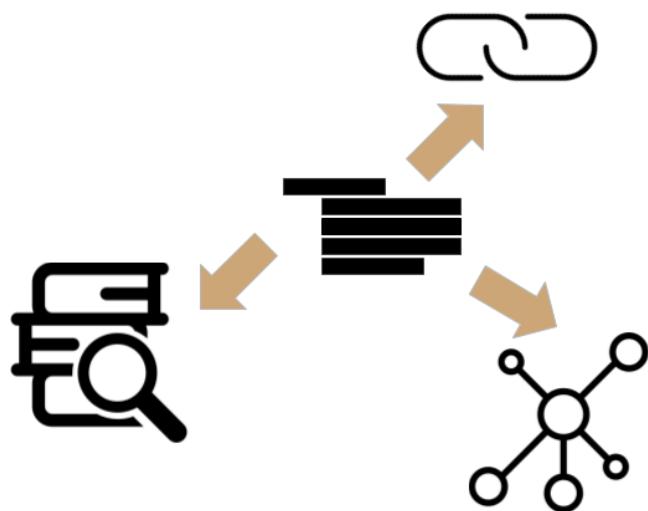


Benoît Verhaeghe

## Étude d'une application GWT



Tuteurs : Mme Etien, M. Anquetil

Inria Lille Nord Europe - RMoD  
février 2018

# Table des matières

<b>1</b>	<b>Contexte</b>	<b>2</b>
1.1	Contexte général . . . . .	2
1.2	Problématique . . . . .	2
1.3	Description du problème . . . . .	2
<b>2</b>	<b>État de l'art</b>	<b>4</b>
2.1	Migration de plateforme . . . . .	4
2.2	Migration de librairie . . . . .	4
<b>3</b>	<b>Modélisation</b>	<b>5</b>
3.1	Structure d'une application . . . . .	5
3.2	Outils d'analyse . . . . .	7
3.3	Visualisation . . . . .	7
3.4	Travail futur . . . . .	9
<b>4</b>	<b>L'Adhérence</b>	<b>10</b>
4.1	Adhérences internes à un framework . . . . .	10
4.1.1	Complexité d'un widget . . . . .	10
4.1.2	Complexité d'un ensemble de widgets . . . . .	11
4.1.3	Migration par couche ou groupe . . . . .	12
4.2	Adhérence entre framework . . . . .	13
4.2.1	Framework vers Framework . . . . .	13
4.2.2	Classe d'un framework vers les autres . . . . .	14
4.3	Travail à faire sur l'adhérence . . . . .	15
<b>5</b>	<b>Conclusion</b>	<b>16</b>
	<b>Bibliographie</b>	<b>17</b>

# 1 Contexte

## 1.1 Contexte général

Ce rapport est tiré de mon projet de fin d'étude à Polytech Lille. Il s'agit d'un projet de recherche 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.

## 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 4. Le back-end est une application monolithique et doit évoluer vers une architecture de services Web. Le changement de framework<sup>1</sup> graphique est imposé par l'arrêt du développement de GWT par Google et le problème de compatibilité arrière entre Angular 4 et Angular 3. Le passage à une architecture à services est aussi souhaité pour améliorer l'offre commerciale et la rendre plus flexible.

Pour ce projet de fin d'études, je n'ai dû traiter que le passage de GWT à Angular.

## 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.

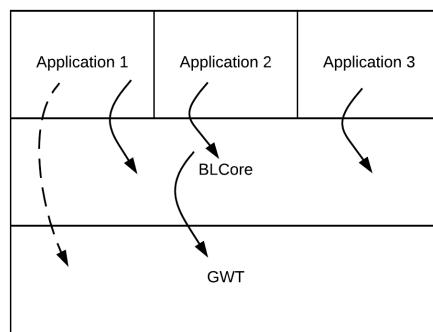


FIGURE 1 – Structure application

Les applications de Berger-Levrault sont complexes. Ce sont les plus importantes application 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

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).

tâche couteuse 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 leur taille mais aussi et surtout dans le changement de langage. En effet, les applications sont développées intégralement en Java tout comme le framework GWT. Or, pour utiliser Angular 4, le programme doit être écrit en TypeScript. La question qui se pose est de savoir si nous pouvons garder le code source Java et, si c'est le cas, celui correspondant à quelles couches de la Figure 1 ?

J'ai trouvé trois manières d'effectuer la migration :

- Réécrire l'ensemble des applications de Berger-Levrault pour qu'elles utilisent directement du TypeScript. Les problèmes ici sont qu'il existe beaucoup d'applications et que le code métier est écrit en Java. Il faudra donc de toutes façons faire co-exister Java et TypeScript. De plus, même si cette solution est choisie, il faudra aider les développeurs à migrer le code Java des parties graphiques en code TypeScript de façon semi-automatisée pour faciliter la migration.
- Migrer la couche BLCore puis migrer les applications en utilisant cette couche. Ce sera alors la couche BLCore qui fera le lien entre Java et TypeScript.
- Continuer à développer en Java, et modifier le générateur de GWT pour en créer un qui génère du TypeScript plutôt que du HTML, CSS et Javascript.

Mon objectif est donc d'évaluer la complexité de chacune de ces solutions. Pour cela, j'ai dans un premier temps étudié la structure d'une application de Berger-Levrault. Puis, j'ai voulu estimer la complexité de migrer la couche BLCore. Caractériser les liens entre deux frameworks est alors une étape qui pourrait aider les développeurs dans leur travail. C'est ce que j'appelle l'étude de l'adhérence entre deux frameworks. Quelles classes je dois migrer ? Par laquelle je dois commencer ? Sont les questions auxquelles j'ai essayé de répondre.

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 plusieurs centaines de classes.

## 2 État 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 application d'un langage à un autre. On y aborde par exemple la migration de plateforme pour des interfaces graphiques. 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 XUL<sup>2</sup>. 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.

### 2.2 Migration de librairie

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

Teyton *et al.* [2] 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.* [3] 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.* [4] ont voulu créer une approche visant à faire correspondre l'API<sup>3</sup> 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.

L'article de Phan *et al.* [5] propose 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

---

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

3. 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.

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.

## 3 Modélisation

La création des outils d'analyses est la tâche qui m'a pris le plus de temps. Elle est divisée en deux parties. Une première partie consiste à décrire une application de Berger-Levrault. La seconde partie correspond à la création des visualisations à partir de la structure d'une application.

Berger-Levrault m'a fourni leur application *bac-à-sable* afin de pouvoir tester mon travail sur une application à taille humaine et normalement construite suivant les mêmes principes que les applications en production.

### 3.1 Structure d'une application

Après analyse et discussion avec Berger-Levrault, j'ai pu extraire la structure générale de leur application *bac-à-sable*.

L'application *bac-à-sable* utilise les fichiers de configuration suivants :

- *application.module.xml* qui détaille l'ensemble des phases (pages web) de l'application.
- *coreincubator.gwt.xml* qui permet de définir les scripts et les css à utiliser dans l'application. Le fichier décrit aussi des redéfinitions de classes qui se font au moment du preprocessing.

Ci-dessous, la classe *BLUserPreferencesNull* sera remplacée durant le preprocessing par *BLUserPreferences*. Berger-Levrault utilise ce type de fichier pour définir le comportement de ses logiciels à la compilation et non via des options sélectionnables par l'utilisateur.

```
<replace-with class="fr.bl.client.core.refui.base.gwt.BLUserPreferencesNull">
    <when-type-is class="fr.bl.client.core.refui.base.gwt.BLUserPreferences" />
</replace-with>
```

La Figure 2 présente une page web de l'application *bac-à-sable*. J'en détaille la structure ci-dessous et je fais correspondre les éléments de l'application avec la Figure 2.

Les applications de Berger-Levrault sont composées de **Phases** qui correspondent aux pages web auxquelles nous pouvons accéder. Dans la Figure 2, la phase correspond à la partie encadrée en noir. Les phases sont représentées par une classe Java et sont identifiables grâce au fichier *application.module.xml* qui les nomme. Le code ci-dessous correspond à la déclaration de la phase *PHASE\_MAQUETTE\_I18N* qui est liée à la classe *fr.bl.client.coremaquette.PhaseMaquetteI18n*.

```
<phase codePhase="PHASE_MAQUETTE_I18N"
    codeValue="GEN_I18N" className="fr.bl.client.coremaquette.PhaseMaquetteI18n"
    raccourci="I18" titre="I18n" description="Exemple d'internationalisation." />
```



FIGURE 2 – Page web de l’application bac-à-sable de Berger-Levrault

Une phase ne contient pas l’ensemble du code correspondant à une page web, mais contient des **Pages Métiers**. Les pages métiers sont identifiables dans le programme via les classes qui implémentent *IPage-Metier*. Ce sont les pages métiers qui vont décrire une portion d’une page web. Dans la Figure 2, il n’y a qu’une page métier. Elle est encadrée en rouge.

Certaines phases sont dites *stand-alone*. Cela signifie qu’elles ne contiennent pas de page métier et décrivent seules la totalité des pages web auxquelles elles correspondent.

Les phases stand-alone et pages métiers peuvent contenir des instances de **Widget**. Il s’agit des composants graphiques que l’on peut voir sur la page web (*i.e.* les boutons, les tableaux, les listes, etc.) et des objets qui définissent le placement des composants sur une interface graphique (*i.e.* les panels). J’ai mis en évidence, en vert, certains widgets dans la Figure 2 (un tableau, un label, des liens). Tous les éléments graphiques de la page web présenté Figure 2 sont des widgets. Les widgets sont définis dans *BLCore*.

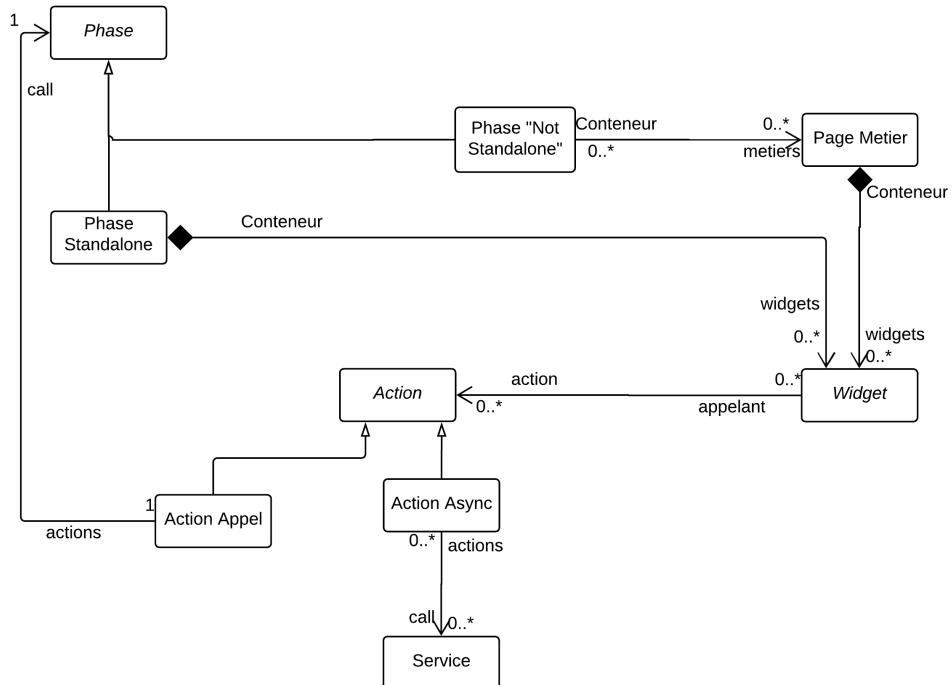


FIGURE 3 – Meta-Model d’un application de Berger-Levrault

Ces derniers peuvent avoir des **Actions**. A ce jour, j'ai travaillé sur deux types d'actions :

- les **actions d'appels** qui correspondent au changement d'une page. Par exemple le clic sur un lien hypertexte.
- les **actions asynchrones** qui font une requête sur un **Service** distant. Par exemple une liste qui se remplit après le chargement du site.

Je n'ai pas encore pris en compte les actions qui visent à effectuer des changements sur la page web courante. Par exemple, l'utilisateur est sur une page web qui contient un formulaire, et en fonction des choix qu'ils effectuent (comme cocher ou non une boîte à cocher) des éléments graphiques doivent s'afficher. Ce sont les actions dites **actions complexes**.

À partir de toutes ces informations, j'ai conçu un meta-modèle de l'application de Berger-Levrault. Comme on peut le voir Figure 3, on y retrouve les différents éléments composant une l'application.

### 3.2 Outils d'analyse

Une fois les différents concepts présents dans l'application et les liens existants entre eux identifiés, j'ai cherché à décrire les applications.

Pour ce faire, j'ai extrait les différentes éléments de l'application à partir du code source de l'application grâce à l'outil de modélisation Moose<sup>4</sup> (implémenté en Pharo<sup>5</sup>).

Linstanciation des concepts identifiés à la Figure 3, pour l'application *bac-à-sable* s'est faite par requêtage du code source. Par exemple une classe implémentant l'interface *IPageMetier* est une page métier.

J'ai trouvé un total de 56 phases et 57 pages métiers. Il y a aussi 4356 instanciations de widget au sein de l'application.

J'ai également étudié la complétude de mon travail. Toutes les phases et toutes les pages métiers sont détectées. Toutefois, il reste encore quelques problèmes dans la détection des instances de widgets et dans les liens entre ces derniers et les services ou phases à cause d'une association absente dans le meta-modèle que j'ai construit.

### 3.3 Visualisation

La Figure 4 présente la visualisation du modèle de l'application *bac-à-sable*. On y retrouve deux importantes composantes connexes et de nombreuses petites.

Les deux composantes devraient être reliées pour n'en former qu'une. Je n'ai pas encore réussi à extraire certaines informations du code source ce qui m'empêche d'ajouter les liens entre les composantes.

On voit dans la composante située en partie haute deux hubs (entourés en noir sur la Figure 4). En analysant le code source de l'application, j'ai retrouvé ces éléments. Ce sont les *home* de l'application, respectivement la *PhaseMaquetteHome* et *PhaseAccueilIncubator*.

Les liens gris représentent l'appartenance d'un élément à un autre (*i.e.* un lien gris allant d'une page métier à un widget signifie que le widget est contenu dans la page métier). Les *fleurs* de la Figure 4 sont donc la représentation de pages web. Elles contiennent la phase au centre et ses éventuelles pages métiers ainsi que les widgets qui les composent autour.

4. Moose est une plateforme pour l'analyse de logiciels et données.

5. Pharo est un langage de programmation objet, réflexif et dynamiquement typé

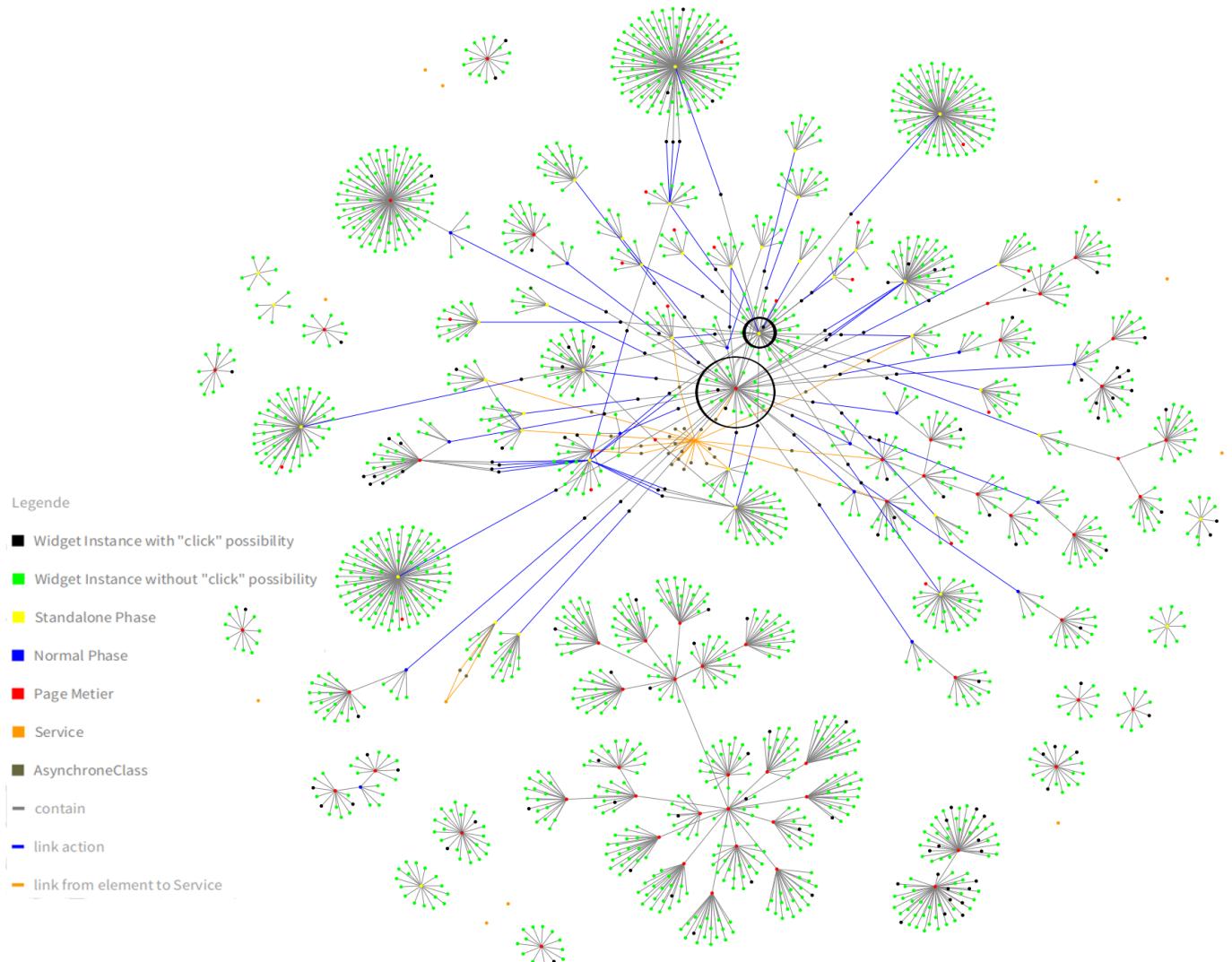


FIGURE 4 – Représentation de l'application bac à sable dans sa globalité

Les liens bleus représentent le passage à une autre page métier en utilisant un widget. Les fleurs sont donc logiquement reliées les unes aux autres par des liens bleu. Ce sont les actions d'appels.

Les éléments oranges représentent des services potentiellement appelés. On en retrouve beaucoup qui n'ont pas de lien vers l'application. J'ai vérifié et ils ne sont effectivement pas appelés.

Certaines fleurs ne sont reliées à des deux importantes composantes connexes. Ce sont des fleurs composées de classes utilitaires. Elles permettent de faire des tests.

Cette visualisation a été construite de façon itérative. Elle permet de mettre en évidence certaines erreurs, par exemple dans cette version il manque certains éléments dans le meta-modèle ou il y a une erreur dans la façon de construire le modèle de l'application *bac-à-sable*. En effet, la présence des deux importantes composantes connexes distinctes ne reflète pas la réalité. Il manque un lien entre les deux. Reste à trouver lequel et comment le mettre en évidence. C'est par observation des différentes visualisations obtenues que nous avons complété le meta-modèle initialement défini avec l'aide de Berger-Levrault, à partir de la connaissance des développeurs.

### 3.4 Travail futur

Une fois que toutes les composantes d'une application seront bien détectées et ajoutées au modèle, je pourrai faire une étude de scalabilité de cette solution sur des applications réelles de Berger-Levrault. Cela me permettra aussi de vérifier que le modèle que j'ai créé est valable pour toutes les applications de Berger-Levrault et que les visualisations que j'ai créées sont exploitables sur les applications en production.

Il y aura donc une étape de recherche sur les outils et visualisations que l'on peut construire au-dessus du modèle.

## 4 L'Adhérence

Si l'on souhaite migrer seulement une partie de l'application de Berger-Levrault, nous devons soit migrer l'ensemble de ses dépendances, soit les “*casser*”.

Une étude de la difficulté de supprimer les liens entre deux parties d'un programme est donc nécessaire pour évaluer la complexité de la migration de ces dernières. Chercher l'adhérence entre les éléments au sein des frameworks de Berger-Levrault et entre les frameworks utilisés par Berger-Levrault devrait me permettre d'évaluer la complexité à les séparer.

Mon hypothèse est que le nombre de référence entre deux éléments augmentent l'adhérence entre eux. Je suppose que s'il y a plusieurs références entre deux classes, le nombre de méthode concernées par ces références est aussi un facteur augmentant l'adhérence. L'héritage entre deux classes peut aussi représenter l'adhérence. Le nombre de méthodes dans les classes et le nombre de méthodes redéfinies impacteraient l'adhérence. J'ai essayé de confirmer ces hypothèses sur l'application *bac-à-sable*.

Je présente dans cette section plusieurs expériences réalisées pour tenter de caractériser l'adhérence entre l'application *bac-à-sable*, BLCore et GWT. Ces expériences ont apporté quelques informations mais n'ont pas été suffisamment concluantes. De nouvelles expériences seront nécessaires.

### 4.1 Adhérences internes à un framework

Une première question est quelle est l'adhérence entre les éléments au sein de BLCore. En effet, il s'agit de l'élément central de l'architecture écrite par Berger-Levrault. BLCore repose sur GWT et est utilisé par les applications de Berger-Levrault. Une solution à la migration de l'ensemble des applications serait de commencer par migrer BLCore, puis de migrer les applications.

Les classes essentielles de BLCore utilisées par les applications sont les classes définissant les widgets. C'est donc par l'étude de l'adhérence entre ces derniers que j'ai commencé l'exploration du framework.

#### 4.1.1 Complexité d'un widget

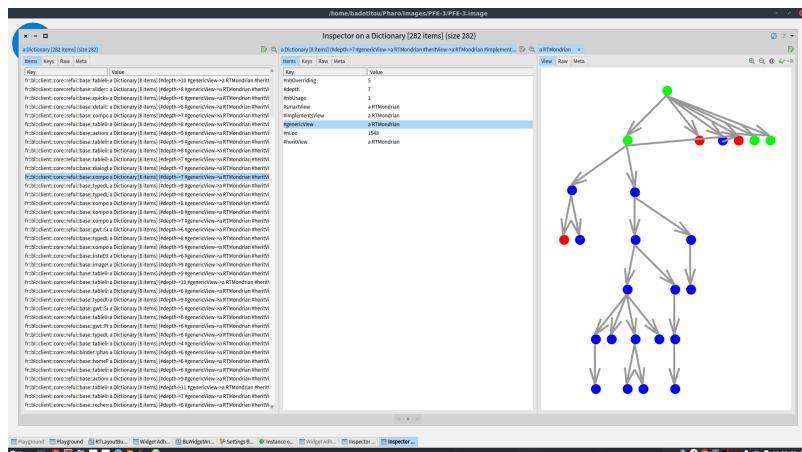


FIGURE 5 – Métriques pour les widgets

Ma première hypothèse était que l'adhérence est liée à la complexité du widget.

Un widget étant une classe, je me suis basé sur ce qui définit la complexité d'une classe. J'ai donc calculé pour chacune des classes son nombre de lignes de code, sa profondeur dans la hiérarchie de classes (combien de surclasses) et le nombre de classes qui lui font référence.

J'ai développé un outil permettant d'extraire des informations du modèle. Ensuite, j'ai créé un outil de visualisation de ces informations pour pouvoir facilement naviguer dans les résultats. La Figure 5 présente l'outil de visualisation que j'ai développé avec les informations que j'ai extraites pour le widget *BLChoixListeEtOrdre*. Le premier panneau permet de choisir un widget pour lequel je veux détailler les informations. Le deuxième affiche les métriques et le troisième affiche son schéma avec la hiérarchie de classes et les références. Les noeuds verts représentent les classes faisant partie de BLCore et impliquées dans la hiérarchie de classes, les bleus représentent celles de GWT qui sont impliquées dans la hiérarchie de classes. Les éléments rouges sont des classes qui sont référencées et peuvent faire partie de BLCore ou de GWT. Un lien d'un élément vert vers un élément rouge signifie que la classe "*verte*" référence la classe "*rouge*" par un appel de méthode ou un accès à un attribut.

J'ai dans un premier temps considéré l'implémentation d'interfaces comme un héritage.

Grâce à ces informations, nous pouvons certes définir de manière plus claire la complexité d'une classe, mais il est encore impossible de dire si une classe est "*adhérente*" à une autre et plus précisément je ne peux pas dire quelle est la classe la plus adhérente.

De plus, le schéma peut être amélioré notamment l'affichage de la hiérarchie de classe qui est présentée à l'inverse du sens habituel (de haut en bas plutôt que de bas en haut). Le visuel des liens devrait également évoluer pour permettre de distinguer selon qu'ils désignent un héritage ou une référence entre éléments. On devrait également changer la représentation des éléments provenant de BLCore ou GWT. En effet, une classe provenant de BLCore peut être représenté en rouge, si elle fait partie d'une relation de référence, ou en vert, si elle fait partie de la hiérarchie de classe.

#### 4.1.2 Complexité d'un ensemble de widgets

L'impossibilité de comparer un widget à d'autres peut venir d'un manque d'informations concernant leur possibles liens. En effet, je veux comparer des éléments pouvant être dépendants les uns des autres. J'ai donc décidé de regrouper tous les widgets sur un même schéma.

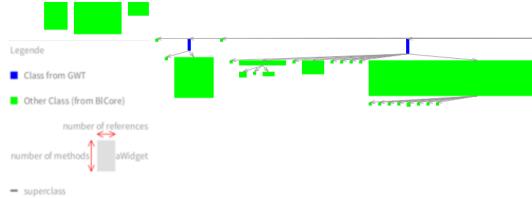


FIGURE 6 – Hiérarchie de classes des widgets

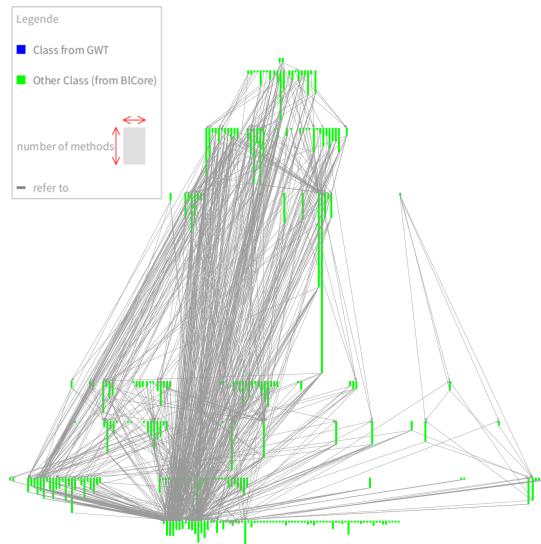


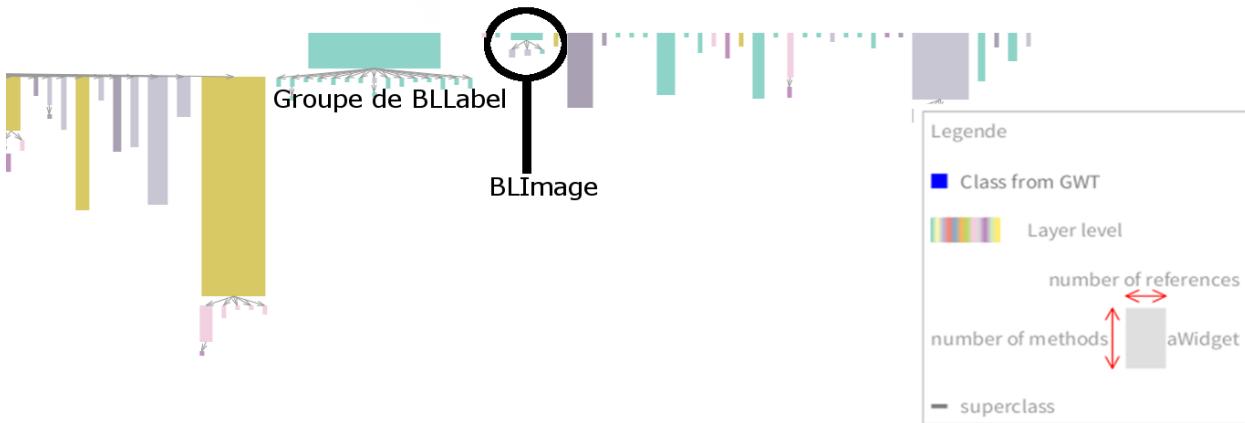
FIGURE 7 – Références entre widgets

La Figure 6 présente une partie du graphe de la hiérarchie de classes définissant les widgets. Chaque classe, et donc widget, est représentée par un rectangle. J'ai décidé de faire varier la hauteur des rectangles en fonction du nombre de méthodes utilisées pour définir le widget. De plus, la largeur change en fonction du nombre de références vers ce widget. Cette dernière valeur peut être biaisée à cause des références provenant de l'application *bac-à-sable*. Cependant, les deux métriques permettent d'évaluer la complexité d'une classe et donc d'un widget. Nous voyons plusieurs hiérarchie de classe. C'est ce que j'appelle des groupes. Ces groupes mettent en évidence l'héritage entre les widgets et cela peut être pris en compte dans le calcul de l'adhérence entre les widgets. La Figure 6 me permet donc de dégager des groupes de widget grâce à la hiérarchie de classe. Cela me permet de faire l'hypothèse qu'il faudrait migrer les widgets par groupe.

La Figure 7 présente les références entre les classes. J'ai représenté ces invocations entre classes grâce à un système de couches. Les classes ne faisant appel à aucune autre se trouvent dans la première couche (en bas du schéma). Si une classe invoque une autre classe, elle se trouvera dans la couche supérieur à celle de la seconde. On retrouve également pour chaque classe, en hauteur, le nombre de références à cette dernière. Comme pour la Figure 6, ce chiffre peut être biaisé par l'application *bac-à-sable*, mais nous avons néanmoins déjà un aperçu de la complexité des widgets. La largeur ici ne varie pas. La Figure 7 m'a permis de trouver huit couches de widgets. Grâce à cette information, j'ai émis l'hypothèse qu'il faudrait migrer les widgets par couches en commençant par la première car elle ne fait pas de références à une autre couche.

À partir de ces schémas, j'ai émis les hypothèses suivantes : Il est plus simple de commencer à migrer les classes qui se trouvent en bas de la Figure 7. Et, dans le cas de la migration d'une classe, il faudrait migrer groupe par groupe les widgets.

#### 4.1.3 Migration par couche ou groupe



Pour vérifier que la migration par couche ou groupe est plus simple, j'ai voulu regarder comment les groupes se positionnent par rapport aux couches et inversement. J'ai aussi changé la manière dont les couches sont calculées de façon à ce que les classes qui s'appellent entre elles, et donc forment un cycle, soient sur la même couche.

Dans la Figure 8, toutes les classes font partie de BLCore. Chaque couleur correspond à un groupe. On trouve des groupes où toutes les classes font partie de la même couche. Ce qui montre une interdépendance

entre les éléments qui proviennent des cycles que j'ai détectés. On retrouve, par exemple, les classes *BLImage*, *BLImageViewer* et *BLViewableImage* (nommées sur la Figure 8) ou toutes les classes de type *BLLabel* (entourées sur la Figure 8) qui font partie en même temps de la même couche et du même groupe. Cependant, cette seule information ne permet pas de conclure sur le niveau d'adhérence entre les éléments car nous ne disposons d'aucune métrique sur la complexité générée par l'héritage.



FIGURE 9 – Références entre widget

La Figure 9 montre les références entre les widgets et met en couleurs les widgets provenant du même groupe. Contrairement à la Figure 7, si il existe un cycle de dépendance entre les widgets, ils se trouveront dans la même couche. De cette manière, j'élimine les références seulement liées à la hiérarchie de classe entre les widgets. Ce schéma permet de voir si un groupe fait appel à un autre. Cependant, il ne permet pas de définir si une référence est fortement liée à l'adhérence. Lorsque que j'aurai défini plus précisément l'importance des liens, je pourrai alors confirmer ou non mes hypothèses et ainsi dire si la répartition par classe ou par groupe à un intérêt.

Les Figure 8 et Figure 9 ne m'ont pas permis de conclure sur l'adhérence entre les éléments d'un framework. Des expériences doivent être menées sur l'impact d'une référence entre deux éléments sur l'adhérence entre ces derniers.

## 4.2 Adhérence entre framework

La recherche des liens entre les classes internes à un BLCore ne m'ont pas permis de définir clairement ce qu'est l'adhérence ou comment l'évaluer. J'ai donc essayé de la caractériser en analysant les liens entre deux frameworks.

### 4.2.1 Framework vers Framework

Une première étude consiste à regarder les références entre les frameworks. J'ai pu créer des schémas affichant les références entre l'application *bac-à-sable*, BLCore et GWT.

Mon objectif est de trouver quelles classes adhèrent à quelles autres d'un autre framework. Quelles sont les classes les plus adhérentes d'un framework vers un autre ? Le but final étant de trouver les classes par lesquelles commencer la migration.

Les trois travaux que j'ai effectués m'ont permis de créer les Figure 10, Figure 11 et Figure 12.

La Figure 10 et la Figure 11 affichent une partie du graphe représentant les références entre l'application *bac-à-sable* et BLCore et entre BLCore et GWT. Les Figures ne sont pas lisibles car il y a trop d'éléments affichés. Chaque classe a une couleur en fonction du nombre de références qu'elle reçoit ou émet. La couleur est de plus en plus foncée si le nombre de références augmente. Les liens possèdent aussi une largeur qui

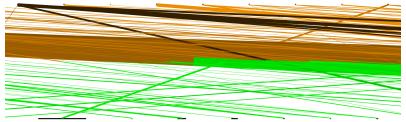


FIGURE 10 – Application *bac-à-sable* vers BLCore

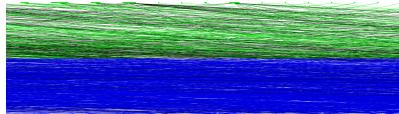


FIGURE 11 – BLCore vers GWT

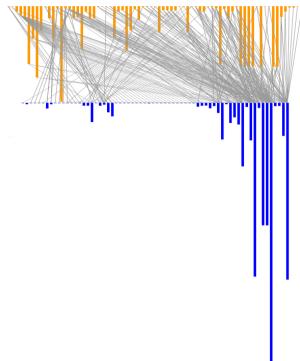


FIGURE 12 – *bac-à-sable* vers GWT

change en fonction du nombre de références qu'elle représente. S'il y a beaucoup de liens entre deux classes, le trait qui le représente est plus large. Enfin, un lien change de couleur en fonction du nombre de méthode utilisés pour faire les références. Comme pour les classes, plus la couleur est foncée, plus le nombre de méthodes impliqué est important. Le trop grand nombre d'éléments affichés sur le schéma ne permet pas de conclure sur l'adhérence entre les éléments. Une visualisation plus locale, avec seulement une classe pour laquelle on regarde les références, devrait m'apporter plus d'information.

La Figure 12 affiche les références entre l'application *bac-à-sable* et GWT. La hauteur des éléments représente le nombre total de références sortantes (pour les applications Berger-Levrault) ou entrantes (pour GWT). Cette figure ne nous permet pas de conclure sur l'adhérence. Elle donne néanmoins des informations sur l'application. En effet, d'après la structure de l'application que nous avons détaillée Figure 1, il ne devrait pas y avoir de références entre l'application *bac-à-sable* et GWT. Une étude de ces références mettra en évidence des différences entre ce que nous avions défini et la réalité du développement.

Le trop grand nombre d'informations obtenues ne permet pas de faire une visualisation globale simple à analyser et ne m'aide pas à définir clairement l'adhérence et comment l'utiliser pour comparer deux classes.

#### 4.2.2 Classe d'un framework vers les autres

La visualisation globale étant trop complexe pour apporter des résultats exploitables, j'ai décidé de créer des visualisations locales pour chercher à qualifier la complexité à migrer d'une seule classe.

J'ai donc créé le navigateur de la Figure 13. La partie de gauche me permet de sélectionner un widget parmi l'ensemble des widgets de BLCore. Le panneau de droite contient deux onglets. Ils permettent de sélectionner si l'on veut afficher les références entre le widget et l'application *bac-à-sable* ou entre le widget et GWT. Pour la Figure 13, c'est l'onglet “de application vers widget” qui est sélectionné.

Grâce au navigateur je peux étudier en détail les dépendances entre la classe d'un framework et d'un autre framework. Je peux donc trouver les classes les plus référencées et qualifier la complexité des références par leurs nombres et le nombre de méthodes impliquées dans les références des classes source et destination.

Cependant, je ne sais pas si le nombre de références et la qualification que j'ai faite sont réellement des facteurs complexifiant la migration d'une classe. Je dois effectuer manuellement une de ces migrations d'un composant simple pour l'évaluer.

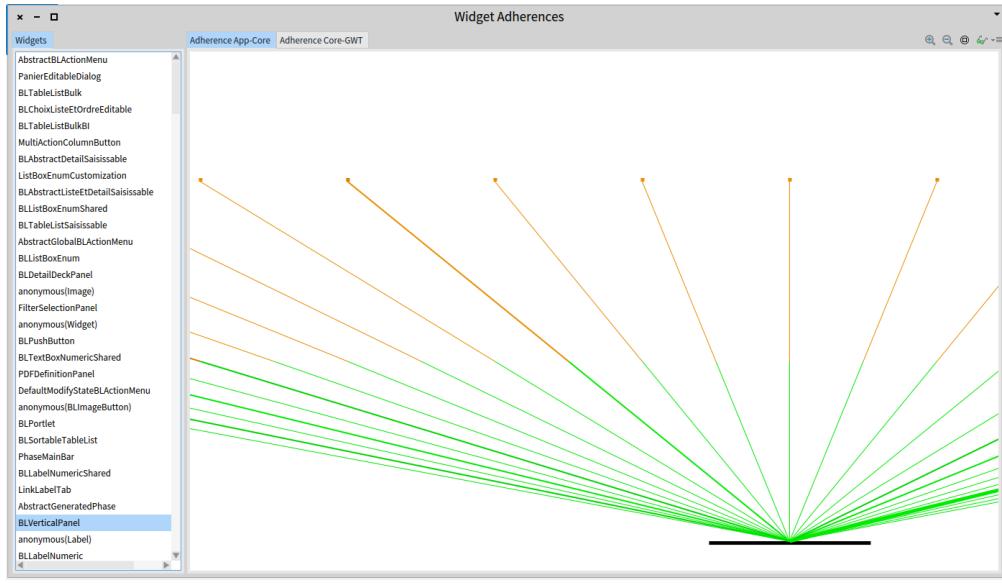


FIGURE 13 – Navigateur références entre les framework

#### 4.3 Travail à faire sur l'adhérence

Toutes les informations récoltées ne m'ont pas permis de définir complètement et précisément ce qu'est l'adhérence. Elles doivent être complétées par une évaluation de la complexité de migrer un élément de BLCore. Ainsi je pourrai mesurer l'impact des références entre classes.

Pour cela, j'aimerais effectuer des migration manuellement. Ces expériences me permettront de juger l'importance des références entre les classes.

## 5 Conclusion

J'ai effectué trois grands travaux pendant mon projet de fin d'études.

Un état de l'art dans lequel j'ai positionné mon travail dans les domaines de la *migration de plateforme* et de la *migration de librairie*. En particulier, j'ai trouvé des articles concernant des travaux proches ou similaires au mien. La mise en correspondance des API des framework que je souhaite migrer semble être la démarche à suivre pour effectuer une migration entre deux langages.

J'ai créé le meta-modèle de l'application *bac-à-sable* et modéliser l'application de Berger-Levrault dans sa quasi-totalité. Il reste à réaliser une étude de scalabilité de ma solution et développer des outils de visualisation facilitant le travail des développeurs.

Finalement, j'ai recherché des solutions pour estimer l'adhérence entre deux entités. Bien que je n'ai pas réussi à utiliser les résultats que j'ai obtenus. Cela m'a permis d'émettre certaines hypothèses sur la manière de faire la migration.

## 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] 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.
- [3] 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.
- [4] 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.
- [5] 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.