GUI Migration using MDE from GWT to Angular 6: An Industrial Case

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

GUI Migration using MDE from GWT to Angular 6: An Industrial Case

Benoît Verhaeghe[1,2], Anne Etien[1], Nicolas Anquetil[1], Abderrahmane Seriai[2],
Laurent Deruelle[2], Stéphane Ducasse[1], Mustapha Derras[2]
[1]Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France
{firstname.lastname}@inria.fr
[2]Berger-Levrault, France
{firstname.lastname}@berger-levrault.com

***Abstract*—During the evolution of an application, it happens that developers must change the programming language. In the context of a collaboration with Berger-Levrault, a major IT company, we are working on the migration of a GWT application to Angular. We focus on the GUI aspect of this migration which, even if both frameworks are web Graphical User Interface (GUI) frameworks, is made difficult because they use different programming languages and different organization schema. Such migration is complicated by the fact that the new application must be able to mimic closely the visual aspect of the old one so that the users of the application are not disrupted. We propose an approach in four steps that uses a meta-model to represent the GUI at a high abstraction level. We evaluated this approach on an application comprising 470 Java (GWT) classes representing 56 pages. We are able to model all the web pages of the application and 93% of the widgets they contain, and we successfully migrated 26 out of 39 pages (66%). We give examples of the migrated pages, both successful and not.**

***Index Terms*—User Interfaces, Industrial case study, Java, Angular, GWT, Migration, Model-Driven Engineering**

I. Introduction

During the evolution of an application, it is sometimes necessary to migrate its implementation to a different programming language and/or Graphical User Interface (GUI) framework [1]. Web GUI frameworks in particular evolve at a fast pace. For example, in 2018 there were two major versions of Angular, three major versions of React.js and four versions of Vue.js. This forces companies to update their software systems regularly to avoid being stuck in old technologies.

Our work takes place in collaboration with Berger-Levrault, a major IT company selling Web applications developed in GWT. However, GWT is no longer being updated with only one major release since 2015. As a consequence, Berger-Levrault decided to migrate its applications to Angular 6.

There are many published papers on supporting GUI migration (*e.g.* [6, 12, 14]). None of them discuss the case of GUI migration of web-based applications.

We present an approach to help developers migrate the GUI of their web-based software systems. This approach includes a GUI meta-model, a strategy to generate the model, and how to create the target GUI. To validate this approach, we developed a tool which migrates Java GWT applications to Angular. Then, we validated our approach on an industrial project that is used to present all the widgets and their usage. It is composed of 470 Java classes and 56 web pages. Our approach imported correctly 93% of the widgets and 100% of the pages. Since not all the existing widgets are re-implemented in Angular, we tried to migrate 39 pages and were successful (same visual appearance) for 26 of them (66%).

The following are the contributions of the paper:

- an approach to migrate the GUI of an application
- a GUI meta-model; and
- a tool that implements our approach to migrate Java GWT application's user interface to Angular.

First, in Section II, we review the literature on GUI meta-modeling. We describe the context of our project in Section III. In Section IV, we describe our migration approach. We present our implementation in Section V. Section VI describes the experiment we made to validate our approach. In Section VII, we present our results. Finally, in Section VIII we discuss the results obtained with our tool and future work.

II. State of the Art

To define a migration strategy, we identified research work related to GUI migration. Some of the proposed approaches do not perform a full migration, but only a part of it.

We identified three techniques to create a representation of the GUI: static, dynamic, or hybrid.

Static. The static strategy consists in analyzing the source code and extracting information from it. It does not execute the code of the analyzed application.

Fleurey *et al.* [3], Lelli *et al.* [7], Silva *et al.* [15] and Staiger [16] used tools that analyze source code of desktop applications. The tools look for widget definition in the source code, then they analyze the methods that invoked or are invoked by the widgets and identify the relationships between widgets and their visual properties.

Sánchez Ramán *et al.* [13] and Garcés *et al.* [4] developed approaches to extract the GUI of Oracle Forms applications. Their approaches consist in the creation of the hierarchy of widgets from their position specified in external files.

Apart from the classical problem of showing all the potential facts rather than only the real one, another limitation appears for example, with a client/server application, when a part of the graphical interface depends on the result of a request to a server.

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

Dynamic. The dynamic strategy consists in the analysis of the graphical interfaces of an application while it is running. It explores the application state by performing all the actions on the user interface of the software system and extracting the widgets and their information.

Memon *et al.* [8], Samir *et al.* [12], Shah and Tilevich [14] and Morgado *et al.* [10] developed tools that implement a dynamic strategy. However, the solutions proposed are only available for desktop rather than Web applications.

Despite the dynamic analysis allows one to explore all the windows of an application, if a request is done to build a GUI, the dynamic analysis does not detect this information which may be essential for a full representation of a GUI.

Hybrid. The hybrid strategy tries to combine the advantages of the static and dynamic analyses.

Gotti and Mbarki [5] used a hybrid strategy to analyze Java applications. The static analysis finds the widgets and attributes of a user interface and how they are structured. Then, the dynamic analysis executes all the possible actions linked to a widget and analyzes if a modification occurs on the interface.

Despite the usage of both static and dynamic analysis, the hybrid strategy does not solve the request problem inherent to client/server applications.

None of the authors considered the migration from web GUI to web GUI. Also, none had the constraint of keeping similar layout except Sánchez Ramán *et al.* [13]; however, they worked on Oracle Forms applications which are very different from a web GUI. As a consequence, their work is not directly applicable to our case study.

III. Context of the Migration Project

The goal of our work is to migrate the user interfaces from a GUI framework to another. This is an industrial project, migrating web applications from GWT to Angular. In Section III-A we list some constraints that we must fulfill. In Section III-B we describe the main differences between GWT applications and Angular ones.

A. Constraints

We identify the following constraints for our case study:

- *From GWT to Angular*. In the context of the collaboration with Berger-Levrault, our migration approach must work with Java GWT as source language and TypeScript Angular as target one.
- *Approach adaptability*. Our approach should be as adaptable as possible to different contexts. For example, it can be used with different source and target languages.
- *Keep visual aspect*. The migration must keep the visual aspect of the target application as close as possible to the original.
- *Code quality conservation*. Our approach should produce code that looks familiar to the developers of the source application. As far as possible, the target code should keep the same structure, identifiers and comments.
- *Automatic*. An automatic solution makes the approach more accessible. It would be easier to use an automatic approach on large system [9].

B. Comparison of GWT and Angular

GWT is a framework that allows developers to write a web application in Java. Angular is a front-end web application platform that allows developers to write a web application with the TypeScript language.

GWT applications and Angular ones have several differences concerning: web page definition, their style and the configuration files. Before explaining these three differences, we note one major similarity: both GWT and Angular applications have a main CSS file to define the general visual aspect of the application.

- **Web Page Definition.** In the GWT framework, only one Java file is necessary to define a web page . It includes the graphical components, their properties and behaviors. In Angular, each web page is considered as a sub-project and represented by a file hierarchy. It contains two files: an HTML file, containing the widgets of the web page and their organizations; and a TypeScript file, containing the code to execute when an action is performed.
- **Visual Aspect.** The visual aspect of a web page includes color or dimension of specific displayed elements. In the case of GWT, the specific visual aspect is defined in the Java file of the web page definition. In Angular, there is an optional distinct CSS file.
- **Configuration Files.** For the configuration files, GWT uses one XML file that defines the binding between a Java file, a web page and the URL of the web page. For Angular, there are two configuration files: *module*, defines the components of the application, *e.g.* web pages, distant services and graphical component; and, *routing*, defines for each web page, its associated URL.

IV. Migration Approach

This section presents the migration approach we designed. First (Section IV-A), we propose a categorization of the GUI source code. In Section IV-B, we describe the migration process we designed. Finally, Section IV-C presents our GUI meta-model.

A. GUI Application Structure

We decided to use a divide-and-conquer strategy to reduce the migration problem in multiple sub-problems. To do so, we define three categories of source code: the visual code; the behavioral code; and the business code.

1) Visual Code: The visual code describes the visual aspect of the GUI. It contains the elements of the interface. It defines the inherent characteristics of the components, such as the ability to be clicked or their color and size. It also describes the position of these components compared to others.

2) Behavioral Code: The behavioral code defines the action/navigation flow that is executed when a user interacts with the GUI. It is also possible that an action is automatically triggered following an outside event. The behavioral code contains control structures (*i.e.* loop and alternative).

3) Business Code: The business code is specific to an application. It includes the rules of the application, the distant server address and the application-specific data.

B. Migration Process

From the state of the art, the constraints and the decomposition of the user interfaces, we designed an approach for the migration.

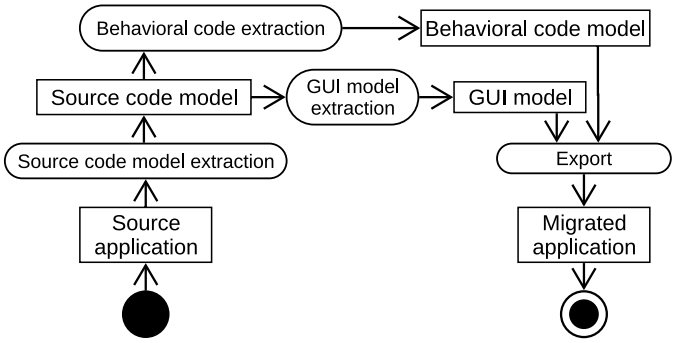


Fig. 1: Our GUI migration process

The process, represented in Figure 1, is divided into the four following steps:

1. *Extraction of the source code model.* In our case study, the source program is written in Java GWT: The extraction produces a FAMIX model [2] of the application using a meta-model capturing Java concepts. We also need to parse the XML configuration file described in Section III-B.
2. *Extraction of the GUI model.* We analyze the source code model to detect the *Visual code* elements describing the GUI and we build a GUI model from these elements. The GUI meta-model is described Section IV-C.
3. *Extraction of the behavioral code.* Once the GUI model is generated, we use it to identify the parts of the source code model corresponding to *Behavioral code*.
4. *Export.* We re-create the GUI and the *Behavioral code* in the target language. This step exports the user interface files and the configuration files of the application.

Note that currently we do not treat the *Business code* of the application. This will be the focus of future work.

C. GUI Meta-Model

To represent the user interfaces of desktop or web-based applications, we designed the meta-model presented in Figure 2. In the rest of this section, we present the entities of the meta-model.

The **Page** represents the main container of a graphical interface. It is either a *window* of a desktop application or a web page. A **Page** contains several **Widgets**.

A **Widget** is a component of the interface. It has multiple **Attributes** and **Actions**. We use the Composite design pattern with the entities **Leaf** and **Container**. This representation of the DOM is heavily used in the literature.

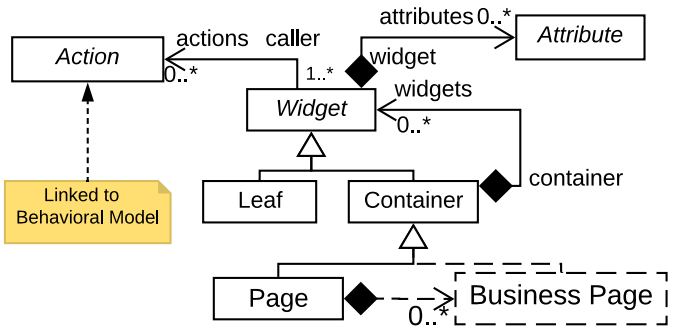


Fig. 2: GUI meta-model

An **Attribute** represents the information of a widget and can change its visual aspect. Some common attributes are the height and the width to precisely define the size of a widget and the *text* attribute that contains the text of a button.

An **Action** is specific to widgets. It corresponds to the Behavioral code of the GUI. It represents an interaction between the user and the graphical interface.

V. Implementation

To test our approach, we implemented a migration tool. It is implemented in Pharo and the meta-model is represented using the Moose platform.

Note that we did not implement the *Behavioral code* migration.

A. Case Study

Applications at Berger-Levrault (our industrial partner) are based on the BLCore framework. This framework consists in 763 classes in 169 packages. It is developed by the company on top of GWT and defines the widgets that developers should use. It also encourages some coding conventions. The BLCore framework is used for the development of 8 large applications.

For the Berger-Levrault applications, we add a new entity (**Business Page**) to the GUI meta-model presented Section IV-C) to fit the company's specific organization.

B. Import

In part because of the complexity of setting up a tool to run automatically and capture all screens of such large web applications, we rely on static analysis to create our model. The results so far seem to indicate that it will be sufficient.

As presented Section IV-B, the creation of the GUI model is divided in two steps: the source code model extraction and the GUI model extraction. For the source code meta-model, we use the Java meta-model of Moose [2, 11] which comes with a Java extractor.

For the second step of the extraction, our tool creates the GUI model from the source code model and an analysis of the XML configuration file. The entities we want to extract are, first, the **Pages**. We parse the XML configuration file in which is defined the information about the pages (see Section III-B).

Then, the tool looks for the **Business pages** and the **Widgets**. A **Business page** is defined by a Java GWT class which implements the IPageMetier interface. A **Widget** is defined by a class which is a subclass of the GWT class Widget. Then, for both **Business pages** and **Widgets**, the tool looks where their constructors are called and from that, it deduces and creates a link between the **Widget** and its container.

Finally, to detect attributes which belong to a widget, the tool detects in which Java variable the widget has been assigned. Then, it searches the methods invoked on this variable. If it invokes a method *"setX"*, it creates an attribute. This heuristic was found in the literature [12, 15].

C. Export

Once the GUI model is generated, it is possible to export the application. To generate the code of the target application, the tool visits the GUI model. The visitor creates the folders of the target application and the configuration files. Then, it visits the pages. For each **Page**, the visitor creates an Angular sub-project in the form of a directory containing several configuration files and a default blank web page. Then, for each business page of the current visited **Page**, the visitor generates one HTML file and one TypeScript file.

VI. Validation

We experimented our strategy on Berger-Levrault's *kitchensink* application. This software system, dedicated to developers, presents all the components available for building a user interface. This application is smaller than a production one but works exactly the same way. It contains 470 Java classes and represents 56 web pages. The *kitchensink* application, as the other applications of Berger-Levrault, does not have tests. As a consequence, we can not validate the migration with tests of the GWT application.

The validation is done in three steps: First, we check the constraints defined in Section III-A; Second, we validate that all GUI entities of interest are extracted and correctly extracted; Third, we validate that we can re-export these entities in Angular and that the result is correct.

For the second validation, we manually identify and count the entities in the *kitchensink* application and compare the results of the tool to this count. Our analysis focuses on the migration of three entities: **Pages**, **Business Pages**, **Widgets**.

- **Pages.** From the XML configuration file of the application we manually count 56 pages. This configuration file also provides the name of each page.
- **Business Pages.** As explained before, the business pages correspond to a concept specific to Berger-Levrault. They are defined in the BLCore framework as a Java class which implements the interface IPageMetier. Thanks to this heuristic, we manually count 76 **Business Page** instances in the original application.
- **Widgets.** In the literature survey, we did not find an automatic way to evaluate the detection of widgets. Checking all widgets in the application would be long and error-prone as there are thousands of them. As a fallback solution, we take a sample of the pages of the *kitchensink* application and count the widget in the DOM of these pages. We consider a sample of 6 **Pages** which represents a bit more than 10% of the **Pages** of the application. These **Pages** are of different sizes and contain different kinds of widgets. In total, we found 238 **Widgets** in these 6 **Pages**. To get a more exact idea of the representativeness of our sample, we also count the number of **Widget** creation in the code. There are 2,081 such creation. This may not represent the exact number of widgets in the entire application, but it is a good estimate. We also check that the **Widgets** are correctly placed in the DOM of the interface (*i.e.*, they belong to the right **Container** in the GUI model).

In our results we consider only the recall of the tool because the precision is always 100% (there are no false positive). This is a sign that the BLCore framework provides clear (if not complete) heuristics to identify the entities.

For the third validation, we check that the entities are exported correctly: In the Angular application, each **Page** corresponds to a sub-project and is represented by a folder. The name of the folder must correspond to the name of the **Page**. The **Business pages** are represented by a sub-folder inside the **Page** project. The names must also match at this level.

We also check visually that the exported **Page** "looks like" the original one. This is a subjective evaluation, and we are looking for options to automate it in the future.

VII. Results

This section presents the results of the migration validation on Berger-Levrault's *kitchensink* application. In Section VII-A, we confront the exported result with the constraints defined in Section III-A. Section VII-B and Section VII-C summarize respectively the extraction and the export results.

A. Satisfaction of Constraints

We set the following constraints in Section III-A: *From GWT to Angular*, *Approach adaptability*, *Code quality conservation*, *Keep visual aspect*, and *Automatic*.

Our tool can use Java code as input and generate Angular code. The exported code is compilable and executable. The target application can be displayed. We can thus confirm that our tool fulfill the GWT to Angular constraint.

Our tool is applicable on other source target technologies. Our heuristics have been designed to be easy to adapt, A user of our tool can thus add a new kind of widget for the import or the export phases. Those possibilities satisfied the adaptability constraint.

The *Code quality conservation* and *Keep visual aspect* constraints are discussed in Section VII-C.

Finally, the results described here were obtained automatically from application of our tool to the subject application. This validates the last constraint.

B. Extraction Results

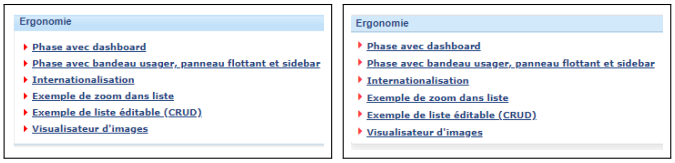
The tool extracted 56 **Pages** from the original application. This corresponds to the number of pages defined in the configuration file of the *kitchensink* application.

The tool extracted 76 **Business pages**. This value corresponds exactly to the number of business pages in the original application. Moreover, the tool correctly assigned each **Business page** to its proper **Page**.

We got 100% of the **Widgets** on the evaluated sample were correctly detected. However, 27 out of the 238 **Widgets** of our sample (11%) were not correctly assigned to their parent container. This problem comes from one single **Business page** (containing 75 **Widgets** in total).

C. Export Results

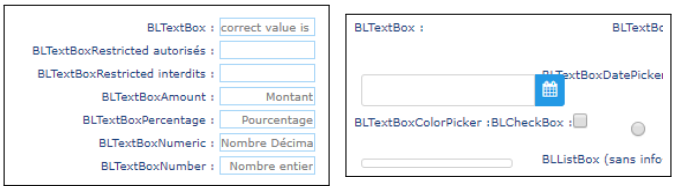
All exported pages conserve their original name.



(a) GWT original (b) Angular migration

Fig. 3: Visual comparison of a **Business page** migration

Figure 3 presents the visual differences between the original (GWT) version, left hand, and the migrated (Angular 6) one, right-hand. We can see that there are only minimal differences. In the exported version, the color of the header is a bit clearer, and the lines are a little more distant.



(a) GWT original (b) Angular migration

Fig. 4: Visual comparison of a **Business page** migration: All the **Widgets** are migrated but with a wrong layout.

Figure 4 presents the visual differences for the **Page** *Input box*. Again on the left-hand side there is the original **Page** and on the right-hand side is the same **Page** after the migration. Because the two images are large, we trimmed them to display this area of interest. Even though the two images look completely different, all the widgets are present in the migrated version. The visual differences are due to a problem in the layout management. The visual constraint is thus partially satisfied.

VIII. Conclusion and Future Works

In this paper, we exposed a preliminary work on the problem of visual preservation and respect of the target architecture during the GUI migration of an application. We proposed an approach based on a GUI meta-model and a migration process in four steps. We implemented this process in a tool to perform the migration from GWT applications to Angular 6. Then, we validated our tool with an experiment on a *kitchensink* application.

We were able to extract correctly all pages of the application and 89% of the widgets. The migration results are visualizing equivalent as long as complex widgets (*e.g.* GridLayout) are not used. Dealing with these layouts is our next challenge.

References

[1] J. Brant, D. Roberts, B. Plendl, and J. Prince. Extreme maintenance: Transforming Delphi into C#. In *ICSM'10*, 2010.

[2] S. Ducasse, N. Anquetil, U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[3] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jezéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In *Model Driven Engineering Languages and Systems*, volume 4735, pages 482–497, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Viera, F. Melo, and J. M. Soto. White-box modernization of legacy applications: The oracle forms case study. *Computer Standards & Interfaces*, pages 110–122, October 2017.

[5] Z. Gotti and S. 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.

[6] M. E. Joorabchi and A. Mesbah. Reverse engineering iOS mobile applications. In *19th Working Conference on Reverse Engineering (WCRE 2012)*, pages 177–186. IEEE, 2012.

[7] V. Lelli, A. Blouin, B. Baudry, F. Coulon, and O. 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.

[8] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 260–269. IEEE, 2003.

[9] 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.

[10] I. C. Morgado, A. Paiva, and J. P. Faria. Reverse engineering of graphical user interfaces. In *ICSEA 2011 : The Sixth International Conference on Software Engineering Advances*, 2011.

[11] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press.

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

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

[14] E. Shah and E. Tilevich. Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, \& VMIL'11*, pages 255–260. ACM, 2011.

[15] J. C. Silva, C. C. Silva, R. D. Goncalo, J. Saraiva, and J. C. Campos. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 181–186. ACM Press, 2010.

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