correspondent à l'ajout d'un widget dans le panneau principal grâce à la méthode add().

Enfin, pour détecter les attributs et les actions qui appartiennent à un widget, l'outil détecte dans quelle variable Java le widget a été affecté. Ensuite, il recherche les méthodes invoquées sur cette variable. Si un widget invoque la méthode *"addClickHandler"*, il crée un événement. S'il invoque une méthode *"setX"*, il crée un attribut. Ces heuristiques ont été trouvées dans la littérature [22, 25]. Dans la Figure 5, le BLLinkLabel, dont la variable est lblPg, est lié à un événement et à un attribut. Les lignes 5 à 9 correspondent à la création d'un événement avec le code exécutable. La ligne 10 correspond à l'ajout de l'attribut enabled. avec la valeur false.

3. Pharo est un langage de programmation orienté objet inspiré de Smalltalk. http://pharo.org/
4. Moose est une plateforme d'analyse de logiciels et de données. http://www.moosetechnology.org/
5. verveineJ https://github.com/moosetechnology/verveineJ

5.3 Exportation

Une fois le modèle GUI généré, il est possible d'exporter l'application. Pour générer le code de l'application cible, l'outil inclut un exportateur. L'exportateur crée les dossiers de l'application cible et les fichiers de configuration. Puis, il visite les pages. Pour chaque **Page**, l'exportateur crée un sous-projet Angular sous la forme d'un répertoire contenant plusieurs fichiers de configuration et une page web vide par défaut. Ensuite, pour chaque business page de la **Page** visité, l'exportateur génère un fichier HTML et un fichier TypeScript. Pour le fichier HTML, l'exportateur construit le DOM grâce au patron de conception Composite utilisé dans le méta-modèle GUI (voir Section 4.2). Chaque widget fournit ses attributs et actions à l'exportateur.

6 Validation

Dans cette section, nous décrivons l'application industrielle sur laquelle nous avons utilisé notre outil pour valider notre approche. La Section 6.1 présente l'application industrielle. La Section 6.2 présente les métriques que nous avons utilisées pour évaluer notre approche.

6.1 Application industrielle

Nous avons expérimenté notre approche sur l'application *kitchensink* de Berger-Levrault. Ce logiciel, dédié aux développeurs, a pour but de regrouper en une seule et même application l'ensemble des composants disponible pour construire une interface graphique. Cette application est plus petite qu'une application de production mais utilise le framework BLCore. Le framework de l'entreprise nous garantit que le fonctionnement de l'application *kitchensink* est exactement le même que les applications industrielles. L'application contient 470 classes Java et représente 56 pages web. Bien que ce soit l'application de démonstration pour les développeurs, l'application *kitchensink* contient des irrégularités dans le code.

Notez que l'application *kitchensink*, comme les autres applications industrielles de la société, n'a pas de test. Il n'est donc pas possible d'utiliser des tests pour valider la migration.

6.2 Métriques de validation

La validation se fait en trois étapes : premièrement, nous vérifions les contraintes définies dans la Section 3.1 ; deuxièmement, nous validons que toutes les entités d'intérêt de l'interface graphique sont extraites et correctement extraites ; troisièmement, nous validons que nous pouvons réexporter ces entités dans Angular et que le résultat est correct.

Pour la première validation, nous identifions et comptons manuellement les entités de l'application *kitchensink* et comparons les résultats obtenus avec l'outil. Notre analyse porte sur la migration de trois entités : **Pages**, **Business Pages** et **Widgets**

- **Pages.** À partir du fichier de configuration XML de l'application, nous comptons manuellement 56 pages. Ce fichier de configuration fournit également le nom de chaque page.
- **Business Pages.** Comme expliqué précédemment, les business pages correspondent à un concept propre à Berger-Levrault. Ils sont définis dans le framework BLCore comme une classe Java qui implémente l'interface IPageMetier. Grâce à cette heuristique, nous comptons manuellement 76 instances **Business Page** dans l'application originale.
- **Widgets.** Dans l'état de l'art, nous n'avons pas trouvé de moyen automatique d'évaluer la détection de widgets. La vérification de tous les widgets de l'application serait longue et sujette aux erreurs car il y en a des milliers. Comme solution de repli, nous prenons un échantillon des pages de l'application *kitchensink* et comptons les widgets dans le DOM de ces pages. Nous considérons un échantillon de 6 **Pages** qui représente un peu plus de 10% des **Pages** de l'application. Ces **Pages** sont de différentes tailles et contiennent différents types de widgets. Au total, nous avons trouvé 238 **Widgets** dans ces 6 **Pages**. Pour avoir une idée plus précise de la représentativité de notre échantillon, nous comptons aussi le nombre de création de **Widgets** (*i.e.* `new AWidgetClass()`) dans le code. Il existe 2 081 créations de ce type. Cela peut ne pas représenter le nombre exact de widgets dans l'application entière, mais c'est une bonne estimation. Nous notons que le nombre de **Widgets** dans notre échantillon (un peu plus de 10 % des pages) représente également un peu plus de 10 % de notre estimation du nombre total de widgets.
  Pour l'évaluation, nous vérifions également que les **Widgets** sont correctement placés dans le DOM de l'interface graphique (*i.e.* ils appartiennent au bon **Container** dans le modèle GUI).

Dans nos résultats, nous ne considérons que le rappel de l'outil car la précision est toujours de 100 % (il n'y a pas de faux positif). C'est un signe que le framework BLCore fournit des heuristiques claires (sinon complètes) pour identifier les entités.

Pour la deuxième validation, nous vérifions que les entités sont correctement exportées. Dans l'application Angular, chaque **Page** correspond à un sous-projet et est représentée par un dossier. Le nom du dossier doit correspondre au nom de la **Page**. Les **Business Pages** sont représentées par un sous-dossier dans le projet de la **Page**. Les noms doivent également correspondre à ce niveau.

Nous vérifions également visuellement que le **Page** exporté "ressemble" à l'original. Il s'agit d'une évaluation

subjective, et nous cherchons des options pour l'automatiser à l'avenir.

7 Résultats

Cette section présente les résultats de la validation de la migration de l'application *kitchensink* de Berger-Levrault. La Section 7.2 résume les résultats de l'extraction. Dans la Section 7.1, nous confrontons le résultat exporté avec les contraintes définies Section 3.1.

7.1 Satisfaction des contraintes

Nous avons défini les contraintes suivantes Section 3.1 : *Depuis GWT vers Angular*, *Adaptabilité de l'approche*, *Préservation du visuel*, *Conservation de la qualité du code*, and *Automatique*.

Notre outil peut utiliser du code Java en entrée et générer du code Angular. Le code exporté est compilable et exécutable. L'application cible peut être affichée. Nous pouvons donc confirmer que notre outil remplit la contrainte depuis GWT vers Angular.

Notre outil est utilisable sur d'autres technologies cibles source. Nos heuristiques ont été conçues pour être faciles à adapter, un utilisateur de notre outil peut ainsi ajouter un nouveau type de widget pour les phases d'importation ou d'exportation. Nous décrivons brièvement une petite expérience dans ce sens Section 8.5. Ces possibilités répondent à la contrainte d'adaptabilité.

Les contraintes *Conservation de la qualité du code* et *Préservation du visuel* sont discutées dans la Section 7.3, dans les résultats de la troisième validation.

Enfin, les résultats décrits ici ont été obtenus automatiquement en applicant notre outil sur l'application de Berger-Levrault. Ceci valide la dernière contrainte.

7.2 Résultats de l'extraction

La Table 2 résume les résultats de l'extraction.

TABLE 2 – Résultats de l'extraction

	Pages	**Business Pages**	**Widgets** (sample)
Nombre	56	76	238
Correctement importé	100%	100%	89%

L'outil a extrait 56 **Pages** de l'interface graphique originale. Cela correspond au nombre de pages définies dans le fichier de configuration de l'application *kitchensink*.

L'outil a extrait 76 **Business Pages**. Cette valeur correspond exactement au nombre de business pages de l'application d'origine. De plus, l'outil assigne correctement chaque **Business Page** à sa **Page**.

Nous avons obtenu 100 % des **Widgets** sur l'échantillon évalué qui ont été correctement détectés. Cependant, 27 des 238 **Widgets** de notre échantillon (11%) n'ont pas été correctement affectés à leur conteneur. Tous ces problèmes viennent d'une seule et unique **Page** (contenant 75 **Widgets** au total).

7.3 Résultats à l'exportation

Nous avons vérifié manuellement le nom des 56 pages exportées. Elles conservent toutes leur nom d'origine.

(a) GWT original (b) Angular migration

FIGURE 6 – Comparaison du visuelle de la migration d'une **Page**

Figure 6 présente les différences visuelles entre la version originale (GWT), à gauche, et celle migrée (Angular 6), à droite. On voit qu'il y a peu de différences. Dans la version exportée, la couleur de l'en-tête est un peu plus claire, et les lignes sont un peu plus éloignées.

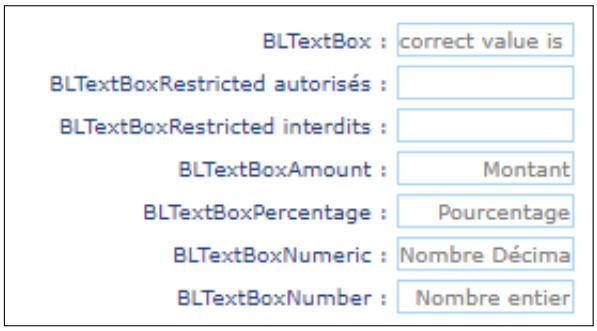
Figure 7 présente les différences visuelles pour la **Page** *Input box*. De nouveau sur le côté gauche il y a la **Page** original et sur le côté droit la même **Page** après la migration. Comme les deux images sont grandes, nous les avons rognées pour afficher cette zone d'intérêt. Même si les deux images semblent complètement différentes, tous les widgets sont présents dans la version migrée. Les différences visuelles sont dues à un problème dans la gestion de la mise en page. La contrainte visuelle est ainsi partiellement satisfaite. Ce point est discuté Section 8.1.

8 Discussion

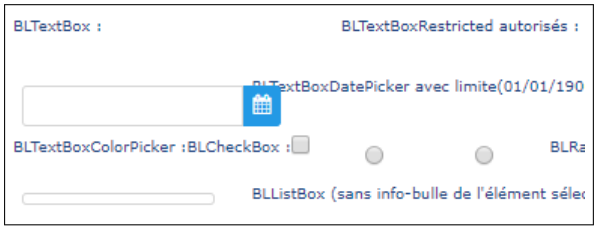
La Section 8.1 et la Section 8.2 présentent deux parties de l'interface utilisateur sur lesquelles nous n'avons pas travaillé. La Section 8.3 discute de l'impact du choix de *kitchensink* comme étude de cas. La Section 8.4 met en évidence les difficultés de validation à grande échelle de notre outil. Enfin, la Section 8.5 traite de l'impact du framework BLCore.

8.1 Gestion de la mise en page

Comme indiqué dans la Section 7.3, l'exportation peut donner des résultats incorrects à cause de problèmes de mise en page (Figure 7). Cela est dû à la représentation de la mise en page dans notre méta-modèle d'interface graphique. Actuellement, les modèles sont représentés dans notre méta-modèle d'interface graphique sous forme d'attribut sur un **Widget Conrtainer** définissant si les enfants de ce widget sont placés l'un à côté de l'autre ou l'un en dessous de l'autre (*i.e.* Mise en page verticale ou horizontale). Cependant, il existe beaucoup d'autres mises en page [14]. Par exemple, le framework BLCore fournit le widget



(a) GWT original



(b) Angular migration

FIGURE 7 – Comparaison du visuelle de la migration d'une **Page** : Tous les **Widgets** sont migrés mais avec un mauvais layout

`BLGrid`, un **Widget** héritant de la classe GWT `Grid` et implémentant une mise en page de grille. Actuellement, les mises en page complexes ne sont pas prises en compte dans notre méta-modèle d'interface graphique.

Une solution est proposée par Sánchez Ramán *et al.* [23]. Ils ont conçu un méta-modèle de mise en page. L'idée consiste à lier des widgets à une mise en page et à combiner les mises en page pour créer une représentation précise. Les auteurs ont défini un sous-ensemble de mise en page possible à connecter aux widgets.

De plus, avec de telles mises en page, la position des **Widgets** enfants peut être calculée au moment de l'exécution. Par exemple, dans une grille, les enfants peuvent être positionnés en fonction des valeurs de certaines variables `ligne` et `colonne`. Trouver ces valeurs est une tâche compliquée avec une analyse statique. C'est un cas où une approche hybride pourrait être nécessaire.

8.2 Gestion du code comportemental et métier

Actuellement, seule la partie visuelle de l'interface graphique est migrée. Pour prendre en compte l'ensemble de l'application, les migrations du *code comportemental* et du *code métier* (voir Section 3.3) sont nécessaires. Le *code comportemental* représente les interactions de l'utilisateur (clic, glisser-déposer, survoler, ...) et les structures de contrôle (*i.e.* loop et alternative). Dans le cas d'une application client/serveur, les requêtes au serveur font partie du *code comportemental*, alors que la requête en elle-même et les données appartiennent au *code métier*.

8.3 Application demo

Bien que les résultats soient encourageants, nous avons évalué notre outil sur l'application *kitchensink*. L'application *kitchensink* est un bon terrain d'entraînement pour notre outil car elle contient toute les types de widgets que les développeurs ont à leur disposition et la façon de les utiliser. Cependant, elle peut s'écarter des applications de production car elle doit contenir moins d'irrégularités ou d'astuces de codage que ces dernières.

8.4 Outils de validation

La validation automatique de la migration est actuellement un problème non résolu. Il est possible de vérifier manuellement le résultat de la migration pour quelques pages mais il est préférable de le faire automatiquement pour des centaines de pages (plus de 400 sur les applications de Berger-Levrault).

Nous n'avons trouvé, dans la littérature, que peu d'approches envisageant une validation visuelle automatique. Dans deux articles [11, 23], les auteurs comptent simplement le nombre de widgets dans les applications sources et les applications cibles. Mais nous avons vu dans Figure 7 que cela ne garantit pas la similarité visuelle. Un autre article [19] propose de comparer les captures d'écran de l'application originel et de l'application exportée pixel par pixel. Cependant, nous avons vu dans Figure 6 que des écrans aux différences à peine distinguables peuvent avoir des différences au niveau des pixels.

8.5 Impact de BLCore

Comme expliqué dans la Section 5.1, les applications de Berger-Levrault sont basées sur le framework BLCore. En spécialisant GWT, BLCore fournit des widgets spécifiques et une API dédiée. Cela peut ou non avoir un impact sur notre approche. Pour évaluer ce possible impact, et aussi pour valider la généralité de notre approche, nous avons réalisé deux petites expériences considérant (i) Spec (un framework d'interface utilisateur de bureau dans Pharo[6]) comme framework source et, (ii) Seaside (un framework Web dans Pharo [4] – également décrit à seaside.st) comme framework cible. Ces expériences prennent donc en compte différents langages de programmation (Pharo au lieu de Java (GWT) et TypeScript), différents frameworks d'interface graphique et différentes applications web et de bureau. Nous avons expérimenté la migration de l'interface graphique de l'application de démonstration de Spec vers Angular, et la migration de l'application *kitchensink* de Berger-Levrault vers Seaside.

Certaines conclusions sont :

— Il était plus difficile d'importer du code Spec que GWT à cause d'une plus grande variabilité dans la définition de l'interface graphique. Nous concluons que le framework BLCore a facilité notre travail sur l'importation en normalisant la façon de construire les pages.
— Pour Seaside, il était facile de migrer des widgets simples (*e.g.* Label, Button, Panel), mais le framework BLCore définit également des widgets complexes sans équivalent direct dans Seaside. Une bibliothèque similaire à BLCore devrait être définie dans Seaside pour faciliter la migration.
— La capacité de migration de notre méta-modèle d'interface graphique et l'extraction en deux étapes (d'abord, l'extraction du code source, puis l'extraction du modèle d'interface graphique, voir Figure 4) est validée par le fait que nous avons pu migrer une application de bureau Pharo avec peu de travail supplémentaire.

9 Conclusion et travaux futurs

Nous avons créé un outil avec des résultats prometteurs sur la représentation de l'interface graphique pour migrer des applications GWT vers Angular. Dans ce qui suit, nous concluons la présentation de ce travail et proposons quelques pistes de recherche que nous voulons explorer.

9.1 Conclusion

Dans cet article, nous avons exposé un travail préliminaire sur le problème de la préservation visuelle et le respect de l'architecture cible lors de la migration de l'interface graphique d'une application. Nous avons proposé une approche basée sur un méta-modèle d'interface graphique et un processus de migration en trois étapes. Nous avons implémenté ce processus dans un outil pour effectuer la migration des applications GWT vers Angular 6. Ensuite, nous avons validé notre outil avec une expérience sur une application de démonstration (*kitchensink*). Nous avons pu extraire correctement toutes les pages de l'application et 89 % des widgets. L'application migré a le même visuel que l'application originelle tant que des widgets complexes (*e.g.* GridLayout) ne sont pas utilisés. Notre prochain défi sera de faire face aux problèmes de mise en page.

Notre solution nous permet également de respecter les conventions de nommage utilisées dans l'application source ainsi que la structure du code dans la mesure où les différences entre les frameworks GUI le permettent.

9.2 Travaux futurs

Afin d'améliorer la migration d'une interface graphique, nous allons améliorer notre méta-modèle et notre outil pour supporter la gestion des mises en page et du code comportemental et métier.

Nous n'avons pas trouvé d'approche ou de métriques pour évaluer automatiquement la validité des écrans migrés. Il est donc important de trouver une nouvelle façon d'évaluer si les écrans migrés préservent l'aspect visuel des écrans originaux.

Avoir un bon méta-modèle d'interface graphique ouvre également la porte à un constructeur d'interface graphique générique qui pourrait la créer dans plusieurs frameworks d'interface graphique.

Références

[1] Marco Brambilla and Piero Fraternali. *Interaction flow modeling language : Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[2] John Brant, Don Roberts, Bill Plendl, and Jeff Prince. Extreme maintenance : Transforming Delphi into C#. In *ICSM'10*, 2010.

[3] Jonathan Cloutier, Segla Kpodjedo, and Ghizlane El Boussaidi. WAVI : A reverse engineering tool for web applications. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–3. IEEE, 2016.

[4] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone, and Michael Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010.

[5] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0 : an Interexchange Format and Source Code Model Family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[6] Johan Fabry and Stéphane Ducasse. *The Spec UI Framework*. Square Bracket Associates, 2017.

[7] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jezéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735, pages 482–497, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Kelly Garcés, Rubby Casallas, Camilo Álvarez, Edgar Sandoval, Alejandro Salamanca, Fredy Viera, Fabián Melo, and Juan Manuel Soto. White-box modernization of legacy applications : The oracle forms case study. *Computer Standards & Interfaces*, pages 110–122, October 2017.

[9] Zineb Gotti and Samir Mbarki. Java swing modernization approach - complete abstract representation based on static and dynamic analysis :. In *Proceedings of the 11th International Joint Conference on Software Technologies*, pages 210–219. SCITEPRESS - Science and Technology Publications, 2016.

[10] Tomokazu Hayakawa, Shinya Hasegawa, Shota Yoshika, and Teruo Hikita. Maintaining web applications by translating among different ria technologies. *GSTF Journal on Computing*, page 7, 2012.

[11] Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering iOS mobile applications. In *2012 19th Working Conference on Reverse Engineering*, pages 177–186. IEEE, 2012.

[12] R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models : Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998.

[13] Valéria Lelli, Arnaud Blouin, Benoit Baudry, Fabien Coulon, and Olivier Beaudoux. Automatic detection of GUI design smells : The case of blob listener. *EICS '16 Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 12, 2016.

[14] Simon Lok and Steven Feiner. A survey of automated layout techniques for information presentations. *Proceedings of SmartGraphics*, 2001 :61–68, 2001.

[15] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping : reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269. IEEE, 2003.

[16] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3) : 137–157, 2007.

[17] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1) :1–30, 2012.

[18] Moore, Rugaber, and Seaver. Knowledge-based user interface migration. In *Proceedings 1994 International Conference on Software Maintenance*, pages 72–79. IEEE Comput. Soc. Press, 1994.

[19] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. Detecting and Summarizing GUI Changes in Evolving Mobile Apps. *arXiv :1807.09440 [cs]*, July 2018.

[20] I Coimbra Morgado, Ana Paiva, and J Pascoal Faria. Reverse engineering of graphical user interfaces. In *ICSEA 2011 : The Sixth International Conference on Software Engineering Advances*, 2011.

[21] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose : an agile reengineering environment. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press.

[22] Hani Samir, Amr Kamel, and Eleni Stroulia. Swing2script : Migration of Java-Swing applications to Ajax Web applications. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007.

[23] Óscar Sánchez Ramán, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2) :147–186, 2014.

[24] Eeshan Shah and Eli 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*, pages 255–260. ACM, 2011.

[25] João Carlos Silva, Carlos C. Silva, Rui D. Goncalo, João Saraiva, and José Creissac 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*, pages 181–186. ACM Press, 2010.

[26] Stefan Staiger. Reverse engineering of graphical user interfaces using static analyses. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 189–198. IEEE, 2007.

[27] Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring. On the modernization of explorviz towards a microservice architecture. In *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*. CEUR Workshop Proceedings, 2018.